



UNIVERSITY OF THESSALY
SCHOOL OF ENGINEERING
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

**BOOSTING OF NEURAL NETWORK TRAINING BY
TOPOLOGY PRUNING**

Diploma Thesis

Chouliaras Andreas

Supervisor: Katsaros Dimitrios

Volos 2021



UNIVERSITY OF THESSALY
SCHOOL OF ENGINEERING
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

**BOOSTING OF NEURAL NETWORK TRAINING BY
TOPOLOGY PRUNING**

Diploma Thesis

Chouliaras Andreas

Supervisor: Katsaros Dimitrios

Volos 2021



ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

**ΕΠΙΤΑΧΥΝΣΗ ΕΚΠΑΙΔΕΥΣΗΣ ΝΕΥΡΩΝΙΚΩΝ
ΔΙΚΤΥΩΝ ΜΕ ΧΡΗΣΗ ΤΕΧΝΙΚΩΝ ΚΛΑΔΕΜΑΤΟΣ
ΤΟΠΟΛΟΓΙΑΣ**

Διπλωματική Εργασία

Χουλιάρης Ανδρέας

Επιβλέπων/πουσα: Κατσαρός Δημήτριος

Βόλος 2021

Approved by the Examination Committee:

Supervisor **Katsaros Dimitrios**

Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Bellas Nikolaos**

Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Thanos Georgios**

Member of Laboratory Teaching Staff, Department of Electrical and Computer Engineering, University of Thessaly

Date of approval: 15-9-2021

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor Prof. Dimitrios Katsaros for his invaluable advice and continuous support throughout my research. Your insightful feedback and the material you readily provided, helped me exceed my powers and brought my work to a higher level. Secondly, I would like to acknowledge the help of the Post Doc Researcher Evangelia Fragkou for her contribution to the literature review and for her precious feedback.

I would also like to thank the Emeritus Professor Dr. Elias Houstis. With his valuable teachings and tenacity, he pushed me to surpass my limits multiple times and sparked my interest in pursuing a career in Data Science. To my eyes, he was a true mentor during the latest years of my studies.

I can't omit my parents to whom I will forever be grateful, for their unconditional love and support, and their unwavering belief in me, who were beside my every stressful moment, my every success and failure, every joy and sadness. I am very fortunate to be your son.

Last but not least, I would like to thank all my friends, for cheering me up on my difficult and stressful times, and for providing me a safe space to relax from the difficulties and the adversities of life.

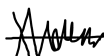
"You don't have to see the whole staircase, just take the first step."

Martin Luther King Jr.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant



Chouliaras Andreas

15-9-2021

Abstract

Recent research in the field of deep learning aim to reduce the model size of a Neural Network, using methods of Pruning in order to minimize its computational costs and its data requirements, in order to be capable to run on devices with memory constraints. In this thesis, we employ a pruning technique in order to sparsify Multi-Layer Perceptron (MLP) Networks, where their number of connections are pruned and restored at each epoch. We introduce three new techniques that use changing weight evolution rates, each one of which following a rule: Linear Decreasing Variation (LDV), Oscillating Variation (OSV) or Exponential Decay (EXD). We conducted experiments on many different datasets using MLP Networks, implemented in Python using sparse matrix operations, and evaluated based on their memory footprint, speed and accuracy, when compared to the SET procedure and their dense MLP counterpart. The results showed that the EXD method is the best performing method, achieving the lowest time costs, with memory footprints of the same level as all our tester sparse methods and accuracy results of the same approximate level to the other methods tested, while in many cases even surpassing them. Furthermore, we show that the sparse models using Keras, are under-performing by a great margin to the dense one. This is a little counter-intuitive to the benefits of model sparsification. As a result, we present an improved version of the SET implementation in Keras, using the *Callbacks API*, making the SET implementation more efficient and closer to its original concept. This managed to bring the the sparse models training costs to the same level as the dense one. In addition, we compared the relative time costs between sparse and dense models on both the Keras Implementation and our Python implementation that uses sparse matrix operations. We show that the sparse models are substantially faster to the dense ones when using sparse matrix operations, indicating that the differences in computational costs, when using Keras, are due to the fact that Keras don't use sparse matrix operations. This confirms many claims stating that the current frameworks cannot fully benefit for the application of Sparse Network Topologies.

Περίληψη

Πρόσφατες έρευνες στη Βαθιά Μάθηση έχουν ως στόχο την ελάττωση του μεγέθους των μοντέλων Νευρωνικό Δικτύων, χρησιμοποιώντας τεχνικές κλαδέματος για να μειώσουν υπολογιστικά κόστη και απαιτήσεις σε μνήμη, ώστε να μπορούν να τρέξουν σε μικρές συσκευές. Σε αυτήν την διπλωματική χρησιμοποιούμε μια τεχνική κλαδέματος για την αραιοποίηση Multi-Layer Perceptron (MLP) δικτύων, όπου ο αριθμός συνδέσεων τους κλαδεύεται και αποκαθίσταται σε κάθε εποχή. Εισάγουμε τρεις νέες τεχνικές που εφαρμόζουν μεταβλητούς ρυθμούς εξέλιξης βαρών, με καθεμία να ακολουθεί έναν από τους επόμενους κανόνες: Γραμμική Φθίνουσα Μεταβολή (LDV), Μεταβλητή Ταλάντωση (OSV), Εκθετική Παρακμή (EXD). Διεξήγαμε πειράματα σε αρκετά σετ δεδομένων με δίκτυα MLP, υλοποιημένα σε Python χρησιμοποιώντας πράξεις αραιών πινάκων, και αξιολογήθηκαν βάση του αποτύπου μνήμης, της ταχύτητα και της ακρίβειας τους σε σχέση με την μέθοδο SET και των αντίστοιχων πυκνών δικτύων. Τα αποτελέσματα ανέδειξαν την μέθοδο EXD ως πιο αποδοτική, πετυχαίνοντας τα χαμηλότερα χρονικά κόστη, με ακρίβεια αντίστοιχη των άλλων δοκιμασμένων μεθόδων, και σε πολλές περιπτώσεις λίγο μεγαλύτερη ενώ το αποτύπωμα μνήμης παραμένει ισάξιο των άλλων αραιών μεθόδων. Επιπλέον δείχνουμε ότι τα αραιά μοντέλα σε Keras υποαποδίδουν των πυκνών. Όντας λίγο αντίθετο με τα αναμενόμενα οφέλη της αραιώσης, παρουσιάζουμε μια βελτιωμένη έκδοση της υλοποίησης του SET σε Keras, με χρήση του Callbacks API, κάνοντας τα αραιά μοντέλα αποδοτικότερα, κατορθώνοντας να φέρει αραιά και πυκνά μοντέλα πιο κοντά σε κόστος εκπαίδευσης. Επιπλέον συγκρίναμε τα σχετικά χρονικά κόστη ανάμεσα σε αραιά και πυκνά μοντέλα, τόσο στην Keras όσο και στην Python υλοποίηση που εφαρμόζει πράξεις αραιών πινάκων. Δείχνουμε πως τα αραιά μοντέλα είναι σημαντικά ταχύτερα των πυκνών με την χρήση πράξεων αραιών πινάκων, δείχνοντας πως οι διαφορές σε χρονικά κόστη με την χρήση του Keras, οφείλονται στην μη χρήση πράξεων αραιών πινάκων. Αυτό επιβεβαιώνει ισχυρισμούς που δηλώνουν ότι οι σύγχρονες υποδομές δεν μπορούν να αξιοποιήσουν πλήρως τα οφέλη των δικτύων με αραιή τοπολογία.

Table of contents

Acknowledgements	ix
Abstract	xi
Περίληψη	xiii
Table of contents	xv
List of figures	xix
Abbreviations	xxi
1 Introduction	1
1.1 Subject	1
1.1.1 Contributions	2
1.2 Structure	2
2 Artificial Neural Networks	3
2.1 Introduction to Neural Networks	4
2.1.1 Neural Networks, in General	5
2.1.2 Interconnections Among Nodes	7
2.1.3 Design Issues for Neural Networks	7
2.2 Dense Feed-forward Networks	8
2.2.1 Linear Algebra Notation	8
2.2.2 Activation Functions	10
2.2.3 Loss Functions	14
2.3 Training Neural Networks	17

2.3.1	Gradients, Jacobians, and the Chain Rule	17
2.3.2	Iterating Gradient Descent	19
2.3.3	The Stochastic Gradient Descent Variation	19
2.3.4	The Backpropagation Algorithm	20
2.3.5	Incremental vs Minibatch Learning	22
2.3.6	Heuristic Modifications of Backpropagation	23
2.3.7	Tensors	26
2.4	Regularization	27
2.4.1	Norm Penalties	28
2.4.2	Dropout	29
2.4.3	Dataset Augmentation	30
3	Sparse Neural Networks	31
3.1	Related Work	32
3.2	The Lottery Ticket Hypothesis	33
3.3	A Small Reference to Network Science Concepts	35
3.3.1	Regular Graphs	35
3.3.2	Small-World Networks	35
3.3.3	Scale-Free Networks	36
3.3.4	Erdős–Rényi Random Graphs	36
3.4	Weight Masks	37
3.5	Sparse Evolutionary Training (SET)	37
3.5.1	The SET Procedure	38
3.5.2	The Biological Background	41
4	Proposed Techniques	43
4.1	Motivations from Biology	44
4.2	Taking the logic behind SET one step further	45
4.3	Linear Decreasing Variation (LDV)	45
4.4	Oscillating Variation (OSV)	46
4.5	Exponential Decay (EXD)	47
4.6	The Importance of Training Speed	47
4.7	The Tensorflow-Keras Implementation Incidents	48

5	Evaluation & Results	51
5.1	Experimental Evaluation	51
5.2	Evaluation Settings	52
5.2.1	Hardware Utilized and Software Implementations	52
5.2.2	Experimentation Categories	52
5.2.3	The Memory Footprint	54
5.2.4	Python Code vs Keras	55
5.3	Results	55
5.3.1	Comparing Performance on more Datasets	55
5.3.2	Comparing Keras Implementations	61
5.3.3	Comparing Speed Using Sparse Matrix Operations	63
5.3.4	Comparing Performance using 4 Hidden Layers	64
6	Conclusions	69
6.1	Summary and Conclusions	69
6.2	Future work	70
	Bibliography	71
	APPENDICES	75
A	Utilized Software and Tools	77
A.1	The Implementations Used	77
A.1.1	Custom Python Implementation	77
A.1.2	Custom Python Implementation	77
A.2	Tensorflow	78
A.3	Keras	78
A.4	Datasets	79
A.4.1	The LUNG Dataset	79
A.4.2	The ORL Dataset	79
A.4.3	The Prostate-GE Dataset	80
A.4.4	The GLIOMA Dataset	80
A.4.5	The Fashion MNIST Dataset	81

B	Experimental Results and Figures	83
B.1	Comparing Performance on more Datasets	83
B.2	Comparing Keras Implementations	84

List of figures

2.1	Perceptron, the fundamental unit of Neural Networks.	5
2.2	The general case of a neural network.	6
2.3	A graphical representation of the Heaviside Step Activation Function.	11
2.4	A graphical representation of the Logistic Sigmoid Activation Function.	12
2.5	An example application of the Softmax Activation Function.	13
2.6	A graphical representation of the ReLU Activation Function.	13
2.7	Finding the proper Learning Rate.	19
2.8	An example error performance surface of two variables.	24
2.9	The impact of Momentum in Training.	25
2.10	A Neural Network before and after applying Dropout.	29
2.11	How Data Augmentation helps us generate more data.	30
3.1	Different kinds of Network Structures	37
3.2	An illustration of the weight evolution procedure. For each sparse connected layer, SC^k (a) of a Neural Network, at the end of a training epoch a fraction of the weights closest to zero, are removed (b) . Then a number of weights is randomly added that is equal to the number of weights removed (c) . The process is repeated at the end of each epoch for the rest of the training phase (d)	39
5.1	Accuracy of the Python code on the LUNG Dataset.	56
5.2	Training and Inference times on the LUNG Dataset.	56
5.3	Accuracy of the Python code on the ORL Dataset.	57
5.4	Training and Inference times on the ORL Dataset.	57
5.5	Accuracy of the Python code on the Prostate-GE Dataset.	58
5.6	Training and Inference times on the Prostate-GE Dataset.	58
5.7	Accuracy of the Python code on the GLIOMA Dataset.	59

5.8	Training and Inference times on the GLIOMA Dataset.	59
5.9	Accuracy of the Python code on the Fashion MNIST Dataset.	60
5.10	Training and Inference times on the Fashion MNIST Dataset.	60
5.11	Comparing the number of weights used between the Dense MLP and the Sparse methods in the Fashion MNIST Dataset.	61
5.12	Comparative graph of Training Times using the old Keras implementation versus our Keras implementation in the Fashion MNIST Dataset.	62
5.13	Comparative graph of Inference Times using the old Keras implementation versus our Keras implementation in the Fashion MNIST Dataset.	63
5.14	Accuracy Graph on the Fashion MNIST dataset comparing the proposed techniques using our improved Keras implementation.	64
5.15	Training and Inference times on the Fashion MNIST Dataset using the custom Python code.	65
5.16	Training and Inference times on the Fashion MNIST Dataset using the custom Python code with 4 Hidden Layers.	65
5.17	Accuracy Graph on the Fashion MNIST dataset comparing the proposed techniques, using the custom Python code with 4 Hidden Layers.	66
5.18	Accuracy Graph of the first 100 epochs on the Fashion MNIST dataset comparing the proposed techniques, using the custom Python code with 4 Hidden Layers.	66
A.1	ORL Dataset Sample	80
A.2	Fashion MNIST Dataset Sample	81

Abbreviations

Acc	Accuracy
ANN	Artificial Neural Network
EXD	Exponential Decay
CR	Compression Rate
LDV	Linear Decreasing Variation
MLP	Multi-Layer Perceptron
MLP Small	MLP Network with 3 hidden layers as 1024-512-512 neurons
MLP 1000	MLP Network with 3 hidden layers as 1000-1000-1000 neurons
MLP 4k	MLP Network with 3 hidden layers as 4000-1000-4000 neurons
MLP 4k4L	MLP Network with 4 hidden layers as 4000-4000-4000-4000 neurons
OSV	Oscillating Variation
SET	Sparse Evolutionary Training

Chapter 1

Introduction

Artificial Neural Networks are very powerful computing systems that are widely popular today for being able to solve complex problems with great accuracy. This complexity and power needs heavy computational requirements, making them hard to utilize on small weaker devices. The greatest reason is because they consume a lot of time to train and leave great memory footprints. A portion of the current research efforts has turned towards sparsification methods that try to address those problems.

1.1 Subject

To make Neural Networks run faster and use less memory a lot of research has been conducted. Among them, we separate the Sparse Evolutionary Training (SET) procedure, that shows great potential because it manages to significantly reduce the model's memory footprint while at the same time it achieves slightly greater accuracy than its dense counterpart. But in the original paper that was introduced there little to no exploration about the procedure's time performances. In our work we test SET procedure in more datasets to study its time costs and we even present three new variant methods that we aspire to outperform SET. Among them a method that uses Exponential Decay to its weight evolution procedure, manages to stand out among the sparse methods and proves to be a clear winner.

Furthermore, we take the Keras implementation given by the creators of the SET procedure and after showing that the sparse methods perform worse in speed than their dense counterpart, we manage to improve it considerably, resulting to speeds of the same level. Finding this equality in speed a little counter-intuitive to the memory footprint and reduced

complexity used by our sparse methods we test our methods on a custom implementation that utilizes smart sparse matrix operations, proving that the Tensorflow framework cannot fully benefit by the reduced training costs that are resulting from the sparsification of dense models, indicating it as a big obstacle in the research of sparse neural network models.

1.1.1 Contributions

Our contributions are summarized as:

1. Testing the SET procedure on more datasets.
2. Addressing training and test speeds.
3. Designing three new algorithms to compare their accuracy with SET and dense MLPs.
4. Creating a better Keras Implementation that improves the speed of sparse methods.
5. Showing that some of our proposed methods perform better than SET and dense MLPs.
6. Indicate that current frameworks cannot fully utilize Sparse Network Topologies.

1.2 Structure

The rest of this work is structured as follows: Chapter 2 makes an introduction to Artificial Neural Networks, from their design and their structure to their training and model regularization. Chapter 3 refers to Sparse Neural Networks: the related work, the Lottery Ticket Hypothesis, how sparsification is represented and implemented. In chapter 3 is also detailed how the SET procedure works along with the why and how we think it can be improved. In Chapter 4 we introduce our proposed methods and our inspiration background. In Chapter 5 we define our evaluation settings, we present our results and we discuss their meaning. Finally in chapter 6 we conclude our work and we give some general ideas for future works.

Details about the software and tools we utilized along with the datasets used can be found in the Appendix A. Technical details about the experiments we discuss in chapter 5 can be found in Appendix B.

Chapter 2

Artificial Neural Networks

”Just by reading these lines our brains use a complex biological neural network. Humans have a highly interconnected set of around 10 billion neurons to facilitate reading, breathing, motions and thinking. Each one of those biological neurons, has a rich assembly of tissue and chemistry, the complexity of which, if not the speed, exceeds that of a microprocessor. Some of this neural structure is obtained at birth and other parts grow and evolve with time and experience.” [1]

Scientists have been trying to understand how biological neural networks function for many years, but the progress is very slow and scientists need to overcome new obstacles time after time. It is now known however, that almost all biological neural functions are stored in neurons and in the connections between them. Learning is considered to be the establishment of new connections between neurons or the modification of existing ones. All these made scientist wonder that, even though we have only a rudimentary understanding of biological neural networks, can a small set of simple artificial ”neurons” be constructed, that can be trained to serve a useful purpose? I believe we already know the answer...

These artificial neurons are nothing more but a simple abstraction of biological neurons, brought to life as elements in computer programs or perhaps in silicon circuits by using maths and good-old programming. These networks of artificial neurons, even though they don’t have even a fraction of the complexity and power of the human brain, they can be trained to perform useful tasks. And some times with accuracy and speed, even better than that of the humans.

Through the years, the research in the field grew enormously and the application of neural networks spread to every corner imaginable in science. Today, neural networks find applica-

tions in many industries like Aerospace, Automotive, eCommerce, Finance and Healthcare to name only a few.

This chapter was based on some very interesting books I urge you to check out. Firstly I liked Chapter 12 and 13 of the 3rd edition of the book *Mining Massive Datasets* of Jure Leskovec, Anand Rajaraman and Jeff Ullman [2]. It captures all the basic theory of neural networks without going too much into the details, keeping it short and understandable for people that are not very familiar with this field. Another book I found really helpful is *Neural Networks Design* of Martin T. Hagan [1]. I find it to be a complete study of the entire field, with details in the theoretical and mathematical background that doesn't miss anything. Finally I can't forget to mention *Dive into Deep Learning* [3] an online interactive book with coding examples, community contributions and active discussions.

2.1 Introduction to Neural Networks

The fundamental unit of a neural network is the neuron or the perceptron as we often see it called. Their concept albeit important is simple and it consists of three basic elements:

1. A set of *synapses*, *connections* or *links* (many names, same functionality), each one of those guiding the input towards the "main body" of the neuron. Each one of those links has an internal value, called *weight* which is nothing more than a number that is multiplied with the input value to this link and produces a weighted value of the input.
2. An operator, which is usually an adder that takes as the weighted inputs and adds them, producing a weighted sum as a result.
3. An activation function is then used to transform the result of the operator in an output range suitable to the results it needs to produce. This function can be as simple as a threshold function, producing results in the range of "yes" or "no", 0 or 1 etc, or it can be a function that produces a real numerical value. This result is the output of the entire neuron.

These simple neurons can separate inputs into two classes, as long as the classes are linearly separable. The figure 2.1 displays the design of the perceptron.

We can observe in this figure one of the inputs being 1. This omission was deliberate. Its corresponding weight, displayed as w_0 , it is usually called *bias* and is nothing more than a

displacement of the entire weighted sum by a flat value. We can imagine that the neuron is producing a hyper-plane where the weights determine the slopes and the bias determines the intercept. We can do without the bias but it offers one more degree of freedom when trying to separate data in higher dimensions and that makes our life somewhat easier.

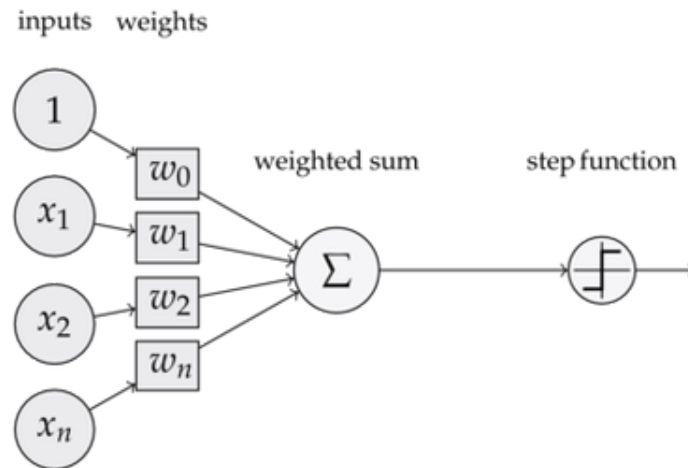


Figure 2.1: Perceptron, the fundamental unit of Neural Networks.

