Universidade de Aveiro
2019
Departamento de Eletrónica,
Telecomunicações e Informática

Ricardo Jorge
Bastos Cordeiro de
Jesus

Observações em redes neuronais

Insights on neural networks

Universidade de Aveiro
Departamento de Eletrónica, Telecomunicações e Informática
2019

Ricardo Jorge Bastos Cordeiro de Jesus

Observações em redes neuronais

Insights on neural networks

"*Artificial intelligence is the new electricity*"

— Andrew Ng

Universidade de Aveiro
2019

Departamento de Eletrónica,
Telecomunicações e Informática

Ricardo Jorge Bastos Cordeiro de Jesus

Observações em redes neuronais

Insights on neural networks

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Rui L. Aguiar, Professor catedrático do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Sergey N. Dorogovtsev, Investigador coordenador do Departamento de Física da Universidade de Aveiro.

o júri / the jury

presidente / president

Prof. Doutor Joaquim João Estrela Ribeiro Silvestre Madeira
Professor Auxiliar, Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

vogais / examiners committee

Prof. Doutor João Manuel Portela de Gama
Professor Associado, Faculdade de Economia da Universidade do Porto

Prof. Doutor Rui Luís Andrade Aguiar
Professor Catedrático, Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro (orientador)

agradecimentos / acknowledgements

First and foremost, a huge thanks to my supervisors, Prof. Rui Aguiar (DETI), Dr. Sergey Dorogovtsev (DFIS), Dr. Mário Antunes (IT), Dr. Rui Costa (DFIS), and Prof. José Mendes (DFIS), for all their support and guidance. Without them this thesis would only be a shadow of what it is.

Second, to the research group I have been associated with the past few years, Aveiro Telecommunications and Networking Group (ATNoG), Instituto de Telecomunicações (IT), where I could discuss and develop many ideas. IT supported this work in the scope of Project UID/EEA/50008/2019.

Last but not least, I wish to thank all my family and friends. They have made writing this thesis so much more enjoyable. They are for sure a huge part of it.

Palavras-chave

redes neuronais artificiais, aprendizagem profunda, aprendizagem automática, inteligência artificial, efeitos da inicialização.

Resumo

É impossível ignorar os muitos avanços que aprendizagem automática, e em particular o seu método de eleição, aprendizagem profunda, têm proporcionado à nossa sociedade. No entanto, existe um sentimento crescente de que ao longo dos anos a área se tem vindo a tornar confusa e pouco clara, com alguns investigadores inclusive afirmando que aprendizagem automática se tornou na alquimia dos nossos tempos. Existe uma necessidade crescente de (voltar a) compreender em profundidade as ferramentas usadas, já que de outra forma o progresso acontece às escuras e, frequentemente, por tentativa e erro. Nesta dissertação conduzimos testes com redes neuronais artificiais dirigidas, com o objetivo de compreender os fenómenos subjacentes e encontrar as suas causas. Começamos por testar com um conjunto de dados sintético. Usando um problema amostra, descobrimos que a configuração dos pesos de redes treinadas evolui de forma a mostrar correlações que podem ser compreendidas atendendo à estrutura das amostras do próprio conjunto de dados. Esta observação poderá revelar-se útil em áreas como Inteligência Artificial Explicável, de forma a clarificar porque é que um dado modelo funciona de certa forma. Descobrimos também que a mera alteração da função de ativação de uma camada pode causar alterações organizacionais numa rede, a nível do papel que os nós nela desempenham. Este conhecimento poderá ser usado em áreas como Aprendizagem por Transferência, de forma a desenvolver critérios precisos sobre os limites/condições de aplicabilidade destas técnicas. Enquanto experimentávamos com este problema, descobrimos também que a configuração inicial dos pesos de uma rede pode condicionar totalmente a qualidade do mínimo para que ela converge, mais do que poderia ser esperado. Esta observação motiva os nossos restantes resultados. Continuamos testes com conjuntos de dados do mundo real, em particular com o MNIST e HASYv2. Desenvolvemos uma estratégia de inicialização, à qual chamamos de inicialização densa por fatias, que funciona combinado os méritos de uma inicialização esparsa com os de uma inicialização típica (densa). Descobrimos também que a configuração inicial dos pesos de uma rede persiste ao longo do seu treino, sugerindo que o processo de treino não causa atualizações bruscas dos pesos. Ao invés, é maioritariamente um processo de afinação. Visualizamos este efeito ao marcar as camadas de uma rede com letras do abecedário e observar que as marcas se mantêm por centenas de épocas de treino. Mais do que isso, a escala reduzida das atualizações dos pesos aparenta ser uma impressão digital (isto é, uma condição necessária) de treino com sucesso — enquanto o treino é bem sucedido, as marcas permanecem. Baseados neste conhecimento propusemos uma estratégia de inicialização inspirada em filtros. A estratégia mostrou bons resultados durante o treino das redes testadas, mas simultaneamente piorou a sua generalização. Perceber as razões por detrás deste fenómeno pode permitir desenvolver novas estratégias de inicialização que generalizem melhor que as atuais.

Keywords artificial neural networks, deep learning, machine learning, artificial intelligence, initialization effects.

Abstract

The many advances that machine learning, and especially its workhorse, deep learning, has provided to our society are undeniable. However, there is an increasing feeling that the field has become little understood, with researchers going as far as to make the analogy that it has developed into a form of alchemy. There is the need for a deeper understanding of the tools being used since, otherwise, one is only making progress in the dark, frequently relying on trial and error. In this thesis, we experiment with feedforward neural networks, trying to deconstruct the phenomenons we observe, and finding their root cause. We start by experimenting with a synthetic dataset. Using this toy problem, we find that the weights of trained networks show correlations that can be well-understood by the structure of the data samples themselves. This insight may be useful in areas such as Explainable Artificial Intelligence, to explain why a model behaves the way it does. We also find that the mere change of the activation function used in a layer may cause the nodes of the network to assume fundamentally different roles. This understanding may help to draw firm conclusions regarding the conditions in which Transfer Learning may be applied successfully. While testing with this problem, we also found that the initial configuration of weights of a network may, in some situations, ultimately determine the quality of the minimum (i.e., loss/accuracy) to which the networks converge, more so than what could be initially suspected. This observation motivated the remainder of our experiments. We continued our tests with the real-world datasets MNIST and HASYv2. We devised an initialization strategy, which we call the Dense sliced initialization, that works by combining the merits of a sparse initialization with those of a typical random initialization. Afterward, we found that the initial configuration of weights of a network "sticks" throughout training, suggesting that training does not imply substantial updates — instead, it is, to some extent, a fine-tuning process. We saw this by training networks marked with letters, and observing that those marks last throughout hundreds of epochs. Moreover, our results suggest that the small scale of the deviations caused by the training process is a fingerprint (i.e., a necessary condition) of training — as long as the training is successful, the marks remain visible. Based on these observations and our intuition for the reasons behind them, we developed what we call the Filter initialization strategy. It showed improvements in the training of the networks tested, but at the same time, it worsened their generalization. Understanding the root cause for these observations may prove to be valuable to devise new initialization methods that generalize better.

Contents

Contents i

List of Figures v

List of Tables ix

Acronyms xi

1 Introduction 1

1.1 Motivation 4

1.2 Problem scope/Objectives 5

1.3 Thesis organization 5

2 Background on artificial neural networks 7

2.1 Neural networks: The genesis 7

2.1.1 The McCulloch-Pitts neuron 8

2.1.2 Hebb's learning rule 9

2.1.3 The Perceptron 10

2.1.4 The first winter 11

2.2 Neural networks: The second age 12

2.2.1 Backpropagation 12

2.2.2 The second winter 13

2.2.3 The new era 14

2.3 Contemporary neural networks 16

2.3.1 The neurons 17

2.3.2 The activation functions 18

2.3.3 The multilayer perceptron . . . 20
2.3.4 Convolutional neural networks . . . 21
2.4 Optimization algorithms . . . 23
2.4.1 Gradient descent . . . 24
2.4.2 Momentum . . . 26
2.4.3 AdaGrad . . . 27
2.4.4 RMSProp . . . 29
2.4.5 Adam . . . 29
2.5 Initialization strategies . . . 30
2.5.1 Glorot's initialization . . . 31
2.5.2 He's initialization . . . 32
2.5.3 LeCun's initialization . . . 33
2.5.4 Sparse and lightning initializations . . . 33

3 Data, processing and tools 35
3.1 Synthetic dataset . . . 35
3.1.1 Generation rules . . . 36
3.1.2 Input/output correlations . . . 38
3.1.3 Instantiation for training . . . 40
3.2 Real-world datasets . . . 41
3.2.1 MNIST . . . 41
3.2.2 HASYv2 . . . 43
3.3 Data pipeline . . . 45
3.3.1 Architecture . . . 45
3.3.2 The frameworks used and the choice of Python . . . 51
3.3.3 TensorFlow vs PyTorch . . . 52
3.3.4 Computing nodes . . . 56

4 Probing the learning process 59
4.1 Base settings of the training experiments . . . 60
4.1.1 The network . . . 60
4.1.2 Dataset and training parameters . . . 61
4.2 Training results obtained with the reference parameters . . . 61
4.3 Varying the output activation function . . . 64

4.4 Inspecting a network's configuration of weights 67
4.4.1 Input/output strengths 68
4.4.2 Layer-wise correlations of weights 69
4.5 Trajectories followed during training 72
4.5.1 Varying the batch size 74
4.5.2 Varying the learning rate 75
4.5.3 Varying the initialization point 76
4.6 Similarity between learning trajectories 77
4.6.1 Distance between uncorrelated trajectories 78
4.6.2 Distance between correlated trajectories 79

5 Effect of the initial configuration of weights on artificial neural networks 81
5.1 Reference settings 82
5.2 Preliminary tests with initialization strategies 83
5.2.1 Lightning-based initialization 83
5.2.2 The Dense Sliced Initialization 86
5.3 Inspecting a network's initial configuration of weights 90
5.3.1 The lottery ticket hypothesis 91
5.3.2 Similarity between the initial and final configurations of weights of a network (first impressions) 92
5.4 Tuning of weights during training 95
5.4.1 Marking the initial configuration of weights 96
5.4.2 Untrainability and loss of initialization mark 100
5.5 The filter initialization 103

6 Discussion and Conclusions 111
6.1 Future work 112
6.2 Final considerations 113

References 115

List of Figures

1.1 The rise of interest in Deep learning. 3

2.1 Example of a McCulloch-Pitts neuron. 8
2.2 Logical functions implemented with McCulloch-Pitts neurons. 9
2.3 Perceptrons. 10
2.4 Top-1 accuracy on ImageNet and number of parameters of well-known deep networks vs. operations. 16
2.5 A simple neural network with three layers. 17
2.6 An artificial neuron with bias input. 18
2.7 Typical activation functions used in neural networks. 19
2.8 Network representation of Kolmogorov's theorem. 21
2.9 Weight sharing in convolutional layers. 22
2.10 Inside a convolutional network. 23
2.11 Iterates of gradient descent with and without momentum. 27

3.1 A perfect neural network implementation for the synthetic dataset. 37
3.2 Covariance between input and output bits of the synthetic dataset. 38
3.3 Sample images from MNIST train dataset. 42
3.4 Sample images from HASYv2 train dataset (reverse color). 44
3.5 Distribution of the HASYv2 samples among classes. 44
3.6 Data pipeline architecture. 46
3.7 Schema of the database used for the experiments with the synthetic dataset. . . 48
3.8 Schema of the database used for the experiments with the real-world datasets. . 48
3.9 Organization of workers 49

3.10 Deep learning frameworks ranking. 52

4.1 Architecture of the networks trained. 60
4.2 Test loss resulting from different test runs using the reference parameters presented in Sec. 4.1. 62
4.3 Distribution of the networks of Fig. 4.2 that have reached a certain test loss value ϵ by a given iteration. 63
4.4 Distribution of the networks of Fig. 4.2 that have reached a certain test loss value ϵ by a given iteration, after their output is binarized to 0 and 1. 63
4.5 Training of different networks employing sigmoid activation in the output layer. 65
4.6 Result of binarizing the outputs of the networks of Fig. 4.5. 66
4.7 Distribution of the networks of Fig. 4.5 with sigmoid output layer, after their output is binarized to 0 and 1. 66
4.8 Propagation of activity for networks with linear and sigmoid output activations. 69
4.9 Paths of length two through layers. 70
4.10 Correlation of nodes at a certain distance 71
4.11 Training of different networks sharing the same initialization. 73
4.12 Trajectories varying with batch size. 74
4.13 Trajectories varying with learning rate. 75
4.14 Trajectories varying with the initialization point. 77
4.15 $\langle w \rangle$ along training for varying batch size. 78
4.16 Distance between unrelated networks trajectories. 79
4.17 Distance between correlated trajectories. 79

5.1 Architecture of the networks trained. 83
5.2 Error obtained with the lightning-based initialization. 86
5.3 The dense sliced initialization. 87
5.4 Error obtained with the dense sliced initialization when input nodes are sampled with probability proportional to their correlation with the output. 89
5.5 Error obtained with the dense sliced initialization when input nodes are sampled uniformly. 90
5.6 Loss and accuracy of networks on MNIST. 92
5.7 Distribution of the product of weights of the initial and final configurations of a network. 93

5.8 Spread of initialization seeds to nearby weights. 94
5.9 Mosaic of letters before and after training. 97
5.10 Training of neural networks marked with letters. 98
5.11 Evolution of a layer of weights marked with the letter `A`. 98
5.12 Distribution of the accumulated gradient of a layer of weights marked with the letter `A`. 99
5.13 Final values of the weights of a network's layer as a function of the respective initial values. 100
5.14 Untrainability of small neural networks marked with letters. 101
5.15 Deviation of converging and diverging networks. 103
5.16 Examples of initialization filters created by the Filter initialization. 106
5.17 Results of training networks initialized with the Filter initialization. 107
5.18 Results of training networks initialized with the Filter initialization, using a separate dataset for initialization purposes. 108
5.19 Loss of networks initialized with the Filter initialization along training. 109

List of Tables

3.1 Physical specifications of the computing nodes in use. 56
3.2 Versions of the main software used in the computing nodes. 57

4.1 Bit error rate for a small number of errors. 67

Acronyms

ANN	Artificial neural network
API	Application programming interface
BER	Bit error rate
CNN	Convolutional neural network
CPU	Central processing unit
DBMS	Database management system
DL	Deep learning
EMA	Exponential moving average
EWMA	Exponentially weighted moving average
GD	Gradient descent
GPU	Graphics processing unit
IO	Input/Output
MLP	Multilayer perceptron
MSE	Mean squared error
NIST	National Institute of Standards and Technology
PCC	Pearson correlation coefficient
RAM	Random-access memory
ReLU	Rectified linear unit
SGD	Stochastic gradient descent
SSH	Secure Shell
VPN	Virtual private network
XAI	Explainable AI

1 Introduction

> *We can only see a short distance ahead, but we can see plenty there that needs to be done.*
>
> — Alan Turing, *Computing Machinery and Intelligence*

A couple of years ago, few people would have expected machines to accomplish what they have accomplished, or to succeed in tasks the way they have proved successful. This feeling is particularly true when considering pattern recognition and many other information processing tasks. Humans have long been considered the undisputed champions in these chores. However, advances in areas such as machine learning have allowed computers to find their place at the heart of a large number of these applications (LeCun, Bengio, & Hinton, 2015), such as image classification (He, Zhang, Ren, & Sun, 2016; Krizhevsky, Sutskever, & Hinton, 2012; Szegedy et al., 2015), video recommendation (Koren, 2009; Salakhutdinov, Mnih, & Hinton, 2007), natural language processing (Collobert et al., 2011; Jean, Cho, Memisevic, & Bengio, 2015; Sutskever, Vinyals, & Le, 2014), and many others. It seems we are experiencing a period where humans and machines augment each other and achieve more together than either could do alone. This cooperation had been suggested for long, for instance, by computer scientist Licklider (1960). He stated this symbiosis would manifest as a very tight coupling between the human and the electronic members of the partnership, and that computing machines would evolve to do the routinizable work that must be done to prepare the way for insights and decisions in technical and scientific thinking.

Perhaps the first well-known personality to realize and recognize the potential of the human-machine cooperation was the chess grandmaster Garry Kasparov. As he discusses in (Kasparov, 2010), after losing to IBM's Deep Blue, he had the idea to conceive a match where, instead of humans playing against machines, human and machine would play as partners. As Kasparov himself puts it, "the idea was to create the highest level of chess ever played, a synthesis of the best of man and machine". His

findings were quite insightful. A month earlier, he had defeated 4–0 his opponent (the chest grandmaster Veselin Topalov, until recently the world's number one ranked player) in rapid chess. However, in the match with the aid of the computer, they scored a 3–3 draw. Kasparov recognized his advantage in calculating tactics had been nullified by the machine. These conclusions were further supported by a second tournament where contestants could compete in any way they wanted — in teams with other players or computers. It revealed that teams of human plus machine dominated the strongest computers — even chess-specific supercomputers were no match for a strong human player using a relatively weak laptop. Moreover, and possibly more surprisingly, the winner turned out not to be a grandmaster with a state-of-the-art computer, but a pair of amateur chess players using three computers at the same time. Kasparov concluded that the combination of "weak" human plus machine and better process was superior to a strong computer alone and, more remarkably, superior to a strong human plus machine and inferior process.

The reason behind the success of this symbiosis may be understood by Moravec's Paradox (Moravec, 1988; Rasskin-Gutman, 2009). Things that are very easy for us humans, like walking without stumbling while avoiding objects in our way, seem to be considerably difficult for machines. Meanwhile, multiplying two one hundred-digit numbers is trivial for a computer, whereas most of us would struggle to carry such a calculation. It seems that we humans are more proficient in understanding and improving simple processes that we perform the worse than complex innate ones at which we excel (M. Minsky, 1988). While this may be contradictory and even paradoxical, at least it appears to be useful. It allows us to battle our own weaknesses effectively.

Nevertheless, perhaps we should take care where we lead computers to, since Licklider (1960), in his discussion about the human-machine symbiosis, highlights that it seems entirely possible for machines, in due course, to outdo the capabilities of the human brain. Curiously, Kurzweil (1990) predicted that between 2020 and 2070 machines will pass the Turing test, and few years later, Moravec (1998) predicted that the hardware required to match the general intellectual performance of the human brain would be available in cheap machines in the 2020s. It seems we will not have to wait for long until we find out whether these predictions are correct.

In the past few years, Deep learning (DL), a family of machine learning methods based on Artificial neural networks (ANNs), has proved fundamental in enabling humans to do more with the help of their computers. As a very recent example, in a joint effort between DeepMind[1] and the University of Oxford, Assael, Sommerschield, and Prag (2019) created Pythia, the first ancient text restoration model that uses deep neural

[1] https://deepmind.com/

networks to recover missing characters from damaged text input. It sets the state-of-the-art in ancient text restoration, achieving 30.1% character error rate compared to 57.3% of human epigraphists. Pythia's authors highlight its importance to assist, guide, and advance the ancient historian's task.

The development of new deep learning models that improve or enable some task has been a constant ever since deep learning started gaining popularity. One may say that the work of Krizhevsky et al. (2012) was one of the first ones to significantly raise the interest of the research community in deep learning (particularly in deep convolutional networks). The interest has been renewed and increased continuously throughout the years. Another landmark work (of the many that appeared in-between), AlphaGo Zero (Silver et al., 2017), was a remarkable deep learning model that learned tabula rasa (i.e., starting without any prior knowledge) how to beat Go world champions by playing against itself. The rise in interest of deep learning is evidenced by the sheer number of Google searches for this term, shown in Fig. 1.1 (the increase is even more evident when compared with other well-known machine learning methods).

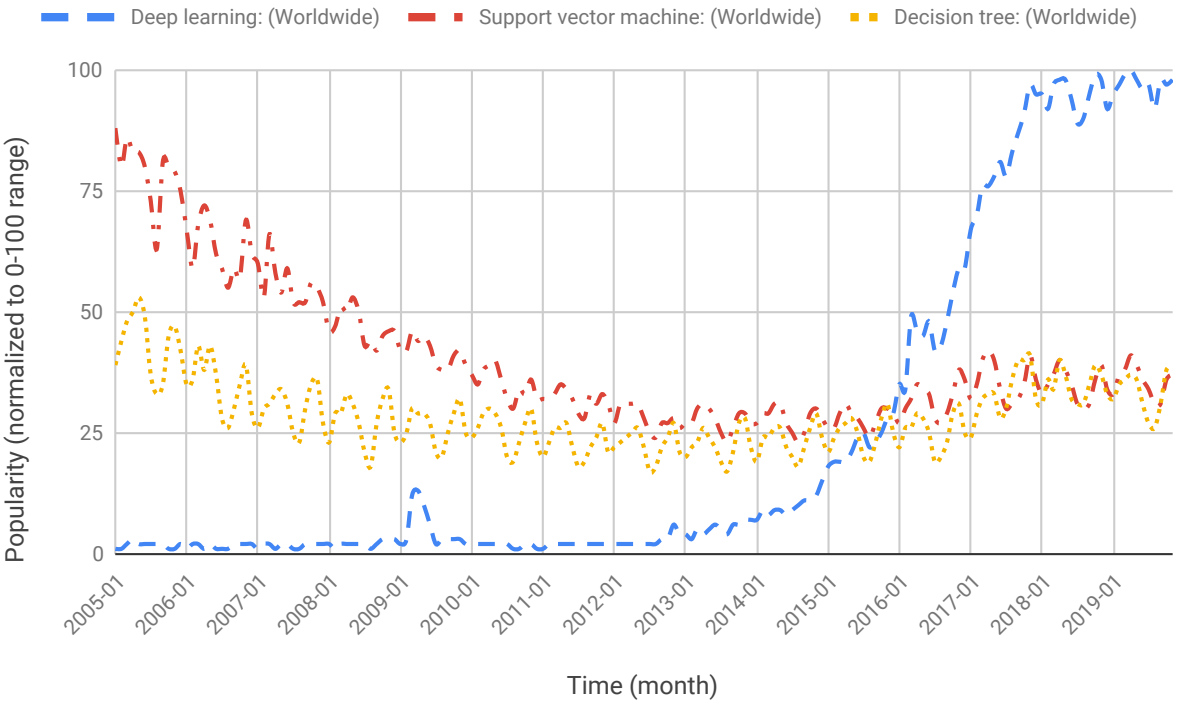


Figure 1.1: The rise of interest in Deep learning based on the number of Google searches for it. Source: Google Trends (https://www.google.com/trends).

Notwithstanding the many advances that deep learning has supported, one cannot ignore the problems that it has created along the way. It seems that, despite the many doors it opened, the improvements in the area have not, in general, been backed by rigorous, theoretical knowledge. The lack of our understanding of the methods we are continuously applying (and even evolving/improving) was recently put out bluntly by

Google's AI researcher Ali Rahimi in his presentation for the NIPS 2017 Test-of-Time Award[2] and on a follow-up paper (Sculley, Snoek, Wiltschko, & Rahimi, 2018). He states that machine learning (and, especially, deep learning) has become the alchemy of our age. It is this very same realization that motivates this thesis.

1.1 Motivation

It seems that machine learning has become a field that values new methods that beat previous ones on a given task or benchmark more than it values deep and precise understanding of those methods (Sculley et al., 2018). During his presentation at NIPS'17, Rahimi played an exercise where he asked two simple questions. First, he asked his audience how many projects they had been involved with the past year, in which they were trying out different techniques so that they could achieve state of the art results in some problem. Afterward, Rahimi asked how many experiments they run during the same time where they were chasing down an explanation — the root cause — for a strange phenomenon they observed. He highlights that there is much research of the former kind, but very few of the latter, and concludes by recalling that the goal of science is not to win, but to improve our knowledge.

Note that the issue Rahimi was referring to is different from AI's reproducibility problem (Hutson, 2018a), which relates to researchers not being capable of replicating each other's works due to inconsistent experimental and publication practices. As an example, Gundersen and Kjensmo (2018) found that only 6% of the 400 papers on AI they surveyed shared the code of the algorithms proposed in them. The issue also differs from the black box/interpretability problem, which relates to the difficulty of explaining how a particular AI model came to its conclusions (Voosen, 2017). The latter issue relates with Explainable AI (XAI), a recent and also important topic that seeks to increase the transparency of AI models, allowing, for instance, the implementation of the right to an explanation[3]. Rahimi's main issue is not a machine learning system that is a black box, but an entire field that has become a black box.

We do not believe the world is as dark as Rahimi painted it, nor do we believe it is as bright as Facebook's chief AI scientist, Yann LeCun, commented when replying to Rahimi's talk[4] — that AI is not alchemy, it is engineering, and engineering artifacts have almost always preceded the theoretical understanding. Benjamin Recht, co-author of Rahimi's alchemy keynote talk, seems to believe in a compromise with which we

[2] E.g., https://www.youtube.com/watch?v=Qi1Yry33TQE. Accessed 2019-10-20.

[3] These topics are ever more important in the European Union (EU), which is pressuring AI solutions affecting EU citizens to be lawful, ethical and robust. It recently published its Ethics guidelines for trustworthy AI (High-Level Expert Group on AI, 2019).

[4] https://www.facebook.com/yann.lecun/posts/10154938130592143. Accessed 2019-10-20.

agree, suggesting that both types of research are needed, one to determine where failure points come from, so that reliable systems may be built, and another to create ever more impressive systems (Hutson, 2018b).

In this work, we try to dismantle (part of) the machinery living inside neural networks, and understand some of the processes there happening. Our main objective is to try to develop insights about these processes, which may be used in future works to uncover more profound knowledge about artificial neural networks and deep learning. To achieve this, we follow the idea laid down by Rahimi in his talk — we take simple systems and try to understand them in-depth, in the hope that our findings may serve as stepping stones to understand more complex settings in the future.

1.2 Problem scope/Objectives

The first objective of this thesis is to study how artificial neural networks trained on a synthetic dataset (that can be precisely characterized, Sec. 3.1) evolve with training with respect to a broad set of factors, such as architecture, learning rate, and batch size. We take an in-depth view on the configurations of weights acquired by the trained networks, and discuss them at the light of the structure of the dataset itself.

The second objective, somewhat a corollary of the first, is to study the impact that the initial configuration of weights of a network has on its training and function. This study is performed resorting to "real life" datasets, and analysing the drift between the initial and final configurations of weights.

1.3 Thesis organization

This thesis is organized as follows. Chapter 2 introduces background topics on artificial neural networks along with some recent advances in the area. Chapter 3 presents aspects related with the experiments carried, namely, the datasets used, the programs developed to carry the experiments and process their results, and the way they interact. Chapters 4 and 5 constitute the body of work of the thesis. The former, Chap. 4, describes an exploratory stage where simple experiments were carried on a synthetic dataset, with the aim of gaining a general understanding of what is happening during the training of a neural network. The latter, Chap. 5, builds upon the knowledge acquired in the previous chapter and carries a more directed study on the effect of the initialization of a neural network in its training and function. Finally, Chap. 6 concludes the thesis, pointing out (hopefully) interesting venues for future research.

2 Background on artificial neural networks

You have to know the past to understand the present.

— Carl Sagan

Artificial neural networks (or neural networks for short) came to existence motivated by our interest to study the human brain (a field called Connectionism). They were first developed to model our biological neural networks, but it soon became clear that they could be used for a lot more. Throughout the years, there have been periods of significant interest and progress in the area, followed by years where progress dwindles. In this chapter, we start by taking an overview of the history of neural networks. We believe it is fundamental to know how something started so that we can build upon this knowledge and make steady progress. Then, we discuss topics involved with the design and training of contemporary neural networks, like common architectures, optimizers, and initialization strategies.

The chapter is organized as follows. In Secs. 2.1 and 2.2 we look into the history of neural networks, from its primordials in Sec. 2.1, to our current times in Sec. 2.2. Then, in Sec. 2.3, we overview the major components that are used for constructing the neural networks of our days, as well as to perhaps the most important architectures currently used, the multilayer perceptron and convolutional neural networks. In Sec. 2.4, we examine the optimization algorithms that are used to train them. Finally, in Sec. 2.5, we consider the initialization of neural networks and some of the most common strategies to do so.

2.1 Neural networks: The genesis

Nowadays, we use artificial neural networks as the chief tools of machine learning, applying them to lots of different problems and typically obtaining surprisingly good

results. Consider, for instance, the recent work of Breen, Foley, Boekholt, and Zwart (2019), where a deep fully-connected neural network model[1] obtains accurate solutions to Newton's problem of solving the equations of motion for three bodies under their own gravitational force, at fixed computational cost (being millions of times faster than the state-of-the-art solver). However, it happens that neural networks were initially conceived not as tools for machine learning, but as devices to help understanding how our human brain works. In this section, we overview the main contributions at the origin of neural networks, until the field's so-called "first winter" (which happened around 1970).

2.1.1 The McCulloch-Pitts neuron

In their seminal work on the logical analysis of nervous activity, Warren McCulloch, a neurophysiologist, and Walter Pitts, a logician, ended up formulating the first artificial neuron (McCulloch & Pitts, 1943). The so-called McCulloch-Pitts neuron (M-P neuron) was based on the following assumptions (among others) (McCulloch & Pitts, 1943, Part II).

- The activity of a neuron is an "all-or-none" process (i.e., it is binary, the neuron either fires or it does not, there is no state in between).
- A neuron fires if the sum of its excitatory inputs reaches a specific threshold and it receives no inhibitory input. Otherwise, it does not fire.
- The structure of a network of neurons is static, meaning that it does not change with time.

Figure 2.1 illustrates one such neuron.

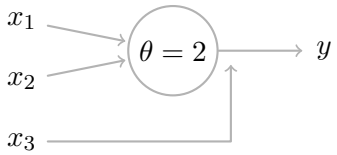
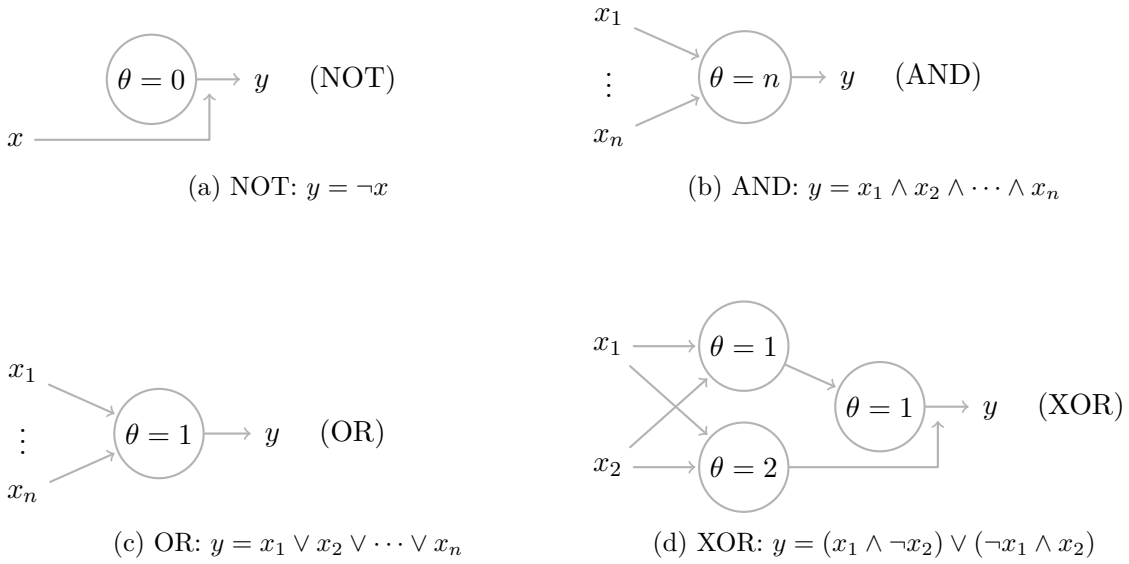


Figure 2.1: Example of a McCulloch-Pitts neuron.

The M-P neuron works as follows. The inputs x_i and output y take the logical values `0` and `1`. Two kinds of inputs exist: excitatory and inhibitory. In the figure, the inputs x_1 and x_2 are excitatory inputs, whereas the input x_3 is an inhibitory one. If any inhibitory input is active, the neuron outputs `0`. Otherwise, if the sum of the excitatory inputs reaches the value of θ, called the threshold, the neuron fires, outputting `1`. Else,

[1] More specifically, their network consists of 10 densely connected layers each with 128 nodes using the Rectified linear unit (ReLU) activation function, except for the output layer, which uses the linear activation function.

the neuron outputs `0`. As a result, the neuron in the figure can be seen as implementing the logical expression $x_1 \wedge x_2 \wedge \neg x_3$, since both x_1 and x_2 must be active for the neuron to receive an input which reaches its firing threshold, and, moreover, the inhibitory input x_3 must be inactive. If the threshold was lowered from two to one, the logical expression implemented would be $(x_1 \vee x_2) \wedge \neg x_3$. Figure 2.2 shows how some boolean functions may be implemented using M-P neurons. Notice that one is capable of implementing any logical function by combining these simple neurons, since the boolean operators `AND` and `NOT` (for example) can be implemented, as shown in the figure, and they constitute a functionally complete set of boolean operators.



(a) NOT: $y = \neg x$

(b) AND: $y = x_1 \wedge x_2 \wedge \cdots \wedge x_n$

(c) OR: $y = x_1 \vee x_2 \vee \cdots \vee x_n$

(d) XOR: $y = (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$

Figure 2.2: Logical functions implemented with McCulloch-Pitts neurons.

2.1.2 Hebb's learning rule

In the late 1940s, Donald Hebb observed that neural paths are strengthened each time that they are used (Hebb, 1949). He argued that the connection between two neurons is increased if they fire at the same time. This theory came to be known as Hebb's rule, and it originated a method of updating the weights of an artificial neural network, and inspired many others. The general idea is to update the weights of a network as each input/set of inputs is presented to it, according to whether the weights activate simultaneously or separately (the connection is increased in the former case and reduced in the latter). Hebb's observation also motivated the work of Rosenblatt, which would originate the perceptron, a significant development in the history of neural networks.

2.1.3 The Perceptron

In 1957, intrigued with the operation of the eye, Frank Rosenblatt, a psychologist at Cornell Aeronautical Laboratory, New York, began working on single-layer neural networks called the perceptrons[2] (Rosenblatt, 1957, 1958), the oldest neural network still in use today. He was motivated by the earlier works of McCulloch and Pitts and Hebb. However, he considered McCulloch and Pitts' symbolic logic and boolean algebra unsuited for the mathematical analysis of "systems where only the gross organization can be characterized, and the precise structure is unknown" (Rosenblatt, 1958). Moreover, he considered the M-P neuron to some extent "nonbiological", for instance, requiring excessive specificity of connections and synchronization, not accounting for the randomness of real biological neural networks. As a result, Rosenblatt, in some sense, simplified the M-P neurons, removing unnecessary restrictions. One of his major breakthroughs was providing the neurons' input links with weights, which denote the strength of the connection. In doing this, he also eliminated the "all or nothing" inhibitory inputs — they would be addressed by using negative weights. Note that, while the inputs and outputs of the perceptron are still the integers `0` and `1`, the network itself works with real numbers. The neurons of the perceptron compute the weighted sum of their inputs, and fire iff the sum reaches a specific threshold. Figure 2.3 illustrates the model.

(a) Fixed θ. (b) Learnable θ.

Figure 2.3: Perceptrons. In one (left) the threshold is fixed, whereas in the other (right) it is learnable.

Another important distinction to the McCulloch-Pitts neuron is that the thresholds of the neurons can be made learnable by providing each neuron with an extra input, which always receives the value of `1` (denoted the neuron's bias). This way, the weight of the connection functions as the neuron's threshold, while allowing it to be changed/adjusted as any other weight. One of the training procedures developed for Rosenblatt's perceptrons was the "error-correction procedure". It worked by adjusting the weights of neurons that incorrectly classified an input so that they would adequately recognize it (Nilsson, 2009, Sec. 4.2.1).

[2]The term perceptron comes from photoperceptron (a unit responding to optical patterns as stimuli). The term itself is frequently used to refer to either a single unit or a network of such units.

As will be seen later, this kind of unit is very similar to the ones used in building the neural networks of our days. The neurons of the perceptron are nothing more than a specific instance of our current model of a neuron, where one uses as activation function (a function applied to the weighted sum of the inputs of a neuron) the Heaviside step function (U),

$$U(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}.$$

2.1.4 The first winter

Rosenblatt's perceptron and its learning schedule draw significant interest in the research community (and even the general public[3]). Some viewed the perceptrons as having remarkable powers of self-organization, and as being the first concrete step into the development of true artificial intelligence (Newell, 1969). However, as time revealed, perhaps the Rosenblatt's perceptrons draw too much attention, causing too many people to have inflated expectations from them. When the limitations of the perceptrons started to surface — such as they being incapable of learning the simple boolean exclusive-or (XOR) problem — the community started to have doubts about their potential. The desolation burst open with the book of Marvin Minsky, co-founder of MIT's AI Lab, and Seymour Papert, co-director of the same laboratory (M. Minsky & Papert, 1969).

The book carried an extensive and thorough study of perceptrons[4] and revealed many of their limitations. The fundamental problem raised by it was related to the perceptrons' incapability of working with non-linearly separable problems (such as the XOR problem), combined with the difficulties of deploying (and training) multi-layer networks of perceptron neurons. The latter problem relates, first, with the computational cost of evaluating multi-layer networks (one should recall that at the time the computers were in their infancy), and second, with the lack of more evolved learning procedures (particularly, gradient-based techniques, which were being overlooked since the perceptrons were binary).

Their book is frequently regarded as being at the origin of the so-called "AI winter", a period that lasted till the early 1980s, where research in artificial intelligence saw a marked demise, and funding was extensively cut. Minsky and Papert themselves acknowledge this effect in the expanded version of their book (M. L. Minsky & Papert, 1988, p. xiii). However, they highlight that the shift of research that their book motivated

[3] https://www.nytimes.com/1958/07/13/archives/electronic-brain-teaches-itself.html

[4] In their book, Minsky and Papert consider more general perceptrons than those of Rosenblatt, though they use the same name in a tribute to the latter's pioneering work.

was, all in all, a necessity — in their own words, "a prerequisite for understanding more complex types of network machines". Moreover, they note that despite the severe limitations of the perceptron (and even because of them), it proved itself to be a very rich subject to study (M. L. Minsky & Papert, 1988, Sec. 13.2). All things considered, it seems that only after realizing the limitations their methods had, the research community could develop improved and even novel methods to overcome the shortcomings of the methods they had.

2.2 Neural networks: The second age

After the end of the first winter, the field of neural networks saw significant developments, like the famous backpropagation algorithm. However, the lack of maturity of the hardware and software of the time, as well as the limited availability of the data required to train neural network models, caused the field to see a second (smaller) winter by the mid-1990s. Nevertheless, as technology picked up, so did the developments in neural networks, and with them came the many breakthroughs that we have seen in the recent years. In this section, we describe briefly this period of the history of neural networks: the end of the first winter, backpropagation, and the reasons that lead to the second winter. Finally, we discuss some of the recent advances in the area.

2.2.1 Backpropagation

In the 1980s, the second wave of research in neural networks sprouted, initiated in 1982 by Hopfield networks (Hopfield, 1982), which are fully-connected recurrent binary networks. They were followed in 1985 by Boltzmann machines (Ackley, Hinton, & Sejnowski, 1985), which are binary stochastic recurrent networks with hidden units. The latter were found to be capable of discovering features representing complex regularities in the training data[5]. Perhaps of even more importance, 1986 saw the development of backpropagation (Rumelhart & Hintonf, 1986), motivated by the change from using the step function as the typical activation function of neurons to using sigmoidal activations (which only become computationally feasible due to the advances in computer hardware that had occurred).

Backpropagation is an algorithm used by optimizers (algorithms used for updating the weights of neural networks). It computes the gradient of the loss function of a network with respect to its weights, enabling the application of gradient-based methods to minimize the loss function of a network by adjusting its weights. It works by

[5]They were brought to spotlight somewhat recently in the context of the Netflix Prize (Koren, 2009; Salakhutdinov et al., 2007).

successively applying the chain rule of calculus backward through a network. Without backpropagation, evaluating the gradient of the loss function would become prohibitively expensive, even for reasonably small networks. Note, however, that backpropagation itself only describes how to compute the gradient of the loss function of a network with respect to its weights — how the gradient is used to update the weights is a different matter, that the optimization algorithms are responsible for addressing (we present a few such algorithms in Sec. 2.4).

