Thesis for the degree of Master of Science

Brunel University of London

Department of Mathematics

Course in Statistics with data analytics

2021/22

Loss functions in deep learning: a comparative review

Author:
Jean Charles Kouame'

Supervisor:
Professor Keming Yu

Date: September 5, 2022

Abstract

An experimental approach founded on theoretical background is what allows us to find logical answers to fundamental questions, and gives us some important guidelines on how to choose a reasonable solution to the problem that we want to solve. By manipulating different parts of a neural network algorithm, it is possible to observe how its behaviour changes, and if this process is done by modifying one piece at a time it is possible to identify what are the reasons for this change, and judge why it is so. In this dissertation we aim to follow an experimental approach, and to understand and describe the design choices for solving a classification problem, focusing particularly on confronting different loss functions.

Contents

Chapter 1

Introduction

1.1 Research aims

Machine learning algorithms are statistical methods used to solve many varieties of problems. They can be classified into four representing types, such as supervised, unsupervised, semi-supervised, and reinforcement learning [62]. It is sufficient to consult a simple source such as Google Trends, that we can see how their popularity has increased significantly in the last years, and how they are being used across a growing number of industries. In general, their effectiveness depends on different aspects regarding their implementation, ranging from theoretical to applied ones. This requires the users to be trained thoroughly on each one of them. This thesis focuses on one very important aspect, which is the choice and interpretation of the system's loss function. We want to explore how loss functions impact a machine learning system, by using a comparative approach, to fully understand their implications, strengths and weaknesses. It is important to highlight that the aim of this thesis is not ranking different loss functions, but comparing them, exposing their main aspects. In fact, one important lesson learned by conducting this research, has been understanding that loss functions must be chosen because of their compatibility with the objective function, rather than the performances they scored on past problems. Other aims of this thesis are:

- Outlining a background theory of the main machine learning methods and loss functions
- Introducing the overfitting issue and how it is relevant for learning systems
- Introducing deep learning methods, with the use of detailed charts, describing the main characteristics and use
- Confronting different loss functions by solving an image classification problem
- Conducting research with an experimental approach, recording results and drawing conclusions based on them
- Giving a critical interpretation of the obtained results, highlighting its coherence with the background theory

We will do so by selecting a real problem to solve (MNIST Handwritten digits classification) and adapting different loss functions in two baseline architectures, and observing the changes in the system. The metrics used to track the changes are the "accuracy" and the "training loss", which allow us to make a comparison between different implementations.

1.2 Walk-through

This work develops by describing the theoretical background that would allow us to understand deep learning, trying to list the most relevant steps that are needed to understand the functioning of these networks. The second part, from chapter 6, analyses and confronts different loss functions applied to a problem (image classification), and draw the conclusions based on the results that have been observed during the experiments.
In chapters 2 and 3 we want to set out a framework for learning systems, to which we will refer and be guided in our analysis. Also we want to introduce the general concepts of what a loss function is and how to formulate an objective function. In particular, the chapters develop by considering regression and classification problems separately, and also separates loss function for regression and for classification. In general, before describing the most relevant methods, it will be given a definition of the loss functions used for that particular category of problem. In chapter 3, we want discuss the problem of overfitting, presenting some approaches to address the issue.
In chapter 4 we highlight the second fundamental component of learning systems: optimising a learning system. To do that we describe extensively the theoretical concept of backpropagation. We will also make clear how the gradients from the backpropoagation phase are used by optimisation methods such as gradient descent and stochastic gradient descent.
In chapter 5 we give some background on deep learning and the most relevant methods. We will layout their functioning by drawing the diagrams and defining them. The chapter will then focus with more attention on the presentation of Convolutional neural network (CNNs), because of their relevance in the application of image classification problems.
In chapters 6 and 7 we will compare different loss functions, and record the obtained results. We will also give a brief description of network design and show the results on a series of comparative charts. The conclusion will sum up and organize the results from the experiments.

Chapter 2

Objective of statistical learning

2.1 A learning framework

Learning algorithms are designed as a combination of three fundamental parts, that have to undergo careful attention by the user that defines them. Statistical learning means that the specific computations performed by the systems are not programmed, but instead learned. To make learning happen in the most efficient way possible, we need to define the following aspects [3]:

- **Objective function**: or cost function, describes if the learning goal is reached, by quantifying how well the prediction aligns with the expected results.
- **Optimization method**: defines how the learning will happen, or in more specific terms, how the weights that characterize the network are updated. This strategy is important to maximize the efficiency during the computation tasks.
- **Network architecture**: are the pathways and connections of the information flow, units are implemented by varying a wide range of possible combinations, ranging from highly simplified operations to relatively complex models.

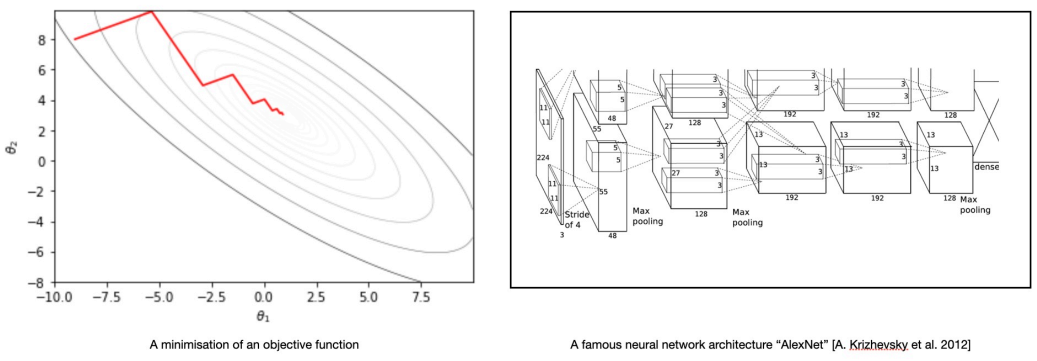

θ₂
θ₁
A minimisation of an objective function
Stride of 4
Max pooling
Max pooling
Max pooling
dense
A famous neural network architecture “AlexNet” [A. Krizhevsky et al. 2012]

Figure 2.1: The three fundamental elements of statistical learning, A minimisation of an objective function [14] and A famous neural network architecture "AlexNet" [44]

The image above (fig. 2.1) represents the three components that we defined. On the left a minimisation of an objective function obtained with the optimization method of gradient descent (code in Appendix A.1). On the right an example of a famous neural network architectures, Alexnet [44]. The choice of each of these components has a fundamental impact on the performances of the learning systems. We will see in the following chapters why this is important.

2.2 Intuition of loss function

Algorithms are used to learn sets of parameters that can generalize a task such as regression or classification, on a specific data set. We can define how good or how bad the learning process is proceeding, by measuring the difference between the prediction given from an algorithm (with a current set of parameters) and the desired score. The function that measures this difference is called "loss function". Let us consider a data set defined as $\{(x_i, y_i)\}_{i=1}^N$, where x_i is a data point of the set and y_i is the correspondent ground truth. We can define a function that predicts a score for each point in the set:

$$s_i = f(x_i, W) \tag{2.1}$$

where W are a set of weighting parameters. Then the score s_i quantifies the belief of the network that the input x_i is equal to its ground truth y_i. The score will then be compared with the ground truth y_i, via a function γ called loss function. The function γ, which defines a rule to measure how good or bad the prediction is:

$$l_i = \gamma(f(x_i, W), y_i) \tag{2.2}$$

The final cost will be obtained by averaging the sum of the losses over all training examples:

$$L = \frac{1}{N}\sum_{i=1}^{N} l_i \tag{2.3}$$

2.3 Loss functions for regression

2.3.1 Mean squared error

The Mean Squared Error (MSE) is a measurement of how well a line approximates a set of data points. It is calculated by averaging the sum of every squared error (difference between the data point and the approximation of the same data point) across the whole data set [51]:

$$\mathrm{MSE} = \frac{1}{\mathrm{n}}\sum_{i=1}^{\mathrm{n}} (\mathrm{y}_i - \hat{\mathrm{y}}_i)^2 \tag{2.4}$$

where n is the number of samples over which the MSE is computed, y_i is the true value, and $\hat{\mathrm{y}}_i$ is the prediction. The errors are squared to remove any negative sign, and make this function continuous and differentiable. The MSE gives more importance to large differences, and for this reason the final prediction is strongly affected by large errors. We will see later in chapter 3 how the effect of large errors can be limited.

2.3.2 Mean Absolute Error

The Mean Absolute Error (MAE) is also used in regression methods. It is the mean of the absolute values of the individual prediction errors, over all points in the data set [73].

$$\mathrm{MAE} = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i| \tag{2.5}$$

where y_i is the true value, and $\hat{y}_i$ is the prediction. This function is non differentiable in every point, and therefore the MAE is less frequently implemented with respect to the MSE.

2.4 Regression methods

Regression methods are statistical approaches, that want to describe the relationship that exists between some variables and a dependent variable. These methods try to predict a quantitative score of a so called dependent variable, and the given values of a set of predicting variables, called independent variables. There are many available regression models, such as logistic regression, polynomial regression, quantile regression to cite some of the most representative[17]. Let us focus on one of these methods to highlight its formulation, which is a good representative of these type of models.

2.4.1 Multiple linear regression model

Multiple linear regression extends the concept of simple linear regression [67], where the relationship with the response variable y is defined by a one-dimensional independent variable x. In multiple linear regression, the x variable gets the dimension of k. If we consider a data set defined as $\{(x_i, y_i)\}_{i=1}^{N}$, where x_i is a data point of the set of dimension N, and y_i is the correspondent ground truth, the model will be defined as:

$$y_i = b + w_1 \cdot x_{i,1} + w_2 \cdot x_{i,2} + \ldots + w_k \cdot x_{i,k} + e_i \tag{2.6}$$

where "k" is the dimension of the predicting variable x_i, and "i" is the subscript referring to the individual training example from the data set. The number of total parameters is $k+1$ (k weights, and 1 bias). The error term $_i$ is defined as:

$$e_i = y_i - \hat{y}_i \tag{2.7}$$

and it is assumed belonging to a normal distribution with zero mean and constant variance σ^2. Each w_j coefficient represents the variation in the response variable[57], per unit increase in the associated predictor variable when all the other predictors are held constant [57]. We are interested in estimating the parameters (b, w), and they are obtained with the ordinary least squares method (OLS):

$$S(b, w) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{2.8}$$

$$\min_{b,w} \quad S(b, w) \tag{2.9}$$

where S(b,w) represents the sum of the squared errors to be minimized. We can do so by finding the derivatives of S(b,w) with respect of b and w, $\frac{\partial S(b,w)}{\partial b}$, $\frac{\partial S(b,w)}{\partial w_j}$, where $j = 1, \ldots, k$ is the j-th weight w_j. These derivatives have to be made equal to zero, and have to be solved for b and w_j. We can rewrite the linear system in matrix form:

$$y = XW + e$$

$$\begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_N \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & x_{12} & \dots & x_{1k} \\ 1 & x_{21} & x_{22} & \dots & x_{2k} \\ \dots & \dots & \dots & \dots & \\ 1 & x_{N1} & x_{N2} & \dots & x_{Nk} \end{bmatrix} \begin{bmatrix} b \\ w_1 \\ \vdots \\ w_k \end{bmatrix} + \begin{bmatrix} e_0 \\ e_1 \\ \dots \\ e_n \end{bmatrix}$$

where X is an (N x $(k+1)$) matrix containing N observations from the data set and k-columns of independent variables plus a column of ones. Instead y is an Nx1 vector representing the ground truth, and W is a $[(k+1) \times 1]$ vector, and e is an Nx1 vector. The vector of the residuals is defined as:

$$e = y - X\hat{W} \tag{2.10}$$

The sum of squared residuals (RSS) is simply $e^T e$. To find the $\hat{W}$ that minimizes the sum of squared residuals, we need to take the derivative of RSS with respect of W, equal them to zero and solve for W. The estimator of W is defined in the following closed form:

$$\hat{W} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y} \tag{2.11}$$

These kind of models are used, for example, to predict the price of a house given some predicting variables such as square meters, city, number of rooms, etc. The multiple regression model is also based on the following assumptions[58]:

- the independent and dependent variables are linearly dependent
- the independent variables are uncorrelated
- residuals are normally distributed with mean 0 and variance equal to 1

2.5 Loss functions for classification

Loss functions for classification are functions that quantify the cost when a prediction is not accurate in a classification task. We will consider three main loss functions: hinge loss, cross-entropy loss and Kullback–Leibler divergence loss. These functions are implemented in many classification methods for performing classification tasks.

2.5.1 Hinge Loss

When thinking about Hinge loss, we should think at the x axis of the chart as the distance of the sample data point from a margin, in the case that the sample is correctly classified. The hinge loss function is defined as [46]:

$$H(f(x), y) = \max(0, 1 - y \cdot f(x)) = (1 - y \cdot f(x))^+ \tag{2.12}$$

where f(x) is the model prediction (which can be thought of as the distance from the margin), y is the ground truth label (-1 or 1). The loss is 0 when f(x) $\geq$ 1, while it equals

$(1 - y \cdot f(x))$ when $f(x) < 1$ and the sample is correctly classified.

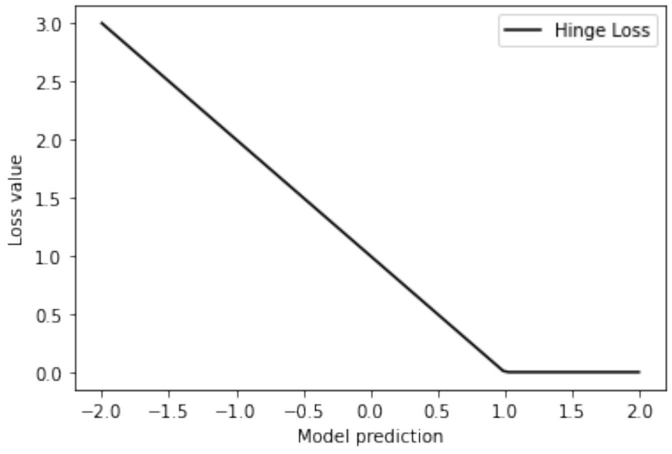


Figure 2.2: Hinge loss function, Appendix A.2

Support vector machines are particularly suited in separating two classes with respect to a hyperplane because they make use of this kind of loss function [40]. The hinge loss function is convex and continuous [52], but it is not differentiable in $y\cdot f(x)=1$. Consequently, the hinge loss function is not adapted to work with methods that involve gradient descent [52]. This function assigns a positive value to the error only when the data point is near or from the wrong side of the margin, and does not assign any error to correctly predicted data points that have a distance bigger than 1 from the margin, because in this type of problem it is not important the margin by which a correct classification is made.

2.5.2 Cross-entropy

Entropy is the metric with which we measure the quantity of information of a system [45], and we will call it H. We want to minimise this amount in many problems. The definition of Entropy for a probability distribution is:

$$H = -\sum_{i=1}^{n} y_i \cdot \log y_i \tag{2.13}$$

where y_i is the probability of event i, and n is the total number of possible events. By using $\log_2$ for our calculation we can think of entropy as the smallest number of bits it would take us to encode a message [12]. Now that we can quantify the entropy, we can measure how much information is lost when we substitute our observed distribution for an approximation of it [12]. Cross entropy is the average amount of information transferred to describe an event [28]. For discrete classes probability distributions, cross-entropy is defined as the sum over all possible classes to be predicted, of the real probability of the training example to belong to a certain class, times the logarithm of the network's predicted probability [21]:

$$CE = -\sum_{i=1}^{\text{classes}} y_i \cdot \log \hat{y}_i \tag{2.14}$$

where y_i is the real probability of a training example of belonging to the i-th class. $\hat{y}_i$ is the predicted probability score for the i-th class. It is important to notice that usually y_i assumes either 0 or 1 value.

This means that when the probability of the training example is 0, the loss doesn't grow: this is so because we are not interested in knowing the loss of the wrong classes, but only the loss of the true classes. Also the sign "–" at the beginning of the equation, balance the negative sign of the logarithm of S_i, which is a number between 0 and 1 after the scores have been normalized.

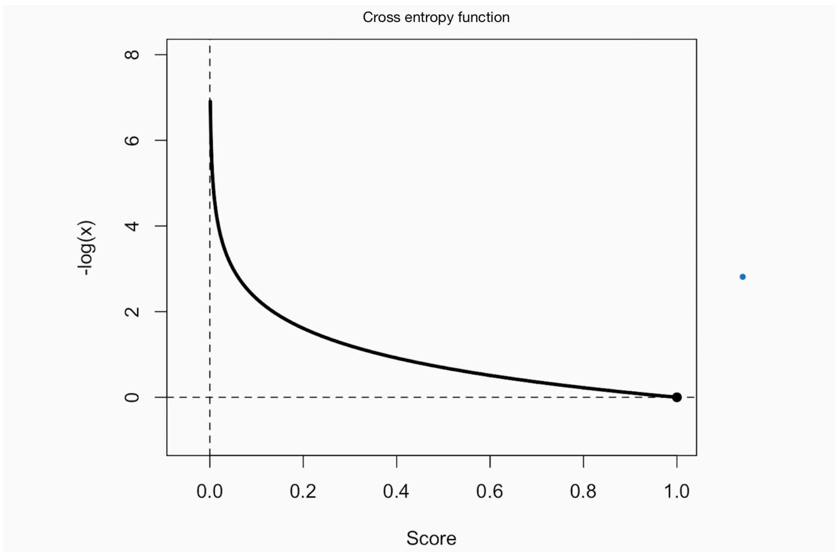


Figure 2.3: Cross-entropy function, Appendix A.3

Categorical cross entropy for one-hot encoding

When the ground truth vector represents a set of classes, and only one class is true, the label vector can be encoded as a set of zeros for all classes that are not true, and a 1 for the real class of the object. This type of encoding will make all the terms of the cross entropy equal to zero, except the one corresponding to the multiplication between the true label and the predicted score for the true class. In this particular case cross entropy can be written as:

$$\mathrm{CCE} = -(1) \cdot \log(\hat{y}_T) \tag{2.15}$$

where y_T is the score for the correct class. We can refer to this loss also as "Multi-class" cross entropy [72]. In most of the cases when using this type of loss, a softmax function is applied to the output layer [77]. The softmax function performs the following operation for each score of the output layer:

$$f(\hat{y}_i) = \frac{e^{\hat{y}_i}}{\sum_{j=1}^{C} e^{\hat{y}_j}} \tag{2.16}$$

where $\hat{y}_i$ is the predicted score fot the i-th class. If we substitute eq. 2.14 in 2.12 we will obtain the following formulation for a multi-class cross entropy with a softmax activation function:

$$\mathrm{CCE} = -\sum_{i=1}^{\text{classes}} y_i \cdot \log \frac{e^{\hat{y}_i}}{\sum_{j=1}^{C} e^{\hat{y}_j}} \tag{2.17}$$

Binary cross-entropy

Binary cross entropy is used when there are only two possible classes (true or false), and when, unlike the CCE case, the loss computed for every predicted output component is not affected by other component values [50]. This is why this loss is often used for multi-label classification [72]. The formulation for the binary cross-entropy is:

$$\mathrm{BCE} = -\sum_{i=1}^{2} y_i \log(\hat{y}_i) = -y_1 \log(\hat{y}_1) - (1 - y_1)\log(1 - \hat{y}_1) \tag{2.18}$$

where y_1 is the true score for class C_1 and $\hat{y}_1$ is the predicted score for the same class. The terms for the second class C_2 can be expressed in terms of C_1, as $y_2 = 1 - y_1$, and $\hat{y}_2 = 1 - \hat{y}_1$ for the ground truth and the predicted score values respectively. Alternatively BCE can be written as:

$$\mathrm{BCE} = \begin{cases} -\log(\hat{y}_1) & if \quad y_1 = 1 \\ -\log(1 - \hat{y}_1) & if \quad y_1 = 0 \end{cases} \tag{2.19}$$

2.5.3 Kullback–Leibler divergence

Kullback-Leibler Divergence is the difference between cross entropy and entropy [28]. By modifying the formula for the entropy, we obtain the KL divergence loss formulation.

$$D_{KL}(p \| q) = \sum_{i=1}^{n} p(x_i) \cdot (\log p(x_i) - \log q(x_i)) \tag{2.20}$$

where p and q are two different probability distributions. Equation 2.19 can also be written as:

$$D_{KL}(p \| q) = \sum_{i=1}^{n} p(x_i) \cdot \log(\frac{p(x_i)}{q(x_i)}) \tag{2.21}$$

meaning that with KL divergence we can measure the lost information when we approximate the true distribution with a different one [12].

2.6 Classification methods

Classification methods predict the target class by learning from the training data [38]. We can distinguish between binary classification, multi-class classification and multi-label classification problems. We will now present the most representative examples for classification methods.