However, most problems of interest and importance are not linearly separable. For this reason we consider the design of *neural networks*, which are collections of perceptrons or nodes, where the outputs of one rank (or layer) of nodes becomes the inputs to the nodes of the next. The last layer of nodes produces the output of the entire neural network. The training of neural networks with many layers requires enormous numbers of training examples and resources, but has proven to be an extremely powerful technique, referred to as deep learning, when it can be used.

2.1.1 Neural Networks, in General

When combining numerous neurons on a single network, we usually imagine a configuration as depicted in figure 2.2 which is that of a directed graph. The first or *input layer*, is the input, which is presumed to be a vector of some length n . Each component of the vector $[x_1, x_2, \dots, x_n]$ is an input to the network. After that, there are one or more *hidden layers* and finally, at the end, an *output layer*, which gives the result of the network. Each one of the layers can have a different number of nodes, and in fact, choosing the right number of nodes at each layer is an important part of the design process for neural networks. Especially, note that the output layer can have many nodes. This refers to problems that needs each input to

be classified to many different classes, with one output node for each class.

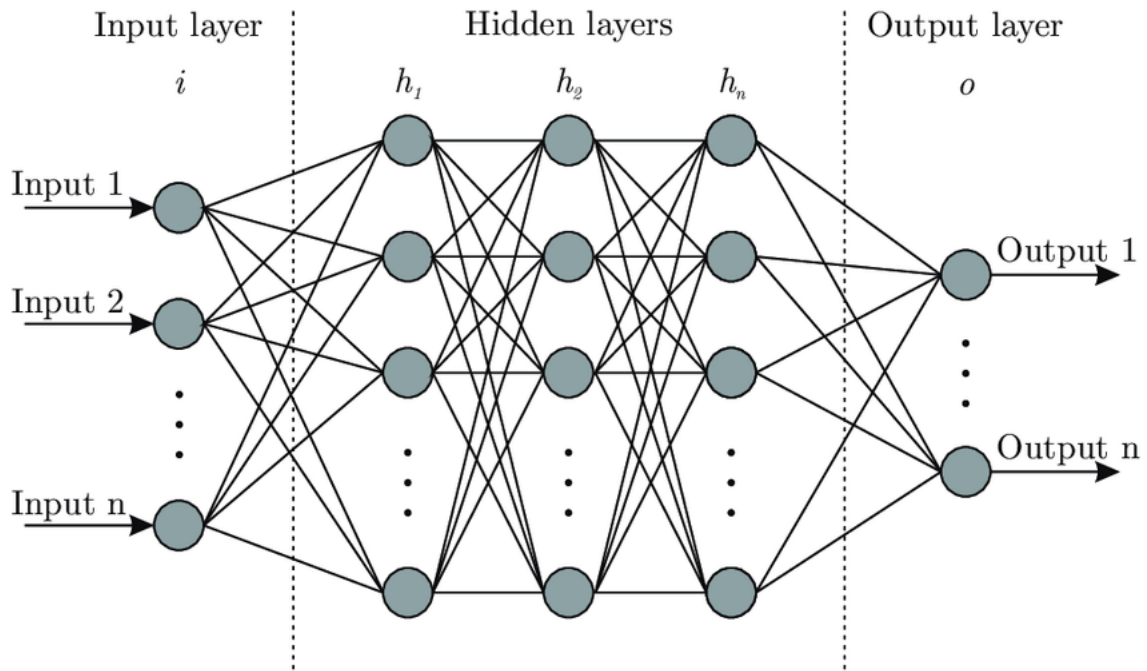


Figure 2.2: The general case of a neural network.

Each layer, except for the input layer, consists of one or more nodes, which we arrange in the column that represents that layer. We can think of each node as a perceptron producing an output y_i . The inputs to a node are the outputs of some or all of the nodes in the previous layer (x_i). Associated with each input to a node is a *weight* (w_i). The output of the node depends on the following expression:

$$y_i = f \left(\sum x_i w_i \right)$$

where the sum is over all the inputs x_i , and w_i is the respective weights to that input. The function f simply refers to the activation function used in this layer.

Sometimes, we want the output to be either 0 or 1; the output is 1 if the sum is positive and 0 otherwise. This can be achieved by the use of simple step-functions. However as we shall see later, it is often more convenient, when trying to learn weights for a neural network that solves some problem, to have outputs that are almost always close to 0 or 1 but not the exact values. The reason, intuitively, is that it is then possible for the output of a node to be a continuous function of its inputs. We can then use some optimization algorithm to find the ideal values of all the weights in the network.

2.1.2 Interconnections Among Nodes

Neural Networks can differ in how the nodes at one layer are connected to the nodes at next layer. From this differentiation many interconnection options arise, but for the scopes of this project only two are relevant:

1. Fully-Connected. This is the most usual and general case where each node takes as inputs, the outputs of every node of the previous layer. The resulting network is dense with many parameters to be trained.
2. Random. For some m we pick for each node, m nodes from the previous layer and make those, and only those to be inputs for this node. The resulting network has a random sparse structure with an adjustable number of parameters to be trained, usually considerably less than its corresponding dense network.

We will see in chapter 3 that fully connected networks are computationally expensive, while a randomly interconnected network can achieve similar accuracy to a fully connected one, with way less computational cost.

2.1.3 Design Issues for Neural Networks

Building a neural network to solve a given problem is partially art and partially science. Before we even begin to train a network, by finding the weights on the inputs that serve our goal best, we need to make a number of important design decisions. We need to decide on the number of layers we will use, the number of neurons each of them have. We must decide how the outputs of one layer connects to the inputs of the next layer and for how many epochs we will train the network. While the above decisions cover the greater spectrum of the programmer level there are even more decisions that can arise, depending on the particular circumstances.

In later sections we shall see that there are even more decisions to be made when we train the neural network, that are more specific to the way the training happens. These include, deciding on the cost function we choose to minimize to express what weights are best for the network, what activation function to use in each layer and which optimization algorithm we will use on the training input to calculate the optimal weights.

The choices we mention above, while most times arise from our understanding of optimization algorithms, cost functions and the data we have, many times are made by trial and error, or techniques that explore different combinations of the tried choices.

2.2 Dense Feed-forward Networks

The true value of neural networks comes from our ability to design them based on the training data we have. To design a network, there are many choices that must be made, such as the number of layers and the number of nodes for each layer etc., as was discussed in the previous Section. These choices can be more art than science. The computational part of training, which is more science than art, is primarily the choice of weights for the inputs to each node. The techniques for selecting weights usually involves convergence using an optimization algorithm which is, most of the times, a variant of gradient descent. But gradient descent requires a cost function, which must be a continuous function of the weights. The basic perceptron networks operate with outputs of 0 or 1, so the outputs are not normally continuous functions of the inputs. In this section, we shall discuss various ways that one can use to modify the behavior of the nodes in a neural network, so that the outputs become continuous functions of the inputs, therefore a reasonable cost function can be applied to the outputs that will also be continuous.

2.2.1 Linear Algebra Notation

The way of describing neural networks as an interconnection of nodes (or perceptrons) with distinct connections between each one of them, while useful for educational purposes, isn't usually the way that computers create, train, or save them. Linear Algebra Notation offers not only brevity in expressing complex neural network structures, but also better performance. Modern graphics processing units (GPU's) are designed to compute linear-algebra operations in a highly parallel way. Multiplying matrices and vectors using simple expressions is much faster than coding each node or using nested loops.

Using linear algebra, we can express the input layer of nodes as a column vector of the inputs like $\mathbf{x} = [x_1, x_2, \dots, x_m]$, while a hidden layer with n nodes can be expressed as $\mathbf{h} = [h_1, h_2, \dots, h_n]$. The weights of the inputs connected to the first node of the hidden layer can be then expressed as $\mathbf{w}_1 = [w_{11}, w_{12}, \dots, w_{1m}]$ and similarly the weights of the inputs connected

to the second node of the hidden layer as $\mathbf{w}_2 = [w_{21}, w_{22}, \dots, w_{2m}]$ and so on for all m weight vectors.

The intercept or bias inputs to the hidden layer nodes form a vector $\mathbf{b} = [b_1, b_2, \dots, b_n]$ often called bias vector. The step function is then applied to produce the result of each node in a hidden layer with the following expression:

$$h_i = \text{step}(\mathbf{w}_i^T x + b_i), \text{ for } i = 1, 2, \dots, n$$

We can further simplify the expressions by merging the weight vectors in a single $n \times m$ weight matrix W , where the i^{th} row of W is w_i^T . The whole hidden layer can be thus be expressed as:

$$\mathbf{h} = \text{step}(W\mathbf{x} + \mathbf{b})$$

In this case the step function operates element-wise on the $W\mathbf{x} + \mathbf{b}$ vector. With similar logic we can design all hidden layers as well as the final output layer the result of which would match the nature of the problem. We can see that for each layer we have a weight matrix we need to "train" so that all those parameters are set in a suitable way so that the task at hand is achieved. We will see how this goal is achieved in later sections.

Linear Algebra Notation works just as great when we have larger inputs, many more hidden layers and many more nodes in each hidden layer. We just need to expand the weight matrix and bias vector accordingly. That is, the W matrix needs to have one more row for each node in the layer and one more column for each output of the previous layer, or input if this is the first layer. The bias vector has one component for each node in the layer. It is also very easy to handle cases where there needs to be more than one output in the network. For example, in a multi-class classification problem, we need to have so many output nodes y_n as the number of the target classes n . So for a given input, the outputs specify the probability that the input belongs to the corresponding class. This arrangement results in an output vector $\mathbf{y} = [y_1, y_2, \dots, y_n]$, where n is the number of classes.

The network in our example uses a non linear step function. More generally, as we will see in the following section, we can use any other nonlinear function, usually called *activation function* or transfer function.

This design of neural networks that consist of many layers, each one of whose inputs are taken from the previous layer and they "feed" their outputs to the next layer in a serial manner

is generally called feed-forward networks, since there are no branches or cycles.

2.2.2 Activation Functions

A node in a neural network was initially designed to give a 0 or 1 (yes or no) output. Often, we want to modify that output in many ways, depending on the type of data we use or the problem we try to solve. They are differentiable operators that transform the results of a neurons calculations to a result suitable to the neuron's purpose. In cases like theses we apply an activation function to the output of a node. In some other cases, the activation function takes all the outputs of a layer and modifies them as a group. One of the most known algorithms used for neural network training is gradient descent. Thus, we need activation functions that collaborate well enough with the gradient descent. In particular, we look for the following properties in the activation functions we use:

1. The function should be continuous and differentiable everywhere (or almost everywhere).
2. The derivative of the function does not *saturate* (i.e., become very small, tending towards zero) over its expected input range. Very small derivatives tend to stall the learning process.
3. The derivative does not *explode* (i.e., become very large, tending towards infinity), since this would lead to issues of numerical instability

Because activation functions are a fundamental aspect of neural networks, we will briefly refer to the most known amongst them.

Threshold Functions

Threshold Functions are function that its output is determined based on the range its input belongs to. The step function we have previously mentioned belongs into this kind of activation functions. The definition of the step function is:

$$H_s(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \quad (2.1)$$

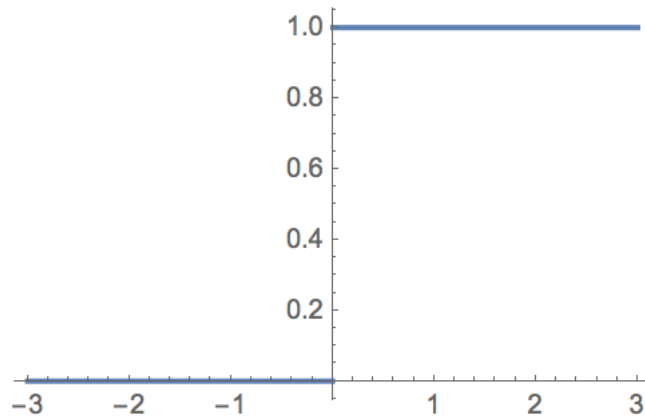


Figure 2.3: A graphical representation of the Heaviside Step Activation Function.

Other kind of threshold functions can have more branches, different cutoff points or output values etc. The aforementioned step function is the one most usually used and is commonly referred to as *Heaviside Function* (figure 2.3). The step function however fails to satisfy the properties (2) and (3) we look for in an activation function. Its derivative explodes at zero and is zero everywhere else. Thus, the step function is not suitable for gradient descent and is not a good choice for deep neural networks.

The Sigmoid

If we cannot use the step function, we look for alternatives in the class of *sigmoid functions* - their name derived because of the S-shaped curve that these functions exhibit. They are strictly increasing functions that achieve an acceptable mimicry of the step function but having a balance between linear and nonlinear behavior. The most commonly used sigmoid function is the *logistic sigmoid* which is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-ax}} = \frac{e^{ax}}{1 + e^{ax}} \quad (2.2)$$

Where a is a parameter that tweaks the slope. Variations in the a parameter can produce smoother or steeper slopes. We notice that the logistic sigmoid has value of 0.5 at $x = 0$ (Figure 2.4). For large x , the sigmoid function approaches 1, and for large, negative x the sigmoid approaches 0. The logistic sigmoid, like most activation functions are applied to vectors element-wise, so if $[x_1, x_2, \dots, x_n]$ then

$$\sigma(x) = [\sigma(x_1), \sigma(x_2), \dots, \sigma(x_n)]$$

The logistic sigmoid has several advantages over the step function as a way to define the output of a perceptron. The logistic sigmoid is continuous and differentiable, so it enables us to use gradient descent to discover the best weights. Since its output values is in the range of $[0,1]$, it is possible to interpret the outputs of the network as a probability. However, the logistic sigmoid saturates very quickly as we move away from the "critical region" around zero. So the derivative goes towards zero and the gradient-based learning can stall out. That is, weights almost stop changing, once they get away from zero.

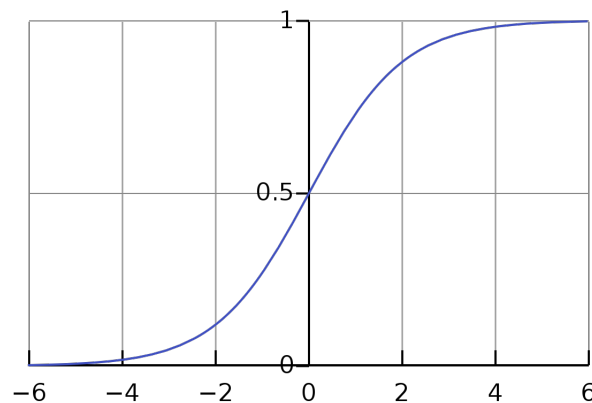


Figure 2.4: A graphical representation of the Logistic Sigmoid Activation Function.

If the desired output need to be in the range of $[-1, 1]$ instead of the range $[0, 1]$, that the logistic sigmoid produces, we can use the *hyperbolic tangent function* defined by:

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

Allowing the activation function produce negative values may yield some practical benefits, depending on the data.

Softmax

The *softmax* function differs drastically from sigmoid functions in that it does not operate element-wise on a vector. Rather, the softmax function applies to an entire vector. If $\mathbf{x} = [x_1, x_2, \dots, x_n]$, then its softmax $\mu(\mathbf{x}) = [\mu(x_1), \mu(x_2), \dots, \mu(x_n)]$ where:

$$\mu(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \quad (2.3)$$

Softmax pushes the largest component of the vector towards 1 while pushing all the other components towards zero (Figure 2.5). Also, all the outputs sum to 1, regardless of the sum of

the components of the input vector. Thus, the output of the softmax function can be interpreted as a probability distribution. A common application is to use softmax in the output layer for a classification problem. The output vector has a component corresponding to each target class, and the softmax output is interpreted as the probability of the input belonging to the corresponding class. Softmax has the same saturation problem as the sigmoid function, since one component gets larger than all the others. There is a simple workaround to this problem, however, when softmax is used at the output layer. In this case it is usual to pick *cross entropy* as the loss function, which undoes the exponentiation in the definition of softmax and avoids saturation. Cross entropy is explained in a following section.

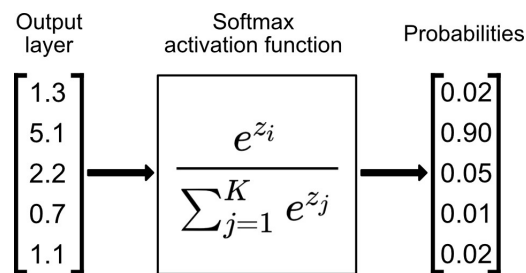


Figure 2.5: An example application of the Softmax Activation Function.

Rectified Linear Unit

The *rectified linear unit*, or ReLU, is defined as:

$$f(x) = \max(0, x) = \begin{cases} x, & \text{for } x \geq 0 \\ 0, & \text{for } x < 0 \end{cases} \quad (2.4)$$

and most usually recognizable through its graphical representation (Figure 2.6).

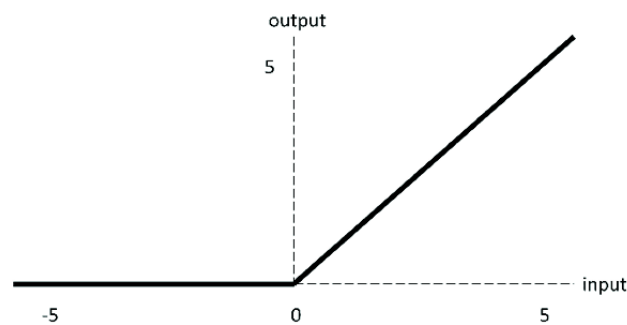


Figure 2.6: A graphical representation of the ReLU Activation Function.

The name of this function derives from the analogy to half-wave rectification in electrical engineering. The function is not differentiable at 0 but is differentiable everywhere else, including at points arbitrarily close to 0. In practice, we "set" the derivative at 0 to be either 0 (the left derivative) or 1 (the right derivative). In modern neural networks, a version of ReLU has replaced sigmoid as the default choice of activation function. The popularity of ReLU derives from two properties:

1. The gradient of ReLU remains constant and never saturates for positive x , speeding up training. It has been found in practice that networks that use ReLU offer a significant speedup in training compared to a sigmoid activation.
2. Both the function and its derivative can be computed using elementary and efficient mathematical operations without any exponentiation.

ReLU does suffer from a problem related to the saturation of its derivative when $x < 0$. Once a node's input values become negative, it is possible that the node's output get "stuck" at zero through the rest of the training. This is called the dying ReLU problem. There has been various approaches to cope with this problem the contents of which, drifts away from the scope of this project.

2.2.3 Loss Functions

"A loss function quantifies the difference between a model's predictions and the correct values observed in the real world"[2]. We can define the correct values that the neural network model should produce for an input \mathbf{x} as \hat{y} and the model's actual prediction as \mathbf{y} . Then we describe the loss function as $L(y, \hat{y})$ and is defined so that it quantifies the prediction error for a single input. Usually we use the loss function over a large set of observations, such as the entire training set. In that case, we usually average the losses over the number of observations used in the loss function.

We will refer to two study cases on the loss functions. The first is when we have a single output node, resulting in a real value. These cases are referred to as "regression loss". In the second case we have several output nodes, each of which indicated the probability of the input being in a particular class. These cases as referred to as "classification loss".

Regression Loss

Let's suppose that our model has a single output node which produces a single real value as a prediction. For a training example (\mathbf{x}, \hat{y}) we have \mathbf{x} as the input and \hat{y} as the observed correct value we want our model to learn to predict. We also consider as y the actual prediction the model produces. The *squared error loss* $L(y, \hat{y})$ of this prediction is:

$$L(y, \hat{y}) = (y - \hat{y})^2$$

The square exponentiation is used for two main reasons. Mostly because we prefer the loss functions to produce non-negative values and secondly to intensify the impact of the error, making the training converge faster. In general we compute the loss over a set of predictions. If the training set consists of input-output pairs as $T = \{(\mathbf{x}_1, \hat{y}_1), (\mathbf{x}_2, \hat{y}_2), \dots, (\mathbf{x}_n, \hat{y}_n)\}$ while the model's predictions are $P = \{y_1, y_2, \dots, y_n\}$. The *mean squared error* (MSE) for the trained set is:

$$L(P, T) = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

Another popular loss function is the Root Mean Squared Error (RMSE) which is simply the square root of the Mean Squared Error. We usually prefer the MSE as we don't have the square root and that simplifies the derivative of the loss function, that we need during the training. In any case, when we minimize the MSE we also minimize the RMSE.

