OVERCOMING NOISE AND VARIATIONS IN LOW-PRECISION NEURAL NETWORKS

A Dissertation
Presented to
The Academic Faculty

By

Devon D. Janke

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

August 2021

Copyright © Devon D. Janke 2021

OVERCOMING NOISE AND VARIATIONS IN LOW-PRECISION NEURAL NETWORKS

Approved by:

Dr. David Anderson, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Arijit Raychowdhury
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Aaron Lanterman
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Shaolan Li
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Date Approved: May 6, 2021

Any intelligent fool can make things bigger and more complex. It takes a touch of genius —
and a lot of courage — to move in the opposite direction.

E. F. Schumacher

I dedicate this dissertation to my wife, Natalee. I would never have made it to where I am nor become who I am now without your constant, patient support. Thank you for being my greatest catalyst for growth.

ACKNOWLEDGEMENTS

I cannot begin to give thanks for those who supported my success without first acknowledging the earnestness to learn and succeed that my parents planted in me at an early age. They gave so much time and energy to my education and helped me to get on this path.

Nance Ericson gave me my first internship at Oak Ridge National Labs, where he showed me a variety of exciting projects that were available in the world of research. It was because of the hours he spent personally mentoring me and being my friend that I was able to really appreciate the importance of what I was learning and how crucial it was for me to thoroughly understand and internalize each principle.

I had my first independent research project under Shiuh-hua Chiang at Brigham Young University, Provo. The trust he placed in me to work and succeed taught me that I am capable and creative enough to contribute to the state of the art. He pushed me to apply for more prestigious graduate schools than I ever would have considered. I never would have submitted my application to Georgia Tech without his guidance.

A large part of my discovery for my current passions came through trial and error, and the time spent doing research with Dr. Ayazi and Dr. Rincon-Mora were a big part of that. They encouraged me to broaden my horizons of my interests and taught me how to begin a thorough research project and develop an intuition for technical material.

Andy Smith and the guys in "group" were crucial to my personal development. I will never forget the hundreds of hours they gave to helping me discover and redefine the way I view myself. I do not believe I would have successfully completed my doctorate without following the encouragement from my wife to seek counsel from Andy to begin this path of change.

I must also acknowledge my committee and the time they gave up from their own research to check my work, ask questions, and provide feedback. A special thanks to Arijit Raychowdhury for meeting with me personally as I begin my work on this topic.

I am very grateful to Tom Darbonne for trusting me with this fascinating research topic and for the opportunities you gave me for professional growth. Tom gave a wealth of time to me in one-on-one mentoring as I navigated the realm of entrepreneurialism as a complete novice.

Of course, I cannot neglect to give my gratitude to David Anderson. David was very supportive of me exploring this field despite needing to learn from the very basics of machine learning and provided all the support I needed while pushing me to be creative and self-sufficient in my research. His academic, professional, and personal guidance are invaluable and helped drive me to finish successfully.

TABLE OF CONTENTS

Acknowledgments v

List of Tables xi

List of Figures xii

Introduction xvii

Chapter 1: Intelligent Sensor Processing 1

1.1 Sensors on the Edge 2

1.2 Machine Learning and Neural Networks 3

1.2.1 Neural networks 3

1.2.2 Common layer types 5

1.2.3 Activation functions 7

1.3 General Overview of Machine Learning Algorithms 8

1.3.1 Data Preprocessing 8

1.3.2 Forward Propagation 9

1.3.3 Loss Function 9

1.3.4 Backpropagation 10

1.3.5 Parameter Update 10

1.4 Mini-batch Training: Redundant Training Sets 11

Chapter 2: Implementing Neural Networks 15

2.1 Digital Implementation 15

2.1.1 Binary Arithmetic 16

2.2 Powering Down 19

2.3 Analog Hardware 22

2.3.1 Arithmetic blocks 22

2.3.2 Limitations 25

2.4 Introducing Analog Into Machine Learning and Neural Networks 27

2.4.1 Full Replacement 28

2.4.2 Computation Acceleration 29

2.4.3 Deployment 30

2.5 Training-to-Deployment Translation 31

2.6 Modeling PVT in Analog Neural Networks 33

Chapter 3: Generalizing and Fitting 36

3.1 Improving Generalization 38

3.1.1 Dropout Training 38

3.1.2 Data Augmentation 39

3.1.3 Early Stopping 40

3.1.4 Weight Regularization 40

3.2 Device Fitting 41

3.2.1 Understanding the Problem 42

3.2.2 Breaking from Traditional Techniques 42

3.3 Population Training 43

Chapter 4: Building robustness into the network 47

4.1 Selecting a better activation function 48

4.2 Effects of the Network Shape 51

4.2.1 Effect of depth 51

4.2.2 Effect of layer sizes 53

4.2.3 Conclusion: best parameters for variation resilience 55

4.3 Sparse network connections 57

Chapter 5: Generating the Best Sparse Networks 63

5.1 Genetic Programming 63

5.2 Synaptic Pruning 65

5.2.1 One-time pruning 66

5.2.2 Gradual Pruning 69

5.2.3 Repeated full pruning 72

5.3 Again, But With More Parameters 75

Chapter 6: Future Work 80

6.1 Better Modeling 80

6.2 Improving the Sparse Network Search 80

6.3 Low-power Binary Multiplication: Bit Shift 81

6.3.1 Quantization Error 83

6.3.2 Training for One-Hot Quantization . . . 87
6.3.3 N-Hot Quantization . . . 88

Chapter 7: Conclusion . . . 90

Appendix A: Datasets . . . 94
A.1 Custom Voice Activity and Noise Dataset . . . 94
A.2 Microsoft Deep Noise Suppression (MDNS) Challenge . . . 95
A.3 The Wisconsin Breast Cancer Dataset . . . 95
A.4 Electrical Grid Stability Dataset . . . 95
A.5 MNIST . . . 95

Appendix B: Voice Activity Detection — Feature Extraction . . . 96

Appendix C: Modeling Feature Noise . . . 99

References . . . 111

LIST OF TABLES

4.1 Neural Network Layers and Sizes Used In Comparing Activation Functions 51

4.2 Activation Functions Compared and Their Associated Equations 51

5.1 Neural Network Layers and Sizes Used In Sparsity Analysis 66

LIST OF FIGURES

1.1 Illustration of neuron/synapse connectivity in human brain. 4

1.2 Block diagrams for a single artificial neuron and a MLP 4

1.3 Depictions of convolutional and recurrent layers of neural networks 6

1.4 Three common activation functions used in nonlinear neural networks. . . . 8

1.5 Consequences of using a learning rate that is too small (left) or too large (right) 11

1.6 Color map plots showing how many of the samples used for training closely resemble one another, both in terms of correlation and euclidean distance. . 12

1.7 Plot showing how repetition in data does not justify randomly selecting a subset to use for training 13

1.8 Plot showing the effectiveness of mini-batch optimization in machine learning 14

2.1 Diagram of the basic Von Neumann architecture 16

2.2 Transistor diagrams for (N)AND and XOR binary logic gates. 16

2.3 Gate- and block-level diagrams for the binary half and full adders. 17

2.4 Example of binary addition and ripple adder architecture 17

2.5 Example of binary multiplication and ripple-adder-based multiplier architecture 19

2.6 Plots showing the relationship between tranisistor count and binary number bit length. 20

2.7 Plot illustrating the relative performance per power for GPUs and TPUs in relation to CPUs. 22

2.8 Block and transistor diagrams for components in analog multipliers 23

2.9 Kirchoff's Current Law and an OpAmp voltage adder 24

2.10 Three simple architectures that can act as (a) an artificial neuron and (b) a complete fully-connected layer in a neural network. 25

2.11 Examples of nonlinear phenomena in analog multipliers 26

2.12 CMOS floating gate transistor 27

2.13 Illustration of how the noise vectors transform one network into many variations 35

3.1 Examples of edge-case images that may be misclassified. 37

3.2 Illustrations of the concept of fitting a function to the data. 37

3.3 An example of how connections may be disabled for dropout training . . . 39

3.4 Figures demonstrating how a single image can become six different examples in a training set by applying simple alterations. 39

3.5 Visualization of the early-stop point in relation to the validation and training loss as a function of number of epochs trained. 40

3.6 Effects of (a) L2 normalization and (b) dropout training on device overfitting. 43

3.7 Performance of population training with gradients averaged. The colors represent different numbers of noised networks that are trained in parallel, with darker lines using more devices in parallel. 45

3.8 Performance of population training with gradients summed. 46

3.9 Performance of population training with summed gradients and learning rate proportional to $\sqrt{p}$. 46

4.1 Activation functions and some of their derivatives 49

4.2 Difference in performance variation when using similar activation functions but with different slopes 50

4.3 Comparison of activation functions including ReLU with a variety of slopes 52

4.4 Plots demonstrating the relationship of various network layer sizes affects the median and IQR of the network accuracies when subjected to variations. The networks represented here have four hidden layers. 54

4.5 Plots demonstrating the relationship of various network layer sizes affects the median and IQR of the network accuracies when subjected to variations. The networks represented here have three hidden layers. 56

4.6 Plots demonstrating the relationship of various network layer sizes affects the median and IQR of the network accuracies when subjected to variations. The networks represented here have two hidden layers. 56

4.7 Demonstration of how introducing sparsity to a neural network serves to reduce the effects of variations in the neural network parameters when trained and tested under the same conditions as those in Fig. 4.2b. 58

4.8 Plots demonstrating the relationship of various network layer sizes affects the median and IQR of the network accuracies when subjected to variations. The networks represented here have four hidden layers, and the Electrical Grid Stability dataset is used. 59

4.9 Plots demonstrating the relationship of various network layer sizes affects the median and IQR of the network accuracies when subjected to variations. The networks represented here have three hidden layers, and the Electrical Grid Stability dataset is used. 60

4.10 Plots demonstrating the relationship of various network layer sizes affects the median and IQR of the network accuracies when subjected to variations. The networks represented here have two hidden layers, and the Electrical Grid Stability dataset is used. 60

4.11 Plots demonstrating the relationship of various network layer sizes affects the median and IQR of the network accuracies when subjected to variations. The networks represented here have four hidden layers, and the Wisconsin Breast Cancer dataset is used. 61

4.12 Plots demonstrating the relationship of various network layer sizes affects the median and IQR of the network accuracies when subjected to variations. The networks represented here have three hidden layers, and the Wisconsin Breast Cancer dataset is used. 62

4.13 Plots demonstrating the relationship of various network layer sizes affects the median and IQR of the network accuracies when subjected to variations. The networks represented here have two hidden layers, and the Wisconsin Breast Cancer dataset is used. 62

5.1 Illustration of mutation and crossover in genetic programming 64

5.2 Plots showing how differing methods of weight initialization for training sparse neural networks may affect the training of the final neural network . 67

5.3 Results of using the “prune once” technique to generate sparse neural networks 68

5.4 Results of using gradual pruning to introduce sparsity into a neural network 70

5.5 Results of using repeated full pruning to introduce sparsity into a neural network 74

5.6 Results for repeated full pruning methods when starting out with larger hidden layers with 12, 20, and 30 neurons per layer 77

5.7 Results for gradual pruning methods when starting out with larger hidden layers with 12, 20, and 30 neurons per layer 78

5.8 Results for repeated full pruning methods when starting out with larger hidden layers with 12, 20, and 30 neurons per layer 79

6.1 Pruning methods using larger neural networks with a subset of the MNIST dataset. 82

6.2 Example of multiplying a number by $4 = 2^2$, which is the same as a shifting all bits to the left (or “decimal point” to the right) two positions. 83

6.3 Plot showing how the full-precision values between -32 and 32 are quantized down to one-hot form (closest power of two). Histogram of the resulting errors when comparing the original value to the quantized value . . 86

6.4 Histogram of quantization error when using equation (6.7) 86

6.5 Plots showing the histograms of quantization errors for two- and three-hot quantization 89

B.1 Example schematics for possible implementations of the signal and noise level estimators used in feature detection. 97

B.2 Example signals generated by the circuits in Fig. B.1. 98

C.1 Plots demonstrating the gaussian nature of the features extracted when subjected to gaussian noise added to various parameters. The vertical black line is the ideal value, and t is the number of samples from the start of the audio (at 16kSps). The time shown is randomly selected from the given sample and feature. 102

C.2 For a given extracted feature and across all bands, the standard deviation and mean of the feature value is calculated at all time points, and the standard deviations are all normalized to the mean. The vertical black line is the average standard deviation for that feature and band. 103

SUMMARY

Machine learning and artificial intelligence have become commonplace in all aspects of everyday life. They drive the decision-making process for major corporations by deriving insights from data that may have otherwise gone unnoticed. At they same time, they provides simple and accurate information about the world to help average consumers optimize their own lives. From doorbells to refrigerators and smartphones to security systems, machine learning has had an irreplaceable impact on the way people carry out their day-to-day activities, and its reach is only becoming more and more ubiquitous.

Traditional machine learning algorithms and neural networks are implemented using powerful digital computational architectures such as GPUs, TPUs, and FPGAs, demonstrating high performance and successfully completing previously impossible tasks. Unfortunately, the power required to train and generate predictions with the neural networks is too high to be implemented in energy-constrained systems such as implants and edge devices. Many of these systems would significantly benefit from on-board neural networks that could respond to stimuli in real time. The important question that this work seeks to address is how to bring the game-changing power of neural networks closer to the edge of the internet of things without significant degradation of performance or battery life.

CHAPTER 1
INTELLIGENT SENSOR PROCESSING

In the age of the "Internet of Things" (IOT), data is transmitted to and from almost anything imaginable including phones, televisions, cameras, ovens, and even doorbells. These connected devices have integrated sensors that collect information about their surroundings: audio, visuals, temperature, and anything else that can be quantified. After collecting the data, so-called smart devices adjust their performance and carry out tasks based on user interaction and preferences. The key to transforming simple transducers into smart sensors lies in machine learning and neural networks.

Machine learning has transformed the way people approach data processing. In a machine learning algorithm, the computer reads through vast amounts of data in a fraction of the amount of time required by humans. Using this data, it ajusts the parameters of a neural network by performing repetitive guess, check, and update cycles via billions of multiplications, additions, and read and write operations on devices made up of billions of transistors. In this way, the machine "learns" the trends and patterns within the data and generates an approximation of the functions that govern the task, such as classification or prediction. This learned information is part of what is referred to as artificial intelligence (AI).

A properly designed and trained neural network can perform prediction and classification tasks as well as or better than the average person. Many of these neural networks can be designed, trained, and deployed on a consumer-grade computer. In a sense, machine learning has made data processing more accessible by leveraging the mass amounts of computational power and data now available.

In order to attain such impressive performance, some neural network architectures have grown from a few layers of fully-connected neurons to networks such as Microsoft's T-

NLG, natural language processing (NLP) network with 17 billion trainable parameters [1]. Training such large networks to achieve state-of-the-art accuracy can take days, weeks, or months of constant computing before arriving at the final trained state.

As these models continue to grow, the energy needed for training and inferencing with them grows as well. The process of training can demand between 100 W and 250 W on a single GPU, with each training step costing at least 200 mJ per image [2] The energy required to make predictions varies depending on the architecture, quoted at about 60 mJ and 1180 mJ per image for state-of-the-art architectures AlexNet and VGG-16 respectively on the Nvidia GeForce GTX Titan X GPU [3]. This not only goes contrary to the energy-reduction mantra in an economy focused on a greener future; it inhibits classifiers from being implemented in power-starved IoT edge devices.

1.1 Sensors on the Edge

It can likely be assumed that with adequate time, energy, data, and hardware, any problem can be learned via machine learning and approximated with a neural network. However, many applications that would benefit from added intelligence cannot afford all of these resources due to space or energy limitations. A simple approach to overcoming computational power limits such as in edge devices in the IoT is to send sensor data elsewhere for processing before performing a desired task, but for many cases this may not be feasible or desirable.

Frequent wireless transmission of data from edge devices requires significant amounts of energy, decreasing operational lifetime [4, 5]. Even in cases where energy limitations are not an issue, transmitting data elsewhere for computation adds latency to the data-processing pipeline and introduces security risks like interception of or tampering with data. Local computing is one of the major appeals of edge AI, so turning to distributed computing methods is not a desirable way forward.

One specific application that we explored was livestock behavioral monitoring for early detection of illnesses. In this case, the end goal is to attach a device to an animal, such as a

cow or a pig, that will sense movement, sounds, and other information from the animal and its environment. This information could then be used to predict whether or not the animal is sick or if livestock managers need to pay close attention to it. Cattle and swine tend to move around a lot; it is not likely that they will stay in one place to continuously transmit data to wireless receiver points, and they definitely won't want to be connected to a power supply. In cases like these, on-board computing is the only option for providing accurate and timely information for livestock caretakers to make informed decisions.

These and other edge applications present a new set of challenges that are not common to most high-performance state-of-the-art neural networks.

1.2 Machine Learning and Neural Networks

Machine learning is the process of training some mathematical model, such as a neural network or an equation, to approximate some function that governs a task such as prediction or classification. To better understand how the machine learning process can be altered for improved performance, a thorough understanding of neural networks and the process of training is required. This section gives an overview of what neural networks are and how they work and an intuitive explanation for how training can bring a randomly-generated NN to its final state. While the structure of the neural network and the training process may seem complicated, the basic functions needed to make it work are very simple.

1.2.1 Neural networks

Neural networks are meant to imitate the functionality of the brain. The brain is made up of neurons which are interconnected via synapses (see Fig. 1.1); each neuron can output electric pulses to other neurons, and these pulses allow the neurons to communicate and tell the body what to do. As the brain learns, these synaptic connections grow weaker or stronger based on which synaptic connections are most important for a given task. The basic building block for artificial NNs is called the "neuron" because it behaves and learns in a

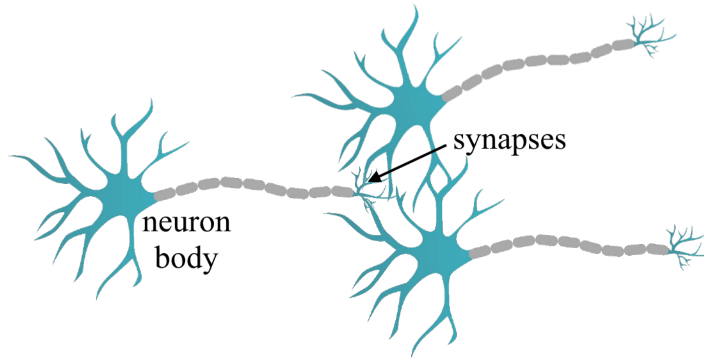


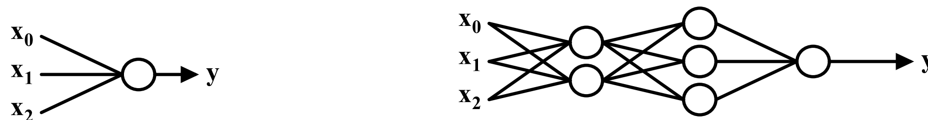
Figure 1.1: Illustration of neuron/synapse connectivity in human brain.

similar way. The artificial neuron is a single output of a layer in a NN. The value of the output is governed by a linear combination of its weighted inputs; these weights are often referred to as the synaptic weights of the neuron.

Fig. 1.2a shows a simplified diagram of a single neuron with (1.1) as the function relating the inputs to the output.

$$y = g(w_0 x_0 + w_1 x_1 + w_2 x_2) \tag{1.1}$$

Put simply, all of the inputs x_i are summed together, weighted by scalars w_i. The sum is usually passed through a nonlinear function g (i.e., the activation function) such as the hyperbolic tangent or sigmoid functions to get the output y. One of the most common layers used in NNs, the fully-connected layer, is formed by stacking multiple neurons such that every input is connected to each output neuron. In this configuration, the output of each neuron is a linear combination of all the inputs, which can be written in a simplified way as



(a) Artificial neuron model

(b) Multi-layer perceptron (MLP)

Figure 1.2: Block diagrams representing (a) an artificial neuron and (b) a multi-layer perceptron.

in (1.2), where m is the number of inputs, n is the number of outputs, and g is the activation function that follows the layer.

$$\bar{y} = g(W\bar{x}), \bar{x} \in \mathbb{R}^m, \bar{y} \in \mathbb{R}^n, W \in \mathbb{R}^{n \times m} \tag{1.2}$$

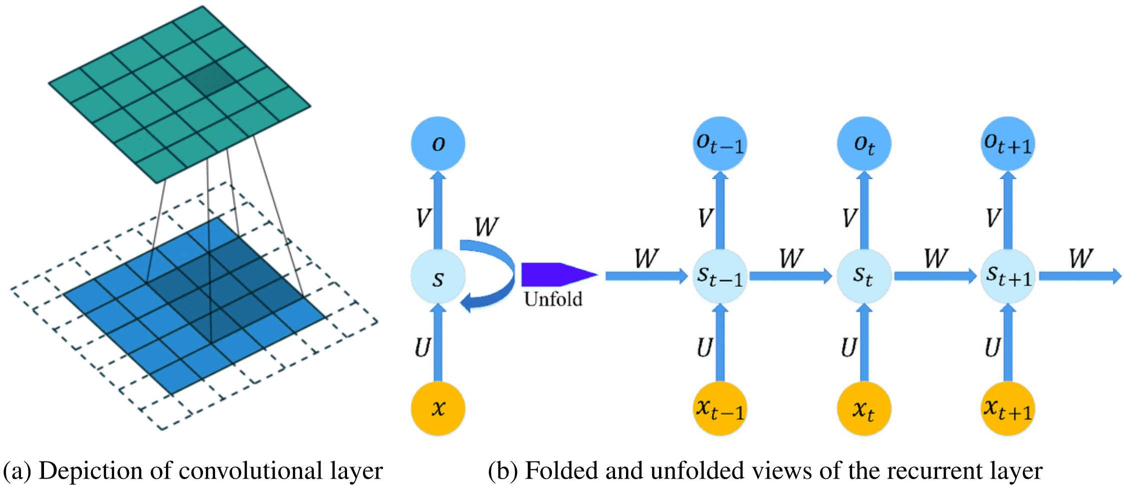
A sequence of fully-connected layers and non-linear activation functions makes up the multi-layer perceptron (MLP), shown in Fig. 1.2b, which is one of the earliest architectures used in both regression and classification tasks and is still widely used today.

The full equation for the network shown in Fig. 1.2b is given in (1.3). The equation is purposefully overcomplicated and redundant so that very complex relationships between variables can be approximated; depending on the actual transfer function, the network and its function may need to grow or shrink to provide a better fit.

$$\begin{aligned}
\bar{y} &= g_2(W\bar{s}) = g_2(Wg_1(V\bar{r})) = g_2(Wg_1(Vg_0(U\bar{x}))) \\
y &= g_2(w_0 s_0 + w_1 s_1 + w_2 s_2) \\
&= g_2(w_0(g_1(v_{00}r_0 + v_{01}r_1) + w_1(g_1(v_{10}r_0 + v_{11}r_1) + w_2(g_1(v_{20}r_0 + v_{21}r_1))) \\
&= g_2(w_0(g_1(v_{00}g_0(u_{00}x_0 + u_{01}x_1 + u_{02}x_2) + v_{01}g_0(u_{10}x_0 + u_{11}x_1 + u_{12}x_2)) \\
&\quad + w_1(g_1(v_{10}g_0(u_{00}x_0 + u_{01}x_1 + u_{02}x_2) + v_{11}g_0(u_{10}x_0 + u_{11}x_1 + u_{12}x_2)) \\
&\quad + w_2(g_1(v_{20}g_0(u_{00}x_0 + u_{01}x_1 + u_{02}x_2) + v_{21}g_0(u_{10}x_0 + u_{11}x_1 + u_{12}x_2))))
\end{aligned} \tag{1.3}$$

1.2.2 Common layer types

Different types of neural network layers work better with different types of data. As an example, the fully connected layer works well with data that can be represented as a vector, such as a list of attributes for a house or a plant. On the other hand, image data is two-dimensional (2D) and may contain important relationships between neighboring pixels that may not be captured when represented as a one-dimensional (1D) vector. Color images are usually represented by three images that contain information about the red, blue, and green



(a) Depiction of convolutional layer

(b) Folded and unfolded views of the recurrent layer

Figure 1.3: Depictions of convolutional and recurrent layers of neural networks. In (a), the green square on the top represents the output of the convolutional layer, the dark area on the lower blue square represents the area of the filter and the input values that are used to calculate the output (from [8]). In (b), a depiction of the recurrent layer shows how the layer can be "unfolded" to visualize how the different time steps in the input data interact (from [7]).

light intensity, making the image more like a three-dimensional (3D) object. Video and audio data include a time component, which becomes even more impractical to represent as a single vector.

Neural networks for image classification tasks have worked much better using two-dimensional synapse structures (known as filters). With these types of layers, each neuron output is a weighted linear combination of a 2D subset of the image. The filter is swept over the image horizontally and vertically to produce the full output, which is why these types of layers are called "convolutional" [6]. For data with temporal relationships, recurrent neural network layers provide a way to detect patterns and relationships between different points in time. A feedback path in the network uses both the current datapoint and a combination of the outputs from the data at earlier points in time to generate the current time output [7]. These and other types of layers are extremely important and provide more tools for working with all kinds of data media; for the work and purposes of this research, our focus will be on the multi-layer perceptron and fully-connected layers.

1.2.3 Activation functions

The activation function is a critical part of any neural network. Most real-world systems have transfer functions (i.e., input-output relationships) that are nonlinear, so in order to represent them accurately, the approximating function or decision boundary must also be nonlinear. Recall that the function that represents a fully-connected layer is a linear combination of the inputs to the layer for each neuron in the layer. The same is also true for convolutional and recurrent layers. As such, the best that a neural network made up of only these layers can achieve is a linear function.

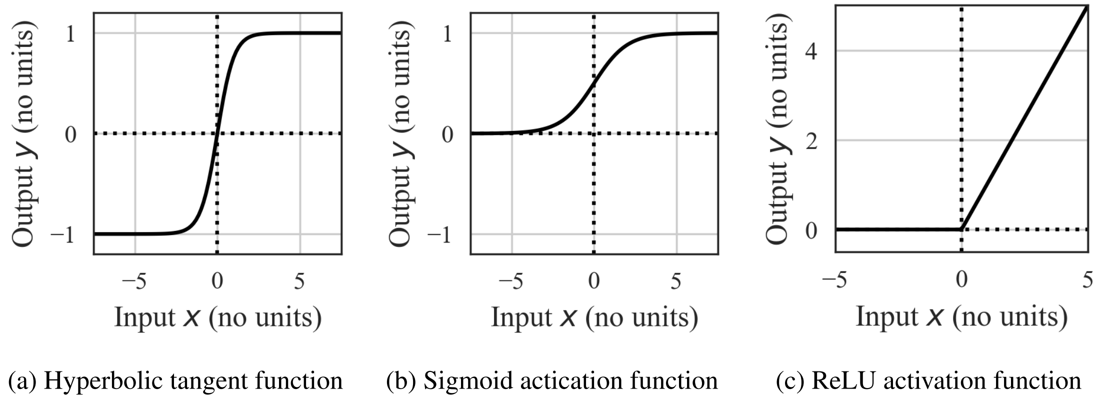
By inserting nonlinear functions in various points within the network (e.g., between layers), it becomes possible to approximate nonlinear functions as well, and adding more layers with more nonlinear functions adds to the amount of nonlinearity that can be estimated. This is the purpose of activation functions, which are represented by the g_i functions in (1.3). There are a number of activation functions that have been used in practice, most of which are described in [9], along with their benefits and problems and a comparison of their performance. Three of the most common activation functions are the hyperbolic tangent (1.4), sigmoid (1.5), and rectified linear unit (ReLU) (1.6) functions, shown in Fig. 1.4.

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1.4}$$

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{1.5}$$

$$ReLU(x) = \begin{cases} x, x > 0 \\ 0, x \leq 0 \end{cases} \tag{1.6}$$

Usually, the same activation function is used on the output of every neuron from a given layer. For example, if the ReLU function is used on the output of layer i, then the output of



(a) Hyperbolic tangent function (b) Sigmoid actication function (c) ReLU activation function

Figure 1.4: Three common activation functions used in nonlinear neural networks.

each neuron in the layer y_{ij} can be represented as in

$$y_{ij} = g_i(\bar{v}_j \bar{x}_i) \tag{1.7}$$

which, when all the output values are stacked in vector form, becomes (1.2).

1.3 General Overview of Machine Learning Algorithms

The goal of the machine learning algorithm is to set the synaptic weights of a neural network to values such that passing a set of input values to the network will generate an output that matches some expected result. Because the function that governs the input/output relationship (i.e., the transfer function) is unknown, the best that the network can do is approximate it.

1.3.1 Data Preprocessing

Before the data can be used to train the neural network, it usually needs to be scaled or processed into a more practical form. For example, audio data is difficult to interpret just as a string of values representing the signal magnitude, and some features may be on different orders of magnitude, such as the sqare-foot area of a house versus the number of bedrooms. There are a number of different ways to preprocess data to make it usable. While the

different means of extracting and preparing features in data are beyond the scope of this research, one example for how we created usable features for audio classification is given in appendix B. The importance of the scale of each of the features is covered more detail in the next sections.

1.3.2 Forward Propagation

The forward propagation step computes the initial predicted output $\hat{y}$ based on the inputs $\bar{x}$ and the current synaptic weights of the network. This step is so named because the inputs (i.e., the features of the input data) are propagated from one layer to the next until the output values are generated. These output values can be used to generate predictions, but it is these raw output values that are used during training to determine how to update the parameters. Since the outputs of each layer are simply linear combinations, the math functions needed for forward propagation are multiplication, addition, and the activation functions (see (1.3)).

1.3.3 Loss Function

The output from the forward propagation step $\hat{y}$ is compared to the list of ground-truth values or classes y to calculate an error or "loss" value for the network and training data. The loss function is determined by the task and can be customized based on the nature of the dataset, the class priority, and other training adjustments such as regularization. Almost any differentiable function can be used as long as it increases with error and decreases as the network predicts more accurately, though some functions work better than others. For binary classification tasks, the cross entropy loss function shown in (1.8) is most commonly used, while mean-squared error loss (1.9) is most common for regression. In general, the operations required for loss calculations are addition, subtraction, multiplication, integration, and a nonlinearity such as log or square.

$$loss = \frac{1}{M} \sum^{M} ylog(p) + (1 - y)log(1 - p) \tag{1.8}$$

$$loss = \frac{1}{M}\sum^{M}(y-p)^2 \tag{1.9}$$

1.3.4 Backpropagation

Backpropagation is the process of determining how to adjust internal parameters to reduce the loss function. This can be done by individually perturbing every weight by a small amount and calculating the amount of change in the loss, but there is a much more efficient means of calculating all of the gradients which takes advantage of the linear algebra (i.e., matrix multiplications) within the network as well as the chain rule in differentiation.

The name "backpropagation" is derived from the way these gradients are calculated. First, the derivative of the loss is calculated with respect to the last layer of the network to obtain the gradients Δ_k for the last layers synaptic weights W_k. Using the chain rule and the gradients Δ_k, the derivative with respect to the next-to-last layer is calculated to obtain Δ_{k-1}. This iterative process is repeated as the gradients Δ_i are propagated backward to the very first layer. The math involved in this process is outside the scope of this work, but the functions required are addition, multiplication, and the derivative(s) of the activation functions(s) used in the network.

1.3.5 Parameter Update

Finally, all the parameters are updated in the network using the calculated gradients. To decrease the error of network, they are adjusted in the opposite direction of the slope calculated during backpropagation and at a fraction of the full slope with the fraction determined by a parameter called the "learning rate" α (1.10).

$$W_i' = W_i - \alpha\Delta_i \tag{1.10}$$

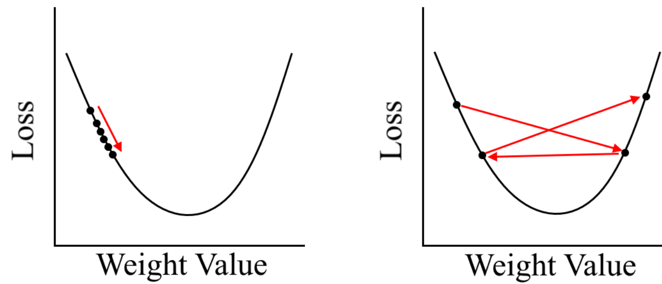


Figure 1.5: Consequences of using a learning rate that is too small (left) or too large (right)

The learning rate can have a huge impact on the training performance of the network as a small value will cause the network to take longer to train, but a large value may prevent the network from converging at the optimal solution (see Fig. 1.5). To avoid this, the learning rate can start large and shrink over time as the loss decreases.

The parameter update method just discussed is known as gradient descent. It works very well in most cases, but it does have a tendency to get stuck in non-optimal local minima instead of the global minima. There are many other optimization algorithms available to help get around this problem, though this is not discussed in detail in this work. For the four steps needed to train a network, the mathematical functions needed are addition, subtraction, multiplication, integration, the activation functions and their derivatives, and a nonlinearity for the loss function. This is important because it shows that the process of machine learning can achieve successful results representing very complex functions using only simple operations.

