COVOLUTIONAL NEURAL NETWORKS BASED ON NON-EUCLIDEAN OPERATORS

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

MASTER OF SCIENCE

IN

ELECTRICAL AND ELECTRONICS ENGINEERING

By

Diaa Hisham Jamil Badawi

January 2018

Covolutional Neural Networks based on non-Euclidean Operators
By Diaa Hisham Jamil Badawi
January 2018

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Ahmet Enis Çetin(Advisor)

Ramazan Gökberk Çinbiş

Tolga Çukur

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

COVOLUTIONAL NEURAL NETWORKS BASED ON NON-EUCLIDEAN OPERATORS

Diaa Hisham Jamil Badawi
M.S. in Electrical and Electronics Engineering
Advisor: Ahmet Enis Çetin
January 2018

Dot product-based operations in neural net feedforwarding passes are replaced with an ℓ_1-norm inducing operator, which itself is multiplication-free. The neural net, which is called AddNet, retains attributes of ℓ_1-norm based feature extraction schemes such as resilience against outliers. Furthermore, feedforwarding passes can be realized using fewer multiplication operations, which implies energy efficiency. The ℓ_1-norm inducing operator is differentiable w.r.t its operands almost everywhere. Therefore, it is possible to use it in neural nets that are to be trained through standard backpropagation algorithm. AddNet requires scaling (multiplicative) bias so that cost gradients do not explode during training. We present different choices for multiplicative bias: trainable, directly dependent upon the associated weights, or fixed. We also present a sparse variant of that operator, where partial or full binarization of weights is achievable.
We ran our experiments over MNIST and CIFAR-10 datasets. AddNet could achieve results that are 0.1% less accurate than a ordinary CNN. Furthermore, trainable multiplicative bias helps the network to converge fast. In comparison with other binary-weights neural nets, AddNet achieves better results even with full or almost full weight magnitude pruning while keeping the sign information after training. As for experimenting on CIFAR-10, AddNet achieves accuracy 5% less than a ordinary CNN. Nevertheless, AddNet is more rigorous against impulsive noise data corruption and it outperforms the corresponding ordinary CNN in the presence of impulsive noise, even at small levels of noise.

Keywords: deep learning, convolutional neural network, l1 norm, energy efficiency, binary weights, impulsive noise.

ÖZET

ÖKLİDCE MENSUP OLMAYAN OPERATÖRLER BAZNDA KONVOLÜSYONEL SİNİR AĞILARI

Diaa Hisham Jamil Badawi
Elektrik ve Elekronik Mühendisliği, Yüksek Lisans
Tez Danışmanı: Ahmet Enis Çetin
Ocak 2018

Sinir ağı kapsamında, besleme-iletme pasosu geçişindeki nokta bazlı işlemler, çarpma işlemi gerektirmeyen bir ℓ_1-norm indükleyici operatör ile değiştirildi. AddNet denilen sinir ağı, aykırı değerlere karşı dayanıklılık gibi ℓ_1-norma dayalı öznitelik çıkarma şemalarını özümsemektedir. Ayrıca, besleme-iletme pasoları daha az çarpma işlemleri kullanarak gerçekleştirilebilir, bu da enerji verimliliğini ima eder. ℓ_1-norm indükleyici operatör, neredeyse her yerde işlenenlerine göre türevlenebilir. Bu nedenle, standart Backpropagation algoritması ile eğitilecek olan sinir ağlarında kullanması mümkündür. AddNet, zarar gradyanlarının eğitim sırasında patlamaması için ölçekleme (çarpımsal) bir yan gerektirir. Çarpımsal yanı için farklı seçenekler sunuyoruz: eğitilebilir, doğrudan ilişkili ağırlıklara bağlı, veya sabit. Ayrıca, o operatörün seyrek bir varyantını sunuyoruz sunuyoruz ve böylelikle, kısmi veya tam benirizasyona ulaşabiliyoruz. Denemelerimizi MNIST ve CIFAR-10 veri setleri üzerinden yürüttük. AddNet, ortalama bir CNN'den 0.1% daha az doğru sonuç elde edebilir. Ayrıca, eğitilebilir çarpımsal yanı, ağın hızla yakınsamasına yardımcı olur. Yükleri ikili olan diğer sinir ağlarıyla karşılaştırıldığında, AddNet daha iyi sonuçlar elde eder; eğitildikten sonra, işaret bilgilerini tutarken tam veya neredeyse tam ağırlığı büyüklüğünde budama yaparken bile. CIFAR-10 üzerinde deneylere gelince, AddNet ortalama bir CNN'den 5% daha az doğruluğa ulaşıyor. Yine de AddNet, verilerinin dürtüsel gürültü nedeniyle bozulmasına karşı daha titizdir ve dürtüsel gürültünün bulunduğu yerde ortalama bir CNN'den daha iyi performans gösterir, küçük gürültü seviyelerinde olsa bile.

Anahtar sözcükler: derin öğrenme, konvolüsyonel sinir ağı, l1-norm, enerji verimliliği, ikili ağırlıklar, dürtüsel gürültü.

Acknowledgement

First and foremost, I would like to thank my supervisor Prof. A. Enis Çetin for his wise guidance, patience and his suggestions regarding this work. It has been an honour for me to work with Prof. A. Enis Çetin and I look forward to working with him in the future.
I would like also to thank the jury members: Asst. Prof. Ramazan Gökberk Çinbiş and Asst. Prof. Tolga Çukur for their invaluable comments and suggestions.
I would like to thank Prof. Fatoş Yarman-Vural and her research group at METU University for our earlier fruitful discussions.
I would like to thank my friends Ma'en Mallah and Abdullah Al-Kilani for their help in proofreading and translating this work.
Finally, I would like to thank my family for their love and support.

To the memory of my mother ...

Contents

1 Introduction **1**

1.1 Overview 1

1.2 Organization of This Thesis 2

2 Background **4**

2.1 Introduction 4

2.2 Multilayer Perceptron 5

2.2.1 Feedforward and Backpropagation Equations 7

2.3 Convolutional Neural Networks 13

2.3.1 feedforward and backpropagation in ConvNets 17

2.3.2 Historical Background and Advancement 19

2.4 Benchmark Datasets 22

2.4.1 MNIST Dataset 22

2.4.2 CIFAR Dataset 22

3 Related Work **24**

4 Non-Euclidean Operators and Neural Nets **27**

4.1 Overview 27

4.2 ℓ_1-norm Inducing Operators 28

4.2.1 Properties of Operator $\oplus$ and $\oplus_s$ 30

4.2.2 Operator $\oplus$ and Energy Efficiency 31

4.2.3 Operator $\oplus$ and Noise 32

4.3 AddNet: Neural Network based on Operator $\oplus$ 33

4.3.1 AddNet: Feedforwarding Pass 34

4.3.2 Importance of Multiplicative Bias 35

4.3.3 Backpropagation Pass 38

4.3.4 Choices for Multiplicative Bias 41

4.3.5 Sparse Operator $\oplus$ 43

5 Experimental Results and Discussion **45**

5.1 Experiment 1: AddNet over MNIST Dataset 46

5.1.1 Case 0: Ordinary ConvNet 47

5.1.2 Case 1: AddNet with Constant Multiplicative Bias 49

5.1.3 Case 2: AddNet with Normalized ℓ_1 Norm Multiplicative Bias 49

5.1.4 Case 3: AddNet with Standard Deviation Based Multiplicative Bias 51
5.1.5 Case 4: AddNet with Trainable Multiplicative Bias 52
5.1.6 Case 5: Binarized Weights Network 52
5.1.7 Performance with Salt-and-Pepper Noise 55
5.2 Experiment 2: AddNet over CIFAR-10 Dataset 57
5.2.1 Case 0: Ordinary ConvNet 58
5.2.2 Case 1: AddNet with Constant Multiplicative Bias 59
5.2.3 Case 2: AddNet with Normalized ℓ_1 Norm Multiplicative Bias 59
5.2.4 Case 3: AddNet with Standard Deviation Based Multiplicative Bias 60
5.2.5 Case 4: AddNet with Trainable Multiplicative Bias 61
5.2.6 Performance with Salt-and-Pepper Noise 64

6 Conclusion 65

A MNIST Samples with Salt-and-Pepper Noise 76

B CIFAR-10 Samples with Salt-and-Pepper Noise 78

List of Figures

2.1 visualization of a perceptron with 3-point input 6

2.2 visualization of an MLP with one hidden layer, blue nodes correspond to neurons(perceptrons) and white nodes correspond to bias nodes 6

2.3 Sigmoidal activation (left) and hyperbolic tangent activation (right) and their derivatives. Blue lines correspond to the functions themselves and red lines to their derivatives. Dashed lines correspond to the x- and y-axes. 8

2.4 A typical ConvNet, inspired by [1] 14

2.5 Visualisation of ReLU and its variants, black lines corresponds to ReLU, green line and red line correspond to LeakyReLU with leakage factor of 5 and 3, respectively 16

2.6 Example samples from MNIST dataset (not shown to scale) . . . 23

2.7 Example samples from CIFAR-10 dataset (not shown to scale) . . 23

4.1 Visualization of Operator $\oplus$ based neuron, where $\sum$ is the accumulation of the "weighted" input, a is the scaling factor and $f(.)$ is a non-linear activation 33

5.1 MNIST experiment: cost convergence for cases with constant multiplicative bias. Refer to Table 5.5 with the corresponding symbol for full description of the case. 50

5.2 MNIST experiment: performance-memory score of AddNets w.r.t BWN at different sparsity levels. Cases 1-4 correspond to: constant, ℓ_1-norm based, standard deviation and trainable multiplicative bias. The real-valued (unsuppressed) weights are 32-bit long. 54

5.3 MNIST experiment: performance-memory score of AddNets w.r.t BWN at different sparsity levels. Cases 1-4 correspond to: constant, ℓ_1-norm based, standard deviation and trainable multiplicative bias. The real-valued (unsuppressed) weights are 16-bit long. 54

5.4 CIFAR Experiment: ConvNet loss convergence (solid line) and testing accuracy (dashed line) w.r.t batches 59

5.5 CIFAR experiment: AddNet loss convergence (solid line) and testing accuracy (dashed line) w.r.t batches (ℓ_1-norm based multiplicative bias) 60

5.6 CIFAR experiment: AddNet loss convergence (solid line) and testing accuracy (dashed line) w.r.t batches (standard deviation based bias) 61

5.7 CIFAR experiment: comparison between ℓ_1 case convergence ($\square$) and standard deviation case convergence ($*$). The two vertical lines correspond to the points at which the highest accuracy occurred for both cases 62

5.8 Loss convergence (solid line) and testing accuracy (dashed line) w.r.t batches 62

5.9 CIFAR experiment: comparison between trainable $\mathbf{a}^l$ case convergence ($*$) and ℓ_1 norm based $\mathbf{a}^l$ case convergence ($\square$). The two vertical lines correspond to the points at which the highest accuracy occurred for both cases 63

5.10 CIFAR experiment: comparison between the convergence of ConvNet ($\square$) and AddNet ($*$) case with ℓ_1 based multiplicative bias. The vertical lines correspond to the points in training at which the highest accuracy occurs for both cases. 63

List of Tables

4.1 Additive noise impact on Operator $\oplus_s$ 33

5.1 MNIST experiment: neural net architecture 46

5.2 MNIST experiment: number of elements-wise multiplication operations 47

5.3 MNIST experiment: ConvNet classification error for the normal case against different normalized levels of sparsity. Classification error is in percent. Suppressed weights correspond to the percentage of the weights below the sparsity level. 48

5.4 MNIST experiment: ConvNet classification error in percent with different sparsity choices 48

5.5 MNIST experiment: AddNet classification error in percent for different choices for multiplicative bias a^l 49

5.6 MNIST experiment: classification accuracy results for hybrid AddNet with constant multiplicative bias (case $*$ in Table 5.5) w.r.t different sparsity levels. Affected weights are those whose magnitudes are suppressed and only their signs are kept. 50

5.7 MNIST experiment: AddNet classification error in percent with ℓ_1 norm-based multiplicative bias with different activation choices. . 51

5.8 MNIST experiment: AddNet classification error in percent with ℓ1-based multiplicative bias for different sparsity levels 51

5.9 MNIST experiment: AddNet classification error in percent with standard deviation-based multiplicative bias for different sparsity levels 52

5.10 MNIST Experiment: AddNet classification error in percent with a trainable multiplicative bias for different sparsity levels 52

5.11 MNIST Experiment: comparison of classification error between BWN and hybrid AddNet with different activation choices 53

5.12 visualization of example MNIST images with different SAP levels 56

5.13 MNIST Experiment: classification error in percent over SAP-corrupted MNIST dataset at different levels 56

5.14 CIFAR Experiment: neural net architecture 57

5.15 CIFAR Experiment: number of elements-wise multiplication operations 58

5.16 CIFAR experiment: test classification error in percent for three CIFAR-10 models with SAP-corrupted testing data over different levels 64

Chapter 1

Introduction

1.1 Overview

Artificial Neural Networks have become more popular recently owing to their high success in fields such as computer vision [2, 3, 4, 5, 6, 7], speech recognition [8, 9, 10] and natural language processing [11, 12, 13].
Despite their success, neural networks are considered computationally intensive and conventional architectures developed are not suitable to perform recognition task with limited processing and energy. In order to address this problem, there have been many attempts to come up with lightweight energy efficient neural networks either by using quantization and approximation techniques [14], dedicated hardware [15] or novel mathematical models [16, 17, 18].
The ℓ_1 norm has been used in parameters estimation in order to realise sparse solutions [19, 20] as a replacement for classical ℓ_2 based solutions, ℓ_1. Furthermore, ℓ_1-norm feature extraction schemes are more resilient against outliers [21, 22, 23, 24] than ℓ_2 based schemes. This has motivated the development of ℓ_1-inducing operators, such as in [25, 26] as a replacement to conventional dot-product-based approaches.
In this work, we introduce AddNet: a convolutional neural network in which dot-product based operations such as: matrix multiplication and tensor convolution

are replaced with the ℓ_1-norm inducing operator suggested in [25]. The operator is referred to as operator $\oplus$ (reads "oplus"). This vector induces the a scaled ℓ_1 norm by a factor of two.
The importance of this work is two-fold: Firstly, since AddNet is based on an ℓ_1 inducing operator, it is expected to possess properties of other ℓ_1 feature extraction schemes, such as: resilience against outliers and impulsive noise [27]. Secondly, since the above-mentioned operator is multiplier-less, feedforwarding passes in AddNets involve carrying out fewer multiplication operations and instead perform non-linear addition with sign compensation. This is of great importance when it comes to reducing energy needed in feedforwarding pass.
Additionally, we show that AddNet is trainable through standard backpropagation: a big advantage when it comes to using high-level deep learning libraries such as tensorflow [28], Theano [29] and Caffe [30]. However, we need to apply multiplicative bias in order to control gradient through backpropagation, this multiplicative bias is inexpensive compared to dot-product based operations in conventional neural networks.
Furthermore, we show that the weights in AddNets can be fully or partially binarized without much sacrifice of performance. This means that AddNet can realize binary-weighted neural networks such as in [16, 17, 18], while retaining flexibility between network size on hardware and performance.

1.2 Organization of This Thesis

The structure of this thesis goes as follows: Chapter 2 contains a comprehensive background about neural networks especially convolutional neural nets. The background covers basic concepts, mathematical formulation of feedforwarding and backpropagation passes and a brief history about ConvNets.
Chapter 3 is a survey of the recent techniques and architectures that aim to yield more energy efficient neural nets.
Chapter 4 is the core chapter in which we discuss the ℓ_1-norm based Operator $\oplus$ in details. We also mathematically express AddNet. We also mathematically show that multiplicative bias is needed in order to be able to train AddNet. In this

regard, we discuss the different choices regrading multiplicative bias in details. Furthermore, we discuss the feasibility of making a sparse variant of operator $\oplus$ for further increase in efficiency.

Chapter 5 presents our experimental results and provide discussion on them. We provide a cross comparison study between the different choices possible when considering AddNets, such as: the scope of applying operator $\oplus$ in the network, the type of activation, the choice for multiplicative bias. We also compare AddNet with Binarized Weight Network (BWN), which is introduced in [16]. We also show the performance with full and partial weight binarization in AddNet. In addition to that, we compare the performance of AddNet and ordinary convNets when the data is corrupted by salt-and-pepper noise.

Finally, in chapter 6 we state our conclusions regarding this work.

Chapter 2

Background

2.1 Introduction

Artificial neural networks (or simply neural nets) are a class of machine learning algorithms that are loosely inspired by biological neural networks, where the building block, the neuron, can be seen as a mathematical abstraction of the biological neuron, where it ”fires” activation according to the input signal [31]. The ultimate objective of neural nets in the broadest sense is to be able to perform meaningful input-output mapping [32].

There have been many models of neural nets, among which is the multilayer perceptron, which has been very successful in supervised learning. In this chapter, we provide a brief background about multilayer perceptron (MLP) and convolutional neural nets. In Sec 2.2, we provide a mathematical formulation of MLP. Furthermore, we briefly explain basic concepts regarding: supervised learning, classification task, data separability and gradient based learning. In Sec 2.3, we explain basic concepts about convolutional neural net: an important subclass of MLPs, which is of the main interest in this work, since our experimentation was to study non-euclidean operator -based convolutional neural nets. Furthermore, a brief historical background on convoutional neural nets development is provided. In both sections, we formulate feedforward and backpropagation passes

equations, which are compared later with our non-euclidean operator-based neural networks. The **mathematical notations** employed in this chapter are used in later chapters in this thesis.

2.2 Multilayer Perceptron

Multiplayer perceptrons are a class of feedforward neural networks whose building block is the perceptron and which has at least three layers. A perceptron is a mathematical modelling of biological neuron, in that it has connections with input units, input gate where input is accumulated based on the "strength" of these connections, and an output gate, which fires a response based on the strength of the accumulated signal. In this regard, the neuron behaviour can be mathematically understood as posing a boundary hyperplane in the data space, and decide on the response based on the location of the data point w.r.t the boundary hyperplane. In other words, a neuron can separate data points linearly. This can be expressed mathematically as follows:

$$f(\mathbf{w}^T\mathbf{x}+b) = \begin{cases} 0 & \mathbf{w}^T\mathbf{x} < b \\ 1 & \mathbf{w}^T\mathbf{x} \geq b \end{cases} \tag{2.1}$$

where $\mathbf{x}$ is the input vector, $\mathbf{w}$ is the weight strength vector and b is a bias (threshold) term. $f(.) : \mathbb{R}^N \rightarrow [0,1]$ is the activation function, where N is the dimensionality of input data.

The question as to how to find $\mathbf{w}$ and bias term such that data separation is meaningful was address by Rosenbaltt's perceptron algorithm [33], which is an iterative algorithm used to update the weights based on the readily available information about where they should belong (true class) and the current response of the perceptron (actual class). The algorithm converges when perceptron can assign every data point to its actual class, given that data is linearly separable.

