POLITECNICO DI TORINO

DIPARTIMENTO DI ELETTRONICA E TELECOMUNICAZIONI

Corso di Laurea in Ingegneria Elettronica

Tesi di Laurea Magistrale

Extreme Learning and Data Parallelization for Single Incremental Task scenario

Relatore
Maurizio MARTINA

Correlatore:
Guido MASERA

Candidato
Davide DI FEBBO

Anno accademico 2018 – 2019

Abstract

The objective of this project was to study the Extreme Learning Machine (ELM) algorithm, capable of training a Single Layer Feed-forward Network (SLFN) faster than the standard gradient-based algorithm, and exploit it to develop a network which learns new tasks incrementally, without forgetting the previous ones. This application is known as Continuous Learning and it one of the many difficult scenarios of machine learning. The ELM algorithm is able to adjust the network parameters in a single step. This not only allows the network to be trained in an extremely small amount of time, but it also provides the best solution and optimization to the training process.
We also examined two ELM techniques to train the network using batches of data samples, they are known as Online-Sequential Extreme Learning Machine (OS-ELM) and Parallel Extreme Learning Machine (P-ELM). We noticed that the data parallelization used by P-ELM retains informations of all the encountered batches. Therefore, we exploits this characteristic to cope with the problem of catastrophic forgetting in Continuous Learning.
The results obtained are comparable with the current available algorithm but the adaptability of the ELM classifier, in a Convolutional Neural Network (CNN), changes depending on the network complexity.

Contents

List of Figures

List of Tables

Introduction

Artificial Neural Networks (ANN) are powerful structures capable of learning and performing tasks without the necessity to program them. These structures are characterized by a learning phase, controlled by an algorithm, where the network parameters are properly adjusted. Most algorithms are Gradient-based, where the parameters are iteratively tuned and, in order to obtain acceptable results, lots of steps are required. Therefore, the training takes a great amount of time.
Extreme Learning Machine (ELM) algorithm, proposed by Huang et al. [1], is able to adjust the network parameters in single step. This means that the training time is drastically reduced. This algorithm also provides the best generalization to the training process.
In the first part of this work, we are going to provide a detailed background about machine learning and artificial neural networks. We are also going to talk about some methods and network structures, mostly used in nowadays applications.
In the second part of the project we are going to analyse the Extreme Learning algorithm, in particular the Online-Sequential Extreme Learning Machines (OS-ELM) and the Parallel Extreme Learning Machine (P-ELM), which are two algorithms that allow the training to be performed using data distributed in batches.
The final part will cover the implementation of one of these two algorithm on the Single Incremental Task scenario, and the results will be compared with the best algorithm developed for these applications.

Chapter 1

State of the Art

1.1 Background on Machine Learning

Machine Learning is a branch of **Artificial Intelligence (AI)** and it refers to computer systems that, once built, are capable of learning and performing certain tasks without explicitly program them. More precisely they exploit algorithms able to analyse data and extract informations used in the learning phase. The process from which the machine learns is called *Training*. Machine Learning find several applications in many areas, for example in the health field is used to recognize/diagnose pathologies, in smart cars for autonomous drive, they are also used for deep computing, objects or face recognition, and so on.

1.1.1 Common applications and problems

As mentioned there are different tasks that a learning machine can perform. In particular it is possible to distinguish two main applications, which are **Regression** and **Classification** problems. The *Regression* problem refers to when the machine is fed with a data distribution and its purpose is to build a continuous model function that best approximate the input behaviour. As can be seen from figure 1.1 the red dots represent a certain data distribution, for example it may represent the price of an house (y axis) depending on its dimension (x axis), or the blood fat depending on the weight of a person, or the reliability of an object over time, and so on. It is also possible to have different dependencies for the same output, called *features*, in fact the price of an house depends not only on its dimension but also on the number of rooms, the construction date and many other features. As shown in figure the distribution is not continuous, meaning that we do not have information for every value on the x axis. For this reason the machine has to learn its shape and try to approximate it with a continuous function, represented with a blue line in the figure. Once trained the machine will be able to predict the result y for every input value x.

The result for this type of problem does not need to be precise because is im-

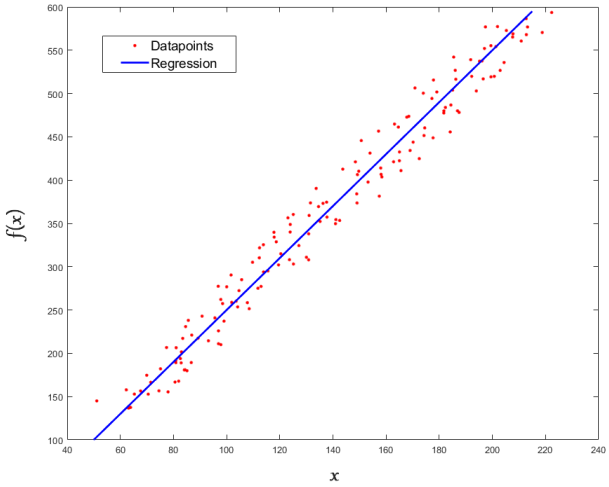


Figure 1.1: Regression problem

possible to cover every case of house prices and also they depend on many factors. However the error must be minimized. The example, in figure 1.1, shows a simple case where the trend is linear, thus a simple first grade polynomial is enough to approximate the data distribution This case is also called *Linear Regression*, however depending on the number of input features the complexity increases and therefore the number of parameters that the machine must learn.
Regarding the *Classification* problem the machine has to decide whenever a condition is True or False. An example could be recognizing when a received e-mail is either spam or not, also to identify cancer or other diseases in medical field, or if an image contains a dog or not.
The fundamental difference with the regression problem is that the result must be precise, there are no approximations, the performance is given by the ratio between the number of correct classifications and the total input samples. The table shown in figure 1.2 is called **Confusion Matrix**. Considering the previous example of the dog, there are four possible outcomes, **True Positive (TP)** and **True Negative (TN)** if the machine correctly recognizes the presence and absence of a dog in the image, respectively, **False Positive (FP)** and **False Negative (FN)** if the machine made a mistake on recognizing a dog in an image without dogs and vice versa respectively.

In order to correctly evaluate the performances, by checking the correct classifications respect to the total number of sample it is not enough because, considering a dataset with 1000 samples of which 999 are negative classifications (0) and the last sample is a positive classification (1), by building a system which outputs only zeros for every input we obtain 99% accuracy which is an excellent result however

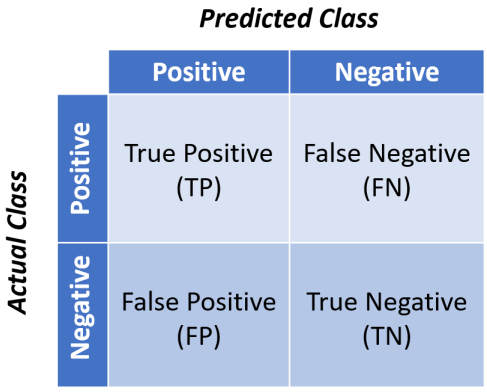


Figure 1.2: Confusion matrix for classification problem

the machine is completely useless since is unable to classify anything. For this reason there is another method to evaluate the performances which is by defining the *Precision* (**P**) and the *Recall* (**R**):

$$\begin{aligned} \mathbf{P} &= \frac{\texttt{True Positive}}{\texttt{True Positive + False Positive}} \\ \mathbf{R} &= \frac{\texttt{True Positive}}{\texttt{True Positive + False Negative}} \end{aligned} \tag{1.1}$$

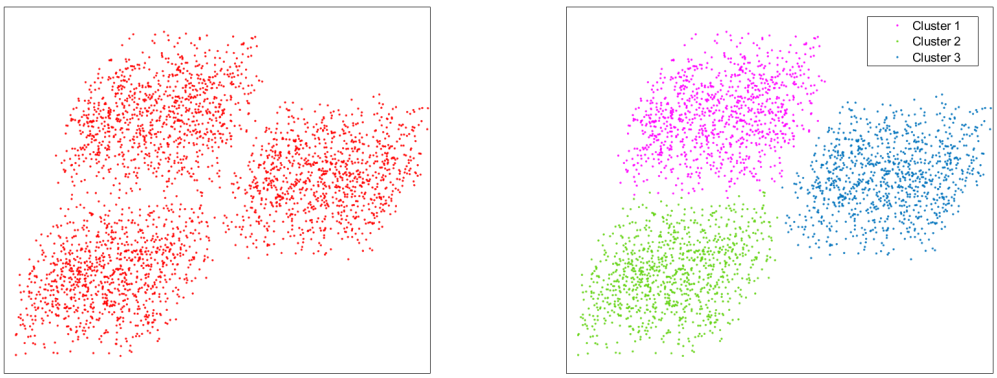
considering the same situation as before, the computed *Precision* and *Recall* would be both 0, since none of the positive samples has been correctly classified. This two evaluation methods clearly show that, the machine being used, is not able to perform the task.
The classification problem is not just limited to the True/False (1/0) cases, also referred as *Binary Classification*, but it can extend to Multi-Class Classification where the output specify the class membership of an input sample. Let's suppose that the inputs are images of ten different animals and we want the machine to recognize which is the animal represented in a certain image, then there will ten different outputs and each one will represent a specific class. Also for this case we can construct a confusion matrix and evaluate the performances.

1.2 Training methods

In machine learning there are different training methods, depending on the characteristics of the input data available and the task to execute. The common ones are **Supervised**, **Unsupervised** and **Semi-Unsupervised** learning.
Before introducing these methods there is another concept to clarify, in machine learning there are different type of input data, more precisely it is important to

distinguish **Training** data, **Testing** data and **Validation** data, they are all extracted from a dataset but they have different tasks. The machine learns by using the *Training* data then, in order to evaluate the performances, it uses the *Testing* data. The *Validation* data are instead used for parameters optimizations and performance improvements.
In *Supervised* learning the machine is fed with training data, each data is labelled with the expected output result (Regression) or class (Classification) to be obtained, the algorithm then compares the actual output with the expected one and learns by adapting its parameters and trying to get close to the expected result. In the *Unsupervised* case the data in the training set are not labelled. This means that the machine does not know the class or the output to be expected, its purpose is to find a pattern or a cluster in the data distribution in order to label the data itself. As can be seen from figure 1.3a the original distribution has no information, however it is possible to distinguish groups of data. The machine, after the training process, should be able to organize the data as shown in figure 1.3b, this operation is also called **Clustering**.

The *Semi-Supervised* learning, as the name says, has the characteristic of both



(a) Unlabelled data distribution

(b) Clustered data distribution

previous methods. In this case only part of the input samples are labelled and the machine has to label the remaining data using also the clustering algorithm. There are other methods such as **Fine-Tuning**, used to update some of the system parameters in order to improve performances, and **Reinforced Learning** used when the system as to adapt to the environment in order to maximize the prediction effectiveness.

1.2.1 Gradient Descent algorithm

It is now important to describe, more deeply, the process of learning, for this reason let's consider back the regression problem, in order to approximate the input distribution we need to construct polynomial and correctly adjust its parameters:

$$y^{(j)} = f(w_0 + w_1 x_1^{(j)} + ... + w_n x_n^{(j)}) \tag{1.2}$$

for simplicity let's assume $f(w_i, x_i^{(j)})$ to be a simple linear function and that the output depends on just one feature, the expression then becomes:

$$y^{(j)} = w_0 + w_1 x_1^{(j)} \tag{1.3}$$

during training, the machine updates the values of w_0 and w_1, which are called **weights**, so that the function, given by the polynomial 1.3, matches or approximates the given distribution. This means that the prediction $y^{(j)}$ (actual output) needs to be equal or similar to the desired result $t^{(j)}$ (expected output) for every input sample $x_i^{(j)}$.
There are many algorithm used to perform such operation, they exploit the **Cost Function** (or **Loss Function**).
This function expresses the difference between the two mentioned output. For regression problems an example is the *Squared Error Function*:

$$C(w_0, w_1) = \frac{1}{2m} \sum_{j=1}^{m} (y^{(j)} - t^{(j)})^2 \tag{1.4}$$

Instead, for the classification problem, also referred as *Logistic Regression*, the cost function is different due to the output nature (the expected output can be either 0 or 1):

$$C(w_0, w_1) = -\frac{1}{m} \left[\sum_{j=1}^{m} t^{(j)} \log(y^{(j)}) + (1 - t^{(j)}) \log(1 - y^{(j)}) \right] \tag{1.5}$$

Where m represents the total number of input samples available in the dataset, y_j is the prediction, obtained using the j-th sample, and t_j is the corresponding result to be obtained with that sample.
$C(w_0, w_1)$ is the cost function, it is obtained summing the difference of y_j and t_j for every sample in the dataset It can be noticed that, for a good approximation, this difference must be small or zero, in fact it basically represents the overall error committed by the machine. Therefore we need the cost function to be as lowest as possible.
In this case, $C(w_0, w_1)$ depends on the weights w_0 and w_1 thus it will be a second order function, like the one shown in figure 1.4. The goal is to minimize it by adjusting the two weights. The algorithm used for this operation is called **Gradient**

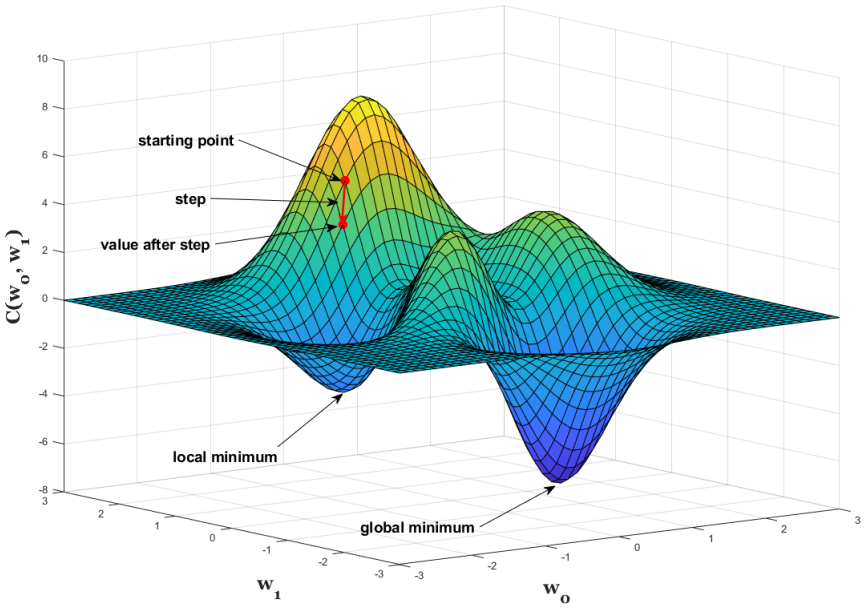


Figure 1.4: Two-varible Cost Function example

Descent, it exploits the gradient to search for the direction where the function decreases and the weights are updated using the following assignment:

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} C(w_0, w_1) \tag{1.6}$$

The expression (1.6) is not an equation, the symbol ($\leftarrow$) stands for assignment, meaning that the value of w_i is changed with the result of the operation. Basically every weights is initialized to a random small value (between a specific range), given these weights the cost function will assume a certain value which will be the *starting point* of the training phase. From this point the partial derivative is computed with respect to every weight in the network to be updated, **one by one**, meaning that for every weight there will be the corresponding partial derivative of the cost function. Finally we can update all of them using (1.6).
By doing this we move the cost function value from the *starting point* to another smaller value in the direction of the max variation, downward. The length of the path (also called *step*) travelled between the *starting point* and the new value depends on the partial derivative (steepness of the function) and the α parameter called **Learning Rate**. In fact using a big value of α the dashes are longer and we are able to get to the minimum faster, however the direction of the max variation not always brings to a minimum and making huge steps may result in longer paths that may lead to worse performances. On the other hand, using small value of α the

steps are smaller but the precision will be higher and the convergence is granted. However the smaller the α the longer is the time required to reach the minimum and thus the training end.
Another problem caused by using big value of α is that the value of the cost function may diverge. It may happen that the steps are so big that the minimum is surpassed and the function starts to increase again.
In order to make sure that the chosen *Learning Rate* α gives the right results, we can represent the cost function value versus the training steps and check if it is converging to a small value, close to zero. In fact, when the *Learning Rate* is too big the graph diverges or oscillates between small and big values.
As can be seen the figure 1.4, there are different minimum of the function, with different minimum values. Depending on how the weights are initialized the *starting point* could be in any position in the graph meaning that by training the machine in these different position we may end up in different minimum, therefore different performances are obtained. For example the minimum on the right (the deeper one) gives better performances respect to the one on the left, since the cost function thus the error is lower.
There are no special ways to know which initialization gives the best performance if not by building the cost function graph. This example, however, shows the case where only two weights are considered, but working with many other input features the graph becomes impossible to represent.

The figure 1.4 is a simple example of a cost function shape. There might be

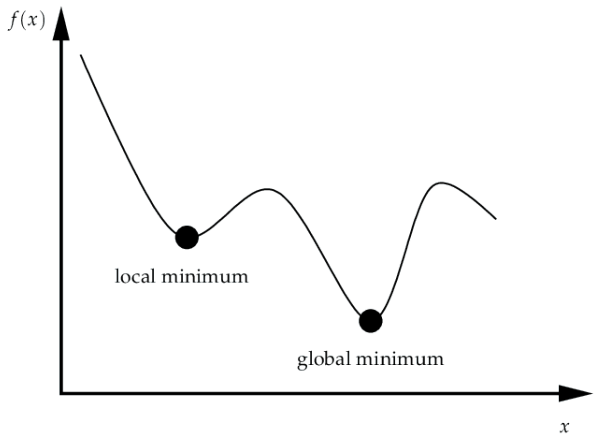


Figure 1.5: Local and Global minimums

some cases where the local minimums are small and localized in the middle of a downhill, as shown in figure 1.5, if the learning rate is too small than the training could remain stuck in that point. A solution to this problem is by adding a term

in (1.6):

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} C(w_0, w_1) + \eta \Delta w_i \tag{1.7}$$

the coefficient η is called *momentum*. Basically this term take in consideration how much the weight changed in the previous step so that in the moment the training reaches a local minimum (zero gradient) the weight value changes anyway. If the step is big enough then the small local minimum can be surpassed. The entity of this term is controlled by the momentum coefficient η.
Another important thing to notice is that each step requires the computation of the cost function and its derivative respect to every weight, this means that the network has to evaluate every input sample in the dataset, for this reason each step is also called *epoch*.

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} \left(\frac{1}{2m} \sum_{j=1}^{m} (y^{(j)} - t^{(j)})^2 \right) \tag{1.8}$$

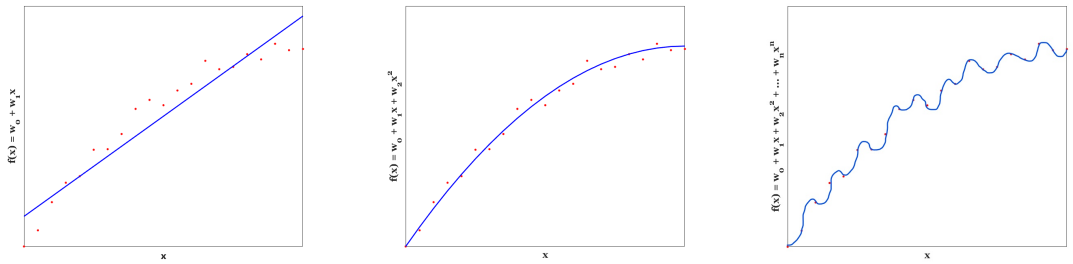
There is another possibility where, instead of computing the gradient with respect to every sample in the dataset, we compute the sum in the equation 1.8 only for a small portion of the dataset, generally called **batch**. By doing this and after updating every weight, we obtain that the direction of the next step does not move precisely towards the cost function minimum. However, repeating the operation for every batch in the dataset and setting the right parameters we can still obtain convergence to the minimum. This is done in order to speed up the computations and, thus, reduce the training time. This algorithm is referred as **Stochastic Gradient Descent (SGD)**. The batches can also contain one sample, this means that the weights are updated each time we receive a sample. The precision however is affected by this number, it is always better to have enough samples in the batch. In SGD each epoch is determined when all the batches in the dataset are evaluated for the update.

