TECHNICAL UNIVERSITY OF CRETE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

Overparametrized Deep Neural Networks: Convergence and Generalization Properties

Technical University of Crete

by

Christos Polyzos

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DIPLOMA DEGREE OF

ELECTRICAL AND COMPUTER ENGINEERING

September 2023

THESIS COMMITTEE

Professor Athanasios P. Liavas, *Thesis Supervisor*
Professor George N. Karystinos
Professor Michalis Zervakis

Abstract

In this thesis, we consider deep neural networks for Machine Learning. We depict neural networks as weighted directed graphs and we represent them as parametric functions that receive an input and compute an output, or prediction, given some fixed parameters, the weights and the biases. The quintessence of a neural network is the feed-forward model, in which the underlying graph does not contain cycles (acyclic graph) and the parametric function is defined in a compositional, or hierarchical, way.

Throughout our presentation, we focus on a supervised learning setting, where our neural network model, or learner, has access to a training set that contains examples of how pairs of input-output data are related. In other words, supervised learning amounts to learning from examples.

Given a training set, depending whether the outputs have real or categorical values, we consider regression and logistic regression. For each setting, we provide the basic statistical framework and construct a loss function known as the empirical risk. We train our neural network by minimizing the empirical risk w.r.t. its parameters by using gradient-based optimization methods. The gradient of the loss function is computed via the back-propagation algorithm.

We showcase the convergence and generalization properties of different algorithms (deep neural network models and optimization methods) using real-world data.

Acknowledgements

I would like to thank my thesis supervisor, Professor Athanasios Liavas, for his continuous guidance and support he offered me, all the way up to my graduation.

I would also like to thank my closest friends, Georgios S., Michail O., Emmanouil T. and Maria P. for providing me a welcome distraction out of the school.

Finally, I would like to give special thanks to my family for believing in me, and supporting me and my decisions throughout my life. For that, I will be grateful for ever.

Table of Contents

Chapter 1

Introduction

Artificial neural networks have demonstrated dominating performance over the last few decades in various fields. On a theoretical side, a long line of work has been focusing on training artificial neural networks with two layers, showing theoretical results of significant importance.

The theory of artificial neural networks with more than two layers remains still largely unsettled. However, they are used in practice to solve large-scale machine learning problems, due to their expressive power and generalization capabilities.

Chapter [1] is a brief introduction to artificial neural networks providing a notation section, in which we describe our conventions. Chapter [2] defines artificial neural networks as structures that can be represented by directed graphs. Additionally, it provides a basic optimization and statistical framework for machine learning. Chapter [3], presents the back-propagation algorithm, which computes the gradient of the loss function. Chapter [4] presents some of the state-of-the-art optimization methods that are used for the training procedure. Finally, Chapter [5] provides experimental results over different machine learning problems.

1.1 Notation

Throughout our presentation, we adopt standard mathematical notation. In this section, we describe our conventions and provide Table [1.1], which summarizes our notation.

We denote scalars with lowercase letters (e.g., x, α) and vectors by using boldface lowercase letters (e.g., $\mathbf{x}$, $\boldsymbol{\alpha}$). The jth coordinate of a vector $\mathbf{x}$ is denoted by $x_j = [\mathbf{x}]_j$. We use boldface uppercase letters (e.g., $\mathbf{X}$, $\mathbf{A}$) to denote matrices. The (j, m) element of a matrix $\mathbf{A}$ is denoted by $A_{j,m} = [\mathbf{A}]_{j,m}$. Additionally, we denote sets and graphs by calligraphic uppercase letters (e.g., $\mathcal{S}, \mathcal{G}$).

Table 1.1: Summary of notation

symbol	meaning
$\mathbb{R}$	the set of real numbers
$\mathbb{R}_+$	the set of non-negative real numbers
$\mathbb{R}^d$	the set of d-dimensional real vectors
$\mathbb{N}$	the set of natural numbers
$\mathbb{I}$[expression]	equals 1 if expression is true and 0 o.w.
x, $\mathbf{x}$	a scalar and a (column) vector, respectively
x_j	the jth element of vector $\mathbf{x}$
sign(x)	equals 1 if $x > 0$, 0 if $x = 0$ and -1 o.w.
$\|\mathbf{x}\|_2$	$= \left(\sum_{j=1}^d x_j^2\right)^{1/2}$, if $\mathbf{x} \in \mathbb{R}^d$
$\|\mathbf{x}\|_1$	$= \sum_{j=1}^d \|x_j\|$, if $\mathbf{x} \in \mathbb{R}^d$
$\mathbf{A} \in \mathbb{R}^{m \times n}$	$m \times n$ matrix over $\mathbb{R}$
$\mathbf{A}^\top$	transpose matrix of $\mathbf{A}$
$[\mathbf{A}]_{j,m}$ or $A_{j,m}$	the (j, m) element of $\mathbf{A}$
$[\mathbf{A}]_{:,j}$ or $\mathbf{A}_{:,j}$	the jth column of $\mathbf{A}$
$[\mathbf{A}]_{j,:}$ or $\mathbf{A}_{j,:}$	the jth row of $\mathbf{A}$
$\mathbf{I}_m \in \mathbb{R}^{m \times m}$	the $m \times m$ identity matrix
$\mathcal{X}$	input or feature domain (set)
$\mathcal{Y}$	output or label domain (set)
$\mathcal{S}$	$= \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^n$ the training set of n samples
$\mathcal{G}$	$= (\mathcal{V}, \mathcal{E})$ a graph with $\|\mathcal{V}\|$ nodes and $\|\mathcal{E}\|$ edges
$\mathcal{N}_L$	a neural network architecture with L layers
$\mathcal{V}_\ell$	the neurons (nodes) at layer ℓ
$v_j^{(\ell)} \in \mathcal{V}_\ell$	the jth neuron (node) at layer ℓ
$(v_m^{(\ell-1)}, v_j^{(\ell)}) \in \mathcal{E}$	a directed edge between neurons $v_m^{(\ell-1)}$ and $v_j^{(\ell)}$
k_ℓ	$= \begin{cases} \|\mathcal{V}_\ell\| - 1, \ell < L \\ \|\mathcal{V}_\ell\|, \ell = L \end{cases}$, the number of neurons at layer ℓ excluding biases
$\mathbf{W}^{(\ell)} \in \mathbb{R}^{k_\ell \times k_{\ell-1}}$	the weight parameter matrix at layer ℓ
$\mathbf{b}^{(\ell)} \in \mathbb{R}^{k_\ell}$	the bias parameter vector at layer ℓ
$\boldsymbol{\theta} \in \mathbb{R}^{\|\mathcal{E}\|}$	the parameter vector including all weight and bias parameters with $\|\mathcal{E}\| = \sum_{\ell=1}^L k_\ell(k_{\ell-1} + 1)$
$\{\boldsymbol{\theta}^{(t)}\}_{t=0}^T$	a sequence (set) of parameter vectors returned by a training algorithm after $T < \infty$ iterations
$\boldsymbol{\theta}^* \in \mathbb{R}^{\|\mathcal{E}\|}$	(sub-)optimal parameter vector
$\mathcal{H}_{\mathcal{N}_L}$	a hypothesis class of a neural network
$\mathbf{h}_{\mathcal{N}_L,\boldsymbol{\theta}} \in \mathcal{H}_{\mathcal{N}_L}$	a hypothesis for a fixed neural network $\mathcal{N}_L$ and parameter vector $\boldsymbol{\theta}$
$\hat{\mathbf{y}}_{\boldsymbol{\theta}}(\mathbf{x})$	$= \mathbf{h}_{\mathcal{N}_L,\boldsymbol{\theta}}(\mathbf{x})$, denotes the prediction of a fixed neural network when it is fed by an input $\mathbf{x} \in \mathcal{X}$
$\mathcal{L} : \mathbf{h}_{\mathcal{N}_L,\boldsymbol{\theta}}(\mathcal{X}) \times \mathcal{Y} \to \mathbb{R}$	a loss function, for $\mathbf{h}_{\mathcal{N}_L,\boldsymbol{\theta}} \in \mathcal{H}_{\mathcal{N}_L}$
$g'(x) = \frac{dg(x)}{dx}$	is the derivative of $g : \mathbb{R} \to \mathbb{R}$
$\frac{\partial g(\mathbf{x})}{\partial x_j}$	the partial derivative of $g : \mathbb{R}^d \to \mathbb{R}$ w.r.t. x_j, $j \in [d]$
$\nabla g(\mathbf{x})$	$= [\frac{\partial g(\mathbf{x})}{\partial x_1} \cdots \frac{\partial g(\mathbf{x})}{\partial x_d}]^\top$ the gradient of $g : \mathbb{R}^d \to \mathbb{R}$
$\partial g(\mathbf{x})$	the subdifferential (set) of $g : \mathbb{R}^d \to \mathbb{R}$
$\odot$	element-wise multiplication operator

Scalar functions of several variables are denoted by lowercase letters (e.g., $f(\mathbf{x})$), while vector functions of several variables are denoted by boldface lowercase letters (e.g., $\mathbf{f}(\mathbf{x})$).

In the context of machine learning, a common approach is to define the input or feature domain set as $\mathcal{X}$ and the output or label domain set as $\mathcal{Y}$. We adopt this notation and, in our analysis, we omit, unless it is stated otherwise, that $\mathcal{X} \subseteq \mathbb{R}^d$ and $\mathcal{Y} \subseteq \mathbb{R}^{d'}$ if the output domain set consists of d'-dimensional real value vectors over $\mathbb{R}$ or $\mathcal{Y} = \{1, \ldots, c\}$, for some positive integer $c > 0$, which corresponds to the number of classes in a classification problem.

Of course, the input domain set $\mathcal{X}$ can also represent more complicated inputs. For example, for grey-scale images, if $\mathcal{X} \subseteq \mathbb{R}^{d_1 \times d_2}$, and for RGB images where $\mathcal{X} \subseteq \mathbb{R}^{3 \times d_1 \times d_2}$, with the first dimension denoting the red-green-blue channel of the image.

Chapter 2

Artificial Neural Networks and Machine Learning

2.1 Feed-Forward Neural Networks

An artificial neural network is a computational model inspired by the structure of the biological neural networks of the human brain. A neural network can be represented as a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ [1, ch. 20, p. 268], with $|\mathcal{V}|$ nodes and $|\mathcal{E}|$ edges. The nodes correspond to the neurons and each (directed) edge of the graph is associated with a weight value, which determines the strength of the communication link between the connected nodes. Generally, each neuron computes a scalar function.

In this thesis, we focus on feed-forward neural networks, in which the underlying graph does not contain cycles. Furthermore, we assume that the feed-forward network is organized in layers. In fact, any directed acyclic graph can be arranged topologically in order to have a layered structure [2, ch. 22.4, p. 612]. In particular, we partition the set of nodes $\mathcal{V}$ into a union of nonempty subsets, $\mathcal{V} = \bigcup_{\ell=0}^{L} \mathcal{V}_\ell$, with $\mathcal{V}_\ell \neq \emptyset$, $\forall \ell \in [L]$, such that every edge in $\mathcal{E}$ connects some node of $\mathcal{V}_{\ell-1}$ with some node in $\mathcal{V}_\ell$, for all $\ell \in [L]$.

The first layer, $\mathcal{V}_0$, is called the input layer. It contains $d+1$ neurons, where $d \in \mathbb{N}$ is the dimensionality of the input space. Let us feed the neural network with the input vector $\mathbf{x} \in \mathcal{X} \subseteq \mathbb{R}^d$, $d \in \mathbb{N}$. For every $j \in [d]$, the output of the jth neuron is simply x_j, the jth coordinate of the input vector. The output of the $(d+1)$th neuron in $\mathcal{V}_0$ is always the constant 1.

Next, we compute the output for each neuron in the graph. Fix the jth neuron at layer ℓ, $v_j^{(\ell)} \in \mathcal{V}_\ell$, for some $j \in \{1, \ldots, |\mathcal{V}_\ell|\}$. Let $z_j^{(\ell)}(\mathbf{x})$ and $a_j^{(\ell)}(\mathbf{x})$ denote the input and the output of neuron $v_j^{(\ell)}$, respectively, when the neural network is fed with the input vector $\mathbf{x} \in \mathcal{X}$. Then, for $j \in \{1, \ldots, |\mathcal{V}_\ell|\}$ and $\forall \ell \in [L]$, we have

$$z_j^{(\ell)}(\mathbf{x}) = \sum_{k=1}^{|\mathcal{V}_{\ell-1}|} W_{j,k}^{(\ell)} a_k^{(\ell-1)}(\mathbf{x}), \tag{2.1}$$

$$a_j^{(\ell)}(\mathbf{x}) = \sigma_\ell \left(z_j^{(\ell)}(\mathbf{x}) \right), \ell \neq L, \tag{2.2}$$

where $W_{j,m}^{(\ell)} \in \mathbb{R}$ denotes the weight value of the edge $(v_m^{(\ell-1)}, v_j^{(\ell)}) \in \mathcal{E}$

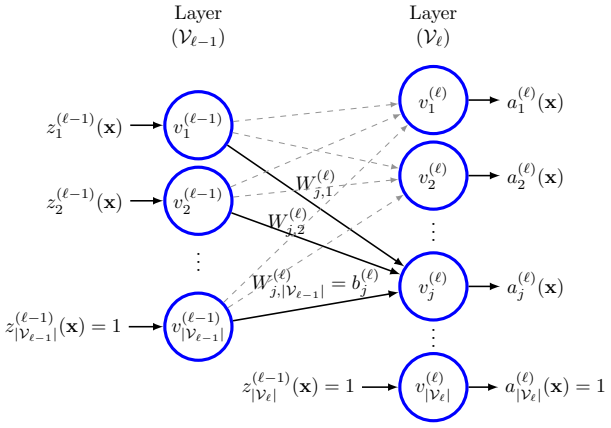


Figure 2.1: The input and output for each neuron $v_k^{(\ell-1)}$ are given by $z_k^{(\ell-1)}(\mathbf{x})$ and $a_k^{(\ell-1)}(\mathbf{x})$, respectively, for all $k \in \{1, \ldots, |\mathcal{V}_{\ell-1}|\}$. The input of the neuron $v_j^{(\ell)}$ is a weighted sum over all the outputs of the neurons of the previous layer and is given by $z_j^{(\ell)}(\mathbf{x})$ and its output is $a_j^{(\ell)}(\mathbf{x})$.

that connects the neurons $v_m^{(\ell-1)}$ and $v_j^{(\ell)}$ and $\sigma_\ell : \mathbb{R} \to \mathbb{R}$ is the activation function[1] at the ℓth layer. We use the ℓ symbol for the activation function to denote that different layers may compute different activation functions.

Each neuron $v_{|\mathcal{V}_\ell|}^{(\ell)}$, $\ell \in [L]$, outputs the constant 1 as it is shown in Fig [2.1]. To better understand the use of the bottom neuron for each $\ell \in [L]$, we consider $k_\ell = |\mathcal{V}_\ell| - 1$, $\forall \ell \in [L-1]$ and $k_L = |\mathcal{V}_L|$. Then, expression (2.1) can be equivalently written as

$$\begin{aligned} z_j^{(\ell)}(\mathbf{x}) &= \sum_{k=1}^{k_{\ell-1}} W_{j,k}^{(\ell)} a_k^{(\ell-1)}(\mathbf{x}) + W_{j,|V_{\ell-1}|}^{(\ell)} a_{|V_{\ell-1}|}^{(\ell-1)}(\mathbf{x}) \\ &= \sum_{k=1}^{k_{\ell-1}} W_{j,k}^{(\ell)} a_k^{(\ell-1)}(\mathbf{x}) + b_j^{(\ell)}, \end{aligned} \tag{2.3}$$

for all $\ell \in [L]$, where $b_j^{(\ell)} = W_{j,|V_{\ell-1}|}^{(\ell)}$ and $a_{|V_{\ell-1}|}^{(\ell-1)}(\mathbf{x}) = 1$. The use of the last neuron is to simply add the term $b_j^{(\ell)} \in \mathbb{R}$, called bias, to the weighted sum of the outputs of the neurons $\{v_m^{(\ell-1)}\}_{m=1}^{k_{\ell-1}}$, which are connected to the neuron $v_j^{(\ell)}$ via the weighted edge $(v_m^{(\ell-1)}, v_j^{(\ell)})$, $\forall m \in [k_{\ell-1}]$ and $j \in [k_\ell]$.

[1] We assume that each node at layer ℓ computes the same activation function $\sigma_\ell(\cdot)$.

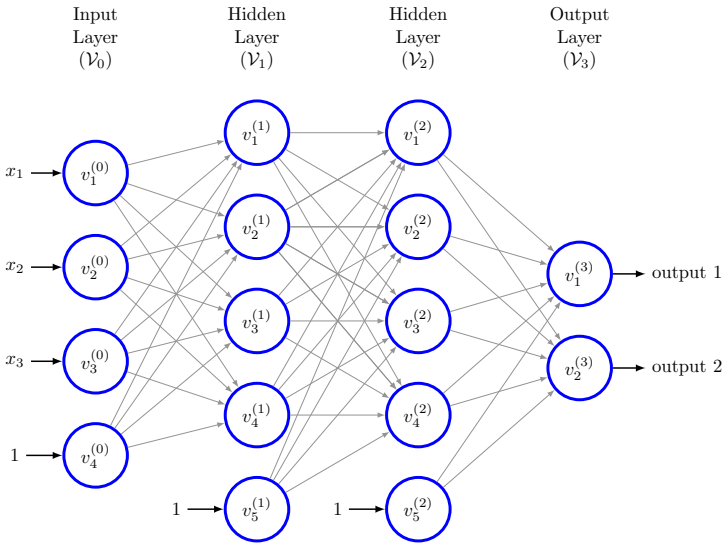


Figure 2.2: Illustration of a neural network with three layers (excluding $\mathcal{V}_0$) with size 16, width 5, and depth 3. The network is fed with an input vector $\mathbf{x} \in \mathbb{R}^{|\mathcal{V}_0|-1} = \mathbb{R}^3$ and outputs a two-dimensional vector in $\mathbb{R}^{|\mathcal{V}_3|} = \mathbb{R}^2$. Note that, for each layer, except the output layer $\mathcal{V}_3$, the last neuron with no incoming edges is fed with a constant 1 and it transmits it to the output.

Henceforth, expressions (2.1) and (2.2) are reformulated as

$$z_j^{(\ell)}(\mathbf{x}) = \sum_{k=1}^{k_{\ell-1}} W_{j,k}^{(\ell)} a_k^{(\ell-1)}(\mathbf{x}) + b_j^{(\ell)}, \tag{2.4}$$

$$a_j^{(\ell)}(\mathbf{x}) = \sigma_\ell \left(z_j^{(\ell)}(\mathbf{x}) \right), \ell \neq L, \tag{2.5}$$

for $j \in [k_\ell]$ and $\ell \in [L]$.

