Università degli Studi di Padova

Department of Information Engineering

Master Thesis in Computer Engineering

Minimum spanning tree reconstruction using autoencoders

Supervisor
Simone Milani
Università di Padova

Master Candidate
Riccardo Castelletto

to my family and Martina, because they never left me alone.

Abstract

The estimation of the minimum spanning tree (MST) is one of the most important tasks of graph theory and presents many different applications in different technical fields, ranging from network management to multimedia forensics. Since this graph theory problem has many applications in various fields, algorithms of every kind have been studied and developed specifically for this task. The current work propose a new approach to estimate the minimum spanning tree for a complete graph whose edge weights are affected by a significant amount of noise. The main idea is to model the MST estimation as an image denoising problem, where the input noisy image corresponds to the dissimilarity matrix and the desired output is a binary adjacency matrix.

The denoising task was implemented using a deep learning strategy based on autoencoder structures, and in this work different typologies of autoencoder are discussed. The estimation results were compared to those obtained with state-of-the-art algorithm by means of some metrics measuring the percentage of correctly-estimated roots, leaves and relations among nodes. The results obtained are very encouraging because the model outperforms the existing strategies in accuracy and flexibility. Moreover, the designed solution can be easily extended to other MST estimation problems.

Contents

Abstract v

List of figures x

List of tables xv

1 Introduction 1

2 Neural Networks: a short introduction 3

2.1 Brain model 3

2.2 Neuron 4

2.3 Neural network 6

2.4 Training 7

2.4.1 Learning 7

2.4.2 Backpropagation 10

2.4.3 Problems 13

2.5 Validation 14

2.6 Test 15

2.7 Overfitting/Underfitting 15

3 Deep neural networks 17

3.1 Convolutional neural network 21

3.2 Residual neural network 25

3.3 Autoencoder 27

3.4 Mathematical approach 29
3.4.1 Encoder 29
3.4.2 Decoder 29
3.5 Regularized autoencoders 30
3.5.1 Sparse autoencoders 30
3.5.2 Denoising autoencoders 32
3.5.3 Contractive autoencoders 34
3.6 Applications of autoencoders 35

4 Minimum Spanning Tree **37**
4.1 Graph notation 37
4.2 Properties 39
4.3 Algorithms 41
4.3.1 Brute-force algorithm 41
4.3.2 Borůvka algorithm 42
4.3.3 Prim's algorithm 43
4.3.4 Kruskal algorithm 44
4.4 Applications 45

5 Applications of minimum spanning tree estimation **47**
5.1 Introduction 47
5.2 Image filtering 47
5.3 Mathematical problems 48
5.4 Image segmentation 49
5.5 Clustering 49
5.6 Image phylogeny 51
5.7 Baseline 54
5.7.1 Phylogenetic analysis of a set of images 55

5.7.2 Analysis 57
5.7.3 Test datasets 58
5.7.4 Results 59
5.7.5 Conclusions 60

6 Neural models for minimum spanning tree estimation **61**
6.1 Architectures 61
6.1.1 Modified autoencoder 61
6.1.2 ResNet 63
6.1.3 U-Net 65

7 Experimental results **69**
7.1 Dataset creation 69
7.2 Metrics used 72
7.3 MNIST and first partial results 73
7.4 UCID analysis to find the best model 76
7.5 Actual analysis 80
7.5.1 Complete graphs 81
7.5.2 Dropped nodes 84
7.5.3 Audio samples 86

8 Conclusion **87**

References **89**

Acknowledgments **97**

Ringraziamenti **98**

Listing of figures

2.1 Neuron comparison. 4
2.2 Multiple hidden layers. 9
2.3 Dropout. 9
2.4 Learning rate. 10
2.5 Weights. 12
2.6 Overfit-Underfit. 16

3.1 Fully-connected neural network. 18
3.2 Fully-connected neural network with many layers. 19
3.3 Features. 20
3.4 Output of convolutional layer. 22
3.5 Max-pooling. 23
3.6 Avg-pooling. 23
3.7 Learned invariance. 24
3.8 Convolutional Neural Network. 25
3.9 ResNet block. 26
3.10 First ResNet architecture. 27
3.11 Autoencoder schema. 28
3.12 Autoencoder example. 30
3.13 Kullback Leibler example. 32
3.14 A denoising autoencoder. 33
3.15 An example of a denoising autoencoder. 34

3.16 An example of coloring autoencoders. 35
3.17 An example of dimensionality reduction using autoencoders. 36
3.18 An example of watermark removal using autoencoders. 36

4.1 Example of multiple MSTs. 39
4.2 Example of cycle property. 40
4.3 Example of a cut-set. 40
4.4 Complexity of brute-force MST. 42
4.5 Borůvka algorithm. 43
4.6 Prim's algorithm. 44
4.7 Kruskal algorithm. 45

5.1 How image segmentation with MST works. 50
5.2 Phylogeny tree. 53
5.3 Joly et al. work about near-duplicate detection. 54
5.4 Oriented Kruskal example. 58

6.1 Simple schema of the modified autoencoder. 63
6.2 Encode phase of the ResNet model built for this work. 64
6.3 Decoder of the ResNet model. 65
6.4 Representation of the first proposed U-Net. 66
6.5 The U-Net built for this work. 67

7.1 Some samples from the UCID dataset. 70
7.2 Dissimilarity matrix example. 71
7.3 Groundtruth matrix example. 72
7.4 An example of MNIST digit. 74
7.5 Loss of the standard autoencoder. 75
7.6 Loss of the ResNet autoencoder. 75

7.7 Loss of the U-Net autoencoder. 76
7.8 Losses with 8 nodes. 81
7.9 Losses with 16 nodes. 82
7.10 Losses with 32 nodes. 83
7.11 Metrics of the dropped 16 nodes dataset. 85
7.12 Metrics of the dropped 32 nodes dataset. 85

Listing of tables

7.1 Modified autoencoder comparison table 77
7.2 Res-Net based autoencoder comparison table 78
7.3 U-Net based autoencoder comparison table 79
7.4 U-Net based autoencoder with 8 nodes per graph 81
7.5 U-Net based autoencoder with 16 nodes per graph 83
7.6 U-Net based autoencoder with 32 nodes per graph 84
7.7 Analysis on the 8 nodes graph made with audio samples 86

1

Introduction

The minimum spanning tree (MST) estimation is an important task in graph theory that presents several applications in different fields, randing from network management, cluster analysis, image segmentation and denoising, topology and multimedia forensics, to mention only some of them. This mathematical structure is able to correctly estimate the inner relationships within a set of data, extrapolating them from a complete graph structure describing the relations interlying among all the elements of the input dataset. Many real-life situations could be represented with this data structure. Therefore, almost every context is a possible test-bed for the minimum spanning tree computation.

Common situations where the MST problem finds today a dense research are the treatment of the data, with fields like clustering and image manipulation, where this problem has been exploited to obtain important results about the inner compositions of the data considered, and some pure mathematical concepts, where resembling the given information to a graph and then trying to find the inner MST-like frames between them, can highlight important

structures that are useful to comprehend the initial problem. Shortly, the minimum spanning tree estimation has a solid utility everywhere there is the need of finding a defined relation between a representation of some data.

This work proposes to approximate the minimum spanning tree of a set of data as the denoised version of the matrix made by the values of dissimilarity between couples of samples from the same set. Consequently, under this assumption, the minimum spanning tree finding problem can be simplified to a denoising problem. The denoising algorithms in this work were implemented using deep learning strategies, in particular to a specific structure of deep learning models, called autoencoder. The autoencoder, as one of the first learning models presented, has a simple structure that, up to today, has been used and re-engineered several times to face a broad kind of applications. One of these is the denoising.

Starting from some preliminary studies on the reconstruction of the minimum spanning tree to estimate the phylogeny tree of a set of data, this work describes how to reconstruct a valid MST using some of the most advanced neural methodologies; in our experimental results we tested the robustness and the flexibility of this approach by changing the size of the analyzed datasets and by removing randomly some elements.

Further analysis on corrupted datasets highlights the good behaviour of this approach even in realistic situations, where some of the data could be missing.

"Very simple. Just keep adding layers until the test error does not improve anymore."

Unknown

2

Neural Networks: a short introduction

2.1 Brain model

The human brain is composed of connected structures that can interact with each other, transmit the information and produce a precise reaction to a particular input. These structures are the neurons, electrically excitable cells connected by synapses and axons. The main function of these kind of cells is to receive a signal in response to a well-defined stimulus and then forward it, after some processing, to the next cells, which are connected to them. The signal is an electrical/chemical process which is generated by any internal or external interaction of the body. The final result of this propagation process is an interpretation of the stimulus or a reaction to it.

Recently, this connection concept has been proposed to build a learning model for computers.

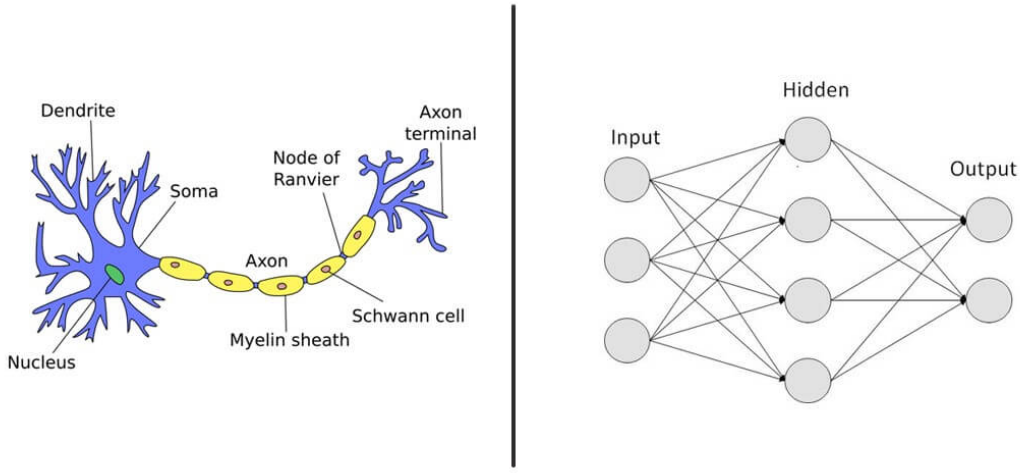


Figure 2.1: Comparison between biological neurons and artificial neurons [1].

2.2 Neuron

An artificial neuron is a mathematical function which takes one or more inputs and combines them to create an output. Interconnecting these neurons makes possible to create a network, which is able, after a learning phase where all the variables of the network are set, to generate an output which is coherent with the given task. The visual comparison between biological neuron and artificial neuron is visible in figure 2.1.
Each artificial neuron implements the function:

$$y_k = \phi(\sum_{j=0}^{m} w_{kj} x_j) \tag{2.1}$$

where y_k is the output of the k-th neuron, x_j is the value of the j-th input, w_{kj} is the weight of the edge connecting input *j* to neuron *k* and ϕ is a function called *activation function*.
A single unit of these has a limited learning ability, since it can implement a simple linear decision function. Their concatenation, instead, can perform very complex tasks.
The main components of equation 2.1 are:

- **Values**: these are the inputs given to the model;
- **Weights**: they control how much each input affects the output value;
- **Activation function**: the sum of all the input values multiplied by their corresponding weight is then processed by this non-linear function. The final result is the actual output of the neuron. Since this function defines the type of output obtained, there are many of them, all suitable for different problems;

These are some of the most used activation functions:

- **Step**: the output of this function is a binary value, which depends on a given threshold θ;

$$f(x) = \begin{cases} 0 & x < \theta \\ 1 & x \geq \theta \end{cases} \tag{2.2}$$

- **Sigmoid**: it is a simple non-linear function, whose derivative can be easily computed.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

  The function is "S"-shaped, and the output is between 0 and 1. While the algorithm is in the training phase, the activation function needs to be derived several times, and therefore, the use of sigmoid can lighten the computation;

- **Rectifier**: this non-linear function returns the positive part of the input argument:

$$f(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases} \tag{2.4}$$

  It has been shown that deeper networks can be trained more effectively [2] by exploiting this function in their layers;

- **Softmax**: this function is a way to convert a k-dimensional vector into another k-dimensional vector where each of its value is between 0 and 1, and their sum is 1. The first objective of this function is to provide a likelihood value for each output value instead of a hard value.

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}} \tag{2.5}$$

2.3 Neural network

A neural network is a system where several of these neurons are interconnected to form a single structure. The neurons are typically organized into layers, where the "input layer" represent the data to be processed and the output layer is the result of the processing. All the other layers are called "hidden layers"; the number of layers identifies the depth of the network.

Each layer is connected to the next one by a certain number of edges, and the connection pattern affects the behaviour of the entire network. The most common pattern is the "fully connected", where each neuron in the layer i is connected with each neuron at layer $i + 1$; it is important to notice that each edge has its weight, which is a parameter that will be adjusted during the training phase. A greater number of edges will cause a greater number of parameters to train and the performance of the entire network can slow down. Usually, there's one more parameter in each neuron, called bias: the idea behind the bias is to add in every linear combination describing the output of the given neuron a value that it does not depend on previous neurons, but it's constant; since the bias value is fixed, the only parameter to adjust is its weight.

By connecting a layer with l neurons and a bias to another layer of n neurons, the parameters concerning this particular part of the network will be $(l+1) \times n$, so the complexity grows easily.

2.4 Training

This is the main phase for the creation of the model. It uses a training set, i.e. a set of data where each sample is made by the data in itself and a ground truth value, which is the information that the model will need to predict. The ground truth will be used to understand how much the current prediction done on its relative sample of data is far from the expected value, helping the network to learn from its errors. The phase of training is primarily divided into two distinct moments: forward propagation and backward propagation.

- **forward propagation**: the input data from the training set are fed to the network; it is then propagated to the end and an output is produced. This output is compared to the ground truth, generating an error defined by a disparity or loss function;
- **backward propagation**: the error is then propagated in the opposite direction in order to correct the weight values and refine the final accuracy.

This procedure is repeated for the whole training set, trying to adjust the parameters of the model for every data in it.

2.4.1 Learning

The learning phase adjusts the weights of the model. At the beginning, the weighs are initialised to an arbitrary value, e.g. 1. Then the dataset is split into training, validation and test set. The actual training comes when the training set is processed by the network, and its output is compared to the ground truth of the data. An analysis on the error generated in this comparison will lead to an adjustment of the weights of every edge to improve its accuracy.

The error measured by the loss function, which is a metric that parameterize the distance between the network output and the ground truth value. Some of the most-widely used metrics are:

- **Mean Square Error (MSE)**: it is the average of the squared difference between predictions and values obtained. With this function, outliers (observations that are very far from their actual values) are penalized, because their big difference from the truth is squared.

$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n} \tag{2.6}$$

- **Mean Absolute Error**: it is very similar to the MSE, but the difference between observation and the ground truth depends on the absolute value. This function is slightly more complicated than the MSE while calculating its gradient, and the outliers are not so penalized.

$$MAE = \frac{\sum_{i=1}^{n}|y_i - \hat{y}_i|}{n} \tag{2.7}$$

- **Mean Bias Error**: this function is not so common, and it is very similar to the MAE.

$$MBE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)}{n} \tag{2.8}$$

  There could be both positive and negative errors; the MBE is not very accurate, but it is useful to describe the trend of the training, if the bias is positive or negative.

Loss functions can, of course, be customized according to the problem analysed: further in this work an example of custom loss function will be used and explained.

There are also some parameters, called *hyperparameters*, that cannot be trained, but are defined at the beginning of the training phase and they can determine how well the entire process will perform [3]:

- Number of hidden layers: if there are many hidden layers in the network, then the accuracy can increase, but the computation can be slower. On the other hand, if there are few hidden layers, the model will not have enough parameters to train and get a good approximation of the data, so the training will fail. In other words, it will underfit. Figure 2.2 shows an example of neural network with more than one hidden layer.

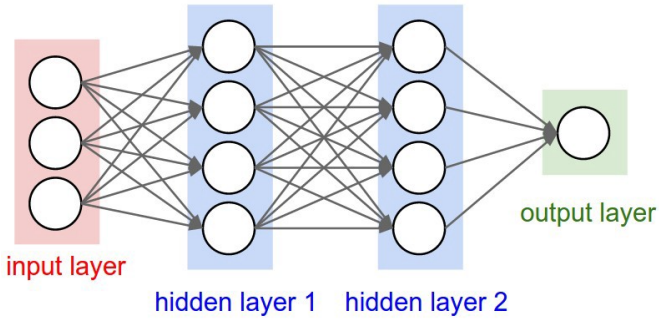


Figure 2.2: An example of neural network with more than a single hidden layer [4].

- Dropout: it is used to avoid overfitting. It consists in ignoring a fixed number of neurons from each layer while training. This will keep the processing power of the network intact while avoiding overfitting it on a specific training set. In figure 2.3 it is possible to see how dropout works.

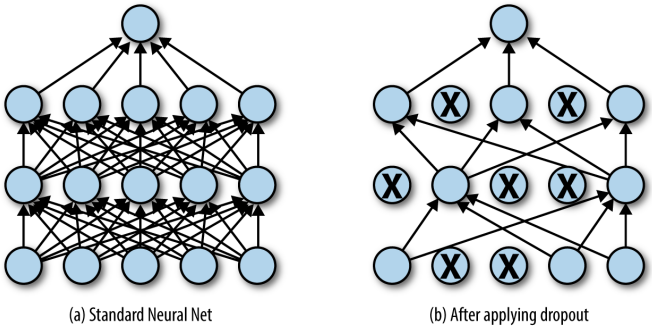


Figure 2.3: How dropout works [5].

- Activation function: it introduces nonlinearity in the model. In particular, the rectifier is the most popular and the sigmoid is the one for binary predictions. While doing multi-class predictions, the softmax function applied to the output layer is the best choice.

- Weight initialization: how the weights on the edges are initialized, normally done accordingly to the activation function of each layer. The most used distribution is the uniform one.

- Learning rate: it defines how quickly the network updates its parameters. If this value is low the entire process is slower, but the overall algorithm arrives at its optimum value smoothly. If the value is larger, the algorithm is faster but it could not converge: this difference is shown in figure 2.4.

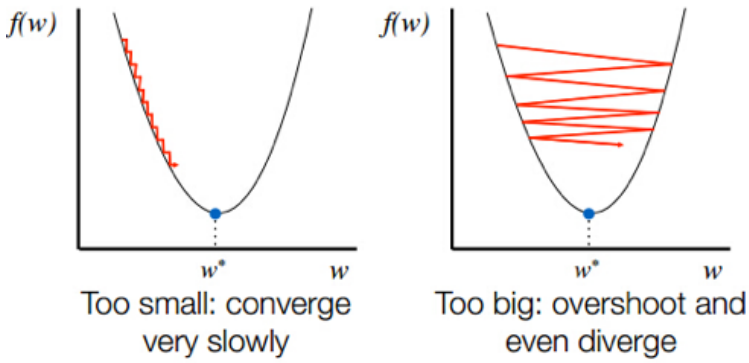


Figure 2.4: Differences between learning rates [6].

- Number of epochs: an "epoch" occurs when the entire training set is processed through the network. Therefore, the number of epochs sets how many times this entire set is used in the training phase.

- Batch size: it sets how many samples feed to the network before changing the parameters. A higher batch size will speed up the training algorithm, because the new parameters are calculated less often, but it will be less accurate.