1.2.2 The overfitting problem

Another important problem to consider is the machine capability to learn data without "memorizing" them. Considering figure 1.6 it can be seen yet another example of data distribution. Using the polynomial 1.3 with any possible combination of w_0 and w_1, the approximation does not give good results since the two graph have different shapes, this case is called **Underfitting** (also referred as **High Bias**). The parameters given to the system are not enough to obtain a good representation of the data distribution and the accuracies for both training and testing data are low. By increasing the order of the polynomial we can improve the machine capability to adapt to more complex cases, meaning that some of the training input data intersect correctly the approximation and the other are close to it, but the overall

directionality and shape is similar to the distribution. In this case, for both training and testing data the machine gives high performance and good generalization, figure 1.6b.
Lastly, when the polynomial order is big, thus there are many parameters, its precision increases so that it can perfectly intersect all the training data but the shape becomes inappropriate for a good generalization, meaning that the performances for the training data is maximized and the machine correctly predict them, but for the testing data the prediction are inaccurate. This case is called **Overfitting** (also referred as **High Variance**) and can be interpreted as the machine *"memorizing"* rather than *"learning"*.

The problem is how to know, regardless, when the machine is situated in one of



(a) High Bias (Underfit) (b) Good generalization (c) High Variance (Overfit)

Figure 1.6: The three possible fitting conditions

these three conditions, one way is comparing the model accuracies obtained, for the *High Bias* case both training and testing accuracy converge to the same value, however the accuracy is small due to low precision when approximating the distribution. For the *High Variance* case there is a big difference between training and testing accuracy, in particular the latter is way smaller showing the machine incapability to generalize. A good fitting is shown in the last figure where both training and testing obtain almost the same accuracy close to the desired one.
Overfitting is a very common problem in machine learning, there are several ways to reduce it, one of them is to increase the training data so that the machine gathers more particularity in order to better generalize .The number of features also affect the result, it is better to have less features but meaningful rather than many features correlated one from another.
It is difficult to manually gather new training data, in many cases the number of input samples in a dataset is to low and the overfitting is inevitable. It is possible to create new data from the old one contained in the dataset, for example, an image rotated by 90 degree can be considered as a new image. This also helps the machine to recognize the same object even if it has different positioning. Another example could be adding noise to an image, change the intensity, contrast, illumination and so on. This technique, which uses old samples to create new ones, is called **Data**

Augmentation.
Another important solution to the overfitting problem is by using **Regularization**. As said the objective, in the training process, is to minimize the *Cost Function* changing the weights value, however they can assume any possible value as long as the function is close to zero. Considering an n-degree polynomial having big value on higher degrees the shape likely becomes the one show in figure 1.6c, instead if those value have smaller contribution then the shape is similar to the one in figure 1.6b. To implement the regularization the *Cost Function* 1.5 can be modified as follows:

$$C(w_0, w_1) = \frac{1}{2m}\sum_{j=1}^{m}(y_i^{(j)} - t_i^{(j)})^2 + \frac{\lambda}{2m}\sum_{i=1}^{n} w_i^2 \tag{1.9}$$

The added term $\frac{\lambda}{2m}\sum_{i=1}^{n} w_i^2$ simply represent the summation of all the squared weights in the network, in this way, since the function 1.9 must be minimized, then this term must be reduced too, meaning that smaller weights are preferred in the learning process. In this way the function shape is controlled and the overfit is reduced. The parameter λ is called *penalty coefficient* and determines the magnitude. With a small λ the effect of the regularization is also small and vice versa. This regularization is also called *Weight Decay*, *Ridge Regression* or *L2* regularization since the weights in the summation are squared, in fact it also exists the *L1* regularization with the following form:

$$C(w_0, w_1) = \frac{1}{2m}\sum_{j=1}^{m}(y_i^{(j)} - t_i^{(j)})^2 + \frac{\lambda}{2m}\sum_{i=1}^{n} |w_i| \tag{1.10}$$

In this case the absolute value of the weights summation is considered. This technique is also called *Lasso Regression* and the difference with the previous one is that it only reduces the weights holding redundant features leaving the important ones rather than reducing all of them. It is extremely useful when the objective is the feature extraction.
The same regularization rules apply to every other types of cost function.

1.3 Deep Neural Networks

Machine Learning takes inspiration from the human brain, therefore is based on the neurons behaviour. As can be seen from figure 1.7, each neuron is composed by multiple inputs called Dendrites, one output called Axon and the nucleus which elaborates the input impulses and, depending on the result of the operation, the neuron provides another impulse on the output if a certain threshold is surpassed. Another important aspect of a neuron are the Synapses, they make sure that the corresponding input impulse is properly weighted before entering the neuron for the elaboration. A neuron learns by adapting the Synapses to react in a certain way depending on the input impulses they receive. Starting from this point of view

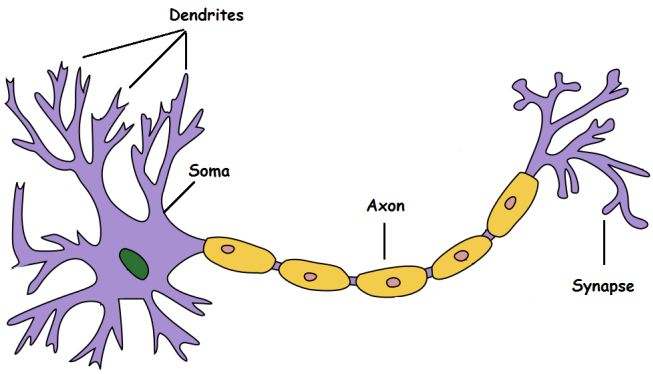


Figure 1.7: Human brain neuron representation

it is possible [25] to arithmetically represent a neuron as a node with multiple input features x_i (Dendrites) and one output y (Axon). It is important to distinguish the *input features* from the number of *input samples*. The features describe the various characteristics of a single input sample (considering for example an object we would have the size, color, weight, height as possible features, but they belong to the same object (sample)), in fact, as shown in figure 1.8 the node has multiple input but they all refer to the features of a single input sample. One way to distinguish these two parameters is through nomenclature, from now on the input will be represented as $x_i^{(j)}$ and the output as $y^{(j)}$ with $i = 1,2,..,n$ where n is the number of features and $j = 1,2,..,m$ where m is the number of input samples. In order to simulate the Synapses behaviour, each input is multiplied by a value, called *weight* w_i. The node performs a specific operation represented by a function of the weighted sum $f(\sum(w_i x_i^{(j)} + b))$, where the parameter b is called *bias*. A single node (figure 1.8)

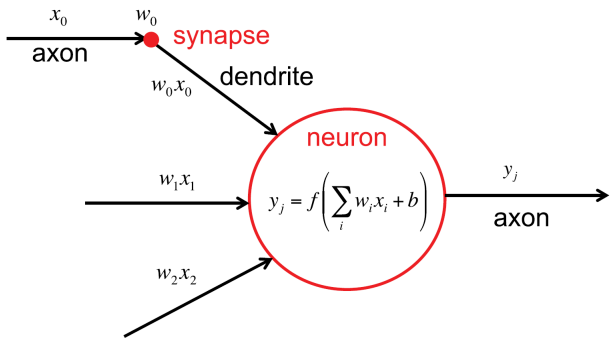


Figure 1.8: Approximative representation of a neuron behaviour. Source [25]

can perform many simple tasks but they have some limitations when dealing with complex operations. For example the boolean AND, OR and NOT, they can be easily performed by one node, with specific configurations, however considering the

XOR or XNOR operations it can be proved that with any combination of weights, inputs and biases the single node cannot correctly perform the task. The solution is to use the multiple nodes and combine their results.

The structure shown in figure 1.9 is a small and simple example of what is called

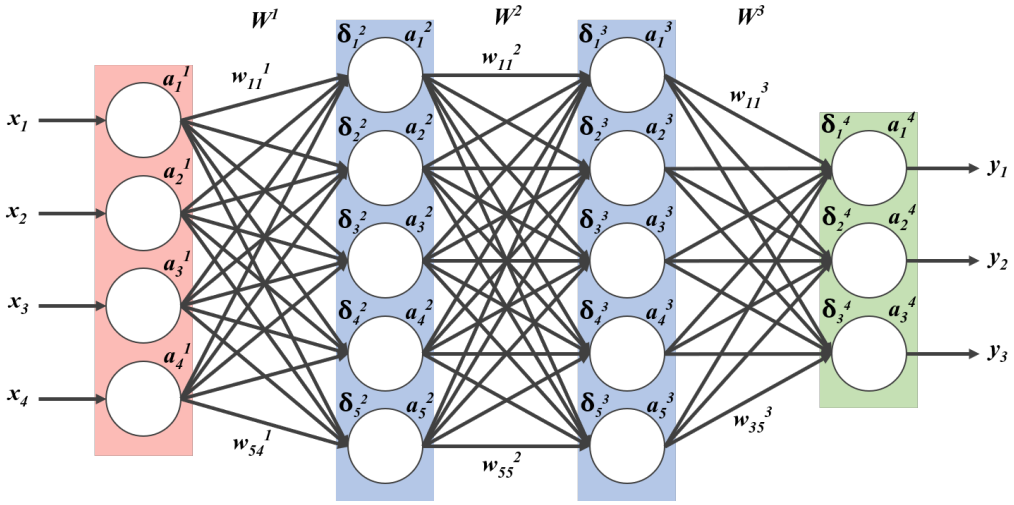


Figure 1.9: Artificial Neural Network

Neural Network (NN) or **Artificial Neural Network (ANN)**, the red nodes delimit the first layer called **Input Layer**. They are not actual neurons in fact they just deliver the datum x_i on the output, the symbol is the same for a convenient representation. The blue layers instead are called **Hidden Layers**, in this case the nodes are like the one discussed before. The last layer is called **Output Layer** where the result of all the operations are gathered. There can be more than just one hidden layer in fact all the layers, excluding the first and the last, are called so. The structure of a neural network is inspired to human brain (though its precise behaviour it is still unknown), in fact the connection between nodes is similar to how neurons are connected with each other. Depending on the task to be performed, the number of hidden layers and nodes can increase, the network with more than just one hidden layer are often referred to **Deep Neural Network (DNN)**. The deeper the network the more complex are the operations that the machine can learn.

As can be seen from figure 1.9, the output of each node, from a specific layer, goes as input in every node in the following one, the layer with this type of configuration is called *Fully-Connected*. This means that all the nodes take every possible information from the previous layer. Considering an image, for example, each node evaluates every pixel in order to elaborate the output.

As mentioned earlier, the nodes in a neural network performs an operation which is basically a function of the weighted sum, these function are called **Activation**

Functions, there are several types of activation functions depending on the type of operation and task the machine has to perform. One example of this function could be the one shown in figure 1.10a. Using the step function, when a certain threshold is surpassed, the neuron fires (the node output is one) otherwise the neuron remains silent (the node output is zero).This type of nodes are generally called *perceptrons*, considering a network based on this network we can still perform some operations however in training phase, since the output could be 0 or 1, small variation in the parameters could alter completely the network behaviour. For example, if the machine correctly classify a certain number of input, updating even a single weight could drastically worsen the performances.

This problem can be solved by using other types of activation function, the most

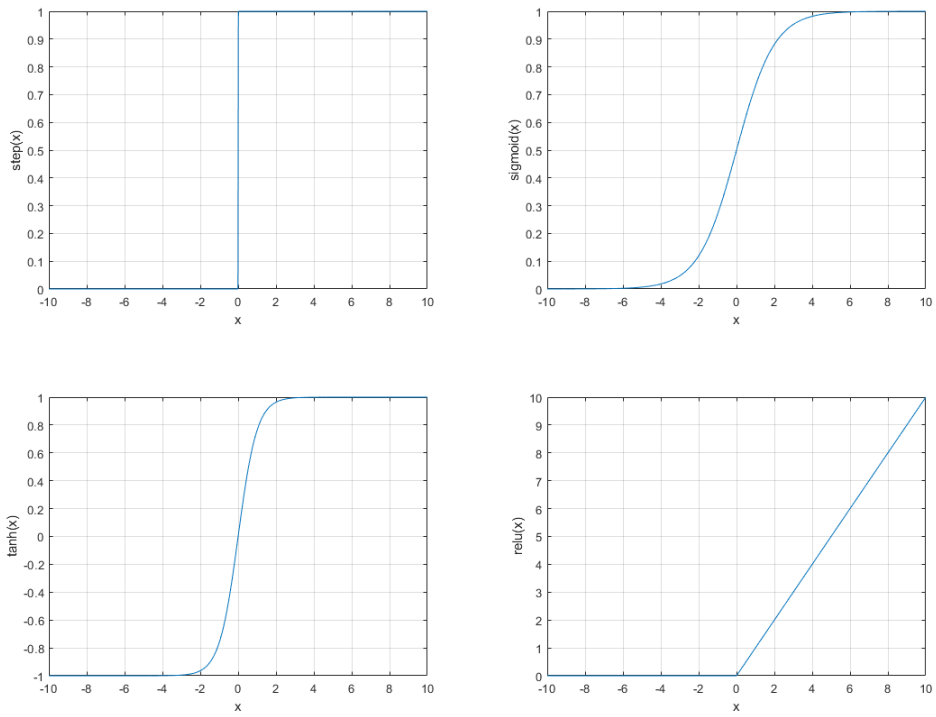


Figure 1.10: Step (top left), Sigmoid (top right), Tanh (bottom left) and ReLU (bottom right) functions

common one is the *sigmoid* function. The formula is the following:

$$y = \frac{1}{1 + e^{-\sum(w_i x_i + b)}} \tag{1.11}$$

its representation is shown in figure 1.10b. With this solution the node output can assume any value between 0 and 1, meaning that it is also disposed to small

variations when the network weights are updated. Since in classification problems the output can either be 0 or 1 then values greater than 0.5 can be considered as True otherwise False.
In the multi-classification case the output nodes are more than one, the node with the greatest output among all will represent the predicted class.
There are other types of activation functions, for example the *tanh* (figure 1.10c) which is similar to the *sigmoid*, the basic *linear* function, the *Rectified Linear Unit (ReLU)* (figure 1.10d) and so on. Regarding the latter, it can be noticed that it is not bounded between 0 and 1 like the others, the range of values is [0, +inf], the particularity of this function is that it eliminates the problem of *vanishing gradient*. Considering the sigmoid gradient is at its maximum when x=0, however for |x|»0 it converges to 0. Since the learning speed depends on the gradient entity, in this condition, the training is significantly slow, the *ReLU* instead has the same gradient independently from the value of x thus the speed is not affected in any condition.
Another important characteristic is that the *ReLU* grants more sparsity in the network activations. Sparsity means that a similar amount of firing and non-firing nodes are present in the network, in fact all the nodes with x<0 have output zero meaning that they do not contribute on the network prediction, on the contrary a network prediction have a dense representation when an high percentage of nodes with non-zero activation are needed to represent the network output.This happens when the *sigmoid* or similar function are used, it is better to have a sparse representation because is more beneficial in terms of computational cost.
The *ReLU* has also some drawbacks, for the nodes with x<0 in fact we have that the gradient is zero, this makes ineffective the gradient descent algorithms for updating the weights, making part of the network passive to the training process. This is referred to as the *dying ReLU*.
To simplify and to better understand the computations inside the neural network, the node *activations* will be indicated as $a_{k_l}^{(l)}$ where l represents the layer number where the node is placed, k_l represent the node number in the l^{th} layer. The weights are organized in matrices which form is $\mathbf{W}^{(l)} \in \mathbf{R}^{k_{l+1} \mathtt{x} k_l}$ where k_l is the number of nodes in the layer, for example $w_{34}^{(2)}$ indicates the weight in the branch connecting the 4^{th} node of the 2^{nd} layer to the 3^{rd} node in the following layer. As an example, considering the network in figure 1.9:

$$\mathbf{W}^{(1)} = \begin{bmatrix} w_{11}^{(1)} & ... & w_{14}^{(1)} \\ . & & . \\ . & ... & . \\ . & & . \\ w_{51}^{(1)} & ... & w_{54}^{(1)} \end{bmatrix}_{k_2 \mathtt{x} k_1} \qquad a^{(1)} = \begin{bmatrix} a_1^{(1)} \\ . \\ . \\ . \\ a_4^{(1)} \end{bmatrix} = \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_4 \end{bmatrix}_{k_1 \mathtt{x} 1} \tag{1.12}$$

the activations in the 2-nd layer ($a_{k_l}^{(2)}$) are computed as follows:

$$a^{(2)} = f\left(\mathbf{W}^{(1)} * a^{(1)}\right) = \begin{bmatrix} a_1^{(2)} \\ . \\ . \\ . \\ a_5^{(2)} \end{bmatrix}_{k_2 \mathtt{x} 1} \tag{1.13}$$

1.3.1 Backpropagation algorithm

In neural network, due to the presence of multiple nodes and layers, the application of the gradient descent algorithm becomes more complex, in particular the overall error from every node must be considered in order to update weights and biases. Normally, in neural network, the input from the first layer is forwarded to the last layer, this operation is also referred as ***Inference***.
There exist another operation where the error from the output result is carried backward, this is done in the training phase in order to correctly update the weights, the algorithm which performs this operation is called ***Backpropagation***.
The objective is to minimize the output cost function. In Artificial Neural Network we have different output nodes, therefore we must consider the error committed in each one of them. Considering we have L layers in the network, the cost function is depends of the activations on the output layer ($a^{(L)}$). The results in the output layer depend, in turn, on all the weights ($\mathbf{W}$) present:

$$C(a^L) = C(\mathbf{W}) = \frac{1}{2}\sum_j (y_j - a_j^{(L)}) \tag{1.14}$$

where j represents the j^{th} input samples, and y_j the expected output on that sample. In order to correctly update weights and biases, we must define the corresponding gradient for each of these parameters:

$$\begin{aligned} \Delta w_{ij}^{(l)} = -\alpha \frac{\partial}{\partial w_{ij}^{(l)}} C(\mathbf{W}) \\ l = 1, ..., L \end{aligned} \tag{1.15}$$

Let's suppose that all the network weights and biases are randomly initialized, when an input sample is propagated to the output the result it is different from the expected one, this means that there will be an error for each node on the last layer. The vector containing these values will be called $\delta^{(l)}$, this also applies to every layer of the network.
In the last section we introduced the *activations* and weight matrices, the error on the last layer can be simply obtained as the difference between the actual output and the expected one:

$$\delta^{(L)} = (a^{(L)} - y) \tag{1.16}$$

Now it is possible to back propagate it to compute the error committed in the other layers:

$$\delta^{(l)} = (\mathbf{W}^{(l)})\delta^{(l+1)} .* f'(\mathbf{W}^{(l-1)}a^{(l-1)}) \tag{1.17}$$

Basically, like in the Inference where the input is multiplied with the weights in the forward propagation, the same thing happens in Backpropagation where, this time, is the error to be multiplied with the weights and the derivative of the activation function being used. The equation (1.15) is the same as (1.6) however every weight

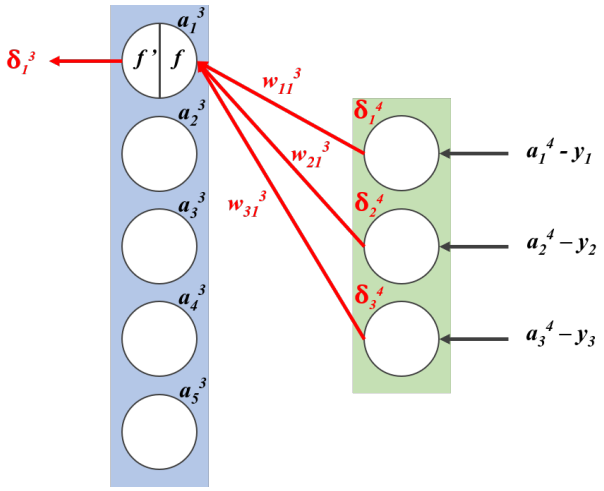


Figure 1.11: Back propagating error

has different effect on the cost function depending on where they are located in the network, in fact, the gradient is generally stronger for weights located to the output layer and tends to have less and less impact for previous layers, this means that in those layers the update have a minor effect. This phenomenon is addressed as ***Vanishing Gradient***. Once the error in every layer has been found, it is possible to its correlation to the cost function with the following simple formula:

$$\frac{\partial}{\partial w_{ij}^{(l)}} C(\mathbf{W}) = a_j^{(l-1)}\delta_i^{(l)} \tag{1.18}$$

In summary the Backpropagation algorithm is performed with the following steps:

1: Input *inference* to obtain the actual output $a^{(L)}$.

