Università degli Studi Di Padova

Dipartimento di Fisica e Astronomia "Galileo Galilei"

Corso di laurea in Fisica

TESI DI LAUREA IN FISICA

Deep Learning techniques to search for New Physics at LHC

Relatore
MARCO ZANETTI

Candidato:
GAIA GROSSO

ANNO ACCADEMICO 2016-2017

Sommario

L'obiettivo di questa tesi è applicare i più recenti algoritmi di apprendimento automaticoper risolvere problemi di discriminazione tra processi di segnale e processi di fondo in fenomeni di collisione tra particelle elementari che hanno luogo a LHC, l'acceleratore di particelle di Ginevra. La prima parte del lavoro consiste in un'introduzione sull'algoritmo di apprendimento automatico e sull'architettura delle reti neurali usate per implementarlo. Successivamente viene fornita una breve descrizione dell'esperimento fisico che si svolge a LHC e delle usuali tecniche di analisi. Infine, dopo aver delineato il particolare processo di segnale da noi preso in considerazione, vengono presentati i principali risultati ottenuti dal nostro studio. In particolare verrà provata la tesi secondo la quale architetture neurali complesse, in particolare le reti profonde, allenate a processare dati relativi alle variabili cinematiche del fenomeno osservato sono in grado di uguagliare o persino superare le prestazioni di reti neurali più semplici, quelle non profonde, allenate con dati relativi a variabili non lineari ricavate dalle grandezze cinematiche per ridurre lo spazio delle fasi. Questo risultato è di forte interesse per i prossimi sviluppi nella ricerca a LHC dal momento che fornisce uno strumento valido per la selezione di segnali di nuova fisica, per i quali potrebbe non essere chiaro quale siano le variabili non lineari da ricostruire.

Abstract

The purpose of this thesis is to apply more recent machine learning algorithms based on neural network architectures in order to discriminate signal from background processes in particle collisions experiments which take place at LHC, in Geneva. First part of this work concerns with neural network architecture and learning algorithm brief description. Then we outline LHC experiments and analysis tools. Finally we introduce our work focusing on the physics of signal process used for our tests and we show our principal results. In particular, we shall confirm that complex neural architectures, namely deep networks, trained on raw kinematics features of particles produced in the process are able to equal or even surpass performances of simpler neural architectures, namely shallow networks, trained on few non linear variables derived from kinematic ones to reduce phase space. This result has a great impact on particle physics research carried on at LHC since it gives a valid alternative analysis tool to classify signals of new physics, especially when non linear features of interest will be yet unknown.

Contents

Introduction

The physics of data

Physicist's work is based on observing. He watches facts and drives conclusions about how they happen. Physics has been subject of human investigation since abstract concepts started being made by our mind. This because driving conclusions means deducing a general truth from a series of particular events having something in common. As for the mind of a child, the level of understanding of the world have been changed through the centuries from ancient times up to now, especially thanks to the development of technologies for observation, which have allowed to enlarge the range of *visible* phenomena. Mankind has actually surrounded himself with extensions of his senses able to collect information from the external world; as a consequence of that, he had to think up new analysis tools to help data to be interpreted. At the beginning, when the only objects men were able to scan were in scale of human visible, there was the first approach to logical and mathematical languages. Then they had to be expanded to keep up with growth of information complexity.

During last century, scientists came to explore both subatomic world and first stages of Universe; thanks to detectors, radioscopes, telescopes and others they collected at incredible speeds more and more large numbers of data which can assume a variety of shapes. Now they often are named banally *big data*. To handle them, human engine was no more sufficient and, as for human senses, also for human brain new extensions were designed, firstly *calculators* and then *computers*.

Calculators speeded up evaluation of a single mathematical operation, while computers allowed the assessment of multiple operations at a time, opening the way to more and more complex algorithms.

Nowadays computer science is an integral part of physicists' skill sets since Laws of Physics are often intricate functions and some of them can not be determined analytically. The principal usages of computers in physicists' work are numerically solving no analytical functions by approximation (*Deterministic Methods*) and generating data sample knowing their probability distribution (*Stochastic Methods*). The two methods satisfy mathematical calculus requirements. However, both of them can be implemented only if we take for granted to already know those functions we want to approximate from our knowledge of Physics Laws.
What if we don't have a theoretical model?

> *"Physics emerged from the realization that mathematics could be used as language to reason about our observations of the natural world. From Galileo onwards, the goal of modern experimental physics has been to isolate the phenomena of interest in an experiment for which theoretical physics could posit the underlying mathematical model by just thinking about it. This leads to a strategy that statistics and machine learning would call generative models, as they are capable to generate synthetic data prior to any observations (also known as predictions). In general physicists do not think of models being generative because what else could they be? But it is a choice nonetheless."* [1]

More recent developments in the field of informatics meet the requirements of abstraction capability necessary to infer concepts by evidence. They are application of *Artificial Intelligence*: complex algorithms able to fit a large number of given data to just as large number of parameters in order to estimate an engineering model, or rather an *empirical model*. Those algorithms actually *learn* in a similar way as a complex of human neurons do, thus they are called *neural networks*.
Men give to the neural network a set of known inputs and respective outputs, called *training data*; thanks to them the net can experience and varying the value of the parameters such that the resulting model would approximate the data as best as it could. These methods, for which model training outputs have to be already known to the net, are called *supervised learning algorithms* and generally they deal with events classification. Conversely, when outputs are not needed we talk about *unsupervised learning algorithms*, concerning clustering of events bearing similarities.
The exploit of machine learning techniques for data analysis has been delayed in joining physical matters, since the generative approach is more of all ingrained in this subject.

[1] *The physics of data*, Jeff Byers [1]

First evidences of machine learning great successes contributed to revolutionize the ideal of model in physics, thereby facilitating a more data-centric approach, described by *Bayesian statistics.*

The Bayesian statistics is based on moving the accent from the simple assumption of a theoretical model probability ($P(model)$), called *Prior*, to the deeper concept of "probability of the model, given the data" ($P(model|data)$), called *Posterior.* The latter can be computed as follows

$$P(model|data) = \frac{P(data|model)P(model)}{P(data)}$$

This expression tells us that, while the prior completely ignore how behaves real world, the posterior is weighted by the evidence of a certain data set, $P(data)$ and by the probability of observing that set of data taking for granted the model ($P(data|model)$).

Jeff Byers' words in merits summarize exhaustively what the point is:

> *"The prior over models does not represent anything physical in the world but, instead, our varying confidence about models in the world in our minds."* [2]

New computational tools allow to choose models comparing the theoretical ones with those constructed starting from real data.

Certainly, even though the name *artificial intelligence* might be misleading, learning algorithms are perhaps able to tell us something about how the data was created but nothing about why. This is matter for human deduction.

The topic of this thesis is to present the most recent achievement in particle physics data analysis due to supervised learning algorithms based on deep learning networks (Chapter 4 and 5). Before starting, we will introduce what machine learning is (Chapter 1 and 2) and the physical context of elementary particle within which we would like to apply it (Chapter 3). At the end of the essay we would finally show which advantages this kind of computational algorithms would bring to High Energy Physics statistical analysis (Chapter 6), and future prospectives about their deplyment (Chapter 7).

[2] *The physics of data*, Jeff Byers [1]

Part I

Neural Networks

Chapter 1

The architecture of Neural Networks

The purpose of this chapter is to describe an algorithm which, given some data about an event as inputs, may be able to label it as event of interest, namely a "signal", or not of interest ("background"). More precisely, we are interested in building a computing engine able to learn some kind of relation between several inputs and a single output, a map $f : \mathbb{R}^n \to \mathbb{R}$, which could be used as predictive model. Machine learning algorithms based on neural networks computation are what we need.

1.1 Perceptron

The first step to get into neural networks is to describe the simplest artificial neuron: the perceptron.

The perceptron gets a set of binary inputs $(x_1, ..., x_n)$ and returns a single binary *output* through the rule:

$$output = \begin{cases} 0 & if\ \sum_i^n w_i x_i \leq threshold \\ 1 & if\ \sum_i^n w_i x_i \geq threshold \end{cases}$$

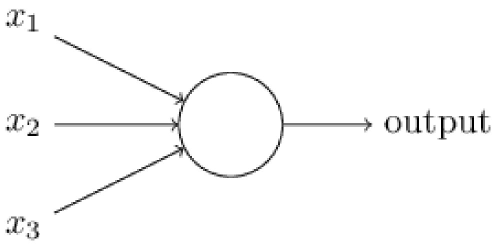


Figure 1.1: Diagram of a perceptron [2]

where the *threshold* value is a measure of how easy is to get the perceptron to fire (output a 1) and the weights, $(w_1, ..., w_n)$, enable the neuron to give different significance to every piece of information in input. It is possible to simplify this description by defining the bias

$$b = -thresold$$

and the scalar product

$$w \cdot x = \sum_i^n w_i x_i$$

where $w = (w_1, ..., w_n)$ and $x = (x_1, ..., x_n)$ represent the set of weights and that of inputs.
In this terms, the output becomes

$$output = \begin{cases} 0 & if\, w \cdot x + b \leq 0 \\ 1 & if\, w \cdot x + b \geq 0 \end{cases}$$

Every choice of the parameters (w, b) determines a different predictive model. Because of the binary nature of the output, the perceptron is an algorithm of binary classification.

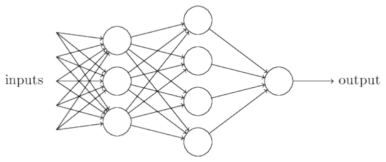


Figure 1.2: Neural network of perceptrons [2]

Considering the perceptron as a unit, it is possible to create more complex neural structures, where only few preceptrons (input neurons) take as input the set of data $(x_1, ..., x_n)$, while the others take as input the outputs of the previous, creating a net as shown in Figure 1.2.

This kind of neuron could be very useful to build logic circuits, since it makes simple computing fundamental operations as AND, OR, NAND, but it shows some limits when it is used to learn an algorithm. Let's suppose we have a neural network of perceptron which has learned an algorithm by finding the best set of parameters to describe the data; in order to state weather the net has worked well or not we need to associate an error to the model. The learning algorithm works well if small changes in the model lead to small changes in the output. For a perceptron, the smallest change in the output is 1, which is actually the only one possible; so every small variation in one of the parameter, big enough to provoke a variation, completely overturns the result. This is why the perceptron can not be a good choice as unit of building for a learning network. The matter to be solved is the discreteness of the output.

1.2 Sigmoid

Sigmoid neuron overcomes the lacks of perceptron neuron. It takes a set of real inputs, included between 0 and 1, and return a single real output, included between 0 and 1. To compute the output, first the variable z is evaluated as

$$z = w \cdot x + b$$

and then z is used to evaluate the *logistic function* (or *sigmoid function*)

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

which is a specific form of what is called, in terms of neural network, *Activation function*. It is evident that both perceptron activation function and sigmoid one are particular forms of the *Fermi function*

$$F(z) = \frac{1}{1 + e^{-az}}$$

with different values of the parameter a: $a = 1$ for the sigmoid function and $a \rightarrow +\inf$ for the perceptron one. So the perceptron can be thought as a simpler limit case of the sigmoid neuron.

As we can see from Figure 1.3, the logistic function takes values that smoothly vary in the interval between 0 and 1, up to saturation. This smoothness is crucial since it leads to the following expression for the output error

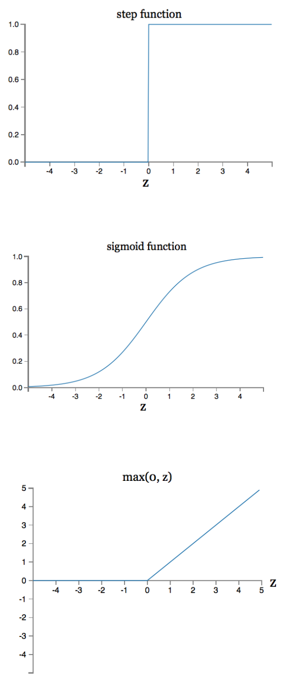


Figure 1.3: Activation functions [2]

$$\Delta outut \approx \sum_j \frac{\partial output}{\partial w_j} \Delta w_j + \frac{\partial output}{\partial b} \Delta b$$

which is a linear function of Δw_j and Δb. That means, by choosing small enough Δw_j and Δb it is possible to achieve any desired small $\Delta outut$. So, in principle, there is no limit to the improvement of a neural network, such that it makes sense looking for a best configuration in order to find more more correct models. This is why last works in artificial neural networks use sigmoid neurons as fundamental constituent.

Model of artificial neurons Sigmoid function is not the only function that can be used for artificial neural networks. As it was been remarked above, the smoothness is the required property for a good activation function. So any other function that regularly increase from any minimum value to a maximum one would be the same. Here there are some examples of neurons commonly used in artificial neural networks:

- **Sigmoid neuron:** already discussed in the section above $\sigma(z) = \frac{1}{1+e^{-z}}$
- **Tanh neuron:** is a rescaled version of sigmoid function $\sigma(z) = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
- **Rectified linear neuron:** its peculiarity is the absence of an asymptotic value for saturation $\sigma(z) = max(0, z)$

The reason why only few possible continuous functions are implemented in neurons is the suitability of their derivatives $\frac{\partial output}{\partial w_j}$ and $\frac{\partial output}{\partial b}$: functions composed of exponential expression are the easiest to derivate and imply the smallest computational effort. We have already encountered these partial derivatives in the expression of $\Delta output$, but their effective role will be explained completely in next chapter.

1.3 Neural Network

Now that is clear what an artificial neuron is and how it works, we can go on describing a complex structure composed of neurons: the neural network. [1]

In principle, any architecture based on neuron-like nodes is possible. Anyway learning algorithm performance is strictly dependent on the particular design chosen for the net. This is why neural networks researchers have developed typical design heuristics for the most common cases.