In Fig[2.1], we depict the computation of the output of any neuron at the ℓth layer by using the expressions (2.4) and (2.5).

The layers $\mathcal{V}_1, \ldots, \mathcal{V}_{L-1}$ are called hidden layers, while the last layer $\mathcal{V}_L$ is called the output layer. Depending on the problem, the output layer may contain one or more neurons, whose output is the output of the neural network. The number of layers, L, is also known as the depth, or capacity, of the neural network, while the width of the network is defined as $\max_\ell |\mathcal{V}_\ell|$.

Of course, the analysis of feed-forward neural networks can be extended to convolutional neural networks. Further details regarding convolutional neural networks, convolutional layers and, generally, convolutional arithmetic in the context of deep learning can be found in [3, ch. 6.3, p. 124], [4, ch. 14.2.1, p. 464] and [5].

2.2 Neural Networks as Parametric Functions

From a general perspective, we define the architecture of a neural network with $L \in \mathbb{N}$ layers as the triplet $\mathcal{N}_L = \left(\mathcal{V}, \mathcal{E}, \{\sigma_\ell\}_{\ell=1}^{L-1}\right)$, where $\mathcal{V} = \bigcup_{\ell=0}^{L} \mathcal{V}_\ell$ [1, ch. 20.2, p. 270]. Generally, once an architecture is specified, we can treat the neural network as a parametric function $\mathbf{h}_{\mathcal{N}_L,\boldsymbol{\theta}} : \mathbb{R}^{k_0} \to \mathbb{R}^{k_L}$, where $\boldsymbol{\theta}$ serves the role of a parameter vector to the function $\mathbf{h}_{\mathcal{N}_L,\boldsymbol{\theta}}(\cdot)$. In particular, vector $\boldsymbol{\theta}$ collects the weights and biases of the neural network in a single vector, such as

$$\boldsymbol{\theta} = \begin{bmatrix} \boldsymbol{\theta}_1^\top & \boldsymbol{\theta}_2^\top & \cdots & \boldsymbol{\theta}_L^\top \end{bmatrix}^\top \in \mathbb{R}^{|\mathcal{E}|}, \tag{2.6}$$

where $\boldsymbol{\theta}_\ell = \begin{bmatrix} \text{vec}\left(\mathbf{W}^{(\ell)}\right)^\top & \left(\mathbf{b}^{(\ell)}\right)^\top \end{bmatrix}^\top \in \mathbb{R}^{k_\ell(k_{\ell-1}+1)}$, $[\mathbf{W}^{(\ell)}]_{j,m} = W_{j,m}^{(\ell)}$ and $[\mathbf{b}^{(\ell)}]_j = b_j^{(\ell)}$, for $j \in [k_\ell]$ and $m \in [k_{\ell-1}]$, for each $\ell \in [L]$.

Note that the number of parameters, or the number of edges of the underlying graph corresponding to the architecture $\mathcal{N}_L$, can be computed as $|\mathcal{E}| = \sum_{\ell=1}^{L} k_\ell(k_{\ell-1}+1)$.

Now, let us feed the network $\mathcal{N}_L$ with an input vector $\mathbf{x} \in \mathcal{X} \subseteq \mathbb{R}^d$, with $k_0 = |\mathcal{V}_0| - 1 = d$. The output of the neural network is defined as

$$\mathbf{h}_{\mathcal{N}_L,\boldsymbol{\theta}}(\mathbf{x}) := \begin{bmatrix} z_1^{(L)}(\mathbf{x}) & z_2^{(L)}(\mathbf{x}) & \cdots & z_{k_L}^{(L)}(\mathbf{x}) \end{bmatrix}^\top \in \mathbb{R}^{k_L}. \tag{2.7}$$

Depending on the problem, the number of neurons of the last layer, k_L, can be selected accordingly in order to represent scalar or vector mappings. The intermediate number of neurons $\{k_\ell\}_{\ell=1}^{L-1}$ corresponding to the hidden layers $\{\mathcal{V}_\ell\}_{\ell=1}^{L-1}$ determine the expressiveness of the parametric function $\mathbf{h}_{\mathcal{N}_L,\boldsymbol{\theta}}(\mathbf{x})$.

In machine learning, the parametric function $\mathbf{h}_{\mathcal{N}_L,\boldsymbol{\theta}}(\cdot)$ is known as a hypothesis [1, ch. 20.2, p. 270]. Generally, by changing the parameter vector $\boldsymbol{\theta}$, we obtain a different hypothesis. The class of all possible hypotheses for a fixed neural network architecture is called the hypothesis class and is given by

$$\mathcal{H}_{\mathcal{N}_L} = \{\mathbf{h}_{\mathcal{N}_L,\boldsymbol{\theta}} : \mathbb{R}^{k_0} \to \mathbb{R}^{k_L} \mid \boldsymbol{\theta} \in \mathbb{R}^{|\mathcal{E}|}\}^2. \tag{2.8}$$

[2] A more abstract way to describe the hypothesis class of a fixed neural network that can be also fed by images is: $\mathcal{H}_{\mathcal{N}_L} = \{\mathbf{h}_{\mathcal{N}_L,\boldsymbol{\theta}} : \mathcal{X} \to \mathbf{h}_{\mathcal{N}_L,\boldsymbol{\theta}}(\mathcal{X}) \subseteq \mathbb{R}^{k_L} \mid \boldsymbol{\theta} \in \mathbb{R}^{|\mathcal{E}|}\}$, where $\mathcal{X} \subseteq \mathbb{R}^{d \times d}$, if the input domain set consists of gray-scale images or $\mathcal{X} \subseteq \mathbb{R}^{3 \times d \times d}$ for RGB images, having red, green and blue channels.

2.3 Basic Machine Learning and Optimization Framework

In supervised machine learning, we have access to training data that contain examples of how input[3] vectors $\mathbf{x} \in \mathcal{X}$ relate to outputs [4] $\mathbf{y} \in \mathcal{Y}$. In particular, we assume that our model, or learner, has access to a training dataset $\mathcal{S} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{n}$, $n \in \mathbb{N}$, where $(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{X} \times \mathcal{Y}$, with $\mathcal{X} \subseteq \mathbb{R}^d$ and $\mathcal{Y} \subset \mathbb{R}^{d'}$, for $d' \in \mathbb{N}$, if outputs are numerical, or $\mathcal{Y} = \{1, \dots, c\}$, for $c > 1$ classes, if outputs are categorical.

Furthermore, we assume that each pair $(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{X} \times \mathcal{Y}$ is drawn independently from the joint probability distribution $p : \mathcal{X} \times \mathcal{Y} \to \mathcal{P} \subseteq \mathbb{R}_+$, representing the true relationship between inputs and outputs. Of course, the distribution $p(\cdot, \cdot)$ is not known to the learner.

2.3.1 Risk Minimization

Ideally, for a fixed neural network architecture $\mathcal{N}_L$, we aim to choose a hypothesis $\mathbf{h}_{\mathcal{N}_L,\boldsymbol{\theta}} \in \mathcal{H}_{\mathcal{N}_L}$ parameterized by the vector $\boldsymbol{\theta} \in \mathbb{R}^{|\mathcal{E}|}$ that minimizes the expected risk, $R : \mathbb{R}^{|\mathcal{E}|} \to \mathbb{R}$, defined as

$$\begin{aligned} R(\boldsymbol{\theta}) &:= \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim p}\left[\mathcal{L}\left(\mathbf{h}_{\mathcal{N}_L,\boldsymbol{\theta}}(\mathbf{x}), \mathbf{y}\right)\right] \\ &= \int_{\mathcal{X}\times\mathcal{Y}} \mathcal{L}\left(\mathbf{h}_{\mathcal{N}_L,\boldsymbol{\theta}}(\mathbf{x}), \mathbf{y}\right) dp(\mathbf{x}, \mathbf{y}), \end{aligned} \tag{2.9}$$

where $\mathcal{L} : \mathbf{h}_{\mathcal{N}_L,\boldsymbol{\theta}}(\mathcal{X}) \times \mathcal{Y} \to \mathbb{R}$ is a loss function.

While our primal goal is to minimize expression (2.9), this is impossible since the distribution $p(\cdot, \cdot)$ is not known to the learner. However, the learner has access to the training set $\mathcal{S}$, whose pairs are drawn independently according to the distribution $p(\cdot, \cdot)$.

Next, we define the empirical risk function $R_{\mathcal{S}} : \mathbb{R}^{|\mathcal{E}|} \to \mathbb{R}$ as

$$R_{\mathcal{S}}(\boldsymbol{\theta}) := \frac{1}{|\mathcal{S}|} \sum_{i=1}^{|\mathcal{S}|} \mathcal{L}\left(\mathbf{h}_{\mathcal{N}_L,\boldsymbol{\theta}}(\mathbf{x}_i), \mathbf{y}_i\right). \tag{2.10}$$

In machine learning, a common approach is to employ the empirical risk (2.10), instead of the expected risk (2.9), as the objective function, which we minimize w.r.t. the parameter vector $\boldsymbol{\theta}$, hoping that the solutions of the empirical and the expected risk are sufficiently close.

[3] The input is also called feature, attribute, predictor, regressor, covariate, explanatory variable, controlled variable or independent variable.

[4] The output is also called response, regressand, label, explained variable or dependent variable.

2.3.2 Regularized Risk Minimization

Instead of minimizing the empirical risk, we can use a regularized empirical risk as an objective function. We define the regularized empirical risk as

$$\tilde{R}_{\mathcal{S}}(\boldsymbol{\theta}) := R_{\mathcal{S}}(\boldsymbol{\theta}) + \lambda\Omega(\boldsymbol{\theta}), \tag{2.11}$$

where $\Omega : \mathbb{R}^{|\mathcal{E}|} \to \mathbb{R}_+$ is a regularization penalty term and $\lambda > 0$ controls the importance of the regularization. For $\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{\theta}\|_2^2$ the regularization is known as ℓ_2-regularization, while for $\Omega(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|_1$ the regularization is known as ℓ_1-regularization. Further details regarding to regularization can be found in [3, ch. 5.3, p. 93] and [6, ch. 3.1.4, p. 145].

2.3.3 Simplified Notation

Expressions (2.9) and (2.10) show explicitly how the expected and empirical risks depend on the loss function. In particular, the loss function depends on the neural network architecture $\mathcal{N}_L$, the parameter vector $\boldsymbol{\theta}$ and the input-output training pair $(\mathbf{x}, \mathbf{y}) \in \mathcal{X} \times \mathcal{Y}$. However, when discussing certain algorithmic ideas (i.e., back-propagation, optimization methods), a simplified notation is preferable, since it leads to straightforward and simplified expressions.

First, we define

$$\hat{\mathbf{y}}_{\boldsymbol{\theta}}(\mathbf{x}) := \mathbf{h}_{\mathcal{N}_L,\boldsymbol{\theta}}(\mathbf{x}) \tag{2.12}$$

as the output vector of a fixed neural network $\mathcal{N}_L$ parameterized by $\boldsymbol{\theta}$, when it is fed with the input vector $\mathbf{x} \in \mathcal{X}$. Combining expressions (2.7) and (2.12) we get that $\hat{\mathbf{y}}_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{z}^{(L)}(\mathbf{x})$, where $[\mathbf{z}^{(L)}(\mathbf{x})]_j = z_j^{(L)}(\mathbf{x})$, $\forall j \in [k_L]$.

Additionally, we define the loss w.r.t. parameters $\boldsymbol{\theta}$ as

$$f(\boldsymbol{\theta}) := \mathcal{L}\left(\hat{\mathbf{y}}_{\boldsymbol{\theta}}(\mathbf{x}), \mathbf{y}\right), \tag{2.13}$$

over an arbitrary input-output training pair $(\mathbf{x}, \mathbf{y}) \in \mathcal{X} \times \mathcal{Y}$. In a similar manner, we define the sample loss incurred by the parameter vector $\boldsymbol{\theta}$ w.r.t. the ith sample as

$$f_i(\boldsymbol{\theta}) := \mathcal{L}\left(\hat{\mathbf{y}}_{\boldsymbol{\theta}}(\mathbf{x}_i), \mathbf{y}_i\right), \tag{2.14}$$

where $(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{S}$, for some $i \in [n]$. We can write the empirical risk as the arithmetic mean of the sample losses:

$$R_{\mathcal{S}}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\boldsymbol{\theta}). \tag{2.15}$$

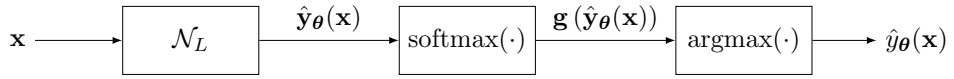


Figure 2.3: A graphical representation for the logistic regression setting. We feed the neural network with an input $\mathbf{x}$ and the network predicts $\hat{\mathbf{y}}_{\boldsymbol{\theta}}(\mathbf{x})$. Then, the prediction passes through a softmax operation and computes $\mathbf{g}\left(\hat{\mathbf{y}}_{\boldsymbol{\theta}}(\mathbf{x})\right)$, where $\left[\mathbf{g}\left(\hat{\mathbf{y}}_{\boldsymbol{\theta}}(\mathbf{x})\right)\right]_i = g_i\left(\hat{\mathbf{y}}_{\boldsymbol{\theta}}(\mathbf{x})\right),\ i \in [c]$. Last, the network predicts the class $\hat{y}_{\boldsymbol{\theta}}(\mathbf{x})$, by using the argmax operation over the softmax function.

2.4 Parametric Modelling with Neural Networks

In this section, we introduce two basic parametric models for learning: regression and logistic regression. The key point of parametric modelling is that it contains some parameters $\boldsymbol{\theta}$, which are learned from the training data $\mathcal{S}$.

In the context of neural networks, changing the weight parameters $\boldsymbol{\theta}$ leads to selecting a different hypothesis $\hat{\mathbf{y}}_{\boldsymbol{\theta}}(\mathbf{x})$ from the hypothesis class $\mathcal{H}_{\mathcal{N}_L}$.

2.4.1 Regression

In this setting, we wish to find a "pattern" in the data structure —a functional relationship between inputs $\mathbf{x} \in \mathcal{X}$ and numerical outputs $\mathbf{y} \in \mathcal{Y} \subseteq \mathbb{R}^{d'}$. A common approach is to use the squared-loss [6, ch. 5.2, p. 234] as an error metric between the prediction of the neural network $\hat{\mathbf{y}}_{\boldsymbol{\theta}}(\mathbf{x})$ and the true label $\mathbf{y}$, given by

$$\begin{aligned} f(\boldsymbol{\theta}) &= \frac{1}{2}\|\hat{\mathbf{y}}_{\boldsymbol{\theta}}(\mathbf{x}) - \mathbf{y}\|_2^2 \\ &= \frac{1}{2}\|\mathbf{z}^{(L)}(\mathbf{x}) - \mathbf{y}\|_2^2. \end{aligned} \tag{2.16}$$

Intuitively, the squared loss is zero if $\hat{\mathbf{y}}_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{y}$ and grows quadratically as the Euclidean distance between $\hat{\mathbf{y}}_{\boldsymbol{\theta}}(\mathbf{x})$ and $\mathbf{y}$ increases.

Since the squared-loss function is selected for the regression setting, the empirical risk is formulated as

$$R_{\mathcal{S}}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\|\hat{\mathbf{y}}_{\boldsymbol{\theta}}(\mathbf{x}_i) - \mathbf{y}_i\|_2^2. \tag{2.17}$$

2.4.2 Logistic Regression

Logistic regression is used for classification problems. In this setting, we wish to find a functional relationship between input $\mathbf{x} \in \mathcal{X}$ and categorical

outputs $y \in \mathcal{Y} \subseteq \{1, \ldots, c\}$, for $c > 1$. In order to formalize this setting, we use the so-called softmax function [3, ch. 3.2, p. 49], [3, ch. 6, p. 117].

The idea behind using the softmax function is that we wish to interpret the outputs of the neural network as probabilities. Therefore, the output vectors should be non-negative and sum to one. For that, we define the softmax function as:

$$\mathbf{g}(\mathbf{u}) := \frac{1}{\sum_{k=1}^{s} e^{u_k}} \begin{bmatrix} e^{u_1} & e^{u_2} & \cdots & e^{u_s} \end{bmatrix}^\top, \tag{2.18}$$

where $\mathbf{u} \in \mathbb{R}^s$, $s > 0$. Furthermore, we define

$$\hat{y}_{\boldsymbol{\theta}}(\mathbf{x}) := \underset{i \in [c]}{\operatorname{argmax}}\, g_i\left(\hat{\mathbf{y}}_{\boldsymbol{\theta}}(\mathbf{x})\right) \in \mathcal{Y}, \tag{2.19}$$

as the prediction of the class of the neural network, where each function $g_i(\mathbf{u}) = [\mathbf{g}(\mathbf{u})]_i$ denotes the ith coordinate of the softmax function evaluated at $\mathbf{u} \in \mathbb{R}^s$, $s > 0$, for $i \in [s]$ (see Fig [2.3]).

The loss function for the training pair $(\mathbf{x}, y)$ is given by

$$\begin{aligned} f(\boldsymbol{\theta}) &= -\ln g_y(\hat{\mathbf{y}}_{\boldsymbol{\theta}}(\mathbf{x})) \\ &= -\ln\left(\frac{e^{z_y^{(L)}(\mathbf{x})}}{\sum_{k=1}^{c} e^{z_k^{(L)}(\mathbf{x})}}\right) \\ &= -z_y^{(L)}(\mathbf{x}) + \ln \sum_{k=1}^{c} e^{z_k^{(L)}(\mathbf{x})}, \end{aligned} \tag{2.20}$$

which is known as the cross-entropy loss function [3, ch. 6.2, p. 119], while the empirical risk is reformulated as

$$R_{\mathcal{S}}(\boldsymbol{\theta}) = -\frac{1}{n} \sum_{i=1}^{n} \ln g_{y_i}(\hat{\mathbf{y}}_{\boldsymbol{\theta}}(\mathbf{x}_i)). \tag{2.21}$$

2.5 Statistical Perspective of Learning

In order to derive a principled way of selecting an appropriate loss function w.r.t. a given supervised machine learning setting (e.g., regression, logistic regression), we present a statistical interpretation of learning. Additional information can be found in [6, ch. 5.2, p. 233] and [6, ch. 5.7, p. 277], [6, ch. 5.7, p. 278], [4, ch. 10.2.3, p. 338]. See also [7, ch. 14, p. 569], [7, ch. 16, p. 623].