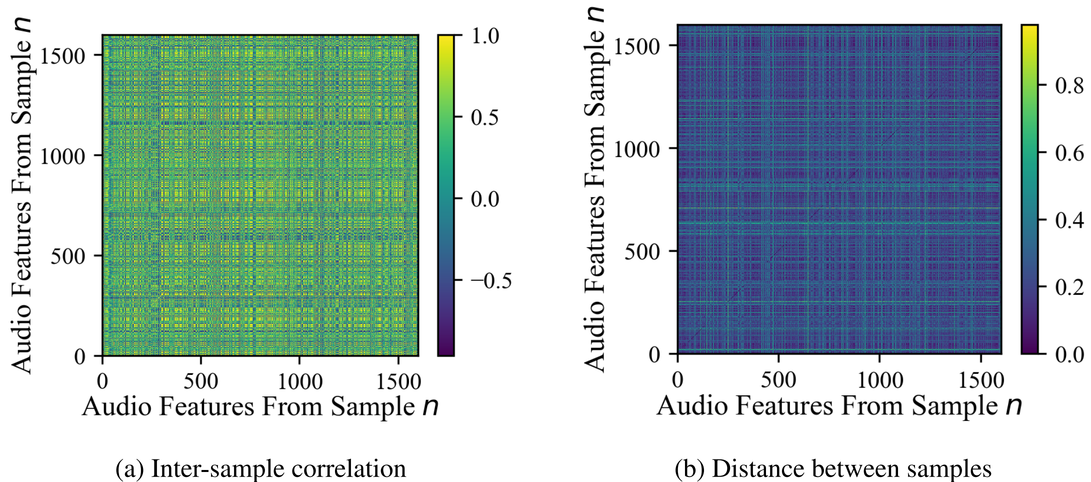
1.4 Mini-batch Training: Redundant Training Sets

For two of the datasets that were used in our experiments, the data was audio MP3 files, all of which were sampled at 16 kSps. The dominant frequencies for human voice are in the low hundreds of Hz (i.e., 100-400) when considering all ages and genders, so this sampling rate should be more than adequate to capture both the fundamental frequencies and other frequency components of voice. Because our application of voice activity detection required being able to detect human speech at any moment in time, five seconds of audio,

enough for a short phrase to be spoken, would contain 80,000 samples for the training or test set. Taking into account multiple possibilities for signal-to-noise ratio (SNR), speakers, vocal fluctuations, and different noise sources, the number of audio clips needed to perform reliable grows to an unreasonably high number, easily more than billions of samples for a thorough dataset.

Fortunately, it is not necessary to use every sample in the input data for each weight update step. In our case, the features we were using were dependent on the envelope of the audio rather than the actual signal value. The envelope changes much less drastically than the audio signal, so many neighboring samples are very similar to one another, and with the same speaker and background noise, it is likely that there will be many similar input values throughout the audio clip.

Fig. 1.6 shows an example of how similar the features are for an audio clip generated using the Microsoft DNS Challenge dataset and library. There are 1600 randomly selected samples from over 30 seconds (480000 samples) of audio. Fig. 1.6a is a color map of how correlated each sample is to each of the other samples. The correlation value ranges from -1 to 1, where 0 represents no correlation and -1 and 1 represent perfectly negative or positive



(a) Inter-sample correlation

(b) Distance between samples

Figure 1.6: Color map plots showing how many of the samples used for training closely resemble one another, both in terms of correlation and euclidean distance. In (a), the light colors represent high positive correlation. In (b), the dark colors represent a small euclidean distance between samples.

correlation respectively. Most of the plot is light green or yellow, which shows that many of the feature vectors are highly-correlated with one another. Fig. 1.6b is a color map that shows the measure of the euclidean distance between each feature vector. Since most of the plot is darkly-colored, it shows that most of the feature vectors are not far apart from one another.

When training a neural network with traditional gradient descent, the machine learning algorithm uses all data from the training dataset to calculate the loss and the gradients for each trainable parameter in the network. The gradient of the loss is calculated for each sample from the data and averaged together to get the final result. The problem with this is that the neural network must process every input before taking the next step. Because there is so much correlation between the samples in our dataset, it may seem that we can use only a small fraction of the dataset to reduce the amount of time and energy required to train the neural network However, it not usually obvious which samples are the most critical, and randomly selecting a subset to use for training will lead to a significant difference in performance between the fractional training set and the rest of the data. Fig. 1.7 demonstrates that randomly selecting a subset of the data will lead to a difference in performance between the subset and the full training set, such that the full training set essentially becomes a

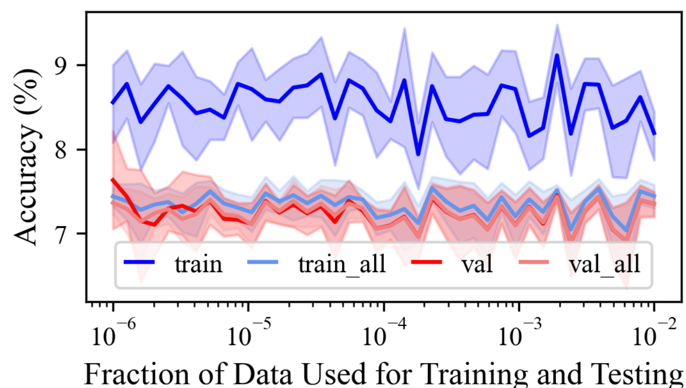


Figure 1.7: This plot shows how repetition in data does not justify randomly selecting a subset to use for training. The blue lines are the training datasets and training and the red lines represent the validation data. The light colored area surrounding each line shows how the results varied over ten different tests.

validation set. Rather than sorting through the data to hand-pick which to use, it is more effective to rotate through all of the samples in the data set but in smaller batches.

Research has shown that a neural network can be trained effectively by processing only a portion of the training data before taking an update step and then looping through the entire dataset in these smaller batch sizes [10]. This method of training became known as "mini-batch gradient descent". Stochastic gradient descent takes this even further by taking using only one random data point for each update step. Because of the similarities in the inputs, the algorithm is able to move the weights of the neural network in the general direction of the desired final state much more quickly because it is taking steps more frequently. The path to the end result is not as direct, but the final accuracy is just as good as the accuracy would have been if the entire batch of inputs were used for training. Fig. 1.8 shows how, with the Microsoft DNS Challenge audio, we can reliably train and test the neural network using only 0.02% of the data for each step with good matching to using the entire dataset, and we have almost perfect matching using 1% of the data.

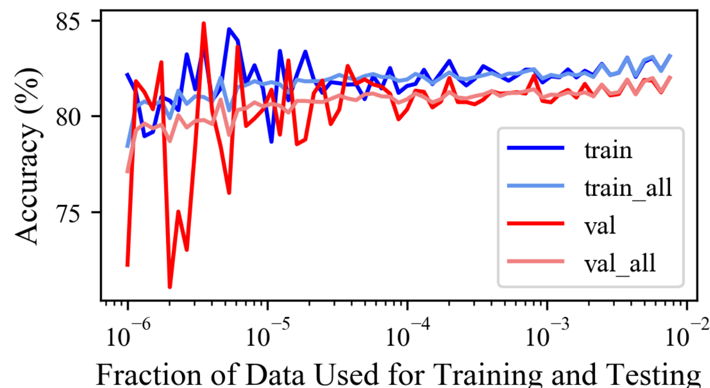


Figure 1.8: This plot shows how only a small fraction of the Microsoft DNS Challenge dataset can be used for each gradient calculation while still expecting the performance to match what could be attained if the entire dataset was used for training. The blue lines represent the portion of the dataset used for training and the red lines represent the portion that was set aside for validation.

CHAPTER 2

IMPLEMENTING NEURAL NETWORKS

2.1 Digital Implementation

Almost all modern machine learning is implemented in digital hardware. Whereas analog computation was more established in the mid-1900s, digital storage and computing quickly overtook it in terms of density and accuracy as CMOS technologies progressed at a rate famously observed and predicted by Gordon Moore [11]. This explosion of the capabilities of digital systems is what has allowed modern machine learning to be so successful.

Hundreds and thousands of gigabytes of information can be stored in chips smaller than a US dime, and CPUs with four or more processing threads are common in practically all smartphones. For tasks outside of the capabilities of portable devices, data can be transmitted and processed with remote cloud computing services on large amounts of computation hardware, such as Google Cloud's TPU pods. Regardless of the platform, practically all general purpose digital hardware is capable of performing all the mathematical operations required to train a neural network.

Unfortunately, the price for added computational power is just that: power. Progress in increasing performance of digital systems is approaching a wall [5, 12]. Traditional transistors are reaching the physical limitations of silicon, and practices of digital architectures such as the standard Von Neumann have speed and energy issues due to repeated off-chip memory access [5, 13, 14] as shown in Fig. 2.1. Aside from that, the circuit blocks for performing basic arithmetic such as addition and multiplication tend to be made up of a large number of transistors.

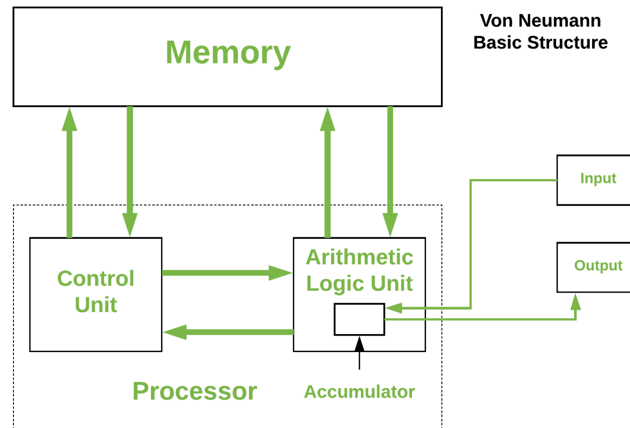
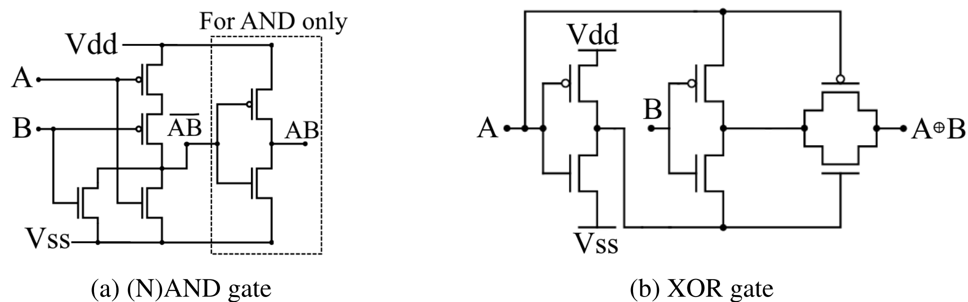


Figure 2.1: Diagram of the basic Von Neumann architecture for flexible computer processing. When multiplying a vector by a matrix, where the matrix is stored in memory, each number in the input vector is loaded one at a time and multiplied with each associated value in the matrix, which also must be iteratively loaded from memory. Delay and loss are associated with the data request and transmission along the interconnecting data lines. Taken from [15].

2.1.1 Binary Arithmetic

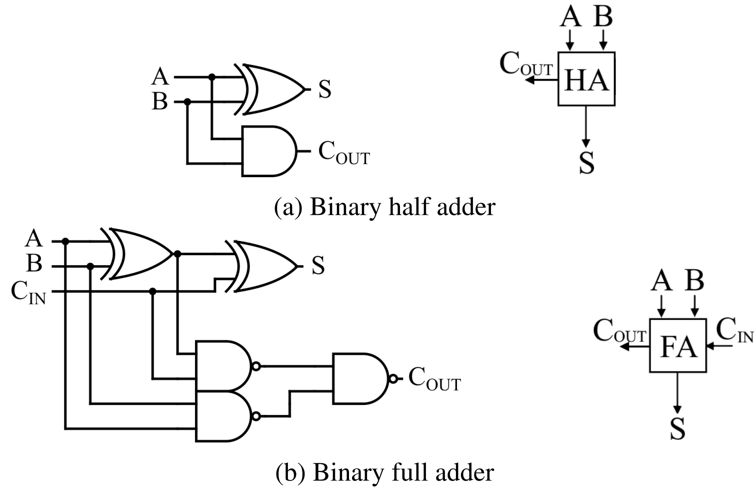
The basic units for binary arithmetic are the full adder and half adder, which are made up of AND, NAND, and XOR logic gates. The schematic diagrams for each of these logic gates are shown in Fig. 2.2. The half adder, shown in Fig. 2.3a, takes in two bits and outputs two one-bit values: the sum and the carry-over bits. It takes one AND gate and one XOR gate to form this logic block for a total of 12 transistors. The full adder in Fig. 2.3b takes two input bits and a carry-in bit to generate the sum and carry-out bits. This logic block is made up of two XOR gates and three NAND gates, which takes 24 transistors.



(a) (N)AND gate

(b) XOR gate

Figure 2.2: Transistor diagrams for (N)AND and XOR binary logic gates.



(a) Binary half adder

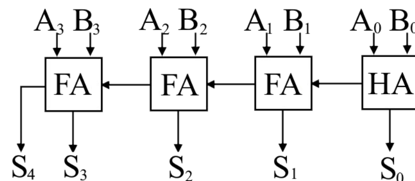
(b) Binary full adder

Figure 2.3: Gate- and block-level diagrams for the binary half and full adders.

Adding two N-bit binary numbers can be accomplished by chaining together $N - 1$ full adders and one half adder, as shown in Fig. 2.4b. This particular adder is referred to as the “ripple” adder because of how the carry-out bit must be calculated before the next block can complete its own sum and carry-out values, making the output calculation ripple across the blocks. To reduce the delay time caused by the rippling calculation, the adder can calculate all of the carry-out bits independently so that each adder’s output can be generated in parallel, but this requires significantly more logic gates. The number of transistors T_{ADD}

$$
\begin{array}{cccccc}
 & 1 & 1 & 1 & 0 & \\
 & & 1 & 0 & 1 & 1 \\
+ & & 0 & 1 & 1 & 0 \\
\hline
 & 1 & 0 & 0 & 0 & 1
\end{array}
$$

(a) Numerical binary addition



(b) Chaining four adder blocks together to add two four-bit binary numbers

Figure 2.4: This figure shows an example of how binary addition is carried out with two four-bit binary numbers. The lighter-colored numbers in the top row of the upper figure are the carry-over bits. The lower figure is a block diagram for how three full adders and one half adder can be chained together to create a four-bit ripple adder.

needed for a an N-bit ripple adder can be calculated as

$$\begin{aligned} T_{ADD} &= (N-1)T_{full} + T_{half} \\ &= (N-1)[2T_{XOR} + 3T_{NAND}] + T_{XOR} + T_{AND} \\ &= (2N-1)T_{XOR} + 3(N-1)T_{NAND} + T_{AND}. \end{aligned} \tag{2.1}$$

Binary multiplication can also be accomplished using these basic adder blocks. By following the same pattern for binary multiplication as shown in 2.5a, a simple multiplier using N-bit ripple adders can be generated as in 2.5b. This multiplier is intended for unsigned binary numbers, so different architectures are needed for signed or floating point numbers. However, using unsigned-only architectures will still give a relatively accurate view of how large and complex digital arithmetic circuits can be.

For a N-bit multiplier, the transistor count T_{MULT} can be calculated as

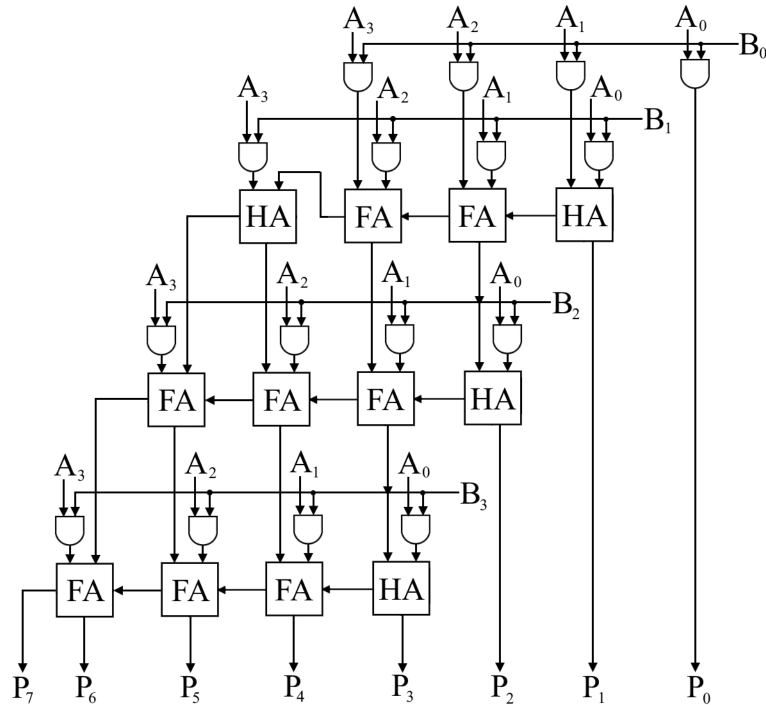
$$\begin{aligned} T_{MULT} &= (N^2 - 2N)T_{full} + NT_{half} + N^2 T_{AND} \\ &= (N^2 - 2N)(2T_{XOR} + 3T_{NAND}) + N(T_{XOR} + T_{AND}) + N^2 T_{AND} \\ &= (2N^2 - 3N)T_{XOR} + (3N^2 - 6N)T_{NAND} + (N^2 + N)T_{AND}. \end{aligned} \tag{2.2}$$

The plots in Fig. 2.6 show how the number of transistors grows to accommodate higher-bit arithmetic. While the relationship is linear for the adder, there is a quadratic increase in the transistor count for the multiplier as the number of bits increases. As stated before, there are alternate architectures that can change the speed and reduce the number of transistors, and these numbers do not directly apply to other binary number formats. However, even an optimized 32-bit float multiplier requires over 20,000 transistors [16], which is close to what is shown in the lower plot in Fig. 2.6.

```
                1   0   1   1
            x   0   1   1   0
-----------------------------

    1   1   1   1   0   0
                1   0   1   1   x 0
+           1   0   1   1       x 1
+       1   0   1   1           x 1
+   1   0   1   1               x 0
-----------------------------
    1   0   0   0   0   1   0
```

(a) Numerical binary multiplication



(b) Block diagram for four-bit unsigned binary multiplier

Figure 2.5: This figure shows an example of how binary multiplication is carried out with two four-bit binary numbers. The lighter-colored numbers in the upper figure are the carry-over bits. The lower figure is a block diagram for how three full adders and one half adder can be chained together to create a four-bit ripple adder.

2.2 Powering Down

Significant research has been invested into methods of reducing the amount of energy required by neural networks, both in terms of training and inferencing with models. Lowering numeric precision has proven promising for many applications. Instead of using 32- or 64-bit numbers, some researchers have found that a neural network only needs to work

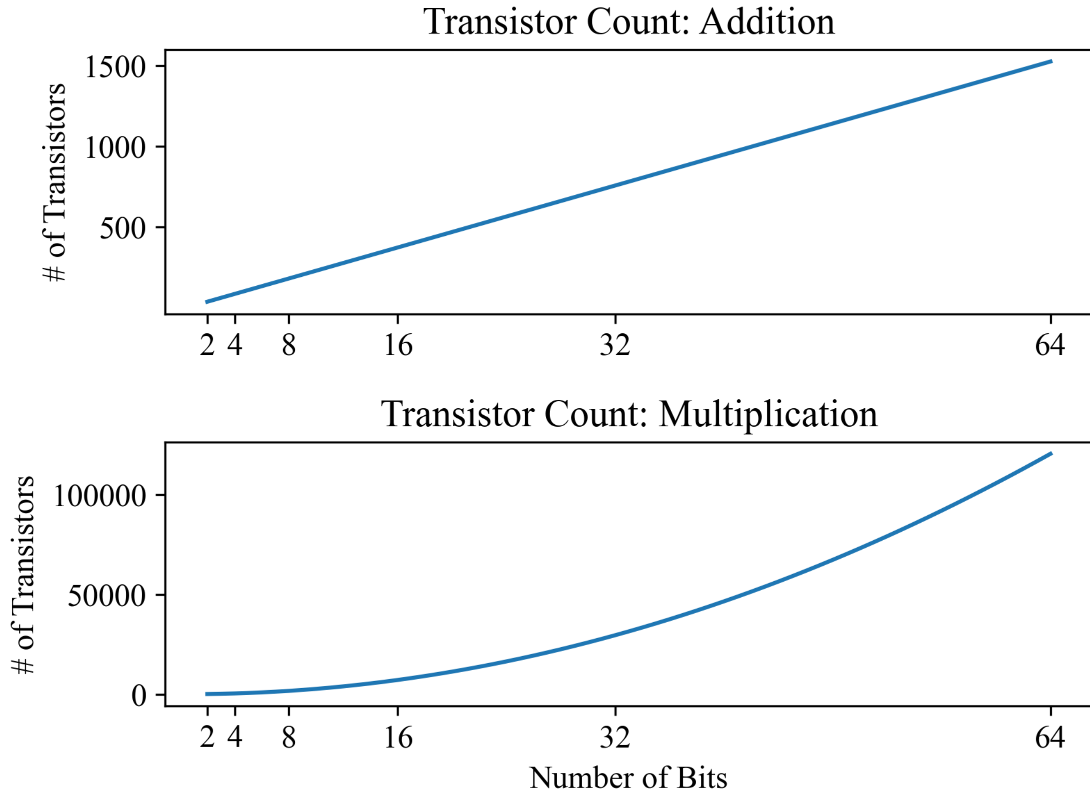


Figure 2.6: Plots showing the relationship between tranisistor count and binary number bit length.

with four or eight bits of precision with little to no loss in performance [17]. An alternative neural network architecture that has been explored more recently is the binarized neural network (BNN) [12, 18–23]. This type of neural network takes an extreme approach to precision reduction. For BNNs, the forward propagation step is performed with weights and/or activation function outputs fixed to either -1 or 1. In the backpropagation step, the actual weights and gradients are stored with full precision and clipped to stay within the (-1, 1) range. These networks have drastically reduced energy consumption, and the accuracy is reduced only by a small amount in many cases.

Another method for reducing computation energy requirements is eliminating unneeded connections in a neural network (i.e., sparsity) [24]. Once again, this method was taken by looking at how biological brains develop. Throughout the early years of human development, the number of neurons in the brain increases, but after reaching some age, the number starts

to decrease. This suggests that the brain eliminates unneeded or redundant neurons or synaptic connections while still maintaining similar or better performance. Sparsity will be covered in greater detail in chapter .

Some researchers have shown that it is possible to shrink a neural network down to a smaller size with minimal loss in accuracy by changing the reference values used for the "ground truth" when computing the loss function. Instead of using the labels from the dataset, the class labels predicted by a larger pre-trained neural network are used to train a smaller or compressed version of the network. This is known as Knowledge Transfer [25]. An alternative method, Knowledge Distillation, uses the raw output values, of the larger network rather than the class labels [25].

Aside from techniques that reduce the size or complexity of the model or training algorithm, there is also the possibility of using specialized hardware. Most computers use CPUs and GPUs, which are general-purpose computing devices. However, the generality of the hardware means that there is excess hardware that may not be needed for neural networks, increasing the size and power required. Tensor processing units (TPUs) are better suited for matrix and vector operations. As such, they are capable of completing an order of magnitude more operations for the same amount of power as GPUs, and CPUs (see Fig. 2.7) [26].

Neural network architectures that operate using different mechanisms than standard digital arithmetic can also lead to energy savings. In an attempt to more closely mimic the brain, Spiking Neural Networks encode information into continuous-time voltage spike trains. Because of this, the computation only occurs when spikes are present, and it can run asynchronously and in discrete sections. These networks rely on integrate-and-fire circuits, which are not present in standard computational hardware; special neuromorphic hardware is used for deployment [27]. Finally, there has been also been a movement to return to the roots of electronic computing by re-exploring analog circuits.

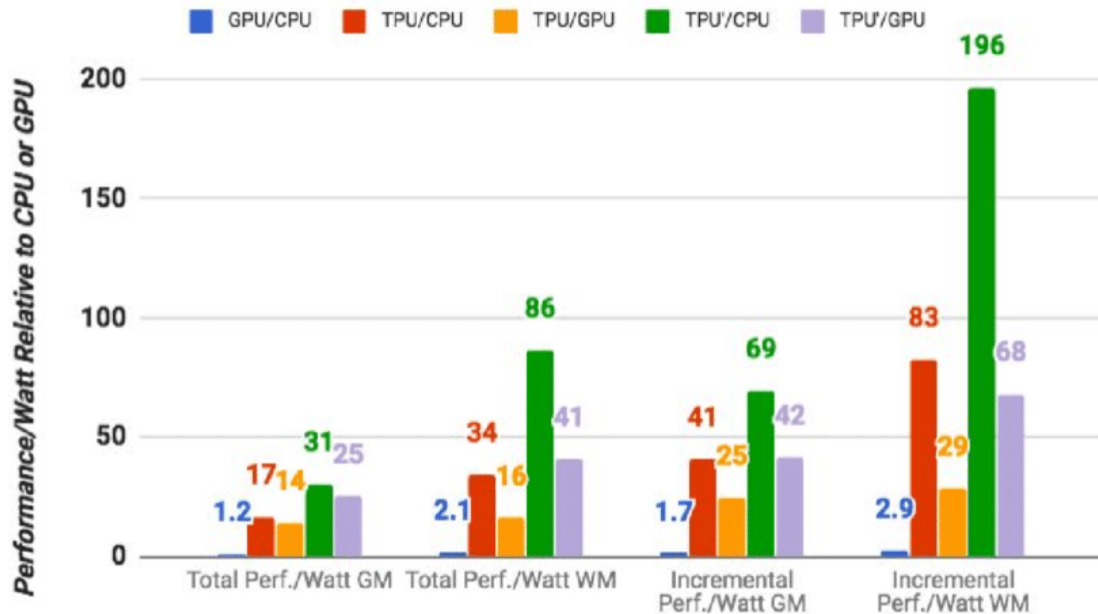
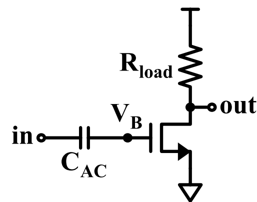


Figure 2.7: Plot illustrating the relative performance per power for GPUs and TPUs in relation to CPUs. The TPU' represents a second-generation improved TPU core. For more information about this plot, see [26].

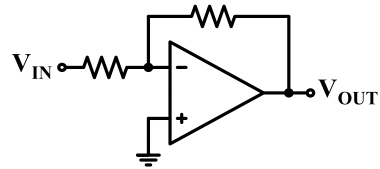
2.3 Analog Hardware

2.3.1 Arithmetic blocks

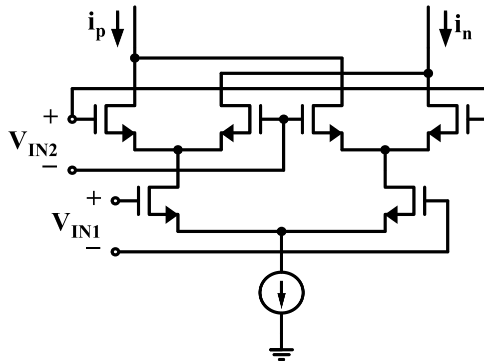
Although analog computing was largely abandoned for the digital revolution, many have been looking to it as the solution to the limitations with current digital machine learning. Rather than relying on digital arithmetic and logic, analog computing uses the physical properties of semiconductor devices [28]. An operation such as multiplication, which requires thousands of transistors in digital, can be reduced to only a handful of transistors in analog. Fig. 2.8 illustrates the small number of transistors that can be used to create a pure analog multiplier. The number of transistors in the multiplier depends on the specifications required, such as linearity and whether the synaptic weight (i.e., gain) should be able to represent either positive or negative values but the opamp represented in Fig. 2.8b can use five to seven transistors (about as much as an AND gate) or as much as ten to twenty for



(a) Single-transistor analog multiplier



(b) Analog OpAmp multiplier

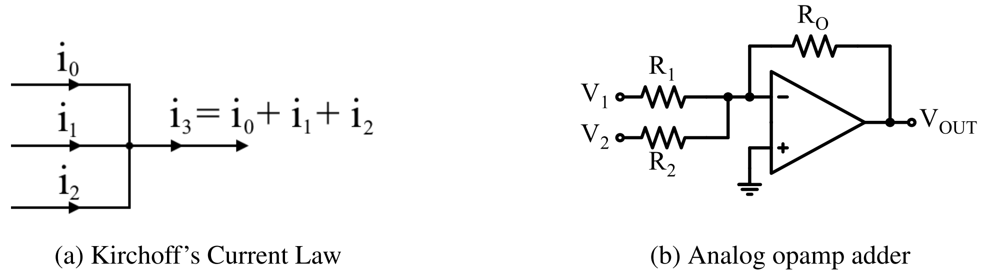


(c) Input multiplying stage of a Gilbert multiplier

Figure 2.8: Block and transistor diagrams for components in analog multipliers. (a) is the basic single-transistor multiplier, which supports limited two-quadrant multiplication. (b) is a more complex operational amplifier based multiplier, which offers more flexibility and benefits such as extended linearity. (c) is the Gilbert multiplier, which is used when multiplying two analog values that can take on both positive or negative values, such as in RF mixers.

Gilbert multipliers for four-quadrant multiplication as in Fig. 2.8c.

Other mathematic operations such as summation and integration can also be implemented with analog devices with similar transistor count reduction. According to Kirchoff's current law, the current in one conductive path (i.e., wire) is equal to the summation of the currents in all other connected wires, as in Fig. 2.9a. Using this law, we can build a simple summation block using only a few devices that regulate current flow into a wire. Fig. 2.9b shows how this can be done using resistors to set the summation weights. The input voltages on the left are translated to currents by their respective resistors, and the currents are summed together at the input of the opamp. The total current is translated back to a voltage by the feedback



(a) Kirchoff's Current Law

(b) Analog opamp adder

Figure 2.9: Current summation is a built-in property of electronic devices and can be used to add voltages translated to currents through resistors.

resistor above the opamp. This relationship can be represented as

$$v_{OUT} = \frac{R_1}{R_O}v_1 + \frac{R_2}{R_O}v_2 \tag{2.3}$$

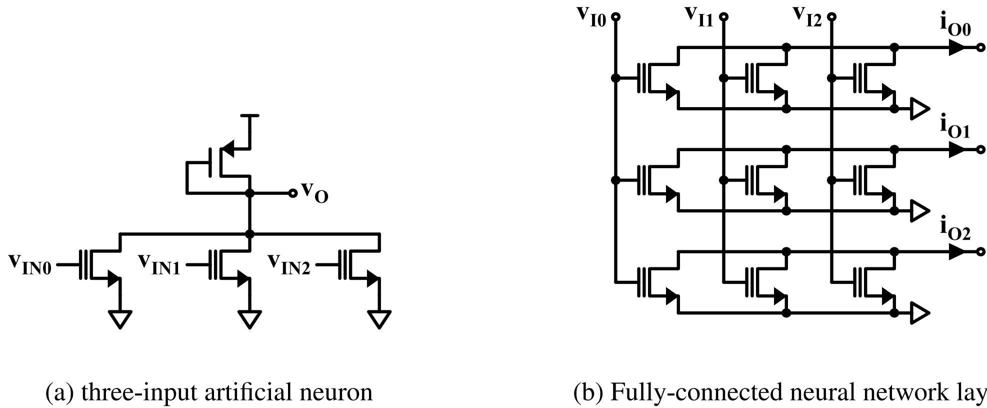
The voltage across a capacitor v_{cap} is approximated as

$$v_{cap} = \int_{t_1}^{t_2} i_{cap} dt \tag{2.4}$$

where i_{cap} is the current into the capacitor, so with one component, we can integrate currents over time. Because of the reduced transistor count and simple structures, analog circuits are comparatively low-power, faster, and smaller than their digital counterparts [12, 28–38]. Using these simple architectures, it is possible to create an artificial neuron and a vector-matrix multiplier (i.e., fully-connected layer) as in Fig. 2.10.

Analog circuits are also not limited by clocking or serial computation, where one set of instructions must complete before another set can start, unlike their digital counterparts. Multiple system components will operate in parallel independent of other blocks, further enhancing overall computation speed [37]. This is normally only possible in digital if there are multiple independent processing cores (or threads), and even then, the parallelism is not as massive.

A research team at IBM has recently demonstrated all of these benefits by employing analog in-memory computation [39]. As stated earlier, memory access is an expensive



(a) three-input artificial neuron

(b) Fully-connected neural network layer

Figure 2.10: Three simple architectures that can act as (a) an artificial neuron and (b) a complete fully-connected layer in a neural network.

process for high-speed systems, both in terms of time and energy. The large blocks needed for digital logic and arithmetic necessitate this separation. The inherent smaller size and built-in parallelism of analog circuits allows us to build the multipliers and adders right next to each other, elimitating the read and load times for multiplicative weights. It is also possible to store the weights as voltages directly on the multipliers, as in V_B in Fig. 2.8a.