In classification tasks, the aim of the machine is to realize separating boundary (or boundaries) between different data instances and henceforth attribute meaningful labels (or classes) to these areas. In real life, the boundaries to be realized

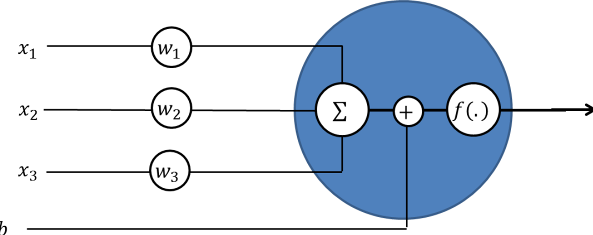


Figure 2.1: visualization of a perceptron with 3-point input

are far from linear and are expected to be arbitrarily complex. This means that a perceptron cannot simply do such classification tasks. Nonetheless, Minsky and Papert showed that one hidden layer is needed to serve as intermediate mapping in order to solve the famous **XOR** problem [34]. Furthermore, Cybenko et. al. universally proved that multilayer perceptrons with one hidden layer and non-linear activations are theoretically capable of approximating any continuous mapping, i.e. realize boundaries of any continuous non-linearities [35], This is where MLP comes to importance in solving real-world non-trivial problems.

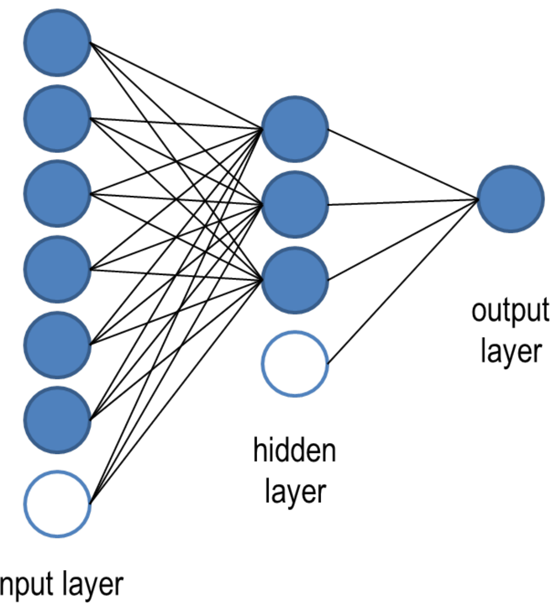


Figure 2.2: visualization of an MLP with one hidden layer, blue nodes correspond to neurons(perceptrons) and white nodes correspond to bias nodes

Nonetheless, it is not possible to use perceptron update rule because it is not

known what the internal representation of the hidden layers should be. Rumehart et. al. devised Backpropagation algorithm, a gradient-based update rule that "propagates" a differentiable error criteria through all layers. The weight are updated based on the gradient of the error sensitivity based on calculus chain rule [36]. This work has made using MLPs feasible. Since backpropagation is gradient-based algorithm, the end-end connections and nodes should be differentiable. This means that hard-limit activations as defined in 2.1 cannot be used. Real-valued soft limits: such as sigmoid function and tangent function are alternatives in that they approximate hard-limit however, differentiable. Sigmoid function $sig : \mathbb{R} \rightarrow [0, 1]$ is defined as follows:

$$sig(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

Hyperbolic tangent $tanh : \mathbb{R} \rightarrow [-1, 1]$ is defined as follows:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.3}$$

both functions are monotonously increasing and continuous, with their limits as follows:

$$\lim_{x \to \infty} sig(x) = 1$$
$$\lim_{x \to -\infty} sig(x) = 0$$
$$\lim_{x \to \infty} tanh(x) = 1$$
$$\lim_{x \to -\infty} tanh(x) = -1$$

The derivatives for sigmoid and hyperbolic tangent are given respectively as follows:

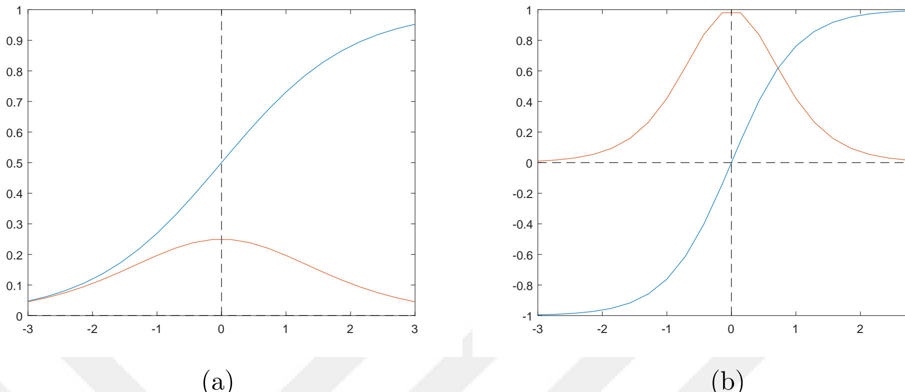
$$sig'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} \equiv sig(x)\big(1 - sig(x)\big) \tag{2.4}$$

$$tanh'(x) = 1 - \Big(\frac{e^x - e^{-x}}{e^x + e^{-x}}\Big)^2 \equiv 1 - tanh^2(x) \tag{2.5}$$

Visualization of both functions and their derivatives are shown in Fig. 2.3

2.2.1 Feedforward and Backpropagation Equations

Feedforward neural networks pass input from lower layers to higher layers, starting from the presentation layer, which is merely the data input layer, up until the



(a) (b)

Figure 2.3: Sigmoidal activation (left) and hyperbolic tangent activation (right) and their derivatives. Blue lines correspond to the functions themselves and red lines to their derivatives. Dashed lines correspond to the x- and y-axes.

output layer, without any loops in contrast to other models such as: recurrent neural network [37].

When talking about training neural network, we are also interested in backpropagation passes, which go from the output layer up until the input layer.

2.2.1.1 Feedforwarding

As for studying feed forward and backpropagation passes, our mathematical notation is as follows: Let $\mathbf{x}^n$ be input n and t^n be the true corresponding label (desired output). Let superscript $l \in \{1, 2, ..., L\}$ be the layer index, starting from 1, in which case it corresponds to the input, this layer is known as presentation layer. The Lth layer is the output layer, where $L > 2$ depends on the model choice. We assume that there is at least one hidden layer. Let v_i^l be the pre-activation of neuron i in layer l. In the case of fully connected layer l, it corresponds to the weighted sum of all activations of the preceding layer $l-1$ accumulated at that neuron's input gate. Let w_{ji}^l be the weight connecting neuron i from the layer $l-1$ with the input gate of neuron j in layer l. Let b_j^l be the bias term of that neuron, i.e. the offset value at the input gate of that neuron. Let u_i^l be the activation of neuron i in layer l, i.e. the response of the neuron after applying non-linearity. Let $f(.)$ be the non-linear activation function.

In the case of fully connected layer, Activation u_i^l can be written as:

$$u_j^l := f(v_j^l) = f\Big(\sum_i^{I-1} w_{ij}^l u_i^{l-1} + b_j^l\Big) \tag{2.6}$$

In matrix notation, vector $\mathbf{u}^l$ can be expressed as follows:

$$\mathbf{u}^\mathbf{l} := f(\mathbf{v}^l) = f(\mathbf{W}^{l^T}\mathbf{u}^{l-1} + \mathbf{b}^l) \tag{2.7}$$

where $f(.)$ in 2.7 is the element-wise nonlinearity application, $\mathbf{W}^l$ is the weight matrix connecting layer $l-1$ with later l, the superscript T denotes transpose. The importance of adding a bias term is to apply affine transformation w.r.t the input. Therefore, the pre-activation is not restricted to a value of 0 when the input to its corresponding neuron is $\mathbf{0}$.

In classification tasks, the response of the highest layer is the prediction of a NN. In a binary classification task, we can express prediction by using only one output neuron, all that is needed to interpret the response is to set a threshold as a separating boundary between negative and positive. Applying non-linearity at the end does not add any value to the prediction itself. Nevertheless, it is convenient to bound the response to values from $[-1, 1]$ or $[0, 1]$ so as to be able to set a practical cost criterion that is important for training NNs, in addition to making the prediction more interpretable by humans.

In the case of M-ary classification, i.e. categorizing data into M classes, $t^n \in \{c_1, c_2, ...c_M\}$ where c_i is the ith class, e.g 'Apple' in an image recognition task. In reality, we simply assign a unique numeric value from 1 to M for each class c. Therefore, t^n simplifies to a numeric value from 1 to M.

As for expressing prediction, it is not possible to use only one neuron for a simple reason: there is no class order in categorical data and, thus, the distance criterion should be as follow:

$$d(c_m, c_n) = \begin{cases} 0 & m = n \\ D & otherwise \end{cases} \tag{2.8}$$

what Eq 2.8 implies is that classifying $\mathbf{x}^n$ into any class except its true one should be as bad as classifying it into any other false class. Therefore, NNs should preserve this distance criterion inherit to categorical data. This is achievable by transforming true labels t's and predicted labels y's into one-hot vectors. One-hot vectors are simply vectors of M dimensions with all unit bases multiplied by

zero, except the ith base, where i is the numeric value of the label. Based upon that, one can set the output layer to have M neurons, each of which exclusively corresponds to one of possible M outcomes. The prediction is then:

$$y^n := prediction(x^n) = argmax(\mathbf{v}^L(x^n)) \tag{2.9}$$

where L is the last layer index, n is the instance index. Ideally, we want $y^n = t^n$. Note that each output neuron preforms binary classification by telling whether input x^n is of that neuron class or not.

It is also common practice to normalize the output layer response so that values are bound to an interval $[0, 1)$. This is done by applying softmax operator, which is defined as follows:

$$s_i = \frac{e^{v_i^L}}{\sum_{j=1}^{M} e^{v_j^L}} \tag{2.10}$$

This ensures that values sum up to unity. Softmax is particularly important when used in adjacency to cross-entropy loss function.

2.2.1.2 Backpropagation

Since backpropagation algorithm is gradient-based [36], this means that it is suited for end-end differentiable computational graphs. However, in the case of supervised learning, the network prediction as defined in (2.9) is itself not differentiable w.r.t any node in the network. Therefore, a cost function should be devised.

Cost function can be understood as a quantification of the network performance taking into account the desired response of the network. Therefore, training a neural nets boils down to minimizing the cost given input data. Let $J(\mathbf{x}^n, t^n; \{\mathbf{W}\})$ be the cost function of input $\mathbf{x}^n$ when its corresponding true label is t^n. As for choices for cost functions, one of the earliest choices is the square error defined as follows:

$$J(\mathbf{x}^n, t^n; \{\mathbf{W}\}) := \frac{1}{2}(\mathbf{u}^L - \mathbf{t})^T(\mathbf{u}^L - \mathbf{t}) \tag{2.11}$$

where $\mathbf{t}$ is the one hot encoded vector of the true label t^n. In the case of binary classification, another important cost metric is the cross entropy metric. In the

case of binary classification it is defined as follows:

$$J(\mathbf{x}^n, t^n; \{\mathbf{W}\}) := (1 - t^n)log(1 - u^L) - t^n log(u^L) \tag{2.12}$$

where u^L is the sigmoidal response of the output neuron In case of M-ary classification softmax is usually applied over the output layer and cross entropy cost is defied as follows:

$$J(\mathbf{x}^n, t^n; \{\mathbf{W}\}) := -\sum_{i}^{M} t_i^n log(s_i) \tag{2.13}$$

where t_i^n is the ith component of the one-hot encoded vector corresponding to scalar t^n, s_i is the ith softmax logit.

In order to propagate the error (or cost) backward, the error sensitivity w.r.t output bias is defined as:

$$\boldsymbol{\delta}^L := \nabla_{\mathbf{b}^L} J := \frac{\partial J}{\partial \mathbf{b}^L} \tag{2.14}$$

The dimensionality of $\boldsymbol{\delta}^L$ is identical to that of the bias vector $\mathbf{b}^L$ and thus to that of $\mathbf{v}^L$. This sensitivity depends on: 1) the response of the output neurons, 2) the true labels and 3) the choice of the cost criterion J. As for using cross entropy with softmax logits. The sensitivity w.r.t the ith bias is:

$$\delta_i^L = \frac{\partial J}{\partial b_i^L} = \frac{\partial J}{\partial v_i^L} = \sum_{m=1}^{M} \frac{\partial J}{\partial s_m} \frac{\partial s_m}{\partial v_i^L} = s_i - t_i \tag{2.15}$$

where in (2.15) t^i is the ith component of the one-hot true vector, s_i is the softmax output of the ith neuron. Note that δ's in this case $\in (-1, 1)$.

The sensitivity is then propagated to from layer $l+1$ to layer l using the following recursive formula:

$$\boldsymbol{\delta}^l = (\mathbf{W}^{l+1}\boldsymbol{\delta}^{l+1}) \circ f'(\mathbf{v}^l) \tag{2.16}$$

where in (2.16) $\circ$ denotes Hadamard (element-wise) multiplication between the vector resulting from multiplying the weight matrix with delta of the higher layer $l+1$ and vector $f'(\mathbf{v}^l)$, $f'(.)$ is the element-wise application of the derivative function. Since activation $f(.)$ has an specific analytical form, its derivative is known analytically.

The ultimate goal of back-propagation is to find the derivative of the cost function w.r.t weights so as to use the derivative information to update the weights. In this regard, the sensitivity vector δ^l is used directly to update the weights of

its corresponding layers using direct application of calculus chain rule, the cost function gradient w.r.t weights of layer l is as follows:

$$\nabla_{\mathbf{W}^l} J = \frac{\partial J}{\partial \mathbf{W}^l} = \mathbf{u}^{l-1}(\boldsymbol{\delta}^{\boldsymbol{l}})^T \tag{2.17}$$

The gradient obtained from (2.17) will be used to update the respective weight matrix iteratively using the following formula:

$$\mathbf{W}^l_{I+1} = \mathbf{W}^l_I + \psi\Big(\frac{\partial J}{\partial \mathbf{W}^l}\Big) \tag{2.18}$$

Furthermore, the sensitivity vectors will directly be used to update the biases as follows:

$$\mathbf{b}^l_{I+1} = \mathbf{b}^l_I + \psi(\boldsymbol{\delta}^l) \tag{2.19}$$

Where I is the iteration index, ψ is the gradient-based update rule.
As for update rules, the simplest rule is Gradient Descent algorithm (GD), in which the subjects are updated in a direction exactly opposite to the gradient, i.e. $\psi(\nabla) = -\eta\nabla$, where η is the learning rate. Nevertheless, Gradient Descent is never used and Stochastic Gradient Descend (SGD) and its variants are applied. In SGD, the input-label tuples are randomly permuted and then iterated over. This is important to insure that different samples are independent from each other, and thus, the network is guaranteed to not learn any relations between different samples as it is irrelevant to conventional classification tasks [38].
Since optimizing the cost function is non-convex, the updating algorithm will get stuck in a local minimum. As a matter of fact, it is very unlikely to ever hit the global minimum. Therefore, other algorithms have been devised to help the parameters, to some extent, escape poor local minima. One algorithm is the momentum update rule [38], which can be expressed mathematically as follows:

$$\begin{aligned} \mathbf{V}_{I+1} &= \mathbf{V}_I - \eta\frac{\partial J}{\partial \mathbf{W}^l} \\ \mathbf{W}^l_{I+1} &= \mathbf{W}^l_I + \mu\mathbf{V}_{I+1} \end{aligned} \tag{2.20}$$

where $\mathbf{V}$ is the momentum term, μ is the momentum rate. There many variants of SGD that are readily implemented in high level deep learning libraries such as:

Tensorflow [28], Caffe [30], Theano [29] and others.

2.3 Convolutional Neural Networks

Convolutional Neural Networks (also known as ConvNets or CNNs) are a class of feedforward neural networks where spatial convolution is the operation applied between inputs and weights instead of applying ordinary matrix multiplication. Conventionally, CNNs are feedforward neural networks that have convolution operations induced at least between two layers. Usually, the layers are convolutional except the last few layers (usually 2 or 3) which are fully (densely) connected[39]. Mathematically speaking, discrete convolution is defined as follows:

$$(x * w)_n := \sum_k w_k x_{n-k} := \sum_k x_k w_{n-k} \tag{2.21}$$

However, in the context of deep learning, convolution is sometimes defined as follows:

$$(x * w)_n := \sum_k w_k x_{n+k} \tag{2.22}$$

The difference between 2.21 and 2.22 is that in 2.21 we perform kernel (w) or input (x) flipping in contrast of 2.22. It is important to note that the second definition does not quality for proper convolution but rather cross-correlation, which is not commutative ($\sum_k w_k x_{n+k} \neq \sum_k x_k w_{n+k}$). Nevertheless, as far as feedforwarding and backpropagation are concerned, this distinction is not of importance, and one can use either definitions without any practical differences.

Since we are dealing with tensors in deep learning networks, definition 2.21 is straightforwardly extended as follows:

$$\big(I * W\big)(x, y) := \sum_j \sum_i W(i, j) I(x - i, y - j) \tag{2.23}$$

Equation 2.23 refers to the case of having 2D input. However, it can be extended to 3D input and 3D kernels. Convolution between input I and kernel W results in a 2D output or a **feature map**. In ConvNets, however, we have multiple filters at a certain levels that comprise a **filter bank**, each of which convolves

with input I to produce a feature map stacked at a certain position. Therefore, a more generic definition of convolution in feedforward pass can be articulated as follows:

$$V(x, y, k) := \big(I * W_k\big)(x, y) \tag{2.24}$$

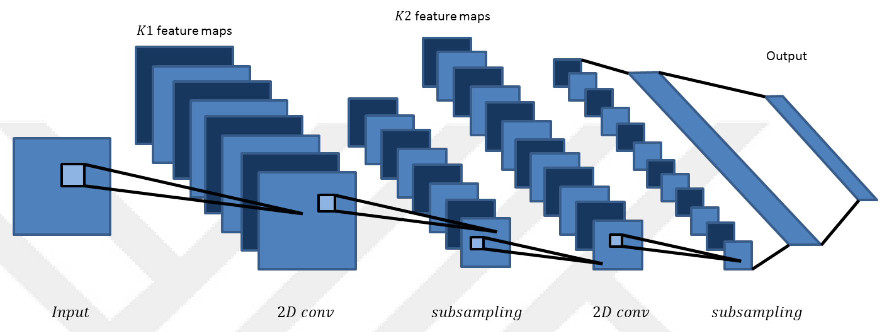


Figure 2.4: A typical ConvNet, inspired by [1]

Looking back at 2.23, it can be seen that the depth of the output or the number of feature maps only depends upon the number of the filters in the filter bank. The spatial size depends on the convolution type as well as the striding parameters (explained below).

Figure 2.4 visually demonstrates a typical Convnet. As it can be seen, the spatial size of high-level features maps is quite small and, in some architectures can be singular, i.e. 1×1, whereas the depth increases. This is to allow CNNs to capture many complex patterns that will be then fed into the higher fully connected layers to perform the intended task, such as: classification, detection or localization.

