EXPLORING COMPLEXITY REDUCTION IN DEEP LEARNING

by

Sourya Dey

A Dissertation Presented to the
FACULTY OF THE USC GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the Requirements for the Degree
DOCTOR OF PHILOSOPHY
(ELECTRICAL ENGINEERING)

August 2020

Copyright 2020 Sourya Dey

Let yourself be silently drawn by the strange pull of what you really love.
–Rumi

Acknowledgements

Pursuing a Ph.D. is a beautiful thing since it allows one to be free to explore uncharted territory instead of being burdened by constraints of banality. In this regard, I would like to thank my co-advisors – Professors Peter Beerel and Keith Chugg – without whose help and guidance I would not have been able to explore the research directions that have led to the material in this dissertation.

I am grateful to past and present members of my research team who have helped and collaborated with me in my research. Special thanks to my primary collaborators – Kuan-Wen Huang, Yinan Shao, Diandian Chen, and Saikrishna Chaitanya Kanala. I also acknowledge the contribution of secondary collaborators who have since moved on – Zongyang Li, Saad Zafar, Mahdi Jelodari Mamaghani, Zheng Wu, Jiajin Xi, Venkata Nishanth Narisetty, and Kiran Nagendra; and current research team members whose constant feedback has been invaluable – Souvik Kundu, Prof. Leana Golubchik, Marco Paolieri, Arnab Sanyal, and Andrew Schmidt. I am also indebted to Prof. Panayiotis Georgiou, Prof. Mahdi Soltanolkotabi, Prof. Antonio Ortega, and Mohammed Nasir for help in specific efforts.

I would like to thank the agencies that have funded my research and helped to pay the bills – National Science Foundation Software and Hardware Foundations (NSF SHF) Grant 1763747, and Defense Threat Reduction Agency (DTRA) in association with the Scalable Acceleration Platform Integrating Reconfigurable Computing and Natural Language Processing Technologies (SAPIENT) team and University of Southern California Information Sciences Institute (USC ISI). I am grateful to Diane Demetras and Annie Yu for their help in administrative matters related to the progress of my Ph.D. and presentation of my research to the outside world.

Heartfelt thanks to my family members for their constant love and support, despite being located halfway around the world. Nothing would have been possible without them.

Finally, thanks to *you* the reader for picking up this dissertation detailing my research efforts since spring 2016. I hope you have as much enjoyment reading it as I had writing it.

Author's note: This dissertation is being completed while the COVID-19 pandemic is gripping the world, and the US has been hit particularly hard. These are interesting times we live in.

Table of Contents

List of Tables

List of Figures

List of Acronyms

AI Artificial Intelligence

ASIC Application Specific Integrated Circuit

ASR Automatic Speech Recognition

AutoML Automated Machine Learning

BN Batch Normalization

BO Bayesian Optimization

BRAM Block Random Access Memory

CI Confidence Interval

CNN Convolutional Neural Network

CPU Central Processing Unit

DnC Deep-n-Cheap

DSP Digital Signal Processing

FC Fully Connected

FLOPS Floating Point Operations per Second

FMNIST Fashion MNIST

FPGA Field Programmable Gate Array

gcd Greatest Common Divisor

GPU Graphics Processing Unit

LDPC Low Density Parity Check

LED Light Emitting Diode

LUT Look Up Table

MFCC Mel-frequency Cepstral Coefficient

MLP Multilayer Perceptron

NAS Neural Architecture Search

NN Neural network

PCA Principal Component Analysis

ReLU Rectified Linear Unit

RNN Recurrent Neural Network

RTL Register Transfer Level

SGD Stochastic Gradient Descent

TPC Test Prediction Comparison

UART Universal asynchronous receiver-transmitter

Abstract

Deep learning has become a powerful tool for cutting-edge machine learning and artificial intelligence applications such as data classification and computer vision. Deep learning uses neural networks comprising many trainable parameters, which gives rise to significant computational complexity, particularly during their training phase. Such complexity is often prohibitively large given the available time, financial and computational resources, and thereby restricts the study and usage of deep learning systems.

This dissertation makes two major contributions towards reducing complexity of neural networks in deep learning. Firstly, we propose *pre-defined sparsity* – a technique to reduce computational complexity during both training and inference phases. We analyze the resulting sparse connection patterns in an attempt to understand network performance, and introduce a proof-of-concept hardware architecture leveraging sparsity to achieve on-device training and inference. Secondly, we introduce *Deep-n-Cheap* – an automated machine learning framework targeted towards training complexity reduction. The framework is open-source

and applicable to a wide range of datasets and types of neural networks. As an additional third contribution, we engineer a family of synthetic datasets of algorithmically customizable difficulty for benchmarking neural networks and machine learning methodologies.

Related Publications and Software

Publications

Note that the citation counts in the URLs linked below may be inaccurate. Please see the author's Google Scholar page for a full list of citations.

S. Dey, K. W. Huang, P. A. Beerel and K. M. Chugg, "Pre-defined Sparse Neural Networks with Hardware Acceleration," in *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 332–345, Jun 2019.
IEEE: https://ieeexplore.ieee.org/document/8689061

S. Dey, S. C. Kanala, K. M. Chugg and P. A. Beerel, "Deep-n-Cheap: An Automated Search Framework for Low Complexity Deep Learning," under review at *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML-PKDD)*, 2020.
arXiv pre-print: https://arxiv.org/abs/2004.00974

[Awarded Best Paper] **S. Dey**, K. M. Chugg and P. A. Beerel, "Morse Code Datasets for Machine Learning," in *9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pp. 1-7, Jul 2018.
IEEE: https://ieeexplore.ieee.org/document/8494011

S. Dey, Y. Shao, K. M. Chugg and P. A. Beerel, "Accelerating training of deep neural networks via sparse edge processing," in *26th International Conference on Artificial Neural Networks (ICANN)* Part 1, pp. 273–280, Springer, 2017.
Springer: https://link.springer.com/chapter/10.1007/978-3-319-68600-4_32

S. Dey, P. A. Beerel and K. M. Chugg, "Interleaver design for deep neural networks," in *51st Annual Asilomar Conference on Signals, Systems, and Computers (ACSSC)*, pp. 1979–1983, Oct 2017.
IEEE: https://ieeexplore.ieee.org/document/8335713

S. Dey, K. W. Huang, P. A. Beerel and K. M. Chugg, "Characterizing sparse

connectivity patterns in neural networks," in *2018 Information Theory and Applications Workshop (ITA)*, pp. 1–8, Feb 2018.
IEEE: https://ieeexplore.ieee.org/document/8502950

S. Dey, D. Chen, Z. Li, S. Kundu, K. W. Huang, K. M. Chugg and P. A. Beerel, "A Highly Parallel FPGA Implementation of Sparse Neural Network Training," in *2018 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pp. 1–4, Dec 2018. Expanded preprint version available on arXiv.
IEEE: https://ieeexplore.ieee.org/document/8641739

Software

Please see the author's Github page for additional information.

https://github.com/souryadey/deep-n-cheap
Deep-n-Cheap: AutoML framework balancing performance and complexity
v1.0 released on April 2, 2020

https://github.com/souryadey/predefinedsparse-nnets
Pre-defined sparse neural networks

https://github.com/souryadey/morse-dataset
Morse code datasets for training artificial neural networks

Chapter 1

Introduction

1.1 Neural networks

Neural networks (NNs) have proven to be ubiquitous in modern machine learning and Artificial Intelligence (AI) applications such as object classification [1], self-driving cars [2], Automatic Speech Recognition (ASR) [3], and so on. Such NNs are usually deep, implying that there are hidden layers between the input (such as a picture of a cat), and the output (such as a probability distribution indicating what animal the picture is). The depth of a NN can vary from just one or two hidden layers, as is common for Multilayer Perceptrons (MLPs) [4], to several hundred for Convolutional Neural Networks (CNNs) [5].

1.1.1 Complexity of neural networks

NNs used for classification typically rely on large amounts of labeled data to train – a process during which the numerical values of its internal parameters are tuned to facilitate better inference – measured as classification performance on unseen data. As more data have become available, the size and complexity of NNs have

grown sharply, with modern NNs containing millions [1] or even billions of trainable parameters [6]. These massive NNs come with the cost of large computational and storage demands. The current state of the art is to train large NNs on Graphics Processing Units (GPUs) in the cloud – a process that can take days to weeks even on powerful GPUs [1, 6, 7] or similar programmable processors with multiply-accumulate accelerators [8]. This translates to a large financial impact when deploying NNs, cf. [9–11]. Therefore, several prominent researchers [12] have identified complexity reduction as a key step to NN acceleration, which is a general term referring to speeding up the deployment and operation of a NN. Acceleration is generally performed post-training to reduce complexity of inference only, *e.g.* methods for quantization, compression, and grouping parameters [13–16]. However, the training complexity of NNs remains a major bottleneck.

1.1.2 Automated Machine Learning

There are a number of choices which play a crucial role during the deployment of NNs to solve specific problem(s). Some of these are the number of layers, the structure of the layers, how fast the network should learn, and so on. These form the domain of Automated Machine Learning (AutoML). Unfortunately, the design of NNs is a largely empirical process and there is no clear understanding regarding, for example, how many layers and neurons are appropriate for classifying a picture of a handwritten digit into one out of ten classes. Existing efforts in

AutoML can broadly be classified into two categories – a) AutoML frameworks such as [17–19] that are generalized software packages to design NNs, and b) novel search techniques geared towards specific problems [20–24]. The former category does not focus on complexity, often resulting in extremely long search costs and training times, while the latter category is not generalized for different datasets or geared towards training complexity reduction. For example, the final model found in [24] takes 3 days to train on the small CIFAR-10 dataset. Thus, the issue of training NNs being a major bottleneck is reinforced.

1.2 Dissertation Contributions

The contributions of this dissertation are broadly divided into three categories – a) pre-defined sparsity for complexity reduction, b) Deep-n-Cheap – a customizable AutoML framework trading off performance with complexity, and c) a family of synthetic datasets for machine learning applications.

1.2.1 Pre-defined sparsity

1. We proposed the novel technique of **pre-defined sparsity** – a method to reduce the complexity of NNs during both training and inference phases. This technique applies to MLPs or the MLP portion of NNs. In particular, our results show that MLP complexity, both in terms of the number of parameters

(storage) and the number of operations to be performed (computation), can be reduced by factors greater than 5X without significant performance loss [25–27].

2. We analyzed **trends and design guidelines** for selecting a pre-defined sparse MLP [25, 26]. These studies help to accelerate the search for good pre-defined sparse MLPs given any problem.

3. We proposed a **hardware architecture** to leverage the benefits of pre-defined sparsity [25, 27]. The architecture is flexible in the sense that the complexity of the network to be implemented is decoupled from the available hardware resources on the given device (such as Field Programmable Gate Array (FPGA)(s)). Thus, MLPs of varying sizes can be supported on various hardware platforms. Moreover, the architecture supports both training and inference phases of a MLP. To the best of our knowledge, we are the first to propose such a flexible hardware architecture with the potential to accelerate both training and inference.

4. We developed a working **FPGA implementation** of the hardware architecture [28]. This serves as a proof-of-concept of our ideas.

5. We proposed and analyzed the properties of a class of pre-defined sparse MLP **connection patterns** which are suited to our hardware architecture in the sense that they allow maximum throughput [25, 29]. In particular, we

showed that such hardware-friendly sparse patterns result in no performance degradation compared to other sparse patterns.

6. We developed a metric called *scatter* to characterize the 'goodness' of a connection pattern in order to **predict NN performance prior to training**. This is helpful in filtering out bad connection patterns without incurring the computational expenses of training.

Our work on pre-defined sparsity can be found on Github [30].

1.2.2 Automated Machine Learning

7. We have developed **Deep-n-Cheap** – an open-source AutoML framework which, to the best of our knowledge, is the first to target training complexity reduction in terms of both time and storage. Deep-n-Cheap searches over NN architectures and training settings and, at present, supports both CNNs and MLPs.

8. We have developed Deep-n-Cheap in a way such that it offers users **extensive customizability** options so as to be usable for both benchmark and custom datasets.

9. We introduced the novel *ramp distance* to characterize the **similarity between different NN configurations**.

10. We conducted investigations and drew various **insights from our AutoML work** such as *search transfer* for generalizing NN architectures across different architectures.

Our AutoML framework Deep-n-Cheap can be found on Github [31].

1.2.3 Dataset Engineering

11. We have created a **family of synthetic datasets** on classification of Morse code symbols [32]. Owing to their algorithmic generation capability, it is simple to tune the parameters of these datasets. This results in a large number of classification problems of varying difficulty, which can be used to benchmark different NNs and other machine learning algorithms. This work resulted in a 'Conference Best Paper' award.

12. We developed several **metrics to indicate the difficulty of a dataset** and evaluated their merits.

The Morse datasets are open-source and available on Github [33] and IEEE Data Port [34].

1.3 Dissertation Organization

This dissertation is organized as follows. Chapter 2 contains background necessary for understanding NNs and the work in this dissertation. Chapter 3 introduces

pre-defined sparsity, along with trends and guidelines for selecting connection patterns. Chapter 4 discusses our hardware architecture and its FPGA implementation. Chapter 5 analyzes sparse connection patterns, and metrics to predict NN performance. Chapter 6 discusses our efforts in creating a family of synthetic datasets, along with metrics to characterize their difficulty. Chapter 7 discusses Deep-n-Cheap, and associated contributions of our AutoML work. Finally, Chapter 8 concludes this work.

Chapter 2

Background

In this chapter, we provide the background necessary for understanding the concepts and terms used in this work. We will begin with mathematical notation, then move on to notation and operations specific to neural networks.

2.1 Mathematical Notation

We will use the numerator layout convention for matrix calculus. A complete set of rules for this convention is given in [35]. We summarize some of the key rules below:

- Vectors are written as lower case bold letters, such as $\boldsymbol{x}$, and can be either row (dimensions $1 \times n$) or column (dimensions $n \times 1$). Column vectors are the default choice, unless otherwise mentioned. Individual elements are indexed by bracketed superscripts, *e.g.* $x^{(i)}$, where $i \in \{1, \cdots, n\}$.

- Matrices are written as upper case bold letters, such as $\boldsymbol{X}$. A matrix with dimensions $m \times n$ corresponds to m rows and n columns. Individual elements are indexed by bracketed double superscripts for row and column, respectively, *e.g.* $X^{(i,j)}$, where $i \in \{1, \cdots, m\}$, $j \in \{1, \cdots, n\}$.

- The **derivative** of f with respect to x is $\dfrac{\partial f}{\partial x}$, where both x and f can be scalars, vectors, or matrices. The **gradient** of f with respect to x is $\nabla_x f = \left(\dfrac{\partial f}{\partial x}\right)^T$, *i.e. gradient is transpose of derivative.* The derivative is important because it obeys the chain rule of calculus and helps to derive several results. The gradient is important because it is the form used in updating NN parameters, as will be described in Section 2.2.3.

Some relevant forms of derivatives and gradients are described next.

Scalar-by-scalar

Both f and x are scalars. We do not define the gradient in this case. The derivative is the scalar $\dfrac{\partial f}{\partial x}$, also written as f'.

Scalar-by-vector

f is a scalar, $\boldsymbol{x}$ is a vector of dimensions $n \times 1$. Then the derivative is a $1 \times n$ row vector:

$$\frac{\partial f}{\partial \boldsymbol{x}} = \begin{bmatrix} \dfrac{\partial f}{\partial x^{(1)}} & \dfrac{\partial f}{\partial x^{(2)}} & \cdots & \dfrac{\partial f}{\partial x^{(n)}} \end{bmatrix} \tag{2.1}$$

and the gradient $\boldsymbol{\nabla}_{\boldsymbol{x}} f$ is its transposed column vector.

Vector-by-vector of equal size

Both $\boldsymbol{f}$ and $\boldsymbol{x}$ are vectors of dimensions $n \times 1$. Then the derivative is the Jacobian matrix of dimensions $n \times n$:

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial f^{(1)}}{\partial x^{(1)}} & \cdots & \frac{\partial f^{(1)}}{\partial x^{(n)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f^{(n)}}{\partial x^{(1)}} & \cdots & \frac{\partial f^{(n)}}{\partial x^{(n)}} \end{bmatrix} \tag{2.2}$$

Vectorized scalar function

This is a scalar-to-scalar function applied element-wise to a vector, *e.g.* :

$$f\left(\begin{bmatrix} x^{(1)} \\ x^{(2)} \\ \vdots \\ x^{(n)} \end{bmatrix}\right) = \begin{bmatrix} f\left(x^{(1)}\right) \\ f\left(x^{(2)}\right) \\ \vdots \\ f\left(x^{(n)}\right) \end{bmatrix} \tag{2.3}$$

In this case, both the derivative and gradient are the same $n \times n$ diagonal matrix, given as:

$$\boldsymbol{\nabla}_{\boldsymbol{x}} \boldsymbol{f} = \begin{bmatrix} f'\left(x^{(1)}\right) & & & 0 \\ & f'\left(x^{(2)}\right) & & \\ & & \ddots & \\ 0 & & & f'\left(x^{(n)}\right) \end{bmatrix} \tag{2.4}$$

An equivalent form of this is to take the diagonal and express it as a $n \times 1$ vector:

$$\boldsymbol{f}' = \begin{bmatrix} f'\left(x^{(1)}\right) \\ f'\left(x^{(2)}\right) \\ \vdots \\ f'\left(x^{(n)}\right) \end{bmatrix} \tag{2.5}$$

To realize the effect of this equivalent form, let's say we want to multiply the gradient $\boldsymbol{\nabla}_{\boldsymbol{x}} \boldsymbol{f}$ from (2.4) with some n-dimensional (column) vector $\boldsymbol{a}$. Achieving

the same result with $\boldsymbol{f}'$ from (2.5) would require the *Hadamard product* $\circ$, defined as element-wise multiplication of 2 vectors:

$$(\boldsymbol{\nabla}_{\boldsymbol{x}}\boldsymbol{f})\,\boldsymbol{a} = \boldsymbol{f}'(\boldsymbol{x}) \circ \boldsymbol{a} = \begin{bmatrix} f'\left(x^{(1)}\right) a^{(1)} \\ f'\left(x^{(2)}\right) a^{(2)} \\ \vdots \\ f'\left(x^{(n)}\right) a^{(n)} \end{bmatrix} \tag{2.6}$$

2.2 Notation and Basic Operations for Neural Networks

A NN, sometimes referred to as an Artificial Neural Network to distinguish it from biological neural networks found in living organisms, is an interconnected set of **nodes, or neurons**, with the capability to learn mathematical representations of the data fed to it. This ability to learn is referred to as **training**. Once a NN is trained, it may be used for **inference**, where it is operated on new data different from what it was trained on, and its performance based on some metric(s) is noted.

We will initially describe notation for a **Multilayer Perceptron (MLP)**, which is a commonly used NN for classification problems, and assume **supervised learning**, *i.e.* where labels are provided for different training samples and the NN is asked to learn from them.

A set of nodes is referred to as a **layer**. An $(L+1)$-layer MLP has N_i nodes in the i^{th} layer, described collectively by the **neuronal configuration** $\boldsymbol{N}_{\text{net}} = (N_0, N_1, \cdots, N_L)$,[1] where layer 0 is the input layer. We use the convention that layer i is to the 'right' of layer $i-1$, or 'next' to layer $i-1$.

There are L **junctions** between layers, with junction i, $i \in \{1, 2, \cdots, L\}$, connecting the N_{i-1} nodes of its left layer $i-1$ with the N_i nodes of its right layer i. For a **Fully Connected (FC)** MLP, all the nodes in a layer connect to all the nodes in its next and previous layers, if present. These connections are defined using **edges**, which have associated **weight** values. Weights in junction i are collected as a matrix $\underset{N_i \times N_{i-1}}{\boldsymbol{W}_i}$. Note that the rows (1st index) refer to the next layer, and the columns (2nd index) refer to the previous layer. This is to facilitate matrix-vector multiplications, as will be seen in (2.7a). Additionally, individual nodes in all layers except for the input have **bias** values. For layer i, the biases are collected as a vector $\underset{N_i \times 1}{\boldsymbol{b}_i}$.

Note that we denote layer or junction number via subscript. This is why we denote individual elements of matrices and vectors as bracketed superscripts, as shown in Section 2.1. For example $b_2^{(3)}$ refers to the bias of the 3rd neuron in layer 2, and $W_1^{(4,5)}$ refers to the weight of the edge connecting the 5th node in layer 0 to

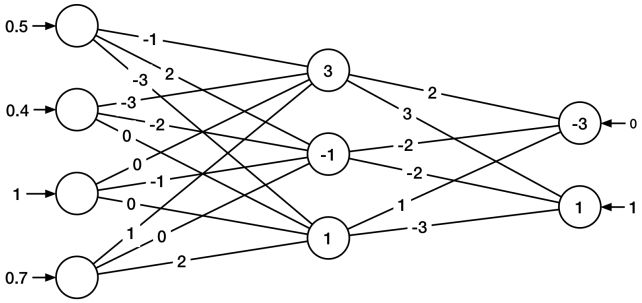
[1] We make a distinction between an ordered tuple of numbers providing information, such as $\boldsymbol{N}_{\text{net}}$, and a vector, such as $\boldsymbol{x}$. The former is not meant to be used in algebraic calculations, hence its individual elements are written with round brackets around them. The latter is used in algebra and hence its individual elements are written with the more conventional square brackets around them, as in the equations presented so far in this chapter.

the 4th node in layer 1. This notation suits us since throughout this work we will need to reference layer numbers far more than individual weight or bias values, and hence benefit from using subscripts for layer number instead of the somewhat inconvenient bracketed superscripts.

Weights and biases are all **trainable parameters**, which means that their values change during training. This is ideally done until the values converge, *i.e.* the change in values obtained by training on more data is negligible. In practical scenarios, training is often stopped due to time or computational constraints instead of waiting for convergence.

The processes of training and inference are affected by some values and decisions which the network does not learn, instead, they are adjusted by an entity external to the NN such as a human. These values are known as **hyperparameters**, and describe quantities such as how many layers the network should have, how many inputs should be trained on in parallel, and so on. The difficulty of designing and adjusting hyperparameter values has led to a new branch of study known as **automated machine learning (AutoML)**, which will be expounded in Chapter 7.

Next we describe the three operations of a NN. These are summarized in Fig. 2.1.



$$\boldsymbol{a}_1 = \mathrm{ReLU}\left(\begin{bmatrix} -1 & -3 & 0 & 1 \\ 2 & -2 & -1 & 0 \\ -3 & 0 & 0 & 2 \end{bmatrix}\begin{bmatrix} 0.5 \\ 0.4 \\ 1 \\ 0.7 \end{bmatrix} + \begin{bmatrix} 3 \\ -1 \\ 1 \end{bmatrix}\right) = \begin{bmatrix} 2 \\ 0 \\ 0.9 \end{bmatrix}$$

$$\boldsymbol{a}_2 = \mathrm{Softmax}\left(\begin{bmatrix} 2 & -2 & 1 \\ 3 & -2 & -3 \end{bmatrix}\begin{bmatrix} 2 \\ 0 \\ 0.9 \end{bmatrix} + \begin{bmatrix} -3 \\ 1 \end{bmatrix}\right) = \begin{bmatrix} 0.08 \\ 0.92 \end{bmatrix}$$

FF

$$\boldsymbol{\delta}_1 = \left(\begin{bmatrix} 2 & 3 \\ -2 & -2 \\ 1 & -3 \end{bmatrix}\begin{bmatrix} 0.08 \\ -0.08 \end{bmatrix}\right) \circ \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -0.08 \\ 0 \\ 0.32 \end{bmatrix} \qquad \boldsymbol{\delta}_2 = \begin{bmatrix} 0.08 \\ 0.92 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.08 \\ -0.08 \end{bmatrix}$$

BP

$$\boldsymbol{b}_1 \leftarrow \begin{bmatrix} 3 \\ -1 \\ 1 \end{bmatrix} - 0.6\begin{bmatrix} -0.08 \\ 0 \\ 0.32 \end{bmatrix} = \begin{bmatrix} 3.05 \\ -1 \\ 0.8 \end{bmatrix} \qquad \boldsymbol{b}_2 \leftarrow \begin{bmatrix} -3 \\ 1 \end{bmatrix} - 0.6\begin{bmatrix} 0.08 \\ -0.08 \end{bmatrix} = \begin{bmatrix} -3.05 \\ 1.05 \end{bmatrix}$$

$$\boldsymbol{W}_2 \leftarrow \begin{bmatrix} 2 & -2 & 1 \\ 3 & -2 & -3 \end{bmatrix} - 0.6\begin{bmatrix} 0.08 \\ -0.08 \end{bmatrix}\begin{bmatrix} 2 & 0 & 0.9 \end{bmatrix} = \begin{bmatrix} 1.9 & -2 & 0.96 \\ 3.1 & -2 & -2.96 \end{bmatrix}$$

$$\boldsymbol{W}_1 \leftarrow \begin{bmatrix} -1 & -3 & 0 & 1 \\ 2 & -2 & -1 & 0 \\ -3 & 0 & 0 & 2 \end{bmatrix} - 0.6\begin{bmatrix} -0.08 \\ 0 \\ 0.32 \end{bmatrix}\begin{bmatrix} 0.5 & 0.4 & 1 & 0.7 \end{bmatrix} = \begin{bmatrix} -1.02 & -3.02 & 0.05 & 1.03 \\ 2 & -2 & -1 & 0 \\ -3.1 & -0.08 & -0.19 & 1.87 \end{bmatrix}$$

UP

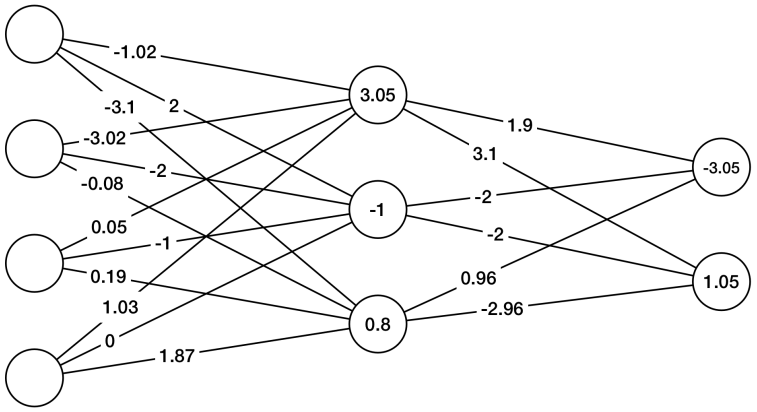


Figure 2.1: Complete training flow for a single input sample for a MLP with $\boldsymbol{N}_{\text{net}} = (4, 3, 2)$. (a) Original NN, presented with an input $\boldsymbol{a}_0 = [0.5, 0.4, 1, 0.7]^T$ and corresponding one-hot ground-truth labeling $\boldsymbol{y} = [0, 1]^T$. FF proceeds with ReLU and Softmax activations for junctions 1 and 2 respectively, BP uses cross-entropy cost, and UP uses $\eta = 0.6$. No regularization is applied and batch size $M = 1$. (b) After updating the weights and biases of the original NN.

2.2.1 Feedforward (FF)

This process starts by accepting an external input datum. Each input is represented by a vector of **features** $\underset{N_0 \times 1}{\boldsymbol{a}_0}$, and a ground truth labeling $\underset{N_L \times 1}{\boldsymbol{y}}$ denoting the ideal output layer values. Then, $\forall i \in \{1, 2, \cdots, L\}$, the FF operation proceeds as:

$$\underset{N_i \times 1}{\boldsymbol{s}_i} = \boldsymbol{W}_i \boldsymbol{a}_{i-1} + \boldsymbol{b}_i \tag{2.7a}$$

$$\underset{N_i \times 1}{\boldsymbol{a}_i} = \boldsymbol{h}\left(\boldsymbol{s}_i\right) \tag{2.7b}$$

$$\underset{N_i \times N_i}{\boldsymbol{H}'_i} = \frac{\partial \boldsymbol{a}_i}{\partial \boldsymbol{s}_i} \tag{2.7c}$$

where $\boldsymbol{s}$ is the linear output, $\boldsymbol{h}$ is the non-linear **activation function**, and $\boldsymbol{a}$ is the **activation** output. Note that (2.7c) is not strictly a part of FF, but it is sometimes computed since the derivative values are required for BP.

For most layers, $\boldsymbol{h}$ is a vectorized scalar function applied to $\boldsymbol{s}_i$. In such cases, $\boldsymbol{H}'_i$ is a diagonal matrix and can be reduced to the vector $\underset{N_i \times 1}{\boldsymbol{h}'_i}$, as shown in Section 2.1. One such common activation function which we will use frequently is **Rectified Linear Unit (ReLU)** [36], shown below in its scalar-to-scalar form:

$$\mathrm{ReLU}(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases} \tag{2.8a}$$

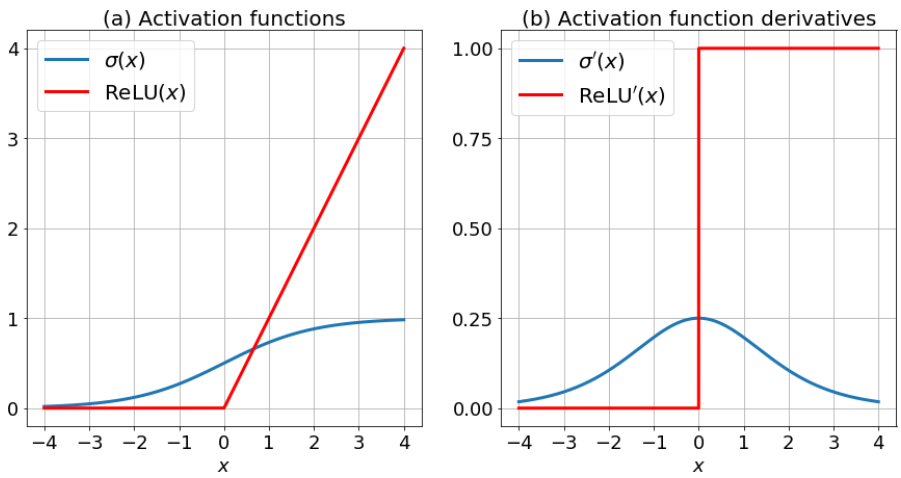


Figure 2.2: Comparison of ReLU and sigmoid (a) activations, and (b) activation derivatives.

$$\text{ReLU}'(x) = \begin{cases} 1, & x > 0 \\ [0,1], & x = 0 \\ 0, & x < 0 \end{cases} \tag{2.8b}$$

An alternative to ReLU is the sigmoid function given as:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2.9a}$$

$$\sigma'(x) = \sigma(x)(1-\sigma(x)) \tag{2.9b}$$

Fig. 2.2 illustrates ReLU and sigmoid.

For the last layer ($i = L$) however, $\boldsymbol{h}$ is usually the **softmax** function which outputs a probability distribution. This is a vector-to-vector function where each

element of the output vector depends on each element of the input vector, hence $\boldsymbol{H}'_L$ is not (necessarily) a diagonal matrix.

$$\boldsymbol{a}_L = \text{Softmax}\left(\boldsymbol{s}_L\right) = \left[\frac{e^{s_L^{(1)}}}{\sum_{i=1}^{N_L} e^{s_L^{(i)}}} \quad \frac{e^{s_L^{(2)}}}{\sum_{i=1}^{N_L} e^{s_L^{(i)}}} \quad \cdots \quad \frac{e^{s_L^{(N_L)}}}{\sum_{i=1}^{N_L} e^{s_L^{(i)}}}\right]^T \tag{2.10}$$

2.2.2 Backpropagation (BP)

The final layer activations $\boldsymbol{a}_L$ are compared with the ground-truth labels $\boldsymbol{y}$ to compute a scalar **cost (or loss)** function C. The cost function we use is **cross-entropy**:

$$C = -\sum_{i=1}^{N_L} y^{(i)} \ln a_L^{(i)} \tag{2.11}$$

The **delta** (or error) values for every layer are computed next. These are the gradients of the cost with respect to $\boldsymbol{s}$:

$$\underset{N_L \times 1}{\boldsymbol{\delta}_L} = \left(\frac{\partial C}{\partial \boldsymbol{s}_L}\right)^T = {\boldsymbol{H}'_L}^T \boldsymbol{\nabla}_{\boldsymbol{a}_L} C \tag{2.12a}$$

$$\underset{N_i \times 1}{\boldsymbol{\delta}_i} = \left(\frac{\partial C}{\partial \boldsymbol{s}_i}\right)^T = {\boldsymbol{H}'_i}^T \boldsymbol{W}_{i+1}^T \boldsymbol{\delta}_{i+1} \qquad \forall i \in \{1, 2, \cdots, L-1\} \tag{2.12b}$$

The above equations can be derived using the chain rule for derivatives.

Classification is often done using **one-hot labels**, *i.e.* all the elements in $\boldsymbol{y}$ are 0s except for the correct class, which has value 1. For example, when classifying

an image of a digit into one out of ten possible classes $0-9$, if the correct class for a particular input sample is 3, its one-hot ground truth labeling would be $\boldsymbol{y} = [0,0,0,1,0,0,0,0,0,0]^T$. For the commonly occurring case of one-hot labels, cross-entropy cost, softmax activation for the output layer, and a scalar-to-scalar activation applied element-wise to all other layers such as ReLU, (2.12) simplifies to:

$$\underset{N_L \times 1}{\boldsymbol{\delta}_L} = \boldsymbol{a}_L - \boldsymbol{y} \tag{2.13a}$$

$$\underset{N_i \times 1}{\boldsymbol{\delta}_i} = \left(\boldsymbol{W}_{i+1}^T \boldsymbol{\delta}_{i+1}\right) \circ \boldsymbol{h}_i' \qquad \forall i \in \{1, 2, \cdots, L-1\} \tag{2.13b}$$

Note that the sigmoid derivative has a maximum value of 0.25, as shown in Fig. 2.2(b). As a result, the delta values can become very small when backpropagating through deep networks. This phenomenon is known as the vanishing gradient problem [37], and is the reason why sigmoid activations have generally fallen out of favor with MLPs.

2.2.3 Update (UP)

The goal of training a NN is to minimize the cost. An optimization algorithm commonly used for achieving this goal is **Stochastic Gradient Descent (SGD)**, wherein the value of a particular trainable parameter of the NN is updated using the gradient of the cost with respect to it. The gradients for a single input sample

are given as $\boldsymbol{\nabla}_{\boldsymbol{W}_i} C = \boldsymbol{\delta}_i \boldsymbol{a}_{i-1}^T$ and $\boldsymbol{\nabla}_{\boldsymbol{b}_i} C = \boldsymbol{\delta}_i$. Knowing the true gradient requires knowledge of the underlying data distribution, which is usually not obtainable. In SGD, the true gradients are approximated by averaging the single sample gradients over a **batch** (sometimes called minibatch) of M input samples. This gives the SGD update rule $\forall i \in \{1, 2, \cdots, L\}$ as:

$$\boldsymbol{W}_i \leftarrow \boldsymbol{W}_i - \frac{\eta}{M} \sum_{m=1}^{M} \left(\boldsymbol{\delta}_i \boldsymbol{a}_{i-1}^T \right)^{[m]} \tag{2.14a}$$

$$\boldsymbol{b}_i \leftarrow \boldsymbol{b}_i - \frac{\eta}{M} \sum_{m=1}^{M} \left(\boldsymbol{\delta}_i \right)^{[m]} \tag{2.14b}$$

where η is the **learning rate** hyperparameter determining how fast the network should learn, and square-bracketed superscript m denotes input sample number m (not to be confused with round-bracketed superscripts for individual elements in a vector or matrix).

While trainable parameters refers to weights and biases, we will use the term **network parameters** to collectively refer to $\{\boldsymbol{a}_i, \boldsymbol{h}'_i, \boldsymbol{\delta}_i, \boldsymbol{W}_i, \boldsymbol{b}_i\}$, $\forall i \in \{1, 2, \cdots, L\}$. [2] These network parameters account for the total storage cost.

[2]The range of $\boldsymbol{h}'_i$ is from 1 to $L-1$ since final layer softmax derivatives are directly combined with cost derivatives to yield $\boldsymbol{\delta}_L$.

2.3 Training and Inference

A NN is initially trained using all the operations described, *i.e.* FF, BP and UP. Inputs are traversed in batches until all inputs in the training dataset are exhausted, this constitutes an **epoch** of training. A NN can be trained for dozens or even hundreds of epochs depending on the performance desired. Two metrics for determining performance are a) the cost, which should be minimized, and more commonly, b) the classification accuracy, computed as the fraction of inputs correctly classified, which should be maximized.

Validation is commonly done after every epoch to determine how the NN will behave on unseen data. This validation data is a separate subset of the training data. The hyperparameters are adjusted according to validation performance. For example, if training performance keeps improving but validation performance deteriorates, the NN is **overfitting**. This implies that the NN is learning its training data 'too well' and failing to generalize on unseen data. This typically happens when the number of trainable parameters is far more than the number of input data samples. In such cases, training should be stopped, or regularization may prove to be helpful.

Once training is complete, the NN performs inference on test data. Inference performance, such as **classification accuracy on test data**, often serves as the defining measure for the quality of a NN. Note that inference only involves computation of (2.7a) and (2.7b), and hence is of much lower complexity than

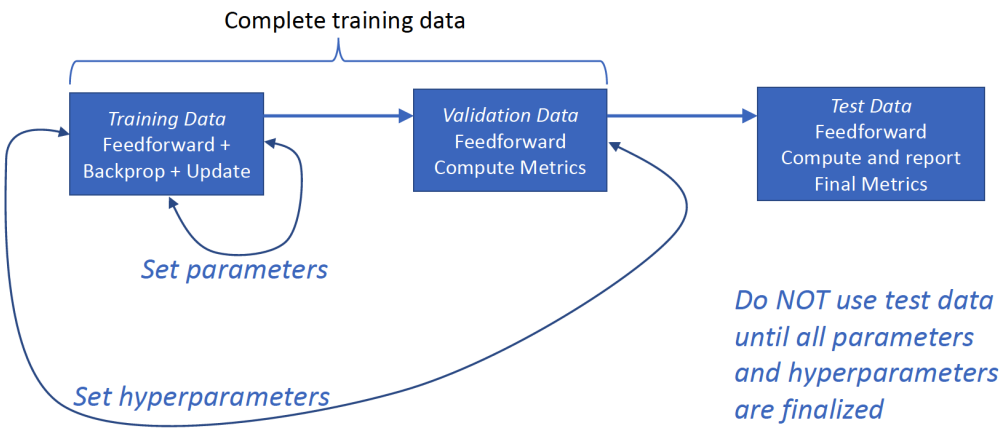


Figure 2.3: Flowchart showing the correct way to use data and set parameters and hyperparameters.

training. The complete cycle of using data for training and inference is summarized in Fig. 2.3. Next we discuss some techniques used during training and inference.

Regularization

The idea behind regularization is to impose an additional constraint on the cost objective to make the NN generalize better to unseen data (*i.e.* validation and test), and combat overfitting. A typical way to do this is by imposing an **L2 penalty** on the magnitudes of the weights. The regularized cost function becomes:

$$C = C_0 + \lambda \left\| \boldsymbol{\mathcal{W}} \right\|_2^2 \tag{2.15}$$

where C_0 is the original cost function such as given in (2.11), λ is a hyperparameter denoting the importance of making the weights small relative to minimizing the original cost function, and $\boldsymbol{\mathcal{W}}$ is the vector containing all the weights across all

junctions of the network, *i.e.* concatenation of the flattened matrices $W_i \quad \forall i \in \{1, 2, \cdots, L\}$. Typical values for λ range between 10^{-2} to 10^{-6}.

Regularization helps to improve performance particularly when NN is over-parametrized, *i.e.* it has a large number of trainable parameters. As we shall see later in this work, imposing pre-defined sparsity is a form of regularization since it reduces the number of trainable parameters.

Initializing trainable parameters

Initializing all weights to a constant such as 0 usually slows down learning. A popular technique to initialize weights is according to a normal distribution, such as **Glorot Normal** [38]

$$W_i \sim \mathcal{N}\left(0, \frac{2}{d_i^{\text{in}} + d_i^{\text{out}}}\right) \tag{2.16}$$

or **He Normal** [39]:

$$W_i \sim \mathcal{N}\left(0, \frac{2}{d_i^{\text{in}}}\right) \tag{2.17}$$

where $\mathcal{N}(\mu, \sigma^2)$ denotes the Normal distribution with mean μ and variance σ^2.

Biases can be initialized with 0s, or a small positive constant to help ReLU units remain active (*i.e.* not operating on inputs less than 0).

Optimizer

SGD, as given in (2.14), is a baseline optimizer for reducing cost. Other adaptive optimizers extend SGD by incorporating techniques such as momentum and bias correction (see [40] for an overview of these techniques). A popular optimizer is **Adam** [41], which has default values ($\eta = 0.001, \rho_1 = 0.9, \rho_2 = 0.999, \epsilon =$ machine epsilon), and works as shown below (for any arbitrary trainable parameter p).

$$v_1 \leftarrow \rho_1 v_1 + (1 - \rho_1)\nabla_p C \tag{2.18a}$$

$$v_2 \leftarrow \rho_2 v_2 + (1 - \rho_2)\left(\nabla_p C\right)^2 \tag{2.18b}$$

$$\tilde{v}_1 = \frac{v_1}{1 - \rho_1^t} \tag{2.18c}$$

$$\tilde{v}_2 = \frac{v_2}{1 - \rho_2^t} \tag{2.18d}$$

$$p \leftarrow p - \frac{\eta}{\sqrt{\tilde{v}_2} + \epsilon}\tilde{v}_1 \tag{2.18e}$$

where t is time, *i.e.* ρ_1^t is ρ_1 raised to the tth power. Time is incremented after each update (*i.e.* after the value of p changes). The time step at which the above equations occur is $t + 1$, and the initial values for v_1 and v_2 are both 0. Note that the final equation (2.18e) is similar to the regular SGD update equation with $M = 1$.

A possible modification to Adam is to add a decay parameter d such that for time step $t+1$, instead of using η, the optimizer uses $\frac{\eta}{1+dt}$. This leads to smoother convergence.

Dropout

The technique of dropout was introduced in [42]. Applying dropout to any layer results in a fraction of nodes in that layer being *dropped*, *i.e.* zeroed out. The nodes are randomly chosen for every batch, however, the fraction to be dropped typically remains fixed and is referred to as the *drop probability* p. Dropout is done only during training, and the learned weight values are multiplied by $(1-p)$ during inference to compensate.

Batch Normalization

Batch Normalization (BN) [43] has been found to be an effective in training deep NNs. Typically intermediate outputs are normalized before applying the activation function. Any variable x is normalized by subtracting its mean and dividing by its standard deviation over a batch of samples, *i.e.* $x \leftarrow \frac{x-\mu_x}{\sigma_x}$. Following this, the normalized variable is subjected to an affine transformation, *i.e.* $x \leftarrow \gamma x + \beta$, where γ and β are trainable parameters.

