Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

LOW-RANK APPROXIMATION IN NEURONAL NETWORKS - ESI Group Internship

DEGREE FINAL WORK

Degree in Computer Engineering

Author: Vicent Dolz Martinez
Tutor: Eva Onaindia de la Rivaherrera
Company tutor: Francisco Jose Chinesta Soria

Course 2019-2020

Resum

Els algoritmes d'intel·ligència artificial han experimentat una dràstica millora en l'última dècada generant models molt precisos en tots els camps d'estudi. Això ens ha portat a un nou problema d'emmagatzematge massiu de dades i llargs temps d'entrenament per a aquests models. Les aproximacions de baix ordre són una tècnica de compressió que redueix tant la quantitat de paràmetres emmagatzemats com els càlculs realitzats. Fent ús de descomposició tensorial, la unitat bàsica d'una xarxa neuronal, som capaços de accelerar els entrenaments. Donada aquesta descomposició, proposem un nou mètode d'entrenament basat en enfocaments de baix rang que proporcionarà major convergència del nostre model usant menys dades i afegint enriquiments que evitarà l'estancament de l'aprenentatge. Farem servir xarxes neuronals completament connectades (NN) així com xarxes neurals convolutivas (CNN).

Paraules clau: Baix ordre, descomposició tensorial, CNN

Resumen

Los algoritmos de inteligencia artificial han experimentado una drástica mejora en la última década generando modelos muy precisos en todos los campos de estudio. Esto nos ha llevado a un nuevo problema de almacenamiento masivo de datos y largos tiempos de entrenamiento para estos modelos. Las aproximaciones de bajo orden son una técnica de compresión que reduce tanto la cantidad de parámetros almacenados como los cálculos realizados. Haciendo uso de descomposición tensorial, la unidad básica de una red neuronal, somos capaces de acelerar los entrenamientos. Dada esta descomposición, proponemos un nuevo método de entrenamiento basado en enfoques de bajo rango que proporcionará mayor convergencia de nuestro modelo usando menos datos y añadiendo enriquecimientos que evitará el estancamiento del aprendizaje. Usaremos redes neuronales completamente conectadas (NN) así como redes neurales convolutivas (CNN).

Palabras clave: Bajo orden, Descomposición tensorial, CNN

Abstract

Artificial intelligence algorithms have experienced a dramatic improvement in the last decade generating very precise models in all fields of study. This has led us to a new problematic of mass data storage and long training times for these models. Low-rank approximations are a compression technique that reduce both the amount of stored parameters and the computations. Making use of tensors decomposition, the basic unit of a neural network, we are able to speeds up trainings. Given this decomposition, we propose a new training method based on low-rank approaches that will provide greater convergence of our model using less data and adding enrichments that will prevent the stagnation of learning.

We will use fully-connected neural network (NN) as well as convolutional neural network (CNN).

Key words: Low-Rank, Tensor decomposition, CNN

Contents

List of Figures

List of Tables

CHAPTER 1
ESI Group

1.1 Company history

get it right®

Figure 1.1: ESI Group logo

ESI (Engineering System International) was founded as Engineering Systems International in France in 1973 by Alain de Rouvray along with three other recent PhD.s from the University of California Berkeley: Jacques Dubois, Iraj Farhoomand and Eberhard Haug. The company initially operated as a consulting company for European defense, aerospace and nuclear industries.

ESI Group provides virtual prototyping software that simulates a product's behaviour during testing, manufacturing and real-life use. Engineers in a variety of industries use its software to evaluate the performance of proposed designs in the early phases of the project with the goal of identifying and eliminating potential design flaws.

The goal of Virtual Prototyping is to obtain real-life results without physically testing real-life parts. In its unique approach to Virtual Prototyping, ESI integrates the properties inherited from materials and manufacturing processes into the virtual model to dramatically improve the accuracy and predictability of the virtual prototype. The result is a drastic drop in hardware prototyping.

The ESI goal is to improve industrial product development by enabling manufacturers to pre-certify their products virtually. To give you a concrete example, the ESI customer Expliseat, a young French start-up who launched with just three employees, recently used ESI software to develop the lightest commercial aircraft seat ever and certified it in just 18 months. Their design reduces seat weight by 50%, which translated into an annual 3% - 5% fuel saving per aircraft.

(a) Real prototypes

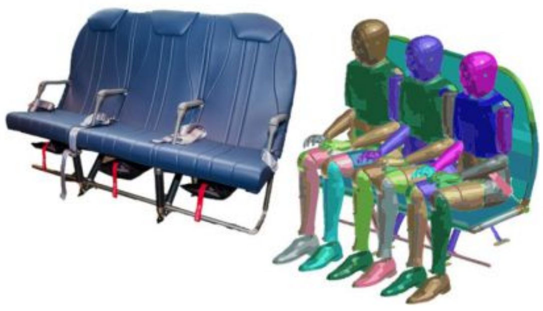
(b) Real seat (left) and virtual seat prototype (right)

Figure 1.2: Expliseat Titanium Seat

Industrial manufacturers now routinely rely on computer simulation during product development to conduct extensive tests. These virtual tests ensure products meet set performance requirements in various domains such as crash worthiness, stiffness and strength, acoustics, durability, comfort and maneuverability. In most cases, performance in one domain affects others yet each of these areas is typically evaluated using its own suite of dedicated simulation tools and each test is conducted by a different team working in silos, resulting in time-consuming and costly iterations. Converting simulation models for use across different platforms is tenuous; and generating highly accurate results is challenging.

ESI eliminates these problems with its End-to-End offering, which links multiple simulation domains to a central and unique "single core model". This enables engineers to predict the performance across domains of an actual production part as it is built and used, taking the manufacturing process into consideration including casting, stamping, welding, assembly or additive manufacturing—, rather than just analyzing a nominal computer-aided design (CAD) model.

1.2 Introduction

The concept of artificial intelligence as Machine Learning and neural network algorithms has been around for decades. These algorithms require on the one hand an enormous amount of computational power and on the other hand a very high volume of data. It is for this reason that in the last 10 years with the arrival of the Big Data, the increase of the computational strength of the processing units and the increase of experts in this field have led to the algorithms being able to perform more complex tasks. But this is a double-sided coin. As the algorithms are more effective and their capacity of generalization increases, also the complexity of the problems that we look for to solve also increases.Due to this complexity, to obtain data to train these algorithms has become very expensive and the computational force is not enough.For this reason, methods are being developed to simplify the problems to be solved.

1.2.1. Motivation

The idea of doing an internship at Esi Group from the beginning was to put into practice what I had learned in my career and acquire knowledge in my specialty of computing by using machine learning algorithms and applying artificial intelligence techniques to solve industrial problems. All the mentioned with the objective of giving me a base of experience as a data scientist, the job profile I would like to dedicate myself to.

From the company's point of view, my purpose was to contribute to their area of work, virtual prototyping. My co tutor, Rubén Ibañez, with whom I was going to work and I decided to carry out a research dedicated to make less expensive the neural networks making use of methods like the low rank approaches or the data compression.

1.2.2. Objectives

The aim of the research is to reduce the data stored and the number of coefficients to be trained by the neural networks during and outside of learning. This objective will be addressed in two ways.The first, at the internal level of the network by modifying its internal architecture demonstrating that low-range approaches can be used. And the second, at the external level of the network, that is applying compression methods to the data that we will use throughout the training.

1.2.3. Methodology

Regarding the internal modification of the network, the structure we will use in the first part will be to create a standard neural network used in the industry and analyze its performance. We will then create a neural network with a separate format using the methodology of low-order approaches and compare its performance with the normal.

We will apply this same approach to a convolutional neural network, one of the most widely used architectures. And finally we will study the use of compression methods in the data set that we will use in the training process.

1.2.4. Implementation

For this study we will use the Python programming language. Python is perfect for data processing and strongly used in the industry providing a huge community which makes this language ideal. We will make us of the Tensorflow library making use of its low level functions to implement the neural networks with our methodology.

CHAPTER 2
State of the art

2.1 Introducing Artificial Intelligence

The pioneers of Artificial Intelligence dreamed of building complex machines that had the same characteristics as human intelligence back in the 1950s.

Today, although programming something as complex as the human mind still seems distant to us, we are experiencing a tremendous advance in the use of Machine Learning. And for a few years now, specifically Deep Learning. Both of these are part of Artificial Intelligence, which was designed to make machines smarter, even more so than humans.

Artificial Intelligence can be divided into two main groups: robust AI and applied AI.

- **Robust Artificial Intelligence or Strong AI:[9]** deals with a real intelligence in which machines have similar cognitive capacity to humans, that is, it acts more like a brain, so instead of classifying, it makes use of clustering to create associations and process data. The idea is to imitate the results, but in this case we are not sure of the outcome. For example talking to a human, where you can predict that you will answer a particular question, but instead the answer may not be the same given the same question to the same person.

  Another example is the AI in games, where an AI with reinforced learning[1] learned to play various Atari games, and in some of them achieved in just 2.5 hours exceed human performance. To everyone's surprise by leaving more time to generate strategies that were not planned.

  Even with these examples in specific fields, experts believe that the real strong AI will take some time to arrive.

- **Applied Artificial Intelligence Weak AI:[9]** this is where the use we make through algorithms and guided learning with Machine Learning and Deep Learning comes in. Machine Learning[10] in its most basic use is the practice of using algorithms to parse data, learn from it and then be able to make a prediction or suggestion about something. Programmers must perfect algorithms that specify a set of variables to be as accurate as possible in a particular task. The machine is trained using a large amount of data giving the opportunity for the algorithms to be perfected.

Weak AI	**Strong AI**
It is a narrow application with a limited scope.	It is a wider application with a more vast scope.
This application is good at specific tasks.	This application has an incredible human-level intelligence.
It uses supervised and unsupervised learning to process data.	It uses clustering and association to process data.
Example: Siri, Alexa.	Example: Advanced Robotics

Table 2.1: Comparative table of Weak and Strong AI

Since the early days of artificial intelligence, algorithms have evolved with the aim of analysing and obtaining better results: decision trees, inductive logic programming (ILP), clustering to store and read large volumes of data, Bayesian networks and a wide range of techniques that data science programmers can take advantage of.

2.2 Machine Learning

Machine Learning is a discipline in the field of Artificial Intelligence that, through algorithms, gives computers the ability to identify patterns in massive data to make predictions. This learning allows computers to perform specific tasks autonomously, that is, without the need to be programmed.

The term was first used in 1959. However, it has gained relevance in recent years due to increased computing power and the boom in data. Machine learning techniques are, in fact, a fundamental part of Big Data.

Machine Learning algorithms are divided into three categories, the first two being the most common:

- **Supervised Learning[5]:** these algorithms have a previous learning based on a system of labels associated to some data that allow them to make decisions or predictions. An example is a spam detector that labels an e-mail as spam or not depending on the patterns it has learned from the e-mail history (sender, text/image relation, keywords in the subject, etc.).

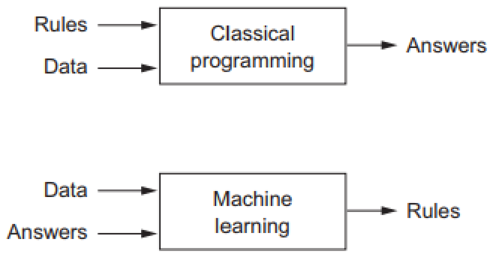

Figure 2.1: Machine Learning: new programming paradigm

Within this category we have **Convolutional neural networks (CNN)**, we will explain this architecture later since we will use it in our research.

- **Unsupervised learning[6]:** These algorithms have no prior knowledge. They deal with the chaos of data in order to find patterns (see figure 2.2) that allow them to be organized in some way. For example, in the field of marketing they are used to extract patterns from massive data from social networks and create highly targeted advertising campaigns.

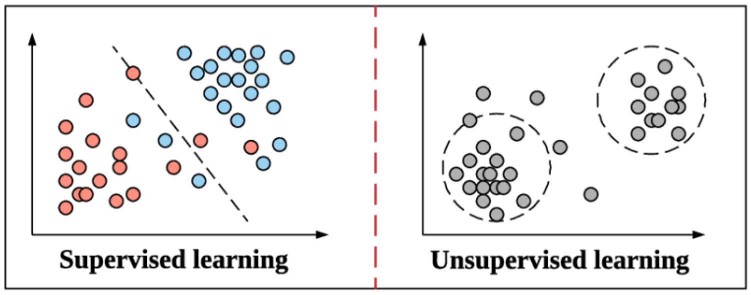

Figure 2.2: Supervised learning vs Unsupervised learning

- **Reinforcement Learning[11]:** Its objective is that an algorithm learns from its own experience. That means, it is capable of making the best decision in different situations according to a trial and error process in which the right decisions are rewarded. It is currently being used to enable facial recognition, make medical diagnoses or classify DNA sequences.

Continuing with the evolution of Machine Learning, in the last decade a technique of Machine Learning has been spreading more and more strongly known as Deep Learning.

2.3 Deep Learning

By definition, Deep Learning is a subset within the field of Machine Learning (see figure 2.3), which preaches the idea of learning by example.

