



EFFICIENT BUILDING BLOCKS FOR HARDWARE NEURAL NETWORKS

ADEDAMOLA WURAOLA

DEPARTMENT OF ELECTRICAL, COMPUTER AND SOFTWARE ENGINEERING
THE UNIVERSITY OF AUCKLAND

A DISSERTATION SUBMITTED TO THE DEPARTMENT OF ELECTRICAL,
COMPUTER AND SOFTWARE ENGINEERING IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

NOVEMBER 2020

Contents

Abstract	vi
Acknowledgement	vii
Acronyms	viii
Publications	ix
Patents	x
Lists of Figures	xv
Lists of Tables	xxi
1 Introduction	1
1.1 Neural Network Building Blocks	1
1.1.1 Activation Function	2
1.1.2 Multiplication	3
1.1.3 Data Representation	4
1.2 Motivation	4
1.3 The Proposed Square Law Solutions	5
1.4 Research Contributions	5
1.5 Thesis Outline	8
2 Square Nonlinearity: A Computationally Efficient Activation Function	10
2.1 Introduction	10
2.2 Concepts	12
2.3 Analysis	19

2.3.1	Computational Footprint on Intel CPUs	21
2.3.2	Computational Footprint on the ARM M3 Processor	23
2.3.3	Decision Boundary and Activation Over Time	24
2.4	Performance Comparison	25
2.4.1	SQLN for Shallow Supervised Learning	25
2.4.2	Log_SQLN for Binary Logistic Regression	32
2.4.3	SQLN and Log_SQLN for Recurrent Neural Network	33
2.5	Conclusion	34
3	Efficient Hardware Calculator for SQLN Function	36
3.1	Introduction	36
3.2	Concepts	38
3.2.1	Counter-based SQLN	39
3.2.2	Multiplier-based SQLN	42
3.3	Hardware Implementation	43
3.3.1	Multi-clock/Counter Solution	43
3.3.2	Single-clock Solution	44
3.3.3	Multiplier Solution	45
3.4	Resource Utilisation	45
3.4.1	Counter-based Implementation	46
3.4.2	Multiplier-based Implementation	47
3.4.3	Lookup Table-based Implementation	47
3.4.4	Single-clock Solution Implementation	48
3.4.5	Discussion	48
3.5	Inference Performance Accuracy	50
3.5.1	Performance Accuracy on UCI datasets	50
3.5.2	Performance Accuracy on Experimental Rotational Dataset	53
3.5.3	Performance Accuracy on MNIST Dataset	53
3.6	Conclusion	54
4	Asymmetric Square-based Activation Functions for Deep Learning	56
4.1	Introduction	56
4.2	Novel Activation Functions	58

4.2.1	Smooth Square-based Asymmetric Activation Functions	59
4.2.2	Square-based Output Layer Activation Function	62
4.3	Analysis	62
4.3.1	Computational Footprint on Intel CPUs	62
4.3.2	Computational Footprint on the ARM M3 Processor	63
4.4	Experimental Results	64
4.4.1	Activation Over Time and Inference Time	65
4.4.2	Experiments using SQLU	66
4.4.3	Experiments using Sqish for Very Deep Networks	69
4.4.4	Experiments using SqREU	70
4.4.5	Experiments using SQ_Softplus for Restricted Boltzmann Machine	70
4.4.6	Experiments using SQMAX	71
4.5	Conclusion	72
5	Resource Efficient Asymmetric Activation Functions Generator	73
5.1	Introduction	73
5.2	Concepts	76
5.3	Analysis	79
5.3.1	Computational Footprint on Embedded NIOS II Processor	79
5.3.2	Resource Footprint: Arithmetic Logic Unit Implementation	80
5.4	Hardware Implementation	81
5.5	Results and Discussion	82
5.5.1	Resource Utilisation of Asymmetric Function Generator	83
5.5.2	Resource Utilisation of SQ-GEN	84
5.6	Conclusion	85
6	Computationally Efficient Radial Basis Function	86
6.1	Introduction	87
6.2	RBF Networks and RBF Kernels	88
6.3	Nonlinear Support Vector Machine and RBF Kernels	89
6.4	Novel Square Nonlinear Radial Basis Function (SQ-RBF)	90
6.5	Mercer's Theorem Proof for SQ-RBF Kernel	91
6.6	Hardware Implementation of Square-based Gaussian RBF	92

6.7	Software Experimental Results	93
6.7.1	SQ-RBF on RBFNN Problems	93
6.7.2	SQ-RBF Kernel on SVM Classification Problems	98
6.8	Hardware Experimental Results and Discussion	98
6.9	Conclusion	100
7	Resource Efficient Implementation of Machine Learning Models	101
7.1	Introduction	102
7.1.1	Recurrent Neural Networks	102
7.1.2	Feed Forward Neural Networks	104
7.2	Concepts	106
7.2.1	Gated Activation	106
7.2.2	Quantised Scaling Unit (QSU)	112
7.3	Hardware Implementation	114
7.3.1	Hardware Implementation of Gated Activation	115
7.3.2	Hardware Implementation of QSU	116
7.4	Experimental Results	116
7.4.1	LSTM Software Experimental Results	117
7.4.2	GRU Software Experimental Results	119
7.4.3	FNN Software Experimental Results	119
7.4.4	Resource Utilisation of QSU and Standard Multiplier	120
7.4.5	LSTM Hardware Experimental Results and Discussion	120
7.4.6	GRU Hardware Experimental Results and Discussion	123
7.4.7	FNN Hardware Experimental Results and Discussion	123
7.5	Conclusion	124
8	Evaluation of Learnable Asymmetric Activation Functions for Deep Learning	125
8.1	Introduction	126
8.2	Related Work	127
8.2.1	Single Learnable Functions	127
8.2.2	Multiple Learnable Functions	127
8.2.3	Others	128
8.3	Parametric Square-based Asymmetric Activation Functions	129

8.4	Multiple Square Units	130
8.5	Experiments	131
8.5.1	CIFAR-10	133
8.5.2	CIFAR-100	135
8.5.3	SVHN	135
8.6	Conclusion	138
9	Conclusions and Future Work	139
9.1	Conclusions	139
9.2	Future Work	141
	Bibliography	142

Abstract

The field of artificial intelligence and its paradigms continue to achieve state-of-the-art accuracy in various tasks. The future of artificial intelligence-based solutions will be populated with smart devices that require low computational power and inexpensive hardware platforms. Typically, the machine learning algorithms are complex, iterative, time-consuming, and hence are usually executed on general-purpose or high-performance computers. The computational engine for inferencing is significantly less complex than machine learning algorithms. While standard computers can incorporate sophisticated floating-point units (FPU), this is not the case with embedded processors that may host a simplified FPU. On dedicated inference engines, FPUs may not be possible due to cost (space, power, and speed). To solve this problem, we focus on efficient algorithms for various building blocks of neural networks. Our approach starts by introducing the *square law* that significantly reduces the computation requirements of machine learning models by eliminating the need for mathematical operators such as exponent, floating-point division, square root, and logarithm. The square law algorithm can reduce computation time on CPU by 1.3x to 4.3x and on ARM processors by 4x to 169x without hurting the prediction accuracy. We also discovered that square law can be applied across a wide variety of machine learning building blocks. We propose distinct technologies to make a complete neural network on a chip. The square law-based solutions use standard digital building blocks and can be implemented on ASICs or FPGAs. On ASIC platform, our algorithm records area efficiency (throughput per gate) of 1.79x to 3.75x over baselines without hurting the prediction accuracy.

Acknowledgement

The biggest gratitude to my supervisor for not only being a supervisor but a mentor, thank you for believing in me, enduring all my naivety, and making me a better researcher. Your time and guidance have been indispensable to my graduate career, and your imagination is truly inspiring. Thank you for exposing me to everything - grant writing, supervision, and patents.

I would like to acknowledge the financial support received from the Faculty of Engineering, the University of Auckland in the form of the Faculty of Engineering Doctoral Scholarship Award.

I would like to express my deepest gratitude and special thanks to Harry She and Emily Melhuish at Halter for giving me the opportunity to explore during my internship within the company. For me, it was a unique experience to be at Halter and to do the research on data engineering.

Thank you to the numerous anonymous reviewers of my articles that provided valuable feedback and broke me until there was nothing left to break.

Thank you to the following groups of people and organisations who helped make my doctoral education more rounded by giving me opportunities outside the scope of my main degree, including those at the Center for Innovation and Entrepreneurship, UniServices, Nokia, The University of Auckland Human Participants Ethics Committee (UAHPEC), and the Postgraduate Students Association.

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. Thank you Lola, Lolade, and Ife for being the best siblings anyone can ask for. Thank you to Busayo and Lamide.

Words are not enough to say how grateful I am to Matt, thank you for always listening to my rants and supporting me throughout this journey, I only can wish to be half as caring and loving as you.

To the friends I made and lost during these last few years, I say thank you, you each played an important role physically and emotionally.

Finally, I thank my great God and Saviour, Jesus Christ, without whom none of this would have been possible.

Acronyms

ALM	Adaptive Logic Module
ALU	Arithmetic Logic Unit
ANN	Artificial Neural Network
ASIC	Application Specific Integrated Circuit
CNN	Convolution Neural Network
CORDIC	Coordinate Rotation Digital Computer
CPU	Central Processing Unit
DNN	Deep Neural Network
DSP	Digital Signal Processor
FF	Flip Flops
FPGA	Field Programmable Gate Array
GEMM	General Matrix Multiply
GPU	Graphical Processing Unit
LSTM	Long Short Term Memory
LUT	LookUp Table
MLP	Multilayer Perceptron
RBF	Radial Basis Function
RNN	Recurrent Neural Network
SLFNN	Single Layer Feed-Forward Neural Network
SQLU	Square Linear Units
SQNL	Square Nonlinearity
SVM	Support Vector Machine
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

Publications

1. Wuraola, A., Patel, N., & Nguang, S.K., Efficient Activation Functions for Embedded Inference Engines (2021), *Elsevier Neurocomputing*, vol. 442, pp 73-88.
2. Wuraola, A., & Patel, N., SQLN: A new computationally efficient activation function (2018), Proceedings of the *IEEE International Joint Conference on Neural Networks (IJCNN)*.
3. Wuraola, A., & Patel, N., Computationally Efficient Radial Basis Function (2018), In *International Conference on Neural Information Processing*. Springer, Cham.
4. Wuraola, A., & Patel, N., Stochasticity-Assisted Training in Artificial Neural Network (2018), In *International Conference on Neural Information Processing*. Springer, Cham.
5. Wuraola, A., Patel, N., & Nguang, S.K., Resource Efficient Activation Functions for Neural Network Accelerator, *Elsevier Neural Networks* (Under Review).

Patent

Patel, N., Wuraola, A. inventors; Nokia, Inc, assignee. Activation Function Implementation. United Kingdom Patent Application No. GB2007934.9, 2020, January 3.

List of Figures

1.1	Different Variant of RELU with Increasing Computational Complexity.(LReLU [1], APLU [2],ELU [3], GELU [4], SELU [5], dSiLU [6], ELISH [7], SRS [8]). Square Law is the algorithm proposed in this thesis for simple symmetric and asymmetric activation functions.	3
1.2	Thesis Outline	6
2.1	Potential ASIC Implementation Block Schematic of Proposed Nonlinear Activation Function	13
2.2	Pictorial Description of Transfer Characteristics of the Proposed Method	14
2.3	Simulated Activation Functions Showing Approximate TanSig and LogSig Behaviour. The traces with the jitter are the outputs of the simulated activation function while the solid trace is a plot of Equation 2.1 with the fitted parameters.	14
2.4	Top - Pictorial Analysis of Nonlinear Action. Bottom - Comparison of fitted model and closed form model	16
2.5	Forward mappings - TanSig ($f_t(x)$), ElliotSig ($f_e(x)$), and Proposed SQLN Function ($f_s(x)$)	20
2.6	Derivative mappings - TanSig ($f'_t(x)$), ElliotSig ($f'_e(x)$), and Proposed SQLN Function ($f'_s(x)$)	20
2.7	Forward mappings - Sigmoid ($f_s(x)$) and Proposed Log_SQLN Function ($f_{LogSQ}(x)$) . . .	21
2.8	Derivative mappings - Sigmoid ($f'_s(x)$) and Proposed Log_SQLN Function ($f'_{LogSQ}(x)$) . .	21
2.9	a) Toy dataset: TanSig decision boundary b) Toy dataset: SQLN decision boundary c) MNIST dataset: TanSig hidden layer activations during training d) MNIST dataset: SQLN hidden layer activations during training e) MNIST dataset: TanSig histogram distribution of activations f) MNIST dataset: SQLN histogram distribution of activations.	24
2.10	Results of Experiment 1 For Traditional Backpropagation - BC:Breast Cancer, IS: Ionsphere	27

2.11 Results of Experiment 1 For Levenberg Macquardt Backpropagation - BC:Breast Cancer, IS: Ionosphere	27
2.12 Results of Experiment 1 For Resilient Backpropagation - BC:Breast Cancer, IS: Ionosphere	28
2.13 Results of Experiment 2 For Traditional Backpropagation - BC:Breast Cancer, IS: Ionosphere	28
2.14 Results of Experiment 2 For Levenberg Macquardt Backpropagation - BC:Breast Cancer, IS: Ionosphere	29
2.15 Results of Experiment 2 For Resilient Backpropagation - BC:Breast Cancer, IS: Ionosphere	29
3.1 SQLN - A Symmetric Activation Function. a) The conceptual impact of Equation 3.1 is shown for symmetric activation function realization. $U(k)$ is plotted on the x-axis and n , the netsum, along the y-axis. The area enclosed in green represents the output mapping $f(n_1)$. The inset shows the form of the mapping for $-M \leq n < M$ b) When $n = \pm i \times \Delta$ i.e. midway between any two contiguous $U(k)$, the $f(n)$ mapping is exact. c) When $n = \pm(\frac{\Delta}{2} + i \times \Delta)$ i.e. exactly equal to any element of $U(k)$, the deviation from the ideal is a maximum.	40
3.2 Profile of deviation from ideal for $R = 8$ when N is reduced to 4 and 8. a) Plots the deviation for the positive input range for different values of C . b) Plots the frequency distribution of the deviation as a probability	42
3.3 Schematic of SQLN Activation Function using Multi-clock Methodology	43
3.4 Schematic of SQLN Activation Function using Single-clock Methodology. (RSH is Right Shift Operation)	44
3.5 Hardware Implementation of SQLN using the custom multiplier (special square operator). The $\div 4, \times 4$ operations are right and left shift operations respectively, hence, have no impact on the computational or resource footprint.	45
3.6 Fisher's Iris Classification Dataset - Experimental Result (Modelsim code is synthesisable)	51
3.7 Breast Cancer Classification Dataset - Experimental Result (Modelsim code is synthesisable)	52
3.8 Simple Fit Regression Dataset - Experimental Result (Modelsim code is synthesisable) .	52
3.9 Sine Function Regression Dataset - Experimental Result (Modelsim code is synthesisable)	53
3.10 Cosine Sine Function Regression Dataset - Experimental Result (Modelsim code is synthesisable)	54

4.1	Activation Functions: All the proposed square-based against their corresponding complex-based functions	59
4.2	Top: First hidden layer activations during training with ELU Function, Bottom: First hidden layer activations during training with SQLU Function.	65
4.3	Top: Last hidden layer activations during training with ELU Function, Bottom: Last hidden layer activations during training with SQLU Function.	66
4.4	SQLU networks evaluated on MNIST. a) Training set cross entropy loss for different activation functions. b) Validation set cross entropy loss for different activation functions.(Best viewed in colour)	67
4.5	SQLU and ELU networks evaluated on CIFAR10 with Data Augmentation to show that SQLU is not inferior to ELU. a) Training set cross entropy loss for SQLU and ELU functions. b) Validation set cross entropy loss for SQLU and ELU functions. (Best viewed in colour)	67
4.6	Autoencoder training on MNIST: a) Training reconstruction error using different activation functions. b) Test reconstruction error using different activation functions.(Best viewed in colour)	68
4.7	Comparison of Conventional softplus and SQ_softplus: Left: Training loss, Right: Test error.	71
5.1	An Asymmetric SQLN Activation Function	76
5.2	A Parameterised Asymmetric SQLN Activation Function	78
5.3	Hardware Implementation of SQLU using the custom multiplier (special square operator). The $\div 4, \times 4$ operations are right shift operations and hence, have no impact on the computational or resource footprint. Slight modification will result in other asymmetric functions.	80
5.4	The implementation of a custom square operator ($f_s(n) = n \times - n $) using a multiplier.	81
5.5	SQLN family implementation schematics: a) SQLN using the custom multiplier $f_s(x) = x \times - x $. b) Log_SQLN. c) SQLU d) SQ_Softplus. Note: The $\div 2$ and $\div 4$ operations are right shift operations and hence, have no impact on the computational or resource footprint.	81
5.6	Schematics Asymmetric Activation Function Generator using multi-clock methodology.	82
6.1	The SQ-RBF and Gaussian RBF Kernels	91

6.2	Schematic of SQ-RBF Activation Function using Multi-clock Solution	93
7.1	Difference between conventional and proposed LSTM cell with the elimination of element-wise multiplication. (QSU: Quantised Scaling Unit, L_S: Log_SQNL, Gated Act: Gated Activation).	106
7.2	Difference between conventional and proposed GRU cell with the elimination of element-wise multiplication. (QSU: Quantised Scaling Unit, L_S: Log_SQNL, Gated Act: Gated Activation).	106
7.3	Gated Activation: loss evaluated at $n = 0$	107
7.4	Gated Activation: loss evaluated at $0 \leq n \leq U_{MAX} - C$	108
7.5	Gated Activation: loss evaluated at $U_{MAX} - C \leq n \leq U_{MAX} + C$	108
7.6	Gated Activation: loss evaluated at $U_{MAX} + C \leq n \leq 2U_{MAX}$	109
7.7	a) Gated Activation, $R = 8, N = 8$: It shows that at a constant input, say netsum = 40 and $C = 64$, the output is $f(40, 64) = 33.75$. Thus, $f(40, 40) = 24 \approx \frac{40}{64} \times f(40, 64) = 21.09$. b) This shows the error varies with both n and C . The maximum error is always at $n = 2^{R-2}$, and $C = 2^{R-3}$, With $R = 8$, the maximum error is binary value of 4.	111
7.8	Profile of deviation from ideal $f(n, C)$ a) Plots the deviation for the positive input range for different values of C . b) Plots the frequency distribution of the deviation as a probability.	113
7.9	Gated Activation Implementation.	115
7.10	Schematic of Quantised Scaling Unit Hardware Implementation.	116
8.1	A visualisation of Multiple Square Units, varying α . The closer α is to 0, the more it is like SQ_Softplus, α greater than 1, shows SqREU and $\alpha = 1.0$ is SQLU.	131
8.2	CIFAR10: Convergence curves for training sets of fixed and learnable activation functions on different WRN architectures. a) WRN-40-1 b) WRN-40-4 c) WRN-16-4 d) WRN-16-8. The parametric SQLU (PSQLU-1 and PSQLU-2) converges fastest than the fixed SQLU activation function for all architecture depths and widths.	134
8.3	CIFAR100: Convergence curves for training sets of fixed and learnable activation functions on different WRN architectures. a) WRN-40-1 b) WRN-40-4 c) WRN-16-4 d) WRN-16-8. The parametric SQLU (PSQLU-1 and PSQLU-2) converges fastest than the fixed SQLU activation function for all architecture depths and widths.	136

8.4	SVHN: Convergence curves for training sets of fixed and learnable activation functions on different WRN architectures. a) WRN-40-1 b) WRN-40-4 c) WRN-16-4 d) WRN-16-8. The parametric SQLU (PSQLU-1 and PSQLU-2) converges fastest than the fixed SQLU activation function for all architecture depths and widths.	138
-----	---	-----

List of Tables

2.1	CPU Performance of Commonly used Mathematics Function and the Activation Functions. EP: Enhanced Performance.	22
2.2	Speedups: CPU Performance of Symmetric Activation Functions. EP: Enhanced Performance.	22
2.3	Computational time of Tanh, ElliotSig, SQLN, and ISRU functions and their derivatives using ARM M3 processor. This is an average of 1000 calculations.	23
2.4	Speedups: Computational time of Tanh, SQLN, ElliotSig, ISRU function and their derivative using ARM M3 processor. This is an average of 1000 calculations.	24
2.5	Sensitivity and Specificity on Classification Dataset	31
2.6	Accuracy and G-mean on Classification Dataset	31
2.7	Convergence Speed On MNIST Dataset	32
2.8	Performance Accuracy of Sigmoid/LogSig Vs SQLN based Logistic Regression. A comparable result between Sigmoid and Log_SQLN is achieved.	33
2.9	Performance Accuracy using square-based LSTM network. Changing the activation function to SQLN yielded about 3% for the Kaggle Model and 1% for the IMDB dataset improvement in performance accuracy. Use of Hardtanh and Hardsig performs worse than TanSig and sigmoid functions. (SQLN and log_SQLN to replace TanSig and sigmoid respectively)(Results are average of 5 runs, and an estimated mean of accuracy with the confidence of 95% is recorded)	34
2.10	Performance Accuracy using square-based GRU network. (Results are average of 5 runs, and an estimated mean of accuracy with the confidence of 95% is recorded)	34
3.1	Indicative Gate Usage	46
3.2	Estimated Counter Based SQLN Gate Usage (GU) with Various N	46

3.3	Resource Utilisation Summary of SQLN Implementation using direct (D_SQLN) method, LUT (L_SQLN) method and counter-based (C_SQLN)	48
3.4	Resource utilisation of single-clock solution, LUT and multiplier-based solution.	48
3.5	Ratio of Gates Usage for Normalised Throughput	49
4.1	Speedups: CPU Performance of asymmetric activation functions. Run on Intel Xeon with Vector Function Data.	63
4.2	Computational time of ELU, SQLU, Softplus (Softp) and SQ_Softplus (SQ_Softp) , SqREU, REU, Swish and Sqish functions and their derivatives using ARM M3 processor. This is an average of 1000 calculations.	64
4.3	Speed ups: Computational time of asymmetric functions and their derivatives using ARM M3 processor. This is an average of 1000 calculations.	64
4.4	MNIST preliminary analysis showing: training time, inference time in seconds, and percentage performance accuracy. Here SQLU consistently performs better than ELU.(Average of five runs)	65
4.5	Performance Accuracy on CIFAR - 10 using ReLU, ELU and SQLU activation functions on already defined architecture. We used the ResNet20 (20 layer ResNet) Version 1 [9]. The results show that SQLU is not inferior to ELU and hence, can replace ELU whenever the computational time and resource-efficient Inference block is essential. * ELU diverges. (Results are average of five runs, and an estimated mean of accuracy with the confidence of 95% is recorded).	68
4.6	Performance Accuracy on CIFAR - 100 using ReLU, ELU and SQLU activation functions on already defined architecture. We used the ResNet20 (20 layer ResNet) Version 2 [10]. * ELU diverges. (Results are average of five runs, and an estimated mean of accuracy with the confidence of 95% is recorded).	68
4.7	Performance Accuracy on SVHN using ReLU, ELU and SQLU activation functions on already defined architecture. We used the Convnet ([11]). The results show that SQLU is not inferior to ELU and hence, can replace ELU whenever the computational time and resource-efficient Inference block is essential. (Results are average of five runs, and an estimated mean of accuracy with the confidence of 95% is recorded).	69

4.8	Performance Accuracy on Tiny ImageNet using ReLU, ELU and SQLU activation functions on already defined architecture. We used the DenseNet and model parameters defined in [12]. We use the ResNet50 (50 layer ResNet) [10] and model parameters defined in [13].	69
4.9	Sqish: Performance Accuracy on CIFAR - 10 using activation functions on already defined architecture. The results show that Sqish is not inferior to swish and hence, can replace swish whenever the computational time and resource-efficient Inference block is essential. (Results are average of five runs, and an estimated mean of accuracy with the confidence of 95% is recorded).	70
4.10	Sqish: Performance Accuracy on CIFAR - 100 using activation functions on already defined architecture. The results show that Sqish is not inferior to swish and hence can replace swish whenever the computational time and resource-efficient Inference block is essential. (Results are average of 5 runs, and an estimated mean of accuracy with the confidence of 95% is recorded)	70
4.11	SqREU: Performance Accuracy on Fashion MNIST (FMNIST), CIFAR-10 and CIFAR-100 using activation functions on already defined architecture. The results show that SqREU is not inferior to REU and hence, can replace REU whenever the computational time and resource-efficient Inference block is essential. (Results are average of five runs, and an estimated mean of accuracy with the confidence of 95% is recorded).	70
4.12	Performance Accuracy on SVHN CIFAR-10 and CIFAR-100 using softmax and SQMAX as the output layer activation function. The hidden layer function is left as ReLU. We changed the flatten layer to the Global Averaging Pooling for the VGG-19 architecture. The results show that SQMAX is not inferior to Softmax and hence, can replace softmax whenever the computational time and resource-efficient inference block is essential.	71
4.13	Computational time of exponential and square operators using ARM M3 processor. . . .	72
5.1	The optimised values of different activation functions for hardware implementation ($U_{MAX} = 2^{R-2}$ and $M = 2^{R-1}$)	82

5.2	Resource utilisation of asymmetric activation function implementation using a custom Booths Radix-4 multiplier (mult), multi-clock (counter, $N = 8$), and LUT solution. As displayed the counter based method for the asymmetric activation function consistently outperform the multiplier solution on both ASIC and FPGA platforms. At a lower resolution, the LUT performs slightly better than the counter solution on FPGA but worse on ASIC when compared with the counter solution. The LUT on the other hand is only better for lower resolution and can not accommodate applications where higher resolution is required. The counter solution scales well across different resolutions.	83
5.3	The ratio of gates usage for normalised throughput. The counter-based solution performs better than the multiplier and LUT solutions both at a lower resolution and higher resolution. At higher resolution, the counter-based solution achieves extremely high throughput when compared to the LUT.	84
5.4	Resource Utilisation of SQLN,SQLU,SQ_Softplus Combination Implementation using custom implementation	84
6.1	Common and Approximate RBF Kernels	89
6.2	Experiment 1: Performance Comparison on SinE Function. The best result is shown in bold.	94
6.3	Experiment 2: Performance Comparison on SinE Function. The best result is shown in bold.	94
6.4	Experiment 1: Performance Comparison on Nonlinear Dynamic System Identification. The best result is shown in bold.	95
6.5	Experiment 2: Performance Comparison on Nonlinear Dynamic System Identification. The best result is shown in bold.	96
6.6	Experiment 1: Performance Comparison on Mackey-Glass Time Series Prediction. The best result is shown in bold.	96
6.7	Experiment 2: Performance Comparison on Mackey-Glass Time Series Prediction. The best result is shown in bold.	97
6.8	Experiment 1: Performance Comparison on Triangular Function Approximation. The best result is shown in bold.	97
6.9	Experiment 2: Performance Comparison on Triangular Function Approximation. The best result is shown in bold.	97

6.10	Test Error and Training Time (in Seconds) of Gaussian Vs. SQ-RBF based SVM. The best result is shown in bold.	98
6.11	FPGA logic utilisation of the proposed designs in relation to previous works using various activation function block array configurations	99
6.12	Resource Utilisation on SinE dataset	100
7.1	LSTM Equations for the Conventional and proposed method.	103
7.2	GRU Equations for the Conventional and proposed method.	104
7.3	Character-level perplexity on the US baby name dataset. There is no difference in perplexity when using the baseline model and Gated Activation method (small is better). . .	117
7.4	Word-level perplexity on the Penn Tree Bank dataset. Using Gated Activation shows a negligible increase in the perplexity on this task (small is better).	118
7.5	Word-level perplexity on the WikiText-2. Using Gated Activation shows a negligible increase in the perplexity on this task (small is better).	118
7.6	Test error rate of LSTM on MNIST and Fashion MNIST. By eliminating the element-wise multiplication, there is a negligible performance degradation between conventional and proposed LSTM models validating the usability of our approach. (Results are average of five runs, and an estimated mean of accuracy with confidence of 95% is recorded)	119
7.7	Character-level Accuracy. There is no difference in accuracy when using the baseline model and Gated Activation method.	119
7.8	Accuracy using floating point full precision and INT5 quantisation.	120
7.9	Resource utilisation of QSU and Standard Booth's Multiplier (for an 8-bit system: input A is 8 bits, input B is 6 bits)	120
7.10	Resource utilisation of LSTM cell	122
7.11	Resource utilisation of LSTM cell (Replacing DSP with Booth's Algorithm)	122
7.12	Resource utilisation of GRU cell	123
7.13	Resource utilisation of GRU cell (Replacing DSP with Booth's Algorithm)	123
7.14	Resource utilisation of Quantised Neural network models with QSU and Booth multiplier	124
8.1	Computation: Mathematical Operator of commonly used learnable activation functions.(LMA: Linear Mixed Activation, NGA: Nonlinear Gated Activation).	129

8.2	The structure of wide residual network topology [14] used in our experiments. Hyperparameter k controls the width of the network and d controls the depth. Groups of convolutions are shown in brackets. Number is how many blocks of layers are used in succession. $N = \frac{d-4}{6}$.	132
8.3	CIFAR-10: The performance accuracy for each activation/topology pair tested in our first experiment. Results are average of five runs, and an estimated mean of the performance accuracy is recorded. The best activation for each topology is shown in bold . The parametric functions achieve higher accuracy, at very small margin, than their non-parametric baselines.	134
8.4	CIFAR-100: The performance accuracy for each activation/topology pair tested in our first experiment. Results are average of five runs, and an estimated mean of the performance accuracy is recorded. The best activation for each topology is shown in bold . The parametric functions achieve higher accuracy, at very small margin, than their non-parametric baselines.	135
8.5	SVHN: The performance accuracy for each activation/topology pair tested in our first experiment. Results are average of five runs, and an estimated mean of the performance accuracy is recorded. The best activation for each topology is shown in bold . The parametric functions achieve higher accuracy, at very small margin, than their non-parametric baselines.	137

Co-Authorship Form

This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

Chapter 2

Wuraola Adedamola, and Nitish Patel. "SQLN: A new computationally efficient activation function." 2018 International Joint Conference on Neural Networks (IJCNN). IEEE, 2018. DOI: 10.1109/IJCNN.2018.8489043

Nature of contribution
by PhD candidate

conceptualization, experiments and analysis drafting the original paper

Extent of contribution
by PhD candidate (%)

90


CO-AUTHORS

Name	Nature of Contribution
Nitish Patel	Main Supervisor: research advise , developing the work, proofreading of the manuscript

Certification by Co-Authors

The undersigned hereby certify that:

- ❖ the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- ❖ that the candidate wrote all or the majority of the text.

Name	Signature	Date
Nitish Patel		2020/10/28

Co-Authorship Form

This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

Chapter 6: sections 6.2, 6.4, 6.7.1

Wuraola, Adedamola, and Nitish Patel. "Computationally Efficient Radial Basis Function." International Conference on Neural Information Processing. Springer, Cham, 2018.

Nature of contribution
by PhD candidate

conceptualization, experiments and analysis, drafting the original paper

Extent of contribution
by PhD candidate (%)

90

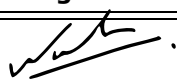
CO-AUTHORS

Name	Nature of Contribution
Nitish Patel	Main Supervisor: research advise, proofreading of the manuscript

Certification by Co-Authors

The undersigned hereby certify that:

- ❖ the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- ❖ that the candidate wrote all or the majority of the text.

Name	Signature	Date
Nitish Patel		2020/10/28

Co-Authorship Form

This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

Chapter 4: Sections 4.2, 4.3, 4.4

"Efficient Activation Functions for Embedded Inference Engines" (Neurocomputing)

Nature of contribution
by PhD candidate

conceptualization, experiments and analysis drafting the original paper

Extent of contribution
by PhD candidate (%)

90

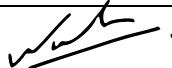
CO-AUTHORS

Name	Nature of Contribution
Nitish Patel	Main Supervisor: research advise, proofreading of the manuscript
Sing Kiong Nguang	Co-supervisor: advice and discussion

Certification by Co-Authors

The undersigned hereby certify that:

- ❖ the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- ❖ that the candidate wrote all or the majority of the text.

Name	Signature	Date
Nitish Patel		2020/10/28
Sing Kiong Nguang	Deceased	2020/10/28

Co-Authorship Form

This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

Chapter 3 and 5 Sections 3.2, 3.3, 3.4, 5.2, 5.3, 5.4 and 5.5

"Resource Efficient Activation Functions for Neural Network Accelerator" (Neural Networks)

Nature of contribution by PhD candidate	conceptualization, experiments and analysis drafting the original paper
Extent of contribution by PhD candidate (%)	90

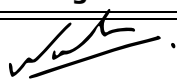
CO-AUTHORS

Name	Nature of Contribution
Nitish Patel	Main Supervisor: research advise, conceptualization, proofreading of the manuscript
Sing Kiong Nguang	Co-supervisor: discussion and advise

Certification by Co-Authors

The undersigned hereby certify that:

- ❖ the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- ❖ that the candidate wrote all or the majority of the text.

Name	Signature	Date
Nitish Patel		2020/10/28
Sing Kiong Nguang	Deceased	2020/10/28

Chapter 1

Introduction

The field of artificial intelligence and its paradigms continue to achieve state-of-the-art accuracy in image recognition [15], language processing [16], autonomous vehicle [17], and other domains [18]. One of the successes of artificial intelligence and its paradigms is due to the availability of high computing hardware platforms. There is a growing interest in underlying hardware platforms for computing neural network operations. Distributed computing infrastructure using several Central Processing Unit (CPU) cores [19] or the use of high-end power-hungry Graphical Processing Units (GPUs) [15] are the two major platforms. Other commonly used platforms of choice are the Field Programmable Gate Arrays (FPGAs) and Application Specific Integrated Circuits (ASICs). Hardware implementation of machine learning models exhibits several challenges due to finite hardware resources, memory constraints, power, and latency. Current neural network models are computationally expensive and memory intensive. The associated computational complexity is highly undesirable from a real-time operation and low power consumption perspective, leading to considerable problems for constrained computing platforms (e.g. mobile devices) that suffer from limitations such as low computational power, low memory capacity, short battery life, strict miniaturisation requirements, and in some cases, lack of necessary support for floating-point arithmetic [20]. Current research [21–23] in the hardware implementation of neural network results in custom hardware-based neural network accelerators surpassing general-purpose processors in terms of energy efficiency and throughput.

1.1 Neural Network Building Blocks

The current trend in neural networks empowered Internet of Things (IoT), autonomous vehicles, and embedded applications to result in advancements in hardware and embedded platforms. However, un-

like CPU and GPU, FPGAs and ASIC are resource and memory constrained. The two fundamental operations of any neural network architecture are the General Matrix Multiply (GEMM) and activation functions. The following are the commonly used elementary functions in the field of machine learning: multiplication (matrix multiplication, convolution, element-wise multiplication), exponentials, logarithms, adders, and floating-point division [22, 24]. The elementary functions can be implemented using lookup tables [21], direct computations such as polynomial power series evaluations [25], hybrid approaches [26], iterative approaches [27] and piecewise approximations [22, 28].

1.1.1 Activation Function

At the heart of every machine learning architecture lies a linear transformation followed by an activation function $f()$. The nonlinear activation function is an important building block of any artificial neural network architecture. Activation functions lie at the core of machine learning architectures allowing them to learn arbitrarily complex mappings and perform complex tasks. Without the activation function, a neural network is a linear regression model. Over the years, activation functions have been shown to increase performance accuracy [3], make it possible to create deeper layers [29], speed up training [3] among others. Many neural network applications can benefit from accurate, scalable, and low-power calculation of nonlinear activation functions. Accurate lookup tables (LUTs) require very large real estate on ANN chips and floating-point engines consume a lot of power and take time to process.

There are several types of machine learning models, each with their own characteristics and applications. Some activation functions are only useful in some machine learning models. For example, deep neural networks are characterised by using asymmetric non-saturating activation functions like ReLU and its variants. Long Short Term Memory (LSTM), a variant of recurrent neural networks, uses symmetric saturating activation functions for its gating mechanism. A normalising function, such as softmax, found use in multiclass classification problems. Sigmoid found use in generative adversarial networks, attention models, and binary classification problems. Sigmoid has significance in terms of its interpretability as probability; it is intuitive to use it for gating or binary classification problems [30]. The study of the optimal activation function is an active area of research. Over the years, researchers have proposed several activation functions to improve network performance (ability to train deeper layers, faster convergence, better accuracy, and others). There is a constant rise in the computational complexity in formulating these activation functions. Figure 1.1 shows the increasing complexity of some popular activation functions found in the literature for the last decade. Apart from the computationally simple ReLU, all other activation functions are expensive with increasing computation.

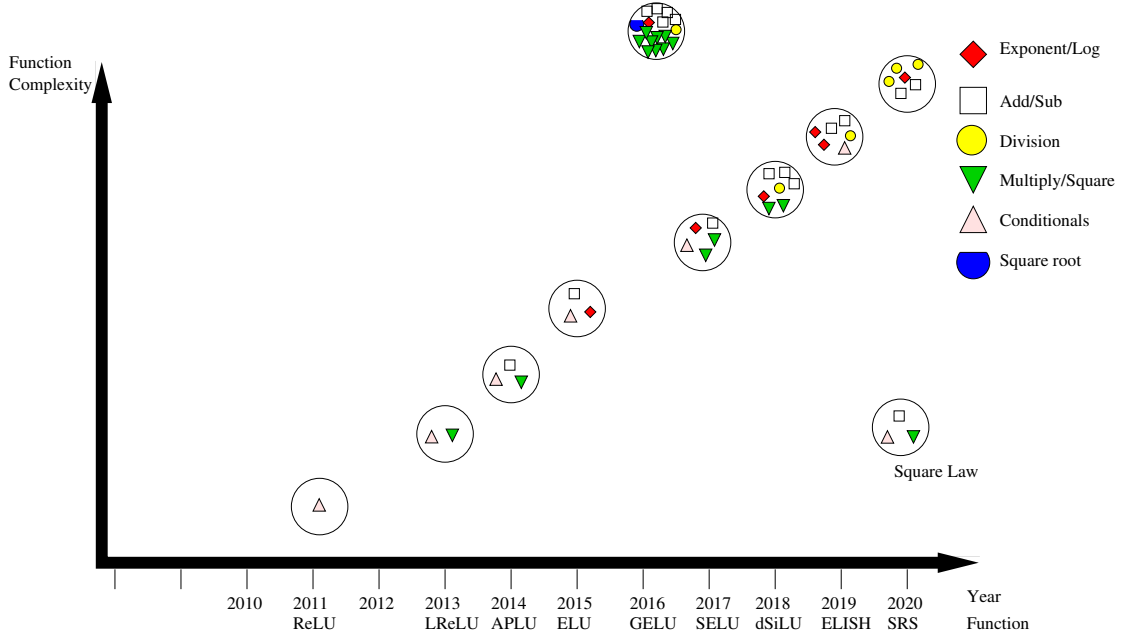


Figure 1.1: Different Variant of ReLU with Increasing Computational Complexity.(LReLU [1], APLU [2], ELU [3], GELU [4], SELU [5], dSiLU [6], ELISH [7], SRS [8]). Square Law is the algorithm proposed in this thesis for simple symmetric and asymmetric activation functions.

1.1.2 Multiplication

Multipliers are the most area and power-intensive arithmetic operators of the hardware implementation of any neural network architecture. The cost of a multiplier varies as the square of the precision for small width operands while the cost of adders and accumulators varies as a linear function of the precision [31]. As discussed in [32], multiplication and accumulation operations are directly related to the computation requirements of the neural network. Therefore, solving the high computation requirements of neural networks is equivalent to reducing the multiplication and accumulation operations. On software-based platforms, techniques such as pruning [33, 34] and compact architectural designs [35, 36] have been shown to reduce the multiplication requirements of network architecture. Furthermore, hardware and software platforms can benefit from the reduction of operands and operations precision, resulting in energy and power efficiency.

Reduction of weights and activations to binary [37] or ternary [20] results in the elimination of multiplications. Multiplications are replaced in binary and ternary networks with a shift operation, XNOR [38], and population count operation. On certain datasets and neural network models, reducing the precision of multipliers (which results in a reduction in area and power) does not affect the performance accuracy [24].

1.1.3 Data Representation

Floating point, fixed point (full precision or quantised) are commonly used to represent neural network operation data. Data representation and precision for the neural network have a direct impact on memory, latency, and resource utilisation on software and hardware platforms. Quantisation [39] shifts values from floating-point continuous values to reduced bit discrete values (fixed-point). A 32-bit floating-point and integer multiplication consume 3.7pJ and 0.2pJ respectively on a 45nm ASIC technology [40]. A lot of work is being done on reducing the data precision during training and inferencing operation of machine learning applications.

The arithmetic operations of neural networks can be truncated down to an 8-bit fixed-point without significant deterioration in inference performance [41–44]. To make neural networks practical on embedded systems, many researchers have focused on training networks with quantised weights and activations. For example, weights may be constrained to take on integer, binary or ternary values, or maybe represented using low-precision (8 bits or less) fixed-point numbers. Extensive experiments performed in [45] show that lower precision (e.g., binary or ternary) requires a lot more training time/epoch to achieve the same accuracy level as higher bits such as 3-8 bits precision.

1.2 Motivation

In recent years, Artificial Intelligence and its paradigms have seen a great spike in both academics and the commercial environment. Several existing software implementations of machine learning models provide high classification accuracy. However, these implementations cannot be used in embedded systems and applications because of the intensive computations required by the individual building blocks of machine learning architectures. In addition, embedded systems development requires meeting challenging constraints such as low cost, limited resources, and low power consumption. Devices that make use of machine learning paradigms are becoming more common. The use of machine learning involves a learning phase, an implementation engine, and an algorithm to match the problem. The implementation engine and inference engine could be a standard computer system or an embedded processor as in mobile phones. Processor chips in mobile phones are now equipped with an artificial neural network engine that can be configured to execute the outcome of a machine learning algorithm. Two important and integral parts of a machine learning implementation is the matrix multiplication and computation of a nonlinear function called the activation function. The GEMM operation is the most computational workload in deep learning. Several optimisations and parallelisation techniques have been introduced in the literature

to speed up this process. With GEMM operations, computationally complex (functions with exponent, logarithm, trigonometric, and floating-point division) activation functions dominate the computational intensity. Using the power series expansion, an exponential function would take at least three multiplications, two floating-point divisions, and several additions. With the acceleration of GEMM operations, the computational intensity introduced by computationally complex activation functions becomes more significant. Hence, the motivation for this work which shows that a fast and resource efficient activation function calculation is essential. The hardware implementation of this work primarily targets the reduction of resource utilisation while the software implementation targets state-of-the-art performance accuracy.

1.3 The Proposed Square Law Solutions

There are two ways to view the implementation of machine learning architecture. Firstly, software implementations on CPUs and GPUs. Secondly, hardware implementation on dedicated digital devices such as on FPGAs and ASICs. A multiplier and a nonlinear mapping function form the core of an ANN inference engine. Our research contribution is based on two innovations that can capitalise on the parallelism offered by FPGA or ASIC. Firstly, we have developed a novel algorithm to build a family of nonlinear functions. The proposed family of functions are based on the square-law algorithm. Our proposed functions use compositions of a square operation and a binary shift to achieve both symmetric and asymmetric nonlinearities. Importantly, a parallel implementation of our proposed method would take fewer silicon resources than transcendental activation functions. Our proposed method is expected to have an execution time that is similar to the Multiply and Accumulate (MAC) operation and hence is very relevant. Secondly, we have developed a low resource multiplier. Importantly, we estimate our algorithms to be more efficient in terms of real-estate, therefore delivering a higher throughput per unit area of silicon. Our solutions offer the potential of making the ANN inference engines compact and hence attractive for embedded solutions. We offer distinct technologies to make a complete neural network on a chip. Our solutions use standard digital building blocks and can be implemented on ASICs or FPGAs. Computationally, our solution eliminates the need for complex mathematics operators such as exponent, division, logarithm, square roots, and others.

1.4 Research Contributions

Overall, the thesis structure and contribution is shown in Figure 1.2. This thesis focuses on designing

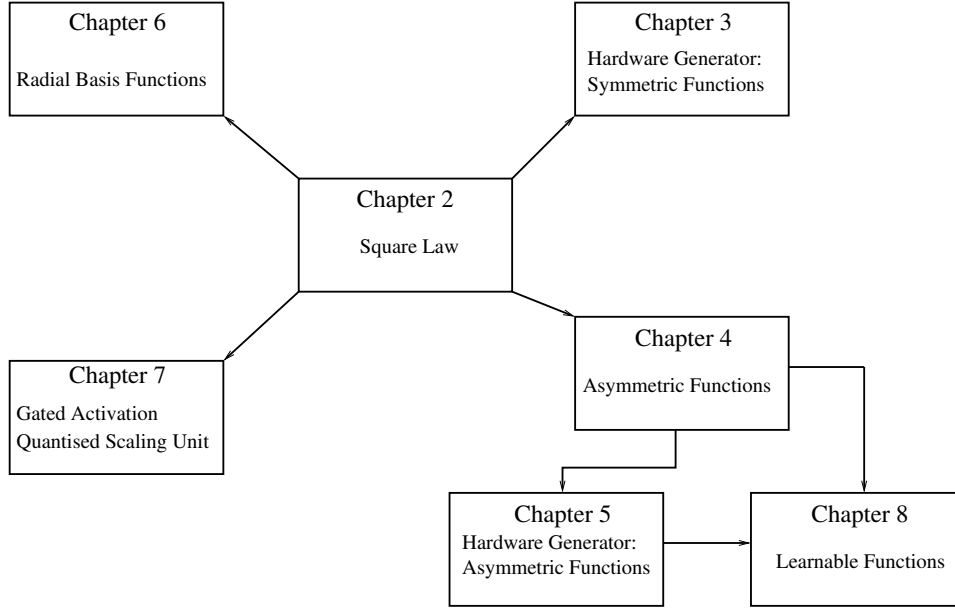


Figure 1.2: Thesis Outline

and implementing novel arithmetic blocks require for efficient and effective hardware implementation of machine learning architectures. The contribution of this thesis can be divided into software and hardware. The solutions proposed in this thesis can be used on software (CPU, ARM, and GPU) as well as on hardware (FPGA and ASIC). The main contributions of this thesis are:

- The development of eight novel activation functions and applications of each. These activation functions are computationally efficient and are accompanied with simple mathematical formulas. We show the universal applicability of the square law in artificial neural networks architectures such as CNN, DNN, RBM, and RNN. Furthermore, the square law can be applied to statistical models such as SVM and logistic regression architectures. Due to the presence of the square function, they are referred to as SQuare NonLinearity (SQNL) family and include: (Chapters 2, 4 & 6)
 - Square-based Nonlinearity (SQNL), which is computationally efficient and finds applications in RNN and shallow feed-forward neural networks.
 - Log Square-based Nonlinearity (Log_SQNL), which is the unipolar version of SQNL, and finds application in logistic regression, RNN, and binary classifiers.
 - Square Linear Unit (SQLU), whose property is identity in positive and nonlinear in the negative region, applications in DNN and CNN.
 - Square REU (SqREU) identity in positive and nonlinear in the negative region, applications

in DNN and CNN.

- Square Swish (Sqish), non-convex, non-monotonic function with applications in deep neural networks.
 - Square Softplus (SQ_Softplus), a soft version of ReLU with applications in RBM and some DNN architectures.
 - Square Radial Basis Function (SQ-RBF), a symmetric bell-shaped curved function with applications in RBFNN and nonlinear multiclass SVM.
 - Square Softmax (SQMAX) normalising function for the output layer of neural networks.
- Development and implementation of hardware efficient method for implementing the SQL family of activation functions. This method is divided into the following: (Chapters 3 & 5)
 - Multi-clock implementation
 - Single-clock implementation
 - Embedded multiplier based implementation
 - Development and implementation of a multifunctional generator (SQ-GEN). The symmetric and asymmetric implementations share commonalities and have also been integrated into a multifunctional generator. This would be attractive in headless inference engines like the designs of TPU™, Nervana™, and NVDLA™.
 - Elimination of resource and power-hungry multipliers in feedforward and recurrent neural networks. Our solution is divided into two, namely: (Chapter 7)
 - Gated Activation: this solution simultaneously computes a symmetric activation function with an integrated scaling functionality. Therefore, this eliminates two of the three element-wise multipliers in an LSTM cell, thereby bringing significant benefits to custom hardware in terms of silicon area and power consumption.
 - Quantised Scaling Unit: A resource-efficient approximate multiplier that replaces the third element-wise multiplier and can potentially replace the resource-hungry multipliers in quantised neural networks.
 - Computationally efficient learnable asymmetric activation functions for deep learning: (Chapter 8)
 - Investigation of several effects of parameterisation on convergence speed and performance accuracy.

- A new learnable combined square-based activation function called Multiple Square Units (MSU). MSU synthesises properties from SQLU, SqREU, SQ_Softplus, and others.

1.5 Thesis Outline

The rest of the thesis is organised as follows:

- In Chapter 2, we present a computationally efficient symmetric activation function. Our proposal uses a novel algorithm named square law and presents a detailed evaluation of this function in comparison to the baselines. The computational footprint on the Intel CPUs and ARM M3 processor is presented.
- In Chapter 3, we present a novel algorithm for generating symmetric activation functions in hardware. Three hardware implementation solutions and computational footprint on embedded devices are explored. We carried out extensive resource utilisation and inference performance on FPGA.
- In Chapter 4, we present four new asymmetric activation functions for deep neural networks. The computational footprint and performance accuracy of these functions show superiority when compared to the state-of-the-art.
- In Chapter 5, the ALU implementation, hardware implementation of the asymmetric activation functions is presented. The computational footprint on an embedded NIOS II processor, FPGA, and ASIC is presented. We also present a generator that can combine multiple activation functions with very minimal resources, The advantages of square-law based functions in low-end hardware devices is discussed.
- In Chapter 6, the FPGA based radial basis neural network and support vector machine architectures are explored. Of particular interest is the popular Gaussian function. We compared our proposed function to the conventional and show speedup and lower area usage on an FPGA.
- In Chapter 7, we propose two solutions for resource-efficient recurrent and feedforward neural networks. The digital implementation of our solution is described in detail. Extensive experiments are performed to show the usability and performance of the two solutions.
- In Chapter 8, the thesis returns to asymmetric activation functions for use in deep learning with a particular interest in evaluating learnable asymmetric functions. The effect of three parameterisa-

tion concepts is explored. We propose an efficient and simple learnable combination of square-law based asymmetric activation functions.

- In Chapter 9, conclusions are presented, and avenues for future work for hardware/embedded neural network building blocks for the training phase are discussed.

Chapter 2

Square Nonlinearity: A Computationally Efficient Activation Function

Abstract

A new symmetric activation function is proposed. This activation function uses the square operator to introduce the required nonlinearity as compared with the use of an exponential term in the popular TanSig. Smaller computational operation count characterises the proposed activation function. The key to the effectiveness of this function is its fast computation. Overall, on different Intel processing units, the proposed function is 1.9x to 3.2x faster than the computationally expensive TanSig. The derivative of the function is linear, resulting in a quicker gradient computation. The scaled and shifted version of the proposed activation function results in a new function. This new function is morphologically similar to sigmoid. The effectiveness and efficiency of the proposed activation function have been compared with TanSig and the computationally efficient ElliotSig functions using selected UCI and MNIST datasets on single-layer feedforward neural networks. An empirical comparison suggests that the proposed function outperforms TanSig and ElliotSig in convergence time as well as in generalisation metrics for most datasets. Further empirical comparison using these square nonlinearities on logistic regression and recurrent neural network architectures results in better performance accuracy.

2.1 Introduction

In recent years, Artificial Intelligence and its paradigms have seen a great spike in both academics and the commercial environment. Deep learning, machine learning, neuromorphic computing, spiking neu-

ral networks, and some other Artificial Intelligence paradigms are very popular due to state-of-the-art simulation, emulation environment, embedded systems, and availability of large data for training. During the implementation process of modelling an Artificial Neural Network (ANN), training data, network architecture, and activation functions among others are important parameters. Activation functions are responsible for the introduction of nonlinearity. They are viewed as the fundamental, vital building blocks that will help achieve the complex capabilities ANNs are expected to deliver [46,47]. In [47–51] the nonlinearity of activation functions is justified if a universal approximator is required. Additional properties of such activation functions include boundedness, continuous, smooth, and monotonically increasing. Any activation function with these characteristics is capable of continuous mapping and has the potential ability to learn and approximate very complex nonlinear mappings in single-layer feedforward neural networks .

