School of Information Sciences and Technology

Department of Statistics

Deep Learning: Theoretical Approaches and Applications in Quantitative Finance

CHARALAMPOS SALIS

In Collaboration with: EUROBANK

Powered by: TensorFlow 2.0

Bachelor Thesis

Submitted to the Department of Statistics

of the Athens University of Economics and Business

Supervisors: Athanasios N. Yannacopoulos, Georgios I. Papayiannis, Stavros Vakeroudis

Athens,

September 2022

Dedicated to the people who have supported me throughout my education, especially Professors Athanasios N. Yannacopoulos and Georgios I. Papayiannis for their unreserved assistance during the writing of this thesis.

Contents

Abstract

Neural Networks enjoy an impressive success as modelling tools in a variety of fields. Their ability to capture non-linear complex relationships and their sandbox nature renders them both efficient and easy to use. However, despite their extensive use, neural networks are often presented in a applied computational manner, deprived of rigorous mathematical background. The objective of this thesis is to present the topic of neural networks from both a computational and a mathematical perspective to disclose that these two approaches complement and do not substitute each other.

After the presentation of fundamental neural network key elements, we examine the numerical optimization theory and algorithms that are employed during model fitting, explore the mathematical background of neural networks as function approximators and, afterwards, the most important neural network architectures. Finally, an application of neural network modelling in portfolio management is given to complement the theory with empirical results.

Περίληψη

Τα νευρωνικά δίκτυα σημείωσαν μία εντυπωσιακή επιτυχία ως εργαλεία μοντελοποίησης σε πληθώρα επιστημονικών πεδίων. Η δυνατότητά τους να συλλαμβάνουν σύνθετες, μη γραμμικές σχέσεις και η ελευθερία που προσφέρουν για την κατασκευή τους τα καθιστά τόσο αποτελεσματικά, όσο και εύκολα στη χρήση. Ωστόσο, παρ΄όλη τη διαδεδομένη χρήση του, τα νευρωνικά δίκτυα, συνήθως, παρουσιάζονται με έναν εφασμοσμένο υπολογιστικό τρόπο, στερώντας τους ένα ενδελεχές μαθηματικό υπόβαθρο. Ο στόχος της παρούσας διπλωματικής είναι να παρουσιάσει τα νευρωνικά δίκτυα τόσο από μία υπολογιστκή, όσο και από μία μαθηματική προσέγγιση, ώστε να δειχθεί ότι οι δύο προσεγγίσεις αλληλο-συμπληρώνουν και δεν υποκαθιστούν η μία την άλλη.

Αφού πρώτα παρουσιαστούν κάποια θεμελειώδη στοιχεία των νευρωνικών δικτύων, εξετάζονται η θεωρία βελτιστοποίησης και οι υπολογιστικοί αλγόριθμοι που χρησιμοποιούνται κατά την προσαρμογή των μοντέλων, το μαθηματικό υπόβαθρο των νευρωνικών δικτύων ως μέθοδοι προσέγγισης συναρτήσεων και οι σημαντικότερες αρχιτεκτονικές. Τέλος, προσφέρεται μία εφαρμογή των νευρωνικών δικτύων στη διαχείριση χαρτοφυλακίου, ώστε η θεωρία να συμπληρωθεί με εμπειρικά αποτελέσματα.

Chapter 1

Introduction: An Overview of Neural Networks

Emotions are enmeshed in the neural networks of reason

Antonio Damasio

1.1 Introduction

We begin the exploration of neural networks by examining their basic building block; the neuron. After formulating the mathematical context of the neuron and its components, we delve into feedforward networks, the most foundational deep learning architecture.

In this chapter we are interested in the intuition and the framework of feedforward networks from both a theoretical and applied background, while also exploring their limitations. Although we visit the topic of training from a general perspective, we treat the actual training (optimization) algorithms as a black box. In that sense, Chapter 2 is a natural continuation of this Chapter, diving deep inside the topic of optimization theory. It is vital to note that certain elements of this Chapter, such as activation and loss functions or the back-propagation algorithm are also important for network architectures other than the feedforward, which are examined in Chapter 4.

1.2 Perceptrons and Abstract Neurons

Initially, we define one of the most general building blocks of neural networks; the abstract neuron.

Definition 1.2.1. *An abstract neuron is the tuple (x, w, ϕ, y), where: x are the neuron's inputs, w are the weights and ϕ is the activation function, from which the output $y = \phi(w^T x)$ is derived.*

One other component of an abstract neuron is the bias b and its corresponding weight w_b. However, in order to avoid over-complicating our formulas, we consider bias as another element of the input vector (the zeroth element x_0 and its weight w_0).

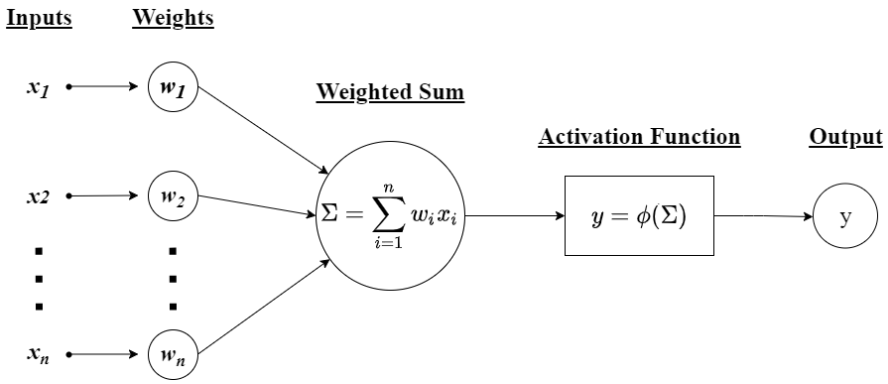


Figure 1.1: Format of a typical Neuron

Before we delve into the learning process of a neuron, we have to examine its fundamental elements that are used during training; the activation and loss functions.

1.2.1 Activation Functions

The choice of activation function defines the nature of both the individual neuron and the neural network altogether. We can categorize them into groups, each one having advantages and disadvantages from a theoretical and a computational perspective.

Linear Functions

Linear activation functions have the form:

$$\phi(x) = a\ x, \text{ where } a > 0$$

If $a = 1$, we have a special case of linear function; the identity activation function.

Although a network equipped with mostly linear activation functions can be easily trained, it is mostly incapable of learning complex mapping functions. Another problem with linear functions is that they restrict the use of back-propagation. Hence, they are mainly deployed to the output layer of a feedforward network.

Step Functions

The family of step functions model a neural stimulus as a jump. Two of the most notable members are:

· Heaviside function

$$H(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$$

· Signum function

$$S(x) = \begin{cases} -1 & x < 0 \\ 1 & x \geq 0 \end{cases}$$

Both functions are not differentiable at $x = 0$.

The limitations of step functions are that they do not support back-propagation and cannot be used in multiclass classification problems, because of non-smoothness.

Hockey-Stick Functions

This category encompasses some of the most commonly used activation functions. From this point onwards, we move away from linearity in order to gain the ability to learn non-linear target functions.

· Rectified Linear Unit (ReLU)

$$ReLU(x) = xH(x) = max\{x, 0\}$$

One important remark is that ReLU is differentiable with the exception of 0, and: $ReLU'(x) = H(x)$ and, thus, allows the use of back-propagation. However, the fact that its derivative is not a smooth function could be problematic in applied problems. Furthermore, as it is does not activate all neurons simultaneously, it is computationally much more efficient than other widespread activation functions. Finally, its non-saturating property accelerates the convergence of gradient descent algorithms to an optimum.

On the other hand, the main drawback of this function is the dying ReLU problem, meaning that during back-propagation the weights of some neurons with consecutive negative values are not updated, which renders them "dead" neurons that are never activated.

· Parametric Rectified Linear Unit (PReLU)

$$PReLU(x) = \begin{cases} ax & x < 0 \\ x & x \geq 0 \end{cases} \quad \text{, where } a > 0$$

Parametric ReLU is a modification of vanilla ReLU that fixes the dying neurons problem. However, the use of constant a could lead to inconsistent predictions for negative values and slower convergence rates of optimization algorithms.

· Exponential Rectified Linear Units (ELU)

$$ELU(x) = \begin{cases} x & x > 0 \\ a(e^x - 1) & x \leq 0 \end{cases}$$

ELU is an effort to smooth ReLU, which solves the dead ReLU problem even more efficiently than the PReLU. Its main flaws are that it is computationally more demanding due to the introduction of the exponential term and in certain architectures could cause the exploding gradient problem.

· Sigmoid Linear Units (SLU)

$$\phi(x) = x\sigma(x) = \tfrac{x}{1+e^{-x}}$$

The SLU is the product of a sigmoid and a linear function. Its derivative is given by the formula:

$$\phi'(x) = \sigma(x)[1 + x\sigma(-x)]$$

The main difference between SLU and the rest of the Hockey-Stick type functions is that the SLU is not monotonic and has a minimum. This biologically-inspired property can be advantageous in certain problems.

· Softplus

$$sp(x) = ln(1 + e^x)$$

Softplus is a smooth approximation of the ReLU activation function. It is differentiable with the sigmoid function as its derivative:

$$sp'(x) = \frac{1}{1+e^{-x}}$$

In practise, the softplus function provides a convenient alternative to the ReLU due to its ability to provide the back-propagation algorithm with better results.

Sigmoidal Functions

The members of the sigmoid family are smooth and can sufficiently approximate any step function. Thus, the use of the back-propagation algorithm is easier and more solid with this type of functions, especially from an applied perspective, which renders them widespread.

Before we elaborate on the members of the sigmoid family, we first have to examine the properties of sigmoidal functions.

Definition 1.2.2. *A function $\sigma : \mathbb{R} \to [0, 1]$ is sigmoidal, if:*

$$\lim_{x\to-\infty} \sigma(x) = 0 \text{ and: } \lim_{x\to\infty} \sigma(x) = 1$$

An important concept is that of the discriminatory functions. The intuition behind is based on the idea that a measure μ quantifies beliefs used to evaluate information. Hence, the concept is that using all possible input data x, if under the belief μ the evaluation of the neuron $\sigma(w^T x + \theta)$ vanishes, then μ is a void belief.

Definition 1.2.3. *Let $\mu \in M(I_n)$. A function σ is called discriminatory for measure μ if:*

$$\int_{I_n} \sigma(w^T x + \theta) d\mu(x) = 0, \ \forall w \in \mathbb{R}^n, \theta \in \mathbb{R}$$

then:

$$\mu = 0$$

Now, we can consider the connection between sigmoidal and discriminatory functions.

Proposition 1.2.1. *Any continuous sigmoidal function is discriminatory for all measures $\mu \in M(I_n)$.*

Having the mathematical background, we can now examine the sigmoidal family.

· Logistic Function

$$\sigma_c(x) = \frac{1}{1+e^{-cx}}$$

The logistic function is a smooth approximation of the Heaviside step function, where the constant c calibrates the sensitivity to neural stimuli. Its derivative is given by the formula:

$$\sigma_c' = c\sigma_c(1 - \sigma_c)$$

The logistic activation function is commonly used, because it is differentiable and can provide the back-propagation algorithm with smooth gradients. It can also conduct probabilistic predictions, which is useful in binary classification problems. On the other hand, the use of logistic function could cause the vanishing gradient problem.

· Hyperbolic Tangent

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

Similarly with the logistic function, the hyperbolic tangent can be considered as a smooth approximation of the signum function. Its derivative satisfies the following differential equation, which can be used by the back-propagation algorithm:

$$t'(x) = 1 - t^2(x)$$

The main advantage of the hyperbolic tangent is that its output is centered at zero, giving it an edge over the logistic function, especially in classification problems, and making the learning process more efficient. However, the vanishing gradient problem remains.

· Softsign Function

$$so(x) = \frac{x}{1+|x|}$$

The softsign function is similar with the hyperbolic tangent, although its tails are quadratic polynomials and not exponential and, therefore, saturating slower. Softsign is also differentiable, with derivative:

$$so'(x) = \frac{1}{(1+|x|)^2}$$

Its slower saturation coupled with its ability to support the back-propagation algorithm could lead to better performance than that of the hyperbolic tangent in certain problem settings.

Classification Functions

In a multiclass classification problem, where each observation could be classified in one of n categories, with $n \geq 2$, we could model the category i where an observation belongs as a coordinate vector of $\mathbb{R}^n$, i.e.: $e_i = (0, .., 0, 1, 0, ..., 0)$, with 1 being the i-th element. This problem setting requires a specific activation function that can make probabilistic predictions about the form of the corresponding coordinate classification vector for each datum.

· Softmax Function

$softmax(x) = z$, where the j-th coordinate is given as: $z_j = \frac{e^{x_j}}{||e^x||}$

Typically, we consider the L^1-norm: $||e^x|| = \sum_{i=1}^{n} e^{x_i}$. The softmax function returns a vector of $\mathbb{R}^n$, where each element is the probability that the observation belongs to that category and can be considered as the generalization of the logistic function in higher dimensional classification problems. For this reason, softmax is usually selected as the activation function of the network's output layer.

Bumped-type Functions

A more experimental type of activation functions are the bumped-like functions. Due to their smoothness they can allow the use of the back-propagation algorithm and recent results have shown that networks equipped with those functions require fewer hidden layers and, thus, are trained much faster than conventional neural networks. Most notable members are:

· Gaussian Function

$$f(x) = e^{-x^2}$$

· Double Exponential Function

$$f(x) = e^{-\lambda|x|}, \text{ with: } \lambda > 0$$

1.2.2 Cost Functions

We could treat a neural network as a function, modifying an input using a certain rule to produce an output. During the learning process, the network

calibrates its weight values to approximate a target function sufficiently. In order to obtain a mathematical formulation of this objective, we need to have a measure of divergence between the network's output and the target value.

A loss function is a function that is capable of quantifying the aforementioned divergence. Similarly with activation functions, loss functions can be categorized into families, each one used in specific problem settings. The concept of training is that by changing its weight values, the network minimizes its loss function and, consequently, solves an optimization problem defined by that function. It is important to keep in mind that not only the setting of the problem, but also the network architecture itself determine the choice of loss function.

Supremum Loss Function

Suppose that for input $x \in [0,1]$, the task of network's output mapping f_w is to approximate a target function $g : [0,1] \to \mathbb{R}$. The supremum loss function has the following form:

$$L(w) = \sup_{x\in[0,1]} |f_w(x) - g(x)|$$

In an applied setting, where only n values of the target function are known, the supremum loss function becomes:

$$L(w) = \max_{i\in\{1,\dots,n\}} |f_w(x_i) - g(x_i)|$$

Although supremum loss is of great importance from a theoretical perspective, in the sense that it uncovers the general intuition behind loss functions in a mathematical context, it has limited applied value.

L^2-Loss Function

Assuming again that for input $x \in [0,1]$, our goal using the neural network is to approximate a target function $g : [0,1] \to \mathbb{R}$ that is also square-integrable, then the L^2-loss function is the distance between output and target value using the L^2-norm:

$$L(w) = \int_0^1 [f_w(x) - \phi(x)]^2 dx$$

When only n target points are known, the loss function takes the form:

$$L(w) = ||f_w(x) - \phi(x)||_2^2 = \sum_{i=1}^n [f_w(x_i) - \phi(x_i)]^2$$

By defining the L^2-loss we made an important step to solve practical problems. It is easy to see that for fixed: $x, \phi(x) \in \mathbb{R}^n$, this loss function is smooth and, thus, can be minimized by optimization algorithms discussed in Chapter 2.

Mean Square Error

We move away from fixed input and target points and start to introduce random variables to our loss functions in order to be able to formulate real-life problems. Let X a random variable that is the input of our network. Then, using the output random variable $Y = f_w(X)$, we approximate the target random variable Z. A measure of divergence between Y and Z is their expected squared difference:

$$L(w) = \mathbb{E}[(Y - Z)^2]$$

Before we further examine this loss function, we first have to show that minimizing the expected squared difference is equivalent with minimizing the cost. The set of all square-integrable random variables defined on a probability space forms a Hilbert space with dot product: $\langle X, Y \rangle = \mathbb{E}(XY)$. This yields the norm: $||X||^2 = \mathbb{E}(X^2)$ and, consequently, the distance: $d(X, Y) = ||X - Y||$. Thus, the expected squared difference is the squared distance and, hence, minimizing it minimizes the cost itself.

Now, we can explore the mean square error in an applied setting, where we have n realizations of the random variables X and Z, i.e. the sequence of tuples: $\{(x_i, z_i)\}_{i=1}^n$ is known. Then, the loss function is modified as follows:

$$L(w) = \frac{1}{n} \sum_{i=1}^{n} [f_w(x_i) - z_i]^2$$

The mean squared error loss function is usually deployed in regression problems due to its nature to depend on only the magnitude of the error and not its direction.

Finally, we have to examine the case where X and Z are independent. Using conditional expectations, one can show that the best output would be: $Y = \mathbb{E}(Z)$, which renders the neural network impractical.

Cross-Entropy

The idea behind most methods from this point onwards is that we measure the divergence between probability densities of random variables and not the proximity of the random variables themselves.

We consider two densities on $\mathbb{R}$, p and q. The negative log-likelihood: $-l_q(x) = -\ln q(x)$ quantifies the information given by q. We define the cross-entropy of p with respect to q as the information given by q from the perspective of p:

$$S(p, q) = \mathbb{E}_p[-l_q] = -\int_{\mathbb{R}} p(x) \ln q(x) dx$$

Cross-Entropy is usually used in classification problems, because it can quantify the cost increase when the predicted category diverges from the actual one.

Another useful notion is that of the Shannon entropy: $H(p) = -\int_{\mathbb{R}} p(x) \ln p(x) dx$. The relationship between cross-entropy and Shannon entropy is the following:

Proposition 1.2.2. *It holds: $S(p, q) \geq H(p)$, where: $S(p, q)$ is the cross-entropy of p with respect to q and $H(p)$ is the Shannon entropy of p.*

Kullback-Leibler Divergence

We define the Kullback-Leibler divergence as the difference between the cross-entropy and the Shannon entropy: $D_{KL}(p||q) = S(p, q) - H(p)$. From Proposition 2.2.1, we are guaranteed that: $D_{KL}(p||q) \geq 0$.

Both cross-entropy and Kullback-Leibler divergence are popular loss functions, especially in classification problems. The reason behind that is that they are related with the maximum likelihood method in the sense that the weights vector value w^* that minimizes the cross-entropy or the Kullback-Leibler divergence (since Shannon entropy does not depend on w, the minimization problems are the same) is the maximum likelihood estimator of w.

Jensen-Shannon Divergence

A loss function that is used for training GANs is the Jensen-Shannon divergence, given by the formula:

$$D_{LS}(p||q) = \frac{1}{2}(D_{KL}(p||m) + D_{KL}(q||m)), \text{ where: } m = \frac{p+q}{2}$$

We can examine further the properties of this loss function:

Proposition 1.2.3. *The Jensen-Shannon divergence has the following properties:*

(i) It is non-negative.

(ii) It is symmetric.

(iii) $D_{LS}(p||q) = 0 \Leftrightarrow p = q$

Renyi Entropy

The Renyi entropy is a generalization of the Shannon entropy.

$$H_\alpha(p) = \frac{1}{\alpha} \ln \int p(x)^\alpha dx, \text{ where: } \alpha > 0 \text{ and } \alpha \neq 1$$

A special case of Renyi entropy is the quadratic Renyi entropy:
$H_2(p) = \frac{1}{2} \ln \int p(x)^2 dx$, due to the ability to estimate its value using only a subset of the initial data set.

Estimating the Cost Function

In a typical machine learning problem we have a data set consisting of tuples: $\{(x_i, z_i)\}_{i=1}^n$, where z_i is the target variable and x_i is the value of the predictor variable (or the vector of predictor variables in the multivariate case). Although we have at our disposal the entire data set, sometimes we deploy stochastic optimization algorithm that use only a subset of the data set (called mini-batch) to train the model in order to be computationally more efficient. Thus, we need to have a way to estimate the cost function of the entire data set using only a mini-batch.

· Mean Squared Error: Using a subset of the entire data set: $\{(x_j, z_j)\}_{j=i_1}^{i_k}$, where: $i_1, ..., i_l \in \{1, ..., n\}$ and: $i_1 < i_2 < ... < i_k$, we can estimate the mean squared error as follows:

$$\mathbb{E}[(Z - Y)^2] \approx \frac{1}{k} \sum_{j=i_1}^{i_k} [z_j - f(x_j; w)]^2$$

· Quadratic Renyi Entropy: Because we need to estimate a density function using a sample of our data set, we replace the density p in quadratic Renyi entropy with a sampled based density using a window W_σ:

$$\hat{p}(x) = \frac{1}{k} \sum_{j=i_1}^{i_k} W_\sigma(x, x_j)$$

Using the quadratic potential energy: $U(p) = \int p(x)^2 dx$, the formula of quadratic Renyi entropy is modified as follows:

$$H_2(p) = -\ln U(\hat{p})$$

In this general setting we cannot proceed any further. In applied problems, the next step would be to make an assumption about the window W_σ and then estimate the quadratic potential energy. This would yield an estimation for the value of quadratic Renyi entropy.

1.2.3 Neuron Types

Perceptron Model

The perceptron is the most simple neural network, consisting of only one neuron. The original perceptron was not intended to be an algorithm, but a machine with custom built hardware to perform the training with heuristic updates. It maps a real-valued input $x \in \mathbb{R}$ into a binary valued output $y \in \{0, 1\}$ and can be considered as a simple binary classifier.

The activation function of a perceptron is the Heaviside function, used to map the weighted sum of neuron's inputs into the binary set $\{0, 1\}$. However,

the perceptron does not have a formal loss function, because initially the goal was to minimize misclassifications using in a heuristic and hardware-based way. The whole process was similar with the stochastic gradient descent method presented in Chapter 3, although it was far more crude.

Other than its significance as a binary classifier, the perceptron is also important in logical computation, being able to implement logical gates AND and OR. However, a single perceptron is unable to learn more complex computational calculations, such as XOR. While this caused deep learning research to stagnate at the time, it was the stimulus to discover more complex network architectures.

Sigmoid Neuron

A sigmoid neuron expands on the concept of the perceptron model, by mapping the input $x \in \mathbb{R}$ into the whole interval $(0, 1)$ and not only into two discrete values. Its activation functions are members of the sigmoid family and for each member, we can obtain a different type of sigmoid neuron. Nonetheless, from a mathematical viewpoint their only difference is the saturation's rate of change.

A special case of sigmoid neuron is the one equipped with the logistic activation function. In a typical problem setting, where we have input vector x, the output of the logistic neuron is given as: $y = \sigma(w^T x)$, where: $\sigma(x) = \sigma_1(x)$. Because the logistic function is a smooth approximation of the Heaviside step function, we can approximate a perceptron using a logistic neuron, for $c \to \infty$.

Other than being part of a neural network, a sigmoid neuron can also perform classification tasks, by learning using logistic regression.

Linear Neuron

Another special neuron type is the linear neuron. It is equipped with the linear activation function: $\phi(x) = x$ and the mean squared error loss function. Its applications include pattern recognition, data filtering and linear function approximation.

In an applied problem we have observations from random variables $X_1, ..., X_n$ and corresponding weights $w_1, ..., w_n$. By using the convention that:
$X_0 = -1$, the output of the linear neuron is: $\phi(Y) = Y = w^T X$, with:
$X = [X_0, X_1, ..., X_n]^T$.

During the learning process, we try to find the optimal value of weights that minimizes the loss function: $w^* = argmin_w \mathbb{E}[(Z - Y)^2]$. Typically in deep learning, we use computational optimization algorithms to perform this task, but in this case we can find the exact solution of the problem. We have:

$\mathbb{E}[(Z - Y)^2] = \mathbb{E}[Z^2] - 2\mathbb{E}[ZX^T]w + w\mathbb{E}[XX^T]w = c - 2b^T w + w^T Aw$

where: $A = \mathbb{E}[XX^T]$ and $b = \mathbb{E}[ZX]$

By defining: $f(w) = c - 2b^T w + w^T Aw$, we get the gradient: $\nabla f = 2Aw - 2b$. The optimal value w^* is found as the solution of the system: $\nabla f = 0 \Rightarrow Aw = b$ and its solution could be approximated using numerical linear algebra methods. Thus, while a linear neuron has limited capabilities, it can be trained far more efficiently than more complex neurons.

Adaline

Adaline is another neural network classifier consisting of a single neuron, similarly with the perceptron model. Its activation function is the signum function and its goal was to predict the label of a binary variable. The main difference between the Adaline and the perceptron is that during training, Adaline calibrates its weights based on the weighted sum of the inputs and not in terms of network's output, using an algorithm called α-LMS, which was carried out with electrically variable resistors.

It is notable that for inputs of equal length, the α-LMS algorithm essentially minimizes the mean-squared error using the gradient descent update rule. For this reason, it is considered as the predecessor of gradient based methods.

Madaline

Madaline is one of the first multilayered neural networks, consisting of a hidden layer of Adaline neurons and a fixed logical gate that produces the network's output. Due to the use of signum activation function in the hidden layer, we cannot use the back-propagation algorithm to perform training. Hence, three distinct training algorithms have been proposed, called Rule 1, Rule 2 and Rule 3 respectively.

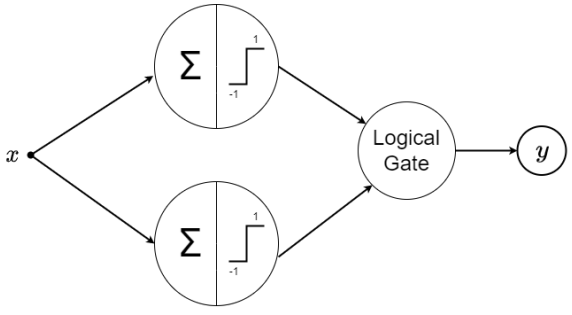


Figure 1.2: Madaline with a hidden layer of two Adaline neurons

Neurons of Continuum Input

A continuum input neuron is a special type of neuron used to construct continuum neural networks. Its input x is continuum over $[0,1]$ and it is equipped with a weighting measure over that interval, thus, each value x has corresponding weight: $wd(x)$. The output of these neurons is calculated as follows:

$y = \sigma(\int_0^1 x dw(x))$

It is easy to see that for different weighting measures, we get different types of continuum input neurons.