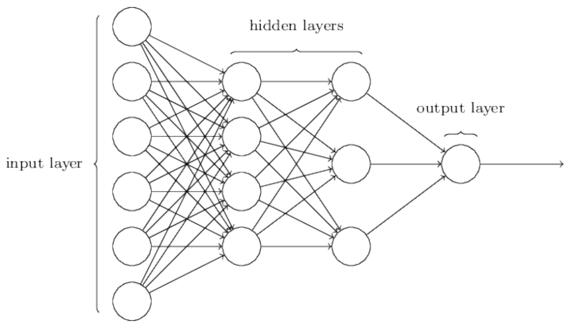


Figure 1.5: Feedforward neural network[2]

Feedforward neural networks. In general, as we can see from the example shown in Figure 1.5, a feedforward neural network is composed of an *input layer*, on the left, an *output layer*, on the right, and, in the middle, the so called *hidden layers*. This kind of multiple layers network is sometimes called *multilayer perceptrons (MLPs)* for historical reasons, even if it is not actually composed of perceptrons. The name *Feedforward*, instead, put in evidence the absence of loop in feeding the input neurons: information is always fed forward, never fed back, contrary to what happens in *Recurrent neural networks*.

Recurrent neural networks. It is a class of artificial neural networks where connections between neuron-nodes form a circle. For the purposes of this work, we will focus on feedforward neural networks since its learning algorithm is more powerful and spread to solve the kind of problem we deal with.

[1] **About notation:**

Before going on, we need to introduce a simple notation to refer to the weights and the biases of each neuron in the model. This is necessary since the number of model parameters can become very huge. Let's suppose having a deep neural network with n inputs, L hidden layers; be m_l the number of neurons of l-th layer and be the final output layer composed of one neuron only. Let's focus on the j-th neuron in the l-th hidden layer: this neuron would have one bias, and a number of weights equal to the number of output of the $(l-1)$-th layers. So every neuron has got $(m_{l+1}+1)$ parameters. This means for every layer there are $[m_l(m_{l+1}+1)]$ parameters, and for all the hidden layers their sum is equal to

$$\sum_{l=1}^{L}[m_l(m_{l+1}+1)] \tag{1.1}$$

The whole weights can be summed up in a 3-dimensional tensor, where the first label corresponds to the number of the layer, the second one to the number of the neuron in the considered layer, and the last one to the number of input to which it is associated. So, for the j-th neuron of the l-th hidden layer, the weights are represented as: $\vec{w}^l_j = (w^l_{j,1}, ...w^l_{j,m_{l-1}})$.
For the biases the case is simpler: a 2-dimensional tensor, namely a matrix, is worthwhile; so the bias of the j-th neuron in the l-th layer is represented as b^l_j.
To prevent incomprehensions for the rest of this work, we will assume this notation, writing first, as apex, the number of the layer and then, as subscript, the position in the layer. For the weights it will be often used the vectorial notation $\vec{w}^l_j$; anyway in the case we would specifically refer to the i-th coordinate, it would be written as subscript too $(w^l_{j,i})$.

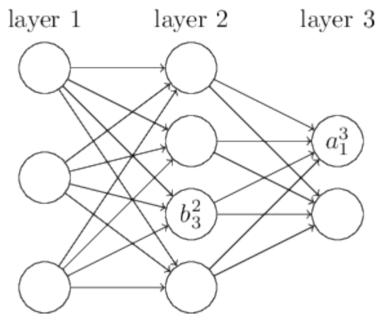


Figure 1.4: Example of notation. [2]

Chapter 2

Learning Algorithm

Last chapter we said we will focus on feedforward neural networks; they are made of a set of layers, each one composed of sigmoid neurons. We have already seen how a single neuron works and how the system of neurons in its totality constitutes a generative model due to its total set of parameters. But we have not mentioned yet how this complex architecture is able to learn the right model to evaluate outputs from dataset inputs. This is the purpose of this chapter.

2.1 Loss function

The first thing to be said is that the net needs to experience; in other words, it needs a set of inputs $\{\vec{X}\}$ and the relative set of expected outputs $\{y(\vec{X})\}$ to verify its performance; this data are called *training sample*. Larger the training sample is, more information the net is provided, thus the efficiency in learning grows. In fact the learning algorithm is essentially based on fitting model parameters to given data. In few words, we need to find the weights and biases so that the outputs of the net comes as close as possible to their true values, and this becomes easier and easier by increasing the number of events processed by the net. To quantify how much the model prediction gets near to the true values we introduce the *cost function*, $C((\vec{w},\vec{b}), y_{true}, y_{pred}(\vec{X}))$. The specific form of the cost function can vary but in any case it still depends on the parameters of the model, $(\vec{w},\vec{b})$, on the expected values for all the outputs, $\{y_{true}\}$ and on the values for the outputs predicted by the net $\{y_{pred}(\vec{X})\}$. Some of the most spread cost functions are:

- **Quadratic loss function:**
$$C(\vec{w},\vec{b}) = \frac{1}{2N}\sum_{\vec{X}} ||y_{true} - y_{pred}(\vec{X})||^2 \tag{2.1}$$
where N is the number of events processed by the net, $||\cdot||$ denotes the usual length of an n-dimensional vector [1], and the sum is over the whole training sample.

- **Cross-entropy loss function:**
$$C(\vec{w},\vec{b}) = \frac{1}{N}\sum_{\vec{X}}\sum_{j}[y^j_{true}\, ln(y^j_{pred}) + (1-y^j_{true})\, ln(1-y^j_{pred})] \tag{2.2}$$
where $\vec{y}_{true}$ and $\vec{y}_{pred}$ are expressed in components, so the sum over j denotes the sum over the n components of vectors $\vec{y}$[2].

Fixing the set of data, the cost function (also named as *loss function*) depends on the weights and the biases which constitute the neural model. As this parameters get close to their best values as the cost decreases. So, starting from an arbitrary set of parameters, $(\vec{w},\vec{b})_{init}$ [3], we need to find a good rule to update their values in order to decrease the loss function as fast as possible. In practice, this rule consists in an algorithm of optimization, briefly called *optimizer*.

[1]for an n-dimensional vector $\vec{v} = (v_1, ..v_n)$: $||\vec{v}||^2 = (v_1^2 + v_2^2 + ... + v_n^2) = \sum_j v_j^2$

[2]assuming that y_{true} and y_{pred} are n-dimensional vectors means that the net is characterized by a generic n-neuron output layer, so the net will return n values between 0 and 1; on the other hand, the sign of vector above the X remarks that also the input layer is multidimensional, be m its dimension, so $\vec{X} = (x_1, ..., x_m)$

[3]we will discuss later how choosing the proper initialization values.

2.2 Optimizer

As for the cost function, also the choice of the algorithm for optimization is free. Anyway, there are particular algorithms whose efficiency have been proved. In the paragraphs below we aregoing to describe the algorithm used for this work.

Gradient Descent. We have seen that C depends on the model parameters $\{\vec{w}, b\}_j^l$. Let's call them as an unique generic vector $\vec{\nu} = (\nu_1, .., \nu_n)$, and let's assume its n component to be continue. For small variations of $\vec{\nu}$ we can express a variation of the value of C as

$$\Delta C(\vec{\nu}) \approx \frac{\partial C}{\partial \nu_1}\Delta\nu_1 + ... + \frac{\partial C}{\partial \nu_n}\Delta\nu_n \tag{2.3}$$

If we define the gradient of C as

$$\nabla C = (\frac{\partial C}{\partial \nu_1}, ..., \frac{\partial C}{\partial \nu_n})$$

and all the single variations of ν_i as a unique vector

$$\Delta\vec{\nu} = (\Delta\nu_1, ..., \Delta\nu_n)$$

we can synthesize the expression 2.3 as follows

$$\Delta C(\vec{\nu}) \approx \nabla C \cdot \Delta\vec{\nu} \tag{2.4}$$

Since the cost function has to decrease so that the accuracy of the predictions grows, we need ΔC to be negative. To do so, just chose $\Delta\vec{\nu}$ as

$$\Delta\vec{\nu} = -\eta\nabla C \tag{2.5}$$

where η is a parameter called *learning rate* and it has to be $\eta > 0$. Thus the variation of C ($\Delta C = -\eta\nabla C \cdot \nabla C = -\eta||\nabla C||^2$) is certainly negative. More over, it can be proved that the choice of $\Delta\vec{\nu}$ in 2.5 makes $|\Delta C|$ maximus, speeding up the decreasing of C.
Anyway we can not leave out that for approximation in 2.3 to be valid, the variation of every single parameter ($\Delta\nu_i$) must be small enough, that means it has to be fixed how much at most any parameter could be changed fixing the norm of $\Delta\nu$; if we introduce this limit as a new parameter $\epsilon = ||\Delta\nu||$, η can be written as

$$\eta = \frac{\epsilon}{||\nabla C||}$$

and the law for updating ν becomes

$$\nu' = \nu - \eta\Delta C \tag{2.6}$$

Requiring ϵ to be small enough is equivalent to require a η has to. So, as we will see more precisely in next chapter, one of the principal aspects of the learning algorithm is the choice of the right learning rate in order to speed up the process but still being valid the approximation. In the specific case where the parameters $(\nu_1, ..., \nu_n)$ are the weights and the biases $\{\vec{w}, b\}_j^l$, the 2.6 becomes

$$w_j'^l = w_j^l - \eta\frac{\partial C}{\partial w_j^l} \tag{2.7a}$$

$$b'^l = b^l - \eta\frac{\partial C}{\partial b^l} \tag{2.7b}$$

Stochastic Gradient Descent. The equations 2.2 represents the Gradient Descent updating rule from a theoretic point of view. In practice, fixed a value for η [4], we need to compute the partial derivatives of C in all the parameters of the model. As already mentioned, the number of all the parameters can be very large. In every single update of the model, the computer need to keep saved all parameters values runtime; thus the memory of computer used to run the algorithm sets a limit to the complexity of the net to be processed and this limit can not be controlled by us. Another parameter which influences the attainments of optimization

[4]we will see that this value does not need to be constant during the whole process

algorithms is the number of events to be processed at a time, and this time we can somehow control it; let's see how.
We have said that to compute the gradient descent we need to calculate the partial derivatives of C. Let's give a look to the forms of these derivatives. Taking as example the *Quadratic cost function* expressed in 2.1, we can notice that C has the following form

$$C = \frac{1}{n}\sum_{\vec{X}} C_{\vec{X}}$$

where $C_{\vec{X}} = \frac{1}{2} \cdot ||y_{true} - y(\vec{X})_{pred}||^2$, $\vec{X}$ represents a single event processed and n is the total number of events. So a generic partial derivatives can then be expressed as

$$\frac{\partial C}{\partial \nu_i} = \frac{1}{n}\sum_{n} \frac{\partial C_{\vec{X}}}{\partial \nu_i}$$

and the gradient of C can be written as

$$\nabla C = \frac{1}{n}\sum_{n} \nabla C_{\vec{X}} \tag{2.8}$$

The 2.8 is a mean over the n events of the gradient of $C_{\vec{X}}$. From a computationally point of view it means that the algorithm would compute every single $\nabla C_{\vec{X}}$, keep it save till all the $\nabla C_{\vec{X}}$ would be computed and than sum up them. If the number of events is of the order of million events it is easy to understand that this would be very expensive and would require large memories. In order to overcome to this problem the algorithm actually implemented for gradient descent optimizer is the so called *Stochastic Gradient Descent*. The adjective *stochastic* suggests the presence of a random mechanism. In fact, the solution proposed by this methods is to take small sample of randomly chosen training inputs, assuming to be composed of m events $\{x_1, x_2, ..., x_m\}$, and estimate ∇C as a mean over this restricted sample:

$$\nabla C \approx \tilde{\nabla} C = \frac{1}{m}\sum_{\{\vec{X}_m\}} \nabla C_{\vec{X}_i} \tag{2.9}$$

Using the restricted sample, called *mini-batch*, the rule for update weights and biases in becomes

$$w'^l_j = w^l_j - \frac{\eta}{m}\sum_{i=1}^{m} \frac{\partial C_{\vec{X}_i}}{\partial w^l_j} \tag{2.10a}$$

$$b'^l = b^l - \frac{\eta}{m}\sum_{i=1}^{m} \frac{\partial C_{\vec{X}_i}}{\partial b^l} \tag{2.10b}$$

The update is repeated till all the events in the total sample are taken out. This loop is called *epoch*.
The set of the mini-batch size is fundamental: choosing it too large leads to reduce speed of the algorithm; on the other hand, choosing it smaller and smaller brings to more and more imperfect estimations of ∇C, since mean estimator tends to the true value if m tends to infinity. Anyway all we care about is the cost function goes down, no matter for the exact value of ∇C.
Moreover, it is important to set a proper number of total epoch for training the net: as it will be explained in subsection 2.3.1, the choice of epochs number is strictly linked to the problem of overfitting the net.

2.3 Fitting the model

In previous sections principal aspects of the learning algorithm has been described. Now we are going to show how this algorithm works, what troubles could occur and possible solutions to them.

2.3.1 The problem of overfitting

The power of deep neural network models is a huge number of well disposed parameters which lets the model to reproduce data with high precision. The negative aspect is that the net could even use all these parameters to overly reproduce the specific training data, becoming inefficient in describing new sets of data (as *testing samples*) and loosing its usefulness. So big number of parameters is not always synonym of good model.

