BACHELOR'S DEGREE THESIS

Degree in Statistics

Title:
Dimensionality reduction for clustering with deep neural networks

Author: Agustin Fernández Felguera

Advisor: Ferran Reverter and Esteban Vegas

Department: Genetics, Microbiology and Statistics

Academic year: 2019-20

UNIVERSITAT DE BARCELONA

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat de Matemàtiques i Estadística

Bachelor's Degree Thesis

Degree in Statistics

Dimensionality reduction for clustering with deep neural networks

September 1, 2020

Author: Agustin Fernández Felguera

Advisors: Ferran Reverter Comes and Esteban Vegas Lozano

Facultat d'Economia i Empresa

Facultat de Matemàtiques i Estadística

Abstract

Nowadays, high dimensional data is ubiquitous: you can think for example in images, videos or texts. Unfortunately, this property can harm seriously the performance of some algorithms. In this project, I analyse how dimensionality reduction can help clustering improve its performance. In order to do that, I distinguish three different clustering strategies: Traditional, two-stages and deep clustering. In the first one, the clustering is applied to the raw data while in the other two it is applied to a low dimensional representation. I focus especially on the latter approach, which has shown promising performance in the last years. The differences between these approaches are illustrated doing a series of experiments and visualisations and comparing the results.

Keywords: Artificial intelligence, Clustering, Dimensionality reduction, Autoencoders, Deep clustering.

AMS2010: Classification and discrimination; cluster analysis (**62H30**)
Neural nets and related approaches (**62M45**)

Resum

Actualment, les dades d'alta dimensió són omnipresents: es pot pensar, per exemple, en imatges, vídeos o textos. Malauradament, aquesta propietat pot perjudicar greument el rendiment d'alguns algorismes. En aquest projecte, analitzo com la reducció de dimensionalitat pot ajudar a que el *clustering* millori el seu rendiment. Per fer-ho, distingeixo tres estratègies de clusterització diferents: la tradicional, la de dues etapes i el *Deep clustering*. En la primera, l'agrupació s'aplica a les dades brutes mentre que en les altres dos s'aplica a una representació de baixa dimensió. En el treball em centro especialment en aquest últim enfocament, que ha demostrat un rendiment prometedor en els darrers anys. Les diferències entre aquestes estratègies s'il·lustren fent una sèrie d'experiments i visualitzacions i comparant els resultats.

Paraules clau: Intelligència artificial, Clusterització, Reducció de la dimensionalitat, Autoencoders, Deep clustering.

Contents

Page

1 Introduction 1

2 Main concepts 4

I Dimensionality reduction 7

3 Overview 8

3.1 Key concepts 9

3.2 The curse of dimensionality 10

4 Principal Component Analysis 13

4.1 How does it work 13

4.2 Advantages 14

4.3 Limitations 14

5 t-Stochastic Neighbor Embedding 16

5.1 Calculating distances in the high and low dimensional spaces 17

5.2 Objective function: KL divergence 20
5.3 Gradient descent 21
5.4 Weaknesses of the algorithm 22
5.5 Uniform Manifold Approximation and Projection (UMAP) 23

6 Artificial neural networks **24**

6.1 Elements of an Artificial Neural Network 25
6.1.1 Activation functions 27
6.2 Training 28
6.2.1 Loss functions 28
6.2.2 Optimisation 29
6.2.3 Complete training cycle 30
6.3 Convolutional Neural Networks 31
6.4 Building a neural network 33
6.5 Autoencoders 35
6.5.1 Advantages of Autoencoders for DR 36

II Clustering **38**

7 Overview **39**

7.1 Classification 39
7.2 Evaluation 40
7.2.1 Metrics 41

7.3 Why dimensionality reduction can be useful for clustering . . . 43
7.3.1 Clustering strategies using DR . . . 44

8 k-means **46**

8.1 How does it work . . . 47
8.2 Advantages and limitations . . . 47

9 Deep Clustering **48**

9.1 Deep convolutional embedding clustering algorithm (DCEC) . . . 49
9.1.1 Pretraining with the CAE . . . 50
9.1.2 Training stage . . . 51
9.1.3 Parameter initialisation . . . 52
9.1.4 Soft assignment . . . 52
9.1.5 KL divergence minimisation . . . 53
9.1.6 Optimisation . . . 53

III Applied analysis **54**

10 Methodology **55**

10.1 Description of the data . . . 55
10.1.1 Digital images . . . 55
10.1.2 Datasets . . . 56
10.2 Deep Learning software . . . 59
10.2.1 Python . . . 59

10.2.2 Tensorflow 59
10.2.3 Keras 59
10.3 Cloud computing 60
10.4 Open code 61

11 Applied dimensionality reduction **63**
11.1 Fashion dataset 64
11.1.1 Traditional algorithms 64
11.1.2 Autoencoders 68
11.2 MNIST dataset 76
11.3 Cifar 10 79
11.4 Discussion of the results 80
11.4.1 Global and local structure preservation 80

12 Applied clustering **82**
12.1 Clustering with the original data 83
12.1.1 K-means 83
12.1.2 HDBSCAN 83
12.2 Clustering with the reduced spaces 84
12.3 Evaluation of the clustering results 87
12.3.1 Fashion dataset 87
12.3.2 All the datasets 88
12.4 Discussion of the results 89

13 Applied Deep Clustering **91**

13.1 DCEC Implementation 91

13.1.1 DCEC training 93

13.1.2 Discussion of some improvements 97

13.2 Final comparison of results 98

Conclusions **99**

References **101**

Appendix **105**

13.3 Fashion code 106

List of Figures

2.1 AI Classification (Source: Modified from Qubole.com) 6

3.1 High dimensions are more sparse (Source: [4]) 11

6.1 General structure of a node (Source: cs231n.stanford.edu/slides) 26

6.2 Example of ANN (Source: VIASAT) 26

6.3 Max pooling layer (Source: cs231n.github.io) 32

6.4 AlexNet CNN (Source: Neurohive.io) 33

6.5 Structure of an autoencoder (Source: [17]) 36

9.1 Representation of joint learning in an autoencoder 48

10.1 Representation of digit 8 - MNIST Dataset (Source: [30]) 57

10.2 Example of some samples of the Fashion Dataset 58

10.3 Example of Cifar10 RGB image 58

10.4 Functionality and easy of use relationship (Source: pyimagesearch.com) 60

10.5 Google colaboratory system 61

11.1 Visualisation by PCA - Fashion 64

11.2 Visualisation by UMAP - Fashion 66
11.3 Visualisation by tsne - Fashion 67
11.4 Input layer (Source: ml4a.github.io) 68
11.5 Encoder of the CAE 70
11.6 Decoder of the CAE 70
11.7 Evolution of the losses 71
11.8 Random samples reconstructed - Fashion 72
11.9 Visualisation by CAE - Fashion 73
11.10Visualisation by AE128d and UMAP - Fashion 73
11.11Visualisation by CAE Guo et.al. - Fashion 74
11.12CAE proposed by Xifen Guo et.al. 75
11.13Visualisation by PCA - MNIST 76
11.14Visualisation by t-SNE - MNIST 77
11.15UMAP: (n_neighbors= 15, min_dist=0.1) - MNIST 77
11.16UMAP: (n_neighbors= 30, min_dist=0) - MNIST 78
11.17Visualisation by CAE - MNIST 78
11.18Visualisation by umap - Cifar 79
11.19Structure preservation - MNIST 81

12.1 K-means cluster's centres 83
12.2 UMAP and K-means 86
12.3 HDBSCAN with 2D UMAP 86

12.4 HDBSCAN with 30D UMAP 86

12.5 K-means with CAE output - MNIST 90

13.1 DCEC model from Guo et.al. 95

13.2 Training process DCEC model from Guo et.al. 96

13.3 Visualisation of the embedding layer 96

List of Tables

12.1 Quantitative clustering performance 87

12.2 Quantitative clustering performance (NMI) 88

13.1 Deep clustering performance 98

Chapter 1

Introduction

It is well known that in the last decade there has been an explosion of the volume of data available. But it's not just that. Today, data also comes from a wide variety of formats such as images, texts or sensors among many others. These new formats also known as unstructured data have usually a very high number of dimensions and are much more complex and heterogeneous than traditional tabular data. For example, the pixel intensity vectors used to represent images have usually thousand of dimensions.

This is not a trivial problem, but rather a fundamental obstacle in empirical science: the number of variables to measure can be unwieldy and at times even deceptive, because the underlying relationships or *intrinsic structure* can often be quite simple. This complexity can create bottlenecks in the data analysis process and can worsen the performance of many algorithms that are affected by the *curse of dimensionality*.

Therefore, it is increasingly important the extraction and discovery of the knowledge hidden in the raw data, a process known as unsupervised learning. Right now, machine learning research is mainly about making computers understand data the same way humans do. But the focus is going to shift to getting computers to understand things that humans aren't able because of the vast amount of data that have to be processed.

Here is when comes into play dimensionality reduction. This field can help a lot when it comes to removing noise, since it reduces the number of dimensions of the data with a minimum loss of information. It is useful to visualise the structure of the data, to save computer resources or simply to get new insights. Furthermore, dimensionality reduction can be used to improve the performance of another tasks such as clustering, as they don't have to be independent to each other. By combining both tasks, either in separate ways, or jointly as a single algorithm, it is possible to achieve better results that if only clustering is used.

Aim of the project

The objective of this project is to explore how dimensionality reduction can be used to improve clustering results. In order to do this, three different clustering strategies should be analysed:

Traditional clustering: It is the use of a clustering algorithm alone, without additional help.

Two-stages clustering: It consists in using a dimensionality reduction algorithm to reduce the dimensionality of the data before applying clustering.

Deep clustering: This category uses deep neural networks to perform jointly, or at the same time, dimensionality reduction and clustering. Here I focus on analysing a clustering algorithm called Deep Convolutional Embedding Clustering (DCEC) that is based mainly on the dimensionality reduction induced by an autoencoder.

I analyse how these three strategies perform by clustering image data from different datasets and comparing the results.

Through the development of this Thesis, I pretend to accomplish the following goals:

- Introduce the literature corresponding to dimensionality reduction and clustering.
- Provide a general overview of how these techniques perform on images.
- Use dimensionality reduction to visualise the structure of the data and detect clusters.
- Use deep neural networks as a method of dimensionality reduction.
- Implement a Deep clustering algorithm called DCEC.
- Compare Deep clustering methods with traditional statistical techniques.

Structure

This bachelor's degree thesis is structured into two main parts, one theoretical and the other practical.

In the first part I present all the theory underneath the algorithms I want to apply later. I start with a general overview of the field of dimensionality reduction to go on to explain different algorithms, paying especial attention to neural networks. Next I provide a description of clustering analysis and how these two fields are interrelated. Lastly, I present the DCEC, a Deep Clustering algorithm.

In the second part I focus on the different applications of the theory and I do some experiments. This part is form by four chapters. In the first one, the methodology, I describe the data sources, the software and the analysis process used. In the second one, I explore how different dimensionality reduction techniques perform on different dataset by looking mostly at the two dimensional representations and how useful they can be at detecting clusters. In the third chapter, I perform clustering both in the raw data and in the low dimensional spaces generated previously. Lastly, I apply the DCEC algorithm to different datasets and I analyse its results with more traditional clustering techniques.

Summary of the methodology

Data The thesis focuses on a specific data type: images. In the applied section I use three datasets with different characteristics. Two datasets have only black and white images, while the other one has colour images of bigger size.

Coding language Even though the language R has good machine learning packages the thesis is done using Python. There are mainly two reasons for this: First, the cloud computing service used only provides support for python and second, the integration of Python with other Deep Learning frameworks is excellent.

Deep Learning software In recent years the software applied to Deep Learning has evolve extraordinarily fast. I take advantage of these developments by using the library Keras and the platform Tensorflow to implement Deep Learning algorithms in a relatively easy way.

Cloud computing Cloud computing allows you to access to computing power in an affordable way. I use Google's cloud computing environment Colab because it is free and easy to use. This environment allows you to execute code on Google's cloud servers, meaning you can leverage the power of Google's hardware, including GPUs and TPUs, regardless of the computer you are using.

Open code All of the algorithms used in the thesis were already implemented and in almost all of them, the source code was available. This code in most of the cases could be found in Github, a platform that allows hosting and sharing your code very easily.

Chapter 2

Main concepts

First of all, it is important to put in context some of the concepts that will be used throughout the work. Many of them are not new, but have been used in statistics for decades. They are only called differently[1]. Others, more computed-related, are very recent and have started to be used in the industry and academia only a few years ago when technological conditions have started to allow it.

Artificial intelligence (AI): While the roots are long and deep, the history of artificial intelligence as we think of it today spans less than a century. It is the science of mimicking human abilities, namely, being able to carry out tasks in a way that we would consider "smart". It is an interdisciplinary field with multiple approaches, but the two more important are Machine Learning and Deep Learning. They have receive much attention in recent years.

Machine Learning (ML) is a branch of AI that automates analytical model building. It uses methods from statistics, operations research or computer science to find hidden insights or patterns in data without being explicitly programmed where to look or what to conclude. It is based around the idea that we should really just be able to give machines access to data and let them learn for themselves. Although this field is very linked to statistics there are several fundamental differences between them: While ML mainly focus on prediction for large-scale problems (usually for the short term), statistics also deals with estimation (based on surface plus noise models) and attribution (significance of the variables) [1].

Currently machine learning presents three major trends or paradigms used to train models:

[1]Some remarkable examples of name changes include the use of feature instead of variable, input instead of explanatory variable or output/target instead of independent variable.

Reinforcement learning Takes place when a machine learns through trial and error until it finds the best way to complete a given task. The system learns through its mistakes to modify its behaviour based on "rewards" for completing the assigned task, without being specifically programmed to do it in a certain way. It is a very recent trend that still needs research to enter into production.

Supervised learning Is the most used paradigm and occurs when machines are trained with labelled data. For example, images with descriptions of the things that appear in them. The algorithm the machine uses is able to select these labels in other databases. Therefore, if a group of images has been labelled that show dogs, the machine can identify similar images.

Unsupervised learning Is a type of machine learning algorithm used to draw inferences from datasets consisting of input data without labelled responses. Within this group we can find among others the tasks of **Dimensionality Reduction** [2] and **Clustering**. Clustering algorithms are not programmed to detect a specific type of data, but to look for examples that are similar and can be grouped together.

Deep learning (DL): Is a particular subfield of ML that uses very big **artificial neural networks** (ANN) to train models. It takes advantage of advances in computing power and improved training techniques to learn complex patterns in large amounts of data. Common applications include image and speech recognition (supervised learning), but it can also be used for dimensionality reduction with the use of **autoencoders** or for clustering. Conceptually the main advantage of deep learning over machine learning is that the former advocates to solve the problem end-to-end, simplifying the whole process, while ML algorithms usually break the problem down into different parts, solve them individually and combine them to get the result. In practice though, the most important difference between deep learning and traditional machine learning techniques is its performance as the scale of data increases. When the amount of data is very large (**big data**), usually the performance of a neural network is significantly better than that of a machine learning algorithm.

Lastly, within the field of deep learning and as a form of unsupervised learning we can define Deep Clustering, one of the main topics of this thesis:

Deep clustering is the process of learning deep feature representations that favour the clustering task using deep neural networks. The most promising algorithms in this subfield are a form of **joint unsupervised learning** in the sense that they try to simultaneously learn a nice latent representation and perform clustering.

[2] Although it is not widely used, dimensionality reduction can also be supervised. Some examples of supervised dimension reduction are Linear discriminant analysis (LDA) or some types of neural networks with a mid-bottleneck layer.

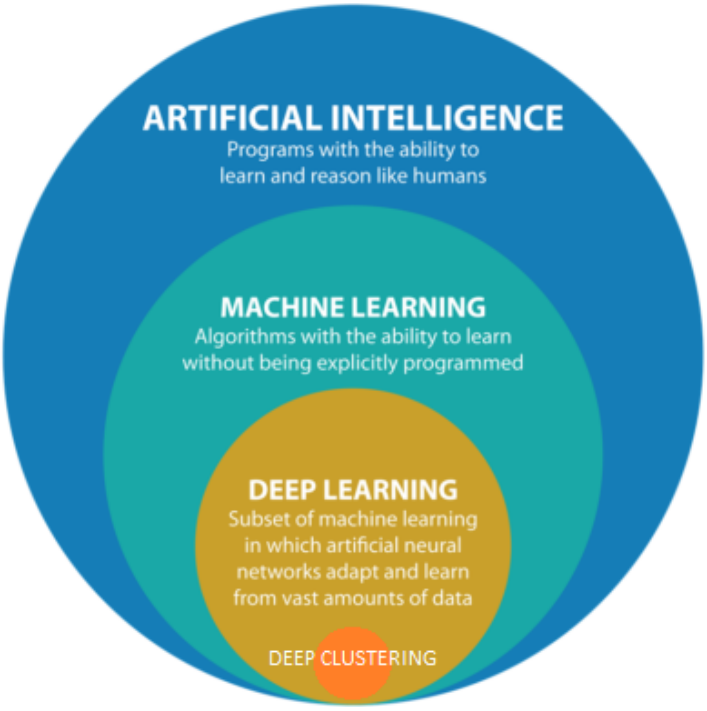


Figure 2.1: AI Classification (Source: Modified from Qubole.com)

Part I

Dimensionality reduction

Chapter 3

Overview

In the field of machine learning dimensionality reduction approaches can be divided into feature selection and feature extraction.

- **Feature extraction**[1]: Features are projected into a new space with lower dimensionality.
- **Feature selection**: The aim is to select a small subset of features that minimise redundancy and maximise relevance to the target (e.g. class label). Popular feature selection techniques include: Information Gain, Chi Squares, Fisher Score, and Lasso, to name a few.

In this thesis, when talking about dimensionality reduction I will always make reference to feature extraction.

Dimensionality reduction or feature extraction is the transformation of high-dimensional data into a meaningful representation of reduced dimensionality. Its aim is to preserve as much of the significant structure of the original data as possible in the low-dimensional map. In other words, we want the original space $\mathcal{X} = \{x_1...x_n\}$ of dimensions (n , p) and the embedded space $\mathcal{Y} = \{y_1...y_n\}$ of dimensions (n , q) to be as *similar* as possible. Ideally, the reduced representation should have a dimensionality that corresponds to the **intrinsic dimensionality of the data**, which is the minimum number of parameters needed to account for all the observed properties of the data.

This field is not new. In fact, one of the most widely used dimensionality reduction techniques,

[1]Feature extraction is also called feature projection. Additionally, in the field of machine learning people use the term *features* instead of variables.

Principal Component Analysis (PCA), dates back to Karl Pearson in 1901. It has been a long-standing research topic in academia and industry for two major reasons. First, the increasingly large volume of data is challenging constantly the existing computing capability. Second, the notion of *intrinsic structure* allows us to remove some redundant dimensions facilitating, among others, classification, visualisation, and compression of high-dimensional data [2].

However, in recent years new tools, more sophisticated, have been discovered. Some of them are spectral methods, t-SNE (2008), LargeVis (2016), UMAP (2018) or Autoencoders (that took off in the last decade). Their main advantage is that they can reduce well complex data as they are non-linear techniques and thus, are more flexible than the traditional ones.

3.1 Key concepts

When using dimensionality reduction techniques you have to pay attention to several concepts. Next I will provide an overview of some of them:

How the original data is structured A key idea in dimensionality reduction is that if the data lies in a q-dimensional (q << p) subspace of the p-dimensional space, and if we can identify the subspace, then there exists a transformation which loses no information and allows the data to be represented in this q-dimensional space. If the data lies in a (linear) subspace then the transformation is linear; instead, if the data lie in a q-dimensional (curved) manifold, the transformation is non-linear. For real world data, nonlinear dimensionality reduction techniques may offer an advantage, because real world data is likely to form a highly nonlinear manifold [2, p.1]. Linear techniques make a more restrictive assumption but in return they are usually more intuitive and computationally faster.

Interpreatibility When interpreting the embedded space one must always be cautious as some algorithms can lead to erroneous interpretations. The axes of the embedded space can be interpreted in some algorithms such as factor analysis or PCA (where the dimensions are the directions of greatest variance in the source data), but with others they have no inherent meaning. Distances between datapoints are often very dependent on the chosen parameters and can be misleading as well. For example, some techniques allow you to see the presence of clusters in the data, but the distance between clusters can have no meaning. This is especially true for autoencoders since you cannot do any interpretation of the low dimensional space at all.

How well the distance structure is preserved within the data Regarding structure preservation, algorithms tend to fall into two categories: those that seek to preserve geometry at all scales within the data (global structure) and those that favour the preservation

[2] It is not clear though whether the different tasks (e.g. reduce computational load, condense the number of variables, visualisation etc.) can share a similar solution or a different tool is needed for each task.

of local distances over global distances (local structure). In order to form good clusters is especially important to preserve correctly the global structure of the data in order to have meaningful distances between groups. You want techniques that preserve the topology of the data. However, according to [3], local approaches have two principal advantages: i) computational efficiency: they involve only sparse matrix computations which may yield a polynomial speedup; ii) representational capacity: they may give useful results on a broader range of manifolds, whose local geometry is close to Euclidean, but whose global geometry may not be.

Objective function can be convex or non-convex In the latter case the algorithms can get stuck in local optima, getting this way a suboptimal result. However, in many occasions suboptimally optimising a sensible objective function is a more viable approach than optimising a convex objective function that contain obvious flaws [2, p.25]. This is due to the fact that in the design of a non-convex technique, there is much more freedom to construct a sensible objective function.

How important are the hyperparameters Some techniques have a lot of hyperparameters and their final result can vary widely depending on the chosen values. Moreover, often it is difficult to decide objectively which are these values.

3.2 The curse of dimensionality

The curse of dimensionality refers to all the problems that arise when working with high dimensional data, that did not occur in low dimensional spaces. Some of these problems are **data sparsity, multicollinearity, multiple testing, overfitting, numerical instability or computing costs, among others**[4]. In general they are alike and interconnected. The problem is that data analysis tools are most often designed having in mind intuitive properties and examples in low-dimensional spaces [5], but these properties can be very different in high dimensional spaces. The underlying problem is that we just have no way to use intuition accurately in higher dimensions; we have to rely exclusively on mathematics.

- As dimensionality grows there are less observations per region. The volume of the space increases so fast that the available data become sparse. You can see that effect in figure 3.1, which shows the distribution of points within a region in 1 and 2 dimensions. Additional dimensions spread out the points until, in very high dimensions, they are almost equidistant from each other, hence distance and similarity measures become increasingly meaningless and unreliable, completely masking clusters and other structures.

  But it is not just that the amount of space goes up, but that it's arranged in a particular way. How points are distributed at higher dimensions are not the same as in one dimension.

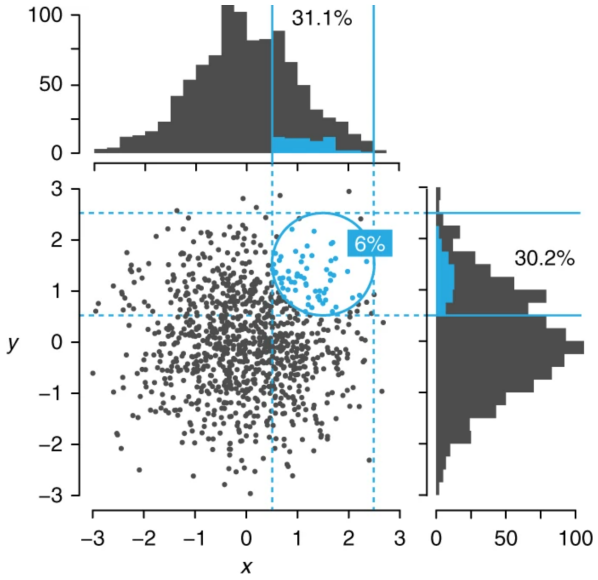


Figure 3.1: High dimensions are more sparse (Source: [4])

- As dimensionality increases, there is simply more space that has to be searched for many combinatorial problems. The number of parameters to be estimated increases exponentially as the number of variables included in a model increases.
- In some fields, it is easy to encounter a situation in which the number of variables is higher than the number of observations (n << p). Then, we have perfect multicollinearity: we can always express at least one of the variables as a linear combination of the others, thus yielding perfect multiple correlation.
- The number of samples needed to estimate an arbitrary function with a given level of accuracy grows exponentially with respect to the number of input variables (i.e., dimensionality) of the function.

Solutions Increasing the number of observations is the obvious way of tackling the issue, although this method is not often viable in practice. Working from the opposite direction and reducing the number of dimensions directly, often appears to be a more pragmatic solution, although dimensionality reduction techniques may themselves be affected by the curse of dimensionality.

In this thesis I will use mainly three different techniques:

- **PCA**: It is widely used due to its simplicity and intuitiveness. It will be useful as a benchmark.

- **t-SNE**: It is the current state-of-the-art algorithm for dimension reduction for visualisation but it has also been applied successfully for clustering purposes. The reason is its ability to analyse nonlinearities and its capability of adapting to the local and global structure of the data.
- **Autoencoder**: It is a completely new concept since it is based in an artificial neural network. It is very flexible and can be adapted to different kinds of data such as images (Convolutional Autoencoder).

Moreover, I will study how these algorithms can help detect and form clusters.

Chapter 4

Principal Component Analysis

Principal component analysis (PCA) is one of the oldest and most popular dimensionality reduction techniques. It's the first tool that use most of the researchers to get a sense of their data before applying other more complex methods. They tend to start with PCA to, among other things, visualise the relationship between data points, see which are the most representative variables, or understand the intrinsic dimensionality of the data.
In this thesis we will use PCA mostly as a benchmark to compare with more advanced methods.

4.1 How does it work

PCA finds the directions of maximum variance in the high-dimensional data and project it onto a smaller dimensional subspace while retaining most of the information. One way to think of it, there are many, is with an optimisation problem that seeks a *reconstructed* matrix as similar as possible to the original:

$$Min \sum_{i=1}^{n} \sum_{j=1}^{p} (X_{ij} - (YZ)_{ij})^2$$

The loss function uses the mean squared error (i.e. L2 norm) to measure how similar is the resulting matrix (YZ) from the original data.

$X \in \mathbb{R}^{nxp}$ is the original data matrix with n observations and p variables. $Y \in \mathbb{R}^{nxq}$ is the new representation of the data and $Z \in \mathbb{R}^{qxp}$ is the matrix that projects the data into the new

representation. Since usually $q << p$, the new space will have a much lower dimensionality.

This problem can be solved using classic lineal algebra: You substract the mean of the data, compute the covariance matrix and then, extract the eigenvectors and eigenvalues of this matrix. Finally, in order to reduce the dimensions of the data, you must select a subset of the eigenvectors to project the data into the new (sub)space.

4.2 Advantages

PCA is widely applied mostly because it is simple and easy to use, and produces good enough results. Some of its strengths are the following:

- Is completely non-parametric. This means that the answer is unique and independent of the user. Hence, it is often used as a black box.
- Has an analytical solution and the objective function is convex.
- It's possible to interpret the low dimensional space. You can even "attach" a name to each principal component.
- The algorithm allows to quantify the importance of each dimension to describe the variability of a data set. Provides a means for comparing the relative importance of each dimension.
- PCA is particularly useful when strong correlation between variables exists, since it creates a space of uncorrelated and orthogonal variables.
- It is computationally fast compared with other algorithms.

4.3 Limitations

At the same time, the algorithm makes several **assumptions**. This section provides a brief summary of some of them.

- Is a linear dimensionality reduction technique. When doing PCA you are asking: Is there another basis, which is a linear combination of the original basis, that best expresses our data set?

- Large variances have important structure. Hence, principal components with larger associated variances represent interesting structure, while those with lower variances represent noise.
- The principal components are orthogonal. This assumption provides an intuitive simplification that makes PCA soluble with linear algebra decomposition techniques.

These assumptions are not true in many cases, especially that of linearity. Apart of this, PCA is non-parametric so is not as flexible as another techniques -in the sense that you cannot add extra information to the algorithm-, and it tends to be highly affected by outliers in the data because of the way the loss function is made (i.e. quadratic function).

Chapter 5

t-Stochastic Neighbor Embedding

T distributed-Stochastic Neighbor Embedding (**t-SNE**) was introduced in 2008 by Laurens van der Maaten and Geoffrey Hinton [6] and has become a very popular technique since then. In particular, it the state-of-the-art algorithm for the visualisation of the structure of images and texts in two and three dimensions. But it also seems to reduce very well the dimensionality of images for clustering purposes. It has been proved that t-SNE is able to recover well-separated clusters [7].

The algorithm belongs to the class of methods known as **manifold learning**. They seek to describe datasets as low-dimensional manifolds embedded in high-dimensional spaces. The advantage of these methods as opposed to more traditional techniques is that they have the ability to preserve non linear relationships in the data. Other property of t-SNE is that it is capable of capturing much of the local structure of the high-dimensional data very well, while also revealing global structure such as the presence of clusters.

The **process of the algorithm** is the following:

1. Compute pairwise *similarities* between points in the original space using a Gaussian kernel.
2. Compute pairwise *similarities* between mapped points in the embedded space using a t-Student kernel.
3. Use some objective function to measure the *discrepancy* between similarities in the data and similarities in the map and derive the gradient.
4. Iterate the process in order to continue minimising the cost function. This is usually done using gradient descent with momentum.

In practice, the user has to set four hyperparameters basically. According to the terminology of Python's framkework *sklearn* these are n_components, perplexity, learning_rate and n_iter [sklearn.manifold.TSNE]. A brief definition together with the recommended range of values is provided next:

- **n_components**: The number of dimensions of the embedded space. We usually want two or three dimensions to plot the points in a scatterplot.
- **perplexity**: Is a parameter related to the number of nearest neighbours you want to use in the calculations. Larger datasets usually require a larger perplexity. Typical values are between 5 and 50. This parameter has a complex effect on the resulting embedded space.
- **learning_rate**: The parameter used in the update of the mapped points during the iterations of gradient descent.
- **n_iter**: The number of iterations for the optimisation of the cost function.

Below, I explain in detail all the steps of the algorithm and what is the relevance of each hyperparameter.

5.1 Calculating distances in the high and low dimensional spaces

To capture the structure of the data most of the dimensionality reduction methods need a measure of distance. What this algorithm does is first calculate the Euclidean distance and then transform it into probabilities representing similarities. This is done both in the high-dimensional space $\mathcal{X} = \{x_1...x_n\}$ of dimensions (n , p) and in the low-dimensional space $\mathcal{Y} = \{y_1...y_n\}$ of dimensions (n , q).

Similarities in the original space

In the high-dimensional space the probabilities are calculated using a Gaussian Kernel with mean x_i and variance σ_i. The similarity of datapoint x_j to datapoint x_i is represented with the conditional probability $p_{j|i}$. That is to say, the probability that x_i would pick x_j as its neighbour if neighbours were picked in proportion to their probability density under a Gaussian centered at x_i.

$$p_{j|i} = \frac{exp(-\|x_i - x_j\|^2/2\sigma_i^2)}{\sum_{k \neq i} exp(-\|x_i - x_k\|^2/2\sigma_i^2)} \tag{5.1}$$

For nearby datapoints, $p_{j|i}$ is relatively high, whereas for widely separated datapoints, $p_{j|i}$ will be almost infinitesimal since the tails of the Normal distribution are very small. It also should be noted that this formula is assuming local linearity on the manifold since the Euclidean distance is being used in the numerator ($\|x_i - x_k\|^2$).
Lastly, the conditional probability $p_{i|j}$ is calculated in the same way.

Since we don't want asymmetric probabilities in the objective function afterwards, we have to calculate the joint probabilities p_{ij}. That is achieved doing a sort of average between the two conditional probabilities.

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n} \tag{5.2}$$

Similarities in the embedded space

In the low-dimensional map, it is necessary to use a probability distribution that has much heavier tails than a Gaussian to convert distances into probabilities. This allows a moderate distance in the high-dimensional space to be faithfully modelled by a much larger distance in the map and, as a result, it eliminates the unwanted attractive forces between map points that represent moderately dissimilar datapoints. Usually a Student t-distribution with one degree of freedom is used, but a different number of degrees of freedom could be adequate as well.

$$q_{ij} = \frac{(1 + \|y_i - y_j\|^2/\alpha)^{-\frac{\alpha+1}{2}}}{\sum_{k \neq l}(1 + \|y_k - y_l\|^2/\alpha)^{-\frac{\alpha+1}{2}}} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_{k \neq l}(1 + \|y_k - y_l\|^2)^{-1}} \tag{5.3}$$

Where α are the degrees of freedom of the Student's t distribution. As said, the usual value is 1. On the other hand, since we are using a t-Student kernel, σ_i is not needed.

Gaussian Kernel The Gaussian kernel is used to compute the conditional probabilities in the original space. Therefore, it's worth mentioning some of its characteristics. Since the value of the kernel decreases with distance and ranges between zero (in the limit) and one (when $\boldsymbol{x} = \boldsymbol{x'}$), it has a ready interpretation as a similarity measure.

$$K(\boldsymbol{x}, \boldsymbol{x'}) = exp(\frac{-\|\boldsymbol{x} - \boldsymbol{x'}\|^2}{2\sigma^2}) \tag{5.4}$$

The adjustable parameter sigma plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand. If overestimated, the exponential will behave almost linearly and the higher-dimensional projection will start to lose its non-linear power. In the other hand, if underestimated, the function will lack regularisation and the decision boundary will be highly sensitive to noise in training data.

How to determine σ_i

The parameter σ_i is set in such a way that the conditional probability $p_{j|i}$ mentioned earlier has a fixed perplexity. The perplexity is decided by the user and is the same for all the observations. This hyperparameter can be interpreted as a smooth measure of the effective number of neighbours. It says (loosely) how to balance attention between local and global aspects of your data.

$$Perplexity(P_i) = 2^{H(P_i)}$$

where H(P_i) is the Shannon entropy of P_i measured in bits

$$H(P_i) = -\sum_j p_{j|i} \cdot log_2(p_{j|i})$$

If the data is sparse the user should set a high perplexity value, meaning the value of σ_i will be high and it will give higher probabilities to more distant neighbours.

Information theory measure: Shannon Entropy

It's used to quantify the amount of uncertainty in an entire probability distribution. In other words, the Shannon entropy of a distribution is the expected amount of information in an event drawn from that distribution.

$$H(p_i) = -\sum_i p_i \, log_b(p_i)$$

Where b is usually 2 (when we talk about *bits*) or e (for *nats*).
The logarithm is useful because it allows probable events to provide a low information content in addition to making independent events have additive information

5.2 Objective function: KL divergence

The key of the algorithm is that if the map points y_i and y_j correctly model the similarity between the high-dimensional datapoints x_i and x_j, then, the conditional probabilities p_{ij} and q_{ij} will be very similar. In other words, t-SNE aims to find a low-dimensional data representation that minimizes the mismatch between P and Q. To measure the validity of this point the Kullback-Leibler divergence is used. Then, the cost function is given by the sum of KL divergences over all datapoints:

$$Cost = D_{KL}(P||Q) = \sum_i \sum_j p_{ij} \cdot log(\frac{p_{ij}}{q_{ij}}) \tag{5.5}$$

It's important to note that we are computing the divergence from Q to P and not the other way around. Since it's a non-symmetrical measure this fact has consequences, one of them is that points in the embedding space tend to go to the center. This is contrarrested by using the t-Student kernel to measure similarities in the embedded space as it gives more weight to distant points (explained in subsection 5.1).

Information theory measure: Kullback-Leibler Divergence

Is a standard function to measure how different two distributions are.
It is the expectation of the logarithmic difference between the probabilities P and Q, where the expectation is taken using the probabilities P.
$D_{KL}(P||Q) = \sum_{x \in X} P(x) \cdot log(\frac{P(x)}{Q(x)})$

$D_{KL}(P||Q)$ Is the divergence from Q to P. It can also be considered the information gain achieved if Q is used instead of P.
Properties:

- $D_{KL} = 0$ if and only if $P(x) = Q(x)$
- It's not a true distance since it's not symmetrical: $D_{KL}(P||Q) \neq D_{KL}(Q||P)$ for some P and Q.
  This asymmetry means that there are important consequences to the choice of whether use $D_{KL}(P||Q)$ or $D_{KL}(Q||P)$.
- It's non-negative and the value of the divergence increases as P(x) and Q(x) become more different.
- It's well-defined for discrete and continuous distributions alike.

5.3 Gradient descent

Derivative of the Cost function with respect to the mapping points

$$\frac{\partial C}{\partial y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)(1 + \|y_i - y_j\|^2)^{-1} \tag{5.6}$$

The derivative of the cost function, that is, the KL divergence, depends on the difference between similarities in the high and in the low dimensional representations and the difference between embedded points.

Update of the mapping points

$$\mathcal{Y}_t = \mathcal{Y}_{t-1} + \eta \frac{\partial C}{\partial \mathcal{Y}} + \alpha(t)(\mathcal{Y}_{t-1} - \mathcal{Y}_{t-2}) \tag{5.7}$$

The points of the embedded space $\mathcal{Y}$ are updated at each iteration. The algorithm tries to decrease the cost function moving these points within the low-dimensional representation.

5.4 Weaknesses of the algorithm

T-SNE compares favourably to other dimensionality reduction techniques, though it has a series of downfalls. As stated by van der Maaten in his seminal paper [6], the algorithm has mainly three potential weaknesses:

1. **It is unclear how t-SNE performs on general dimensionality reduction tasks**. It has been used very successfully to visualise the structure of images in two dimensions. However, it is unclear if this specific task can be extrapolated to the more general case of reduction of d number of dimensions.
2. The relatively local nature of t-SNE makes it sensitive to the **curse of the intrinsic dimensionality** of the data. When t-SNE computes the probabilities in both the low and high dimensional spaces it uses the euclidean distance between points, assuming this way, that they are locally linear. In datasets with a high intrinsic dimensionality and an underlying manifold that is highly varying this assumption may be violated.
3. **The algorithm is not guaranteed to converge to a global optimum of its cost function**. Since the cost function is not convex a different initialisation can lead to different results. Furthermore, the algorithm has several hyperparameters whose values can change the results noticeably.