The nonlinearity of the activation function can also speed up the training of ANN as described in [52–54]; this characteristic is lacking in some types of functions. An activation function with "*simple*" mathematical terms in its expression can lead to significant speed improvement for solving certain Multilayer Perceptron (MLP) ANN problems. In a simulation or emulation environment, specific activation functions are selected based on the favourable properties required by the learning algorithms - particularly those that are reliant on gradient determination. Before the current wave of deep learning, activation functions used in neural network architectures are of a "*squashing*" type. The sigmoid and TanSig activation functions tends to be a designer's choice as opposed to the hard-limiting functions. In literature, other forms of activation functions have been proposed to solve one or more problems associated with the popular sigmoidal functions. Authors in [55] proposed an activation function called Hexpo capable of eliminating the vanishing gradient issue but this function, as well as its derivative, is computationally expensive due to the presence of exponent term. Another form of activation function capable of eliminating the vanishing gradient problem is the ReLU reported in [56]. The non-saturating characteristics of ReLU function have led to training of very deep networks. ReLU will be discussed fully in chapter 4.

Performance of an ANN model is partly based on the choice of activation function and thus has led to various work in the literature on the factors that make a good activation function and how to choose a good activation function [57] [58]. The authors in [59] discussed the various monotonic and bounded activation function types with their influences on the overall performance of MLP ANN by comparing their mean square error to a fixed number of iterations. The different activation function based on the same architecture and data gives different performance accuracy with the TanSig activation function

resulting in the best accuracy [60]. While computational speed is of importance when using simulators, generalisability (performance with unseen data) is a universal requirement.

This chapter presents two new activation functions that address both of these issues. The first function is referred to as the SQNL and its performance is compared with the TanSig and ElliotSig [61] activation functions. The choice of benchmarking SQNL against TanSig and ElliotSig functions is based on their similar shape and the same output range $(-1, +1)$. The second function is referred to as the Log_SQNL and its performance is compared to the logistic sigmoid activation function.

The TanSig and sigmoid activation functions are computationally expensive due to the presence of an exponential term and floating-point division in their mathematical expression. The ElliotSig is an approximation of the TanSig function because software implementations, as well as some straightforward computing hardware, may not support exponential term directly. Therefore, eliminating the use of exponential term present in TanSig and sigmoid can lead to faster simulation. Using exponential-based functions as activation function for any neural network training can slow down training since it includes a call to high order exponent term.

The proposed method is motivated by an intention to construct a nonlinear activation function on ASIC or FPGA hardware. This chapter does not explore the hardware implementation. A separate chapter (Chapter 3) is dedicated for presenting the novel algorithm for the implementation of these activation functions. The following are the contributions of this chapter:

- Introduction of *two* new computationally efficient activation functions.
- Evaluation of the computational footprint and speedups of the proposed activation functions on Intel CPU and ARM processors.
- Experiments on the hidden representation and gradient computation of the new activation functions during training.
- Usability of the proposed activation functions across multiple machine learning models with an increase in inference speed and sometimes performance accuracy.

2.2 Concepts

This proposal uses a hard nonlinearity and a time average to produce a precise square law nonlinearity. This nonlinearity can also be modeled as an approximate TanSig function. Figure 2.1 shows, as a block diagram, a possible hardware implementation schematic. It comprises an adder, subtractor, and

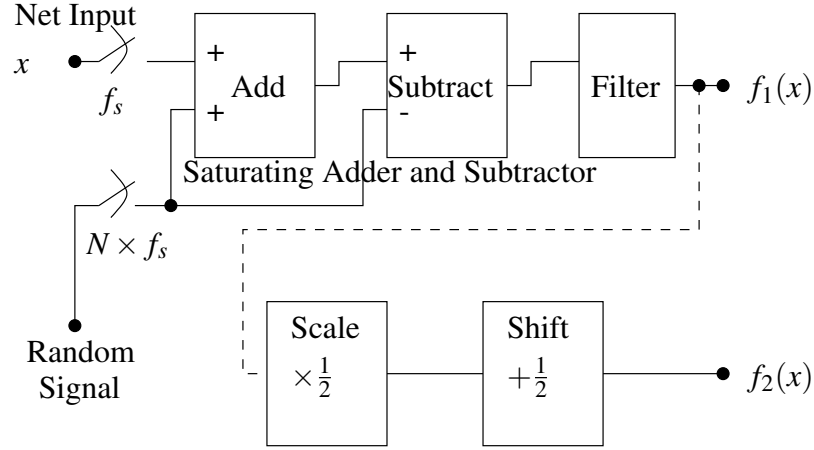


Figure 2.1: Potential ASIC Implementation Block Schematic of Proposed Nonlinear Activation Function

an averaging filter. First, the input and an oversampled random signal are added. The adder is saturating (hard-limited) and hence all sums over the defined limits are clipped. The same random signal is then subtracted from the sum. The subtractor is also required to be hard-limited. The subtraction attempts to recover the original input. If the input was small in magnitude, then the addition with the random signal will not result in clipping, and hence the original input is recovered after the subtraction. However, if the input is large in magnitude, then the addition of the random values will lead to clipping in some of the additions, and the following subtraction will not restore the original signal. The filter averages the restored outputs. The clipping and averaging effectively add a negative offset where the offset is larger when the input is closer to the clipping levels and smaller when the input is closer to zero. The resulting transfer characteristic at f_1 in Figure 2.1 is nonlinear.

Figure 2.2 shows this around $\approx (1, 1)$. If the distribution of the random signal has a very small variance, then the nonlinearity will be narrower, i.e., it will not manifest till the signal is very close to $(1, 1)$. However, if the variance is large, then the nonlinearity is manifest much before $(1, 1)$, i.e., closer to $(0, 0)$. Hence, the variance can be used to shape the nonlinearity.

Figure 2.1 has been simulated in MATLAB. A uniform distribution with a range of ± 1.0 was used. The adder has its hard-limit set at ± 1.0 while the subtractor is at ± 2.0 . The data at $f_1(x)$ was fitted to an approximate TanSig function defined in Equation 2.1. A particular MATLAB simulation results in $a = -1.81$ and $b = 0.18$ (c.f. $a = -2.0$ and $b = 0$ for TanSig) and exhibiting RMS error of 0.023. Thus $f_1(x) \approx f_A(x)$. Figure 2.3 plots the MATLAB simulation.

$$f_A(x) = \frac{2}{1 + e^{ax+bx^3}} - 1 \quad (2.1)$$

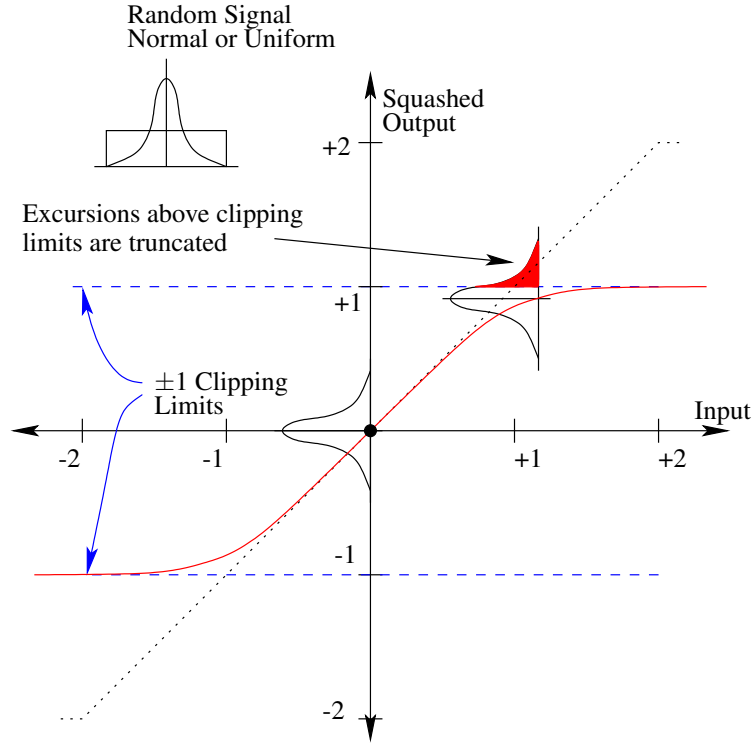


Figure 2.2: Pictorial Description of Transfer Characteristics of the Proposed Method

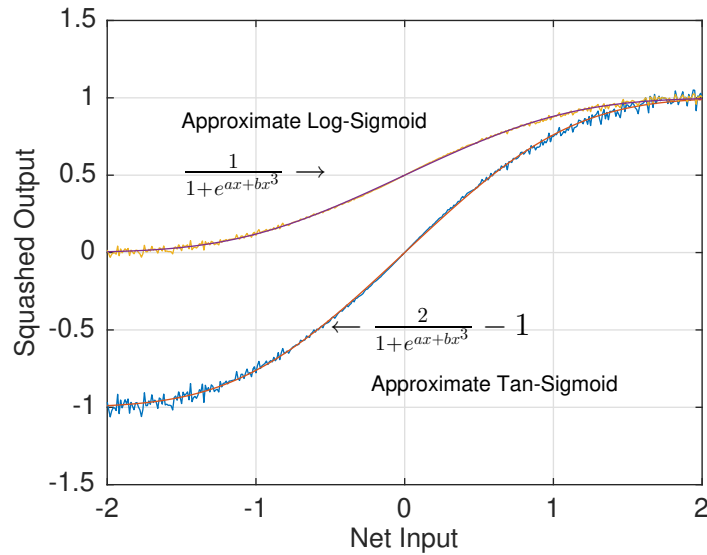


Figure 2.3: Simulated Activation Functions Showing Approximate TanSig and LogSig Behaviour. The traces with the jitter are the outputs of the simulated activation function while the solid trace is a plot of Equation 2.1 with the fitted parameters.

Although this technique generates an approximate TanSig function, it will be shown that its exact equation is more attractive in both forward and backward propagation. A closed-form expression for this technique will be developed using Figure 2.4. If $f_1(x)$ is scaled and shifted by 0.5, an approximate LogSig is produced. This has also been simulated and presented in Figure 2.3.

Definition 2.1

Assume that the clipping levels are ± 1.0 . The random signal is uniform, i.e., $\{u|u \sim \mathcal{U}[-1.0, 1.0]\}$. The input x is the netsum from the input or previous layers where $x \in \mathbb{R}$.

Theorem 2.1

The offset as a consequence of clipping and averaging is given by

$$o(x) = \frac{x^2}{4}, \text{ where } x \in \mathbb{R} \quad (2.2)$$

Proof

Proof of Theorem 2.1: Figure 2.4 a) depicts an input of zero. Since $-1.0 \leq x + u \leq 1.0$, the clipping limits have not been exceeded, and hence there is no change in the recovered magnitude. Figure 2.4 b) depicts a positive input ($x = n$). When the random signal is added, the values in which the sum extends into the red zone are clipped. The offset is the triangular area in the clipping zone and can be determined by integrating the clipped samples in the range $[1.0, 1.0 + n]$.

$$o(x) \Big|_{x=n} = \frac{1}{2} \int_1^{1+n} (x-1) dx = \frac{n^2}{4} \quad (2.3)$$

Thus, the general form of the offset is given by

$$o(x) = \frac{x^2}{4} \quad (2.4)$$

■

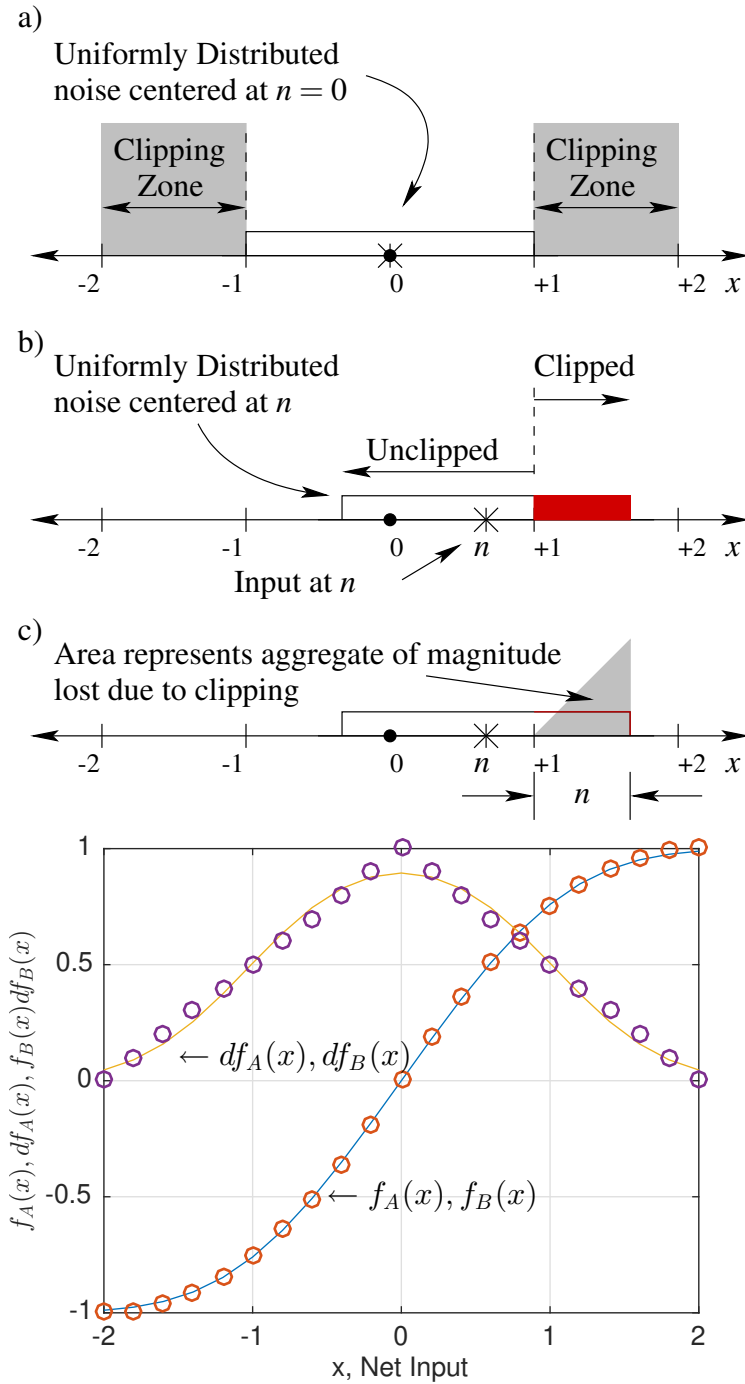


Figure 2.4: Top - Pictorial Analysis of Nonlinear Action. Bottom - Comparison of fitted model and closed form model

Theorem 2.2

Using definition 2.1, the bipolar transfer characteristic is given by

$$f_B(x) = \begin{cases} 1 & : x > 2.0 \\ x - \frac{x^2}{4} & : 0 \leq x \leq 2.0 \\ x + \frac{x^2}{4} & : -2.0 \leq x < 0 \\ -1 & : x < -2.0 \end{cases} \quad (2.5)$$

Proof

Proof of Theorem 2.2: For all positive inputs, the minimum value of $x + u$, i.e., $\min(x + u) > -1.0$ and hence is unaffected. However, all additions of $x + u > 1.0$ are clipped and the subsequent subtraction of u results in a reduction in magnitude. This is captured by

$$f(x) = x - o(x), \text{ where } x > 0$$

However, for negative inputs, it is the maximum value, i.e., $\max(x + u) < 1.0$ that is not affected but rather all values $x + u < -1.0$ are clipped and the subsequent subtraction results in a positive offset. This is given by

$$f(x) = x + o(x), \text{ where } x < 0 \quad \blacksquare$$

Given the square-law nature of the proposed activation function, it will be referred to as *SQNL* in subsequent discussions.

Scaling and shifting Equation 2.5 by half as shown in Figure 2.1, a new transfer characteristic is given by Equation 2.6. This function will be referred to as *Log_SQNL*.

$$f(x) = \begin{cases} 1 & : x > 2.0 \\ (\frac{1}{2}(x - \frac{x^2}{4})) + \frac{1}{2} & : 0 \leq x \leq 2.0 \\ (\frac{1}{2}(x + \frac{x^2}{4})) + \frac{1}{2} & : -2.0 \leq x < 0 \\ 0 & : x < -2.0 \end{cases} \quad (2.6)$$

Derivative

The SQNL and Log_SQNL functions can potentially replace the equivalent exponential-based activation functions. Equations 2.7 and 2.8 describe the derivatives of SQNL and Log_SQNL, respectively .

Square Nonlinearity (SQNL)

$$f'_B(x) = \begin{cases} 0 & : x > 2.0 \\ 1 - \frac{x}{2} & : 0 \leq x \leq 2.0 \\ 1 + \frac{x}{2} & : -2.0 \leq x < 0 \\ 0 & : x < -2.0 \end{cases} \quad (2.7)$$

Square Logistic Sigmoid (Log_SQNL)

$$f'(x) = \begin{cases} 0 & : x > 2.0 \\ \frac{1}{2} - \frac{x}{4} & : 0 \leq x \leq 2.0 \\ \frac{1}{2} + \frac{x}{4} & : -2.0 \leq x < 0 \\ 0 & : x < -2.0 \end{cases} \quad (2.8)$$

The derivative of interest is the Log_SQNL (replacement sigmoid) which achieves a maximum at 0.5 when input is 0.0, unlike sigmoid, which achieves a maximum at 0.25. This means that the gradient signal flows through the Log_SQNL are stronger than the sigmoid. It will be shown later that this is an advantage in recurrent neural network architectures.

Remarks

1. The solid traces in Figure 2.4 (bottom) shows the results of $f_A(x)$, i.e., Equation 2.1 and its derivative $df_A(x)$. The circles trace the closed form equation $f_B(x)$ (Equation 2.5) and its derivative $df_B(x) = 1 \mp \frac{x}{2}$. Equations 2.1 and 2.5 ($f_A(x)$ and $f_B(x)$) are closely matched and suggest that either could be used. However, the closed form is not only accurate but also computationally attractive.
2. The SQNL activation function is continuous ($-\infty - +\infty$), easily differentiable ($f_B(x)$), bounded outputs (range = ± 2.0), and symmetrical as described by the universal approximation theorem [49, 62, 63].
3. Equation 2.5, exhibits a quadratic nonlinearity. Its compact form is desirable because it uses a square operation instead of an exponent.
4. The derivative, Equation 2.7, is well-behaved in the range of ± 2.0 .
5. The optimised derivative of Equation 2.5 requires one difference and one multiplication by 0.5. Both of these are single cycle operations and hence have a computational advantage over the

derivative of the TanSig function given by $(1 - a)(1 + a)$, where $a = \text{TanSig}(x)$.

6. The square law based logistic sigmoid (Log_SQNL) will act as a replacement for the computationally expensive logistic sigmoid $f(x) = 1/(1 + \exp(-x))$.
7. The derivative of Log_SQNL (Equation 2.8) requires two shift operations and one difference. This is computationally efficient over the derivative of LogSig function given by $o(1 - o)$ where $o = \text{LogSig}(x)$.

2.3 Analysis

Equation 2.5 has been benchmarked against the well established TanSig activation function and also against the computationally efficient ElliotSig function [61] and recently proposed ISRU function [64]. The ElliotSig and ISRU function are defined in Equation 2.9 and Equation 2.10, respectively.

$$f(x) = \frac{x}{1 + |x|}, \quad f'(x) = (1 - |f(x)|)^2 \quad (2.9)$$

$$f(x) = x \left(\frac{1}{\sqrt{1 + \alpha x^2}} \right), \quad f'(x) = \left(\frac{1}{\sqrt{1 + \alpha x^2}} \right)^3 \quad (2.10)$$

The positive half of the forward and derivative mappings of TanSig, ElliotSig, and SQNL functions are shown in Figure 2.5 and Figure 2.6. The plot shows that SQNL and TanSig functions approach the asymptotes $[-1, +1]$ quickly as opposed to the ElliotSig. This is a disadvantage for ElliotSig because it means that it will require more iterations to converge for large values [65] because it does not go to 1 or -1 as fast as TanSig and SQNL.

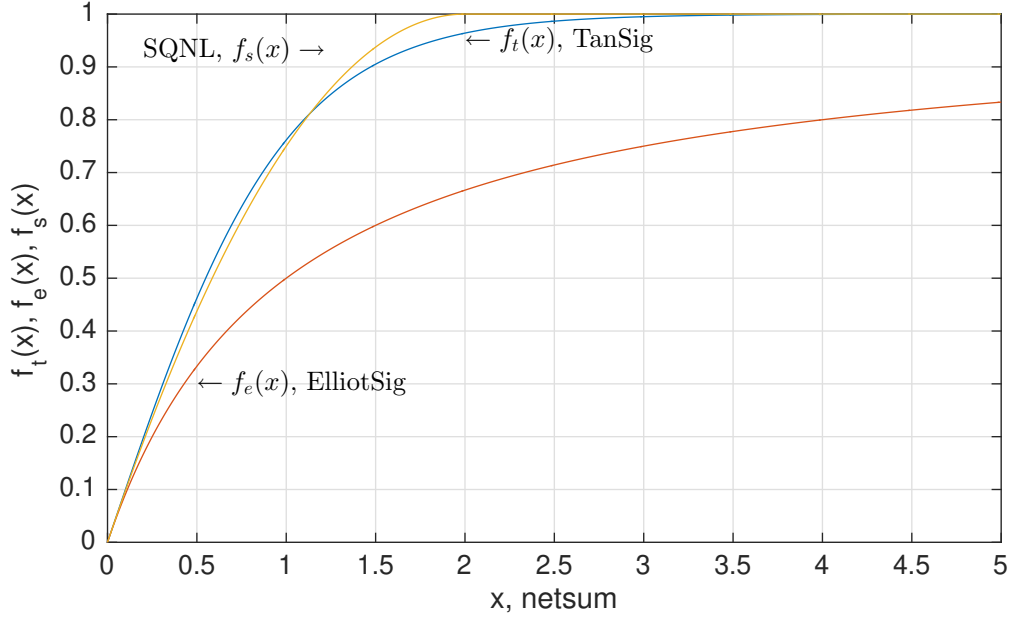


Figure 2.5: Forward mappings - TanSig ($f_t(x)$), ElliotSig ($f_e(x)$), and Proposed SQNL Function ($f_s(x)$)

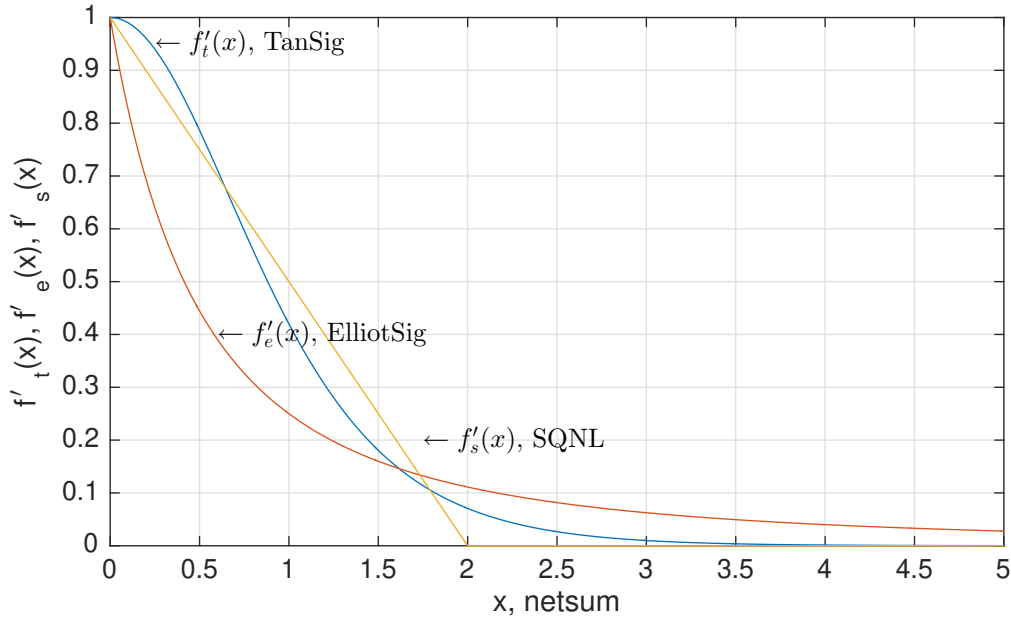


Figure 2.6: Derivative mappings - TanSig ($f'_t(x)$), ElliotSig ($f'_e(x)$), and Proposed SQNL Function ($f'_s(x)$)

Equation 2.6 is also benchmarked against the popular logistic sigmoid function. The positive half of the forward and derivative mappings of the Log-SQNL and sigmoid functions are shown in Figure 2.7 and Figure 2.8.

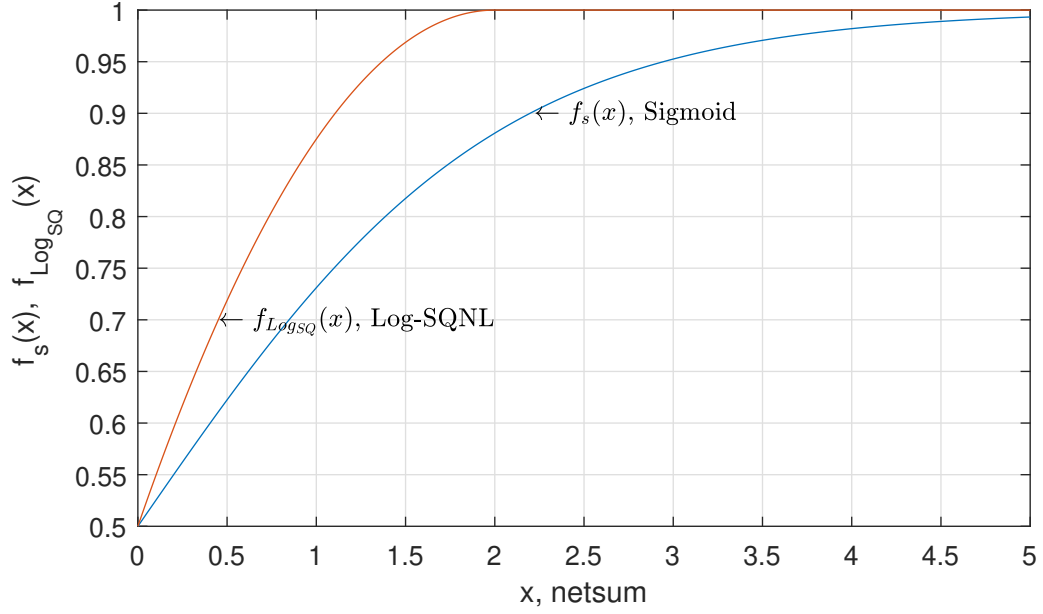


Figure 2.7: Forward mappings - Sigmoid ($f_s(x)$) and Proposed Log_SQNL Function ($f_{LogSQ}(x)$)

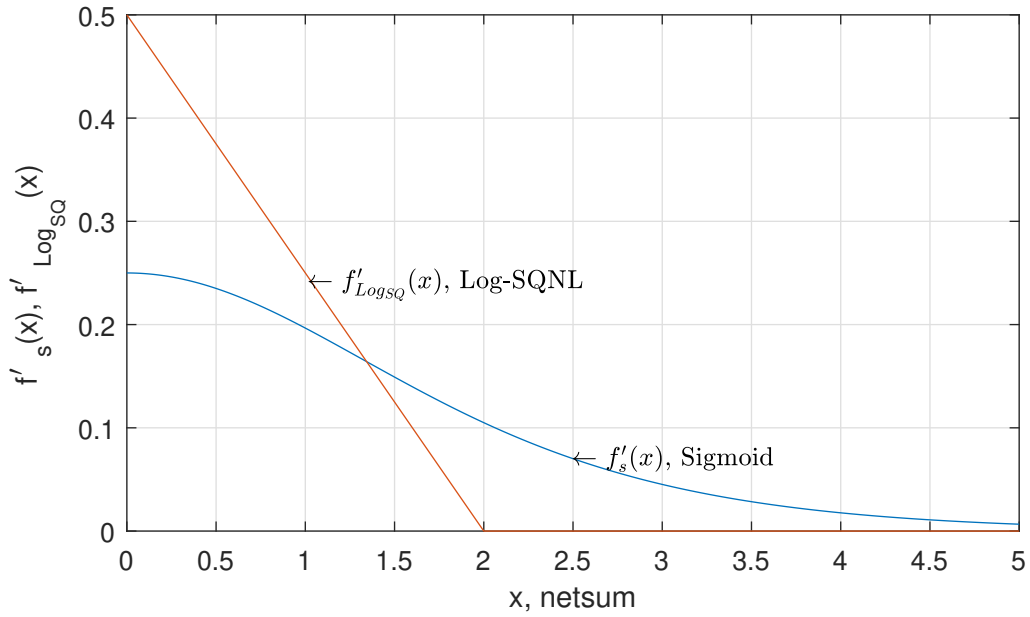


Figure 2.8: Derivative mappings - Sigmoid ($f'_s(x)$) and Proposed Log_SQNL Function ($f'_{LogSQ}(x)$)

2.3.1 Computational Footprint on Intel CPUs

The main advantage of SQNL and Log_SQNL functions over the baselines is that SQNL and Log_SQNL nonlinearities are based on square (multiplication) operator. Square operations are faster to evaluate than exponential for many generations of systems [66]. Intel [66] publishes Clocks per Element (CPE) for several vector functions on their "Vector Mathematics (VM) Performance and Accuracy Data" website. Table 2.1 shows the performance on three different Intel processors for the mathematical operations. The

CPE for each activation function is computed based on this. Vectors of 1000 elements with randomly generated numbers were used, and an average was taken to obtain the results.

Table 2.1: CPU Performance of Commonly used Mathematics Function and the Activation Functions. EP: Enhanced Performance.

Vector Function Single Precision (EP)	Intel Xeon 2699 v3 (Haswell AVX2)	Intel Xeon 2699 v4 (Broadwell AVX2)	Intel Xeon 6148 (Skylake AVX-512)
Square	0.20	0.17	0.13
InvSqrt	0.64	0.54	0.22
Exp	0.80	0.75	0.52
Division	0.54	0.45	0.38
Multiplier	0.25	0.21	0.16
Add/Sub	0.25	0.21	0.16
Abs	0.25	0.22	0.15
TanSig	2.34	2.04	1.54
ISRU	1.59	1.34	0.83
SQNL	0.75	0.64	0.47
ElliotSig	1.04	0.88	0.69
Sigmoid	1.59	1.41	1.06
Log_SQNL	1.00	0.85	0.63

Table 2.2 shows the speedups reported for square-based activation function when compared with the benchmarks. It can be seen that square-based activation functions show a significant speedup over the benchmarks; this will mainly be useful in CPU based inference engines.

Table 2.2: Speedups: CPU Performance of Symmetric Activation Functions. EP: Enhanced Performance.

Vector Function Single Precision (EP)	Intel Xeon 2699 v3 (Haswell AVX2)	Intel Xeon 2699 v4 (Broadwell AVX2)	Intel Xeon 6148 (Skylake AVX-512)
TanSig/SQNL	3.1×	3.2×	1.9×
ISRU/SQNL	2.1×	2.1×	1.8×
ElliotSig /SQNL	1.4×	1.4×	1.5×
Sigmoid /Log_SQNL	1.6×	1.3×	1.7×

2.3.2 Computational Footprint on the ARM M3 Processor

Table 2.3 shows the time taken to compute 1000 (double precision and Q16.16 fixed point) activations and the achieved speedups on a Cypress PSoC 5 (ARM M3, 24MHz). Table 2.3 shows the full computational time in milliseconds of Tanh, ElliotSig, SQNL, and ISRU activation functions as well as their respective derivatives. Table 2.4 shows the speedup achieved by SQNL function relative to the benchmarks.

Table 2.3: Computational time of Tanh, ElliotSig, SQNL, and ISRU functions and their derivatives using ARM M3 processor. This is an average of 1000 calculations.

Function	Forward (ms)	Derivative (ms)
Tanh Floating Point	182.17	12.78
Tanh Fixed Point	8.11	2.23
ElliotSig Floating Point	32.87	41.43
ElliotSig Fixed Point	14.14	20.88
ISRU Floating Point	92.18	270.13
ISRU Fixed Point	51.89	149.19
SQNL Floating Point	15.22	9.87
SQNL Fixed Point	1.86	0.89

The processor does not include a hardware exponential instruction and consumes 182ms to compute 1000 Tanh activations. The multiplication and division operations have better hardware support, and hence the SQNL mapping consumes only 15ms. Subsequent discussions will make comparisons relative to the Tanh timings to factor out the absolute clock frequency. An open-source fixed point library `libfixmath` [67] using a Q16.16 format was evaluated. The fixed point Tanh activation is 22.5 (182.17/8.11) times faster than the double-precision while the derivative is 5.7 times faster. The SQNL formulation is very well suited for fixed-point instructions. The performance benefits are attributed to two aspects:

- The square operator: On ARM M3 processors, the multiply instruction takes 3-7 clock cycles
- The binary division ($/4$): A full division operation takes between 2-12 clock cycles, but SQNL requires a divide 4, which is technically a shift operation. For this shift operation, in reality, only the upper bits need to be propagated and hence consume no resources or time.

Table 2.4: Speedups: Computational time of Tanh, SQNL, ElliotSig, ISRU function and their derivative using ARM M3 processor. This is an average of 1000 calculations.

Function	Forward	Derivative
Tanh/SQNL Floating Point	$11.97 \times$	$1.29 \times$
Tanh/SQNL Fixed Point	$4.37 \times$	$2.49 \times$
ElliotSig/SQNL Floating Point	$8.55 \times$	$3.66 \times$
ElliotSig/SQNL Fixed Point	$19.22 \times$	$19.83 \times$
ISRU/SQNL Floating Point	$6.05 \times$	$27.36 \times$
ISRU/SQNL Fixed Point	$27.89 \times$	$167.62 \times$

2.3.3 Decision Boundary and Activation Over Time

This analysis focuses on the decision boundary and activation values during training. We compared the hyperplane (decision boundary) for two morphologically similar functions: SQNL and the popular Tan-Sig. We used a two dimensional, nonlinearly separable dataset generated from a uniformly distributed random number of sample size 200 with a noise of 20%. We trained a single layer neural network with three hidden neurons. We used gradient descent without any other hyperparameter except for the learning rate and the change in activation functions. The hyperplane achieved after training is illustrated in Figure 2.9a-b. As shown, although the functions are mathematically different, they are broadly similar and hence result in similar decision boundaries.

We performed another experiment on the MNIST [68] dataset. We used a single hidden layer of 128 neurons of SQNL activation functions with softmax units at the output layer.

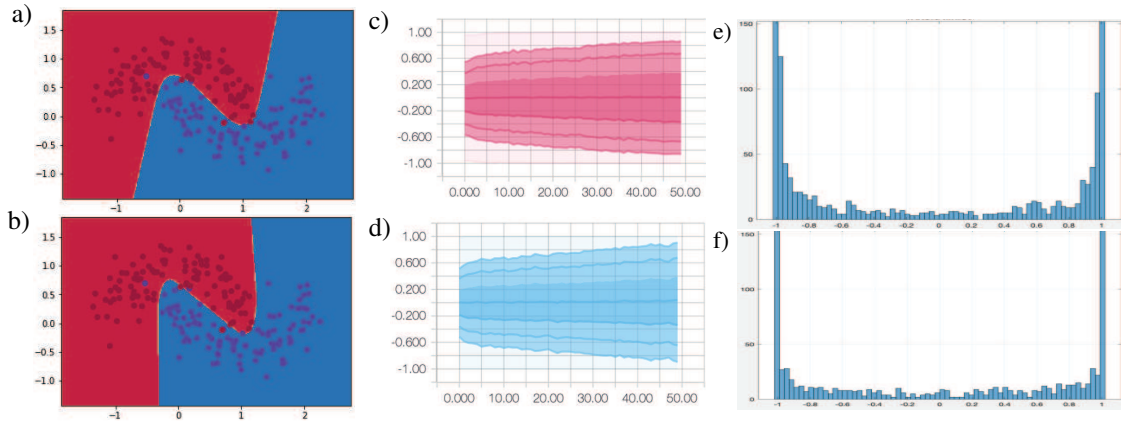


Figure 2.9: a) Toy dataset: TanSig decision boundary b) Toy dataset: SQNL decision boundary c) MNIST dataset: TanSig hidden layer activations during training d) MNIST dataset: SQNL hidden layer activations during training e) MNIST dataset: TanSig histogram distribution of activations f) MNIST dataset: SQNL histogram distribution of activations.

The difference between SQNL and TanSig activations during training, as illustrated by the activation over time in Figure 2.9c-d, is minimal. Furthermore, Figure 2.9e-f shows the histogram of the activa-

tion values, and there is little to no difference in the output. This further supports the claim that the computationally efficient SQLN can successfully replace the computationally expensive TanSig.

2.4 Performance Comparison

In this section, comprehensive experiments are performed on the replacement of baseline activation functions (TanSig, sigmoid, ElliotSig) with square-based activation functions. Three neural network architectures are explored and the performance and advantages of square based activation function are shown empirically.

2.4.1 SQLN for Shallow Supervised Learning

The forward and derivative of the SQLN expression make it straightforward for optimisation. In other words, the division and square operation can be optimised and replaced with multiplication. This is not possible for TanSig and ElliotSig functions. Division and exponential operations are known to be more computationally expensive (speed) than multiplication operations (Table 2.1). The performance accuracy and convergence of the SQLN function will be proved using the traditional backpropagation, Levenberg-Marquardt backpropagation, and Resilient backpropagation (Rprop) variants. The choice of different training algorithm is to prove that SQLN is independent of the training algorithm. The performance on the training set and network generalisation (performance with unseen data) is presented. Commonly used datasets from UCI repository [69] have been selected for these experiments. The pattern set was randomly partitioned for training and validation in an 80:20 ratio. The minimum network configuration comprised of five neurons in the hidden layer, with both hidden and output nodes having SQLN/TanSig/ElliotSig activation function for all classification problems. On the other hand, for the regression problems, the output nodes have the linear activation function. The networks were trained with the same initial conditions (randomisation of weights and biases). The performance comparison was based on two criteria: one distinguishes on the convergence speed while the other is based on generalisability. The performance comparison of these two criteria is expressed with respect to TanSig by normalising as shown in Equation 2.11. If T_N is the normalised ratio, T_{TS} is the performance of the TanSig function (baseline performance), and T_{FUT} is the performance of the function under test.

$$T_N = \frac{T_{TS}}{T_{FUT}} \quad (2.11)$$

Thus, a result greater than 1.0 indicates a performance better than TanSig. All experiments were carried out on a 64-bit windows server 2012 R2 standard with eight processors and the Intel Xeon CPU running at 2.90GHz with 32.0GB of installed memory (RAM) in MATLAB environment.

Experiment 1

Here, the purpose is to compare the number of epochs to reach a predefined Mean Square Error (MSE). The weights and biases of each network were identically initialised. The training was terminated when an MSE criterion was reached. The experiment was repeated a total of 100 times, averaged, and the results are presented. Figure 2.10 shows the results of using the traditional backpropagation algorithm for training the selected datasets. Here, the number of epochs has been normalised using Equation 2.11. Thus, data points above 1.0 indicate fewer epochs taken relative to a TanSig network. With $\approx 90\%$ success rates, all the networks exhibited a similar generalisation (based on unseen test data). The Wine and HeartC datasets do not meet the MSE criterion after 20000 epochs for all of the activation functions. The SQNL's performance exceeds and matches the TanSig and ElliotSig in most of the datasets. Using the same initial conditions, the three activation functions were used alongside with Levenberg Marquardt and Resilient Backpropagation variants with their results displayed in Figure 2.11 and Figure 2.12, respectively. Using the Levenberg Marquardt algorithm, the SQNL function continuously outperforms both TanSig and ElliotSig in terms of convergence speed for datasets that meet the MSE criterion. Finally, Figure 2.12 shows that SQNL reaches convergence faster than the other two functions except for the Wine datasets which fail to reach the predefined MSE. In conclusion, SQNL has proven to be an efficient function in attaining quick convergence during training. The breast cancer and ionosphere datasets are referred to as BC and IS in our graphical representation.

Experiment 2

Here, we evaluate the network's performance after a fixed number of training epochs across three training algorithms. The networks were trained for a fixed number of epochs. Figure 2.13, Figure 2.14, and Figure 2.15 compare the performance of the SQNL and ElliotSig relative to that of the TanSig. Figure 2.13 and Figure 2.15 show that when using backpropagation, the SQNL based networks perform better at generalisation than the ElliotSig. However, when the SQNL is compared with TanSig, then, as with experiment 1, neither show a clear superiority. Figure 2.14 compares these using the Levenberg-Marquardt algorithm. Here too, the SQNL performs better than the ElliotSig, however, no distinction can be made between the SQNL and the TanSig.

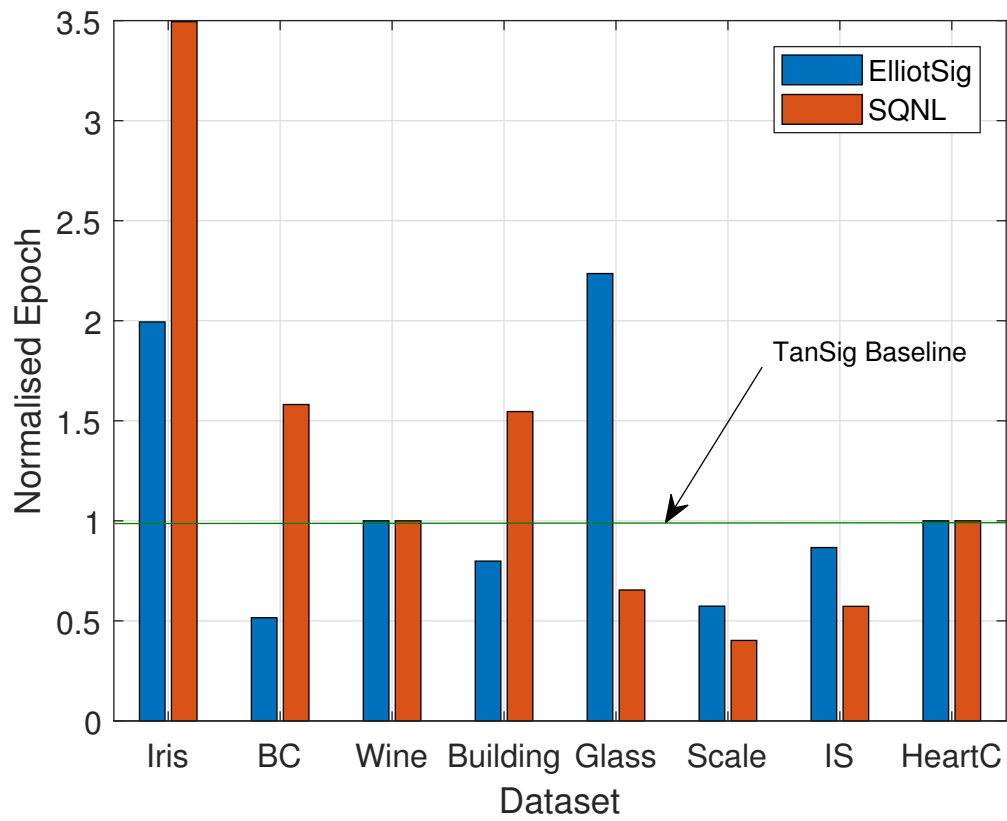


Figure 2.10: Results of Experiment 1 For Traditional Backpropagation - BC:Breast Cancer, IS: Ionosphere

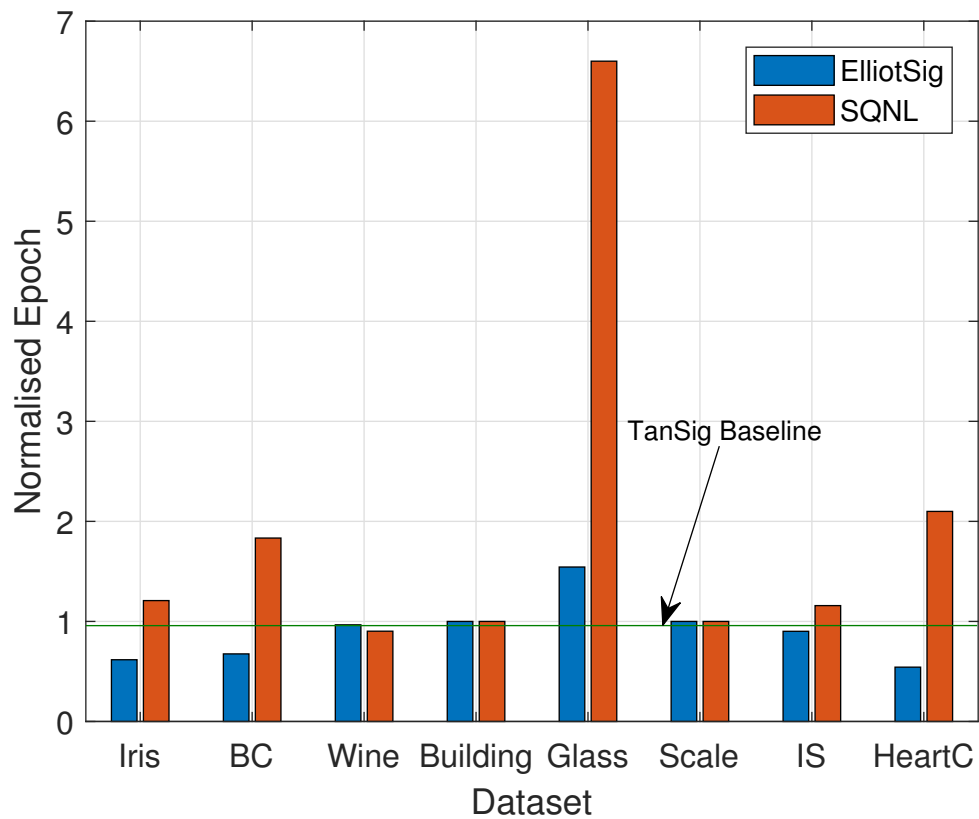


Figure 2.11: Results of Experiment 1 For Levenberg Macquardt Backpropagation - BC:Breast Cancer, IS: Ionosphere

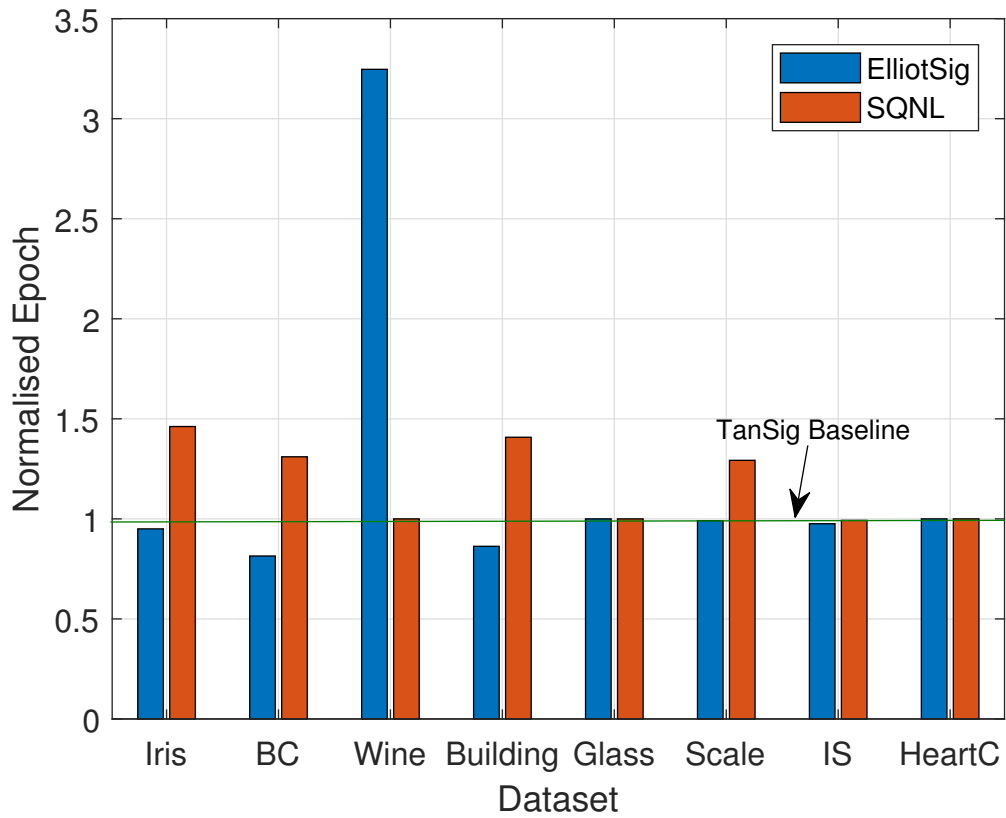


Figure 2.12: Results of Experiment 1 For Resilient Backpropagation - BC:Breast Cancer, IS: Ionosphere

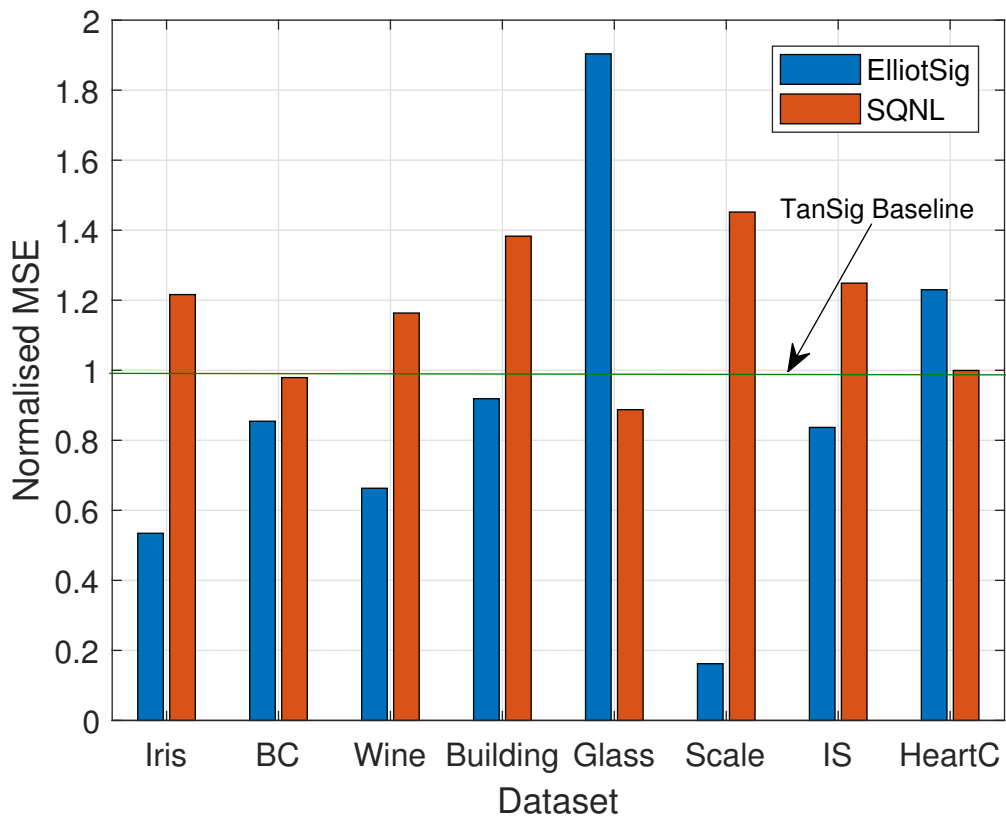


Figure 2.13: Results of Experiment 2 For Traditional Backpropagation - BC:Breast Cancer, IS: Ionosphere

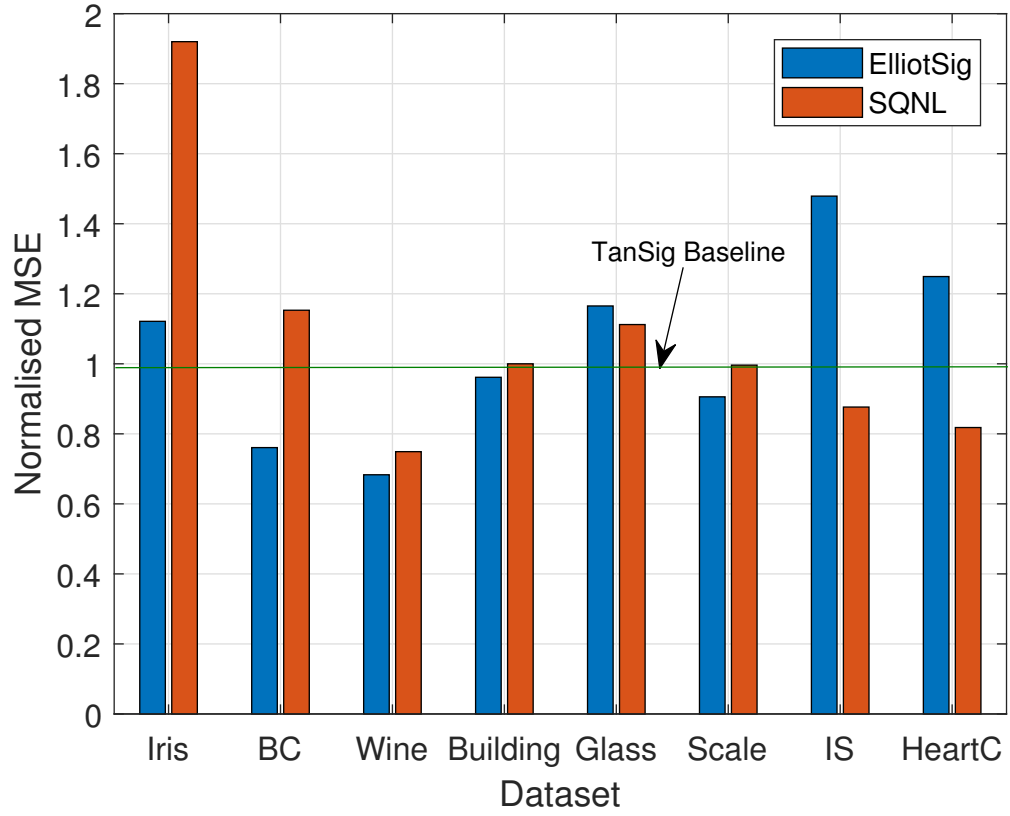


Figure 2.14: Results of Experiment 2 For Levenberg Macquardt Backpropagation - BC:Breast Cancer, IS: Ionosphere

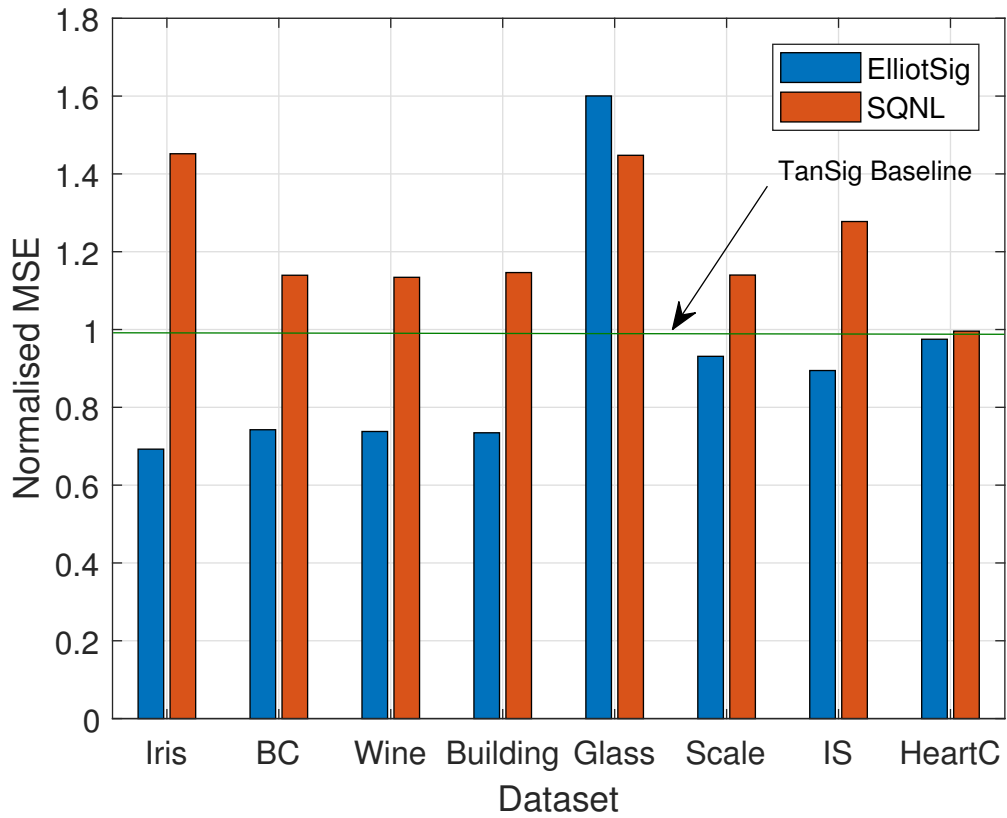


Figure 2.15: Results of Experiment 2 For Resilient Backpropagation - BC:Breast Cancer, IS: Ionosphere

With the Glass dataset, the ElliotSig consistently performs better than the others. The SQNL always performs better on the Iris dataset. Although the overall performance of the SQNL is better than the others, the variation in performance suggests a strong data set dependence, and hence any conclusions on the superiority of any one of the activation functions are premature and potentially not possible to establish. Importantly, the SQNL is not inferior to the well-established TanSig.