· Neuron with Dirac measure: Using the Dirac measure δ_{x_0}, defined for every measurable set $a \in \mathbb{B}([0,1])$ as:

$$\delta_{x_0}(A) = \begin{cases} 1 & x_0 \in A \\ 0 & x_0 \notin A \end{cases}$$

the output of the network is:

$$y = \sigma(\int_0^1 x d\delta_{x_0}(x)) = \sigma(x_0)$$

Because there are no parameters to optimize, there is no need to perform training in this case.

· Neuron with discrete measure: If we have a finite subset X of $[0,1]$ and a discrete measure μ, such that:

$$\mu(A) = \sum_{x_i \in X} w_i \delta_{x_i}(A)$$

then, the output is calculated as follows:

$$y = \sigma(\int_0^1 x d\mu(x)) = \sigma(\sum_{j=1}^n w_j x_j)$$

Using a cost function, such as the mean-squared error, we can then minimize it to obtain the optimal value of weights.

· Neuron with Lebesgue measure: Consider an absolute continuous measure μ with respect to the Lebesgue measure $\lambda_{[0,1]}$. Using the Radon-Nikodym theorem, we get a non-negative measurable function p on $[0,1]$, such that: $d\mu(x) = p(x)dx$. Thus, the network's output is:

$$y = \sigma(\int_0^1 x d\mu(x)) = \sigma(\int_0^1 x p(x) dx)$$

The training's goal is to optimize $p(x)$ in order to minimize the error functional: $F(p) = \frac{1}{2}[\sigma(\int_0^1 x p(x) dx) - z]^2$. Without making more assumptions about measure μ, we cannot proceed any further.

1.3 Feedforward Architecture

Feedforward neural networks (FNN) are one of the most important network architectures, both from a historical and practical perspective. A feedforward network tries to approximate a target function f^* by defining the mapping $y = f(x; w)$ and adjusting the value of parameter vector w during training.

While FNNs have applications in both regression and classification problems and are often deployed, they also act as the foundation stone to move to more complex models, such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks(RNNs), which can be obtained by performing modifications to the basic structure of a FNN.

1.3.1 Architecture Design & Elements

As stated above, the goal of a FNN is to approximate a target function f^* using the mapping f. We can treat f as a function composition, namely, for a FNN with n total layers, f has the form: $f = f^{(n)}(f^{(n-1)}(...(f^{(1)}(x))))$, where each function $f^{(k)}$ represents a layer.

A layer is a vertical stack of neurons. In a FNN we have three possible layers: the input layer, which receives data and passes them to the rest of the network, the output layer that returns the network's output and the hidden layers that lie between the input and the output layer. Each neuron of the hidden layers receives inputs from the neurons of the previous layer and passes its output to neurons of the next layer. If all neurons of every layer are connected with all neurons of the previous and next layer, then the FNN is fully-connected.

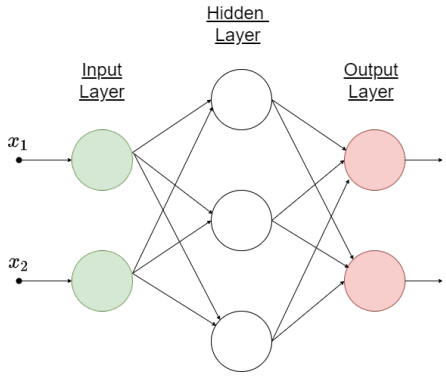


Figure 1.3: Fully-connected FNN

The Importance of the Activation Function

As stated previously, non-linear activation functions enable the network to learn a non-linear target function. Linear activation functions, with the identity being the most common, can make training more fast, but the targets they can approximate are limited. The following results [1] explore the limitations of the identity activation function.

Theorem 1.3.1. *A FNN that uses the identity activation function in all of its layers reduces to a single-layer network that performs linear regression.*

Theorem 1.3.2. *A FNN that uses the identity activation function in all of its hidden layers and its output layer consists of one neuron with the sign activation function and the perceptron criterion as loss function reduces to the perceptron model.*

1.3.2 Training Process

In Subsection 1.2.3 we discussed how some neurons perform their training using specific algorithms or calculating the exact solution of a system. The problem is that in FNNs those methods are rendered useless. From optimization theory we can employ gradient-based methods but the main difficulty here is that we have a function composition and the numerical evaluation of gradients can be computationally prohibitive, especially in FNNs with multiple hidden layers. This caused deep learning research to stagnate for many years, until the back-propagation algorithm solved the problem in a simple and efficient way.

This subsection dives into the training process of FNNs from a general context, treating the optimization algorithms themselves as a black box. The goal is to present the setting, common problems and different strategies of training that are independent of the optimization algorithms, which are discussed in the next Chapter.

Back-Propagation

The back-propagation algorithms consists of two parts: the forward pass, which is also performed when the network makes predictions, and the backward pass. Before we explore the algorithms themselves, we have to study the mathematical background of the algorithm. It is important to note that, although most of the times we treat bias as a member of inputs with its own corresponding weight,

in this case we have to make a distinction between inputs and biases, in order to simplify our calculations.

We start with defining the error for the i-th neuron of the l-th layer (with L being the output layer):

$$\delta_i^l = \frac{\partial L(\hat{y},y)}{\partial z_i^l}$$

which can also be expressed in gradient terms as:

$$\delta_i^l = \nabla_{z^l} \mathcal{L}$$

By applying the chain rule for the output layer's error, we get:

$$\delta_i^L = \sum_{j=1}^{size(l)} \frac{\partial \mathcal{L}}{\partial \hat{y}_j} \cdot \frac{\partial \hat{y}_j}{\partial z_i^l} = \frac{\partial \mathcal{L}}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial z_i^l}$$

Because: $\hat{y}_i = a_i^L = \phi^L(z_i^L)$, where: $z_i^l = {w_i^l}^T a_i^l + b_i^l$, we get:

$$\delta_i^L = \frac{\partial \mathcal{L}}{\partial \hat{y}_i} \phi'^L(z_i^L)$$

and in gradient form:

$$\delta^L = \nabla_{\hat{y}} \mathcal{L} \odot \phi'^L(z^L)$$

This yields the error vector of the output layer. For the other layers, we have:

$$\delta_i^l = \frac{\partial \mathcal{L}}{\partial z_i^l} = \sum_{j=1}^{size(l)} \frac{\partial \mathcal{L}}{\partial z_j^{(l+i)}} \cdot \frac{\partial z_j^{(l+1)}}{z_i^l} = \sum_{j=1}^{size(l)} \delta_j^{(l+1)} \cdot \frac{\partial z_j^{(l+1)}}{\partial z_i^l}$$

The inputs of the i-th neuron of the l-th layer z_i^l satisfy the recursive relationship: $z_j^{(l+1)} = \sum_m w_{jm}^{(l+1)} \phi^l(z_j^l) + b_j^{(l+1)}$. This result in conjunction with the fact that inputs of different neurons of a layer are independent returns:

$$\frac{\partial z_j^{(l+1)}}{\partial z_i^l} = w_{ji}^{(l+1)} \phi'^l(z_i^l)$$

Combining those results above, we receive:

$$\delta_i^l = \sum_{j=1}^{size(l+1)} w_{ji}^{(l+1)} \delta_j^{(l+1)} \phi'^l(z_i^l) = W^{(l+1)} \delta^{(l+1)} \odot \phi'^l(z^l)$$

By further applying the chain rule and performing the necessary calculations, we get the gradients we were looking for:

$$\nabla_{W^l} \mathcal{L} = \delta^l {a^{(l+1)}}^T$$

and

$$\nabla_{b^l} \mathcal{L} = \delta^l$$

Having the above quantities, we can then provide them to a gradient-based optimization algorithm to find the optimal values of weights and biases. Finally, after the back-propagation is performed, we could unify weights and biases again and treat them as one quantity.

Having examined the mathematical background, we can now proceed with the algorithms themselves. As stated above, the back-propagation process consists of a forward and a backward pass, where the forward pass in essentially nested inside the backward. Thus, we have the following pair of algorithms:

Algorithm 1 Forward Propagation

Require: $N \geq 0$: Network depth L
Require: W^l, $l = 1, ..., L$: Weight matrices
Require: b^l, $l = 1, ..., L$: Bias parameters
Require: x: Network's input
Require: y: Network's target output
  $a^0 = x$
  for l in $1, 2, ..., L$ **do**
    $z^l = {W^l}^T a^{(l-1)} + b^l$
    $a^l = \phi(z^l)$
  end for
  $\hat{y} = a^L$
  Calculate: $\mathcal{L}(\hat{y}, y)$

Algorithm 2 Back-Propagation

Require: $N \geq 0$: Network depth L
Require: W^l, $l = 1, ..., L$: Weight matrices
Require: b^l, $l = 1, ..., L$: Bias parameters
Require: x: Network's input
Require: y: Network's target output
  Run Forward Propagation Algorithm
  $\delta^L = \nabla_{z^L} \mathcal{L}(\hat{y}, y)$
  for l in $L-1, L-2, ..., 1$ **do**
    $\delta^l = W^{(l+1)^T} \delta^{(l+1)} \odot \phi'^{(l)}(z^l)$
  end for
  for l in $1, ..., L$ **do**
    $\nabla_{W^l} \mathcal{L}(\hat{y}, y) = \delta^l a^{(l-1)^T}$
    $\nabla_{b^l} \mathcal{L}(\hat{y}, y) = \delta^l$
  end for

Setup and Initialization

This subsection we examine some pre-training issues we face in neural networks and the methodology we use to solve them. Inherently, our perspective here is more applied than the rest of the section, although newest research tries to adopt a more mathematically concrete treatment of the matter.

· **Hyperparameters Tuning**: In our approach we consider two major categories of network parameters:

a) Primary model parameters are the weights and biases and are optimized using back-propagation and numerical optimization algorithms on a training data set.

b) Hyperparameters are additional parameters, such as parameters of deployed optimization algorithms (e.g. learning rates or number of epochs), the number of hidden layers or regularization constants. They are tuned in the validation set using specific methods.

The goal of this subsection is to explore the main hyperparameters tuning methods:

i. Grid Search: Grid search is an exhaustive search method performed on a specific interval of hyperparameters values. The main problem with this method is that its cost increases exponentially with the number of hyperparameters and, thus, rendering it practically unusable in real-life complex problems.

ii. Random Sampling Grid Search: This method is a modification of grid search that draws samples randomly values from the grid search's interval of

hyperparameters values. Because we search only limited values and not the whole interval, random sampling grid search is computationally more efficient.

iii. Bayesian Optimization: Bayesian optimization methods could be deployed in bid data problems, where the training of models can be performed for a very limited amount of times and we need to select only one value of the hyperparameters vector. However, these methods are often too slow in real problems and are not preferred.

· **Feature Preprocessing**: Data preprocessing can sometimes help during training. The main methods are:

i. Mean-centering: The column-wise means are subtracted from each data point.

ii. Normalization: The standard deviation is divided from each feature value.

iii. Standardization: Combination of mean-centering and normalization.

Feature preprocessing is often used as it can help with the problem of ill-conditioning, i.e. the sensitivity of the loss function to different parameters fluctuates.

· **Initialization**: Initialization refers to the process of setting the initial values of primary model parameters before the beginning of training. As it will be shown in the next Chapter, every numerical optimization algorithm requires a starting point in order to work properly. Initialization is important in neural networks, because networks often display stability problems, i.e. the activations' strength of each layer exhibits fluctuations.

In an applied setting, initialization is usually performed through sampling from a zero-mean normal distribution with a small standard deviation value. Random sampling guarantees that initialized values differ from neuron to neuron, an element that prevents the creation of identical features in a layer. The problem with this approach is that it does not take account of the interaction between neurons of different layers, which can be a problem in non fully-connected FNNs. Solutions to this issue have been proposed, such as the Xavier initialization, where the standard deviation is modified as: $\sqrt{\frac{2}{size(nodes_{in})-size(nodes_{out})}}$.

Overfitting & Generalization

A problem faced by high complexity models is overfitting, meaning that the model changes extremely with the introduction of new data, leading to inconsistent predictions. The solution to this problem is regularization, a way of penalizing extreme fluctuations through by controlling the values of primary model parameters. This method is called Penalty-Based Regularization.

When performing penalty-based regularization, we modify the loss function as follows:

$$\mathcal{L}^{new}(\hat{y}, y) = \mathcal{L}(\hat{y}, y) - \Omega(w)$$

The form of $\Omega(w)$ term depends on the specific regularization method. There are two primary methods:

L-1 Regularization: $\Omega(w) = ||w||_1 = \sum_i w_i$

L-2 Regularization: $\Omega(w) = \frac{1}{2}||w||_2^2$

In general, L-2 regularization outperforms L-1. However, L-1 is preferred in specific problem settings because it can act as a feature selector by returning sparse solutions, through deactivation of specific neurons, and, thus, creating a sparse neural network.

It is important to note that the back-propagation algorithm can work with regularization without problems.

1.3.3 Training Strategies

Training a neural network is essentially a complex optimization task. Thus, the solutions we receive may not be optimal enough to have sufficient generalization power. On the other hand, pre-training has shown empirically that through simplification of the initial optimization problem and then gradually adjusting the difficulty can return better solutions. Two approaches of pre-training strategies are continuation and curriculum learning.

Continuation Learning: We start from a simplified version of the original optimization problem and progress until we reach the initial problem. This is done by defining a loss function sequence $\{L_n\}_{n=1}^{N}$ with progressing difficulty using techniques such as blurring. This renders continuation learning a model-centric approach.

Curriculum Learning: We start training from simple data points (the difficulty of a datum is defined by the modelling problem itself) and add difficulty progressively. Thus, curriculum learning is a data-centric approach. Most of the time, best results are obtained by using a random mixture of simple and difficult data, with the proportion of difficult data in the mixture increasing over time. This variant is called Stochastic Curriculum Learning.

Chapter 2

Optimization Theory and Methods

> If you optimize everything, you will always be unhappy.
>
> *Donald Knuth*

2.1 Introduction

The concept of numerical optimization algorithms is to approximate the minimum of a given function, under the existence of some constraints. In deep learning, the use of constraints is not usually an element of the optimization problem due to our ability to modify the cost functions of our toolkit accordingly. However, the study of optimization theory in a general setting is vital to understand the underlying background of those algorithms.

In this Chapter, both general theory and applied methods of optimization in deep learning are presented.

2.2 Convex Optimization: Theory and Numerical Algorithms

In this section we examine deterministic convex optimization theory and algorithms from a theoretical perspective. Although convexity rarely holds in neural

networks, as they can be treated like a composition of functions, convex methods can be the starting point to move to more advanced algorithms. Furthermore, due to their simplicity from a mathematical and a computational viewpoint, they are often deployed in deep learning problems. Thus, a more formal study of their properties and limitations is necessary.

2.2.1 Gradient Descent Algorithm

In this subsection, we introduce one of the most known and basic optimization algorithms; the gradient descent. We concentrate on the mathematical construction of the algorithm and examine possible weaknesses that can become the foundation stone to make the leap to more efficient and powerful methods. A more computational approach of the gradient descent algorithm can be found in the next section.

The concept behind the gradient descent algorithm is to find the minimum of a function f by moving to the direction that minimizes the value of the function for a given step. Thus, we search for the unit vector u, which denotes the direction that decreases the most of the function by moving a small step of size η.

Suppose we start from an initial point x_0. We can express the difference between the function's value before and after we make a step of size η from x_0 using the linear approximation as follows [2]:

$$f(x_0 + \eta u) - f(x_0) = \sum_{j=1}^{n} \frac{\partial f}{\partial x_j} \eta \, u^j + o(\eta^2) = \eta \langle \nabla f(x_0), u \rangle + o(\eta^2)$$

Our goal is to find u using the above result. Because we want to have the maximum negative difference between the value of the function before and after the step and u is a unit vector, i.e. $||u|| = 1$, we use the Cauchy inequality for the dot product of the above formula:

$$-||\nabla f(x_0)|| \, ||u|| \leq \langle \nabla f(x_0), u \rangle \leq ||\nabla f(x_0)|| \, ||u||$$

and conclude that in order to obtain the minimum value, we need:

$$u = -\frac{\nabla f(x_0)}{||\nabla f(x_0)||}$$

We can also quantify the change in the value of the function:

$$f(x_0 + \eta u) - f(x_0) = \eta \langle \nabla f(x_0), u \rangle = -\eta \, ||\nabla f(x_0)||$$

The idea behind the algorithm is to construct a sequence of points: $\{x_n\}_{n=1}^{N}$ using a loop, where in each iteration we move to the direction that minimizes the value of the function for a given step of size $\eta > 0$. We refer to said step size as learning rate. An outline of the algorithm is:

Algorithm 3 Gradient Descent - Mathematical Build

Require: $N \geq 0$: Number of iterations, η: Learning rate
  Initialize starting point of the sequence x_0
  for n in 1, 2, ..., N **do**
    Construct the n-th element of the sequence as follows:
      $x_n = x_{n-1} - \eta \frac{\nabla f(x_{n-1})}{||\nabla f(x_{n-1})||}$
  end for

Although the above solution has the potential to solve the problem, it has a major flaw; by having a fixed learning rate, it is easy to move away from the minimum point x^* we are looking for. The reason is that: $||x_n - x_{n-1}|| = \eta$, thus, the sequence $\{x_n\}_{n=1}^{N}$ does not converge. The solution to this problem is to modify the algorithm, so that the learning rate is not fixed, but dependent on the number of each iteration. We refer to that type of learning rate as adjustable learning rate. A first step would be to make the learning rate proportional to the gradient of the function, hence: $\eta_n = \delta \cdot ||\nabla f(x_n)||$, $\delta > 0$. Consequently, our new algorithm is[2]:

Algorithm 4 Gradient Descent - Final Build

Require: $N \geq 0$: Number of iterations, δ: Learning rate's constant
  Initialize starting point of the sequence x_0
  for n in 1, 2, ..., N **do**
    Construct the n-th element of the sequence as follows:
      $x_n = x_{n-1} - \delta \nabla f(x_{n-1})$
  end for

The above algorithm is the most known form of the gradient descent. Contrary to common belief, this algorithm has the capability (to a degree) to adjust the learning rate to each iteration, thus, the chance of missing the minimum x^* is less than expected.

Now that we have formulated the algorithm, we want to examine its limitations. We have to make two assumptions about the function we want to minimize:

(i) It is differentiable. Otherwise, we cannot provide the algorithm with a gradient during each step.

(ii) It is convex. If the assumption of convexity does not hold, there is no guarantee that we will find a global minimum (or a minimum at all).

The first restriction is not of great importance in deep learning; the majority of loss functions are differentiable (and for non-differentiable functions we can employ some computational tricks performed during back-propagation). However, it is actually vital, from a mathematical viewpoint, to generalize our

results for non differentiable functions as well. On the other hand, the second assumption is very restrictive both from a theoretical and from an applied perspective, because the typical loss landscape consists of hills and valleys, having many minima, maxima and saddle points and, thus, rendering the gradient descent algorithm almost useless.

Finally, we can examine the convergence conditions of the algorithm, by using the following result [2]:

Proposition 2.2.1. *The sequence* $\{x_n\}_{n=1}^N$ *constructed from Algorithm 4 (Gradient Descent - Final Build) is convergent, iff:* $\nabla f(x_n) \xrightarrow{n\to\infty} 0$.

It is easy to see, that if function f is continuously differentiable and x^* is a global minimum of f, then, if: $x_n \to x^*$, by using the above proposition, we have: $\nabla f(x_n) \to \nabla f(x^*) = 0$, which agrees with the necessary condition from theory. Additionally, the learning rate is controlled by the Lipschitz constant of ∇f.

Line Search Method

The line search method is the first modification of the gradient descent algorithm that we will examine and can be considered as an expansion of the concept of adjustable learning rate. The idea is that we choose to move to a point on the direction that the objective function f is minimized [2]. To achieve this, we have to update the learning rate's value as follows:

$$\eta_n = \arg\min_\eta f(x_n - \eta \nabla f(x_n))$$

Thus, we create a new algorithm that generates sequences for both visited points $\{x_n\}_{n=1}^N$ and learning rates $\{\eta_n\}_{n=1}^N$.

Algorithm 5 Gradient Descent - Exact Line Search Method

Require: $N \geq 0$: Number of iterations, η_0: Initial learning rate
  Initialize starting point of the sequence x_0
  for n in 1, 2, ..., N **do**
    Update the learning rate's value:
      $\eta_n = \arg\min_\eta f(x_n - \eta \nabla f(x_n))$
    Construct the n-th element of the sequence as follows:
      $x_{n+1} = x_n - \eta_n \nabla f(x_n)$
  end for

2.2.2 Subgradient Descent

In this section we begin to expand the concept of the gradient descent algorithm to more complex problems. In particular, we generalize the steepest descent method for functions that are not differentiable. Although in the context of deep learning this is not usually the case, it is of great importance from a mathematical standpoint to make this generalization.

We first need to express the general convex optimization problem. The goal is to find the value:

$$f^* := \min_{x \in X} f(x)$$

$$\text{where: } X \subseteq \mathbb{R}^m$$

At first, we revisit the update step of the general gradient descent algorithm with adjustable learning rate:

$$x_{n+1} = x_n - \eta_n \, \nabla f(x_n)$$

The necessary modifications of the above method we need to make are [6]:

(i) Replace the gradient of the objective function with a subgradient, i.e. a member of the subdifferential of f (see Appendix): $g(x_n) \in \partial f(x_n)$, because according to the general convex optimization problem, the objective function is not necessarily differentiable.

(ii) Generalize the update step to solve constrained problems. In the constrained case, where $X \neq \mathbb{R}^m$, the updated point x_{n+1} could drift away from the set X. Thus, we need to use a projection to push it back.

Consequently, during each step we move to the point:

$$x_{n+1} = \arg\min_{x \in X} ||x - (x_n - \eta_n \, g(x_n))||_2^2, \text{ for: } g(x_n) \in \partial f(x_n) \text{ and } \eta_n > 0$$

Finally, we can re-write the above formula in order to have a more manageable form [6]:

$$\begin{aligned} x_{n+1} &= \arg\min_{x \in X} ||x - (x_n - \eta_n \, g(x_n))||_2^2 \\ &= \arg\min_{x \in X} [\eta_n \langle g(x_n), x \rangle + \frac{1}{2} ||x - x_n||_2^2] \end{aligned}$$

Remark 1. *The problem with using subgradients is that a subgradient is not necessarily a descent direction. Therefore, it must be chosen appropriately to ensure that we move towards a direction that decreases the value of f.*

The subgradient descent algorithm has the following form:

Algorithm 6 Subgradient Descent

Require: $N \geq 0$: Number of iterations
  Initialize starting point of the sequence x_0
  for n in 1, 2, ..., N **do**
    Construct the n-th element of the sequence as follows:
      $x_{n+1} = \arg\min_{x \in X} \ [\eta_n \langle g(x_n), x \rangle + \frac{1}{2}||x - x_n||_2^2]$
  end for

Now that we have built the algorithm, we can proceed to examine its mathematical properties and limitations. We start from a general setting of convex optimization problems and then continue with more specialized cases.

General Non-smooth Convex Problems

At first, we consider an objective function that is convex and Lipschitz continuous over a domain X. Our goal is to examine if our method converges to the solution.

The following result can be used to characterize the updated points after each iteration and is the step-stone for the rest of the proofs of this section [6].

Lemma 2.2.1. *Consider the sequence $\{x_n\}_{n=1}^N$ generated by Algorithm 6 (Subgradient Descent) and let $n \in \{1, ..., N\}$. Then:*

$$\eta_n \langle g(x_n), x_{n+1} - x_n \rangle + \frac{1}{2}||x_{n+1} - x_n||_2^2 \leq \frac{1}{2}||x - x_n||_2^2 - \frac{1}{2}||x - x_{n+1}||_2^2$$

Now, we can examine the convergence properties of the subgradient descent [6]:

Theorem 2.2.1. *Consider the sequence $\{x_n\}_{n=1}^N$ generated by Algorithm 6 (Subgradient Descent). Under the assumption of Lipschitz continuity, it holds:*

$$\sum_{n=s}^N \eta_n [f(x_n) - f(x)] \leq \frac{1}{2}[||x - x_n||_2^2 + M^2 \sum_{n=s}^N \eta_n^2$$

Finally, we want to specify the choice for the learning rate η_n [6].

Corollary 2.2.1. *Let: $D_x^2 = max_{x_1, x_2 \in X} \frac{||x_1 - x_2||_2^2}{2}$. Under the assumption that the number of iterations N is fixed and: $\eta_n = \sqrt{\frac{2D_x^2}{NM^2}}$, for: $n = 1, 2, ..., N$, then:*

$$f(\bar{x_1}^N) - f^* = \frac{\sqrt{2}MD_x}{2\sqrt{N}}, \ \forall N \geq 1$$

where:

$$\bar{x_s}^N = (\sum_{n=s}^{N} \eta_n)^{-1} \sum_{n=s}^{N} \eta_n x_n$$

The problem of this approach is that we consider that the number of iterations is known in advance. While this makes sense from a computational perspective, from a mathematical one it is very restrictive. Consequently, we can specify the choice for the learning rate η_n, without fixing the number of iterations [6]:

Corollary 2.2.2. *Suppose:* $\eta_n = \sqrt{\frac{2D_x^2}{nM^2}}$, *for:* $n = 1, 2, ..., N$. *Then:*

$$f(\bar{x}^N_{\lceil N/2 \rceil}) - f^* \leq \frac{(1+2\log 3)MD_x}{2(\sqrt{2}-1)sqrtN+1}, \forall N \geq 3$$

The flaw of the above results is that the output solution is a weighted average of all members of the generated sequence $\{x_n\}_{n=1}^N$, whereas we want to find only one point x_k as the solution. Without any extra limitations for our optimization problem, we can only consider as an output solution the member of the generated sequence $\hat{x}_k$, that minimizes the value of the objective function, i.e. that satisfies:

$$f(\hat{x}_k) = \min_{i=1,...,N} f(x_i)$$

Non-smooth Strongly Convex Problems

In this subsection we add the restriction that our objective function is also strongly convex.

First of all, the following theorem [6] provides an upper bound for the gap between the objective function's value at a point generated by the subgradient algorithm and its value at any point of its domain.