In Deep Learning, instead of teaching the computer a huge list of rules to solve a problem, we give it a model that can evaluate examples and a small collection of instructions to modify the model when errors occur. Over time, we hope that

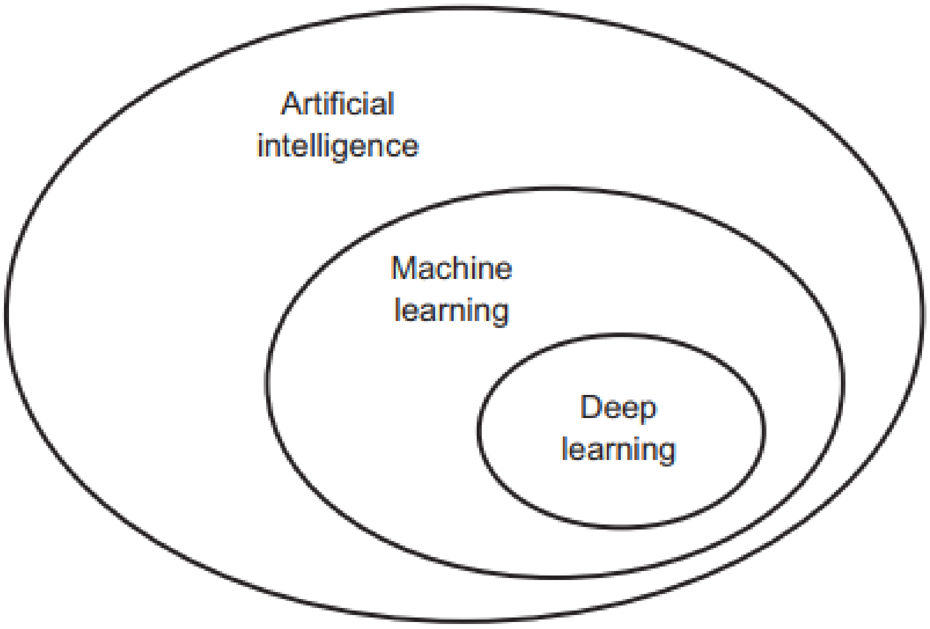

Figure 2.3: Deep Learning sub field of artificial intelligence

these models will be able to solve the problem extremely accurately, because the system is able to extract patterns.

Although there are different techniques to implement Deep Learning, one of the most common is to simulate a system of artificial networks of neurons within the data analysis software.

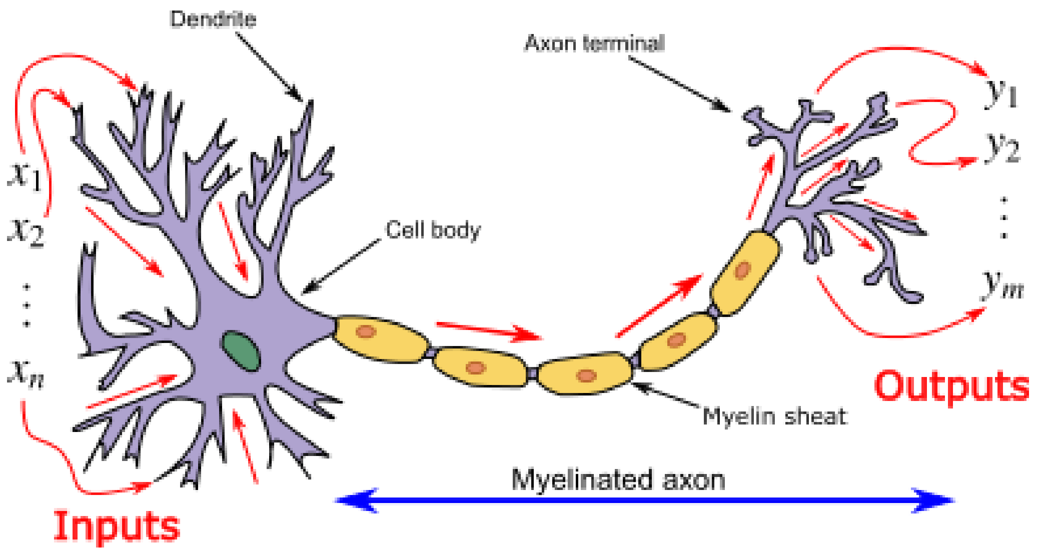

Figure 2.4: Neuron ,with signal flow from inputs at dendrites to outputs at axon terminals

These networks are inspired by the biological functioning of our brain composed of the interconnection between neurons as can be seen in figure 2.4. This type of neuron is similar to the McCulloch-Pitts Neuron model [2]. In our case, simplifying, this artificial network of neurons is composed of different layers, connections and a direction in which data is propagated through each layer with a specific task of analysis.

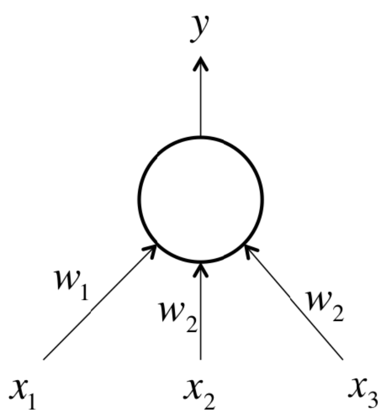

Figure 2.5: Simplified Neuron according to figure 2.4

The aim is to provide enough data to the neuron layers so that they can recognize patterns, classify and categorize them. One of the great advantages is to work from unlabeled data and analyze its patterns of behavior and occurrence.

For example, you can take an image as input information for the first layer. There it will be partitioned into thousands of pieces (see figure 2.6) that each neuron will analyze separately. We can analyze the color, the shape, etc... each layer is expert in a characteristic and assigns a weight to it. Finally, the end layers of neurons collect that information and provide a result.

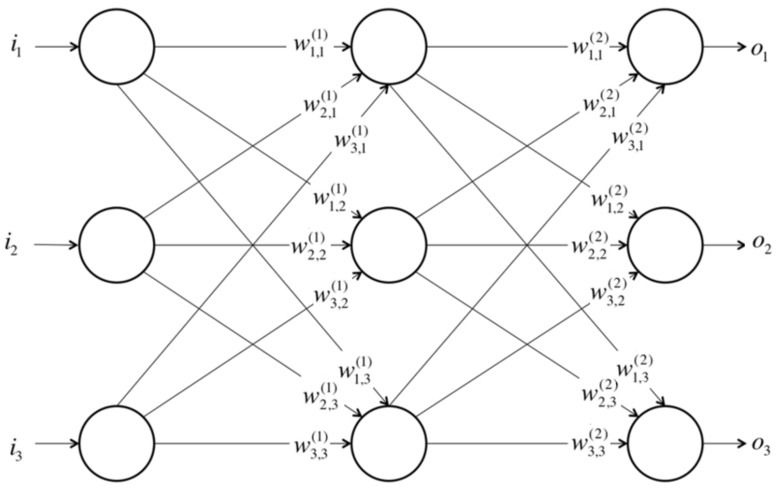

Figure 2.6: Neural network with one hidden layer

Each neuron assigns a weight to the input as a correct or incorrect result relative to its task. The output is determined by the sum of these weights:

$$y^{(i)} = w_1 x_1^{(i)} + w_2 x_2^{(i)} + w_3 x_3^{(i)} \tag{2.1}$$

If we use the example of an image with a cup, we can analyze on one side its shape, its texture with respect to the background, the disposition of the handle, if it has a handle, if it is supported on a table, etc.. The neural network will conclude whether or not it is a signal. Based on training we can conclude with better probabilities of success in each of the layers.

The number of automatic learning techniques is very large. Lately, neural networks have become popular due to their high precision and it seems that the rest of the ML techniques have been left aside. But the truth is that there are many other very powerful techniques such as Gradient Boosting or Random Forest which is based on decision trees.

2.4 Gradient Boosting and Random Forest

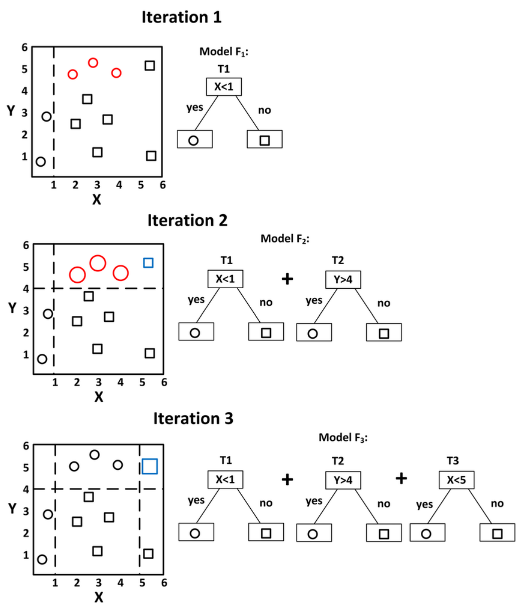

Figure 2.7: Simple example of visualizing Gradient Boosting.

All the methodology that will be explained belongs to the group of techniques known as Gradient boosting [3] and Random forest, machine learning techniques for regression and classification problems. They are based on making use of a set of weak prediction models to make a stronger one(see figure 2.7).

As examples of these techniques we have, Bagging, Decision tree learning, Gradient Tree boosting.

The weak decision trees are generated sequentially, with each tree being created in such a way as to correct the errors of the previous tree. Apprentices are usually "shallow" trees, typically only one, two or three levels deep. One of the parameters of this type of argument is the learning rate, which controls the degree of improvement of a tree with respect to the previous one. A small learning rate means a slower improvement but better adaptation to the data, which generally translates into improvements in the result at the cost of greater consumption of resources.

The weak decision trees are generated sequentially, with each tree being created in such a way as to correct the errors of the previous tree. Apprentices are usually "shallow" trees, typically only one, two or three levels deep.

One of the parameters of this type of argument is the learning rate, which controls the degree of improvement of a tree with respect to the previous one. A small learning rate means a slower improvement but better adaptation to the data, which generally translates into improvements in the result at the cost of greater consumption of resources.

2.5 Convolutional neural network (CNN)

In this type of architecture, artificial neural networks are modeled where the neurons correspond to receptive fields similar to the neurons in the visual cortex V1 of a human brain. This type of networks are very effective for :

- Object detection and categorization
- Image classification and segmentation

The goal of CNN is to learn higher order characteristics using the convolution operation. Since convolutional neural networks can learn input-output relationships (where the input is an image), in convolution, each output pixel is a linear combination of the input pixels.

But what does "convolution" mean? Convolution is the filtering of an image using a mask. Different masks produce different results. The masks represent the connections between neurons in previous layers.

These layers progressively learn the higher order characteristics of the raw input. This process of learning automatic features is the main feature of the deep learning model, called feature extraction.

Convolutional neural networks are formed using two types of layers: convolutional and pooling. The convolution layer transforms the input data using the convolution mathematical operation. The convolution operation describes how to merge two different sets of information.

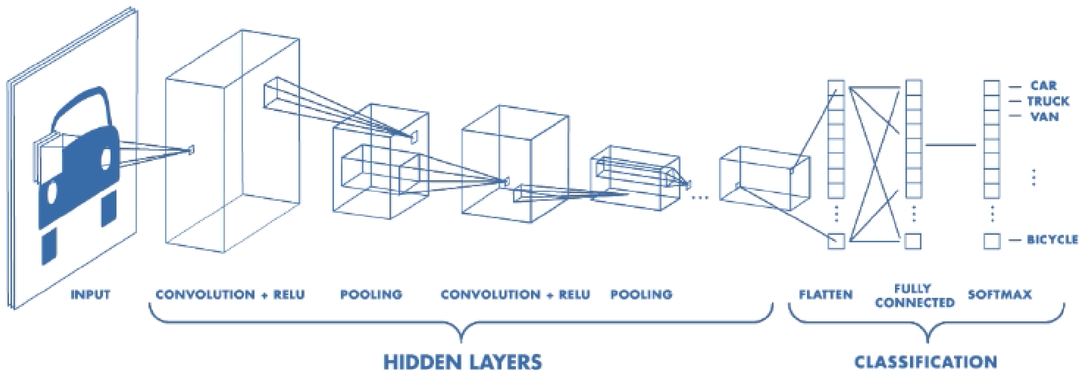

Figure 2.8: Scheme of a convolutional neural network

After the convolution layer(s) a pooling layer is used. The function of the pooling layer(s) is to summarize the responses of the nearby outputs. The two main features of the pooling layer are, first, the pooling layer progressively reduces the spatial size of the data. And second, the pooling layer helps to obtain an invariant representation to a small translation of the input.

2.6 Compression models, SVD, POD, PGD, ...

Due to a new engineering paradigm based on recent simulation, in recent years the dimensional complexity of mesh based approaches has been increasing (see figure 2.9). These approaches have been increasing the degrees of freedom to such an extent that their computational cost has become unmanageable. One of these approaches of ESI Group, is the use of parametric prediction models to simulate mechanical behavior in industrial pieces.

These compression models allow us to compress a set of data assuming some losses and reducing the volume of these as well as the dimensionality. Among all the different techniques, there are the Singular Value Decomposition (SVD), the Proper Orthogonal Decomposition (POD)[8] and Proper Generalized Decomposition (PGD)[7].

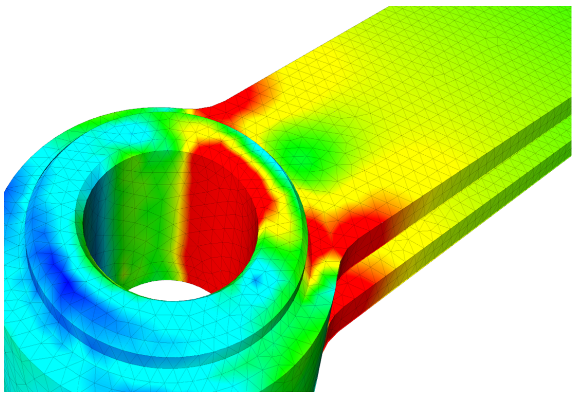
Figure 2.9: Finite element mesh piece.

In order to better understand the methodology used throughout the work, we will explain how Singular Value Decomposition works.

The idea is that any matrix M of $m \times n$ can be represented as a low rank approximation of three matrices such that:

$$M_{m\times n} = U_{m\times m}\Sigma_{m\times n}V^T_{n\times n}$$

for example:

$$M = \begin{bmatrix} u_1 & u_2 & u_3 \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 & 0 & 0 & 0 \\ 0 & \sigma_2 & 0 & 0 & 0 \\ 0 & 0 & \sigma_3 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_1^T \\ v_2^T \\ v_3^T \\ v_4^T \\ v_5^T \end{bmatrix}$$

$$M = \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \sigma_3 u_3 v_3^T$$

being U and V two orthogonal columns matrix and Σ, a diagonal matrix with r singular values that represents the rank of M. This r value is the number of linearly independent columns/rows.