Applying the backpropagation algorithm to very large neural network models, like the ones used today, implied developing structures such as computational graphs and techniques like autodiff methods. These approaches, when combined, allow one to compute the derivatives of a function that results from the composition of elementary function as the function itself is evaluated, allowing the efficient implementation of backpropagation. These topics are further addressed in Sec. 3.3.3.1.

2.2.2 The second winter

Although not as evident as the first winter, a second winter affected neural network research between 1996–2006 (LeCun, 2019). The reasons behind it were different from those of the first winter. It was caused by the lack of technology and data to properly train the neural networks that were being developed. First, the hardware was still too slow to execute a large number of floating-point operations sufficiently fast, which caused even somewhat primitive networks (especially if compared to those of today) to take weeks to train. Second, data were not as widely available as it is today, which limited the applicability of neural networks mainly to simple character and speech recognition problems (to which few, but some, datasets existed). Finally, there were very few software libraries developed (and even fewer open-source) to experiment with neural networks, which hindered their widespread.

Only when the hardware caught up, large datasets were built, and open source-minded frameworks were developed, did the research in neural networks (and particularly deep learning) boomed, causing their widespread and successful application in many different scenarios[6].

[6]Ironically, it seems unfortunate that Moore's Law — which was starting to make computers powerful enough to enable the usage of deep learning models in interesting real-world problems — is coming to an end (Waldrop, 2016). It now seems that CPU performance is expected to double only every 20 years (Hennessy & Patterson, 2017), instead of every two years. Sadly, many of the solutions that have been developed over the years to circumvent the problem for general-purpose programs (e.g., branch prediction, speculative execution, or hierarchical caching) are not particularly useful for deep learning computations. Due to this, application-specific devices for deep learning, like Google's Edge TPU (Tensor Processing Unit, https://cloud.google.com/edge-tpu/), are emerging in the market, as well as new custom-designed floating-point representations like bfloat16 (Wang & Kanwar, 2019), better adapted towards the needs of deep learning models. It seems that hardware is once again posing

2.2.3 The new era

Today, after the many highs and lows that the field has seen, deep learning is a key enabler of the many technological improvements we are experiencing. Deep learning is being successfully employed in problems such as self-driving cars, visual perception, medical image analysis, bioinformatics, physics, speech recognition, object recognition, drug discovery, genomics, among many others (LeCun, 2019; LeCun et al., 2015; Shrestha & Mahmood, 2019).

G. E. Hinton, Osindero, and Teh (2006) carried one of the earliest works that breathe new life into deep learning, after neural networks' second winter. It presented a novel and efficient strategy for training deep belief networks. The method was based on greedily pre-training each hidden-layer (one at a time) in an unsupervised manner, followed by a complete fine-tuning stage where all the layers would be trained simultaneously. It pioneered layer-wise pretraining. Shortly after its publication, it was scrutinized and generalized by Bengio, Lamblin, Popovici, and Larochelle (2007). The authors of the latter work found supporting evidence that the method helps upper layers (i.e., those closer to the output) find better representations of high-level abstractions of the input, which helps training and generalization significantly. The unsupervised nature of the method turned out to work remarkably well in a variety of tasks, especially when the amount of labeled data was limited (which at the time was a significant problem, and one that is still not completely solved nowadays).

Soon after, in 2009, Raina, Madhavan, and Ng (2009) accomplished the first large-scale deployment of a (deep belief) neural network. It was made possible by the opportunities opened by unsupervised learning methods (and their potential to use vast amounts of unlabeled data), along with the advent of fast and easy to program graphics processing units (GPUs), which the authors used in their work. The increase in the number of free parameters in their model to other contemporary models is striking. The largest models at the time had around 4 million free parameters (G. E. Hinton & Salakhutdinov, 2006), whereas their model had 100 million. Moreover, by using GPUs, they could reduce the training time of their model from several weeks to a single day.

Around the same time, interest grew in a particular type of deep feedforward network that had been achieving increasingly more practical success — the convolutional neural networks. They were introduced in (LeCun et al., 1989; LeCun, Boser, et al., 1990). We discuss the structure of convolutional networks briefly in Sec. 2.3.4. Meanwhile, in the remaining of this section, we report important practical advances achieved with them.

This type of network had been mostly neglected by mainstream computer vision

difficulties for deep learning, but this time there is great interest from the research community, and industry alike, to overcome them (Dean, 2019).

and machine learning communities until 2012, when AlexNet (Krizhevsky et al., 2012) won the ImageNet competition with a convolutional neural network, nearly halving the error rates achieved by the competing approaches. Its success revolutionized computer vision, establishing convolutional neural networks as the dominant approach to nearly all recognition and detection tasks (LeCun et al., 2015).

Nowadays, it is typical to deploy convolutional neural networks with 1 to 10 billion connections, 10 million to 1 billion parameters, and 8 to 20 layers (and beyond). Famous architectures include AlexNet (Krizhevsky et al., 2012) with 8 layers and winner of ILSVRC'12[7]; VGG-16 and -19 (Simonyan & Zisserman, 2014) with 16 and 19 layers, respectively (the latter was first runner-up in ILSVRC'14[8]); GoogLeNet, also known as Inception v1, (Szegedy et al., 2015), with 22 layers and winner of ILSVRC'14; and ResNet (He et al., 2016), with a whopping 152 layers and winner of ILSVRC'15. For perspective, a study by Karpathy (2014) found out that a human who trains on ILSVRC for approximately 20 hours achieves a top-5 error rate[9] of 5.1%. The RestNet network, winner of ILSVRC'15, achieves a top-5 error rate of 3.6%, achieving superhuman performance. Other recent networks showing promising results, suppressing those of ResNets, are DenseNets (Huang, Liu, Van Der Maaten, & Weinberger, 2017).

Figure 2.4 shows the top-1 accuracy of some of the well-known networks mentioned above on the ImageNet dataset, along with their sizes (number of parameters) and the operations (additions and multiplications) they require in each forward pass. For a comprehensive analysis of conventional deep learning models concerning metrics such as the accuracy they achieve, memory footprint, power consumption, among others, refer to (Canziani, Paszke, & Culurciello, 2016).

The success of the previously mentioned deep convolutional networks is so immense that it has caused the research community to seek new problems to solve. The focus has been shifting from image classification (i.e., having a classifier identifying whether an image contains a dog or a cat) to image segmentation (dividing/partitioning an image into various parts/segments) and object detection (identifying where dogs, or cats, or both are present in an image). Few examples follow. Mask R-CNN (He, Gkioxari, Dollár, & Girshick, 2017) pioneered the efforts in object detection with good results, but it has since been suppressed by Detectron2 (Wu, Kirillov, Massa, Lo, & Girshick, 2019). Meanwhile, DensePose (Güler, Neverova, & Kokkinos, 2018) reports achieving real-time

[7]ImageNet Large Scale Visual Recognition Challenge (ILSVRC, http://www.image-net.org/challenges/LSVRC/) is a well-known competition for evaluating algorithms in object detection and image classification.

[8]We skipped the winner of ILSVRC'13, ZFNet (Zeiler & Fergus, 2014), since it uses a network with 8 layers highly based on AlexNet.

[9]Top-5 error rate means computing the percentage of times that a target label is not present in the set of 5 top-scoring classes output by a classifier.

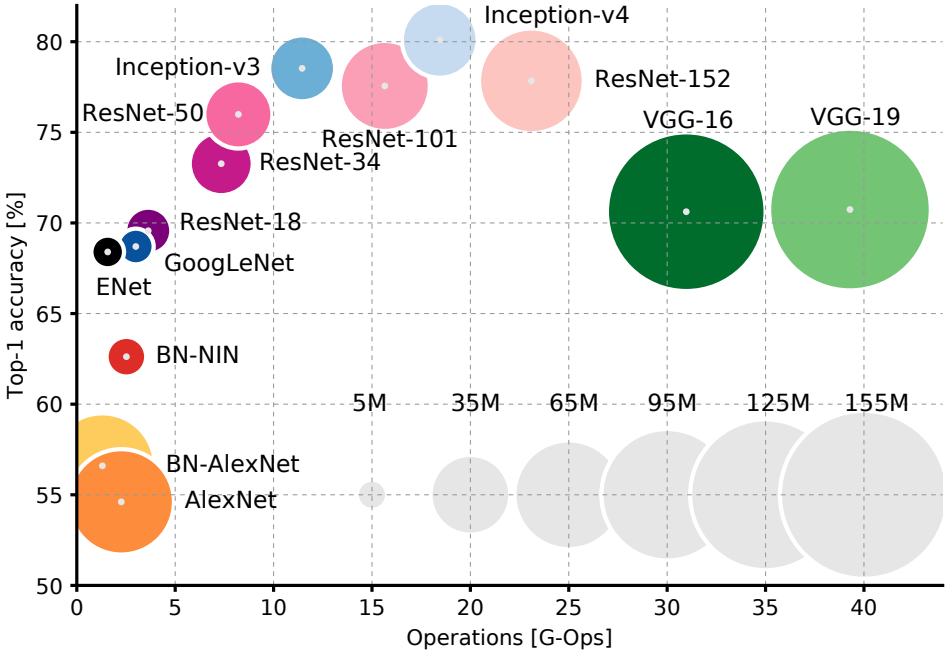


Figure 2.4: Top-1 accuracy on ImageNet versus amount of operations required for a single forward pass. The size of the blobs is proportional to the number of a network's parameters; a legend is reported in the bottom right corner, ranging from 5×10^6 to 155×10^6 parameters. Source: Canziani, Paszke, and Culurciello (2016).

body-pose estimation (i.e., mapping pixels of a human in an image to a surface-based representation of the human body). Finally, Deniz et al. (2018) use 3D convolutions to carry segmentation on magnetic resonance images.

Despite the undisputed success of deep learning models, there is still a large gap in our understanding of how or why they work. To some extent, it seems that deep learning is often seen as being more of an art than of a science. As a result, while it is essential to continue to push the limits of what we can currently do with deep networks further, it also becomes crucial to gain a deeper knowledge about their internals. Otherwise, our progress may eventually be blocked by the lack of our understanding.

2.3 Contemporary neural networks

As we have seen, a neural network is a biologically-inspired graph-based structure. It consists of a usually large number of neurons (or units or even nodes) joined together in a pattern of connections. These neurons are typically organized into three main groups: (i) the input units, which receive the information to be processed; (ii) the output units, which hold the results of the processing; and (iii) the hidden units, which are all the nodes in between the first two, receiving inputs from preceding units and providing outputs to succeeding ones. This organization is one of the main reasons why neural

networks are regarded as black-box models. The only interfaces of a neural network are its input and output nodes — all the logic in between is hidden. Note how this structure resembles that of our own nervous system, where our sensory neurons (input units) trigger a response in the motor neurons (output units) through all the other intermediate neurons (hidden units) in our brain (Buckner & Garson, 2019). Figure 2.5 illustrates an elementary neural network.

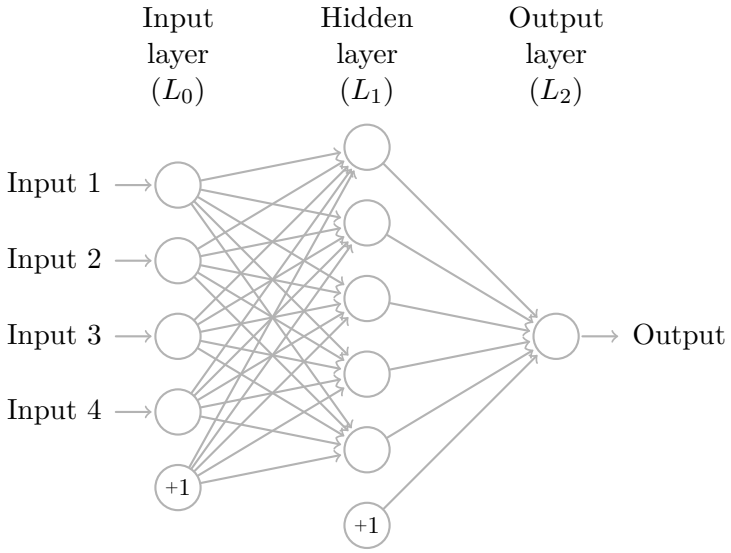


Figure 2.5: A simple neural network with three layers.

Neurons are very simple structures that mostly carry some computation on their inputs and propagate their results. Famous neurons in the history of artificial neural networks are the McCulloch-Pitts neuron and the perceptron, which we have seen in previous sections (Sec. 2.1.1 and Sec. 2.1.3, respectively). The neurons used nowadays are mostly adaptations/generalizations of the perceptron. In this section we start by discussing the elementary components of neural networks, particularly, neurons (Sec. 2.3.1) and activation functions (Sec. 2.3.2). Then, we present two important network architectures, the multilayer perceptron (in Sec. 2.3.3), and convolutional neural networks (in Sec. 2.3.4), alongside relevant results related to them (for instance, concerning their expressiveness).

2.3.1 The neurons

The typical neuron model used nowadays is very similar to Rosenblatt's perceptron. One such neuron is illustrated in Fig. 2.6. In general, a neuron receives a set of signals x_i as input and computes its output y by evaluating an activation function f on the

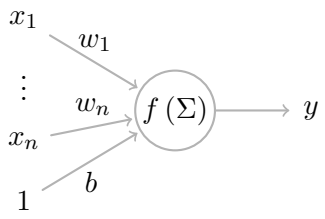


Figure 2.6: An artificial neuron with bias input.

weighted sum of its inputs, i.e.,

$$y = f\left(\sum_i x_i w_i + b\right).$$

The inputs of a node are either provided directly to the model (as actual input values) or are the output of previous nodes that are connected to it. Likewise, the output is either an actual output value or an input to another node (or even both). By successively connecting nodes, one constructs an artificial neural network. In short, a neural network essentially receives a set of inputs through its input units, and the signal is continuously propagated through its hidden units until it reaches the output units, which will output the overall activation values of the network.

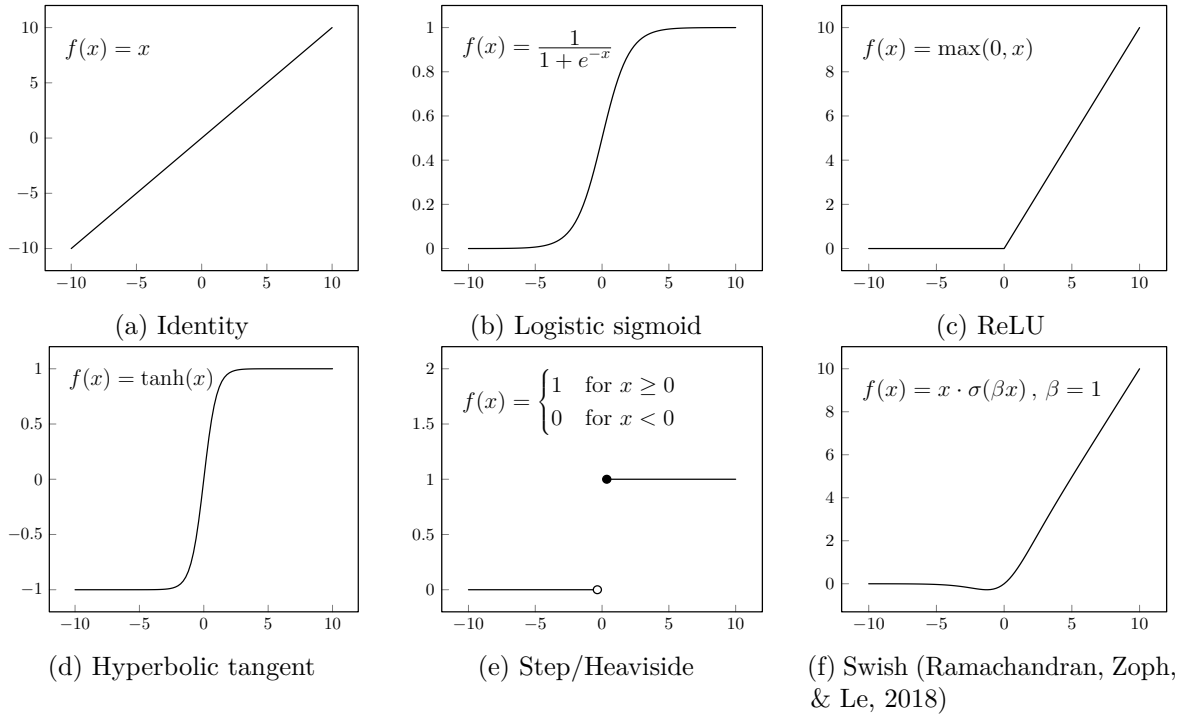
Connecting the units are weights, which represent the strength of the connection between them. They may either be positive or negative, denoting the "excitation" or the "inhibition", respectively, of the receiving unit to the activity of the sending unit. It is the weights that in some sense set the activation pattern of a network, and it is by adjusting them that a network can approximate some desired function (i.e., it is by adjusting the weights of a network that it "learns"/trains how to perform some task). A few such training algorithms are discussed in Sec. 2.4.

2.3.2 The activation functions

As we saw above, the output y of a neuron results from the evaluation of some (typically nonlinear) function f, called the activation function, on the weighted sum of the neurons' inputs, i.e., $y = f\left(\sum_i x_i w_i + b\right)$. The activation function f is usually a function that tries to mimic the "firing" of a brain neuron. Figure 2.7 portrays common activation functions[10].

Currently, the most successful and frequently used activation function seems to be the Rectified Linear Unit (ReLU) (Goodfellow, Bengio, & Courville, 2016, Chap. 6).

[10]The identity activation function ($f(x) = x$, Fig. 2.7a) is typically only used in the input and, in certain cases, the output layers of a neural network, as successive linear layers can be reduced to a single one (thus rendering them dull).



(a) Identity (b) Logistic sigmoid (c) ReLU

(d) Hyperbolic tangent (e) Step/Heaviside (f) Swish (Ramachandran, Zoph, & Le, 2018)

Figure 2.7: Typical activation functions used in neural networks.

The rationale behind its success lies with it being "almost linear", in that it is a piecewise function with two linear pieces, which allows it to retain many of the properties that make linear models easy to optimize with gradient-based methods (Goodfellow et al., 2016). Moreover, the fact that it truncates negative values originates sparse activations, where a significant amount of nodes output 0, which seems to agree with the observation that the neurons in our brains also activate sparingly (Glorot, Bordes, & Bengio, 2011).

Notwithstanding the widespread usage of ReLU, the study and development of new activation functions is still an interesting research topic. For instance, recently Ramachandran, Zoph, and Le (2018) have proposed a new activation function called Swish, defined as

$$f(x) = x \cdot \sigma(\beta x)\,,$$

where $\sigma(x)$ is the logistic function (Fig. 2.7b) and β is either a constant or a trainable parameter. Figure 2.7f plots Swish for $\beta = 1$.

Swish was found by automatic search. Its authors carried an experiment based on reinforcement learning where they evaluated combinations of different unary and binary functions, and the combination later called Swish came on top. Note the similarities between Swish and ReLU. As $\beta \to \infty$, Swish approaches ReLU, since $\sigma(\beta x)$ approaches 0 or 1, depending on the sign of x. By replacing ReLU with Swish, its authors obtained improvements of 0.5–1% on several models trained on ImageNet, without fine-tuning the remaining training hyperparameters, which suggests it may be an activation function

worth considering in the future.

2.3.3 The multilayer perceptron

Multilayer perceptrons (MLPs) are networks constructed by composing layers of neurons (of the type presented in Sec. 2.3.1) together. They are multilayer fully-connected feedforward networks, i.e., networks where the nodes of one layer have weights connecting them to all the other nodes of adjacent layers, and where the input signal propagates through the network from the input to the output layer, on a layer-by-layer basis. Figure 2.5 illustrates a MLP.

These networks have been applied successfully to a variety of problems in fields such as speech or image recognition, although nowadays other architectures (such as Convolutional neural networks (CNNs)) are more commonly used. Despite the decrease in their usage, they should still not be overlooked, since they possess interesting properties regarding their expressiveness. Furthermore, they are still typically used as the last layers of more complex architectures (such as convolutional-based networks).

These networks can be seen as a mathematical function mapping input to output values. The function itself is formed by composing many simpler functions. Interestingly, it was proven that any continuous function on a compact domain can be approximated with any given precision by an MLP with a single hidden layer and using any continuous sigmoidal activation function (Cybenko, 1989). Before his major result, however, there was another very important work by Andrey Kolmogorov.

In 1957, Andrey Kolmogorov proved a universal representation theorem for continuous functions, which states that any multivariate continuous function on a compact domain can be expressed as a finite sum of continuous functions of a single variable (Kolmogorov, 1957). More formally, Kolmogorov's theorem established that any continuous multivariate function $f\colon [0,1]^n \to \mathbb{R}$ can be represented in the form

$$f(x_1, \ldots, x_n) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^{n} \phi_{q,p}(x_p) \right)$$

where Φ_q are properly chosen continuous real functions and $\phi_{q,p}$ are continuous real function on $[0,1]$. The inner functions $\phi_{q,p}$ can be chosen independently of the function f. The idea behind Kolmogorov's theorem is illustrated in Fig. 2.8, where a generic transformation M maps $\mathbb{R}^n$ inputs into several unidimensional transformations, that are afterward summed together.

While this interpretation is not directly applicable to building neural networks (since, for instance, the functions Φ_q would have to be parameterized, which the theorem does

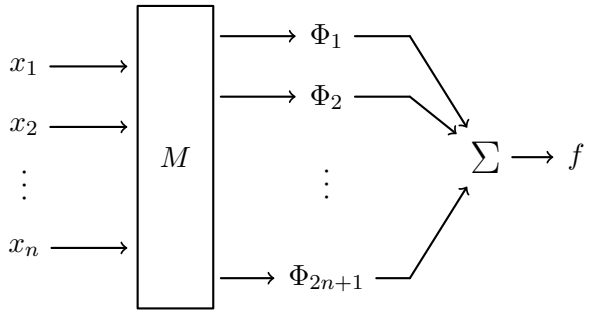


Figure 2.8: Network representation of Kolmogorov's theorem. Based on http://neuron.eng.wayne.edu/tarek/MITbook/chap2/2_3.html.

not capture), it alludes to the usage of parallel and layered structures for multivariate function approximation.

Meanwhile, the landmark proof of Cybenko (1989) (independently proven by Hornik, Stinchcombe, and White (1989) and Funahashi (1989)) is directly applicable to neural networks. It states that "any continuous function of n real variables on a compact domain can be approximated arbitrarily well by finite linear combinations of compositions of a fixed univariate sigmoidal function and a set of affine functionals". In other words, a MLP with a single hidden layer, using a sigmoidal activation function in that layer and linear activation function on the output layer is capable of approximating any given function to any degree of accuracy.

An important detail of this result is that it makes no considerations about the number of nodes required in the hidden layer for the desired accuracy, apart from it being finite. Of course, in practice, finite may not always be feasible. Nonetheless, the theorem still implies that if one of these networks fails to approximate a given function to the desired accuracy, it must be due to an inadequate choice of parameters or an insufficient number of hidden nodes, and not due to an inborn incapability of the network to the particular problem.

More recently, Hanin and Sellke (2017) has proved that MLPs employing the ReLU activation function in their hidden units, with an arbitrary number of layers, and each layer having $n + 1$ nodes, can approximate arbitrarily well any continuous, real-valued function of n variables on a compact domain.

2.3.4 Convolutional neural networks

Convolutional neural networks (CNNs) (LeCun, Boser, et al. (1990) and LeCun et al. (1989, Secs. 3.5–3.7)) are neural networks specifically designed for processing data that show some form of locality, i.e., whose contents are arranged in such a way that nearby inputs are meaningful together (form features). Examples of such types

of data include time series or images. They have proven to be hugely successful at such tasks, constantly occurring in state of the art architectures developed for datasets such as ImageNet and many other real-life problems. We have seen some of the many achievements of convolutional networks in Sec. 2.2.3.

The components that distinguish CNNs from fully-connected networks are mainly three, described in LeCun's seminal paper (LeCun et al., 1989). The first is to connect each unit in a hidden layer to only a small number of neighboring units in the preceding layer. The position of the hidden unit in its layer should reflect the position of the units it connects to in the layer before it, so that the locality of the units is retained. The groups of weights that connect a unit to the ones before it are typically called kernels. To some extent, restricting the number of connections between units pushes the hidden units into constructing local features by forcing them to combine (only) local sources of information.

The second aspect of CNNs is to consider that distinctive features of the data may appear anywhere, which is particularly true for images. For instance, consider that a particular object we want to identify may appear anywhere in a picture. Convolutional networks address this by weight sharing, i.e., by applying the same kernel to different positions of the input, creating "planes" that result from the evaluation of a kernel at the different positions of the input. Figure 2.9 illustrates the structure obtained by connecting layers with weight sharing.

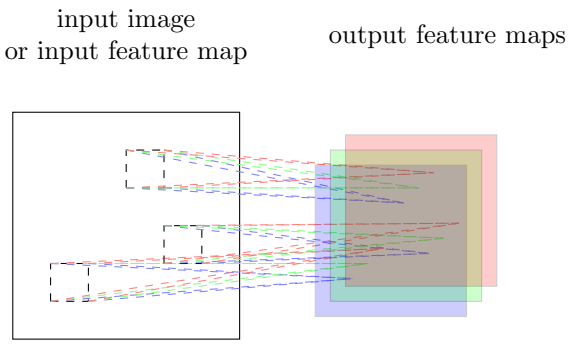


Figure 2.9: Basic structure obtained by employing weight sharing. Each plane (appearing in the right) is the result of convolving the same set of weights through the input image (at the left). Different weights are used for different planes.

Finally, the third aspect is one that results from the two above. By applying many convolutional layers one after the other, one can achieve what is called hierarchical feature extraction, where layer after layer the "abstraction" of the features that the layers learn keeps on increasing.

There are other components usually employed in convolutional networks besides the actual convolutional layers, such as max-pooling layers. The units of these layers are

assigned with the maximum value of a small number of nearby units in the preceding layer. Their goal is to reduce the dimension of the representation and create invariance to small shifts and distortions (LeCun et al., 2015). The typical architecture of a simple convolutional network includes two or three stages, each composed of a convolutional layer, non-linear activation (typically ReLU), and pooling layer. These stages are usually followed by two or three fully-connected layers. Figure 2.10 illustrates the typical organization of a convolutional network.

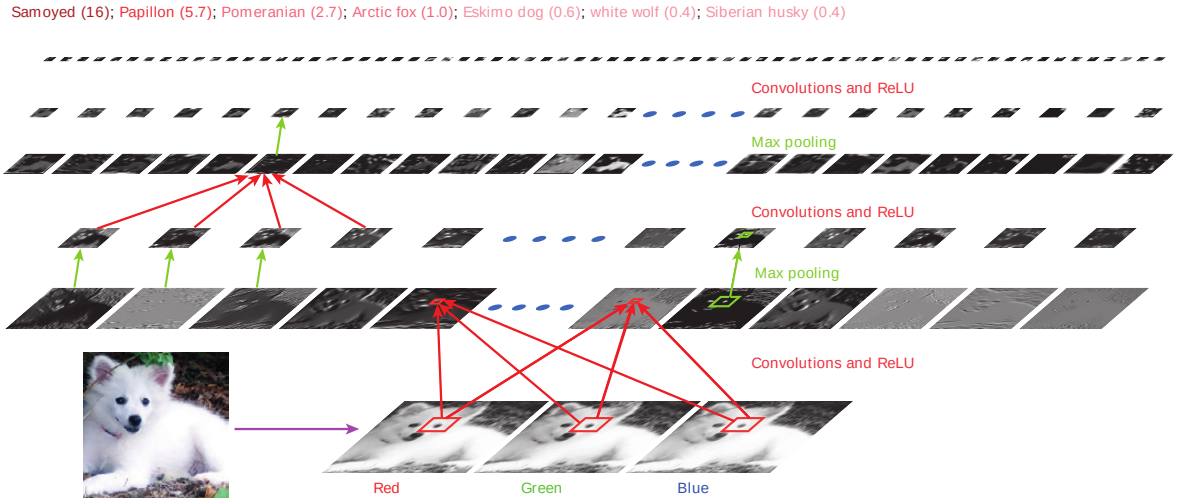


Figure 2.10: Inside a convolutional network. The outputs of each layer (horizontally) of a typical convolutional network architecture applied to the image of a Samoyed dog (bottom left; and the RGB (red, green, blue) inputs, bottom right). Each rectangular image is a feature map corresponding to the output for one of the learned features, detected at each of the image positions. Information flows bottom up, with lower-level features acting as oriented edge detectors, and a score is computed for each image class in the output. Source: LeCun, Bengio, and Hinton (2015).

Recent works, like those of Yarotsky (2018) and Zhou (2019), provide universal approximation theorems for special cases of convolutional neural networks. However, the general universal approximation problem for CNNs is not yet solved.

2.4 Optimization algorithms

Optimization is the process of tuning the parameters of a network, typically its weights, to improve its accuracy (or some other metric of correctness). Of course, optimally we would like to find the best possible parameters for the network, which means finding the global minimum of its cost (or loss) function. Sadly, we lack the tools to efficiently and effectively chart the cost function's landscape thoroughly for the global minimum — we can only aim (with certainty) for a local minimum. Nevertheless, this may not be as bad as it seems, as it was recently shown that, under

few assumptions and conditions (such as overparameterization), the quality (i.e., depth) of local minima tends to improve toward the global minimum as a network's depth and width increases (Kawaguchi, Huang, & Kaelbling, 2019).

Many optimization techniques are used in neural networks, the most basic one being Gradient descent (GD). However, as will be seen later, GD is not appropriate for large-scale scenarios, where neural networks are now typically employed. A more suitable while still straightforward variation is Stochastic gradient descent (SGD), which is also the starting point of many of the more evolved approaches. Examples of the more advanced variations are stochastic gradient descent with momentum (Polyak, 1964; Rutishauser, 1959), RMSprop (G. Hinton, Srivastava, & Swersky, 2012) and Adam (Kingma & Ba, 2015). Below we will present GD, SGD, and few other variants. Goodfellow et al. (2016, Chap. 8) provides a more complete review of optimizers for neural networks.

2.4.1 Gradient descent

Gradient descent (or steepest descent) is a first-order iterative optimization algorithm used for finding the minimum of a function. The idea is to, at each iteration, update the parameters of a function in the direction in which it decreases most rapidly. Letting f denote the function to be minimized, and w its vector of weights (i.e., parameters), the gradient descent strategy can be written

$$w_{t+1} = w_t \underbrace{-\alpha \nabla f(w_t)}_{\Delta w_t},$$

where Δw_t is the step in the weight space to take at time (i.e., iteration) t, α is a parameter usually called the learning rate (or step size) that controls the magnitude of the update, and the gradient $\nabla f(w_t)$ is the vector of first partial derivatives,

$$\nabla f(w_t) = \left(\frac{\delta f(w_t)}{\delta w_{t,1}}, \ldots, \frac{\delta f(w_t)}{\delta w_{t,n}} \right),$$

where n is the number of weights.

In most, if not all, modern applications of machine learning, $\nabla f(w)$ is not the true gradient of $f(w)$ but only an approximation. Computing the true gradient requires a pass over all the training data, the cost of which is usually prohibitively expensive. Instead, randomized alternatives are usually employed, namely SGD and mini-batch gradient descent. In the former case, the gradient is estimated from a single training sample, wheres in the latter it is estimated on a mini-batch, i.e., a number of randomly selected training samples. However, note that the term "SGD" is commonly used to

refer to either pure SGD or mini-batch SGD in most frameworks and academic works, and the batch size is indicated as needed.

In the lack of the true gradient, the (stochastic) gradient descent update rule becomes (Darken & Moody, 1992)

$$\Delta w_t = -\alpha \left(\nabla f(w_t) + \xi(w_t) \right),$$

where $\nabla f(w_t)$ represents the true, inaccessible gradient, and $\xi(w_t)$ an approximation of its error, typically assumed to have zero mean (which is true for an unbiased estimator).

Notice that the error $\xi(w_t)$ introduced by SGD may in some cases be beneficial. First, as was previously said, in contemporary machine learning problems, it is usually just not possible to compute the true gradient. Thus SGD (or one of its variations) is not merely a choice — it is the only option available. Second, the noise in the gradient may in fact help the system evade shallow local minima (by the minima not being deep enough to resist the erratic behavior introduced by the noise).

As highlighted by Sutton (1986), the way (S)GD works has two shortcomings that may affect its effectiveness in learning. One of its problems is that gradient descent prioritizes changing weights which, during training, have shown to be important, instead of using perceived useless ones. This gives rise to feature interference when learning different patterns. Another problem is that GD is a particularly poor procedure for traversing ravines (places which curve more sharply in certain directions than in others), which are common landscapes in neural networks, since it is constantly changing directions instead of sticking to a stable one. Moreover, gradient descent procedures make their largest changes to the weights whose first partial derivatives are greatest (in absolute value). While this may seem reasonable, there are at least two reasons for doing the exact opposite (Sutton, 1986). First, a small derivative frequently indicates a shallow, mildly-curving part of the surface, in which large steps should be made to leave it quickly. In contrast, a large derivative may indicate a very steep and sharply curving region, where one should be more prudent with the step size used to avoid instability. Second, weights with large derivatives will most likely be those that already play an important role in the behavior of the network, as those are the ones affecting it the most. However, when the network has to adapt to new data (creating new features in the process), it should do so minimizing the interference with the already existing features, which suggests that these weights that have larger derivatives should be changed least. This is not what happens. Later we will see how some of these shortcomings may be circumvented by using more complex strategies.

2.4.2 Momentum

As we saw in the discussion above concerning the shortcoming of gradient descent when traversing narrows valleys, rather than proceeding in the single direction of the gradient at each step, we should follow a direction that is somehow constructed to be conjugate to, as far as possible, the previous directions traversed. One of the simplest methods trying to achieve this is called Momentum (Polyak, 1964; Rutishauser, 1959), which combines previous gradients with the current one in a Exponential moving average (EMA) fashion[11].

Essentially, recall that in "vanilla" gradient descent we have the update

$$w_{t+1} = w_t - \alpha \nabla f(w_t).$$

The idea of momentum is to provide GD with "short-term memory", resulting in

$$\begin{aligned} v_0 &= 0 \\ v_{t+1} &= \beta v_t + \alpha \nabla f(w_t) \\ w_{t+1} &= w_t - v_{t+1}, \end{aligned}$$

where α and β control the weight given to the current gradient and to the past accumulation of gradients, respectively. Typically, β is used with a value of 0.9, whereas α is treated as the "learning rate" of the system and is usually optimized manually.

While the change may seem tiny, it affects the behavior of GD significantly, mainly by dampening the oscillations that it faces near shallow valleys or ravines. It works by, in some sense, adding inertia to GD's trajectory, which smooths and accelerates it. The effect is illustrated in Fig. 2.11.

Momentum turned out to be a core method used in conjunction with SGD, allowing works such as (Ilya Sutskever, Martens, Dahl, & Hinton, 2013) to achieve results previously obtained only with Hessian-free optimization[12]. Goh (2017) provides a straightforward discussion about momentum and why and how it works.

[11] It is not really a EMA as the weights of the values being averaged do not necessarily sum to one, but the principle remains.

[12] The authors actually combined momentum with a well-designed random initialization and a slowly increasing schedule for the momentum parameter. They concluded that both the initialization and the momentum were crucial towards obtaining their results.

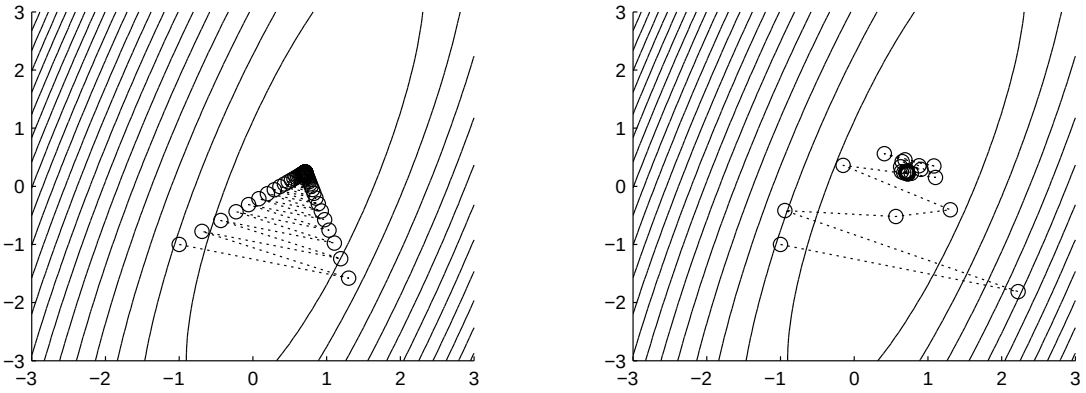


Figure 2.11: Iterates of gradient descent (left panel) and gradient descent with momentum (right panel), starting at $(-1, -1)$. Source: Sun (2015).

2.4.3 AdaGrad

The high-dimensional non-convex nature of the optimization problems at play in neural networks may lead to different sensitivities on each dimension. A given learning rate may be just too small to be efficient in some dimension but too large in another. This realization has led to the development of optimization algorithms that employ per-parameter learning rates.

Notably, one of the earliest such algorithms used for this purpose is AdaGrad (Adaptive Gradient) (Duchi, Hazan, & Singer, 2011), which individually adjusts the learning rate of each parameter by scaling it inversely proportionally to the square root of the sum of all previous squared values of the gradient.

Letting $g_t \equiv \nabla f(w_t)$, the update rule of this algorithm is

$$w_{t+1} = w_t - \eta \, G_t^{-\frac{1}{2}} g_t,$$

with η being the global learning rate, and $G_t = \sum_{i=1}^{t} g_i g_i^\intercal$.

Due to the computational complexity of computing $G_t^{-\frac{1}{2}}$, the update rule actually used in practice is

$$w_{t+1} = w_t - \eta \, \mathrm{diag}\,(G_t)^{-\frac{1}{2}} g_t.$$

To better understand what this means we will now unfold the operations. First,

letting m denote the number of parameters of the neural network, we have

$$\begin{aligned}
\operatorname{diag}(G_t) &= \operatorname{diag} \sum_{i=1}^{t} g_i g_i^{\intercal} \\
&= \operatorname{diag} \sum_{i=1}^{t} \left(\begin{bmatrix} g_i^{(1)} \\ g_i^{(2)} \\ \vdots \\ g_i^{(m)} \end{bmatrix} \begin{bmatrix} g_i^{(1)} & g_i^{(2)} & \cdots & g_i^{(m)} \end{bmatrix} \right) \\
&= \sum_{i=1}^{t} \begin{bmatrix} \left(g_i^{(1)}\right)^2 & 0 & \cdots & 0 \\ 0 & \left(g_i^{(2)}\right)^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \left(g_i^{(m)}\right)^2 \end{bmatrix}.
\end{aligned}$$

Finally, letting $G_t^{(i)} = \sum_{j=1}^{t} \left(g_j^{(i)}\right)^2$ and rewriting the update rule in this unfolded manner, we obtain

$$\begin{aligned}
w_{t+1} &= w_t - \eta \operatorname{diag}(G_t)^{-\frac{1}{2}} g_t \\
&= w_t - \eta \begin{bmatrix} \frac{1}{\sqrt{G_t^{(1)}}} & 0 & \cdots & 0 \\ 0 & \frac{1}{\sqrt{G_t^{(2)}}} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{\sqrt{G_t^{(m)}}} \end{bmatrix} \begin{bmatrix} g_t^{(1)} \\ g_t^{(2)} \\ \vdots \\ g_t^{(m)} \end{bmatrix} \\
&= w_t - \eta \begin{bmatrix} \frac{g_t^{(1)}}{\sqrt{G_t^{(1)}}} \\ \frac{g_t^{(2)}}{\sqrt{G_t^{(2)}}} \\ \vdots \\ \frac{g_t^{(m)}}{\sqrt{G_t^{(m)}}} \end{bmatrix}
\end{aligned}$$

As the expression shows, by using just $\operatorname{diag}(G_t)$ instead of the complete G_t, we avoid the expensive computation of its root and inverse, which would be impractical in high-dimensional scenarios. With this simplification, the added complexity is just linear in the number of parameters.