2.5.1 Probabilistic Interpretation of Neural Networks

So far, we have viewed neural networks as a general class of parametric functions. However, it is possible to provide another view of neural networks by

giving a probabilistic interpretation [6, ch. 5.2, p. 233]. We denote the probability distribution function (pdf) for regression problems, or probability mass function (pmf) for logistic regression, respectively, over possible vector outputs $\mathbf{y} \in \mathcal{Y}$, given an input vector $\mathbf{x} \in \mathcal{X}$, parameterized by the vector $\boldsymbol{\theta} \in \mathbb{R}^{|\mathcal{E}|}$ as

$$q(\mathbf{y} \mid \mathbf{x}; \boldsymbol{\theta}) \in \mathcal{Q} \subseteq \mathbb{R}_+. \tag{2.22}$$

The distribution in (2.22) is known as the likelihood function. We distinguish between the likelihood according to the model (neural network), $q(\mathbf{y} \mid \mathbf{x}; \boldsymbol{\theta})$, and the likelihood w.r.t. the true data distribution $p(\mathbf{y} \mid \mathbf{x})$[5]. We assume that the probabilistic model, parameterized by $\boldsymbol{\theta}$, is "proper" (flexible enough), meaning that, $\exists \boldsymbol{\theta}^* \in \mathbb{R}^{|\mathcal{E}|}$ such that $q(\mathbf{y} \mid \mathbf{x}; \boldsymbol{\theta}^*) = p(\mathbf{y} \mid \mathbf{x})$.

2.5.2 Maximum Likelihood Estimation

In the Maximum Likelihood (ML) setting, we treat $\boldsymbol{\theta}$ as a deterministic unknown parameter and we formulate the ML estimation by solving the optimization problem

$$\begin{aligned} \boldsymbol{\theta}_{\text{ML}} &= \text{argmax}_{\boldsymbol{\theta}}\, q(\mathbf{Y} \mid \mathbf{X}; \boldsymbol{\theta}) \\ &= \text{argmax}_{\boldsymbol{\theta}}\, \ln q(\mathbf{Y} \mid \mathbf{X}; \boldsymbol{\theta}). \end{aligned} \tag{2.23}$$

Here, $q(\mathbf{Y} \mid \mathbf{X}; \boldsymbol{\theta})$ is the joint likelihood distribution[6] function of all observed outputs $\mathbf{y}_i \in \mathcal{Y}$ given all inputs $\mathbf{x}_i \in \mathcal{X}$, parameterized by $\boldsymbol{\theta}$, where each pair $(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{S}$, $\forall i \in [n]$, $\mathbf{Y} = \begin{bmatrix} \mathbf{y}_1^\top & \dots & \mathbf{y}_n^\top \end{bmatrix}^\top$ and $\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^\top & \dots & \mathbf{x}_n^\top \end{bmatrix}^\top$. Assuming that the n points are statistically independent, the joint likelihood factorizes as

$$\begin{aligned} q(\mathbf{Y} \mid \mathbf{X}; \boldsymbol{\theta}) &= \prod_{i=1}^{n} q(\mathbf{y}_i \mid \mathbf{x}_i; \boldsymbol{\theta}) \\ \iff \ln q(\mathbf{Y} \mid \mathbf{X}; \boldsymbol{\theta}) &= \sum_{i=1}^{n} \ln q(\mathbf{y}_i \mid \mathbf{x}_i; \boldsymbol{\theta}) \end{aligned} \tag{2.24}$$

Therefore, the maximization problem (2.23) can be equivalently reformulated as:

$$\begin{aligned} \boldsymbol{\theta}_{\text{ML}} &= \text{argmax}_{\boldsymbol{\theta}} \sum_{i=1}^{n} \ln q(\mathbf{y}_i \mid \mathbf{x}_i; \boldsymbol{\theta}) \\ &= \text{argmax}_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} \ln q(\mathbf{y}_i \mid \mathbf{x}_i; \boldsymbol{\theta}). \end{aligned} \tag{2.25}$$

[5] Theoretically, if $p(\mathbf{x}, \mathbf{y})$ is known, we can compute $p(\mathbf{x})$ and $p(\mathbf{y})$ and apply the Bayes' rule to compute $p(\mathbf{y} \mid \mathbf{x})$.

[6] Pdf for regression problems or pmf for logistic regression problems.

For the regression setting, we treat $\mathbf{y}$ as a continuous random variable. We assume that the conditional distribution of each output $\mathbf{y}_i$ given the input $\mathbf{x}_i$ and parameterized by $\boldsymbol{\theta}$ is

$$q(\mathbf{y}_i \mid \mathbf{x}_i; \boldsymbol{\theta}) = \mathcal{N}\left(\mathbf{y}_i; \hat{\mathbf{y}}_{\boldsymbol{\theta}}(\mathbf{x}_i), \beta^2\mathbf{I}\right), \tag{2.26}$$

for each $i \in [n]$, where

$$\mathcal{N}(\mathbf{y}; \boldsymbol{\mu}, \mathbf{C}) := \frac{1}{(2\pi)^{m/2}|\mathbf{C}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{y}-\boldsymbol{\mu})^\top \mathbf{C}^{-1}(\mathbf{y}-\boldsymbol{\mu})\right) \tag{2.27}$$

is the pdf of an m-dimensional Gaussian vector $\mathbf{y}$ parameterized with mean $\boldsymbol{\mu} \in \mathbb{R}^m$ and covariance matrix $\mathbf{C} = \mathbf{C}^\top \in \mathbb{R}^{m\times m}$. In our case, the mean for each term is $\hat{\mathbf{y}}_{\boldsymbol{\theta}}(\mathbf{x}_i) \in \mathbb{R}^{d'}$ and the covariance matrix is $\beta^2\mathbf{I} \in \mathbb{R}^{d'\times d'}$. Substituting (2.26) in (2.25), leads to minimizing the squared-loss, which we have selected as a loss function for the regression setting in (2.17).

In logistic regression (classification), we treat $y \in \{1\ldots, c\}$ as a discrete random variable. In this setting, expression (2.25) simplifies to

$$\text{argmax}_{\boldsymbol{\theta}} \frac{1}{n}\sum_{i=1}^{n} \ln q(y_i \mid \mathbf{x}_i; \boldsymbol{\theta}), \tag{2.28}$$

since each y_i is a scalar [3, ch. 3.2, p. 45]. Furthermore, we assume that the conditional distribution of each output y_i, given the input $\mathbf{x}_i$ and parameter $\boldsymbol{\theta}$, is

$$q(y_i \mid \mathbf{x}_i; \boldsymbol{\theta}) = \text{Cat}\left(y_i; \mathbf{g}\left(\hat{\mathbf{y}}_{\boldsymbol{\theta}}(\mathbf{x}_i)\right)\right), \tag{2.29}$$

for each $i \in [n]$, where

$$\text{Cat}(y; \mathbf{r}) := \prod_{j=1}^{c} r_j^{\mathbb{I}[y=j]} \tag{2.30}$$

is the c-dimensional categorical distribution over c different events [4, ch. 10.3, p. 346]. The vector $\mathbf{r} = \begin{bmatrix} r_1 & \cdots & r_c \end{bmatrix}^\top$, where $0 \le r_i \le 1$, $\forall i \in [c]$ and $\sum_{i=1}^{c} r_i = 1$, specifies the probabilities for each outcome over c possible events. Substituting (2.29) in (2.28) leads to minimizing the cross-entropy loss, which we have selected as a loss function for the logistic regression setting in (2.21). Additional information can be found in [6, ch. 5.2, p. 233], [4, ch. 4.1, p. 105].

2.5.3 Maximum A-Posteriori Estimation

In the Maximum A-Posteriori (MAP) setting, the parameter $\boldsymbol{\theta}$ is treated as a random variable. Given the observations $\mathbf{X}$ and $\mathbf{Y}$, the most probable

parameter $\boldsymbol{\theta}$ is given by the MAP estimation, by solving the optimization problem

$$\begin{aligned}\boldsymbol{\theta}_{\text{MAP}} &= \text{argmax}_{\boldsymbol{\theta}} q(\boldsymbol{\theta} \mid \mathbf{X}, \mathbf{Y}) \\ &= \text{argmax}_{\boldsymbol{\theta}} \ln q(\boldsymbol{\theta} \mid \mathbf{X}, \mathbf{Y}).\end{aligned} \tag{2.31}$$

Applying the Bayes' rule, we obtain

$$\begin{aligned}q(\boldsymbol{\theta} \mid \mathbf{X}, \mathbf{Y}) &= \frac{q(\mathbf{Y} \mid \mathbf{X}, \boldsymbol{\theta}) q(\boldsymbol{\theta} \mid \mathbf{X})}{q(\mathbf{Y} \mid \mathbf{X})} \\ \iff \ln q(\boldsymbol{\theta} \mid \mathbf{X}, \mathbf{Y}) &= \ln q(\mathbf{Y} \mid \mathbf{X}, \boldsymbol{\theta}) + \ln q(\boldsymbol{\theta}) + \ln q(\mathbf{Y} \mid \mathbf{X})^{-1},\end{aligned} \tag{2.32}$$

since random vectors[7] $\mathbf{X}$ and $\boldsymbol{\theta}$ are independent, that is, $q(\boldsymbol{\theta} \mid \mathbf{X}) = q(\boldsymbol{\theta})$. Additionally, we note that $q(\mathbf{Y} \mid \mathbf{X})^{-1}$ is not a function of $\boldsymbol{\theta}$[8]. Hence, after disregarding the term $\ln q(\mathbf{X}, \mathbf{Y})^{-1}$, the maximization problem (2.31) can be equivalently expressed as

$$\boldsymbol{\theta}_{\text{MAP}} = \text{argmax}_{\boldsymbol{\theta}} \ln q(\mathbf{Y} \mid \mathbf{X}, \boldsymbol{\theta}) + \ln q(\boldsymbol{\theta}). \tag{2.33}$$

The first term $\ln q(\mathbf{Y} \mid \mathbf{X}, \boldsymbol{\theta})$ is the logarithmic joint-likelihood, which has been discussed in (2.23).

We note that MAP estimation leads to adding an extra term $\ln q(\boldsymbol{\theta})$, known as the log-prior of the parameters. For $q(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta}; \mathbf{0}, \beta^2 \mathbf{I})$, the log-prior term corresponds to ℓ_2-regularization [4, ch. 11.3, p. 382], while, for $q(\boldsymbol{\theta}) = \prod_{i=1}^{|\mathcal{E}|} \text{Laplace}(\theta_i; 0, \frac{1}{\beta})$, the log-prior corresponds to ℓ_1-regularization [4, ch. 11.4, p. 377], where

$$\text{Laplace}(x; \mu, \beta) := \frac{1}{2\beta} \exp\left(-\frac{|x - \mu|}{\beta}\right), \tag{2.34}$$

is the Laplace pdf. Last, if we assume $q(\boldsymbol{\theta}) = \prod_{i=1}^{|\mathcal{E}|} \mathcal{U}(\theta_i; \alpha, \beta)$, where

$$\mathcal{U}(x; \alpha, \beta) := \begin{cases} \frac{1}{\beta - \alpha}, \; x \in [\alpha, \beta], \\ 0, \text{ o.w.} \end{cases} \tag{2.35}$$

is the uniform pdf, then MAP reduces to ML estimation. Additional details can be found in [4, ch. 4.5.1, p. 119], [6, ch. 5.7, p. 278].

[7] In fact, $\mathbf{X}$ is a random matrix, however, it can be viewed as a random vector by defining the following mapping: $\mathbf{X} \mapsto \text{vec}(\mathbf{X})$. Similarly, we can define $\mathbf{Y} \mapsto \text{vec}(\mathbf{Y})$, for the $\mathbf{Y}$ random matrix.

[8] The denominator of expression (2.32) is known as the marginal likelihood, since $q(\mathbf{Y} \mid \mathbf{X}) = \int_{\mathbb{R}^{|\mathcal{E}|}} q(\mathbf{Y} \mid \mathbf{X}, \boldsymbol{\theta}) dq(\boldsymbol{\theta})$.

Chapter 3

The Back-Propagation Algorithm

Back-propagation is a very important algorithm for training feed-forward neural networks [8]. In this chapter, we focus on computing the gradient of the empirical risk w.r.t. the neural network parameters, that is, $\nabla R_{\mathcal{S}}(\boldsymbol{\theta})$, for some fixed neural network $\mathcal{N}_L$ parameterized by $\boldsymbol{\theta}$ and a training set $\mathcal{S}$.

We present the back-propagation for a certain class of feed-forward neural networks. Of course, this formula can be extended to any feed-forward architecture. Additional information for the back-propagation algorithm can be found in [1, ch. 20.6, p. 277], [6, ch. 5.3, p. 141], [9, ch. 6.5, p. 204], [3, ch. 6.2, p. 142], [10, ch. 4.8, p. 140] and [4, ch. 13.3, p. 434].

3.1 Overview of Back-Propagation

Let us fix a neural network architecture $\mathcal{N}_L$, for some $0 < L < \infty$, parameterized by the vector $\boldsymbol{\theta}$ and let $(\mathbf{x}, \mathbf{y}) \in \mathcal{X} \times \mathcal{Y}$ be an arbitrary input-output training pair. We feed the neural network with the input $\mathbf{x}$ and evaluate the inputs and outputs for each layer as follows:

$$\begin{aligned}
&\mathbf{a}^{(0)}(\mathbf{x}) = \mathbf{x}, \\
&\left.\begin{aligned}
\mathbf{z}^{(\ell)}(\mathbf{x}) &= \mathbf{W}^{(\ell)}\mathbf{a}^{(\ell-1)}(\mathbf{x}) + \mathbf{b}^{(\ell)}, \\
\mathbf{a}^{(\ell)}(\mathbf{x}) &= \boldsymbol{\sigma}_\ell(\mathbf{a}^{(\ell)}(\mathbf{x}))\,,
\end{aligned}\right\} \text{, for } \ell \in [L-1], \\
&\mathbf{z}^{(L)}(\mathbf{x}) = \mathbf{W}^{(L)}\mathbf{a}^{(L-1)}(\mathbf{x}) + \mathbf{b}^{(L)},
\end{aligned} \tag{3.1}$$

where $\boldsymbol{\sigma}_\ell(\mathbf{u}) = \begin{bmatrix}\sigma_\ell(u_1) & \cdots & \sigma_\ell(u_s)\end{bmatrix}^\top$, for some vector $\mathbf{u} \in \mathbb{R}^s$, $s > 0$, and $\sigma_\ell(\cdot)$ is the activation that each neuron computes at hidden layer $\mathcal{V}_\ell$. We define the procedure described in expression (3.1) as the forward-phase of the back-propagation.

Recall that the loss for regression problems is the squared-error, given by

$$f(\boldsymbol{\theta}) = \frac{1}{2}\|\mathbf{z}^{(L)}(\mathbf{x}) - \mathbf{y}\|_2^2. \tag{3.2}$$

Also, recall for the logistic regression, the input-output training pair reduces to $(\mathbf{x}, y)$, because the outputs are categorical and we select the cross-entropy-

error, which is given by

$$f(\boldsymbol{\theta}) = -z_y^{(L)}(\mathbf{x}) + \ln \sum_{k=1}^{c} e^{z_k^{(L)}(\mathbf{x})}, \tag{3.3}$$

since $\hat{\mathbf{y}}_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{z}^{(L)}(\mathbf{x})$, the output of the neural network, is the output of the last layer.

Next, we denote the gradients of the errors w.r.t. each layer $\ell \in [L]$ as

$$\begin{aligned} d_f \mathbf{z}^{(\ell)}(\mathbf{x}) &:= \nabla_{\mathbf{z}^{(\ell)}(\mathbf{x})} f(\boldsymbol{\theta}), \\ d_f \mathbf{a}^{(\ell)}(\mathbf{x}) &:= \nabla_{\mathbf{a}^{(\ell)}(\mathbf{x})} f(\boldsymbol{\theta}), \end{aligned} \tag{3.4}$$

where

$$\begin{aligned} \nabla_{\mathbf{z}^{(\ell)}(\mathbf{x})} f(\boldsymbol{\theta}) &= \left[\frac{\partial f(\boldsymbol{\theta})}{\partial z_1^{(\ell)}(\mathbf{x})} \quad \cdots \quad \frac{\partial f(\boldsymbol{\theta})}{\partial z_{k_\ell}^{(\ell)}(\mathbf{x})} \right]^\top, \\ \nabla_{\mathbf{a}^{(\ell)}(\mathbf{x})} f(\boldsymbol{\theta}) &= \left[\frac{\partial f(\boldsymbol{\theta})}{\partial a_1^{(\ell)}(\mathbf{x})} \quad \cdots \quad \frac{\partial f(\boldsymbol{\theta})}{\partial a_{k_\ell}^{(\ell)}(\mathbf{x})} \right]^\top. \end{aligned} \tag{3.5}$$

Once we define the errors w.r.t. the layers, we begin the backward phase. In this phase, each $d_f \mathbf{z}^{(\ell)}(\mathbf{x})$ and $d_f \mathbf{a}^{(\ell)}(\mathbf{x})$ is computed recursively, in the opposite direction that we did in the forward phase. To start the recursions, we first compute $d_f \mathbf{z}^{(L)}(\mathbf{x})$, the gradient of the error w.r.t. the last layer L, which depends on the choice we made for $f(\boldsymbol{\theta})$. For regression problems, we get

$$d_f \mathbf{z}^{(L)}(\mathbf{x}) = \mathbf{z}^{(L)}(\mathbf{x}) - \mathbf{y}, \tag{3.6}$$

while, for logistic regression, we obtain

$$\begin{aligned} d_f z_j^{(L)}(\mathbf{x}) &= \frac{\partial}{\partial z_j^{(L)}(\mathbf{x})} \left(-z_y^{(L)}(\mathbf{x}) + \ln \sum_{k=1}^{c} e^{z_k^{(L)}(\mathbf{x})} \right) \\ &= -\mathbb{I}[j = y] + \frac{e^{z_j^{(L)}(\mathbf{x})}}{\sum_{k=1}^{c} e^{z_k^{(L)}(\mathbf{x})}} \\ &= -\mathbb{I}[j = y] + g_j \left(\mathbf{z}^{(L)}(\mathbf{x}) \right), \end{aligned} \tag{3.7}$$

since $g_j(\cdot) = [\mathbf{g}(\cdot)]_j$, $j \in [c]$ is the jth coordinate of the softmax function. Next, we present the backward phase of the back-propagation algorithm. The backward phase proceeds with the following recursions

$$d_f \mathbf{z}^{(\ell)}(\mathbf{x}) = d_f \mathbf{a}^{(\ell)}(\mathbf{x}) \odot \boldsymbol{\sigma}'_\ell \left(\mathbf{z}^{(\ell)}(\mathbf{x}) \right), \ \ell \neq L, \tag{3.8}$$

$$d_f \mathbf{a}^{(\ell-1)}(\mathbf{x}) = \left(\mathbf{W}^{(\ell)} \right)^\top d_f \mathbf{z}^{(\ell)}(\mathbf{x}), \tag{3.9}$$