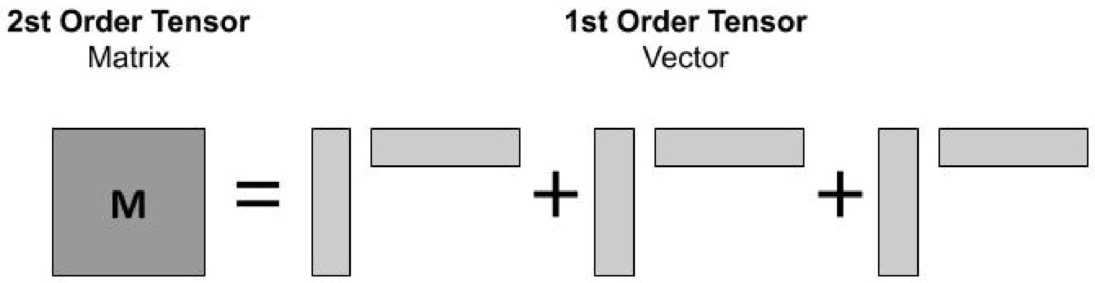

Figure 2.10: Low-Rank approximation of M.

In the figure 2.10 we see the matrix M of $Rank(M) = 3$, represented as a combination of 3 sets vector column and row. Each of these sets is what we will call throughout this work as enrichment.

CHAPTER 3

Is it possible to separate a neural network?

For this first part the objective is to see if it is possible to train a neural network in a separate format using low order approaches while maintaining the same accuracy as a neural network in normal format.

In the first point we will create on one hand the regression problem to be solved and on the other hand a standard neural network to solve it. We will analyze the performance of this normal format. In the second point we will explain the methodology we want to apply and we will create a network in a separate format and analyze its performance. In the last point we will compare the accuracy, coefficients to train and stored memory of both networks.

3.1 Non separated neural network

To begin with we needed a simple problem to solve, the easiest thing is to predict a transfer function that we had created, so we decided to create our own training and test set to feed the network

$$WX = Y \tag{3.1}$$

Our goal was simply, see if it was possible to separate the transfer function, see if it train correctly with this modification and the model converge. Our training set X and Y consisted of a third-order tensor $[n, x, y]$, with N being the input size we would pass to the layer and x and y the number of rows and columns. The second-order tensor $[y, x]$, was initialized with random values. On the other hand we need a transfer function. We generate a matrix that will be the transfer function that we will look for to obtain, and from this we generate the output set.

```
def generate_input(self, no_input=100, ):
    np.random.seed(1)
    self.input = np.empty((self.res,self.res,no_input))

    for i in range(no_input):
        Y = np.random.rand( self.res,self.res)
        self.input[:,:,i]=Y
        return self.input

    def generate_W_lrank(self, no_rank=2): # Create W separated
        np.random.seed(1)

        self.W_rows = np.random.rand(self.res,no_rank)
        self.W_cols = np.random.rand(self.res,no_rank)
        return self.W_rows, self.W_cols

    def w_full_reconstruction(self): # Create full W
        self.W_full = np.tensordot(self.W_rows, self.W_cols, axes=(1,1)
       )
        return self.W_full

    def output_generator_full(self):
        self.output_full = np.tensordot(self.W_full,self.input, axes
       =(1,0))
        return self.output_full

    def output_generator_lrank(self):
        aux = np.tensordot(self.W_cols, self.input, axes=(0,0))
        self.output_full_separated = np.tensordot(self.W_rows, aux,
       axes=(1,0))
            return self.output_full_separated

```

Algorithm 3.1: Data generation

Once we have all the data, we create an Adaline (**ADA**ptative **LIN**ear **E**lement) type neuron(see figure 3.1, based on McCulloch-Pitts's neuron model [2].

$$Y_i = \sum_{j=1}^{n} w_{ij} x_j + b_j \tag{3.2}$$

It is generally composed of a single layer of n neurons (therefore n output values) with m inputs with the following characteristics:

- The m inputs represent a vector x of input that belongs to the space R^m.
- For each neuron, there is a w matrix of synaptic weights that indicate the connection strength between the input values and the neuron. In practice they represent the weighting of each input on the neuron.
- A bias vector b.
- y is the output of the model.

We train the model in normal format using as loss function the mean square error (MSE):

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (\hat{Y}_i - Y_i)^2 \tag{3.3}$$

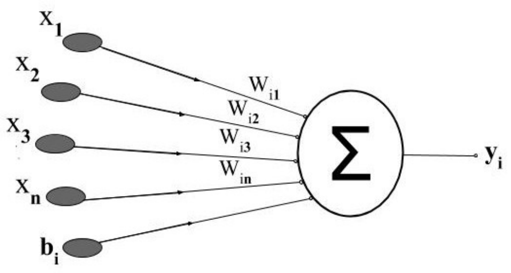

Figure 3.1: Adaline linear neuron

```
    loss = tf.losses.mean_squared_error(pY,iY) #MSE

```

Algorithm 3.2: Mean squared error loss function.

Being n the size of the tensor , $\hat{Y}$ the estimated values (prediction) and Y the observed values (output). To reduce the loss we use as optimizer the gradient descent to minimize it:

$$w(t+1) = w(t) - \epsilon \nabla E(w(t))$$

```
    optimizer = tf.train.GradientDescentOptimizer(learning_rate =
   learning_rate).minimize(loss)
    acc =  tf.pow((1 - loss /(tf.reduce_sum(tf.pow(iY, 2))/(2*res*
   res*no_inputs))),3)
```

Algorithm 3.3: Gradient descent optimizer and accuracy measures.

To find the optimal weight setting by the descent method gradient we operated as follows. Start from $t = 0$ of a certain weight configuration $w(0)$ and calculate the direction of the maximum variation of the loss function $E(w)$ in $w(0)$ which would be given by its gradient in $w(0)$. The sense of the maximum variation would point to a hill in the hypersurface represented by the function $E(w)$. The parameters w are then modified in the opposite direction to that indicated by the error function gradient. In this way, it is possible to carry out a decrease through the error hypersurface, approaching in a certain amount the local minima valley. The previous process of descending the gradient is iterated until that said local minima is reached. What we get with this optimizer is that the bigger the error with respect to the output the bigger the update it produces. To control how big this update is, we have the learning rate, a numerical parameter that dictates the magnitude of the descent direction. A higher learning rate means a faster convergence if the model converges.

Once the architecture is established, i.e. our neuron, input and output data, it is time to test the relevant parameters and adjust for the optimal use. In the next figure we train it with a $learning\ rate = 0.15$, $epochs = 12,000$ without batch size,which is equivalent to 12 000 gradient descent steps, and a $Wrank = 20$. This last parameter allows us to adjust the amount of coefficients to be trained simultaneously given the mathematical complexity behind it. In this approach we did not make use of it cause the training variable it is not splitted, so at this moment we are training a two order tensor of $[100 \times 100]$, it meas 10,000 coefficients.

In order to be able to assess the accuracy, we use the loss value as an indicator since we have the actual transfer function at all times. In this way, with the mean square error we get an idea of how much the coefficients of our approach vary from the real one. This means that a loss value equal to zero is that the transfer function we are approaching is equal to the real one:

$$Lossvalue = 0 \Rightarrow \hat{W} = W$$

After making several tests and having adjusted the parameters we obtain the following graph that shows the evolution of the loss value along the epochs. We can see that the neuron trains well, at just over 4000 epochs(see figure 3.2) we reach a loss value close to 0. It should be noted that by iteration we are adjusting 10,000 coefficients.

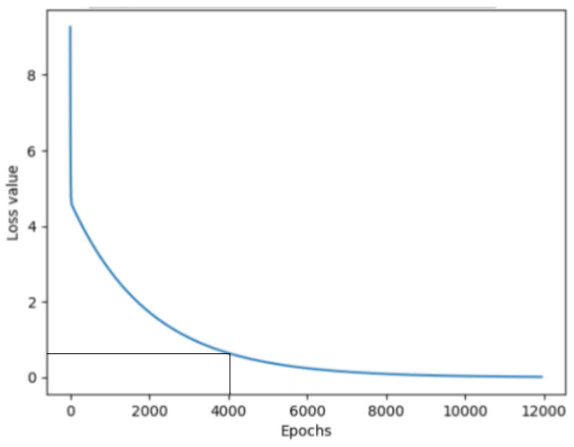

Figure 3.2: Loss evolution along epochs in the non separated model.

The next step is to adapt our neuron to see if it is able to train separately and analyze how this affects the accuracy of the model. On the other hand, see in Tensorflow what primary functions it has to be able to implement our network since the functions used in this library are of very high level and for our approach we need to go to the tensorial computations.

3.2 Separated neural network

For this new approach in separate format, we have separated the tensor from the transfer function into two first-order tensor , i.e. into a row vector and a column vector. In this way we are left with:

$$W = W_{rows} \otimes W_{cols} \tag{3.4}$$

We train these two vectors separately by making their tensor product to obtain our full W matrix. Normally, only one separated couple of W_{rows} and W_{cols} is not enough and it is required several enrichments to recover the full operator. The number of modes requires to recover the initial matrix can be seen as the rank of the W operator that we are seeking.

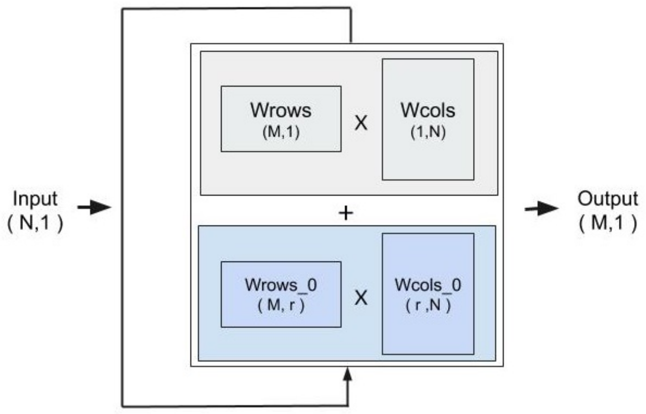

Figure 3.3: Methodology.

At this point we introduce the concept of enrichment 3.3, on which the learning of our neuron and the knowledge extracted will be based. This enrichment consists of training a a couple of W_{row} and W_{col} until its accuracy is stagnant, at this point the variable is stored and re-initialized. All this previous enrichments will be used in the next training iteration as previous knowledge. From a Tensorflow perspective, the previous enrichments will be introduced in the network throughout placeholders. The placeholders are entities in the network which need to be feed by the user and they are not trained by the optimizer. In such a way, we can keep the memory of the previous enrichments into our network without affecting the training of the next enrichment.

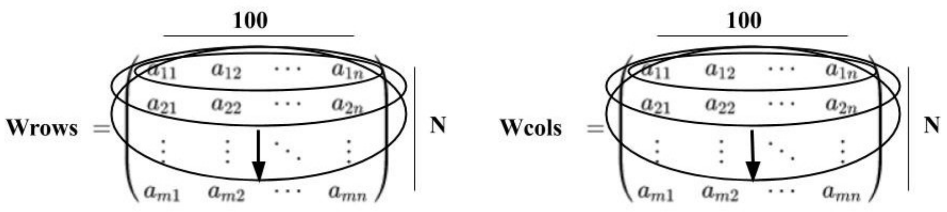
100
Wrows =
N
a_{11} a_{12} ⋯ a_{1n}
a_{21} a_{22} ⋯ a_{2n}
a_{m1} a_{m2} ⋯ a_{mn}
100
Wcols =
N

Figure 3.4: Enrichment tensors.

This methodology in the Python programming language using the functions offered by the Tensorflow library is reflected in the following way:

- The two tensors W_{rows} and W_{cols} created as **tf.placeholders** like this:

```
iWrows = tf.placeholder("float", [res, None], name= '
    Transfer_functions_LR')
iWcols = tf.placeholder("float", [res, None], name= '
    Transfer_functions_LR')
```

Algorithm 3.4: Placeholders to enrich the Tensorflow session.

The placeholders will be fed by the dictionaries at session level in the same way as the input and output data set.

These two variables will increase in size after each enrichment as can be seen in figure 3.4.

- At the Tensorflow graph level it is proposed as follows:

```
Wrows = tf.Variable(tf.ones([res, 1])) #Wrows variable initialize
Wcols = tf.Variable(tf.ones([res, 1])) #Wcols variable initialize
mean_Wrows = tf.reduce_mean(Wrows)
Wrows_adim = Wrows / mean_Wrows


Wrows_total = tf.concat([iWrows, Wrows_adim], 1)
Wcols_total = tf.concat([iWcols, Wcols], 1)

pYaux = tf.tensordot(Wcols_total, iX, axes=(0,0))
pY = tf.tensordot(Wrows_total, pYaux, axes=(1,0))
```

Algorithm 3.5: Graph flow.

Two **tf.variables** W_{rows} and W_{cols} are created and initialized to ones that will be the variables to train. One of these two variables is dimensionalised and both concatenated to the dictionary with the previous enrichment in order to generate the complete W tensor and calculate the error, then apply the optimizer and calculate the accuracy of the model (see algorithm 3.6).

```
loss = tf.losses.mean_squared_error(pY,iY)
optimizer = tf.train.GradientDescentOptimizer(learning_rate =
    learning_rate).minimize(loss)
acc =  tf.pow((1 - loss /(tf.reduce_sum(tf.pow(iY, 2))/(2*res*res*
    no_inputs))),3)
```

Algorithm 3.6: Loss optimizer and accuracy functions.

With the flow of the network created it is necessary to create a stop condition within the session to enrich the training. This stop should occur when the learning is stagnant, i.e. our precision function does not advance more than a value that we specify. This is the most complex part that needs to be manually adjusted since it will be the stop condition to enrich and determine if our model is robust and converges the learning or if it diverges and stays underfitted.

```
 if((diff_loss < fb_rate[i]) & (step>100)):
    print('\n\n')
    # print('*****************************************\n',_loss)
    dt.get_loss_conv(_loss)
    _Wrows_adim, _Wcols = sess.run([Wrows_adim, Wcols],
    feed_dict={    iX : input_data,
                    iY : output_data,
                    iWrows : w_rows_ant,
                    iWcols : w_cols_ant })
    w_rows_ant = np.concatenate((w_rows_ant , _Wrows_adim), axis =
   1)
    w_cols_ant = np.concatenate((w_cols_ant, _Wcols), axis = 1)
```

Algorithm 3.7: Stop condition and enrichment.