2.4.2 Backpropagation

Several optimisation techniques can be used to adjust the parameters of the network. The first possible way is a brute-force algorithm, which means that every possible combination of values is tried for the edges of the network. Of course, there are limitations to this approach: if the weights are assumed to be values between two given thresholds, let's say $-l$ and l, with a

step between each possible value of ϵ, each weight will have to be tried $\sim \frac{2 \cdot l}{\epsilon}$ times to cover all possibilities. This amount of calculation is required for every edge and at the end, the resulting weights are those who minimize the loss. On these terms, the brute-force approach appears to be suitable only for small networks, with a few parameters to train, and the final result is not very precise. Talking about a normal fully connected network, with thousands of nodes, the edges will be so much that the brute-force method is infeasible due to the computational power and the overall time to compute the final result.

Mathematics comes to help because it is known that the derivative of a given function in a certain point is the rate at which the function is changing value in that point. It's possible to apply this concept to the loss function, to understand how it behaves as the internal weights change at a given rate.

What is interesting is the rate of which the error changes relative to the changes in the weights. For example, it could happen that if the weights change a little, e.g. $newW = W + \epsilon$, the overall sum of the errors can be several times greater than ϵ. So this method can be applied also to decrease the error, by decreasing the values of the weights. Therefore, the overall learning process is [7]:

1. Check the derivative of the loss function to see how much the error is increasing/decreasing.

2. If the value is positive, it means that if the weights are increased, the error is also increasing. The solution is to decrease the weights.

3. If the value is negative, it means that if the weights are increasing, the error is decreasing. The best thing to do is to increase the weights.

4. If the value is 0, the stable point is reached, and the algorithm is over

In figure 2.5 how the weights change.

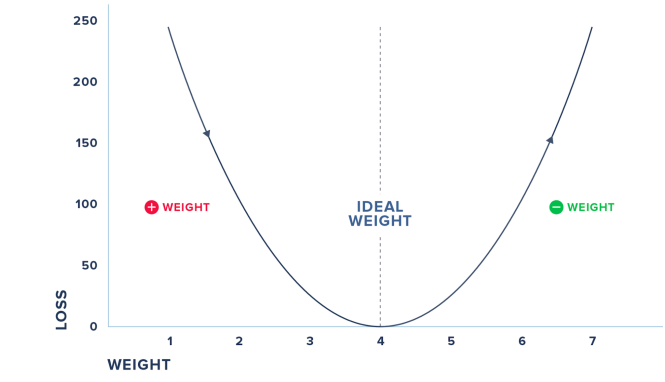


Figure 2.5: How to change the weights with regards to the sign of the gradient [7].

So, the goal of the backpropagation algorithm [7] is to change the weights inside the network such that the next results are closer to their target value.

This algorithm is briefly described as follows: it works by looking at the error generated by the feed-forward of a particular input and trying to understand how much a change in a particular edge (connected to the output) will affect the total error. This estimation is done through the partial derivative of the total error concerning the considered weight.

$$\frac{\partial C}{\partial w_i} \tag{2.9}$$

Explaining how this rate is transferred from the final error to the desired weight is possible through the chain rule:

$$\frac{\partial C}{\partial w_i} = \frac{\partial C}{\partial z_j} \cdot \frac{\partial z_j}{\partial a_j} \cdot \frac{\partial a_j}{\partial w_i} \tag{2.10}$$

where a_j is the linear combination of the nodes as input to node j, and z_j is this value fed to the activation function, e.g. the output of the network in neuron j.

The single error of neuron j is defined as δ, in particular:

$$\delta = \frac{\partial C}{\partial z_j} \tag{2.11}$$

This value is then multiplied by a factor η, which is the learning rate, and the final quantity has to be subtracted from the weight that is considered. This entire procedure is done for every weight connected to the layer, in this case, the output.
For hidden layers, the process is similar, but need to take into account the fact that now every neuron in this layer contributes to the next one.

$$\frac{\partial C}{\partial z_i} = \frac{\partial C_1}{\partial z_i} + \frac{\partial C_2}{\partial z_i} \tag{2.12}$$

After having done the calculations for all the neurons in the same layer, the procedure is repeated for every layer in a backwards direction, until the input layer is reached. At this point, all the weights can be updated with their newly found value and other samples are fed to the network as inputs; then the algorithm is called again until the last batch of data is treated.

2.4.3 Problems

Some challenges could arise while optimizing the network during the training phase [8]:

- Ill-conditioning: this issue appears if the loss function has a steep derivative, i.e. even a small change in the input or weight increases the cost function. One consequence of this problem is that learning turns out to be very slow because the learning rate must be reduced to face a strong curvature. Newton's method is very useful to optimize convex functions, but it cannot be used yet since it must be adapted to the neural network context;

- Local minima: looking for a minimum in a generic function with an iterative procedure can lead the training algorithm to settle on a local minimum, which can be very different from the lowest achievable value. If the values are not so different, falling into a local minimum is not such a great problem for the overall learning result; otherwise, the trained network will have suboptimal accuracies.

- Flat regions: saddle points are zero gradient regions that can be the result of the optimization algorithm. In particular, in high-dimensional spaces, saddle points are more common than local minima. It seems that the standard optimization strategies stop their iterative learning process while facing saddle points.

- Cliffs: while on a steep region, the gradient could make a huge step and jump off the entire cliff structure. This is solved by the gradient clipping, that reduces the update step according to the size of the gradient: bigger gradient means a small step, and vice-versa;

- Long-term dependencies: they happen when the same operation is repeated through various steps. To make an example, repeating the multiplication with a particular weights matrix can lead to the exploding gradient problem, which generates very steep surfaces, like cliffs;

- Inexact gradients: usually, the optimization algorithms assume that the computed gradient is correct, but in practice, this value is noisy or biased, so the gradient is not perfect. The problem can be avoided by accounting for imperfections in the gradient while designing the algorithm or by choosing a loss function that is easier to approximate.

2.5 Validation

The validation is an optional phase of the learning process of a network, where a new set of data, called validation set, is used to decide when to stop the training phase. The idea is to calculate an estimation of the overall loss of each network with this set that is not part of the training set. The validation set is used to simulate a test set, by calculating a validation loss on the input data. The backpropagation algorithm can find the optimal parameters of a network

by adjusting the weights of the edges in the model, but this can lead the network to a situation where the result keeps floating around its optimal value. Validation has been introduced to avoid this loss of precision by keeping under control when the training procedure would generate a good test loss and stopping the training algorithm at its optimum. Usually, if after 10 epochs of training the validation loss has not improved, the learning procedure is interrupted and the network is taken as it is.

The validation, due to its ability to emulate the behaviour of a test set, is also used to understand which one, between different configurations of the model, is the best to fit the data.

2.6 Test

Once various models have been trained, and the best one has been identified through validation, it is possible to assess the performance of the fully specified model.

The test set used is a set of data that have to be extracted from the same distribution as the training and validation set, to ensure consistency. If the model fits both the training set and the test set similarly, it means that no overfitting occurred. In case the test accuracy is lower than the training accuracy the generated model does not generalize the task it has been trained for.

2.7 Overfitting/Underfitting

Statistically speaking, overfitting occurs when the trained model proves to be tailored very closely or in an almost exact way to a particular set of data, i.e., the training data[9]. This implies that further sets can not be reliably predicted by this model.

Underfitting is the opposite of overfitting. In this case, the model is not able to adequately

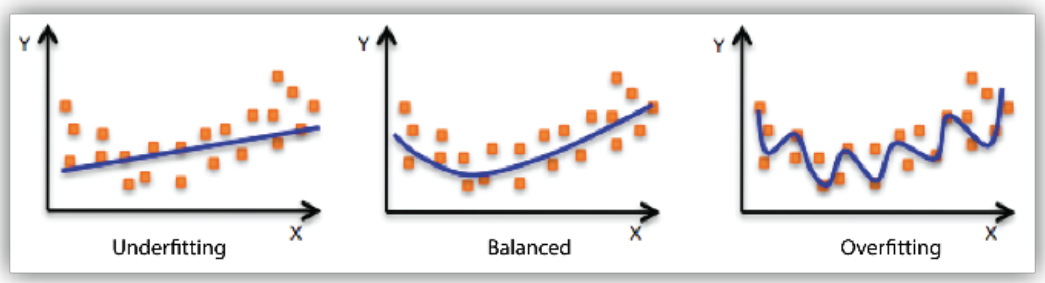


Figure 2.6: Overfitting vs underfitting [10].

capture the inner structure of the data, bringing poor performance even on training data[9]. This problem can be related to the structure of the model itself; in figure 2.6 a comparison between overfitting, underfitting and a correct balanced model.

3

Deep neural networks

A neural network is a data processing architecture which can implement an unknown function. This is made possible by the training phase, in which the network learns how to process data and provide an answer which is as correct as possible with respect to the ground truth values.
The Universal Approximation Theorem [11] states that:

"A feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of R^n"

This theorem essentially says that a network with a single hidden layer can approximate in a good way every single continuous function in the real space of any dimension [12]. In this way, every continuous function can be disassembled in its minimal parts. The weights on the edges will care about how much that particular neuron value counts for the final result. As reported

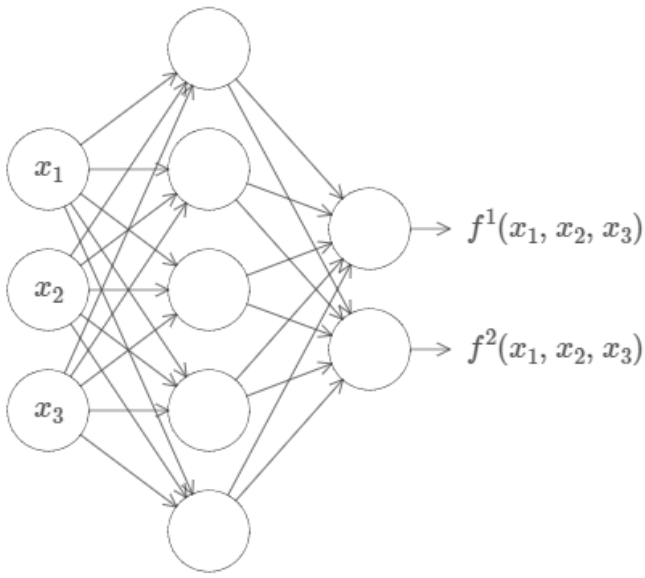


Figure 3.1: Example of a fully-connected neural network with multiple outputs [12].

in the previous, weight values are learned during the training phase. These weights will then contribute to the final layer, with one or more output neurons; in figure 3.1, how multiple outputs are organized. But there are two caveats to face while declaring this statement: a neural network with a single hidden layer can only learn *continuous* functions and they are only *approximate* versions of themselves.

If a function is *discontinuous*, it is generally not possible to represent or approximate it with a network of this kind. This is a not so surprising fact because usually, the functions that a network is called to compute are continuous. In this case, the network will try to learn a continuous approximation, which normally is enough to represent it with a good accuracy [12].

Another limitation is the fact that the final result is an approximated representation. The accuracy is directly connected to the number of neurons in the hidden layers. If there are few neurons, the network will not be able to understand the relational structure between input and output; if there are more of them, more nuances can be caught by the layer and the approximation will be better.

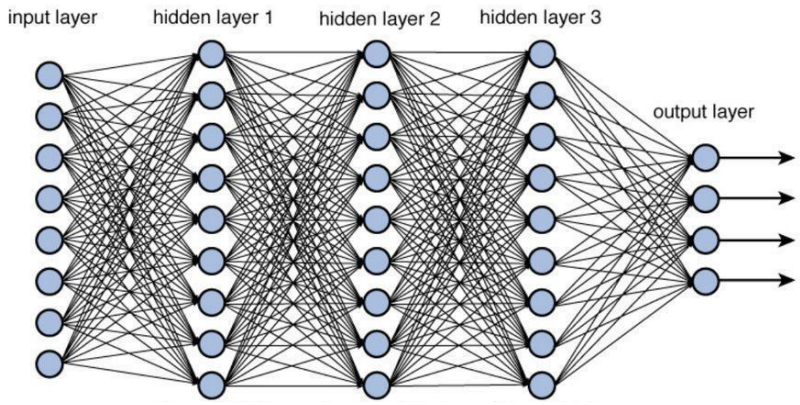


Figure 3.2: An example of a deep fully-connected neural network [13].

This fact leads to a major drawback: the total amount of neurons required can be exponentially large. In particular, the more complex is the needed function, more neurons there will be in the layer. Since every neuron is connected to all the other ones in the following layer, if the number of neurons grows, the total complexity, which is defined by the size of the set of the neurons and the size of the set of the edges, grows in a very steep way.

To overcome these problems, the most adopted strategy is to add one or more hidden layers. This allows the network to have fewer neurons per hidden layer, fastening the feed-forward phase of the algorithm and increasing the total number of edges. This can bring a better approximation of the function that is been considered. The overall complexity of the system grows, as it can be seen in figure 3.2, but this allows to compute a wider family of functions, both linear and non-linear. In general, a neural network is defined as "deep" if there are multiple hidden layers between the input and output layers.

A deep neural network can generate a compositional model of an object where it is represented as a composition of primitive data, arranged in layers. With more layers, the input can be decomposed in its primitives at a deeper level, regardless of the complexity of the object [8].

This feature comes to help when harder problems are taken into account: real-life artificial intelligence applications require a deep analysis of every observed data. To make a couple of

examples: a pixel of a certain subject in a picture can change its colour values if the photo is taken in different moments of the day, even if the context and the subject itself are the same; a registration of a phrase spoken by a person could be very different from the registration of the same phrase done by another person: there are many factors that can lead to a complete mistake, like the accent and the tone of voice [8].

Deep neural networks can overcome these problems by considering only the object itself, ignoring every misleading factor, decomposing the computation into nested simple functions, each one belonging to a different layer. The abstraction grows as the input proceed into the network, allowing an easier computation, as it is show in figure 3.3.
Besides their slightly different approach and scope, deep neural networks are very similar to standard neural networks, since the seconds are just a particular case of the firsts.

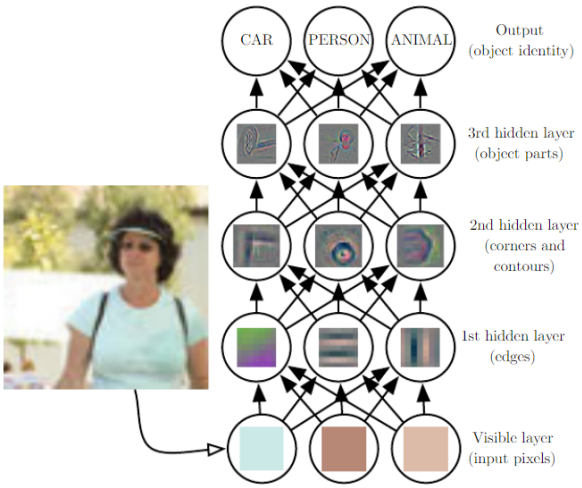


Figure 3.3: Some features captured by a neural network [8].

Many different deep networks have been created to face particular tasks, like face and speech recognition or image classification; these structures have been tested for years and at this point they are capable to fulfil their duty and reached a good level of reliability. In this work, in parallel with the standard implementation of deep neural networks, have been also considered

two alternative architectures: *convolutional neural networks* and *residual neural networks*.

3.1 Convolutional neural network

Convolutional neural networks are specialized kind of network thought to process data that has a grid-like topology. An example could be an image dataset, where each piece of data is an image, that is seen as a 2D grid of pixels. As the name says, the particularity of this architecture lays on the convolutional term: it means that at least one layer in the network implements the convolution in place of the standard matrix multiplication [8].

Matematically the convolution is an operation on two functions f and g which produces a third function expressing how the shape of the first is modified by the other one. It is defined as the integral of the product of the two functions with the second one that has been inverted and shifted.

$$(f \circledast g)(t) := \int_{-\infty}^{+\infty} f(\tau)g(t-\tau)d\tau \tag{3.1}$$

$$(f \circledast g)(t) := \sum_{k=-\infty}^{+\infty} f(k)g(t-k) \tag{3.2}$$

The first term usually is referred to as the input, while the second is the kernel. The output is sometimes called feature map [14].

While the input is usually a multidimensional array of data, the kernel is usually a multidimensional array of parameters, which will be trained and adjusted by the network.
From figure 3.4 above it can be seen that the output of the convolution between a grid of values and a kernel is just a linear combination of the values involved, which in the case of the kernel they are always the same; those who change are those about the portion of input considered in that particular iteration.
Typically, a single convolutional layer is divided into three operations or stages:

1. **Convolution stage**: in this stage, the layer performs various convolutions in parallel and produces a set of linear activations;

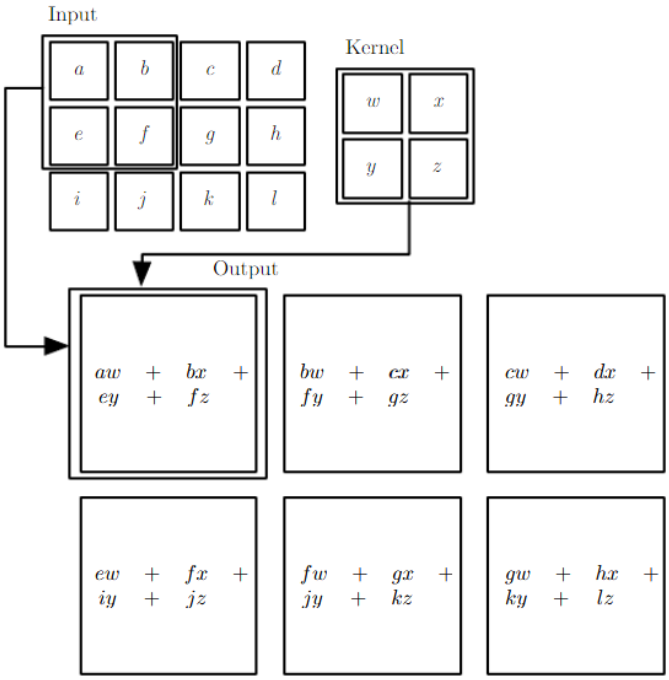


Figure 3.4: Output of a convolutional layer [8].

2. **Detector stage**: in this phase, the output set of the first one is run through a non-linear activation function, such as the ReLu;

3. **Pooling stage**: here the output of the detector stage is further modified with a pooling operation, which is a function that replaces the output of a net with a summary of the surrounding outputs;

The most common pooling functions used [15] are:

- **Max pooling**: where the final output is the maximum value in the considered set of values fed to the function. An example is shown in figure 3.5;

- **Min-pooling**: the same as max-pooling, but the value taken is the smallest one;

- **Average pooling**: here the final output, as the name suggests, is the average of all the values that the function takes as input. Figure 3.6 is a clear example;

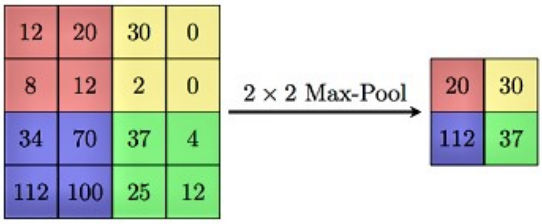


Figure 3.5: How max-pooling works [16].

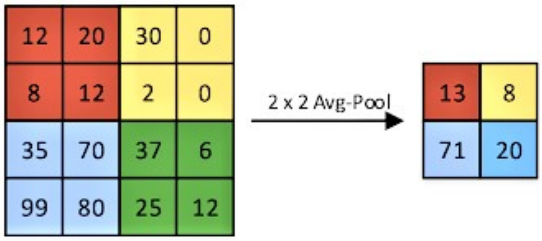


Figure 3.6: How avg-pooling works [17].