2: Output error computation $\delta^{(L)}$ (using 1.16).

3: Error backpropagation to compute the other $\delta^{(l)}$ in every layer (using 1.17).

4: Gradient computation (using 1.18).

5: Weights update (using 1.6)

The backpropagation algorithm is a simple and powerful algorithm used in almost any application in machine learning. As mentioned the major problem is represented with the slow convergence of the Gradient-Descent algorithm, and with an entire network to update this operation takes even more.

1.3.2 Convolutional Neural Network

Convolutional Neural Network (CNN) are networks mainly used for image processing, they are composed by the combination of three different type of layers: **Convolutional Layer**, **Pooling Layer** and **Fully-Connected Layer**.
In the *Fully-Connected* configuration one node from a layer is connected to every node from the previous layer. This layout is the same as the one seen in figure 1.9 and it is the base component of a simple Neural Network.
One node from a *Convolutional Layer*, instead, is not connected to every node from a previous layer but only to a portion of them. Let's consider for example an image, it can be seen as a 3-dimensional matrix (**height** x **width** x **depth**) where *height* and *width* represent, respectively, the number of pixels in the rows and columns and the *depth* represent the number of **Feature Maps** (also called **Channels**). Each matrix element is considered as a node of the input layer in the network. The example in figure 1.12 shows how the input image is processed using

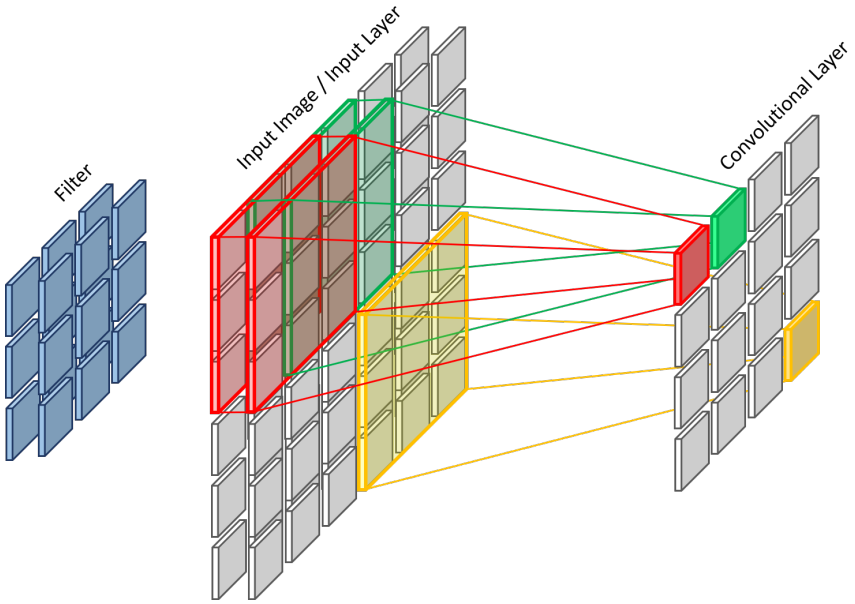


Figure 1.12: Convolution operation, graphic representation

a convolutional layer, for this case the image depth is 2, however it is possible to have different number of channels. The Red, Green and Blue (RGB) components are generally used.

As can be seen each node in the hidden layer computes the weighted sum only for a specific area (called **Receptive Field**), by doing this it is possible to extract useful information concerning the spatial dependency, since pixels close to each other have high correlation.
The weights used in the summation are contained in a matrix (the blue one shown in figure 1.12), called **Filter** or **Kernel**, which depth must match with that of the input image. The same filter is shared for every node in the hidden layer, meaning that every node will apply the same weights for the assigned areas.
Basically we slide the filter matrix in the image matrix and the result of each step is conveyed in one corresponding node. More precisely we start from the red area (in figure 1.12), we compute the weighted sum between image and filter and the result is assigned to one node, then we move to the green area and perform the same operation until the last window, which is the yellow one. This operation is similar to the convolution between two signal where one slides over the other.
It is possible to use more than one *filter* in a convolutional layer, in fact this will determine the number of feature maps to be analysed in the next layer. With different filters it is possible to extract many useful information from the initial image, for example it is possible to highlight the edges of an object and learn its shape. More generally the convolution allows us to convert the input image to another, more complex, representation. These weights are not manually chosen

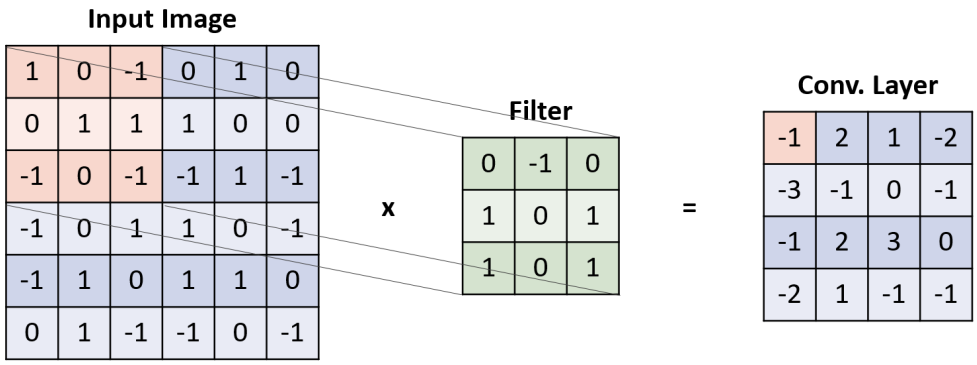


Figure 1.13: Convolution operation, numerical representation

but they are learned, by the machine, during training with backpropagation.
There are two fundamental parameters found in convolutional layer that modify how the convolution in performed, these parameters are the **Stride** and **Zero-Padding**.
The *Stride* defines how many slots the window slides, in case of an image the slots correspond to the pixels. In the example in figure 1.13 the stride is 1, in fact the windows moves by one position both horizontally and vertically. This is

done because, with a small stride, two adjacent window do not provide many new information due to the fact that they are almost overlapped one on the other.
The *Zero-Padding* adds elements to the matrix border with value zero as shown in figure 1.14. By doing this it is possible to modify the dimensionality of the image after the convolution, also it makes possible to extract spatial information on the pixel near the border. Considering the example in figure 1.13 the input image has dimension 6x6x1, since the stride is 1 and there is no zero-padding , the dimension of the convolutional layer will be 4x4x1, by adding one layer of zero-padding the dimensionality of the convolutional layer will become 6x6x1, the same as the input image.
By modifying both this parameters the number of nodes in the convolutional layer

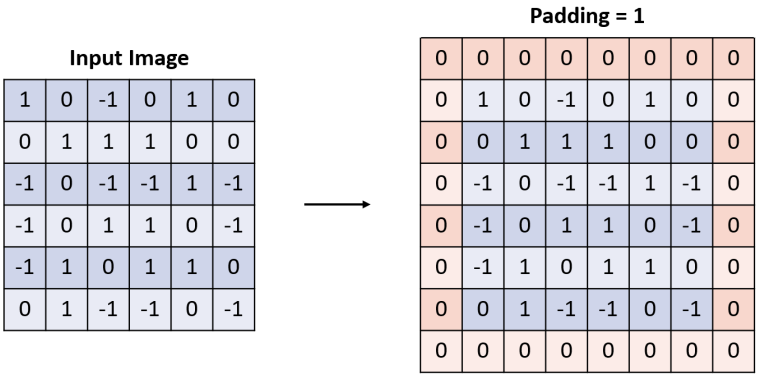


Figure 1.14: One level padding applied to a matrix

change, this could be useful to shrink the number of parameters in the network. Considering that the image has dimension [**IH**x**IW**x**ID**] and the filter then it is possible to compute the layer dimension [**H**x**W**x**D**] obtained after the convolution and, thus, the number of nodes by using the following formula:

$$\begin{aligned} H &= \frac{(IH - FH + 2P)}{S} + 1 \\ W &= \frac{(IW - FW + 2P)}{S} + 1 \\ D &= K \end{aligned} \tag{1.19}$$

where FH represent the filter height, FW the filter width, P the layers of zero-padding added and S the stride. The depth D depends on the number of filter, for example, if K filters are used then $D = K$.
Another important characteristic for convolutional networks is that the number of weights are reduced, since each node connects with only a part of the previous layer. This allows to store less *hyper parameters* in terms of memory and, more importantly, the network is less likely to overfit.

Since the convolutional layer only computes the weighted sum of the input sample, we need to specify the non-linear activation function to apply, the ReLU function is often used.
The pooling layer is often inserted after a convolution, its purpose is to reduce the redundancy of some nodes inside one layer. More precisely it reduces the number of nodes and, thus, the size of the new image representation so that the network requires less parameters and the overfit is controlled. Also, for this layer, the input matrix is divided in windows, the value inside the windows are compared together and one "winner" node is selected above all. The chosen node depends on the type of pooling being used. The **Max Pooling**, for example, selects the node with the highest value and the other are discarded. It is possible to choose the windows size which will determine the number of nodes on the next layer. This operation is shown in figure 1.15.

In the example the window size is set to 2 for both height and width, the result

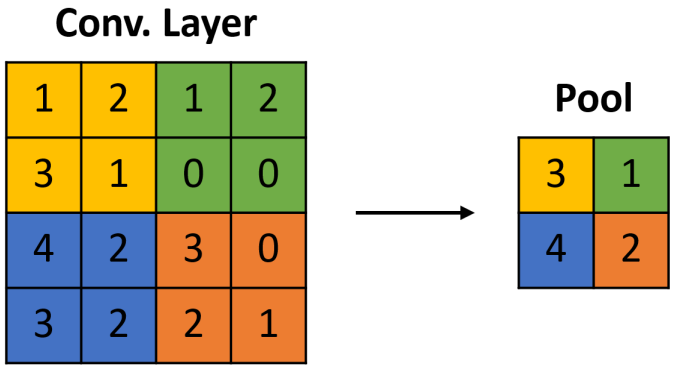


Figure 1.15: Pooling operation, numerical representation

is that the input matrix dimension is halved, more precisely the number of node is determined by the following formula:

$$\begin{aligned} H &= \frac{(IH - FH)}{S} + 1 \\ W &= \frac{(IW - FW)}{S} + 1 \\ D &= ID \end{aligned} \tag{1.20}$$

where FH and FW represent the window height and width. Differently form the convolutional layer there is no zero-padding, and the depth remains the same as the input image, generally the stride coincide with the window size, however it can be changed.
There is another type of pooling method which is the **Average Pool** and it computes the average of every node instead of picking one, they can be both used inside

the network however, in many applications, the max-pooling shows better performances.
By combining all these layer we are able to build an efficient network for image processing, in particular the machine is capable of distinguish and extract complex features to obtain better result in several tasks of classification and object recognition. A simple architecture example of a *Convolutional Neural Network* is shown in figure 1.16.

During the years there have been built different networks and some of them won

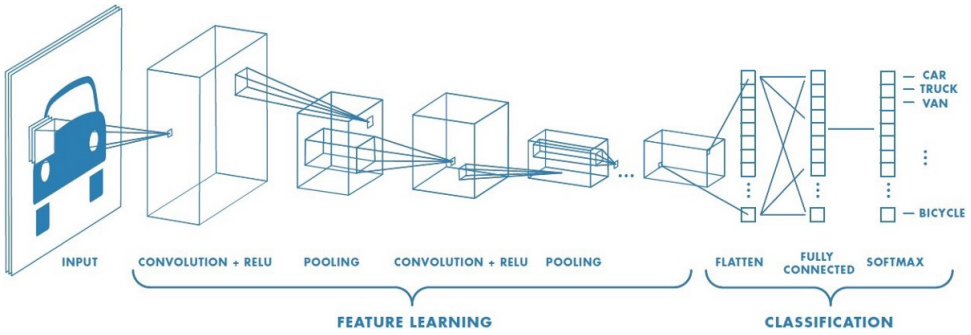


Figure 1.16: Convolutional Neural Network structure. Source: [28]

the ImageNet Challenge [5] due to their good performances, for this reason they are used as standard networks and they are commonly found in many applications.
The first network developed that won the challenge was the **LeNet** [6]. There are different version, the most used is the *LeNet-5* (the number generally indicates the number of layers) which is composed by two convolutional layers each one followed by an average pooling operation and two fully-connected layers.
Over time other standard implementations have been provided, for example the **AlexNet** [7] it shows better performances respect to LeNet due to the fact that it adds more layers and more complexity, also the convolutional layers are divided in two part and stacked together.
Other deeper implementation are the **VGG** [8] with 16 to 19 layers, then the **GoogLeNet** [9] with 22 layers that introduced the "Inception Module" where, in some part of the network, there are different types of layer connected in parallel also, thanks to this module, the number parameters are drastically reduced.

Lastly there is the **ResNet** [10] that achieved the highest performance on the ImageNet challenge, this network introduces "short-cut connection" to contrast the problem of the *Vanishing Gradient*, this happens when the network is too deep, during backpropagation the gradient becomes smaller and smaller and the weights from layers too far away don not get updated. The short-cut connection, shown in figure 1.18, takes the input x and add it to the input mapping $F(x)$ by skipping some layers, this allows to have a better optimization in the learning process since the mapping has a reference with the input, for this reason is called residual

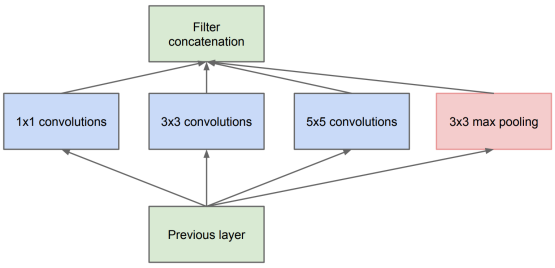


Figure 1.17: Inception Module. Source: [9]

mapping.

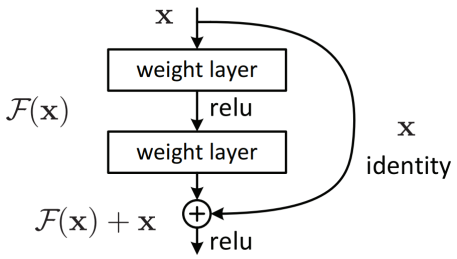


Figure 1.18: Residual Learning through shortcut connection. Source: [10]

1.4 Introduction to datasets

Depending on the task that the machine has to perform, we need to gather data in order to train the network, this is a difficult step when the task is complex. As mentioned in the other sections, to prevent overfitting the amount of data must be big enough. Considering image classification there are a lot of standard dataset provided by research institutes and can be easily obtained and used to fit any application.

The **MNIST** is a dataset of images containing handwritten numbers, from 0 to 9, in black and white colors, and each image has a size of 28x28 pixels. The set has a total of 70.000 samples of which 60.000 are used as *Training* samples, and the remaining 10.000 for the *Testing*. Each sample is associated with the corresponding label, with a total of 10 classes. These numbers were taken by 250 different writers to better differentiate each digit. The MNIST is often used to test the functionality of a network since it is a simple dataset which requires little computational effort. In figure 1.19 are shown some of the digits from the MNIST.

Another important dataset, used in some applications, is the **CIFAR**, there are actually two of them **CIFAR-10** and **CIFAR-100**, where the number indicates

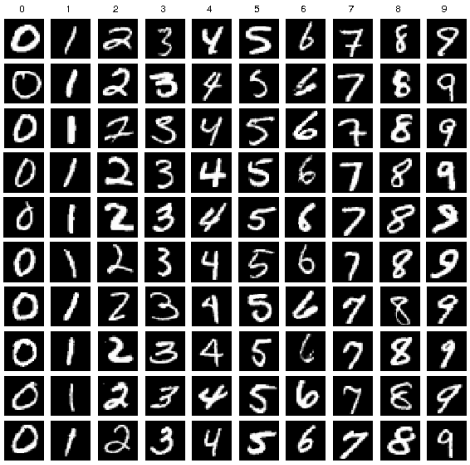


Figure 1.19: Some digits extracted from the MNIST dataset

the total different classes. They are a set of RGB images with size 32x32 pixels, each containing object or animals. The *Training* set is composed of 50.000 images,

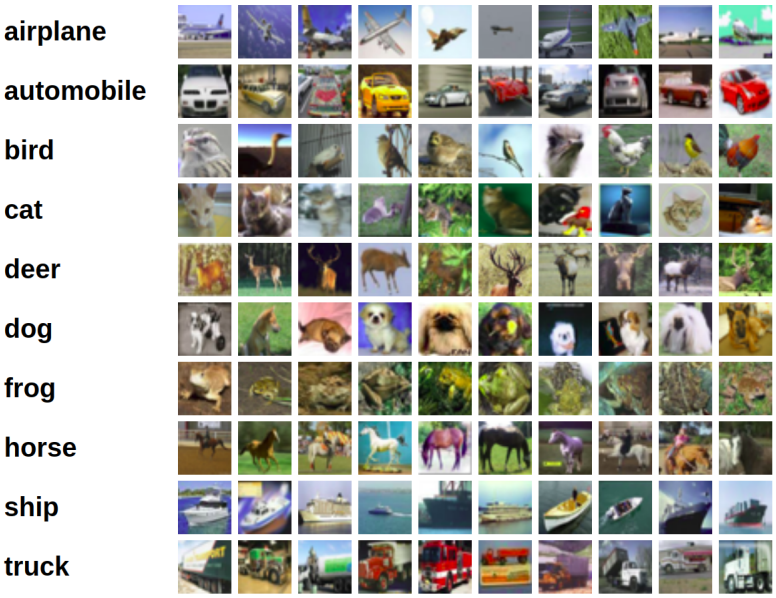


Figure 1.20: Some images extracted from the CIFAR-10 dataset

and the *Testing* set is composed of 10.000, in figure 1.20 are shown some of them, for every class, extracted from the CIFAR-10.
Compared to the MNIST, the CIFAR is more complex because each sample is much different from the others, due to the different colors, backgrounds and details of each element in a class.

One last important dataset is the one related to the **ImageNet Large Scale Visual Recognition Challenge (ILSVRC)**, which is a challenge which is held every year to evaluate the performance of neural networks built by researcher to examine the progress of image recognition.
The dataset contains 10.000.000 images used for training with more than 10.000 different classes, the tesing includes 150.000 photos extracted from *Flickr* site, of which, 50.000 are used as validation set and the remaining as a test set.
The mentioned datasets are for image classification only, however, it is possible to find many other datasets for any kind of application, for both regression and classification problems.