When a net has already learned all the general information and starts updating in order to learn features specifically owned by training data, we say the net is *overfitting* (or also *overtraining*). In order to recognize when the net has effectively ended to learn we introduce the already mentioned *validation sample*[5]. To monitor the course of the training and notice when the net start overfitting, we observe the trend of accuracyover the epochs [6], respectively in training and validation sample. As we can see in examples shown in Figure 2.1, both loss function and accuracy tend to saturation for the validation sample, while for training sample they continue varying, although slowly. Since it is not used in cost function computation and so for parameters update, validation sample is not affect by the the continuous improvement of accuracy due to the model learning from data. In the first epochs, where the trend of the two accuracies is very similar, the neural net is learning those characteristics that the two sample have in common. After that, only the training accuracy continue improving, which means that the net is learning from the training sample but without any gain for the validation one[7]. That's the point when the training process has to be stopped in order to prevent overfitting.

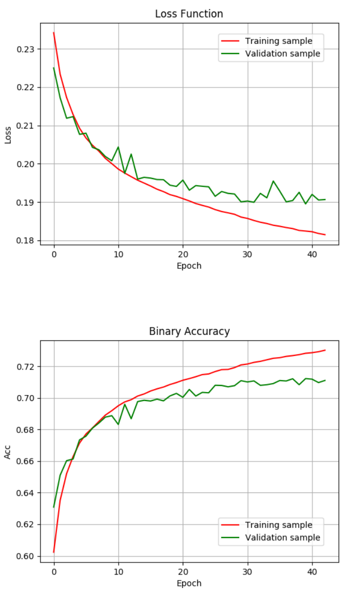


Figure 2.1: Comparing training and validation sample.

2.3.2 Reducing overfitting: regularization techniques

We have understood when to stop training. We are also interested in knowing if there are solutions which eventually reduce, at least partially, the overlap between training accuracy and validation one.
First of all, the gap goes down by increasing the number of data processed. This happens because a larger set of data means more experience for the net, which learn better to distinguish and emphasize the features owned by all the data and not specific of one set. Anyway it is not possible to increase the number of events processed indefinitely, as it would imply an higher and higher demand of memory, which has a limit fixed by the engine used.
A second possible solution is to reduce the size of the net, or reduce the number of neurons, since this means reducing the extent to which overfitting occurs. But if we would study the performances of more and more complex networks in order to achieve higher and higher accuracies, we would have fixed the architecture of the network and it would not be changed.
So, fixed the architecture of the net and maximum size of training sample, other solutions that do not require a rise in the sample size or a change of net architecture consist in modifying the algorithm; they are called *regularization techniques*. Let's see the principal ones.

L2 regularization also called *weight decay* consists in adding to the loss function the extra term

$$\Omega(w) = \frac{\lambda}{2n}||w||^2 = \frac{\lambda}{2n}\sum_{l,j} \vec{w}_j^{l2}$$

called *regularization term*. As usual n is the training sample size, while λ is a new hyper-parameter called *regularization parameter*. Thus the cost function becomes

$$C = C_0 + \Omega(w)$$

[5]Validation data is going to be used as checking sample for the right set of all the *hyper-parameter* of the net

[6]In Figure 2.1 is shown the plot of binary accuracy. The result of model predictions, which are float type, are rounded respect to 0.5; then is counted the number of true positive and true negative predictions ($N_{correct}$), whose value is equal to the relative label (y_{true}), and this sum is divided by the number of total events in the sample (N_{total}):

$$BinaryAccuracy = \frac{N_{correct}}{N_{total}}$$

.

[7]For the monitor of the algorithm we take as reference the accuracy; in fact what we are interested is how well the model reproduce the truth value (label) of the validation data. The loss function can be seen as a proxy of accuracy

and the law for updating

$$w'^l_j = w^l_j - \eta \frac{\partial C_0}{\partial w^l_j} - \eta \frac{\lambda}{n} w^l_j \tag{2.11a}$$

$$b'^l = b^l - \eta \frac{\partial C}{\partial b^l} \tag{2.11b}$$

L1 regularization adds an extra term to the cost function too. This term is given by

$$\Omega(w) = \frac{\lambda}{n} \sum_{l,j} |\vec{w}^l_j|$$

As for *L2 regularization*, also in this case the new term penalizes models with high weights, but the effects on the rule for updating is different respect to the previous one. With *L1 regularization* the rule is

$$w'^l_j = w^l_j - \eta \frac{\partial C_0}{\partial w^l_j} - \eta \frac{\lambda}{n} sign(w^l_j) \tag{2.12a}$$

$$b'^l = b^l - \eta \frac{\partial C}{\partial b^l} \tag{2.12b}$$

The correction to the non-regularized rule in 2.12a is given by a term whose absolute value is constant but whose sign changes according to that of the weight [8]. On the contrary, in 2.11a, the correction is proportional to the absolute value of the specific weight.
While in the first case weights always decrease of the same amount, in the second case the higher weights are most shrunk than the smaller ones. The result is that *L1 regularization* tends to drive towards zero most of the weights keeping few fundamental weights not null.

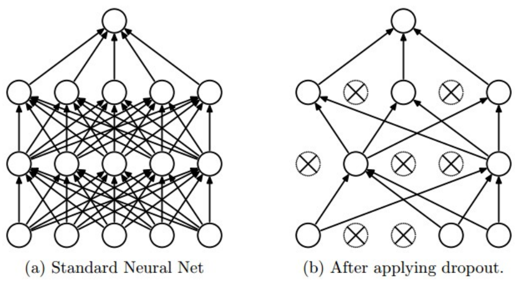


Figure 2.2: Example of dropout techinque of regularitazion. [4]

Dropout acts on the network without modifying the cost function. In practice applying dropout technique to an hidden layer consists in randomly hiding a rate of the neurons composing the layer. This mechanism constrains the net to reproduce the results without the contribution of a casual subset of parameters, thus any neuron is forced to learn regardless of the rest of the net. The advantage of dropout is that it brings to prformances resembling those obtained by mediating results of a large set of different models trained on the same data.

> *"Every time an input is presented, the neural network samples a different architecture, but all these architectures share weights."* [9]

That is the point: the effects of overfitting on the final parameters are different in every model architecture. This is why they are cut down by repeating the learning process on several nets and then mediating over all the results. Because of the computational and time expense of repeating over and over the learning algorithm for complex and big networks (which can need even few days), the dropout technique represents the more efficient alternative to that. Each hidden neuron is usually *dropped out* with a probability of 0.5 every time a new input is presented. Then, to compute the accuracy on the testing sample, all the neurons are kept and their output are multiplied by a factor 0.5 to take in account their probability. Set in this way, the net requires about double the number of epochs to converge, only a factor 2, which is a low cost compared with that necessary to repeat the all algorithm several times.

2.3.3 Weights initialization

Since the learning algorithm is not self-starting, in section 2.1 we have already mentioned the necessity of initializing model parameters. Now we are going to say something more about this topic. Let's try to understand why a proper choice for initialization is fundamental.

[8]Note that $sign(w)$ is not defined for $w = 0$; anyway, since we do not need to decrease w if it is already null, we can simply define $sign(0) = 0$ thus no changes are brought on it.

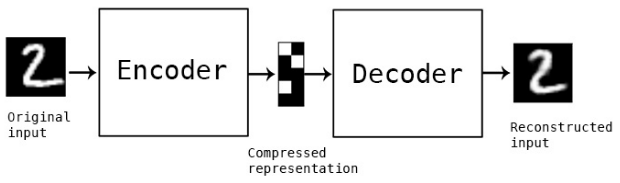
[9]From [5]

We start from the related problem of weights saturation. Let's suppose to initialize all the parameters of a considered net by random values distributed as a normal gaussian, with mean 0 and standard deviation 1. Imagine to compute the output of one neuron in the net; first we need to evaluate z as

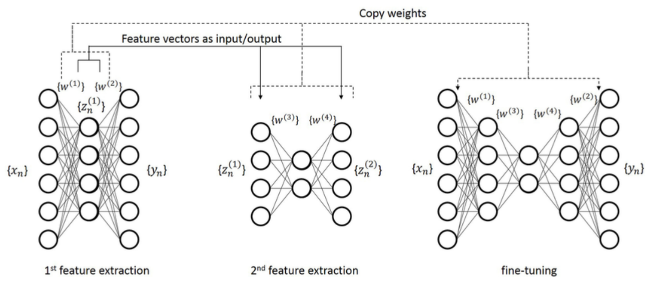
$$z = \sum_i w_i x_i + b$$

where the sum is over all the input x_i of the neuron, and then calculate $\sigma(z)$ using a specific chosen activation function. If we assume the input $\{x_i\}$ to be known, then z will result also distributed as a normal gaussian with mean 0 but standard deviation equal to $\sqrt{n}$ where n is the number of parameters, the weights and thee bias, of the neuron. Bigger the inputs size, bigger the number of weights and the standard deviation too. This means that there is a good probability that either $z \gg 1$ or $z \ll -1$, thus $\sigma(z)$ will result 0 or 1 with the same good probability. As can be seen in figure 1.3 of section 1.2, in general for the activation functions the derivative tends to 0 for $z \ll -1$ and $z \gg 1$ so the changes made thanks to the gradient descent algorithm will be very small and this means the net is going to learn slowly and that the improvement achieved by the model could be even so small that we could not observe it. In practice, it is said the network has saturated.
Saturation is a problem for the training: if a neuron accidentally evolves towards a configuration with $\sigma(z) \to 0$ or $\sigma(z) \to 1$ it has poor probability to come back, even if that configuration is not the best one; so the values of its parameters stay quite the same till the end of the training inhibiting the correct process learning.

Weights initialization determines also the speed of learning. A proper choice of initial model parameters allows to reach the best learned configuration in a fewer number of iterations. There are not precise dispositions on what is the best way of initializing, considering that the solution changes for each kind of net and dataset. However, the introduction of a new technique of unsupervised initialization could solve the question.



(a) **Autoencoder scheme** [15].



(b) **Example of autoencoder pretraining**. On the left the two autoencoder used to pretrain the first and the second layer in the deep network on the right.[17].

Figure 2.3

Autoencoder pre-training. In neural networks terminology, we speak about *Autoencoder networks* referring to those networks which learn to reproduce as outputs their own inputs. More precisely, as Figure 2.3a outlines, they are all characterized by a first encoding part and a second decoding one. Let's suppose to give n inputs to the autoencoder; the first side of neural architecture, the *encoder*, compresses all inputs information towards a final layer of m neuron; then, in the second side of the net, the *decoder* takes the *compressed representation*, i.e. the m neurons in the middle, and decompresses it up to a final n-neuron output layer. The final structure assumes the shape of two symmetric bottlenecks.

We are interested in Autoencoder networks due to their possible application on unsupervised pre-training techniques. Given a neural network and its associated model to be trained, we define *pre-training technique* a starting algorithm which trains other neural networks; we can extract from them the final values of some parameters of theirs and use them as initial values for the given net. The adjective *unsupervised* denotes that to run this algorithm it is not necessary to associate an already known output label to the input sample; this is true for autoencoder networks which compare their output to the initial input. Autoencoder networks can be used to build such an algorithm to determines layer by layer some initial values for weights and biases of a deep network DN. Taking as reference Figure 2.3b, we can describe an example of autoencoder pretraining[10]. Let's start from the first layer (L_1). We build an autoencoder having

[10]Pay attention because the figure can be misleading: the inputs and the outputs are represented with circular shape, while the

two layers: the first one (the $encoder_1$) takes the inputs and gives as output the same of L_1; the second one (the $decoder_1$) takes n_1 inputs and gives a number of outputs equal to the inputs one. The autoencoder is then trained comparing the outputs to the inputs till the algorithm has finished learning. Then we save the final parameters of the $encoder_1$ as initial value for L_1 in DN. Let's go to the second layer (L_2). As in the previous case, we build an autoencoder with two layers. This time the inputs are that obtained as output by $encoder_1$; the number of $encoder_2$ outputs is the same of L_2 and that of $decoder_2$ are once again the same taken as inputs by $ecoder_2$. After training the net, this time we save the final parameters of $encoder_2$. The methods is analogue for all the successive hidden layers till the end of DN. Thus we have built an algorithm which attributes weights and biases to our net without the need of any arbitrary assumption. Training techniques which make use of autoencoder pretraining are generally called semi-supervised learning algothms (SSL) for the presence of an initial unsupervised training and a final supervised fine-tuning. This computational tool is showing all its strength in several applications; some of them are mentioned in [16].

layers are actually the lines connecting the input and output circles. In fact model parameters denoted with $w^{(i)}$ are put between the circles columns, in correspondence of the lines connecting them

Part II

Searching for Exotic Particles in High-Energy Physics with Deep Learning

Chapter 3

The Physics of LHC

Up to now, it has been generally described a new computational engine that is neural networks machine learning. The purpose of this thesis is to apply such powerful tool to data of high energy physics collected at the Large Hadron Collider (LHC) in Geneva. LHC is the largest particle accelerator in the world. The four experiments carried on at LHC detect a huge number of adrons collisions per second[1] and for each event they record various forms of data. The large size, the variety and the high rate of collisions production at LHC are the reasons why they can be named as *big data* and they are one of the more suitable dataset for testing the efficiency of this kind of computational engine. On the other hand, employing more and more efficient machine learning algorithms to LHC datasets will let high energy Physics community to extract from data all the available information. This would allow to reduce the waste of data.

Let's look in detail how the experiments on LHC operate and what they observe. As we can see in Figure 3.1, the Large Hadron Collider is composed of one main ring, 27 km long, made of superconducting magnetic tubes inside which two separated protons beams run, one clockwise and the other counterclockwise. Thousands of magnets constituting the tubes are used to make beams trajectory curve enough to keep the particles inside the pipes; others are used to keep the particles in bunches in order to maximize the cross section; the rest of them is used to accelerate the particles so that they could acquire energy. The minor rings shown in the figure, the *Proton Synchrotron Booster (PSB)*, the *Proton Synchrotron (PS)* and the *Super Proton Synchrotron (SPS)*, are used in the preparatory phases of acceleration, just after the protons have been extracted from Hydrogen atoms. When the particles achieve the right energy the two beams flowing in LHC are made to collide inside four detectors: ALICE, ATLAS, CMS and LHCb (marked by yellow arrows in Figure 3.1).
The energy of the beams is fundamental since it determines the range of phenomena that could be observed after the collision. Since June 2015 proton-proton collisions has achieved the energy of 13 TeV.