2.3.2 Limitations

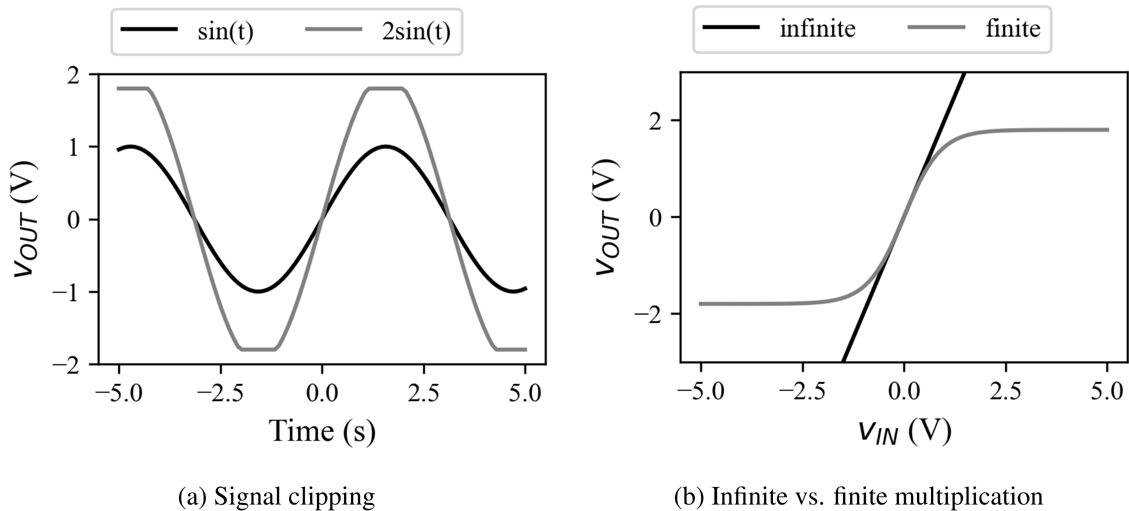
Despite all these benefits that analog can offer to computing, it has not been widely adopted for several important reasons. The most significant limitation in analog circuits is the inter-device variability. Due to small inconsistencies in the fabrication process, voltage sources, and temperature, often referred to collectively as PVT variation, post-fabrication, devices such as capacitors and resistors can vary up to 25% from their expected values making it difficult to know their actual properties and causing offset and mismatch errors [40–42].

The relationships between physical properties of semiconductor devices are nonlinear and have limited ranges, which may lead to unexpected operation. For example, if the circuit is operating on a voltage supply range of -1.8 Volts to 1.8 Volts and we want to multiply a unit sine wave by two, it would be impossible to correctly represent the sine wave using the analog circuit because the range is restricted to a peak magnitude of 1.8 Volts. The resulting

sine wave would be clipped as in Fig. 2.11a. The actual transfer function for a multiplier is not a straight line as expected for linear systems; it is more like a hyperbolic tangent function, where smaller inputs may be multiplied by an amount similar to the expected gain, but larger inputs would become distorted by the limited range (see Fig. 2.11b). The range of input values allowed with minimal distortion in te output is referred to as the input dynamic range.

Analog also has no perfect long-term storage; most storage methods suffer from leakage, low-precision, or limitations in writability [12, 37, 43]. Capacitors are ubiquitous in physical circuits, whether they are placed intentionally or as parasitic effects of the manufacturing process. While they are very useful for short-term voltage storage, they do not work well as long-term storage. Capacitors are typically charged through "switches", or transistors turned on and off. When the "switch" opens, it actually becomes a high-resistance path rather than an open circuit. If there is any difference in voltage between the two sides of the transistor, current will leak through and change the value over time.

Another option is floating gate transistors. These are similar to the transistor in Fig. 2.8a, but the capacitor C_{AC} is built into the transistor with the intermediate gate completely



(a) Signal clipping

(b) Infinite vs. finite multiplication

Figure 2.11: (a) The clipping of a sinusoid multiplied by 2 inside a limited voltage range. (b) The black line is a linear function, $y = 2x$, and the grey line represents what an actual analog $2x$ multiplier would look like.

isolated from any conductive path (see Fig. 2.12). The electrical isolation allows the voltage to stay virtually unchanged for over a decade. The voltage is set on the floating gate by using second-order effects of CMOS transistors. Using hot-electron injection, the charge (voltage) is reduced. Electron tunneling removes negative charge from the gate, reducing the voltage. Both of these processes require very-high voltages, which are difficult to generate on-chip and degrade the device oxide, leading to more device property changes. A more recent alternative for long-term storage is the memristor, which is used in the memory array in [39]. As the name suggests, it is essentially a variable resistor that holds its value even when disconnected from the rest of the circuit. While it has its own challenges, it has been used with a lot of success. For the remainder of this dissertation, it will be assumed that floating-gate transistors are being used to store the multiplicative weights.

The challenges described here are the reason why analog circuits are not as flexible as their digital counterparts [12, 29, 30, 40], and training methods such as backpropagation are more difficult to implement on chip [40, 44]. All these issues limit analog computation to applications with low- to medium-precision data [12, 29, 42, 45].

2.4 Introducing Analog Into Machine Learning and Neural Networks

The techniques used to minimize the inherent issues depend on the approach taken to introduce analog into the computation and classification flow of common machine learning architectures. The most common ways are via full replacement, computation acceleration, and deployment.

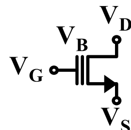


Figure 2.12: CMOS floating gate transistor

2.4.1 Full Replacement

Introducing analog circuits by full replacement is the effort to complete the entire machine learning process (i.e., forward computation, backpropagation, weight learning/storage, and classification) entirely with analog blocks. The basic functions needed to implement ML are inherently available in hardware, including summation, multiplication by scalar, and nonlinear functions. Integrals and derivatives are also computable using common circuit components. Using all of these, it is theoretically possible to train and use an analog classifier implementing many popular ML algorithms.

This approach is very desirable, especially for power-starved computing-on-the-edge applications. Allowing disconnected devices to independently handle sensor output would significantly reduce the power drain from constantly broadcasting data to a base station for processing [4], and it would increase the level of privacy and security for such devices. These classifiers would be able to learn from their immediate environment and then make decisions without ever leaving the field. Because the actual device undergoes training, the classifier learns in the presence of the device variations, which results in performance competitive with an ideal digital classifier [33, 37, 43, 44, 46–52]. Many prototypes are reported in research with surprising amounts of success, and quite a few commercial products have been developed.

The biggest drawback for these types of classifiers is that the information learned is very difficult to share and reproduce. Because each fabricated circuit can vary significantly from another, the multiplicative weights learned on one chip are unlikely to directly translate to another without some amount of retraining. Being able to read the weight values is another issue entirely as each read process can also introduce noise that corrupts the learned value [42]. Because of this, the classifier becomes a black box, where very little intuition can be gained about what has been learned. Each device must be trained by the end user and is not ready for immediate use.

Backpropagation, one of the most common training algorithms, can be very difficult to

implement accurately in analog. Although it is possible to calculate gradients, it is difficult to design a reliable differentiation circuit because they are heavily influenced by device variations and noise. Some have managed to use backpropagation on chip [30, 48, 53, 54], but others have opted for alternative learning rules such as weight perturbation learning [31, 44, 55, 56], which randomly adjusts each weight in series or parallel and keeps the weights that result in better performance. This algorithm takes much longer to converge, but it is much easier to implement in analog.

2.4.2 Computation Acceleration

Network training is by far the most demanding step in machine learning in terms of power and resources. As the boundaries of what digital can do have become more difficult to push, many have looked to analog hardware to accelerate the learning process while also reducing the power required [13, 39, 57]. Rather than completely do away with the entire digital system, this approach seeks to augment the capabilities of the system by using analog circuits for the most demanding tasks, such as matrix multiply-and-accumulate functions. However, while this may accelerate the actual computation, latency is introduced due to the required data conversion.

Except in the case of multi-core systems, digital calculations are completed serially—the next set of multiplications cannot start until the current set has completed. Part of the reason for this is because digital architectures make use of resource sharing, which allows the same hardware to be used for multiple tasks at different time intervals. A major time and energy bottleneck in digital classifiers is the reading and loading of synaptic weight values from digital memory to the multiplier blocks [39]. This is a problem both during training and in deployment. On the other hand, analog is inherently parallel. [39] demonstrates that the entire vector-matrix multiplication step in forward propagation can be simplified to a single parallelized step by implementing analog multipliers within analog memory, thus eliminating the read and load steps. Since far fewer components are required to perform

arithmetic tasks, massive parallelism is very easy to implement when computing with analog circuits.

The most common way to implement this is to convert sensor data using an analog-to-digital converter (ADC), perform any necessary pre-processing on the data in the digital domain, and then convert each value back to analog using digital-to-analog converters (DACs). In order to store results and interface with the digital system, values will also need to be converted back with an ADC. To ensure accuracy, each of these DACs and ADCs must either be trimmed, calibrated, and designed with large silicon real estate. Each DAC or ADC also introduces latency to the training path, which offsets the speed benefit from using analog. Some applications have suggested completing the feature extraction and pre-processing stages in analog as well, eliminating one ADC and one DAC [20].

One huge issue with using analog computation in a digital system is that high linearity and precision are required. For the acceleration to be useful, there must be a direct translation from analog circuit to digital model, which requires precise trimming and control of each data converter and arithmetic block. This level of precision is especially required if the intention is to use the learned weight values in a digital system or other hardware classifiers for deployment. Full replacement and acceleration are both very important steps toward low-power machine learning, but they have significant hurdles to becoming reality.

2.4.3 Deployment

The third method for replacing digital with analog in machine learning systems is to implement the trained digital system as an analog classifier. In this case, the training is completed on a digital system such as a computer, and the learned network characteristics (including feature extraction, and multiplicative weights) are downloaded to the device [13, 19, 28, 32, 36, 58–60]. After the device is programmed, it is ready for use in its selected application. There are three benefits to this approach. First, the deployed classifiers are inherently low-power compared to contemporary digital systems and can operate for

much longer. Second, the same set of weights can be used for all devices, improving the manufacturing throughput. Third, the weights are externally programmable, allowing for some flexibility should the weights need updating or if device-level trimming is implemented.

Just as with the other methods, the best set of parameters for one device is not the best for another due to process variations and other randomness such as when programming weights onto analog storage. For the same reason, the trained weights of an ideal system cannot be downloaded to an analog classifier and produce the same accuracy. To minimize the performance degradation due to these nonidealities, the architecture and training algorithm must generalize well to all variations that could occur within the classifier [61].

2.5 Training-to-Deployment Translation

For any analog classifier to be useful, the weights it learns or is programmed with need to result in good accuracy, but each device will perform differently even if programmed with the exact same voltages. This is one benefit of training on the manufactured chip, as in the full-replacement option. During the training, the weights will learn to account for all the variations and non-idealities, maximizing the accuracy for each individual device, but this is very time-consuming and significantly increases manufacturing costs and device complexity [61, 62].

To overcome this, a few groups have looked to transfer learning, which is the process of taking a neural network trained for a specific task and re-purposing it for a similar one. The assumption is made that the network is close to where it needs to be since the new application is not very different, so only the last few layers are re-trained. This significantly reduces the power and time required for training. In hardware, a similar approach can be taken. After the network is trained either digitally or on other hardware, the weights can be transferred to similar but different hardware and only the last few layers retrained [39, 50]. While this does reduce the penalty and challenge of on-chip training, the actual implmentation, especially for backpropagation, is another hurdle by itself. Therefore, it is

desirable to have one set of weights that works with good accuracy on all devices without retraining.

To determine the required parameters, the circuit and the model used in training must match well [28, 58, 63]. Some have worked to force the circuit, including the multipliers and activation functions, to match the model [28, 53, 63–67], but this requires unreasonable amounts of careful design and silicon real estate. It is more logical to characterize the device using predetermined models used in circuit simulations (e.g., SPICE), extract the behavior of each designed block after design and layout, and to use the resulting transfer functions or best-fit equations during training. Even with these models, there is still the issue of accounting for unexpected behavior from physical implementations. This work addresses the question of how to account for all noise and variations and produce a set of parameters that generalize well for all possible device outcomes.

Because each circuit is different, achieving high accuracy on the digital model is not an acceptable goal. The resulting parameters would be over-fit to the ideal model and have varying degrees of performance in actual hardware. Some have proposed characterizing fabricated devices and using the measured variations during training [36, 38, 59]. While this may provide some benefit for a single wafer, it is not guaranteed that these measurements will match other wafer runs. Measuring many devices from at least a few runs would yield better results but is time consuming and expensive.

Rather than directly measuring the devices, regularization techniques can help reduce the overfitting problem. Common techniques like L2 normalization, dropout training, and adding noise to the input signals can provide some benefit, but a better understanding of which parameters are overfitting the ideal model would be more effective. In [23, 61], noise is introduced in the weights for each epoch/sample. Similarly, [38] adds noise to the input of each new layer; the desired result is that the network will be trained to increase accuracy even in the presence of variation. Each of these methods of normalization should be compared for their effectiveness in variation resilience.

The purpose of this research is to improve the performance of neural networks that are trained using a digital model and implemented/deployed on analog hardware. Specifically, it is our goal to develop methods for designing and training a neural network such that it will still perform with good accuracy and with some amount of predictability even after the weights and input values have been corrupted by noise (representing inter-device variations and system noise together). Because analog computing is a form of approximate computation, we will also show that the methods we use can also be applied to digital low-precision systems.

2.6 Modeling PVT in Analog Neural Networks

Noise and other random variables in analog hardware take on many forms. Noise can be random — caused by the trapping and releasing of electrons from molecular bonds, energy injections from radiation, the random motion of electrons due to thermal energy, and other unpredictable phenomena. It can also be caused by electromagnetic interference from nearby conduction lines within the same or neighboring circuits. Variations in performance can be caused by temperature changes, aging of the device, and the finite precision of the fabrication process. In analog neural networks, where the multiplicative synaptic weights are set using physical phenomena such as an electric field, the precision of the programming is finite, so it is very possible that the actual programmed weight could be higher or lower than the intended value. In our work, we wanted to simulate the effect of these random noise sources on the performance of a neural neural network. In other words, after applying noise to the feature extraction stage and the synaptic weights of the network, we wanted to see what the distribution of accuracies looked like for a large number of corrupted variants of one ideal trained network.

Studies have shown that the values of passive components (i.e., resistors and capacitors) can vary as high as 25% from their expected values because of process variations [40–42]. The actual variation profile is similar to the gaussian distribution. For passive value

variations, the randomness is a multiplicative noise, but other variations such as charge injections and programming errors are better modeled as additive noise. We decided to use a combination of additive and multiplicative noise for adding noise to the synaptic weights because additive noise has less of an effect on larger values and multiplicative noise has less effect on smaller values.

After training a few of the different neural networks we tested with a few datasets, we found that the maximum magnitude of one of the synaptic weights was typically about 3. Using this information we arbitrarily chose a combination of additive and multiplicative noise profiles such that around the maximum weight value, the percentage of the noise was only slightly dominated by multiplicative noise and the total noise had a standard deviation of about 7%. The paramters for multiplicative gaussian noise are given in (2.5), and the parameters for additive noise are in (2.6).

$$\mu = 1, \sigma = 0.035 \tag{2.5}$$

$$\mu = 0, \sigma = 0.07 \tag{2.6}$$

We represent the variation in devices by a set of random vectors with values taken from the gaussian distributions given above and as described in (Appendix). Each "device" is represented by three random vectors: one for feature noise, one for multiplicative noise for the synaptic weights, and the other for additive noise for the weights. Each vector contains one value for each of the parameters to be corrupted by variations. To create a new "device" for testing or training, the ideal parameter values are multiplied by or added to the associated random vectors, and inference or training is performed as in the ideal case. In this way, we can represent 1000 variations on the same device by 1000 sets of three random vectors, as illustrated in Fig. 2.13.

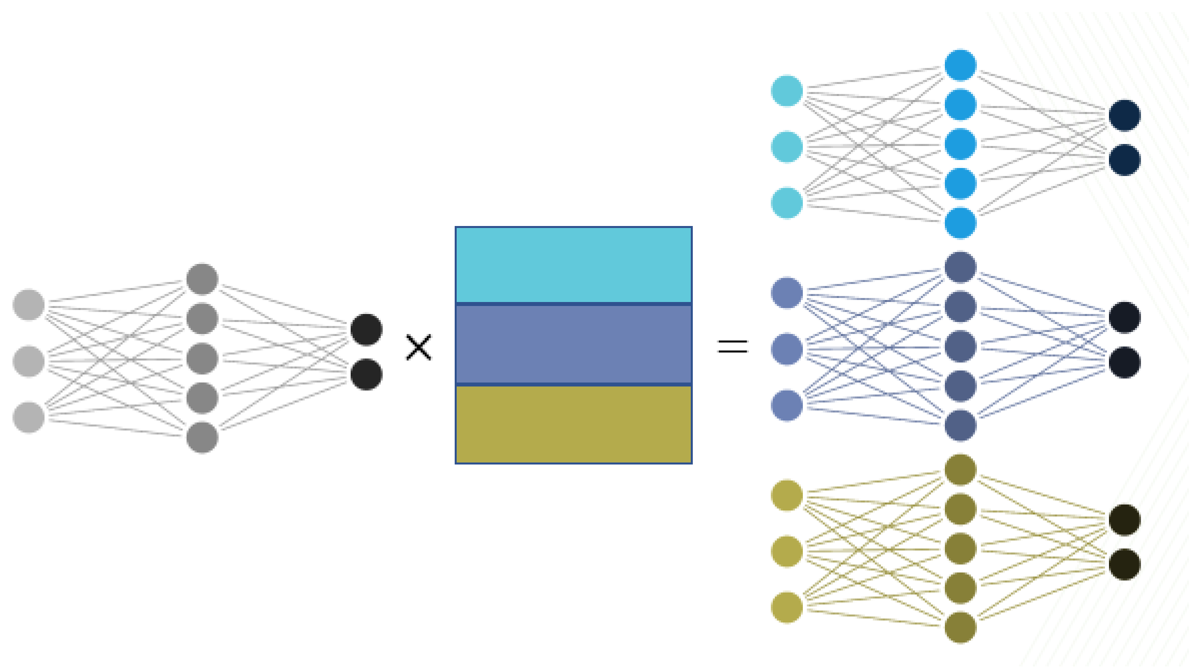


Figure 2.13: Illustration of how the noise vectors transform one network into many variations

CHAPTER 3

GENERALIZING AND FITTING

Generalizing neural networks has been and continues to be a topic of interest in the ML field. ML can achieve exceptional levels of accuracy because of the vast amounts of data that is available thanks to the internet and the ubiquity of sensors. However, a neural network can only be as good as the data it is trained on. As was described earlier, the algorithm for machine learning involves minimizing some given function where the function is calculated using predicted and ground-truth data from a finite dataset. Ideally, we would have an instance for every possible input that the network would see, but this is impractical and impossible for most tasks.

If we have an image classifier that predicts if a photo contains a cat, a dog, or some other animal, we can collect every image on the internet of a four-legged house pet, but there are many other factors that come into play. What if we are given an image where a dog happens to look like a cat or another animal (see Fig. 3.3)? There are also camera quality differences, lighting, and an infinite number of angles from which a camera may capture the animal's image. Since it is impossible to capture all these cases to train a classifier, it is important to make sure that the classifier will perform as well as possible on data that was not used for training.

One important metric for improving a neural network is how well it generalizes to unseen data. Another common way to describe it is how well the network fits the data. As important as it is for a neural network to get a high accuracy score on the training data, if its performance drastically decreases on a separate but similar test dataset, that makes it insufficient to be used in the real world. In this case, we would say the neural network has "overfit" the training data. The opposite extreme of this condition is when the neural network does not perform well on either the training set nor the test data, such as when the neural

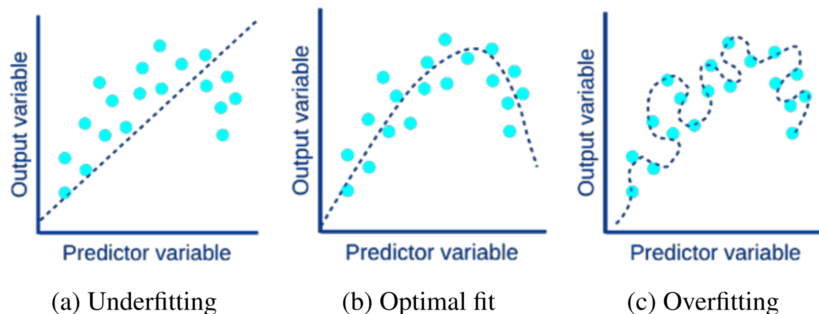
(a) Dog (left) and cat (right).

(b) Dog (left) and bear (right).

Figure 3.1: Examples of edge-case images that may be misclassified.

network is first initialized; this is known as "underfitting" the data.

An example of what this may look like in a regression task is shown in Fig. 3.2, where Fig. 3.2a shows an underfit function for the data and Fig. 3.2c shows an overfit function. The goal for machine learning is to find a function that fits the data well enough to be accurate on the training data while also being able to give an accurate prediction given a new input, as in Fig. 3.2b.



(a) Underfitting (b) Optimal fit (c) Overfitting

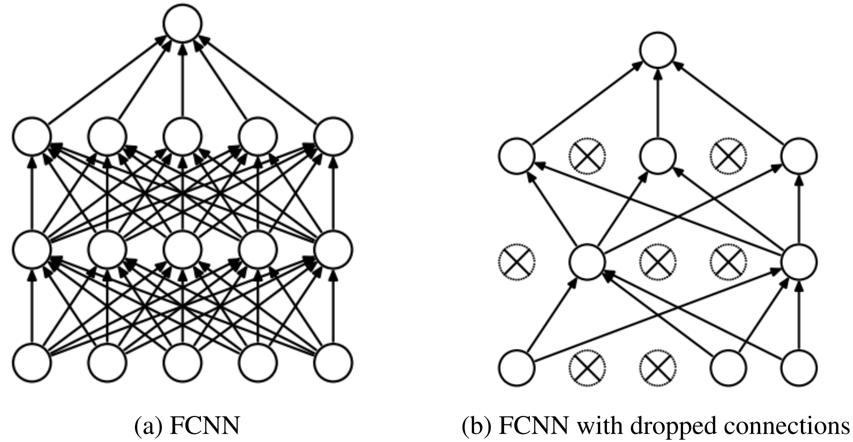
Figure 3.2: Illustrations of the concept of fitting a function to the data.

3.1 Improving Generalization

Some basic preventative steps can be taken to avoid overfitting the training data, such as using a smaller network and increasing the size of the training dataset. The neural network needs to have enough layers and neurons in order to approximate the complex relationships between the input features of the data, but making it too large increases the chances of being to be too accurate on the training set. By increasing the number of samples in the training set, the machine learning algorithm is able to increase performance in more cases. However, sometimes it is not wise to make the network any smaller, and our dataset cannot have infinite samples, so other methods are needed. There have been several proposed techniques for combatting overfitting in machine learning. Some examples include dropout training, data augmentation, early stopping, and L1 or L2 regularization.

3.1.1 Dropout Training

Overfitting is caused by the over-parameterized, redundant function represented by the neural network learning the training data too well. Dropout training, also called "dilution", avoids this by randomly disconnecting portions of the neural network during training, as in Fig. 3.3b. For example, say the neurons in the kth layer of the neural network have a 50% chance of being disconnected (i.e., the output is zeroed) in any given cycle of the training process. In this way, the layer learns to perform its given part of the task with roughly half of its neurons. In other words, the neurons become less dependent on one another and are able to give correct outputs under a greater variety of internal conditions [68]. Once the network is trained, all the neurons are activated for inference, and the neurons are able to work together without being over-trained. Dropout training has been shown to significantly improve accuracy on unseen data samples for a variety of input media [69].

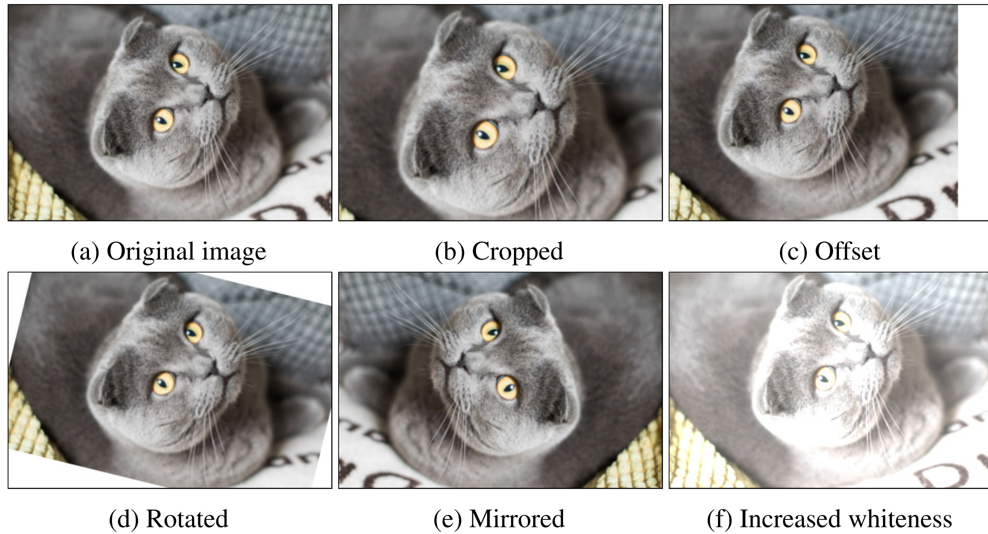


(a) FCNN (b) FCNN with dropped connections

Figure 3.3: An example of how connections may be disabled for dropout training. For each new epoch, a new set of connections are disabled; effectively, a new network is used for each epoch of the training cycle. Image taken from [69].

3.1.2 Data Augmentation

Data augmentation is artificially increasing the size of a dataset by creating "augmented" versions of the training data. For example, for the dog/cat image classifier discussed before, one image of a cat could be made into nine different images easily by adding transformations such as mirroring, rotating, offsetting, cropping, and changing the brightness or coloring (see Fig. 3.4). As stated before, having more examples allows the neural network to learn



(a) Original image (b) Cropped (c) Offset

(d) Rotated (e) Mirrored (f) Increased whiteness

Figure 3.4: Figures demonstrating how a single image can become six different examples in a training set by applying simple alterations.

more possibilities; however, there is still the possibility of overfitting the augmented dataset, so more steps may be needed to prevent this.

3.1.3 Early Stopping

In the early stages of the training of a neural network, the performance of the network on the training set and a validation set is poor (loss is high). After a number of epochs and parameter updates, the performance on both sets should improve. At the point where the network starts overfitting the training data, the loss calculated for the validation set stops improving and may even degrade over time. With early stopping, the loss of both datasets are tracked, and the training is stopped when the validation loss stops improving or starts increasing. The early stop point is illustrated in Fig 3.5. In some cases, a "best" state is saved and reloaded after the stop condition is met. Early stopping is a simple and common way to combat overfitting, but its success is again dependent on what is in each dataset; it is possible to overfit the training set and the validation set as well.

3.1.4 Weight Regularization

Studies have shown that neural networks with larger synaptic weight values are more prone to overfit a dataset. [70] explains this is because larger values cause sharper changes in the learned function. Weight regularization is a way to force the neural network to learn smaller synaptic weights. The most common regularization methods are L1 and L2 regularization.

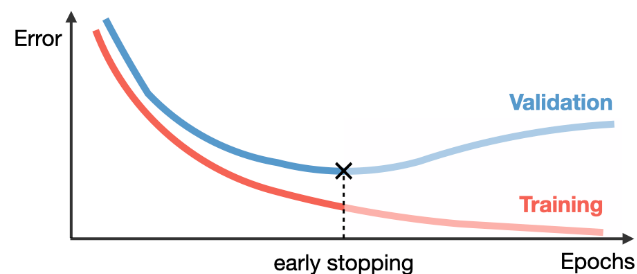


Figure 3.5: Visualization of the early-stop point in relation to the validation and training loss as a function of number of epochs trained.

The basic idea of either method is to add a term to the loss function that is dependent on the magnitude of the weights in the neural network. As the weights grow more positive or negative, the loss increases, so the machine learning algorithm works to balance the loss due to inaccuracy and the loss caused by high-magnitude synaptic weights. L1 and L2 differ in that L1 regularization use the sum of the absolute values of the weights, as in (3.1), and L2 uses the sum of the squares of each weight, as in (3.2). This difference means L1 regularization is more likely to create zero-valued weights. Whether to use one or the other is dependent on the application and is one of the many parameters to explore when designing the ML algorithm.

$$LOSS_{L1} = f_{loss} + \lambda\Sigma|w_{ijk}| \tag{3.1}$$

$$LOSS_{L2} = f_{loss} + \lambda\Sigma|w_{ijk}|^2 \tag{3.2}$$

3.2 Device Fitting

One of the challenges of analog computing is the variability between fabricated devices; no two devices will give the exact same result, and the result cannot be predicted with high precision. Along with that, thermal noise, drift, and interference from nearby signals can cause further uncertainty. Neural networks intended to work on analog systems or other systems with unreliable parameters must be designed and trained in such a way that they will work well on any device (from the intended set of devices) and remain robust to noise and possible variations. This challenge sounds very similar to data overfitting, as discussed in the previous section, but the solutions needed to overcome the device-overfitting problem are quite different.

3.2.1 Understanding the Problem

When a neural network is designed for and deployed in a digital system with high precision, as in most common use cases, we know that the results of the final system should exactly match the performance of the network when it was trained. In binary, a four is guaranteed to be a four; the standards for binary number representations allow for this guarantee. When implementing this same system in an analog neural network, there is no guarantee. The effective final value programmed to the device could be 4.1, and two minutes later it could be 3.9 or any other value within some range. The function actually represented by the programmed analog neural network will not exactly match the one learned during training. Most likely, the final function could be close to the ideal, but even a small change in the function defining the decision boundary could result in drastically different results; in critical applications, this may not be allowable.

3.2.2 Breaking from Traditional Techniques

At first approach to this issue, it seems intuitive that using some of the same techniques used for date fitting should transfer over and at least help with device fitting; our simulations show that this is not the case. In Fig. 3.6, we show the results of applying dropout training and L2 regularization. For a wide range of L2 alpha parameter values and dropout probabilities from 0 to 1, no improvement in either metric is observed. Early stopping can be used to reduce the interquartile range of the resulting accuracy, but only at the sacrifice of accuracy. One form of data augmentation did seem to provide part of a solution, but structured augmentation as in Fig. 3.4 is not necessary.

Unlike the examples given for data augmentation in 3.1.2, rather than adding some systematic mutation to the input, some have tried adding noise to the input values and in other parts of the network. Some previous works have proposed adding noise to the inputs of all layers in the network, rather than just the input of the first layer [38]. Others have added noise to every weight in the network as in [23, 61]. We simulated both cases; the

results are shown in the following section.

3.3 Population Training

In keeping with the idea that adding more noise to more parts of the network could lead to better resilience to variations, we came up with the idea that if we were to train multiple “devices" in parallel, it may be possible to take a more direct path the ideal solution. At the same time, since all devices are trained in parallel, we would have a better guarantee that the performance would carry over to a wider range of variations on the same network. We call this type of training “population training”.

To implement this, we start with an ideal network (no variations added). To represent each of the variations, a set of “device vectors" is generated, where each vector contains one value for each of the parameters to be varied during training, including synaptic weights and any parameters that may vary in the feature extraction stage. To calculate the gradient for one device, all the ideal reference parameters are multiplied with their associated random value in the device vector. The loss and gradient are calculated as usual, and then the

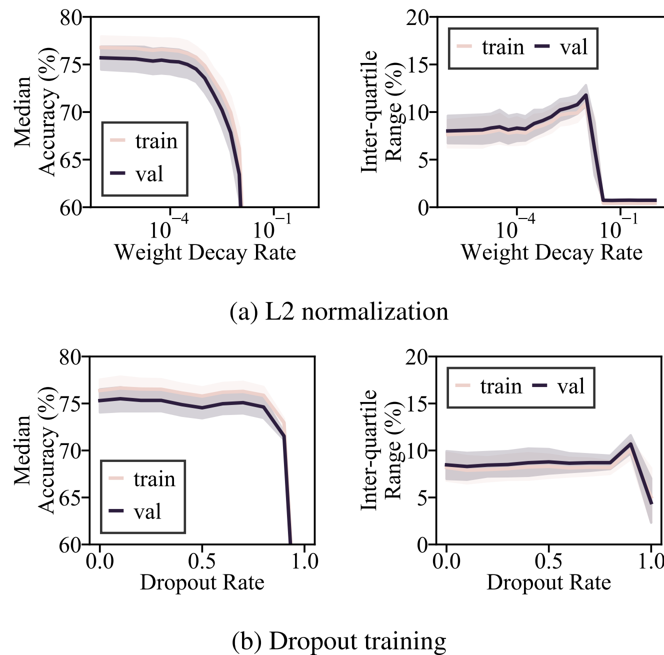


Figure 3.6: Effects of (a) L2 normalization and (b) dropout training on device overfitting.

original reference weights are multiplied with the next device vector to create the next device and calculate its associated loss and gradients. After repeating this process for each of the vectors in the set, the calculated gradients are averaged together, and the reference weights are updated using the calculated average gradients. This process is repeated until the convergence or stop criteria are met.