2.4.4 RMSProp

One problem with AdaGrad is that it typically induces a premature and excessive decrease in the effective learning rate, caused by accumulating all past gradients. Furthermore, while AdaGrad enjoys some desirable theoretical properties in the context of convex optimization (Goodfellow et al., 2016), in practice, it reveals subpar for the training on neural networks.

One alternative algorithm designed to perform better in the nonconvex setting is RMSProp (G. Hinton et al., 2012), which modifies AdaGrad by changing the gradient accumulation into an exponentially weighted moving average of the squared gradients. Letting R_t denote the accumulation variable (initially set to 0), the update rule results in

$$R_t = \rho R_{t-1} + (1-\rho) \begin{bmatrix} \left(g_t^{(1)}\right)^2 \\ \left(g_t^{(2)}\right)^2 \\ \vdots \\ \left(g_t^{(m)}\right)^2 \end{bmatrix}$$

$$w_{t+1} = w_t - \eta \begin{bmatrix} \frac{g_t^{(1)}}{\sqrt{R_t^{(1)}}} \\ \frac{g_t^{(2)}}{\sqrt{R_t^{(2)}}} \\ \vdots \\ \frac{g_t^{(m)}}{\sqrt{R_t^{(m)}}} \end{bmatrix},$$

where ρ denotes the weight given to the current iteration's gradient in relation to the past history. A value of 0.9 is typical.

In practice, RMSProp has become one of the most commonly used optimization methods for deep learning, since its gradient scaling strategy makes it desirable for traversing both the nonconvex regions of the weight landscape and the convex structures, and, besides, it is efficient to compute.

2.4.5 Adam

Another very widely used adaptive optimization algorithm is Adam (Kingma & Ba, 2015), getting its name from "adaptive moments". Adam tries to combine RMSprop with momentum. First, momentum is applied to acquire an estimate of the first-order moment of the gradient (prior to any rescaling), by computing a Exponentially weighted moving average (EWMA) of it. Second, Adam computes the (uncentered) second-

moment of the gradient, intending to scale the gradient with it (akin to what RMSProp also does).

One subtle difference introduced by Adam is that it includes bias corrections to the estimates of both moments to try to account for their initialization at the origin (notice that RMSprop does not carry this kind of correction).

Letting S_t and R_t denote the running estimates of the first- and the (uncentered) second-order moments of the gradient (respectively), both initially set to 0, the resulting update rule can be written as (with operations such as power, square root, and so on, carried element-wise)

$$S_t = \rho_1 S_{t-1} + (1-\rho_1) g_t, \quad \hat{S}_t = \frac{S_t}{1-\rho_1^t}$$
$$R_t = \rho_2 R_{t-1} + (1-\rho_2) g_t^2, \quad \hat{R}_t = \frac{R_t}{1-\rho_2^t}$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{R}_t}} \hat{S}_t,$$

where $\rho_{1,2}$ have the same meaning as ρ in RMSprop ($\rho_{1,2}^t$ are used to, in some sense, correct the initializations of R_t and S_t). Values close to one, such as 0.9 for ρ_1 and 0.999 for ρ_2, are typically used.

Currently, SGD with and without momentum, RMSProp, and Adam are the most popular optimization algorithms commonly used (Goodfellow et al., 2016). However, typically, the choice of the algorithm used is not based on some rational judgment, but instead by the familiarity of the user with it, which helps at tuning its parameters.

2.5 Initialization strategies

A point sometimes overlooked when training artificial neural networks — but one that may have profound implications in the results obtained — is the initial configuration of weights assigned to a network, also called its initialization. It is known that the initialization of a network plays a pivotal role in the training of neural networks (Glorot & Bengio, 2010; He, Zhang, Ren, & Sun, 2015; LeCun, Bottou, Orr, & Müller, 1998; Yam & Chow, 2000). For example, Chapelle and Erhan (2011) showed that, by using the initialization strategy proposed by Glorot and Bengio (2010) alongside stochastic gradient descent, one could train the autoencoder of G. E. Hinton and Salakhutdinov (2006) to better results than its original authors did. Moreover, by combining momentum with a well-chosen random initialization scheme, Ilya Sutskever et al. (2013) achieved comparable performance to that of Hessian-free methods in their tests.

The initialization strategies in use nowadays are mostly heuristic, seeking to achieve some desired properties at least during the first few iterations of training. However,

it is usually not clear which properties are kept during training nor how they vanish. Moreover, it is also not clear why some initializations are better from the point of view of optimization, whereas from the point of view of generalization they prove to be worse. The current understanding between the effect of the initial configuration of weights and its effect on generalization is notably lacking (Goodfellow et al., 2016, Sec. 8.4).

Usually, the initial configuration of weights of a network is drawn from uniform or Gaussian distributions, centered at zero, whereas biases are initialized with zero or with some small constant. It is important that the initialization of the weights breaks the symmetry between different units (i.e., avoid situations where two hidden units with the same activation function and connected to the same nodes start with the same initial parameters). Meanwhile, while the choice of uniform or Gaussian distributions does not seem to be particularly important (Goodfellow et al., 2016), the scale of the distribution (from which the initial weights are drawn) does. The most common initialization strategies — those of Glorot and Bengio (2010), He et al. (2015), and LeCun, Bottou, et al. (1998) — all define rules for choosing the variance that the distribution from which the initial weights are draw should have. Below we will briefly present and discuss a few common (or recent) initialization strategies. We will also discuss an interesting recent topic, the lottery ticket hypothesis, although details of this discussion are postponed to Sec. 5.3.1.

2.5.1 Glorot's initialization

The most commonly used initialization scheme is perhaps Glorot uniform[13] (Glorot & Bengio, 2010). It is the default strategy in both Keras[14] and Tensorflow[15]. Glorot's initialization assigns the initial weights w of a layer with m inputs and n outputs by sampling them from a distribution with variance

$$\mathrm{Var}\left[w\right] = \frac{2}{m+n}.$$

If the distribution used is uniform (the case of Glorot's uniform initialization), this results in

$$w \sim U\left(-\frac{\sqrt{6}}{\sqrt{m+n}}, \frac{\sqrt{6}}{\sqrt{m+n}}\right).$$

[13]Also known as Xavier uniform initialization — Xavier Glorot is the first author of the paper where the method was presented. Some libraries use the author's first name to refer to the method, whereas others use his last name. In the paper itself, the authors call it the *normalized initialization*.

[14]Check, e.g., https://keras.io/layers/core/ and https://keras.io/layers/convolutional/. Accessed 2019-11-05.

[15]https://www.tensorflow.org/versions/r1.15/api_docs/python/tf/get_variable. Accessed 2019-11-05.

Glorot's initialization is the result of a compromise between two observations made by its authors. On the one hand, one wants to have the same activation variance across all the layers of a network, i.e., to have the variance of the signal that passes through each layer of a network, during a forward pass, approximately the same. On the other hand, one also wants to have the same gradient variance across all layers, i.e., to have the variance of the gradient at each layer, during a backward pass, approximately the same. The reason behind this is to avoid exploding or vanishing gradients. These two constraints would push the variance of the weights w of a layer with m inputs and n outputs to have

$$\mathrm{Var}\left[w\right] = \frac{1}{m} \quad \text{and} \quad \mathrm{Var}\left[w\right] = \frac{1}{n},$$

respectively. Since the two constraints may not be met simultaneously, the authors proposed using the previously mentioned expression,

$$\mathrm{Var}\left[w\right] = \frac{2}{m+n},$$

which is exact if $m = n$ (i.e., the number of input and output nodes of the layer is the same).

Finally, note that the analysis behind Glorot's initialization assumes that the network is in a linear regime during the initial steps of training. Despite the assumption being possibly violated, it seems that many strategies designed with similar assumptions in mind happen to perform reasonably well even on nonlinear cases (Goodfellow et al., 2016).

2.5.2 He's initialization

He's initialization (He et al., 2015, Sec. 2.2) is, to some extent, based on Glorot's. However, the authors of the He initialization argue that the assumption that a network starts in a linear regime does not hold for networks employing the ReLU activation. As a result, they derive a theoretically more sound initialization for ReLU networks.

In their paper, the authors follow approximately the same analysis of Glorot and Bengio (2010), and conclude that for the weights w of a layer with m inputs, the variance of the weights should be[16]

$$\mathrm{Var}\left[w\right] = \frac{2}{m}.$$

For a uniform distribution, this means that the weights should be drawn from

$$w \sim U\left(-\frac{\sqrt{6}}{\sqrt{m}}, \frac{\sqrt{6}}{\sqrt{m}}\right).$$

[16]More precisely, the expression presented here scales the forward signal of the network, which is the most common approach. In their paper, the authors point out that if the initialization appropriately scales the forward signal, then it also scales the backward signal, and vice-versa.

2.5.3 LeCun's initialization

LeCun's initialization (LeCun, Bottou, et al., 1998, Sec. 4.6) is designed for sigmoid-based networks. The idea is to initialize the network so that the sigmoids are activated mostly on their linear regime (i.e., around zero), since elsewhere, in the saturation regions, the derivatives become too small, which prevents proper training. This goal is achieved by coordinating the normalization of the training set, the choice of the sigmoid, and the choice of weight initialization. To achieve the latter goal (the part that we are interested in in this section), the weights w of a layer with m inputs should have a variance of

$$\mathrm{Var}\left[w\right] = \frac{1}{m}.$$

As a result, if the values of the initial weights are to be drawn from a uniform distribution, this results in

$$w \sim U\left(-\frac{\sqrt{3}}{\sqrt{m}}, \frac{\sqrt{3}}{\sqrt{m}}\right).$$

2.5.4 Sparse and lightning initializations

The sparse initialization (Martens, 2010, Sec. 5) is based on setting a fixed number of incoming weights connecting to each node with nonzero values, and the remaining weights to zero. The rationale for doing this is to avoid situations where the scale of the weights becomes too small, which may occur with strategies such as He's or Glorot's initializations that scale the weights based on the number of nodes of their layer.

Another initialization strategy that initializes weights sparingly is the lightning initialization (Pircher, Haspel, & Schlücker, 2018). It consists of choosing a fixed number of end-to-end paths along the network (connecting input to output nodes, through intermediate hidden nodes), setting the weights along these paths with nonzero values, and the remaining ones with zero. The motivation is that random initializations create suboptimal information flows between input and output nodes. By limiting the initial number of input-output paths and ensuring they are complete (not interrupted by weights too close to zero), the backpropagated information should be better transferred to the layers near the input. In this thesis, we carried a few tests with a novel method inspired by the lightning initialization. They are presented in Sec. 5.2.1.

3 Data, processing and tools

It takes a lot of hard work to make something simple.

— Steve Jobs

This chapter presents the datasets used for the training experiments carried in Chapters 4 and 5. Moreover, it also presents the machines where the tests were carried, the programs developed, and the frameworks used, along with other relevant information. The chapter is organized as follows. In Sec. 3.1, we present and discuss a synthetic dataset of our making, which we use mainly in Chap. 4. Afterward, in Sec. 3.2, we present the "well-known" datasets used mainly in Chap. 5. Finally, in Sec. 3.3, we present the programs we used for carrying the experiments, their interactions, the frameworks/software libraries they use, and other worthwhile considerations about, for instance, the problems experienced while developing them.

3.1 Synthetic dataset

To better understand the processes going on during the training of artificial neural networks, we felt the need to start experimenting with a toy problem. With this in mind, this section focuses mainly on presenting a simple, synthetic dataset constructed by us and used in Chap. 4. The rationale behind using this dataset is twofold. On the one hand, it allows easier comparison between statistics of the dataset itself and those found in a network during and after its training, so that any overlapping between the two can be easily identified. Using more complex datasets would make this task much more difficult, since the datasets would hide/mask such relations, making them easily missed. On the other hand, the particular rules defining the dataset allow it to be implemented perfectly by a known neural network architecture and configuration of weights, which we can use as a reference configuration. This knowledge allows one to

confront the organization acquired by a trained network with the reference architecture and reason about how similar they are, and why.

3.1.1 Generation rules

The dataset can be viewed as containing samples of a filter identifying the pattern `0110`. More specifically, an input sample x is a random binary string of length n, i.e., $x \in \{0,1\}^n$, whereas the respective output y (another binary string of length n) is computed such that its i-th bit is `1` if and only if

$$\begin{aligned} x_i &= 0 \\ x_{i+1 \bmod n} &= 1 \\ x_{i+2 \bmod n} &= 1 \\ x_{i+3 \bmod n} &= 0. \end{aligned}$$

An example of such an input/output pair of samples of length $n = 10$ is

```
in (x):   0100110011
out (y):  0001000100
```

Notice the rightmost `1` in the output above. That bit is `1` since the indices are taken modulo the length of the samples, as if the input sample is padded with its three first bits, i.e.,

```
in (x):   0100110011(010)
out (y):  0001000100
```

This is done to make both input and output samples of the same length, so that all elements of the samples have the same properties, irrespectively of the index they occupy.

As said previously, as this mapping stands, it can be interpreted as a filter matching `0110`, and it can be expressed as (all indices are taken modulo n, the length of the samples)

$$y_i = (1-x_i)x_{i+1}x_{i+2}(1-x_{i+3}), \quad i = 0,\, 1,\, \ldots,\, n-1.$$

An attractive property of this filter is that it can be implemented by a multilayer perceptron using a hidden layer of n nodes and employing the ReLU activation function, and an optional output layer also having n nodes and using the identity function as activation. Figure 3.1 illustrates the network.

Coming up with this solution is a matter of considering a few characteristics of the filter and leveraging on the ReLU nonlinearity to accommodate them. First, for an output y_i, if either x_i or x_{i+3} are `1`, the input to the respective ReLU unit should be

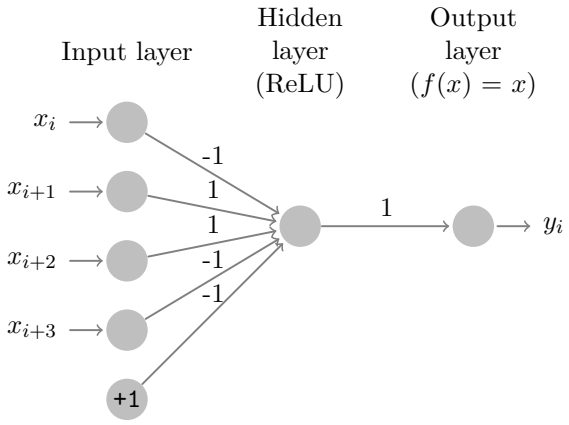


Figure 3.1: A perfect neural network implementation for the synthetic dataset.

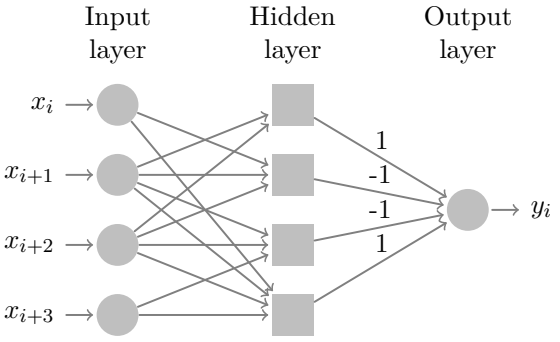
negative, independently of the value of the remaining bits, so that the ReLU's output is `0`. Second, the bias unit should be set so that, if x_i and x_{i+3} are `0` but only one of x_{i+1} and x_{i+2} is `1`, it pulls the ReLU input down to negative values (or zero) so that the ReLU outputs `0`. Finally, if x_i and x_{i+3} are `0` and x_{i+1} and x_{i+2} are `1`, then the net sum of x_{i+1}, x_{i+2}, and the bias should equal `1`. These conditions can be met by using solely weights (and bias) with absolute value 1, and adjusting their signs accordingly[1].

Despite being somewhat simple, this dataset contains several interesting properties. First, it allows the construction of arbitrarily large datasets of varying input/output widths. Second, and more importantly, the output is determined entirely and in a well-defined manner by the input in a way that can be understood and rationalized about. Furthermore, it even lends itself to a perfect implementation in a neural network. This is very important in that it allows one to compare the configuration a given network

[1] We note in passing that an old and generally unknown kind of feedforward neural network, called ProdNet (Durbin & Rumelhart, 1989), could also be used for a perfect implementation of this filter, as illustrated in the figure to the right. This kind of networks introduce "product units" which compute the weighted product of their inputs, i.e.,

$$z_j^h = \prod_i \left(a_i^{h-1}\right)^{w_{ij}^h},$$

where z_i^h denotes the input of unit i of layer h; a_i^h the output of node i of layer h, i.e., $a_i^h = f(z_i^h)$ for an activation function f; and w_{ij}^h the weight connecting node i of layer $h-1$ to node j of layer h. These units, although more expressive than the classical summation ones, lead to more difficult training due to a more complex landscape of the loss function, with more local minima. As a result, they have fallen in disuse.



ProdNet implementation of the filter without using any nonlinear activation functions. Squares denote product units, whereas circles denote conventional summation units. Omitted links have weight 0 and unlabeled ones have weight 1.

acquires to a well-known, "perfect" one, that we understand.

3.1.2 Input/output correlations

One may note that, in the synthetic dataset described above, four consecutive input bits fully determine one output bit. Particularly, output bit y_i depends (only) on inputs $x_i, x_{i+1}, x_{i+2}, x_{i+3}$, where the indices are taken modulo the length of the bit strings. This means there is a dependence (or "correlation" in a general sense) between y_i and the corresponding inputs, and none with all the other inputs. This is evidenced in Fig. 3.2.

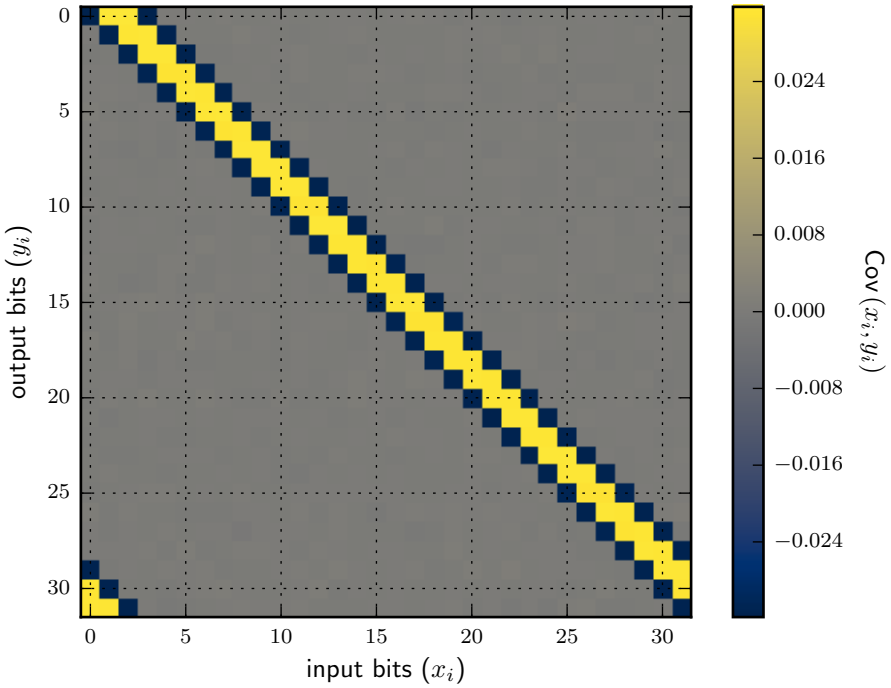


Figure 3.2: Covariance between input and output bits of the synthetic dataset (words of length 32).

The figure plots the covariance ($\mathrm{Cov}(X, Y) = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])]$) between each pair of input and output bits. The way the problem is defined gives rise to the diagonal observed (since one output bit depends solely on four consecutive input bits, counted from the index of the output), and the fact that the indices "wrap-around" originates the artifact in the bottom left corner (which is the continuation of the four input bits affecting each of the three bottom output bits).

If a trained network is evaluated with valid input words, it should produce a covariance matrix similar to the one above, since the dependence between inputs and outputs should also be present in the function the network has learned. The next logical question is whether this information exists within the parameters of a network,

without the need to explicitly evaluate the network to observe it. This implies somehow interpreting the weights of the network.

A weight can be perceived as the importance a node gives to another. This means that, in general, a node will assign larger weights to preceding nodes it deems more important, and smaller weights to less significant ones. However, comparing weights is not as trivial as it may seem at first sight. First, the magnitude of the weights depends on the scale of the signal received through the preceding nodes, which may be significantly different. For instance, if a node consistently outputs a value very close to zero, the weights that connect that node to other nodes may be large, and, still, the overall stimulation received by the receiving nodes will be negligible. Second, two (or more) weights may cancel each other out, for instance, by having opposing signs and connecting to preceding nodes that always output approximately the same values. These weights may be irrelevant for the node they connect to, but still they may have significantly large magnitudes.

Due to this, it is more natural to consider the total flow between two nodes of a network, through all the possible paths connecting them. This is advantageous due to several reasons. First, it allows the aggregation of flows that pass through different intermediate nodes, which, to some extent, embodies the effect of weights cancelling out. Second, by considering the complete paths connecting two nodes, one gets a better representation of the overall strength of the connection, since, for instance, if one large weight in a path is followed by a small one, the strength of the path will amount to "medium", as it should.

Aggregating all paths between a given input node and an output node can be achieved by considering a network stripped from all nonlinearities (i.e., a "pure" linear version of the network), providing the input node with an input of `1`, and observing the strength with which it reaches the output node.

At first sight, it may seem that considering all possible paths along all intermediate nodes and computing their overall sum is computationally inhibiting. However, it is, in fact, a very simple operation, which may even be carried for all input/output nodes at once. This happens since a linear neural network can be reduced to a single layer of weights by multiplying the network's weight matrices. In particular, ignoring biases, for a linear neural network of l layers of weights ($l+1$ layers of nodes), where w_{ij}^k denotes the weight connecting node i of layer $k-1$ to node j of layer k; a_j^k the activation of node j of layer k; and a^0 and a^l represent, respectively, the input and output of the

network, we have

$$
\begin{gathered}
a^0 \equiv x \\
a^1_j = \sum_i a^0_i w^1_{ij} \Leftrightarrow a^1 = w^{1\intercal} a^0 = w^{1\intercal} x \\
a^2 = w^{2\intercal} a^1 = w^{2\intercal} w^{1\intercal} x \\
\vdots \\
a^l = \left(w^1 w^2 \cdots w^l\right)^\intercal x.
\end{gathered}
$$

This means the linear network can be collapsed into a single layer of weights (the product $w^1 w^2 \cdots w^l$) and afterward evaluated with $x = \mathbb{1}_i$ for computing the propagation activity of node i. However, evaluating the aggregation matrix independently for each input node is unnecessary, since, in fact, the collapsed weight matrix already contains the propagation activity for all pairs of input/output nodes, which is what we are looking for. This procedure is used in Sec. 4.4.1 to compute the strength of the flows of trained neural networks and compare them against the correlations of the dataset presented above, in Fig. 3.2.

3.1.3 Instantiation for training

Based on the generation rules described in Sec. 3.1.1 we created an instance of the synthetic dataset with $n = 30$ and containing 100 000 training and 100 000 testing samples[2]. The train and test samples are created by generating binary strings of length n and where each bit has a probability p of being a `1` (and $1-p$ of being a `0`). We used $p = 0.5$. These strings constitute the input samples. The output samples are obtained by computing the images of the strings generated. This dataset is used for the experiments carried in Chapter 4. Notice that its size (number of train and test samples summed) represents only at most $\approx 0.02\%$ of the dataset creatable with $n = 30$. Despite this, preliminary tests showed that more massive datasets did not affect in any significant way training. Hence, this smaller size was chosen to keep the memory requirements of the training program small. Listing 3.1 shows an example of how such a dataset can be created (for conciseness $n = 15$ is used).

Finally, note that, due to the sparsity of ones in the output, a network that is hard-coded to output only `0`'s will achieve an accuracy of $^{15}/_{16} \approx 94\%$. This is one of the reasons why the training of the networks using this dataset, in Chapter 4, was carried for so long — to allow the networks to converge to solutions where they provide

[2]Note that, usually, datasets are split in 80–20% or 70–30% train/test ratios, mainly due to the scarcity of data samples. However, since we do not have this problem, we opted for a 50/50% split, in order to have a more accurate estimation of the networks' test loss.

```
>>> import generator as gen
>>>
>>> g = gen.Generator(seed=0)
>>> train = g.generate(n=15,p=0.5,k=10)
>>> test = g.generate(n=15,p=0.5,k=10)
>>>
>>> for x,y in zip(*train):  print(x, '->', y)
...
[1 1 1 1 0 1 0 1 1 0 1 1 1 1 0] -> [0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
[0 0 1 1 1 1 1 0 1 0 1 0 1 1 0] -> [0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
[0 1 0 1 0 1 1 1 1 1 0 0 1 0 1] -> [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[1 0 0 0 0 1 0 1 0 0 0 1 0 0 0] -> [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 1 0 1 0 1 0 1 1 1] -> [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 1 0 1 0 1 1] -> [0 0 0 0 0 0 0 1 0 0 0 0 1 0 0]
[0 1 0 1 0 0 1 0 1 0 1 0 1 1 0] -> [0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
[1 1 1 0 1 0 1 1 0 1 0 1 1 1 1] -> [0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[1 1 1 1 0 1 0 0 1 0 1 0 0 0 1] -> [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[1 1 1 1 0 1 0 0 1 1 1 0 1 1 1] -> [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

Listing 3.1: Example of how to create an instance of the synthetic dataset with $n = 15$, $p = 0.5$, and containing 10 samples for the training and testing sets.

close to perfect outputs (with an accuracy of almost 100%, which a network that only outputs 0's is incapable of achieving).

3.2 Real-world datasets

This section introduces the real-world datasets used in Chapter 5, MNIST and HASYv2, and describes them briefly. The usage of standard datasets is essential to draw sensible conclusions about the generality of our results on more complex problems.

3.2.1 MNIST

The MNIST (Modified National Institute of Standards and Technology) dataset was initially presented by LeCun, Bottou, Bengio, Haffner, et al. (1998) and is found at http://yann.lecun.com/exdb/mnist/. It is a large database of handwritten digits, constructed from National Institute of Standards and Technology (NIST)'s Special Database 1 (SD-1) and Special Database 3 (SD-3)[3]. While SD-1 and SD-3 contain binary images, MNIST's images are greyscale (we will see why below). SD-3 was initially conceived as NIST's training set, whereas SD-1 was its test set. However, SD-3 happens to be much "cleaner" and easier to recognize than SD-1. The reason for this is that SD-3 was collected among Census Bureau employees, while SD-1 was collected among high-school students. The disparity between the difficulty of the two datasets made it

[3]NIST's database was published as a complete collection by Grother (1995). It was recently re-released in a more modern file format in 2016 (see https://www.nist.gov/srd/nist-special-database-19).

necessary to mix them, since, to draw proper conclusions from learning experiments, the results should be independent of the particular choice of train and test sets among the complete set of samples.

To construct MNIST, SD-1 and SD-3 were split into train and test sets. The split was performed based on the writer of the samples so that samples from half the writers went to the train set and samples from the other half went to the test set. This resulted in a SD-1 train and test sets with approximately 30 000 samples each, and a SD-3 train and test sets with approximately another 30 000 samples each. This originated MNIST's train and test sets with approximately 60 000 samples each. However, in their original paper, LeCun, Bottou, et al. (1998) used only a subset of 10 000 samples for the test set (5000 from SD-1 and 5000 from SD-3). This originated the typical 60 000 train/10 000 test split used nowadays.

NIST's samples received some processing before being merged to MNIST. The original black and white images were size normalized to fit in a 20×20 pixel box, while retaining their aspect ratio. The resulting images became greyscale instead of "pure" black and white, an outcome of the anti-aliasing performed by the normalization. Then, the images were centered in a 28×28 pixel grid. This was accomplished by translating the 20×20 image to position its center of mass at the center of the 28×28 grid[4]. Figure 3.3 shows randomly selected samples from MNIST's training set.

Figure 3.3: Sample images from MNIST train dataset.

[4]There are other versions of the dataset used in the experiments of LeCun, Bottou, et al. (1998), but they are not used nowadays. We are presenting only the version that originated MNIST.

3.2.2 HASYv2

The HASYv2 dataset[5] (Thoma, 2017) is somewhat similar to MNIST, in that both are very low resolution handwritten datasets. However, MNIST contains only 10 classes, whereas HASYv2 contains 369, including Arabic numerals (`0-9`), Latin characters (`a-z`, `A-Z`), and many other mathematical symbols (e.g., α, x, $\cdot$, ...). For the complete list of classes of the dataset, refer to (Thoma, 2017, Tables VI to XIV). Moreover, while MNIST is greyscale, HASYv2 is "pure" black and white.

HASYv2 is derived from the Handwriting Recognition Toolkit (HWRT), first used and described by Thoma (2014). HWRT is an online recognition dataset that does not store samples as images but instead as point-sequences. HWRT (and consequently HASYv2) are merged datasets. Almost 92% of HWRT comes from Detexify[6], while the remaining recordings come from Write Math[7]. The goal of both projects is to allow users to identify LaTeX commands in situations where the users know the glyph of the symbol they want (its "shape"), but do not know the particular name of the LaTeX command that generates it. After being collected, the recordings were manually inspected to remove unreasonable symbols (Thoma, 2017). HWRT and HASYv2 have the same recordings and number of classes; HASYv2 is the result of rendering HWRT into 32×32 pixel boxes, to simplify its usage.

It has the peculiarity that, in some cases, similar images belong to different classes. This situation arises due to HASYv2's author making the distinction between the meaning of *symbol* and *glyph*. He defines a symbol as an atomic semantic entity with a given visual appearance, whereas a glyph is a single typesetting entity. Thus (Thoma, 2017),

- Two different symbols can have the same glyph. For instance, the symbols generated by the LaTeX commands `\sum` and `\Sigma` both render to $\sum$ but have different semantics, and hence are different symbols.
- Two different glyphs can have the same semantic meaning. For example, both `\phi` (ϕ) and `\varphi` (φ) represent the small Greek letter "phi". As a result, even though they have the same meaning, since they have different visual appearances, they are different symbols.

However, note that there is not a direct mapping between symbols and LaTeX commands. As an example, the LaTeX commands `\alpha` (α) and `\upalpha` ($\mathrm{\alpha}$)[8] render to different

[5] https://zenodo.org/record/259444
[6] http://detexify.kirelabs.org/
[7] http://write-math.com/
[8] These are the unslanted variants of the Greek lowercase letters, which can be found, for instance, in the `upgreek` LaTeX package. It is conventional to use unslanted letters for uppercase Greek, and slanted letters for lowercase Greek (Knuth, 1986, p. 434).

glyphs. Nevertheless, they have the same semantic meaning and are hand-drawn the same way. Consequently, they are the same symbol.

The HASYv2 dataset contains around 168 000 black and white images of size 32×32, each labeled with one of 369 labels. Figure 3.4 shows some samples of the dataset.

Figure 3.4: Sample images from HASYv2 train dataset (reverse color).

The dataset is unbalanced. The ten classes with most recordings represent around 16% of the dataset. Figure 3.5 plots the distribution of the samples.

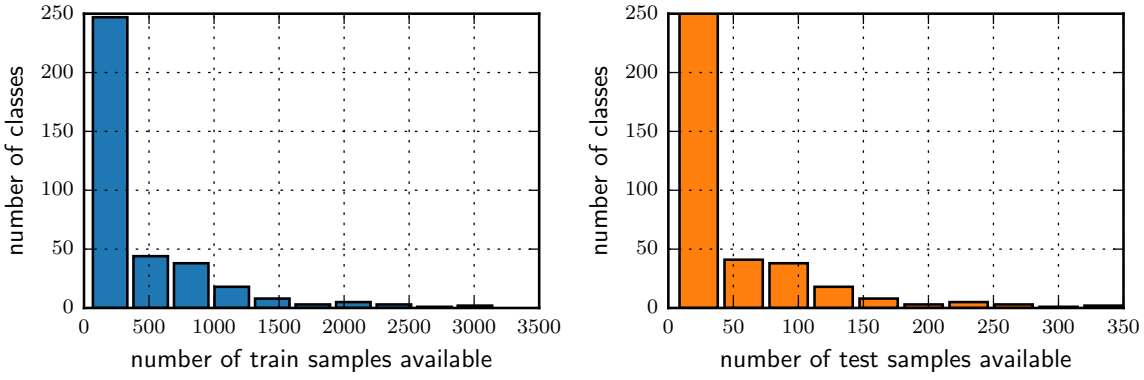


Figure 3.5: Distribution of the HASYv2 samples among classes.

To load the dataset we used the script `hasy_tools.py`[9], which is developed and maintained by HASYv2's author. We used the train/test split that the tool provides.

[9]https://github.com/MartinThoma/HASY/blob/master/hasy_tools.py

However, note that during development we found a possible source of errors in the script. It sets the random number generators of both Python and NumPy to a fixed value, which may affect the results of the experiments made with it (since different test runs will lead to the same results). The error was fixed in the local installations used in this work, and the author of the package was notified of the error. Nevertheless, at the time of writing, the error has not yet been fixed upstream.

3.3 Data pipeline

In this section we present the architecture of the programs developed and used to carry our training experiments, as well as the most important frameworks used by them. We start by looking into the overall architecture of the solution and its components in Sec. 3.3.1. Then, in Sec. 3.3.2 we present the main frameworks/libraries used with them, and in Sec. 3.3.3 we briefly discuss the fundamental differences between two of them, TensorFlow and PyTorch. Finally, in Sec. 3.3.4, we present the specifications of the machines and versions of the software used.

3.3.1 Architecture

The architecture conceived is organized into three main entities: a storage server, workers, and clients. The architecture is depicted in Fig. 3.6. The idea is to have a storage server that is contacted by both workers and clients to store and retrieve results, respectively. The workers are machines where training experiments are carried, usually having significant processing power. The clients are typically less powerful machines, such as laptops, where the results of the experiments are analyzed. However, one should note that the actual separation between the three entities is, to a large extent, only conceptual. At some point, a machine typically used as a worker may be used to process some of the results, functioning as a client, or vice-versa. Moreover, as will be seen in Sec. 3.3.1.1, the machine currently being used as the storage server also functions as a worker. The three entities are discussed in more detail in the sections that follow.

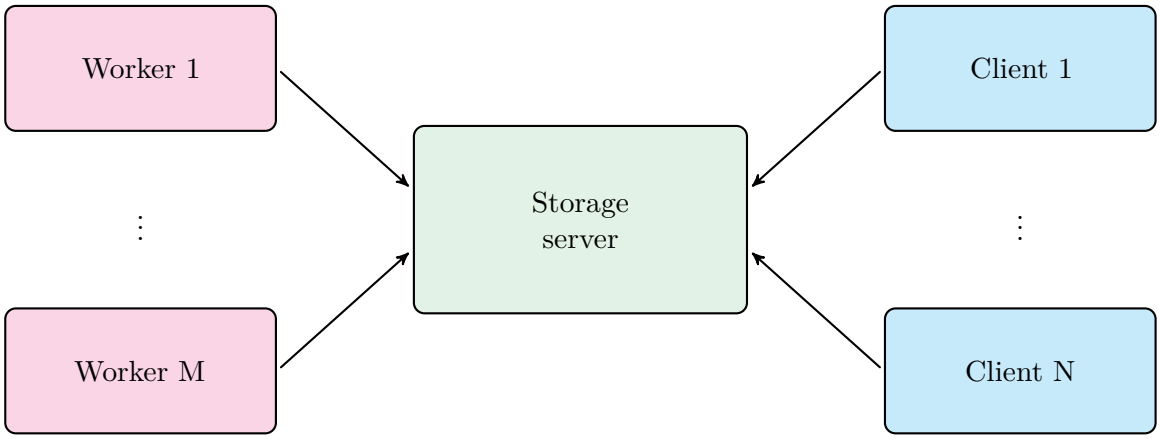


Figure 3.6: Data pipeline architecture. Workers contact a storage server to deposit results of training. Clients contact the storage server to retrieve the training results to process and analyze them.

3.3.1.1 Storage server

The storage server is the machine responsible for storing the results of the training experiments. It is a node running a Docker container[10] of a Postgres Database management system (DBMS). The container is based on the official Postgres image[11]. The database system is used to store the results of the experiments alongside other metadata associated with them, for instance, the learning rate used, or the configuration of the network trained. There are two main databases used for this purpose, one used for the experiments with the synthetic dataset, another for the remaining tests, with the real-world datasets. The former is significantly more structured than the latter, since it was designed to accommodate more variable test scenarios.

Access to the storage server is mediated through the methods of the class `Com` found in file `com.py`[12]. The class holds configurations to access the server and handles the establishment of a connection between the contacting peer (a worker or client) and the server automatically. In case the peer cannot reach the storage server directly (due to, for instance, firewall restrictions), the connection is automatically routed through a Secure Shell (SSH) tunnel using a known host as a proxy (this is also done automatically, and it is particularly useful for clients connecting the storate server from outside campus and without a Virtual private network (VPN) configured). The class implements a context manager (meant to be used with Python's `with` statement) to achieve this behavior in a transparent manner. An example is given in List. 3.2.

The schemas of the databases are presented in Figs. 3.7 and 3.8, for the experiments involving the synthetic and the real-world datasets, respectively. The database for

[10]https://www.docker.com/
[11]https://hub.docker.com/_/postgres
[12]https://github.com/rj-jesus/msc-thesis/blob/master/postgres/com.py