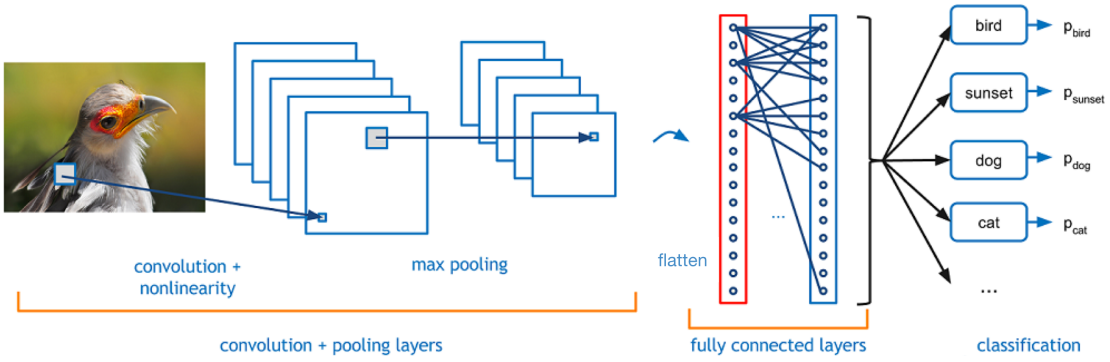


Figure 2.4: Example of a simple CNN for object classification, showing convolution, pooling and FC layers. Figure modified from [44].

2.4 Convolutional neural networks

We introduced several core concepts of NNs using MLPs. This section introduces CNNs – widely used for image classification. The dimensions of input images is typically (c, h, w), which are respectively the number of **channels** (or filters), height, and width. Channels for the input typically encode color information, *e.g.* the input image of a bird in Fig. 2.4 has 3 channels for red, green and blue.

Like MLPs, CNNs are also a form of 'feedforward' NNs (not to be confused with the FF operation) in the sense that there are no state variables and cycles within the network. CNNs primarily include **convolutional layers**, which consist of a number of **filters** of some **kernel** / **window** / **filter size**, which perform a linear correlation operation on a group of nodes in a left layer, then apply a non-linearity like ReLU to get the value of a single node in the right layer. This is repeated for the next group of left nodes until the whole left layer is covered, and then repeated for a certain number of channels.

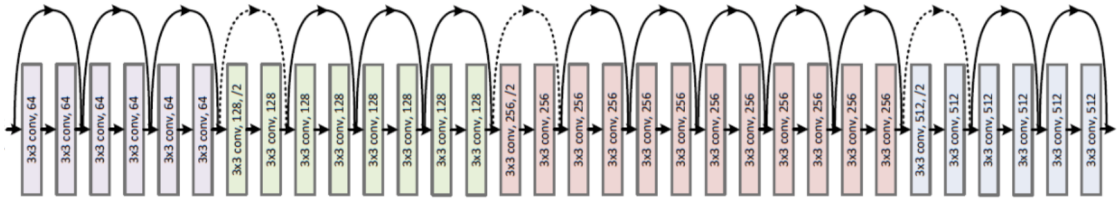


Figure 2.5: Example of shortcut connections in a CNN, where each rectangular block is a convolutional layer and the curved arrows are shortcuts. Figure courtesy [45, Fig. 2].

CNNs are typically deeper than MLPs, *e.g.* some popular CNNs in the literature have dozens of layers [12, 45]. Apart from convolutional layers, CNNs have layers to downsample the image being processed, so as to preserve relevant, higher order abstractions. This can be done via **pooling** layers, or using **strides**. The latter skips nodes when applying the correlation filter, while the former considers a group of adjacent nodes and preserves only a single value from them, such as the maximum (*max pooling*), or the average (*average pooling*). The *pool size* refers to the number of adjacent nodes being considered for pooling. Downsampling also has the advantage of keeping the total number of nodes to a manageable number since the number of filters increase as one goes deeper into the CNN. Accordingly, some works [45] downsample each dimension of the image by a factor of 2 whenever the number of filters doubles.

CNNs also typically have BN and dropout layers. A popular technique used to alleviate the vanishing gradient problem in deep CNNs is to use **shortcuts** or skip connections, introduced in [45]. These form direct paths between layers not adjacent to each other, as shown in Fig. 2.5.

CNNs used for classification also perform FF, BP and UP operations. While the ideas are the same as described in Section 2.2, the exact mathematical details differ. In particular, the weights for a convolutional layer are collected in a 4-dimensional tensor of dimensions (c_o, c_i, h, w), which are respectively the number of output channels (*i.e.* for the next convolutional layer), the number of input channels (*i.e.* for the current convolutional layer being processed), the height, and the width of the image. For the example in Fig. 2.4, let us assume the image is of size 32×32 pixels. The layer following the input has 5 filters, as shown. Thus, the weight tensor has dimensions $(5, 3, 32, 32)$.

As shown on the right side of Fig. 2.4, CNNs used for classification tasks sometimes have MLP layers following the convolutional portion. Prior to encountering these, the outputs from the convolutional portion needs to be *flattened*, *i.e.* converted from (c, h, w) format to a single dimension, *i.e.* N_0 as per our MLP notation. Typically a *global average pooling* layer precedes flattening, this is simply average pooling with pool size equal to the image height and width such that the pooling output is of dimensions $(c, 1, 1)$. The MLP portion is followed by a softmax classifier, as described previously in Section 2.2.1. Note that the MLP layers are optional, in fact, this work discusses CNNs with them (Chapter 3), as well as without them (Chapter 7).

Note that we are not going into the exact mathematical details for CNNs, or into a further exposition of CNNs, since these are not relevant for this work.

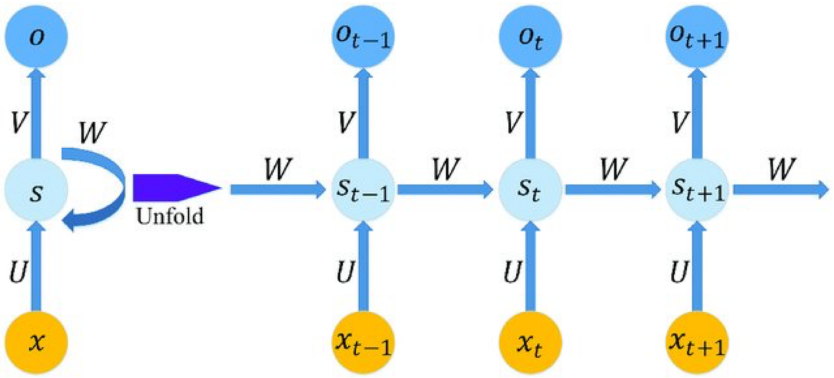


Figure 2.6: Example of a RNN with input x, states s and output o. Time is denoted by t, while U, V and W denote linear operations. Non-linear operations are not shown here for simplicity. Figure courtesy [51, Fig. 4].

The interested reader is encouraged to visit various excellent resources such as [44, 46–50] for more details on CNNs.

2.5 Recurrent neural networks

Contrary to MLPs and CNNs, Recurrent Neural Networks (RNNs) have internal **states** and may have cycles in their network structure. They are primarily used for learning dependencies over time, such as speech or video signals. Such an input signal is broken up into segments in time and may be operated on in a way similar to MLPs, *i.e.* a linear combination followed by a non-linearity, to generate intermediate states and outputs at different instances of time. A RNN is shown in Fig. 2.6. However, we will not discuss RNNs in detail since they are not essential for understanding this work.

Chapter 3

Pre-Defined Sparsity

This chapter discusses one of the major ideas driving our research – pre-defined sparsity – which deletes parameters and simplifies a NN prior to training. We will begin with related work for complexity reduction of NNs. This also includes hardware implementations, which serve as related work for our own hardware efforts described in Chapter 4.

3.1 Related Work

As mentioned in the Introduction, modern NNs suffer from parameter explosion in the sense that the number of trainable parameters ranges from millions to billions [1, 5, 6, 39]. These demand large amounts of memory to store and arithmetic resources to operate on, particularly during training. Moreover, training a network with too many parameters makes it likely to overfit [52], and memorize undesirable noise patterns [53]. However, most forms of complexity reduction aim to simplify and accelerate the inference phase only, not the more complexity-intensive training phase. Some of these are described next.

The work in [13] used vector quantization to compress the weights of deep CNNs, while [14] grouped weights into hash buckets, effectively implementing a form of coarser granularity in weight values. This approach is taken to its extreme by using only values $+1$ and -1 for weights [54]. The efforts in [7, 15, 16] use a number of techniques for reducing the storage footprint of the weights, such as deleting weights with low values (pruning), retraining only necessary weights, and Huffman coding for efficient storage. All these methods need to first train the fully connected (FC) NN with all weights present, before deciding compression strategies for inference. Similarly, other pruning and trimming methods post-process a trained FC NN to remove weights [55–57].

The technique of dropout, introduced in Chapter 2, is an ensemble method which actually increases overall training complexity since many different configurations need to be trained. Moreover, the final NN used for inference is FC, *i.e.* uncompressed. Other mathematical methods employed during training include using low-precision arithmetic [58–61], special matrices to structure the weights [62, 63], and regularizers to discard unimportant weights [64, 65] for inference, however, the latter two method classes lead to more efficient inference at the cost of increased training complexity.

Some custom hardware implementations for NNs have been developed such as Application Specific Integrated Circuit (ASIC)-based [16, 56, 66–69], and FPGA-based [54, 55, 63, 70–72]. We note that while all these methods have achieved

excellent results in making inference faster and more efficient **on-device**, training is typically done for the complete uncompressed network off-device, such as on a power-hungry GPU or cloud server. Note that there have been some efforts in on-device training [73–77]. However, these have not been targeted towards complexity reduction and hence some have been limited to implementing NNs with a fairly small number of total nodes – 21 in [75], 83 in [74], and a more impressive 221 in [73] (some of the pipelining ideas used in our hardware architecture described in Chapter 4 were inspired by [73]).

In summary, based on related work and the present state of NNs, we identify two open problems – a) Design methods and algorithms to reduce the complexity of *training* NNs, and b) a flexible hardware architecture to support both low-complexity training and inference on-device. This chapter proposes a method aimed at tackling the first problem.

3.2 Structured Pre-defined sparsity

Pre-defined sparsity refers to a class of methods where a NN is made sparse by removing some of its edges (connections) *prior to training*. This implies that both training and inference use a NN of lower complexity as compared to a regular non-sparse NN, such as one having FC layers. To the best of our knowledge, we are the first to propose pre-defined sparsity as a technique for complexity reduction in [27].

Note that several other authors have recently proposed pre-defined sparsity [78–80] independent of our work.

Note that our work on pre-defined sparsity applies to MLPs. Thus, when we mention 'NN' in the context of our efforts on pre-defined sparsity, we are either referring to a MLP, or to the MLP portion of a different NN (such as a CNN).

To understand pre-defined sparsity, we need to recall and add to the basic definitions from Section 2.2. The **out-degree** of a node j in the left layer $i-1$ of junction i, $d_i^{\text{out}(j)}$, is the total number of edges connecting it to layer i to its right. Likewise, the **in-degree** of a node j in the right layer i of junction i, $d_i^{\text{in}(j)}$, is the total number of edges connecting it to layer $i-1$ to its left. When these numbers are constant for all nodes in a layer, the superscript can be omitted and we get d_i^{out} and d_i^{in} for junction i. Note that for a conventional FC junction, $d_i^{\text{out}} = N_i$ and $d_i^{\text{in}} = N_{i-1}$. We refer to the total number of edges in junction i as $|\boldsymbol{W}_i|$, thus $|\boldsymbol{W}_i| = N_{i-1}N_i$ for FC.

➤ **Definition 1:** ***Pre-defined sparsity*:** A junction i is pre-defined sparse if it does not have all N_iN_{i-1} edges present. A NN is pre-defined sparse if it has at least one junction which is pre-defined sparse.

➤ **Definition 2:** ***Structured pre-defined sparsity*:** Structured pre-defined sparsity is pre-defined sparsity with fixed in- and out-degrees for each junction.

Thus, every node in layer $i-1$ has a fixed $d_i^{\text{out}} \leq N_i$, and every node in layer i has a fixed $d_i^{\text{in}} \leq N_{i-1}$. This leads to $|\boldsymbol{W}_i| = N_{i-1}d_i^{\text{out}} = N_i d_i^{\text{in}}$, thus

$|\boldsymbol{W}_i| \leq N_{i-1}N_i$. The **density** of junction i is measured relative to FC and denoted as $\rho_i = |\boldsymbol{W}_i|/(N_{i-1}N_i)$. (Occasionally we may refer to sparsity as the opposite of density, so a junction which is 20% dense is 80% sparse). As will be shown in subsequent chapters, imposing the structured constraint leads to performance improvement and ease of hardware implementation as compared to distributing connections randomly. For the remainder of this work, pre-defined sparsity will always refer to the structured form unless otherwise mentioned.

The **overall density** of a pre-defined sparse NN is defined as:

$$\rho_{\text{net}} = \frac{\sum_{i=1}^{L} |\boldsymbol{W}_i|}{\sum_{i=1}^{L} N_{i-1}N_i} = \frac{\sum_{i=1}^{L} N_{i-1}d_i^{\text{out}}}{\sum_{i=1}^{L} N_{i-1}N_i} = \frac{\sum_{i=1}^{L} N_i d_i^{\text{in}}}{\sum_{i=1}^{L} N_{i-1}N_i} \tag{3.1}$$

i.e. the total number of edges in the NN as a fraction of the total number of edges in the corresponding FC NN. Thus, specifying $\boldsymbol{N}_{\text{net}} = (N_0, N_1, \cdots, N_L)$ and either of the **out-degree configuration** $\boldsymbol{d}_{\text{net}}^{\text{out}} = (d_1^{\text{out}}, d_2^{\text{out}}, \cdots, d_L^{\text{out}})$ or the **in-degree configuration** $\boldsymbol{d}_{\text{net}}^{\text{in}} = (d_1^{\text{in}}, d_2^{\text{in}}, \cdots, d_L^{\text{in}})$ completely determines the density of each junction and the overall density. Fig. 3.1 illustrates these basic concepts for a simple MLP.

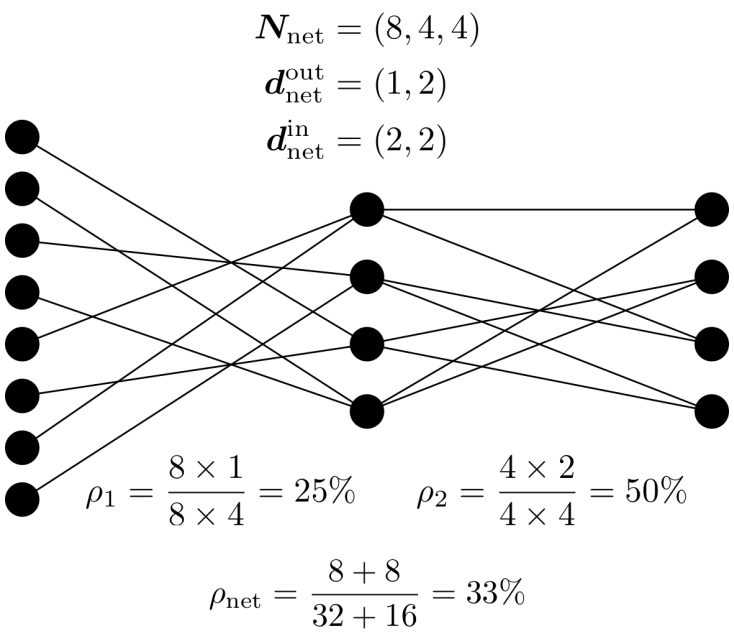


Figure 3.1: Illustrating basic concepts of structured pre-defined sparsity.

The 'pre' in pre-defined sparsity

We would like to emphasize that all densities of a pre-defined sparse NN are set prior to training, and then held fixed throughout training and inference. This means that the **connection pattern**, *i.e.* which node(s) connects to which node(s) and which edges are present and absent, does not change for the NN once it is set prior to training at the beginning.

As regards other NNs, the connection pattern in CNNs is different from MLPs – every node in a convolution layer receives inputs from various filters applied to a localized region of the inputs in its previous layer. For RNNs, a node may receive raw inputs and those from adjacent network states, which themselves may be MLPs or CNNs. A complete investigation of pre-defined sparsity for recurrent

and convolution layers is beyond the scope of this work, however, as will be shown by our experiments, our efforts in pre-defined sparsity extend to MLP layers which may be part of a different network such as a CNN. Note that followup work in our research group [81] has applied pre-defined sparsity to convolutional layers [82,83].

3.2.1 Motivation and Preliminary Examples

Pre-defined sparsity can be motivated by inspecting the histogram for trained weight values in a FC NN. There have been previous efforts to study such statistics [7,84], however, not for individual junctions. Fig. 3.2 shows weight histograms for each junction in both a 2-junction and 4-junction FC NN trained on the MNIST dataset on handwritten digit classification (datasets will be described in more detail in Section 3.3.1). Note that many of the weights are zero or near-zero after training, especially in the earlier junctions. This motivates the idea that some weights in these junctions could be set to zero (*i.e.* the edges excluded).

Even with this intuition, it is unclear that one can pre-define a set of weights to be zero and let the NN learn around this constraint. Fig. 3.2(c) and (h) show that, in fact, this is the case – *i.e.* this shows classification accuracy performance on the test set as a function of the overall density ρ_{net} for structured pre-defined sparsity. The circled point on the extreme right of each subfigure is the FC case. Note that even at $\rho_{\text{net}} = 20\%$, the degradation in classification performance is

within 1%. Since the computational and storage complexity is directly proportional to the number of edges in the NN, operating at an overall density of 20% results in a 5X reduction in complexity both during training and inference. These preliminary examples serve to show the **effectiveness of pre-defined sparsity in reducing complexity of NNs while incurring minimal performance loss**. Detailed numerical experiments in Section 3.3 will further build on these preliminary examples.

3.2.2 Structured Constraints

In terms of the weight matrix, if junction i is pre-defined sparse, then $\boldsymbol{W}_i$ has zeroes indicating the absence of edges and non-zero elements indicating the weight values of present edges. Note that in the course of random initialization or training updates, weight values of present edges can also become zero. However, the absent edges in a NN will always remain absent, *i.e.* :

$$\mathbb{I}\left(\boldsymbol{W}_i^{(j,:)} \neq 0\right) \leq d_i^{\text{in}} \qquad \forall j \in \{1, \cdots, N_i\} \tag{3.2a}$$

$$\mathbb{I}\left(\boldsymbol{W}_i^{(:,k)} \neq 0\right) \leq d_i^{\text{out}} \qquad \forall k \in \{1, \cdots, N_{i-1}\} \tag{3.2b}$$

where $\mathbb{I}$ is the indicator function, which has value 1 if its argument is true, otherwise 0.

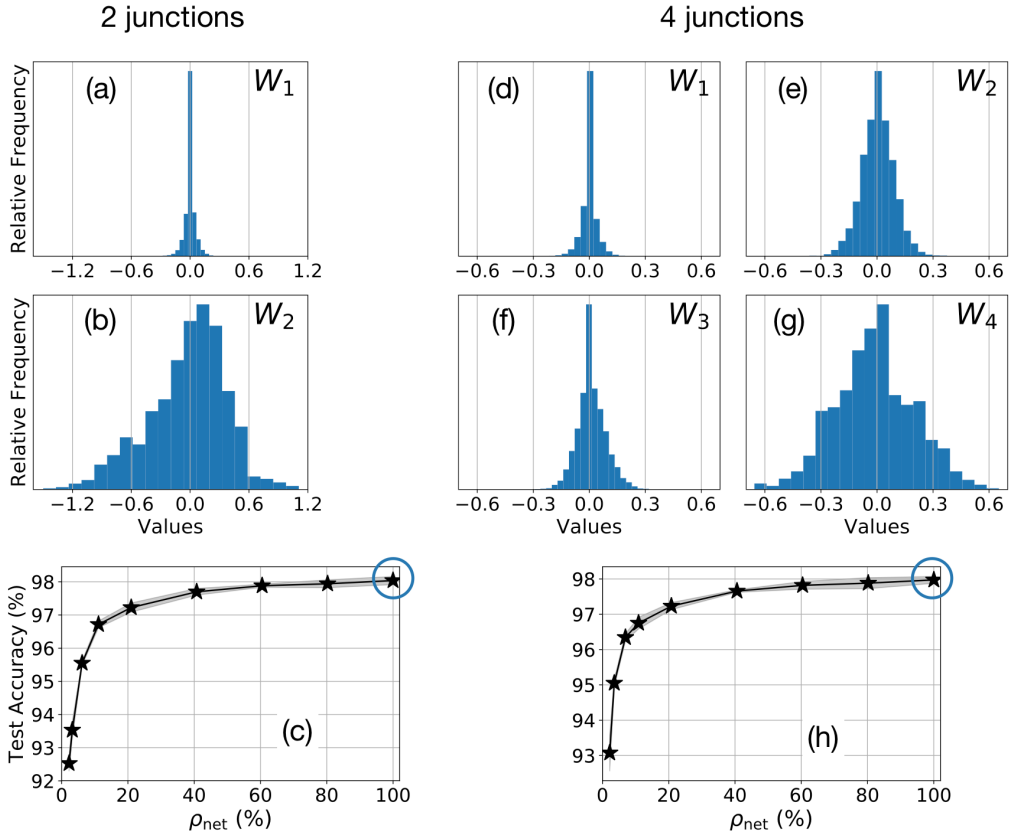


Figure 3.2: Histograms of weight values in different junctions for FC NNs trained on MNIST for 50 epochs, with (a-b) $\boldsymbol{N}_{\text{net}} = (800, 100, 10)$, and (d-g) $\boldsymbol{N}_{\text{net}} = (800, 100, 100, 100, 10)$. Test set classification accuracy shown in (c,h) for different NNs with same $\boldsymbol{N}_{\text{net}}$ and varying ρ_{net}. The FC cases ($\rho_{\text{net}} = 100\%$) are circled to provide a baseline. The remaining ρ_{net} values are set by reducing ρ_1 since junction 1 has more weights close to zero in the FC cases.

Imposing the structured constraint restricts the number of possible density values. As an example, the NN shown in Fig. 2.1 with $\boldsymbol{N}_{\text{net}} = (4, 3, 2)$ cannot have $d_1^{\text{out}} = 2$ (*i.e.* $\rho_1 = 2/3$) since that would lead to $d_1^{\text{in}} = 8/3$, and a node cannot have a fractional number of connections. This leads to:

Theorem 1

For a given junction i, the total number of different ρ_i values possible is equal to the Greatest Common Divisor (gcd) of N_i and N_{i-1}.

Proof: Consider a NN junction i. Its density ρ_i cannot be arbitrary, since $\rho_i = d_i^{\text{out}}/N_i = d_i^{\text{in}}/N_{i-1}$, where d_i^{out} and d_i^{in} are natural numbers satisfying the equation $N_{i-1}d_i^{\text{out}} = N_i d_i^{\text{in}}$. Therefore, the number of possible ρ_i values is the same as the number of $\left(d_i^{\text{out}}, d_i^{\text{in}}\right)$ values satisfying the structured pre-defined sparsity constraints:

$$d_i^{\text{out}} = \frac{N_i d_i^{\text{in}}}{N_{i-1}}, \qquad d_i^{\text{in}} \leq N_{i-1}, \qquad d_i^{\text{out}}, d_i^{\text{in}} \in \mathbb{N} \tag{3.3}$$

where $\mathbb{N}$ denotes the set of natural numbers.

The smallest value of d_i^{in} which satisfies $d_i^{\text{out}} \in \mathbb{N}$ is $N_{i-1}/\text{gcd}(N_{i-1}, N_i)$, and other values are its integer multiples. Since d_i^{in} is upper bounded by N_{i-1}, the total number of possible $\left(d_i^{\text{out}}, d_i^{\text{in}}\right)$ is $\text{gcd}(N_{i-1}, N_i)$. Thus, the set of possible ρ_i is

$$\left\{ \rho_i \in (0,1] \,\middle|\, \rho_i = \frac{k}{\text{gcd}(N_{i-1}, N_i)}, k \in \mathbb{N} \right\}. \tag{3.4}$$

which is a set of cardinality equal to the gcd of N_i and N_{i-1}. ■

As a concrete example, consider a NN with $\boldsymbol{N}_{\text{net}} = (117, 390, 13)$ (this will arise in our experiments on the TIMIT dataset described in Section 3.3). The

number of possible junction densities are $\gcd(117, 390) = 39$ and $\gcd(390, 13) = 13$. Therefore, the possible junction densities are:

$$\rho_1 \in \left\{ \frac{1}{39}, \frac{2}{39}, \cdots, \frac{39}{39} \right\}, \qquad \rho_2 \in \left\{ \frac{1}{13}, \frac{2}{13}, \cdots, \frac{13}{13} \right\}. \tag{3.5}$$

The above proof and example for Theorem 1 was achieved in collaboration with Kuan-Wen Huang.

3.2.3 Modifications to Neural Network Operations

The basic operations of a NN – FF, BP and UP – are modified for the case of structured pre-defined sparsity. The primary change is that only the present weights are used for computation, which leads to reduction in complexity. Now we will present the modified equations for the commonly occurring case of one-hot labels, cross-entropy cost, softmax activation for the output layer, and a scalar-to-scalar activation applied element-wise to all other layers. The matrix-vector products are split into summations to highlight usage of only the present weights.

The FF equations (2.7) become:

$$s_i^{(j)} = \sum_{f=1}^{d_i^{\text{in}}} W_i^{(j,k_f)} a_{i-1}^{(k_f)} + b_i^{(j)} \tag{3.6a}$$

$$a_i^{(j)} = h\left(s_i^{(j)}\right) \tag{3.6b}$$

$$h_i'^{(j)} = \frac{\partial a_i^{(j)}}{\partial s_i^{(j)}} \tag{3.6c}$$

where (3.6a) and (3.6b) are $\forall i \in \{1, 2, \cdots, L\}$, and (3.6c) is $\forall i \in \{1, 2, \cdots, L-1\}$. This is because softmax activation derivatives are directly combined with cross-entropy cost derivatives in the final layer to give (3.7a), hence are not separately required to be computed.

The BP equations (2.13) become:

$$\delta_L^{(j)} = a_L^{(j)} - y^{(j)} \tag{3.7a}$$

$$\delta_i^{(j)} = {h'}_i^{(j)} \left(\sum_{f=1}^{d_i^{\text{out}}} W_{i+1}^{(k_f, j)} \delta_{i+1}^{(k_f)} \right) \tag{3.7b}$$

where (3.7b) is $\forall i \in \{1, 2, \cdots, L-1\}$.

The UP equations (2.14), assuming batch size $M = 1$ to reduce clutter, become:

$$b_i^{(j)} \leftarrow b_i^{(j)} - \eta \delta_i^{(j)} \tag{3.8a}$$

$$W_i^{(j,k)} \leftarrow W_i^{(j,k)} - \eta a_{i-1}^{(k)} \delta_i^{(j)} \tag{3.8b}$$

where both (3.8a) and (3.8b) are $\forall i \in \{1, 2, \cdots, L\}$, and (3.8b) is only for those (j, k) node pairs which have an edge connecting them.

For all the equations (3.6)–(3.8) above, the single superscript j refers to nodes in a layer and hence has range $\{1, 2, \cdots, N_i\}$. Double superscripts denote *only those edges which are present*. For example, in (3.6a), the summation for a particular row j of $\boldsymbol{W}_i$ is carried out over d_i^{in} different values from $W_i^{(j,k_1)}$ to $W_i^{(j,k_{d_i^{\text{in}}})}$. Considering Fig. 3.1 as a concrete example, when doing the computation for the 1st right neuron

in junction 1 (*i.e.* $i = 1$, $j = 1$), we get $k_1 = 5$ and $k_2 = 7$. This is because nodes 5 and 7 from layer 0 connect to node 1 of layer 1.

This concludes the theory relating to pre-defined sparsity. In the next section, we present related performance results.

3.3 Performance Results, Trends and Guidelines

Fig. 3.2(c) and (h) showed some preliminary examples regarding the potential of pre-defined sparsity to reduce complexity with minimal degradation in performance. This section analyzes results and trends observed when experimenting with several different classification datasets via software simulations. We intend the following to provide guidelines on designing pre-defined sparse NNs.

Guidelines for designing pre-defined sparse NNs

1. The performance of pre-defined sparsity is better on datasets that have more inherent redundancy (Section 3.3.2).

2. Junction density should increase to the right, *i.e.* junctions closer to the output should generally have more density than junctions closer to the input (Section 3.3.3).

3. Larger and more sparse NNs are better than smaller and denser NNs, given the same number of layers and trainable parameters. Specifically, 'larger' refers to more hidden neurons (Section 3.3.4).

The reader is reminded that when we refer to pre-defined sparsity, we are actually referring to structured pre-defined sparsity, *i.e.* fixed in- and out-degrees. The remainder of this section first describes the datasets we experimented on, and then examines these trends in detail.

3.3.1 Datasets and Experimental Configuration

NNs for classification need significant amounts of quality labeled data to train and perform inference. Here we discuss some widely used datasets made publicly available for this purpose, and the training configuration we designed for each of them.

MNIST handwritten digits [85]

The Modified National Institute of Standards and Technology (MNIST) database has 70,000 images of handwritten digits from $0-9$, split into 60,000 for training and 10,000 for test. We further split the training set into 50,000 for actual training and 10,000 for validation. Each image is 28×28 pixels of varying shades of gray values, from 0 signifying completely black to 255 signifying completely white. We rasterized each input image into a single layer of 784 features. This is the *permutation-invariant* format, wherein spatial information from the inputs are not used, implying that the ordering of the 784 pixels is not important. On certain occasions we added 16 input features which are always trivially 0 so as to get 800

features for each input. This leads to easier selection of different sparse network configurations. In Fig. 3.2(c) for example, we used 800 input neurons and 100 hidden neurons since $\gcd(800, 100) > \gcd(784, 100)$, so more values of ρ_{net} could be simulated. We verified that adding extra always-0 input features did not alter performance. Also note that no data augmentation was applied.

Reuters RCV1 corpus of newswire articles [86]

Reuters Corpus Volume I (RCV1) is an archive of newswire stories, each assigned categories based on its content. The classification categories are grouped in a tree structure, *e.g.* a top-level category like politics can be subdivided into second level categories for international, national and state, international can be further subdivided into third level categories for Asia, Africa and so on. An article can have multiple categories, however, we used preprocessing techniques inspired by [4] to isolate only those articles which belonged to a single second level category. We finally obtained 328,669 articles in 50 categories, split into 50,000 for validation, 100,000 for test, and the remaining 178,669 for training.

The original data has a list of token strings for each story, for example, a story on finance would frequently contain the token 'financ' (which can refer to 'finance', 'financial', 'financier' and so on). We chose the most common 2000 tokens across all articles and computed counts for each of these in each article. Each count x

was transformed into $\log(1+x)$ to form the final 2000-dimensional feature vector for each input.

TIMIT speech corpus [87]

TIMIT (presumably abbreviated from Texas Instruments, Massachusetts Institute of Technology) is a speech dataset comprising approximately 5.4 hours of 16 kHz audio commonly used in ASR. A modern ASR system has three major components: (i) preprocessing and feature extraction, (ii) acoustic model, and (iii) dictionary and language model. A complete study of an ASR system is beyond the scope of this work. Instead we focus on the acoustic model which is typically implemented using a NN. The input to the acoustic model is feature vectors and the output is a probability distribution on phonemes (*i.e.* speech sounds), *i.e.* the classification targets for our experiments are different phonemes. We used a phoneme set of size 39 as defined in [88].

We extracted 25ms speech frames with 10ms shift, as in [4], and computed a feature vector of 39 Mel-frequency Cepstral Coefficients (MFCCs) for each frame. MFCCs are obtained from a sequence of operations – windowing, Fourier transform, mapping powers on to mel scale [89], taking logarithm, discrete cosine transform, measuring amplitude. We used the complete training set of 818,837 training samples (462 speakers), 89,319 validation samples (50 speakers), and 212,093 test samples (118 speakers).

CIFAR images [90]

The Canadian Institute For Advanced Research (CIFAR) datasets come in two forms – 10 classes (CIFAR-10), and 100 classes (CIFAR-100). Each has 50,000 training samples (which we split into 40,000 for actual training and 10,000 for validation), and 10,000 test samples. The samples are images with dimensions $(c, h, w) = 3, 32, 32$, and depict objects such as different vehicles and animals. We experimented on the CIFAR-100 dataset with a CNN. The CNN has 3 blocks, each block has 2 convolution layers with window size 3×3 followed by a max pooling layer of pool size 2×2. The number of filters for the six convolution layers are (from left to right) 60, 60, 125, 125, 250 and 250. This results in a total of approximately one million trainable parameters in the convolutional portion of the network. Batch normalization is applied before activations. The output from the 3rd block, after flattening into a vector, has 4000 features. This is the input to a MLP. Typically dropout is applied in the MLP portion, however we omitted it there since pre-defined sparsity is an alternate form of parameter reduction. Instead we found that a dropout probability of 0.5, *i.e.* half the nodes and weights dropped for each batch of training, applied to the convolution blocks improved performance. No data augmentation was applied.

Experimental Configuration

For each dataset, we performed classification using one-hot labels and measured accuracy on the test set as a performance metric.[1] We also calculated the top-5 test set classification accuracy for CIFAR-100, *i.e.* the percentage of samples for which the ground-truth label was among the top 5 softmax outputs of the network.

We found the optimal training configuration and hyperparameters for each FC setup by doing a grid search using validation performance as a metric. This resulted in choosing ReLU activations for all layers except for the final softmax layer. He initialization worked best for the weights; while for biases, we found that an initial value of 0.1 worked best in all cases except for Reuters, for which zeroes worked better. The Adam optimizer was used with all parameters set to default, except that we set the decay parameter to 10^{-5} for best results. We used a batch size of 1024 for TIMIT and Reuters since the number of training samples is large, and 256 for MNIST and CIFAR. All experiments were run for 50 epochs of training and all

[1]The NN in a complete ASR system would be a 'soft' classifier and feed the phoneme distribution outputs to a decoder to perform 'hard' final classification decisions. Therefore for TIMIT, we computed another performance metric called Test Prediction Comparison (TPC), measured as KL divergence between predicted test output probability distributions of sparse vs the respective FC case. In other words,

$$\text{TPC} = \frac{1}{212093} \sum_{i=1}^{212093} \sum_{j=1}^{39} A_{ij} \log \left(\frac{A_{ij}}{B_{ij}} \right) \tag{3.9}$$

where $\underset{212093\times39}{A}$ and $\underset{212093\times39}{B}$ are the complete output test matrices for the 212,093 test samples with 39 labels each for the FC and sparse cases, respectively. Lesser values of TPC are better as they indicate minimal performance degradation due to sparsification. We found that performance results obtained using the TPC metric were qualitatively very similar to those obtained from the test accuracy metric, hence we omit the TPC results.

hyperparameters listed so far were kept the same when sparsifying the network to maintain consistency.

The only exception is regularization. We found from validation that sparser networks need lesser regularization than denser networks. This confirms our belief that **pre-defined sparse NNs are less prone to overfitting due to having fewer trainable parameters**. Accordingly we applied an L2 penalty to the weights, but reduced the coefficient λ as ρ_{net} decreased.

Each experiment was run at least five times to average out randomness and the 90% Confidence Intervals (CIs) for each metric are shown as shaded regions (this also holds for the results in Fig. 3.2(c,h)).

Experiments were conducted using the Keras deep learning library [91] (version 2.2.x) with Tensorflow [92] 1.x backend.

3.3.2 Dataset Redundancy

Many machine learning datasets have considerable redundancy in their input features. For example, one may not need information from the $\sim$800 input features of MNIST to infer the correct image class. We hypothesize that pre-defined sparsity takes advantage of this redundancy, and will be less effective when the redundancy is reduced. To test this, we changed the feature vector for each dataset as follows:

- MNIST: Principal Component Analysis (PCA) is a technique to compute variances in each feature across all inputs in the dataset, and assign more

'importance' to features with large variance. This is because such features are more discriminatory, *i.e.* observing their values will provide the network with more information. We used PCA to reduce the feature count of MNIST to the most important (least redundant) 200, *i.e.* a factor of ∼4X.

- Reuters: Size of the feature vector is the number of most frequent tokens considered. We reduced this from 2000 to 400, *i.e.* by 5X.

- TIMIT: We both reduced and increased the number of MFCCs by 3X to 13 and 117, respectively. Note that the latter increases redundancy.

- CIFAR-100: In the CNN used for CIFAR, a source of redundancy is the depth of the convolution+pooling portion which extracts features and discriminates between classes before the MLP performs final classification. In other words, the convolution blocks ease the burden of the MLP. So a way to reduce redundancy and increase the classification burden of the MLP is to lessen the effectiveness of the convolution layers by reducing their number. Accordingly, we used a single convolution layer with 250 filters of window size 5×5 followed by a 8×8 max pooling layer. This results in the same number of features, 4000, at the input of the MLP as the original network, but has reduced redundancy for the MLP.

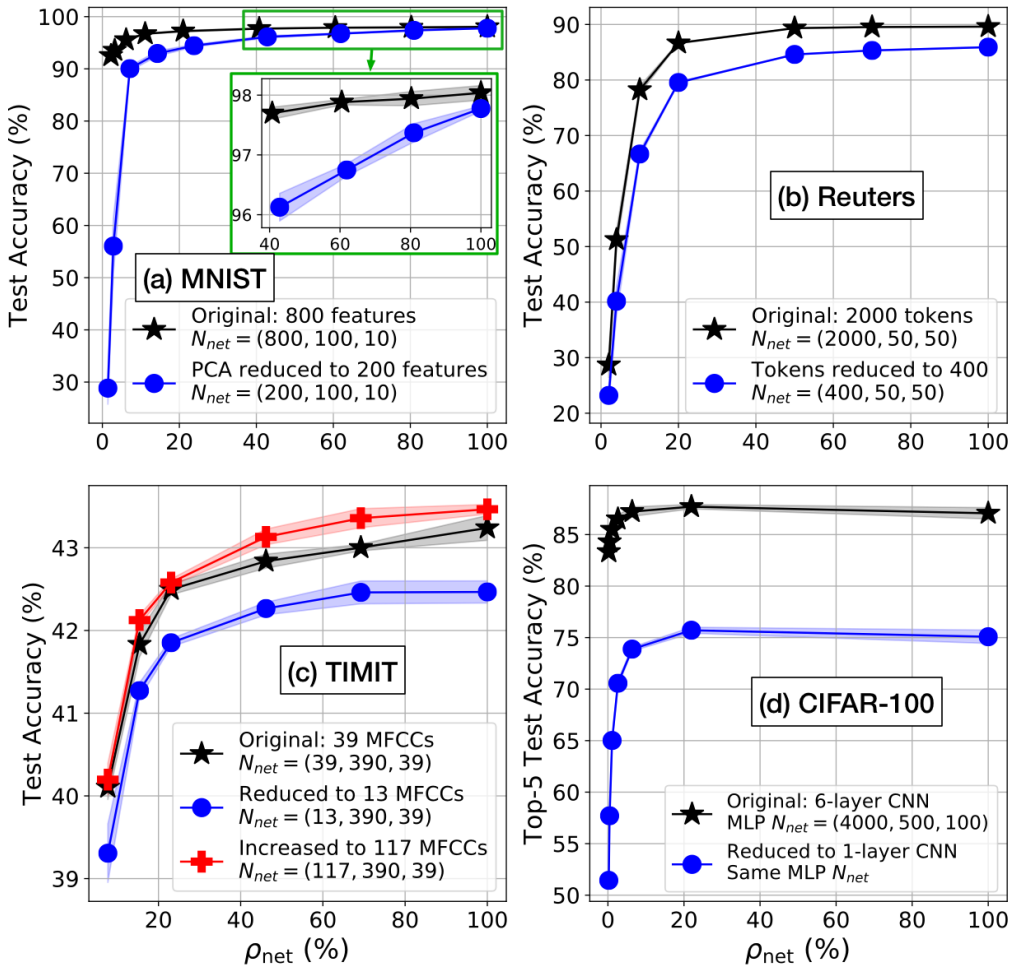


Figure 3.3: Comparison of classification accuracy as a function of ρ_{net} for different versions of datasets – original, reduced in redundancy by reducing feature space (MNIST, Reuters, TIMIT) or performing less processing prior to the MLP (CIFAR-100), and increasing redundancy by enlarging feature space (TIMIT). Higher density points for MNIST are magnified.

Classification performance results are shown in Fig. 3.3 as a function of ρ_{net}. For MNIST and CIFAR-100, the performance degrades more sharply with reducing ρ_{net} for the networks using the reduced redundancy datasets. To explore this further, we recreated the histograms from Fig. 3.2 for the reduced redundancy datasets, *i.e.* a FC NN with $\boldsymbol{N}_{\text{net}} = (200, 100, 10)$ training on MNIST after PCA.

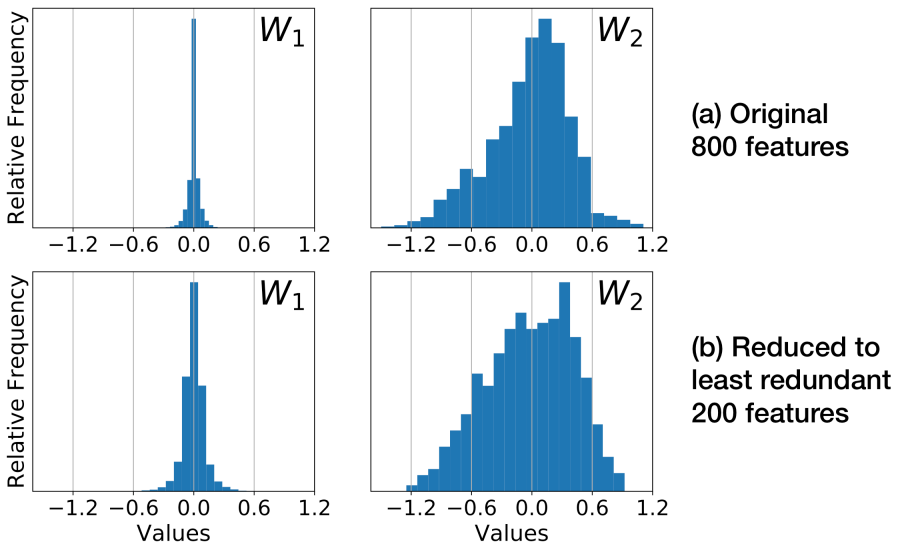


Figure 3.4: Comparing histograms of weight values for (a) original MNIST with $\boldsymbol{N}_{\text{net}} = (800, 100, 10)$, and (b) MNIST reduced to least redundant 200 features such that $\boldsymbol{N}_{\text{net}} = (200, 100, 10)$, after training FC NNs for both.

We observed a wider spread of weight values, implying less opportunity for sparsification (*i.e.* fewer weights were close to zero), as shown in Fig. 3.4. Similar trends are less discernible for Reuters and TIMIT, however, reducing redundancy led to worse performance overall.

The results in Fig. 3.3 further demonstrate the effectiveness of pre-defined sparsity in greatly reducing network complexity with negligible performance degradation. For example, even the reduced redundancy problems perform well when operating with half the number of connections. For CIFAR in particular, *FC performs worse than an overall MLP density of around 20%*. Thus, in addition to reducing complexity, structured pre-defined sparsity may be viewed as an alternative to dropout in the MLP for the purpose of improving classification.

3.3.3 Individual junction densities

The weight value histograms in Figs. 3.2 and 3.4 indicate that latter junctions, particularly junction L closest to the output, have a wide spread of weight values. This suggests that a good strategy for reducing ρ_{net} would be to use lower densities in earlier junctions, *i.e.* $\rho_1 < \rho_L$.

This is demonstrated in Fig. 3.5 for the cases of MNIST, CIFAR-100 and Reuters, each with $L = 2$ junctions in their MLPs. Each curve in each subfigure is for a fixed ρ_2, *i.e.* reducing ρ_{net} across a curve is done solely by reducing ρ_1. For a fixed ρ_{net}, the performance improves as ρ_2 increases. For example, the circled points in Reuters both have $\rho_{\text{net}} = 4\%$, but the starred point with $\rho_2 = 100\%$ has approximately 40% better test accuracy than the pentagonal point with $\rho_2 = 2\%$. The trend clearly holds for MNIST and Reuters, and is also discernible for CIFAR-100.