Population training was tested under three conditions:

1. Population gradients averaged

2. Population gradients summed

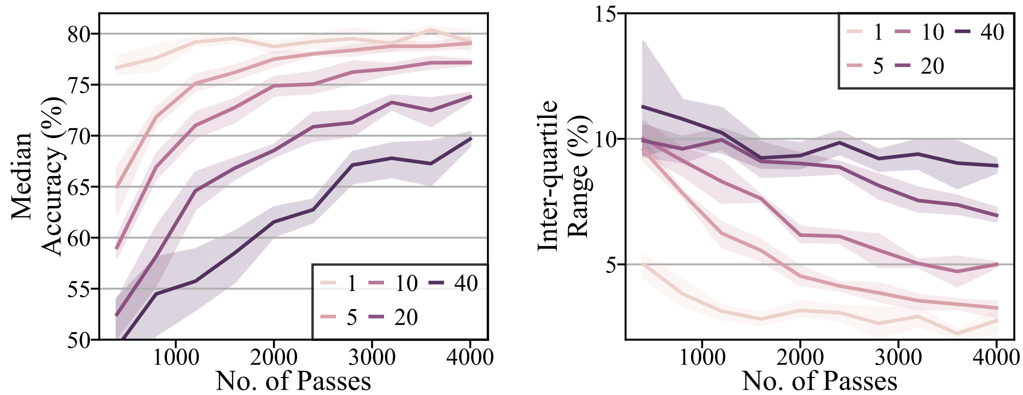
3. Same as 2 but with learning rate increasing with population size

Rather than plotting the performance against the number of epochs, the performance is tracked based on the number of passes on the neural network. In the case of a population of 1 (a different random device for each epoch), one epoch is a single forward and backward pass. For a population of 5, one epoch is five forward and backward passes. Plotting against the number of passes checks to ensure the extra training cost is worth the change in performance. The following simulations were performed using the Microsoft DNS dataset (see appendix A).

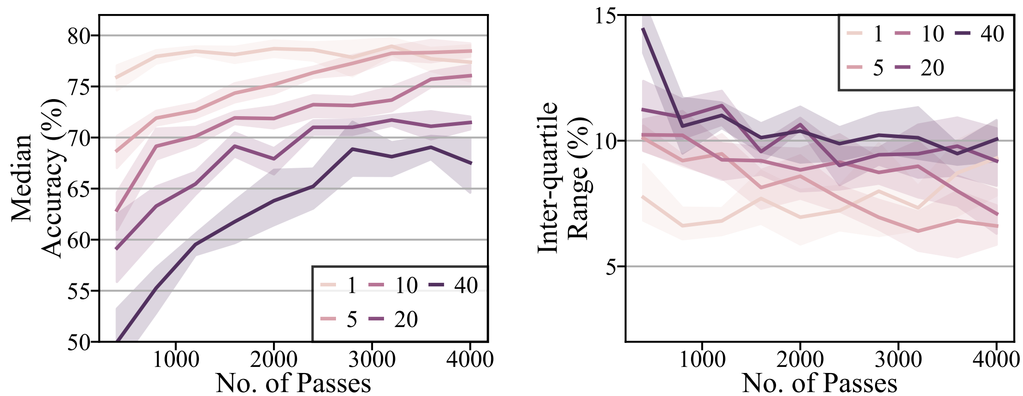
The first case is shown in Fig. 3.7a; training with a population of devices does not seem to offer any benefit over simply adding noise to all weights (i.e., population of 1). Fig. 3.7b also shows a repeated simulation where noise is added only to the input of each layer. These plots show a clear advantage to varying all of the weights and features rather than just adding noise at the input to each layer.

The second simulated case seeks to address the fact that population training only updates the weight every p passes (p = population size). By using the sum of the gradients instead of the average, the calculated gradient for larger populations should be larger (larger step size). In this case (see Fig. 3.8) the larger populations do converge toward the final state faster than before, but they still do not offer any benefit over the $p = 1$ case, even up to

20000 passes. Increasing the learning rate with the population size as in Fig. 3.9 also does not show any significant improvement. The difference in performance using a single device variation per update vs many is very similar to what is observed when comparing stochastic, mini-batch, and batched gradient descent.



(a) Noise added to all weights, features



(b) Noise added to layer inputs, features

Figure 3.7: Performance of population training with gradients averaged. The colors represent different numbers of noised networks that are trained in parallel, with darker lines using more devices in parallel.

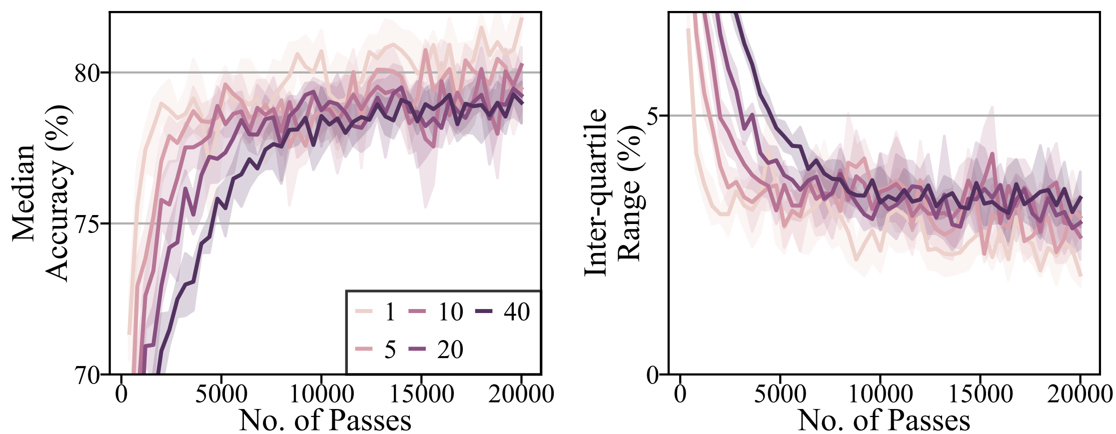


Figure 3.8: Performance of population training with gradients summed.

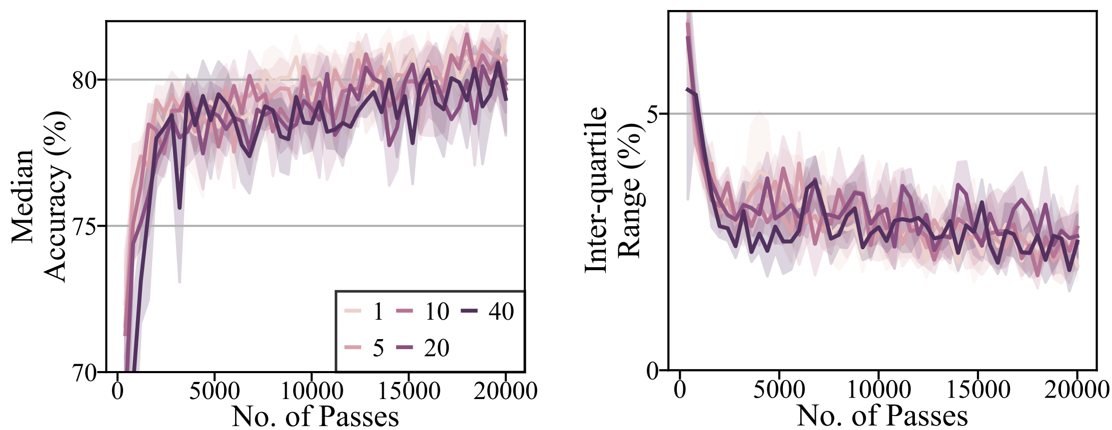


Figure 3.9: Performance of population training with summed gradients and learning rate proportional to $\sqrt{p}$.

CHAPTER 4
BUILDING ROBUSTNESS INTO THE NETWORK

Designing the architecture of a neural network comes with a lot of free variables. Along with deciding which types of layers to use, the activation functions, number of layers, and dimensions of each layer can be changed to get different levels of performance. After starting with an initial guess for an architecture and checking its performance after training, there are some general rules that make it easier to adjust and improve the network.

For an underfit neural network, two simple changes that can be made are to either add more neurons to one or more layers or add more layers. Deciding which works best may require some experimentation, but usually the decision can be made based on how complex the relationship is between the input and the output. Intuitively, one can look at adding neurons as expanding the number of elements in the linear combinations going into the next layer, while adding layers adds more synapses as well as an additional layer of nonlinearity. As stated in the previous chapter, care must be taken to make sure that the larger neural network is not allowed to overfit the data.

Since the depth and number of neurons in a layer can affect how the network fits the data, we wanted to know if changing the same characteristics would have any impact on the device fitting. To test this, we trained and tested a fully-connected neural network with three different datasets: the Microsoft DNS Challenge, Electrical Grid Stability, and Wisconsin Breast Cancer datasets (see appendix A). For each, we swept through a number of hidden layers and various combinations of layer sizes for each layer. After each network completed training, it was tested on 1000 randomized variations of the trained network, and the average and inter-quartile range of the accuracies were saved. We used interquartile range as the statistic for characterizing the spread of the accuracy performance because the distribution of accuracy was skewed about the mean rather than balanced as in the gaussian distribution.

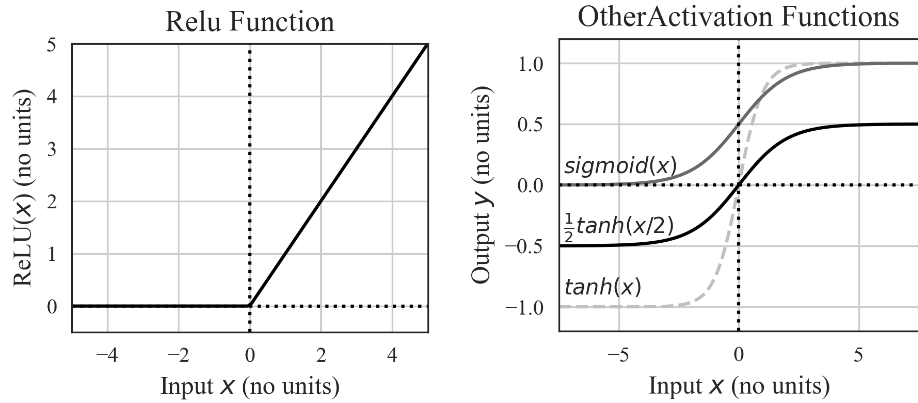
4.1 Selecting a better activation function

Before getting into how the depth and layer size affects the network's resilience against variations, it is important to discuss one other way to reduce the effects of parameter variations. For a fully connected neural network, recall that the output of a neuron is a weighted linear combination of all the inputs that is then passed through some nonlinear function. As an example, assume a fully-connected layer has a ReLU activation function, shown in Fig. 4.1a. If we vary the weights of one of a neuron by a given amount, that amount of change in that synapse output is directly reflected on the output (assuming a positive-valued input to the ReLU). The change directly affects the output value because the slope of the ReLU is 1. However, if we were to reduce the slope, the amount of change that passes through the neuron will also be reduced.

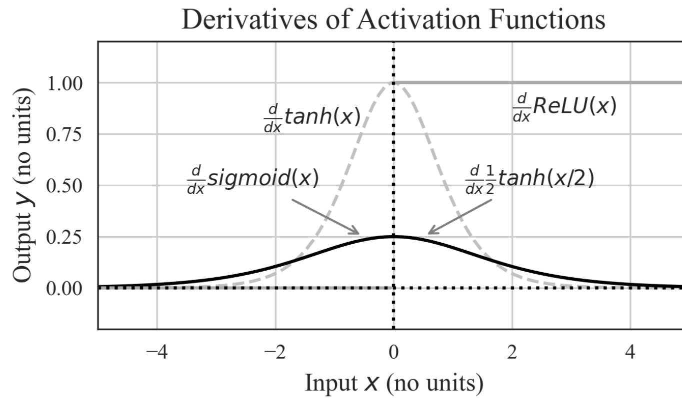
A ReLU function is not preferable for analog implementation. It is difficult to implement a linear half-wave rectifier with analog circuits, especially when compared to other possible activation functions, such as the hyperbolic tangent function (tanh). The peak slope of the tanh function is 1, as shown in Fig. 4.1c, but a few small adjustments can decrease the maximum. The slope of a function can be decreased by simply multiplying it by a value less than one; this compresses the function along the y (i.e., independent variable) axis. The slope can also be reduced by multiplying the x (i.e., independent) variable by a value less than one, which stretches the function along the x axis. For example, we can adjust the tanh function as in (4.1), which is plotted in Fig. 4.1b. Fig. 4.1c shows that the peak slope is reduced to 0.25. Interestingly, this new function has the same derivative as the sigmoid function (see Fig. 4.1b), with the only difference between the two functions being that the sigmoid function has an offset on the y axis of 0.5.

$$0.5tanh(0.5x) = sigmoid(x) - 0.5 \tag{4.1}$$

Using this "compressed tanh" function for both training and inference does significantly



(a) ReLU activation function (b) Other examples of activation functions

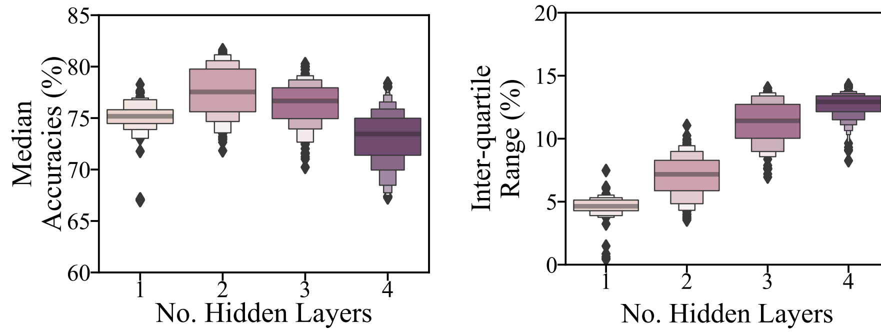


(c) Derivatives of activation functions in (b)

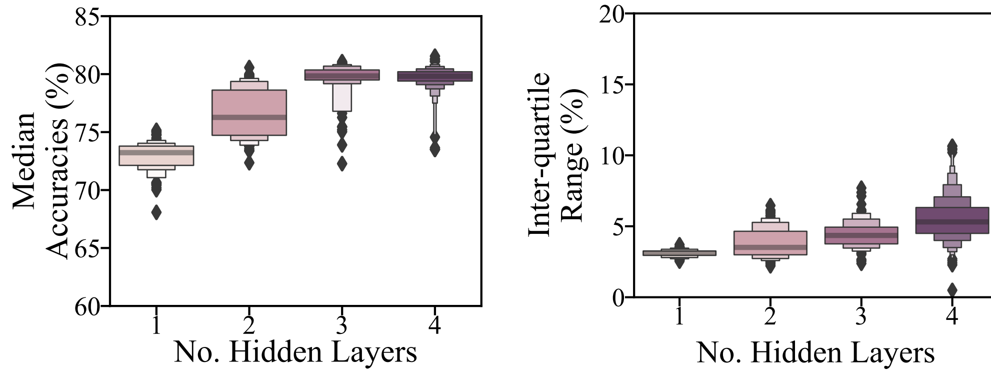
Figure 4.1: Activation functions and some of their derivatives

reduce the spread of the accuracy distribution of a neural network subjected to variations. At the same time, the average accuracy also noticeably increases. Fig. 4.2 shows how this is true for different network depths with varying layer sizes over hundreds of initializations. While this change in the activation function does bring benefits in terms of device fitting, it should also be noted that it is possible that reducing the slope of the activation function too much could lead the network to underfit the data, so just as much care should be taken when trying to balance device fitting with accuracy.

The type of plot in Fig. 4.2 is referred to as the “boxenplot” in the Python visualization library seaborn and is referred to as the “letter-value plot” by its creaters in [71]. It is very similar to a standard boxplot in that the center box shows the median or second quartile



(a) Using tanh activation



(b) Using the "compressed tanh" activation

Figure 4.2: Difference in performance variation when using similar activation functions but with different slopes. The plots in (a) show a variety of network sizes using the hyperbolic tangent activation, and (b) is using a compressed version as shown in (4.1).

(represented by the grey line near the middle) as well as the boundaries between the upper and lower quartiles. The upper boundary of the center box is the third quartile and the lower boundary is the first quartile. Unlike the boxplot, the letter-value plot offers more insight into the spread of the values outside the center box, which only describes the middle 50% of the data. The small dark-grey diamonds above and below each plot represent the outliers of the data as well as the maximum and minimum values.

To underscore the importance of reduced slope in reducing variation, neural networks with the shapes as in Table 4.1 are used with the activation functions as in Table 4.2. The results of the simulations (using the Electrical Grid Stability dataset) are shown in 4.3. One important note is that the ReLU function does seem to do better at suppressing variations than the original hyperbolic tangent function. This is surprising because the slope of ReLU

Table 4.1: Neural Network Layers and Sizes Used In Comparing Activation Functions

Size	Inputs	Hidden Layer Sizes	Outputs
2 Hidden Layers	12	[12,6]	1
3 Hidden Layers	12	[12,12,6]	1
4 Hidden Layers	12	[12,12,12,6]	1

Table 4.2: Activation Functions Compared and Their Associated Equations

Activation	Equation
ReLU4	$4 \times ReLU(x)$
ReLU2	$2 \times ReLU(x)$
ReLU1	$ReLU(x)$
ReLU0.5	$0.5 \times ReLU(x)$
ReLU0.25	$0.25 \times ReLU(x)$
Tanh	$tanh(x)$
CompTanh	$0.5 \times tanh(0.5x)$
CompTanh2	$0.25 \times tanh(0.25x)$

above $x = 0$ is one whereas only the peak slope of the hyperbolic tangent is one. However, when we consider the fact that half of the ReLU function is zero and therefore has a slope equal to zero, this makes more sense. This relationship is still true when comparing the ReLU divided by four and the compressed tanh function, where the peak slop of the compressed tanh equals the above-zero slope of the ReLU divided by four. However, since hyperbolic tangent shapes are simple to implement in analog, we will use it instead of the ReLU function.

4.2 Effects of the Network Shape

4.2.1 Effect of depth

The plots shown in Fig. 4.2 above gives our first look into how the depth of the network affects the accuracy distribution for a network under variations. For the original network

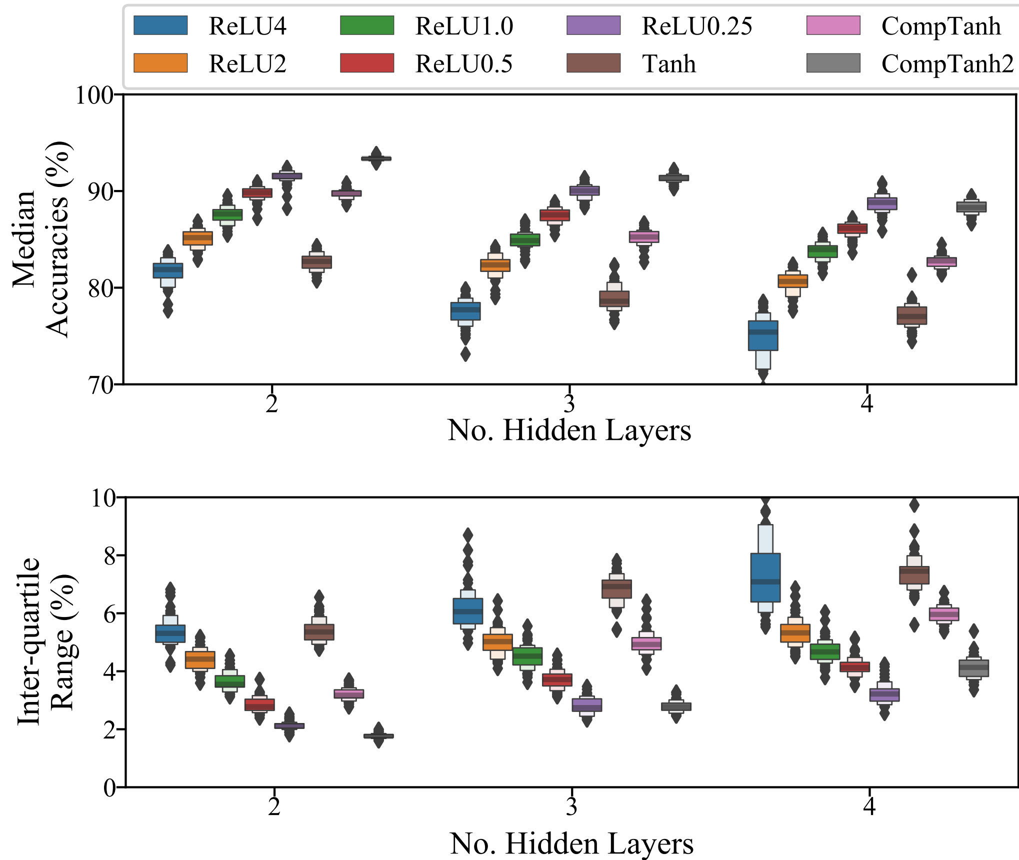


Figure 4.3: Comparison of how different activation functions affect the performance variation when the weights and features are subjected to noise.

with the uncompressed tanh function (Fig. 4.2a, we see that, as expected, increasing the number of layers increases the ability to fit the complex relationships in the data and give better accuracy. However, adding more layers also increases the inter-quartile range of the accuracy distribution. Eventually, the spread of accuracies grows to the point where the average accuracy starts to decrease because the spread will increase toward lesser values rather than equally toward higher and lower values. Replacing the activation with the compressed tanh (i.e., the zero-centered sigmoid) function reduces the spread and allows for more layers to better fit the data while also keeping the interquartile range limited (Fig. 4.2b. We did not simulate with more than four hidden layers, so the plots do not show when the spread gets so large that the average accuracy starts to decrease.

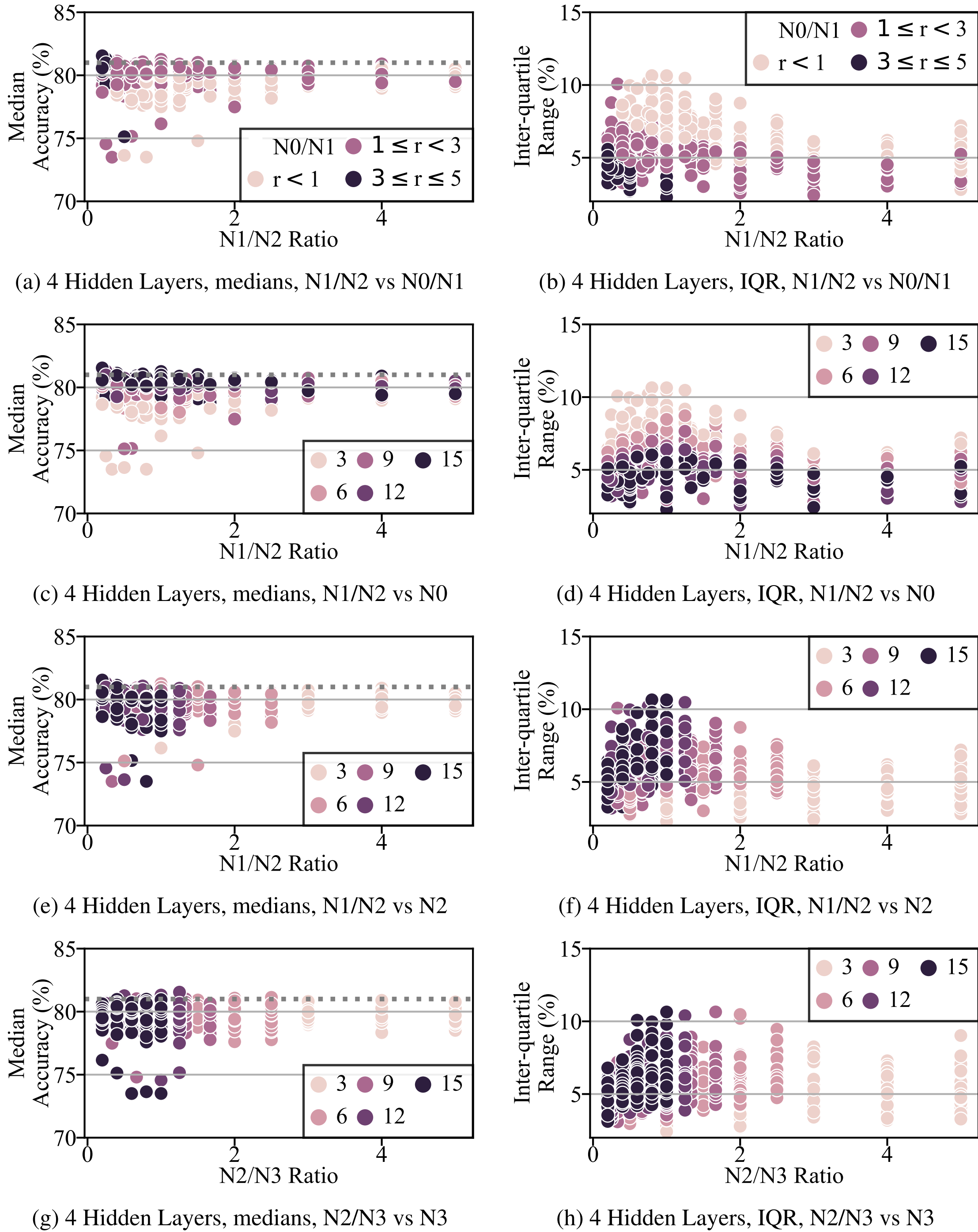
The fact that parameter noise has a greater impact with more layers is not at all surprising.

Noise in the parameters of the feature extraction stage are present in every neuron of the first hidden layer because the output of that neuron is the linear combination of all of the inputs to the layer (i.e., the features) where the weights of the combination also have noise. The neurons in the next layer are a noised linear combination of all the corrupted outputs from the previous layer, and this noise continues to be compounded with more and more layers. In a similar way, multi-stage analog amplifiers take special care to ensure that the earlier amplification stages suppress noise as much as possible because the noise at the input to the first layer is amplified across all of the subsequent gain stages. Based on the observations in these experiments, it is reasonable to assume that suppressing parameter variations in the feature extraction and shallower hidden layers will have a greater impact on reducing the performance variation. They also give further justification for using activation functions that have a gain/slope less than one.

4.2.2 Effect of layer sizes

The next two pages contain plots that show the accuracy of a number of neural networks with between one and three hidden layers and either 3, 6, 9, 12, or 15 nodes in each layer. This experiment was repeated for three different datasets, including the Microsoft DNS Challenge, Electrical Grid Stability dataset and Wisconsin Breast Cancer datasets. The goal of these simulations and plots is to discover how the size relationships between layers affects the performance of the network with variations or if there is a layer that has more impact than the others.

Fig. 4.4 shows plots for neural networks with four hidden layers. To give an example for how to interpret these plots, consider 4.4(a)-(b). This is a plot of the accuracies for four-hidden-layer fully-connected neural networks where the x axis is the ratio of the number of neurons in the second hidden layer (N1) to the number in the second hidden layer (N2) and the colors are a discrete representation of the ratio of the number of neurons in the first hidden layer (N0) to N1. A very clear trend in these plots is that the median accuracy tends



(a) 4 Hidden Layers, medians, N1/N2 vs N0/N1

(b) 4 Hidden Layers, IQR, N1/N2 vs N0/N1

(c) 4 Hidden Layers, medians, N1/N2 vs N0

(d) 4 Hidden Layers, IQR, N1/N2 vs N0

(e) 4 Hidden Layers, medians, N1/N2 vs N2

(f) 4 Hidden Layers, IQR, N1/N2 vs N2

(g) 4 Hidden Layers, medians, N2/N3 vs N3

(h) 4 Hidden Layers, IQR, N2/N3 vs N3

Figure 4.4: Plots demonstrating the relationship of various network layer sizes affects the median and IQR of the network accuracies when subjected to variations. The networks represented here have four hidden layers. Note that 'r' stands for "ratio" and represents the ratio indicated in the legend. For example, in (b), r is equal to the ratio of the number of neurons in the first hidden layer to the number of neurons in the second hidden layer. The dotted horizontal line on the median accuracy plots represent the best training accuracy for a trained neural network with the same number of hidden layers and layer sizes, tested without adding noise to the parameters.

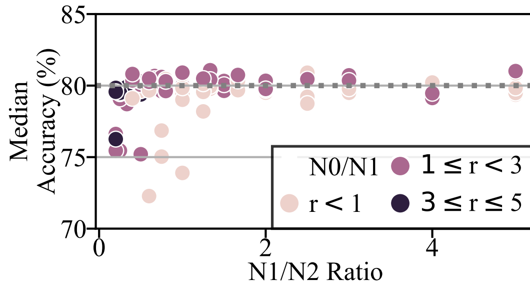
to be higher for networks with more neurons in the first layer than in the second layer and with more neurons in the second layer than in the third layer. The interquartile range of the data tends to be lower under these conditions as well. In other words, these plots suggest that we can increase resilience to variations when this condition is met:

$$N0 > N1 > N2 \tag{4.2}$$

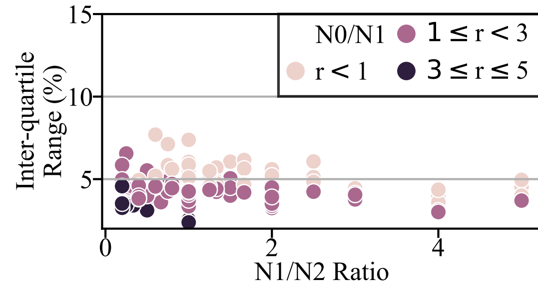
However, this is not a perfect indication since it seems that some networks that follow this condition still end up with lower median accuracy and higher interquartile ranges. Looking at Fig. 4.4(c)-(d) gives more insight, where we can see that networks with a higher number of neurons in the first hidden layer perform best (i.e., at least 12 neurons). Fig. 4.4(e)-(f) shows that there is a weak inverse relationship between N2 and variation resistance, meaning less neurons in the third hidden layer leads to higher median accuracy and lower interquartile range. Finally, Fig. 4.4(g)-(h) show that $N2 > N3$ and a lower number of neurons in the last hidden layer (N3) tends to result in better performance, though the correlation is weak.

4.2.3 Conclusion: best parameters for variation resilience

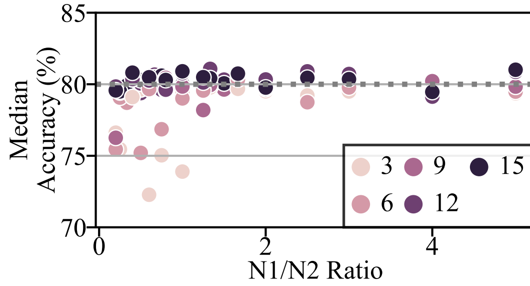
The results shown in Fig. 4.5 and Fig. 4.6 agree with the conclusions gathered from the simulations with four hidden layers. Deeper layers should have fewer neurons than shallower layers, and, as is the case for data fitting, the network should be kept to the fewest layers needed and with as many neurons in the first layer as needed to ensure proper fitting to the data. Figs. 4.8 – 4.3 demonstrate that the same patterns can be observed when using the same networks with the Electrical Grid Stability data set, and Figs. 4.11 – 4.13 show that the results for the Wisconsin Breast Cancer dataset also agree. Our proposed explanation for why layer sizes should decrease toward deeper layers is very intuitive. The noise in earlier layers is added together in a linear combination, as is the nature of a fully-connected neural network. The number of times that the noise is replicated is equal to the number of neurons



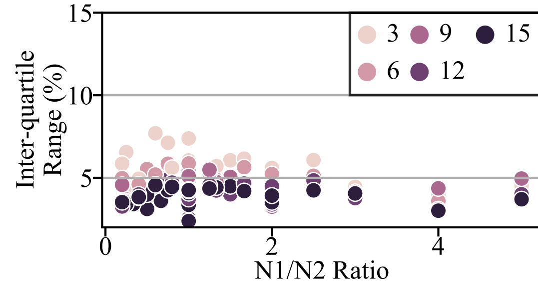
(a) 3 Hidden Layers, medians, N1/N2 vs N0/N1



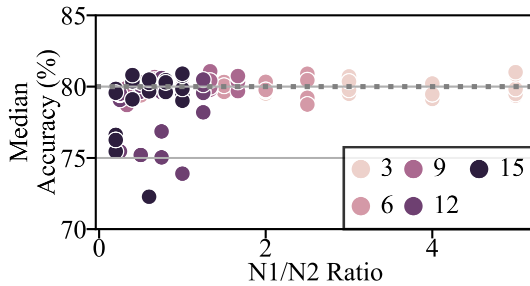
(b) 3 Hidden Layers, IQR, N1/N2 vs N0/N1



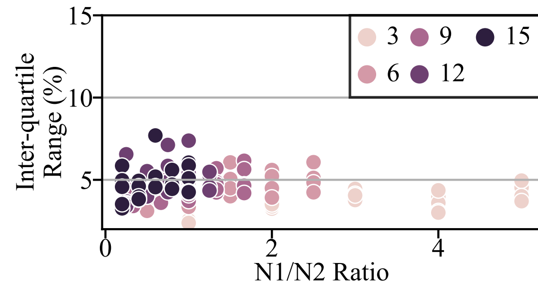
(c) 3 Hidden Layers, medians, N1/N2 vs N0



(d) 3 Hidden Layers, IQR, N1/N2 vs N0

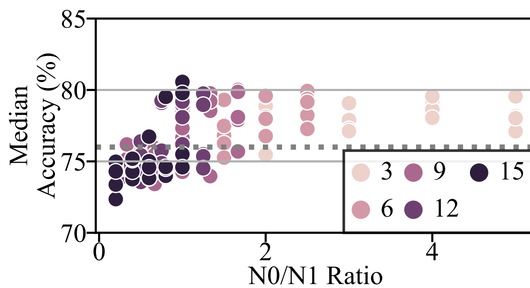


(e) 3 Hidden Layers, medians, N1/N2 vs N2

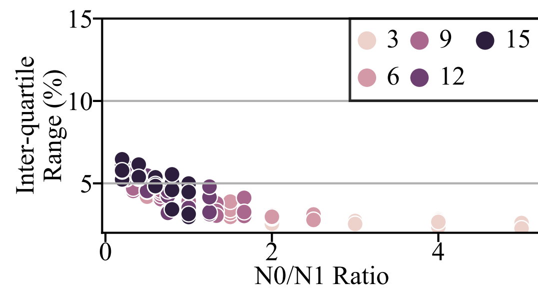


(f) 3 Hidden Layers, IQR, N1/N2 vs N2

Figure 4.5: Same simulations as in Fig. 4.4, but the networks represented here have three hidden layers. See Fig. 4.4 for additional explanation.