```
>>> from com import Com
>>>
>>> with Com() as com:
>>>     # Use `com' to access the database.
>>>     # Internally, the connection may be routed
>>>     # through a SSH tunnel automatically.
>>>     # The simplest method, `get', is used for direct
>>>     # `SELECT' SQL statements, e.g.,
>>>     results = com.get(f"""
...         SELECT ...
...         FROM ...
...         WHERE ...
...         """,
>>>         to_dict=True)
>>>
>>>     for result in results:
>>>         # ...
```

Listing 3.2: Example of how to create a connection to the storage server using the class `Com`.

the experiments involving the synthetic dataset is overall more complex than the database for the experiments with the real-world datasets. This is mainly due to the unpredictability of the experiments that we wanted to carry with the synthetic dataset, which was used during an early exploratory stage of the work (Chap. 4). The experiments involving the real-world datasets were more targeted, which led to the need of a less complex database.

The actual machine used as the host for the storage container changed as time passed by. This happened mainly due to changes to the infrastructure where the storage server and the workers were kept. It started as an instance of a Proxmox Virtual Environment[13], but soon moved to an instance of an OpenStack platform[14]. However, it has since been moved to a bare-metal server, which is also used as the main worker node (this was done due to storage space constraints, and to reduce the cost of input/output operations when running training experiments).

From the point of view of both the workers and the clients, changing the storage server is a matter of updating the IP address of the machine in com.py and its access credentials. Currently, the workers use only the storage container on the bare-metal server to deposit results. In contrast, clients use both the container on bare-metal and the one on the OpenStack instance, the former for more recent and the latter for older results. The results in the Proxmox container were migrated to OpenStack's, since the Proxmox environment was completely dismantled.

Even though this varies considerably, our typical test runs each occupies approximately 1 GB in disk space. The largest culprit for this substantial size is us periodically

[13] https://www.proxmox.com/
[14] https://www.openstack.org/

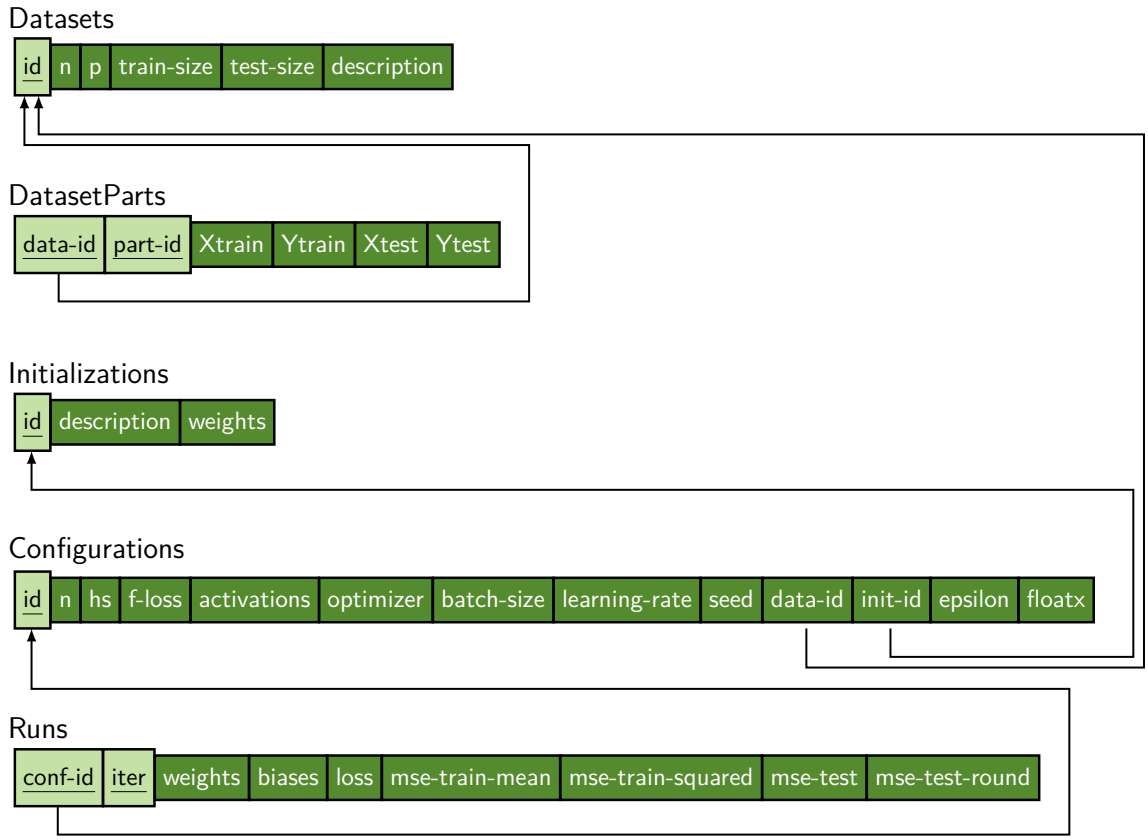


Figure 3.7: Schema of the database used for the experiments with the synthetic dataset.

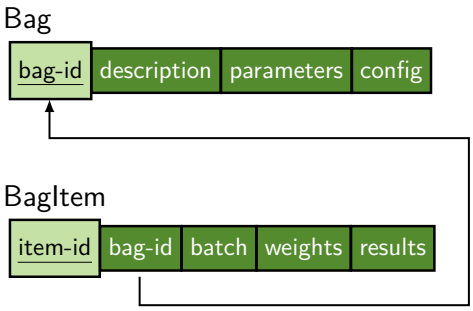


Figure 3.8: Schema of the database used for the experiments with the real-world datasets.

collecting the values of the weights of a network along its training. However, this is rarely something that we can avoid, since analysing them is a core part of this work.

3.3.1.2 Workers

The workers are machines where training experiments are carried and whose results are stored in the storage server. They are typically servers with reasonable computational power. However, there is nothing stopping an ordinary laptop or "budget" computer from being used as a worker. As long as it reports the results of training to the storage

server, it qualifies as a worker in this architecture. The general organization of workers is depicted in Fig. 3.9.

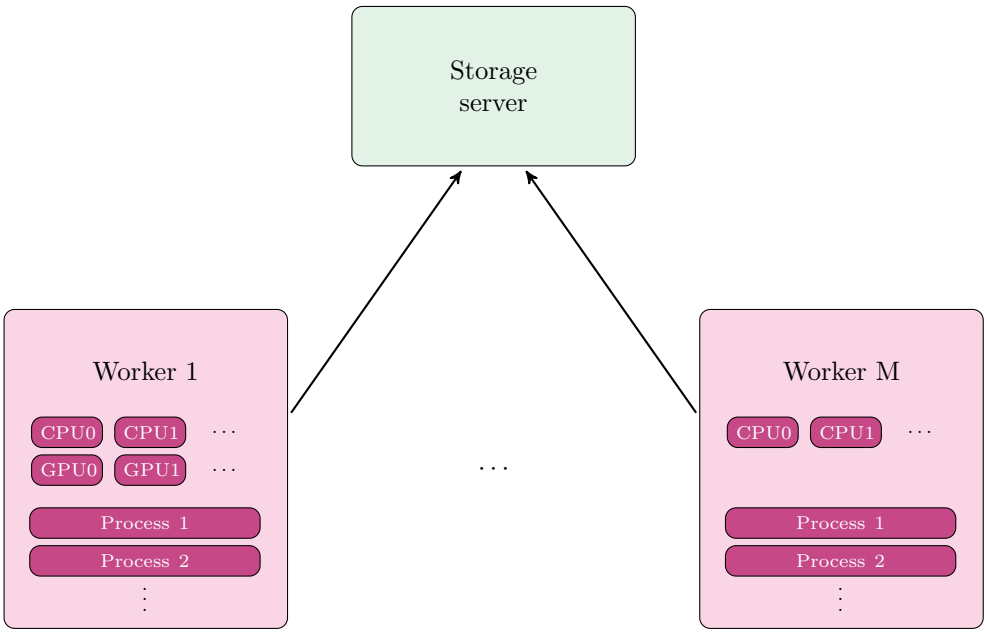


Figure 3.9: Simplified organization of workers. A worker is a computing node with processing units (CPUs and possibly GPUs), that typically runs training processes in parallel. They report their results to the storage server (Sec. 3.3.1.1).

As the figure shows, the workers are heterogeneous systems. They all have processing units, but some may only have central processors (CPUs), whereas others may also include Graphics processing units (GPUs). They must also have to have a connection to the storage server, which may be running under the same operating system, on the same hypervisor, on the same local network, or even on different networks.

The access between the workers and the storage server is mediated by a class defined in com.py, as seen in Sec. 3.3.1.1. This allows the standardization of the results so that different workers (with different hardware or software characteristics) report their results in a consistent manner. This is crucial to the clients, since this way they do not have to worry about the peculiarities that the workers may have. Moreover, by concentrating their results on a single, well-known machine, the workers simplify to great lengths the operation of the clients. However, this incurs performance costs mainly related to IO. Typically, the workers report training results every epoch or with some other periodicity (in the number of iterations). This causes training to be halted until the transference of the results is completed.

This penalty is minimized by running multiple training processes in parallel. This way, while a process is blocked waiting for its IO operation to conclude, others are still running, competing for CPU, and making use of the machine's processing resources. The downside to doing this is that more RAM is used than what would be strictly

required. This happens primarily since each process will have its own neural network model and its copy of the dataset in memory. The latter, avoiding the replication of the dataset, could be circumvented by using, for instance, shared memory. However, doing so was never really required — every machine/instance always had more than enough RAM available. The resources' bottleneck always lied with processing power and storage space, not lack of RAM.

Similarly to what happened with the storage server, the main worker units have also changed throughout time. Initially, we used two instances in Proxmox. These were not particularly performant, mainly since the Proxmox's bare-metal servers were outdated and not configured to pass through their CPUs nor their flags. This changed when our research group's cloud infrastructure moved to OpenStack. The two instances on Proxmox were discontinued, and two new ones were created on OpenStack, which was configured to pass through the hosts CPUs (exposing the extensions they support). This accounted for a significant boost in performance. More recently, the research group acquired a new computing node, which included a GPU installed. This machine was left as a bare-metal server used for particularly computation-intensive tests of the group — it was not added to OpenStack. This machine is currently used as the primary worker unit for the experiments carried, as well as the storage server. The two instances on OpenStack are used sparingly when a large number of training tests has to be run in a short amount of time (to parallelize the multiple runs).

As an aside, when the bare-metal server was bought, its GPU (GeForce RTX 2080) was very recent. It used the recent (at the time) CUDA 10.0, which TensorFlow did not support in its stable branch. Due to this, and to make the most out of the computational power of the machine, TensorFlow and most of its GPU-related dependencies were compiled specifically targeting the architecture of the machine and its CUDA Compute Capabilities.

3.3.1.3 Clients

The clients are machines that connect to the storage server with the purpose of processing data. They are highly heterogeneous, but this is usually a non-issue since they access and process the data independently. The most commonly used clients are laptops where results are analyzed. However, in some situations, more capable machines (typically used as workers) have been used to carry some more complex processing.

Most programs run by clients are written in Python and make extensive use of the "canonical" scientific libraries NumPy[15] and SciPy[16]. The rest are written in C and use

[15]https://numpy.org/

[16]https://www.scipy.org/scipylib/index.html

libraries such as GNU GSL[17]. For data visualization purposes, the main libraries used were Matplotlib[18] and, sporadically, Seaborn[19] (which is to some extent a front-end to Matplotlib). This setup proved robust and flexible enough for all the experiments carried.

3.3.2 The frameworks used and the choice of Python

In this work, Python[20] was chosen as the primary programming language for creating our neural network models and training them, since it is the *de facto* language for research in artificial neural networks and deep learning. This happens mainly due to its outstanding library support in many respects, from general-purpose scientific libraries such as NumPy and SciPy, to the libraries targeting deep learning, for instance, TensorFlow[21], Keras[22], and PyTorch[23]. To further see this, consider Fig. 3.10, which plots the ranking of various deep learning frameworks published at Towards Data Science[24] based on a number of factors, including online job listings, Google search volume, ArXiv articles, GitHub activity, among others. Aside from being open source, all the libraries (apart from one) work with Python[25]. Even though some of them have bindings for other languages too, the supremacy of Python in this regard makes it the language of choice.

Figure 3.10 also shows that the indisputable winner framework is TensorFlow, with Keras coming in second and PyTorch in third. These are the three main frameworks used in this work when training with neural networks. We use Keras with its TensorFlow backend as the primary framework with which to express and train the models. We resort to PyTorch when Keras is not flexible enough to carry some experiments as easily. Below we carry a more extensive discussion on the Keras & TensorFlow versus PyTorch topic.

[17] GNU Scientific Library, https://www.gnu.org/software/gsl/.
[18] https://matplotlib.org/
[19] https://seaborn.pydata.org/
[20] https://www.python.org/
[21] https://www.tensorflow.org/
[22] https://keras.io/
[23] https://pytorch.org/
[24] https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a
[25] The exception is Deep Learning for Java (DL4J), which, unsurprisingly, works with Java.

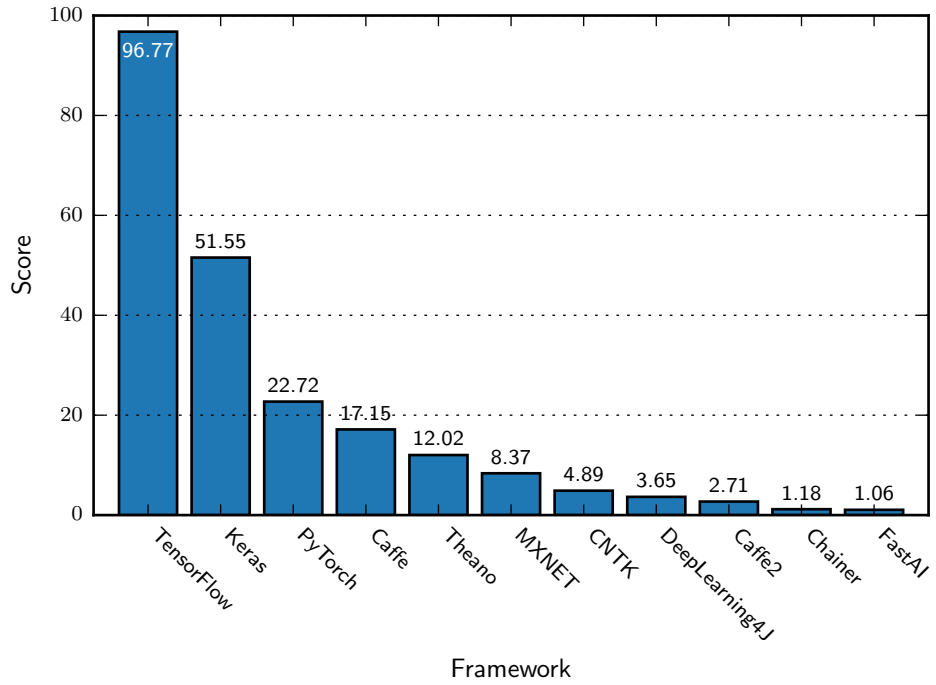


Figure 3.10: Deep learning frameworks ranking, as computed at https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a.

3.3.3 TensorFlow vs PyTorch

The main framework used throughout this work to implement and train neural networks was TensorFlow (Martín Abadi et al., 2015), through its Keras (Chollet et al., 2015) API. However, PyTorch (Paszke et al., 2017) was also sporadically used when carrying several tests, due to it allowing a more straightforward implementation of several operations. This section discusses a few of the differences between TensorFlow and PyTorch, in the hope of enlightening readers about when using one may be more beneficial than using the other.

In a way, both PyTorch and TensorFlow are similar, particularly in that in both frameworks, the user defines a neural network model as a computational graph that represent the computations carried in the network, and uses automatic differentiation techniques (provided by the framework) to compute and apply the gradients according to some optimization strategy. The main difference between the two lies in the kind of graphs they build. TensorFlow uses static computational graphs, whereas PyTorch uses dynamic ones. Before continuing, a brief description of what a computational graph is and what it does is due.

3.3.3.1 Backpropagation, Computation graphs and Automatic differentiation algorithms

In order to be able to discuss the main differences between TensorFlow and PyTorch, in this section we provide a very brief review on backpropagation, computation graphs, and automatic differentiation. The latter two are core components for carrying the backpropagation algorithm efficiently. Baydin, Pearlmutter, Radul, and Siskind (2018) provides a thorough review of these topics targeted towards machine learning.

To grasp the idea behind backpropagation, consider the derivatives of the loss function of a multilayer perceptron (Sec. 2.3.3) with respect to its weights. Let J denote the loss function being minimized; w_{ij}^k the weight connecting node i of layer $k-1$ to node j of layer k; z_j^k the total input that node j of layer k receives prior to activation, i.e., $z_j^k = \sum_i a_i^{k-1} w_{ij}^k + b_j^k$; a_j^k the activation of node j of layer k, i.e., $a_j^k = f\left(z_j^k\right)$ for an activation function f (a_0 denotes the input, $a_0 \equiv x$); and, finally, b_j^k the bias of node j of layer k. Then, by using the chain rule of calculus, for a link in the output layer of a network with l layers of weights ($l+1$ of nodes) we have

$$\begin{aligned}\frac{\partial J}{\partial w_{ij}^l} &= \frac{\partial z_j^l}{\partial w_{ij}^l}\frac{\partial a_j^l}{\partial z_j^l}\frac{\partial J}{\partial a_j^l} \\ &= a_i^{l-1}\frac{\partial a_j^l}{\partial z_j^l}\frac{\partial J}{\partial a_j^l}.\end{aligned}$$

Meanwhile, for links in the hidden layers we have

$$\begin{aligned}\frac{\partial J}{\partial w_{ij}^h} &= \frac{\partial z_j^h}{\partial w_{ij}^h}\frac{\partial a_j^h}{\partial z_j^h}\frac{\partial J}{\partial a_j^h} \\ &= \frac{\partial z_j^h}{\partial w_{ij}^h}\frac{\partial a_j^h}{\partial z_j^h}\sum_k \frac{\partial z_k^{h+1}}{\partial a_j^h}\frac{\partial a_k^{h+1}}{\partial z_k^{h+1}}\frac{\partial J}{\partial a_k^{h+1}} \\ &= a_i^{h-1}\frac{\partial a_j^h}{\partial z_j^h}\sum_k w_{jk}^{h+1}\frac{\partial a_k^{h+1}}{\partial z_k^{h+1}}\frac{\partial J}{\partial a_k^{h+1}}.\end{aligned}$$

These expressions show that the derivatives of the loss with respect to each weight can be computed based on partial results computed during the forward pass of an input through the network (e.g., the values z_i^k or a_i^k) and, perhaps more importantly, based on results obtained during the backward pass itself. For instance, notice how determining $\frac{\partial J}{\partial a_j^h}$, which is necessary for evaluating $\frac{\partial J}{\partial w_{ij}^h}$, is achieved using the derivatives of the loss with respect to the activation of the succeeding layer, i.e., $\frac{\partial J}{\partial a_k^{h+1}}$. This is the core of backpropagation. However, to carry these computations efficiently, new structures and methods had to be developed, namely, computational graphs and autodiff techniques.

There does not seem to be a single, reference description of computational graphs (Goodfellow et al., 2016, Sec. 6.5.1). Here we present the definition given by DyNet[26] (Neubig et al., 2017), that describe a computational graph as "a symbolic representation of a complex computation that can both be executed and differentiated using autodiff algorithms".

Generally, a computational graph is (usually) a directed and acyclic graph where edges represent data dependencies, and nodes are variables that may be of scalar, matrix, or tensor type (among others). A node with an incoming edge is a function of that edge's tail node. The nodes represent the successive chaining of functions of the initial input nodes. These functions typically include the binary arithmetic operations, the unary sign switch, and elementary functions such as the exponential, the logarithm, and the trigonometric functions. The main feature of these graphs is that the nodes, aside from storing the actual value of their computation, also record its derivatives with respect to their inputs, which allows the gradient of the function the graph implements to be computed by the chain rule by working backward, since

$$(f \circ g)'(x) = f'(g(x))g'(x).$$

This feature is, in fact, the core idea of automatic differentiation (also called autodiff, or, sometimes, autograd), where, in essence, the computation of a particular function that results from the composition of other "elementary" functions is augmented with the calculation of various derivatives. Then, combining the derivatives of the constituent operations following the chain rule, one obtains the derivative of the overall composition.

Note that automatic differentiation is not the same as other forms of computing derivatives "automatically" (i.e., by means of a computer). Other methods include (i) manually working out derivatives and coding them; (ii) numerical differentiation using finite difference approximations; and (iii) symbolic differentiation using expression manipulation (typically in computer algebra systems) (Baydin et al., 2018). However, all these methods pose problems related with their scalability or efficiency.

Computational graphs and automatic differentiation are the foundational components used to implement the backpropagation algorithm, even though they are frequently overlooked.

3.3.3.2 TensorFlow's static and PyTorch's dynamic graphs

We now continue the discussion about TensorFlow's static and PyTorch's dynamic computational graphs, and the implications of the difference. The difference between

[26] https://github.com/clab/dynet, a neural network library developed primarily by Carnegie Mellon University.

the two essentially means that, in TensorFlow, a computation graph is defined once and executed over and over again, using a mechanism typically called placeholders to provide different input data to the graph (which is the only thing changing across different iterations). Meanwhile, in PyTorch, a new computation graph is defined every forward pass.

Each methodology has its own merits. On the one hand, static graphs allow for lots of optimization since, due to their static nature, the framework has the freedom to, for instance, merge graph operations or decide upon strategies to distribute the graph across different machines/computing nodes. Most of these optimizations are only reasonable in a context where the graph is not expected to change frequently, since otherwise, the (expensive) optimization procedures have to be repeatedly run, consuming a significant amount of time. This effect is quite similar to the idea of compiling a program with the optimization flags active — usually, the program will take significantly longer to compile with the flags. However, the added cost is typically amortized over the course of many executions of the program (which hopefully will run a lot faster). On the other hand, this potential efficiency comes at a cost for the framework's user. In essence, most of the operations that are to be carried have to be present in the graph during all time, which makes the resulting programs more rigid/static. This inflexibility becomes troublesome when one wants, for instance, to carry different operations of the graph inside a loop, depending on the input data or some other factor. In contrast, with a dynamic graph (which is built every iteration), one can come up with a much more natural (and clean) solution.

Moreover, one has to be careful when trying to achieve "dynamic behavior" with TensorFlow's static graphs. As a very crude example, on one occasion during this work, we experimented with a variety of weight update strategies. To achieve the behavior we desired, in some situations, we kept adding nodes to TensorFlow's graphs based on the operations we wanted to happen. While this achieves the desired goals, there are two blatant issues with this method. First and foremost, TensorFlow's graphs are append-only, which means that one cannot remove operations after adding them without going through great lengths. Moreover, despite unused nodes not being processed (not wasting CPU time), they do take memory, and after adding thousands of nodes, it adds up. Another major problem is that, since there are constantly more and more operations in the graph, and the graph is being recompiled at each iteration, the additional nodes cause the compilations to take increasingly longer time to terminate.

There are ways to circumvent this, for instance, using a base graph and a per-iteration copy graph to be used and dispensed. However, at some point, one realizes that for situations like this, it is just far simpler to use a more flexible solution such as the kind PyTorch provides. With the latter framework, details such as these are much more

hidden away from the user (the programmer). The graphs are implicitly built on a per-iteration basis, which is significantly more comfortable to work with. Finally, notice that even though computationally-wise the approach using dynamic graphs may seem (and is) less efficient than the static graph approach, one should not decide which to use based solely on execution time. In practice, for this kind of problems where one does not seek "a commercial, heavily optimized product", one must also consider the time it takes to implement the desired solution. In many cases, it may turn out that it is much more reasonable to write a more transparent and natural solution than the fastest one. In research, we feel it is paramount to minimize the time from idea to test, and, ultimately, to the analysis of the test's results.

3.3.4 Computing nodes

The specifications of the computing nodes are presented in Tab. 3.1. The machine most recently used as both storage server and main worker node is the one named `jake`. It runs bare-metal and includes an AMD Ryzen™ Threadripper™ 2920X at 3.5 GHz and a Nvidia GeForce RTX 2080.

Hostname (hypervisor)	Function	CPU	GPU	CPUs	RAM
`hometree` (OpenStack)	Storage	Intel® Xeon® Processor E5-2620 v4	—	8	16 GB
`neytiri-1` (OpenStack)	Worker	Intel® Xeon® Processor E5-2620 v4	—	32	32 GB
`neytiri-2` (OpenStack)	Worker	Intel® Xeon® Processor E5-2620 v4	—	32	32 GB
`jake` (bare-metal)	Storage and Worker	AMD Ryzen™ Threadripper™ 2920X	Nvidia GeForce RTX 2080 (8 GB RAM)	24	32 GB

Table 3.1: Physical specifications of the computing nodes in use.

Meanwhile, the versions of the software being used are expressed in Tab. 3.2. Versions of other libraries used (e.g., NumPy or SciPy) are not shown since they have much more stable APIs and are typically pulled automatically when installing other packages. All computing nodes run Ubuntu 18.04 LTS. They also use Python 3.6, which is the most recent version on Ubuntu's repositories at the time of writing.

Hostname	Operating system	Postgre-SQL	CUDA Toolkit	Python	Keras	Tensor-Flow	PyTorch
`hometree`	Ubuntu 18.04 LTS	10.8	—	—	—	—	—
`neytiri-1`	Ubuntu 18.04 LTS	—	—	3.6	2.2	1.11	1.1
`neytiri-2`	Ubuntu 18.04 LTS	—	—	3.6	2.2	1.11	1.1
`jake`	Ubuntu 18.04 LTS	11.3	10.0	3.6	2.2	1.12	1.1

Table 3.2: Versions of the main software used in the computing nodes.

Note, however, that the programs developed should run on more recent versions of the software used too, since, in general, only conventional methods were used.

4 Probing the learning process

The beginning of knowledge is the discovery of something we do not understand.

— Frank Herbert

probing, verb: to make a searching exploratory investigation

This chapter intends to unveil some of the processes happening during the training of a neural network. It describes an exploration stage of the work we performed, where our primary goal was not to study a particular behavior known/chosen *a priori*, but to obtain some intuition about what is going on. The synthetic dataset was used to this end, providing a straightforward yet nontrivial problem that can be understood and reasoned about easily. Moreover, knowing how a neural network can implement it perfectly allows us to establish a base/reference architecture that is known to have the capacity to represent the problem successfully. As a result, in case of unsuccessful training, one can dismiss the possibility of having a network that is just too small or "unfit" for the problem.

The chapter is organized as follows. We start by presenting, in Sec. 4.1, the reference network architecture used and parameters with which most of the experiments of this chapter were carried. In Sec. 4.2 we present the training results obtained with these settings. In Secs. 4.3 and 4.4 we vary the output activation function and interpret the changes that these modification causes in the organization of the network, respectively. Then, in Sec. 4.5, we carry a few tests where we vary learning parameters such as the learning rate and batch size used, and observe that a network's initial configuration of weights determines, to a great extent, the shape of its loss function. Finally, in Sec. 4.6, we briefly evaluate the similarity between learning trajectories.

4.1 Base settings of the training experiments

Given the general complexity of neural networks as well as the sheer amount of (hyper)parameters involved in their specification and training, it is impossible to test all the possible combinations of parameters. Even if it was feasible, doing so would not be advisable, since it would not provide easily human-understandable value. One should try to simplify and isolate all conditions as much as possible, in order to make the most sense out of the observations made. To this end, this section presents the reference settings under which the tests presented in this chapter were performed.

4.1.1 The network

The base architecture used throughout this chapter is a multilayer perceptron with a single hidden layer of 100 nodes. Due to the choice of the dataset used for training (discussed in Sec 4.1.2) both the input and output layers have size 30. Having the ideal implementation that was presented in Sec. 3.1.1 in mind, the hidden layer uses the ReLU activation function. Figure 4.1 shows a diagram of the resulting architecture.

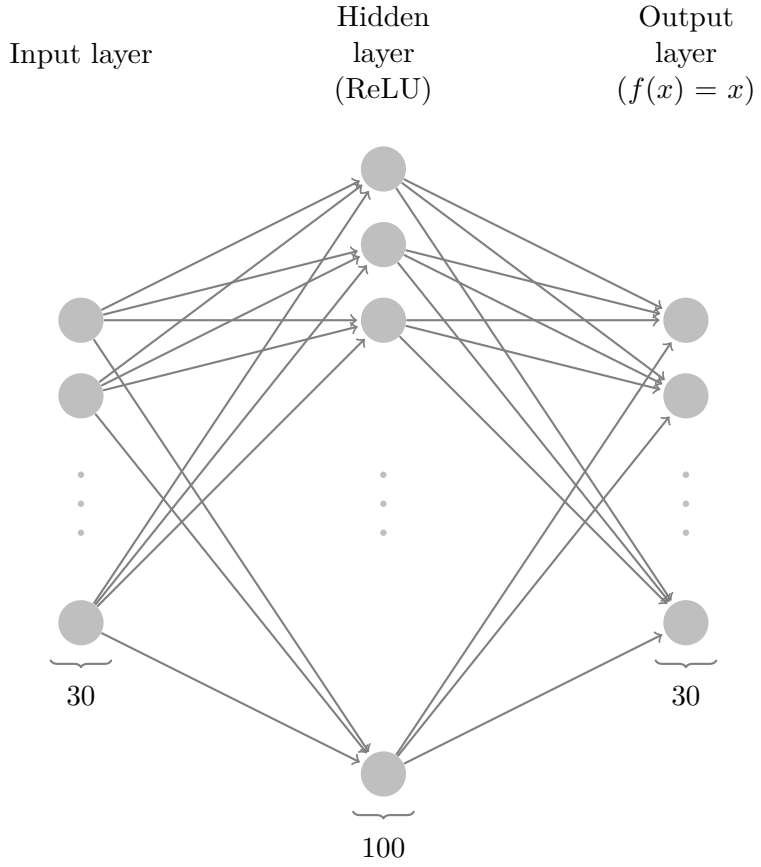


Figure 4.1: Base architecture of the networks trained (bias units have been omitted).

Notice that even though the ideal implementation strictly requires only 30 nodes in

the hidden layer, the reference architecture has around three times more nodes. This over-sizing was done to allow for other organizations to emerge and to ease training. This approach goes in agreement with the general notion of training neural networks that are far larger (i.e., having much higher representational capacity) than what a given problem demands (Zhang, Bengio, Hardt, Recht, & Vinyals, 2017).

4.1.2 Dataset and training parameters

We use the synthetic dataset described in Sec. 3.1.3 due to the reasons explained throughout Sec. 3.1. We recall that the size of the input and output samples of the dataset we created is 30 bits, which is why the reference architecture of Sec. 4.1.1 uses 30 input and output nodes. The remaining training parameters (hyperparameters) used are: (i) learning rate (lr) of 0.01; (ii) batch size (bs) of 128 samples; (iii) Mean squared error (MSE) loss function; and (iv) SGD as optimizer. These are typical values/choices, usually the defaults of the frameworks used. In general, each test in the sections of this chapter vary either one of these parameters, or elements of the architecture. Moreover, unless otherwise stated, all loss function curves shown are for the test set (i.e., they represent the test loss).

In general, this chapter avoids summaries of results. They represent an aggregated view of many networks (i.e., test runs resulting from different initial conditions), and the condensation of information may hide relevant traits in the behavior of some networks, which may be important to gain an intuition about features of the processes unfolding during their training. Notwithstanding this, they are still used whenever appropriate.

4.2 Training results obtained with the reference parameters

Figure 4.2 shows the test loss function resulting from the training of 64 different training runs with the base settings presented in Sec. 4.1. The training lasted for 10^7 iterations (an iteration is a step/update of the learning procedure), so that the test loss of all networks fell close to the machine's epsilon (around 10^{-7} for the 32-bit float type used).

The fact that all networks reached this level (slightly above 10^{-7}) in the test set shows that the dataset has an appropriate size to avoid overfitting (the training set is sufficiently representative to eliminate the case of overfitting). Moreover, the architecture is large enough to ease training and avoid that the networks get stacked at a poor minimum. Notwithstanding this, as is visible in the figure, some networks spend a significant amount of time at what seems to be a plateau/set of plateaus before eventually finding their way to a "good" minimum. Figure 4.3 summarizes the

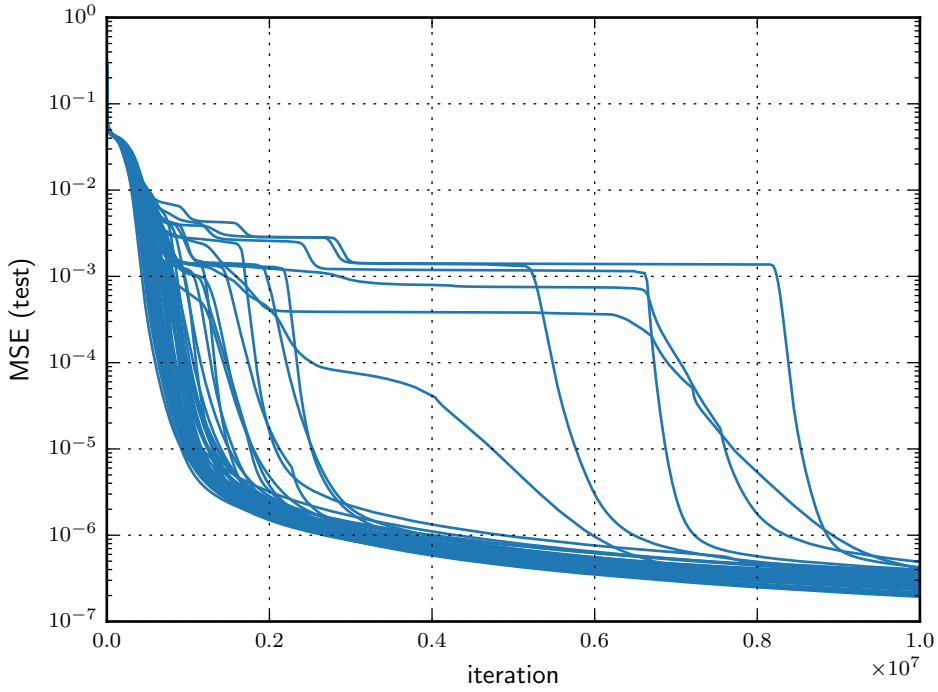


Figure 4.2: Test loss resulting from different test runs using the reference parameters presented in Sec. 4.1. 64 independent runs are plotted.

percentage of the networks previously shown in Fig. 4.2 that have reached a particular loss value by a given training iteration.

It illustrates that, as expected, it takes increasingly more time to reach lower loss values. Furthermore, it shows there is a large gap between reaching a loss below 10^{-5} and reaching a loss below 10^{-6}. It also shows that, initially, the loss measured by the networks decreases substantially, but then progress is slowed down considerably. This is a known behavior exhibited by stochastic gradient descent methods (Bottou, Curtis, & Nocedal, 2018, Sec. 3.3).

More surprisingly, if the outputs of the network are binarized (set to `0` or `1` if smaller or larger than 0.5, respectively), something more interesting is observable. Notice that this is just a different way of interpreting the networks' outputs. Training itself is not changed at all. In fact, this experiment was conducted with the values already gathered from training the previous networks. Figure 4.4 shows the results.

By binarizing the outputs — which essentially turns the MSE into the "bit error rate" of the network — a peculiar trend in the learning process is revealed. As happened in the previous case (when using a network's outputs "unchanged"), there is a gap in the time it takes to improve from below 10^{-2} to below 10^{-3}. However, suddenly, improving from below 10^{-3} to below 10^{-6} happens over the course of a very small number of iterations.

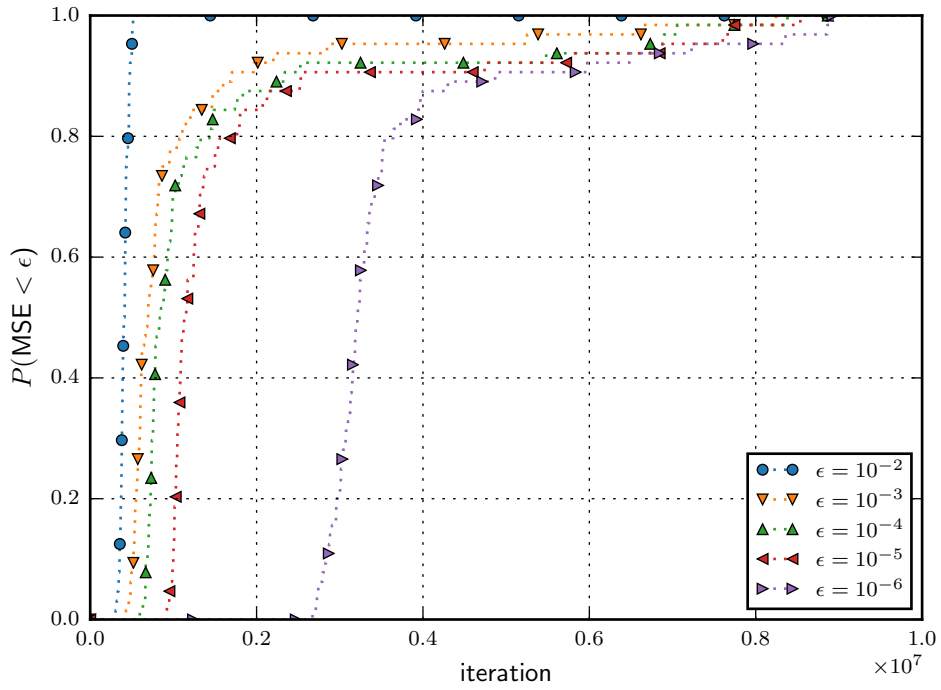


Figure 4.3: Distribution of the networks of Fig. 4.2 that have reached a certain test loss value ϵ by a given iteration.

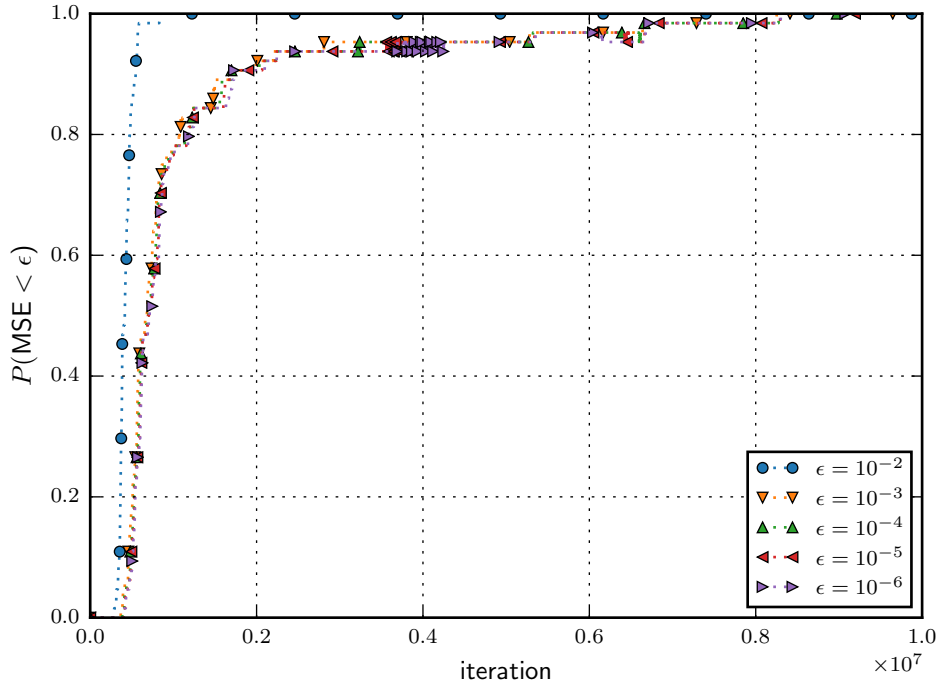


Figure 4.4: Distribution of the networks of Fig. 4.2 that have reached a certain test loss value ϵ by a given iteration, after their output is binarized to 0 and 1.