One problem with MSE is that it is very sensitive to outliers due to the squared term. Even a few outliers can contribute very highly to the loss and weaken the effect of other points, making the training process susceptible to wide swings. There has been a lot of research and a lot of approaches to cope with this problem, but the study of such methods drifts away from the scope of this project.

Classification Loss

When dealing with classification problems we have at least two classes that we need our data points to classify into. When we just have two classes it is called a binary classification problem, while having more than two classes is called a multi-class classification problem. We will study classification loss in the case of multi-class problems as it is more generic and includes the case of binary classification problems as a more simplified case.

Let's suppose our multi-class classification problem has n target classes as: C_1, C_2, \dots, C_n . We consider each point in the training set, again here like previously, in input-output pairs of the form \mathbf{x}, \mathbf{p} where \mathbf{x} as the input and $\mathbf{p} = [p_1, p_2, \dots, p_n]$ as the output. In this case p_i gives us the probability that the input belongs to the corresponding class C_i . The sum of all those probabilities should always be 1. When we are sure that the an input belong to a class C_i it should be true that the probability $p_i = 1$ and $p_j = 0$ for $i \neq j$. But in the general case we can interpret p_i as the level of certainty that an input \mathbf{x} belongs to the class C_i , and \mathbf{p} as the probability distribution over the target classes.

In a similar manner we define the output predictions of a neural network model as a vector $\mathbf{q} = [q_1, q_2, \dots, q_n]$ of probabilities, with $\sum_i q_i = 1$ just as before. We interpret \mathbf{q} as a probability distribution over the target classes, with q_i gives the probability that the input belongs to the corresponding class C_i . This could be achieved by using the Softmax activation function on the final layer of a neural network. So we now have two probability distributions, as \mathbf{q} the model's prediction and as \mathbf{p} the observed values in our data. We need to quantify the distance between those two probability distributions to order to be able to train such network. For this need we can use the entropy of a discrete probability distribution \mathbf{p} that is defined as:

$$H(\mathbf{p}) = - \sum_{i=1}^n p_i \log p_i$$

Which simple indicates the level of uncertainty inherent in the variable's possible outcomes. As a product of information theory, when we know an approximation distribution \mathbf{q} and target distribution \mathbf{p} , then the additional information needed to represent an event from the \mathbf{p} distribution using the \mathbf{q} distribution is called cross-entropy and is defined as:

$$H(\mathbf{p}, \mathbf{q}) = - \sum_{i=1}^n p_i \log q_i$$

We can observe that $H(\mathbf{p}, \mathbf{p}) = H(\mathbf{p})$, and generally it is true that $H(\mathbf{p}, \mathbf{q}) \geq H(\mathbf{p})$. This cross-entropy is the most commonly used loss function for classification problems which usually appears in the following forms. Binary Cross-Entropy an implementation used for binary classification problems and Categorical Cross-Entropy for multi-class classification problems.

2.3 Training Neural Networks

We have presented so far how neural networks are designed, how they calculate predictions, and how we can quantify their error through loss functions. It is now time to study how they are trained. This "training" we have mentioned so many times already, is nothing more than an algorithm that tries to find what values each weight in the network should have, in order to minimize the average loss on the training set. Or, to express it more simply, to find weights that make the network's predictions have the least error possible.

To successfully train a neural network, firstly we need a large enough dataset. This dataset, which should essentially contain pairs of inputs and outputs, must be representative of the data the network could encounter in the future. It is possible however, to find parameters that produce low training loss, but they perform poorly in new data. This phenomenon is called overfitting and when it appears, it means essentially that the network learned a portion of the noise and intrinsic variation in the training set. So, when presented with data without these noise characteristics, the model fails to discern the correct patterns in the data.

For the moment, we shall ignore overfitting and we will focus on the way the neural networks are trained. This is achieved by a very well known optimization algorithm, the gradient descent and is applied to neural networks via a very elegant algorithm called *backpropagation* that allows us to compute these gradients efficiently.

2.3.1 Gradients, Jacobians, and the Chain Rule

Before describing the backpropagation algorithm we need to address the rather deep mathematical background. We shall present it though, as simple as it can get for completeness and not educational reasons, because otherwise we would need entire chapters in order to fully understand it. There are a lot of great educational books out there.

The gradient descent optimization algorithm can find a local minimum by taking steps in the direction opposite of the gradient of the function at the current point, in an iterative way. Accordingly, the backpropagation algorithm needs to compute the gradient of the loss function with respect to the parameters (or weights) of the network. Then, through many iterations it can adjust those parameters slightly, in directions that reduce the loss function, to the point when the loss function, and the weights consequently, remain mostly unchanged.

Let's recall the definition of the gradient: given a function $f : \mathbb{R}^N \rightarrow \mathbb{R}$ from a real-

valued vector to a scalar, with $\mathbf{x} = [x_1, x_2, \dots, x_n]$ and $y = f(\mathbf{x})$ then, the gradient of y with respect to \mathbf{x} , denoted by $\nabla_{\mathbf{x}}y$ is defined as:

$$\nabla_{\mathbf{x}}y = \left[\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, \dots, \frac{\partial y}{\partial x_n} \right]$$

Let's take for example the mean squared error loss function L . This function, just as f , produces a scalar value from a real-valued vector, in our case the \mathbf{y} vector:

$$L(y) = \frac{1}{n} \|\mathbf{y} - \hat{\mathbf{y}}\|^2 = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

The gradient of L can be then easily calculated with respect to \mathbf{y} as:

$$\nabla_{\mathbf{y}}L = \frac{1}{n} [2(y_1 - \hat{y}_1), 2(y_2 - \hat{y}_2), \dots, 2(y_n - \hat{y}_n)] = \frac{2}{n} (\mathbf{y} - \hat{\mathbf{y}})$$

The generalization of the gradient to vector-valued functions is called the *Jacobian*. Given a function $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$ and $\mathbf{f} = f(\mathbf{x})$. Then, the Jacobian $J_{\mathbf{x}}(\mathbf{f})$ of \mathbf{f} is given by:

$$J_{\mathbf{x}}(\mathbf{f}) = \left[\frac{\partial \mathbf{f}}{\partial x_1}, \dots, \frac{\partial \mathbf{f}}{\partial x_m} \right] = \begin{bmatrix} \nabla_{\mathbf{x}}^T f_1 \\ \vdots \\ \nabla_{\mathbf{x}}^T f_n \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \dots & \frac{\partial f_n}{\partial x_m} \end{bmatrix}$$

We will also make use of the *chain rule* for derivatives from calculus. If $u = g(x)$ and $y = f(u) = f(g(x))$, then the chain rule states that:

$$\frac{dy}{dx} = \frac{dy}{du} \frac{du}{dx}$$

This works with multivariate functions, like, $y = f(u_1, u_2)$ where $u_1 = g(x)$ and $u_2 = h(x)$, then we have:

$$\frac{dy}{dx} = \frac{dy}{du_1} \frac{du_1}{dx} + \frac{dy}{du_2} \frac{du_2}{dx}$$

We can use the chain rule on vector functions, expressing it in terms of gradients and Jacobians. If we have, for example, $\mathbf{u} = g(\mathbf{x})$ and $y = f(\mathbf{u}) = f(g(\mathbf{x}))$ then we can get:

$$\nabla_{\mathbf{x}}y = J_{\mathbf{x}}(\mathbf{u}) \nabla_{\mathbf{u}}y$$

And if we have multivariate vector functions, like $y = f(\mathbf{u}_1, \mathbf{u}_2)$ where $\mathbf{u}_1 = g(\mathbf{x})$ and $\mathbf{u}_2 = h(\mathbf{x})$, then

$$\nabla_x y = J_x(\mathbf{u}_1)\nabla_{\mathbf{u}_1} y + J_x(\mathbf{u}_2)\nabla_{\mathbf{u}_2} y$$

2.3.2 Iterating Gradient Descent

The gradient descent algorithm (GD), using a loss function $L(y)$, in the k^{th} iteration is:

$$w^m(k+1) = w^m(k) - a\nabla_{w^m} L(y)$$

For the weights in the layer, and:

$$b^m(k+1) = b^m(k) - a\nabla_{b^m} L(y)$$

for the biases in the layer. The variable a is called learning rate and is a hyperparameter, which means it is not learned during the training, but the user defines it before training. This value is a small number (typically around 0.01) that makes the algorithm gradually converge to the minimum. A large learning rate may make the algorithm faster but makes the convergence unstable, with wide variances in the later parts of the training. The Figure 2.7 shows how different learning rate values affect the training process.

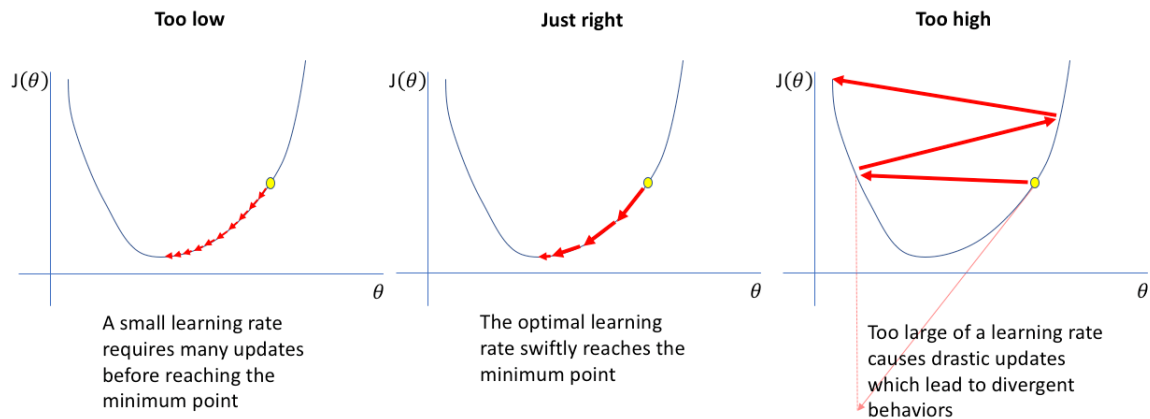


Figure 2.7: Finding the proper Learning Rate.

2.3.3 The Stochastic Gradient Descent Variation

The default gradient descent algorithm requires to pass through all training examples, and then calculate the mean loss which will be used to update the weights and biases via the back-propagation algorithm. This is great for problems with smooth and convex error performance

surface because it moves directly towards an optimal solution, with a few big steps. But because most real world problems are not smooth or convex, most modern frameworks use a clever variation of the gradient descent algorithm *The Stochastic Gradient Descent (SGD)*. The SGD algorithm computes the gradients using a single or a small random sample of the training data, in which case we see it as *Minibatch SGD*.

The *Stochastic* term in its name arises because, instead of going through all the inputs in the dataset one by one and then, update the weights and biases once, we just present one random example and then we update the parameters. On the following iterations we present one by one more examples from the dataset and update the parameters accordingly until all examples in the network are used. None training example is used twice in a training epoch. When the whole dataset is presented the epoch ends and the next one starts which starts presenting examples from the whole dataset again one by one.

The benefit of the SGD is that the trajectory towards the global minimum of the loss function is follows a path with a slow and careful rate, which in fact gives us more chances for convergence. The Backpropagation optimization algorithm in the following section assumes that the Stochastic Gradient Descent algorithm is used, which means a single training example is presented for every weight and bias update.

2.3.4 The Backpropagation Algorithm

The first step to the Backpropagation algorithm is to propagate the input through the network. If we denote as a^m the output of the neuron, for any given layer m and has a layer $m + 1$ as the next layer of the network, the following are true:

$$\mathbf{a}^{m+1} = W^m \mathbf{y}^m + \mathbf{b}^m \quad \text{Neuron value for the layer } m$$

$$\mathbf{y}^{m+1} = f(\mathbf{a}^{m+1}) \quad \text{Activation value for the layer } m$$

The f is the activation function used for the m layer. For the input layer we can use:

$$\mathbf{y}^0 = \mathbf{x} \quad \text{where } \mathbf{x} \text{ is the input data}$$

For the final layer M , we assume:

$$\mathbf{a}^M = W^{M-1} \mathbf{y}^{M-1} + \mathbf{b}^{M-1}$$

$$\hat{\mathbf{y}}^M = f(\mathbf{a}^M)$$

This is the final output to the whole network. The next step is to evaluate the predicted value \hat{y} against the observed y value by using the Loss function. After that, we calculate the Gradients of the loss function for every weight and bias in each layer.

For the weights in the m layer, the gradient is:

$$\nabla_{W^m} L = J_{W^m}(\mathbf{a}^m) \nabla_{\mathbf{a}^m} L \quad \text{applying the chain rule}$$

Just as before: $\mathbf{a}^{m+1} = W^m \mathbf{y}^m + \mathbf{b}^m$ where \mathbf{y}^m are the outputs of the m layer

Taking the Jacobian from the equation before, we get:

$$J_{W^m}(\mathbf{a}^m) = \mathbf{y}^m \quad \text{calculating the Jacobian}$$

The result is the vector \mathbf{y}^m due to the element wise application of the activation function, which makes the Jacobian just a diagonal matrix. The final value of the Loss function's gradient is:

$$\nabla_{W^m} L = \nabla_{\mathbf{a}^m} L \cdot \mathbf{y}^m \quad \text{Final Gradient}$$

A similar set of equations can be applied to the biases:

$$\nabla_{b^m} L = J_{b^m}(\mathbf{a}^m) \nabla_{\mathbf{a}^m} L \quad \text{applying the chain rule}$$

$$J_{b^m}(\mathbf{a}^m) = 1 \quad \text{calculating the Jacobian}$$

$$\nabla_{b^m} L = \nabla_{\mathbf{a}^m} L \cdot 1 \quad \text{Final Gradient}$$

The common part in both equations is often called "local gradient" or "sensitivity" and is expressed as follows:

$$\mathbf{s}^m = \nabla_{\mathbf{a}^m} L \quad \text{Local Gradient}$$

The way they the local gradients are calculated is mathematically more intricate and follows an inverse process in which we can express every local gradient using the local gradient

of the next layer, except for the last layer's local gradient, which we can calculate right after the forward pass calculations. We will present the resulting equations for completeness but if you want the proof it can be easily found in most educational book in this field:

Firstly the local gradient is calculated in the final layer, where it takes the special following form:

$$\mathbf{s}^M = \nabla_{\mathbf{a}^M} L \odot \frac{da^M}{dy^M}$$

The Hadamard product ($s \odot t$) is just an element wise vector product. After that we roll back to calculate the local gradient to each of the previous layer:

$$\mathbf{s}^m = (W^{m+1})^T s^{m+1} \odot \frac{da^m}{dy^m}$$

As we can see the local gradient propagation is following a reverse order:

$$s^M \rightarrow s^{M-1} \rightarrow \dots \rightarrow s^2 \rightarrow s^1$$

It is obvious now that the algorithm's name comes from this order the local gradients take to calculate through the network.

The gradients we finally calculated are used in the weight and bias update process as we saw in the iterating gradient descend section. So they now become:

$$W^m(k+1) = W^m(k) - a \nabla_{\mathbf{a}^m} L \cdot \mathbf{y}^m$$

For the weights in the layer, and:

$$b^m(k+1) = b^m(k) - a \nabla_{\mathbf{a}^m} L$$

2.3.5 Incremental vs Minibatch Learning

The backpropagation algorithm we described above follows the *incremental training* of the Stochastic gradient Descent Algorithm, which means the weights and biases are updated after each input is presented. The cost of these updates is small on its own, but for excessively large datasets these costs add up to a significant increase in training time. Furthermore using only a single training example to update the network parameters can introduce some noise in the network which can ultimately affect the network's ability to make correct predictions. For these reasons, most modern libraries provide the minibatch training modification which is called *Minibatch Stochastic Gradient Descent*.

With the Minibatch SGD, we present the inputs to the network in small batches and we calculate the gradient of the loss function for each one of those inputs. The total gradient of the batch is calculated as the mean of the gradients of the individual input's losses. For a batch size B the above equations would look like these:

$$W^m(k+1) = W^m(k) - \frac{a}{B} \sum_{b=1}^B \nabla_{\mathbf{a}_b^m} L \cdot \mathbf{y}_b^m$$

$$b^m(k+1) = b^m(k) - \frac{a}{B} \sum_{b=1}^B \nabla_{\mathbf{a}_b^m} L$$

Because each input has the same probability to be presented in the network, and as long as the batches are considerably smaller in size than the whole dataset, the Minibatch SGD has around the same accuracy as the normal SGD algorithm but it is trained faster due to the smaller number of parameter updates that happen during each epoch. Moreover, the noise that is introduced with SGD evens out by averaging the gradients of the whole minibatch. Raising the batch size too much makes the method regress to the normal Gradient descent method which takes big steps towards the minimum that may make it unstable for convergence. Using suitable batch size relative to the training dataset size can place this method in a sweet spot between SGD and GD where it can keep the best of both worlds.

2.3.6 Heuristic Modifications of Backpropagation

The mathematics behind the backpropagation algorithm guarantee that the loss function will be minimized when used on linear problems or problems that their error performance surface has only global minimum. When applied to multi-layer networks, however we observe a quite different behavior. Those networks are used for non-linear problems, which create high-dimensional error performance surfaces, in which finding a global minimum can be quite hard. And most problems in the real world tend to be such.

While for linear problems the performance surface of a neural network may have a single minimum point and constant curvature, in high dimensional surfaces we can have many local minimum points, and curvatures that can change widely in different regions of the parameter space. For example in figure 2.8 even with only two weights or variables changing we can already see how intricate the error performance surface can get.

In such cases the algorithm can fall into a local minimum early on, possibly never finding

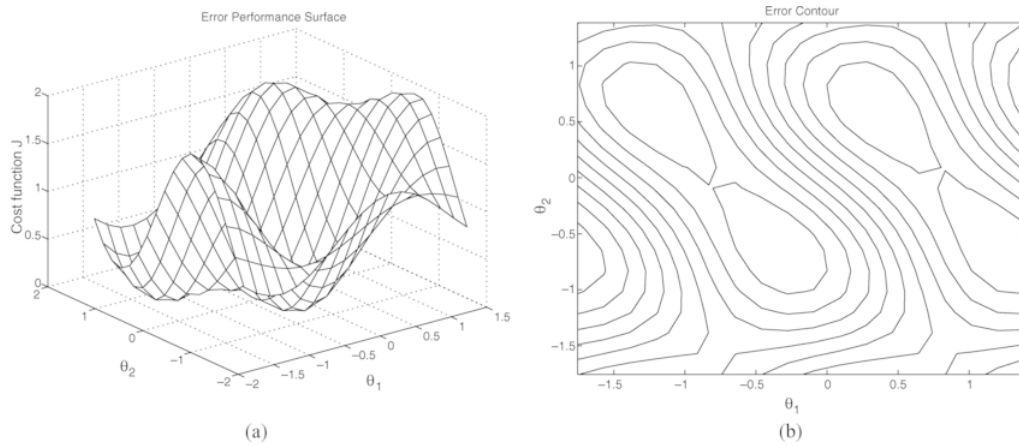


Figure 2.8: An example error performance surface of two variables.

the global minimum, or if the surface gets too "flat" it may take a lot more training iterations that we have available. Initialising the weights with small random values sometimes can help to avoid falling in a local minimum valley, especially if the area near zero is a saddle point where, depending on the starting point, the network can follow completely different trajectories. But this initialization can only get us so far. We need more powerful tools at our disposal to overcome such adversities.

Momentum

From physics we know that Momentum is a physical property of a moving body that enables it based on its mass and velocity, to continue in its trajectory even when external opposing forces are applied, which means overshoot. For example, a car that travels in high speed and then suddenly hits the brakes, the car will skid and stop after a short distance overshooting the mark on the ground when it firstly hit the brakes.

The same concept can be applied to neural networks during training, where the update direction tends to resist change when momentum is added to the update scheme. When the neural network approaches a shallow local minimum it's like applying brakes but not sufficiently to instantly affect the update direction and magnitude. Hence the neural networks trained this way will overshoot past smaller local minima points and only stop in a deeper global minimum.

Thus momentum in neural networks helps them get out of local minima points so that a more important global minimum may be found. Too much of momentum may create issues as well as systems that are not stable, and they may create oscillations that grow in magnitude

as the training goes on. For such cases there needs to be a decay term so this resistance to change can wear out in the later steps of the training process.

The simple mathematical concept of momentum most usually used in neural networks is defined as:

$$W(k+1) = \gamma W(k) - (1-\gamma)C \quad \text{For } 0 \leq \gamma < 1$$

Where C is the change in the value w between two time intervals

For a momentum coefficient γ the update equations would now look like these:

$$W^m(k+1) = \gamma W^m(k) - (1-\gamma) \frac{a}{B} \sum_{b=1}^B \nabla_{\mathbf{a}_b^m} L \cdot \mathbf{y}_b^m$$

$$b^m(k+1) = \gamma b^m(k) - (1-\gamma) \frac{a}{B} \sum_{b=1}^B \nabla_{\mathbf{a}_b^m} L$$

The momentum coefficient is widely used in most modern frameworks and even though it doesn't mathematically guarantee convergence to the global minimum, experimentally it usually performs very well both in accuracy and in training time. Figure 2.9 shows how different the trajectory in the error performance surface is when momentum is applied.