Lastly, two more minor pitfalls should be added: First, **it is slow compared to other methods**. t-SNE has a computational and memory complexity that is quadratic in the number of datapoints. Second, **it is difficult to interpret and understand** and can lead to misleading conclusions if not interpreted with caution.

Despite all these weaknesses this algorithm is useful for this project because i) it is still the state-of-the-art for visualisation purposes and ii) the loss function of the algorithm that will be used to perform Deep clustering is very similar to the one t-SNE uses.

5.5 Uniform Manifold Approximation and Projection (UMAP)

What started as a mathematical motivation to try to justify the approach used in t-SNE, ended in an algorithm called UMAP [8]. While t-SNE is built by fixing some of the problems of previous algorithms, UMAP builds a mathematically accurate solution. That is why UMAP is competitive with t-SNE for visualisation quality and arguably preserves more of the global structure with superior run time performance. UMAP's topological foundations allow the algorithm to scale to significantly larger dataset sizes than are feasible for t-SNE.

Finally, UMAP has no computational restrictions on embedding dimension, making it viable as a general purpose dimension reduction technique for machine learning. This is particularly important when the intention is to use the low dimensional representation for further machine learning tasks such as clustering or anomaly detection. More information about the advantages of UMAP over t-SNE can be found in [9].

The hyperparameters of UMAP, as stated in [10], are the following:

- **d**: The number of dimensions of the embedded space.
- **n_neighbours**: The number of neighbours to consider when approximating the local metric, which is often the Euclidean distance. Smaller values will ensure detailed manifold structure is accurately captured (at a loss of the "big picture" or global structure), while larger values look at larger neighbourhoods of each point, and thus, the local structure is not properly captured.
- **min_dist**: It provides the minimum distance apart that points are allowed to be in the low dimensional representation. Low values on min_dist will result in potentially densely packed regions, but will hopefully help to detect clusters in the data.
- **n_epochs**: The number of iterations of the optimisation algorithm.

Furthermore, the greater flexibility of this algorithm allows us to think in two next directions in dimensionality reduction:

- **Make use of labels for supervised dimensionality reduction** with the aim of interpreting better the internal structure of the data.
- **Combine numerical and categorical data**. The algorithm just needs a distance measure for the variables.

Chapter 6

Artificial neural networks

Artificial neural networks (ANNs) are a set of algorithms inspired by information processing and distributed communication nodes in biological systems. They are called neural networks because of the loosly resemblence with the neurons of the human brain. ANN have evolved greatly in the last decade and now they are the state-of-the-art in the fields of pattern recognition, computer vision or natural language procession, among others. Today, they are undoubtedly the main tool of the Deep Learning field.

Their popularity in recent years comes mostly due to three factors:

1. **Exponential increase of available data**: Neural networks start to outperform traditional algorithms when they have access to large amounts of data.
2. **Improvements of existing hardware**: Development of GPUs that allow operations to be performed in parallel, greatly accelerating processing time.
3. **Theoretical developments** such as new activation functions or initialisation methods, that overcame some of the previous obstacles.

First, I will focus on the simplest ANN, which is called **Feedforward neural network**, and in the next sections I will explain more complex structures. The feedforward neural network was the first and simplest type of artificial neural network devised and it can be seen as an efficient nonlinear function approximator based on gradient descent optimisation. It is named "feedforward" because as stated in [11] *nodes within a particular layer are connected only to nodes in the immediately "downstream" layer, so that nodes in the input layer activate only nodes in the subsequent hidden layer, which in turn activate only nodes in the next hidden layer, and so on until the nodes of the final hidden layer, which innervate the output layer.* An illustration

of the system can be seen in figure 6.2. Information flows from the input layer x through the intermediate layers until it reaches the output y.

In the next section I will start explaining the elements of a feedforward neural network although there are many other architectures. Some of the most popular are **Convolutional neural networks** (CNN), **Recurrent neural networks** (RNN) and **Autoencoders** (AE). The autoencoder is the structure specialised in unsupervised learning and thus it is the most interesting one for this project.

6.1 Elements of an Artificial Neural Network

- A **node** or **neuron** is the core feature of a neural network. They are represented as $a_j^{[l]}$, which denotes the node j in the layer l. The nodes receive information from other nodes or from the input and so they are interconnected and grouped in layers. What a node does is calculate a weighted sum of its inputs, add a bias and then transmit the "signal" to the next neurons. In 6.1 you can see the structure of a node and its elements:
  - The **Net function** or Lineal operation is represented by z or sometimes by $\sum$. As denoted in equation 6.1 this function is just the sum of the weights, the outputs of the previous layers and the bias.
  - The **Activation function** is symbolised as $g(.)$ and it is applied to the output of the Lineal operation. Typically is a very simple function, easy to compute. A summary of the most important functions will be provided in the next subsection.
- A **layer** refers to a group of nodes that operate together at a specific depth within the network. By adding more layers and more units within a layer, a deep network can represent functions of increasing complexity. They can be classified as input, hidden and output layers so it is important to distinguish the layers of the neural network and to which one we are referring to:
  - The **input layer** contains the raw data, one variable in each node. It is represented as $X \in \mathbb{R}^{n_x \times m}$ where n_x is the input size and m is the number of samples in the dataset.
  - The **hidden layer** are the layers between the input and the output and they compose the internal structure of the network. In figure 6.2 you can see two intermediate or hidden layers.
  - The **output layer** is the simplest layer and usually consists of a single output or node. However for multi-class or multi-label classification problems it can have several nodes. It is represented as $\hat{y} \in \mathbb{R}^{n_y \times m}$. When an ANN has a large number of hidden layers (usually more than one is enough), then it is called a **Deep neural network**. On the other hand an autoencoder is a specific type of ANN in which the output

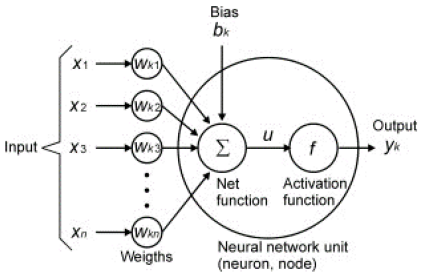


Figure 6.1: General structure of a node (Source: cs231n.stanford.edu/slides)

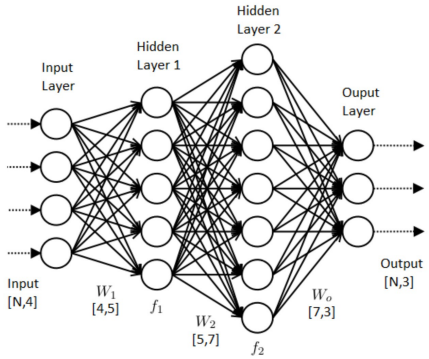


Figure 6.2: Example of ANN (Source: VIASAT)

layer is the same than the input layer, and what matters is the information provided by the intermediate layer.

- Lastly, neural networks are characterised by their large number of **parameters**. ANN have often thousands or even millions of parameters which give them a lot of flexibility. They can be of two types:
  - **Weights** (W): In the figure 6.2, the weights are represented by the lines that interconnect the nodes. The number of weights increases exponentially with the number of nodes.
  - **Biases** (b): There is one for each node so they are much less numerous than the weights.

To sum up, the following equation shows how is calculated the output of the node j of the layer l:

$$a_j^{[l]} = g^{[l]}(\underbrace{\sum_{k=1}^{n^{[l-1]}} w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]}}_{\text{(Lineal Operation)}}) = \underbrace{g^{[l]}}_{\text{(Activation function)}}(z_j^{[l]}) \tag{6.1}$$

Where the superscript denotes the layer and the subscript represents the number of the element in the layer.

This equation can be generalised easily to account for all the nodes by using matrix notation and vectorized functions. According to figure 6.2, the equations that represent the neural network would be the following:

$$
\begin{aligned}
\underset{(5x1)}{a^{[1]}} &= \underset{(5x1)}{g^{[1]}}(\underset{(5x4)}{W^{[1]}}\underset{(4x1)}{\boldsymbol{x}^{(i)}} + \underset{(5x1)}{b^{[1]}}) = g^{[1]}(z^{[1]}) \\
\underset{(7x1)}{a^{[2]}} &= \underset{(7x1)}{g^{[2]}}(\underset{(7x5)}{W^{[2]}}\underset{(5x1)}{a^{[2]}} + \underset{(7x1)}{b^{[2]}}) = g^{[2]}(z^{[2]}) \\
\underset{(3x1)}{\hat{\boldsymbol{y}}} &= \underset{(3x1)}{g^{[3]}}(\underset{(3x7)}{W^{[3]}}\underset{(7x1)}{a^{[2]}} + \underset{(3x1)}{b^{[3]}}) = g^{[3]}(z^{[3]})
\end{aligned}
$$

Where the rows represent the features or variables and the columns stand for the samples or observations. In this case the input $x^{(i)}$ is a vector because when we train one sample at a time but it can also be a set of samples, in which case it would be a matrix instead of a vector. I will explain this in the optimisation subsection.

The process represented by these three equations, in which the output $\hat{y}$ is obtained from the input after the appropriate calculations on each layer of the network, is called **forward propagation**. We can see that it is just a combination of function composition and matrix multiplication.

6.1.1 Activation functions

There are different kinds of activation functions. Moreover, each layer of the network can have a different function. Below I present a list of the most common transformations:

- **Linear function**

$$g(z) = a \cdot z \tag{6.2}$$

  Where a is a constant.

- **Sigmoid or logistic function**

$$g(z) = \frac{e^z}{1+e^z} = \frac{1}{1+e^{-z}} \tag{6.3}$$

- **Rectified Linear Unit (ReLu)**

$$g(z) = max(0, z) \tag{6.4}$$

- **Parametric ReLu**

$$g(z) = max(0, a \cdot z) \tag{6.5}$$

In the specific case in which the parameter a equals 0.01 the transformation is called **Leaky ReLu**.

Activation functions are a key part in a neural network mainly due to two factors:

- **They limit the range of values that the nodes can take**: For example, the sigmoid function transform the value of the node into "probabilities" as the range of the output values is [0,1]. Instead, the ReLu can be very useful when you do not want negative values in the network.
- Although the functions are quite simple **they introduce non-linearity to the network**. Therefore, when the network is deep enough, that is to say, it has a multitude of layers, it can approximate very complex function.

6.2 Training

The training of neural networks is similar to the process followed by other machine learning algorithms. This phase consists in optimising the parameters (W, b) of the neural network which minimise differences between output predictions and given ground truth values on a given dataset.

In this process two things have to be considered:

- A **loss function** ($\boldsymbol{L}$). Given some predictions $\hat{y}$ and the output y, the loss function $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ measures the discrepancy between them. The **cost function** (denoted as $\boldsymbol{J}(W, b)$) is the average of the loss function of the entire training set.
- An **optimisation** method to minimise the cost function efficiently, and hopefully reach or end closer to the global minimum. Given the parameters W and b; and the corresponding cost function $J(W, b)$, we can find at least a local minimum using an optimisation algorithm.

6.2.1 Loss functions

The choice of loss function is tightly related to the choice of output unit and the type of problem at hand. Next, I present the most common functions:

Discrete loss functions

- **Categorical cross entropy (CE)**

$$L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{i=1}^{n} y_i \cdot log(\hat{y}_i) \tag{6.6}$$

When there is only one output ($n = 1$), then it is called BCE:

- **Binary cross entropy (BCE)**

$$L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -(y \cdot log(\hat{y}) + (1 - y) \cdot log(1 - \hat{y})) \tag{6.7}$$

Is used for example to approximate a logistic regression.

Continuous loss functions

- **Root Mean Square Error (RMSE)**

$$L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{6.8}$$

- **Mean Absolute Error (MAE)**

$$L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \sum_{i=1}^{n}|\hat{y}_i - y_i| \tag{6.9}$$

6.2.2 Optimisation

Gradient descent is one of the most popular algorithms to perform optimisation and by far the most common way to optimise neural networks. At the same time, there are several implementations to optimise gradient descent that aim to overcome some of its limitations. Some of the most popular are **RMSprop**, **Adaptative Moment Estimation (Adam)** and **AdaDelta**. To see the formulae and advantages of these methods you can consult [12]. In this subsection I will only summarise the functioning of Gradient Descent ans its variations: Batch and Mini-Batch Gradient Descent and Stochastic Gradient Descent.

As stated by Sebastian Ruder in [12], *gradient descent is a way to minimise an objective function $J(\theta)$ (e.g the cost function) parametrised by a model's parameters (W, b) by updating the parameters in the opposite direction of the gradient of the objective function $\bigtriangledown_\theta J(\theta)$ with respect to the parameters.*

Batch gradient descent algorithm uses the entire training set to update the parameters:

$$\theta = \theta - \eta \cdot \bigtriangledown_\theta J(\theta) \tag{6.10}$$

The parameter η called **learning rate** multiplies the gradient and determines the size of the steps we take to reach a (local) minimum.

Stochastic gradient descent (SGD) in contrast performs a parameter update for each training sample $(x^{(i)}, y^{(i)})$:

$$\theta = \theta - \eta \cdot \bigtriangledown_\theta J(\theta; x^{(i)}; y^{(i)}) \tag{6.11}$$

Mini-Batch gradient descent performs an update for every mini-batch of n training samples:

$$\theta = \theta - \eta \cdot \bigtriangledown_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \tag{6.12}$$

Where common mini-batch sizes range usually from 32 to 256 training samples.

Depending on the amount of data, we make a trade-off between the accuracy of the parameter update and the time it takes to perform an update. SGD updates the parameters very frequently but with low accuracy and high variance while Batch gradient descent does the opposite.

A part from the number of samples we use in each training pass, another important hyperparameter to take into account is the number of complete passes (known as **epochs**) through the training dataset.

Unfortunately, nonlinearity of neural networks causes most of the used cost functions to become nonconvex. This means that ANN are usually trained by using iterative, gradient-based optimisers that just drive the cost function to a very low value since it is not guaranteed to find a global minimum. Moreover, the algorithm is sensitive to the values of the initial parameters.

Backpropagation

In practice, the process of updating the parameters is carried out by an algorithm backpropagation, which is a short form for "backward propagation of errors"; meaning, the algorithm computes the error vectors backward, starting from the final layer. Although this algorithm is very important as it provides us with a way of computing the gradient of the cost function efficiently and of updating the parameters of the network, all the deep learning softwares have automated this process and the users do not have to worry about the specific formulae.

6.2.3 Complete training cycle

Here I would like to put all the pieces together and summarise the training process of a neural network. The training recipe is the following:

1. Initialise the parameters assigning small random values to them.
2. For each set of training samples:
   (a) Do the forward propagation to calculate the outputs $\hat{y}$.
   (b) Evaluate the loss function for the training examples introduced in the network using the calculated output, that is, compare the output with the expected values.
   (c) Backpropagate the error to each and every one of the parameters that make up the network.
   (d) Update the parameters using an optimisation algorithm in a way that the total loss is reduced and a better model is obtained.
   (e) Stop when:
      - Early stopping: The decrease of the error is slower than a chosen threshold.
      - Or the algorithm has reached the maximum number of epochs set.

6.3 Convolutional Neural Networks

A CNN is a network that use **convolutions** in place of general matrix multiplication (i.e. a **fully connected layer** [1]), at least in one of its layers. It is applied mainly to image data and some of their applications are image classification, object detection or facial recognition, to name a few. CNNs are one example of how to effectively incorporate domain knowledge into the network architecture.

CNNs are used when the data share some of these properties [13, p.6]:

- They are stored as multi-dimensional arrays.
- They feature one or more axes for which ordering matters (e.g., width and height axes for an image).
- One axis, called the channel axis, is used to access different views of the data (e.g., the red, green and blue channels of a colour image).

Natural images have many statistical properties that are not exploited when an affine transformation is applied; in fact, all the axes are treated in the same way and the topological information is not taken into account. Therefore, it is important to use some layers that take advantage of these properties. The most common is the convolution but there are others:

[1] Fully connected layers are the traditional layers used in feedforward neural network in which all the nodes are interconnected between them.

- **Convolutional layers**: They take a group of input nodes and compute a single output, reducing thus the dimension of the network. A convolutional layer's output shape is affected by the shape of its input as well as the choice of **filter size**, the **padding** and **strides** [2]. This contrasts with fully-connected layers, whose output size is independent of the input size.

- **Pooling layers**: These operations reduce the size of feature maps or input matrix (down sampling) by using some function to summarise subregions. The most common kind of pooling is max pooling, which adds nonlinearity to the network and consists in splitting the input in (usually non-overlapping) patches and outputting the maximum value of each patch. In figure 6.3 you can see how this layer works: The max pooling, with size (of the filter) 2x2, applies max to 4 numbers (little 2x2 squares) of the input matrix. Two strides is the number of pixels shifts over this matrix.

- Other kind of layers are the **UpSampling** layer, the **Transpose convolution** or the **Separable convolution**.

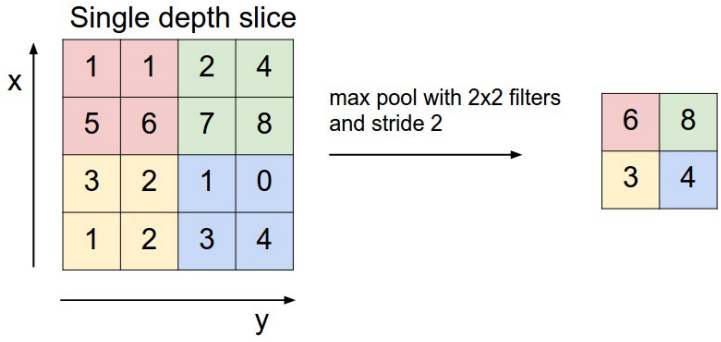


Figure 6.3: Max pooling layer (Source: cs231n.github.io)

These types of layers enable CNNs to dramatically lower the number of unique model parameters and to significantly increase network sizes without requiring a corresponding increase in training data. To see more in detail the layers used in CNNs look up [11, p.327], [13] or [14].

One important example of CNN is **AlexNet** [15], an architecture that had a large impact on the field of computer vision. In figure 6.4 you can see that the network combines alternatively convolutional and pooling layers and the number of parameters in each layer is decreasing. Moreover, the last three layers are fully connected and the last one uses a softmax activation function, which is useful for classification problems which several categories. However, although this architecture was a major breakthrough in computer vision back in 2012, nowadays CNNs have often hundreds of layers and are much more complex.

[2] I will explain these terms in the applied section as needed.

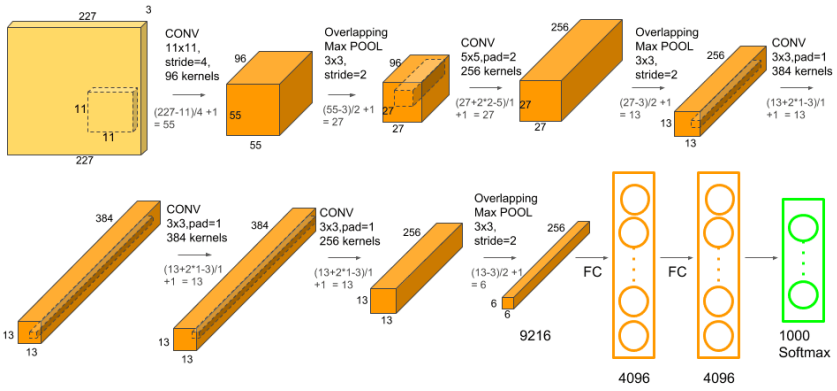


Figure 6.4: AlexNet CNN (Source: Neurohive.io)

6.4 Building a neural network

When building a neural network there are a lot of decisions to make:

- **Related to the blocks of the network**:
  - Number of hidden layers and their nodes
  - Choice of activation functions
  - Hyperparameters of some types of layers such as convolutions
- **Related to the training phase**:
  - Loss function
  - Choice of the optimisation algorithm: Standard gradient descent, Adam, RMSprop etc.
  - Hyperparameters of the optimisation algorithm:
    * The batch size controls the number of training samples to work through before the model's internal parameters are updated.
    * The number of epochs controls the number of complete passes through the training dataset.
    * Learning rate
    * Other hyperparameters specifics to the optimisation algorithm.

- **Regularisation methods**: You need an algorithm that will perform well not just on the training data, but also on new inputs. Regularisation methods are explicitly design to reduce the test error, possibly at the expense of increased training error.
  - L^2 parameter norm penalty known as Weight decay or as ridge regression.
  - Dropout
  - Batch normalisation
  - **Early stopping**: In which moment the training will stop. If the algorithm doesn't reduce the loss function significantly then it is not necessary to complete all the epochs.

6.5 Autoencoders

In recent years, motivated by the success of deep neural networks in supervised learning, unsupervised deep learning approaches are now also widely used for dimensionality reduction. For unsupervised tasks the most common approach is the use of **Autoencoders (AE)**. An autoencoder, in short, is a neural network that is trained to attempt to copy its input to its output. It may be thought of as being a special case of feedforward network and may be trained with all the same techniques.

The structure of an autoencoder can be divided into two parts:

$$\text{Encoder} : \boldsymbol{z} = g(\boldsymbol{x}) \tag{6.13}$$

$$\text{Decoder} : \boldsymbol{x'} = f(\boldsymbol{z}) \tag{6.14}$$

Where $\boldsymbol{x}$ is the input vector; $\boldsymbol{z}$ is the **embedding layer or latent space** and $\boldsymbol{x'}$ is the reconstruction done by the autoencoder.

If an autoencoder succeeds in simply learning to set $f(g(\boldsymbol{x}))$ everywhere, then it is not especially useful. Instead, AE are designed to be unable to learn to copy perfectly. Usually they are restricted in ways that allow them to copy only approximately, and to copy only input that resembles the training data. Because the model is force to prioritise which aspects of the input should be copied, it often learns useful properties of the data [11, p.499].

In order to achieve this, many different structures have been developed; often with different applications in mind. Some of the most relevant are the **stacked autoencoder** (Vincent et al., 2010) or the **sparse autoencoder** (Ng, 2011) that take insights from traditional techniques such as PCA and make use of deep neural networks to learn non linear mappings from the data domain to low-dimensional latent spaces. Another different kind is the **variational autoencoder**, which at the same time learns useful representations and allows the synthesis of novel data samples that are consistent with the training data. However, not all the autoencoders work only in a pure unsupervised mode. **Structuring AutoEncoders** [16] enhance traditional autoencoders with a weak supervision, using only a very small amount of additionally labelled data. Thus, the data can be separated into several classes which are not obvious in the data itself.

The most common structure though, is called **undercomplete autoencoder** and it is the one that I will refer to from now on since it is the one I will apply later on. An under complete autoencoder is an autoencoder in which the dimension of z (also called bottleneck or embedding layer) is lower than the dimension of the input x. The encoder maps the data points through the hidden layers to a low dimensional latent space from where the decoder reconstruct the input. Learning an undercomplete representation forces the network to capture the most important features of the data (which is the objective of dimensionality reduction). Therefore, when using an undercomplete AE we are not interested in the output but in the properties of z. You can see a representation of this in figure 6.5: Usually the encoder and decoder are symmetric (they

have the same number and type of layers) and the input and output are very similar, though not identical.

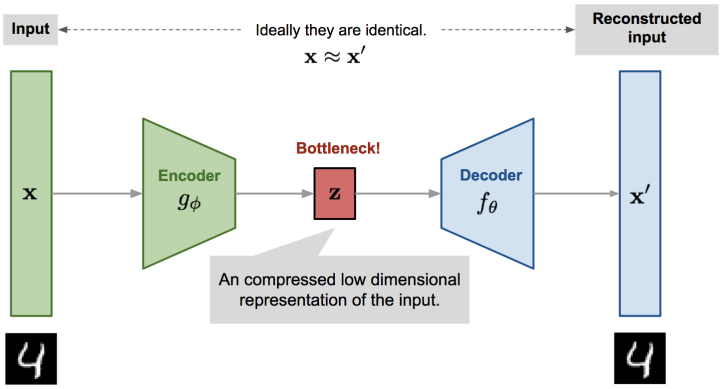


Figure 6.5: Structure of an autoencoder (Source: [17])

The learning process is described as minimising a loss function

$$L(\boldsymbol{x}, f(g(\boldsymbol{x})))$$

The most common loss functions L used in this networks are the cross entropy and the root mean square error. These functions penalise $f(g(\boldsymbol{x}))$ for being dissimilar from $\boldsymbol{x}$. Therefore, we want to obtain a reconstruction x' as close as possible to the original input x (ideally they should be identical).

Convolutional autoencoders (CAE) are a type of Convolutional Neural Networks whose objective is to reduce the dimensionality of images. The only difference with undercomplete AE is that they use a different kind of layers (i.e. convolutional or pooling layers, those typical from CNN), to take advantage of the 2D image structure.

6.5.1 Advantages of Autoencoders for DR

By now, several dimension reduction techniques along with their problems have been seen. Luckily some of these weaknesses can be corrected using autoencoders. Next, I provide a list of some of the potential benefits of using this algorithms:

a) Autoencoders can adapt their structure to different types of input. For example, a CAE take advantage of the implicit structure of image data to learn better representations and reduce the necessary number of parameters, improving its performance.

b) Autoencoders reduce the input using the Encoder's part of the network but are also capable of reconstruct the input from the latent space using the Decoder. This has proven useful for example with Variational autoencoders (VAE) which can generate new data similar to the input received and thus, they can be generative models.

c) Autoencoders can identify highly-varying manifolds potentially better than a local method such as t-SNE as they contain more than one layer of nonlinearity. Certainly, they can learn a more powerful nonlinear generalisation of PCA. They can recover the intrinsic geometric structure of a broad class of data structures.

d) Moreover, autoencoders scale well to both the number of variables and samples. Therefore, performing t-SNE on a data representation produced by, for example, an autoencoder is likely to improve the quality of the constructed visualisations [18].

e) And last of all, the flexibility of neural networks allow them to perform jointly dimensionality reduction and clustering in what it is called Deep Clustering.

However, like all algorithms, CAEs also have relevant problems. Among them, the **lack of theoretical foundations** stands out. For CAE and autoencoders in general, is very difficult to know how a parameter or the architecture of the network will influence the low dimensional space. ANNs are often considered a **black box** because the mapping function is too difficult to analyse. Using an autoencoder you just cannot know which properties the embedding space has, or how is the shape of the potential clusters. Moreover, since deep learning it is a relatively new field, the theory is not well structured.

Part II

Clustering

Chapter 7

Overview

Clustering deals with the data structure partition in an unknown area. It seeks to separate the data into groups, according to certain similarity measures, in a way that *similar* data points are placed in the same cluster, and *dissimilar* data points are placed in different clusters.

As an unsupervised learning paradigm, clustering does not require the original data to be labelled in advance. Instead, it aims at automatically exploring the inherent structure of the data to help people acquire an in-depth appreciation of the key properties of the data.

7.1 Classification

In practice, clustering is a notoriously hard task, whose outcome is affected by a number of factors including data acquisition and representation, use of preprocessing such as dimensionality reduction, the choice of clustering criterion and optimization algorithm or initialization, among others. Due to this complexity each clustering algorithm has its own strengths and weaknesses. There is no clustering algorithm to rule them all: the user should choose carefully the algorithm given the data and the criterion to optimize.

Therefore, a large number of clustering methods have been proposed throughout the years. These methods are characterized by different ways of measuring homogeneity, diverse procedures for searching the optimum partition, and various problem dependent restrictions. The most conventional categories of clustering algorithms are **partitioning**, **hierarchical** and **density based** but traditional clustering algorithms can be classified into up to 9 categories [19]; and there are many more.

On the other hand, the algorithms can also be classified according to their output: Clustering results can be either "hard" or "soft". **Hard clustering** is about grouping the data items such that each item is only assigned to one cluster. On the contrary, in **soft clustering** the items are assigned probabilities which are essentially expressing the strength of the belonging of the item into the cluster.

7.2 Evaluation

A common practice for investigating clustering techniques is by empirical studies where a set of benchmark datasets are used to quantitatively evaluate the performance of specific algorithms. This evaluation can be done in two different ways:

- **Internal or unsupervised validation**: It is based on the information provided by the data used as input to the clustering algorithm, that is to say, the evaluation of the clustering is compared only with the result itself.
- **External or supervised validation**: It uses external information. In general, this information is provided in the form of external class labels for the training examples. Is largely used for synthetic data and for tuning clustering algorithms.

To evaluate the quality of the clusters correctly, there are several factors that must be taken into account:

a) For high dimensional non-trivial datasets, it is rarely the case that *true* cluster labels are readily available. To overcome this issue, there are two options: use only internal evaluations metrics or, if we want to employ external evaluation metrics, use classification datasets. On on hand, internal evaluation metrics are not as powerful as the external ones since they only use the information provided by the data. On the other hand, when using classification datasets the key problem is that the labels are defined for classification purposes, not clustering, and mixing the two scenarios can produce unpredictable consequences. In those datasets the labeling is only an indicator of the class property instead of the intrinsic distribution of the data.

b) There is no association provided by the clustering algorithm between the external class labels (also known as the ground-truth) and the predicted cluster labels. Therefore, the metrics used are more complex than the ones used in a supervised classification problem.

c) There are needed several evaluation metrics due to the fact that there are many ways to understand what clusters are.

More information about these issues can be found in [20, Critical Note on the Evaluation of Clustering Algorithms].

7.2.1 Metrics

Clustering accuracy and normalized mutual information will be the main metrics that I will use to evaluate the clustering results. Both of them are external measures. However, I will also employ some internal measures such as the Silhouette score. An overview of some of them can be found in [21].

Clustering accuracy (ACC)

Accuracy for classification is the sum of the diagonal elements of the confusion matrix divided by the number of samples. However, since in clustering we do not know the true labels of the samples, we have to define this measure by finding the best match between the class labels y and the cluster labels $\hat{y}$ (hence the use of permutations):

$$\mathrm{ACC}(y, \hat{y}) = \max_{\mathrm{perm} \in \mathrm{P}} \frac{1}{n} \sum_{i=1}^{n-1} \mathbb{1}\{\mathrm{perm}(\hat{y}_i) = y_i\}$$

Where P is the set of partitions (e.g.clusters) of the data. It is important to note that there are K! permutations in P which is quite high and the computation of accuracy is therefore prohibitive. There are several algorithms that are used to speed up the calculations.

Mutual information (MI)

The Mutual Information is a symmetric measure that quantifies the mutual dependence between two random variables, or the information that two random variables share. In clustering, it can be used to determine the similarity of two clustering patterns y and $\hat{y}$ of a dataset X.

$$MI(y, \hat{y}) = \sum_{i=1}^{k_y} \sum_{j=1}^{k_{\hat{y}}} p_{ij} \cdot log(\frac{p_{ij}}{p_i p_j})$$

Where p_{ij} denotes the probability that a point belongs to both y_i and $\hat{y}_j$; p_i is the probability

that the random object in X falls into y_i; and p_j is the probability that the random object in X falls into $\hat{y}_j$.

Normalized mutual information (NMI)

One normalized version of MI is the Normalized mutual Information.

$$NMI(y, \hat{y}) = \frac{MI(y, \hat{y})}{\frac{1}{2}[H(y) + H(\hat{y})]}$$

Where $MI(y, \hat{y})$ is the mutual information metric and $H()$ is the entropy function.

Another modification of Mutual information is the **Adjusted mutual information** (AMI). It is a measure that corrects the effect of chance in the creation of the clusters. The range of both NMI and AMI is [0,1]: Values close to zero indicate two label assignments that are largely independent (or a random assignment), while values close to one indicate significant dependence.

Rand index (RI)

$$RI(y, \hat{y}) = \frac{a + b}{\binom{n}{2}}$$

Where:

a: the number of pairs of elements in X that are in the same subset in y and in the same subset in $\hat{y}$

b: the number of pairs of elements in X that are in different subsets in y and in different subsets in $\hat{y}$

The denominator is the number of unordered pairs in a set of n elements.

There is an **adjusted RI for chance** (called ARI). The ARI is bounded between 0 and 1. When its value is 0, then the RI equals its expected value (under the generalized hypergeometric distribution assumption for randomness).

7.3 Why dimensionality reduction can be useful for clustering

In this section I provide an overview of some of the reasons that make dimensionality reduction useful for the clustering task. However, when applying both tasks, dimensionality reduction and clustering, you also have to take into account the assumptions and limitations of each technique. Therefore, I will discuss as well some problems that arise when applying dimensionality reduction for clustering.

a) Many applications require the clustering of large amounts of high-dimensional data. Most clustering algorithms, however, do not work effectively and efficiently in high-dimensional space, which is due to the so-called *curse of dimensionality*[1]. In addition, the high-dimensional data often contains a significant amount of noise which causes additional effectiveness problems. Usually, data noise is either due to imperfection in the technologies that collected the data or the nature of the source of this data itself (one can think of images for example). Dimensionality reduction mitigates the *curse of dimensionality* and other undesired properties of high-dimensional spaces. Additionally, in many cases it also reduces computational cost.

b) Dimensionality reduction allows to visualise the intrinsic structure of the data intuitively using two and three-dimensional plots. It can be useful for example to decide how many clusters can be made or even to directly cluster the data. But it has several drawbacks:

   - There may be, for example, relationships among clusters that can't be represented in 2-dimensions.
   - There are many assumptions that we make about characteristics of lower dimensional spaces based on our experience in three dimensional Euclidean space. There is a conceptual barrier that makes it difficult to have proper intuition of the properties of high dimensional space and its consequences in high dimensional data behavior.
   - The interpretation can be misleading. For example, in most embedding spaces distances between clusters are distorted and thus, it makes no sense interpreting cluster placement. Some users might try to read into the cluster placement and not only the cluster membership getting erroneous insights.
   - There are other techniques that allow doing similar things. For example, it is possible to determine the optimal number of clusters by inspecting a *dendrogram* or by optimizing a criterion, such as the within cluster sums of squares or the average silhouette.

   So what do you believe more: the output of your favourite clustering algorithm together with your favourite heuristic for identifying the number of clusters, or what you see on the plot of some dimensionality reduction method?

[1] For a refresher of the curse of dimensionality problem see the Dimensionality reduction section (page 10). In short, when the dimension of the input space is very high, clustering often becomes ineffective due to unreliable similarity metrics (e.g. euclidean distance).

c) Dimensionality reduction not only reduces noisy dimensions but it can also help to distinguish better the clusters by modifying the structure of the data. They can uncover the latent structure in datasets. The catch here is that the distances -and even the densities in some cases-, of the data are distorted in many dimensionality reduction algorithms. Therefore, it is important to use a clustering algorithm robust to these distortions. Some non-distance based clustering techniques like FMM (Finite Mixture Models) or **HDBSCAN** (Density-based Models) can be useful.

d) There are data driven algorithms that solve for the feature space and cluster memberships jointly. Most of these algorithms fall into the field of deep clustering which uses deep neural networks.

In addition to everything that has been said, I think it is appropriate to make three more general statements:

- First, given a sufficiently powerful cluster algorithm, one would never need to apply any dimensionality reduction technique.
- Second, a key issue in all the applications of dimensionality reduction is the transformation of the data from the original space into a new space with lower dimension, which cannot be linked easily to the variables in the original space. Therefore, further analysis of the new space is problematic since there is often no physical meaning for the transformed variables.
- Third, the best algorithm for one specific application of dimensionality reduction doesn't necessarily has to be the best candidate for another application (such as clustering).

7.3.1 Clustering strategies using DR

Because of the aforementioned benefits that DR can report to clustering, different approaches have been put into practice. As a summary of what has been said, I would like to mention the following classification:

Traditional clustering: It is the use of a clustering algorithm alone, without any additional help. This creates several problems: The algorithm may suffer the curse of dimensionality, the computations are in many occasions much more slow, you have no clues about the structure of the data and in brief, the data has noise that you do not want.

Two-stages clustering: The first applications of DR to clustering used this strategy.

This approach first applies feature transformation [2] in order to perform clustering afterwards. These algorithms directly take advantage of existing unsupervised learning frameworks and techniques. For example, you can use an autoencoder to learn low dimensional features of the original dataset, and then run k-means algorithm to get clustering results. Furthermore, instead of autoencoders you can think of more traditional machine learning algorithms such as t-SNE, UMAP and so on; there is a wide range of possibilities to explore.

Deep clustering: This category uses deep neural networks to perform jointly, or at the same time, dimensionality reduction and clustering. Thus, this new field combines representation learning and clustering into a unified framework which can directly cluster the original data end-to-end. In this project I focus on analysing a clustering algorithm called Deep Convolutional Embedding Clustering (DCEC) that is based mainly on the dimensionality reduction induced by an autoencoder. This is one of the many algorithms that have appeared recently inside this category and that shows promising performance.

[2] By feature transformation I refer to dimensionality reduction but the term also includes other techniques such as feature engineering. Feature engineering is the process of extracting manually features from the data using domain knowledge. Although it is a labour intensive task and it requires specialists on the topic, it takes advantage of prior learning of the data.

Chapter 8

k-means

k-means is probably the most popular clustering algorithm. It is quite simple to understand yet powerful. The algorithm aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest centroid, also called **means** or **cluster centers**. Two assumptions are the basis of the k-means model: The cluster center is the arithmetic mean of all the points belonging to the cluster and each point is closer to its own cluster center than to other cluster centers.

Thus, k-means minimises the **inertia**, or **within-cluster sum-of-squares criterion**:

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\boldsymbol{x} \in S_i} \|\boldsymbol{x} - \boldsymbol{\mu_i}\|_2^2 \qquad ; \quad i = 1...k$$

Where k is the total number of clusters, S identifyies the cluster to which $\boldsymbol{x}$ belongs and μ represent the centroids.

Inertia sums the euclidean distances of all the points within a cluster from the centroid of that cluster. The lesser the inertia value, the better our clusters are. Trying to minimise this function is very complex due to all the possible ways in which the n observations can be allocated in the k groups. Fortunately there are several heuristics that converge quickly to local best solutions.

8.1 How does it work

The typical approach to minimise the inertia involves an intuitive iterative approach known as **Lloyd's algorithm**. The procedure is the following:

1. Choose some cluster centers.
2. Repeat until *convergence*:
   (a) 1st-Step: Assign each point to the nearest centroid.
   (b) 2nd-Step: Create new centroids by taking the mean value of all of the samples assigned to each cluster center.