2.6.1 Support vector machines

Support vector machines are non-probabilistic binary linear classifiers, that during training aim to maximise the width of the gap between two categories [59]. If we consider a data set composed by data points belonging to one of two categories, an SVM training algorithm will determine a separating hyperplane that separates one category from the other [59]. New examples are then mapped into that same space and predicted to belong to a category based on which side of the hyperplane they fall. We will define SVMs as linear classifiers, but they are able to classify also non-linearly separable classes by using the so called kernel trick, that maps the training data set into a high-dimensional feature space [71]. Let us define the perpendicular distance of a data point from an hyperplane $H : \boldsymbol{w}^T \cdot \boldsymbol{x} + w_0 = 0$ as [34]:

$$\mathrm{D} = \frac{|\boldsymbol{w}^{\mathrm{T}} \cdot \boldsymbol{x} + \mathrm{w}_0|}{||\boldsymbol{w}||_2} \tag{2.22}$$

where $\boldsymbol{w} = [w_1 w_2 ... w_p]$ is the vector of the hyperplane coefficients, and $\boldsymbol{x} = [x_1 x_2 ... x_p]^T \in \mathbb{R}^p$ is the data point vector of p-dimension. The distance of the closest data point from the hyperplane H is called margin [34]:

$$\mathrm{M} = \min_{\boldsymbol{x} \in \mathrm{S}} \quad \frac{|\boldsymbol{w}^{\mathrm{T}} \cdot \boldsymbol{x} + \mathrm{w}_0|}{||\boldsymbol{w}||_2} \tag{2.23}$$

where S is the set of the training data, $|\boldsymbol{w}^T \cdot \boldsymbol{x} + w_0|$ the non-perpendicular distance, and $||\boldsymbol{w}||_2$ normalizes the non-perpendicular distance to a perpendicular one. We now want to find an hyperplane by changing $(\boldsymbol{w}, w_0)$ maximizing M. [34]:

$$\begin{cases} \max_{\boldsymbol{w}, w_0} \quad M \\ \text{s.t.} \quad y_i \cdot (\boldsymbol{w}^T \cdot \boldsymbol{x}_i + w_0) \geq 0 \quad \forall \text{ i} \end{cases} \tag{2.24}$$

where $y_i \cdot (\boldsymbol{w}^T \cdot \boldsymbol{x}_i + w_0)$ is the perpendicular distance and y_i is the training example ground truth. The maximization happens by updating the parameters of the hyperplane $(\boldsymbol{w}, w_0)$, which represent the position of the hyperplane in the space. The condition in this problem requires that every training point is classified correctly (the product between the ground truth and the sign of the distance need to have the same sign), therefore we can consider this a "hard margin" condition. By substituting the definition of M we can obtain the following [34]:

$$\begin{cases} \max_{\boldsymbol{w}, w_0} \quad \min_{\boldsymbol{x} \in S} \quad \frac{|\boldsymbol{w}^T \cdot \boldsymbol{x} + w_0|}{||\boldsymbol{w}||_2} \\ \text{s.t.} \quad y_i \cdot (\boldsymbol{w}^T \cdot \boldsymbol{x}_i + w_0) \geq 0 \quad \forall \text{ i} \end{cases} \tag{2.25}$$

Because the hyperplane has the property of being scale invariant, we can impose $(\boldsymbol{w}, w_0)$ in such a way that the distance of the closest data point is $\min_{x \in S} |\boldsymbol{w}^T \cdot \boldsymbol{x} + w_0| = 1$. This causes the maximization problem to become a minimization one [34]:

$$\max_{\boldsymbol{w}, w_0} \frac{1}{||\boldsymbol{w}||_2} \cdot 1 = \min_{\boldsymbol{w}, w_0} ||\boldsymbol{w}||_2 \tag{2.26}$$

The final maximum margin classifier formulation is the following [34]:

$$\begin{cases} \min_{\boldsymbol{w}, w_0} ||\boldsymbol{w}||_2 \\ \text{s.t.} \quad y_i \cdot (\boldsymbol{w}^T \cdot \boldsymbol{x}_i + w_0) \geq 1 \quad \forall \text{ i} \end{cases} \tag{2.27}$$

We can notice that the support vectors have been defined as those training points for which it is true $y_i \cdot (\boldsymbol{w}^T \cdot x_i + w_0)$.

Soft constraints

We can re-formulate the maximum margin classifier problem in a similar one, which implement a variable that allows the data points to cross the margins, or also allows the margins to expand after the closest point. The new problem becomes [34]:

$$\begin{cases} \min_{w,w_0} \quad \|\boldsymbol{w}\|_2 + C\sum_{i=1}^{n}\xi_i \\ \text{s.t.} \quad y_i \cdot (\boldsymbol{w}^T \cdot \boldsymbol{x}_i + w_0) \geq 1 - \xi_i \quad \forall \text{ i} \\ \text{s.t.} \quad \xi_i \geq 0 \quad \forall \text{ i} \end{cases} \tag{2.28}$$

The parameter ξ_i is called "slack variable" and allows the constraint to be more permissive, by classifying correctly also those data points that have a perpendicular distance smaller than 1 ($y_i \cdot (\boldsymbol{w}^T \cdot \boldsymbol{x}_i + w_0) \geq 1 - \xi_i$).

- if $\xi_i = 0$ the data point respects the margin and has a perpendicular distance from the hyperplane greater than 1, lying from the correct side of the margin.
- if $0 > \xi_i \geq 1$ the data point lies between the hyperplane and the correct margin
- if $0 > \xi_i > 1$ the data point lies from the wrong side of the hyperplane but it is still considered correctly classified

C is a parameter that regularizes the trade off between a low training error and a high model's variance, helping the model to make a better generalisation [75]. If C is large the optimization algorithm will shrink $(\boldsymbol{w}, w_0)$ leading to a hyperplane that does not missclassifies any of the training example . This will lead to an over-fit of the data set, losing the ability to generalize. Instead if C is too little then the objective function will not have restrictions in enlarging $(\boldsymbol{w}, w_0)$, leading to a softer margin, with the consequence of a bigger training error.

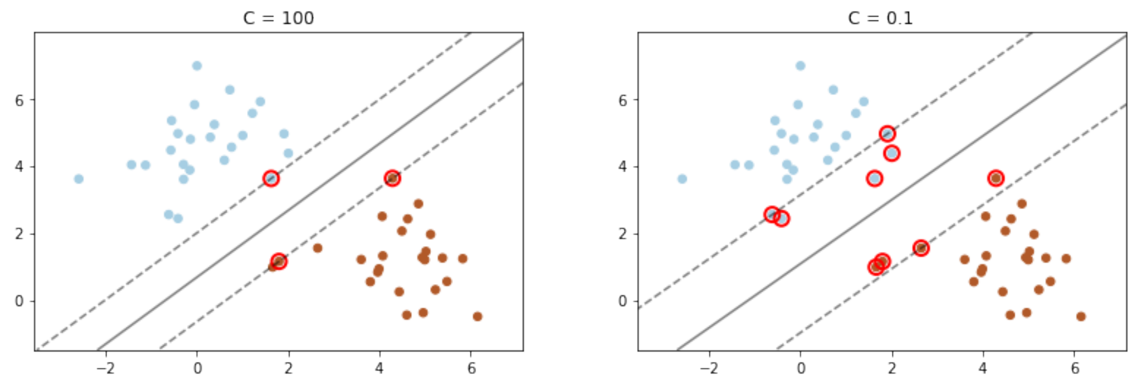


Figure 2.4: Choice of hyperparameter C in a linear kernel SVM [64], Appendix A.4

2.6.2 Naïve Bayes classifiers

Bayes' theorem is the principle on which it is possible to build a class of classifiers that share the same principle and are able to detect the class C_k labels of data inputs. The Bayes'theorem states that:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} \tag{2.29}$$

where $P(A|B)$ is a conditional probability: it is the probability that the event A will occur if B is true. It is also called the posterior probability of A given B. $P(B|A)$ is also a conditional probability: the probability of event B occurring given that A is true. $P(A)$ and $P(B)$ are the observed probability of A and B. They are known as the marginal/prior probability.

Model

Let us assume a data instance $\mathbf{x} = [x_1, \ldots, x_n] \in \mathbb{R}^n$ is given. We are interested in predicting its class C_k, which can assume K different values. We can express the Bayes' theorem as:

$$P(C_k|\boldsymbol{x}) = \frac{P(\boldsymbol{x}|C_k) \cdot P(C_k)}{P(\boldsymbol{x})} \tag{2.30}$$

where $P(C_k|\boldsymbol{x})$ is the prediction for C_k after an input feature is given to the classifier. $P(\boldsymbol{x}|C_k)$ is the prior probability extracted from the training data, $P(\boldsymbol{x})$ is the probability of the combination of the features. Naïve Bayes classifiers assume that the features are independent from each other, therefore $P(\boldsymbol{x}|C_k) = \prod_{i=1}^{n} P(x_i|C_k)$. Also $P(C_k)$ and $P(\boldsymbol{x})$ are respectively the observed probability of the class C_k and the observed probability of the features combination $\boldsymbol{x}$. The prior probabilities correspond to proportions of classes in the sample [**ccc**]. Equation 2.9 can be re-written as:

$$P(C_k|x_1, \ldots, x_n) = \frac{\prod_{i=1}^{n} P(x_i|C_k) \cdot P(C_k)}{P(x_1, \ldots, x_n)} \tag{2.31}$$

Since the denominator $P(x_1, \ldots, x_n)$ is a constant with respect to the class label C_k, the conditional probability is proportional to the numerator:

$$P(C_k|x_1, \ldots, x_n) \propto \prod_{i=1}^{n} P(x_i|C_k) P(C_k) \tag{2.32}$$

These calculations are performed on the log scale:

$$\log(P(C_k|x_1, \ldots, x_n)) \propto \sum_{i=1}^{n} \log(P(x_i|C_k)) + \log(P(C_k)) \tag{2.33}$$

The classifier is the function that finds the class label $\hat{C_k}$ for some K with the highest log-posterior probability:

$$\hat{C} = \underset{k \in 1,\ldots,K}{\arg\max} \sum_{i=1}^{n} \log(P(x_i|C_k)) + \log(P(C_k)) \tag{2.34}$$

Chapter 3

Overfitting issues

The learning process depends strictly from the characteristics of the data that we feed into the network. The parametrization learned will aim to generalize the structure behind the data. Some problems can come-up when the network learns an extremely similar representation of the training data set, but misses generalizing data that it never saw. This phenomenon is called "overfitting", and it is important to take it into careful consideration when implementing a machine learning problem. We will present in this chapter different methods that make it possible for the network to learn "better" parameters: Ridge regression and Lasso regression are methods used in regression problems, as well as the Huber loss function, that allows the system to be less sensitive to the effects of outliers.

3.1 Bias-Variance trade off

When building a model it is important to measure the expected test error to asses the performances of it. This assessment can help us in choosing between different models when compared. The analysis of the test error reveals that when a model tries to fit too closely some data points, it will have to decrease the error made, but it will have to increase the way it can move around its mean, creating "wiggly" functions. The opposite is also true, so when the model tries to reduce the regression function from being "wiggly", reducing the variance, it will have to increase the error made. These two conditions can bring respectively to an overfitting or underfitting of the training data. Ideally, we want to find a model that can both capture the information in the training data, but also generalize well to unseen data. To achieve this result we need to find the best balance between the two problems.

Now, suppose that we have a data set $\{(x_i, y_i)\}_{i=1}^{N}$ and that an approximating function $f(x)$ exists. We can choose between different model functions that are defined as $\hat{f}(x)$. It is possible to demonstrate [33] that the expected error can be defined as:

$$E[(y - \hat{f}(x))^2] = \text{Bias}[\hat{f}(x)]^2 + \text{Var}[\hat{f}(x)] + \sigma^2 \tag{3.1}$$

where σ^2 is an irreducible error due to the data distribution, that is not possible to manipulate. We can instead give an intuition of what the other two error terms represent:

$$\text{Bias}[\hat{f}(x)] = E[\hat{f}(x)] - f(x) \tag{3.2}$$

where $E[\hat{f}(x)]$ is the mean of our model predictions, while $f(x)$ is out target: therefore the difference between these two points quantifies how much our model is drawn in to predicting a different value from the real one. We can say that the model is being "biased" to a particular kind of solution. We can also say that the bias is a characteristic of the model. The term representing the variance is defined as:

$$\mathrm{Var}[\hat{f}(x)] = E[(E[\hat{f}(x)] - \hat{f}(x))^2] \tag{3.3}$$

where the inside term $E[\hat{f}(x)] - \hat{f}(x))^2$ is always positive because of the squaring, and it represents how far the expectation of the model's prediction is from the model's predictions. We can repeat that for different training points, and then taking the expectation of the distribution obtained, capturing how much the models prediction moves on average around its mean. The figure below (fig. 3.1) represents intuitively the concepts that we have just presented.

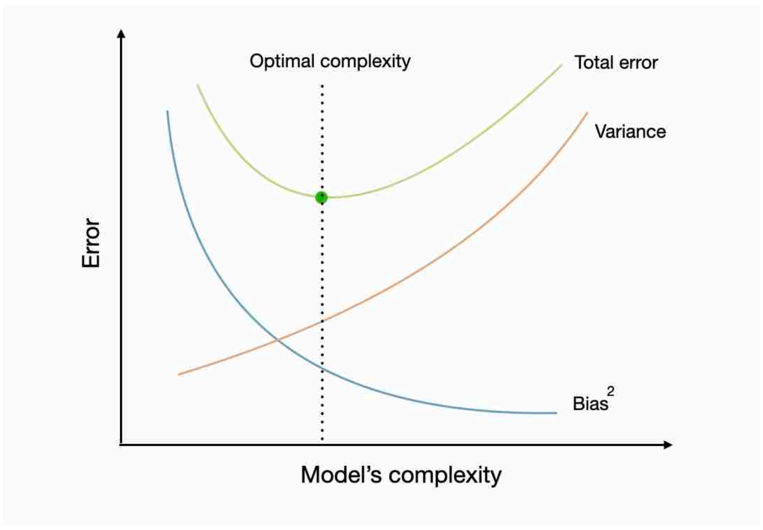


Figure 3.1: Bias-variance trade off

3.2 Regularization

To deal with the bias-variance trade off, one of the possible methods is to "shrink" the model's weights, in order to have an impact on the variance of the model, by reducing it. This method is called "regularization", and it is used for tuning the model's function by adding an additional penalty term in the cost function. There are two main types of regularizations: the Ridge and the Lasso regression methods [35].

3.2.1 Ridge regression

This method is very similar to OLS method. Let us start by considering the OLS estimator formulation:

$$\hat{\mathbf{w}}_{\mathbf{RSS}} = (\mathbf{X}^{\mathbf{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathbf{T}}\mathbf{Y} \tag{3.4}$$

we then just need to add a constant term λ to the elements of the diagonal of the matrix $\mathbf{X}^{\mathbf{T}}\mathbf{X}$ before inverting it [55]. Then the ridge regression estimator will result in:

$$\hat{\mathbf{w}}_{\mathbf{Regularized}} = (\mathbf{X}^{\mathbf{T}}\mathbf{X} + \lambda\mathbf{I}_{\mathbf{P}})^{-1}\mathbf{X}^{\mathbf{T}}\mathbf{Y} \tag{3.5}$$

where $\mathbf{I}_p$ is a diagonal matrix of the same dimension of $\mathbf{X}$. The ridge regression parameter is $\lambda||\mathbf{w}||_2^2$ and is used to penalize the RSS [55]:

$$\min_{w,b} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 + \lambda||w||_2^2 \tag{3.6}$$

where $||w||_2^2 = \sum_{i=1}^{p} w_j^2$ and p is the number of weights, and $\hat{y}_i = b + \boldsymbol{XW}$ and λ is a tuning parameter ≥ 0 arbitrarily chosen. The new term has the effect of reducing the parameters' size, shrinking them towards zero. When $\lambda = 0$ the penalty term has no effect on the minimization problem, and it will be reduced to a simple RSS minimization. If $\lambda \to \infty$ then the effect will be one of shrinking the weights to zero, causing also the variance to be equal to zero. [35]. In this last case, the regression function will have to be a constant (zero variance), with a very high expected error rate.

3.2.2 Lasso regression

Lasso regression has a very similar formulation to the Ridge regression one, except that the regularization term instead of being a L2-norm it will be a L1-norm.

$$\min_{w,b} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 + \lambda||w||_1 \tag{3.7}$$

where $||w||_1 = \sum_{j=1}^{P} w_j$ and $\hat{y}_i = b + \boldsymbol{XW}$, with P being the number of weights to estimate. Lasso regression tries to remedy one of the disadvantages of Ridge regression, which is that Ridge regression always includes all the p-weights in the final regression. Unlike Ridge regression, Lasso is able to select a subset of the weights and setting the non-impactful one equal to zero. Because of this, Lasso regression produces models that are more interpretable that the ones from Ridge [60].

It is possible to demonstrate that Ridge and Lasso regressions are equivalent to solve the following problems [35]:

$$\min_{w,b} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \qquad \text{s.t.} \quad ||w||_2^2 \leq c \tag{3.8}$$

$$\min_{w,b} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \qquad \text{s.t.} \quad ||w|| \leq c \tag{3.9}$$

Figure 3.2 illustrates a visualization of these two problems, where the light blue areas identify the constraints, and the plane is the space of all possible weights that minimize the objective function.

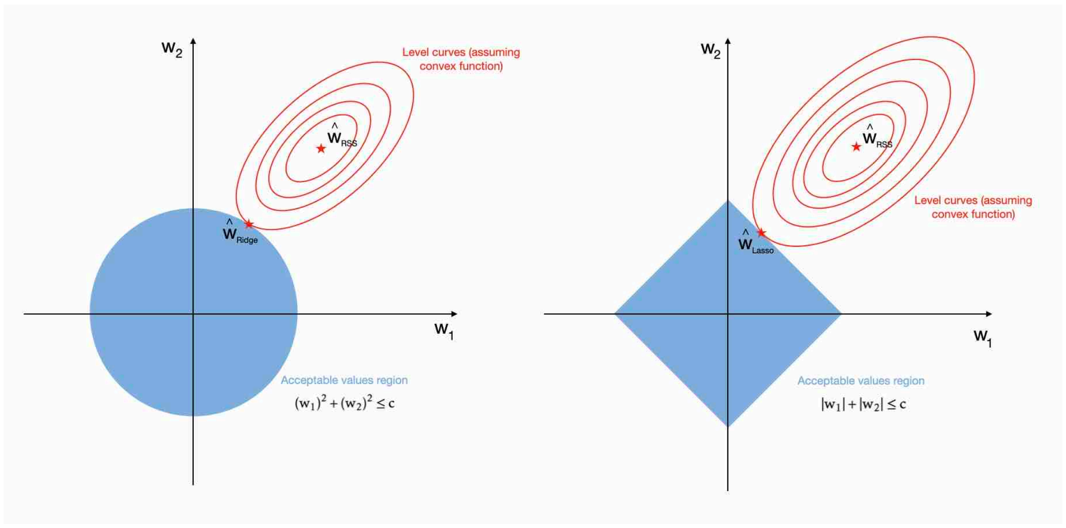


Figure 3.2: Visualization of Ridge and Lasso regressions

3.3 Robust regression

Real world data sets are more likely to contain data that not necessarily deliver information regarding the problem to solve. These kind of data, are described as "outliers", and the effect that they have, is to influence the model in a deceitful way. That is why there is the necessity to build models that are able to avoid the impact of such data. Robust regression is a technique that can reduce the impact of outliers. It should be noted that the linearity assumption is still needed for proper inference using robust regression [4].

3.3.1 Limiting the effect of outliers

In robust regression, the estimations are consistent only under certain assumptions. It is usually assumed that the distribution of the error term is symmetric or the data are symmetrically contaminated by outliers. Other assumptions to make in robust regression are [53]:

- The relationship between the independent variable and dependant variables is linear.
- The expected value of the residuals is zero $E[\epsilon_i] = 0$.
- Residuals ϵ_i are normally distributed with mean 0 and variance σ^2.
- Homoscedasticity: all ϵ_i have the same variance σ^2.
- All ϵ_i are independent.

In OLS outliers are heavily weighted, because the cost function has a quadratic form, and rapidly increases for residual values above 1. Inversely, in OLS, values below 0 are underestimated because of the squaring. When the data are contaminated with extreme values, the regression coefficients estimated with the OLS method are therefore largely making the regression less accurate. In fact outliers have the capacity to stretch the regression line in their direction.

By considering the dataset "Starts" in "robustbase" from R, we can show an intuitive example of what is the effect of outlier on the linear regression line obtained through the OLS method. In the plot on the left side, we can see the regression line obtained with the "lm" method in R, and on the right side the regression line obtained with the "lmrob" method.

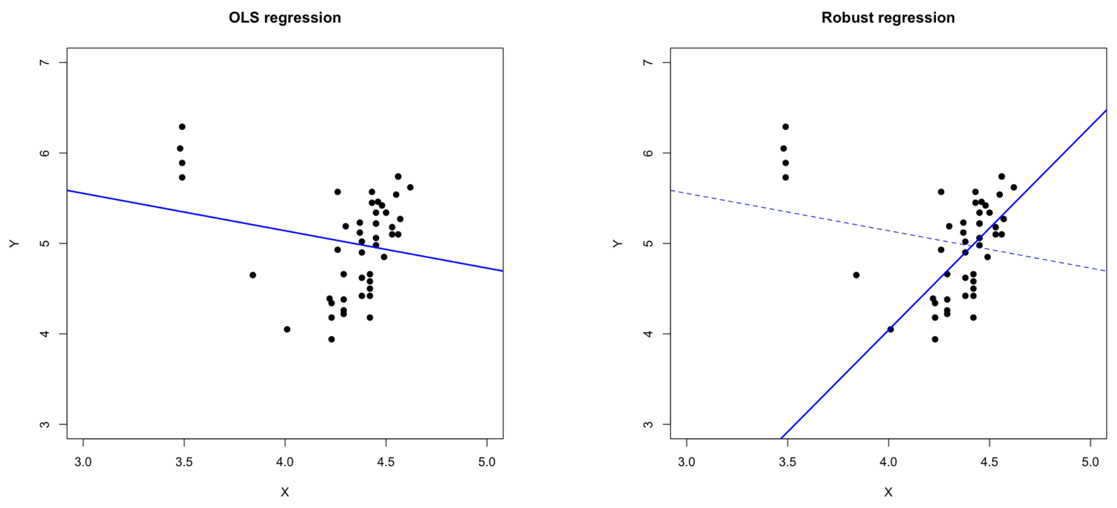


Figure 3.3: Comparison between OLS regression and Robust regression, Appendix B.1

3.3.2 M-estimators

M-estimators are a generalizations of a Maximum Likelihood estimator [22]. They attempt to reduce the influence of outliers by replacing the squared residuals in OLS by another function of the residuals. Squaring the error terms is not the only way of dealing with them. It is possible to define any function $H(\epsilon)$, that we can use to minimize the total cost of the regression model. These functions need to have the following properties [22]:

- Always non-negative $H(\epsilon_i) \geq 0$
- Equal to zero when argument is zero $H(0) = 0$
- Symmetric $H(-\epsilon_i) = H(\epsilon_i)$
- Monotonic $|\epsilon_i| > |\epsilon_j| \Rightarrow H(\epsilon_i) > H(\epsilon_j)$
- Continuous

When the function is monotone, if the residuals are big, the function will also be big and we will penalize these residuals more than others. The general loss function of residuals can be expressed as:

$$S = \sum_{n=1}^{n} H(\epsilon_i) \tag{3.10}$$

When minimizing ordinary least square, the derivative of the function S with respect of the k-th regression parameter β_k is:

$$\frac{\partial \mathrm{S}}{\partial \beta_{\mathrm{k}}} = \sum_{\mathrm{i}=1}^{\mathrm{n}} \epsilon_{\mathrm{i}} \mathrm{x}_{\mathrm{ki}} \tag{3.11}$$

but for the general function of residuals we instead have to derive H with respect to the residual, multiplied by the derivative of the residual with respect of the parameter, which gives the following formulation:

$$\frac{\partial \mathrm{S}}{\partial \beta_{\mathrm{k}}} = \sum_{\mathrm{i}=1}^{\mathrm{n}} \frac{\partial \mathrm{H}}{\partial \epsilon_{\mathrm{i}}} \mathrm{x}_{\mathrm{ki}} \tag{3.12}$$

So depending on the function H(ϵ) we will just need to compute the derivative of the function, and apply an iterative process to determine the $\hat{\beta}$ coefficients. Let us define a function Weight:

$$\mathrm{w}_{\mathrm{i}} = \frac{1}{\epsilon_{\mathrm{i}}} \frac{\partial \mathrm{H}}{\partial \epsilon_{\mathrm{i}}} \tag{3.13}$$

giving that $\frac{\partial H}{\partial \epsilon_i} = w_i \epsilon_i$ then we will obtain:

$$\sum_{\mathrm{n}=1}^{\mathrm{n}} \frac{\partial \mathrm{H}}{\partial \epsilon_{\mathrm{i}}} \mathrm{x}_{\mathrm{ki}} = \sum_{\mathrm{n}=1}^{\mathrm{n}} \mathrm{w}_{\mathrm{i}} \epsilon_{\mathrm{i}} \mathrm{x}_{\mathrm{ki}} \tag{3.14}$$

and this form can be solved with simple algebra. This means that we are doing a linear regression using weights (w_i). At this point the process called IRLS (iteratively reweighted Least Squares) method is used to obtain the final weights.