(a) 2 Hidden Layers, medians, N0 vs N1



(b) 2 Hidden Layers, IQR, N0 vs N1

Figure 4.6: Same simulations as in Fig. 4.4, but the networks represented here have two hidden layers. See Fig. 4.4 for additional explanation.

in the next layer.

Assume a one-hidden-layer neural network with one input, three neurons in the hidden layer, and one output. Given a synaptic weight matrix $\widetilde{A}$ with all weights equal to one and a noise distribution with $\sigma = \sigma_A$, the output is equal to

$$\widetilde{y} = \widetilde{a_0}x_0 + \widetilde{a_1}x_1 + \widetilde{a_2}x_2 \tag{4.3}$$

Since the noise distribution for each weight is about the same, if all inputs are roughly the same magnitude, the noise distribution of $\widetilde{y}$ would have $\sigma = 3\sigma_A$. If the number of neurons in the hidden layer was expanded to ten, then the distribution for $\widetilde{y}$ would then have $10\sigma_A$. This idea of limiting noise replication is corroborated by the fact that the more extreme layer size ratios (i.e., 3 or more) are consistently better at limiting the variations in the output. To make sure that the variations in the weights are replicated as little as possible, all layers should decrease in size toward deeper layers, and higher-ratio reductions are better.

4.3 Sparse network connections

One of the limitations for ultra-low-power analog neural networks discussed in chapter 2 is low input dynamic range, meaning the input is restricted to a small range of values to prevent unexpected behavior from the network. For ultra-low-power transistors operating in the subthreshold region, only a small amount of current can go through the channel of the MOSFET, so the number of inputs to each neuron must be kept low. We followed the advice of a colleague familiar with subthreshold neural networks that a reasonable number would be about three synapses per neuron (not including the bias input). For a fully connected neural network, that would mean that each layer could have a maximum of three neurons, and only three features could be used as inputs. Such a small network with so few inputs would be unlikely to perform with reasonable accuracy in most cases, even in an ideal device without variations. Instead of limiting the number of neurons and inputs, we can instead

disable some of the connections such that each neuron has no more than three synaptic connections. This is also known as adding sparsity to the network.

Before getting into any details about how sparse networks are generated, we will finish this chapter by showing that sparsity in the neural network reduces the effects of the weight variation by following the conclusion found in the last subsection. By reducing the number of inputs to a neuron, we reduce the amount of possible noise that goes into the neuron, which in turn reduces the amount of noise that is propagated through to the output. Fig. 4.7 shows the same simulations that were performed in the previous section, but this time a sparse architecture with only three inputs per neuron was used for each possible network shape; the only difference is the addition of the five-hidden-layer neural network simulations. In all cases, the interquartile range of accuracies is significantly reduced compared to the simulations shown in 4.2, though the median accuracies are slightly reduced at the same time.

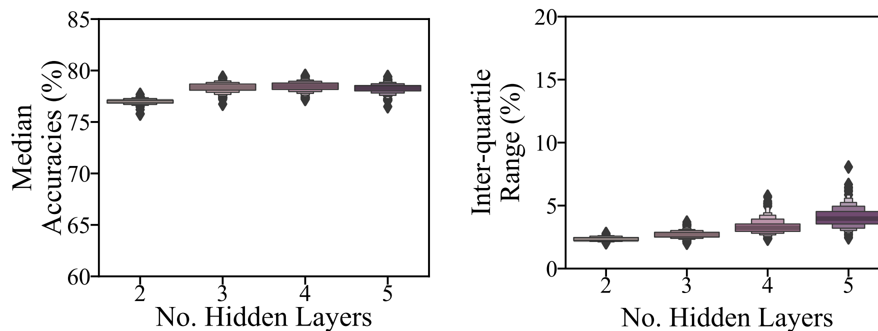
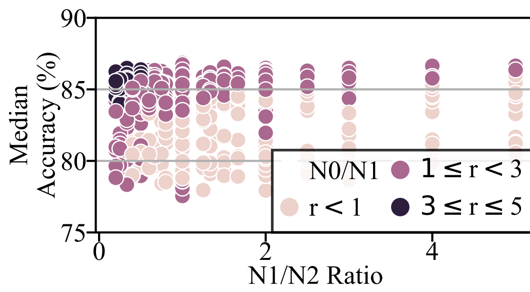
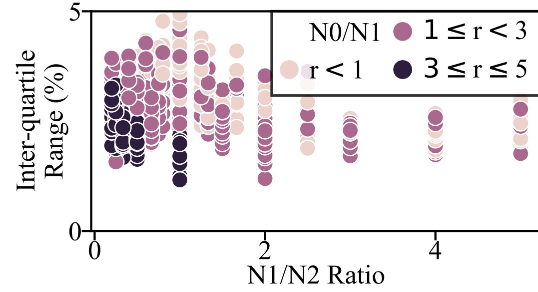


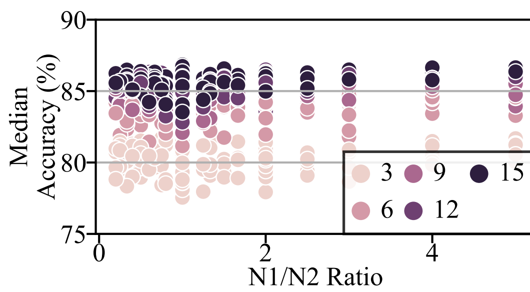
Figure 4.7: Demonstration of how introducing sparsity to a neural network serves to reduce the effects of variations in the neural network parameters when trained and tested under the same conditions as those in Fig. 4.2b.



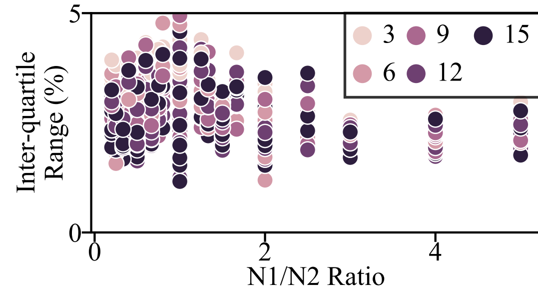
(a) 4 Hidden Layers, medians, N1/N2 vs N0/N1



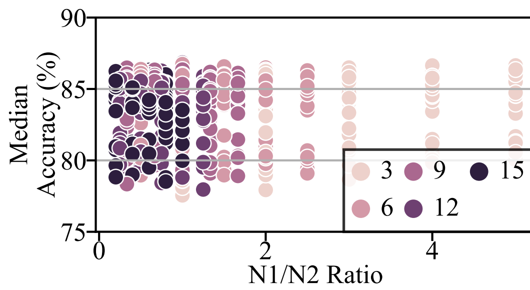
(b) 4 Hidden Layers, IQR, N1/N2 vs N0/N1



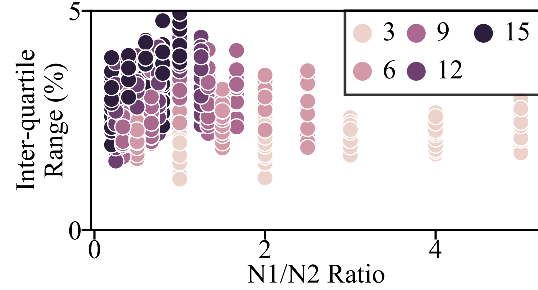
(c) 4 Hidden Layers, medians, N1/N2 vs N0



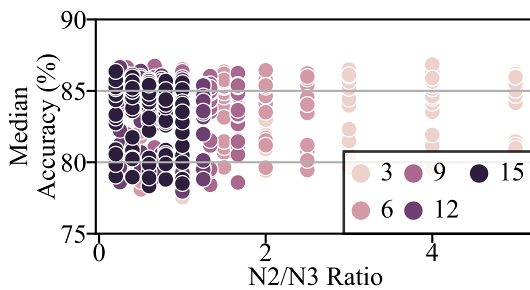
(d) 4 Hidden Layers, IQR, N1/N2 vs N0



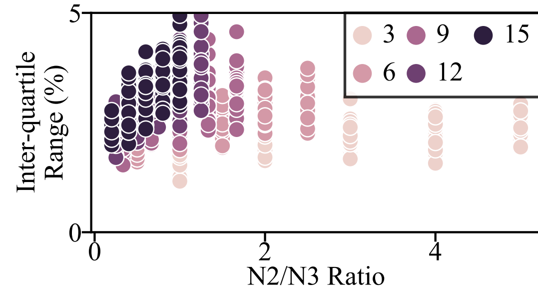
(e) 4 Hidden Layers, medians, N1/N2 vs N2



(f) 4 Hidden Layers, IQR, N1/N2 vs N2

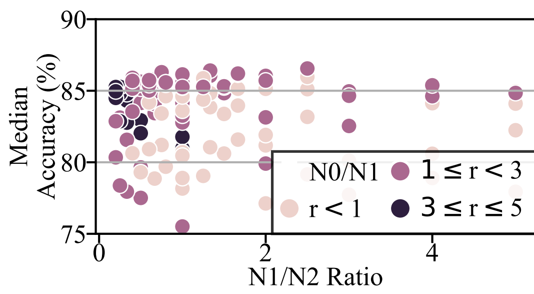


(g) 4 Hidden Layers, medians, N2/N3 vs N3

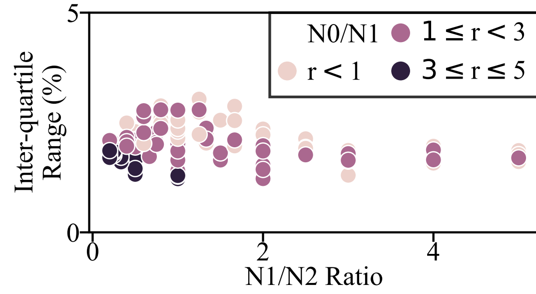


(h) 4 Hidden Layers, IQR, N2/N3 vs N3

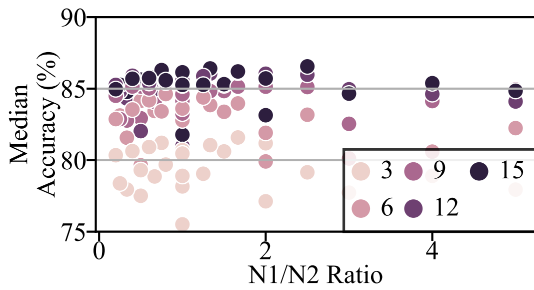
Figure 4.8: Same simulations as in Fig. 4.4; the networks represented here have four hidden layers, and the Electrical Grid Stability dataset is used. See Fig. 4.4 for additional explanation.



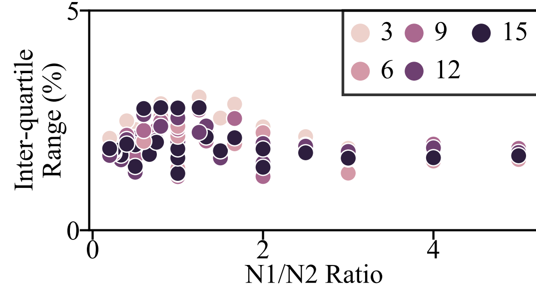
(a) 3 Hidden Layers, medians, N1/N2 vs N0/N1



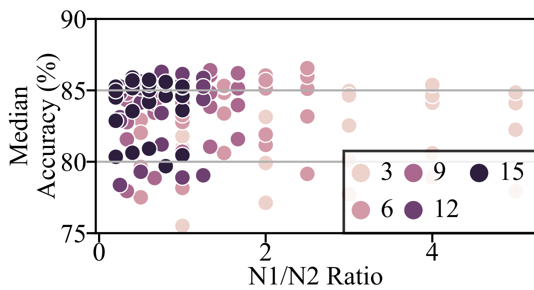
(b) 3 Hidden Layers, IQR, N1/N2 vs N0/N1



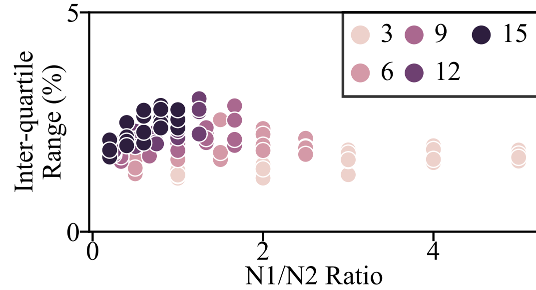
(c) 3 Hidden Layers, medians, N1/N2 vs N0



(d) 3 Hidden Layers, IQR, N1/N2 vs N0

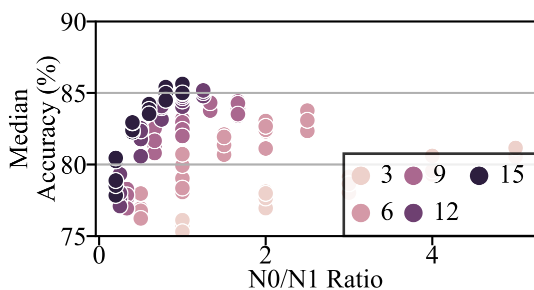


(e) 3 Hidden Layers, medians, N1/N2 vs N2

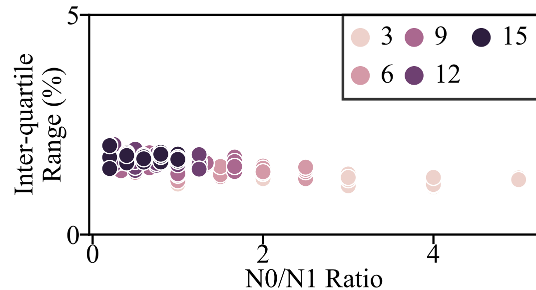


(f) 3 Hidden Layers, IQR, N1/N2 vs N2

Figure 4.9: Same simulations as in Fig. 4.4; the networks represented here have three hidden layers, and the Electrical Grid Stability dataset is used. See Fig. 4.4 for additional explanation.



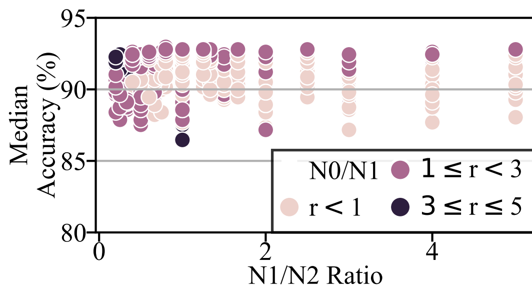
(a) 2 Hidden Layers, medians, N0 vs N1



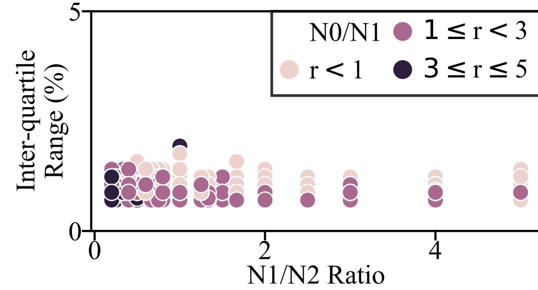
(b) 2 Hidden Layers, IQR, N0 vs N1

Figure 4.10: S

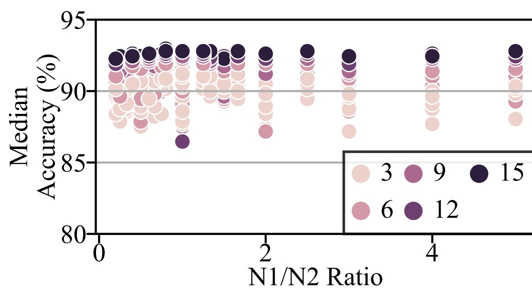
ame simulations as in Fig. 4.4; the networks represented here have two hidden layers, and the Electrical Grid Stability dataset is used. See Fig. 4.4 for additional explanation.



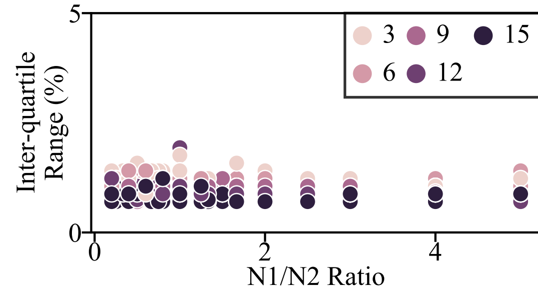
(a) 4 Hidden Layers, medians, N1/N2 vs N0/N1



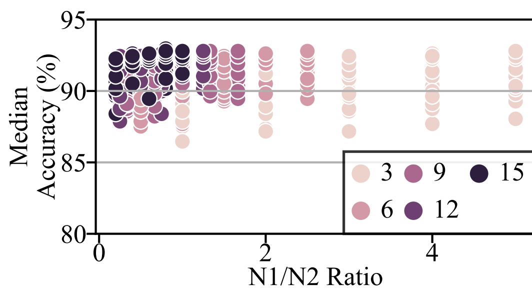
(b) 4 Hidden Layers, IQR, N1/N2 vs N0/N1



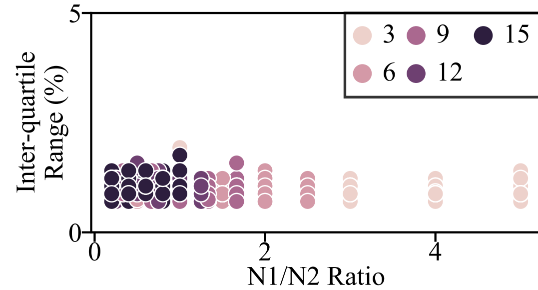
(c) 4 Hidden Layers, medians, N1/N2 vs N0



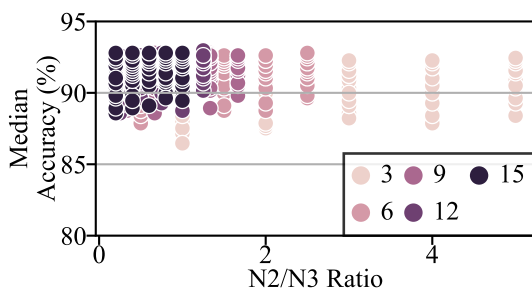
(d) 4 Hidden Layers, IQR, N1/N2 vs N0



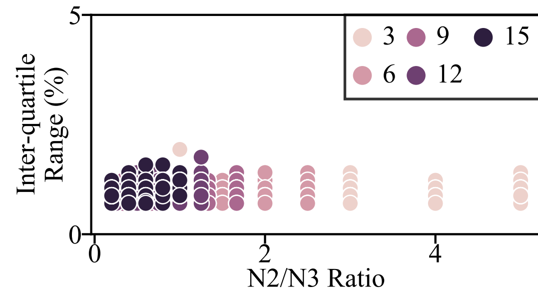
(e) 4 Hidden Layers, medians, N1/N2 vs N2



(f) 4 Hidden Layers, IQR, N1/N2 vs N2

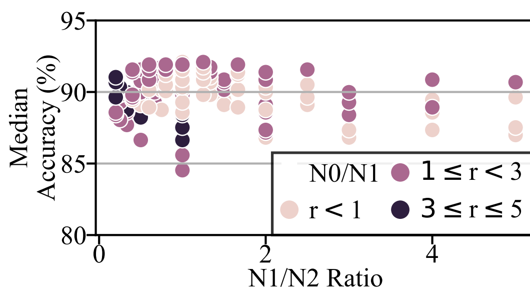


(g) 4 Hidden Layers, medians, N2/N3 vs N3

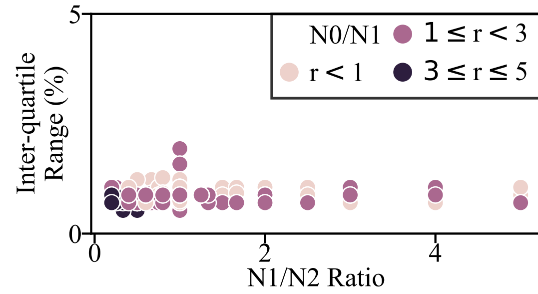


(h) 4 Hidden Layers, IQR, N2/N3 vs N3

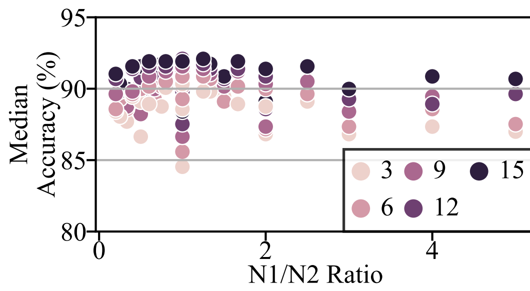
Figure 4.11: Same simulations as in Fig. 4.4; the networkss represented here have four hidden layers, and the Wisconsin Breast Cancer dataset is used. See Fig. 4.4 for additional explanation.



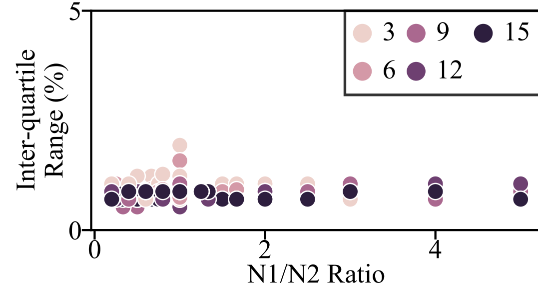
(a) 3 Hidden Layers, medians, N1/N2 vs N0/N1



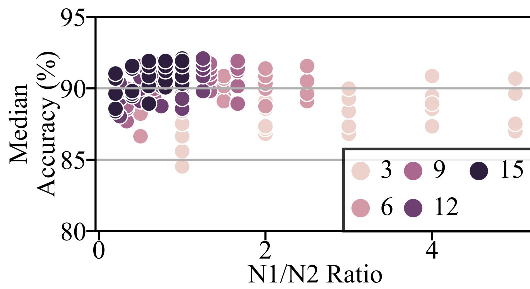
(b) 3 Hidden Layers, IQR, N1/N2 vs N0/N1



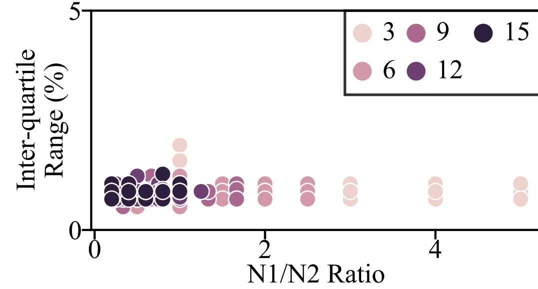
(c) 3 Hidden Layers, medians, N1/N2 vs N0



(d) 3 Hidden Layers, IQR, N1/N2 vs N0

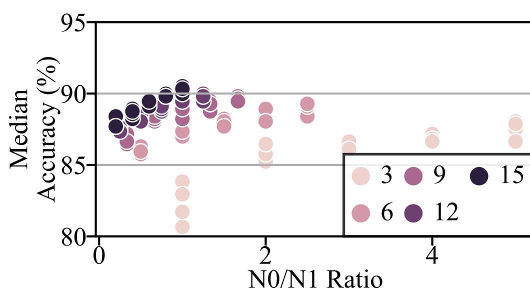


(e) 3 Hidden Layers, medians, N1/N2 vs N2

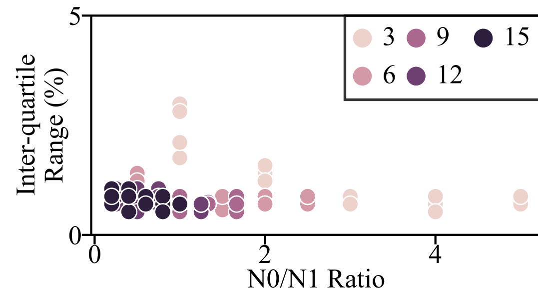


(f) 3 Hidden Layers, IQR, N1/N2 vs N2

Figure 4.12: Same simulations as in Fig. 4.4; the networks represented here have three hidden layers, and the Wisconsin Breast Cancer dataset is used. See Fig. 4.4 for additional explanation.



(a) 2 Hidden Layers, medians, N0 vs N1



(b) 2 Hidden Layers, IQR, N0 vs N1

Figure 4.13: Same simulations as in Fig. 4.4; the networks represented here have two hidden layers, and the Wisconsin Breast Cancer dataset is used. See Fig. 4.4 for additional explanation.

CHAPTER 5

GENERATING THE BEST SPARSE NETWORKS

Sparse artificial neural networks were inspired by the way the human brain develops. On average, the human brain grows neurons until about 18 months of age; after this, the brain gradually prunes the unneeded connections over the course of several years [72]. This demonstrates that the brain recognizes that some of its synaptic connections are not needed and that some neurons can perform their jobs just as well with fewer inputs. Research has shown that sparse connectivity can work just as well with artificial neural networks.

Recall that one of biggest problems with digital neural networks is the energy needed to train the network and use it for inference. With fewer connections or an overall less complex architecture, the amount of energy needed can be reduced along with the computational complexity. Of course, part of the benefit of machine learning is that the neural network can be trained to recognize the unknown patterns and relationships within a dataset; people generally do not have enough knowledge about a dataset to know which synapses are safe to disable. Somehow, the training algorithm needs to be able to learn the synaptic weights as well as which neurons or individual synapses are critical to the accuracy of the network. There is a lot of literature about the best way to generate sparse neural networks, with each researcher coming to differing conclusions about which algorithms work best. In general, these methods can be categorized as genetic programming, one-time pruning, and gradual pruning.

5.1 Genetic Programming

Simply put, genetic programming is a method of generating a high-performance piece of code or some other simulated structure or device using the theories surrounding genetic evolution. In the beginning, a random "population" of initial guesses are generated within a

range of given criteria. Each member of the population is scored based on its performance at a given task (e.g., accuracy of a neural network, power usage by a circuit, lines of code). A better score increases the probability that it will be kept from the population and used to generate a new set of "offspring" that will become part of the next generation of the population. This idea is taken from the theory of "survival of the fittest", which is why this comparison step is often referred to as the "fitness test".

The methods for generating the new members in the next generation also borrow from the theory of evolution. Similar to how the characteristics of children are taken from a random mixture of the genetic properties of its parents, the next generation is created by taking portions of two or more "parents" in the current generation and mixing random subsections and/or parameters. This is referred to as "crossover". Next-generation offspring can also be created by randomly mutating a parent or one of the offspring already generated(e.g., adding a line of code or removing a layer from a neural network). This is similar to genetic mutation, where a new characteristic appears that has no link to the parents. Both crossover and mutation are illustrated in Fig. 5.1 The actual implementations of selection, crossover, and mutation are beyond the scope of this paper.

Genetic programming has been used successfully in a number of fields; it has been

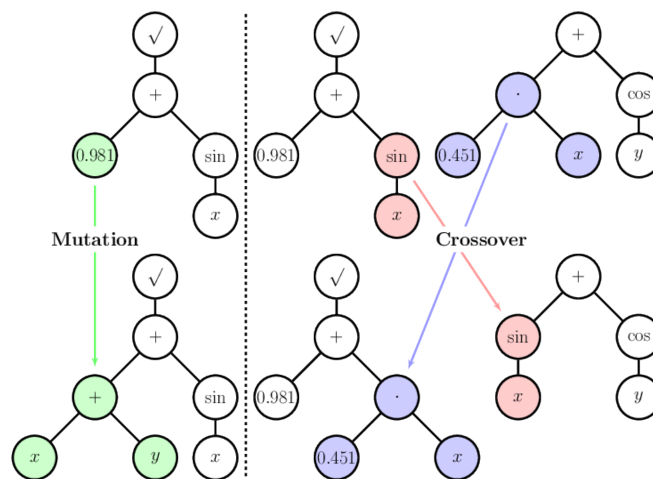


Figure 5.1: Illustration of mutation and crossover in genetic programming (from [73]). In this example, the algorithm is trying to generate a mathematic function using basic building blocks such as addition, numeric values, trigonometric functions, and variables.

used to create ideas that match, compete with, or outperform the state of the art, including patented materials. However, sometimes genetic programming may overcomplicate the problem at hand, especially with a larger population size. As an example, if we started with a population of twenty sparse neural networks (a small/moderate population size), trained the weights, and then tested each to compare their accuracy, this would require the time needed to train all twenty of the randomly-generated neural networks. However, we found that by using common pruning methodology, we had a high probability of generating a high-performance sparse neural network in ten or fewer full training cycles. At the same time, creating the next generation often gave very different results because the initialization of the neural network has a significant impact of the final state of the weights, including which connections are important. This unnecessary complication of the problem led us to focus entirely on pruning-based methods for generating sparse neural networks.

5.2 Synaptic Pruning

The term "pruning" is just what it sounds like: removing weaker parts of a trained neural network while keeping other sections that are more crucial to the performance of the network. The pruning can remove entire neurons or channels or only a few synaptic connections. Some of the hyperparameters for pruning that can be changed include how often to prune the network during training, how much of the network to prune at one time, and how to numerically score the usefulness of a synapse, neuron, or channel. Some concluded that it is best to prune the unneeded connections all at one time and then retrain the remaining weights. Others have demonstrated that better performance can be achieved by gradually pruning the network. As for scoring the connections to determine importance, the most common method is to rate them based on magnitude and eliminating the lowest-magnitude weights. Neurons can also be scored based on the magnitude of the activation function such that neurons that consistently get near-zero values out of the activation can be removed, or neurons can be pruned if their synaptic weights are similar to others in the same part of the

Table 5.1: Neural Network Layers and Sizes Used In Sparsity Analysis

Size	Inputs	Hidden Layer Sizes	Outputs
3 Hidden Layers	12	[12,6,3]	1
4 Hidden Layers	12	[12,12,6,3]	1

network (e.g., layer). These comparisons can be performed locally (e.g., within the neuron or layer) or across the entire network.

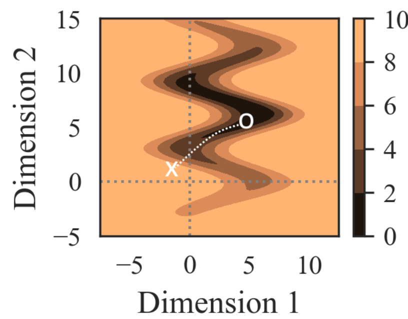
In this work, we will rely on magnitude-based scoring because our main goal is to create sparsity in each neuron rather than to increase sparsity in the network as a whole. We will describe and compare the ability of each pruning method to reliably generate high-accuracy sparse neural networks, at least in the case of fully-connected neural networks. This analysis will be done in terms of one-time pruning and gradual pruning using a variety of algorithms with the underlying data taken from the Electrical Grid Stability dataset. For the first set of results, we started with network shapes that followed the guidelines we found from our analysis in the previous chapter. Specifically, the neural network layer sizes are as given in Table 5.1.

5.2.1 One-time pruning

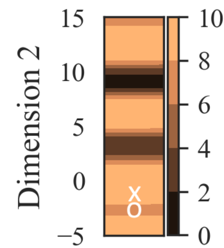
For one-time pruning, the nerual network is trained to a point (e.g., a number of epochs), and then then all the connections deemed as unneeded are removed from the network. Along with this, there has been discussion as to whether the remaining weights should be randomly reinitialized, returned to their original pre-trained state, or kept as they were before continuing the training process. Our results showed that the resulting sparse architecture is dependent on the first initialization. If the weights are randomly reinitialized, the new starting weights may not be able to reach the same performance with the new architecture since the dimensionality of the search space has been reduced. For the same reason, returning to the original weight values may not allow the network to find the same best state. An example for both of these cases in two dimensions is shown in 5.2a-5.2c. For this reason, it

is better to keep the same weights from when the network was pruned and continue training from there rather than retraining from scratch.

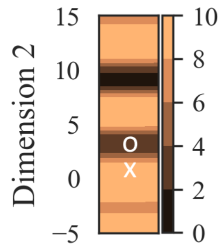
Removing any amount of dimensionality in the weight search space during training inhibits the ability of the machine learning algorithm to train the weights to the best performance possible. If the network is overparameterized from the start, then the effects of reduced dimensionality may not be as apparent. In this case, where some dimensions are permanently removed after the network has been partially or fully trained, it may become impossible for the network to return to its peak performance. 5.2d shows an example in two dimensions where the path to better performance is blocked by a higher-loss wall after the network has been pruned.