Classifier Performance

The geometric mean (G-mean) and its variants have also been used to strengthen our claims of the effectiveness of the SQNL. The sensitivity, specificity, and G-mean metrics mathematical expressions are given in equations below

$$Sensitivity = \frac{TP}{TP + FN} \quad (2.12)$$

$$Specificity = \frac{TN}{TP + FP} \quad (2.13)$$

$$G - mean = \sqrt{Sensitivity \times Specificity} \quad (2.14)$$

where TP indicates the number of positive elements predicted as positive, TN indicates the number of negative elements predicted as negative, FP indicates the number of negative elements predicted as positive, FN indicates the number of positive elements predicted as negative. How well a classifier can recognise/classify positive examples is described as the classifier's sensitivity. On the other hand, specificity is how well negative examples are recognised by the classifier. The geometric mean of the specificity and sensitivity of a classifier is the G-mean.

Table 2.5 shows clearly that for all datasets, the classifiers trained with SQNL function were able to recognise positive examples. Whereas the specificity of each dataset was not conclusive on which function is superior. However, Table 2.6 shows a distinct performance of SQNL for all datasets using the accuracy and G-mean metrics.

Performance on MNIST Dataset

The MNIST dataset was used to prove the convergence ability of the SQNL function on large datasets. One-third of the data was set aside for testing. After several initial analyses, a hidden layer of 250 neurons was selected because it gave up to 98.6% on unseen data. We set up our experiment using the same initialisation parameters. 100 networks were trained to get an average convergence time. The output neurons were set to softmax activation function. The hidden neurons used the SQNL function

Table 2.5: Sensitivity and Specificity on Classification Dataset

Data Set	Function	Sensitivity (%)	Specificity (%)
Breast Cancer	ElliotSig	89.4	96.8
	TanSig	89.6	97.8
	SQNL	91.5	97.8
Glass	ElliotSig	90.9	80.0
	TanSig	90.6	72.7
	SQNL	93.8	81.8
Diabetics	ElliotSig	71.7	65.9
	TanSig	73.2	69.0
	SQNL	74.8	68.1
Ionosphere	ElliotSig	76.9	84.2
	TanSig	79.2	94.4
	SQNL	83.7	75.0
HeartC	ElliotSig	75.0	61.5
	TanSig	79.3	68.0
	SQNL	82.1	69.2
Ovarian	ElliotSig	77.8	100
	TanSig	72.2	96.2
	SQNL	92.3	93.5

Table 2.6: Accuracy and G-mean on Classification Dataset

Data Set	Function	Accuracy (%)	G-mean (%)
Breast Cancer	ElliotSig	94.3	93.0
	TanSig	95.0	93.6
	SQNL	95.7	94.6
Glass	ElliotSig	88.4	85.3
	TanSig	86.0	81.2
	SQNL	90.7	87.6
Diabetics	ElliotSig	70.1	68.7
	TanSig	72.1	71.1
	SQNL	72.7	71.4
Ionosphere	ElliotSig	78.9	80.5
	TanSig	83.1	86.5
	SQNL	80.3	79.2
HeartC	ElliotSig	68.5	67.9
	TanSig	74.1	73.4
	SQNL	75.9	75.4
Ovarian	ElliotSig	90.9	88.2
	TanSig	86.4	83.3
	SQNL	93.2	92.9

Table 2.7: Convergence Speed On MNIST Dataset

Function	Epoch
ElliotSig	17.74
TanSig	8.45
SQNL	8.11

and the selected benchmarks. Table 2.7 shows the average over 100 of the number of epochs to reach the predefined performance goal. On average, SQNL slightly outperforms TanSig, hence confirming our claim that SQNL can be used in place of TanSig.

2.4.2 Log_SQNL for Binary Logistic Regression

In a logistic regression problem, a cost function, update rule, and nonlinearity is required. Consider a binary classification problem of n -dimensional feature vector given by $X = (X_1, X_2, X_3, \dots, X_n)$ and the class variable represented as $Y \in \Theta = \{\theta_1, \theta_2\}$. The probability that $Y = \theta_1 \mid X = x$ is denoted by $p_1(x)$. For binary logistic regression model, it is assumed that

$$\log \frac{p_1(x)}{1 - p_1(x)} = \beta^T x + \beta_0 \quad (2.15)$$

where $\beta \in \Re^n$ and $\beta_0 \in \Re$ are learning parameters. Solving Equation 2.15 for $p_1(x)$

$$p_1(x) = \frac{1}{1 + \exp[-(\beta^T x + \beta_0)]} \quad (2.16)$$

We show that we can eliminate the computationally expensive exponent term in Equation 2.16 with a Log_SQNL defined in Equation 2.6. Therefore, Equation 2.16 becomes $p_1(x) = \text{Log_SQNL}(\beta^T x + \beta_0)$ where Log_SQNL is described by Equation 2.6.

By maximising the conditional log-likelihood described in Equation 2.17, we can estimate the parameters β and β_0 given a learning set $\{(x_i, y_i)\}_{i=1}^d$ is

$$\ell(\beta, \beta_0) = \sum_{i=1}^d y_{i1} \log p_1(x_i) + (1 - y_{i1}) \log(1 - p_1(x_i)) \quad (2.17)$$

where $y_{i1} = 1$ if $y_i = \theta_1$ and $y_{i1} = 0$ otherwise.

The Performance Accuracy of LogSig versus Log_SQNL based Regression is shown in Table 2.8. Best performance accuracy is shown in bold font with the training set up parameters of 10000 epoch and 0.1 learning rate. The description of the datasets is available at [70].

Table 2.8: Performance Accuracy of Sigmoid/LogSig Vs SQLN based Logistic Regression. A comparable result between Sigmoid and Log_SQLN is achieved.

Dataset	LogSig Accuracy (%)	Log_SQLN Accuracy (%)
Sonar	78.50	75.36
Breast	96.58	96.58
Glass	86.77	88.42
Ionosphere	82.84	83.33
HeartC	61.09	61.09
Diabetics	60.09	56.01
Liver Disorder	57.27	57.65
Lymphography	31.03	31.03
Lenses	1.0	1.0

2.4.3 SQLN and Log_SQLN for Recurrent Neural Network

Softly saturating nonlinear functions makes it difficult to make hard decisions and thereby slows down the training in RNN, as described in [71]. The exponential is the cause of the soft-saturation in the sigmoid and hyperbolic tangent in recurrent networks. Since SQLN is based on quadratic, it has an inbuilt advantage of making hard decisions. The problem of negligible gradients is reduced in SQLN because the derivative given by $f'(x) = 1 \pm x/2$ will give precise zero at saturation (which is at $x = 2$). Moreover, the strong gradients associated with Log_SQLN (section 2.2) are important for well-flowing gradients over time. Therefore, we applied our functions (SQLN and Log_SQLN to replace TanSig and LogSig, respectively) to LSTM and GRU recurrent architectures. In our analysis, we included the approximate TanSig referred to as Hardtanh and approximate sigmoid referred to as Hardsig. Hardtanh is defined mathematically as $\max(-1.0, \min(1.0, x))$ while Hardsig is defined as $\max(0.0, \min(1.0, (x + 2.0)/4.0))$. It is important to know that SQLN and Log_SQLN are not an approximation to TanSig and sigmoid activation function but rather, standalone functions.

Long Short Term Memory

The memory and several cells associated with RNN made them highly computationally intensive. As described in [72], much research has been directed into improving the execution efficiency of RNN models, among which are parallelisation, model pruning, quantisation, and finally, data access accelerator. One interesting aspect of LSTM that has been neglected is the gates activation functions. An LSTM cell has five gates, which are equivalent to five activation functions.

The IMDB (a large movie review dataset) contains 25,000 highly popular movie reviews. The model and description we used is available at [73]. The Kaggle dataset and the model are available at [74]. The

model and description for the MNIST dataset are available at [75]. The Sequential MNIST dataset and the model description is described in [76]; we used the row-by-row Sequential MNIST example. We replaced all sigmoid and TanSig in the reference model with their SQNL counterparts.

Table 2.9: Performance Accuracy using square-based LSTM network. Changing the activation function to SQNL yielded about 3% for the Kaggle Model and 1% for the IMDB dataset improvement in performance accuracy. Use of Hardtanh and Hardsig performs worse than TanSig and sigmoid functions. (SQNL and log_SQNL to replace TanSig and sigmoid respectively)(Results are average of 5 runs, and an estimated mean of accuracy with the confidence of 95% is recorded)

Dataset	TanSig/ Sigmoid (%)	Hardtanh/ Hardsig(%)	SQNL/ Log_SQNL(%)
IMDB	86.3 \pm 0.9	82.58 \pm 3.3	87.14 \pm 0.8
Kaggle	87 \pm 2.9	87.8 \pm 1.5	90.2 \pm 0.4
MNIST	98.6 \pm 0.1	98.5 \pm 0.1	98.7 \pm 0.1
Sequential	95.9 \pm 1.8	90.3 \pm 3.8	95.9 \pm 0.5

Gated Recurrent Units

The GRU is known as the simpler RNN architectures. The GRU architecture is made up of three gates which is equivalent to three activation functions. We follow the same approach as defined in the LSTM experiment. The architecture for MNIST is available at [76]. Finally, we used the dynamic RNN as defined in [76] and modified it to GRU. This was used to perform dynamic computation over sequences with variable length. Table 2.10 shows the performance accuracy and our proposed model shows a higher performance accuracy when compared to the baseline.

Table 2.10: Performance Accuracy using square-based GRU network. (Results are average of 5 runs, and an estimated mean of accuracy with the confidence of 95% is recorded)

Dataset	Tanh/ Sigmoid (%)	Hardtanh/ Hardsig(%)	SQNL/ Log_SQNL(%)
MNIST	85.15 \pm 0.9	84.45 \pm 1.2	85.16 \pm 1.3
Toy	98.0 \pm 0.2	97.0 \pm 0.3	99.0 \pm 0.2

2.5 Conclusion

In this chapter, we proposed a novel nonlinear activation function with proven and accurate mathematical equations. The proposed SQNL and Log_SQNL functions circumvent the use of the exponential term and floating-point division resulting in a reduction in computational time. Their derivatives can be highly optimised and require a single cycle operation. Another advantage of the SQNL function is that it eliminates the disadvantages of ElliotSig, therefore leading to faster convergence for large val-

ues. The performance accuracy and convergence speed of the SQL function have been proven using three variants of the backpropagation algorithm. Selected UCI datasets were used to evaluate performance. Network architectures using the proposed SQL activation function train faster to reach similar performance goals than the baseline functions. Also, network generalisability is mostly better. The SQL consistently records better accuracy and G-mean metric analysis for all datasets. The SQL is a potentially worthy alternative because it offers faster-training speeds without adverse effects on generalisability. The hard nonlinearity of SQL and Log_SQL is particularly useful for RNN architectures. We record state-of-the-art accuracy for some of the tested datasets.

Chapter 3

Efficient Hardware Calculator for SQLN Function

Abstract

The direct implementation of the preferred nonlinear function for single-layer feed-forward and recurrent neural network architectures presents a problem in hardware. This is a result of the presence of floating-point division and exponentiation operations in these functions. These operations require a large amount of time and hardware resources. This has led to the implementation of these nonlinear functions on hardware using approximation methods or lookup tables. These methods all have one of these challenges associated with them. In this chapter, we present a novel algorithm for generating the SQLN function on hardware. The hardware implementation of the proposed algorithm is highly amenable to parallelisation and extremely resource-efficient when compared to other forms of hardware approximation. Also, the nonlinearity and the numerical precision are adjustable and hence offer additional options for optimising hardware designs. For an 8-bit resolution, we recorded 1.79x to 3.75x throughput per gate on an ASIC platform when compared to lookup table implementations. With 12-bit resolution, we record 14.77x to 30.35x throughput per gate.

3.1 Introduction

Hardware implementation of ANN is viewed as the next technological frontier [77–79]. Although research in this area has been active over several decades, there has been a recent spike in activity due to a multitude of factors: availability of low power computing engines and FPGAs, demand for ‘smart’

solutions to improve the quality of life and others. Implementation of ANN poses challenges on several fronts. One dominant issue is the drive to achieve biological realism [80]. Thus, hand-in-hand with hardware developments, there has been progress in network architectures and implementation methodologies that capitalise on the availability of parallel hardware [81, 82]. The biggest shift in paradigm to achieve biological realism is based on single-bit signal encoding/representation techniques. The two broad classes in single bit representations are stochastic and deterministic bit-streams [83–86]. However, the most dominant paradigm is the multi-bit representation in which signals are propagated through multipliers and non-linear elements. The nonlinear elements are viewed as the fundamental, vital building blocks that will help achieve the complex capabilities that ANNs are expected to deliver [25, 46]. Neural network architectures such as shallow multilayer perceptron, recurrent neural networks, restricted boltzmann machines, auto-encoders popularly use sigmoid and hyperbolic tangent (TanSig/Tanh) activation functions as opposed to the simple rectifier linear units (ReLU) use in deep neural networks.

The sigmoid and Tanh nonlinear functions are the resource-hungry aspects of hardware ANN. The mathematical characteristic (the expression having a division and an exponent) of the sigmoidal activation functions makes the direct implementation of sigmoid activation functions resource hungry and hence very difficult [87]. This has therefore led to various reports in the literature, exploring different ways in which the activation functions can be efficiently and effectively implemented on hardware. The various implementations on hardware platforms (FPGA and ASIC) have been tailored either towards improving accuracy, as described in [88, 89]; improved computational speed [90, 91]; or efficient resource usage [92]. One interesting work is proposed in [93] in which an extended stochastic logic implementation of the hyperbolic tangent was implemented on FPGA. They report a reasonable reduction in resource usage but also a reduction in effective precision.

Based on our review, the most common implementation approach of the activation functions on FPGA is the use of Look-Up Table (LUT) which is used to store the values of the activation functions as used in neurons. LUTs require pre-calculation of the activation functions mapping before being loaded into memory. LUT is memory intensive and does not scale well with increasing resolution and precision. Other forms of improvement on the LUT include the work of [94] in which there is a combination of LUT and a piecewise linear approximation. Truncated Taylor series consisting of approximation methods such as the sum of steps, piecewise linear, were discussed in [95–97] along with their associated performances. Specifically, piecewise linear is associated with relatively high mean absolute error and use of multipliers even though this approach is one of the most direct methods in terms of implementation on FPGAs [87, 88]. Polynomial approximation of higher orders, such as Lagrange

polynomial interpolation, Chebyshev polynomial interpolation, and least square method polynomial interpolation, gives a very low error but they are exceptionally resources-hungry due to the number of arithmetic operations performed for each value [88]. Finally, CORDIC implementation is another highly studied approach for hardware implementation of ANN as a result of its primary advantage of using the same resource for several functions but its pipelined, iterative nature could detract from its performance [98, 99]. There have been other variations of CORDIC approach, such as hybrid CORDIC [100], in which CORDIC was combined with LUT and pipeline technology. Authors in [101] and [88] both describe the use of adder and shifter CORDIC implementation and adder, multiplier, and shifter, respectively in which the latter approach increases the accuracy at the expense of increased resource usage.

While computational speed is paramount in a simulator, low resource utilisation and the absence of congestion is of importance for hardware implementation. The goal of this chapter is to present a hardware implementation of the SQLN nonlinear activation function that meets both of these requirements. The following are the notable contributions of this chapter:

- We discussed *two* new resource-efficient ways of implementing the SQLN activation function on hardware.
- Introduction of custom instruction for executing SQLN. This is a single-cycle operation which is advantageous in low-end devices.
- We performed extensive resource utilisation counts using the two methods and the popular lookup table approach.
- We have demonstrated the hardware implementation and performance accuracy of our proposed activation functions using datasets in single-layer feed-forward neural network classification and regression domains.

3.2 Concepts

In this section, we discuss two methods of implementing the SQLN function on hardware. The first method is a novel resource-efficient implementation suitable for large scale parallelism on FPGAs or ASICs. We refer to this method as counter-based SQLN. The second method is based on the use of the standard multiplier. We show that by wrapping a standard multiplier with combinatorial logic, the

SQNL function can be evaluated as quickly as a multiply operation. Hence, the SQNL can also be easily implemented using DSPs on FPGA/CPU based platforms.

3.2.1 Counter-based SQNL

The basis of the counter-based SQNL function implementation is encapsulated in Equation 3.1.

$$f(n) = \frac{1}{N} \sum_{k=1}^N (((n + U(k))_C) - U(k))_M \quad (3.1)$$

The input n , a signed integer, is iteratively added to every element of a predefined sequence $U(k)$ of length N . The adder saturates at predefined upper and lower limits C . The sequence is now subtracted (saturates at M) from this sum and the partial sums are averaged. The underlying morphology of the output is quadratic and can be designed to generate integer SQNL function. Given a binary resolution R , the SQNL activation function generator will use the following parameters:

- The effective range of the input is $M = [-2^{R-1}, 2^{R-1} - 1]$.
- The sequence is an integer non-repeating sequence, $U(k) = \{-U_{MAX}, \dots, U_{MAX}\}$. The length of $U(k)$ is a design decision.
- The saturation levels, $\pm C$ of the adder are defined by $C = U_{MAX} = 2^{R-2}$.

Without saturation, $(n + U(k)) - U(k) = n$ and $f(n) = n$. However, some of the elements of $U(k)$ will result in a saturation and hence $(n + U(k)) - U(k) \neq n$. In Figure 3.1 a), the conceptual impact of Equation 3.1 is shown with $U(k)$ plotted on the x-axis and n , the netsum, along the y-axis. For $n = n_1$, in the absence of saturation, each partial product of $f(n_1)$ will be unaffected and would be plotted as a horizontal line between $[-U_{MAX}, U_{MAX}]$. However, with saturation, some of the positive elements of $U(k)$ will result in saturation resulting in $(n_1 + U(k)) - U(k) \leq n_1$. These partial sums follow the saturation boundary and hence, $f(n_1) < n_1$. The yellow zone shows the consequent saturation loss. The behaviour for negative values, i.e., $n = n_2$ is similar to that for positive inputs.

The average of the mapping corresponds to $f(n)$ and can be easily determined. If the length of $U(k)$ is very large, $f(n)$ can be obtained by a simple computation of $f(n) = \frac{\text{area}}{2U_{MAX}}$. The slope of the saturation profile is -1 and hence, will result in Equation 3.2.

$$f(n) = \frac{1}{2U_{MAX}} \left[n \times 2U_{MAX} - \frac{1}{2} n \times n \right]$$

$$f(n) = n - \frac{n^2}{4U_{MAX}} \quad (3.2)$$

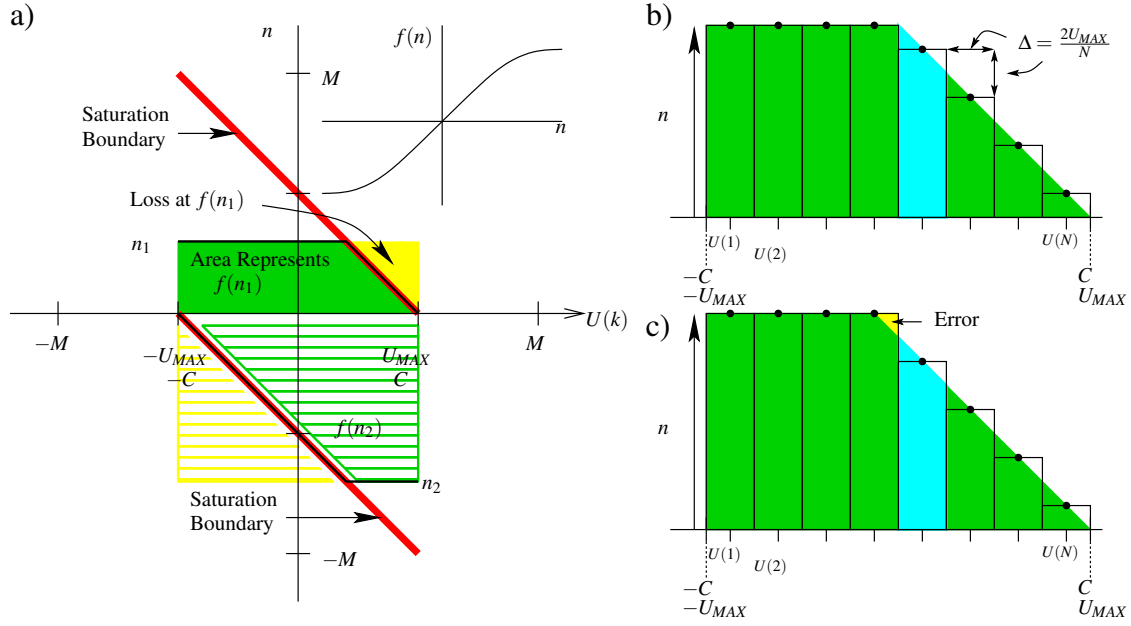


Figure 3.1: SQNL - A Symmetric Activation Function. a) The conceptual impact of Equation 3.1 is shown for symmetric activation function realization. $U(k)$ is plotted on the x-axis and n , the netsum, along the y-axis. The area enclosed in green represents the output mapping $f(n_1)$. The inset shows the form of the mapping for $-M \leq n < M$ b) When $n = \pm i \times \Delta$ i.e. midway between any two contiguous $U(k)$, the $f(n)$ mapping is exact. c) When $n = \pm(\frac{\Delta}{2} + i \times \Delta)$ i.e. exactly equal to any element of $U(k)$, the deviation from the ideal is a maximum.

A large N would adversely impact the throughput of the gate level implementation of Equation 3.1 and hence a pragmatic length of $U(k)$ is preferable. The range $[-U_{MAX} \cdots U_{MAX}]$ is uniformly divided into N sections and $U(k)$ are selected to be the midpoints of each section. The use of midpoints offers the advantage of each section being evenly distributed around zero and thus circumventing the asymmetry of M around zero.

Figure 3.1 b) and c) pictorially depict potential profiles with $N = 8$. If $\Delta = \frac{2U_{MAX}}{N}$, the input in Figure 3.1 b) is selected to be at any integer multiple of Δ , i.e., $n = \pm i \times \Delta$, while in Figure 3.1 c) $n = \pm(\frac{\Delta}{2} + i \times \Delta)$.

The determination of area in the saturation region can be viewed as progressively diminishing rectangles or trapezoids. A gate level implementation would execute $\frac{1}{N} [\sum^N (((n_1 + U(k))_C) - U(k))_M]$ hence, in the following discussion, we consider each section as a rectangle

$$f(n) = \frac{1}{2U_{MAX}} [n \times \Delta + n \times \Delta + \cdots + (n - \Delta) \times \Delta + ((n - \Delta) - \Delta) \times \Delta + ((n - \Delta) - 2\Delta) \times \Delta + \cdots ((n - \Delta) - p\Delta) \times \Delta]$$

$$f(n) = \frac{1}{2U_{MAX}} \left[n.N.\Delta - \sum_{i=1}^p \frac{\Delta^2}{2} - \sum_{i=1}^{p-1} i\Delta^2 \right]$$

Here, $\Delta = \frac{2U_{MAX}}{N}$ and $n - p\Delta = 0$ and hence,

$$f(n) = \frac{1}{2U_{MAX}} \left[n.N.\Delta - \frac{p\Delta^2}{2} - \frac{(p-1)p\Delta^2}{2} \right]$$

$$f(n) = n - \frac{n^2}{4U_{MAX}}$$

Recall that $U_{MAX} = \frac{M}{2}$. Repeating the methodology for negative inputs results in

$$f(n) = \begin{cases} -\frac{M}{2} & : n < -M \\ n + \frac{n^2}{2M} & : -M \leq n < 0 \\ n - \frac{n^2}{2M} & : 0 \leq n \leq M \\ \frac{M}{2} - 1 & : n > M \end{cases} \quad (3.3)$$

Impact of N

N influences the throughput for the hardware implementation of Equation 3.1. Reducing the length of $U(k)$ has a direct influence on the throughput. A consequence of a small and pragmatic value of N results in a linear change in $f(n)$ during the progression from $n = i \times \Delta$ to $n = (i+1) \times \Delta$. This represents a deviation from Equation 3.2. We define deviation as the difference between the ideal SQNL mapping and the mapping produced by a hardware efficient implementation.

Figure 3.1 c) visualises this deviation when $n \neq \pm i \times \Delta$. This deviation is at maximum when $n = \pm(\frac{\Delta}{2} + i \times \Delta)$. The deviation is exactly $\frac{(\Delta/2)^2}{4U_{MAX}}$. With $R = 8$ and $N = 8$, the absolute value of this deviation is 0.25, i.e., less than one bit. Reducing N will imply fewer clocks are required. With $R = 8$, Figure 3.2 plots the theoretical deviations with $N = 4$ and $N = 8$. Figure 3.2 a) only shows the deviations for positive n because negative values match the positive values. The deviation is at maximum when n is equal to any of the values of $U(k)$ and zero midway between any two consecutive $U(k)$. The profile at $N = 4$ clearly shows that increasing N reduces the deviation.

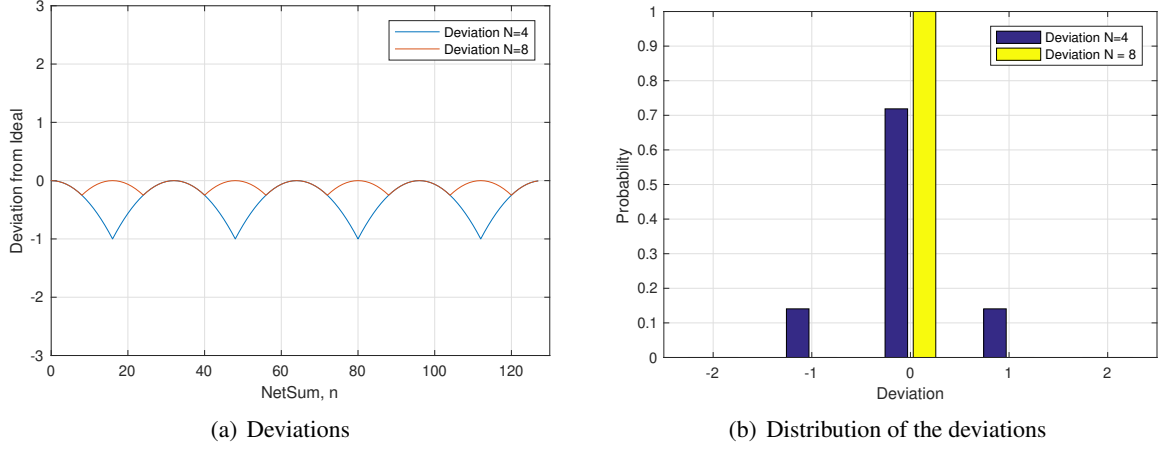


Figure 3.2: Profile of deviation from ideal for $R = 8$ when N is reduced to 4 and 8. a) Plots the deviation for the positive input range for different values of C . b) Plots the frequency distribution of the deviation as a probability

Figure 3.2 b) shows the frequency distribution of the deviations with $N = 4$ and $N = 8$. It shows that the deviations, for both, are at most ± 1 bit away from the ideal mapping. With $N = 8$, the histogram shows that the deviations are within ± 0.5 . Theoretically, there will be a zero-bit error if $N = 8$ when $R = 8$. However, if N is reduced to 4, more than 70% of the range will still exhibit zero-bit errors.

The choice of values for M , C and U_{MAX} are specifically chosen to be aligned to binary boundaries (2^p). This offers gate level implementation efficiencies. The gate level operations of Equation 3.1 are standard building blocks: adders, binary counters, and shift operators.

3.2.2 Multiplier-based SQLN

The simplicity of the SQLN function makes direct implementation with hardware multipliers possible. An examination of Equation 3.3 shows that the square operation can be augmented with combinatorial logic so as to perform a conditional product defined in Equation 3.4.

$$f_s(n) = n \times -|n| \quad (3.4)$$

In Equation 3.4, $f_s(n)$ is referred to as a special square operator. Since hardware multipliers are common in most embedded processors, this square operator could be implemented in the processor and available as a custom instruction. Using this special operator, Equation 3.3 can be compactly written as shown in Equation 3.5, where $b = -|n|$.

$$f(n) = n + \frac{n * b}{2M} \quad (3.5)$$

Most FPGAs host DSP blocks that also integrate a MAC (MUL and ADD) operation. The use of multiplier/DSP blocks with logic that implements Equation 3.4 offers an alternative pathway with a throughput that is exactly that of the multiplier.

3.3 Hardware Implementation

The SQLN activation function has been implemented on an Altera Cyclone V device (5CSXFC6D6F31C6). The Quartus Prime Standard 18.1 Edition and Modelsim 10.1d are the integrated development environment on which all the hardware implementation is performed. We discuss three methods of implementation.

3.3.1 Multi-clock/Counter Solution

Figure 3.3 shows a complete schematic with various bus-widths highlighted. The code is parametrised, and hence the saturation of the adder and subtractor change with the resolution R . The input and output, marked with thick lines, could be a fixed point at a larger bus width than the activation function. The lower Add, Latch, and Right-Shift blocks function as the filter (sum, accumulated, scale). These operate a higher bus width as well. The counter values for an 8-bit ($R = 8$) system with $N = 8$ is $\{\pm 1, \pm 3, \pm 5, \pm 7\} \times 2^{R-2}/N$. This is easily obtained by suitable padding of leading and trailing bits.

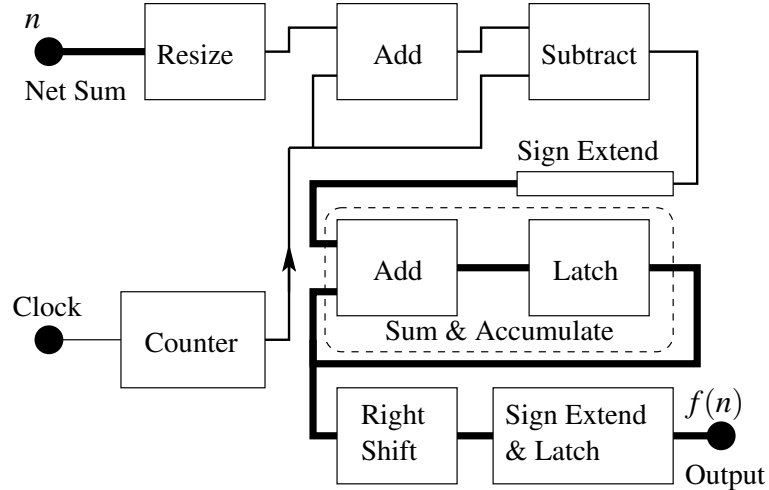


Figure 3.3: Schematic of SQLN Activation Function using Multi-clock Methodology

In Equation 3.3, if M is based on integer powers of 2, the $\frac{1}{2^M}$ is a shift operation. Design decisions R and M are fixed at compilation. Hence, the shift operation consumes zero additional logic because only the upper bits are passed through and will be hardwired by the compiler. Addition/subtraction operations

are combinatorial, and hence extra clock edges are not required.

The values of U_{MAX} and saturation limits of the adder/subtractor can be selected to achieve implementation efficiencies. The optimised values are

- The range of n and $f(n)$ are $[-2^{R-1}, 2^{R-1} - 1]$.
- Adder saturates at -2^{R-2} or $2^{R-2} - 1$.
- The subtractor saturates at -2^{R-1} or $2^{R-1} - 1$.
- The sequence, $U(k)$, consists of N non-repeating values between $[-2^{R-2}, 2^{R-2} - 1]$.

The derivative of Equation 3.3 is given by

$$f'(n) = 1 \pm \frac{n}{M} \quad (3.6)$$

Equation 3.6 requires only a subtractor/adder because the $\frac{1}{M}$ is shift operation that will be hardwired during compilation. Thus the forward and the derivative computations are highly amenable to hardware implementation when compared to most smooth nonlinear functions.

An additional speedup of $2\times$ can be extracted by noting that for all inputs, half of the partial sums due to positive/negative $U(k)$ are constant. Thus, the accumulator could be preloaded with either $n\frac{N}{2}$ or $\pm \sum_{k=0}^{k=N/2} U(k)$. This implies only four clocks are required for $N = 8$.

3.3.2 Single-clock Solution

The fundamental method of generating the SQLN function is based on addition, subtraction, and accumulation; we show that a single cycle is possible which will produce the desired result in one clock cycle. Therefore, the counter sequence can be preloaded, eliminating the need for a counter and the latency associated. The single-cycle solution schematic is shown in Figure 3.4.

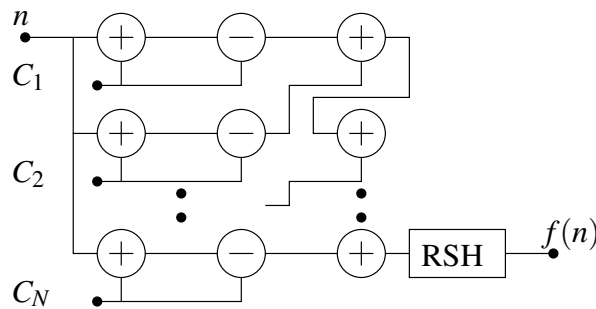


Figure 3.4: Schematic of SQLN Activation Function using Single-clock Methodology. (RSH is Right Shift Operation)

3.3.3 Multiplier Solution

The standard multiplier present in the special square operator introduced in section 3.2.2 is wrapped with combinatorial logic. Equation 3.3 is evaluated as quickly as a multiply operation. The schematic is shown in Figure 3.5.

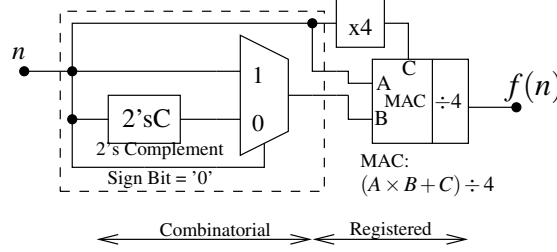


Figure 3.5: Hardware Implementation of SQLN using the custom multiplier (special square operator). The $\div 4$, $\times 4$ operations are right and left shift operations respectively, hence, have no impact on the computational or resource footprint.

We analyse the computational footprint of the SQLN function using the multiplier solution on Altera Cyclone V FPGA. The SQLN mapping with a custom hardware implementation has the potential to achieve the throughput of the MUL(tiply) instruction. Figure 3.5 shows a possible implementation. The schematic shows a square operation achieved using one Multiply and Accumulate (MAC) and additional combinatorial logic that results in the final SQLN. The combinatorial logic is not reliant on any clock and hence, will only impart a propagation delay. It is estimated that this custom instruction will result in an execution time very similar to that of a multiplier, i.e., one to seven clock cycles. Current technology does not allow us to have custom instruction on ARM. Hence, we created custom instructions on the Altera Cyclone V FPGA Nios II for verification. Figure 3.5 was implemented for the SQLN activation function using custom instructions. Experiments show that the SQLN function implemented using proposed custom instruction on Nios II produced the result in one clock cycle.

3.4 Resource Utilisation

We consider both FPGAs and ASIC platforms for resource utilisation analysis. FPGAs use vendor-specific libraries of LUTs, registers, and memory to infer custom designs. The ALM of Cyclone V hosts a fracturable, eight input LUT and four registers. It should be noted that a full six input or subsets of seven and eight input mappings can be achieved. ASICs will certainly offer real-estate efficiencies over FPGAs. We assume that the basic building block is a NAND gate. With this assumption, we estimate the gate usage for the relevant building blocks, shown in Table 3.1. These indicative metrics will be used

in all subsequent comparisons.

Table 3.1: Indicative Gate Usage

Digital Block	Cell/bit	Gates/bit	Total Gate
Single register	-	-	4
1bit full adder	-	-	9
8bit adder	-	-	72
9bit adder	-	-	81
8bit 2's Complement	-	-	80
9bit 2's Complement	-	-	90
2 to 1 mux (1bit)	-	3	3
LUT 8 bits	127	381	2667
LUT 12 bits	2047	6141	67551
Booth 8 bits	-	-	754
Booth 12 bits	-	-	1124

Hence, we present key elements from the resource usage reports of their synthesis tools. We compare the multiplier solution, multi-clock solution, and the use of LUT on the SQLN activation function. Single instances of the three methods - counter (C.SQLN), multiplier (D.SQLN), and LUT (L.SQLN) - have been implemented and their resource usage reported.

Table 3.2: Estimated Counter Based SQLN Gate Usage (GU) with Various N

N	R = 8			R = 12		
	2^2	2^3	2^7	2^2	2^3	2^{11}
Adder	72	72	72	108	108	108
Subtractor	81	81	81	121	121	121
Counter	8	12	28	8	12	44
Sum & Accumulate	90	99	135	126	135	207
Sum & Accumulate Register	40	44	60	56	60	92
Output Register	32	32	32	48	48	48
Multiplexer	48	48	48	72	72	72
Total Gate Usage	371	388	456	539	556	692

3.4.1 Counter-based Implementation

From Figure 3.3, of the three main elements, the adder and the subtractor will only consume logic elements (no registers). The counter-based implementation (C.SQLN) will require $R - 1$ registers to maintain the sum-and-accumulate count, a $2 \times R$ register for the accumulator, and an R bit register to latch the output. The total register usage is estimated as $4 \times R$. On a Cyclone V, with $R = 8$ and $N = 8$, the C.SQLN will require 34 ALMs and 24 registers while with $R = 12$, the usage increases, moderately to 46 ALMs and 31 registers.

With an ASICs perspective, the gate usage of various configurations of the SQLN is shown in Ta-

ble 3.2. The row entitled 'Total Gate Usage' estimates the gates required for each configuration. As expected, the gate usage increases with an increase in N . With an increase in resolution (8 bits \rightarrow 12 bits), the gate usage also increases. Whilst the numbers will not hold any significance until they are compared with other technologies, it is noteworthy that a 16-bit resolution results in a moderate increase in gate count with the C_SQNL.

3.4.2 Multiplier-based Implementation

On FPGAs, Figure 3.5 could be implemented using embedded hard multipliers. However, to estimate the resources on an ASIC, a radix-4 Booth's algorithm was implemented on a Cyclone V using VHDL. In addition to combinatorial operations, this multiplier will require the following: a counter, two shift registers, a sum-and-accumulate block, and an output register. With $R = 8$, we estimate the gate usage of the multiplier to be about 754 gates and the full SQNL function (Figure 3.5) would require 963 gates. On the Cyclone V, an 8 bit SQNL function requires 103 ALMs and 63 registers, which increases to 173 ALMs and 75 registers for a 12 bit function.

3.4.3 Lookup Table-based Implementation

The current FPGAs are LUT centric, and their utilisation metrics need clarity. The ALMs on a Cyclone V FPGA hosts two single bit full adders, four registers, and a fracturable 8-input LUT. The 8-input LUT can completely map a 6-input function but only a subset of 7/8-input functions.

For an 8-bit SQNL function, a full 8-input LUT would map the entire space and hence require eight ALMs (one for every input bit). However, on the Cyclone V, the partial 8-input LUT requires 23 ALMs. With foreknowledge of its LUT mappings, the design was modified to map only half of the input space (127×7) together with a 2's complement. A 6-input LUT would require two ALMs per bit, and the full mapping (128×7) will require 14 LUTs. The 2's complement function will consume another 14 LUTs, and hence the maximum estimated usage is 42 LUTs. However, the subsets of 7/8-input mappings offer optimisations and results in lower usage. The Quartus synthesiser reports 17 ALMs and shown in Table 3.3. A 12-bit SQNL, with only half-space mapping, will require 2048×11 mapping with a 2's complement. A 6-input LUT will require 352 ALMs, while a 7-input LUT will require 352 ALMs. Quartus reports 234 ALMs for the SQNL function.

The estimated gate equivalent footprint of each ALM (adders, registers, and LUT) is 754 gates. Recognising that the FPGA building blocks are general purpose and consequently, their ALM usage will be inflated, we will use Table 3.1 to evaluate a LUT solution. The table lists the gate usage of

Table 3.3: Resource Utilisation Summary of SQLN Implementation using direct (D_SQLN) method, LUT (L_SQLN) method and counter-based (C_SQLN)

Resolution	Methods	Gates (Estimated)	ALM	FF
8 bits	D_SQLN	963	103	63
	L_SQLN	2779	17	8
	C_SQLN	388	34	24
12 bits	D_SQLN	1400	173	75
	L_SQLN	67719	234	12
	C_SQLN	556	46	31

a 128x7 and 2048x11 LUT built from a generic two to one multiplexor cell. Each cell requires three gates. Thus, a read-only 128x7 LUT will require 2667 gates. A complete 8-bit SQLN function will then require 2779 gates, while a 12-bit function will require 67719 gates.

If the LUTs need to be repurposed with a different mapping, RAM cells can be used. The RAM storage will require 128x7 registers (3584 gates) and address decoding. These estimates are validated by figures published by Intel/Altera’s [102] 128x8 memory bank, which consumes about 4620 gates.

3.4.4 Single-clock Solution Implementation

Table 3.4 shows the results of single-clock solution for the SQLN activation function against the LUT and the multiplier-based solution. Single-clock solution consistently outperforms the multiplier-based solution. At higher resolution, single-clock solution out-performs the LUT.

Table 3.4: Resource utilisation of single-clock solution, LUT and multiplier-based solution.

Methods	8 bits		12 bits	
	ALM	FF	ALM	FF
Single-clock	60	-	96	-
LUT	17	8	234	12
Multiplier	103	63	173	75

3.4.5 Discussion

We briefly discuss the resource utilisation of existing methods of implementing Tanh, sigmoid, and other exponential based activation functions presented in the literature. Authors in [103] use the Altera Floating Point Exponent (ALTFP_EXP) mega function. A single ALTFP_EXP will consume 420 ALMs, 681 registers, 12 DSP blocks, and 362 blocks of memory bits. This implementation is the least used in the literature due to its high resource utilisation and latency (17 clock cycles). The authors in [104] propose two solutions for the direct implementation of Tanh which require 21 multipliers, 13 additions, and one

division operation for McLaurin interpolation and eight multipliers, 11 additions, and two division operations for Pade approximation. The LUT design in [105] reported 187 LUTs, 102 slices, and 1348 gates count on Xilinx Virtex-4 chip. The same work reported 108 LUTs, 58 slices, and 1029 gates count for their PWL approximation method. The use of the multiplier-less piecewise linear approximation (PLAN) method is associated with increased approximation error. Authors in [106] reported the PLAN method consuming 167 to 368 logic elements on the Altera Cyclone III device. The use of Taylor series expansion is another popular approximation method for exponential-based activation functions. An eighth order Taylor series for Tanh will require at least seven multipliers, seven floating point divisions, subtraction, and addition operators. Finally, authors in [27] recorded 726 registers, 1687 LUTs, and 34 multipliers on Virtex-5 device using the CORDIC algorithm to implement Tanh.

A real-time inference engine would benefit from parallelism in which several activations are computed rapidly. Here, we compare the resource usage of each of the three methods (C_SQNL, D_SQNL, and L_SQNL) with a focus on ASIC implementations.

With $R = 8$ and $N = 128$, the $128\times$ oversampling is a drawback of the C_SQNL. However, the reduction of oversampling to $4\times$ ($N = 4$) or $8\times$ ($N = 8$) is a practical alternative. Although the $4\times$ or $8\times$ oversampling introduces single bit mapping errors, if the network were to be trained with the piecewise linear function, this error would be irrelevant. In the case studies to follow, the performance of some networks showed no difference, while others exhibited minor variations.

The L_SQNL is a single clock operation while the D_SQNL and C_SQNL require more clocks. The gate/ALM/Register usage varies with R and N and hence, the three methods are compared by normalising their throughputs. With $R = 8$, the Booths algorithm takes four clocks, and therefore four instances of the Booths multiplier will produce four outputs after four clocks. The ratio of normalised gate counts is presented in Table 3.5.

Table 3.5: Ratio of Gates Usage for Normalised Throughput

	R=8		R=12	
	N=4	N=8	N=8	N=16
C_SQNL vs D_SQNL	5.06	2.42	3.78	1.83
C_SQNL vs L_SQNL	3.75	1.79	30.45	14.77
D_SQNL vs L_SQNL	0.74		8.06	

D_SQNL vs L_SQNL

For $R = 8$, a D_SQNL implementation would take 963×4 gates, while an L_SQNL would take 2779 gates. This implies that LUT implementation would offer a better silicon real-estate utilisation. How-

ever, if $R = 12$, Booth's implementation (D_SQNL) would fit $8\times$ more functions.

C_SQNL vs D_SQNL

The C_SQNL implementation with $R = 8, N = 8$, takes four clocks with a single bit error on 20% of the mappings. However, it takes 388 gates compared to the 963 for the D_SQNL. This implies that one could fit 2.48, i.e., an array of C_SQNL would take up less than half the space of the D_SQNL. With $N = 4$, about half the mappings have a single bit error, but the space savings are over 500%. For a 12 bit resolution, the throughput per gate of the C_SQNL is much better than D_SQNL.

C_SQNL vs L_SQNL

The single-cycle performance and its simplicity may make the LUT attractive. However, the throughput per gate of the LUT is inferior to the C_SQNL. With $R = 8, N = 4$, the LUT requires about four times more space, and at $N = 8$, it is almost two times. The LUT does not scale well, and at higher resolutions, either the D_SQNL or the C_SQNL would be a better choice.

3.5 Inference Performance Accuracy

We show the performance accuracy between counter-based SQNL in ModelSim and MATLAB. Simulations and experiments have been performed on the ModelSim simulation engine of the Altera Framework using VHDL as the programming language. The experiments in this section are divided into three categories based on the datasets. Four classification and regression datasets are selected from the UCI repository. The second aspect of this section is based on the experimental dataset generated. The final aspect is the performance verification on a large dataset.