Theorem 2.2.2. *Consider the sequence* $\{x_n\}_{n=1}^N$ *generated by Algorithm 6 (Subgradient Descent). Under the assumptions of Lipschitz Continuity and Strong Convexity, if a sequence* $\{w_n\}_{n=1}^N$, *with* $w_n \geq 0$, $n = 1, ..., N$, *exists, such that:*

$$\frac{w_n(1-\mu\eta_n)}{\eta_n} \leq \frac{w_{n-1}}{\eta_{n-1}}$$

and $x \in X$, *then:*

$$\sum_{n=1}^{N} w_n[f(x_n)-f(x)] \leq \frac{w_1(1-\mu\eta_1)}{2\eta_1}||x-x_1||_2^2 - \frac{w_N}{2\eta_N}||x-x_{N+1}||_2^2 + M^2 \sum_{n=1}^{N} w_n\eta_n$$

Now, we can specify the form of the sequences $\{\eta_n\}_{n=1}^N$ and $\{w_n\}_{n=1}^N$ to examine the gap between our output solution $f(\bar{x_1}^N)$ and the optimal value [6].

Corollary 2.2.3. *If:* $\eta_n = \frac{2}{\mu n}$ *and* $w_n = n$, $\forall n \geq 1$, *then:*

$$f(\bar{x_1}^N) - f(x) + \frac{\mu N}{2(k+1)}||x_{N+1} - x||^2 \leq \frac{4M^2}{\mu(N+1)}, \forall x \in X$$

Thus, we managed to bound both $f(\bar{x}_1^N) - f(x^*)$ and $||x_{N+1} - x^*||^2$ using an $\mathcal{O}(\frac{1}{N})$ term. Finally, in a more applied setting, we could substitute $\bar{x}_1^N$ with $\hat{x}_k$, similarly with the General Non-Smooth Convex case.

Smooth Convex Problems

We assume that our objective function f is differentiable and convex, with Lipschitz continuous gradients, i.e. a smooth convex function. For smooth functions, the subgradient is a singleton consisting of only the gradient and, thus, by replacing the subgradient with the gradient: $g(x_n) = \nabla f(x_n)$ in Algorithm 1 (Subgradient Descent), we create a new algorithm; the projected gradient method.

Algorithm 7 Projected Gradient

Require: $N \geq 0$: Number of iterations
  Initialize starting point of the sequence x_0
  for n in 1, 2, ..., N **do**
    Construct the n-th element of the sequence as follows:
      $x_{n+1} = P_X(x_n - \eta_n f(x_n))$
    where P_X is the projection operator on the closed convex set X.
  end for

Remark 2. *The update step of Projected Gradient Descent algorithm can be alternatively made using the formula:*

$$x_{n+1} = \arg\min_{x \in X} \ [\eta_n \langle \nabla f(x_n), x \rangle + \tfrac{1}{2}||x - x_n||_2^2]$$

As our first result [6], we show that the use of the projected gradient algorithm in smooth convex problems guarantees that the sequence of function values at the points of the iteration $\{f(x_n)\}_{n=1}^N$ is monotonically non-increasing.

Lemma 2.2.2. *Consider the sequence* $\{x_n\}_{n=1}^N$ *generated by Algorithm 7 (Projected Gradient). Under the additional assumption of smoothness and for learning rate:* $\eta_n \leq \frac{2}{n}$, *we have:*

$$f(x_{n+1}) \leq f(x_n)$$

Using the above result, we can examine the convergence properties of the algorithm with fixed learning rate in the context of smooth convex problems [6].

Theorem 2.2.3. *Let the sequence $\{x_n\}_{n=1}^N$ be generated by Algorithm 7 (Projected Gradient). If the assumption of smoothness holds and for learning rate: $\eta_n = \eta = \frac{1}{L}$, $\forall n \geq 1$, then:*

$$f(x_{N+1}) - f(x) \leq \frac{1}{2\eta N}||x - x_1||_2^2, \forall x \in X$$

Smooth Strongly Convex Problems

Having examined each case individually, we can now combine the results of the two previous subsections to examine the convergence properties of the projected gradient algorithm.

Consider the sequence $\{x_n\}_{n=1}^N$ generated by Algorithm 7 (Projected Gradient). Under the assumption that smoothness and strongly convexity hold and for the choice of fixed learning rate: $\eta_n = \eta = \frac{1}{L}$, $\forall n = 1, ..., N$, we have:

$$||x - x_{N+1}||_2^2 \leq (1 - \frac{\mu}{L})^N \ ||x - x_1||_2^2$$

Finally, in order to find a solution x_s convergent to the global minimum x^* (i.e. $||x_S - x^*||^2 \leq \epsilon$, $\forall \epsilon > 0$), by using the above theorem we have:

$$(1 - \frac{\mu}{L})^N ||x^* - x_1||_2^2 \leq \epsilon \Rightarrow N \geq \frac{L}{\mu} \ \log\Big(\frac{||x^* - x_1||_2^2}{\epsilon}\Big)$$

In conclusion, for smooth and strongly convex objective functions, if we choose learning rate $\eta = \frac{1}{L}$ and number of iterations $N \geq \frac{L}{\mu} \ \log\Big(\frac{||x^* - x_1||_2^2}{\epsilon}\Big)$, it is proven that the projected gradient method converges to the global minimum.

2.2.3 Mirror Descent

In the previous subsection we examined the convergence properties of the subgradient descent algorithm under various assumptions for our objective function. Although the subgradient method has the ability to adapt to the geometry of the problem through the use of projections, it is linked with the Euclidean structure of $\mathbb{R}^m$. Hence, in this methods, the update step is similar to that of the subgradient descent, but instead of the Euclidean norm, we use Bregman's distance V (see Appendix). Our next move would be to generalize the subgradient descent algorithm to adjust to non-Euclidean optimization problems. This ability can provide better accuracy certificates in specific optimization settings.

The Mirror Descent algorithm has the following form:

Algorithm 8 Mirror Descent

Require: $N \geq 0$: Number of iterations
  Initialize starting point of the sequence x_0
  for n in 1, 2, ..., N **do**
    Construct the n-th element of the sequence as follows:
      $x_{n+1} = \arg\min_{x \in X} \left[\eta_n \langle g(x_n), x\rangle + V(x_n, x)\right]$
  end for

The Lemma below is important for the proofs of this subsection [6].

Lemma 2.2.3. *Consider the sequence $\{x_n\}_{n=1}^N$ generated by Algorithm 8 (Mirror Descent). For any $x \in X$, it holds:*

$$\eta_n \langle g(x_n), x_{n+1} - x\rangle + V(x_n, x_{n+1}) \leq V(x_n, x) - V(x_{n+1}, x)$$

Having stated the above Lemma, we can now examine the convergence properties of the mirror descent algorithm [6]. In order to move away from the Euclidean structure, we impose the extra assumption for the subgradients: $||g(x_n)||_* \leq M$.

Theorem 2.2.4. *Let $\{x_n\}_{n=1}^N$ the sequence generated by Algorithm 8 (Mirror Descent). Then:*

$$f(\bar{x_s}^N) - f^* \leq (\sum_{n=s}^N \eta_n)^{-1}[V(x_s, x^*) + \tfrac{1}{2}M^2 \sum_{n=s}^N \eta_n^2]$$

where: x^* is an arbitrary solution of the general convex optimization problem.

Finally, we can provide a specific choice of constant step-size [6].

Corollary 2.2.4. *Under fixed number of iterations and choosing: $\eta = \sqrt{\frac{2D_x^2}{NM^2}}$, where: $D_x^2 := max_{x_1,x} V(x_1, x)$, we have:*

$$f(\bar{x_1}^N) - f^* \leq \frac{\sqrt{2}MD_x}{\sqrt{N}}$$

We can also introduce variable learning rates without knowing the iterations' number in advance [6].

Corollary 2.2.5. *If: $\eta_n = \frac{D_x}{M\sqrt{n}}$, then:*

$$f(\bar{x}_{\lceil \frac{N}{2} \rceil}) - f^* \leq \mathcal{O}(1)(\frac{MD_x}{\sqrt{N}})$$

By comparing our accuracy certificates for both subgradient and mirror descent, we can see that for both methods the gap between the algorithm's solution and the real one is bounded by a $\mathcal{O}(\frac{1}{\sqrt{N}})$ term. Hence, from an algorithmic perspective the two methods perform similarly. The potential benefit of the mirror descent over the subgradient descent method is that it generally converges faster in strongly convex or smooth problems.

2.2.4 Accelerated Gradient Descent

The objective of this subsection is to present the accelerated gradient descent algorithm; a significant improvement of the simple gradient descent method, in the setting of smooth convex optimization problems.

At first, we will re-define strong convexity using Bregman's distance: A function f is strongly convex in terms of Bregman's distance if: $\exists \mu \geq 0$, such that:

$$f(y) \geq f(x) + \langle \nabla f(x_n), y - x \rangle + \mu V(x, y)$$

This alternative definition is a vital part of the update step of our algorithm.

The Vanilla Accelerated Gradient Descent algorithm has the following form:

Algorithm 9 Accelerated Gradient Descent

Require: $N \geq 0$: Number of iterations
  Initialize starting points x_0 and $\bar{x_0}$
  for n in 1, 2, ..., N **do**
    Construct the n-th search point:
      $\underline{x}_n = (1 - q_n)\bar{x}_{n-1} + q_n x_{n-1}$
    Construct the n-th search point used for proximity control:
      $x_n = \arg\min_{x \in X} \{\eta_n [\langle \nabla f(\underline{x}_n), x \rangle + \mu V(\underline{x}_n, x)] + V(\underline{x}_{n-1}, x)\}$
    Construct the n-th point of the output solution sequence:
      $\bar{x}_n = (1 - a_n)\bar{x}_{n-1} + a_n x_n$
  end for

where: $q_n \in [0, 1]$, $\eta_n \geq 0$ and $a_n \in [0, 1]$ are hyper-parameters of the algorithm.

We can now proceed to examine general convergence properties of the algorithm in smooth optimization problems [6]. A first step would be to revisit the update step of the proximity control update point x_n.

Lemma 2.2.4. *Consider a convex function $p : X \to \mathbb{R}$, the points $\tilde{x}, \tilde{y} \in X$ and the scalars $\mu_1, \mu_2 \geq 0$. If the distance generating function of Bregman's divergence $v : X \to \mathbb{R}$ is differentiable convex and:*

$$\hat{u} \in \arg\min\{p(u) + \mu_1 V(\tilde{x}, u) + \mu_2 V(\tilde{y}, u); u \in X\}$$

then $\forall u \in X$, it holds:

$$p(\hat{u}) + \mu_1 V(\tilde{x}, \hat{u}) + \mu_2 V(\tilde{y}, \hat{u}) \leq p(u) + \mu_1 V(\tilde{x}, u) + \mu_2 V(\tilde{y}, u)$$

We can now examine a recursive property of the accelerated gradient descent method [6].

Proposition 2.2.2. *Consider $(\underline{x}_n, x_n, \bar{x}_n) \in X \times X \times X$, generated by Algorithm 9 (Accelerated Gradient Descent). If:*

$$a_n \geq q_n,\ \tfrac{L(a_n - q_n)}{1 - q_n} \leq \mu \text{ and: } \tfrac{Lq_n(1-a_n)}{1-q_n} \leq \tfrac{1}{\eta_n}$$

then $\forall x \in X$:

$$f(\bar{x}_n) - f(x) + a_n(\mu + \tfrac{1}{\eta_n})V(x_n, x) \leq (1 - a)[f(\bar{x}_{n-1}) - f(x)] + \tfrac{a_n}{\eta_n} V(x_{n-1}, x)$$

Finally, the following result explores the convergence properties of the accelerated gradient descent methods in smooth convex optimization problems [6]:

Theorem 2.2.5. *Consider $(\underline{x}_n, x_n, \bar{x}_n) \in X \times X \times X$. If:*

$$a_n = q_n,\ La_n \leq \tfrac{1}{\eta_n} \text{ and: } \tfrac{\eta_n(1-a_n)}{a_n} \leq \tfrac{\eta_{n-1}}{a_{n-1}},\ \forall n = 1, ..., N$$

then it holds:

$$f(\bar{x}_N) - f(x^*) + \tfrac{a_N}{\eta_N} V(x_N, x^*) \leq \tfrac{a_N \eta_1 (1-a_1)}{\eta_N a_1}[f(\bar{x}_0) - f(x^*)] + \tfrac{a_n}{\eta_n} V(x_0, x^*)$$

Specifically, for the selection: $q_n = a_n = \frac{2}{n+1}$ and: $\eta_n = \frac{n}{2L}$, we have:

$$f(\bar{x}_n) - f(x^*) \leq \tfrac{4L}{N(N+1)} V(x_0, X^*)$$

An important application of the above theorem is that in order to obtain an output solution $\bar{x} \in X$ with an accuracy certificate of ϵ (i.e. $f(\bar{x}) - f(x^*) \leq \epsilon$), we need: $\mathcal{O}(\frac{1}{\sqrt{\epsilon}})$ iterations. In order to converge to a solution faster, we have to impose more restrictions to our problem [6]; namely, move to the smooth and strongly convex case ($\mu > 0$).

Theorem 2.2.6. *Consider $(\underline{x}_n, x_n, \bar{x}_n) \in X \times X \times X$, generated by Algorithm 9 (Accelerated Gradient Descent). If all hyper-parameters are fixed and satisfy the restrictions of Proposition 2.2.2, with the additional restriction that: $\frac{1}{\eta(1-a)} \leq \mu + \frac{1}{\eta}$, then $\forall x \in X$:*

$$f(\bar{x}_N) - f(x) + a(\mu + \tfrac{1}{\eta})V(X_{N-1}, x) \leq (1-a)^N[f(\bar{x}_0) - f(x) + a(\mu + \tfrac{1}{\eta})V(x_1, x)]$$

Additionally, for the specific selection of: $a = \sqrt{\frac{\mu}{L}}, q = \frac{a - \frac{\mu}{L}}{1 - \frac{\mu}{L}}$ *and:* $\eta = \frac{a}{\mu(1-a)}$*, we get:*

$$f(\bar{x}_N) - f(x) + a(\mu + \tfrac{1}{\eta})V(X_{N-1}, x) \leq$$
$$(1-a)^N[f(\bar{x}_0) - f(x) + (1 - \sqrt{\tfrac{\mu}{L}})^N(\mu + \tfrac{1}{\eta})V(x_1, x)]$$

Now, we can show that the number of iterations required to get an output solution that guarantees ϵ divergence from the optimal solution is bounded by an $\mathcal{O}(\sqrt{\frac{L}{\mu}} \cdot log(\frac{1}{\epsilon}))$ term.

2.3 Regarding the Implementation of Gradient Methods

After the mathematical treatment of the convex optimization algorithms, we are ready to revisit the topic from a more applied and computational viewpoint and examine the methods that are used to solve real-life large-scale convex optimization problems.

2.3.1 Gradient Descent

A first step would be to revisit the simple Gradient Descent Algorithm. From a computational perspective, it is a greedy algorithm with complexity $\mathcal{O}(N)$, where: N is the total number of iterations, under the assumption that we can be provided with the value of the gradient at any point in $\mathcal{O}(1)$ time. In the case of unconstrained problems the Projected Gradient becomes the Gradient Descent algorithm and, thus, the accuracy certificates examined in Section 2 can be directly applied to the Gradient Descent method.

2.3.2 Line Search Methods

While the Gradient Descent is a straightforward algorithm, its Line Search variants in applied problems differentiate themselves significantly more from the theoretical ones.

From an applied perspective, the problem of line search is to find an appropriate step length that reduces adequately the value of the objective function f. Consider the function ϕ:

$$\phi(a) = f(x_n - \eta \nabla f(x_n)),\ a > 0$$

The theoretical algorithm (Algorithm 5 - Exact Line Search) assumes that we have at our disposal a global optimizer of ϕ at any step. However, in most problem settings, even a local optimizer of ϕ can have a prohibitive computational cost. For this reason, an alternative methodology has been developed; inexact line search methods.

Inexact Line Search

The idea of inexact line search methods is to obtain a satisfying step size at minimal cost [7]. One of the most efficient inexact methods is backtracking line search.

Algorithm 10 Backtracking Line Search

Require: $N \geq 0$: Number of iterations and decrease criterion C
  Initialize starting point $\eta^{(0)} > 0$ and $l = 0$
  for n in 1, 2, ..., N **do**
    while $C(f(x_n + \eta^{(l)} \nabla f(x_n)), f(x_n))$ not True **do**
      $\eta^{(l+1)} = \tau \eta^{(l)}$, $\tau \in (0, 1)$
      $l = l + 1$
    end while
    $\eta_n = \eta^{(l)}$
  end for

Backtracking line search is a step to more applied algorithms with enhanced computational performance. However, in order to have a complete algorithm, we need to specify the algorithm's decrease criterion C. One of the most used approaches is the Armijo Rule. From a general perspective, instead of $\nabla f(x_n)$, we move to a direction p_n at every step (in steepest descent methods those two are identical). Then, the Armijo rule has the following form:

$$f(x_n + \eta p_n) \leq f(x_n) + c_1 a \nabla f(x_n)^T p_n$$

It is important to mention that the Armijo rule is the first of the Wolfe conditions and is also referred to as sufficient decrease condition [7]. The second Wolfe condition; curvature condition:

$$\nabla f(x_n + \eta_n p_n)^T p_n \geq c_2 \nabla f(x_n)^T p_n$$

Wolfe conditions impose more restrictions but generally guarantee a better step size.

2.3.3 Momentum Methods

It could be said that the momentum methods are a more applied version from the general and theoretical Accelerated Gradient Descent algorithm. The main idea behind them is that by introducing velocity (or extra energy) to our system, then the algorithm could avoid getting stuck in local minima. The target is to reach a stable equilibrium; hopefully a global minimum. The standard momentum algorithm has the following form:

Algorithm 11 Momentum Method

Require: $N \geq 0$: Number of iterations
  Initialize starting points x_0 and u_0
  for n in 1, 2, ..., N **do**
    Update the velocity:
      $u_n = \mu u_{n-1} - \eta \nabla f(x_{n-1})$
    Construct the n-th search point:
      $x_n = x_{n-1} + u_n$
  end for

where $\eta > 0$ and $\mu \in [0, 1]$ is the momentum coefficient. It is easy to see that for: $\mu = 0$, the momentum method becomes the gradient descent.

Convergence Analysis

At first, we can obtain the following important recursive property for velocity:
$u_{n+1} = \mu_{n+1} u_0 - \eta \sum_{i=0}^{n} \mu_{n-i} b_i$

The following two results are important to examine the convergence properties of the momentum method [2].

Proposition 2.3.1. *Let $\{a_n\}_n$ be a sequence of real numbers that converges to 0 and $\sum_{n=0}^{\infty} b_n$ be an absolute convergent series of real numbers. Then:*

$$lim_{n\to\infty} \sum_{i=0}^{n} a_i b_{n-i} = 0$$

Also, by using the definition of convolution series we can show [2]:

Theorem 2.3.1. *Consider two series of real numbers: $\sum_{n=0}^{\infty} a_n$ and $\sum_{n=0}^{\infty} b_n$, one convergent and the other absolute convergent. Then, their convolution series is convergent and it holds:*

$$\sum_{n=0}^{\infty} (\sum_{i=0}^{n} a_i b_{n-i}) = (\sum_{n=0}^{\infty} a_n) \, (\sum_{n=0}^{\infty} b_n)$$

Now we can obtain convergence results for the sequences: $\{x_n\}_{n\geq 0}$ and $\{u_n\}_{n\geq 0}$ [2].

Proposition 2.3.2. *i. If $\nabla f(x_n)$ converges to 0, then the momentum sequence $\{u_n\}_{n\geq 0}$ is also convergent to 0.*

ii. If $\sum_{n=0}^{\infty} ||\nabla f(x_n)||$ is convergent, then, both the momentum and search point sequences $\{x_n\}_{n\geq 0}$ and $\{u_n\}_{n\geq 0}$ are convergent.

Nesterov Accelerated Gradient

One of the most important variants of the Momentum Method is the Nesterov Accelerated Gradient, which adjusts better the velocity at each step, rendering the method more stable than the Vanilla Momentum Method. Using empirical evidence, we could conclude that the Nesterov Accelerated Gradient algorithm performs better than any other deterministic convex optimization method, while also having the capacity to escape local minima.

In order to build the algorithm, we have to modify the update step of velocity as follows:

Algorithm 12 Nesterov Accelerated Gradient

Require: $N \geq 0$: Number of iterations
  Initialize starting points x_0 and u_0
  for n in 1, 2, ..., N **do**
    Construct the velocity:
      $u_n = \mu u_{n-1} - \eta \nabla f(x_{n-1} + \mu u_{n-1})$
    Update the n-th search point:
      $x_n = x_{n-1} + u_n$
  end for

2.4 Stochastic Convex Optimization: Theory and Numerical Algorithms

A crucial drawback of deterministic convex optimization methods is their slow performance, especially when dealing with a large volume of data, a typical case in the context of machine learning. Stochastic optimization algorithms are a big leap towards more fast methods, promising faster convergence results without sacrificing too much accuracy.

This section has a similar format with the previous one; we first examine the algorithms from a mathematical perspective and then move to a more computational and applied approach. It is important to mention that due to the nature of these methods, a more rigorous and detailed mathematical analysis is imperative in order to understand the underlying mechanisms of those algorithms.

2.4.1 Stochastic Mirror Descent

We first start with the stochastic variant of the mirror descent algorithm. Consider the following modification of the general optimization problem:

$f^* := min_{x \in X}\{f(x) = \mathbb{E}[F(x, \xi)]\}$

where: $X \subset \mathbb{R}^m$ non-empty bounded closed convex set and ξ a random vector with probability distribution P, supported on set $\Xi \subset \mathbb{R}^d$ and:
$F : X \times \Xi \to \mathbb{R}$.

Under the assumptions that $\forall x \in X$, the function $F(., \xi)$ is convex on X and that the expectation: $\mathbb{E}[F(x, \xi)] = \int_{\Xi} F(x, \xi) dP(\xi)$ is well defined and finite valued, then we have that function f is also convex and finite valued on X. With the extra assumption that f is continuous on X, the modified general optimization problem stated above becomes a convex optimization problem.

One important hurdle we have to overcome in order to solve the problem is to compute the expectation $E[F(x, \xi)]$ with satisfying accuracy in higher dimensions. For this reason we adopt a computational approach based on Monte Carlo methods.

Before we examine the stochastic mirror descent we need to state the following assumptions [6]:

1) It is possible to generate an i.i.d. sample $\xi_1, ..., \xi_n$ of realizations of the random vector ξ.

2) There is a stochastic first-order oracle, which $\forall x \in X$ and $\xi \in \Xi$ returns a stochastic subgradient, i.e. a vector $G(x, \xi) \in \partial F(x, \xi)$, such that: $g(x) := \mathbb{E}[G(x, \xi)]$ is well defined.

3) For the stochastic subgradient descent, we assume that $\forall x \in X$:

i. $\mathbb{E}[G(x, \xi_t)] = f'(x) \in \partial\Psi(x)$

ii. $\mathbb{E}[||G(x, \xi_t) - f'(x)||_*^2] \leq \sigma^2$

The stochastic mirror descent algorithm has the following form:

Algorithm 13 Stochastic Mirror Descent

Require: $N \geq 0$: Number of iterations
Initialize starting point of the sequence x_0
for n in 1, 2, ..., N **do**
Construct the n-th element of the sequence as follows:
$x_{n+1} = \arg\min_{x \in X} [\eta_n \langle G_n, x \rangle + V(x_n, x)]$
where: $G(x_n) := G(x_n, \xi_n)$
end for

Now, we can examine the convergence properties of this algorithms under certain constrains for our objective function f.

General Non-smooth Convex Problems

At first, we assume that our objective function f is convex and Lipschitz continuous over X. This implies that f has bounded subgradients. It is easy to see that Lemma 3.1.3 still holds if we replace g_n with G_n. Using this remark, we can examine the convergence properties of the stochastic mirror descent algorithm applied in non-smooth convex optimization problems [6].

Theorem 2.4.1. *Consider $\{x_n\}_{n=1}^N$ be generated by Algorithm 13 (Stochastic Mirror Descent). Then:*

$$\mathbb{E}[f(\bar{x}_s^N)] - f^* \leq \left(\sum_{n=s}^N \eta_n\right)^{-1} \left[\mathbb{E}[V(x_s, x^*)] + (M^2 + \sigma^2) \sum_{n=s}^N \eta_n^2\right]$$

where: x^* is an arbitrary solution of the modified general optimization problem stated above and the expected value is taken with respect to $\xi_1, ..., \xi_n$.

Under the assumption that the number of iterations N is known, by optimizing the above formula with respect to the learning rate we get the following value of learning rate:

$$\eta_n = \frac{D_x}{\sqrt{N(M^2+\sigma^2)}}$$

This gives us an accuracy estimate:

$$\mathbb{E}[f(\bar{x}_1^N) - f(x^*)] \leq 2D_x \sqrt{\frac{M^2+\sigma^2}{N}}$$

Our above accuracy certificate is an upper bound of the expected approximation error. In order to have a complete picture of the strength of the stochastic mirror descent method, we also have to examine how those bounds behave on probabilities with large deviations. Using the Markov Inequality to the expected approximation error, we get:

$$Pr[f(\bar{x}_1^N) - f(x^*) > \epsilon] \leq \frac{2D_x(M^2+\sigma^2)}{\epsilon\sqrt{N}},\ \forall \epsilon > 0$$

Thus, in order to obtain a solution of the optimization problem, such that: $Pr[f(\bar{x}_1^N) - f(x^*) > \epsilon] < k$, for $k \in (0,1)$, we need to run the stochastic mirror descent algorithm for: $\mathcal{O}(\frac{D_x^2(M^2+\sigma^2)}{k^2\epsilon^2})$ iterations.

We can obtain better probability bounds by making additional assumptions for the distribution $G(x,\xi)$. In particular, we suppose that $\forall x \in X$, holds:

$$\mathbb{E}\{\exp\{\frac{||G(x,\xi)-g(x)||_*^2}{\sigma^2}\}\} \leq \exp\{1\}$$

Using the above assumption (Assumption 4), we get the following result for the martingale-difference sequence [6]:

Lemma 2.4.1. *Consider the sequence of i.i.d. random variables $\xi_{[N]} = \{\xi_n\}_{n=1}^N$ and $\zeta_N = \zeta(\xi_{[N]})$ deterministic Borel functions of $\xi_{[N]}$, with: $\mathbb{E}_{\xi_{[N-1]}}(\zeta_N) = 0$ almost surely (a.s.), and: $\mathbb{E}_{\xi_{[N-1]}}[\exp\{\frac{\zeta_n^2}{\sigma_N^2}\}] \leq \exp\{1\}$ a.s. for $\sigma_n > 0$ deterministic. Then:*

$$\forall \lambda \geq 0\text{: } Pr\{\sum_{n=1}^N \zeta_n > \lambda\sqrt{\sum_{n=1}^N \sigma_n^2}\} \leq \exp\{-\frac{\lambda^2}{3}\}$$

Proposition 2.4.1. *Under the above assumption and for constant learning rates: $\eta_n = \frac{D_x}{\sqrt{N(M^2+\sigma^2)}}$, $\forall \lambda > 0$ we have:*

$$Pr\{f(\bar{x}_1^N) - f(x_n) > \frac{3D_x}{\sqrt{n}}(\sqrt{M^2+\sigma^2} + \lambda\sigma)\} \leq \exp\{-\lambda\} + \exp\{-\frac{\lambda^2}{3}\}$$

Using the above result, we can bound the required number of iterations to find a solution: $Pr[f(\bar{x}_1^N) - f(x^*) > \epsilon] < k$ with: $\mathcal{O}(\frac{D-x^2(M^2+\sigma^2)\log\left(\frac{1}{k}\right)}{\epsilon^2})$.