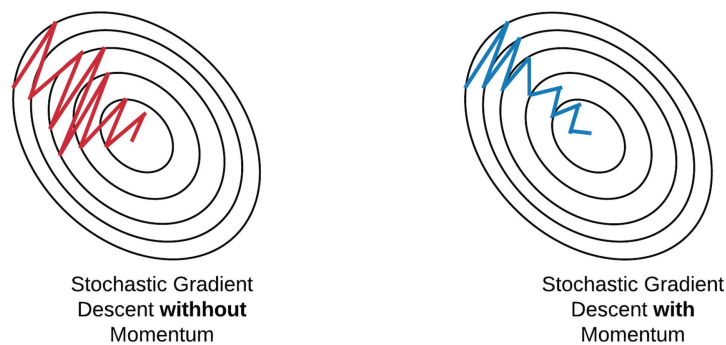


Figure 2.9: The impact of Momentum in Training.

Variable Learning Rate

We understand so far that the learning rate works like the speed with which the network converges to the minimum. If we raise it the algorithm takes bigger steps towards the loss function's gradients, so why do we keep it usually so low? If the algorithm is heading towards convergence a big learning rate will create big steps that may throw us away from the global minimum. if on the other hand we are inside a big flat surface a small learning rate will waste

many epochs to overcome it, where it would be very important to use those extra epochs near the global minimum for better chances of convergence. For this reason we tried to find a suitable enough learning rate which is neither too small nor too large.

Ideally we would want the learning rate to rise in flat surfaces, and decrease when the slope starts becoming steeper. A lot of techniques have been tried for the Variable Learning Rate approach which usually follow some design rules that modify the learning rate according to some performance metrics that are calculated during the training process. These contain but are not limited to:

1. If the squared error increases by more than a set percentage ζ (typically one to five percent) after a weight update, then the weight update is discarded, the learning rate is multiplied by some factor $0 < \rho < 1$, and the momentum coefficient, if used, is set to zero.
2. If the squared error decreases after a weight update, then the weight update is accepted and the learning rate is multiplied by some factor $\eta > 1$. If the momentum coefficient γ is zero it is reset to its original value.
3. If the squared error increases by less than ζ , the weight update is accepted but the learning rate remains unchanged. If the momentum coefficient γ is zero it is reset to its original value.

If used correctly, with this technique we can gain a little performance than just using momentum, but their excessive use of user-defined hyperparameters makes this technique easier to decrease performance than increase it. And in most cases requires a lot of trial and error to find the perfect combination on those hyperparameter values, which consumes much more time than we often have in disposal. It is only recommended for datasets with small training times and even for those cases for optimising prediction performance.

The reason we refer to Variable Learning Rate, even though it is rarely used, is because it was the source of inspiration to the modifications we introduce to the *Sparse Evolutionary Training (SET)* algorithm presented in Chapter 3.

2.3.7 Tensors

So far we imagined the inputs to a neural network as a one dimensional array of vectors. But we can equally use inputs of higher dimensions. Then if inputs can be of a higher di-

mension so can the outputs be presented as such higher dimensional structures. Similarly we imagined each layer of a neural network as columns of neurons but no one forbids us from organizing them in a higher dimensional manner. For example a digital image is defined by an array of pixels with some length and width. Pixel is represented by three real numbers, the intensity of each one of the three primary colors, red, green and blue. So, We can imagine the image as an 2D array of three dimensional vectors. The natural generalization of vectors and matrices is the *tensor*, which is an n -dimensional array of scalars.

The backpropagation algorithm we described works for vectors but was not for high-dimensional tensors. In such cases we can use a simple trick, the *flattening* of the tensor which unrolls the tensor to a nested and ordered collection of vectors, this way we can now address problems like image classification and many more, with the same techniques and tools, just by altering a little how we imagine and we represent data.

2.4 Regularization

As we have previously stated, the goal of the gradient descent algorithm and of the training of any predictive model in general is to minimize the errors in the predictions it makes. This is quantifiable by minimizing a loss function, over the data we use for training, reaching higher and higher accuracy in each step. The real goal however, is to create a model that can perform as well as to new data of which the model is unfamiliar with. We would ideally want to minimize the loss function on those new data but obviously with this being impossible, we settle for the loss in the training data as a performance index.

Many times though, the model's ability to discern patterns in the data becomes a little too capable for its own good. This means that the model can recognize patterns where there shouldn't be any, usually created due to the noise contained to the data or due to the lack of enough data to make the model representative of real world scenarios. This problem as we have previously mentioned is called overfitting, and we can recognize it when a predictive model can achieve higher accuracy in training data than on a test set comprised of new unprecedented data. The usual practice is to split the dataset in a training set and a test set. We use the training set for training, and we evaluate the model based on its performance over the test set. If it appears to be a great drop in accuracy on the test set comparatively to the training set, we know the model has overfit. On datasets that have independent examples, we split the

dataset into the two subsets following a defined ratio of usually 80:20 percent or 70:30 for sufficiently large datasets. It is advised to use a random sampling method for this process to eliminate any bias in the data. We have to be careful, however when dealing with sequence-learning problems, such as time-series, because in such cases the order the examples have in the data contains useful information for successful forecasting.

The problem of overfitting exists for all machine-learning models. It is more prevalent neural networks though, because they have a lot more parameters than other models, and their design leaves more space for overfitting to occur. A lot of techniques have been developed to help us reduce overfitting coming together under the umbrella term *model regularization*. We usually trade some of the model's accuracy on the training dataset for better accuracy on test data, making the model's predictive capabilities more generic. The most popular of those techniques, we are going to discuss in this section.

2.4.1 Norm Penalties

The goal of gradient descent is to minimize the loss function to a global minimum. Many times however, a single minimum doesn't exist, or it is not clear and is very closely obfuscated by a lot of local minima nearby. So it is usual to for a model to learn a local minimum, but even between one another not all local minima generalize just as well. Furthermore, many times models that contain small absolute values are able to generalise better than models with very large weights.

So, we can enforce our model to keep relatively small weight values by introducing a penalty term to the loss function. Let's say that \mathbf{w} is the weight matrix in the model and L_0 the original loss function we usually use. We define the regularized L_2 -norm loss function as follows:

$$L = L_0 + d\|\mathbf{w}\|^2$$

Using the L_2 -norm of the weights as penalty we shrink large weight values while keeping small ones relatively unchanged. The parameter d , is a hyperparameter usually referred to as weight decay, help balance the trade off between minimizing loss and penalizing weights. It usually has a small value usually lower than 0.1 depending on the dataset. Even if we prefer the weights to have small values weights that have strong predictive ability still need to be able to contribute as much as they can, and to give them this opportunity the weight value

needs to be sufficiently low.

Moreover, instead of the L_2 -norm we could penalize the weights based on the L_1 -norm as follows:

$$L = L_0 + d \sum_i |\mathbf{w}|$$

In practice the L_2 -norm is generally more preferred and works for most applications. The L_1 -norm tends to produce models where many of its weights are zero, and this usually helps in feature selection by eliminating features that are not important

2.4.2 Dropout

Dropout is a technique that randomly deletes a fraction of the neurons that aims to reduce overfitting. When a node is deleted, all connections from and towards that neuron are also deleted. Then we perform the forward propagation and backpropagation steps of the algorithm for the minibatch, calculating gradients and updating weight and biases using the modified network. After the minibatch finished process we restore all the deleted neurons and weights and, after selecting the new minibatch, a new random subset of nodes is deleted, repeating the process. The fraction of the nodes that are deleted each time is a hyperparameter, and is called *dropout rate*.

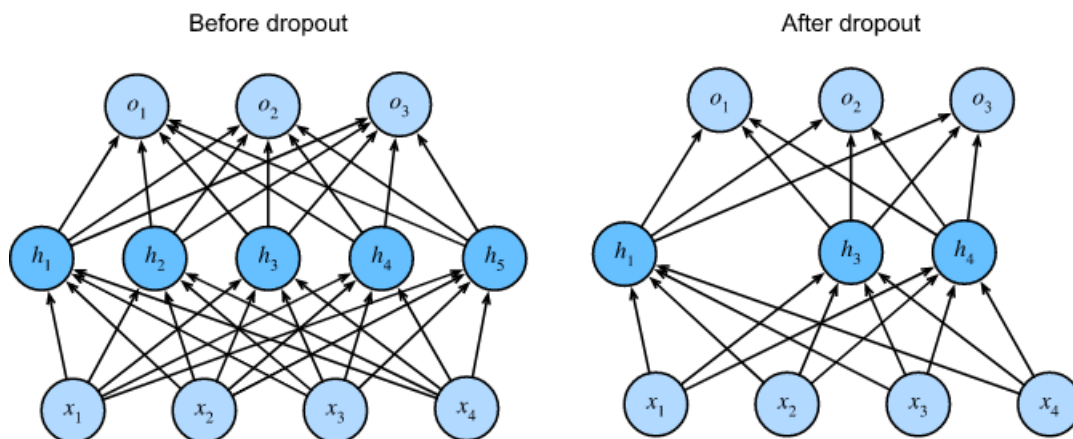


Figure 2.10: A Neural Network before and after applying Dropout.

Several hypotheses have been put together on why dropout can reduce overfitting. One of the most convincing arguments is that it enables a single neural network to operate as a collection of smaller sub-networks that are trained at the same time, with each one of them de-

veloping its own pattern recognition rules that are aggregated to a single result s the weighted average of the solutions of the constituent sub-networks.

2.4.3 Dataset Augmentation

The power of machine-learning comes from finding patterns in data and for the best results we need the most data we can have, or we can handle. It makes sense to have better performing models if we can provide even more training data. Big datasets tend to lead to less overfitting, and statistics support this view. More training data means outliers and noise becomes less frequent and the model’s ability to generalize becomes better and better.

On situations where the training data available is limited, we can create additional synthetic examples by applying transformations and covering more cases. It isn’t always applicable and the way we create these additional examples need to be well studied and scientifically supported in order to retain properties of the real data and to not simply add noise and biases. It is hard to explain how this is done because is highly depends on the form of the data.

The easiest example though, is from unstructured data such as images. Let’s take for example the images of numbers. In such cases data augmentation can create new images where the numbers have a slight positional displacement, slight rotations, changes in colors and brightness, blurriness and so on (Figure 2.11). These new images are valid for training because in the real world a model should be able to recognise the numbers even through such conditions.

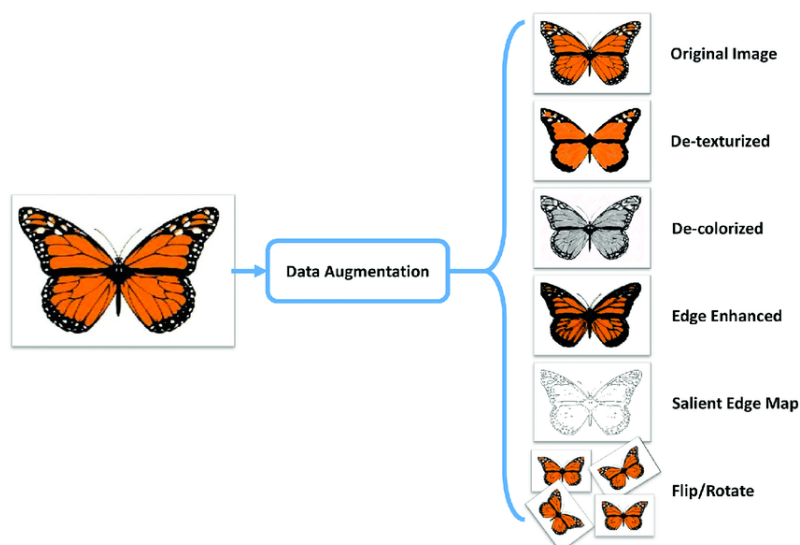


Figure 2.11: How Data Augmentation helps us generate more data.

Chapter 3

Sparse Neural Networks

As we have previously seen the parameters of a neural network can be expressed using matrices and vectors. A matrix is called sparse when most of its elements are zero. Consequently, when we refer to sparse neural networks, we can consider them as networks with most of its weights as zeros. Using modern computing, sparse matrices can be represented in computers with much less space than their normal matrix representation. This way we can efficiently compress neural networks for use on small IoT devices. The problem is that in most real world scenarios, good performing neural networks often usually have a great number of parameters, usually around several hundred thousands or a few millions of weights. These neural network sizes makes their use prohibiting on such small devices.

The most common ways to compress neural networks so far, use techniques that usually belong into one of the following categories:

1. We can train a rather small network that operates sufficiently well and then, after training, compress the network using techniques that prune its parameters.
2. Apply quantization methods that reduce memory footprint by transforming the weights into other units that usually introduce some error due to approximation.
3. Or use knowledge distillation methods that involves training a big model with top performance and then using its predictions to train a smaller one.

In all those cases the obvious cost is time, because except from the usual model training, more work and consequently more time is required in order to compress such networks for smaller devices. It would be really convenient if we could train and compress a neural network at the same time while also keeping its predicting power the same. Wait a minute...

3.1 Related Work

The literature on speeding up neural network training has a long history and it dates back to the late '80 – early '90. We divide this research into different categories of techniques based on their characteristics. The listing is by no means extensive, but it is an effort to cover the most representative and/or more recent members of each family.

One of the first categories of acceleration methods includes members that meant to replace the traditional gradient (steepest) descent optimization method. Steepest descent is based on a first order Taylor series approximation of the performance function (mean square error) and it is very slow. Therefore, methods based on second order Taylor series were investigated, such as Newton's method and particular adaptations of it, e.g., the Levenberg-Marquardt algorithm [4] which is much faster. Other algorithms that departed from the first order gradient concept, are those based on conjugate gradient [5], and the similar in spirit quasi-Newton method of Broyden-Fletcher-Goldfarb-Shanno (BFGS), along with its variations, e.g., L-BFGS [6]. Recently, fast optimizers have been proposed such as Adam, Adadelta, Nadam [7].

Another category for accelerating neural training is those based on adopting variable learning rates. For instance, the delta-bar-delta method [8] assigns to each network parameter its own learning rate that varies at each iteration; similar in spirit is the SuperSAB method [9].

The recently introduced technique of dropout [10] constitutes the founding member of a new category, which accelerates training by randomly dropping units during training. Several adaptations of it have been proposed for various applications and various neural architectures, e.g., [6]. Similar in spirit, are the methods which compute only a subset of gradients during back propagation, e.g., meProp [11] [12] which prunes neurons based on how many times a neuron is updated by the back propagation.

The category of hardware-based accelerators is gaining significance and hardware architectures such as FPGAs [13], multicore CPUs [14], TPUs [15] are increasingly used for neural training and inference.

A very recent and intriguing piece of work [16] suggests that only a small part of a neural network is responsible for carrying out accurately a particular prediction, and thus if we can find a subnetwork that can achieve it and then train only this, then we can gain significant speedups in training. We will explain the Lottery Ticket Hypothesis in more detail in the following section.

The category of methods that are related mostly to the present work are those based on neural topology sparsification. Dropout can be considered one of those in the sense that removing a neuron is equivalent to removing all its connections. However, there have been proposed in the literature methods which specifically prune the connection between the neurons. For instance, the works [17] [18], and [19]. However, these linkage sparsification techniques do not aim at mimicking the topological structure of real neural networks, but are mainly based on eliminating close-to-zero weighted connections. The most closely related work to ours is the SET procedure of Mocanu et al. reported in [20], in which they start from a completely unstructured topology basis, i.e., purely random network, having a specific, stable number of connections removed in every epoch, which is not efficient enough, especially when we are in the last epoch and the model is almost trained.

Overall, the methods developed in this work can be used in conjunction with any member of any category described above to accelerate training and reduce the memory footprint.

3.2 The Lottery Ticket Hypothesis

Thanks to the work of Frankle et al. [16], a new wave of research starts in the neural networks field that challenges the way we have been designing neural networks so far. The process of training machine learning models is one of the areas in which data scientists often face the compromise between theory and the constraints of real world solutions. More often than not, a neural network architecture that seems ideal for a specific problem can't be fully implemented because the cost of training would be prohibiting. Typically, the initial training of a neural networks requires large datasets and days of expensive computation usage. The results are very large neural network structures with a lot of neurons densely interconnected through numerous layers the result of which is an astronomical number of trained parameters. This structure often needs to be subjected to optimization techniques to remove some of the connections and adjust the size of the model.

The question that bothered AI researchers for decades is weather we actually need all those large neural network structures to begin with. Those dense architectures are likely to perform the wanted task but the cost might render them inapplicable in a real-world scenario. The question that essentially gave birth to the Lottery Ticket Hypothesis is weather we can perform the initial task with smaller more sparse architectures.

Using an analogy from the gambling world the training of machine learning is often compared to winning the lottery by buying most of the tickets.

The Lottery Ticket Hypothesis. *A randomly-initialized, dense neural network contains a sub network that is initialized such that—when trained in isolation—it can match the test accuracy of the original network after training for at most the same number of iterations.*

This small sub network is often referred to as the winning ticket.

If this is true, it means we have a great confidence that at least one sub network exists, which can have the same test accuracy as the original, and sometimes even greater than the original, with only a small fraction of the initial parameters used. This may revolutionise the way we design neural networks, unlocking us a path to more powerful network architectures that utilize more efficiently the resources they use.

This hunt for the winning ticket can't be done without a systematic method though. It is indeed helpful to conduct random search in the subspace of dense networks to finding equivalent sub-networks, but if not handled carefully the efforts may be fruitless. A sparse training algorithm, from one hand should be able to search a wide range of combinations, but from the other should give the testing sub-networks some time to reach maturity or to show what they can achieve. We can't possibly expect from any random sub-network to perform at peak performance when compared to its fully connected dense counterpart, from only a couple of training epochs.

From the other hand we often don't have the luxury to test random sub-networks indefinitely, surpassing the training time of their dense counterpart. So, summarize we need algorithms that can do the following:

1. Searches a wide range of sub networks derived from their fully connected counterpart.
2. Gives enough time and space for these sub-networks to reach maturity.
3. Find some that can perform at the same or higher level of accuracy than the original.
4. And finally, their size should be considerably smaller than the original, which means the resources required are considerably less.

At the same time, it would be of great importance if such algorithms could keep the total training cost equal to or even better less than that of their dense counterpart. Time is always of great importance and in real world situations any time saved from the training of a neural

network could be utilised for fine-tuning the resulting model before lunch, or earlier lunch that could lead to competitive advantages.

3.3 A Small Reference to Network Science Concepts

Any neural network among other properties like exhibits characteristics of graphs from classical network theory. Neurons can be related to vertices and weights can be related to edges, making the study of its resulting topologies an interesting field of research. Even for fully connected neural networks the resulting topology after the training changes a lot, meaning that many training algorithms and network designs produce topologies that can be correlated to those found in graph theory. In the figure 3.1, from the Journal of Computing in Civil Engineering [21], we can find visualizations of the network structured we will explain below.

3.3.1 Regular Graphs

In regular graphs all nodes have the same number of links with other nodes. The resulting network structure can be represented by a lattice with vertices of degree k , which are named k -regular graph. In Neural Networks, fully connected layers can be considered bipartite regular graphs, with the nodes of each layer as the graph's disjoint sets.

3.3.2 Small-World Networks

In small-world networks there is a small likelihood for any two given nodes to be neighbors but their neighbors have a bigger chance for them to be neighbors. This means that every node can be reached from any other node in a small number of hops or steps. The number of hops required is significantly smaller proportionally to the number of nodes that exist in the network. Specifically, a small-world network is defined to be a network where the typical distance L between two randomly chosen nodes (the number of steps required) grows proportionally to the logarithm of the number of nodes N in the network, that is:

$$L \propto \log N$$

while the clustering coefficient is not small. Some examples of Small-World networks

include social networks, electric power grids, food webs, the underlying infrastructure of the internet, brain neurons and websites.

3.3.3 Scale-Free Networks

Scale-Free Networks have degree distributions that follow a power law, at least asymptotically. The characteristics of the network are independent of the size of the network, which means that when the network grows the underlying structure remains the same. Although many real networks gives us a hard time classifying them between small-world and scale-free networks, because there is a gray zone in their border, the truth is that inherently scale-free networks in nature are rare [22].

3.3.4 Erdős–Rényi Random Graphs

Random graphs is the general term used when applying probability distributions over graphs. It may indicate a random shape or structure or a random process that generates them. Among them the Erdős–Rényi random graph model is almost exclusively referenced in any mathematical context and refers to two closely related models for creating them.

- In the $G(n, M)$ model a graph is chosen uniformly at random from the list of all possible graphs that have n nodes and M edges.
- In the $G(n, p)$ model a graph is generated by connecting nodes randomly. An edge between two nodes exists in the graph with probability p and is independent from any other edge. The distribution degree for any particular vertex is binomial where n is the total number of vertices. But as $n \rightarrow \infty$ and $np = \text{constant}$ the distribution becomes Poisson.