Usually, the choice of the pooling method depends on the input data: if the dataset is composed of images, by using the average pooling the final result will be an image with smoothed colours and some information about the edges will be lost; on the other hand, using the max pooling will bring out more information from darker images, and vice-versa for the min pooling.

In every case, pooling helps to make the representation invariant to small changes of the input, so the final result will be more robust [15].
This is the case of pooling over results of the same convolutions, but this operation is effective even when applied to results coming from separately parametrized convolutions.
On the example in figure 3.7 the same input is fed to three different filters, which try to match the input with a rotated version of it. A match in one of these filters will bring out a larger output than the others, so a max-pooling over the outputs of the filters will be able to obtain a large response in any case, even with slightly different inputs.

The size of each layer reduces at each stage, and this is caused by both the convolutional

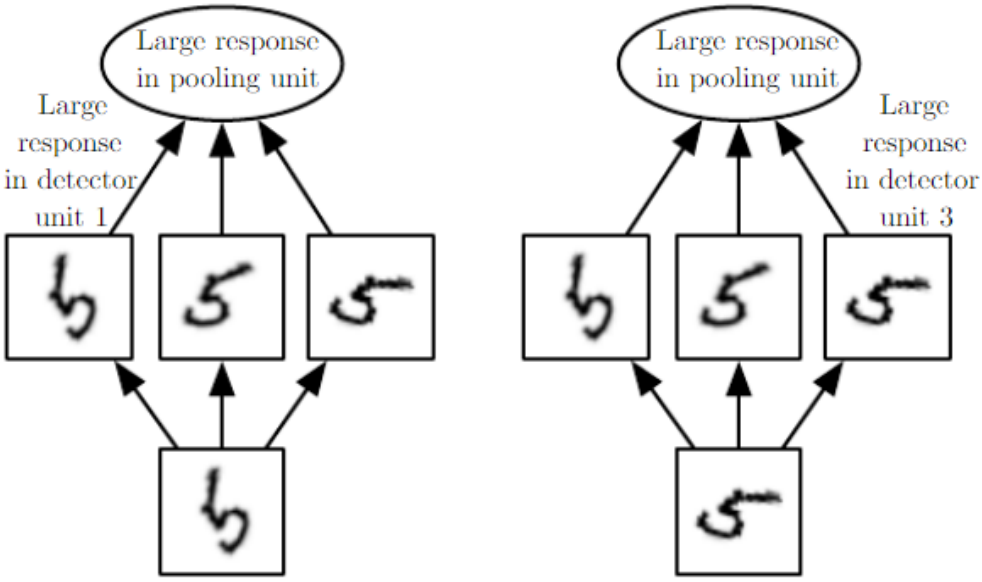


Figure 3.7: Example of learned invariances [8].

stage and the pooling stage. This happens because the filtering operation, which involves a linear combination between the values of the filter and the image, will make the involved pixels collapse into a single one, reducing the dimensionality; the further stage, the pooling with size q x q, is responsible for the elimination of some unwanted information, reducing again the size, obtaining a final output of size (N/q) x (M/q).

For example, an image of 100×80 pixels as input, after the first convolutional layer with filters 5×5 and pooling of size 2×2 will result in a feature map of size 48×38.
So this means that while going on with the computation, the entire input will collapse to a really small feature map e consequently a lot of information will be lost; the number of kernels comes to help in this situation because the further one gets in the network, the more filters there will be per layer. It is a general rule that while shrinking the size of the single feature map, more of them will be generated.
Very far away from the input layer, depending on the initial size, the feature set will become more and more similar to a layer of a fully-connected neural network. This entire structure is graphically described in figure 3.8

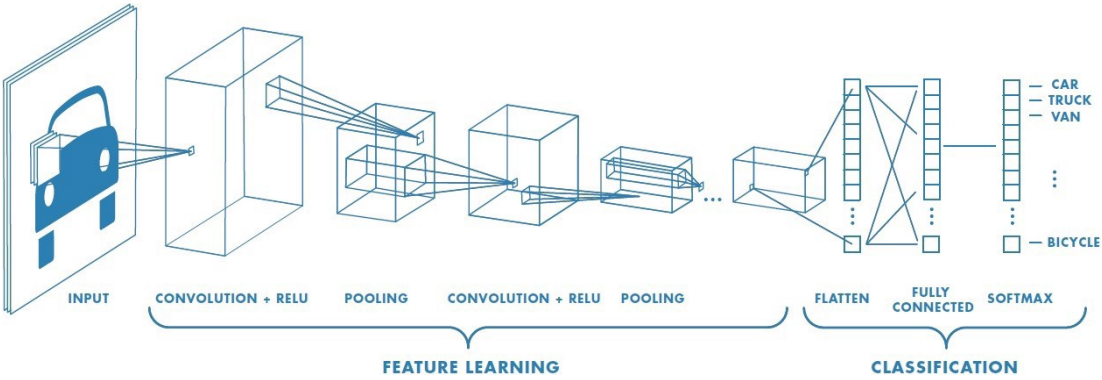


Figure 3.8: Usual structure of a convolutional neural network [18].

Generally speaking, convolutional neural networks decompose the input in various features using the convolutional operation. The features will be more and more abstract as the considered layer is far away from the input one, and few final fully-connected layers complete the model allowing this structure to perform standard machine learning tasks on grid-like datasets.

3.2 Residual neural network

Residual neural networks were born [19] to face the problem of degradation: if a model is trained to map a certain function and if some layers mapping the identity function are added, the overall accuracy of the network should be not lower than its shallow counterpart. In real cases, it can be shown that adding layers to a network leads to a degradation in performance of the overall network because the error is hard to propagate on the earlier layers; this is due to the exponential reduction of the gradient while travelling backwards the network to adjust the weights.

The idea behind ResNet is to make a network or a set of layers learn not the needed function *H(x)*, but the difference between this function and the identity function, and this quantity *f(x)* is called residual:

$$H(x) = f(x) + x \quad \text{->} \quad f(x) = H(x) - x \tag{3.3}$$

The x is simply obtained by a skip connection between the input layer and the needed one. Each one of these structures, where there are few layers in the middle of a shortcut connection between the first and the last, is called Block [20], and an example of it is in figure 3.9.

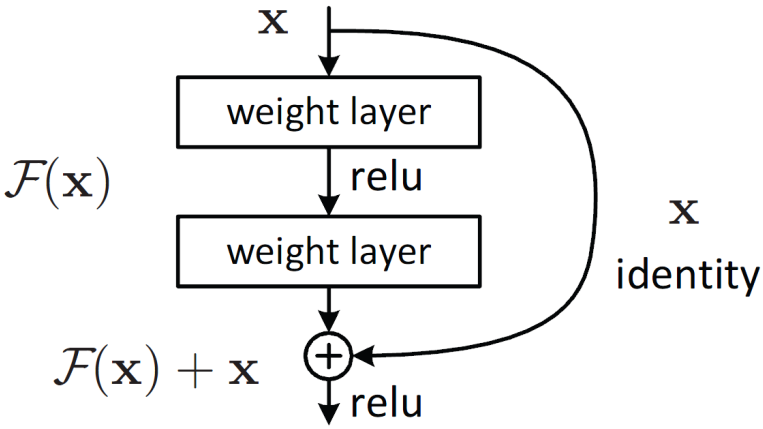


Figure 3.9: Example of a single block of a residual neural network [19] [20] [21].

As can be seen from the above figure, the needed input x is given to a series of layers which can approximate $f(x)$, called residual, and add at the output of this function the input, in an element-wise operation.

Usually, the input and the output of the layers have the same size, to ease the operation of adding, but in the cases in which this do not happen, zero-padding techniques or 1×1 convolutions are used to adapt the dimensions. The learning phases and the backpropagation of the error seen in the fully-connected approach are then applicable here without any problem. About the model that has been described earlier in this paragraph, adding one or more block instead of the usual layers, will maintain the accuracy of the shallow network without increasing the error: this happens because it is easier to force a function like $f(x)$ described by the layers in the block, to remain simple, because the skip connection will exclude some of its overall complexity by bringing over the function $f(x) = x$ [20]. In this way, ResNet gives the layers a reference point to start mapping the identity function, which is the shortcut connection x, instead of let them create a new identity function for each block.

The idea behind ResNet, (see figure 3.10), was to increase the number of hidden layers (with

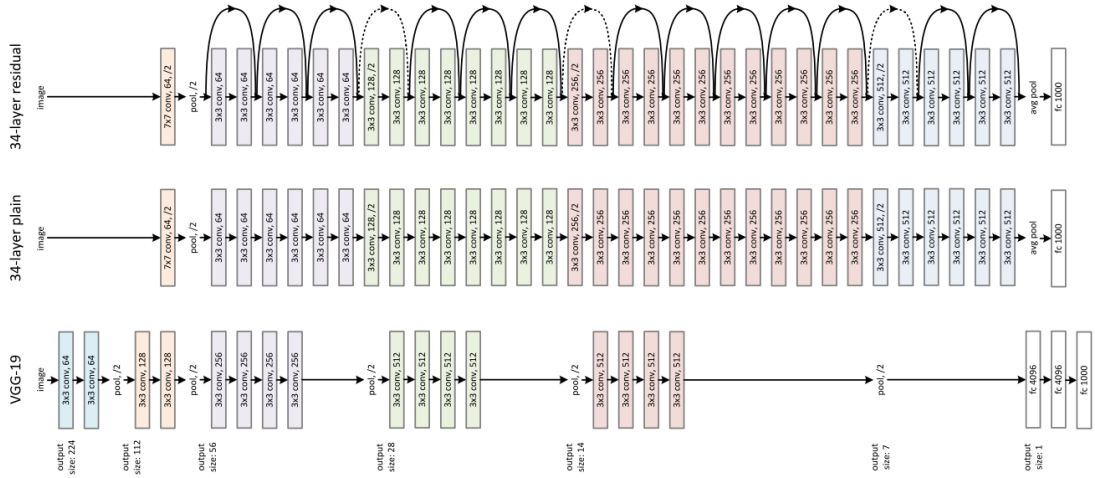


Figure 3.10: First ResNet architecture [21].

respect to a fully-connected approach) at a feasible computational cost. The neural network with the biggest number of layers is a ResNet with more than a thousand layers, while the biggest number of layers for a fully-connected neural network is still less than thirty [20]. The thousand-layer model has been proved to be able to generate a training error $< 0.1\%$, while keeping the test error to a good 7.93% [21].

3.3 Autoencoder

There are a lot of different neural network architectures. Some are the best while treating images, others that can be used to learn accurately complex functions more accurately. One of them is the autoencoder, a neural way to compress data [8]. This kind of neural network can be related to both compression and encoding: they were developed to take a vector from R^n and translate it into an *m-dimensional* representation, with $m < n$.

This neural architecture learns how to extract the most important features of the input and build a smaller, yet complete, representation of it. Then, the inverse process is possible, taking the compressed code and obtaining the original data after having applied a decoding function.

The simplest autoencoder can be modelled by a neural network with just three layers [8]:

1. Input: this layer, as usual, takes the desired data and feeds it into the network;

2. Code: this is the layer of the code, where the reduced representation of the input sample will be shown. Its size is the length of the encoding of the data;

3. Output: in this architecture, the size of the output must be the same as the size of the input, to allow the network to understand if the entire model caught the reduction and the following expansion of the features in a reliable way;

These phases can be grouped into two main operations, visible also in figure 3.11:

- **Encoder**: the first and the second layer, represent an encoding function from R^n to R^m;

- **Decoder**: the second and the third and last layers implement a decoding function from R^m to R^n, to get the original data back from the coded representation;

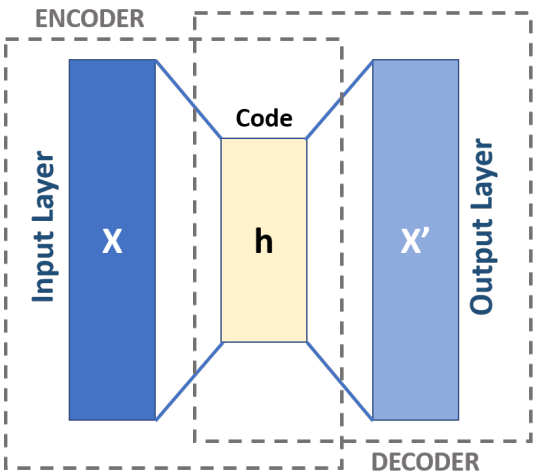


Figure 3.11: The organization of the layers in a simple autoencoder [22].

This means that autoencoders do not need to have a labelled datasets to learn because their loss function describes how much the obtained output is similar to the input that generated it.

3.4 Mathematical approach

The two parts of a simple fully-connected autoencoder, the encoding and the decoding parts, are, respectively, two functions ϕ and ψ, such that

$$\phi : X \to F \tag{3.4}$$

$$\psi : F \to X \tag{3.5}$$

with

$$\phi, \psi = \arg\min_{\phi,\psi} \ \|X - (\psi \circ \phi)X\|^2 \tag{3.6}$$

3.4.1 Encoder

The encoder phase takes the input $x \in R^n$ and codes it into $z \in R^m$:

$$z = \sigma(Wx + b) \tag{3.7}$$

with W as the weight matrix of the encoder, b as the bias and σ as the activation function.

3.4.2 Decoder

The decoder takes in input $z \in R^m$ and tries to decode it back to $x \in R^n$:

$$x' = \sigma'(W'z + b') \tag{3.8}$$

where W' is the weight matrix of the decoder, b' is the bias, ans σ' its activation function. Subsequently, the algorithm has to minimize a function of this shape:

$$f(x, x') = \|x - x'\|^2 = \|x - \sigma'(W'(\sigma(Wx + b)) + b')\|^2 \tag{3.9}$$

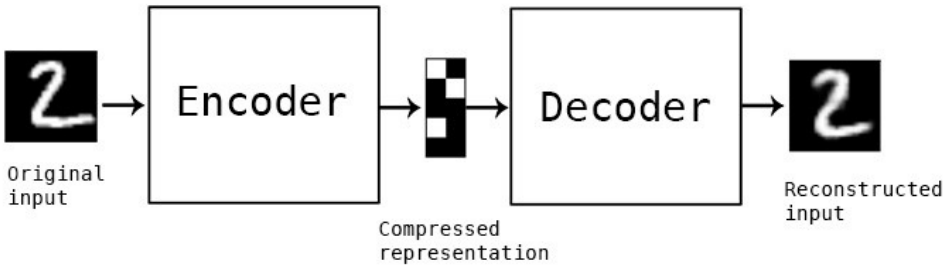


Figure 3.12: A sample application of an autoencoder with MNIST data [23].

3.5 Regularized autoencoders

There are autoencoders where the number of neurons in the hidden layer is the same or even bigger than the size of the original data. These borderline cases can lead the model to learn the representation of hidden feature structures, but in the most of the cases, the learned function includes the identity. Various techniques has been developed to avoid this phenomenon, and the architectures resulting are knows as *regularized autoencoders* [24] [22].

3.5.1 Sparse autoencoders

Sparse autoencoders have more neurons in the hidden layers than in the input one. However, these neurons are not all active at the same time, only few of them per iteration can be triggered. This technique can make the model learn all the possible features of the input dataset in the hidden layer, without having all of them involved in the analysis of a single input [24] [25] [26].

In more detail, this sparsity constraint is a "penalty" which is added to the loss function and depends on the hidden layer; then this sum is what will be minimized in the training phase:

$$f(x, x') + \Omega(z) \tag{3.10}$$

A neuron is "active" when its output is close to the maximum value allowed by the activation function, and "inactive" in the other case. In the following explanation, the assumed activation function is the *sigmoid*, with maximum output 1 and minimum output 0. Let a_j be the activation factor the j-th neuron in the hidden layer and x_i the input neuron connected to the considered hidden unit. The average activation over m samples of the hidden unit j is defined as follows:

$$\hat{\rho}_j = \frac{1}{n} \sum_{i=1}^{n} (a_j x_i) \tag{3.11}$$

where $\hat{\rho}_j$ has to be forced to be close to 0, which means that the most of the neurons is inactive; therefore, given the "sparsity parameter" ρ, which is a constant that defines the percentage of contemporary active neurons for a given input, the method forces $\hat{\rho}_j = \rho$ [26].

So, the penalty will penalize $\hat{\rho}_j$ when it deviates significantly from ρ; this is done with the Kullback-Leibler divergence:

$$KL(p_x || p_y) = p_x log \frac{p_x}{p_y} + (1 - p_x) log \frac{1 - p_x}{1 - p_y} \tag{3.12}$$

for generic p_x and p_y probability mass distributions, defined in the same probability space. This quantity is the indication of how much the random variable p_x is related to the random variable p_y. It's some sort of asymmetric distance function, since it's close to 0 if the two random variables have similar distributions, and it's positive and greater than 0 if the two distributions are very different; in particular, the more uncorrelated the two distributions are, the bigger will be their divergence value [26].

In this case, the KL divergence needs to be summed all over the hidden units in layer 2:

$$\sum_{j=1}^{m} KL(\rho||\hat{\rho}_j) = \sum_{j=1}^{m} \rho log \frac{\rho}{\hat{\rho}_j} + (1-\rho) log \frac{1-\rho}{1-\hat{\rho}_j} \tag{3.13}$$

With this penalty function, if $\hat{\rho}_j = \rho$, $KL(\rho||\hat{\rho}_j) = 0$, otherwise the divergence is greater than 0. In the example below, ρ has been set equal to 0.2, and the plot of $KL(\rho||\hat{\rho}_j)$ for different values of $\hat{\rho}_j$ is in figure 3.13. As can be seen in figure 3.13, when $\hat{\rho}_j = 0.2$, the divergence is 0, and the network will learn to activate certain hidden neurons concerning the input [25].

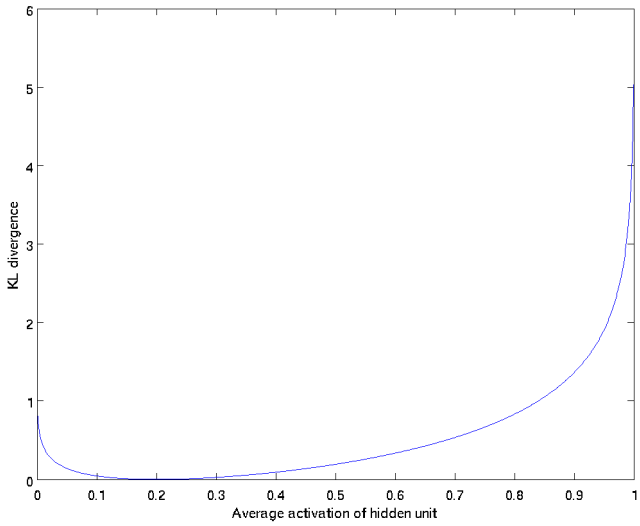


Figure 3.13: KL divergence as $\hat{\rho}_j$ varies [26].

3.5.2 Denoising autoencoders

Denoising autoencoders, graphically described in figure 3.14, belong to a specific architecture which task is to remove the noise that affects an image to obtain a cleansed version. The main differences between this approach and the others are:

- The input of the training phase is made of partially corrupted data;

- No compact representation is required, but the reconstructing strategy is changed.