3.3.3 Huber loss

Huber extended the idea of using M-estimators for solution of regression problems through minimization of a smooth, symmetrical and monotonic function of residuals [22]. The Huber function tries to get the best from the LS (least square estimation) and the LAD (least absolute deviation) [49] by adopting the following formulation:

$$H(\epsilon) = \begin{cases} \epsilon^2/2 & for \quad |\epsilon| \le k \\ k|\epsilon - k^2/2| & for \quad |\epsilon| > k \end{cases} \tag{3.15}$$

usually with $k = 1.345\sigma$. It is evident how this function has a quadratic form when $|\epsilon_i|$ are small, while it assumes a linear form when $|\epsilon_i|$ increases. This strategy decreases the impact of big residuals in the regression.

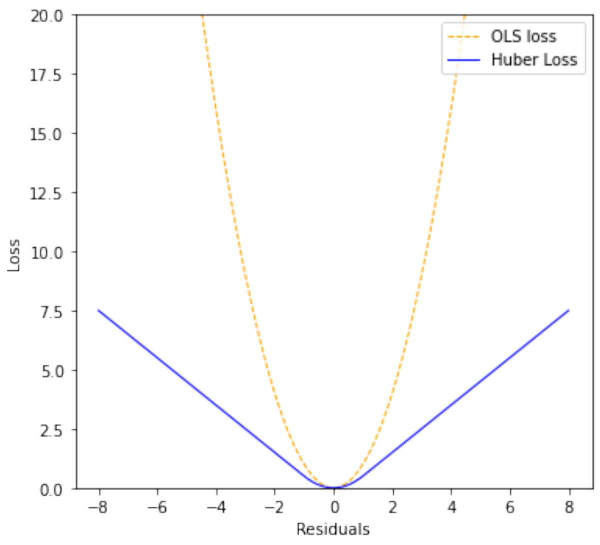


Figure 3.4: Huber and L2 loss functions compared, Appendix B.2

As we can see from the graphics in fig.3.4 the Huber Loss function shows a lower value for residuals that are far from the mean, instead the OLS function shows a rapidly increasing loss function value for the same increase of distance in the residuals.

Chapter 4

Neural networks and optimisation

Neural networks are made of a sequence of multiple layers of interconnected nodes, each one connected to the preceding one. This progression allows the network to refine its prediction with each subsequent layer. This passage, from input to output is called forward propagation. The input and output layers of a deep neural network are called "visible" layers, while all layers in between are called "hidden" layers. Another important process performed by these networks is called backpropagation: once the computation has reached the output layer, the error in predictions is computed and then the weights and biases are adjusted by moving backwards through the layers [27]. Together, forward propagation and backpropagation allow a neural network to be trained. In this chapter we will focus on giving an overview of these two important processes.

4.1 Backpropagation

4.1.1 Framework

The process that we will describe is non-trivial, therefore we will need to outline some details before talking about forward and backward propagation. Let us define the mathematical elements and the operation that will be involved in the discussion. In particular we will define all the elements on the basis of the framework illustrated in fig. 4.1, which represent the three basic elements to implement a neural network: the input, hidden, and output layers [7]. It is important to remark that the notation for the subscripts plays a fundamental role in the correct understanding of the functioning. In particular the subscripts (k,j) refers to the elements of the preceding and the succeeding layers respectively. This will help us in understanding the following equations.

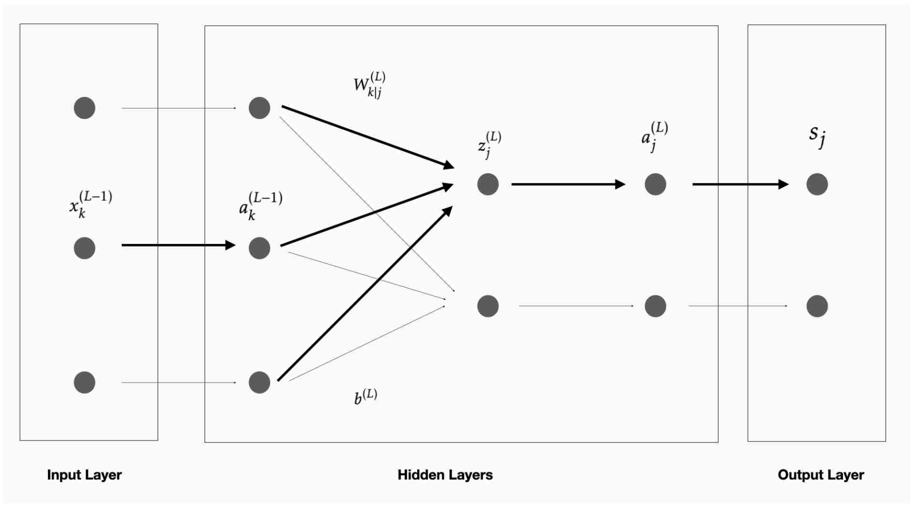


Figure 4.1: Neural network fundamental scheme

Input layer

In the case of the initial input layer, the input information is the data coming from an instance of the training data set (e.g. n-dimensional). Each data instance, will be treated as a vector X of n-features, associated with a label y. The equation representing the input of the framework is:

$$x_k^{(1)} = a_k^{(1)}, \quad k = 1, ..., n \tag{4.1}$$

where k is the position of the element of X in the input layer, and the superscript $^{(1)}$ indicates that we are referring to the input layer as the first layer of the network. All consecutive layers will have a superscript $L = 1, ..., Z$, where Z is the total number of the layers in the network. Also, a_k is the result of the node after the application of an activation function. Only in the case of the input layer, the activation function multiplies the input by 1.

Hidden layers

Let us define the linear combination z_j, that goes into the j-th neuron of the next layer (L), which is the linear combination of the inputs coming from a preceding layer $(L-1)$:

$$z_j^{(L)} = W_{k|j}^{(L)} \cdot a_k^{(L-1)} + b^{(L)} \tag{4.2}$$

where $W_{k|j}^{(L)}$ is the weights' matrix and k represent the position of the input in the $(L-1)$ layer and j the position of the neuron in the layer that receives the input. It is important to highlight that the elements of the weights' matrix, get the subscript jk, indicating in the first position the element that receives the input and in the second position the element that gives the input. This is necessary to maintain coherence with how the equations are expressed [31]. The term $b^{(L)}$ is the bias term (at layer L).
The equation for the activation functions are defined as a non-linear transformation applied to the linear combination $z_j^{(L)}$:

$$a_j^{(L)} = \sigma(z_j^{(L)}) \tag{4.3}$$

These two operations (linear combination and activation function are applied one after the other at the node's input data) are the two basic operations that happen at the hidden layers level.

Weights's matrix

Each element of the weights' matrix is expressed as:

$$w_{jk}^{(L)} \tag{4.4}$$

where j is the number of the neuron in the (L) layer, while k is the number of the neuron in $(L-1)$. The matrix form is the following:

$$W^{(L)} = \begin{bmatrix} w_{11}^{(L)} & w_{12}^{(L)} & \dots & w_{1p}^{(L)} \\ w_{21}^{(L)} & w_{22}^{(L)} & \dots & w_{2p}^{(L)} \\ \dots & \dots & \dots & \dots \\ w_{n1}^{(L)} & w_{n2}^{(L)} & \dots & w_{np}^{(L)} \end{bmatrix} \tag{4.5}$$

where each row represents the set of weights connecting the j-th neuron in the layer L with the neurons from $k = 1, \dots, p$, with p being the number of the neurons in layer $(L-1)$. Each linear combination $z_j^{(L)}$ requires to compute a multiplication between two vectors:

$$z_j^{(L)} = W_{k|j}^{L} \cdot a_k^{(L-1)} + b^{(L)} \tag{4.6}$$

where $W_{k|j}^{L}$ and $a_k^{(L-1)}$ are respectively one line and one column vector:

$$z_j^{(L)} = \begin{bmatrix} w_{j1}^{(L)} & w_{j2}^{(L)} & \dots & w_{jp}^{(L)} \end{bmatrix} \cdot \begin{bmatrix} a_1^{(L-1)} \\ a_2^{(L-1)} \\ \vdots \\ a_p^{(L-1)} \end{bmatrix} + b^{(L)} \tag{4.7}$$

Each $z_j^{(L)}$ represents the linear combination coming into the j-th neuron in the (L)-th layer.

Forward pass

From the input layer to the output layer, the network performs the operations that we just described.

- Once the input is given, the network will obtain the linear combinations $z^{(L)}$ at each neuron in the succeeding layer (L), as described in equation 4.7.
- The chosen activation function $\sigma()$ is then applied to $z^{(L)}$ in each neuron.
- Once it has reached the output layer, the vector S will be the vector of the predicted scores, where s_j is one of its elements.

- The error will then be computed in the chosen loss function by using the output vector S and the ground truth vector$Y$.
- Let us consider we are computing a simple MSE (mean square error) [18] as the cost function:

$$C_0 = \sum_{j=1}^{N} (s_j^{(L)} - y_j)^2 \tag{4.8}$$

where N is the number of elements in the score vector S. The cost will be computed on each training example, and then averaged on the dimension of the training data set, giving the final average cost of the network for the current set of parameters:

$$C = \frac{1}{p} \sum_{i=1}^{p} C_k \tag{4.9}$$

where k is index of the k-th training example, and p the size of the training set.

The cost value is therefore obtained with a forward computation from the input layer to the output layer. We refer to the sequence of operations as "forward pass".

Chain rule

We now want to minimize the cost function. To do this we will need to compute the derivative of the cost function with respect to the input parameters of the preceding layer, the weight $W_{k|j}^{(L)}$, the bias $b^{(L)}$ and the activation function $a_k^{(L-1)}$ [19]. By looking at the graph in fig. 4.2 we can understand that a change in any of the cited inputs, will generate a change in $z_j^{(L)}$, and this one will affect $a^{(L)}$, and this one will influence the final cost C_0.

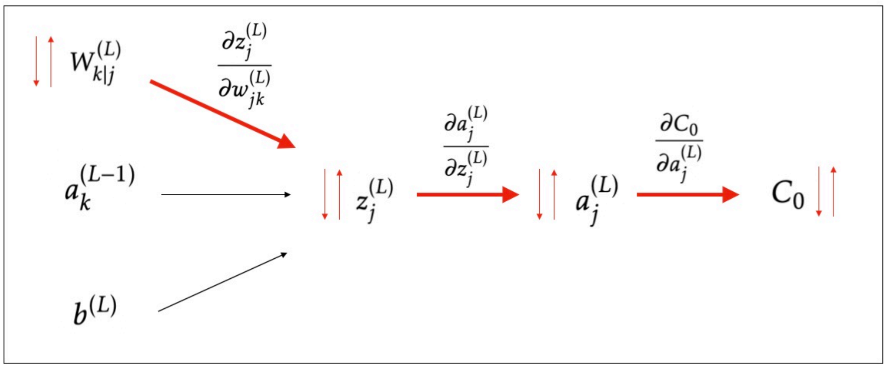

$W_{k|j}^{(L)}$
$\frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}}$
$a_k^{(L-1)}$
$b^{(L)}$
$z_j^{(L)}$
$\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}}$
$a_j^{(L)}$
$\frac{\partial C_0}{\partial a_j^{(L)}}$
C_0

Figure 4.2: Chain rule representation

We are interested in obtaining the derivative of the cost function $\frac{\partial C_0}{\partial W_{k|j}^{(L)}}$, and by recalling the chain rule we can obtain the wanted equation by multiplying the partial derivatives at each stage along the neural path that connects the output to the parameter that causes the initial change, $W_{k|j}^{(L)}$ in this case:

$$\frac{\partial C_0}{\partial w_{jk}^{(L)}} = \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} \cdot \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \cdot \frac{\partial C_0}{\partial a_j^{(L)}} \tag{4.10}$$

We can explicitly differentiate each term of this equation, as we know the equation of each one of them, obtaining what follows:

$$\frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} = (w_{jk}^{(L)} \cdot a_k^{(L-1)} + b_j^{(L)})' = a_k^{(L-1)} \tag{4.11}$$

$$\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} = (\sigma(z_j^{(L)}))' = \sigma'(z_j^{(L)}) \tag{4.12}$$

$$\frac{\partial C_0}{\partial a_j^{(L)}} = (a_j^{(L)} - y_j)^2 = 2 \cdot (a_j^{(L)} - y_j) \tag{4.13}$$

The final equation is:

$$\frac{\partial C_0}{\partial w_{jk}^{(L)}} = a_k^{(L-1)} \cdot \sigma'(z_j^{(L)}) \cdot 2 \cdot (a_j^{(L)} - y_j) \tag{4.14}$$

This same logic can be applied to the other two terms ($a_k^{(L-1)}$ and $b^{(L)}$), that influence the cost C_0, as represented in the graphics below:

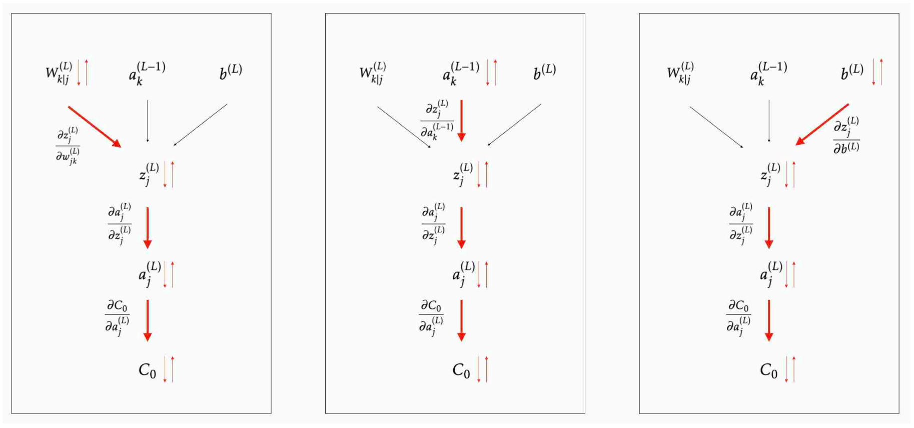

$W_{k|j}^{(L)}$
$a_k^{(L-1)}$
$b^{(L)}$
$\frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}}$
$\frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}}$
$\frac{\partial z_j^{(L)}}{\partial b^{(L)}}$
$z_j^{(L)}$
$\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}}$
$a_j^{(L)}$
$\frac{\partial C_0}{\partial a_j^{(L)}}$
C_0

Figure 4.3: Chain rule representation for other parameters

Even if $a_k^{(L-1)}$ does not influence directly the cost function, it is useful to keep track of it, because now we can iterate the same chain rule idea to see how much the cost function is sensitive to previous weights and biases. To compute the influence of the bias term $b^{(L)}$ on the cost C_0, it is sufficient to substitute the first term $\frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}}$ with the term $\frac{\partial z_j^{(L)}}{\partial b^{(L)}}$:

$$\frac{\partial C_0}{\partial b^{(L)}} = \frac{\partial z_j^{(L)}}{\partial b^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z_j^{(L)}} \cdot \frac{\partial C_0}{\partial a^{(L)}} \tag{4.15}$$

By doing so, we computed the dependence from the bias term, where before the influence was determined by the weights. If we want to track the influence of the activation function we will write as follows:

$$\frac{\partial C_0}{\partial a_k^{(L-1)}} = \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}} \cdot \frac{\partial a^{(L)}}{\partial z_j^{(L)}} \cdot \frac{\partial C_0}{\partial a^{(L)}} \tag{4.16}$$

The equation in 4.16 is where the whole idea of the "backward propagation" or "backpropagation" comes from: the whole chain repeats again in the layer $(L-2)$:

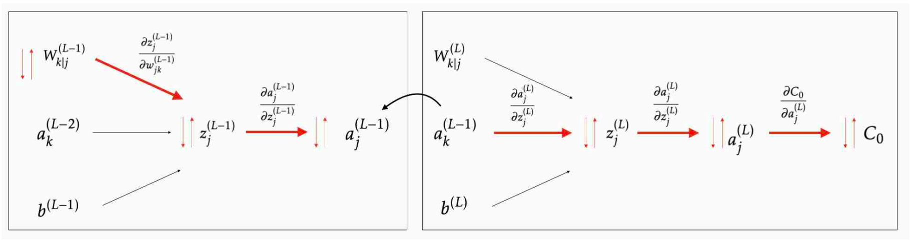

$W_{k|j}^{(L-1)}$
$\frac{\partial z_j^{(L-1)}}{\partial w_{jk}^{(L-1)}}$
$a_k^{(L-2)}$
$z_j^{(L-1)}$
$\frac{\partial a_j^{(L-1)}}{\partial z_j^{(L-1)}}$
$a_j^{(L-1)}$
$b^{(L-1)}$
$W_{k|j}^{(L)}$
$a_k^{(L-1)}$
$\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}}$
$z_j^{(L)}$
$\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}}$
$a_j^{(L)}$
$\frac{\partial C_0}{\partial a_j^{(L)}}$
C_0
$b^{(L)}$

Figure 4.4: Chain rule extension

Once the derivatives with respect to each weight and each bias in the whole network have been computed, we can obtain the gradient of the cost function with respect to each one of them, with the chain rule. The gradient will be used to update the weights and biases with the gradient descent rule [43]. The gradient will be the derivative of the cost function with respect of all weights and biases of the network:

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w_{11}^{(1)}} \\ \vdots \\ \frac{\partial C}{\partial b^{(L)}} \end{bmatrix} \tag{4.17}$$

4.2 Optimisation methods

Optimization methods can be divided into two main categories [70]: first-order methods, which involve gradient descent, stochastic gradient descent, Nesterov Accelerated Gradient Descent, Adaptive Learning Rate Methods (like AdaGrad [25]), Variance Reduction Methods, Momentum methods like ADAM [39], and second-order optimization methods, like Conjugate Gradient and Newton's method, which can be used for addressing the highly non-linear problems. First-order methods, are widely used for their simplicity in implementation. The difficulty in high-order methods lies in the operation and storage of the inverse matrix of the Hessian matrix [70]. Let us consider two of the most relevant methods.

4.2.1 Gradient descent

The gradient descent method is the most common optimization method [26]. The idea of gradient descent, is that variables are updated in an iterative way following the opposite direction of the objective function's gradient. In the formulation of the gradient descent a hyperparameter is defined, called "learning rate" (η), which determines the step size at every iteration, and thus influences the number of iterations to reach the optimal value. Let us consider the following cost function:

$$h(\theta) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(y_i, f_\theta(x_i)) \tag{4.18}$$

where $f_\theta(x_i) = \sum_{j=1}^{D} \theta_j x_j$, is a linear combination of n-features, y_i is the ground truth of the i-th training example, and $\mathcal{L}(y_i, f(x_i))$ is the loss function. The gradient of eq. 4.18 is defined as the vector of the partial derivatives of the function at the point θ:

$$\nabla h(\theta) = \begin{bmatrix} \frac{\partial h(\theta)}{\partial \theta_1} \\ \frac{\partial h(\theta)}{\partial \theta_2} \\ \vdots \\ \frac{\partial h(\theta)}{\partial \theta_n} \end{bmatrix} \tag{4.19}$$

The gradient descent updating rule is defined as:

$$\theta_{k+1} = \theta_k - \eta \nabla h(\theta_k) \tag{4.20}$$

where η is the learning rate parameter, θ_k are the parameters at step k. The gradient descent method is simple to implement. The solution is the global optimal point when the objective function is convex.

4.2.2 Stochastic gradient descent

The idea of stochastic gradient descent is computing the cost function using only a sample randomly chosen from the training set, and updating the parameters with its gradient, instead of calculating the exact value of the gradient for all training examples at every iteration [70]. A typical training object consists in fact of the sum of individual losses for each training case such that:

$$h(\theta) = \frac{1}{m}\sum_{i=1}^{m} h_i(\theta) \tag{4.21}$$

If m is very big, the computation of the GD becomes excessively expensive. We can reformulate the GD as the sum of the individual gradients of each cost function:

$$\nabla h(\theta) = \frac{1}{m}\sum_{i=1}^{m} \nabla h_i(\theta) \tag{4.22}$$

The general idea is to randomly sub-sample a "min-batch" of training cases $S \subset 1, 2, ..., m$ of size $b << m$ and estimate the Gradient as:

$$\tilde{\nabla} h(\theta) = \frac{1}{b} \sum_{i \in S} \nabla h_i(\theta) \tag{4.23}$$

The Stochastic Gradient Descent (SGD) then replaces the classic GD with its mini batch estimate:

$$\theta_{k+1} = \theta_k - \eta \tilde{\nabla} h(\theta_k) \tag{4.24}$$

SGD increases the overall optimization efficiency at the expense of more iterations, but the increased iteration number is insignificant compared with the high computation complexity caused by large numbers of samples.

Chapter 5

Deep Learning Architectures

5.1 Deep learning

Deep learning is a subset of machine learning, which is a subset of artificial intelligence. Artificial intelligence is the theory and development of computer systems able to perform tasks normally requiring human intelligence, such as visual perception, speech recognition, decision-making, and translation between languages. The earliest predecessors of modern deep learning were simple linear models motivated from a neuroscientific perspective. [29].