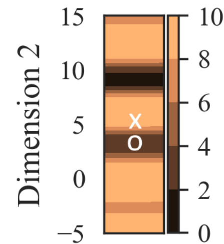
(a) 2D Training



(b) 1D Training after reducing dimensionality of search space (sparsity) and reinitializing the weights to new values.



(c) 1D Training after reducing dimensionality of search space (sparsity) and reinitializing the weights to original values.



(d) 1D Training after reducing dimensionality of search space (sparsity) and continuing to train from most recent values.

Figure 5.2: Plots showing how differing methods of weight initialization for training sparse neural networks may affect the training of the final neural network. In all plots, the white 'x' represents the starting point for the training algorithm. The 'o' is the final state after training. The colors in each plot represent the value of the loss function that the optimization algorithm is trying to minimize. An ideal algorithm for adding sparsity to a neural network would be able to always find the global minima for the loss function after the dimensionality has been reduced.

We simulated multiple initializations of a neural network that were trained and pruned using the one-time pruning method with a variety of wait times (epochs). The resulting performances (both ideal accuracy and under variations) are shown in Fig. 5.3. The performances for random pruning are shown on the same plot as a performance reference point; if a pruning algorithm cannot do better than random initialization, then it is not a useful algorithm.

Our simulation results suggest that for smaller fully-connected neural networks, one-time pruning does not offer any benefit to randomly generating a sparse architecture. If anything, it either decreases the possible accuracy, increases the spread of accuracies, and increases the amount of time needed for training. It is unclear as to why the prune-once method did

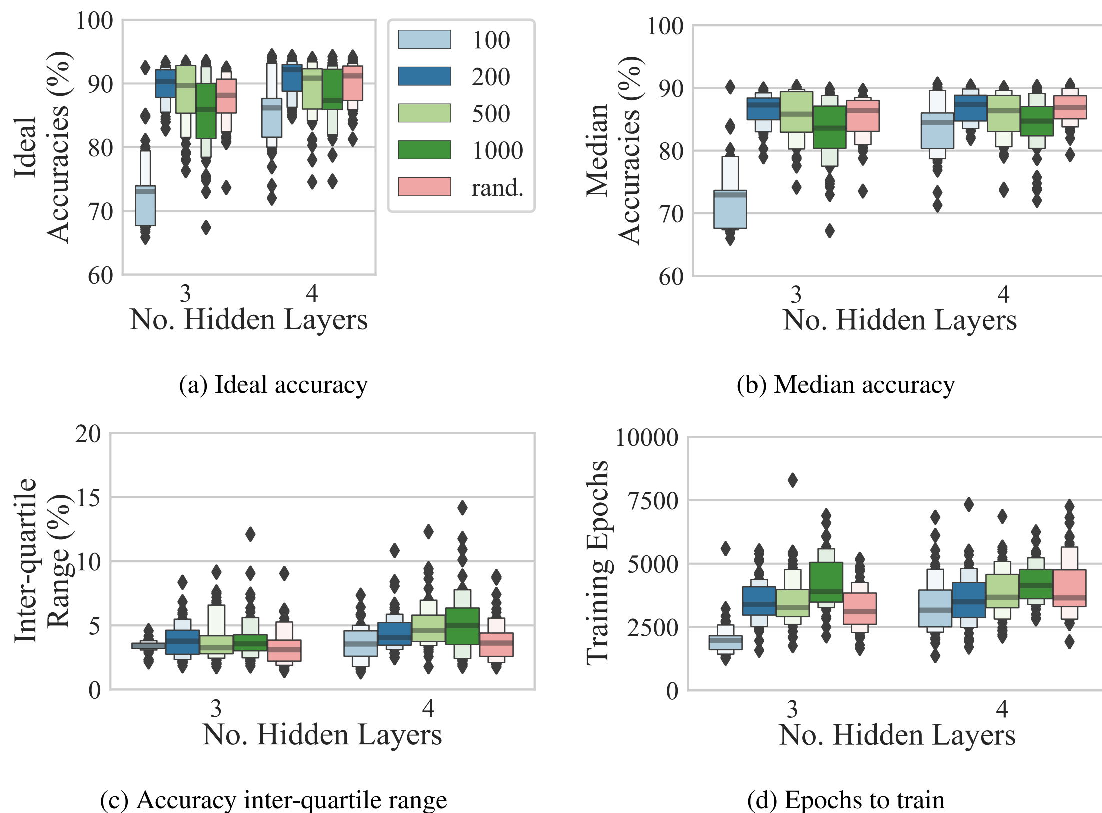


Figure 5.3: Results of using the "prune once" technique to generate sparse neural networks. The results of 50 initializations of the same neural network tested on 1000 network variations are represented by each boxen plot. The numbers in the legend represent the number of epochs of training before pruning the network, and rand. is the case where the sparsity is set randomly before training starts.

not perform any better than random generation, but it is likely related to the limitations in reducing the dimensions of the parameter search space. The performance first increased with the number of wait epochs, but then it deteriorated after the wait time was extended, which suggests that waiting too long to prune the network allows it to start fitting the data too well with the full dimensionality, after which similar performance is then unreachable after removing some of those dimensions.

5.2.2 Gradual Pruning

Gradual pruning can be broken down further into a few sub-algorithms. We can gradually prune all layers in parallel, start with the shallower layers and move to deeper layers, or we can do the reverse and start with deeper layers All three of these will be simulated and compared in this section. Based on the findings about reducing the dimensionality of the search space given in the previous section, our initial guess was that gradual pruning would be better able to find a sparse neural network; instead of eliminating the pruned dimensions all at once, the dimensionality would be slowly reduced. In theory, this should help mitigate the problems faced by one-time pruning. The results of our simulations are shown in 5.4.

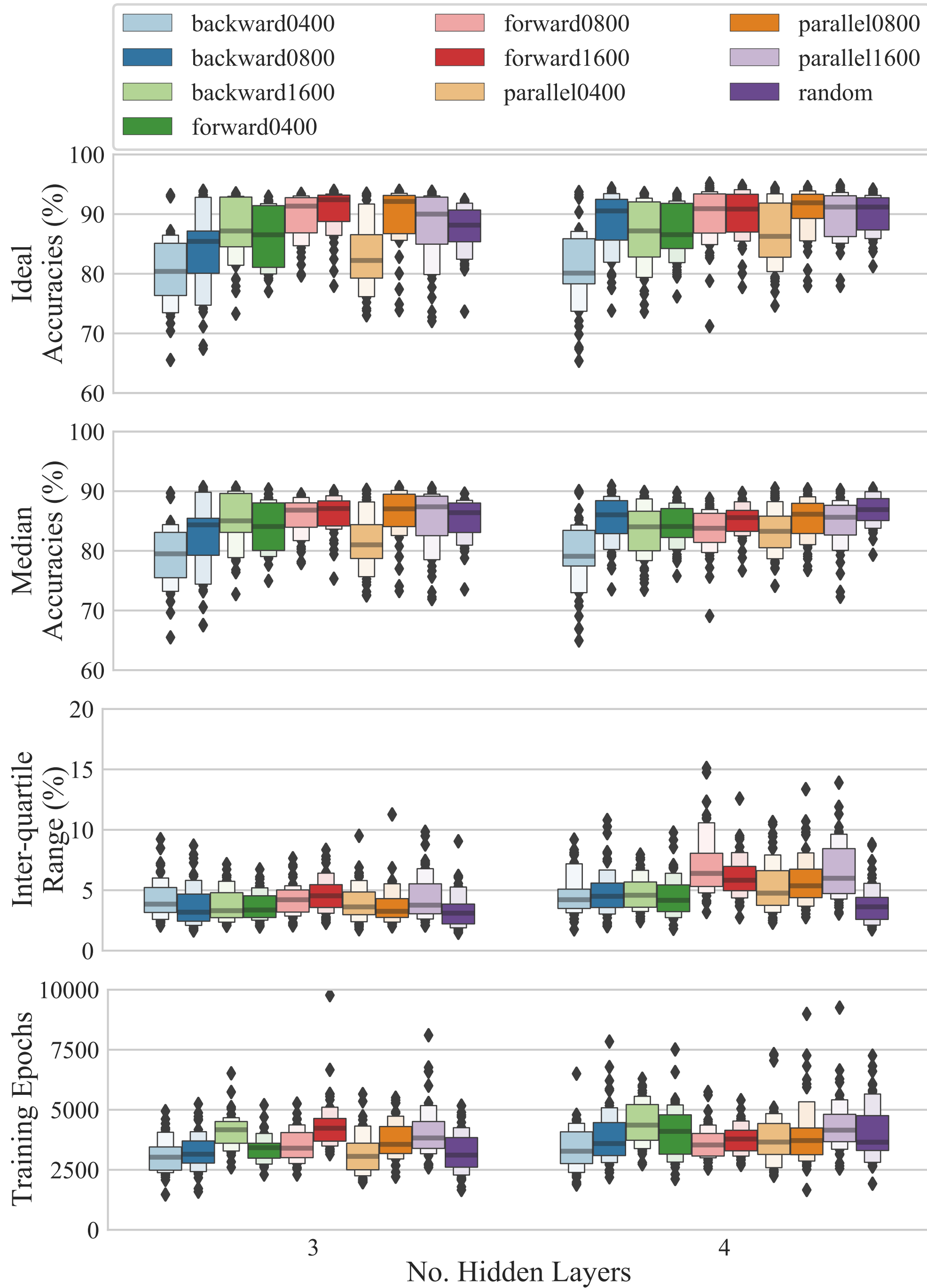


Figure 5.4: Results of using gradual pruning to introduce sparsity into a neural network. In each legend label, the first part tells the direction of the pruning (backward = deepest to shallowest, parallel = all layers at once), and the second value is the number of epochs over which the entire pruning process is completed. For example, "forward_0800" prunes one synapse in every neuron in the first layer until all neurons in the layer have only three inputs, and then it starts to prune the next deepest layer. Each pruning step is spaced across 800 epochs, at which point the pruning process is considered completed.

One interesting finding from the simulation of different methods of gradual pruning is that pruning forward was more effective than backward or parallel pruning. Our intuition is that this is caused by the fact that both parallel and backward pruning are more likely to disable connections that have not yet been confirmed to be “low impact”. When synapses in deeper layers are pruned, larger chunks of the estimated function are removed. As a visual example, recall the network shown in 1.2b and described by (1.3). If we disable the connection between the first layer top neuron with the first input feature x_0, the function for the network becomes

$$\begin{aligned} y = g_2(w_0(g_1(v_{00}g_0(u_{01}x_1 + u_{02}x_2) + v_{01}g_0(u_{10}x_0 + u_{11}x_1 + u_{12}x_2)) \\ + w_1(g_1(v_{10}g_0(u_{01}x_1 + u_{02}x_2) + v_{11}g_0(u_{10}x_0 + u_{11}x_1 + u_{12}x_2)) \\ + w_2(g_1(v_{20}g_0(u_{01}x_1 + u_{02}x_2) + v_{21}g_0(u_{10}x_0 + u_{11}x_1 + u_{12}x_2)))) \end{aligned} \tag{5.1}$$

while removing the connection between the second layer top neuron and the output neuron results in

$$\begin{aligned} y = g_2(w_1(g_1(v_{10}g_0(u_{00}x_0 + u_{01}x_1 + u_{02}x_2) + v_{11}g_0(u_{10}x_0 + u_{11}x_1 + u_{12}x_2)) \\ + w_2(g_1(v_{20}g_0(u_{00}x_0 + u_{01}x_1 + u_{02}x_2) + v_{21}g_0(u_{10}x_0 + u_{11}x_1 + u_{12}x_2)))) \end{aligned} \tag{5.2}$$

Because backward and parallel pruning remove connections in the deeper layers near the beginning of the pruning algorithm, it may be removing connections that may still prove to be necessary later on. Forward pruning is still prone to deleting needed connections in the last layer, but the difference in performance suggests that the network has had more time to train the deeper layers and confirm which connections are more important. This may also be the reason why allowing more epochs for the pruning process (e.g., 1200 instead of 200) results in better networks for both backward and parallel pruning.

The possible range of performance for gradually pruning networks is noticeably improved over the one-time pruning method; the median and ideal accuracy are both at least comparable to random sparsity. In some cases, it gradual pruning is better, though it is

still not guaranteed to outperform randomized sparsity. The inter-quartile range of the accuracy of 1000 variations of a neural network that is gradually pruned is worse than random initialization in most cases. This can be improved by adding noise to the training process to make sure that the learned weights do not result in an over-fit network.

5.2.3 Repeated full pruning

The main obstacle that we want to avoid in pruning the neurual network is that pruning reduces the dimensionality of our weight search space and may prevent us from reaching an optimal point. In other words, we want to be able to find the best-performing sparse network without limiting the search space of the optimization algorithm. The only way to do this is to allow all weights to be updated even after pruning. This of course will not result in a sparse neural network; the network will still be a dense fully connected neural network. However, it is possible that as we prune and retrain, one of the resulting sparse architectures that is created along the way will be a good candidate for a final sparse architecture. By keeping track of the loss of each sparse network that is generated, we can compare the performance of multiple architectures at once while continuing to search through the entire space.

For this method, we prune the neural network to the desired level of sparsity after N epochs; the training continues to apply gradient descent to all of the weights, including the weights that were reset to zero by pruning. This initial sparse network would be backed up, and the calculated loss would be saved as the "best loss" so far. Every N epochs, this process would repeat, and the loss of the pruned network would be computed and compared to the loss of the previous best pruned network. If the new pruned network ends up having a lower loss, the new network is saved and replaces the previous back-up, and the saved best loss value is updated. Similar to how early stopping works in training, this continues until no new better sparse network is discovered after M prunings; at this point, the best sparse network is loaded from the backup, and training continues as in the the other methods, with only the remaining weights being updated by the optimization algorithm.

We found that this method of searching for sparse neural networks, while not a perfect solution, was significantly more likely to generate a high-accuracy sparse network than any of the other methods previously discussed, as shown in Fig. 5.5. The performance is better both in terms of ideal accuracy and performance under parameter variations, but waiting too long between prunings decreases the chances of finding a good sparse network. We observed that pruning the network less frequently (e.g., every 50 or 100 epochs) resulted in performance worse than random initialization, so these examples are not plotted here for the sake of space.

We then wondered if gradually pushing the lower weights to zero instead of simply zeroing them each time we pruned the network would lead to similar or better results faster. Our reasoning was that zeroing so many of the dimensions at once would cause the path to the best sparse network to be very jagged, sometimes taking large steps in the wrong directions. By reducing the weights instead of zeroing them, we hoped that the backtracking would be minimized. We decided to follow the example used in [74] where weights close to zero were pushed to zero more quickly and weights closer to the most significant weights were less-drastically reduced.

The method we use to reduce the lower weights is to first determine the top s synaptic weights in each neuron (represented by a row in the weight matrix) in each layer (in terms of magnitude). A coefficient matrix is generated for each layer with the same number of values as there are weights in the layer. The coefficients corresponding to the tops s weights in each neuron are one. The other coefficients c_j for a given neuron are calculated by taking the ratio of its own magnitude divided by the magnitude of the lowest of the top s weights for that neuron. Optionally, this ratio can be raised to a power p, as shown in (5.3). An example is given for deriving the new weights in (), where $s = 2$ and $p = 3$. Note that the $\odot$ represents the Hadamard product, or elementwise multiplication of the matrices.

$$coefficients_{ij} = \left(\frac{|w_{ij}|}{min(max_3(row_j))} \right)^p \tag{5.3}$$

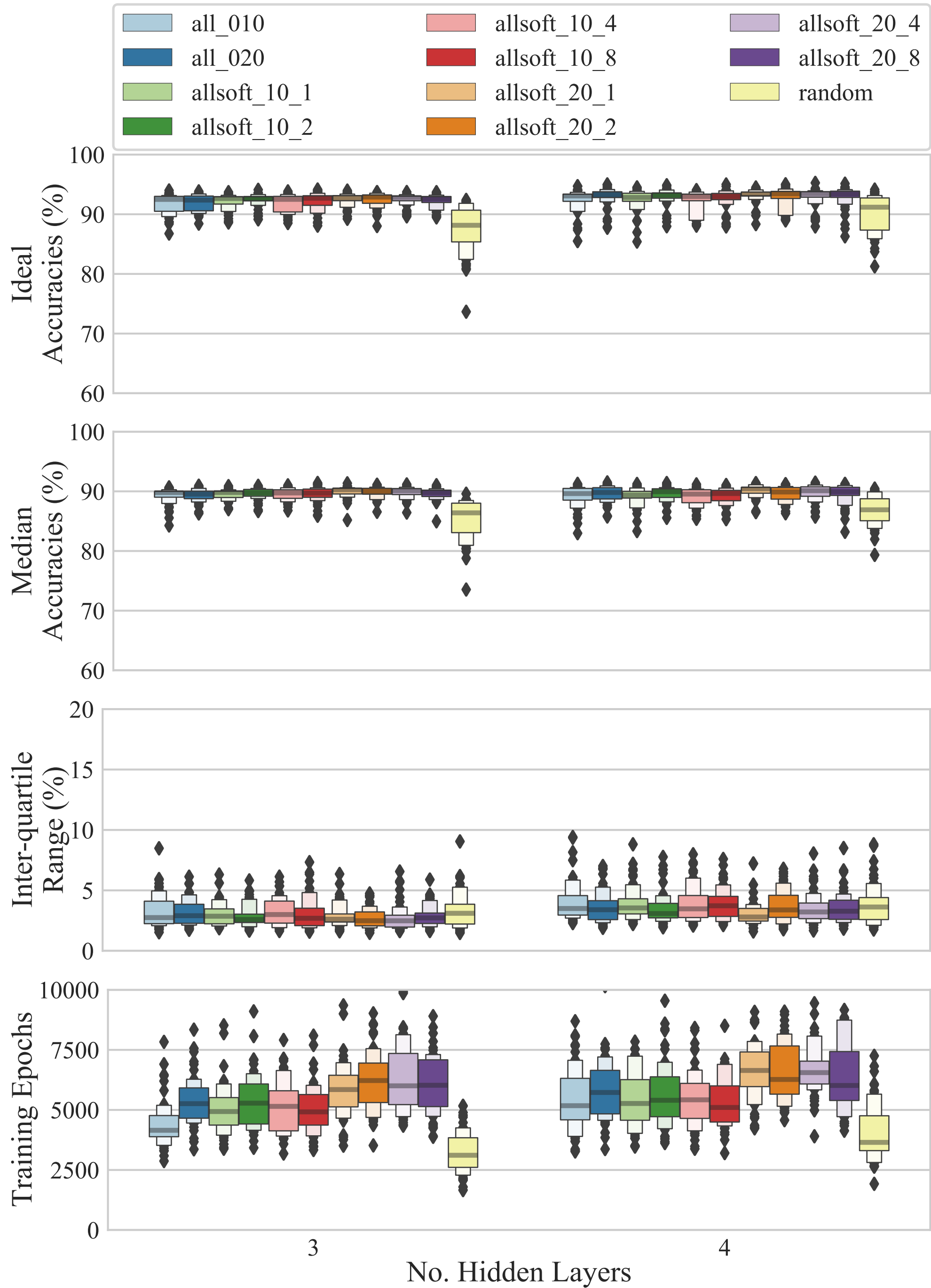


Figure 5.5: Results of using repeated full pruning to introduce sparsity into a neural network. In each legend label, the first number tells the number of epochs waited before pruning, and the second number (if present) is the power of the softening function (5.3).

$$W_m = \begin{bmatrix} -2 & 1 & 2 & -1 \\ -1 & -3 & 0 & 1 \\ 3 & 1 & -3 & -2 \end{bmatrix} \Rightarrow C_m = \begin{bmatrix} 1 & 1/2^3 & 1 & 1/2^3 \\ 1/2^3 & 1 & 0 & 1 \\ 1 & 1/3^3 & 1 & 2^3/3^3 \end{bmatrix} \Rightarrow W'_m = W_m \odot C_m \quad (5.4)$$

Fig. 5.5 shows that using the softer weight reduction function may result in similar performance to the case where all lower weights are zeroed, but the number of epochs required for training may increase.

5.3 Again, But With More Parameters

For the simulations in the previous section, recall that we ran the simulations using neural networks with layer sizes described by Table 5.1. While these pre-set shapes may work well for fully-connected neural networks as discussed in the preceding chapter, they start the sparsity search space off with an already reduced dimensionality. In this section, we ran the same simulations, but this time every layer was assigned the same number of neurons. This was repeated for layer sizes of 12, 20, and 30 neurons in each hidden layer.

The results for these simulations for the prune-once, gradual, and full-prune methods are shown in Figs. 5.6 – 5.8 respectively. As was the case before, the prune-once method is least effective with backward gradual pruning also performing relatively poorly. Again, repeated full and soft pruning gave the best results and showed some improvement as the initial network size increased. One surprising change is that random sparsity generation performs better as the size of the initial neural network increases; this is likely because an increased number of neurons in each layer makes it more likely that the right synaptic combinations will be made in each layer, even if selected randomly. It is interesting, then, that purposeful methods such as prune once and gradual pruning do not achieve similar results, but this could be an effect of removing search dimensions while in the middle of a search, which as we have seen may interrupt the current search path and get the optimization function stuck in a local minima.

It actually makes a lot of sense that starting with a larger size can give a better result, especially in terms of ideal (no variation) accuracy. Although each test involved the same level of input sparsity (three synapses per neuron), the architectures with a larger initial size are more likely to end up with more parameters in their final form, which leads to better

fitting. At the same time, this may also lead to overfitting, but this was not observed in these simulations.

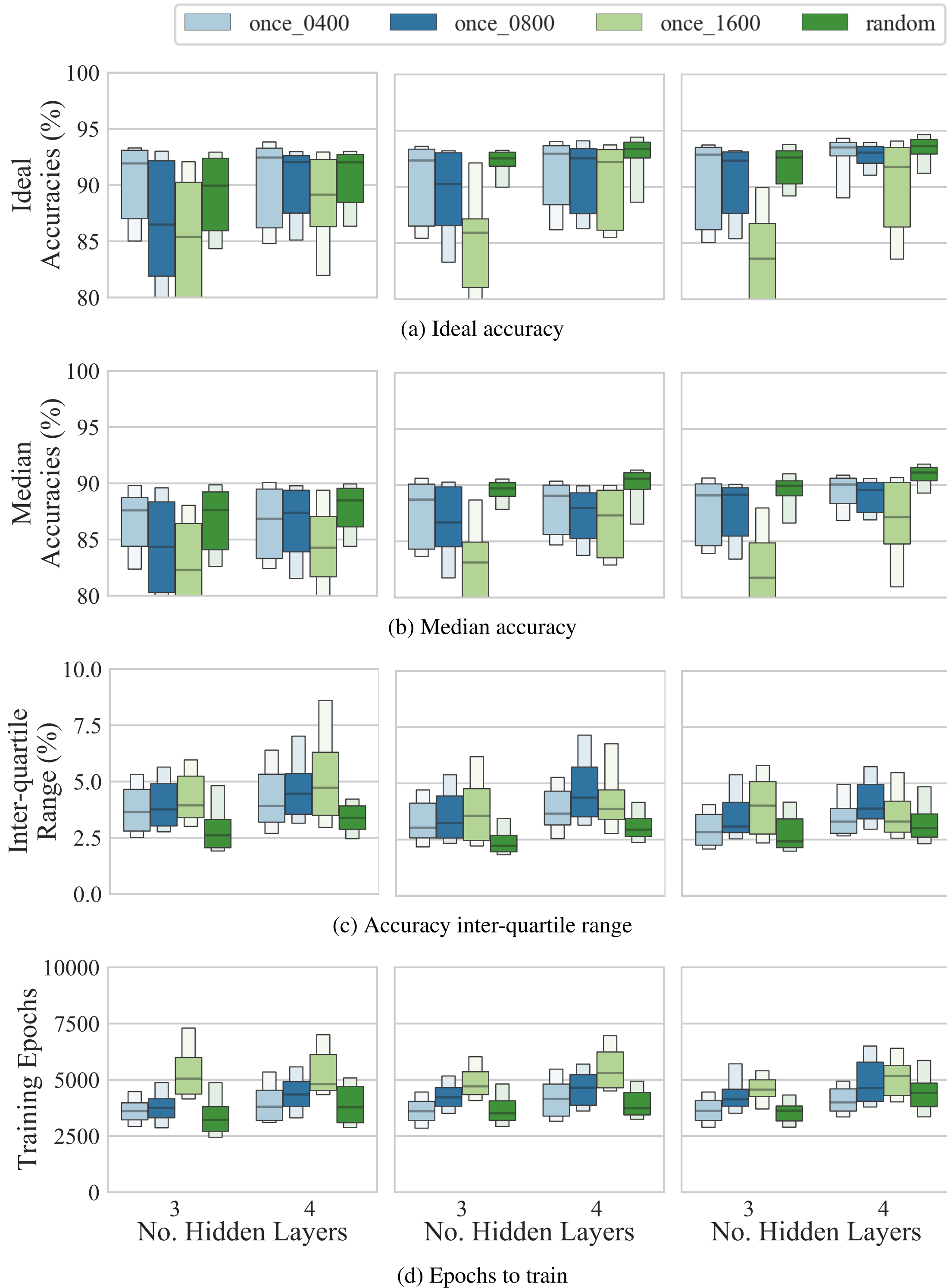
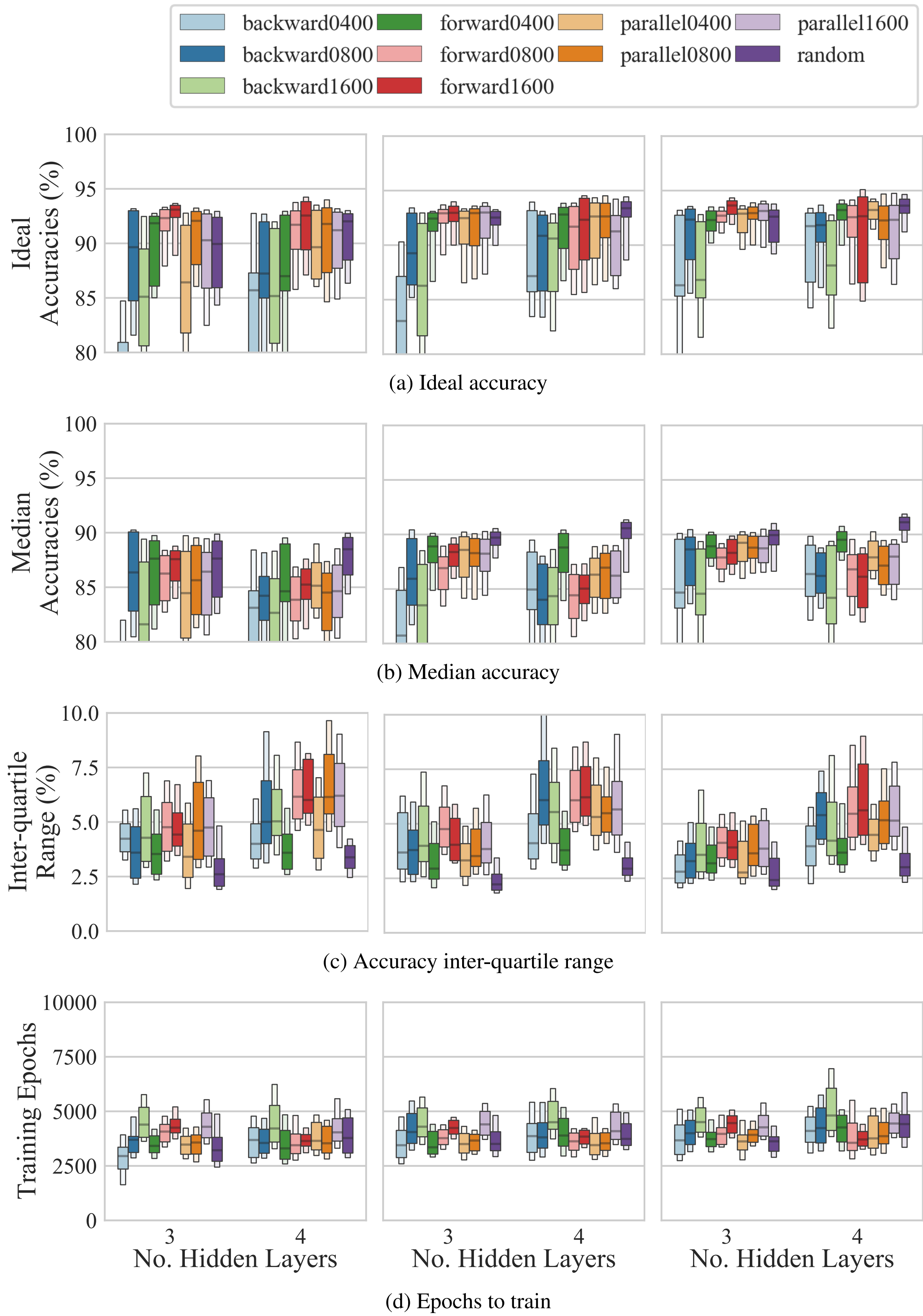


Figure 5.6: From right to left, the plots in each row show the results for repeated full pruning methods when starting out with larger hidden layers with 12, 20, and 30 neurons per layer respectively. Note that the range of the y axis has been reduced in this plot. Outliers are hidden for clarity.



(a) Ideal accuracy

(b) Median accuracy

(c) Accuracy inter-quartile range

(d) Epochs to train

Figure 5.7: From right to left, the plots in each row show the results for gradual pruning methods when starting out with larger hidden layers with 12, 20, and 30 neurons per layer respectively. Outliers are hidden for clarity.

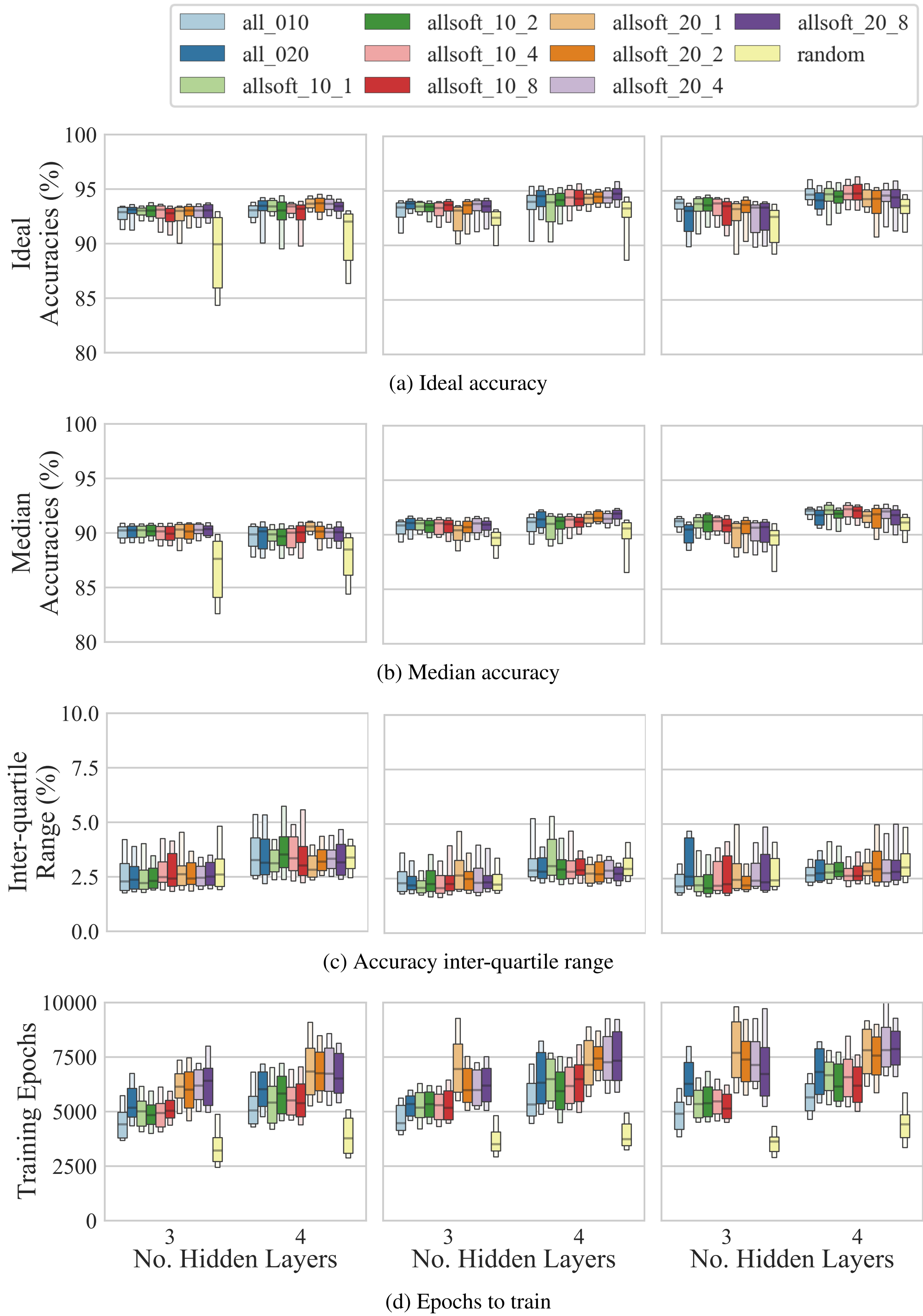


Figure 5.8: From right to left, the plots in each row show the results for repeated full pruning methods when starting out with larger hidden layers with 12, 20, and 30 neurons per layer respectively. Note that the range of the y axis has been reduced in this plot. Outliers are hidden for clarity.