3.5.1 Performance Accuracy on UCI datasets

The performance of the counter-based SQNL function has been evaluated using the datasets available on the UCI repository [107]. A custom SQNL activation function was created (8-bit, $M = \pm 128$, $C = \pm 64$, $U = \pm 64$) to be used with MATLAB's training algorithms. This activation function was created with an Intel/Altera Cyclone V FPGA implementation in mind. Since 18-bit multipliers are available on the Cyclone V, the standard floating-point variables have been scaled such that the input spans ± 100 while the binary outputs are either $+64$ or -64 . Various network architectures were tried. The pattern set was randomly partitioned for training and validation in an 80:20 ratio. The minimum network configuration

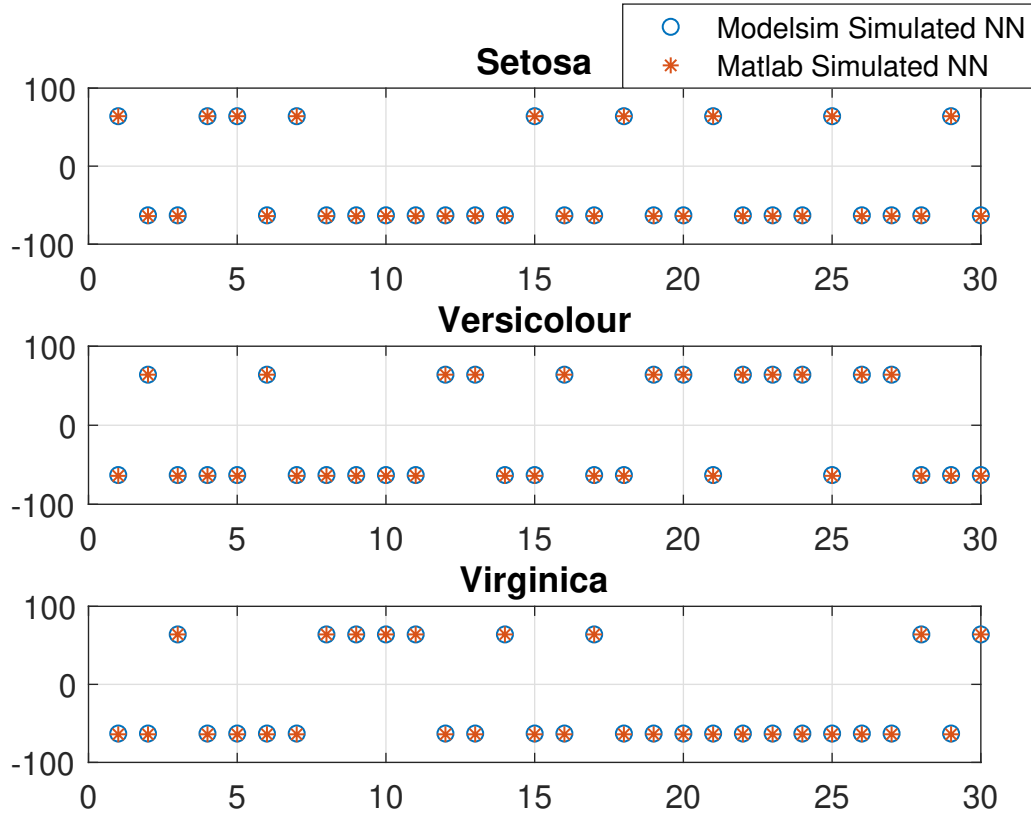


Figure 3.6: Fisher's Iris Classification Dataset - Experimental Result (Modelsim code is synthesisable)

comprised of five neurons in the hidden layer, with both hidden and output nodes having the SQLN activation function. The network was trained using MATLAB while the inferencing was done on Altera Modelsim for Cyclone V FPGA device.

Using the architecture defined above, the networks were simulated using synthesisable VHDL code with the SQLN activation function. The results of the classification datasets (Breast Cancer and Iris [107]) are shown in Figure 3.6 and Figure 3.7, respectively. Figure 3.7 shows an exact match between MATLAB's floating-point (red asterisks) and Modelsim results (blue circles).

Finally, to validate the usability of our function for regression datasets, the following experiments were conducted and reported. A MATLAB regression dataset referred to as "Simple Fit" was also trained using the same approach. A network architecture comprising five hidden SQLN neurons and a linear output neuron gave the best fit. The dataset was divided into the 80:20 ratio. Similarly, we generated a sine wave and trained the network using five hidden SQLN neurons and a linear output neuron. The regression datasets of Figure 3.8 and Figure 3.9 show good agreement between the MATLAB (blue circle) and Modelsim (red asterisks) results for the simple fit datasets and between the MATLAB (red line) and Modelsim (blue circle) results for the sine wave. The difference between the MATLAB and Modelsim simulations can be attributed to fixed-point data representation precision.

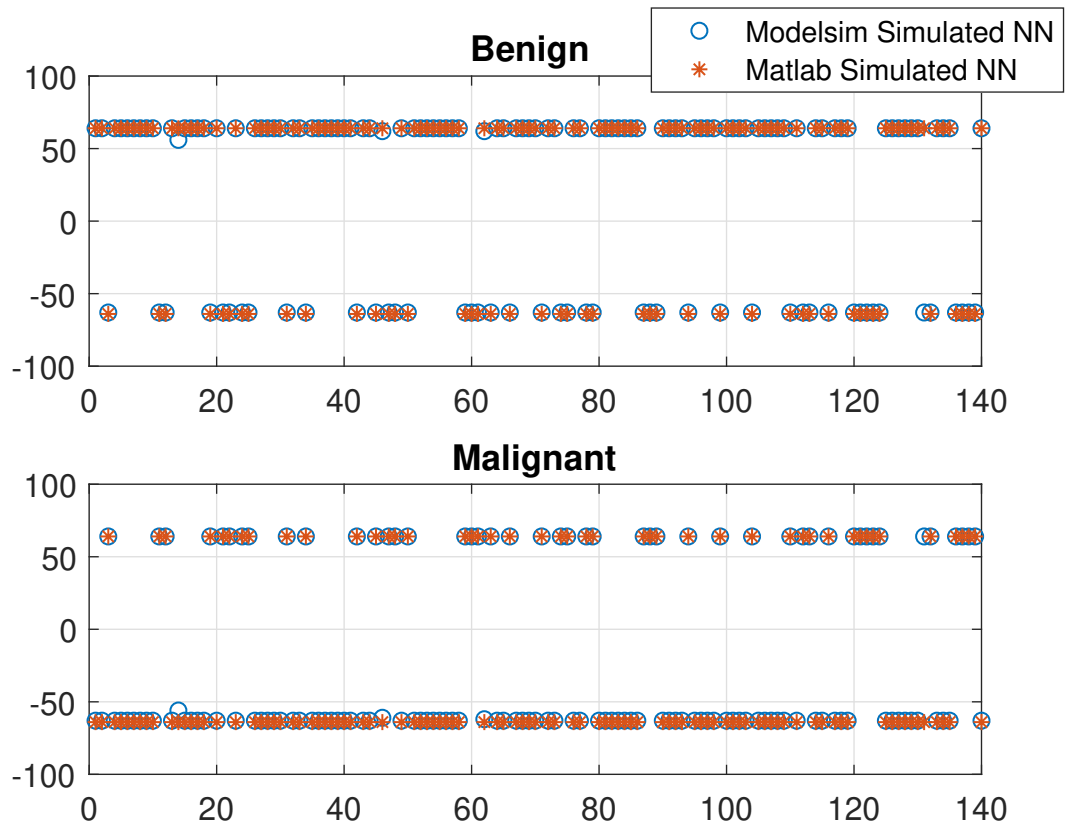


Figure 3.7: Breast Cancer Classification Dataset - Experimental Result (Modelsime code is synthesizable)

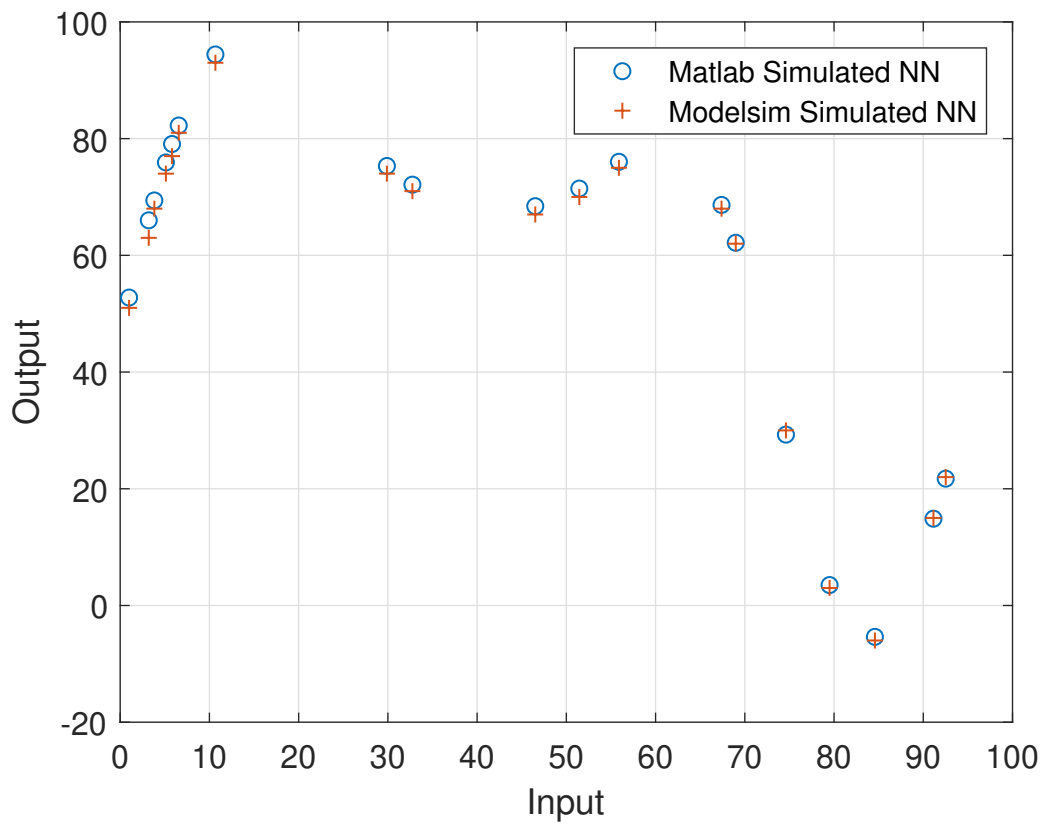


Figure 3.8: Simple Fit Regression Dataset - Experimental Result (Modelsime code is synthesizable)

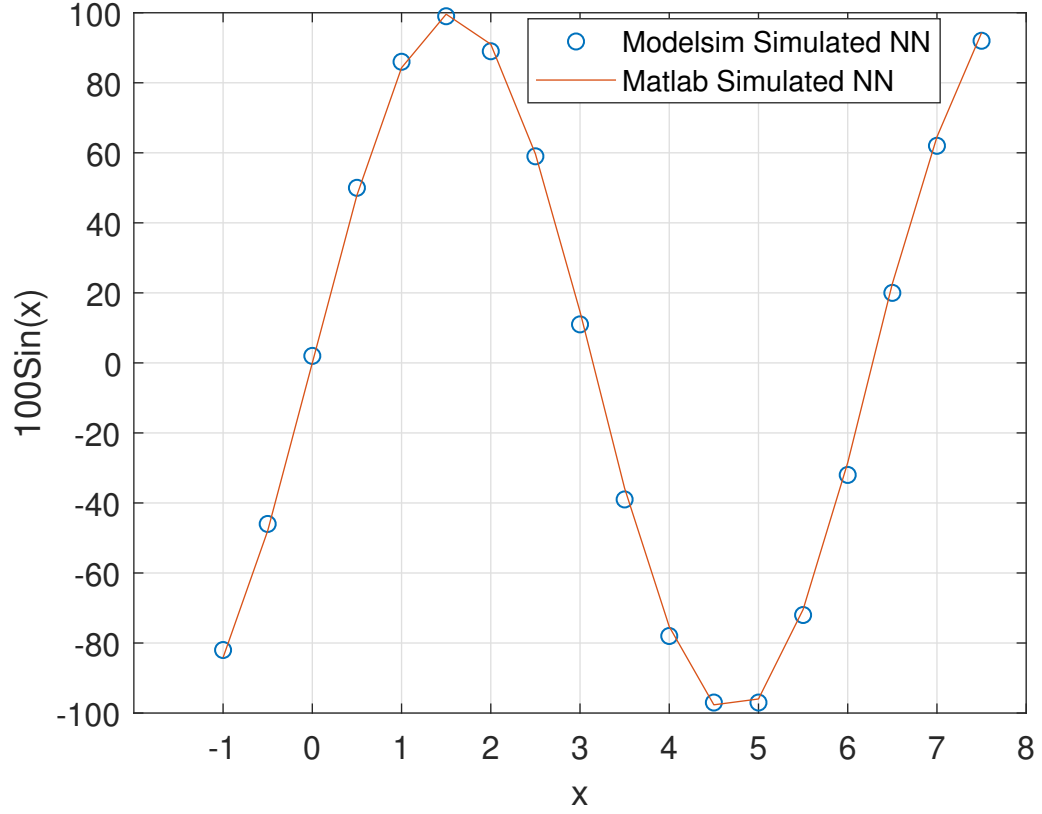


Figure 3.9: Sine Function Regression Dataset - Experimental Result (Modelsim code is synthesisable)

3.5.2 Performance Accuracy on Experimental Rotational Dataset

Figure 3.10 shows a rotation sensing IC (Honeywell APS00B) that uses a magnetoresistive sensor to determine rotation. The magnetic field of a magnet mounted on the shaft/wheel rotates with the shaft. The IC, in a stationary frame, produces two quadrature sinusoids, labelled as Cos and Sin. The lack of purity of the sinusoids as a result of misalignment and non-matching gains could lead to errors in determining the shaft angle. Hence, an ANN could be justified instead of the use of trigonometry. The synthetic data was generated (in MATLAB) by adding 10% noise to the Cos and Sin values. A neural network with 15 hidden nodes was trained with the noisy Cos and Sin signals to infer the rotation. Figure 3.10 shows the responses of a MATLAB and Modelsim simulation. The ANNs use 8-bit SQLN activation functions, and the inferred rotation in MATLAB and VHDL are closely matched except at the ± 180 degree rotations - corresponding to -50 and +50. At these extremes, the noise introduces unresolvable uncertainties when the rotation wraps across ± 180 .

3.5.3 Performance Accuracy on MNIST Dataset

The performance of the counter-based SQLN function has been evaluated using the MNIST dataset. This is to show the usability of the proposed SQLN function on much bigger datasets. The network

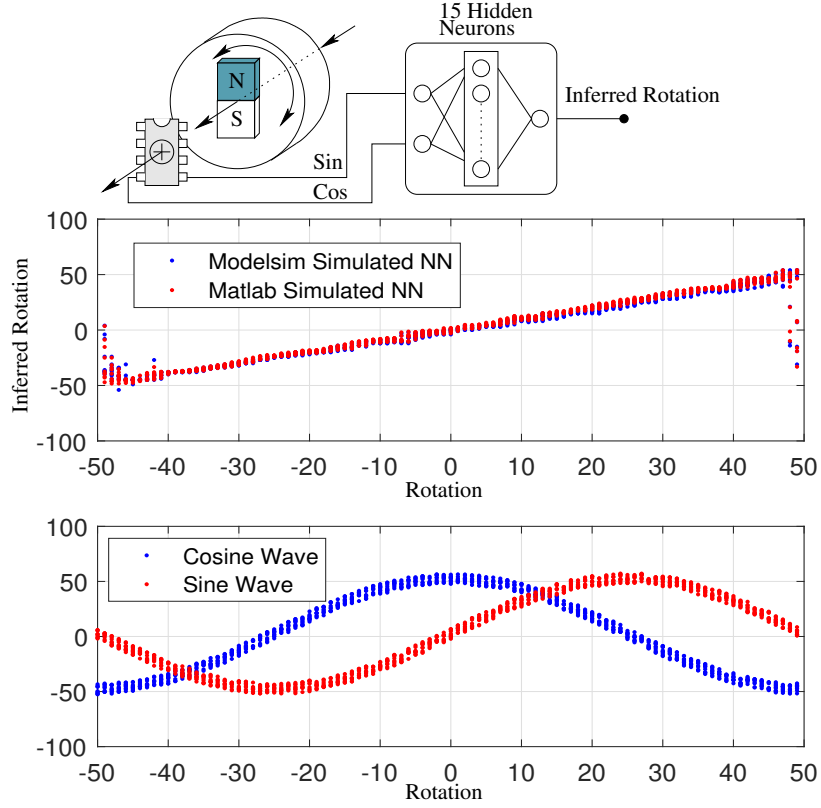


Figure 3.10: Cosine Sine Function Regression Dataset - Experimental Result (Modelsim code is synthesisable)

deployed 100 hidden counter-based SQLN neurons. The output neurons are set to a softmax activation function. With floating point operations, we achieved an accuracy of 96.1% on test dataset. The same accuracy was achieved for its hardware counterpart using a 27-bit fixed point representation (Q13.14, i.e., 13 integer and 14 fractional bits).

All the networks for each of the datasets discussed were tested with different values of $N = 4, 8, 16$. The classification problems showed no difference in accuracy while regression datasets displayed slight variations. This demonstrated that reduced N does not have an adverse effect on the performance accuracy of the datasets.

3.6 Conclusion

We have introduced a novel method for generating computational and resource-efficient SQLN activation functions. The experiments were performed on an Altera Cyclone V FPGA. Although the counter based implementation uses lower resources when compared to other methods, the SQLN function have also been implemented using hardware multiplier. We show that combinatorial logic wrapped around a standard multiplier offers an attractive alternative to the counter implementation. Comparisons with

LUT implementation and a counter-based SQNL function with an 8-bit resolution offers an estimated throughput (per gate) speedup of 1.79x to 3.75x. Similarly, a speedup of 2.48x to 5.06x is estimated with multipliers. Higher throughputs per gate have also been estimated with 12-bit implementations. The single-cycle hardware implementation of the proposed method is also resource-efficient for higher resolution and provides area savings when compared to LUT. The derivative of the digital SQNL function is linear. The implementation of this derivative only requires a combinatorial operation making it highly amenable to on-chip training. Results of computer and VHDL simulations are presented for various datasets, and these show that the counter-based SQNL doesn't cause any form of performance degradation.

Chapter 4

Asymmetric Square-based Activation Functions for Deep Learning

Abstract

In training deep neural networks, the defacto activation function is the asymmetric function. This function is non-saturating, which helps to eliminate the vanishing gradient problem. The importance of the choice of the activation function cannot be overemphasised. Activation functions can influence network training convergence, performance accuracy, and can make training and inference stages computationally expensive. This chapter introduces five new square-based activation functions for embedded ALUs that consume only one instruction cycle with the potential of being resource efficient when constructed in silicon. We show that the proposed functions are computationally efficient when compared with baseline functions (functions containing exponent, logarithm, floating-point division, and square root). We analyse the hidden representations of trained neural network models in an attempt to explain the performance gain, and speedup observed when using square nonlinearities. We show a speedup of 1.9x to 4.3x for the proposed functions using metrics provided by Intel. On an embedded ARM processor, our method achieves a speedup of at least 4.37x. These functions will find particular importance in low-end hardware devices with limited hardware capabilities.

4.1 Introduction

Machine learning is moving towards edge computing evidence from big companies like Google and Intel, creating neural computing sticks. Deep neural networks are characterised by computational com-

plexity and requiring high memory. Thus, such networks are usually trained using powerful hardware. There is an increasing interest in training and deploying neural networks directly on battery-powered devices such as cell phones or other platforms. Such low-power embedded systems are memory and power limited, and in some cases, lack necessary support for floating-point arithmetic [108]. For many applications, the inference phase should possess low power consumption, faster execution, and low computational complexity. The inference phase is mainly made up of the GEMM operation and the activation units. The complexity of the activation function will impact the execution speed of the inference phase and hence have an impact on power and speed.

In literature, there is a wide spectrum of activation functions [3, 64, 109]. Some network architecture dynamically adjusts the activation functions [110, 111], increasing computation and area - especially in hardware. In recent years, developing activation functions that speed up the training step while producing accurate results has attracted greater attention [112, 113]. Designing activation functions that enable fast training of accurate DNN is an active area of research [2]. The complexity of deep neural networks spans across learning parameters ($>$ nine million for ImageNet problem using AlexNet, DenseNet, ResNet models), training algorithm, and activation functions. There is active research in tackling the complexity of deep neural network architectures evidence from DenseNet to SqueezeNet.

Activation functions are one of the important components of any neural network architecture. These functions differ in forms and range but have one important usage - the introduction of nonlinearity. These nonlinearities in neural network architectures are for the sole purpose of making them able to learn complex patterns. The effort to find a suitable activation function has increased [3, 64, 109] due to the need to improve performance accuracy, reduction in complexity, and better convergence speed. However, apart from the ReLU used in deep neural networks, all other activation functions proposed over the years are computationally expensive and also require more parameters (Figure 1.1). Some of the functions, however, have outperformed ReLU based networks at the expense of introducing more complexity to the network architecture. Most of these activation functions are computationally intensive due to the presence of complex mathematical operations such as exponential, division, and square root. The fast approximation of the exponential function described in [114] is associated with limitations. ReLU is by far the simplest of all activations but is characterised by dying neurons and requiring batch normalisation [3].

The inferencing action, mainly influenced by the shape of the activation function, has attracted the attention of hardware implementations in NVIDIA and Google's machine learning engines. NVIDIA [113] introduced a new cuDNN library which provides a series of inference optimisations for GPUs.

The new Long Short Term Memory (LSTM) and Gated Recurrent Unit (GRU) block in the cuDNN library (CuDNNLSTM and CuDNNGRU) are shown to be faster due to the optimisation involved by eliminating the computationally intensive activation functions [113]. Google’s TPU [115] have a hard-wired activation function to eliminate the resource-intensive activation functions.

We extended the square law (Chapter 2) to accommodate for varieties of activation functions. Our proposed functions eliminate the need for an exponent, floating-point division, and logarithm for non-linearity by using a combination of a square/multiplication and shift operations. The following are the contributions of this chapter:

- Introduction of *five* new computationally efficient asymmetric activation functions.
- We show the computational footprint of the proposed square-based functions on Intel CPUs.
- We empirically show the activation over time and the inference time, as well as the speedups, obtain using ARM M3 processor.
- We present the usability of the proposed square-based functions across multiple machine learning models with an increase in inference speed and sometimes performance accuracy.

4.2 Novel Activation Functions

In this section, we propose and discuss five new activation functions for deep learning architectures based on our square law. These functions are referred to collectively as the SQL family due to possessing the square nonlinearity characteristics. We benchmark the SQL family with computationally complex activation functions (functions with exponent, logarithm, trigonometric, and floating-point division) found in the literature [3, 4, 116]. It is important to note that the SQL family functions are *not an approximation*. They are standalone functions that are mathematically unique. The computationally complex activation functions are referred to collectively as baseline family. Figure 4.1 shows individual baseline functions against the square-based functions. Furthermore, we computed the computational footprint of the SQL family and baseline functions on Intel CPUs and ARM M3 processor. We performed experiments showing the activation over time, and inference time.

In chapter 2, we proposed a computationally efficient symmetric activation function referred to as

SQNL based on square law. It is defined in Equation 4.1.

$$f(x) = \begin{cases} 1 & : x > 2.0 \\ x - \frac{x^2}{4} & : 0 \leq x \leq 2.0 \\ x + \frac{x^2}{4} & : -2.0 \leq x < 0 \\ -1 & : x < -2.0 \end{cases} \quad (4.1)$$

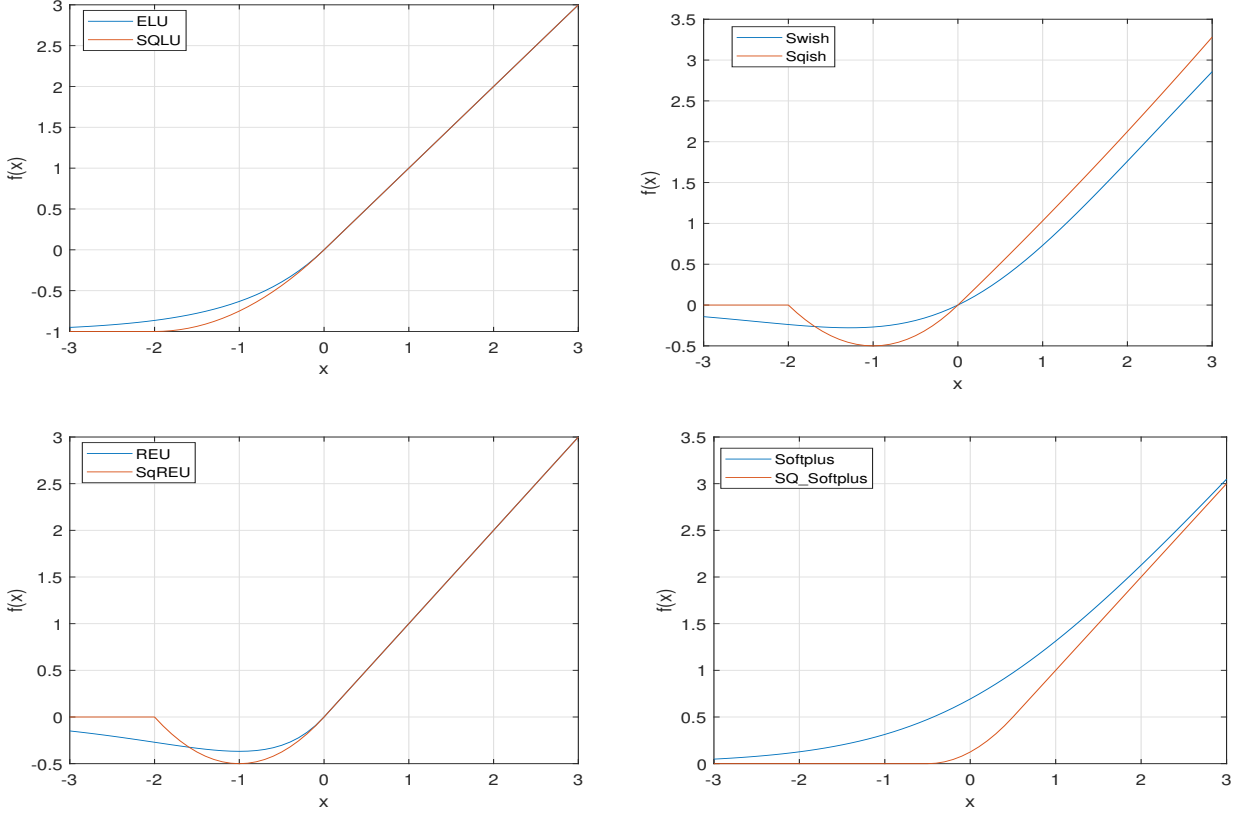


Figure 4.1: Activation Functions: All the proposed square-based against their corresponding complex-based functions

The square law can be adapted to produce other forms of activation functions. We propose a series of smooth asymmetric activation functions useful in the hidden layers. In addition, we present the replacement for the normalising softmax function which is used in the output layer.

4.2.1 Smooth Square-based Asymmetric Activation Functions

Several variants of ReLUs have been proposed in the literature and shown to outperform ReLU. Of particular interest are the smooth asymmetric functions [3, 4, 116]. Asymmetric functions with negative values are of importance because they are responsible for pushing the mean activations closer to zero [3]. This is responsible for faster training, sometimes the elimination of batch normalisation, and leads to

increased accuracies. However, a particular similarity to all these variants of ReLU is the introduction of computationally expensive mathematical operators like the exponent term, logarithm, and floating-point division. We will discuss each of these smooth variants of ReLU functions, their advantages, and their replacement based on our square law.

Square Linear Units (SQLU)

Using the square nonlinearity, we define a new nonlinear function called Square Linear Unit (SQLU). The SQLU function is defined in Equation 4.2.

$$f(x) = \begin{cases} x & : x > 0 \\ x + \frac{x^2}{4} & : -2.0 \leq x \leq 0 \\ -1 & : x < -2.0 \end{cases} \quad (4.2)$$

Equation 4.2 can be compactly written as $f(x) = \max\{0, x\} + \min\{0, x + x^2/4\}$ after clipping the negative input to -2 . SQLU is morphologically similar to the Exponential Linear Units (ELU) defined in [3] but computationally simple. Figure 4.1 shows the similarity in the shape of ELU and SQLU. SQLU is not an approximation of ELU but it is derived from the square law defined in Equation 4.1. The ELU has been shown to achieve higher classification accuracies with a learning/convergence speed up as compared to ReLU. This is as a result of its ability to push mean unit activations closer to zero. However, the function is computationally expensive due to the presence of the exponent term. The authors reported that ELU networks are about 5% slower on ImageNet datasets than ReLU networks, further confirming the complexity of ELU functions.

Batch Normalisation and ELU/SQLU

We recognise that ReLU is one of the most popular functions for DNN architectures. It was reported that it could lead to dead neurons and bias shift [3]. According to the analysis provided in [3, 117], they have reduced the undesired bias shift effect without the natural gradients, either by centring the activation of incoming units at zero or by using activation functions with negative values. ReLU networks have been shown in literature [3, 118] to benefit from batch normalisation. The requirement of batch normalisation [118] to reduce the undesired bias shift is computationally expensive ($x_{bn} = \gamma(\frac{x-\mu}{\sigma}) + \beta$, where μ and σ are the mean and the standard deviation of each mini-batch, γ and β are the learnable parameters responsible for improving the representation power of the model). Per unit activation: two multiplications, square root, one division, additions, and subtractions are required for batch normalisation. These

additional computations effectively negate the simplicity of the ReLU when batch normalisation is required. It has been shown [3] that ELU networks do not benefit from batch normalisation. The SQLU morphology is characteristically similar to the computationally expensive ELU activation function. The characteristics of the SQLU function will be able to reduce the bias shift with minimum computational complexity. Finally, it will be shown in Chapter 5 that SQLU can be implemented as a single cycle function, and hence, the ReLU and SQLU will exhibit the same computational load.

Square Swish (Sqish)

Using the square nonlinearity, we define a new nonlinear function called Square Swish (Sqish) with a morphology similar to Swish and GELU but computationally simple. Swish and GELU proposed in [4, 116] are non-convex, non-monotonic functions which are not linear in the positive domain by exhibiting curvature at all points. Swish is defined mathematically as $f(x) = x \times \sigma(x)$ while GELU is approximated as $f(x) = 0.5x(1 + \tanh[\sqrt{2/\pi}i(x + 0.044715x^3)])$. The Sqish function is defined in Equation 4.3.

$$f(x) = \begin{cases} x + \frac{x^2}{2^5} & : x > 0 \\ x + \frac{x^2}{2} & : -2.0 \leq x < 0 \\ 0 & : x < -2.0 \end{cases} \quad (4.3)$$

Equation 4.3 can be compactly written as $f(x) = \max\{0, x + x^2/2^5\} + \min\{0, x + x^2/2\}$. Equation 4.3 is not an approximation of Swish and GELU functions.

Square REU (Sqreu)

Authors in [119] proposed an activation function which aimed to take advantage of ReLU and Swish functions. This function is called REU and defined as $f(x) = \max\{0, x\} + \min\{0, x \times \exp(x)\}$. This function shares the positive region characteristics of ReLU and the negative region characteristics of the Swish function. We defined a new activation function based on our square law. This function defined in Equation 4.4 is morphologically similar to REU and we refer to this function as SqREU.

$$f(x) = \begin{cases} x & : x > 0 \\ x + \frac{x^2}{2} & : -2.0 \leq x \leq 0 \\ 0 & : x < -2.0 \end{cases} \quad (4.4)$$

Equation 4.4 can be compactly written as $f(x) = \max\{0, x\} + \min\{0, (x + x^2/2)\}$.

Square Softplus (SQ_Softplus)

The square-based softplus (SQ_Softplus) is introduced to replace the Softplus function $f(x) = \log(1 + \exp(x))$. The proposed function is defined in Equation 4.5.

$$f(x) = \begin{cases} x & : x > 0.5 \\ \frac{(x+0.5)^2}{2} & : -0.5 \leq x \leq 0.5 \\ 0 & : x < -0.5 \end{cases} \quad (4.5)$$

Equation 4.5 can be compactly written as $f(x) = \max\{0, x\} + \min\{0, (x + 0.5)^2/2\}$.

4.2.2 Square-based Output Layer Activation Function

The choice of activation function for the last layer of a neural network architecture depends on the task. For regression problems, the common activation of choice is the linear function. Sigmoid and Softmax functions are generally used in binary and multi-class classification applications, respectively. Sigmoid is also used as a gating function in recurrent neural networks. Sigmoid and softmax are computationally expensive due to the presence of the exponential function. We present an alternative to softmax function using our square law method. The alternative to sigmoid, called Log_SQNL, has already been discussed in Chapter 2.

The square-based softmax (SQMAX) is introduced to replace the computationally expensive softmax function $f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^M e^{x_j}}$. The softmax function takes a vector of M dimensions and returns a probability distribution also of M dimensions. Each element of the output is in the range $(0, 1)$ and the sum of the elements of M is 1.0. The proposed function is defined in Equation 4.6. Where $c = 4$ is a predefined offset.

$$f(x_i) = \frac{(x_i + c)^2}{\sum_{j=1}^M (x_j + c)^2} \quad (4.6)$$

4.3 Analysis

4.3.1 Computational Footprint on Intel CPUs

Intel [66] provides detailed metrics on the computation time of various math operations. Vectors of 1000 elements with randomly generated numbers were used, and an average was taken to obtain the results. An extensive comparison of the CPU performance of the required mathematical functions for activation functions under test is available in Table 2.1 (Chapter 2). Table 4.1 shows the speedups

Table 4.1: Speedups: CPU Performance of asymmetric activation functions. Run on Intel Xeon with Vector Function Data.

Single Precision	E5- 2699 (Haswell)	E5-2699 (Broadwell)	Gold 6148 (Skylake)
ELU/SQLU	2.3×	2.5×	2.3×
ISRLU /SQLU	3.5×	3.5×	2.9×
Swish/Sqish	4.1×	4.3×	4.2×
REU/Sqreu	2.3×	2.5×	2.3×
Softplus /SQ_Softplus	2.3×	1.9×	2.1×

reported for square-based activation function when compared with the baseline benchmarks. It can be seen that square-based activation functions show a significant speedup over the baseline functions; this will mainly be useful in CPU based inference engines.

4.3.2 Computational Footprint on the ARM M3 Processor

Table 4.2 shows the full computational time in milliseconds of the proposed asymmetric activation functions as well as their respective derivatives against their respective baselines. Table 4.3 shows the speedup achieved by the square-based functions relative to the benchmarks. The SQLU is $8.56\times$ faster with double precision and $19.2\times$ faster with Q16.16 fixed-point operations (Table 4.3). The Softplus mapping is the most expensive in double or Q16.16 precision arithmetic. The proposed SQ_Softplus offers very significant speeds ups of $39.2\times$ and a substantial $169.8\times$ for double and Q16.16 precision, respectively.

Table 4.2: Computational time of ELU, SQLU, Softplus (Softp) and SQ_Softplus (SQ_Softp) , SqREU, REU, Swish and Sqish functions and their derivatives using ARM M3 processor. This is an average of 1000 calculations.

Function	Forward (ms)	Derivative (ms)
ELU Floating Point	131.14	125.85
ELU Fixed Point	34.84	12.66
SQLU Floating Point	15.33	34.42
SQLU Fixed Point	1.81	0.64
Softplus Floating Point	252.83	154.58
Softplus Fixed Point	109.49	5.27
SQ_Softplus Floating Point	6.44	6.53
SQ_Softplus Fixed Point	0.64	0.64
Swish Floating Point	168.17	178.91
Swish Fixed Point	136.10	143.57
Sqish Floating Point	18.60	11.68
Sqish Fixed Point	8.95	2.74
REU Floating Point	71.83	72.22
REU Fixed Point	66.08	65.82
SqREU Floating Point	12.54	8.29
SqREU Fixed Point	5.32	1.69

Table 4.3: Speed ups: Computational time of asymmetric functions and their derivatives using ARM M3 processor. This is an average of 1000 calculations.

Function	Forward	Derivative
ELU/SQLU Floating Point	$8.55 \times$	$3.66 \times$
ELU/SQLU Fixed Point	$19.22 \times$	$19.83 \times$
ISRLU/SQLU Floating Point	$8.55 \times$	$3.66 \times$
ISRLU/SQLU Fixed Point	$19.22 \times$	$19.83 \times$
Softplus/SQ_Softplus Floating Point	$39.23 \times$	$23.68 \times$
Softplus/SQ_Softplus Fixed Point	$169.79 \times$	$8.18 \times$
Swish/Sqish Floating Point	$9.04 \times$	$15.32 \times$
Swish/Sqish Fixed Point	$15.21 \times$	$52.39 \times$
REU/SqREU Floating Point	$5.73 \times$	$8.71 \times$
REU/SqREU Fixed Point	$12.42 \times$	$38.95 \times$

4.4 Experimental Results

To empirically evaluate the proposed method, we investigated popular machine learning models with our activation functions. In terms of performance, the square-based functions are not inferior to the baseline functions, and sometimes they are better. We made no changes to predefined architectures and hyperparameters. All the predefined architectures were designed with the ReLUs activation function. We replace the ReLU activation function with different activation functions and show good comparability between the activation functions.

4.4.1 Activation Over Time and Inference Time

We trained an eight hidden layer deep neural network on MNIST dataset. We followed the same settings as defined in [3]. Each layer consisted of 128 neurons; we used the stochastic gradient descent with mini-batches of size 64 and learning rate of 0.01. Table 4.4, shows the accuracy and computational time. This further supports the claim that the computationally efficient SQLU can successfully replace the computationally expensive ELU.

Table 4.4: MNIST preliminary analysis showing: training time, inference time in seconds, and percentage performance accuracy. Here SQLU consistently performs better than ELU.(Average of five runs)

Function	Training Time (s)	Inference Time (s)	Accuracy (%)
ReLU	1146.29	108.12	99.03
ELU	1324.07	114.88	99.27
SQLU	1302.03	110.15	99.71

Furthermore, we present the activations over time of the first and the penultimate layer in Figure 4.2 and 4.3 respectively for SQLU and ELU functions. Figure 4.2 and 4.3 show similar values of activations over time.

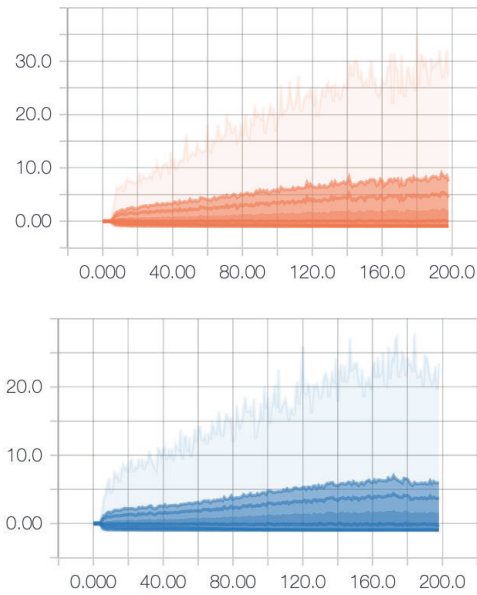


Figure 4.2: Top: First hidden layer activations during training with ELU Function, Bottom: First hidden layer activations during training with SQLU Function.

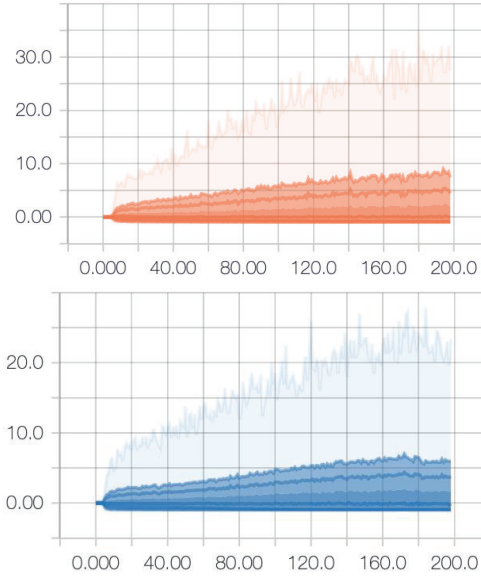


Figure 4.3: Top: Last hidden layer activations during training with ELU Function, Bottom: Last hidden layer activations during training with SQLU Function.

4.4.2 Experiments using SQLU

We investigate the learning behavior of SQLUs on unsupervised and supervised learning. We perform the analysis on the following architectures: deep autoencoders, deep neural networks, and simple deep convolution neural networks. The performance accuracy of SQLU was also explored on CIFAR-10, CIFAR-100 [120], SVHN [121], and Tiny ImageNet [122] datasets using various CNN architectures.

Learning Behaviour of SQLU: Train a simple deep MLP on the MNIST

We first want to verify the usability of SQLU in deep multilayer perceptron networks. Fully connected deep neural networks with SQLUs, ELUs ($\alpha = 1.0$), ReLUs, and LReLUs ($\alpha = 0.1$) were trained on the MNIST digit classification dataset. Each network has eight hidden layers of 128 neurons, and was trained for 300 epochs by stochastic gradient descent with learning rate 0.01 and mini-batches of size 64 as described in [3]. The weights have been initialised according to [9]. The training error and the test error for ELU and SQLU are the same and perform better than other functions as shown in Figure 4.4.

Learning Behaviour of SQLU: Train a simple deep CNN on the CIFAR10 small images dataset

To evaluate SQLU networks on CNN models, we follow the simple CNN model defined in Keras [123]. KerasNet: a convolutional neural network included in the Keras framework. It is made up of four convolutional layers and two fully connected layers, and it employs both max pooling and dropout.

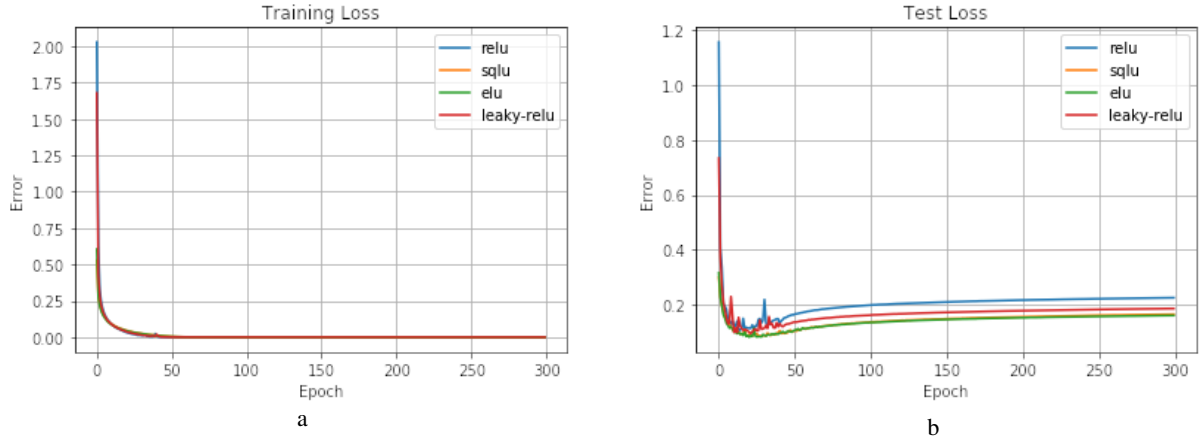


Figure 4.4: SQLU networks evaluated on MNIST. a) Training set cross entropy loss for different activation functions. b) Validation set cross entropy loss for different activation functions.(Best viewed in colour)

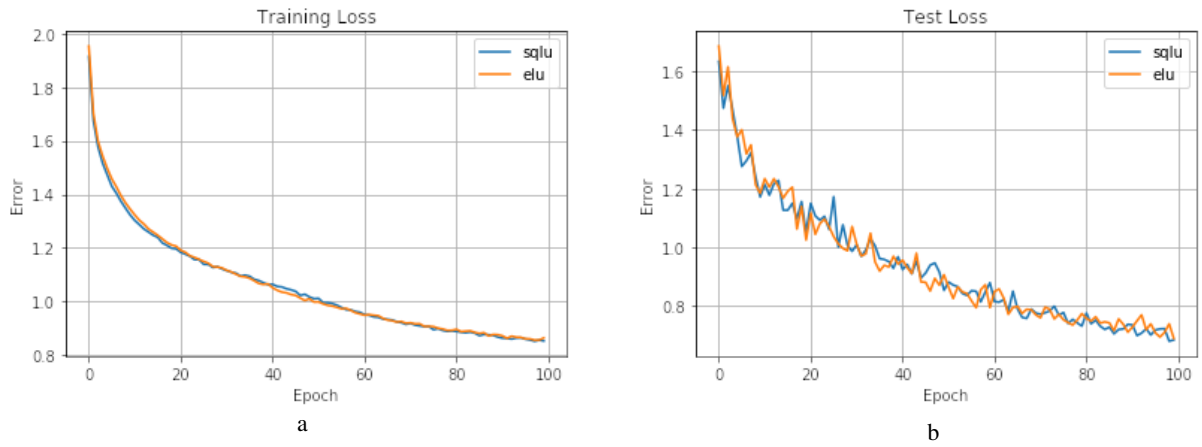


Figure 4.5: SQLU and ELU networks evaluated on CIFAR10 with Data Augmentation to show that SQLU is not inferior to ELU. a) Training set cross entropy loss for SQLU and ELU functions. b) Validation set cross entropy loss for SQLU and ELU functions. (Best viewed in colour)

Figure 4.5 shows that SQLU and ELU achieve the same results. Therefore, the computationally efficient SQLU can successfully replace the ELU.

Learning Behaviour of SQLU: Autoencoder Learning

To evaluate SQLU networks at unsupervised settings, we followed [124, 125] and trained a deep autoencoder on the MNIST dataset. The encoder part consisted of four fully connected hidden layers with sizes 1000, 500, 250, and 30, respectively. The decoder part has a similar connection to the encoder. For learning, we applied stochastic gradient descent using the fixed learning rate of 0.01 with mini-batches of 64 samples for 500 epochs. Figure 4.6 shows that SQLU and ELU outperform the competing activation functions in terms of training / test set reconstruction error.

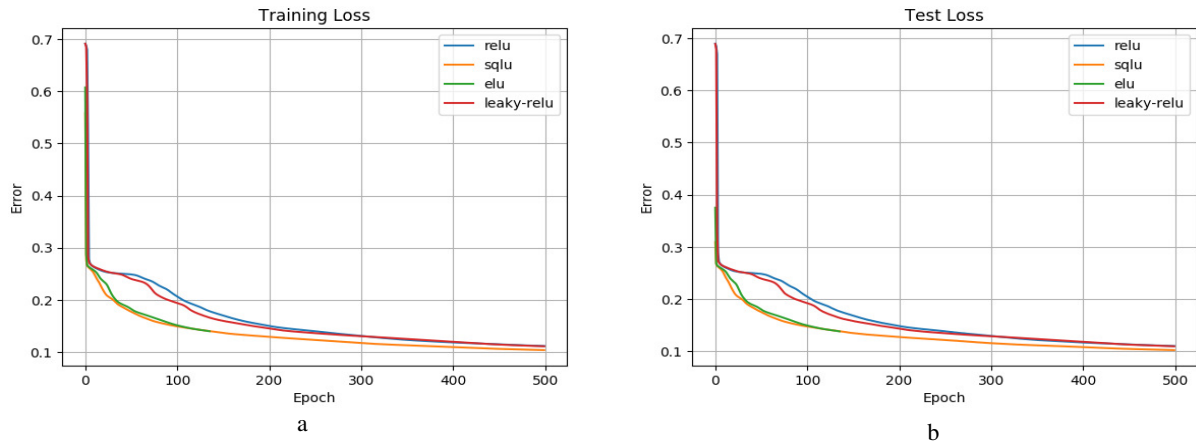


Figure 4.6: Autoencoder training on MNIST: a) Training reconstruction error using different activation functions. b) Test reconstruction error using different activation functions.(Best viewed in colour)

Table 4.5: Performance Accuracy on CIFAR - 10 using ReLU, ELU and SQLU activation functions on already defined architecture. We used the ResNet20 (20 layer ResNet) Version 1 [9]. The results show that SQLU is not inferior to ELU and hence, can replace ELU whenever the computational time and resource-efficient Inference block is essential. * ELU diverges. (Results are average of five runs, and an estimated mean of accuracy with the confidence of 95% is recorded).

Network	Activation Functions		
	ReLU (%)	ELU (%)	SQLU (%)
ResNet20	91.77 \pm 0.15	91.40 \pm 0.14	91.77 \pm 0.2
VGG-19	93.4 \pm 0.31	93.3 \pm 0.34	93.2 \pm 0.46
NIN	88.25 \pm 0.36	-*	88.88 \pm 0.33

SQLU Performance Accuracy: Deep supervised Learning

The closest form to the SQLU is the ELU [3]. The SQLU and ELU only differ in the negative zone. SQLU is characterised by larger values of nonlinearity at the linear regime when compared to ELU. We use the same network architecture provided in [9, 126, 127], and only replace the activation functions from ReLU to SQLU and ELU. This code is available at [128]. We applied the architectures to CIFAR 10, CIFAR 100, SVHN (descriptions available at [129]), and Tiny ImageNet Datasets. The results are presented in Tables 4.5, 4.6, 4.7 and 4.8.

Table 4.6: Performance Accuracy on CIFAR - 100 using ReLU, ELU and SQLU activation functions on already defined architecture. We used the ResNet20 (20 layer ResNet) Version 2 [10]. * ELU diverges. (Results are average of five runs, and an estimated mean of accuracy with the confidence of 95% is recorded).

Network	Activation Functions		
	ReLU (%)	ELU (%)	SQLU (%)
ResNet20	71.6 \pm 0.36	71.8 \pm 1.3	71.8 \pm 0.01
VGG-19	70.8 \pm 0.28	70.5 \pm 0.33	70.6 \pm 0.47
NIN	61.91 \pm 0.32	-*	63.01 \pm 0.31

Table 4.7: Performance Accuracy on SVHN using ReLU, ELU and SQLU activation functions on already defined architecture. We used the Convnet ([11]). The results show that SQLU is not inferior to ELU and hence, can replace ELU whenever the computational time and resource-efficient Inference block is essential. (Results are average of five runs, and an estimated mean of accuracy with the confidence of 95% is recorded).

Network	Activation Functions		
	ReLU (%)	ELU (%)	SQLU (%)
Convnet	95.72 ± 0.19	95.61 ± 0.41	95.67 ± 0.2
VGG-19	95.53 ± 0.06	95.33 ± 0.08	95.30 ± 0.03
NIN	94.28 ± 0.09	94.39 ± 0.06	94.41 ± 0.03

Table 4.8: Performance Accuracy on Tiny ImageNet using ReLU, ELU and SQLU activation functions on already defined architecture. We used the DenseNet and model parameters defined in [12]. We use the ResNet50 (50 layer ResNet) [10] and model parameters defined in [13].

Network	Activation Functions		
	ReLU (%)	ELU (%)	SQLU (%)
DenseNet	62.73	62.77	62.75
ResNet50	51.61	51.11	51.60

Our results show that applying the proposed activation function sometimes leads to the same/better performance than that of the reference model. It is important to note that we are interested in showing the usability of SQLU by using the same model set up by previous work and only replace the activation function. The speedup was not recorded because other factors (such as dropout, maxpooling and so on) contribute to the training and inference time. Hence, reliable comparisons of time taken to train are difficult. However, Table 4.1 shows that computing SQLU layer is $2.3\times$ faster than the ELU layer on Intel CPU. It is of great interest to see that SQLU in most cases outperforms ELU, even though they are morphologically similar. However, SQLU and ELU differ significantly in the negative region and the derivative function.

4.4.3 Experiments using Sqish for Very Deep Networks

We compare Sqish to ReLU and Swish activation functions on the CIFAR-10 and CIFAR-100 datasets [120]. We follow the experimental settings recorded in [116] and train Sqish activation function on the following very deep architectures: ResNet-164 [10], Wide ResNet 28-10 (WRN) [14], and DenseNet 100-12 [130]. Table 4.9 and 4.10 shows the performance accuracy on CIFAR-10 and CIFAR-100 datasets, respectively.

Table 4.9: Sqish: Performance Accuracy on CIFAR - 10 using activation functions on already defined architecture. The results show that Sqish is not inferior to swish and hence, can replace swish whenever the computational time and resource-efficient Inference block is essential. (Results are average of five runs, and an estimated mean of accuracy with the confidence of 95% is recorded).

Model	ResNet-164	WRN 28-10	DenseNet 100-12
ReLU	92.24 \pm 0.10	95.57 \pm 0.13	93.65 \pm 0.52
Swish	91.64 \pm 0.13	95.66 \pm 0.10	94.12 \pm 0.38
Sqish	92.69 \pm 0.23	95.68 \pm 0.10	94.01 \pm 0.58

Table 4.10: Sqish: Performance Accuracy on CIFAR - 100 using activation functions on already defined architecture. The results show that Sqish is not inferior to swish and hence can replace swish whenever the computational time and resource-efficient Inference block is essential. (Results are average of 5 runs, and an estimated mean of accuracy with the confidence of 95% is recorded)

Model	WRN 28-10	DenseNet 100-12
ReLU	78.9 \pm 0.50	73.39 \pm 0.99
Swish	79.76 \pm 0.82	74.63 \pm 2.07
Sqish	79.50 \pm 0.26	74.03 \pm 1.82

The results in Tables 4.9 and 4.10 show that Sqish is not inferior to Swish in performance accuracy. Swish and our square based equivalent Sqish consistently match or outperform ReLU on every model for CIFAR-10 and CIFAR-100.