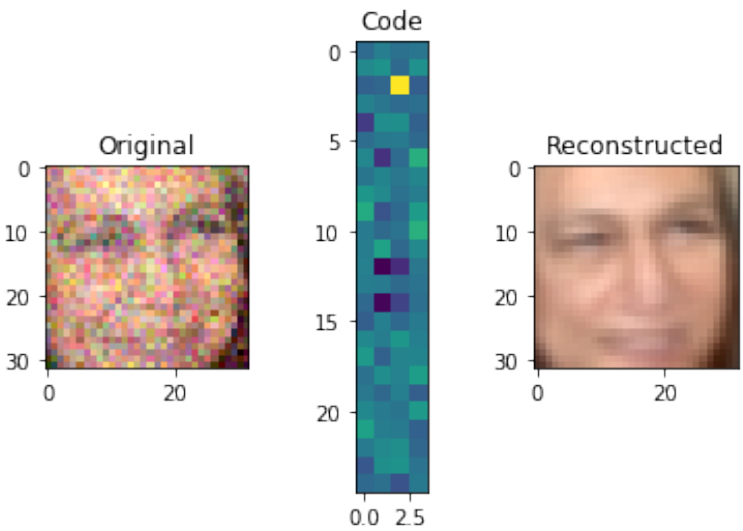


Figure 3.14: A visual description of a denoising autoencoder [27].

In particular, this approach is called to a representation of the corrupted image in the encoder phase. The input can be, for example, corrupted by a Gaussian noise:

$$\hat{x} = x + \mathcal{N}(\mu, 0) \tag{3.14}$$

and the corresponding hidden layer can be simply represented as

$$z = \sigma(W\hat{x} + b) \tag{3.15}$$

Then, the decode phase or reconstruction aims to minimize the difference between the output of the network and the original, uncorrupted input:

$$\widetilde{x} = x \tag{3.16}$$

with

$$\widetilde{x} = \sigma(Wz + b) \tag{3.17}$$

The final result of a successful training phase is a complete network that takes in input a normal signal and produces its denoised version, as figure 3.15 shows.

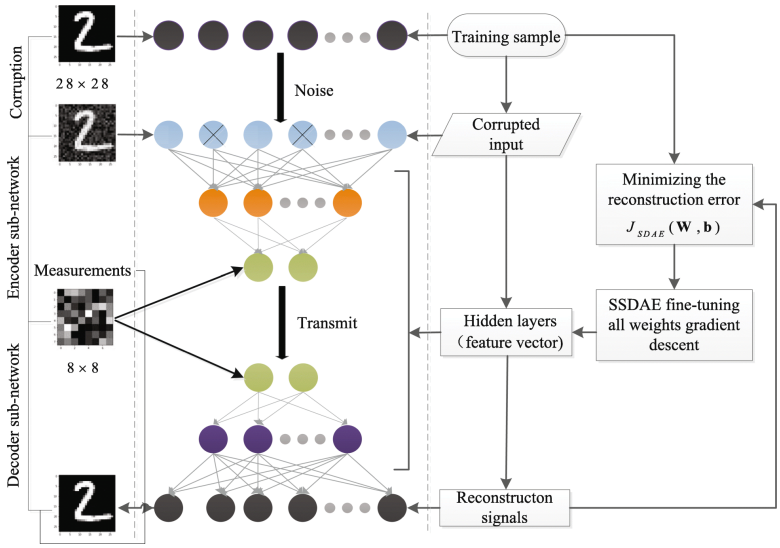


Figure 3.15: Briefly, how a DAE works [28].

3.5.3 Contractive autoencoders

Contractive autoencoders are neural networks that force the algorithm to learn a robust function [29]. They're quite similar to sparse autoencoders in their working since they try to minimize the loss function plus a determined value.

This value to be added to the loss is a factor that depends on the norm of the partial derivative matrix with respect of the input, known as *Jacobian*. The final factor to be added is then:

$$\Omega(z, x) = \lambda \sum_i \|\nabla_x z_i\|^2 \tag{3.18}$$

This penalty is applied to training input only, so the trainer learns useful features about the training distribution. The symbol ∇_x is the partial derivative matrix concerning input, and

this means that the model is forced to penalize high gradient situations, which corresponds to a high value of the loss, in the representation z of the data x.

3.6 Applications of autoencoders

Autoencoders are simple to understand and useful for many applications. It's possible to build them in every shape, and one can add all the needed hidden layers. Over the years, various applications have been discovered to be easy to implement on this kind of neural network, so it's possible to create a fine model which has a great accuracy while performing its task [30]. Some of these applications are:

- **Image colouring**: by understanding which colour is normally associated to a specific shape, it's possible for an autoencoder which learns with black/white images and their colourized version to take a greyscale image and apply to it a plausible colour palette. A example in figure 3.16;

Figure 3.16: Example of a b/w image colored with an autoencoder. The real image is in the middle [30].

- **Dimensionality reduction**: instead of using Principal Component Analysis (PCA), a valid technique based on the variance along the dimensions of representation of the data, it's possible to configure an autoencoder to make it learn the patter of the most important features in a piece of data, and use them to rebuild the initial data in the more accurate way possible. This is visible in figure 3.17;

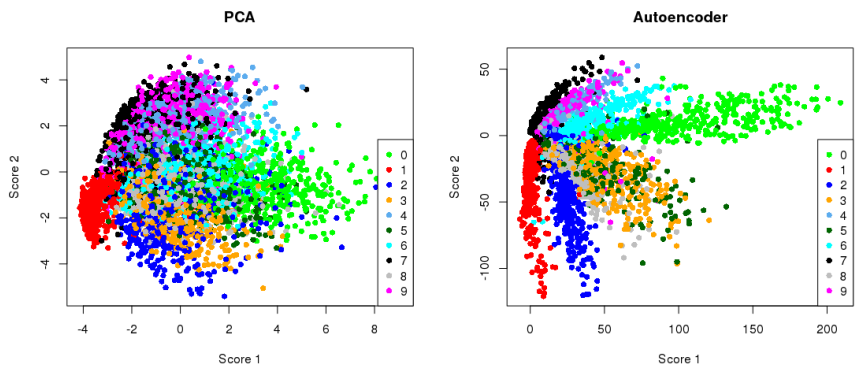


Figure 3.17: Example of feature reduction in a dataset [30].

- **Watermark removal**: watermarks are generally symbols that are added to data to prevent their usage in an unauthorized way. They can be removed by making an autoencoder learn their pattern and then extracting it like in figure 3.18 from the desired input;

Figure 3.18: The watermark on the left is successfully removed from the middle image. The result is on the right [30].

Of course, there's continuous research about the possibilities of this neural architecture. These stated above are just examples, but there are many applications of this kind of network, making it one of the most versatile neural network on the scene.

4

Minimum Spanning Tree

4.1 Graph notation

The definition of a minimum spanning tree, called in this work MST for the sake of conciseness, directly derives from a graph notation. A graph is a discrete mathematical structure composed of various nodes or vertices that are connected between each other with a certain number of edges. Formally, say that a graph is an ordered couple $G = (V, E)$, where V and E are, respectively, the set of the nodes and the set of the edges, such that $E \subseteq V \times V$: an element of E is a couple of elements of V. The graph can be *oriented* or *non-oriented*: in the first case E is an asymmetric relation, in the second one it is symmetric. If the graph is complete, it means that every node is connected with a distinct edge to every single other node [31].

Given an oriented graph $G = (V, E)$, with a cost for every edge called *weight*, if $w(u, v) \in$

R^+ is a weight function for G, the MST for G is a set of edges such that:

$$w(T) = \sum_{(u,v)\in T} w(u,v) \tag{4.1}$$

which is the solution of the following problem:

$$\begin{cases} min \sum_{(i,j)\in E} w_{ij}x_{ij} \\ \sum_{(i,j)\in E} x_{ij} = |V| - 1 \\ \sum_{i\in E, j\notin E} x_{i,j} + \sum_{i\notin E, j\in E} x_{i,j} \geq 1 \end{cases} \tag{4.2}$$

where w_{ij} is the weight of the edge going from i to j, and x_{ij} is equal to 1 if the edge (i,j) is taken in the MST.

1. The first row is just a cost function which describes the sum of the edges chosen in the MST;

2. The second constraint describes that the number of edges must be exactly the number of vertices minus one because, by the tree property, a tree with n nodes has exactly $n-1$ edges;

3. The last one says that there must not be cycles. This expression is described using the concept of *cut*, which is a partition of the graph into two disjoint subsets.

There are of course more than one minimum spanning tree that can be obtained from the same graph, like is show in figure 4.1; if the overall sum of chosen edges is the same it means, of course, that there are more possibilities to obtain a minimum, and this means that all the solutions with the same cost are valid.

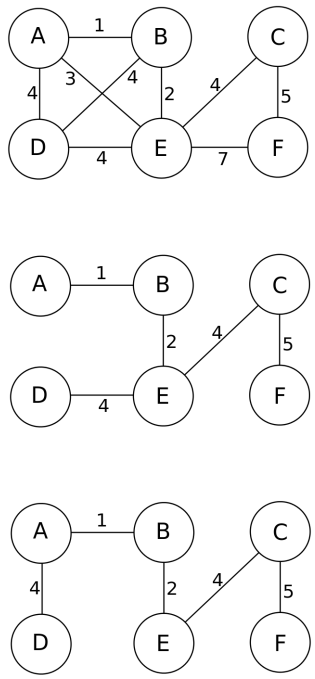


Figure 4.1: The two bottom graphs are two possible MSTs for the graph at the top. They have the same overall cost [31].

4.2 Properties

Minimum Spanning Trees have some important properties [31], like:

- Multiplicity. As already said, there is the possibility that the same graph has several MSTs.

- Uniqueness. This is the opposite of the previous statement, but in certain situations, there is only one minimum spanning tree in a graph. This happens when all the edges have distinct weights; if there are edges that have the same value, the graph falls in the other case.

- Minimum-cost subgraph. If there are only non-negative weights, the minimum spanning tree is, in fact, the subgraph with the minimum overall cost, because graphs with

cycles surely have a bigger overall weight.

- Cycle property. Given any cycle in the graph, if the weight of one of its edges is greater than the weights of all the others, than this edge cannot belong to an MST. An example in figure 4.2.

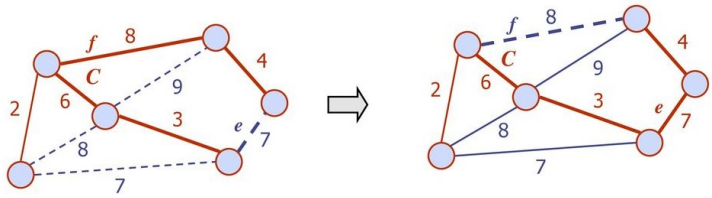


Figure 4.2: In the figure, the only edge that cannot belong to an MST is the one with weight 8, because is the largest value between all the weights in the cycle [32].

- Cut property. Defined a cut as a partition of nodes dividing a graph in two subgraphs, a *cut-set* is the set of all edges that connect two nodes that belong to different subgraphs. For any cut in the graph, if there's an edge in its cut-set with a weight that is strictly smaller than the weights of the other edges belonging to the same cut-set, then this edge is part of every MST of the graph. Figure 4.3 shows an example of cut-set.

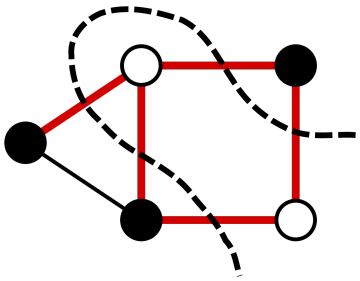


Figure 4.3: This is a simple graph with a cut; the edges in the cut-set are those which connect a black node to a white node, and vice-versa [33].

- Minimum-cost edge. If in a graph there's a unique edge with the minimum-cost, then this edge belongs to every minimum spanning tree.

- Contraction. If in a graph there's a minimum spanning tree, a subtree of it can be collapsed to a single node and the MST of the initial graph is the sum of the value of single vertex plus the MST of the collapsed graph.

4.3 Algorithms

Various algorithms have been developed to find a minimum spanning tree given a graph [31]. One of the first of them to be published is the Borůvka algorithm. After that, more algorithm were discussed, such as Prim's and Kruskal. All the algorithms, except the brute-force approach, are greedy, which means that they build the final solution with locally correct smaller solutions, exact, so they give a solution which is not approximated or probabilistic, optimal because the result they propose is the best they can get.

4.3.1 Brute-force algorithm

The first algorithm that can be hypothesized, for every mathematical or informatic problem, is a brute-force algorithm [34], which in cases like this consists into considering all the possible combinations and then choosing the ones which satisfy the initial condition. In particular, for the minimum spanning tree problem on a complete graph with n nodes, the brute-force approach is as follows:

1. Consider all possible trees in the given graph.

2. Among all the trees, find the one with exactly n nodes with the minimum sum of the weights.

The main issue with this types of algorithms is the computational load. Cayley, in 1889 [35], discovered that given a complete graph of $n \geq 2$ nodes, the number of all possible

spanning trees is:

$$f(n) = n^{n-2} \tag{4.3}$$

This means that just with 10 nodes the number of all possible trees is $10^8 = 100000000 \rightarrow$ one hundred million possibilities.

Considering that a normal graph will surely have more than 10 nodes and then after having found all the trees there's the procedure of finding the one of minimum cost, it's easy to understand that an approach like this one is too expensive in terms of time and computational power. Just to explain, in figure 4.4 it is possible to see that $n!$ is one one of the quickest

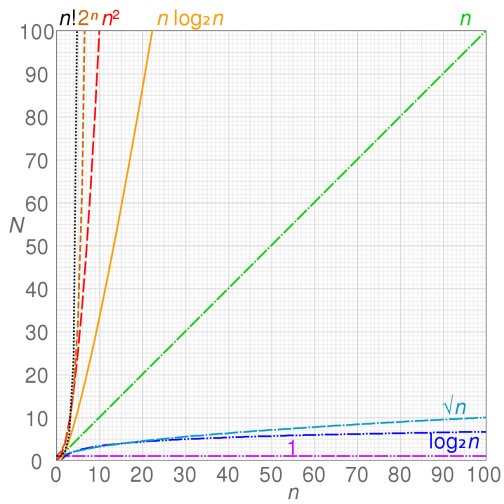


Figure 4.4: Plots of the principal functions, n^n not shown [36].

functions to diverge. The n^{n-2} curve, not displayed, goes to ∞ even faster than $n!$.

4.3.2 Borůvka algorithm

Developed in 1926 to solve a problem about electricity supply in Moravia, this algorithm [37] starts, like shown in figure 4.5, by considering a graph, with a set of nodes and a set of edges; the single node is considered as a connected component, and the first step is to add to the MST, which is empty at first, the edge connecting the considered vertex to another one with the lowest cost. After this phase, the algorithm is repeated considering the newly

created connected components, and this iterates until there is only one connected component with all the vertices inside. This result is the minimum spanning tree for that graph. The

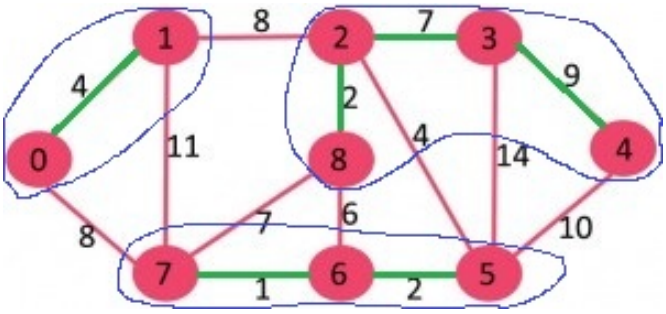


Figure 4.5: Example of the second iteration of the Borůvka algortihm on a graph. The connected components are circled in black, and they contain nodes that are connected by edges with the lowest weights available [37].

complexity of the Borůvka algorithm is $O(ElogV)$, where E is the number of edges and V is the number of vertices.

4.3.3 Prim's algorithm

Born in 1930, this algorithm [38] assumes that the graph is organized as an *adjacency list*, a data structure where each element represents the list of the neighbours of a node in the graph. The procedure to obtain a minimum spanning tree of a graph using this method is visible in figure 4.6 and is:

1. Given a weighted graph with n vertices, choose one of them.
2. Find the nearest vertex, in terms of edge weight, and add it to the MST. A vertex can be chosen only if it's not already in the tree. If starting from a node there are more edges with the same lowest weight, choose one randomly.
3. Iterate the procedure until there are n nodes in the MST.

This algorithm has a complexity that depends on the data structure used to represent the adjacency between the nodes; usually the most used structure is the *priority queue*, and in this

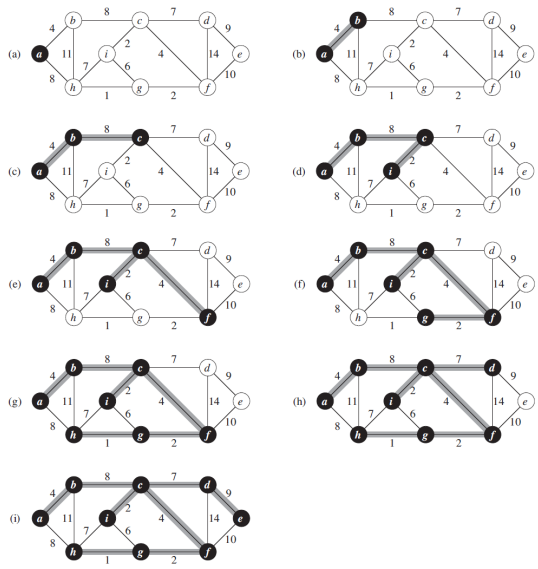


Figure 4.6: Example of how the Prim's algorithm works [39].

case, the complexity is $O(ElogV)$. This value can be further decreased if the structure used is a *Fibonacci heap*: the overall complexity will drop at $O(logV)$.

4.3.4 Kruskal algorithm

The Kruskal algorithm [40], designed in 1956, has a simple idea behind it: if the edges are sorted increasingly concerning their weight, the edges with a lower value will be the best suitable for a minimum spanning tree. This procedure is displayed in figure 4.7.

The algorithm orders the edges as said before and then includes them in the MST iteratively considering the edge with the lowest weight first. An edge is ignored if it forms cycles with the others already included in the tree. The algorithm stops when all vertices are visited. The worst case for the complexity of this algorithm is $O(E\dot{V})$, but this can be lowered to $O(ElogV)$.

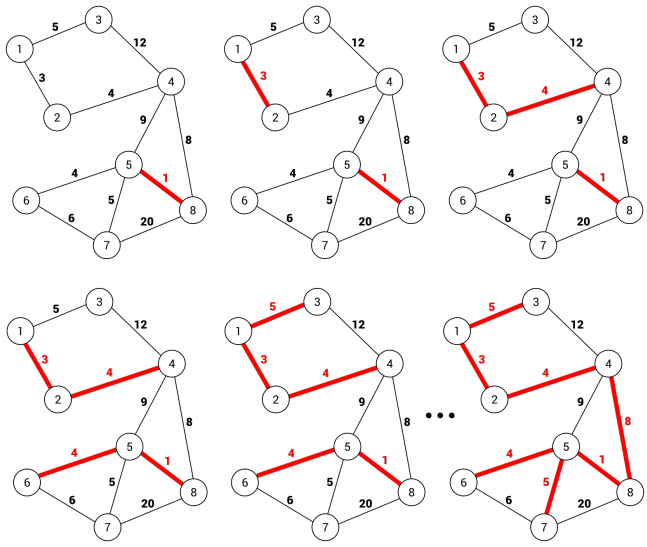


Figure 4.7: Functioning of the Kruskal algorithm [41].

4.4 Applications

Minimum spanning trees are very useful while designing networks, like a computer network, a telecommunication network or even a water or energy supply network.
They're useful also in taxonomy, to classify organisms or abstract concepts, image registration and segmentation, handwritten recognition, circuit design, and phylogeny.