This trend extends to NNs with more than two junctions. Fig. 3.6 compares classification performance by trading off ρ_2 and ρ_3 (keeping ρ_1 fixed) for different three-junction NNs training on MNIST. For each individual plotted curve (same color), ρ_3 is kept constant and ρ_{net} is varied by varying ρ_2. Note that all the results are qualitatively similar to Fig. 3.5, *i.e.* for the same ρ_{net}, better results are obtained for higher values of ρ_3. Note that ρ_1 is fixed for each subfigure at fairly low values, as given in the caption of Fig. 3.6. We also experimented with keeping ρ_1 fixed at higher values, such as 50%. Since the input layer has the most neurons,

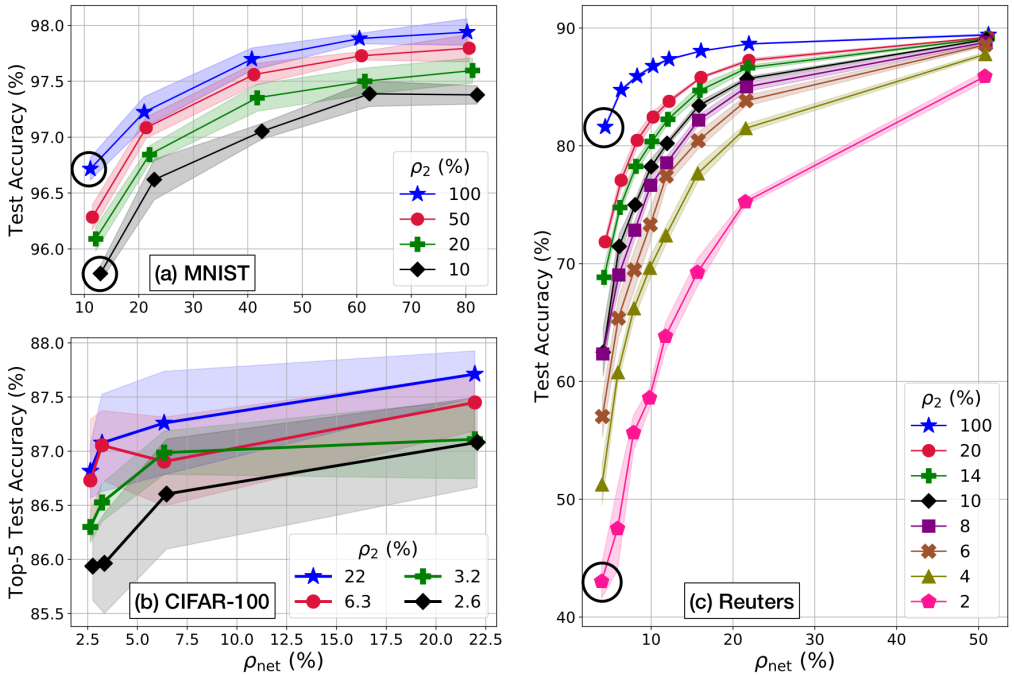


Figure 3.5: Comparison of classification accuracy as a function of ρ_{net} for different ρ_L, where $L = 2$. Black-circled points show the effects of ρ_2 when ρ_{net} is the same. $\boldsymbol{N}_{\text{net}}$ values are $(800, 100, 10)$ for MNIST, $(2000, 50, 50)$ for Reuters, and $(4000, 500, 100)$ for the MLP portion in CIFAR-100.

this resulted in junction 1 dominating the total number of weights. As a result, changing ρ_2 and ρ_3 had little effect on accuracy. Hence we decided to include the more interesting low ρ_1 cases in Fig. 3.6.

We further observed that this trend of $\rho_{i+1} > \rho_i$ improving performance is related to the redundancy inherent in the dataset, and may not hold for datasets with very low levels of redundancy. To explore this, results analogous to those in Fig. 3.5 are presented in Fig. 3.7 for TIMIT, but with varying sized MFCC feature vectors – *i.e.* datasets corresponding to larger feature vectors will contain more redundancy. The results in Fig. 3.7(c) are for 117-dimensional MFCCs and

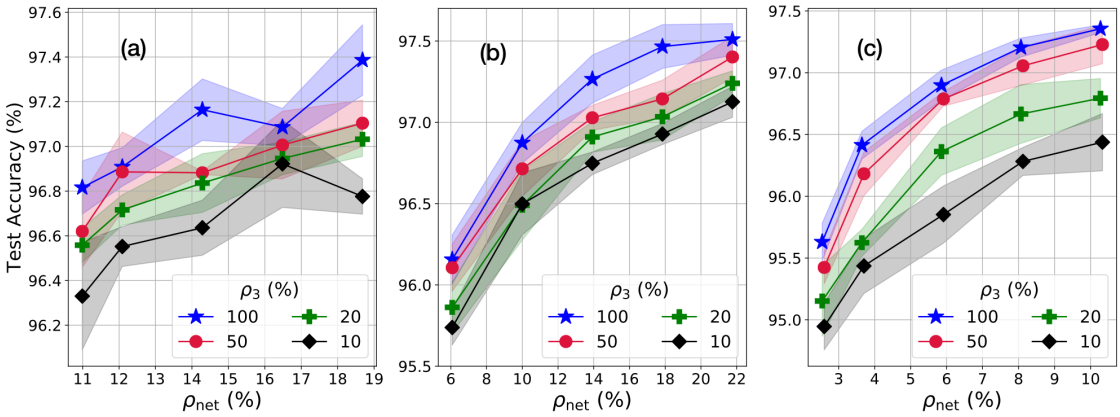


Figure 3.6: Comparison of classification accuracy as a function of ρ_{net} for ρ_2 vs ρ_3 in three-junction MNIST networks, keeping ρ_1 fixed. (a) $\boldsymbol{N}_{\text{net}} = (800, 100, 100, 10)$, $\rho_1 = 10\%$, (b) $\boldsymbol{N}_{\text{net}} = (800, 100, 200, 10)$, $\rho_1 = 4\%$, (c) $\boldsymbol{N}_{\text{net}} = (800, 200, 100, 10)$, $\rho_1 = 1\%$.

are consistent with the trend in Fig. 3.5. However, for a MFCC dimension of 13, this trend actually reverses – *i.e.* junction 1 should have higher density for better performance. This is shown in Fig. 3.7(b), where each curve is for a fixed ρ_1. This reversed trend is also observed for the case of 39 dimensional feature vectors, considered in Fig. 3.7(a), where $\boldsymbol{N}_{\text{net}} = (39, 390, 39)$. Due to this symmetric neuronal configuration, for each value of ρ_{net} on the x-axis in Fig. 3.7(a), the two curves have complementary values of ρ_1 and ρ_2 ($\rho_1 \neq \rho_2$) – *e.g.* the two curves at $\rho_{\text{net}} = 7.69\%$ have (ρ_1, ρ_2) values of $(2.56\%, 12.82\%)$ and $(12.82\%, 2.56\%)$. We observe that the curve for $\rho_1 < \rho_2$ is generally worse than the curve for $\rho_2 < \rho_1$, which indicates that junction 1 should have higher density for improving performance in this case as well.

Fig. 3.7(d) depicts the results for Reuters with the feature vector size reduced to 400 tokens. While junction 2 is still more important (as in Fig. 3.5(c) for

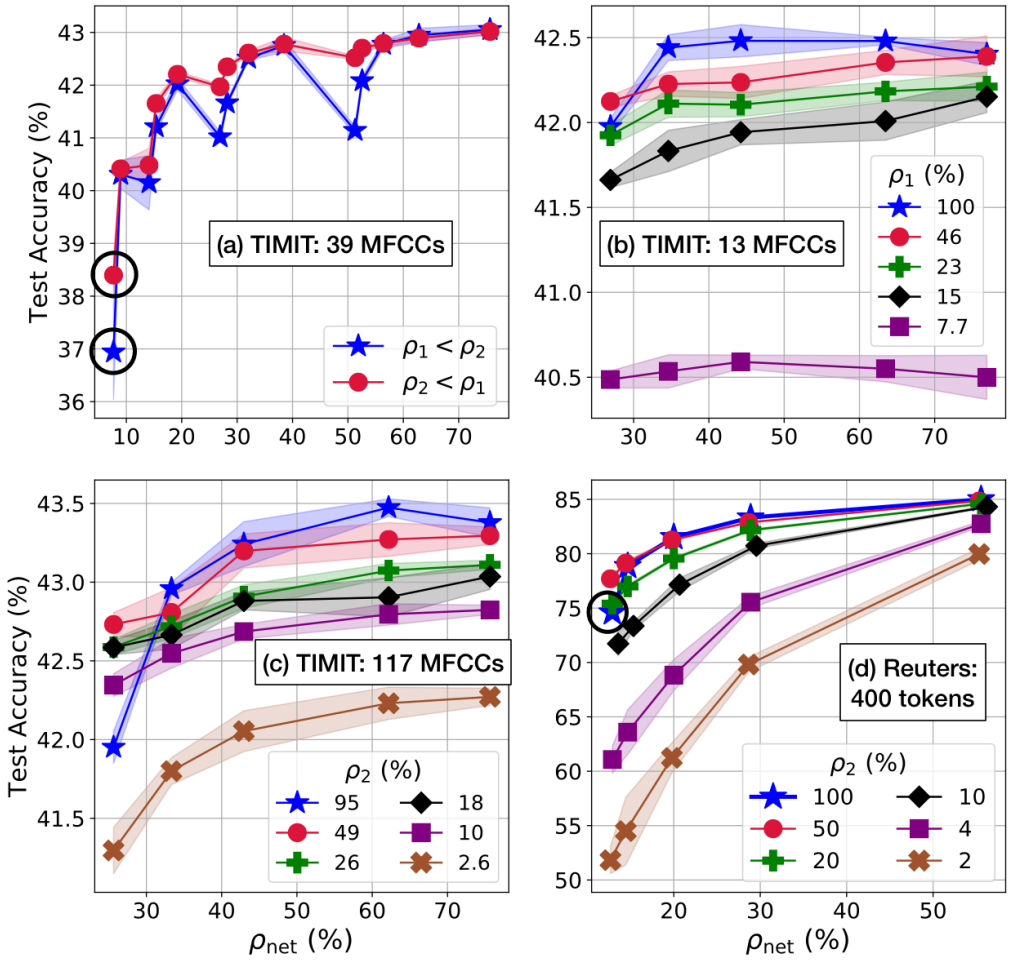


Figure 3.7: Comparison of classification accuracy as a function of ρ_{net} for: (a) TIMIT with 39 MFCCs for the two cases where one junction is always sparser than the other and vice-versa. Black-circled points show how reducing ρ_1 degrades performance to a greater extent. (b) TIMIT with 13 MFCCs for different ρ_1. (c,d) TIMIT with 117 MFCCs, and Reuters reduced to 400 tokens, for different ρ_2. $\boldsymbol{N}_{\text{net}}$ values are (a) $(39, 390, 39)$, (b) $(13, 390, 39)$, (c) $(117, 390, 39)$, (d) $(400, 50, 50)$.

the original Reuters dataset), notice the circled star-point at the very left of the $\rho_2 = 100\%$ curve. This point has very low ρ_1. Unlike Fig. 3.5(c), it crosses below the other curves, indicating that it is more important to have higher density in the first junction with this less redundant set of features.

In summary, if an individual junction density falls below a certain value, referred to as the *critical junction density*, it will adversely affect performance regardless of the density of other junctions. This explains why some of the curves cross in Fig. 3.7. The critical junction density is much smaller for earlier junctions than for later junctions in most datasets with sufficient redundancy. However, the critical density for earlier junctions increases for datasets with low redundancy.

3.3.4 'Large and sparse' vs 'small and dense' networks

As mentioned previously, NNs with lower values of ρ_{net} need lower values of the L2 penalty coefficient λ for regularization. This raises the question – why not design a small FC NN, *i.e.* one with a small number of nodes in its layers? This would also have a low number of parameters to begin with and would thus have low complexity.

We experimented with this aspect and observed that when keeping the total number of trainable parameters the same, sparser NNs with larger hidden layers (*i.e.* more neurons) generally performed better than denser networks with smaller hidden layers. This is true as long as the larger NN is not so sparse that individual junction densities fall below the critical density, as explained in Sec. 3.3.3. While the critical density is problem-dependent, it is usually low enough to obtain significant complexity savings above it. Thus, 'large and sparse' is better than 'small

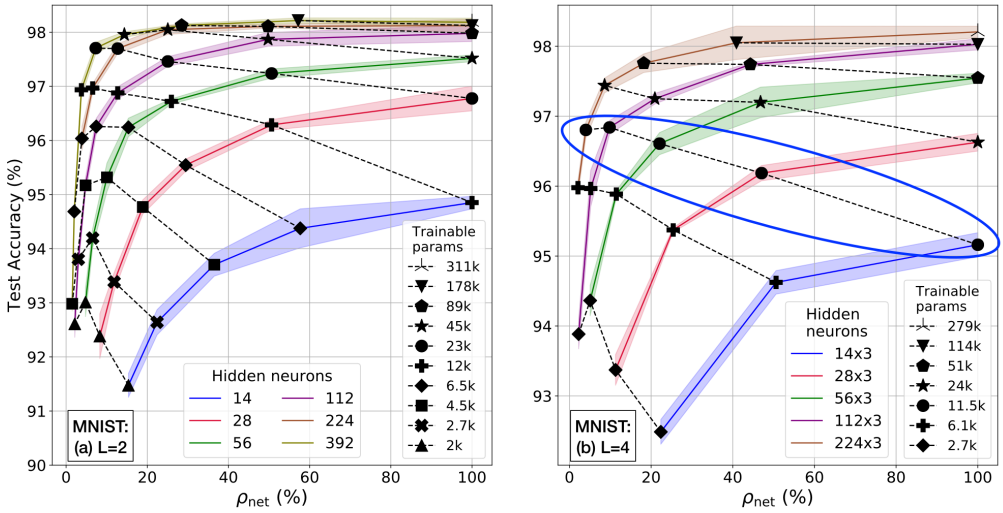


Figure 3.8: Comparing 'large and sparse' to 'small and dense' networks for MNIST with 784 features, with (a) $\boldsymbol{N}_{\text{net}} = (784, x, 10)$ (on the left), and (b) $\boldsymbol{N}_{\text{net}} = (784, x, x, x, 10)$ (on the right). Solid curves (with the shaded CIs around them) are for constant x, black dashed curves with same marker are for same number of trainable parameters. The final junction is always FC. Intermediate junctions for the $L = 4$ case have d^{out} values similar to junction 1.

and dense' for many practical cases, including NNs with more than one hidden layer (*i.e.* $L > 2$).

Fig. 3.8 shows this for networks having one (two) and three (four) hidden layers (junctions) trained on MNIST. For the four-junction network, all hidden layers have the same number of neurons. Each solid curve shows classification performance vs ρ_{net} for a particular $\boldsymbol{N}_{\text{net}}$, while the black dashed curves with identical markers are configurations that have approximately the same number of trainable parameters. As an example, the points with circular markers (with a big blue ellipse around them) in Fig. 3.8(b) all have the same number of trainable parameters and indicate that the larger, more sparse NNs perform better. Specifically, the network with $\boldsymbol{N}_{\text{net}} = (784, 112, 112, 112, 10)$ and $\boldsymbol{d}^{\text{out}}_{\text{net}} = (10, 10, 10, 10)$

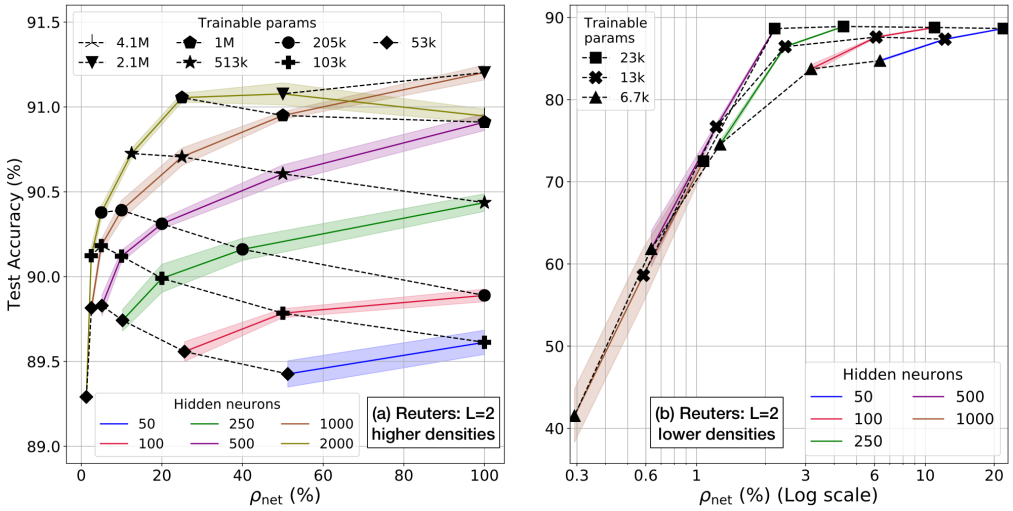


Figure 3.9: Comparing 'large and sparse' to 'small and dense' networks for Reuters with 2000 tokens, with $\boldsymbol{N}_{\text{net}} = (2000, x, 50)$. The x-axis is split into higher values on the left (a), and lower values on the right in log scale (b). Solid curves (with the shaded CIs around them) are for constant x, black dashed curves with same marker are for same number of trainable parameters. Junction 1 is sparsified first until its number of total weights is approximately equal to that of junction 2, then both are sparsified equally.

corresponding to $\rho_{\text{net}} = 9.82\%$ performs significantly better than the FC network with $\boldsymbol{N}_{\text{net}} = (784, 14, 14, 14, 10)$, and other smaller and denser networks, despite each having 11,500 trainable parameters. Increasing the network size further to $\boldsymbol{N}_{\text{net}} = (784, 224, 224, 224, 10)$, and reducing ρ_{net} to 4% to fix the number of trainable parameters at 11,500 leads to performance degradation. This is because this ρ_{net} was achieved by setting $\rho_2 = \rho_3 = 2.68\%$, which appears to be below the critical density.

Fig. 3.9 summarizes the analogous experiment on Reuters with similar conclusions. Both subfigures are for the same results with the x-axis split into higher and lower density range (on log scale), to show more detail. Observe that the trend of 'large and sparse' being better than 'small and dense' holds for subfigure (a), but

reverses for (b) since densities are very low (the black dashed curves have positive slope instead of negative). This is due to the critical density effect.

Fig. 3.10(a) shows the result for the same experiment on TIMIT with four hidden layers. The trend is less clearly discernible, but it exists. Notice how the black dashed curves have negative slopes at appreciable levels of ρ_{net}, indicating 'large and sparse' being better than 'small and dense', but high positive slopes at low ρ_{net}, indicating the rapid degradation in performance as density is reduced beyond the critical density. This is exacerbated by the fact that TIMIT with 39 MFCCs is a dataset with low redundancy, so the effects of very low ρ_{net} are better observed.

Fig. 3.10(b) for the MLP portion of CIFAR-100 shows similar results as TIMIT, but on a log x-scale for more clarity. As noted in Sec. 3.3.2, the best performance for a given $\boldsymbol{N}_{\text{net}}$ occurs at an overall density less than 100%. It appears that for any $\boldsymbol{N}_{\text{net}}$ for CIFAR-100, peak performance occurs at around 10–20% overall MLP density. We are intrigued by this trend, and plan to investigate further.

Some results have been omitted due to being qualitatively similar to those already shown. These are experiments on TIMIT with $L = 2$ (similar to 3.10(a)), Reuters with $L = 3$ (similar to 3.9), and CIFAR100 with the reduced redundancy having a single convolutional layer (similar to 3.10(b)).

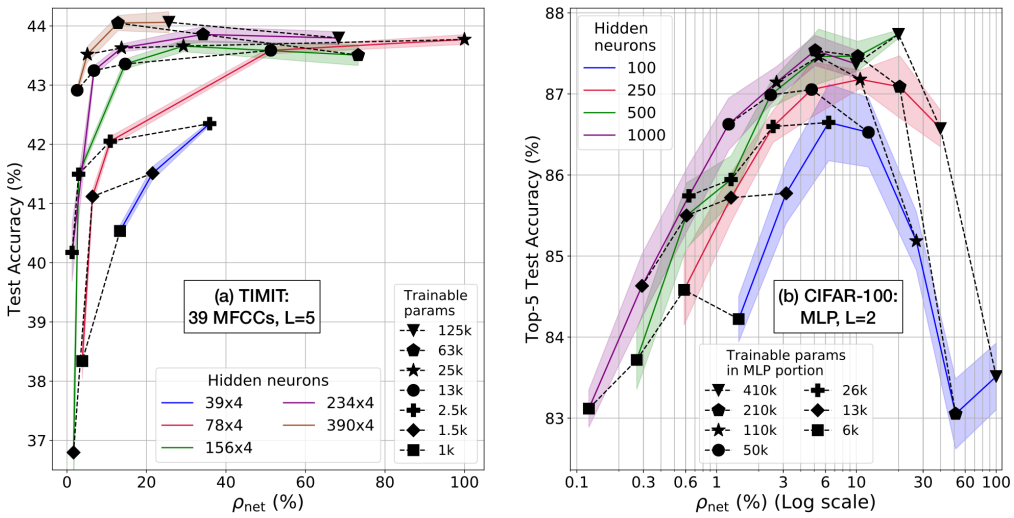


Figure 3.10: Comparing 'large and sparse' to 'small and dense' networks for (a) TIMIT with 39 MFCCs and $\boldsymbol{N}_{\text{net}} = (39, x, x, x, x, 39)$ (on the left), and (b) CIFAR-100 with the deep 9-layer CNN and MLP $\boldsymbol{N}_{\text{net}} = (4000, x, 100)$ with log scale for the x-axis (on the right). Solid curves (with the shaded CIs around them) are for constant x, black dashed curves with same marker are for same number of trainable parameters (in the MLP portion only for CIFAR). Since TIMIT has symmetric junctions, we tried to keep input and output junction densities as close as possible and adjusted intermediate junction densities to get the desired ρ_{net}. CIFAR-100 is sparsified in a way similar to Reuters in Fig. 3.9.

3.4 Summary

This chapter introduced the concept of pre-defined sparsity as a method of complexity reduction and presented performance results indicating its effectiveness and delineating trends to design such sparse NNs. Apart from published papers, our work is also available on Github [30]. The next chapter will present a concrete application of complexity reduction – a hardware architecture we designed and implemented to leverage pre-defined sparsity.

Chapter 4

Hardware Architecture

This chapter describes our proposed flexible hardware architecture which can leverage structured pre-defined sparsity to both train and test NNs of any complexity on-device. We begin with an overview of the key features, which are encapsulated in Fig. 4.1.

➤ **Definition 3:** ***Degree of parallelism*:** The degree of parallelism z_i is the number of edges processed in parallel in junction i.

Thus, we call our architecture **edge-based**. z_i determines the number of clock **cycles** taken to process a complete junction. Fig. 4.1(a) shows an example junction i with 6 edges processed using $z_i = 3$. The $z_i = 3$ blue edges are processed in cycle 0 (we begin numbering from 0), and the remaining $z_i = 3$ pink edges in cycle 1. A given hardware device can support some largest values of $\{z_i\}$, so NNs with more edges will simply take more cycles to process.

For a given processing operation in junction i, there are z_i computational units to perform it and z_i memories to store each quantity associated with it. This is shown in Fig. 4.1(b) for the FF operation, *i.e.* there are $z_i = 3$ computational

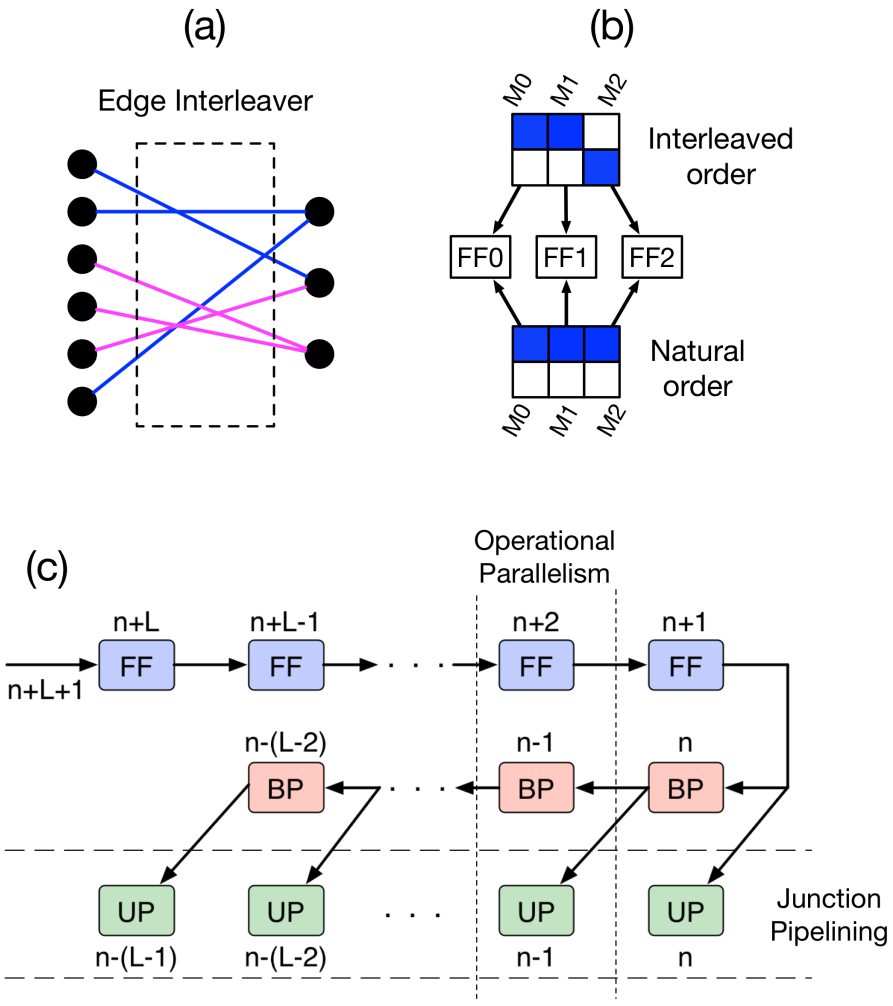


Figure 4.1: (a) Processing $z_i = 3$ edges in each cycle (blue in cycle 0, pink in cycle 1) for some junction i with a total of 6 edges. (b) Accessing $z_i = 3$ memories – M0, M1 and M2 shown as columns – from two separate banks, one in natural order (same address from each memory), the other in interleaved order. Clash-freedom is achieved by accessing only one element from each memory. The accessed values are fed to $z_i = 3$ processors to perform FF simultaneously. (c) Operational parallelism in each junction (vertical dotted lines denote processing for one junction), and junction pipelining of each operation across junctions (horizontal dashed lines) for different inputs.

units FF0–FF2. A challenge with this architecture is that in order to achieve high-throughput without memory duplication, the parallel memories must be accessed in two manners: **natural order** and **interleaved order**. Natural order accesses

memory elements sequentially. Interleaved order leads to the possibility of memory contention. This leads us to define:

➤ **Definition 4: *Clash*:** A clash arises if the same memory needs to be accessed multiple times in the same cycle, which may lead to stalls and/or wait states.[1]

➤ **Definition 5: *Clash-free*:** The clash-free condition occurs when the NN connection pattern and hardware architecture is designed such that no clashes occur during memory accesses.

Thus, the z_i computational units must access the memories such that no clashes and memory contention occur. Natural and interleaved order accesses are shown for $z_i = 3$ memories on the bottom and top of Fig. 4.1(b), respectively. Notice that there is only one shaded cell per column in cycle 0. The unshaded cells will be accessed in cycle 1. The concept of clash-freedom is tied to the connection patterns in the junctions, as will be expounded in Chapter 5.

Apart from clash-freedom to maximize throughput, some other characteristics regarding memory requirements guided us in developing the proposed architecture. Firstly, since weight memories are the largest (*i.e.* the number of weights is more than any other network parameter), their number should be minimized. Secondly, having a few deep memories is more efficient in terms of power and area than

[1]Some multi-ported memories can be accessed more than once in a cycle, however, clashes can still occur when trying to access more elements than the number of ports. To be more specific, a) for single-ported memories, attempting two reads or two writes or a read and a write in the same cycle is a clash, and b) for simple dual-ported memories with one port exclusively for reading and the other exclusively for writing, a read and a write can be performed in the same cycle, but attempting to perform two reads or two writes in the same cycle will result in a clash.

having many shallow memories [93]. These help to alleviate the key concern of large storage requirement when implementing NNs in hardware. Memory organization is described in Section 4.2.

In addition to edge processing in a given junction, our architecture is **pipelined across junctions**. Thus, the z_i values are selected to set the number of cycles required to process a layer to a constant – *i.e.* junctions with more weights have larger z_i so that the computation time of all junctions is the same. Furthermore, the **three operations associated with training are performed in parallel**, *i.e.* FF, BP, and UP. These concepts are shown in Fig. 4.1(c) and elaborated in Section 4.1. An $(L+1)$-layer NN has L junction pipeline stages so that the **throughput**, *i.e.* the frequency of processing input samples, is determined by the time taken to perform a single operation in a single junction.

Summary of architectural features

1. Edge-based, *i.e.* not tied to a specific number of nodes in a layer.

2. Flexible, *i.e.* the amount of logic is determined by the degree of parallelism which trades size for speed. This means that the architecture is also compatible with conventional FC junctions, as will be shown in Section 4.6.

3. Fully pipelined for the parallel operations associated with NN training. The architecture can also operate in inference only mode by eliminating the logic and memory associated with BP and UP, and the $\boldsymbol{h}'$ computations in (3.6c).

4.1 Junction pipelining and Operational parallelism

Our edge-based architecture is motivated by the fact that all three operations – FF, BP, UP – use the same weight values for computation. Since z_i edges are processed in parallel in a single cycle, the time taken to complete an operation in junction i is $C_i = |\boldsymbol{W}_i| / z_i$ cycles. The **degree of parallelism configuration** $\boldsymbol{z}_{\text{net}} = (z_1, \cdots, z_L)$ is chosen to achieve $C_i = C \quad \forall\, i \in \{1, \cdots, L\}$. This allows efficient junction pipelining since each operation takes exactly C cycles to be completed for each input in each junction, which we refer to as a **junction cycle**. This determines throughput.

The following is an analysis of Fig. 4.1(c) in more detail for an example NN with $L = 2$. While a new training input numbered $n + 3$ is getting loaded as $\boldsymbol{a}_0$, junction 1 is processing the FF stage for the previous input $n + 2$ and computing $\boldsymbol{a}_1$. Simultaneously, junction 2 is processing FF and computing cost $\boldsymbol{\delta}_L$ via cost

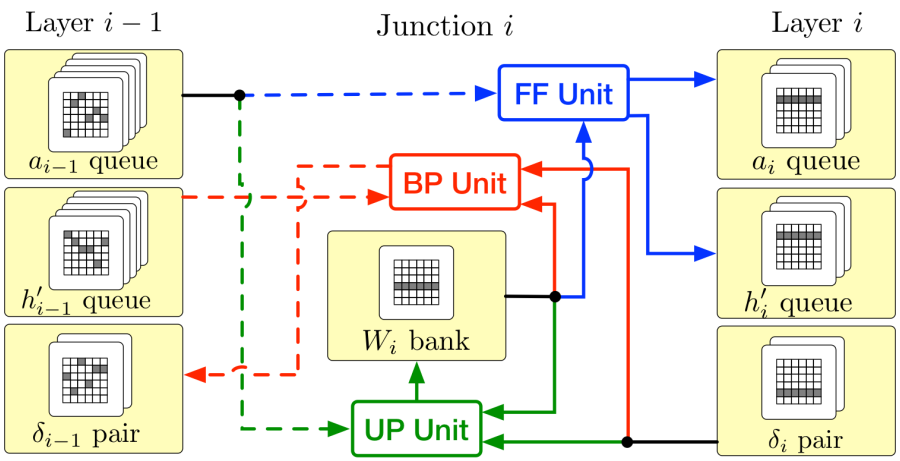


Figure 4.2: Architecture for parallel operations for an intermediate junction i ($i \neq 1, L$) showing the three operations along with associated inputs and outputs. Natural and interleaved order accesses are shown using solid and dashed lines, respectively. The $\boldsymbol{a}$ and $\boldsymbol{h}'$ memory banks occur as queues, the $\boldsymbol{\delta}$ memory banks as pairs, while there is a single weight memory bank.

derivatives for input $n+1$. It is also doing BP on input n to compute $\boldsymbol{\delta}_1$, as well as updating (UP) its parameters from the finished $\boldsymbol{\delta}_L$ computation of input n. Simultaneously, junction 1 is performing UP using $\boldsymbol{\delta}_1$ from the finished BP results of input $n-1$.[2] This results in *operational parallelism* in each junction, as shown in Fig. 4.2. The combined speedup is approximately a factor of $3L$ as compared to doing one operation at a time for a single input.

Notice from Fig. 4.2 that there is only one weight memory bank which is accessed for all three operations. However, UP in junction 1 needs access to $\boldsymbol{a}_0$ for input $n-1$, as per the weight update equation (3.8b). This means that there need to be $2L+1 = 5$ left activation memory banks for storing $\boldsymbol{a}_0$ for inputs $n-1$ to $n+3$,

[2]Note that BP does not occur in the first junction because there are no $\boldsymbol{\delta}_0$ values to be computed

Table 4.1: Hardware Architecture Total Storage Cost Comparison for $\boldsymbol{N}_{\text{net}} = (800, 100, 10)$ FC vs sparse with $\boldsymbol{d}_{\text{net}}^{\text{out}} = (20, 10)$, $\rho_{\text{net}} = 21\%$

Parameter	Expression	Count (FC)	Count (sparse)
$\boldsymbol{a}$	$\sum_{i=0}^{L-1} (2(L-i)+1) N_i$	4300	4300
$\boldsymbol{h}'$	$\sum_{i=1}^{L-1} (2(L-i)+1) N_i$	300	300
$\boldsymbol{\delta}$	$2\sum_{i=1}^{L} N_i$	220	220
$\boldsymbol{b}$	$\sum_{i=1}^{L} N_i$	110	110
$\boldsymbol{W}$	$\sum_{i=1}^{L} N_i d_i^{\text{in}}$	**81000**	**17000**
TOTAL	Σ (All above)	85930	21930

i.e. a **queue**-like structure. Similarly, UP in junction 2 will need $2(L-1)+1 = 3$ queued banks for each of its left activation $\boldsymbol{a}_1$ and its derivative $\boldsymbol{h}_1'$ memories – for inputs from n (for which values will be read) to $n+2$ (for which values are being computed and written). There also need to be 2 banks for all $\boldsymbol{\delta}$ memories – 1 for reading and the other for writing. Thus junction pipelining requires multiple memory banks, but only for layer parameters $\boldsymbol{a}$, $\boldsymbol{h}'$ and $\boldsymbol{\delta}$, *not* for weights. The number of layer parameters is insignificant compared to the number of weights for practical networks. This is why pre-defined sparsity leads to significant storage savings, as quantified in Table 4.1 for the circled FC point vs the $\rho_{\text{net}} = 21\%$ point from Fig. 3.2(c). Specifically, memory requirements are reduced by 3.9X in this case. Furthermore, the computational complexity, which is proportional to the number of weights for a MLP, is reduced by 4.8X. For this example, these complexity reductions come at a cost of degrading the classification accuracy from 98.0% to 97.2%.

4.2 Memory organization

For the purposes of memory organization, edges are numbered sequentially from top to bottom on the right side of the junction. Other network parameters such as $\boldsymbol{a}$, $\boldsymbol{h}'$ and $\boldsymbol{\delta}$ are numbered according to the neuron numbers in their respective layer. Consider Fig. 4.3 as an example, where junction i is flanked by $N_{i-1} = 12$ left neurons with $d_i^{\text{out}} = 2$ and $N_i = 8$ right neurons, leading to $|\boldsymbol{W}_i| = 24$ and $d_i^{\text{in}} = 3$. The three weights connecting to right neuron 0 are numbered 0, 1, 2; the next three connecting to right neuron 1 are numbered 3, 4, 5, and so on. A particular right neuron connects to some subset of left neurons of cardinality d_i^{in}.

Each type of network parameter is stored in a **bank of memories**. The example in Fig. 4.3 uses $z_i = 4$, *i.e.* 4 weights are accessed per cycle. We designed the weight memory bank to have the minimum number of memories to prevent clashes, *i.e.* z_i, and their depth equals C_i. Weight memories are read in natural order – 1 row per cycle (shown in same color).

Right neurons are processed sequentially due to the weight numbering. The number of right neuron parameters of a particular type needing to be accessed in a cycle is upper bounded by $\left\lceil z_i/d_i^{\text{in}} \right\rceil$, which leads to $z_{i+1} \geq \left\lceil z_i/d_i^{\text{in}} \right\rceil$ in order to prevent clashes in the right memory bank. (This does not limit most practical designs, as will be shown in Section 4.5). For FF in Fig. 4.3 for example, cycles 0 and 1 finish computation of $a_i^{(0)}$ and $a_i^{(1)}$ respectively, while cycle 2 finishes computing both $a_i^{(2)}$ and $a_i^{(3)}$. For BP or UP, everything remains same except for

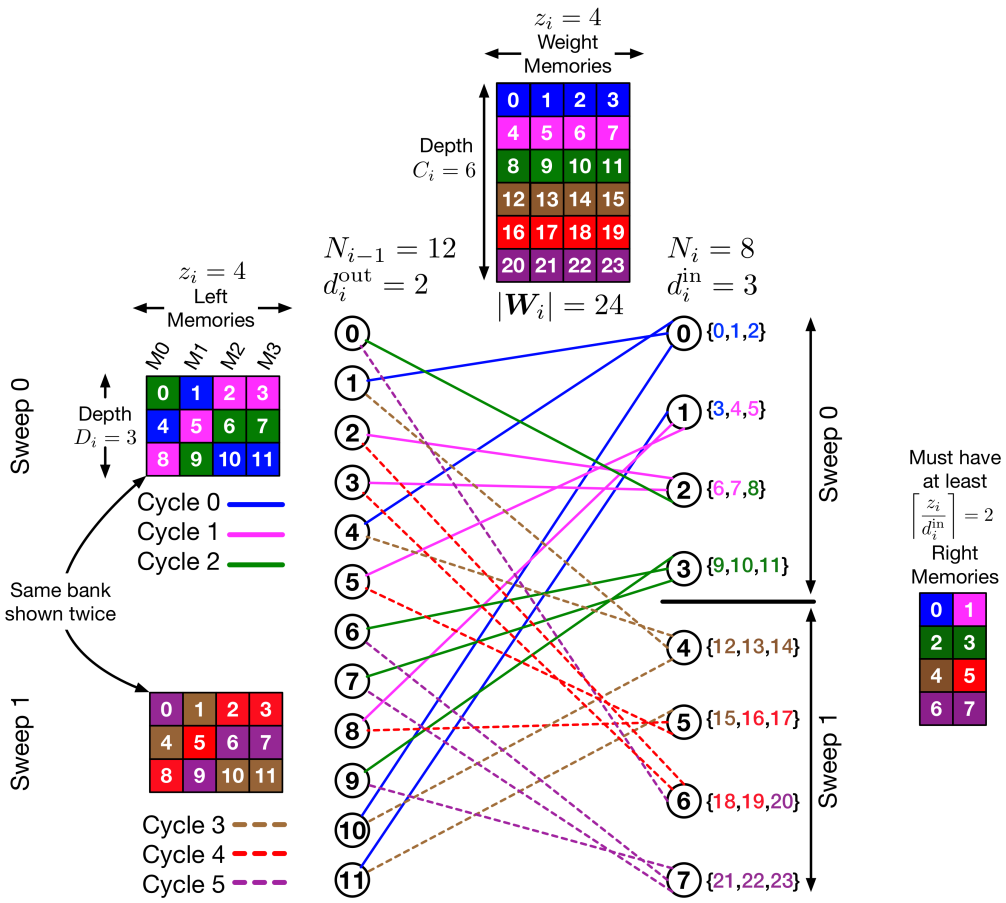


Figure 4.3: An example of processing inside junction i with $z_i = 4$ memories in the weight and left banks, and $z_{i+1} = 2$ memories in the right bank. The banks are represented as numerical grids, each column is a memory, and the number in each cell is the number of the edge / left neuron / right neuron whose parameter value is stored in it. Edge are sequentially numbered on the right (shown in curly braces). $z_i = 4$ weights are read in each of the six cycles, with the first three colored blue, pink and green, respectively. These represent sweep 0, while the next 3 (using dashed lines) colored brown, red and purple, respectively, represent sweep 1. Clash-freedom leads to at most one cell from each memory in each bank being accessed each cycle. Weight and right memories are accessed in natural order, while left memories are accessed in interleaved order.

the right memory accesses. Now $\delta_i^{(0)}$ and $\delta_i^{(1)}$ are used in cycle 0, $\delta_i^{(1)}$ and $\delta_i^{(2)}$ in cycle 1, and $\delta_i^{(2)}$ and $\delta_i^{(3)}$ in cycle 2. Thus the maximum number of right neuron parameters ever accessed in a cycle is $\left\lceil z_i / d_i^{\text{in}} \right\rceil = 2$.

Since edges are interleaved on the left, in general, the z_i edge processing logic units will need access to z_i parameters of a particular type from layer $i-1$. So all the left memory banks have z_i memories, each of depth $D_i = N_{i-1}/z_i$, which are accessed in interleaved order. For example, after D_i cycles, N_{i-1} edges have been processed – *i.e.* $(D_i \times z_i) = N_{i-1}$. We require that each of these edges be connected to a different left neuron to eliminates the possibility of duplicate edges. This completes a **sweep**, *i.e.* one complete access of the left memory bank. Since each left neuron connects to d_i^{out} edges, d_i^{out} sweeps are required to process all the edges, *i.e.* each left activation is read d_i^{out} times in the whole junction cycle. The reader can verify that D_i cycles multiplied by d_i^{out} sweeps results in C_i total cycles, *i.e.* one junction cycle.

4.3 Clash-freedom

We define a **clash** as attempting to perform a particular operation more than once on the same memory at the same time, which would stall processing. The idea of **clash-freedom** is to pre-define a pattern of connections and z values such that no operation in any junction of the NN results in a clash. Section 4.2 described how z values should be designed to prevent clashes in the weight and right memory banks.

This subsection analyzes the left memory banks, which are accessed in interleaved order. Their memory access pattern should be designed so as to prevent

clashes. Additionally, the following properties are desired for practical clash-free patterns. Firstly, it should be easy to find a pattern that gives good performance. Secondly, the logic and storage required to generate the left memory addresses should be low complexity.

We generate clash-free patterns by initially specifying the left memory addresses to be accessed in cycle 0 using a **seed vector** $\boldsymbol{\phi}_i \in \{0, 1, \cdots, D_i - 1\}^{z_i}$. Subsequent addresses are cyclically generated. Considering Fig. 4.3 as an example, $\boldsymbol{\phi}_i = (1, 0, 2, 2)$. Thus in cycle 0, we access addresses $(1, 0, 2, 2)$ from memories $(M0, M1, M2, M3)$, *i.e.* left neurons $(4, 1, 10, 11)$. In cycle 1, the accessed addresses are $(\boldsymbol{\phi}_i + 1)$ % $D_i = (2, 1, 0, 0)$, and so on. Since $D_i = 3$, cycles 3–5 access the same left neurons as cycles 0–2.