Chapter 2

Extreme Learning Machines

As mentioned the Gradient Descent algorithm, apart from representing one of the most exploited algorithm in the learning process, one of its cons is the time spent training due to the multiple steps required to perform the operation. This problem has been widely addressed and many solutions have been developed using different type of algorithms and different types of networks, in particular Huang et al. [1] proposed an new method to train **Single Layer Feed-forward Network (SLFN)** without the need to tune (adjust) the network parameters. Their algorithm is able to learn the output weights in just one step, leaving the other weights unchanged after being randomly initialized. This allows the network to learn in a small amount of time, moreover this technique shows a better generalization respect to other algorithms.
Basically, considering a neural network with a single hidden layer, the input weights are randomly initialized and the output weights are computed analytically, this algorithm has been called **Extreme Learning Machine (ELM)** due to its extreme speed compared to the traditional gradient descent methods.
Let's assume we have a neural network with a single hidden layer, like the one shown in figure 2.1, in this exposition N will represent the number of input samples $(\mathbf{x}_i,\ \mathbf{t}_i)$ in the dataset, $\mathbf{x}_i = [x_{i1}, x_{i2}, ..., x_{in}]^T \in \boldsymbol{R}^n$, $\mathbf{t}_i = [t_{i1}, t_{i2}, ..., t_{im}]^T \in \boldsymbol{R}^m$, n the number of input features, m the number of output classes, L the nodes in the hidden layer, $\mathbf{w}_j = [w_{j1}, w_{j2}, ..., w_{jn}]^T$ the weights connecting the jth hidden node and all the input nodes, $\boldsymbol{\beta}_j = [\beta_{j1}, \beta_{j2}, ..., \beta_{jm}]^T$ the weights connecting the jth hidden node with the output nodes, b_j is the bias for the jth hidden node and finally $f(x)$ is the activation function for the nodes in the hidden layer. Using these definitions we can model the network output as follows:

$$\sum_{j=1}^{L} \boldsymbol{\beta}_j f(\mathbf{w}_j \cdot \mathbf{x}_i + b_j) = \mathbf{y}_i \qquad i = 1, ..., N \tag{2.1}$$

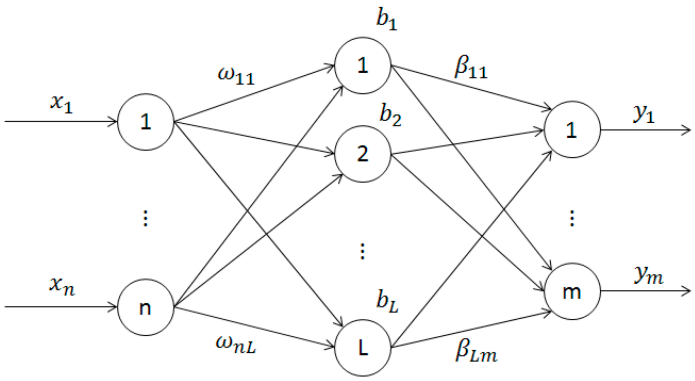


Figure 2.1: Single Layer FeedForward Network

The objective [2] is to find a combination of β_j, w_j and b_j such that the N input samples can be approximated with zero error $\sum_{i=1}^{N} ||\mathbf{y}_i - \mathbf{t}_i|| = 0$, where $\mathbf{y}_i$ represents the actual output and $\mathbf{t}_i$ the expected one, in this case called target output.

$$\sum_{j=1}^{L} \boldsymbol{\beta}_j f(\mathbf{w}_j \cdot \mathbf{x}_i + b_j) = \mathbf{t}_i \tag{2.2}$$

Let's now define the *Hidden Layer Matrix* $\mathbf{H}$ and the *Target Matrix* $\mathbf{T}$ as:

$$\mathbf{H} = \begin{bmatrix} f(\mathbf{w}_1 \cdot \mathbf{x}_1 + b_1) & \dots & f(\mathbf{w}_L \cdot \mathbf{x}_1 + b_1) \\ . & & . \\ . & \dots & . \\ . & & . \\ f(\mathbf{w}_1 \cdot \mathbf{x}_N + b_N) & \dots & f(\mathbf{w}_L \cdot \mathbf{x}_N + b_N) \end{bmatrix}_{N\text{x}L} \quad \mathbf{T} = \begin{bmatrix} \mathbf{t}_1^T \\ . \\ . \\ . \\ \mathbf{t}_N^T \end{bmatrix}_{N\text{x}m} \tag{2.3}$$

then we can rewrite 2.2 as:

$$\mathbf{H}\beta = \mathbf{T} \tag{2.4}$$

As stated in [1], given an SLFN with L hidden nodes and activation function $f(x) : \mathbf{R} \to \mathbf{R}$ infinitely differentiable, then the network is able to learn at most L distinct input samples with zero error.
With this assumptions it can be proved that, for any w_i and b_i randomly chosen from any interval $\mathbf{R}^n$ and $\mathbf{R}$ respectively, the matrix $\mathbf{H}$ is invertible meaning that the condition $||\mathbf{H}\beta - \mathbf{T}|| = 0$ can be surely satisfied.
Given the fact that in most cases L«N and using the same conditions as before, then choosing any small value $\epsilon > 0$ there exists β such that $||\mathbf{H}\beta - \mathbf{T}|| < \epsilon$ is satisfied. This means that, even with a few nodes in the hidden layer, the network is able to approximate the input sample with a small error, just by computing the output weights leaving the other randomly initialized parameters unchanged. The objective, however, is to find a specific combination that minimizes this error.

It is fundamental now to understand how to compute the output weights β.
Let's start considering L = N, in this case the matrix $\mathbf{H}$ is square and invertible, as said before the training sample can be approximated without errors, also the output weights can be computed just by using the following formula, derived from 2.4:

$$\beta = \mathbf{H}^{-1}\mathbf{T} \tag{2.5}$$

It is common however that the number of training samples is much greater respect to the number of hidden nodes (L « N), the matrix $\mathbf{H}$ in this case in non-square and the smallest norm solution can be found applying this other formula:

$$\widehat{\beta} = \mathbf{H}^{\dagger}\mathbf{T} \tag{2.6}$$

the operation $\mathbf{H}^{\dagger}$ is called *Moore-Penrose generalized inverse* and it is equivalent to:

$$\mathbf{H}^{\dagger} = (\mathbf{H}^{T}\mathbf{H})^{-1}\mathbf{H}^{T} \tag{2.7}$$

The output weights matrix $\widehat{\beta}$ from 2.6, which is different from β, indicates the solution of the equation which gives the smallest error, this also means that the algorithm always brings to a global minimum:

$$\left|\left|\mathbf{H}\widehat{\beta} - \mathbf{T}\right|\right| = \min_{\beta} ||\mathbf{H}\beta - \mathbf{T}|| \tag{2.8}$$

this is an important property of extreme learning machines. For traditional training method in order to reach the minimum many steps are required and in most cases gradient descent algorithm leads to local minimum, in this case instead one only step is required, also the solution found with 2.6 is unique.
There are several ways to compute the Moore-Penrose pseudo-inverse, the simplest is through *Orthogonal Projection* method thus computing the matrix multiplication as shown in 2.7, however there might be some cases where the matrix $\mathbf{H}^{T}\mathbf{H}$ is not invertible due to some singularities. To overcome this problem *Singular Value Decomposition* (SVD) method can be used, where the $\mathbf{H}$ matrix is decomposed in three distinct matrices $\mathbf{H} = \mathbf{V}\mathbf{\Sigma}\mathbf{U}^{*}$ where $\mathbf{\Sigma}$ is diagonal and both $\mathbf{V}$ and $\mathbf{U}$ are unitary square matrices, now the it is possible to compute the pseudo-inverse as $\mathbf{H}^{\dagger} = \mathbf{V}\mathbf{\Sigma}^{\dagger}\mathbf{U}^{*}$.
In gradient descent algorithm the a correct *learning rate* value is fundamental and the choice is not immediate but depends on many factors, for these reason several validation tests must be done before reaching a good compromise. Also the number of *epochs*, the *momentum*, the *batch size* and the *penalty coefficient* are the other hyper parameters to keep in consideration when dealing with this algorithm. In fact, another important property of Extreme Learning Machines is the number of hyper parameters that must be set up before the training phase, in this case we just need the number of nodes in the hidden layer.
In the previous sections we introduced the overfitting problem and how it is important to minimize not only the training error but also the weights learned in order

to better optimize the network and improve generalization, for Extreme Learning Machine the objective is the same but applies to the minimization of output weights since the other are randomly initialized, this is equivalent of solving the following problem [11]:

$$\begin{aligned} Minimize: \ L_{P_{ELM}} &= \frac{1}{2}\left|\left|\boldsymbol{\beta}\right|\right|^2 + C\frac{1}{2}\sum_{i=1}^{N}\left|\left|\boldsymbol{\xi_i}\right|\right|^2 \\ Subjected\ to: \ \mathbf{h}\left(\mathbf{x}_i\right)\boldsymbol{\beta} &= \mathbf{t}_i^T - \boldsymbol{\xi}_i^T, \qquad i = 1, ..., N \end{aligned} \tag{2.9}$$

where $\mathbf{h}(\mathbf{x}_i)$ is equivalent to $f(\mathbf{w}\mathbf{x}_i + b)$, and $f(x)$ is the non-linear function applied to the weighted sum. The parameter C is similar to the λ in the regularization formula 1.9, it is used as a trade off for minimizing both training error and weight values. As we said $\mathbf{t}_i = [t_{i1}, ..., t_{im}]$ represents the target vector, the index i refers to the i^{th} input sample out of the total N, m is the number of classes and, thus, the number of output nodes.
The elements in the target vector are all zeros except the value in the position corresponding to the class to which the input sample belongs, for example, if the inputs is labelled as the 5^{th} class, then the vector has 1 in the 5^{th} position and 0 everywhere else. This is basically equivalent to the one hot encoding where every state is distinguished by the position of 1 in the vector.
By using the Karush-Kuhn-Tucker (KKT) theorem [12] and obtaining the optimality conditions, we can solve the problem in the equation 2.9 and, depending on the number of training sample respect to the number of hidden nodes, we have two possible solution to find the output weights $\boldsymbol{\beta}$:
When the number of Training Sample is smaller or comparable with the number of Hidden Nodes then we can apply the following equation:

$$\boldsymbol{\beta} = \mathbf{H}^T\left(\frac{\mathbf{I}}{C} + \mathbf{H}\mathbf{H}^T\right)^{-1}\mathbf{T} \qquad N \leq L \tag{2.10}$$

where $\mathbf{I}$ is the *Identity* matrix, while $\boldsymbol{\beta}$, $\mathbf{H}$ and $\mathbf{T}$ are the same matrices shown in (2.3).
hen the number of Training Sample is much bigger than the number of Hidden Nodes then this other equation is used:

$$\boldsymbol{\beta} = \left(\frac{\mathbf{I}}{C} + \mathbf{H}^T\mathbf{H}\right)^{-1}\mathbf{H}^T\mathbf{T} \qquad N \gg L \tag{2.11}$$

In general it is possible to use either both the solution in any application since they are equivalent, however, the computational cost of these two approaches is different and must be taken into account, considering, for example, the dimension of the two matrices $A = \mathbf{H}^T\mathbf{H}$ $[L \cdot L]$ and $B = \mathbf{H}\mathbf{H}^T$ $[N \cdot N]$, we have that for a large dataset ($N \gg L$) computing the inverse of B requires a big amount of time

respect to the inverse of A, they both provide the same result but the complexity is different. Now it is possible to determine the steps required to run the algorithm of an ELM but first we define the network, this means we need to define the number of nodes L and the activation function $f(x)$ in the hidden layer, also we suppose to have a training set composed of N samples ($\mathbf{x}_i$, $\mathbf{t}_i$):

Step 1: Random input weights $\mathbf{w}_j$ and biases $\mathbf{b}_j$ initialization , with $j = 1, ..., L$

Step 2: Computation of the hidden layer matrix $\mathbf{H}$

Step 3: Pseudo-inverse application $\mathbf{H}^{\dagger}$

Step 4: Output weights $\boldsymbol{\beta}$ computation

Considering the classification problem we may have two situation, single-class and multi-class tasks, for a single-class the network has a one output node and, in order to evaluate the performances, we look at the sign of the result to decide whether the input sample is a positive or a negative match.
For the multi-class task instead we have as many output nodes as the number of classes to distinguish, in this case every output node will provide its result and the chosen class will be decided by the node with the highest value among all, more precisely:

$$\begin{aligned} \mathbf{y}(\mathbf{x}_i) = \mathbf{h}(\mathbf{x}_i)\boldsymbol{\beta} = \mathbf{h}(\mathbf{x}_i) \left(\left(\frac{\mathbf{I}}{C} + \mathbf{H}^T\mathbf{H} \right)^{-1} \mathbf{H}^T\mathbf{T} \right) \\ prediction(\mathbf{x}_i) = max\left(\mathbf{y}(\mathbf{x}_i)\right) \end{aligned} \tag{2.12}$$

where $\mathbf{y}(\mathbf{x}_i)$ is the output vector where each element contain the result of each node in the layer.

2.1 Base implementations of ELM

2.1.1 Setting up the environment

Before proceeding to implement a simple ELM and study its behaviour and characteristics, we first need a brief introduction about the programming environment. There are several programming languages used to build learning machines and the choice depends on many factors, like the reliability, the accessibility, the simplicity and so on. Among all, the most used is **Python**.
It is an high-level language which supports object-oriented programming, it is very accessible and easy to learn. There are all kinds of library and many of them are specifically developed for machine learning and artificial intelligence.
Other valid choices could be **JavaScript**, **C++**, **C#**, **R** however, for this project, we are going to develop in *Python* and **PyCharm** will be used as the *Integrated Development Environment (IDE).*

To approach machine learning in python it is important to choose the library for the development, the available options are **Numpy** used for complex matrix operations, **Matplotlib** for data visualization, **Pandas** for data analysis, **TensorFlow** works with *tensors* which are generalization of vectors and matrices, easily managed by *Graphic Processing Units (GPU)*.
For building up neural network we have **scikit-learn** working on *Numpy*, **Keras** based on *TensorFlow* and **Pytorch** which is based on **Torch**, a library developed in *Lua* programming language, which also uses tensor for the various computations. *Keras* it is commonly used, however we chose **Pytorch** since it is more flexible and makes many operations easier.

2.1.2 Simulations and results

We start now implementing the algorithm exposed in the last section, analyzing its behaviour and performances on a dataset.
As said we build the network using the library *Pytorch*, more precisely we use the *torch.nn* class to create the various layers.

```
class ELM(nn.Module):

  def __init__(self):
    super(ELM, self).__init__()
    self.hidden_layer = nn.Linear(input_nodes, hidden_nodes)
    self.output_layer = nn.Linear(hidden_nodes, classes, bias=False)

  def forward(self, x):
    x = x.view(x.size(0), -1)
    h = torch.sigmoid(self.hidden_layer(x))
    x = self.output_layer(h)
    return h, x
```

With this code section we create a class and define a neural network with one fully-connected hidden layer and one fully-connected output layer. We also specify the number of nodes that they need. Notice that the output layer does not have a bias vector because the ELM does not require it. The class does not know a priori how the two layers are connected or how the inference is performed, for this reason we need the *forward(x)* method where we also insert the activation function to be used in the hidden layer.
The number of input nodes depends on the dataset used, for example, the MNIST needs 784 nodes since the image has 28x28 pixels, for the CIFAR-10 instead we need 3072=3x32x32 beacuse, other than the 32x32 pixels, it also have the three color components RGB as feature maps.
The number of hidden nodes, instead, are not chosen with a precise method, generally we start with a reasonable value, then we look at the performances and see if any other value improve them.

The number of output nodes are the same as the number of classes in the dataset, for both MNIST and CIFAR-10 we have 10 classes.
Now we can move to the next step which is implementing all the computation. The first thing to do is random initialize the weights and biases in the hidden layer, this operation can be simply achieved using the *torch.Tensor.uniform_(-1,1)* method which samples the data from an **Uniform Distribution**.

```
w = elm.hidden_layer.weight
w.data.uniform_(-1, 1)
b = elm.hidden_layer.bias
b.data.uniform_(0, 1)
```

elm.hidden_layer.weight and *elm.hidden_layer.bias* are needed to access, respectively, the weights and biases from the specified layer, then we use **w** and **b** to copy the data obtained from the uniform distribution and insert them to the network parameters. The range of the has been set from -1 to 1, different initializations may provide different results.
Now we can start with the training phase:

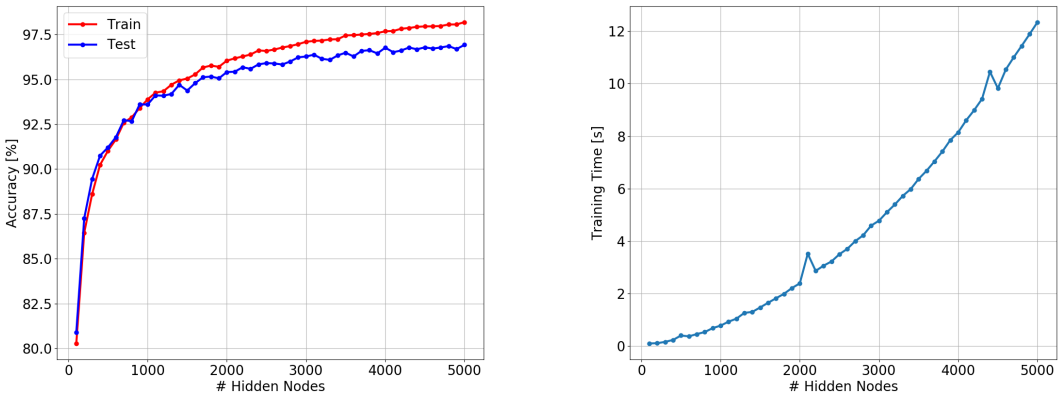
```
C = 0.01
classes = 10
I = torch.eye(hidden_nodes, hidden_nodes)
T = torch.eye(classes)[t]
H, _ = elm.forward(X)
M = I/C + torch.mm(H.t(), H)
P = torch.mm(H.t(), T)
M = torch.inverse(M)
B = torch.mm(M, P)
```

As for the hidden nodes, the regularization parameter C is chosen depending on the performances, for example starting without regularization ($C \gg 1$) and then decreasing the value.
The tensor **I** is the *Identity* matrix seen in the equation (2.9), **X** is the input matrix with dimension [N x (image_size)] where each row represent one input sample (with a total of N samples) and the elements in the column contain the value for each pixel in the corresponding image. The matrix **T** is the target matrix [N x c], where c is the number of classes.
The pytorch library *torchvision* provide some standard datasets including MNIST and CIFAR. The input samples are organized in a four dimensional tensor with [batch_size x depth x width x height], supposing that the batch size corresponds to the number of sample in the dataset (N), we have [N x depth x width x height]. In order to meet the requirements for the matrix **X** we have to reshape the 4D tensor into a 2D [N x (depth $\cdot$ width $\cdot$ height)], in fact the first line in the *forward(x)* method in the ELM class performs this operation.
Torchvision also provides the labels as integer for each sample, meaning that we have a vector (indicated as t in the code section) of N elements, so we need to use

the one hot encoding to create the required target matrix **T**. This can be done by exploiting the *torch.eye()* function, used declare the identity matrix, and providing the position of the ones in the matrix ($T = torch.eye(classes)[t]$).
The matrix **H** is extracted computing the activations in the hidden layer, without considering the output of the network. Then the identity matrix, scaled by the value C, is summed with the result of the matrix multiplication between **H** transposed and **H**, storing the computation in the matrix **M**. Then another multiplication is performed between **H** transposed and **T** storing the result in **P**. Finally, the **M** matrix is inverted and multiplied with **P** obtaining the output weights which are substituted with the current output weights present in the network.
As we said, in order to understand if the classification with multiple classes is correct, we check the maximum results between the output nodes and we compare the index corresponding to the node with the label provided by the dataset, if they coincide then the classification is correct.

```
correct = 0
_, output = elm(X)
prediction = output.data.max(1)[1]
correct += pred.eq(target.data).sum()
accuracy = 100. * correct.item() / len(loader.dataset)
```

Now let's analyze the performances obtained with the MNIST dataset, more precisely let's compute the accuracy, for both training and test sets, and the training time varying some parameters in the network.
We first set the parameter C to an high value ($C = 1 * 10^8$), meaning that there is a negligible regularization in the final computation, then we start with 100 nodes in the hidden layer and perform the training measuring the time to perform the algorithm. Once the training is completed we compute both the train and test accuracy and then we increase by 100 number of nodes and repeat the entire operation, until 5000 are reached.
All the computations have been carried out using an Intel Core i7-7700HQ CPU (2.80 GHz). As can be seen in figure 2.2a, with only 100 nodes the accuracy, on the MNIST dataset, surpasses the 80%. At the beginning there is a sharp increase which flatten from 1000 nodes onwards. It can be noticed that both the accuracies are initially close to each other, however, too many nodes leads to ovefitting, in fact, the test accuracy converges even more than the training accuracy so their difference starts to increase more and more.
Regarding the training time, shown in figure 2.2b, with 5000 nodes the training is completed within 19 seconds, since the accuracy with 2000 nodes is not that different with the one obtained with 5000 we can choose a smaller number of nodes so that we take even less amount of time.
Another thing to notice about the training time is the function shape, which has a quadratic trend. This problem is attributable to the inverse computation, it is the most complex operation in the algorithm so it takes more times respect to the