5

Applications of minimum spanning tree estimation

5.1 Introduction

Whenever a set of data can be somehow written in a graph-like structure, finding the Minimum Spanning Tree (MST) can be extremely useful in order to understand the dependencies among each element. As a matter of fact, it is needed to have a fast and effective way to calculate it even when the weights on the edges are noisy.

5.2 Image filtering

Image filtering is one of the fields of application of the theory behind minimum spanning trees. Many effective algorithms have been developed to filter an image and smooth out some noise or unwanted details, like mean and Gaussian filters, which are both linear, and rely on

combining pixel values in a considered window according to a given kernel. Others, like the median, are not linear and rely on ordering data and choosing a representative value.

There are also more complex filters, like the bilateral, which combines the concept of physical and colour distance with the gaussian filtering, to obtain and edge-preserving transformation. From this idea, Stawiaski and Meyer [42], developed their adaptive image filtering, based on minimum spanning trees. The algorithm considers a small window and the neighbourhood of pixels in it: a graph $G = (V, E, W)$ is built with these pixels considering as a weight between i and j the absolute difference between their grey levels. The graph obtained in this way is then used to create an MST that follows the shape of the image because similar pixels, which usually lies on the same side of an edge, have similar grey values and the MST algorithm will connect them. All the trees obtained with the window in different positions of the image, are then fused into a single one. The final image will transform the pixel values along the minimum spanning tree to delete most of the noise. As a result, the final signal to noise ratio of the image is almost doubled with respect to a traditional filtering technique.

5.3 Mathematical problems

Another application of the minimum spanning tree concept has been applied by de Figueiredo and Gomes [43] on a mathematical problem: they proved that euclidean minimum spanning tree can reconstruct differentiable arcs from a sufficiently dense set of samples. Euclidean MST are a particular subset of the minimum spanning tree family, where the function used to set the weights is an euclidean distance. This strategy can be applied when the samples are points that can be represented in euclidean spaces.

Briefly speaking, this problem aims at finding the shape of a curve given a small subset of points of that curve. Humans have a natural predisposition in finding structures, so it's natural for them to find the shape just with a look. A computer has not such perception, since each point is a couple of coordinates in an euclidean bi-dimensional space, in this case. The minimum spanning tree structure allows to reconstruct the curve with the smaller euclidean

distance, and this method has proved to be effective into reconstructing more general curves and when the data are altered with noise.

5.4 Image segmentation

The minimum spanning tree approach can be exploited in the 2D image segmentation as well. The segmentation of an image is one of the most basic problems about low-level image processing: it divides a grey-scale image into parts which have low differences in their tonality. The problem has further extended to coloured images, to recognize patterns or structures in them. Two main types of algorithms have been proposed to segment images: boundary-detection approaches and clustering-based approaches. Xu and Uberbacher [44] developed an innovative solution, based on minimum spanning trees. It first builds a graph where the weights on the edges are determined by the distance between the two pixels, measured by the differences between the two greylevels. This means that the image edges are those couples of pixels that present larger weights; consequently, building a minimum spanning tree will identify and join those pixels with similar values, i.e. the smooth regions within the same object. There will be only one final minimum spanning tree, but some of the edges with the biggest weights can be removed to obtain different regions. The approach of this algorithm is visible in figure 5.1, applied on a letter "T". The algorithm works very well for 2D grey-level images, and it has been tested successfully also on noisy images.

5.5 Clustering

Clustering is another hot topic in modern informatics. Given a set of n points defined as coordinates in a finite space, it is possible to divide them into k different subsets, called "clusters", with the properties:

- Each cluster is disjointed from the others.
- The union of all the clusters returns the initial set.

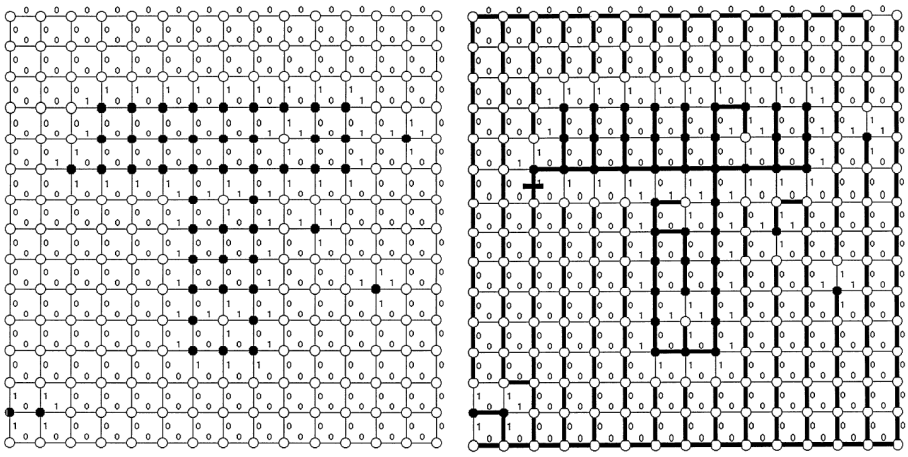


Figure 5.1: The image shows how the algorithm behaves while segmenting a grey-level image [44].

This happens such that the points in the same clusters are "similar" between each other and "different" from the points in the other clusters. This similarity property is obtained with an objective function.

In literature there are various approaches to perform this technique; some solutions start from a single cluster containing all the points and then subdivide it into smaller ones following a precise distance function; others start from n single-point clusters and then aggregate them into bigger clusters; others build the structure of the clusters based on a hierarchy between the points, and so on. The most important and used algorithm for clustering is the k-means, where each cluster is built by comparing each point with the means of the clusters. There's a limitation in this approach: it only produces convex clusters.

The approach proposed by Miller, Nowozin and Lampert [45] should avoid this limitation using a different idea based on minimum spanning trees. The initial set is considered composed by samples coming from the same distribution $p(X)$ and the labels, which are the cluster identifier, come from the distribution $p(Y)$; these labels are assigned such that the mutual information between $p(X)$ and $p(Y)$ is maximized. In this case, the objective function is:

$$I(X, Y) = D_{KL}(p(X, y)||p(X)p(Y)) \tag{5.1}$$

where D_{KL} is the Kullback-Leibler divergence, to detect the differences between the two distributions.

In particular, after the construction of a graph with the considered points, where the weights on the edges are the euclidean distance between a point and the others, the minimum spanning tree is built with regards to a function which derives from the Kullback-Leibler divergence, built to penalize the connections between faraway points and lower the overall entropy, to obtain more stable clusters; this method is able to return a joint set of clusters. Finally, $k - 1$ edges of the minimum spanning tree are removed to divide the single cluster into k clusters; the edges to be removed are chosen by a trade-off between edge length and cluster size. This method is proved to be more effective than the traditional ones even if it assumes that the initial data come from an absolute continuous distribution.

5.6 Image phylogeny

Image phylogeny is a very researched topic, due to its fundamental utility in many research fields. The first environment suitable for this topic is the internet, because nowadays the digital content is available almost everywhere and easily redistributable, even illegally.
There are several main areas where this problem finds its natural application [46], such as:

- Reduce the number of version of a document. This can be useful for storage purposes because since there could be various versions of the same document on the internet, by understanding which one is the common ancestor it's possible to delete all duplicates and save storage space.

- Tracking the distribution of a document. By collecting the same document from various sources on the internet, it is possible to understand the sequence of sites that spread the document. As an example, a video content created and posted on the internet, can be reposted by a newspaper website, for example, and then reposted by another website. This video will suffer a sequence of little modifications and coding steps. These will

make it very different from its first version, even if apparently the content looks the same. By exploiting the phylogeny technique it is possible to understand which one is the very first version of the video and track its path through the internet.

- Copyright protection. By reconstructing the phylogeny of a resource whose intellectual property has been put to hazard by an illegal treatment on the internet, it's also possible to understand who has the right to own it. An official document belonging to a specific person, for example, can be downloaded and re-edited by somebody else who can claim its property; The phylogenetic analysis can understand which one of the two identical documents derives from the other.

- Illegal material detection. With the phylogeny is also possible to track and recognize some illegal material surfing the internet and find its origin.

This problem is actual and significant; it's easy to detect image duplicates, but in these last years researchers have focused on finding a way to recognize near-duplicate images, which is not so immediate as a duplicate could be. Recently, the focus has been moved to finding the structure evolution within a set of images, and many important steps were made.
The most recent result is the identification of some sort of relationships between images in a set [46]. This has been done through the Image Phylogeny Tree (IPT). The set of images is rearranged in a tree structure, where images corresponds to the nodes and the structure of edges reproduce the dependencies between them; traversing the graph from leaves to the root it is possible to reconstruct the evolution of each content. The name comes from the biological realm, due to its similarity with the evolution of natural species. For example, the phylogeny tree of the species which the dog belongs to is shown in figure 5.2.
Just as organisms evolve and change their biological structure over centuries, documents can evolve in different versions of themselves, by the continuous process of manipulation through the internet. These modifications can be unintentional, for example, if a document

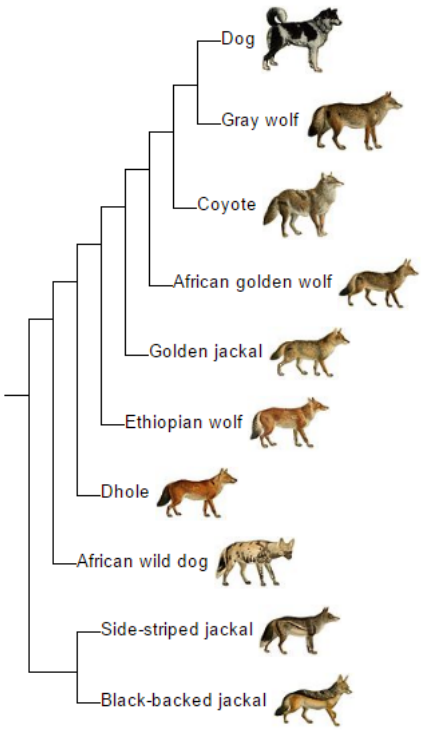


Figure 5.2: Example of the phylogeny tree of the dog. This approach considers an image in each leave and the inner nodes are the common ancestors of the connected leaves [47].

is encoded with a different format or it's badly copied, or intentional if a forger deliberately modifies a document with malicious intents.

The first approach to this problem could be referred to some watermark or fingerprinting techniques. These were known as *traitor tracing* strategies, and consisted in signing the document with a mar. After the content has leaked over the internet, it's possible to recover its history by analyzing the marks and their variations. The problem with these methods is that they aren't always possible because the marks can be destroyed by some transformations. Of course, if a watermark is applied to a document there is always the possibility that previous unidentified copies of the same document are already available on the internet and without the protection of the mark. Also, if the presence of the mark is a known fact, some cheaters may try to circumvent them.

According to the Joly et al. [46] work, with the general idea explained in figure 5.3, a near-duplicate is a version of a document which remains similar to the original and immediately recognizable. They propose an interpretation of the word "duplicate" in an image context, assuming that a document D' is a near-duplicate of the document D if $D' = T(D)$, where T is a set of licit transformations and D is the original document. This set of transformations T is a family that can contain several combinations of transformations, so every document D_1 can be written as a transformation of the original document D with a transformation T which is the combination of all the transformations applied.

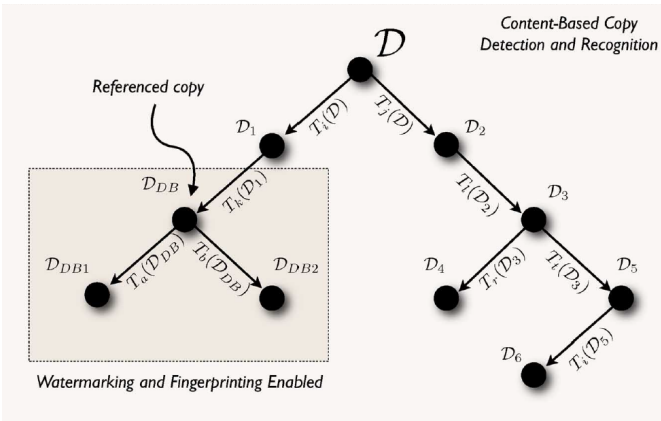


Figure 5.3: This is the idea of Joly et al., where a near-duplicate tree is built with the information obtained. If the watermarking technique can be applied, it's easier to track the transformations, while it's harder if the analysis can rely only on the content of the image [46].

This "duplicate" definition can be seen as a pairwise relationship, because it's a correlation between couples of documents, and it's also transitive, since if a document A is a duplicate of B, and B is a duplicate of C, then A is also said to be a duplicate of C.

There are several techniques to perform near-duplicate detection, like watermarking and fingerprinting, already discussed in the previous section. A more content-based approach consists in analyzing the data itself to understand its changes between two different examples.

After the development of these techniques, many steps were made to overcome the concept

of near-duplicate, and embrace the idea to find a correlation between a bigger number of documents based on their inner structures; for example, Kennedy et al. [48] proposed that the relationships within a set of images can be identified with a pixel by pixel analysis, with the hypothesis that an image that has been heavily manipulated cannot give a child which is less manipulated; in this way, they introduced a directionality in the image manipulation. They also designed a way to build a graph to understand how the images in a set are related to each other, based on the previous idea: some detectors are involved in deciding if there was a manipulation from an image to another one.

More recently De Rosa et al. [49] stated that the single image can be written as a combination of two different parts, one representing the real scene and one about the content-independent part of the image; they assume that two images are dependent if it exists an image processing function, seen as a combination of scaling, rotation, cropping, colour change and JPEG compression, that is able to approximately transform one of these images in the other one. This method brings to the construction of a graph since dependent images on two different nodes are connected by an edge; from this graph, then, cycles and low-value edges are removed.

The last work developed about this topic is directly related to this approach, but it doesn't base its near-duplicate idea on independent parts of the image because if the capturing conditions of the images were different, it could be really difficult to spot the similarities. This mechanism is described in the paper by Dias et al. [50] which this work takes inspiration from.

5.7.1 Phylogenetic analysis of a set of images

The phylogenetic analysis of a set of data relies on a *dissimilarity function*, which parameterizes how close are two elements in the set. The lower the metric between two contents, the more direct the dependency. As a matter of fact, applying a MST algorithm on a complete dissimilarity graph associated to a set of near-duplicate images, similar images will be more likely near on the final tree, and the opposite occurs with dissimilar images, which will be far

away from each other in the final data structure. Naming $T_{\overrightarrow{\beta}}$ an image transformation that maps image I_A to image I_B, the dissimilarity function between a couple of images I_A and I_B is defined as the minimum

$$d_{I_A,I_B} = \min_{\overrightarrow{\beta}} \left| I_B - T_{\overrightarrow{\beta}}(I_A) \right| \tag{5.2}$$

where the difference between the two images is point-wise over the possible values of the parameter $\overrightarrow{\beta}$. This equation does not represent a distance function, and in case of different dimensions, this operation will return the residual of the non-transformed image.
Given n images, the first thing to do is to calculate the dissimilarity between each couple of images, in an asymmetric way, since $d_{I_A,I_B} \neq d_{I_B,I_A}$. To accomplish this procedure, there's the need to consider a set of different transformations T to be used to derive an image from another one. Every transformation can be obtained as a combination of the functions in the following family:

- **Resampling**: an image can be resized and result bigger or smaller than its origin;
- **Cropping**: the image can be cropped to cut some data from it;
- **Affine warping**: rotation, distortions and translations are all available to create a new image modifying an existing one;
- **Brightness and contrast**: by changing the pixel values with some known functions to adjust the colors;
- **Compression**: the JPEG algorithm can be seen as a lossy transformation;

The first problem to face while detecting the dissimilarity between two images I_A and I_B, for every couple of images in the given set, is to estimate in a reliable way the transformation T in terms of the vector $\overrightarrow{\beta}$ that parametrizes it. The best way to perform this operation is by:

1. Using the SURF algorithm, some robust features are detected in each one of the two images, and the corresponding points are calculated. This will produce a set of couples of matched points to estimate the transformation.

2. Estimate the transformation that maps the keypoints of an image to those found in the other one. The transformation, as already stated, can be an affine warping function. Here the above points are made more robust to improve the reliability of the calculations.

3. The colours need to be equalized to understand properly the variations between each couple of images. For each colour channel of I_B mean and variance are calculated and these values are used to normalize the colour channels of I_A. In this section, also non-linear operators, such as gamma correction, are used to simulate a life-like behaviour while considering near-duplicate images.

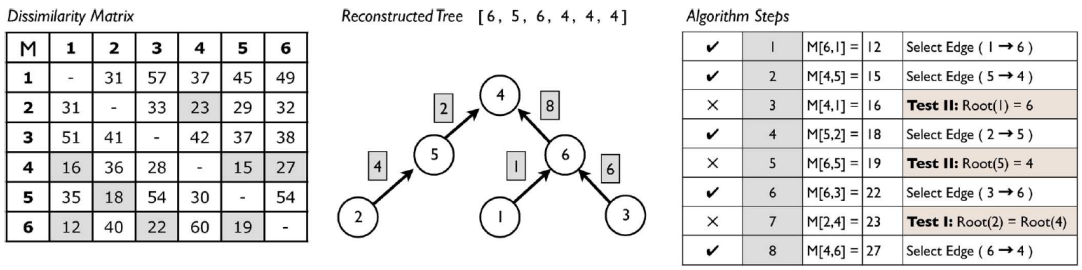
4. The result of the previous steps is compressed using the quantization table of I_B. Once one of the two images have been adapted to the other one with the previous procedure and they have both been uncompressed, the *minimum squared error* technique is used to calculate their point-wise dissimilarity as a final result.

5.7.2 Analysis

The next step of this procedure is to translate this set of dissimilarities between pairs of images into a matrix which can be used to describe the entire set. The creators of this method proposed an algorithm to solve the optimal branching problem, which deal on finding a minimum spanning tree in a non-oriented graph with a known root. Previous attempts to solve this known problem was by Edmonds [51] and Chu and Liu [52], independently. They developed an $O(mn)$ algorithm with n as the number of nodes and m edges. Further

researches on this topic led to different versions of this algorithm, which complexity was finally set as $O(m + nlogn)$. Currently, this represents the best implementation for this problem. The different approach of this work deals with oriented graphs without a known root: the idea is to find the root and then build the oriented tree with an $O(n^2logn)$ complexity; this algorithm has been called *Oriented Kruskal*.

The procedure of Oriented Kruskal starts with a dissimilarity matrix M built upon a set of n near-duplicate images. The algorithm starts considering the graph as a forest, which means that every node is a tree on its own with only one node. Then, the procedure sorts the positions (i, j) of M from lowest to highest value, and joins the single trees with a low dissimilarity. It does this until $n - k$ edges are introduced in the graph, where k is the number of needed trees. Intuitively, the final solution at the original problem will be obtained with $k = 1$, forcing the algorithm to find one source and, consequently, one tree. Example of this is in figure 5.4.



M	1	2	3	4	5	6
1	-	31	57	37	45	49
2	31	-	33	23	29	32
3	51	41	-	42	37	38
4	16	36	28	-	15	27
5	35	18	54	30	-	54
6	12	40	22	60	19	-

✔	1	M[6,1] =	12	Select Edge (1 → 6)
✔	2	M[4,5] =	15	Select Edge (5 → 4)
×	3	M[4,1] =	16	**Test II:** Root(1) = 6
✔	4	M[5,2] =	18	Select Edge (2 → 5)
×	5	M[6,5] =	19	**Test II:** Root(5) = 4
✔	6	M[6,3] =	22	Select Edge (3 → 6)
×	7	M[2,4] =	23	**Test I:** Root(2) = Root(4)
✔	8	M[4,6] =	27	Select Edge (6 → 4)

Figure 5.4: As can be seen by the figure, representing a simple execution of Oriented Kruskal, the positions of the matrix are sorted and evaluated as belonging to the MST. The tests on the right are relative to some checks done by the algorithm during its functioning; in particular, test I triggers when the edge joins two different trees, and test II is called when the edges joins two trees and the endpoint is a root [46].