We found that this technique results in a large number of possible connection patterns, and randomly sampling from this set results in performance comparable with non-clash-free NNs. These ideas are further discussed in Chapter 5.

4.4 Batch size

As shown in (2.14), for a batch size of M, the UP operation in (3.8) is performed only once for M inputs by using the average over the M gradients. Our architecture performs an UP for every input and therefore may be viewed as having batch size one. However, the processing in our architecture differs from a typical software

implementation with $M = 1$ due to the pipelined and parallel operations. Specifically, in our architecture, FF and BP for the same input use different weights, as implied by Fig. 4.1(c). However, this deviation from the conventional notion of batch size did not result in performance degradation in our initial hardware implementation (presented in Section 4.7). We would also like to point out that there is considerable ambiguity in the literature regarding ideal batch sizes (cf. [94, 95]).

The concept of batch size for our architecture can be modified in several different ways. Firstly, by performing UP once every ($M > 1$) samples. As an example, a practical case for a deep MLP could be $M = 64$ and $L = 5$. For all such cases where $M >> L$, FF and BP will use the same weights for most inputs in a batch. Secondly, a more conventional minibatch update can also be obtained by completely removing the UP logic from the junction pipeline. After performing FF and BP on M samples, the pipeline will be flushed and the averaged gradients over M samples used to update all parameters in all junctions simultaneously. This would eliminate the UP arithmetic units from the pipeline, at the cost of increased storage for accumulating intermediate values from (3.8). However, we have not implemented these ideas in actual hardware yet.

4.5 Architectural Constraints

The depth of left memories in our hardware architecture is $D_i = N_{i-1}/z_i$. Thus N_{i-1} should preferably be an integral multiple of z_i. This is not a burdening

constraint since the choice of z_i is independent of network parameters and depends on the capacity of the device. In the unusual case that this constraint cannot be met, the extra cells in memories can be filled with dummy values such as 0.

There are also 2 conditions placed on the z_i values to eliminate stalls in processing: for all layers $i \in \{1, \cdots, L\}$, (i) $|\boldsymbol{W}_i| / z_i = C$, and (ii) $z_{i+1} \geq \lceil z_i / d_i^{\text{in}} \rceil$. Let us examine how much of a burden these constraints impose on choosing a network configuration. Without loss of generality, (i) can be written as:

$$\begin{aligned} \frac{|\boldsymbol{W}_{i+1}|}{z_{i+1}} &= \frac{|\boldsymbol{W}_i|}{z_i} \\ \Rightarrow \quad z_{i+1} &= \frac{|\boldsymbol{W}_{i+1}|\, z_i}{|\boldsymbol{W}_i|} \\ \Rightarrow \quad z_{i+1} &= \frac{z_i d_{i+1}^{\text{out}}}{d_i^{\text{in}}} \end{aligned} \tag{4.1}$$

Then (ii) becomes:

$$\begin{aligned} \frac{z_i d_{i+1}^{\text{out}}}{d_i^{\text{in}}} &\geq \left\lceil \frac{z_i}{d_i^{\text{in}}} \right\rceil \\ \Rightarrow \quad d_{i+1}^{\text{out}} &\geq \frac{d_i^{\text{in}}}{z_i} \left\lceil \frac{z_i}{d_i^{\text{in}}} \right\rceil \end{aligned} \tag{4.2}$$

which needs to be satisfied $\forall\ i \in \{1, \cdots, L-1\}$.

In practice, it is desirable to design z_i / d_i^{in} to be an integer so that an integral number of right neurons finish processing every cycle. This simplifies hardware implementation by eliminating the need for additional storage, for example, of the

intermediate activation values during FF. In this case, (4.2) reduces to $d_{i+1}^{\text{out}} \geq 1$, which is always true.

For non-integral z_i/d_i^{in}, there are two cases. If $z_i > d_i^{\text{in}}$, (4.2) reduces to $d_{i+1}^{\text{out}} \geq 2$. On the other hand, if $z_i < d_i^{\text{in}}$, there is no bound on the right hand side of (4.2). In general, note that (4.2) becomes a burdening constraint only if d_i^{in} is large, and d_{i+1}^{out} and z_i are both desired to be small. This corresponds to earlier junctions being denser than later, which is typically not desirable according to the observations in Sec. 3.3.3, or to very limited hardware resources. We thus conclude that (4.2) is not a limiting constraint in most practical cases.

4.6 Special Case: Processing a FC junction

Fig. 4.4 shows the FC version of the junction from Fig. 4.3, which has 96 edges to be accessed and operated on. This can be done keeping the same junction cycle $C_i = 6$ by increasing z_i to 16, *i.e.* using more hardware. On the other hand, if hardware resources are limited, one can use the same $z_i = 4$ and pay the price of a longer junction cycle $C_i = 24$, as shown in Fig. 4.4. *This demonstrates the flexibility of our architecture.*

Note that FC junctions are clash-free in all practical cases due to the following reasons. Firstly, the left memory accesses are in natural order just like the weights, which ensures that no more than one element is accessed from each memory per cycle. Secondly, $\left\lceil z_i/d_i^{\text{in}} \right\rceil = 1$ for all practical cases since $z_i \leq N_{i-1}$, as discussed

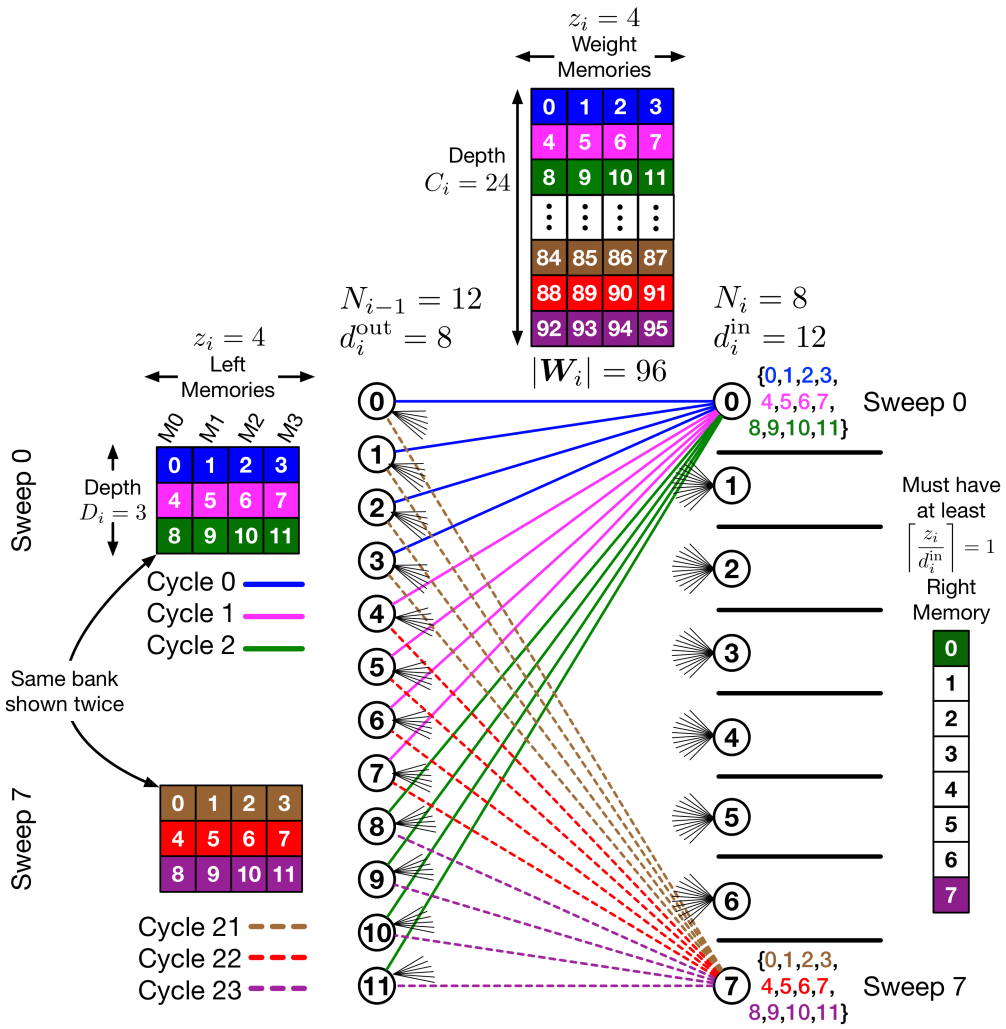


Figure 4.4: Processing the FC version of the junction from Fig. 4.3. For clarity, only the first 12 and last 12 edges (dashed) are shown, corresponding respectively to right neurons 0 and 7, sweeps 0 and 7, cycles 0–2 and 21–23.

in Section 4.5, and $d_i^{\text{in}} = N_{i-1}$ always for FC junctions. This means that at most one right neuron is processed in a cycle, so clashes will never occur when accessing the right memory bank. As an example, in Fig. 4.4, one right neuron finishes processing every 3rd cycle.

Note that compared to Fig. 4.3, the weight memories in Fig. 4.4 are deeper since C_i has increased from 6 to 24. However, the left layer memories remain the same size since $N_{i-1} = 12$ and $z_i = 4$ are unchanged, but the left memory bank is accessed more times since the number of sweeps has increased from 2 to 8. Also note that even if cycle 0 (blue) accesses some other clash-free subset of left neurons, such as $\{4, 5, 6, 7\}$ instead of $\{0, 1, 2, 3\}$, the connection pattern would remain unchanged. This implies that *different memory access patterns do not necessarily lead to different connection patterns*. This point will be discussed further in Chapter 5.

4.7 FPGA Implementation

This section describes our implementation of the architecture described thus far. Certain efforts in this implementation were achieved by Diandian Chen, – such as Universal asynchronous receiver-transmitter (UART) interfacing and observing outputs via Light Emitting Diodes (LEDs) – and hence have not been presented in detail in this dissertation.

We developed our implementation on a Digilent Nexys-4 DDR board containing a FPGA from the **Xilinx Artix-7** family with part number XC7A100T-1CSG324C [96]. This is a fairly modest FPGA in terms of capacity and performance, cf. the Xilinx Virtex UltraScale+ series [97] or the cloud-based FPGAs from Amazon Web Services [98] for more powerful alternatives. However, our

purpose was to obtain an initial proof-of-concept in which we could explore efficient design styles and optimize the Verilog Register Transfer Level (RTL) code to make it more robust and scalable. The Artix-7 device serves this purpose. We are currently associated with the Scalable Acceleration Platform Integrating Reconfigurable Computing and Natural Language Processing Technologies (SAPIENT) team and the University of Southern California Information Sciences Institute (USC ISI) in extending the RTL to run on more powerful FPGAs. The development was achieved using the Xilinx Vivado software environment.

Our initial proof-of-concept implementation used the MNIST dataset. We used powers of 2 for most network configuration values to simplify the hardware realization. Accordingly we padded each input with 0s to extend the number of features from 784 to 1024 (this does not alter performance, as mentioned in Section 3.3.1), and each ground-truth output with 0s to extend the number of labels from 10 to 32. Note that the output encoding remains one-hot.

4.7.1 Network Configuration and Training Setup

We implemented a structured pre-defined sparse NN with $\boldsymbol{N}_{\text{net}} = (1024, 64, 32)$ and $\boldsymbol{d}_{\text{net}}^{\text{out}} = (4, 16)$, leading to $\rho_{\text{net}} = 7.576\%$. The complete configuration is given in Table 4.2. The values shown were chosen on the basis of hardware constraints and experimental results, which will be described in Sections 4.7.2 and 4.7.3.

Table 4.2: Implemented Network Configuration

Junction Number (i)	1	2
Left Neurons (N_{i-1})	1024	64
Right Neurons (N_i)	64	32
Out-degree (d_i^{out})	4	16
Number of Weights ($\lvert \boldsymbol{W}_i \rvert = N_{i-1} \times d_i^{\text{out}}$)	4096	1024
In-degree ($d_i^{\text{in}} = W_i/N_i$)	64	32
z_i	128	32
Junction cycle ($\lvert \boldsymbol{W}_i \rvert / z_i$) [a]	32	32
Density ($\rho_i = W_i/(N_{i-1}N_i)$)	6.25%	50%
Overall Density	$\rho_{\text{net}} = 7.576\%$	

[a]In terms of number of clock cycles. Not considering the additional clock cycles needed for memory accesses.

We selected 12,544 inputs to comprise 1 epoch of training. Learning rate (η) is initially 2^{-3}, decay is implemented by halving after the first 2 epochs, then after every 4 epochs until its value became 2^{-7}. Designing η to always be a power of 2 leads to the multiplications with η in eq. (3.8) getting reduced to bit shifts. Predefined sparsity leads to a total number of trainable parameters = $(\lvert \boldsymbol{W}_1 \rvert = 4096) + (\lvert \boldsymbol{W}_2 \rvert = 1024) + (\lvert \boldsymbol{b}_1 \rvert = N_1 = 64) + (\lvert \boldsymbol{b} \rvert_2 = N_2 = 32) = 5216$, which is much less than 12,544, so we theorized that overfitting was not an issue. We verified this using software simulations, and hence did not apply any regularization.

4.7.2 Bit Width Considerations

Parameter Initialization

We initialized weights using the Glorot Normal technique (2.16), which translates to a three standard deviation range of ± 0.51 for junction 1 and ± 0.61 for junction 2 in our NN configuration described in Table 4.2.

The biases in our architecture are stored along with the weights as an augmentation to the weight memory banks. So we initialized biases in the same manner as weights. Software simulations showed that this led to no degradation in performance from the conventional method of initializing biases with constant values such as 0 or 0.1. This makes sense since the maximum absolute value from initialization is much closer to 0 than their final values when training converges, as shown in Fig. 4.5.

To simplify the RTL, we used the same set of $|\boldsymbol{W}_i| / z_i$ unique values to initialize all weights and biases in junction i. Again, software simulations showed that this led to no degradation in performance as compared to initializing all of them randomly. This is not surprising since an appropriately high value of initial learning rate will drive each weight and bias towards its own optimum value, regardless of similar values at the start.

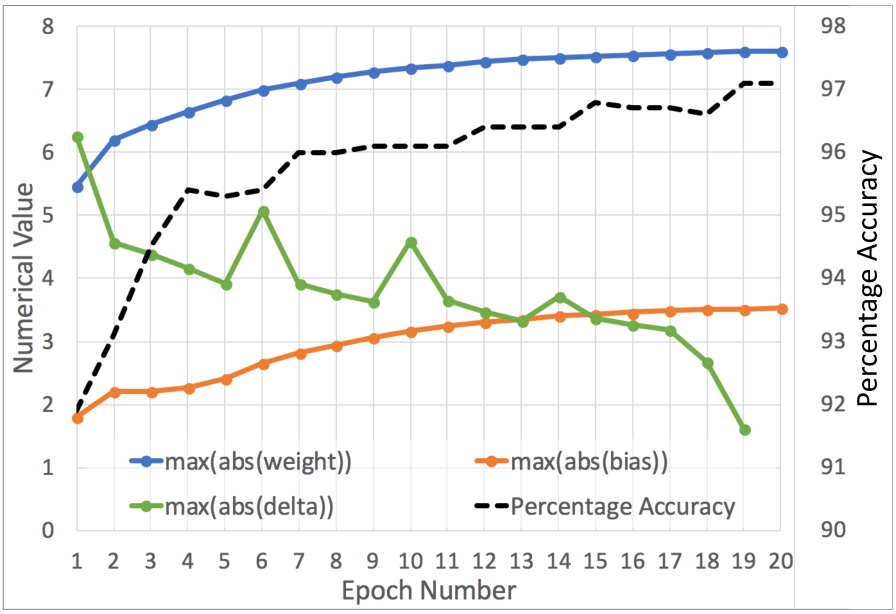


Figure 4.5: Maximum absolute values (left y-axis) for all weights, biases, and deltas in the NN, and percentage classification accuracy (right y-axis), as the NN is trained.

Fixed Point Configuration

We recreated the aforementioned initial conditions in software and trained our configuration to study the range of values for network variables until convergence. The results for are shown in Fig. 4.5 for all weights, biases and delta values. We used the **sigmoid** activation function for our hardware implementation, which has output range $= [0, 1]$.

To keep the hardware optimal, we decided on the **same fixed point bit configuration** for all computed values of network parameters — $\{\boldsymbol{a}_i, \boldsymbol{h}'_i, \boldsymbol{\delta}_i, \boldsymbol{W}_i, \boldsymbol{b}_i\}$, $i \in \{1, 2\}$. This uses less hardware resources compared to implementations such as a) [60], which uses fixed point adders, but more resource-intensive floating point

multipliers and floating-to-fixed-point converters, or b) [59], which incurs additional Digital Signal Processing (DSP) resources due to its use of probabilistic fixed point rounding techniques.

Our configuration is characterized by the bit triplet (b_w, b_n, b_f), which are respectively the total number of bits, integer bits, and fractional bits, with the constraint $b_w = b_n + b_f + 1$, where the 1 is for the sign bit. This gives a numerical range of $[-2^{b_n}, 2^{b_n} - 2^{-b_f}]$ and precision of 2^{-b_f}. Fig. 4.5 shows that the maximum absolute values of various network parameters during training stays within 8. Accordingly we set $b_n = 3$. We then experimented with different values for the bit triplet and obtained different utilizations for the FPGA Look Up Table (LUT) (*i.e.* logic) resources. The results are shown in Table 4.3. For this hardware implementation, we measured accuracy as classification performance on the last 1000 training samples[3]. Noting the diminishing returns and impractical utilization of hardware resources for high bit widths, we chose the bit triplet $(12, 3, 8)$ as being the optimal case, *i.e.* our results are from a **12-bit implementation**. Later on we will also discuss a 10-bit implementation.

[3]This differs from the conventional method of measuring performance as classification accuracy on the test set. We chose the last 1000 training samples to measure performance on in order to simplify the measurement techniques for this initial proof-of-concept implementation.

Table 4.3: Effect of Bit Width on Performance

b_w	b_n	b_f	FPGA LUT Utilization %	Accuracy after 1 epoch	Accuracy after 15 epochs
8	2	5	37.89	78	81
10	2	7	72.82	90.1	94.9
10	3	6	63.79	88	93.8
12	3	8	83.38	90.3	96.5
16	4	11	112	91.9	96.5

Dynamic Range Reduction due to Sparsity

We found that sparsity leads to reduction in the dynamic range of network parameters, since the summations in (3.6) and (3.7) are over smaller ranges. This motivated us to use a special form of adder and multiplier which preserves the bit triplet between inputs and outputs by clipping large absolute values of output to either the positive or negative maximum allowed by the range. For example, 10 would become 7.996 and -10 would become -8.

Fig. 4.6 analyzes the worst clipping errors by comparing the absolute values of the argument of the sigmoid function in the hidden layer, *i.e.* $\boldsymbol{s}_1 = \sum \boldsymbol{W}_1 \boldsymbol{a}_0 + \boldsymbol{b}_1$ from (3.6a), for our sparse case vs. the corresponding FC case (which would have $d_1^{\text{out}} = 64$, $d_2^{\text{out}} = 32$). Notice that the sparse case only has 17% of its values clipped due to being outside the dynamic range afforded by $b_n = 3$, while the FC case has 57%. The sparse case also has a smaller variance. This implies that the hardware errors introduced due to finite bit-width effects are less pronounced for our pre-defined sparse configuration as compared to FC.

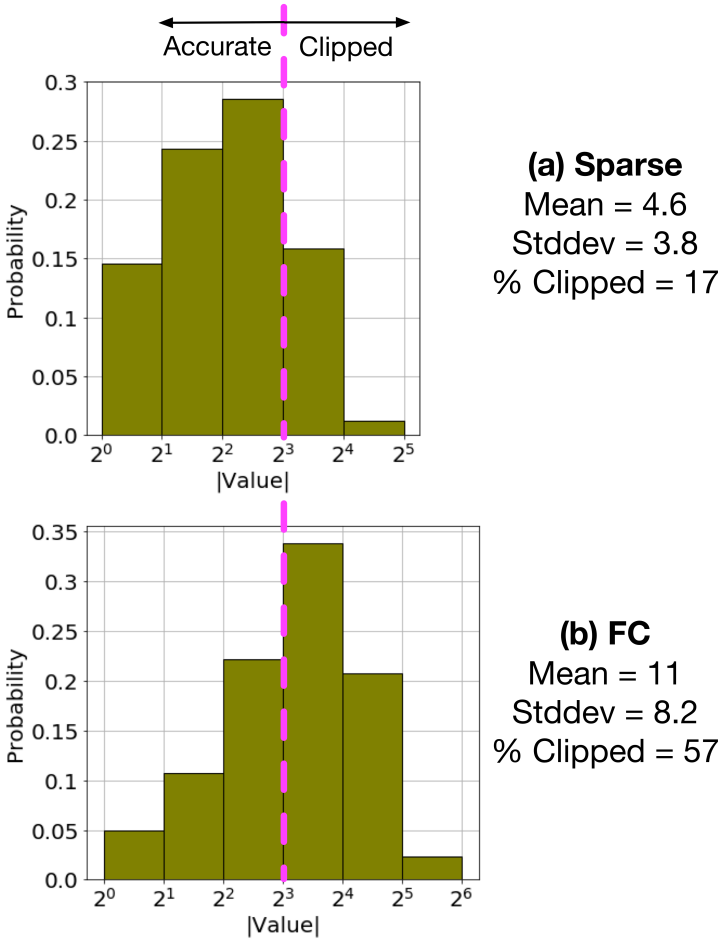


Figure 4.6: Histograms of absolute value of $\boldsymbol{s}_1$ with respect to dynamic range for (a) sparse vs. (b) fully connected cases, as obtained from software simulations. Values right of the pink line are clipped.

Why we chose sigmoid activations?

As has been amply demonstrated in literature (*e.g.* [1, 12, 45]), the ideal ReLU activation function ((2.8)) is more widely used than sigmoid ((2.9)) due to the former's better performance. This is primarily due to elimination of the vanishing gradient problem. Note also that ReLU has a tendency towards generating sparse outputs, *i.e.* several nodes output zero values for activation.

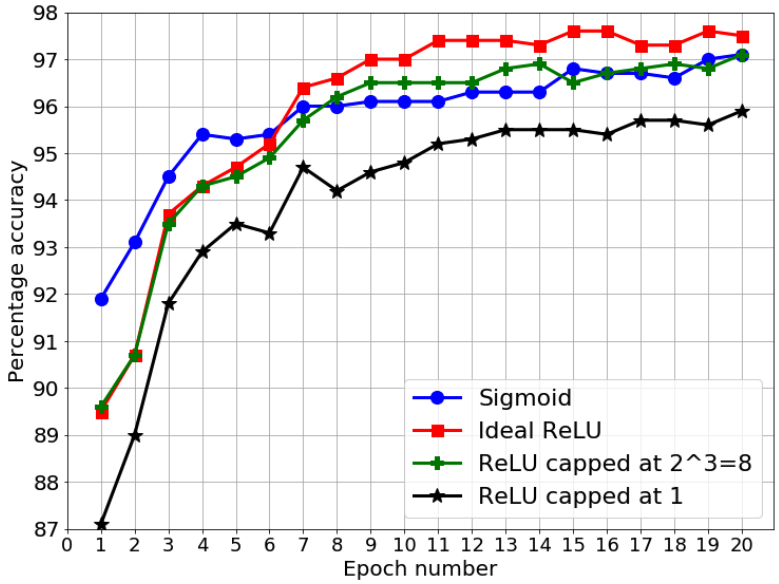


Figure 4.7: Comparison of activation functions for $\boldsymbol{a}_1$, as obtained from software simulations.

However, ideal ReLU is not practical for hardware due to its unbounded range. We experimented with a modified form of ReLU where the outputs were clipped to a) 8, which is the maximum supported by $b_n = 3$, and b) 1, to preserve bit width consistency in the multipliers and adders and ensure compatibility with sigmoid activations. Fig. 4.7 shows software simulations comparing sigmoid with these cases. Note that ReLU clipped at 8 converges similar to sigmoid, but sigmoid has better initial performance. Also, there is no need to promote extra sparsity by using ReLU because our configuration is already sparse. Finally, in our hardware implementation, sigmoid does not suffer from vanishing gradient problems due to just having two junctions. We therefore concluded that sigmoid activation is the

best choice for *all* layers. For the output layer, we simply picked the correct class as corresponding to the output node with the maximum sigmoid output value.

4.7.3 Implementation Details

Sigmoid Activation

The sigmoid function uses exponentials, which are computationally infeasible to obtain in hardware. So we pre-computed the values of $\sigma(\cdot)$ and $\sigma^{'}(\cdot)$ and stored them in LUTs. Interpolation was not used, instead we computed sigmoid for all 4096 possible 12-bit arguments up to the full 8 fractional bits of accuracy. On the other hand, its derivative values were computed to 6 fractional bits of accuracy since they have a range of $[0, 2^{-2}]$. The number of sigmoid LUTs required is $\sum_{i=1}^{L} z_i/d_i^{\text{in}} = 3$.

Arithmetic Units

As indicated in Table 4.2, we designed z_i/d_i^{in} to be an integer for both junctions, which ensures that in every cycle, an integral number of right neurons finish their FF processing. This implies that the FF summations in (3.6a) can occur in a single cycle, and eliminates the need for storing partial sums. The total number of multipliers required for FF is $\sum_{i=1}^{L} z_i = 160$. The summations also use a tree adder of depth $= \log_2\left(d_i^{\text{in}}\right)$ for every neuron processed in a cycle.

The BP summation in (3.7b) will need several cycles to complete for a single left neuron since weight numbering is permuted on the left. This necessitates storing $\sum_{i=2}^{L} z_i$ partial sums, however, tree adders are no longer required. Note that (3.7b) has 2 multiplications, and moreover, BP does not occur in the first junction, so the total number of multipliers required for BP is $2\sum_{i=2}^{L} z_i = 64$.

The UP operation in each junction i requires z_i adders for the weights and z_i/d_i^{in} adders for the biases, since that many right neurons are processed every cycle. Only the weight update requires multipliers, so their total number is $\sum_{i=1}^{L} z_i = 160$.

Our FPGA device has 240 DSP blocks. Accordingly, we implemented the 224 FF and BP multipliers using 1 DSP for each, while the other 160 UP multipliers and all adders were implemented using logic.

Memories and Data

All memories were implemented using Block Random Access Memory (BRAM). The memories for $\boldsymbol{a}$ and $\boldsymbol{h}'$ never need to be read from and written into in the same cycle, so they are single-port. The $\boldsymbol{\delta}$ memories are true dual-port, i.e. both ports support reads and writes. This is required due to their read-modify-write nature, since they accumulate partial sums. The 'weight+bias' memories are simple dual-port, with one port used exclusively for reading the kth cell in cycle k, and the other for simultaneously writing the $(k-1)$th cell. These were initialized using Glorot normal values while all other memories were initialized with zeroes.

Note that regardless of access order, the logic to compute memory addresses simply consists of z_i incrementers. This is because natural order accesses occur sequentially, while interleaved order accesses also occur sequentially according to the seed vector. A more complete description of clash freedom is deferred to Chapter 5.

The ground truth one-hot encoding for all 12,544 inputs were stored in a single-port BRAM, and initialized with word size = 10 to represent the 10 MNIST outputs. After reading, the word was padded with 0s to make it 32-bit long. On the other hand, the input data was too big to store on-chip. Since the native MNIST images are $28 \times 28 = 784$ pixels, the total input data size is $12544 \times 784 \times 8 = 78.68$ Mb, while the total device BRAM capacity is only 4.86 Mb. So the input data was fed from a host computer using the UART interface.

Network Configuration

Here we explain the choice of NN configuration in Table 4.2. We initially picked $N_2 = 16$, which is the minimum power of 2 above 10. Since later junctions need to be denser than earlier ones to improve performance, we experimented with junction 2 density and show its effects on performance in Fig. 4.8. We concluded that 50% density is optimum for junction 2. Attempting to meet the architectural constraints in Section 4.5 led to $z_1 = 256$, which was beyond the capacity of our FPGA. So we increased N_2 to 32 and set z_2 to the minimum value of 32, leading

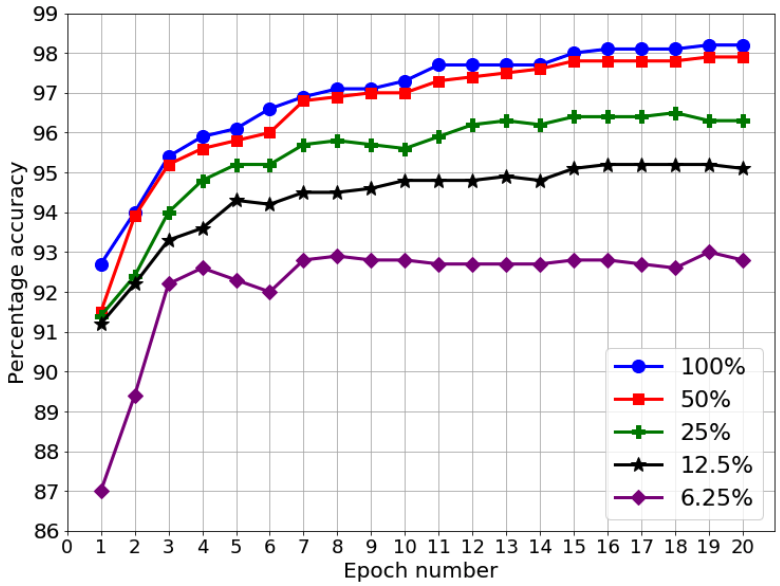


Figure 4.8: Performance for different ρ_2, keeping ρ_1 fixed at 6.25%, as obtained from software simulations.

to $z_1 = 128$. We also experimented with $d_1^{\text{out}} = 8$, but the resulting accuracy was within 1 percentage point of our final choice of $d_1^{\text{out}} = 4$.

Results

We stored the results of several training inputs and fed them out to 10 LEDs on the board, each representing an output from 0-9. The FPGA implementation performed according to RTL simulations and close to the ideal floating point software simulations, giving 96.5% accuracy in 14 epochs of training. Additionally, we also used Vivado to test a 10-bit implementation corresponding to the bit triplet $(10, 2, 7)$ – this yielded 95% accuracy. Fig. 4.9 shows our FPGA implementation in action.

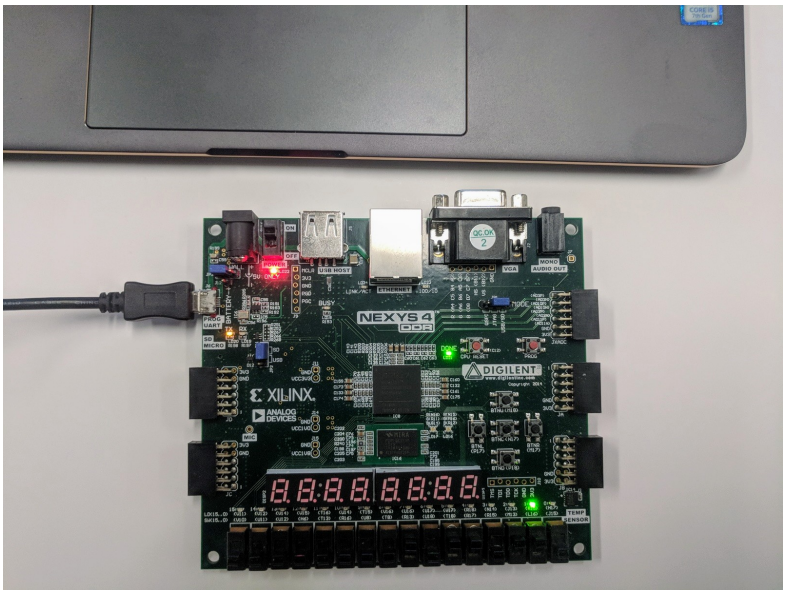


Figure 4.9: Our design working on the Xilinx XC7A100T-1CSG324C FPGA.

Timing and Additional Pipelining

A junction cycle in our design is actually $(|\boldsymbol{W}_i| / z_i + 2) = 34$ clock cycles since each set of z_i weights in a junction need a total of 3 clock cycles for each operation, as explained in Fig. 4.10. To summarize, the first and third are used to compute memory addresses, while the second is the most time-consuming since it performs most of the arithmetic computations. This determines our clock frequency, which is 15MHz. For the additional 10-bit case, the clock frequency is 20MHz.

One way to improve the clock frequency is to insert pipeline stages in the middle cycle of the three required for each operation, as shown in Fig. 4.10 (this is not to be confused with junction pipelining). There is, however, an issue here. The calculation of $\boldsymbol{\delta}$ values for BP is done by reading the $\boldsymbol{\delta}$ memories, modifying values, and writing back. This read-modify-write operation takes place multiple times in

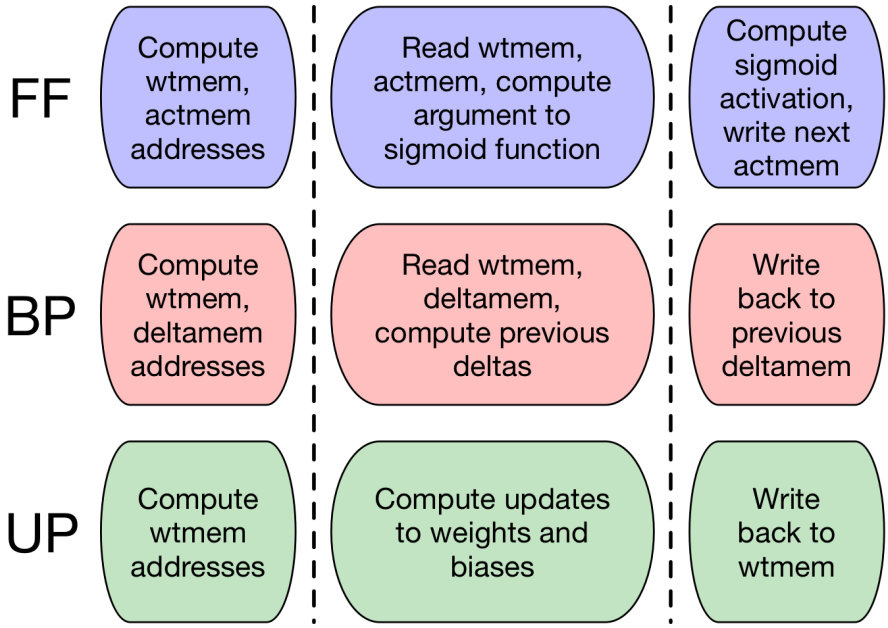


Figure 4.10: Breaking up each operation into 3 clock cycles.

a junction cycle, and the memory addresses accessed are repeated in cycles. Since the depth of the $\boldsymbol{\delta}$ memories is N_{i-1}/z_i, the read-modify-write operation must finish in $N_{i-1}/z_i - 1$ cycles to preserve fidelity of the $\boldsymbol{\delta}$ values. Thus, the maximum number of extra pipeline stages than can be inserted is $N_{i-1}/z_i - 2$.

In Table 4.2 for example, no pipeline stages can be inserted since junction 2 has $N_{i-1}/z_i = 64/32 = 2$. We attempted to insert 2 extra pipeline stages and, although the clock frequency went up from 15 MHz to 25 MHz for the 12-bit case and from 20 MHz to 30 MHz for the 10-bit case, the accuracy dropped (expectedly, since the $\boldsymbol{\delta}$ values are not accurate) to 88% for 12-bit and 85% for 10-bit.

To properly experiment with the introduction of extra pipeline stages, we implemented a new network configuration with $\boldsymbol{N}_{\text{net}} = (1024, 128, 32)$, $\boldsymbol{d}_{\text{net}}^{\text{out}} = (4, 8)$,

i.e. $\boldsymbol{d}_{\text{net}}^{\text{in}} = (32, 32)$, and $\boldsymbol{z}_{\text{net}} = (128, 32)$. This results in $N_{i-1}/z_i = 4$ for junction 2, so 2 extra pipeline stages can be inserted. We inserted these and the performance was exactly the same as the non-pipelined case, while the clock frequency improved as described above – 25 MHz for the 12-bit case and 30 MHz for the 10-bit case.

4.8 Summary

This chapter described our proposed hardware architecture and detailed an initial FPGA implementation we developed. While this is a good initial proof-of-concept, there are significant engineering tasks required to demonstrate state-of-the-art training speeds. However, we have characterized possible pipelining options to speed up the design and are currently working with other hardware teams to further develop this work.

Chapter 5

Connection Patterns

A fully connected neural network consists of fully connected junctions, *i.e.* every node in a layer connects to every node in its previous and next layer. This gives rise to a single possible **connection pattern**. On the other hand, a structured pre-defined sparse NN with a fixed $\boldsymbol{N}_{\text{net}}$ and $\boldsymbol{d}_{\text{net}}^{\text{out}}$ has several different ways of connecting the nodes in each junction. Fig. 5.1 illustrates this for a single junction with 4 left nodes and 2 right nodes. Subfigure (a) shows the FC case, while subfigures (b)–(g) show the six possible ways to arrange the connections for the 50% density case.

In practice, we desire connection patterns which give good performance and are hardware friendly, *i.e.* lead to clash-free memory accesses and are simple to implement. The remainder of this chapter will elaborate on these ideas.

5.1 Biadjacency Matrices

First we will introduce a useful concept. The biadjacency matrix for a bipartite graph is a rectangular matrix with 1 and 0 denoting the presence and absence of an edge, respectively [99]. Accordingly:

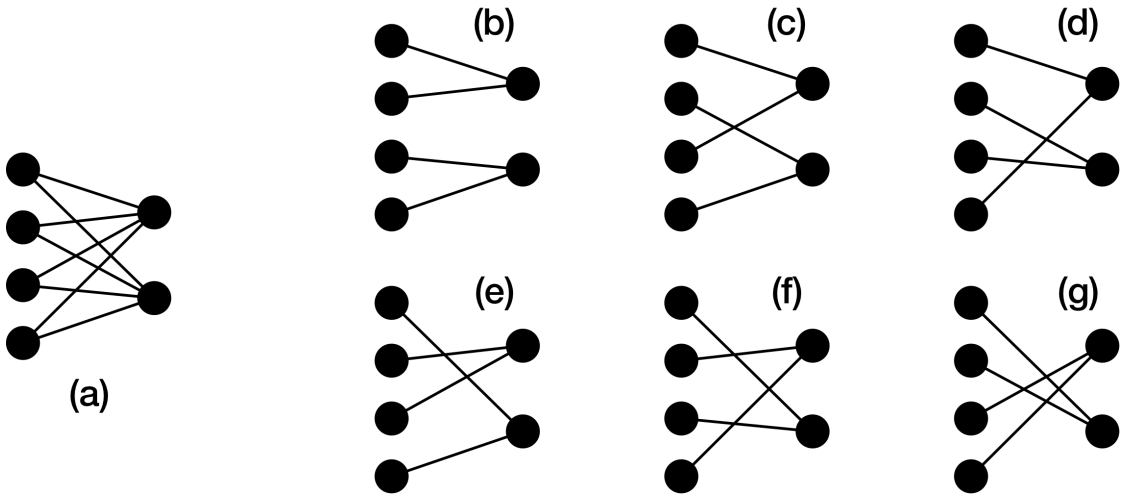


Figure 5.1: Different connection patterns for a single-junction network with $\boldsymbol{N}_{\text{net}} = (4, 2)$. (a) The FC case, with just one possible connection pattern. (b)–(g) Six different ways of arranging connections when $\boldsymbol{d}^{\text{out}}_{\text{net}} = (1)$, *i.e.* $\boldsymbol{d}^{\text{in}}_{\text{net}} = (2)$. Notice that in each case, each left node has exactly one outgoing connection and each right node has exactly two incoming connections.

➤ **Definition 6:** ***Biadjacency matrix*:** The biadjacency matrix $\underset{N_i \times N_{i-1}}{\boldsymbol{\mathcal{B}}_i}$ for junction i is defined such that $\mathcal{B}_i^{(j,k)} = 1$ if an edge exists between node k in layer $i-1$ and node j in layer i, otherwise $\mathcal{B}_i^{(j,k)} = 0$.

Then, if junction i is structured pre-defined sparse, $\boldsymbol{\mathcal{B}}_i$ has exactly d_i^{in} 1s in each row and exactly d_i^{out} 1s in each column, *i.e.* :

$$\mathbf{1}^T \boldsymbol{\mathcal{B}}_i = d_i^{\text{out}} \mathbf{1}^T \tag{5.1a}$$

$$\boldsymbol{\mathcal{B}}_i \mathbf{1} = d_i^{\text{in}} \mathbf{1} \tag{5.1b}$$

where **1** is the vector of all 1s with size set by the context. Moreover, the density can be computed as the fraction of 1s in the biadjacency matrix:

$$\rho_i = \frac{\sum_{j=1}^{N_i} \sum_{k=1}^{N_{i-1}} \boldsymbol{\mathcal{B}}_i^{(j,k)}}{N_i N_{i-1}} \tag{5.2}$$

Now we attempt to tackle the question of the number of possible connection patterns in a junction i, which we denote as G_i^{cp}. We saw that the single junction in Fig. 5.1 with the structured constraints $d_i^{\text{out}} = 1$ and $d_i^{\text{in}} = 2$ had $G_i^{\text{cp}} = 6$. These are the six possible (2×4) $\{0, 1\}$-only matrices with each row summing to 2 and each column summing to 1, as given below:

$$\left\{ \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \right\}$$

Generalizing this, we can state:

Theorem 2

Given a junction i with N_{i-1}, N_i and ρ_i (from which d_i^{out} and d_i^{in} can be computed), the number of possible connection patterns it can have, G_i^{cp}, is the same as the number of possible matrices of dimensions $N_i \times N_{i-1}$

consisting only of $\{0, 1\}$ entries such that sum of each row equals d_i^{in} and sum of each column equals d_i^{out}.

Moreover, the number of possible connection patterns in a complete L-junction NN is

$$G_{\text{net}}^{\text{cp}} = \prod_{i=1}^{L} G_i^{\text{cp}} \tag{5.3}$$

Finding the exact value of G_i^{cp} for any junction may not be feasible, cf. [100]. However, computing the exact value is usually of little interest. As will be discussed subsequently, finding a class of connection patterns with certain desirable properties is of more importance.

5.2 Clash-free memory access patterns

Section 4.3 introduced the idea of a clash as attempting to perform a particular operation more than once on the same memory at the same time, which would stall processing. For convenience in the upcoming discussion, Fig. 4.3 is repeated as Fig. 5.2.

Our edge-based architecture described in Chapter 4 is inspired by architectures proposed for iterative decoding of modern sparse-graph-based error correction codes (*i.e.* Turbo and Low Density Parity Check (LDPC) codes) (cf., [101, 102]).

For modern codes, the clash-free property of the memories is ensured by defining clash-free interleavers (*i.e.* permutations) [103], or clash-free parity check matrices [102]. Similarly in our architecture, the connection pattern in a junction is defined by an interleaver, *i.e.* a mapping from sequentially numbered edges on the right to permuted numbering on the left. This leads to interleaved order access of the left memory banks, and the possibility of clashes. We defined a simple algorithm to prevent this from happening – start with a seed vector $\boldsymbol{\phi}_i$ specifying the left memory addresses to be accessed in cycle 0, then perform $(\boldsymbol{\phi}_i + 1)$ % D_i for every subsequent cycle. This defines a certain left **memory access pattern** for junction i, and can be visualized using the shaded left memories in Fig. 5.2. Similar to connection patterns, we define G_i^{map} and $G_{\text{net}}^{\text{map}}$ for the number of possible left memory access patterns in junction i and a complete NN, respectively (*i.e.* $G_{\text{net}}^{\text{map}} = \prod_{i=1}^{L} G_i^{\text{map}}$).