Convolutional neural networks differ from multi layer perceptrons in that the weights (or kernels) in the convolutional layers have smaller size than input size and, thus, certain input units contribute to an output unit. This can be restated as: output has **local receptive fields**. Secondly, weights are shared across spatial dimensions. The former constraint yields efficiency in computation [40] since a fewer add-multiply operations, and, thus, dot product operations are needed to calculate the output in feedforward pass. Furthermore, this allows kernels to learn local features such as: directional edges, points and corners. Parameter sharing is important when it comes to visual features since the location of a feature is usually unimportant but rather its presence. In addition to that, **spatial pooling** is often applied between convolutional layers. Spatial size reduction,

which is achievable one way by pooling, is important to reduce the response size in feedforward and, therefore, enable higher layers of learning data-dependent generic features, e.g. facial patterns in a face recognition task. Combining these specifics, ConvNets are, to some extent, shift, scale and distortion invariant feature extractors [41].
Just as in other feedforward NNs, non-linearity is applied between hidden layers or feature maps in ConvNets. Although any sigmoidal non-linearity is a valid choice, the most commonly used non-linear activation is Rectified linear function (or $ReLU$), which is defined as follows:

$$ReLU(x) = max(0, x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases} \tag{2.25}$$

Although it is not clear why choosing ReLU in ConvNets has yielded better results, it can be argued that ReLU is less likely to cause vanishing gradient (see section 2.3.2). Another advantage of using ReLUs is that it yields sparse output, since all negative responses will be mapped to zero. This is beneficial in terms of efficiency as these units do not need be multiplied by weight connections while carrying out convolution, which can be utilized in the context of sparse matrix multiplication.[add ref]. Other variants of basic ReLU include: Leaky ReLU which is defined as: $LeakyReLU(x; \epsilon) = max(\frac{x}{\epsilon}, x)$, where ϵ is the leakage parameter, which determines how much of the negative signal should be 'leaked'. Note that when $\epsilon = 1$, the mapping reduces to identity(linear) mapping.

Pooling operation, as the name indicates, realizes a singular value out of a local area (say 2×2) based on a certain criterion. The most common choices are: average pooling, maximum pooling and p-normed pooling.

$$I_{pool}(x, y) = max_{(x,y) \in Area}(I(x, y)) \tag{2.26}$$

Depending on the domain selection, we can distinguish between 3 types of convolution: 'Same' convolution, 'Valid' convolution and 'Full' convolution.The difference between these thee types is the support of the convolution operation.

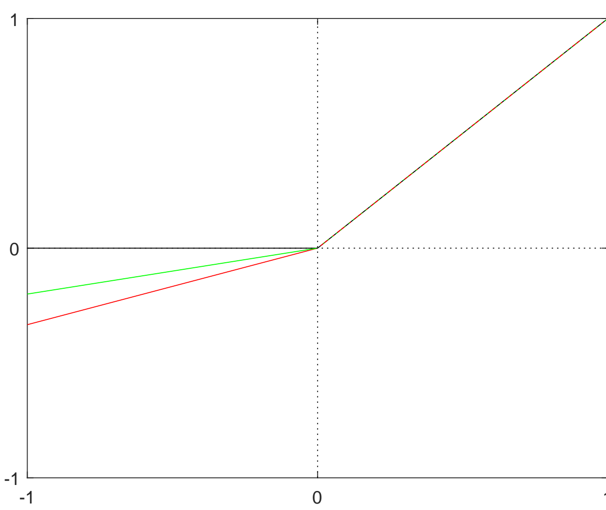


Figure 2.5: Visualisation of ReLU and its variants, black lines corresponds to ReLU, green line and red line correspond to LeakyReLU with leakage factor of 5 and 3, respectively

Since we're dealing with discrete domain ($\mathbb{Z}$), the support $Supp(.) \subset \mathbb{Z}^n$ is defined as $\{\vec{x} \in \mathbb{Z}^n \quad s.t. \quad (I * W)(\vec{x}) \neq 0\}$. In 1-D case, 'Full' convolution has the largest support possible, where $|Supp(I * w)| = |Supp(I)| + |Supp(W)| - 1$. In 'Same' convolution the domain is chosen such that the support of the convolution output is the same as that of the input I. In 'Valid' convolution the domain is restricted only to cases where W and I convolve on I support and, thus, $|Supp(I * W)| = |Supp(I)| - |Supp(W)| + 1$. 'Same' convolution is used the most since the support (and thus the size) of the output is the same as that of the input, which simplifies design.

When talking about ConvNets, there is an important additional parameter ascribable to convolution operation, which is **Striding**. Strides determine how much spacing in the input should be taken while performing convolution. In the abovementioned definitions, vertical and horizontal strides are set to 1, that is, we slide the kernel by one pixel (vertically or horizontally) to find the (vertically or horizontally) adjacent output. Let S_V, S_H denote vertical and horizontal striding parameters, respectively, then strided convolution can be defined as follows:

$$Conv2D(I, W; S_H, S_V) = \sum_j \sum_i W_k(i,j) I(S_H x - i, S_V y - j) \tag{2.27}$$

Strided convolution can be thought of as ordinary spatial convolution followed by a sub-sampling layer. It is obvious that with striding > 1 the size of feature maps will be reduced by $S_H S_V$. Striding achieves spatial size reduction, just as pooling does. Therefore, one can dispense with using pooling by choosing striding parameters > 1. In [42] the authors advocate for using strided convolution and eliminate pooling layers., while achieving a very high classification accuracy over CIFAR datasets.

2.3.1 feedforward and backpropagation in ConvNets

2.3.1.1 feedforwarding pass

Just like other feedforwarding NNs, learning weights in ConvNets is gradient-based. Therefore, in this subsection, feedforward pass and backpropagation equations are derived for CNNs.

Since we are dealing with grey-scale or RGB-color input images as well as 3D tensors, mathematical formulation needs further indexing than in the case of MLP, the input and responses are vectorized. Using the same notation convention adopted in 2.2.1, let $\mathbf{V}^l$ and $\mathbf{U}^l \in \mathbb{R}^{\mathbf{X}\times\mathbf{Y}\times\mathbf{D}}$, where $\mathbf{X}$ and $\mathbf{Y}$ are the spatial domains on which the tensor is defined (the support) and $\mathbf{D}$ is the depth domain of that tensor. As for the kernels, filter banks are denoted as follows: for layer l, let the filter bank be a 4-D tensor such that $\mathbf{W}^{\mathbf{l}} \in \mathbb{R}^{\mathbf{I}\times\mathbf{J}\times\mathbf{D}\times\mathbf{K}}$, where I and J are the filter domain in x and y directions, respectively, $\mathbf{D}$ is the depth domain and K is the filter index within the filter bank. the kth filter in $\mathbf{W}^l$ is a 3-D tensor whose depth is identical to that of the input U^{l-1}. for simplicity, it can be denoted as $\mathbf{W}_k^l \in \mathbb{R}^{\mathbf{I}\times\mathbf{J}\times\mathbf{D}}$. Based on this indexing scheme, the feedforward pass can be written as follows:

$$\mathbf{V}^l(k) := Conv2D(\mathbf{W}_k^l, \mathbf{U}^{l-1}) + b_k^l \tag{2.28a}$$

in scalar notation

$$v^l(x,y,k) := \sum_{d=1}^{D}\sum_{j\in J}\sum_{i\in I} w_k^l(i,j,d)u^{l-1}(x-i,y-j,d) + b_k^l \tag{2.28b}$$

according to (2.28), filter $\mathbf{W}_k^l$ will convolve with $\mathbf{U}^{l-1}$ and yield the 2-D pre-activation of the kth feature map. $\mathbf{V}^l$ does not depend on the depth of the filter or the output $\mathbf{U}^{l-1}$, but rather on x,y and the index k. [add fig]
The bias $\mathbf{b}^l$ is usually taken as a vector $\in \mathbb{R}^{\mathbf{K}}$ so that for each pre-activation map V_k^l, a scaler bias term is added to yield affine transformation. This make a difference to the MLP case when it comes to backpropagating the error.
The highest feature map, which conventionally has a small spatial size, e.g. 6×6 and a large depth, e.g. 512 is vectorized before being fed into the fully connected layer, henceforth, formulation in Sec. 2.3.1 are used.

2.3.1.2 backpropagation pass

In order to derive error sensitivities and gradient w.r.t, calculus chain rule is used. As for error sensitivity w.r.t pre-activation, it can derived as follows:

$$\delta^l(x,y,z) := \frac{\partial J}{v^l(x,y,z)} = \sum_k\sum_{y'}\sum_{x'} \frac{\partial J}{\partial v^{l+1}(x',y',k)}\frac{\partial v^{l+1}(x',y',k)}{\partial v^l(x,y,z)} \tag{2.29}$$

The first RHS term inside the summation in(2.29) is $\delta^{l+1}(x',y',k)$ by definition. As for the second term, it is important to note that k^{th} feature map is not depended upon any filter except the kth one. Therefore, the second term inside the summation in (2.29) becomes:

$$\frac{\partial v^{l+1}(x',y',k)}{\partial v^l(x,y,z)} = w_k^{l+1}(x'-x,y'-y,z)f'\big(v^l(x,y,z)\big) \tag{2.30}$$

Plugging (2.30) into (2.29), it becomes:

$$\delta^l(x,y,z) = \Big(\sum_k\sum_{y'}\sum_{x'} \delta^{l+1}(x',y',k)w_k^{l+1}(x'-x,y'-y,z)\Big)f'\big(v^l(x,y,z)\big)\Big) \tag{2.31a}$$

or in tensor notation

$$\boldsymbol{\delta}_z^l = \Big(\sum_k Conv2D\big(\boldsymbol{\delta}_k^{l+1}, rot\{\mathbf{W}_k^{l+1}\}(z)\big)\Big) \circ f'\big(\mathbf{V}^l(z)\big) \tag{2.31b}$$

where in (2.31) $rot\{.\}$ implies rotating the filter in x and y directions by 180°, the convolution is carried out between the zth slice of the rotated kernel w_k^l and δ^{l+1}, at k^th position at a time. This is carried out for each kernel w_k^l and its corresponding delta map δ_k^{l+1} before being added up into a single 2D tensor, which is multiplied element-wise by tensor $f'\big(v^l(x, y, z)\big)$. the convolution in (2.31) is type 'full' if the convolution in feedforwarding is type 'valid' in order to ensure dimensionality consistency in backpropagation.[add figure]
The error sensitivity w.r.t bias b_z^l is as follows:

$$\frac{\partial J}{\partial b_z^l} = \sum_y \sum_x \delta^l(x, y, z) \tag{2.32}$$

The error gradient w.r.t weight kernel w_l^k is:

$$\frac{\partial J}{\partial w_k^l(x, y, z)} = \sum_{y'} \sum_{x'} \frac{\partial J}{\partial v^l(x', y', k)} \frac{\partial v^l(x', y', k)}{\partial w_k^l(x, y, z)} \tag{2.33}$$

The first RHS in (2.33) is $\delta^l(x', y', z')$, the second term is evaluated as:

$$\frac{\partial v^l(x', y', k)}{\partial w_k^l(x, y, z)} = u^l(x' - x, y' - y, z) \tag{2.34}$$

Plugging (2.34) into (2.33) yield the following formula:

$$\frac{\partial J}{\partial w_k^l(x, y, z)} = \sum_{y'} \sum_{x'} \delta^l(x', y', k) u^l(x' - x, y' - y, z)$$

or in tensor notation

$$\frac{\partial J}{\partial \mathbf{W}_k^l(z)} = Conv2D(\boldsymbol{\delta}_k^l, rot\{\mathbf{U}^l(z)\}) \tag{2.35}$$

2.3.2 Historical Background and Advancement

Convolution in CNNs can be understood as a special form of vector product where not all input contributes to a certain output unit and the weights are shared. The

idea of parameters sharing in feedforward neural networks dates back to 1988, where it was first applied in "Time-delay Neural Networks" in phoneme recognition tasks [10]. However, the first attempt to apply parameter sharing in neural network for image recognition tasks was done by le Cun et. al [40] in 1989. In that work, weight-shared networks were used in classifying binary images of handwritten digits. 2×2 weights sets were shared across the 16×16 input images and the output of the intermediate layers and a fully connected layer was used in the end. Sigmoidal activation was applied to impose non-linearity. Furthermore, these networks were trained using the then recently developed back-propagation algorithm[43] and showed +10% performance improvement in inference accuracy over fully connected networks. In 1990, weight-shared networks were used in a more challenging recognition task: recognizing zipcodes for handwritten images [6].

As these results were promising, using deeper model in more challenging recognition tasks would be sought. However, this strive was frustrated by encountering a problem related to gradient-based learning, i.e. backpropagation itself: the problem of vanishing gradient [44]. Since back-propagation algorithm uses calculus chain rule of derivatives to calculate the gradient of the loss (or cost) function w.r.t a layer, the gradient will vanish if the error sensitivity propagated from higher layers is too small. This mostly happens when sigmoidal response is in the saturated areas ($f(x) \approx 0$ or 1). Therefore, the derivative then is close to zero. This made training early layers very hard. Other activations used then such as: Hyperbolic tangent also incur the same problem (derivative is effectively zero when $f(x) \approx -1$ or 1). Due to the resulting inefficiency in gradient-based learning and due to the computational limitations at the time, training deeper models was infeasible and thus it was not possible to obtain satisfactory results promised by the theoretical capabilities of better generalization[45]. This could not be simply solved by using larger learning rates, as a new problem would emerge: the problem of exploding gradient. In addition to that, support vector machines (SVM)[46] caught interest among researchers in 1990s at the expense of neural networks. Nevertheless, there was ongoing, albeit limited, success in applying CNNs in computer vision tasks such as: optical charachter recognition (OCR) as well as facial recognition [41]. For further details see [1].

In 2003, Simard et. al. [47] achieved then state-of-the-art recognition performance over MNIST dataset [48] with performance error equal to 0.4% with CNNs by devising novel dataset expansion techniques: elastic and affine distortion. Valid data augmentation techniques, by which the labels are preserved, help the model be more transformation-invariant and, thus, generalize better.

In 2006, Hinton et. al. [49] devised a new scheme so as to accelerate training deep MLP models with many layers. In that work greedy layer-wise unsupervised training is carried out initially. Afterwards, the layers are cascaded and the learned weights serve as initialization values for the supervised task (e.g. classification). In the supervised learning phase, the gradient-based optimizer only "fine-tunes" the values of the weights. This work caught the attention of many researchers in the upcoming years and more effort was devoted to achieving better results in hard computer vision tasks

In 2010, Cireşan et. al. [50] trained large MLP networks (including up to 9-layer models) and could achieve classification accuracy of $\approx$ 0.35% error rate over MNIST data set using old-school end-to-end backpropagation training with no need to pre-process the data or carry out any layer-wise pre-training. This work is perhaps one of earliest attempts to expedite training deep models by making use of the parallelization capabilities offered by Graphic Processing Units (GPUs). With the advent of more powerful and dedicated GPUs as well as GPU-supported deep learning libraries, using GPUs has become pervasive in training deep models. GPUs have surpassed CPU clusters in their parallelized computational capabilities.

In 2012, Krizhevsky et. al. devised a deep ConvNet and trained over ImageNet dataset [2]. The importance of this work comes out of the fact that it was the first successful attempt in using CNNs or neural networks in a challenging dataset and being able to achieve ground-breaking results within the computer vision society. This work has changed the perception of neural nets among scholars and has prompted further development in the field. Other famous networks include: VGG net [5], GoogleNet [3] and many others.

2.4 Benchmark Datasets

One key factor in the advancement of deep learning research is the availability of large scale dataset that have been collected throughout the years. Although the ultimate objective is to achieve good performance over real-world datasets. Benchmark datasets serve as a standard way of assessing performances among different models and architectures. Furthermore, benchmark datasets can be adequately hard such that achieving good results is on par of real-world recognition problems.
There are plenty of common datasets used in the community. However, we will discuss only the two datasets relevant to our experiments: MNIST and CIFAR dataset.

2.4.1 MNIST Dataset

MNIST dataset consists of 0-9 digit Gray scale images, i.e. has 10 classes [48]. The images size is 28×28 pixels. The dataset has $60,000$ images split between training and validation data as well as $10,000$ test images. Currently, the state-of-the-art accuracy over the test data is 99.8% achieved by DropConnect [51]. Example photos are shown in Fig. 2.6.

2.4.2 CIFAR Dataset

CIFAR-10 consists of natural coloured images [7]. The dataset has $60,000$ images with 10 classes, each of which has $6,000$ images. The images are of size 32 RGB pixels. $50,000$ images are training images and $10,000$ are testing images . CIFAR-100 is simiar to CIFAR-10 but with 100 classes. This means that fewer examples are available in CIFAR-100 for each class, which makes a classification task over CIFAR-100 much harder than over CIFAR-10. This is evident in their state-of-the art rate of recognition, which that of CIFAR-100 roughly 76.0% [52], while

Figure 2.6: Example samples from MNIST dataset (not shown to scale)

the state of the art in the case of CIFAR-10 is roughly 96.5% [4, 53]. Sample photos are shown in Fig. 2.7.

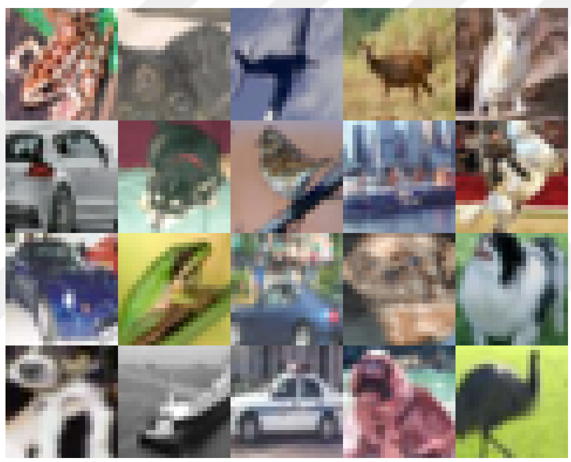


Figure 2.7: Example samples from CIFAR-10 dataset (not shown to scale)

Chapter 3

Related Work

Neural Networks have become very popular in recent years in computer vision, natural language processing and speech recognition thanks as they have achieved state-of-the-art performance surpassing classical machine learning techniques at near-human performance levels [2, 53, 52]. Deep Neural Networks, however, are computationally intensive since the number of arithmetic operations involved in feed-forward pass is very high. In fact, in a typical ConvNets, the number of add-multiply operations is usually in order of tens of millions. This has been an obstacle towards using ConvNets in systems where processing and energy are limited [54], such as: Mobile phones and smart sensory devices.

There has been interest in energy efficient neural networks from different points of view. Weights quantization is one approach to achieving efficiency. Techniques such as: precision scaling and computation skipping can save up energy [14]. Dedicated energy efficient Neuromorphic systems have been developed [15, 55]. Specialized hardware such as FPGA has been used in achieving energy efficiency [56].

In 2013, Wan e.t al. [51] proposed **DropConnect**, in which weight connections are randomly dropped during feedforwarding pass. The weight dropping scheme serves as regularization. Furthermore, connections dropping implies fewer arithmetic operations during feedforwarding. However, DropConnect is only limited to dense connections.

In 2015, Courbariaux et. al. [18] proposed **BinaryConnect**, which is a DNN where weights are binarized during feedforwarding pass, either deterministically or stochastically. In the backpropagation pass, the sensitivities are calculated for the binary weights but weights values are retained for parameters update. Binarization in BinaryConnect serves as regularization. YodaNN [57] is an ASIC (Application-specific integrated circuit) design of binaryConnect that could achieve up to 32× energy reduction than other ASICs CNNs. BinaryConnect networks were extended to TernaryConnect, where weights can have values of either : $-1, 0, 1$ [58].