5.7.3 Test datasets

The testing data set has been created with 1238 images taken from the Uncompressed Color Image Database (UCID), where each image has been transformed and duplicate several times to obtain a final dataset with 250000 cases. From this last data set, another one was created

but paying attention at dropping the root for each tree, to force the algorithm to face another big difficulty.
For the uncontrolled data set, some real-case images take from the internet were transformed and duplicated to get nearly a thousand test cases.

In every case, the transformed images were obtained from the standard dataset images with the application of a transformation like resampling, scaling, rotation, diagonal correction, cropping, colour and contrast change and compression.
These transformations were applied in a way such that the final result is a near-duplicate, so they have parameters with small ranges.

To test this idea, various metrics have been created to understand how good was the detection of the root and the leaves, edges and ancestors sets of the reconstructed tree.

5.7.4 Results

The results show that when the trees are complete, the percentages of correct reconstruction are stable between the sizes considered. In particular, the root is correctly found in almost all the cases while edges, leaves and the ancestors are guessed with a precision of 80% on average. Dropping some links brings some difficulties, because in this case as the size of the tree is lower, i.e. the number of dropped connections is bigger, the edges, leaves and ancestor statistics drop heavily around the 50%, rising to 80% as the connections grow. The root is affected less drastically, going from the 90% with 5 nodes up to almost a 100% with 50 nodes.

If between the dropped nodes there's the root, the resulting IPT has to be compared with the original forest, and in this case, as the number of nodes goes down, the root is way harder to identify, with a 40% of success with 5 nodes. From 15 to 50 nodes, root, leaves and edges are successfully found with an average between 60% and 80%; the ancestry metric is the one that performs better in this situation because it's almost always above 90%. In the uncontrolled scenario, on average the worst metric to satisfy is the first one, which says that is not good if a new image is not a child of its generator image, with more the 20% of errors on all the

test cases. The best metric is the fourth one, so it happened only in 0.2% of the considered situations that the two considered trees have a different root, which is good. The precision metric returned a 73% of perfect reconstruction with the injection of a modified image.

5.7.5 Conclusions

The applications of this work are several, like forensics (especially on the dropped root situation) and copyright enforcement, to understand the changes in the document. Also, new dedicated metrics were developed to estimate the performances of tree-like structures. To conclude, the approach is promising, since it has a good precision in detecting the phylogeny of a set of images.

6

Neural models for minimum spanning tree estimation

6.1 Architectures

The neural network structures tested for this approach are three architectures with different capabilities, and they are a modified version of the standard autoencoder, a ResNet-based autoencoder and a UNet-based one.

6.1.1 Modified autoencoder

The first autoencoder taken into account is, as already stated, a modified version of a traditional fully-connected hourglass network, described earlier in this text. While the usual fully-connected autoencoder is a neural network divided into two distinct parts, where the first one encodes the input creating a latent representation of the data, and the second one

decodes the representation to obtain the input back, this variant concatenates the input to the latent representation, as it can be seen from figure 6.1, to try to encode only the noise and then extrapolate it from the input obtaining its denoised version in output.

The procedure of transformation of the data is as follows:

- **Encode:**
  - Input layer of 64 neurons;
  - First representation of the data with 32 neurons;
  - Second representation of the data with 24 neurons;

- **Decode:**
  - Concatenation of the input: 88 in total;
  - Back to the size of the first latent representation, 32 neurons;
  - Concatenation of the input: 96 neurons;
  - Output layer of 64 neurons;

This model considers an 8×8 input image, and as can be seen from its description, the input data is flattened to a 1-dimensional vector to be worked. Its dimension is reduced by half and then reduced again with a lower rate. Concatenating the input to its representation should theoretically guide the network to learn the representation of the error, obtaining the output by subtraction from the noisy input data.

To test the network with the MNIST dataset in a first analysis, the used autoencoder was not the one just described, but it was a normal fully-connected autoencoder as it was described in the dedicated chapter; since the nature of the two structures is similar, without convolutions and shortcut between the layers, the results obtained can be considered in the same way.

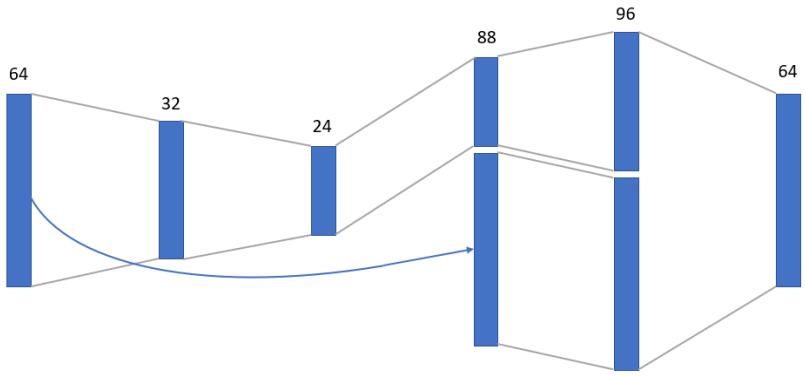


Figure 6.1: This structure is a graphic render of the modified autoencoder described in this section.

6.1.2 ResNet

This particular network architecture, created to build very deep neural networks while maintaining a good error and speed of computation, was considered in this work due to its inner characteristic of learning only the residuals between the output and the input.

The central unit of this network is the *block*: each block is just a series of convolutions with a certain number of filters of different sizes. The block considered in this work is composed of three convolutional layers:

1. (1×1) **convolution**: also called "dimensionality reduction layers", used to raise the number of feature maps;

2. (3×3) **convolution**: with padding and stride, to lower the size of the feature maps;

3. (1×1) **convolution**: usually with a larger number of filters, to generate even more feature maps. This layer is known as *bottleneck*, which means that this layer is slower because it generates many features; in other ResNet architectures, this layer can have a number of filters that is up to four times the usual number that there is in the other layers in the same block.

The number of filters varies and is greater as the center of the network approaches, to become smaller and back to 1 on the output.

The first of the two blocks of the encoder considers 8 filters for the first two layers and 16 for the last. For the second block, these numbers are, respectively, 16 and 32. The input data is gradually decomposed into various feature maps, as it's shown in figure 6.2, while at the same time is reduced in size; when a block is concluded, the input is summed to the result, for the ResNet property: if the number of feature maps between the two is the same, the input is average-pooled to get the correct size; if the values are different, to the input is applied a (1×1) convolutional filter with stride 2 to get the correct number of maps and, at the same time, lower their size to make them match correctly. After both the blocks are applied, the result is flattened to get the final encode of the input data.

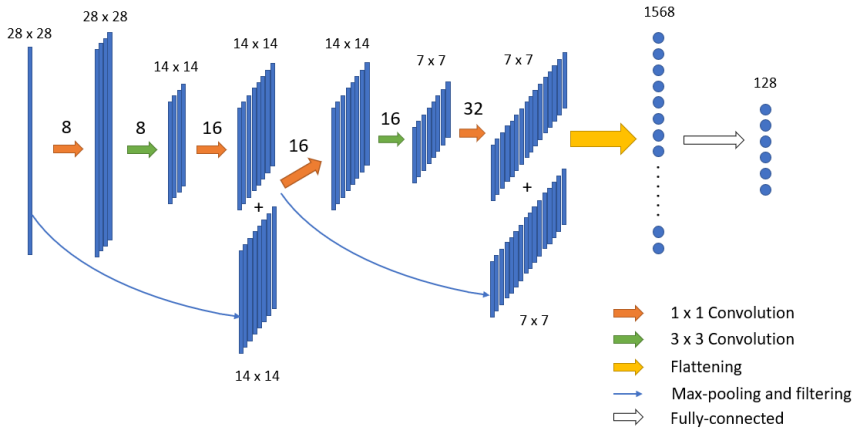


Figure 6.2: This figure shows only the encoding part of the ResNet: on the arrows, divided by colour to identify different sizes of convolutional filters, there is a number which indicates how many filters are applied on that layer.

The decoding phase is similar to the encoding one, but it's performed in the opposite way: the encoding of the input data is elaborated by a fully-connected part which raises the number of units, and then the usual block structures with sum of the adapted input is applied other two times, with a lower number of filter as the output approaches. This time, the input is upsampled with stride 2 if the number of feature maps is the same between the two

layers, or it is convoluted with the inverse convolution and stride 2 if they're different. What mostly changes from the encode phase, besides the operations to transform the input, are the operations in the block: the convolution is now inverted, but with the same parameters; the (1×1) convolution becomes a way to decrease the number of maps and the (3×3) convolution with padding and stride 2 can enlarge the size of the feature maps. After two blocks, the final result is a single image with the same size as the input; the entire procedure can be visualized in figure 6.3. Since the main idea of ResNet is to learn the difference between

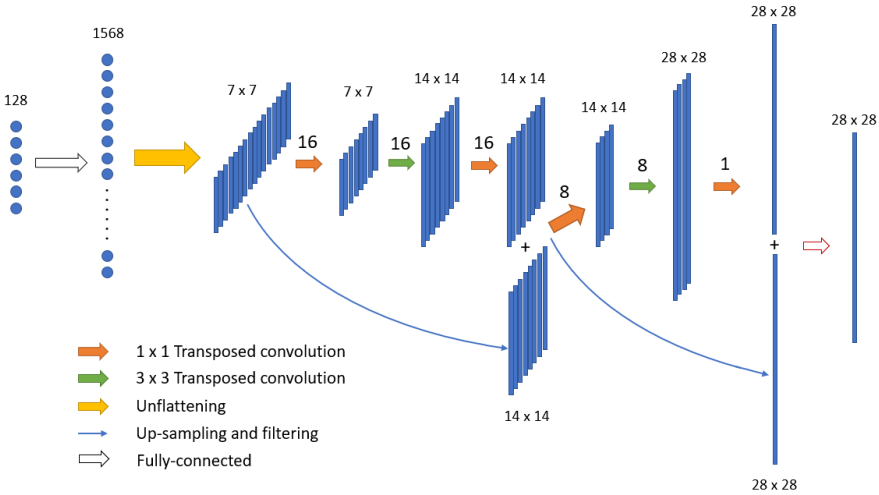


Figure 6.3: In this image is clearly visible the succession of layers that decodes the encoded data following the ResNet specifications; as in the encoder figure, on the arrows there's the number of maps reached with that operation.

the output and the input, the task of denoising should be easier with this approach than with other architectures.

6.1.3 U-Net

U-Net is a convolutional neural network that was created to face the problem of image segmentation in biomedical scope. It was developed in Freiburg, Germany, by Ronneberger, Fischer and Brox [53]. The model proposed is a convolutional neural network where the reduction of dimensionality is divided into layers, corresponding to the several feature maps obtained by the elaboration of the initial data.

U-Net, the name of the network, has been chosen because of the visual shape of the model when it was first proposed and the sequence of the convolutional layers was described entirely. What is immediately visible from the image 6.4 of the original U-Net, besides its shape, is the

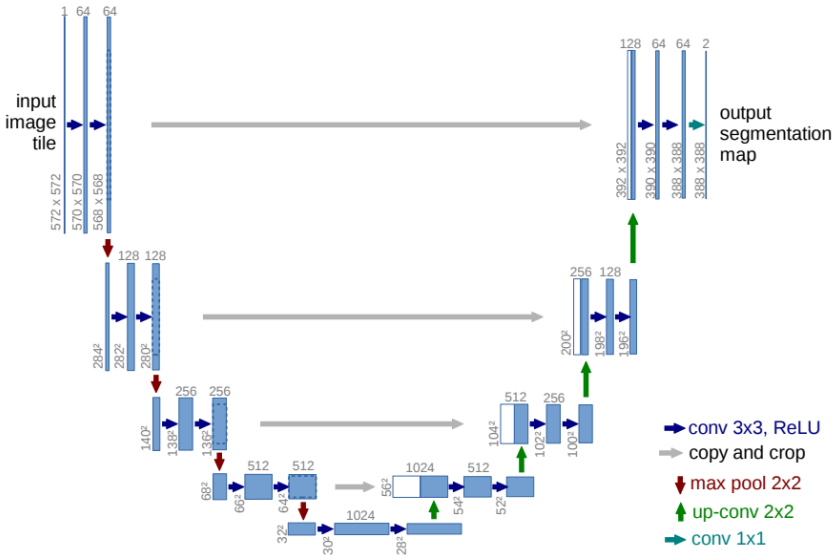


Figure 6.4: This is the very first implementation of U-Net [53], built to face biomedical image segmentation.

clear division into two distinct parts, comparable and conceptually similar to the encoding and decoding parts of an autoencoder, and the presence of a connection between the feature maps of the left segment and those of the right one, coupled by their size; this last one is the particular detail about U-Net: feature maps of the same size are concatenated between the encoding and the decoding phase when the data have the same dimensions again to re-consider high resolution features treated in the encoding part.

The model built for this work, in figure 6.5, derives directly from the U-Net idea, maintaining the skip connections and exploiting the natural subdivision in encoding half and decoding half to apply the concept of autoencoder.

First of all, the number of dimensionality reduction cannot be that big due to the initial tests with the MNIST dataset, composed of images of size 28×28, so the custom architecture will reduce the size of the input only two times. Every time the size changes, a block of operations

is applied to the data, and each one of these blocks is composed by two times the application of a 2D convolution with fixed-size kernels and padding, to obtain n feature maps that have the same size as the input, with the rectifier as activation function. To reduce the size of the features, a max-pooling layer of size (2×2) is applied and then some dropout is added to reduce overfitting. Every time the size of the data changes, the number of filters is doubled, obtaining more features map as the dimensionality lowers. This is what happens concerning the encoding phase.

Then, the operations applied in every block are similar to those in the previous phase, but they are inverted, so the max-pooling operation becomes a transposed convolution with a stride of 2 and padding, to obtain again the size of the second level, and the features belonging to that particular level are concatenated to the result. After some dropout, the normal 2D convolutions of the encoding part are again applied to the maps, and this completes the decoding block. When the original size of the input is reached, as a last operation a (1×1) convolution, known as the dimensionality reduction layer, is applied, to obtain a final result with the same size of the input.

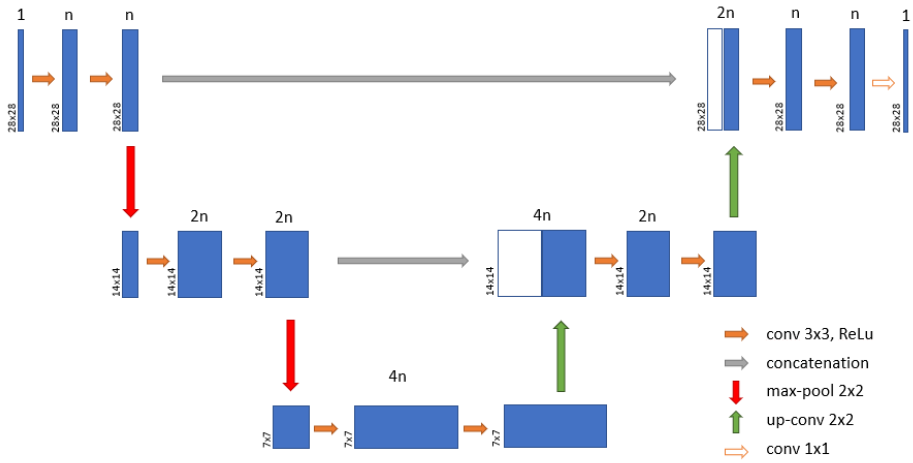


Figure 6.5: The graphical representation of the U-Net based architecture built for this work.

This architecture should be able to perform well since it was designed to extrapolate information from an image, and separating the noise from the corrupted image is the task of

this work.

7

Experimental results

7.1 Dataset creation

The main dataset used in this work is the Uncompressed Color Image Database, known as UCID [54], which is a dataset of 1338 images, visible in figure 7.1, that was built to standardize the usage of test data concerning images. It was presented under the justification that usually the authors of a new paper concerning image treatment adopt their images while training and evaluating the models they present; for the creators of the UCID, this fact makes the comparisons between results of different papers difficult to compare. As another difficulty, sometimes the analysis are done with different levels of compression, which complicates even more the comparisons. UCID was created to fill the gap between different analysis, and it was used in this work because some of its final results, to match the case of the phylogeny tree problem, come from a manipulation of sets of images.

For every sample of the UCID dataset, a family of images is built through the application

Figure 7.1: The images in the UCID dataset are very different from each other, to ensure that the tests can be performed on different parameters conditions, like contrast and luminosity [55].

of some transformation. Each one of these has a given range of action, which is set to simulate the normal variations that can occur to a resource while surfing the internet. These transformations can, of course, be combined to obtain more complex ones; then, colour correction can be performed at the end of the procedure on all colour channels. These are the considered transformations and their range:

1. **Geometry**

   Resampling: [90%, 110%]

   Global scaling: [90%, 110%]

   Rigid Rotation: [-5°, 5°]

   Scaling by axis: [90%, 110%]

   Rotation: [-5°, 5°]

   Off-diagonal correction: [0.95, 1.05]

2. **Cropping**

   Cropping: [0%, 5%]

3. **Color**

   Brightness: [-10%, 10%]

   Contrast: [-10%, 10%]

   Gamma: [0.9, 1.1]

4. **Compression**

   Re-compression: [50%, 100%]

As can be seen, the applied transformations have narrow ranges, due to the realistic assumption that an image cannot be changed completely by these transformations and at the same time remain a near-duplicate of that specific image.

Once the near-duplicate images have been created, the dissimilarity between each possible couple is calculated and this value is put in a matrix in the correct position; e.g. the dissimilarity between the image i and the image j corresponds to the number in position (i, j) in the matrix, and this is done for every position; the diagonal of the matrix, relative to the couple (i, i), is not computed, under the assumption that an image is the perfect duplicate of itself, so there is no dissimilarity, and the final value in the matrix is 0. Thus, the matrix is related

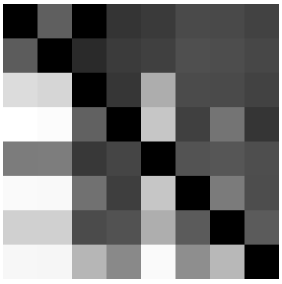


Figure 7.2: This image shows a generic dissimilarity matrix for a given graph.

to a single family of n images, where one is the original one, while the remaining $n - 1$ are near-duplicate versions of it obtained with the application of some simple transformations as explained in the previous chapter; an example of dissimilarity matrix is figure 7.2.

The ground truth matrix is a representation of the minimum spanning tree structure in the image set considered. It's represented as a matrix where the position (i, j) has its maximum value (255 for the representation in 8 bits) if i is connected to j and is its father node in the tree structure. Of course, all the positions (i, i) are equal to 0, since in a minimum spanning tree there are no cycles in the same node, i.e. a node cannot be its father. Just as an example, as it can be seen in figure 7.3, if in the row i there are two positions $\neq 0$, (i, j) and (i, k), it

means that in the tree the node i has two children, j and k. In particular, each ground truth

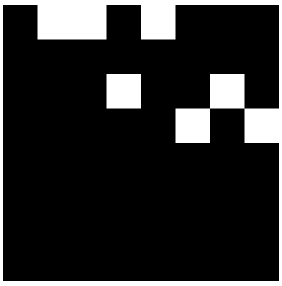


Figure 7.3: This figure represents a generic groundtruth matrix.

image is related to a dissimilarity image, and the task of this work is to make an autoencoder learn the inner strategies to construct the ground truth from the dissimilarity matrix, building the matrix form of the tree.