When $z_i \geq d_i^{\text{in}}$, which is expected to be true for practical cases of implementing sparse NNs on powerful hardware devices, we state a result without formal proof: G_i^{cp} is lower bounded by G_i^{map}. Thus, for a given NN with $\boldsymbol{N}_{\text{net}}$ and $\boldsymbol{d}_{\text{net}}^{\text{out}}$ and a given hardware device with $\boldsymbol{z}_{\text{net}}$, $G_{\text{net}}^{\text{map}}$ **is a measure of the number of possible predefined sparse connection patterns which are clash-free and hardware friendly**.

For the case of $z_i < d_i^{\text{in}}$ however, it could happen that $G_i^{\text{cp}} < G_i^{\text{map}}$. In particular, a FC junction has only one possible connection pattern, as shown in Fig.

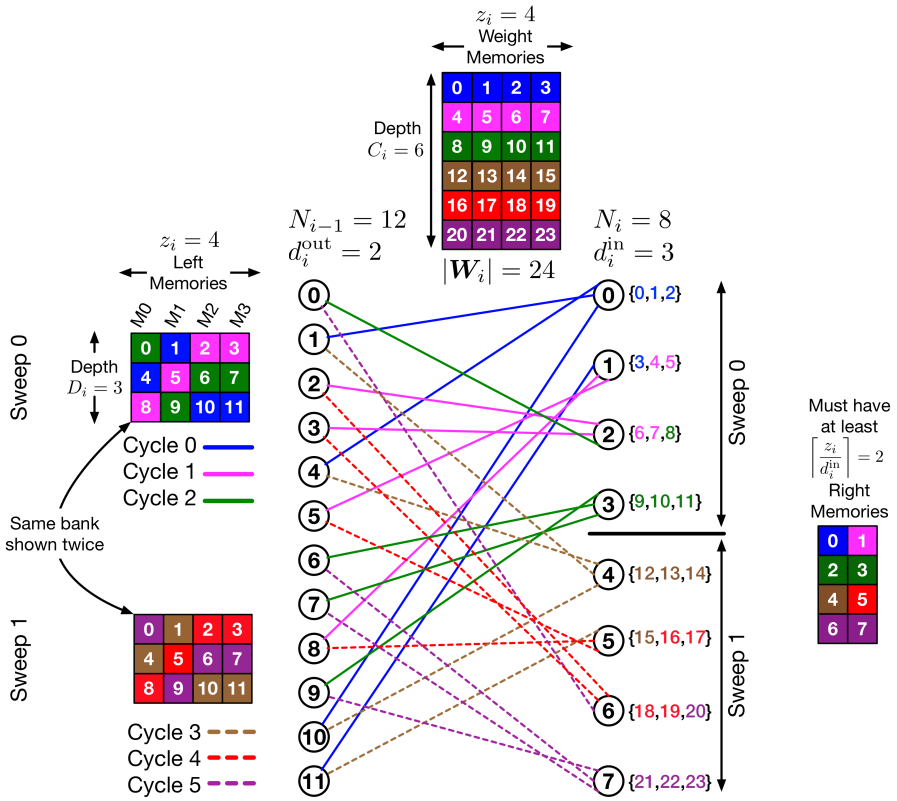


Figure 5.2: Repeating Fig. 4.3 for convenience.

5.1(a), but there can be many possible left memory access patterns. As an example, consider the left memory access pattern for the FC junction in Fig. 4.4. Cycle 0 accesses left nodes $\{0, 1, 2, 3\}$, while cycle 1 accesses left nodes $\{4, 5, 6, 7\}$. These can be switched to get a different clash-free left memory access pattern, but the connection pattern would not change.

5.2.1 Types of memory access patterns

We refer to the seed vector technique discussed previously as **Type 1** memory accesses. This is recapitulated in Fig. 5.3(a). This approach only requires storing

the z_i-dimensional seed vector $\boldsymbol{\phi}_i$, and uses z_i incrementers to generate subsequent addresses. We can compute G_i^{map} as the number of possible ways of designing $\boldsymbol{\phi}_i$, *i.e.* $G_i^{\text{map}} = {D_i}^{z_i}$. This is the simplest and most hardware friendly method for memory accesses we could come up with. However, if desired, there can be other types of memory accessing which increase G_i^{map} (and therefore $G_{\text{net}}^{\text{map}}$ and the number of possible connection patterns) at the cost of more hardware. These are discussed next.

In **Type 2**, a new seed vector $\boldsymbol{\phi}_i$ is defined for every sweep. Considering the example in Fig. 5.3(b), $\boldsymbol{\phi}_i = (1, 0, 2, 2)$ for sweep 0, but $(2, 0, 0, 0)$ for sweep 1. Since there are d_i^{out} sweeps, there will be d_i^{out} different $\boldsymbol{\phi}_i$ vectors for each junction. This results in $G_i^{\text{map}} = {D_i}^{z_i d_i^{\text{out}}}$. This approach requires storing d_i^{out} different $\boldsymbol{\phi}_i$s, and uses z_i incrementers to generate subsequent addresses. Our hardware implementation (described in Section 4.7) used type 2 memory accesses. The seed vectors for all sweeps were pre-generated and hard-coded into FPGA logic.

In **Type 3**, the constraint of cyclically accessing the left memories is also eliminated. Instead, any cycle can access any cell from each of the memories. This means that storing $\boldsymbol{\phi}_i$ is not enough, the entire sequence of memory accesses needs to be stored as a matrix $\boldsymbol{\Phi}_i \in \{0, 1, \cdots, D_i - 1\}^{D_i \times z_i}$. In Fig. 5.3(c) for example, $\boldsymbol{\Phi}_i = ((1, 0, 2, 2), (0, 2, 1, 0), (2, 1, 0, 1))$ for sweep 0. Every sweep would also have a different $\boldsymbol{\Phi}_i$, resulting in $G_i^{\text{map}} = (D_i!)^{z_i d_i^{\text{out}}}$. Thus, this approach requires storing

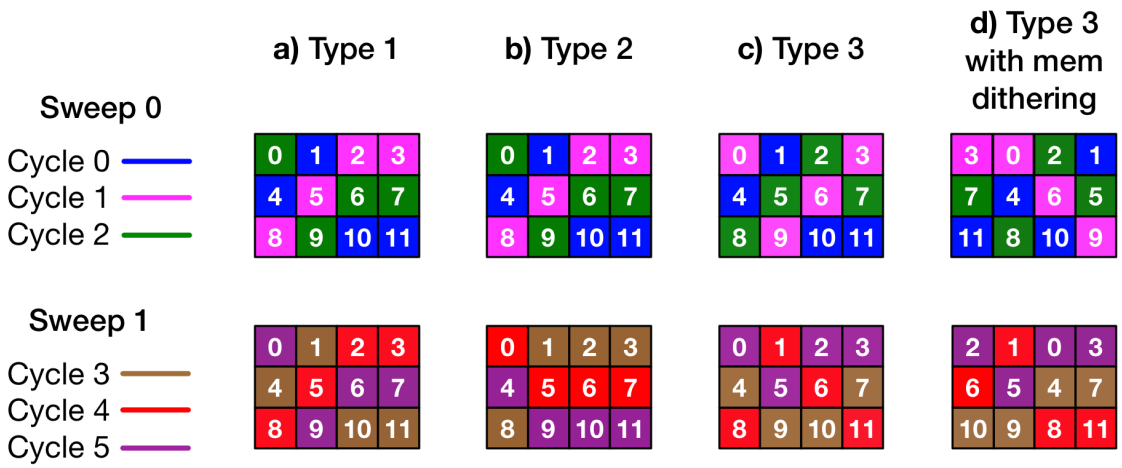


Figure 5.3: (a-c) Various types of clash-free left memory access patterns, and (d) memory dithering for Type 3, using the same left neuronal structure from Fig. 5.2 as an example. The grids represent different access patterns for the same memory bank. The number in each cell represents the left neuron number whose parameter is stored in that cell. Cells sharing the same color are read in the same cycle.

d_i^{out} different $\mathbf{\Phi}_i$s, but removes the need for having z_i incrementers to compute subsequent addresses.

The most general kind of clash-freedom resulting in the biggest G_i^{map} arises from ignoring the effect of sweeps. We call this **Type 4**. This means that each address in each left memory of junction i can be accessed in any cycle, as long as it is accessed a total of d_i^{out} times during the junction cycle (consisting of $C = D_i d_i^{\text{out}}$ cycles). However, this may lead to connectivity patterns where a pair of neurons in adjacent layers have more than one connection between them, which is invalid due to redundancy. Hence we skip the details, but state the resulting count of possible NNs: $G_i^{\text{map}} = \left(\frac{C!}{\left(d_i^{\text{out}}!\right)^{D_i}} \right)^{z_i}$.

A technique that can be applied to all the types of clash-freedom is **memory dithering**, which is a permutation of the z_i memories (*i.e.* the columns) in a

bank. This permutation can change every sweep, as shown in Fig. 5.3(d). Memory dithering incurs no additional incrementers, but incurs an additional storage cost because of the z_i permutation, which goes up to possibly d_i^{out} different z_i permutations for Types 2 and 3. However, G_i^{map} is increased by a factor K_i, as described next.

- If d_i^{in}/z_i is an integer, an integral number of cycles are required to process each right neuron. Since a cycle accesses all memories, dithering has no effect and $K_i = 1$.

- If z_i/d_i^{in} is an integer greater than 1, the effects of dithering on connectivity patterns are only observed when switching from one right neuron to the next within a cycle. This results in

$$K_i = \left(\frac{z_i!}{d_i^{\text{in}}!^{\frac{z_i}{d_i^{\text{in}}}}} \right)^{d_i^{\text{out}}} \tag{5.4}$$

  for Types 2 and 3, and the d_i^{out} exponent is omitted for Type 1 since the access pattern does not change across sweeps.

- When either of z_i or d_i^{in} does not perfectly divide the other, an exact value of K_i is hard to arrive at since some proper or improper fraction of right neurons are processed every cycle. In such cases, K_i is upper-bounded by $(z!)^{d_i^{\text{out}}}$.

Table 5.1: Comparison of Clash-free Left Memory Access Types and associated Hardware Cost for a Single Junction i with $(N_{i-1}, N_i, d_i^{\text{out}}, d_i^{\text{in}}, z_i) = (12, 12, 2, 2, 4)$

Type	Memory Dithering	K_i	G_i^{map}	Cost to Compute Memory Addresses	
				Storage	Incrementers
1	No	N/A	${D_i}^{z_i} = 81$	$z_i = 4$	$z_i = 4$
	Yes	6	$K_i {D_i}^{z_i} = 486$	$2z_i = 8$	$z_i = 4$
2	No	N/A	${D_i}^{z_i d_i^{\text{out}}} = 6561$	$z_i d_i^{\text{out}} = 8$	$z_i = 4$
	Yes	36	$K_i {D_i}^{z_i d_i^{\text{out}}} \approx 236\text{k}$	$2z_i d_i^{\text{out}} = 16$	$z_i = 4$
3	No	N/A	$(D_i!)^{z_i d_i^{\text{out}}} \approx 1.68\text{M}$	$N_{i-1} d_i^{\text{out}} = 24$	None
	Yes	36	$(D_i!)^{z_i d_i^{\text{out}}} \approx 60\text{M}$	$(N_{i-1} + z_i)\, d_i^{\text{out}} = 32$	None

Table 5.1 compares the count of possible left memory access patterns and associated hardware cost for computing memory addresses for Types 1–3, with and without memory dither. The junction used is the same as in Fig. 5.2, except N_i is raised to 12 such that d_i^{in} becomes 2 and allows us to better show the effects of memory dithering. K_i is calculated as given in (5.4).

5.3 Comparison between classes of Pre-defined Sparsity

At this point, the reader might be wondering about the performance of such clash-free patterns. To study this, we group pre-defined sparsity into three classes in order of least to most restrictive, or equivalently, from most to least number of possible connection patterns, or equivalently, from 'difficult to realize' to 'simple to realize' on our hardware architecture.

- Random, wherein connections are randomly distributed without regard for fixed in- and out-degrees. Given N_i and N_{i-1}, a member of this class is completely defined by ρ_i.

- Structured, which imposes the constraints of fixed in- and out-degrees. Given N_i and N_{i-1}, a member of this class is completely defined by d_i^{out}.

- Clash-free, which further imposes the constraint of clash-free memory access patterns. Based on hardware friendliness, we can further expand clash-free into subclasses – from type 3 with memory dithering to type 1 with no memory dithering (*i.e.* the opposite order of Table 5.1). Given N_i and N_{i-1}, a member of this class is completely defined by d_i^{out} and z_i.

Table 5.2 compares performance on different datasets for these three classes. For the clash-free class, *we chose the most restrictive Type 1 with no memory dithering*, and experimented with different $\boldsymbol{z}_{\text{net}}$ settings to simulate different hardware environments:

- Reuters: One junction cycle is 50 cycles for all the different densities. This is because we scale $\boldsymbol{z}_{\text{net}}$ accordingly, corresponding to a more powerful hardware device being used for each NN as ρ_{net} increases.

- CIFAR-100 and MNIST: These simulate cases where hardware choice is limited, such as a high-end, a mid-range and a low-end device being available. Thus three different $\boldsymbol{z}_{\text{net}}$ values are used for CIFAR depending on ρ_{net}.

- TIMIT: We keep $\boldsymbol{z}_{\text{net}}$ constant for different densities. Junction cycle length varies from 90 cycles for $\rho_{\text{net}} = 7.69\%$ to 810 for $\rho_{\text{net}} = 69.23\%$. This shows that when limited to a single low-end hardware device, denser NNs can be processed in longer time by simply changing $\boldsymbol{z}_{\text{net}}$.

Table 5.2 confirms that **the most restrictive and hardware friendly clash-free pre-defined sparse connection patterns do not lead to any statistically significant performance degradation**. Thus, for most practical purposes, one can opt for maximum hardware simplicity and design for Type 1 clash-freedom with no memory dithering. For certain pathological cases of particularly small junctions, one may opt for the other Types and incorporate memory dithering to incorporate more flexibility in getting a particularly desired connection pattern. Finally, we would like to mention that while clash-free connection patterns are focused on ease of hardware implementation, they can also be good for software implementation – *e.g.* the clash-free class could serve as the basis for pattern selection in software packages.

We also observed that random pre-defined sparsity performs poorly for very low density networks, as shown by the blue values. This is possibly because there is non-negligible probability of neurons getting completely disconnected, leading to irrecoverable loss of information.

Table 5.2: Comparison of Pre-Defined Sparse Classes

$\boldsymbol{d}_{\text{net}}^{\text{out}}$	$\rho_{\text{net}}(\%)$	$\boldsymbol{z}_{\text{net}}$	Test Accuracy Performance (%)		
			Clash-free	Structured	Random
MNIST: $\boldsymbol{N}_{\text{net}} = (800, 100, 100, 100, 10)$, FC test accuracy $= (98 \pm 0.1)\%$					
$(80, 80, 80, 10)$	80.2	$(200, 25, 25, 4)$	97.9 ± 0.2	97.9 ± 0.2	97.8 ± 0.2
$(60, 60, 60, 10)$	60.4	$(200, 25, 25, 4)$	97.6 ± 0.1	97.8 ± 0.1	97.6 ± 0.2
$(40, 40, 40, 10)$	40.6	$(200, 25, 25, 5)$	97.5 ± 0.1	97.7	97.6 ± 0.1
$(20, 20, 20, 10)$	20.8	$(200, 25, 25, 10)$	97.2 ± 0.2	97.2 ± 0.1	97.1 ± 0.1
$(10, 10, 10, 10)$	10.9	$(200, 25, 25, 25)$	96.7 ± 0.1	96.8 ± 0.2	96.7 ± 0.2
$(5, 10, 10, 10)$	6.9	$(100, 25, 25, 25)$	96.3 ± 0.1	96.3 ± 0.1	96.2 ± 0.1
$(2, 5, 5, 10)$	3.6	$(80, 25, 25, 50)$	95 ± 0.2	95.1 ± 0.1	95 ± 0.3
$(1, 2, 2, 10)$	2.2	$(80, 20, 20, 100)$	93.3 ± 0.3	93.1 ± 0.5	92 ± 0.3
Reuters: $\boldsymbol{N}_{\text{net}} = (2000, 50, 50)$, FC test accuracy $= (89.6 \pm 0.1)\%$					
$(25, 25)$	50	$(1000, 25)$	89.4 ± 0.1	89.3	89.4
$(10, 10)$	20	$(400, 10)$	87 ± 0.1	86.7 ± 0.1	86.5 ± 0.1
$(5, 5)$	10	$(200, 5)$	78.5 ± 0.5	78.2 ± 0.7	77.5 ± 0.6
$(2, 2)$	4	$(80, 2)$	53.3 ± 1.8	51.2 ± 1.7	46.8 ± 2.9
$(1, 1)$	2	$(40, 1)$	28.4 ± 2.4	28.7 ± 2.3	28 ± 1.9
TIMIT: $\boldsymbol{N}_{\text{net}} = (39, 390, 39)$, FC test accuracy $= (43.2 \pm 0.2)\%$					
$(270, 27)$	69.2	$(13, 13)$	43 ± 0.1	43	43 ± 0.1
$(180, 18)$	46.2		42.7 ± 0.1	42.8 ± 0.1	42.9 ± 0.1
$(90, 9)$	23.1		42.1 ± 0.1	42.5 ± 0.1	42.4 ± 0.1
$(60, 6)$	15.4		41.5 ± 0.1	41.8 ± 0.2	41.9 ± 0.1
$(30, 3)$	7.7		40.5 ± 0.2	40.1 ± 0.2	39.4 ± 0.8
CIFAR-100 [a]: $\boldsymbol{N}_{\text{net}} = (4000, 500, 100)$, FC top-5 test accuracy $= (87.1 \pm 0.6)\%$					
$(100, 100)$	22	$(2000, 250)$	87.5 ± 0.2	87.7 ± 0.2	87.4 ± 0.3
$(29, 29)$	6.4		86.8 ± 0.3	87.2 ± 0.5	87.1 ± 0.2
$(12, 12)$	2.6	$(400, 50)$	86.3 ± 0.2	86.5 ± 0.4	86.6 ± 0.4
$(5, 5)$	1.1		85.3 ± 0.5	85.5 ± 0.5	85.7 ± 0.3
$(2, 2)$	0.4	$(80, 10)$	84.1 ± 0.5	84.3 ± 0.3	83.8 ± 0.3
$(1, 1)$	0.2		83 ± 0.5	83.3 ± 0.4	81.7 ± 0.7

[a]For CIFAR-100, given values of $\boldsymbol{N}_{\text{net}}$, $\boldsymbol{d}_{\text{net}}^{\text{out}}$, $\boldsymbol{z}_{\text{net}}$ and ρ_{net} are just for the MLP portion, which follows convolution and pooling layers to form the complete NN, as described in Section 3.3.1. Reported values are top-5 test accuracies obtained from training on the complete NN.

5.4 Comparison to other methods of sparsity

As we have seen, pre-defined sparsity is a method of complexity reduction 'from the get-go', *i.e.* prior to training. It can be thought of as a preemptive method since it is foresees that most NNs are over-parametrized, hence, most of the weights can be absent from the start without appreciable performance loss. In this sense, **pre-defined sparsity should not be compared on an apples-to-apples basis with methods which train the complete FC NN and then reduce complexity based on training results**. Nevertheless, for the purposes of thoroughness, our previous work [25] compared our method of pre-defined sparsity with the most restrictive Type 1 clash freedom against two other methods of sparsifying NNs.

This comparison and the consequent results were achieved by Kuan-Wen Huang, hence are not presented in this dissertation. We state the primary conclusion – even though clash-free patterns are highly structured and pre-defined, there is no significant performance degradation when compared to advanced methods for producing sparse models which exploit specific properties of the dataset or learn sparse patterns during training. In fact, the performance of our method is comparable to FC NNs at most values of ρ_{net}. The reader is encouraged to refer to Section V of [25] for details.

5.5 Metrics for Connection Patterns

This section discusses possible methods and metrics to judge the 'goodness' of a connection pattern before training a NN and computing inference accuracies. These methods can be helpful in filtering out bad connection patterns without incurring the time and computational expense of training. We will use the concept of biadjacency matrices introduced earlier in Section 5.1.

Biadjacency matrices for individual junctions can be multiplied to yield the effective connection pattern for an **equivalent junction** spanning any two junctions x and y, *i.e.* $\underset{N_y \times N_{x-1}}{\boldsymbol{\mathcal{B}}_{x:y}} = \prod_{i=y}^{x} \boldsymbol{\mathcal{B}}_i$, where element $\boldsymbol{\mathcal{B}}_{x:y}^{(j,k)}$ denotes the number of paths from the kth neuron in layer $x-1$ to the jth neuron in layer y. Thus, these elements are integers ≥ 0. For the special case where $x = 1$ and $y = L$, we obtain the input-output biadjacency matrix $\boldsymbol{\mathcal{B}}_{1:L}$, *i.e.* the equivalent junction is the whole NN. Each element of $\boldsymbol{\mathcal{B}}_{1:L}$ is the number of paths from a certain input neuron to a certain output neuron.

As an example, consider the NN with $\boldsymbol{N}_{\text{net}} = (6, 4, 2)$ shown in Fig. 5.4(a). The input-output biadjacency matrix is $\boldsymbol{\mathcal{B}}_{1:2} = \boldsymbol{\mathcal{B}}_2 \boldsymbol{\mathcal{B}}_1$, the equivalent out-degree is $d_{1:2}^{\text{out}} = d_1^{\text{out}} d_2^{\text{out}} = 2$, and equivalent in-degree is $d_{1:2}^{\text{in}} = d_1^{\text{in}} d_2^{\text{in}} = 6$. The equivalent input-output junction $1:2$ is shown in Fig. 5.4(b). Note how the equivalent biadjacency matrix has double connections between some pairs of neurons. This indicates that there are multiple paths connecting certain inputs to certain outputs.

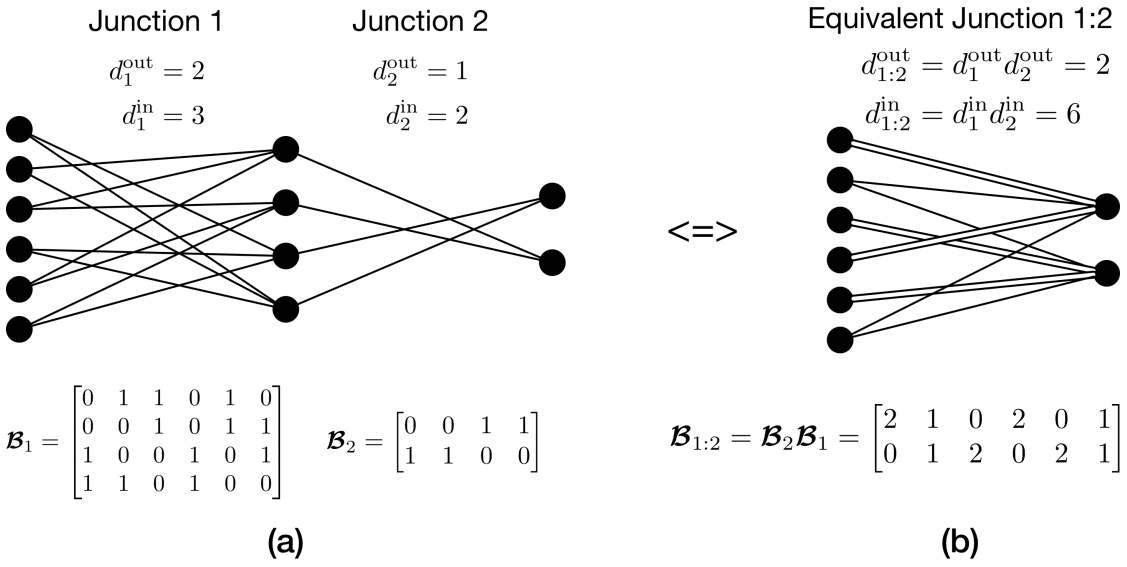


Figure 5.4: (a) Biadjacency matrices for a NN with $\boldsymbol{N}_{\text{net}} = (6, 4, 2)$, $\boldsymbol{d}_{\text{net}}^{\text{out}} = (2, 1)$, $\boldsymbol{d}_{\text{net}}^{\text{in}} = (3, 2)$. (b) Equivalent input-output junction and its biadjacency matrix. Double lines indicate double connections between the same pair of neurons.

Note that an equivalent junction is only an abstract concept that aids visualizing how neurons connect across junctions. The reader should not mistake it for an actual junction; such an understanding would be incorrect because it would ignore the effects of the non-linearity in the hidden layer. However, constructing equivalent junctions is helpful in formalizing some metrics for connection patterns, as will be discussed in the remainder of this section.

5.5.1 Window biadjacency matrices

We now attempt to characterize the quality of a pre-defined sparse connection pattern, *i.e.* we try to find the best possible way to connect neurons to optimize

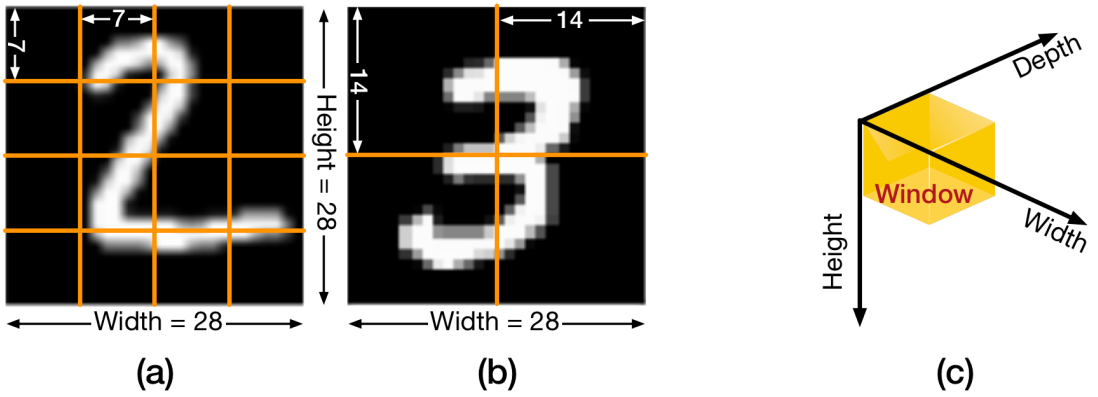


Figure 5.5: (a–b) Images can be divided into 2D windows corresponding to width and height at the output of the input layer of a NN. For example, a 28×28 MNIST image can be divided into (a) 16 windows of size 7×7 each, or (b) 4 windows of size 14×14 each. (c) Example of a 3D window where there is a 3rd dimension for depth, such as color images like CIFAR, or the output of a convolutional layer.

performance. As indicated in Chapter 3, one reason behind the success of pre-defined sparsity is that there exists redundancy / correlated information in typical NN problems. Intuitively, we assume that left neurons of a junction can be grouped into **windows** depending on the **dimensionality** of the left layer output. For example, the input layer for a NN training on MNIST would have 2D windows, each of which might correspond to a fraction of the image, as shown in Fig. 5.5(a–b). When outputs from a NN have an additional depth dimension, such as color channels in CIFAR, each window is a cuboid capturing fractions of width, height and depth, as shown in Fig. 5.5(c).

The idea behind constructing windows is that *neurons in a layer should get information from all portions of the entity its adjacent layer is attempting to represent.* Thus, we will try to maximize the number of left windows to which each

right neuron connects. To realize the importance of this, consider the MNIST output neuron representing digit 2. Let's say the sparse connection pattern is such that when the connections to output 3 are traced back to the input layer, they all come from the top half of the image. This would be undesirable since the top half of an image of a 2 can be mistaken for a 3, as in Fig. 5.5(a–b). A good pre-defined sparse connection pattern will try to avoid such scenarios by spreading the connections to any right neuron across as many input windows as possible. The problem can also be mirrored so that every left neuron connects to as many different right windows as possible. This ensures that local information from left neurons is spread to different parts of the right layer. The grouping of right windows will depend on the dimensionality of the input to the right layer.

The window size is chosen to be the minimum possible such that the ideal number of connections from or to it remains integral. In order to achieve the minimum window size, we let the number of left windows be d_i^{in} and the number of right windows be d_i^{out}. So the number of neurons in each left and right window is N_{i-1}/d_i^{in} and N_i/d_i^{out}, respectively. The example from Fig. 5.4 is reproduced in Fig. 5.6. For example, since $d_1^{\text{in}} = 3$, the inputs must be grouped into 3 windows of size 2 so that ideally one connection from each reaches every hidden neuron. The 1st and 3rd hidden neurons achieve this and are colored green, while the other two do not and are colored red.

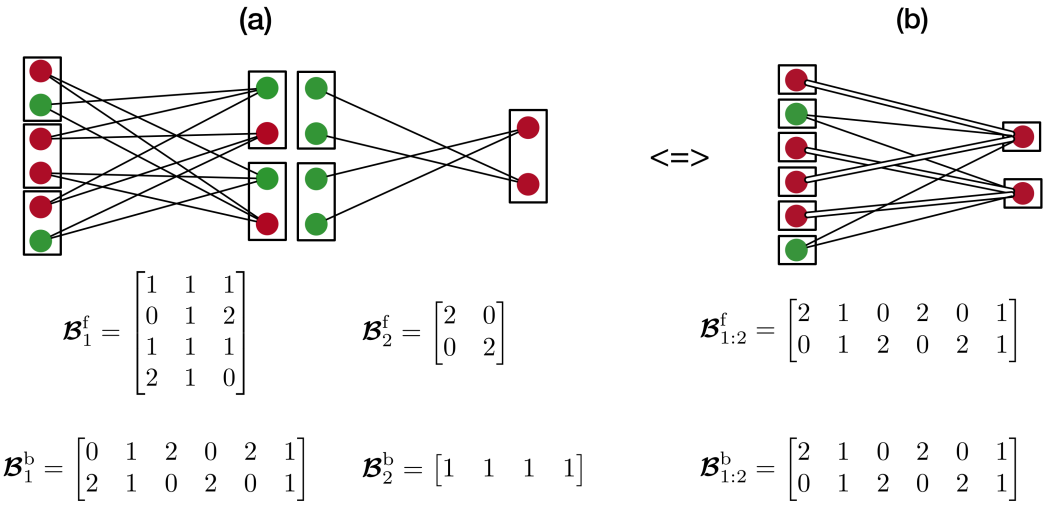
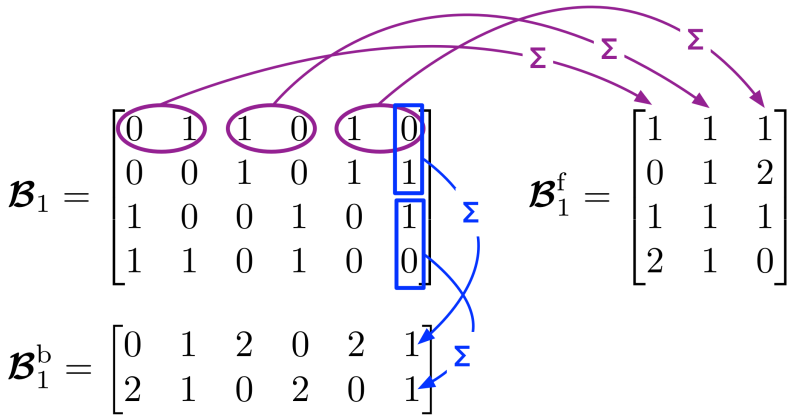


Figure 5.6: Window biadjacency matrices and scatter (to be discussed in Section 5.5.2) for the NN in Fig. 5.4. Green neurons indicate ideal connectivity. The hidden layer is split into two parts to show separate constructions of $\mathcal{B}_1^{\mathrm{f}}$ and $\mathcal{B}_2^{\mathrm{b}}$. (a) shows the complete NN, while (b) shows the equivalent input-output junction $1:2$. The final scatter value S is bolded within the scatter vector $\boldsymbol{S}^{\mathrm{net}}$.

Then we construct the forward window biadjacency matrix $\underset{N_i \times d_i^{\mathrm{in}}}{\mathcal{B}_i^{\mathrm{f}}}$ and backward window biadjacency matrix $\underset{d_i^{\mathrm{out}} \times Ni-1}{\mathcal{B}_i^{\mathrm{b}}}$ by summing up entries of $\mathcal{B}_i$ as shown in Fig. 5.7. These window biadjacency matrices describe connectivity between windows in a layer and neurons on the opposite side of the junction. Forward indicates windows on the left to neurons on the right, and vice-versa for backward. Ideally, *every window biadjacency matrix for a single junction should be the all 1s matrix, which signifies exactly 1 connection from every window to every neuron on the opposite side.*



$$\boldsymbol{\mathcal{B}}_1 = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \end{bmatrix} \quad \boldsymbol{\mathcal{B}}_1^{\mathrm{f}} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 2 \\ 1 & 1 & 1 \\ 2 & 1 & 0 \end{bmatrix}$$

$$\boldsymbol{\mathcal{B}}_1^{\mathrm{b}} = \begin{bmatrix} 0 & 1 & 2 & 0 & 2 & 1 \\ 2 & 1 & 0 & 2 & 0 & 1 \end{bmatrix}$$

Figure 5.7: Constructing window biadjacency matrices for junction 1 of the NN in Fig. 5.4(a). The forward window biadjacency matrix $\boldsymbol{\mathcal{B}}_1^{\mathrm{f}}$ is constructed by summing (Σ) across $d_1^{\mathrm{in}} = 3$ windows of each row of $\boldsymbol{\mathcal{B}}_1$, each window is of size $N_0/d_1^{\mathrm{in}} = 2$. The backward window biadjacency matrix $\boldsymbol{\mathcal{B}}_1^{\mathrm{b}}$ is constructed by summing (Σ) down $d_1^{\mathrm{out}} = 2$ windows of each column of $\boldsymbol{\mathcal{B}}_1$, each window is of size $N_1/d_1^{\mathrm{out}} = 2$.

Note that these matrices can also be constructed for equivalent junctions, *i.e.* $\boldsymbol{\mathcal{B}}_{x:y}^{\mathrm{f}}$ and $\boldsymbol{\mathcal{B}}_{x:y}^{\mathrm{b}}$, by multiplying matrices for individual junctions. It could happen that the effective $d_{x:y}^{\mathrm{in}}$ for layer y becomes more than N_{x-1}, or the effective $d_{x:y}^{\mathrm{out}}$ for layer $x-1$ becomes more than N_y. In such cases, the number of neurons in each window is rounded up to 1. This scenario signifies that there are multiple paths connecting neurons in the equivalent junction, *i.e.* $\boldsymbol{\mathcal{B}}_{x:y}^{\mathrm{f}}$ and/or $\boldsymbol{\mathcal{B}}_{x:y}^{\mathrm{b}}$ have entries greater than 1. In such cases, a good connection pattern should try to distribute the connections as evenly as possible. as quantified in Section 5.5.2.

5.5.2 Scatter

We propose **scatter** S as a proxy for the goodness of a pre-defined sparse MLP connection pattern. In other words, it can be useful in predicting NN performance

before training. To compute scatter, we count the number of entries greater than or equal to 1 in the window biadjacency matrix. If a particular window gets more than its fair share of connections to a neuron on the opposite side, then it is depriving some other window from getting its fair share. This should not be encouraged, so we treat entries greater than 1 the same as 1. Scatter is the average of the count, *i.e.* for junction i:

$$S_i^{\mathrm{f}} = \frac{1}{N_i d_i^{\mathrm{in}}} \sum_{j=1}^{N_i} \sum_{k=1}^{d_i^{\mathrm{in}}} \mathbb{I}\left(\boldsymbol{\mathcal{B}}_i^{\mathrm{f}(j,k)} \geq 1\right) \tag{5.5a}$$

$$S_i^{\mathrm{b}} = \frac{1}{d_i^{\mathrm{out}} N_{i-1}} \sum_{j=1}^{d_i^{\mathrm{out}}} \sum_{k=1}^{N_{i-1}} \mathbb{I}\left(\boldsymbol{\mathcal{B}}_i^{\mathrm{b}(j,k)} \geq 1\right) \tag{5.5b}$$

The subscript and superscript notation for scatter is the same as the biadjacency matrices, *i.e.* f and b denote forward (left windows to right neurons) and backward (right neurons to left windows), respectively. As an example, the NN in Fig. 5.6(a) has $S_1^{\mathrm{f}} = 10/12 = 0.83$ since 10 out of the 12 elements in $\boldsymbol{\mathcal{B}}_1^{\mathrm{f}}$ are ≥ 1. The other scatter values can be computed similarly to form the scatter vector $\boldsymbol{S}^{\mathrm{net}} = \left(S_1^{\mathrm{f}}, S_1^{\mathrm{b}}, S_2^{\mathrm{f}}, S_2^{\mathrm{b}}, S_{1:2}^{\mathrm{f}}, S_{1:2}^{\mathrm{b}}\right)$. Notice that $\boldsymbol{S}^{\mathrm{net}}$ will be all 1s for FC NNs, which is the ideal case. Incorporating sparsity leads to reduced $\boldsymbol{S}^{\mathrm{net}}$ values. Finally, we define:

➤ **Definition 7: *Scatter*:** Scatter is a measure of how well-distributed the connections from neurons in any layer of a NN are to its adjacent layers. It is measured as the minimum element in the scatter vector, *i.e.* :

$$S = \min \boldsymbol{S}^{\text{net}} \tag{5.6}$$

In Fig. 5.6 for example, $\boldsymbol{S}^{\text{net}} = (0.83, 0.67, 0.5, 1, 0.67, 0.67)$, so $S = 0.5$ (shown in bold). Our experiments indicate that any low value in $\boldsymbol{S}^{\text{net}}$ leads to bad performance, so we picked the critical minimum value. These experiments are described next.

Experimental Results on Scatter

We ran experiments to evaluate scatter using structured pre-defined sparse (non-clash-free) NNs training on three datasets: a) Morse code symbol recognition, b) MNIST, and c) CIFAR-10 images. (a) will be described in Chapter 6, while (c) has the same image format as CIFAR-100, but 10 classes instead of 100 (see Section 3.3.1 for more details on MNIST and CIFAR). The reader is encouraged to refer to our previous work [26] for complete details on the network and training configuration. We skip those details here and focus on the results instead. These are shown in Fig. 5.8, which plots classification performance on validation data vs. scatter S. Subfigures (a–b) show that performance gets better with increasing scatter. Subfigure (c) does not show the trend clearly, we hypothesize that this is

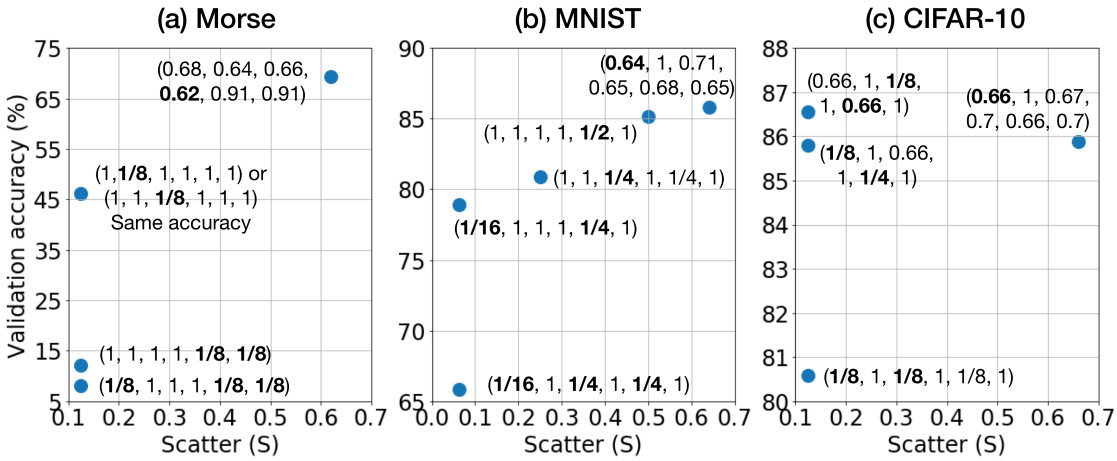


Figure 5.8: Performance vs. scatter for structured pre-defined sparse (non-clash-free) NNs training on (a) Morse (b) MNIST, and (c) CIFAR-10, part of which is convolutional. All minimum values that need to be considered to differentiate between connection patterns are bolded.

due to using convolution layers in the NN for CIFAR-10, leading to reduced impact of the MLP portion.

$\boldsymbol{S}^{\text{net}}$ is shown alongside each point. When S is equal for different connection patterns, the next minimum value in $\boldsymbol{S}^{\text{net}}$ needs to be considered to differentiate the networks, and so on. Considering the Morse results, the leftmost three points all have $S = 1/8$, but the number of occurrences of $1/8$ in $\boldsymbol{S}^{\text{net}}$ is 3 for the lowest point (8% accuracy), 2 for the second lowest (12% accuracy) and 1 for the highest point (46% accuracy). For the MNIST results, both the leftmost points have a single minimum value of $1/16$ in $\boldsymbol{S}^{\text{net}}$, but the lower has two occurrences of $1/4$ while the upper has one.

We draw several insights from these results. Firstly, although we defined S as a single value for convenience, there may arise cases when other (non-minimum)

elements in $\boldsymbol{S}^{\text{net}}$ are important. Secondly, our results indicate that *scatter is a sufficient metric for performance, not necessary.* In other words, a network with a high S value should perform well, but a network with a slightly lower S value than another cannot be conclusively dismissed as being worse. But if a network has multiple low values in $\boldsymbol{S}^{\text{net}}$, it should be rejected. Finally, carefully choosing which neurons to group in a window will increase the predictive power of scatter. A priori knowledge of the dataset will lead to better window choices.

The rightmost point in each subfigure of Fig. 5.8 corresponds to structured pre-defined sparsity as we have described so far, *i.e.* the only constraints are fixed in- and out-degree. To be more specific, the connections can arrange themselves in any way they want as long as these constraints are obeyed. On the other hand, all other points were generated by specifically planning individual connections, *i.e.* designing the complete biadjacency matrices manually. These perform poorly because when we designed a particular junction to have high values in $\boldsymbol{S}^{\text{net}}$, it invariably led to low values for another junction, leading to a low value for the final S. This explains why the unplanned patterns perform the best.

Note that our experiments on scatter are not as exhaustive as those performed to discover trends and guidelines in pre-defined sparsity – as detailed in Section 3.3 – which also help in indicating how well a pre-defined sparse NN can be expected to perform. Nevertheless, our research group has plans to develop scatter and

other NN performance-predicting metrics further with a view towards applications in automated machine learning – a topic described in Chapter 7.

5.6 Summary

This chapter explored different methods of connectivity in sparse NNs. In particular, we demonstrated that imposing hardware-friendly restrictions on sparsity does not lead to performance loss, and developed metrics to characterize the 'goodness' of a sparse NN. This concludes the primary discussions on sparsity. The remainder of this dissertation will discuss our other contributions.