This behavior suggests that a network's weights experience a "phase transition" where they evolve from being poorly coordinated to being highly organized. In other

words, it appears that during a significant amount of training time, a network keeps providing inconsistent outputs. However, all of a sudden, its weights find concordance, which allows it to give proper outputs consistently. The major surprise arises due to this transition happening so fast.

4.3 Varying the output activation function

The reference curves presented in Fig. 4.2 showed that the evolution of the loss function varies widely among different training instances. A possible explanation for this contrasting behavior may lie in the activation function employed in the output layer. While the ideal implementation does use the identity activation ($f(x) = x$) in the output layer, this function may be too "unrestricted". By not having their outputs limited to reasonable values, the networks that use this function may be more susceptible to poor initializations that, for instance, cause few output nodes to start with substantially large values.

In this section, we consider the usage of an alternative, commonly used activation function in the output layer, the logistic sigmoid function, $\sigma(x) = 1/(1+e^{-x})$. Its output is limited to the range $]0, 1[$, which means that it cannot implement the filter perfectly (but it should be able to do so to any desirable accuracy, at least as far as the machine's precision allows it to). The ideal implementation will not apply to this activation function, as discussed later in Sec. 4.4.2. Notwithstanding these concerns, it may still help training by limiting the networks' outputs to a reasonable interval. Figure 4.5 plots the test loss of 64 sigmoid output networks trained alongside the loss curves of the linear output networks that were presented previously (to allow for easier comparison).

The difference shown by the loss function of the two architectures is evident. On the one hand, the sigmoid output networks are much more "well-behaved" than the reference networks (which use linear output activation). On the other hand, they train to worse loss values than the linear output networks do. This dichotomous behavior happens most likely due to two main reasons. The first one relates to our initial argument about the range of the sigmoid activation function. By limiting the range of a network's outputs between zero and one, the per output node error of the sigmoid networks will never grow too large, which causes the loss function to not vary too much. The second relates to the derivative of the sigmoid function. Since $\sigma'(x) \in]0, 1/4]$ — in opposition to the derivative of the identity function, which is one everywhere — the updates that the sigmoid networks undergo are in general smaller than those of the networks with linear output activation.

Despite the apparent slower learning showed by the sigmoid networks, it is once

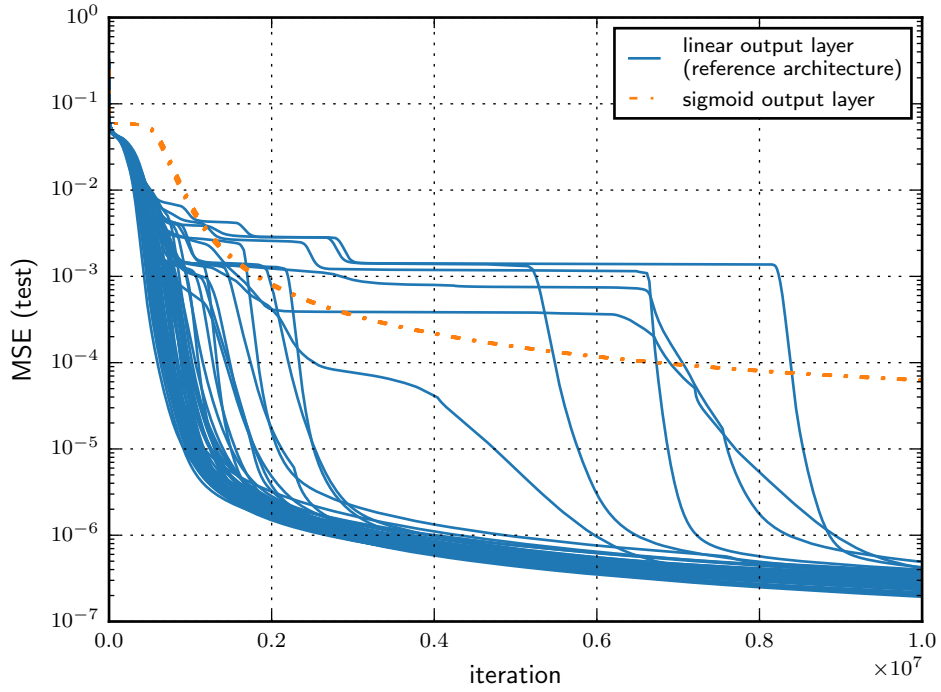


Figure 4.5: Training of different networks employing the sigmoid activation function in the output layer, alongside the training of the networks of Fig. 4.2. 64 test runs for each output activation function are plotted.

again interesting to consider the "bit error rate" of these networks, and compare it against that of the networks having an identity output layer. Figure 4.6 shows these results.

The binarization confirms that the sigmoid networks are training successfully. Nonetheless, unlike the networks with linear output, they do not seem to show the phase transition that was visible in Fig. 4.4, at least as evidently/markedly as the linear output networks did. This transition, for the networks with linear output activation, is observable in Fig. 4.6 by the sharp fall that the bit error rate exhibits (almost like raindrops). These sharp decreases are not observable in the sigmoid output networks, as Fig. 4.7 further confirms.

Finally, another unusual feature of the plot of Fig. 4.6 is the horizontal lines visible near its bottom (for instance, between a MSE of 0 and 10^{-6}). They arise due to the bit error rate taking only discrete values (in other words, it is only possible to incorrectly label a finite number of bits, which means that the overall error may also only take a discrete set of values). This characteristic becomes especially evident at such a small scale, like the one the figure includes between 0 and 10^{-6}. By computing by hand the bit error rate for a small number of errors (e.g., 0, 1, 2, etc.) and considering that there are 100 000 samples in the test set, each having 30 bits, one obtains the exact level of the bottom horizontal lines. Table 4.1 provides these values. Notice that the bit error

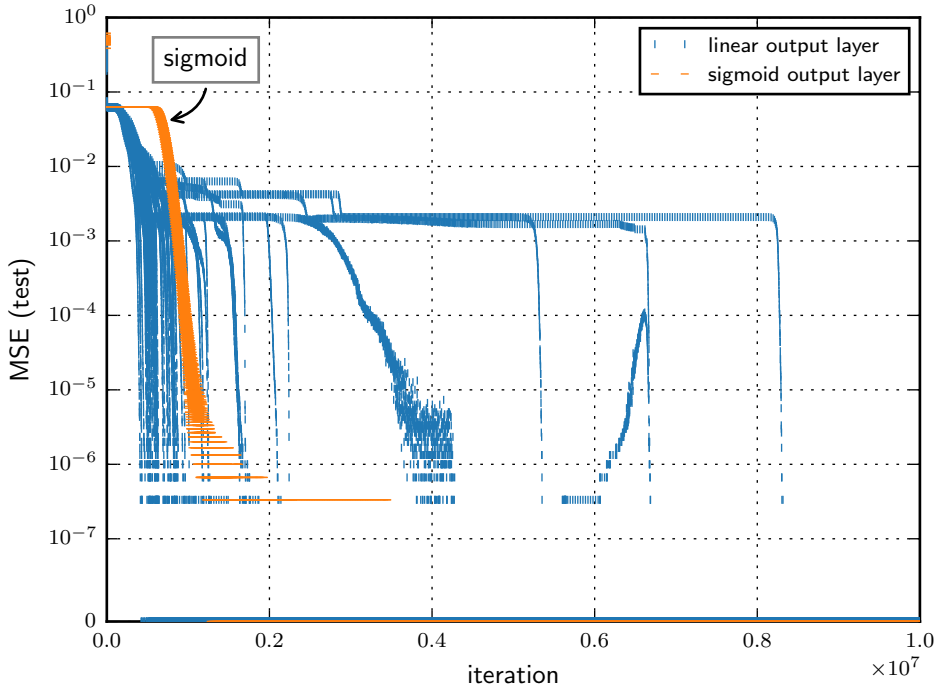


Figure 4.6: Result of binarizing the outputs of the networks of Fig. 4.5.

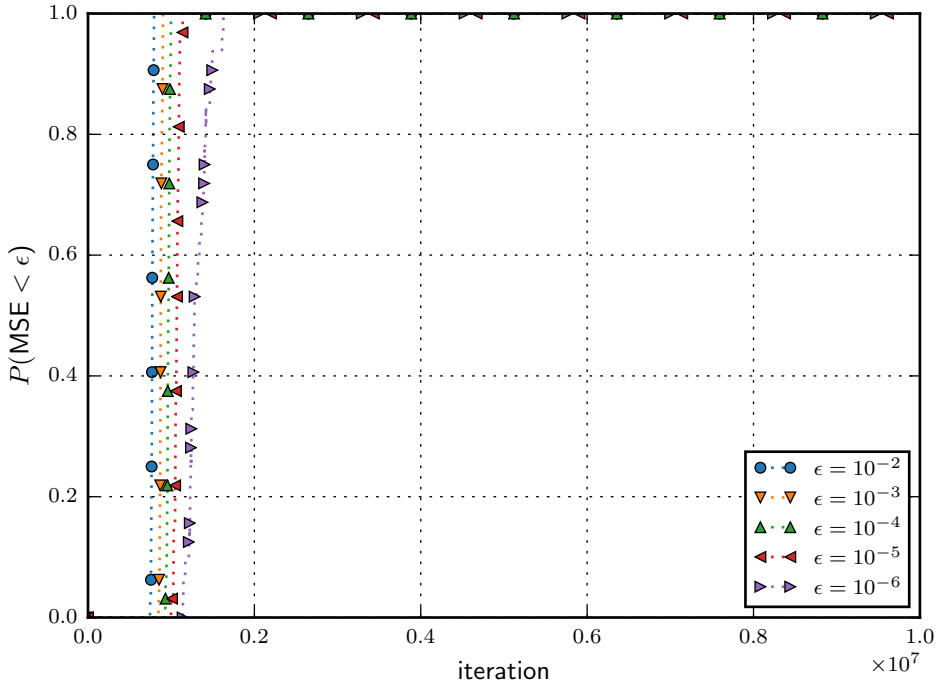


Figure 4.7: Distribution of the networks of Fig. 4.5 with sigmoid output layer that have reached a certain test loss value ϵ by a given iteration, after their output is binarized to 0 and 1. Notice that the transition across loss levels is much less sharp than the transition observed in Fig. 4.4 for networks with linear output layer.

rate calculated matches the height of the four bottom horizontal lines perfectly.

# errors	MSE ("bit error rate")
0	0
1	3.33×10^{-7}
2	6.67×10^{-7}
3	1.00×10^{-6}

Table 4.1: Bit error rate for a small number of errors. The BER computed matches the four bottom horizontal lines of Fig. 4.6.

4.4 Inspecting a network's configuration of weights

The previous section presented two different network architectures, both showing signs of successful training. The networks having linear activation function in the output layer seem to be less "well-behaved" than those having sigmoid, but the loss function of the former reaches lower levels than the latter. Despite this contrasting behavior, both kinds of networks show approximately the same discrimination ability, as shown by the binarization/rounding procedure. One interesting question that arises at this point lies with understanding how these two kinds of networks are implementing their solution to the filter problem (in which the dataset is based), or whether it is possible to do so altogether. This is the primary goal of this section. As we will see below, in general we cannot say precisely the organization the networks should come to in neither case, mainly because there are many equivalent solutions. What we can do is try to understand the roles that their nodes are playing in the organization that they have acquired, and, given the architecture of the networks, consider how reasonable they are.

We are already familiar with the ideal implementation proposed in Sec. 3.1.1, which the architecture with linear activation in the output is capable of implementing. Despite this, note that the implementation suggested is not the only perfect solution possible given this architecture. As an example, consider a network that uses two hidden nodes (instead of one) for each group of four consecutive input bits. Then, the respective output node connects to the two hidden units with a weight of 0.5 for each. Essentially, in this case, two hidden nodes are recognizing the same four-bit pattern, and the respective output node aggregates these hidden nodes' results. The same idea could be applied to 3, 4, 5, ..., hidden nodes.

Meanwhile, the networks using sigmoid in the output layer should lead to a fundamentally different implementation. In the ideal implementation, the hidden nodes are responsible for identifying the patterns. Due to this, the value leaving each hidden node

will be 1 if the four bits connecting to it do match the pattern, and 0 otherwise. If this value is used as input to a sigmoid, even if multiplied by the weight of the connection linking the hidden node and output node, the sigmoid will only output a value in $[0.5, 1[$ or $]0, 0.5]$, depending on the sign and magnitude of the weight (and we want 0 and 1 as the outputs of the networks).

4.4.1 Input/output strengths

In Sec. 3.1.2 we saw that in the synthetic dataset, one output bit is correlated with four input bits, which is evident since one output bit depends only on four specific, consecutive input bits. It is sensible to assume that trained networks show similar input/output dependence. Further, this should happen irrespectively of the output activation function used, i.e., both linear and sigmoid output networks should show some relation between specific four input nodes and one output node.

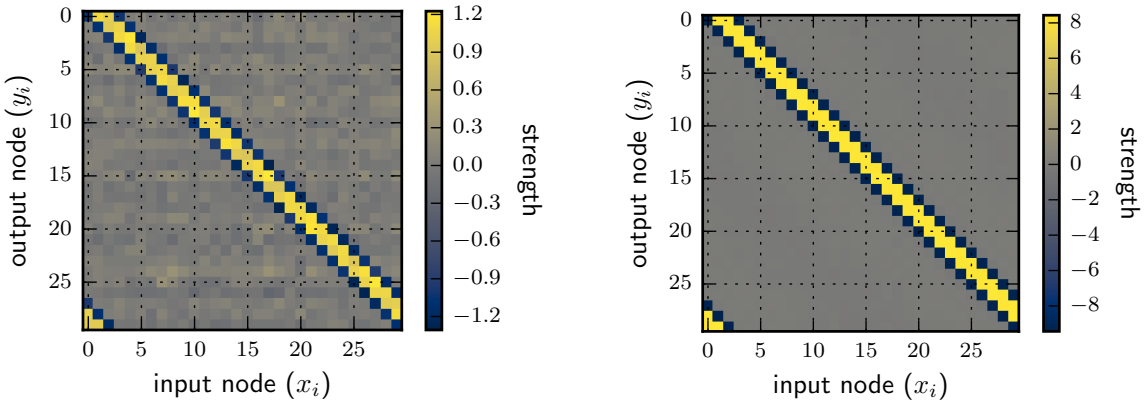
A possible way of verifying this presumption is achieved by considering the "aggregated strength" of the paths of a network connecting input to output nodes (through intermediary hidden nodes). In essence, the strength of these paths gives an idea about the overall sensibility an output has to an input node. For a linear neural network (i.e., a network without nonlinear activation functions) this is sometimes called the total end to end propagation of activity (Saxe, McClelland, & Ganguli, 2014). It is defined as the sum of the product of weights along all paths connecting an input to an output node. As we saw in Sec. 3.1.2, a possible way of computing it at once for all input/output pairs of nodes is achieved by successively multiplying the matrices of weights of each layer, i.e.[1]

$$w_{Tot} = \left(w^1 w^2 \cdots w^{N_l}\right)^\intercal.$$

Figure 4.8 shows the strengths measured for networks with linear and sigmoid output activations.

Scale aside, the figure shows remarkable resemblances with the correlation matrix of Fig. 3.2, presented in Sec. 3.1.2, and where the correlation between input and output variables had been computed from the dataset alone. These results suggest that the organization of a network (i.e., its weights) evolves to increase the aggregated strength of the most meaningful input/output connections. Moreover, two different architectures lead to similar strengths (once again, ignoring their scale, which must be related to the particular activation functions of the networks), which are themselves similar to the correlations computed directly from the dataset. This resemblance suggests a simple, yet interesting, connection between the dataset and the configuration of weights learned

[1] w^k is the weight matrix of layer k (where an entry w^k_{ij} denotes the weight connecting node i of layer $k-1$ to node j of layer k).



(a) Network with linear output activation.

(b) Network with sigmoid output activation.

Figure 4.8: Propagation of activity ("overall connection strength") matrices for networks employing linear and sigmoid functions in the output layer. A single network is shown for each architecture, but other test runs show the same results. Notice how, scale aside, these matrices are very similar to one another and to the correlation matrix presented in Fig. 3.2.

by a network. Despite the two architectures showing approximately the same behavior (except for the scale), can we infer more about their internal organization solely from looking at their configuration?

4.4.2 Layer-wise correlations of weights

Instead of considering the overall, end-to-end strength of the paths linking an input to an output node, one may try to compartmentalize and conceive a way of computing correlations[2] between weights of a particular layer of a network. Computing the correlation of a more restricted set of weights should provide a more detailed description of the organization of a network (enlightening us about the roles played by its nodes).

We know from the formulation of the dataset that groups of four consecutive input bits are correlated, but inputs that are further apart are not. To group nodes that should have similar roles in a network, we may exploit the previous observation and conceive a "distance" property between pairs of nodes of a network. Given the structure of the dataset, the distance may be written as the difference between the indices of two nodes in their layers. For two nodes belonging to the same layer or layers of the same size, this results in

$$\operatorname{dist}(n_i, n_j) = \begin{cases} \min(j - i,\ N + i - j) & \text{if } j \geq i \\ \operatorname{dist}(n_j, n_i) & \text{otherwise} \end{cases},$$

[2]Here "correlation" is used in a broad sense, such as a non-obvious, general association between two variables.

where N is the number of nodes of the layer/layers that the nodes n_i and n_j belong to.

The second branch of the expression serves to ensure that we carry the main computation, in the first branch, with $j \geq i$. The min function in the first branch is used to allow counting the nodes as if they "wrap-around" (e.g., we want the distance between the first and the last nodes to be equal to 1, instead of $n - 1$). This small tweak is important since the dataset itself is written as if the bits in the input samples wrap-around, and we want our distance function to simulate the same behavior.

Consider now the different ways of traveling through the layers of a three-layered network using a single intermediate layer. We can start at layer 0 (`L0`), go to layer 1 (`L1`), and back to `L0`, which we write as `L0-L1-L0`. In total, there are six ways of constructing such paths of length two, which we illustrate in Fig. 4.9 to the right, namely:

1. `L0-L1-L0`;
2. `L0-L1-L2`;
3. `L1-L0-L1`;
4. `L1-L2-L1`;
5. `L2-L1-L0`[3];
6. `L2-L1-L2`.

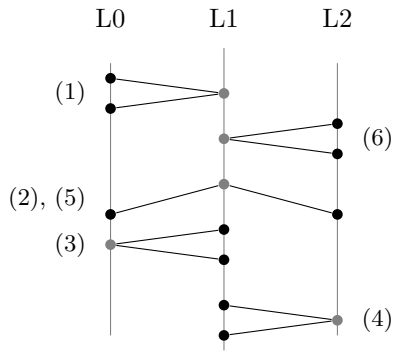


Figure 4.9: Paths of length two through layers. The numbers refer to the enumeration to the left. Black dots represent endpoints of the path, while grey dots an intermediate hop.

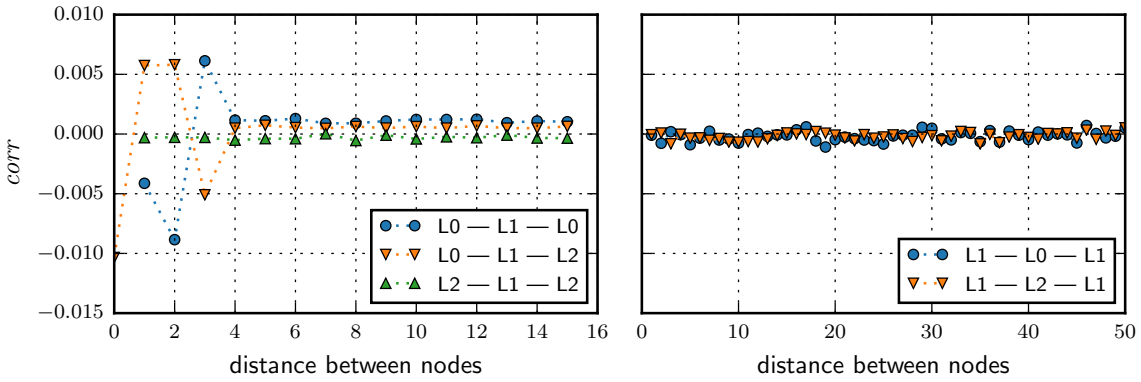
Finally, for each type of path we will compute the covariance of the weights in those paths that start and end in nodes at a given distance from each other. As an example, consider the path `L0-L1-L2` and its endpoint nodes at a distance of two. Let n_i^l denote the node i of layer l, w_{ij}^l the weight that links node i of layer $l-1$ to node j of layer l, and N^l the number of nodes of layer l. Then, the correlation of the nodes in this type of path (`L0-L1-L2`) at distance two is

$$\left\langle \frac{1}{N^1} \sum_k w_{ik}^1 w_{kj}^2 - \frac{1}{N^1 N^1} \sum_k w_{ik}^1 \sum_k w_{kj}^2 \right\rangle_{\left\{ i,j\colon dist(n_i^0, n_j^2) \,=\, 2 \right\}}.$$

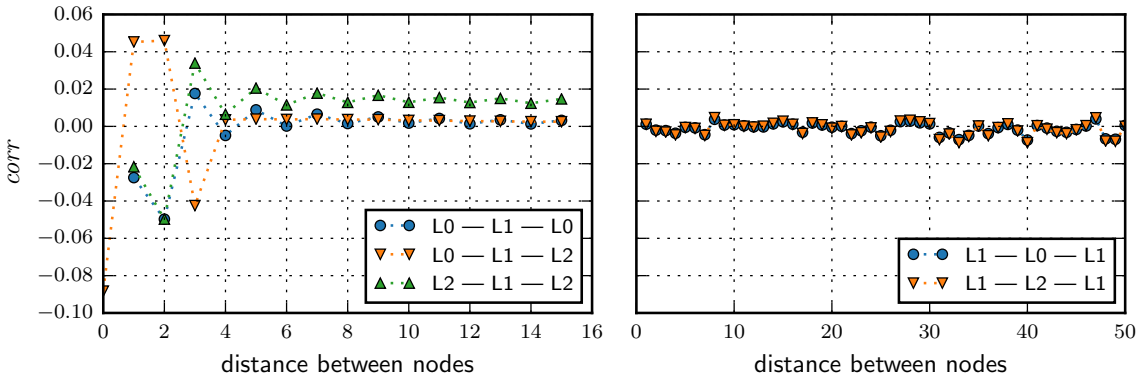
The general idea of the expression is to compute the correlation between nodes of two layers (the endpoints of each path), which are at a certain distance. The correlation is measured by the weights that connect the nodes through an intermediate layer (the middle layer in the path). Figure 4.10 plots these correlations for each possible distance and for each path.

For the networks with linear output activation, the results for the path `L0-L1-L0` suggest that these networks are following the ideal implementation. First, one can see

[3]As will be seen, this is the same as `L0-L1-L2`.



(a) Network using a linear function in the output layer.



(b) Network using a sigmoid function in the output layer.

Figure 4.10: Correlation of nodes at a certain distance for each possible path. The results are presented for a single network of each kind, but they are typical. Other test runs show the same behavior, but they were omitted for clarity.

that the nodes of the first layer are correlated up to a distance of 3, while for larger distances the correlation is close to 0. These results agree with the ideal implementation, since in it the input sequence `0110` is identified by an hidden node connected (solely) to four consecutive input nodes (which means that, as we are observing, weights participating in these funnels are correlated, whereas other weights are not). Moreover, the sign of the correlations in this path tells us even more about the organization of these networks. We can see that weights connecting nodes that are at a distance of one or two are negatively correlated, whereas weights connecting nodes at a distance of 3 are positively correlated. This agrees with the ideal implementation, since, in each funnel, two of the three possible pairs of weights at a distance of one have opposing signs, resulting in overall negative correlation. Similarly, for a distance of two, the two possible pairs of weights of a filter have opposing sings, resulting in negative correlation. Finally, the single pair of weights of a funnel at a distance of three have the same sign, which results in positive correlation.

In path L2-L1-L2 one can also see that nodes of the last layer (L2) are mostly uncorrelated. This behavior also agrees with the ideal implementation — in it, one hidden node does not influence more than one output node, no matter their distance. Meanwhile, the correlations of the nodes belonging to the path L0-L1-L2 show what we already saw in Fig. 4.8a — once again, up to a distance of 3, the correlation of the nodes varies (with its sign matching that shown in Fig. 4.8a), but it stays close to 0 for larger distances. The plots for the other two paths, L1-L0-L1 and L1-L2-L1 show correlations very close to 0 for all distances, but this is expected since the hidden nodes do not have any meaningful spatial arrangement (they could be permuted without this affecting the network), which makes the distance property we used unsuited to group them. Overall, these results suggest that the organization acquired by the networks with linear output activation is similar to that of the ideal implementation.

Finally, the networks with sigmoid output activation (Fig. 4.10b) show apparent similarities with the linear output activation networks if one ignores the difference of scale (which is related to the particular activation function used). The major exception is found for the path L2-L1-L2, which agrees with our previous observation that, in the sigmoid output networks, the output layer plays a different role than in the linear output networks. In the sigmoid output networks, the last layer plays an active role identifying the 0110 pattern, whereas in the ideal implementation (and the linear output networks) the hidden layer is the sole responsible for doing this (and the output layer is a bypass). Considering these results, it seems that, in the sigmoid networks, nodes in the hidden layer do combine the information of few input nodes (as the correlations for the path L0-L1-L0 show), but their values are themselves combined in the output layer, to fully identify the relevant pattern. Due to this, close nodes (i.e., with a distance less than four) in the output layer may share the partial computations that some hidden nodes made.

4.5 Trajectories followed during training

We saw in previous sections what typical loss function curves look like (Secs. 4.2 and 4.3) and verified to some degree that these networks acquire organizations that we can mostly understand (Sec. 4.4). Meanwhile, we also saw that the loss function of the networks with linear output activation shows much more erratic behavior than that of the sigmoid output networks. We considered some reasons why this may happen, for instance, the difference in scale of the derivatives of the identity and sigmoid functions. It seems worthwhile to try to further pinpoint what affects the variability of the linear output activation networks.

A first question one may pose is whether initializing a network with the configuration of weights that another network has at some point during its training causes the learning trajectories of the two networks to be different (i.e., to "diverge"). Figure 4.11 shows the results of this experiment.

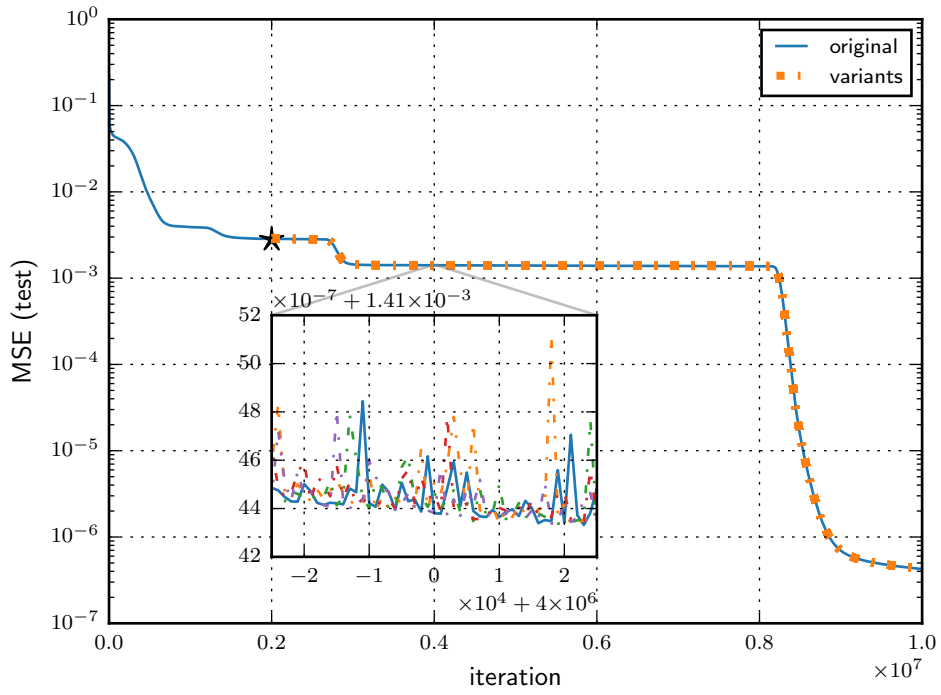


Figure 4.11: Training of 64 different networks, employing linear activation in the output layer, and using as initialization the weights that a reference network (plotted in solid blue line) had at the iteration marked by the symbol ⋆. The inset plots a zoom of the main plot, but it only shows 4 variants (of the 64 possible) for clarity.

Interestingly, the learning trajectories seem more stable than what one could initially expect. The figure shows 64 different networks initialized with the configuration a reference network (plotted in solid blue line) had at iteration 2×10^6 (marked by the star ⋆). Surprisingly, the loss curve of these networks overlaps perfectly (at the macroscopic level of the plot) with the curve of the reference network. This behavior suggests that, by the time the configuration of weights of the reference network is collected, the minimum to which it will converge (or, at least, its quality) is already determined, alongside the trajectory it follows towards it.

A big part of this apparent stability may lie in the dataset itself, which is possibly simple enough to allow that after some initial training, the learning trajectories get, to a large extent, fixed. On the other hand, the stochasticity of the learning algorithm should lead the network through different trajectories in the weight space (the inset in Fig. 4.11 confirms that, at a small scale, this is in fact what happens). The strangest

part of this experiment is that the loss function of all the networks pass through the same plateaus, for the same amount of time, and fall at the same instants.

During the training of a neural network, two primary hyperparameters affect the stability and speed of training — the batch size and the learning rate. Can they be used to cause deviations in the learning trajectories of the networks of Fig. 4.11, more so than the microscopic variations seen in the zoomed region?

4.5.1 Varying the batch size

Figure 4.12 shows the test loss of different networks initialized with a reference initial configuration (the weights of a reference network, at the iteration marked by the star), and trained with varying batch sizes. Five different instances of training are plotted for each batch size.

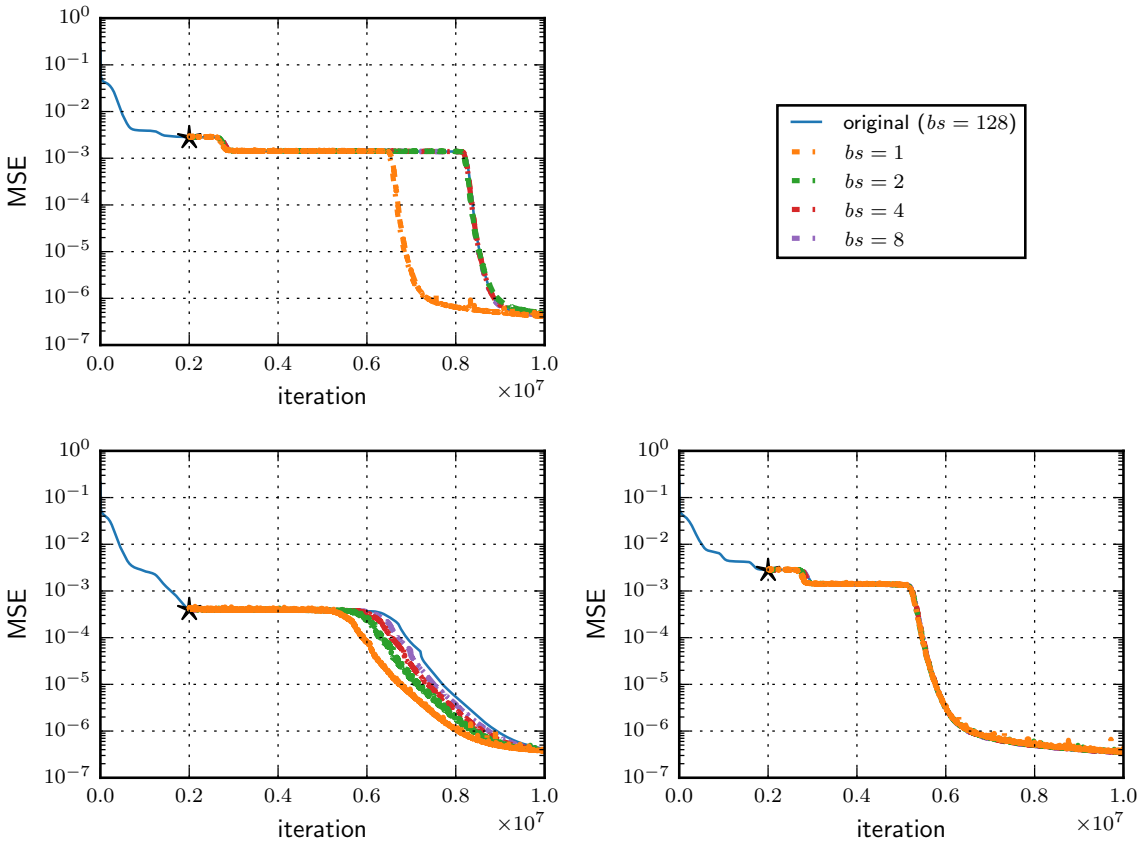


Figure 4.12: Training of different networks using different batch sizes. The networks were initialized with the weights the reference network (of their plot) had at the iteration marked by the star. Different plots concern different reference networks (the top left is the same as in Fig. 4.11). Five test runs are plotted for each batch size.

These results suggest that, in these networks, the batch size can only affect the speed with which the networks reach their minimum, but cannot change its quality. Moreover,

the paths taken by the networks to reach their minimum seem to be very similar, with the loss curve exhibiting consistently the same shape. It is also interesting to note that, as expected, smaller batch sizes cause small instabilities that are particularly visible at the bottom of the plots by the "noise" visible in some of the curves.

In Sec. 4.6 we will return to these results, to further analyze whether these networks are following different trajectories (in their space of parameters) that happen to have the same loss, or if their trajectories are actually the same.

4.5.2 Varying the learning rate

In Sec. 4.5.1 we considered the effect of the batch size regarding the stability of learning trajectories. This section carries a similar test, but for varying learning rates. Fig. 4.13 presents these results.

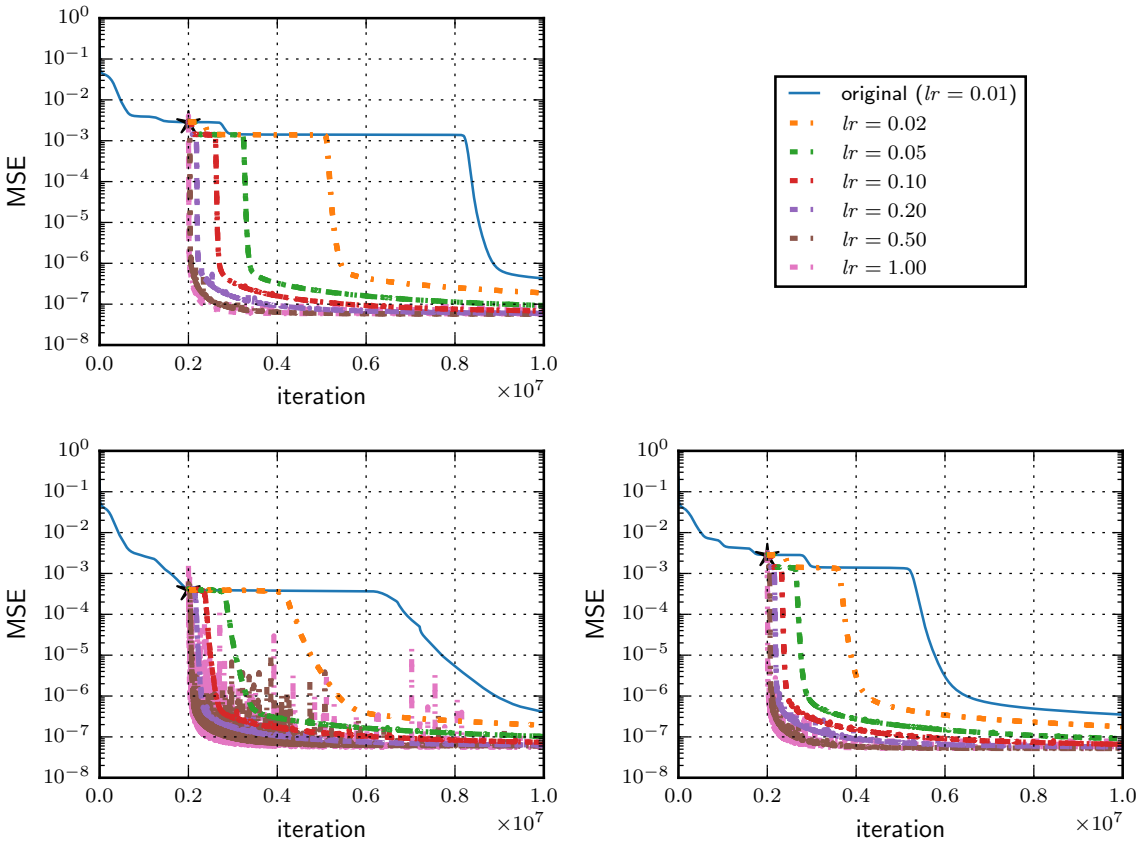


Figure 4.13: Training of different networks using different learning rates. The networks were initialized with the weights the reference network (of their plot) had at the iteration marked by the star. Different plots concern different reference networks (the top left is the same as in Fig. 4.11). Five test runs are plotted for each learning rate.

The conclusions are similar — once again, the learning rate only seems to affect the speed with which a minimum is reached, but not its quality. Furthermore, the

minimum seems to be very similar for networks sharing the same initial configuration — which suggests that the quality of the minimum reached is highly dependent on the initialization. Moreover, it is possible to observe that networks trained with larger learning rates reach the minimum sooner than those trained with lower learning rates, with the time taken to do so being proportional to the learning rate used. This relation is expected since, by taking larger steps, the minimum will be reached proportionally faster.

These results also suggest that if the training of the networks of Fig. 4.2 lasted longer (for instance, ten times longer), then a loss level below 10^{-7} should have been reached. Coincidently, for the larger learning rates used, we can also observe the "instabilities" (the noise) that arise once the loss falls below 10^{-6}.

It seems that neither the batch size nor the learning rate are capable of significantly altering the paths the networks follow towards the minima that they reach. Remember that, so far, we have been considering networks that were initialized with the configuration of weights of a network (the "reference" network) after it was trained for a number of iterations. We know from Fig. 4.2 that completely unrelated test runs (i.e., networks whose initial configurations have no connection) lead to different loss curves. Perhaps we can only observe such behavior by initializing networks with configurations of weights of the reference network that resulted from less training than we have been using thus far.

4.5.3 Varying the initialization point

The last variable that we still need to experiment with is the point at which we collect the configuration of weights of the reference network to initialize other networks. With this in mind, Fig. 4.14 shows the evolution of the loss function of different networks when they share the same initial configuration, collected from the same "reference" network, but at different points in time.

It shows that the overall shape of the loss curve of two networks is significantly determined simply by initializing the networks with the same initial configuration (even prior to any training). Notice from Fig. 4.2 that the overall shape of the loss of networks starting with different initial configurations is quite variable. This variability disappears simply by fixing the initial configuration, which suggests the existence of a connection between the initial configuration of weights of a network and its evolution along time (i.e., training). This link between the initial weight configuration of a network and the effectiveness of its learning will be the focus of further research in Chapter 5, where we look for a more concrete explanation and motivation for its existence.

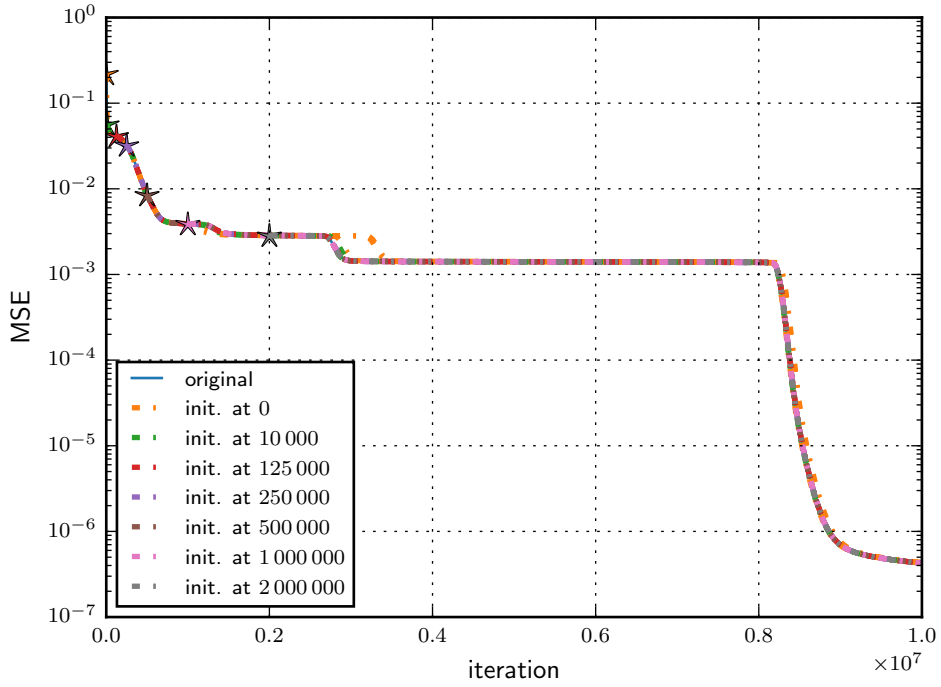


Figure 4.14: Training of different networks varying the instant when the weights of the reference networks are collected and used to initialize the child networks.

4.6 Similarity between learning trajectories

In Sec. 4.5 we saw that the quality of the minima reached by different instances of networks, initialized with the same initial configuration, seems to be highly similar. We also saw that the learning trajectories the networks follow in their weight space seem related, since their loss curve shows the same overall shape. This section presents few preliminary tests we carried about the actual similarity of the trajectories of the networks. To get an initial feeling of the trajectories followed by networks that share the same initial configuration of weights (which we typically call "correlated networks"), consider Fig. 4.15.

It shows the mean of the weights (and biases) of the networks of Fig. 4.12 (top left) along training[4]. With the exception of the curves for a batch size of 1, the means of the weights follow the shape of the reference network somewhat closely. Furthermore, as expected, as the batch size increases, the shape of the curves gets closer and closer to the reference curve (since larger batches lead to better estimates of the gradient). Notice that even though the networks trained with smaller batch sizes suffer larger displacements in their trajectories, they do not necessarily train to better minima, as we observed in Fig. 4.12. Finally, note that the dents in the figure, around iterations

[4]In the figure we have also included curves for networks trained with larger batch sizes to evidence that for larger batch sizes, the curves get closer to the reference.

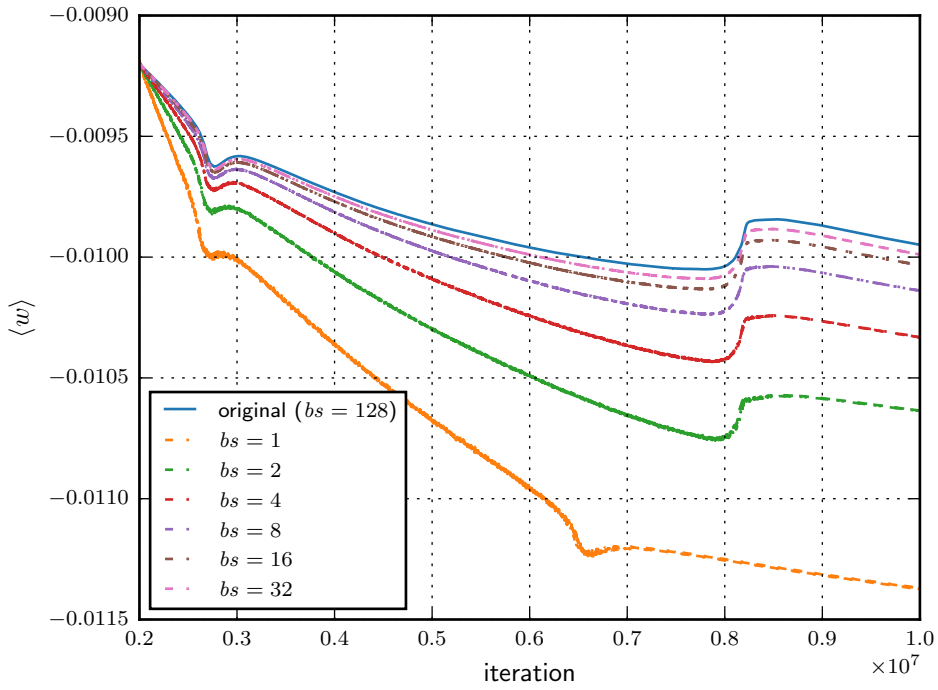


Figure 4.15: Mean of the weights along training of networks sharing the same initialization, for varying batch sizes. The networks considered are the ones shown in Fig. 4.12, top left, alongside curves for networks trained with larger batch sizes. Five independent test runs are shown for each batch size.

0.28×10^7 and 0.82×10^7, happen at the same time that the loss curves fall sharply (observable in Fig. 4.12, top left), and that by doubling the batch size used to train the networks, their mean curves get closer to the reference curve by approximately half their initial distance.

4.6.1 Distance between uncorrelated trajectories

The similarity of trajectories will be mostly evaluated based on the Euclidean distance (i.e., ℓ_2-norm) of the configurations of two networks along training. However, since we have no reasonable scale for this value, before considering the distance of trajectories of networks that have the same initialization (i.e., "correlated trajectories"), it is sensible to consider the distance of uncorrelated trajectories. This serves mainly to establish a baseline value for comparison purposes.

Figure 4.16 shows the Euclidean distance between networks whose initial configurations are unrelated. It shows that, initially, the trajectories seem to approach each other, but this only lasts a few iterations. Afterward, they get further and further apart, until stabilizing at a distance between 18.5 and 19.

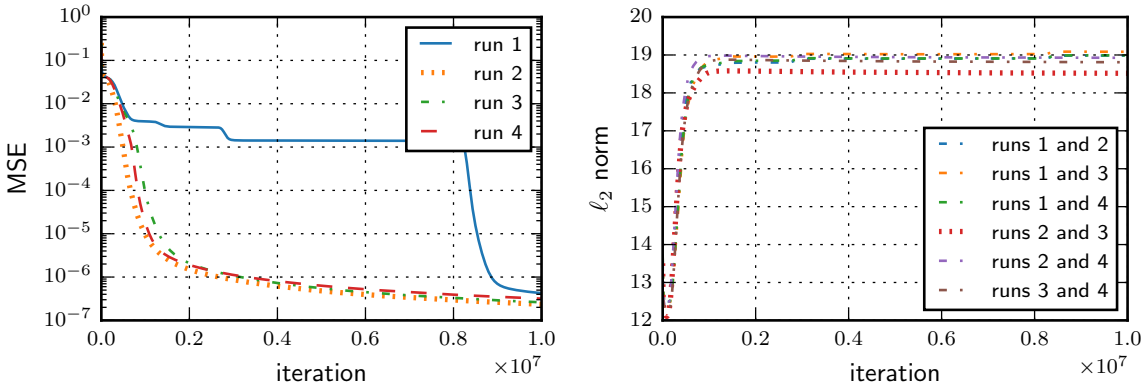


Figure 4.16: Distance between unrelated networks trajectories. (left) test loss of the networks considered. (right) ℓ_2-norm of the difference of two test runs' weights.

4.6.2 Distance between correlated trajectories