By following this approach, the procedure of associating a dissimilarity matrix describing a graph with its ground truth representing a minimum spanning tree between the node of the graph, with the metric of dissimilarity used also to build the MST, can be extended to the general procedure of obtaining the minimum spanning tree of a generic graph seen as a matrix. This is just the generalization of the MST finding problem, so by considering different matrices size, associated with different size graphs, it could be possible to add the technique developed in this work to the long list of already existing algorithms about this topic.

Another level of abstraction can be reached by considering the dissimilarity matrix just as a noisy version of the ground truth; in this case it is possible to try to denoise the matrix and train a network to do it automatically and generically, and this is exactly what the idea of this work is about. Therefore, the denoising problem, with this structure of the data, can be related to the minimum spanning tree finding problem.

7.2 Metrics used

To evaluate these approaches four different metrics have been developed to estimate how well the reconstructed tree follows the actual phylogeny of the considered images. These metrics

have different implementations depending on the selected case; the first case occurs when the reconstructed IPT is compared to its ground truth. The metrics are presented as follows:

$$\textbf{Root} \ : R(IPT_1, IPT_2) = \begin{cases} 1, & if Root(IPT_1) = Root(IPT_2) \\ 0, & otherwise \end{cases} \tag{7.1}$$

$$\textbf{Leaves} \ : L(IPT_1, IPT_2) = \frac{|L_1 \cap L_2|}{|L_1 \cup L_2|} \tag{7.2}$$

$$\textbf{Edges} \ : E(IPT_1, IPT_2) = \frac{|E_1 \cap E_2|}{n-1} \tag{7.3}$$

$$\textbf{Ancestry} \ : A(IPT_1, IPT_2) = \frac{|A_1 \cap A_2|}{|A_1 \cup A_2|} \tag{7.4}$$

where E_i, L_i and A_i are, respectively, the sets containing all the edges, leaves and ancestors of the tree i. Each one of these metrics evaluates a different useful property and it can be used not only to see if the resulting tree is valid but also can highlight how the overall algorithm performs.

As an interpretation, the root metric returns a positive result if the root of the reconstructed tree has been found correctly; the edges metric return the fraction of the correct guessed edges over the number of expected ones, and this happens similarly for the leaves and ancestor metrics;

7.3 MNIST and first partial results

The first train and test procedures of the architectures have been with the MNIST dataset. It is a set of handwritten numbers from 0 to 9, represented in 8-bits, where each digit is related to an image of size 28×28. These digits have been collected from the different writing styles of the employees of the American Census Bureau and American high school students [56]. Then, these numbers were quantized at 8-bits to have a uniform representation of the information among the entire set; the dataset is in total made of 60000 different training images and 10000

test images, each one with its label representing the correct number displayed. Due to the

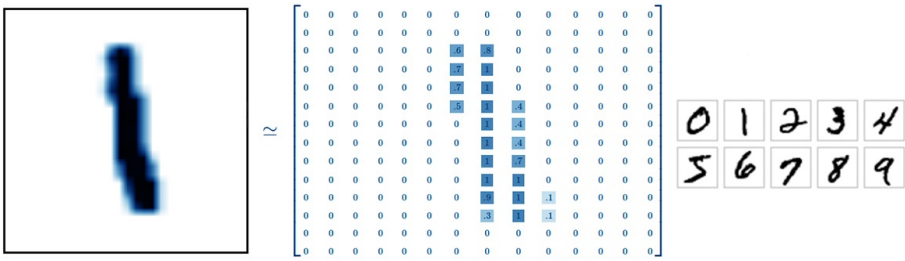


Figure 7.4: In this image it's possible to see how the data are kept in the MNIST dataset; in particular, this example is simplified and the images have a size of 14×14, but the idea behind is the same [57].

simplicity and the wideness of the data in it, the MNIST dataset is usually adopted while testing the idea behind an approach concerning two-dimensional data; even for this work, the prototypes of networks related to the three architectures seen in the previous section, have been tried with this dataset.

In this work, the MNIST dataset has been used to test the capabilities of the very first implementations of the networks. Since the problem faced was about denoising, and this dataset comes without any noise, it was necessary to add it to the images to test the architectures properly: every image in the dataset was copied, and a noise matrix has been added to the copy. A noise matrix is a 28×28 matrix M where each element is extracted from a Gaussian distribution:

$$M(i,j) \sim \mathcal{N}(0,1) \qquad \forall (i,j) \in [0,27] \tag{7.5}$$

The two matrices are then summed element-wise; in this way, the noisy dataset just obtained became the actual set of data to be fed to the networks, and the original data, where each image is coupled with its noisy counterpart, became the ground truth to be obtained from the noisy data.

The only analysis done to understand how well the noisy MNIST dataset behaves, was on the loss and the accuracy of the models, calculated on 10 epochs of training. As it can be seen in the figure 7.5, the loss is constantly near 0, which means that the standard autoencoder behaves very well while training the noisy MNIST.

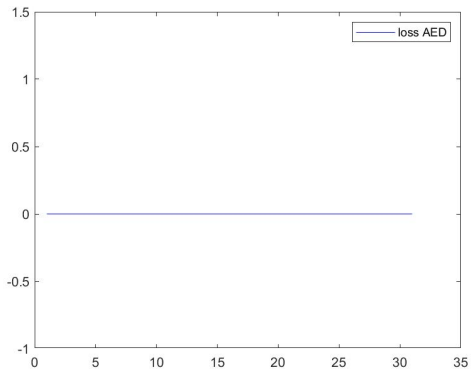


Figure 7.5: The plot of the losses for every epoch of training of the standard autoencoder.

A similar result can be visualized in figure 7.6. It's possible to see that the training and validation losses are very low and similar to each other, so the network is not overfitting. This means that the ResNet based model isn't too specific for the training data and can behave very well even with other samples of the same distribution.

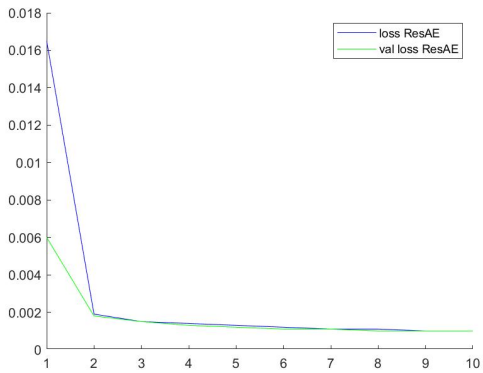


Figure 7.6: The plot of the training losses of the ResNet-based autoencoder. In green, the loss of the validation phase for every epochs.

The different behaviour of the U-Net based model can be visualized in figure 7.7 where the validation loss is higher than the training loss. To confirm the bad behaviour, the accuracy is way lower than it should be to be satisfying. To be fair, the statistics are clearly improving, and this could mean that a greater number of training epochs could be the solution to obtain

better measures with this approach.

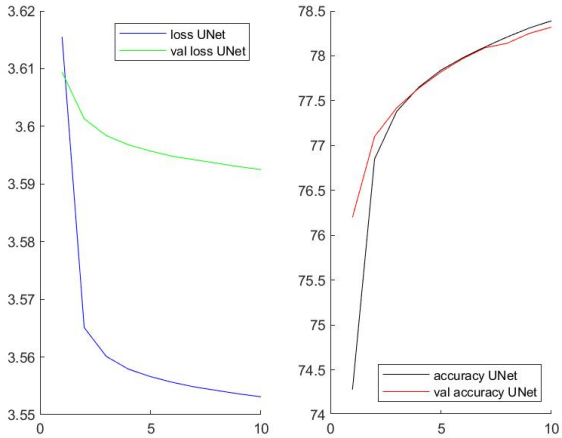


Figure 7.7: The plots of the training and validation losses of the model built with U-Net coupled with the respective accuracies.

7.4 UCID analysis to find the best model

To test the architectures with the UCID dataset, there's the need to build correct graph structures using the images of the set. From every image, three graphs are built; each graph is made by seven near-duplicate images that have been created from the original one, generating a structure with 8 nodes. The set of dissimilarity and ground truth matrices obtained are then fed to the model, divided as 35% training set, 35% validation set and the remaining 30% as test set, with a training phase of 100 epochs. The metrics adopted to test the reliability of the different systems are those that have been presented earlier in this work: the percentage of correct capture of roots, leaves, edges and ancestors. The algorithm used for comparison is the Oriented Kruskal, used in the same paper.

The results are the following:

- **Modified autoencoder**: in table 7.1 are shown the results of a first training of the network. The tests were performed on the modified autoencoder presented earlier

in this chapter, and on a different version of it, where the layer with 24 neurons has been reduced to 8 neurons, to simulate dropout to ease the computation and possibly achieve better results.

As it is possible to notice, the values of root, edges and ancestors are always slightly lower than the results obtained by Oriented Kruskal, denoting a not so good behaviour of the network; only the leaves metric performs better. Overall, the values don't change too much between the two versions of autoencoder. This inefficiency could be due to some overfitting, since the final loss, relative to the test set, is way greater than the train loss.

Algorithm	% Root	% Leaves	% Edges	% Ancestors	Train Loss	Test Loss
Modified AE	65.56	86.27	38.01	29.88	0.0220	0.1065
Reduced	66.57	86.55	37.77	29.39	0.0274	0.1071
OK	67.27	70.98	47.83	32.57	-	-

Table 7.1: Modified autoencoder comparison table

- **Res-Net autoencoder**: the obtained values relative to the Res-Net based architecture are presented in table 7.2. In parallel with the already presented Res-Net based approach, a slightly different model is also tested: this variant adds more layers of batch normalization, to adapt the inner parameters to the different scalings of the input data. The results of these Res-Net based models are not so good as they were expected by looking at the losses of the model with the MNIST dataset: although the root and leaves precisions are better than those of the Oriented Kruskal approach, edges and ancestors metrics perform worse than the standard algorithm. On average, the Res-Net model is not better than the modified autoencoder approach, since the only metric in which the last one is better is the root metric. The add of some batch normalization layers can improve the root and leaves detection, but lowers the probabilities of finding edges and ancestors; surely, with the batch normalization, both losses gain some percentage points.

Looking at the losses, and in particular, to the differences between train loss and test loss, it's possible to think that the main reason for the bad results is, as it was stated for the modified autoencoder, some overfitting.

Algorithm	% Root	% Leaves	% Edges	% Ancestors	Train Loss	Test Loss
Res-Net	70.66	85.96	26.55	24.81	0.0206	0.1307
Normalized	71.65	88.22	24.52	22.94	0.0194	0.1277
OK	67.27	70.98	47.83	32.57	-	-

Table 7.2: Res-Net based autoencoder comparison table

- **U-Net autoencoder**: to discuss the performances of this last approach, based on U-Net, the same training procedure was performed on the same network with a different number of starting filters, which number will vary through the model following the usual rule of U-Net: doubling at each reduction of dimensionality, going back to 1 as the end of the decoder part approaches. The number of filters considered are 2, 4, 8, 16 and 32.

  By briefly looking at 7.3, the results look promising, since every filter numerosity will lead to a model that performs better than the basic Oriented Kruskal algorithm, by almost doubling the percentage of the leaves and edges metrics. The other metrics are better in a less dramatic way, but still greater than the values relative to the referring algorithm. The train and test losses are close to each other, confirming the absence of overfitting.

  A more complete view highlights a clear hierarchy of performances between the different approaches considered: with 2 starting filters, the results are better than the referring algorithm, but raising this number will improve almost all the metrics. It looks like a median-high number of filters leads to the best overall results.

The main reason of the fails of the modified autoencoder and the Res-Net approaches could be found in:

Initial filters	% Root	% Leaves	% Edges	% Ancestors	Train Loss	Test Loss
2	82.55	95.03	64.75	47.87	0.0578	0.0565
4	96.36	92.79	73.33	52.84	0.0404	0.0382
8	95.58	92.06	76.06	58.84	0.0319	0.0335
16	96.70	92.82	75.24	61.24	0.0231	0.0355
32	97.71	94.75	72.56	64	0.0167	0.0381
OK	67.27	70.98	47.83	32.57	-	-

Table 7.3: U-Net based autoencoder comparison table

1. since the modified autoencoder is a fully connected neural network and the input data, besides their low dimensionality, are clearly more similar to images than other type of data, a pixel-by-pixel analysis brought by this approach could be not the best way to detect and find an agglomerate structure like the minimum spanning tree is in a better way than a standard algorithmic procedure. The creation of autoencoders in a fully-connected way are of course possible and effective, but in this situation, the noise is totally casual and not so easy to isolate;

2. the Res-Net was built to learn the residuals of a function, which is the difference between the function itself and the identity: the noise that was added in the previous MNIST tests was a Gaussian additive noise, which has a known mathematical structure, and could be easily enclosed in the residuals learned; this explains why the losses of the Res-Net in the previous tests were the best among all the models. In these tests, the noise was not Gaussian additive but random, so it was harder for the model to understand its structure and lead to good results;

Finally, the results suggest that the best approach to continue the analysis is the U-Net based model, with 8, 16, or 32 filters.

Further analysis will regard the U-Net models divided by the number of starting filters, but with an extension: the loss function used during the training phase in the previous tests was a combination of the standard MSE with another value weighted by a parameter λ, set to 0.016 after some previous tests:

$$loss(a, b) = loss_1 + 0.016 \cdot loss_2 \tag{7.6}$$

where $loss_1$ is the MSE on the input and output matrices, and $loss_2$ is the average value of the Frobenius norm of the predicted matrix; this norm is a normalization factor, used to lower the complexity of the result.

The new loss function adopted adds a value to the previous sum; this factor $loss_3$ represents a thresholding procedure of the output image, where the predicted data is thresholded with a factor 0.7 and this binary image obtained is used to calculate the MAE with the ground truth image. The multiplicative λ_2 value has been set to 0.05. Thus, the new loss becomes:

$$loss(a, b) = loss_1 + 0.016 \cdot loss_2 + 0.05 \cdot loss_3 \tag{7.7}$$

Another condition has been considered while training the models: if the validation loss doesn't improve in 10 epochs, then the training is stopped and the model is declared ready to test. This guarantees not only that the training will use less time to be completed, but also that the model uses the best parameters achieved because the loss can also become worse than its optimum value. The condition is set on the validation loss because since the validation is like a test simulation, controlling the value of this metric will give the best results with newly fed data.

7.5.1 Complete graphs

The tests were performed with the new loss and termination condition on the U-Net based model with matrices relative to 8, 16 and 32 nodes graphs. These matrices have been obtained in the same exact way as those of the previous tests and the datasets have a dissimilarity matrix and a ground truth matrix for each graph. In figure 7.8 there is the plot of the losses about

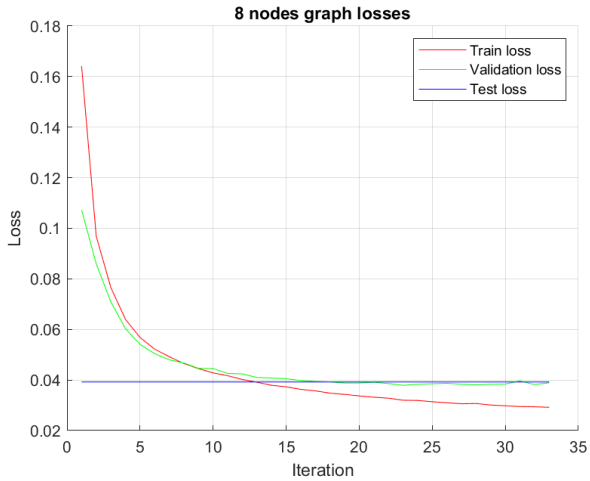


Figure 7.8: This graph shows the comparison between the losses of the train, validation and test phases with 8 nodes per graph.

training, validation and testing of the graphs with 8 nodes. It is possible to see that the train loss is the one that goes lower, but the validation is very similar to the test loss, which is a single number, at the end of the procedure. Note that these data are relative to the model running with 16 initial filters, just to visually compare the differences.

Initial filters	% Root	% Leaves	% Edges	% Ancestors	Test Loss
8	94.95	91.28	72.98	52.54	0.03699
16	95.54	91.65	74.48	56.64	0.03772
32	93.91	88.73	72.59	53.65	0.03973
OK	66.26	70.23	47.41	32.52	-

Table 7.4: U-Net based autoencoder with 8 nodes per graph

In table 7.4 are presented the results of the U-Net autoencoder with images of size 8×8, which means that each graph has 8 nodes relative to the same number of images. The table shows that apparently, the best number of initial filters is 16 because all the metrics are better than those of the other strategies about a different number of initial filters. This could mean that 16 is the right number that balances the tradeoff between a greater number of features and a high complexity, with 32 filters, and a lower number of features and a low complexity, with 8 filters. But this situation is not stable, because the precisions are very close: in a more general way, it is possible to say that all approaches work very well since they all obtain better results than the Oriented Kruskal algorithm. In the same way figure 7.8 does, figure 7.9 shows

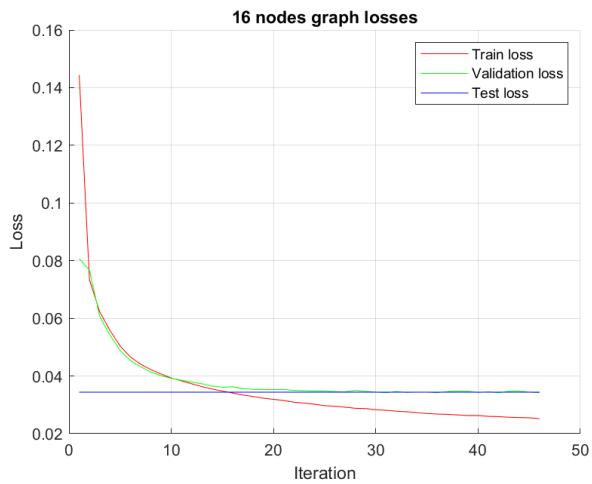


Figure 7.9: This graph highlights the differences between the losses with 16 nodes per graph.

the main differences between the losses while training the dataset of graphs with 16 nodes and 16 initial filters. The overall behaviour of the losses is very similar and coherent with the previous case.

A similar situation can be viewed in tables 7.5 and 7.6, where the strategy of having 16 initial filters is no longer the best one, but the metrics are very close and this means that all the approaches have a similar behaviour, which anyway looks better than Oriented Kruskal. To complete the visual interpretation of the losses for every size of graph, figure 7.10 shows that in the 32 nodes case, the losses behave in a standard way, also seen in both figures 7.8 and 7.9.

Initial filters	% Root	% Leaves	% Edges	% Ancestors	Test Loss
8	91.17	90.94	68.20	46.64	0.03310
16	93.10	90.09	66.22	46.30	0.03442
32	92.96	88.92	67.21	46.79	0.03358
OK	60.97	70.55	39.95	25.86	-

Table 7.5: U-Net based autoencoder with 16 nodes per graph

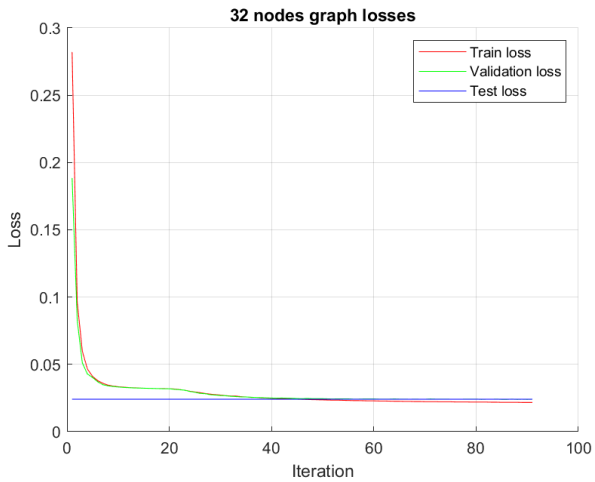


Figure 7.10: This graph displays the train, validation and test losses with 32 nodes per graph.