Chapter 6

Dataset Engineering

Thus far, this document has discussed methods and architectures for complexity reduction of neural networks primarily aimed towards classification problems. An underlying assumption was that there is a large amount of high quality labeled data available to train and benchmark the performance of different NNs. The effects of dataset size on network performance has been explored in [104], in particular, more data is beneficial in reducing overfitting and improving robustness and generalization capabilities of NNs [105, 106]. However, it is often a challenge to obtain adequate amounts of quality labeled data, and several entities [107–109] have identified this as a bottleneck to pushing the frontiers of NN performance.

A possible solution is to obtain data by synthetic instead of natural means. **Synthetic data** is generated using computer algorithms instead of being collected from real-world scenarios. The advantages are that a) computer algorithms can be tuned to mimic real-world settings to desired levels of accuracy, and b) a theoretically unlimited amount of data can be generated by running the algorithm long enough. Synthetic data has been successfully used in problems such as 3D

imaging [110], point tracking [111], breaking Captchas on popular websites [112], and augmenting real world datasets [113].

We conducted some investigations on synthetic data with the underlying motive of creating **several low-redundancy datasets**, *i.e.* which would challenge a pre-defined sparse NN more than datasets with considerable redundancy such as discussed in Section 3.3.2. We came up with a family of synthetic datasets on **classifying Morse codewords**. Morse code is a system of communication where each letter, number or symbol in a language is represented using a sequence of dots and dashes, separated by spaces. It is widely used to communicate in situations where voice is not possible, such as helping people with disabilities talk [114–116], or where message transmission needs to be achieved using only 2 states [117], or in rehabilitation and education [118]. Morse code is a useful skill to learn and there exist cellphone apps designed to train people in its usage [119, 120].

Our classification problem groups Morse codewords into 64 character classes comprising letters, numbers and symbols. This is different from previous works with just two classes for dots and dashes [114, 115, 117], and other approaches using fuzzy logic [117], time series to decode English words in Morse code [121, 122], or radial basis functions [123]. We also investigated **metrics to characterize the difficulty of a dataset**. In summary, we refer to our efforts discussed in this chapter as *dataset engineering*. We published these efforts in [32], which won a

'Conference Best Paper' award. The codebase is **available on Github [33]**. The key contributions are as follows:

Contributions in dataset engineering

1. To the best of our knowledge, we are the first to develop an algorithm for generating machine learning datasets of varying difficulty – measured as test set classification accuracy of a NN training on it. Harder datasets lead to lower accuracy, and vice-versa. Encountering harder datasets leads to aggressive exploration of hyperparameters and learning algorithms, which ultimately increases the robustness of NNs training on them. Some particularly hard datasets are useful for testing the limits of our sparse NNs.

2. We introduce metrics to quantify the difficulty of a dataset before having a NN train on them. While some of these arise from information theory, we also come up with a new metric which achieves a high correlation coefficient with the eventual accuracy obtained after training and inference.

3. This work is one of few to introduce a spatially 1-dimensional dataset. This is in contrast to the wide array of image and character recognition datasets which are usually 2D such as MNIST, where each image has width and height, or 3D such as CIFAR, where each image has width,

height and depth (color channels). The dimensionality of the inputs is important when benchmarking pre-defined sparse connection patterns, since some of the metrics discussed in Chapter 5 depend on dimension.

6.1 Generating Algorithm

We picked **64 class labels** for our dataset – the 26 English letters, the 10 Arabic numerals, and 28 other symbols such as (, +, :, etc. Each of these is represented by a sequence of dots and dashes in Morse code, for example, + is represented as • — • — •. So as to mimic a real-world scenario in our algorithm, we imagined a human or a Morse code machine writing out this sequence within a frame of fixed size. Wherever the pen or electronic instrument touches is darkened and has a high intensity, indicating the presence of dots and dashes, while the other parts are left blank, *i.e.* spaces.

Step 1 – Frame Partitioning

For our algorithm, each Morse codeword lies in a frame which is a vector of 64 values, *i.e.* **64 features** per input. Within the frame, the length of a sequence having consecutive similar values is used to differentiate between a dot and a dash. In the baseline dataset, a dot can be 1-3 values wide and a dash 4-9. This is in accordance with international Morse code regulations [124] where the size or

duration of a dash is around three times that of a dot. The space between a dot and a dash can have a length of 1-3 values. The exact length of a dot, dash or space is chosen from these ranges according to a uniform probability distribution. This is to mimic the human writer who is not expected to make each symbol have a consistent length, but can be expected to make dots and spaces around the same size, and dashes longer than them. The baseline dataset has no leading spaces before the 1st dot or dash, *i.e.* the codeword starts from the left edge of the frame. There are trailing spaces to fill up the right side of the frame after all the dots and dashes are complete.

Step 2 – Assigning Values for Intensity Levels

All values in the frame are initially real numbers in the range $[0, 16]$ and indicate the intensity of that point in the frame. For dots and dashes, the values are drawn from a normal distribution with mean $\mu = 12$ and standard deviation $\sigma = 4/3$. The idea is to have the 'six-sigma' range from $(12 - 3 \times 4/3) = 8$ to $(12 + 3 \times 4/3) = 16$. This ensures that any value making up a dot or a dash will lie in the upper half of possible values, *i.e.* in the range $[8, 16]$. The value of a space is exactly 0. Once again, these conditions mimic the human or machine writer who is not expected to have consistent intensity for every dot and dash, but can be expected to not let the writing instrument touch portions of the frame which are spaces.

Step 3 – Noising

Noise in input samples is often deliberately injected as a means of avoiding overfitting in NNs [105], and has been shown to be superior to other methods of avoiding overfitting [125]. This was, however, the secondary reason behind our experimenting with noise. The primary reason was to deliberately make the data hard to classify and test the limits of different NNs processing it. Noise can be thought of as a human accidentally varying the intensity of writing the Morse codeword, or a Morse communication channel having noise. The baseline dataset has no noise, while others have additive noise from a mean-zero normal distribution applied to them. Fig. 6.1 shows the 3 steps up to this point. Finally, all the values are normalized to lie within the range $[0, 1]$ with precision of 3 decimal places.

Step 4 – Mass Generation

Steps 1-3 describe the generation of one input sample corresponding to some particular class label. This **can be repeated as many times as required for each of the 64 class labels**. This demonstrates a key advantage of synthetic over real-world data – the ability to generate an arbitrary amount of data having an arbitrary prior probability distribution over its classes. The baseline dataset has 7000 examples for each class, for a total of 448,000 examples.

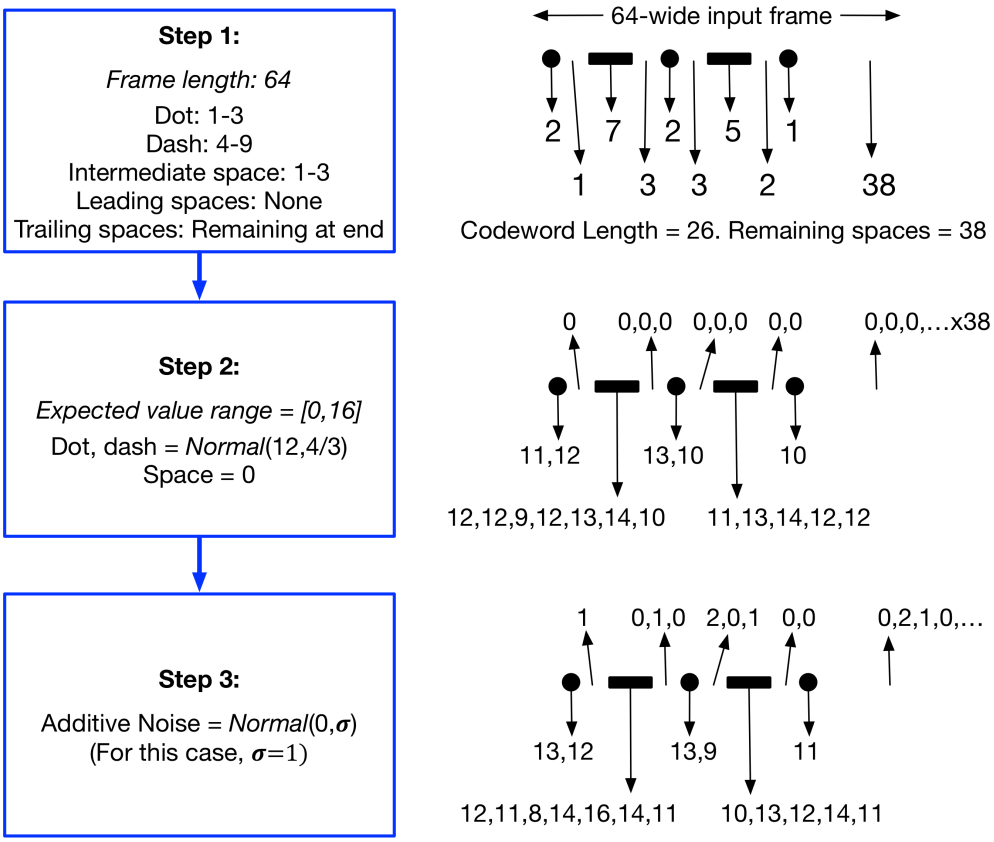


Figure 6.1: Generating the Morse codeword • — • — • corresponding to the + symbol. The first 3 steps, prior to normalizing, are shown. Only integer values are shown for convenience, however, the values can and generally will be fractional. $Normal(\mu, \sigma)$ denotes a normal distribution with mean = μ, standard deviation = σ. For this figure, $\sigma = 1$.

6.1.1 Variations and Difficulty Scaling

The baseline dataset is as described so far, except that $\sigma = 0$, *i.e.* it has no additive noise. We experimented with the following variations in datasets:

1. Baseline with **additive noise** = $Normal(0, \sigma)$, $\sigma \in \{0, 1, 2, 3, 4\}$. These are called *Morse 1.σ*, *i.e. 1.0* to *1.4*, where *1.0* is the baseline.

2. Instead of having the codeword start from the left edge of the frame, we introduced a random number of **leading spaces**. For example, in Fig. 6.1, the codeword occupies a length of 26 values. The remaining 38 space values can be randomly divided between leading and trailing spaces. This increases the difficulty of the dataset since no particular set of neurons are expected to be learning dots and dashes as the actual codeword could be anywhere in the frame. Just like variation 1, we added noise and call these datasets *Morse 2.*σ, $\sigma \in \{0, 1, 2, 3, 4\}$.

3. There is no overlap between the lengths of dots and dashes in the datasets described so far. The difficulty can be increased by making **dash length = 3-9 values**, which is exactly according to the convention of having dash length thrice of dot length. This means that dashes can masquerade as dots and spaces, and vice-versa. This is done on top of introducing leading spaces. These datasets are called *Morse 3.*σ, σ being as before.

4. The Morse datasets only have 64 input features, which is quite small compared to some others such as MNIST (784 features) or CIFAR (3072 features). To decrease dataset difficulty, we introduced **dilation** by a factor of 4. This is done by scaling all lengths in variation 3 by a factor of 4, *i.e.* frame length (number of features) becomes 256, dot sizes and space sizes are 4-12, and dash size is 12-36. These datasets are called *Morse 4.*σ, σ being as before.

5. Increasing the number of training examples, *i.e.* the **size of the dataset**, makes it easier to classify since a NN has more labeled training examples to learn from. Accordingly we chose *Morse 3.1* and scaled the number of examples to obtain *Morse Size* x, $x \in \{1/8, 1/4, 1/2, 2, 4, 8\}$. For example, *Morse Size* $1/2$ has 3500 examples for each class, for a total of 224,000 examples.

6.2 Neural Network Results and Analysis

6.2.1 Results

We initially designed a FC MLP NN to benchmark the different variations of the Morse datasets. N_0 always matches the frame length, *i.e.* 256 for the *Morse 4.*σ datasets and 64 for all others, while $N_L = 64$ to match the number of classes. We used a single hidden layer with 1024 neurons, *i.e.* $\boldsymbol{N}_{\text{net}} = (256, 1024, 64)$ for *Morse 4.*σ and $\boldsymbol{N}_{\text{net}} = (64, 1024, 64)$ for all other datasets. The performance, *i.e.* accuracy, generally increases on adding more hidden neurons, however, we stuck with 1024 since values above that yielded diminishing returns. The hidden layer has ReLU activations, while the output is a softmax probability distribution. We used the Adam optimizer with default parameters, He normal initialization for the weights, and trained for 30 epochs using a minibatch size $M = 128$. We used $5/7$th of the total examples for training, and $1/7$th each for validation and testing. All reported accuracies are those obtained on the test samples unless otherwise mentioned.

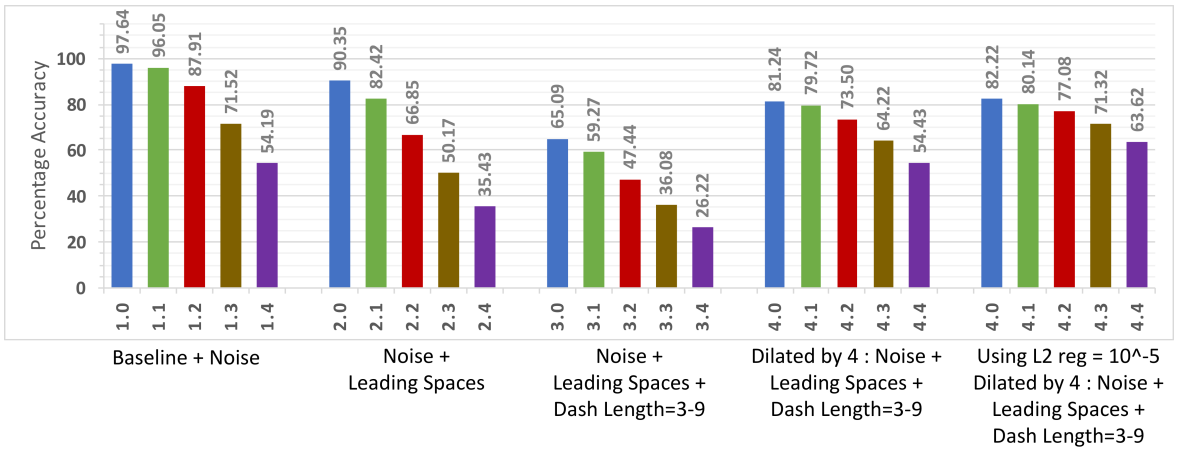


Figure 6.2: Percentage classification accuracies obtained by the FC NN (described in Section 6.2.1) on the test set of different Morse datasets. The rightmost set of bars corresponds to *Morse 4.*σ with L2 regularization with $\lambda = 10^{-5}$.

Validation results suggested that no regularization was required when training on *Morse 1.*σ, *2.*σ and *3.*σ. However, the NN for *Morse 4.*σ is more prone to overfitting due to having more input neurons, leading to more weight parameters. Accordingly we applied L2 regularization with $\lambda = 10^{-5}$, which was the best value as determined from validation.

Test accuracy results after training the NN on the different Morse datasets are shown in Fig. 6.2. As expected, increasing the standard deviation of noise results in drop in performance. This effect is not felt strongly when $\sigma = 1$ since the 3σ range can take spaces to a value of 3 (on a scale of $[0, 16]$, *i.e.* before normalizing to $[0, 1]$), while dots and dashes can drop to $8-3 = 5$, so the probability of a space being confused with a dot or dash is basically 0. Confusion can occur for $\sigma \geq 2$, and gets worse for higher values, as shown in Fig. 6.3.

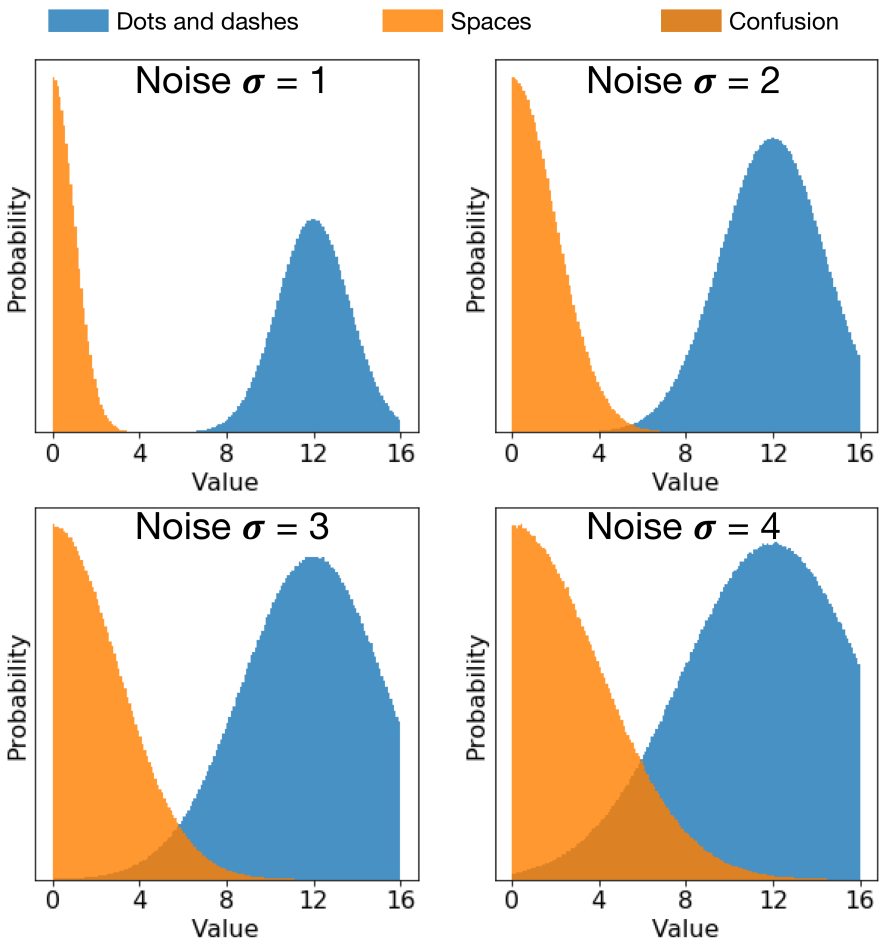


Figure 6.3: Effects of noise leading to spaces (orange) getting confused (brown) with dots and dashes (blue). Higher values of noise σ lead to increased probability of the brown region, making it harder for the NN to discern between 'dots and dashes' and spaces. The x-axis in each plot shows values in the range $[0, 16]$, *i.e.* before normalizing to $[0, 1]$.

Since the codeword lengths do not often stretch beyond 32, the first half of neurons usually encounter high input intensity values corresponding to dots and dashes during training. This means that the latter half of neurons mostly encounter lower input values corresponding to spaces. This aspect changes when introducing leading spaces, which become inputs to some neurons in the first half. The result

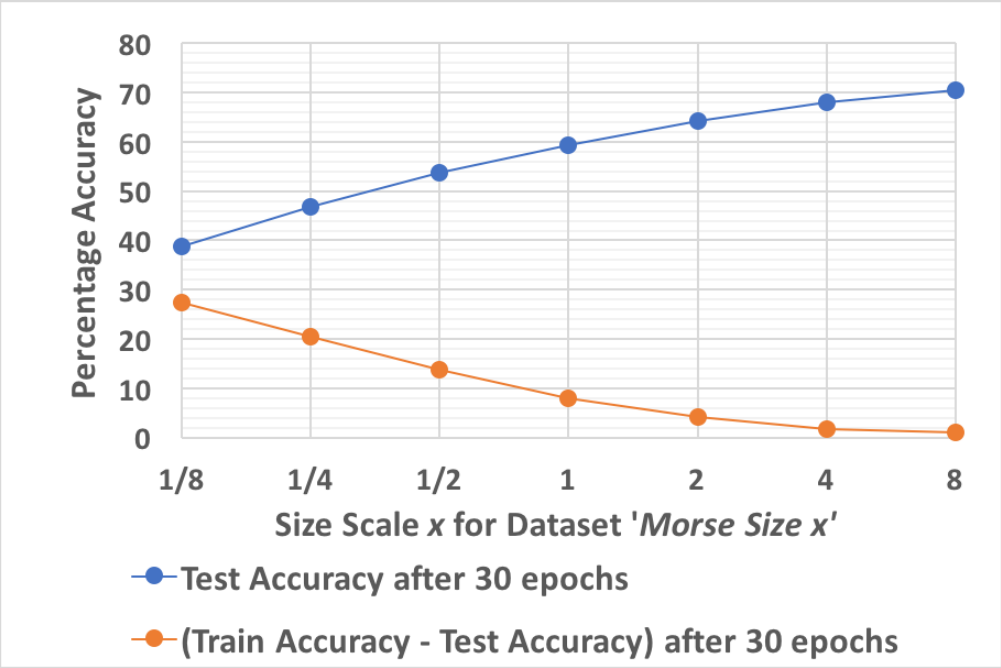


Figure 6.4: Effects of increasing the size of *Morse 3.1* by a factor of x on test accuracy after 30 epochs (blue), and (Training Accuracy − Test Accuracy) after 30 epochs (orange).

is an increase in the variance of the input to each neuron. As a result, accuracy drops. The degradation is worse when dashes can have a length of 3-9. Since the lengths are drawn from a uniform distribution, $1/7$th of dashes can now be confused with $1/3$rd of dots and $1/3$rd of intermediate spaces. As an example, for the + codeword which has 2 dashes, 3 dots and 4 intermediate spaces, there is a $(2/9 \times 1/7 + 3/9 \times 1/3 + 4/9 \times 1/3) = 29\%$ chance of this confusion occurring. Dilating by 4, however, reduces this chance to $(2/9 \times 1/25 + 3/9 \times 1/9 + 4/9 \times 1/9) = 9.5\%$. Accuracy is better as a result.

Increasing dataset size has a beneficial effect on performance. Giving the NN more examples to train from is akin to training on a smaller dataset for more

epochs, with the important added advantage that overfitting is reduced. This is shown in Fig. 6.4, which shows improving test accuracy as the dataset is made larger. At the same time, the difference between final training accuracy and test accuracy reduces, which implies that the network is generalizing better and not overfitting. Note that *Morse Size 8* has 3 million labeled training examples – a beneficial consequence of being able to cheaply generate large quantities of synthetic data.

6.2.2 Results for Pre-Defined Sparse Networks

Fig. 6.5 shows classification performance for 4 different Morse datasets when using our method of structured pre-defined sparse NNs. Both junctions are sparsified equally. Note how the baseline dataset only has mild performance degradation even when ρ_{net} is reduced to $1/4$, while performance drops off much more rapidly when dataset variations are introduced. These variations lead to increased information content per neuron, *i.e.* reduced redundancy. This enforces the trend observed in Section 3.3.2 of reduced redundancy datasets being less robust to sparsification.

Also note that as density is reduced, *Morse 4.2* has the best performance out of the non-baseline models tested in Fig. 6.5. This is because it has more weights to begin with, due to the increased number of input neurons. For example, the performance at $\rho_{\text{net}} = 1/4$ for *Morse 4.2* is better than FC for *Morse 2.1*. This enforces the trend of 'large and sparse' NNs performing better than 'small and

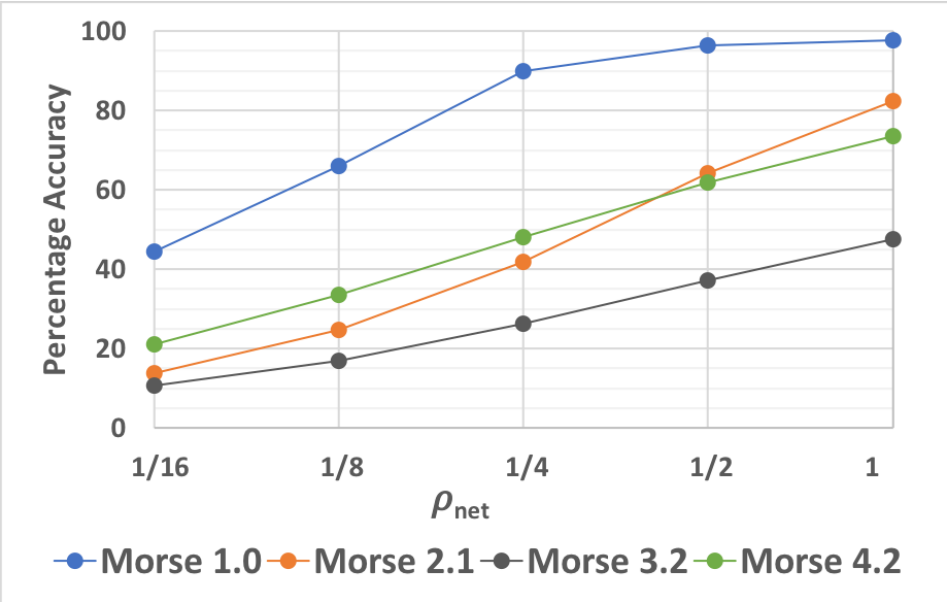


Figure 6.5: Effects of imposing pre-defined sparsity on classification performance for different Morse datasets.

dense' NNs, as discussed in Section 3.3.4. However, there is a subtle difference, while Section 3.3.4 experimented on varying numbers of hidden neurons, here it is the number of input neurons which varies.

We also experimented with the trend of latter junctions requiring a higher density than former, as discussed in Section 3.3.3, on the dataset *Morse 1.0*. Similar to the case of $\boldsymbol{N}_{\text{net}} = (39, 390, 39)$ for TIMIT, the NN used for Morse code has symmetric junctions since $\boldsymbol{N}_{\text{net}} = (64, 1024, 64)$. We set up the experiment somewhat differently than TIMIT. We swept over several possible (ρ_1, ρ_2) value pairs such that they all led to the same ρ_{net} value. We did this for two different ρ_{net} values – 25% in Fig. 6.6(a) and 50% in Fig. 6.6(b), and plot the peak validation accuracy obtained in 30 epochs. The black dashed line is for the case $\rho_1 = \rho_2$, and

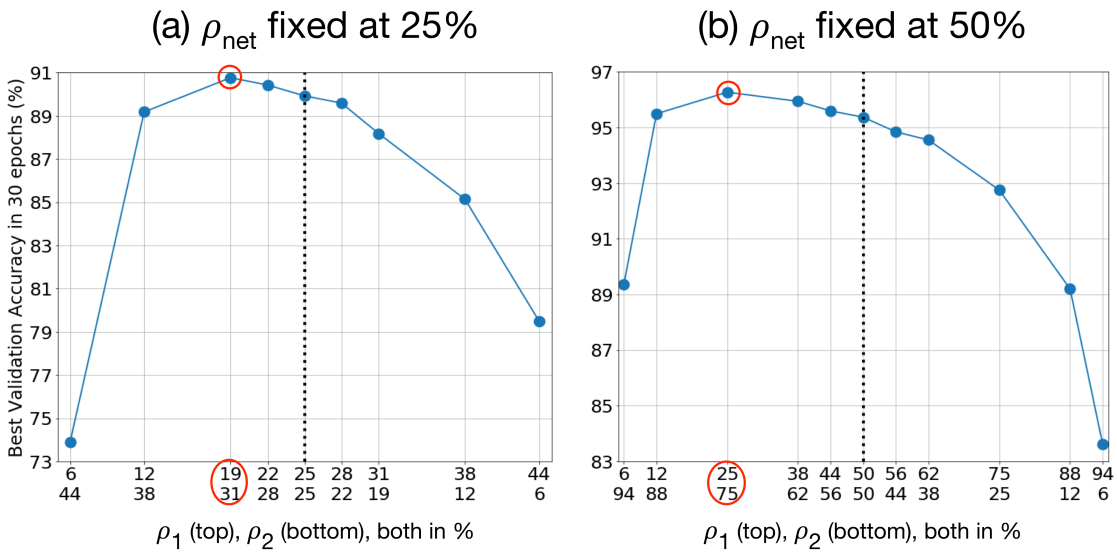


Figure 6.6: Validation performance results for varying ρ_1 (x-axis top) and ρ_2 (x-axis bottom) individually so as to keep ρ_{net} fixed at (a) 25%, (b) 50%. The black dashed line is when $\rho_1 = \rho_2$, while the red circles indicate peak performance. The NN has $\boldsymbol{N}_{\text{net}} = (64, 1024, 64)$ and dataset used is *Morse 1.0*.

should be the point where the performance peaks provided both junctions have equal importance. The fact that performance peaks *to the left* of the black dashed line for both figures (red circles) indicates that a higher value for ρ_2 is beneficial, as indicated by the tend in Section 3.3.3. For example when $\rho_{\text{net}} = 50\%$, performance peaks at $(\rho_1, \rho_2) = (25\%, 75\%)$. Also note that points to the extreme left and right go lower than the critical density for junctions 1 and 2, respectively.

6.3 Metrics for Dataset Difficulty

This section discusses possible metrics for quantifying how difficult a dataset is to classify. Each sample in a dataset is a point in an N_0-dimensional space. For

the Morse datasets (not considering dilation), $N_0 = 64$. There are N_L classes of points, which is also 64 in this case. The classification problem is essentially finding the class of any new point. Any machine learning classifier will attempt to learn and construct **decision boundaries** between the classes by partitioning the whole space into N_L regions. The samples of a particular class m are clustered in the mth region. Suppose a particular input sample actually belongs to class m. The classifier commits an error if it ranks some class j, $j \neq m$, higher than m when deciding where that input sample belongs. The probability of this happening is $P_{PW}(j|m)$, where subscript PW stands for pairwise and indicates that the quantity is specific to classes j and m. The overall probability of error $P(E)$ would also depend on the prior probability $P(m)$ of the mth class occurring. Considering all classes in the dataset, $P(E)$ is given as:

$$\sum_{m=1}^{N_L} P(m) \left[\max_{\substack{j \in \{1,2,\cdots,N_L\} \\ j \neq m}} P_{PW}(j|m) \right] \leq P(E) \leq \sum_{m=1}^{N_L} P(m) \sum_{\substack{j=1 \\ j \neq m}}^{N_L} P_{PW}(j|m) \quad (6.1)$$

This is a standard result in decision theory, cf. [126].

The pairwise probabilities can be approximately computed by assuming that the locations of samples of a particular class m are from a Gaussian distribution with mean located at the centroid c_m, which is the average of all samples for the class. To simplify the math, we take the average variance across all N_0 dimensions within a class – this gives the variance σ_m^2 for class m. The distance between 2

classes m and j is the L2-norm of the displacement vector between their centroids, *i.e.* $d(m,j) = \|c_m - c_j\|_2$. A particular class will be more prone to errors if it is close to other classes. This can be quantified by looking at $\frac{d_{\min}(m)}{\sigma_m}$, where the numerator is given as:

$$d_{\min}(m) = \min_{\substack{j \in \{1,2,\cdots,N_L\} \\ j \neq m}} d(m,j) \tag{6.2}$$

With the Gaussian assumption, (6.1) simplifies to $V_{\text{lower}} \leq P(E) \leq V_{\text{upper}}$, where:

$$V_{\text{lower}} = \sum_{m=1}^{N_L} P(m) Q\left(\sqrt{\frac{d_{\min}(m)^2}{4{\sigma_m}^2}}\right) \tag{6.3a}$$

$$V_{\text{upper}} = \sum_{m=1}^{N_L} P(m) \sum_{\substack{j=1 \\ j \neq m}}^{N_L} Q\left(\sqrt{\frac{d(m,j)^2}{4{\sigma_m}^2}}\right) \tag{6.3b}$$

where $Q(.)$ is the tail function for a standard Gaussian distribution. See [126] for a further exposition of these concepts.

The **lower and upper bounds for error** V_{lower} and V_{upper} can thus be used as metrics for dataset difficulty, since higher values for them imply higher probabilities of error, *i.e.* lower accuracy. A simpler metric can be obtained by just considering $\frac{\sigma_m}{d_{\min}(m)}$. Higher values for this indicate that a) class m is close to some other class and the NN will have a hard time differentiating between them, and b) Variance of class m is high, so it is harder to form a decision boundary to separate inputs

having labels m from those with other labels. Since $\frac{\sigma_m}{d_{\min}(m)}$ is different for every class, we experimented with ways to reduce it to a single measure such as taking the minimum, the average and the median. The average worked best, which gives our 3rd metric, which we call the **distance metric** V_{dist}:

$$V_{\text{dist}} = \frac{\sum_{m=1}^{N_L} \frac{\sigma_m}{d_{\min}(m)}}{N_L} \tag{6.4}$$

Therefore, high values of V_{dist} lead to low accuracy.

The 4th and final metric is the **threshold metric** V_{thresh}, which we came up with. To obtain this, we first compute the class centroids just as before. Then we compute the L1-norm between every pair of centroids and average over N_0, *i.e.* :

$$d_1(m, j) = \frac{\|c_m - c_j\|_1}{N_0} \tag{6.5}$$

Since all N_0 features in each input sample are normalized to $[0, 1]$, all the elements in all the centroid vectors also lie in the range $[0, 1]$. So the d_1 number for every pair of classes is always between 0 and 1, in fact, it is proportional to the absolute distance between the 2 classes. Then, we simply count how many of the d_1 numbers are less than a threshold, which we empirically set to 0.05. This gives V_{thresh}, *i.e.* :

$$V_{\text{thresh}} = \sum_{m=1}^{N_L} \sum_{\substack{j=1 \\ j \neq m}}^{N_L} \mathbb{I}\left(d_1(m, j) < 0.05\right) \tag{6.6}$$

Table 6.1: Correlation coefficients between metrics and accuracy

Metric	r
V_{lower}	-0.59
V_{upper}	-0.64
V_{dist}	-0.63
V_{thresh}	-0.64

The higher the value of V_{thresh}, the lower the accuracy. Note that the total number of d_1 values will be $\binom{N_L}{2}$, so the count for V_{thresh} will typically be higher for datasets that have more classes. This is a desired property since more number of classes usually makes a dataset harder to classify. Note that the maximum value of V_{thresh} for the Morse datasets is $\binom{64}{2} = 2016$.

6.3.1 Goodness of the Metrics

We computed V_{lower}, V_{upper}, V_{dist} and V_{thresh} values for all the Morse datasets and plotted these with the test set classification accuracy results obtained from the FC NN in Section 6.2.1. The results are shown in Fig. 6.7, while the Pearson correlation coefficient r of each metric with the accuracy is given in Table 6.1. Note that the metrics are an indicator of dataset difficulty, so they are negatively correlated with accuracy. It is apparent that the V_{upper} and V_{thresh} metrics are the best since their r values have the highest magnitude.

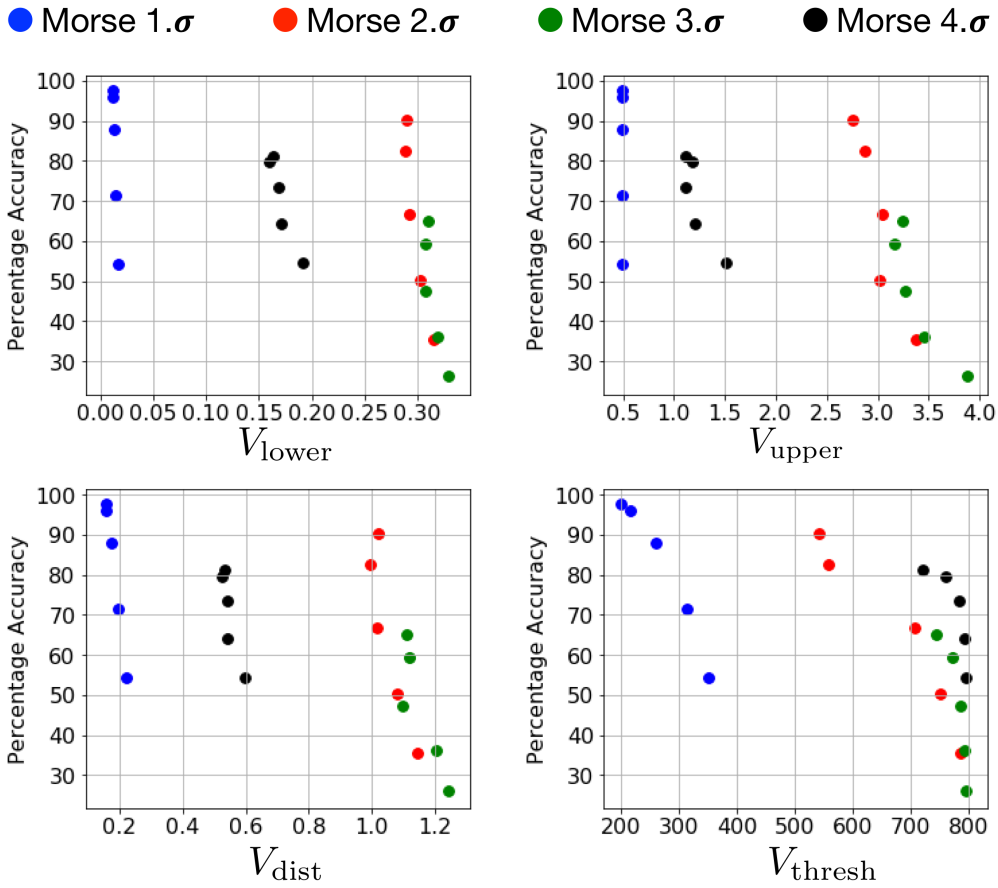


Figure 6.7: Plotting each metric for dataset difficulty vs. percentage accuracy obtained for datasets *Morse 1.*σ (blue), 2.σ (red), 3.σ (green) and 4.σ (black). The accuracy results are using the FC NN, as reported in Section 6.2.1. Color coding is just for clarity, the r values in Table 6.1 take into account all the points regardless of color.

6.3.2 Limitations of the Metrics

As mentioned, each class has a single variance value which is the average variance across dimensions. This is a reasonable simplification to make because our experiments indicate that the variance of the variance values for different dimensions is small. However, this simplification possibly leads to the error bounds V_{lower} and V_{upper} not being sufficiently tight. A possible improvement, involving significantly

more computation, would be to compute the $N_0 \times N_0$ covariance matrix K_m for each class.

It is worthwhile noting that **all these metrics are a function of the dataset only and are independent of the machine learning algorithm or training setup used**. On the other hand, percentage accuracy depends on the learning algorithm and training conditions. As shown in Fig. 6.4, increasing dataset size leads to accuracy improvement, *i.e.* the dataset becoming easier, since the NN has more training examples to learn from. However, increasing dataset size drives all the metric values towards indicating higher difficulty. This is because the occurrence of more examples in each class increases its standard deviation σ_m and also makes samples of a particular class more scattered, leading to reduced values for d and d_1. We hypothesize that these shortcomings of the metrics are due to the fact that most variations of the Morse datasets have low redundancy, while the metrics (the error bounds in particular) are designed for high signal-to-noise ratio (SNR) problems, *i.e.* high redundancy.

6.4 Summary

This chapter presented an algorithm to generate datasets of varying difficulty on classifying Morse code symbols. While the results have been shown for NNs, any machine learning algorithm can be tried and the challenge arising from more difficult datasets used to fine tune it. The datasets are synthetic and consequently

may not completely represent reality unless statistically verified with real-world tests. However, the different aspects of the generating algorithm help to mimic real-world scenarios which can suffer from noise or other inconsistencies. This chapter highlights one of the biggest advantages of synthetic data – the ability to easily produce large amounts of it and thereby improve the performance of learning algorithms. The given Morse datasets are also useful for testing the limits of various learning algorithms and identifying when they fail or possibly overfit/underfit. Finally, the metrics discussed, while not perfect, can be used to understand the inherent difficulty of the classification problem on any dataset before applying learning algorithms to it.

Chapter 7

Automated Machine Learning

This chapter discusses one of the major components of our research – automated machine learning (AutoML) – where our efforts encompass both CNNs and MLPs. In particular, we have developed an open-source framework to search for deep learning models balancing performance and complexity. We call it **Deep-n-Cheap**, which can be found at https://github.com/souryadey/deep-n-cheap [31].

7.1 Motivation and Related Work

Manually designing NNs, particularly CNNs, is challenging since they typically have a large number of interconnected layers and require a large number of decisions to be made regarding hyperparameters. As discussed in Chapter 2, hyperparameters, as opposed to trainable parameters like weights and biases, are not learned by the network. They need to be specified and adjusted by an external entity, *i.e.* the designer. They can be broadly grouped into two categories – a) architecture hyperparameters and training hyperparameters.

➤ **Definition 8: *Architecture hyperparameters*:** Architecture hyperparameters are the design choices concerned with the structure of NNs, including but not limited to the number, type and connection patterns across layers.

➤ **Definition 9: *Training hyperparameters*:** Training hyperparameters are the design choices relevant to performing the processes of training and inference in a NN, including but not limited to the initial learning rate and its scheduling, weight decay coefficient, and batch size.

The difficulty of manually designing hyperparameters to find a good NN is exacerbated by the fact that several hyperparameters interact with each other to have a combined effect on the final performance. The problem of searching for good NNs has resulted in several efforts towards automating this process. These efforts include **AutoML frameworks** such as Auto-Keras [17], AutoGluon [18] and Auto-PyTorch [19], which are open source software packages applicable to a variety of tasks and types of NNs. The major focus of these efforts is on providing user-friendly toolkits to search for good hyperparameter values.

Several other efforts place more emphasis on novel techniques for the search process. These can be broadly grouped into Neural Architecture Search (NAS) efforts such as [20, 21, 24, 127–133], and efforts that place a larger emphasis on training hyperparameters over architecture [23, 134–136]. An alternate grouping is on the basis of search methodology – a) reinforcement learning, where a controller NN is trained to optimize the search objective [20, 129, 137], b) evolution / genetic

operations, where a good NN architecture is selected using genetic operations such as crossover and mutation [21, 130, 131], c) Bayesian Optimization [22, 134, 136, 138,139], and/or d) one-off techniques such as tree-structured algorithms [135,140], the successive halving algorithm [23], and using hypernetworks [141]. Although the efforts described in this paragraph often come with publicly available software, they are typically not intended for general purpose use, *e.g.* the code release for [24] only allows reproducing NNs on two datasets. This differentiates them from AutoML frameworks.

As has been discussed in earlier chapters, deep NNs often suffer from **complexity** bottlenecks – either in storage, quantified by the **total number of trainable parameters N_p**, or computational, such as the number of Floating Point Operations per Second (FLOPS) or the time taken to perform training and/or inference. Prior efforts on NN search penalize inference complexity in specific ways – latency in [24], FLOPS in [132], and both in [133]. However, inference complexity is significantly different from training since the latter includes backpropagation and parameter updates every batch. For example, the resulting network for CIFAR-10 in [24] takes a minute to perform inference, but hours to train. Moreover, while there is considerable interest in popular benchmark datasets, in most real-world applications deep learning models need to be trained on custom datasets for which

readymade, pre-trained models do not exist [142–144]. This leads to an increasing number of resource-constrained devices needing to perform training on the fly, *e.g.* self-driving cars.

The computing platform and software libraries being used are also important, *e.g.* we found that changing batch size has a greater effect on training time per epoch on GPU than Central Processing Unit (CPU). Therefore, calculating the FLOP count is not always an accurate measure of the time and resources expended in training a NN. Platform specificity also applies to our work on pre-defined sparsity, *e.g.* multiplying sparse matrices using the `torch.sparse` package from the popular deep learning library Pytorch [145] does not generally lead to significant reductions in time as compared to multiplying dense matrices, as explored for various platforms in [146, 147]. Some other works attempting to reduce training time are [148] and [149] – the latter focuses on finding the quickest training time to get to a certain level of performance. However, all these efforts use manual methods, not search frameworks. In summary, we therefore identify training complexity reduction as one key goal for a NN search framework.

7.2 Overview of Deep-n-Cheap (DnC)

We have developed Deep-n-Cheap (DnC) – an open-source AutoML framework to search for deep learning models. We specifically target the training complexity bottleneck by including a penalty for **training time per epoch $\boldsymbol{t}_{\text{tr}}$** in our search

Figure 7.1: Deep-n-Cheap complete logo.

objective. The penalty coefficient can be varied by the user to obtain a family of networks trading off performance and complexity. Additionally, we also support storage complexity penalties for N_p.