In 2016, Kim et. al. [17] proposed **Bitwise Neural Network (BNN)**, where input, weights and activation are 1-bit valued. In BNNs, XNOR is the binary operation applied in feedforwarding between the weights and input from lower layers. The weights are compressed by using hyperbolic tangent activations. Afterwards, the real-valued trained weights are retrained into binary weights using noisy backpropagation. However, the method was not tried in deep learning architecture.

In 2016, Rastegari et. al [16] proposed two networks: Binary Weight Network and **XNOR** network, where in the former, weights are binary-valued and in the latter weights and input tensors are both binary-valued. In BWN, a pre-activation real-valued scaling scheme is used in order to make the network trainable using standrad backpropagation. Likewise, in XNOR network, another scaling factor is used to apprixmate dot product between binary weights and binary inputs.

Ternary Weight Networks (TWN) were proposed by Zhang et. al. [59], where weights can take up 3 values $\{-1, 0, +1\}$ instead of binary values. TWN outperforms BWN while still being energy efficient.

In 2009, Tuna et. al [25], proposed an ℓ_1-norm inducing multiplier-less binary operator, upon which our AddNet is built. This operator was used to realize the so-called co-difference matrix between feature vectors. Co-difference matrix resembles co-variance matrix but dot product is replaced with the referred operator. It was shown that image descriptors based on co-difference matrices can perform as well as those based on co-variance matrices in image classification and identification tasks. This multiplier-less operator has found applications in computer vision and signal processing [26, 60, 61].

In 2017, Afrasiyabi et. al. [62], applies the ℓ_1-norm inducing operator in neural net in order to achieve energy efficiency. The work shows that a multiplier-less network based on the above-mentioned operator can solve the famous **XOR** problem and it shows good results on MNIST dataset. However, this work only investigated multilayer perceptrons. Our earlier work was carried out during the same time of the above-mentioned work. Furthermore, this thesis focuses on using the multiplier-less operator in deep neural networks and carrying out experiments on harder datasets, such as: CIFAR-10. Furthermore, we present more mathematical choices for the multiplicative bias in AddNets.

Chapter 4

Non-Euclidean Operators and Neural Nets

4.1 Overview

This chapter describes non-euclidean ℓ_1-inducing operators and their applications in Neural Nets as a replacement for ordinary dot products used in realizing responses across neural networks. In this ℓ_1-norm scheme, realizing responses through weighted sums is multiplication-free This means that fewer multiplication operations overall from end-to-end in feedforwarding passes, which is energy saving [16, 18, 17]. The operator of interest is called **Oplus** and notated as $\oplus$. We call a neural net which is partially or fully based on operator $\oplus$ **addNet**. In addition to energy saving, operator $\oplus$ possess some ℓ_1-norm features such as resilience against outliers. This means that AddNets will behave differently than normal neural nets when the data is corrupt.
The outline of this chapter is as follows: Sec. 4.2 introduces operator $\oplus$ as an ℓ_1-norm inducing operator. Furthermore, discussion about energy efficiency is provided as well as the behaviour of $\oplus$-based systems under noisy inputs.
In the second part (Sec.4.3), we introduce AddNets and their building blocks:

AddNeurons, we formulate feedforwarding pass equations as well as backpropagation. Furthermore, we introduce **Multiplicative Bias**: a normalization scheme at the response level that is necessary so that AddNet can behave properly in feedforward and backpropagation. A mathematical justification is also presented. In addition to that, we present different choices for this multiplicative bias that we investigated in our experimentations. We also discuss "sparse" operator $\oplus$, where only the signs of the weights are kept and the magnitudes are discarded.

4.2 ℓ_1-norm Inducing Operators

The ℓ_1 norm is a non-Euclidean norm that belongs in Minkowski norms family, i.e. it satisfies Minowski inequality [63].

$$||x+y||_p \leq ||x||_p + ||y||_p \tag{4.1}$$

where $p \geq 1$. The ℓ_p-norm is defined on discrete vector spaces as follows:

$$||\mathbf{x}||_p := \Big(\sum_{i=1}^{N} |x_i|^p\Big)^{1/p} \tag{4.2}$$

The Euclidean (ℓ_2) norm is defined as follows:

$$||\mathbf{x}||_2 := \sqrt{\sum_{i=1}^{N} |x_i|^2} \tag{4.3}$$

In the case of the ℓ_1 norm $p = 1$ and (4.2) reduces to:

$$||\mathbf{x}||_1 := \sum_{i=1}^{N} |x_i| \tag{4.4}$$

where $|.|$ is the absolute value. The gradient of the ℓ_1 norm w.r.t its input vector $\mathbf{x}$ is:

$$\nabla_{\mathbf{x}} ||\mathbf{x}||_1 = sgn(\mathbf{x}) \tag{4.5}$$

Where sgn is the element-wise application of Signum function, i.e. $sgn(\mathbf{x}) := \{sgn(x_i)\}_{i=1}^{N}$, where N is the dimension of the vector. Signum function ($sgn(.)$) is defined as follows:

$$sgn(x) = \begin{cases} 1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases} \tag{4.6}$$

The ℓ_2 norm can be induced for vector $\mathbf{x}$ using dot product $||\mathbf{x}||_2^2 =< \mathbf{x}, \mathbf{x} >$. Based on the definition of Signum function, one can write $|x|$ as $x.sgn(x)$ and therefore induce the ℓ_1 norm of vector $\mathbf{x}$ as follows:

$$||\mathbf{x}|| := \sum_{n=i}^{N} |x_i| = \sum_{n=i}^{N} x_i sgn(x_i) =< \mathbf{x}, sgn(\mathbf{x}) > \tag{4.7}$$

However, the ℓ_1-inducing operation in (4.7) is not commutative since $< \mathbf{x}, sgn(\mathbf{y}) > \not\equiv < \mathbf{y}, sgn(\mathbf{x}) >$. In order to overcome the non-commutativity problem, several binary operations have been defined [25, 26, 64]. The first operator $\oplus$ is defined as follows:

$$\mathbf{x} \oplus \mathbf{y} := \sum_{i=1}^{N} sgn(x_i.y_i)\big(|x_i| + |y_i|\big) \tag{4.8}$$

where N is the dimensionality of both vectors. Using the fact that $sgn(x_i.y_i) \equiv sgn(x_i).sgn(y_i)$ and the fact that $|x| \equiv sgn(x).x$, (4.8) can be re-written as:

$$\mathbf{x} \oplus \mathbf{y} := \sum_{i=1}^{N} \big(sgn(x_i).y_i + sgn(y_i).x_i\big) \tag{4.9}$$

It is helpful to express the vector operation $\oplus$ as a summation of scalar operations as follows:

$$\mathbf{x} \oplus \mathbf{y} := \sum_{i=1}^{N} sgn(x_i.y_i)\big(|x_i| + |y_i|\big) := \sum_{i=1}^{N} x_i \oplus_s y_i \tag{4.10}$$

where the subscript s in $\oplus_s$ stands for "scalar".

Based on (4.9), we can break operation $\oplus$ into dot-product operations as follows:

$$\mathbf{x} \oplus \mathbf{y} \equiv < sgn(\mathbf{x}), \mathbf{y} > + < \mathbf{x}, sgn(\mathbf{y}) > \tag{4.11}$$

Expressing operator $\oplus$ in terms of the dot-product and element-wise Signum operations is handy when it comes to high-level implementations from a practical

point of view. Operator $\oplus$ induces a scaled ℓ_1 norm as follows:

$$\mathbf{x} \oplus \mathbf{x} = 2||\mathbf{x}||_1 \tag{4.12}$$

It is worth mentioning that inducing ℓ_1-norm can be achieved through other commutative vector binary operations, such as: the Min operator and the Max operator, defined respectively as follows:

$$\mathbf{x} \odot_{\downarrow} \mathbf{y} := \sum_{i=1}^{N} sgn(x_i.y_i)min(|x_i|, |y_i|) \tag{4.13}$$

$$\mathbf{x} \odot_{\uparrow} \mathbf{y} := \sum_{i=1}^{N} sgn(x_i.y_i)max(|x_i|, |y_i|) \tag{4.14}$$

Both operators in (4.13) and (4.14) induce ℓ_1 norm, i.e. $\mathbf{x} \odot_{\downarrow} \mathbf{x} \equiv \mathbf{x} \odot_{\uparrow} \mathbf{x} = ||\mathbf{x}||_1$

4.2.1 Properties of Operator $\oplus$ and $\oplus_s$

In addition to its ability to induce the ℓ_1, operator $\oplus$ possesses other properties that are worth mentioning:

4.2.1.1 Commutativity

Proposition 1. *Operator $\oplus$ is commutative*

Proof. Since operator $\oplus_s$ is commutative, $\oplus$ is summation of commutative operations, therefore it is commutative. □

4.2.1.2 Sign Preservation

Perhaps the most important property of operator $\oplus_s$ is its ability to preserve the sign of normal multiplication on a scalar level.

Proposition 2. *Operator $\oplus_s$ preserves the sign of normal multiplication.*

Proof. since $sgn(|x| + |y|) \equiv 1$, $sgn(x.y)(|x| + |y|) = sgn(x.y)$ □

The sign preservation property on a scalar level is an important property which can be extended to a vector level to make $\oplus$ resemble dot product. In this regard, Tuna et al. [25] defines a "co-difference" matrix that resembles normal co-variance matrix and use it as image feature descriptor. Co-variance matrix can be defined as:

$$Cov(\mathbf{F}) = \frac{1}{N-1} \sum_{k=1}^{N} (\mathbf{f}_k - \boldsymbol{\mu})(\mathbf{f}_k - \boldsymbol{\mu})^T \tag{4.15}$$

As for co-difference matrix, it is defined as follows:

$$Cod(\mathbf{F}) = \frac{1}{N-1} \sum_{k=1}^{N} (\mathbf{f}_k - \boldsymbol{\mu}) \oplus (\mathbf{f}_k - \boldsymbol{\mu})^T \tag{4.16}$$

where $\boldsymbol{\mu}$ is the mean vector estimate of features vectors.

4.2.1.3 Non-linearity

Proposition 3. *Operator $\oplus_s$ is non-linear*

Proof. A counterexample to linearity: $2 \oplus_s 3 = 5$, $-1 \oplus_s 3 = -4$
$(2 + -1) \oplus_s 3 = 4$, however $(2 \oplus_s 3) + (-1 \oplus_s 3) = 5 - 4 = 1 \neq 4$ □

4.2.2 Operator $\oplus$ and Energy Efficiency

From a computational point of view, operator $\oplus$ can be implemented in a multiplication-free scheme. Looking at definition (4.8), $sgn(x_i.y_i) \equiv sgn(x_i).sgn(y_i)$, therefore, The multiplication between the two *signum* terms can be realized using inexpensive operations: **XOR** in the case of binary sign or using 2-bit simple logic in case the of ternary sign. the term $|x_i| + |y_i|$ can be realized using unsigned addition and normal addition will be needed eventually to sum up the all the terms contributed by all N components of vector tuple

$(\mathbf{x}, \mathbf{y})$.
There has been interest in recent years in using fixed-point arithmetic in neural networks [65, 66, 67] . This motivation stems from the fact that arithmetic in NNs are error-tolerant, thanks to the high dimensionality data and the redundancy present and the non-uniqueness of the weights that can achieve targeted recognition rate [40]. Furthermore, techniques such as batch normalization [68] and local response normalization [2] help control the range of responses and the weights in feedforwarding passes. Fixed-point representations have simpler arithmetic and therefore more energy efficient.
Since it can be implemented using using ($+$, $|+|$, SL) operations, where, $|x|$ is unsigned addition, SL represents simple 1-bit or 2-bit logic. Operator $\oplus$ is based on operations that consume less energy in fixed-point arithmetic.

4.2.3 Operator $\oplus$ and Noise

The ℓ_1-norm is a more robust metric against outliers [21, 22, 23, 24] than classical ℓ_2-norm metrics, which known to be sensitive towards noise and outliers. It is worth mentioning that ℓ_1 based schemes can also be used in adaptive filtering for α-stable processes, an important class of non-Gaussian processes [69]. Since operator $\oplus$ induces ℓ_1-norm, operator $\oplus$ is expected to be able to account less for outliers in data due to its additive nature rather than normal dot product. In case of additive noise, we can write the response of operator $\oplus$ as follows:

$$\begin{aligned}(\mathbf{x}+\boldsymbol{\varepsilon}) \oplus \mathbf{y} &= \sum_{i}^{N} sgn\big((x_i+\varepsilon_i).y_i\big)(|x_i+\varepsilon_i|+|y_i|) \\ &= \sum_{i}^{N} \Big(sgn(x_i+\varepsilon_i).y_i + sgn(y_i).(x_i+\varepsilon_i)\Big)\end{aligned} \tag{4.17}$$

where $\mathbf{x}$ is the input vector, $\mathbf{y}$ can be considered as the system parameters and ε is additive noise to input vector. Looking at (4.17), we can see that the greatest effect that ϵ_i can have is on term $sgn(x_i+\varepsilon_i)$, if $|\varepsilon_i| > |x_i|$ and $sgn(\varepsilon_i) \neq sgn(x_i)$ then it will result in total sign inversion, otherwise, the sign will be reserved and epsilon will affect the amplitude as demonstrated in Table 4.1

Table 4.1: Additive noise impact on Operator $\oplus_s$

case	response	absolute error
$\lvert x_i\rvert > \lvert\varepsilon_i\rvert$	$sgn(x_i.y_i)\big(\lvert x_i+\varepsilon_i\rvert + \lvert y_i\rvert\big)$	$\big\lvert \lvert x_i+\varepsilon_i\rvert - \lvert x_i\rvert \big\rvert = \lvert\varepsilon_i\rvert$
$\lvert x_i\rvert < \lvert\varepsilon_i\rvert \quad sgn(x_i) = sgn(\varepsilon_i)$	$sgn(x_i.y_i)\big(\lvert x_i+\varepsilon_i\rvert + \lvert y_i\rvert\big)$	$\big\lvert \lvert x_i+\varepsilon_i\rvert - \lvert x_i\rvert \big\rvert = \lvert\varepsilon_i\rvert$
$\lvert x_i\rvert < \lvert\varepsilon_i\rvert \quad sgn(x_i) \neq sgn(\varepsilon_i)$	$-sgn(x_i.y_i)\big(\lvert x_i+\varepsilon_i\rvert + \lvert y_i\rvert\big)$	$\lvert x_i\rvert + \lvert x_i+\varepsilon_i\rvert + 2\lvert y_i\rvert$

As it can be seen from Table 4.1, the absolute error in the first two cases does not depend on the system parameters $\mathbf{y}$ contrary to multiplication, where absolute error is $|\varepsilon_i y_i|$. if the variance of $\boldsymbol{\varepsilon}$ is smaller than that of $\mathbf{y}$ then the relative error of $\oplus$ will be smaller than that of normal multiplication for the first two cases in Table 4.1.

4.3 AddNet: Neural Network based on Operator $\oplus$

In this section, we describe Our non-Euclidean based neural nets, namely AddNets. AddNets are feedforwarding neural nets in which dot product in some or all neural nets are replaced by operator $\oplus$ as it can be seen from Fig. 4.1.

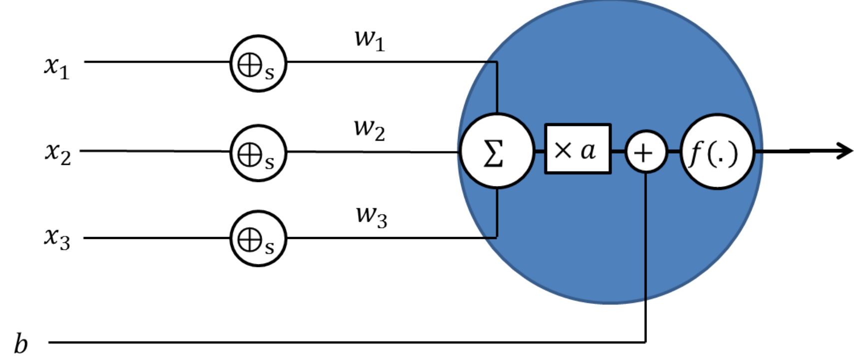


Figure 4.1: Visualization of Operator $\oplus$ based neuron, where $\sum$ is the accumulation of the "weighted" input, a is the scaling factor and $f(.)$ is a non-linear activation

Neurons are replaced on a layer level. The scaling factor, which will be called

multiplicative bias, is important so as to train the network as shown by the experimental results. Since operator $\oplus$ is non-linear by itself, applying non-linearity to activation is not essential. The (additive) bias term is also applied so that the neuron is not restricted to output of 0 when its input is **0**. However, since $x \oplus_s 0^+ = x$ and $x \oplus 0^- = -x$, bias becomes of importance for cases where both input and weights are both close to zero.

4.3.1 AddNet: Feedforwarding Pass

Notations in this section follow the those adopted in Sec. 2.2.1 and Sec. 2.3.1. Based on the description above, we can mathematically express pre-activation $\mathbf{v}^l$ for dense (fully connected) additive-neurons layer as follows:

$$\begin{aligned} v_j^l &:= a_j^l \sum_{i=1}^{I} w_{ij}^l \oplus_s u_i^{l-1} + b_j^l \\ &:= a_j^l \mathbf{w}_j^T \oplus \mathbf{u}^{l-1} + b_j^l \end{aligned} \tag{4.18}$$

where $\mathbf{w}_j$ is the jth column of matrix $\mathbf{W}^l$, subsequently, activation $\mathbf{u}^l$ is written as follows:

$$\begin{aligned} u_j^l := f(v_j^l) &= f\Big(a_j^l\big(\sum_{i=1}^{I} w_{ij}^l \oplus_s u_i^{l-1}\big) + b_j^l\Big) \\ &:= f\Big(a_j^l\big(\sum_{i=1}^{I} w_{ij}^l sgn(u_i^{l-1}) + \sum_{i=1}^{I} sgn(w_{ij}^l) u_i^{l-1}\big) + b_j^l\Big) \end{aligned} \tag{4.19}$$

In matrix notation, vector $\mathbf{u}^l$ can be expressed as follows:

$$\mathbf{u^l} := f(\mathbf{v}^l) = f\Big(\mathbf{a}^l \circ \big(sgn\big(\mathbf{W}^{l^T}\big)\mathbf{u}^{l-1} + \mathbf{W}^{l^T} sgn\big(\mathbf{u}^{l-1}\big)\big) + \mathbf{b}^l\Big) \tag{4.20}$$

where $sgn(.)$ is the element-wise application of signum function over a tensor ($\mathbf{W}^l$ and $\mathbf{u}^{l-1}$ in (4.20)), $\mathbf{a}^l$ is the multiplicative bias vector with dimensionality equal to that of $\mathbf{u}^l$ and $\mathbf{v}^l$, $\circ$ is Hadamard (element-wise) multiplication carried between the multiplicative bias $\mathbf{a}^l$ and the output of operator $\oplus$.

Since operator $\oplus$ is non-linear, one can set $f(.)$ to identity activation ($f(x) = x$), in which case vector $\mathbf{u}^l \equiv \mathbf{v}^l$. Therefore, a model with three layers with the 2nd

layer based on $\oplus$ and with identity activation serves as an ordinary model with one hidden layer with non-linear activation, albeit the non-linearity applied is weight-input dependent. It is worth mentioning that identity mapping is also used in many architectures such as residual neural networks [70].