4.4.4 Experiments using SqREU

We replace the conventional computationally intensive REU function with the simple SqREU. We carried out a similar experiment as described in [119] and show in Table 4.11 that the SqREU consistently outperforms ReLU and REU functions.

Table 4.11: SqREU: Performance Accuracy on Fashion MNIST (FMNIST), CIFAR-10 and CIFAR-100 using activation functions on already defined architecture. The results show that SqREU is not inferior to REU and hence, can replace REU whenever the computational time and resource-efficient Inference block is essential. (Results are average of five runs, and an estimated mean of accuracy with the confidence of 95% is recorded).

	LeNet-5 FMNIST	ResNet-110	
		CIFAR-10	CIFAR-100
ReLU	90.44 \pm 0.15	92.27 \pm 0.22	69.29 \pm 0.93
REU	90.57 \pm 0.10	92.72 \pm 0.20	70.26 \pm 0.43
SqREU	90.61 \pm 0.19	92.86 \pm 0.21	71.03 \pm 0.39

4.4.5 Experiments using SQ_Softplus for Restricted Boltzmann Machine

In this experiment, we replaced the softplus function ($f(x) = \log(1 + \exp(x))$) commonly used in RBM and sometimes in DNN architecture with our simple square-based function referred to as SQ_softplus,

defined in Equation 4.5. The model description and architecture used is defined and available at [131]. The result shows that SQ_softplus function and softplus achieve the same results in terms of accuracy, as shown in Figure 4.7. Both softplus and SQ_softplus networks show a similar training loss and test error.

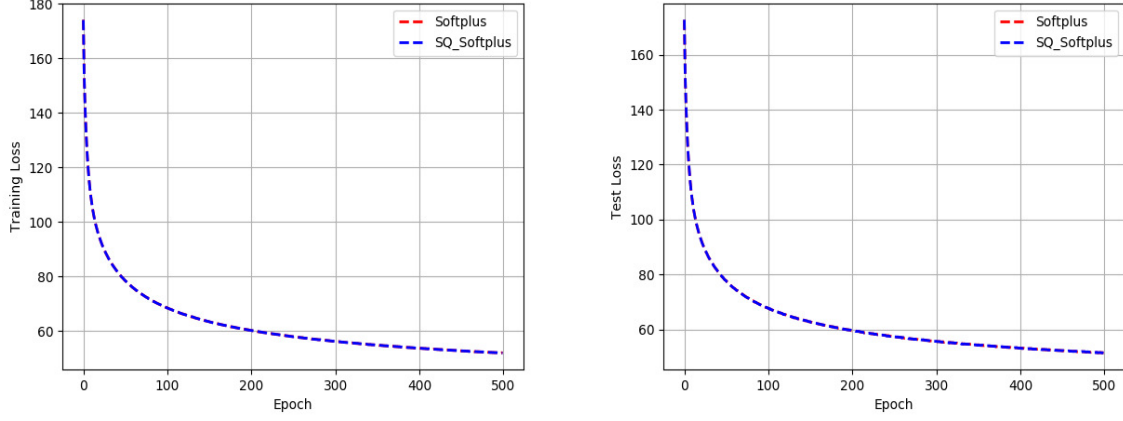


Figure 4.7: Comparison of Conventional softplus and SQ_softplus: Left: Training loss, Right: Test error.

4.4.6 Experiments using SQMAX

For this experiment, the softmax and SQMAX gives comparable performance accuracy across all datasets and architectures as shown in Table 4.12.

Table 4.12: Performance Accuracy on SVHN CIFAR-10 and CIFAR-100 using softmax and SQMAX as the output layer activation function. The hidden layer function is left as ReLU. We changed the flatten layer to the Global Averaging Pooling for the VGG-19 architecture. The results show that SQMAX is not inferior to Softmax and hence, can replace softmax whenever the computational time and resource-efficient inference block is essential.

	NIN		VGG-19
	SVHN	CIFAR-100	Tiny ImageNet
Softmax	95.10	61.91	62.47
SQMAX	95.10	61.37	62.46

The difference between the softmax and SQMAX function is the replacement of the exponent term with a square term. Therefore computationally, on an Intel CPU processor, a square operator is $4\times$ faster than an exponent term. Table 4.13 shows the computational time on an ARM M3 processor of the exponent and square terms. The floating-point square operator is $25.69\times$ faster than the exponent operator. The fixed point precision square operator achieves a speedup of $20.46\times$.

Table 4.13: Computational time of exponential and square operators using ARM M3 processor.

Precision	Exponential Time (ms)	Square Time (ms)
Floating point	128.98	5.41
Fixed Point	116.03	5.67

4.5 Conclusion

The matrix multiply unit, accumulator unit, and activation unit makes up a typical inference engine. In this chapter, we have proposed replacing the computationally complex activation unit with our square-based activation unit. We introduce five new computationally simple activation functions. These square-based functions are not an approximation of baseline activation functions but are standalone functions. We show the initial speedup of functions using metrics provided by the Intel processor. On various Intel CPUs, a speedup ranging from 1.3x to 4.3x is achieved. We empirically demonstrated the activation over time and the inference time, as well as the speedups obtained using ARM M3 processor. Overall, on the ARM M3 processor, the square-based functions obtained a speedup of 5.73x to 169.79x when compared to baseline. The derivatives are linear in nature and also achieved a speedup of 3.66x to 38.95x on the ARM M3 processor. We have demonstrated extensively that square-based activation functions achieve comparable or better performance accuracy on several datasets and neural network architectures. The advantage of square-based nonlinearity will manifest in the inference engine block.

Chapter 5

Resource Efficient Asymmetric Activation Functions Generator

Abstract

Implementations of machine learning models in resource-limited embedded systems are becoming highly desired. This has led to a need for resource-efficient building blocks for computing the mathematical operations required for neural network training and inferencing. Efficient activation functions for low-end hardware devices with limited hardware capabilities are important. In this chapter, we present an algorithm for generating asymmetric activation functions. The digital implementation of the proposed algorithm is highly amenable to parallelisation and extremely resource-efficient when compared to other forms of hardware approximation. Furthermore, we introduced an arbitrary multifunctional generator called SQ-GEN. We show that it is efficient and is characterised with arbitrary precision without affecting the resource usage. We record an area-saving on FPGA and ASIC platforms with different precision.

5.1 Introduction

Convolutions, multiplication, activations, pooling, and normalisation are the major mathematical operations in which the majority of the computing effort of various deep learning architecture training and inference stage is allocated. Hardware implementation of these operations possesses several challenges [132]. There are many nonlinear functions used as activation functions for deep learning neural network models. Some of these functions, such as ReLU, are very simple and can be implemented trivially with simple operators on hardware; others require some type of approximation method or extra

memory by using a lookup table (LUT). Every variant of ReLU or other deep learning based activation functions are computationally expensive. For example, Leaky ReLU [29] and PReLU [1] require a multiplier, ELU [3], swish [116], and SELU [5] require an exponential, softmax requires exponential and division. These mathematical functions are computationally expensive on hardware and hence, leads to challenges for hardware-based deep learning architectures.

The impact of activation function in both area and latency for hardware accelerator is important. Implementing an accurate, low cost, power efficient, and low latency activation function is an important aspect of implementing deep learning models on FPGAs. There have been multiple activation functions proposed by researchers over the years due to a need for higher performance accuracy, better convergence speed, reduced computational cost, and so on. However, most activation functions include primitive operators such as exponential and division functions. There is a high area cost associated with implementing these functions on FPGAs. Efficient hardware implementation of computationally intensive activation functions will contribute to designing effective and efficient deep learning accelerators.

In literature, various approaches have been proposed for implementing efficient hardware transcendental activation functions. These methods falls into three broad categories: look-up table (LUT), piecewise approximation, and hybrid methods. LUT method has been shown to generally be resource and memory intensive for higher resolution and precision [133]. The structure of a LUT is not amenable to direct expansion when there is a need for factors such as change in resolution, activation function, and so on. Large table sizes provide better results but increase the memory requirements. There are different types of piecewise approximation which include piecewise linear [28, 134], piecewise nonlinear [26], and others [27, 135]. In most cases, piecewise approximation requires one or more multipliers [134]. As described in [24, 37], multipliers are resource hungry and power hungry devices on hardware. Other types of piecewise approximation have been shown to not require any multiplier but rather only comparators, multiplexers, shift operators, storage of several coefficients. For a small degradation in performance accuracy for some problem space, shift operator based piecewise approximation activation functions will suffice.

Authors in [136] show that activation functions affect the learning and generalisation capabilities of neural networks. In most machine learning architectures, each neuron in the hidden and output layers needs some type of activation function. The rationale for focusing on the efficient hardware implementation of computationally expensive activation functions include the following: (i) not all machine learning architectures can use the simple ReLU, (ii) some computationally expensive activation functions have been shown in literature to out perform ReLU [3, 29, 116]. For example, functions like ELU have the

advantage of not requiring computationally expensive batch normalisation, decreasing training time and improving performance accuracy. (iii) The number of activation functions per layer is increasing with machine learning model complexity. For example, the popular VGG [127] has two dense layers of fully connected neurons of 4096 activation functions each before the final layer of softmax function. Deep autoencoder for MNIST [124] has a network with layers of width 1000, 500, 250, 30, 250, 500, 1000 ELU functions, and 784 neurons of sigmoid function. ELU-network [3] for ImageNet dataset contains two layers of 4096 ELU functions in the fully connected layers and other convolutional layers containing ELU function. Finally, SELU function has recorded state-of-the-art results using feed forward neural networks for different datasets. The new state-of-the-art accuracy recorded for Tox21 dataset requires each hidden layer to have between 1024 and 2048 SELUs.

Therefore, hardware implementation of networks with any of these computationally expensive functions will lead to high resources and power consumption from the activation function alone; thus, showing the need and importance of efficient hardware implementation of computationally expensive activation functions. Overall, the number of hardware activation function components can be significant. Hence, optimisation of activation function circuits could dramatically decrease neural network area and power requirements [135, 137]. As a result of this analysis, efficient execution/implementation of matrix multiplication operations should not be the only focus of designers during hardware implementation. Special attention should be paid to other components of hardware implementation of machine learning models hence, the focus of this chapter, which is to propose resource efficient hardware implementation of asymmetric activation functions. Our proposed hardware implementation method is simple to implement in digital circuits as the complexity of the computations can be significantly reduced by using subtraction, sum, and accumulate.

The following are the contributions of this chapter:

- Computationally and resource-efficient asymmetric activation functions generator.
- We introduced an arbitrary generator called SQ-GEN: multifunctional (symmetric and asymmetric activation functions) generator. We show that it is efficient and is characterised with arbitrary precision without affecting the resource usage.
- Arithmetic logic unit implementation and the associated resource footprint of the square-based asymmetric activation functions.
- Resource and computational footprint on FPGA and ASIC hardware platforms

5.2 Concepts

As with the symmetric functions generator defined in Chapter 3, the basis of implementing asymmetric functions is encapsulated in Equation 3.1 of Chapter 3 ($f(n) = \frac{1}{N} \sum_{k=1}^N (((n + U(k)))_C - U(k))_M$) with slight modifications. The asymmetric function can be affected by making two changes.

- $U(k) = \{-2U_{MAX}, \dots, 0\}$
- Adder saturation $C = \{-U_{MAX}, M\}$ where $U_{MAX} = 2^{R-2}$ and $M = 2^{R-1}$, i.e., only the lower saturation boundary is required.

Figure 5.1 plots the impact of the saturating adder at three different inputs - n_1 , n_2 , and n_3 . With n_1 , some of the partial sums of $U(k)$ (close to zero) do not experience saturation and hence their partial sums, i.e., $((n_1 + U(k))_C - U(k))_M = n_1$. These correspond to the horizontal section of the profile. Unlike the symmetrical function, the saturation in this case causes an increase in the values of the partial sums $((n_1 + U(k))_C - U(k))_M > n_1$ and corresponds to the sloping section of the profile shown in Figure 5.1b. Thus $f(n_1) > n_1$.

The situation for n_2 is similar to that for n_1 . However, for $n = n_3$, none of elements in $U(k)$ result in saturation and hence $f(n_3) = n_3$. The complete mapping is sketched in the top right quadrant of Figure 5.1 a).

As discussed with the symmetric generator, a pragmatic N is necessary. The partial sums for $N = 8$ are shown in Figure 5.1 b) overlayed with an idealised profile. A closed form expression can be obtained

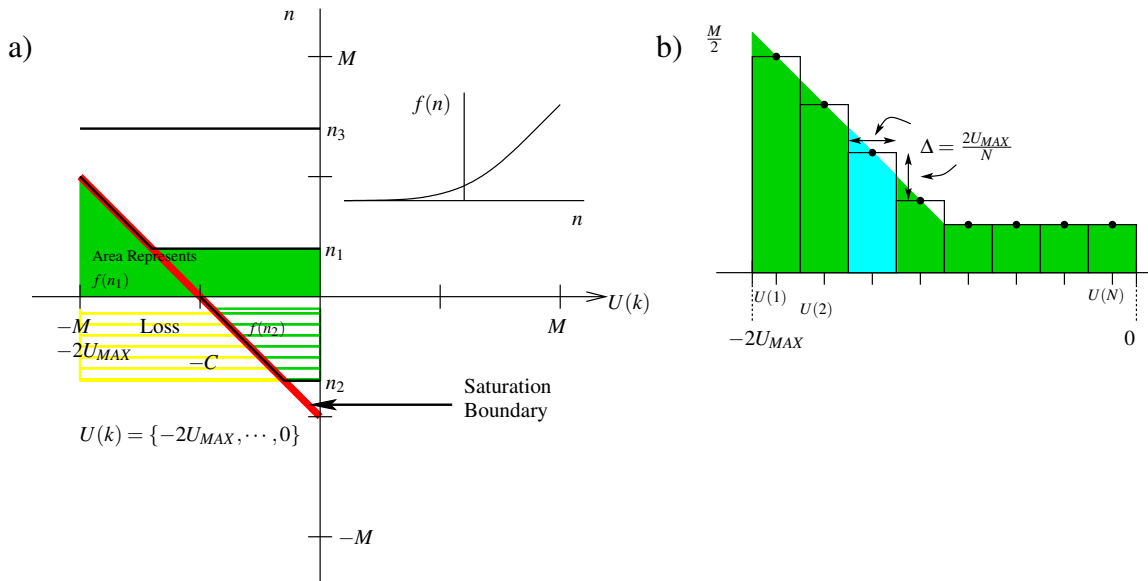


Figure 5.1: An Asymmetric SQLN Activation Function

by viewing each segment in Figure 5.1 b) as rectangles or trapezoids. Here, we consider each segment as a trapezoid. It should be noted that the results for rectangular segments are identical.

For $0 \leq n \leq U_{MAX}$,

$$f(n) = \frac{1}{2U_{MAX}} [\text{area}(U(1)) + \text{area}(U(2)) + \dots + \text{area}(U(p)) \times \Delta + n \times \Delta + n \times \Delta]$$

Here, $\Delta = \frac{2U_{MAX}}{N}$ and $U_{MAX} - p\Delta = n$,

$$\begin{aligned} \text{area}(U(1)) &= \frac{\Delta^2}{2} + (p\Delta - \Delta) + n\Delta \\ \text{area}(U(2)) &= \frac{\Delta^2}{2} + (p\Delta - 2\Delta) + n\Delta \\ \text{area}(U(p)) &= \frac{\Delta^2}{2} + (p\Delta - p\Delta) + n\Delta = \frac{\Delta^2}{2} + 0 + n\Delta \end{aligned}$$

Hence,

$$f(n) = \frac{1}{2U_{MAX}} \left[\sum_{i=1}^p \frac{\Delta^2}{2} + \sum_{i=1}^{p-1} i\Delta^2 + \sum_{i=1}^p n\Delta \right] = \frac{(U_{MAX} + n)^2}{4U_{MAX}}$$

It can be shown that for $-U_{MAX} \leq n \leq 0$, $f(n) = \frac{(U_{MAX} + n)^2}{4U_{MAX}}$. Since we select $U_{MAX} = \frac{M}{2}$ and hence the complete mapping is,

$$f(n) = \begin{cases} 0 & : n < -\frac{M}{2} \\ \frac{(\frac{M}{2} + n)^2}{2M} & : -\frac{M}{2} \leq n \leq \frac{M}{2} \\ n & : n > \frac{M}{2} \end{cases} \quad (5.1)$$

The activation function as a result of Equation 5.1 will subsequently be referred to as SQ_Softplus.

We define another asymmetrical function that has an identity in the positive region but a smooth nonlinearity in the negative region. This incorporates a parameter that defines the negative asymptotic limit. The SQNL based asymmetric functions can also be parameterised and offer an argument with similar impact. For this mapping, we modify $U(k)$ such that,

- $U(k, \alpha) = \{-2U_{MAX}, \dots, 0\} + \alpha$, where $0 \leq \alpha \leq \frac{M}{2}$

As α is increased from 0 to $M/2$, the saturation boundary moves to the right and downwards by α . This is shown in Figure 5.2 b). The impact on the partial sums at a particular input $n = n_1$, at an arbitrary α as well as at $\alpha = 0$ is shown in Figure 5.2 a). The profiles of partial product has been reproduced in Figure 5.2 c). For $U(k, 0)$ the saturation leads to a bigger offset and hence $f(n_1, 0) > f(n_1, \alpha)$. The

negative limit of $f(n, \alpha) = -\alpha$. The inset in Figure 5.2 a) sketches the mapping.

$$f(n) = \begin{cases} -\alpha & : n < -\frac{M}{2} - \alpha \\ \frac{(\frac{M}{2} + n + \alpha)^2}{2M} - \alpha & : -\frac{M}{2} - \alpha \leq n \leq \frac{M}{2} - \alpha \\ n & : n > \frac{M}{2} - \alpha \end{cases} \quad (5.2)$$

This activation function (Equation 5.2) at $\alpha = 64$ will be called SQLU. It is important to note that when $\alpha = 0$, then Equation 5.2 is the same as Equation 5.1, and hence the function becomes SQ_Softplus as defined earlier.

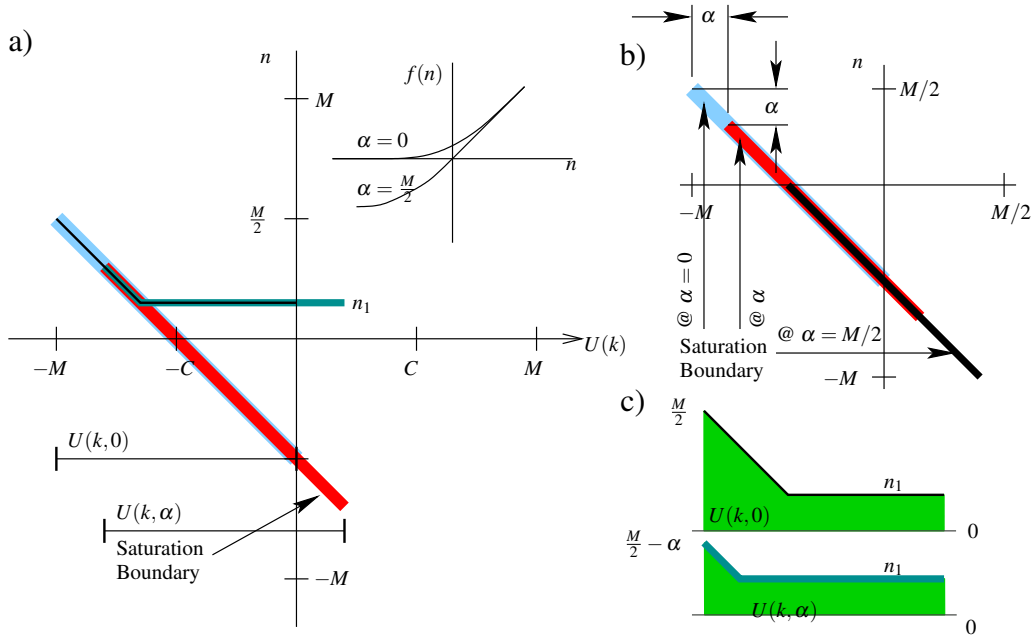


Figure 5.2: A Parameterised Asymmetric SQNL Activation Function

The algorithm inherently offers a wide spectrum of mapping. In addition to the two mappings discussed earlier, other variations are discussed below. These modifications require a very small increase in logic gate usage.

SqREU A non monotonic mapping can be obtained by arranging α in Equation 5.2 to vary conditionally. For this mapping, we modify Equation 5.2 such that,

- at $n < -\frac{M}{2} - \alpha$, $f(n) = 0$
- at $n > \frac{M}{2} - \alpha$, $f(n) = n$

- at $-\frac{M}{2} - \alpha \leq n \leq \frac{M}{2} - \alpha$, $\alpha = \frac{M}{2} + n$, hence $f(n)$ is derived as follows:

$$\begin{aligned}
 f(n) &= \frac{(\frac{M}{2} + n + \alpha)^2}{2M} - \alpha \\
 \alpha &= \frac{M}{2} + n \\
 f(n) &= \frac{(\frac{M}{2} + n + \frac{M}{2} + n)^2}{2M} - \frac{M}{2} - n \\
 f(n) &= \frac{(M + 2n)^2}{2M} - \frac{M}{2} - n
 \end{aligned}$$

Let $n = \beta \cdot M$

$$\begin{aligned}
 f(n) &= \frac{(M + 2\beta M)^2}{2M} - \frac{M}{2} - \beta M \\
 f(n) &= \frac{M^2(1 + 2\beta)^2}{2M} - M(\frac{1}{2} + \beta) \\
 f(n) &= M \left[\frac{(1 + 2\beta)^2 - (1 + 2\beta)}{2} \right] \\
 f(n) &= M \left[\frac{(1 + 2\beta)2\beta}{2} \right] \\
 f(n) &= M\beta(1 + 2\beta) \\
 f(n) &= n(1 + 2\frac{n}{M})
 \end{aligned}$$

Therefore the complete mapping for SqREU function is,

$$f(n) = \begin{cases} 0 & : n < -\frac{M}{2} - \alpha \\ n + \frac{2n^2}{M} & : -\frac{M}{2} - \alpha \leq n \leq \frac{M}{2} - \alpha \\ n & : n > \frac{M}{2} - \alpha \end{cases} \quad (5.3)$$

SQINE Modifying $U(k)$ to custom values offers an alternative to the Sine function on hardware. This mapping offers a worst case of two bits error on 16.34% of values.

5.3 Analysis

5.3.1 Computational Footprint on Embedded NIOS II Processor

The square-based asymmetric functions mapping with a custom hardware implementation has the potential to achieve the throughput of the MUL(tiply) instruction. Figure 5.3 shows a possible implementation. The schematic shows a square operation achieved using one Multiply and Accumulate (MAC)

and additional combinatorial logic that results in the final SQLU. The combinatorial logic is not reliant on any clock and hence will only impart a propagation delay. It is estimated that this custom instruction will result in an execution time very similar to that of a multiplier, i.e., one to seven clock cycles.

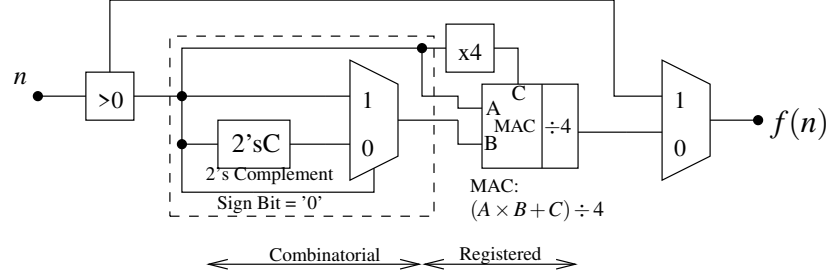


Figure 5.3: Hardware Implementation of SQLU using the custom multiplier (special square operator). The $\div 4, \times 4$ operations are right shift operations and hence, have no impact on the computational or resource footprint. Slight modification will result in other asymmetric functions.

Current technology does not allow us to have custom instruction on ARM. Hence, we created custom instructions on the Altera Cyclone V FPGA Nios II for verification. Figure 5.3 was implemented for all the square-based activation functions using custom instructions. The simple ReLU function was implemented using custom instruction. Experiments show that all the square-based activation functions implemented using proposed custom instruction on Nios II produce the result in one clock cycle like the simple ReLU function. Therefore, computationally speaking, all the proposed square-based functions, and ReLU are the same. The transcendental functions are written directly in C using the inbuilt math library. The computational time of floating point and fixed point custom instructions for the SQL family and their exponential based equivalent shows a consistent speedup for square-based nonlinearities.

5.3.2 Resource Footprint: Arithmetic Logic Unit Implementation

The square operation in Equation 4.1 can be replaced with a modified 'square' operator, defined as $f_s(x) = x \times -|x|$. This operator can be used to produce other nonlinearities at an extremely low computational cost. Since hardware multipliers are common in most embedded processors, this square operator could be implemented in the processor and available as a custom instruction. One possible implementation is shown in Figure 5.4.

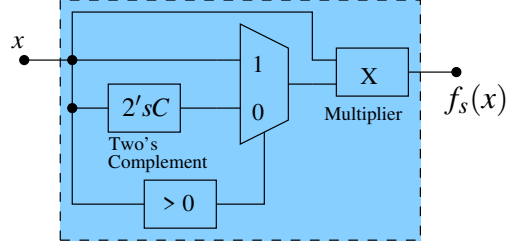


Figure 5.4: The implementation of a custom square operator ($f_s(n) = n \times -|n|$) using a multiplier.

Each of the proposed activation functions will make use of the special square operator for efficient calculations. The 2's complement and the data selector are combinatorial, and hence, its timing is very comparable to a standard multiplier. Using this operation, the execution of SQLN, (Equation 4.1) reduces to $f(x) = x - f_s(x)$. This is shown pictorially in Figure 5.5a.

Besides the fact that the nonlinearity accounts for a relatively short computational time, the physical footprint of the SQLN family is small. So if an inference engine implementation is a focus, then the SQLN family will take minimal resource space as compared to the exponential based functions. This is as shown in Figure 5.5a-d for the SQLN family.

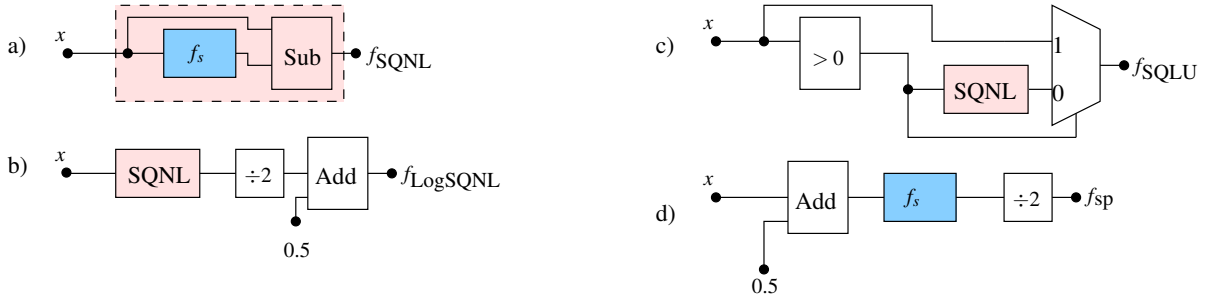


Figure 5.5: SQLN family implementation schematics: a) SQLN using the custom multiplier $f_s(x) = x \times -|x|$. b) Log_SQLN. c) SQLU d) SQ_Softplus. Note: The $\div 2$ and $\div 4$ operations are right shift operations and hence, have no impact on the computational or resource footprint.

5.4 Hardware Implementation

The asymmetric activation functions have been implemented on an Altera Cyclone V device (5CSXFC6D6F31C6). The Quartus Prime Standard 18.1 Edition and Modelsim 10.1d are the integrated development environment on which all the hardware implementation is performed. We will only discuss the multi-clock/counter solution for the asymmetric functions in this section. The single-cycle solution and multiplier-based solutions follow the same approach as already discussed in chapter 3.

Multi-clock/Counter solution - Figure 5.6 shows a complete schematic with various bus-widths highlighted. The code is parameterised, and hence the saturation of the adder and subtractor change

with the resolution R . Section 6.4 describes the different changes to the parameters based on the type of function. The asymmetric counter is $\{-1, -3, -5, \dots, -15\} \times 2^{R-2}/N$. This is easily obtained by suitable padding of leading and trailing bits. The summation of α with the counter values results in asymmetric functions. The input and output, marked with thick lines, could be fixed point at a larger bus width than the activation function. The lower Add, Latch, and Right-Shift blocks function as the filter (sum, accumulated, scale). The optimised values of the parameters are defined based on the activation function and summarised in Table 5.1. An additional speedup of $2\times$ can be extracted by noting that, for all inputs, half of the partial sums due to positive/negative $U(k)$ are constant. Thus, the accumulator could be preloaded with either $n\frac{N}{2}$ or $\pm \sum_{k=0}^{N/2} U(k)$. This implies only four clocks are required for $N = 8$. The single-cycle and the multiplier solution is as described in chapter 3.

Table 5.1: The optimised values of different activation functions for hardware implementation ($U_{MAX} = 2^{R-2}$ and $M = 2^{R-1}$)

Function	Alpha (α)	Adder saturates (C)	Subtractor saturates (M)
SQLU	2^{R-2}	$-U_{MAX}, M$	-2^{R-1} or $2^{R-1} - 1$
SQ_Softplus	0	$-U_{MAX}, M$	-2^{R-1} or $2^{R-1} - 1$
SqREU	$\frac{M}{2} + n$	$-U_{MAX}, M$	-2^{R-1} or $2^{R-1} - 1$

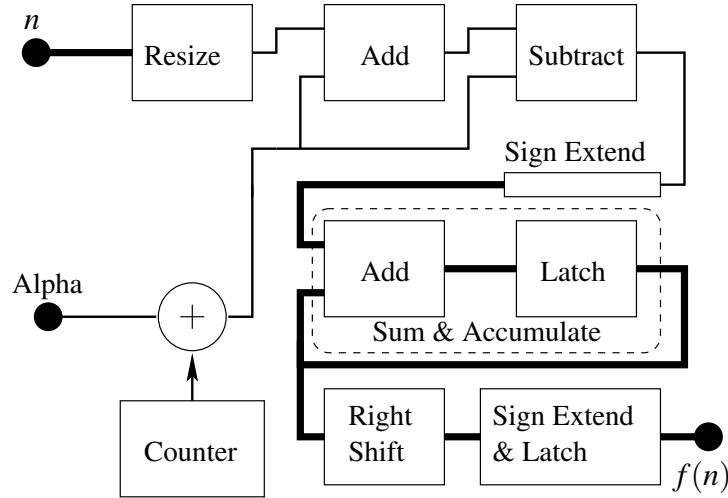


Figure 5.6: Schematics Asymmetric Activation Function Generator using multi-clock methodology.

5.5 Results and Discussion

In this section, we present the resource utilisation of the hardware implementation of asymmetric functions. Furthermore, we introduce and discuss the resource utilisation of a square-based multifunctional generator.

Table 5.2: Resource utilisation of asymmetric activation function implementation using a custom Booths Radix-4 multiplier (mult), multi-clock (counter, $N = 8$), and LUT solution. As displayed the counter based method for the asymmetric activation function consistently outperform the multiplier solution on both ASIC and FPGA platforms. At a lower resolution, the LUT performs slightly better than the counter solution on FPGA but worse on ASIC when compared with the counter solution. The LUT on the other hand is only better for lower resolution and can not accommodate applications where higher resolution is required. The counter solution scales well across different resolutions.

R	Methods	SQLU ($\alpha = 2^{R-2}$)			SQ_Softplus ($\alpha = 0$)		
		Gates	ALM	FF	Gates	ALM	FF
8	Mult	987	124	63	826	100	54
	LUT	2779	17	8	2747	19	8
	Counter	460	31	22	460	29	24
12	Mult	1473	187	93	1232	153	80
	LUT	67719	234	12	67671	125	12
	Counter	664	41	31	664	39	30

5.5.1 Resource Utilisation of Asymmetric Function Generator

We compare the multiplier solution, multi-clock solution, and the use of LUT on the proposed square-based functions. Table 5.2 shows the resource utilisation of the asymmetric functions. Both FPGAs and ASIC platforms have been considered.

The LUT based solution is a single clock operation while the multiplier and counter-based solutions require more clocks. The gate/ALM/Register usage varies with R and N hence, the three methods are compared by normalising their throughputs. With $R = 8$, the Booths and counter algorithms take four clocks hence, with $N = 8$, the counter offers a $987/460 = 2.14$ greater throughput per used gate. The ratio of gate counts usage for normalised throughput is presented in Table 5.3. The asymmetric function uses the same clocks as the symmetric function, although it uses slightly more gates. The clocks associated with the asymmetric function are independent of the resolution (i.e., $R = 8$ and $R = 12$ will both require four clocks for $N = 8$) and thus the dynamic range can be increased without a significant impact on timing. Although the LUT offers single-cycle performance, the throughput per gate of the LUT is generally inferior, it does not scale well, and provision of a table update mechanism is expensive. Although a counter-based implementation consistently offers a higher throughput per gate, the availability of a multiplier would be attractive at higher resolutions if larger values of N are necessary. The resource utilisation of the single-cycle solution offers area savings when compared to LUT and multiplier-based solution.

Table 5.3: The ratio of gates usage for normalised throughput. The counter-based solution performs better than the multiplier and LUT solutions both at a lower resolution and higher resolution. At higher resolution, the counter-based solution achieves extremely high throughput when compared to the LUT.

		R = 8		R = 12	
		N = 4	N = 8	N = 8	N = 16
SQLU - $\alpha = 2^{R-2}$	Counter Vs Multiplier	4.45	2.14	3.33	1.62
	Counter Vs LUT	3.14	1.51	25.50	12.43
	Multiplier Vs LUT	0.70		7.66	
SQ_Softplus - $\alpha = 0$	Counter Vs Multiplier	3.73	1.80	2.78	1.36
	Counter Vs LUT	3.10	1.49	25.48	12.42
	Multiplier Vs LUT	0.83		9.15	

Table 5.4: Resource Utilisation of SQNL,SQLU,SQ_Softplus Combination Implementation using custom implementation

Methods	8 bits		12 bits	
	ALM	FF	ALM	FF
Counter	31	26	39	32

5.5.2 Resource Utilisation of SQ-GEN

The beauty of our custom method is the ability to switch between two or more square-based activation functions with just a minor tweak to the coefficient loaded. Using mode, we can select which activation function to use. The symmetric (already discussed in Chapter 3) and asymmetric implementations share commonalities and have also been integrated into a multifunctional generator. Multiple instances can share a single counter output which would extract additional efficiencies. This would be attractive in headless inference engines like the designs of TPU™, Nervana™, and NVDLA. We refer to this block that can produce different functions as SQ-GEN. The resource utilisation increase is shown, which is about 5%. This is not possible if we are using direct methods or the LUT method. This further shows that an SQ-GEN can switch from one function to another by just changing the loaded sequence. Therefore, the SQNL, SQ_Softplus, and SQLU functions can be combined into one block which can be changed with "mode select". The combined block is particularly useful in network architectures that employ two or more activation functions, for example an LSTM-FCN model [138] have symmetric (Tanh, Sigmoid), asymmetric (ReLU, SQLU), and normalising (softmax, sqmax) activation functions. Therefore we can replace the need for several LUTs or approximation methods with a single block of SQ-GEN.

The proposed algorithm and the implementation has demonstrable advantages over the current state-of-the-art as summarised below:

Counter At low/mid resolutions the counter based solution gives the maximum density of activation functions. The counter based solution permits encapsulation of symmetric and asymmetric func-

tions into a single entity. Thus, activation functions can be dynamically adapted during training and inferencing. Our experiments on various models show that the absolute error due to $N \leq 8$ has no impact on model performance. However if $R > 16$ and $N > 16$, the multiplier may be more efficient but if $N \leq 8$ the counter based method is much more attractive.

Multiplier A multiplier solution would be attractive if free multipliers are available and if high/full precision is essential. Furthermore, since our square-law based functions are not an approximation, the direct solution will not introduce any approximation errors when compared to the piecewise linear approximation methods found in the literature.

LUT A LUT based function under performs compared with a counter based solution. At higher resolutions, the LUT is not a practical option due to an excessive gate usage. At lower resolutions ($R < 8$), a LUT may outperform a counter based solution on an FPGA but this is device dependent.

5.6 Conclusion

We have introduced a novel method for generating computational and resource-efficient asymmetric activation functions. The consequent mappings have been analytically obtained and experimentally verified. The experiments were performed on an Altera Cyclone V FPGA. Although the counter based implementation uses lower resources when compared to other methods, the proposed functions have also been implemented using hardware multipliers. We show that combinatorial logic wrapped around a standard multiplier offers an attractive alternative to the counter implementation. Comparisons with LUT implementation, a counter-based asymmetric function with an 8-bit resolution offers an estimated throughput (per gate) speedup of 1.51x to 3.14x. Similarly, a speedup of 2.14x to 4.45x is estimated with multipliers. Higher throughputs per gate have also been estimated with 12-bit implementations. The single-cycle hardware implementation of the proposed method is also resource-efficient for higher resolution and provides area savings when compared to LUT. The proposed method can produce many other mappings with relatively small modifications. It is attractive in applications that require a dynamic mapping change.

Chapter 6

Computationally Efficient Radial Basis Function

Abstract

We introduced a square-law based RBF kernel, called Square RBF (SQ-RBF), which is computationally efficient and effective due to the elimination of the exponential term. In contrast to the Gaussian RBF, SQ-RBF requires smaller computational operation count and direct implementation without a call to higher order library. The derivative of the SQ-RBF is linear which will improve gradient computation and makes its applicability in gradient-based training algorithms attractive. We present the hardware implementation of the SQ-RBF using three methods. Empirically, SQ-RBF leads not only to faster learning but also requires significantly fewer neurons than Gaussian RBF on Radial Basis Function Neural Networks. On average, we recorded a speedup in training time of about 8% for SQ-RBF based networks without affecting the overall generalisability of the network. SQ-RBF uses about 10% fewer neurons than Gaussian RBF hence making it very attractive. Furthermore, the SQ-RBF was modified for support vector machines. Speedup was recorded with proposed quadratic-based kernel transformation on support vector machines, showing the computational simplicity of the SQ-RBF kernel function. Finally, in terms of resource utilisation, our proposed hardware implementation shows significant area savings when compared to existing methods.

6.1 Introduction

In the field of Artificial Neural Network (ANN) and machine learning, there is a constant need for improvement in cost-energy-performance. There is an exponential growth in data, and the more data available for an ANN model during training, the better the inference (prediction) stage. This has led to several architectures being developed to be used during training as well as inference. Graphical Processing Units (GPUs) were able to provide speedups that were not possible when using Central Processing Units (CPUs). Furthermore, Google introduced an application specific unit dedicated to machine learning. This is referred to as Tensor Processing Units (TPUs) which has been described to be about 15x and 30x faster than GPUs and CPUs respectively [115]. This shows that computational speed is as important as performance accuracy when it comes to any neural network architecture.

Gaussian Radial Basis Functions are commonly used in various machine learning architectures among which are multilayer perceptron, Radial Basis Function Neural Network (RBFNN) and Support Vector Machine (SVM). The presence of the exponential term in the Gaussian function made its direct implementation on hardware not possible except through approximation methods. Therefore, the elimination of the exponential term will not only alleviate the computational intensity, but also lead to faster simulations. The introduction of GPU and other platforms for training neural networks has led to significant speedups in simulation time but as described in [139–141], the exponential function is not implemented with hardware on GPUs but rather with software library. With this, the speedup possible using GPUs is defeated. Over the years, there have been different approximations of the Gaussian RBF, and an exhaustive list can be found in [142]. The authors in [143] proposed a replacement for the Gaussian kernels which is adapted by combining three activation functions, namely sigmoid, multi quadratic, and Gaussian activation functions. However, this new activation function still makes use of the exponent term heavily.

This chapter aims to present a computationally efficient and improved version of the Gaussian RBF activation function. This new function, although having a similar convex shape as the Gaussian function is characterised by just multiplication and subtraction mathematical operators, and it is capable of achieving better performance. The new function requires only multiplications, subtractions, and logical operations to obtain a bell-shaped Gaussian-type function. Moreover, the derivative of the function is purely combinatorial (comprising just addition and subtractions) which makes gradient computation easier when used in gradient-based algorithms. The function proposed in this chapter can replace the existing Gaussian function through its well-formed bell-shaped nonlinearity. Additionally, extensive ex-

periments have been performed to show the performance and advantages of the proposed function on RBFNN and SVM architectures. The following are the contributions of this chapter:

- Introduction of a new computationally efficient activation function for Radial Basis Function and Support Vector Machine Architectures.
- Hardware implementation of the proposed function with significant area savings.
- Improved training and inference computational time on various datasets.

6.2 RBF Networks and RBF Kernels

RBFNN was introduced by Broomhead and Lowe [144] in the late 1980s and has found significant applications and success in areas such as function approximation, interpolation, dynamic system design, and classification. They perform excellently to date in any form of approximation problem, and their excellent approximation capabilities have been studied extensively in literature [145, 146]. The most important part of the RBFNN architecture is the hidden layer neurons, commonly referred to as RBF kernels. As described in [147], the RBF kernels transform the input patterns (data) into a new feature space through the help of a strictly positive, radially symmetric activation function. Researchers have been working to improve RBFNN training with more focus on the training of RBFNN. In this work we shift focus to the centre selection through the RBF kernel. The importance of the right RBF kernel was extensively illustrated in [147]. One of the most widely used RBF kernels is the Gaussian kernel. RBF networks are similar to MLP in that they are feedforward neuron networks architecture, but they are characterised with a single hidden layer whereas MLP networks can have more than one hidden layer. Each hidden layer unit is characterised by a radial activation function. The output layer, on the other hand, implements the weighted sum of the hidden node outputs. A survey carried out in [148] describe RBF networks as a current trend in the successful modelling of various industrial processes.

The training of RBFNN is in two stages: identification of the radial basis centre for the RBF neurons in the hidden layers and the learning of the weights present in the hidden layers to the output layer. Several methods can be used for the centre definition, namely random sampling of input data, unsupervised clustering (commonly K-means), and Self Organising Map (SOM). On the other hand, the weights can be tuned using supervised linear perceptron training, backpropagation, Moore Penrose Pseudo Inverse, Least Mean Squares, and some additional training methods as proposed in literature [148, 149]. Like their MLP counterparts, the performance accuracy, convergence speed, and generalisability of an

Table 6.1: Common and Approximate RBF Kernels

RBF Kernel	Expression
Gaussian	$f(x) = \exp^{-(\beta x)^2}$
Multiquadric	$f(x) = \sqrt{1 + (\beta x)^2}$
Inverse Multiquadric	$f(x) = \frac{1}{\sqrt{1 + (\beta x)^2}}$
Thin-plate Spline	$f(x) = x^2 \log(\beta x^2)$
C ⁴ Matern	$f(x) = \exp^{-\beta x} \cdot (3 + 3\beta x + \beta x)^2$
Approximate Gaussian	$f(x) = \frac{1}{1 + (\beta x)^2}$
Approximate Gaussian	$f(x) = \frac{1}{1 + (\beta x)^4}$

RBFNN depends mostly on architecture, initialisation heuristics, choice of activation function, regularisation techniques, and learning algorithms.

Among all the different types of activation functions for RBF networks, Gaussian function tends to be the most popularly used. Unfortunately, this function is characterised by an exponent term. Exponent function is generally computationally expensive due to a call to high order function term. As described in [147], Radial basis kernels fall under the family of functions whose value depends on the distance between the variable x and a defined centre point c . In other words, an RBF kernel $f(x, c) = ||x - c||$. There are various types of RBF kernel $f(x, c)$ as seen in literature. Table 6.1 shows some kernels with their associated mathematical expressions.

The expressions in Table 6.1 all show a need for multiple computationally expensive operations. However, the two approximate Gaussian functions do not have exponent terms and square roots and are "simple" as described in literature [150]. In addition, the divisor term is not a power of two hence, shift operation cannot be used, thereby requiring division which is resource intensive when compared to logical operation. The next section will show a new function that is computationally efficient and more effective than all the RBF kernels listed in Table 6.1. The mappings of each of the functions listed in Table 6.1 are significantly different. A suitable normalising technique is required such that the narrower functions are not adversely affected. There are many ways of normalising the functions, but in this chapter, we have opted to compare only the Gaussian RBF.

6.3 Nonlinear Support Vector Machine and RBF Kernels

The classification power and flexibility of support vector machines lies in the choice of kernel [151]. The commonly used nonlinear kernels are listed in [152] which are computationally expensive due to the presence of exponent, trigonometric functions, square roots, divisions, and so on. The quadratic kernel $k(x, y) = 1 - \frac{||x - y||^2}{||x - y||^2 + c}$ and the non-positive multi-quadratic kernel defined as $\sqrt{||x - y||^2 + c^2}$

were proposed due to their lower computational cost as compared to the popular Gaussian and have been shown to excel in applications where training time is significant. The kernel function K used in SVM must possess the following characteristics: symmetric, continuous in nature, and possessing a positive definite gram matrix [153]. Polynomial and RBF/Gaussian kernels are the two most commonly used kernel families. The kernel function K must satisfy Mercer's condition [152]: $g(\vec{x})$ such that $\int g(\vec{x})^2 d\vec{x}$ is finite. We must have that $\int K(\vec{x}, \vec{z}) g(\vec{x}) g(\vec{z}) d\vec{x} d\vec{z} \geq 0$.

The Gaussian kernel is, understandably, the first choice of a kernel for nonlinear SVM classifiers due to the following: it requires fewer numbers of hyperparameters in contrast to polynomial kernels [154]. Polynomial kernel defined as $K(x_i, x_j) = (\gamma x_i^T x_j + r)^d, \gamma > 0$ has three hyperparameters unlike Gaussian kernel ($K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2), \gamma > 0$) with only one hyperparameter. These hyperparameters can influence the complexity of the model. Secondly, Gaussian kernels are known to have fewer numerical difficulties [154]. The novel square-based gaussian function will fit right in.

6.4 Novel Square Nonlinear Radial Basis Function (SQ-RBF)

We define a new convex kernel that makes use of a square-law and eliminates the exponential term present in Gaussian expression. We referred to this expression as SQ-RBF and it is defined in Equation 6.1.

$$f(x) = \begin{cases} 1 - \frac{x^2}{2} & : x \leq 1.0 \\ 2 - \frac{(2-x)^2}{2} & : 1.0 \leq |x| < 2.0 \\ 0 & : |x| \geq 2.0 \end{cases} \quad (6.1)$$

Although, SQ-RBF is similar in shape to Gaussian RBF as illustrated in Figure 6.1, it is in contrast to Gaussian kernel; SQ-RBF has smoother asymptotes which can be beneficial during training by using lesser hidden neurons. Furthermore, the approximate Gaussian kernels shown in Table 6.1 are not computationally simple as SQ-RBF because of the presence of non-power of two divisors. SQ-RBF divisor is a power of two which is a shift operation and is not computationally expensive when compared with the former.

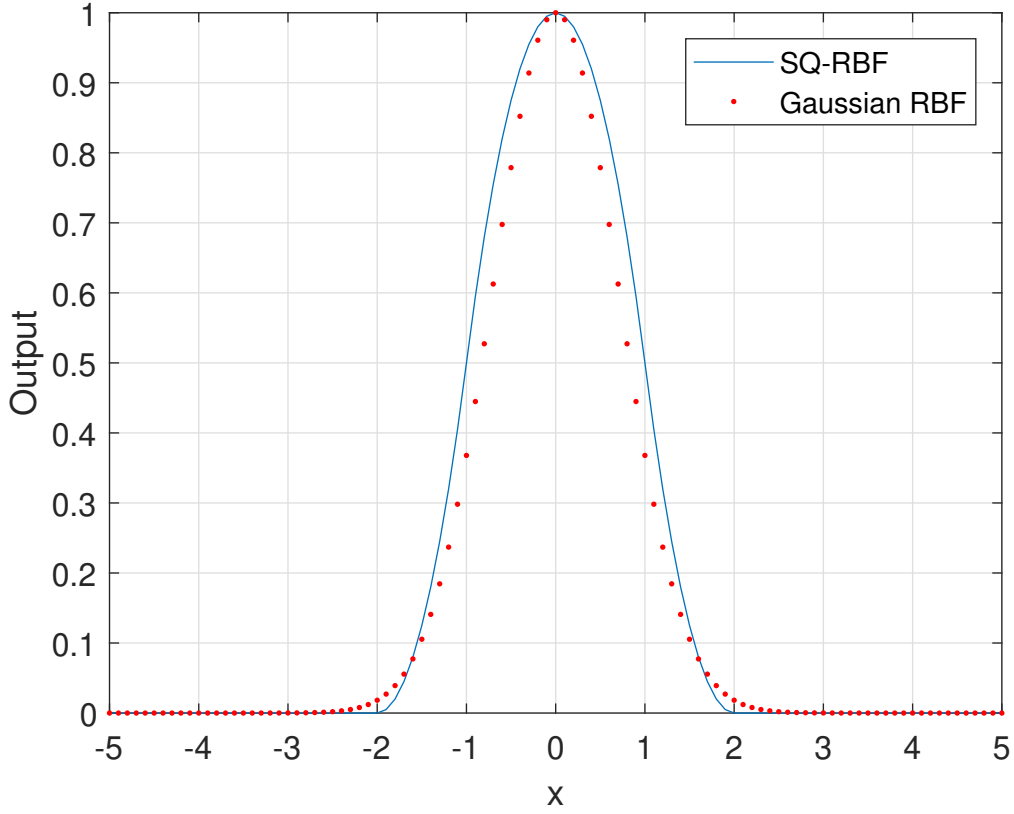


Figure 6.1: The SQ-RBF and Gaussian RBF Kernels

6.5 Mercer's Theorem Proof for SQ-RBF Kernel

Kernel functions used for SVM architectures must satisfy Mercer's theorem. Therefore, in this section, we prove that Equation 6.1 satisfies Mercer's theorem. There are two rules in Mercer's theorem. We show below that the SQ-RBF kernel satisfies both of these rules.

1. Kernel function must be symmetric. Generally, a kernel is a continuous function $K(x,y)$ that takes two arguments x and y and maps them to a real value independent of the order of the arguments. Mathematically, this is defined in Equation 6.2.

$$K(x,y) = K(y,x) \quad (6.2)$$

Equation 6.3 defines the SQ-RBF kernel, where $u = x - y$

$$k(x,y) = \begin{cases} 1 - \frac{(\|u\|^2)^2}{2} & : |u| \leq 1.0 \\ 2 - \frac{(2 - \|u\|^2)^2}{2} & : 1.0 \leq |u| < 2.0 \\ 0 & : |u| \geq 2.0 \end{cases} \quad (6.3)$$

Equation 6.3 is symmetric because $(\|x - y\|)^2 = (\|y - x\|)^2$

2. A kernel function must be positive semidefinite: every Gram matrix $K_{ij} = k(x_i, x_j)$ is positive semidefinite. Mathematically, positive semidefinite is defined as Equation 6.4

$$\sum_{i=1}^n \sum_{j=1}^n K(x_i, x_j) c_i c_j \geq 0 \quad (6.4)$$

The conditions given in Equation 6.4 will always make the output of $k(x, y) \geq 0$, hence, satisfying Mercer's theorem of positive semidefinite.