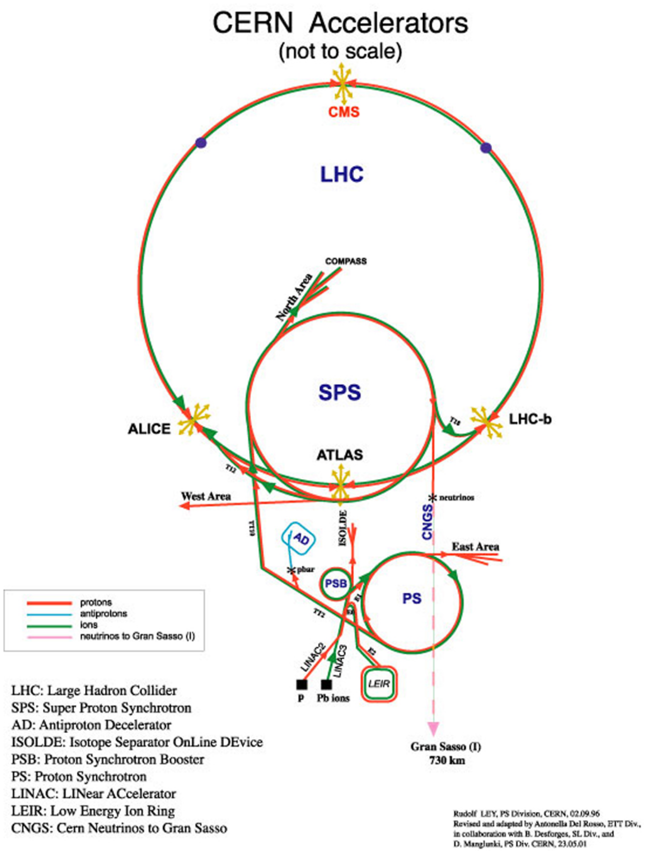
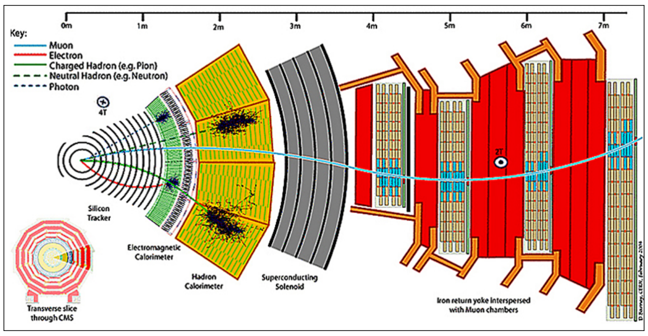

CERN Accelerators
(not to scale)
CMS
LHC
COMPASS
North Area
SPS
ALICE
LHC-b
ATLAS
T12
T18
West Area
neutrinos
CNGS
T110
AD
ISOLDE
East Area
pbar
PSB
PS
TT2
E1
E0
LINAC2
LINAC3
E2
LEIR
p
Pb ions
Gran Sasso (I)
730 km
protons
antiprotons
ions
neutrinos to Gran Sasso (I)
LHC: Large Hadron Collider
SPS: Super Proton Synchrotron
AD: Antiproton Decelerator
ISOLDE: Isotope Separator OnLine DEvice
PSB: Proton Synchrotron Booster
PS: Proton Synchrotron
LINAC: LINear ACcelerator
LEIR: Low Energy Ion Ring
CNGS: Cern Neutrinos to Gran Sasso
Rudolf LEY, PS Division, CERN, 02.09.96
Revised and adapted by Antonella Del Rosso, ETT Div., in collaboration with B. Desforges, SL Div., and D. Manglunki, PS Div. CERN, 23.05.01

Figure 3.1: LHC complex. [9]

[1]Three of them (named CMS, ATLAS and LHCb) only take care of proton-proton collisions while ALICE is concerned with collision between lead ions too.

Protons are not elementary particles, they are made of three quarks, two top and one down, keep together by *strong forces* mediated by *gluons*. This is why different combination of products are possible. The strong interaction acts as a *dynamic vacuum*, which means that all around the three main quarks are continuously generated and annihilated couples of *quark* and relative *antiquark*, or couples of gluons and anti gluons which determine, due to a screen effect, the intensity of the attraction or repulsion between the three static quarks. So, when two protons collide, actually all these particles merge themselves generating an explosion in which new particles combinations are pulled away with high kinetic energy and revealed by the detectors.
Particles detectors, usually cylinder shape, surround the point where the collision takes place; different technologies are used to observe different features of the particles going across them. The closest one is a *vertex detector* of silicon pixels and strips used to track charged particles near to the point of interaction; then there is the *charged particle tracking chamber* which distinguishes positive and negative charges measuring the curvature of the particles in a magnetic field; next layer consists in two *calorimeters*: one absorbing the *electromagnetic-showers* and the other the *hadronic-showers*; finally the *muon chambers* which absorb the heaviest particles called *muons*.


0m
1m
2m
3m
4m
5m
6m
7m
Key:
Muon
Electron
Charged Hadron (e.g. Pion)
Neutral Hadron (e.g. Neutron)
Photon
4T
2T
Silicon Tracker
Electromagnetic Calorimeter
Hadron Calorimeter
Superconducting Solenoid
Iron return yoke interspersed with Muon chambers
Transverse slice through CMS
Figure 3.2: Inside a detector (CMS). [11]

All these components together are able to absorb all the energy issued by the collision except that carried out by *neutrinos*, elementary particles which can not be detected through this kind of detectors.
From the pieces of information collected by each layer of the detector, high energy physicists are able to go back to the nature of the particles produced by the collision and study the fundamental Laws of Physics which regulate these processes. Anyway, almost always particles detected by instruments are not the initial products of the interaction. They constitute the traces of pre-existent particles so unstable that they decade into other particles just after the collision, before the instrument could detect them. This intermediate metastable state is called *resonance*.

Resonances and collisions The name *resonance* in subnuclear physics comes from the analogy with resonant systems in classical physics. In the simple case of a one dimensional model, given an object of mass m that oscillates along x with a proper frequency ω_0 and equilibrium in 0, if it is perturbed by a periodic external force with variable frequency ω,

$$F(t) = F_0 cos(\omega t)$$

then the solution for x is

$$x(t) = A cos(\omega t + \delta)$$

where the amplitude of oscillation is

$$A = \frac{\frac{F_0}{m}}{\sqrt{(\omega_0^2 - \omega^2) + \Gamma^2\omega^2}}$$

and the phase

$$\delta = arctg\left(\frac{\Gamma\omega}{\omega_0^2 - \omega^2}\right)$$

The effect of the forcer on the object is maximum when $\omega = \omega_R \approx \omega_0$, as in Figure 3.3.

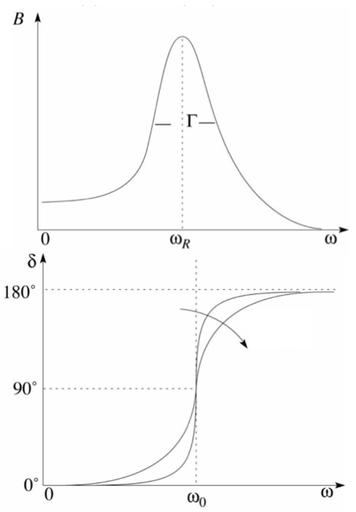


Figure 3.3: Resonance in classical systems

Metastable particles can be observed as resonances too. Interactions between elementary particles can be modeled as traditionally

- *elastic collision*: when particles before and after the interaction stay the same.
- *anelastic collision*: when particles come out from the collision are not the same that collided.

Total impulse and total energy laws of conservation delimit the range of energy and impulses each particle can reach after the collision, giving shape to the so called *phase space*. In physics the *phase space* is a multidimensional space where all the possible configurations of the system lay. Each configuration is described by particular values of the observables of the system.
After hadron-hadron collisions the system we deal with is composed of all the produced particles. Each of them is characterized by the measurements of three spatial coordinates:

- the momentum transverse to the beam direction (p_T)
- the polar angle (θ)
- the azimuthal angle (ϕ)

For convenience, at hadron colliders, the polar angle (θ) is replaced by the *pseudorapidity*, defined as

$$\eta = -ln\left(tg\left(\frac{\theta}{2}\right)\right)$$

So if we have n products, the phase space is $3n$-dimensional and each point of this space, identified with a $3n$-tuple of coordinates, determines a different possible configuration of the global system. Analyzing all the events stored during the run of the accelerator, physicists can build an histogram counting the events distributed on the phase space. Then this distribution can be compared to that expected by the theoretical models. Up to now, the model of reference for elementary physics is the *Standard Model (SM)*, a description of particle physics processes which seems to explain as well as possible the experiments results. The presence of anomalous peaks in the experimental density distributions which do not find correspondences in SM, suggest the presence of new resonances, that means new metastable particles, not included in the model, or not yet observed. A recent example of that kind of resonance is the discover in 2014 of the *Higgs boson*, firstly theorized in 1964. We re going to discuss about it again talking about the signal process simulated to make the dataset for our tests.

Data analysis: approximating the Likelihood
Finding resonances is anything but trivial for the following reasons. First of all, physicists need a huge number of events to carry out statistical analysis. Secondly, the phase space is so wide that they have to restrict it. But often it is not enough: resonances, let's call them *signals*, can be so slight that they can not significantly be discriminate from *background* events. Thus they need to make new variables from the raw ones which better highlight differences and then isolate a subspace of events choosing a range for them. For example, let's consider a reaction of the following type

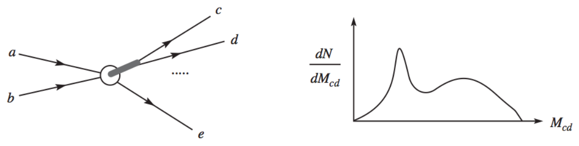


Figure 3.4: Schematic of a resonant formation study [7]

$$a + b \to c + d + e$$

To find out if there is any resonant intermediate state we first have to choose the resonance to look for; let's suppose that the resonant state, call it R, produces the particles c and d. Thus the reaction can be also written as follows

$$a + b \to R + e \to c + d + e$$

Then we have to choose a proper variable able to put the resonance in evidence. Since in the passage

$$R + e \to c + d + e$$

the particle e stays the same, we can suppose that the invariant mass of R would be the same of the system of the two particles c and d. So we can look for resonance on the events distribution of along a new non linear features that is the invariant mass of the particle R

$$M_R = \sqrt{(E_c + E_d)^2 - |\vec{p}_c + \vec{p}_d|^2}$$

If actually the resonance exists, the plot of the cross section should have the form shown in Figure 3.4 on the right side, where the smooth shape of the background distribution is broken by the presence of a peak.

Moreover, if we are searching a particular resonance of the particles c and d but other resonances have already been included in the theoretical model, we need to compare the expected resonance curve with the experimental one in order to establish if there are new configurations which can contribute to that peak, that means finding new physics. For this reason, analysts make use of a particular statistical tool named *Likelihood ratio test*. It compares different hypothesis by doing the ratio of the two relative Likelihood functions (Neumann Pearson lemma) and comparing the result with a reference value k_α:

$$\frac{L(\{x\}|H_1)}{L(\{x\}|H_0)} > k_\alpha$$

Each Likelihood function $L(\{x\}|H)$ represents the probability of an hypothetic theoretical model H, given a fixed experimental sample $\{x\}$ in the phase space. If the n events constituting the experimental sample are independent and identically distributed, the Likelihood function has the following form:

$$L(\{x\}|H(\theta)) = p(\{x\}|\theta) = Pois(n|\nu(\theta)) \prod_{e=1}^{n} p(x_e|\theta)$$

where e refers to a generic event and θ to the parameters of the hypothetical model. So making the ratio between the null hypothesis H_0 (the experimental data are distributed as the expected background) and another hypothesis H_1 which assumes a specific metastable configuration means comparing the two probabilities of fitting experimental evidence.
Because the high dimensionality of the phase space, often it is not possible to analytically compute the likelihood function so physicists resort to Monte Carlo simulations to represent the values the function takes on the phase space. Anyway, the phase space is so wide that it is impossible simulating enough events to fully cover it. This is why, actually, it is necessary reducing the phase space dimensionality. Significant part of Data Analysis is approximating the likelihood as best as possible.
Machine learning with deep neural networks overcomes this troubles, offering new tools to handle all raw features, without needing to reduce the phase space, as it provides signal and background distributions without going through Likelihood.

Chapter 4

Baldi, Sadowski and Whiteson

In order to experiment neural networks capabilities in support of physicists' work of analysis at LHC, we aim to reproduce Baldi, Sadowski and Whiteson's work of analysis with machine learning, pubblished on 5th June 2014with title *Searching for Exotic Particles in High-Energy Physics with Deep Learning* [1]. Our work will make use of the same dataset of simulated events used by them and, in order to tune in short time the net, we will also take as reference the hyper-parameter optimization already found by them. Thanks to their work we have built starting neural networks already well-working, so that we could procede studing them and discussing about deep neural networks advantages and limit in the scope of High Energy Physics.