Likewise, we can define feedforward pass for convolutional layers based on operator $\oplus$ as follows:

$$\begin{aligned}\mathbf{V}_k^l := \mathbf{A}_k^l \circ \Big(& Conv2D\big(\mathbf{W}_k^l, sgn(\mathbf{U}^{l-1})\big) \\ & + Conv2D\big(sgn(\mathbf{W}_k^l), \mathbf{U}^{l-1}\big)\Big) + b_k^l\end{aligned} \tag{4.21}$$

In scalar notation, 4.21 becomes:

$$\begin{aligned}v^l(x,y,k) := a^l(x,y,k)\Big(& \sum_{d=1}^{D}\sum_{j\in J}\sum_{i\in I} w_k^l(i,j,d)sgn\big(u^{l-1}(x-i,y-j,d)\big) \\ & + \sum_{d=1}^{D}\sum_{j\in J}\sum_{i\in I} sgn\big(w_k^l(i,j,d)\big)u^{l-1}\big(x-i,y-j,d\big)\Big) + b_k^l\end{aligned} \tag{4.22}$$

Although (4.21) and (4.22) imply $\mathbf{A}^l$ be a rank-3 tensor whose size is identical to that of $\mathbf{V}^l$, this would be the most generic case. Indeed, $\mathbf{A}^l$ can be of any appropriate dimensionality $\leq \mathbf{X}^l \times \mathbf{Y}^l \times \mathbf{Z^l}$, where $\mathbf{X}^l$, $\mathbf{Y}^l$ and $\mathbf{Z}^l$ are the dimensions of tensor $\mathbf{V}^l$.

For most cases, we chose $\mathbf{A}^l \in \mathbb{R}^k$, i.e. a vector, where k is the depth of pre-activation $\mathbf{V}$, which is identical to the number of filters in the filter bank $\mathbf{W}^l$. The motivation is to regularize each feature map $\mathbf{V}_k^l$ (a rank-2 tensor) on its own. In this case, $a^l(x,y,k) \equiv a_k^l$ and each feature map will share a single multiplicative bias, just as it shares a scalar additive bias. Broadcasting is used to perform the element-wise multiplication in (4.21).

4.3.2 Importance of Multiplicative Bias

With multiplicative bias set to $\mathbf{1}$, models with $\oplus$ layers, either dense or convolutional, are not able to learn and loss does not decrease. This can be attributed to two reason.

The first reason is that the output of operator $\oplus$ is prohibitively large, due to its

the additive nature, and thus a scaling scheme is needed to control the range of its response, in contrast to dot product operations, as in the case of conventional layers, which are naturally scaling (since multiplication itself is scaling).
To demonstrate this idea, we can consider two three-layer models, the former is dot-product based and the second is $\oplus$-based. For both models, we can assume that the 1st layer is the input (presentation layer). Furthermore, we can assume that input $\mathbf{x}$ is N-dimensional, preprocessed such that it is zero-mean and with variance σ_x^2. Weights $\mathbf{W}^1 \in \mathbb{R}^{M\times N}$ and $\mathbf{W}^2 \in \mathbb{R}^{1\times M}$ are initialized as: $w^1 \in \mathbf{W}^1 \sim \mathcal{N}(0,\sigma_1)$ and $w^2 \in \mathbf{W}^2 \sim \mathcal{N}(0,\sigma_2)$, biases are set to zero initially. Furthermore, let $\mathbf{u}^1$ and u^2 be the activations of the hidden layer and the output neuron, respectively. The bias $\mathbf{b}$ is initially set to zero. Before training, the weights are totally random and independent from input vectors $\mathbf{x}$.
For normal (multiplicative) neurons, we can write feed-forwarding as follows:

$$\left(u_i^1\right)_{MULT} = f\left(v_i^1\right)_{MULT} = f\left(\mathbf{w}_i^{1T}\mathbf{x} + b_i^1\right) \tag{4.23}$$

$$\left(u^2\right)_{MULT} = f\left(v_i^1\right)_{MULT} = \mathbf{w}^{2T}\mathbf{u}^1 + b^2 \tag{4.24}$$

where the subscript "MULT" denotes multiplicative dot-product-base connections and T denotes transpose of the column weight vectors. We assume that no activation is applied in the last layer. Likewise, we can write feedforwarding passes for AddNet as follows:

$$\left(u_i^1\right)_{ADD} = f\left(v_i^1\right)_{ADD} = f\left(a_i^1\mathbf{w}_i^{1T} \oplus \mathbf{x} + b_i^1\right) \tag{4.25}$$

$$\left(u^2\right)_{ADD} = f\left(v_i^1\right)_{ADD} = \mathbf{w}^{2T} \oplus \mathbf{u}^1 + b^2 \tag{4.26}$$

where the subscript "ADD" denotes additive $\oplus$-based connections. The mean and variance for ${v_i^1}_{MULT}$ are:

$$mean(v_i^1) = E(v_i^1) = 0 \tag{4.27a}$$

$$var(v_i^1) = E({v_i^1}^2) = N\sigma_w^2\sigma_x^2 \tag{4.27b}$$

In the case of ${v_i^1}_{ADD}$ with $\mathbf{a}^1$ set to an all-ones vector, the mean and the variance are:

$$mean(v_i^1) = E(v_i^1) = 0 \tag{4.28a}$$

$$var(v_i^1) = E({v_i^1}^2) = N\sigma_w^2 + N\sigma_x^2 + 2N\sqrt{\frac{2}{\pi}}\gamma(\mathbf{x})\sigma_w \tag{4.28b}$$

where the term $\sqrt{\frac{2}{\pi}}\sigma_w$ arises from the expectation of $|w|$'s, which have a p.d.f of "Half-Folded Normal Distribution"; a special case of the so-call "Folded Normal Distribution" [64], where the mean of the random variables themselves is zero.
$\gamma(\mathbf{x})$ is a term dependent upon input x and it comes from the expectation of the ℓ_1 norm of the input vector, i.e. $\gamma(\mathbf{x}) = E(||\mathbf{x}||_1)$. This requires further knowledge about the statistics of the input. Nevertheless, it is strictly positive. If we assume that the input vector is i.i.d with p.d.f from each point, (4.28b) becomes:

$$var(v_i^1) = E({v_i^1}^2) = N\sigma_w^2 + N\sigma_x^2 + N\frac{4}{\pi}\sigma_w\sigma_x \tag{4.29}$$

However, this assumption can be too bold. Nevertheless, it shows that $E(||\mathbf{x}||_1) = O(\sigma_x)$ in the least.
As it can be seen from (4.27b) and (4.28b), the variance of the pre-activation layer in the operator $\oplus$ based model is always greater than that of the input, i.e. σ_x^2, in contrast with the normal case, where it is controlled by the multiplicative term σ_w^2, for which an appropriate weight initialization scheme such as: Xavier initialization [71] or He initialization [72] can help control the range of the pre-activation in order to prevent vanishing-exploding back-propagated gradient [44].
Even in the case of very deep models, ReLU can lead to explosion in feedforward with poor weight initialization conditions [72]. In both schemes, weights are initialized so that $var(v_i^1)$ in Xavier initialization and $var(u_i^1)$ in He initialization are set to unity w.r.t input variance, with the difference that He initialization takes into account ReLU activation. The activations are as follows, for Xavier and He initialization, respectively:

$$w_{Xavier}^l \sim U\left[-\frac{\sqrt{6}}{\sqrt{N^l + N^{l+1}}}, \frac{\sqrt{6}}{\sqrt{N^l + N^{l+1}}}\right] \tag{4.30}$$

$$w_{HE}^l \sim U\left[-\frac{2}{\sqrt{N^l}}, \frac{2}{\sqrt{N^l}}\right] \tag{4.31}$$

where U denotes uniform distribution.
Nonetheless, in AddNets, pre-activations are not linearly dependent upon the

statistics of the weights, therefore, we cannot control the variance by merely controlling the weights, as the term $N\sigma_x^2$ still does not depend on the weights. In order to overcome this, pre-activations $\mathbf{v}$ need to be explicitly normalized element-wise, in which case, (4.28b) becomes:

$$var(v_i^1) = E({a_i}^2)\Big(N\sigma_w^2 + N\sigma_x^2 + 2N\sqrt{\frac{2}{\pi}}\gamma(\mathbf{x})\sigma_w\Big) \tag{4.32}$$

with appropriate choices of a_i, we can control the variance such that it is in order of that of normal case.

Note that the analysis is also valid for convolutional layers. The second reason is directly related to backpropagation, which is explained in Sec. 4.3.3.

4.3.3 Backpropagation Pass

Studying the nature of backpropagated cost sensitivities is important so as to determine whether gradient-based learning is feasible. Firstly, the partial derivatives of the scalar operator $\oplus_s$, which account for each input-weight connection are:

$$\frac{\partial x \oplus_s w}{\partial w} \equiv \frac{\partial\big(sgn(w)x + sgn(x)w\big)}{\partial w} = sgn(x) + 2x\delta(w) \tag{4.33a}$$

$$\frac{\partial x \oplus_s w}{\partial x} \equiv \frac{\partial\big(sgn(w)x + sgn(x)w\big)}{\partial x} = sgn(w) + 2w\delta(x) \tag{4.33b}$$

where $\delta(.)$ is the Dirac-delta function (not to be confused with $\boldsymbol{\delta}$, the vector of error sensitivity that is back propagated), which arises from the discontinuity of signum function at zero. Since $\delta(.)$ is zero almost everywhere, it is not suited to gradient calculation, therefore it can be omitted from (4.33a) and (4.33b), th simplified partial derivatives are:

$$\frac{\partial x \oplus_s w}{\partial w} \approx sgn(x) \tag{4.34a}$$

$$\frac{\partial x \oplus_s w}{\partial x} \approx sgn(w) \tag{4.34b}$$

As in Sec 2.2.1, $\boldsymbol{\delta}^l := \frac{\partial J}{\partial \mathbf{b}^l}$, is the sensitivity propagated to layer l from the upper layer, with layer L being the top (output) layer, down to layer 2, which is the first hidden layer. J is the cost function of input-label tuple $(\mathbf{x}, t)$, where t is the

true label associated with input $\mathbf{x}$.
Since at the least the highest layer L is multiplicative, the cost sensitivities $\boldsymbol{\delta}$'s are back-propagated through dot-product-based layers based on the recursive formula (2.16) and calculating weights gradients of these layers follow (2.33), just as in normal neural nets.
As for operator-$\oplus$ based lower layers, The recursive formula (2.16) becomes then:

$$\begin{aligned}\boldsymbol{\delta}^l &:= \frac{\partial J}{\partial \mathbf{b}^l} \equiv \frac{\partial J}{\partial \mathbf{v}^l} \\ &= sgn(\mathbf{W}^{l+1})(\boldsymbol{\delta}^{l+1} \circ \mathbf{a}^{l+1}) \circ f'(\mathbf{v}^l)\end{aligned} \tag{4.35}$$

where $\mathbf{W}^{l+1}$ is the weight matrix associated with layer $l+1$. $\circ$ denotes Hadamard product, $f'(.)$ is the derivative of activation function. In the case of identity activation, the last term in (4.35) is omitted. Note that the multiplication between matrix $sgn(W^{l+1})$ and vector $\boldsymbol{\delta}^{l+1} \circ \mathbf{a}^{l+1}$ is ordinary matrix multiplication. The derivation is carried out under the definitions of partial derivatives for operator $\oplus_s$ in (4.34b), i.e. omitting the resulting Dirac-delta term.
Based on (4.35) and the partial derivative definition in (4.33b), the Cost gradient w.r.t to weight matrix $\mathbf{W}^l$ can be realized using calculus chain rule. Nevertheless, $\mathbf{a}^l$ can have direct dependence upon the weights $\mathbf{W}^l$ in feedforwarding, and thus ut should be taken into account when deriving the Cost gradient of the weights, therefore the chain rule for the gradient is

$$\nabla_{\mathbf{W}^l} J = sgn(\mathbf{u}^{l-1})(\boldsymbol{\delta}^l \circ \mathbf{a}^l)^T + rep(\nabla_{\mathbf{a}^1} J, col)^T \circ \left\{ \frac{\partial \mathbf{a}^l}{\partial \mathbf{W}^l} \right\}_{iij} \tag{4.36}$$

where in (4.36) $rep(., col)$ denotes repetition of the subject vector column wise, $\frac{\partial \mathbf{a}^l}{\partial \mathbf{W}^l}$ is a rank-3 tensor arising from deriving vector $\mathbf{a}^l$ w.r.t $\mathbf{W}^l$, $\left\{ \frac{\partial \mathbf{a}^l}{\partial \mathbf{W}^l} \right\}_{iij}$ is the rank-2 'diagonal' sub-tensor (matrix) with sub-indexing equal to iij, where i and j are $\in \{1, 2, ..., N\}$ and $\in \{1, 2, ..., M\}$, respectively, where N and M are the dimensions of layer $l-1$ and layer l responses, respectively. The quantity $\nabla_{\mathbf{a}^l} J$ is the cost function sensitivity, which is a vector quantity also realized using calculus chain rule as follows:

$$\nabla_{\mathbf{a}^l} J = \boldsymbol{\delta}^l \circ (\mathbf{W}^{l-1^T} sgn(\mathbf{u}^{l-1}) + sgn(\mathbf{W}^{l-1})^T \mathbf{u}^{l-1}) \tag{4.37}$$

The dependency between the weight matrix and the corresponding multiplicative bias impacts the mathematical formulas of backpropagation. When $\mathbf{a}^l$ has no

direct relation with the weight matrix, (4.36) simplifies to:

$$\nabla_{\mathbf{W}^l} J = sgn(\mathbf{u}^{l-1})(\boldsymbol{\delta}^l \circ \mathbf{a}^l)^T \tag{4.38}$$

However, $\mathbf{a}^l$ will still impact the learning process overall. In order to see the importance of multiplicative bias, we can assume that all activations are identity for the sake of the argument as it is a valid activation function in the case of AddNet operator $\oplus$ based layers, furthermore, if the input is zero mean, any pre-activation with identity activation will also have zero-mean. Initially, the variance of δ_i^l can be found as follows:

$$\begin{aligned} var(\delta_i^l) = E({\delta_i^l}^2) &= E\Big(\big(\sum_k sgn(w_{ik})\delta_k^{l+1} a_k^{l+1}\big)^2\Big) \\ &= E\Big(\sum_k (sgn(w_{ik}))^2 {\delta_k^{l+1}}^2 {a_k^{l+1}}^2\Big) \\ &= \sum_k E({\delta_k^{l+1}}^2) E({a_k^{l+1}}^2) \end{aligned} \tag{4.39}$$

The analysis in (4.39) is drawn from [73] and [72]. We assume that the weights are zero-mean *i.i.d*'s and so are the multiplicative biases. Therefore, no co-relation is found between different multiplicative terms. Therefore, the variance is:

$$var(\delta_i^l) = K^l var(\delta^{l+1}) var(a^{l+1}) \tag{4.40}$$

where K is the number of connections from layer l. We can see that the problem arises from the fact that the variance of the signs of the weights is much larger than that of the weights themselves. Without any multiplicative bias, δ will grow exponentially by a factor of:

$$\frac{var(\delta^{l_{sub}})}{var(\delta^{l_{up}})} \propto \prod_{l=l_{sub}}^{l_{up}} K^l \tag{4.41}$$

where l_{up} is the index of the highest additive layer and l_{sub} is the index of the subject layer (a lower layer). This will lead to an exploding gradient in the lower layers.

4.3.4 Choices for Multiplicative Bias

There are plenty of options when it comes to selecting the multiplicative bias that need to account for the constraints mentioned in Sec.4.3.2 and Sec.4.3.3

4.3.4.1 Choice 1: constant

In this case, multiplicative bias $\mathbf{a}$ is chosen to be a constant small enough that it can it does not cause the loss to explode. Potential choices are

$$a^l = \frac{1}{N} \tag{4.42a}$$

$$a^l = \frac{1}{H \times W \times D} \tag{4.42b}$$

Another choice is:

$$a^l = \sigma_{w^l_{init}} \tag{4.43}$$

Finally:

$$\frac{\sigma_{w^l_{init}}}{\sqrt{N}} \tag{4.44a}$$

$$\frac{\sigma_{w^l_{init}}}{\sqrt{H \times W \times D}} \tag{4.44b}$$

Where N is the dimension of the input layer (in case of dense connection), H and W are the kernel width and height, respectively, and D is the kernel depth in the case of convolutional layers. Thus, $H \times W \times D$ is the size of the receptive fields. $\sigma_{w^l_i nit}$ is the standard deviation of the weights before training. Notice that in this case a^l is fixed across a layer l. This can be too limiting for the ability of the network to generalize.

4.3.4.2 Choice 2: Normalized ℓ_1 norm

In this case $\mathbf{a}^l$ is directly dependent upon the weights, and calculated as:

$$a^l_i = \frac{||\mathbf{w}^l_i||_1}{N} \tag{4.45a}$$

$$a_k^l = \frac{||\mathbf{W}_k^l||_1}{H \times W \times D} \tag{4.45b}$$

This normalization choice is the same as the one adopted in [16], where it arises as the least-square-solution of minimizing the Euclidean distance between the weights and their binarization according to this equation:

$$a_{min}^l = argmin(||\mathbf{w}_i^l - a^l sgn(\mathbf{w}^l)||_2^2) \tag{4.46}$$

where $||.||_1$ is the ℓ_1 norm of a vector. Although our operator is non-linear w.r.t input, it is not possible to formulate an input-independent minimization formula. Nevertheless, this choice was worth investigating. In this case, the multiplicative bias vector will be $\mathbf{a}^l \in \mathbb{R}^N$ in the fully connected case and $\mathbf{a}^l \in \mathbb{R}^K$ in the convolutional case, where k is the number of filters in the filter bank, i.e. each feature map $\mathbf{V}_k^l(x, y)$ will be normalized by a scalar a_k^l.

4.3.4.3 Choice 3: Standard Deviation

In this case, $\mathbf{a}^l$ is realized as follows:

$$a_i^l = \sigma(\mathbf{w}_i^l) \tag{4.47a}$$

$$a_k^l = \sigma(\mathbf{W}_k^l) \tag{4.47b}$$

where σ is the standard deviation estimate of the weights. The dimensionality of vector $\mathbf{a}^l$ is the same as in the case of normalized ℓ_1 norm scheme as in Sec. 4.3.4.2.

4.3.4.4 Choice 4: Trainable through Backpropagation

In this case, $\mathbf{a}^l$ is learned through backpropagation and is updated based on the its corresponding gradient given in (4.37). Furthermore, since it is not directly related to the weight matrix or tensor, this opens up for more flexibilty as to the dimensionality of $\mathbf{a}^l$ in convolutional layers since it can range from a vector of size K (as in the above-mentioned choices) to a rank-3 tensor $\mathbf{A}^l$ of size $X \times Y \times K$

where X and Y are the response spatial directions and K is the number of the feature maps or its depth.

As for initializing $\mathbf{a}^l$, reasonable initialization can be $\mathbf{a}^l_{init} = \sigma(\mathbf{w}^l_{init})$.