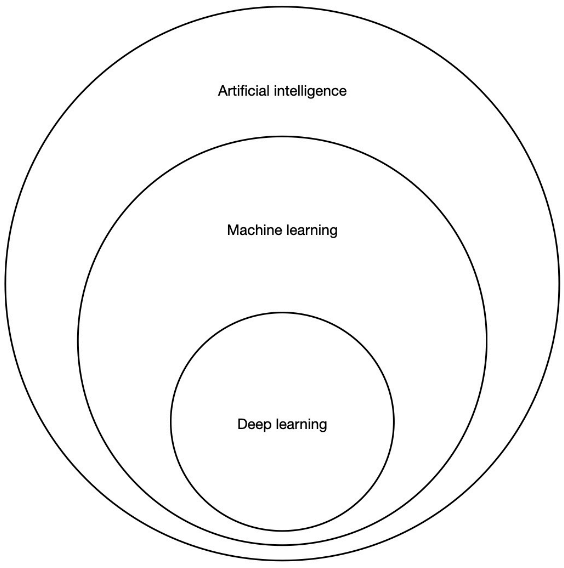


Figure 5.1: Venn diagram of deep learning methods

In deep learning networks concepts are hierarchically organized, enabling the networks to learn complex concepts by building them out from simpler ones [29]. The simplest representative of deep learning networks is the Multilayer Perceptron (MLP) network [61], which performs a mapping of some set of input values to output values via a weighted function. Other examples are Recurrent Neural Networks (RNNs) [65], Convolutional Neural Netowkrs (CNNs) [48], Self-organizing maps (SOMs) [41] to cite some of the most relevant. However, deep learning can be safely regarded as the study of models that involve a greater amount of composition of either learned functions or learned concepts than traditional machine learning does [54]. By quickly analyzing some of the most relevant networks, we would like to outline a general understanding of how deep learning systems work. We will see that they are based on the organization of three fundamental parts: input layer, hidden layers and output layers. These three basic units are organized in various types of cells, that make the particular network adapt to a certain task.

5.1.1 Multilayer perceptrons (MLPs)

A Multilayer Perceptron (MLP) is constituted of at least three layers of nodes: an input layer, a hidden layer and an output layer. All the layers are fully connected, meaning that each output from a layer's nodes is connected to each neuron of the succeeding layer. This class of networks is also a feed-forward type of network. Each node can be interpreted as a fundamental unit that can receive a set of inputs and combine them in a weighted sum. [15]. An example of MLP is represented in fig. 5.2.

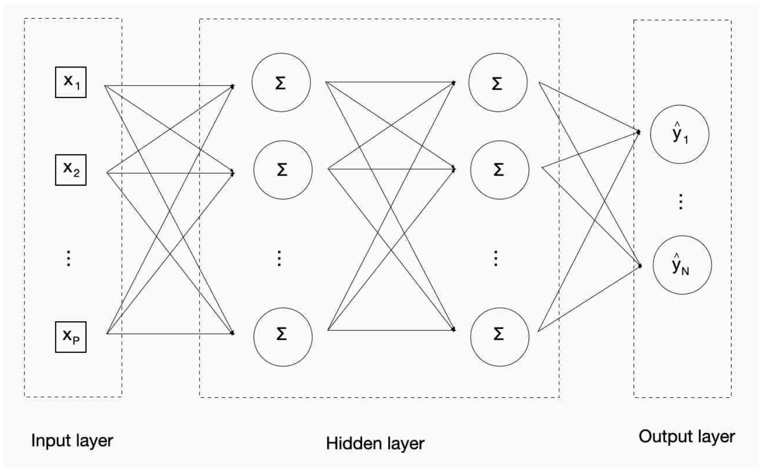


Figure 5.2: Scheme of a Multilayer perceptron

5.1.2 Recurrent neural networks (RNNs)

Recurrent neural networks (RNNs) are type of neural networks where the link between nodes create a cycle, in such a way that the output from some nodes is able to affect the subsequent input at the same nodes [79], at the following timestep. These deep learning algorithms are commonly used to process temporal sequences, such as language translation, natural language processing (nlp) and speech recognition [20]. We can represent Recurrent neural networks with the following "folded" and "unfolded" schemes, that illustrate the cyclical nature of the network and also its time evolving characteristic:

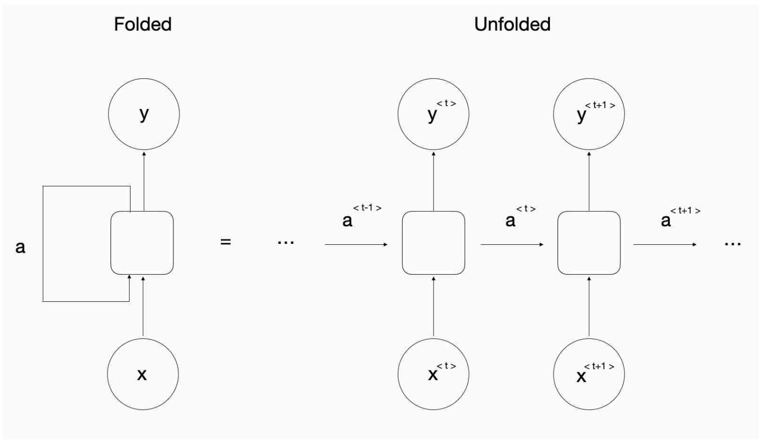


Figure 5.3: Scheme of a Recurrent neural network

Standard feed-forward networks have different weights across each layer, while RNNs share the same weights within each layer of the network [20]. Following a more detailed representation of the internal logic of a recurrent neural network that describes the cyclical nature of the network [47]:

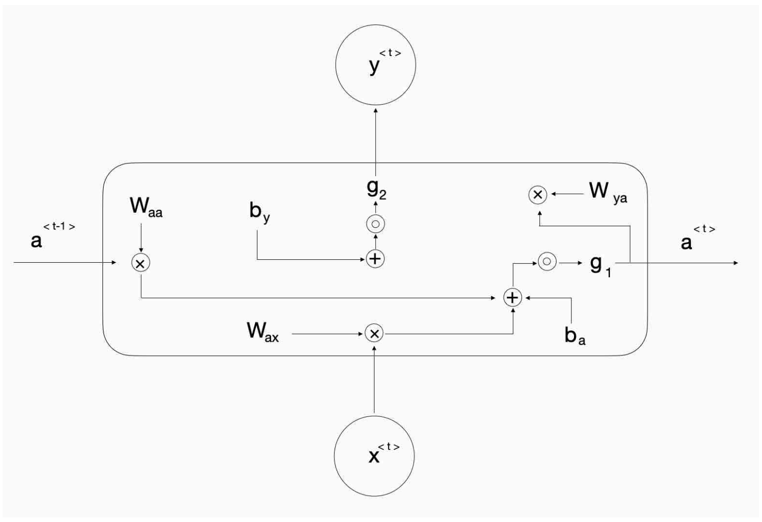


Figure 5.4: Details of a Recurrent neural network [47]

The representation in fig. 5.4 describes how for each timestep "t" the activation function $a^{<t>}$ and the output $y^{<t>}$ are expressed as follows (from [47]):

$$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a) \tag{5.1}$$

$$y^{<t>} = g_2(W_{ya}a^{<t>} + b_y) \tag{5.2}$$

where W_{ax}, W_{aa}, W_{ya}, b_a, b_y are coefficients that are shared temporally and g_1 and g_2 are activation functions [47].

5.1.3 Autoencoders

Autoencoders belong to the unsupervised methods, and their scope is to learn encoded feature representations of the training set. The first networks architecture for thinking about autoencoders is an architecture that constrains the hidden layers to have a limited the number of nodes, usually smaller than the number of nodes of the input. This limitation in the size of information that can pass along the network is the main characteristic of Autoencoders, and it is referred to as "Bottleneck" [36]. The bottleneck contains the most relevant attributes to reconstruct the input in the form of an "encoded" space. The final aim of an autoencoder is to learn a function that can describe the encoded features to rebuild the inputs. The chart in fig. 5.5 describes the three main parts of an autoencoder [9]:

- Latent space (Bottleneck): it can be thought of as the central (hidden) layer of the network, containing a compressed representation of the training set. For this reason it is the main feature of an autoencoder. In order for the compression to happen, the size of the number of nodes are restricted at an order of magnitude smaller than the input data.

- Encoder: it comprises the input layer and the hidden layers on the left side of the bottleneck layer(s). From this side of the network the "compression" of the training set features happens, and end up in the bottleneck layers which contain the most compressed representation of the input.

- Decoder: it starts at the right side of the bottleneck and ends with the output layer. In this module the “decompression” of the encoded knowledge happens, until the reconstruction of the output reaches the same size of the input. The goal of the reconstruction is to have an output as similar as possible to the original one.

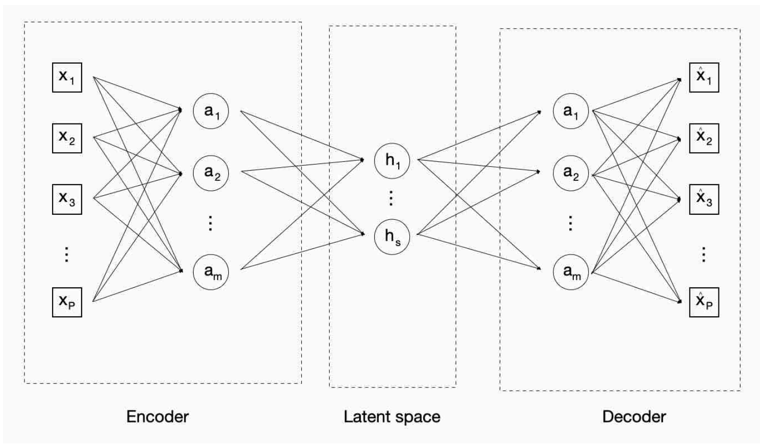


Figure 5.5: Autoencoder conceptual representation

More formally, let us suppose that we have a set of training set containing D-examples $x^{(i)}$ where $x^{(i)} \in \mathbb{R}^n$. The network tries to learn a function $h(x^{(i)}) \approx x^{(i)}$ [6]. When fed with unlabeled inputs x and framed to output a representation $\tilde{x}$, this network can be trained by minimization of the "reconstruction" error, $L(x, \tilde{x})$, which measures how different the original input and output are [36].

5.1.4 Generative Adversarial Networks (GANs)

Generative Adversarial Networks (GANs) are "generative" [78] models trained via an "adversarial" process. It is composed by a generative network that creates a distribution similar to the training distribution, and a discriminative network that tries to distinguish between real training examples and fake training examples. The adversarial process is an "implicit" method, as the network improves its performances by generating more "realistic" inputs. The training involves two "players": a Generator and a Discriminator [30]. The Generator receives as input a simple distribution, and produces an output that is the sampling from that input distribution. A subsequent Discriminator will judge if the sample distribution is similar to the real data distribution. If not the generator will produce a second output that tries to reduce the difference with the real data.

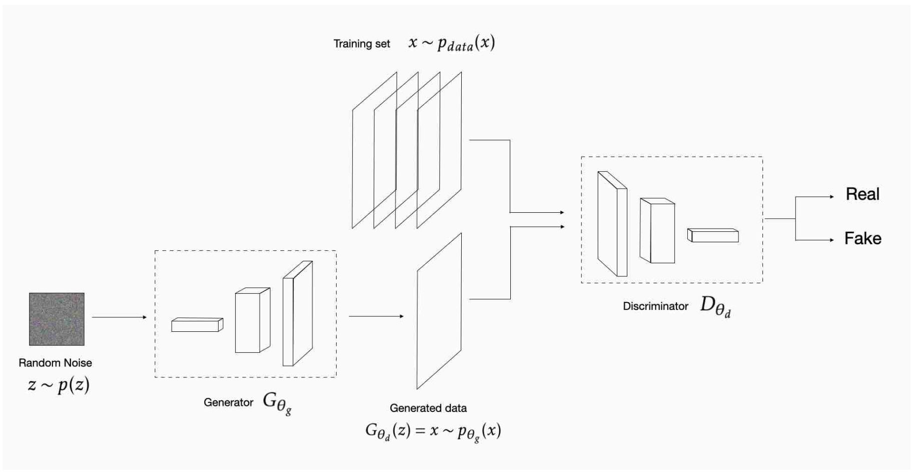

Training set $x \sim p_{data}(x)$
Real
Fake
Discriminator D_{θ_d}
Random Noise
$z \sim p(z)$
Generator G_{θ_g}
Generated data
$G_{\theta_d}(z) = x \sim p_{\theta_g}(x)$

Figure 5.6: Conceptual GAN architecture

The goal is to train the Generator and the Discriminator jointly in a adversarial training [30]:

$$\min_{\theta_g} \max_{\theta_d} [\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p_z} \log(1 - D_{\theta_d}(G_{\theta_d}(z)))] \tag{5.3}$$

The term $D_{\theta_d}(x)$ represents the discriminator output for real data, while $D_{\theta_d}(G_{\theta_d}(z)))$ the discriminator output for fake data $G(z)$s. The discriminator θ_d wants to maximise the objective function such that $D_{\theta_d}(x)$ is close to 1 (real) and $D_{\theta_d}(G_{\theta_d}(z)))$ is close to 0 (fake). At the opposite, the generator (θ_g) wants to minimise the objective function such that $D_{\theta_d}(G_{\theta_d}(z)))$ is close to 1. The strategy implemented is usually an alternating strategy such that [30]:

- Gradient Ascent on the discriminator

$$\max_{\theta_d} [\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p_z} \log(1 - D_{\theta_d}(G_{\theta_d}(z)))] \tag{5.4}$$

- Gradient Descent on the generator

$$\min_{\theta_g} \mathbb{E}_{z \sim p_z} \log(1 - D_{\theta_d}(G_{\theta_d}(z)))] \tag{5.5}$$

The Generative Adversarial Networks (GANs) architecture was first described in the 2014 paper by Ian Goodfellow, et al. titled “Generative Adversarial Networks”. Since then many variations of the GAN have been produced, improving these networks in tasks such as better training and generation.

5.1.5 More deep learning architectures

We have just given some relevant examples of the fundamentals of deep learning methods, but those briefly described in the previous paragraphs are only a small subset of a much greater extended panorama of deep architectures. Other relevant methods regard architecture such as: Convolutional neural networks, proposed for the first time by Yann LeCun et al. in 1989 [48], Self-organizing maps, introduced by T. Kohonen in 1982 [42], Boltzmann Machines introduced by Ackley, Hinton and Sejnowski in 1985 [2] and Reinforcement learning introduced by Rich Sutton in the 1980s [10]. For the aim of our analysis we will now give a detailed review of one of the most relevant networks that is fundamental for many tasks involving image processing: convolutional neural networks.

5.2 Convolutional neural networks (CNNs)

Convolutional neural networks characterize themselves from other deep learning algorithms because of their superior performance in the tasks of image and audio analysis. They are in fact particularly suited for classifying images, because of the hierarchical organization that they can implement, making the task of recognizing an image feasible for a machine. The computer vision fields is composed by different problems that involve images: image recognition, localization, detection, semantic segmentation and annotation. Among those, image recognition can be considered the baseline problem, as it forms the basis for the rest of the more structured computer vision problems. Convolutional neural networks has been deployed since the 1990s, where a research group (Y. LeCun, L. Bottou, Y. Bengio) at ATT developed a convolutional network for reading checks [29]. But in the last years, starting from 2012, CNNs have seen a significant growth in the range of possible application, such as high resolution images, due to the rise of efficient graphics processing units (GPUs) and the availability of large-scale data sets.[13]. They are the composition of three main types of layers:

- Convolutional layer
- Pooling layer
- Fully-connected layer

The combination of these three layers, generate a network that performs an analysis that can be divided in to two main tasks:

- Feature detection
- Classification

In particular, unlike a regular Neural Network, the layers of a convolutional network (ConvNet) have neurons arranged in 3 dimensions: width, height and depth. The figure below shows a conceptual CNN architecture. We can observe one convolutional layer, followed by a pooling operation [69]. After the last pooling, the feature maps get flattened and are used as the input of the fully-connected layer that will perform the classification task.

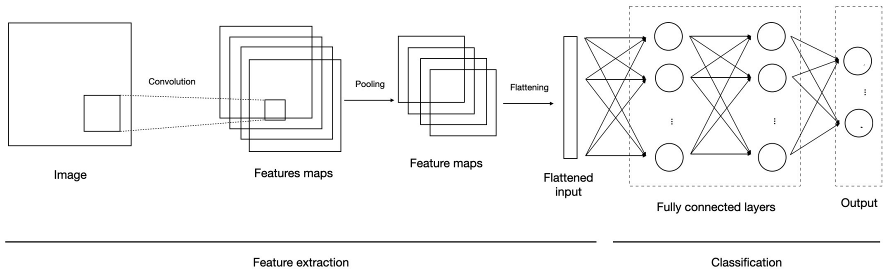


Figure 5.7: Convolutional neural network architecture

The convolutional layers and the pooling layers are particularly useful in the feature detection task, and can be follow by further additional convolutional and pooling layers, while the fully-connected layer is always the last part of the architecture, and is deputed to perform the classification task. These networks have an important propriety: hierarchy. After each layer, the CNN becomes able to identify more complex portions of the image, connecting together the features that it has been able to recognize in the earliest stages [69].

5.2.1 Convolutional layer

In the convolutional layer one basic operation is performed, which is the dot product. This operation is made by sliding between the kernel over the section of the input to which it is aligned. We can consider the kernel and the section of the input as two separate matrices A and B of the same size WxL, on to which the dot product is defined as:

$$A \cdot B = \sum_{i=1}^{n} a_i b_i \tag{5.6}$$

where $n = WxL$ is the number of elements of each matrix. The kernel matrix is also referred to as "filter" matrix. As already said the filter matrix slides over the input matrix, changing its position of a certain number of pixels at each iteration. The number of pixels by which the filter moves across the input at each operation is called Stride. Usually the filter size is a $3x3$ matrix, and the stride is commonly equal to 1. Following a representation of the principle of the convolution:

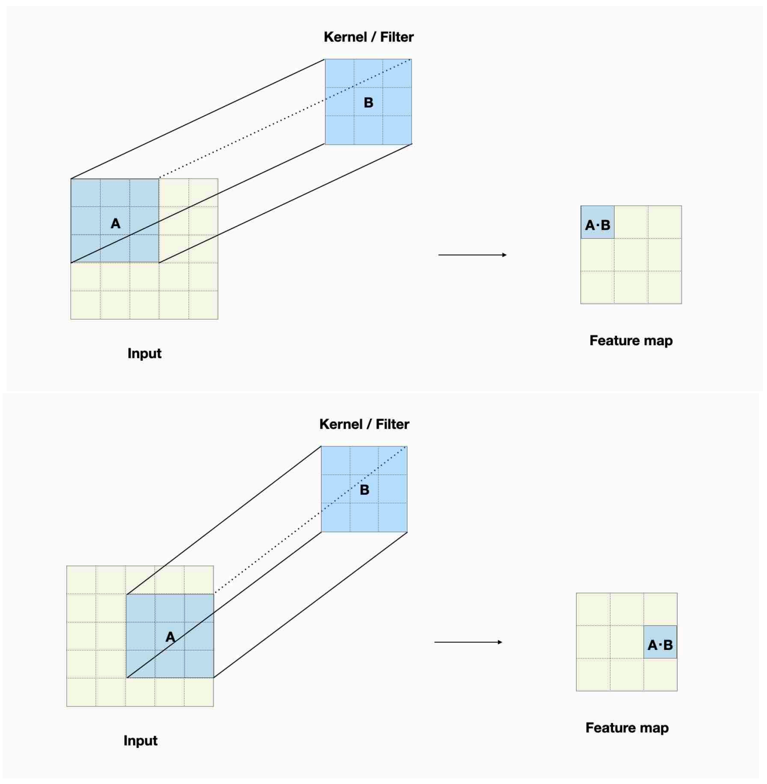


Figure 5.8: Dot product

We should consider that the input also has a depth dimension. That is because usually images have more than one channel, like in the RGB case, where the depth is equal to 3. We can refer to the input as the input volume, with dimensions WxHxC. The kernel will have a smaller spatial (2D) dimension, but will always have the same depth as the input volume. The product of these two tensors will always be 2D matrix, called feature map. Also, each convolutional layer can have k kernels, with $k \geq 1$, therefore each of them, multiplied by the input volume, will produce a different feature map of depth equal to 1. The sum of the feature maps in the output is called "output volume". [69].

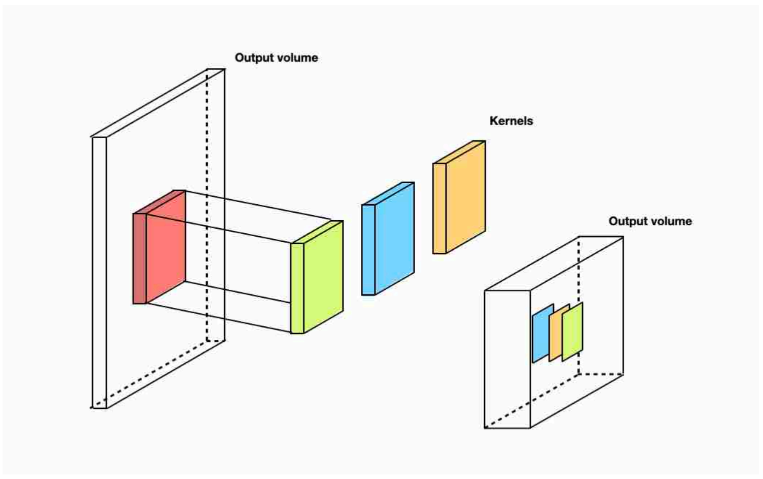


Figure 5.9: Output volume representation

Spatial connectivity

As we can observe from fig. 5.8, the neurons are connected only to a certain region of the input volume. The spatial extent of the region is called "receptive field", and has the same size of the filter [69]. We can also observe that the depth extension of the output volume is equal to the number of filters used in the convolution layer, in this example 3. The spatial (2D) size of the output volume is instead affected by three hyperparameters, that it is possible to choose in the design phase of the network. These hyperparameters are:

- Depth
- Stride
- Zero-padding

By keeping in consideration the second and third hyperparameters, we can define the output's spatial size in the horizontal and vertical directions [69]:

$$\mathrm{O} = \frac{\mathrm{I} - \mathrm{F} + \mathrm{P}_{\mathrm{left}} + \mathrm{P}_{\mathrm{right}}}{\mathrm{S}} + 1 \tag{5.7}$$

$$\mathrm{Q} = \frac{\mathrm{Y} - \mathrm{F} + \mathrm{P}_{\mathrm{up}} + \mathrm{P}_{\mathrm{down}}}{\mathrm{S}} + 1 \tag{5.8}$$

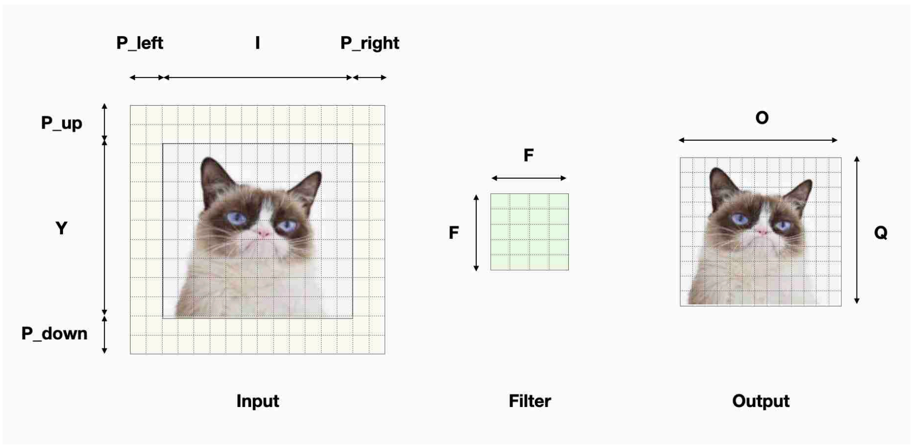


Figure 5.10: Spatial size

We can also define the value (F) of each neuron before the activation, by considering a 2D input with the following equation[23]:

$$F = \sum_{i=1}^{4} \sum_{j=1}^{4} w_{ij} x_{i+p,j+q} + b \tag{5.9}$$

where (p, q) is the position of the hidden neuron in the hidden layer. When the filter multiplies a 3D input, we will compute equation 5.9 for each channel, and then add them together.