As we see in the code piece of the session (algorithm 3.7), when we enrich we take out of the session the two tensor **Wrows** and **Wcols** trained at that time that have the current knowledge and we concatenate them to a tensor **w_rows_ant** and **w_cols_ant** that will have all the previous knowledge acquired and will be what enters as placeholder in the next epoch.

3.3 Results

After adjusting the parameters to obtain good network trainings, we compare the results obtained in the model with the separate format to the results of the network in normal format.

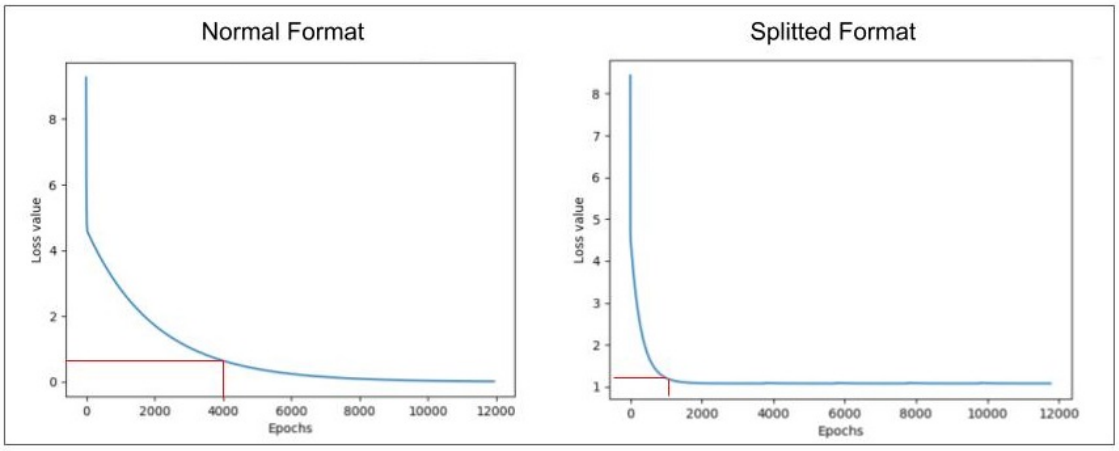

Figure 3.5: Comparative of losses evolution along epochs.

As we can see in the figure 3.5, the normal format needs 4000 to have a lost close to 0, while in the separate version only 1000 epochs.

In this plot (figure 3.6) it shows the loss value along the enrichments. After eight feed backs we are stuck at a loss value of about 1. This improvement not only translates into speed but also into memory. As we can see down here.

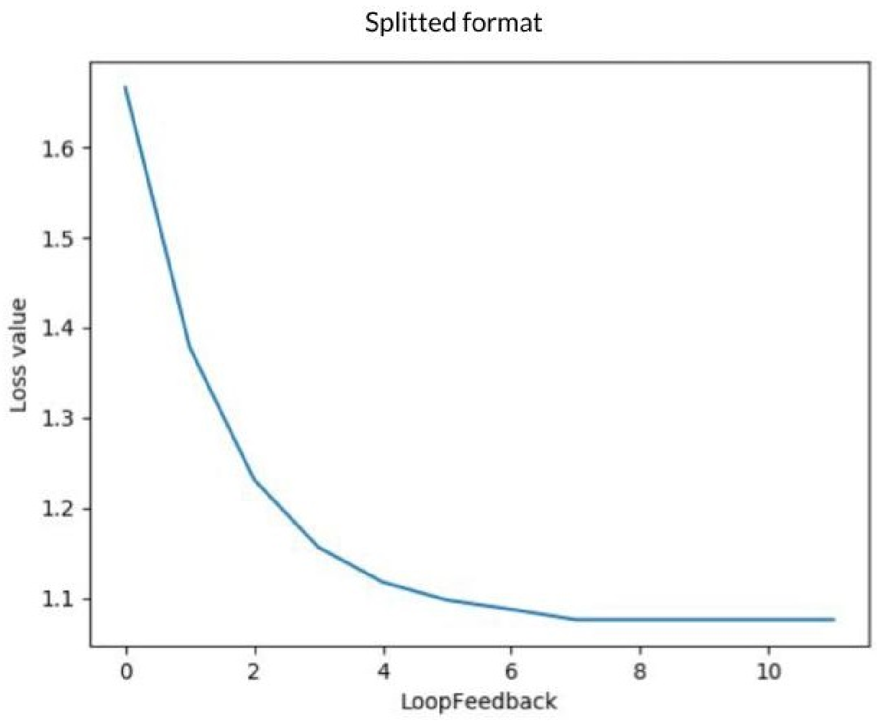

Figure 3.6: Smallest loss values along enrichments.

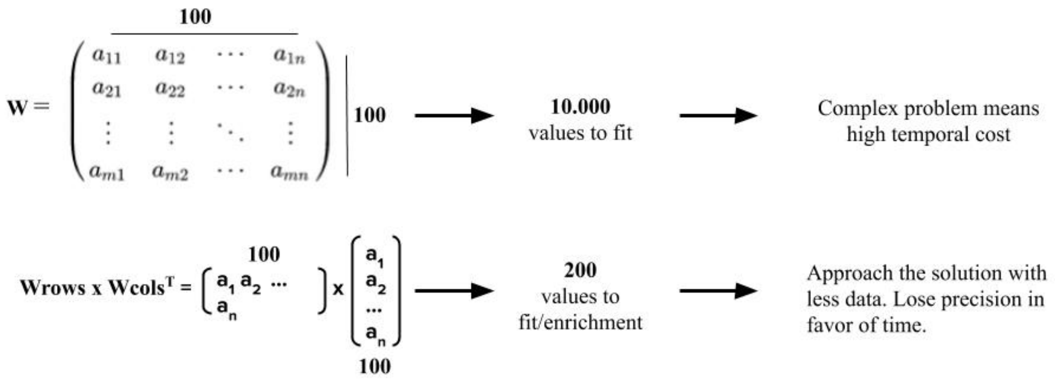
W =
100
100
10.000 values to fit
Complex problem means high temporal cost
Wrows x Wcols^T =
100
100
200 values to fit/enrichment
Approach the solution with less data. Lose precision in favor of time.

Figure 3.7: Models comparative.

On the one hand we have a clear improvement in the number of coefficients we have to train. We went from 10,000 coefficients per epoch, to about 200 for the number of enrichments, in our case as you can see in the figure 7, there are eight. That leaves 1,600 coefficients against 10,000.

The separate format leaves us with certain problems. On the one hand we have the learning rate, having to train two separate vectors can happen that it diverges and that the optimizer does not achieve convergence. One way to solve this is to adapt the learning rate dynamically. Then we have the enrichments, how many to do during the training and how many to do. The solution we have proposed to this is to measure the precision in each period and to compare the current one with the previous ones and given the difference of both decide dynamically the moment to enrich.

As a summary of this part we see that this separate format trains just as well as the standard used but with slight improvements in the number of coefficients to train and the number of epochs in exchange for having to add certain parameters to adjust to avoid divergence problems. The next step is to see how this methodology works in a more complex problem such as a classification problem using another type of architecture such as a convolutional neural network.

CHAPTER 4

Separated Convolutional Neural network on classification problem

In this part of the research we seek to apply this methodology of separate format to a more complex problem. We decided to use convolutional neural networks and work on an image classification problem. This architecture has gained much reputation in the industry for its results in image processing as well as for its versatility for all kinds of fields. As in the previous chapter, in the first point we will work with a normal format, in the second one we will modify the previous network to a separate format with the same image dataset and we will compare the results. Then we will analyze the problem statement of applying this methodology to the Convolutional neural network and see if we can improve the operational structure. And finally compare the results between formats.

4.1 Traditional CNN

As a data set for this part we will use the Mnist Dataset, is an acronym that stands for the Modified National Institute of Standards and Technology dataset. It is a widely used and deeply understood dataset of 55,000 small square 28×28 pixel gray scale images of handwritten single digits between 0 and 9 (see figure 4.1). The task is to classify a given image of a handwritten digit into one of 10 classes representing integer values from 0 to 9 , inclusively.

The MNIST data is stored in the Tensorflow library, we can just import it from there:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data", one_hot=True)

data_size = 55000

img_x = np.array(mnist.train.images[:data_size]) #Image
img_y = np.array(mnist.train.labels[:data_size]) #Label
```

Algorithm 4.1: Import dataset.

We initialize some parameters and create placeholders,as **tf.placeholder**, which reserves the space, but has no data as a variable.

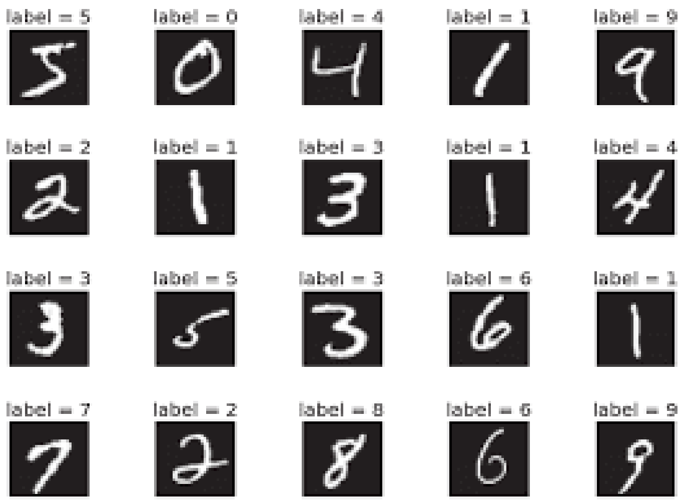

Figure 4.1: Mnist dataset.

```
img_size = 28
n_classes = 10
batch_size = 256

x = tf.placeholder('float', [None, img_size*img_size])
y = tf.placeholder('float')

keep_rate = 0.8
```

Algorithm 4.2: Initializing a few parameters.

The next step is to define the filters that we will train on the network. These filters will be defined as **tf.Variable**, that is to say as the variables that will be trained in our network.

```
with tf.name_scope('biases_set') as scope:
    biases = {  'b_conv1':tf.Variable(tf.random_normal([32])),
                'b_conv2':tf.Variable(tf.random_normal([64])),
                'b_fc':tf.Variable(tf.random_normal([1024])),
                'out':tf.Variable(tf.random_normal([n_classes]))}

gen_tensor = generated_tensor([5,5,1,32],"Conv1",[[0,1,2,3]]) #
    Convolution 1 filter
gen_tensor_1 = generated_tensor([5,5,32,64],"Conv2",[[0,1,2,3]]) #
    Convolution 2 filter
gen_fc = generate_fc([7*7*64,1024],'fc_1', [[1,2]], rank = 1) #
    Fully connected 1
gen_fc_1 = generate_fc([1024,n_classes],'fc_2', [[1,2]], rank = 1)
    # Fully connected 2
```

Algorithm 4.3: Variables to train.

In the algorithm 4.3 extract, we make use of class created by us that allows us to generate a filter or transfer function W with a given shape as the first parameter, a name to identify the filter in the logs and finally the separability of the filter.

These class called **generated_tensor** and **generate_fc**, in order to not overload the document with code, will be in the Appendix.

The next step is to define the operations of our network:

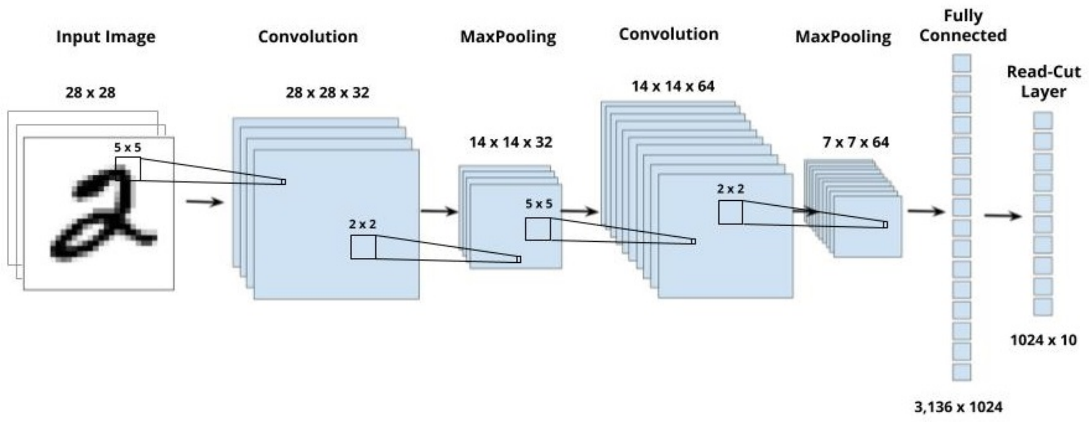

Figure 4.2: Network operations.

Our network consists of applying two convolution operations (see figure 4.3) to our input data, each of these operations followed by its rectified linear unit (ReLU) activation function and a maxpool function respectively.

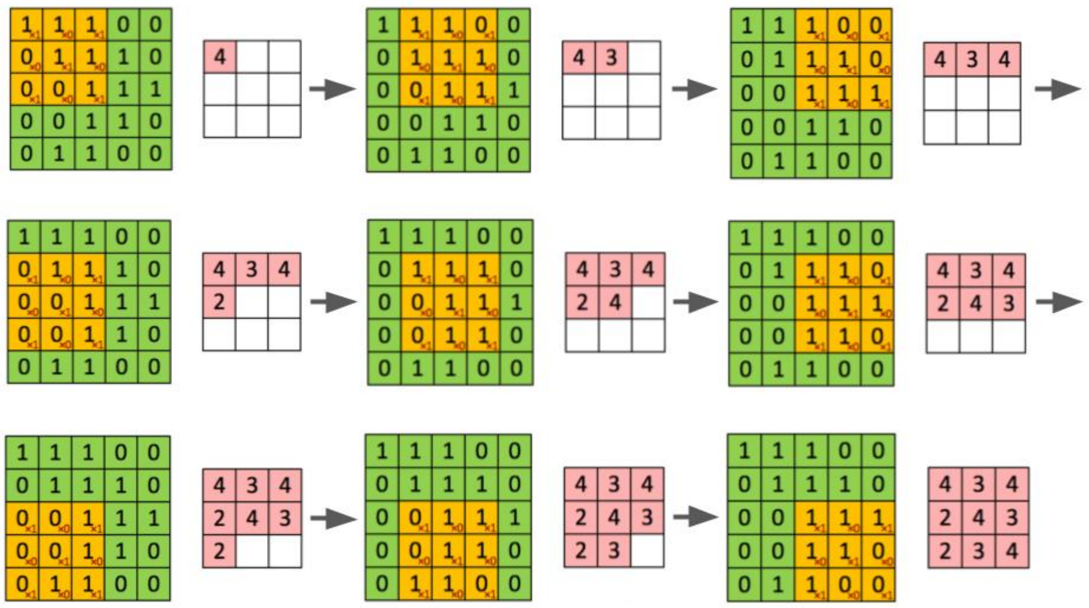

Figure 4.3: Convolution functionality example.

These two operations consist of applying a kernel to each image with the purpose of extracting its features, an example of the operation in figure X. In our network this kernel is 5 x 5.

The ReLU function (see figure 4.4) transforms the entered values by cancelling the negative values and leaving the positive values as they are. As advantages, this function is not limited and behaves very well with images and therefore in convolutional networks.

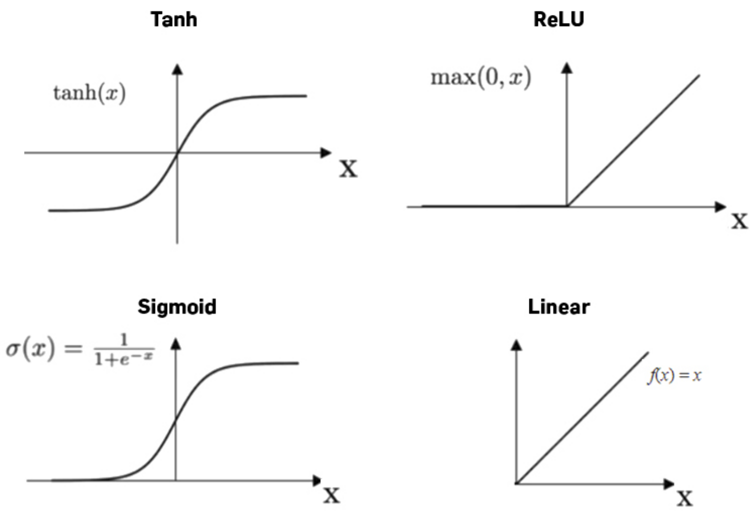

Figure 4.4: Activation function examples.

The idea of this function is to provide our model with the ability to generalize to a new set of data. As other options we have the Sigmoid, which limits us in a range of 0 to 1, perfect for models that seek to predict a probability.