It is important to note that $\oplus$-based AddNeurons have leverage over Binarized Weight Networks (BWN) devised in [16] in that its response is differentiable w.r.t its input weights as (4.33a) and (4.34a) show, whereas in the case BWN, the derivative w.r.t weight input (without the term $\mathbf{a}^l$ in (4.45a) and (4.45b)) is Dirac-delta, which is practically zero. Therefore, models with binarized weight that are restricted to establishing direct mathematical relation between the normalization term (multiplicative bias) and its associated weights, i.e. $\frac{\partial \mathbf{a}^l}{\partial \mathbf{W}^l} \neq 0$. Therefore, training networks with binarized weights with purely trainable multiplicative bias terms will not be possible using standard backpropagation.

4.3.5 Sparse Operator $\oplus$

Since operator $\oplus_s$ relies on the sign of the weights as well as the weights themselves. One can eliminate the weights that are small enough while keeping their corresponding sign. This can be mathematically expressed as follows:

$$(x \oplus_s w)_{sparse} = \begin{cases} sgn(xw)\big(|x| + |w|\big) & |w| > T \\ sgn(xw)\big(|x|\big) \equiv sgn(w)x & |w| \leq T \end{cases} \tag{4.48}$$

where T is a threshold, consequently we can define the sparse vector operator $\oplus$ as follows:

$$(\mathbf{x} \oplus \mathbf{w})_{sparse} = \sum_i (x_i \oplus w_i)_{sparse} \tag{4.49}$$

In order to select an appropriate thresholding scheme, T is related to the vectors $\mathbf{w}$ in fully connected layers or the rank-3 tensors $\mathbf{W}$ in convolutional layers as follows:

$$T(\mathbf{w}^l_i) = t.max(|\mathbf{w}^l_i|) \tag{4.50a}$$

$$T(\mathbf{W}^l_k) = t.max(|\mathbf{W}^l_k|) \tag{4.50b}$$

where t in (4.50a) and (4.50b) is a scalar ranging from 0 to 1. (4.50a) corresponds to thresholding a weight column vector in fully connected layers, i.e. for weight

matrix $\mathbf{W} \in \mathbb{R}^{M \times N}$ there will be different N thresholds for each N column vector. (4.50b) corresponds to thresholding rank-3 weight tensors in a filter bank $\mathbf{W}^l \in \mathbb{R}^{H \times W \times D \times K}$ such that there are K thresholds for each filter in the filter bank.

It is worth mentioning that when t in (4.50a) and (4.50b) is set to 1. All the weights magnitudes will be set zero, and only the sign information is kept. This, however, reduces to a binarized weight scheme similar to models where weights are binarized after training.

Chapter 5

Experimental Results and Discussion

In this chapter, we report AddNet experimental results over MNIST and CIFAR-10 datasets. In this regard, we investigated many scenarios under different multiplicative bias choices as well as different activation functions. In addition, we studied different cases concerning the scope of applying operator $\oplus$ in the network, i.e. full application in all layers, or partial application in some layers while keeping others multiplicative (dot-product based). Throughout this chapter, we refer to normal dot-product based neural nets as simply **ConvNets**, since all networks studied in this chapter are convolutional. Furthermore, we call addNets with multiplicative hidden layers **hybrid AddNets** and an addNet with all layers (except the output layer) based on operator $\oplus$ **full AddNets**
Taking into account the above-mentioned criteria, we carried out a comparative study regarding loss convergence, achievable test accuracy, effect of sparsity and resilience against salt-and-pepper noise. We implemented our AddNets using Tensorflow 1.3.

5.1 Experiment 1: AddNet over MNIST Dataset

In this experiment, the task is to perform classification over MNIST dataset. MNIST dataset consists of 0-9 digit images with a size of 28×28 Grayscale images. For more details about MNIST dataset, see Sec.2.4.1. The archeticture used in this section is a direct adaptation of Tensorflow example MNIST model. The neural net has two convolutional layers, two max-pooling layers, one dense layer and a weighted layer with 10-way softmax applied over the output neurons. As for the activation function used, we investigated different choices for ReLU, LeakyRelu as well as identity activations. Convolution used in the convolutional layers is of type 'same', which yields a response of the same size as that of the input. For full architectural details, see Table. 5.1.

Table 5.1: MNIST experiment: neural net architecture

Layer	Specifications	Response size
Conv layer 1	Filter bank: 32 5×5 filters	$28 \times 28 \times 32$
Max-pool layer 1	2×2 max pooling	$14 \times 14 \times 32$
Conv layer 2	Filter bank: 64 $5 \times 5 \times 32$ filters	$14 \times 14 \times 64$
Max-pool layer 2	2×2 max pooling	$7 \times 7 \times 64$
Fully Connected layer	Input size: $7 \times 7 \times 64$	512
Softmax layer	Input size: 512	10

When considering the original ConvNet: a CNN with conventional dot-product based connections, the numbers of element-wise multiply operations are given in Fig. 5.2. Notice that bias operations are neglected. The number of total $(+, \times)$ operations needed to realize dot products is roughly twice that of multiply, neglecting adding bias.

The variants of Model 1 concern the following factors: the type of operation applied in each layer (either dot product or operator $\oplus$), the activation functions and the choice for multiplicative bias.

Table 5.2: MNIST experiment: number of elements-wise multiplication operations

Layer	Number of element-wise multiplication operations
Conv layer 1	$574,592$
Conv layer 2	$8,388,608$
Fully connected layer	$1,605,632$
Softmax layer	$5,120$
Total	$10,873,952$

In order to constrain the choices, the same batch size is used throughout experimentation, that is 64 samples per batch. The cost function used is cross-entropy with softmax logits. The optimizer used was Momentum optimizer, with exponentially decaying weight update factor with initial rate = 0.01 and a decaying factor = 0.95. The momentum rate was 0.9. The number of training epochs was 15. Dropout of rate 50% was applied during training.

5.1.1 Case 0: Ordinary ConvNet

This is the original neural net with all of its input-weight connections are dot-product based, i.e. multiplicative. According to Tensorflow website, the test classification accuracy of this model is 99.18%, which equals the accuracy that we were able to obtain. Furthermore, ℓ_2 regularization term for the fully connected weights and biases is added to the loss is added to the total loss. The regularization factor is 5×10^{-4}.
In inference (testing) phase, we applied sparsity over the convolutional weights using the following criterion:

$$S(w;T) = \begin{cases} 0 & |w| \leq T \\ w & otherwise \end{cases} \tag{5.1}$$

where T is the normalized sparsity level as introduced in (4.50a) and(4.50b). Applying sparsity, however, is different from the scheme of $\oplus_{sparse}$ defined in (4.49). However, it can be indicative as to how sparsity-tolerant the network is. The results are given in Table 5.3.

Table 5.3: MNIST experiment: ConvNet classification error for the normal case against different normalized levels of sparsity. Classification error is in percent. Suppressed weights correspond to the percentage of the weights below the sparsity level.

Sparsity level	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8
Classification error	0.82	0.81	0.84	1.00	2.01	5.20	25.28	54.15	71.86
Conv 1 suppressed weights %	0.0	13.9	26.8	43.6	56.5	68.6	78.6	84.6	89.8
Conv 2 suppressed weights %	0.0	20.7	40.3	57.6	71.5	82.6	90.3	95.4	98.4

As it can be seen from Table 5.3, the best classification accuracy occurs at a sparsity level = 0.1. Since weight suppression is a deterministic process, this means that some weights contribute negatively to the overall classification process. Furthermore, more weights get suppressed in the second convolutional layers (20.7%) than in the first convolutional layer (13.9%), which means that in the second layer, there are fewer weights with remarkably larger values.

As for different choices for ReLU function, we investigated using LeakyRelu defined as $LeakyReLU(x;\epsilon) = max(\frac{x}{\epsilon}, x)$, where ϵ is the leakage factor. LeakyRelu is only applied in realising the convolutional responses. The leakage factors investigated are 3 and 5, with test classification errors w.r.t different levels of sparsity given in Table 5.4. As it can be seen, LeakyRelu's persistently give higher classification accuracy that normal ReLU activations.

Table 5.4: MNIST experiment: ConvNet classification error in percent with different sparsity choices

Activation Type	Sparsity levels								
	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8
Relu	0.82	0.81	0.84	1.00	2.01	5.20	25.28	54.15	71.86
LeakyRelu ($\epsilon = 5$)	0.78	**0.74**	**0.74**	0.95	1.32	2.01	6.14	21.89	46.32
LeakyRelu ($\epsilon = 3$)	0.78	0.76	0.87	1.05	1.63	3.08	8.89	31.13	41.82

5.1.2 Case 1: AddNet with Constant Multiplicative Bias

In this case, we replace the dot-product based two convolutional layers with operator $\oplus$-based connections, while keeping the fully connected layer and the output layer multiplicative (dot-produce based). According to Table 5.2, this utilizes roughly 90% of the add-multiply operations in this model.
As for the choices of the multiplicative bias, we investigated the choices mentioned in 4.3.4.1. The results are in Table 5.5. Note that these are the best results obtained across different trials. The convergence of the cost function some cases is given in Fig. 5.1. Note that the loss presented in 5.1 corresponds to the same trials as those reported in Table 5.5.

Table 5.5: MNIST experiment: AddNet classification error in percent for different choices for multiplicative bias a^l

Choice	a^l values	Cassification error
$a^l = \sigma_{w^l{}_{init}}$	$a^1 = a^2 = 0.1$	0.87% (ReLU)* 1.18% (Identity)
$a^l = \frac{1}{H \times W \times D}$	$a^1 = \frac{1}{5 \times 5}$ $a^2 = \frac{1}{5 \times 5 \times 32}$	4.22% (ReLU)†
$a^l = \frac{\sigma_{w^l{}_{init}}}{\sqrt{H \times W \times D}}$	$a^1 = \frac{0.1}{\sqrt{5 \times 5}}$ $a^2 = \frac{0.1}{\sqrt{5 \times 5 \times 32}}$	8.87% (ReLU)$^\diamond$ 3.11% (Identity)

It is important to note that when $a^l = \frac{\sigma_{w^l{}_{init}}}{\sqrt{H \times W \times D}}$ (last row in Table 5.5) the cost function does not always converge. In fact, with *ReLU* activation, in more than 50% of the cases loss convergence does not occur, at least within 15 epochs.

5.1.3 Case 2: AddNet with Normalized ℓ_1 Norm Multiplicative Bias

The choice in this case was to set a^l as in Sec. 4.3.4.2. In this regard, we studied a hybrid AddNet a full AddNet. Test classification results are in Table 5.7. As

Table 5.6: MNIST experiment: classification accuracy results for hybrid AddNet with constant multiplicative bias (case $*$ in Table 5.5) w.r.t different sparsity levels. Affected weights are those whose magnitudes are suppressed and only their signs are kept.

Sparsity level	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
Classification error %	**0.87**	**0.87**	0.90	0.92	**0.89**	0.90	0.95	0.93	0.95	0.97	1.20
Conv 1 affected weights %	0	21.89	37.0	49.3	62.3	73.3	82.1	87.0	91.0	94.0	100.0
Conv 2 affected weights %	0	18.1	35.1	50.7	64.3	75.6	84.4	91.1	95.9	98.8	100.0

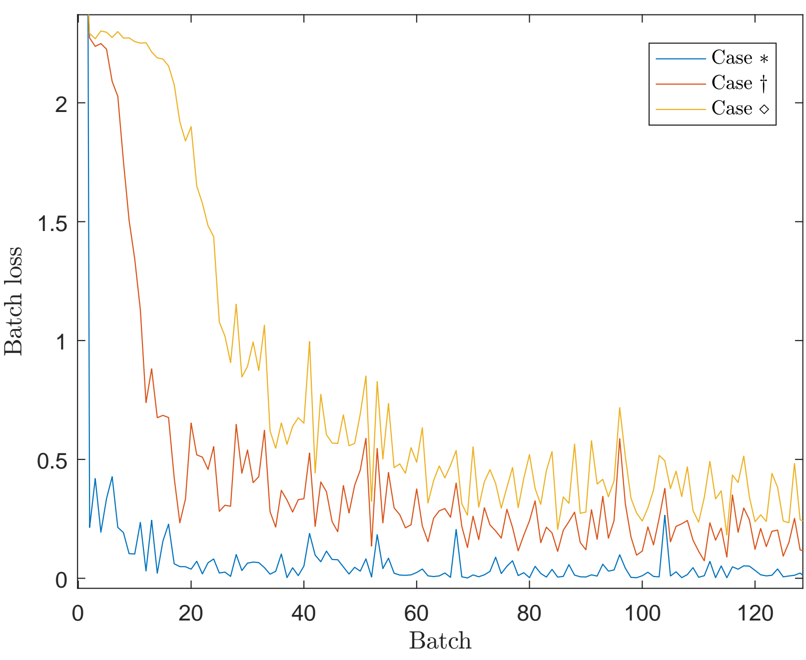


Figure 5.1: MNIST experiment: cost convergence for cases with constant multiplicative bias. Refer to Table 5.5 with the corresponding symbol for full description of the case.

for cost function convergence, its rate was almost similar for different activation choices and there was no noticeable difference.

Table 5.7: MNIST experiment: AddNet classification error in percent with ℓ_1 norm-based multiplicative bias with different activation choices.

Activation type	Classification error %
ReLU	1.17
LeakyReLU ($\epsilon = 5$)	0.89
LeakyReLU ($\epsilon = 3$)	0.91
Identity	1.17

Table 5.8: MNIST experiment: AddNet classification error in percent with $\ell 1$-based multiplicative bias for different sparsity levels

Activation type	sparsity levels										
	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
LeakyReLU ($\epsilon = 5$)	0.89	0.90	0.92	**0.86**	0.90	1.06	1.16	1.24	1.41	1.55	1.57
LeakyReLU ($\epsilon = 3$)	0.91	0.93	0.90	0.98	0.96	0.97	1.03	1.02	1.07	1.17	1.58

Regarding the case of full AddNet, we tried two networks with two different activation choices: ReLU and Identity. The test classification errors over MNIST were 1.08% and 1.13%, respectively.

5.1.4 Case 3: AddNet with Standard Deviation Based Multiplicative Bias

The choice for a^l in this case is the standard deviation estimate of the weights as in Sec. 4.3.4.3. In this regard, we studied using ReLU activation and Identity activation in a hybrid model. The test classification errors were 1.03% and 1.06%, respectively. Table 5.9 shows the results w.r.t different sparsity levels.

Table 5.9: MNIST experiment: AddNet classification error in percent with standard deviation-based multiplicative bias for different sparsity levels

activation type	sparsity levels										
	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
ReLU	1.03	1.02	1.05	1.05	1.04	1.02	1.13	1.10	1.18	1.22	1.35
Identity	1.06	1.06	1.09	1.11	1.12	1.11	1.18	1.18	1.13	1.20	1.31

5.1.5 Case 4: AddNet with Trainable Multiplicative Bias

The choice here is to set $\mathbf{a}^l$ as a trainable variable in the computational graph with initialization $a^l_{kinit} = \sigma_{winit}$. We investigated two hybrid models with ReLU and Identity activation. The results are in Table 5.10.

Table 5.10: MNIST Experiment: AddNet classification error in percent with a trainable multiplicative bias for different sparsity levels

activation type	Sparsity levels										
	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
ReLU	0.96	0.97	1.04	1.02	1.09	1.09	1.05	1.15	1.25	1.31	1.47
Identity	1.44	1.41	1.47	1.39	1.47	1.67	1.70	1.66	1.77	1.83	2.30

5.1.6 Case 5: Binarized Weights Network

We also compared our AddNet with BWN [16], where the weights are binarized during training and in order to use standard back propagation, a^l is introduced as $\frac{||\mathbf{w}^1||_1}{H\times W\times D}$. Therefore, we compare our AddNet with the same ℓ_1 norm choice for multiplicative bias. In order to insure consistency , we implemented hybrid BWN (the fully connected layer is not binarized) with different activation choices. The comparative results are given in Table 5.11.

As it can be seen from Table 5.11, hybrid AddNet outperforms BWN for all activation choices. Identity is not a valid option for BWNs since they are linear input-weight regimes. Although BWN beats AddNet when weights are

Table 5.11: MNIST Experiment: comparison of classification error between BWN and hybrid AddNet with different activation choices

Activation type	network type		
	Hybrid BWN	Hybrid AddNet	
		No sparsity	Full sparsity
ReLU	1.21%	1.17%	4.09%
LeakyReLU($\epsilon = 5$)	1.25%	0.89%	1.57%
LeakyReLU($\epsilon = 3$)	1.23%	0.91%	1.58%
Identity	——	1.17%	1.95%

fully sparse, AddNet can achieve better results with partially suppressed weights as shown previously.

From a hardware resource point of view, one major advantage of BWN is that it requires less memory storage. Likewise, in AddNets, weight suppression can lead to memory saving. Nevertheless, AddNet has flexibility between performance requirements and memory resources constraints. In this regard, we can devise a balanced performance-memory score metric for a neural net as follows:

$$score_{NN} := \frac{1}{error \times size} \tag{5.2}$$

where *size* corresponds to the storage size of the weights and *error* is the inference classification error. For notational clarity, let A denote AddNet and B denote BWN. We can define an optimization ratio between AddNet and BWN as follows:

$$ratio_{A/B} := \frac{score_A}{score_B} = \frac{error_B \times size_B}{error_A \times size_A} \tag{5.3}$$

Looking at the optimization ratio defined in (5.3), we can see that if $ratio > 1$, AddNet has an overall better performance-memory score than BWN, and vice versa. As for the network size, the suppressed weights will contribute 1-bit each (assuming binary sign). The fully represented weights, however, will contribute different sizes based on their precision. In any case, we considered 16-bit and 32-bit real-valued unsuppressed weights precision under the assumption that accuracy does not decrease when using 16-bit. We compared the best results for each multiplicative bias scheme in AddNets at different sparsity (suppression) levels against BWN, based on the ratio defined in (5.3). The results are in Fig. 5.2 and Fig. 5.3.

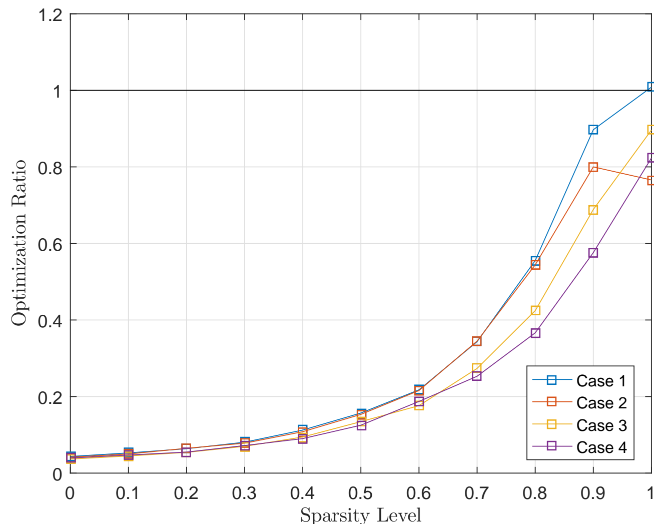


Figure 5.2: MNIST experiment: performance-memory score of AddNets w.r.t BWN at different sparsity levels. Cases 1-4 correspond to: constant, ℓ_1-norm based, standard deviation and trainable multiplicative bias. The real-valued (unsuppressed) weights are 32-bit long.