Number of parameters

The number of parameters are the values associated with each filter. If a filter has size (n x n), and the input has depth m, it means that each filter will have (n x n x m) parameters. The total number of parameters at each layer, is the sum of the total number of parameters from all the filter in the layer k. The final number will be (n x n x m x k) [32].

5.2.2 Activation function

The output obtained after the convolution operation, is commonly passed through an activation function. The most common function in convolutional layers is the ReLU (rectified linear unit) function:

$$\text{ReLu}(x) = \max(0, x) \tag{5.10}$$

The activation function has the effect of adding non-linearity into the convolutional neural network. Another function can also be used after a convolutional layer, which is the Sigmoid function:

$$S(x) = \frac{1}{1 + e^{-x}} \tag{5.11}$$

5.2.3 Pooling layer

Pooling is applied after the convolutional layer, but not necessarily after every one. The pooling operation resizes the input to a smaller spatial size, with the aim of reducing the number of total parameters in the network, making the computation faster. The operation is performed by a window that has a certain spatial size (height and width) and a certain stride. The window acts independently on every feature map of the output volume. The most common operation implemented by the pooling window is the Max function, which extracts the maximum value from the window at each stride, but in addition it is possible for the pooling to perform other functions, like Average pooling, which extracts the mean value from the same window [69].

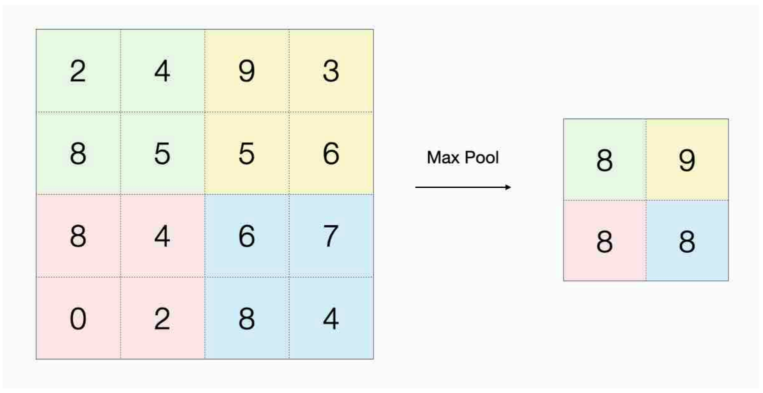


Figure 5.11: Max pooling

Pooling filters are most commonly of size 2x2 with a stride of 2 along both width and height. We can say that in general a pooling operation accepts an input of size x_1 x y_1 x z_1, and given the pooling filter size of A and B, respectively the spatial extent of the filter and the stride, outputs a volume of x_2 x y_2 x z_2 such that [69]:

$$x_2 = \frac{(x_1 - A)}{(B+1)} \tag{5.12}$$

$$y_2 = \frac{(y_1 - A)}{(B+1)} \tag{5.13}$$

$$z_2 = z_1 \tag{5.14}$$

5.2.4 Fully-connected layer

A fully connected layer is a layer where the neurons have a full connection to every input in the preceding layer. As seen in MLPs, a series of fully connected layers are a practical way of learning non-linear combinations from the high-level features coming from the previous convolutional layers, and performing the final classification task. Before the first fully connected layer, the output of the last convolutional layer gets "flattened" into a column vector, meaning that the data becomes a suitable input for the coming MLP network, which operates as we say in the previous paragraph, in a series of training cycles (epochs) composed by a forward and a backward propagation phase [69].

Chapter 6

A comparative review of loss functions

The aim of this thesis is to explore and observe how substituting components in a neural network can affect performance change. In particular we want to study the system from a loss function point of view. We implemented two different types of architecture, a 5-convolutional layer architecture and a 3-dense layer triplet loss adapted type of network, and confronted them on the same image classification task. The data set used has been the MNIST Handwritten Digits, for its reduced size and consequential lower computational power absorption. In this chapter we will plot the result of the training for each different implementation by plotting the accuracy and loss function charts.

Setup The computer setups that have been used for training is a Mackbook Air (M1, 2020), with 16 GB of RAM memory. We implemented the cross entropies losses in Jupyter Notebook with Python 3.9.10, Keras 2.8.0, scikit-learn 1.1.1, tensorflow-macos 2.8.0. The triplet loss has been implemented on Google Colab with Python 3.7.13, Keras 2.8.0, scikit-learn 1.0.2, tensorflow 2.8.2.

Data The MNIST handwritten digits database, is composed of 60,000 training examples and 10,000 test examples of handwrittend digits ranging from 0 to 9. Each instance is a 28 x 28 pixels image on grayscale.

6.1 Network design

In network design we want to define a practical neural network architecture, a learning rule and a training technique that effectively accomplish the task we want to solve [24].
The first aspect to take into consideration is the nature of the data set: in fact, depending on the type of data that are available, we should decide which are the tools and frameworks that best adapt to that kind of data. To give an example, when working with images, we might find out that a framework is better suited for working with vector and another with tensors. As we can see we need to address these issues even before choosing which framework we will use to implement our architecture. Also the availability of data is an equally important aspect: trivial tasks might be solved with already existing ready-to-use databases, but that could not be the case for tasks in which we need to gather some data first.
Next we want to design the network itself: this implies deciding the number of layers and

how many nodes implementing at each layer. Usually architectures are designed with a constant number of nodes for all hidden layers, but it can also be the case to design networks with a decreasing or a variable number of nodes across the layers [66]. Each layer can then be completed by adding an activation function (such as the ReLu) and a Pooling layer. From experience we find out that we do not want these architectures neither to be too big, neither too small. This is why often, 1 to 5 hidden layers will obtain the best performances. This problem is linked to the fact that these many layers, are sufficient to learn all possible features for the size and variance of most of the data set.
Defining the loss function and the optimization method is our next step: the loss function makes explicit what we want to minimize during the training, which implicitly means that we are choosing what the network should learn, or what is its objective function. Usually these functions are grouped in loss function for regression problems and for classification problems, as we described in chapter 2. The optimization methods instead are important to make our network as fast and reliable as possible: they have the task to access the gradients stored after the backpropagation and update the weights in the direction that minimizes the loss. The most used optimization methods in deep learning are the stochastic gradient descent and the Adam method.
Finally we are left to design the training loop: this part is the most complex and the most dependent upon the functions of the framework that we are using. We start by deciding the Batch size and the number of epochs. Also in the design of this phase we can address issues such as Batch normalization, vanishing or exploding gradients, overfitting and weight initialization.

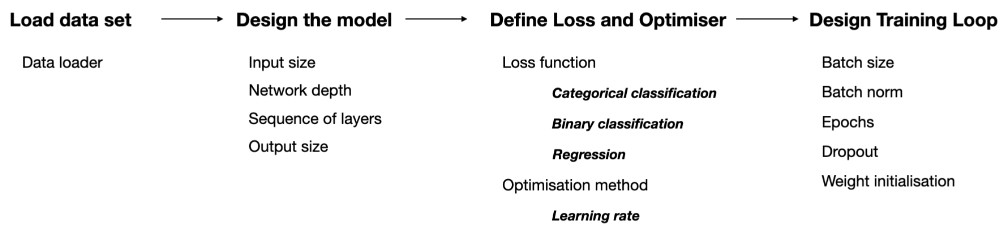


Figure 6.1: Network design steps

6.1.1 Networks used

The creation of a network happens in two fundamental steps. We used Keras as the framework for both models. The first step is to define the layer sequence: we used the "model = Sequential()" method for the first network, and the "model()" class for the second one.

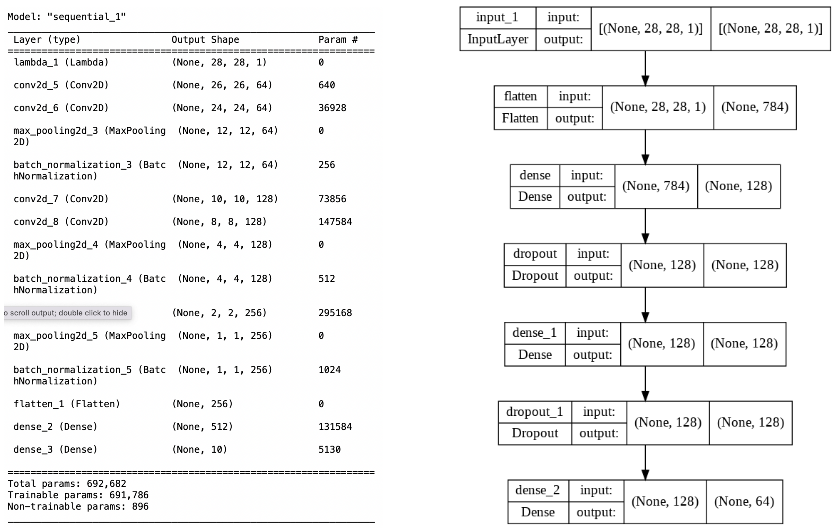
```
# First network sequential() method call
model=Sequential()
model.add(Lambda(standardize,input_shape=(28,28,1)))
model.add(Conv2D(filters=64, kernel_size = (3,3), activation="relu", input_
    shape=(28,28,1)))
model.add(Conv2D(filters=64, kernel_size = (3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(BatchNormalization())
model.add(Conv2D(filters=128, kernel_size = (3,3), activation="relu"))
model.add(Conv2D(filters=128, kernel_size = (3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(BatchNormalization())
model.add(Conv2D(filters=256, kernel_size = (3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(BatchNormalization())
model.add(Flatten())
model.add(Dense(512,activation="relu"))
model.add(Dense(10,activation="relu"))
```

```
# Second network model() class call
input_image = Input(shape=image_input_shape)
x = Flatten()(input_image)
x = Dense(128, activation='relu')(x)
x = Dropout(0.1)(x)
x = Dense(128, activation='relu')(x)
x = Dropout(0.1)(x)
x = Dense(embedding_size)(x)
base_network = Model(inputs=input_image, outputs=x)
```

The second step was the compilation with the method "model.compile()". The compilation is the final step in creating a model. Compiling the model allows us to put together all the information that we need to complete the definition of the network: the loss function to be used and the optimization method. We will show only one example of how to compile the model.

```
# Compiling the model
model.compile(loss="CategoricalCrossentropy", optimizer="adam", metrics=["
    CategoricalAccuracy"])
```

In the images below we can see the two different networks used for implementing the cross entropy losses and the one for implementing the triplet loss.



```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lambda_1 (Lambda)           (None, 28, 28, 1)         0

 conv2d_5 (Conv2D)           (None, 26, 26, 64)        640

 conv2d_6 (Conv2D)           (None, 24, 24, 64)        36928

 max_pooling2d_3 (MaxPooling  (None, 12, 12, 64)       0
 2D)

 batch_normalization_3 (Batc  (None, 12, 12, 64)       256
 hNormalization)

 conv2d_7 (Conv2D)           (None, 10, 10, 128)       73856

 conv2d_8 (Conv2D)           (None, 8, 8, 128)         147584

 max_pooling2d_4 (MaxPooling  (None, 4, 4, 128)        0
 2D)

 batch_normalization_4 (Batc  (None, 4, 4, 128)        512
 hNormalization)

o scroll output; double click to hide (None, 2, 2, 256)  295168

 max_pooling2d_5 (MaxPooling  (None, 1, 1, 256)        0
 2D)

 batch_normalization_5 (Batc  (None, 1, 1, 256)        1024
 hNormalization)

 flatten_1 (Flatten)         (None, 256)               0

 dense_2 (Dense)             (None, 512)               131584

 dense_3 (Dense)             (None, 10)                5130

=================================================================
Total params: 692,682
Trainable params: 691,786
Non-trainable params: 896
_________________________________________________________________
```

input_1 | input: | [(None, 28, 28, 1)] | [(None, 28, 28, 1)]
InputLayer | output:
flatten | input: | (None, 28, 28, 1) | (None, 784)
Flatten | output:
dense | input: | (None, 784) | (None, 128)
Dense | output:
dropout | input: | (None, 128) | (None, 128)
Dropout | output:
dense_1 | input: | (None, 128) | (None, 128)
Dense | output:
dropout_1 | input: | (None, 128) | (None, 128)
Dropout | output:
dense_2 | input: | (None, 128) | (None, 64)
Dense | output:

Figure 6.2: on the left Model 1, on the right Model 2

Both models have been trained with the "model.fit()" method. The full implementation of both networks that we used to run our loss function analysis comes from [37] and [74] and can be found in Appendix C.1 and C.2.

6.2 Experiments

In the following paragraphs the obtained results will be reported. These results have been obtained with the two neural networks that we described in the previous paragraph. The aim of these experiments was to confront how different loss functions perform in terms of training loss and accuracy. The confrontation has been useful to gain practical insights on their implementation. All the experiments have been performed on the MNIST handwritten digits database, and because of the nature of this type of problem we considered loss functions closely related to the solution of classification problems.
The results for accuracy have been obtained by submitting the "sample_submission.csv" file that we extracted from each model in the Kaggle's Digit Recognizer competition submission webportal `https://www.kaggle.com/competitions/digit-recognizer/submit`.

6.2.1 Categorical cross entropy

By recalling the definition that we already gave in the previous chapters, the cross entropy is formulated as:

$$CE = -\sum_{i=1}^{n} y_i \cdot \log \hat{y}_i \tag{6.1}$$

where n is the output size (number of classes), and $\hat{y}_i$ is the prediction output for the corresponding class, while y_i is the ground truth value for the predicted class. This loss is a good measure of how different two discrete probability distributions are from each other. In categorical cross entropy the sum of all $\hat{y}_i$ is constrained to be 1, meaning that CCE is better suited for single-label classification problems. The minus sign ensures that the loss gets smaller when the distributions get closer to each other. The first task that we considered for implementing categorical cross entropy, was to use one-hot encoding for the target values y.

```
# One-hot encoding
y = to_categorical(y)
```

Another thing to consider was what kind of activation function we would have to use in the output layer of the network. The two possible options were a Sigmoid function or a Softmax function, and we opted for the second one, although we have noticed that both types of function have a comparable performance on the network's accuracy. Let us highlight again the end of the network where the dense layers and the softmax activation function are implemented:

```
# Fully connected layers for classification
model.add(Flatten())
model.add(Dense(512,activation="relu"))
model.add(Dense(10,activation="relu"))
model.add(Dense(10,activation="softmax"))
```

Finally we chose what type of loss function to compile the model with. All the loss functions used in this experiment except for the Triplet loss, have been implemented from the Keras library. The metric "CategoricalAccuracy" for measuring the accuracy since it calculates how often predictions match one-hot encoded labels.

```
# CategoricalCrossentropy loss
model.compile(loss="CategoricalCrossentropy",
              optimizer="adam",
              metrics=["CategoricalAccuracy"])
```

The results that we obtained from running 25 epochs (number of training loops), are illustrated in the charts of fig. 6.3:

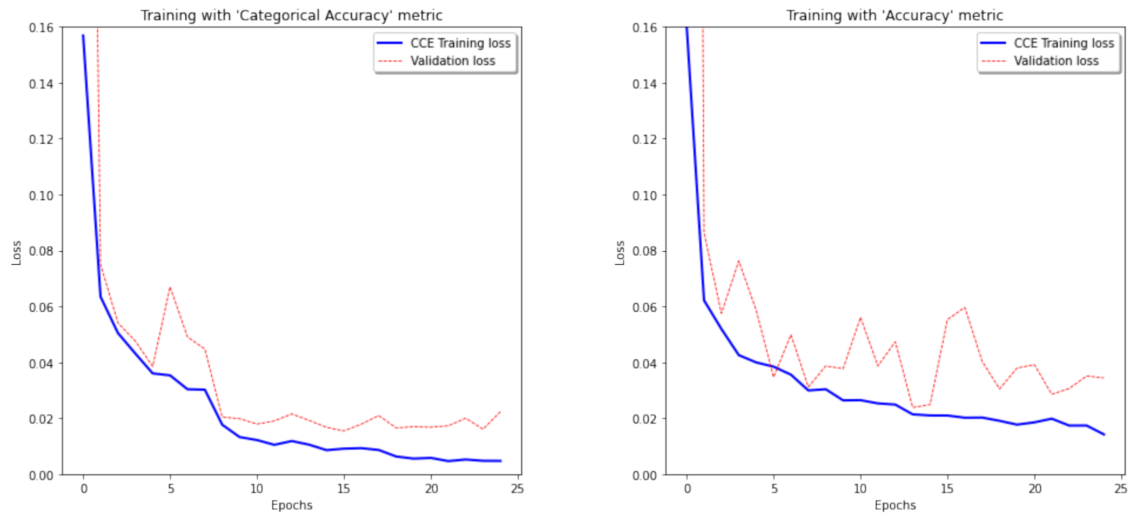


Figure 6.3: Training loss for CCE using different accuracy metrics

From the charts above we can see how the model is well posed, as the validation loss follows the training loss, signaling that the network is learning to generalize appropriately. According to the above experiment results, if the task is multi-class classification and true labels are encoded as a one-hot, we might have 2 options: either using Sigmoid or the Softmax function in the output layer. Problems would arise instead if we would forget to use an activation function We obtain the following final accuracy of 99.55%.

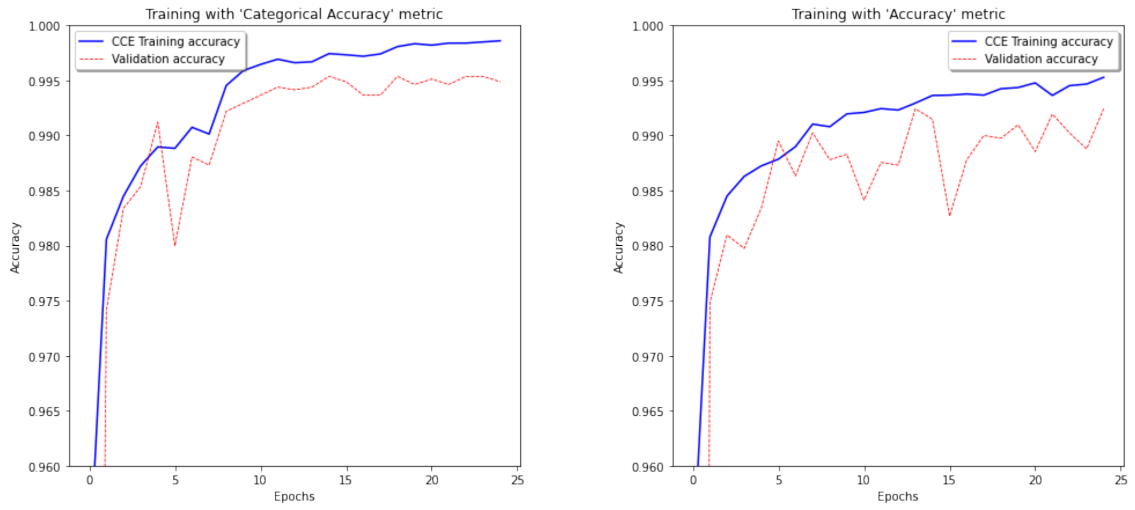


Figure 6.4: Training accuracy for CCE using different accuracy metrics

Sparse categorical cross entropy loss

In multi-class classification problems, the previous categorical cross entropy loss is the primary choice. That loss requires that your labels are one-hot encoded, which could not always be the case. In that case we use sparse categorical cross entropy loss. This loss function performs the same type of categorical cross entropy loss, but it works on integer targets instead of one-hot encoded ones. The last layer has 10 units (same as the number of problem classes). So the output (y_pred) will be 10 floating points and the true label (y) is a vector of 10 values of number between 0 and 9. We will not implement one-hot encoding with this type of loss. The final layers will be as follows:

```
# Fully connected layers for classification
model.add(Flatten())
model.add(Dense(512,activation="relu"))
model.add(Dense(10,activation="relu"))
# Output layer
model.add(Dense(10,activation="softmax"))
```

while when compiling the model we will choose "SparseCategoricalCrossentropy" as the loss function and the following parameters:

```
# SparseCategoricalCrossentropy loss
model.compile(loss="SparseCategoricalCrossentropy",
              optimizer="adam",
              metrics=["accuracy"])
```

Also in this case it is confirmed that training with the more appropriate accuracy metric (Sparse categorical accuracy in this case), smooths out the training loss curve, indicating a slightly less "wiggly" training curve.