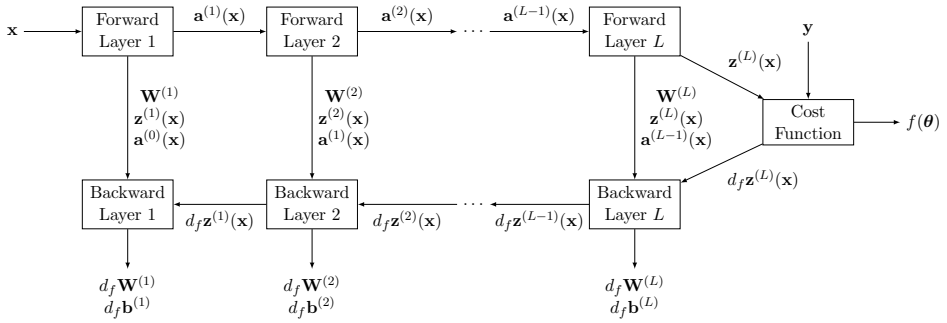


Figure 3.1: A computational graph of the back-propagation algorithm, given a training pair $(\mathbf{x}, \mathbf{y})$. We start with the input in the upper left corner and propagate forward and evaluate the cost (error) function. We also cache the values $\mathbf{z}^{(\ell)}(\mathbf{x})$ and $\mathbf{a}^{(\ell)}(\mathbf{x})$ for all $\ell \in [L]$. Once we compute the error function, we propagate the errors $d_f\mathbf{z}^{(\ell)}(\mathbf{x})$ backward and compute the gradients w.r.t. the network parameters $\mathbf{W}^{(\ell)}$, $\mathbf{b}^{(\ell)}$ for all $\ell \in [L]$. The equations behind the computational graph are given in (3.10) and (3.11).

for $\ell \in [L]$ where $\boldsymbol{\sigma}'_\ell(\mathbf{u}) = \begin{bmatrix}\sigma'_\ell(u_1) & \cdots & \sigma'_\ell(u_s)\end{bmatrix}^\top$, for some vector $\mathbf{u} \in \mathbb{R}^s$, $s > 0$, where $\odot$ denotes the element-wise multiplication operator. Expressions (3.8) and (3.9) will be proved in section [3.2] (see expressions (3.18) and (3.19), respectively). After each term $d_f\mathbf{z}^{(\ell)}(\mathbf{x})$ has been computed, we are able to compute the gradients w.r.t. the weights and the biases. The gradients, for each layer $\ell \in [L]$, can be computed as

$$d_f\mathbf{W}^{(\ell)} = d_f\mathbf{z}^{(\ell)}(\mathbf{x}) \left(\mathbf{a}^{(\ell-1)}(\mathbf{x})\right)^\top, \tag{3.10}$$

$$d_f\mathbf{b}^{(\ell)} = d_f\mathbf{z}^{(\ell)}(\mathbf{x}), \tag{3.11}$$

where

$$d_f\mathbf{W}^{(\ell)} := \begin{bmatrix} \frac{\partial f(\boldsymbol{\theta})}{\partial W^{(\ell)}_{1,1}} & \cdots & \frac{\partial f(\boldsymbol{\theta})}{\partial W^{(\ell)}_{1,k_{\ell-1}}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f(\boldsymbol{\theta})}{\partial W^{(\ell)}_{k_\ell,1}} & \cdots & \frac{\partial f(\boldsymbol{\theta})}{\partial W^{(\ell)}_{k_\ell,k_{\ell-1}}} \end{bmatrix}, \tag{3.12}$$

and

$$d_f\mathbf{b}^{(\ell)} := \begin{bmatrix} \frac{\partial f(\boldsymbol{\theta})}{\partial b^{(\ell)}_1} & \cdots & \frac{\partial f(\boldsymbol{\theta})}{\partial b^{(\ell)}_{k_\ell}} \end{bmatrix}^\top. \tag{3.13}$$

Similarly, expressions (3.10) and (3.11) will be proved in section [3.2] (see expressions (3.16) and (3.17), respectively).

So far, we considered the back-propagation for a single input-output training pair $(\mathbf{x}, \mathbf{y})$. In Fig [3.1], we outline the back-propagation for a single input-output training pair. However, we want to compute the gradient

w.r.t. the empirical risk $R_{\mathcal{S}}(\boldsymbol{\theta})$, where, in this case, $\mathcal{S}$ denotes a full-batch or a mini-batch. We observe that

$$\nabla R_{\mathcal{S}}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\boldsymbol{\theta}), \tag{3.14}$$

where $f_i(\cdot)$ denotes the error for the training pair $(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{S}$, for $i \in [n]$. Therefore, one way to compute $\nabla R_{\mathcal{S}}(\boldsymbol{\theta})$ is to run the equations (3.1), (3.6) for regression or (3.7) for logistic regression, (3.8) and (3.9) for each training pair that belongs in the training set $\mathcal{S}$ and average their results $d\mathbf{W}^{(\ell)}$ and $d\mathbf{b}^{(\ell)}$. In Algorithm 1, we provide a simple implementation of the back-propagation.

Algorithm 1 Back-Propagation

Require: training set: $\mathcal{S} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{n}$, neural network $\mathcal{N}_L$
  for $i = 1, \ldots, n$ **do**
    feed the neural network with $\mathbf{x}_i$ and perform the forward-phase (3.1)
    if regression problem **then**
      compute $d_{f_i}\mathbf{z}^{(L)}(\mathbf{x}_i)$ according to (3.6), where $\mathbf{y} = \mathbf{y}_i$
    else if logistic regression problem **then**
      compute $d_{f_i}\mathbf{z}^{(L)}(\mathbf{x}_i)$ according to (3.7), where $y = y_i$
    end if
    for $\ell = L-1, \ldots, 0$ **do**
      compute $d_{f_i}\mathbf{a}^{(\ell)}(\mathbf{x}_i)$, $d_{f_i}\mathbf{z}^{(\ell)}(\mathbf{x}_i)$ by (3.9) and (3.8), respectively
      compute $d_{f_i}\mathbf{W}^{(\ell+1)}$, $d_{f_i}\mathbf{b}^{(\ell+1)}$ by (3.10) and (3.11), respectively
    end for
    store $\nabla f_i(\boldsymbol{\theta}) = \left[\mathrm{vec}\left(d_{f_i}\mathbf{W}^{(1)}\right); d_{f_i}\mathbf{b}^{(1)}; \cdots; \mathrm{vec}\left(d_{f_i}\mathbf{W}^{(\ell)}\right); d_{f_i}\mathbf{b}^{(\ell)}\right]$
  end for
  compute $\nabla R_{\mathcal{S}}(\boldsymbol{\theta})$ by using (3.14)
  return $\nabla R_{\mathcal{S}}(\boldsymbol{\theta})$

The back-propagation algorithm can be generalized to any feed-forward[1] neural network architecture [4, ch. 13.3.4, p. 441].

3.2 Derivation of Back-Propagation Equations

In this section, we derive the equations of the back-propagation. To do that, we write the forward-phase expressions (3.1) in element-wise fashion as

$$\begin{aligned} z_j^{(\ell)}(\mathbf{x}) &= \sum_{k=1}^{k_{\ell-1}} W_{j,k}^{(\ell)} a_k^{(\ell-1)}(\mathbf{x}) + b_j^{(\ell)}, \\ a_j^{(\ell)}(\mathbf{x}) &= \sigma_\ell\left(z_j^{(\ell)}(\mathbf{x})\right), \end{aligned} \tag{3.15}$$

[1] Recall that the underlying graph of a feed-forward neural network is directed and acyclic.

with $a_j^{(0)}(\mathbf{x}) = x_j$, for each $j \in [k_\ell]$ and $\ell \in [L-1]$. We want to compute the derivatives of the error function $f(\boldsymbol{\theta})$ w.r.t. $W_{j,k}^{(\ell)}$ and $b_j^{(\ell)}$. Applying the chain rule of calculus, we obtain

$$\begin{aligned}\frac{\partial f(\boldsymbol{\theta})}{\partial W_{j,m}^{(\ell)}} &= \sum_{k=1}^{k_\ell} \frac{\partial f(\boldsymbol{\theta})}{\partial z_k^{(\ell)}(\mathbf{x})} \frac{\partial z_k^{(\ell)}(\mathbf{x})}{\partial W_{j,m}^{(\ell)}} \\ &= \frac{\partial f(\boldsymbol{\theta})}{\partial z_j^{(\ell)}(\mathbf{x})} \frac{\partial z_j^{(\ell)}(\mathbf{x})}{\partial W_{j,m}^{(\ell)}} \\ &= \frac{\partial f(\boldsymbol{\theta})}{\partial z_j^{(\ell)}(\mathbf{x})} a_m^{(\ell-1)}(\mathbf{x}),\end{aligned} \tag{3.16}$$

for each $j \in [k_\ell]$, $m \in [k_{\ell-1}]$, since $z_k^{(\ell)}(\mathbf{x})$ depends on $W_{j,m}^{(\ell)}$ only if $k = j$ (other terms reduce to zero) and the term $\frac{\partial z_j^{(\ell)}(\mathbf{x})}{\partial W_{j,m}^{(\ell)}} = a_m^{(\ell-1)}(\mathbf{x})$. Similarly, for the biases, we have

$$\begin{aligned}\frac{\partial f(\boldsymbol{\theta})}{\partial b_j^{(\ell)}} &= \frac{\partial f(\boldsymbol{\theta})}{\partial z_j^{(\ell)}(\mathbf{x})} \frac{\partial z_j^{(\ell)}(\mathbf{x})}{\partial b_j^{(\ell)}} \\ &= \frac{\partial f(\boldsymbol{\theta})}{\partial z_j^{(\ell)}(\mathbf{x})},\end{aligned} \tag{3.17}$$

for each $j \in [k_\ell]$, since $\frac{\partial z_j^{(\ell)}(\mathbf{x})}{\partial b_j^{(\ell)}} = 1$.

Again, we apply the chain rule to express the derivatives of the error function w.r.t. the pre-activations $z_j^{(\ell)}(\mathbf{x})$ and the activations $a_j^{(\ell)}(\mathbf{x})$, such as

$$\begin{aligned}\frac{\partial f(\boldsymbol{\theta})}{\partial z_j^{(\ell)}(\mathbf{x})} &= \sum_{k=1}^{k_\ell} \frac{\partial f(\boldsymbol{\theta})}{\partial a_k^{(\ell)}(\mathbf{x})} \frac{\partial a_k^{(\ell)}(\mathbf{x})}{\partial z_j^{(\ell)}(\mathbf{x})} \\ &= \frac{\partial f(\boldsymbol{\theta})}{\partial a_j^{(\ell)}(\mathbf{x})} \frac{\partial a_j^{(\ell)}(\mathbf{x})}{\partial z_j^{(\ell)}(\mathbf{x})} \\ &= \frac{\partial f(\boldsymbol{\theta})}{\partial a_j^{(\ell)}(\mathbf{x})} \sigma_\ell' \left(z_j^{(\ell)}(\mathbf{x}) \right),\end{aligned} \tag{3.18}$$

for each $\ell \in [L-1]$ and

$$\begin{aligned}\frac{\partial f(\boldsymbol{\theta})}{\partial a_j^{(\ell-1)}(\mathbf{x})} &= \sum_{k=1}^{k_\ell} \frac{\partial f(\boldsymbol{\theta})}{\partial z_k^{(\ell)}(\mathbf{x})} \frac{\partial z_k^{(\ell)}(\mathbf{x})}{\partial a_j^{(\ell-1)}(\mathbf{x})} \\ &= \sum_{k=1}^{k_\ell} \frac{\partial f(\boldsymbol{\theta})}{\partial z_k^{(\ell)}(\mathbf{x})} W_{k,j}^{(\ell)},\end{aligned} \tag{3.19}$$

for each $\ell \in [L]$, since

$$\begin{aligned}\frac{\partial z_k^{(\ell)}(\mathbf{x})}{\partial a_j^{(\ell-1)}(\mathbf{x})} &= \frac{\partial}{\partial a_j^{(\ell-1)}(\mathbf{x})}\left(\sum_{u=1}^{k_\ell} W_{k,u}^{(\ell)} a_u^{(\ell-1)}(\mathbf{x}) + b_k^{(\ell)}\right) \\ &= W_{k,j}^{(\ell)}.\end{aligned} \tag{3.20}$$

Using the notation $d_f\mathbf{W}^{(\ell)}$, $d_f\mathbf{b}^{(\ell)}$, $d_f\mathbf{z}^{(\ell)}(\mathbf{x})$ and $d_f\mathbf{a}^{(\ell)}(\mathbf{x})$ as we defined earlier, expressions (3.16), (3.17) become (3.10) and (3.11), while expressions (3.18) and (3.19) become (3.8) and (3.9), respectively.

3.3 Computational Complexity of Back-Propagation

One of the most important aspects of back-propagation is its computational complexity [6, ch. 5.3.3, p. 246]. The computational cost of the back-propagation is dominated by the cost of the matrix multiplications. During the forward-phase of the algorithm, we perform $O(|\mathcal{E}|)$ scalar multiplications, $O(|\mathcal{E}| + |\mathcal{V}|)$ additions and $O(|\mathcal{V}|)$ element-wise evaluations of the activation. Typically, the number of edges $|\mathcal{E}|$ of the underlying graph $\mathcal{G}$ is much larger than the number of nodes $|\mathcal{V}|$, therefore the overall bulk of computation of the forward-phase of the algorithm is dominated[2] by the matrix multiplication cost $O(|\mathcal{E}|)$.

During the backward-phase, we multiply by the transpose of each weight matrix, which has computational cost $O(|\mathcal{E}|)$. Next, we perform element-wise operations of cost $O(|\mathcal{V}|)$ Therefore, since $|\mathcal{E}|$ is much larger than $|\mathcal{V}|$, the computational cost for the backward-phase of also $O(|\mathcal{E}|)$, hence, the computational cost of the back-propagation algorithm is $O(|\mathcal{E}|)$.

[2]We assume that each addition and scalar multiplication has (approximately) the same computational cost of $O(1)$ operations. Additionally, we assume, for simplification, that each activation is linear ($\sigma_\ell(x) = x$, $\forall \ell \in [L-1]$). Therefore, the computational cost for the activation is insignificant.

Chapter 4

Training Deep Neural Networks

The problem of minimizing continuous and differentiable functions of several variables has been widely studied [11, 12, 13] and many of the conventional minimization algorithms are directly applicable to the training of (deep) neural networks. All these algorithms require to specify a starting point $\boldsymbol{\theta}^{(0)}$, which corresponds to the initialization of the parameters of an arbitrary feed-forward neural network $\mathcal{N}_L$. Let $\mathcal{A}(\mathcal{N}_L)$ denote an iterative algorithm that updates the network parameters by using the following formula:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + \Delta\boldsymbol{\theta}^{(t)}, \tag{4.1}$$

where $t \geq 0$ denotes the current iteration, while vector $\Delta\boldsymbol{\theta}^{(t)}$ is some incremental term, which could likely decrease the value of the objective function in the next iteration. We denote the trajectories of an iterative algorithm $\mathcal{A}(\mathcal{N}_L)$ as $\Theta = \{\boldsymbol{\theta}^{(t)}\}_{t=0}^{T}$, where $0 < T < \infty$ denotes the iteration when the algorithm is terminated. We denote by $\boldsymbol{\theta}^*$ a (sub)-optimal solution of the minimization problem. If $\boldsymbol{\theta}^* \in \Theta$, we say that the training algorithm $\mathcal{A}(\cdot)$ found a (sub)-optimal solution. If $\lim_{t\to\infty} \boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^*$, we say that algorithm $\mathcal{A}(\cdot)$ converged to (sub)-optimal solution $\boldsymbol{\theta}^*$.

4.1 First-Order Optimization Methods

In this section, we consider iterative optimization methods that use only the first-order derivatives of the objective function. Additional information for first-order optimization methods can be found in [9, ch. 8.3, p. 294], [4, ch. 8.4, p. 288], [3, ch. 5.5, p. 104], [7, ch. 6.3, p. 265] and [14] .

4.1.1 Gradient Descent

The gradient descent, sometimes referred to as steepest descent, updates iteratively the parameters of a neural network by using the formula (4.1), such that, at step t, we move a short distance in the direction of the greatest rate of decrease of the objective function. This direction is the negative gradient evaluated at iteration $\boldsymbol{\theta}^{(t)}$:

$$\Delta\boldsymbol{\theta}^{(t)} = -\alpha_t \nabla R_{\mathcal{S}}(\boldsymbol{\theta}^{(t)}), \tag{4.2}$$

where $\alpha_t > 0$ is known as step size or learning rate. In practice, a constant step size $\alpha_t = \alpha$ is used to simplify the training procedure, however, there is a serious difficulty with this approach. For example, if the step size α is too large, then the algorithm may not converge. Conversely, if α is too small, the algorithm will lead to painfully slow convergence, hence, very long computation time.

4.1.2 Stochastic and Batch Optimization

Optimization methods for machine learning fall into two broad categories. We refer to them as stochastic and batch. The prototypical stochastic optimization method is the stochastic gradient [15], which, in the context of minimizing the empirical risk, is given by

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \alpha_t \nabla f_i(\boldsymbol{\theta}^{(t)}), \tag{4.3}$$

where $f_i(\boldsymbol{\theta})$ is the sample loss over the training pair $(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{X} \times \mathcal{Y}$, for some random $i \in \{1, \ldots, n\}$, and $\alpha_t > 0$ is the step size at iteration $t \geq 0$. Obviously, the stochastic step $-\alpha_t \nabla f_i(\boldsymbol{\theta}^{(t)})$, for some random $i \in [n]$, is computationally less expensive that the full gradient step $-\alpha_t \nabla R_{\mathcal{S}}(\boldsymbol{\theta})$.

In practice, it is most common to do something in between. Instead of computing the full gradient, or a stochastic step, we compute the gradient over a mini-batch [3, ch. 5.5, p. 106]. We decompose the training set $\mathcal{S}$ into $\frac{n}{n_b}$ mini-batches, denoted by $\mathcal{S}_m = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i \in \mathcal{I}_m}$, where the indexes are given by $\mathcal{I}_m = \{(m-1)n_b + 1, \ldots, mn_b\}$, $\forall m \in [\frac{n}{n_b}]$, and $n_b > 0$ denotes the batch-size[1]. One complete pass over the training data is called an epoch (obviously, an epoch requires $\frac{n}{n_b}$ iterations). The objective function over the mth batch is the empirical risk over $\mathcal{S}_m$, that is,

$$R_{\mathcal{S}_m}(\boldsymbol{\theta}) = \frac{1}{|\mathcal{S}_m|} \sum_{i \in \mathcal{I}_m} f_i(\boldsymbol{\theta}). \tag{4.4}$$

The iterative formula for the batch-version of the gradient descent method is given by

$$\begin{aligned} \boldsymbol{\theta}^{(t+1)} &= \boldsymbol{\theta}^{(t)} - \alpha_t \nabla R_{\mathcal{S}_m}(\boldsymbol{\theta}^{(t)}) \\ &= \boldsymbol{\theta}^{(t)} - \frac{\alpha_t}{|\mathcal{S}_m|} \sum_{i \in \mathcal{I}_m} \nabla f_i(\boldsymbol{\theta}^{(t)}). \end{aligned} \tag{4.5}$$

Next, we implement a generalized gradient-based training procedure for training a feed-forward neural network $\mathcal{N}_L$. For each epoch, we randomly shuffle the training data, and thereafter we divide it into mini-batches in an ordered manner, as we explained earlier.