Figure 4.17 shows the distance between correlated trajectories of networks trained with varying batch sizes (but starting with the same configuration, the point shown in Fig. 4.11).

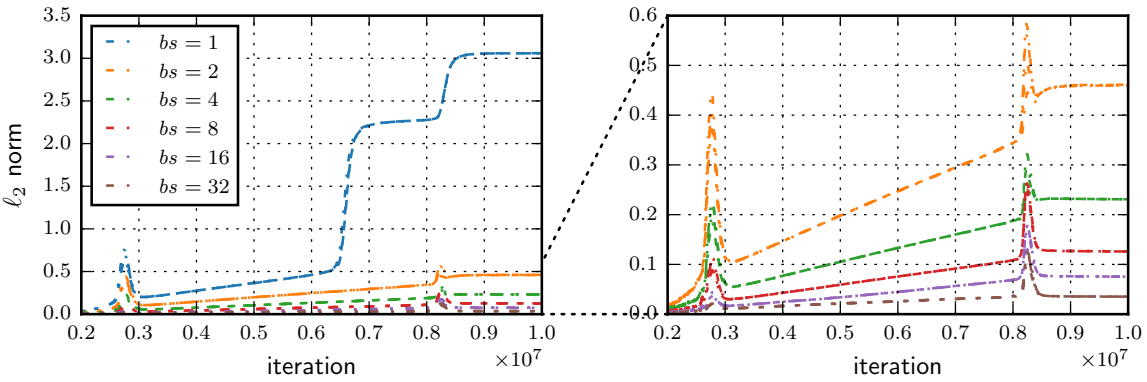


Figure 4.17: Distance between correlated trajectories for varying batch sizes. The distance is measured as the ℓ_2-norm between the reference network, trained with batch size 128 and portrayed in Fig. 4.11, and other networks initialized with the configuration of the reference network and trained with the batch sizes specified in the figure. The loss of these curves for batch sizes of 1–8 has been shown in Fig. 4.12. Networks trained with larger batch sizes do not show noticeably different losses than those trained with batch size of 8. Five different networks are shown for each batch size (the lines overlap).

First, we will focus on interpreting the results for batch sizes greater than one (Fig. 4.17, right). For these values, the plot shows that, whenever the loss function falls sharply (around iterations 2.8×10^6 and 8.2×10^6), there is a significant increase in the distance between the trajectories. However, soon after, the distance decreases considerably again. This behavior suggests that the networks fall in the loss landscape at slightly different instants in time but to approximately the same minimum. The

peaks appear since, even if the networks fall at only slightly different iterations, this tiny difference is enough to cause a significant difference in their positions (especially if the fall is deep). Meanwhile, while the networks are in the plateaus (between iterations 2×10^6 and 2.8×10^6, and 3×10^6 and 8.2×10^6), they seem to be getting further apart at a steady pace, suggesting that they are approximately on the same trajectory, but traveling at different velocities (caused by the usage of different batch sizes between them). This effect is particularly visible between iterations 3×10^6 and 8.2×10^6, where the distance between the trajectories of the networks is increasing linearly. Moreover, the slope of the lines seems to be inversely proportional to the batch size. Around iteration 8.4×10^6, the networks seem to reach a minimum, since, from that time onwards, their distance does not change.

The only networks that, at a glance, do not seem to show the same behavior are the ones trained with a batch size of 1. However, the difference in their behavior can be explained by the early drop in the loss function that they experience (exhibited in Fig. 4.12). It happens at around iteration 6.4×10^6, which is consistent with the jump in the distance observed in Fig. 4.17. It seems that training these networks with a batch size of 1 has allowed them to converge to different minima than the networks trained with larger batch sizes. This idea is supported by the observation that once the reference network falls to its own minimum at around iteration 8.2×10^6, instead of lowering as happened in the other cases, the distance between it and the networks trained with batch size one increases. As a result, this seems indicative that the minima of the networks are different, despite them being of similar quality, as Fig 4.12 showed.

5 Effect of the initial configuration of weights on artificial neural networks

Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.

— Antoine de Saint-Exupéry

effect, noun: power to bring about a result

Using the synthetic dataset, Chapter 4 explored some aspects of the training process of neural networks and the structure arising in their configuration of weights. Among other things, we saw that, for the networks considered there, weights along paths connecting pairs of related input/output nodes show much larger combined strength than unrelated pairs (Sec. 4.4.1). We also saw that, given the way the dataset is constructed, one could use local correlations to infer, to some extent, the organization that the weights of trained networks acquire (Sec. 4.4.2). Perhaps even more valuable towards the contents of the present chapter, we also observed the initial configuration of weights of a network markedly influencing its training (Sec. 4.5 and, specifically, Sec. 4.5.3). The main objective of this chapter is to explore the reasons behind the effect the initial configuration of weights seems to have in the training and functioning of a network. Alongside studying these effects, we also propose and test a few novel initialization strategies.

The chapter is organized as follows. In Sec. 5.1 we present the reference network architecture and other training parameters, such as learning rate and batch size, used for the tests carried in this chapter. Then, in Sec. 5.2 we present a few results with initialization strategies of our design. Afterward, in Secs. 5.3 and 5.4, motivated by observations we made while working on the strategies of Sec. 5.2, we explore the connections between the initial and final configurations of weights of neural networks.

Finally, in Sec. 5.5 we present a initialization strategy based on the findings we made throughout the chapter.

5.1 Reference settings

In the spirit of the work presented in the previous chapter, most of the results shown here are based on a reference network architecture and two datasets. The architecture is a Multilayer perceptron (MLP) with two hidden-layers, based on a sample configuration provided by Keras and available at their repository[1]. The difference between the architecture used in this work and the one they make available is that the one used by us does not include dropout as regularization strategy and uses vanilla Stochastic gradient descent (SGD), instead of RMSprop. The reason behind these changes is seeking a simpler architecture: we are much more interested in understanding the mechanisms of the learning processes than in carrying the most effective/efficient training. Figure 5.1 shows the architecture of the network.

The two datasets used for training are MNIST and HASYv2, which were presented in Secs. 3.2.1 and 3.2.2, respectively. The same base architecture is used irrespectively of the particular dataset used for training. The only differences are the number of nodes in the input and output layers, which are automatically adjusted to the particular dataset used. The networks trained with MNIST use 784 input and 10 output nodes, whereas the networks trained with the HASYv2 use 1024 input and 369 output nodes.

In most cases, training is carried for a very long time, typically 100 or 1000 epochs (far into the overfitting regime). This approach is followed so that long-lasting effects can manifest. The remaining hyperparameters used, unless otherwise stated, are (i) learning rate of 0.1; (ii) batch size of 128 samples; and (iii) categorical cross-entropy loss function. These are common values that were found to work well. As stated previously, notice that, in general, these parameters are not tuned towards the fastest training possible. In fact, they are particularly conservative for the networks trained with the MNIST dataset (where a learning rate of 1, for example, could be used successfully). We followed this approach to keep all settings as simple as possible, and to allow the usage of the same hyperparameters on both datasets (MNIST and HASYv2).

[1] https://raw.githubusercontent.com/keras-team/keras/master/examples/mnist_mlp.py. Accessed 2019-11-05.

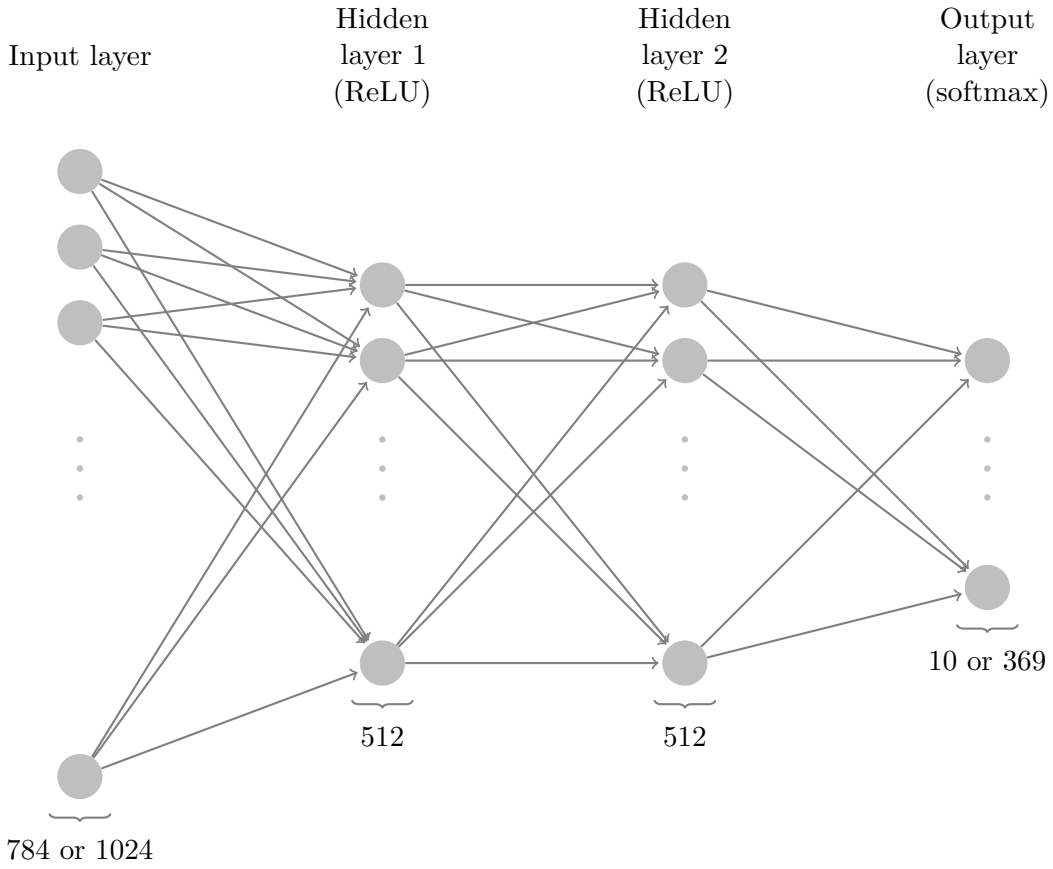


Figure 5.1: Base architecture of the networks trained (bias units have been omitted). 784 input and 10 output nodes are used when training with MNIST, whereas 1024 input and 369 output nodes are used for the HASYv2 dataset.

5.2 Preliminary tests with initialization strategies

In this section we experiment with initialization strategies inspired by ideas we developed while working on Chapter 4. Our rationale is that, by combining our findings about the strength and correlations of the input/output paths with existing initialization methods, we may improve them. After having tested them, we conclude that these ideas may be worth pursuing. However, we acknowledge that while developing them we found an interesting feature of the learning process (which we study in more detail from Sec. 5.3 onwards), that ultimately carried us away from further developing the initialization strategies presented in the current section.

5.2.1 Lightning-based initialization

We start by considering a simple initialization strategy based on the Lightning initialization of Pircher et al. (2018), which was previously presented in Sec. 2.5.4. To summarize, the lightning initialization involves choosing a number of end-to-end paths

along the network (from input to output nodes), setting the weights along these paths with nonzero values, and the remaining weights to zero.

The variation of the Lightning initialization proposed here is based on creating a certain number of input-output paths per output node, initializing the weights along the paths with a fixed value, and the remaining weights to zero. The paths are created as follows. For each output node, we rank the input nodes according to the correlation[2] they show with the output node. A number of the top-scoring input nodes are connected to the output node by, for each input node selected, setting the weights of a random path connecting it to the output node, through nodes of the intermediate hidden layers, with a fixed value. The remaining weights are set to zero. The intermediate nodes in the hidden layers are chosen from "bands", one for each output node, so that situations where random paths cross at an intermediate node may only arise for paths that lead to the same output node. The number of input nodes per output node (i.e., the number of paths created per output node) and the initial value with which the weights are initialized are parameters of the strategy. Algorithm 5.1 provides pseudocode for the procedure.

Figure 5.2 presents a matrix with the results of initializing networks with this procedure for several combinations of the number of inputs chosen and initial value of the weights. Unfortunately, it is not possible to compare our results to the Lightning initialization (Pircher et al., 2018), since in their paper the authors have used different network architectures and hyperparameters. Moreover, it is not clear whether the results they report are based on the train or test datasets, and since the paper does not reference the source code used to carry their experiments we cannot reproduce them. Due to this, we restrict ourselves to comparing our results with the commonly used Glorot uniform initialization (presented in Sec. 2.5.1), which we show in the first row of the matrices plotted in the figure.

The figure shows that both parameters affect training. There is a sweet spot when choosing 10% of the top-scoring input nodes per output node, alongside an initial weight strength of 0.5–0.75. Near this region, there is an improvement in the train loss when compared to the baseline, and a marginal improvement in the test loss. However, in general, the initialization does not generalize better, since most combinations of the parameters of the algorithm lead to a higher test loss. Furthermore, slight changes in these parameters lead to poor training (particularly on the test set). The lack of robustness the method shows to its parameters seems to suggest that it is not general.

This initialization scheme seems to be too "evasive", needing significant fine-tuning in order to be useful (otherwise, it may even behave "catastrophically"). Notwithstanding

[2]We used the Pearson correlation coefficient (PCC), $\rho_{X,Y} = \mathrm{Cov}\,(X, Y)/(\sigma_X \sigma_Y)$, to measure the correlation between a pair of input and output variables.

Algorithm 5.1: Pseudocode for the lightning-based initialization

Data: List of the weight matrices (wl) of a network. A set of input (x) and output (y) training samples (to use for computing the correlation between input and output variables). A number k specifying the number of inputs to choose per output node. A number s specifying the initial value (the "strength") with which the weights in the paths are initialized.

Result: The weight matrices w in wl are initialized according to our lightning-based initialization with the given parameters.

Let ni and no denote the no. inputs and outputs of the network (respectively)
foreach $w \in wl$ *do* ▷ Initialize all weight matrices w with zero
    $w \leftarrow 0$
end
for $j \leftarrow 1$ *to* no *do*
    $\rho \leftarrow$ pearson-corrcoef(x, y, j) ▷ Compute the PCC between all inputs and the j-th output. ρ will be a vector of size ni where its i-th entry contains the PCC between the input i and output j
    $I \leftarrow \text{rank}(\rho, k)$ ▷ Select the set of k top-scoring inputs based on the absolute value of their correlation with the output
    foreach $i \in I$ *do* ▷ Create one path at a time (each starting in node i)
        foreach $w \in wl$ *do* ▷ Create the path layer by layer
            Let nr denote the no. nodes to the right of the layer of weights w (hence, if w is the last layer, $nr = no$)
            if w *is the last layer* *then* ▷ Choose a node o connected to node i whose link we will initialize. If w is the last layer of weights, that node is j
                $o \leftarrow j$
            else ▷ Otherwise, choose it at random from the j-th group of nodes, so that the same node o may only be used in different paths if they lead to the same output node. The groups are assigned sequentially, e.g., if $nr = 512$ and there are four output nodes ($no = 4$), the first group will be $\{1, 2, \ldots, 128\}$, the second $\{129, 130, \ldots, 256\}$, and so on
                $o \leftarrow \text{rand}((j-1) \cdot \lfloor nr/no \rfloor + 1, j \cdot \lfloor nr/no \rfloor)$
            end
            $w_{io} \leftarrow s$ ▷ Set the weight chosen with the desired initial value
            $i \leftarrow o$ ▷ We will use the node o that we chose in the current layer as the starting node in the next (to avoid breaking the path)
        end
    end
end

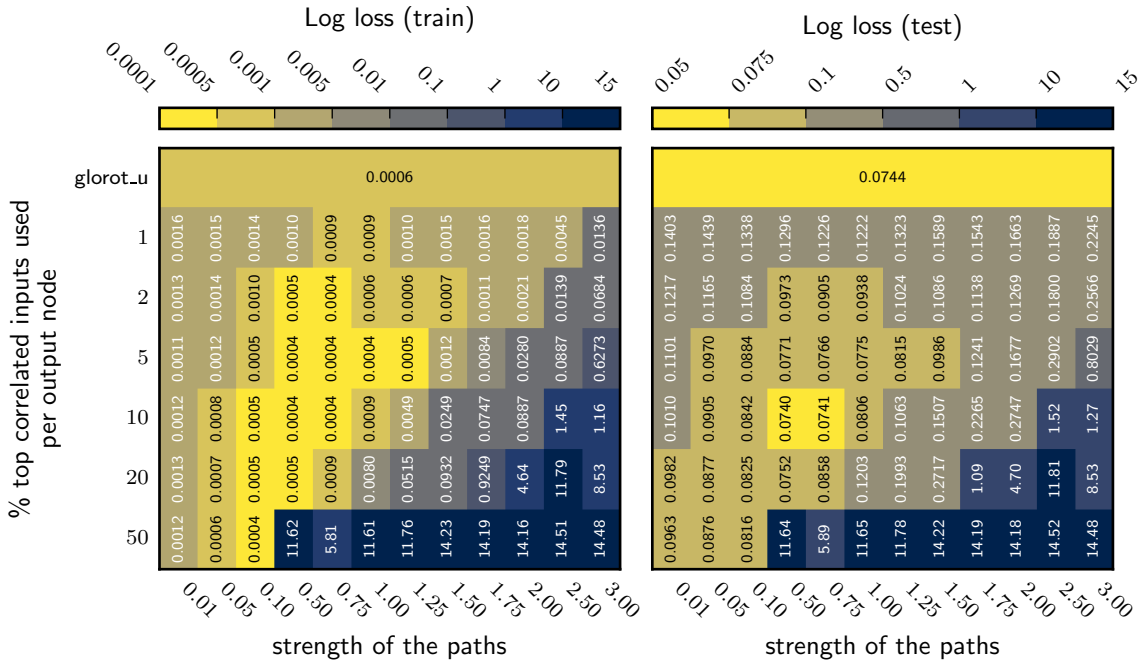


Figure 5.2: Error obtained with our lightning-based initialization for several combinations of parameters. Training lasted for 100 epochs on the MNIST dataset. Each cell shows the average of five independent runs. The left figure shows the training error, whereas the right one shows the test. The first row on both figures (labeled `glorot_u` in the left) contains the baseline result obtained with Glorot's uniform initialization.

its downfalls, this strategy inspired us to pursue a more conservative procedure that borrows ideas from it (e.g., the notion of bands) and merges them into a typical "fully random initialization". This new strategy is described in the section that follows.

5.2.2 The Dense Sliced Initialization

This section presents the Dense Sliced Initialization (DSI). The idea of this strategy is to densely initialize end-to-end bands (or "slices") in a network according to a typical random initialization, and set the remaining weights (i.e., weights that connect nodes that belong to different bands) to zero. In other words, the weights that connect nodes belonging to the same band/slice are initialized with random weights, but the remaining connections (between bands) are set to zero.

The method works as follows. The nodes in each hidden layer are divided into n groups (the number of bands desired), which are labeled 1 to n. The set of groups having the same label (at different layers) is called a slice. The weights connecting groups of nodes in consecutive layers that are part of the same slice are initialized with random values from some distribution. The remaining weights (the ones that connect groups with different labels) are initialized with zero. Then, a random output class is

assigned to each slice. The slice's nodes of the last hidden layer are connected to the assigned output node with random weights, whereas the remaining weights (connecting to other output nodes) are set to zero. Finally, a random selection of the input nodes is connected to the slice in a similar fashion. These nodes are either sampled uniformly (at random) or are sampled with probability proportional to the correlation they show with the output node the slice connects to[3]. Figure 5.3 illustrates the procedure, and Alg. 5.2 provides pseudocode for it.

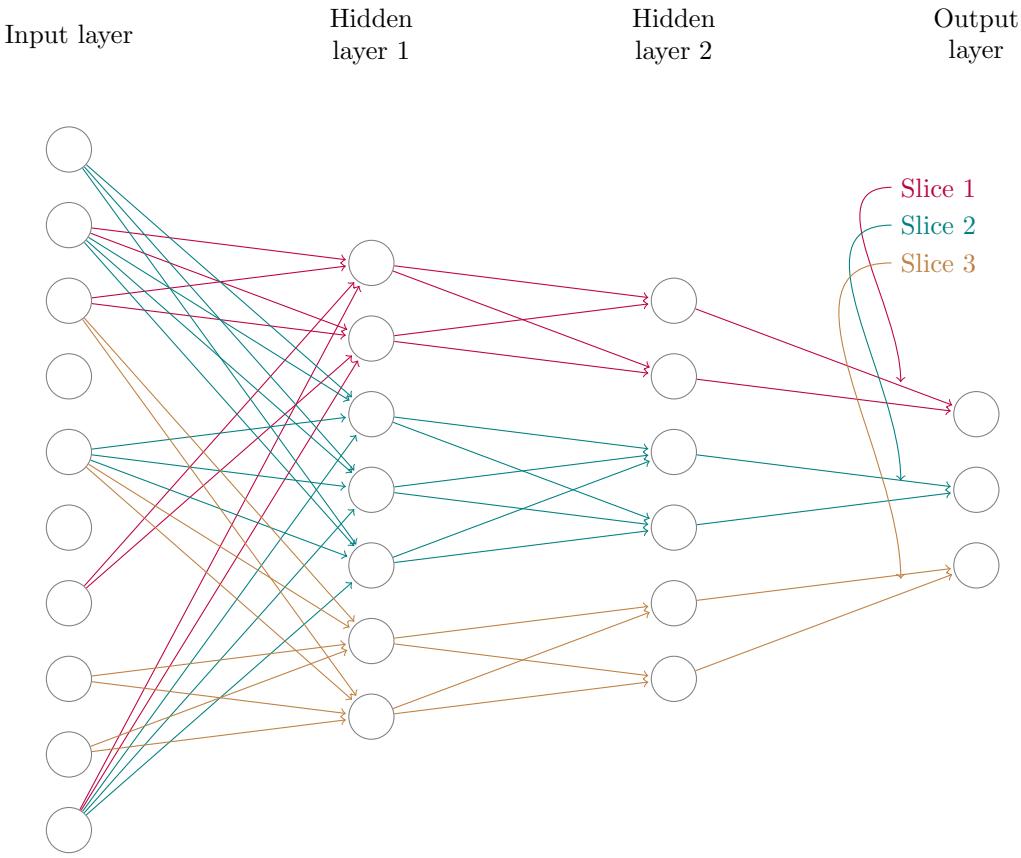


Figure 5.3: The dense sliced initialization. Conceptually, this strategy separates the nodes of a network up to the first hidden layer into slices (in the figure we show three). The weights connecting nodes that belong to the same slice are initialized with random values, in a similar manner to a typical random initialization. The remaining weights (that connect nodes that belong to different slices) are initialized with zero (they were omitted in the figure for clarity). Finally, a random group of input nodes is selected for each slice and connected to it.

The rationale behind this strategy is that random initializations may create conflicting paths of information within the network. The learning algorithm must solve these inconsistencies before it can start working on a proper solution. Creating confined regions within the network that start mostly independent from each other (the slices) may help during the early stages of training since they should reduce the amount of

[3]Similarly to Sec. 5.2.1, we have used PCC to measure the correlation between input and output nodes.

Algorithm 5.2: Pseudocode for the dense sliced initialization

Data: List of the weight matrices (wl) of a network. A number k specifying the number of inputs to connect per slice. A number n specifying the number of slices to create.

Result: The weight matrices w in wl are initialized according to the dense sliced initialization with the given parameters.

Let ni and no denote the no. inputs and outputs of the network (respectively)
foreach $w \in wl$ *do* ▷ Initialize all weight matrices w with zero
    $w \leftarrow 0$
end
for $i \leftarrow 1$ *to* n *do* ▷ Initialize one slice at a time
    if $i \leq no$ *then* ▷ Choose the output node that is assigned to the slice. The first no slices are assigned sequentially, so that as as long as $n \geq no$, each slice gets assigned a different output with certainty
        $o \leftarrow i$
    else ▷ The remaining slices get assigned a random output
        $o \leftarrow \text{choose}(no)$
    end
    $si \leftarrow \text{choose}(ni, k)$ ▷ Choose k input nodes to connect to the slice (i.e., whose weights are going to be initialized with nonzero values). The input nodes may be sampled uniformly as we are doing here, or based on their correlation with the output node o
    foreach $w \in wl$ *do* ▷ Create the slice by initializing its weights layer by layer
        Let nl denote the number of nodes to the left of the layer of weights w and nr the no. nodes to the right (in the first iteration $nl = ni$, and in the last $nr = no$)
        if w *is the last layer* *then* ▷ Choose the nodes of the layer to the right of w that belong to this slice. If w is the last layer of weights, the only such node is o
            $so \leftarrow \{o\}$
        else ▷ Otherwise, choose the nodes sequentially. E.g., if $nr = 512$ and there are eight slices ($n = 8$), the first slice gets nodes $\{1, 2, \ldots, 64\}$, the second gets $\{65, 66, \ldots, 128\}$, and so on. To simplify, we wrote the expression below assuming $nr \equiv 0 \mod n$. If that is not the case, the "extra" nodes can be assigned to different slices sequentially or at random.
            $so \leftarrow \{(i-1) \cdot \lfloor nr/n \rfloor + 1, (i-1) \cdot \lfloor nr/n \rfloor + 2, \cdots, i \cdot \lfloor nr/n \rfloor\}$
        end
        ▷ Initialize the weights connecting the nodes in si and so by drawing them from a uniform distribution so that they have a variance of $2/(|si| + |so|)$
        foreach $\hat{\imath} \in si$ *do*
            foreach $\hat{o} \in so$ *do*
                $w_{\hat{\imath}\hat{o}} \sim U\left(-\sqrt{6/\left(|si| + |so|\right)}, \sqrt{6/\left(|si| + |so|\right)}\right)$
            end
        end
        $si \leftarrow so$ ▷ The nodes that were to the right of the current matrix of weights (those in so) are the ones that are to the left of the next matrix (in si)
    end
end

conflicting flows, as well as the effort required to eliminate them. In essence, the idea is to remove from the learning algorithm the burden of resolving the conflicting flows, at the expense of a possibly less complex/rich initial configuration.

Figures 5.4 and 5.5 show the loss obtained for networks initialized with the dense sliced initialization, for a combination of the number of slices used and the number of inputs connected to each slice. The reference initialization strategy, Glorot uniform initialization, is shown in the first row of the plots.

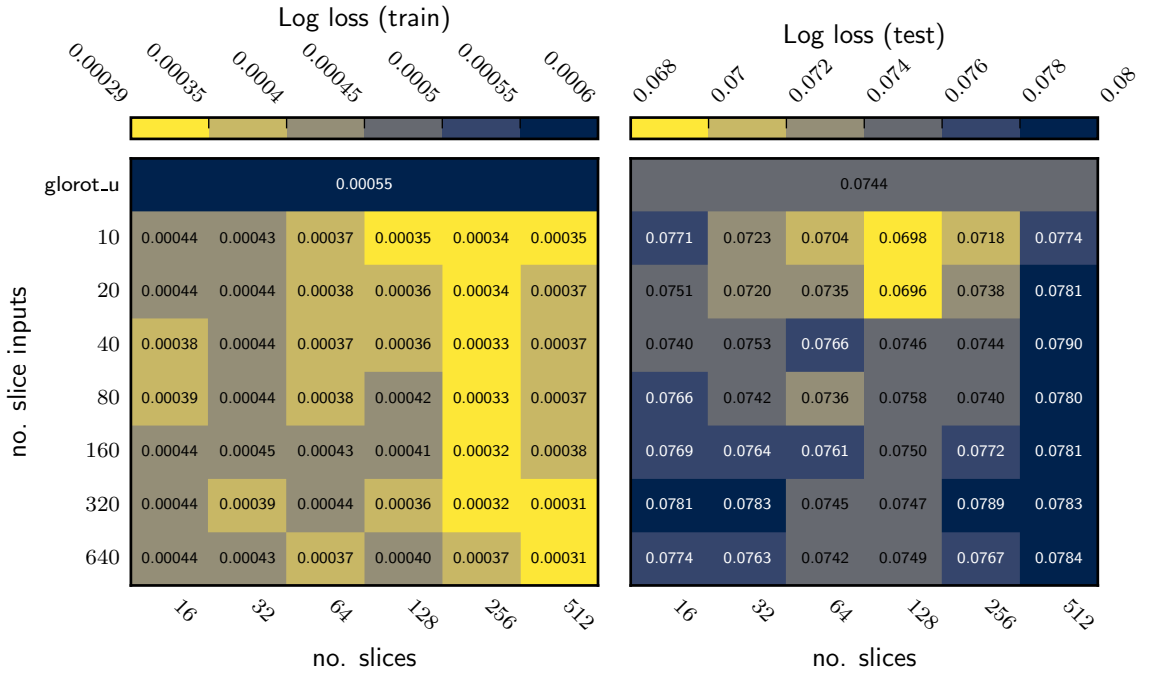


Figure 5.4: Error obtained with the dense sliced initialization for several combinations of parameters and when input nodes are sampled with probability proportional to the correlation they show with the output. Training lasted for 100 epochs on the MNIST dataset. Each cell shows the average of five independent runs. The left figure shows the train loss, whereas the right figure shows the test loss. The first row on each matrix shows the baseline loss obtained with Glorot's uniform initialization.

Two main parameters of the algorithm were tested, the number of slices and the number of inputs provided to each slice. The algorithm seems mostly robust with respect to both these parameters, since all configurations achieve somewhat comparable train loss (which is always smaller than the baseline train loss, with the improvements ranging from 20 to 45%). However, in general the strategy does not seem to lead to significantly better generalization, since the best test loss values obtained are comparable to the baseline. Moreover, it seems that sampling input nodes uniformly to connect to the slices leads to comparable (even marginally better) results than sampling them based on their correlation with the output of the slice. Overall, the algorithm shows good results and looks like a promising technique to consider in future research.

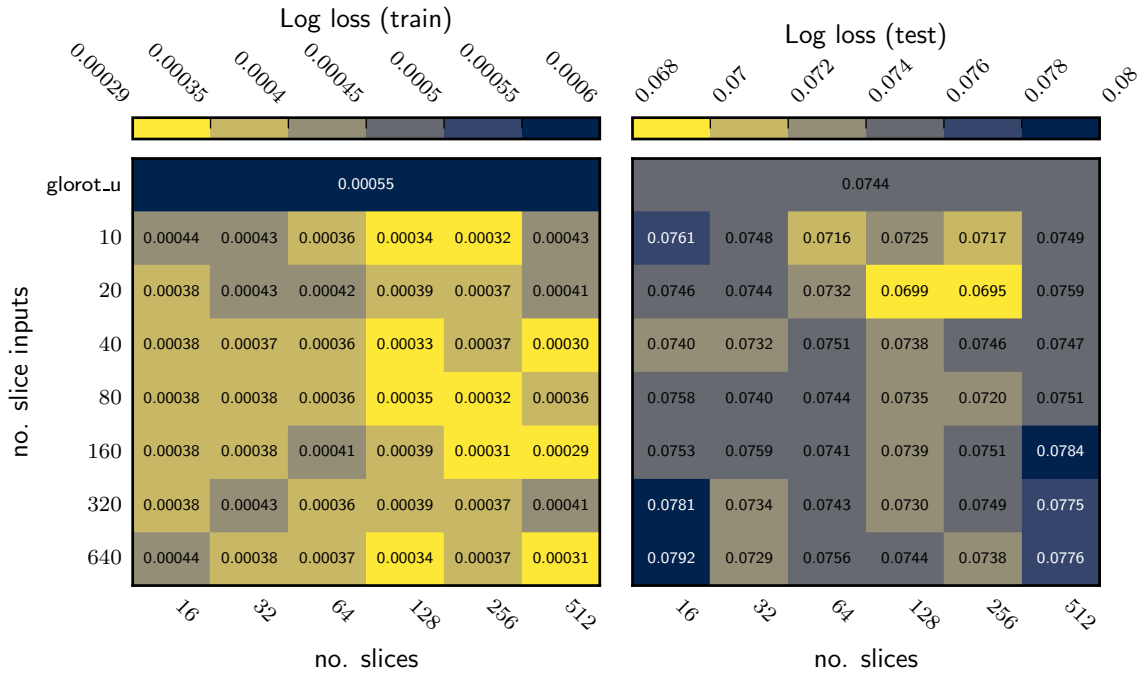


Figure 5.5: Error obtained with the dense sliced initialization for several combinations of parameters and when input nodes are sampled uniformly. Refer to Fig. 5.4 for the conditions of this experiment and the meaning of the figure (the difference between the two figures is that, in the present one, the input nodes connected to each slice are sampled uniformly, whereas in Fig. 5.4 they are sampled with probability proportional to their correlation with the output).

5.3 Inspecting a network's initial configuration of weights

In the previous chapter, in Sec. 4.5, we saw that the initial configuration of weights of a network seems to be capable of shaping its training to a significant degree. Furthermore, while working on the strategies previously presented, in Secs. 5.2.1 and 5.2.2, we observed a trend in training, which will be discussed in the following sections, that was not all that obvious to us initially — it seems that the final configuration of weights of a network does not change that much away from the initial one. In other words, it seems that training is, to some extent, a fine-tuning process that meticulously updates the weights of the initial configuration. Further sections of this chapter will explore this observation in more detail. We will observe how much the weights, after training, differ from the initial ones, and find that the deviations are relatively small, to the point that marks of the initial configuration of weights of a network are still perceptible in its weights even after extensive periods of training. Moreover, the small nature of these deviations seems to be a fingerprint of trainability — if a network is not trainable, then it seems that it loses the traces of its initial state. These findings led us to an initialization strategy that we present in Sec. 5.5, where we observe that when the initial configuration of weights is arranged using samples from the training dataset, we

obtain significant improvements to the training accuracy measured, but at the expense of worse generalization.

5.3.1 The lottery ticket hypothesis

As we stated in Sec. 2.5, and somewhat experienced firsthand throughout the previous sections, the initial configuration of weights of a network is capable of playing a crucial role in its training. However, as we reviewed in Sec. 2.5, the reasons making the initialization so important are not all that clear. Nonetheless, a recent work, which we overview below, may help to explain these reasons, at least partially.

It was recently observed by Frankle and Carbin (2019) that typically, a randomly initialized dense neural network contains subnetworks (winning tickets) that can match the test accuracy of the original network when trained in isolation for the same amount of time (the Lottery Ticket Hypothesis). Importantly, these subnetworks are part of the initialization (i.e., the initial configuration of weights and biases), instead of being a product of the training process, meaning that they are present since the very beginning of training, and not constructed along training. This hypothesis suggests that there are specific structures in a random initialization that are crucial for training and to allow a network to achieve high accuracy.

Frankle and Carbin (2019) reported consistently finding winning tickets that are less than 10–20% the initial size of many different networks they tested, such as several fully-connected and convolutional feed-forward architectures trained on MNIST and CIFAR10. It has been known for a long time (e.g., Han, Pool, Tran, & Dally, 2015; Hassibi & Stork, 1993; G. Hinton, Vinyals, & Dean, 2015; LeCun, Denker, & Solla, 1990) that it is possible to reduce the size of trained networks to less than 10% of their initial size without affecting their accuracy significantly (a process usually called pruning). What had not yet been realized was that this could, in theory, be achieved prior to any training. If one succeeds in identifying these networks (the winning tickets) before training, one could save tremendous amounts of training time. Moreover, identifying the important parts of a configuration of weights and pruning the rest prior to any training should improve generalization too (by reducing the number of adjustable parameters of the model, following Occam's razor principle). Unfortunately, Frankle and Carbin (2019) find the winning subnetworks only after training with the complete initial network.

These results suggest that success in training depends largely on particular patterns existing in the initial configuration of weights of a network, but the nature or composition of these patterns remains unclear. Further sections of this chapter explore the link between the initial and final configurations of weights in feedforward networks, trying to understand what happens to make these patterns so important.

5.3.2 Similarity between the initial and final configurations of weights of a network (first impressions)

In this section, in order to start acquiring some intuition about the relations that seem to exist between the initial and final configurations of weights of trained neural networks, we will consider 5 networks trained for 1000 epochs on MNIST, whose loss (and accuracy) are shown in Fig. 5.6. As the figure shows, the networks are already clearly in overfitting, long past the point of best generalization. However, this is not a concern to us since we are interested in observing long lasting effects of the training process (which, as a result, are transversal to both under- and overfitting regimes). We can see that there are two groups of networks, one that converged to a loss above 10^{-4}, and another that is converging slightly above a loss of 10^{-5}.

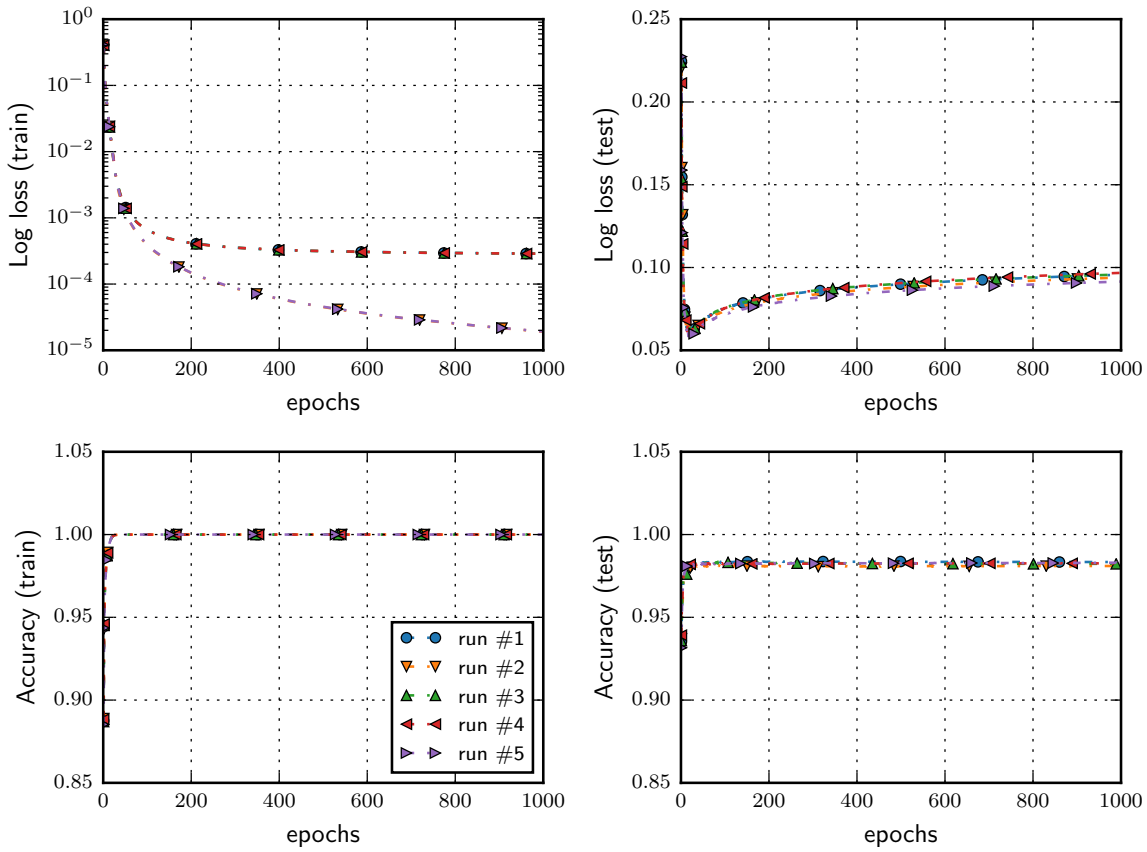


Figure 5.6: Train (left) and test (right) loss (top) and accuracy (bottom) of five networks trained with the MNIST dataset for 1000 epochs.

During our experiments, we initially examined the initial and final configurations of weights of these networks by visual inspection. We did this by overlaying images of the weight matrices of the initial and final configurations (from here on denoted w_i and w_f,

respectively) of a network on top of each other and "flicking" them[4]. By initial weights we mean the values of the weights of a network prior to any training. Similarly, by final weights we refer to the values of a network's weights once training is stopped.

Despite being a rather primitive method, this approach provided us with an initial intuition about what, to some extent, is happening to these configurations — they are surprisingly relatable, in that clusters of weights that appear by chance in an initial configuration are typically clearly identifiable (and sometimes even "enhanced") in the final configuration. The main problem of this approach is that it does not lend itself well to be presented on paper (e.g., this thesis). As an alternative, we will start by looking into the distribution of the products of initial and final weights of a trained network. This measures how many weights swap their sign during training. We observe that the vast majority of the weights of a network conserve their sign, as Fig. 5.7 shows.

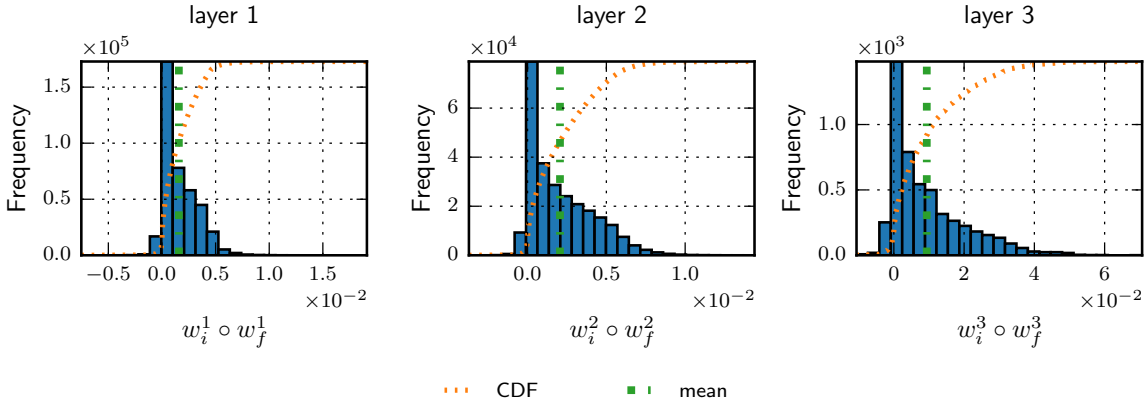


Figure 5.7: Distribution of the entrywise product (Hadamard product) of the initial and final configurations of weights (i.e., $w_i \circ w_f$), for run no. 1 of Fig 5.6. The results for the other test runs follow the one presented (they are omitted for clarity). The cumulative distribution function (CDF) of the distribution in the respective plot was normalized to span the full y-axis.

It plots the distribution of the entrywise product (i.e., Hadamard product, $\circ$) between the initial and final configurations of weights, $w_i \circ w_f$, for each layer of weights of a network. The products are positive if the sign a weight had prior to training is the same it has after training, and negative otherwise. This is what happens in general, as the figure shows. This observation comes somewhat at a surprise, since there is no apparent reason as to why this should happen. As an aside, note that the mean of the distribution of these products happens to be the covariance between w_i and w_f, i.e.,