6.6 Hardware Implementation of Square-based Gaussian RBF

The SQLN function generator proposed in Chapter 3 is modified to produce the SQ-RBF mapping. Overall, to achieve SQ-RBF from the SQLN function generator, three things have to be done. The following are the modifications to the SQLN function generator:

- The absolute value of netsum n is required because the output graph of the SQ-RBF function is a characteristic symmetric "bell curve" shape with positive values. Therefore, the new netsum n is defined as $n = |n|$.
- The magnitude of the netsum is required to be shifted by a value called $nshift$. This $nshift$ is defined as $nshift = 2^{R-2}$. This shift operation is performed by subtracting the magnitude of netsum from $nshift$. That is $|n| - nshift$.
- The original sequence, $U(k)$, consists of N non-repeating values between ± 64 . The new sequence to achieve the SQ-RBF function requires the addition of an offset to $U(k)$. This offset is defined as $O = -2^{R-2}$. The resolution R is a positive integer value, which is the binary resolution of the input and output.

In summary, the counter values are modified by adding an offset. For an 8-bit ($R = 8$) system with $N = 8$, the counter values are $\{\pm 1, \pm 3, \pm 5, \pm 7\} \times 2^{R-2}/N$ and the offset value is -2^{R-2} . The netsum input (n) is modified by finding its magnitude and subtracting a shift value 2^{R-2} . Figure 6.2 shows the schematic for implementing SQ-RBF.

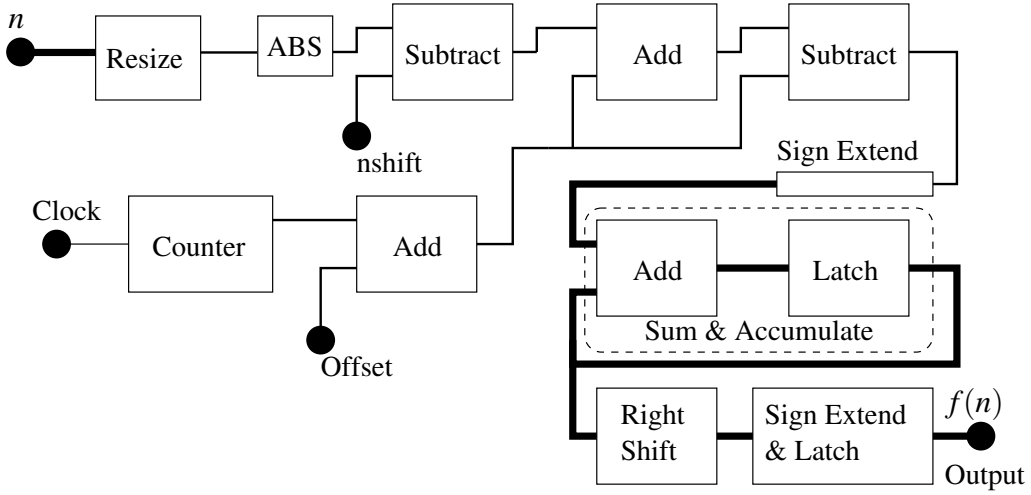


Figure 6.2: Schematic of SQ-RBF Activation Function using Multi-clock Solution

As with the SQNL function, SQ-RBF can also be implemented using the single-cycle and the multiplier solutions.

6.7 Software Experimental Results

In this section, experiments are performed on RBFNN and SVM architectures. The focus is on the training time and the performance accuracy when compared to the Gaussian kernel. All the simulations are run in MATLAB version 2017b Environment. The system is a 64-bit Windows 7 Enterprise with Intel Core i7 processor running at 3.40GHz with 16.0GB of installed memory (RAM). Equation 6.1 will be used for RNFNN architecture while Equation 6.3 will be used for SVM architecture.

6.7.1 SQ-RBF on RBFNN Problems

Two experiments are presented to validate and demonstrate the effectiveness of the proposed function. Our proposed method is compared with the Gaussian function as the hidden layer activation function on a series of benchmark tests and one new problem set. The benchmark tests, as used in [148], are the SinE function approximation dataset, nonlinear dynamic system identification, and finally the Mackey-Glass time series prediction dataset. Performance of the proposed function has been measured by the computational training time, the number of RBF kernels required, and performance accuracy on both the training set and test set which is commonly used in literature to verify and demonstrate the effectiveness and performance of the proposed RBF kernel function. The performance accuracy was based on two criteria, namely: the number of neurons required to get to a specified mean square error (MSE) and the

generalisability independent of the number of neurons. Therefore, the criteria for stopping experiment 1 are based on the MSE goal, while for experiment 2, the stopping criteria are the number of RBF kernels. All the experiments were performed a total of 100 times, and we present the mean results.

SinE Function Approximation

A rapidly changing function named SinE is used in this experiment to evaluate and compare the performance of the SQ-RBF and Gaussian RBF. SinE function is given in Equation 6.5.

$$y = 0.8 \exp(-0.2x) \sin(10x) \quad (6.5)$$

We defined a set (x, y) of 500 data points for the training set and 150 data points for the testing samples. The value of the input x was uniformly distributed in the interval $[0, 3]$ for the training and test samples.

Experiment 1

In this experiment, we defined a specified MSE on which the training stops, in order to know the number of RBF kernels (neurons) needed. The MSE goal is defined as the $0.01 * (\text{mean}(\text{variance}(\text{training target})))$. The kernel spread is given as 0.15. Table 6.2 shows the results. The results in bold perform better, and all the results are an average of 100 trials of network initialisation.

Table 6.2: Experiment 1: Performance Comparison on SinE Function. The best result is shown in bold.

RBF Kernel	Training Time (seconds)	Testing MSE	Number of Neurons
Gaussian	0.6189	0.0060	90
SQ-RBF	0.5555	0.0067	84

Experiment 2

In this experiment, the stopping criteria are the number of RBF kernels. This number is defined as the size of the training samples. The same method was used as in Experiment 1 to set up the remaining parameters. Table 6.3 shows the approximation results obtained with Gaussian and SQ-RBF kernels in terms of the error and computational time.

Table 6.3: Experiment 2: Performance Comparison on SinE Function. The best result is shown in bold.

RBF Kernel	Training Time (seconds)	Training MSE
Gaussian	5.3898	2.93x10e-19
SQ-RBF	5.1141	6.75x10e-15

As shown in Tables 6.2 and 6.3, the time used in training an SQ-RBF based network is shorter than the Gaussian-based network. The performance accuracy of the SQ-RBF is sometimes better and comparable to the Gaussian function. Finally, in terms of RBF kernels needed, SQ-RBF based network requires less than Gaussian-based network; hence, saving computational time as well as network size.

Nonlinear Dynamic System Identification

We evaluated the SQ-RBF on the identification of the nonlinear dynamic system which is one of the commonly used benchmarks for approximation capability of RBF networks. The dataset was obtained from a device functioning as a hair dryer; for full description, readers are referred to [155]. There are ten inputs and one output in this model. We simulated the output using time $t = 1$ to 1000. We used 80% of the data as training and the remaining as test data.

Experiment 1

We defined our target MSE as $0.01 * (\text{mean}(\text{variance}(\text{training target})))$ and the RBF kernel spread (width) as 1.8. Table 6.4 shows the average of 100 runs using the SQ-RBF and Gaussian RBFs. In terms of wall-clock time, SQ-RBF is about 14% faster computationally than the Gaussian kernel. Furthermore, SQ-RBF requires 26 fewer RBF kernels when compared with Gaussian without any detriment to the performance accuracy. This is a significant saving in both computational time and resources without any negative effect on the accuracy.

Table 6.4: Experiment 1: Performance Comparison on Nonlinear Dynamic System Identification. The best result is shown in bold.

RBF Kernel	Training Time (seconds)	Testing MSE	Number of Neurons
Gaussian	5.5499	0.0197	368
SQ-RBF	4.7737	0.0162	342

Experiment 2

Here, the following parameters were changed due to the nature of this dataset. The kernel width remains the same; we constrain the numbers of RBF kernels to be 450 because the number of the target is large. The focus is on how long (in terms of time) it takes to finish a training run to acquire the lowest error possible. Table 6.5 shows the results obtained.

Table 6.5: Experiment 2: Performance Comparison on Nonlinear Dynamic System Identification. The best result is shown in bold.

RBF Kernel	Training Time (seconds)	Training MSE
Gaussian	11.0813	9.09x10e-4
SQ-RBF	10.82	7.49x10e-4

Mackey-Glass Time Series Prediction

Mackey-Glass time series is one of the popular benchmark problems found in the literature for evaluating the performance of RBF networks. The Mackey-Glass series is non-periodic for τ greater than 17 and periodic otherwise. The initial values (for time t less than τ) were generated from uniformly distributed pseudo-random numbers. Equation 6.6 is used to generate the remaining values, $\tau = 30$, 1500 data samples were extracted and used for training while 500 samples were used for testing.

$$\frac{ds}{dt} = \frac{0.9 * s(t) + (0.2 * s(t - \tau))}{1 + s(t - \tau)^{10}} \quad (6.6)$$

Experiment 1

Using the same approach, we defined our target MSE as $0.1 * (\text{mean}(\text{variance}(\text{training target})))$ and the RBF kernel spread (width) as 1.8. Table 6.6 shows the average of 100 runs using the SQ-RBF and Gaussian RBFs. In terms of wall clock time, SQ-RBF is about 8% faster computationally than the Gaussian kernel. Furthermore, SQ-RBF requires 19 fewer RBF kernels when compared with Gaussian without any detriment to the performance accuracy.

Table 6.6: Experiment 1: Performance Comparison on Mackey-Glass Time Series Prediction. The best result is shown in bold.

RBF Kernel	Training Time (seconds)	Testing MSE	Number of Neurons
Gaussian	20.2748	0.0140	450
SQ-RBF	18.6580	0.0173	431

Experiment 2

Here, the following parameters were changed due to the nature of this dataset. The kernel width remains the same; we constrain the numbers of RBF kernels to be 450 because the number of the target is large. The focus is on how long (in terms of time) it takes to finish a training run to acquire the lowest error possible. Table 6.7 shows the results obtained.

Table 6.7: Experiment 2: Performance Comparison on Mackey-Glass Time Series Prediction. The best result is shown in bold.

RBF Kernel	Training Time (seconds)	Training MSE
Gaussian	20.1224	0.0091
SQ-RBF	19.5357	0.0060

Inverse Cosine Function Approximation

This dataset aims to use RBFNN to infer trigonometry angle. The repeating cosine wave is characterised with a magnitude of 1 and a phase offset of π . We generated 2000 data points, using 1600 as training and the remaining 400 as the test set.

Experiment 1

Using the same approach, we defined our target MSE as $0.01 * (\text{mean}(\text{variance}(\text{training target})))$ and the RBF kernel spread (width) as 0.15. Table 6.8 shows the average of 100 runs using the SQ-RBF and Gaussian RBFs. In terms of wall clock time, SQ-RBF is faster computationally than the Gaussian kernel.

Table 6.8: Experiment 1: Performance Comparison on Triangular Function Approximation. The best result is shown in bold.

RBF Kernel	Training Time (seconds)	Testing MSE	Number of Neurons
Gaussian	0.1603	0.0213	3
SQ-RBF	0.1588	0.0189	3

Experiment 2

Here, the following parameters were changed due to the nature of this dataset. The kernel width remains the same, the number of epoch was left to be the number of samples in the dataset. The focus is on how long (in terms of time) it takes to finish a training run to acquire the lowest error possible. Table 6.9 shows the results obtained.

Table 6.9: Experiment 2: Performance Comparison on Triangular Function Approximation. The best result is shown in bold.

RBF Kernel	Training Time (seconds)	Training MSE
Gaussian	295.28	9.7208×10^{-8}
SQ-RBF	260.94	4.4768×10^{-9}

6.7.2 SQ-RBF Kernel on SVM Classification Problems

The description of the dataset is available at [70]. The results are averaged over five runs with each run having 500 epochs. An estimated mean of accuracy with the confidence of 95% is recorded. The results' out-of-sample misclassification rate by using 10-fold cross validation is presented in Table 6.10. The SQ-RBF consistently used less training time for all the datasets as recorded in Table 6.10, with smaller or comparable test error. This further proves the computational simplicity of the SQ-RBF Kernel function.

Table 6.10: Test Error and Training Time (in Seconds) of Gaussian Vs. SQ-RBF based SVM. The best result is shown in bold.

Dataset	Gaussian		SQ-RBF	
	Training Time (s)	Test Error	Training Time (s)	Test Error (%)
Adult	9.3 ±0.9	0.3±0.02	8.9 ±0.5	0.2±0.02
Spiral	0.4±0.2	0.0	0.3 ±0.1	0.0
Covtype	2142±12	0.5±0	2141 ±1	0
Epsilon	2768 ±29	0.5±0	2519 ±24	0.5 ±0

6.8 Hardware Experimental Results and Discussion

We carried out a comprehensive comparison between our multi-clock solution implementation of the SQ-RBF and the approximation methods of Gaussian RBF found in literature. Authors in [103] use the Altera Floating Point Exponent (ALTFP_EXP) megafunction to implement the Gaussian RBF. A single ALTFP_EXP will consume 420 ALMs, 681 registers, 12 DSP blocks, and a 362 block of memory bits. This implementation is the least used in the literature due to its high resource utilisation and latency (17 clock cycles). Another implementation of Gaussian RBF in the literature [156, 157] is the use of Xilinx CORDIC IP Core. This IP core will consume 908 LUTs and 897 Registers, approximately, for a 16-bit system. The following are the other commonly used approximation methods and a brief description of each.

- Taylor Series: The Taylor series expansion for exponential function is defined in Equation 6.7.

$$e^{-x} = 1 - x + \frac{1}{2!}x^2 - \frac{1}{3!}x^3 + \frac{1}{4!}x^4 - \frac{1}{5!}x^5 + \frac{1}{6!}x^6 - \frac{1}{7!}x^7 + \frac{1}{8!}x^8 + \dots + \text{HOT} \quad (6.7)$$

Where HOT is the high order terms and are ignored during the hardware implementation. The larger the order terms, the lower the approximation error obtained. Equation 6.7 can be written

compactly for simplicity as shown in Equation 6.8.

$$e^{-x} = ((\dots(\frac{1}{8}x - 1)\frac{1}{7}x + 1)\frac{1}{6}x - 1)\frac{1}{5}x + 1)\frac{1}{4}x - 1)\frac{1}{3}x + 1)\frac{1}{2}x - 1)x + 1 + \dots + \text{HOT} \quad (6.8)$$

Authors in [158–160] implemented the Gaussian RBF using Taylor’s series of different orders ranging from 4 to 12. On an FPGA, Equation 6.8 (8th order) will require at least seven multipliers, seven divisions, four additions and four subtractions. A 4th order Taylor series, explored in [159], makes use of three multipliers, three dividers and three adders. The adders add the outputs of every order.

- Look up table: The authors in [161, 162] use the ROM and RAM to implement the Gaussian RBF. An 8-bit Gaussian RBF implemented on an FPGA using ROM will use five ALMs and a 1024 block of memory bits while a 16-bit will use 26 ALMs and 524,288 block of memory bits. This shows that LUT-ROM based Gaussian implementation is not scalable for higher resolution.
- Piecewise Linear: A 5-segment piecewise approximation of Gaussian RBF was proposed in [163] to eliminate the need for the time-consuming exponent term present in Gaussian RBF. This 5-segment piecewise linear approximation is made up of two multipliers, several additions and subtractions, and floating point division.

We implemented a 6th order polynomial and LUT-based Gaussian RBF on Altera Modelsim for the implementation on Altera Cyclone V (5CSXFC6D6F31C6) FPGA. The results, alongside the proposed methods in this chapter, are shown in Table 6.11. As shown, the polynomial approximation method requires 6 DSPs for 8 and 12 bits resolution while our multiplier-based solution requires just a single DSP. Furthermore, the multi-clock and single-clock solutions are scalable for higher resolution as compared to LUT solution.

Table 6.11: FPGA logic utilisation of the proposed designs in relation to previous works using various activation function block array configurations

Method	8 bits			12 bits		
	ALM	FF	DSP	ALM	FF	DSP
6th order Polynomial	26	-	6	38	-	6
LUT	22	8	-	234	12	-
Multi-clock Solution (proposed)	28	25	-	38	34	-
Single-cycle Solution (proposed)	103	-	-	136	-	-
Multiplier Solution (proposed)	10	-	1	14	-	1

Finally, we created an RBF network structure of 15 hidden neurons and one output neuron to solve

the SinE function approximation problem. We simulated the result of this network on Altera Modelsim. We recorded in Table 6.12 the resource utilisation of the network with different Gaussian RBF implementations on Altera Cyclone V (5CSXFC6D6F31C6) FPGA.

Table 6.12: Resource Utilisation on SinE dataset

Method	ALM	FF	DSP
Poly6	491	-	90
Multi-clock Solution (proposed)	393	235	18
Single-cycle Solution (proposed)	1316	-	18
Multiplier Solution (proposed)	253	-	30

6.9 Conclusion

In this chapter, we proposed a computationally efficient and effective RBF kernel. This novel RBF kernel improves the training time without any detriment to performance accuracy. We also recorded a consistent reduction in the number of RBF kernels required when using our function as to the Gaussian function. Two experiments were performed on four benchmarks and showed the generalisability of our function as well as convergence speed. On average, we recorded a speedup in training time of about 8% and a decrease in the numbers of neurons to 10%. We see a consistent speed up in the kernel transformation for SVM training using our proposed computationally efficient kernel as compared to the Gaussian kernel. Finally, we recorded significant resource savings with our method when compared to other methods found in the literature.

Chapter 7

Resource Efficient Implementation of Machine Learning Models

Abstract

In this chapter, the efficient implementation of feedforward and recurrent neural network models are explored. Long Short Term Memory (LSTM) is a computationally intensive, resource, and power-hungry type of recurrent neural network. We propose a solution that simultaneously computes a symmetric activation function with an integrated scaling functionality. This effectively eliminates two of the three element-wise multipliers in an LSTM cell. Also, this built-in scaling requires no additional computation time because it is integrated within the computation of the nonlinear mapping. This approach replaces the need to compute several Tanh activation functions and element-wise multipliers separately. This brings significant benefits to custom hardware in terms of silicon area and power consumption. A resource-efficient approximate multiplier is also proposed to eliminate the third element-wise multiplier and potentially replace the resource-hungry multipliers. The proposed approximate multiplier is useful in quantised neural networks due to its inherent quantised property. We empirically show that applying the proposed multiplier results in comparable performance accuracy with baseline. We evaluate the performance accuracy of our method using LSTMs and GRUs models on various problem areas. We demonstrate that our method achieves competitive results with negligible loss of performance. On the hardware side, we present custom hardware implementation of LSTM and GRU cells with our method compared with other benchmarks. Ultimately, we show that LSTMs with our method can achieve up to 3.5x resource footprint saving compared to the hard activation implementation.

7.1 Introduction

The two main types of neural networks based on connection are the Feed Forward Neural Networks (FFNN) and the Recurrent Neural Networks (RNN). In this chapter we discuss two ways of resource efficient implementation of these two networks.

7.1.1 Recurrent Neural Networks

Long Short Term Memory [164] and Gated Recurrent Unit (GRU) [165] are RNN models achieving state-of-the-art accuracy in several applications. Authors in [165] proposed GRU architecture which is a modification to the LSTM RNN. The GRU, like the LSTM, solves the vanishing gradient problem using an update and reset gate. Due to the simpler gating structure of the GRU model, reduced number of gates and thereby parameters, it is considered to be an efficient alternative to the LSTM RNN. LSTM requires a large number of parameters and high computational complexity. The computational cost and size of LSTM models continue to grow to achieve better model accuracy [166]. This will make the implementation of LSTM models on embedded devices and hardware with finite on-chip resources difficult to achieve. Latency and resource utilisation reduction will contribute immensely to efficient hardware implementation of LSTM. In an LSTM inference accelerator, there are two main operations contributing to the high resource utilisation [21, 166], namely (i) matrix-vector multiplication and (ii) element-wise operations (consisting of three element-wise multipliers and five transcendental activation functions). Multipliers consume the most space and are power-hungry arithmetic operators of the hardware implementation of any neural network models [24].

FPGA implementation of LSTM has been explored in literature [21, 167–169]. LSTM’s computational intensity has led to various ways of efficient implementation of LSTM. Weight pruning and compression are techniques that lower memory requirements and reduce complexity [21]. Other complexity reduction methods are data representation and multiplier reduction through sparse LSTM. In literature, data representation in LSTM models go from 32-bits to binary [169]. Commonly, the activations and weights are represented with 8-bits or 16-bits [21, 167, 168]. The approximate multiplier [170, 171] is on the increase in hardware neural networks. Two popular works in literature have developed LSTM FPGA inference engine, namely C-LSTM [172] and ESE [21]. The authors in [21] build an Efficient Speech Recognition Engine (ESE) with sparse LSTM on FPGA. In the model, 16 multipliers were instantiated for element-wise multiplications per channel (total channel is 32).

Several methods have been proposed in the literature to reduce the computational time and resource

Table 7.1: LSTM Equations for the Conventional and proposed method.

Conventional Equations [164]	Proposed Equations
$i_t = \sigma(W_{xi}X_t + W_{hi}h_{t-1} + b_i)$	$i_t = \text{L_S}(W_{xi}X_t + W_{hi}h_{t-1} + b_i)$
$f_t = \sigma(W_{xf}X_t + W_{hf}h_{t-1} + b_f)$	$f_t = \text{L_S}(W_{xf}X_t + W_{hf}h_{t-1} + b_f)$
$o_t = \sigma(W_{xo}X_t + W_{ho}h_{t-1} + b_o)$	$o_t = \text{L_S}(W_{xo}X_t + W_{ho}h_{t-1} + b_o)$
$\tilde{c}_t = \tanh(W_{xc}X_t + W_{hc}h_{t-1} + b_c)$	$n_t = W_{xc}X_t + W_{hc}h_{t-1} + b_c$
$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$	$c_t = \text{QSU}(f_t, c_{t-1}) + \text{gated_activation}(n_t, i_t)$
$h_t = o_t \odot \tanh(c_t)$	$h_t = \text{gated_activation}(c_t, o_t)$

utilisation of the matrix-vector multiplication operations [21, 172]. Authors in [173] propose the use of memoisation optimisation to eliminate three out of the six matrix vector multiplications present in GRU computation. The element-wise multiplication and activation functions account for about 30% of the overall runtime of a 128 hidden unit size GRU computation carried out on a 2.3 GHz Intel Xeon E5-2699v3 server [173]. The activation function of choice for LSTM and GRU networks is the sigmoid and Tanh functions. Unfortunately, the exponentiation and floating-point divisions present in these activation functions consume a large number of hardware resources. Considering the computation resource constraints of hardware platforms, researchers over the years have proposed different ways in which these activation functions can be implemented with a minimal trade-off between performance accuracy and resource consumption [172]. This has led to the use of computationally efficient piecewise linear replacements such as hard Tanh (defined mathematically as $\max(-1.0, \min(1.0, x))$) and sigmoid ($\max(0.0, \min(1.0, (x + 2.0)/4.0))$) to replace the soft transcendental functions. In summary, the hardware implementation of transcendental activation functions for LSTM models can be broadly categorised into two - the use of piecewise linear approximation method [167], [174], [175] and the use of lookup table [21], [176]. On the other hand, the element-wise multiplication has received little or no contribution to effective implementation in literature.

The architecture of LSTM cell [164] without peep hole connections is shown in Figure 7.1a. This cell accepts an input sequence $X = (x_1; x_2; x_3; x_4; \dots; x_T)$ (each of x_t is a vector corresponding to time t) and produces an output sequence $Y = (y_1; y_2; y_3; y_4; \dots; y_T)$. Conventional LSTM cell is characterised by the equations in column 1 of Table 7.1. Column 1 of Table 7.2 shows the equations of a standard GRU cell. At time step t , x_t is the input vector, z_t is the update gate vector, r_t is the reset gate vector, h_t is the hidden state and output vector, W_z and W_r are the trainable and recurrent weight matrices for the update and reset gates, respectively [138].

The activation functions are applied to four gates ($i_t, f_t, o_t, \tilde{c}_t$) and also at the computation of the output h_t for an LSTM cell. The five activation functions are applied element-wise to vectors of size H.

Table 7.2: GRU Equations for the Conventional and proposed method.

Conventional Equations [165]	Proposed Equations
$z_t = \sigma(W_z X_t + W_z h_{t-1})$	$z_t = \text{L.S}(W_z X_t + W_z h_{t-1})$
$r_t = \sigma(W_r X_t + W_r h_{t-1})$	$r_t = \text{L.S}(W_r X_t + W_r h_{t-1})$
$\tilde{h}_t = \tanh(W(r_t \odot h_{t-1}) + W X_t)$	$n_t = W(\text{QSU}(r_t, h_{t-1})) + W X_t$
$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$	$h_t = \text{QSU}((1 - z_t), h_{t-1}) + \text{gated_activation}(n_t, z_t)$

If $H = 128$ hidden units, $5 \times H = 640$ activations need to be evaluated. In the absence of a dedicated exponential unit, an exponent computation is iterative and will possibly consume several multiplication cycles. Furthermore, assuming D is the dimension of x_t (as well as h_t and c_t), the number of weights is $8 \times D^2$. Thus, at each inference step, the number of operations in the $M \times V$ is $8 \times D^2$, and $8 \times D$ for the element-wise operators [168]. Given that matrix-vector operations contribute largely to latency and resources, methods described in [21, 168, 172] offer high throughput but rely on uninterrupted dataflow. Element-wise operations will generally be handled separately, and hence effort at reducing their latency and resource utilisation is well spent if this operation does not impact the dataflow.

7.1.2 Feed Forward Neural Networks

The training and inferencing phase of neural networks are faced with bottlenecks such as computational complexity, large memory requirements, and the need for power-hungry GPUs. Neural network operations performed on a GPU typically have a large memory footprint that is too large for FPGAs and portable embedded devices. Machine learning algorithms are iterative, complex, and time consuming. The training algorithm, as well as the inference engine, is characterised with being reliant on requiring large number of arithmetic multiplications. Multiplication accounts for more than 70% operation of the training and inferencing of neural networks [177]. However, on hardware such as ASIC and FPGAs, multipliers are the most power and space-hungry arithmetic operators of the hardware implementation of neural networks. The possibility of using reduced multiplier precision for neural network operations have been heavily explored in the literature [24]. Recently, quantised neural networks have become highly in demand to meet the ever growing deep network architectures [39, 178, 179].

Quantisation is the process of converting the floating point values in a machine learning model to a fixed point representation. There are several quantisation fixed point representations that have been explored in literature ranging from binary to 32-bit [41–43, 180]. Authors in [181] and [180] propose the use of binary $(-1, 1)$ and ternary $(-1, 0 + 1)$ weights respectively. Activations and gradients and other computationa are kept at 32-bits. The work in [180, 182, 183] propose a framework that uses 8-bits for the whole model. The 8-bit data representations were shown to achieve similar performance

accuracy with the full floating point precision. However, binary and ternary data representation shows a little degradation in accuracy [180]. Furthermore, as described in [184], 2-bit and 4-bit fixed point quantisations show better hardware trade-off on MNIST and CIFAR-10 datasets while 4-bit and 8-bit provide the best trade-off in large-scale tasks like AlexNet on ImageNet dataset. On hardware, binary neural networks have been shown to not necessarily provide efficient resource utilisation due to large amount of parameters required for binary neural networks to achieve the same level of model accuracy with high precision alternatives [184].

Authors in [24] show that the cost of a fixed point multiplier varies as the square of the precision of its operands. Therefore, the use of limited precision to full precision on hardware offers several merits such as area reduction in terms of memory requirements and arithmetic multiplication, reduction in latency, and power consumption. The use of low precision, fixed point representation for neural network values have led to elimination of multipliers or reduced precision multiplication. For example, binary and ternary networks use shift, addition, XNOR, and population count to replace the resource-hungry multiplication. However 4-bit and higher data representation will still require a multiplier. A 9×9 fixed point Xilinx Virtex-6 multiplier costs 115 LUTs and 110 registers and produces results after four clock cycles [185]. An 8×8 fixed point representation will use just a little bit lower and a 4×4 will use approximately 55 LUTs and 50 registers.

We propose two entities to address a more efficient implementation of the feed forward and recurrent neural networks: the Gated Activation and the Quantised Scaling Unit (QSU). The Gated Activation is a nonlinear activation function that inherently incorporates a scaling capability. This eliminates the need for a dedicated element-wise multiplier in a RNN architecture. The QSU eliminates the element-wise multiplier required by the forget gate (f_t) and the 'previous' memory cell (c_{t-1}) in an LSTM cell. The QSU also has the potential to be used for matrix-vector multiplication for quantised neural network architectures. Figures 7.1b and 7.2b show LSTM and GRU cells modified with our proposed entities, and column 2 of Tables 7.1 and 7.2 lists the corresponding equations. The following are the contributions of this chapter:

- Elimination of element-wise multiplications in the LSTM and GRU cells.
- Resource efficient quantised scaling unit to replace the computationally expensive multipliers.
- The efficient implementation of the LSTM cell achieves about a $3.5\times$ reduction in resource footprint when compared with conventional methods.

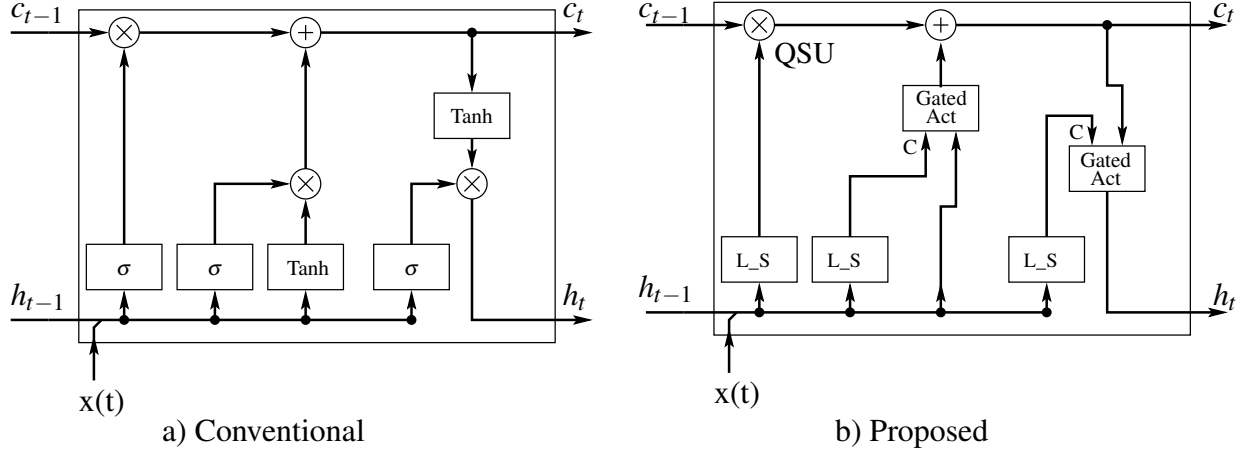


Figure 7.1: Difference between conventional and proposed LSTM cell with the elimination of element-wise multiplication. (QSU: Quantised Scaling Unit, L_S: Log_SQNL, Gated Act: Gated Activation).

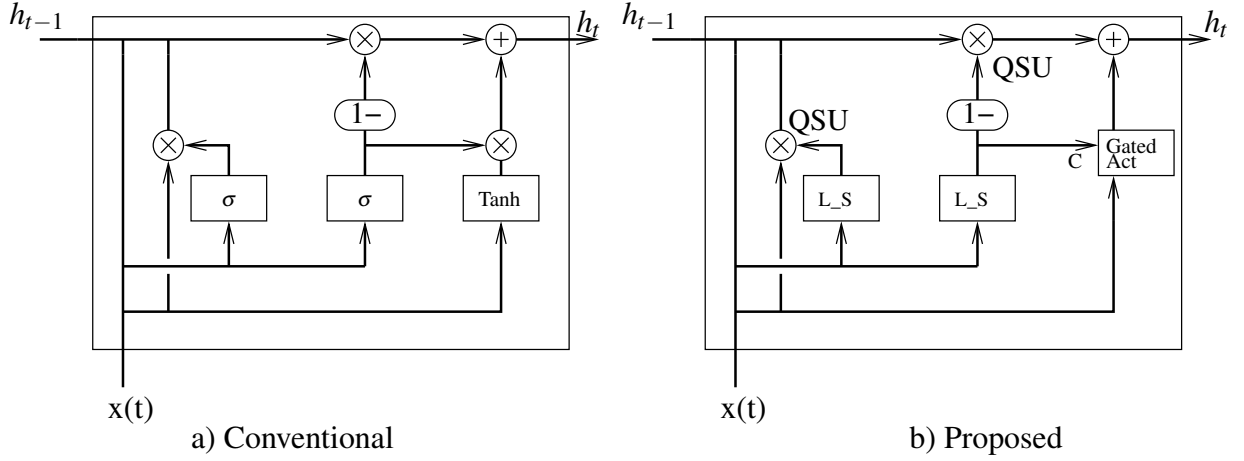


Figure 7.2: Difference between conventional and proposed GRU cell with the elimination of element-wise multiplication. (QSU: Quantised Scaling Unit, L_S: Log_SQNL, Gated Act: Gated Activation).

7.2 Concepts

In this section, we present the proposed methods of eliminating multiplication in feed forward and recurrent neural networks.

7.2.1 Gated Activation

The proposed method uses a hard nonlinearity and a time average to produce the result for element-wise multiplication between an input and a symmetric nonlinearity, i.e., $k \times G(n)$. The core of our implementation is based on a saturating adder and a non-repeating sequence. The underlying formulation is given

in Equation 7.1.

$$f(n, C) = \frac{1}{N} \sum_{k=1}^N ((n + U(k))_C - U(k))_M = n \mp L(n, C) \quad (7.1)$$

In Equation 7.1, $f(n, C)$ is the nonlinear mapping of an integer input n and C is the gate coefficient (i.e., the sigmoid output). Within the algorithm, the saturation levels of the adder are $\pm C$. With an R bit binary resolution, $-2^{R-1} \leq n \leq 2^{R-1} - 1$ and $0 \leq C < 2^{R-2}$. This makes $\max(f(n, C)) = C$ and $\min(f(n, C)) = -C$. Finally, $U(k) = \{-U_{MAX} \cdots U_{MAX}\}$ is the monotonic, non-repeating sequence of length N with $U_{MAX} = 2^{R-2}$. Note $\max(C) = U_{MAX}$.

The input is repeatedly summed with each element of the sequence and averaged. Without any saturation, $(n + U(k)) - U(k) = n$ and $f(n) = n$. However, any saturation in $(n + U(k))$ will result in $(n + U(k)) - U(k) \neq n$. The saturation constitutes a loss and this loss is quadratic. The average loss and $f(n, C)$ have been analytically determined. Figures 7.3-7.6 plot the loss at different values of $U(k)$ across the positive range of the input n . The mapping is symmetric for the negative range.

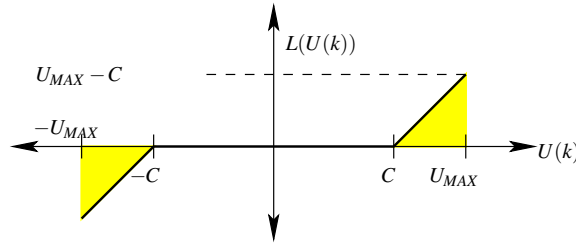


Figure 7.3: Gated Activation: loss evaluated at $n = 0$.

At $n = 0$, $(n + U(k))_C$ will only saturate when $U(k) > C$ or $U(k) < -C$. The saturation zones are shown as yellow triangles in Figure 7.3. For $-C \leq U(k) \leq C$ the loss is zero while for $U(k) > C$ the loss is positive and for $U(k) < -C$ the loss is negative. If $\Delta = U_{MAX} - C$

$$L(n, C) = \frac{1}{2U_{MAX}} \left[-\frac{1}{2}\Delta^2 + 0 + \frac{1}{2}\Delta^2 \right] = 0$$

$$f(n, C) = n$$

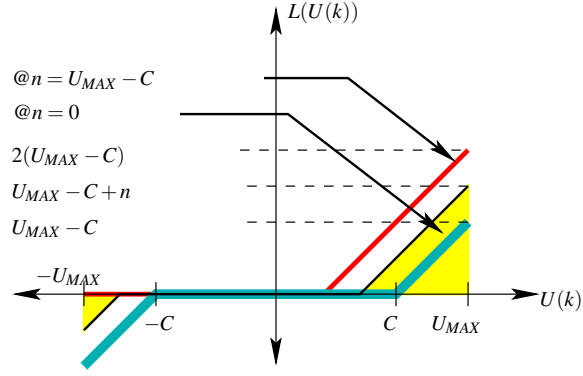


Figure 7.4: Gated Activation: loss evaluated at $0 \leq n \leq U_{MAX} - C$.

Figure 7.4: For $0 \leq n \leq U_{MAX} - C$, the saturation is asymmetrical. At $n = U_{MAX} - C$, none of the negative elements of $U(k)$ result in saturation. If $\Delta = U_{MAX} - C$,

$$L(n, C) = \frac{1}{2U_{MAX}} \left[-\frac{1}{2}(\Delta - n)^2 + \frac{1}{2}(\Delta + n)^2 \right]$$

$$f(n, C) = n \frac{C}{U_{MAX}}$$

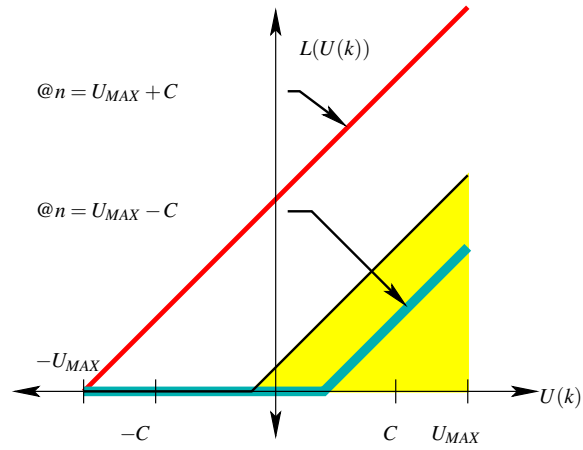


Figure 7.5: Gated Activation: loss evaluated at $U_{MAX} - C \leq n \leq U_{MAX} + C$.

For $U_{MAX} - C \leq n \leq U_{MAX} + C$ and $\Delta = U_{MAX} - C$, the loss from saturation is represented by the

triangular zone shown in Figure 7.5.

$$L(n, C) = \frac{1}{2U_{MAX}} \left[\frac{1}{2} (U_{MAX} - C + n)^2 \right]$$

$$f(n, C) = n - \frac{(\Delta + n)^2}{4U_{MAX}}$$

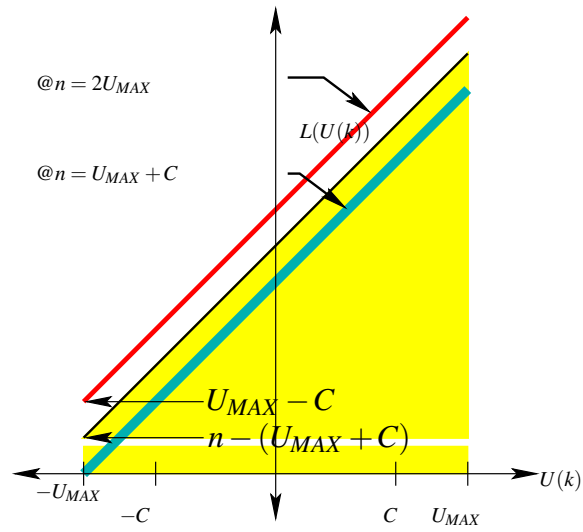


Figure 7.6: Gated Activation: loss evaluated at $U_{MAX} + C \leq n \leq 2U_{MAX}$.

For $U_{MAX} + C \leq n \leq 2U_{MAX}$, the loss is represented by a rectangular and a triangular zone as shown in Figure 7.6.

$$L(n, C) = \frac{[2U_{MAX}(n - (U_{MAX} + C)) + 2U_{MAX}^2]}{2U_{MAX}}$$

$$= n - C$$

$$f(n, C) = n - (n - C) = C$$

If $\Delta = U_{MAX} - C$ then

$$f(n, C) = \begin{cases} -C & : n < -(U_{MAX} + C) \\ n + \frac{(-\Delta + n)^2}{4U_{MAX}} & : -(U_{MAX} + C) \leq n < -\Delta \\ n \frac{C}{U_{MAX}} & : -\Delta \leq n \leq \Delta \\ n - \frac{(\Delta + n)^2}{4U_{MAX}} & : \Delta < n \leq (U_{MAX} + C) \\ C & : n > (U_{MAX} + C) \end{cases} \quad (7.2)$$

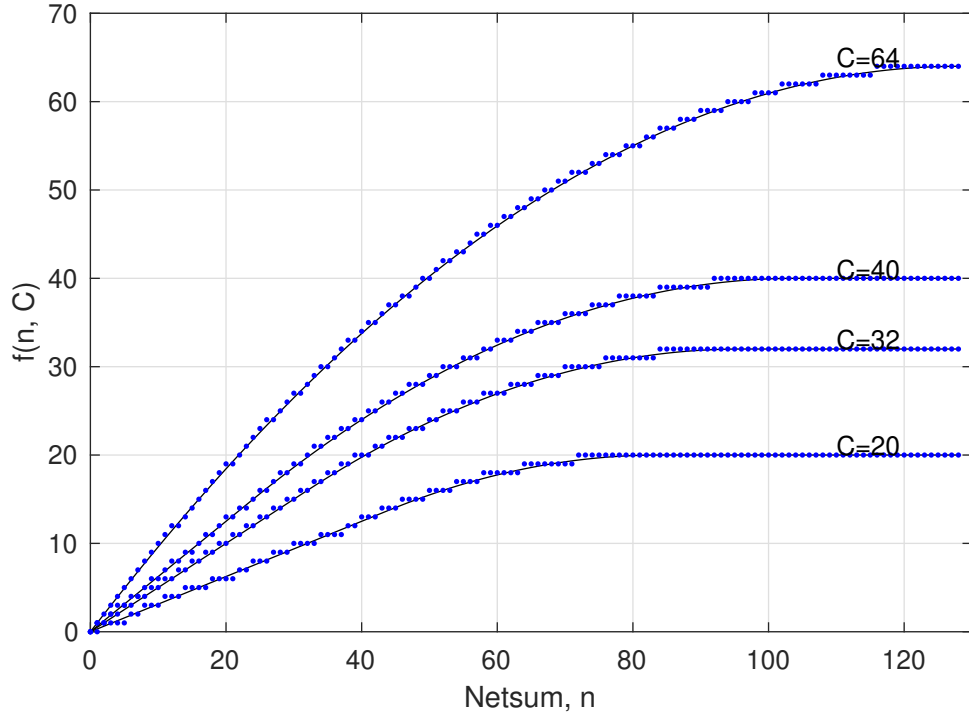
At the limit $C = 2^{R-2} = U_{MAX}$ and if $G(n) = f(n, 2^{R-2})$ then Equation 7.2 gives $f(n, C) \approx k \times G(n)$, where $k = C/U_{MAX}$. The mappings of $f(n, C)$, with $R = 8$ and $N = 8$, for different values of n and C are shown in Figure 7.7a.

The algorithm, as modelled by Equation 7.2, exhibits a small error from the ideal $k \times G(n)$. The plot with $C = 64$ corresponds to $G(n) = f(n, 64)$. With $C = 40$, $f(n, 40) \approx \frac{40}{64}G(n)$. Specifically at $n = 40$, $G(40) = 33.75$, $f(40, 40) = 24$ but $\frac{40}{64} \times 33.75 = 21.09$, and hence the gated activation exhibits an error of -2.91 . The error can be determined by Equation 7.3.

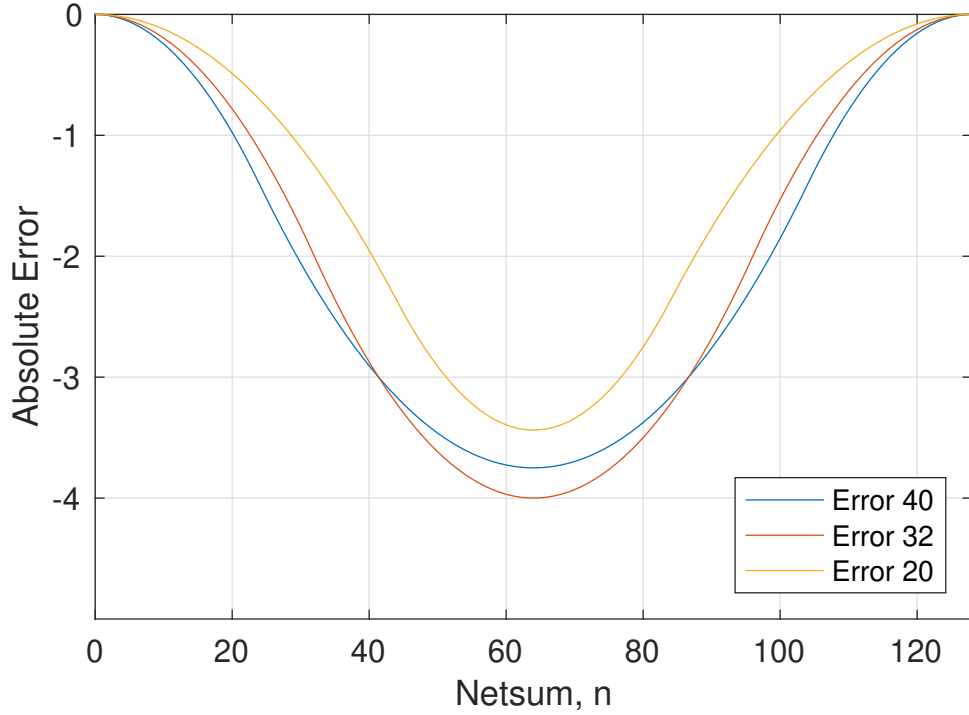
$$E(n, C) = \frac{C}{U_{MAX}}G(n) - \left(n + \frac{-(\Delta + n)^2}{4U_{MAX}} \right) \quad (7.3)$$

From Equation 7.3, it can be shown that the maximum error is at $n = \pm 2^{R-2}, \forall C$. It can also be shown that the error is maximum at $C = 2^{R-3}$. For $R = 8$, maximum error at $f(64, 32)$ is 4.0 (3.13%). The complete error profile is shown in 7.7b. The figure shows that the maximum error is -4 and this occurs at $n = 64$. In addition the worst case error occurs when $C = 32$.

In summary, $f(n, C)$ replaces both the Tanh and the element-wise multiplier. These functions are both very resource efficient when compared to their exponential based counterparts. The deviations from the ideal behaviour are small and should not influence the network performance. The difference between a conventional LSTM cell and the proposed LSTM cell solution is illustrated in Figure 7.1. Figure 7.2 shows the difference between a conventional GRU cell and the proposed GRU cell solution. On CPUs, these functions will also be faster than exponential based functions.



(a) Gated Activations for $R = 8, N = 4$



(b) Error between $k \times G(n)$ and $f(n, C)$

Figure 7.7: a) Gated Activation, $R = 8, N = 8$: It shows that at a constant input, say netsum = 40 and $C = 64$, the output is $f(40, 64) = 33.75$. Thus, $f(40, 40) = 24 \approx \frac{40}{64} \times f(40, 64) = 21.09$. b) This shows the error varies with both n and C . The maximum error is always at $n = 2^{R-2}$, and $C = 2^{R-3}$, With $R = 8$, the maximum error is binary value of 4.

Impact of N

The length of the sequence $U(k)$ has a direct influence on the throughput. Reducing N will imply fewer clocks are required. A good choice of N is either 4 or 8. With $R = 8$, $N = 4$ causes a deviation (difference between full precision and a pragmatic N) of only ± 1 bit and when $N = 8$, the deviation is less than one bit. Figure 7.8 plots the deviations from the ideal mapping as a consequence of changing R and N .

For $R = 8$, the deviation profile in Figure 7.8 a) and c) show that an increase in N results in many small deviations from the ideal. The corresponding frequency distributions show that 30% of the values would exhibit a single bit error. With $N = 8$, the deviations are smaller than ± 0.5 , and hence will produce no errors, suggesting that $N > 8$ would serve no benefit. With $N = 8$, if R changes from 8 to 12, the deviation increases ratiometrically and this corresponds to a maximum of ± 4 -bit error, with at least half the values exhibiting no errors.

The results of our experiments will show that these errors are inconsequential because the learning algorithms factor these errors in the learning differentials. The benefits of a reduced resource usage with the elimination of the multipliers far outweighs the small errors.

7.2.2 Quantised Scaling Unit (QSU)

The QSU calculates $i \times j$ where i and j are binary integers and j is always positive. Internally, the QSU multiplier implementation successively adds the i input to itself j times. Our measurements show that the QSU takes fewer resources than a Booths Radix-4 implementation. The following shows the operation of a QSU:

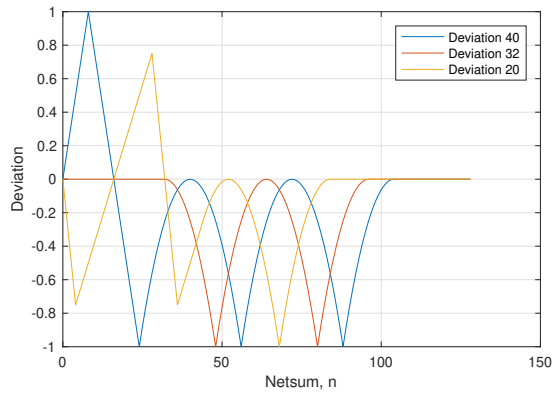
Definition 7.2.1. An oversampling rate is defined as 2^O where O is an integer constant ($0 > O \leq 7$). This oversampling rate determines the clock cycles and the quantisation level of the multiplicand/scale j .

Definition 7.2.2. The multiplicand is defined as a quantised level of $2^{(O+1)}$. Therefore input $j = j/2^{(O+1)}$.

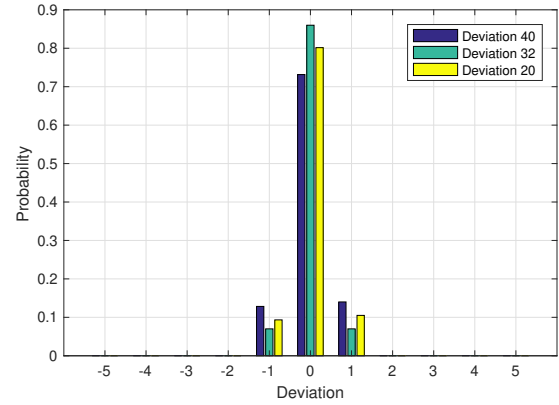
Definition 7.2.3. The result of $i \times j$ is available at 2^O clocks.

Due to the simplicity of the QSU, it is best explained numerically. For $O = 3$, and a resolution of 8-bits for multiplier value i and 3-bits for multiplicand value j .

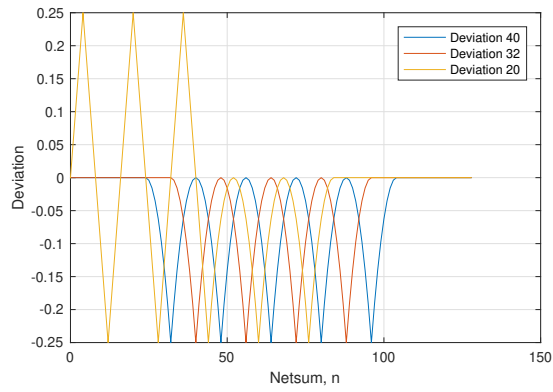
Example 7.2.4. Multiplier value $i = 12$ and multiplicand value $j = 3$ can be obtained by: three additions, and a sum and accumulate (SAC) block controlled by a counter that counts from 0 to $2^O - 1$.