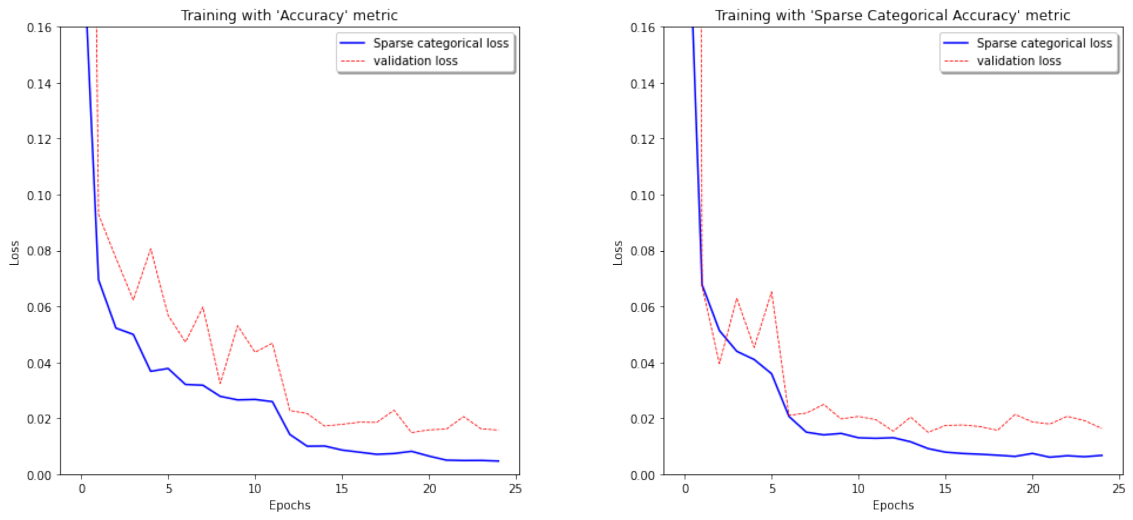


Figure 6.5: Training loss for SCE using different accuracy metrics

Overall we can conclude that the way accuracy is computed only affects the metric computation, but not the effective performance of the network. To confirm what is the true performance of the network we preferred to predict the results of the test data set and confront the prediction with the ground truth.

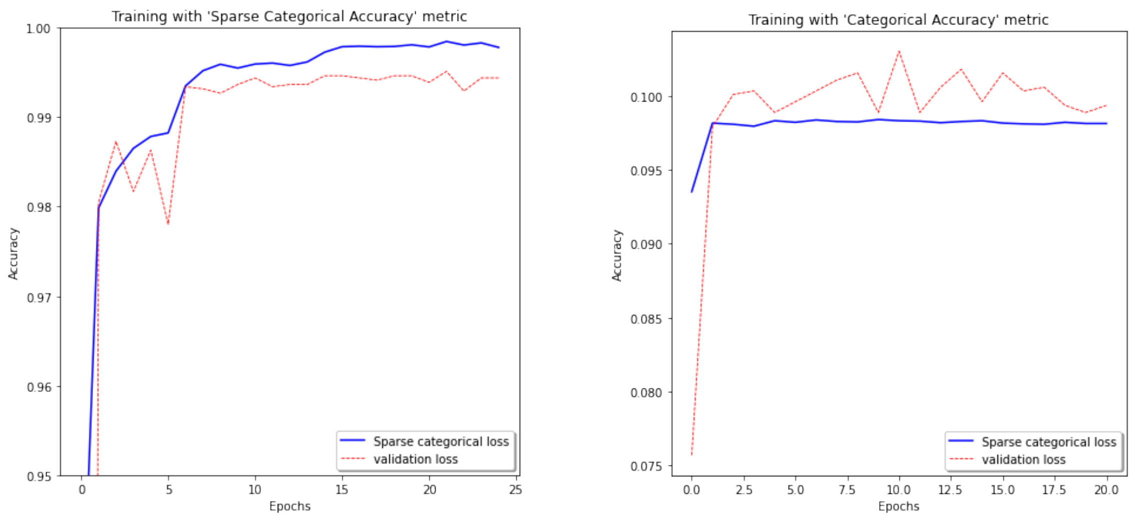


Figure 6.6: Training accuracy for SCE using different accuracy metrics

In the charts below we can see how sparse categorical cross entropy (SCE) and CCE are very similar to each other, confirming that the fundamental nature of the training has not changed.

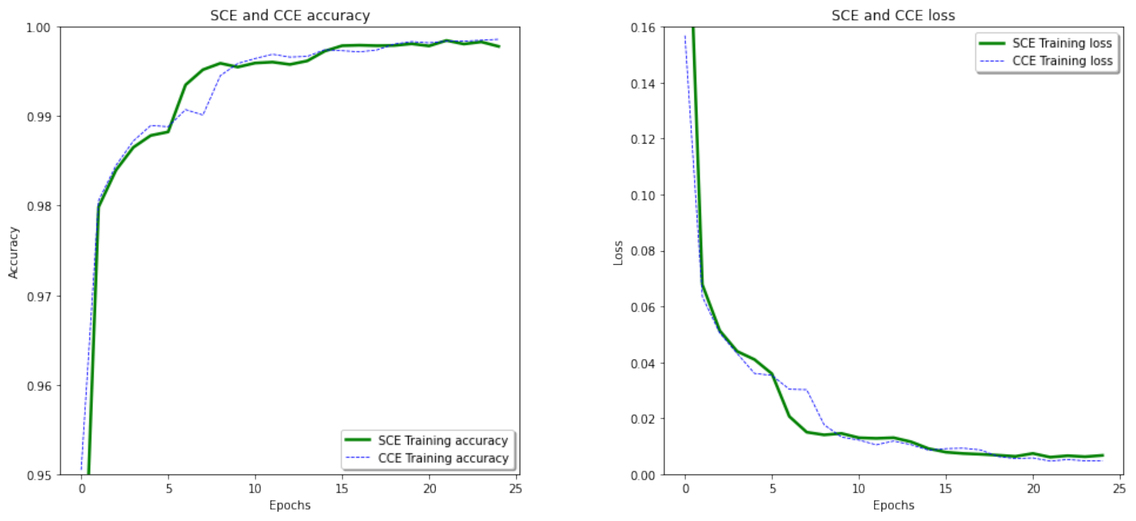


Figure 6.7: Confronting SCE with CCE

We can deduce that the best combination is given by a network that has a sparse target vector, a softmax output layer, while the metric used does not have a determinant impact on the performances. The final accuracy in this case is 99.51%.

6.2.2 Binary Cross entropy loss

Binary relevance is an intuitive solution for learning from multi-label examples [5]: in fact multi-label targets can be encoded as an alternative of 1 (the label belongs to the training example) or 0 (the label does not belong to the training example). If we consider an example in which we have the predicted output $\hat{y}$ as a vector of size n, and the target values y encoded as a sequence of 0 and 1, the binary cross entropy can be expressed as follows:

$$BCE = -\frac{1}{n}\sum_{i=1}^{n} y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i) \tag{6.2}$$

where n is the length of the output vector, y_i is the i-th target value and $\hat{y}_i$ is the i-th predicted value. This is equivalent to the average result of the categorical cross entropy loss function applied to many independent classification problems, each problem having only two possible classes. Therefore a multi-class problem can be trained using such a kind of cross entropy. By implementing the binary cross entropy in the model.compile function:

```
# Binary Cross Entropy loss
model.compile(loss="BinaryCrossentropy",
              optimizer="adam",
              metrics=["accuracy"])
```

we obtain the following results of accuracy and training error:

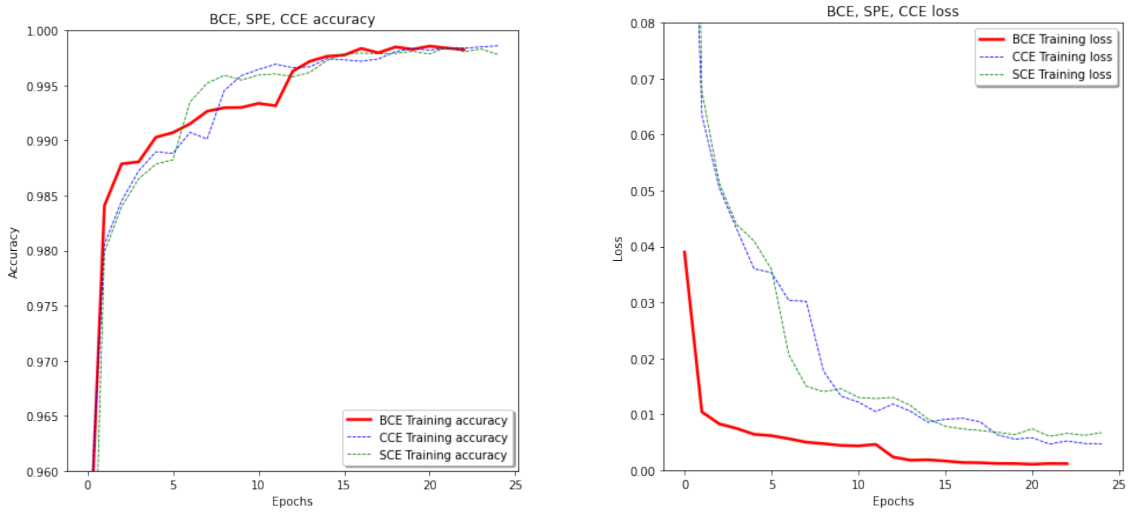


Figure 6.8: Confronting BCE with CCE, and SCE

As we can see from the plots below, the accuracy reached by the network using the binary cross entropy from the Keras library is in the same order as the one reached by SCE and CCE, even if is slightly inferior (99.46%). The error is lower than the preceding training errors, but that does not contribute in making the accuracy better than the previous cases.

6.2.3 Kullback-Leibler Divergenge loss

In multi-class problems instead of considering the categorical cross entropy loss function that only takes into account the argmax of the predictions generated, we can consider the softmax output as a probability distribution, and compare it with a target distribution, by using the KL-Divergence loss [1]. By one-hot encoding the target

```
# Without one-hot encoding
y = to_categorical(y)
```

and by setting the final layer is also activated by a softmax function

```
# Softmax final layer
model.add(Dense(10,activation="softmax"))
```

and selecting "KLDivergence" as loss function:

```
# KL-Divergence loss
model.compile(loss="KLDivergence",
              optimizer="adam",
              metrics=["accuracy"])
```

we can obtain the best performance reached. The results that we obtained are plotted in fig. 6.9.

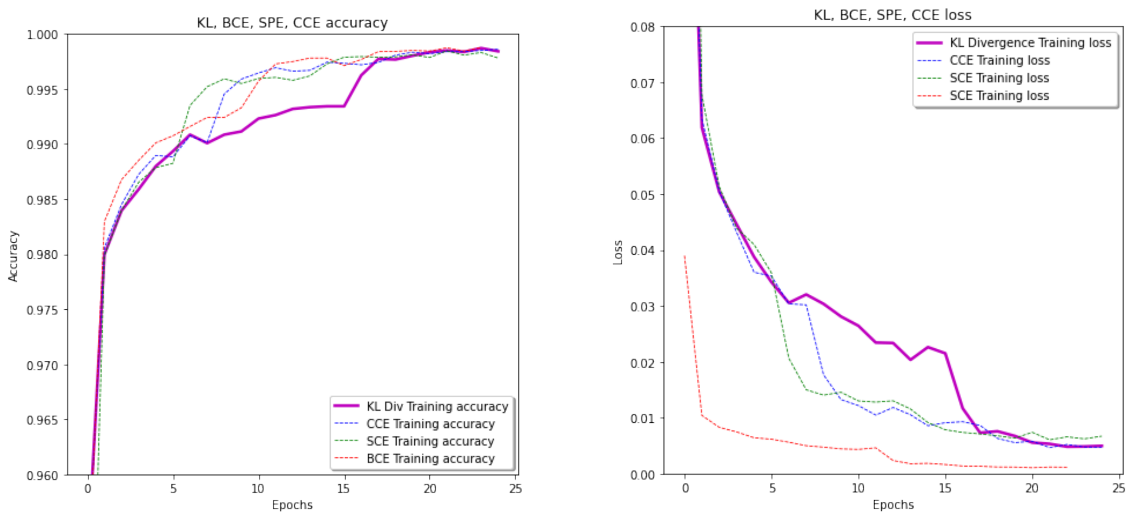


Figure 6.9: Confronting KL Divergence with CCE, SCE, BCE

We can observe that the loss follows closely to the CE losses, reaching an accuracy of 99.56%, the highest reached in the experiments. This shows that KL Divergence is a valid alternative to CCE, SCE and BCE.

6.2.4 Triplet Loss

The triplet loss has been introduced by the paper "FaceNet: A Unified Embedding for Face Recognition and Clustering" [63]. In most of supervised learning problems, there a fixed number of classes and the training is conducted by comparing the predictions with the target values. However in some cases, like in face recognition we need to be able to compare two unknown faces and say whether they are from the same person or not. Therefore triplet loss does not compare the prediction with a ground truth label, but instead training is conducted with triplets of data points, that are defined as follows:

- Anchor (a), any arbitrary data point
- Positive (p), a data point with the same class as the anchor point
- Negative (n), a data point with a different class from the anchor point

There are three types of triplets (easy, hard and semi-hard triplets) and they can be generated with two different methods (offline mining and online mining) [56]. Triplet loss is a distance metric that operates on a triplet, with the aim of minimizing the distance between the anchor and the positive points, and maximizing the distance between the anchor and the negative points. Its mathematical definition is trivial:

$$\mathrm{TL} = \max(\mathrm{d(a,p)} - \mathrm{d(a,n)} + \mathrm{margin}, 0) \tag{6.3}$$

where $d(a,p)$ is the distance between the anchor and the positive data point, which is intended to be minimized, $d(a,n)$ is the distance between the anchor and the negative point, which is intended to be maximized instead. This means that, after the training, the positive examples will be closer to the anchor while the negative examples will be farther from it. The implementation of the network for this type of loss can be found in the Appendix C.2. The training loss obtained is plotted together with the ones from the previous experiments in the following chart:

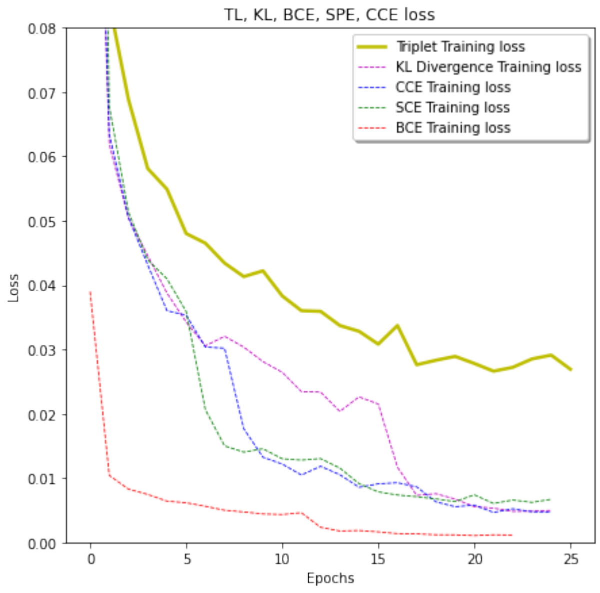


Figure 6.10: Confronting Triplet Loss with CCE, SCE, BCE and KL Divergence

The final accuracy obtained with this method was 97.91%, which is almost a 1.5% drop with respect to all other implementations. We can deduce that triplet loss is not the best type of loss to classify a problem like the one we considered. In the next chapter we will analyze the conclusions that we have been able to infer from the experiments.

Chapter 7

Conclusion

A deep learning architecture is a system composed of different pieces that influence each other. To be able to study how each one of these pieces influences the others we need to isolate them and make them work on different regimes, and measure how the final architecture response changes. In this thesis we focused on how the type of loss function influences the system and compared it with other functions. We made some interesting discoveries from observing the results. The main ones are outlined in the following paragraphs. We experimented using different neural networks and different loss functions: by doing so we can aim to identify general "good practice" rules, that could still be valid also when building any other kind of deep architecture.
Regarding the loss function experiments we considered two main classes of them: the first related to the cross entropy and the second one related to the concept of distance (triplet loss).
Another aspect that has emerged is that loss function can also have an impact on the choice of network architecture too. Triplet loss can not be implemented on the 5-Convolutional layer architecture that we used for the other loss functions, and we had to implement a second network to be able to use it. We then gave a justification on why cross entropy is better adapted for this type of task.

7.1 Influence of encoding categorical variables

One-hot encoding is used in multi-class classification problems. In these problems n-classes are the possible outcome of the classification, and they form a vector of length n. When one-hot encoding we transform a vector in a matrix, where each line corresponds to one class of the old vector, and is composed by zeros except for one element equal to one, that corresponds to the true category. The figure below represents the concept just described:

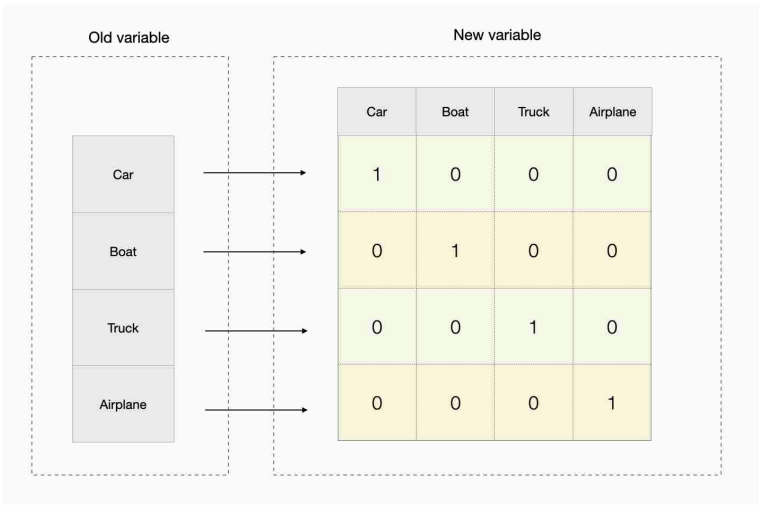


Figure 7.1: One-hot encoding

Each of these vectors can be thought of as a probability distribution where the ground truth has probability of 1. Also the prediction of the network can be thought of as a probability vector of mutually-exclusive outcomes, meaning that the prediction vector has only positive elements between 0 and 1 and their sum must equal to 1. This means that making one part of the vector larger must shrink the sum of the remaining components. This combination works well when using cross entropy, as we saw in the previous chapter. CCE, BCE and KL require one-hot encoding, as cross entropy is a metric interested in understanding how far we are from the correct prediction, by setting to zero all the elements of the output that do not correspond to the ground truth. If we do not one-hot encode our target variable we would get an error of this kind:

```
(0) INVALID_ARGUMENT:  logits and labels must have the same first dimension,
    got logits shape [128,10] and labels shape [1280]
```

This is signaling that the output is one-hot encoded, while the target label is a sparse vector and should either change the encoding of the ground truth vector or use a different loss function.

7.2 Activation functions

Output layer activation functions

Activation functions are used at the end of any layer. In the hidden layers the ReLu function is mostly used, while in the output layer two types of activation functions are generally applied: Sigmoid and Softmax. The first one is used in binary classification, as the function maps $(-\infty, +\infty)$ to $(0, 1)$. The second one is used instead in multi-class classification problems, as it partitions the probability among those classes. The formulation for these functions are:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{7.1}$$

for the Sigmoid function, and:

$$S(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \tag{7.2}$$

for the Softmax function. In our experiment we determined that they have a similar effect on the final accuracy, and do not impact differently the overall performance of the network. The reason for this similarity in performance by using either a softmax or a sigmoid activation function in the output layer, is that both have a similar effect on data transformation. The sigmoid function smooths out each value in a continuous range between 0 and 1. The softmax gets the same result by standardizing each element between 0 and 1 like they would all belong to the same distribution. Generally [11], for multi-class classification, the recommended activation function is the softmax. Although, sigmoid is mostly used for binary classification and multi-label classification.

7.3 Softmax removal

We show in the figure below the effect that turning off the activation function in the output layer has on the training error and on the accuracy. This means that the final layer will be activated only by the ReLU function. But this function only flattens negative values to zero. This prevents the cross entropy to compute the loss in the case that the correct class has been predicted with a negative value. In classification therefore ReLU is not used [16]. This is what we observe when using a ReLU activation function as the output in a multi-class classification problem:

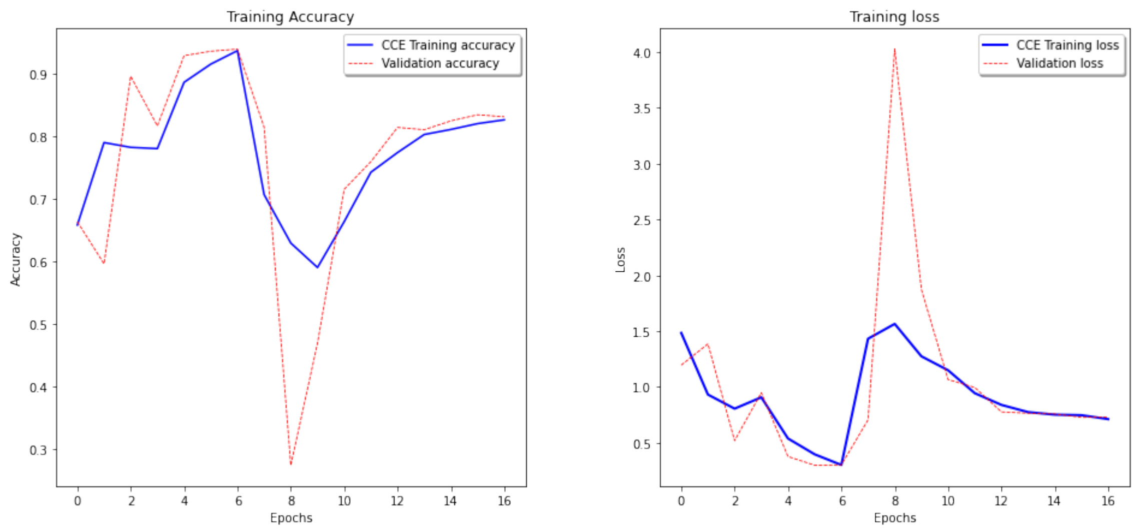


Figure 7.2: Training loss and accuracy when softmax function is removed

The algorithm is not able to learn an appropriate generalization of the data set as the error does not decrease with the passing of the epochs. As we could expect, because this is a classification problem, using ReLU at the output layer causes a similar effect independently from the loss function used.

7.4 Which solution shall we adopt?

7.4.1 Confronting cross entropy and KL Divergence

We observed that the best implementation has been obtained with "CategoricalCrossentropy" and "KLDivergence" loss function. The training curves are all very similar, and the results for accuracy (99.55% and 99.56% respectively) makes us conclude that cross entropy is a better measurement in multi-class problems. From the chart below in fig. 7.3 we can observe that the cross entropy group (CCE, SCE, KL Divergence and BCE) all have similar performances. Only the triplet loss (TL) has a drop of almost 1.5% in accuracy.