Accuracy Certificates

In this subsection we propose lower and upper bound estimates for the optimal value of the modified general optimization problem stated previously in general non-smooth convex settings using the stochastic mirror descent algorithm.

Let N be the number of iterations. Then, we define:

$$v_n = \frac{\eta_n}{\sum_{i=1}^n \eta_i} \text{ and } \tilde{x}_N = \sum_{n=1}^N v_n x_n$$

We also define the following functions:

$$f^N(x) = \sum_{n=1}^{N} u_n[f(x_n) + g(x_n)^T(x - x_n)]$$
$$\hat{f}^N(x) = \sum_{n=1}^{N} v_n[F(x_n, \xi_n) + G(x_n, \xi_n)^T(x - x_n)]$$

Finally, we denote:

$$f^N_{down} = min_{x \in X} f^k(x) \text{ and } f^N_{up} = \sum_{n=1}^{N} v_n f(x_n)$$

It is easy to see that f^N_{down} and f^N_{up} constitute lower and upper bounds of f^* respectively. Because $v_n > 0$ and $\sum_{n=1}^{N} v_n = 1$, f^N_{down} underestimates f on X due to the convexity of f. On the other hand, since $\tilde{x}_N \in X$, it holds: $f^* \leq f(\tilde{x}_N)$ and following the convexity of f: $f(\tilde{x}_N) \leq f^N_{up}$.

We also have that:

$$\mathbb{E}(f^N_{down}) \leq f^* \leq \mathbb{E}(f^N_{up})$$

The problem with the above quantities is that they are unknown. Thus, we will use the equivalent computable functions [6]:

$$\underline{f}^N = min_{x \in X} \hat{f}^N(x) \text{ and } \bar{f}^N = \sum_{n=1}^{N} v_n F(x_n, \xi_n)$$

Proposition 2.4.2. *It holds:*

$$\mathbb{E}(\underline{f}^N) \leq f^* \leq \mathbb{E}(\bar{f}^N)$$

Consequently, $\underline{f}^N$ and $\bar{f}^N$ are also upper and lower bounds of the optimal solution.

We now have to evaluate the accuracy of those bounds. First, we consider the following quantities: $\Delta_n = F(x_n, \xi_n) - f(x_n)$ and $\delta_n = G(x_n, \xi_n) - g(x_n)$. By making the assumption about Δ_n, that: $\exists Q > 0$, such that $\forall n \geq 0$: $E(\Delta_n^2) \leq Q^2$, we can get the following result [6]:

Lemma 2.4.2. *Let $\{\zeta_n\}_{n=1}^{j}$ sequence of $\mathbb{R}^n$ elements and construct the sequence $\{v_n\}$ in X^0 with the following recursive way:*

$$\cdot\, v_1 \in X^0$$
$$\cdot\, v_{n+1} = \arg\min_{x \in X}\{\langle \zeta_n, x \rangle + V(v_n, x)\}$$

Then, $\forall x \in X$, it holds:

$$\cdot \langle \zeta_n, v_n - x \rangle \leq V(v_n, x) - V(v_{n+1}, x) + \frac{||\zeta_n||_*^2}{2}$$
$$\cdot \sum_{n=1}^{j} \langle \zeta_n, v_n - x \rangle \leq V(v_1, x) + \frac{1}{2} \sum_{t=1}^{j} ||\zeta_n||_*^2$$

Having the above Lemma, we can examine the accuracy of the aforementioned lower and upper bounds [6].

Theorem 2.4.2. *Suppose that the Assumptions 1-3 stated at the beginning of the section and the assumption about Δ_n hold. Then:*

$$\cdot \mathbb{E}(f_{up}^N - f_{down}^N) \leq \frac{4D_x^2 + (2M^2 + 3\sigma^2) \sum_{n=1}^N \eta_n^2}{2 \sum_{n=1}^N \eta_n}$$
$$\cdot \mathbb{E}(|\bar{f}^N - f_{up}^N|) \leq Q \sqrt{\sum_{n=1} N v_n^2}$$
$$\cdot \mathbb{E}(|\underline{f}^N - f_{down}^N|) \leq \frac{D_x^2 + (M^2 + \sigma^2) \sum_{n=1}^N \eta_n^2}{\sum_{n=1}^N \eta_n} + (Q + 2\sqrt{2} D_x \sigma) N^{-\frac{1}{2}}$$

Moreover, adopting constant learning rates: $\eta_n = \frac{D_x}{\sqrt{N(M^2+\sigma^2)}}$, we get:

$$\cdot \mathbb{E}(f_{up}^N - f_{down}^N) \leq \frac{7 D_x \sqrt{M^2 + \sigma^2}}{2\sqrt{N}}$$
$$\cdot \mathbb{E}(|\bar{f}^N - f_{up}^N|) \leq Q N^{-\frac{1}{2}}$$
$$\cdot \mathbb{E}(|\underline{f}^N - f_{down}^N|) \leq \frac{2 D_x \sqrt{M^2 + \sigma^2}}{\sqrt{N}} + (Q + 2\sqrt{2} D_x \sigma) N^{-\frac{1}{2}}$$

Smooth Convex Problems

In this subsection we make the additional assumption that f is a convex function with Lipschitz continuous gradient.

By observing that the following inequality holds:

$$||g(x_t)||_*^2 \leq 2||g(x_1)||_*^2 + 2L^2 D_x^2,$$

we can get an upped bound for the rate of convergence of the stochastic mirror descent $\mathbb{E}[f(\bar{x}_1^N) - f(x^*)]$:

$$\mathcal{O}(1)[\frac{D_x(||g(x_1)||_* + L D_x + \sigma)}{\sqrt{N}}]$$

The main problem with the direct application of the stochastic mirror descent in smooth convex problem is that we do not take advantage of the smoothness properties of f. We can get better results by modifying the algorithm as follows:

Algorithm 14 Modified Stochastic Mirror Descent

Require: $N \geq 0$: Number of iterations
  Initialize starting point of the sequence x_0
  for n in 1, 2, ..., N **do**
    Construct the $n+1$-th element of the sequence as follows:
      $x_{n+1} = \arg\min_{x \in X} \left[\eta_n \langle G_n, x\rangle + V(x_n, x)\right]$
      where: $G(x_n) := G(x_n, \xi_n)$
    Construct the $n+1$-th element of the output solution as follows:
      $x_{n+1}^{mod} = (\sum_{i=1}^{n} \eta_i)^{-1} \sum_{i=1}^{n} \eta_i x_{i+1}$
  end for

At first, we need the following recursive result [6].

Lemma 2.4.3. *Consider learning rates, such that:* $L\eta_n < 1$, $\forall n > 1$ *and define* $\delta_n = G(x_n, \xi) - g(x_n)$. *Then:*

$$\eta_n[f(x_{n+1}) - f(x_n)] + V(x_{n+1}, x) \leq V(x_n, x) + \Delta_n(x),\ \forall x \in X$$

where:

$$\Delta_n(x) = \eta_n \langle \delta_n, x - x_n\rangle + \frac{||\delta_n||_*^2 \eta_n^2}{2(1 - L\eta_n)}$$

We can now examine the convergence properties of the modified stochastic mirror descent for general learning rate η_n [6].

Theorem 2.4.3. *Consider* $\{x_n^{mod}\}_{n \geq 1}$ *the sequence generated by Algorithm 14 (Modified Stochastic Mirror Descent) using learning rate:* $0 < \eta_n < \frac{1}{2}L$, $\forall n \geq 1$. *Then:*

i. Under Assumption 3, it holds:

$$\mathbb{E}[f(x_{N+1}^{mod}) - f^*] \leq K_0(N),\ \forall N \geq 1$$

where:

$$K_0(N) = (\sum_{n=1}^{N} \eta_n)^{-1}[D_x^2 + \sigma^2 \sum_{n=1}^{N} \eta_n^2]$$

ii. Under Assumptions 3 and 4, $\forall \lambda \geq 0$, *and* $N \geq 1$*:*

$$Pr\{f(x_{N+1}^{mod}) - f^* > K_0(N) + \lambda K_1(N)\} \leq \exp\left\{-\frac{\lambda^2}{3}\right\} + \exp\{-\lambda\}$$

where:

$$K_1(N) = (\sum_{n=1}^{N} \eta_n)^{-1}[D_x \sigma \sqrt{\sum_{n=1}^{N} \eta_n} + \sigma^2 \sum_{n=1}^{N} \eta_n^2]$$

Thus, the modified stochastic mirror descent provides us with a sharper upper bound for the rate of convergence than the direct application of the stochastic mirror descent.

It is notable that mirror descent algorithms and their variants are not optimal when applied to smooth and strongly convex problems, rendering the mathematical properties of their convergence for these problems unnecessary.

2.4.2 Stochastic Accelerated Gradient Descent

The stochastic accelerated gradient descent solves the family of optimization problems:

$$\Psi^* = \min_{x \in X} \{\Psi(x) = f(x) + h(x)\}$$

where $X \in \mathbb{R}^m$ is closed and convex, h is a convex function with known structure and f is a general convex function, such that for $L \geq 0$, $M \geq 0$ and $\mu \geq 0$:

$$\mu V(x, y) \leq f(y) - f(x) - \langle f'(x), y - x \rangle \leq \frac{L}{2}||y - x||^2 + M||y - x||$$

with: $f'(x) \in \partial f(x)$. We only have access to stochastic first order information about f, i.e. for a given $x_t \in X$, the stochastic first order oracle returns: $F(x_t, \xi)$ and $G(x_t, \xi)$, such that:

$$E[F(x_t, \xi)] = f(x_t) \text{ and } E[G(x_t, \xi)] = g(x_t) \in \partial f(x_t)$$

where $\{\xi_n\}_{n \geq 1}$ sequence of i.i.d. random variables.

Our goal is to find an ϵ-solution to the above problem, i.e. a point $\bar{x} \in X$, such that: $E[\Psi(\bar{x}) - \Psi^*] \leq \epsilon$. If $\mu \leq 0$, then by complexity theory of convex programming we get an upper bound for the minimum number of iterations needed to find the solution:

$$\mathcal{O}(1)\{\sqrt{\frac{L}{\mu}} \log \frac{L||x_0 - x^*||^2}{\epsilon} + \frac{(M+\sigma)^2}{\mu\epsilon}\}, \text{ if } \mu > 0$$

and:

$$\mathcal{O}(1)\{\sqrt{\frac{L||x_0 - x^*||^2}{\epsilon}} + \frac{\sigma^2}{\epsilon^2}\}, \text{ if } \mu = 0$$

In this sub-section we will introduce the stochastic accelerated gradient descent algorithm that achieves the lower complexity bounds stated above.

Algorithm 15 Stochastic Accelerated Gradient Descent

Require: $N \geq 0$: Number of iterations
  Initialize starting points x_0 and $\bar{x}_0$
  for n in 1, 2, ..., N **do**
    Construct the n-th search point:
      $\underline{x}_n = (1 - q_n)\bar{x}_{n-1} + q_n x_{n-1}$
    Construct the n-th search point used for proximity control:
      $x_n = \arg\min_{x \in X}\{\eta_n[\langle G(x_n, \xi_n), x\rangle + h(x) + \mu V(\underline{x}_n, x)] + V(x_{n-1}, x)\}$
    Construct the n-th point of the output sequence:
      $\bar{x}_n = (1 - a_n)\bar{x}_{n-1} + a_n x_n$
  end for

Initially, we need to examine further function Ψ [6].

Lemma 2.4.4. *Consider $\bar{x}_n$ the n-th point of the output sequence generated by the stochastic accelerated gradient descent and $(\bar{x}_{n-1}, x_n) \in X \times X$. Then:*

$$\Psi(\bar{x}_n) \leq (1 - a_n)\Psi(\bar{x}_{n-1}) + a_n[f(x) + \langle f'(x), x_n - x\rangle + h(x_n)] + \\ + \frac{l}{2}||\bar{x}_n - x||^2 + M||\bar{x}_n - x||$$

for any $x \in X$

The following propositions describe recursive results for the stochastic accelerated gradient descent [6].

Proposition 2.4.3. *Consider the given pair $(x_{n-1}, \bar{x}_{n-1}) \in X \times X$ and the tuple $(\underline{x}_n, x_n, \bar{x}_n)$ generated by the stochastic gradient descent algorithm. If the following conditions hold:*

$$\frac{q_n(1 - a_n)}{a_n(1 - q_n)} = \frac{1}{1 + \mu\gamma_n}$$
$$1 + \mu\gamma_n = La_n\gamma_n$$

then we have:

$$\Psi(\bar{x}_n) \leq (1 - a_n)\Psi(\bar{x}_{n-1}) + a_n l_\Psi(\underline{x}_n, x) + \\ + \frac{a_n}{\gamma_n}[V(x_{n-1}, x) - (1 - \mu\gamma_n)V(x_n, x)] + \Delta_n(x)$$

where $x \in X$ and:

$$l_\Psi(\underline{x}_n, x) = f(\underline{x}_n) + \langle f'(\underline{x}_n), x - \underline{x}_n\rangle + h(x) + \mu V(\underline{x}_n, x)$$
$$\Delta_n(x) = \frac{a_n\gamma_n(M + ||\delta_n||^2)}{2[1 + \mu\gamma_n - La_n\gamma_n]} + a_n\langle\delta_n, x - x^+_{n-1}\rangle$$
$$x^+_{n-1} = \frac{\mu\gamma_n}{1 + \mu\gamma_n}\underline{x}_n + \frac{1}{1 + \mu\gamma_n}x_{n-1}$$

Proposition 2.4.4. *Consider $\{\bar{x}_n\}_{n\geq 1}$ the output sequence produced by the stochastic accelerated gradient descent algorithm. If the sequences $\{a_n\}_{n\geq 1}$ and $\{q_n\}_{n\geq 1}$ are selected, such that the conditions of Proposition 2.4.3 hold, then we have:*

$$\Psi(\bar{x}_n) - \Gamma_k\sum_{n=1}^{k}[\frac{a_n}{\Gamma_n}l_\Psi(\underline{x}_n, x)] \leq \Gamma_k(1 - a_1)\Psi(\bar{x}_1) + \\ + \Gamma_k\sum_{n=1}^{k}\frac{a_n}{\gamma_n\Gamma_n}[V(x_{n-1}, x) - (1 + \mu\gamma_n)V(x_n, x)] + \Gamma_k\sum_{n=1}^{k}\frac{\Delta_n(x)}{\Gamma_n}$$

for any $x \in X$, $k \geq 1$ and:

$$\Gamma_n = \begin{cases} 1, n = 1 \\ (1 - a_n)\Gamma_{n-1}, n \geq 2 \end{cases}$$

Finally, the following theorem establishes some important convergence results of the vanilla stochastic accelerated gradient descent [6].

Theorem 2.4.4. *Consider the sequences $\{a_n\}_{n\geq 1}$, $\{q_n\}_{n\geq 1}$ and $\{\gamma_n\}_{n\geq 1}$, such that $a_1 = 1$ and the conditions of Proposition 2.4.3 hold. Additionally, for the $\{a_n\}_{n\geq 1}$ and $\{\gamma_n\}_{n\geq 1}$ holds:*

$$\frac{a_n}{\gamma_n\Gamma_n} \leq \frac{a_{n-1}(1+\mu\gamma_{n-1})}{\gamma_{n-1}\Gamma_{n-1}}$$

A. If Assumption 3 holds, then:

$$\mathbb{E}[\Psi(\bar{x}_k) - \Psi^*] \leq B_e(k)$$

where:

$$B_e(k) = \frac{\Gamma_k}{\gamma_1}V(x_0, x^*) + \Gamma_k\sum_{n=1}^{k}\frac{a_n\gamma_n(M^2+\sigma^2)}{\Gamma_n(1+\mu\gamma_n - La_n\gamma_n)}$$

B. If Assumption 4 holds, then:

$$Pr\{\Psi(\bar{x}_k) - \Psi^* \geq B_e(k) + \lambda B_p(k)\} \leq \exp\{\frac{\lambda^2}{3}\} + \exp\{-\lambda\}$$

for any $\lambda > 0$ and $k \geq 1$, with:

$$B_p(k) = \sigma\Gamma_k R_X(x^*)(\sum_{n=1}^{k} \frac{a_n^2}{\Gamma_n^2})^{\frac{1}{2}}\Gamma_k \sum_{n=1}^{k} \frac{a_n\gamma_n\sigma^2}{\Gamma_n(1+\mu\gamma_n - La_n\gamma_n)}$$
$$R_X(x^*) = \max_{x\in X} ||x - x^*||$$

C. For compact X, then the condition for $\{a_n\}_{n\geq 1}$ and $\{\gamma_n\}_{n\geq 1}$ becomes:

$$\frac{a_n}{\gamma_n\Gamma_n} \geq \frac{a_{n-1}}{\gamma_{n-1}\Gamma_{n-1}}$$

Both A and B hold, with the modification:

$$B_e(k) = \frac{a_k D_X}{\gamma_k} + \Gamma_k \sum_{n=1}^{k} \frac{a_n\gamma_n(M^2+\sigma^2)}{\Gamma_n(1+\mu\gamma_n - La_n\gamma_n)}$$

General Convex Problems

Initially, we consider problems, where the objective function f is not strongly convex. In this problem setting, it holds: $\mu = 0$ and, thus, the parameters of the stochastic accelerated gradient descent algorithm are modified as: $a_n = q_n$. In this subsection, two stochastic accelerated gradient descent algorithms with different step-size policies are presented.

The first algorithm and its convergence properties derive directly from Theorem 2.4.4 [6].

Proposition 2.4.5. *Consider:*

$$a_n = \frac{2}{n+1} \text{ and } \eta_n = \eta \cdot n, \forall n \geq 1$$

for $\eta \leq \frac{1}{4L}$. If Assumption 3 holds, then:

$$\mathbb{E}[\Psi(\bar{x}_n) - \Psi^*] \leq C_{e,1}(n), \forall n \geq 1$$

with:

$$C_{e,1}(n) = \frac{2v(x_0,x^*)}{\eta N(N+1)} + \frac{4\eta(M^2+\sigma^2)(N+1)}{3}$$

If Assumption 4 also holds, $\forall \lambda > 0$ and $N \geq 1$, we have:

$$Pr\{\Psi(\bar{x}_n) - \Psi^* > C_{e,1}(N) + \lambda C_{p,1}(N)\} \leq \exp\{-\frac{\lambda^2}{3}\} + \exp\{-\lambda\}$$

Using this step-size policy, we can examine the optimal rate of the algorithm's convergence. By having fixed iterations' number N and parameters $\{a_n\}_{n\geq 1}$ and $\{\eta_n\}_{n\geq 1}$ as described in Proposition 2.4.5, where:

$$\eta = \min\{\tfrac{1}{4L}, (\tfrac{3V(x_0,x^*)}{2(M^2+\sigma^2)N(N+1)^2})^{\frac{1}{2}}\}$$

then, the following derive directly from Proposition 2.4.5:

$$C_{e,1}(N) \leq \frac{8LV(x_0,x^*)}{N(N+1)} + \frac{4\sqrt{2(M^2+\sigma^2)V(x_0,x^*)}}{\sqrt{3N}} = C^*_{e,1}(N)$$

If Assumption 3 holds, then, using Proposition 2.4.5, we get the optimal expected rate of convergence of non-strongly convex problems:

$$\mathbb{E}(\Psi(\bar{x}_n) - \Psi^*) \leq C^*_{e,1}(N)$$

It is important to mention that, in order to find the rate of convergence when applying this algorithm, we need to have an estimate of $V(x_0, x^*)$.

One weakness of the above algorithm is that we need to precisely fix the number of iterations N. The next proposed step-size policy overcomes this issue [6].

Proposition 2.4.6. *If X is a compact set and $\{x_n\}_{n\geq 1}$ is the sequence produced by the stochastic accelerated gradient descent method with parameters:*

$$a_n = \tfrac{2}{n+1} \text{ and } \tfrac{1}{\eta_n} = \tfrac{2L}{n} + \eta\sqrt{n},\ \forall n \geq 1$$

with $\eta > 0$. If Assumption 3 holds, then:

$$\mathbb{E}[\Psi(\bar{x}_N) - \Psi^*] \leq C_{e,2}(N),\ N \geq 1$$

where:

$$C_{e,2}(N) = \tfrac{4LD_X}{N(N+1)} + \tfrac{2\eta D_X}{\sqrt{N}} + \tfrac{4\sqrt{2}}{3\eta\sqrt{N}}(M^2+\sigma^2)$$

By setting the learning rate:

$$\eta = \left(\tfrac{2\sqrt{2}(M^2+\sigma^2)}{3D_X}\right)^{\frac{1}{2}}$$

we get the optimal rate of convergence for the algorithm described by Proposition 2.4.6:

$$\mathbb{E}[\Psi(\bar{x}_N) - \Psi^*] \leq C^*_{e,2}$$

where:

$$C^*_{e,2} = \tfrac{4l\bar{V}(x^*)}{N(N+1)} + 4\left(\tfrac{2\sqrt{2}D_X(M^2+\sigma^2)}{3N}\right)^{\frac{1}{2}}$$

Non-Smooth Strongly Convex Problems

In this subsection, a stochastic accelerated gradient descent algorithm with specific step-size policy that achieves optimal rate of convergence in non-smooth and strongly convex optimization problems is presented [6].

Proposition 2.4.7. *Consider the output sequence* $\{\bar{x}_n\}_{n\geq 1}$ *of the stochastic accelerated gradient descent algorithm with parameters:*

$$a_n = \frac{2}{n+1},\ \frac{1}{\eta_n} = \frac{\mu(n-1)}{2} + \frac{2L}{n} \text{ and } q_n = \frac{a_n}{a_n+(1-a_n)(1+\mu\eta_n)}$$

If Assumption 3 holds, then we have:

$$\mathbb{E}[\Psi(\bar{x}_N) - \Psi^*] \leq D_e(N)$$

where:

$$D_e(N) = \frac{4LV(x_0,x^*)}{N(N+1)} + \frac{4(M^2+\sigma^2)}{\mu(N+1)},\ \forall N \geq 1$$

The algorithm proposed by Proposition 2.4.7 is optimal for non-smooth and strongly convex optimization problems and is a significant improvement for smooth and strongly convex problems.

Finally, under Assumption 3 and by applying Markov's inequality to the result of Proposition 2.4.7, we get:

$$Pr\{\Psi(\bar{x}_N) - \Psi^* \geq \lambda D_e(N)\} \leq \frac{1}{\lambda}$$

Thus, in order to find an (ϵ, Λ)-solution, i.e. a $\bar{x} \in X$, such that: $Pr\{\Psi(\bar{x}) - \Psi^* < \epsilon\} \geq 1 - \Lambda$, the number of iterations required has complexity bound:

$$\mathcal{O}\{\frac{1}{\Lambda}(\sqrt{\frac{LV(x_0,x^*)}{\epsilon}} + \frac{M^2+\sigma^2}{\mu\epsilon})\}$$

Smooth and Strongly Convex Problems

In smooth and strongly convex problems, a modification of the vanilla stochastic accelerated gradient descent algorithm is needed to reach optimal rate of convergence. The new algorithm is presented below [6]:

Algorithm 16 Multi-Epoch Stochastic Accelerated Gradient Descent

Require: Point $p_0 \in X$ and bound Δ_0, such that: $\Psi(p_0) - \Psi(x^*) \leq \Delta_0$

for k in 1, 2, ... **do**

1. Run N_k iterations of the vanilla stochastic accelerated gradient descent with:

$x_0 = p_{k-1}$
$a_n = \frac{2}{n}$
$q_n = a_n$
$\eta_n = \eta_k \cdot n$
where:
$N_k = \lceil \max\{4\sqrt{\frac{2L}{\mu}}, \frac{64(M^2+\sigma^2)}{3\mu\Delta_0 2^{-k}}\} \rceil$
$\eta_k = \min\{\frac{1}{4L}, [\frac{3\Delta_0 2^{-(s-1)}}{2\mu(M^2+\sigma^2)N_k(N_k+1)^2}]^{\frac{1}{2}}\}$

2. Set $p_k = \bar{x}_{N_k}$, with $\bar{x}_{N_k}$ being the solution of the vanilla stochastic accelerated gradient descent

end for

The following proposition presents the convergence results of the above multi-epoch variation [6].

Proposition 2.4.8. *Consider $\{p_k\}_{k\geq 1}$ be the sequence produced by the multi-epoch stochastic accelerated gradient descent algorithm. If Assumption 3 holds, then:*

$$\mathbb{E}[\Psi(p_k) - \Psi^*] \leq \Delta_k = \Delta_0 2^{-s}$$

This multi-epoch algorithm finds a solution $\bar{x} \in X$, such that: $\mathbb{E}[\Psi(\bar{x}) - \Psi^*] \leq \epsilon$ for any $\epsilon \in (0, \Delta_0)$ in maximum $\lceil \log(\frac{\Delta_0}{\epsilon}) \rceil$ epochs. Finally, the number of required iterations is bounded by:

$$\mathcal{O}\{\sqrt{\frac{L}{\mu}} \max(1, \log(\tfrac{\Delta_0}{\epsilon}) + \tfrac{M^2+\sigma^2}{\mu\epsilon}\}$$

2.5 Regarding the Implementation and Extensions of Stochastic Gradient Methods

Stochastic optimization methods have been widely applied to train deep learning models due to their ability to perform training in a fast and reliable way. In this section, the most important methods that are used in applied problems are presented.