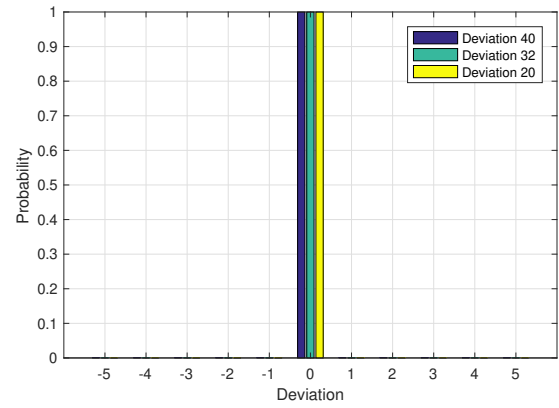
(a) Deviation, $R = 8$, $N = 4$



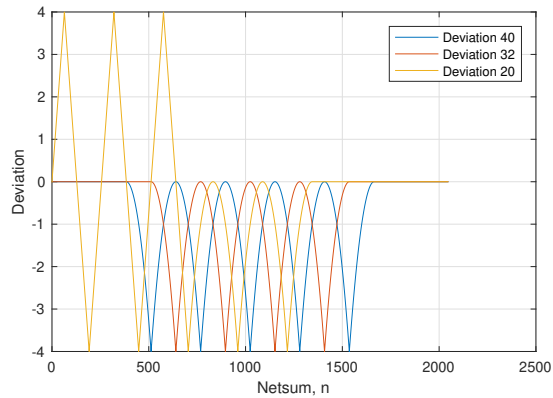
(b) Distribution of the deviations, $R = 8$, $N = 4$



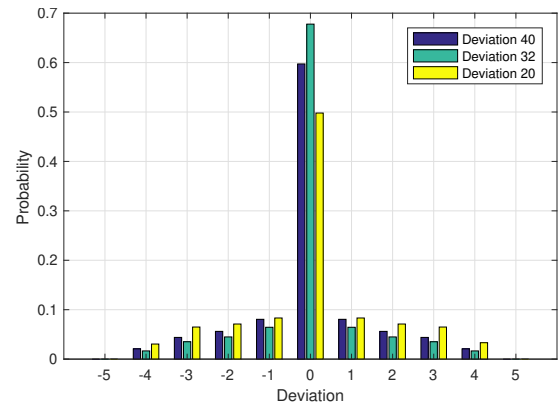
(c) Deviations, $R = 8$, $N = 8$



(d) Distribution of the deviations, $R = 8$, $N = 8$



(e) Deviations, $R = 12$, $N = 8$



(f) Distribution of the deviations, $R = 12$, $N = 8$

Figure 7.8: Profile of deviation from ideal $f(n, C)$ a) Plots the deviation for the positive input range for different values of C . b) Plots the frequency distribution of the deviation as a probability.

$$12 \times 3 = 12 + 12 + 12 + 0 + 0 + 0 + 0 + 0$$

Therefore, Example 7.2.4 shows that while i can represent signed or fixed point numbers, the maximum range of j is $[0, 7]$.

If the width of j is increased by 1, then, the maximum range of j increases to $[0, 15]$. This would require $O = 4$ and thus $2^O = 16$ clocks. However, we present an alternative solution that will circumvent this clock cycle doubling.

Example 7.2.5. Consider a multiplier value $i = 12$ and multiplicand value $j = 11$. Since $11_{10} = 1011_2 = 1000_2 + 0011_2$. Here, the most significant bit (MSB) is 1, and hence the multiplication can be expressed as

$$\begin{aligned} 12_{10} \times 11_{10} &= 12_{10} \times (1000_2 + 0011_2) \\ 12_{10} \times (1000_2 + 0011_2) &= 12_{10} \times 1000_2 + 12_{10} \times 0011_2 \end{aligned}$$

The first term $12_{10} \times 1000_2$ is a left shift by three while the second term, $12_{10} \times 0011_2$, is a sum and accumulate of 12. This will be achieved in three clocks. Thus, the SAC block can be preloaded with the $12_{10} \times 1000_2 = 96_{10}$ prior to initiating the sum and accumulate block. Therefore, Example 2 is expressed as $12 \times 11 = [(96) + 11 + 11 + 11 + 0 + 0 + 0 + 0 + 0]$

Remark 7.2.6. Since $12_{10} \times 1000_2$ is an arithmetic left shift by 3-bits, it can be achieved in a combinatorial logic and thus takes minimal time and resources.

Remark 7.2.7. The resolution of the multiplicand j can be further increased by 1-bit to represent signed multipliers. The sign bit of j can be used to modify the multiplier i by simply negating i prior to the sum and accumulate operation.

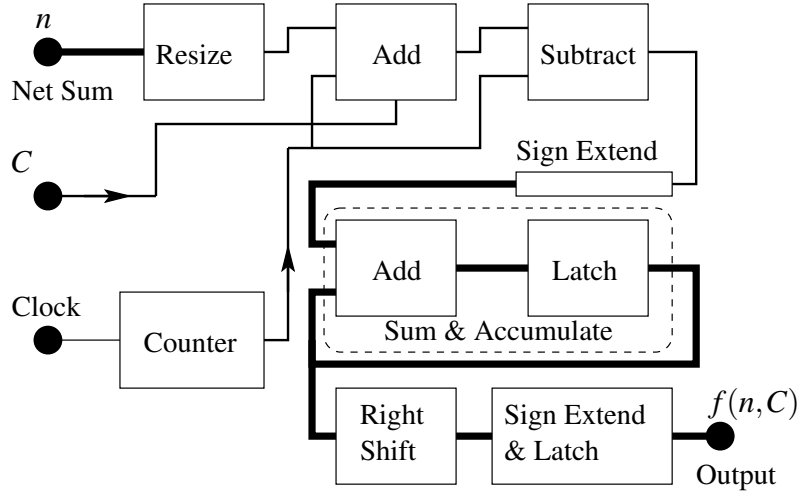
Example 7.2.8. For signed multiplication;

$$i \times -11 = (-i \times 8) + (-i) + (-i) + (-i) + 0 + 0 + 0 + 0 + 0$$

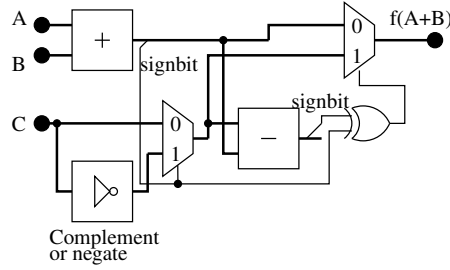
The binary resolution of signal c_{t-1} present in an LSTM cell may be larger than R but the effective binary resolution of $s(n)$ is $R - 2$ because $0 \leq s(n) \leq 2^{R-2}$ (where $s(n)$ is the Log_SQNL function). This reduced resolution offers an opportunity to consider an alternative: the QSU.

7.3 Hardware Implementation

The gated activation and QSU methods have been implemented on an Altera Cyclone V device (5CSXFC6D6F31C6). The Quartus Prime Standard 18.1 Edition and Modelsim 10.1d are the integrated development environment on which all the hardware implementation is performed.



(a) Schematic of the hardware implementation of gated activation method.



(b) Schematic of the Parametrised Saturation Adder (A is the netsum (n), B is the counter value ($U(k)$) and C is the scaling value (C))

Figure 7.9: Gated Activation Implementation.

7.3.1 Hardware Implementation of Gated Activation

The Gated Activation requires an additional input for the gate signal and an adder with adjustable saturation. Figure 7.9a shows a complete schematic with various bus-widths highlighted. The code is parametrised, and hence the saturation of the adder and subtractor change with the resolution R . The lower Add, Latch, and Right-Shift blocks function as the filter (sum, accumulate, scale). These operate at a higher bus-width as well. Figure 7.9 b) shows a block diagram illustrating the operation of the adder block in Figure 7.9a. This parameterised saturation adder is responsible for gated activation.

The parameter $U(k)$ is implemented to generate $\{\pm 1, \pm 3, \pm 5, \pm 7\} \times 2^{R-2}/N$. This sequence is only a binary counter from zero to three with leading and trailing edge padding. Typically, the signals that generate the netsum inputs operate at larger word widths than the activation function. This is accommodated by the Resize and Sign Extend blocks. The choice of $U_{MAX} = \max(C) = 2^{R-2}$ arranges for the output range to be 50% of the input range. It provides a wide nonlinear range with a good linear section and a simple quadratic mapping equation.

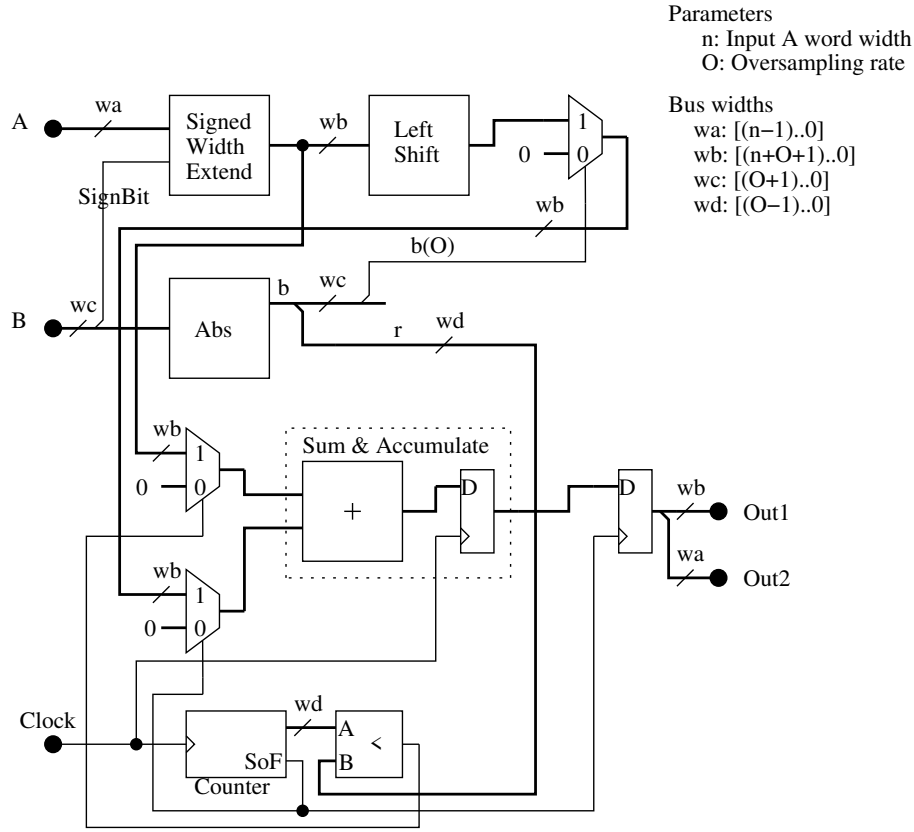


Figure 7.10: Schematic of Quantised Scaling Unit Hardware Implementation.

7.3.2 Hardware Implementation of QSU

Figure 7.10 shows a complete schematic with various components and bus-width details. The left shift operation consumes zero additional logic because only the upper bits are passed through and will be hard-wired by the compiler. Addition/subtraction operations are combinatorial, and hence extra clock edges are not required. There is an absolute block for obtaining the modulus of the multiplicand, a sum and accumulate block, and several multiplexers.

7.4 Experimental Results

In this section, we show two results, firstly, the performance accuracy of our proposed methods in software simulations using different datasets. Secondly, we discuss the hardware resource utilisation and usability of our proposed methods.

Table 7.3: Character-level perplexity on the US baby name dataset. There is no difference in perplexity when using the baseline model and Gated Activation method (small is better).

Method	Validation Set	Test Set
Baseline [187]	1.55	1.63
Hard Activation	1.56	1.64
Gated Activation (proposed)	1.55	1.63

7.4.1 LSTM Software Experimental Results

We show that the elimination of the element-wise multiplier obtains comparable prediction accuracy. We use the baseline LSTM as a benchmark on language modelling, character recognition, and image recognition datasets. We use predefined models available for each of these datasets and replace the element-wise multipliers and the activation functions. For all the experiments carried out in this section, the baseline refers to using the equations defined in column 1 of Table 7.1. The activation function is the sigmoid and Tanh. The hard activation method refers to replacing the activation with the computationally efficient hard sigmoid and hard Tanh. The gated activation method refers to our proposed method (using Equation 7.2). For a fair comparison, all the experiments carried out in this section use models and hyperparameters that were designed for Tanh and sigmoid. We replaced these functions with hard activations and our proposed method.

Language Modelling

We conduct character-level and word-level prediction experiments to show the usability of our proposed method on language modelling problems. We use perplexity (PPL) as the measure to evaluate the performance of an algorithm for language modelling (the lower, the better), defined as $PPL = \exp(\frac{L}{T})$, where L is the cross-entropy loss and T is the number of tokens in the test set.

- **Character Level Language Modelling** The character-level language modelling is performed on the US Baby’s First Names [186]. This dataset contains 1825k names. The full data and model description is available at [187]. The results of using the baseline, hard activations, and our method is shown in Table 7.3.
- **Word Level Language Modelling** We conduct word-level prediction experiments on the Penn Tree Bank (PTB) corpus [188] and the WikiText-2 corpus [189].
 - PTB corpus dataset: The PTB corpus consists of 929k training words, 73k validation words,

Table 7.4: Word-level perplexity on the Penn Tree Bank dataset. Using Gated Activation shows a negligible increase in the perplexity on this task (small is better).

Method	Validation Set	Test Set
Baseline Non-regularised model [16]	120.7	114.5
Hard Activation	122.92	117.84
Gated Activation (proposed)	121.4	115.2

Table 7.5: Word-level perplexity on the WikiText-2. Using Gated Activation shows a negligible increase in the perplexity on this task (small is better).

Method	Validation Set	Test Set
Baseline Non-regularised model [16]	141.34	133.19
Hard Activation	151.31	143.71
Gated Activation (proposed)	142.24	135.16

and 82k test words, and it has 10k words in its vocabulary. We adopted the non-regularised small LSTM model defined in [16]. We use the small model which consists of 200 hidden units per layer and the same hyperparameters. The results are shown in Table 7.4.

- **WikiText-2 corpus:** The WikiText-2 corpus consists of 2088K training, 217K validation, and 245K test tokens, and has a vocabulary of 33K words. We adopted the non-regularised small LSTM model defined in [16]. We use the small model which consists of 200 hidden units per layer and the same hyperparameters. The results are shown in Table 7.5.

Image Classification

As a quick illustration to show that our gated activation algorithm is not limited to texts, we conduct experiments on the sequential MNIST and Fashion MNIST classification tasks.

- **Sequential MNIST** The dataset consists of a training set of 60K and a test set of 10K 28×28 gray-scale images. We used the first 50K for training and the last 10K images for validation. In every time step, we sequentially use one row of the image as the input (28×1), which results in a total of 28-time steps. We use one hidden layer of LSTM of size 128 neurons. We used a batch size of 128, epoch of 200, the learning rate of 0.001, and the RMSprop training algorithm.
- **Sequential Fashion MNIST** This dataset proposed in [190] is a direct drop-in alternative to the digit based MNIST data. It is a more challenging classification task than the simple MNIST digits

Table 7.6: Test error rate of LSTM on MNIST and Fashion MNIST. By eliminating the element-wise multiplication, there is a negligible performance degradation between conventional and proposed LSTM models validating the usability of our approach. (Results are average of five runs, and an estimated mean of accuracy with confidence of 95% is recorded)

Method	MNIST	Fashion MNIST
Baseline [191]	0.89% \pm 0.02	9.92 % \pm 0.07
Hard Activation	1.49% \pm 0.05	10.40% \pm 0.09
Gated Activation (proposed)	1.01% \pm 0.02	10.11 % \pm 0.14

data. Fashion MNIST has the same dimension and shape as the digit based MNIST. We performed the same experiments described above and use the same hyperparameters.

7.4.2 GRU Software Experimental Results

We follow the same experiment setup as in the LSTM. We conduct character-level prediction experiments to show the usability of our proposed method on language modelling problems. The character-level language modelling is performed on the dataset and GRU model defined in [192]. This dataset contains 9249 words. The results of using the baseline, hard activations, and our method is shown in Table 7.7.

Table 7.7: Character-level Accuracy. There is no difference in accuracy when using the baseline model and Gated Activation method.

Method	Accuracy (%)
Baseline [192]	85.9
Hard Activation	84.9
Gated Activation (proposed)	85.8

7.4.3 FNN Software Experimental Results

We carried out experiments to show the effect our QSU has on the performance accuracy of selected datasets. We apply our QSU to four binary classification problems and Thyroid dataset which is a multi-classification problem with three output elements. The High Time Resolution Universe Survey (HTRU2) [193] dataset has 16,259 samples with 1,639 features of real pulsars. The Covtype [194] is made up of 581012 samples with 54 features. The Adult [194] dataset has 13 features and 32,561 samples. Epsilon [195] contains 100,000 samples with 2,000 features. The Thyroid [194] dataset is

made up of 7200 patients with 21 patient attributes. We quantised the inputs and the activations during training to five bits. We compare the quantised method to the full precision implementation. Table 7.8 shows the performance accuracy between full and quantised precisions. Our results demonstrate that our quantisation method yield comparable accuracies to full precision representation.

Table 7.8: Accuracy using floating point full precision and INT5 quantisation.

Dataset	Precision	Accuracy (%)
Thyroid	Full	93.65
	Quantised	93.01
HTRU2	Full	92.68
	Quantised	91.28
Adult	Full	85.08
	Quantised	84.90
Covtype	Full	100
	Quantised	100
Epsilon	Full	96.95
	Quantised	96.93

7.4.4 Resource Utilisation of QSU and Standard Multiplier

We compared the resource utilisation of a standard Booth’s multiplier with our proposed QSU. Since one of the inputs of our QSU is quantised, this was taken into consideration during the comparison with the Booth’s multiplier. The resource utilisation is shown in Table 7.9. The Booth’s Radix-4 multiplier uses approximately $85/55 = 3\times$ and $55/30 = 1.8\times$ more ALMs and FFs, respectively than the QSU.

Table 7.9: Resource utilisation of QSU and Standard Booth’s Multiplier (for an 8-bit system: input A is 8 bits, input B is 6 bits)

Method	8 bits		16 bits	
	ALM	FF	ALM	FF
Booth’s Multiplier	85	55	129	79
QSU	28	30	44	47

7.4.5 LSTM Hardware Experimental Results and Discussion

The resource utilisation of the LSTM cell is carried out on the Altera Cyclone V device (5csx6c6d6f31c6). The synthesis results have been obtained using Quartus Prime version 18.1 for the Cyclone V device. For this analysis, we built an LSTM cell incorporating just the gates and element-wise multiplication operation. The resource count of this cell shows the difference between using hard nonlinearities, approx-

imate soft nonlinearities, and the proposed gated activation The experiments performed in this section are described below

- **Conventional LSTM:** the conventional LSTM implementation requires the use of element-wise multipliers and the conventional activation functions. The sigmoid and Tanh are implemented on FPGA using the following competing methods found in the literature:
 - **Piecewise Linear Approximation :** In [172], the basic form of $ax + b$ is used to construct approximate Tanh and Sigmoid using 12 breakpoints. Each of the breakpoints are hardcoded in VHDL.
 - **Lookup table implementation of Tanh and Sigmoid** as used in [21]. The table contents have been precalculated and used in long 'WHILE LOOP' with 'IF' statements. A memory block could have been used but the size of memory blocks will vary with FPGA family and vendor. More importantly, all FPGAs internally use LUT and the use of a 'WHILE LOOP' with 'IF' statements is likely to be the optimal method. In terms of 'information' storage, the use of memory or LUTs should result in a similar gate usage.
 - **Hard Tanh and Hard Sigmoid:** hard Tanh and hard sigmoid are the computationally efficient piecewise linear replacement for Tanh and sigmoid functions, respectively. Hard Tanh is defined mathematically as $\max(-1.0, \min(1.0, x))$ and hard sigmoid is defined as $\max(0.0, \min(1.0, (x + 2.0)/4.0))$. Each breakpoints is hardcoded in VHDL using 'IF' statements.

Each of these competing methods require a multiplier. The DSP's in the Cyclone V were used to implement these multipliers. To permit a quantitative comparison, these methods have also been implemented using a custom Booths Radix-4 multiplier (in VHDL) and their resource usage has been obtained.

- **Gated Activation (proposed):** the LSTM cell with no element-wise multipliers, all the element-wise multipliers are replaced by the quantised multiplier and gated activation algorithm. The sigmoid activation function is also replaced with our proposed Log_SQNL described in Chapter 3.

The three element-wise multiplication requires three hardware multipliers. These can make use of built-in DSP blocks or coded in HDL. The Altera built-in DSP blocks have native support for up to three signal processing precision levels (three 9 x 9, two 18 x 18, or one 27 x 27 multiplier) in the same variable-precision DSP block [196]. This can be observed in Table 7.10 in which the LUT and

Table 7.10: Resource utilisation of LSTM cell

Method	8 bits			16 bits		
	ALM	FF	DSP	ALM	FF	DSP
Piecewise Activation [172]	81	-	7	107	-	7
LUT [21]	96	16	2	6,568	32	2
Hard Activation	20	-	2	53	-	2
Gated Activation (proposed)	52	55	-	150	175	-

Table 7.11: Resource utilisation of LSTM cell (Replacing DSP with Booth's Algorithm)

Method	8 bits		16 bits	
	ALM	FF	ALM	FF
Piecewise Activation	548	285	1056	530
LUT	276	148	6817	271
Hard Activation	181	150	383	191
Gated Activation (proposed)	52	55	150	175

hard activation implementations use two DSPs. Note that Table 7.10 shows the resource utilisation of the hardware LSTM cell using built-in DSP. Since the resource usage of the built-in multipliers is difficult to establish, a Booths Radix-4 was implemented in VHDL. An 8-bit Booth's Radix-4 multiplier will cost 98 ALM, 63 FF, and 196 ALM, 120 FF for 16-bit resolution. Table 7.11 tabulates the resource utilisation of the methods using Booth's multiplier. The hard activation method will consume $181/52 = 3.48 \times$ ALMs and $150/55 = 2.73 \times$ FFs than our method for an 8-bit system. The PWL method will consume $548/52 = 10.54 \times$ ALMs and $285/55 = 5.18 \times$ FF and the LUT method will consume $276/52 = 5.31 \times$ ALMs and $148/55 = 2.69 \times$ FF than our proposed method. This will increase for higher resolutions.

Throughput Analysis If the latency of the activations is T_A and that of each multiplier is T_M , the maximum latency of the LSTM cell is at the h_t terminal and can be quantified by

$$T_D = \max((T_A + T_M), (T_A + T_M)) + T_A + T_M = 2(T_A + T_M)$$

The 8-bits Booths Radix-4 algorithm takes four clocks. Thus, the Hard Tanh and the LUT will have a latency of eight clocks, while the PWL will require 16 clocks. With $N = 8$, SQLN based solutions will have a latency of 20 clocks. However, Hard Tanh requires multipliers. Each of the LSTM methods has

been implemented using our Booths multiplier, and the resource utilisation is tabulated in Table 7.11. The Hard Tanh requires 181 ALMs, and hence its throughput in terms of ALM is 181×8 vs 52×20 of the proposed method. The gated activation has a higher throughput at $R = 8$.

7.4.6 GRU Hardware Experimental Results and Discussion

The resource utilisation of the GRU cell is carried out following the description given for the LSTM cell in section 7.4.5. As expected, GRU cell consumes less resources than the LSTM cell. Table 7.12 shows the resource utilisation of our proposed method in comparison with three other methods found in literature. Overall, our method continuously use less resources. Table 7.13 shows the resource utilisation by replacing the inbuilt DSP with the Booth's Radix-4 multiplier.

Table 7.12: Resource utilisation of GRU cell

Method	8 bits			16 bits		
	ALM	FF	DSP	ALM	FF	DSP
Piecewise Activation [172]	54	-	5	94	-	5
LUT [21]	61	16	2	5225	28	2
Hard Activation	5	8	2	38	-	2
Gated Activation (proposed)	52	54	-	160	166	-

Table 7.13: Resource utilisation of GRU cell (Replacing DSP with Booth's Algorithm)

Method	8 bits		16 bits	
	ALM	FF	ALM	FF
Piecewise Activation	404	215	811	408
LUT	240	138	5337	251
Hard Activation	184	134	382	203
Gated Activation (proposed)	52	54	160	166

The 8-bits Booths Radix-4 algorithm takes four clocks. Thus, the Hard Tanh and the LUT will have a latency of four clocks, while the PWL will require eight clocks. With $N = 4$ and $N = 8$, SQNL based solutions will clearly have a higher throughput at $R = 8$ or $R = 16$.

7.4.7 FNN Hardware Experimental Results and Discussion

We build shallow quantised networks using Quartus to show the resource utilisation of the QSU based network against the standard Booth's multiplier units. Table 7.14 shows the result of the resources. QSU based neural networks achieve area savings of approximately $1.66\times$ and $1.32\times$ for ALM and FF, respectively when compared to Booth's multiplier based network architectures.

Table 7.14: Resource utilisation of Quantised Neural network models with QSU and Booth multiplier

Network	QSU		Booth's Multiplier	
	ALM	FF	ALM	FF
4-5-3	1290	1009	2181	1362
9-5-2	1945	1469	3359	2016
2-15-1	1737	1401	2887	1845

The following are the advantages of the proposed solutions: i) Gated Activation eliminates two multipliers in an LSTM cell leading to resource-efficient hardware implementation of LSTM networks. ii) Precise analytical expressions are available for CPU based inference and training engines. iii) The QSU consumes fewer resources than a standard Booths multiplier.

7.5 Conclusion

We have proposed a novel algorithm and its hardware implementation that computes a symmetric activation function with an integrated scaling functionality. The algorithm has a unique advantage in which, without using a multiplier, it also simultaneously scales the output by an external input. In an ASIC implementation of an LSTM, the non-requirement of a multiplier implies that the proposed solution outperforms the simplest function - Hard Tanh. In a conventional LSTM, this proposal can replace all five activations functions and two of the three element-wise multipliers. A closed-form expression for the mapping has been obtained, and hence CPU-based training engines can be used for offline training. We also propose a compact multiplier, the QSU, that has the potential to replace the final multiplier. The QSU capitalises on the reduced word width from the gate generators. The QSU also has the potential to replace multipliers in quantised neural networks. The entities proposed in this chapter offer options for a complete hardware solution of the LSTM cell. The proposal is resource-efficient and also offers higher throughput. We have tested our proposed algorithms on different datasets and show that our performance is comparable to the conventional LSTM and the Hard Tanh variant.

Chapter 8

Evaluation of Learnable Asymmetric Activation Functions for Deep Learning

Abstract

We have discussed several activation functions, their advantages, and usage in diverse neural network architectures. All the activation functions that have been proposed and discussed so far are fixed and not adaptable during training. Determining optimal activation function for deep neural networks is still an active area of research. The flexibility of a neural network model has been shown in the literature to increase by introducing learnable activation functions. In this chapter, we empirically investigate the effect of parameterising different regions of learnable activation functions. The parametric asymmetric square-based units are shown to achieve faster convergence when compared to fixed activation functions. Our experiments indicate that there is no significant difference in convergence speed of parameterising both the positive region and negative region of an activation function and negative region-only parameterising. Parameterising activation functions with a small number of adaptable parameters introduces the need for multipliers, division operators, and others. In this chapter, we further introduce a learnable efficient, and computationally simple Multiple Square Units (MSU) function. This function synthesises many useful properties of three already defined asymmetric square-based activation functions. This function is computationally efficient as a result of eliminating the commonly used mathematical operators such as a multiplier, floating-point division, square, logarithm, and others to introduce learnable parameters. Empirically, MSU outperforms other activation functions.

8.1 Introduction

As described in [2], the choice of activation function affects the network performance and changes learning representation. During the course of this research, we have discussed several activation functions characterised by different shapes and ranges. Apart from some properties that make some activation functions suitable or not suitable for particular tasks, picking an activation function is a trial and error process. Therefore, making activation functions as a hyperparameter to be learned during training may help shed light on the issue of the optimal activation function. The choice of activation function in the deep neural network is an active area of research [197,198]. Unlike the conventional activation functions whose parameters are fixed, parametric functions contain learnable parameters that can be learned with network weights and biases 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 activation functions [111]. In [1], parametric ReLUs were demonstrated to outperform traditional ReLUs on the ImageNet task. Furthermore, parametric ELUs were similarly demonstrated to be superior to non-parametric ELUs on the MNIST, CIFAR-10/100, and ImageNet tasks [199].

As described in the literature [1, 111, 199], the following are the advantages of parametric functions: i) help to reduce network overfitting ii) speed up training convergence iii) parametric activations achieve (marginally) higher accuracy than their non-parametric counterparts. Parametric functions, due to their learnable parameters, tend to synthesise many useful properties of other activation functions iv) improve the total response region during learning and hyperplane carving. A parametric activation function is more dynamic and capable of adapting as per the requirements of its neighboring layers [200]. The following are the notable downsides of parametric activation functions: i) the introduction of extra learnable parameters ii) the need for mathematical operators (such as addition, subtraction, multiplication, and division) to include these parameters with the activation functions. Multiplication and division are computational intensive operators both in software and hardware. Multipliers are said to be resource-hungry and computationally expensive [181].

This has motivated us to investigate the parameterised square-based asymmetric activation function in this chapter. The square asymmetric functions, discussed in Chapter 5, has an inbuilt variable that can be parameterised and give the result without the need for a multiplier. Hence, saving on resources, computing power, and latency. We also investigated the effect of parameterising different regions of an activation function.

8.2 Related Work

In this section, we review parametric activation functions by categorising them into three, based on the method for combining and deriving the activation functions.

8.2.1 Single Learnable Functions

In this category, we discuss all the activation functions as a result of combining a set of trainable parameters with standard/fixed activation function. Generally, the numbers of learnable parameters are low. Therefore, the expressive power of the activation functions in this category are limited [197].

- Parametric ReLU [1]: the parametric α is responsible for transforming the negative part as defined $\text{PReLU}_\alpha(x) = \max(x, \alpha \cdot x)$. The α parameter is learned jointly with the weights and biases present in the model using stochastic gradient algorithm. The authors in [1] avoid pushing α to zero by not using weight decay during the training. If α is zero then the PReLU becomes the standard ReLU.
- Parametric Exponential Linear Unit (PELU) [199]: PELU is defined as $f_{\beta, \gamma}(x) = \max(\frac{\beta}{\gamma}x, \beta \cdot (\exp(\frac{\beta}{\gamma}) - 1))$. The learnable β, γ are used to adapt the positive and negative region of the ELU function. The parameters β and γ are learned alongside the model using gradient descent algorithm.
- Parametric Deformed Exponential Linear Unit (PDELU) [198]: PDELU is a slight modification of ELU with t -deformed exponential function defined as $\exp_t(x) = \exp(x)$ at $t = 1$ and $\exp_t(x) = [1 + (1 - t)x]_+^{\frac{1}{1-t}}$ at $0 < t < 2$. PDELU is defined as $f(x_i) = \max(0.0, x_i) + \min(0.0, \alpha([1 + (1 - t)x_i]^{\frac{1}{1-t}} - 1))$.

8.2.2 Multiple Learnable Functions

Activation functions in this category are modelled in terms of linear or nonlinear combinations of two or more standard activation functions. Functions that fall under this category contain a fusion of two or more activation functions using probabilistic and/or hierarchical ways.

- Elastic Exponential Linear Unit (EELU) [201]: combines the ReLU, LReLU, ELU, and MPELU activation functions. The positive slope of EELU is modified from the Gaussian distribution with a randomised standard deviation.

It is defined as $f(x_{i,j}^{(c)}) = \max(0.0, k_{i,j}^{(c)} x_{i,j}^{(c)}) + \min(0.0, \alpha^{(c)} (\exp^{\beta^{(c)} x_{i,j}^{(c)}} - 1))$. At $k = 1, \alpha = 0$, the function is ReLU and at $\alpha = \beta = k = 1$, the function becomes ELU.

- Multiple Parameter Exponential Linear Unit (MPELU) [202] : the aim of MPELU is to unify ReLU, PReLU and ELU functions by sharing their advantages thereby leading to better convergence and performance accuracy.

The definition of MPELU is $f(x_i) = \max(0.0, x_i) + \min(0.0, \alpha (\exp^{\beta_c \cdot x_i} - 1))$. The values of β_c and α controls the activation function; for example $\alpha = \beta = 1$, the resulting function is ELU, while at $\alpha = 0$ and $\alpha = 25.6302, \beta = 0.01$ the functions are ReLU and PReLU, respectively.

- Linear Mixed Activation functions [203]: it synthesises LReLU and ELU functions. It is defined as $f(x) = \beta \cdot \text{LReLU}(x) + (1 - \beta) \cdot \text{ELU}(x)$, $\beta \in [0, 1]$ learned from data.
- Soft++ [204]: this activation function synthesises PReLU and Softplus functions by parameterising the slope in the negative domain and the exponent. Soft++ is defined as $f(x) = \ln(1 + \exp^{k \cdot x}) + \frac{x}{c} - \ln(2)$, c is the slope coefficient which sets the slope in the negative domain and k is the exponent coefficient.
- Adaptive Blending Units (ABUs) [110]: a trainable linear combination of a set of basic activation functions. The following are the set of activation functions in ABU; Tanh, ELU, ReLU, the identity and Swish. ABU for the i th layer is defined as $g_i(x) = \sum_j \alpha_{ij} \cdot f_j(x)$ with $i = 1, \dots, n$ and $j = 1, \dots, m$.
- Bendable Linear Units (BLU) [111]: A parametric activation function that can synthesise the following activation functions LReLU, ReLU, identity, and Softplus. It is defined as $f(\alpha, \beta, x) = \beta(\sqrt{x^2 + \alpha^2} + \varepsilon - \alpha) + x$.
- Nonlinear Mixed Activation functions [203]: The gated activation is defined as $f(x) = \sigma(\beta x) \cdot \text{LReLU}(x) + (1 - \sigma(\beta x)) \cdot \text{ELU}(x)$.

8.2.3 Others

Authors in [2] proposed Adaptive Piecewise Linear Units (APLU) which model functions as a sum of hinge-shaped functions. Defined as $\text{APLU}(x) = \max(0, x) + \sum w_{u,k} \max(0, -x + b_{u,k})$, where $w_{u,k}, b_{u,k}$ variables are learned during the network training. APLU is similar to the popular maxout [205] functions with fewer parameters. A further improvement to the recently proposed APLU parametric activation function is called SPLASH [206]. Shifted ReLU defined in [207] is another learnable function

Table 8.1: Computation: Mathematical Operator of commonly used learnable activation functions.(LMA: Linear Mixed Activation, NGA: Nonlinear Gated Activation).

Function	Multipliers	Division	Exponent	Additions/ Subtractions	Parameters	Others
PLReLU	1	-	-	-	1	-
PELU	1	1	1	-	2	-
BLU	3	-	-	4	3	1 square root
Soft++	-	1	1	4	2	2 ln
LMA [203]	3	-	1	3	1	-
NGA [203]	3	2	3	5	1	-
SReLU [207]	1	-	-	2	4	-

that combines three linear functions using four learnable parameters. It is important to know that during training, SReLU is initialised as a LReLU before adaptively learning the parameters. At certain parameter values, SReLU can synthesise ReLU, LReLU, and PReLU. SReLU, APLU, and maxout functions are characterised by learning both convex and non-convex functions. A Variable Activation Function (VAF) sub-network proposed in [208] is another form of trainable activation function. It is a network composed of a single hidden layer, fixed activation function, and an output layer. It can be viewed as an activation function in the form of a single layer feedforward neural network with sets of learnable α and β as hidden and output layer connections. VAF is defined as $z_i = \text{VAF}(a_i) = \sum_{j=1}^k \beta_j g(\alpha_j a_i + \alpha_{0j}) + \beta_0$, where $\alpha_j, \alpha_{0j}, \beta_j$ and β_0 are the parameters to be learned during training. Another recently proposed parametric activation function is called Padé Activation Units (PAU) [209] based on the Padé approximation [210]. Computationally, PAU functions are expensive due to the presence of polynomial of order $m, n > 1$ and floating-point division.

In summary, apart from the PReLU function, all the other functions discussed are heavily parameterised and include computationally expensive mathematical functions. This is a problem for power and resource-constrained embedded and mobile devices. Table 8.1 shows the mathematical operator requirements of some of the discussed parametric activation functions.

8.3 Parametric Square-based Asymmetric Activation Functions

We revisited the square-based asymmetric activation functions defined in Chapter 4 and introduce learnable parameter. Equations 8.1, 8.3, and 8.5 are three different square-based asymmetric activation functions with negative region only parameterisation, where $0 \leq \alpha \leq 1$. The positive region exhibits an identity function like ReLU. This is the most common parametric approach in literature and is found in popular functions like PReLU, PELU, and so on. A newer, parametric approach is learning the whole

function (both the negative and positive region) and the square-based format is defined in Equations 8.2, 8.4, and 8.6, where $0 \leq \alpha \leq 1$. These activation functions have slope greater than 1 for positive inputs. Therefore they can model identity function at $\alpha = 0$. At $\alpha \neq 0$ the functions has slopes greater than 1. A slope greater than 1 for positive inputs avoids a vanishing gradient in deep networks. The best way to parameterise asymmetric activation functions is still an open question. We aim to investigate the effect of this two forms of parameterisation on performance and training convergence.

$$f(x)_{\text{psqlu}} = \begin{cases} x & : x > 0 \\ \alpha(x + \frac{x^2}{4}) & : -2.0 \leq x \leq 0 \\ -\alpha & : x < -2.0 \end{cases} \quad f(x)_{\text{psqlu}} = \alpha \begin{cases} x & : x > 0 \\ x + \frac{x^2}{4} & : -2.0 \leq x \leq 0 \\ -1 & : x < -2.0 \end{cases} \quad (8.1) \quad (8.2)$$

$$f(x)_{\text{psqreu}} = \begin{cases} x & : x > 0 \\ \alpha(x + \frac{x^2}{2}) & : -2.0 \leq x \leq 0 \end{cases} \quad f(x)_{\text{psqreu}} = \alpha \begin{cases} x & : x > 0 \\ x + \frac{x^2}{2} & : -2.0 \leq x \leq 0 \end{cases} \quad (8.3) \quad (8.4)$$

$$f(x)_{\text{psqish}} = \begin{cases} x + \frac{x^2}{32} & : x > 0 \\ \alpha(x + \frac{x^2}{2}) & : -2.0 \leq x < 0 \\ 0 & : x < -2.0 \end{cases} \quad f(x)_{\text{psqish}} = \alpha \begin{cases} x + \frac{x^2}{32} & : x > 0 \\ x + \frac{x^2}{2} & : -2.0 \leq x < 0 \\ 0 & : x < -2.0 \end{cases} \quad (8.5) \quad (8.6)$$

8.4 Multiple Square Units

We introduce a new parametric square-based asymmetric activation function called multiple square units (MSU). MSU have single parameter: α , where $\alpha \geq 0$. MSU is shown in Equation 8.7. MSU is multiple square units which consist of SQLU, Sq_Softplus, SqREU activation functions, and others. We aim to show that this parameterised approach learns, or is comparable to existing parametric functions but its hardware implementation, resource utilisation, and computationally complexity is small.

$$f(x)_{\text{MSU}} = \begin{cases} -\alpha & : x < -1 - \alpha \\ \frac{(1+x+\alpha)^2}{4} - \alpha & : -1 - \alpha \leq x \leq 1 - \alpha \\ x & : x > 1 - \alpha \end{cases} \quad (8.7)$$

Parametric square-based functions are trained simultaneously with all the network parameters during back-propagation. Using the chain rule, the gradients of f with respect to α is defined in Equation 8.8.

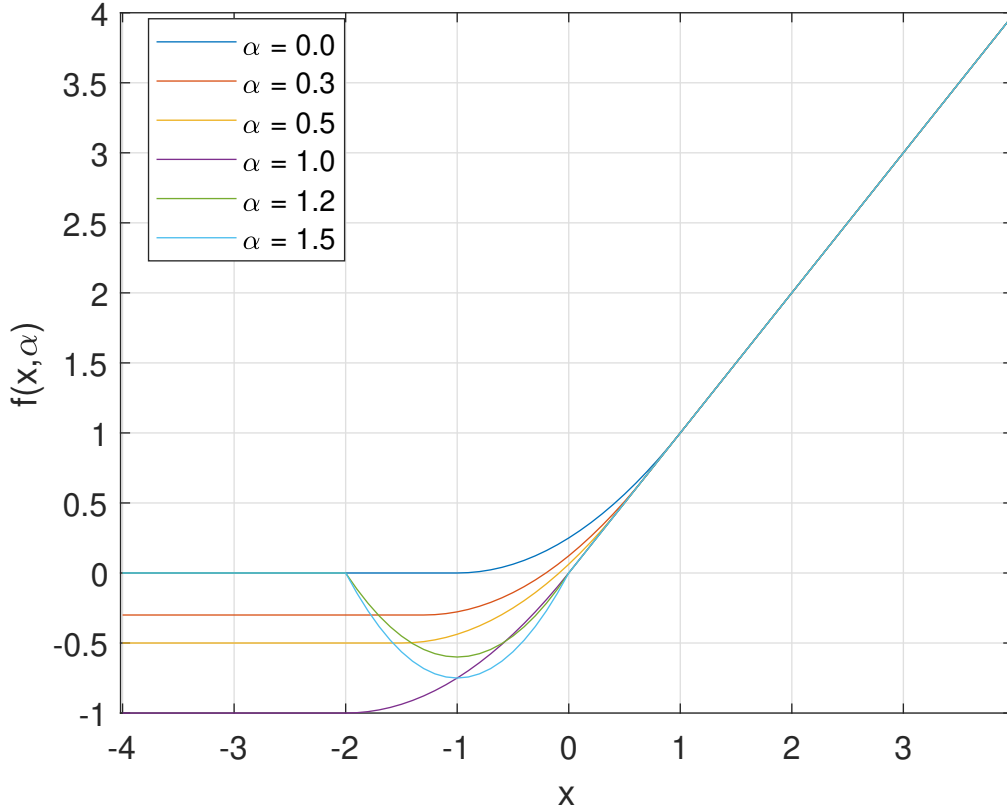


Figure 8.1: A visualisation of Multiple Square Units, varying α . The closer α is to 0, the more it is like SQ_Softplus, α greater than 1, shows SqREU and $\alpha = 1.0$ is SQLU.

$$\frac{\partial f(x)_{\text{MSU}}}{\partial \alpha} = \begin{cases} -1 & : x < -1 - \alpha \\ \frac{x + \alpha - 1}{2} & : -1 - \alpha \leq x \leq 1 - \alpha \\ 0 & : x > 1 - \alpha \end{cases} \quad (8.8)$$

The following are the desirable properties of the MSU activation function:

- It can synthesise various square-based asymmetric activation functions thereby leading to better convergence and performance accuracy.
- It is square-based, and hence computationally cheap.
- On hardware, the counter-based implementation of MSU is resource-efficient by eliminating the power and resource-hungry multipliers.

8.5 Experiments

All our experiments use the Wide Residual Network (WRN) topology [14]. The WRN topology is selected because of the presence of d and k hyperparameters which are use for controlling the depth

Table 8.2: The structure of wide residual network topology [14] used in our experiments. Hyperparameter k controls the width of the network and d controls the depth. Groups of convolutions are shown in brackets. Number is how many blocks of layers are used in succession. $N = \frac{d-4}{6}$.

Layer	Stride	Number
conv1 $16 \times 3 \times 3$	(1, 1)	1
[conv2 $16k \times 3 \times 3$]	(1, 1)	N
conv3 $32 \times 3 \times 3$	(2, 2)	1
[conv4 $32k \times 3 \times 3$]	(1, 1)	$N - 1$
conv5 $64 \times 3 \times 3$	(2, 2)	1
[conv6 $64k \times 3 \times 3$]	(1, 1)	$N - 1$
average pooling 8×8	-	1

and width of the topology respectively. Therefore, a wide variety of experiments can be performed for different network sizes with the proposed functions. Table 8.2 shows the description of a WRN- d - k topology.

We used the Keras deep learning library with the TensorFlow backend for all tests. We investigate the effect of the following parameterisation:

- Study the effect of parameterising the negative region of the activation function as expressed in Equations 8.1, 8.3 and 8.5.
- Study the effect of parameterising the whole function (i.e., bipolar parameterisation) as expressed in Equations 8.2, 8.4, and 8.6.
- Study the effect of our new parameterising, i.e., parameterise the negative and part of the positive: parameterise the slope in both the negative and positive domains as well as synthesising the properties of established square-based asymmetric functions.

In our first experiment, we compare 11 activation functions across four topologies in order to investigate the potential accuracy gains of different ways of parameterising learnable activation functions as well as how the accuracies are affected by the network depth and width. In our second experiment, we compared all the square-based learnable activation trained for a short amount of time to test how quickly networks with each activation converge.

In all our experiments, we use the CIFAR-10, CIFAR-100, and SVHN datasets on four groups of activation functions:

1. Fixed and piecewise linear units: ReLU, SQLU, SqREU, and Sqish.
2. Negative region only learnable units: PReLU, PELU, PSQLU-1, PSqREU-1, and PSqish-1.

3. Bipolar - positive and negative region learnable units: APLU, BLU, , PSQLU-2, PSqREU-2, and PSqish-2.
4. Multiple/ensemble units: MSU, we disregarded training with the other ensemble units found in literature because, they are computationally expensive.

In our first experiment, the weights are initialised with the He initialisation [1]. We employed the L2 regularisation with an initial weight decay set to 0.0005. Each model is trained using stochastic gradient descent with a mini-batch size of 128 and a momentum term of 0.9. The learning rate started from 0.1 and decayed to 0.0008 over the course of the training using an epoch-based scheduler. We set the training run to 200 epochs and recorded an average of five runs for each model. The code that produced the reported results is available at the following URL: <https://github.com/awur978/msu>.

8.5.1 CIFAR-10

CIFAR-10 is a 32×32 color image classification dataset. There are 50,000 training sets and 10,000 test sets. We preprocessed the data using global contrast normalisation and ZCA whitening. Table 8.3 shows the results of experiment 1 on CIFAR-10 dataset. As shown, across all architectures, MSU marginally outperforms all other activation functions (fixed, bipolar function, and negative-only parameterised learnable activation functions).

Overall, on an average, all parameterised activation functions perform better than the fixed activation functions. Our investigation into bipolar and negative-only region parameterisation shows that the accuracy across learnable bipolar functions and negative-only region is largely not consistent across different architecture depths and widths. For example, the WRN-16-8 shows that the negative-only region parameterisation consistently outperforms parameterising the whole activation function. Whereas, the WRN-40-1 architecture shows a majority of the bipolar parameterisation outperforming the negative-only region parameterisation. This still shows that for CIFAR-10, the architecture has a major influence on performance accuracy as compared to the choice of activation functions.

Figure 8.2 shows the training curves for four WRN architectures with fixed, bipolar, and negative-only region parameterisation of SQLU function. We observe that the parametric activation functions (PSQLU-1 and PSQLU-2) all learn more quickly than the non-parametric SQLU function for all the WRN architectures. We also observe no difference in convergence speed between bipolar and negative-only region parameterisation. The same is recorded for other parametric activation functions defined.

Table 8.3: CIFAR-10: The performance accuracy for each activation/topology pair tested in our first experiment. Results are average of five runs, and an estimated mean of the performance accuracy is recorded. The best activation for each topology is shown in **bold**. The parametric functions achieve higher accuracy, at very small margin, than their non-parametric baselines.

Parameters	Activation	WRN-40-1	WRN-40-4	WRN-16-4	WRN-16-8
Fixed	ReLU	93.43	94.67	94.67	95.24
	SQLU	92.76	94.60	94.16	94.76
	Sqish	92.59	94.81	94.59	95.31
	SqREU	93.38	94.92	94.60	95.27
Negative	PReLU	92.16	95.05	94.18	94.71
	PELU	93.50	95.36	95.04	95.14
	PSQLU-1	92.81	94.53	94.18	95.01
	PSqish-1	92.66	95.02	94.97	95.13
	PSqREU-1	93.13	94.63	94.64	95.49
Bipolar	APLU	92.18	94.27	94.43	94.54
	BLU- α	93.42	94.69	94.66	94.78
	BLU- β	93.51	94.54	94.44	94.71
	PSQLU-2	92.13	94.6	94.25	94.41
	PSqish-2	93.14	94.91	94.89	95.02
	PSqREU-2	93.07	95.06	94.33	95.26
Multiple	MSU	94.54	95.48	94.91	95.45

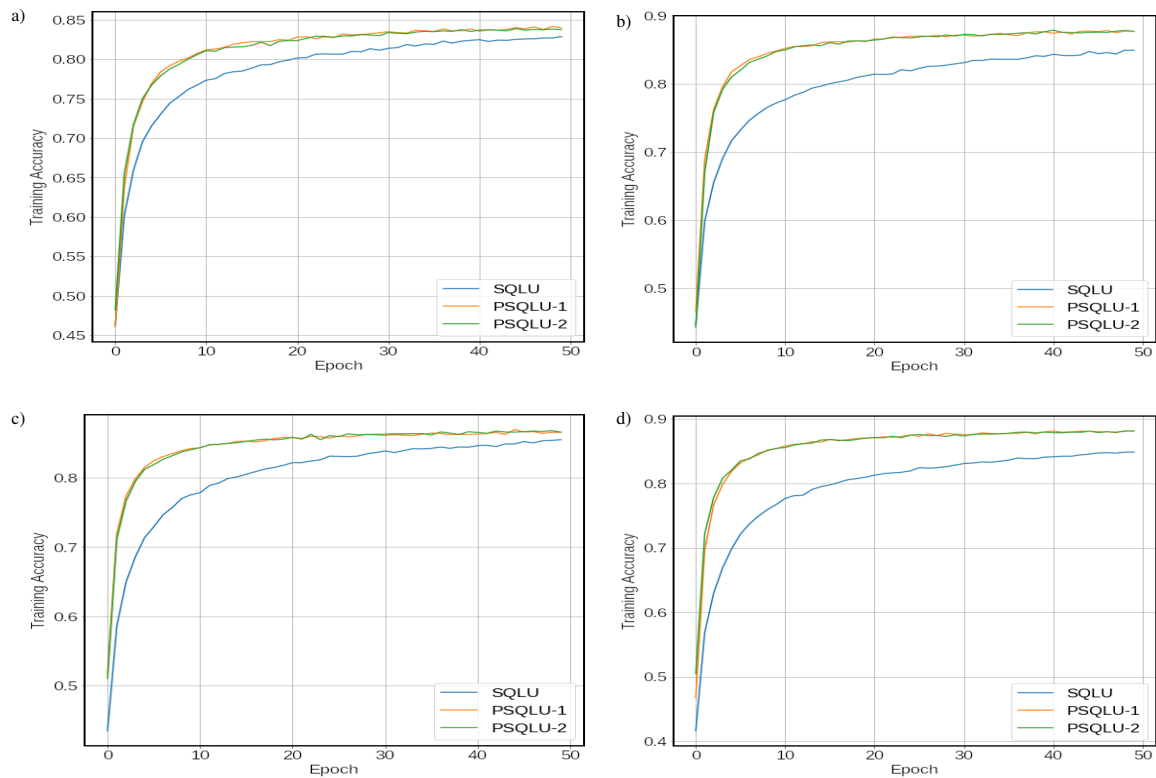


Figure 8.2: CIFAR10: Convergence curves for training sets of fixed and learnable activation functions on different WRN architectures. a) WRN-40-1 b) WRN-40-4 c) WRN-16-4 d) WRN-16-8. The parametric SQLU (PSQULU-1 and PSQULU-2) converges fastest than the fixed SQLU activation function for all architecture depths and widths.

Table 8.4: CIFAR-100: The performance accuracy for each activation/topology pair tested in our first experiment. Results are average of five runs, and an estimated mean of the performance accuracy is recorded. The best activation for each topology is shown in **bold**. The parametric functions achieve higher accuracy, at very small margin, than their non-parametric baselines.

Parameter	Activation	WRN-40-1	WRN-40-4	WRN-16-4	WRN-16-8
Fixed	ReLU	69.66	77.74	75.88	78.14
	SQLU	69.36	76.60	76.33	78.10
	Sqish	70.40	77.68	76.01	77.57
	SqREU	70.40	76.73	74.80	77.47
Negative	PSQLU-1	69.37	77.27	74.00	76.46
	PSqish- 1	68.65	76.72	74.53	77.17
	PSqREU-1	68.93	76.84	74.47	77.51
Bipolar	PSQLU-2	69.42	75.89	75.72	77.37
	PSqish-2	70.27	76.73	75.82	77.88
	PSqREU-2	70.70	76.98	75.25	77.80
Multiple	MSU	69.70	77.94	76.97	78.45

8.5.2 CIFAR-100

CIFAR-100 dataset is a 32×32 colour image that has 100 classes. CIFAR-100 has 50,000 training images and 10,000 test images. We follow the same preprocessing as the CIFAR-10 and employ data augmentation for data preprocessing.