[4]By flicking we mean showing the weights of the initial configuration of weights, replacing the resulting image with one for the weights of the final configuration, going back to the initial image and repeating continuously. E.g., consider a slideshow with two slides, each containing an image that fully occupies it, and the user is continuously swapping between the two slides in order to try to spot the differences between the images (akin to the effect of a thaumatrope).

$\mathbb{E}[w_i \circ w_f] = \mathrm{Cov}(w_i, w_f)$, since the distribution from which the initial configuration is drawn is symmetrical ($\mathbb{E}[w_i] = 0$). Thus, the plots also show that the initial and final configurations of weights are correlated.

Another intriguing effect observed when we flicked the initial and final configurations of weights is that many groups of weights seem to start in a disposition that is enhanced by training. In other words, it seems that a cluster of weights that overall starts with, for instance, more positive values (likewise for negative values) influences nearby weights to be positive too, as if the initial groups of coordinated weights function as seeds that serve as "guidelines" for the training process. To observe this effect, consider the experiment whose results are shown in Fig. 5.8.

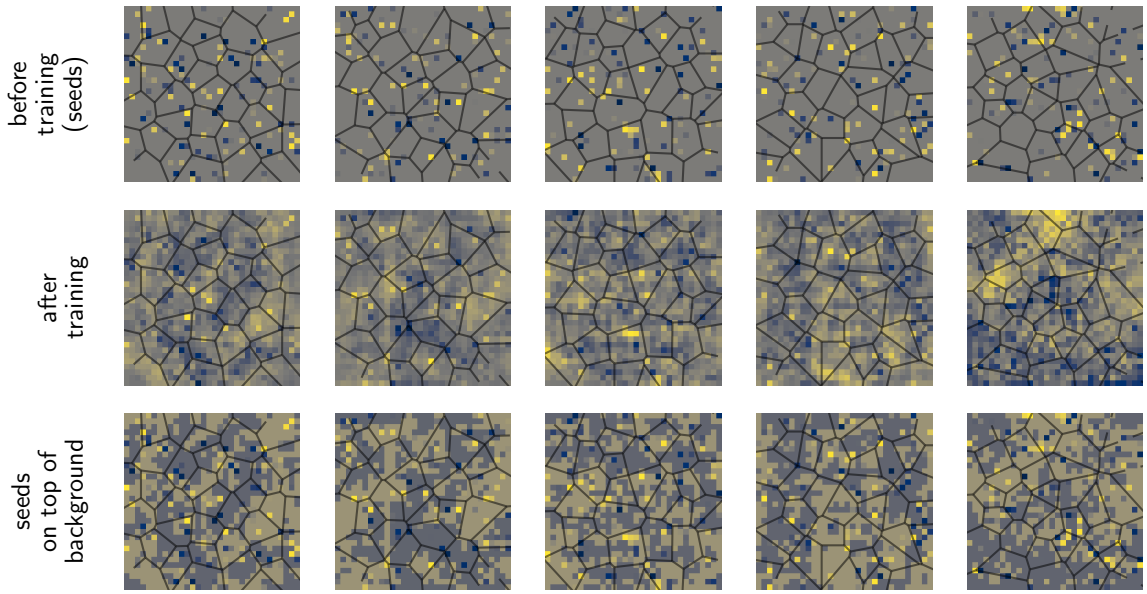


Figure 5.8: Visualization of the spread of initialization seeds on five networks trained on the HASYv2 dataset (for 100 epochs). The top row shows, in each subplot, the weights connecting the input nodes to a randomly selected node of the first hidden layer. The weights have been reshaped into a 32×32 matrix to mimic the way they would connect to input samples. The middle row shows the values of the same weights after training. The bottom row shows the initial nonzero weights (the seeds) overlaid on top of a background given by the sign of the final weights (brighter colors denotes positive sign, darker colors denote negative). The black lines plot a Voronoi diagram computed by clustering the seeds and serve mainly to provide visual guides across the different rows.

The figure plots weights that connect input nodes to randomly sampled hidden nodes of a set of networks trained on HASYv2. Each hidden node connects to 1024 input nodes, the number of pixels of the HASYv2 inputs. During initialization, all but 128 of these weights were set to zero. This allows one to observe how the nonzero weights affect the zero weights. We call these nonzero weights "seeds". Each vector of weights connecting input nodes to a hidden node is rearranged into a 32×32 matrix

(the shape of the input samples). A random selection of these matrices is plotted in the top row of Fig. 5.8. The same weights are plotted beneath, in the middle row, after 100 epochs of training. It seems that, in general, seeds that are more positive or more negative propagate their sign to nearby weights, creating regions that are tendentiously positive or negative. To highlight this behavior, the bottom row in the figure shows the seed weights overlaid on top of a background that takes color based on the sign of the final weights (darker color denotes negative sign, brighter color positive). For visual aid, the seeds were clustered into 40 clusters by the K-Means algorithm (Lloyd, 1982). Afterward, the Voronoi diagram of the clusters' centroids was overlaid on the images of the figure.

These observations suggest that chance causes certain groups of weights to start with a higher concentration of positive or negative values. These fluctuations originate clusters of weights that happen to be good (or, at least, "better than the others") recognizers, or "filters", of particular features of the dataset. As a result, instead of creating these filters from nothing, the learning process seems to take advantage of their natural occurrence and tune the weights appearing in these regions, so that they become proper recognizers for the features of the dataset.

Notice that, in the experiments portrayed in Fig. 5.8, we considered only the first layer of weights of a neural network and set around 90% of its weights to zero prior to any training. However, the effect we are suggesting is applicable to all the layers of a network and regardless of setting, or not, weights to zero. The effect simply becomes more visible in the first layer of weights, since in the remaining layers it is not trivial to define which weights "are close". In the first layer, doing so is eased due to the fact that nearby input pixels are correlated, and hence so should be the weights that connect to them. Finally, setting a fraction of the weights to zero helps merely by removing the clutter arising from otherwise showing hundreds of weights in a small area.

The idea of filters that emerge in the initialization due to small fluctuations in the initial configuration of weights of a network, and its implications, is the primary matter investigated in the sections that follow. Notice that this idea may also provide different insights to the lottery ticket hypothesis. The winning tickets may simply be the weights that happen to start, by chance, in configurations that make them better for particular patterns present in the dataset.

5.4 Tuning of weights during training

The previous section, Sec. 5.3.2, suggested that a random initialization may create fluctuations in the initial configuration of weights that cause particular groups of weights

to be reasonably good at identifying certain patterns of the dataset. Considering this, one may conceive training to consist, to a large degree, of (fine-)tuning these groups (and nearby weights) to improve the identification of patterns. The aim of this section is to study the scale of the fine-tuning, i.e., how much the weights are typically updated during training, and pursue ways of identifying this behavior.

5.4.1 Marking the initial configuration of weights

In moderately-sized networks, due to their large number of parameters, it becomes challenging to identify clusters of weights whose initial and final values are similar/related. Nonetheless, by resorting to our pattern matching capabilities, we may be capable of doing so if we somehow imprint some characteristic (i.e., visible) mark in the initial configuration of weights. By knowing where to look for marks, we may more easily verify whether they are or not present after training. Moreover, it allows us to track the mark along the training, hence enabling us to understand better what happens to the weights of the network. We achieved this by stamping letters from the Latin alphabet on a layer of a neural network. The idea is to introduce some clear and arbitrary pattern on the network that can be viewed effortlessly at a macroscopic level.

If the mark is still visible after the weights suffer a considerable amount of updates, this would suggest that training is not an abrupt process that causes the weights of a network to drift far from their initial state, but more like a fine-tuning process, where interesting patterns introduced by chance during the initialization are tuned to fit a particular dataset well. Figure 5.9 shows a mosaic of initial vs. final configurations of weights that were stamped with letters.

The stamping process was performed by considering a bitmap with the same shape as the matrix of weights of the layer being marked. Then, we rasterized the desired letter to the bitmap, and used it as a binary mask to clear the weights that laid outside the trace of the letter (or vice-versa, i.e., clearing the weights that were inside the trace of the letter) — in the figure we included cases of clearing weights inside and output a letter. The figure shows that, in general, the letters are clearly visible after training. Perhaps even more surprisingly, they do not affect the quality of training, in that independently of the letter chosen (or whether a letter is present or not) the networks train to approximately the same loss value. This is further evidenced in Fig 5.10, which shows the training of networks both without marks and marked with the vowels `A`, `E`, `I`, `O`, `U`, alongside their loss, when training lasts for as much as 1000 epochs (the networks are long into the overfitting regime and have almost fully converged).

While being macroscopic observations, the previous figures portray that, to some extent, randomly initialized networks do retain characteristics of their initialization along

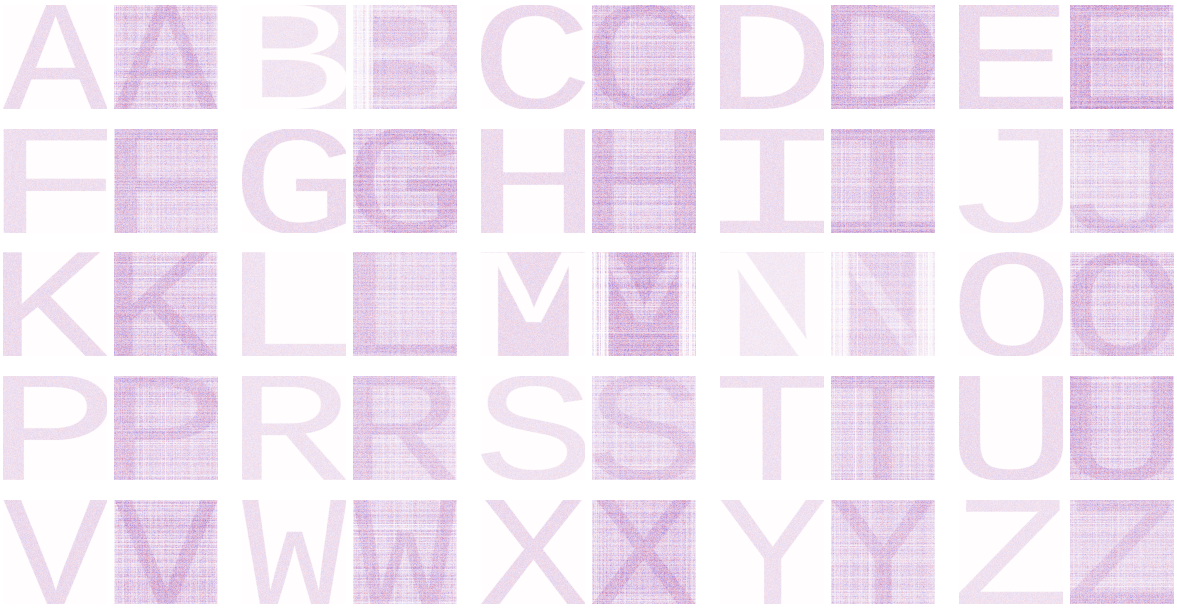


Figure 5.9: Stamping of letters on the weights of a neural network. A pixel in an image represents a weight of a network. Each pair of images represents a training realization where the middle layer of weights of a network was stamped with the letter that the images show. The left image shows the weights of the marked layer prior to training, whereas the right image shows the same weights after. Training was carried on the HASYv2 dataset for 50 epochs. Each pair of images was normalized to the same color range, so that the same color in the two images refers to the same numerical value. The letter `Q` was left out of the figure due to its similarity with the letter `O` (the English alphabet contains 26 letters, the dimensions of the image allowed showing 25).

the whole training, and transfer them into their final applications. Moreover, the effect is sufficiently strong to allow us humans to easily identify such marks, especially if they are deliberately embedded into the configuration of weights of a network. Furthermore, doing so surprisingly does not seem to noticeably affect the training of the networks.

A question that may arise at this point is whether the gradient (i.e., the updates suffered by the networks) of the marked networks shows any trace of the letters imprinted on them. The answer seems to be negative, as Fig. 5.12 shows.

One could expect to see traces of the mark in the gradient, but the fact that we do not may be explained by recalling that we are considering the middle layer of weights of the network. As we mentioned previously, the way the "meaningful" groups of weights may be constructed is unpredictable in layers that are not the first one. I.e., there may be weights inside the letter that promote the emergence of initialization filters, similarly to what we observed when we considered the initialization seeds in Fig. 5.8. However, unlike what we observed in the seeds, it is unlikely that those weights are nearby each other in the weight matrices, since they connect to nodes that have no meaningful spatial arrangement. As a result, weights that are "far apart" may form a group and they may be updated in unison, but the gradient itself, by showing all

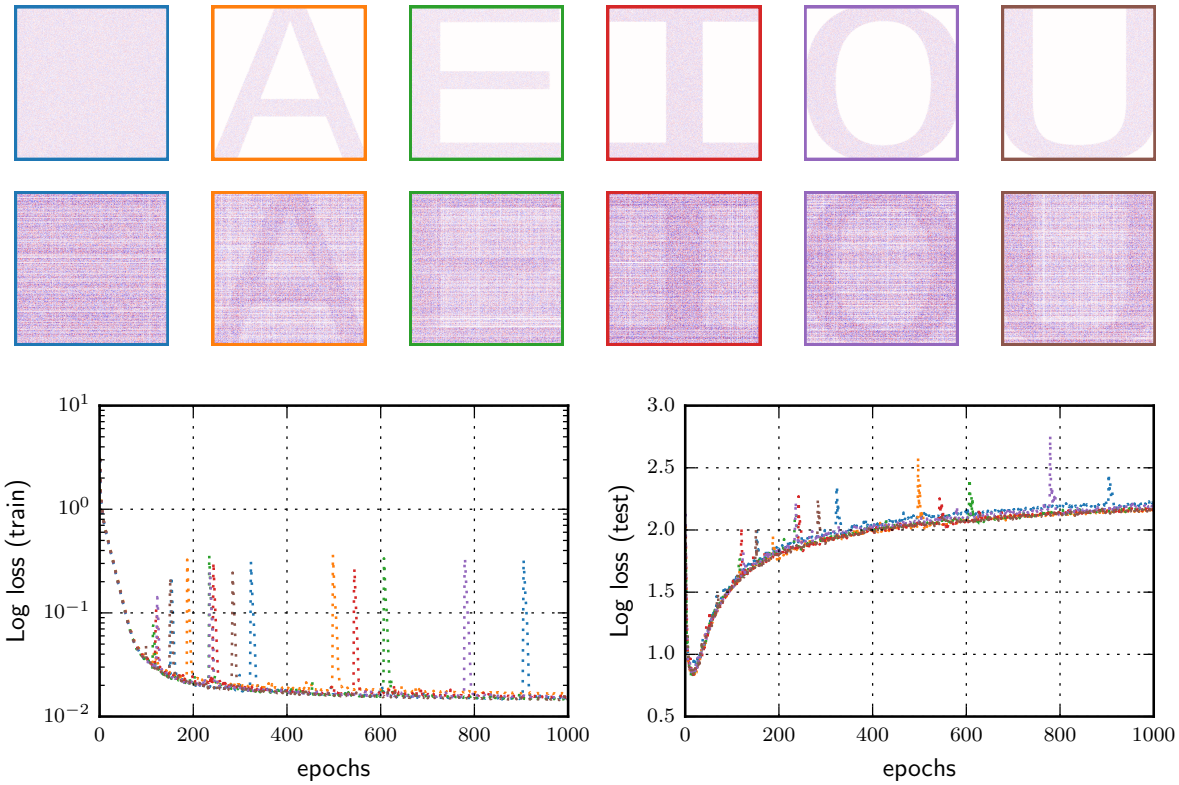


Figure 5.10: Training of marked (`A`, `E`, `I`, `O`, `U`) and unmarked (the first subplot) networks. Each network was trained for 1000 epochs with the HASYv2 dataset. The first set of plots (on the top row) show the initial configuration of weights of the middle layer of the networks trained. The same weights, after training, are plotted on the second row. Matrices showing the same mark were scaled to the same color range. The bottom subplots show the train (left) and test (right) loss measured during the training of these networks. The color of the border of each weight matrix is the same used to plot its loss function. The evolution of the network marked with `A` is shown in detail in Fig. 5.11 (similar results are obtained for the other marks).

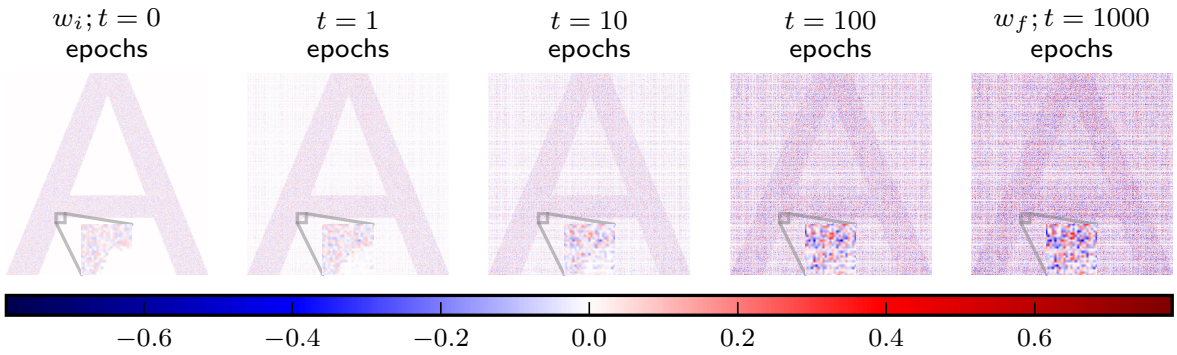


Figure 5.11: Evolution of the middle layer of weights of the network of Fig. 5.10 marked with an `A` along training. The colors of the different subplots were scaled to the same range (so that the same color corresponds to the same value in different subplots).

the weights in their spatial, meaningless organization, will not portray the coordinated updates. To our eye we will mostly see “noise”.

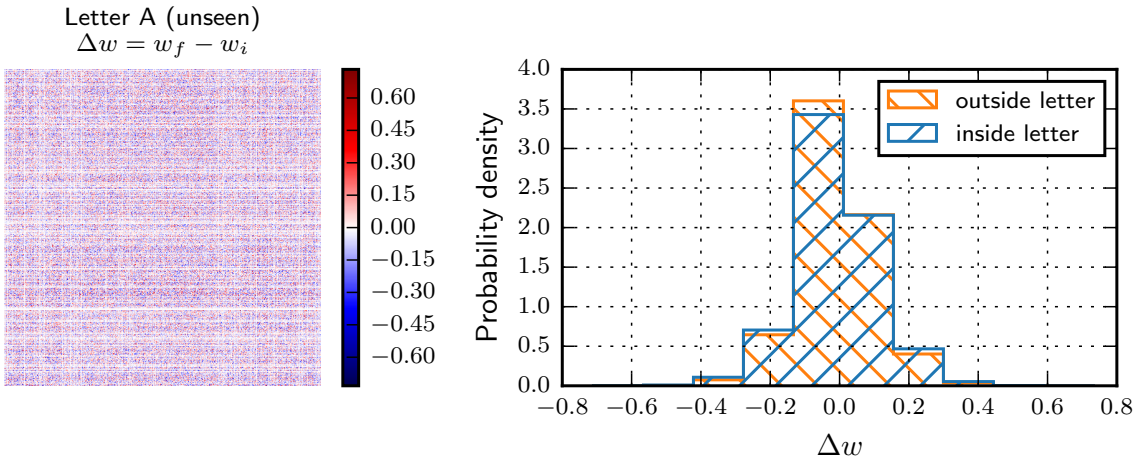


Figure 5.12: Distribution of the accumulated gradient (i.e., $\Delta w = w_f - w_i$) of the network of Fig. 5.10 marked with an `A`. The actual value of the gradient of the weights belonging to the layer marked with the letter are plotted on the left, in the usual pixel grid form, whereas their distribution is plotted on the right, separated into weights that started inside the letter and those that started outside.

We further consider the distribution of the final value of the weights of a network's layer in relation to their initial values in Fig. 5.13. The figure shows that weights that start with larger values tend to experience larger updates than the weights that start closer to zero. Moreover, the sign of the updates tends to agree with the original sign of the weights, as is observable by the tilt in the interquartile range at the plot. Furthermore, there is a large concentration of values around the line $y = x$, suggesting that a large portion of the weights are not changed at all.

These observations suggest that the initial size of a weight highly affects the scale of the updates it undergoes during training. The reason behind this may be that, due to their initial larger relative size, such weights are more likely to cause "impactful fluctuations" in the initial configuration of weights of a network. Hence, they are more likely to participate in groups of weights that are refined by the training process[5].

[5]The effect we are trying to convey is similar to the birthday effect in sports (e.g., Helsen, Van Winckel, & Williams, 2005). The birthday effect is related with the observation that the distribution of the birth month of professional athletes worldwide in sports such as soccer, baseball, ice hockey, and tennis, is skewed towards the months that make the children in a grade the oldest (typically the beginning of the calendar year, from January to March). It is the result of a cumulative effect over the course of many years. By being born at the beginning of the year, these athletes were the oldest in their classes when attending school. By being the oldest, they were typically more physically and/or emotionally developed (even if only marginally). Because of this, they were more likely to be deemed more "talented" than their younger peers, which made them receive enhanced coaching and training. Being favored (due to a factor that is largely based on chance, or, at least, to which they were oblivious) increased the gap between them and their younger colleagues, perpetuating the cycle. Over many years, this process made them into the best athletes. In essence, the effect we are observing in neural networks, caused by the fluctuations in a network's initial configuration of weights, may be very similar to the effect of the birth month in athletes — chance creates a small variation,

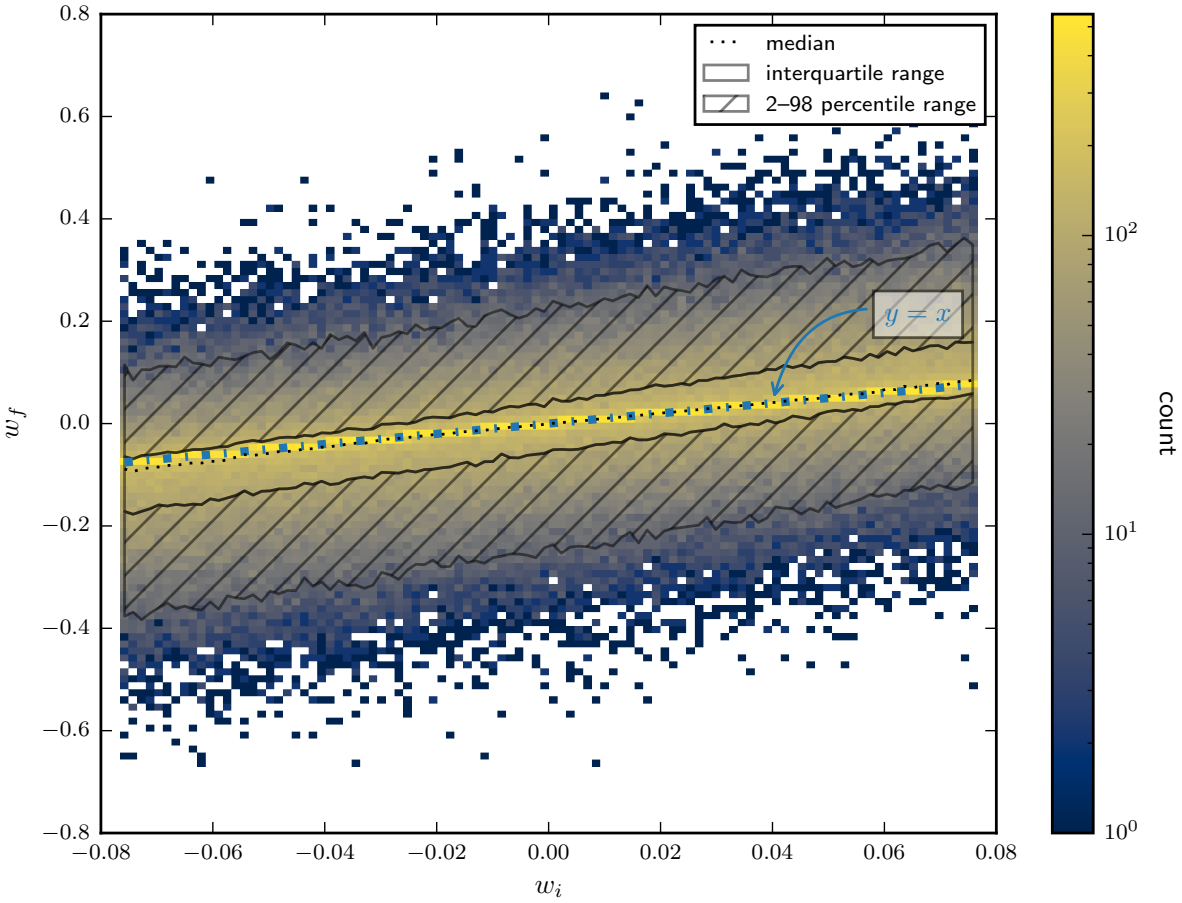


Figure 5.13: Final values of the weights of the second layer of the unmarked network of Fig. 5.10. Notice the large concentration of values around the line $y = x$, as well as the tilt in the interquartile range (initially large positive weights tend to become even larger, and likewise for negative weights).

In agreement with this idea, notice that the weights that start close to zero tend to be updated in either the positive or negative direction. This may be explained by them being "neutral" to the group they start at, causing them to be easily adjusted to positive or negative values. Finally, notice that the figure also reinforces the previous observation (in Fig. 5.7) that a large amount of the weights of a network preserve their sign along training.

5.4.2 Untrainability and loss of initialization mark

The idea of groups of weights that by chance start in good arrangements, causing them to identify particular patterns of the dataset better than other weights, also allows for an easy explanation as to why smaller networks (that are still seemingly large enough for successful training) are more difficult to train — simply, by having fewer

but it perpetuates and accumulates.

weights, the chance of such "good groups" to appear is reduced. Figure 5.14 portrays this behavior.

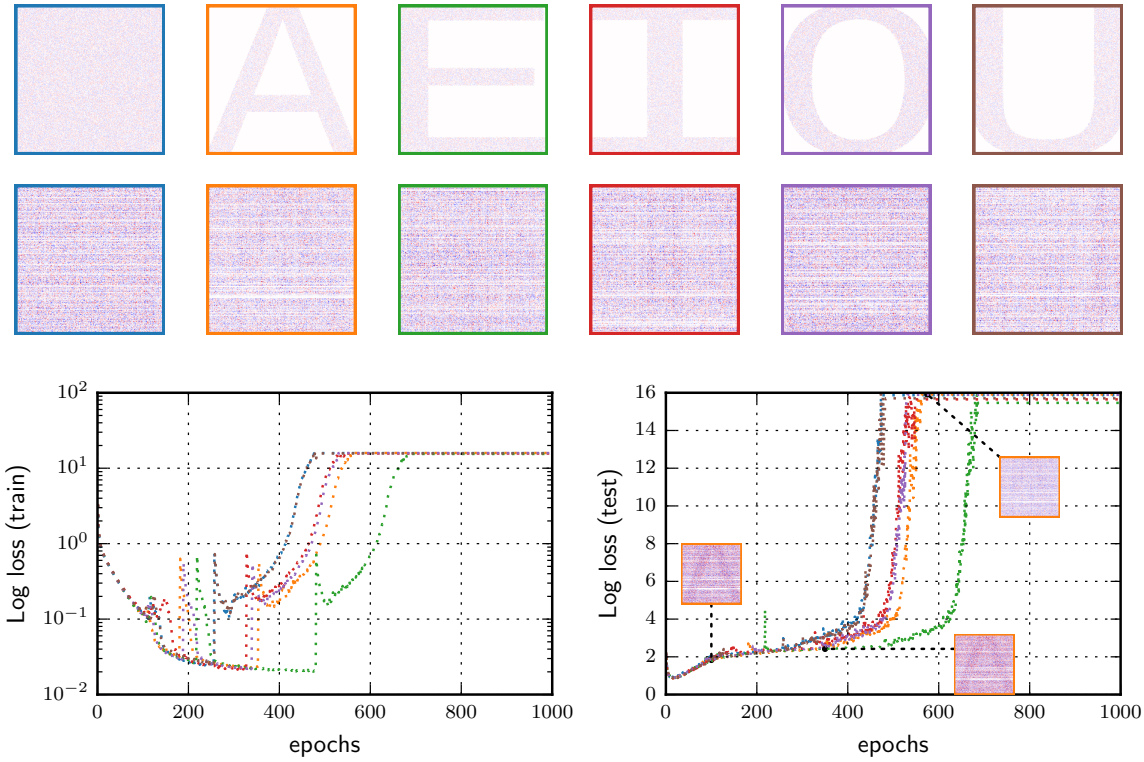


Figure 5.14: Untrainability of small neural networks marked with letters. The architecture of the networks trained is similar to the reference architecture presented in Sec. 5.1, but instead of using 512 nodes in each hidden layer, these experiments use 256. The first set of plots (on the top row) show the initial configuration of weights of the middle layer of the networks trained. The same weights, after training, are plotted on the second row. The matrices have not been scaled to the same color range, since otherwise the letters in the first row would be unrecognizable due to the scale of the weights after training. Training lasted for 1000 epochs on the HASYv2 dataset. Notice how the marks have in general disappeared when training stops, and the loss values "explode"/diverge. However, initially, all networks seem to be training successfully. Moreover, as the insets show for the letter `A`, the marks seem to disappear as the loss explodes.

The figure shows runs of marked networks, with smaller hidden layers (half the size of the reference architecture in Sec. 5.1), eventually completely "diverging", i.e., their loss function eventually explodes to a very high value and does not lower. Moreover, the final weights of the networks show almost no sign of the marks that were imprinted in their initial configuration of weights (in fact, the marks seem to disappear with the explosion of the loss). Notice that the networks are initially learning and making progress towards a minimum. Nevertheless, they at some point seem to overshoot it and completely diverge from their initial trajectory (never finding a reasonable one again).

Despite the smaller networks failing their training and their initialization mark

being lost, it may be a big leap to base these observations solely on the reduced expressiveness of the initialization (i.e., the reduced number of good arrangements of weights that may appear in the initial configuration of a network, due to the existence of less weights). Changing the architecture of a network surely causes effects on its own, which hinders our capacity to reason about our observations. To better relate the initial configuration of weights of a network to its untrainability, it would be better to incite a network to diverge, if possible, only by manipulating its initial configuration in some sensible manner, but without otherwise changing other parameters (such as its architecture). In some sense, we seek to reduce the variability (or expressiveness) of the initial configuration of a network in the least intrusive manner possible.

We achieved this by reducing the number of different sets of weights that connect the nodes of a layer to each node of the layer following. As an example, consider the first layer of weights (i.e., the one between the input layer and the first hidden layer). The reference architecture, for networks trained with HASYv2, has 1024 input nodes and 512 nodes in the first hidden layer. Thus, it contains 512 sets of weights, each consisting of the 1024 weights that connect the inputs nodes to each of the nodes of the hidden layer. To reduce the variability of this layer we copy the weights that connect the input nodes to one hidden node (i.e., a set of weights) to other hidden nodes. To reduce the variability of the configuration of weights of a network as a whole, we apply the procedure sequentially (and independently) to all its layers. The number of distinctive sets of weights that are left in each layer controls its expressiveness, since it directly affects the number of filters that may appear in the configuration, allowing us to test how the presence of filters affects the trainability of a network.

Even though the procedure does not change the theoretical representational capacity of a network in any obvious manner (meaning that backpropagation should be capable of leading networks initialized with the procedure with the same freedom it would lead "typically initialized" networks), as Fig. 5.15 (left) shows, it does affect the expressiveness of a network. By using a small number of distinctive sets of weights, the respective networks show the same divergence behavior previously observed when training smaller networks (in Fig. 5.14). Moreover, by comparing the mean absolute deviation between the configuration of weights of a network at step t (denoted by w_t) and its initial configuration (w_0), i.e., $\mathbb{E}\left[|w_t - w_0|\right]$, we can observe that converging networks drift much less away from their initial configuration than diverging networks.

Some insights may be extracted from these results. First, it seems that the "expressiveness" of a network's initial configuration of weights is instrumental to the success of its training. This expressiveness seems to be directly linked with the rudimentary filters that arise by chance in a random initialization since, by reducing their number, we are capable of inducing diverging behavior. Second, by comparing the results for the cases

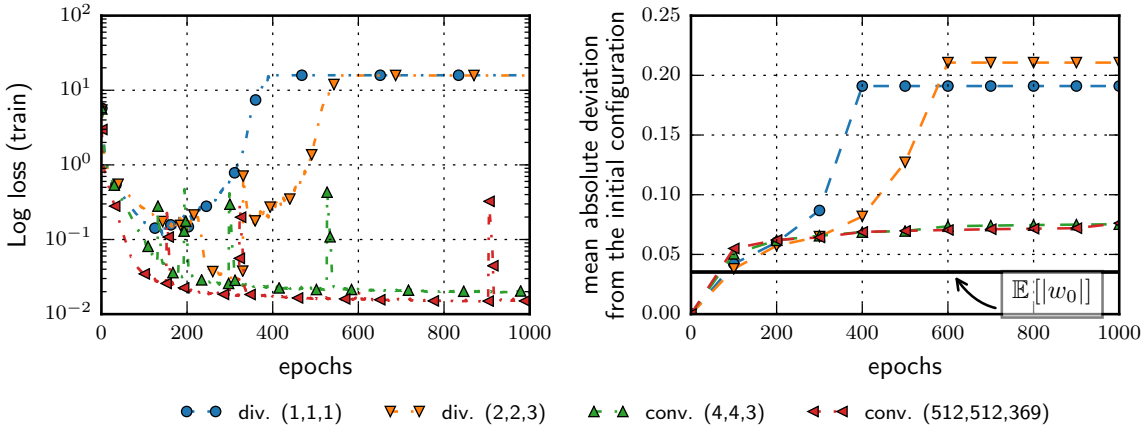


Figure 5.15: Deviation of converging and diverging networks. Training lasted for 1000 epochs on the HASYv2 dataset. The networks trained have the reference architecture described in Sec. 5.1. For clarity, a single test curve is plotted for each case, but other test runs show similar behavior. The keywords `div.` and `conv.` in the legend denote, respectively, diverging and converging behavior. The tokens `(•,•,•)` denotes the number of unique sets of weights used in each layer. Note that, in this sense, `(512,512,369)` is equivalent to a "fully random" initialization.

`(2,2,3)` and `(4,4,3)` (the former showing divergence, wheres the latter convergence), we can conclude that the expressiveness required in an initial configuration is easily achieved with a random initialization strategy — in our experiments, as little as 4 sets per layer are sufficient to ensure the required variability. Moreover, even very restricted initializations (like the case `(4,4,3)`) show the same mean deviation from their initial configuration as do completely random initializations (the case `(512,512,369)`), further suggesting that training is, to a large degree, a fine-tuning process. Finally, it is interesting to notice that the mean absolute deviation "explodes" at the same time that the loss does.

5.5 The filter initialization

So far we have seen that there seem to be weights that are initialized in ways that make them preferential for specific patterns present in a dataset. One may conceive these groups as rough "filters" for particular features of the dataset, that the learning process accentuates. Since these filters, being created by chance, seem to be a crucial aspect for the success in the training of a network, as we saw previously, perhaps their creation could be promoted deliberately in the initial configuration of weights by some guided process. In essence, one could hope that by initializing a network with filters somewhat more tuned towards the dataset that they will be used with, the networks will train better.

This section presents a simple (and somewhat "naive") strategy that seeks to achieve this. It turns out that our strategy significantly improves training. However, it often worsens generalization. While not useful for all scenarios due to its lack of generalization, it suggests that initialization strategies that promote the creation of such filters may be worth further considering.

The first question to contemplate is how to introduce meaningful features in the initial configuration of weights. One possible idea is to use samples from the dataset itself, particularly the training set or an "initialization set" (used solely with the purpose of creating the filters), to somehow adjust the initial weights in a sensible way. Notice that the difference between using the training set or an alternative third set, the initialization set, is that in the first case, the samples used in the creation of the filters are also used for training, whereas, in the second case, they are not used further. We will be considering these two alternatives below. In the future, it may be interesting to test the usage of completely uncorrelated datasets, created using for instance geometric shapes or parts of plots of equations.

We will concentrating on initializing the first layer of weights alone, since this is the layer interacting directly with input samples (which, due to this, is more simple to reason about). The remaining layers are initialized with Glorot's uniform initialization. However, the procedure could be extended to initialize deeper layers, or, alternatively, they could be initialized with the dense sliced initialization of Sec. 5.2.2.

The strategy is based on initializing the set of weights of each node of the first hidden layer of a network by overlapping portions of input images on top of each other and adding white noise to the result. More specifically, we start by assigning each node of the first hidden layer an output class (e.g., in the case of the MNIST dataset, each node would be assigned one digit from `0` to `9`). Then, the weights connecting each of these nodes to the input nodes (i.e., its set of weights) are reshaped into a matrix the size of the input images, to mimic an actual pixel grid (this step is not necessary, it is used just to simplify the explanation of how the procedure works). The resulting matrix is divided into blocks of weights (for instance, 9×9 blocks), and the weights belonging to each of these blocks are initialized independently by overlapping samples of the training/initialization set of the output class chosen for the respective hidden node. The overlapping is performed by summing the samples chosen, after they have been centered and normalized (i.e., instead of using directly an input sample x, we use its z-score, $z = (x - \mu_x)/\sigma_x$). After overlapping all the desired samples, the result is scaled and some uniform noise added to it, to increase the variability of the initialization. Through this scaling and noise addition, one may adjust the statistical properties of the initial configuration of weights to accommodate the heuristics described in Sec. 2.5. We have carried this last step to achieve a variance in the initial configuration of weights

as described in Glorot's initialization (Sec. 2.5.1), i.e., $\mathrm{Var}[w] = 2/(m+n)$ (where m and n denote the inputs and outputs of the layer of weights w, respectively). We do this by weighting the variance of the weights resulting solely from overlapping samples with the variance of the noise that we will be adding, so that when summed they equal Glorot's variance (in this sense, the larger the weight given to the variance resulting solely from the dataset, the smaller the scale of the noise that is added). Algorithm 5.3 presents pseudocode for this procedure.

Algorithm 5.3: Pseudocode for the filter initialization

Data: A matrix of weights $w_{m\times n}$. A set of input (x) and output (y) samples to use for the initialization. A number gs specifying the grid size to use as in $gs\times gs$. A number ns specifying the no. samples to overlap per each block. The weight γ to give to the variance that results from overlapping samples of the dataset.

Result: w is initialized according to the filter initialization with the given parameters.

$w \leftarrow 0$ ▷ Initialize all entries in w to zero

for $j \leftarrow 1$ *to* n *do*

  $c \leftarrow \text{choose}(y)$ ▷ Select a class c among the output classes in y

  $s \leftarrow \text{reshape}(w_{\bullet j}, x)$ ▷ Select the submatrix of weights s that connects the input nodes 1 to m to node j, and reshape it to the same shape of the input samples

  $B \leftarrow \text{divide-into-grid}(s, gs)$ ▷ Divide s into a set of $gs\times gs$ groups of weights (the blocks of weights where input samples will be overlapped)

  foreach $b \in B$ *do*

    $S \leftarrow \text{sample}(x, c, ns)$ ▷ Choose a set of ns input samples chosen at random and belonging to class c

    foreach $s \in S$ *do*

      $b \leftarrow b + (s - \mu_s)/\sigma_s$ ▷ Normalize s and accumulate the result in b

    end

  end

end

$\Gamma \leftarrow 2/(m+n)$ ▷ Compute the goal variance Γ of Glorot's initialization

$w \leftarrow w \cdot \sqrt{\gamma \cdot \Gamma / \mathrm{Var}[w]}$ ▷ Scale w so that $\mathrm{Var}[w] = \gamma \cdot \Gamma$

$u \sim U\left(-\sqrt{3\,(1-\gamma)\cdot\Gamma},\ \sqrt{3\,(1-\gamma)\cdot\Gamma}\right)$ ▷ Draw random samples $u_{m\times n}$ from a uniform distribution so that $\mathrm{Var}[u] = (1-\gamma)\cdot\Gamma$

$w \leftarrow w + u$ ▷ Add u to w (after this, $\mathrm{Var}[w] = \Gamma$)

This initialization strategy introduces a few parameters to consider: the shape of the grid, the number of overlapped samples, and the scale of the noise added. Recall that the overall idea is to facilitate the emergence of the "initialization filters", while still maintaining variability in the initial configuration. All these parameters try to help

achieving this goal in complementary ways. Overlapping samples allows the creation of configurations with a broader capacity of recognizing important patterns of a class of the dataset than using a single sample would. However, by overlapping too many samples, we could overload the configuration (the filters could become "sluggish"), worsening them. Employing a grid of blocks and overlapping samples separately into each of the blocks helps to mitigate this problem, since more samples are used in constructing the filter, as desired, but they are not all overlapped on top of each other — they are distributed through separate regions of the set of weights. Finally, introducing noise creates small and unpredictable irregularities (i.e., variability), which should help increasing the spectrum of applicability of the filters. Figure 5.16 provides examples of the initialization filters created by this procedure.