(a) Train and Test accuracies

(b) Training time

Figure 2.2: Performances on MNIST dataset varying the number of nodes in the hidden layer

other computations. The inverse is applied on the matrix given by the result of the multiplication between $\mathbf{H}^T$ and $\mathbf{H}$, the dimension of this matrix is [hidden_nodes x hidden_nodes], this means that if the nodes are doubled the matrix elements quadruple. In figure 2.3 it is shown the confusion matrix for the MNIST case,

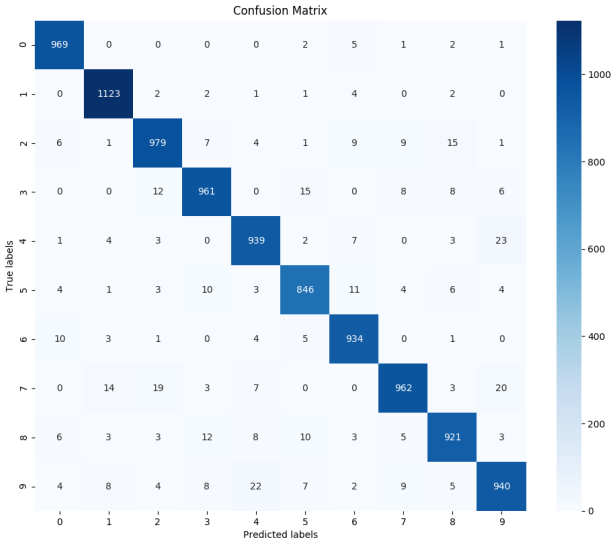


Figure 2.3: Confusion matrix on the test set (MNIST)

evaluated on the test set and a network with 2000 hidden nodes.
The y axis contains the true labels and the x axis the predicted ones, every element in the diagonal is a correct prediction, all the others are mistakes. From this matrix it is easy to realize where the machine misses the most, in fact the digit '4' has been mistaken 23 times with the '9' digits since the shape is similar respect to every other number, the opposite is also true where digit '9' has been mistaken 22 times with the '4'.
As said the computations where executed by the CPU, however, it is possible to perform the training using the Graphics Processing Unit (GPU), they are basically CPUs but they have a lot more processors. The processors in the GPU are less powerful respect to the CPU, however, it is possible to perform many operation in parallel drastically reducing the amount of training time. In order to do this we

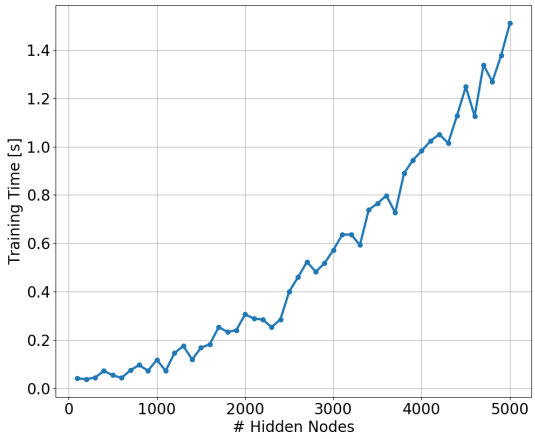
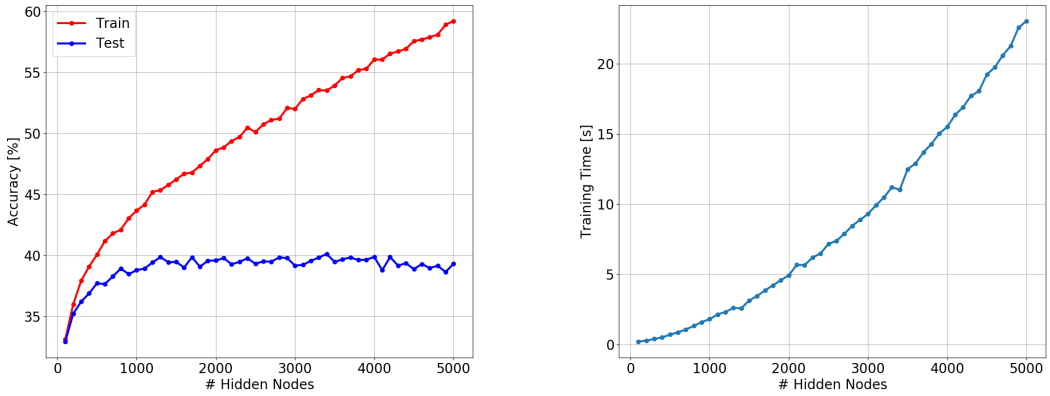


Figure 2.4: Training time using GPU

exploit the CUDA toolkit for python. **Compute Unified Device Architecture (CUDA)** is an architecture developed by **NVIDIA** which allows GPU to be used as general purpose processors with high parallelization, this means that all the operations we need to perform are delivered to the GPU which will perform them in a short amount of time. Basically all the tensors are passed to the GPU that will handle every operation, returning then the result. Pytorch with the CUDA toolkit allows to manually pass every variable to the GPU RAM using the *torch.cuda()* method. It is important that all the variable are correctly passed to the GPU otherwise there will not be compatible. Figure 2.4 shows how the time required, to train the same network, has been drastically reduced by the GPU respect to the previous case with the CPU, the results have been obtained using an NVIDIA GeForce GTX 1050 graphic card, with 2GB RAM.
The same tests has been carried out also for the CIFAR10 dataset exploiting the

CPU. Just by looking at figure 2.5a, it clearly shows different performances, the test accuracy converges to a value less than 40% even when the number of nodes is high, while the training accuracy linearly increases, the overfitting here is imposing and increases even more adding many nodes. This shows how the CIFAR-10 is a

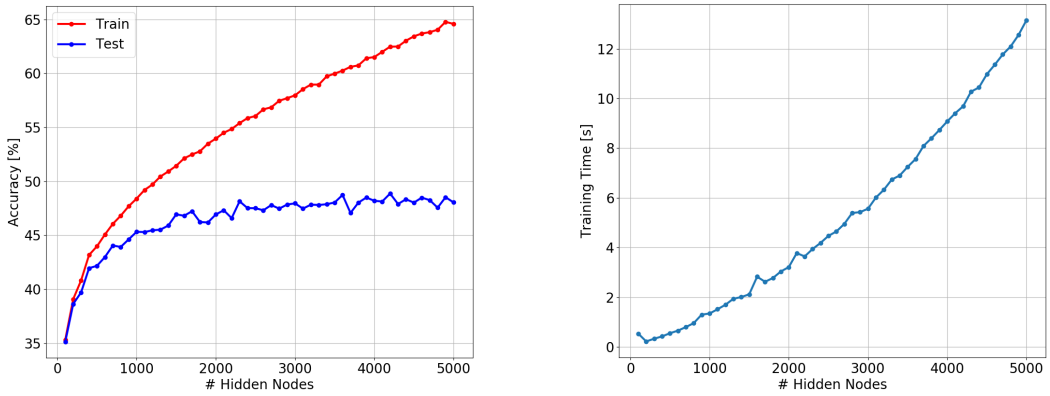


(a) Train and Test accuracies

(b) Training time

Figure 2.5: Performances on CIFAR10 dataset varying the number of nodes in the hidden layer

more complex dataset respect to the MNIST and a single hidden layer isn't just enough to obtain a good accuracy, for this reason we need to extend the network and enhance its capability. One way to do this is by exploiting Convolutional Neural Networks (CNN). The training time is obviously increased due to the fact that each image in the CIFAR-10 dataset has 32x32x3 pixels instead of 28x28 in the MNIST, so the time required is higher, while the shape has remained the same for the reason exposed earlier.
The performances in figure 2.6 have been obtained using the activation function *ReLU* instead of the *Sigmoid*. As mentioned, it grants more sparsity and reduces the effect of vanishing gradient. In fact, the accuracies have increased by 7%. Also we can see how the overfitting decreased respect to the previous case. Since computing the ReLU function is easier, the training time also decreased. The confusion matrix for the CIFAR-10 is show in figure 2.7, we can clearly see which classes make the predictions difficult for the machine. The "truck" for example is mistaken a lot with the car, but more evident is the diagonal from the "bird" to the "dog" which is not much highlighted. For the animals there are a lot of different species and characteristics which makes the classification even more difficult. For this reason we need to extract more complex features and to do this we implement the CNN.
The ELM algorithm is known for its speed respect to the standard gradient-based algorithm, in this regard we trained two networks, one with backpropagation and



(a) Train and Test accuracies

(b) Training time

Figure 2.6: Performances on CIFAR10 dataset varying the number of nodes in the hidden layer, with the ReLU activation function

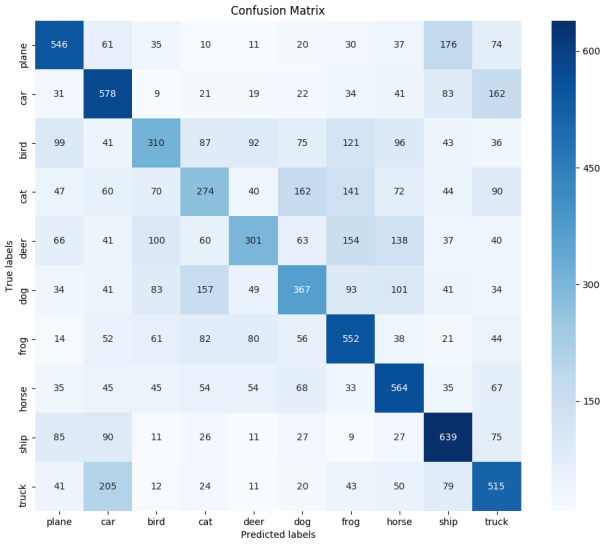


Figure 2.7: Confusion matrix on the test set (CIFAR-10)

the other with extreme learning. Both networks have 2000 nodes in the hidden layer. For backpropagation we set the learning rate to 0.005, the momentum to 0.9 and weight decay to $5 \cdot 10^{-4}$. The ELM parameters are the same used for the MNIST dataset implemetiation. We measured the training time and obtained: Extreme Learning Machine algorithm = 3.90 seconds

Backpropagation algorithm = 74.35 seconds.

2.1.3 Transfer Learning using Extreme Learning Machines

Convolutional Neural Networks are composed by combinations of different types of layers. More precisely, the convolutional and the pooling ones are used for **features extraction** and they are placed in the first part of the network, the last layers are generally fully-connected and they are used as **classifiers**, they basically elaborate the features extracted by the convolutional layers and generate the final predictions.
The CNN are trained using Backpropagation with Stochastic Gradient Descent (SGD) algorithms. The main problem are the local minimum of the cost function and during training we may end up in those points. Extreme learning machine have the capability to train SLFN always providing the smallest least square solution, bringing to global minimum, however it is important to have quality features extraction in the CNN side, otherwise the performance are not maximized.
The solution is, thus, training the SLFN classifier of the CNN with ELM algorithm

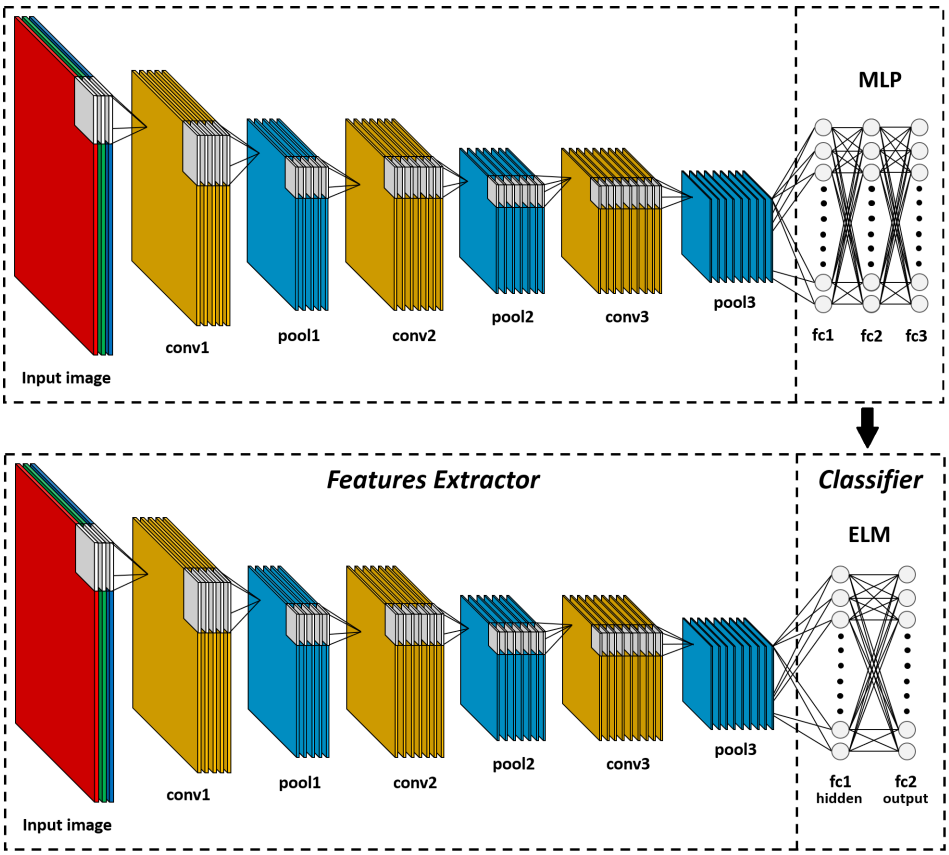


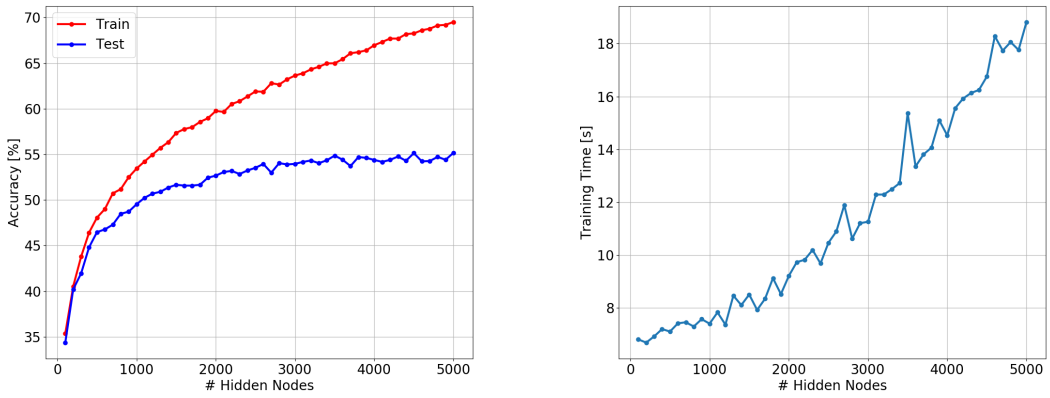
Figure 2.8: Convolutional Neural Network trained with Extreme Learning Machine

to obtain better generalization.
The training process is organized in different phases, first backpropagation is applied to the CNN until convergence, then the fully-connected layers are removed from the CNN and the ELM is inserted along (figure 2.8). Finally the ELM is trained to classify the image with the features it receives, giving better accuracies respect to the fully-connected layers trained with backpropagation.
Training deep CNN, however, requires a lot of time, for this reason there are some pre-trained CNN available on complex dataset (for example the ImageNet). These pre-trained model can be used is several ways. We can, for example, remove the fully-connected layers and use only the CNN part as a features extractor with fixed weights, then train the new classifier in the dataset we need for the application. We could also perform *fine-tuning* on the CNN instead of keeping the weights frozen, so that the network specializes on extracting useful informations from the new dataset. These operations are often referred as **Transfer Learning**, because we exploit features learned in another process (on another more complex dataset) for our training. If the dataset used in a certain implementation is similar to the one of the pre-trained model, then the extracted features will be more effective.
To see how CNN improve the accuracy we implement a simple neural network with two convolutional layer each followed by a pooling layer (max-pool) and two fully connected layers:

```
class CNNELM(nn.Module):
  def __init__(self):
    super(CNNELM, self).__init__()
    self.pool_layer = nn.MaxPool2d(2, 2)
    self.conv_layer_1 = nn.Conv2d(3, 20, 5)
    self.conv_layer_2 = nn.Conv2d(20, 50, 5)
    self.hidden_layer = nn.Linear(5*5*50, 2000)
    self.output_layer = nn.Linear(2000, 10, bias=False)

  def forward(self, x):
    x = self.pool(F.relu(self.conv1(x)))
    x = self.pool(F.relu(self.conv2(x)))
    x = x.view(x.size(0), -1)
    h = F.relu(self.fch(x))
    x = self.fco(h)
    return h, x
```

the arguments in *nn.Conv2d(in_features, out_features, kernel_size)* specify respectively the number of input feature (for the first convolutional layer since we are going to use the CIFAR-10 dataset we need to set this parameter to 3), then the number of output feature corresponding to the number of filters (or kernels) we want to use and, lastly, the size (both height and width) of the filters.
For this implementation all the weights in the CNN are randomly initialized and fixed, the training will be performed on the ELM classifier only exploiting the CPU.



(a) Train and Test accuracies

(b) Training time

Figure 2.9: CNN-ELM performances

In figure 2.9 we can see that even with random features extraction we have an improvement in accuracy by 7% respect to the implementation with the ReLU (figure 2.6) and 15% for the implementation with Sigmoid (figure 2.5). This also depends on the number of feature maps (filters), in fact, considering that the same number of kernels are used for both the convolutional layers in the network, by changing this number we obtain different results, as shown in 2.10 There are several appli-

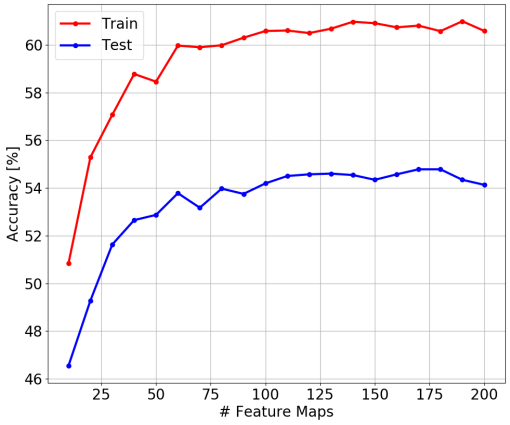


Figure 2.10: Train and Test accuracies varying the number of feature maps

cations where transfer learning is used with ELM [20]-[24], these network are often referred as CNN-ELM. Their configurations may, however, change depending on the applications.