4.1 The purpose: Shallow Network and Deep Network

4.1.1 Shallow a Deep Networks Complexity

Thanks to Horrnik, Stinchcombe and White's work *Multilayer Feedforward Networks are Universal Approximators*, we can state that

Theorem *Feed forward networks are capable of arbitrarily accurate approximation to any real-valued continuous function over a compact set.*

This theorem has been proved both for single hidden layer and multi hidden layer networks[2], establishing that these kind of networks are *universal approximators*. For single hidden layers networks (*Shallow Networks*) the mapping accuracy grows with the number of neurons in the layer, while for multi layers networks (*Deep Networks*) it grows principally increasing the number of hidden layers. More precisely, the complexity of a space S is often measured by the sum $B(S)$ of the *Betti numbers*[3]. Theoretical studies published in [14] demonstrates that

Proposition 1 *For network architectures with a single hidden layer, the sum of the Betti numbers, $B(S_\chi)$, grows at most polynomially with respect to the number of the hidden units h, i.e., $B(S_\chi) \in O(h^n)$, where n is the input dimension.*

Proposition 2 *For deep networks, $B(S_\chi)$ can grow exponentially in the number of the hidden units, i.e. $B(S_\chi) \in \Omega(2^h)$.*

This is why *Circuits complexity theory* tells us that raising the complexity of a neural network by adding more neurons per layer is more expensive than increasing the number of the layers keeping their size fixed. For

[1] [13]

[2] See [12] for the full treatment and demonstration

[3] It is possible to measure the complexity of a given function

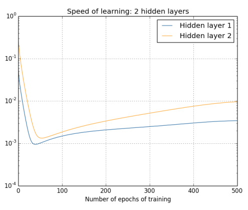
$$f_\chi : \mathbb{R}^n \to \mathbb{R}$$

by the topological complexity of the space

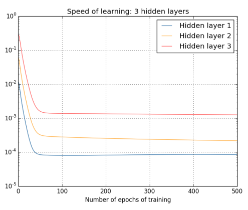
$$S_\chi = \{x \in \mathbb{R} | f_\chi(x) > 0\}$$

, that is, in the neural networks scope, the ensemble of the network inputs belonging to the positive class. The topological complexity of a n-dimensional space can be described through n numbers called the Betti numbers [14] this reason, it should be more efficient improving the performances of the net moving from *Shallow networks* to *Deep Networks*.

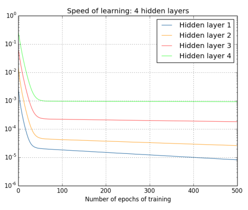
Anyway when we try training deep networks some impediments occurs first in tuning their hyper-parameters and then in speeding up the learning, affected by the so called *Vanishing gradient problem*.



(a) 2 hidden layers.



(b) 3 hidden layers.



(c) 4 hidden layers.

Figure 4.1: Example of Vanishing Gradient Problem. Figures show how the speed decreases moving towards the first hidden layers and how the latter decreases itself by adding hidden layers to the net.[2]

The former has not got a solution: the set of hyper-parameters is based on previous experiences and relies on rules of thumbs. But let's spend some few words on the latter. The vanishing gradient problem deals with the different speeds of learning of different hidden layers in a deep neural network.
It has been observed that neurons in the earlier layers learn much more slowly than neurons in later layers. In fact, adding new hidden layers to a deep architecture, the speed of the first hidden layer decreases extending the time needed by the deep network to complete the training. For this matter there is not a solution yet, but it is possible, for example, alleviating the problem of the increasing training time in the following ways:

- Increasing the training dataset size.
- Speeding up the computation with graphics processors.
- Using new learning algorithms such as Autoencoder pre-training and Dropout method to prevent over-fitting.

The main intent of Baldi, Sadowski and Whiteson was to demonstrate that, although deep networks are difficult to train for the reasons we have just mentioned, recent developments in machine learning made possible to overcome this difficulties, so that deep networks performances can equal those of shallow networks or even exceed them.
This is a great news, in particular for physicists: the common use of machine learning in particle physics up to now was giving as input to a shallow network few no linear variables, constructed cleverly by humans to reduce the kinetic phase space dimensions. In this approach, machine learning comes into play only in the last steps of analysis, requiring a lot of manual work by physicists about features construction. Positive results with deep neural networks would mean that statistical analysis can be carried on even if physicists do not know exactly which are the right high-level features: we could try only using deep networks to exploit the all raw kinetic features. But deep networks can help even when some high-level features has already been found. In fact, as we are going to show, considering both the raw features and the high ones leads to better performances in signal recognition, which means that deep network engine can extract more information by raw data than that extracted by human brain through high-level features.

4.2 The topic of the study: process involving new exotic Higgs bosons

Standard Model (SM) for particle physics describes Higgs Field mediator as a single particle h^0 with mass $m_{h^0} = 125\, GeV$. Signal processes we are searching for follow a possible extension of SM where no more only one Higgs particle is present, but five: h^0, H^0, H^+, H^- and A. This process in particular deals with the searching of new exotic Higgs bosons H^0 and $H^\pm$. They should be observed as a resonant state in two gluons fusion processes. The sequence of reactions is the following:

$$g\, g \to H^0 \to H^\pm\, W^\pm \to h^0\, W^\pm\, W^\pm \to b\, \bar{b}\, W^\pm\, W^\pm \quad (4.1)$$

As we can see from Figure 4.2, the process involving the resonant state we are interested in (*Signal process*) competes with another process having the same decay products. It can be written as

$$g\, g \to g \to t\, \bar{t} \to b\, W^+\, \bar{b}\, W^- \quad (4.2)$$

Since the latter has actually the same products of our signal, it can be used as irreducible background for our study. Since there is no way to distinguish it from signal observing their outputs, the only thing we can do is finding differences in their kinematic features and this is actually what we would try on neural network algorithms.

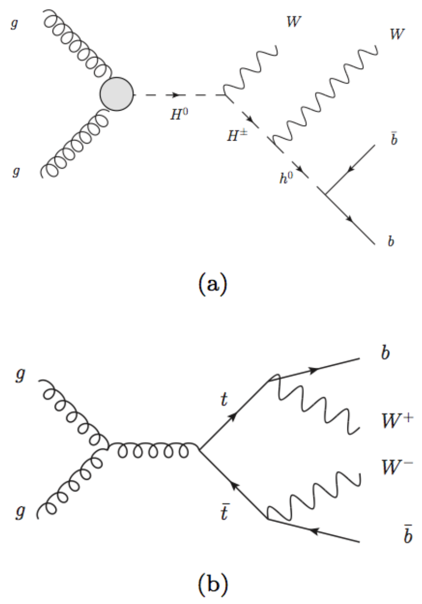


Figure 4.2: Diagrams for Higgs benchmark. (a) Diagram describing the signal process involving new exotic Higgs bosons H_0 and $H_\pm$. (b) Diagram describing the background process involving top-quarks (t). In both cases, the resulting particles are two W bosons and two b-quarks. [13]

4.3 Dataset

To train the net we used the same simulated events used in [13][4]. In both signal and background processes the couple of W bosons and that of b quarks decay too: W bosons can decay to a lepton and neutrino or to an up-type quark and a down-type quark. Thus what detectors see are their final products.
To simulate events for this work it has been chosen the *semi-leptonic decay mode*, that is one W bosons decaying to a lepton and a neutrino ($l\nu$) and the other W boson in a pair of *jets* (which correspond to a couple of up-type and down-type quarks). Thus the selected final products of 4.1 and 4.2 are: $l\nu\, b\, j\, j\, b$.
Events have to satisfied the following requirements for transverse momentum (p_T) and pseudorapidity (η):

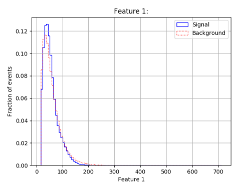
- Exactly one lepton, electron or muon, with $p_T > 20\, GeV$ and $|\eta| < 2.5$.
- At least four jets, each with $p_T > 20\, GeV$ and $|\eta| < 2.5$.
- b-tags on at least two of the jets, indicating that they are likely due to b-quarks rather than gluons or lighter quarks.

All these requirements are summed up by 21 low-level features:

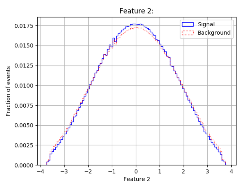
- 4 jets, each of them described through 4 variables: p_t, η, ϕ and the $b\, tag$.
- 1 lepton described through 3 variables: p_t, η, ϕ.
- 1 neutrino indirectly described by: missing energy magnitude and missing energy ϕ.

Their distributions are shown in Figure A.1[5]

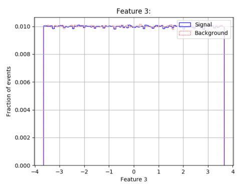
[4]Simulated events are generated with the MADGRAPH5 event generator assuming an energy of $\sqrt{s} = 8\, TeV$ for protons collisions as at the latest run of the Large Hadron Collider, with showering and hadronization performed by PYTHIA and detector response simulated by DELPHES. For the benchmark case here, $m_{H^0} = 425\, GeV$ and $m_{H^\pm} = 325\, GeV$ has been assumed.[13]
[5]see Appendix A



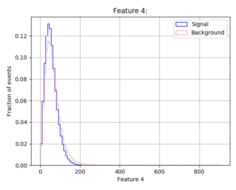
(a) Lepton p_T (GeV)



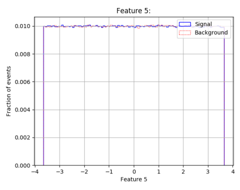
(b) Lepton η



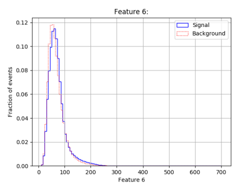
(c) Lepton ϕ (rad)



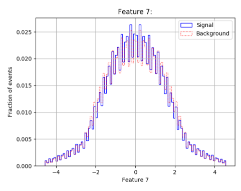
(d) Missing momentum p_T (GeV)



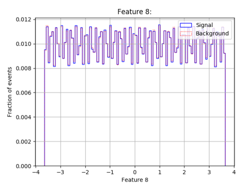
(e) Missing momentum ϕ (rad)



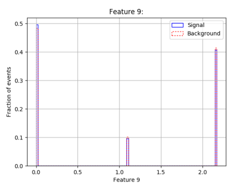
(f) Jet 1 p_T (GeV)



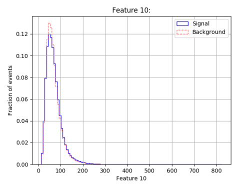
(g) Jet 1 η



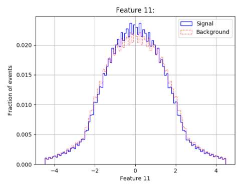
(h) Jet 1 ϕ (rad)



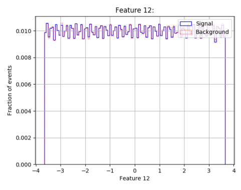
(i) Jet 1 $b-tag$



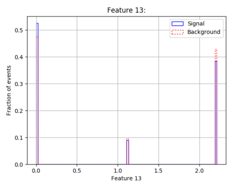
(j) Jet 2 p_T (GeV)



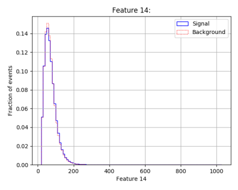
(k) Jet 2 η



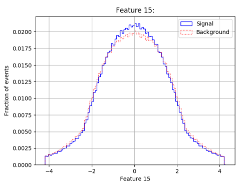
(l) Jet 2 ϕ (rad)



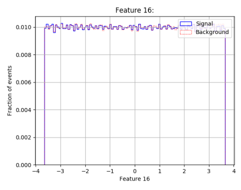
(m) Jet 2 $b-tag$



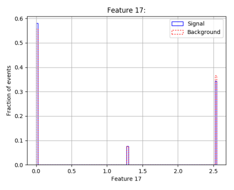
(n) Jet 3 p_T (GeV)



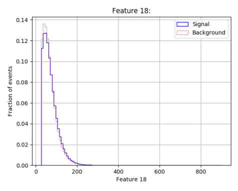
(o) Jet 3 η



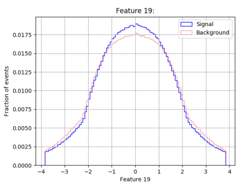
(p) Jet 3 ϕ (rad)



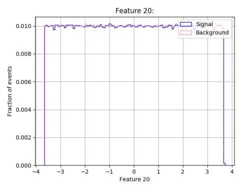
(q) Jet 3 $b-tag$



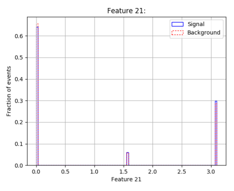
(r) Jet 4 p_T (GeV)



(s) Jet 4 η

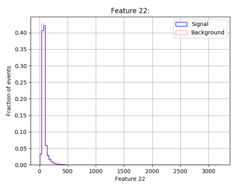


(t) Jet 4 ϕ (rad)

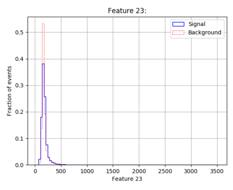


(u) Jet 4 $b-tag$

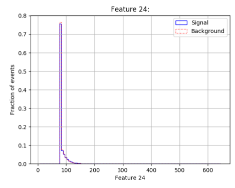
Figure 4.3: Low-level features.



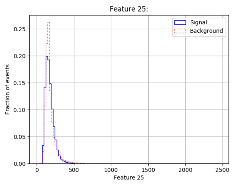
(a) $M_{jj}(GeV/c^2)$



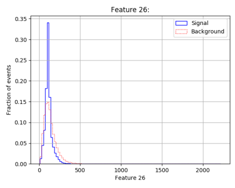
(b) $M_{jjj}(GeV/c^2)$



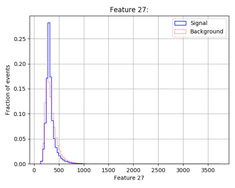
(c) $M_{l\nu}(GeV/c^2)$



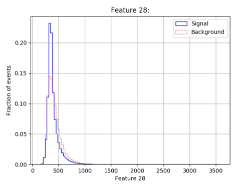
(d) $M_{jl\nu}(GeV/c^2)$



(e) $M_{bb}(GeV/c^2)$



(f) $M_{Wbb}(GeV/c^2)$



(g) $M_{WWbb}(GeV/c^2)$

Figure 4.4: High-level features.