Table 8.4 shows the results of experiment 1 on CIFAR-100 dataset. We find that in all but one case (WRN-16-4), the parametric activation functions consistently outperform the fixed functions. MSU function outperforms all the other activation functions except in WRN-40-1 architecture. This shows the superiority of MSU in terms of performance accuracy across datasets and architectures. As observed with the CIFAR-10 dataset, parameterising the whole activation function yields a marginally better performance accuracy than the negative-only parameterised activation for CIFAR-100 dataset. Figure 8.3 shows the convergence curve for experiment 2.

8.5.3 SVHN

The SVHN dataset [121] is a coloured 32×32 real-world digit recognition dataset consisting of photos of house numbers in Google Street View images. There are 73257 images for training and 26032 images for testing. We follow the same procedure used for the CIFAR-10 experiments. Table 8.5 shows the results of experiment one on SVHN dataset.

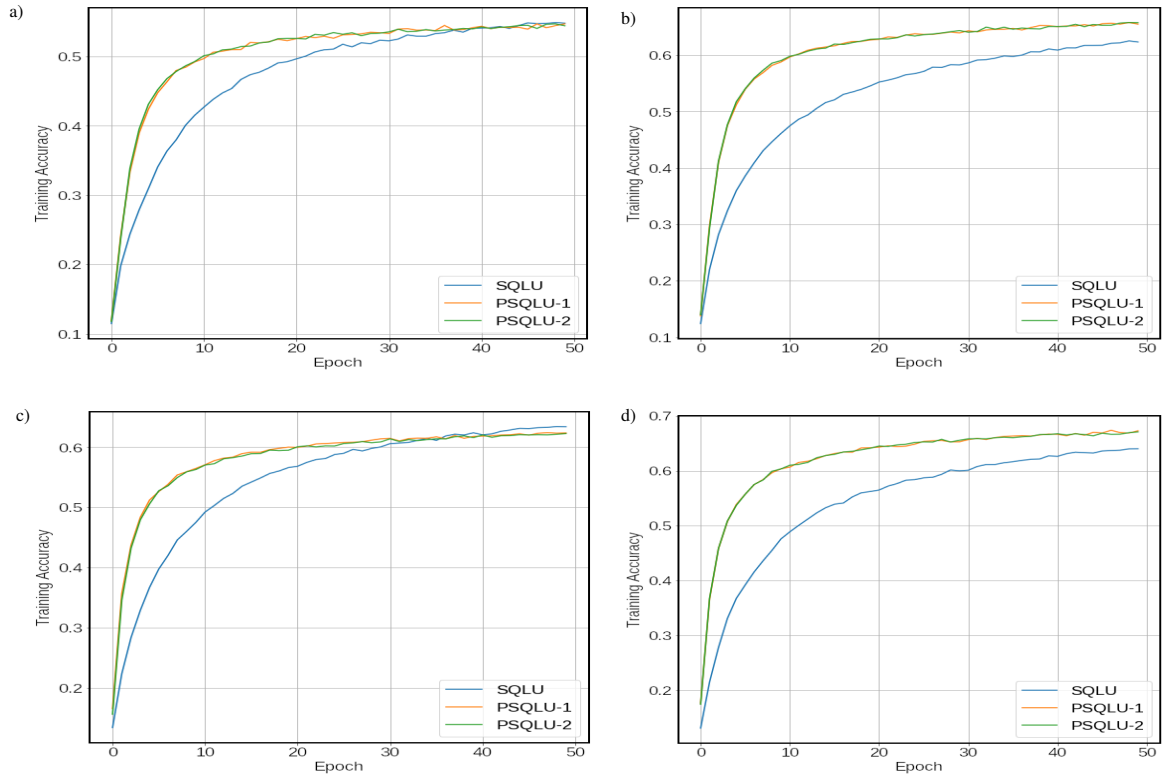


Figure 8.3: CIFAR100: Convergence curves for training sets of fixed and learnable activation functions on different WRN architectures. a) WRN-40-1 b) WRN-40-4 c) WRN-16-4 d) WRN-16-8. The parametric SQLU (PSQLU-1 and PSQLU-2) converges fastest than the fixed SQLU activation function for all architecture depths and widths.

Table 8.5: SVHN: The performance accuracy for each activation/topology pair tested in our first experiment. Results are average of five runs, and an estimated mean of the performance accuracy is recorded. The best activation for each topology is shown in **bold**. The parametric functions achieve higher accuracy, at very small margin, than their non-parametric baselines.

Parameters	Activation	WRN-40-1	WRN-40-4	WRN-16-4	WRN-16-8
Fixed	ReLU	95.29	95.88	95.50	95.59
	SQLU	95.52	95.46	95.20	95.45
	Sqish	95.51	95.78	95.44	95.92
	SqREU	95.52	95.95	95.68	95.95
Negative	PSQLU-1	95.39	95.76	95.27	95.44
	PSqish- 1	95.28	95.58	95.68	95.58
	PSqREU-1	95.55	95.95	95.42	95.90
Bipolar	PSQLU-2	95.29	95.47	95.52	95.30
	PSqish-2	95.34	95.83	95.67	95.79
	PSqREU-2	95.26	95.68	95.68	95.68
Multiple	MSU	95.23	95.99	95.50	95.41

The results of our experiment on SVHN dataset is particularly interesting because all the results show negligible differences. The lowest recorded accuracy is 95.23% and the highest is 95.95%. Parameterising of any type doesn't seem to have a significant effect over the performance accuracy across all the architectures for SVHN dataset. Figure 8.4 shows the convergence curve for experiment 2. There is no significant convergence speed-up between fixed and parametric SQLU activation functions for all the different WRN architecture.

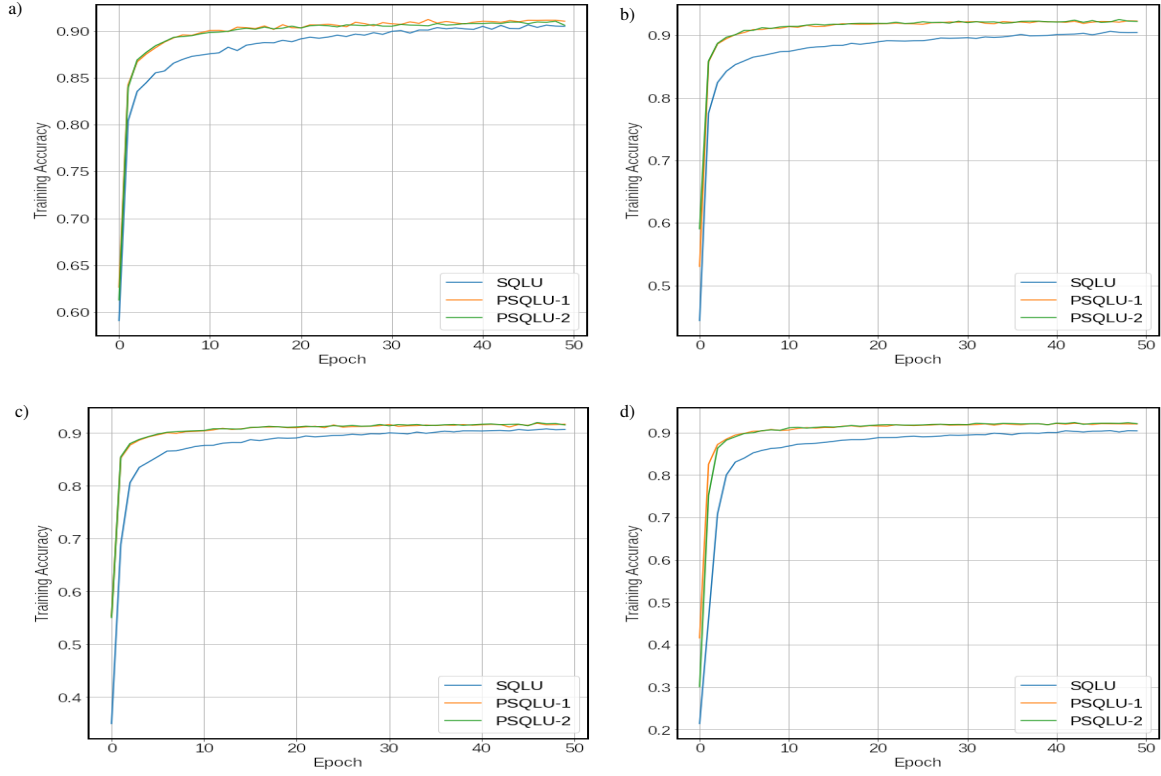


Figure 8.4: SVHN: Convergence curves for training sets of fixed and learnable activation functions on different WRN architectures. a) WRN-40-1 b) WRN-40-4 c) WRN-16-4 d) WRN-16-8. The parametric SQLU (PSQULU-1 and PSQULU-2) converges fastest than the fixed SQLU activation function for all architecture depths and widths.

8.6 Conclusion

In this chapter, we investigated the effect of parameterising different regions of fixed activation functions. Furthermore, we presented a novel activation function, multiple square unit that continuously generalises among SQLU, SQ_softplus, SqREU, and other activation functions. Compared to other learned and fixed activation functions, including ReLU and its variants, MSU functions show superior performance across various datasets and network architectures with different widths and depths. MSU can be easily implemented on hardware using our proposed asymmetric generator. We show that parametric activation functions speed up learning and reduce training time. The convergence speedup characteristics of parametric activation functions makes them desirable in time, power, and resource-limited embedded and mobile devices.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

Neural network is changing every aspect of our lives. State-of-the-art neural networks are computationally intensive therefore making deploying neural network solutions on embedded systems and hardware challenging. Unlike software solutions, the hardware solution is limited in resources and power. To address this problem, we presented software and hardware methods for improving the performance accuracy and efficiency of various neural network architectures. This thesis focuses on improving the efficiency of neural network architectures by eliminating resource-intensive and computationally complex machine learning building blocks.

On software platforms, we propose several computationally efficient activation functions aim at obtaining state-of-the-art performance accuracy at simple computation. On various Intel CPUs, a speedup ranging from 1.3x to 4.3x is achieved. On an ARM M3 processor, we achieve a 4.37x to 169.79x range of speedup. We have demonstrated extensively that our proposed square-based activation functions achieve comparable or better performance accuracy on several datasets and neural network architectures.

We propose two novel nonlinear symmetric activation functions with simple mathematical equations. The proposed functions circumvent the use of the exponential term and floating-point division resulting in a reduction in computational time. Their derivatives can be highly optimised and require a single cycle operation. Networks using the SQNL function train faster to reach predefined performance goals.

Additionally, we focus on computationally efficient activation functions for deep learning architectures. Particularly, we introduce five new asymmetric activation functions for deep learning. We show a speedup of 1.9x to 4.3x for the various functions using metrics provided by the Intel processor and discuss the advantage of square-based nonlinearity in CPU-based inference engine and low-end hardware

devices with limited hardware capabilities. The square-based radial basis kernel proposed shows, on average, a speedup in training time of about 8% for SQ-RBF based networks without affecting the overall generalisability of the network. SQ-RBF uses about 10% fewer neurons than Gaussian RBF, hence making it very attractive. Furthermore, the SQ-RBF was modified for support vector machines. The speedup was also recorded with the proposed quadratic-based kernel transformation on support vector machines.

On hardware platforms, we offer solutions that can construct nonlinear activation functions using significantly fewer resources. We also propose the elimination of resource and power-hungry multipliers through gated activation and quantised scaling unit for recurrent and feedforward neural networks. Our SQNL based solutions offer the potential of making the artificial neural network inference engines compact, and hence attractive for embedded solutions. Our solutions use standard digital building blocks and can be implemented on ASICs or FPGAs.

We have introduced a novel method for generating computational and resource-efficient symmetric and asymmetric activation functions. The experiments were performed on an Altera Cyclone V FPGA. Although the counter based implementation uses lower resources when compared to other methods, the proposed functions have also been implemented using hardware multipliers. We show that combinatorial logic wrapped around a standard multiplier offers an attractive alternative to the counter implementation. Comparisons with LUT implementation, a counter-based symmetric function with an 8-bit resolution, offers an estimated throughput (per gate) speedup of 1.79x to 3.75x. Similarly, a throughput (per gate) speedup of 2.42x to 5.06x is estimated with multipliers. Higher throughputs per gate have also been estimated with 12-bit implementations. The single-cycle hardware implementation of the proposed method is also resource-efficient for higher resolution and provides area savings when compared to LUT. The proposed method can produce many other mappings with relatively small modifications. It is attractive in applications that require a dynamic mapping change.

We also extended the square law algorithm for element-wise multiplication in RNN models. The algorithm has a unique advantage in which, without using a multiplier, it also simultaneously scales the output by an external input. In an FPGA implementation of an LSTM, the non-requirement of a multiplier implies that the proposed solution outperforms the simplest function - Hard Tanh. We propose a compact multiplier, the QSU. The QSU capitalises on the reduced word width from the gate generators. In a conventional LSTM, our proposed solutions can replace all five activations functions and the three element-wise multipliers.

9.2 Future Work

This work has improved the inference engine of a variety of neural network architectures through developing and implementing the important building blocks.

The simplicity and linearity of the derivatives of our proposed activation functions can be explored further for creating on-chip training using gradient-based algorithm. The implications of the simple derivative of the square-law based activation functions on online training can be further explored in terms of convergence speed, performance accuracy, and resource utilisation.

Although the square law is closely aligned with the current generation of ANN formulations, the square-law based nonlinear generator algorithm can also be applied to Spiking Neural Networks. In a future development, the speedup options to software based solutions can be explored.

Furthermore, the N sequence discussed in Chapter 3 can be further investigated for lower values by studying the effects on training and performance accuracy. Even though the morphology of square-based functions proposed in this work is similar to benchmark functions, they still differ in saturation levels, linear regions, and others. Hence, weight initialisation techniques specifically tailored for square-law functions can be investigated for faster convergence and better performance accuracy.

Fourier neural networks, which were first introduced in late 90's [211, 212], are resurfacing again [212–215]. Fourier neural networks transform a signal from the time or space domain to the frequency domain in a process similar to the Fourier transform by using sinusoidal activation function [213]. The square law can be further investigated to form a sinusoidal function without the trigonometric complexity associated. We have already established SQINE and further investigation is to be performed on hardware using Fourier neural networks.

Diverse activation function is another area in which different activation functions are used per neuron, per layer. Also new architectures such as FCN-LSTM require both symmetric and asymmetric activation functions. Further investigation and analysis on the proposed SQ-GEN on machine learning models can be explored.

Recently, focus has been shifted from fixed activation functions to learnable activation functions. As discussed in Chapter 8, learnable activation functions have a variety of advantages over fixed functions. However, the algorithm for updating the parameters should be further explored. At the moment, gradient descent is the algorithm of choice; investigating other types of algorithms may further improve convergence speed and performance accuracy.

Bibliography

- [1] K. He, X. Zhang, S. Ren, and J. Sun, “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification,” in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.
- [2] F. Agostinelli, M. Hoffman, P. Sadowski, and P. Baldi, “Learning activation functions to improve deep neural networks,” *arXiv preprint arXiv:1412.6830*, 2014.
- [3] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, “Fast and accurate deep network learning by exponential linear units (elus),” in *Proceedings of the 6th International Conference on Learning Representation (ICLR 2016)*, 2016.
- [4] D. Hendrycks and K. Gimpel, “Gaussian error linear units (gelus),” *arXiv preprint arXiv:1606.08415*, 2016.
- [5] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, “Self-normalising neural networks,” in *Advances in Neural Information Processing Systems*, 2017, pp. 971–980.
- [6] S. Elfving, E. Uchibe, and K. Doya, “Sigmoid-weighted linear units for neural network function approximation in reinforcement learning,” *Neural Networks*, vol. 107, pp. 3–11, 2018.
- [7] M. Basirat and P. M. Roth, “The quest for the golden activation function,” *arXiv preprint arXiv:1808.00783*, 2018.
- [8] Y. Zhou, D. Li, S. Huo, and S.-Y. Kung, “Soft-root-sign activation function,” *arXiv preprint arXiv:2003.00547*, 2020.
- [9] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
- [10] H. Kaiming, X. Zhang, S. Ren, and J. Sun, “Identity mappings in deep residual networks,” in *European conference on computer vision*. Springer, 2016, pp. 630–645.

- [11] R. Dimitrios. (2020) SVHN classification with CNN. <https://www.kaggle.com/dimitriosroussis/svhn-classification-with-cnn-keras-96-acc>, (accessed 3 March 2020).
- [12] Z. Abai and N. Rajmalwar, “Densenet models for tiny imagenet classification,” *arXiv preprint arXiv:1904.10429*, 2019.
- [13] S. Giri. (2019) Resnet model on tiny imagenet. https://github.com/sonugiri1043/Train_ResNet_On_Tiny_ImageNet, (accessed 28 February 2020).
- [14] S. Zagoruyko and N. Komodakis, “Wide residual networks,” in *British Machine Vision Conference*, 2016.
- [15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, 2012, pp. 1097–1105.
- [16] W. Zaremba, I. Sutskever, and O. Vinyals, “Recurrent neural network regularisation,” *arXiv preprint arXiv:1409.2329*, 2014.
- [17] S. Mozaffari, O. Y. Al-Jarrah, M. Dianati, P. Jennings, and A. Mouzakitis, “Deep learning-based vehicle behavior prediction for autonomous driving applications: A review,” *IEEE Transactions on Intelligent Transportation Systems*, 2020.
- [18] D. De Silva, S. Sierla, D. Alahakoon, E. Osipov, X. Yu, and V. Vyatkin, “Toward intelligent industrial informatics: A review of current developments and future directions of artificial intelligence in industrial applications,” *IEEE Industrial Electronics Magazine*, vol. 14, no. 2, pp. 57–72, 2020.
- [19] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, “Project adam: Building an efficient and scalable deep learning training system,” in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 571–582.
- [20] F. Li, B. Zhang, and B. Liu, “Ternary weight networks,” *arXiv preprint arXiv:1605.04711*, 2016.
- [21] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang *et al.*, “Ese: Efficient speech recognition engine with sparse lstm on FPGA,” in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 75–84.
- [22] S. Vassiliadis, M. Zhang, and J. G. Delgado-Frias, “Elementary function generators for neural-network emulators,” *IEEE transactions on neural networks*, vol. 11, no. 6, pp. 1438–1449, 2000.

- [23] C. Gao, D. Neil, E. Ceolini, S.-C. Liu, and T. Delbruck, "Deltarnn: A power-efficient recurrent neural network accelerator," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 21–30.
- [24] M. Courbariaux, Y. Bengio, and J. David, "Training deep neural networks with low precision multiplications," *International Conference on Learning Representation*, 2014.
- [25] B. DasGupta and G. Schnitger, "The power of approximating: a comparison of activation functions," pp. 615–622, 1993.
- [26] P. Pushpa and K. Manimala, "Implementation of hyperbolic tangent activation function in vlsi," *International Journal of Advanced Research in Computer Science & Technology*, vol. 2, pp. 225–228, 2014.
- [27] V. Tiwari and N. Khare, "Hardware implementation of neural network with sigmoidal activation functions using cordic," *Microprocessors and Microsystems*, vol. 39, no. 6, pp. 373–381, 2015.
- [28] H. Amin, K. M. Curtis, and B. R. Hayes-Gill, "Piecewise linear approximation applied to non-linear function of a neural network," *IEE Proceedings-Circuits, Devices and Systems*, vol. 144, no. 6, pp. 313–317, 1997.
- [29] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. ICML*, vol. 30, no. 1, 2013, p. 3.
- [30] M. Goyal, R. Goyal, P. V. Reddy, and B. Lall, "Activation functions," in *Deep Learning: Algorithms and Applications*. Springer, 2020, pp. 1–30.
- [31] J. P. David, K. Kalach, and N. Tittley, "Hardware complexity of modular multiplication and exponentiation," *IEEE Transactions on Computers*, vol. 56, no. 10, pp. 1308–1319, 2007.
- [32] M. A. Hanif, R. Hafiz, M. U. Javed, S. Rehman, and M. Shafique, "Energy-efficient design of advanced machine learning hardware," in *Machine Learning in VLSI Computer-Aided Design*. Springer, 2019, pp. 647–678.
- [33] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Advances in neural information processing systems*, 1990, pp. 598–605.
- [34] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantisation and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

- [35] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” *arXiv preprint arXiv:1704.04861*, 2017.
- [36] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, “Squeezenet: Alexnet-level accuracy with 50x fewer parameters and 0.5 mb model size,” *arXiv preprint arXiv:1602.07360*, 2016.
- [37] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, “Binarised neural networks,” in *Advances in neural information processing systems*, 2016, pp. 4107–4115.
- [38] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, “Xnor-net: Imagenet classification using binary convolutional neural networks,” in *European conference on computer vision*. Springer, 2016, pp. 525–542.
- [39] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, “Quantised neural networks: Training neural networks with low precision weights and activations,” *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.
- [40] M. Horowitz, “1.1 computing’s energy problem (and what we can do about it),” in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 2014, pp. 10–14.
- [41] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, “Deep learning with limited numerical precision,” in *International Conference on Machine Learning*, 2015, pp. 1737–1746.
- [42] D. Lin, S. Talathi, and S. Annapureddy, “Fixed point quantisation of deep convolutional networks,” in *International conference on machine learning*, 2016, pp. 2849–2858.
- [43] K. Hwang and W. Sung, “Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1,” in *2014 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE, 2014, pp. 1–6.
- [44] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, “Neural networks with few multiplications,” *arXiv preprint arXiv:1510.03009*, 2015.
- [45] P. Bacchus, R. Stewart, and E. Komendantskaya, “Accuracy, training time and hardware efficiency trade-offs for quantised neural networks on FPGAs,” in *International Symposium on Applied Reconfigurable Computing*. Springer, 2020, pp. 121–135.

- [46] M. I. Jordan *et al.*, “Why the logistic function? a tutorial discussion on probabilities and neural networks,” 1995.
- [47] B. DasGupta and G. Schnitger, “The power of approximating: a comparison of activation functions,” in *Advances in neural information processing systems*, 1993, pp. 615–622.
- [48] H. N. Mhaskar and C. A. Micchelli, “How to choose an activation function,” in *Advances in Neural Information Processing Systems*, 1994, pp. 319–326.
- [49] K. Hornik, M. Stinchcombe, and H. White, “Multilayer feedforward networks are universal approximators,” *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.
- [50] J. Han and C. Moraga, “The influence of the sigmoid function parameters on the speed of back-propagation learning,” in *International Workshop on Artificial Neural Networks*. Springer, 1995, pp. 195–201.
- [51] P. Chandra and Y. Singh, “Feedforward sigmoidal networks-equicontinuity and fault-tolerance properties,” *IEEE Transactions on Neural Networks*, vol. 15, no. 6, pp. 1350–1366, 2004.
- [52] P. Chandra, “Sigmoidal function classes for feedforward artificial neural networks,” *Neural Processing Letters*, vol. 18, no. 3, pp. 205–215, 2003.
- [53] P. Chandra, U. Ghose, and A. Sood, “A non-sigmoidal activation function for feedforward artificial neural networks,” in *2015 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2015, pp. 1–8.
- [54] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, “Efficient backprop,” in *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48.
- [55] S. Kong and M. Takatsuka, “Hexpo: A vanishing-proof activation function,” in *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2017, pp. 2562–2567.
- [56] V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines,” in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.
- [57] B. Xu, R. Huang, and M. Li, “Revise saturated activation functions,” *arXiv preprint arXiv:1602.05980*, 2016.

- [58] B. DasGupta, H. T. Siegelmann, and E. Sontag, "On the complexity of training neural networks with continuous activation functions," *IEEE Transactions on Neural Networks*, vol. 6, no. 6, pp. 1490–1504, 1995.
- [59] B. Karlik and A. V. Olgac, "Performance analysis of various activation functions in generalised mlp architectures of neural networks," *International Journal of Artificial Intelligence and Expert Systems*, vol. 1, no. 4, pp. 111–122, 2011.
- [60] J. M. Sopena, E. Romero, and R. Alquezar, "Neural networks with periodic and monotonic activation functions: a comparative study in classification problems," 1999.
- [61] D. L. Elliott, "A better activation function for artificial neural networks," Tech. Rep., 1993.
- [62] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural networks*, vol. 4, no. 2, pp. 251–257, 1991.
- [63] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken, "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function," *Neural networks*, vol. 6, no. 6, pp. 861–867, 1993.
- [64] B. Carlile, G. Delamarter, P. Kinney, A. Marti, and B. Whitney, "Improving deep learning by inverse square root linear units (isrlus)," *arXiv preprint arXiv:1710.09967*, 2017.
- [65] MATLAB, 9.7.0.1190202 (R2019b). Natick, Massachusetts: The MathWorks Inc., 2018.
- [66] Intel, "Vector mathematics (vm) performance and accuracy data," 2018. [Online]. Available: <https://intel.ly/2GPbsNt>
- [67] MIT, "https://github.com/petteriaimonen/libfixmath," 2009. [Online]. Available: <https://github.com/PetteriAimonen/libfixmath>
- [68] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *ATT Labs [Online]*. Available: <http://yann.lecun.com/exdb/mnist>, vol. 2, 2010.
- [69] A. Asuncion and D. Newman, "Uci machine learning repository," 2007.
- [70] D. Dheeru and E. Karra Taniskidou, "UCI machine learning repository," 2017. [Online]. Available: <http://archive.ics.uci.edu/ml>

- [71] C. Gulcehre, M. Moczulski, M. Denil, and Y. Bengio, “Noisy activation functions,” in *International Conference on Machine Learning*, 2016, pp. 3059–3068.
- [72] S. Sen and A. Raghunathan, “Approximate computing for long short term memory (lstm) neural networks,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2266–2276, 2018.
- [73] J. Brownlee. (2019) Sequence classification with LSTM recurrent neural networks in python with keras. <https://machinelearningmastery.com/sequence-classification-lstm-recurrent-neural-networks-python-keras/>, (accessed 3 March 2019).
- [74] G. Roopal. (2017) LSTM tensorflow. https://github.com/roopalgarg/lstm-tensorflow/blob/master/sentiment_lstm_mini_batch_gd.ipynb, (accessed 3 June 2019).
- [75] K. Harrison. (2016) Rnn w/ LSTM cell example in tensorflow and python. <https://pythonprogramming.net/rnn-tensorflow-python-machine-learning-tutorial/>, (accessed 3 June 2019).
- [76] D. Aymeric. (2017) Tensorflow examples. https://github.com/aymericdamien/TensorFlow-Examples/blob/master/examples/3_NeuralNetworks/recurrent_network.py, (accessed 3 June 2019).
- [77] A. Dinu, M. N. Cirstea, and S. E. Cirstea, “Direct neural-network hardware-implementation algorithm,” *IEEE Transactions on Industrial Electronics*, vol. 57, no. 5, pp. 1845–1848, 2010.
- [78] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, “Optimising fpga-based accelerator design for deep convolutional neural networks,” in *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, 2015, pp. 161–170.
- [79] C. Wang, L. Gong, X. Li, and X. Zhou, “A ubiquitous machine learning accelerator with automatic parallelisation on FPGA,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 10, pp. 2346–2359, 2020.
- [80] F. Z. Wang, L. O. Chua, X. Yang, N. Helian, R. Tetzlaff, T. Schmidt, C. Li, J. M. G. Carrasco, W. Chen, and D. Chu, “Adaptive neuromorphic architecture (ana),” *Neural Networks*, vol. 45, pp. 111–116, 2013.

- [81] A. Sripad, G. Sanchez, M. Zapata, V. Pirrone, T. Dorta, S. Cambria, A. Marti, K. Krishnamourthy, and J. Madrenas, “Snava—a real-time multi-FPGA multi-model spiking neural network simulation architecture,” *Neural Networks*, vol. 97, pp. 28–45, 2018.
- [82] J. Shawash and D. R. Selviah, “Real-time nonlinear parameter estimation using the levenberg–marquardt algorithm on field programmable gate arrays,” *IEEE Transactions on Industrial Electronics*, 2013.
- [83] S. M. Bohte, J. N. Kok, and H. L. Poutre, “Error-backpropagation in temporally encoded networks of spiking neurons,” *Neurocomputing*, vol. 48, no. 1, pp. 17–37, 2002.
- [84] L. M. Reyneri, “A performance analysis of pulse stream neural and fuzzy computing systems,” *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 42, no. 10, pp. 642–660, 1995.
- [85] W. Gerstner and W. M. Kistler, *Spiking neuron models: Single neurons, populations, plasticity*. Cambridge university press, 2002.
- [86] Y. C. Yoon, “LIF and simplified srm neurons encode signals into spikes via a form of asynchronous pulse sigma–delta modulation,” *IEEE transactions on neural networks and learning systems*, vol. 28, no. 5, pp. 1192–1205, 2017.
- [87] A. W. Savich, M. Moussa, and S. Areibi, “The impact of arithmetic representation on implementing mlp-bp on FPGAs: A study,” *IEEE Transactions on Neural Networks*, vol. 18, no. 1, pp. 240–252, 2007.
- [88] V. Tiwari and N. Khare, “Hardware implementation of neural network with sigmoidal activation functions using cordic,” *Microprocessors and Microsystems*, vol. 39, 2015.
- [89] Z. Hajduk, “High accuracy FPGA activation function implementation for neural networks,” *Neurocomputing*, vol. 247, pp. 59–61, 7/19 2017.
- [90] R. W. Duren, R. J. M. II, P. D. Reynolds, and M. L. Trumbo, “Real-time neural network inversion on the src-6e reconfigurable computer,” *IEEE Transactions on Neural Networks*, vol. 18, no. 3, pp. 889–901, 2007.
- [91] C. Torres-Huitzil, B. Girau, and C. Castellanos-Sánchez, “On-chip visual perception of motion: A bio-inspired connectionist model on FPGA,” *Neural Networks*, vol. 18, no. 5, pp. 557–565, 2005.

- [92] D. Baptista and F. Morgado-Dias, “Low-resource hardware implementation of the hyperbolic tangent for artificial neural networks,” *Neural Computing and Applications*, vol. 23, no. 3-4, pp. 601–607, 2013.
- [93] V. Canals, A. Morro, A. Oliver, M. L. Alomar, and J. L. Rosselló, “A new stochastic computing methodology for efficient neural network implementation,” *IEEE transactions on neural networks and learning systems*, vol. 27, no. 3, pp. 551–564, 2016.
- [94] A. Gomperts, A. Ukil, and F. Zurfluh, “Development and implementation of parameterised FPGA-based general purpose neural networks for online applications,” *IEEE Transactions on Industrial Informatics*, vol. 7, no. 1, pp. 78–89, 2011.
- [95] A. Tisan and J. Chin, “An end-user platform for FPGA-based design and rapid prototyping of feedforward artificial neural networks with on-chip backpropagation learning,” *IEEE Transactions on Industrial Informatics*, vol. 12, no. 3, pp. 1124–1133, 2016.
- [96] H. Hikawa, “A digital hardware pulse-mode neuron with piecewise linear activation function,” *IEEE Transactions on Neural Networks*, vol. 14, no. 5, pp. 1028–1037, 2003.
- [97] N. J. Cotton and B. M. Wilamowski, “Compensation of nonlinearities using neural networks implemented on inexpensive microcontrollers,” *IEEE Transactions on Industrial Electronics*, vol. 58, no. 3, pp. 733–740, 2011.
- [98] A. Armato, L. Fanucci, E. P. Scilingo, and D. D. Rossi, “Low-error digital hardware implementation of artificial neuron activation functions and their derivative,” *Microprocessors and Microsystems*, vol. 35, no. 6, pp. 557–567, 8 2011.
- [99] I. Sahin and I. Koyuncu, “Design and implementation of neural networks neurons with radbas, logsig, and tansig activation functions on FPGA,” *Elektronika ir Elektrotechnika*, vol. 120, no. 4, pp. 51–54, 2012.
- [100] X. Chen, G. Wang, W. Zhou, S. Chang, and S. Sun, “Efficient sigmoid function for neural networks based FPGA design,” in *International Conference on Intelligent Computing*. Springer, 2006, pp. 672–677.
- [101] V. Tiwari, S. Vyas, and N. Khare, “Hardware efficient implementation of neural network.”
- [102] Intel-Altera. Gate counting methodology for APEX 20K devices. [Online]. Available: <https://bit.ly/2IwPFbJ>

- [103] Z.-C. Fan and W.-J. Hwang, "Efficient VLSI architecture for training radial basis function networks," *Sensors*, vol. 13, no. 3, pp. 3848–3877, 2013.
- [104] Z. Hajduk, "High accuracy FPGA activation function implementation for neural networks," *Neurocomputing*, vol. 247, pp. 59–61, 2017.
- [105] V. Saichand, S. Arumugam, N. Mohankumar *et al.*, "FPGA realisation of activation function for artificial neural networks," in *2008 Eighth International Conference on Intelligent Systems Design and Applications*, vol. 3. IEEE, 2008, pp. 159–164.
- [106] I. Tsmots, O. Skorokhoda, and V. Rabyk, "Hardware implementation of sigmoid activation functions using fpga," in *2019 IEEE 15th International Conference on the Experience of Designing and Application of CAD Systems (CADSM)*. IEEE, 2019, pp. 34–38.
- [107] I. University of California, "Machine learning repository.[online] available: <https://bit.ly/2iabpok>, accessed may 9 2017."
- [108] L. Hao, S. De, Z. Xu, C. Studer, H. Samet, and T. Goldstein, "Training quantised nets: A deeper understanding," in *Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS 2017)*, 2017.
- [109] Y. Li, P. L. K. Ding, and B. Li, "Training neural networks by using power linear units (polus)."
- [110] L. R. Sütfield, F. Brieger, H. Finger, S. Füllhase, and G. Pipa, "Adaptive blending units: Trainable activation functions for deep neural networks," in *Science and Information Conference*. Springer, 2020, pp. 37–50.
- [111] L. B. Godfrey, "An evaluation of parametric activation functions for deep learning," in *2019 IEEE International Conference on Systems, Man and Cybernetics (SMC)*. IEEE, 2019, pp. 3006–3011.
- [112] F. Chollet *et al.*, "Keras," <https://github.com/fchollet/keras>, 2015.
- [113] Nvidiacom, "Nvidia cuDNN," 2014. [Online]. Available: <https://developer.nvidia.com/cudnn>
- [114] N. N. Schraudolph, "A fast, compact approximation of the exponential function," *Neural Computation*, vol. 11, no. 4, pp. 853–862, 1999.
- [115] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *2017*

- ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 1–12.
- [116] P. Ramachandran, B. Zoph, and Q. V. Le, “Searching for activation functions,” *ICLR Workshop Track*, year=2017.
 - [117] S.-I. Amari, “Natural gradient works efficiently in learning,” *Neural Computation*, 1998.
 - [118] S. Ioffe and C. Szegedy, “Batch normalisation: Accelerating deep network training by reducing internal covariate shift,” in *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, 2015.
 - [119] Y. Ying, J. Su, P. Shan, L. Miao, X. Wang, and S. Peng, “Rectified exponential units for convolutional neural networks,” *IEEE Access*, vol. 7, pp. 101 633–101 640, 2019.
 - [120] A. Krizhevsky, G. Hinton *et al.*, “Learning multiple layers of features from tiny images,” 2009.
 - [121] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, “Reading digits in natural images with unsupervised feature learning,” 2011.
 - [122] Anonymous. (2016) Tiny imagenet visual recognition challenge. <https://tiny-imagenet.herokuapp.com/>, (accessed 25 May 2020).
 - [123] Keras. (2019) Keras examples. https://github.com/keras-team/keras/blob/master/examples/cifar10_cnn.py, (accessed 3 April 2019).
 - [124] G. Desjardins, K. Simonyan, R. Pascanu *et al.*, “Natural neural networks,” in *Advances in Neural Information Processing Systems*, 2015, pp. 2071–2079.
 - [125] J. Martens, “Deep learning via hessian-free optimisation.” in *ICML*, vol. 27, 2010, pp. 735–742.
 - [126] M. Lin, Q. Chen, and S. Yan, “Network in network,” *arXiv preprint arXiv:1312.4400*, 2013.
 - [127] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” 2014.
 - [128] L. Wei. (2018) Convolutional neural networks for CIFAR-10. <https://github.com/BIGBALLON/cifar-10-cnn>, (accessed 25 May 2019).
 - [129] A. Krizhevsky, V. Nair, and G. Hinton, “Cifar-10 (canadian institute for advanced research).” [Online]. Available: <http://www.cs.toronto.edu/~kriz/cifar.html>

- [130] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, “Densely connected convolutional networks,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.
- [131] S. Xi. (2017) Restricted boltzmann machine. <https://github.com/XiSHEN0220/Restricted-Boltzmann-Machine>, (accessed 25 May 2019).
- [132] E. Wang, J. J. Davis, R. Zhao, H.-C. Ng, X. Niu, W. Luk, P. Y. Cheung, and A. George, “Deep neural network approximation for custom hardware: Where we’ve been, where we’re going,” *ACM Computing Surveys (CSUR)*, vol. 52, no. 2, pp. 1–39, 2019.
- [133] K. K. Tan, T. H. Lee, and S. Huang, *Precision motion control: design and implementation*. Springer Science & Business Media, 2007.
- [134] D. Larkin, A. Kinane, V. Muresan, and N. O’Connor, “An efficient hardware architecture for a neural network activation function generator,” in *International Symposium on Neural Networks*. Springer, 2006, pp. 1319–1327.
- [135] T. Yang, Y. Wei, Z. Tu, H. Zeng, M. A. Kinsy, N. Zheng, and P. Ren, “Design space exploration of neural network activation function circuits,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 10, pp. 1974–1978, 2018.
- [136] K. Basterretxea, J. M. Tarela, I. del Campo, and G. Bosque, “An experimental study on nonlinear function computation for neural/fuzzy hardware design,” *IEEE transactions on neural networks*, vol. 18, no. 1, pp. 266–283, 2007.
- [137] M. Tommiska, “Efficient digital implementation of the sigmoid function for reprogrammable logic,” *IEE Proceedings-Computers and Digital Techniques*, vol. 150, no. 6, pp. 403–411, 2003.
- [138] F. Karim, S. Majumdar, and H. Darabi, “Insights into LSTM fully convolutional networks for time series classification,” *IEEE Access*, vol. 7, pp. 67 718–67 725, 2019.
- [139] J. Reese and S. Zaranek, “GPU programming in MATLAB,” *MathWorks News&Notes. Natick, MA: The MathWorks Inc*, pp. 22–5, 2012.
- [140] N. Ploskas and N. Samaras, *GPU programming in MATLAB*. Morgan Kaufmann, 2016.
- [141] A. Wuraola and N. Patel, “SQLN: A new computationally efficient activation function,” in *2018 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2018, pp. 1–7.

- [142] W. Duch and N. Jankowski, "Transfer functions: hidden possibilities for better neural networks." in *ESANN*. Citeseer, 2001, pp. 81–94.
- [143] G. A. Hoffmann, "Adaptive transfer functions in radial basis function (RBF) networks," in *International Conference on Computational Science*. Springer, 2004, pp. 682–686.
- [144] D. Lowe, "Multi-variable functional interpolation and adaptive networks," *Complex Systems*, vol. 2, pp. 321–355.
- [145] Y. Arora, A. Singhal, and A. Bansal, "A study of applications of RBF network," *International Journal of Computer Applications*, vol. 94, no. 2, 2014.
- [146] M. Ring and B. M. Eskofier, "An approximation of the gaussian RBF kernel for efficient classification with SVMs," *Pattern Recognition Letters*, vol. 84, pp. 107–113, 2016.
- [147] B. Xu, F. Shen, J. Zhao, and T. Zhang, "A self-adaptive growing method for training compact RBF networks," in *International Conference on Neural Information Processing*. Springer, 2017, pp. 74–81.
- [148] X. Meng, P. Rozycki, J.-F. Qiao, and B. M. Wilamowski, "Nonlinear system modeling using RBF networks for industrial application," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 3, pp. 931–940, 2017.
- [149] Y. Liu, J. Zhao, and Y. Xiao, "C-rbfnn: A user retweet behavior prediction method for hotspot topics based on improved RBF neural network," *Neurocomputing*, vol. 275, pp. 733–746, 2018.
- [150] W. Duch and N. Jankowski, "Survey of neural transfer functions," *Neural computing surveys*, vol. 2, no. 1, pp. 163–212, 1999.
- [151] P. J. Moreno, P. P. Ho, and N. Vasconcelos, "A kullback-leibler divergence based kernel for SVM classification in multimedia applications," in *Advances in neural information processing systems*, 2004, pp. 1385–1392.
- [152] M. Drewnik and Z. Pasternak-Winiarski, "Svm kernel configuration and optimisation for the handwritten digit recognition," in *IFIP International Conference on Computer Information Systems and Industrial Management*. Springer, 2017, pp. 87–98.
- [153] S. University, "Nonlinear SVMs," 2008. [Online]. Available: <https://stanford.io/2CMEoSf>

- [154] C.-W. Hsu, C.-C. Chang, C.-J. Lin *et al.*, “A practical guide to support vector classification,” 2003.
- [155] L. Ljung, “System identification,” *Wiley Encyclopedia of Electrical and Electronics Engineering*, 2001.
- [156] M. Berberich and K. Doll, “Highly flexible FPGA-architecture of a support vector machine,” 2014.
- [157] M. Qasaimeh, A. Sagahyroon, and T. Shanableh, “FPGA-based parallel hardware architecture for real-time image classification,” *IEEE Transactions on Computational Imaging*, vol. 1, no. 1, pp. 56–70, 2015.
- [158] J. Kim and S. Jung, “Implementation of the RBF neural chip with the back-propagation algorithm for on-line learning,” *Applied Soft Computing*, vol. 29, pp. 233–244, 2015.
- [159] P. Chen, H. Tsai, C. Lin, and C. Lee, “FPGA realisation of a radial basis function based nonlinear channel equalizer,” in *International Symposium on Neural Networks*. Springer, 2005, pp. 320–325.
- [160] H.-H. Chou, Y.-S. Kung, N. V. Quynh, and S. Cheng, “Optimised FPGA design, verification and implementation of a neuro-fuzzy controller for PMSM drives,” *Mathematics and Computers in Simulation*, vol. 90, pp. 28–44, 2013.
- [161] A. C. De Souza and M. A. Fernandes, “Parallel fixed point implementation of a radial basis function network in an FPGA,” *Sensors*, vol. 14, no. 10, pp. 18 223–18 243, 2014.
- [162] Y. Ago, K. Nakano, and Y. Ito, “A classification processor for a support vector machine with embedded dsp slices and block RAMs in the FPGA,” in *2013 IEEE 7th International Symposium on Embedded Multicore Socs*. IEEE, 2013, pp. 91–96.
- [163] G.-B. Huang, P. Saratchandran, and N. Sundararajan, “An efficient sequential learning algorithm for growing and pruning RBF (GAP-RBF) networks,” *IEEE transactions on systems, man, and cybernetics, part B (Cybernetics)*, vol. 34, no. 6, pp. 2284–2292, 2004.
- [164] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
- [165] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using rnn encoder–decoder for statistical machine translation,”

- in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1724–1734.
- [166] Y. Guan, Z. Yuan, G. Sun, and J. Cong, “FPGA-based accelerator for long short-term memory recurrent neural networks,” in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2017, pp. 629–634.
 - [167] J. C. Ferreira and J. Fonseca, “An FPGA implementation of a long short-term memory neural network,” in *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2016, pp. 1–8.
 - [168] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. Zhang, “Efficient and effective sparse lstm on FPGA with bank-balanced sparsity,” in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 63–72.
 - [169] V. Rybalkin, A. Pappalardo, M. M. Ghaffar, G. Gambardella, N. Wehn, and M. Blott, “Finn-1: Library extensions and design trade-off analysis for variable precision lstm networks on FPGAs,” in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 89–897.
 - [170] E. Azari and S. Vrudhula, “An energy-efficient reconfigurable lstm accelerator for natural language processing,” in *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 2019, pp. 4450–4459.
 - [171] H. Sim and J. Lee, “A new stochastic computing multiplier with application to deep convolutional neural networks,” in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.
 - [172] S. Wang, Z. Li, C. Ding, B. Yuan, Q. Qiu, Y. Wang, and Y. Liang, “C-lstm: Enabling efficient LSTM using structured compression techniques on FPGAs,” in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 11–20.
 - [173] E. Nurvitadhi, J. Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr, “Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC,” in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2016, pp. 1–4.

- [174] I. Kouretas and V. Paliouras, “Hardware aspects of long short term memory,” in *25th IEEE International Conference on Electronics, Circuits and Systems*, 2018, pp. 525–528.
- [175] A. X. M. Chang, B. Martini, and E. Culurciello, “Recurrent neural networks hardware implementation on FPGA,” *arXiv preprint arXiv:1511.05552*, 2015.
- [176] M. Lee, K. Hwang, J. Park, S. Choi, S. Shin, and W. Sung, “FPGA-based low-power speech recognition with recurrent neural networks,” in *IEEE International Workshop on Signal Processing Systems*. IEEE, 2016, pp. 230–235.
- [177] S. Li, “Towards efficient hardware acceleration of deep neural networks on FPGA,” Ph.D. dissertation, University of Pittsburgh, 2018.
- [178] H. Li, S. De, Z. Xu, C. Studer, H. Samet, and T. Goldstein, “Training quantised nets: A deeper understanding,” in *Advances in Neural Information Processing Systems*, 2017, pp. 5811–5821.
- [179] Y. Guo, “A survey on methods and theories of quantised neural networks,” *arXiv preprint arXiv:1808.04752*, 2018.
- [180] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, “Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients,” *CoRR*, vol. abs/1606.06160, 2016. [Online]. Available: <http://arxiv.org/abs/1606.06160>
- [181] M. Courbariaux, Y. Bengio, and J.-P. David, “Binaryconnect: Training deep neural networks with binary weights during propagations,” in *Advances in neural information processing systems*, 2015, pp. 3123–3131.
- [182] A. B. V. S. D. Karkada, V. M. S. C. K. Datta, and V. Saletore, “Efficient 8-bit quantisation of transformer neural machine language translation model.”
- [183] TensorFlow, “Quantisation-aware training,” 2017. [Online]. Available: <https://github.com/tensorflow/tensorflow/blob/v1.13.1/tensorflow/contrib/quantize/README.md>
- [184] J. Su, N. J. Fraser, G. Gambardella, M. Blott, G. Durelli, D. B. Thomas, P. H. Leong, and P. Y. Cheung, “Accuracy to throughput trade-offs for reduced precision neural networks on reconfigurable logic,” in *International Symposium on Applied Reconfigurable Computing*. Springer, 2018, pp. 29–42.

- [185] Xilinx, “Logiccore ip multiplier v11.2,” 2011. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/mult_gen_ds255.pdf
- [186] Kaggle. (2016) US baby names: Explore naming trends from babies born in the US. <https://www.kaggle.com/kaggle/us-baby-names/version/1#NationalNames.csv>, (accessed 3 April 2020).
- [187] N. Virdee. (2018) Lstm neural network from scratch. <https://www.kaggle.com/navjindervirdee/lstm-neural-network-from-scratch>, (accessed 3 April 2020).
- [188] M. Marcus, B. Santorini, and M. A. Marcinkiewicz, “Building a large annotated corpus of english: The penn treebank,” 1993.
- [189] S. Merity, C. Xiong, J. Bradbury, and R. Socher, “Pointer sentinel mixture models,” *International Conference on Learning Representation*, 2017.
- [190] H. Xiao, K. Rasul, and R. Vollgraf, “Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms,” *arXiv preprint arXiv:1708.07747*, 2017.
- [191] H. Fayek, “Matdl: A lightweight deep learning library in matlab,” *Journal of Open Source Software*, vol. 2, no. 19, p. 413, 2017.
- [192] S. Deepak. (2017) GRU implementation. https://github.com/deepakrana47/GRU_implementation, (accessed 25 May 2020).
- [193] R. J. Lyon, B. Stappers, S. Cooper, J. Brooke, and J. Knowles, “Fifty years of pulsar candidate selection: from simple filters to a new principled real-time classification approach,” *Monthly Notices of the Royal Astronomical Society*, vol. 459, no. 1, pp. 1104–1123, 2016.
- [194] D. Dua and C. Graff, “UCI machine learning repository,” 2017. [Online]. Available: <http://archive.ics.uci.edu/ml>
- [195] Anonymous. (2016) Epsilon in the LIBSVM website. <http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#epsilon>, (accessed 25 March 2020).
- [196] Intel. (2018) Cyclone V device overview. <https://intel.ly/2UbUwVY>, (accessed 3 April 2020).
- [197] S. Scardapane, S. Van Vaerenbergh, S. Totaro, and A. Uncini, “Kafnets: Kernel-based non-parametric activation functions for neural networks,” *Neural Networks*, vol. 110, pp. 19–32, 2019.

- [198] Q. Cheng, H. Li, Q. Wu, L. Ma, and N. N. King, “Parametric deformable exponential linear units for deep neural networks,” *Neural Networks*, 2020.
- [199] L. Trottier, P. Gigu, B. Chaib-draa *et al.*, “Parametric exponential linear unit for deep convolutional neural networks,” in *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2017, pp. 207–214.
- [200] S. Balaji, T. Kavya, and N. Sebastian, “Learn-able parameter guided activation functions,” in *Proceedings of SAI Intelligent Systems Conference*. Springer, 2020, pp. 583–597.
- [201] D. Kim, J. Kim, and J. Kim, “Elastic exponential linear units for convolutional neural networks,” *Neurocomputing*, 2020.
- [202] Y. Li, C. Fan, Y. Li, Q. Wu, and Y. Ming, “Improving deep neural network with multiple parametric exponential linear units,” *Neurocomputing*, vol. 301, pp. 11–24, 2018.
- [203] S. Qian, H. Liu, C. Liu, S. Wu, and H. San Wong, “Adaptive activation functions in convolutional neural networks,” *Neurocomputing*, vol. 272, pp. 204–212, 2018.
- [204] A. Ciuparu, A. Nagy-Dăbâcan, and R. C. Mureșan, “Soft++, a multi-parametric non-saturating non-linearity that improves convergence in deep neural architectures,” *Neurocomputing*, vol. 384, pp. 376–388, 2020.
- [205] I. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, “Maxout networks,” in *International conference on machine learning*. PMLR, 2013, pp. 1319–1327.
- [206] M. Tavakoli, F. Agostinelli, and P. Baldi, “Splash: Learnable activation functions for improving accuracy and adversarial robustness,” *arXiv preprint arXiv:2006.08947*, 2020.
- [207] X. Jin, C. Xu, J. Feng, Y. Wei, J. Xiong, and S. Yan, “Deep learning with s-shaped rectified linear activation units,” in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, 2016, pp. 1737–1743.
- [208] A. Apicella, F. Isgrò, and R. Prevete, “A simple and efficient architecture for trainable activation functions,” *Neurocomputing*, vol. 370, pp. 1–15, 2019.
- [209] A. Molina, P. Schramowski, and K. Kersting, “Padé activation units: End-to-end learning of flexible activation functions in deep networks,” in *International Conference on Learning Representations*, 2019.

- [210] C. Brezinski and J. Van Iseghem, “Padé approximations,” *Handbook of Numerical Analysis*, vol. 3, pp. 47–222, 1994.
- [211] J. M. Sopena, E. Romero, and R. Alquezar, “Neural networks with periodic and monotonic activation functions: a comparative study in classification problems,” 1999.
- [212] P. Liu, Z. Zeng, and J. Wang, “Multistability of recurrent neural networks with nonmonotonic activation functions and unbounded time-varying delays,” *IEEE transactions on neural networks and learning systems*, vol. 29, no. 7, pp. 3000–3010, 2017.
- [213] A. Zhumeckenov, M. Uteuliyeva, O. Kabdolov, R. Takhanov, Z. Assylbekov, and A. J. Castro, “Fourier neural networks: A comparative study,” *arXiv preprint arXiv:1902.03011*, 2019.
- [214] V. Sitzmann, J. N. Martel, A. W. Bergman, D. B. Lindell, and G. Wetzstein, “Implicit neural representations with periodic activation functions,” *arXiv preprint arXiv:2006.09661*, 2020.
- [215] G. Parascandolo, H. Huttunen, and T. Virtanen, “Taming the waves: sine as activation function in deep neural networks,” 2016.