DnC searches for both architecture and training hyperparameters. While the architecture search derives some ideas from literature, we have striven to offer the user a considerable amount of customizability in specifying the search space. This is important for training on custom datasets which can have significantly different requirements than those associated with benchmark datasets. DnC primarily uses Bayesian Optimization (BO) and currently supports classification tasks using CNNs and MLPs.

Key features / contributions of our AutoML work

1. **Complexity**: To the best of our knowledge, DnC is the only AutoML framework targeting training complexity reduction. We show results on several datasets on both GPU and CPU. Our models achieve performance comparable to state-of-the-art, with training times that are

1-2 orders of magnitude less than those for models obtained from other AutoML and search efforts.

2. **Usability**: DnC offers a highly customizable three-stage search interface for both architecture and training hyperparameters. As opposed to Auto-Keras and AutoGluon, our search includes a) batch size that affects training times, and b) architectures beyond pre-existing ones found in literature. As a result, our target users include those who want to train quickly on custom datasets. As an example, our framework achieves the highest performance and lowest training times on the Reuters RCV1 dataset, as described in 3.3.1.

3. **Insights**: We conduct investigations into the search process and draw several insights that will help guide a deeper understanding of NNs and search methodologies. A notable aspect is *search transfer*, where we found that the best NNs obtained from searching over one dataset give good performance on a different dataset. This helps to improve generalization in NNs – such as on custom datasets – instead of purely optimizing for specific problems. We also introduce a new similarity measure for BO and a new distance function for NNs. We empirically justify the value of our greedy three-stage search approach over less

Table 7.1: Comparison of Features of AutoML Frameworks

Framework	Architecture search space	Training hyp search	Adjust model complexity
Auto-Keras	Only pre-existing architectures	No	No
AutoGluon	Only pre-existing architectures	Yes	No
Auto-PyTorch	Customizable by user	Yes	No
Deep-n-Cheap	Customizable by user	Yes	Penalize t_{tr}, N_p

greedy approaches, and the superiority of BO over random and grid search.

The features of DnC compared to other AutoML frameworks are listed in Table 7.1. Further performance and complexity comparisons will be shown in Section 7.5.

7.3 Our Approach

Given a dataset, our framework searches for NN configurations through sequential stages in multiple search spaces. Each configuration is trained for the same number of epochs, *e.g.* 100. There have been works on extrapolating NN performance from limited training [128, 150], however we train for a large number of epochs to predict with significant confidence the final performance of a NN after convergence. Configs are mapped to objective values using:

$$f(\text{Config}) = \log\left(f_p + w_c f_c\right) \tag{7.1}$$

where w_c controls the importance given to reducing complexity. The goal of the search is to minimize f. Its components are:

$$\text{Performance term:} \quad f_p = 1 - (\text{Best Validation Accuracy}) \tag{7.2a}$$

$$\text{Complexity term:} \quad f_c = \frac{c}{c_0} \tag{7.2b}$$

where c is the complexity metric for the current configuration (either t_{tr} or N_p), and c_0 is a reference value for the same metric (typically obtained for a high complexity configuration in the space). Lower values of w_c focus more on performance, *i.e.* improving accuracy. One key contribution of this work is characterizing higher values of w_c that lead to reduced complexity NNs that train fast – these also reduce the search cost by speeding up the overall search process.

7.3.1 Three-stage search process

The search is divided into 3 stages, summarized in Fig. 7.2.

Stage 1 – Core architecture search

➤ **Definition 10:** ***Core architecture*:** The core architecture of a NN is defined as its depth and width. For CNNs, this is the number of convolutional layers (depth) and the number of channels in each (width). For MLPs, this is the number of hidden layers (depth) and the number of nodes in each (width).

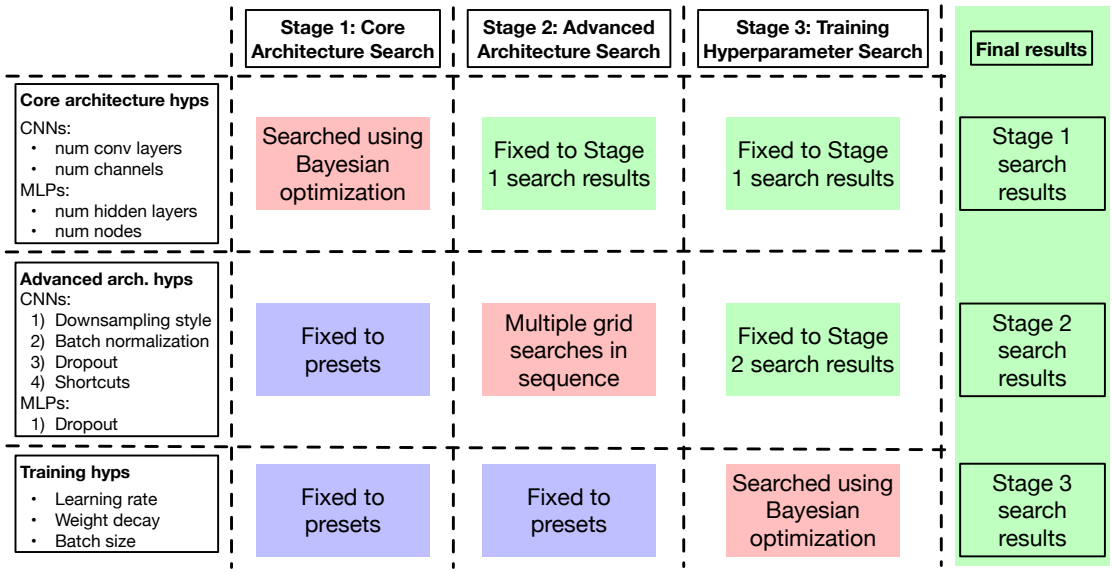


Figure 7.2: Three-stage search process for DnC.

The depth and width form the combined search space for Stage 1. Other architecture hyperparameters such as BN and dropout layers and all training hyperparameters are fixed to presets that we found to work well across a variety of datasets and network depths. BO is used to minimize f and the corresponding best configuration is the Stage 1 result.

Stage 2 – Advanced architecture search

➤ **Definition 11:** ***Advanced architecture*:** The advanced architecture of a NN is defined as the collection of all hyperparameters necessary to define the architecture of the NN apart from its width and depth.

This stage starts from the resulting architecture from Stage 1 and uses grid search to search for the following CNN hyperparameters through a sequence of

sub-stages – 1) whether to use strides or max pooling layers for downsampling, 2) amount of BN layers, 3) amount of dropout layers and drop probabilities, and 4) amount of shortcut connections, where each shortcut skips over 2 layers. This is not a combined space, instead grid search first picks the downsampling choice leading to the minimum f value, then freezes that and searches over BN, and so on. This ordering yielded good empirical results, however, reordering is supported by the framework. For MLPs, there is a single grid search for dropout probabilities. Note that the list of advanced architecture hyperparameters is not exhaustive. As in the previous stage, training hyperparameters are fixed to presets. The result from Stage 2 is the result from the final sub-stage.

Some architecture hyperparameters in Stage 2 are mapped to a space of fractions which indicate the fraction of convolutional layers to which the hyperparameter is applied. For example, a BN fraction of half indicates every other convolutional layer is followed by a BN layer. As another example, shortcut connections can be applied after every other convolutional layer (fraction 1), or after every 4th convolutional layer (fraction half). These examples are shown in Fig. 7.3.

Stage 3 – Training hyperparameter search

The architecture is finalized after Stage 2. In Stage 3 – identical for CNNs and MLPs – we search over the combined space of initial learning rate η, weight decay λ and batch size, using BO to minimize f. The result is combined with the

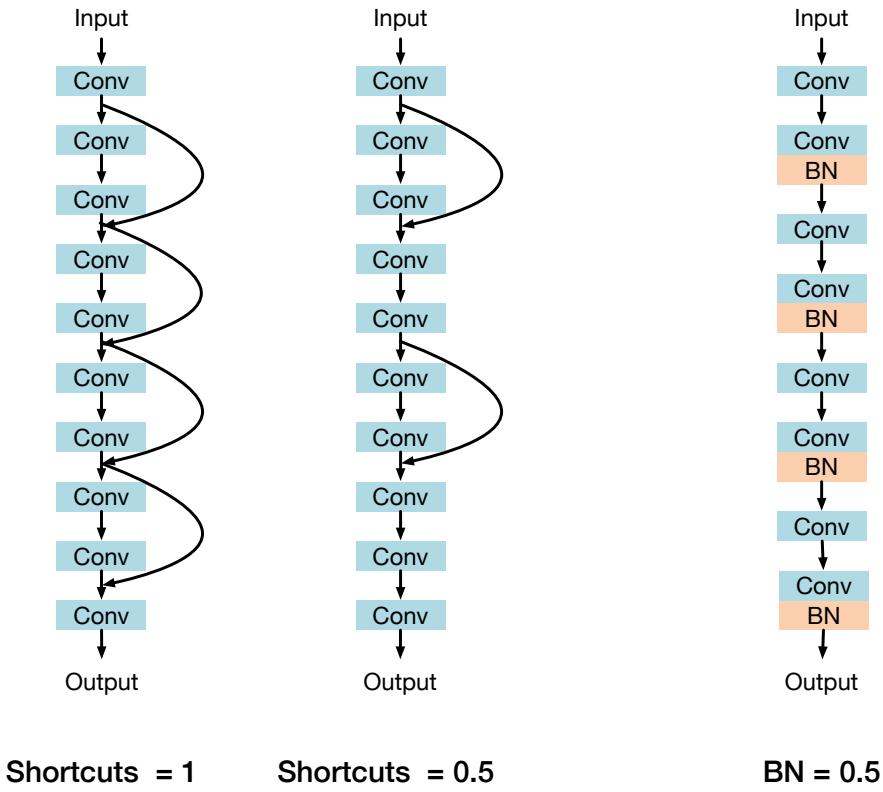


Figure 7.3: *Left* and *Middle*: Shortcut connections with fraction 1 and half, respectively. *Right*: BN with fraction half. Convolutional layers are shown in blue, BN layers in orange, and shortcut connections as black curved arrows.

architecture at the end of stage 2 to get the final configuration – comprising both architecture and training hyperparameters.

Other hyperparameters

We recognize that there are other hyperparameters in NNs that have not been explored in the present work, In particular, we keep activation function fixed to ReLU, use He normal initialization for weights and a constant value of 0.1 to initialize biases, always apply global average pooling after the convolutional portion,

fix pool sizes to 2×2, kernel sizes to 3×3, and do not apply pre-defined sparsity. While the present release – Deep-n-Cheap v1.0 does not support searching over these hyperparameters – these are excellent topics for future extensions to the framework.

7.3.2 Bayesian Optimization

Bayesian Optimization is useful for optimizing functions that are black-box and/or expensive to evaluate such as f, which requires NN training. Moreover, derivatives of f do not exist since the search space is discrete, and f is non-convex and noisy. These factors also combine to make BO an excellent choice.

The initial step when performing BO is to sample n_1 configurations from the search space, $\{\boldsymbol{x}_1, \cdots, \boldsymbol{x}_{n_1}\}$, calculate their corresponding objective values, $\{f(\boldsymbol{x}_1), \cdots, f(\boldsymbol{x}_{n_1})\}$, and form a Gaussian prior. The mean vector $\boldsymbol{\mu}$ is filled with the mean of the f values, and covariance matrix $\boldsymbol{\Sigma}$ is such that $\Sigma_{ij} = \sigma(\boldsymbol{x}_i, \boldsymbol{x}_j)$, where $\sigma(\cdot, \cdot)$ is a *kernel function* that takes a high value $\in [0, 1]$ if configurations $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are similar.

Then the algorithm continues for n_2 steps, each step consisting of sampling n_3 configurations, picking the configuration with the maximum *acquisition function value*, computing its f value, and updating $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ accordingly. Computing the acquisition function is significantly faster than computing f, since the former does

not require training a NN. Note that BO explores a total of $n_1 + n_2 n_3$ states in the search space, but the expensive f computation only occurs for $n_1 + n_2$ states.

The reader is referred to [151] for a complete tutorial on BO.

Similarity between NN configurations

We begin by defining the *distance* between values of a particular hyperparameter k for two configurations $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$. Larger distances denote dissimilarity. We initially considered the distance functions defined in Sections 2 and 3 of [152], but then adopted an alternate one that resulted in similar performance with less tuning. We call it the *ramp* distance:

$$d\left(x_{ik}, x_{jk}\right) = \omega_k \left(\frac{|x_{ik} - x_{jk}|}{u_k - l_k} \right)^{r_k} \tag{7.3a}$$

where u_k and l_k are respectively the upper and lower bounds for k, ω_k is a scaling coefficient, and r_k is a fractional power used for stretching small differences. Note that d is 0 when $x_{ik} = x_{jk}$, and reaches a maximum of ω_k when they are the furthest apart. x_{ik} and x_{jk} are computed in different ways depending on k:

- If k is batch size or number of layers, x_{ik} and x_{jk} are the actual values.
- If k is η or λ, x_{ik} and x_{jk} are the logarithms of the actual values.
- When k is the hidden node configuration of a MLP, we sum the nodes together across all hidden layers. This is because we found that the sum has

a greater impact on f than considering layers individually, *e.g.* a configuration with three 300-node hidden layers has a closer f value to a configuration with one 1000-node hidden layer than a configuration with three 100-node hidden layers.

- When k is the convolutional channel configuration of a CNN, we calculate individual distances for each layer. If the number of layers is different, the distance is maximum for each of the extra layers, *i.e.* ω. This idea is inspired from [152], as compared to alternative similarity measures in [17, 22]. We follow this layer-by-layer comparison because our prior experiments showed that the representations learned by a certain convolutional layer in a CNN are similar to those learned by layers at the same depth in different CNNs. Additionally, this approach performed better than the summing across layers as in MLPs.

Each individual distance $d\left(x_{ik}, x_{jk}\right)$ is converted to its kernel value $\sigma\left(x_{ik}, x_{jk}\right)$ using the squared exponential function, then we take their convex combination for all K hyperparameters using coefficients $\{s_k\}$ to finally get $\sigma\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right)$. An example is given in Fig. 7.4.

$$\sigma\left(x_{ik}, x_{jk}\right) = \exp\left(-\frac{d^2(x_{ik}, x_{jk})}{2}\right) \tag{7.3b}$$

$$\sigma\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) = \sum_{k=1}^{K} s_k \sigma\left(x_{ik}, x_{jk}\right) \tag{7.3c}$$

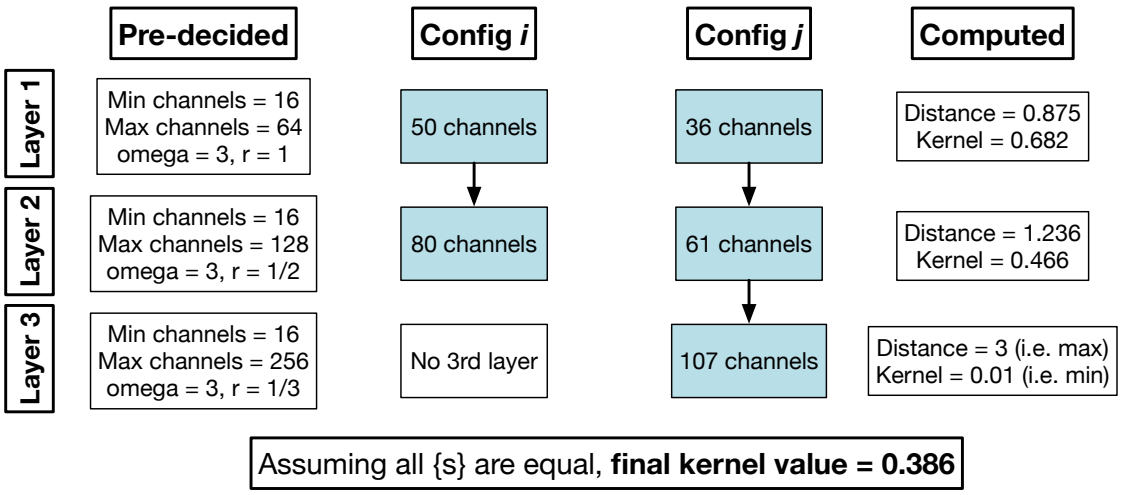


Figure 7.4: Calculating Stage 1 similarity for two convolutional channel configurations: $\boldsymbol{x}_i = [50, 80]$ and $\boldsymbol{x}_j = [36, 61, 107]$. Taking the 1st convolutional layer as an example, the pre-decided values are $u_1 = 64$, $l_1 = 16$, $\omega_1 = 3$ and $r_1 = 1$ (more details on these choices in Sec. 7.4). The distance is $d_1 = 3 \times [(50 - 36)/(64 - 16)]^1 = 0.875$, and kernel value is $\sigma_1 = \exp\left(-0.5 \times 0.875^2\right) = 0.682$. Similarly we get $\sigma_2 = 0.466$ and $\sigma_3 = 0.01$ (note that $d_3 = \omega_3$ due to the absence of the 3rd layer in $\boldsymbol{x}_i$). Combining these using $s_1 = s_2 = s_3 = 1/3$ yields $\sigma\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) = 0.386$.

The validity of our covariance kernel can be proved as follows. We note that since x_{ik} and x_{jk} are scalars, d in eq. (7.3a) is the Euclidean distance. It follows from the properties of the squared exponential kernel that $\sigma\left(x_{ik}, x_{jk}\right)$ in eq. (7.3b) is a valid kernel function. So if a kernel matrix $\boldsymbol{\Sigma_k}$ were to be formed such that $\Sigma_{k_{ij}} = \sigma\left(x_{ik}, x_{jk}\right)$, then $\boldsymbol{\Sigma_k}$ would be positive semi-definite. Writing eq. (7.3c) in matrix form gives $\boldsymbol{\Sigma} = \sum_{k=1}^{K} s_k \boldsymbol{\Sigma_k}$. Since a convex combination of positive semi-definite matrices is also positive semi-definite, it follows that $\boldsymbol{\Sigma}$ is a valid covariance matrix.

Acquisition function

The acquisition function we choose is **expected improvement**, given as:

$$EI(\boldsymbol{x}) = (f^* - \mu_{\text{post}} - \xi)P\left(\frac{f^* - \mu_{\text{post}} - \xi}{\sigma_{\text{post}}}\right) + \sigma_{\text{post}}p\left(\frac{f^* - \mu_{\text{post}} - \xi}{\sigma_{\text{post}}}\right) \quad (7.4)$$

where f^* is the current optimum value of f, μ_{post} and σ_{post} are the posterior mean and covariance values obtained by updating the distribution with the current state, $P(\cdot)$ and $p(\cdot)$ are respectively the cumulative and probability density functions of the standard normal distribution, and ξ trades off exploration vs exploitation.

7.4 Experimental Results

This section presents results of our search framework on different datasets for both CNN and MLP classification problems, along with the search settings used. Note that most of these *settings can be customized by the user* – this leads to one of our key contributions of using limited knowledge from literature to enable wider exploration of NNs for various custom problems. We used the Pytorch library on two platforms: a) *GPU* – an Amazon Web Services p3.2xlarge instance that uses a single NVIDIA V100 GPU with 16 GB memory and 8 vCPUs, and b) *CPU* – a mid-2014 Macbook Pro CPU with 2.2 GHz Intel Core i7 processor and 16GB 1.6 GHz DDR3 RAM. For BO, we used $n_1 = n_2 = 15$ and $n_3 = 1000$.

7.4.1 Datasets and loading

We used the following datasets for CNN experiments:

- CIFAR images: As described in Section 3.3.1, the CIFAR images are of dimensions $(c, h, w) = (3, 32, 32)$, split into 40,000 for training, 10,000 for validation, and 10,000 for test. We applied **standard augmentation** in the form of channel-wise normalization, random crops from 4 pixel padding on each side, and random horizontal flips.

- Fashion MNIST (FMNIST) images: The FMNIST dataset has the same dimensions as MNIST, *i.e.* $(c, h, w) = (1, 28, 28)$, split into 50,000 for training, 10,000 for validation, and 10,000 for test.

Augmentation requires Pytorch data loaders that incur timing overheads as compared to storing the whole dataset in memory as a single unit. As a result, we show results on unaugmented CIFAR-10 as well, where t_{tr} is significantly less compared to the augmented case.

For MLP experiments, we used the Reuters RCV1 corpus and the MNIST dataset in permutation-invariant format, both of which are described in Section 3.3.1. Additionally, we also used the FMNIST in permutation-invariant format, *i.e.* 784 input features, similar to MNIST. None of the MLP datasets use augmentation, *i.e.* there are no timing overheads due to data loaders.

7.4.2 Convolutional Neural Networks

All CNN experiments are on GPU.

For **Stage 1**, we use BO to search over CNNs with 4–16 convolutional layers, the first of which has $c_1 \in \{16, 17, \cdots, 64\}$ channels and each subsequent layer has $c_{i+1} \in \{c_i, c_i + 1, \cdots, \min(2c_i, 512)\}$ channels. We allow the number of channels in a layer to have arbitrary integer values, not just fixed to multiples of 8. Kernel sizes are fixed to 3x3. Downsampling precedes layers where c_i crosses 64, 128 and 256 (this is due to GPU memory limitations). During Stage 1, all convolutional layers are followed by BN and dropout with 30% drop probability. Configs with more than 8 convolutional layers have shortcut connections. Global average pooling and a softmax classifier follows the convolutional portion. There are no hidden classifier layers since we empirically obtained no performance benefit. For both Stages 1 and 2, we used the default Adam optimizer with $\eta = 10^{-3}$, decayed by 80% at the half and three-quarter points of training, batch size of 256, and $\lambda = \mathbb{I}(N_p \geq 10^6) \times N_p/10^{11}$, $\mathbb{I}$ being the indicator function. We empirically found this rule to work well.

For **Stage 2**, the first grid search is over all possible combinations of using either strides or max pooling for the downsampling layers. Second, we vary the fraction of BN layers through $[0, 1/4, 1/2, 3/4]$. For example, if there are 7 convolutional layers, a setting of $1/2$ will place BN layers after convolutional layers 2, 4, 6 and 7. Third, we vary the fraction of dropout layers in a manner similar to BN, and

drop probabilities over $[0.1, 0.2]$ for the input layer and $[0.15, 0.3, 0.45]$ for all other layers. Finally, we search over shortcut connections – none, every 4th layer, or every other layer. Note that any shortcut connection skips over 2 layers.

For **Stage 3**, we used BO to search over a) $\eta \in \{10^x\}$ for $x \in [1, 5]$, b) $\lambda \in \{10^x\}$ for $x \in [-6, -3]$, with λ converted to 0 when $x < -5$, and c) batch sizes in $[32, 33, \cdots, 512]$. We found that batch sizes that are not powers of 2 did not lead to any slowdown on the platforms used.

The penalty function f_c uses normalized t_{tr}, since this is the major bottleneck in developing CNNs. Each configuration was trained for 100 epochs on the train set and evaluated on the validation set to obtain f_p. We ran experiments for 5 different values of w_c: $[0, 0.01, 0.1, 1, 10]$. The best network from each search was then trained on the combined training and validation set, and evaluated on the test set for 300 epochs to get final test accuracies and t_{tr} values.

As shown in Fig. 7.5, we obtain a family of networks by varying w_c. Performance in the form of test accuracy trades off with complexity in the form of t_{tr}. The latter is correlated with search cost and N_p (albeit not strictly). The last row of figures directly plot the performance-complexity tradeoff. These curves rise sharply towards the left and flatten out towards the right, indicating diminishing performance returns as complexity is increased. This highlights one of our key contributions – allowing the user to choose fast training NNs that perform well.

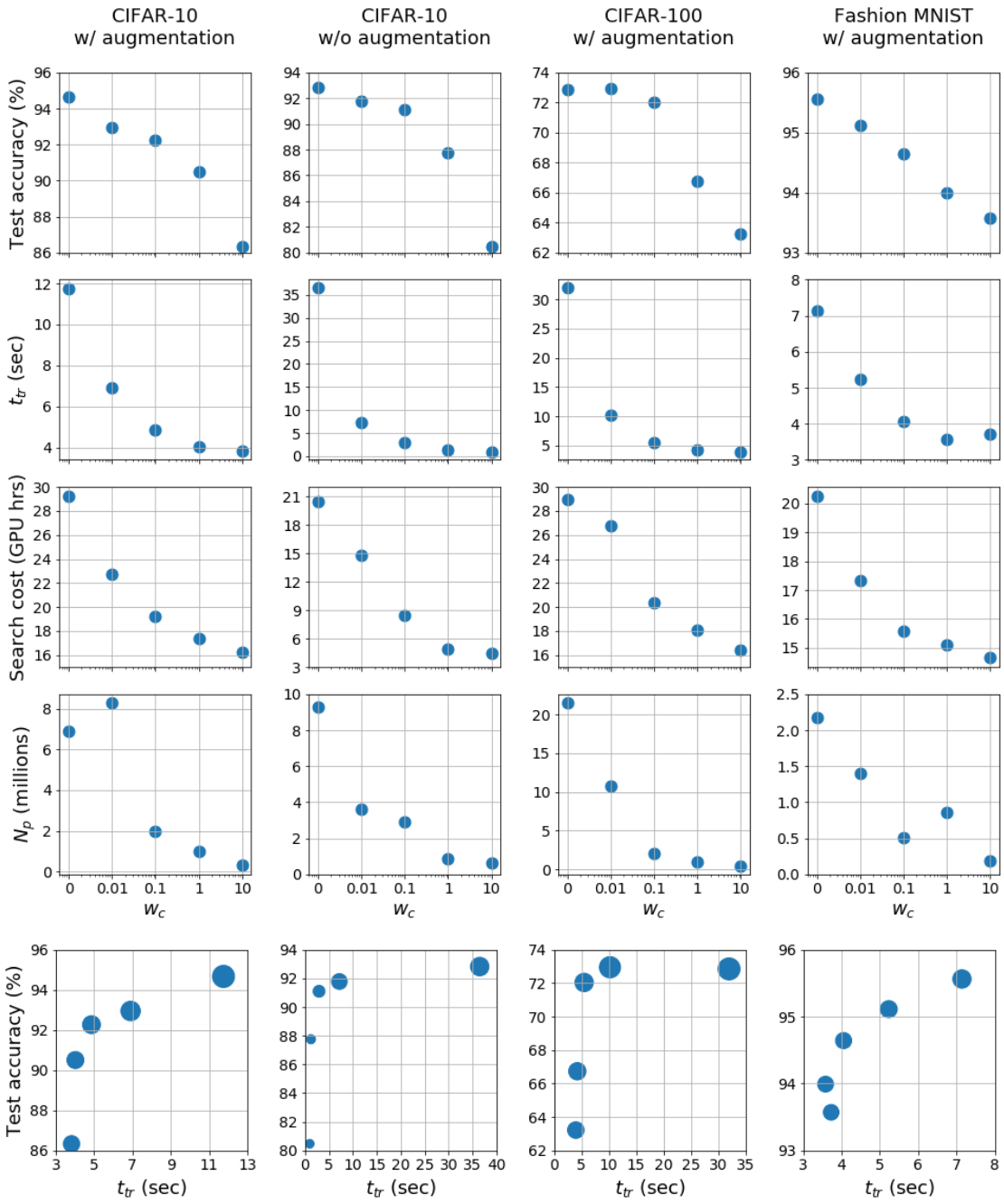


Figure 7.5: Characterizing a family of NNs for CIFAR-10 augmented (1st column), unaugmented (2nd column), CIFAR-100 augmented (3rd column) and FMNIST augmented (4th column), obtained from DnC for different w_c. We plot test accuracy in 300 epochs (1st row), t_{tr} on combined train and validation sets (2nd row), search cost (3rd row) and N_p (4th row), all against w_c. The 5th row shows the performance-complexity tradeoff, with dot size proportional to search cost.

Figure 7.6: The best configurations found by DnC for CIFAR-10 augmented for $w_c = [0, 0.01, 0.1, 1, 10]$ from left to right. Each configuration shows the architecture with input at top and output at bottom. Blue rectangles denote convolutional layers with their c value and stride shown as '/2', orange BN, purple dropout with drop probability, green pooling, and finally yellow denotes the softmax classifier. The table at the bottom shows the best training hyperparameter values obtained.

Taking augmented CIFAR-10 as an example, the best configurations found by DnC are shown in Fig. 7.6. Note that we achieve good performance with NNs that have irregular $\{c\}$ values and are also not very deep, in particular, the $w_c = 0$ configuration just has 14 convolutional layers and yields a test accuracy of almost 95%. These findings are consistent with those in [153], which discussed the **benefits of using wider and shallower networks**, *i.e.* less layers with more channels. As regards the effect of w_c, note how this architecture has a BN layer following every convolutional layer, while the architecture for $w_c = 10$ has no BN layers so as to reduce t_{tr}. Note also how the weight decay λ values are strictly correlated with the N_p values in the 4th row, 1st column of Fig. 7.5. This is expected since more trainable parameters implies a higher chance of overfitting, hence requiring a larger L2 regularization coefficient. Finally, it is interesting that we obtain a best η of 0.001 in most cases – as suggested in the original Adam paper [41].

7.4.3 Multilayer Perceptrons

We ran CPU experiments on the MNIST and FMNIST datasets, and GPU experiments on the Reuters RCV1 dataset.

For **Stage 1**, we search over 0–2 hidden layers for MNIST and FMNIST, number of nodes in each being 20–400. These numbers change for RCV1 to 0–3 and 50–1000 since it is a larger dataset. Every layer is followed by a dropout layer with

20% drop probability. Training hyperparameters are fixed as in the case of CNNs, with the difference that $\lambda = \mathbb{I}(N_p \geq 10^4) \times N_p/10^9$ for MNIST and FMNIST and $\lambda = \mathbb{I}(N_p \geq 10^5) \times N_p/10^{10}$ for RCV1. For **Stage 2**, we do a grid search over drop probabilities in $[0, 0.1, 0.3, 0.4, 0.5]$, and for **Stage 3**, the training hyperparameter search is identical to CNNs.

We ran separate searches for individual penalty functions – normalized t_{tr} and normalized N_p. The latter is owing to the fact that MLPs often massively increase the number of parameters and thereby storage complexity of NNs [1]. The train-validation-test splits for MNIST and FMNIST are 50k-10k-10k, and 178k-50k-100k for RCV1. Candidate networks were trained for 60 epochs and the final networks tested after 180 epochs. As before, $w_c \in [0, 0.01, 0.1, 1, 10]$ for MNIST and FMNIST. For RCV1, the results for $w_c = 10$ were mostly similar to $w_c = 1$, so we replace 10 with 0.03. The plots against w_c are shown in Fig. 7.7, where pink dots are for t_{tr} penalty and black crosses are for N_p penalty.

The trends in Fig. 7.7 are qualitatively similar to those in Fig. 7.5. When penalizing N_p, the two lowest complexity networks in each case have no hidden layers, so they both have exactly the same N_p (results differ due to different training hyperparameters). Of interest is the subfigure on the bottom right, indicating much longer search times when penalizing N_p as compared to t_{tr}. This is because time is not a factor when penalizing N_p, so the search picks smaller batch sizes that increase t_{tr} with a view to improving performance. Interestingly enough, this does

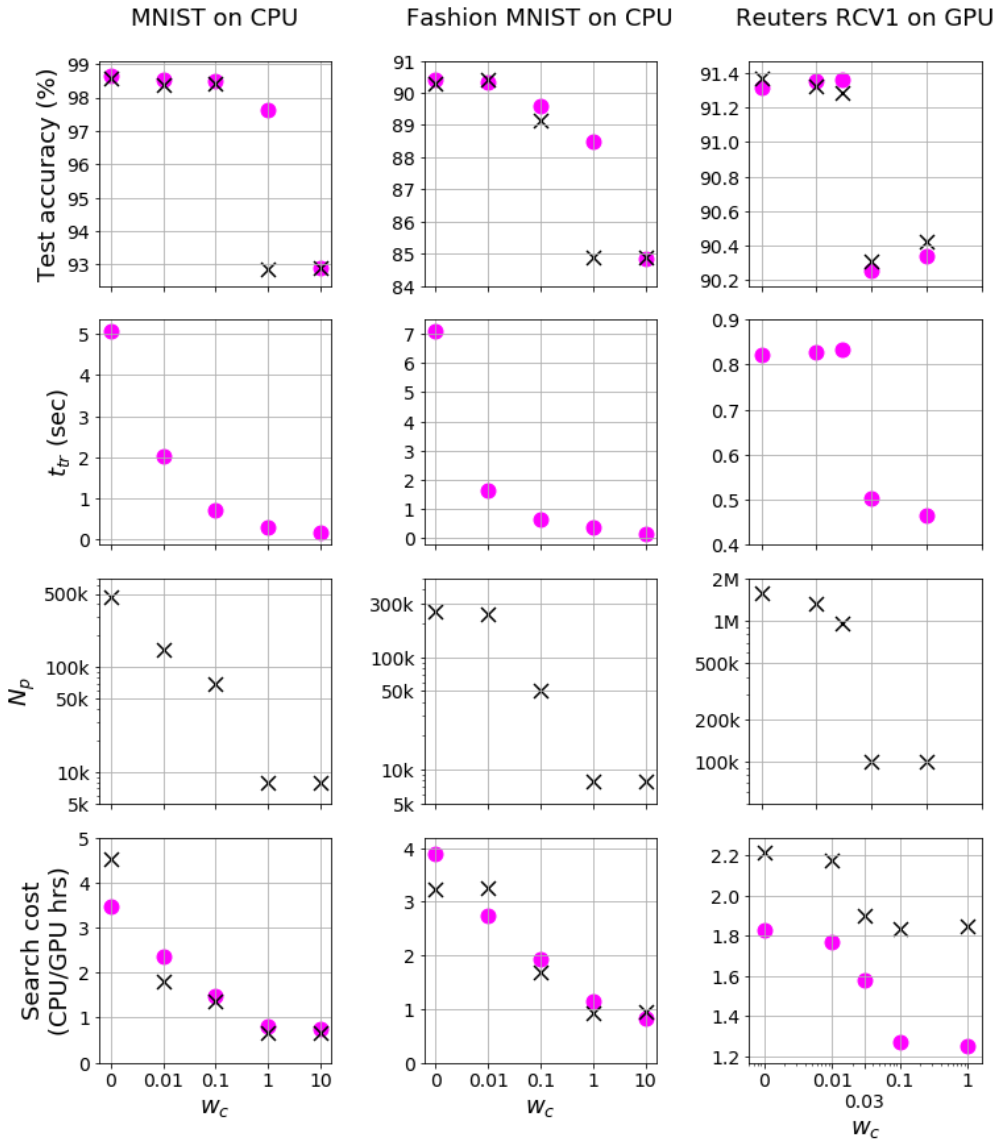


Figure 7.7: Characterizing a family of NNs for MNIST (1st column) and FMNIST (2nd column) on CPU, and RCV1 (3rd column) on GPU, obtained from DnC for different w_c. We plot test accuracy in 180 epochs (1st row), t_{tr} on combined train and validation sets (2nd row), N_p (3rd row), and search cost (4th row), all against w_c. The search penalty is t_{tr} for the pink dots and N_p for the black crosses.

not actually lead to performance benefit as shown in the subfigure on the top-right, where the black crosses occupy similar locations as the pink dots.

7.5 Comparison to related work

To the best of our knowledge, only DnC allows the user to specifically penalize complexity of the resulting models. This allows our framework to find models with performance comparable to other state-of-the-art methods, while significantly reducing the computational burden of training. This is shown in Table 7.2, which compares the search process and metrics of the final model found for CNNs on CIFAR-10, and Table 7.3, which does the same for MLPs on FMNIST and RCV1 for DnC and Auto-PyTorch only, since Auto-Keras and AutoGluon do not have explicit support for MLPs at the time of writing[1]. The comparisons were achieved with the help of Saikrishna Chaitanya Kanala.

Note that Auto-Keras and AutoGluon do not support explicitly obtaining the final model from the search, which is needed to perform separate inference on the test set after the search. As a result, in order to have a fair comparison, Tables 7.2 and 7.3 use metrics from the search process – t_{tr} is for the train set and the performance metric is best validation accuracy. These are reported for the best model found from each search. Auto-Keras and AutoGluon use fixed batch sizes across all models, however, Auto-PyTorch and DnC also do a search over batch sizes. We have included batch size since it affects t_{tr}. Each configuration for

[1]There is no mention in the documentation and multiple unresolved issues on Github regarding the absence of MLPs

Table 7.2: Comparing Frameworks on CNNs for CIFAR-10 augmented on GPU

Framework	Additional settings	Search cost (GPU hrs)	Best model found from search			
			Architecture	t_{tr} (sec)	Batch size	Best val acc (%)
Proxyless NAS[a]	Proxyless-G	96	537 convolutional layers	429	64	93.22
Auto-Keras[b]	Default run	14.33	Resnet-20 v2	33	32	74.89
AutoGluon	Default run	**3**	Resnet-20 v1	37	64	88.6
	Extended run	101	Resnet-56 v1	46	64	91.22
Auto-Pytorch	'tiny cs'	6.17	30 convolutional layers	39	64	87.81
	'full cs'	6.13	41 convolutional layers	31	106	86.37
Deep-n-Cheap	$w_c = 0$	29.17	14 convolutional layers	10	120	**93.74**
	$w_c = 0.1$	19.23	8 convolutional layers	4	459	91.89
	$w_c = 10$	16.23	4 convolutional layers	**3**	256	83.82

[a]Since Proxyless NAS is a search methodology as opposed to an AutoML framework, we trained the final best model provided to us by the authors [154]. This model was trained in [24] using stochastic depth and additional cutout augmentation [154] – yielding an impressive 97.92% accuracy on their test set. The result shown here was obtained without cutout or stochastic depth, and the validation accuracy is reported to compare with the metrics available from Auto-Keras and AutoGluon. The primary point of including Proxyless NAS is to compare to a model with state-of-the-art accuracy that has been highly optimized for CIFAR-10.

[b]Auto-Keras does not support image augmentation at the time of writing this paper [155], so we used results from the unaugmented dataset.

each search is run for the same number of epochs, as described in Sec. 7.4. The exception is Auto-PyTorch, where a key feature is variable number of epochs.

We note that for CNNs, DnC results in both the fastest t_{tr} and highest performance. The performance of Proxyless NAS is comparable, while taking 43X more time to train. This highlights **one of our key features – the ability to find models with performance comparable to state-of-the-art while massively reducing training complexity**. The search cost is lowest for the default AutoGluon run, which only runs 3 configurations. We also did an extended run for $\sim$ 100 models on AutoGluon to make it match with DnC and Auto-Keras – this results in the longest search time without significant performance gain.

Table 7.3: Comparing AutoML Frameworks on MLPs for FMNIST and RCV1 on GPU

Framework	Additional settings	Search cost (GPU hrs)	Best model found from search				
			MLP layers	N_p	t_{tr} (sec)	Batch size	Best val acc (%)
Fashion MNIST							
Auto-Pytorch	'tiny cs'	6.76	50	27.8M	19.2	125	**91**
	'medium cs'	5.53	20	3.5M	8.3	184	90.52
	'full cs'	6.63	12	122k	5.4	173	90.61
Deep-n-Cheap (penalize t_{tr})	$w_c = 0$	0.52	3	263k	0.4	272	90.24
	$w_c = 10$	**0.3**	1	**7.9k**	**0.1**	511	84.39
Deep-n-Cheap (penalize N_p)	$w_c = 0$	0.44	2	317k	0.5	153	90.53
	$w_c = 10$	0.4	1	**7.9k**	0.2	256	86.06
Reuters RCV1							
Auto-Pytorch	'tiny cs'	7.22	38	19.7M	39.6	125	88.91
	'medium cs'	6.47	11	11.2M	22.3	337	90.77
Deep-n-Cheap (penalize t_{tr})	$w_c = 0$	1.83	2	1.32M	0.7	503	**91.36**
	$w_c = 1$	**1.25**	1	**100k**	**0.4**	512	90.34
Deep-n-Cheap (penalize N_p)	$w_c = 0$	2.22	2	1.6M	0.6	512	**91.36**
	$w_c = 1$	1.85	1	**100k**	5.54	33	90.4

For MLPs, DnC has the fastest search times and lowest t_{tr} and N_p values – this is a result of it searching over simpler models with few hidden layers. While Auto-PyTorch performs slightly better for the benchmark FMNIST, our framework gives better performance for the more customized RCV1 dataset.

7.6 Investigations and insights

7.6.1 Search transfer

One goal of our search framework is to find models that are applicable to a wide variety of problems and datasets suited to different user requirements. To evaluate this aspect, we experimented on whether a NN architecture found from searching

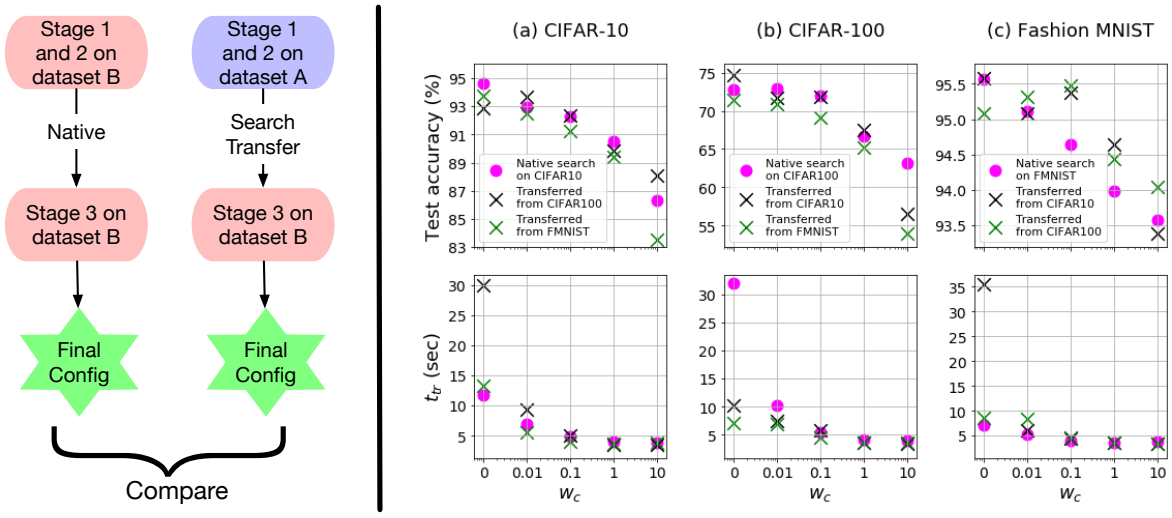


Figure 7.8: *Left*: Process of search transfer – comparing configurations obtained from native search with those where Stage 3 is done on a dataset different from Stages 1 and 2. *Right*: Results of CNN search transfer to (a) CIFAR-10, (b) CIFAR-100, (c) FMNIST. All datasets are augmented. Pink dots denote native search.

through Stages 1 and 2 on dataset A can be applied to dataset B after searching for Stage 3 on it. In other words, how does transferring an architecture compare to 'native' configurations, *i.e.* those searched for through all three stages on dataset B. This process is shown on the left in Fig. 7.8. Note that we repeat Stage 3 of the search since it optimizes training hyperparameters such as weight decay, which are related to the capacity of the network to learn a new dataset. This is contrary to simply transferring the architecture as in [137].