For what concerns reducing phase space dimensionality, the theoretic hypothesis regarding signal and background processes allows to construct new features which better highlight the differences in the two processes. These features are the invariant masses of the metastable particles. In particular, for signal processes the following resonant decays have been theorized and the related invariant masses have been computed:

- $W \to l\nu$: the invariant mass $M_{l\nu}$ should show a peak at the known mass of the W boson M_W.
- $W \to jj$: the invariant mass M_{jj} should show a peak at the known mass of the W boson M_W.
- $h^0 \to b\bar{b}$: M_{bb} should show a peak at M_{h^0}.
- $H^\pm \to W^\pm h^0$: M_{Wbb} should show a peak at $M_{H^\pm}$.
- $H^0 \to WH^\pm$: M_{WWbb} should show a peak at M_{H^0}.

They represent 5 high-level features. On the other hand, for background processes the same W decays are expected, but another one can be taken in account:

- $W \to l\nu$: as for signal processes, the invariant mass $M_{l\nu}$ should show a peak at the known mass of the W boson M_W.
- $W \to jj$: as for signal processes, M_{jj} should show a peak at M_W.
- $t \to Wb$: both $M_{l\nu b}$ and M_{jjb} should show a peak at M_t.

Thus two more invariant masses, $M_{l\nu b}$ and M_{jjb}, must be computed. In total, we have 7 high-level features. Figure 4.4 shows their distributions.

Before being passed on to neural networks datasets have been standardized as follows:

- For those features which could assume negative values, they have assumed a normal Gaussian data distribution; all the values have been shifted by the mean value of feature distribution; then they have been renormalized dividing them by the deviation standard of the set. In this way the final set of data had mean 0 and standard deviation 1.
- For those features whose maximum value is greater than 1, they have assumed an exponential data distribution; in this case they only divided all the data by their mean value, as the final sets of data has got mean equal to 1.

Handling data in such way would facilitate calculus.

Baldi, Sadowski and Whiteson have published a 11 million data with all the 28 features included[6].

[6]The data are availa in the UCI Machine Learning Repository:
`archive.ics.uci.edu/ml/datasets/HIGGS`

4.4 Computational tools

Due to the huge number of data to process and the various degree of freedom of deep network models, the computational cost necessary to run the algorithm is not a marginal issue. Running the code, a single calculator must be able to keep all input data uploaded (for us, the file containing all the data is about 8 Gb) and, at the same time, to process subparts of them using stochastic gradient descent. During one epoch, all the initial and final parameters of the model has to be temporally stored together. This is why bigger the calculator's memory is the more complex model would be trained. To make the most of the computational power, this great quantity of stored information should be handled in such a way that the code would go through it nimbly, without making the computation heavy.

Python and new packages to handle big data PYTHON is a high-level programming language that makes our case. It allows the usage of several libraries written to meet the needs of almost all the scientific areas. In particular, more than one library have been written to make use of machine learning techniques and the developments of these utilities is still in progress. We decided to move from the code written by Peter Sadowski[7] to a new code using more recent machine learning packages. In particular we decided to use KERAS functionalities.

KERAS[8] is a high-level neural networks Application Programming Interface (API), written in PYTHON and capable of running on top of TENSORFLOW[9], CNTK[10], or THEANO[11]. KERAS library contains all the basic computational tool necessary to build and run a machine learning algorithm based on neural networks. It let to chose between different kind of models based on feedfarward, convolutional, recurrent layers and many others; the principal learning methods have already been implemented too. But even more useful is the possibility of personalizing the code by implementing new functionalities through backend utilities. This turned out to be helpful in handling plots and encoding analysis results .

Table 4.1: **Virtual machines properties**

RAM	16 GB
VCPUs	8 CPU
Disk	25 GB

Virtual machines According to what has been said so far, machine learning algorithms need calculators with a huge memory and all the facilities for the encoding, that is PYTHON and its machine learning libraries, such as PANDAS, NUMPY, KERAS and so on. For this reason, we run our code on a cluster of six virtual machines instantiated on the *Cloud area padovana* [12]. The six machines were used in a standalone mode, that means as single calculators[13]. Thus it was possible to try running six different versions of the algorithm at the same time, allowing to accumulate results quickly.

At the beginning, the code was run using THEANO backend for KERAS library, but it seemed not to be compatible with our virtual machines: they went out of memory and swapped till the process was killed; it happened when the number of events used to train the net overstepped a threshold about at 100 000 events. We tried to find an explanation through the lines of the code but there was nothing we could do to solve the problem. So we tried to run the code on another computer, with the same RAM of the virtual machines, and we found out that the problem did not occur anymore. So we supposed there was some unknown hitch on how the loaded libraries handled the code in TEANO backend on the virtual machines. We thus moved to TENSORFLOW and noticed that the code started running perfectly. Therefore, from that moment on, we have kept TENSORFLOW backend as default configuration.

4.5 Methods

I proceeded in the study of Deep Networks taking as reference the study in [13]. I set the network hyper-parameters as described in the papers and I trained the models using three different set of inputs: the 21 low-level, the 7 high-level features and all them together. I did several runs changing the size of the dataset, the number of hidden layers, the number of neurons per layer and the weights initialization. Other parameters of the method stayed the same for all the runs.

[7]The source is available at: `https://github.com/uci-igb/higgs-susy`
[8]Keras documentation at: `https://keras.io`
[9]TENSORFLOW documentation at: `https://www.tensorflow.org`
[10]CNKT documentation at: `https://docs.microsoft.com/en-us/cognitive-toolkit/`
[11]THEANO documentation at: `http://deeplearning.net/software/theano/`
[12]login at: `https://cloud-areapd.pd.infn.it/dashboard/auth/login/`
[13]A more efficient way of running codes on a cluster is by distributing part of one single algorithm to each component of the cluster. In this way the process would speed up. But it is not treated in this work; Chapter 7 will tell something more about this topic.

4.5.1 Validation and testing split

Starting form the full set of data I constructed three subset dividing the full set in ten parts and choosing one of them as validation sample, another one as testing sample and the sum of the remaining ones as training sample. Thus, when all the 11 million events were used, validation and testing sample were composed of 1.1 million events and training sample of 8.8 million events. The code is written such that 10 different choices of testing and validation split are available, thus the same algorithm could be executed repeatedly.

4.5.2 Models

We considered two kind of feedforward neural network: the *Shallow Network*, that is a single hidden layer network, and the *Deep Network*, a multi hidden layers network. The input layer could have 21, 7 or 28 neurons depending on which subset of features was used. The output layer has always been one single neuron with floating output between 0 and 1. Thus the model would be a binary classifier, that means a model able to collocate with a certain probability input events into two different label categories. The hidden layers have all the same number of neurons. In order to compare the results with that gained in [13], the final value chosen for the number of hidden layers was 3 and for the number of neuron per layers was 300.

4.5.3 Predetermined hyper-parameters set up

Those parameters of the learning algorithm which have stayed the same during all the run are described below.

Activation functions. For input and hidden layers a *tanh* activation function has been set, while for the output the *sigmoid* one.

Mini-batch size It has been set at 100.

Learning rate It has been scheduled with an initial value of $\eta_0 = 0.05$ and has been reduced by a factor $q = 1.0000002$ every batch computation till it reached the minimum value $\eta_{min} = 10^{-6}$ which stayed constant. In practice, the rule for learning rate update was

$$\eta' = \frac{\eta}{q}$$

Starting from the relation $\eta_1 = \frac{\eta_0}{q}$, knowing the batch number ($batch$) the rule can be rewritten as

$$\eta' = \begin{cases} \frac{\eta}{q^{batch}} & if \quad \eta > \eta_{min} \\ \eta & if \quad \eta = \eta_{min} \end{cases}$$

Momentum As for the learning rate, also momentum has not remained constant; its initial value has been set at $\mu_0 = 0.9$ and it has been increased linearly over the first 200 epochs till it reached the final value of $\mu_{max} = 0.99$ which stayed constant. The rule for update is then

$$\mu' = \begin{cases} \mu_0 + \frac{\Delta\mu}{\Delta e} e & if \quad \mu < \mu_{max} \\ \mu & if \quad \mu = \mu_{max} \end{cases}$$

where e stays for the actual epoch, $\Delta\mu = \mu_{max} - \mu_0$ and $\Delta e = 200$.

Number of epochs To prevent over fitting, an early stopping algorithm has been inserted [14]. It schedules to stop training when the accuracy computed over the validation set has not changed more than a factor 0.00001 in 20 epochs.

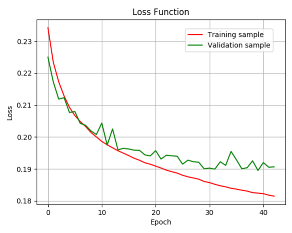
4.5.4 Experiment on new tools for machine learning

Autoencoder pretraining Implementing autoencoder pretraining is not difficult; as we described in subsection 2.3.3, autoecoder networks for pretraining are only two layers long. The obstacles appear when we tried to run the code with large datasets. The process went out of memory at the end of the second layer pretraining. The cause is that computational expense was too much for a single memory. For this reason we performed autoencoder tests using a restricted sample of 2 million events.

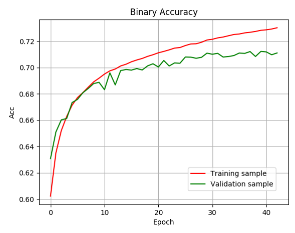
[14]This algorithm has been modified respect to the one described in [13]. The latter seemed not to be adeguate to the network, since it scheduled a minimum 200 epochs to be executed, while our networks went over fit in this way.

Dropout When we used dropout technique we apply it only on hidden layers with a standard drop rate of 0.5. We did our tests on deep network with 3 hidden layer and 300 units per layer varying the size of dataset. In particular we tested the net using as training data 200 000 events, 2 million events and then 8.8 million.

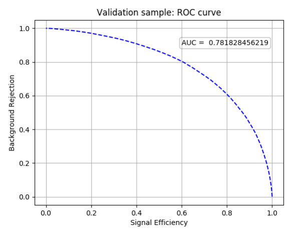
4.6 Analysis



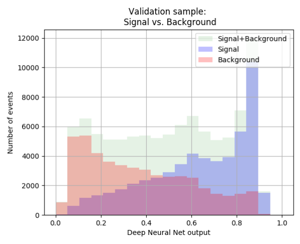
(a) Loss function over epochs.



(b) Accuracy over epochs.



(c) ROC curve for validation sample.



(d) Histogram of signal and background prediction distributions (not normalized).

Figure 4.5: Example of analysis results

As described in the paragraph above, several studies have been executed with deep and shallow networks . In order to estimate the stability and the eventual presence of over-fitting troubles, we plotted the trends of loss function and binary accuracy over the epochs, both for the training and the validation sample.

In addition we built an histogram of the predicted events over the possible outputs of the model; to this scope, the output range $[0, 1]$ has been divided into 20 bins. The net discriminates well if the predicted signal distribution presents a peak towards 1 and the predicted background distribution towards 0.

To compare the results, we used the most common parameter of evaluation in machine learning scope, that is the *Area Under the ROC Curve (AUC)*; as we will see later, it can be related to particle physics *significance*.
ROC is the acronym for *Receiver Operating Characteristic*, which is a graphical method to observe binary classifiers efficiency. In this case it has been constructed by plotting the signal efficiency versus the background rejection, varying the point of cut, that is the value between 0 and 1 over which the events are classified as signals by the net. Signal efficiency is the ratio between the number of true signal-tagged events (namely those signal events whose predicted output is over the cut value) and the total number of signal-tagged events; the second one is the ratio between the number of true background-tagged events (namely those background events whose predicted output is under the cut value) and the total number of background events. More AUC results near to the unity the best classification is. The cut value has been varied between 0 and 1 with a step of 0.01, thus the ROC curve has been constructed over 100 points. Some examples of the output of a training analysis is proposed in Figure 4.5.

4.6.1 Figure of Merit

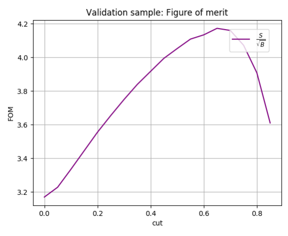


Figure 4.6: Example of FOM

One more quantity was computed to get the results closer to physical scopes. That is called *Figure Of Merit (FOM)*. Maximizing this quantity means maximizing signal and background discrimination, so it helps us to choose the more efficient cut point for signal classification. In other words, considering background distribution over the phase space expected by our theoretical model, a good choice of the cut point allows physicists to determine wether a difference in experimental distributions can be identified as new signal or not. We assumed as FOM the following quantity:

$$FOM = \frac{S}{\sqrt{B}}$$

where S stays for the total number of labeled events evaluated as signals by the net while B for those evaluated as background in function of the chosen cut point. The term $\sqrt{B}$ represents the error on the number of background events (B) assuming they follow a

Poisson distribution. More precisely, to state wether signal events (S) are actually a resonance or only statistical fluctuations, the effective error should take in account also the error on the number of signal events, that is $\sqrt{S}$, as signals follow poissonian distribution too. So the correct expression for the error should be

$$\sigma = \sqrt{B+S} \tag{4.3}$$

where the two contributes are added in quadrature. Since in our case $S \ll B$ we approximate 4.3 as follows

$$\sigma = \sqrt{B+S} \approx \sqrt{B}$$

Data used for running this work were simulated as described in subsection 4.3 and they were built such that signal and background would have the same probability; thus, for a large dataset we expected about 50% events labeled as signals and equally 50% as backgrounds. The consequence is that, when discrimination is good enough, the number of signal predicted events should be quite the same of background predicted. In data collected by experiments a signal of new physics is usually a rare phenomenon which competes with lots of standard processes included in background. So the real cross section of signal events may be so small that it may be confused with background. FOM computation allows to find the optimal cut point to put in evidence the presence of signals.