2.5.1 Stochastic Gradient Descent

The stochastic gradient descent algorithm can be treated as a special case of a stochastic mirror descent algorithm. As mentioned in Chapter 2, stochastic algorithms run by performing sub-sampling during each epoch to produce the epoch's corresponding mini-batch. When a mini-batch S is given in the objective function L, we can calculate the gradient $\nabla_w L(S)$, which is an approximation of the gradient of the entire data-set's objective function $\nabla_w L(X)$.

In order to build the stochastic gradient descent algorithm, we need to modify the generic gradient descent as follows:

Algorithm 17 Stochastic Gradient Descent

Require: N: Number of Iterations, η_n: Epoch's Learning Rate
  for n in 1, 2, ..., N **do**
    Sample s_n data-points
    Update the parameters' values as follows:
      $w^{(n)} \leftarrow w^{(n-1)} - \eta_n \nabla_w J(s_n)$
  end for

The main advantage of stochastic gradient descent is its lower memory requirement and faster performance per each epoch. These characteristics render the stochastic variation of gradient descent more efficient than its deterministic counterpart, even though more epochs are needed to reach a satisfactory minimum.

Convergence in Smooth and Strongly Convex Problems

We will prove the convergence of the stochastic gradient descent algorithm in smooth and strongly convex problem settings. Similarly with the stochastic

mirror descent, we operate in a probability space $(\Omega, \mathcal{F}, \mathbb{P})$ and want to optimize a function $F(x, \xi)$, with ξ being a random variable and $x \in \mathbb{R}^m$. We denote:

$$f(x) = \mathbb{E}_\xi[F(x, \xi)]$$

The stochastic gradient descent algorithm is used to minimize the value of f.

The following theorem [8], the convergence properties of stochastic gradient descent are presented, under the additional assumption that $\nabla_x F$ is bounded.

Theorem 2.5.1. *Suppose that F is differentiable with bounded gradients:*

$$\exists B > 0 : \sup_\xi ||\nabla_x F(x, \xi)||^2 \leq B, \ \forall x \in \mathbb{R}^m$$

and f is smooth and strongly convex. Then:
(i) f has a unique minimum x^*.

(ii) $\forall n \geq 0$: $d_{n+1} \leq (1 - \eta_n \mu) d_n + \eta_n^2 B$

where: $d_n = \mathbb{E}[||x_n - x^*||^2]$, B is the bound of F and μ is the coefficient of strong convexity of f.

(iii) $\forall \epsilon > 0$ $\exists \eta > 0$, such that for $\eta_n = \eta$:

$$\limsup_{n \to \infty} d_{n+1} \leq \epsilon$$

(iv) For the learning rates' sequence $\{\eta_n\}_{n \geq 1}$, we have:

$$\eta_n \to 0 \text{ and } \sum_{n \geq 1} \eta_n = \infty$$

Then, it holds: $d_n \to 0$, i.e. $\lim_{n \to \infty} x_n = x^*$. where the convergence is L^2 convergence of random variables.

Proof. We will prove each result individually.

(i) The existence and uniqueness of the optimum derive directly from the assumptions of smoothness and strongly convexity.

(ii) We have:

$$\begin{aligned} d_{n+1} &= \mathbb{E}[||x_{n+1} - x^*||^2] = \mathbb{E}[||x_n - \eta_n \nabla_x F(x_n, \xi_n) - x^*||^2] = \\ &= \mathbb{E}[||x_n - x^*||^2] + \eta_n^2 \mathbb{E}[||\nabla_x F(x_n, \xi_n)||^2] - 2\eta_n \mathbb{E}[\langle x_n - x^*, \nabla_x F(x_n, \xi_n) \rangle] \end{aligned}$$

First, it holds [1]:

$$\mathbb{E}[\langle x_n - x^*, \nabla_x F(x_n, \xi_n) \rangle] = \mathbb{E}[\langle x_n - x^*, \nabla_x f(x_n) \rangle]$$

[1] For the sigma-algebra $\mathcal{F}_n$ generated by $(x_1, ..., x_n, \xi_1, ..., \xi_{n-1})$, we have that ξ_n is independent of $\mathcal{F}_n$. For any random variables U measured with respect to $\mathcal{F}_n$ and V independent of $\mathcal{F}_n$, it holds: $\mathbb{E}[g(U, V)] = \mathbb{E}[\mathbb{E}[g(U, V)|\mathcal{F}_n]] = \mathbb{E}[\int g(U, V) P_V(du)]$

Then:

$$\mathbb{E}[\langle x_n - x^*, \nabla_x f(x_n)\rangle] \geq \mathbb{E}[f(x_n) - f(x^*) + \frac{\mu}{2}||x_n - x^*||^2] \geq$$
$$\geq \frac{\mu}{2}\mathbb{E}[||x_n - x^*||^2] = \frac{\mu}{2}d_n$$

Finally, we have:

$$d_{n+1} = d_n + \eta_n^2\mathbb{E}[||\nabla_x F(x_n, \xi_n)||^2] - 2\eta_n\mathbb{E}[\langle x_n - x^*, \nabla_x F(x_n, \xi_n)\rangle] \Rightarrow$$
$$\Rightarrow d_{n+1} \leq d_n + \eta_n^2 B - 2\eta_n\frac{\mu}{2}d_n = (1 - \eta_n\mu)d_n + \eta_n^2 B$$

(iii) For a constant learning rate η, the inequality in (ii) becomes:

$$d_{n+1} - \tfrac{\eta}{\mu}B \leq (1 - \eta\mu)(d_n - \tfrac{\eta}{\mu}B)$$

Since the positive part function is increasing, for $\eta < \frac{1}{\mu}$, we have:

$$(d_{n+1} - \tfrac{\eta}{\mu}B)_+ \leq (1 - \eta\mu)(d_n - \tfrac{\eta}{\mu}B)_+$$

and iteratively for $k \geq 1$:

$$(d_{n+k} - \tfrac{\eta}{\mu}B)_+ \leq (1 - \eta\mu)^k(d_n - \tfrac{\eta}{\mu}B)_+$$

By taking $k \to \infty$, we get: $\lim \sum_k (d_k - \frac{\eta}{\mu}B) = 0$, which proves (iii) for $\eta < \frac{1}{\mu}$ and $\eta < \frac{\epsilon\mu}{B}$.

(iv) For non-constant learning rate η_n and arbitrary fixed constant ϵ, we obtain from the inequality of (ii):

$$d_{n+1} - \epsilon \leq (1 - \eta_n\mu)(d_n - \epsilon) + \eta_n(\eta_n B - \mu\epsilon)$$

and for large enough n, such that: $\eta_n(\eta_n B - \mu\epsilon) < 0$, we have:

$$d_{n+1} - \epsilon \leq (1 - \eta_n\mu)(d_n - \epsilon)$$

By performing the same process with positive part function as in (iii), we get:

$$(d_{n+1} - \epsilon)_+ \leq (1 - \eta_n\mu)(d_n - \epsilon)_+$$

and iterating:

$$(d_{n+k} - \epsilon)_+ \leq \prod_{i=n}^{n+k-1}(1 - \eta_i\mu)(d_n - \epsilon)_+$$

The convergence proof is concluded by using Lemma 2.5.1 [8] □

Lemma 2.5.1. *Consider $\mu > 0$ and $\{\eta_n\}_n$ a sequence of positive real numbers, such that: $\eta_n \to 0$ and $\sum_{n\geq 1} \eta_n = \infty$. Then $\forall n \geq 0$, we have:*

$$\lim_{k\to\infty} \prod_{i=n}^{n+k}(1 - \eta_i\mu) = 0$$

Proof. $\forall x \in [0, 1]$, it holds: $\log(1 - x) \leq -x$. Then:

$$0 \leq \prod_{i=n}^{n+k}(1 - \eta_i\mu) = \exp\Big(\sum_{i=n}^{n+k} \log(1 - \eta_i\mu)\Big) \leq \exp\Big(\sum_{i=n}^{n+k}(-\eta_i\mu)\Big) \xrightarrow{k\to\infty} 0$$

□

It is worth noting that, as a stochastic mirror descent algorithm, the stochastic gradient descent is not optimal for smooth and strongly convex problems.

2.5.2 AdaGrad

AdaGrad is a modified version of the stochastic gradient descent algorithm that adjusts its learning rate during each epoch [2]. Denote the gradient of the the cost function of the mini-batch s_n: $g_n = \nabla_w L(s_n)$. Then, we construct the matrix:

$$G_n = \sum_{t=1}^{n} g_t g_t^T$$

and use the above matrix in conjunction with a constant η to produce a variable learning rate. The AdaGrad algorithm has the following form:

Algorithm 18 AdaGrad

Require: N: Number of Iterations, η: Learning Rate's Constant
  for n in 1, 2, ..., N **do**
    Sample s_n data-points
    Construct the matrix:
      $G_n = \sum_{t=1}^{n} g_t g_t^T$
    Update the parameters' values as follows:
      $w^{(n)} \leftarrow w^{(n-1)} - \eta G_n^{-\frac{1}{2}} \nabla_w J(s_n)$
  end for

Because the calculation of $G_n^{-\frac{1}{2}}$ is expensive in higher dimensions, we can modify the update step by using only the diagonal elements of matrix G_n:

$$w^{(n)} \leftarrow w^{(n-1)} - \eta\, diag(G_n)^{-\frac{1}{2}} \nabla_w J(S)$$

2.5.3 RMSProp

RMSProp is a variant of stochastic gradient descent based on running averages. The running average of each epoch can be expressed recursively as:

$$u_n = \gamma u_{n-1} + (1-\gamma)(\nabla_w J(s_{n-1}))^2,\ \gamma \in (0,1)$$

where: $(\nabla_w J(s_{n-1}))^2$ denotes the element-wise square of vector $\nabla_w J(s_{n-1})$. The algorithm itself is presented below:

Algorithm 19 RMSProp

Require: N: Number of Iterations, η: Learning Rate's Constant, γ: Running Average's Constant, ϵ: Small Scalar to prevent Division by 0
  for n in 1, 2, ..., N **do**
    Sample s_n data-points
    Calculate the running average
      $u_n = \gamma u_{n-1} + (1-\gamma)(\nabla_w J(s_n))^2$
    Update the parameters' values as follows:
      $w^{(n)} \leftarrow w^{(n-1)} - \eta \frac{\nabla_w J(s_n)}{\sqrt{|u_n|+\epsilon}}$
  end for

2.5.4 Adam

Adam can be considered as the applied version of stochastic accelerated gradient descent and is inspired by both AdaGrad and RMSProp methods. Adam consists of two exponential moving averages; one for the mean and the other for the variance. A crude version of mean and variance averages respectively are presented below:

$$m_n = \beta_1 m_{n-1} + (1-\beta_1)\nabla_w J(s_n)$$
$$u_n = \beta_2 u_{n-1} + (1-\beta_2)(\nabla_w J(s_n))^2$$

We want to find the unbiased version of theses moving averages. First, we expand the recursive formulas:

$$m_n = (1-\beta_1)\sum_{t=1}^{n} \beta_1^{n-t}\nabla_w J(s_t) \Rightarrow \mathbb{E}[m_n] = (1-\beta_1)\mathbb{E}[\nabla_w J(s_n)]$$
$$u_n = (1-\beta_2)\sum_{t=1}^{n} \beta_2^{n-t}(\nabla_w J(s_t))^2 \Rightarrow \mathbb{E}[u_n] = (1-\beta_2)\mathbb{E}[(\nabla_w J(s_n))^2]$$

Hence, the unbiased versions are given as:

$$\hat{m}_n = \frac{m_n}{1-\beta_1^n}$$
$$\hat{u}_n = \frac{u_n}{1-\beta_2^n}$$

Finally, the algorithm has the following form:

Algorithm 20 Adam

Require: N: Number of Iterations, β_1: Mean Average's Constant, β_2: Variance Average's Constant, w_0: Starting Point, ϵ: Small Scalar to prevent Division by 0

for n in 1, 2, ..., N **do**

Sample s_n data-points

Calculate the mean's moment:

$\hat{m_n} = \frac{m_n}{1-\beta_1^n}$

Calculate the variance's moment:

$\hat{u_n} = \frac{u_n}{1-\beta_2^n}$

Update the parameters' values as follows:

$w^{(n)} \leftarrow w^{(n-1)} - \eta \frac{\hat{m_n}}{\sqrt{|u_n|}+\epsilon}$

end for

2.5.5 AdaMax

AdaMax is a variant of Adam with the infinity norm. The algorithm can be expressed as:

Algorithm 21 AdaMax

Require: N: Number of Iterations, β_1: Mean Average's Constant, β_2: Variance Average's Constant, w_0: Starting Point
Initialize: $m_0 = u_0 = 0$
for n in 1, 2, ..., N **do**
Sample s_n data-points
Calculate the mean's moment:
$m_n = \beta_1 m_{n-1} + (1 - \beta_1)\nabla_w J(s_n)$
Calculate the variance's moment:
$u_n = \max\{\beta_2 u_{n-1}, |\nabla_w J(s_n)|\}$
Update the parameters' values as follows:
$w^{(n)} \leftarrow w^{(n-1)} - \frac{\eta}{1-\beta_1^n}\frac{m_n}{u_{n-1}}$
end for

Chapter 3

Approximation Theory and Applications in Neural Networks

> All exact science is dominated by the idea of approximation.
>
> *Bertrand Russell*

3.1 Some Fundamental Density Results

In this section, some important density theorems from real analysis are presented with their applications in neural networks' learning process.

Specifically, in order to learn a function on a compact interval, a neural network must construct a sequence of continuous functions that converges uniformly to the target function. While the construction is a relatively simple task, uniform convergence is more challenging and requires the use of specific approximation results from real analysis to be shown [2].

3.1.1 Stone-Weierstraß Theorem

From this point onward, for a compact set K in $\mathbb{R}^n$ we just need to consider: $K = I_n = [0,1] \times ... \times [0,1]$. Additionally, the set of continuous real valued functions on K: $C(K)$ can be treated as a metric space [2], with the metric:

$$d(f,g) = \max_{x \in K} |f(x) - g(x)|, \forall f, g \in C(K)$$

Definition 3.1.1. *Consider a metric space (S,d) and a $\mathcal{A} \subset S$. $\mathcal{A}$ is called dense in S, if $\forall g \in S$ exists a sequence of elements in $\mathcal{A}$: $\{f_n\}_{n \geq 1}$, such that:*

$$\lim_{n \to \infty} d(f_n, g) = 0$$

Theorem 3.1.1 (Stone-Weierstraß)**.** *Consider a compact set K in $\mathbb{R}^n$ and $\mathcal{A}$ an algebra of continuous real-valued functions on K that satisfies the following properties:*

(a) $\mathcal{A}$ separates the points of K.

(b) $\mathcal{A}$ contains the constant functions.

Then, $\mathcal{A}$ is a dense subset of $C(K)$.

The following result derives directly from Stone-Weierstraß.

Proposition 3.1.1. *Consider a continuous function $f : [a,b] \times [c,d] \to \mathbb{R}$. Then, $\forall \epsilon > 0$ exist $N \geq 1$ and continuous functions $g_i \in C[a,b]$ and $h_i \in C[c,d]$, $i = 1, .., N$, such that:*

$$\max_{x,y} |f(x,y) - \sum_{i=1}^{N} g_i(x) h_i(y)| < \epsilon$$

The above result can also be extended to a finite product of compact intervals [2].

Application to Neural Networks

Consider a feedforward network with a real-valued input x, an one-dimensional output y and a single hidden layer with N neurons and activation function: $\phi(x) = \cos(x)$. The output is given by the formula:

$$y = b^{(2)} + \sum_{j=1}^{N} W_j^{(2)} \cos\Big(W_j^{(1)} x + b_j^{(1)}\Big)$$

It can be shown that the aforementioned neural network can represent continuous periodic functions [2]. Let f be a periodic and continuous function with period T. The weights connecting the input and hidden layer of the neural network are given as: $W_j^{(1)} = jv$, where: $v = \frac{2\pi}{T}$. By applying the trigonometric formulas, we can get an alternative expression for the output:

$$y = b^{(2)} + \sum_{j=1}^{N} a_j \cos\left(\frac{2\pi j x}{T}\right) - c_j \sin\left(\frac{2\pi j x}{T}\right)$$

where: $a_j = W_j^{(2)} \cos\left(b_j^{(2)}\right)$ and : $c_j = W_j^{(2)} \sin\left(b_j^{(2)}\right)$. The output is dense in the set of continuous periodic functions on $\mathbb{R}$. Hence, by applying Proposition 3.1.1, we get that $\forall \epsilon > 0$, exist $N \geq 1$ and $a_j, b_j, v \in \mathbb{R}$, such that:

$$|a_0 + \sum_{j=1}^{N} a_j \cos(jvx + b_j) - f(x)| < \epsilon$$

Because cosine is an even function, we only need to choose $v > 0$ [2]. Then, by selecting a decreasing sequence $\epsilon = \frac{1}{n}$, we get the sequence v_n that approximates the real sequence $v = \frac{2\pi}{T}$ corresponding to function f.

We can also modify this method to extract hidden sequences from a continuous signal: $z = f(t)$. By taking the value v as a hyperparameter and selecting the values that minimize the cost function, we get the proper frequencies of the signal [2]. This modelling approach can be useful in stock price analysis.

3.1.2 Wiener's Tauberian Theorem

This section examines the methods of learning time-dependent signals [2].

Signals in $L^1(\mathbb{R})$

Theorem 3.1.2. *Consider a signal $f \in L^1(\mathbb{R})$ and the translation function $f_\theta(x) = f(x + \theta)$. The linear span of the family $\{f_\theta; \theta \in \mathbb{R}\}$ is dense in $L^1(\mathbb{R})$, iff the Fourier transform of f never takes the value zero on the real domain.*

The above theorem has applications in neural networks [2]. Consider a feedforward network with one hidden layer and an integrable activation function f, such that: $\hat{f} \neq 0$. Then, for any function $g \in L^1(\mathbb{R})$ and $\epsilon > 0$. For the output of the network $G(x)$ it can be shown by applying Theorem 3.1.2 that there is a proper selection of weights a_j and biases θ_j, such that: $||g - G||_1 < \epsilon$, i.e. there is a one hidden layer network with activation function f that approximately represents g.

Signals in $L^2(\mathbb{R})$

Theorem 3.1.3. *Consider a signal $f \in L^2(\mathbb{R})$ and the translation function $f_\theta(x) = f(x + \theta)$. The linear span of the family $\{f_\theta; \theta \in \mathbb{R}\}$ is dense in $L^2(\mathbb{R})$, iff the zero set of the Fourier transform of f is Lebesgue negligible.*

Similarly with the $L^1(\mathbb{R})$ case, it can be shown that any function $g \in L^2(\mathbb{R})$ can be approximately represented by an one hidden layer neural network with activation function, such that: $\{x : \hat{f}(x) = 0\}$ is Lebesgue negligible.

3.2 Universal Approximators

The performance of neural network models is attributed to their ability to approximate a variety of real-valued functions. In this section, we present the mathematical context of this behaviour. While we do not delve on the learning process that approximates a target function, we prove that there is an appropriate learning process that can perform that approximation. This renders the whole modelling process rational, as we search for a solution that indeed exists.

The concept of the learning process of a neural network is that we have real-valued inputs x and z targets, given by: $z = f(x)$, with f being a deterministic function, such that: $f \in (\mathcal{S}, d)$, where the metric space $(\mathcal{S}, d)$ is defined contextually. Through the learning process, a neural network produces functions g from the output space $\mathcal{U}$, with the assumption that: $\mathcal{U} \subset \mathcal{S}$. In fact, the appropriate choice of activation function guarantees this inclusion assumption.

We can consider a neural network as a universal approximator of the metric space $(\mathcal{S}, d)$, in the sense that the output space $\mathcal{U}$ is d-dense in $\mathcal{S}$ [2]:

$$\forall f \in \mathcal{S} \text{ and } \epsilon > 0, \exists g \in \mathcal{U}, \text{ such that: } d(f - g) < \epsilon$$

Due to this, the space $\mathcal{U}$ is also referred to as approximation space and the whole approximation process as learning.

Learning Continuous Functions

We will examine why a one hidden layer neural network has the ability to learn any continuous function $f \in C(I_n)$. At first, some results of functional analysis that are needed for the proofs of this section [2] are presented.

Lemma 3.2.1. *Consider $\mathcal{U}$ a linear, non-dense subspace of a linear space $\mathcal{X}$ and $x_0 \in \mathcal{X}$, such that:*

$$dist(x_0, \mathcal{U}) \geq \delta$$

for $\delta > 0$. Then, there is a bounded linear functional L on $\mathcal{X}$, such that:

$$\begin{aligned} &i.\ ||L|| \leq 1 \\ &ii.\ L(u) = 0\ \forall u \in \mathcal{U} \Leftrightarrow L_{|\mathcal{U}} = 0 \\ &iii.\ L(x_0) = \delta \end{aligned}$$

We can also restate the above result in terms of dense subspaces.

Lemma 3.2.2. *Consider $\mathcal{U}$ a linear, non-dense subspace of a linear space $\mathcal{X}$. There is a bounded linear functional L on $\mathcal{X}$, such that: $L \neq 0$ and $L_{|\mathcal{U}} = 0$.*

At this point it is important to mention that $C(I_n)$ is a linear subspace of continuous functions on I_n as a normed space with:

$$||f|| = \sup_{x \in I_n} |f(x)|, \forall f \in C(I_n)$$

From this point, we also refer to the space of finite signed Baire measures on I_n as $M(I_n)$ [2]. Baire measures are employed due to their compatibility with compactly supported functions.

Lemma 3.2.3. *Consider $\mathcal{U}$ a linear, non-dense subspace of $C(I_n)$. There is a measure $\mu \in M(I_n)$, such that:*

$$\int_{I_n} h d\mu = 0, \forall h \in \mathcal{U}$$

We are now ready to examine the approximation abilities of specific neural networks for continuous target functions [2]. At first, we consider networks with discriminatory activation functions (see Chapter 2).

Proposition 3.2.1. *Consider a continuous discriminatory function σ. Then, the following finite sum:*

$$G(x) = \sum_{j=1}^{N} a_j \sigma(W_j^T x + \theta_j)$$

is dense in $C(I_n)$, with $w_j \in \mathbb{R}^m, a_j, \theta_j \in \mathbb{R}$.

An intuitive explanation of the above result could be that $\forall f \in C(I_n)$ and $\epsilon > 0$, there is a finite sum $G(x)$ of the form stated above, such that:

$$|G(x) - f(x)| < \epsilon, \forall x \in I_n$$

with $G(x)$ being the output of an one hidden layer neural network with σ activation function. The next result expands on the above concept.

Theorem 3.2.1. *Consider σ an arbitrary continuous sigmoidal function. The following finite sum:*

$$G(x) = \sum_{j=1}^{N} a_j \sigma(W_j^T x + \theta_j)$$

is dense in $C(I_n)$, with $w_j \in \mathbb{R}^m, a_j, \theta_j \in \mathbb{R}$.

Finally, the following result loosens even further the restrictions on the activation function, in order to describe the approximation properties of a bigger family of neural networks. The drawback is that we move away from Σ-class neural networks (networks that approximate a target using only sums) to a $\Sigma\Pi$-class networks (networks that approximate a target using sums of products).

Theorem 3.2.2. *Consider a continuous, non-constant activation function* $\phi : \mathbb{R} \to \mathbb{R}$. *The following sum:*

$$G(x) = \sum_{k=1}^{M} \beta_k \prod_{j=1}^{N_k} \phi(w_{jk}^T x + \theta_{jk})$$

is dense in $C(I_n)$, *with* $w_{jk} \in \mathbb{R}^m, \beta_j, \theta_{jk} \in \mathbb{R}$.

Proof. Consider $\mathcal{U}$ the set containing all sums of products of the form:

$$G(x) = \sum_{k=1}^{M} \beta_k \prod_{j=1}^{N_k} \phi(w_{jk}^T x + \theta_{jk})$$

Since the activation function ϕ is continuous, for any $G, G_1, G_2 \in \mathcal{U}$, it holds:

$$\cdot G_1 + G_2 \in \mathcal{U} \qquad \cdot G_1 G_2 \in \mathcal{U} \qquad \cdot \lambda G \in \mathcal{U}, \forall \lambda \in \mathbb{R}$$

and, thus, $\mathcal{U}$ is an algebra of continuous real functions on I_n. We shall show that the conditions of the Stone-Weierstraß hold:

(i) $\mathcal{U}$ separates points of I_n: Consider $x, y \in I_n$, with $x \neq y$. We show that $\exists G \in \mathcal{U}$, such that: $G(x) \neq G(y)$.

Since ϕ is non-constant, then $\exists a, b \in \mathbb{R}$, with $a \neq b$, such that: $\phi(a) \neq \phi(b)$. By selecting to points x and y from the hyperplanes $\{w^T x + \theta = a\}$ and $\{w^T x + \theta = b\}$, then: $G(u) = \phi(w^T u + \theta)$ separates them:

$$G(x) = \phi(w^T x + \theta) = \phi(a)$$
$$G(y) = \phi(w^T y + \theta) = \phi(b) \neq \phi(a)$$

(ii) $\mathcal{U}$ contains non-zero constants: Consider θ, such that: $\phi(\theta) \neq 0$ and select a vector: $w = (0, ..., 0) \in \mathbb{R}^m$. Then: $G(x) = \phi(w^T x + \theta) = \phi(\theta) \neq 0$ is a non-zero constant. Additionally, by multiplying with any non-zero real number λ, we prove that all non-zero constants are contained in $\mathcal{U}$.

Finally, by applying the Stone-Weierstraß theorem, we get that $\mathcal{U}$ is dense in $C(I_n)$. □

3.3 Exact Learning

Exact learning is a type of learning process, where the network is trained to represent exactly a target and not by tuning its weight values using a numerical optimization algorithm as discussed in Chapter 2.

3.3.1 Learning Finite Support Functions

The simplest case of learning is that of finite support functions. The following result [2] explores the ability of one hidden layer neural networks with Heaviside activation function to represent any finite support function on $\mathbb{R}^m$.

Proposition 3.3.1. *Consider an arbitrary function $g : \mathbb{R}^m \to \mathbb{R}$ and a set $S = \{x_1, ..., x_n\}$ of n distinct points of $\mathbb{R}^m$. There is a sum:*

$$G(x) = \sum_{i=1}^{n} a_i H(w_i^T x + \theta_i)$$

with $w_i \in \mathbb{R}^k, \theta_i, a_i \in \mathbb{R}$, such that:

$$G(x_j) = g(x_j)$$

3.3.2 ReLU Learning

We initially define a ReLU network in the following definition [2].