```
def conv2d(x,W):
  return tf.nn.conv2d(x,W,strides=[1,1,1,1], padding='SAME')
# ...
conv1 = tf.nn.relu(gen_tensor.conv2d(x,W_conv1)+ biases['b_conv1'])
# ...
conv2 = tf.nn.relu(conv2d(conv1, W_conv2) + biases['b_conv2'])
```

Algorithm 4.4: ReLU + Convolution functions

The purpose of the maxpooling operation is to reduce the dimensionality of the data while maintaining the features extracted by the convolution .

The operation of this function is mainly to choose the size of a pooling window and the type, such as maximum, average or other type of value.

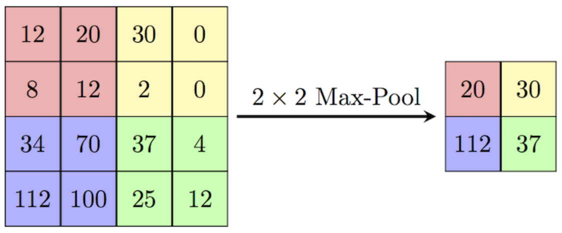

Figure 4.5: Maxpool operation.

The code would be as follows:

```python
def maxpool2d(x):
  return tf.nn.max_pool2d(x, ksize=[1,2,2,1], strides=[1,2,2,1],
   padding='SAME')

#...
conv1 = maxpool2d(conv1)
#...
conv2 = maxpool2d(conv2)
```

Algorithm 4.5: Maxpool function

As a summary of the mentioned operations we have the convolutions that will extract the features of the images based on the chosen kernel. Through the ReLU activation function we give our model the ability to generalize and with the maxpool functions we reduce the dimensionality of our model. As a last step we need to somehow classify all the data extracted up to this point, for this two completely connected layers will take care of the classification of each image to a label.

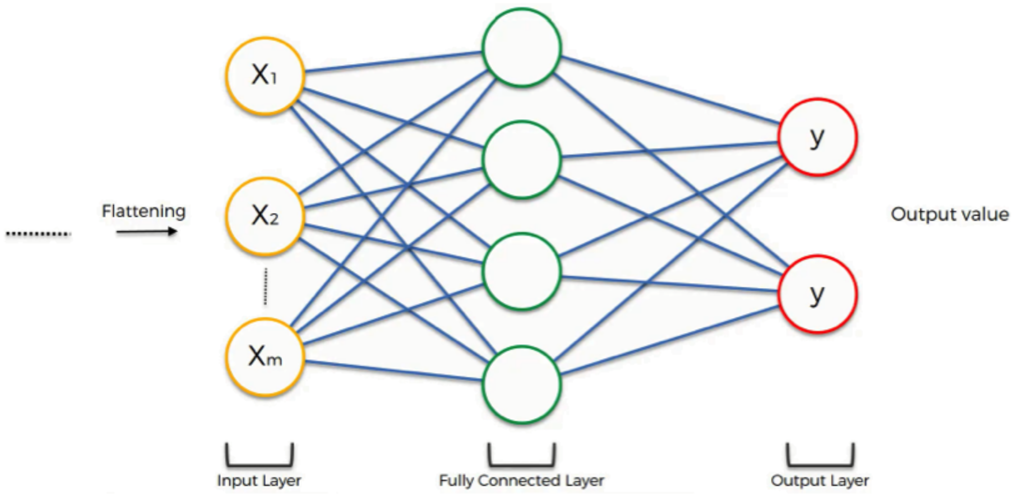

Figure 4.6: Fully-connected layers.

From our last operation Maxpool we make a flattening of our extracted data to a single vector and pass it to two completely connected layers. The first of these layers will be in charge of weighting each of the extracted characteristics in relation to its label. The second layer will provide a probability of each label.It is worth mentioning that to avoid over-fitting, a dropout is applied between both layers. This dropout randomly discards a percentage chosen by us from all the weightings.

```python
fc = tf.reshape(conv2,[-1,7*7*64]) # flattening

#Fully_connected 1
fc = tf.nn.relu(tf.matmul(fc, W_fc) + biases['b_fc'])
fc = tf.nn.dropout(fc, keep_rate)

```

```
#Fully_connected 2
output = tf.matmul(fc, out ) + biases['out']
```

Algorithm 4.6: Flattening and fully-connected layers

Before training the network we define our cost function and the optimizer we will use. In this case, as we are in a classification model we will use softmax cross entropy as a cost function. This function will check the labelling error of our images. In the simple neural network of the previous point we used the average of the quadratic error since we were facing a regression problem and not a classification one.

```
cost = tf.reduce_mean( tf.compat.v1.losses.softmax_cross_entropy(y,
    prediction))
optimizer = tf.train.AdamOptimizer().minimize(cost)
```

Algorithm 4.7: Cost and optimizer

The softmax function or standardised exponential function is a generalisation of the logistics function. It is used to compress a dimensional **K** vector **z**, from arbitrary real values to a dimensional **K** vector $\sigma(z)$, from real values in the range $[\mathbf{0},\mathbf{1}]$.

The function is given by:

$$\sigma : \mathbb{R}^K \rightarrow [0,1]^K$$

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} : j = 1, ..., K$$

The Softmax function calculates the probability distribution of the event over 'n' different events. In general terms, this function will calculate the probabilities of each target class over all possible target classes. The main advantage of using Softmax is the range of output probabilities. The range will be 0 to 1, and the sum of all probabilities will equal one. If the Softmax function used for the multiple classification model returns the probabilities of each class and the target class will have a high probability.

Cross entropy defined as,

$$H(p,q) = -\sum_{\forall x} p(x) log(q(x))$$

indicates the distance between what the model believes the output distribution should be, and what the original distribution really is.

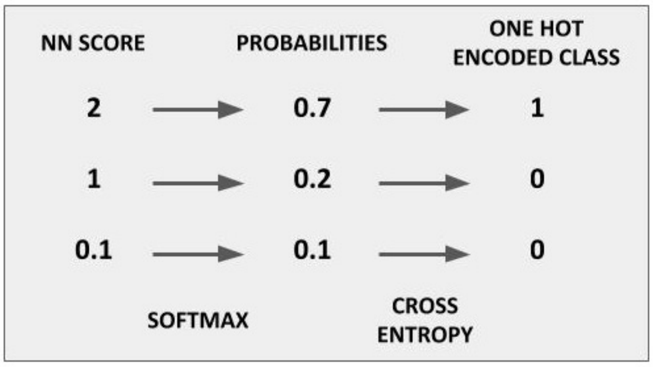

Figure 4.7: Softmax and cross-entropy example.

We propose Adam optimizer, a method for efficient stochastic optimization that only requires first-order gradients with little memory requirement. The method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients; the name Adam is derived from adaptive moment estimation. [4]

```
correct =tf.equal(tf.argmax(prediction,1),tf.argmax(y,1))
accuracy = tf.reduce_mean(tf.cast(correct, 'float')).eval(eval_dict
   )
```

Algorithm 4.8: Accuracy function

With all the architecture of the stability model we create the session in Tensorflow and train it by measuring the accuracy of the correct classifications with respect to the total number.

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    eval_dict = {x:mnist.test.images, y:mnist.test.labels}

    for epoch in range (epochs):
      temp = 0
      loss = 0
      for aux in range(int(data_size/batch_size)):
        temp += 1
        ###############################################
        feed_dict= {x: epoch_x, y: epoch_y}
        for gen_tensor_instances in layers:
          feed_dict.update(gen_tensor_instances.feed_dict_gt)

        # feed_dict = build_feed_dict(batch_size, layers, aux)
        ################################################
        _, c = sess.run([optimizer,cost ], feed_dict=feed_dict)
        loss += c


      print('temp', temp)
      print('Epoch', epoch, 'completed out of',epochs,'loss:',loss)

      #Build eval_dict
      for i in layers:
        eval_dict.update(i.feed_dict_gt)

      correct = tf.equal(tf.argmax(prediction,1),tf.argmax(y,1))
      epoch_acc = tf.reduce_mean(tf.cast(correct, 'float')).eval(eval_dict)

      print('Epoch_acc: ', epoch_acc)
```

Algorithm 4.9: Tensorflow session

The 4.9 algorithm shows us an extract of the complete Tensorflow session that we will use to train the model. In the Appendix you will find the complete code.

Once the network is trained, we plot the evolution of accuracy over epochs (see figure 4.8).

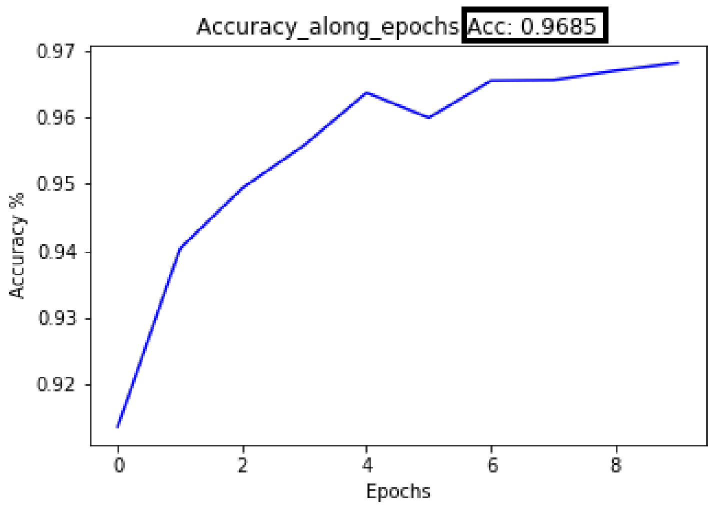

Figure 4.8: Accuracy along epochs.

We observe that throughout the **8 epochs** we obtain a good precision of **0.9685**,in the testing dataset. So this architecture in normal format trains well and is accurate. If we analyse the memory section of each variable trained, we are left with the following table:

Layer	Filter shape	Coefficients
Convolution2D_0	$[5 \times 5 \times 1 \times 32]$	800
Convolution2D_1	$[5 \times 5 \times 32 \times 64]$	52,600
Fully_connected_0	$[3136 \times 1024]$	3,211,264
Fully_connected_1	$[1024 \times 10]$	10,240

Table 4.1: Coefficients to train per layer.

We observe that for every era we have to train more than 3,000,000 coefficients between all the layers. The next step is to see if with the separate format applied to this convolutional network architecture we can reduce this amount of coefficients while maintaining the same accuracy.

4.2 Separated CNN

For this approach to the Convolutional Neural Network in a separate format we will maintain the same architecture as the normal format in order to legitimise the comparison and be on equal terms. That is, same operations in the same order, with the same cost function and the same optimizer.

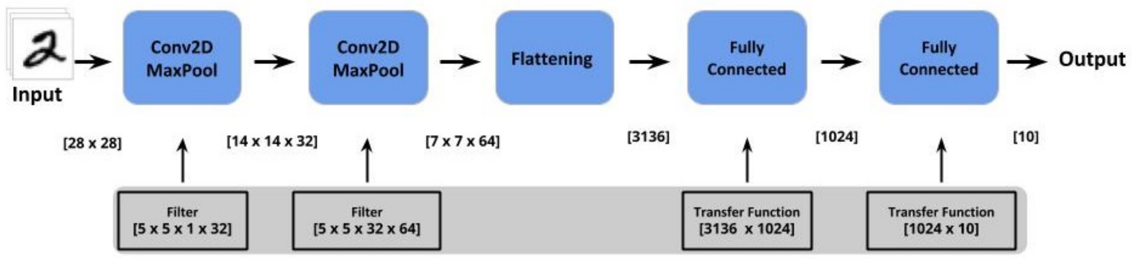

Figure 4.9: CNN architecture.