Thanks to all these observables, several studies have been done: it has been studied the evolution of a fixed deep networks changing the size of the training inputs; the response of a deep network to the variation of the number of hidden layers; that of a shallow network to the variation of the number of neurons in the single hidden layer; the differences caused by different choices in weights and biases initialization and those caused by the introduction of dropout.

Varying data size For this study I used two different models: a shallow network built as described above, with 1 hidden layer and 300 neurons per each of them; a deep network with the same set but 3 hidden layers. As dataset I used that of low-level features. The total number of events (i. e. the sum of training, validation and testing data) used for the tests are: 5000, 10 000, 50 000, 100 000, 500 000, 1 000 000, 5 000 000, 10 000 000.

Varying the number of hidden layers For this study I used a deep network with a number of hidden layers that varies between 1 and 6; each hidden layer has got 300 units. The training sample was composed of 8.8 million events, 1.1 million for the validation and the testing samples.

Varying the number of neurons per layer For this study I used a shallow network with a only one hidden layer. The number of units in the hidden layer used for the tests are: 7, 100, 300, 1000, 2000, 10 000, 20 000. The training sample was composed of 8.8 million events, 1.1 million for the validation and the testing samples.

Changing weights initialization Starting from a deep network with 3 hidden layers and 300 units per layer, I trained the net first with a random normal initialization, then with a uniform and finally using autoencoders pretraining.

Random normal initialization schedules parameters sampled from a gaussian distribution with mean 0 and standard deviation of 0.1 in the input neurons, 0.001 in the output single neuron and of 0.05 in the hidden neurons.

Uniform initialization schedules parameters sampled from a constant distribution probability whose value $\frac{1}{n}$ depends on the number of units per layer n.

Autoencoder pretrainings have been implemented as well described in subsection 2.3.3. All the learning hyper-paramters set for the fine-tuning have been used also to set autoencoder networks.

Part III

Conclusions

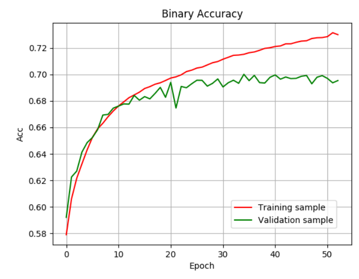
Chapter 5

Results

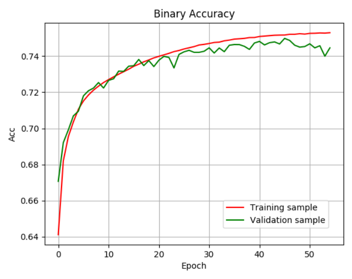
5.1 Performances optimization

5.1.1 Data set size

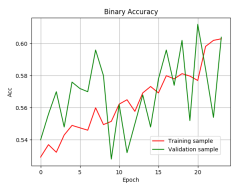
The set of figures reported in 5.1 shows how the accuracy of the learning process changes varying the number of example passed to the deep network to do experience[1].



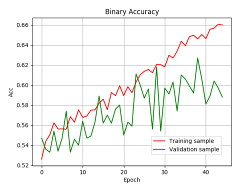
(a) 400 000 training events.



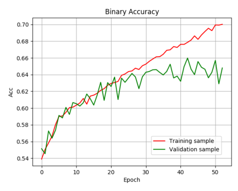
(b) 4 000 000 training events.



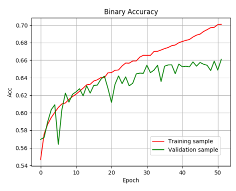
(c) 4000 training events.



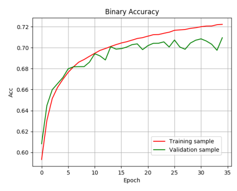
(d) 10 000 training events.



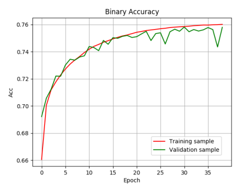
(e) 40 000 training events.



(f) 80 000 training events.



(g) 800 000 training events.



(h) 8 000 000 training events.

Figure 5.1: textbfAccuracy Comparison. Study of over-fitting varying dataset size.

The red lines mark the trend of the accuracy on training samples while the green ones on the validation ones. Points on the green lines oscillate more than those of the red ones, in particular for smaller datasets in 5.1c, 5.1d and 5.1e. This happens because the model is updated in order to fit the training sample and not the validation one, whose role is only to check for stopping training. When the dataset are small the model does not generalize well, or rather overfits the training data.
The model therefore adapts itself to the events it has experienced.
Increasing the number of events (see in the order the plots 5.1f, 5.1a, 5.1g, 5.1b and 5.1h), a saturation curve becomes evident both for training and validation data; this shape lets us guess the presence of an asymptotic maximum for the accuracy reachable by the algorithm.

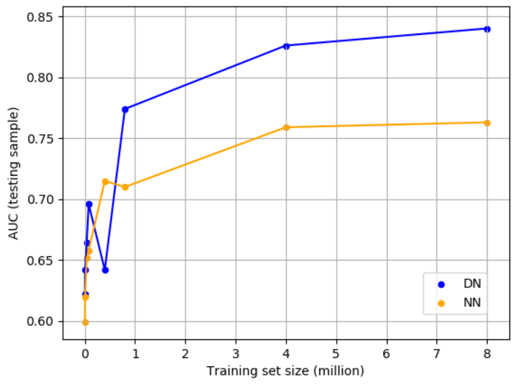
[1]The same plots for the shallow network are not shown here, since there are not relevant variations on the trend respect to that of the deep network. The only interesting aspect is that oscillations on validation accuracy stay evident also for large dataset. This because here we used the low-level features to train the algorithm and the shallow network has more difficulty to generalize starting from raw features than the deep one.

Training set size	AUC NN	AUC DN
4 000	0.599	0.622
8 000	0.620	0.642
40 000	0.652	0.664
80 000	0.658	0.696
400 000	0.715	0.642
800 000	0.710	0.774
4 000 000	0.759	0.826
8 000 000	0.763	0.840

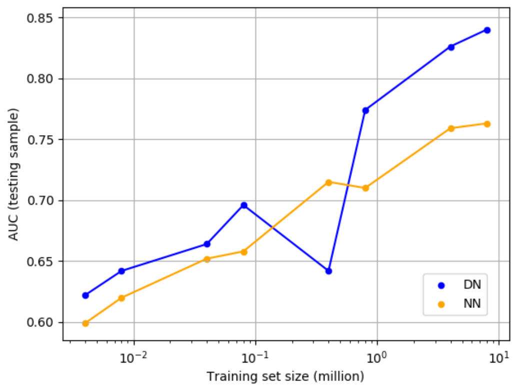
Table 5.1: **Study of dataset size.** Comparison of deep networks with different number of training events in term of the Area Under the ROC Curve (AUC). The deep networks processing 21 raw features have 3 hidden layers and 300 units in each of them. Autoencoder pre-training and dropout have not been used.

As figures 5.1a and 5.1b highlight well the saturation value for training and validation accuracies is not the same, but they tend to near as the samples expand. The gap too is a consequence of the over fitting, and the fact that it gets smaller adding events to the samples proves that increasing the size of the datasets helps to solve the problem of overfitting.

Another logical consequence of expanding the training sample is that the performance improves. This is obvious since if we pass on to the neurons more information they do learn more. Less obvious is how quick the performance improves varying the size of training dataset, that means asking what is its derivative respect to the sample size.



(a) **AUC vs. number of events**



(b) **AUC vs. number of events (log scale)**

Figure 5.2: AUC performances vs. data size.

In figure 5.2a is delineated the variation of AUC with the data size, both for the deep network and the shallow one. A part from one point it is evident that, using low-level features, the deep network performs better than shallow one regardless of the size of the sample used to train them.
Since we have only one single run for each test, we are not able to associate a statistical error to the AUC results; an error could justify the switched behavior at 400 000 events that is almost surely a fluctuation.
Anyway we know that the error would decrease increasing dataset size so we can be more confident in last results. As the accuracy in the previous plots, also the AUC suggests the presence of a saturation point. Shallow network perform seems to near that point slower than the deep one.
In fact, if we consider the logarithmic scale version of AUC vs. sample size (5.2b) and try to do a linear interpolation we find the following results

Deep network:

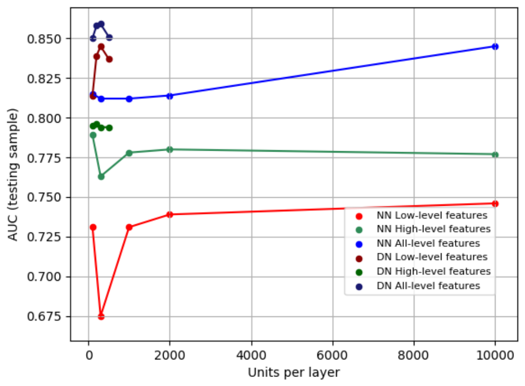
$$a_1 = 0.069 \pm 0.003 \ \ (4.9\%)$$
$$a_0 = 0.772 \pm 0.005 \ \ (0.6\%)$$

Shallow network:

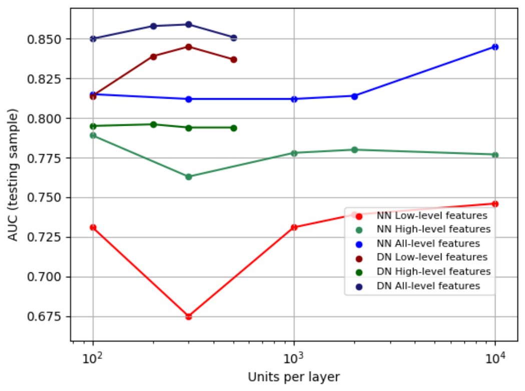
$$a_1 = 0.050 \pm 0.002 \ \ (3.874\%)$$
$$a_0 = 0.715 \pm 0.003 \ \ (0.377\%)$$

where a_0 and a_1 represent respectively the 0-degree and the 1-degree coefficients of the linear polynomial function. It is clear that the logarithmic speed of shallow network is lower than those of deep network, that means the first one is nearer to its saturation point than the second one is, which implies that if we suppose to add even more events, the deep network would still learn something new while the shallow network actually much less. This result is the first proof that deep networks have great potentiality.

5.1.2 Layer size



(a) **AUC vs. number of units per layer**



(b) **AUC vs. number of units per layer(log scale)**

Figure 5.3: AUC performances vs. number of units per layer.

In this paragraph we will show the results of tests over different number of hidden layers for both the shallow network (SN) and the deep network (DN). These examples are useful to compare the raising of complexity due to the addition of neurons per layer, with that of deep network due to the addition of hidden layers.
The latter would be discussed in the following paragraph.
As we can see from Figure 5.3, the trend for shallow network seems staying constant, in particular when trained with high-level features; more variations are visible in performances with low-level features. On the contrary, for deep network the variation is clearly evident in all three cases. Anyway, also in this case, the one which varies more widely is that trained with low-level features. This is a first evidence of how complexity of input data influences the learning algorithm. Moreover, if we focus on DN-low level and DN-all features trends we can see a smooth peak corresponding to the model with 300 units per layer, which is the final configuration chosen by Baldi, Sadowski and Whiteson.

Table 5.2 reports the results for shallow networks with 10, 100, 300, 1000, 2000 and 10 000 units per layer.
The same tests was executed by Baldi, Sadowski and Whiteson, thus we can easily compare their results to ours. For this topic, go to subsection 5.5.

5.1.3 Number of hidden layers

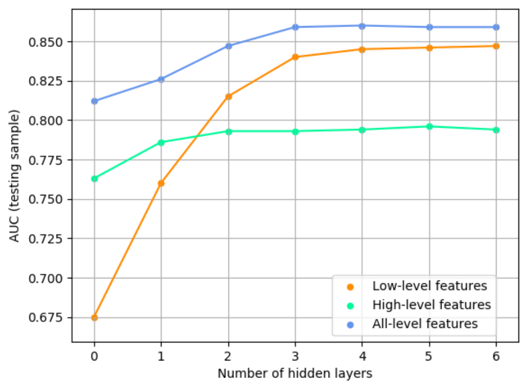


Figure 5.4: **Performances trends.** Figure showing the different the speeds using low, high or all features for training the deep networks. Values used for the plot are reported in Table 5.3

As we mentioned before, in this paragraph it is shown a study of deep network performances raising up its complexity by adding more hidden layers to the net. First we can suppose that raising deep model complexity would bring to more evident improvements than raising the shallow one. Anyway we have to consider the effects of the vanishing gradient problem (see 4.1), which put a limit to AUC improvements.
Upper limit is evident from Figure 5.4 and it is clear that it differs for each input set of features: more features the higher threshold is.

Table 5.3 reports AUC results for this set of tests.

Table 5.2: **Study of deep architecture: layer size.** Comparison of deep networks with 3 hidden layers of different sizes in term of the Area Under the ROC Curve (AUC). Autoencoder pre-training and dropout have not been used.