Definition 3.3.1. *A ReLU-feedforward network with input $x \in \mathbb{R}^n$ and output $y \in \mathbb{R}^m$ is constructed by a sequence of $L-1$ natural numbers $\{l_i\}_{i=1}^{L-1}$ representing the widths of the hidden layers and a set of L affine functions, such that: $A_1 : \mathbb{R}^n \to \mathbb{R}^{l_1}$, $A_L : \mathbb{R}^{L-1} \to \mathbb{R}^m$ and $A_i : \mathbb{R}^{l_{i-1}} \to \mathbb{R}^{l_i}$, $i = 2, ..., L-1$. The output of this neural network is given by the following composition:*

$$f = A_L \circ ReLU \circ A_{L-1} \circ ... \circ A_2 \circ ReLU \circ A_1$$

Representing Maxima

Feedforward networks with ReLU activation functions in their hidden layers and linear activation functions in their output layer have the capability to represent the maximum function $f(x_1, ..., x_n) = \max\{x_1, ..., x_n\}$ [2].

Proposition 3.3.2. *The maximum function $f(x_1, ..., x_n) = \max\{x_1, ..., x_n\}$ can be exactly represented by a ReLU-feedforward network with depth:*

$$L = \begin{cases} 2k, \text{ if } n = 2^k \\ 2(\lfloor \log_2 n \rfloor + 1), \text{ if } n \neq 2^k \end{cases}$$

Swallow and Deep ReLU networks

We examine the difference of swallow and deep neural networks with ReLU activation function [2].

Proposition 3.3.3. *Consider a ReLU-network N^1 with input's dimension d, a single hidden layer of width n and output dimension 1. There exists another ReLU-network N^2 that represents the same function as N^1 on $[0, 1]^d$ with input dimension d but $n + 2$ hidden layers, each of width $d + 2$.*

3.3.3 Kolmogorov-Arnold-Sprecher Theorem

The first theorem is that of Kolmogorov-Arnold [2].

Theorem 3.3.1 (Kolmogorov Arnold)**.** *Any continuous function $f(x_1, ..., x_n)$ defined on I_n, with $n \geq 2$ can be written in the form:*

$$f(x_1, ..., x_n) = \sum_{j=1}^{2n+1} \chi_j (\sum_{i=1}^{n} \psi_{ij}(x_i))$$

where χ_j, ψ_{ij} are one-variable continuous functions and ψ_{ij} are also monotone functions, which are not dependent on f.

The second theorem is a refined version of the above result [2].

Theorem 3.3.2 (Sprecher)**.** *For each integer $n \geq 2$, there exists a real, monotone increasing function $\psi(x)$, with $\psi([0, 1]) = [0, 1]$, dependent on n and having the property that for each pre-assigned number $\delta > 0$, there exists a rational number $0 < \epsilon < \delta$, such that every real continuous function $f(x_1, ..., x_n)$ defined on I_n can be represented as:*

$$f(x_1, ..., x_n) = \sum_{j=1}^{2n+1} \chi(\sum_{i=1}^{n} \lambda^i \psi(x_i + 2(j-1)) + j - 1)$$

where the function χ is real and continuous and λ is a constant independent of f.

The value of the above result is that it ensures that a neural network with 2 hidden layers and activation functions ψ and χ of each layer respectively can exactly represent any continuous function on I_n. The practical problem is that, once again, we only have an existence result; although we know that a solution exists, we can not specify it.

Chapter 4

Neural Network Architectures

> Life is architecture and
> architecture is the mirror of life.
>
> *I. M. Pei*

4.1 Introduction

After the initial success of feedforward neural networks in relatively simple problems, the machine learning community experimented on more complex problem settings to examine if a proper deep learning solution can be found. During the last few years many advanced neural network architectures have been developed to handle different and complex data, such as image or text. These advancements led to the widespread applicability of deep learning methods in a variety of fields, from computer vision and natural language processing to finance.

4.2 Convolutional Neural Networks

Convolutional neural networks are specialized neural networks designed to process grid-structured data. Examples are 1-dimensional data, such as time-series or 2-dimensional images. Additional dimensions could be added to describe more accurately a specific data type, e.g. colour values in image data.

The biological inspiration of convolutional neural networks were the experiments of Hubel and Wiesel on the visual cortex of cats. Since then, they have

been successfully applied to many problem settings, the majority of which involve image data. Despite this, convolutional neural networks can handle any type of spatial or temporal data, due to their grid-like topology.

4.2.1 Basic Architecture Design

The structure of convolutional networks is similar to a feedforward, however its layers are sparsely connected and have a specific spatial organization [1]. Typically, a convolutional network consists of three types of layers: the convolution layer, the pooling layer and the fully connected layer. Both convolution and pooling layers can be interpreted as 3-dimensional grid structures, with application-specific dimensions. In the rest of this subsection, a brief description of each type of layer is given, with a more extensive and mathematical examination conducted afterwards.

A convolution layer performs the convolution operation on its input. The convolution operation requires a kernel, i.e. a 3-dimensional square structure with the same depth with its corresponding layer and smaller spatial dimensions. By placing the kernel at each possible position of a data point, the convolution filters the input's information and produces a feature map.

After a convolution operation, a nonlinear activation function is usually applied to construct the final output of the convolution layer. The most common choice of activation function is the ReLU, due to its speed and ability to avoid the vanishing gradients problem.

A pooling layer follows a convolution layer and is used to compress its output, using a filter of smaller dimensions than the feature map. Pooling does not only reduce the computational cost of the whole network, but also provides a way to solve overfitting problems in convolutional networks.

The final layer of a convolutional network is a common feedforward network. Its input is the flattened output of the final pooling or convolution layer and the choice of its activation function is application-dependent.

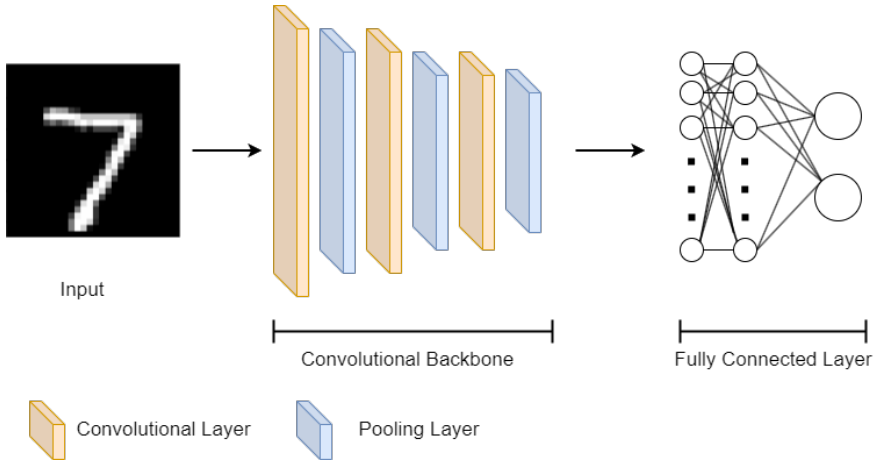


Figure 4.1: Example of a Convolutional Neural Network

4.2.2 Convolution Operation

From a general perspective, a convolution is an operation between two real-valued functions that measures the overlap of its first argument as it is shifted over the second argument [2]. The convolution is typically denoted as:

$$z(t) = (y * w)(t)$$

In terms of neural networks, the convolution's first argument is called input and the second kernel. Both input and kernel are tensors. As previously mentioned, the convolution's output is referred to as feature map.

Due to its general nature, it is necessary to examine the convolution operation under specific structure cases of input signals.

1-Dimensional signals

· Discrete Signals

A 1-dimensional discrete signal can be considered as an infinite sequence of real numbers: $y = \{y_n\}_{n\in\mathbb{Z}}$, with y_n representing the signal's amplitude at time t_n.

In this case, the convolution's kernel is reduced to a compact support sequence of weights: $w = \{w_n\}_{n\in\mathbb{Z}}$. Then, the convolved signal z of y and w, is the infinite sequence: $z = y * w = \{z_n\}_{n\in\mathbb{Z}}$, where:

$$z_n = \sum_{k=-\infty}^{\infty} y_{n+k} w_k$$

· Continuous Signals

A continuous signal is a signal, whose amplitude is a temporal continuous function: $y = y(t)$.

The kernel is a temporal continuous function with compact support: $w = w(t)$. The convolved signal can be obtained as:

$$z = z(t) = (y * w)(t) = \int_{\mathbb{R}} y(u + t)w(u)du$$

In the typical case of convolutional layers with 1-dimensional signals, the input is a compact supported signal: $x = (x_1, ..., x_n)$ and the sliding kernel is: $w = (w_1, ..., w_k)$, with $k < n$. During the convolution operation the kernel slides with respect to the input as described above and the lag of this sliding is called stride.

2-Dimensional Signals

· Discrete Signals

A 2-dimensional discrete signal can be considered as an infinite matrix: $y = [y_{i,j}]_{i,j\in\mathbb{Z}}$.

The kernel is the compact support matrix: $w = [w_{i,j}]_{i,j\in\mathbb{Z}}$ and the convolved signal $z = y * w$ is also an infinite matrix: $[z_{i,j}]_{i,j\in\mathbb{Z}}$, where:

$$z_{i,j} = \sum_{k=-\infty}^{\infty} \sum_{r=-\infty}^{\infty} y_{i+k,j+r} w_{k,r}$$

In the typical case of convolutional layers with 2-dimensional signals, the kernel is placed on the input matrix and, beginning from the top left, moves horizontally and vertically, with the amount of pixels slid per each move called stride, as in the 1-dimensional case.

In the case of 3-dimension signals, each element of the third dimension can be separately convoluted, e.g. each value of red, green and blue in coloured images. However, using variants of the convolution algorithm, one can convolute all three dimensions simultaneously. Finally, there are also variants to handle higher-dimensional inputs.

Equivariance Property

Equivariance refers to the property that the output of a function changes according to a change of its input [2]. More specifically, a function f is equivariant to a function g, if the following relationship holds:

$$f(g(x)) = g(f(x))$$

For the case of the convolution operation, the convolution is equivariant to the function that shifts its input. This ability permits convolutional networks to detect local patterns in data regardless of their positioning.

4.2.3 Pooling

Pooling is an operation that compresses the information of an input by selecting only important local features. As mentioned before, this provides an efficient way to deal with the overfitting problem and renders the computational cost of the whole convolutional network more manageable.

Approximating Continuous Functions

In this subsection, the max-, min- and average-pooling techniques are presented in the context of continuous functions defined on compact sets [2]. For simplicity reasons, only the closed compact set $[a, b] \subset \mathbb{R}$ is considered, however the generalization of the following results to higher dimensions, where f is defined on a compact set $K \subset \mathbb{R}^n$ is straightforward.

· Max-pooling: Consider a continuous function $f : [a, b] \to \mathbb{R}$ and the equidistant partition of $[a, b]$ of size (stride) $\frac{b-a}{n}$:

$$a = x_0 < x_1 < ... < x_{n-1} < x_n = b$$

By defining: $M_i = max_{[x_{i-1},x_i]} f(x)$, a new function that can approximate f can be constructed as:

$$S_n(x) = \sum_{i=1}^{n} M_i I_{[x_{i-1},x_i)}(x)$$

The approximation of f using the S_n function is referred to as max-pooling.

· Min-pooling: By defining the minimum function: $m_i = min_{[x_{i-1},x_i]}$ and using the same partition as above, we can construct the approximating function:

$$s_n = \sum_{i=1}^{n} m_i I_{[x_{i-1},x_i)}(x)$$

The approximation of f using s_n is called min-pooling. Between max- and min-pooling, the following relationship holds:

$$s_n(x) \leq f(x) \leq S_n(x)$$

· Average-pooling: The average function of f on the interval $[x_{i-1}, x_i]$ can be defined as:

$$\mu_i = \frac{1}{x_i - x_{i-1}} \int_{x_{i-1}}^{x_i} f(u)du$$

Average-pooling refers to the approximation of f using the function:

$$A_n(x) = \sum_{i=1}^{n} \mu_i I_{[x_{i-1}, x_i)}(x)$$

After defining the operations of max-, min- and average-pooling, it is essential to provide the theoretical background that all these methods approximate sufficiently f.

Theorem 4.2.1. *Consider a continuous function $f : [a, b] \to \mathbb{R}$. All three function sequences $\{S_n\}_{n\in\mathbb{N}}$, $\{s_n\}_{n\in\mathbb{N}}$ and $\{A_n\}_{n\in\mathbb{N}}$ defined above converge uniformly to f on $[a, b]$, with $n \to \infty$.*

Translation Invariance

One important property of max- and min-pooling operations is their invariance property, which can also be generalized in higher dimensions [2]. By defining the translation operator: T_a for a function f defined on $\mathbb{R}$, such that: $(T_a \circ f)(x) = f(x - a)$ and $\mathcal{P}(f)$ either the max- or min-pooling operation on f, we have the following results:

Proposition 4.2.1. *Consider a continuous function $f : \mathbb{R} \to \mathbb{R}$. For any small enough value a, there exists partition of $\mathbb{R}$, such that:*

$$\mathcal{P}(T_a \circ f) = \mathcal{P}(f)$$

4.2.4 Bayesian Interpretation of Convolutional Networks

Types of Prior Distributions

A prior distribution is a way to incorporate beliefs before getting the data by being a probability distribution on a model's parameters [3]. There are two main categories of prior distributions; weak and strong. Weak priors are probability distributions with high entropy, permitting the model's parameters to change greatly based on the data, whereas strong priors are distributions with low entropy, heavily restricting changes of parameters' values. Additionally, an infinitely strong prior is a prior distribution that nullifies some of the parameters and restricts their use, even if data support their existence in the model.

Bayesian Interpretation

Convolutional networks could be treated as feedforward networks with an infinitely strong prior over their weights. Using this prior, we get a feedforward with sparse weight matrices of each layer, where the weights of each hidden layer are shared with its next one, albeit shifted. The mechanisms that create said prior are the convolution and pooling operations.

Convolution placed an infinitely strong prior over the parameters of a layer, encoding the belief that the layer should learn only local interactions and patterns and this learning is equivariant to translation. On the other hand, pooling places the infinitely strong prior that each unit is invariant to small translations and thus, its information can be compressed.

The bayesian connection between convolutional and feedforward networks helps clarify some specific details about the convolutional architecture. Firstly, being essentially prior distributions, both convolution and pooling can cause underfitting, in case their assumptions are inaccurate. Secondly, by having the prior of equivariance of translation, convolutional networks should only be compared to other convolutional architectures, because permutation invariant models without convolution must learn the concept of topology during training.

4.3 Recurrent Neural Networks

The majority of machine learning techniques are designed to handle multidimensional data, independent from one another, which restricts their ability to handle sequential data, such as time series or text. On the other hand, in order to construct an architecture that can process sequential data, we have to keep in mind that the temporal order of data, often referred to as time-stamp, has to be preserved inside the model and that the handling of all inputs is done in a similar manner. Simultaneously, the model must have a fixed number of parameters, but also have the ability to handle data dynamically.

Recurrent neural networks exploit the idea of having a variable number of layer to process data. In particular, by having a layer corresponding to a single input, the model creates a direct connection between its layers and time-stamps, enabling interactions between data located in different position of the sequence. Finally, by using the same parameters per each layer, the model contains a fixed number of parameters and handles all data in a similar way.

It should be noted that sequential data could be considered as 1-dimensional grid structures, and thus, can be handled by convolutional networks, because convolution enables spatial or, in this case, temporal weight sharing. However, this approach is swallow and can not permit dynamic handling of sequences.

4.3.1 Basic Architectures

Computational Graphs & Dynamical Systems

A computational graph is a directed graph, whose nodes express mathematical computations. Before we can portray the structure of recurrent networks as a computational graph, we have to first examine its depiction as a dynamical system. A dynamical system depicts the evolution of a system defined on a state space through time and is denoted as [2]:

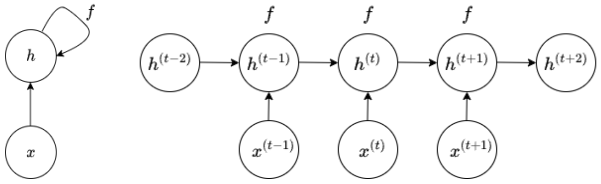
$$s^{(t)} = f(s^{(t-1)}; \theta)$$

with: $s^{(t)}$ being the state of the system at time t. We can adopt a similar notation for the evolution of the hidden layers of a recurrent network:

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta)$$

where we consider data point $x^{(t)}$ as an external signal to the system. Using the above notation, it is clear that when the recurrent networks performs a prediction based on previous values of the sequence, uses the hidden layer $h^{(t)}$ as an indicator of the system's evolution.

There are two possible ways to depict a recurrent network. The first one is to present it as a circuit-like structure (Figure 4.2a), which is based on the approach using dynamical systems. On the other hand, we could imagine a recurrent network as an unfolded computational graph (Figure 4.2b), with the same length as that of the sequence. The computational graph approach breaks down the structure of the dynamical system and depicts the evolution of the network for a specific sequence, rather than its general form.



(a) Circuit Diagram (b) Unfolded Computational Graph

Figure 4.2: Possible depictions of a Recurrent Neural Network

Design Patterns

· Simple Recurrent Network

A simple recurrent network maps the input sequence to an output sequence with an one-to-one relationship and its hidden units are recurrently connected. The unfolded computational graph of this type of network is presented below:

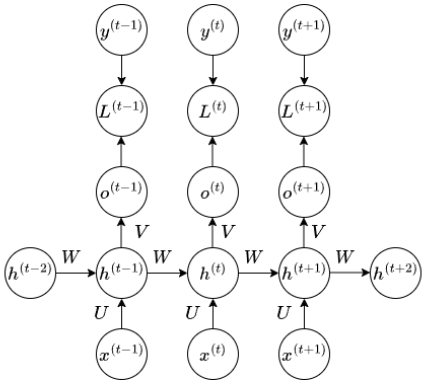


Figure 4.3: Simple Recurrent Neural Network

In order to present a complete description of the above model, we also have

to define its forward propagation equations. We will make the additional assumptions that the target is a discrete variable and the network utilizes the hyperbolic tangent as its activation function. By considering discrete variables, we can the network's output o as the vector containing the un-normalized log-probabilities of each target's value. Then, we apply softmax to the network's output, in order to obtain the normalized probabilities. After initializing the system's prime state $h^{(0)}$, we get the following system of update equations:

$$
\begin{aligned}
a^{(t)} &= b + Wh^{(t-1)} + Ux^{(t)} \\
h^{(t)} &= tanh(a^{(t)}) \\
o^{(t)} &= c + Vh^{(t)} \\
\hat{y}^{(t)} &= softmax(o^{(t)})
\end{aligned}
$$

where b and c represent the biases and W and V the weight matrices and $t \in \{1, ..., T\}$, with T being the length of the sequence. Finally, the total loss is defined as the sum of losses corresponding to each member of the input sequence:

$$L^{(T)} = \sum_{t=1}^{T} L^{(t)}$$

Typically, for discrete targets, the negative log-likelihood is considered as an appropriate loss criterion. Using the above update equations and the loss function, we can proceed to train the model. The actual training is conducting using a modification of the back-propagation mechanism; the back-propagation through time. Although this process will be presented in the next section, it is important to mention that in architectures with hidden-to-hidden connections it renders training expensive. This inspired many modifications of the simple recurrent network's architecture that boost training efficiency.

· Recurrent Network with Output Recurrence

By having recurrent connections between its output and the next hidden unit, the network becomes less powerful, because, then, we anticipate that the output summarizes all past information. This is not usually the case, as network's outputs are designed solely to approximate target variable's values. However, the advantage of this architecture is that, for a choice of loss function that compares each individual prediction with the corresponding target's value, training can be parallelized and, thus, far more efficient from a computational perspective.

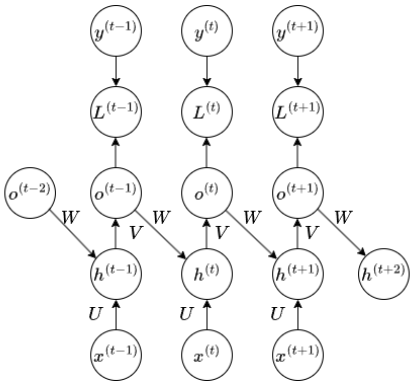


Figure 4.4: Recurrent Network with Output Recurrence

Under the assumptions of the previous architecture. the forward propagation equations are modified as follows:

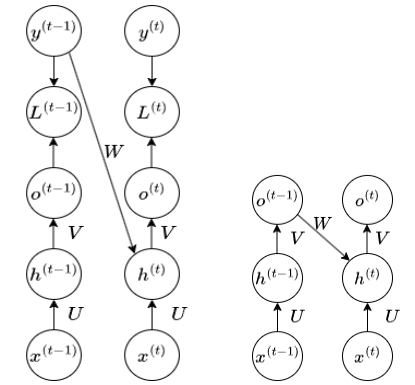
$$
\begin{aligned}
a^{(t)} &= b + Wo^{(t-1)} + Ux^{(t)} \\
h^{(t)} &= tanh(a^{(t)}) \\
o^{(t)} &= c + Vh^{(t)} \\
\hat{y}^{(t)} &= softmax(o^{(t)})
\end{aligned}
$$

· Teacher Forcing

Teacher forcing is a technique inspired by the maximum likelihood criterion, where in each step the target value of the previous step is used during training. We can define the conditional maximum likelihood criterion as:

$$\log p(y^{(1)}, y^{(2)}|x^{(1)}, x^{(2)}) = \log p(y^{(2)}|y^{(1)}, x^{(1)}, x^{(2)}) + \log p(y^{(1)}|x^{(1)}, x^{(2)})$$

To incorporate the aforementioned idea in the model's architecture, instead of using hidden-to-hidden connections, we have to connect recurrently the target value of a time-stamp with the next hidden unit.



(a) Training Graph (b) Testing Graph

Figure 4.5: Recurrent Network with Teacher Forcing

Then, the forward propagation equations become:

$$
\begin{aligned}
a^{(t)} &= b + Wy^{(t-1)} + Ux^{(t)} \\
h^{(t)} &= tanh(a^{(t)}) \\
o^{(t)} &= c + Vh^{(t)} \\
\hat{y}^{(t)} &= softmax(o^{(t)})
\end{aligned}
$$

The main advantage of teacher forcing is that we can avoid back-propagation through time, in condition that there are not hidden-to-hidden connections. Hence, training time becomes much faster.

· Single-Output Recurrent Networks

Recurrent networks with single outputs are mainly encoders, i.e. networks that encode the entire sequence in a feature vector. The feature vector is then used as an fed to other networks, such as a feedforward classifier or a decoder network for further processing.

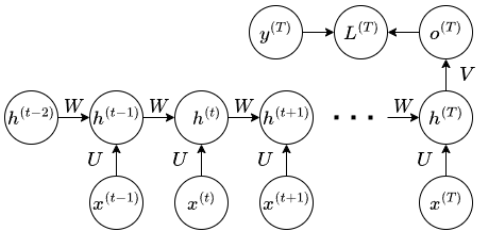


Figure 4.6: Recurrent Network with Single Output

And the corresponding forward equations:

$$
\begin{aligned}
a^{(t)} &= b + Wh^{(t-1)} + Ux^{(t)} \\
h^{(t)} &= tanh(a^{(t)}) \\
o^{(T)} &= c + Vh^{(T)} \\
\hat{y}^{(T)} &= softmax(o^{(T)})
\end{aligned}
$$

4.3.2 Training Procedure & Common Challenges

Back-Propagation Through Time

We will demonstrate the back-propagation through time algorithm for the simple recurrent neural network architecture defined in the previous section. We also make the assumptions that in order to obtain the final output $\hat{y}^{(t)}$, the softmax activation function is used and that as loss function we utilize the negative log-likelihood.

The back-propagation algorithm for recurrent networks can be described as follows:

1. We perform a forward pass using the forward equations and obtain the errors at each time stamp using the negative log-likelihood loss function.

2. For every time-stamp, we treat the parameters (weight matrices and bias vectors) as time-dependent, even though the parameters of different layers in time are shared, and compute the gradients.

3. We calculate the gradients of all time-stamps by summing up the shared parameter gradients of each time-stamp.

Now, we proceed to calculate the parameter gradients. The computational graph of a recurrent network contains the parameters W, U, V, b and c, as well as $x^{(t)}$, $h^{(t)}$, $o^{(t)}$ and $L^{(t)}$ types of nodes. Before we begin calculating the parameter gradients, we have to calculate the gradients of each type of node. At first, we need to calculate the gradient of loss:

$$\frac{\partial L}{\partial L^{(t)}} = 1$$

Then, the output's gradient is given as:

$$(\nabla_{o^{(t)}} L_i^{)} = \frac{\partial L}{\partial o^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - I(y^{(t)}, i)$$

In order to obtain the hidden unit node's gradient, we must take two cases. On the one hand, at the final time-stamp T, the result of $h^{(T)}$ is fed only to the output node $o^{(T)}$. Thus:

$$\nabla_{h^{(T)}} L = V^T \cdot \nabla_{o^{(T)}} L$$

On the other hand, to compute the gradient of hidden units for every other time-stamp $t \in \{1, ..., T-1\}$, we have to take account that their output is fed to both the output node $o^{(t)}$ and the next hidden unit $h^{(t+1)}$. Hence, we need to compute these gradients recursively:

$$\begin{aligned}\nabla_{h^{(t)}} L &= [\frac{\partial h^{(t+1)}}{\partial h^{(t)}}]^T \cdot \nabla_{h^{(t+1)}} L + [\frac{\partial o^{(t)}}{\partial h^{(t)}}]^T \cdot \nabla_{o^{(t)}} L \\ &= W^T diag[1 - (h^{(t+1)})^2] \cdot \nabla_{h^{(t+1)}} L + V^T \cdot \nabla_{o^{(t)}} L\end{aligned}$$

Having the gradients of all necessary nodes, we can proceed to calculate the actual parameters' gradients:

$$\begin{aligned}\nabla_c L &= \sum_{t=1}^{T} [\frac{\partial o^{(t)}}{\partial c}]^T \nabla_{o^{(t)}} L = \sum_{t=1}^{T} \nabla_{o^{(t)}} L \\ \nabla_b L &= \sum_{t=1}^{T} [\frac{\partial h^{(t)}}{\partial b}]^T \nabla_{h^{(t)}} L = \sum_{t=1}^{T} diag[1 - (h^{(t)})^2] \cdot \nabla_{h^{(t)}} L \\ \nabla_V L &= \sum_{t=1}^{T} \sum_i [\frac{\partial L}{\partial o_i^{(t)}}] \cdot \nabla_{V^t} o_i^{(t)} = \sum_{t=1}^{T} (\nabla_{o^{(t)}} L)(h^{(t)})^T \\ \nabla_W L &= \sum_{t=1}^{T} \sum_i [\frac{\partial L}{\partial h_i^{(t)}}] \cdot \nabla_{W^t} h_i^{(t)} = \sum_{t=1}^{T} diag[1 - (h^{(t)})^2](\nabla_{h^{(t)}} L)(h^{(t-1)})^T \\ \nabla_U L &= \sum_{t=1}^{T} \sum_i [\frac{\partial L}{\partial h_i^{(t)}}] \cdot \nabla_{U^t} h_i^{(t)} = \sum_{t=1}^{T} diag[1 - (h^{(t)})^2](\nabla_{h^{(t)}} L)(x^{(t)})^T\end{aligned}$$

· Truncated Back-propagation Through Time

In case of long input sequences, many computational problems arise, especially regarding the convergence rate and the memory-usage of the back-propagation algorithm. A simple solution would be to implement a similar thought with the stochastic optimization algorithms; after the forward pass, execute the back-propagation recursion for a subset of the original sequence. The inputs of this new sub-sequence must, however, be ordered according to their respective time-stamp in the initial sequence.