Given enough time, k-means will always converge, however this may be to a local minimum. This is highly dependent on the initialization of the centroids. As a result, the computation is often done several times, with different initializations and is picked the one that has a lower inertia value.

8.2 Advantages and limitations

Due to the way the algorithm is built, it is recommended to use it only **when clusters are spherical, well separated, with similar volumes and non-overlapping**. The fact that the algorithm does not work well in a wide range of cluster structures, makes it normally outperformed by other algorithms. Moreover, the number of clusters must be selected beforehand and the globally optimal result may not be achieved.

Chapter 9

Deep Clustering

In recent years, motivated by the success of deep neural networks (DNNs) in supervised learning, unsupervised deep learning approaches are now widely used for clustering. These new approaches can be classified in a new subfield called **Deep clustering**.

Most deep clustering algorithms try to **jointly optimise feature learning and clustering** by explicitly defining a clustering loss in the neural network, which is often an autoencoder as show in figure 9.1. Given an unlabelled dataset, it will iteratively learn the neural network parameters unsupervisedly and cluster the samples.

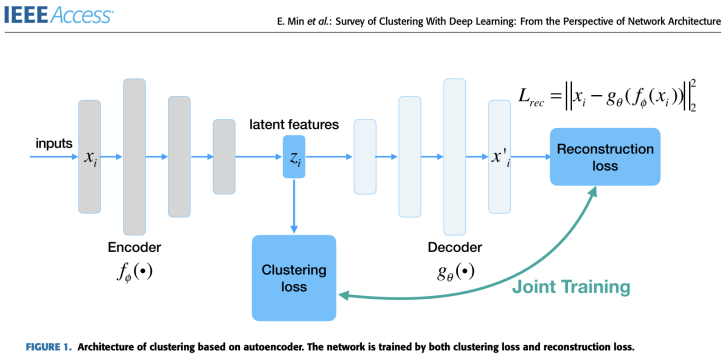


Figure 9.1: Representation of joint learning in an autoencoder

The intuition behind the joint unsupervised learning is that better feature representation will facilitate clustering, while better clustering results will help representation learning. Recent research has shown that optimising the two tasks jointly can substantially improve the performance of both. Furthermore, you can keep the advantages of jointly optimising the two tasks, while exploiting the deep neural network's ability to approximate any nonlinear function. Additionally, a DNN is also useful to adjust the clustering process easily to different types of inputs

such as images (by using a CNN) or text (by using a RNN). On top of all of this, learned representations generalise easily across datasets. Thus, it is possible for example to train a deep clustering algorithm on one dataset, and then cluster images from another (but related) dataset using the image features extracted via the neural network.

Despite all the advantages, deep clustering leaves a lot of problems unsolved. For example, what types of neural networks are proper? How to provide guidance information i.e. to define clustering oriented loss function? Which properties of data should be preserved during the transformation? [22] There is a lot left to explore and that is why the research has continued evolving at a fast pace in the last years.

Algorithms that jointly accomplish feature transformation and clustering gained popularity around 2016 when several papers where published. [Yang et. al., 2016 23] proposes a recurrent framework for **Joint Unsupervised LEarning (JULE)** of deep representations and image clusters, which integrates two processes into a single model with a unified weighted triplet loss and optimises it end-to-end. Specifically, they use agglomerative clustering and a CNN, and treat the clusters assignment as a soft labels. The **Deep Embedded Clustering (DEC)** [Xie et al., 2016 24] learns a mapping from the original space to a low-dimensional latent space using an undercomplete autoencoder, which can obtain feature representations and cluster assignments end-to-end. One year later, the DEC algorithm is improved by [Guo et al., 2016 22] who put the focus on assuring the local structure preservation of the embedded space. More recently, Facebook's researchers have focus on scalable Deep clustering algorithms applied to non-curated raw datasets [Caron et al., 2019 25] using similar approaches. For a more detailed survey of deep clustering algorithms I refer to [26]. In the future, we can expect an increasingly similar performance between unsupervised and supervised methods for classification.

In the next section and in the applied part, I will focus on a specific deep clustering algorithm called **Deep Convolutional Embedded Clustering (DCEC)** that uses a convolutional autoencoder (CAE) to cluster images represented by pixel intensities. This algorithm is presented in [Guo et. al., 2017 27] and is based heavily on the DEC algorithm already mentioned. It adds two things: The CAE and the preservation of the data structure by keeping the autoencoder's loss.

9.1 Deep convolutional embedding clustering algorithm (DCEC)

Although CAEs have a lot of hyperparameters, the output of the networks is less sensitive to them than compared to other methods such as t-SNE. This robustness is an important property of the algorithm since, when applied to real data, supervision is not available for hyperparameter cross validation [Xie et al., 2016 24].

The DCEC algorithm is compound mainly by three components. The main element is a deep neural network, specifically, a **convolutional autoencoder** (CAE). This autoencoder is used to generate a latent space adequate to achieve good performance in the formation of clusters. Secondly, the DCEC uses **k-Means** to assign the clustering labels at each iteration. Lastly, the clustering loss function is heavily based on that of the **t-SNE** algorithm.

It has several stages:

1. **Pretraining**: The autoencoder is optimised to reconstruct the input as well as possible. With this aim, a traditional loss function is used, for example MSE or Cross-entropy.
2. **Training considering clustering and reconstruction**: In the second stage, the algorithm starts with the parameters of the pretraining, but now, the optimisation is done differently. The loss function consist of a component to ensure feature learning and another that takes into account the formation of proper clusters.
3. **Clustering**: Once the algorithm converges, k-means is applied to the values of the embedding layer in order to form the definitive clusters.

Each step of the process will be explained in detail below.

9.1.1 Pretraining with the CAE

We start with a convolutional autoencoder, already explained in 6.5 (page 35), which is form by an encoder and a decoder:

$$Encoder: \ z = g(x) \tag{9.1}$$
$$Decoder: \ x' = f(z) \tag{9.2}$$

Where x is the input vector; z is the embedding layer or latent space and x' is the reconstruction done by the autoencoder. The embedding layer will be crucial later to form the centroids (using k-means).
In order to have a good autoencoder -with a low loss value-, we want x as close as possible to the autoencoder's reconstruction x'. In order to achieve this, we will minimise a loss function subject to some parameters (weights and biases). The most widely used loss function is the mean square error though alternatively the binary cross entropy can also be used if the inputs are properly normalised. Here is the cost function $J(\theta)$ of both:

$$MSE = \frac{1}{n}\sum_{i}^{n} \|f(g(x_i)) - x_i\|_2^2$$

$$Binary\ cross\ entropy = -\frac{1}{n}\sum_{i}^{n} x_i \cdot log(x_i') + (1 - x_i) \cdot log(1 - x_i')$$

Where $i = 1...n$ represents each sample and n is the total number of observations.

However the structure of this autoencoder has some distinct characteristics:

- As mentioned before, it's a convolutional autoencoder. Therefore, it uses convolutions and downsampling layers in the encoder and transposed convolutions and upsampling in the decoder.
- **The number of units in the embedding layer has to be the same than the number of clusters you want**. If these quantities differ, it's not possible to compute the clustering loss. As the number of cluster is much lower than the number of variables of the data, we are dealing with an undercomplete autoencoder.

The optimisation of the weights and biases is done as usual.

9.1.2 Training stage

Here is were the actual DCEC algorithm starts: once the weights of the convolutional autoencoder converge, the algorithm starts the training of the complete model with a slightly different loss function:

$$L = L_r + \lambda L_c \tag{9.3}$$

Where L_r is the loss of the autoencoder's reconstruction (MSE or Binary CE as usual) and L_c is the loss associated with the formation of clusters (will be defined below).

We need both terms because autoencoders can preserve local structure of the data generating distribution and the clustering component can improve the assignation of clusters, which is actually the final objective.

9.1.3 Parameter initialisation

In order to start the training we need:

1. **Parameters of the autoencoder**, that is, weights and biases. We will use the resulting parameters of the CAE pretaining optimisation.
2. **Initial cluster centroids** μ_j where $j = 1...K$ and K is the total number of classes. They will be calculated initially using k-means over the embedding layer at the end of the pretaining phase. The k-means algorithm will be run several times (e.g. 10 or 20) and the case with the best performance -with the lowest within-cluster sum-of-squares-, will be selected. The centroids will be needed to compute L_c.

To optimise L_c the algorithm alternates between two steps: In the first step, we compute a soft assignment between the embedded points and the cluster centroids. In the second step, we update the autoencoder's parameters and refine the cluster centroids by learning from current high confidence assignments using an auxiliary target distribution. The formulas will be explained next.

9.1.4 Soft assignment

The Student's t-distribution is used as a kernel to measure the similarity between embedded point z_i and centroid μ_j.

$$q_{ij} = \frac{(1 + \|z_i - \mu_j\|^2/\alpha)^{-\frac{\alpha+1}{2}}}{\sum_{j'}(1 + \|z_i - \mu_{j'}\|^2/\alpha)^{-\frac{\alpha+1}{2}}} \tag{9.4}$$

1. z_i is the embedding layer of the sample i.
2. α are the degrees of freedom of the Student's t distribution. We will set $\alpha = 1$ because it's complicated to fine-tune the hyperparameter since we are using unsupervised-learning.
3. Most important: q_{ij} can be interpreted as the probability of assigning sample i to cluster j. It's a soft assignment, similar to that of the Gaussian Mixture Model. In contrast, k-means do a hard assigment, that is, each observation can only be in one cluster (no overlap is allowed).

The **clustering assignment** (put a label on each observation), is done using $max_j(q_{ij})$. This means, that for each observation you choose the cluster which has a higher soft assignment.

9.1.5 KL divergence minimisation

Now, the loss component associated to the clustering is defined. It uses the KL divergence together with the soft assignment and an auxiliary or target distribution p_i.

$$L_c = D_{KL}(p(x)||q(x)) = \sum_i \sum_j p_{ij} \cdot log(\frac{p_{ij}}{q_{ij}}) \tag{9.5}$$

where i is associated to the sample while j is connected to the clusters. And the auxiliary element is defined as,

$$p_{ij} = \frac{\frac{q_{ij}^2}{\sum_i q_{ij}}}{\sum_j (\frac{q_{ij}^2}{\sum_i q_{ij}})} \tag{9.6}$$

The target distribution (P), which serves as a ground truth, ideally needs three properties [Xie et al., 2016 24]: i) Strengthen predictions, ii) put more emphasis on data points asigned with high confidence, iii) normalise loss contribution of each centroid to prevent large clusters from distorting the hidden feature space.

This clustering strategy can be seen as a form of **self-learning**: The algorithm learns from "high confidence predictions", which in turn helps to improve low confidence ones. However, the downside is that the DCEC can perform very badly if the initial assignment of the centroids (after the pretrainig) is not good.

The KL objective function and the soft assignment are similar to those made by t-SNE, which were explained in 5.2 (page 20).

9.1.6 Optimisation

As in the optimisation of the CAE during the pretraining, the algorithm SGD or some of its variations will be used (e.g. Adam or RMSProp) to update jointly the parameters of the autoencoder and the centroids μ_{ij}.
As stated in page 29, the formula for gradient descent is the following:

$$\theta_t = \theta_{t-1} - \theta \nabla_\theta J(\theta) \tag{9.7}$$

Where θ are the parameters of the autoencoder (W, b) and the centroids μ_{ij}.

We stop our procedure when less than a determined % *(tol)* of points change cluster assignment between two consecutive iterations.

Part III

Applied analysis

Chapter 10

Methodology

10.1 Description of the data

10.1.1 Digital images

The applied section of this project focuses on a specific data type: **digital images**. Digital images are structures form by matrices of numbers, typically a two-dimensional (2D) grid. Each number represents a **pixel** which is the smallest controllable element of a picture represented on the screen. The intensity of a pixel is variable but its value range from 0 to 255. In picture 10.1 you can see how this works.

An **RGB image**, sometimes referred to as a truecolour image, is stored in most software as an $Height$ x $Width$ x $Depth$ data array. Height and width represent the number of pixels while the depth represents the number of channels of the picture. Most of the time an image has a Depth = 3 that defines red, green, and blue colour components for each individual pixel. Image 10.3 has shape (32x32x3) so it has a total of 3072 numbers. The formula is:

$$\text{Size} = \text{height} \cdot \text{width} \cdot \text{depth} \tag{10.1}$$

I decided to work with images for several reasons:

> Images are form by thousands of pixels, each one representing a feature. They are a very high dimensional data type (with thousands of dimensions) which makes dimensionality reduction techniques more important. Moreover, usually this kind of data has a lot of

noise, which is either due to imperfection in the technologies that collected the data (e.g. cameras) or the nature of the source of the data itself.

On the other hand, in the last decades there has been an explosion in visual content. However, much of this content is unlabelled. Manually labelling data is a time-consuming, laborious, and often expensive process. in this sense, clustering is a promising task to make the most of unlabelled datasets since it can classify images unsupervisedly.

There are other more pragmatic reasons too. Pixel intensity is represented as a number so there is no problem with the combination of numerical and categorical features. Additionally, images does not have missing values or other irregularities; and although raw image datasets require a lot of preprocessing, there are several tidy datasets available.

10.1.2 Datasets

Datasets are usually randomly divided between training and test sets. We need to make this division because the objective of a final model is to perform well both on the data that we used to train it (e.g. the training dataset) and the new data on which the model will be used to make predictions (which is approximated by the test set).

- **Trainig set**: It is the set of samples that we use to build the model (i.e. estimate the parameters).
- **Test set**: It is the set that serves as a proxy for new data and provides an unbiased evaluation of a final model fitted on the training dataset. Usually this set represents a 20% of the data we have so it is much more smaller than the trainig set. It is very useful to do comparison between different models and it is necessary to detect **overfitting**. A model overfits when it learns the training dataset too well, performing well on these data but badly on the test set or in new data.

The characteristics of the training and test set have to be the same, meaning both sets have to come from the same population. Often there is another set, the validation set, that it is used to tune the hyperparameters of the models and perform model selection. But it is not always necessary.

In this project we use three datasets with different characteristics. Two datasets have only black and white images, while the other have colour images of bigger size. Next, I provide a description of the datasets:

Grey-scale images

MNIST [Web page] This is a dataset of 60,000 28x28 grey-scale images of the 10 digits, along with a test set of 10,000 images. It is a subset of a larger set available from NIST. The digits have been size-normalised and centred in a fixed-size image. In figure 10.1 you can see the digit 8 with the value of its pixels.

Fashion MNIST [28] This is a dataset of 60,000 28x28 grey-scale images of 10 fashion categories, along with a test set of 10,000 images. This dataset can be used as a drop-in replacement for MNIST. The dataset was created in 2017 by using the Zalando's article images (so it's a real example though it is processed). The class labels are from 0 to 9 respectively: T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle bot. In figure 10.2 you can see 60 random samples of the dataset.

Colour images

CIFAR-10 [29] This is a dataset of 50,000 32x32 colour training images and 10,000 test images, labelled over 10 categories. The 10 classes of the dataset are: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck. The problem here is that the images have backgrounds and are very tiny and therefore are harder to identify. The previous datasets have images with no background, in the images only appears the object of interest. In 10.3 there is an example of a dog.

The datasets are available directly in Keras, so it is not necessary to download them.

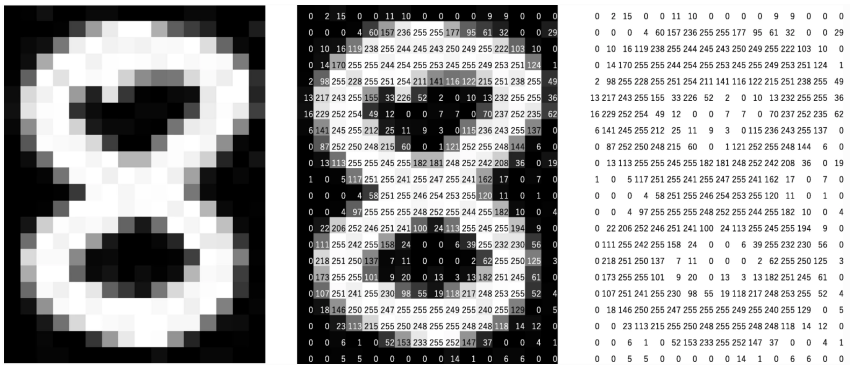


Figure 10.1: Representation of digit 8 - MNIST Dataset (Source: [30])

Figure 10.2: Example of some samples of the Fashion Dataset

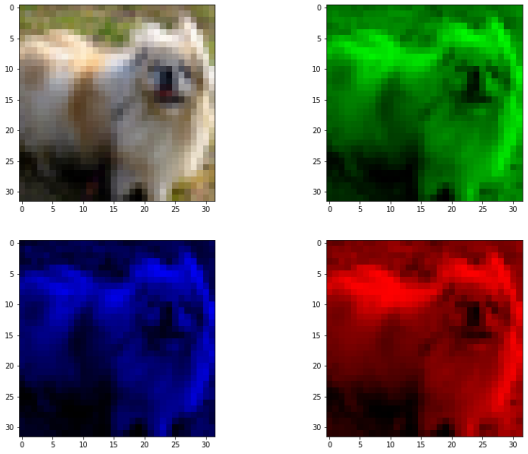


Figure 10.3: Example of Cifar10 RGB image

10.2 Deep Learning software

In this project I use three major programs: Python, a general programming language; and Tensorflow and Keras which are programs specialised in Deep learning.

10.2.1 Python

Python is arguably the best programming language for Machine learning applications. A part from the wide range of available functions and libraries it has available, it can also be used to easily access Deep learning software such as Tesorflow and Keras. Moreover, many cloud services provide support and are configured for Python.

Specifically, from Python I will use some libraries such as **numpy** for numerical manipulation, **matplotlib** for plotting and **Sklearn** for deploying machine learning algorithms.

10.2.2 Tensorflow

TensorFlow was originally developed by Google for internal use, but it was released on open-source in 2015. Tensorflow is a symbolic maths library used for machine learning and training neural networks, both in research and production. Today, it is the most popular framework for deep learning and it has a comprehensive, flexible ecosystem of tools, libraries, and community resources. Some of them can be found in the official github page.

10.2.3 Keras

Keras was originally created and developed by Google AI Developer/Researcher, Francois Chollet, and it was also released open-source on 2015. He wanted to make a high-level API, an easy-to-use program, to iterate on their experiments faster. Keras is only an interface to a program such as tensorflow, it does not perform the calculations. Keras requires a backend. A backend is a computational engine — it builds the network graph/topology, runs the optimizers, and performs the actual number crunching.

As of mid-2017, Keras was finally fully adopted and integrated into TensorFlow. This means that you can define your model using the easy to use interface of Keras and then adopt TensorFlow if you need i) specific TensorFlow functionality or ii) need to implement a custom feature that Keras does not support but TensorFlow does [31].

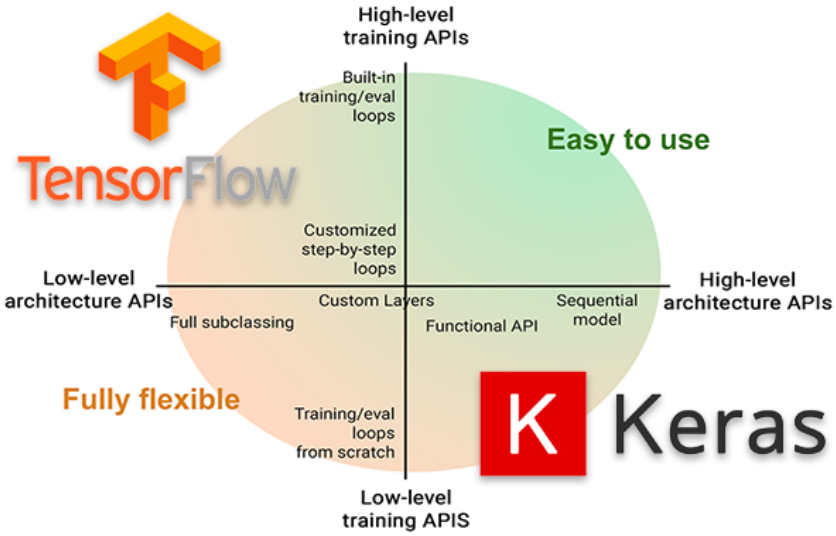


Figure 10.4: Functionality and easy of use relationship (Source: pyimagesearch.com)

10.3 Cloud computing

For this project I used **Google Colaboratory** (known as Colab) to do all the computations. Colab is a free tool on the cloud provided by Google that is useful for experimenting with machine learning models. In 10.5 you can see a screenshot of the software.

Colab allows you to execute Python in your web browser and i) it does not require any configuration, ii) it gives you free access to cloud servers, iii) it is easy to share your code with other people. For me the most important part was the access to computing power: with colab you can access GPUs and TPUs [1] which speed up substantially the training time of the neural networks, compared to a traditional computers that use only CPU cores.

Colab uses Jupyter notebooks, scripts similar to Rmarkdown documents, to write code an display its output. In this project, I used Colab to do the computations, but then I downloaded the notebooks and transformed them to *pdf* via Latex. One of the *pdfs* with the code is in the appendix, the others are available on Github (see next paragraph).

[1] The tensor processing unit (TPU), is an application-specific integrated circuit built specifically for machine learning and tailored for TensorFlow, which was announced by Google in 2016.

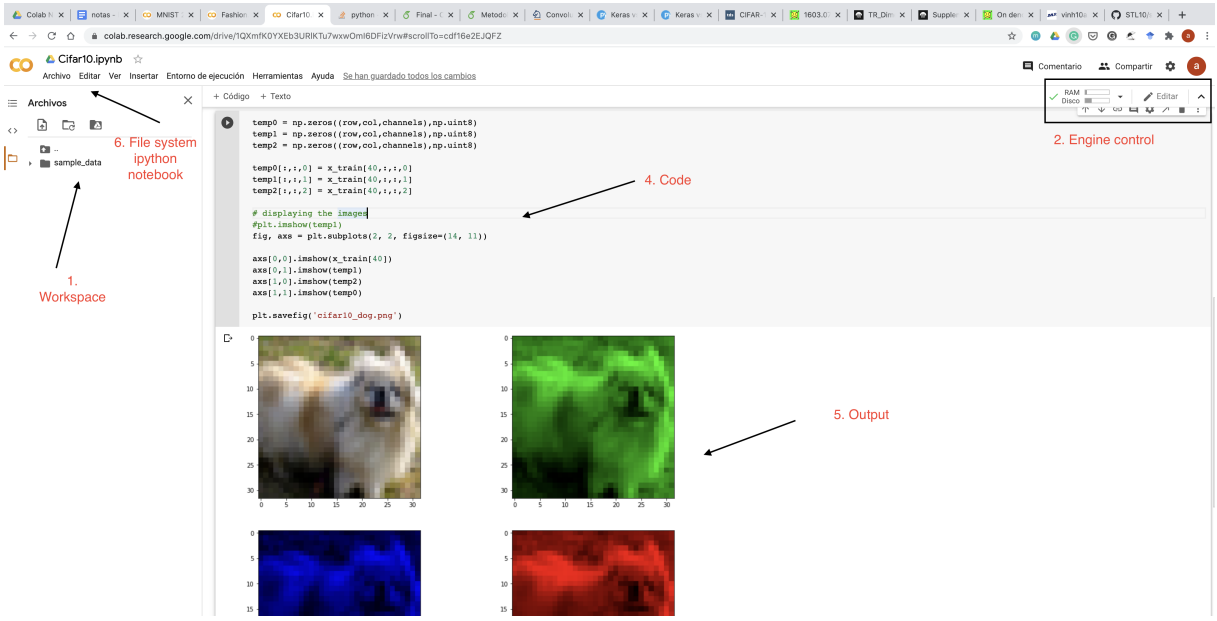


Figure 10.5: Google colaboratory system

10.4 Open code

Throughout the project I re-used code that was publicly available on Github. **Github** is a platform that allows you to share your code very easily. Often, machine learning researchers upload the code they have use in their publications on Github so that anyone can see or use it. This was useful especially when it came to the Deep embedding clustering algorithm since it is not a trivial problem. I was able to adapt the code that researchers had done. However, I had to modify some parts so that they could be run in Colab. I also have used Github to upload the Jupyter notebooks I have created throughout the project.

Organisation of the practical part

After the methodology, the applied section is form by three chapters. In these chapters I analyse comprehensibly the Fashion dataset and I provide a summary of the results of the other datasets to avoid repetition. The Fashion dataset is more complex and interesting than MNIST, which has already been analysed widely by the Machine Learning community. Moreover, it uses real-data, and since they are black and white images the calculations are faster than with colour datasets. All the code can be found in the appendix (for the Fashion dataset) or in Github, for the rest of the datasets.

The idea of these chapters is to experiment and analyse different dimensionality reduction and clustering techniques and how well they fit together. Therefore, I focus on rapid iteration and experimentation of the algorithms rather than in complex details.

Steps of the process:

The first step is to download and normalise the data. I use pixel intensities as input, though other measures are possible as well. The final range of the input's values is [0,1].

1. In the chapter of **dimensionality reduction** I use the following algorithms: PCA, UMAP, t-SNE and different Autoencoders. It is important to distinguish between DR for visualisation and DR for clustering purposes. This chapter focuses mainly on the former task.
2. In the chapter of clustering I differentiate two parts. In the first part I make clusters with the original data (**traditional clustering**) while in the second one I use embedded spaces of 2 and 30 dimensions to perform the clustering (**two-stages clustering**).
3. In the last chapter, I show how to implement the **Deep clustering** algorithms called DCEC. Additionally, I compare its results with those obtained in the previous chapters. It is especially interesting the comparison between two-stages clustering and Deep clustering.

Chapter 11

Applied dimensionality reduction

This chapter is divided in four sections: One for each dataset (Fashion, MNIST, Cifar) and another one for commenting some of the results. The largest one is that of the Fashion dataset as it is the dataset I analyse in detail. This section is divided between **traditional and machine learning** (PCA, t-SNE, UMAP) algorithms and the **Deep learning approach** (e.g. Autoencoders). The reason for that, it is that I want to focus especially in autoencoders because they are the main part of the DCEC algorithm.

To perform DR using PCA and UMAP I joined the training and test sets to get more observations. To train the Autoencoders I used the training set, but for plotting the results I employed the test set. However, to use t-SNE I selected a random sample of 10,000 thousand observations; otherwise, the calculation would be too slow, they would have taken too many hours.

Difference between visualisation and clustering In this chapter I focus mainly on DR for visualisation and in the next one I use DR to perform clustering afterwards. The approach is similar but it presents some major differences. Two of the most important are:

- For visualisation you need a reduced space of 2 or 3 dimensions; that is, the dimensionality may need to be reduced beyond the inherent dimensionality of data. It is well-known that a high-dimensional data set cannot in general be faithfully represented in a lower-dimensional space, such as the plane. Hence a visualisation method needs to choose what kinds of errors to make. On the other hand, in DR for clustering the number of dimensions can be arbitrary, though usually it is set between 10 and 50 dimensions. In this project I use mainly 10 and 30 dimension.
- Some values of the hyperparameters can be better for visualisation than clustering, and vice versa. This is especially important for manifold learning algorithms, as I will explain in the next pages. The same can happen with the architecture of a neural network.

11.1 Fashion dataset

11.1.1 Traditional algorithms

Principal component analysis

In figure 11.1 is displayed the output of PCA. In the left image you can see the representation with the observation differentiated by colour, which represent the ten different classes. In the right image, there is no additional information, it is a more realistic approach to unsupervised learning. The representation shows some structure but clearly a linear method is not enough in this case.

Implementing a model using *sklearn* is very straightforward: You only have to define the model and its parameters and then apply *fit()* to perform the training. The same instructions are valid for any model trained using this library.

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import scale

pcaf_2d = PCA(n_components=2)
pca_2d = pcaf_2d.fit_transform(x_full)
```

Listing 11.1: Implementation of PCA with sklearn

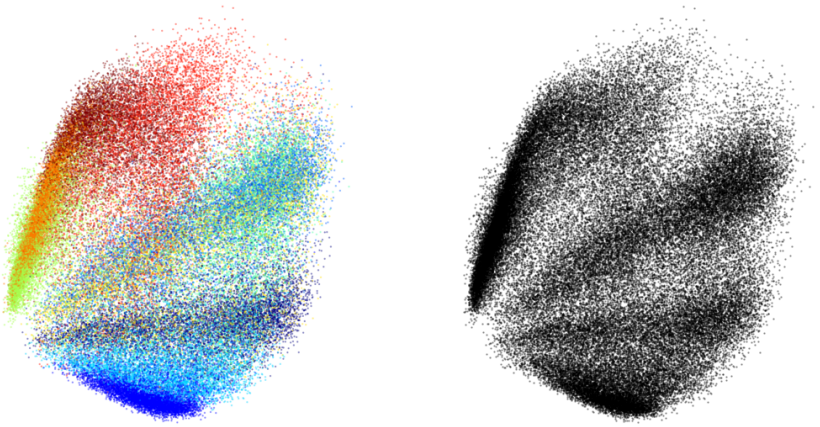


Figure 11.1: Visualisation by PCA - Fashion

UMAP

UMAP as t-SNE is highly dependent on its parameters, so it is important to choose them carefully. Moreover, some values are more adequate to clustering while others are more suitable for visualisation. Based on [32] and [10] I choose these parameters:

- **Visualisation**: UMAP(n_neighbors = 15 , min_dist = 0.1 , dimensions = 2)
- **Clustering**: UMAP(n_neighbors = 30 , min_dist = 0 , dimensions = 10)

The parameter **n_neighbors** controls how UMAP balances local versus global structure in the data. I will refer to this later on. Low values of n_neighbors will force UMAP to concentrate on very local structure (potentially to the detriment of the big picture), while large values will push UMAP to look at larger neighbourhoods of each point when estimating the manifold structure of the data, losing fine detail structure for the sake of getting the broader of the data. Therefore, for clustering it is preferable to use larger values, to take more into account the global structure.

The **min_dist** parameter controls how tightly UMAP is allowed to pack points together. It, quite literally, provides the minimum distance apart that points are allowed to be in the low dimensional representation. This means that low values of min_dist will result in clumpier embeddings. Larger values of min_dist will prevent UMAP from packing point together and will focus instead on the preservation of the broad topological structure instead. In clustering it is preferable to use a value as low as possible, hence the 0.

```
1 import umap
2
3 fit = umap.UMAP( n_neighbors=15, min_dist=0.1, n_components=2, metric="euclidean
      ", random_state=semilla)
4 umap_2d = fit.fit_transform(x_full)
```

Listing 11.2: Implementation of UMAP with sklearn

In figure 11.2 you can see **four big groups**: On the left, there are grouped the ankle boot, the sneaker and the sandal classes. On the top of the image, in blue there is the trouser category. On the left-bottom, in red you can see the bag class. Finally, on the right, the are the coat, shirt, dress, pullover and t-shirt/top. Moreover, in this last group, the shirt and t-shirt/top categories are quite mixed as it is logical since they are quite similar.

However, using a different parameters, with the objective of doing clustering, it is possible to distinguish better the clusters (see the appendix for more visualisations).

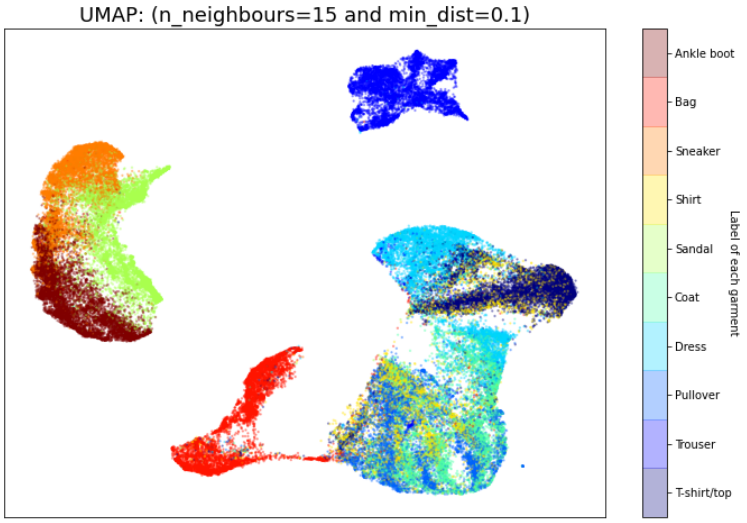


Figure 11.2: Visualisation by UMAP - Fashion

t-SNE

To perform t-SNE I selected a random sample of 10,000 observations because otherwise it would take too much time. Using this sample it took 11 minutes to finish the calculations. Based on [33] and [18] the values I choose are Perplexity = 25, and a learning rate = 10. The perplexity value is similar to the parameter n_neighbors of the UMAP; it balances the local and global structure.