[1] Note that $|\mathcal{S}_m| = |\mathcal{I}_m| = n_b$, $\forall m \in [\frac{n}{n_b}]$.

Algorithm 2 Generalized (batched) Stochastic Gradient Training Method

Require: training set: $\mathcal{S} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{n}$, numEpochs > 0 , $n_b > 0$
  $t = 0$
  initialize $\boldsymbol{\theta}^{(0)}$
  for epoch $= 0, \ldots,$ numEpochs **do**
    randomly shuffle the training data set $\mathcal{S}$ and generate $\mathcal{S}_m$, $\forall m \in [\frac{n}{n_b}]$
    for $m = 1, \ldots, \frac{n}{n_b}$ **do**
      Compute $\nabla R_{\mathcal{S}_m}(\boldsymbol{\theta}^{(t)})$ via back-propagation
      $\boldsymbol{\theta}^{(t+1)} =$ iteration_step($\boldsymbol{\theta}^{(t)}$, α_t, $\nabla R_{\mathcal{S}_m}(\boldsymbol{\theta}^{(t)})$, $\cdots$)
      $t = t + 1$
    end for
  end for

The generality of Algorithm 2 can be seen in various ways. First, any gradient-based training algorithm can be implemented by defining an appropriate iteration step procedure. Different iteration step procedures generally have different hyperparameters. Second, $\nabla R_{\mathcal{S}_m}(\boldsymbol{\theta}^{(t)})$ can either represent a stochastic gradient if each $\mathcal{S}_m$ has exactly one training pair or a batch gradient (or full gradient) according to the mechanism that splits the training data $\mathcal{S}$ into batches $\mathcal{S}_m$. For the batch-version of the gradient descent formula, the iteration step procedure can be implemented by using the expression (4.5).

4.1.3 Line Search

The algorithms which we already presented in the previous sections involve a sequence of step-sizes $\{\alpha_t\}_{t=1}^{T}$ for some $T > 0$. The process of finding a step α_t is called line search, since it is a minimization procedure on the one-dimensional $s(\alpha) = g(\boldsymbol{\theta}^{(t)} + \alpha\Delta\boldsymbol{\theta}^{(t)})$, where $g : \mathbb{R}^{|\mathcal{E}|} \to \mathbb{R}$ is any objective function we wish to minimize. Next, we describe the most popular step-size selection rules:

- Constant step. We choose $\alpha_t = \alpha > 0$ for any $t \in [T]$.
- Exact line search. We solve the minimization problem

$$\alpha_t \in \operatorname{argmin}_{\alpha>0} g\left(\boldsymbol{\theta}^{(t)} + \alpha\Delta\boldsymbol{\theta}^{(t)}\right) . \tag{4.6}$$

- Backtracking. The choice of α_t is done by the following procedure. Fist, we initialize α_t to some initial guess α_0. Then, while the condition $g\left(\boldsymbol{\theta}^{(t)}\right) - g\left(\boldsymbol{\theta}^{(t)} + \alpha_t\Delta\boldsymbol{\theta}^{(t)}\right) < -c\alpha_t\nabla g\left(\boldsymbol{\theta}^{(t)}\right)^{\top}\Delta\boldsymbol{\theta}^{(t)}$, for some $c > 0$, holds true, we set $\alpha_t \leftarrow \beta\alpha_t$. In other words, the step size α_t is chosen such that the condition

$$g\left(\boldsymbol{\theta}^{(t)}\right) - g\left(\boldsymbol{\theta}^{(t)} + \alpha_t\Delta\boldsymbol{\theta}^{(t)}\right) \geq -c\alpha_t\nabla g\left(\boldsymbol{\theta}^{(t)}\right)^{\top}\Delta\boldsymbol{\theta}^{(t)} \tag{4.7}$$

is satisfied.

For sufficiently small α_t, we can approximate $s(\alpha_t)$ by using first order Taylor (linear extrapolation). Therefore, we can write

$$\begin{aligned} s(\alpha_t) = g\left(\boldsymbol{\theta}^{(t)} + \alpha_t \Delta\boldsymbol{\theta}^{(t)}\right) &\approx g\left(\boldsymbol{\theta}^{(t)}\right) + \alpha_t \nabla g\left(\boldsymbol{\theta}^{(t)}\right)^\top \Delta\boldsymbol{\theta}^{(t)} \\ &\leq g\left(\boldsymbol{\theta}^{(t)}\right) + c\alpha_t \nabla g\left(\boldsymbol{\theta}^{(t)}\right)^\top \Delta\boldsymbol{\theta}^{(t)}, \end{aligned} \tag{4.8}$$

since $c \in (0, \frac{1}{2})$ and $\nabla g\left(\boldsymbol{\theta}^{(t)}\right)^\top \Delta\boldsymbol{\theta}^{(t)} < 0$, because, $\Delta\boldsymbol{\theta}^{(t)}$ is a descent direction. Expression (4.8) states that the condition (4.7) eventually terminates, if a_t is sufficiently small ($\beta \in (0, 1)$). See [12, ch. 9, p. 464] for further details.

We have already discussed the limitations of the constant step option. Its main advantage is its simplicity, but it is unclear how to choose the appropriate non-negative a. The exact line search option may seem at first sight attractive, however, it is not always possible to solve the problem (4.6), therefore, it is not a suitable option. The backtracking option can be seen as a compromise. We neither choose a constant step size nor we solve the exact line search minimization problem, but we find a sufficiently good step-size, satisfying the condition (4.7).

4.1.3.1 Armijo line search

By using the empirical risk, $R_{\mathcal{S}_m}(\boldsymbol{\theta})$, over the mini-batch $\mathcal{S}_m \subseteq \mathcal{S}$ as the objective function and $\Delta\boldsymbol{\theta} = -\nabla R_{\mathcal{S}_m}(\boldsymbol{\theta})$, expression (4.7) becomes

$$R_{\mathcal{S}_m}\left(\boldsymbol{\theta}^{(t)}\right) - R_{\mathcal{S}_m}\left(\boldsymbol{\theta}^{(t)} - \alpha_t \nabla R_{\mathcal{S}_m}\left(\boldsymbol{\theta}^{(t)}\right)\right) \geq c\alpha_t \|\nabla R_{\mathcal{S}_m}\left(\boldsymbol{\theta}^{(t)}\right)\|_2^2. \tag{4.9}$$

The Armijo line-search selects a step-size, such that condition (4.9) is satisfied [16, ch. 3.1, p. 3]. We note that Armijo condition makes use only on additional function (and not gradient) evaluations, which, in the context of neural networks, corresponds to additional forward passes, involving only the current batch $\mathcal{S}_m$, for some $m \in [\frac{n}{n_b}]$.

4.1.4 Accelerated Methods

In this section, we use the gradient information in a more clever way to potentially achieve faster convergence rates. One simple technique is to add a momentum term in the gradient descent formula.

4.1.4.1 The Heavy-Ball Method

One of the most elementary accelerated methods is the heavy-ball method of Polyak [17]. The iterative formula of this method has the form

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \alpha_t \nabla g\left(\boldsymbol{\theta}^{(t)}\right) + \mu_t \left(\boldsymbol{\theta}^{(t)} - \boldsymbol{\theta}^{(t-1)}\right), \tag{4.10}$$

where $g(\cdot)$ is any objective function we wish to minimize w.r.t. $\boldsymbol{\theta}$, α_t is the step-size and μ_t is known as the momentum term. To better understand the effect of the momentum, we rewrite equation (4.10) as

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + \Delta\boldsymbol{\theta}^{(t)}, \tag{4.11}$$

$$\Delta\boldsymbol{\theta}^{(t)} = -\alpha_t \nabla g\left(\boldsymbol{\theta}^{(t)}\right) + \Delta\boldsymbol{\mu}^{(t)}, \tag{4.12}$$

$$\Delta\boldsymbol{\mu}^{(t)} = \mu_t\left(\boldsymbol{\theta}^{(t)} - \boldsymbol{\theta}^{(t-1)}\right). \tag{4.13}$$

4.1.4.2 Nesterov Accelerated Gradient

We now describe Nesterov's method for accelerating the gradient iterative formula [18]. Each iteration of this method has the form

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \alpha_t \nabla g\left(\boldsymbol{\theta}^{(t)} + \mu_t\left(\boldsymbol{\theta}^{(t)} - \boldsymbol{\theta}^{(t-1)}\right)\right) + \mu_t\left(\boldsymbol{\theta}^{(t)} - \boldsymbol{\theta}^{(t-1)}\right), \tag{4.14}$$

where α_t and μ_t are the hyperparameters that should be defined. Note the similarity to the heavy ball formula (4.10). The only difference is that in (4.10) the gradient of the objective function is evaluated at $\boldsymbol{\theta}^{(t)}$, whereas, in (4.14), the gradient is evaluated at $\boldsymbol{\theta}^{(t)} + \mu_t\left(\boldsymbol{\theta}^{(t)} - \boldsymbol{\theta}^{(t-1)}\right)$. For implementation purposes, it is more convenient to introduce an auxiliary sequence $\{\boldsymbol{y}^{(t)}\}$ and rewrite the formula (4.14) as follows

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{y}^{(t)} - \alpha_t \nabla g\left(\boldsymbol{y}^{(t)}\right), \tag{4.15}$$

$$\boldsymbol{y}^{(t+1)} = \boldsymbol{\theta}^{(t+1)} + \mu_t\left(\boldsymbol{\theta}^{(t+1)} - \boldsymbol{\theta}^{(t)}\right), \tag{4.16}$$

where $\boldsymbol{y}^{(0)}$ is randomly initialized (random guess) and α_t and μ_t are the hyperparameters. We set $\boldsymbol{\theta}^{(0)} = \boldsymbol{y}^{(0)}$. At iteration t, we can select an appropriate α_t via line search (backtracking). Furthermore, for the choice of μ_t, we can use the variant of Nesterov's acceleration for the (weaken) convex or the strongly convex case. For the (weaken) convex case, we define μ_t with reference to another scalar sequence λ_t in the following manner:

$$\begin{aligned} \lambda_{t+1} &= \frac{1}{2}\left(1 + \sqrt{1 + 4\lambda_t^2}\right), \\ \mu_t &= \frac{\lambda_t - 1}{\lambda_{t+1}}, \end{aligned} \tag{4.17}$$

with $\lambda_0 = 0$. Additional information for accelerated gradient methods can be found at [19, ch. 6, p. 31]. In the context of neural networks, we can use Nesterov acceleration without additional hyper-parameters by using line search and expressions (4.17). According to [16, ch. 6.2, p. 7], we can use, for example, Armijo line search and expressions (4.17) to train a neural network by using Nesterov acceleration.

4.2 Difficulties on Training Deep Neural Networks

4.2.1 Generalization: Underfitting and Overfitting

Once a neural network $\mathcal{N}_L$ is trained, it will return a model that predicts an output for any new input we feed to it. Of course, the model depends on the training set $\mathcal{S}$, since if we train the same neural network using a different training dataset, this would typically result in a different model with probably different predictions. The central challenge is that our methods must perform well on unseen data, not just those on which our model was trained. This ability is called generalization. To test the generalization ability of our model, we split the dataset into a training set and a (hold-out) validation set.

Typically, we estimate the generalization error by measuring the performance of a neural network over the validation set, in which the learner, or the neural network, has access only for evaluation[2]. One interesting question that arises is how we can affect the generalization performance on the validation set when the learner is trained by using the training set. See [3, ch. 4.2, p. 59] for further details.

The two factors that determine how well a training algorithm perform are the ability to:

- Make the training error sufficiently small.
- Make the gap between training error and the validation error small (generalization gap).

These two factors correspond to the two challenges we encounter in training neural networks: underfitting and overfitting. We experience underfitting when the neural network is not able to obtain a sufficiently low error value on the training set and overfitting when the generalization gap is too large.

Next, we focus on ways that prevent overfitting. This is crucial, since a deep neural network may contain millions of parameters.

4.2.1.1 Early Stopping

Perhaps, the simplest way to prevent overfitting is to use a heuristic early-stopping mechanism that terminates the training procedure when the generalization error starts to increase, hence, the gap between the training error and the generalization error begins to increase.

[2]We compute the empirical risk over the validation set and we track the errors as the training of the neural network proceeds.

4.2.1.2 Regularization

One popular way to reduce overfitting is to use a regularization term in the objective. The regularized objective function is the regularized empirical risk, which is given by

$$\tilde{R}_{\mathcal{S}}(\boldsymbol{\theta}) = R_{\mathcal{S}}(\boldsymbol{\theta}) + \lambda\Omega(\boldsymbol{\theta}), \tag{4.18}$$

where $\lambda > 0$ is the penalty term that controls the strength of the regularization $\Omega(\boldsymbol{\theta})$. Two common regularizations are the ℓ_1 and ℓ_2 regularization.

The gradient of ℓ_2-regularization is given by

$$\nabla\tilde{R}_{\mathcal{S}}(\boldsymbol{\theta}) = \nabla R_{\mathcal{S}}(\boldsymbol{\theta}) + \lambda\boldsymbol{\theta}. \tag{4.19}$$

Note that the gradient $\nabla R_{\mathcal{S}}(\boldsymbol{\theta})$ can be computed via the back-propagation. The iterative update of the parameters can be expressed as

$$\begin{aligned} \boldsymbol{\theta}^{(t+1)} &= \boldsymbol{\theta}^{(t)} - \alpha_t \left(\nabla R_{\mathcal{S}}(\boldsymbol{\theta}^{(t)}) + \lambda\boldsymbol{\theta}^{(t)} \right) \\ &= (1 - \alpha_t\lambda)\boldsymbol{\theta}^{(t)} - \alpha_t \nabla R_{\mathcal{S}}(\boldsymbol{\theta}^{(t)}). \end{aligned} \tag{4.20}$$

We can see that the regularized empirical risk has modified the update rule of the parameters.

Another common regularization strategy is the ℓ_1-regularization. In this case, the subdifferential of the regularized empirical risk is given by

$$\partial\tilde{R}_{\mathcal{S}}(\boldsymbol{\theta}) = \nabla R_{\mathcal{S}}(\boldsymbol{\theta}) + \lambda\partial\|\boldsymbol{\theta}\|_1. \tag{4.21}$$

The iterative steps of the parameters are given by

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \alpha_t \left(\nabla R_{\mathcal{S}}(\boldsymbol{\theta}^{(t)}) + \lambda \text{sign}(\boldsymbol{\theta}^{(t)}) \right), \tag{4.22}$$

where $\text{sign}(\boldsymbol{\theta}) \in \partial\|\boldsymbol{\theta}\|_1$.

4.2.2 Parameter Initialization

Since the objective function for the (deep) neural network is non-convex, parameters initialization plays a significant role on how effectively the optimization algorithm can train the neural network, as well as what kind of solutions it ends up with. Therefore, parameter initialization strategies may affect both the training and the generalization. The understanding of how the initial points affect the generalization is quite primitive and there is not much guidance on how to select initial points from the viewpoint of generalization. Hence, most modern initialization strategies are heuristics. Next, we present some common heuristic strategies that are used for parameter initialization.

One heuristic strategy is to sample weights independently from a uniform distribution, $W_{i,j}^{(\ell)} \sim \mathcal{U}(-\alpha_\ell, \alpha_\ell)$ for $i \in [k_\ell]$, $j \in [k_{\ell-1}]$ and $\ell \in [L]$, where

$$\alpha_\ell = \sqrt{\frac{6}{k_{\ell-1} + k_\ell}}, \tag{4.23}$$

with mean 0 and variance[3] $\alpha_\ell^2/3$. This is known as Xavier-initialization, and has been proposed in [20]. If we use $\alpha_\ell = \sqrt{\frac{3}{k_{\ell-1}}}$, the initialization method is called Kaiming-initialization, and has been proposed in [21]. We note that it is not necessary to use a Uniform distribution for the weight initialization. Gaussian distribution is also a common choice for weight initialization. For example, we can sample weights with Gaussian distribution, $W_{i,j}^{(\ell)} \sim \mathcal{N}(0, \beta_\ell^2)$, where $\beta_\ell^2 = \frac{2}{k_{\ell-1}+k_\ell}$, for Xavier-initialization or $\beta_\ell^2 = \frac{2}{k_{\ell-1}}$ for Kaiming-initialization.

Although the above derivations assume that neurons compute linear activations, it has been empirically observed than these initialization techniques can work well even when neurons compute non-linear activations [4, ch. 13.4.5, p. 449].

4.2.3 Other Difficulties

To complete this section, we briefly report some other difficulties that may occur, while training deep neural networks.

The training of a deep neural network can be extremely difficult, since the distribution of each input layer changes during the training, as the parameter vector $\boldsymbol{\theta}$ changes. In the context of neural networks, we refer to the change of the distribution of the internal nodes of a deep neural network, during the training phase, as Internal Covariate Shift. The use of Batch Normalization layers reduces the Internal Covariate Shift and accelerates the training of a deep neural network [22].

When training deep neural networks, the gradient of the objective function tends to become either very small (vanishing-gradient) or very large (exploding-gradient) [4, ch. 13.4.2, p. 443]. We can reduce the exploding gradient problem by using gradient clipping. The vanishing gradient problem is more difficult and depends on the initialization and the architecture of the neural network. One solution for the vanishing gradient problem is to use residual neural networks or ResNets [23, 24, 25].

[3] If $X \sim \mathcal{U}(\alpha, \beta)$, then $\mathbb{E}[X] = \frac{\beta+\alpha}{2}$ and $\mathbb{V}[X] = \frac{(\beta-\alpha)^2}{12}$.

Chapter 5

Experiments

In this chapter, we present our experimental results. In Section [5.1], we describe the experimental setup. In Section [5.2], we showcase the convergence and generalization properties of several (simple) neural network architectures with different size and depth, over real-world datasets, which can be found in the following page[1]. Last, in Section [5.3], we report our observations.

5.1 Experimental Setup

5.1.1 Datasets

We focus on tabular datasets for multi-classification, where $\mathbf{X} \in \mathbb{R}^{n \times d}$ and $\mathbf{y} \in \{1, \ldots, c\}^n$, with $n < 15\text{K}$. We estimate the performance of several neural network architectures by using k-fold cross-validation as it is presented in [3, ch. 4.2, p. 61]. Unless it it stated otherwise, we select $k = 5$ (five splits).