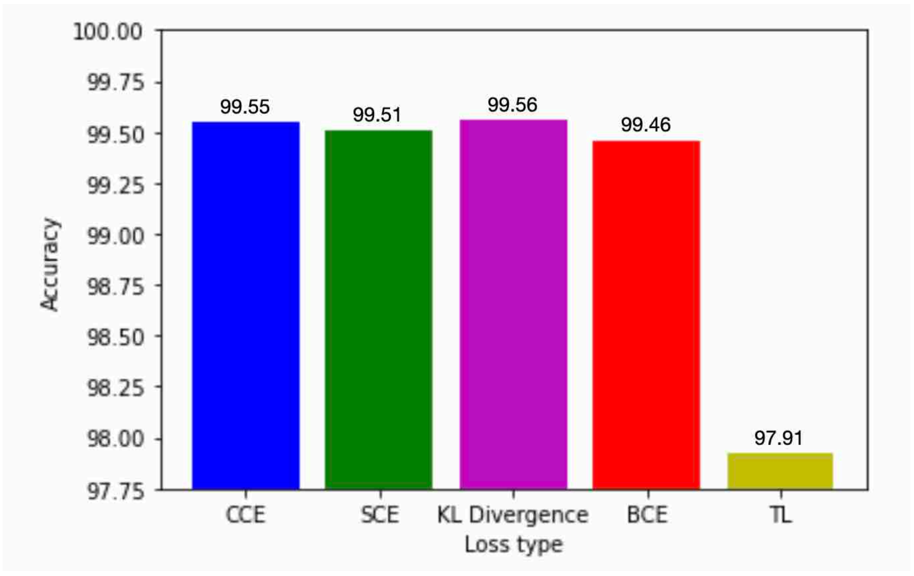


Figure 7.3: Final results

This is because of the nature of the concept of entropy. This concept comes from information theory [68], and defines entropy as a measure of a system's uncertainty. Entropy is defined as:

$$H(x) = -\sum_i p(x_i) \cdot \log p(x_i) \tag{7.3}$$

where H(x) is the avarage of the quantity of information contained by the random variable x, $p(x_i)$ is the probability of a particular event of x and $\log p(x_i)$ is the quantity of information of an event. If for example we consider $\log_2$ then the number of bits for the event x_i are defined as:

$$I(x_i) = -\log_2 p(x_i) \tag{7.4}$$

The Kullback–Leibler Divergence is defined as we already saw as:

$$D_{KL}(p_A \| p_B) = \sum_i p_A(x_i) \cdot \log p_A(x_i) - \sum_i p_A(x_i) \cdot \log p_B(x_i) \tag{7.5}$$

where p_A is the probability distribution of an event A (distribution of the data set) and p_B is the probability distribution of an event B (model prediction). The second term of this equation quantifies the amount of information of the distribution B conditioned by the true distribution A. The first term is the average information contained in A. Therefore the D_{KL} describes how much different B is from A. The cross entropy is instead defined as:

$$H(A,B) = -\sum_i p_A(x_i) \log p_B(x_i) \tag{7.6}$$

where p_A is the probability of the i-th event in A and $\log p_B(x_i)$ is the quantity of information of x_i if it would follow p_B. We can affirm then that:

$$H(A,B) = D_{KL}(A||B) + H(A) \tag{7.7}$$

where $D_{KL}(A||B)$ represents the difference between the model and the data and $H(A)$ is the entropy of the data distribution. If H(A) is constant (the training data set does not change), minimizing $H(A,B)$ corresponds to minimizing $D_{KL}(A||B)$. This explains the similarity in the results between the implementation of the two methods that we observed.

7.4.2 A note on binary cross entropy

We implemented "Binary Cross Entropy" from the Keras library for Python. It is important to specify that binary cross entropy is not just used when the output is a single number that can pick up only two classes [76]. In Keras, the binary Cross-Entropy loss can be used if the true label is binary, without the requirement for the output to be a probability distribution, like in the "Categorical Cross Entropy". Therefore the "binary" name is because it is adapted for binary outputs, and each number of the softmax is aimed at being 0 or 1. This explains why the loss differs from the CCE, SCE, KL losses, while the accuracy remains similar. This type of loss would not be recommended when implementing a multi-class classification problem like the one we considered.

7.4.3 A note on Triplet loss: decreasing accuracy and architectural risk

The use of the Triple loss has presented some important challenges. Its implementation required a different network architecture from the one in which we used the other loss functions. In fig.7.4 we can see a representation of a general CNN architecture, similar to the one we used to implement CCE, SCE, BCE and KL Divergence losses: after the convolutional layers that perform feature extraction, the fully connected layers recognize the features and output a prediction vector, usually applying a softmax activation function before the final output:

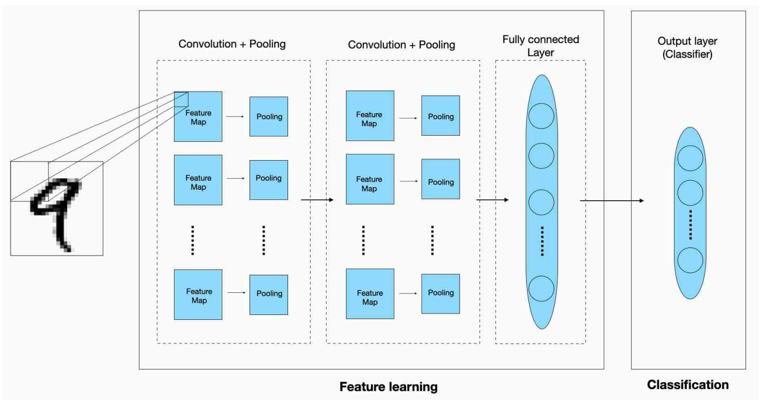


Figure 7.4: Convolutional networks architecture conceptualisation

The second-last layer (the layer before the output layer), is already aware of the final prediction, but needs one more layer (the output layer) to explicit the results in an interpretable way. That is because in the layer before the output the features are already organized at their highest hierarchy. Siamese networks (identical networks connected in parallel) [8], work in a different way instead: their architecture stops at the layer with the highest hierarchy feature composition, where the features are grouped into a final embedding (see fig. 7.5). Siamese networks, confront their outputs computing the distance between them, and if the embeddings obtained are close enough, that means that also the inputs that produced them are from the same class. In triplet loss we use three identical networks connected in parallel, which are fed with three inputs that form the so called "Triplet", which is composed by an anchor, a positive example and a negative example. Each of the inputs pass through one of the Siamese networks, and the output of each network will be an embedding of the input, as fig.7.5 shows. These embeddings can be thought of as belonging to a space where we want to minimise the distance between the anchor and positive embedding, and maximize the distance between the anchor and negative embeddings [56].

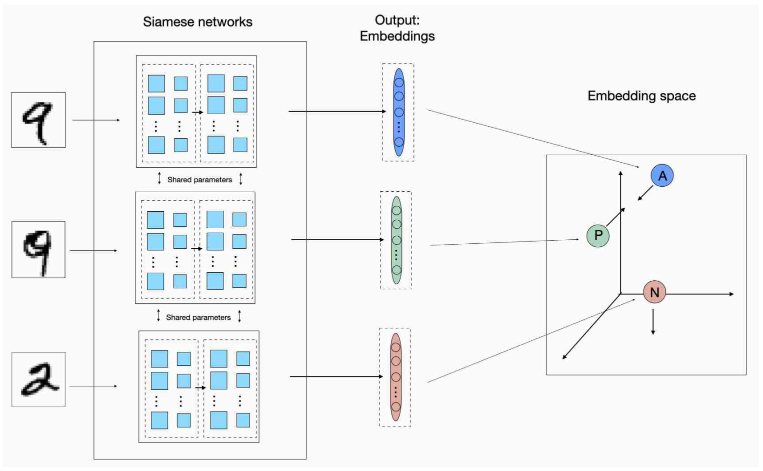


Figure 7.5: Siamese networks architecture conceptualisation

By adopting this approach we changed drastically the way classification is interpreted, as we are not considering entropy as fundamental measurement of diversity, but distances instead. We have also implemented a different (and more complex) type of deep architecture to enable triplet loss. We observed that by building an initial version of this type of network the accuracy has decreased by 1.5 percentage points, with respect to the other methods. Computational time has not increased despite the increased complexity of the network. Even if we are aware that the accuracy can be improved by perfecting the network, we can conclude that the implementation of this network has to be considered carefully before making the decision of adopting it, as it could lead to a riskier implementation, only to achieve the already very good performances reached by simpler networks. We can conclude that:

- The loss function can drastically affect the network's architecture.
- A more complex architecture does not correspond to a positive trade-off between architecture complexity and accuracy.

7.5 Final considerations

In this research we have followed an "ablation-type" of approach, in which we partially make studies removing the activation function from the final layer, but most importantly we plugged in different loss functions and observed how results would have changed. This approach has been rewarding in terms of making evident what are the fundamental aspects to consider in order to make correct design choices. The technical tool used for the implementation (such as Tensorflow and PyTorch), also had an impact on the type of methods to use in the neural network. Each different architecture had a different objective function and structure. On the base of this we then considered what would have been the implementation that guarantees the better efficiency in terms of results versus implementation complexity. It was in fact found that complexity can scale-up quickly, while performances stay similar, or in the worst case degrade. We were also able to give a mathematical interpretation regarding why cross entropy is better adapted than triplet loss to this type of task in par. 7.4.1. We can conclude that the most important approach to adopt when studying neural networks is the empirical one, that observes how results change, when network's components and parameters are changed. In this case we have chosen loss function as the parameter to observe, but this approach can be repeated with any other fundamental part of a network.

Bibliography

[1] Moreno A. What are some applications of the KL-divergence in machine learning? `https://www.quora.com/What-are-some-applications-of-the-KL-divergence-in-machine-learning/answer/Alexander-Moreno-1`. 2015.

[2] David H. Ackley, Geoffrey E. Hinton, and Terrence J. Sejnowski. “A learning algorithm for boltzmann machines”. In: Cognitive Science 9.1 (1985), pp. 147–169. URL: `https://www.sciencedirect.com/science/article/pii/S0364021385800124`.

[3] Blake A. Richards et al. “Nature Neuroscience”. In: A deep learning framework for neuroscience 22.11 (2019), 1761—1770.

[4] Dimitris Bertsimas et al. “Robust Classification”. In: INFORMS 1 (2019). eprint: 2-34.

[5] Zhang ML. et al. In: ().

[6] Chuan Yu Foo Yifan Mai Caroline Suen Adam Coates Andrew Maas Awni Hannun Brody Huval Tao Wang Sameep Tandon Andrew Ng Jiquan Ngiam. Autoencoders UFLDL Tutorial. `http://ufldl.stanford.edu/tutorial/unsupervised/Autoencoders`. Last accessed: 03-Sep-22.

[7] Backpropagation calculus, Deep learning, 3Blue1Brown. `https://www.youtube.com/watch?v=tIeHLnjs5U8`.

[8] Tanmay Bakshi. Modelling MNIST using Triplet Neural Networks. `https://www.youtube.com/watch?v=dG8le1YWUI8&t=574s`. Last accessed: 03-Sep-22.

[9] Hmrishav Bandyopadhyay. Autoencoders in Deep Learning. `https://www.v7labs.com/blog/autoencoders-guide`. Last accessed: 03-Sep-22.

[10] Andrew G. Barto, Richard S. Sutton, and Peter S. Brouwer. “Associative Search Network: A Reinforcement Learning Associative Memory”. In: Biol. Cybern. 40.3 (1981), 201–211. ISSN: 0340-1200. URL: `https://doi.org/10.1007/BF00453370`.

[11] Nikola Basta. The Differences between Sigmoid and Softmax Activation Functions. `https://medium.com/arteos-ai/the-differences-between-sigmoid-and-softmax-activation-function-12adee8cf322`. Last accessed: 03-Sep-22.

[12] Count Bayesie. Kullback-Leibler Divergence Explained. Last accessed: 03-Sep-22.

[13] Ali Beikmohammadi, Karim Faez, and Ali Motallebi. “SWP-LeafNET: A novel multistage approach for plant leaf identification based on deep CNN”. In: Expert Systems with Applications 202 (2022), p. 117470.

[14] Sarat Moka Benoit Liquet and Yoni Nazarathy. Optimization algorithms. `https://deeplearningmath.org/optimization-algorithms.html`. Last accessed: 03-Sep-22.

[15] Carolina Bento. Multilayer Perceptron Explained. https://towardsdatascience.com/multilayer-perceptron-explained-with-a-real-life-example-and-python-code-sentiment-analysis-cb408ee93141. Last accessed: 03-Sep-22.

[16] Can I use Relu activation function in output layer, if not why so? https://www.quora.com/Can-I-use-Relu-activation-function-in-output-layer-if-not-why-so. Last accessed: 03-Sep-22.

[17] T. Cerquitelli and E. Baralis Polytechnic University of Turin. Regression Analysis: Fundamentals. https://dbdmg.polito.it/wordpress/wp-content/uploads/2019/11/Regression-Analysis.pdf. Last accessed: 03-Sep-22.

[18] Chain Rule, Deep Learning Tutorial 15, Codebasics. https://www.youtube.com/watch?v=5ogmEkujoqE. Last accessed: 03-Sep-22.

[19] Chain Rule of Differentiation with BackPropagation, Krish Naik. https://www.youtube.com/watch?v=CRB266Eyjkg. Last accessed: 03-Sep-22.

[20] IBM Cloud. What are recurrent neural networks? https://www.ibm.com/cloud/learn/recurrent-neural-networks. Last accessed: 03-Sep-22.

[21] Cross-entropy for classification. https://towardsdatascience.com/cross-entropy-for-classification-d98e7f974451. Last accessed: 03-Sep-22.

[22] D.Q.F. de Menezes et al. "A review on robust M-estimators for regression analysis". In: Computers Chemical Engineering 147 (2021), p. 107254.

[23] 6S191 MIT DeepLearning. Deep computer vision. http://introtodeeplearning.com/slides/6S191_MIT_DeepLearning_L3.pdf. Last accessed: 03-Sep-22.

[24] Howard B. Demuth et al. Neural Network Design. 2nd. Stillwater, OK, USA: Martin Hagan, 2014. ISBN: 0971732116.

[25] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: J. Mach. Learn. Res. 12.null (2011), 2121–2159.

[26] IBM Cloud Education. Gradient Descent. https://www.ibm.com/cloud/learn/gradient-descent. Last accessed: 03-Sep-22.

[27] IBM Cloud Education. What is deep learning? https://www.ibm.com/cloud/learn/deep-learning. Last accessed: 03-Sep-22.

[28] Entropy, Cross Entropy, and KL Divergence. https://towardsdatascience.com/entropy-cross-entropy-and-kl-divergence-17138ffab87b. Last accessed: 03-Sep-22.

[29] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep Learning. http://www.deeplearningbook.org. MIT Press, 2016.

[30] Ian J. Goodfellow et al. Generative Adversarial Networks. 2014. URL: https://arxiv.org/abs/1406.2661.

[31] How the backpropagation algorithm works. http://neuralnetworksanddeeplearning.com/chap2.html. Last accessed: 03-Sep-22.

[32] How to calculate the number of parameters for convolutional neural network? https://stackoverflow.com/questions/42786717/how-to-calculate-the-number-of-parameters-for-convolutional-neural-network. Last accessed: 03-Sep-22.

[33] Kilian Weinberger Machine Learning for Intelligent Systems Cornell CS4780/CS5780. **Bias-variance trade off**. `https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote12.html`.

[34] Kilian Weinberger Machine Learning for Intelligent Systems Cornell CS4780/CS5780. **SVM**. `https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote09.html`.

[35] Gareth James et al. **An Introduction to Statistical Learning: with Applications in R**. Springer, 2013.

[36] Jeremy Jordan. **Introduction to autoencoders**. `https://www.jeremyjordan.me/autoencoders/`. Last accessed: 03-Sep-22.

[37] Kaggle.com. **MNIST: Simple CNN keras**. `https://www.kaggle.com/code/elcaiseri/mnist-simple-cnn-keras-accuracy-0-99-top-1`. Last accessed: 03-Sep-22.

[38] Brian Ingalls Kim Cuddington Andrew M. Edwards. **Machine learning and classification**. Last accessed: 03-Sep-22.

[39] Diederik P. Kingma and Jimmy Ba. **Adam: A Method for Stochastic Optimization**. 2014. URL: `https://arxiv.org/abs/1412.6980`.

[40] Priyankur Sarkar on knowledgehut. **Support Vector Machines in Machine Learning**. https://www.knowledgehut.com/blog/data-science/support-vector-machines-in-machine-learning.

[41] T. Kohonen. “The self-organizing map”. In: **Proceedings of the IEEE** 78.9 (1990), pp. 1464–1480. DOI: `10.1109/5.58325`.

[42] Teuvo Kohonen. “Self-organized formation of topologically correct feature maps”. In: **Biological Cybernetics** 43 (2004), pp. 59–69.

[43] Simeon Kostadinov. **Understanding Backpropagation Algorithm**. `https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd`. Last accessed: 03-Sep-22.

[44] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: **Advances in Neural Information Processing Systems**. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: `https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf`.

[45] Sebastian Kwiatkowski. **Entropy is a measure of uncertainty**. `https://towardsdatascience.com/entropy-is-a-measure-of-uncertainty-e2c000301c2c`. Accessed: 2022-07-24.

[46] Cornell CS 5787: Applied Machine Learning. **Soft Margins and the Hinge Loss**. `https://www.youtube.com/watch?v=kgTKHq9G21Q`. Last accessed: 03-Sep-22.

[47] Stanford CS 230 Deep Learning. **Recurrent Neural Networks cheatsheet**. `https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks`. Last accessed: 03-Sep-22.

[48] Y. Lecun et al. “Gradient-based learning applied to document recognition”. In: **Proceedings of the IEEE** 86.11 (1998), pp. 2278–2324. DOI: `10.1109/5.726791`.

[49] Chris A. Mack. **From Data to Decisions: Measurement, Uncertainty, Analysis, and Modeling**. http://www.lithoguru.com/scientist/statistics/Lecture56.pdf. 2016.

[50] Vlastimil Martinek. Cross-entropy for classification. https://towardsdatascience.com/cross-entropy-for-classification-d98e7f974451. Accessed: 2022-07-24.

[51] Mean Squared Error : Overview, Examples, Concepts and More. https://www.simplilearn.com/tutorials/statistics-tutorial/mean-squared-error. Last accessed: 03-Sep-22.

[52] Gabor Melli. Hinge-Loss Function. http://www.gabormelli.com/RKB/Hinge-Loss_Function. Accessed: 2022-07-24.

[53] STAT 501 Regression Methods. Robust Regression Methods. https://online.stat.psu.edu/stat501/lesson/13/13.3.

[54] Pragati Baheti at Microsoft. The Essential Guide to Neural Network Architectures. https://www.v7labs.com/blog/neural-network-architectures-guide. Last accessed: 03-Sep-22.

[55] STAT 897D Applied Data Mining and Statistical Learning. Ridge Regression. https://online.stat.psu.edu/stat857/node/155/.

[56] Olivier Moindrot. Triplet Loss and Online Triplet Mining in TensorFlow. https://omoindrot.github.io/triplet-loss. Last accessed: 03-Sep-22.

[57] Multiple linear regression. https://online.stat.psu.edu/stat462/node/131/. Accessed: 2022-07-24.

[58] Multiple Linear Regression. https://www.investopedia.com/terms/m/mlr.asp. Last accessed: 03-Sep-22.

[59] Christopher Desjardins Okan Bulut. Supervised Machine Learning - Part II. Last accessed: 03-Sep-22.

[60] Penalized Regression Essentials: Ridge, Lasso Elastic Net. Last accessed: 03-Sep-22.

[61] Marius-Constantin Popescu et al. "Multilayer perceptron and neural networks". In: WSEAS Transactions on Circuits and Systems 8 (July 2009).

[62] Iqbal Sarker. "Machine Learning: Algorithms, Real-World Applications and Research Directions". In: (Mar. 2021). DOI: 10.20944/preprints202103.0216.v1.

[63] Florian Schroff, Dmitry Kalenichenko, and James Philbin. "FaceNet: A unified embedding for face recognition and clustering". In: 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). IEEE, 2015. URL: https://doi.org/10.1109%2Fcvpr.2015.7298682.

[64] Scikit-Learn. Plot the support vectors in LinearSVC. https://scikit-learn.org/stable/auto_examples/svm/plot_linearsvc_support_vectors.html. Last accessed: 03-Sep-22.

[65] Alex Sherstinsky. "Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network". In: Physica D: Nonlinear Phenomena 404 (2020), p. 132306. URL: https://doi.org/10.1016%2Fj.physd.2019.132306.

[66] Lavanya Shukla. Designing Your Neural Networks. https://towardsdatascience.com/designing-your-neural-networks-a5e4617027ed. Last accessed: 03-Sep-22.

[67] Simple linear regression. https://en.wikipedia.org/wiki/Simple_linear_regression. Last accessed: 03-Sep-22.

[68] Stackexchange. **What is the difference Cross-entropy and KL divergence?** https://stats.stackexchange.com/questions/357963/what-is-the-difference-cross-entropy-and-kl-divergence. Last accessed: 03-Sep-22.

[69] CS231n Stanford. **Convolutional Neural Networks for visual recognition**. https://cs231n.github.io/convolutional-networks/. Last accessed: 03-Sep-22.

[70] Shiliang Sun et al. **A Survey of Optimization Methods from a Machine Learning Perspective**. 2019. URL: https://arxiv.org/abs/1906.06821.

[71] **Support-vector machine**. https://en.wikipedia.org/wiki/Support-vector_machine. Last accessed: 03-Sep-22.

[72] **Understanding Categorical Cross-Entropy Loss, Binary Cross-Entropy Loss, Softmax Loss, Logistic Loss, Focal Loss and all those confusing names**. https://gombru.github.io/2018/05/23/cross_entropy_loss/. Last accessed: 03-Sep-22.

[73] **Understanding Loss Functions in Machine Learning**. https://www.section.io/engineering-education/understanding-loss-functions-in-machine-learning/. Last accessed: 03-Sep-22.

[74] Adrian Ung. **keras-triplet-loss-mnist**. https://github.com/AdrianUng/keras-triplet-loss-mnist. Last accessed: 03-Sep-22.

[75] **What is the influence of C in SVMs with linear kernel?** Last accessed: 03-Sep-22.

[76] **Why binary_crossentropy and categorical_crossentropy give different performances for the same problem?** https://stackoverflow.com/questions/42081257/why-binary-crossentropy-and-categorical-crossentropy-give-different-performances. Last accessed: 03-Sep-22.

[77] **Why use softmax only in the output layer and not in hidden layers?** https://stackoverflow.com/questions/37588632/why-use-softmax-only-in-the-output-layer-and-not-in-hidden-layers. Accessed: 2022-07-24.

[78] Wikipedia. **Generative model**. https://en.wikipedia.org/wiki/Generative_model. Last accessed: 03-Sep-22.