```
from sklearn.manifold import TSNE

tsne1 = TSNE( n_components= 2, perplexity= 25.0, learning_rate= 10.0,
              n_iter=5000, random_state= semilla, n_iter_without_progress=200 )
tsne_results = tsne1.fit_transform(x_rfull)
```

Listing 11.3: Implementation of t-SNE with sklearn

Figure 11.3 shows the visualisation of the dataset using the aforementioned parameters. Surprisingly, some classes such as Bag, Sandal and Sneaker are split in two. The algorithm has created more groups than there really are. This is a common problem. As stated in [33], *You can see some shapes, sometimes*, and *Random noise doesn't always look random*. I would say they are common problems not only in t-SNE, but in many DR techniques. However, the separation between some groups, for example between t-shirt/top, coat and dress seems better than that of UMAP.

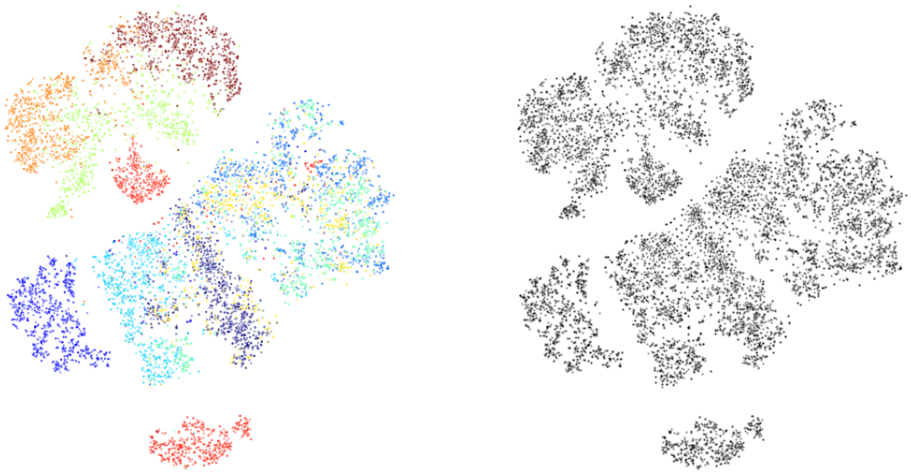


Figure 11.3: Visualisation by tsne - Fashion

In the appendix you can find more visualisations with different parameters.

11.1.2 Autoencoders

Autoencoders can be used for visualisation when the embedding layer has two or three nodes but they also work very well for clustering with higher dimensions because the computational cost of adding more nodes to that layer (and thus, increasing the dimensions of the embedded space), is marginal.

In this subsection I will show two structures of convolutional autoencoders: The first one is created by myself and the second is the one used for the DCEC in its paper [Guo et. al., 2017 27]. Both autoencoders will be reused in the next chapter applied to deep clustering.

Figure 11.4 illustrates how a neural network process an image. The intensity pixels of the image are concatenated in a large vector. In the case of the MNIST dataset, images have dimensions 28x28 and thus, the input of the autoencoder has 784 elements or "variables", loosely speaking.

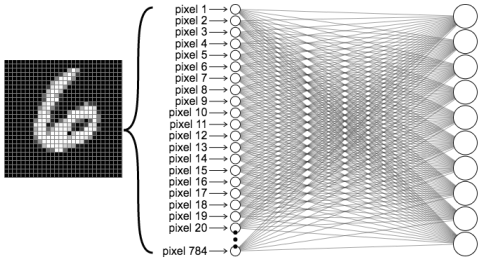


Figure 11.4: Input layer (Source: ml4a.github.io)

Description of the structure

In page 70 you can see the first autoencoder's architecture created by myself. I take inspiration from [34] which creates a similar autoencoder for the MNIST dataset.

Keras represents each layer as a box, and in each box there are several elements: the name and type of layer at the left, and at the right the input and output's dimensions. The dimensions are represented by a tuple of four elements: (? , width , height , depth or channel). The interrogation symbol represents the number (or id.) of the observation. As you can see, the input has a value 1 on the depth as it is a grey-scale image, but convolutional layers can have an arbitrary depth (or what is called number of kernels).

The first autoencoder has:

- 19 hidden layers and a structure that is not completely symmetrical.

- A bottleneck or embedded layer that has 10 dimensions. Therefore, the reduced space is not feasible for visualisation but it is for clustering.
- An encoder, which decreases little by little the dimensions of the input, and a decoder, that increases gradually the size of the embedded space.
- The encoder is form by convolutional and max pooling layers; the decoder uses convolutional and transposed convolutional layers and the upsampling (which is equivalent to the max pooling).
- A Binary cross-entropy loss function. As the inputs are normalised (their values range from 0 to 1), it is fine to use BCE instead of RMSE.
- The Adam optimisation algorithm, with early stopping, 200 epochs, a mini-batch size of 128, and shuffle of samples.
- It does not use any regularisation method as they did not improve the loss.

Summary of Keras implementation

Here you can see the basic implementation using Keras:

```
# 1. Defining the model
CAE = Model(input_img, decoded)
# 2. Instruction for the compilation
CAE.compile(optimizer='adadelta', loss='binary_crossentropy',metrics=['
    mean_squared_error'])

callbacks_list = [ EarlyStopping( monitor='val_loss', patience=10 )]

# 3. Training execution
history_CAE = CAE.fit(x_train_c, x_train_c,
                epochs=150,
                batch_size=128,
                shuffle=True,
                callbacks=callbacks_list,
                validation_data=(x_test_c, x_test_c))
# Next steps: 4. Evaluate and 5. Make predictions
```

Listing 11.4: Implementation of a CAE with Keras

The implementation with Keras is quite straightforward and it only requires 5 steps:

Model You have to indicate which is the model (e.g. architecture) you want to use. You can see an example in page 70, and another different in 75. The layers have to been defined in advanced.

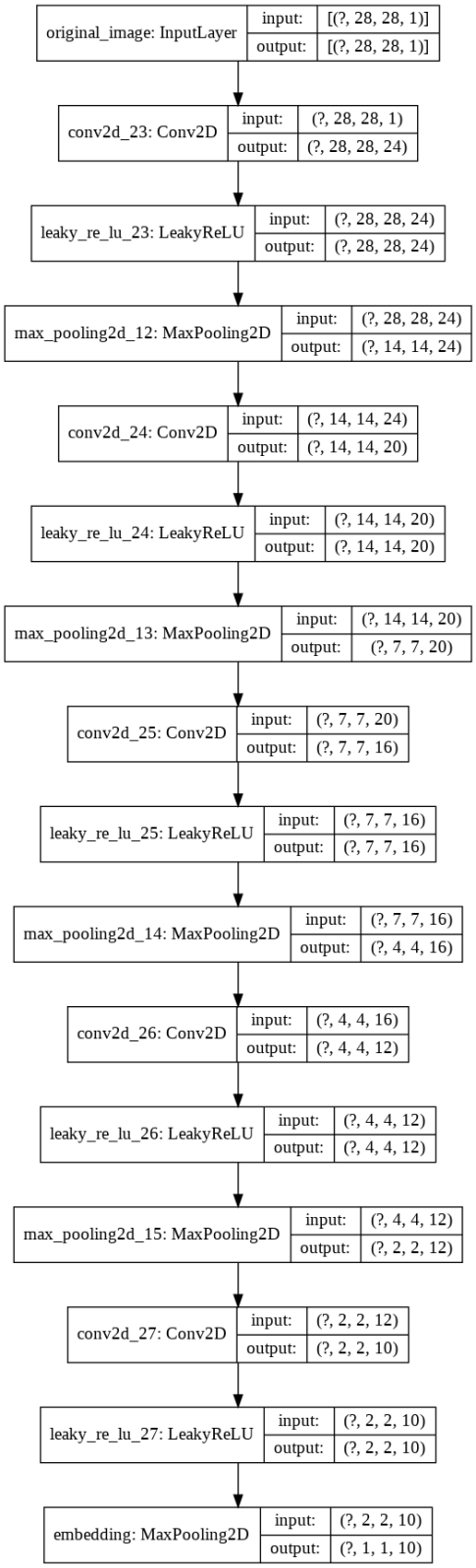


Figure 11.5: Encoder of the CAE

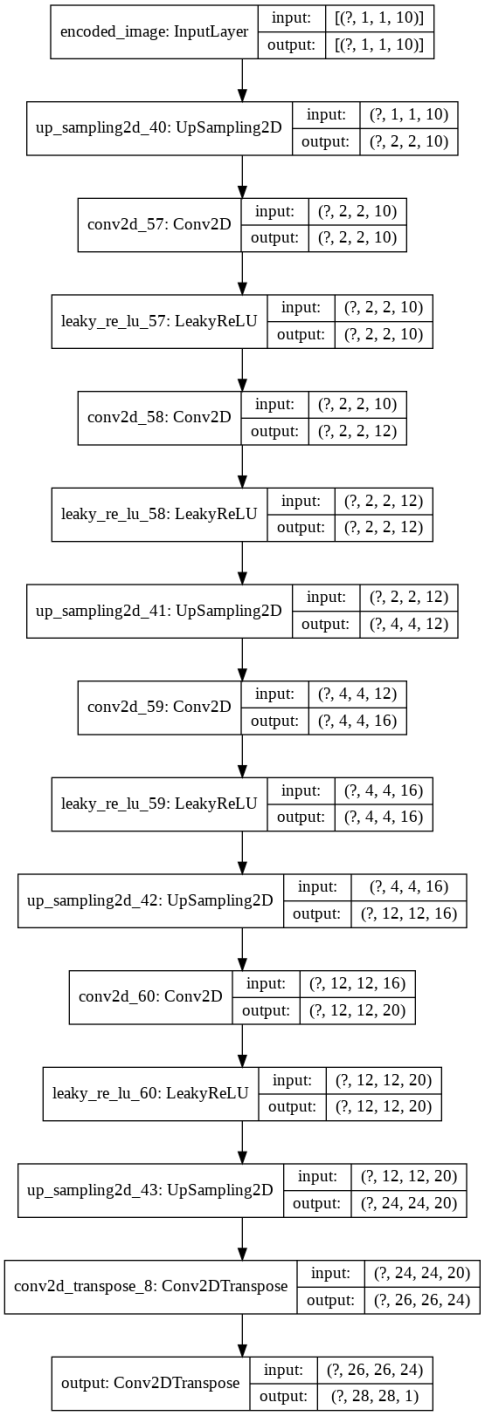


Figure 11.6: Decoder of the CAE

Compilation You have to indicate mainly the optimisation algorithm (Adadelta is a variation of stochastic gradient descent), and the loss function.

Fitting As with *sklearn* there is a fit() function. Here it is important the number of epochs, the batch size and the training and validation data. Exceptionally, in autoencoders the X and target y are the same data (x_train_c for the training in this case).

Evaluation When the parameters (W, b) are already estimated, you can do the evaluation using the test set.

Training

During the training you have to be careful about overfitting since neural networks are prone to that. The evolution of the training and test sets loss is depicted in figure 11.7. Both losses decrease very rapidly during the first iterations and then the pace is slower.

In figure 11.8 you can see some sample outputs (or reconstructions), from the autoencoder (second row), together with their original inputs (first row). Seeing the differences it becomes apparent that the autoencoder focus on reconstructing the general structure or shape of the objects and not the details. The shoelaces do not appear in the reconstructions, the same that the letters or patterns of the sweaters and shirts.

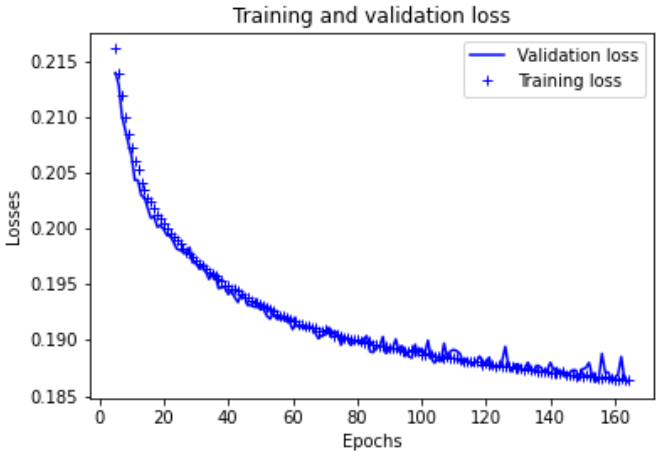


Figure 11.7: Evolution of the losses

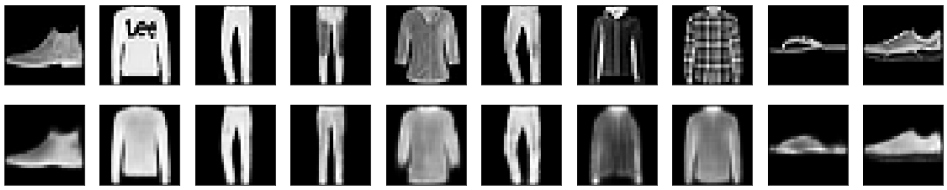


Figure 11.8: Random samples reconstructed - Fashion

Visualisation

To visualise the data, it is only necessary to change the depth of the intermediate layer: From 10 to 2 nodes.

There are many more visualisations on the appendix and Github. In all the cases though, footwear categories are not well separated as happened with the other methods and the *bag* class is separated in two. However, the output of the autoencoder is quite similar in the structure to that of the PCA: The shapes of the cluster are long and oval, and observations are quite spread. Reducing the data to 128 dimensions with an autoencoder and applying UMAP afterwards (image 11.10), gives results similar to those obtained by using only UMAP. It is not clear which ones are better.

Contrary to other methods, it is unclear how the architecture can modify the structure of the embedded space, there is no theoretical basis. However, to compare to autoencoder you can always see which one has a lower loss after training, meaning, which one reconstruct better the output from the embedded space. Hopefully, this will mean that the latent space has captured more information.

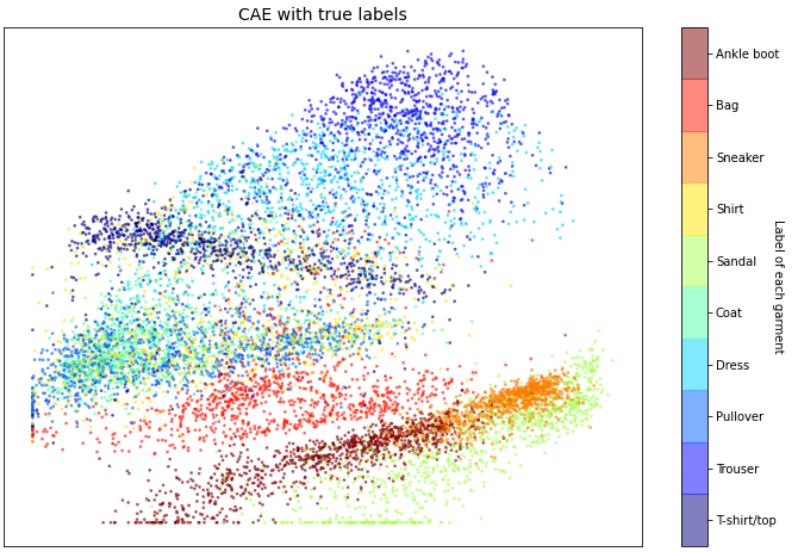


Figure 11.9: Visualisation by CAE - Fashion

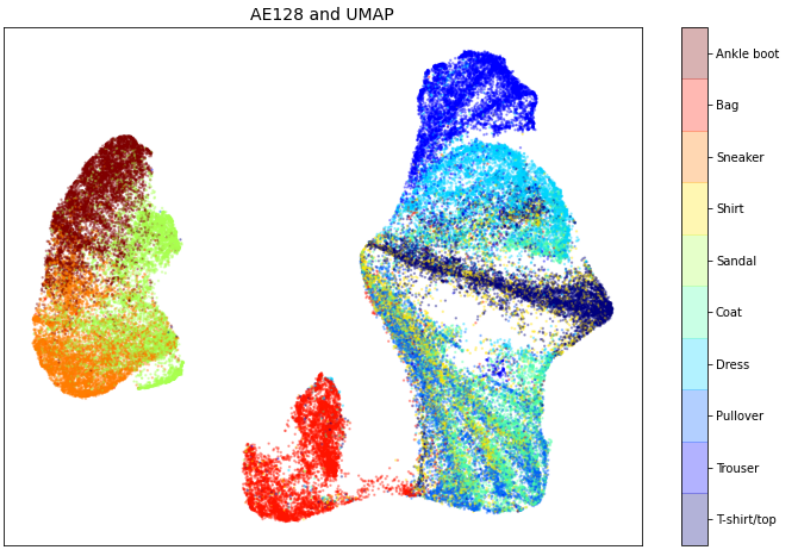


Figure 11.10: Visualisation by AE128d and UMAP - Fashion

A different structure

The second ANN architecture I tried is proposed by [Guo et. al., 2017 27] and shown in 11.12a. I wanted to get a sense of its performance (for visualisation) before applying it to clustering with the DCEC.

Figure 11.12b is a summary of the Keras implementation. This architecture is quite different from the other. It has many more parameters, in part because uses two fully connected layers. It is possible to have CAE with a lot of layers since they do not use many parameters, but with only fully connected layers this approach would not be possible. Furthermore, the encoder increases gradually the number of parameter, which is the opposite of what does the other network.

Figure 11.11 shows the result of that network with 2 nodes in the embedded layer. The binary cross entropy in the test set has a value of 0.325 while for other CAEs it was below 0.31. The output is constrained to positive values and seems to do not work very well.

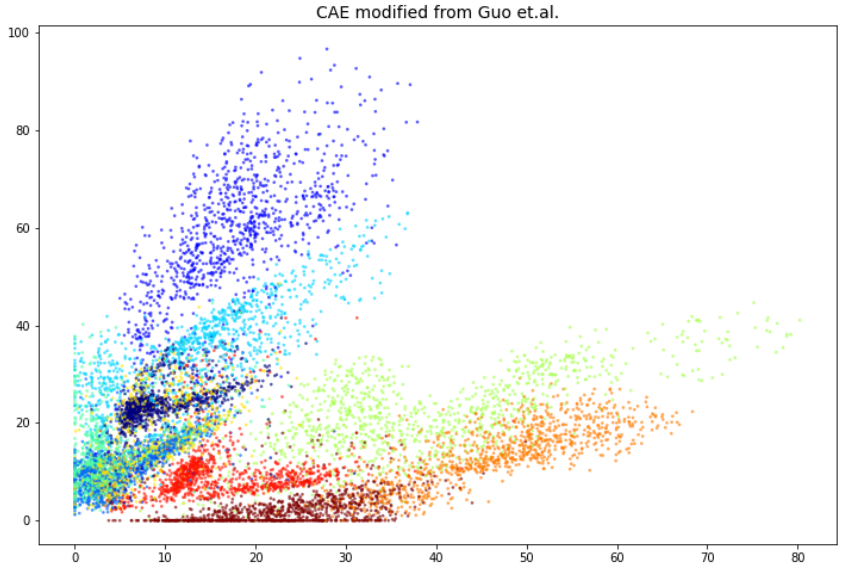


Figure 11.11: Visualisation by CAE Guo et.al. - Fashion

4 Xifeng Guo et al.

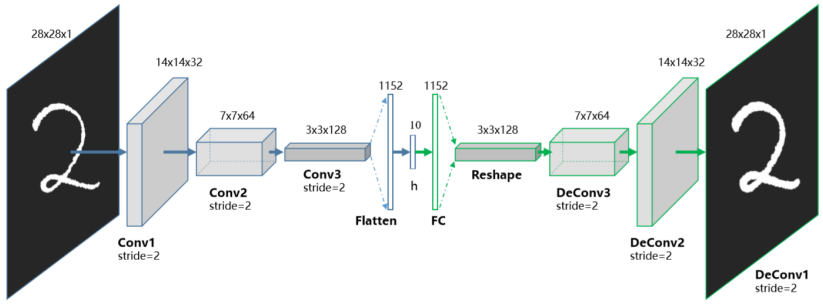


Fig. 1. The structure of proposed Convolutional AutoEncoders (CAE) for MNIST. In the middle there is a fully connected autoencoder whose embedded layer is composed of only 10 neurons. The rest are convolutional layers and convolutional transpose layers (some work refers to as Deconvolutional layer). The network can be trained directly in an end-to-end manner.

(a) Image from paper

```
Layer (type)                 Output Shape              Param #
=================================================================
input_9 (InputLayer)         [(None, 28, 28, 1)]       0
_________________________________________________________________
Conv1 (Conv2D)               (None, 14, 14, 32)        832
_________________________________________________________________
Conv2 (Conv2D)               (None, 7, 7, 64)          51264
_________________________________________________________________
Conv3 (Conv2D)               (None, 3, 3, 128)         73856
_________________________________________________________________
flatten_8 (Flatten)          (None, 1152)              0
_________________________________________________________________
embedded (Dense)             (None, 10)                11530
_________________________________________________________________
dense_8 (Dense)              (None, 1152)              12672
_________________________________________________________________
reshape_8 (Reshape)          (None, 3, 3, 128)         0
_________________________________________________________________
DeConv3 (Conv2DTranspose)    (None, 7, 7, 64)          73792
_________________________________________________________________
DeConv2 (Conv2DTranspose)    (None, 14, 14, 32)        51232
_________________________________________________________________
DeConv1 (Conv2DTranspose)    (None, 28, 28, 1)         801
=================================================================
Total params: 275,979
Trainable params: 275,979
Non-trainable params: 0
```

(b) Keras implementation

Figure 11.12: CAE proposed by Xifen Guo et.al.

11.2 MNIST dataset

In this dataset the classes can be distinguished clearly using t-SNE 11.14 and UMAP 11.16. The CAE 11.17 also gives a good overview of the structure of the dataset but the shape of the groups is very long and oval (it is similar to that of the Fashion dataset). They do not seem to conserve well the structure of the clusters (or local structure), although the vertical axis distinguishes better the cluster than the horizontal axis. Lastly, PCA 11.13 presents some structure but it is highly insufficient.

Based on the results of this dataset, and although this dataset is quite simple, it seems that DR techniques can be very useful, not just for visualisation but also (and especially) for clustering. However, I would say it is difficult to generalise these results.

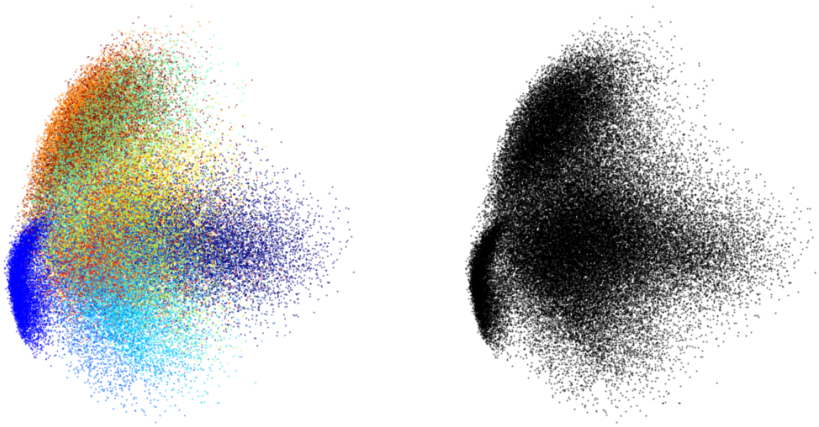


Figure 11.13: Visualisation by PCA - MNIST

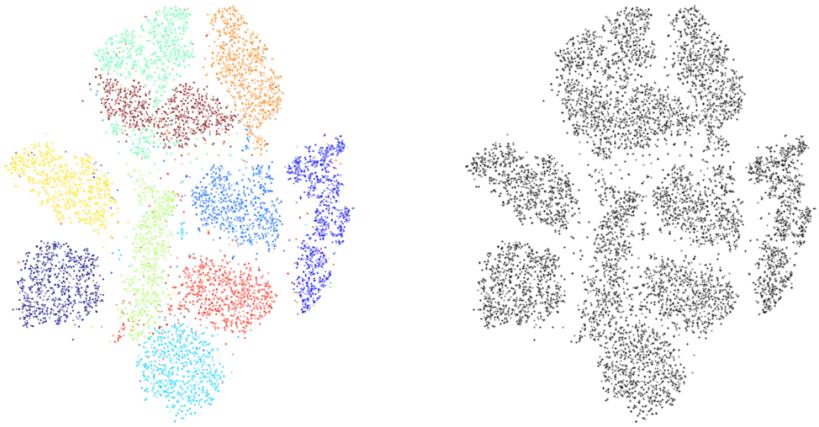


Figure 11.14: Visualisation by t-SNE - MNIST

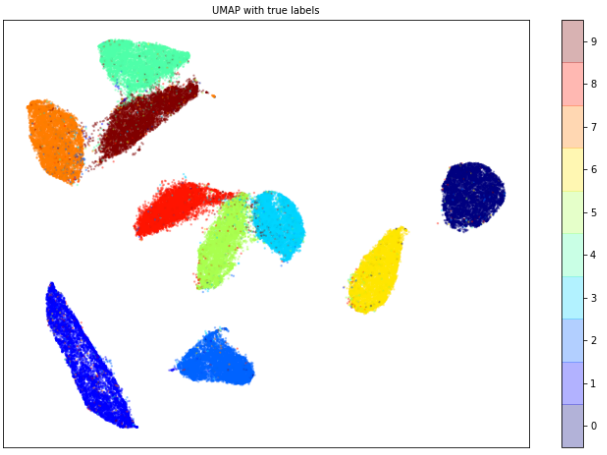


Figure 11.15: UMAP: (n_neighbors= 15, min_dist=0.1) - MNIST

Figure 11.16: UMAP: (n_neighbors= 30, min_dist=0) - MNIST

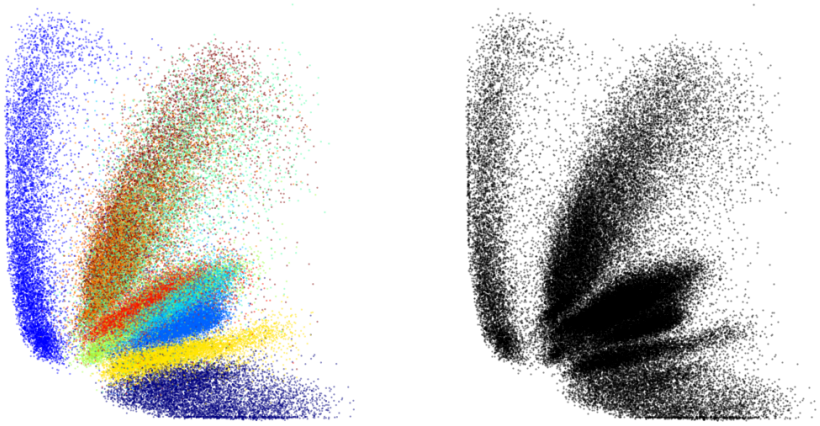


Figure 11.17: Visualisation by CAE - MNIST

11.3 Cifar 10

To perform the visualisation in this dataset I selected a random sample of 10,000 observations. Figure 11.18 shows the visualisation using UMAP: Unexpectedly, you cannot see any structure in this image. Moreover, the results were very bad no matter the algorithm used. It is clear that not all datasets have an structure, but Cifar 10 do have 10 different classes, so unsupervised method should be able to capture some features of these classes.

One possible cause of this bad performance can be the background of the images. As mentioned in the description of the datasets in page 56, the previous datasets have images with no background, in the images only appear the object of interest. Another possible problem is that images have only 32x32 pixels, which is not much. Lastly, the fact that they are colour-images adds another layer of complexity to the dataset.

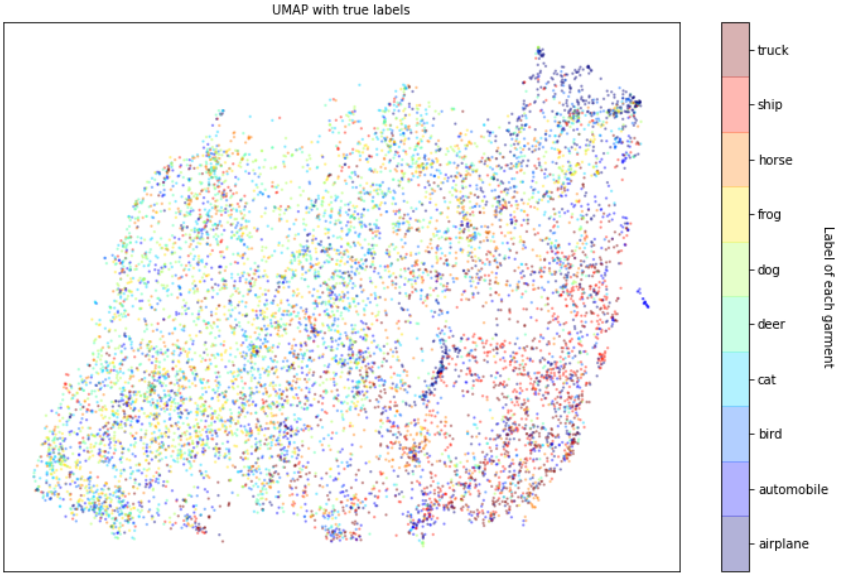


Figure 11.18: Visualisation by umap - Cifar

Some domain-specific techniques, such as Histogram of oriented gradients (HOG), can help to apply unsupervised learning. But their implementation is non-trivial.

11.4 Discussion of the results

Most of the problems related with dimensionality reduction have been explain in the theoretical section 3.1 in page 9. Here I want to focus on structure preservation which is especially important in the manifold learning algorithms.

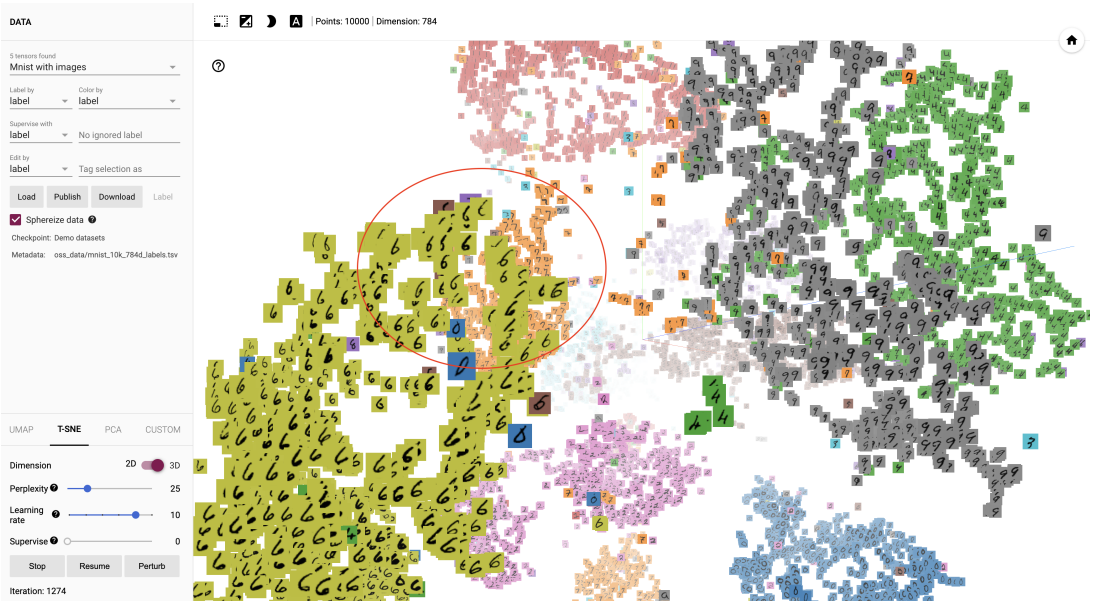
11.4.1 Global and local structure preservation

There is no way to map high-dimensional data into low dimensions and preserve all the structure. So, any approach must make trade-offs, sacrificing one property to preserve another. Here, I will focus on the global-local structure preservation trade-off, which is quite important especially in manifold learning algorithms since they are not capable of retaining both structures.

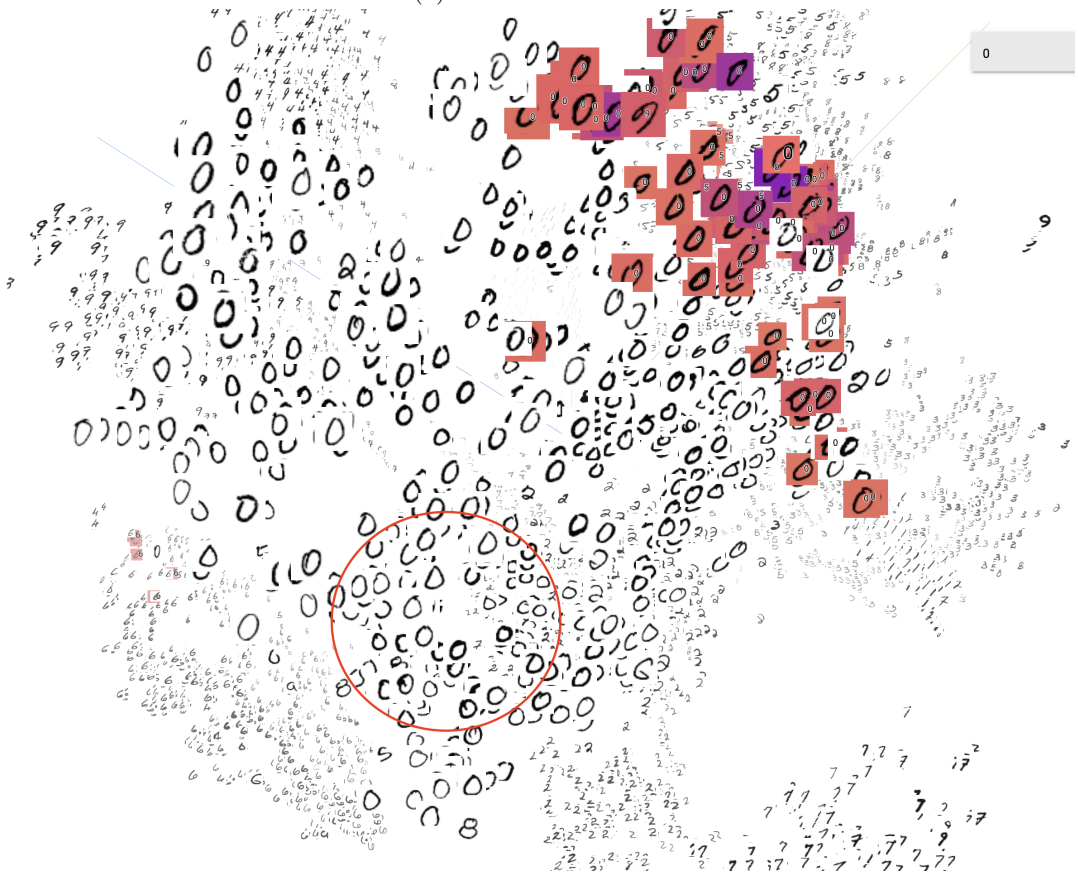
By global data structure it is usually considered i) the distances between the clusters, ii) the correlation between original and transformed centroid coordinates or iii) the shapes of the clusters. Additionally, in practice, we consider that local structure preservation is important for visualisation while global structure preservation is important for clustering [35].

Structure exploration with the MNIST dataset

The MNIST dataset has been widely explored and several visualisations have been made. Using the projector project by tensorflow it is possible to see a 3D dynamic web-based visualisation of this dataset. In the first figure 11.19a, you can see that the numbers 6 that are more similar to the number 9 are closer to that cluster: These samples are long and are stretch out, plus the bottom of the number is similar to a circle (like the one that the 9 has). Global structure works for far samples, between clusters. On the other hand, in 11.19b you can see local structure preservation. Within the same cluster of the number 0, the numbers that are in the lower part of the image have a circular shape. Those that are in the upper part of the image, are longer and are narrower.



(a) Global structure



(b) Local structure

Figure 11.19: Structure preservation - MNIST

Chapter 12

Applied clustering

In this chapter I will perform clustering in **i)** the original datasets and **ii)** the embedded spaces obtained in the previous section. Both approaches can lead to very different results. The third approach, which is Deep clustering, will be applied in the next and last chapter.

To perform the clustering I used the K-means and HDBSCAN algorithms from the Sklearn and hdbscan library. These algorithms are very different from each other, and thus, it is interesting for making comparisons. On the other hand, both suffer from the *curse of dimensionality* since K-means is based on euclidean distances and HDBSCAN is base on densities. However, I am more interested in K-means because it is a crucial part in the DCEC algorithm. Other clustering algorithms could also been used, such as Spectral, Agglomerative clustering or Finite mixture models.

K-means has been explained in chapter 8 (page 46), but just as a reminder we can say that: K-means works well when clusters are convex, spherical and equally dense; and it responds poorly to elongated clusters, manifolds with irregular shapes, unevenly size groups or when data have noise. Clearly it is not always the case that clusters have the proper kind of structure.

HDBSCAN stands for "Hierarchical Density-Based Spatial Clustering of Applications with Noise" and is a density based algorithm that also uses hierarchical (agglomerative) clustering. It is especially robust to noise and it only makes clusters when there is a "high confidence" to do so. However, in practice the major difference with K-means is that not all the observations have to belong to a cluster. Observations can be characterised as **'noise'**. Space does not allow for a comprehensive description and I refer to [36] and [37] for more information.

I place all the clustering results in two tables: In the first one 12.1 you can find the results for

the Fashion dataset using several algorithms and four evaluation metrics. In the second one 12.2, I put the results for all the datasets using only the normalized mutual information (NMI) metric.

12.1 Clustering with the original data

12.1.1 K-means

k-means does not need any parameter. One just have to indicate the number of clusters, the number of initialisations and a seed. It takes roughly 60 seconds to run, so it scales very well.

```
import time
kmeans_raw = cluster.KMeans(n_clusters=10, n_init=10, random_state=1997, init='k
    -means++', n_jobs=-1 ).fit(x_full)
kmeans_labels = kmeans_raw.labels_
```

Listing 12.1: Implementation of K-means with sklearn

Figure 12.1 shows the representation of the ten cluster centres [1] obtained through K-means. They resemble reasonably well to actual images of the dataset and they all represent different objects. Only the second image of the first row, and the third one of the second row, cannot be properly identified. These clusters probably have mixed several classes.

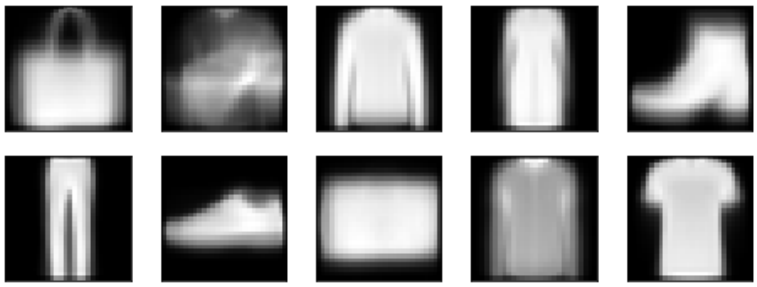


Figure 12.1: K-means cluster's centres

12.1.2 HDBSCAN

The implementation of the algorithm is not done with *sklearn* but with the library *hdbscan*.

[1] Cluster centroids are calculated using the average of the observations of that cluster, so they are not actual observations of the dataset.