As we see in this architecture in the figure 4.9, the convolution layers are fourth-order tensors while the fully-connected layers are second-order tensors. Therefore, to apply a low order approach we decided to separate the convolution layers to two second order tensors and the fully-connected layers to two first order tensors. It would look like this:

Layer	**Normal format**	**Separated format**
Convolution2D_0	$[5 \times 5 \times 1 \times 32]$	$[5 \times 5]$ $[1 \times 32]$
Convolution2D_1	$[5 \times 5 \times 32 \times 64]$	$[5 \times 5]$ $[32 \times 64]$
Fully_connected_0	$[3136 \times 1024]$	$[3136]$ $[1024]$
Fully_connected_1	$[1024 \times 10]$	$[1024]$ $[10]$

Table 4.2: Comparison of normal and separated format.

The code for this separation would be as follows:

```
#Convolution 1
gen_tensor = generated_tensor([5,5,1,32],"Conv1",[[0,1],[2,3]])
#Convolution 2
gen_tensor_1 = generated_tensor([5,5,32,64],"Conv2",[[0,1],[2,3]])
#Fully_connected 1
gen_fc = generate_fc([7*7*64,1024],'fc_1', [1,2], rank = 1)
#Fully_connected 2
gen_fc_1 = generate_fc([1024,n_classes],'fc_2', [1,2], rank = 1)
```

Algorithm 4.10: Separated tensors on variables.

As explained above in the **generated_tensor** method, the third parameter determines the separability of our original tensor. To take the case of convolution

1 as an example, we pass the tensor $[5, 5, 1, 32]$ to it and as a separation we want $[[0, 1], [2, 3]]$, which means a second order tensor with indexes 0 and 1, that is $[5, 5]$ and another second order tensor with indexes 2 and 3, that is $[1, 32]$.

Once all these separated tensors the idea is to train them separately and to measure the error in the reconstruction of the original tensors (see figure 4.10) and in the same way as the previous part when the precision in these separated tensors is stagnant to enrich the net with another set of separated tensors and to use the previous ones as price knowledge.

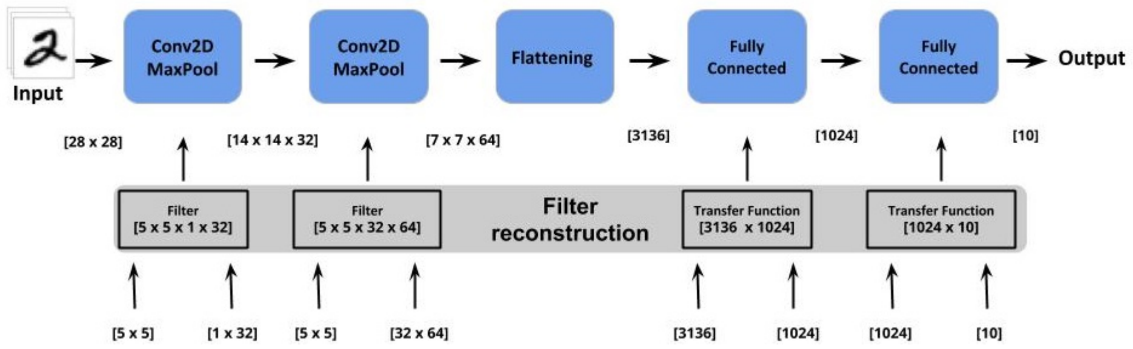

Figure 4.10: CNN architecture with tensor reconstruction.

This reconstruction is done in the operations of the network:

```
#Convolution 1
gen_tensor = generated_tensor([5,5,1,32],"Conv1",[[0,1],[2,3]])
layers.append(gen_tensor)
W_conv1 = gen_tensor.filter_reconstruction() #<---------
conv1 = tf.nn.relu(gen_tensor.conv2d(x,W_conv1)+ biases['b_conv1'
 ])
conv1 = maxpool2d(conv1)

#Convolution 2
gen_tensor_1 = generated_tensor([5,5,32,64],"Conv2"
 ,[[0,1],[2,3]])
layers.append(gen_tensor_1)
W_conv2 = gen_tensor_1.filter_reconstruction()  #<---------
conv2 = tf.nn.relu(conv2d(conv1, W_conv2) + biases['b_conv2'])
conv2 = maxpool2d(conv2)

fc = tf.reshape(conv2,[-1,7*7*64])
#Fully_connected 1
gen_fc = generate_fc([7*7*64,1024],'fc_1', [1,2], rank = 1)
layers.append(gen_fc)
W_fc = gen_fc.filter_reconstruction() #<---------

fc = tf.nn.relu(tf.matmul(fc, W_fc) + biases['b_fc'])
fc = tf.nn.dropout(fc, keep_rate)

#Fully_connected 2
gen_fc_1 = generate_fc([1024,n_classes],'fc_2', [1,2], rank = 1)
layers.append(gen_fc_1)
out = gen_fc_1.filter_reconstruction() #<---------
output = tf.matmul(fc, out ) + biases['out']
```

Algorithm 4.11: Filter reconstruction functions.

We run the network with these modifications, adjust all the parameters of the enrichment part and obtain the following result:

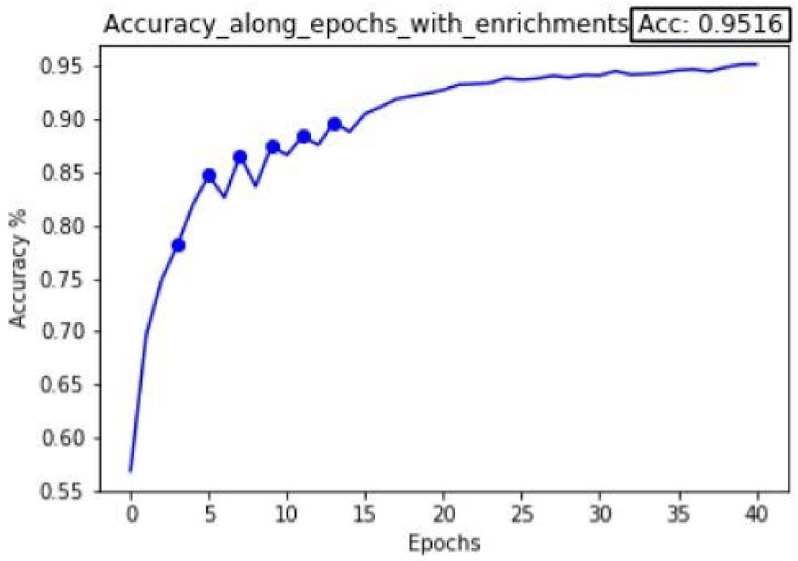

Figure 4.11: Accuracy evolution along epochs in a separated format.

What we see in figure 4.11 is the evolution of precision along the epochs. Each blue dot in the curve represents a moment where the network has been enriched. As we see, some enrichment are followed by a loss of accuracy. This is completely normal since by adding that set of trained variables to the previous data set as already acquired knowledge, these variables are initialized to a random with normal distribution so it can be adjusted again.

This curve shows that the network reaches an accuracy of 0.9516 with 6 enrichments, slightly less than the normal format with an accuracy of 0.9685, so this architecture in separate format has obtained an accuracy very similar to the normal format.

If we analyse the aspect of the memory with this architecture we obtain the following table.

Layer	**Separated shape**	**Coefficients D = 6**		
Convolution2D_0	$[5 \times 5]$ $[1 \times 32]$	~~800~~	$(25 + 32) \cdot D$	**342**
Convolution2D_1	$[5 \times 5]$ $[32 \times 64]$	~~52,600~~	$(25 + 2,048) \cdot D$	**12,438**
Fully_connected_0	$[3136]$ $[1024]$	~~3,211,264~~	$(3,136 + 1,024) \cdot D$	**24,960**
Fully_connected_1	$[1024]$ $[10]$	~~10,240~~	$(1,024 + 10) \cdot D$	**6,204**

Table 4.3: Coefficients to train per layer in separated format.

From the table 4.3, if we compare layer by layer we obtain the following comparisons.

In the case of the first convolution we go from training a fourth order tensor equivalent to 800 coefficients, to two second order tensor for each enrichment equivalent to (25 + 32) D. In this training with 6 enrichments it makes a total of 342 coefficients.

For the second convolution with the same form of tensor separation we went from 52,600 coefficients to (25 + 2,028) D. This makes 12,438 coefficients for 6 enrichments.

In the fully connected layers, we go from a second order tensor to two first order tensor. So we go from 3,211,264 coefficients to 24,960 and on the other hand from 10,240 to 6,204.

We can summarize all this to the fact that the separate model uses in memory 7,324 D coefficients with respect to the more than 3,000,000 of the normal format.

$$7,324 << +3,000,000$$

Given these results we conclude this point with the fact that this separate format with enrichments has a performance in terms of precision equivalent to the normal format used in the industry but with an improvement in terms of memory.

After all, if a matrix is of rank 1, it means that with a single set of row/column vectors the whole matrix could be compressed, that is to say that at the most an enrichment could be obtained the initial matrix. On the other hand, in the worst case if this matrix was of rank N, we would need N sets of row/column, that is N enrichments, to be able to represent the whole matrix. This would be equivalent to training in normal format.

This analysis does not lead to the next point of asking whether if it is necessary to separate all layers, or instead change the format of our network to increase separability.

4.3 Problem Analysis

Due to the question of whether all layers are separable for this problem.We decided to apply a singular value decomposition (SVD) to each of our reconstructed filters and transfer functions and to observe how many modes each of these layers has.

The analysis of SVD was made to CNN in a separate format after the training was completed. The process was to extract from the network the convolution filters and transfer functions of the fully connected layers and apply to these tensors a singular value decomposition. From these decompositions, we plotted the diagonal matrix, which is in charge of keeping the magnitude in increasing order in its diagonal (see figure 4.12).

What we observe in these four curves is that having applied 8 enhancements to the training, we see that especially in the two fully connected at the end they

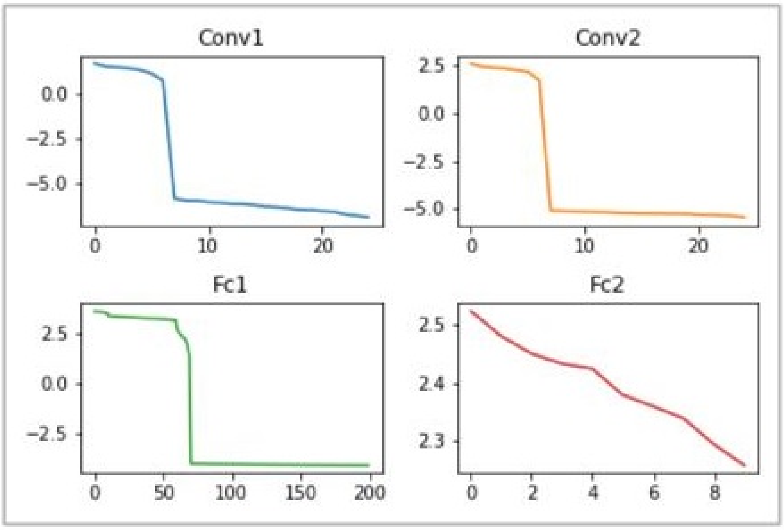

Figure 4.12: Singular value decomposition applied to CNN layers with 8 enrichments.

carry all the weight in the 8 enhancements. The first one fully connected has a rank of 10, so it falls in a value of 80 and the second one fully connected has a rank of 1 so it falls in 8.

This makes us think that the flattening operation breaks the tensorial structure of our network and consequently modifies its operational architecture.

4.4 Modifying separated format on CNN

Due to the problem analysis of each layer at the previous point, we decided to modify our network architecture. We assumed that by moving from the convolution layers we would mix the data with the flattening operation, where we would transform the whole tensor into a single vector and the fully connected layers.

We wanted an architecture that could keep the data structure well separated. So we eliminated the flattening operation and the two fully connected layers and replaced all this with a single fully connected layer which we call high order fully connected. This layer with a fourth order tensor to be trained that on one hand has the characteristics extracted from the convolution layers and on the other hand the part oriented to the classification of the images. It would look like this:

With this new network structure, just as we have done with the other filters, we have separated our new fourth-order tensor into two second-order tensor. So the $[7x7x64x10]$ tensor is separated into one of $[7x7]$ and another of $[64x10]$.