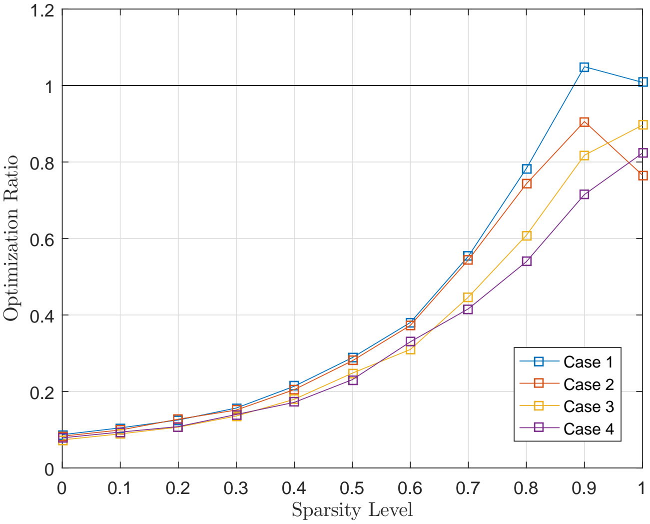


Figure 5.3: MNIST experiment: performance-memory score of AddNets w.r.t BWN at different sparsity levels. Cases 1-4 correspond to: constant, ℓ_1-norm based, standard deviation and trainable multiplicative bias. The real-valued (unsuppressed) weights are 16-bit long.

As it can be seen from Fig. 5.2 and Fig. 5.2, full suppression of the weights yields an optimization ratio equals to 1. Furthermore, with 16-bit representation, and without loss in precision, we could achieve a higher optimization ratio of 1.05 (case 1, sparsity level=0.9). Note that both AddNet and BWN compared are hybrid but the argument holds for other cases.

5.1.7 Performance with Salt-and-Pepper Noise

We also investigated the effect of corrupting the test data with salt-and-pepper noise [27] in order to study the ability of AddNets to generalize in the present of Impulsive data. In this regard, we trained the normal ConvNet and an AddNet with noiseless training data. Then, in the inference phase, we corrupted the test data at various levels and studied the deterioration in the test classification performance.

In the case of Salt-and-Pepper noise, pixel intensities are either kept intact, set to zero or to one randomly. We define SAP level (ρ) as the probability of SAP noise to take place. Therefore, we can write the noising process mathematically as follows:

$$I_{SAP}(x,y) = \begin{cases} I(x,y) & prob = 1-\rho \\ 1 & prob = \rho b \\ 0 & prob = \rho(1-b) \end{cases} \tag{5.4}$$

where in (5.4) I is the original pixel intensity value, ρ is the SAP level, and b is the bias towards setting pixels to 1 (white) or 0 (black). Since MNIST images are mostly black, we set the bias to 0.7 in order to add more "salt" rather than "pepper". Example MNIST samples w.r.t different sap levels are given in Table 5.12.

For this comparative analysis, we compared the normal ConvNet with two hybrid AddNets with trainable multiplicative bias: the former has ReLU activation while the latter has identity activation. The test classification results are given in Table 5.13.

Table 5.12: visualization of example MNIST images with different SAP levels

SAP Level ρ	Example photos
0.05	
0.15	
0.3	
0.7	

As it can be seen from Table 5.13, although SAP persistently deteriorates the performance of any model, at SAP level $\rho = 0.15$, the hybrid Operator $\oplus$ based model with ReLU activation becomes more resilient against salt-and-pepper corruption than the normal dot-based model. The identity-activation based model, however, was the least resilient against salt-and-pepper corruption.

Table 5.13: MNIST Experiment: classification error in percent over SAP-corrupted MNIST dataset at different levels

Model type	SAP levels											
	0	0.05	0.1	0.15	0.2	0.25	0.3	0.35	0.4	0.5	0.6	0.7
ConvNet	0.82	1.02	1.54	2.93	5.02	8.93	14.51	22.73	32.52	51.04	66.50	77.47
hybrid AddNet (ReLU act.)	0.96	1.29	1.57	**2.26**	**3.66**	**5.29**	**8.07**	**11.89**	**17.51**	**32.36**	51.12	67.53
hybrid AddNet (identity act.)	1.44	3.05	4.81	8.11	12.67	18.04	24.85	31.78	39.24	51.88	65.92	76.16

5.2 Experiment 2: AddNet over CIFAR-10 Dataset

In this eperiment, the task is to perform classification over CIFAR-10 dataset, which is far more challenging than MNIST dataset because of the larger variety in patterns in the data that are folded under one class. Therefore, ConvNets should be able to extract more abstract higher-level features. Although models with high classification performance rates comprise too many layers, we selected a rather small model in order to do intensive analysis and compare different variants of AddNeuron with normal convents.
The archeticture used is also an adaptation of Tensorflow example CIFAR-10 model, which consists of two convolutional layers, two max-pooling layers, two fully connected layers and a softmax output layer. In this mode, the input layer takes 28×28 windows of the original 32×32 RGB images. The description of the architecture is given in Table 5.14.
It is important to mention that in the original model, a "Local Response Normalization" [2] Layer is induced after each convolutional layer so as to perform lateral normalization of the response of the convolutional layers. However, we opted to eliminate the local response normalization layer since early investigation proved it unfit for our model and multiplicative bias was applied instead.

Table 5.14: CIFAR Experiment: neural net architecture

Layer	Specifications	Response size
Conv layer 1	filter bank: 64 $5 \times 5 \times 3$ filters	$28 \times 28 \times 64$
Max-pool layer 1	2×2 max pooling	$14 \times 14 \times 64$
Conv layer 2	filter bank: 64 $5 \times 5 \times 64$ filters	$14 \times 14 \times 64$
Max-pool layer 2	2×2 max pooling	$7 \times 7 \times 64$
Fully Connected layer 1	Input size: $7 \times 7 \times 64$	384
Fully Connected layer 2	Input size: 384	192
Softmax layer	Input size: 192	10

The model was trained using batches of size 128. The update rule used is RMSProp [74] with initial learning rate = 10^{-4}, decay factor = 0.9.

Table 5.15: CIFAR Experiment: number of elements-wise multiplication operations

Layer	Number of element-wise multiplication operations
Conv layer 1	$3,447,552$
Conv layer 2	$16,777,216$
Fully connected layer 1	$1,204,224$
Fully connected layer2	$73,728$
Softmax layer	$1,920$
Total	$21,504,620$

We followed the same data augmentation scheme used in training as the one in the original model, which goes as follows: images are randomly cropped with size of 28×28 out of the original 32×32 RGB images. Furthermore, the cropped images are randomly flipped horizontally. Afterwards, brightness and contrast are randomly perturbed. This help augment the data during training so that the model can learn to generalize better. The cost function used is cross-entropy with 10-way softmax.
As it can be seen from Table 5.15, one feedforward pass requires roughly 20 million add-multiply operations, which is twice as computationally demanding as MNIST Model 1. This is because the response of the first convolutional layer comprises 64 feature maps, as opposed to 32 feature maps in MNIST case. Therefore, the computational burden for the 2nd filter banks will be twice. In any case, a hybrid model with Operator $\oplus$ based convolutional layers will optimize roughly 90% of the multiplication operations overall. All activations used in this Model are ReLU.

5.2.1 Case 0: Ordinary ConvNet

In this case, the original model was investigated while keeping local response normalization. The classification accuracy was **86.0**% after almost 60k batches. The cross entropy loss then is around **0.5** (excluding the regularization loss). Without local response normalization, the cost function suffered from divergence. The loss convergence and test accuracy are shown in Fig.5.4.

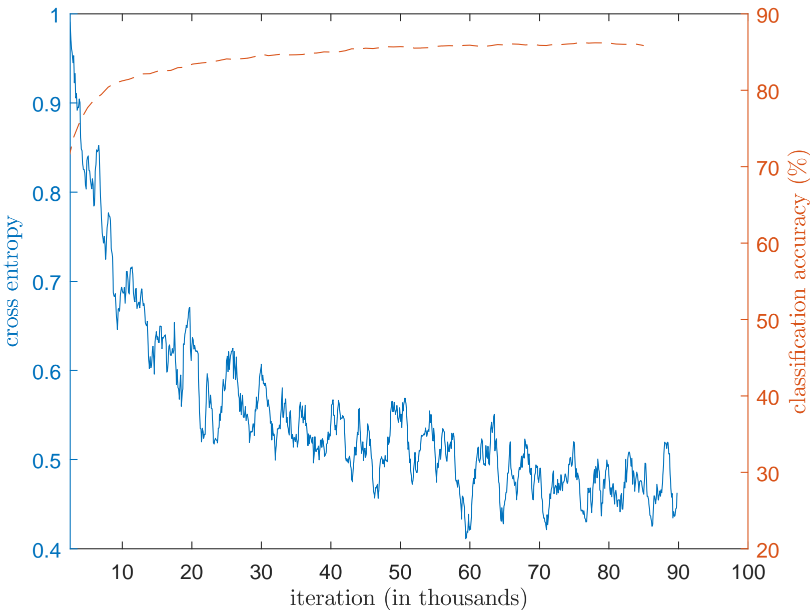


Figure 5.4: CIFAR Experiment: ConvNet loss convergence (solid line) and testing accuracy (dashed line) w.r.t batches

5.2.2 Case 1: AddNet with Constant Multiplicative Bias

Contrary to MNIST experiment, our early investigation showed that using a fixed multiplicative bias did not yield good results. In fact, the accuracy that we were able to achieve did not exceed **70**%. Therefore, we focused on the other normalization schemes.

5.2.3 Case 2: AddNet with Normalized ℓ_1 Norm Multiplicative Bias

In this case we used ℓ_1-based normalization as in Sec. 4.3.4.2. We chose to replace operations in the convolutional layers only, i.e. devising a hybrid model. The highest classification rate was 80.9% which occurred at step 210K as it can be seen from Fig. 5.5. It is worth noting that the model could not do better at classification despite the fact that the cost decrease up to 0.3 after 300k steps.

Loss in 5.5 is smoothed through moving average.

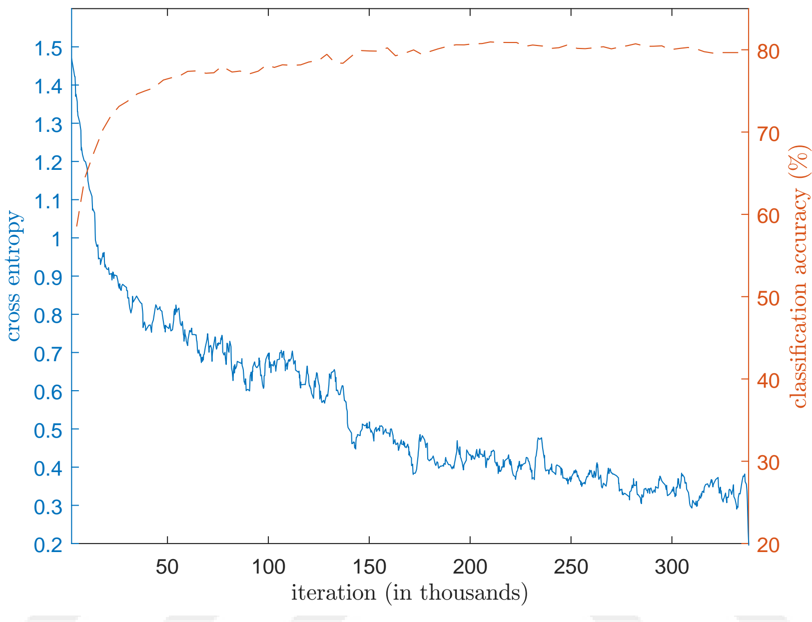


Figure 5.5: CIFAR experiment: AddNet loss convergence (solid line) and testing accuracy (dashed line) w.r.t batches (ℓ_1-norm based multiplicative bias)

5.2.4 Case 3: AddNet with Standard Deviation Based Multiplicative Bias

In this case we used the standard deviation estimate of the weights as in Sec. 4.3.4.3. We investigated a hybrid model as in the previous case. The results are in Fig. 5.6. The highest classification accuracy was 80.1%, which occurred after 245k iterations. This means that the model generalizes a little worse than the case with $\ell 1$ norm based multiplicative bias.

A comparison between the loss convergence and the accuracy of the two models is given in Fig. 5.7, where it shows that $\ell 1$ norm converges faster.

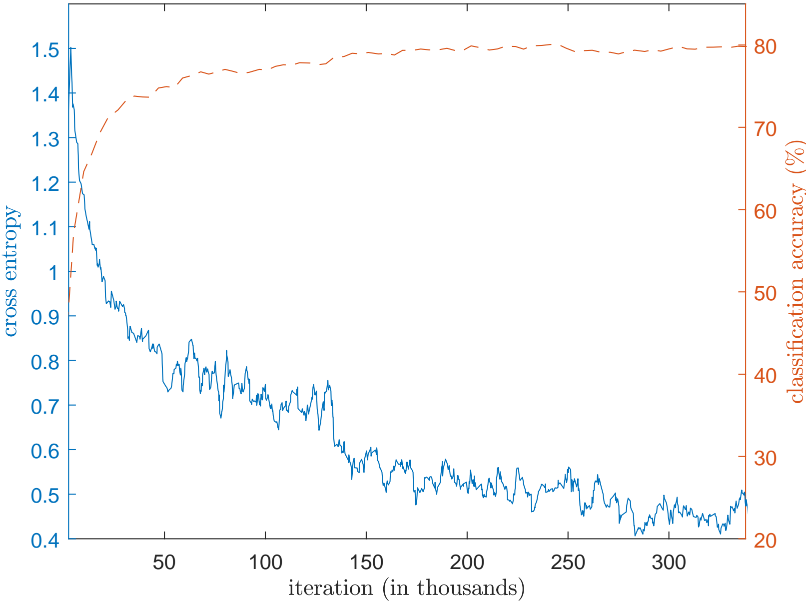


Figure 5.6: CIFAR experiment: AddNet loss convergence (solid line) and testing accuracy (dashed line) w.r.t batches (standard deviation based bias)

5.2.5 Case 4: AddNet with Trainable Multiplicative Bias

In this case, we set $\mathbf{A}^l$ in the convolutional layers as a vector trainable through backpropagation as discussed in Sec. 4.3.4.4. Therefore, $\mathbf{A}^l \equiv \mathbf{a}^l \in \mathbb{R}^{K_l}$, i.e. $\mathbf{a}^l$'s have the same dimensionality as in ℓ_1 norm and standard deviation estimate cases. the results are given in 5.8.

The best accuracy rate was 81.0% which means a trainable $\mathbf{a}^l$ performs as well as ℓ_1 norm based $\mathbf{a}^l$. Convergence and classification accuracy are shown in Table 5.8. Nevertheless, in this case the model reach 81% accuracy faster than in the previous cases (at 185k iterations). Table 5.9 draws a comparison between the convergence of trainable $\mathbf{a}^l$ case and ℓ_1 norm based case.

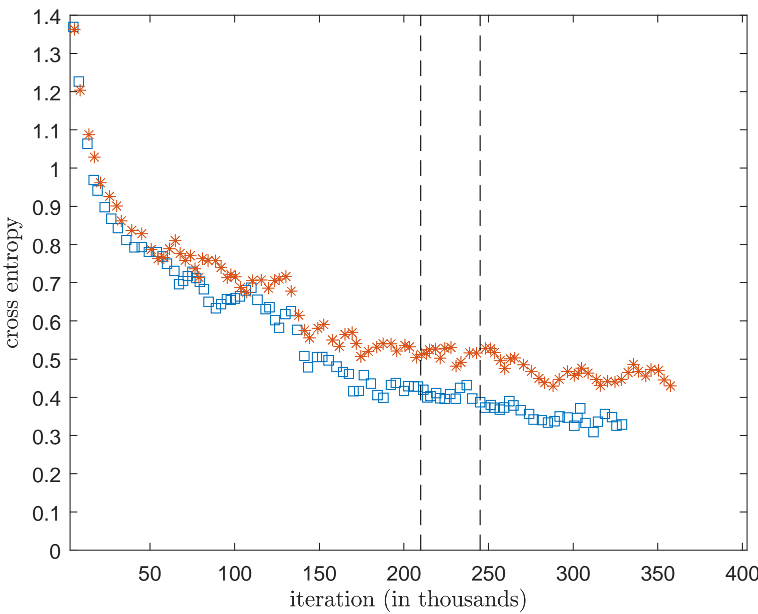


Figure 5.7: CIFAR experiment: comparison between ℓ_1 case convergence (□) and standard deviation case convergence ($*$). The two vertical lines correspond to the points at which the highest accuracy occurred for both cases

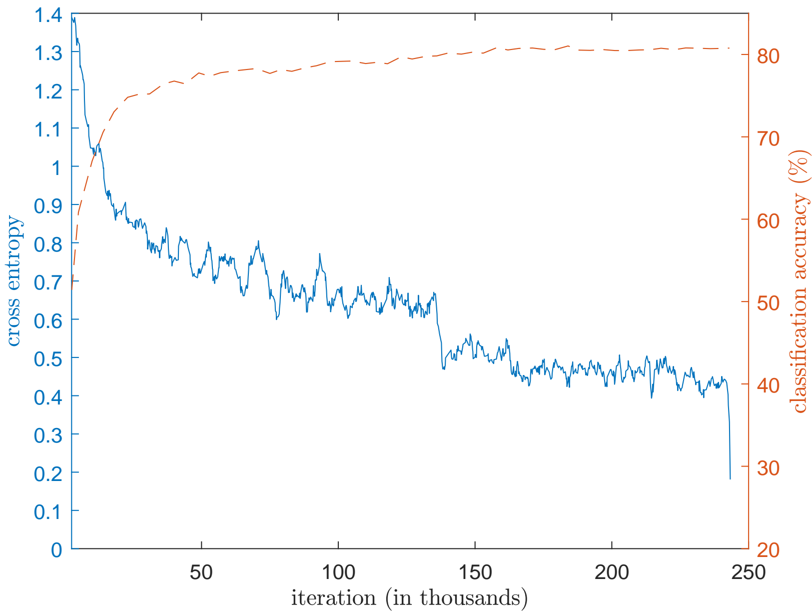


Figure 5.8: Loss convergence (solid line) and testing accuracy (dashed line) w.r.t batches

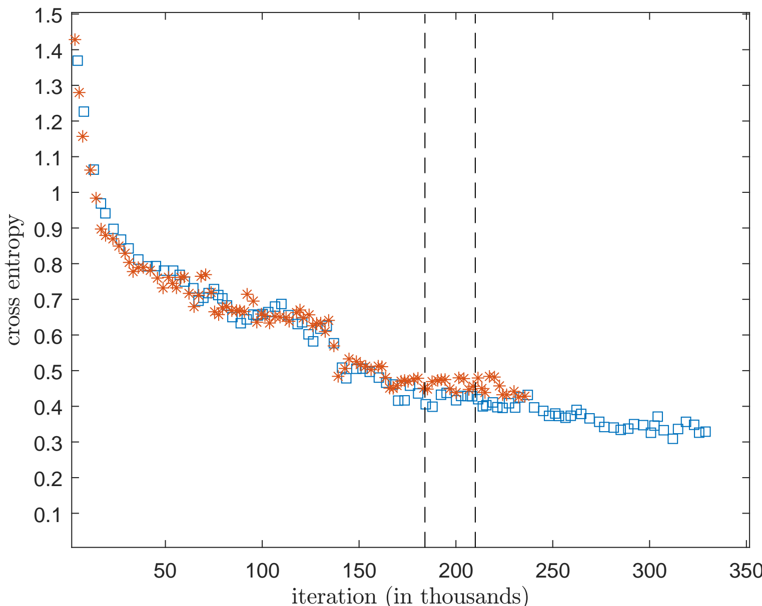


Figure 5.9: CIFAR experiment: comparison between trainable $\mathbf{a}^l$ case convergence ($*$) and ℓ_1 norm based $\mathbf{a}^l$ case convergence ($\square$). The two vertical lines correspond to the points at which the highest accuracy occurred for both cases

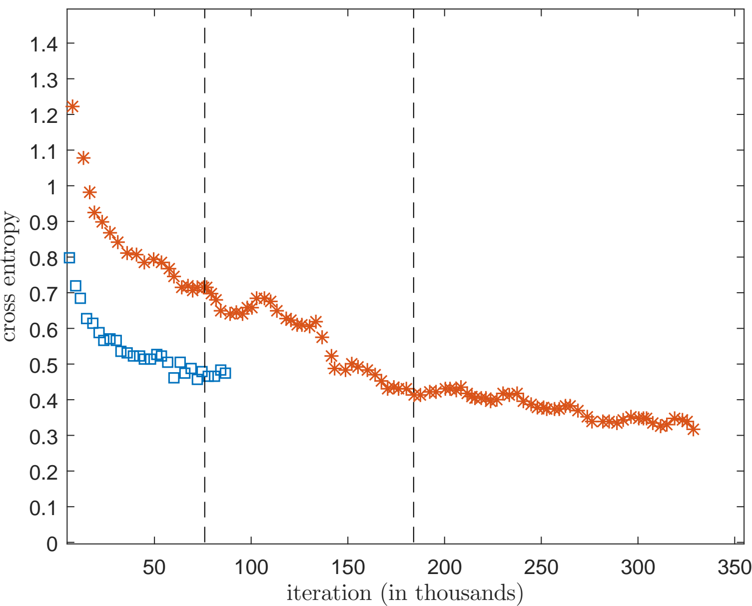


Figure 5.10: CIFAR experiment: comparison between the convergence of ConvNet ($\square$) and AddNet ($*$) case with ℓ_1 based multiplicative bias. The vertical lines correspond to the points in training at which the highest accuracy occurs for both cases.