```
!pip install hdbscan
import hdbscan

clusterer = hdbscan.HDBSCAN(
    min_samples=10,
    min_cluster_size=500
    ).fit(embedding)
```

Listing 12.2: Implementation of HDBSCAN

The **min_cluster_size** fix the smallest size grouping that you wish to consider a cluster. A value of 500 seems reasonable as the sample is form by 60 thousand observations and it is known that each class represent roughly a 10% of the dataset. It is important to note though, that this parameter not only affect the minimum size of the cluster but also changes the optimisation process (and thus, the results).

The **min_samples** parameter provides a measure of how conservative you want your clustering to be. The larger the value, the more conservative is the assignation of clusters. Here, we want a small value because preferably we want to assign a group to each observation (the dataset does not have "noise").

To use the algorithm I first use PCA to reduce the sample to 50 dimensions (the dataset maintains roughly 85% of the total variance). It is important to apply this step because HDBSCAN does not scale too well and additionally it does not work well with high dimensional spaces. The result for the Fashion dataset is the following:

```
Cluster:       [  -1 (noise) |    0   |   1  |    2  ]
Num. of obs:   [      45203  |  5530  | 8454 |  10813]
```

Listing 12.3: Output hdbscan

Therefore, it considers that the 64.5% of the dataset is noise. Apart from that, it creates three groups. The performance does not seem good. However, if we only use the points which have a cluster assigned to calculate the evaluation metrics, we obtain a NMI of 0.662 and an accuracy of 52%, which is similar to that obtained using K-means.

12.2 Clustering with the reduced spaces

To reduce the original high dimensional space I use UMAP, t-SNE and Convolutional Autoencoders. I show in detail the results obtained using UMAP (n_neigbors= 30, min_dist = 0), but you can find the rest in the Appendix. One advantage of UMAP with respect to t-SNE for example, is that it scales very well with the number of components of the embedded space. t-SNE is only feasible for 2D spaces due to the computational cost, but UMAP can produce an arbitrary dimensional space.

K-means

The results of applying K-means to the output of UMAP are shown in figure 12.2. It is important to note that the labels assigned by K-means have no inherent meaning, thus, in this figure the colours are only useful to distinguish between groups within the same plot. Next, I will describe the different boxes:

- **UMAP with true labels**: This is the same visualisation of the previous section, using the same colours.
- **UMAP with prior clustering**: K-means is applied to the original data and the resulting groups are projected into the 2D UMAP visualisation.
- **UMAP 2 components with posterior clustering**: K-means is applied to a 2 dimensional space produce by UMAP and the resulting groups are plotted in the 2D UMAP visualisation.
- **UMAP 30 components with posterior clustering**: K-means is applied to a 30 dimensional space produce by UMAP and the resulting groups are projected into the 2D UMAP visualisation.

The two boxes of the second row are very similar, so it does not make a big difference the number of dimensions of the embedded space. Moreover, the red group, which represents the Bag object as shown in the first plot, is divided into two groups no matter the clustering strategy used. Between applying K-means to the original and to the embedded space there are more differences: Surprisingly, the left group (form by the ankle boot, the sneaker and the sandal), is better represented with the prior clustering. However, overall, as shown in table 12.1, clustering in embedded spaces has a better performance.

HDBSCAN

The results using HDBSCAN are very different from that form by K-means. With PCA and HDBSCAN as explained before, we obtained three clusters. With 2-dimensional UMAP the algorithm obtains six cluster, but using 30 dimensions the number of final cluster is five. For the last two cases the results are shown in 12.3 and 12.4. Additionally, we can see that there are very few "noise" observations, which are represented in grey colour. Most of them are between the red and orange clusters of the first figure.

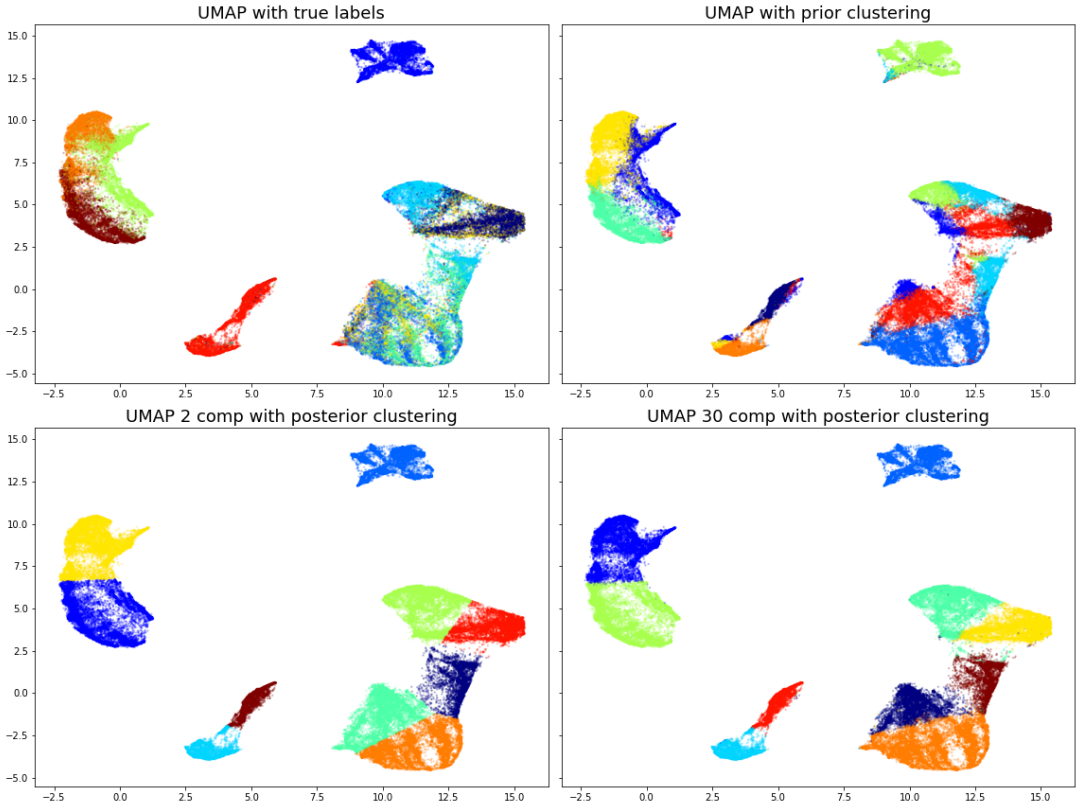


Figure 12.2: UMAP and K-means

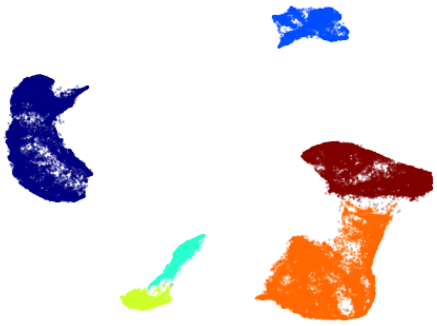


Figure 12.3: HDBSCAN with 2D UMAP

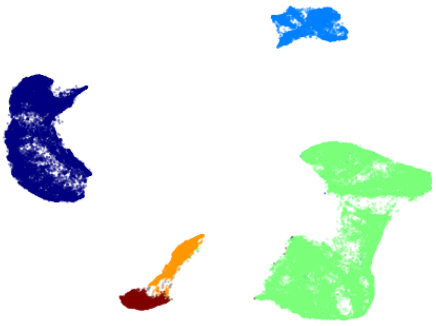


Figure 12.4: HDBSCAN with 30D UMAP

12.3 Evaluation of the clustering results

All the metrics calculated with HDBSCAN ignore the points that are not classified, which makes difficult doing comparisons with k-means. However, here I considered that what it is important is to assign the clusters correctly even though some observations are left without having an assigned group.

12.3.1 Fashion dataset

Table 12.1: Quantitative clustering performance

Metric	NMI	ARI	ACC	SIL
K-means full	0.5284	0.3844	0.5514	0.1517
PCA+HDBSCAN	0.6621	0.4583	0.5275	-
t-SNE2+K-means	0.5842	0.4502	**0.6163**	-
UMAP2+K-means	**0.6433**	**0.4800**	0.5746	0.132
UMAP30+K-means	0.6373	0.4780	0.5833	0.088
UMAP2+HDBSCAN	0.6422	0.3970	0.4361	-
UMAP30+HDBSCAN	0.6064	0.2839	0.3450	-
CAE10+K-means	0.4862	0.3209	0.5087	0.0988

To evaluate the clustering results I used, as stated in page 41, the **Normalised mutual information** (NMI), **Adjusted rand index** (ARI), **Clustering accuracy** (ACC) and **Silhoutte score** (SIL). The Silhouette metric is the only measure which does not make use of the external labels. However, turned out to be very slow to calculate. Another problem is that the measure is generally higher for convex clusters than for other concepts of clusters (density-based). That is why I only calculated this score for K-means.

In any case, the values near zero of the SIL indicate that there are overlapping cluster or that the clusters are not very dense. Using the other metrics it is possible to see that running K-means on low dimensional representations (with t-SNE and UMAP) is the approach that achieve the best performance. Surprisingly, the number of dimensions does not have much influence and, in fact, some metrics are better using only two dimensions.

It is clear though, that this results are very dependent on the parameters chosen, the dataset and the randomly selected samples for some algorithms.

12.3.2 All the datasets

Table 12.2: Quantitative clustering performance (NMI)

Dataset	Fashion	MNIST	Cifar10
K-means full	0.5284	0.4998	0.0793
PCA+HDBSCAN*	0.6621	0.9950	0
t-SNE2+K-means	0.5842	0.7663	-
UMAP2+K-means	0.6433	0.8313	-
UMAP2+HDBSCAN	0.6422	0.9177	-
CAE10+K-means	0.4862	0.2124/0.8898	-

First, unfortunately, some of the results of the Cifar10 dataset were not very reliable so I did not include them. For example, HDBSCAN do not detect any cluster, all the observations are noise. Comparing the other two datasets, the results are much better in MNIST, except for K-means and CAE10+K-means that surprisingly perform worse (in a simpler dataset).

HDBSCAN assign clusters very well (ignoring the observations considered as noise). However, in the first case (PCA+hdbscan), if we include the noise the NMI decrease to 0.3. This is not the case in UMAP2+hdbscan where the NMI with noise equals 0.905, mainly because almost all the observations have a cluster assigned.

Here again, the state-of-the-art DR methods improve the performance of clustering. The same cannot be said for autoencoders. Its results are not clear and it seems that they do not perform as well as expected. Trying several architectures in MNIST the results for NMI ranged from 0.21 to 0.89.

12.4 Discussion of the results

I already have done a theoretical discussion in the section 7.3 (page 43). Here I will try to explain briefly how some of them arise in practice while commenting the results obtained.

Choose the appropriate clustering algorithm This is arguably the most important decision to make and it is especially relevant to the autoencoder's output: As shown in figure 12.5, if k-means is used in some of the autoencoder's embedded spaces, the results can be bad. And this is exactly what the DCEC does: It considers that CAE and K-means can give a high-confidence estimation of clustering centres. Other clustering algorithms than k-means, can be better, though in other datasets this may not be the case.

Deciding the number of clusters Most of the DR techniques separated the BAG class in two groups. It is common to create more groups than there really are which can damage the results if you choose an incorrect number of clusters. Maybe it is interesting to try other algorithms as Hierarchical clustering which can help to select the number of cluster *a posteriori* to get a better idea of this problem.

Deciding the number of components The difficult question to ask is, how to decide the number of components of the reduced space? There is a trade-off: With a higher number of dimensions the curse of dimensionality may affect more the clustering algorithm, while with fewer dimensions you are losing more information.

It is clear that 30 UMAP components retain more structure of the dataset than 2 components and it is also clear that the intrinsic structure of these images is higher than 2. Moreover, the size of the sample is quite large, and with a large number of observations it is possible to use more dimensions without negative consequences. However, surprisingly in some cases the clustering evaluation metrics were better using two dimensions instead of thirty.

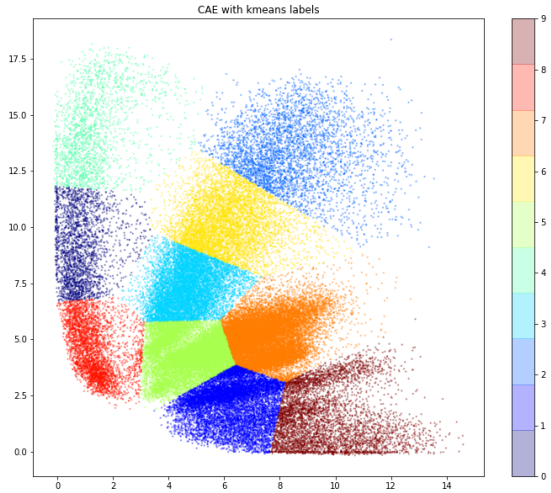


Figure 12.5: K-means with CAE output - MNIST

Chapter 13

Applied Deep Clustering

This chapter is form mainly by two sections. In the first one, I show how is the implementation and the training of the **DCEC**, one of the state-of-the-art Deep clustering algorithms, and I discuss some possible modifications. In the other section I compare its results with those obtained in the previous clustering chapter.

Regarding the DCEC, I used the implementation proposed by Xifeng Guo et. al. which is explained in [27], and the code is publicly available. From now on, I will refer to this paper. To perform the calculations I used only the MNIST and Fashion datasets because Cifar10 did not give good insights. Additionally, I grouped the training and test sets of these datasets and I trained the DCEC using the full datasets in order to have comparable results with those obtained in the article.

The DCEC has been explained in detail in 9.1 (page 49), and some of its components have already been put into practice, such as the convolutional autoencoder presented in page 74. The algorithm learns jointly good representations and cluster formation instead of doing both tasks in separate ways. It is expected to perform better than traditional algorithms, especially for image data.

13.1 DCEC Implementation

The code used in the paper can be found in this Repository. The convolutional autoencoder is implemented in the script *ConvAE.py* using a Sequential model from Keras, and the DCEC is implemented in *DCEC.py* using the Model Subclassing also from Keras [1].

[1]There are three ways to create models in Keras. The Sequential model and the Functional API are similar and are quite easy to use, they allow you to implement standard models. Model subclassing is the most complex

The code is prepared to be used in a computer terminal. The options of the algorithm to be passed to the terminal and their definitions are the following:

```
usage: DCEC.py
train

positional arguments:
  {mnist,usps,mnist-test}

optional arguments:
  -h, --help            show this help message and exit
  --n_clusters N_CLUSTERS
  --batch_size BATCH_SIZE
  --maxiter MAXITER
  --gamma GAMMA         coefficient of clustering loss
  --update_interval UPDATE_INTERVAL
  --tol TOL
  --cae_weights CAE_WEIGHTS       This argument must be given
  --save_dir SAVE_DIR
```

Listing 13.1: Options of the DCEC script

- **n_clusters**: As K-means and other clustering algorithms, you need to decide beforehand how many groups do you want to form.
- **Batch_size and Maxiter**: The first, defines the number of samples that will be propagated through the network (page 29). The second is the number of iterations that the network will perform during the training.
- **Gamma**: Is the weight of the clustering loss.

$$L = L_r + \gamma L_c$$

- **Update_interval and Tol**: The first, defines how many iterations are needed to update the target distribution P (page 53). Contrary to the soft assignment, the target distribution is not updated with every batch. *Tol* is a measure to decide when to stop the algorithm.
- **CAE_weights**: You can save the weights of a pretrained CAE to avoid doing the pretraining each time you want to use the DCEC.

However, since it is not common yet to have personal computers with GPUs adaptable to Deep Learning, the training process has to be done with CPUs and the neural network of the DCEC needs several (around 4) hours to train. So, I decided to adapt the code to the Jupyter notebooks used in Colab, to take advantage of its GPUs.

way of creating models because you have to implement all the parts of the neural network from scratch, though it allows much more customisation. A more exhaustive description can be found in the Keras documentation.

The steps I followed during the preparation of the code are the following:

- Prepare the code for Colab: Load the scripts as a modules.
- Update some functions, install libraries, check compatibility between systems etc.
- Add the Fashion dataset, which was not included in the original code, to evaluate the algorithm in that data.
- Tweak some of the parameters of the model such as:
  - γ (the weight of the clustering loss)
  - α (a parameter of the soft assignment)
- Try some autoencoders, different from the one that is coded in *ConvAE*, to see if the performance can be improved.

Next, I will show how the training was done and in the next section I will show the results using the clustering evaluation metrics.

13.1.1 DCEC training

Figure 13.1 illustrates the architecture of the DCEC algorithm using the CAE implemented in *ConvAE*. The convolutional autoencoder has already been presented in the section 11.1.2 (page 74), and the only difference between this CAE and the DCEC is the additional clustering layer. Now, from the embedding layer there are two ramifications: the decoder and the clustering layer.

To begin the training of the DCEC model, first, you have to pretrain a CAE. Then, the training starts as show in figure 13.2.

- In the left plot you can see the evolution of the three type of losses (global, clustering and autoencoder's reconstruction). The formula of the global loss is the following:

  $$L = L_r + 0.1L_c$$

  All the three losses start at 0 and increase slightly during the first thousand iterations. After the first thousand iterations you can see how the clustering loss decreases very rapidly while the reconstruction loss does not change much. This is because the autoencoder has already been (pre)trained during the pretraining phase. It seems that during the training of the DCEC, the clustering layer has much more effect on the parameters than the autoencoder.

- In the right plot it is represented the evolution of three metrics (ACC, ARI and NMI). What it is important is that all three metrics improve after the training of the DCEC, though the improvements are done in the first iterations.

Additionally, figure 13.3 shows -using UMAP to create a 2D visualisation-, how the 10-dimensional embedding space changes from the pretraining phase to the end of the DCEC training. It is clear that the clustering loss helps to separate the data into different groups. Unfortunately, these groups may not be correct: Some groups have mixed several classes while some classes are separated into several clusters.

The underlying assumption of DEC (and it also extends to the DCEC) is that the initial classifier's high confidence predictions are mostly correct [24]. You can think of it as a form of reinforcement learning. **Arguably, the most important step of the algorithm is the calculation of the centroids using k-means after the pretraining**. However, as the left plot of 13.3 shows, the embedded space of the pretrained CAE not necessarily produce adequate spaces for k-means clustering. In the previous sections, the output of CAEs was even worse for K-means, with elliptic shapes.

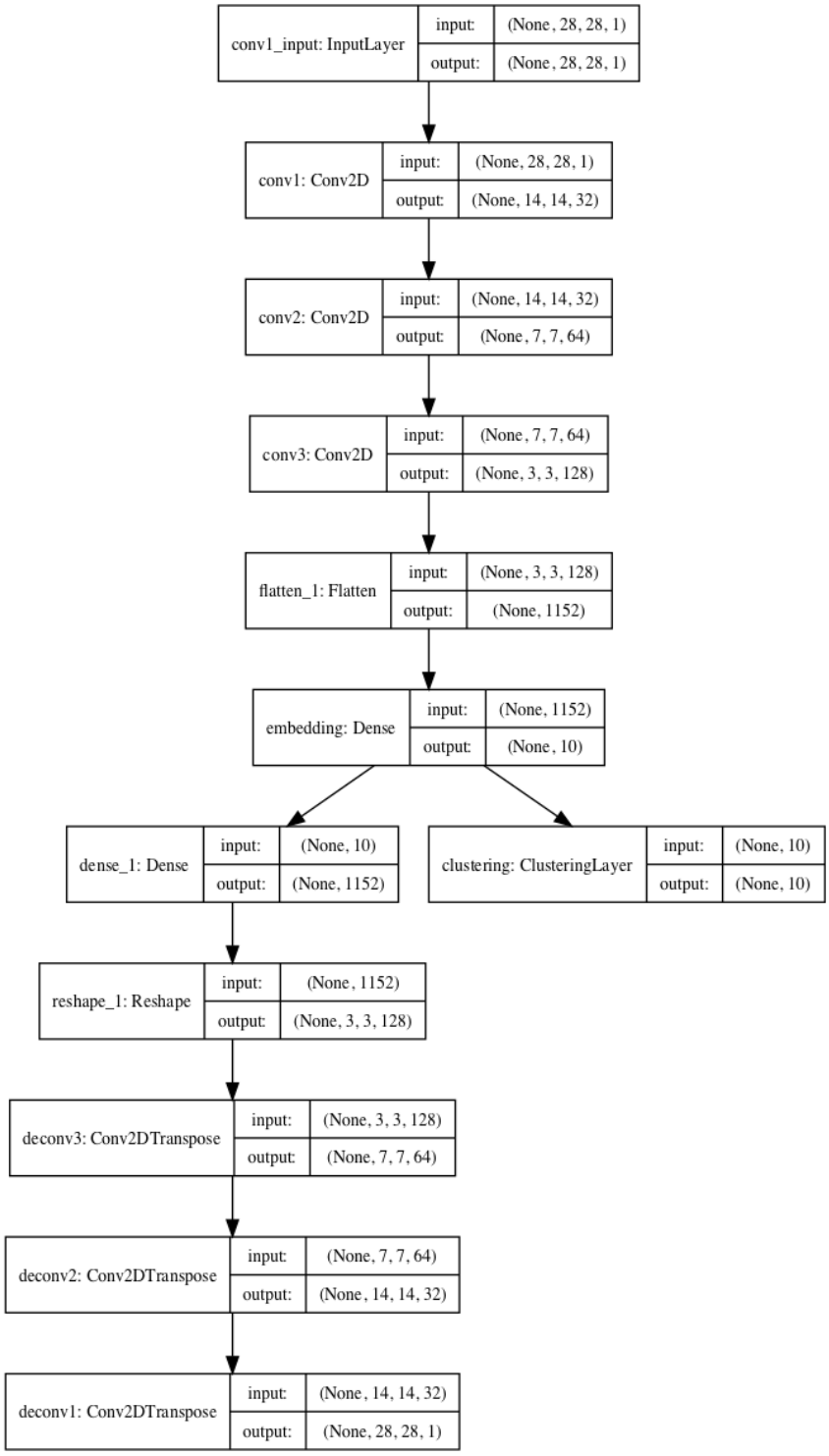


Figure 13.1: DCEC model from Guo et.al.

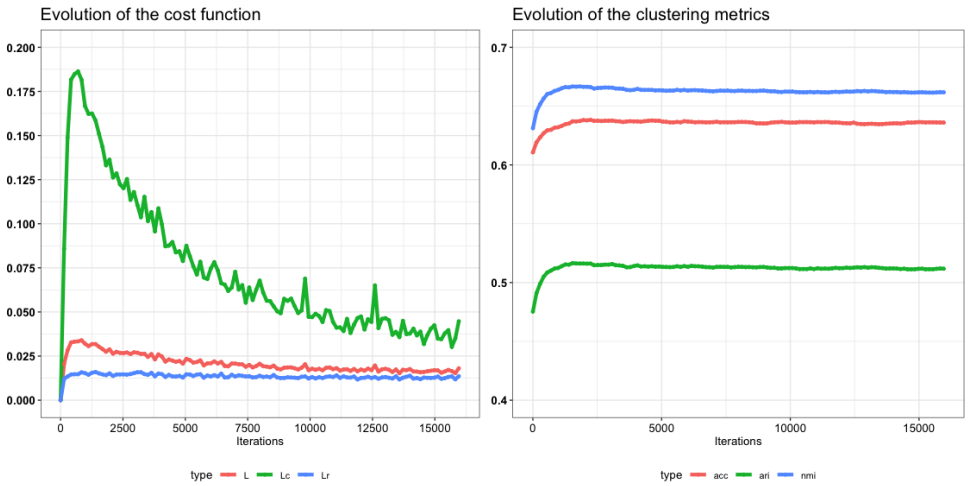


Figure 13.2: Training process DCEC model from Guo et.al.

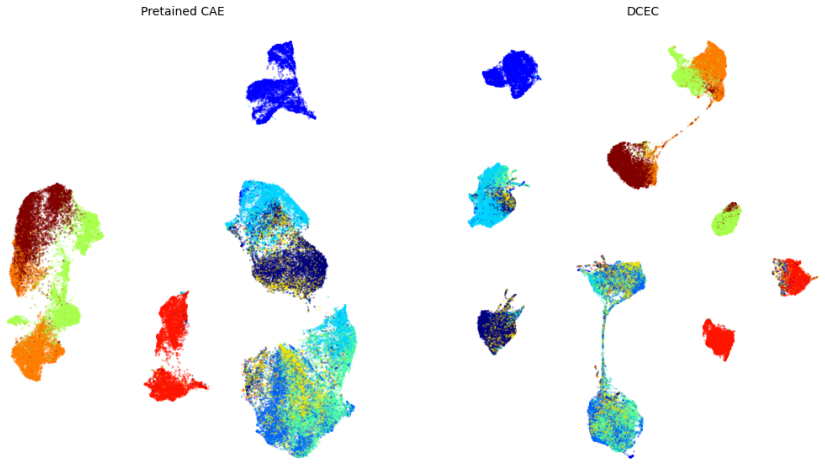


Figure 13.3: Visualisation of the embedding layer

13.1.2 Discussion of some improvements

Deep learning algorithms are improving constantly. This in part due to the great flexibility that neural networks have, which allows them to map very complex functions. That is why I consider that the DCEC has a lot of possibilities (and some problems too):

a) After the pretraining stage, the formation of high confidence centroids is a key step as has been explained. The DCEC uses k-means, which is a very fast algorithm, but there are other alternatives that in general perform better, such as gaussian mixture models. Since when using CAEs there is no way to know which properties will have the clusters in the embedded space, it is important to choose a clustering method that performs well in a wide range of circumstances.

b) Try several network architectures and compare them with the values of the loss functions. It is possible to use more sophisticated convolutional autoencoders.

c) The parameter γ determines the importance of the clustering loss, as have been seen. This parameter can be moved between 0 and see which fits best; it can have quite a lot of influence on the results.

d) Instead of doing the max of the soft assignments to assign a cluster, you can use them to see the confidence at each assignation. Moreover, I consider that in a purely unsupervised mode, it is better to cluster only high confidence samples and not touch the rest.

e) Change alpha, the parameter associated with the soft assignments. The DCEC uses $\alpha = 1$ as stated in 9.1.4 (page 52). Although it is not a very important parameter other values could be more adequate. As stated in [6], a value around three -higher than one in any case-, can be useful when the low dimensional space is not two dimensional but has ten dimensions.

f) Overfitting should be very seriously considered. Some deep clustering algorithms does not use validation sets and thus it is not possible to see if the algorithm generalises well to other similar samples.

   Overfitting is not only important in supervised learning, it is also key for autoencoders. If you want to compress the data, then overfitting does not matter. But if you want to be able to capture a meaningful structure of your data distribution, then it is important to check the validation. At the extreme end of overfitting, one could simply memorise the data and return a one-dimensional "representation" of it in the form of a ID, which tells you exactly nothing of value about the structure of the data.

g) It is important to be careful with the evaluation methods. It is not technically accurate to use classification datasets to see the clustering performance of an algorithm (page 40).

The output of some of these modifications can be found on Github.

13.2 Final comparison of results

Table 13.1: Deep clustering performance

	Fashion			MNIST		
Algorithm	NMI	ARI	ACC	NMI	ARI	ACC
K-means	0.5284	0.3844	0.5514	0.4998	0.3665	0.5346
UMAP2+K-means	0.6433	0.4800	0.5746	0.8313	0.7451	0.8007
CAE10 Guo+K-means (no test)	0.5983	0.4308	0.5534	0.7882	0.7442	0.8365
CAE10+K-means (no test)	0.5430	0.3826	0.5149	0.5918	0.4801	0.615
DCEC Guo ($\gamma = 1$)	0.6072	0.4471	0.5609	0.5195	0.3792	0.5417
DCEC Guo	**0.6616**	**0.5117**	**0.6360**	**0.8898**	**0.8552**	**0.8924**

In this final table there are the three clustering strategies studied during the project, and all use k-means to compare the results: **1)** traditional clustering, **2)** dimensionality reduction and clustering afterwards (two-stages) and **3)** deep clustering, which uses joint dimensionality reduction and clustering.

The DCEC with the CAE architecture as stated by Guo achieves the best performance in all the cases, in these specific datasets (at least). Moreover, the values obtained by the algorithm are similar in all the three metrics, which ensures more consistent results. Changing the value of γ to give more importance to the clustering loss has decreased slightly the performance but the DCEC still obtains good results. The clustering loss can distort the embedding space significantly if it has a high weight.

However, two-stages clustering, using for example UMAP, has values quite close to the DCEC. This method additionally gives insights about the data, since you can visualise their structure, and is computationally fast, so it is not a bad alternative.

Conclusions

In this project I have analysed the reasons why dimensionality reduction can help the clustering task. First, I have reviewed the theory of both fields and then I have perform some experiments using image data and state-of-the-art algorithms among which stand out deep clustering.

Traditional clustering, meaning, using a single algorithm on the raw data, has severe limitations. Data has noise and many algorithms does not perform well on very high dimensional datasets because of the curse of dimensionality. According to the results obtained, this is the strategy that offers the worst results.

In this sense, dimensionality reduction can help to boost clustering results, as they highly depend on the quality of the data representation. Additionally, DR is useful to get an intuition of the data and can help to understand complex algorithms such as neural networks. However, there is still much debate about which latent spaces are good for clustering. In the experiments though, I found that some techniques, especially from the branch of manifold learning, can drastically improve the clustering results in a process known as two-stages clustering.

Lastly, with the rising of Deep learning in recent years, a new clustering approach has appeared. Deep clustering can jointly learn good representations and clusters, so it can learn both task at the same time. Moreover, these algorithms can easily adapt to specific data types, such as images. In this project, I analysed the DCEC, a deep clustering algorithm that uses an autoencoder to reduce the dimension of the data. It is the algorithm that has shown the best results in the experiments. But as the old saying goes, all that glitters is not gold. From my point of view, the DCEC relays too much on the calculation of the initial centroids after the pretaining, which I think, can lead to very bad results if the data are complex. Moreover, the performance of autoencoders as a DR technique has not been as high as expected, since manifold learning methods seem better, at least for visualisation, and the results are quite variable and dependent on the architecture of the network.

Throughout the project I have realised that in unsupervised learning you have a lot of decisions to make: Apart from the intimidating amount of algorithms, both for DR and for clustering, you also have to choose their parameters and decide the number of cluster or dimensions of

the embedded space, among many other things. That is why, in this framework it is especially important to detail all the steps you have followed during the analysis in order to do truly reproducible research.

To end, in the future I think we can expect an increasingly similar performance between unsupervised and supervised methods for labelling processed datasets. This improvement in unsupervised learning will probably come from deep clustering algorithms since they still have a lot of margin to improve, especially regarding their theoretical foundations. Dimensionality reduction will remain being relevant because of the high dimensional data that are appearing continuously and its integration with other algorithms.

References

[1] Bradley Efron. "Prediction, Estimation, and Attribution". In: *Journal of the American Statistical Association* 115 (2020), pp. 636–655. URL: https://doi.org/10.1080/01621459.2020.1762613.

[2] van den Herik HJ van der Maaten LJP Postma EO. "Dimensionality reduction: a comparative review". In: *Technical report, Tilburg University* (2009). URL: https://lvdmaaten.github.io/publications/papers/TR_Dimensionality_Reduction_Review_2009.pdf.

[3] Vin de Silva and Joshua B. Tenenbaum. "Global versus Local Methods in Nonlinear Dimensionality Reduction". In: *Proceedings of the 15th International Conference on Neural Information Processing Systems*. NIPS'02. Cambridge, MA, USA: MIT Press, 2002, pp. 721–728. URL: https://web.mit.edu/cocosci/Papers/nips02-localglobal-in-press.pdf.

[4] Naomi Altman and Martin Krzywinski. "The curse(s) of dimensionality". In: *Nature Methods* 15.6 (2018), pp. 399–400. URL: https://doi.org/10.1038/s41592-018-0019-x.

[5] Michel Verleysen and Damien François. "The Curse of Dimensionality in Data Mining and Time Series Prediction". In: *Computational Intelligence and Bioinspired Systems*. Ed. by Joan Cabestany, Alberto Prieto, and Francisco Sandoval. Springer Berlin Heidelberg, 2005, pp. 758–770.

[6] L.J.P. van der Maaten. "Visualizing High-Dimensional Data Using t-SNE". In: *Journal of Machine Learning Research* (2008). URL: https://lvdmaaten.github.io/tsne/.

[7] George C.Linderman and Stefan Steinerberger. "Clustering with t-SNE, Provably". In: *SIAM Journal on Mathematics of Data Science* (2018). URL: https://epubs.siam.org/doi/abs/10.1137/18M1216134.

[8] Leland McInnes, John Healy, and James Melville. "UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction". In: (2018). arXiv: 1802.03426 [stat.ML]. URL: https://arxiv.org/abs/1802.03426.

[9] Nikolay Oskolkov. *tSNE vs. UMAP: Global Structure*. 2020. URL: https://towardsdatascience.com/tsne-vs-umap-global-structure-4d8045acba17.

[10] Leland McInnes. *Using UMAP for Clustering*. 2018. URL: https://umap-learn.readthedocs.io/en/latest/clustering.html.

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

[12] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: (2016). arXiv: 1609.04747 [cs.LG]. URL: https://arxiv.org/abs/1609.04747.

[13] Vincent Dumoulin and Francesco Visin. "A guide to convolution arithmetic for deep learning". In: (2016). arXiv: 1603.07285 [stat.ML]. URL: https://arxiv.org/abs/1603.07285.

[14] Stanford. *cs231n Convolutional Neural Networks for Visual Recognition*. URL: https://cs231n.github.io/convolutional-networks/.

[15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Neural Information Processing Systems* 25 (Jan. 2012). DOI: 10.1145/3065386. URL: https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[16] Bodo Rosenhahn Marco Rudolph Bastian Wandt. "Structuring Autoencoders". In: (2019). URL: https://arxiv.org/abs/1908.02626.

[17] Lilian Weng. "From Autoencoder to Beta-VAE". In: *lilianweng.github.io/lil-log* (2018). URL: http://lilianweng.github.io/lil-log/2018/08/12/from-autoencoder-to-beta-vae.html.

[18] L.J.P. van der Maaten. "Supplemental Material for Visualizing Data using t-SNE". In: *Journal of Machine Learning Research* (2008). URL: https://lvdmaaten.github.io/publications/misc/Supplement_JMLR_2008.pdf.

[19] Dongkuan Xu and Yingjie Tian. "A Comprehensive Survey of Clustering Algorithms". In: 2 (2015), pp. 165–193. URL: https://doi.org/10.1007/s40745-015-0040-1.

[20] Tiantian Zhang, Li Zhong, and Bo Yuan. "A Critical Note on the Evaluation of Clustering Algorithms". In: (2019). arXiv: 1908.03782 [cs.LG]. URL: https://arxiv.org/abs/1908.03782.

[21] Julio-Omar Palacio-Niño and Fernando Berzal. "Evaluation Metrics for Unsupervised Learning Algorithms". In: (2019). arXiv: 1905.05667 [cs.LG]. URL: https://arxiv.org/abs/1905.05667.

[22] Xifeng Guo et al. "Improved Deep Embedded Clustering with Local Structure Preservation". In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*. 2017, pp. 1753–1759. DOI: 10.24963/ijcai.2017/243. URL: https://doi.org/10.24963/ijcai.2017/243.

[23] Jianwei Yang, Devi Parikh, and Dhruv Batra. "Joint Unsupervised Learning of Deep Representations and Image Clusters". In: (2016). arXiv: 1604.03628. URL: https://arxiv.org/abs/1604.03628.

[24] Junyuan Xie, Ross Girshick, and Ali Farhadi. "Unsupervised Deep Embedding for Clustering Analysis". In: (2015). arXiv: 1511.06335. URL: https://arxiv.org/abs/1511.06335.

[25] Mathilde Caron et al. *Deep Clustering for Unsupervised Learning of Visual Features*. 2018. arXiv: 1807.05520. URL: https://arxiv.org/abs/1807.05520.

[26] Paras dahal. *Deep clustering*. 2019. URL: https://deepnotes.io/deep-clustering.

[27] Xifeng Guo et al. "Deep Clustering with Convolutional Autoencoders". In: *Neural Information Processing*. Ed. by Derong Liu et al. Springer International Publishing, 2017, pp. 373–382.

[28] Han Xiao, Kashif Rasul, and Roland Vollgraf. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. Aug. 28, 2017. arXiv: cs.LG/1708.07747 [cs.LG]. URL: https://arxiv.org/abs/1708.07747.

[29] Alex Krizhevsky. "Learning Multiple Layers of Features from Tiny Images". In: *AISTATS* (2009). URL: https://www.cs.toronto.edu/~kriz/cifar.html.

[30] Richard Kinh Rikiya Yamashita Mizuho Nishio and Kaori Togashi. "Convolutional neural networks: an overview and application in radiology". In: *Insights Imaging* (2018). DOI: https://doi.org/10.1007/s13244-018-0639-9.

[31] Adrian Rosebrock. *Keras vs. TensorFlow – Which one is better and which one should I learn?* 2018. URL: https://www.pyimagesearch.com/2018/10/08/keras-vs-tensorflow-which-one-is-better-and-which-one-should-i-learn/.

[32] Adam Pearce Andy Coenen. *Understanding UMAP*. URL: https://pair-code.github.io/understanding-umap/.

[33] Martin Wattenberg, Fernanda Viégas, and Ian Johnson. "How to Use t-SNE Effectively". In: *Distill* (2016). DOI: 10.23915/distill.00002. URL: http://distill.pub/2016/misread-tsne.

[34] Francois Chollet. *Building Autoencoders in Keras*. 2016. URL: https://blog.keras.io/building-autoencoders-in-keras.html.

[35] NikolayOskolkov. *tSNE vs. UMAP: Global Structure Preservation*. 2020. URL: https://github.com/NikolayOskolkov/tSNE_vs_UMAP_GlobalStructure/blob/master/tSNE_vs_UMAP.ipynb.

[36] Leland McInnes, John Healy, and Steve Astels. "hdbscan: Hierarchical density based clustering". In: *The Journal of Open Source Software* 2.11 (Mar. 2017). DOI: 10.21105/joss.00205. URL: https://doi.org/10.21105%2Fjoss.00205.

[37] Steve Astels Leland McInnes John Healy. *Basic Usage of HDBSCAN* for Clustering*. 2016. URL: https://hdbscan.readthedocs.io/en/latest/index.html.

[38] *Clustering on the output of t-SNE*. 2017. URL: https://stats.stackexchange.com/questions/263539/clustering-on-the-output-of-t-sne.

[39] Adrian Rosebrock. *Keras vs. tf.keras: What's the difference in TensorFlow 2.0?* 2019. URL: https://www.pyimagesearch.com/2019/10/21/keras-vs-tf-keras-whats-the-difference-in-tensorflow-2-0/.

[40] Christopher Olah. *Visualizing MNIST: An Exploration of Dimensionality Reduction*. 2014. URL: https://colah.github.io/posts/2014-10-Visualizing-MNIST/.

[41] Tensorflow. *Projector*. URL: https://projector.tensorflow.org/.

[42] Lan Huong Nguyen and Susan Holmes. "Ten quick tips for effective dimensionality reduction". In: *PLOS Computational Biology* 15.6 (June 2019), pp. 1–19. DOI: 10.1371/journal.pcbi.1006907. URL: https://doi.org/10.1371/journal.pcbi.1006907.

[43] E. Min et al. "A Survey of Clustering With Deep Learning: From the Perspective of Network Architecture". In: *IEEE Access* 6 (2018), pp. 39501–39514. URL: https://ieeexplore.ieee.org/document/8412085.

[44] S. Kaski and J. Peltonen. "Dimensionality Reduction for Data Visualization". In: *IEEE Signal Processing Magazine* 28.2 (2011), pp. 100–104.

Appendix

Next, in this appendix, I will show the code used for the visualisation and clustering in two stages of the Fashion dataset together with its output. The rest of the code, including the one used for deep clustering, can be found in the following repository in the form of Jupyter notebooks.

Contents

1 Download-process-display Data **2**

2 Exploration and visualisation with UMAP **5**

3 Clustering con KMEANS en los datos originales **8**
3.1 K-means en output de 30D de UMAP 9

4 Clustering con HDBSCAN en los datos originales **13**
4.1 Clustering HDBSCAN con UMAP 14

5 PCA section **20**

6 T-SNE section **23**

7 Autoencoders **27**

8 Clustering de la CAE 10 **32**

9 Autoencoder basado en el ejemplo de chollet **33**

10 Otra estructura de CAE **37**

11 CAE 2D para visualizacion **44**

12 Autoencoders + tsne30 **49**

Fashion

September 1, 2020

1 Download-process-display Data

[ ]:
```
%tensorflow_version 2.x
from tensorflow.keras.datasets import fashion_mnist
import numpy as np
import matplotlib.pyplot as plt

(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
# (num_samples, 28, 28).
# (num_samples,).