This change in Tensorflow is as follows:

```
#Convolution 1
gen_tensor = generated_tensor([5,5,1,32],"Conv1",[[0,1],[2,3]])
layers.append(gen_tensor)
W_conv1 = gen_tensor.filter_reconstruction()
conv1 = tf.nn.relu(gen_tensor.conv2d(x,W_conv1)+ biases['b_conv1'])
conv1 = maxpool2d(conv1)

```

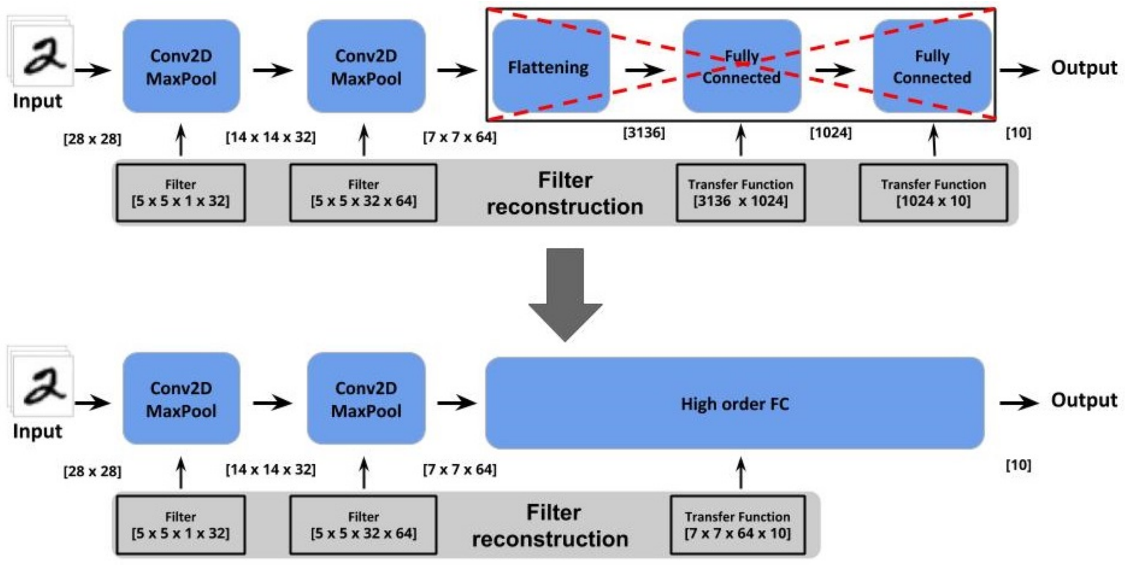

Figure 4.13: High order fully connected added in our architecture.

```
#Convolution 2
gen_tensor_1 = generated_tensor([5,5,32,64],"Conv2",[[0,1],[2,3]])
layers.append(gen_tensor_1)
W_conv2 = gen_tensor_1.filter_reconstruction()
conv2 = tf.nn.relu(conv2d(conv1, W_conv2) + biases['b_conv2'])
conv2 = maxpool2d(conv2)

#High Order Fully connected
gen_fc = generate_fc_4([7,7,64,10], "B", [[0,1],[2,3]], 50)
layers.append(gen_fc)
W_B =  gen_fc.filter_reconstruction()
op3 = tf.tensordot(conv2, W_B,([1,2,3],[0,1,2]))
output = op3 + biases['out']
```

Algorithm 4.12: High order fully connected.

Starting from figure 4.13, with the new structure if we add the separate tensors we would have this diagram:

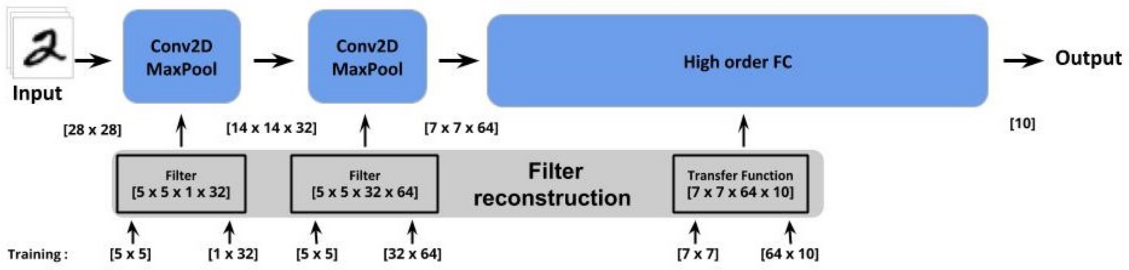

Figure 4.14: High order fully connected layer added in our architecture.

With the modification made and the parameters adjusted we train the network and obtain the following result:

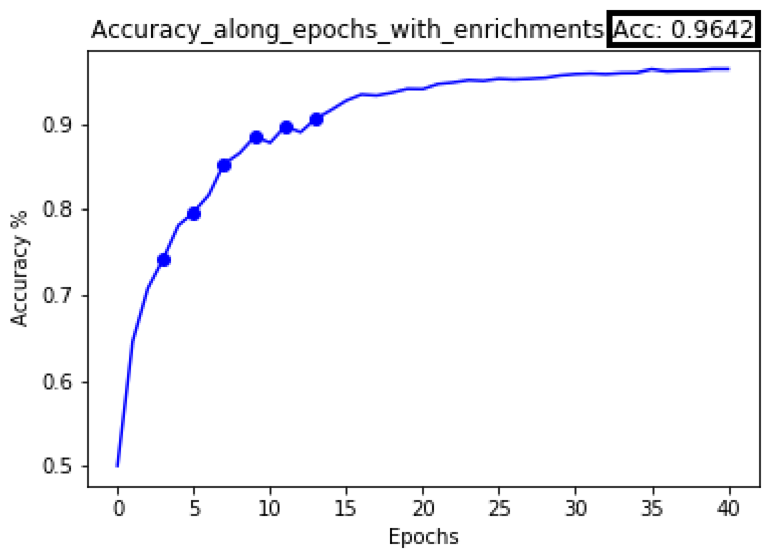

Figure 4.15: Accuracy evolution along epochs with high order fully connected layer.

In the plot in figure 4.15, we see the accuracy evolution over the epochs of our convolutional neural network replacing the flattened operation and the two fully connected layers with a of high order fully connected single layer.

With this modification we slightly increase the accuracy compared to our separate format without modification, with an accuracy of 0.9642 and 6 enrichment.

Our network has a very good performance equivalent to the previous formats. If we analyse the memory aspect, we are left with the following table:

Layer	Separated shape	Coefficients D = 6		
Convolution2D_0	$[5 \times 5]$ $[1 \times 32]$	~~800~~	$(25 + 32) \cdot D$	**342**
Convolution2D_1	$[5 \times 5]$ $[32 \times 64]$	~~52,600~~	$(25 + 2,048) \cdot D$	**12,438**
High order fully_connected	$[7 \times 7]$ $[64 \times 10]$	~~3,221,504~~	$(49 + 640) \cdot D$	**4,134**

Table 4.4: Coefficients to train per layer in modified separated format.

Analyzing the change made with respect to the previous separate format we see that our new high order layer fully connected by training two second order tensors has to adjust (49 + 640) coefficients by the number of enrichments made. In this training there have been 6 enrichments which makes us 4,134 coefficients. This value is considerably lower than 31,164 , the sum of the coefficients trained by the two layers we have replaced.

As a conclusion to this point we have to say that this modification based on the previous analysis has been positive in terms of performance because it has obtained an equivalent precision to the previous formats. On the other hand, from the point of view of the memory, this format has managed to reduce a little more the number of coefficients to adjust with respect to the separate format.

4.5 Results

We have dedicated this chapter to apply the methodology of separate format to a convolutional neural network architecture. We will use this point to compare the results studied previously in each of the formats. On the one hand, the non-separate or normal format commonly used in the industry, on the other hand the separate format maintaining the same structure as the normal format and finally a separate format with a modified structure. First we will make a comparison of the performance of each format. Then, in the memory aspect, we will compare layer by layer the coefficients adjusted in each format. And to finish the coefficients adjusted in each format.

	Normal format	**Separated format**	**Modified separated format**
Accuracy	**0.9685**	**0.9516**	**0.9642**

Table 4.5: Performance comparison table for each format.

Starting with performance, it is worth noting that we have not taken time into account throughout the work. The time is a factor that varies a lot depending on which CPU or graphic is executed and how overloaded each machine can be in each moment, and on the other hand the algorithms can always be optimized more. Looking at table 4.5, we see that the difference in precision in each format is almost identical, and the small differences in precision could be perfectly attributed to chance. But even if this were the case, the three formats have offered the same good performance, ideal for studying the differences in memory without assuming that an increase in the cost of memory could affect the accuracy.

Layer	**Normal format**	**Separated format**	**Modified separated format**
Convolution2D_0	**800**	**342**	**342**
Convolution2D_1	**52,600**	**12,438**	**12,438**
Fully_connected_0	**3,211,264**	**24,960**	∅
Fully_connected_1	**10,240**	**6,204**	∅
High order fully_connected	∅	∅	**4,134**

Table 4.6: Coefficients to train per layer in each format.

Comparing the data stored for each layer in each format (see the table 4.6, we observe that in the convolution layers it does not affect a considerable improvement, going from 800 to 342 coefficients in the first convolution and from 52,600 to 12,438 coefficients in the second convolution. In contrast, the fully connected layer has been considerably reduced from 3,211,263 coefficients in the normal format to 24,960 coefficients in the separate format, which is approximately 128 times less. And if we also take into account that for the modified separate format we have replaced the two fully connected layers with just one, the reduction to 4,134 coefficients compared to the normal format is approximately 779 times less.

	Normal format	**Separated format**	**Modified separated format**
Coefficients to store	**3,274,904**	$7,324 \times D$	$2,819 \times D$
Coefficients stored in our trainings	**3,274,904**	**43,944**	**16,914**

Table 4.7: Performance comparison table for each format.

To conclude the comparison in the memory aspect we observe in table X, how the separate format has positively affected the cost in memory. What we also observe is that our 10-digit classification problem, from 0 to 9, does not require the use of three million coefficients to learn how to classify these images. In the case of the modified separate format with 6 enrichments, we approximate the convolution filters and the transfer functions of the fully connected ones with much less coefficients. In another classification problem, such as the classification of animals or plants, separability may be lower and therefore costs may increase and more enrichments may be needed.

We have applied this methodology of separate format to the interior of the network making use of what we have called enrichment. We could call it a separate internal format. In the following point we are going to try to take this separability out and apply it to data set, it would be a outer separated format.

CHAPTER 5
Physics Beta Reconstruction

In this chapter we will deal with the separate format from outside the network architecture. The idea is to create a problem as if it were a parametric equation of physics. Apply a separate format to our data set to reduce its dimensionality and train with respect to this reduced data set.

5.1 Problem Statement

Currently, generate parametric prediction models to simulate mechanical behaviour in industrial parts has a huge computational cost. As these models increase in parameters and complexity, computing these models with traditional techniques becomes an arduous task and if we also want these models in real time, this task becomes unfeasible.

These parametric models have as input data a set of parameters, such as external forces, conductivities, or for example dimensions, and as output data solutions to an unknown equation.

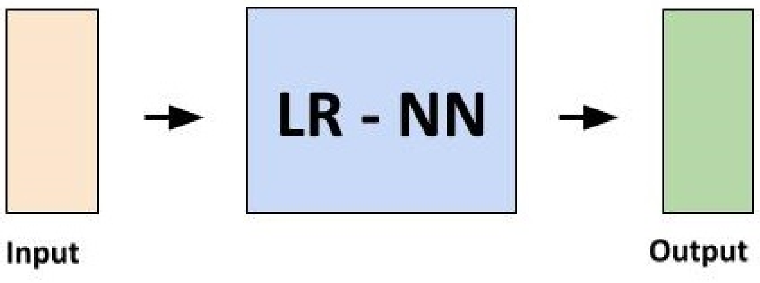

Figure 5.1: Expensive training in the neural network.

Among the different existing techniques, with a neural network (see figure 5.1) we could pass those input and output data, and the network would learn that transfer function that would become the equation that we do not know.

This neural network technique for a model that has a few parameters and therefore little complexity can be useful. But if this complexity increases immediately the cost increases and it becomes unusable.

One solution to this in order to continue using neural networks would be to apply some kind of compression to the data and train on this reduced set.

One way to do this compression is to apply one of the techniques we have already explained, such as singular value decomposition (SVD) to our output data. We choose the output data although it could also be the input data or both at the same time. Usually the industry moves in what is known as Smart Data. This term refers to training models with very little input data due to the high cost of obtaining it.

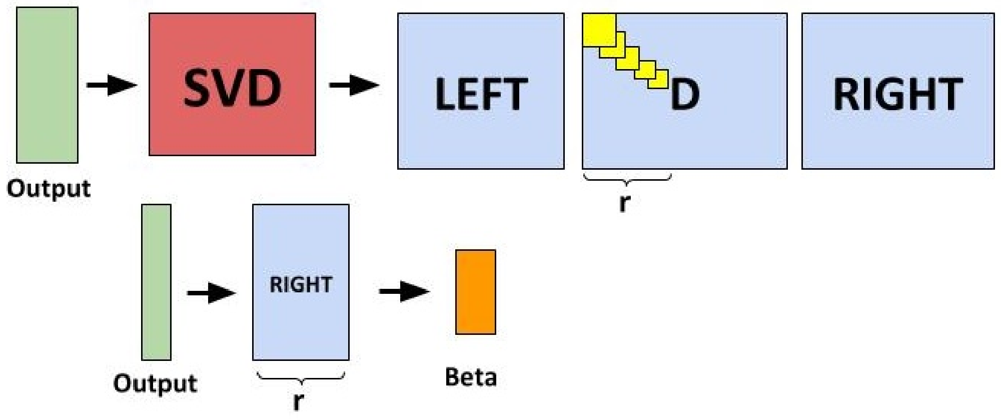

Figure 5.2: SVD applied to the output set.

Based on the functioning of the SVD explained above, the idea is to decompose our output set into three components (see figure 5.2), we have called them the left matrix, the diagonal D matrix and the right matrix. From the analysis of the diagonal d matrix and its number of linearly independent vectors r, we obtain a reduced matrix that we have called beta from the first r components of the right matrix.

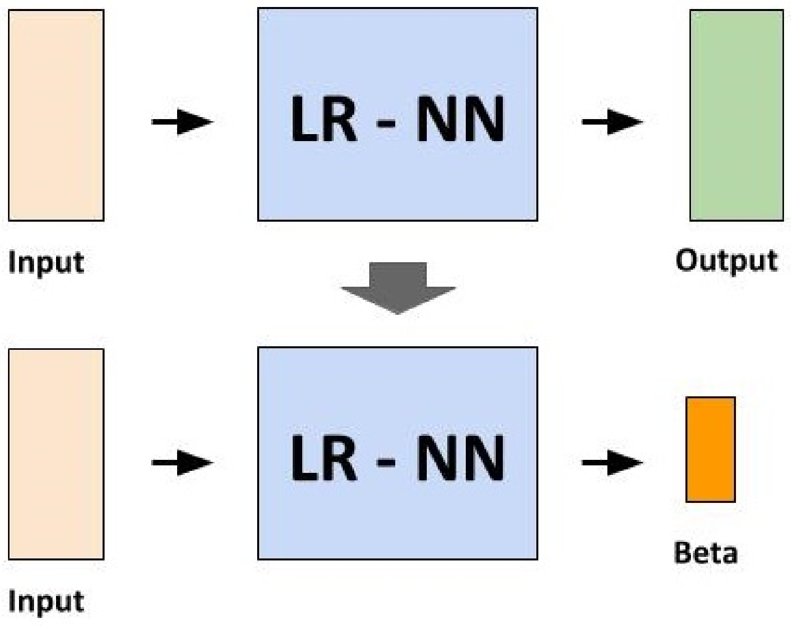

Figure 5.3: Compress training in the neural network.