CHAPTER 6
FUTURE WORK

6.1 Better Modeling

This work is a part of the foundation for methods and practices that are necessary for improving performance in approximate computing, both analog and digital. However, simply using these methods to train a neural network is not all that is required to successfully deploy a neural network onto analog hardware. As was in chapter 2, analog hardware does not accurately represent the linear mathematics involved in neural network layers. Not only are they nonlinear, the multiplicative weights will not be constant for all inputs. In general, the effective gain of the transistor will decrease as the amplitude of the input signal decreases, leading to compression and clipping.

Because of this and other factors, the behavior of the ideal layers used in these simulations will be very different from real circuits. To more faithfully represent the hardware neural network, the training needs to be done using transistor-level models. This may sacrifice how effectively the neural networks can be trained since it would not be possible to take advantage of efficient vector-matrix multipliers. Still, accurate representation of the ideal model combined with methods used here to prepare the network for variations should provide the best chance for designing a robust analog neural network.

6.2 Improving the Sparse Network Search

The results with finding high-accuracy sparse neural networks consistently showed that repeated full pruning works best for smaller networks with fewer parameters. Repeating these simulations with a subset of the MNIST dataset revealed that this may not be the case for larger neural networks. The network shapes we tested were two- and three-hidden-layer

fully-connected networks with 200 neurons in each hidden layer. Using a sparsity of 3 did not give enough accuracy for this small number of layers, but using 5 synapses per neuron gave ideal accuracies close to 100%. The results of pruning using forward and parallel gradual pruning as well as full and full soft pruning are shown in Fig. 6.1. Note that the simulations are incomplete for the two-hidden-layer network and do not include the results for soft full pruning.

In these simulations, it seems that none of the structured pruning methods work as well as simple random initialization, including repeated full pruning. One guess as to why this is may be that, with larger networks, it is best to start with all weights near the origin in order to find the best synaptic weights given a sparse architecture. The optimization function may have been getting stuck in local minima when the final training step was completed after structured pruning. If this is the case, then it may be better for larger networks to have their weights re-initialized or restored to their original state after pruning. More research needs to be done to confirm why the results here were different than for the smaller networks.

One possible change that could be made would be to begin with random pruning and then switch to repeated full pruning after a number of epochs in order to check a few nearby sparse architectures that are nearby in the full-dimensional search space. Another avenue that would be important to explore is if high-accuracy can be achieved using a predetermined structure for sparsity. For example, if a single neuron with three synaptic weights that connected to three neighboring neurons from the preceeding layer, the connectivity between the layers could be simplified, and the length of the connections would be reduced. Since randomized sparsity seems to work well for larger neural networks, it would not be surprising if predetermined sparsity could also be trained to work well in large networks.

6.3 Low-power Binary Multiplication: Bit Shift

Arguably the simplest multiplication to perform in base-10 math, aside from multiply by 0 or 1, is to multiply by 10. In base-10, multiplying by 10 is as simple as shifting the decimal

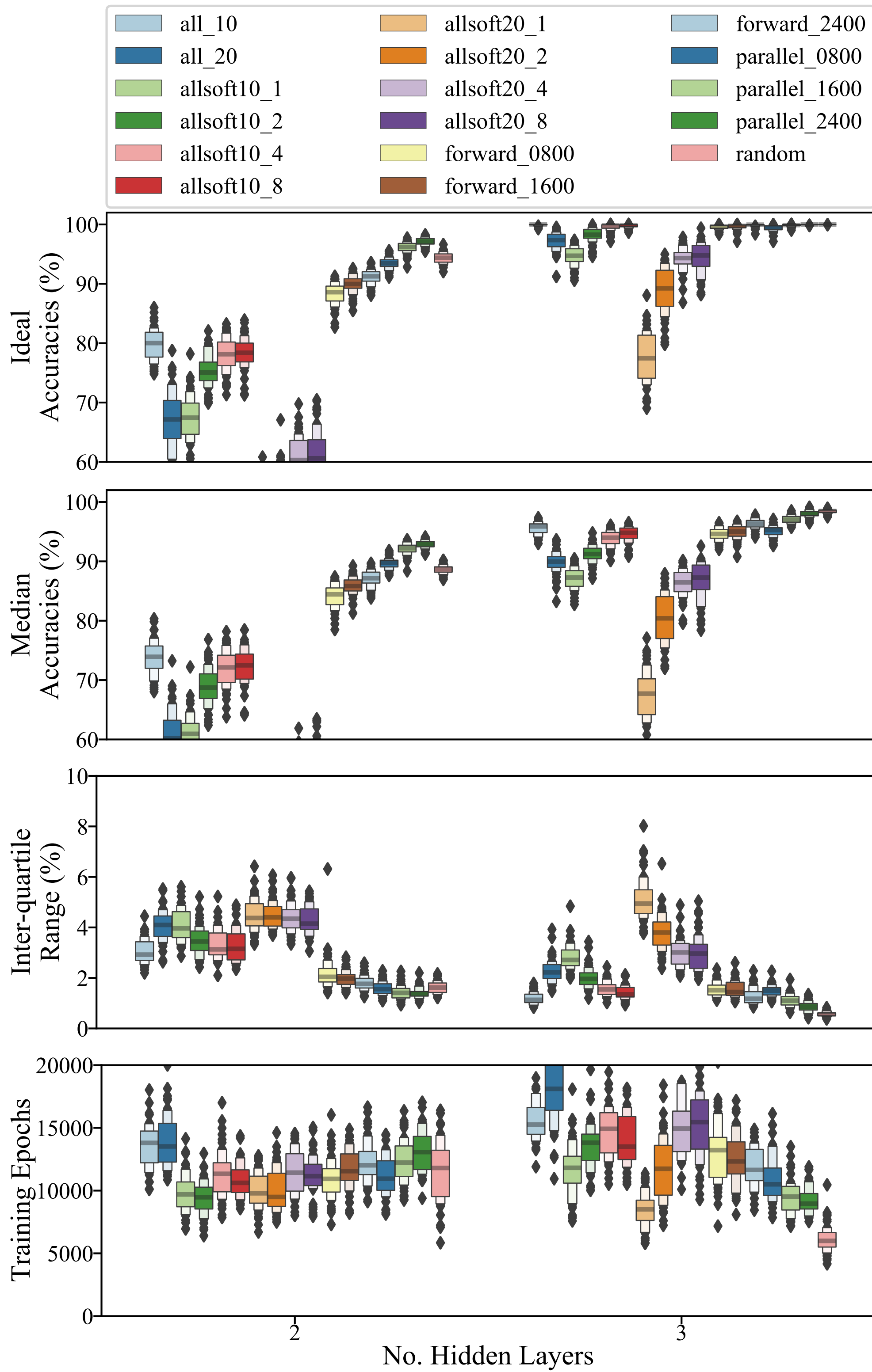


Figure 6.1: Pruning methods using larger neural networks with a subset of the MNIST dataset.

point one digit to the right, and dividing by 10 is the same as shifting the decimal point one digit to the left. Multiplying by any integer power of 10 is the same as shifting the decimal point to the right or left by the same number. For example, $1000 = 10^3 \Rightarrow$ shift decimal right three digits. For the case of base-2 (i.e., binary) math, multiplying by a power of two is the same as shifting all of the bits right or left, depending on the sign and magnitude of the power. Fig. 6.2 shows an example of bit-shift multiplication.

As one can imagine, the amount of energy and transistors required to perform a bit shift is orders of magnitude less than what is required for full-precision multiplication. A common structure to use for bit shifts is the shift register, which is made up of a series of D Flip Flops (DFF), one for each bit. Depending on the architecture, the standard DFF is made up of about 48 transistors [75], so a 32-bit shift register would require about 1536 transistors, which is only 10% of what is needed for a 32-bit multiplier. Using alternative structures for the DFF, an even greater reduction is possible.

6.3.1 Quantization Error

When training a neural network or using a pre-trained network, the weights are usually full-precision; they have been adjusted with enough detail to capture the minute qualities of the dataset. While some of the weights may be able to change small amounts without changing the accuracy, some of the weights are very sensitive to change and can have a major impact on performance. The challenge, then, with quantizing multiplication to a power of two (bit shift) is dealing with the error that results from limiting the precision of the multiplicative weights. A power-of-two binary number is all zeros except for one high bit; in other words, it is a one-hot value.

	0	0	0	1	0	1	1	=	11
x	0	0	0	0	1	0	0	=	4
	0	1	0	1	1	0	0	=	44

Figure 6.2: Example of multiplying a number by $4 = 2^2$, which is the same as a shifting all bits to the left (or "decimal point" to the right) two positions.

Before quantizing the weights, we first must decide on the level of precision we want for our quantized weights. Arbitrarily, we decided to represent out weights as signed fixed precision numbers with one sign bit, five integer bits, and eleven decimal bits. In terms of absolute values, this can represent numbers as large as 32 and as small as $2^{-11} = 0.00048828125$. For most neural networks, it is unlikely that we will ever have a single synaptic weight that is that small, so we will assume that 2^{-11} is the smallest number possible in the following analysis

Algorithm 1 shows how the closest power of two is calculated. Put simply, we take the absolute value of the given number and keep track of the sign. Then we calculate the two closest integer powers of two using the base-two logarithm. Whichever power of two is closer to the original value is returned. Using this logic, the maximum absolute error between the original and quantized values would occur at the average between two powers of two. With this number, the maximum absolute error is about 33%.

$$n = \frac{2^p + 2^{p+1}}{2} = 2^{p-1} + 2^p \tag{6.1}$$

$$err_1 = \frac{n - 2^p}{n} = \frac{2^{p-1} + 2^p - 2^p}{2^{p-1} + 2^p} = \frac{1}{3} \tag{6.2}$$

$$err_2 = \frac{n - 2^{p+1}}{n} = \frac{2^{p-1} + 2^p - 2^{p+1}}{2^{p-1} + 2^p} = -\frac{1}{3} \tag{6.3}$$

However, we would also like to know what the distribution of the errors is between the maximum and minimum error.

The beginning assumption is that any value between the maximum and minimum weight value is equally possible for a synaptic weight to assume, so the value for a weight is a uniform random variable W with minimum -32 and maximum 32:

$$W \sim \mathcal{U}(-32, 32) \tag{6.4}$$

Algorithm 1: Finding the closest power of 2

Input : Number to quantize $n \neq 0$
Output : Closest one-hot value to n

begin

$s \leftarrow \texttt{sign}(n)$
$absn \leftarrow \texttt{abs}(n)$
$p_1 \leftarrow \texttt{floor}\ (log_2(absn))$
$p_2 \leftarrow \texttt{ceil}\ (log_2(absn))$
if $absn - 2^{p_1} < 2^{p_2} - absn$ **then return** $s \times 2^{p_1}$
else return $s \times 2^{p_2}$

Using Python, we simulated randomly selecting one million values from W, finding the closest number that is a power of two, and then calculating the error as in

$$err_2(w) = sign(w) \times \frac{w - w_2}{w} \tag{6.5}$$

where w is the full-precision number and w_2 is the closest power of two. The sign of the weight is used because we are interested in the error of the absolute value of the weight rather than the signed weight. Fig. 6.3 shows how the values are quantized and a histogram of the errors calculated, normalized to the total number of samples; the red line in the plot is the probability density function (PDF) for the error y such that $y = err_2(w)$. The equation for the PDF is

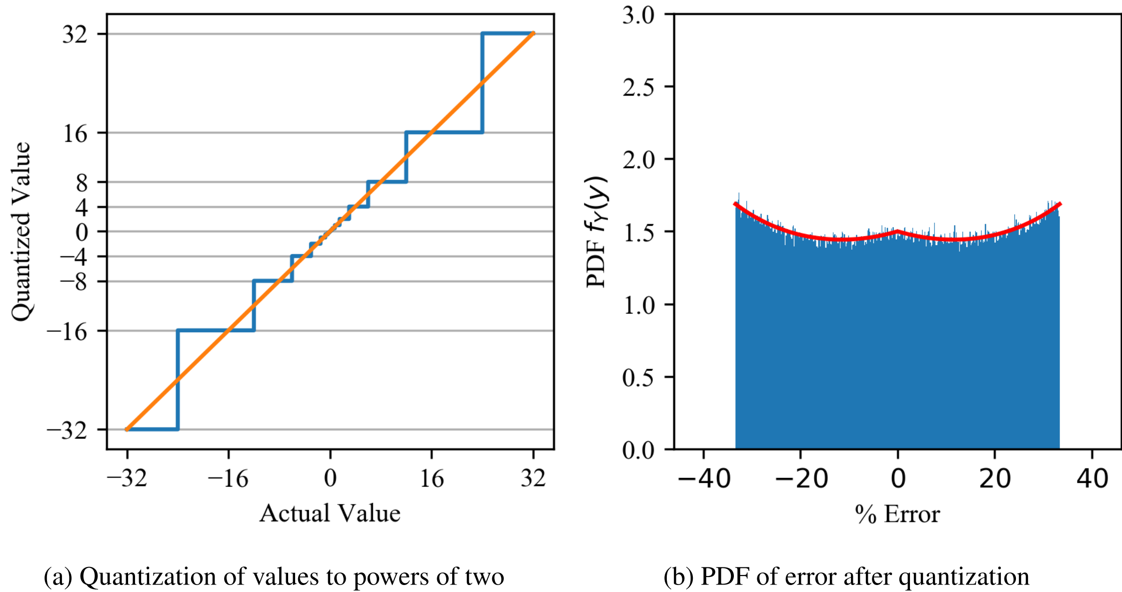
$$f_Y(y) = \begin{cases} \frac{1}{2}\left(\frac{2}{(1-y)^2} + \frac{1}{(1+y)^2}\right), -\frac{1}{3} \leq y < 0 \\ \frac{1}{2}\left(\frac{1}{(1-y)^2} + \frac{2}{(1+y)^2}\right), 0 \leq y \leq \frac{1}{3} \end{cases} \tag{6.6}$$

Others have mistakenly calculated the closest power of two as

$$p = s \times round(log_2(abs(n))) \tag{6.7}$$

However, this may result in even larger error between the actual and quantized values. Fig. 6.4 shows the histogram for the distribution of error using this equation. Obviously, for less error, it is better to use algorithm 1.

The PDF for the error is almost uniform, which means that the quantized weight has



(a) Quantization of values to powers of two

(b) PDF of error after quantization

Figure 6.3: (a) Plot showing how the full-precision values between -32 and 32 are quantized down to one-hot form (closest power of two). (b) Histogram of the resulting errors when comparing the original value to the quantized value.

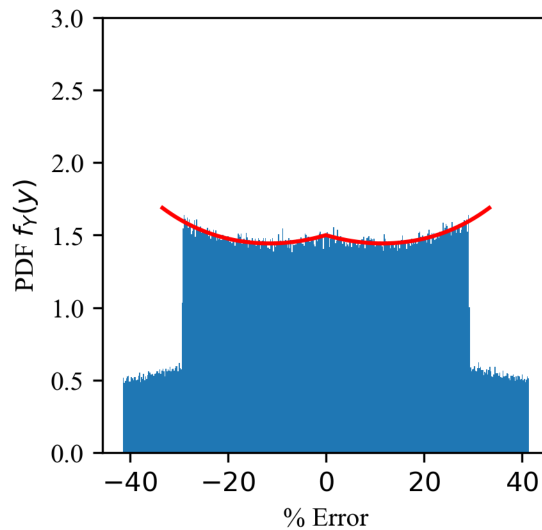


Figure 6.4: Histogram of quantization error when using equation (6.7)

about as much of a chance to have high quantization error (33%) as it is to have low or no quantization error. This is much more error than we were dealing with when considering noise and variation in analog circuits, and analog circuits much more likely to have low variability. However, the biggest difference and benefit to this type of noise is that it is completely predictable. First, we know exactly what the value of each weight is after

training. We also know how each weight will be quantized. For example, we know that the value 3.2 will be rounded to 4, and the error will be 25%. If we know exactly what values the synaptic weights will take after they are quantized, is it possible to train the network such that the final weights are close to powers of two so that the change in accuracy after quantization is minimal?

6.3.2 Training for One-Hot Quantization

In this section, we will attempt to use the same methods we used to suppress errors due to variations in the analog circuit. For these simulations, we will use the power grid stability dataset as well as MNIST to show the effectiveness of each training method. First, we can try training the neural network while adding noise to the weights. The profile for the noise will be similar to the PDF in Fig. 6.3: a uniform random variable with values between $-\frac{1}{3}$ and $\frac{1}{3}$. We can also try adding a term to our loss function that will cause the loss to decrease as the weight values get closer to a power of two, as in (6.8), where q is the quantized weight value. Using reduced-slope activation functions and repeated quantization during training (like repeated full pruning) may also help to train the neural network for bit-shift multiplication.

$$LOSS_{1HOT} = f_{loss} + \lambda\Sigma|w_{ijk} - q_{ijk}| \tag{6.8}$$

One of the challenges with training low-precision neural networks is arriving at low-precision weights while also training with enough precision in the gradients to allow the weights to change over time and overcome the barriers created by quantization. For example, if the gradient is smaller than the value of the least significant bit, the value of the weight will not change. At the same time, there is no way to calculate the derivative or gradient of the quantization step. As such, common optimization algorithms for low-precision must include a means of bypassing the quantized weights, which normally means keeping track of both the full-precision weights as well as the quantized weights. By adding a power-of-two factor to the loss and periodically quantizing the weights, we can train using full precision

and avoid these problems altogether.

6.3.3 N-Hot Quantization

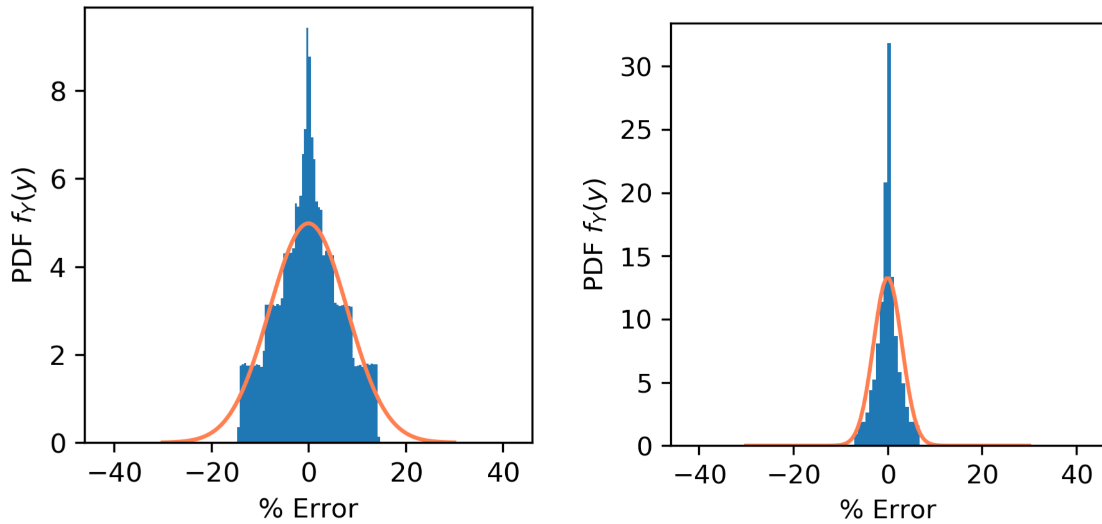
If one bit is not enough to provide the accuracy needed, additional bits may be used. For a single multiplication with one synapse, multiplication with a N-hot (N high bits) number, N shift registers and N-1 adders are needed. For low values of N, this still offers a benefit in terms of transistor count. The algorithm for finding the closest N-hot value to a given number is shown in algorithm 2. Note that in this algorithm, the ClosestPowerOf2 function is given by algorithm 1.

The error historgram for two- and three-hot quantization are plotted in 6.5. The profile of the error for these two cases more closely resemble normal distributions with standard deviations close to $\sigma_2 = 8\%$ and $\sigma_3 = 3\%$ respectively. These normal distributions are plotted as well for comparison. As expected, the standard deviation of the distribution of

Algorithm 2: Finding the closest N-hot value to a given number

Input : Number to quantize $n \neq 0$, Number of hot bits N (integer > 0), Smallest possible power of two $T > 0$
Output : Closest N-hot value to n

```
begin
    if N=1 then
        return ClosestPowerOf2 (n)
    s ← sign(n)
    n ← abs(n)
    absn ← n
    n_2 ← 0
    while N − 1 > 0 do
        p ← floor(log_2(n))
        n ← n − 2^p
        n_2 ← n_2 + 2^p
        N ← N − 1
        if absn < T then break
    n_2 ← n_2 + ClosestPowerOf2(n)
    return s × n_2
```



(a) Two-hot quantization error PDF

(b) Three-hot quantization error PDF

Figure 6.5: Plots showing the histograms of quantization errors for two- and three-hot quantization

error decreases with additional bits, and the maximum possible error is halved for every additional bit. With only two high bits representing a synaptic weight, the equivalent normal distribution for numeric error is very close to the error we used to represent device variations in analog synaptic weights. The methods we used in previous chapters may also prove useful in this case.

CHAPTER 7
CONCLUSION

We have shown that small and simple changes in how a fully-connected neural network is designed and trained can have a significant impact and limit the damage caused by variations in the hardware. Previous techniques designed to regularize the neural networks weights to prevent data overfitting do not directly translate to device overfitting in analog neural networks, but the general ideas can be used to create new methods more targeted to the specific problem of device overfitting. Our simulations have shown that adding noise to the training process by systematically corrupting each parameter that may be affected by variations can help the network to learn to fit a wider variety of weights or handle noise in the input features. We also unintentionally reconfirmed the validity of minibatch and stochastic gradient descent as efficient ways to train a network by showing that it is better to use only one network variation for each training optimization step rather than averaging the gradients of a population of variants.

When deciding on the architecture of the neural network, following these guidelines will lead to less accuracy degradation in the face of corrupted parameters:

1. Activation functions with smaller slopes, especially near $x = 0$, will suppress the variations of the weights in the layer. If it is possible to use a ReLU-like function, it is preferred over the sigmoid and hyperbolic tangent functions.

2. Just as a large number of parameters can cause a neural network to overfit the training data, having more layers and neurons than is necessary will also cause the network to become overfit to one parameter set and less accomodating to variations. Start with smaller neural networks (fewer layers and neurons), and increase the size and depth only as needed.

3. When adding more parameters to improve accuracy, it is better to add more neurons than to add more layers. Adding more neurons only adds more elements to the linear combinations of the following layer while additional layers effectively multiplies the complexity of the approximated function.

4. If more neurons are needed, it is better to add more neurons to the shallower layers. Each layer should have at least the same number of neurons as the next deeper layer.

5. By reducing the number of connections in the neural network (i.e., introducing sparsity), the noise that is propagated through the network is reduced, so there is less impact on the accuracy of the neural network.

In general, it is best to keep the network simple. More parameters means more noise and therefore greater chance in affecting the behavior of the neural network.

One of the biggest challenges in finding a sparse architecture is knowing which connections are okay to remove. Each time a parameter is removed, a dimension of information is lost, and it becomes more difficult to find the best solution with the remaining dimensionality. Techniques that allow access to all available dimensions while searching for a sparse solution are expected to have a better chance of finding the best set of sparse parameters. Still, one aspect of machine learning that adds some uncertainty to the search space is the random initialization of the synaptic weights; depending on their starting values, the final solution could be completely different for each initialization. Because of this, there is no guarantee that the first trained network will be the best, so multiple iterations of training may be required. Using a method that is more likely to produce a solution with high accuracy and robustness would mean fewer training iterations before finding a network with the desired performance.

This research has given more credence to the idea that it is possible to train a digital model of an analog neural network and deploy it to multiple hardware devices with minimal change to the performance. Taking this research to the next step and attempting to train

accurate models of the hardware would be the next step in proving the validity of these findings. A simpler application to explore would be to see how these methods affect approximate computing methods using digital hardware. These would be much easier to implement and test, and there would be no uncertain variability between devices. N-hot multiplication would be a good next candidate.

Appendices

APPENDIX A

DATASETS

In this work, five different datasets are used to test the theories and methods described. Our main task of interest at the start of this project was voice activity detection (VAD), where the goal is to identify human speech both with and without noise or interferance in the audio. However, to confirm the generality of the techniques we present, we needed to use other types of data and explore a variety of network sizes and shapes. The first two datasets listed below are for the VAD task; the other two are for confirming our findings on different data.

A.1 Custom Voice Activity and Noise Dataset

The first task we approached was voice activity detection (VAD), where the goal is to identify human speech both with and without noise or interferance in the audio. For our initial attempts, we used a custom dataset made up of speech samples from The Speech Accent Archive [76] and a variety of noisy audio samples from YouTube, including a lawnmower, a vacuum, cafeteria babble, highway noise, and various machine noise (e.g., air conditioning, dishwasher, etc.). Unfortunately, the links for the YouTube videos were not saved because this dataset was only intended for proof-of-concept work.

There were many issues with this dataset including lack of variety of SNRs and a very limited rane of voice types and noise sources. It was clear that we needed something that would cover more possible environments that a speech detector might be used. Fortunately, the task of picking out voice from noise is a common problem, so there are already a number of great datasets available.

A.2 Microsoft Deep Noise Suppression (MDNS) Challenge

The Microsoft DNS Challenge dataset [77] is intended for deep neural networks to be able to remove the background noise from an audio clip while preserving the human speech. While the purpose is different, the code available for the dataset provided a simple means of generating a large number of noisy speech clips with a variety of SNRs an other customizations.

A.3 The Wisconsin Breast Cancer Dataset

This is a popular dataset that contains the characteristics cells extracted from breast tissue for the purpose of detecting which cells are cancerous and which are benign [78]. Information about the features can be found at
https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic).

A.4 Electrical Grid Stability Dataset

A simple, low-dimension dataset where characteristics of an electrical grid are used to predict whether or not the system is stable [79].

A.5 MNIST

A dataset commonly used for benchmarking and comparison purposes, it is a set of hand-written numberic digits 0–9. Neural networks using this dataset are tasked with correctly identifying which digit is in each image [80].

APPENDIX B

VOICE ACTIVITY DETECTION — FEATURE EXTRACTION

There are many complex algorithms used to extract features from audio data with DSP techniques. Most of these are extremely difficult to implement in analog, and creating a complex feature extraction stage would likely lead to large area and energy overhead. Audio can be classified generally as stationary or transient; stationary audio is semi-constant such as the drone of machinery or babble of a crowd while transient is intermediate and inconsistent such as car horns or hammering. In most situations for VAD, the background audio (noise) is fairly stationary while human speech is more transient, especially when giving commands to digital assistants. Voice also contains harmonic components while most non-speech sounds do not. Our feature extraction stage was derived taking advantage of these characteristics.

The incoming audio signal is first passed through an array of bandpass (BP) filters with selected frequencies with log spacing. The envelope of each band is detected, and the noise and voice signal levels are estimated (NL and SL respectively). NL is estimated as a voltage that slowly follows the minimum of the band envelope; when the envelope is greater than NL, NL increases and vice versa at the same rate. The SL value is estimated as the maximum between the envelope and a voltage that is similar to NL except that it tracks the maximum of the envelope. The tracking voltage rises much faster than it decays. Simplified equations representing this behavior are (B.1) – (B.3); Fig. B.1 contains simplified circuits for obtaining each of these signals, and Fig. B.2 shows examples of these signals. The difference between SL and NL for each band make up the input features for the NN.

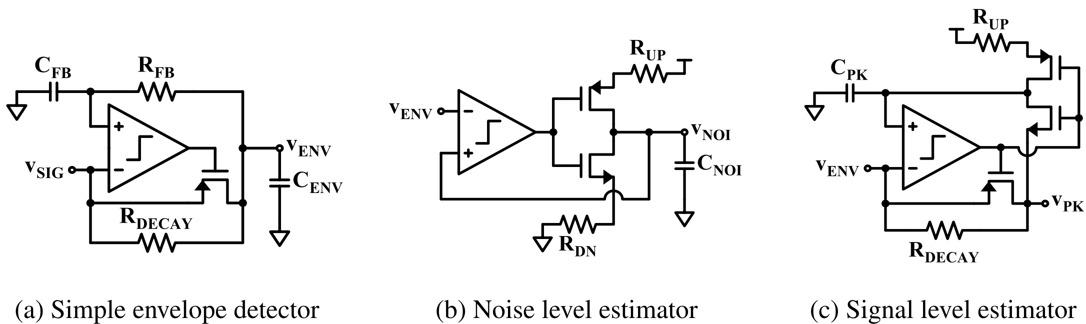
$$V_{NL}(V_{ENV}, V_{NL}, t) = \begin{cases} V_{NL}(t_{cross}) - \tau_f t, & \text{for } V_{ENV} < V_{NL} \\ V_{NL}(t_{cross}) + \tau_r t, & \text{for } V_{ENV} > V_{NL} \end{cases} \tag{B.1}$$

$$V_{SLtrack}(V_{ENV}, V_{SLtrack}, t) = \begin{cases} V_{SLtrack}(t_{cross}) - \tau_f t, & \text{for } V_{ENV} < V_{SLtrack} \\ V_{SLtrack}(t_{cross}) + \tau_r t, & \text{for } V_{ENV} > V_{SLtrack} \end{cases} \tag{B.2}$$

$$V_{SL}(V_{ENV}, V_{SLtrack}) = \begin{cases} V_{SLtrack} & \text{for } V_{ENV} < V_{SL} \\ (V_{ENV} & \text{for } V_{ENV} > V_{SL} \end{cases} \tag{B.3}$$

The feature extraction stage has many parameters that cannot be learned by common training algorithms such as backpropagation (i.e., hyperparameters). Some of the hyperparameters include the range of frequencies for the BP filters; the number of BP filters; the center frequencies, Q, order, and gain for the BP filters; and the time constants for each of the rise and decay rates for the signal and noise level estimators. Hyperparameters are normally set by methodical trial and error, which is the pattern we followed. First, a gridsearch algorithm was used to sweep each of the parameters to discover a range of parameters with the best performance. Random initialization within this range helped ensure that the step size of the grid search did not prematurely rule out any itermediate hyperparamenter values. The values that resulted in the highest accuracy without over-design are shown in Table.

These three genearated signals can be used as features that are input to the neural network. In our earlier work when testing the viability of our research, we used six frequency bands and the difference between the signal and noise features, $SL - NL$. To improve the



(a) Simple envelope detector (b) Noise level estimator (c) Signal level estimator

Figure B.1: Example schematics for possible implementations of the signal and noise level estimators used in feature detection.

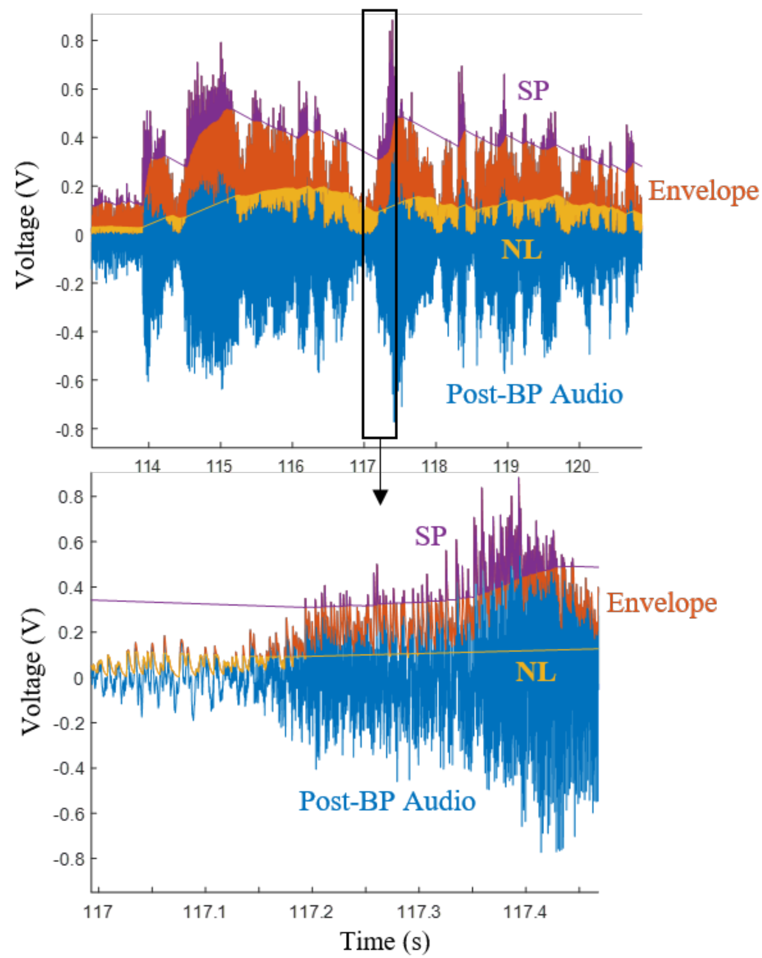


Figure B.2: Example signals generated by the circuits in Fig. B.1.

performance of the network under a wider range of audio levels and SNRs, we also used the NL feature on its own, making a total of 12 features for audio classification.