# (60000, 28, 28)
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.

# (60000, 784)
x_train_d = x_train.reshape(   (len(x_train), np.prod(x_train.shape[1:])  )  )␣
 ↪#60000, 28*28=784
x_test_d = x_test.reshape(     (len(x_test), np.prod(x_test.shape[1:])  )    )␣
 ↪#10000, 784

# (60000, 28, 28, 1)
x_train_c = np.reshape(x_train, (len(x_train), 28, 28, 1))  # 60000, 28, 28, 1
x_test_c = np.reshape(x_test, (len(x_test), 28, 28, 1))     # 10000, 28, 28, 1

n_train = 60000
n_test = 10000
n_full = 70000

x_full = np.concatenate( ( x_train_d, x_test_d ) , axis=0)  # (70000, 784)
y_full = np.concatenate( ( y_train, y_test ) , axis=0)  # (70000,)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-labels-idx1-ubyte.gz
32768/29515 [================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
```

```
datasets/train-images-idx3-ubyte.gz
26427392/26421880 [==============================] - 1s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-labels-idx1-ubyte.gz
8192/5148 [===============================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-images-idx3-ubyte.gz
4423680/4422102 [==============================] - 0s 0us/step
```

[ ]:
```
from keras.datasets import fashion_mnist
import numpy as np
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

x = np.concatenate((x_train, x_test))
y = np.concatenate((y_train, y_test))
x = x.reshape(-1, 28, 28, 1).astype('float32')
x = x/255.
print('fashion_mnist:', x.shape)
print('fashion_mnist:', y.shape)
```

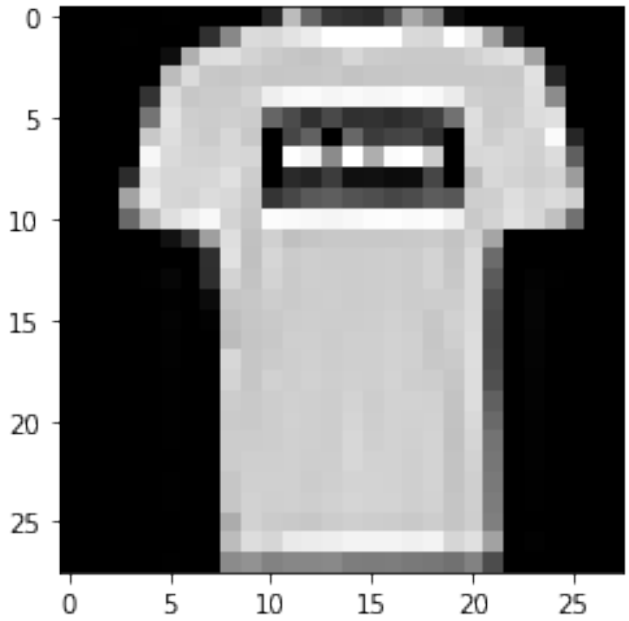
```
fashion_mnist: (70000, 28, 28, 1)
fashion_mnist: (70000,)
```

[ ]:
```
label_dictionnary = {0:'T-shirt/top', 1:'Trouser', 2:'Pullover',
                     3:'Dress', 4:'Coat', 5:'Sandal', 6:'Shirt',
                     7:'Sneaker', 8:'Bag', 9:'Ankle boot' }
noms = list( label_dictionnary.values() )
def true_label(x):
    return label_dictionnary[x]
```

[ ]:
```
import matplotlib.pyplot as plt

image = x_train[1,:,:]
print('this represent a : ' + true_label(y_train[1]))
plt.imshow(image, cmap='gray')
```

```
this represent a : T-shirt/top
```

[ ]:
```
<matplotlib.image.AxesImage at 0x7f09765b7470>
```



[ ]:
```
import numpy as np

semilla = 10
np.random.seed(seed= semilla)
muestra = np.random.randint(0, high= n_full, size=10000, dtype='int')

# https://docs.scipy.org/doc/numpy-1.15.0/reference/routines.random.html
```

[ ]:
```
#np.random.choice(data.ravel(),5,replace=False)
x_rfull = x_full[muestra]
y_rfull = y_full[muestra]

Target_name_train = np.vectorize(true_label)(y_train)
Target_name = np.vectorize(true_label)(y_full)
Target_namer = np.vectorize(true_label)(y_rfull)
Target_name_test = np.vectorize(true_label)(y_test)
```

Different samples of the dataset

[ ]:
```
fig, ax = plt.subplots(6, 10, figsize=(14, 11), facecolor= "azure")

for i, axi in enumerate(ax.flat):
    axi.imshow(x_rfull[i].reshape(28, 28), cmap='gray_r')
    axi.set(xticks=[], yticks=[])
    #axi.get_xaxis().set_visible(False)
    #axi.get_yaxis().set_visible(False)
```

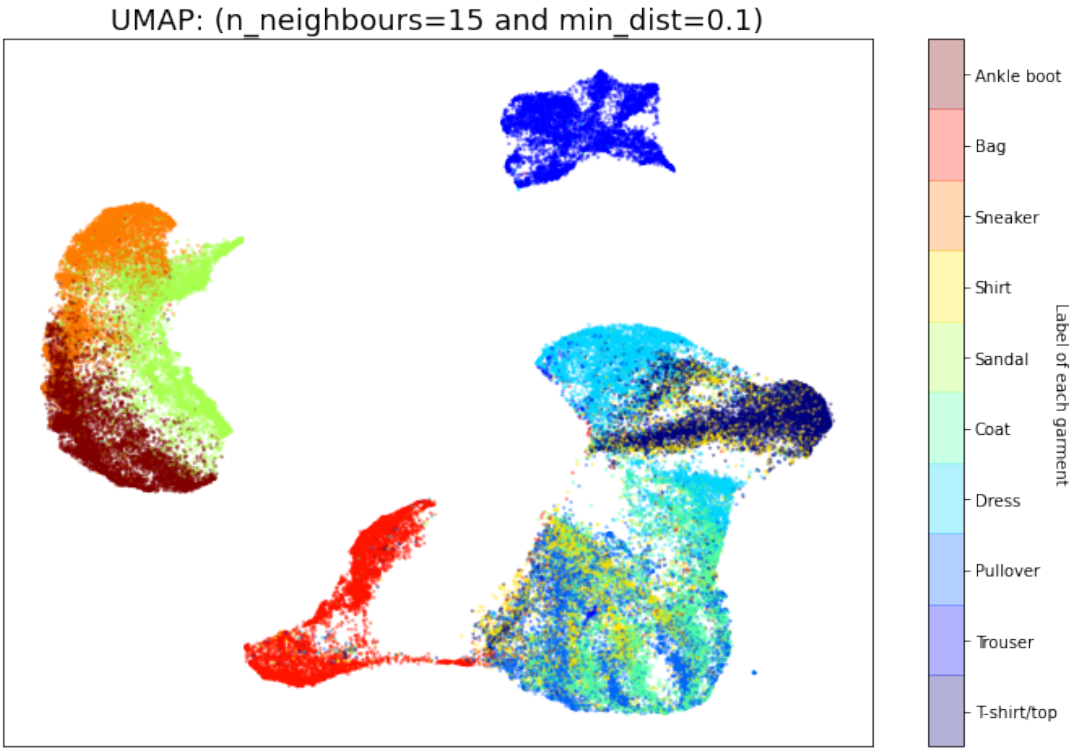
```
#fig.subplots_adjust(hspace=0, vspace=0)

#fig.savefig('subset_fashion.png')
```

2 Exploration and visualisation with UMAP

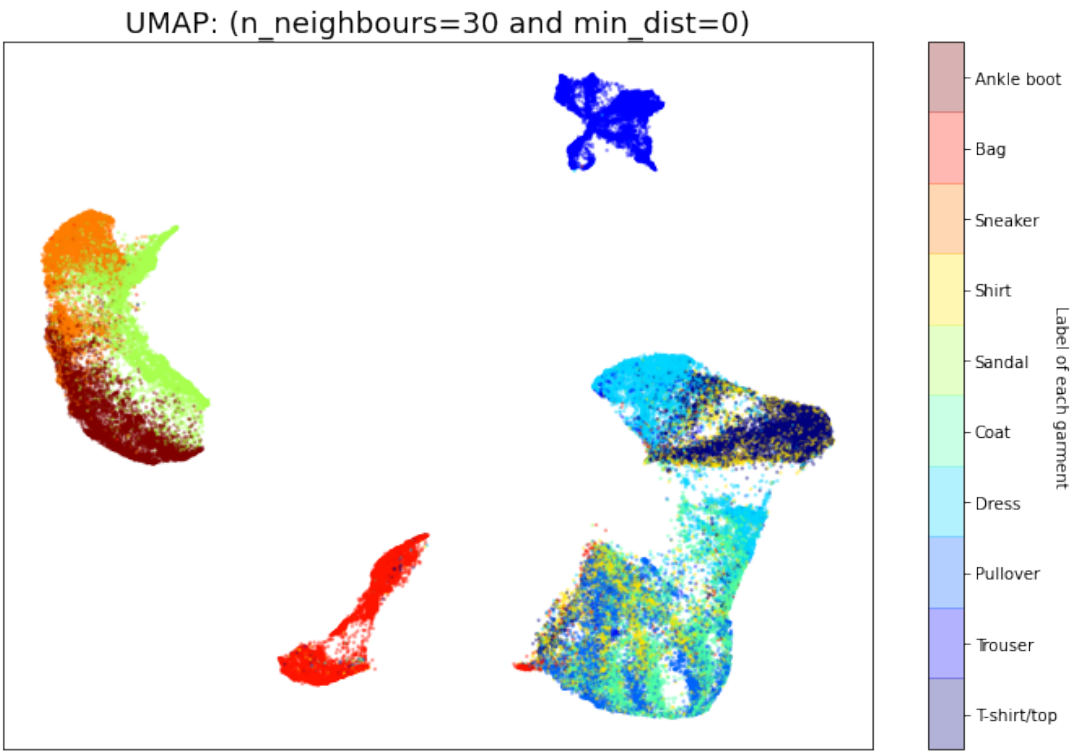
[ ]:
```
import umap

def draw_umap(n_neighbors=15, min_dist=0.1, n_components=2, metric='euclidean',␣
 ↪titol=''):
    fit = umap.UMAP(
        n_neighbors=n_neighbors,
        min_dist=min_dist,
        n_components=n_components,
        metric=metric,
        random_state=1997

    )
    u = fit.fit_transform(x_full);
```

```
    fig, ax = plt.subplots(figsize=(12, 8))
    plt.scatter( u[:,0], u[:,1], c= y_full, cmap=plt.cm.get_cmap('jet', 10),␣
↪s=2, alpha=0.3,label=Target_name)
    plt.setp(ax, xticks = [], yticks = [])
    ax.set_title('UMAP with true labels', fontsize=10)
    plt.title(titol, fontsize=18)

    cbar = plt.colorbar(boundaries = np.arange(11)-0.5, cmap=plt.cm.
↪get_cmap('jet', 10))
    cbar.set_ticks(np.arange(10))
    cbar.set_ticklabels(noms)
    cbar.ax.set_ylabel('Label of each garment', rotation=270, fontsize=10)

    #plt.savefig("UMAP_fashion")
    plt.show()
```

[ ]:
```
draw_umap(n_neighbors=15, min_dist=0.1, n_components=2, metric='euclidean',␣
↪titol='UMAP: (n_neighbours=15 and min_dist=0.1)')
```



[ ]:
```
draw_umap(n_neighbors=30, min_dist=0, n_components=2, metric='euclidean',␣
↪titol='UMAP: (n_neighbours=30 and min_dist=0)')
```



[ ]:
```
#from mpl_toolkits.mplot3d import Axes3D

#fit = umap.UMAP( n_neighbors= 25, min_dist= 0.1, n_components= 3,␣
 ↪random_state=1997, verbose=2  )
#u = fit.fit_transform(x_rfull)
fig = plt.figure(figsize=(14, 11))

ax = fig.add_subplot(111, projection='3d')
ax.scatter(u[:,0], u[:,1], u[:,2], c=y_rfull, cmap= plt.cm.get_cmap('jet', 10),␣
 ↪alpha=0.2, s=15) # s=100, plt.cm.get_cmap('cubehelix', 6), plt.cm.
 ↪get_cmap('jet', 10)
#plt.colorbar(ticks=range(10))
plt.title("UMAP with min_dist=0.1 and n_neighbours=25 ", fontsize=18)
```

[ ]:
```
fitting = umap.UMAP(n_neighbors=30,min_dist=0,n_components=2,random_state=1997)
umap_2d = fitting.fit_transform(x_full)
```

[ ]:
```
umap_2d.shape
```

[ ]:
```
(70000, 2)
```

[ ]:
```
umap_2d[:,1]
```

[ ]:
```
array([ 3.9076104 ,  4.246808  ,  4.0593863 , ...,  0.76575553,
        12.025316  ,  8.665088  ], dtype=float32)
```

[ ]:
```
np.where( umap_2d[:,1] < 1.5 )#, umap_2d[:,0] < 7)
```

[ ]:
```
(array([    5,     7,    18, ..., 69991, 69992, 69997]),)
```

Reducción con clustering
https://umap-learn.readthedocs.io/en/latest/clustering.html

HDBSCAN: Hierarchical Density-Based Spatial Clustering of Applications with Noise

[ ]:
```
# Dimension reduction and clustering libraries
!pip install hdbscan

import umap
import hdbscan
import sklearn.cluster as cluster
from sklearn.metrics import normalized_mutual_info_score, adjusted_rand_score,␣
 ↪adjusted_mutual_info_score, silhouette_score
```

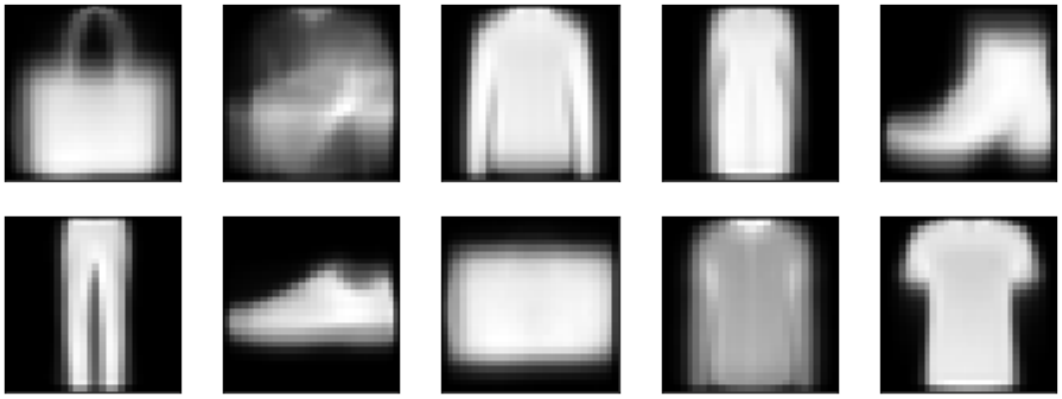
- **adjusted_rand_score**: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.adjusted_rand_
- **adjusted_mutual_info_score**: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.adjuste
  metrics-adjusted-mutual-info-score

3 Clustering con KMEANS en los datos originales

[ ]:
```
import time
time_start = time.time()

kmeans_raw = cluster.KMeans(n_clusters=10, n_init=10, random_state=1997,␣
 ↪init='k-means++', n_jobs=-1 ).fit(x_full) # x_rfull
kmeans_labels = kmeans_raw.labels_

print( 'k-means done! Time elapsed: {} seconds', time.time()-time_start )
```

```
k-means done! Time elapsed: {} seconds 72.0400116443634
```

[ ]:
```
fig, ax = plt.subplots(2, 5, figsize=(8, 3))
centers = kmeans_raw.cluster_centers_.reshape(10, 28, 28)
for axi, center in zip(ax.flat, centers):
    axi.set(xticks=[], yticks=[])
    axi.imshow(center, interpolation='nearest', cmap="gray") #plt.cm.binary

#plt.savefig('kmeans_centers.png')
```



3.1 K-means en output de 30D de UMAP

[ ]:
```
fit = umap.UMAP(
        n_neighbors=25,
        min_dist=0,
        n_components=2,
        metric="euclidean",
        random_state=1997 )
embedding = fit.fit_transform(x_full)
```

[ ]:
```
time_start = time.time()
fit_30 = umap.UMAP(
        n_neighbors=25,
        min_dist=0,
        n_components=30,
        metric="euclidean",
        random_state=1997 )
embedding_30 = fit_30.fit_transform(x_full)
print( 'UMAp done! Time elapsed: {} seconds', time.time()-time_start )
```

```
UMAp done! Time elapsed: {} seconds 332.29367232322693
```

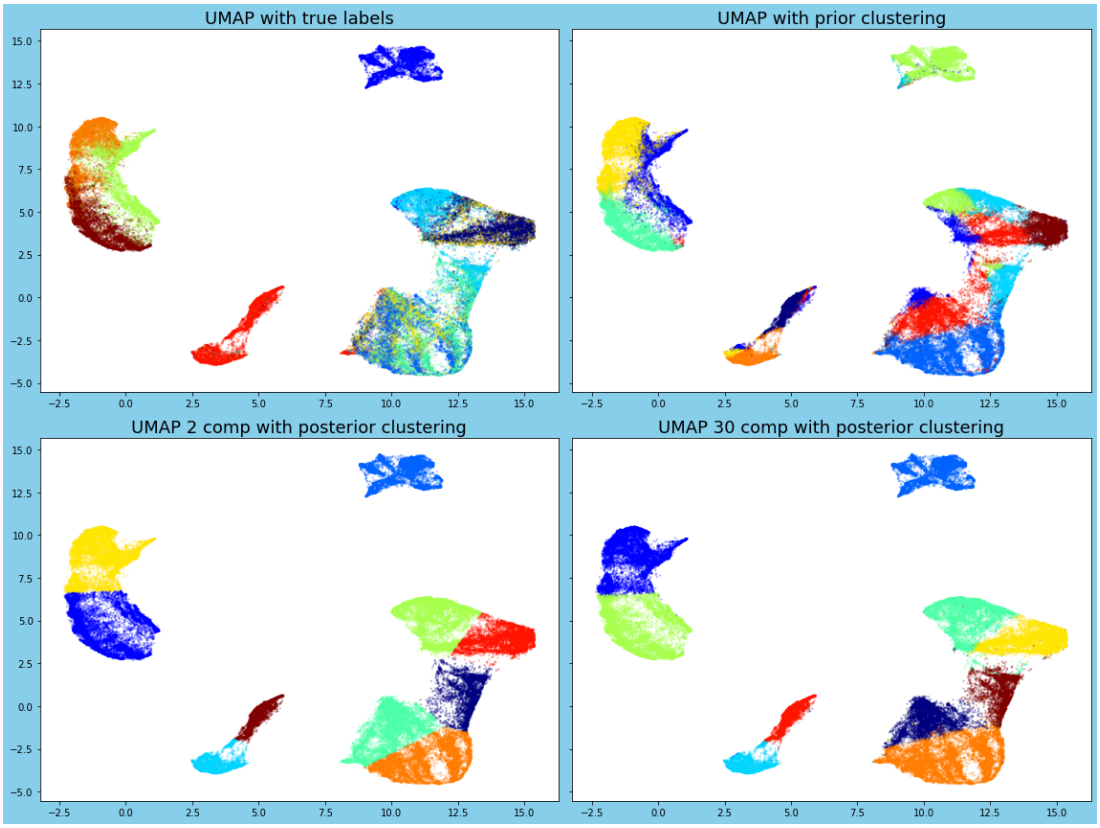
[ ]:
```
kmeans_labels_umap = cluster.KMeans(n_clusters=10, n_init=5, random_state=1997).
 ↪fit_predict( embedding )
kmeans_labels_umap_30 = cluster.KMeans(n_clusters=10, n_init=5,␣
 ↪random_state=1997).fit_predict( embedding_30 )
```

[ ]:
```
def grafico(space, titol='', fitxer=''):

  fig, ax = plt.subplots(figsize=(12, 8))
```

```
  plt.scatter( space[:,0], space[:,1], c= y_train, cmap=plt.cm.get_cmap('jet',␣
 ↪10), s=2 ,alpha=0.3,label=Target_name )
  plt.axis('off')


  ax.set_title( titol , fontsize=16)

  cbar = plt.colorbar(boundaries = np.arange(11)-0.5)
  cbar.set_ticks(np.arange(10))
  cbar.set_ticklabels(noms)
  #cbar.ax.set_ylabel('Label of each garment', rotation=270, fontsize=10)

  fig.savefig( fitxer )
  plt.show()
```

[ ]:
```
grafico(embedding, titol='UMAP: (n=30 dist=0)', fitxer='UMAP 2D')
```

[ ]:
```
#plt.style.use( "default" )

fig, axs = plt.subplots( nrows= 2, ncols= 2,sharey=True, facecolor=␣
 ↪"skyblue",figsize=(16, 12)) #gridspec_kw={'hspace': 0, 'wspace': 0}
(ax1, ax2) , (ax3, ax4) = axs

im = ax1.scatter( embedding[:,0], embedding[:,1], c=y_full, cmap=plt.cm.
 ↪get_cmap('jet', 10) , s= 2 , alpha=0.2)
ax1.set_title('UMAP with true labels', fontsize=18)
ax2.scatter( embedding[:,0], embedding[:,1], c=kmeans_labels, cmap=plt.cm.
 ↪get_cmap('jet', 10), s= 2, alpha=0.2)
ax2.set_title('UMAP with prior clustering',fontsize=18)
ax3.scatter( embedding[:,0], embedding[:,1], c=kmeans_labels_umap, cmap=plt.cm.
 ↪get_cmap('jet', 10), s= 2, alpha=0.2)
ax3.set_title('UMAP 2 comp with posterior clustering',fontsize=18)
ax4.scatter( embedding[:,0], embedding[:,1], c=kmeans_labels_umap_30, cmap=plt.
 ↪cm.get_cmap('jet', 10), s= 2, alpha=0.2)
ax4.set_title('UMAP 30 comp with posterior clustering',fontsize=18)

#cbar = fig.colorbar(im, ticks= range(10), use_gridspec=True,␣
 ↪orientation='vertical' )
#cbar.ax.set_ylabel('Label of each garment')
#cbar.ax.set_yticklabels( noms )

fig.tight_layout()
#plt.colorbar(ticks=range(10), label='Label of each garment' )
#plt.show()
plt.savefig('umap_kmeans.png')
```



[ ]:
```
#from google.colab import drive
#drive.mount('/content/drive')
```

[ ]:
```
#( adjusted_rand_score(y_full, kmeans_labels),␣
 ↪adjusted_mutual_info_score(y_full, kmeans_labels), )
print( 'NMI:', normalized_mutual_info_score(y_full, kmeans_labels) )
print( 'AMI:', adjusted_mutual_info_score(y_full, kmeans_labels) )
print( 'ARI:', adjusted_rand_score(y_full, kmeans_labels) )
print( 'SIL:', silhouette_score(x_full, kmeans_labels) )
```

```
NMI: 0.5284531404464996
AMI: 0.5283323673973551
ARI: 0.38445331366713387
SIL: 0.15170476
```

[ ]:
```
#( adjusted_rand_score(y_full, kmeans_labels_umap),␣
 ↪adjusted_mutual_info_score(y_full, kmeans_labels_umap) )
print( 'NMI:', normalized_mutual_info_score(y_full, kmeans_labels_umap) )
print( 'AMI:', adjusted_mutual_info_score(y_full, kmeans_labels_umap) )
print( 'ARI:', adjusted_rand_score(y_full, kmeans_labels_umap) )
print( 'SIL:', silhouette_score(x_full, kmeans_labels_umap) )
```

```
NMI: 0.643307307553911
AMI: 0.6432161848739049
ARI: 0.4800524268242437
SIL: 0.13201986
```

```
[ ]: #( adjusted_rand_score(y_full, kmeans_labels_umap_30),␣
      ↪adjusted_mutual_info_score(y_full, kmeans_labels_umap_30) )
     print( 'NMI:', normalized_mutual_info_score(y_full, kmeans_labels_umap_30) )
     print( 'AMI:', adjusted_mutual_info_score(y_full, kmeans_labels_umap_30) )
     print( 'ARI:', adjusted_rand_score(y_full, kmeans_labels_umap_30) )
     print( 'SIL:', silhouette_score(x_full, kmeans_labels_umap_30) )
```

```
NMI: 0.6373864820290899
AMI: 0.6372947474832065
ARI: 0.4780827003466693
SIL: 0.08861612
```

```
[ ]: !pip install coclust
     from coclust.evaluation.external import accuracy
```

```
[ ]: print( accuracy( y_full, kmeans_labels ) )
     print( accuracy( y_full, kmeans_labels_umap ) )
     print( accuracy( y_full, kmeans_labels_umap_30 ) )
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/utils/linear_assignment_.py:128:
FutureWarning: The linear_assignment function is deprecated in 0.21 and will be
removed from 0.23. Use scipy.optimize.linear_sum_assignment instead.
  FutureWarning)

0.5514714285714286
0.5746142857142857
0.5833428571428572
```

min_samples: The simplest intuition for what min_samples does is provide a measure of how conservative you want you clustering to be. The larger the value of min_samples you provide, the more conservative the clustering – more points will be declared as noise, and clusters will be restricted to progressively more dense areas. We can see this in practice by leaving the min_cluster_size at 60, but reducing min_samples to 1.

The primary parameter to effect the resulting clustering is **min_cluster_size**. Ideally this is a relatively intuitive parameter to select – set it to the smallest size grouping that you wish to consider a cluster. It can have slightly non-obvious effects however.

4 Clustering con HDBSCAN en los datos originales

[ ]:
```
from sklearn.decomposition import PCA
time_start = time.time()

lowd_mnist = PCA(n_components=50).fit_transform(x_full)
hdbscan_labels = hdbscan.HDBSCAN(min_samples=10, min_cluster_size=500).
 ↪fit(lowd_mnist) # cluster_selection_epsilon

print( 'hdbscan done! Time elapsed: {} seconds', time.time()-time_start )
```

```
hdbscan done! Time elapsed: {} seconds 104.71124744415283
```

[ ]:
```
hdbscan_labels.labels_
```

[ ]:
```
array([-1, -1, -1, ..., -1,  0, -1])
```

[ ]:
```
unique_elements, counts_elements = np.unique(hdbscan_labels.labels_,␣
 ↪return_counts=True)
print("Frequency of unique values of the said array:")
print(np.asarray((unique_elements, counts_elements)))
```

```
Frequency of unique values of the said array:
[[   -1     0     1     2]
 [45203  5530  8454 10813]]
```

[ ]:
```
clustered = hdbscan_labels.labels_ >= 0
( adjusted_rand_score( y_full[clustered], hdbscan_labels.labels_[clustered]  ),␣
 ↪normalized_mutual_info_score(y_full[clustered], hdbscan_labels.
 ↪labels_[clustered]) )
```

[ ]:
```
(0.4583162443836787, 0.6621158615684914)
```

[ ]:
```
print( accuracy(y_full[clustered], hdbscan_labels.labels_[clustered]) )
```

```
0.5275638182038149

/usr/local/lib/python3.6/dist-packages/sklearn/utils/linear_assignment_.py:128:
FutureWarning: The linear_assignment function is deprecated in 0.21 and will be
removed from 0.23. Use scipy.optimize.linear_sum_assignment instead.
  FutureWarning)
```

[ ]:
```
np.sum(clustered) / y_rfull.shape[0]
```

[ ]:
```
0.7413
```

4.1 Clustering HDBSCAN con UMAP

https://hdbscan.readthedocs.io/en/latest/

[ ]:
```
#time_start = time.time()

#clusterer = hdbscan.HDBSCAN(
#    min_samples=10,
#    min_cluster_size=500,
#).fit(x_full)

#print( 'hdbscan done! Time elapsed: {} seconds', time.time()-time_start )

# demasiado lento!
```

[ ]:
```
clusterer_umap_2 = hdbscan.HDBSCAN(
    min_samples=10,
    min_cluster_size=500,
).fit(embedding)
```

[ ]:
```
clusterer_umap_30 = hdbscan.HDBSCAN(
    min_samples=10,
    min_cluster_size=500,
).fit(embedding_30)
```

[ ]:
```
clusterer_umap_2.single_linkage_tree_.plot(cmap='viridis', colorbar=True)
```

[ ]:
```
clusterer_umap_2.condensed_tree_.plot()
```

[ ]:
```
unique_elements, counts_elements = np.unique(clusterer_umap_2.labels_,␣
 ↪return_counts=True)
print("Frequency of unique values of the said array:")
print(np.asarray((unique_elements, counts_elements)))
```

```
Frequency of unique values of the said array:
[[   -1     0     1     2     3     4     5]
 [   37 21008  6590  3575  3003 21642 14145]]
```

[ ]:
```
unique_elements, counts_elements = np.unique(clusterer_umap_30.labels_,␣
 ↪return_counts=True)
print("Frequency of unique values of the said array:")
print(np.asarray((unique_elements, counts_elements)))
```

```
Frequency of unique values of the said array:
[[   -1     0     1     2     3     4]
 [   45 21009  6593 35832  3597  2924]]
```

[ ]:
```
clusterer_umap_2.labels_.max()+2
```

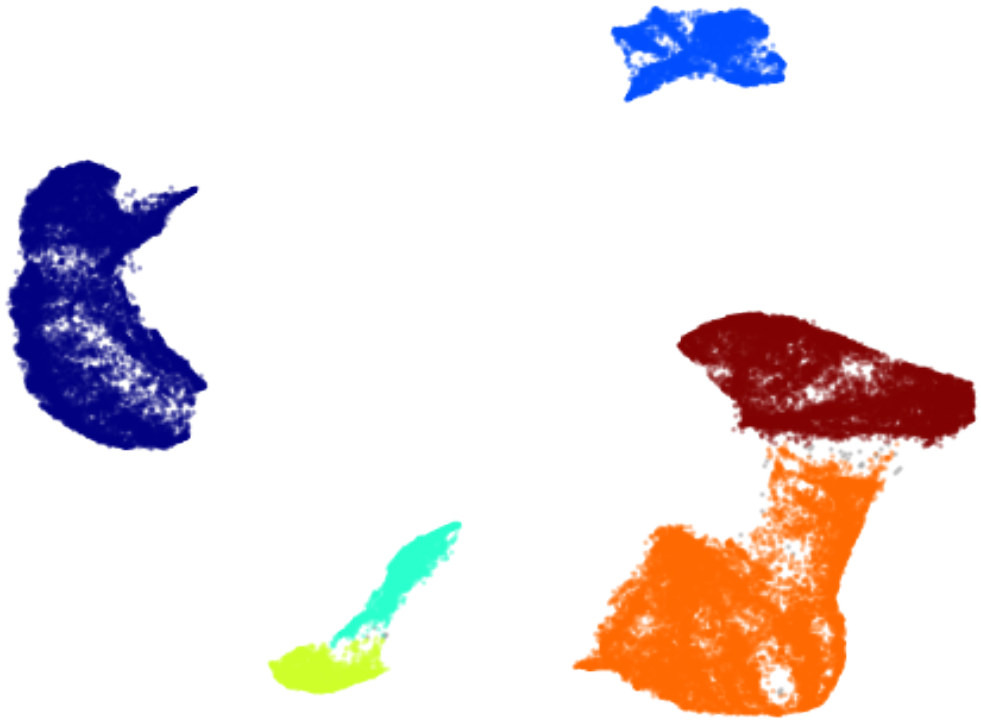
[ ]:
```
7
```

[ ]:
```
clusterer_umap_2.probabilities_
```

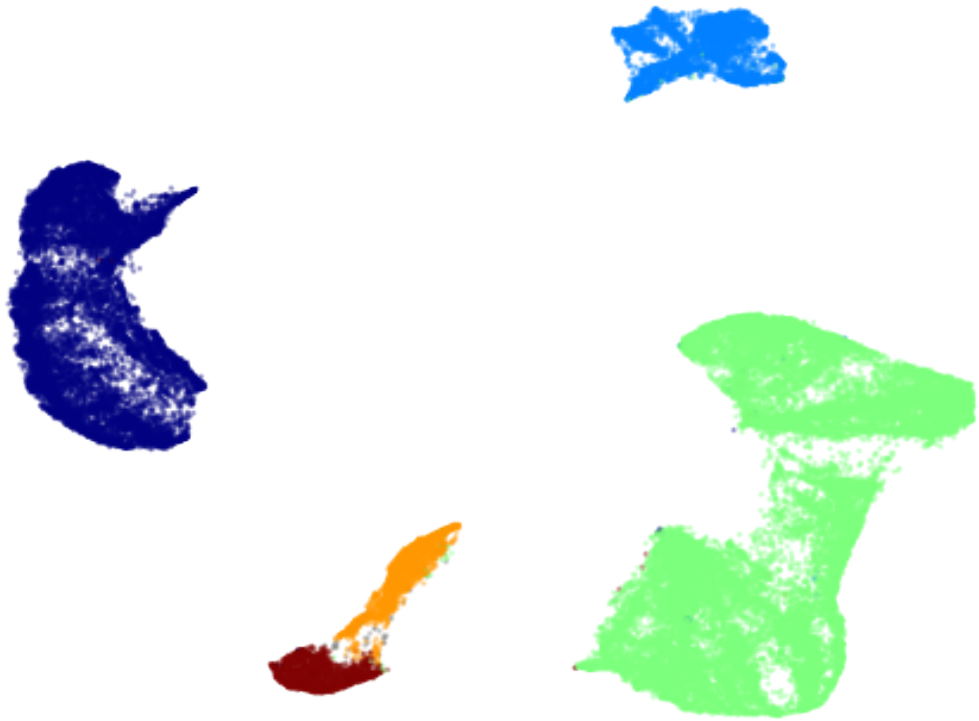
[ ]:
```
array([1.        , 1.        , 0.86546807, ..., 0.97382661, 0.72899473,
       1.        ])
```

[ ]:
```

```

[ ]:
```
fig, ax = plt.subplots(figsize=(8, 6))
clustered2 = clusterer_umap_2.labels_ >= 0

ax.scatter( embedding[ ~clustered2 ,0], embedding[ ~clustered2 ,1], c= "gray" ,␣
 ↪s=2 ,alpha=0.3)  #(0.5, 0.5, 0.5)

ax.scatter( embedding[ clustered2,0], embedding[clustered2,1], c=␣
 ↪clusterer_umap_2.labels_[clustered2] ,
                cmap=plt.cm.get_cmap('jet', clusterer_umap_2.labels_.max()+1),␣
 ↪s=2 ,alpha=0.3)
plt.axis('off')

plt.savefig('umap2_hdbscan.png')
plt.show()
```



[ ]:
```
fig, ax = plt.subplots(figsize=(8, 6))
clustered30 = clusterer_umap_30.labels_ >= 0

ax.scatter( embedding[ ~clustered30 ,0], embedding[ ~clustered30 ,1], c= "gray"␣
 ↪, s=2 ,alpha=0.3)  #(0.5, 0.5, 0.5)

ax.scatter( embedding[ clustered30,0], embedding[clustered30,1], c=␣
 ↪clusterer_umap_30.labels_[clustered30] ,
                cmap=plt.cm.get_cmap('jet', clusterer_umap_30.labels_.max()+1),␣
 ↪s=2 ,alpha=0.3)
plt.axis('off')

plt.savefig('umap30_hdbscan.png')
plt.show()
```



Medidas de evaluacion

[ ]:
```
#from sklearn.metrics import normalized_mutual_info_score,␣
 ↪adjusted_mutual_info_score, adjusted_rand_score, silhouette_score
```

```
print( '2 dimensions' )
print( 'NMI:', normalized_mutual_info_score(y_full, clusterer_umap_2.labels_) )
print( 'AMI:', adjusted_mutual_info_score(y_full, clusterer_umap_2.labels_) )
print( 'ARI:', adjusted_rand_score(y_full, clusterer_umap_2.labels_) )
#print( 'SIL:', silhouette_score(y_rfull, clusterer_umap_2.labels_)

print( '30 dimensions' )
print( 'NMI:', normalized_mutual_info_score(y_full, clusterer_umap_30.labels_) )
print( 'AMI:', adjusted_mutual_info_score(y_full, clusterer_umap_30.labels_) )
print( 'ARI:', adjusted_rand_score(y_full, clusterer_umap_30.labels_) )

# Sin contar las observaciones que no tienen grupo
print( 'No grey observations' )
print( 'NMI:', normalized_mutual_info_score(y_full[clustered2], clusterer_umap_2.
 ↪labels_[clustered2]) )
print( 'NMI:', normalized_mutual_info_score(y_full[clustered30],␣
 ↪clusterer_umap_30.labels_[clustered30]) )
print( 'ARI:', adjusted_rand_score(y_full[clustered2], clusterer_umap_2.
 ↪labels_[clustered2]) )
print( 'ARI:', adjusted_rand_score(y_full[clustered30], clusterer_umap_30.
 ↪labels_[clustered30]) )
```

```
2 dimensions
NMI: 0.6415125297712209
AMI: 0.6414400004120033
ARI: 0.3968751755460086
30 dimensions
NMI: 0.6059112692095248
AMI: 0.6058384105057631
ARI: 0.28386014346282984
No grey observations
NMI: 0.6422358004158011
NMI: 0.6064422630670114
ARI: 0.397018844708548
ARI: 0.2839515156672588
```

Accuracy score

[ ]:
```
#!pip install coclust
#from coclust.evaluation.external import accuracy
#coclust.evaluation.external.accuracy(true_row_labels, predicted_row_labels)
print( accuracy(y_full[clustered2], clusterer_umap_2.labels_[clustered2]) ) #0.
 ↪5346 -Z 53%
print( accuracy(y_full[clustered30], clusterer_umap_30.labels_[clustered30]) )
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/utils/linear_assignment_.py:128:
FutureWarning: The linear_assignment function is deprecated in 0.21 and will be
removed from 0.23. Use scipy.optimize.linear_sum_assignment instead.
```