The second model is preferable for computational experiments as it is easier and faster to use. These graphs are often used in probabilistic methods to research properties in graphs. In practice random graphs have a lot of similarities with small-world graphs but with a key difference. Small-World networks have significantly higher clustering coefficient than random graphs[23].

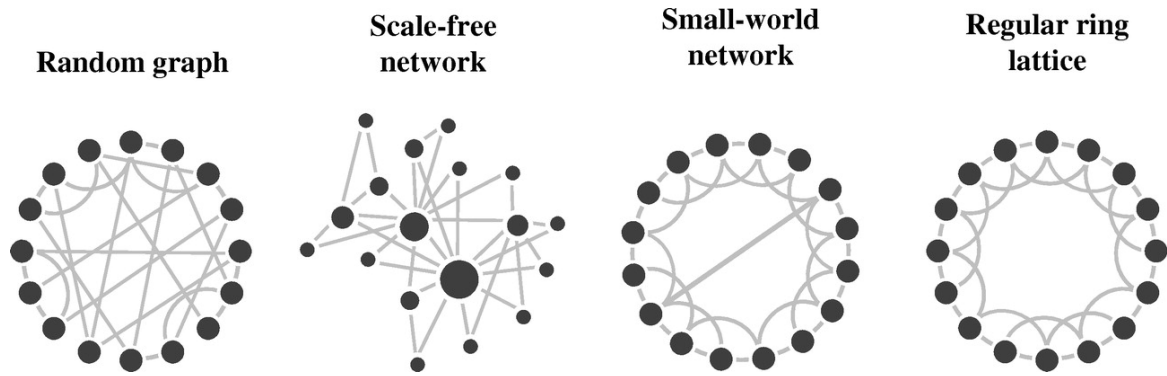


Figure 3.1: Different kinds of Network Structures

3.4 Weight Masks

After analyzing the underlying components of a neural network in Chapter 2, we can understand that the most applied, if not the best, way to represent the interconnections in a neural network is with the weight matrices between each layer. Those are the connections that we ultimately want to prune in order to achieve a sparse network topology. The most intuitive way to approach this task is with *Weight Masks*. A weight Mask can be imagined simply as a matrix of equal dimensions as the matrix that represents the network topology. A Weight Mask contains only zero and ones which when applied to a weight matrix it conducts element-wise multiplication to the elements of the weight matrix. A weight or element in the weight matrix is pruned, if it is multiplied with zero, while on the other case it remains as it is. This way we can determine which weights will be pruned and which weights will stay the same.

We can understand better how a weight mask is applied with a visual experiment:

$$W = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}, M = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}, \text{ Then } W \odot M = \begin{bmatrix} 0 & 4 & 0 \\ 0 & 0 & 0 \\ 3 & 0 & 9 \end{bmatrix}$$

The symbol " \odot " represents the Hadamard product that we have previously mentioned, which is the element wise product between matrices.

3.5 Sparse Evolutionary Training (SET)

The name for the SET algorithm suggests that the algorithm uses a sparse training method that takes an evolutionary algorithm design. SET draws inspiration from the natural simplic-

ity of evolutionary approaches, an area with great interest and a lot of research for many years. While biological neural networks has been shown to have sparse topologies [24][25], Mocanu et al. argue that artificial neural networks have not evolved following these topological features [20][26]. With motivation from the huge parameter size of fully connected networks and the fact that the parameters of trained neural networks follow distributions that places most of the weights around zero [27]. The state-of-the-art approach in machine learning pursues sparse topological connectivity after the training phase as a method to fine-tune the networks inference capabilities [27], with severe time cost on the training phase. The need for reasonable training times motivated Mocanu et al. to follow a more integrated approach to pruning and training.

The SET procedure involves taking a fully connected neural network before training and pruning until it reaches a sparse topology with a controlled and predefined sparsity level. Then, following a random process that prunes existing connections and introduces new connections to the network, it reaches a more structured topology like scale-free or small-world networks.

3.5.1 The SET Procedure

The Set procedure is depicted through a pseudo-code in Algorithm 1. In a hidden layer k of a neural network, the neurons can be collected in a vector $\mathbf{h}^k = [h_1^k, h_1^k, \dots, h_n^k]$, with n the maximum number of neurons for this layer. A neuron from the hidden layer is connected to the neurons of the previous layer namely \mathbf{h}^{k-1} . These connections between the two layers can be expressed in a weight matrix $\mathbf{W}^k \in \mathbb{R}^{n^{k-1} \times n^k}$. In dense networks this weight matrix is dense but with the SET procedure, the weight matrices \mathbf{W} of a sparse connected (SC^k) layer start as an Erdős–Rényi random graphs, in which the probability that a connection exists between the neurons h_i^{k-1} and h_i^k is given by:

$$p(W_{ij}^k) = \frac{\varepsilon(n^k + n^{k-1})}{n^k n^{k-1}} \quad (3.1)$$

Where epsilon $\varepsilon \in \mathbb{R}^+$ is a parameter of SET that controls the sparsity level. With the default value of $\varepsilon = 20$ that the original paper uses we can achieve a sparsity level of around 95% to 97% in most network configurations with at least three hidden layers. This can be translated to a compression ratio of around $\times 20$ to $\times 40$ times depending on the initial network size the value of ε used. The amount of connections the resulting network has, is not

always the same as the algorithm that makes the network sparse is a random process. But with some tweaking one can achieve the desired sparsity level or a suitable number of resulting parameters in the sparse network.

The SET procedure however does not stop there. The resulting sparse networks as is were performing significantly worse than their dense counterparts as shown in the original paper, named FixProb [20]. Referring to the Lottery Ticket Hypothesis, the procedure just isolates a sub-network from the original dense graph which then trains for the full number of epochs. This is just a lottery ticket. To find the winning tickets we would need to conduct an innumerable number of experiments. This is where the synaptic remodeling phenomenon kicks in, in an algorithm called weight evolution. This is illustrated in Figure 3.2 which we used from the original paper [20].

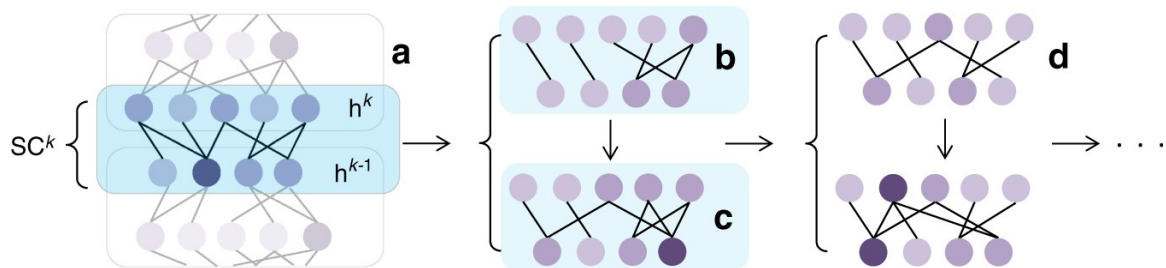


Figure 3.2: An illustration of the weight evolution procedure. For each sparse connected layer, SC^k (a) of a Neural Network, at the end of a training epoch a fraction of the weights closest to zero, are removed (b). Then a number of weights is randomly added that is equal to the number of weights removed (c). The process is repeated at the end of each epoch for the rest of the training phase (d).

The evolutionary part in the name of the SET algorithm comes from the fact that at the end of each epoch a fraction of the weights are pruned and new connections are formed. A fraction ζ of the weights closest to zero are pruned, with zeros in the weight mask, and an equal amount of new connections are formed with the weights initialized as if they had started their training now. Because these weights are closest to zero, we expect that their contribution is minimal and their removal will not induce notable changes in the model's performance [28][18]. Because the algorithm always reconnects the same amount of weights pruned, the total number of parameters in the network stays the same and known from the start. But the weight evolution is turned off at the last epoch so that the final model does not have freshly initialized and untrained parameters that would only be a burden to the network.

Algorithm 1 Pseudo-code of the SET procedure

```

1: Procedure SET (Sparsity level  $\epsilon$ , Rewiring percentage  $\zeta$ )
2: Initialize Fully Connected Neural Network model;
3: for  $i$  in range(1, layers-1) do
4:   Create Sparse Connected layer (SC) with an Erdős–Rényi topology given by  $\epsilon$  and Eq 3.1;
5:   Replace FC with SC layer created;
6: end for
7: initialize training algorithm parameters;
8: for  $i$  in range(1, epochs) do
9:   Feed_Forward();
10:  Back_Propagation();
11:  for  $i$  in range(1, layers-1) do
12:    Remove a fraction  $\zeta$  of the smallest positive weights;
13:    Remove a fraction  $\zeta$  of the largest negative weights;
14:    Add new random connections equal to the amount removed;
15:    if  $i == \text{epochs}$  then
16:      break;           // Do not add new weights on the last epoch
17:    end if
18:  end for
19: end for

```

One could argue whether two epochs are enough for the previous batch of reconnected weight to reach "maturity" and we also wanted to and tested this possibility. The turn-off point was modified in our experiments but while it remained a small number whether 1, 5, or 10 in the scope of 500 epochs the changes were statistically insignificant. But any bigger turn-off point, i.e 50 or 100, would negatively impact the networks performance as the evolutionary search for better parameter was now noticeably less. So we opted to keep the default turn-off point to utilize the maximum evolutionary search we could.

It may be easier to interpret this procedure if we make an analogy of the neural network model to an entity which evolves over time. Strong and more important connections are left intact while weaker and less important connections are pruned, like the selection phase of natural evolution and the new connections added correspond to the mutations that happens during natural evolution over the years. As shown by Mocanu et al. the SET procedure has a noticeable success in keeping the model's accuracy equal and some times, even greater than

its dense counterpart while compressing the network significantly [20].

3.5.2 The Biological Background

Unbeknownst initially to Mocanu et al. [20] a phenomenon exists in biological brains that draws some similarities between the algorithm and biological brains, called synaptic remodeling which happens during sleep.

The information received by an animal is processed by their brain and throughout the course of its awake phase causes the synapses in the brain to strengthen. To counterbalance these changes the synapses should weaken during the sleep phase. De Vivo et al. through experimentation observed a substantial decrease in the interface size of mouse brains after sleep [29]. The largest relative changes was observed among weak synapses, while stronger ones remained more stable. This suggests that there is a distinction between weaker but more numerous synapses and the stronger but considerably less ones. Furthermore, Diering et al. found that synapses undergo changes during sleep/wake cycle which are observed by the concentration increase of some particular gene during the wake phase that differentiate between weaker and stronger synapses and trigger neurological changes in them that when the sleep cycle begins trigger synapse weakening [30]. This means that one of the core functions of sleeping is, among others, to re-normalize the synaptic strength increase during the wake phase.

Keeping the analogy, we could relate the wake phase of an animal with a training epoch on a neural network, that results in the weight update which seems like the synapse strengthening that occurs during the day. Then, the weight evolution phase of the SET Procedure can be related to the sleep phase in animals that activates the synapse weakening procedure. Even though synapse remodeling is a very complex procedure, the SET algorithm applies a simple but understandable abstraction that yields important results nonetheless.

Chapter 4

Proposed Techniques

The SET procedure as we have seen, uses a constant zeta parameter equal to 0.3 or 30% that keeps the evolution rate unchanged throughout training. The weight evolution does not differentiate between first and last epochs, even though it is obvious that a neural network's behavior and stability changes as it proceeds through the training. Drawing inspiration from biology, where the synapse remodeling rate in biological brains changes throughout an animal's life, we tried making this weight evolution rate change during training.

The methods we propose will retain the closest to zero weight pruning algorithm but with a variable zeta parameter that covers two main categories. Firstly, making the hyperparameter zeta follow a genuinely declining function of a linear one and a much steeper exponential one, we simulate the aging factor in biological brains that may help the neural network in our case 'mature' in a much smoother rate. Secondly, causing the parameter zeta to oscillate, we display a seasonal pattern that tries to simulate the seasonal changes in an animal's brain that occur during a chronological year.

The truth is, that a lot of functions can fall into these categories, but the methods we implemented for our experiments were designed to introduce as few new hyperparameters as possible but with decent performance. The idea of making the parameter zeta a variable also draws inspiration from variable learning rate techniques, a concept that has already existed for many years already. From the scenarios we tested we present three variations to the base SET algorithm that are representative enough of our cause. Two where zeta follows a genuinely monotonous variation and another where zeta exhibits oscillations. The whole design drew inspiration from the variable learning rate techniques with which it has many similarities. The variable learning rate is analogous to the variable zeta we try to implement. Our

implementations was based on the original author's custom implementation and their Keras implementation but the latter was implemented in a way that caused the sparse methods to run a lot slower than their dense MLP counterpart. A year or two after the original paper's release, Keras introduced the Callbacks API and using it we constructed an implementation that is more efficient and close to the desired concept.

4.1 Motivations from Biology

In the original paper Mocanu et al. admits that the algorithm used for removing and re-connecting links is very simple and there may be better ways to approach it. The parameter zeta (ζ) used in the SET procedure stays the same throughout the training phase at 0.3 or 30%. The weight evolution does not differentiate between first and last epochs, even though it is obvious that a neural network's behavior and stability changes as it converges towards the minimum over the training. Drawing inspiration from biology, where the synapse remodeling rate in biological brains changes throughout an animal's life, we tried making this weight evolution rate change during training by making the parameter zeta a variable. This may be a good direction to move sparse evolutionary training techniques one step further.

Let's approach this from a different angle. SET was inspired from the synaptic remodeling phenomenon that causes neurons to strengthen during the wake phase and causing them to weaken during the sleep phase, solidifying the knowledge gained throughout the day. In that manner, the epochs of a training session can be similarized to days in the life of a biological brain. But biological organisms age through time and that can induce changes in their neurological parts as well. The brains of young animals have considerable differences in their anatomy than more mature ones. These differences appear not only to the overall number of neurons in their brain but also in the thickness those synapses have and to the rate they reform new neurons as well (a process called neurogenesis). Older animals may have less neurons than younger ones and way smaller neurogenesis rate, but their experience makes them wiser and more knowledgeable, at least until a certain age. This may be indicated through the increased thickness in their synapses that occurs gradually over the years and until they reach adulthood.

4.2 Taking the logic behind SET one step further

To speak with more algorithmic terms, if we change the zeta value throughout training we can simulate the changes to the new neuron formation rate in the brain. The methods we proposed cover two main categories. Firstly, making the hyperparameter zeta follow a genuinely declining function of a linear one and a much steeper exponential one, we simulate the aging factor in biological brains that may help the neural network in our case 'mature' in a much smoother rate. Secondly, causing the parameter zeta to oscillate, we display a seasonal pattern that tries to simulate the seasonal changes in an animal's brain that occur during a chronological year.

We would also like aligning our methods, to the categories Hoefler et al. defined in their research [31] with most of these categories applying to the Mocanu et al. SET procedure as well. Among different compression techniques out there, our methods belong into the model sparsification category because they are the result of applying an Erdős–Rényi random graph to a much bigger initial dense model. This also explains why our methods also fall into the sparse training category. They start from a sparse model which they update during training by modifying its underlying structural topology. During training our methods prune and reconnect parameters at a variable rate that changes during the training process. When choosing candidates for removal, we use a data-free selection based on weight magnitude. And finally, when we need to regrow the network we apply a random regrowth technique that chooses new weights to reconnect randomly. It is of great importance to organize sparse methods for understanding them easier and for easier evaluation that may lead to a more structured and focused research.

4.3 Linear Decreasing Variation (LDV)

In a similar fashion to the variable learning rate methods used by Abbas et al. [32] we applied a linear decreasing curvature to the parameter zeta that is adjusted to the maximum number of epochs. We call this method Linear decreasing Variation (LDV), with the relevant pseudo-code in Algorithm 3 as seen in Algorithm 2. The following equation depicts this procedure:

$$\zeta_i = \zeta_{min} + (\zeta_{max} - \zeta_{min}) \times \frac{max_iter - curr_iter}{max_iter}$$

We set a maximum zeta value of 0.3 and a minimum one of 0.01 so that it starts with the maximum zeta value at the start of the training and linearly decreases to the minimum zeta value over the entire training process.

Algorithm 2 Linear Decreasing Equation Pseudo-code

```

1: Procedure LDV (Sparsity level  $\varepsilon$ ,  $\zeta_{max}$ ,  $\zeta_{min}$ )
2: Initialize Fully Connected Neural Network model;
3: Sparsify_network( $\varepsilon$ );
4: Initialize training algorithm parameters;
5: for  $i$  in range(1,epochs) do
6:   Feed_Forward();
7:   Back_Propagation();
8:   Apply_LDV_rule( $\zeta_{max}$ ,  $\zeta_{min}$ ,  $\kappa$ ,  $i$ , epochs);    // See LDV equation
9:   Weight_evolution( $\zeta$ );
10: end for

```

4.4 Oscillating Variation (OSV)

Here, once again following the steps of Abbas et al. [32] we applied an oscillating variation curve to the zeta parameter, adjusted to the maximum number of epochs. We call this method Oscillating Variation (OSV). Just like the LDV method we need a minimum and maximum zeta value that is defined as 0.3 and 0.01 respectively. The following equation shows how this is done:

$$\zeta_i = \frac{\zeta_{max} + \zeta_{min}}{2} + \frac{\zeta_{max} - \zeta_{min}}{2} \times \cos\left(\frac{2\pi \cdot curr_iter}{T}\right)$$

$$\text{Where: } T = \frac{2 \cdot max_iter}{3 + 2k}$$

In the definition of the period T we use the parameter k to control the frequency of the oscillations and was set equal to 1 for our experiments after performing a small random search. The relevant pseudo-code can be seen in Algorithm 3

Algorithm 3 Oscillating Variation Pseudo-code

```

1: Procedure OSV (Sparsity level  $\varepsilon$ ,  $\zeta_{max}$ ,  $\zeta_{min}$ , Frequency  $\kappa$ )
2: Initialize Fully Connected Neural Network model;
3: Sparsify_network( $\varepsilon$ );
4: Initialize training algorithm parameters;
5: for  $i$  in range(1,epochs) do
6:   Feed_Forward();
7:   Back_Propagation();
8:   Apply_OSV_rule( $\zeta_{max}$ ,  $\zeta_{min}$ ,  $\kappa$ ,  $i$ , epochs);    // See OSV equation
9:   Weight_evolution( $\zeta$ );
10: end for

```

4.5 Exponential Decay (EXD)

Drawing inspiration from the field of finance we used a simple exponential decreasing function that makes the parameter zeta decay each epoch by a constant fraction called interest, as depicted in the following equation:

$$\zeta_i = \zeta_0(1 - interest)^{curr_iter}$$

where ζ_0 is the starting zeta value equal to 0.3 just as the SET procedure and the interest was chosen to be equal to 0.01 for our experiments after trying several values, so that the decay in the zeta parameter was neither too steep nor too gentle that it didn't approach the zero for the number of epochs we used. We call this method Exponential Decay (EXD) and the relevant pseudo-code is depicted in Algorithm 4.

4.6 The Importance of Training Speed

An important view of this project is addressing the training speed of sparse methods and how they compare to fully connected ones. Decreases in memory footprint may not be the only reason sparse methods should be considered. It is also of great importance to find whether SET and our variations can help reduce training time costs or at least stay around the same as their dense counterpart while at the same time keeping the model's accuracy the same, if not greater. Depending on the implementation, this isn't always obvious. Many frameworks today are not designed to handle sparse matrices and as sparse methods gain

Algorithm 4 Exponential Decay Pseudo-code

```

1: Procedure EXD (Sparsity level  $\varepsilon$ , Initial rewiring percentage  $\zeta_0$ , interest)
2: Initialize Fully Connected Neural Network model;
3: Sparsify_network( $\varepsilon$ );
4: Initialize training algorithm parameters;
5: for  $i$  in range(1,epochs) do
6:   Feed_Forward();
7:   Back_Propagation();
8:   Apply_EXD_rule(  $\zeta_0$ , interest,  $i$ );           // Applying the EXD equation
9:   Weight_evolution( $\zeta$ );
10: end for

```

more popularity, we aspire that this will change in the future.

In our experiments, we search to point out the similarities and differences in training speeds among dense neural networks, the base SET procedure and its variations we introduce. For this goal we use the implementations provided by the creators of SET, that contain a custom MLP implementation that doesn't use any particular machine learning library and an implementation that uses the Tensorflow and Keras frameworks that are well known to the machine learning community and many developers use daily.

4.7 The Tensorflow-Keras Implementation Incidents

In the original Tensorflow-Keras implementation, the weight evolution process is done, as the developers state, with an ugly hack. After training the model for one epoch the weight evolution process is executed which modifies the weights by applying a new weight mask on them. For this to work, the weights need to be saved in custom variables that we define before the model is created. Then, to avoid memory increase problems the model gets erased and created again with the updated weights. This whole process is computationally expensive and as we display in Chapter 5 it causes the sparse methods to take more time to train than their dense counterpart, something that does not happen with the custom implementation.