5.2.6 Performance with Salt-and-Pepper Noise

In this part, we report our results regarding the performance of ConvNet and AddNet under salt-and-pepper data corruption, as we did in our MNIST experiment. We compared the best performing AddNet, which was AddNet with trainable multiplicative bias with ConvNet. Salt-and-pepper (SAP) level (ρ) is the probability of inducing SAP noise on a pixel as demonstrated in (5.4). The difference between noise in this case and the MNIST case, is that noise is unbiased (b is set to 0.5 in (5.4)). Furthermore, since CIFAR images have 3 channels, there are total of 8 possible outcomes of "salt and pepper". Sample images at different SAP levels can be found in Appendix B.

Our investigation showed that both models ConvNet and AddNet are very sensitive to SAP noise. Because of that, we used smaller SAP levels ρ. The performance of AddNet and ConvNet over different SAP levels is given in Table 5.16.

Table 5.16: CIFAR experiment: test classification error in percent for three CIFAR-10 models with SAP-corrupted testing data over different levels

Model type	SAP levels ρ								
	0.0	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08
ConvNet	86.0	74.3	60.5	49.2	40.0	31.9	27.4	24.2	24.2
AddNet	81.0	**74.8**	**66.4**	**58.4**	**50.7**	**43.5**	**37.8**	**34.0**	**29.8**

As it can been seen from Table 5.16, with only a SAP level $\rho = 0.01$, the performance of ConvNet deteriorates by almost 12%. Despite that ConvNet outperforms AddNet over noiseless data by 5%, AddNet outperforms ConvNet over all SAP-corrupt data even at a level as small as $\rho = 0.01$.

Chapter 6

Conclusion

In this work, we introduce AddNet: a convolutional neural network based on an ℓ_1-norm inducing operator, which replaces the dot-product-based operations such as: matrix multiplication and tensor convolution with multiplication-free sign-preserving additive operations. The AddNet structure needs a multiplicative bias in each neuron. Different choices for multiplicative bias are presented.
Our experiments over MNIST dataset show that AddNet performs as well as an ordinary convoultional neural network on MNIST dataset. Our comparison between AddNet and Binarized Weight Network (BWN) shows that addNet achieves better results than BWN in general.
In addition, partial and full weight binarization in AddNets is possible, although it decreases AddNet performance to some extent. Nevertheless, it is possible to find a middle ground between performance and memory constraints with partial binarization. In fact, AddNet could beat the accuracy of BWN over MNIST dataset with 98.5% of AddNet convolutional weights binarized. The loss in accuracy may be mitigated by fine-tuning, which is worth investigating in future work.
Our experiments over CIFAR-10 dataset shows that AddNet with fully trainable multiplicative bias converges the fastest among the other choices, which are fixed constant, ℓ_1-based or standard deviation-based choices. The convergence of AddNet was slower than that of ordinary CNNs and the classification accuracy

was -5% lower than that of the corresponding CNN over CIFAR-10 dataset. Nonetheless, with salt-and-pepper corrupt testing data, AddNet better results than CNN. As a matter of fact, with a salt-and-pepper noise level as small as 0.01, AddNet was able to achieve better accuracy at a rate 74.8% compared to 74.3% of the ordinary CNN. This shows that AddNet possesses ℓ_1-norm properties of resilience against outliers. In the context of deep learning, this mean that AddNets can potentially be more resilient against adversarial attacks [75, 76].

Bibliography

[1] Y. LeCun, K. Kavukcuoglu, and C. Farabet, "Convolutional networks and applications in vision," in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pp. 253–256, IEEE, 2010.

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

[3] Y. Sun, Y. Chen, X. Wang, and X. Tang, "Deep learning face representation by joint identification-verification," in *Advances in neural information processing systems*, pp. 1988–1996, 2014.

[4] B. Graham, "Fractional max-pooling," *arXiv preprint arXiv:1412.6071*, 2014.

[5] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[6] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, "Handwritten digit recognition with a back-propagation network," in *Advances in neural information processing systems*, pp. 396–404, 1990.

[7] A. Krizhevsky and G. Hinton, "Convolutional deep belief networks on cifar-10," *Unpublished manuscript*, vol. 40, 2010.

[8] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, *et al.*, "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, 2012.

[9] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pp. 6645–6649, IEEE, 2013.

[10] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K. J. Lang, "Phoneme recognition using time-delay neural networks," *IEEE transactions on acoustics, speech, and signal processing*, vol. 37, no. 3, pp. 328–339, 1989.

[11] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*, pp. 160–167, ACM, 2008.

[12] J. Devlin, R. Zbib, Z. Huang, T. Lamar, R. M. Schwartz, and J. Makhoul, "Fast and robust neural network joint models for statistical machine translation.," in *ACL (1)*, pp. 1370–1380, 2014.

[13] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, pp. 3104–3112, 2014.

[14] B. Moons, B. De Brabandere, L. Van Gool, and M. Verhelst, "Energy-efficient convnets through approximate computing," in *Applications of Computer Vision (WACV), 2016 IEEE Winter Conference on*, pp. 1–8, IEEE, 2016.

[15] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan, "Axnn: energy-efficient neuromorphic systems using approximate computing," in *Proceedings of the 2014 international symposium on Low power electronics and design*, pp. 27–32, ACM, 2014.

[16] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*, pp. 525–542, Springer, 2016.

[17] M. Kim and P. Smaragdis, "Bitwise neural networks," *arXiv preprint arXiv:1601.06071*, 2016.

[18] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in Neural Information Processing Systems*, pp. 3123–3131, 2015.

[19] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 267–288, 1996.

[20] M. Elad and M. Aharon, "Image denoising via sparse and redundant representations over learned dictionaries," *IEEE Transactions on Image processing*, vol. 15, no. 12, pp. 3736–3745, 2006.

[21] Y. Pang, X. Li, and Y. Yuan, "Robust tensor analysis with l1-norm," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 20, no. 2, pp. 172–178, 2010.

[22] N. Kwak, "Principal component analysis based on l1-norm maximization," *IEEE transactions on pattern analysis and machine intelligence*, vol. 30, no. 9, pp. 1672–1680, 2008.

[23] X. Li, Y. Pang, and Y. Yuan, "L1-norm-based 2dpca," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 40, no. 4, pp. 1170–1175, 2010.

[24] Q. Ke and T. Kanade, "Robust l/sub 1/norm factorization in the presence of outliers and missing data by alternative convex programming," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 1, pp. 739–746, IEEE, 2005.

[25] H. Tuna, I. Onaran, and A. E. Cetin, "Image description using a multiplierless operator," *IEEE Signal Processing Letters*, vol. 16, no. 9, pp. 751–753, 2009.

[26] A. Suhre, F. Keskin, T. Ersahin, R. Cetin-Atalay, R. Ansari, and A. E. Cetin, "A multiplication-free framework for signal processing and applications in biomedical image analysis," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 1123–1127, IEEE, 2013.

[27] R. H. Chan, C.-W. Ho, and M. Nikolova, "Salt-and-pepper noise removal by median-type noise detectors and detail-preserving regularization," *IEEE Transactions on image processing*, vol. 14, no. 10, pp. 1479–1485, 2005.

[28] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[29] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley, and Y. Bengio, "Theano: new features and speed improvements," *arXiv preprint arXiv:1211.5590*, 2012.

[30] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 675–678, ACM, 2014.

[31] S. Haykin, *Neural networks: a comprehensive foundation.* Prentice Hall PTR, 1994.

[32] K.-I. Funahashi, "On the approximate realization of continuous mappings by neural networks," *Neural networks*, vol. 2, no. 3, pp. 183–192, 1989.

[33] B. Widrow and M. A. Lehr, "30 years of adaptive neural networks: perceptron, madaline, and backpropagation," *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1415–1442, 1990.

[34] M. Minsky and S. Papert, "Perceptrons.," 1969.

[35] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals, and Systems (MCSS)*, vol. 2, no. 4, pp. 303–314, 1989.

[36] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," tech. rep., California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[37] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural computation*, vol. 1, no. 2, pp. 270–280, 1989.

[38] A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro, "Robust stochastic approximation approach to stochastic programming," *SIAM Journal on optimization*, vol. 19, no. 4, pp. 1574–1609, 2009.

[39] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[40] Y. LeCun *et al.*, "Generalization and network design strategies," *Connectionism in perspective*, pp. 143–155, 1989.

[41] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[42] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for simplicity: The all convolutional net," *arXiv preprint arXiv:1412.6806*, 2014.

[43] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533 EP –, Oct 1986.

[44] S. Hochreiter, "The vanishing gradient problem during learning recurrent neural nets and problem solutions," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 6, no. 02, pp. 107–116, 1998.

[45] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," in *Advances in neural information processing systems*, pp. 153–160, 2007.

[46] M. A. Hearst, S. T. Dumais, E. Osuna, J. Platt, and B. Scholkopf, "Support vector machines," *IEEE Intelligent Systems and their applications*, vol. 13, no. 4, pp. 18–28, 1998.

[47] P. Y. Simard, D. Steinkraus, J. C. Platt, *et al.*, "Best practices for convolutional neural networks applied to visual document analysis.," in *ICDAR*, vol. 3, pp. 958–962, 2003.

[48] Y. LeCun, C. Cortes, and C. J. Burges, "Mnist handwritten digit database," *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, vol. 2, 2010.

[49] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[50] D. C. Ciresan, U. Meier, L. M. Gambardella, and J. Schmidhuber, "Deep big simple neural nets excel on handwritten digit recognition," 2010.

[51] L. Wan, M. Zeiler, S. Zhang, Y. Le Cun, and R. Fergus, "Regularization of neural networks using dropconnect," in *International Conference on Machine Learning*, pp. 1058–1066, 2013.

[52] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," *arXiv preprint arXiv:1511.07289*, 2015.

[53] D. Mishkin and J. Matas, "All you need is a good init," *arXiv preprint arXiv:1511.06422*, 2015.

[54] V. Sze, Y.-H. Chen, J. Einer, A. Suleiman, and Z. Zhang, "Hardware for machine learning: challenges and opportunities," in *Custom Integrated Circuits Conference (CICC), 2017 IEEE*, pp. 1–8, IEEE, 2017.

[55] S. K. Esser, P. A. Merolla, J. V. Arthur, A. S. Cassidy, R. Appuswamy, A. Andreopoulos, D. J. Berg, J. L. McKinstry, T. Melano, D. R. Barch, *et al.*, "Convolutional networks for fast, energy-efficient neuromorphic computing," *Proceedings of the National Academy of Sciences*, p. 201604850, 2016.

[56] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," *Microsoft Research Whitepaper*, vol. 2, no. 11, 2015.

[57] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "Yodann: An ultra-low power convolutional neural network accelerator based on binary weights," in *VLSI (ISVLSI), 2016 IEEE Computer Society Annual Symposium on*, pp. 236–241, IEEE, 2016.

[58] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, "Neural networks with few multiplications," *arXiv preprint arXiv:1510.03009*, 2015.

[59] F. Li, B. Zhang, and B. Liu, "Ternary weight networks," *arXiv preprint arXiv:1605.04711*, 2016.

[60] M. T. Arslan, A. Bozkurt, R. A. Sevimli, C. E. Akbas, and A. E. Cetin, "Approximate computation of dft without performing any multiplications: Application to radar signal processing," in *Signal Processing and Communications Applications Conference (SIU), 2014 22nd*, pp. 850–853, IEEE, 2014.

[61] Y. H. Habiboğlu, O. Günay, and A. E. Çetin, "Covariance matrix-based fire and flame detection method in video," *Machine Vision and Applications*, pp. 1–11, 2012.

[62] A. Afrasiyabi, B. Nasir, O. Yildiz, F. T. Y. Vural, and A. E. Cetin, "An energy efficient additive neural network," in *Signal Processing and Communications Applications Conference (SIU), 2017 25th*, pp. 1–4, IEEE, 2017.

[63] E. Kreyszig, *Introductory functional analysis with applications*, vol. 1. wiley New York, 1989.

[64] F. Leone, L. Nelson, and R. Nottingham, "The folded normal distribution," *Technometrics*, vol. 3, no. 4, pp. 543–550, 1961.

[65] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *International Conference on Machine Learning*, pp. 2849–2858, 2016.

[66] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pp. 1737–1746, 2015.

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

[68] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International Conference on Machine Learning*, pp. 448–456, 2015.

[69] O. Arikan, A. E. Cetin, and E. Erzin, "Adaptive filtering for non-gaussian stable processes," *IEEE Signal Processing Letters*, vol. 1, pp. 163–165, Nov 1994.

[70] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *European Conference on Computer Vision*, pp. 630–645, Springer, 2016.

[71] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 249–256, 2010.

[72] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

[73] B. Xu, R. Huang, and M. Li, "Revise saturated activation functions," *arXiv preprint arXiv:1602.05980*, 2016.

[74] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

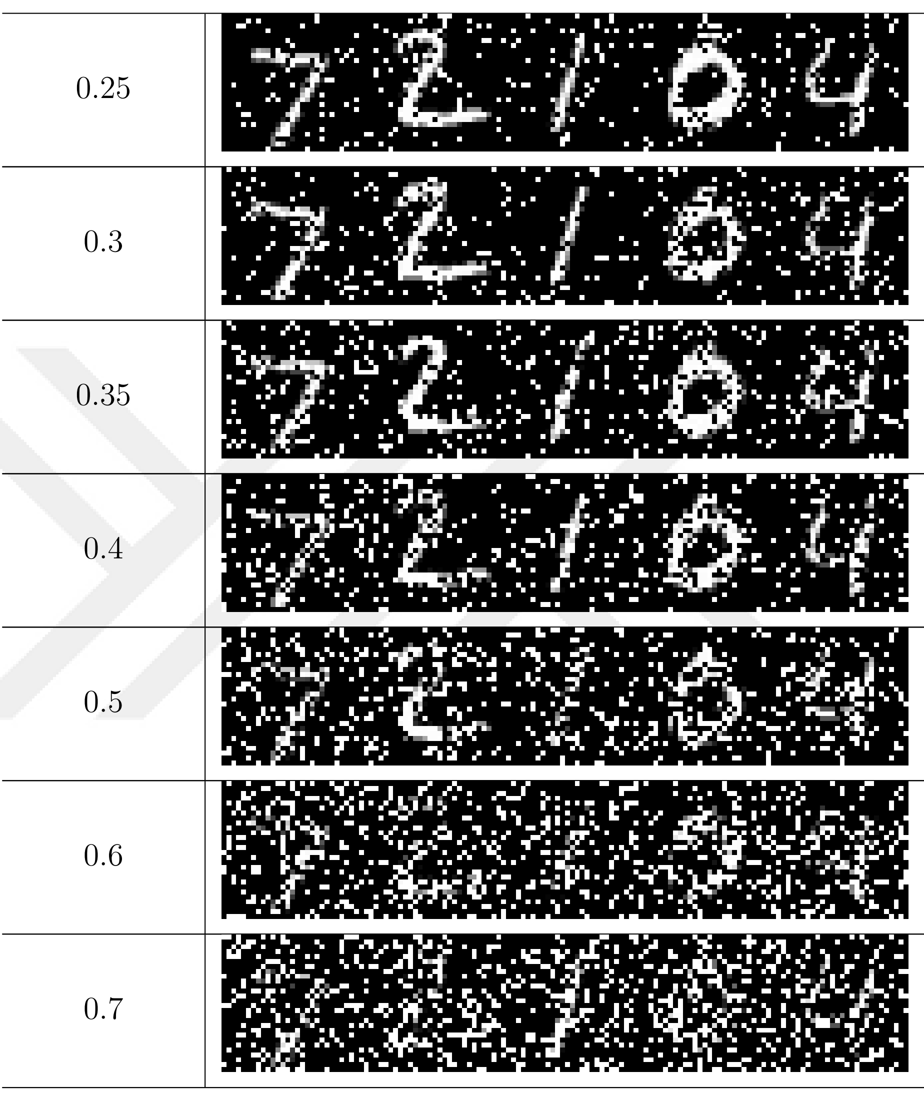
[75] J. Su, D. V. Vargas, and S. Kouichi, "One pixel attack for fooling deep neural networks," *arXiv preprint arXiv:1710.08864*, 2017.

[76] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," *arXiv preprint arXiv:1312.6199*, 2013.

Appendix A

MNIST Samples with Salt-and-Pepper Noise

SAP Level ρ	Example images
0.0	
0.05	
0.1	
0.15	
0.2	


0.25
0.3
0.35
0.4
0.5
0.6
0.7

Appendix B

CIFAR-10 Samples with Salt-and-Pepper Noise

SAP level ρ	example images
0.0	
0.01	
0.02	
0.03	
0.04	
0.05	
0.06	
0.07	