Training Challenges

The main problems during training that arise in recurrent neural networks originate from long-term dependencies in the model itself [3]. More precisely, the weights that correspond to connections between layers of distant time-stamps are significantly smaller than the ones given to small-term connections. From a mathematical perspective, recursive networks can be treated as the composition of the same function over different time-stamps. The basic recursive updating equation, without considering the external input and the non-linear activation function, can be defined using matrix multiplication:

$$h^{(t)} = W^T h^{(t-1)}$$

We can also get an alternative form of the above equation by applying the substitution method:

$$h^{(t)} = (W^t)^T h^{(0)}$$

where: W^t is the matrix product of t consecutive multiplications. Under the assumption that this matrix can be eigen-decomposed, we get:

$$W = Q \Lambda Q^T$$

with Q being an orthogonal matrix and Λ the eigenvalue diagonal matrix. Then, the recursive equation that describes a recurrent network can be expressed as:

$$h^{(t)} = Q^T \Lambda^t Q \cdot h^{(0)}$$

The problem of vanishing and explosive gradients can be mathematically stated using the following Lemma [1]:

Lemma 4.3.1. *Let W be a square matrix, the magnitude of whose largest eigenvalue is λ. Then, the entries of W^t tend to 0 with increasing values of t, if $\lambda < 1$. If $\lambda > 1$, the entries of W^t diverge to large values.*

Early Strategies

Early solutions to the problem of long-term dependencies were based on the idea to organize the model to handle differently long-term and short-term relationships. By having parts that operate on different time-scales, the model can direct information from the past to the present far more efficiently. In this section, some solutions found in [3] are presented.

· Skip Connections Through Time

The idea behind skip connections is to connect directly the distant past with the present through the introduction of connection delay. In a vanilla recurrent network, the hidden unit corresponding to the time-stamp t is directly connected with the next hidden unit at $t+1$, whereas using skip connections with delay d, the t hidden unit is connected to the hidden unit at $t+d$.

The introduction of delay helps significantly with vanishing gradients, because gradients diminish more smoothly. However, the problem of exploding gradients is not affected by skip connections.

· Leaky Units

Leaky units are based on the concept of running averages [3]. A running average $\mu^{(t)}$ of a value $x^{(t)}$ is given by the recursive relationship:

$$\mu^{(t)} \leftarrow \alpha\mu^{(t-1)} + (1-\alpha)x^{(t)}$$

with $\alpha \in [0,1]$ being the self-connection parameter. In recurrent networks, we can add such self-connections to hidden units, where the derivative is close to 1 and, therefore, potentially dangerous. For small values of α, past information fades fast, whereas for large values of α it plays a vital role in the present's output.

· Removing Connections

Another approach to organizing a recurrent network to operate on multiply time-scales is to manually re-organize its connections. By replacing length-one connections with connections of longer length, the network is forced to transfer past information to the present more effectively.

4.3.3 Advanced Recurrent Architectures

Deep Recurrent Neural Networks

The existence of hidden units in a recurrent architecture enables the model to learn non-linear relationships. In certain applications, where the complexity of the problem is increased, we want to increase non-linearity within the model. This can be achieved by stacking hidden layers on top of each other and, thus, adding more depth to the network.

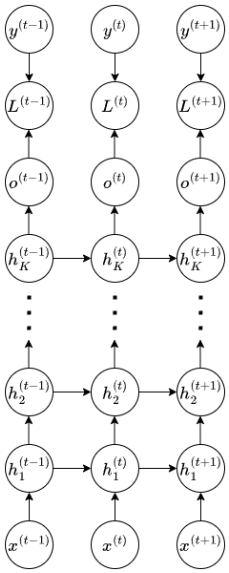


Figure 4.7: Deep Recurrent Network

As a dynamical system, the state of the k-th hidden unit with an activation function ϕ is given by:

$$h_k^{(t)} = \phi(U_k h_{k-1}^{(t)} + W_k h_k^{(t-1)} + b_k)$$

with U_k, W_k and b_k being layer-specific parameters. Finally, the output of the layer of each time-stamp is given by:

$$o^{(t)} = V h_K^{(t)} + c$$

Bidirectional Recurrent Neural Networks

The majority of recurrent architectures are based on the concept that past values of the input sequence influence the present. However, sometimes it is more reasonable to assume that the input of a specific time-stamp is influenced by the entire sequence, e.g. in applications in natural language processing. Bidirectional recurrent networks manage to combine future and past information by consisting of two recurrent networks; one that moves forward and one that moves backwards in time.

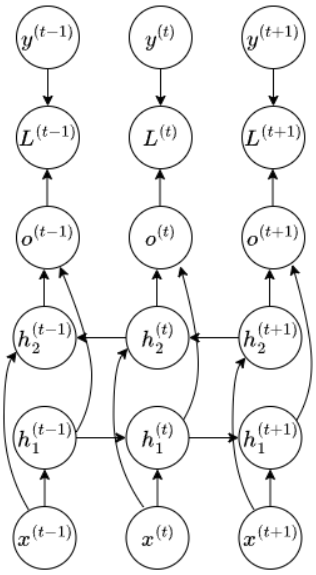


Figure 4.8: Bidirectional Recurrent Network

In the above computational graph, h_1 are the hidden units that transfer information from the past and h_2 transfer information from the future. Using an activation function ϕ, the state of those hidden units is updated as:

$$h_1^{(t)} = \phi(U_1 x^{(t)} + W_1 h_1^{(t-1)} + b_1)$$
$$h_2^{(t)} = \phi(U_2 x^{(t)} + W_2 h_2^{(t+1)} + b_2)$$

And the output is given as:

$$o^{(t)} = V_1 h_1^{(t)} + V_2 h_2^{(t)} + c$$

Recursive Neural Networks

Recursive neural networks are a specific category of recurrent networks that have a deep-tree structure, which gives them the ability to handle data structures as their input [3]. This makes them useful in applications in natural language processing and computer vision.

Long Short-Term Memory

Long short-term memory (LSTM) and gated recurrent units (GRU) networks expand on the idea of leaky units with enabling the self-connection weights to change their value at each time-stamp. However, instead of accumulating information over time, these networks have also a mechanism to decide when to forget past information and, thus, clear their memory. LSTM and GRU networks belong to a family of networks that are called gated recurrent neural networks.

LSTM networks introduce self-loops to determine dynamically whether long-past information should be retained. LSTM introduces a structure called memory cell that can be treated as a special hidden unit. Each cell comprises of three gates; the input gate that reads data into the cell, the output gate, which returns the output of the cell and the forget gate that resets the content of the cell.

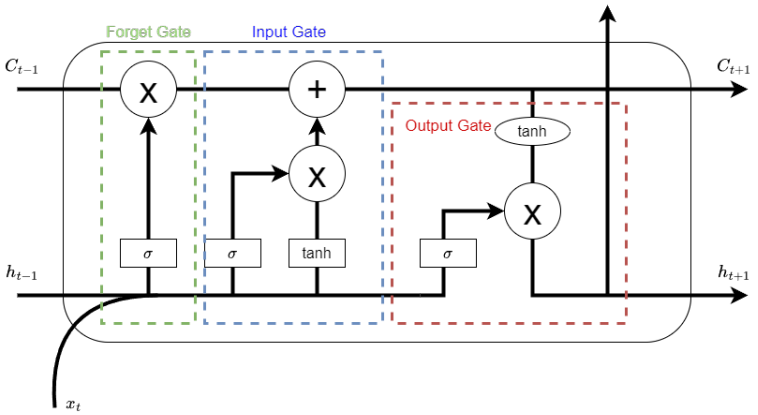


Figure 4.9: LSTM Cell

The above graph can be used as a road-map to perform the computations at each time stamp. By considering multivariate inputs: $\{X_t\}_{t=1}^{T}$, with $X_t \in \mathbb{R}^{n \times d}, \forall t$, the input, output and forget gate are updated as:

$$I_t = \sigma(X_t U_i + H^{(t-1)} W_i + b_i)$$
$$O_t = \sigma(X_t U_o + H^{(t-1)} W_o + b_o)$$
$$F_t = \sigma(X_t U_f + H^{(t-1)} W_f + b_f)$$

Next, we need to introduce a candidate memory cell that determines how much of the new information will be added to the memory:

$$\bar{C}_t = tanh(X_t U_c + H^{(t-1)} W_c + b_c)$$

Then, the memory cell decides how much past memory it will retain and how much new information it will introduce:

$$C_t = F_t \odot C_{t-1} + I_t \odot \bar{C}_t$$

Finally, the hidden unit value is given as:

$$H^{(t)} = O_t \odot tanh(C_t)$$

Gated Recurrent Units

Like LSTM, GRU networks have gated hidden units that determine how much past memory to retain and how much new information to introduce. In a GRU, we have two types of gates; the reset and the update gate.

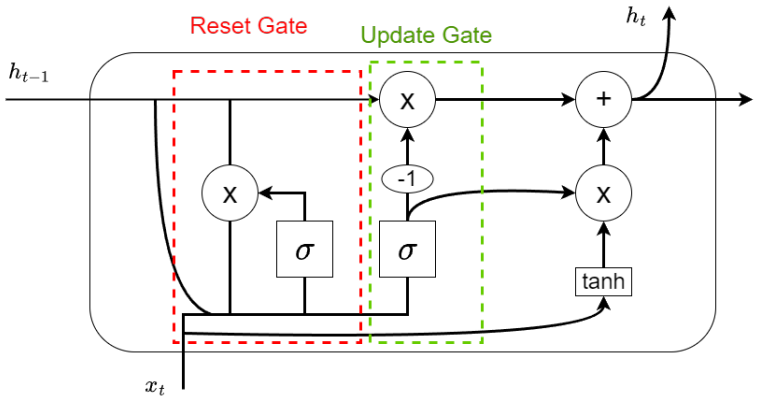


Figure 4.10: GRU Cell

The computations performed by reset and update gates respectfully are:

$$R_t = \sigma(X_t U_r + H^{(t-1)} W_r + b_r)$$
$$Z_t = \sigma(X_t U_z + H^{(t-1)} W_z + b_z)$$

Then, the candidate hidden state is given as:

$$\bar{H}_t = tanh(X_t U_h + (R_t \odot H_{t-1}) W_h + b_h)$$

Finally, the actual output of the hidden unit is calculated as:

$$H - t = Z_t \odot H_{t-1} + (1 - Z_t) \odot \bar{H}_t)$$

4.4 Stochastic Neural Networks I: Boltzmann Machines

All architectures we have seen until now can be treated as input-output non-linear mappings, where by minimizing a loss function, the networks maps the input set into the output set and this whole process can be represented using a directed computational graph. On the other hand, stochastic neural networks learn probabilistic states from the input set and, hence, are often used in unsupervised learning [1]. Boltzmann machines are probabilistic models that model the joint distribution of observed and latent variables. Due to the fact that these models learn probabilistic relationships, they can be represented as an unidirectional computational graph.

4.4.1 Stochastic Neurons

Before exploring Boltzmann and restricted Boltzmann machines, we need to define the stochastic neuron. A stochastic neuron [2] is a computational structure that receives a sequence $x = \{x_n\}_{n=1}^{N}$ as input and outputs the bivariate random variable $Y \in \{0, 1\}$, with:

$$P(Y = 1|x) = \sigma(w^T x + b)$$
$$P(Y = 0|x) = -\sigma(w^T x + b) = \sigma(-w^T x - b)$$

with σ the logistic function.

Training Stochastic Neurons

During training of a stochastic neuron, we utilize the maximum likelihood. We need to make the transformation $U = 2Y - 1$. Then, we have:

$$U = \begin{cases} -1, & Y = 0 \\ 1, & Y = 1 \end{cases}$$

By making this transformation, we get the same distribution of U for both of its values:

$$P(U = 1) = P(U = -1) = \sigma[(w^T \underset{\sim}{x} + b)U]$$

In an applied setting, we have data in the form of the following sequence: $\{(x_n, y_n)\}_{n=1}^{N}$. By creating a sequence: $\{u_n\}_{n=1}^{N}$, with: $u_n = 2y_n - 1$, we get a

new data sequence: $\{(x_n, u_n)\}_{n=1}^N$. Then, we can find the optimal values of the parameters w and b as:

$$\begin{aligned}\theta^* = (w^*, b^*) &= argmax_{w,b} \prod_{n=1}^{N} \sigma[(w^T x_n + b)u_n] \\ &= argmax_{w,b} \ln \prod_{n=1}^{N} \sigma[(w^T x_n + b)u_n] \\ &= argmax_{w,b} \frac{1}{n} \sum_{n=1}^{N} \ln \sigma[(w^T x_n + b)u_n]\end{aligned}$$

We can define the target function that we want to maximize:

$$C(w, b) = \frac{1}{n} \sum_{n=1}^{N} \ln \sigma[(w^T x_n + b)u_n]$$

And optimize it using a gradient ascent scheme, by updating the parameters according to the rule:

$$\begin{aligned}w^{(k)} &= w^{(k-1)} + \eta \nabla_w C(w, b) \\ b^{(k)} &= b^{(k-1)} + \eta \nabla_b C(w, b)\end{aligned}$$

with:

$$\begin{aligned}\frac{\partial C}{\partial W_k} &= \frac{1}{n} \sum_{n=1}^{N} x_n^k u_n \sigma[-(w^T x_n + b)u_n] \\ \frac{\partial C}{\partial b} &= \frac{1}{n} \sum_{n=1}^{N} u_n \sigma[-(w^T x_n + b)u_n]\end{aligned}$$

Simulated Annealing Perspective

From a simulated annealing viewpoint, we can show that a stochastic neuron converges to a perceptron [2]. To prove this, we first have to modify the output's distribution of a stochastic neuron as:

$$\begin{aligned}P(Y = 1|x) &= \sigma_c(w^T x + b) \\ P(Y = 0|x) &= \sigma_c(-w^T x - b)\end{aligned}$$

where: $\sigma_c(x) = \sigma(cx)$, with $c > 0$ being the inverse of temperature, meaning that by increasing the value of c, we decrease the temperature. The target function is modified as well as:

$$C(w,b) = \frac{1}{n}\sum_{n=1}^{N} \ln \sigma_c[(w^T x_n + b)u_n]$$

For $c = 0$, the two possible states, i.e. $Y = 1$ and $Y = 0$ are equiprobable. As $c \to \infty$, we know that σ_c tend almost everywhere to the Heaviside step function [2], thus:

$$P(Y = 1|x) = H(w^T x + b)$$
$$P(Y = 0|x) = 1 - H(w^T x + b)$$

Hence, by decreasing the temperature, we obtain a deterministic perceptron.

4.4.2 Boltzmann Distribution

Boltzmann distribution models the thermodynamic equilibrium of a particle system at a given temperature [2]. In a thermodynamic system with N possible states, we define the random variable of the system's state x that can take the values: $\{x_1, ..., x_N\}$, with corresponding probabilities: $p_n = P(x = x_n), \forall n \in \{1, ..., N\}$. Finally, each state x_n corresponds to a specific energy level of the system $E_n > 0$.

We know that the state of the system changes because of particle interactions and, using the second law of thermodynamics, this change should increase the entropy of the system. Finally, this particle interaction happens in a certain temperature T, proportional to the average system energy: $T \sim \sum_{n=1}^{N} p_n E_n$.

In order to find the distribution $\{p_n\}_{n=1}^{N}$, we need to find the distribution that maximizes the system's entropy in the given temperature T. Therefore, we need to solve the maximization problem:

$$\max_p H(p) = -\sum_{n=1}^{N} p_n \ln p_n$$
$$\text{subject to:} \quad \sum_{n=1}^{N} p_n = 1$$
$$\sum_{n=1}^{N} p_n E_n = k > 0$$

Using Lagrange multipliers, the solution of the problem is:

$$p_n = e^{-\frac{E_n}{T}}$$

4.4.3 Boltzmann Machines

A Boltzmann machine is represented through a state vector $x = (x_1, ..., x_N)$, where $u = (u_1, .., u_d)$ are visible states and $h = (h_1, .., h_m)$ are hidden states. It holds: $x = (u, h)$. The formal definition of a Boltzmann machine [2] is presented below:

Definition 4.4.1. *A Boltzmann machine is a set of n stochastic neurons that form a network with symmetric weights, such that:* $w_{nn} = 0$. *The state of each neuron is modified as:*

$$x_n = \begin{cases} 1, & \textit{with probability } p_n \\ 0, & \textit{with probability } 1 - p_n \end{cases}$$

with:

$$p_n = P(x_n | x_{-n}) = \sigma(\tfrac{\Delta E_n}{T})$$

where: $\Delta E_n = \sum_{j;j \neq i} w_{nj} x_j + b_n$ *is the energy gap of the n-th state and* T *the temperature.*

The energy of a specific state vector is computed as:

$$E(x) = E[(u, h)] = -\sum_n b_n x_n - \sum_{i,j:i<j} w_{ij} x_i x_j$$

The conditional probability of each state p_n originates from the unconditional probability of the state vector: $P(x)$, which can be calculated as [1]:

$$P(x) = \tfrac{1}{Z(x)} \exp\{-E(x)\}$$

with Z being the partition function. The main problem in Boltzmann machines is that the partition function cannot be calculated efficiently and, thus, in an applied setting, we can never find the unconditional probability.

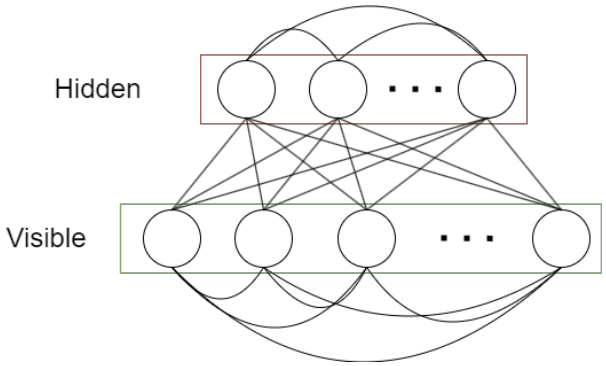


Figure 4.11: Boltzmann Machine's Computational Graph

Data Generation with Boltzmann Machines

Boltzmann machines generate data using Markov Chain Monte Carlo (MCMC) sampling. It samples iteratively from the conditional distribution of each state p_n, until we reach thermal equilibrium [1], a point where the probability distributions have converged. After reaching thermal equilibrium, the generated data derive from the model captured by the Boltzmann machine.

Training Boltzmann Machines

By being distribution approximators, Boltzmann machines generate p distributions to approximate a given distribution q. During training, we need to choose the optimal distribution p^* that minimizes the Kullback-Leibler divergence and, thus, solve the optimization problem [2]:

$$p^* = \arg\min_p \sum_{x \in X} q(x) \ln \frac{q(x)}{p(x;w,b)}$$

By observing that the entropy $H(q)$ is independent of p, we can modify the above optimization problem as:

$$p^* = \arg\max \sum_{x \in X} q(x) \ln p(x;w,b)$$

We define the objective function:

$$C(w,b) = \sum_{x \in X} q(x) \ln p(x;w,b)$$

that we want to maximize. By performing various computations [2] (for a more detailed presentation of said computations, refer to Appendix _), we get the results:

$$\frac{\partial}{\partial w_{ij}} C(w,b) = \mathbb{E}^q[x_i x_j] - \mathbb{E}^p[x_i x_j]$$
$$\frac{\partial}{\partial b_j} C(w,b) = \mathbb{E}^q[x_j] - \mathbb{E}^p[x_j]$$

Then, we can perform the actual training using a gradient ascent scheme:

$$w_{ij}^{(k+1)} = w_{ij}^{(k)} + \eta \frac{\partial}{\partial w_{ij}} C(w,b)$$
$$b_j^{(k+1)} = b_j^{(k)} + \eta \frac{\partial}{\partial b_j} C(w,b)$$

The idea is that Boltzmann distribution is the equilibrium distribution reached in thermal equilibrium. By changing the model parameters, the Boltzmann distribution itself changes.

4.4.4 Restricted Boltzmann Machines

Restricted Boltzmann machines is a special type of Boltzmann machine, where visible states are only permitted to connect with hidden states and vice versa. Hence, we restrict connections between hidden states and connections with visible states.

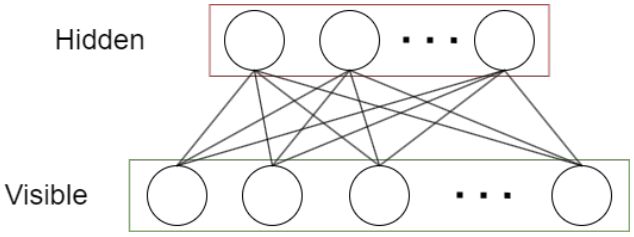


Figure 4.12: Restricted Boltzmann Machine's Computational Graph

Due to this restriction the symmetry between weights is lost. Now, the distinction between hidden and visible states: $x = (u, h)$ is even more important. The energy of a state x is computed as:

$$E(x) = E[(u,h)] = -\tfrac{1}{2} u^T w h - b^T u - c^T h$$

and the unconditional probability of a state (or the joint probability of visible and hidden states) is:

$$p(u,h) = \tfrac{1}{Z(u,h)} \exp\{-E(u,h)\}$$

Once again, the computation of the partition function Z is impossible in an applied setting, and thus, during training, we have to bypass the problem using conditional distributions. At first, we need to observe that the conditional distribution of the hidden states given the visible state is independent [2]. Then, we get that:

$$p(h_n|u) = \frac{\exp\{(\sum_j u_j w_{jn} + c_n)h_n\}}{1+\exp\{\sum_j u_j w_{jn} + c_n\}}$$

From this result derive the following probabilities:

$$p(h_n = 1|u) = \sigma(\sum_j u_j w_{jn} + c_n)$$
$$p(h_n = 0|u) = \sigma(-\sum_j u_j w_{jn} - c_n)$$

Training Restricted Boltzmann Machines

During training of a restricted Boltzmann machine, we can follow the exact same process with a simple Boltzmann machine. This time, the MCMC sampler produces $p(h|u)$ to approximate the distribution $q(h|u)$. Similarly, to minimize the Kullback-Leibler divergence, we have to maximize the objective function:

$$C(w, b) = \sum_h q(h|u) \ln p(h|u)$$

We end up with a similar gradient ascent scheme as before:

$$w_{ij}^{(k+1)} = w_{ij}^{(k)} + \eta[\mathbb{E}^{q_{h|u}}(h_i h_j) - \mathbb{E}^{p_{h|u}}(h_i h_j)]$$
$$c_j^{(k+1)} = c_j^{(k)} + \eta[\mathbb{E}^{q_{h|u}}(h_j) - \mathbb{E}^{p_{h|u}}(h_j)]$$
$$\Delta b = 0$$

4.5 Stochastic Neural Networks II: Generative Adversarial Networks

Like Boltzmann machines, generative adversarial networks (GANs) are deep generative models, with the task to produce artificial data similar to those observed in the training data set. The ability to generate data efficiently made the use of generative adversarial networks widespread in many sectors. In finance, these networks have been applied in derivative pricing and time series forecasting by being the building block of a new type of model; the neural stochastic differential equations.

4.5.1 Density Estimation

The typical task of generative models is to create new samples from a distribution $p_{data}(x)$ based on observations of that distribution (the training data). There are two main categories of distributions that generative models try to reconstruct; parametric and non-parametric.

· Parametric: We estimate the p_{data} distribution using a parametric distribution $p_{model}(x;\theta)$, where the parameters θ are tuned in order to maximize the log-likelihood [2]:

$$\theta^* = \arg\max_\theta \mathbb{E}_x[\ln p_{model}(x;\theta)]$$

and in an applied setting, we estimate the above expected value as:

$$\mathbb{E}_x[\ln p_{model}(x;\theta)] = \frac{1}{n}\sum_{i=1}^{n} \ln p_{model}(x_i;\theta)$$

· Non-Parametric: In non-parametric case, instead of taking a parametric distribution and calibrate its parameters to maximize the log-likelihood, we initially sample a simple distribution [2] and apply a non-linear transformation to those samples in order to approximate the samples (data) from the target distribution p_{data}.

4.5.2 Game Theoretic Approach

The intuition behind generative adversarial networks can be found in adversarial games. In an adversarial game, two players compete for the value of an bivariate objective function $V(x, y)$. The first player controls variable x and wants to minimize the value of the objective function, whereas the second player controls y and wants to maximize the function.

This problem can be treated as a mini-max game, with:

$$(x^*, y^*) = \arg\max_y \min_x V(x, y)$$

We can find the equilibrium using the simultaneous gradient descent method in continuous time [2]. The first player wants to move to the direction that minimizes V, thus:

$$x(t + \eta) = x(t) - \eta \frac{\partial V}{\partial x}$$

On the other hand, the second player moves to the direction that maximizes V:

$$y(t + \eta) = y(t) + \eta \frac{\partial V}{\partial x}$$

For $\eta \to 0$, we get the continuous time differentiable system [2]:

$$\frac{dx}{dt} = -\frac{\partial V}{\partial x}$$
$$\frac{dy}{dt} = -\frac{\partial V}{\partial y}$$

If the equilibrium point exists, it is attained at: $(x^*, y^*) = \lim_{t\to\infty}(x(t), y(t))$ and satisfies the conditions:

$$\frac{\partial V}{\partial x} = 0$$
$$\frac{\partial V}{\partial y} = 0$$

4.5.3 Architecture Design

Generative adversarial networks are based on an adversarial game between two neural networks; the generator and the discriminator. By playing the game iteratively, both networks improve and reach a point, where the generated samples match sufficiently the data.