2.2 Online-Sequential and Data Parallelization

In this section we will examine one of the main problem of extreme learning machines. Their central operation is the pseudo-inverse which requires the hidden layer matrix $\mathbf{H}$ and the target matrix $\mathbf{T}$. As said the dimension of these matrices are respectively $[\mathrm{N} \cdot \mathrm{L}]$ and $[\mathrm{N} \cdot \mathrm{c}]$, where N it the number of samples in the dataset, L is the number of nodes in the hidden layer and c is the number of output classes. In order to compute the output weights all the samples in the dataset needs to be available but this requirement is difficult to satisfy due to the fact that, in most cases, the training data are provided in batches and, for **Online Machine Learning**, the data are given sequentially discarding each step the previous ones. The amount of elements to store in memory becomes unmanageable if the dataset is to vast and complex.
For this reason it has been developed [13] the **Online Sequential - Extreme Learning Machine (OS-ELM)**, capable of training the network with data given one by one or in batches. The idea is to divide the $\mathbf{H}$ and $\mathbf{T}$ in multiple parts and compute the output weights:

$$\mathbf{H} = \begin{bmatrix} \mathbf{H}_0 \\ \mathbf{H}_1 \end{bmatrix} \qquad \mathbf{T} = \begin{bmatrix} \mathbf{T}_0 \\ \mathbf{T}_1 \end{bmatrix} \tag{2.13}$$

Let's consider that in the previous step we computed the output weight for the first batch $\mathbf{H}_0$ and $\mathbf{T}_0$:

$$\beta_0 = \left(\mathbf{H}_0^{\mathrm{T}}\mathbf{H}_0\right)^{-1}\mathbf{H}_0^{\mathrm{T}}\mathbf{T}_0 \tag{2.14}$$

To compute the output weights inserting also the batches $\mathbf{H}_1$ and $\mathbf{T}_1$ we have to theoretically apply:

$$\beta_1 = \left(\begin{bmatrix} \mathbf{H}_0 \\ \mathbf{H}_1 \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} \mathbf{H}_0 \\ \mathbf{H}_1 \end{bmatrix}\right)^{-1} \begin{bmatrix} \mathbf{H}_0 \\ \mathbf{H}_1 \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} \mathbf{T}_0 \\ \mathbf{T}_1 \end{bmatrix} \tag{2.15}$$

however, we need to find another way to compute β_1 without using the previous batches $\mathbf{H}_0$ and $\mathbf{T}_0$, but using β_0 and the new batches only. After a few steps [13], it is possible to find that:

$$\beta_1 = \beta_0 + \mathbf{K}_1^{-1}\mathbf{H}_1^{\mathrm{T}}\left(\mathbf{T}_1 - \mathbf{H}_1\beta_0\right) \tag{2.16}$$

where:

$$\mathbf{K}_1 = \begin{bmatrix} \mathbf{H}_0^T & \mathbf{H}_1^T \end{bmatrix} \begin{bmatrix} \mathbf{H}_0 \\ \mathbf{H}_1 \end{bmatrix} = \mathbf{K}_0 + \mathbf{H}_1^T\mathbf{H}_1 \tag{2.17}$$

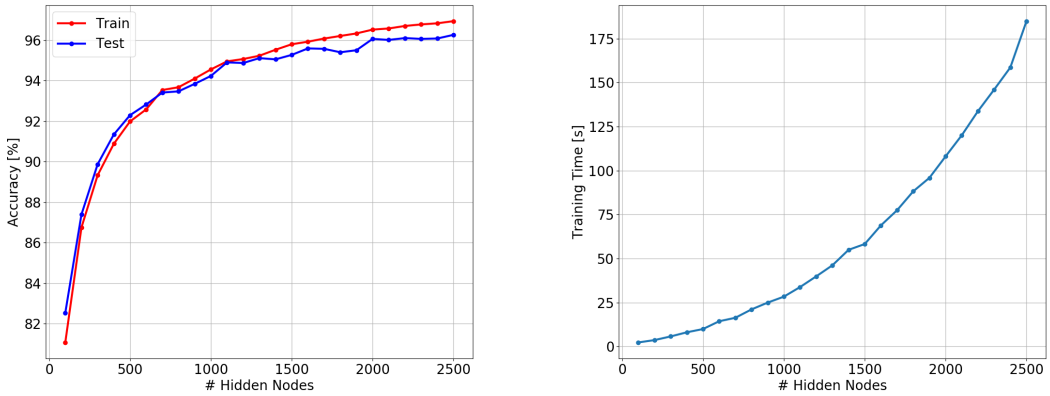
Let's suppose that we already computed β_{k} for k batches and that the (k+1) batch is received. To find the total output weights $\beta_{\mathrm{k+1}}$ we apply the following computations:

$$\begin{aligned} \mathbf{P}_{k+1} &= \mathbf{P}_k - \mathbf{P}_k\mathbf{H}_{k+1}^T\left(\mathbf{I} + \mathbf{H}_{k+1}\mathbf{P}_k\mathbf{H}_{k+1}^T\right)^{-1}\mathbf{H}_{k+1}\mathbf{P}_k \\ \beta_{\mathrm{k+1}} &= \beta_{\mathrm{k}} + \mathbf{P}_{\mathrm{k}}\mathbf{H}_{\mathrm{k+1}}^{\mathrm{T}}\left(\mathbf{T}_{\mathrm{k+1}} - \mathbf{H}_{\mathrm{k+1}}\beta_{\mathrm{k}}\right) \end{aligned} \tag{2.18}$$

where:

$$\mathbf{P}_{k+1} = \mathbf{K}_{k+1}^{-1} = \left(\mathbf{K}_k + \mathbf{H}_{k+1}^T \mathbf{H}_{k+1}\right)^{-1} \tag{2.19}$$

In this way it is possible to compute the output weight for the new batch without the necessity of the previous data. It is important to notice that the matrix $\mathbf{P}_k$ of size [L · L], computed in the previous step, must be kept in memory for the next training phases.
Another important detail is that the Online-Sequential method obtains similar performances as the standard ELM only if the rank of $\mathbf{H}_0$ is equal to the number of nodes in the hidden layer, meaning that the first batch should contain at least L samples. However they must be distinct one from the other, otherwise the rank would be less and we would need more samples.
For this reason the algorithm should first generate β_0 and $\mathbf{P}_0$ then it proceeds to the sequential phase where it receives smaller batches or even single data. The



(a) Train and Test accuracies

(b) Training time

Figure 2.11: OS-ELM performances

performances are computed training the same SFLN, with the same parameters used in the first implementation (figure 2.2), but the MNIST dataset is now provided with batches containing 100 samples each. The results can be analysed in figure 2.11. In this case we arrived to 2000 nodes, instead of 5000, since the accuracy does not improve much. The accuracies show similar results as the base implementation, however the training time increased. This problem is caused by the amount of computations that the algorithm has to perform each step. Just by looking at equation 2.18 we can notice how many matrix multiplications and inverses are present, respect to the base ELM equation.
There is another possible implementation for computing the output weights. Let's

consider the product $\mathbf{H}^T\mathbf{H}$, by dividing $\mathbf{H}$ into different batches we have:

$$\mathbf{H}^T\mathbf{H} = \begin{bmatrix}\mathbf{H}_0^T\mathbf{H}_1^T & \dots & \mathbf{H}_k^T\end{bmatrix} \begin{bmatrix}\mathbf{H}_0\\ \mathbf{H}_1\\ \cdot\\ \cdot\\ \cdot\\ \mathbf{H}_k\end{bmatrix} = \mathbf{H}_0^T\mathbf{H}_0 + \mathbf{H}_1^T\mathbf{H}_1 + ... + \mathbf{H}_k^T\mathbf{H}_k \tag{2.20}$$

this means that the matrix multiplication can be performed adding the contribution of each batch separately, this also applies to the $\mathbf{H}^T\mathbf{T}$ product. In order to compute the output weights we just need to apply the following formula:

$$\boldsymbol{\beta} = \left(\sum_{i=1}^{k}\mathbf{H}_i^T\mathbf{H}_i + \frac{\mathbf{I}}{C}\right)^{-1}\left(\sum_{i=1}^{k}\mathbf{H}_i^T\mathbf{T}_i\right) \tag{2.21}$$

Let's consider that we received k batches and for each of them we computed the two products $\mathbf{H}^T\mathbf{H}$ and $\mathbf{H}^T\mathbf{T}$ and we accumulated their results in the matrix $\mathbf{M}$ and $\mathbf{P}$, respectively. Then, in order to compute the output weights, all we need to do is compute the inverse $\mathbf{M}$ and multiply it with $\mathbf{P}$:

$$\boldsymbol{\beta}_k = \left(\mathbf{M} + \mathbf{H}_k^T\mathbf{H}_k\right)^{-1}\left(\mathbf{P} + \mathbf{H}_k^T\mathbf{T}_k\right) \tag{2.22}$$

This means that we just need to store the matrices $\mathbf{M}$ and $\mathbf{P}$ in memory and the extract them in the training phase. Since the result is the same as the one obtained applying 2.11, the performances in accuracy are not different, if not for some approximations in the computations.
The dimension of $\mathbf{M}$ and $\mathbf{T}$ are, respectively, [L · L] and [L · c], the number of nodes in the hidden layer (L) must be kept limited if the memory budget is limited, although they are the only elements needed to be stored in order to correctly perform the training.
This technique is referred as **Parallel Regularized - Extreme Learning Machine (PR-ELM)** in [26] where it is used for parallel computation, meaning that each batch is given to a different PC which computes the two matrix product and returns them to a central computes which gathers all the products, sum them all together and computes the output weights (figure 2.12).
In [26] it is also shown another possibility where, instead of parallelizing the data, we parallelize the model, meaning that the hidden layer is divided in equal parts and each computes the output weights separately, finally the results are combined together. This can be done when the number of input samples is less than the number of hidden nodes.
We can now evaluate the performances for this method respect to the OS-ELM and the classic method with the entire dataset. We used the same input initialization

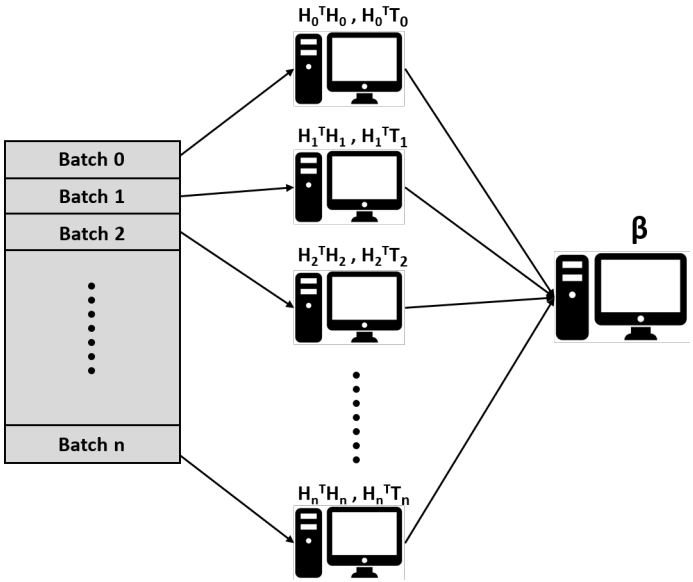
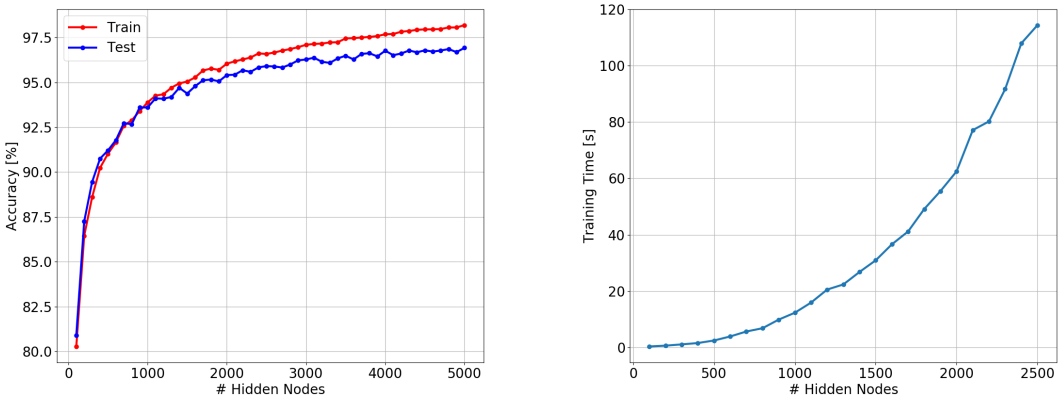


Figure 2.12: Training through parallel computations

as the base implementation shown in figure 2.2, the number of sample per batches has been set to 100 as for the OS-ELM.
As can be seen in figure 2.13, the accuracies basically the same as the base implementation, due to the fact that 2.11 is equivalent to 2.21. This is an important result since the batch can assume any dimension, in this regard we evaluate four different situation of batch size and compute the test accuracy and training time for each case. Let's check now if changing the number of samples in the batch



(a) Train and Test accuracies

(b) Training time

Figure 2.13: P-ELM performances

affects the results. As a reference we use the case with the batch size is equal to the number of samples in the dataset. For the following tests the number of nodes is set to 3000, the parameter C is set to 1 (weak regularization) and the activation function in the hidden layer will be the *ReLU*. The computations are executed exploiting the CPU on the CIFAR-10 dataset. The accuracies for both training and testing samples are evaluated along with the time required for training the entire dataset and the single batch.

BATCH SIZE = 50000:

```
TRAIN accuracy: 58.05%
TEST accuracy: 47.85%
Training time (single batch) = 6.49 s
Training time (entire dataset) = 6.49 s
```

the training time for the single batch is evaluated computing the average between every batches. Now let's consider the case when the number of samples in the batch is higher than the number of nodes in the hidden layer.

BATCH SIZE = 5000:

```
TRAIN accuracy: 58.05%
TEST accuracy: 47.85%
Training time (single batch) = 1.08 s
Training time (entire dataset) = 10.8 s
```

as can be seen the accuracies didn't change because the two equation (2.11) and (2.21) are equivalent, also the random initialization of the input weight is the same for both implementations. The training time for the single batch has reduced due to the fact all the matrices (except for $\mathbf{M}$ and $\mathbf{P}$ which dimension remains the same, since they only depend on the number of hidden nodes and number of classes) have reduced in size, for this reason the computations are faster. However, since the dataset is divided into batches, the total training time is higher because even if the matrices are smaller the number of multiplications and inverses to perform are more.

Let's consider now the case where the batch size is smaller than the number of nodes in the hidden layer. As mentioned, in OS-ELM if the first batch has less sample that the number of nodes in the hidden layer, the training will not be performed correctly.

BATCH SIZE = 1000:

```
TRAIN accuracy: 58.05%
TEST accuracy: 47.85%
Training time (single batch) = 0.44 s
Training time (entire dataset) = 22 s
```

also here, the results do not change and we have the same effects on the time which reduces for the single batch but the total increases.

BATCH SIZE = 1:

```
TRAIN accuracy: 58.05%
TEST accuracy: 47.85%
Training time (single batch) = 0.24 s
Training time (entire dataset) = 12000 s
```

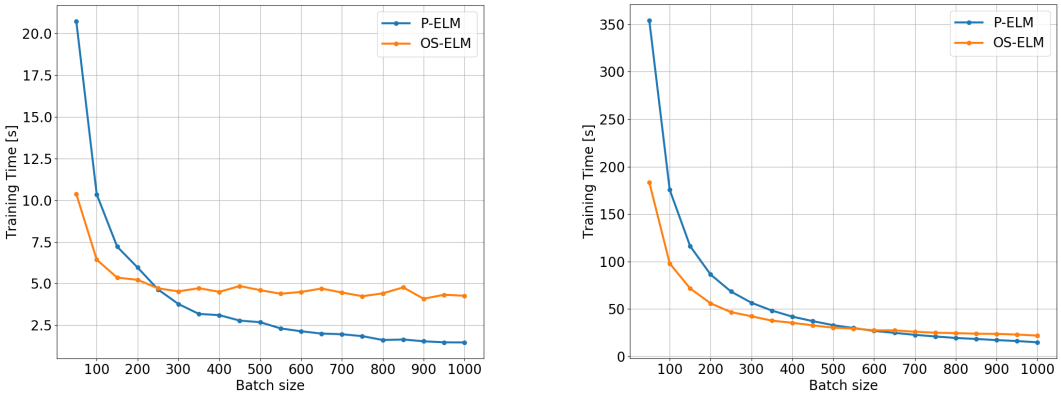
in this last case every batch has a single input sample, as can be seen the accuracy is exactly the same but the total training time, instead, has drastically increased. This happens because the matrix $\mathbf{M}$ maintain its dimension, independently from the batch size, meaning that the number of inverses that the algorithm has to perform (without counting the number of multiplications) matches the number of samples in the dataset. The inverse has the most complexity respect to every other operations in the algorithm, and the time is affected by this problem.
The results obtained are really important as they allow us to train the network without having to much limitations. Another important consequence of using this method is that **the order with which the batches or single samples are provided does not modify the performances obtained in the training**. This means shuffling the dataset or leaving it ordered does not change the final result, what changes is how the machine learns one type of class during training.
Consider, for example, the MNIST dataset, we provide one batch (containing many samples) to the machine and we compute the test accuracy (using all the test set) training the machine only on this batch. If the dataset is shuffled then this single batch may contain samples from all the classes, if the dataset is not shuffled then the batch will contain few classes if not one class only. The test accuracy for this two cases will be different because the machine learns in two different ways.

```
Training ELM with one batch containing only zeros:
batch size = 20
labels: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Accuracy (entire test set): 980/10000 (9.80%)
Accuracy (test set containing only 0s): 980/980 (100.00%)

Training ELM with one batch containing different classes:
batch size = 20
labels: [3, 9, 7, 8, 1, 5, 3, 6, 0, 8, 9, 8, 2, 3, 2, 4, 3, 9, 4, 6]
Accuracy (entire test set): 4146/10000 (41.46%)
Accuracy (test set containing only 0s): 276/980 (28.16%)
```

As can be seen, in the first implementation the machine learns the class 0 with maximum precision, however it has never seen the other classes and so it is not able to differentiate them, thus the accuracy on the entire test set drops. In the second implementation the machine is fed with one batch, with the same size as the previous one, but this time it contains all the possible classes, the results show that the class 0 has not been learnt very well, since it encounters it one time in the batch thus it does not know every possible detail, however the accuracy on the entire dataset is higher because the machine has seen every class but only a small

portion of them.
In figure 2.14 it has been measured and compared the training time for OS-ELM and P-ELM, by varying the batch size. Two cases are evaluated, with different number of nodes. OS-ELM tends to be the best solution with small batches, due



(a) # hidden nodes = 1000 (b) # hidden nodes = 5000

Figure 2.14: Training time OS-ELM and P-ELM

to the fact that, even if it contains more computations respect to the P-ELM case, the inverse operation to be performed depends on the dimension of the batch. In P-ELM the inverse depends on the number of hidden nodes. In fact, considering the case with 1000 hidden nodes and 1000 samples in each batch, we have that both P-ELM and OS-ELM have to perform the inverse on a matrix with same dimension, but the number of operations is bigger for OS-ELM, thus, it requires more respect to the other. But when the batch is small then the OS-ELM has an advantage.
A problem with OS-ELM is that the first batch must contain at least the same number of samples as the number of nodes in the hidden layer, otherwise the performances drop. This could be a problem if in some application the resources are limited and batches with big size are not supported. P-ELM instead does not have this problem, as shown before.
P-ELM suffers from heavy computations when the batch is small, since it has to perform the inverse many times as the number of batches in the dataset. However, it has been noticed that it is not necessary to compute the inverse each time a batch is received. It is possible to accumulate intermediate computations from some batches (or all of them) and perform the inverse as a subsequent step, drastically reducing the training time.
For example, each time a batch is received it is possible to compute just the $\mathbf{M}$ and $\mathbf{P}$ matrices without computing the inverse, which will be performed only after the last batch to obtain the output weights. This method drastically reduces the

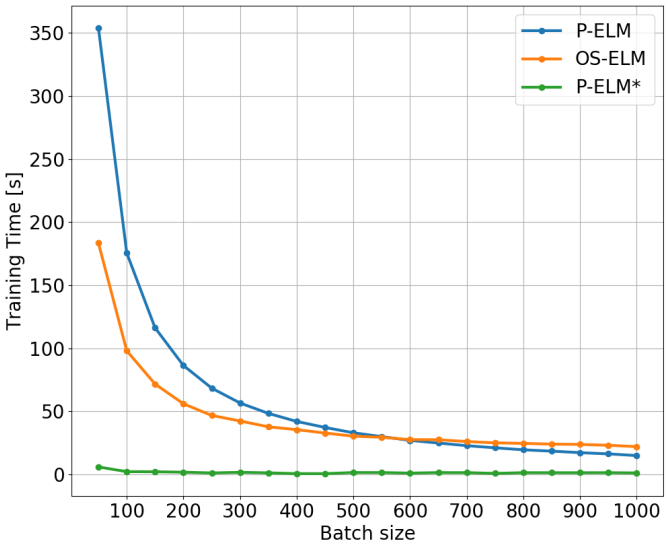


Figure 2.15: Training time of OS-ELM and P-ELM compared with P-ELM*, which accumulates the batches contributions before computing the inverse

training time. This is represented by the green function in figure 2.15, indicated as P-ELM*.
However, by doing this, we update the output weights only at end of the training process and for online sequential this is not the best solution. But, as mentioned, we can decide how many batches accumulate before performing the inverse and compute the output weights. This means that the P-ELM timings can fall between the blue function and the green one.
This may be used when the memory resources are limited and the batches contain to many samples. Basically we divide the batch in smaller mini-batches and compute the output weights after accumulating them. Doing this is perfectly equivalent of training the big batch. This operation can not be executed in OS-ELM, more precisely, we can also divide the batches in mini-batches but we are forced to compute the output weights every time, otherwise the accuracies are not valid. Finally in OS-ELM, depending on the order with which the batches are provided, the accuracy changes, even if only by a small quantity.
In summary the two algorithm show different characteristic which fit differently depending on the application. In this project we are going to use the P-ELM, since it allows on training with a small amount of time (when the batches are accumulated) and since the matrices $\mathbf{M}$ and $\mathbf{P}$ holds informations regarding every sample encountered. The usefulness of the latter characteristic will be explained in the next chapter.