Upon the paper's release the version of Keras used did not offer a better alternative to do this. But today, using the Callbacks API in Keras we can modify the way the training procedure is executed without calling multiple model fits or using custom logic to rewrite the

training process again. Essentially, we can tell Keras to execute the weight evolution process before any new epoch begins and this time without any memory increases or "ugly hacks". With these changes, we expect to reduce the training costs noticeably.

Chapter 5

Evaluation & Results

To propose new techniques is one thing, but to find if they are of any value is a whole other effort, but a necessary one. We needed to find how we could evaluate our work to discover if they add any value to the already existing methods found in literature. This chapter is devoted to this cause.

5.1 Experimental Evaluation

In the experiments we conducted we evaluate our methods using the following metrics:

1. Memory Footprint.
2. Accuracy.
3. Time required for Training.
4. Time required for Inference.

We expect memory footprint to be around the same for all sparse methods as they use the same Erdős–Rényi random graph procedure. It is still important though how they all compare to their dense MLP counterpart. The time required for training is the total amount of real time that passes to train a model. The time required for inference calculates how fast we can make a single prediction, on a single experiment. We conducted three experiments for the Accuracy, Training and Inference Speed results and we present their mean values.

5.2 Evaluation Settings

In this section we refer the tools used to evaluate our work and analyze the procedures with which this was achieved. It is the foundations to our experiments and it helps provide a context in which the results that will arise can be understood and evaluated.

5.2.1 Hardware Utilized and Software Implementations

Our experiments were conducted on a desktop personal computer with an Intel Core i7 processor. The software implementations contain two families: one using custom Python code and the other using Tensorflow and Keras. The custom Python code uses Scipy's sparse matrix operations, which are useful for speed and memory footprint evaluation, while the second, due to the use of Keras, is considered as more trusted and better performing in accuracy which may make a greater appeal to other developers and may give them a greater incentive to use our findings. For more information and details about the two implementations, please refer to Appendix A.

For the sparse models (SET, LDV, EXD, OSV) we assign the sparsity level ε equal to 20 which configures the compression rate depending on the model architecture used. The value zeta stays at 30% for the SET procedure, as this was the proposed value from its creators, and we use this same value as well for the initialization of zeta on our methods. The parameters' initialization that are specific for each method are reported along with the methods they belong to at Chapter 4.

5.2.2 Experimentation Categories

Our contributions fall into a wide range of areas and as a result we cant evaluate them all together in a single experiment. We therefore conducted several experiments that we split into the following categories:

Comparing Performance on more Datasets

In the original paper, the usefulness and validity of the SET procedure was demonstrated with experiments on a small variety of datasets. We deemed notable to test SET along with our techniques on more datasets with wider characteristics to show how sparse implementations

perform in comparison to their dense counterparts. For these experiments we use a Multi-Layer Perceptron (MLP) with three layers, each one of them having 1000 neurons. We use this setup for each of the following datasets:

- LUNG
- ORL
- Prostate_GE
- GLIOMA
- Fashion MNIST

Most of the aforementioned datasets are widely used in machine learning and they come from a wide area where Artificial Neural Networks are employed. For more information about each dataset please refer to Appendix A.

Note. Even though the Fashion MNIST dataset was one of the datasets used in the original paper, we also used it here, in order to replicate experiments and to make comparisons on a common ground.

Each dataset is tested on the SET procedure, our three proposed techniques and their dense MLP counterpart using the custom Python code family. We train the models for 500 epochs using the default values for most hyper parameters. For analytical details about the model, hyperparameter values used and compression rates for each dataset, please refer to Appendix B.

Comparing Keras Implementations

In these category of experiments we compare our proposed techniques along with SET, using the default Keras implementation of the SET procedure used by Mocanu et al., to the new improved implementation we created. We include the dense MLP in these comparisons even though this implementation hasn't changed. As far as the model is concerned we use a neural network setup that is closely related to the experiments of Mocanu et al. We use a Multi Layer Perceptron of 3 hidden layers, with 4000 neurons on the first hidden layer, 1000 in the second and finally 4000 neurons in the third. We train the models for 500 epochs using the default values for most hyper parameters. For analytical details about the model and hyperparameter values used, please refer to Appendix B.

Comparing Speed Using Sparse Matrix Operations

For these experiments we use the Fashion MNIST dataset, and we compare the training and inference speeds between our improved Keras implementation and our custom Python implementation. We use the same network architecture we used on the previous experiment category. The reason for this experiments is to reveal the differences of the Keras API which does not use sparse matrix operations and the custom Python code which does use them. We expect the sparse methods of the custom Python code to perform much better comparatively to the dense MLP due to the utilization of sparse matrix operations.

Comparing Performance using 4 Hidden Layers

Leaving the Dense MLP behind in speed with our previous experiment using the custom Python code, with this experiment we decided to scale up the model complexity by experimenting on a much bigger network architecture. Using once again the Fashion MNIST dataset, we constructed a model with 4 hidden layers with 4000 neurons in each one, using the custom Python code. In these experiments, we leave the dense MLP out and we focus more on the sparse methods. More specifically we compare the SET procedure against our proposed methods to highlight their differences. For analytical details about the model and hyperparameter values used, please refer to Appendix B.

5.2.3 The Memory Footprint

From the designing phase of a neural network using one of the sparse techniques (SET and our proposed methods) we are able to know exactly how sparse the model will be after training. We know this before training because our methods belong to the sparse training category of model pruning, which means that even though the model's parameter's are pruned and reconnected throughout the whole training process, the total amount of parameters at the end of each epoch stays the same as the start. For all experiments we calculate the potential compression rate based on the number of weights kept. In actual memory size this greatly depends on how the program stores the weight matrices. If the weights are stored with dense matrices the compression rates we calculate are not translated to the actual compression rates in memory footprint. And if the weights are stored as sparse matrices the compression rates we calculate will need to be modified depending on the efficiency of the algorithm used.

5.2.4 Python Code vs Keras

From the above experiments we will be in the position to compare differences in performance between the custom Python implementation and our improved Keras implementation on models of the same architecture, on the same dataset and with the hyperparameters similarly tuned. We will be in the position to see how differently our sparse methods perform using sparse matrix operations in the custom Python Code and normal matrix operations with the Keras implementation.

5.3 Results

We will now present the results of our experiments, based on the category they belong.

5.3.1 Comparing Performance on more Datasets

LUNG Dataset

We should not forget that the LUNG dataset has only 203 samples of data and 3,312 features and 5 classes, coming from the field of biology.

Our experiments in this dataset show that all the sparse methods are at least 60% faster in training speed and at least 70% faster in inference speeds (Figure 5.2). Our proposed methods show a slight speedup on training speed and not any significant difference in inference speeds compared to the SET procedure. As far as accuracy is concerned, although the dense MLP stays consistently at the top, the LDV method shows a more smooth convergence than the SET procedure and outperforms it by small margin (Figure 5.1). For the memory footprint all sparse methods achieve a compression rate approximately to $\times 30.9$ (See App B). We can conclude that for this dataset the LDV method performed the best among the sparse methods, and if we can accept a small penalty in accuracy by choosing the LDV method we have a model that trains significantly quicker and leaves a vastly smaller memory footprint than the dense MLP.

ORL Dataset

We should not forget that the ORL dataset has 400 samples of gray-scale face images of 92x112 pixels in each one and 40 classes, belonging in the field of computer vision.

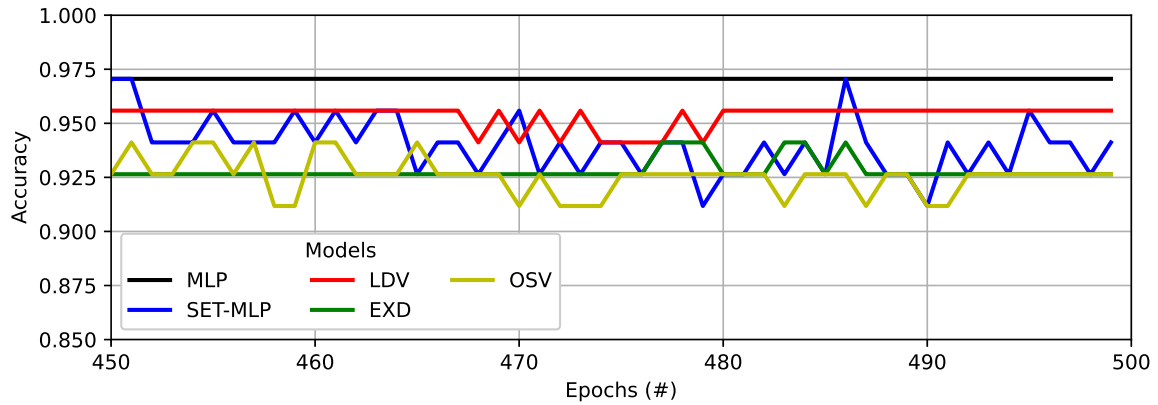


Figure 5.1: Accuracy of the Python code on the LUNG Dataset.

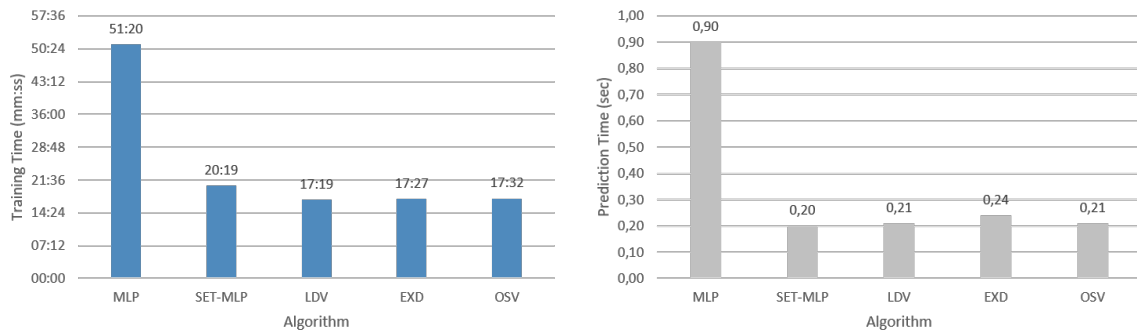


Figure 5.2: Training and Inference times on the LUNG Dataset.

Our experiments in this dataset show that all the sparse methods are at least 62% faster in training speed and at least 50% faster in inference speeds (Figure 5.4). Our proposed methods show a slight speedup on training speed and not any significant difference in inference speeds compared to the SET procedure. As far as accuracy is concerned, the dense MLP failed to train successfully even after many experiments, displaying a highly volatile accuracy (Figure 5.3). From the sparse methods, the LDV this times performs the poorest, but the EXD and OSV methods remain competent to the SET procedure and many times even exceed it. The EXD methods seems to perform slightly better than the SET procedure at all times indicating it as the winner in the accuracy tests. For the memory footprint all sparse methods achieve a compression rate approximately to $\times 22.2$ (See App B). We can conclude that for this dataset the EXD method performed the best among the tested methods in accuracy and even surpassed the the SET procedure and the dense MLP in training and inference speeds.

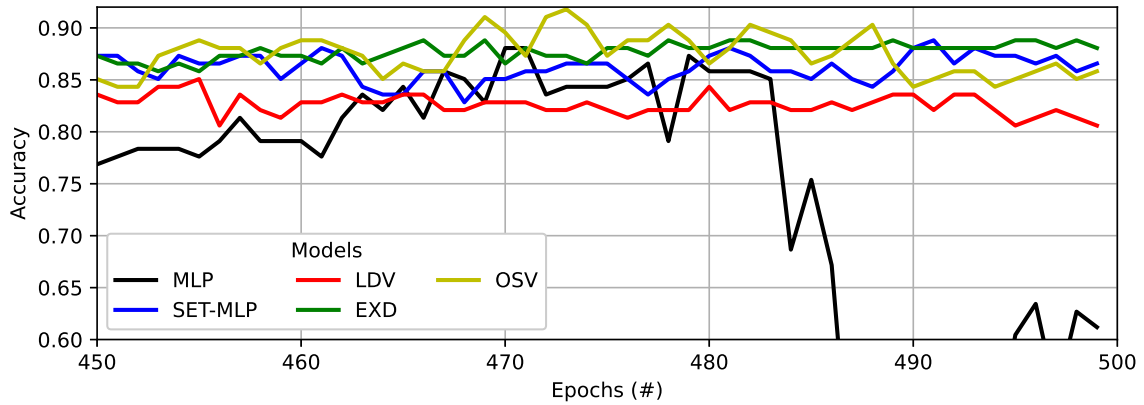


Figure 5.3: Accuracy of the Python code on the ORL Dataset.

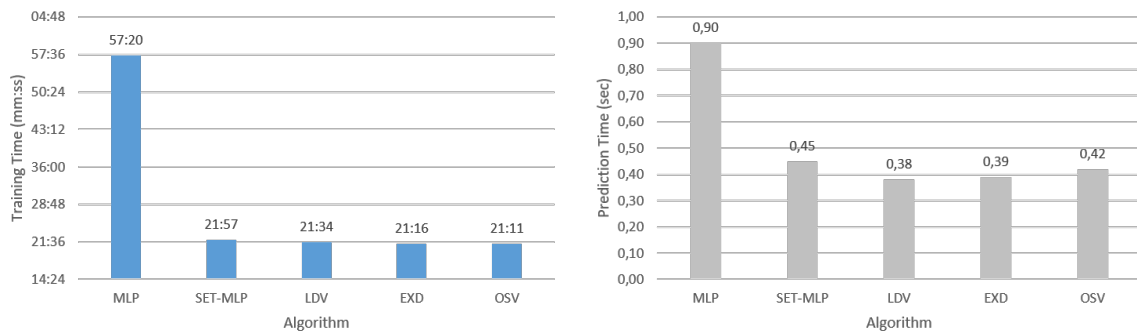


Figure 5.4: Training and Inference times on the ORL Dataset.

Prostate-GE Dataset

We should not forget that the Prostate-GE dataset has only 102 samples of data, 5,966 features and 2 classes, coming from the field of medicine.

Our experiments in this dataset show that all the sparse methods are at least 20% faster in training speed and at least 80% faster in inference speeds (Figure 5.2). Our proposed methods show a slight speedup on training speed and not any significant difference in inference speeds compared to the SET procedure. As far as accuracy is concerned, although the dense MLP stays consistently at the top, the EXD method shows a more smooth convergence than the SET procedure and outperforms it by small margin in small cases (Figure 5.5). For the memory footprint all sparse methods achieve a compression rate approximately to $\times 35.9$ (See App B). We can conclude that for this dataset the EXD method performed the best among the sparse methods, and if we can accept a small penalty in accuracy by choosing the EXD method we can have a model that trains quicker and leaves a vastly smaller memory footprint than the dense MLP.

The huge amount of features in this dataset, may be responsible for the smaller difference in training speeds, compared to the previous datasets. This huge amount of features however, may explain why there is a huge difference in inference times, as the sparse methods use significantly less parameters.

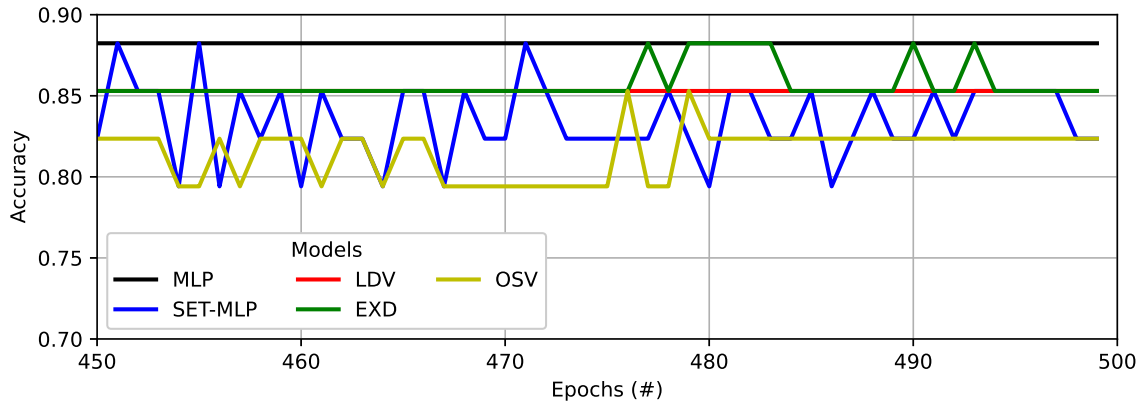


Figure 5.5: Accuracy of the Python code on the Prostate-GE Dataset.

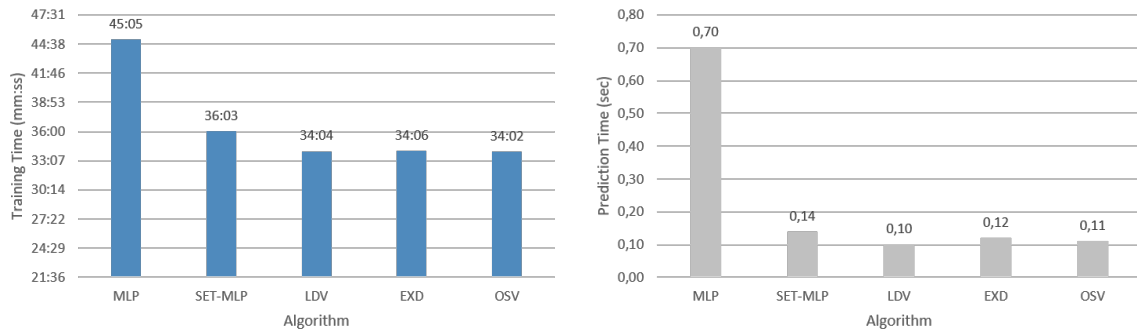


Figure 5.6: Training and Inference times on the Prostate-GE Dataset.

GLIOMA Dataset

We should not forget that the GLIOMA dataset has only 50 samples of data, 4,433 features and 4 classes, coming from the field of biology.

Our experiments in this dataset show that the SET procedure is 20% faster in training speed than the dense MLP (Figure 5.8). Our proposed methods display even higher training speed with the EXD method achieving 25% faster training speed than the dense MLP. In this dataset the inference speed differences are negligible. As far as accuracy is concerned, the dense MLP stays consistently at 0.7, the EXD method displays a same behavior to the SET

procedure and outperforms it by a small margin in some cases (Figure 5.7). For the memory footprint all sparse methods achieve a compression rate approximately to $\times 33.3$ (See App B). We can conclude that for this dataset the EXD method performed the best among the sparse methods, both in accuracy and in training speed.

The huge amount of features in this dataset reminds us of the Prostate-GE dataset, but we have way less samples for train, which may be responsible for the differences we observe in the GLIOMA Dataset which has similar general features to the Prostate-GE Dataset.

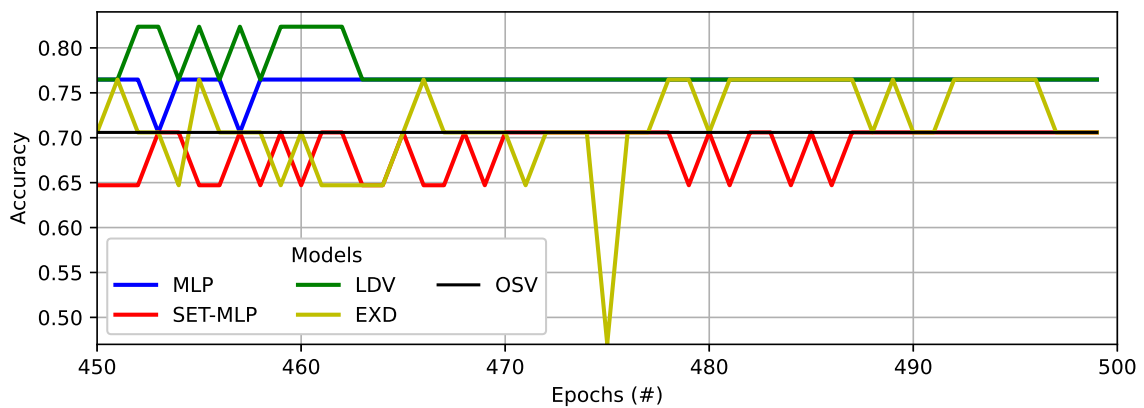


Figure 5.7: Accuracy of the Python code on the GLIOMA Dataset.