5.1.2 Architectures

The configuration of the neural network architecture is denoted by $\mathcal{N}_L^{\text{relu}} = (\mathcal{V}, \mathcal{E}, \{\sigma_\ell\}_{\ell=1}^{L-1})$, where $k_0 = d$ and $k_L = c$, therefore, $|\mathcal{V}_0| = d + 1$ and $|\mathcal{V}_L| = c$, are the input and output layer dimensions. For simplicity, each hidden layer consists of a fixed number of neurons. Let $m > 0$ denote the number of neurons for each hidden layer, therefore $|\mathcal{V}_\ell| = m+1$, $\forall \ell \in [L-1]$. Furthermore, each hidden layer computes ReLU activations, hence, $\sigma_\ell(x) = \text{ReLU}(x)$, $\forall \ell \in [L-1]$.

For each experiment, we generate $\mathcal{N}_1^{\text{relu}}$, $\mathcal{N}_2^{\text{relu}}$, $\mathcal{N}_3^{\text{relu}}$, $\mathcal{N}_4^{\text{relu}}$, $\mathcal{N}_8^{\text{relu}}$ and $\mathcal{N}_{16}^{\text{relu}}$, which, for simplicity, are denoted by "NNet1", "NNet2", "NNet3", "NNet4", "NNet8" and "NNet16", respectively. For a given classification experiment, we observe that k_0 and k_L are fixed and m is a hyperparameter w.r.t. an architecture, which determines the width and the size of the neural network. The width of the architecture $\mathcal{N}_L^{\text{relu}}$ is m, and the size of the underlying graph is $|\mathcal{E}| = m(k_0+k_L+1)+k_L+(L-2)m(m+1)^2$. The hyperparameter m is appropriately selected via trial-and-error ("baby-sitting").

[1] `https://archive.ics.uci.edu/`.

[2] Note that $|\mathcal{E}| = \sum_{\ell=1}^{L} k_\ell(k_{\ell-1}+1) = \sum_{\ell \in \{1,L\}} k_\ell(k_{\ell-1}+1) + \sum_{\ell \in \{2,\ldots L-1\}} m(m+1)$.

We consider the uniform (Kaiming) initialization strategy for sampling $\boldsymbol{\theta}^{(0)}$ (initial guess).

5.1.3 Objective Functions and Metrics

From an optimization perspective, we consider the weighted[3] empirical risk as the objective function, which is defined as

$$R_{\mathcal{S},\mathbf{w}}(\boldsymbol{\theta}) := -\frac{1}{n}\sum_{i=1}^{n} w_{y_i} \ln g_{y_i}(\hat{\mathbf{y}}_{\boldsymbol{\theta}}(\mathbf{x}_i)), \tag{5.1}$$

where $\mathbf{w} = \begin{bmatrix} w_1 & \dots & w_c \end{bmatrix}^\top \in \mathbb{R}^c$ is a non-trainable weight vector, which tries to "balance"[4] the objective function to the minority classes, $g_j(\mathbf{u}) = [\mathbf{g}(\mathbf{u})]_j, \mathbf{u} \in \mathbb{R}^s$, $s > 0$, for $j \in [c]$, and $\mathcal{S} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ can be either a training set or a validation set of size $n > 0$. Note that $R_{\mathcal{S},\mathbf{1}}(\boldsymbol{\theta}) = R_{\mathcal{S}}(\boldsymbol{\theta})$, results to the non-weighted empirical risk, which we presented in Section [2.4.2]. We also consider the regularized objective (ℓ_2-regularization), which is given by

$$\tilde{R}_{\mathcal{S},\mathbf{w}}(\boldsymbol{\theta}) := R_{\mathcal{S},\mathbf{w}}(\boldsymbol{\theta}) + \frac{\lambda}{2}\|\boldsymbol{\theta}\|_2^2. \tag{5.2}$$

In order to benchmark the performance for both the non-regularized and the regularized case, over unseen data (generalization), we define the "Train Loss" and the "Validation Loss" as the value of the objective function without regularization (5.1), over the training set and the validation set, respectively. In a similar manner, we define the "Train Accuracy" and the "Validation Accuracy". The accuracy metric is defined as

$$\text{accuracy}_{\mathcal{S}}(\boldsymbol{\theta}) := \frac{1}{n}\sum_{i=1}^{n} \mathbb{I}[y_i = \hat{y}_{\boldsymbol{\theta}}(\mathbf{x}_i)], \tag{5.3}$$

where $\mathcal{S} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$, can be either a training set or a validation set of size $n > 0$. The accuracy metric measures the performance of a model based on the true predictions on a given dataset $\mathcal{S} \neq \emptyset$.

5.1.4 Training Procedure

We consider two cases: full-batch and (stochastic) mini-batch optimization. In the full-batch case, we train the neural networks using Gradient Descent with Armijo line-search and Nesterov acceleration, denoted by GD(Armijo+Nesterov). In the mini-batch case, we train the neural networks

[3] `https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html`.

[4] Inspired by [26, ch. 4.2, p. 8], we "balance" the objective function according to the proposed heuristic: For a dataset $\mathcal{S} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n, n > 0$, and $y_i \in [c]$, the weight corresponding to the jth class is given by $w_j = \frac{n}{c\sum_{i=1}^n \mathbb{I}[y_i=j]}$.

using Stochastic Gradient Descent with Armijo line-search, without acceleration, denoted by SGD(Armijo). We also consider $n_b = 50$ (batch-size). The training procedure is summarized in Algorithm 2.

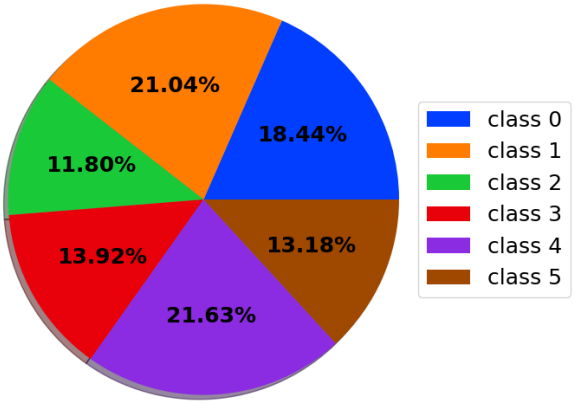
5.2 Experimental results

We showcase our experimental results amongst three multi-classification datasets. In Fig [5.1], we present the population (per class) for each dataset.

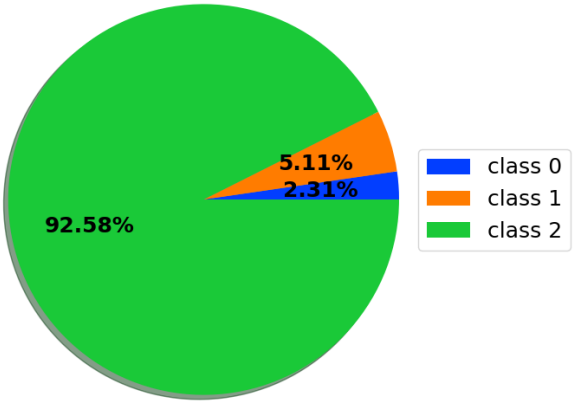
We consider the dataset "Gas Sensor Array Drift Dataset at Different Concentrations" [27], denoted by GSAD, which consists of 128 attributes ($d = 128$) and has six classes ($c = 6$). The task is to detect one of the six different gases, namely Ammonia, Acetaldehyde, Acetone, Ethylene, Ethanol, and Toluene. This dataset consists of $n = 13910$ samples.

Next, we consider the dataset "A Thyroid database suited for training ANNs" [28], denoted by Thyroid Disease, which consists of 21 attributes ($d = 21$) and has three classes ($c = 3$). The task is to determine from the data, whether a person suffers from the thyroid disease (Hypothyroid, Hyperthyroid) or not (Normal). The dataset is in-balanced (only 8% of our population are patients), therefore, a good classifier must be better than 92%. This dataset consists of $n = 7200$ samples.

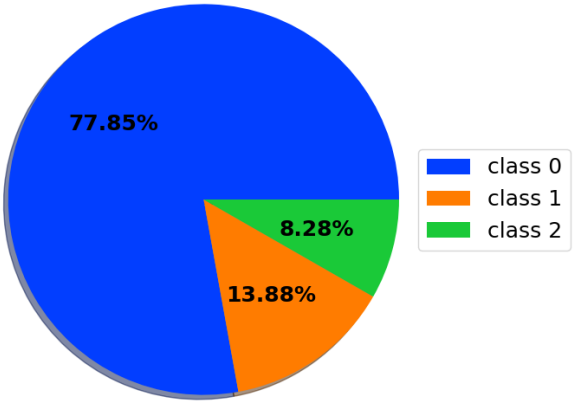
Last, we consider the dataset "Cardiotocography" [29], denoted by CTG, which consists of 23 attributes ($d = 23$) and has three classes ($c = 3$). The task here is to determine the fetal state of a fetus as Normal, Suspect or Pathological (this dataset is also in-balanced). This dataset consists of $n = 2126$ samples.



(a) GSAD dataset, totally $n = 13910$ samples

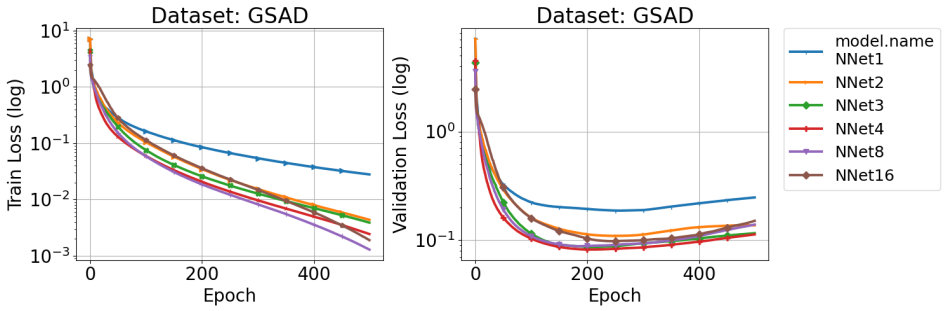


(b) Thyroid Disease dataset, totally $n = 7200$ samples

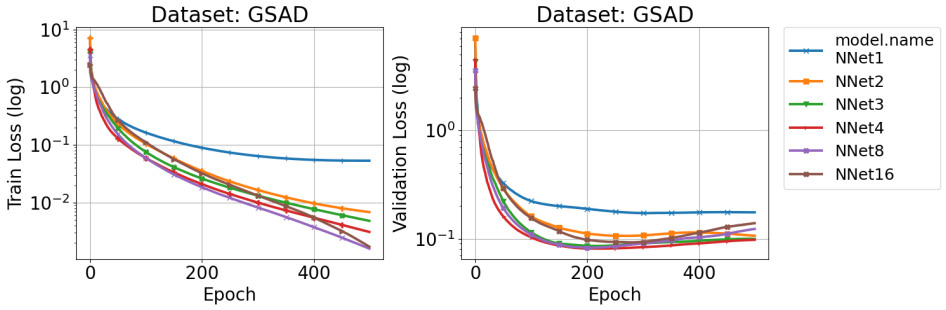


(c) CTG dataset, totally $n = 2126$ samples

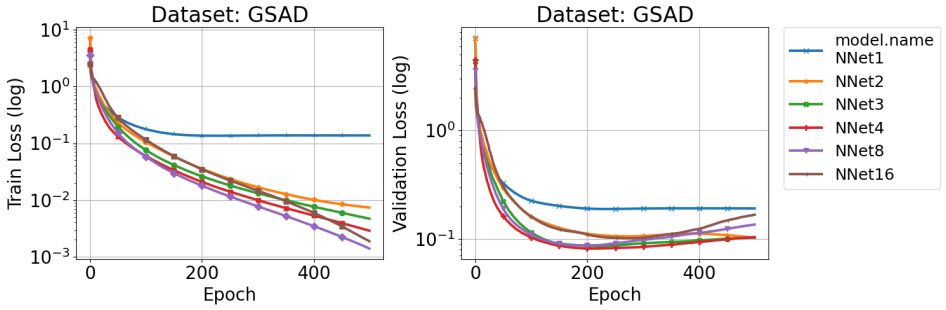
Figure 5.1: Per class population for each dataset.



(a) GD(Armijo+Nesterov) without regularization.

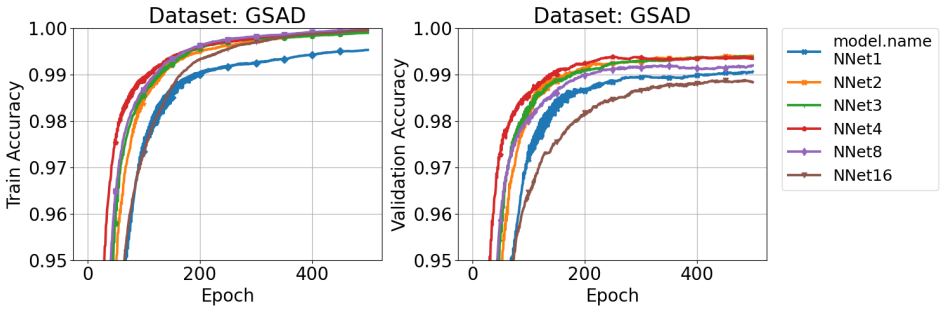


(b) GD(Armijo+Nesterov) with regularization ($\lambda = 10^{-3}$).

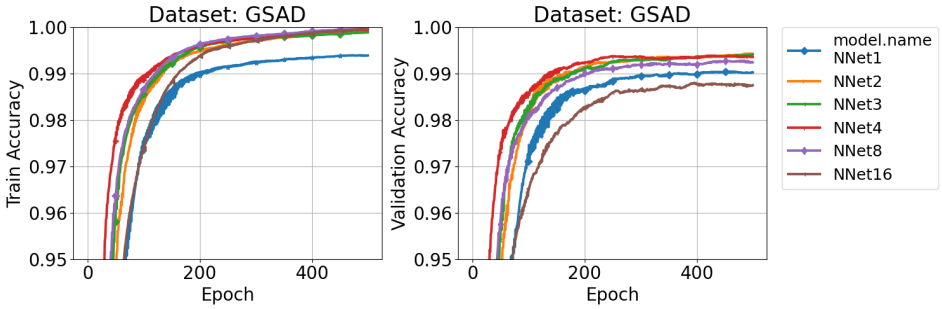


(c) GD(Armijo+Nesterov) with normalized regularization ($\lambda = \frac{0.8}{|\mathcal{E}|}$).

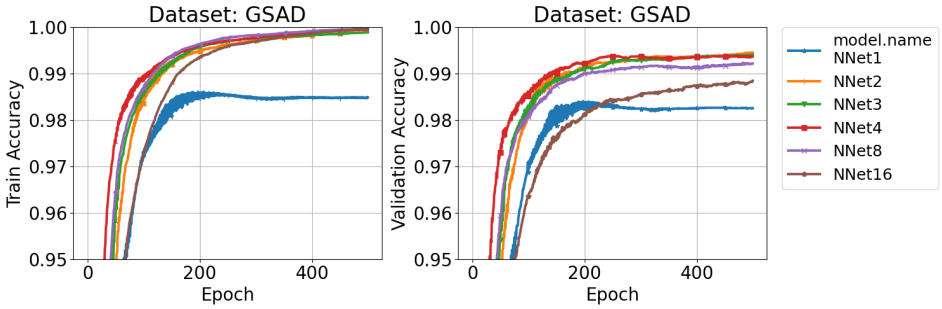
Figure 5.2: Training over the GSAD Dataset using full-batch optimization and evaluating, for each epoch, the "Train Loss" and the "Validation Loss".



(a) GD(Armijo+Nesterov) without regularization.

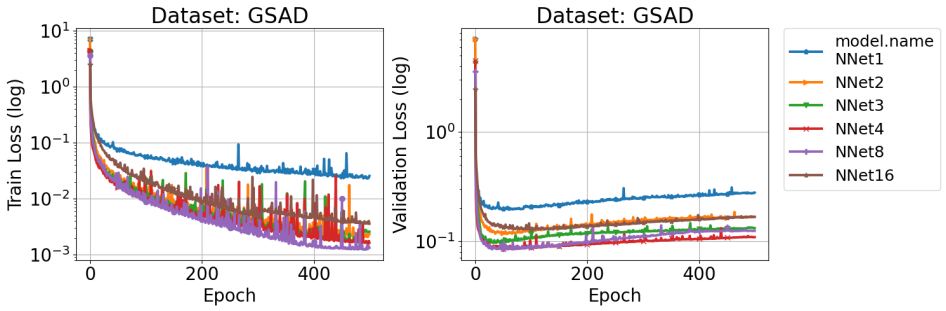


(b) GD(Armijo+Nesterov) with regularization ($\lambda = 10^{-3}$).

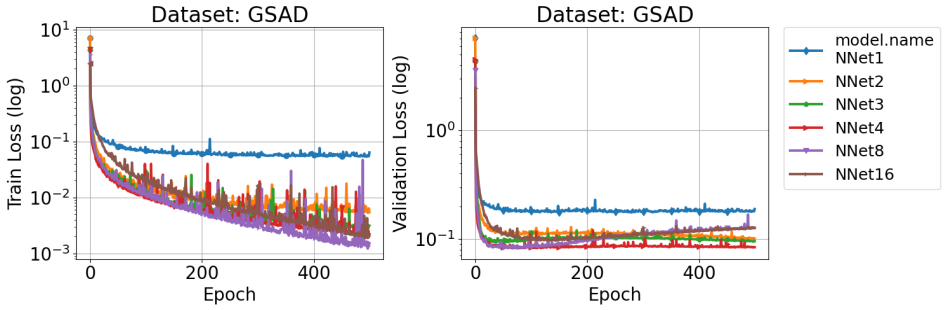


(c) GD(Armijo+Nesterov) with normalized regularization ($\lambda = \frac{0.8}{|\mathcal{E}|}$).

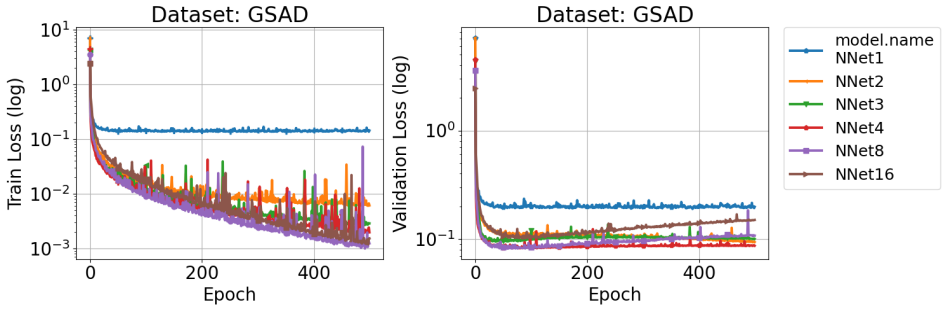
Figure 5.3: Training over the GSAD Dataset using full-batch optimization and evaluating, for each epoch, the "Train Accuracy" and the "Validation Accuracy".



(a) SGD(Armijo) without regularization.