Chapter 3

Continuous Learning - Single Incremental Task

In the last chapter we explained how the Extreme Learning Machine is able to learn with a very few limitations, also we analysed how to provide the machine with different batches each containing only one specific class.
In most cases the number of classes, thus, the number of nodes in the output layer is known a priori and depends on the dataset used or the task to perform, but let's imagine we do not know this number. What we can do is initialize the machine with just one output node and increase this quantity by reading the number of different labels present in a batch, in this way the machine is able to learn new classes, indefinitely.
This operation is referred as **Continuous Learning (CL)** and it is a interesting concept which gives the machine high adaptability to every situation.
To perform this kind of operation is not that simple, one of the main problems is the "**catastrophic forgetting**" [14]. When the machine tries to learn new classes it has do update the weights in the network to adapt to the new data it received, however, this update may modify the network so that some of the old class can be forgotten.
As pointed out in [15] there are basically three scenarios when dealing with the *continuous learning*. We can either choose an architectural approach where the network is modified in order to retain information regarding the knowledge acquired previously. For example some weights in the network can be frozen so they do not participate to the training and the forgetting is reduced, as happens in the *Progressive Neural Networks (PNN)* [16] and *CopyWeights with Re-Init (CWR)* [17]. It is also possible to adopt a solution where the loss function is modified to add a term which makes sure that the update is less effective on the weights that are sensible to small changes, has happens in the *Elastic Weight Consolidation (EWC)* [18] and in *Synaptic Intelligence (SI)* [19]. Another approach could be storing some training

data from the previous steps and combine them with the new data received, in this case we need bigger resources to contain a certain amount of data and in some applications this is not possible.
In continuous learning there are two types of training scenarios. The first is the **Multi-Task (MT)**, where the machine learns different tasks independently one from the other, meaning that the machine performs a training for each task it receives but the computed weights and accuracy are not shared with the same output. It is like having different output vectors in the network and each one is specialized for some classes.

As shown in figure 3.1 there is a part of the network shared, then there are the

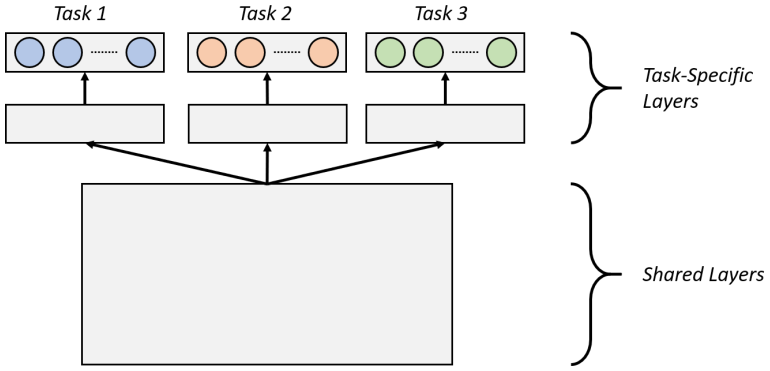


Figure 3.1: Multi-Task learning

task-specific layers trained separately. During training it is also possible to update some of the weights in the shared area. The problem with this type of learning is that it needs separate specialized output vectors, this means that every time we have a new task we need to add another output vector.
The other scenario is the **Single Incremental Task (SIT)**, differently from the multi-task, there is just one output vector which has to perform all the tasks he learned so far. Also, the output vector has to expand if new tasks are encountered and learn them without loosing knowledge of the old ones, all the resources in this case are shared. This is like a classic neural network where the shared part corresponds to the CNN and the output vector corresponds to the classifier, with the difference that the number of tasks may increase over time. This is a more complex situation since a single output vector needs to handle every different operation. Considering, for example, images classification problem we need to add one output node for each new class encountered, train the weights introduced by the new node and adapt the other weights already present in the network. All these without forgetting the old classes.
In order to obtain the optimal performances and avoid the problem of *catastrophic forgetting* we should store all the previous data and use them every time we receive a new batch. This cumulative way to train the network is, however, unrealistic

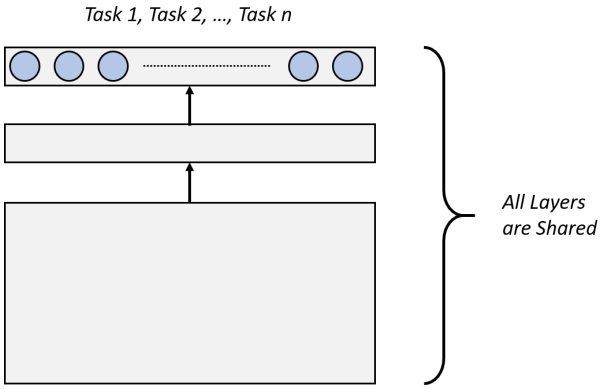


Figure 3.2: Single-Incremental-Task learning

since we would need a big amount of memory and resources. For this reason the implementations shown in [16][17][18][19] are generally used.
In [15] a new technique called **AR1** has been developed, it combines the **Architectural strategy**, which involves freezing part of the network and train a small amount of weights, and **Regularization strategy** used to modify the loss function for selecting the right weights to update during training. More precisely it combines the features from CWR and SI.
In CWR the shared weights ($\mathbf{\Theta}$) in the CNN are trained only for the first batch and then remain frozen, for the other batches the output vector is extended if new classes are encountered. All the weights in the output layer ($\mathbf{w}$) are then trained on the new batches and finally, after a proper scaling, they are substituted with the previous existing weights. The weight scaling is performed because the classes in a batch are unbalanced, some classes contain more samples than others. In CWR this scaling is performed taking count of the weight of a certain class inside the batch, but AR1 proposes a speed-up process for this operation which involves subtracting to the each weight in $\mathbf{w}$ the average of all weights in the matrix, before copying them in the output vector.
To improve performances, AR1 does not keep the CNN weights frozen, some of them are still trained applying fine-tuning. More precisely it implements the regularizations techniques found in EWC and SI. They basically compute the "**importance**" of the weights that need to be tuned, it indicates how the loss function is sensible to small changes of a certain weight in the network, for this reason when fine-tuning is applied the weight with high importance must not undergo large variations otherwise we end up in catastrophic forgetting, thus update is scaled properly by adding a term in equation the 1.6:

$$w_i := w_i - \alpha \frac{\partial}{\partial w_i} C(\mathbf{\Theta}, \mathbf{w}) - \alpha F_i \left(w_i - w_i^*\right) \tag{3.1}$$

where F_i indicates the importance of the i^{th} weight and it correspond to the i^{th} diagonal element of the Fisher information matrix $\mathbf{F}$, and w_i^* is the weight value obtained training the network on the first task upon reaching the loss function minimum.
Summarizing, the AR1 presented in [15] allows to train both $\mathbf{\Theta}$ and $\mathbf{w}$ weights and, at the same time, it prevents catastrophic forgetting through regularization strategy. The steps required to perform the algorithm are the following:

1: Random initialization of $\mathbf{\Theta}$ weights (it is also possible to use pre-trained models)

2: $\mathbf{w}$ and $\mathbf{F}$ are both initialized to zero

3: For each batch received:

- if new classes are encountered then the output layer is expanded
- train both $\mathbf{\Theta}$ (using the SI regularization (3.1)) and $\mathbf{w}$ (without regularization)
- For every class i in the batch:
  - apply scaling on $\mathbf{w}$: $\mathbf{w}[i] \leftarrow \mathbf{w}[i] - avg(\mathbf{w})$

4: Compute accuracy using $\mathbf{\Theta}$ and $\mathbf{w}$ in the network

3.1 Extreme Learning and Continuous Learning

In our experiment we want to exploit the ELM algorithm and apply it in the context of *Continuous Learning* for the *Single Incremental Task* applications. In particular we want to compare the results with the AR1 algorithm [15].
As mentioned, the main problem in continuous learning is the fact that the information about previous data is lost, because dealing with sequential data all the previous batches are discarded.
We talked about data parallelization and how it is possible to perform the training dividing the dataset in different batches without losing performance on the overall result. By looking at the formula 2.22 we have that the output weights are computed by performing the two multiplications ($\mathbf{H}^T\mathbf{H}$ and $\mathbf{H}^T\mathbf{T}$) on the new batch and accumulating the results on $\mathbf{M}$ and $\mathbf{P}$. These two matrices are the fundamental elements needed to implement *Continuous Learning*, they basically contain the information about all the previous batches even if the samples have been discarded. We just need to keep these two matrices stored in memory and update them with the next incoming samples.
This also works with batches containing new classes, the only difference is that now we do not know how many output nodes there will be, for this reason we start from one output node and increase this number depending on the new class we encounter.

Let's first implement a simple SLFN and train it with ELM algorithm on the MNIST dataset, but this time the samples will be split so that only one class is given for each batch.
Remember that **P** is a matrix which dimension is [hn · c] where hn is the number of nodes in the hidden layer and c is the number of known classes. If a new class is encountered we need to add a number of column in P corresponding to the number of new classes in the batch. The matrix **M** instead will be at constant size of [hn · hn].
The algorithm is similar to the one used it the previous section, the difference is that the network architecture must be modified during the run:

```
M = (torch.eye(hn) / C)
P = torch.zeros(hn, classes)
for X, t in dataset:
  max_label = max(t).item()
  if (classes-1) < max_label:
    P = torch.cat((P, torch.zeros(hn, int(max_label-(classes-1)))), 1)
    classes = int(max_label+1)
    elm.fce2 = nn.Linear(hidden nodes, classes, bias=False)
  T = torch.eye(classes)[t]
  H, _ = elm(X)
  M += torch.mm(H.t(), H)
  P += torch.mm(X.t(), T)
  B = torch.mm(torch.inverse(M), P).t()
```

The first operation upon getting the batch is to verify if there is a new class. To do this we check the maximum label and compare with the number of known classes. If this happens, then, we need to increase the columns of **P** by concatenating vectors. The initialization of these vectors must be set to zero, otherwise the contribution of the new batch will be accumulated with whatever values are being held by them. The other operation is to increase the number of nodes in the output layer, to do this we substitute the entire output layer with a new one with more nodes. There is no need to keep the weights trained in the previous step since will be overwritten with the current training. The remaining steps are the same seen previously for the output weights computation in P-ELM, with data parallelization.
The results has been obtained choosing 2000 nodes in the hidden layer with ReLU activation function. After training each batch we computed the accuracy on the entire test set containing also classes that the machine had never seen. We can observe that, for the MNIST, after each step the machine gains more or less 10% accuracy due to the fact that, each step, the machine is learning one class out of the ten present in the test set. The CIFAR-10 is more complex so it reaches less accuracy. The overall performances are the same as the base implementation case, where the machine is trained with the entire dataset. As mentioned data parallelization does not affect the final computation and, thus, the batches order and their content can be freely organized.

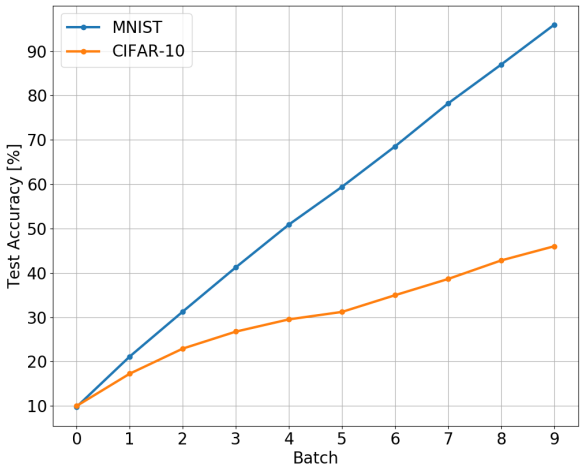


Figure 3.3: Single-Incremental-Task learning with MNSIT and CIFAR-10

3.1.1 Continuous Objects Recognition dataset

There are several different dataset from can be modified and used to test continuous learning applications for example *Permuted MNIST*, *MNIST Split* or *CIFAR10/100 Split*, however they are just the same as the original dataset with a different organization and they do not provide enough complexity for these kind of applications. A dataset proposed in [17] was specifically made to cover the continuous learning field. It is called **Continuous Objects Recognition (CORe50)** and it provides a maximum of 164.866 RGB images of size 128x128 pixels. These images contain 50 different domestic objects from 10 different categories (plug adapters, mobile phones, scissors, light bulbs, cans, glasses, balls, markers, cups, remote controls). The dataset is organized in 11 sessions, providing different background to obtain more generalization, 300 images were extracted for each object in each session. The dataset can be used for different application, for example object detection, segmentation and classification (see [17] for more details).
Our interest is focused on object classification, in particular with the CORe50 it is possible to perform it in two different ways. The first is by considering all the 50 objects as different classes the other is just by recognizing the 10 categories to which the object belongs, the first one is clearly more difficult since we need to differentiate objects in the same categories which are similar one from another.
With the CORe50 dataset it is possible to evaluate three different scenarios for continuous learning:

- **New Instances (NI)**, where new features and conditions of known classes are provided in the incoming batches, this type on training is focused on improving the knowledge of classes that the machine already saw.

Figure 3.4: Some image samples from CORe50

- **New Classes (NC)**, in this case each batch provides only samples of new classes, and the objective is not forgetting what was learned in the previous steps.
- **New Instances and Classes (NIC)**, this is just the combination of the two scenarios above, where in any batch there could be feature for both known classes and new classes and the machine should be able to improve the knowledge without incurring in catastrophic forgetting.

For this project we are going to test the capability of the ELM algorithm applied on a CNN network, for the NC and NIC case since they represent the critical scenarios, comparing the performances with the one obtained with the AR1 algorithm.

3.1.2 Simulations and results

The Convolutional Neural Networks used to evaluate the AR1 algorithm [15] are the **CafféNet** and the **GoogLeNet**. The first problem encountered when using a different dataset, respect to the one used to developed a certain network, is the input image size.
CaffeNet and *GoogLeNet* have been developed relying on the ImageNet, for this reason both the networks have an input size of 227x227 and 224x224 respectively. The 128x128 CORe50 images do not fit in the network, the possible solutions to this problem could be stretching the image or reshape the network properly. As pointed out in [15] these two methods leads to worse performances and, to avoid this, the stride and padding are properly reduced.
There are other modifications applied on the network to implement AR1, in Caffenet the fully-connected layers fc6 and fc7 were halved from 4096 nodes to 2048, fc8

is adapted from 1000 nodes (ImageNet) to 50 nodes (CORe50). Moreover, to correctly fit the CORe50 dataset into the network, the stride of the first convolutional layer (conv1) was modified and set to 2, the padding of the second convolutional layer (conv2) was set to 1.
In GoogleNet the modifications are more complex. GoogLeNet has three output layers, two of which are intermediate. These intermediate output layer are used exclusively for the training phase [9] and they are both preceded by one fully-connected layer, which are removed for the AR1. The kernel size of the average pool, executed before these intermediate output, have been set to 6 instead of 5. The stride and pad for the first convolution (conv1) are set to 1 and 0 respectively and, finally all the three output layer neurons have been modified to fit 50 classes.
AR1 algorithm has been evaluated on the **NC** scenario and compared with other algorithms, the results extracted from [15] are shown in figure 3.5. The *Cumulative*

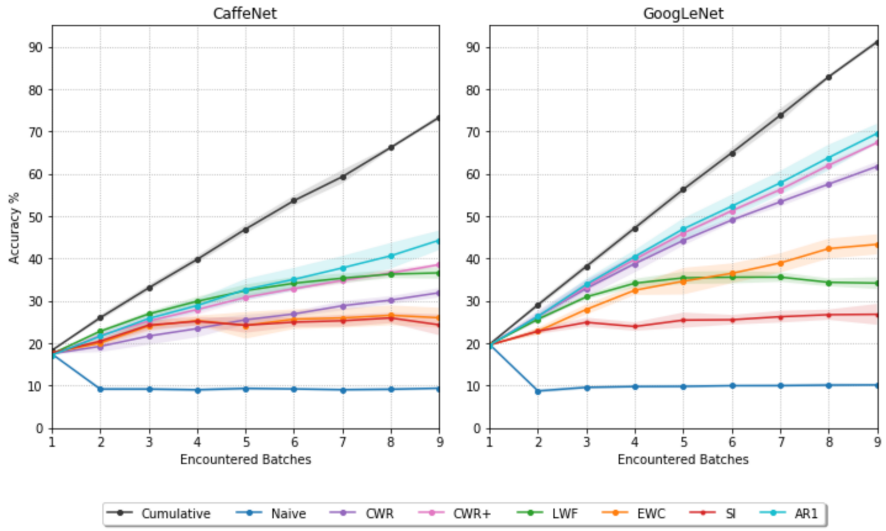


Figure 3.5: Results AR1 compared with other standard CL algorithms

has the best results among all because its algorithm involves storing all the encountered batches and training each step with every one of them. It can be considered as the maximum performance obtainable. The *Naive* method instead are the results obtained when the network is trained just by tuning weights and without any method to prevent forgetting, in fact it shows the worst accuracies.
We are also going to use these two networks for our computations, however, the classifier used for ELM is a little bit different and thus other consideration must be applied regarding the network architecture. Every CNN has its own classifier, represented by the fully-connected layers.
As mentioned, in order to apply extreme learning on the CNN we need to remove these layers and insert the ELM classifier. The ELM classifier comprehend one hidden layer and the output layer with 50 classes. The CaffeNet classifier is composed

by three fully-connected layers (fc6, fc7 and fc8) which are all removed for this implementation. More precisely fc6 has been substituted with the hidden layer (4096 neurons) of the ELM, randomly re-initializing the weights. The layer fc7 has been substituted with the output layer (50 neurons). The other parameters to modify are similar to what performed in AR1, with the difference that in GoogLeNet we do not use the the two auxiliar output, thus those layers are not modified. All the modifications done to the networks are shown in table 3.1 and 3.2. The ImageNet

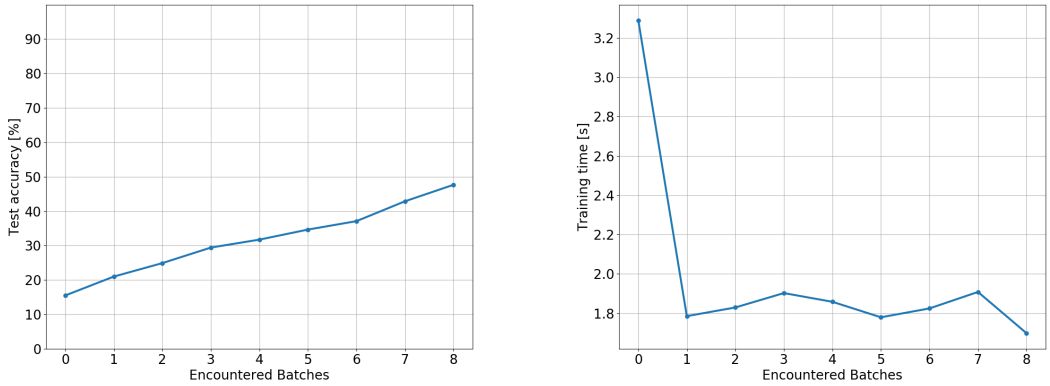
CaffeNet		
Layer	Original	Modified
data (Input)	size: 227x227	size: 128x128
conv1 (convolutional)	stride: 4	stride:2
conv2 (convolutional)	pad: 2	pad: 1
fc6 (fully connected)	neurons: 4096	neurons: 4096 (re-init)
fc7 (fully connected)	neurons: 4096	neurons: 50
fc8 (output)	neurons: 1000	removed