```
  FutureWarning)
```

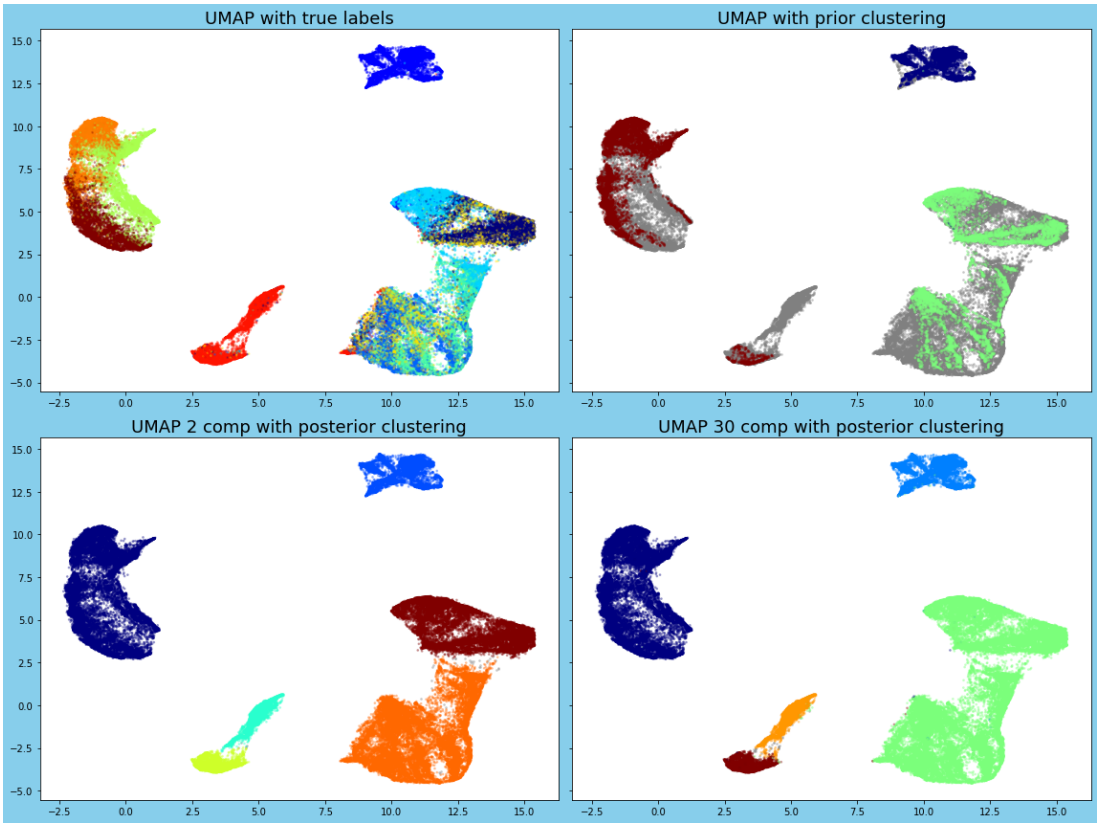
```
0.4361876992124409
0.3450360946322636
```

[ ]:
```
np.sum(clustered) / y_full.shape[0]
```

[ ]:
```
0.9906
```

[ ]:
```
fig, axs = plt.subplots( nrows= 2, ncols= 2,sharey=True, facecolor=
 ↪"skyblue",figsize=(16, 12)) #gridspec_kw={'hspace': 0, 'wspace': 0}
(ax1, ax2) , (ax3, ax4) = axs

ax1.scatter( embedding[:,0], embedding[:,1], c=y_full, cmap=plt.cm.
 ↪get_cmap('jet', 10) , s= 3 , alpha=0.3)
ax1.set_title('UMAP with true labels', fontsize=18)

clustered = hdbscan_labels.labels_ >= 0
ax2.scatter( embedding[ ~clustered ,0], embedding[ ~clustered ,1], c= "gray" ,
 ↪s=3 ,alpha=0.3)
ax2.scatter( embedding[ clustered,0], embedding[clustered,1], c= hdbscan_labels.
 ↪labels_[clustered] ,
                cmap=plt.cm.get_cmap('jet', hdbscan_labels.labels_.max()+1), s=3
 ↪,alpha=0.3)
ax2.set_title('UMAP with prior clustering',fontsize=18)

#clustered3 = clusterer_umap_2.labels_ >= 0
ax3.scatter( embedding[ ~clustered2 ,0], embedding[ ~clustered2 ,1], c= "gray" ,
 ↪s=3 ,alpha=0.3)
ax3.scatter( embedding[ clustered2,0], embedding[clustered2,1], c=
 ↪clusterer_umap_2.labels_[clustered2] ,
                cmap=plt.cm.get_cmap('jet', clusterer_umap_2.labels_.max()+1),
 ↪s=3 ,alpha=0.3)
ax3.set_title('UMAP 2 comp with posterior clustering',fontsize=18)

#clustered4 = clusterer_umap_30.labels_ >= 0
ax4.scatter( embedding[ ~clustered30 ,0], embedding[ ~clustered30 ,1], c= "gray"
 ↪, s=3 ,alpha=0.3)
ax4.scatter( embedding[ clustered30,0], embedding[clustered30,1], c=
 ↪clusterer_umap_30.labels_[clustered30] ,
#                cmap=plt.cm.get_cmap('jet', clusterer_umap_2.labels_.max()+1),
 ↪s=3 ,alpha=0.3)
#ax4.scatter( embedding[ :,0], embedding[:,1], c= clusterer_umap_30.labels_,
                cmap=plt.cm.get_cmap('jet', clusterer_umap_30.labels_.max()+1),
 ↪s=3 ,alpha=0.3)
ax4.set_title('UMAP 30 comp with posterior clustering',fontsize=18)
```

```
#cbar = fig.colorbar(im, ticks= range(10), use_gridspec=True,␣
 ↪orientation='vertical' )
#cbar.ax.set_ylabel('Label of each garment')
#cbar.ax.set_yticklabels( noms )

fig.tight_layout()
#plt.colorbar(ticks=range(10), label='Label of each garment' )
plt.savefig('umap_hdbscan.png')
plt.show()

#plt.savefig('cifar10_dog.png')
```



[ ]:
```
clustered = clusterer_umap_2.labels_ >= 0

with np.printoptions(threshold= np.inf):
    print(embedding[clustered])
```

5 PCA section

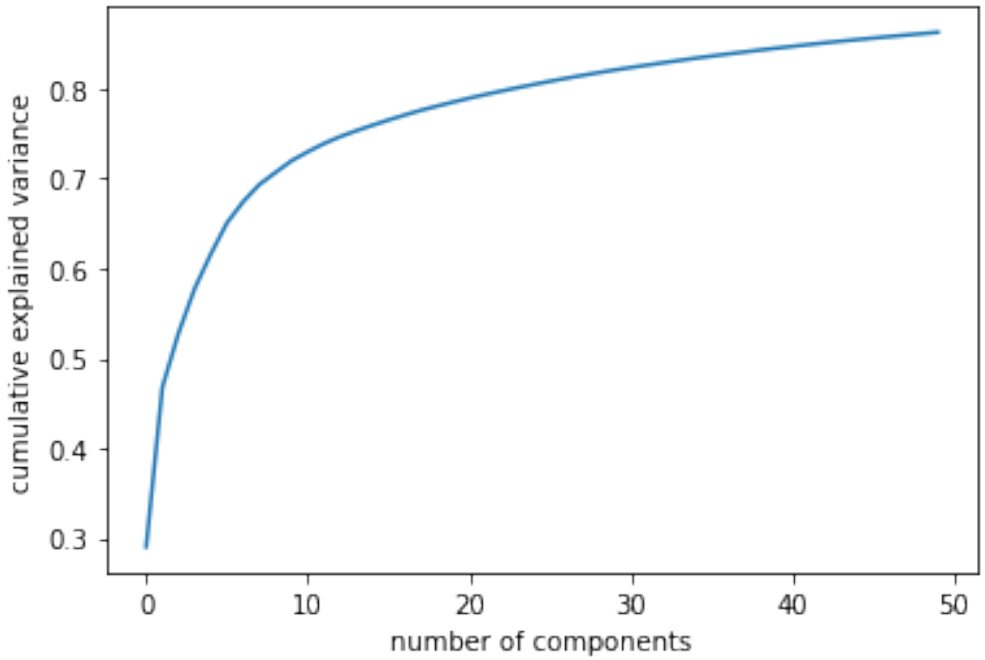
[ ]:
```
from sklearn.decomposition import PCA
from sklearn.preprocessing import scale
#https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

#x_train_d_scaled = scale(x_train_d, axis=0)
# falta escalar los datos

pca_2d = PCA(n_components=2)
pcax_2d = pca_2d.fit_transform(x_full)
```

[ ]:
```
print( pca_2d.explained_variance_ratio_)
print( x_full.shape )
print( pcax_2d.shape )
```

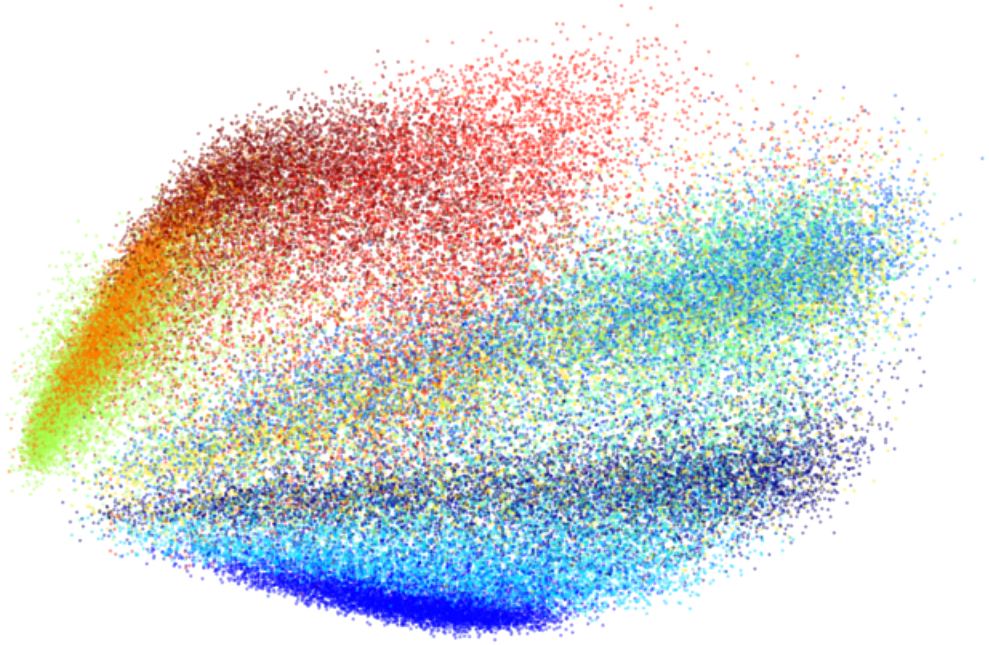
```
[0.29056854 0.177387  ]
(70000, 784)
(70000, 2)
```

[ ]:
```
from sklearn.decomposition import PCA
from sklearn.preprocessing import scale
pca_50d = PCA(n_components=50)
pcax_50d = pca_50d.fit_transform(x_full)
```

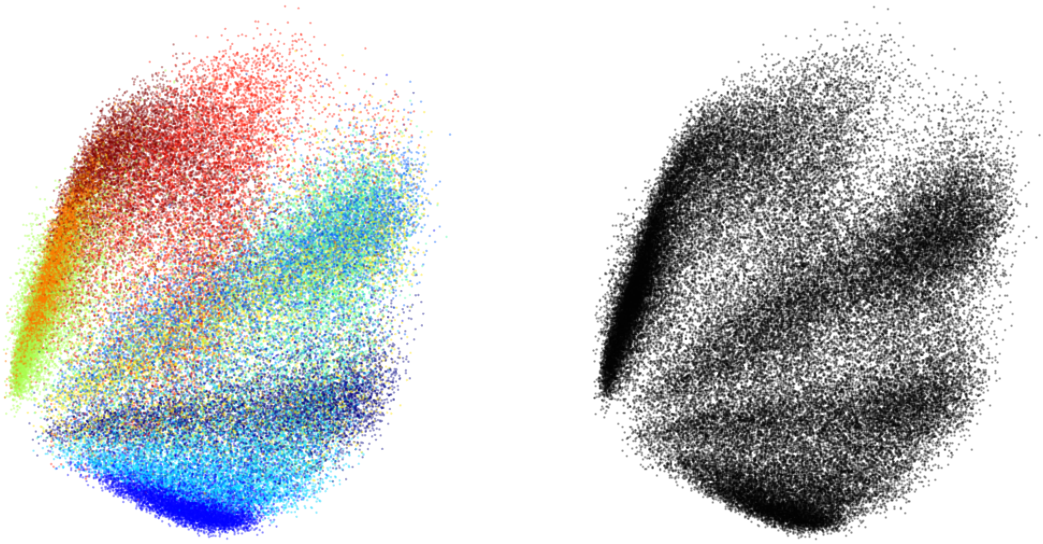
[ ]:
```
plt.plot(np.cumsum(pca_50d.explained_variance_ratio_))
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance')

plt.savefig("Scree_plot")
plt.show()
```



[ ]:
```
fig, ax = plt.subplots(figsize=(12, 8))
plt.scatter( pcax_2d[:,0], pcax_2d[:,1], c= y_full, cmap=plt.cm.get_cmap('jet',␣
 ↪10), s=2 ,alpha=0.3,label=Target_name )
plt.axis('off')
#ax.set_title( 'PCA_fashion' , fontsize=16)
fig.savefig( 'PCA_fashion' )
plt.show()
```



[ ]:
```
#fig, ((ax1, ax2), (ax3, ax4)) = plt.subplot(2, 2)
fig, (ax1, ax2) = plt.subplots( nrows=1,ncols=2,figsize=(18, 10) )

ax1.scatter( pcax_2d[:,0], pcax_2d[:,1], c= y_full, cmap=plt.cm.get_cmap('jet',␣
↪10), s=2 ,alpha=0.3,label=Target_name )
#ax1.set_title('PCA with true labels', fontsize=14)
ax1.axis('off')


ax2.scatter( pcax_2d[:,0], pcax_2d[:,1],  c= "black", s=2 ,alpha=0.
↪3,label=Target_name )
ax2.axis('off')
#ax2.set_title('PCA without labels', fontsize=14)


#plt.title("titol", fontsize=18)
#plt.legend(Target_name, y_rfull)
fig.savefig( 'PCA_fashion' )
plt.show()

#plt.savefig('cifar10_dog.png')
```



6 T-SNE section

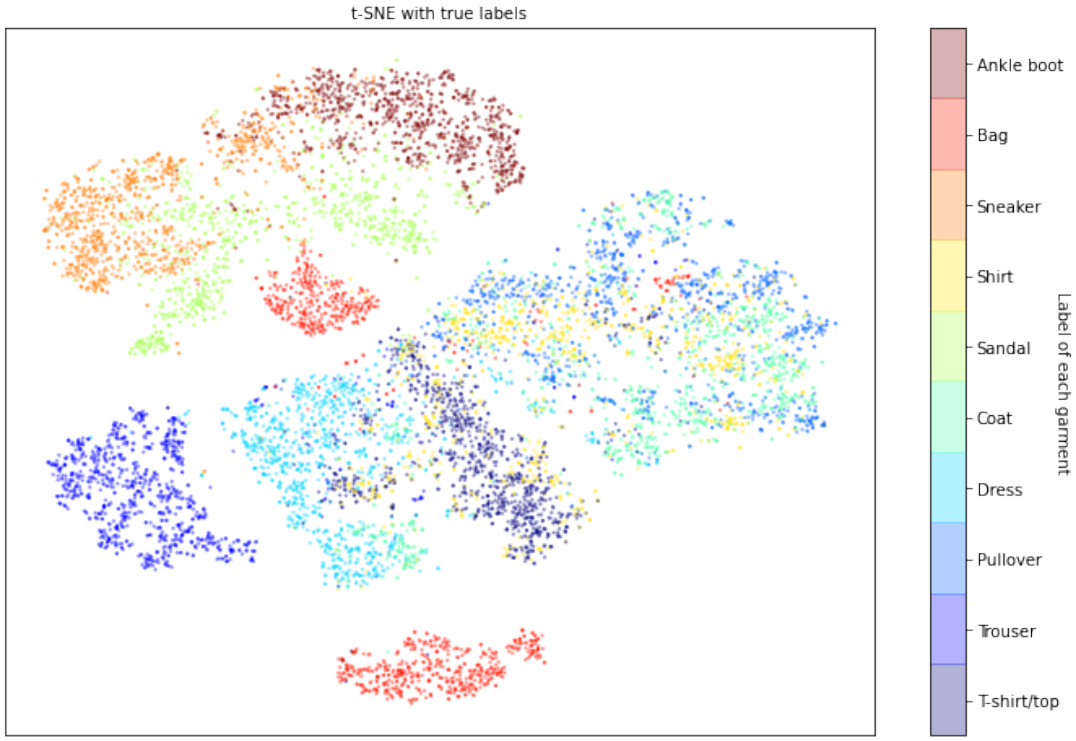
[ ]:
```
from sklearn.manifold import TSNE

tsne1 = TSNE( n_components= 2, perplexity= 25.0, learning_rate= 10.0,␣
 ↪n_iter=5000, random_state= semilla, n_iter_without_progress=200 )
```

[ ]:
```
#import time
time_start = time.time()

tsne_results = tsne1.fit_transform(x_rfull)

print( 't-SNE done! Time elapsed: {} seconds', time.time()-time_start )
```

```
t-SNE done! Time elapsed: {} seconds 600.4053189754486
```

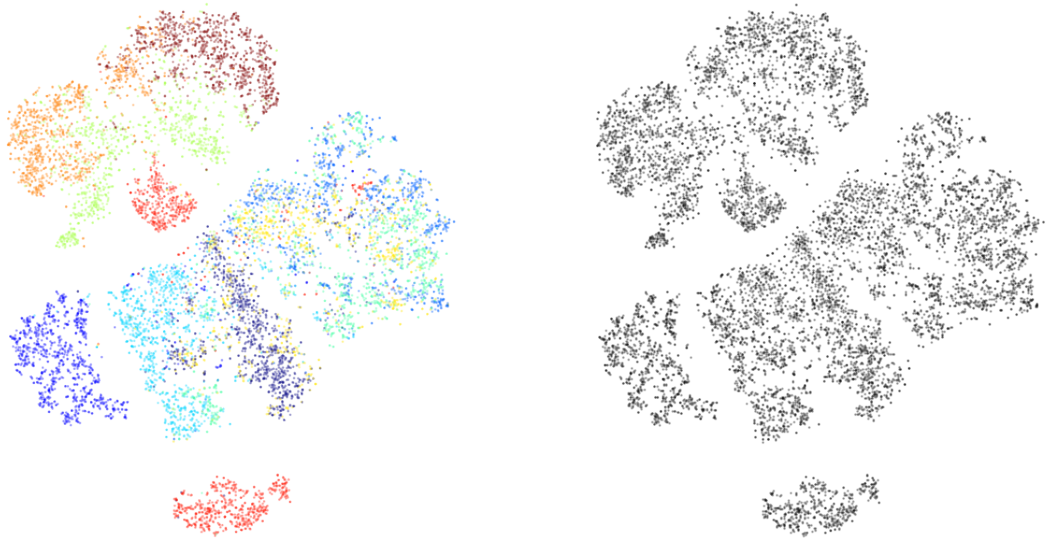
[ ]:
```
fig, ax = plt.subplots(figsize=(12, 8))
#fig = plt.figure()
#ax = fig.add_subplot(111)
#fig, [[ax1, ax2], [ax3, ax4]] = plt.subplots(nrows=2, ncols=2)
plt.scatter( tsne_results[:,0], tsne_results[:,1], c= y_rfull, cmap=plt.cm.
 ↪get_cmap('jet', 10), s=2 ,alpha=0.3,label=Target_namer )
plt.setp(ax, xticks = [], yticks = [])
```

```
ax.set_title('t-SNE with true labels', fontsize=10)

#plt.figure(figsize=(12, 10))
cbar = plt.colorbar(boundaries = np.arange(11)-0.5)
cbar.set_ticks(np.arange(10))
cbar.set_ticklabels(noms)
cbar.ax.set_ylabel('Label of each garment', rotation=270, fontsize=10)

#plt.title("titol", fontsize=18)
#plt.legend(Target_name, y_rfull)
plt.show()
```

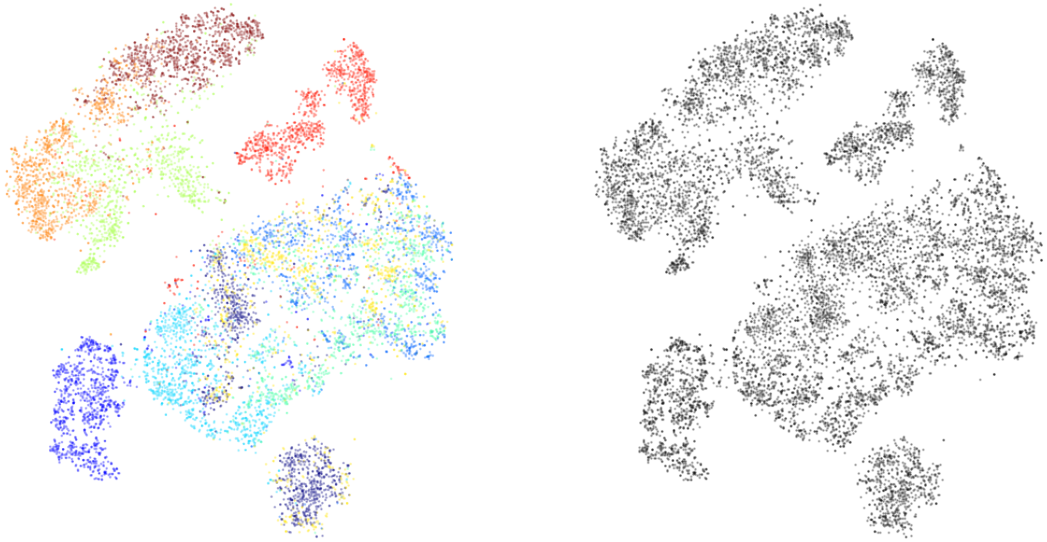


[ ]:
```
fig, (ax1, ax2) = plt.subplots( nrows=1,ncols=2,figsize=(18, 10) )

ax1.scatter( tsne_results[:,0], tsne_results[:,1], c= y_rfull, cmap=plt.cm.
 ↪get_cmap('jet', 10), s=2 ,alpha=0.3, label=Target_namer )
ax1.axis('off')

ax2.scatter( tsne_results[:,0], tsne_results[:,1],  c= "black", s=2 ,alpha=0.
 ↪3,label=Target_name )
ax2.axis('off')
```

```
fig.savefig( 'tsne_fashion' )
plt.show()
```



[ ]:
```
tsne2 = TSNE( n_components= 2, perplexity= 50.0, learning_rate= 10.0,␣
 ↪n_iter=5000, random_state= semilla, n_iter_without_progress=200 )
tsne_results = tsne2.fit_transform(x_rfull)

fig, (ax1, ax2) = plt.subplots( nrows=1,ncols=2,figsize=(18, 10) )

ax1.scatter( tsne_results[:,0], tsne_results[:,1], c= y_rfull, cmap=plt.cm.
 ↪get_cmap('jet', 10), s=2 ,alpha=0.3, label=Target_namer )
ax1.axis('off')

ax2.scatter( tsne_results[:,0], tsne_results[:,1],  c= "black", s=2 ,alpha=0.
 ↪3,label=Target_name )
ax2.axis('off')

fig.savefig( 'tsne_fashion' )
plt.show()
```



tsne + kmeans

[ ]:
```
kmeans_raw_tsne = cluster.KMeans(n_clusters=10, n_init=10, random_state=1997,␣
↪init='k-means++', n_jobs=-1 ).fit(tsne_results) # x_rfull
kmeans_tsne = kmeans_raw_tsne.labels_
```

[ ]:
```
print( '2 dimensions' )
print( 'NMI:', normalized_mutual_info_score(y_rfull, kmeans_tsne) )
print( 'AMI:', adjusted_mutual_info_score(y_rfull, kmeans_tsne) )
print( 'ARI:', adjusted_rand_score(y_rfull, kmeans_tsne) )
print( 'ACC:', accuracy(y_rfull, kmeans_tsne) )
#print( 'SIL:', silhouette_score(y_rfull, kmeans_tsne)  )
```

```
2 dimensions
NMI: 0.5842103860615369
AMI: 0.5834727539600013
ARI: 0.45028305835382515
ACC: 0.6163
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/utils/linear_assignment_.py:128:
FutureWarning: The linear_assignment function is deprecated in 0.21 and will be
removed from 0.23. Use scipy.optimize.linear_sum_assignment instead.
  FutureWarning)
```

7 Autoencoders

[ ]:
```
from tensorflow.keras.layers import Input, Dense, Conv2D, MaxPooling2D,␣
↪UpSampling2D, Conv2DTranspose, BatchNormalization, LeakyReLU, Flatten, Reshape
from tensorflow.keras.models import Model
from tensorflow.keras import backend
from keras import regularizers

import tensorflow as tf
```

[ ]:
```
input_img = Input(shape=(28, 28, 1), name='input')  # adapt this if using␣
↪`channels_first` image data format

x = Conv2D(24, (3, 3), padding='same')(input_img)
x = LeakyReLU(alpha=0.2)(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(20, (3, 3), padding='same')(x)
x = LeakyReLU(alpha=0.2)(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(16, (3, 3), padding='same')(x)
x = LeakyReLU(alpha=0.2)(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(12, (3, 3), padding='same')(x)
x = LeakyReLU(alpha=0.2)(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x =Conv2D(10, (3, 3), padding='same')(x)
x = LeakyReLU(alpha=0.2)(x)
encoded = MaxPooling2D((2, 2), padding='same',name='embedding')(x)
#x = Flatten(name='embedding')(x)

###

#x = Reshape( (3,3,128) )(encoded)
x = UpSampling2D((2, 2))(encoded)
x = Conv2D(10, (3, 3), padding='same')(x)
x = LeakyReLU(alpha=0.2)(x)
#x = UpSampling2D((2, 2))(x)
x = Conv2D(12, (3, 3), padding='same')(x)
x = LeakyReLU(alpha=0.2)(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(16, (2, 2), padding='same')(x)
x = LeakyReLU(alpha=0.2)(x)
x = UpSampling2D((3, 3))(x)
x = Conv2D(20, (3, 3), padding='same')(x)
```

```
x = LeakyReLU(alpha=0.2)(x)
x = UpSampling2D((2, 2))(x)
x = Conv2DTranspose(24, (3, 3), strides=(1, 1), padding='valid',␣
 ↪activation="relu")(x)
decoded = Conv2DTranspose(1, (3, 3), strides=(1, 1), padding='valid',␣
 ↪activation='sigmoid',name='output')(x)

CAE_f = Model(input_img, decoded)

CAE_f.summary()
```

[ ]:
```
tf.keras.utils.plot_model(
    CAE_f,
    to_file="CAE_f_plot.png", # lo guarda en este documento
    show_shapes=True,
    show_layer_names=True,
    rankdir="TB",
    expand_nested=False,
    dpi=96,
)
```

Separar el modelo en **Encoder-Decoder**

[ ]:
```
encoder_input = Input(shape=(28, 28, 1), name='original_image')
x = Conv2D(24, (3, 3), padding='same')(encoder_input)
x = LeakyReLU(alpha=0.2)(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(20, (3, 3), padding='same')(x)
x = LeakyReLU(alpha=0.2)(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(16, (3, 3), padding='same')(x)
x = LeakyReLU(alpha=0.2)(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(12, (3, 3), padding='same')(x)
x = LeakyReLU(alpha=0.2)(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x =Conv2D(10, (3, 3), padding='same')(x)
x = LeakyReLU(alpha=0.2)(x)

encoder_output = MaxPooling2D((2, 2), padding='same',name='embedding')(x)

encoder = Model(encoder_input, encoder_output, name='encoder')
```

[ ]:
```
decoder_input = Input(shape=(1,1,10), name='encoded_image')
```

```
x = UpSampling2D((2, 2))(decoder_input)
x = Conv2D(10, (3, 3), padding='same')(x)
x = LeakyReLU(alpha=0.2)(x)
#x = UpSampling2D((2, 2))(x)
x = Conv2D(12, (3, 3), padding='same')(x)
x = LeakyReLU(alpha=0.2)(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(16, (2, 2), padding='same')(x)
x = LeakyReLU(alpha=0.2)(x)
x = UpSampling2D((3, 3))(x)
x = Conv2D(20, (3, 3), padding='same')(x)
x = LeakyReLU(alpha=0.2)(x)
x = UpSampling2D((2, 2))(x)
x = Conv2DTranspose(24, (3, 3), strides=(1, 1), padding='valid',␣
 ↪activation="relu")(x)

decoder_output = Conv2DTranspose(1, (3, 3), strides=(1, 1), padding='valid',␣
 ↪activation='sigmoid',name='output')(x)

decoder = Model(decoder_input, decoder_output, name='decoder')
```

[ ]:
```
decoder.summary()
```

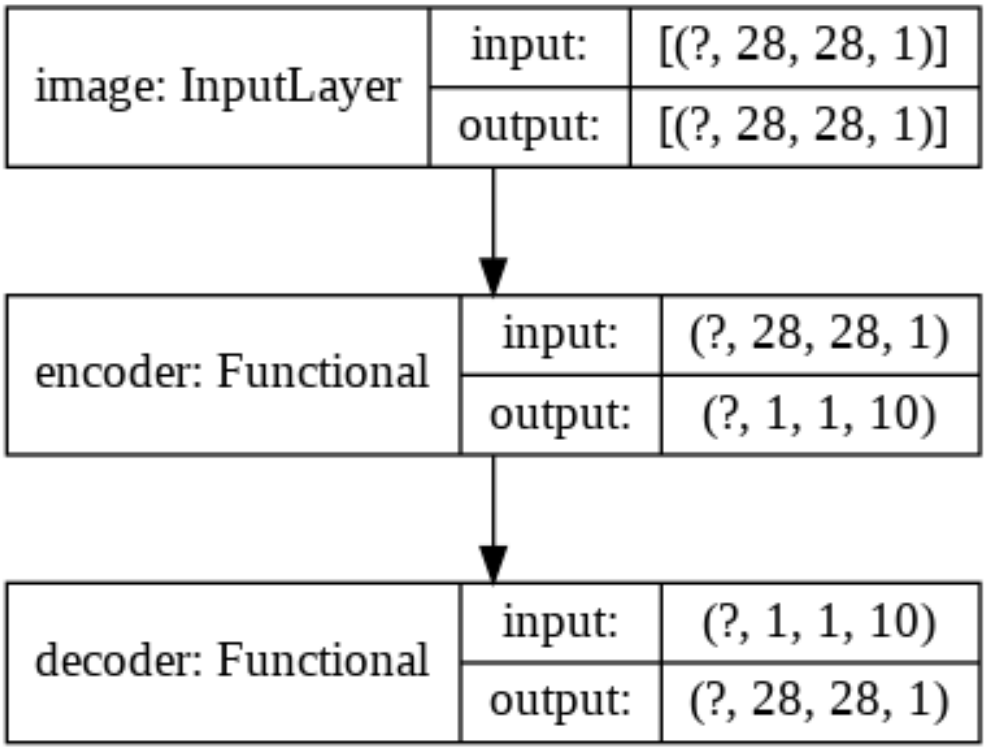
[ ]:
```
autoencoder_input = Input(shape=(28, 28, 1), name="image")
encoded_img = encoder(autoencoder_input)
decoded_img = decoder(encoded_img)
CAE_separated = Model(autoencoder_input, decoded_img, name="CAE_separated")
CAE_separated.summary()
```

```
Model: "CAE_separated"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
image (InputLayer)           [(None, 28, 28, 1)]       0
_________________________________________________________________
encoder (Functional)         (None, 1, 1, 10)          10306
_________________________________________________________________
decoder (Functional)         (None, 28, 28, 1)         10247
=================================================================
Total params: 20,553
Trainable params: 20,553
Non-trainable params: 0
_________________________________________________________________
```

[ ]:
```
tf.keras.utils.plot_model( encoder, to_file="encoder_plot.png",␣
 ↪show_shapes=True, show_layer_names=True)
```

```
tf.keras.utils.plot_model( decoder, to_file="decoder_plot.png",␣
 ↪show_shapes=True, show_layer_names=True)
tf.keras.utils.plot_model( CAE_separated, to_file="CAE_separated_plot.png",␣
 ↪show_shapes=True, show_layer_names=True)
```

[ ]:



[ ]:
```
from  tensorflow.keras.callbacks import EarlyStopping
from  keras.optimizers import Adam


opt = Adam(learning_rate=0.005, epsilon=0.1)

CAE_f.compile(optimizer= opt,␣
 ↪loss='binary_crossentropy',metrics=['mean_squared_error'])

callbacks_list = [ EarlyStopping( monitor='val_loss', patience=10 )]
```

[ ]:
```
historyC10 = CAE_f.fit(x_train_c, x_train_c,
                epochs=400,
                batch_size=256,
                shuffle=True,
```

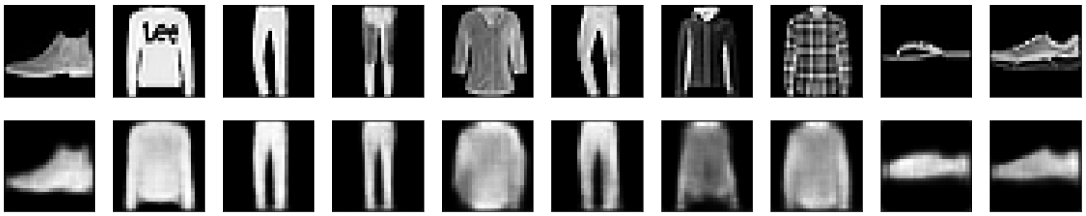
```
                callbacks=callbacks_list,
                validation_data=(x_test_c, x_test_c),
                verbose=1)
```

[ ]:
```
#############

def plot_results( history_NN, min_epoch ):
  max_epoch = len( history_NN.history["val_loss"] )
  val_loss = history_NN.history["val_loss"][min_epoch:max_epoch]
  train_loss = history_NN.history["loss"][min_epoch:max_epoch]
  epochs = range(min_epoch, max_epoch)

  plt.plot( epochs , val_loss, 'b', label='Validation loss')
  plt.plot( epochs , train_loss, 'b+', label='Training loss')
  plt.title('Training and validation loss')
  plt.xlabel('Epochs')
  plt.ylabel('Binary CE')
  plt.legend()

  plt.savefig("training_plot")
  plt.show()

###############

#plot_results( historyC10, 9 )
```

[ ]:
```
CAE_f.save('CAE10adv_fashion.h5')
# Load the model:
# a) keras.models.load_model('clust11.h5')
# b) autoencoder_c = load_model('clust11.h5')
```

[ ]:
```
predM = CAE_f.predict(x_test_c)


n = 10  # how many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
    # display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test_d[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(predM[i].reshape(28, 28))
```

```
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

plt.savefig("reconstruction_images")
plt.show()
```



Visualización de la reconstrucción del input

---

8 Clustering de la CAE 10

[ ]:
```
layer_name = 'embedding'
intermediate_layer_model = Model(inputs=CAE_f.input,
                                 outputs=CAE_f.get_layer(layer_name).output)
embedded_layer = intermediate_layer_model.predict(x_test)
print( embedded_layer.shape )
```

(10000, 1, 1, 10)

[ ]:
```
embedded_layer = embedded_layer.reshape(10000,10)
```

[ ]:
```
kmeans_labels_CAE.shape
```

[ ]: (10000,)

[ ]:
```
import sklearn.cluster as cluster
from sklearn.metrics import normalized_mutual_info_score, adjusted_rand_score,␣
 ↪adjusted_mutual_info_score, silhouette_score
!pip install coclust
from coclust.evaluation.external import accuracy
```

[ ]:
```
kmeans_labels_CAE = cluster.KMeans(n_clusters=10, n_init=10, random_state=1997).
 ↪fit_predict( embedded_layer ) # embedded layer
```

[ ]:
```
print( 'NMI:', normalized_mutual_info_score(y_test, kmeans_labels_CAE) )
print( 'AMI:', adjusted_mutual_info_score(y_test, kmeans_labels_CAE) )
print( 'ARI:', adjusted_rand_score(y_test, kmeans_labels_CAE) )
print( 'ACC:', accuracy(y_test, kmeans_labels_CAE) )
print( 'SIL:', silhouette_score(x_test_d, kmeans_labels_CAE) )
```

```
NMI: 0.48623116118200055
AMI: 0.48531707004835395
ARI: 0.3209213770153575
ACC: 0.5087

/usr/local/lib/python3.6/dist-packages/sklearn/utils/linear_assignment_.py:128:
FutureWarning: The linear_assignment function is deprecated in 0.21 and will be
removed from 0.23. Use scipy.optimize.linear_sum_assignment instead.
  FutureWarning)

SIL: 0.09880054
```

---

9 Autoencoder basado en el ejemplo de chollet

Try autoencoder 128 dim + UMAP to create the visualisation

Fuente: https://blog.keras.io/building-autoencoders-in-keras.html

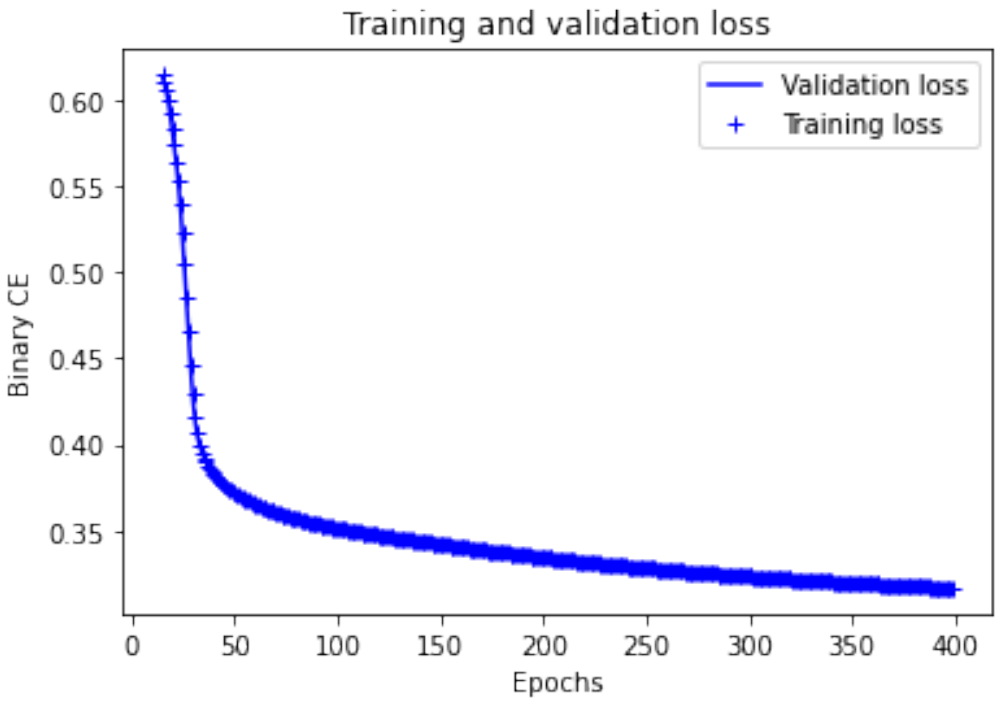
[ ]:
```
from keras.layers import Input, Dense, Conv2D, MaxPooling2D, UpSampling2D
from keras.models import Model
from keras import backend as K

input_img = Input(shape=(28, 28, 1))  # adapt this if using `channels_first`␣
 ↪image data format

x = Conv2D(16, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)