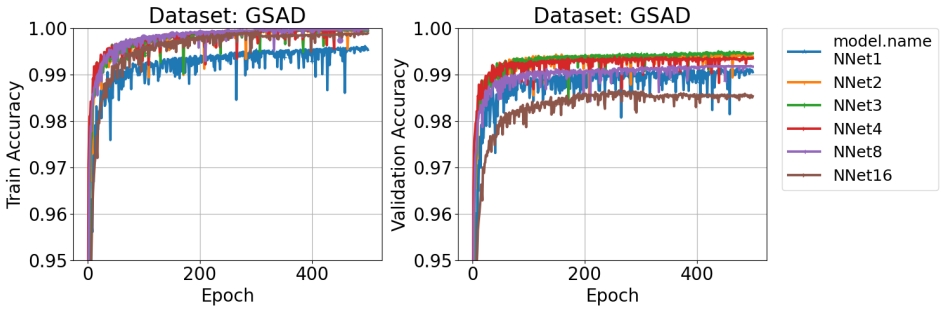


(b) SGD(Armijo) with regularization ($\lambda = 10^{-3}$).

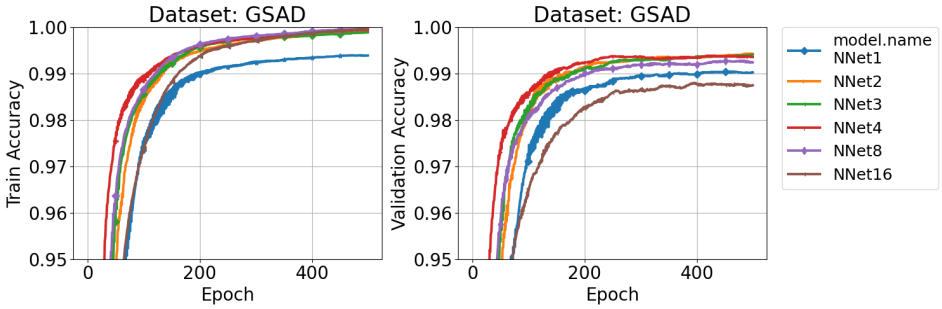


(c) SGD(Armijo) with normalized regularization ($\lambda = \frac{0.8}{|\mathcal{E}|}$).

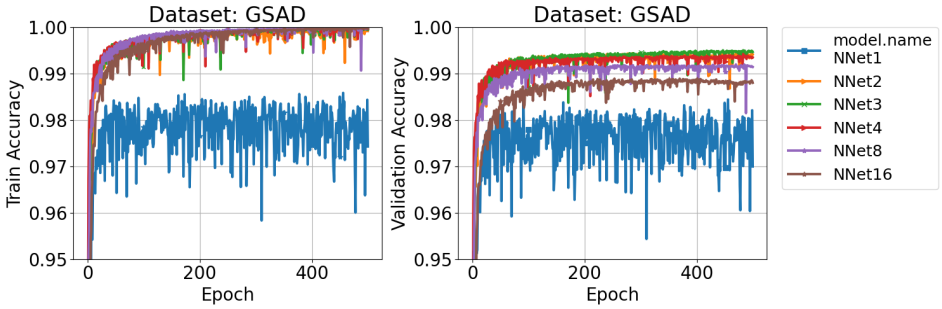
Figure 5.4: Training over the GSAD Dataset using mini-batch optimization and evaluating, for each epoch, the "Train Loss" and the "Validation Loss".



(a) SGD(Armijo) without regularization.

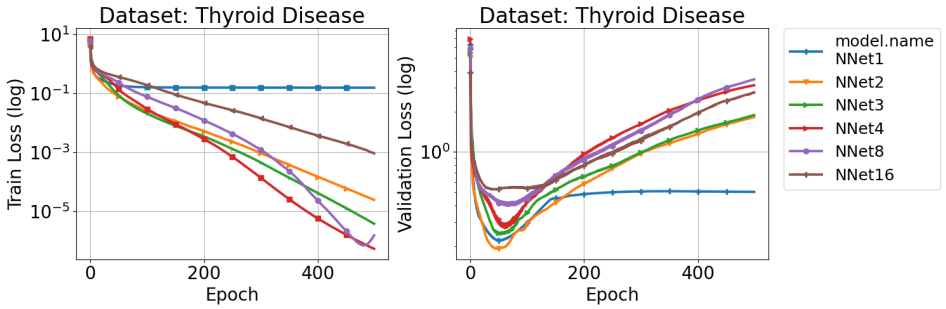


(b) SGD(Armijo) with regularization ($\lambda = 10^{-3}$).

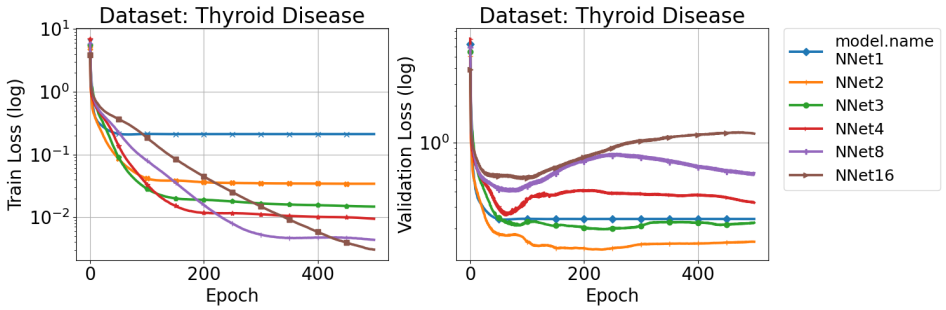


(c) SGD(Armijo) with normalized regularization ($\lambda = \frac{0.8}{|\mathcal{E}|}$).

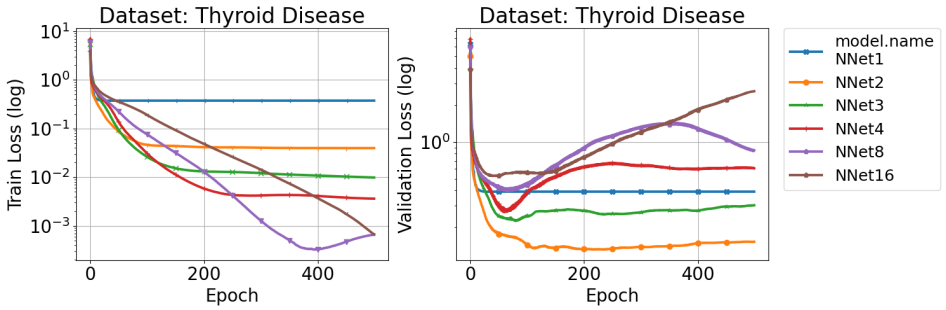
Figure 5.5: Training over the GSAD Dataset using mini-batch optimization and evaluating, for each epoch, the "Train Accuracy" and the "Validation Accuracy".



(a) GD(Armijo+Nesterov) without regularization.

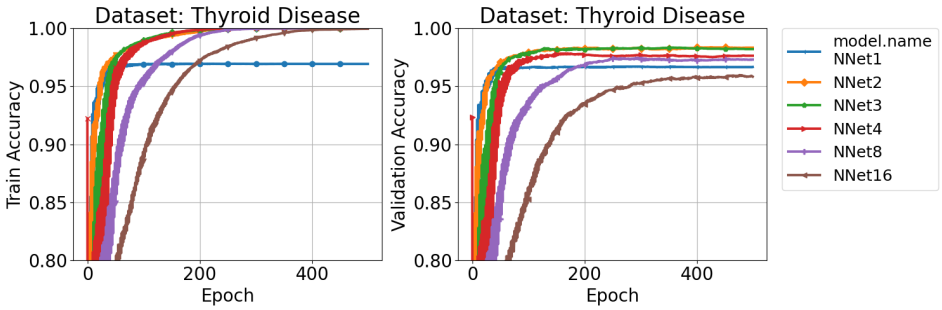


(b) GD(Armijo+Nesterov) with regularization ($\lambda = 10^{-3}$).

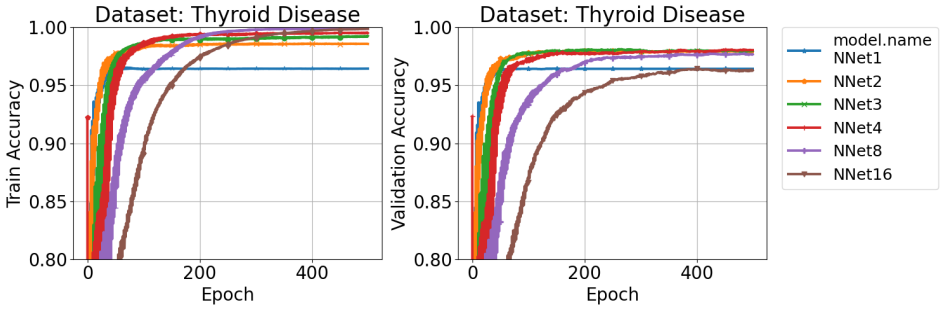


(c) GD(Armijo+Nesterov) with normalized regularization ($\lambda = \frac{0.8}{|\mathcal{E}|}$).

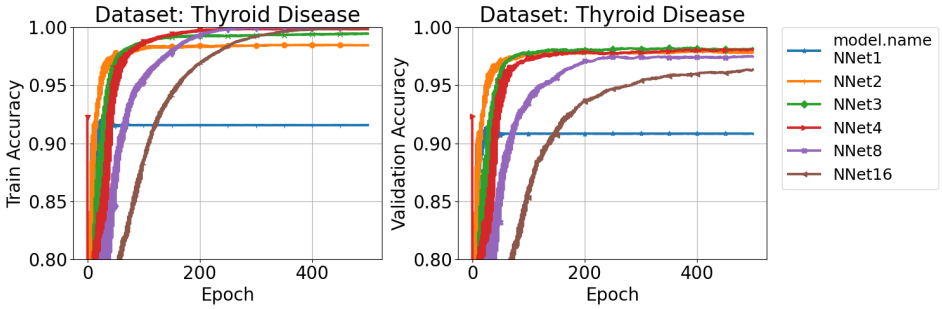
Figure 5.6: Training over the Thyroid Disease Dataset using full-batch optimization and evaluating, for each epoch, the "Train Loss" and the "Validation Loss".



(a) GD(Armijo+Nesterov) without regularization.

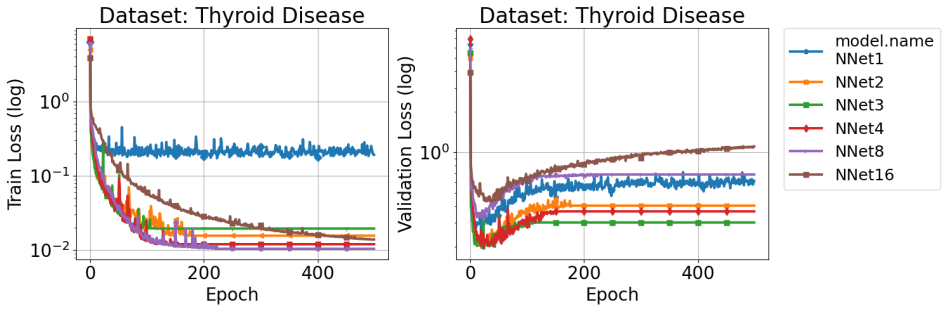


(b) GD(Armijo+Nesterov) with regularization ($\lambda = 10^{-3}$).

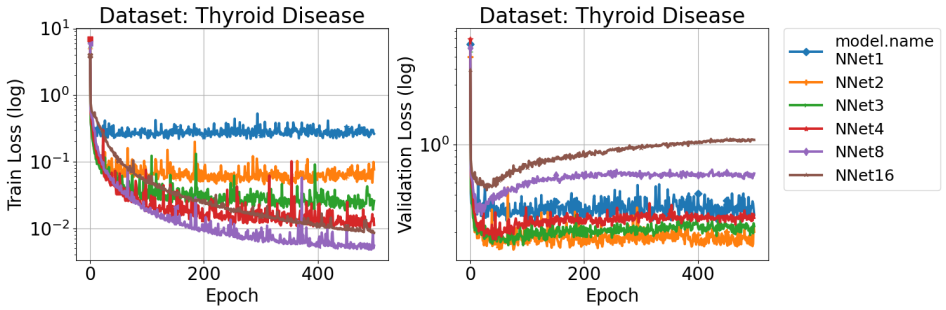


(c) GD(Armijo+Nesterov) with normalized regularization ($\lambda = \frac{0.8}{|\mathcal{E}|}$).

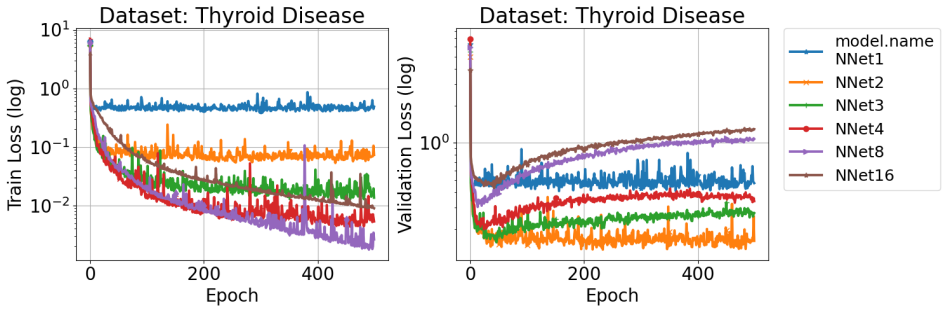
Figure 5.7: Training over the Thyroid Disease Dataset using full-batch optimization and evaluating, for each epoch, the "Train Accuracy" and the "Validation Accuracy".



(a) SGD(Armijo) without regularization.

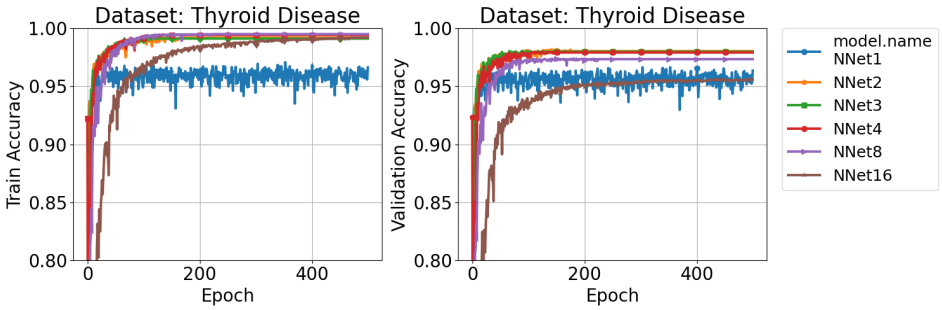


(b) SGD(Armijo) with regularization ($\lambda = 10^{-3}$).

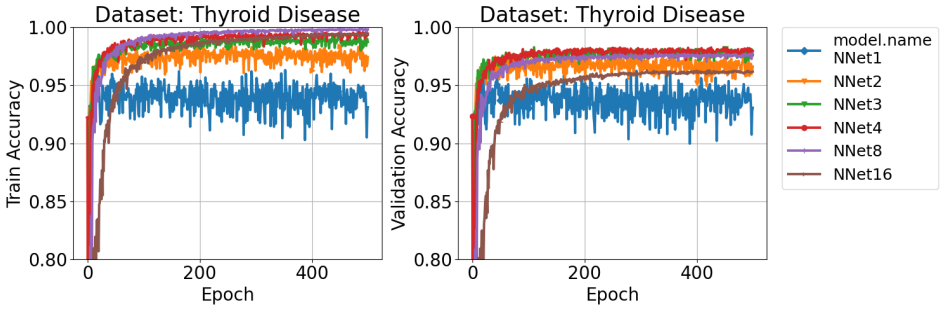


(c) SGD(Armijo) with normalized regularization ($\lambda = \frac{0.8}{|\mathcal{E}|}$).

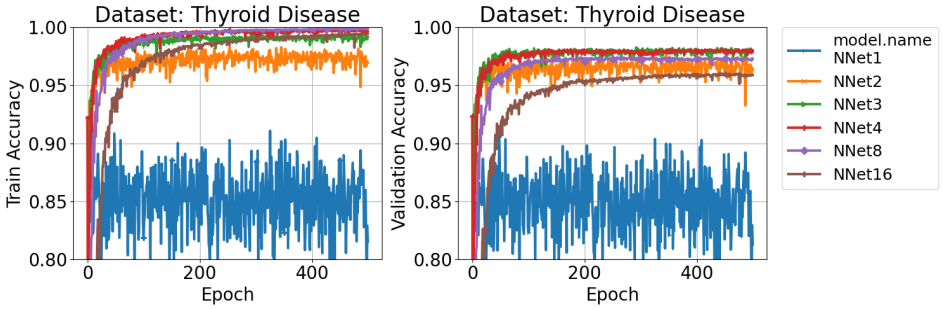
Figure 5.8: Training over the Thyroid Disease Dataset using mini-batch optimization and evaluating, for each epoch, the "Train Loss" and the "Validation Loss".



(a) SGD(Armijo) without regularization.

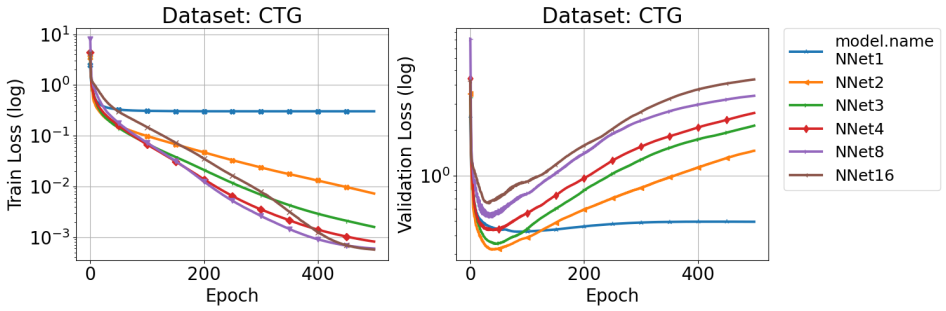


(b) SGD(Armijo) with regularization ($\lambda = 10^{-3}$).

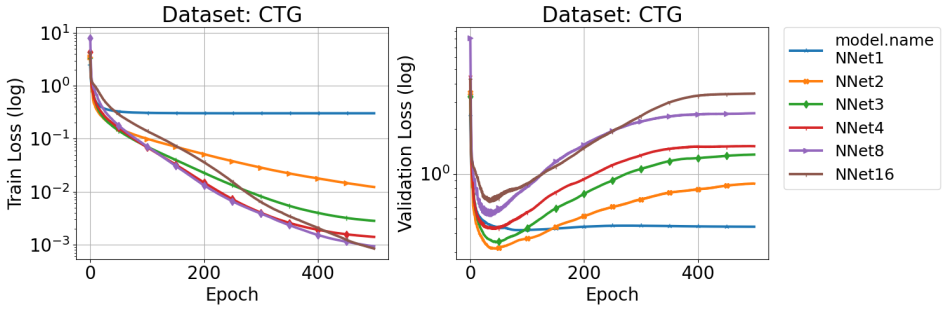


(c) SGD(Armijo) with normalized regularization ($\lambda = \frac{0.8}{|\mathcal{E}|}$).