Once this beta-reduced matrix is created, it would be to train the neural network with respect to beta. Once the network is trained, by passing it a new set of parameters as input, the model obtained would give us a reduced output, so the last step would be to reconstruct the complete output with the right matrix.

Obviously as with all compression techniques a loss of accuracy is assumed in favour of the computational cost and the storage cost reduction.

5.2 Model

Once the operation has been explained, we create our input and output data set based on the following parametric equation,

$$O(i_1, i_2) = ai_1 + bi_2$$

so that our output O depends on two input parameters i1 and i2.

To generate this data set in Python we have created a class that generates all the variables in the equation, applies the SVD to the output and returns the input data and reduced output.

```python
def create_data(self):
    np.random.seed(1)
    self.vars = np.random.rand(self.no_elements, self.no_parameters
   ) # Variables
    self.data_input = np.random.rand(self.no_inputs, self.
   no_parameters) #Input
    self.data_output = np.tensordot( self.data_input, self.vars,
   axes=(1,1)) # Output

def SVD_to_data(self, acc = 0.98):
    u, s, vh = np.linalg.svd(self.data_output)
    plt.plot(np.log(s))
    print(vh.shape)
    print('u shape:', u.shape, '\ndata.input shape', self.
   data_input.shape)

    sum_s = np.sum(s)
    self.no_eigenvectors = len(s)
    aux = 0
    for i in range(1,len(s)+1):
        aux += s[-i]
        cumulative_acc = 1 - aux/sum_s
        if cumulative_acc > acc :
            self.no_eigenvectors = len(s) - i
        elif cumulative_acc < acc:
            break
    self.right_matrix = u[:,0:self.no_eigenvectors]
    beta = np.tensordot(self.data_output, self.right_matrix, axes =
     (0,0))

    return beta
```

Algorithm 5.1: Create dataset

The function applied by the svd has a parameter which defines what precision we want to maintain with respect to the original. As we see in the different graphs in the figure 5.4, if we increase the number of parameters in our equation, the number of linearly independent vectors increases accordingly.

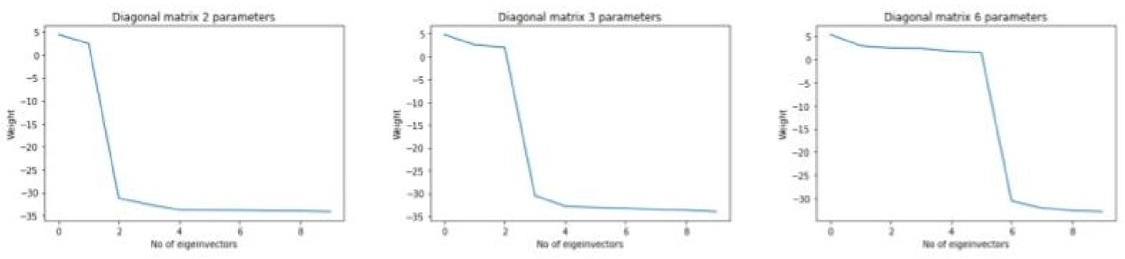

Figure 5.4: Diagonal matrix with different number of parameters.

The next step would be to create our neural network and train it to see the performance with this reduced data set.

```
def fc_neural_network(x):
    weights = {'W1':tf.Variable(tf.random_normal([no_parameters,8])
    ),
                'out':tf.Variable(tf.random_normal([8,no_elements]))
    }

    biases = {'b1':tf.Variable(tf.random_normal([8], dtype='float')
    ),
               'b_out':tf.Variable(tf.random_normal([no_elements]))}

    fc1 = tf.matmul(x, weights['W1']) + biases['b1']
    fc1 = tf.nn.relu(fc1)
    fc1 = tf.nn.dropout(fc1, keep_rate)
    output = tf.matmul(fc1, weights['out']) + biases['b_out']

    return output
```

Algorithm 5.2: Fully connected structure.

We created a simple two-layer fully connected with a ReLU activation function and an 80% dropout to reduce overfitting.

```
#Cost function
cost = tf.reduce_mean(tf.abs(y - prediction)/tf.abs(y))*100

#Optimizer
optimizer = tf.train.AdamOptimizer(learning_rate= 0.001).minimize(
    cost)
```

Algorithm 5.3: Cost and optimizer function.

As an error function we use the Mean Square Error (MSE) and Adam Optimizer. We proceed to analyse the conditions of our training and the results.

5.3 Results

With all the operational structure in place, we establish our data set. We will use 200 samples of 2 parameters and an output of 200 samples with 2000 elements. It is separated into 160 samples for training and 40 for testing.

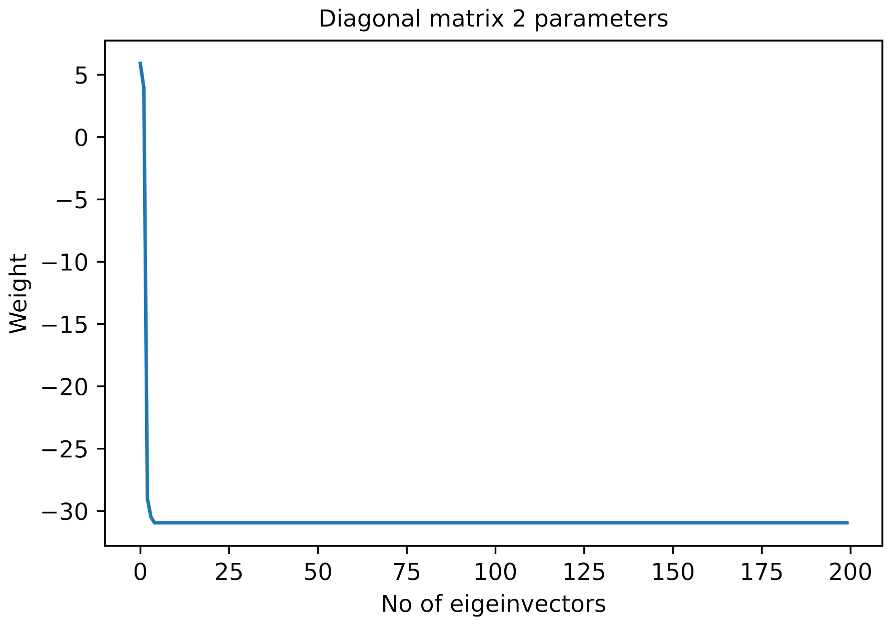

Figure 5.5: Diagonal matrix in our 200 samples compression.

When applying the compression to the output samples we observe that the number of linearly independent vectors (see figure 5.5) is 2, so for an original output of 200 samples of 2000 elements, we get our reduced output containing 200 samples of 2 element.

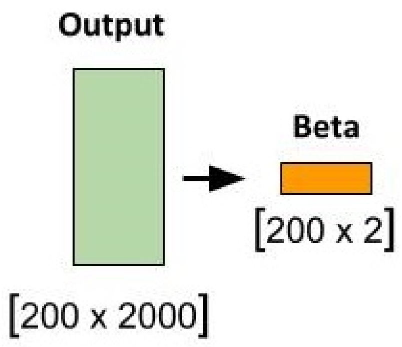

Figure 5.6: Output reduction.

With this reduced output we proceed to train the net and these are the three plots we get:

Because our error measurements are very close to 0, we decided in the training and test graphs to plot both the real and the predicted value of one of the two values of each reduced sample.

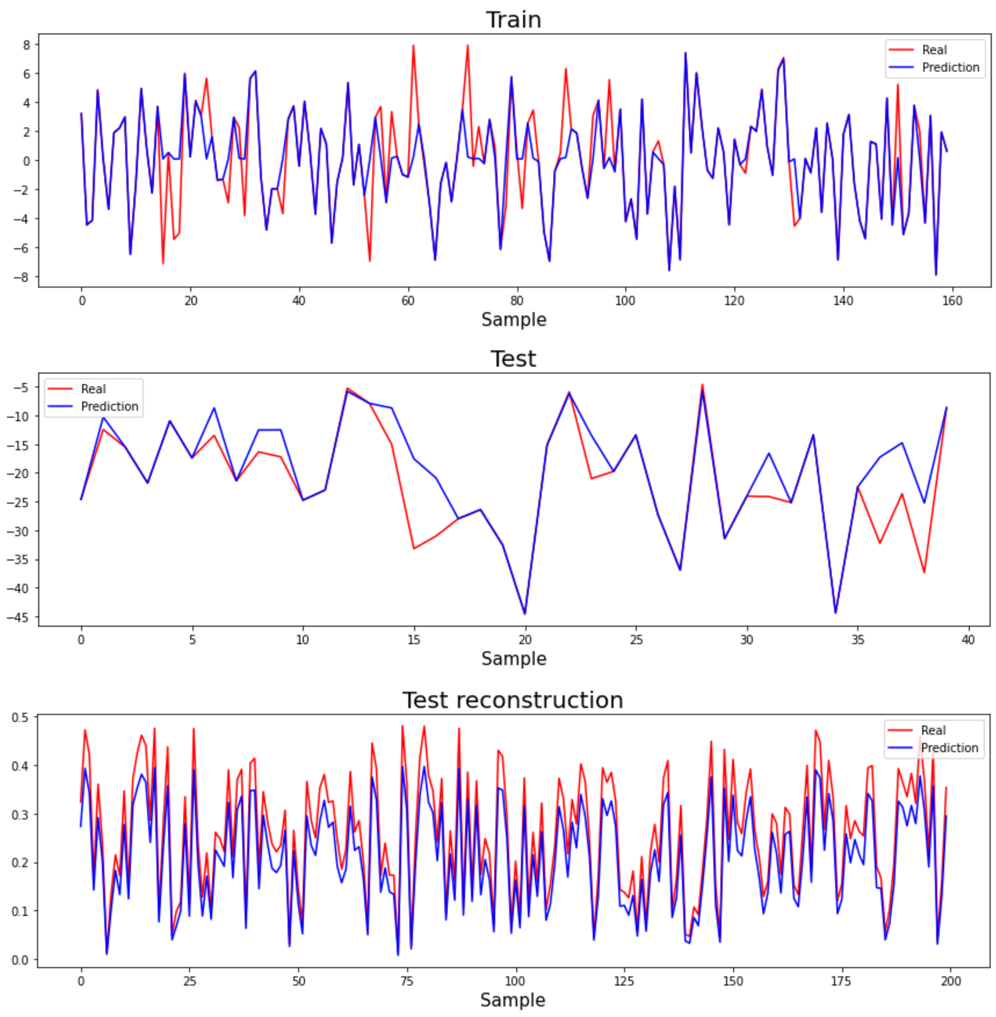

Figure 5.7: Output reduction.

As we see both the train and test charts, the predictions are very close to the actual curves. Therefore the reconstruction of the test into the original output we observe how the prediction correctly maintains the shape of the real curve.

As a conclusion to this last point, we can say that this compression of the output data that we have applied to our data set has meant an improvement in the performance of our network. After all, the SVD we applied to the data set is a not too expensive pre-process. We then train the network with the compressed output, and finally we rebuild it. In the end, the most time consuming part of the process is training the network, so using this type of compression benefits the learning process dramatically.

CHAPTER 6
Conclusions

Due to the high computational costs of ESI Group's simulated parametric models, we have dedicated all this work to finding methods that we can apply to neural networks to increase their performance either by building a separate format within the network or a applying a compression of the data outside the network. The separate format offers the network some versatility because as the accuracy of our network stagnates, enhancements will be added until a point where it cannot be adjusted any further. This method adapts to the complexity of the problem, since an easily separable problem will need few enhancements while a very complex one will need more.

On the other hand, because the complexity of those parametric simulation models, this compression technique is a good solution to reduce the dimensionality of the models. This technique applied outside the network training as a pre-process offers great advantages in time and storage.

The compression technique that we have called Beta-reconstruction applied outside of network training as a pre-process offers great improvements in time and storage. This reduction technique is a good solution to minimize the dimensionality of the models to something more affordable in computer terms that also allows to be applied to real time models.

Bibliography

[1] Vincent-Pierre Berges, Priyanka Rao, Reid Pryzant. Reinforcement Learning for Atari Breakout *Project paper*, University of California, Davis.

[2] Szu, Harold and Rogers, G. Generalized McCullouch-Pitts neuron model with threshold dynamics *07-1992*, pages 535 - 540 vol.3.

[3] Jerome H. Friedman. Greedy Function Approximation A Gradient Boosting Machine *IMS 1999 Reitz Lecture*.

[4] Diederik P. Kingma, Jimmy Lei Ba Adam: A Method for Stochastic Optimization *Machine Learning*.

[5] Martin Riedmiller Advanced supervised learning in multi-layer perceptrons — From backpropagation to adaptive learning algorithms *Computer Standards & Interfaces, Volume 16, Issue 3, 1994*.

[6] H.B. Barlow Unsupervised Learning *Neural Computation 1989 1:3, 295-311*.

[7] Chinesta, F. Ammar, A. Cueto, E. Proper generalized decomposition of multiscale models *International Journal for Numerical Methods in Engineering*.

[8] Y.C. LIANG, H.P. LEE, S.P. LIM, W.Z. LIN, K.H. LEE, C.G. WU, PROPER ORTHOGONAL DECOMPOSITION AND ITS APPLICATIONS—PART I: THEORY *Journal of Sound and Vibration,Volume 252, Issue 3, 2002, Pages 527-544*.

[9] Donald Gillies. *Artificial Intelligence and Scientific Method*. Oxford University Press, 1996.

[10] Cha ZhangYunqian Ma *Ensemble Machine Learning*. Methods and Applications, Springer, Boston, MA.

[11] Richard S. Sutton, Andrew G. Barto *Reinforcement Learning, second edition: An Introduction*. Adaptive Computation and Machine Learning series.