Table 3.1: Network modifications on CaffeNet

GoogLeNet		
Layer	Original	Modified
data (Input)	size: 227x227	size: 128x128
conv1 (convolutional)	stride:2, pad: 3	stride: 1, pad: 0
loss3/classifier (output)	neurons: 1000	removed

Table 3.2: Network modifications on GoogLeNet

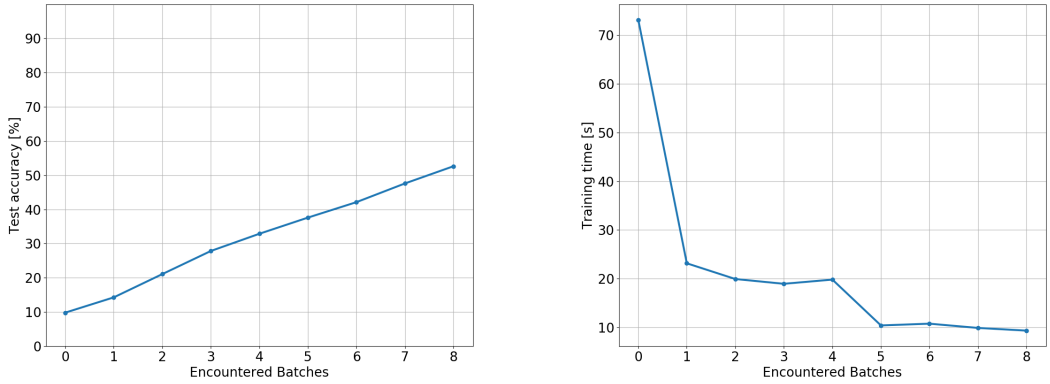
pre-trained model has been used for both CaffeNet and GoogLeNet, obtaining high level of complexity in the features extracted. The results obtained for the CaffeNet implementation is shown in figure 3.6. The first step has been training and testing the network in the New Classes (NC) scenario. The CORe50 provides 9 batches in total for the NC case, the first batch is bigger than the other ones. In particular the first batch provides 10 classes, and the remaining batches provide 5 classes each. As mentioned each batches contains only new classes that the network has never seen. This means that a certain class can not be found in multiple batches. The



(a) Test accuracy (b) Training time

Figure 3.6: Performances obtained after training each batch, on CaffeNet, using ELM algorithm

NC represent the critical situation where the forgetting factor is more evident.
The code is the same as the one used for the SIT test on MNIST and CIFAR-10, where the network starts with one output nodes and, when a new class is recognized, the output is increased.
The results obtained for CaffeNet are shown in figure 3.6. It can be noticed that the accuracy ($\sim$ 15%) for the first batch is lower that the one obtained in AR1 (20%), this may be associated to the fact that, in AR1, all the weights in the CNN are trained (with regularization) obtaining more adaptation to the dataset being used. The trend of the next batches are similar for both P-ELM and AR1. After training the last batch the accuracy for P-ELM reached about 47%, which is slightly higher than the AR1 (in figure 3.5).
The training time has been extracted exploiting the GPU (the one mentioned it the previous chapter) evaluating only the algorithm computation. This means that the dataset extraction is excluded, the time starts when the input matrix (containing the batch) is received and ends when the output weights are updated. The time required is higher on the first batch since it contains more samples, the remaining batches takes more or less the same amount of time to be computed. The same tests have been performed also on GoogLeNet, the results are shown in figure 3.7. As can be noticed the accuracy did not reach the one with the AR1 algorithm, obtaining $\sim$ 50% accuracy. A possible explanation is that this network is more complex than the CaffeNet and training just the last layer is not enough to obtain good generalization. We tried setting different values for C and the range for the input initialization of the ELM input layer and the best results are the one shown in figure.



(a) Test accuracy (b) Training time

Figure 3.7: Performances obtained after training each batch, on GoogLeNet, using ELM algorithm

A solution to this problem could be finding a better adaptation to the ELM classifier inside the network, also normalize the dataset could help improve the accuracy. We tried to apply fine tuning, more precisely we trained the classifier with extreme learning on a percentage of the dataset and then with backpropagation with the remaining percentage.

The results are shown in table 3.3 and as can be seen they did not improve.

Dataset percentage		**NC** scenario
Extreme Learning	Backpropagation	Accuracy (last batch)
80%	20%	35.06%
50%	50%	37.49%
20%	80%	39.72%

Table 3.3: Results fine-tuning with backpropagation performed on the output layer

We also tried apply fine-tuning using two classifier, one trained with backpropagation and one with extreme learning as shown in figure 3.8. The training was also split for the two learning algorithm, like in the previous case. The results obtained are shown in 3.4. Also for this case the accuracy did not improve.

Regarding the NIC scenario, there are no data available for AR1 at the moment. In this case we compared the results obtained using ELM with the CWR algorithm extracted from [17], for just Caffenet. The final results, after the last batch, are basically similar to the one obtained for the NC case, due to the fact that the order

Dataset percentage		**NC** scenario
Extreme Learning	Backpropagation	Accuracy (last batch)
80%	20%	40.02%
50%	50%	40.64%
20%	80%	41.41%

Table 3.4: Results dual hidden layer, one trained with Extreme Learning algorithm and the other with Backpropagation algorithm

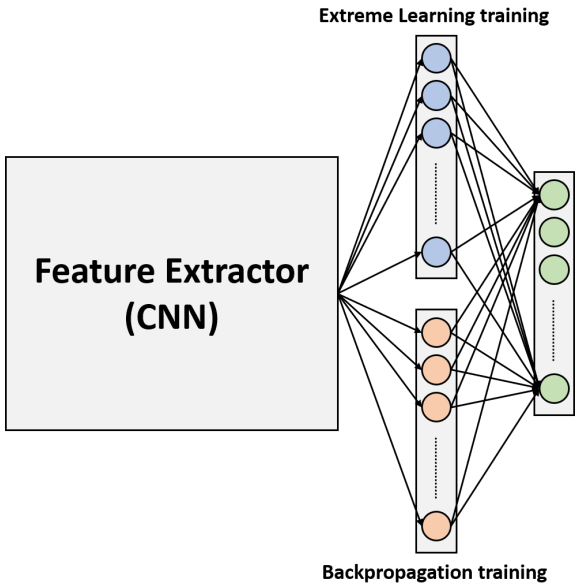
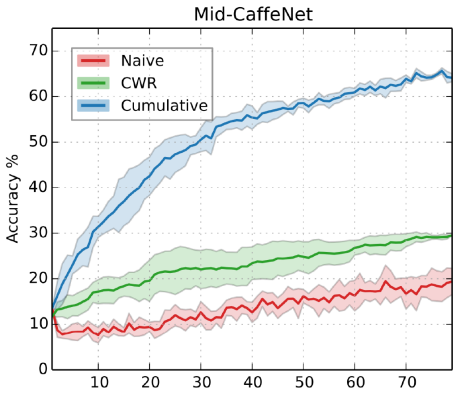


Figure 3.8: Structure dual hidden layer, one trained with Extreme Learning algorithm and the other with Backpropagation algorithm

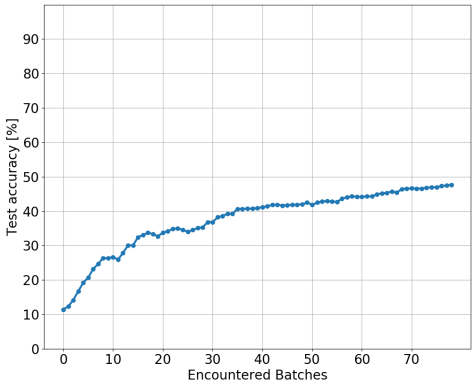
of which the batch are provided and the number of batches with which the dataset is divided does not matter thanks to P-ELM. The accuracy show better results respect to the CWR algorithm.

3.1.3 Storage and Computational Complexity

It is now important to understand the memory requirements and the computational complexity for the P-ELM algorithm.
Starting with the storage requirements the following elements are needed for the algorithm computation:



(a) Test accuracy on CWR, compared with Cumulative and Naive algorithms



(b) Test accuracy with ELM training on CaffeNet

Figure 3.9: Performances on NIC scenario

1: Non-temporary elements to store in memory:

- Weights ($\mathbf{\Theta}$) and biases (b) contained in the CNN (the number of elements depends in the network being used).
- Input weights ELM: matrix of size $(hn \cdot in)$, where hn is the number of nodes in the hidden layer and in is the number of inputs of the ELM classifier.
- Biases hidden layer ELM: vector of size (hn).
- Output weights ($\boldsymbol{\beta}$) ELM: matrix of size $(c \cdot hn)$, where c is the number of classes that the machine knows, thus the number of output nodes.
- Matrix $\mathbf{M}$: matrix of size $(hn \cdot hn)$, which stores the contribution of $\mathbf{H}^T\mathbf{H}$.
- Matrix $\mathbf{P}$: matrix of size $(hn \cdot c)$, which stores the contribution of $\mathbf{H}^T\mathbf{T}$.

2: Temporary elements to store in memory during execution:

- Matrix $\mathbf{X}$: matrix of size $(bs \cdot d \cdot h \cdot w)$, which contains the input samples, where bs is the number of samples that the batch sontains, d is the image depth, h is the image height and w is the image width.
- Matrix $\mathbf{T}$: matrix of size $(bs \cdot c)$, which contains the one hot encoding of the labels corresponding to the input samples.
- Matrix $\mathbf{H}$: matrix of size $(bs \cdot hn)$, which contains the activations of the hidden layer for each sample in the batch.

during the execution there are several optimization that can be performed, for example, once the matrix $\mathbf{H}$ is computed, the matrix $\mathbf{X}$ has no longer use, for this

reason is possible to delete or substitute it with the matrix $\mathbf{H}$.
Apart from the network weights and biases, which need to be stored in any machine learning application, the fundamental for this algorithm are $\mathbf{M}$ and $\mathbf{P}$ matrices. They store the contribution of every encountered batch. Their size depends on the number of nodes in the hidden layer, in particular, this dependency is quadratic for the $\mathbf{M}$ matrix. The number of hidden nodes must be kept limited depending on the memory budget.
In AR1 algorithm, the elements to be stored are: a temporary matrix containing the output weights of the last layer, and two times a matrix containing a number of elements equal to the remaining weights and biases of the entire network not considering the output ones. Two times because one matrix is to store the actual weights and biases ($\mathbf{\Theta}$), the other to store the importance of each of these elements ($\mathbf{F}$).
To compare the storage requirements of these two algorithm, let's consider that wb is the total number of elements in the network excluding the output ones and $(c \cdot hn)$ is the number of output weights.

$$\begin{aligned} \mathbf{AR1} &= 2(wb - c \cdot hn) + c \cdot hn \\ \mathbf{PELM} &= wb + (hn \cdot hn) + 2(c \cdot hn) \end{aligned} \tag{3.2}$$

Considering the CaffeNet, the number of parameters in the CNN are approximately $wn = 62.378.344$. For AR1 $hn = 2048$ and for P-ELM $hn = 4096$. Both have $c = 50$. The number of elements can be now computed obtaining:
AR1 = 124.654.288
P-ELM = 79.565.160
For the P-ELM case, however, the number of elements grows quadratically with the number of nodes in the hidden layer $(hn \cdot hn)$, thus hn must be kept as limited as possible. The algorithm complexity has been evaluated extrapolating each operation performed: These complexity shown are theoretical and they refer to the

OPERATION	**MATRICES SIZE**	**COMPLEXITY**
Matrix multiplication $\mathbf{H}^T\mathbf{H}$	$(hn \cdot bs) * (bs \cdot hs)$	$\mathbf{O}(hn^2 \cdot bs)$
Matrix addition $\mathbf{M} + (\mathbf{H}^T\mathbf{H})$	$(hn \cdot hn) + (hn \cdot hn)$	$\mathbf{O}(hn^2)$
Matrix multiplication $\mathbf{H}^T\mathbf{H}$	$(hn \cdot bs) * (bs \cdot c)$	$\mathbf{O}(hn \cdot bs \cdot c)$
Matrix addition $\mathbf{P} + (\mathbf{H}^T\mathbf{T})$	$(hn \cdot c) + (hn \cdot c)$	$\mathbf{O}(hn \cdot c)$
Matrix inverse $\mathbf{M}^{-1}$	$(hn \cdot hn) + (hn \cdot hn)$	$\mathbf{O}(hn^3)$
Matrix multiplication $\mathbf{M}^{-1} \cdot \mathbf{P}$	$(hn \cdot hn) \cdot (hn \cdot c)$	$\mathbf{O}(hn^2 \cdot c)$

Table 3.5: Computational complexity for each operation in the algorithm

worst case. For example, a multiplication between two squared matrices with size $(d \cdot d)$ has complexity $\mathbf{O}(d^3)$, however there are several algorithm which allows to reduce it. The Coppersmith–Winograd algorithm brings the multiplication complexity from $\mathbf{O}(d^3)$ to $\mathbf{O}(d^{2.376})$.
As can be seen the inverse has the biggest complexity among all the other operation, also the matrix in which the operation is performed has size $(hn \cdot hn)$. This is, in fact, another reason to keep the number of hidden nodes limited.
With P-ELM, as mentioned, it is not mandatory to compute the inverse every time a batch is received. For each batch it is possible to perform the first four computation in the list above, then whenever we need the updated output weights we compute the last two operation. For example, it is possible to compute the inverse one time, after every batch in the dataset is received, this not only drastically reduces time but also the number of computations.

Conclusions

The main purpose of this project was to study the Extreme Learning algorithm, to and examine its properties and extend it to a real problem applications, as the Continuous Learning task. In the first part we exposed the relevant aspect of the algorithm, its strengths and weaknesses. We provided a base implementation and a starting point from the following developments.
Two types of extreme learning machine (OS-ELM and P-ELM) have been studied for their capability of learning even with batches provided sequentially. These two algorithms were, then, compared choosing the most fit for the application. In particular, we observed that the P-ELM is able to store in a matrix the contribution of each batch, so that, even if the batch is discarded in the next step, its information is not lost.
This is a useful property since, in continuous learning, the major problem is the catastrophic forgetting, where new learned classes may damage the knowledge of the old ones. The results on the split MNIST and CIFAR-10 were promising, in fact, we obtained the same performances without loosing knowledge.
When dealing with the CORe50 dataset and with complex CNN we had different results. In particular for CaffeNet the accuracies were comparable with the novel AR1 algorithm, showing that even training a single layer with ELM the learning is enough to obtain good generalization. For GoogLeNet the results were not the ones expected, we obtained lower performances. This could be caused by the complexity of the CNN being used and training only the last layer is not enough to achieve a good adaptation to the dataset used, also adapting the ELM classifier in any CNN requires different parameters settings.
Future work may focus on a method to train the network using both ELM algorithms for the output classifier and regularized fine-tuning applied on the CNN, or increasing the ELM capability by exploiting a multi layer classifier approach.

Bibliography

[1] Guang-Bin Huang, Qin-Yu Zhu, Chee-Kheong Siew, "Extreme learning machine: Theory and applications", Neurocomputing, Volume 70, Issues 1–3, December 2006, Pages 489-501.

[2] Guang-Bin Huang and Haroon A. Babri, "Upper Bounds on the Number of Hidden Neurons in Feedforward Networks with Arbitrary Bounded Nonlinear Activation Functions", IEEE, DOI:10.1109/72.655045.

[3] Yong Peng, Wanzeng Kong, Bing Yang, "Orthogonal extreme learning machine for image classification", Neurocomputing, Volume 266, 29 November 2017, Pages 458-464.

[4] Yong Peng, Wanzeng Kong, Bing Yang, "Orthogonal extreme learning machine for image classification", Neurocomputing, Volume 266, 29 November 2017, Pages 458-464.

[5] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, Li Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge", IJCV, 2015.

[6] Yan LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, "Gradient-Based Learning Appliedto Document Recognition", proceedings of the IEEE, november 1998.

[7] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks", DOI:10.1145/3065386.

[8] Karen Simonyan, Andrew Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition", Published as a conference paper at ICLR 2015.

[9] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, Google Inc., University of North Carolina, Chapel Hill, University of Michigan, Ann Arbor Magic Leap Inc., "Going Deeper with Convolutions", arXiv:1409.4842.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, "Deep Residual Learning for Image Recognition", arXiv:1512.03385.

[11] Guang-Bin Huang, Senior Member, IEEE, Hongming Zhou, Xiaojian Ding, and

Rui Zhang, "Extreme Learning Machine for Regression and Multiclass Classification", IEEE, DOI:10.1109/TSMCB.2011.2168604.

[12] R. Fletcher, "Practical Methods of Optimization: Volume 2 Constrained Optimization", New York: Wiley, 1981.

[13] Nan-Ying Liang, Guang-Bin Huang, Senior Member, IEEE, P. Saratchandran, Senior Member, IEEE, N. Sundararajan, Fellow, IEEE, "A Fast and Accurate Online Sequential Learning Algorithm for Feedforward Networks", IEEE, DOI:10.1109/TNN.2006.880583.

[14] Robert M. French, "Catastrophic forgetting in connectionist networks", Neurocomputing, Volume 3, Issue 4, 1 April 1999, Pages 128-135, DOI:10.1016/S1364-6613(99)01294-2.

[15] Davide Maltoni, Vincenzo Lomonaco, "Continuous Learning in Single-Incremental-Task Scenarios", arXiv:1806.08568.

[16] Andrei A. Rusu, Neil C. Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, Raia Hadsell, "Progressive Neural Networks", arXiv:1606.04671.

[17] Davide Maltoni, Vincenzo Lomonaco, "CORe50: a New Dataset and Benchmark for Continuous Object Recognition", arXiv:1705.03550.

[18] Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell, "Overcoming catastrophic forgetting in neural networks", arXiv:1612.00796.

[19] Friedemann Zenke, Ben Poole, Surya Ganguli, "Continual Learning Through Synaptic Intelligence", arXiv:1703.04200.

[20] Mingxing Duan, Kenli Li, Canqun Yang, Keqin Li, "A hybrid deep learning CNN–ELM for age and gender classification", ScinceDirect, Neurocomputing, Volume 275, 31 January 2018, Pages 448-461, DOI:10.1016/j.neucom.2017.08.062.

[21] Lili Guo, Shifei Ding, "A Hybrid Deep Learning CNN-ELM Model and Its Application in Handwritten Numeral Recognition", ResearchGate, DOI: 10.12733/jcis13987.

[22] Qian Weng, Zhengyuan Mao, Jiawen Lin, and Wenzhong Guo, "Land-Use Classification via Extreme Learning Classifier Based on Deep Convolutional Features", IEEE, DOI: 10.1109/LGRS.2017.2672643.

[23] Andreas Kölsch, Muhammad Zeshan Afzal, Markus Ebbecke, Marcus Liwicki, "Real-Time Document Image Classification using Deep CNN and Extreme Learning Machines", IEEE, DOI: 10.1109/ICDAR.2017.217.

[24] Yujun Zeng, Xin Xu, Yuqiang Fang, Kun Zhao, "Traffic Sign Recognition Using Extreme Learning Classifier with Deep Convolutional Features", DOI: 10.1007/978-3-319-23989-7_28.

[25] Vivienne Szee, Yu-Hsin, Tien-Ju Yang, Joel S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey", IEEE, DOI:

10.1109/JPROC.2017.2761740.

[26] Yueqing Wang, Yong Dou, Xinwang Liu, Yuanwu Lei "PR-ELM: Parallel regularized extreme learning machine based on cluster", Neurocomputing, Volume 173, Part 3, 15 January 2016, Pages 1073-1081, DOI: https://doi.org/10.1016/j.neucom.2015.08.066.

[27] Sinno Jialin Pan and Qiang Yang, Fellow, IEEE "A Survey on Transfer Learning", IEEE, DOI: 10.1109/TKDE.2009.191

[28] https://www.mathworks.com/videos/introduction-to-deep-learning-what-are-convolutional-neural-networks–1489512765771.html