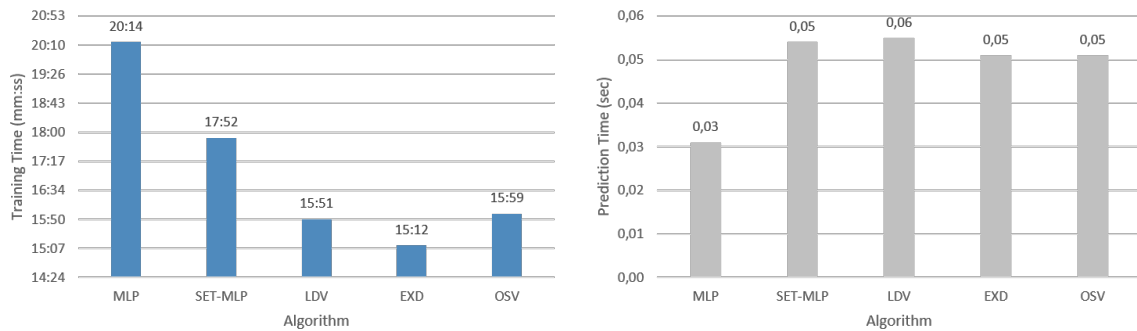


Figure 5.8: Training and Inference times on the GLIOMA Dataset.

Fashion MNIST Dataset

We should not forget that the Fashion MNIST dataset has 60,000 samples of gray-scale images of clothes with 28x28 pixels in each one and 10 distinct classes, belonging in the field of computer vision. In this category of experiments we do not apply data augmentation.

Our experiments in this dataset show that the sparse methods are approximately 54% faster in training speed than the dense MLP (Figure 5.8). Our proposed methods display

slight to no difference to the training time of the SET procedure, and the differences in the inference times are not significant enough to make any impact.

As far as accuracy is concerned, the dense MLP seems to slightly under-perform over the sparse methods, and at times we can see some sudden drops in accuracy (Figure 5.7). We can't pick a clear winner from the sparse methods as their performance is around the same. We could say that the LDV and OSV methods show a slight better performance but our choice is not supported by any statistical significance.

For the memory footprint all sparse methods achieve a compression rate approximately to $\times 21.5$ (See App B).

We can conclude that for this dataset no sparse method is the clear winner. In the following experiments we are more thorough on this dataset by using it on more network architectures using the Python code and with our improved Keras implementation which is much more optimized for accuracy. There is some potential to clear up the confusion on which method is the more suitable for this dataset.

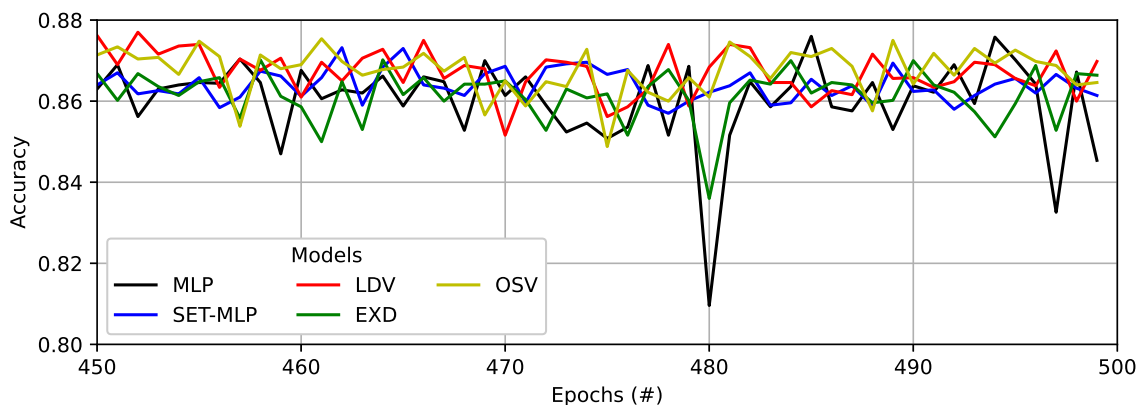


Figure 5.9: Accuracy of the Python code on the Fashion MNIST Dataset.

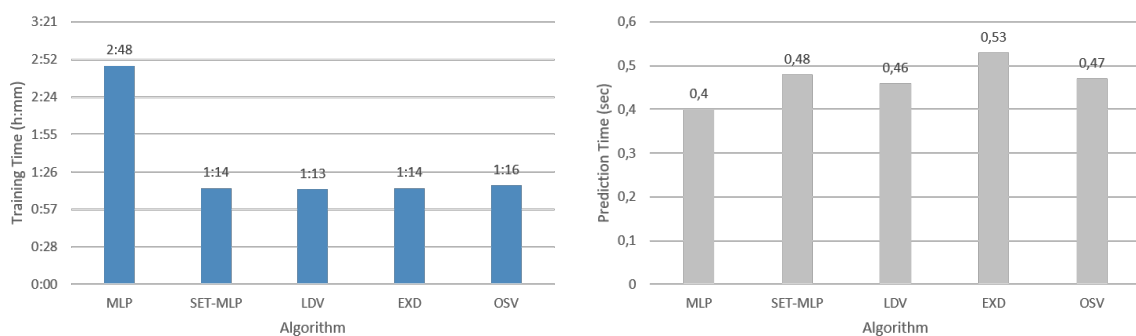


Figure 5.10: Training and Inference times on the Fashion MNIST Dataset.

5.3.2 Comparing Keras Implementations

In this experimentation category, we compare the original Keras implementation to our improved Keras Implementation using the Callbacks API. Firstly, the memory footprint remains unchanged between the two implementations and close to 350,000 parameters as, by design, the Erdős–Rényi sparse graph is solely responsible for the resulting memory footprint of the model. Comparing the sparse models however, to their dense MLP counterpart as seen in Figure 5.11, we see a great difference in the memory footprint. From 11,1 million parameters reduced to 350.000, we achieve a compression rate of $\times 32$, which is equivalent to around 96% reduction in parameters. And this is known to us before even starting to train the models. This is very beneficial, as we can tune the parameter ε from the start to achieve the desired compression rate, a feature that helps a lot when the model is designed to run on small or IoT devices with much less memory capabilities than a desktop computer.

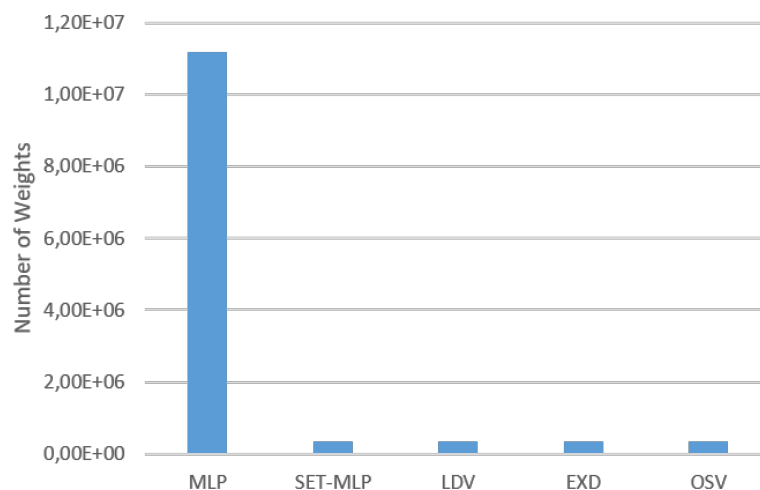


Figure 5.11: Comparing the number of weights used between the Dense MLP and the Sparse methods in the Fashion MNIST Dataset.

The second part of evaluating our methods in this experiment category is the time used for training and inference. As illustrated in Figures 5.12 and 5.13 the speed improvement comes in two ways. Firstly, when compared to the dense MLP we can see that the original implementation was significantly slower (around 16% for the training and around 40% for the inference) than the dense counterpart even though their memory footprint is much better. Using Keras Callbacks these differences are now greatly reduced as the weight evolution is now performed more smoothly. We would expect that the speedup would be even greater,

considering the compression rate achieved, but as Hoefler et al. [31] pointed out this is not always the case when using such libraries. The reason is that, most modern libraries are not designed to use sparse matrices and as a result they don't always exploit their reduced size efficiently.

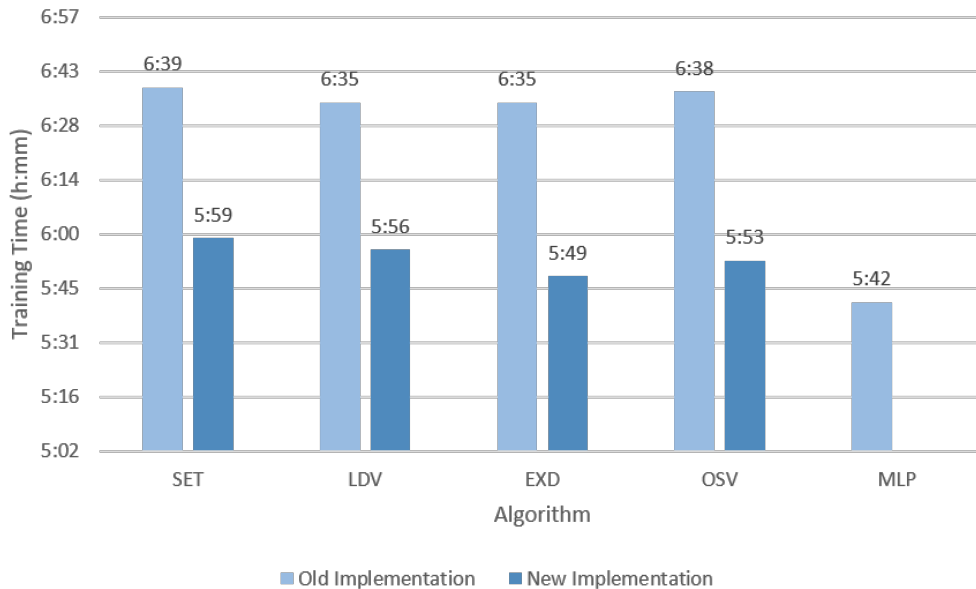


Figure 5.12: Comparative graph of Training Times using the old Keras implementation versus our Keras implementation in the Fashion MNIST Dataset.

Another speed improvement that is evident from the Figures 5.12 and 5.13 is how our methods compare to SET and the dense MLP network. Our methods are consistently better than the SET procedure with the Exponential Decay (EXD) method being the winner among the sparse methods when also taking the improved implementation into consideration. We explain this behavior due to the reduction of weight pruning and reconnecting that is a direct result of the parameter zeta reduction.

The final part of the evaluation in this experiment category involves the accuracy. As we can observe from the Figure 5.14 our methods remain competent to the SET and dense MLP models. We can even see that the EXD method remains slightly above the other methods as the models converge to around 90.5% accuracy. This declares the EXD method as a clear winner in accuracy as well. Our other two methods although being not as good as the EXD, we can say that they start to outperform the SET and dense MLP methods just slightly, as they pass the 400 epochs mark, although this difference is not significant enough.

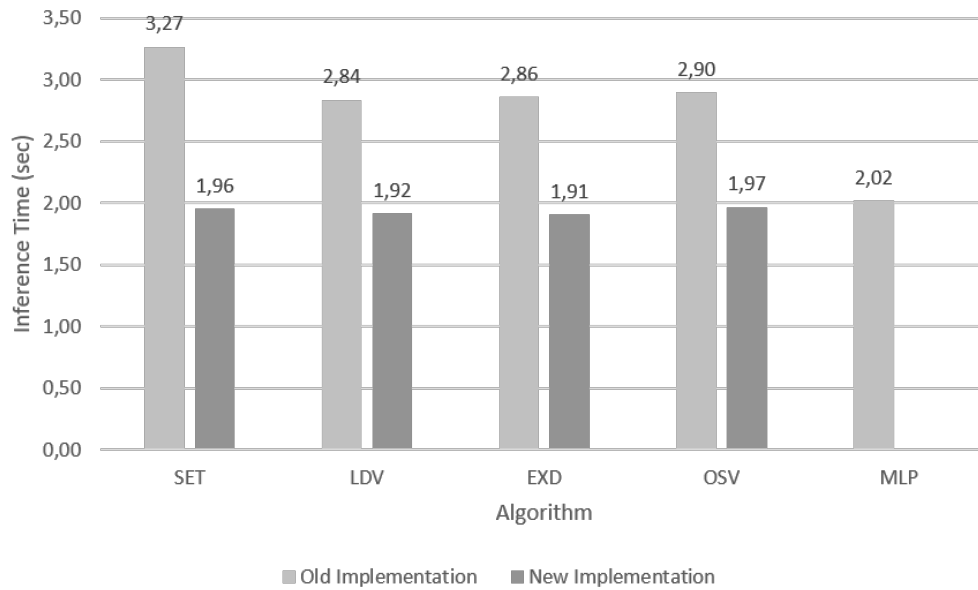


Figure 5.13: Comparative graph of Inference Times using the old Keras implementation versus our Keras implementation in the Fashion MNIST Dataset.

5.3.3 Comparing Speed Using Sparse Matrix Operations

As we stated in the previous experiment, the Keras and Tensorflow frameworks are not able to fully utilize the sparse models we tested and we wanted to find out the margin of this performance issue. For this we conducted the same experiments using our custom Python code that uses sparse matrix operations throughout the model's training and testing. This way the reduction in memory footprint can be translated to fewer matrix operations, leading to better performance. As seen by Figure 5.15 using sparse matrix operations we use around 50% less time for training and around 30% less time for inference predictions.

As we can clearly see, these discrepancies between the Keras implementation and the custom Python code, clearly indicate that the current framework cannot fully benefit in time for the reduced number of parameters the sparse models achieve. Even if the model is saved in sparse form, during training, the mathematical operations required to calculate the weight updates cause the weights to take the form of normal matrices, essentially eliminating any benefit that comes with the sparsification during the training process.

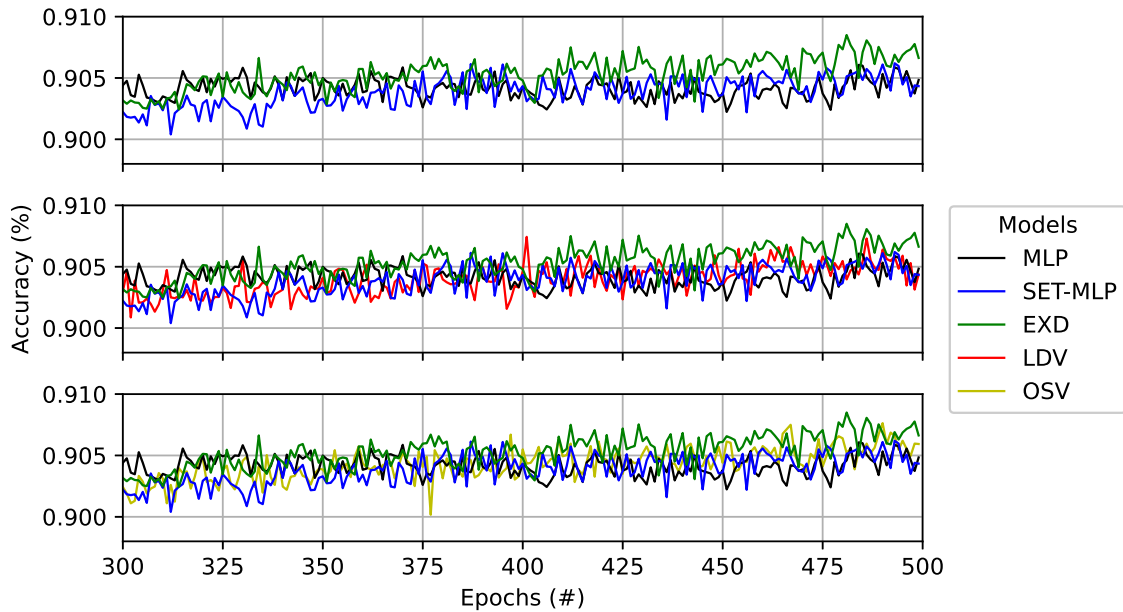


Figure 5.14: Accuracy Graph on the Fashion MNIST dataset comparing the proposed techniques using our improved Keras implementation.

5.3.4 Comparing Performance using 4 Hidden Layers

By this point it is clear that the sparse models that use sparse matrix operations can outperform the dense MLP significantly. So for this experiments, we tried our sparse methods to an even bigger network topology, in order to magnify the differences that comes with our proposed methods over the SET procedure. Using the 4 hidden layer topology we discussed in the previous section the results are as follows.

The weight reduction in this experiment is tremendous. From 51.2 million parameters we reduce them to only 650,000 achieving a compression rate of around $\times 78.7$. This translated to 98.7% fewer parameters in the model, making it possible to fit it in devices that its counterpart dense versions would be impossible to fit in using most of the current popular methods in model shrinking.

As far as time is concerned, our proposed methods are faster up to around 10% in training speeds, as seen in Figure 5.16 and have statistically insignificant changes in inference speeds. Even though this difference isn't a lot we should keep in mind that on most of our experiments one of our proposed methods manage to win SET in accuracy depending on the dataset used. In our case with Fashion MNIST the EXD method is the clear winner in these experiment category just as the previous one (Figure 5.17).

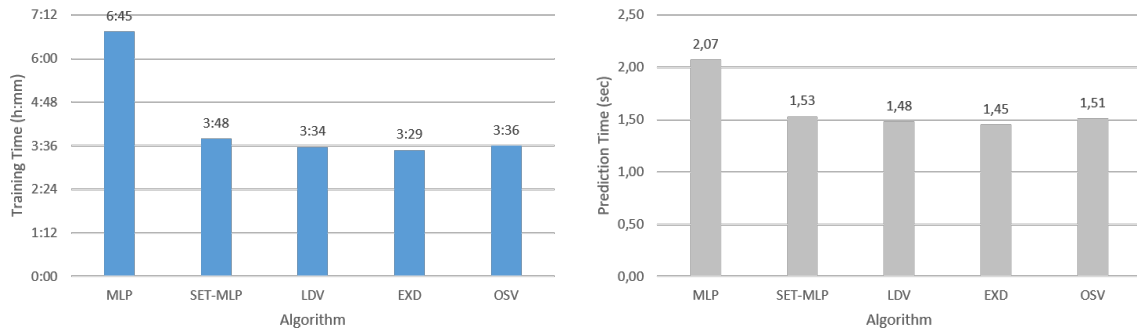


Figure 5.15: Training and Inference times on the Fashion MNIST Dataset using the custom Python code.

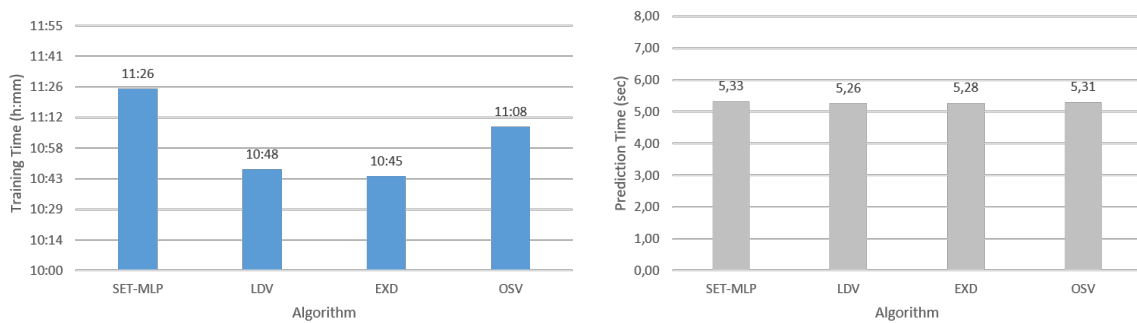


Figure 5.16: Training and Inference times on the Fashion MNIST Dataset using the custom Python code with 4 Hidden Layers.

In the experiment for this category however, a very interesting thing is observed. As seen by the Figure 5.18 in which we display the accuracy for first 100 epochs of the training, we can see a very interesting peculiarity. All methods are stuck to 0.1 for the first epochs, but with some variations depending on the method used. The SET procedure and our LDV method are stuck to 0.1 for around 30 epochs, while the EXD and OSV methods only need around 10 epochs to escape from the 0.1 accuracy. This causes the two aforementioned methods to receive a great head start in training, achieving an accuracy of around 0.8 by the time the other two escape the 0.1 mark. This is a head start of around 25-30 epochs in training, and we reach the 70 epochs before their difference in performance shrink to the point of being unobservable.