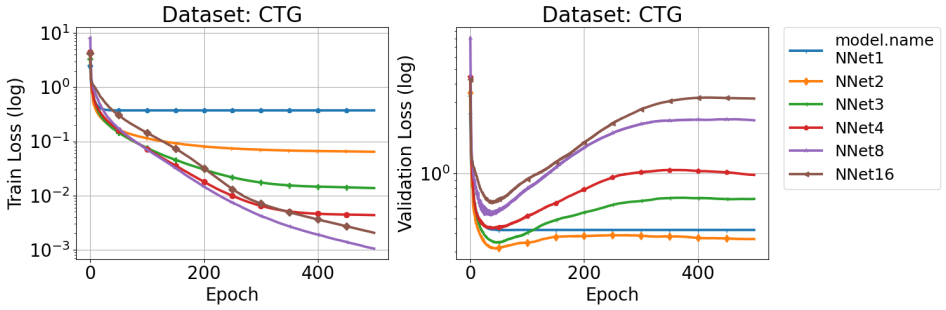
Figure 5.9: Training over the Thyroid Disease Dataset using mini-batch optimization and evaluating, for each epoch, the "Train Accuracy" and the "Validation Accuracy".



(a) GD(Armijo+Nesterov) without regularization.

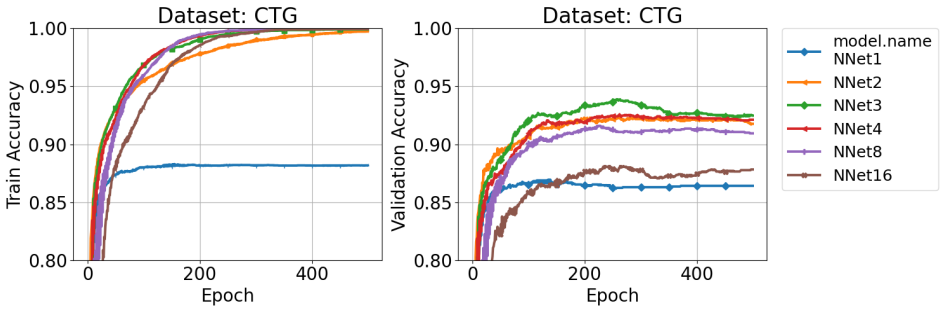


(b) GD(Armijo+Nesterov) with regularization ($\lambda = 10^{-3}$).

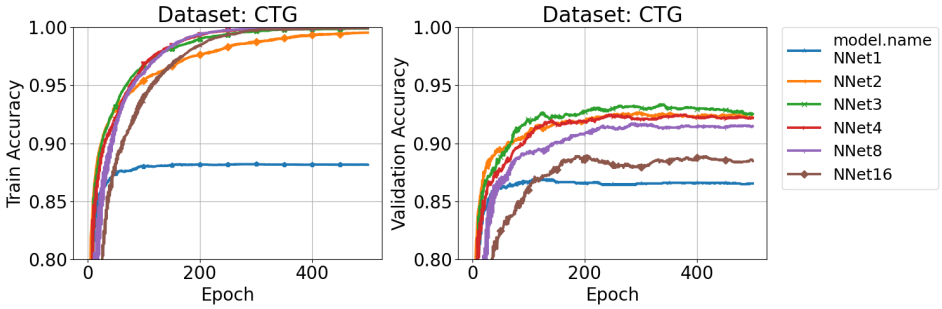


(c) GD(Armijo+Nesterov) with normalized regularization ($\lambda = \frac{0.8}{|\mathcal{E}|}$).

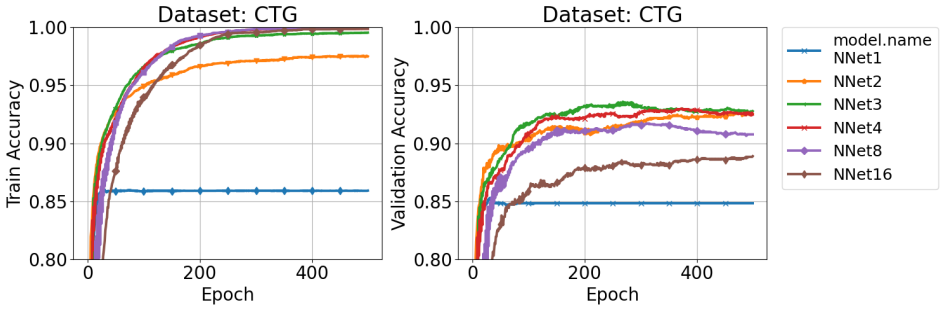
Figure 5.10: Training over the CTG Dataset using full-batch optimization and evaluating, for each epoch, the "Train Loss" and the "Validation Loss".



(a) GD(Armijo+Nesterov) without regularization.

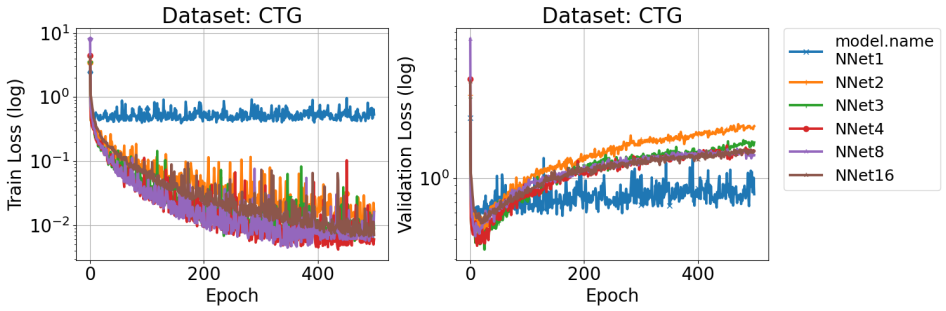


(b) GD(Armijo+Nesterov) with regularization ($\lambda = 10^{-3}$).($\lambda = 10^{-3}$).

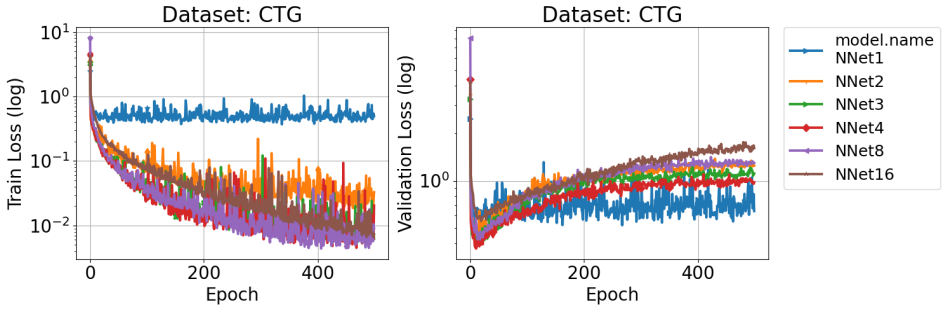


(c) GD(Armijo+Nesterov) with normalized regularization ($\lambda = \frac{0.8}{|\mathcal{E}|}$).

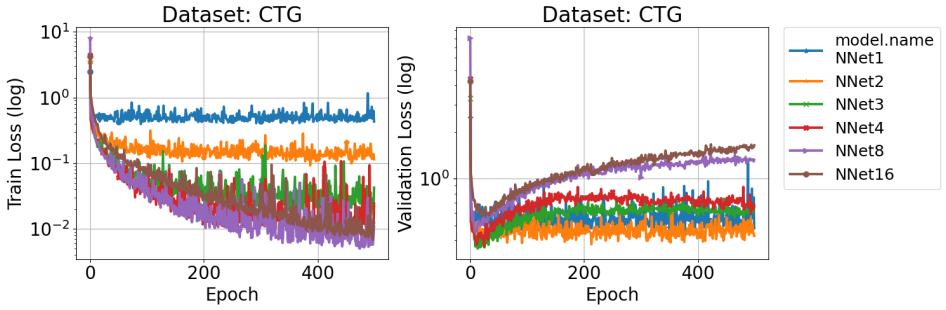
Figure 5.11: Training over the CTG Dataset using full-batch optimization and evaluating, for each epoch, the "Train Accuracy" and the "Validation Accuracy".



(a) SGD(Armijo) without regularization.

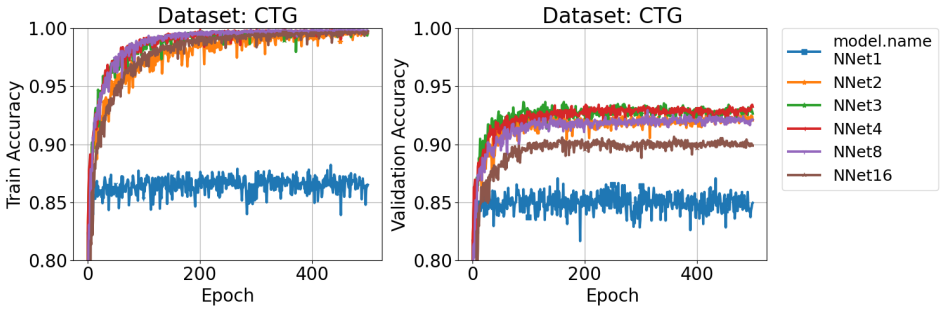


(b) SGD(Armijo) with regularization ($\lambda = 10^{-3}$).

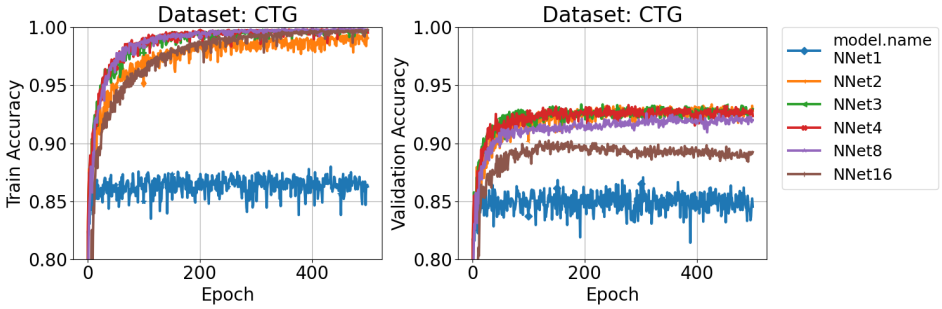


(c) SGD(Armijo) with normalized regularization ($\lambda = \frac{0.8}{|\mathcal{E}|}$).

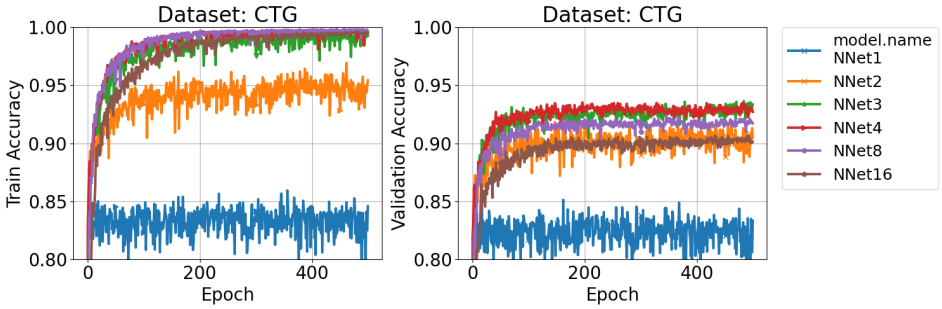
Figure 5.12: Training over the CTG Dataset using mini-batch optimization and evaluating, for each epoch, the "Train Loss" and the "Validation Loss".



(a) SGD(Armijo) without regularization.



(b) SGD(Armijo) with regularization ($\lambda = 10^{-3}$).



(c) SGD(Armijo) with normalized regularization ($\lambda = \frac{0.8}{|\mathcal{E}|}$).

Figure 5.13: Training over the CTG Dataset using mini-batch optimization and evaluating, for each epoch, the "Train Accuracy" and the "Validation Accuracy".

5.3 Observations

We make the following observations:

- Line-search techniques can be used for training deep neural networks. The main advantage is that we can train our models without the need of tuning hyperparameters (i.e., step-size), which reduces the complexity of the training process.

- Model complexity leads to a trade-off between training and generalization, that is, the "Train Loss" and the "Validation Loss", respectively. Typically, more complex models tend to better adapt to the training data, that is, to lead to small "Train loss". However, they usually lead to large "Validation loss" and thus, large generalization gap, indicating that we are possibly overfitting the data (see Figs [5.2a], [5.4a], [5.6a], [5.8a], [5.10a] and [5.12a]). On the other hand, if the model is not sufficiently complex, it leads to relatively large "Train Loss" (and "Validation Loss"), hence, we are possibly underfitting the data (see the performance of "NNet1" architecture, in Figs [5.2a], [5.4a], [5.6a], [5.8a], [5.10a] and [5.12a]).

- Typically, with full-batch optimization, our algorithm converges closer to a global optimal solution (smaller "Train Loss"), however, our model is less resilient to overfitting (the gap between "Train Loss" and "Validation Loss" is big, especially if the model is flexible). On the contrary, with (stochastic) mini-batch optimization, the gap becomes smaller (compare Figs [5.2a], [5.4a] and [5.6a], [5.8a] and [5.10a], [5.12a]).

- Regularization prevents the model from overfitting. Of course, depending on the flexibility of a given model, an appropriate regularization penalty must be selected. Typically, the more flexible the model is, the smaller the regularization penalty (ℓ_2-regularization) must be (see Figs [5.2b], [5.4b], [5.6b], [5.8b], [5.10b] [5.12b] and [5.2c], [5.4c], [5.6c], [5.8c], [5.10c] [5.12c]). Therefore, regularization is another mechanism for reducing the flexibility, by tightening the regularization penalty λ.

Chapter 6

Conclusions

6.1 Conclusion

The presented stochastic line-search algorithms have demonstrated high performance, even compared to conventional algorithms, for training deep neural networks. This highlights that stochastic line-search algorithms can be used in deep learning, offering a "painless" optimization, since the number of forward passes are comparable to the number of forward passes for a conventional training algorithm. Surprisingly, Nesterov acceleration, considering the (weaken) convex variant and full-batch optimization, is able to train a (simple) deep neural network, even if the objective function is highly non-convex.

6.2 Future Work

The future plan is to use stochastic line-search algorithms, and its variants, in modern neural network architectures to test their performance over various tasks. Of course, stochastic line-search algorithms can also be applicable to any parametric model (i.e., SVMs).

Bibliography

[1] S. Shalev-Shcwarz and S. Ben-David, *Understanding Machine Learning: From theory to Algrithms.* Cambridge University Press, 2014.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms.* MIT Press, 2009.

[3] A. Lindholm, N. Wahlström, F. Lindsten, and T. B. Schön, *Machine Learning - A First Course for Engineers and Scientists.* Cambridge University Press, 2022. [Online]. Available: https://smlbook.org

[4] K. P. Murphy, *Probabilistic Machine Learning: An introduction.* MIT Press, 2022. [Online]. Available: probml.ai

[5] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," 2016. [Online]. Available: https://arxiv.org/abs/1603.07285

[6] C. M. Bishop, *Pattern Recognition and Machine Learning.* Springer New York, NY, 2006.

[7] K. P. Murphy, *Probabilistic Machine Learning: Advanced Topics.* MIT Press, 2023. [Online]. Available: http://probml.github.io/book2

[8] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Representations by Back-propagating Errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986. [Online]. Available: http://www.nature.com/articles/323533a0

[9] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning (Adaptive Computation and Machine Learning series).* MIT Press, 2016.

[10] C. M. Bishop, *Neural Network for Pattern Recognition.* Oxford University Press, 1995.

[11] A. Beck, *Introduction to Nonlinear Optimization: Theory, Algorithms, and Applications with MATLAB.* USA: Society for Industrial and Applied Mathematics, 2014.

[12] S. Boyd and L. Vandenberghe, *Convex optimization.* Cambridge university press, 2004.

[13] S. J. Wright and B. Recht, *Optimization for Data Analysis*. Cambridge University Press, 2022.

[14] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," 2016. [Online]. Available: https://arxiv.org/abs/1606.04838

[15] H. Robbins and S. Monro, "A Stochastic Approximation Method," *The Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400 – 407, 1951. [Online]. Available: https://doi.org/10.1214/aoms/1177729586

[16] S. Vaswani, A. Mishkin, I. Laradji, M. Schmidt, G. Gidel, and S. Lacoste-Julien, "Painless stochastic gradient: Interpolation, line-search, and convergence rates," 2019. [Online]. Available: https://arxiv.org/abs/1905.09997

[17] B. Polyak, "Some methods of speeding up the convergence of iteration methods," *USSR Computational Mathematics and Mathematical Physics*, vol. 4, no. 5, pp. 1–17, 1964. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0041555364901375

[18] Y. Nesterov, "Introductory lectures on convex optimization - a basic course," in *Applied Optimization*, 2014.

[19] S. J. Wright, "Optimization algorithms for data analysis," 2017. [Online]. Available: https://optimization-online.org/?p=14296

[20] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, Y. W. Teh and M. Titterington, Eds., vol. 9. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. [Online]. Available: https://proceedings.mlr.press/v9/glorot10a.html

[21] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," 2015. [Online]. Available: https://arxiv.org/abs/1502.01852

[22] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[23] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," 2016. [Online]. Available: https://arxiv.org/abs/1603.05027

[24] ——, "Deep residual learning for image recognition," 2015. [Online]. Available: https://arxiv.org/abs/1512.03385

[25] A. Veit, M. Wilber, and S. Belongie, "Residual networks behave like ensembles of relatively shallow networks," 2016. [Online]. Available: https://arxiv.org/abs/1605.06431

[26] M. Tomz, G. King, and L. Zeng, "Relogit: Rare events logistic regression," *Journal of Statistical Software*, vol. 8, no. 2, p. 1–27, 2003. [Online]. Available: https://www.jstatsoft.org/index.php/jss/article/view/v008i02

[27] A. Vergara, S. Vembu, T. Ayhan, M. A. Ryan, M. L. Homer, and R. Huerta, "Chemical gas sensor drift compensation using classifier ensembles. sensors and actuators b: Chemical." [Online]. Available: https://archive.ics.uci.edu/ml/datasets/gas+sensor+array+drift+dataset

[28] R. Quinlan, "Thyroid Disease," UCI Machine Learning Repository, 1987, DOI: https://doi.org/10.24432/C5D010.

[29] D. Campos and J. Bernardes, "Cardiotocography," UCI Machine Learning Repository, 2010, DOI: https://doi.org/10.24432/C51S4N.