We took the best CNN architectures found from searches on CIFAR-10, CIFAR-100 and FMNIST (as depicted in Fig. 7.5) and transferred them to each other for Stage 3 searching. The results for test accuracy and t_{tr} are shown on the right in Fig. 7.8. We note that the architectures generally transfer well. In particular,

transferring from FMNIST (green crosses in subfigures (a) and (b)) results in slight performance degradation since those architectures have N_p around 1M-2M, while some architectures found from native searches (pink dots) on CIFAR have $N_p > 20$M. However, architectures transferred between CIFAR-10 and -100 often exceed native performance. Moreover, almost all the architectures transferred from CIFAR-100 (green crosses in subfigure (c)) exceed native performance on FMNIST, which again is likely due to bigger N_p. We also note that t_{tr} values remain very similar on transferring, except for the $w_c = 0$ case where there is absolutely no time penalty.

7.6.2 Greedy strategy

Our search methodology is greedy in the sense that it preserves only the best configuration resulting in the minimum f value from each stage and sub-stage. We also experimented with a non-greedy strategy. Instead of one, we picked the three best configurations from Stage 1 – $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3\}$, then ran separate grid searches on each of them to get three corresponding configurations at the end of Stage 2, and finally picked the three best configurations for each of their Stage 3 runs for a total of nine different configurations – $\{\boldsymbol{x}_{11}, \boldsymbol{x}_{12}, \boldsymbol{x}_{13}, \boldsymbol{x}_{21}, \cdots, \boldsymbol{x}_{33}\}$. Following a purely greedy approach would have resulted in only $\boldsymbol{x}_{11}$, while following a greedy approach for Stages 1 and 2 but not Stage 3 would have resulted in $\{\boldsymbol{x}_{11}, \boldsymbol{x}_{12}, \boldsymbol{x}_{13}\}$. We plotted the losses for each configuration for five different values of w_c on CIFAR-10

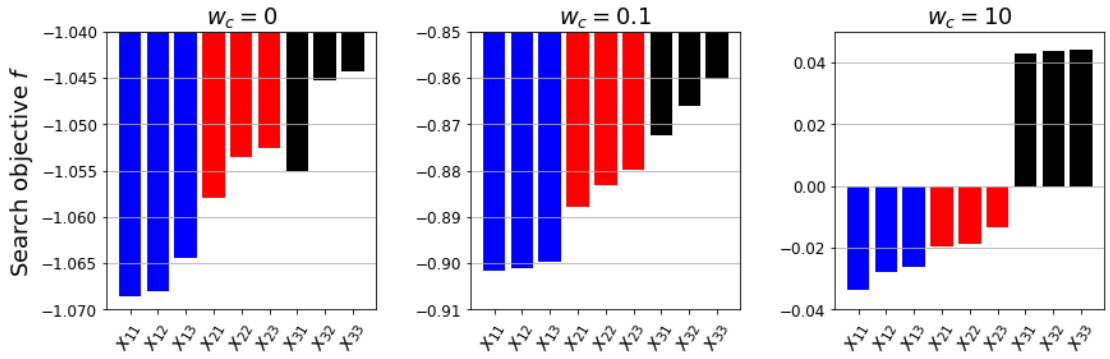


Figure 7.9: Search objective values (lower the better) for three best configurations from Stage 1 (blue, red, black), optimized through Stages 2 and 3 and three best configurations chosen for each in Stage 3. Results shown for different w_c on CIFAR-10 unaugmented.

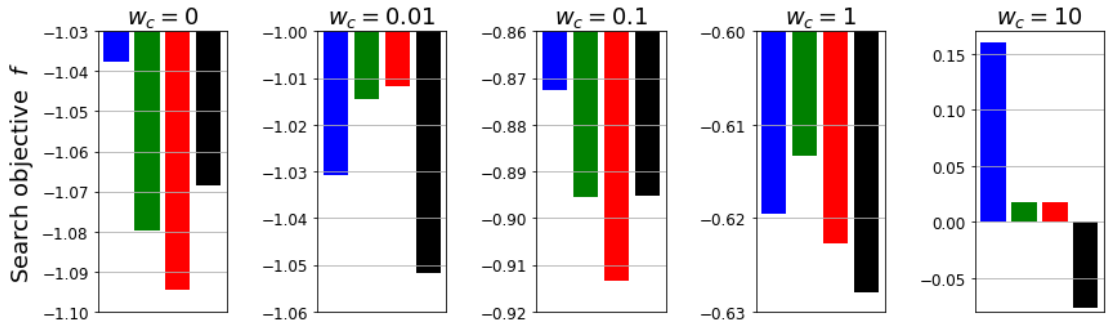


Figure 7.10: Search objective values (lower the better) for purely random search (30 samples, blue) vs purely grid search via Sobol sequencing (30 samples, green) vs balanced BO (15 initial samples, 15 optimized samples, red) vs extreme BO (1 initial sample, 29 optimized samples, black). Results shown for different w_c on CIFAR-10 unaugmented.

unaugmented (Fig. 7.9 shows three of these). In each case we found that following a purely greedy approach yielded best results, which justifies our choice for DnC.

7.6.3 Bayesian optimization vs random and grid search

We use Sobol sequencing [156] – a space-filling method that selects points similar to grid search – to select initial points from the search space and construct the BO

prior. We experimented on the usefulness of BO by comparing the final search loss f achieved by performing the Stage 1 and 3 searches in four different ways:

- Random search: pick 30 prior points randomly, no optimization steps
- Grid search: pick 30 prior points via Sobol sequencing, no optimization steps
- Balanced BO (DnC default): pick 15 prior points via Sobol sequencing, 15 optimization steps
- Extreme BO: pick 1 initial point, 29 optimization steps

The results in Fig. 7.10 are for different w_c on CIFAR-10. BO outperforms random and grid search on each occasion. In particular, more optimization steps are beneficial for low complexity models, while the advantages of BO are not significant for high performing models. We believe that this is due to the fact that many deep nets [153] are fairly robust to training hyperparameter settings.

7.6.4 Ensembling

One way to increase performance such as test accuracy is by having an ensemble of multiple networks vote on the test set. This comes at a complexity cost since multiple NNs need to be trained. We experimented on ensembling by taking the n best networks from BO in Stage 3 of our search. Note that this *does not increase the search cost* as long as $n \leq n_1 + n_2$. However, it does increase the effective number of parameters by a factor of exactly n (since each of the n best configurations have

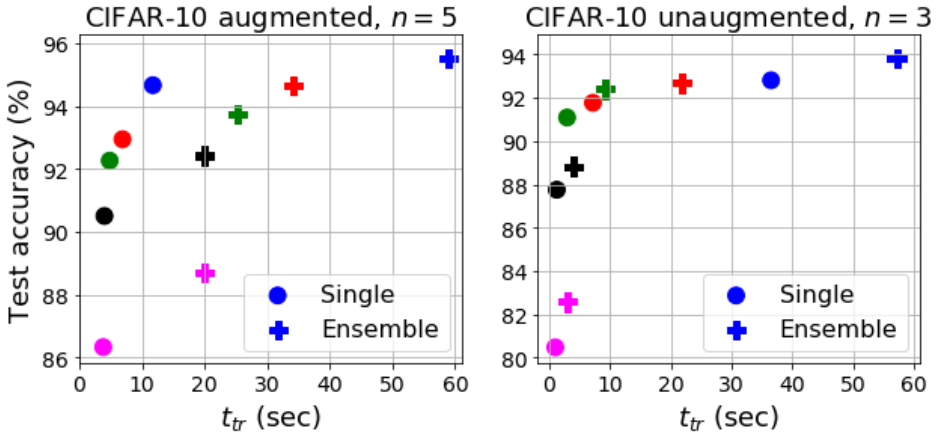


Figure 7.11: Performance-complexity tradeoff for single configurations (circles) vs ensemble of configurations (pluses) for $w_c = 0$ (blue), 0.01 (red), 0.1 (green), 1 (black), 10 (pink). Results using ensemble of 5 for CIFAR-10 augmented, and 3 for CIFAR-10 unaugmented.

the same architecture), and t_{tr} by some indeterminate factor (since each of the n best configurations might have a different batch size).

We experimented on CIFAR-10 unaugmented using $n = 3$ and augmented using $n = 5$. The impact on the performance-complexity tradeoff is shown in Fig. 7.11. Note how the plus markers – ensemble results – have slightly better performance at the cost of significantly increased complexity as compared to the circles – single results. However, we did not use ensembling in other experiments since the slight increases in accuracy do not usually justify the significant increases in t_{tr}.

7.6.5 Changing hyperparameters of Bayesian Optimization

The BO process itself has several hyperparameters that can be customized by the user, or optimized using marginal likelihood or Markov chain Monte Carlo methods

[139]. This section describes the default values we used. Expected improvement involves an exploration-exploitation tradeoff variable ξ. The recommended default is $\xi = 0.01$ [151], however, we tried different values and empirically found $\xi = 10^{-4}$ to work well. Secondly, f is a noisy function since the computed values of network performance are noisy due to random initialization of weights and biases for each new state. Accordingly, and also considering numerical stability for the matrix inversions involved in BO, our algorithm incorporates a noise term σ_n^2. We calculated its value from the variance in f values as $\sigma_n^2 = 10^{-4}$, which worked well compared to other values we tried.

7.6.6 Adaptation to various platforms

While most deep NNs are run on GPUs, situations may arise where GPUs are not readily or freely available and it is desirable to run simpler experiments such as MLP training on CPUs. DnC can adapt its penalty metrics to any platform. For example, the FMNIST results shown in Fig. 7.7 were on CPU, while Table 7.3 shows results on GPU (to do a fair comparison with other frameworks). As a result, the t_{tr} values are an order of magnitude faster, while the performance is the same as expected.

7.7 Summary

This chapter discussed Deep-n-Cheap – the first AutoML framework that specifically considers training complexity of the resulting models during searching. The wall clock training time per epoch that we use is a measure that automatically takes the computing platform and software libraries into account. While our framework can be customized to search over any number of layers, it is interesting that we obtained competitive performance on various datasets using models significantly less deep than those obtained from other AutoML and search frameworks in literature. We also found that it is possible to transfer a family of architectures found using different w_c values between different datasets without performance degradation. The framework uses Bayesian optimization and a three-stage greedy search process – these were empirically demonstrated to be superior to other search methods and less greedy approaches.

DnC currently supports classification using CNNs and MLPs. Our future plans are to extend to other types of networks such as recurrent and other applications of deep learning such as segmentation, which would also require expanding the set of hyperparameters searched over. The framework is open source and offers considerable customizability to the user. We hope that DnC becomes widely used and provides efficient NN design solutions to many users.

Chapter 8

Conclusion

This chapter concludes this dissertation. We provide a summary of the field of neural networks, our achieved contributions, and the work discussed in this document.

8.1 Summary

Despite the recent spurt of interest in NNs, they are not a completely new concept. The perceptron, which can be considered a very simple NN, was created in 1958 by Rosenblatt [157]. After a few decades of research stagnation, interest in NNs was again piqued in 1986 when Rumelhart et al. proposed backpropagation [158]. However, computers in the 1980s and 90s were not equipped to handle machine learning and NNs on a large scale as is done today. It was much more recently in the 2010s, perhaps with the publication of AlexNet [1], that the research interest and usage of NNs have skyrocketed. As a result, NNs have become the backbone of many AI applications nowadays, and are the major driving force behind the wide variety of 'smart' technologies that have become ubiquitous in our lives. This exponential growth in interest is summarized by Figs. 8.1 and 8.2, which show the

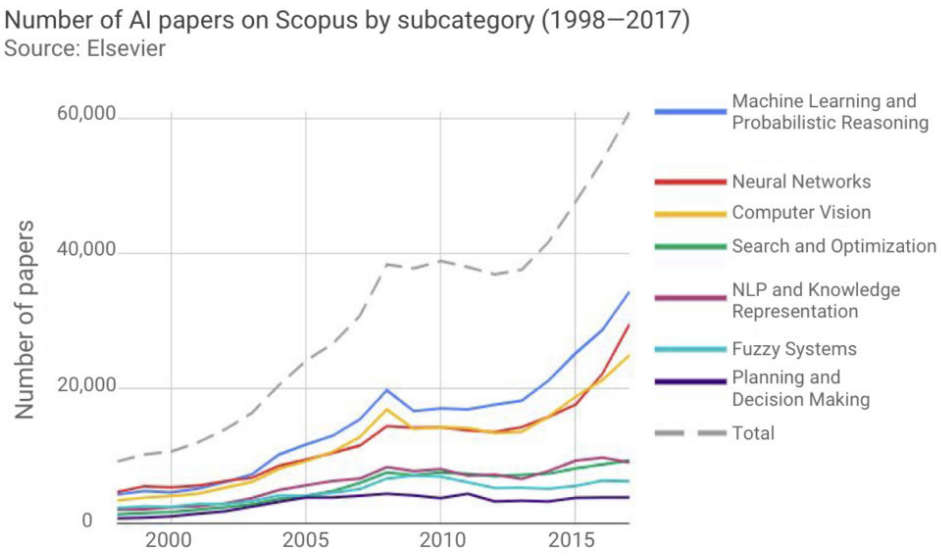


Figure 8.1: Number of AI papers on Scopus from 1998–2017, grouped by subcategory. Figure courtesy [159].

number of papers related to AI published in two widely used research databases – Elsevier's Scopus, and arXiv, respectively. Notice how NN papers on Scopus have tripled in the last decade, while net AI papers on arXiv have grown by more than 10X from 2010 to 2017.

The growth in interest in NNs has also translated to a growth in their complexity. In particular, there has been a tendency in industry players to drive the field of NNs forward by exponentially increasing the number of parameters, as shown in Fig. 8.3. While this parameter explosion has led to pushing the frontiers of what NNs can achieve, it has often sacrificed a deeper understanding of the field and practical considerations such as the carbon footprint [161] and economic

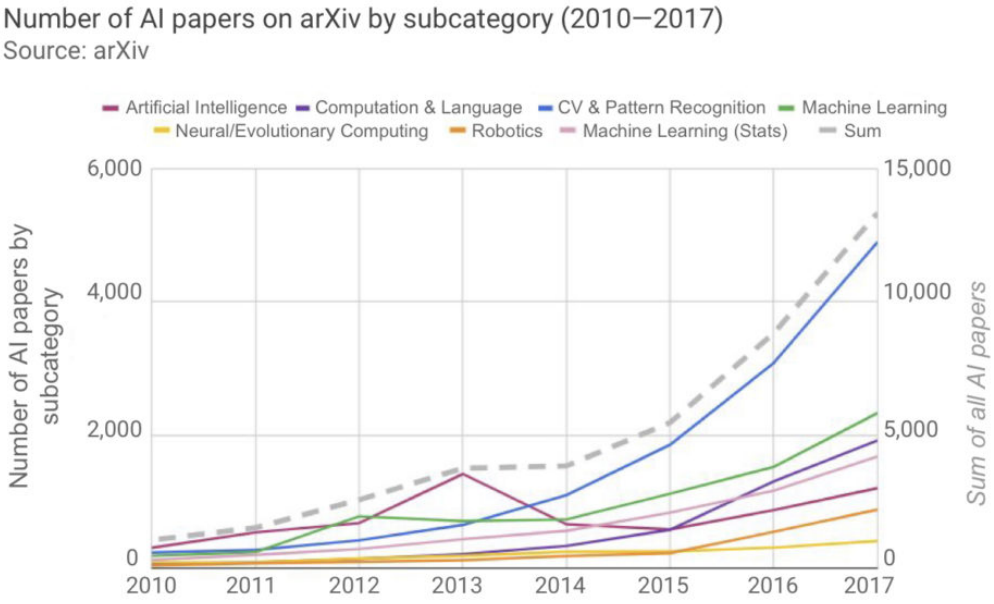


Figure 8.2: Number of AI papers on arXiv from 2010–2017, grouped by subcategory. Figure courtesy [159].

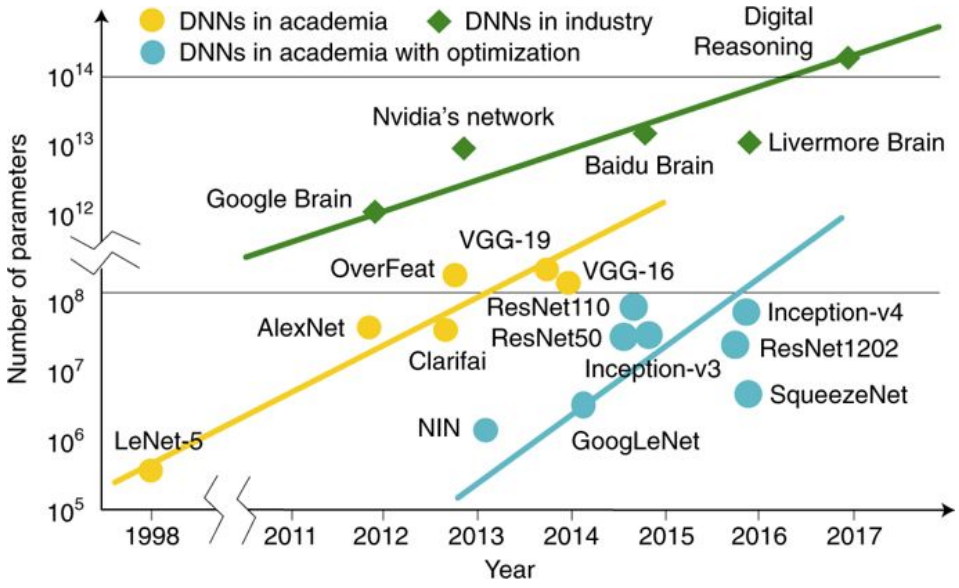


Figure 8.3: Total number of trainable parameters in some popular deep NN architectures over the years. Figure courtesy [160, Fig. 1].

footprint [9–11] of NNs. This status quo is concerning because it puts the reins of NN research into the sole hands of a few (often selfish) entities with the financial wherewithal to deploy GPUs and other computational resources at a scale well beyond the reach of most other entities. In other words, the NN parameter explosion has resulted in an oligarchy of sorts, where the barrier to entry in terms of required computational resources can push many prospective researchers away.

Our Contributions

The work presented in this dissertation attempts to democratize the development of NNs by lowering their complexity. In particular, our proposed method of pre-defined sparsity simplifies the computational burden of NNs from the beginning of their deployment, *i.e.* prior to training. This is crucial for implementing NNs on custom hardware, such as we have developed. We have also analyzed connection patterns with a view to identifying good NN configurations. This segued into our work in automated machine learning, where we have developed a user-friendly framework called Deep-n-Cheap to search for NNs with good performance and low complexity. Despite many existing works suggesting the contrary, we have achieved state-of-the-art performance with NNs that also train quickly. As an additional effort, we have also developed a family of datasets of varying difficulty for benchmarking machine learning algorithms and applications.

8.2 Final Word

The field of NNs is exciting, with new research and novel techniques being developed every day. We believe our research has the potential to enable NN exploration on a wider scale. While this dissertation encompasses a significant amount of work, it is by no means the end of the road. A better understanding of sparse connection patterns, improving the hardware architecture, and maintaining and extending Deep-n-Cheap are a few aspects that will be explored in the future. For more information on our research group, current projects and mission, the reader is encouraged to visit our website [81].

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Advances in Neural Information Processing Systems 25 (NeurIPS)*, 2012, pp. 1097–1105.

[2] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," *arXiv e-print arXiv:1604.07316*, 2016.

[3] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, Nov 2012.

[4] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv e-print arXiv:1207.0580*, 2012.

[5] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," *arXiv e-print arXiv:1602.07261*, 2016.

[6] A. Coates, B. Huval, T. Wang, D. J. Wu, A. Y. Ng, and B. Catanzaro, "Deep learning with COTS HPC systems," in *Proc. 30th Int. Conf. Machine Learning (ICML)*, vol. 28, 2013, pp. III–1337–III–1345.

[7] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural networks," in *Proc. Advances in Neural Information Processing Systems 28 (NeurIPS)*, 2015, pp. 1135–1143.

[8] N. P. Jouppi, C. Young, N. Patil *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *2017 ACM/IEEE 44th Annu. Int. Symp. Computer Architecture (ISCA)*, June 2017.

[9] G. Cloud, "GPUs pricing," https://cloud.google.com/compute/gpus-pricing, accessed on Apr 10th, 2020.

[10] A. AWS, "Amazon EMR pricing," https://aws.amazon.com/emr/pricing/, accessed on Apr 10th, 2020.

[11] M. Azure, "Cloud services pricing," https://azure.microsoft.com/en-us/pricing/details/cloud-services/, accessed on Apr 10th, 2020.

[12] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *IEEE Conf. Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1–9.

[13] Y. Gong, L. Liu, M. Yang, and L. D. Bourdev, "Compressing deep convolutional networks using vector quantization," *arXiv e-print arXiv:1412.6115*, 2014.

[14] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick," in *Proc. 32nd Int. Conf. Machine Learning (ICML)*, 2015.

[15] S. Han, H. Mao, and W. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2016.

[16] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," in *Proc. 43rd Int. Symp. Computer Architecture (ISCA)*, 2016.

[17] H. Jin, Q. Song, and X. Hu, "Auto-keras: An efficient neural architecture search system," in *Proc. KDD*, 2019, pp. 1946–1956.

[18] AWSLabs, "AutoGluon: AutoML toolkit for deep learning," https://autogluon.mxnet.io/#.

[19] H. Mendoza, A. Klein, M. Feurer, J. T. Springenberg, M. Urban, M. Burkart, M. Dippel, M. Lindauer, and F. Hutter, "Towards automatically-tuned deep neural networks," in *AutoML: Methods, Sytems, Challenges*. Springer, 2018, ch. 7, pp. 141–156.

[20] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, "Efficient neural architecture search via parameter sharing," *arXiv e-print arXiv:1802.03268*, 2018.

[21] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, and B. Hodjat, "Evolving deep neural networks," in *Artificial Intelligence in the Age of Neural Networks and Brain Computing*. Academic Press, 2019, ch. 15, pp. 293 – 312.

[22] K. Kandasamy, W. Neiswanger, J. Schneider, B. Póczos, and E. P. Xing, "Neural architecture search with bayesian optimisation and optimal transport," in *Proc. Advances in Neural Information Processing Systems 31 (NeurIPS)*, 2018, pp. 2020–2029.

[23] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: Bandit - based configuration evaluation for hyperparameter optimization," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2017.

[24] H. Cai, L. Zhu, and S. Han, "ProxylessNAS: Direct neural architecture search on target task and hardware," in *Proc. Int. Conf. Learn. Representations (ICLR)*, 2019.

[25] S. Dey, K.-W. Huang, P. A. Beerel, and K. M. Chugg, "Pre-defined sparse neural networks with hardware acceleration," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 332–345, June 2019.

[26] S. Dey, K.-W. Huang, P. A. Beerel, and K. M. Chugg, "Characterizing sparse connectivity patterns in neural networks," in *Proc. 2018 Information Theory and Applications Workshop (ITA)*, Feb 2018, pp. 1–9.

[27] S. Dey, Y. Shao, K. M. Chugg, and P. A. Beerel, "Accelerating training of deep neural networks via sparse edge processing," in *Proc. 26th Int. Conf. Artificial Neural Networks (ICANN)*. Springer, Sep 2017, pp. 273–280.

[28] S. Dey, D. Chen, Z. Li, S. Kundu, K.-W. Huang, K. M. Chugg, and P. A. Beerel, "A highly parallel FPGA implementation of sparse neural network training," in *Proc. Int. Conf. ReConFigurable Computing and FPGAs (ReConFig)*, Dec 2018, pp. 1–4, expanded e-print version available at https://arxiv.org/abs/1806.01087.

[29] S. Dey, P. A. Beerel, and K. M. Chugg, "Interleaver design for deep neural networks," in *Proc. 51st Asilomar Conf. Signals, Systems, and Computers (ACSSC)*, Oct 2017, pp. 1979–1983.

[30] S. Dey, "Github repository: predefinedsparse-nnets," https://github.com/souryadey/predefinedsparse-nnets.

[31] ——, "Github repository: deep-n-cheap," https://github.com/souryadey/deep-n-cheap.

[32] S. Dey, K. M. Chugg, and P. A. Beerel, "Morse code datasets for machine learning," in *Proc. 9th Int. Conf. Computing, Communication and Networking Technologies (ICCCNT)*, Jul 2018, pp. 1–7.

[33] S. Dey, "Github repository: morse-dataset," https://github.com/souryadey/morse-dataset.

[34] ——, "Ieeedataport: Morse code symbol classification," https://ieee-dataport.org/open-access/morse-code-symbol-classification.

[35] ——, "Matrix calculus," University of Southern California, Tech. Rep., 2019, available online at https://www.researchgate.net/publication/332131671_Matrix_Calculus.

[36] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proc. 27th Int. Conf. Machine Learning (ICML)*, 2010, pp. 807–814.

[37] S. Hochreiter, "Untersuchungen zu dynamischen neuronalen netzen," Master's thesis, TU Munich, 1991.

[38] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proc. 13th Int. Conf. Artificial Intelligence and Statistics*, 2010, pp. 249–256.

[39] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification," in *Proc. IEEE Int. Conf. Computer Vision (ICCV)*, 2015, pp. 1026–1034.

[40] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv e-print arXiv:1609.04747*, 2016.

[41] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2014.

[42] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[43] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. 32nd Int. Conf. Machine Learning (ICML)*, 2015.

[44] A. Deshpande, "A beginner's guide to understanding convolutional neural networks," https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/, Jul 2016.

[45] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[46] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, ch. 9, http://www.deeplearningbook.org.

[47] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015, ch. 6.

[48] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *arXiv e-print arXiv:1603.07285*, 2016.

[49] K. Bai, "A comprehensive introduction to different types of convolutions in deep learning," https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-669281e58215, Feb 2019.

[50] K. M. Chugg, "Convolutional neural networks," https://hal.usc.edu/chugg/docs/deep_learning/cnns.pdf, Apr 2020.

[51] W. Bao, J. Yue, and Y. Rao, "A deep learning framework for financial time series using stacked autoencoders and long-short term memory," *PLoS ONE*, vol. 12, Jul 2017.

[52] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. D. Freitas, "Predicting parameters in deep learning," in *Proc. Advances in Neural Information Processing Systems 26 (NeurIPS)*, 2013, pp. 2148–2156.

[53] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning requires rethinking generalization," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2017.

[54] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1," *arXiv e-print arXiv:1602.02830*, 2016.

[55] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Proc. 2016 ACM/IEEE 43rd Annu. Int. Symp. Computer Architecture (ISCA)*, 2016, pp. 1–13.

[56] B. Reagen, P. Whatmough, R. Adolf *et al.*, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proc. 2016 ACM/IEEE 43rd Annu. Int. Symp. Computer Architecture (ISCA)*, 2016, pp. 267–278.

[57] A. Aghasi, A. Abdi, N. Nguyen, and J. Romberg, "Net-trim: Convex pruning of deep neural networks with performance guarantee," in *Proc. Advances in Neural Information Processing Systems 30 (NeurIPS)*, 2017.

[58] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications," *arXiv e-print arXiv:1412.7024*, 2014.

[59] S. Gupta, A. Agarwal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," *arXiv e-print arXiv:1502.02551*, 2015.

[60] K. Kara, D. Alistarh, G. Alonso, O. Mutlu, and C. Zhang, "FPGA-accelerated dense linear machine learning: A precision-convergence trade-off," in *Proc. IEEE Int. Symp. Field-Programmable Custom Computing Machines*, 2017.

[61] A. Sanyal, P. A. Beerel, and K. M. Chugg, "Neural network training with approximate logarithmic computations," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2020, pp. 3122–3126.

[62] V. Sindhwani, T. Sainath, and S. Kumar, "Structured transforms for small-footprint deep learning," in *Proc. Advances in Neural Information Processing Systems 28 (NeurIPS)*, 2015, pp. 3088–3096.

[63] S. Wang, Z. Li, C. Ding, B. Yuan, Y. Wang, Q. Qiu, and Y. Liang, "C-LSTM: Enabling efficient LSTM using structured compression techniques on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2018.

[64] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Proc. Advances in Neural Information Processing Systems 29 (NeurIPS)*, 2016, pp. 2074–2082.

[65] S. Srinivas, A. Subramanya, and R. V. Babu, "Training sparse neural networks," in *IEEE Conf. Computer Vision and Pattern Recognition Workshops (CVPRW)*, July 2017, pp. 455–462.

[66] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. 19th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 269–284.

[67] S. Zhang, Z. Du, L. Zhang *et al.*, "Cambricon-X: An accelerator for sparse neural networks," in *Proc. 2016 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2016, pp. 1–12.

[68] Y. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[69] D. Shin, J. Lee, J. Lee, and H. Yoo, "14.2 DNPU: An 8.1TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks," in *IEEE Int. Solid-State Circuits Conf. (ISSCC)*, Feb 2017, pp. 240–241.

[70] Y. Ma, N. Suda, Y. Cao, S. Vrudhula, and J. Seo, "ALAMO: FPGA acceleration of deep learning algorithms with a modularized RTL compiler," *Integration, the VLSI Journal*, 2018.

[71] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," in *Proc. 2016 ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*. ACM, 2016, pp. 16–25.

[72] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. B. J. Dally, "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2017, pp. 75–84.

[73] R. G. Gironés, R. C. Palero, J. C. Boluda, and A. S. Cortés, "FPGA implementation of a pipelined on-line backpropagation," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 40, no. 2, pp. 189–213, Jun 2005.

[74] N. Izeboudjen, A. Farah, H. Bessalah, A. Bouridene, and N. Chikhi, "Towards a platform for FPGA implementation of the MLP based back propagation algorithm," in *Computational and Ambient Intelligence*, 2007, pp. 497–505.

[75] A. Gomperts, A. Ukil, and F. Zurfluh, "Development and implementation of parameterized FPGA-based general purpose neural networks for online applications," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 1, pp. 78–89, Feb 2011.

[76] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *47th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec 2014, pp. 609–622.

[77] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou, "DLAU: A scalable deep learning accelerator unit on FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 3, pp. 513–517, Mar 2017.

[78] A. Bourely, J. P. Boueri, and K. Choromonski, "Sparse neural network topologies," *arXiv e-print arXiv:1706.05683*, 2017.

[79] A. Prabhu, G. Varma, and A. M. Namboodiri, "Deep expander networks: Efficient deep networks from graph theory," *arXiv e-print arXiv:1711.08757*, 2017.

[80] D. C. Mocanu, E. Mocanu, P. Stone, P. H. Nguyen, M. Gibescu, and A. Liotta, "Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science," *Nature Communications*, vol. 9, 2018.

[81] USC HAL team, "Hardware accelerated learning," https://hal.usc.edu/.

[82] S. Kundu, S. Prakash, H. Akrami, P. B. Beerel, and K. M. Chugg, "PSConv: A pre-defined sparse kernel based convolution for deep CNNs," in *Allerton Conference on Communication, Control, and Computing*, Sep. 2019.

[83] S. Kundu, M. Nazemi, M. Pedram, K. M. Chugg, and B. Peter, "Pre-defined sparsity for low-complexity convolutional neural networks," *IEEE Transactions on Computers*, pp. 1–1, 2020, early Access.

[84] J. Yosinski and H. Lipson, "Visually debugging restricted boltzmann machine training with a 3D example," in *Proc. 29th Int. Conf. Machine Learning (ICML)*, 2012.

[85] Y. LeCun, C. Cortes, and C. J. Burges, "The MNIST database of handwritten digits," http://yann.lecun.com/exdb/mnist/.

[86] D. D. Lewis, Y. Yang, T. G. Rose, and F. Li, "RCV1: A new benchmark collection for text categorization research," *Journal of machine learning research*, vol. 5, pp. 361–397, Apr 2004.

[87] J. S. Garofolo, L. F. Lamel, W. M. Fisher, J. G. Fiscus, D. S. Pallett, N. L. Dahlgren, and V. Zue, "TIMIT acoustic-phonetic continuous speech corpus," https://catalog.ldc.upenn.edu/LDC93S1.

[88] K.-F. Lee and H.-W. Hon, "Speaker-independent phone recognition using hidden markov models," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, no. 11, pp. 1641–1648, Nov 1989.

[89] S. S. Stevens, J. Volkmann, and E. B. Newman, "A scale for the measurement of the psychological magnitude pitch," *The Journal of the Acoustical Society of America*, vol. 8, no. 185, 1937.

[90] A. Krizhevsky, "Learning multiple layers of features from tiny images," Master's thesis, University of Toronto, 2009.

[91] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[92] M. Abadi, A. Agarwal, P. Barham *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," https://www.tensorflow.org/, 2015, software available from tensorflow.org.

[93] N. H. E. Weste and D. M. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed. Pearson, 2010.

[94] P. Goyal, P. Dollar, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch SGD: training ImageNet in 1 hour," *arXiv e-print arXiv:1706.02677*, 2017.

[95] D. Masters and C. Luschi, "Revisiting Small Batch Training for Deep Neural Networks," *arXiv e-print arXiv:1804.07612*, 2018.

[96] Digilent, "Nexys 4 DDR," https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/start.

[97] Xilinx, "VIRTEX ultra-scale+," https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html.

[98] Amazon Web Services, "Amazon EC2 F1 instances," https://aws.amazon.com/ec2/instance-types/f1/.

[99] A. S. Asratian, T. M. J. Denley, and R. Haggkvist, *Bipartite Graphs and Their Applications*. Cambridge University Press, 1998.

[100] A. Barvinok, "On the number of matrices and a random matrix with prescribed row and column sums and 0–1 entries," *Advances in Mathematics*, vol. 224, no. 1, pp. 316–339, 2010.

[101] G. Masera, G. Piccinini, M. R. Roch, and M. Zamboni, "VLSI architectures for turbo codes," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 3, pp. 369–379, 1999.

[102] T. Brack, M. Alles, T. Lehnigk-Emden *et al.*, "Low complexity LDPC code decoders for next generation standards," in *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, 2007, pp. 1–6.

[103] S. Crozier and P. Guinand, "High-performance low-memory interleaver banks for turbo-codes," in *Vehicular Technology Conf., 2001. VTC 2001 Fall. IEEE VTS 54th*, vol. 4, 2001, pp. 2394–2398.

[104] G. M. Weiss and F. Provost, "Learning when training data are costly: The effect of class distribution on tree induction," *Journal of Artificial Intelligence Research*, vol. 19, pp. 315–354, 2003.

[105] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[106] I. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2015.

[107] A. Schumacher, "Problems with imagenet and its solutions," https://planspace.org/20170911-problems_with_imagenet_and_its_solutions/, September 2017.

[108] J. Prendki, "The curse of big data labeling and three ways to solve it," https://aws.amazon.com/blogs/apn/the-curse-of-big-data-labeling-and-three-ways-to-solve-it/, November 2018.

[109] M. Gregory and K. Ocasio, "The [hidden] challenges of ML series: Quadrant 2 data preparation," https://www.ntconcepts.com/the-hidden-challenges-of-ml-series-quadrant-2-data-preparation/, February 2019.

[110] X. Peng, B. Sun, K. Ali, and K. Saenko, "Learning deep object detectors from 3D models," in *Proc. IEEE Int. Conf. Computer Vision (ICCV)*. IEEE Computer Society, 2015, pp. 1278–1286.

[111] D. DeTone, T. Malisiewicz, and A. Rabinovich, "Toward geometric deep SLAM," *arXiv e-print arXiv:1707.07410*, 2017.

[112] T. Anh Le, A. G. Baydin, R. Zinkov, and F. Wood, "Using synthetic data to train neural networks is model-based reasoning," *arXiv e-print arXiv:1703.00868*, 2017.

[113] N. Patki, R. Wedge, and K. Veeramachaneni, "The synthetic data vault," in *Proc. IEEE Int. Conf. Data Science and Advanced Analytics (DSAA)*, 2016, pp. 399–410.

[114] C.-H. Luo and C.-H. Shih, "Adaptive morse-coded single-switch communication system for the disabled," *Int. Journal of Bio-Medical Computing*, vol. 41, no. 2, pp. 99–106, 1996.

[115] C.-H. Yang, C.-H. Yang, L.-Y. Chuang, and T.-K. Truong, "The application of the neural network on morse code recognition for users with physical impairments," *Proc. Institution of Mechanical Engineers, Part H: Journal of Engineering in Medicine*, vol. 215, no. 3, pp. 325–331, 2001.

[116] C.-H. Yang, L.-Y. Chuang, C.-H. Yang, and C.-H. Luo, "Morse code application for wireless environmental control systems for severely disabled individuals," *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 11, no. 4, pp. 463–469, Dec 2003.

[117] C. P. Ravikumar and M. Dathi, "A fuzzy-logic based morse code entry system with a touch-pad interface for physically disabled persons," in *Proc. IEEE Annu. India Conf. (INDICON)*, Dec 2016.

[118] T. W. King, *Modern Morse Code in Rehabilitation and Education: New Applications in Assistive Technology*, 1st ed. Allyn and Bacon, 1999.

[119] R. Sheinker, "Morse code - apps on google play," https://play.google.com/store/apps/details?id=com.dev.morsecode&hl=en, Aug 2017.

[120] F. Bonnin, "Morse-it on the app store," https://itunes.apple.com/us/app/morse-it/id284942940?mt=8, Mar 2018.

[121] D. Hill, "Temporally processing neural networks for morse code recognition," in *Theory and Applications of Neural Networks*. Springer London, 1992, pp. 180–197.

[122] G. N. Aly and A. M. Sameh, "Evolution of recurrent cascade correlation networks with distributed collaborative species," in *Proc. 1st IEEE Symp. Combinations of Evolutionary Computation and Neural Networks*, 2000, pp. 240–249.

[123] R. Li, M. Nguyen, and W. Q. Yan, "Morse codes enter using finger gesture recognition," in *Proc. Int. Conf. Digital Image Computing: Techniques and Applications (DICTA)*, Nov 2017.

[124] *International Morse Code*, Radiocommunication Sector of International Telecommunication Union, Oct 2009, available at http://www.itu.int/rec/R-REC-M.1677-1-200910-I/.

[125] R. M. Zur, Y. Jiang, L. L. Pesce, and K. Drukker, "Noise injection for training artificial neural networks: A comparison with weight decay and early stopping," *Medical Physics*, vol. 36, no. 10, pp. 4810–4818, Oct 2009.

[126] K. Chugg, A. Anastasopoulos, and X. Chen, *Iterative Detection: Adaptivity, Complexity Reduction, and Applications*. Springer Science & Business Media, 2012, vol. 602.

[127] H. Liu, K. Simonyan, and Y. Yang, "DARTS: Differentiable architecture search," *Proc. Int. Conf. Learning Representations (ICLR)*, 2019.

[128] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," in *Proc. European Conf. Comp. Vision (ECCV)*, 2018, pp. 19–35.

[129] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2017.

[130] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *Proc. AAAI*, 2019, pp. 4780–4789.

[131] L. Xie and A. Yuille, "Genetic cnn," in *Proc. Int. Conf. Comp. Vision (ICCV)*, 2017, pp. 1388–1397.

[132] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *arXiv e-print arXiv:1905.11946*, 2019.

[133] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "AMC: AutoML for model compression and acceleration on mobile devices," in *Proc. European Conf. Comp. Vision (ECCV)*, 2018, pp. 784–800.

[134] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Proc. Advances in Neural Information Processing Systems 25 (NeurIPS)*, 2012, pp. 2951–2959.

[135] J. Bergstra, D. Yamins, and D. D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *Proc. 30th Int. Conf. Machine Learning (ICML)*, 2013.

[136] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms," in *Proc. KDD*, 2013, pp. 847–855.

[137] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *Proc. IEEE Conf. Comp. Vision and Pattern Recognition, CVPR*, 2018, pp. 8697–8710.

[138] J. Snoek, O. Rippel, K. Swersky, R. Kiros, N. Satish, N. Sundaram, M. M. A. Patwary, Prabhat, and R. P. Adams, "Scalable bayesian optimization using deep neural networks," *arXiv e-print arXiv:1502.05700*, 2015.

[139] K. Swersky, D. Duvenaud, J. Snoek, F. Hutter, and M. Osborne, "Raiders of the lost architecture: Kernels for bayesian optimization in conditional parameter spaces," in *NeurIPS workshop on Bayesian Optimization in Theory and Practice*, 2013.

[140] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kegl, "Algorithms for hyper-parameter optimization," in *Proc. Advances in Neural Information Processing Systems 24 (NeurIPS)*, 2011, pp. 2546–2554.

[141] A. Brock, T. Lim, J. Ritchie, and N. Weston, "SMASH: One-shot model architecture search through hypernetworks," *arXiv e-print arXiv 1708.05344*, 2017.

[142] R. C. Mayo, D. Kent, and et al., "Reduction of false-positive markings on mammograms: a retrospective comparison study using an artificial intelligence-based CAD," *J. Digital Imaging*, vol. 32, pp. 618–624, 2019.

[143] P. Baldi, P. Sadowski, and D. Whiteson, "Searching for exotic particles in high-energy physics with deep learning," *Nature Communications*, vol. 5, p. 4308, 2014.

[144] E. Santana and G. Hotz, "Learning a driving simulator," *arXiv e-print arXiv:1608.01230*, 2016.

[145] A. Paszke, S. Gross, F. Massa *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, 2019, pp. 8024–8035.

[146] S. Dey, "Sparse matrices in pytorch, part 1: Cpu runtimes," https://towardsdatascience.com/sparse-matrices-in-pytorch-be8ecaccae6.

[147] ——, "Sparse matrices in pytorch, part 2: Gpu runtimes," https://towardsdatascience.com/sparse-matrices-in-pytorch-part-2-gpus-fd9cc0725b71.

[148] G. Huang, Y. Sun, and et al., "Deep networks with stochastic depth," in *Proc. European Conf. Comp. Vision (ECCV)*, 2016, pp. 646–661.

[149] D. Page, "How to train your resnet," https://myrtle.ai/how-to-train-your-resnet/.

[150] B. Baker, O. Gupta, R. Raskar, and N. Naik, "Accelerating neural architecture search using performance prediction," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2018.

[151] E. Brochu, V. M. Cora, and N. de Freitas, "A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning," *arXiv e-print arXiv:1012.2599*, 2010.

[152] F. Hutter and M. A. Osborne, "A kernel for hierarchical parameter spaces," *arXiv e-print arXiv:1310.5738*, 2013.

[153] S. Zagoruyko and N. Komodakis, "Wide residual networks," in *Proc. British Machine Vision Conference (BMVC)*, 2016, pp. 87.1–87.12.

[154] "Private communication with authors regarding proxylessNAS," Mar 2020.

[155] H. Jin, "Comment on 'not able to load best automodel after saving' issue," https://github.com/keras-team/autokeras/issues/966#issuecomment-594590617.

[156] I. M. Sobol, "Distribution of points in a cube and approximate evaluation of integrals," *USSR Computational Mathematics and Mathematical Physics, 1967, 7:4, 86–112*, vol. 7, no. 4, pp. 86–112, 1967, originally in Russian: Zh. Vych. Mat. Mat.

[157] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, pp. 65–386, 1958.

[158] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[159] D. Enskat, "AI index 2018: Europe – china – united states; AI outpaces CS," https://warrenenskat.com/ai-index-2018-europe-china-united-states-ai-outpaces-cs/, Dec 2018.

[160] X. Xu, Y. Ding, S. X. Hu, M. Niemier, J. Cong, Y. Hu, and Y. Shi, "Scaling for edge inference of deep neural networks," *Nature Electronics*, vol. 1, no. 4, pp. 216–222, 2018.

[161] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for deep learning in nlp," *arXiv e-print arXiv:1906.02243*, 2019.