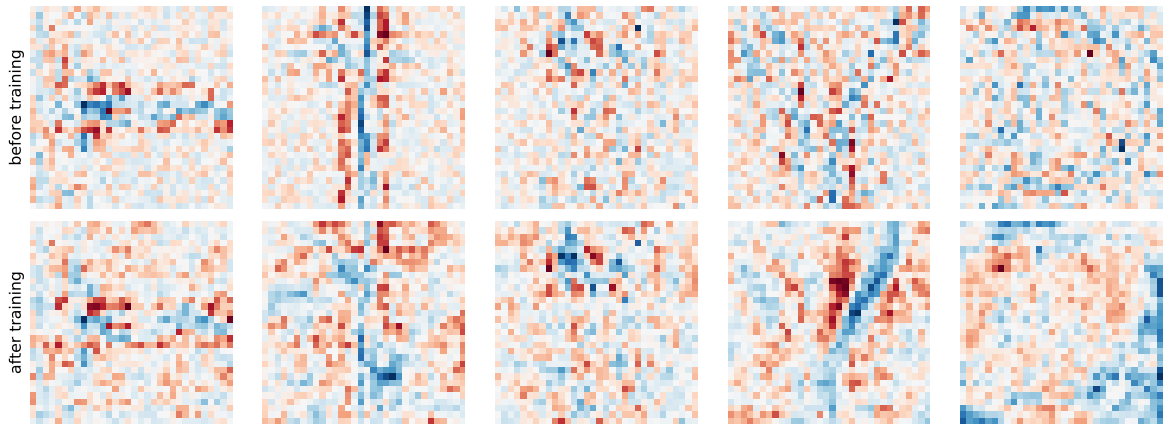


Figure 5.16: Examples of initialization filters created by the procedure described in this section (best viewed in color). Training lasted 100 epochs on the HASYv2 dataset. The filters resulting from the procedure are shown in the first row, while their state after training in the second. To improve visibility, the color of the images was not normalized to the same range.

The figure shows that the filters created by this strategy are, in general, maintained during the training of the network. They are visible (and frequently even enhanced) once training is stopped, making it seem that the procedure provides the learning process with a jump-start by better arranging its initial configuration. The positive effect of our strategy can be confirmed in Fig. 5.17, where a number of combinations of its parameters are tested and compared against the training of networks initialized (solely) with Glorot's uniform initialization.

The figure shows several meaningful trends. First, increasing both the grid size and the number of overlapped samples seems to improve training. Meanwhile, the weight of the variance that is retained from the dataset seems to cause little changes (note that a smaller weight implies adding more noise to compensate, to achieve the variance desired by Glorot's initialization). In general, most of the combinations tested improve training, sometimes to below half the error obtained with the baseline initialization

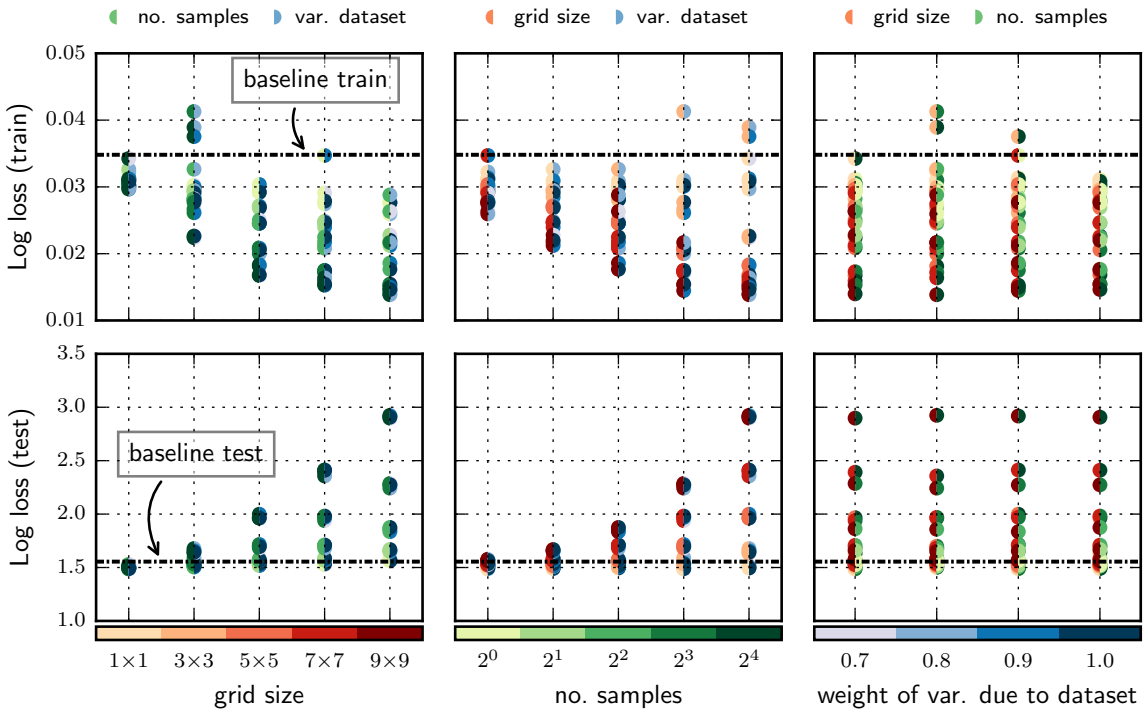


Figure 5.17: Results of training networks initialized with the Filter initialization, for a combination of parameters. In each subplot, a single parameter is varied in the x-axis. The color codes of the markers denote the variations of the remaining parameters (the respective color codes are displayed in the color bars at the bottom of the figure). Each result shown represents the mean of five independent test runs. Training was carried on the HASYv2 dataset and lasted 100 epochs. The baseline results are obtained using Glorot's uniform initialization. Recall that the weight of the variance of the dataset refers to the weight given to the variance obtained solely from overlapping samples to reach the variance of Glorot's initialization (which means that the larger it is, the smaller the scale of the noise added).

strategy. However, interestingly, the exact opposite results are observed in the test set. There, it seems that increasing the grid size and number of samples overlapped worsens generalization (the weight of the variance from the dataset still does not seem to affect training significantly). Moreover, very few results actually achieve a test loss below the baseline, and those that do, do it only marginally.

These results suggest that the motivation behind this initialization strategy — that perhaps by creating more suited filters one could aim for better training — was, perhaps, too correct! By employing such filters, the networks do train better, but sadly these gains can only be harvested for specific purposes, since the strategy spoils generalization. Still, one should not overlook the meaning of these results. They support the idea that, to a great extent, the learning algorithm mostly refines the solution created by the initialization. This suffices for training because random initializations are rich enough to contain suitable arrangements of weights that only need the fine-tuning provided by the optimizer to be made useful.

One possible explanation for the lack of generalization of the configurations created by our approach is that the filters already start too specific to the dataset used for training, due to them being created from the training set. Owing to this possibility, we now consider the alternative discussed previously, based on using samples from a third dataset, disjoint from the training and testing sets, and to which we call the initialization set. This approach should be worse in the training set, but better in the testing one, since the samples used for the creation of the filters are not used in training. This is what happens, as observed in Fig. 5.18.

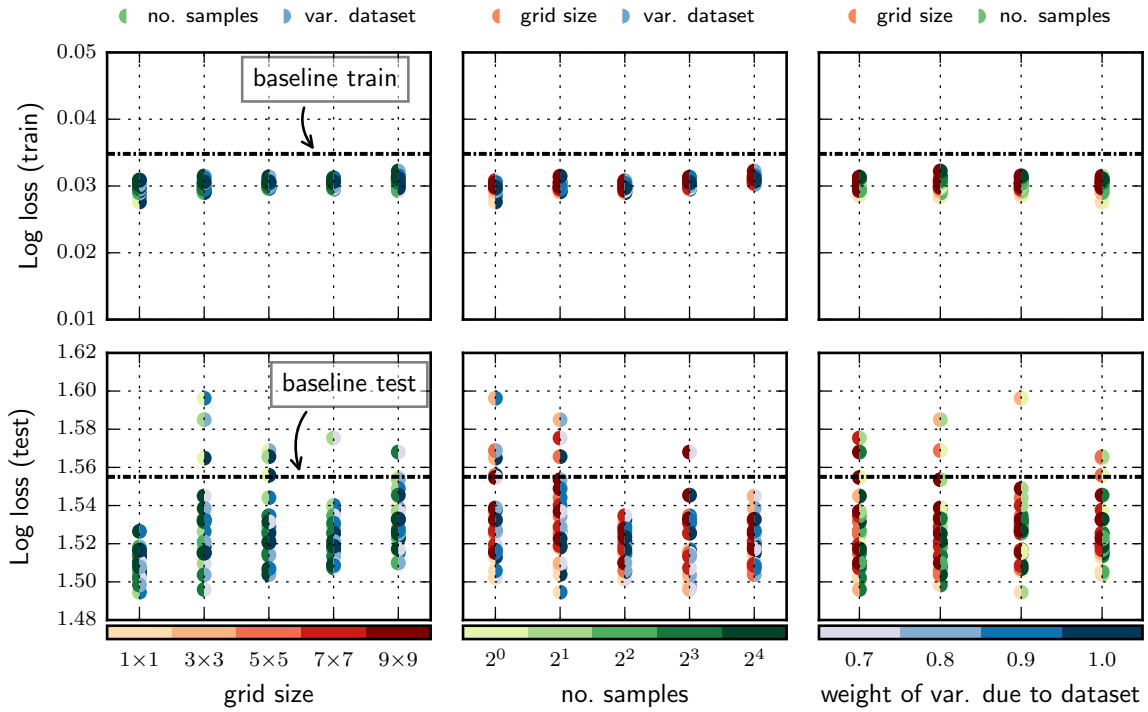


Figure 5.18: Results of training networks initialized with the Filter initialization, for a combination of parameters, and using a separate dataset solely for initialization purposes. The meaning of the figure (e.g., the legends and color codes) are the same as for Fig. 5.18.

The figure shows that using a separate initialization dataset typically still leads to lower train loss than the baseline initialization strategy for almost all combinations of parameters tested. However, as expected, the improvement is much less pronounced than it was when using the training set for creating the filters. Meanwhile, and also as expected, the use of a separate dataset does help to improve generalization. Most of the combinations of parameters tested lead to better test losses than the baseline initialization strategy at the end of training. Moreover, it seems that using more samples in the procedure and smaller grid sizes leads to better generalization.

Despite the improvements to the test loss, this method still presents limitations to its usefulness in practice. The loss we are observing is at the end of training, once

overfitting is already in effect. As Fig. 5.19 shows, the minimum of the test loss (between epochs 10–20) is still lower for a fully random initialization.

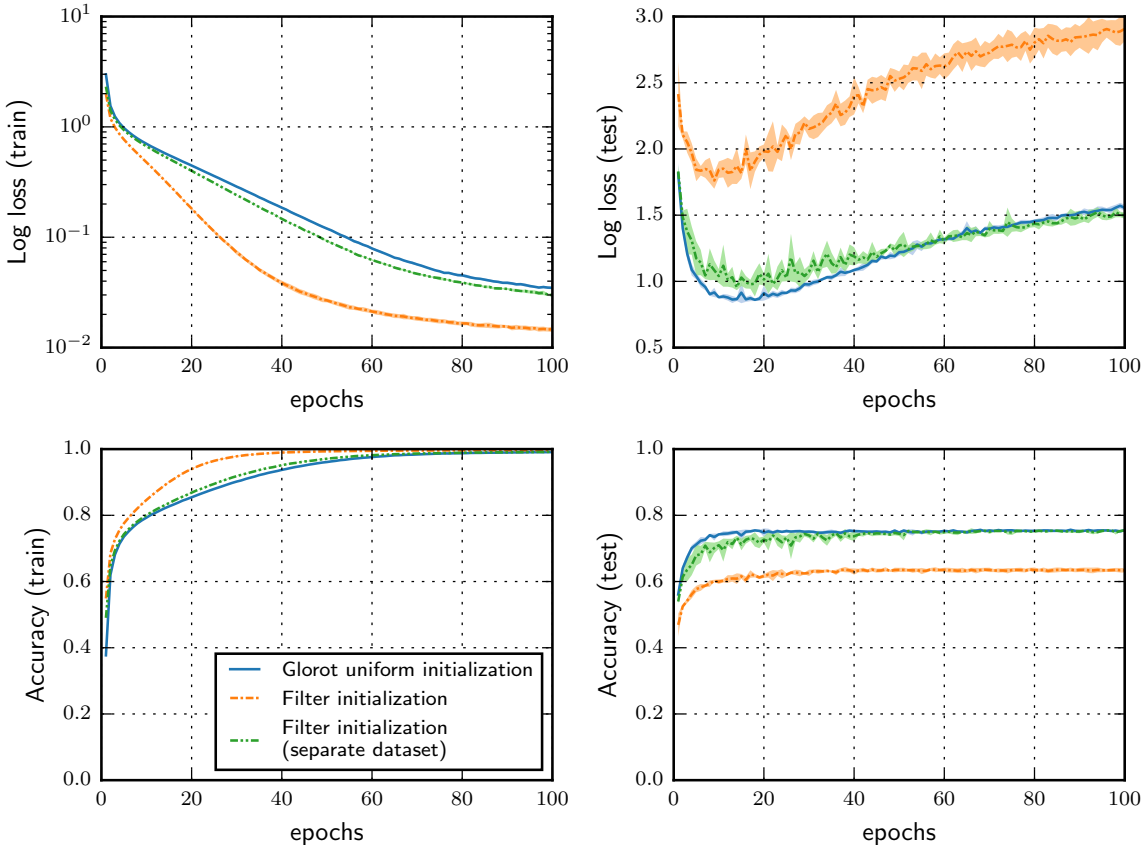


Figure 5.19: Train and test errors of networks initialized with the filter initialization along training. Training lasted 100 epochs. The mean (solid line) and standard deviation (colored area) of 5 test runs are plotted. The parameters of the filter initialization using the training set are a grid size of 9×9, 16 overlapped samples, and no noise added. The parameters using a separate dataset are a grid size of 3×3, 8 overlapped samples, and a weight of 0.7 to the variance resulting from the dataset.

Overall, the strategy shows the predicted behavior, which indicates we may be on the right track to understanding this training phenomenon. Perhaps an approach in-between, through finding a better compromise between the benefits of targeted filters and the generality of randomly created ones, could provide for a more pronounced boost on both train and (especially) test datasets, prior to overfitting. For now, it is only possible to conjecture why patterns created by a random initialization seem to provide more general solutions than building more appropriate filters from the dataset. The reason behind this may simply be that random initializations are entirely fair, in that they have no knowledge of any particular dataset or even feature. For instance, consider that when creating filters from the dataset, more frequent patterns will contribute with heavier weight (due to appearing more times) to the filter. This does not happen with

a random initialization, which will originate suitable arrangements of weights for all the patterns with mostly equal probability.

Until these processes and mechanisms are better understood, it is difficult to develop strategies that offer meaningful improvements consistently (in a large number of cases). On a brighter side, they are fundamental, in that they govern the way neural networks evolve (i.e., learn) and behave. Hence, understanding such phenomena should open doors to profound improvements, or at least offer closure concerning what can and cannot be achieved with these models.

6 Discussion and Conclusions

We may hope that machines will eventually compete with men in all purely intellectual fields. But which are the best ones to start with?

— Alan Turing

In this thesis, we took a glance at the insides of artificial neural networks. We relied on a toy problem to gain an intuition about the processes and mechanisms residing inside neural networks. We achieved this by experimenting with different parameters and confronting the results (e.g., the evolution of the loss function and the configuration of weights) against expectations based on a reference model. We developed and applied methods for interpreting the configurations of weights of networks, seeking to understand their arrangements and function. Using these techniques, we found that the configuration of weights of trained neural networks was highly linked with the dataset we used to train the networks. The weights showed correlations found originally in the dataset alone. We also found that, when solving the same problem, similar (but not equal) neural network architectures may lead to fundamentally different configurations of weights. This behavior means that the nodes in one architecture may take fundamentally different roles in another architecture, even if the two architectures are similar. The simple modification of the activation function of one layer is capable of causing these variations. Finally, still within the scope of this chapter, we observed that the initial configuration of weights is capable of conditioning the whole training of a network. It seems that, in some problems, the quality of the minimum to which networks converge may be highly connected to the point from where networks start their training.

Motivated by these observations, we took a closer look into the effects the initial configuration of weights seems to have in the training and function of neural networks. These tests were performed with the real-world datasets MNIST and HASYv2. We

started by experimenting with an initialization strategy inspired by the Lightning initialization (Pircher et al., 2018), which we found to be highly dependent on the parameters it uses. Looking for a less intrusive initialization strategy, we devised the Dense Sliced initialization, which, to some extent, is a compromise between our lightning initialization and a classic, dense initialization. It showed interesting results on MNIST. During its development, it also revealed intriguing trends of the learning process — the initial configuration of weights left recognizable traces on the final configuration acquired by trained networks. We studied this observation. We started by noting that if one initializes the first layer of weights sparsely (i.e., where most weights apart from a small fraction are set to zero), the initially nonzero weights seem to influence the weights close to them. Typically, the initially zero weights take the sign of the nonzero weights adjacent to them. Afterward, we saw that if we mark the initial configuration of weights of a network with some symbols (we did this with letters) and then train the network, the symbols are typically clearly visible, even when one trains them for as many as a thousand epochs. It seems that this is a necessary condition for successful training, since if the mark does disappear, then the network does not train successfully. Based on these findings and our understanding of the reasons behind them, we developed the Filter initialization, which is based on increasing the number of filters arising in the initial configuration of weights of a network. The technique acts according to our expectations in that it improves training; however, at the same time, it worsens generalization, limiting its applicability to a restricted set of scenarios.

6.1 Future work

The work produced in this thesis was vast, yet, due to its nature, no part of it is ultimately finished. Instead, we believe we opened a few avenues that are worth further investigation in the future. We believe that the work performed with the toy problem, in Chap. 4, was fundamental to equip us with a more profound and stronger intuition regarding the behavior of neural networks. Empowered with it, we decided to study further the effect the initial configuration of weights has on neural networks. We believe that the initial study we carried to understand the meaning of the configuration of weights of a network and how it related to the toy dataset could be instrumental in revealing meaningful connections between the dataset used to train a network and the configuration it acquires through training. Perhaps it could prove useful in areas such as XAI, where it could be used to explain how and why the weights of trained networks reach a particular configuration — which would, in turn, help explain why a particular model behaves in a certain way.

We also believe it is worth to continue the research about the effects of the initial configuration of weights of a network. We intend to do this by developing a way of quantifying the trace left by the initial configuration during training. After this is done, we wish to test different settings and determine precisely the cause behind our observations (and the extent to which they verify). If these experiments are successfully carried, they may shed some light on the importance of the initialization of a network, why it is so vital, and its effects on generalization — and based on this knowledge, new and improved initialization methods may be conceived.

Finally, both the Dense sliced initialization (Sec. 5.2) and the Filter initialization (Sec. 5.5) may be worth more investigation. On the one hand, the former method showed compelling initial results, but the efforts made in it were put on hold due to us focusing on other ideas. However, we believe it is an idea that may provide attractive results if further investigated, with more datasets and different architectures. One the other hand, the latter method, the Filter initialization, seemed to improve training but simultaneously worsened generalization. It may be productive to understand why. If one can determine the cause behind the reduction in generalization, perhaps one can apply it in reverse to obtain the opposite effect — possibly worse training but with better generalization.

6.2 Final considerations

It is undeniable that machine learning, and particularly deep learning, has come a long way since its infancy. It has become a field truly tough to tame, a behemoth, yet one that unquestionably affects our lives — hopefully, usually for the better. Even though it increasingly drives more vital parts of our lives — from governing healthcare or arbitrating credit eligibility to influencing elections — our knowledge and understanding about the methods that are being used are still profoundly lacking. Luckily, our curiosity and need-to-know-why are never satisfied — they keep pushing us toward trying to learn the root causes and the simple explanations behind our real-world experiences and observations.

In the end, we feel like we opened more doors than the ones we closed. However, we believe this only means we improved our chances of uncovering interesting and important features of the dynamics of neural networks — much like the meaningful weights of an initial configuration in Chap. 5, we only need a handful of these doors to lead us in a good trajectory.

References

Ackley, D. H., Hinton, G. E., & Sejnowski, T. J. (1985). A learning algorithm for boltzmann machines. *Cognitive science*, *9*(1), 147–169.

Assael, Y., Sommerschield, T., & Prag, J. (2019). Restoring ancient text using deep learning: A case study on greek epigraphy. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (emnlp-ijcnlp)* (pp. 6369–6376).

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2018). Automatic differentiation in machine learning: A survey. *Journal of machine learning research*, *18*(153).

Bengio, Y., Lamblin, P., Popovici, D., & Larochelle, H. (2007). Greedy layer-wise training of deep networks. In *Advances in neural information processing systems* (pp. 153–160).

Bottou, L. [Léon], Curtis, F. E., & Nocedal, J. (2018). Optimization methods for large-scale machine learning. *Siam Review*, *60*(2), 223–311.

Breen, P. G., Foley, C. N., Boekholt, T., & Zwart, S. P. (2019). Newton vs the machine: Solving the chaotic three-body problem using deep neural networks. *arXiv preprint arXiv:1910.07291*.

Buckner, C., & Garson, J. (2019). Connectionism. In E. N. Zalta (Ed.), *The stanford encyclopedia of philosophy* (Fall 2019). Metaphysics Research Lab, Stanford University.

Canziani, A., Paszke, A., & Culurciello, E. (2016). An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*.

Chapelle, O., & Erhan, D. (2011). Improved preconditioner for hessian free optimization. In *Nips workshop on deep learning and unsupervised feature learning* (Vol. 201, *1*).

Chollet, F. et al. (2015). Keras. https://keras.io. Accessed on 2019-11-04.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., & Kuksa, P. (2011). Natural language processing (almost) from scratch. *Journal of machine learning research*, *12*(Aug), 2493–2537.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, *2*(4), 303–314.

Darken, C., & Moody, J. (1992). Towards faster stochastic gradient search. In *Advances in neural information processing systems* (pp. 1009–1016).

Dean, J. (2019). The deep learning revolution and its implications for computer architecture and chip design. *arXiv preprint arXiv:1911.05289*.

Deniz, C. M., Xiang, S., Hallyburton, R. S., Welbeck, A., Babb, J. S., Honig, S., ... Chang, G. (2018). Segmentation of the proximal femur from mr images using deep convolutional neural networks. *Scientific reports*, *8*(1), 16485.

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, *12*(Jul), 2121–2159.

Durbin, R., & Rumelhart, D. E. (1989). Product units: A computationally powerful and biologically plausible extension to backpropagation networks. *Neural computation*, *1*(1), 133–142.

Frankle, J., & Carbin, M. (2019). The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International conference on learning representations*. Retrieved from https://openreview.net/forum?id=rJl-b3RcF7

Funahashi, K.-I. (1989). On the approximate realization of continuous mappings by neural networks. *Neural networks*, *2*(3), 183–192.

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (pp. 249–256).

Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics* (pp. 315–323).

Goh, G. (2017). Why momentum really works. *Distill*. doi:10.23915/distill.00006

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT press.

Grother, P. J. (1995). Nist special database 19. *Handprinted forms and characters database, National Institute of Standards and Technology*.

Güler, R. A., Neverova, N., & Kokkinos, I. (2018). Densepose: Dense human pose estimation in the wild. In *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 7297–7306).

Gundersen, O. E., & Kjensmo, S. (2018). State of the art: Reproducibility in artificial intelligence. In *Thirty-second aaai conference on artificial intelligence*.

Han, S., Pool, J., Tran, J., & Dally, W. (2015). Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems* (pp. 1135–1143).

Hanin, B., & Sellke, M. (2017). Approximating continuous functions by relu nets of minimal width. *arXiv preprint arXiv:1710.11278*.

Hassibi, B., & Stork, D. G. (1993). Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in neural information processing systems* (pp. 164–171).

He, K., Gkioxari, G., Dollár, P., & Girshick, R. (2017). Mask r-cnn. In *Proceedings of the ieee international conference on computer vision* (pp. 2961–2969).

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the ieee international conference on computer vision* (pp. 1026–1034).

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 770–778).

Hebb, D. (1949). The organization of behavior: A neuropsychological theory.

Helsen, W. F., Van Winckel, J., & Williams, A. M. (2005). The relative age effect in youth soccer across europe. *Journal of sports sciences*, *23*(6), 629–636.

Hennessy, J. L., & Patterson, D. A. (2017). *Computer architecture: A quantitative approach* (6th). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

High-Level Expert Group on AI. (2019). *Ethics guidelines for trustworthy ai*. European Commission. Accessed on 2019-11-24. Retrieved from https://ec.europa.eu/digital-single-market/en/news/ethics-guidelines-trustworthy-ai

Hinton, G. E., Osindero, S., & Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural computation*, *18*(7), 1527–1554.

Hinton, G. E., & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *science*, *313*(5786), 504–507.

Hinton, G., Srivastava, N., & Swersky, K. (2012). Lecture 6e. rmsprop: Divide the gradient by a running average of its recent magnitude. Accessed on 2019-11-04. Retrieved from https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

Hinton, G., Vinyals, O., & Dean, J. (2015). Distilling the knowledge in a neural network. In *Nips deep learning and representation learning workshop*. Retrieved from http://arxiv.org/abs/1503.02531

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, *79*(8), 2554–2558.

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, *2*(5), 359–366.

Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). Densely connected convolutional networks. In *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 4700–4708).

Hutson, M. (2018a). Artificial intelligence faces reproducibility crisis. American Association for the Advancement of Science.

Hutson, M. (2018b). Has artificial intelligence become alchemy? American Association for the Advancement of Science.

Jean, S., Cho, K., Memisevic, R., & Bengio, Y. (2015). On using very large target vocabulary for neural machine translation. In *Proceedings of the 53rd annual meeting of the association for computational linguistics and the 7th international joint conference on natural language processing (volume 1: Long papers)* (pp. 1–10).

Karpathy, A. (2014). What i learned from competing against a convnet on imagenet. Accessed on 2019-11-26. Retrieved from https://karpathy.github.io/2014/09/02/what-i-learned-from-competing-against-a-convnet-on-imagenet/

Kasparov, G. (2010). The chess master and the computer. *The New York Review of Books*, *57*(2), 16–19.

Kawaguchi, K., Huang, J., & Kaelbling, L. P. (2019). Effect of depth and width on local minima in deep learning. *Neural computation*, *31*(7), 1462–1498.

Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. In *3rd international conference on learning representations, ICLR 2015, san diego, ca, usa, may 7-9, 2015, conference track proceedings*. Retrieved from http://arxiv.org/abs/1412.6980

Knuth, D. E. (1986). *The texbook*. Addison-Wesley Professional.

Kolmogorov, A. N. (1957). On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition. In *Doklady akademii nauk* (Vol. 114, *5*, pp. 953–956). Russian Academy of Sciences.

Koren, Y. (2009). The bellkor solution to the netflix grand prize. https://www.netflixprize.com/assets/GrandPrize2009_BPC_BellKor.pdf. Accessed on 2019-11-26.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097–1105).

Kurzweil, R. (1990). *The age of intelligent machines*. MIT Press.

LeCun, Y. (2019). The epistemology of deep learning. Accessed on 2019-11-12. Retrieved from https://video.ias.edu/sites/video/files/lecun-ias-20190222.pdf

LeCun, Y. et al. (1989). Generalization and network design strategies. In *Connectionism in perspective* (Vol. 19). Citeseer.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *nature*, *521*(7553), 436.

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

LeCun, Y., Bottou, L. [Léon], Bengio, Y., Haffner, P., et al. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2324.

LeCun, Y., Bottou, L. [Leon], Orr, G. B., & Müller, K.-R. (1998). Efficient backprop. In *Neural networks: Tricks of the trade* (pp. 9–50). Springer.

LeCun, Y., Denker, J. S., & Solla, S. A. (1990). Optimal brain damage. In *Advances in neural information processing systems* (pp. 598–605).

Licklider, J. C. R. (1960). Man-computer symbiosis. *IRE transactions on human factors in electronics*, (1), 4–11.

Lloyd, S. (1982). Least squares quantization in pcm. *IEEE transactions on information theory*, *28*(2), 129–137.

Martens, J. (2010). Deep learning via hessian-free optimization. In *Icml* (Vol. 27, pp. 735–742).

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, ... Xiaoqiang Zheng. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org. Retrieved from https://www.tensorflow.org/

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*(4), 115–133.

Minsky, M. (1988). *Society of mind*. Simon and Schuster.

Minsky, M. L., & Papert, S. A. (1988). *Perceptrons: Expanded edition*. MIT press.

Minsky, M., & Papert, S. (1969). *Perceptrons: An introduction to computational geometry*. The MIT Press.

Moravec, H. (1988). *Mind children: The future of robot and human intelligence*. Harvard University Press.

Moravec, H. (1998). When will computer hardware match the human brain. *Journal of evolution and technology*, *1*(1), 10.

Neubig, G., Dyer, C., Goldberg, Y., Matthews, A., Ammar, W., Anastasopoulos, A., ... Yin, P. (2017). Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*.

Newell, A. (1969). Perceptrons. an introduction to computational geometry. *Science*, *165*(3895), 780–782. doi:10.1126/science.165.3895.780

Nilsson, N. J. (2009). *The quest for artificial intelligence*. Cambridge University Press.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., ... Lerer, A. (2017). Automatic differentiation in PyTorch. In *Nips autodiff workshop*.

Pircher, T., Haspel, D., & Schlücker, E. (2018). Dense neural networks as sparse graphs and the lightning initialization. *arXiv preprint arXiv:1809.08836*.

Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, *4*(5), 1–17.

Raina, R., Madhavan, A., & Ng, A. Y. (2009). Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning* (pp. 873–880). ACM.

Ramachandran, P., Zoph, B., & Le, Q. V. (2018). Searching for activation functions. In *6th international conference on learning representations, ICLR 2018, vancouver, bc, canada, april 30 - may 3, 2018, workshop track proceedings*. Retrieved from https://openreview.net/forum?id=Hkuq2EkPf

Rasskin-Gutman, D. (2009). *Chess metaphors: Artificial intelligence and the human mind*. MIT Press.

Rosenblatt, F. (1957). *The perceptron, a perceiving and recognizing automaton project para*. Cornell Aeronautical Laboratory.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, *65*(6), 386.

Rumelhart, D. E., & Hintonf, G. E. (1986). Learning representations by back-propagating errors. *NATURE*, *323*, 9.

Rutishauser, H. (1959). Theory of gradient methods. In *Refined iterative methods for computation of the solution and the eigenvalues of self-adjoint boundary value problems* (pp. 24–49). Springer.

Salakhutdinov, R., Mnih, A., & Hinton, G. (2007). Restricted boltzmann machines for collaborative filtering. In *Proceedings of the 24th international conference on machine learning* (pp. 791–798). ACM.

Saxe, A. M., McClelland, J. L., & Ganguli, S. (2014). Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. In *2nd international conference on learning representations, ICLR 2014, banff, ab, canada, april 14-16, 2014, conference track proceedings*. Retrieved from http://arxiv.org/abs/1312.6120

Sculley, D., Snoek, J., Wiltschko, A. B., & Rahimi, A. (2018). Winner's curse? on pace, progress, and empirical rigor. In *6th international conference on learning representations, ICLR 2018, vancouver, bc, canada, april 30 - may 3, 2018, workshop track proceedings*. Retrieved from https://openreview.net/forum?id=rJWF0Fywf

Shrestha, A., & Mahmood, A. (2019). Review of deep learning algorithms and architectures. *IEEE Access*, *7*, 53040–53065.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... Bolton, A., et al. (2017). Mastering the game of go without human knowledge. *Nature*, *550*(7676), 354.

Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Sun, Y. (2015). Notes on first-order methods for minimizing smooth functions. Retrieved from https://web.stanford.edu/class/msande318/notes/notes-first-order-smooth.pdf

Sutskever, I., Vinyals, O., & Le, Q. (2014). Sequence to sequence learning with neural networks. *Advances in NIPS*.

Sutskever, I. [Ilya], Martens, J., Dahl, G., & Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *International conference on machine learning* (pp. 1139–1147).

Sutton, R. (1986). Two problems with back propagation and other steepest descent learning procedures for networks. In *Proceedings of the eighth annual conference of the cognitive science society, 1986* (pp. 823–832).

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 1–9).

Thoma, M. (2014). *On-line Recognition of Handwritten Mathematical Symbols* (Bachelor's Thesis, Karlsruhe Institute of Technology, Karlsruhe, Germany). Retrieved from http://martin-thoma.com/write-math

Thoma, M. (2017). The hasyv2 dataset. *arXiv preprint arXiv:1701.08380*.

Voosen, P. (2017). The ai detectives. American Association for the Advancement of Science.

Waldrop, M. M. (2016). More than moore. *Nature*, *530*(7589), 144–148.

Wang, S., & Kanwar, P. (2019). Bfloat16: The secret to high performance on cloud tpus. *Google Cloud Blog, August*. Accessed on 2019-11-26. Retrieved from https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus

Wu, Y., Kirillov, A., Massa, F., Lo, W.-Y., & Girshick, R. (2019). Detectron2. https://github.com/facebookresearch/detectron2.

Yam, J. Y., & Chow, T. W. (2000). A weight initialization method for improving training speed in feedforward neural network. *Neurocomputing*, *30*(1-4), 219–232.

Yarotsky, D. (2018). Universal approximations of invariant maps by neural networks. *arXiv preprint arXiv:1804.10306*.

Zeiler, M. D., & Fergus, R. (2014). Visualizing and understanding convolutional networks. In *European conference on computer vision* (pp. 818–833). Springer.

Zhang, C., Bengio, S., Hardt, M., Recht, B., & Vinyals, O. (2017). Understanding deep learning requires rethinking generalization. In *5th international conference on learning representations, ICLR 2017, toulon, france, april 24-26, 2017, conference track proceedings*. Retrieved from https://openreview.net/forum?id=Sy8gdB9xx

Zhou, D.-X. (2019). Universality of deep convolutional neural networks. *Applied and Computational Harmonic Analysis*.