At this point many questions arise. Why EXD and OSV escaped so early, Why LDV performed similarly to SET in these first 100 epochs etc. The similarity between the EXD and OSV methods in these first epochs is that they both rapidly reduce the weight evolution rate, even though the OSV method rises it again periodically later. In contrast the LDV method

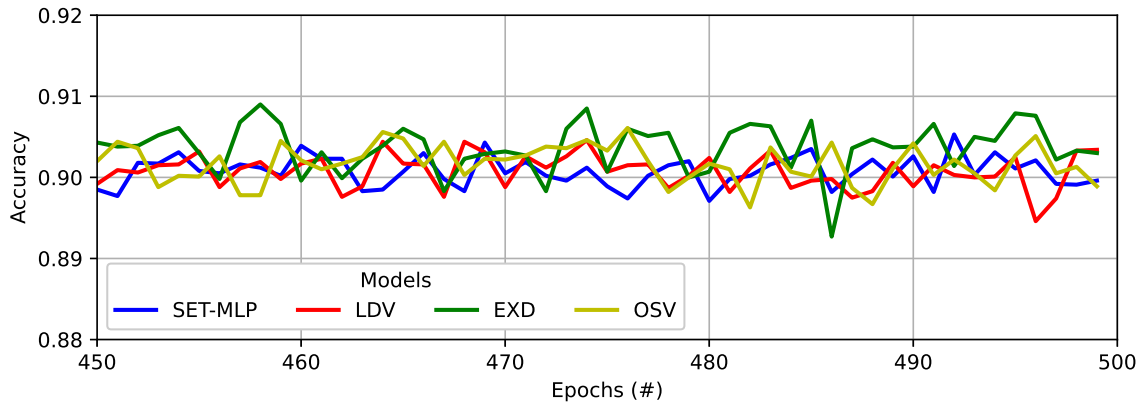


Figure 5.17: Accuracy Graph on the Fashion MNIST dataset comparing the proposed techniques, using the custom Python code with 4 Hidden Layers.

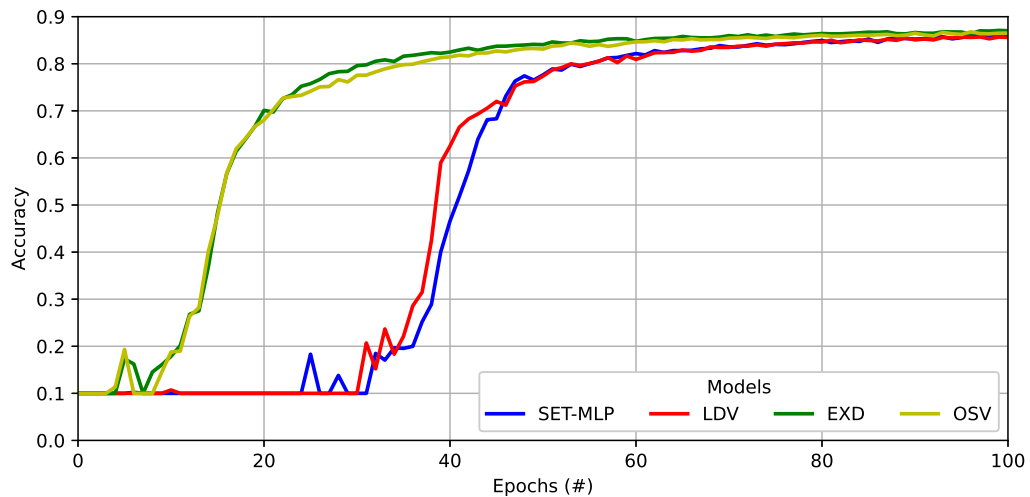


Figure 5.18: Accuracy Graph of the first 100 epochs on the Fashion MNIST dataset comparing the proposed techniques, using the custom Python code with 4 Hidden Layers.

reduces the weight evolution rate linearly, so in these first epochs its rate is closer to the SET procedure than the EXD and OSV methods. This is the reason it behaves in a similar manner as the SET procedure. This is the most logical way to explain these results.

The question is though, why this rapid reduction in the weight evolution rate causes the two methods to unstuck faster. In neural networks with more layers, the changes in the weights are bigger in the final layers but smaller and smaller as we go towards the earliest layers. So, a logical explanation is that if less weights are pruned, the existing weights have more time to grow and to reach maturity making the changes in the gradients to reach the first layers faster. In the SET and LDV methods instead, the weights are more likely to be pruned, bringing

new uninitialised weights instead, and as a result delaying the changes in the gradients from reaching the first layers.

Chapter 6

Conclusions

It is time to take all things presented into consideration by revisiting our efforts and results and giving some ideas and suggestions about future work.

6.1 Summary and Conclusions

In our project we presented three new methods that were build upon the SET procedure with the aim to achieve better performance both in time and accuracy, while also having the same but efficient memory footprint. We tested those methods along with the SET procedure and their dense MLP counterpart on several datasets and we saw our proposed methods won on most of those tests. We addressed the training and inference speeds on all our experiments and we made decisions based on the costs in time as well. Furthermore, we redesigned the Keras implementation used by Mocanu et al. improving the performance in time of a sparse methods compared to the dense MLP. And last but not least, we proved that the current Tensorflow and Keras frameworks cannot fully utilize sparse network topologies by comparing the performance in time on a custom Python implementation that supports sparse matrix operations. This poses a great obstacle to the research about improving neural networks through sparsification techniques.

Finally, among the methods we presented it appears that the Exponential Decreasing Method (EXD) was the overall best in performance, by displaying not only the best accuracy among the sparse methods in most experiments but by being the faster in training and inference speeds as well. It is a method that we believe should be preferred over the SET procedure at most, if not all, cases.

6.2 Future work

We believe that the future of Neural Networks will be sparse, but the current software infrastructure is not there yet. One of the best changes to be done in the future would be to reconfigure all the major ANN libraries to support sparse matrix operations. Tensorflow with Keras should offer the option to work with sparse models, by using matrix operations which would make them more efficient.

As far as our proposed methods is concerned, we designed some simple but efficient options that could be considered instead of the SET procedure. More formulas could be used, that are more complex and even combine multiple functions to achieve an even better result. An even more complex procedure could be designed that decides the weight evolution rate, based on some predetermined metrics that the training procedure already have at its disposal. The approaches towards sparse training are vast, so we hope at least we give some inspiration for new ideas to arise.

Bibliography

- [1] Howard B. Demuth, Mark H. Beale, Orlando De Jess, and Martin T. Hagan. *Neural Network Design*. Martin Hagan, Stillwater, OK, USA, 2nd edition, 2014.
- [2] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, 3 edition, 2020.
- [3] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. Corwin Press, Inc, 2020. <https://d2l.ai>.
- [4] M.T. Hagan and M.B. Menhaj. Training feedforward networks with the marquardt algorithm. *IEEE Transactions on Neural Networks*, 5(6):989–993, 1994.
- [5] C. Charalambous. Conjugate gradient algorithm for efficient training of artificial neural networks. *IEE Proceedings G (Circuits, Devices and Systems)*, 139:301–310(9), June 1992.
- [6] Aryan Mokhtari and Alejandro Ribeiro. Global convergence of online limited memory bfgs, 2014.
- [7] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond, 2019.
- [8] Robert A. Jacobs. Increased rates of convergence through learning rate adaptation. *Neural Networks*, 1(4):295–307, 1988.
- [9] Tom Tollenaere. Supersab: Fast adaptive back propagation with good scaling properties. *Neural Networks*, 3(5):561–573, 1990.
- [10] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1), 2014.

- [11] Xu Sun, Xuancheng Ren, Shuming Ma, and Houfeng Wang. meprop: Sparsified back propagation for accelerated deep learning with reduced overfitting, 2019.
- [12] Xu Sun, Xuancheng Ren, Shuming Ma, Bingzhen Wei, Wei Li, Jingjing Xu, Houfeng Wang, and Yi Zhang. Training simplification and model simplification for deep learning: A minimal effort back propagation method. *IEEE Transactions on Knowledge and Data Engineering*, 32(2):374–387, Feb 2020.
- [13] Sourya Dey, Diandian Chen, Zongyang Li, Souvik Kundu, Kuan-Wen Huang, Keith M. Chugg, and Peter A. Beerel. A highly parallel fpga implementation of sparse neural network training. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–4, 2018.
- [14] Aleksandar Zlateski, Kisuk Lee, and H. Sebastian Seung. Scalable training of 3d convolutional networks on multi- and many-cores. *Journal of Parallel and Distributed Computing*, 106:195–204, 2017.
- [15] Norman P. Jouppi, Cliff Young, Nishant Patil, and David Patterson. A domain-specific architecture for deep neural networks. *Commun. ACM*, 61(9):50–59, August 2018.
- [16] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks, 2019.
- [17] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2016.
- [18] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.
- [19] Sharan Narang, Gregory F. Diamos, Shubho Sengupta, and Erich Elsen. Exploring sparsity in recurrent neural networks. *CoRR*, abs/1704.05119, 2017.
- [20] Decebal Mocanu, Elena Mocanu, Peter Stone, Phuong Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature Communications*, 9, 06 2018.

- [21] Kyle Anderson, SangHyun Lee, and Carol Menassa. Impact of social network type and structure on modeling normative energy use behavior interventions. *Journal of Computing in Civil Engineering*, 28(1):30–39, 2014.
- [22] Anna D. Broido and Aaron Clauset. Scale-free networks are rare, 2018. cite arxiv:1801.03400Comment: 14 pages, 9 figures, 2 tables, 5 appendices.
- [23] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of ‘small-world’ networks. *Nature*, 393(6684):440–442, 1998.
- [24] Steven H. Strogatz. Exploring complex networks. *Nature*, 410(6825):268–276, March 2001.
- [25] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.
- [26] D.C. Mocanu. On the synergy of network science and artificial intelligence. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI 2016)*. AAAI Press, 2016. 25th International Joint Conference on Artificial Intelligence (IJCAI-16), July 9-15, 2016, New York, NY, USA, IJCAI-16 ; Conference date: 09-07-2016 Through 15-07-2016.
- [27] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks, 2015.
- [28] Andreas S. Weigend, David E. Rumelhart, and Bernardo A. Huberman. Generalization by weight-elimination with application to forecasting. In *Proceedings of the 3rd International Conference on Neural Information Processing Systems, NIPS’90*, page 875–882, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.
- [29] Luisa de Vivo, Michele Bellesi, William Marshall, Eric A. Bushong, Mark H. Ellisman, Giulio Tononi, and Chiara Cirelli. Ultrastructural evidence for synaptic scaling across the wake/sleep cycle. *Science*, 355(6324):507–510, 2017.
- [30] Graham H. Diering, Raja S. Nirujogi, Richard H. Roth, Paul F. Worley, Akhilesh Pandey, and Richard L. Huganir. Homer1a drives homeostatic scaling-down of excitatory synapses during sleep. *Science*, 355(6324):511–515, 2017.

-
- [31] Torsten Hoefer, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks, 2021.
- [32] Qamar Abbas, Farooq Ahmad, and Muhammad Imran. Variable learning rate based modification in backpropagation algorithm (mbpa) of artificial neural network for data classification. *Science International*, pages 2369–2380, 01 2016.
- [33] Davide Nardone. Biological datasets for smba, May 2019.
- [34] F.S. Samaria and A.C. Harter. Parameterisation of a stochastic model for human face identification. In *Proceedings of 1994 IEEE Workshop on Applications of Computer Vision*, pages 138–142, 1994.
- [35] Shenglong Yu and Hong Zhao. Rough sets and laplacian score based cost-sensitive feature selection. *PLOS ONE*, 13(6):1–23, 06 2018.
- [36] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

APPENDICES

Appendix A

Utilized Software and Tools

In this appendix we explain any software we use for this thesis along with any frameworks, platforms, API etc. Moreover, we refer to the datasets we used. But first let's start with the two implementation families we used.

A.1 The Implementations Used

A.1.1 Custom Python Implementation

The first implementation family we use consists of a custom Python code without the use of any API or framework that creates and trains neural networks. We used code openly provided through GitHub by Mocanu et al. [20] This code contains the collective efforts of Thomas Hagebols, who performed a thorough analysis on the performance of SciPy sparse matrix operations, Richie Vink for providing a nice vanilla Python implementation of fully connected MLPs ([GitHub Link](#)), and Amarsagar Reddy Ramapuram Matavalam for providing a fast implementation for the "weightsEvolution" method. These implementations use sparse matrices and clever operations using them, leaving a significant smaller memory footprint as a result. The implementations for our techniques was build on top of their code.

A.1.2 Custom Python Implementation

The second, is using the Tensorflow platform with the Keras API for the dense MLPs along with the additions of Mocanu et al. for their SET procedure. Our techniques was built on top of their code. Furthermore, we also provide an improved implementation to the one

Mocanu et al. use that utilizes the Callbacks API in Keras, making the whole training process run faster and more smoothly. We use both to compare their performance differences. We would like to clarify that this family of implementations does not use sparse matrix operations and we expect significant difference in performance since the reduction in parameters does not translate to faster computations.

A.2 Tensorflow

Tensorflow is a machine learning platform designed by the Google Brain team, to make the whole process of acquiring data, creating models, training them, making prediction and so on a little easier. It is an open source library for numerical computations and large-scale machine learning. It provides a convenient front-end API for building applications while executing them using optimized C++. It is one of the most well known and widely used platforms for using neural networks, with competitive performance when compared to other platforms.

It helps built state-of-the-art models without sacrificing speed or performance, which helps developers create models fast and easy, making research more efficient. We use Tensorflow for some of the implementations used in our experiments in conjunction with the Keras API. This way we show how our work could be used in a more intuitive and understandable way and it is a form of encouragement for other developers to apply our findings to other projects.

A.3 Keras

As stated in their official website "Keras is an API designed for humans, not machines." It is on another word, a simple and consistent API that makes the whole process of creating and using a neural network a simple process. The reason is that it minimizes the number of action required by the user for common use cases and it uses methods that automates the creation and training of complex models. A three layer neural network no longer requires 200 lines of coding by the user, but just 5. And now with Keras merge with Tensorflow their collaboration could never been smoother, faster and more direct.

It is used by many institutions, scientific organizations and companies from all around

the world, providing a rich tool set and lower-lever flexibility to house and help implement any research idea, with more than acceptable performance in most cases. Its wide usage and easiness to use is one of our main motivations that persuaded us to include it in our experiments.

A.4 Datasets

In this thesis we used the following datasets, each one of which will be explicitly referred to later in this Appendix:

- LUNG
- ORL
- Prostate-GE
- GLIOMA
- Fashion MNIST

A.4.1 The LUNG Dataset

The LUNG dataset contains in total 203 samples in 5 classes, adenocarcinomas, squamous cell lung carcinomas, pulmonary carcinoids, small-cell lung carcinomas and normal lung, with 139, 21, 20, 6 and 17 samples, respectively. The genes with standard deviations smaller than 50 expression units were removed getting a dataset with 203 samples and 3,312 genes. With its small number of samples and the relatively huge number of features it is considered an unstable dataset to classify. [33]

A.4.2 The ORL Dataset

As stated from the official Website: "The ORL (Our Database of Faces) contains 400 images from 40 distinct subjects. For some subjects, the images were taken at different times, varying the lighting, facial expressions (open / closed eyes, smiling / not smiling) and facial details (glasses / no glasses). All the images were taken against a dark homogeneous background with the subjects in an upright, frontal position (with tolerance for some side

movement) Figure A.1. The size of each image is 92x112 pixels, with 256 grey levels per pixel.” [34].



Figure A.1: ORL Dataset Sample

A.4.3 The Prostate-GE Dataset

Prostate-GE dataset has 102 samples and 5966 features from medical applications to be classified into two distinct classes, found in the UCI Machine Learning Repository. Much like LUNG it has even less samples and more features, making it another highly volatile dataset to classify. [35]

A.4.4 The GLIOMA Dataset

The GLIOMA dataset contains in total 50 samples in 4 classes: cancer glioblastomas, non-cancer glioblastomas, cancer oligodendrogliomas and non-cancer oligodendrogliomas, which have 14, 14, 7, 15 samples, respectively. Each sample has 12,625 genes. After a pre-processing, the dataset has been shrunk to 50 samples and 4,433 genes. Again like LUNG and Prostate-GE it has even less samples and more features, making it another highly volatile dataset to classify.[33]

A.4.5 The Fashion MNIST Dataset

Fashion MNIST is a dataset that consists of a training set of 60,000 images and a test set of 10,000 images. Each image is 28x28 gray-scale image, associated with a label from 10 classes of cloth-ware (Figure A.2). It was designed as a harder alternative to the overused MNIST digit classification dataset, where most models achieve accuracy so high, they can no longer be compared efficiently. With images closer to the needs of today's Computer Vision needs, Fashion MNIST is one of the most popular image classification datasets out there currently. [36]

This is the dataset we mostly use for our experiments due to its sensible size to training time spent ratio.

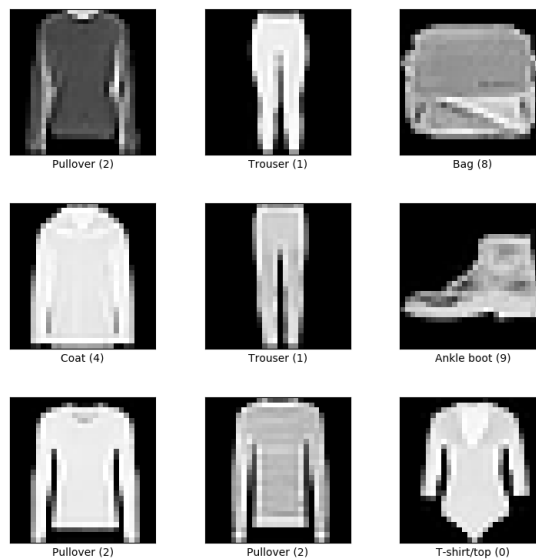


Figure A.2: Fashion MNIST Dataset Sample

Appendix B

Experimental Results and Figures

B.1 Comparing Performance on more Datasets

For this category of experiments we used the following architecture:

Layers	Output Shape	Activation	Rate	Parameters #
Input	x	-	-	0
Dense	1,000	ReLU	-	x * 1,000
Dropout	1,000	-	0.3	0
Dense	1,000	ReLU	-	1,001,000
Dropout	1,000	-	0.3	0
Dense	1,000	ReLU	-	1,001,000
Dropout	1,000	-	0.3	0
Dense	10	SoftMax	-	10,010

We use the Stochastic Gradient Descent with 0.01 learning rate with 0.9 momentum and we use Categorical Cross-entropy as a loss function. We train our models for 500 epochs with a batch size of 100 for the Fashion-MNIST dataset and set as 2 for the rest. In these experiments we do not apply any data augmentation. For the SET procedure we use a parameter $\zeta = 0.3$ while for our proposed methods we use the default parameters we presented in Chapter 4.

The following table sums up the memory footprint results for each dataset:

Dataset	Params (Dense)	Params (Sparse)	Compression Rate	% Decrease
LUNG	5,320,005	$\approx 172,000$	≈ 30.9	≈ 96
ORL	3,067,040	$\approx 138,000$	≈ 22.22	≈ 95
Prostate-GE	7,971,002	$\approx 222,000$	≈ 35.9	≈ 97
GLIOMA	6,441,004	$\approx 193,000$	≈ 33.3	≈ 97
Fashion MNIST	2,797,010	$\approx 130,000$	≈ 21.5	≈ 95

B.2 Comparing Keras Implementations

For this category of experiments we used the following MLP:

Layers	Output Shape	Activation	Rate	Parameters #
Input	784	-	-	0
Dense	4,000	ReLU	-	3,140,000
Dropout	4,000	-	0.3	0
Dense	1,000	ReLU	-	4,001,000
Dropout	1,000	-	0.3	0
Dense	4,000	ReLU	-	4,004,000
Dropout	4,000	-	0.3	0
Dense	10	SoftMax	-	40,010

And it uses a total of 11,185,010 trainable parameters. We use the Stochastic Gradient Descent with 0.01 learning rate and 0.9 momentum and we use Categorical Cross-entropy as a loss function. We train our models for 500 epochs with a batch size of 100. In these experiments we apply data augmentation to the Fashion MNIST Dataset. For the sparse methods we use $\varepsilon = 20$ which leads to models with around 350,000 parameters. For the SET procedure we use a parameter $\zeta = 0.3$ while for our proposed methods we use the default parameters we presented in Chapter 4.

Comparing Speed Using Sparse Matrix Operations

This category of experiments uses the same architecture as the previous experiment category.

Comparing Performance Using 4 Hidden Layers

For this category of experiments we used a MLP network using 4 hidden layers, named MLP 4k4L in our experiments. It follows the following architecture:

Layers	Output Shape	Activation	Rate	Parameters #
Input	784	-	-	0
Dense 1	4,000	ReLU	-	3,140,000
Dropout	4,000	-	0.3	0
Dense 2	4,000	ReLU	-	16,004,000
Dropout	4,000	-	0.3	0
Dense 3	4,000	ReLU	-	16,004,000
Dropout	4,000	-	0.3	0
Dense 4	4,000	ReLU	-	16,004,000
Dropout	4,000	-	0.3	0
Dense	10	SoftMax	-	40,010

And it uses a total of 51,192,010 trainable parameters. We use the Stochastic Gradient Descent with 0.01 learning rate and 0.9 momentum and we use Categorical Cross-entropy as a loss function. We train our models for 500 epochs with a batch size of 100. In these experiments we do not apply any data augmentation. For the sparse methods we use $\varepsilon = 20$ which leads to models with around 650,000 parameters. For the SET procedure we use a parameter $\zeta = 0.3$ while for our proposed methods we use the default parameters we presented in Chapter 4.