· Generator network: The input of a generator network is a random noise form a latent space $\mathcal{Z}$ and its output $x = G(z; \theta^{(g)})$ tries to resemble points from the data space [2], where G is the generator function and $\theta^{(g)}$ the generator's parameters. The generator function must be differentiable and the random variable Z of the latent space $\mathcal{Z}$ must be of less dimension than the output random variable X. Finally, the output random variable reflects the distribution captured by the generator $p_{model}(x; \theta^{(g)})$.

· Discriminator network: The task of the discriminator network is to classify whether its input x is genuine training data. Given its input, it produces as output the probability $D(x;\theta^{(d)})$ that the input belongs to the training data, with $\theta^{(d)}$ being the discriminator's parameters. In case that $D(x;\theta^{(d)}) = 0$, the discriminator completely rejects the input.

When the training begins, the generator produces only random noise and the discriminator has only limited ability to distinct between real and fake data. The objective of the generator is to produce fake data that the discriminator cannot reject. As the game progresses, the discriminator becomes more effective and, to combat this, the generator produces higher quality data. In the end of the training process, the discriminator assigns $D(x;\theta^{(d)}) = 0.5$ to the generator's output and the game terminates. Then, we can discard the discriminator and keep only the generator.

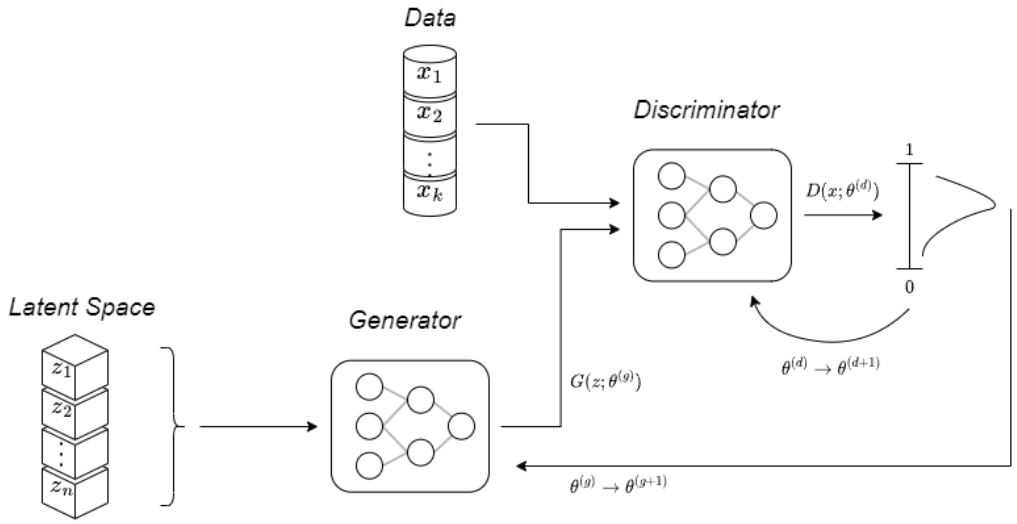


Figure 4.13: Generative Adversarial Network's Architecture

The objective function that the discriminator tries to maximize [2] and, thus, the loss function that the generator tries to minimize is:

$$V(G,D) = \mathbb{E}_{x\sim p_{data}}[\ln D(x)] + \mathbb{E}_{x\sim p_{model}}[\ln(1 - D(x))]$$

By treating the adversarial game between the discriminator and the generator as a mini-max problem, as seen in Section 5.5.2, the optimal generator is given by the problem:

$$G^* = \arg\min_G \max_D V(G,D)$$

and the optimal generator:

$$D_G^* = \arg\max_D V(G, D)$$

Proposition 4.5.1. *For a generator G, the optimal discriminator function is given as [2]:*

$$D_G^*(x) = \frac{p_{data}(x)}{p_{data}(x)+p_{model}(x)}$$

By having this result, we can find the optimal generator function as:

$$G^* = \arg\min_G V(C, D_G^*)$$

To find this minimum value, we have to first find where the objective function is maximized for the optimal discriminator function [2].

Proposition 4.5.2. *The maximum value of the objective function V is:*

$$V(C, D_G^*) = 2D_{JS}(p_{model}(x)||p_{data}(x)) - \ln 4$$

Then, we can minimize the generator function based on the above result [2].

Proposition 4.5.3. *The global minimum*

$$G^* = \arg\min_G V(C, D_G^*)$$

is achieved iff $p_{model} = p_{data}$. The minimum value is: $-\ln 4$.

Training Generative Adversarial Networks

The most common training method of generative adversarial networks is the simultaneous stochastic gradient descent [2]. We sample two mini-batches; one from the actual training data and one from the generated ones, and we apply simultaneously gradient descent and ascent for the discriminator and generator respectfully, in the same way that was described in Section 4.5.2 for players x and y:

$$\theta_{n+1}^{(d)} = \theta_n^{(d)} - \eta\nabla_{\theta^{(d)}} V(\theta_n^{(d)})$$
$$\theta_{n+1}^{(g)} = \theta_n^{(g)} + \eta\nabla_{\theta^{(g)}} V(\theta_n^{(g)})$$

with $\eta > 0$ being the learning rate. Finally, as mentioned in [2], there are still many unsolved problems with training of generative adversarial networks, such as vanishing gradients, mode collapse or failure to converge.

Chapter 5

An Indicative Example: Deep Portfolio Theory

> A good portfolio is more than a long list of good stocks and bonds. It is a balanced whole, providing the investor with protections and opportunities with respect to a wide range of contingencies.
>
> *Harry Markowitz*

5.1 Introduction

Neural networks have been applied to a variety of fields, from computer vision to bio-informatics, presenting accurate modelling results while simultaneously having a sandbox nature compared to other modelling techniques. In this chapter, we present an application of neural networks to a quantitative finance problem; deep portfolios

Deep portfolio theory, as introduced in [4], provides a new data-driven approach to portfolio construction. While retaining the central idea of Markowitz that portfolio allocation is a trade-off problem between risk and expected return,

the proposed procedure is far less model-dependent and adopts an auto-encoding view of the market. Instead of using conventional dimension reduction methods, a key concept of deep portfolios are deep factors [4]; hidden layer abstractions that have non-linear relationship with the input data.

5.2 Deep Portfolio Construction

The goal of deep portfolio theory is to construct a portfolio that performs a specific objective, e.x. replicate of beat a certain index, by using a data-driven procedure. Assuming that the input data can be divided into training and validation data X_{train} and X_{val} respectively, the process proposed in [4] can be broken down into four parts:

I. Auto-encoding

Auto-encodes X_{train} in order to have a more efficient structure of it from an information perspective. This is equivalent with tuning the neural network $F_m(X_{train}; W)$ by solving the regularization problem for a grid of values for the regularization constant λ_m:

$$\min_{W_m} ||X_{train} - F_m(X_{train}; W_m)||_2^2, \text{ with: } ||W_m|| \leq \lambda_m$$

II. Calibration

For a specific objective Y_{train}, e.g. approximate an index, approximate Y_{train} through fitting the neural network $F_p(X_{train}; W)$ by solving the regularization problem for a grid of values for the regularization constant λ_p:

$$\min_{W_p} ||X_{train} - F_p(X_{train}; W_p)||_2^2, \text{ with: } ||W_p|| \leq \lambda_p$$

III. Validation

Select the appropriate λ_m and λ_p values (and consequently the respective neural networks) that balance the trade-off between the following errors:

$$\epsilon^m = ||X_{val} - F_m(X_{val}; W_m^*)||_2^2 \text{ and } \epsilon^p = ||Y_{val} - F_p(X_{val}; W_p^*)||_2^2$$

where W_m^* and W_p^* are the solution of auto-encoding and calibration steps respectively.

IV. Verification

Choose the market and portfolio neural networks F_m and F_p that render the validation step satisfactory by constructing a performance indicator plot.

5.3 Elements of Deep Portfolio Theory

From a mathematical perspective, a linear portfolio can be expressed as: $y = w^T X + b$, with X being the matrix of stock returns and b and w the risk-free rate and the portfolio weights respectively. In the case of target Y approximation, we search for a map F_W, such that: $Y = F_W(X)$. F_W can be treated as a data reduction scheme, as it reduces the input data to approximate the target. During fitting, the parameters W can be found using objective functions that include regularization terms.

5.3.1 Conventional Approaches

Before we develop deep portfolio theory, first we need to approach conventional portfolio theory from an auto-encoding perspective [4]. As reference, we will examine the Markowitz and Black-Litterman models. For each method we have to find how the market information is auto-encoded and then decoded to forecast the evolution of every market's asset.

Markowitz's model can be treated as an encoder, constructed by using only the empirical mean and variance-covariance matrix. From a statistical perspective, these two are sufficient statistics if the market returns follow multivariate normal. However, in an applied setting, the Markowitz's encoder performs poorly due to the structure of financial data, which often display volatility clustering and jumps.

A more advanced approach is the Black-Litterman model, where in the auto-encoding step the investor's beliefs are also included as additional information. More specifically, instead of finding the empirical mean $\hat{\mu}$ and variance-covariance matrix $\hat{\Sigma}$, the Black-Litterman model finds the mean and variance-covariance statistics by solving the regularization problem:

$$||\mu - X||_{\Sigma}^2 + \lambda ||P\mu - q||_{\Omega}^2$$

for a specific investor's view pair (P, q). The problem of the Black-Litterman model is that investor's views do no often reflect the actual evolution of the market.

By approaching the problem from a deep learning perspective, we can solve the regularization problem far more effectively. The main advantage of this method is that by using the performance indicator plot as a predictive tool for the regularization term, the regularization problem we construct incorporates more accurately market information.

5.3.2 Auto-Encoding Perspective of Market Information

As previously stated, the main concept behind deep portfolio theory is to auto-encode available market information and then decode it to forecast asset returns and construct a portfolio accordingly.

Deep Auto-encoders

An auto-encoder is a neural network that learns data coding in an unsupervised manner [5]. Although auto-encoders, which can be treated as a dimension reduction scheme, are related to Principal Component Analysis (PCA), they are more flexible due to their ability to capture both linear and non-linear relationships. An auto-encoder can be broken down into two parts: the encoder that generates a reduced feature representation of the initial data [5] and the decoder, which reconstructs the initial data using the encoder's output. From an architecture's perspective, the auto-encoder has a bottleneck structure, with input and output layers having the same size.

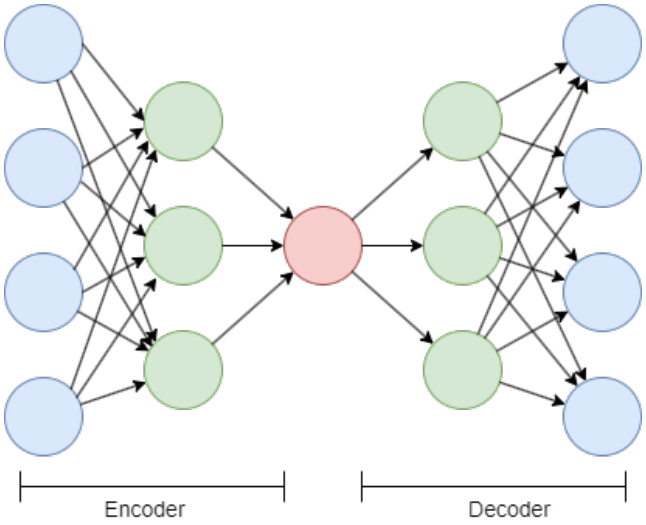


Figure 5.1: Architecture of an Auto-Encoder

For simplicity, we will consider an auto-encoder with only one hidden layer and an activation function f for its hidden nodes. Then, its output can be written as:

$$Y_k(x) = \sum_{j=1}^{N_2} W_{kj}^{(2)} Z_k, \text{ with: } Z_k = f(\sum_{i=1}^{N_1} W_{ji}^{(1)} x_i)$$

Since in the auto-encoder the final model has the form: $X = F_W(X)$, the training can be done using the regularized loss function:

$$\mathcal{L}(W) = \arg\min_W ||X - F_W(X)||^2 + \lambda\phi(W)$$

with

$$\phi(W) = \sum_{ijk} |W_{ji}^{(1)}|^2 + |W_{kj}^{(2)}|^2$$

By introducing a latent factor Z [4] and performing regularization on it, we can break the training process in two steps; the encoding and the decoding. Mathematically, this new problem formulation can be expressed as:

$$\arg\min_{W,Z} ||Z - f(W^{(1)}, X)||^2 + ||X - W^{(2)}Z||^2 + \lambda\phi(Z)$$

where the first term corresponds to the encoder and the rest to the decoder.

Financial Factor Models

Factor models are financial models that approximate financial securities' returns using factors (usually referred to as priced factors); investment characteristics that influence the risk and return of those securities. From a mathematical perspective, a factor model is a linear model that approximates the return sequence $\{r_n\}_{n=1}^N$ using the factors $\{F_k\}_{k=1}^K$ and has the following form:

$$r_n = \sum_{k=1}^K W_{nk}F_k, \, \forall n \in \{1, ..., N\}$$

Using factor models, we can re-write the optimization problem in Step II - Calibration of the proposed procedure as follows [4]:

$$\min_{W,F} \sum_{n=1}^N ||r_n - \sum_{k=1}^K W_{nk}F_k||_2^2 + \lambda \sum_{n,k=1}^{N,K} ||W_{nk}||$$

where the first term represents the reconstruction error and the regularization term is used in Step III to quantify variance-bias trade-off and is important to perform satisfactory out-of-sample predictions. This problem can be solved using a two-step algorithm [4]:

· (A) For given factors F, find the optimal weight values using L^1-norm optimization

· (B) Given the weight values, find the factors using quadratic programming techniques.

In deep learning theory, instead of using quadratic programming, we fit a neural network by introducing a multivariate pay-off function of the initial returns. The theoretical background of multivariate pay-off functions can be found in Kolmogorov-Arnold theorem (see Chapter 3). We can approximate the multivariate pay-off function F as follows:

$$F(x_1, ..., X_n) = \sum_{j=1}^{2N+1} \chi_j \sum_{i=1}^N \psi_{ij}(x_i)$$

for a specific selection of χ_j and ψ_{ij} functions. This representation is important in deep learning, because we know in advance that a deep ReLU feedforward network can approximate F sufficiently. Hence, the quadratic programming problem was reduced to a simple problem of fitting a feedforward network with ReLU activation function.

5.4 Application and Empirical Results

Re-Tracking IBB Index using Deep Portfolio Theory

The objective of this example is to re-track the biotechnology IBB index using a subset of its stocks, using data from January 2012 to April 2016. For the first two steps of the proposed process (auto-encoding and calibration), we examine the period from January 2012 to December 2013 and for the next two steps (validation and verification) we use data from January 2014 to April 2016. Our goal is to reconstruct the index value at each time-step using only a subset of its stocks' values at that time-step.

5.4.1 Traditional Modelling

An initial step would be to perform PCA, instead of using auto-encoders. As discussed previously, auto-encoders are related to PCA and their theoretical advantage is that they are able to capture both linear and non-linear relationships. We will examine if this theoretical result can be verified in an applied setting. Using the first principal component of the whole stock universe, we attempt to represent the index and get the result:

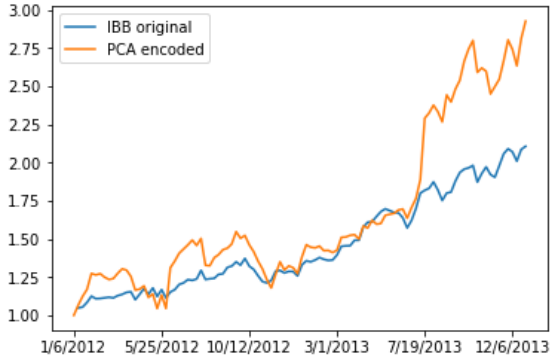


Figure 5.2: PCA encoding calibration results

It is easy to conclude that PCA performs relatively well until July 2013; afterwards its modelling ability collapses. The alternative would be to construct a stock portfolio that tries to reconstruct the index using an auto-encoding perspective.

5.4.2 Deep Learning Modelling

After auto-encoding all stocks, we rank them according to their communal information; information that is common in a large subset of the stock universe and can be treated as a measure of the auto-encoder's performance [4]. Due to the fact that stocks with high communal information entail similar information, we want to select only a restricted number of them (in [4] it is proposed to take only the first 10). On the other hand, we also want to include in our portfolio a number of stocks x with the lowest communal information, to include additional information that is not revealed with the auto-encoding process.

Figure 5.3: Highest and lowest ranking stocks before and after auto-encoding according to communal information

During calibration, we use feedforward networks with ReLU activation function and reconstruct the index for the number of non-communal information stocks $x \in \{15, 35, 55\}$. The results can be found in the following plot:

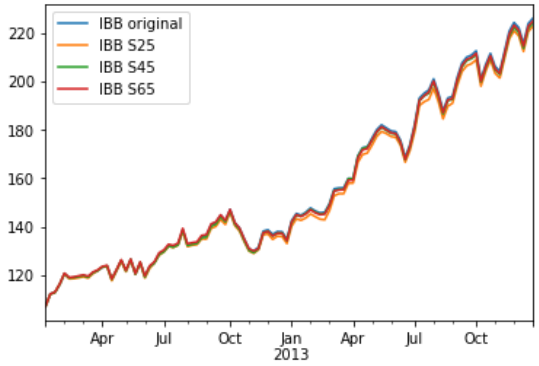


Figure 5.4: IBB calibration using a subset of stocks ($10 + x$, $x \in \{15, 35, 55\}$)

Comparing Results

We can compare the auto-encoding results that we got with a traditional PCA approach to verify whether the use of neural networks is beneficial compared to more traditional methods. By comparing Figure 5.4 and Figure 5.2, it is clear that the auto-encoding yields far better results for all possible portfolios we have considered. Thus, the ability of neural networks to capture non-linear relationships gives them an edge over traditional methods.

5.4.3 Model Validation and Verification

After concluding that deep learning models perform better, we examine the performance of calibrated models in out-of-sample predictions about the future values of the index during validation step. An important remark is that neural networks are extremely vulnerable to overfitting (see Chapter 1). Validation is an important step to observe how our model behaves in more realistic situations. The performance of the deep learning models that were calibrated in Step II, using $10 + x$ stocks, $x \in \{15, 35, 55\}$ is presented in the following graph, with each value being the reconstruction prediction of index value at time t using the portfolio's stocks' values at the same time:

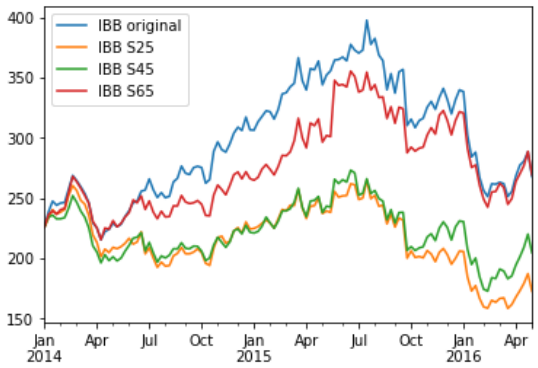


Figure 5.5: Predicted capabilities of calibrated models

It is clear that the portfolio with the biggest number of stocks can re-track the index better. An important remark is that all portfolios capture effectively variations of the index. This is an indication that all portfolios contain stocks that influence heavily the progression of the index.

Using the same process with selecting a subset of communal and non-communal information stocks, we can construct the performance indicator plot [4], which plots the trade-off between the number of stocks in the portfolio and the accuracy error.

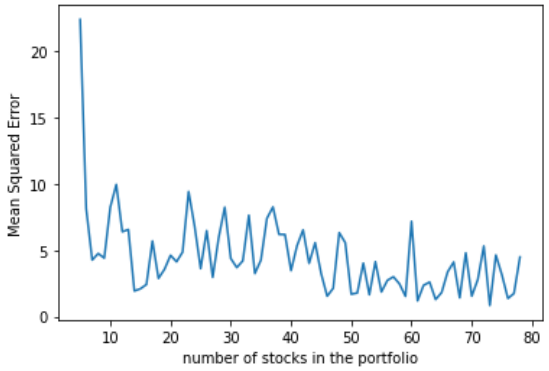


Figure 5.6: Performance Indicator Plot

It can be easily observed that in general, as the number of portfolio's stocks is increased, the error is decreased. However, the tooth-like form of this plots indicates that there are some stock groups that can re-track the index better, compared to including additional stocks. An explanation could be that, as mentioned previously, some stocks heavily influence the progression of the index. Including additional stocks restricts the influence of the dominant stocks in the model's predictions. This result is important because in order to get consistently satisfactory error, we have to include a large number of stocks in our portfolio (greater than 60). On the other hand, by detecting the dominant stocks that influence the index, we can build a smaller portfolio to perform the same exact task.

Finally, during verification step, we conduct model selection by examining the performance indicator plot and try to include only dominant stocks that influence the index. This renders the whole process data-driven and, contrary to traditional approaches, uses the efficient frontier for predictive purposes.

5.4.4 Concluding Remarks

This example was an inferential application of neural networks. By trying to reconstruct and index, we are able to detect which stocks influence its progression significantly, a result that can be later used for predictive purposes. Finally, the use of neural networks instead of traditional techniques can yield better

results, especially when used in combination with regularization that mitigates overfitting problems.

Bibliography

[1] Charu C Aggarwal et al. *Neural networks and deep learning*. Vol. 10. Springer, 2018, pp. 978–3.

[2] Ovidiu Calin. *Deep learning architectures*. Springer, 2020.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[4] JB Heaton, Nicholas G Polson, and JH Witte. "Deep portfolio theory". In: *arXiv preprint arXiv:1605.07230* (2016).

[5] Seshadri Sastry Kunapuli and Praveen Chakravarthy Bhallamudi. "Chapter 22 - A review of deep learning models for medical diagnosis". In: *Machine Learning, Big Data, and IoT for Medical Informatics*. Ed. by Pardeep Kumar, Yugal Kumar, and Mohamed A. Tawhid. Intelligent Data-Centric Systems. Academic Press, 2021, pp. 389–404. ISBN: 978-0-12-821777-1. DOI: https://doi.org/10.1016/B978-0-12-821777-1.00007-0. URL: https://www.sciencedirect.com/science/article/pii/B9780128217771000070.

[6] Guanghui Lan. *First-order and stochastic optimization methods for machine learning*. Springer, 2020.

[7] Jorge Nocedal and Stephen J Wright. *Numerical optimization 2nd edition*. 2006.

[8] Gabrel Turinici. "The convergence of the Stochastic Gradient Descent (SGD): a self-contained proof". In: *arXiv preprint arXiv:2103.14350* (2021).

Appendix A

A.1 Convexity Theory

A.1.1 Differentiable Convex Functions

Definition A.1.1. *If $f : X \to \mathbb{R}$ is differentiable over its domain, f is convex, if and only if:*

$$f(y) \geq f(x) + \langle \nabla f(x), y - x \rangle$$

$\forall x, y \in X$.

A.1.2 Non-Differentiable Convex Functions

The generalization of differentiable convex functions in non-differentiable cases is the subgradient.

Definition A.1.2 (Subgradient)**.** *$g \in \mathbb{R}^n$ is a subgradient of f at $x \in X$, if for any $y \in X$:*

$$f(y) \geq f(x) + \langle g, y - x \rangle$$

The set of subgradients of f at x is called the subdifferential $\partial f(x)$.

A.1.3 Lipschitz Continuous Convex Functions

Definition A.1.3. *A function f is called M-Lipschitz over a set K with respect to a norm $||.||$ if $\forall x, y \in K$ we have:*

$$|f(y) - f(x)| \leq M||y - x||$$

for a constant M*.*

A.1.4 Strongly Convex Functions

Definition A.1.4. *A function* f *is strongly convex, if and only if* $\exists \mu > 0$*, such that:*

$$f(y) \geq f(x) + \langle g(x), y - x\rangle + \frac{\mu}{2}||y - x||_2^2 \ , \ \forall x, y \in X$$

with $g(x) \in \partial f(x)$*. If* f *is also differentiable,* $g(x)$ *is reduced to* $\nabla f(x)$*.*

A.1.5 Smooth Convex Functions

Definition A.1.5. *A function* f *is a smooth convex function, if and only if it is differentiable with Lipschitz continuous gradients:*

$$||\nabla f(x) - \nabla f(y)||_2 \leq M||x - y||_2, \ \forall x, y \in X$$

with $g(x) \in \partial f(x)$*. If* f *is also differentiable,* $g(x)$ *is reduced to* $\nabla f(x)$*.*

A.1.6 Bregman's Distance

Definition A.1.6. *Bregman's distance* $V : X^0 \times X \to \mathbb{R}^+$ *is given by:*

$$V(x, z) = v(z) - [v(x) + \langle \nabla v(x), z - x\rangle]$$

for a distance generating function v*.*

A.2 Approximation Theory

A.2.1 Arzela-Ascoli's Theorem

Instead of constructing a sequence of uniformly convergent continuous functions, another idea is to select a sub-sequence from an existing functions' sequence with these properties.

Theorem A.2.1 (Arzela-Ascoli's Theorem)**.** *Consider a family of continuous functions on* $[a, b]$*:* $\mathcal{F} \subset C[a, b]$*. The following results are equivalent:*

(i) Any sequence $\{f_n\}_{n\geq 1} \subset \mathcal{F}$ *contains a uniformly convergent sub-sequence* $\{f_{n_k}\}_{k\geq 1}$*.*

(ii) $\mathcal{F}$ *is equicontinuous and uniformly bounded.*

A.2.2 Algebra

Definition A.2.1. *An Algebra* $\mathcal{A}$ *is a non-empty set of sets that satisfies the following properties:*

$$
\begin{aligned}
&(i)\ \Omega \in \mathcal{A} \\
&(ii)\ C \in \mathcal{A} \Rightarrow C^c \in \mathcal{A} \\
&(iii)\ C_1, C_2 \in \mathcal{A} \Rightarrow C_1 \cup C_2 \in \mathcal{A}
\end{aligned}
$$