[79] Wikipedia. **Recurrent neural network**. https://en.wikipedia.org/wiki/Recurrent_neural_network. Last accessed: 03-Sep-22.

Appendix A

Appendix A

A.1 Plotting minimisation of an objective function with gradient descent in Python, code from [14]

```
# Imports
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.ticker import LinearLocator
import numpy as np

# Define the function
def f(t):
    return (t[0] + 2*t[1] - 7)**2 + (2*t[0] + t[1] -4)**2

# Define the gradient
def grad(t):
    return np.array([10*t[0] + 8*t[1] - 34, 8*t[0] + 10*t[1] - 38])

# Find the optimal learning rate
def line_search_exact(t, d):
    nume = (t[0] + 2*t[1] - 7)*(d[0] + 2*d[1]) + (2*t[0] + t[1] - 5)*(2*d[0]
    + d[1])
    deno = (d[0] + 2*d[1])**2 + (2*d[0] + d[1])**2
    return -nume/deno

# Compute gradient
def GradientDescent_exact(f, grad, t, niter):
    g = grad(t)
    norm = np.linalg.norm(g)
    d = -g/norm
    t_seq = [t]
    d_seq = [d]

    for _ in range(niter):
        alpha = line_search_exact(t, d)

        # Theta update
        t = t + alpha*d
        t_seq.append(t)

        g = grad(t)
        norm = np.linalg.norm(g)
        d = -g/norm
        d_seq.append(d)
    print('Final point of GD:', t)
    t_seq = np.array(t_seq)
    d_seq = np.array(d_seq)
    return t_seq, d_seq

t_init = np.array([-9.0, 8.0])

t1_range = np.arange(-10.0, 10.0, 0.05)
t2_range = np.arange(-8.0, 10.0, 0.05)
A = np.meshgrid(t1_range, t2_range)
Z = f(A)

fig, ax = plt.subplots()
ax.contour(t1_range, t2_range, Z, levels=np.exp(np.arange(-10, 6, 0.5)), cmap
   ='binary', linewidths = 0.5)

t_seq_gd, d_seq_gd = GradientDescent_exact(f, grad, t_init, niter=10)

ax.plot(t_seq_gd[:, 0], t_seq_gd[:, 1], '-r')

plt.xlabel(r'$\theta_1$')
plt.ylabel(r'$\theta_2$')
plt.legend()
plt.show()
```

A.2 Plotting Hinge Loss in Python

```
# Imports
%matplotlib inline

import matplotlib
import numpy as np
import matplotlib.pyplot as plt

# Define variable x
x = np.linspace(-2.0, 2.0, num=100)

# Define Hinge Loss function
def hingeLoss(x):
    return max([0, 1-x])

# Plot the loss function
hinge, = plt.plot(x, [hingeLoss(i) for i in x], 'k-', label='Hinge Loss')
plt.legend(handles=[hinge], loc=0)
plt.ylabel('Loss value')
plt.xlabel('Model prediction')
```

A.3 Plotting cross entropy in R

```
x=seq(0,1,0.001)

y=-log(x)

plot(x,y, type="l", col="black", lwd=3, xlab="Score", ylab="-log(x)", main="
   Cross-entropy Loss",
     ylim = c(-1,8), xlim = c(-0.05,1))

abline(v = 0, lty=2)

abline(h= 0, lty=2)

points(x = 1, y =0, pch = 19)
```

A.4 Changing hyperparameter C in SVMs in Pyhton, code from [64]

```
# Imports
import matplotlib.pyplot as plt
from sklearn import svm
from sklearn.datasets import make_blobs
from sklearn.inspection import DecisionBoundaryDisplay

X, y = make_blobs (n_samples=50, centers=2, n_features=2, center_box=(0, 9),
    random_state=9)

# fit the model, don't regularize for illustration purposes
clf = svm.SVC(kernel="linear", C=100)
clf.fit(X, y)

plt.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=plt.cm.Paired)

# plot the decision function
ax = plt.gca()
DecisionBoundaryDisplay.from_estimator(
    clf,
    X,
    plot_method="contour",
    colors="k",
    levels=[-1, 0, 1],
    alpha=0.5,
    linestyles=["--", "-", "--"],
    ax=ax,
)
# plot support vectors
ax.scatter(
    clf.support_vectors_[:, 0],
    clf.support_vectors_[:, 1],
    s=100,
    linewidth=1,
    facecolors="none",
    edgecolors="k",
)

plt.title("C = 1")
plt.show()
```

Appendix B

Appendix B

B.1 Plotting OLS regression vs Robust regression in R

```
# Plot regression lines
library(robustbase)
head(starsCYG)
plot(starsCYG, xlim = c(3,5), ylim = c(3,7), pch = 20, cex =1.5, col = "black
   ", xlab="X", ylab="Y", main="Robust regression")
res <- lm(log.light ~ log.Te,
             data= starsCYG)
abline(lm(log.light ~ log.Te,
          data= starsCYG), col = "blue", lty = 2, lwd = 1)

abline(lmrob(log.light ~ log.Te,
          data= starsCYG), col = "blue", lty = 1, lwd = 2)
```

B.2 Plotting OLS loss vs Huber Loss in Python

```
# Plot OLS loss vs Huber loss
import numpy as np
from matplotlib import pyplot as plt
import tensorflow as tf

plt.rcParams["figure.figsize"] = [5.5, 5]
def f(x):
   return x**2
def huber_fn(x):
        is_small_error = tf.abs(x) < 1
        squared_loss = tf.square(x) / 2
        linear_loss  = tf.abs(x) - 0.5
        return tf.where(is_small_error, squared_loss, linear_loss)

x = np.linspace(-8, 8, 100)
plt.plot(x, f(x), color='orange',  linewidth=1, linestyle='--')
plt.plot(x, huber_fn(x), color='blue' ,  linewidth=1)
plt.ylabel('Loss')
plt.xlabel('Residuals')
plt.ylim((0,20))
plt.legend(["OLS loss", "Huber Loss"], loc ="upper right")
plt.show()
```

Appendix C

Appendix C

C.1: Convolutional neural network in Python, code from [37]

```
# Imports
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split
#from sklearn.metrics import confusion_matrix

import keras
from keras.models import Sequential
from keras.layers import Conv2D, Lambda, MaxPooling2D # convolution layers
from keras.layers import Dense, Dropout, Flatten # core layers
from tensorflow.keras.layers import BatchNormalization

from keras.preprocessing.image import ImageDataGenerator

from keras.utils.np_utils import to_categorical

from keras.datasets import mnist

# Set directory
import os
print(os.listdir(''))

# Load data
train = pd.read_csv('')
test = pd.read_csv('')
sub = pd.read_csv('')

X = train.drop(['label'], 1).values
y = train['label'].values

X = X / 255.0

# Reshape image in 3 dimensions (height = 28px, width = 28px , canal = 1)
X = X.reshape(-1,28,28,1)

# One-hot encodign
y = to_categorical(y)
```

```
# Split the train and the validation set for the fitting
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1,
    random_state=0)

X_train__ = X_train.reshape(X_train.shape[0], 28, 28)

mean = np.mean(X_train)
std = np.std(X_train)

def standardize(x):
    return (x-mean)/std

model=Sequential()

model.add(Lambda(standardize,input_shape=(28,28,1)))
model.add(Conv2D(filters=64, kernel_size = (3,3), activation="relu", input_
    shape=(28,28,1)))
model.add(Conv2D(filters=64, kernel_size = (3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(BatchNormalization())
model.add(Conv2D(filters=128, kernel_size = (3,3), activation="relu"))
model.add(Conv2D(filters=128, kernel_size = (3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(BatchNormalization())
model.add(Conv2D(filters=256, kernel_size = (3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(BatchNormalization())
model.add(Flatten())
model.add(Dense(512,activation="relu"))
model.add(Dense(10,activation="relu"))

model.compile(loss="CategoricalCrossentropy", optimizer="adam", metrics=["
    CategoricalAccuracy"])

model.summary()

# With data augmentation to prevent overfitting

datagen = ImageDataGenerator(
        featurewise_center=False,  # set input mean to 0 over the dataset
        samplewise_center=False,  # set each sample mean to 0
        featurewise_std_normalization=False,  # divide inputs by std of the
    dataset
        samplewise_std_normalization=False,  # divide each input by its std
        zca_whitening=False,  # apply ZCA whitening
        rotation_range=15,  # randomly rotate images in the range (degrees, 0
     to 180)
        zoom_range = 0.01, # Randomly zoom image
        width_shift_range=0.1,  # randomly shift images horizontally (
    fraction of total width)
        height_shift_range=0.1,  # randomly shift images vertically (fraction
     of total height)
        horizontal_flip=False,  # randomly flip images
        vertical_flip=False)  # randomly flip images


#datagen.fit(X_train)
train_gen = datagen.flow(X_train, y_train, batch_size=128)
test_gen = datagen.flow(X_test, y_test, batch_size=128)
```

```
epochs = 25
batch_size = 128
train_steps = X_train.shape[0] // batch_size
valid_steps = X_test.shape[0] // batch_size

es = keras.callbacks.EarlyStopping(monitor="val_categorical_accuracy", #
    metrics to monitor
                                   patience=10, # how many epochs before stop
                                   verbose=1,
                                   mode="max", # we need the maximum accuracy
    .
                                   restore_best_weights=True, )

# Reduce learning rate when a metric has stopped improving.
# Models often benefit from reducing the learning rate by a factor of 2-10
    once learning stagnates.
# This callback monitors a quantity and if no improvement is seen for a '
    patience' number of epochs,
# the learning rate is reduced.
rp = keras.callbacks.ReduceLROnPlateau(monitor="val_categorical_accuracy",
                                       factor=0.2,
                                       patience=3,
                                       verbose=1,
                                       mode="max",
                                       min_lr=0.00001)

# Fit the model
history = model.fit_generator(train_gen,
                              epochs = epochs,
                              steps_per_epoch = train_steps,
                              validation_data = test_gen,
                              validation_steps = valid_steps,
                              callbacks=[es, rp])

fig, ax = plt.subplots(1, 1, figsize=(7, 7))
ax.plot(history.history['loss'], color='b', label="CCE Training loss",
    linewidth = '2')
ax.plot(history.history['val_loss'], color='r', label="Validation loss",
    linestyle= "dashed", linewidth = '0.8') #,axes =ax[0])

plt.title("Training loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
legend = ax.legend(loc='best', shadow=True)

# Plot the loss and accuracy curves for training and validation

fig, ax = plt.subplots(1, 1, figsize=(7, 7))
ax.plot(history.history['categorical_accuracy'], color='b', label="CCE
    Training accuracy")
ax.plot(history.history['val_categorical_accuracy'], color='r', label="
    Validation accuracy", linestyle= "dashed", linewidth = '0.8') #,axes =ax
    [0])

plt.title("Training Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
legend = ax.legend(loc='best', shadow=True)

# preprocess test data
```

```
test = test.values
test = test / 255.0
test = test.reshape(-1,28,28,1)

# predict test data
pred = model.predict(test, verbose = 1)
classes_x=np.argmax(pred,axis=1)

# submissions
sub['Label'] = classes_x
sub.to_csv("CNN_keras_sub.csv", index=False)
sub.head()
```

C.2: Triplet loss neural network in Python, code from [74]

```
# Imports
import sys
print(sys.executable)
print(sys.version)
print(sys.version_info)

## dataset
from keras.datasets import mnist

## for Model definition/training
from keras.models import Model, load_model
from keras.layers import Input, Flatten, Dense, concatenate,  Dropout
from tensorflow.keras.optimizers import Adam # - Works

from tensorflow.keras.utils import plot_model
from keras.callbacks import ModelCheckpoint

## required for semi-hard triplet loss:
from tensorflow.python.ops import array_ops
from tensorflow.python.ops import math_ops
from tensorflow.python.framework import dtypes
import tensorflow as tf

## for visualizing
import matplotlib.pyplot as plt, numpy as np
from sklearn.decomposition import PCA

import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix

import keras
from keras.models import Sequential
from keras.layers import Conv2D, Lambda, MaxPooling2D # convolution layers
from keras.layers import Dense, Dropout, Flatten # core layers
from tensorflow.keras.layers import BatchNormalization

from keras.preprocessing.image import ImageDataGenerator

from keras.utils.np_utils import to_categorical
from keras.datasets import mnist

# Include functions for Triplet loss
def pairwise_distance(feature, squared=False):
    """Computes the pairwise distance matrix with numerical stability.

    output[i, j] = || feature[i, :] - feature[j, :] ||_2

    Args:
      feature: 2-D Tensor of size [number of data, feature dimension].
      squared: Boolean, whether or not to square the pairwise distances.

    Returns:
      pairwise_distances: 2-D Tensor of size [number of data, number of data
   ].
    """
    pairwise_distances_squared = math_ops.add(
        math_ops.reduce_sum(math_ops.square(feature), axis=[1], keepdims=True
   ),
        math_ops.reduce_sum(
            math_ops.square(array_ops.transpose(feature)),
            axis=[0],
            keepdims=True)) - 2.0 * math_ops.matmul(feature,
                                                    array_ops.transpose(
   feature))

    # Deal with numerical inaccuracies. Set small negatives to zero.
    pairwise_distances_squared = math_ops.maximum(pairwise_distances_squared,
     0.0)
    # Get the mask where the zero distances are at.
    error_mask = math_ops.less_equal(pairwise_distances_squared, 0.0)

    # Optionally take the sqrt.
    if squared:
        pairwise_distances = pairwise_distances_squared
    else:
        pairwise_distances = math_ops.sqrt(
            pairwise_distances_squared + math_ops.to_float(error_mask) * 1e
   -16)

    # Undo conditionally adding 1e-16.
    pairwise_distances = math_ops.multiply(
        pairwise_distances, math_ops.to_float(math_ops.logical_not(error_mask
   )))

    num_data = array_ops.shape(feature)[0]
    # Explicitly set diagonals to zero.
    mask_offdiagonals = array_ops.ones_like(pairwise_distances) - array_ops.
   diag(
        array_ops.ones([num_data]))
    pairwise_distances = math_ops.multiply(pairwise_distances, mask_
   offdiagonals)
    return pairwise_distances

def masked_maximum(data, mask, dim=1):
    """Computes the axis wise maximum over chosen elements.

    Args:
      data: 2-D float ‘Tensor‘ of size [n, m].
      mask: 2-D Boolean ‘Tensor‘ of size [n, m].
      dim: The dimension over which to compute the maximum.

    Returns:
      masked_maximums: N-D ‘Tensor‘.
        The maximized dimension is of size 1 after the operation.
    """
    axis_minimums = math_ops.reduce_min(data, dim, keepdims=True)
    masked_maximums = math_ops.reduce_max(
        math_ops.multiply(data - axis_minimums, mask), dim,
        keepdims=True) + axis_minimums
    return masked_maximums

def masked_minimum(data, mask, dim=1):
    """Computes the axis wise minimum over chosen elements.

    Args:
      data: 2-D float ‘Tensor‘ of size [n, m].
      mask: 2-D Boolean ‘Tensor‘ of size [n, m].
      dim: The dimension over which to compute the minimum.

    Returns:
      masked_minimums: N-D ‘Tensor‘.
        The minimized dimension is of size 1 after the operation.
    """
    axis_maximums = math_ops.reduce_max(data, dim, keepdims=True)
    masked_minimums = math_ops.reduce_min(
        math_ops.multiply(data - axis_maximums, mask), dim,
        keepdims=True) + axis_maximums
    return masked_minimums

# Define Triplet loss
def triplet_loss_adapted_from_tf(y_true, y_pred):
    del y_true
    margin = 1.
    labels = y_pred[:, :1]


    labels = tf.cast(labels, dtype='int32')

    embeddings = y_pred[:, 1:]

    ### Code from Tensorflow function [tf.contrib.losses.metric_learning.
   triplet_semihard_loss] starts here:

    # Reshape [batch_size] label tensor to a [batch_size, 1] label tensor.
    # lshape=array_ops.shape(labels)
    # assert lshape.shape == 1
    # labels = array_ops.reshape(labels, [lshape[0], 1])

    # Build pairwise squared distance matrix.
    pdist_matrix = pairwise_distance(embeddings, squared=True)
    # Build pairwise binary adjacency matrix.
    adjacency = math_ops.equal(labels, array_ops.transpose(labels))
    # Invert so we can select negatives only.
    adjacency_not = math_ops.logical_not(adjacency)

    # global batch_size
    batch_size = array_ops.size(labels) # was 'array_ops.size(labels)'

    # Compute the mask.
    pdist_matrix_tile = array_ops.tile(pdist_matrix, [batch_size, 1])
    mask = math_ops.logical_and(
        array_ops.tile(adjacency_not, [batch_size, 1]),
        math_ops.greater(
            pdist_matrix_tile, array_ops.reshape(
                array_ops.transpose(pdist_matrix), [-1, 1])))
    mask_final = array_ops.reshape(
        math_ops.greater(
            math_ops.reduce_sum(
                math_ops.cast(mask, dtype=dtypes.float32), 1, keepdims=True),
            0.0), [batch_size, batch_size])
    mask_final = array_ops.transpose(mask_final)

    adjacency_not = math_ops.cast(adjacency_not, dtype=dtypes.float32)
    mask = math_ops.cast(mask, dtype=dtypes.float32)

    # negatives_outside: smallest D_an where D_an > D_ap.
    negatives_outside = array_ops.reshape(
        masked_minimum(pdist_matrix_tile, mask), [batch_size, batch_size])
    negatives_outside = array_ops.transpose(negatives_outside)

    # negatives_inside: largest D_an.
    negatives_inside = array_ops.tile(
        masked_maximum(pdist_matrix, adjacency_not), [1, batch_size])
    semi_hard_negatives = array_ops.where(
        mask_final, negatives_outside, negatives_inside)

    loss_mat = math_ops.add(margin, pdist_matrix - semi_hard_negatives)

    mask_positives = math_ops.cast(
        adjacency, dtype=dtypes.float32) - array_ops.diag(
        array_ops.ones([batch_size]))

    # In lifted-struct, the authors multiply 0.5 for upper triangular
    #   in semihard, they take all positive pairs except the diagonal.
    num_positives = math_ops.reduce_sum(mask_positives)

    semi_hard_triplet_loss_distance = math_ops.truediv(
        math_ops.reduce_sum(
            math_ops.maximum(
                math_ops.multiply(loss_mat, mask_positives), 0.0)),
        num_positives,
        name='triplet_semihard_loss')

    ### Code from Tensorflow function semi-hard triplet loss ENDS here.
    return semi_hard_triplet_loss_distance

# Define base model
def create_base_network(image_input_shape, embedding_size):
    """
    Base network to be shared (eq. to feature extraction).
    """
    input_image = Input(shape=image_input_shape)

    x = Flatten()(input_image)
    x = Dense(128, activation='relu')(x)
    x = Dropout(0.1)(x)
    x = Dense(128, activation='relu')(x)
    x = Dropout(0.1)(x)
    x = Dense(embedding_size)(x)
    base_network = Model(inputs=input_image, outputs=x)
    plot_model(base_network, to_file='base_network.png', show_shapes=True,
    show_layer_names=True)
    return base_network

# Loading the training/validation/testing DATA, as well as some other
    parameters
if __name__ == "__main__":
    # in case this scriot is called from another file, let's make sure it
    doesn't start training the network...

    batch_size = 256
    epochs = 25
    train_flag = True  # either     True or False

    embedding_size = 64

    no_of_components = 2  # for visualization -> PCA.fit_transform()

    step = 10

    # The data, split between train and test sets
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    x_train = x_train.astype('float32')
    x_test = x_test.astype('float32')
    x_train /= 255.
    x_test /= 255.
    input_image_shape = (28, 28, 1)
    x_val = x_test[:2000, :, :]
    y_val = y_test[:2000]

# Train the network
# Network training...
if train_flag == True:
        base_network = create_base_network(input_image_shape, embedding_size)

        input_images = Input(shape=input_image_shape, name='input_image') #
    input layer for images
        input_labels = Input(shape=(1,), name='input_label')    # input layer
     for labels
        embeddings = base_network([input_images])               # output of
    network -> embeddings
        labels_plus_embeddings = concatenate([input_labels, embeddings])  #
    concatenating the labels + embeddings

        # Defining a model with inputs (images, labels) and outputs (labels_
    plus_embeddings)
        model = Model(inputs=[input_images, input_labels],
                      outputs=labels_plus_embeddings)

        model.summary()
        plot_model(model, to_file='model.png', show_shapes=True, show_layer_
    names=True)

        # train session
        opt = Adam(lr=0.0001)  # choose optimiser. RMS is good too!

        model.compile(loss=triplet_loss_adapted_from_tf,
                      optimizer=opt, metrics=['accuracy'])
        filepath = "semiH_trip_MNIST_v13_ep{epoch:02d}_BS%d.hdf5" % batch_
    size
        #checkpoint = ModelCheckpoint(filepath, monitor='val_loss', verbose
    =1, save_best_only=False, period=25)

        checkpoint = [ModelCheckpoint(filepath, monitor='val_accuracy',
    verbose=1, save_best_only=False, period=25),
                      ModelCheckpoint(filepath, monitor='val_loss', verbose=1,
     save_best_only=False, period=25)]

        callbacks_list = [checkpoint]

        # Uses 'dummy' embeddings + dummy gt labels. Will be removed as soon
    as loaded, to free memory
        dummy_gt_train = np.zeros((len(x_train), embedding_size + 1))
        dummy_gt_val = np.zeros((len(x_val), embedding_size + 1))

        x_train = np.reshape(x_train, (len(x_train), x_train.shape[1], x_
    train.shape[1], 1))
        x_val = np.reshape(x_val, (len(x_val), x_train.shape[1], x_train.
    shape[1], 1))

        H = model.fit(
            x=[x_train,y_train],
            y=dummy_gt_train,
            batch_size=batch_size,
            epochs=epochs,
            validation_data=([x_val, y_val], dummy_gt_val),
            callbacks=callbacks_list)

        plt.figure(figsize=(8,8))
        plt.plot(H.history['loss'], label='training loss')
        plt.plot(H.history['val_loss'], label='validation loss')
        plt.legend()
        plt.title('Train/validation loss')
        plt.show()


else:

        #####
        model = load_model('semiH_trip_MNIST_v13_ep25_BS256.hdf5',
                                custom_objects={'triplet_loss_adapted
    _from_tf':triplet_loss_adapted_from_tf})

# Plot accuracy
plt.figure(figsize=(8,8))
plt.plot(H.history['accuracy'], label='training accuracy')
plt.plot(H.history['val_accuracy'], label='validation accuracy')
plt.title('model accuracy')
plt.legend()
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```