# at this point the representation is (4, 4, 8) i.e. 128-dimensional

x = Conv2D(8, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(16, (3, 3), activation='relu')(x)
```

```
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
```

[ ]:
```
#! tensorboard --logdir sample_data
```

[ ]:
```
from keras.callbacks import TensorBoard, EarlyStopping
callbacks_list = [ EarlyStopping( monitor='val_loss', patience=10 )]

history = autoencoder.fit(x_train_c, x_train_c,
                epochs=400,
                batch_size=128,
                shuffle=True,
                validation_data=(x_test_c, x_test_c),
                callbacks=callbacks_list)
```

[ ]:
```
plot_results( history, 15 )
```



[ ]:
```
autoencoder.summary()
```

```
Model: "functional_1"
_________________________________________________________________
```

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 28, 28, 1)]       0
_________________________________________________________________
conv2d (Conv2D)              (None, 28, 28, 16)        160
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 16)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 14, 14, 8)         1160
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 7, 7, 8)           0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 7, 7, 8)           584
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 4, 4, 8)           0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 4, 4, 8)           584
_________________________________________________________________
up_sampling2d (UpSampling2D) (None, 8, 8, 8)           0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 8, 8, 8)           584
_________________________________________________________________
up_sampling2d_1 (UpSampling2 (None, 16, 16, 8)         0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 14, 14, 16)        1168
_________________________________________________________________
up_sampling2d_2 (UpSampling2 (None, 28, 28, 16)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 28, 28, 1)         145
=================================================================
Total params: 4,385
Trainable params: 4,385
Non-trainable params: 0
_________________________________________________________________
```

[ ]:
```
feature_model = Model(inputs=autoencoder.input, outputs=autoencoder.
 ↪get_layer(name='max_pooling2d_2').output)
features = feature_model.predict(x)
print('feature shape=', features.shape)
```

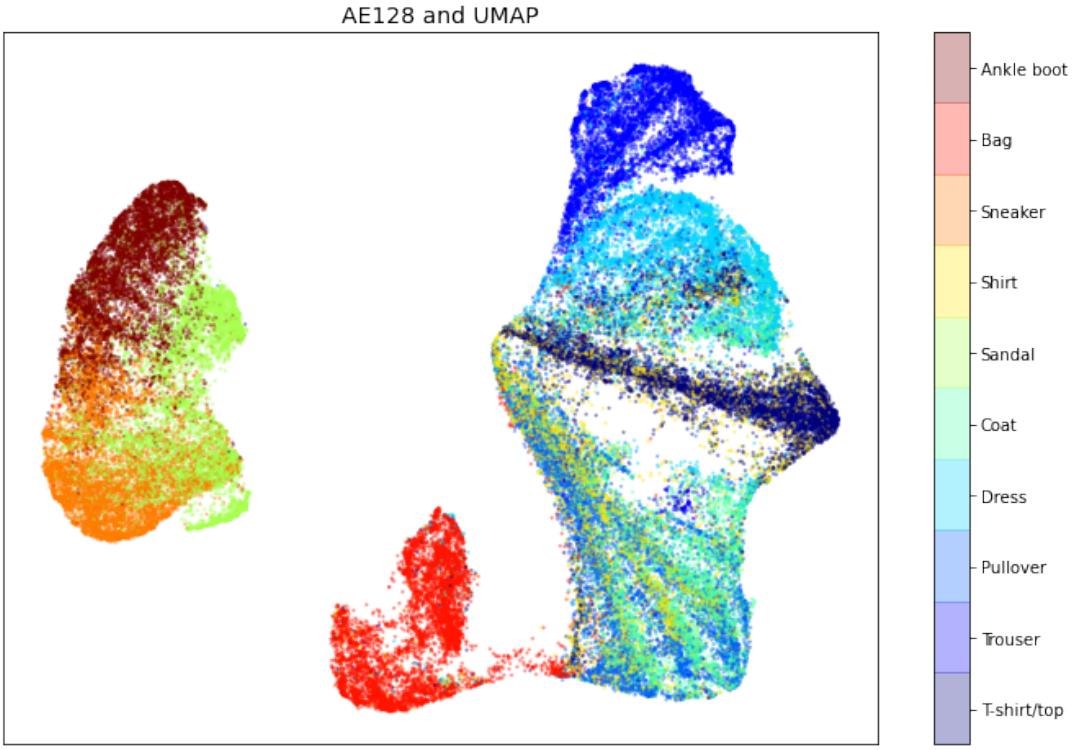
```
feature shape= (70000, 4, 4, 8)
```

[ ]:
```
features = np.reshape(features, newshape=(features.shape[0], -1))
print('feature shape=', features.shape)
```

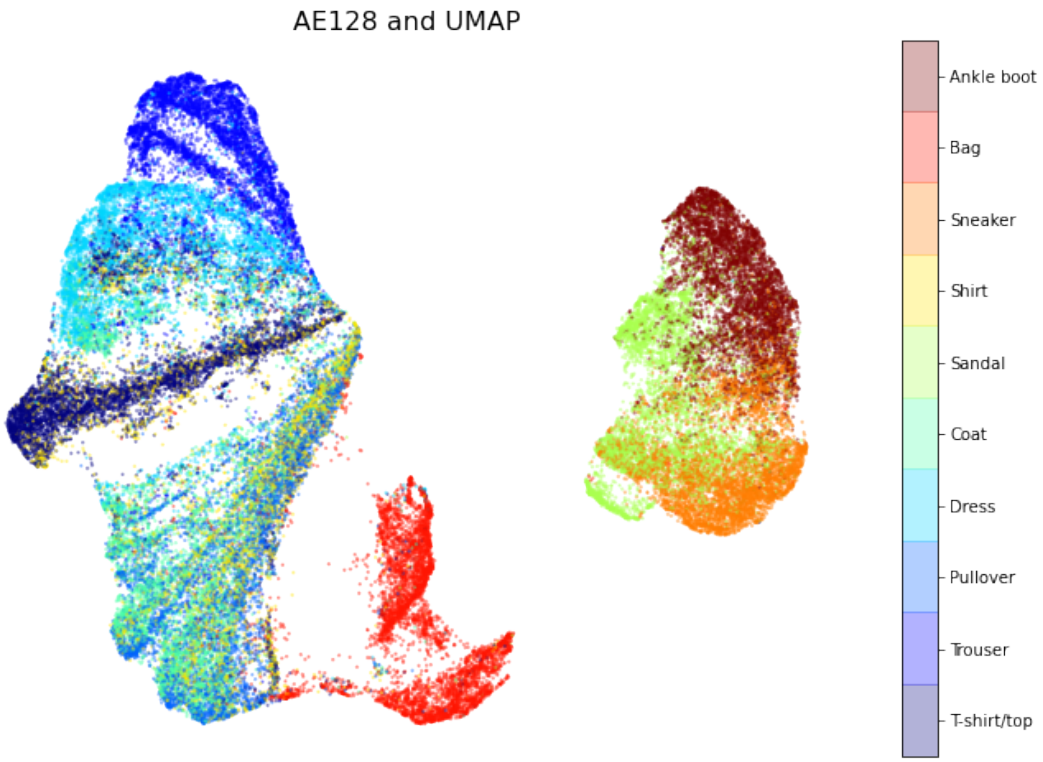
```
feature shape= (70000, 128)
```

[ ]:
```
from umap import UMAP
fitting = UMAP(n_neighbors=15,min_dist=0.1,n_components=2,random_state=1997)
umap_2d = fitting.fit_transform(features)
```

[ ]:
```
fig, ax = plt.subplots(figsize=(12, 8))
plt.scatter( umap_2d[:,0], umap_2d[:,1], c= y, cmap=plt.cm.get_cmap('jet', 10),␣
 ↪s=2, alpha=0.3,label=Target_name)
plt.setp(ax, xticks = [], yticks = [])
ax.set_title('AE128 and UMAP', fontsize=14)
#plt.title(titol, fontsize=18)

cbar = plt.colorbar(boundaries = np.arange(11)-0.5, cmap=plt.cm.get_cmap('jet',␣
 ↪10))
cbar.set_ticks(np.arange(10))
cbar.set_ticklabels(noms)
#cbar.ax.set_ylabel('Label of each garment', rotation=270, fontsize=10)

fig.savefig( 'AE128_UMAP2' )
plt.show()
```



[ ]:
```
grafico( umap_2d, titol="AE128 and UMAP" , fitxer="AE128_UMAP2"  )
```



[ ]:

---

10 Otra estructura de CAE

Comparar CAE con el propuesto en el paper de DCEC Guo et.al.

[ ]:
```
from IPython.display import Image
Image('/CAE_XifenGuo.png', width=862, height=540)
```

[ ]:

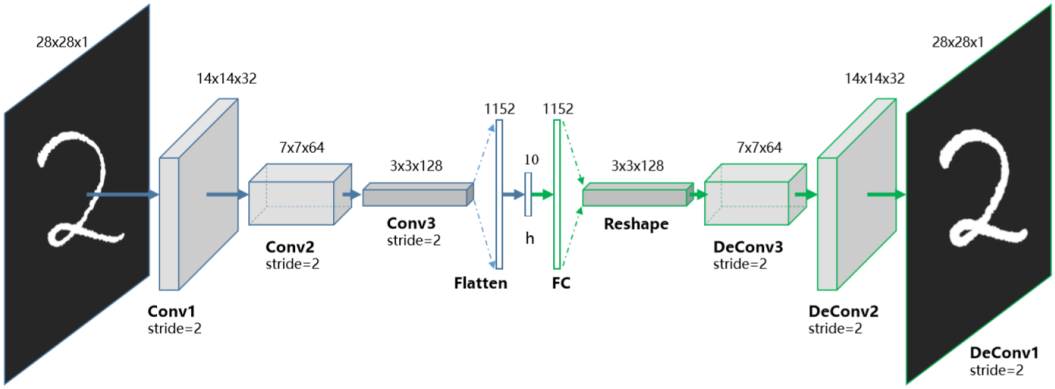


Fig. 1. The structure of proposed Convolutional AutoEncoders (CAE) for MNIST. In the middle there is a fully connected autoencoder whose embedded layer is composed of only 10 neurons. The rest are convolutional layers and convolutional transpose layers (some work refers to as Deconvolutional layer). The network can be trained directly in an end-to-end manner.

[ ]:
```
%tensorflow_version 2.x
from tensorflow.keras.layers import Input, Dense, Conv2D, MaxPooling2D,␣
 ↪UpSampling2D, Conv2DTranspose, LeakyReLU, Flatten, Reshape
# hay que añadir los dos ultimos (Flatten, Reshape)
from tensorflow.keras.models import Model
from tensorflow.keras import backend

input_img = Input(shape=(28, 28, 1))  # adapt this if using `channels_first`␣
 ↪image data format

x = Conv2D(32, (3, 3), strides=(2, 2), padding='same', name="Conv1" )(input_img)
x = LeakyReLU(alpha=5)(x)
x = Conv2D(64, (3, 3), strides=(2, 2), padding='same', name="Conv2" )(x)
x = LeakyReLU(alpha=5)(x)
x = Conv2D(128, (3, 3), strides=(2, 2), padding='valid', name="Conv3" )(x)
x = LeakyReLU(alpha=5)(x)
x = Flatten( name="Flatten")(x)
encoded = Dense( 10, activation='relu', name="Embedding" )(x)

###

x = Dense( 1152, activation='relu' , name="FC" )(encoded)
x = Reshape( (3,3,128) , name="Reshape")(x)
```

```
x = Conv2DTranspose(64, (3, 3), strides=(2, 2), padding='valid',␣
 ↪name="Deconv3")(x) # importante, unico que tiene valid
x = LeakyReLU(alpha=5)(x)
x = Conv2DTranspose(32, (3, 3), strides=(2, 2), padding='same',␣
 ↪name="Deconv2")(x)
x = LeakyReLU(alpha=5)(x)
decoded = Conv2DTranspose(1, (3, 3), strides=(2, 2), padding='same',␣
 ↪activation='sigmoid', name="Deconv1")(x)

autoencoder_c2 = Model(input_img, decoded)

autoencoder_c2.summary()
```

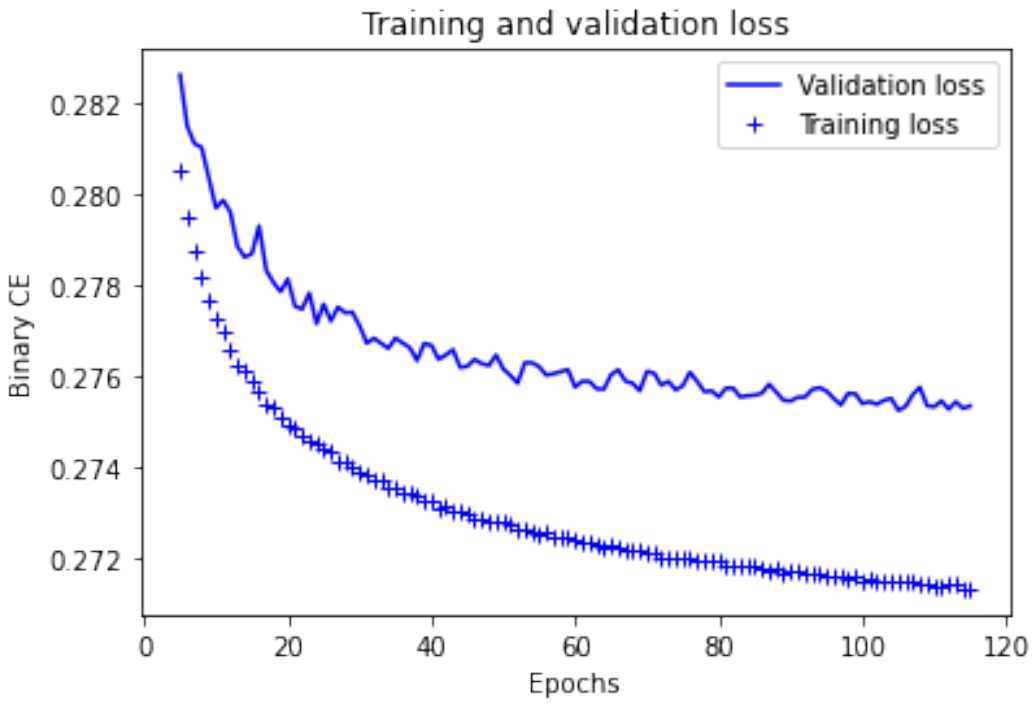
```
Model: "functional_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 28, 28, 1)]       0
_________________________________________________________________
Conv1 (Conv2D)               (None, 14, 14, 32)        320
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 14, 14, 32)        0
_________________________________________________________________
Conv2 (Conv2D)               (None, 7, 7, 64)          18496
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)    (None, 7, 7, 64)          0
_________________________________________________________________
Conv3 (Conv2D)               (None, 3, 3, 128)         73856
_________________________________________________________________
leaky_re_lu_7 (LeakyReLU)    (None, 3, 3, 128)         0
_________________________________________________________________
Flatten (Flatten)            (None, 1152)              0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                11530
_________________________________________________________________
FC (Dense)                   (None, 1152)              12672
_________________________________________________________________
Reshape (Reshape)            (None, 3, 3, 128)         0
_________________________________________________________________
Deconv3 (Conv2DTranspose)    (None, 7, 7, 64)          73792
_________________________________________________________________
leaky_re_lu_8 (LeakyReLU)    (None, 7, 7, 64)          0
_________________________________________________________________
Deconv2 (Conv2DTranspose)    (None, 14, 14, 32)        18464
_________________________________________________________________
leaky_re_lu_9 (LeakyReLU)    (None, 14, 14, 32)        0
_________________________________________________________________
```

```
Deconv1 (Conv2DTranspose)    (None, 28, 28, 1)         289
=================================================================
Total params: 209,419
Trainable params: 209,419
Non-trainable params: 0
_________________________________________________________________
```

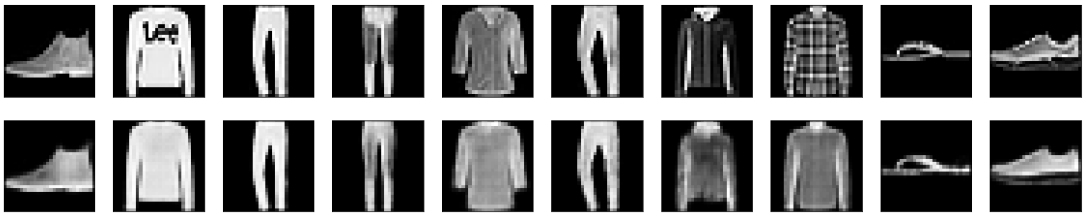
[ ]:
```
from  tensorflow.keras.callbacks import EarlyStopping

autoencoder_c2.compile(optimizer='Adam',␣
 ↪loss='binary_crossentropy',metrics=['mae','mean_squared_error'])

callbacks_list = [ EarlyStopping( monitor='val_loss', patience=10 )]
```

[ ]:
```
history_c2 = autoencoder_c2.fit(x_train_c, x_train_c,
                epochs=400,
                batch_size=128,
                shuffle=True,
                callbacks=callbacks_list,
                validation_data=(x_test_c, x_test_c)  )
```

[ ]:
```
plot_results( history_c2, 5 )
```



[ ]:
```
predM = autoencoder_c2.predict(x_test_c)


n = 10  # how many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
    # display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test_d[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(predM[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

plt.savefig("reconstruction_images")
plt.show()
```



For visualisation

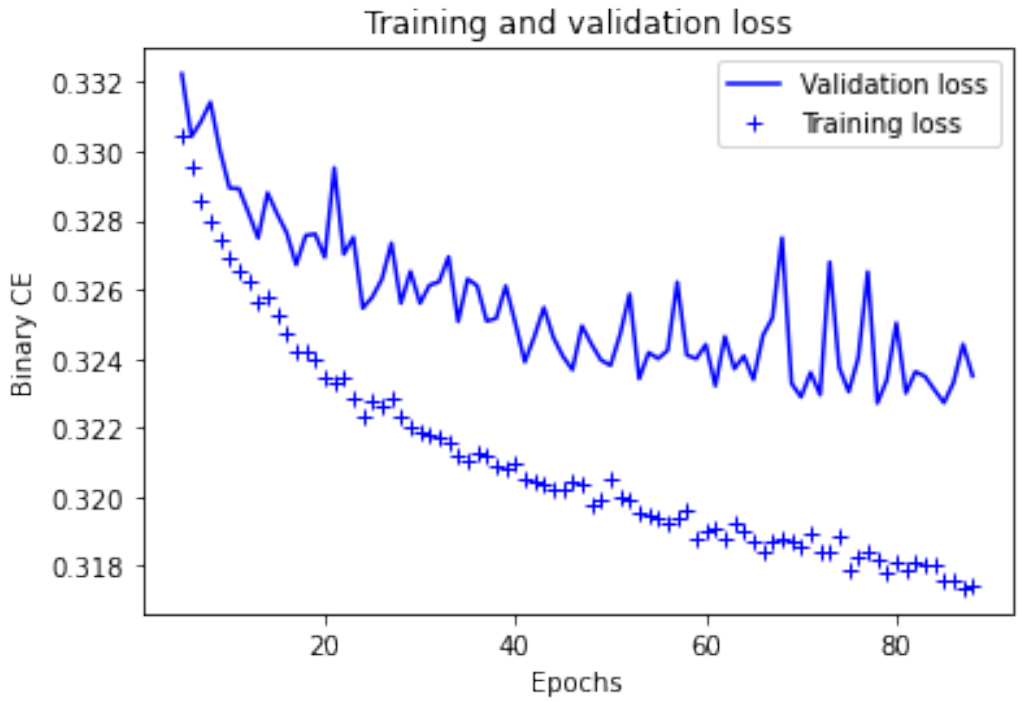
[ ]:
```
%tensorflow_version 2.x
from tensorflow.keras.layers import Input, Dense, Conv2D, MaxPooling2D,␣
 ↪UpSampling2D, Conv2DTranspose, LeakyReLU, Flatten, Reshape
# hay que añadir los dos ultimos (Flatten, Reshape)
from tensorflow.keras.models import Model
from tensorflow.keras import backend

input_img = Input(shape=(28, 28, 1))  # adapt this if using `channels_first`␣
 ↪image data format

x = Conv2D(32, (3, 3), strides=(2, 2), padding='same', name="Conv1" )(input_img)
x = LeakyReLU(alpha=5)(x)
x = Conv2D(64, (3, 3), strides=(2, 2), padding='same', name="Conv2" )(x)
```

```
x = LeakyReLU(alpha=5)(x)
x = Conv2D(128, (3, 3), strides=(2, 2), padding='valid', name="Conv3" )(x)
x = LeakyReLU(alpha=5)(x)
x = Flatten( name="Flatten")(x)
encoded = Dense( 2, activation='relu', name = "Embedding" )(x)

###

x = Dense( 1152, activation='relu' , name="FC" )(encoded)
x = Reshape( (3,3,128) , name="Reshape")(x)
x = Conv2DTranspose(64, (3, 3), strides=(2, 2), padding='valid',␣
 ↪name="Deconv3")(x) # importante, unico que tiene valid
x = LeakyReLU(alpha=5)(x)
x = Conv2DTranspose(32, (3, 3), strides=(2, 2), padding='same',␣
 ↪name="Deconv2")(x)
x = LeakyReLU(alpha=5)(x)
decoded = Conv2DTranspose(1, (3, 3), strides=(2, 2), padding='same',␣
 ↪activation='sigmoid', name="Deconv1")(x)

autoencoder_c2 = Model(input_img, decoded)

#autoencoder_c2.summary()
```

[ ]:
```
from  tensorflow.keras.callbacks import EarlyStopping

autoencoder_c2.compile(optimizer='Adam',␣
 ↪loss='binary_crossentropy',metrics=['mae','mean_squared_error'])

callbacks_list = [ EarlyStopping( monitor='val_loss', patience=10 )]
```

[ ]:
```
history_c2 = autoencoder_c2.fit(x_train_c, x_train_c,
                epochs=150,
                batch_size=128,
                shuffle=True,
                callbacks=callbacks_list,
                validation_data=(x_test_c, x_test_c)  )
```

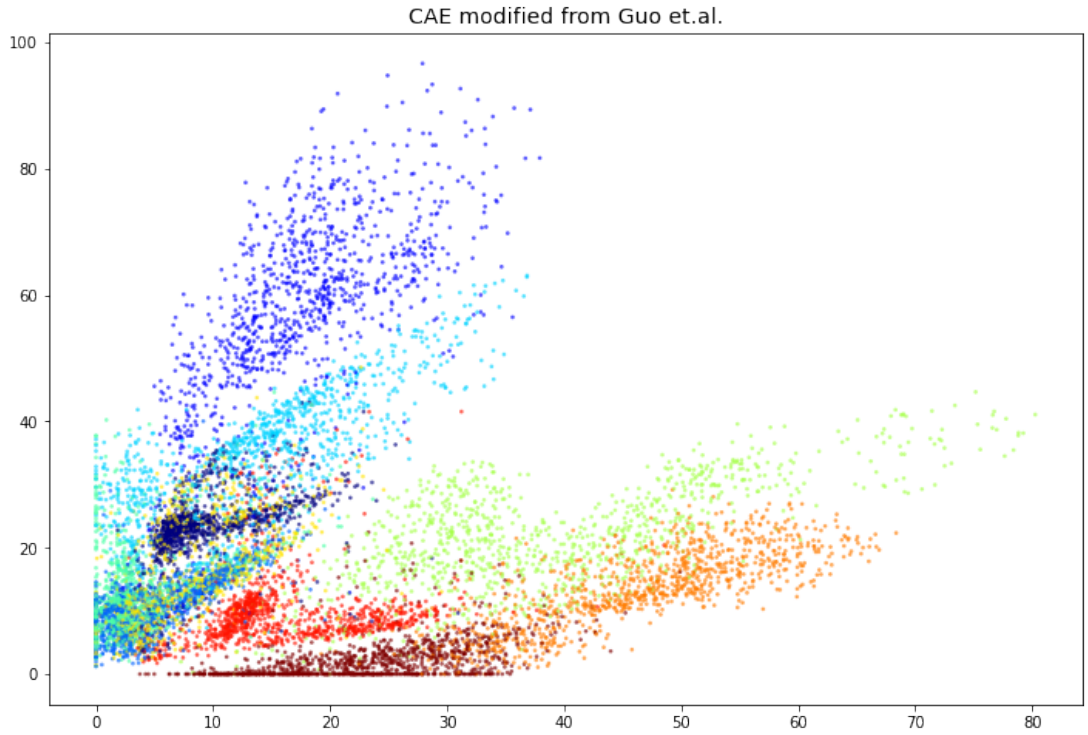
[ ]:
```
plot_results( history_c2, 5 )
```



[ ]:
```
intermediate_layer_model = Model(inputs=autoencoder_c2.input,
                                 outputs=autoencoder_c2.get_layer('Embedding').
 ↪output)
embedded_layer = intermediate_layer_model.predict(x_test_c)
print( embedded_layer.shape )
```

```
(10000, 2)
```

[ ]:
```
fig, ax = plt.subplots(figsize=(12, 8))
plt.scatter( embedded_layer[:,0], embedded_layer[:,1], c= y_test, cmap=plt.cm.
 ↪get_cmap('jet', 10), s=3, alpha=0.5)
#plt.setp(ax, xticks = [], yticks = [])
ax.set_title('CAE modified from Guo et.al.', fontsize=14)
plt.savefig('CAE_Guo_vis')
```



---

11 CAE 2D para visualizacion

[ ]:
```
input_img = Input(shape=(28, 28, 1), name='input')  # adapt this if using␣
 ↪`channels_first` image data format

x = Conv2D(16, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Flatten()(x)
encoded = Dense( 2, activation='relu' ,name="embedded" )(x)

# at this point the representation is 2 dimensional

x = Dense( 128, activation='relu' )(encoded)
x = Reshape( (4,4,8) )(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
```

```
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(16, (3, 3), activation='relu')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

CAE_f = Model(input_img, decoded)

CAE_f.summary()
```

```
Model: "functional_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 28, 28, 1)]       0
_________________________________________________________________
conv2d (Conv2D)              (None, 28, 28, 16)        160
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 16)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 14, 14, 8)         1160
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 7, 7, 8)           0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 7, 7, 8)           584
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 4, 4, 8)           0
_________________________________________________________________
flatten (Flatten)            (None, 128)               0
_________________________________________________________________
embedded (Dense)             (None, 2)                 258
_________________________________________________________________
dense (Dense)                (None, 128)               384
_________________________________________________________________
reshape (Reshape)            (None, 4, 4, 8)           0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 4, 4, 8)           584
_________________________________________________________________
up_sampling2d (UpSampling2D) (None, 8, 8, 8)           0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 8, 8, 8)           584
_________________________________________________________________
up_sampling2d_1 (UpSampling2 (None, 16, 16, 8)         0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 14, 14, 16)        1168
_________________________________________________________________
up_sampling2d_2 (UpSampling2 (None, 28, 28, 16)        0
```

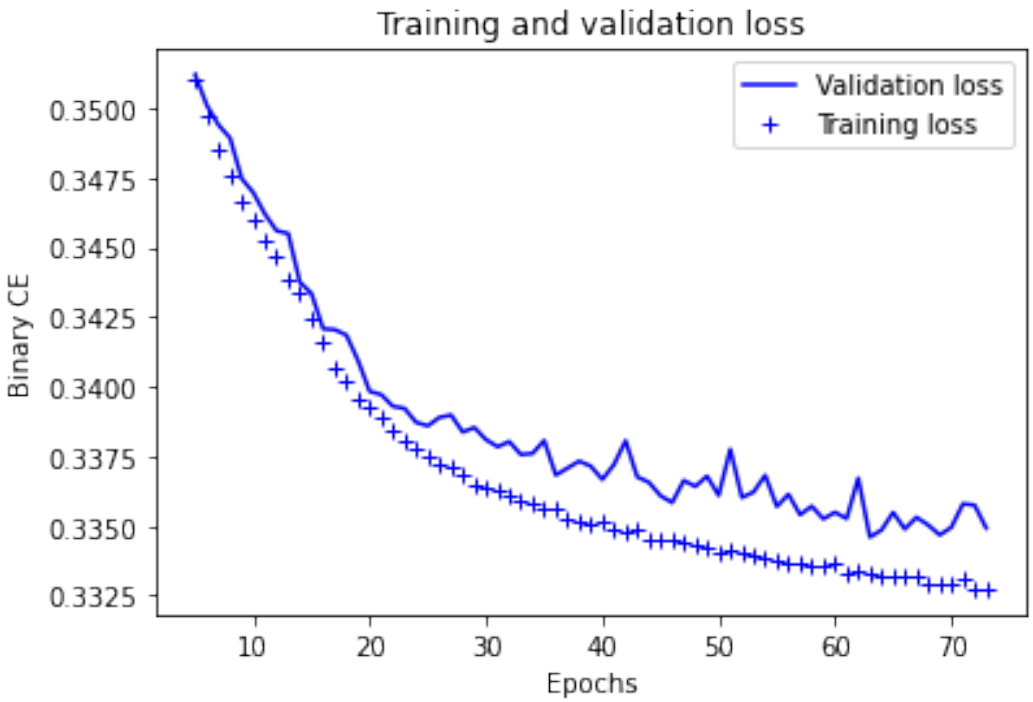
```
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 28, 28, 1)         145
=================================================================
Total params: 5,027
Trainable params: 5,027
Non-trainable params: 0
_________________________________________________________________
```

[ ]:
```
from  tensorflow.keras.callbacks import EarlyStopping

CAE_f.compile(optimizer='Adam',␣
 ↪loss='binary_crossentropy',metrics=['mean_squared_error']) #␣
 ↪experimental_steps_per_execution = 10

callbacks_list = [ EarlyStopping( monitor='val_loss', patience=10 )]
```

[ ]:
```
historyC = CAE_f.fit(x_train_c, x_train_c,
                epochs=120,
                batch_size=128,
                shuffle=True,
                callbacks=callbacks_list,
                validation_data=(x_test_c, x_test_c),
                verbose=1)
```

[ ]:
```
#############

def plot_results( history_NN, min_epoch ):
  max_epoch = len( history_NN.history["val_loss"] )
  val_loss = history_NN.history["val_loss"][min_epoch:max_epoch]
  train_loss = history_NN.history["loss"][min_epoch:max_epoch]
  epochs = range(min_epoch, max_epoch)

  plt.plot( epochs , val_loss, 'b', label='Validation loss')
  plt.plot( epochs , train_loss, 'b+', label='Training loss')
  plt.title('Training and validation loss')
  plt.xlabel('Epochs')
  plt.ylabel('Binary CE')
  plt.legend()

  plt.savefig("training_plot")
  plt.show()

###############


#plot_results( historyC, 5 )
```

[ ]:
```
plot_results( historyC, 5 )
```



[ ]:
```
autoencoder_c.save('CAE1_fashion.h5')
```

Hacer grafico 2D

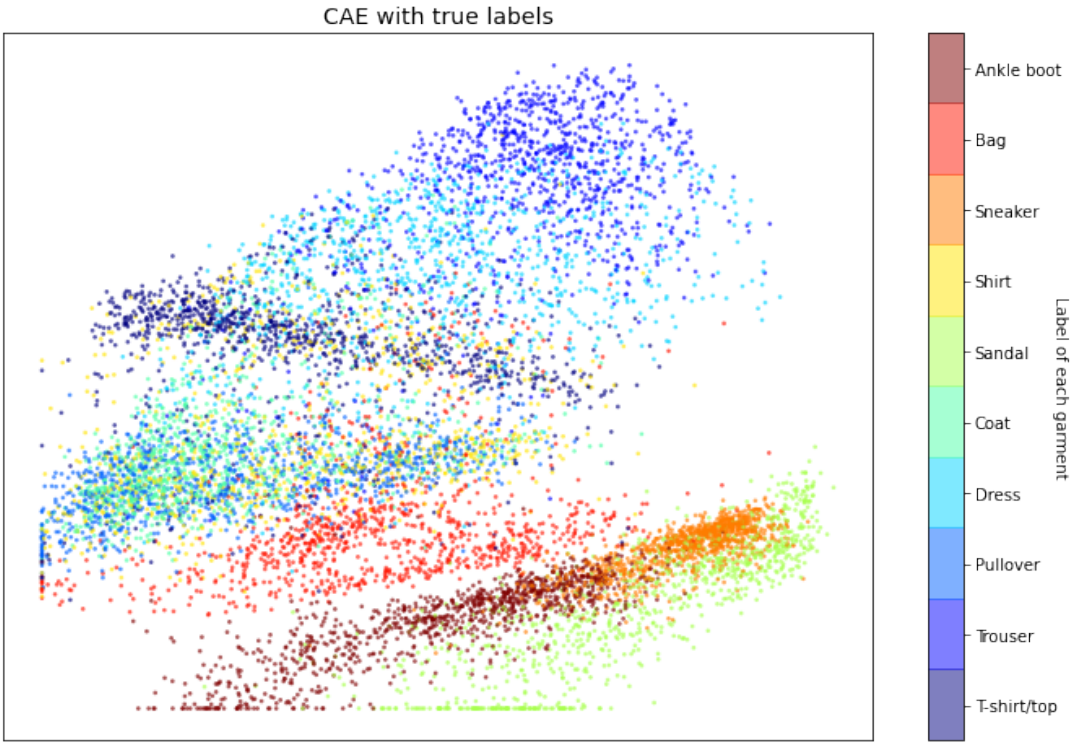
[ ]:
```
intermediate_layer_model = Model(inputs=CAE_f.input,
                                 outputs=CAE_f.get_layer(name='embedded').output)
feature_space = intermediate_layer_model.predict(x_test_c) # se hace con los␣
 ↪10mil datos del test
```

[ ]:
```
#feature_space = np.reshape(feature_space, newshape=(10000, 10)).copy()
feature_space.shape
```

[ ]:
```
(10000, 2)
```

[ ]:
```
fig, ax = plt.subplots(figsize=(12, 8))
plt.scatter( feature_space[:,0], feature_space[:,1], c= y_test, cmap=plt.cm.
 ↪get_cmap('jet', 10), s=3, alpha=0.5,label=Target_name_test)
plt.setp(ax, xticks = [], yticks = [])
ax.set_title('CAE with true labels', fontsize=14)
#plt.title(titol, fontsize=18)
```

```
cbar = plt.colorbar(boundaries = np.arange(11)-0.5, cmap=plt.cm.get_cmap('jet',␣
 ↪10))
cbar.set_ticks(np.arange(10))
cbar.set_ticklabels(noms)
cbar.ax.set_ylabel('Label of each garment', rotation=270, fontsize=10)

fig.savefig( 'CAE_2d_fashion' )
plt.show()
```



[ ]:
```
fig, (ax1, ax2) = plt.subplots( nrows=1,ncols=2,figsize=(18, 10) )

ax1.scatter( tsne_results[:,0], tsne_results[:,1], c= y_rfull, cmap=plt.cm.
 ↪get_cmap('jet', 10), s=2 ,alpha=0.3, label=Target_namer )
ax1.axis('off')

ax2.scatter( tsne_results[:,0], tsne_results[:,1],  c= "black", s=2 ,alpha=0.
 ↪3,label=Target_name )
ax2.axis('off')

fig.savefig( 'tsne_fashion' )
plt.show()
```

https://www.kaggle.com/eliotbarr/fashion-mnist-tutorial

12 Autoencoders + tsne30

[ ]:
```
input_img = Input(shape=(28, 28, 1), name='input')  # adapt this if using␣
 ↪`channels_first` image data format

x = Conv2D(16, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Flatten()(x)
encoded = Dense( 30, activation='relu' ,name="embedded" )(x)

# at this point the representation is 2 dimensional

x = Dense( 128, activation='relu' )(encoded)
x = Reshape( (4,4,8) )(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(16, (3, 3), activation='relu')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

CAE_30d = Model(input_img, decoded)

CAE_30d.summary()
```

[ ]:
```
from  tensorflow.keras.callbacks import EarlyStopping

CAE_30d.compile(optimizer='Adam',␣
 ↪loss='binary_crossentropy',metrics=['mean_squared_error'])

callbacks_list = [ EarlyStopping( monitor='val_loss', patience=10 )]
```

[ ]:
```
historyC30 = CAE_30d.fit(x_train_c, x_train_c,
                epochs=120,
                batch_size=128,
                shuffle=True,
                callbacks=callbacks_list,
                validation_data=(x_test_c, x_test_c),
                verbose=1)
```

[ ]:
```
intermediate_layer_model = Model(inputs=CAE_f.input,
                                 outputs=CAE_f.get_layer(name='embedded').output)
feature_space_30 = intermediate_layer_model.predict(x_test_c)
```

[ ]:
```
feature_space_30.shape
```

[ ]:
```
(10000, 30)
```

[ ]:
```
from sklearn.manifold import TSNE

tsne1 = TSNE( n_components= 2, perplexity= 25.0, learning_rate= 10.0,␣
 ↪n_iter=5000, random_state= semilla, n_iter_without_progress=200 )
tsne_results = tsne1.fit_transform(feature_space_30)
```

[ ]:
```
fig, (ax1, ax2) = plt.subplots( nrows=1,ncols=2,figsize=(18, 10) )

ax1.scatter( tsne_results[:,0], tsne_results[:,1], c= y_test, cmap=plt.cm.
 ↪get_cmap('jet', 10), s=2 ,alpha=0.3,label=Target_name )
#ax1.set_title('PCA with true labels', fontsize=14)
ax1.axis('off')


ax2.scatter( tsne_results[:,0], tsne_results[:,1],  c= "black", s=2 ,alpha=0.
 ↪3,label=Target_name )
ax2.axis('off')
#ax2.set_title('PCA without labels', fontsize=14)


#plt.title("titol", fontsize=18)
#plt.legend(Target_name, y_rfull)
#fig.savefig( 'CAE +' )
plt.show()

#plt.savefig('cifar10_dog.png')
```