This case is particular because it is clear that good loss values are reached after fewer iterations than in the other cases, but then they need more time to stabilize. To maintain coherence between the plots, the data are relative to an analysis of the model with 16 starting filters. Note that in every plot, the number of iterations, i.e. epochs, is never 100, as set during the setup of the model, but it is always lower due to the stopping condition on the validation loss.

In a macroscopical view, in table 7.6 the precisions are degrading as much as the size of the initial data is increasing, denoting that if the input graph is bigger then the detection of the correct MST is harder, even if the difference between the metrics in one or the other case is never greater than 15 percentual points.

The root and leaves metrics are those that degrade less with the increase of the size, while

Initial filters	% Root	% Leaves	% Edges	% Ancestors	Test Loss
8	88.82	89.62	62.38	41.81	0.02458
16	88.92	89.69	63.26	42.55	0.02412
32	90.63	87.92	62.40	41.98	0.02423
OK	55.23	71.91	36.29	23.42	-

Table 7.6: U-Net based autoencoder with 32 nodes per graph

the edges and ancestors metrics have the biggest differences between each approach. A cause in this could be found in the representation itself of the matrices, where the root and the leaves could be detected more easily than edges and ancestors.

7.5.2 Dropped nodes

As a further analysis, it is taken into consideration the concept that there could be some missing nodes in the graph, i.e. missing images from a set. These situations are common in forensics, for example, where there could be some resources missing from the considered set, but the problem needs to be solved anyway with a good precision.

In this work, the decision was to remove half of the images from each graph of 16 and 32 nodes to obtain, respectively, graphs of 8 and 16 nodes each. This drop of vertices has been done programmatically on the already existing datasets, by removing the same half of the columns and rows (except the root) in both the dissimilarity matrix and the ground truth matrix regarding the same graph.

The dropped datasets have been then tested on the weights learned in the learning phases of the standard datasets with the same size, i.e. the dropped 16 nodes dataset has been tested on the weights of the standard 8 nodes dataset, and so on. Figure 7.11 shows that every variation of the autoencoder approach is able to perform better than Oriented Kruskal in all the chosen metrics. In particular, to make a comparison with the non-dropped situation, the root and leaves metric go from an average which is above 90% to barely over 80%, while the edges and ancestors metrics, which are known to be a harder task, have an average degradation of

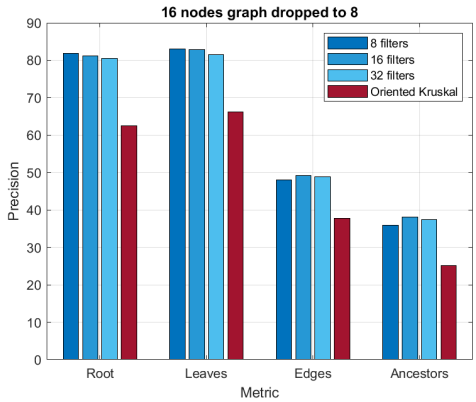


Figure 7.11: A comparison between the metrics of the dropped 16 nodes dataset with various numbers of starting filters and the Oriented Kruskal algortihm.

almost 20 points. The test losses are all around 10%: even though these results are still very good, compared to those obtained with a standard dataset there are signals of a clear difficulty in correctly finding the MST when there are dropped nodes. A similar situation happens while considering the case of a graph with 32 nodes dropped to 16. Figure 7.12 highlights a behaviour which is in line with what has been seen in the previous case, but the percentual are overall worse. This could be due to the not negligible quantity of information removed with 16 vertices. A similar behaviour can be visualized also on the test losses, which are around 6%, higher than the standard case, denoting a general difficulty of detection.

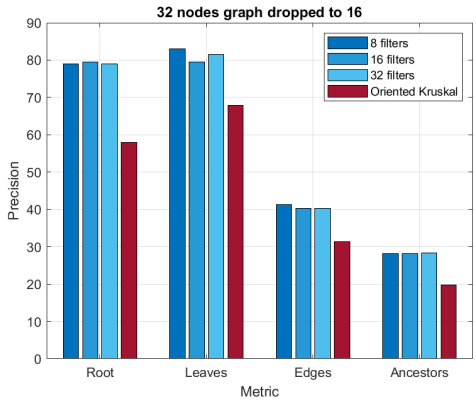


Figure 7.12: The metrics of the dataset with 32 nodes dropped to 16 and the numbers of starting filters compared to the Oriented Kruskal algorithm.

7.5.3 Audio samples

To the same models has been applied another dataset of the same nature but generated with audio samples instead of images. 7 tracks have been created starting from another one and the dissimilarities between the 8 samples of this set are calculated for each couple. These dissimilarities generate the input matrix of the model, while the real way how the samples are related is summarized in the ground truth matrix.

This has been done to generalize this entire approach, by evaluating it on data coming from different sources. Since audio samples are harder to treat than images, the results of the learning should be no better than the results already obtained with that kind of data.

Initial filters	% Root	% Leaves	% Edges	% Ancestors	Test Loss
8	93.40	88.60	63.57	46.00	0.05937
16	91.85	88.61	63.40	46.99	0.06138
32	93.32	89.33	64.83	49.21	0.06166
OK	81.74	66.90	37.93	22.37	-

Table 7.7: Analysis on the 8 nodes graph made with audio samples

In table 7.7 there are the results of the learning of 8×8 dissimilarity matrices and the comparison, as always, with the Oriented Kruskal algorithm. The differences between the neural approach and the standard one are visible by comparing the columns of the table, each one referring to the same metric.

The first look confirms the performances of U-Net, since the precisions in finding the root, leaves, edges and ancestors in neurally reconstructed minimum spanning trees are significantly better than those related to the reconstruction with the Kruskal algorithm. The overall results, as expected, are worse than those resulting from the application of the same model to matrices deriving from images, due to the difficulty of correctly quantifying the dissimilarity between audio samples.

8
Conclusion

In this work, a new approach different from the usual algorithms that proposes to solve the problem of minimum spanning tree finding has been presented. This method adopts a specific variation of a well known neural network structure, called denoising autoencoder, to extract the MST structure from a dissimilarity matrix. The matrix was built by comparing couple by couple the data belonging to a set of near-duplicate samples, under the assumption that the matrix representation of the minimum spanning tree of the graph identified by the data can be well approximated by the application of a denoising operation on the dissimilarity matrix.

Using the MNIST dataset, the best model out of three following different construction strategies has been detected, and on this model, the analysis have been made on different parameters sets. The images dataset has been created starting from the standard UCID. The model has been also tested with some variations of the dataset to match the possible real-life fields of application of this problem. Another dataset has been created starting from a

collection of audio samples.

The results have been presented by a comparison between the new model and an oriented version of the Kruskal algorithm. The numbers show that the U-Net based approach, which was delineated as the best one in this work, outperforms the standard algorithm in each one of the tests done.

However, there are some strong limitations to this approach: since the autoencoder is a neural network, the learning phase implies that a huge amount of fixed-size matrices are fed as input. While the first obstacle can be easily overcome by the modern algorithms, which are able to produce a sufficient number of samples in a reasonable amount of time, the second one is trickier, since a fixed matrix size means that the approach is ready to compute the minimum spanning trees for those size of set it was trained with. Further developments of this idea could include studies on methods to make this approach scalable. The improvements can also consider the training on datasets of real data since those adopted in this work were a derivation of ad hoc created ones. These future modifications could make even better and reliable an approach that looks promising in every analysis where it was employed so far.

References

[1] Smakosh, *Build your perceptron neural net from scratch*, 2017. [Online]. Available: https://dev.to/smakosh/build-your-perceptron-neural-net-from-scratch-4co

[2] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks." in *AISTATS*, ser. JMLR Proceedings, G. J. Gordon, D. B. Dunson, and M. Dudík, Eds., vol. 15. JMLR.org, 2011, pp. 315–323. [Online]. Available: http://dblp.uni-trier.de/db/journals/jmlr/jmlrp15.html#GlorotBB11

[3] *Hyperparameters in artificial neural networks*. [Online]. Available: https://en.wikipedia.org/wiki/Artificial_neural_network

[4] P. Shrivastava, *Challenges in Deep Learning*, 2017. [Online]. Available: https://hackernoon.com/challenges-in-deep-learning-57bbf6e73bb

[5] B. R. Reza Bosagh Zadeh, *Fully Connected Deep Networks*. [Online]. Available: https://www.oreilly.com/library/view/tensorflow-for-deep/9781491980446/ch04.html

[6] *Training Neural Networks Part 1*. [Online]. Available: https://srdas.github.io/DLBook/GradientDescentTechniques.html

[7] A. Moawad, *Neural networks and back-propagation explained in a simple way*, 2018. [Online]. Available: https://medium.com/datathings/neural-networks-and-backpropagation-explained-in-a-simple-way-f540a3611f5e

[8] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[9] *Overfitting*. [Online]. Available: https://en.wikipedia.org/wiki/Overfitting

[10] O. Prakash, *What are the key trade-offs between overfitting and underfitting?*, 2018. [Online]. Available: https://www.quora.com/What-are-the-key-trade-offs-between-overfitting-and-underfitting

[11] *Universal approximation theorem*. [Online]. Available: https://en.wikipedia.org/wiki/Universal_approximation_theorem

[12] M. Nielsen, *A visual proof that neural nets can compute any function*, 2019. [Online]. Available: http://neuralnetworksanddeeplearning.com/chap4.html

[13] R. Parmar, *Training Deep Neural Networks*, 2018. [Online]. Available: https://towardsdatascience.com/training-deep-neural-networks-9fdb1964b964

[14] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *ArXiv e-prints*, 11 2015.

[15] M. Basavarajaiah, *Maxpooling vs minpooling vs average pooling*, 2019. [Online]. Available: https://medium.com/@bdhuma/which-pooling-method-is-better-maxpooling-vs-minpooling-vs-average-pooling-95fb03f45a9

[16] *MAX-Pooling*. [Online]. Available: https://embarc.org/embarc_mli/doc/build/html/MLI_kernels/pooling_max.html

[17] J. P. Dominguez-Morales, "Neuromorphic audio processing through real-time embedded spiking neural networks," Ph.D. dissertation, 12 2018.

[18] S. Saha, *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way*, 2018. [Online]. Available: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[19] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[20] J. X. T. S. Sala, *Guida alle architetture di reti profonde*, 2018. [Online]. Available: https://www.deeplearningitalia.com/guida-alle-architetture-di-reti-profonde/

[21] *Review: ResNet — Winner of ILSVRC 2015 (Image Classification, Localization, Detection)*, 2018. [Online]. Available: https://mc.ai/review-resnet-winner-of-ilsvrc-2015-image-classification-localization-detection/

[22] *Autoencoder*. [Online]. Available: https://en.wikipedia.org/wiki/Autoencoder#Variations

[23] F. Chollet, *Building Autoencoders in Keras*, 2016. [Online]. Available: https://blog.keras.io/building-autoencoders-in-keras.html

[24] N. Hubens, *Deep inside: Autoencoders*, 2018. [Online]. Available: https://towardsdatascience.com/deep-inside-autoencoders-7e41f319999f

[25] *Autoencoders*. [Online]. Available: http://ufldl.stanford.edu/tutorial/unsupervised/Autoencoders/

[26] A. Ng, *CS294A Lecture notes*. [Online]. Available: https://web.stanford.edu/class/archive/cs/cs294a/cs294a.1104/sparseAutoencoder.pdf

[27] A. Abdelaal, *Autoencoders for Image Reconstruction in Python and Keras*, 2019. [Online]. Available: https://stackabuse.com/autoencoders-for-image-reconstruction-in-python-and-keras/

[28] Z. Zhang, Y. Wu, C. Gan, and Q. Zhu, "The optimally designed autoencoder network for compressed sensing," *EURASIP Journal on Image and Video Processing*, vol. 2019, no. 1, p. 56, Apr 2019. [Online]. Available: https://doi.org/10.1186/s13640-019-0460-5

[29] S. Rifai, P. Vincent, X. Muller, X. Glorot, and Y. Bengio, "Contractive auto-encoders: Explicit invariance during feature extraction," in *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ser. ICML'11. USA: Omnipress, 2011, pp. 833–840. [Online]. Available: http://dl.acm.org/citation.cfm?id=3104482.3104587

[30] Sayantini, *Applications of Autoencoders*, 2019. [Online]. Available: https://www.edureka.co/blog/autoencoders-tutorial/

[31] *Minimum spanning tree*. [Online]. Available: https://en.wikipedia.org/wiki/Minimum_spanning_tree

[32] M. Shields, *Minimum spanning tree*, 2018. [Online]. Available: https://slideplayer.com/slide/13700460/

[33] *Maximum cut*. [Online]. Available: https://en.wikipedia.org/wiki/Maximum_cut

[34] *Minimum Spanning Tree lecture*. [Online]. Available: http://www.di-srv.unisa.it/professori/anselmo/MST.pdf

[35] M. Aigner and G. M. Ziegler, *Cayley's formula for the number of trees*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 201–206. [Online]. Available: https://doi.org/10.1007/978-3-642-00856-6_30

[36] *Time complexity*. [Online]. Available: https://en.wikipedia.org/wiki/Time_complexity

[37] *Boruvka's algorithm*. [Online]. Available: https://www.geeksforgeeks.org/boruvkas-algorithm-greedy-algo-9/

[38] *Prim's Minimum Spanning Tree (MST)*. [Online]. Available: https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/

[39] Maha, *Confusion in CLRS's version of Prim's algorithm*, 2015. [Online]. Available: https://cs.stackexchange.com/questions/50964/confusion-in-clrss-version-of-prims-algorithm

[40] *Kruskal's Spanning Tree Algorithm*. [Online]. Available: https://www.tutorialspoint.com/data_structures_algorithms/kruskals_spanning_tree_algorithm.htm

[41] M. Jamro, *Kruskal's algorithm*. [Online]. Available: https://www.oreilly.com/library/view/c-data-structures/9781788833738/a2714e77-af8c-494a-a485-df5521ced3f0.xhtml

[42] J. Stawiaski and F. Meyer, "Minimum spanning tree adaptive image filtering," in *2009 16th IEEE International Conference on Image Processing (ICIP)*, Nov 2009, pp. 2245–2248.

[43] L. H. de Figueiredo and J. de Miranda Gomes, "Computational morphology of curves," *The Visual Computer*, vol. 11, no. 2, pp. 105–112, Feb 1994. [Online]. Available: https://doi.org/10.1007/BF01889981

[44] M. Dewi, A. Armiati, and S. Alvini, "Image segmentation using minimum spanning tree," *IOP Conference Series: Materials Science and Engineering*, vol. 335, p. 012135, 04 2018.

[45] A. C. Müller, S. Nowozin, and C. H. Lampert, "Information theoretic clustering using minimum spanning trees," in *Pattern Recognition*, A. Pinz, T. Pock, H. Bischof, and F. Leberl, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 205–215.

[46] Z. Dias, A. Rocha, and S. Goldenstein, "Image phylogeny by minimal spanning trees," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 2, pp. 774–788, April 2012.

[47] *Canis phylogeny*. [Online]. Available: https://commons.wikimedia.org/wiki/File:Canis_phylogeny_(eng).png

[48] L. Kennedy and S.-F. Chang, "Internet image archaeology: Automatically tracing the manipulation history of photographs on the web," in *Proceedings of the 16th ACM International Conference on Multimedia*, ser. MM '08. New York, NY, USA: ACM, 2008, pp. 349–358. [Online]. Available: http://doi.acm.org/10.1145/1459359.1459406

[49] A. De Rosa, F. Uccheddu, A. Costanzo, A. Piva, and M. Barni, "Exploring image dependencies: a new challenge in image forensics," 02 2010, p. 75410.

[50] Z. Dias, A. Rocha, and S. Goldenstein, "First steps toward image phylogeny," in *2010 IEEE International Workshop on Information Forensics and Security*, Dec 2010, pp. 1–6.

[51] J. Edmonds, "Optimum branchings," in *J. Res. Nat. Bur. Standards*, vol. 71B, 1967, p. 233–240.

[52] Y. i Chu and T. Liu, "On the shortest arborescence of a directed graph," 1965.

[53] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," *CoRR*, vol. abs/1505.04597, 2015. [Online]. Available: http://arxiv.org/abs/1505.04597

[54] G. Schaefer and M. Stich, "UCID: an uncompressed color image database," in *Storage and Retrieval Methods and Applications for Multimedia 2004*, M. M. Yeung, R. W. Lienhart, and C.-S. Li, Eds., vol. 5307, International Society for Optics and Photonics. SPIE, 2003, pp. 472 – 480. [Online]. Available: https://doi.org/10.1117/12.525375

[55] A. Ai-Sadi, M. Bin-Yahya, and A. Almulhem, "Identification of image fragments for file carving," 12 2013.

[56] *MNIST database*. [Online]. Available: https://en.wikipedia.org/wiki/MNIST_database

[57] D. team, *TensorFlow MNIST Dataset and Softmax Regression*, 2019. [Online]. Available: https://data-flair.training/blogs/tensorflow-mnist-dataset/

Acknowledgments

It's always hard trying to include everybody, but I will make a try. As this is the conclusion of a hard journey, I would like to thank those people that brought me here, my mom and my dad, always ready to listen to me and endure me during my endless speeches.

In the same way, Martina: our adventure began just before university, and your presence and support were vital to survive into this world.

Thank you to Laura, Daniele, my uncles and Lory, and all the members of my family: you have been my points of reference, with me every second.

Thank you to all my friends, with their laughs and moments they shared with me.

Also, I would like to thank professor Milani, for the beautiful opportunities given by this thesis, which is part of a scientific paper that has the potential to represent a goal for the informatics of the future (I hope).

Last but not least, you, that surely read the whole thesis just to come to this page; thank you for dedicating some time of your day to me, I hope that my work could be an inspiration for what you are up to.

So, in general,...

...THANK YOU ALL!

Ringraziamenti

E' sempre difficile provare ad inserire tutti, ma ci proverò. Dato che questo rappresenta la fine di un lungo viaggio, vorrei ringraziare quelle persone che mi hanno portato qui, mia mamma e mio papà, sempre pronti ad ascoltarmi e sopportarmi durante i miei discorsi infiniti.

Allo stesso modo, Martina: la nostra avventura è iniziata appena prima dell'università, e la tua presenza e il tuo supporto sono stati essenziali per sopravvivere in un mondo come questo.

Grazie a Laura, Daniele, agli zii e Lorenzo, e a tutti i membri della mia famiglia: siete stati i miei punti di riferimento, con me in ogni istante.

Grazie a tutti i miei amici, con le loro risate e i momenti che hanno condiviso con me.

Inoltre, vorrei ringraziare il professor Milani, per le meravigliose opportunità offerte da questa tesi, parte di un paper scientifico che ha il potenziale di rappresentare un traguardo per l'informatica del futuro (spero).

Ultimo ma non meno importante, grazie a te, che sicuramente hai letto l'intera tesi solo per arrivare a questo momento; grazie per avermi dedicato un po' del tuo tempo prezioso, spero che questo lavoro possa essere di ispirazione per ciò che stai facendo.

Quindi, in generale,...

...GRAZIE A TUTTI!