APPENDIX C
MODELING FEATURE NOISE

One of the challenges that we ran into when trying to simulate or train with noise in the neural network was how to best represent variations that occur in the feature extraction stage. Based on the assumption that most or all the parameters used for feature extraction are controlled by RC time constants (namely the bandpass filter center frequencies and decay rates for the noise and signal level estimators), we decided to characterize the possible variations as multiplicative random values with a gaussian distribution with a mean μ of 1 and a standard deviation of 3σ = 20. This number was used based on the finding that passive values can vary by up to 25% [40–42]. Among the features that were corrupted by random values were the bandpass center frequencies, the filter Q value, filter gain, envelope detector decay time constant, and the rise and fall time constants for the signal peak and noise level estimators.

When generating the features used for classification, the audio signal is passed through the array of bandpass filters, envelope detectors for each band, and the estimators for the signal peak and noise levels for each band. Calculating these features takes a considerable amount of time, and the required time increases with the number of frequency bands and the number of input audio samples. When training an ideal network or with only one device variation, the features can be generated and saved for re-use at the beginning. However, testing the network on multiple variations (hundreds to thousands) or when using population training or noised training with a new device variation for each epoch, the feature extraction must be repeated frequently during training and for each device tested. The amount of time required to train a neural network using the one-varied-device-per-epoch method suddenly increases by more than 100x. Again, the features could all be pre-extracted and saved to a hard drive for later use, but this still creates significant overhead in term of time, and the

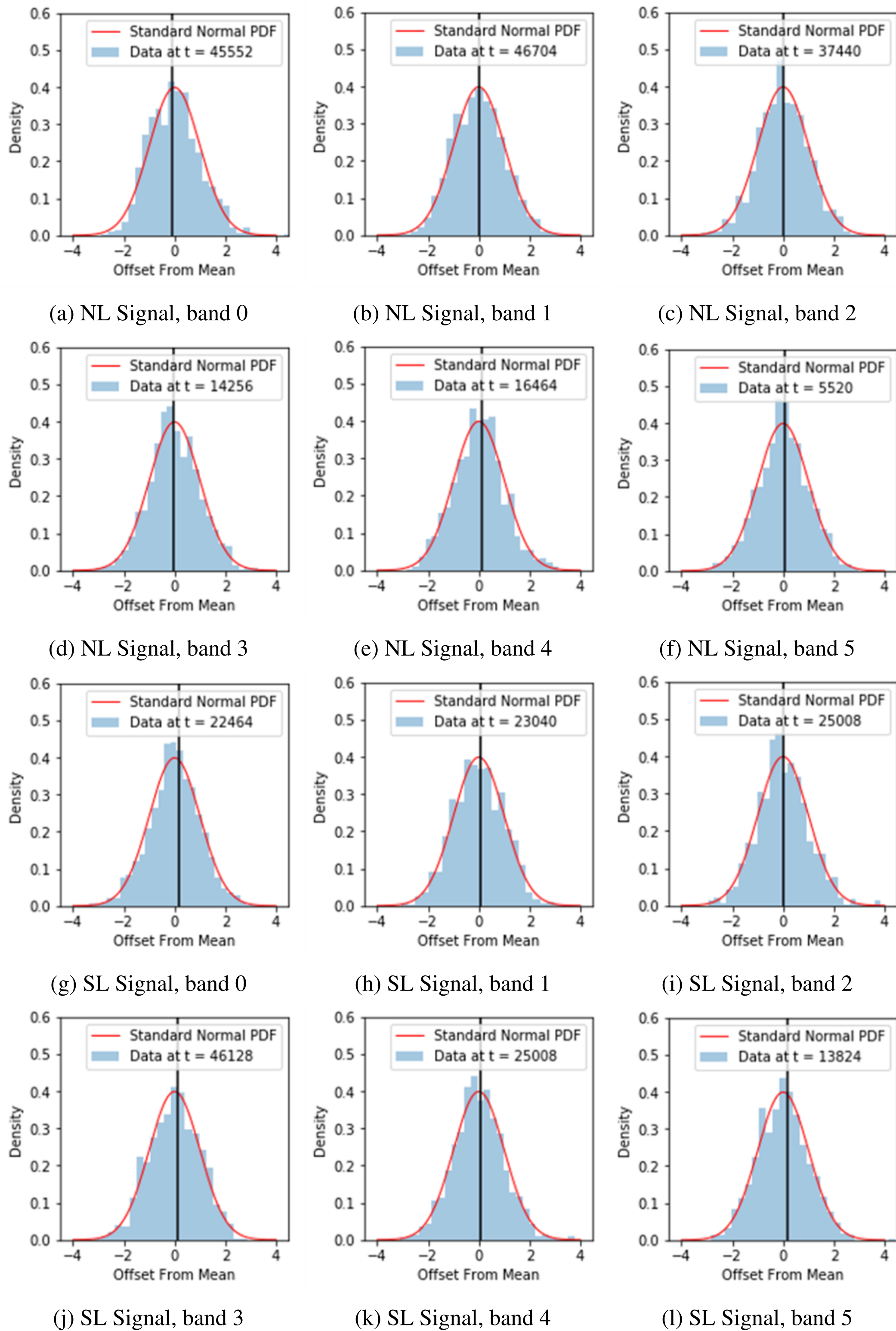
amount of storage space required exceeds several terabytes, depending on the number and length of the audio samples.

To decrease the time needed to extract features and add variations, we characterized the amount of variation that occurs in the final extracted features after multiplying the parameters within the feature extraction stage by the random values as discussed before. The purpose of the characterization was to see if, instead of adding noise to the parameters and re-extracting the features, we could instead extract and save the ideal features and then multiply or add some noise that would result in noisy features with a similar distribution as if we had extracted them with the parameter variations. We did this for multiple audio samples and characterized the randomness observed in the envelope, signal peak, and noise level values for all frequency bands (six in this case).

First, we confirmed that the distribution of the extracted features was a normal distribution. Features were extracted for 1000 variations of the feature extraction stage; for each feature, all 1000 variations were averaged at a point in time and the standard deviation from the mean was calculated. Using these statistics, all 1000 features were normalized (for each point in time individually) and plotted as a histogram. The plots below in Fig. C.1 show that these histograms closely resemble a standard normal distribution. The ideal value was very close to the mean of the variations. This suggests that the distribution of the parameter variations follows a multiplicative gaussian distribution, which supports our goal.

Next, we needed to calculate the standard deviation that we would use for the gaussian distribution that we would draw from when adding noise to a feature. For a given feature, the mean and standard deviation at a point in time were calculated, and the standard deviation was normalized with respect to the mean. The value of this normalized standard deviation was collected for all points in time for that audio sample and plotted as in Fig. C.2. The values roughly follow a right-skewed Gumbel distribution; the distribution parameters were calculated, and the PDF was plotted along with the datapoints to show the similarities. The average normalized standard deviation is around seven or eight percent, with a significant

majority falling below the 10% mark. Using this information, we decided to represent variations in the feature extraction with a gaussian random variable with a mean of one and a standard deviation of 0.1.



(a) NL Signal, band 0 (b) NL Signal, band 1 (c) NL Signal, band 2

(d) NL Signal, band 3 (e) NL Signal, band 4 (f) NL Signal, band 5

(g) SL Signal, band 0 (h) SL Signal, band 1 (i) SL Signal, band 2

(j) SL Signal, band 3 (k) SL Signal, band 4 (l) SL Signal, band 5

Figure C.1: Plots demonstrating the gaussian nature of the features extracted when subjected to gaussian noise added to various parameters. The vertical black line is the ideal value, and t is the number of samples from the start of the audio (at 16kSps). The time shown is randomly selected from the given sample and feature.

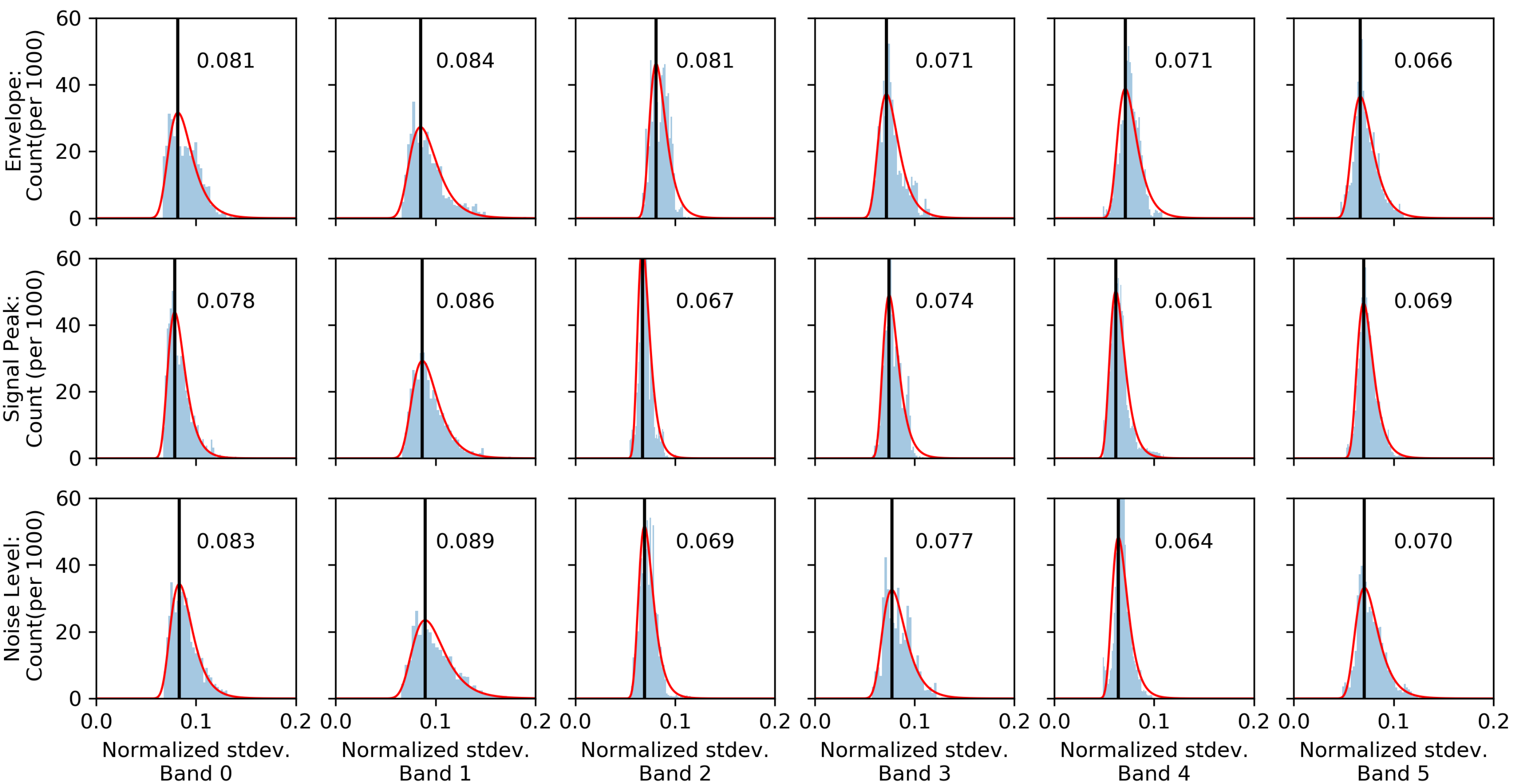


Figure C.2: For a given extracted feature and across all bands, the standard deviation and mean of the feature value is calculated at all time points, and the standard deviations are all normalized to the mean. The vertical black line is the average standard deviation for that feature and band.

REFERENCES

[1] Corby Rosset. *Turing-NLG: A 17-billion-parameter language model by Microsoft*. Accessed Oct. 2020. Microsoft Research Blog.

[2] D. Li et al. "Evaluating the Energy Efficiency of Deep Convolutional Neural Networks on CPUs and GPUs". In: *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom)*, pp. 477–484.

[3] Ermao Cai et al. "NeuralPower: Predict and Deploy Energy-Efficient Convolutional Neural Networks". In: *ArXiv* abs/1710.05420 (2017).

[4] T. Talaska et al. "Analog Programmable Distance Calculation Circuit for Winner Takes All Neural Network Realized in the CMOS Technology". In: *IEEE Transactions on Neural Networks and Learning Systems* 27.3 (2016), pp. 661–73.

[5] Hu Miao et al. "Memristor-based Analog Computation and Neural Network Classification with a Dot Product Engine". In: *Advanced Materials* 30.9 (2018), 1705914 (10 pp.)

[6] Keiron O'Shea and Ryan Nash. "An Introduction to Convolutional Neural Networks". In: *ArXiv* abs/1511.08458 (2015).

[7] Wei Bao, Jun Yue, and Yulei Rao. "A deep learning framework for financial time series using stacked autoencoders and long-short term memory". In: *PLoS ONE* 12 (July 2017).

[8] S. Saha. "A Comprehensive Guide to Convolutional Neural Networks the ELI5 way". In: *Towards Data Science*. URL: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 Accessed 18 Feb 2021.

[9] "Bio-inspired Neurocomputing". In: *Studies in Computational Intelligence* (2021).

[10] Andrew Cotter et al. "Better Mini-Batch Algorithms via Accelerated Gradient Methods". In: *Advances in Neural Information Processing Systems*. Ed. by J. Shawe-Taylor et al. Vol. 24. Curran Associates, Inc., 2011.

[11] G. E. Moore. "Cramming more components onto integrated circuits". In: *Electronics* 38.8 (1965).

[12] Seok Mingoo et al. "Cases for Analog Mixed Signal Computing Integrated Circuits for Deep Neural Networks". In: *2019 International Symposium on VLSI Design, Automation and Test (VLSI-DAT), 22-25 April 2019*. 2019 International Symposium on VLSI Design, Automation and Test (VLSI-DAT). Proceedings. Mingoo, Seok Minhao, Yang Zhewei, Jiang Lazar, A. A. Jae-Sun, Seo: IEEE, 2 pp.

[13] L. Fick et al. “Analog In-Memory Subthreshold Deep Neural Network Accelerator”. In: *2017 IEEE Custom Integrated Circuits Conference (CICC), 30 April-3 May 2017*. 2017 IEEE Custom Integrated Circuits Conference (CICC). Fick, L. Blaauw, D. Sylvester, D. Skrzyniarz, S. Parikh, M. Fick, D.: IEEE, pp. 195–8.

[14] Du Yuan et al. “An Analog Neural Network Computing Engine Using CMOS-Compatible Charge-Trap-Transistor (CTT)”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.10 (2019), pp. 1811–19.

[15] “Computer Organization | Von Neumann architecture”. In: *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/computer-organization-von-neumann-architecture/ Accessed 05 May 2020.

[16] P. Asadi and K. Navi. “A New Low Power 322-bit Multiplier”. In: *World Applied Sciences Journal 2* (2007).

[17] Jong-Hwan Ko et al. “Limiting Numerical Precision of Neural Networks to Achieve Real-time Voice Activity Detection”. In: *IEEE Int. Conf. Acoustics Speech and Signal Processing (ICASSP)*.

[18] Kim Jaehyun, Lee Chaeun, and Choi Kiyoung. “Energy Efficient Analog Synapse/Neuron Circuit for Binarized Neural Networks”. In: *2018 International SoC Design Conference (ISOCC), 12-15 Nov. 2018*. 2018 15th International SoC Design Conference (ISOCC). Jaehyun, Kim Chaeun, Lee Kiyoung, Choi: IEEE, pp. 271–2.

[19] Yang Minhao et al. “A 1uW voice activity detector using analog feature extraction and digital deep neural network”. In: *2018 IEEE International Solid-State Circuits Conference (ISSCC), 11-15 Feb. 2018*. 2018 IEEE International Solid-State Circuits Conference (ISSCC). Minhao, Yang Chung-Heng, Yeh Yiyin, Zhou Cerqueira, J. P. Lazar, A. A. Mingoo, Seok: IEEE, pp. 346–8.

[20] Yang Minhao et al. “Design of an always-on deep neural network-based 1-W voice activity detector aided with a customized software model for analog feature extraction”. In: *IEEE Journal of Solid-State Circuits* 54.6 (2019), pp. 1764–77.

[21] R. Yasuhara et al. “Reliability Issues in Analog ReRAM Based Neural-network Processor”. In: *2019 IEEE International Reliability Physics Symposium (IRPS), 31 March-4 April 2019*. 2019 IEEE International Reliability Physics Symposium (IRPS). Yasuhara, R. Ono, T. Mochida, R. Muraoka, S. Kouno, K. Katayama, K. Hayata, Y. Nakayama, M. Suwa, H. Hayakawa, Y. Mikawa, T. Gohou, Y. Yoneda, S.: IEEE, 5 pp.

[22] D. Bankman et al. “An always-on 3.8J/86processor with all memory on chip in 28nm CMOS”. In: *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pp. 222–224. ISBN: 2376-8606.

[23] D. Miyashita et al. “A Neuromorphic Chip Optimized for Deep Learning and CMOS Technology With Time-Domain Analog and Digital Mixed-Signal Processing”. In: *IEEE Journal of Solid-State Circuits* 52.10 (2017), pp. 2679–2689.

[24] Decebal Constantin Mocanu et al. “Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science”. In: *Nature Communications* 9.1 (2018), p. 2383.

[25] A. Goel et al. “A Survey of Methods for Low-Power Deep Learning and Computer Vision”. In: *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*. 2020, pp. 1–6.

[26] Norman Jouppi et al. “In-Datacenter Performance Analysis of a Tensor Processing Unit”. In: *ACM SIGARCH Computer Architecture News* 45 (June 2017), pp. 1–12.

[27] Qiang Yu et al. “Constructing Accurate and Efficient Deep Spiking Neural Networks With Double-Threshold and Augmented Schemes”. In: *IEEE Transactions on Neural Networks and Learning Systems* (2021), 113.

[28] A. Jayaraj, I. Banerjee, and A. Sanyal. “Common-Source Amplifier Based Analog Artificial Neural Network Classifier”. In: *2019 IEEE International Symposium on Circuits and Systems (ISCAS), 26-29 May 2019*. 2019 IEEE International Symposium on Circuits and Systems (ISCAS). Jayaraj, A. Banerjee, I. Sanyal, A.: IEEE, 5 pp.

[29] P. Kinget and M. S. J. Steyaert. “A programmable analog cellular neural network CMOS chip for high speed image processing”. In: *IEEE Journal of Solid-State Circuits* 30.3 (1995), pp. 235–43.

[30] M. Valle, D. D. Caviglia, and G. M. Bisio. “An experimental analog VLSI neural network with on-chip back-propagation learning”. In: *Analog Integrated Circuits and Signal Processing* 9.3 (1996), pp. 231–45.

[31] Y. Maeda and T. Kusuhashi. “An analog neural network system with learning capability using simultaneous perturbation”. In: *IEICE Transactions on Information and Systems* E82-D.12 (1999), pp. 1627–33.

[32] Dong Puxuan, G. L. Bilbro, and Chow Mo-Yuen. “Implementation of artificial neural network for real time applications using field programmable analog arrays”. In: *2006 International Joint Conference on Neural Networks, 16-21 July 2006*. 2006 International Joint Conference on Neural Networks. Puxuan, Dong Bilbro, G. L. Mo-Yuen, Chow: IEEE, pp. 1518–24.

[33] B. Heruseto et al. “Embedded analog CMOS Neural Network inside high speed camera”. In: *2009 1st Asia Symposium on Quality Electronic Design (ASQED 2009), 15-16 July 2009*. 2009 1st Asia Symposium on Quality Electronic Design (ASQED 2009). Heruseto, B. Prasetyo, E. Afandi, H. Paindavoine, M.: IEEE, pp. 144–7.

[34] R. Dlugosz, T. Talaska, and W. Pedrycz. “Current-mode Analog Adaptive Mechanism for Ultra-Low-Power Neural Networks”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 58.1 (2011), pp. 31–5.

[35] B. Larras et al. “Analog encoded neural network for power management in MPSoC”. In: *Analog Integrated Circuits and Signal Processing* 81.3 (2014), pp. 595–605.

[36] J. Binas et al. “Precise deep neural network computation on imprecise low-power analog hardware”. In: *arXiv* (2016), 21 pp.

[37] E. Rosenthal et al. “A fully analog memristor-based neural network with online gradient training”. In: *2016 IEEE International Symposium on Circuits and Systems (ISCAS), 22-25 May 2016*. 2016 IEEE International Symposium on Circuits and Systems (ISCAS). Rosenthal, E. Greshnikov, S. Soudry, D. Kvatinsky, S.: IEEE, pp. 1394–7.

[38] Moon Suhong, Shin Kwanghyun, and Jeon Dongsuk. “Enhancing Reliability of Analog Neural Network Processors”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.6 (2019), pp. 1455–9.

[39] Stefano Ambrogio et al. “Equivalent-accuracy accelerated neural-network training using analogue memory”. In: *Nature* 558.7708 (2018), pp. 60–67.

[40] A. J. Montalvo, R. S. Gyurcsik, and J. J. Paulos. “Building blocks for a temperature-compensated analog VLSI neural network with on-chip learning”. In: *Proceedings of IEEE International Symposium on Circuits and Systems - ISCAS ’94, 30 May-2 June 1994*. Vol. vol.6. 1994 IEEE International Symposium on Circuits and Systems (Cat. No.94CH3435-5). Montalvo, A. J. Gyurcsik, R. S. Paulos, J. J.: IEEE, pp. 363–6.

[41] “On-Chip Resistors and Capacitors”. In: *Adaptive Techniques for Mixed Signal System on Chip*. Boston, MA: Springer US, 2006, pp. 67–94. ISBN: 978-0-387-32155-4.

[42] S. Agarwal et al. “Designing and Modeling Analog Neural Network Training Accelerators”. In: *2019 International Symposium on VLSI Technology, Systems and Application (VLSI-TSA), 22-25 April 2019*. 2019 International Symposium on VLSI Technology, Systems and Application (VLSI-TSA). Agarwal, S. Jacobs-Gedrim, R. B. Bennett, C. Hsia, A. Van Heukelom, M. S. Hughart, D. Fuller, E. Yiyang, Li Talin, A. A. Marinella, M. J.: IEEE, 2 pp.

[43] T. Lehmann, E. Bruun, and C. Dietrich. “Mixed analog/digital matrix-vector multiplier for neural network synapses”. In: *12th NORCHIP Seminar, 8-9 Nov. 1994*. Vol. 9. Analog Integr. Circuits Signal Process. (Netherlands). Lehmann, T. Bruun, E. Dietrich, C.: Kluwer Academic Publishers, pp. 55–63. ISBN: 0925-1030.

[44] A. J. Montalvo, R. S. Gyurcsik, and J. J. Paulos. “An analog VLSI neural network with on-chip perturbation learning”. In: *IEEE Journal of Solid-State Circuits* 32.4 (1997), pp. 535–43.

[45] T. Yamasaki and T. Shibata. “Analog soft-pattern-matching classifier using floating-gate MOS technology”. In: *IEEE Transactions on Neural Networks* 14.5 (2003), pp. 1257–65.

[46] J. Fieres, K. Meier, and J. Schemmel. “A convolutional neural network tolerant of synaptic faults for low-power analog hardware”. In: *Artificial Neural Networks in Pattern Recognition. Second IAPR Workshop, ANNPR 2006. Proceedings, 31 Aug.-2 Sept. 2006*. Artificial Neural Networks in Pattern Recognition. Second IAPR Workshop, ANNPR 2006. Proceedings (Lecture Notes in Artificial Intelligence Vol. 4087). Fieres, J. Meier, K. Schemmel, J.: Springer-Verlag, pp. 122–32.

[47] Sun Zhuoli, Kang Kyunghee, and T. Shibata. “A self-learning multiple-class classifier using multi-dimensional quasi-Gaussian analog circuits”. In: *2010 IEEE International Symposium on Circuits and Systems. ISCAS 2010, 30 May-2 June 2010*. 2010 IEEE International Symposium on Circuits and Systems. ISCAS 2010. Zhuoli, Sun Kyunghee, Kang Shibata, T.: IEEE, pp. 2330–3.

[48] N. Rajeswaran and T. Madhu. “An analog Very Large Scale Integrated circuit design of Back Propagation Neural Networks”. In: *2016 World Automation Congress (WAC), 31 July-4 Aug. 2016*. 2016 World Automation Congress (WAC). Rajeswaran, N. Madhu, T.: IEEE, 4 pp.

[49] Hsieh Hung-Yi, Li Pin-Yi, and Tang Kea-Tiong. “An Analog Probabilistic Spiking Neural Network with On-Chip Learning”. In: *Neural Information Processing. 24th International Conference, ICONIP 2017, 14-18 Nov. 2017*. Vol. pt.VI. Neural Information Processing. 24th International Conference, ICONIP 2017. Proceedings: LNCS 10639. Springer International Publishing, pp. 777–85.

[50] Jia Kaige et al. “Calibrating Process Variation at System Level with In-Situ Low-Precision Transfer Learning for Analog Neural Network Processors”. In: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), 24-28 June 2018*. 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC). Proceedings. Kaige, Jia Zheyu, Liu Qi, Wei Fei, Qiao Xinjun, Liu Yi, Yang Hua, Fan Huazhong, Yang: IEEE, 6 pp.

[51] D. Rancour and H. Michel. “Self-trained multi-layer analog real-time artificial neural network circuits”. In: *2018 International Conference on Artificial Intelligence, ICAI 2018 at 2018 World Congress in Computer Science, Computer Engineering and Applied Computing, CSCE 2018, July 30, 2018 - August 2, 2018*. 2018 World Congress in Computer Science, Computer Engineering and Applied Computing, CSCE 2018 - Proceedings of the 2018 International Conference on Artificial Intelligence, ICAI 2018. CSREA Press, pp. 84–89.

[52] N. Dey et al. “On-chip learning in a conventional silicon MOSFET based Analog Hardware Neural Network [arXiv]”. In: *arXiv* (2019), 18 pp.

[53] E. Gatt, J. Micallef, and E. Chilton. “An analog VLSI time-delay neural network implementation for phoneme recognition”. In: *Proceedings of the 2000 6th IEEE International Workshop on Cellular Neural Networks and their Applications (CNNA 2000), 23-25 May 2000*. Proceedings of the 2000 6th IEEE International Workshop on Cellular Neural Networks

and their Applications (CNNA 2000) (Cat. No.00TH8509). Gatt, E. Micallef, J. Chilton, E.: IEEE, pp. 315–20.

[54] O. Krestinskaya, K. N. Salama, and A. P. James. “Learning in Memristive Neural Network Architectures Using Analog Backpropagation Circuits”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 66.2 (2019), pp. 719–32.

[55] A. J. Montalvo, R. S. Gyurcsik, and J. J. Paulos. “Toward a general-purpose analog VLSI neural network with on-chip learning”. In: *IEEE Transactions on Neural Networks* 8.2 (1997), pp. 413–23.

[56] V. F. Koosh and R. Goodman. “VLSI neural network with digital weights and analog multipliers”. In: *ISCAS 2001. The 2001 IEEE International Symposium on Circuits and Systems, 6-9 May 2001*. Vol. vol. 3. ISCAS 2001. The 2001 IEEE International Symposium on Circuits and Systems (Cat. No.01CH37196). Koosh, V. F. Goodman, R.: IEEE, pp. 233–6.

[57] A. Shafiee et al. “ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars”. In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 18-22 June 2016*. 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). Shafiee, A. Nag, A. Muralimanohar, N. Balasubramonian, R. Strachan, J. P. Miao, Hu Williams, R. S. Srikumar, V.: IEEE Computer Society, pp. 14–26.

[58] G. Geske, F. Stupmann, and A. Wego. “High speed color recognition with an analog neural network chip”. In: *2003 IEEE International Conference on Industrial Technology, 10-12 Dec. 2003*. Vol. Vol.1. 2003 IEEE International Conference on Industrial Technology (IEEE Cat. No.03TH8685). Geske, G. Stupmann, F. Wego, A.: IEEE, pp. 104–7.

[59] D. Maliuk et al. “Analog neural network design for RF built-in self-test”. In: *2010 IEEE International Test Conference (ITC 2010), 31 Oct.-5 Nov. 2010*. Proceedings 2010 IEEE International Test Conference (ITC 2010). Maliuk, D. Stratigopoulos, H. G. He, Huang Makris, Y.: IEEE Computer Society, 10 pp.

[60] H. Abdelbaki, E. Gelenbe, and S. E. El-Khamy. “Analog hardware implementation of the random neural network model”. In: *Proceedings of IEEE-INNS-ENNS International Joint Conference on Neural Networks, 24-27 July 2000*. Vol. vol.4. Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium. Abdelbaki, H. Gelenbe, E. El-Khamy, S. E.: IEEE Comput. Soc., pp. 197–201.

[61] T. Marukame et al. “Proposal, analysis and demonstration of Analog/Digital-mixed Neural Networks based on memristive device arrays”. In: *2018 IEEE International Symposium on Circuits and Systems (ISCAS), 27-30 May 2018*. 2018 IEEE International Symposium on Circuits and Systems (ISCAS). Marukame, T. Nomura, K. Matusmoto, M. Takaya, S. Nishi, Y.: IEEE, 5 pp.

[62] A. Martinez-Nieto et al. “An accurate analysis method for complex IC analog neural network-based systems using high-level software tools”. In: *2018 IEEE 9th Latin American Symposium on Circuits amp; Systems (LASCAS), 25-28 Feb. 2018*. 2018 IEEE 9th Latin American Symposium on Circuits Systems (LASCAS). Proceedings. Martinez-Nieto, A. Medrano, N. Sanz-Pascual, M. T. Calvo, B.: IEEE, 4 pp.

[63] Hua Ruobing and A. Sanyal. “39fJ analog artificial neural network for breast cancer classification in 65nm CMOS”. In: *2019 IEEE 62nd International Midwest Symposium on Circuits and Systems (MWSCAS), 4-7 Aug. 2019*. 2019 IEEE 62nd International Midwest Symposium on Circuits and Systems (MWSCAS). IEEE, pp. 436–9.

[64] Y. Berg et al. “An analog feed-forward neural network with on-chip learning”. In: *12th NORCHIP Seminar, 8-9 Nov. 1994*. Vol. 9. Analog Integr. Circuits Signal Process. (Netherlands). Berg, Y. Sigvartsen, R. L. Lande, T. S. Abusland, A.: Kluwer Academic Publishers, pp. 65–75. ISBN: 0925-1030.

[65] R. C. Chang et al. “Programmable-weight building blocks for analog VLSI neural network processors”. In: *Analog Integrated Circuits and Signal Processing* 9.3 (1996), pp. 215–30.

[66] L. Gatet, H. Tap-Beteille, and M. Lescure. “Design and test of a CMOS MLP analog neural network for fast on-board signal processing”. In: *13th IEEE International Conference on Electronics, Circuits and Systems, 10-13 Dec. 2006*. 13th IEEE International Conference on Electronics, Circuits and Systems. Gatet, L. Tap-Beteille, H. Lescure, M.: IEEE, pp. 922–5.

[67] Pan Chih-Heng, Hsieh Hung-Yi, and Tang Kea-Tiong. “An analog multilayer perceptron neural network for a portable electronic nose”. In: *Sensors* 13.1 (2013), pp. 193–207.

[68] G. E. Hinton et al. “Improving neural networks by preventing co-adaptation of feature detectors”. In: *ArXiv* abs/1207.0580 (2012).

[69] N. Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *J. Mach. Learn. Res.* 15.1 (Jan. 2014), 19291958.

[70] R. D. Reed and R. J. Marks. *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. Cambridge, MA, USA: MIT Press, 1998. ISBN: 0262181908.

[71] H. Hofmann, K. Kafadar, and H. Wickham. “Letter-value plots: Boxplots for large data”. In: *The American Statistican* (2011).

[72] S. Ackerman. *Discovering the Brain*. Ed. by Sandra Ackerman. ISBN: 978-0-309-46799-5.

[73] Markus Quade et al. “Prediction of Dynamical Systems by Symbolic Regression”. In: *Physical Review E* 94 (Feb. 2016).

[74] K. Gregor and Y. LeCun. “Learning Fast Approximations of Sparse Coding”. In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML’10. Haifa, Israel: Omnipress, 2010, 399406. ISBN: 9781605589077.

[75] Sarah Harris and David Harris. *Digital Design and Computer Architecture: ARM Edition*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2015. ISBN: 0128000562.

[76] S. Weinberger. “Speech Accent Archive”. In: *George Mason University*. URL: http:// accent.gmu.edu Accessed 05 Aug 2019, 2015.

[77] Chandan KA Reddy et al. “ICASSP 2021 Deep Noise Suppression Challenge”. In: *arXiv preprint arXiv:2009.06122* (2020).

[78] “Breast Cancer Diagnosis via Linear Programming”. In: *IEEE Computational Science and Engineering* 2.3 (1995), p. 70.

[79] V. Arzamasov, K. Böhm, and P. Jochem. “Towards Concise Models of Grid Stability”. In: *2018 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*, pp. 1–6.

[80] Yann LeCun, Corinna Cortes, and CJ Burges. “MNIST handwritten digit database”. In: *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist* 2 (2010).