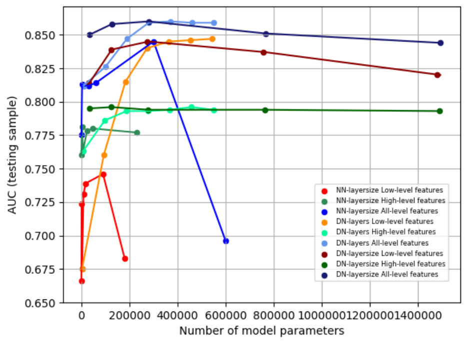
			AUC	
Technique	**Neurons per layer**	**Low-level**	**High-level**	**Complete**
NN	100	0.731	0.780	0.815
	300	0.675	0.763	0.812
	1000	0.731	0.778	0.812
	2000	0.739	0.780	0.814
	10 000	0.746	0.777	0.845
DN	100	0.814	0.795	0.850
	200	0.839	0.796	0.858
	300	0.840	0.793	0.859
	500	0.837	0.794	0.851
	700	0.820	0.793	0.844

Table 5.3: **Study of deep architecture: number of hidden layers.** Comparison of deep networks with different number of hidden layers in term of the Area Under the ROC Curve (AUC). Hidden layers have 300 units in each of them. Autoencoder pretraining and dropout have not been used.

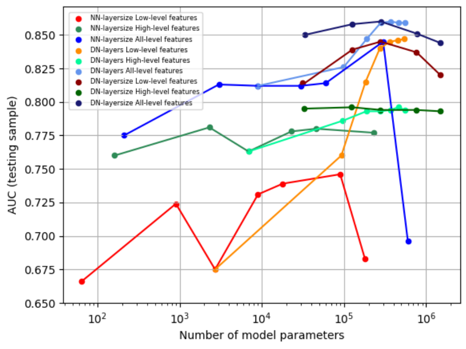
		AUC	
Number of hidden layers	**Low-level**	**High-level**	**Complete**
NN 0 hidden layer	0.675	0.763	0.812
DN 1 hidden layer	0.760	0.786	0.826
DN 2 hidden layers	0.815	0.793	0.847
DN 3 hidden layers	0.840	0.793	0.859
DN 4 hidden layers	0.845	0.794	0.860
DN 5 hidden layers	0.846	0.796	0.858
DN 6 hidden layers	0.847	0.794	0.859

5.2 Model error

To quantify the goodness of the fit we have to take in account two kind of error: the systematic error (*bias*) and the sensitivity of the prediction (*variance*). The first one measures how much the model predictions systematically deviate from their true values regardless the particular dataset used; the second one measures the error depending on the particular set of randomly chosen training data.
The size of the error depends principally on model complexity: by increasing the number of parameters, more data should be captured by the model, thus the *bias* should decrease. On the other hand, since the model would move easily towards the specific training data, the risk of over-fitting raises up and the *variance* too. This is why it is necessary to choose the model in order to find a compromise between the two trends.



(a) **AUC vs. model complexity**



(b) **AUC vs. model complexity (log scale)**

Figure 5.5: **Performances trends.** Figure showing the different the speeds using low, high or all features for networks wih different number of model parameters. Values used for the plot are reported in Table 5.4

Model complexity: *bias*. Figure 5.5 merges results obtained testing shallow and deep network layers size and deep network number of hidden layers, plotting AUC versus total number of model parameters (data used for the plots are reported in Table 5.4). Plots allow us to experimentally observe which model maximizes the performance and at the same time which is the simplest way to reach it.

I decided to plot testing AUC versus number of trainable parameters since the total number of parameters is actually a measure of model complexity which is common to every architecture, while AUC gap from the unity, that is its mathematical expectation, gives the idea of model bias; so a gain in AUC is equivalent to reducing bias. Let's take a look at Figure 5.5a.

First thing to be said is that the choice of input features has a great influence on how the AUC can evolve. Training with raw features (red lines) lets the model to variate quickly respect to training with high-level ones (green lines). This means that information contained in high-level features can be extracted easier even by the simplest model that is the shallow network, but the fact remains that that information is not complete thus performances achieved with high-level features do not equal that with low-level.
On the other hand, low-level features contain all the kinematic information collected by the experiments but it is not easy at all to model it, thus also a little improvement in learning algorithm would bring to the raising of AUC result. Therefore, to more information corresponds an higher upper limit.
Following this argument, the best benefit of all is obviously gained by the usage of all 28 features as inputs (blue lines). In this way, high-level features suggest the net about which route to take and the low-level add more details to refine the results. In few words, the more input units the better result is achieved.

The second aspect to be considered is how increasing the model complexity in the most efficient way. Again in Figure 5.5a we can note that the faster improvement of AUC result is gained by varying the number of hidden layers. For instance, by adding one more layer to the a shallow network with 300 units we can reach an higher value for AUC than what we would obtain by increasing the number of units from 300 to 500.

Table 5.4 reports AUC results related to number of total trainable parameters, namely weights and biases, for all architectures tested for this study.

Table 5.4: **Study of model complexity.** Comparison of networks with different number of trainable parameters in term of the Area Under the ROC Curve (AUC) computed on the testing sample. Autoencoder pretraining and dropout have not been used. For more infomation about networks architecture see 5.1.2 and 5.1.3

Network architecture	**Model parameters**	**AUC Low-level**	**Model parameters**	**AUC High-level**	**Model parameters**	**AUC Complete**
NN 7 units	64	0.666	162	0.760	211	0.775
NN 100 units	901	0.724	2301	0.781	3001	0.813
NN 300 units	2701	0.675	6901	0.763	9001	0.812
NN 1000 units	9001	0.731	23001	0.778	30001	0.812
NN 2000 units	18001	0.739	46001	0.780	60001	0.814
NN 10 000 units	90001	0.746	230001	0.777	300001	0.845
NN 20 000 units	180001	0.683	460001	nan	600001	0.696
300 neuron per layer						
DN 1 hidden layer	93001	0.760	97201	0.786	99301	0.826
DN 2 hidden layers	183301	0.815	187501	0.793	189601	0.847
DN 3 hidden layers	273601	0.840	277801	0.793	279901	0.859
DN 4 hidden layers	363901	0.845	368101	0.794	370201	0.860
DN 5 hidden layers	454201	0.846	458401	0.796	460501	0.858
DN 6 hidden layers	544501	0.847	548701	0.794	550801	0.859
DN 3 hidden layers						
100 units per layer	31201	0.814	32601	0.795	33301	0.850
200 units per layer	125201	0.839	122401	0.796	126601	0.858
300 units per layer	273601	0.845	277801	0.794	279901	0.860
500 units per layer	756001	0.837	763001	0.794	766501	0.851
700 units per layer	1478401	0.820	1488201	0.793	1493101	0.844

Model stability: *variance*. Let's now turn to the second component of the error which is the *variance*. Here is presented the study of model variance for a deep neural network processing each of the three set of features and using the 8.8 million events as training examples.

I decided to measure the variance in term of the AUC variance, computed by repeating the train 7 times and evaluating AUC over the testing sample. As for the bias, this approach is justified by the fact that AUC is strictly related to the accuracy of predictions so, in its own way, this quantity shows fluctuations on the result due to the particular set of examples used to compute it. In table 5.5 is given a final estimation for the 3-hidden model ability of prediction. As we can see the variance is very small, thus the AUC relative error for low-level is only about 0.24%, about 0.10% for high-level, and %0.23 for all features; it means this algorithm has got a great stability and this occurs especially thanks to the large size of training sample.
We can notice a difference in error entity depending on wether raw data are included in the sample or not: the presence of raw data in the inputs causes a rise of relative error. This is another evidence that generalizing from raw data is not trivial.

Table 5.5: **Study of model variance.** Estimation of deep network variance in term of that affecting the Area Under the ROC Curve (AUC) of testing sample. The deep networks have 3 hidden layers and 300 units in each of them. Autoencoder pre-training and dropout have not been used.

DN 3 hidden layers (11 million ev.) **Test nr.**	**AUC** Low-level	High-level	Complete
1	0.846	0.796	0.859
2	0.840	0.795	0.855
3	0.841	0.794	0.856
4	0.844	0.795	0.859
5	0.841	0.796	0.856
6	0.843	0.794	0.856
7	0.844	0.795	0.853
Variance	0.000005	0.0000006	0.000004
σ	0.002	0.0008	0.002
Mean	0.8427±0.0008	0.7950±0.0003	0.8563±0.0008

5.3 Tests on regularization techniques

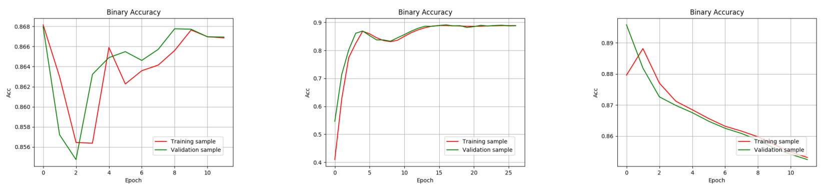
5.3.1 Autoencoder pre-training and weight initialization

We have already talked about autoencoder pretraining in Subsection 2.3.3. Here we aim to add something more about the expected effects of this algorithm on the training, according to [16].

Technique	Init.	AUC Low-level	AUC High-level	AUC Complete
11 million NN	uniform	0.675	0.779	0.803
	normal	0.675	0.763	0.810
11 million DN 3 hidden layers	uniform	0.833	0.793	0.858
	normal	0.840	0.793	0.859
2.5 million DN 3 hidden layers	uniform	0.812	0.793	0.841
	normal	0.813	0.789	0.837
	autoenc	0.803	0.792	0.836

Table 5.6: **Study of weight initialization.** Comparison between different methods of model parameters initialization in term of the Area Under the ROC Curve (AUC). Hidden layers have 300 units in each of them. Autoencoder pre-training and dropout have not been used.

Experiments shown in [16] support the vision of autoencoder pretraining as an unusual form of regularization. In virtually, it has been observed that loss function is an highly unconvex function of the parameters with many distinct local minima. This make its optimization with stochastic gradient descent a challenge for analysis, especially in a regime with large amounts of data, since in this way networks could be influenced more by early examples, determining which local minimum to approach.



(a) Low-level features (b) High-level features (c) All-level features

Figure 5.6: **Pretraining accuracy for first layer**. Study of autoencoder pretraining of first layer for a Deep Network of a 3 hidden layers and 300 units per layer. Both input layer and the 3 hidden ones have been pretrained starting from a uniform initialization and for a maximum of 100 epochs. Dropout have not been used.

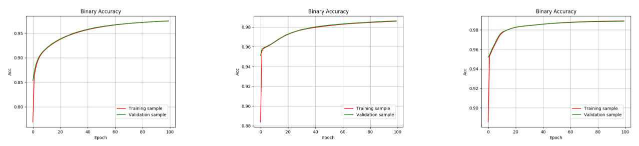
The pre-training procedure restricts the parameters into a small volume of the parameter space, called *local basin of attraction* and increases locally the cost function complexity adding more topological features such as peaks, troughs and plateaus. Thus it gives a good initial point for the fine-tuning process and further it renders locally more difficult to travel significant distances via a gradient descent procedure, putting a constrain over the parameters evolution. The latter is the reason why autoencoder pretraining can be seen as a regularizer: although it does not directly modify the cost function as L_1 or L_2 algorithms do, it gives a *Prior* on the dataset $(P(X))$ which determines the consequent probabilities of the final model configurations $(P(Y|X)^2)$. Briefly, learning $P(X)$ is helpful in learning $P(Y|X)$. What more surprisingly was observed in [16] is that

"Unsupervised pre-training, as a regularizer that only influences the starting point of supervised training, has an effect that, contrary to classical regularizers, does not disappear with more data (at least as far as we can see from our results)."

That sounds good since it means that autoencoder pretraining would always bring to an improvement of the performances.

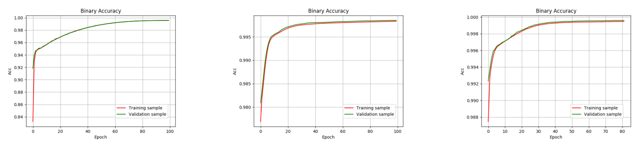
Now let's see if it happens in our experiment too. Using a set of 2 million training examples, the introduction of unsupervised pretraining do not help so much. As reported in Table 5.6, random normal and uniform initialization perform quite always the same or even better than pretraining initialization.

[2] where $P(Y|X)$ denotes the posterior probability of Y given X, X the inputs dataset and Y the output computed by the net.



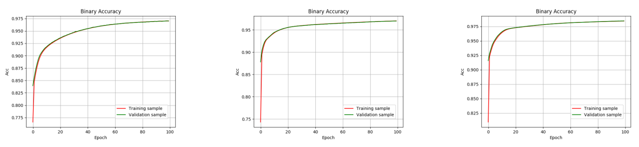
(a) Pretrain layer 2 (b) Pretrain layer 3 (c) Pretrain layer 4

Figure 5.7: **Pretraining accuracy using low-level features**. Study of autoencoder pretraining for a Deep Network of a 3 hidden layers and 300 units per layer. Both input layer and the 3 hidden ones have been pretrained starting from a uniform initialization and for a maximum of 100 epochs. Dropout have not been used.



(a) Pretrain layer 2 (b) Pretrain layer 3 (c) Pretrain layer 4

Figure 5.8: **Pretraining accuracy using high-level features**. Study of autoencoder pretraining for a Deep Network of a 3 hidden layers and 300 units per layer. Both input layer and the 3 hidden ones have been pretrained starting from a uniform initialization and for a maximum of 100 epochs. Dropout have not been used.



(a) Pretrain layer 2 (b) Pretrain layer 3 (c) Pretrain layer 4

Figure 5.9: **Pretraining accuracy using all features**. Study of autoencoder pretraining for a Deep Network of a 3 hidden layers and 300 units per layer. Both input layer and the 3 hidden ones have been pretrained starting from a uniform initialization and for a maximum of 100 epochs. Dropout have not been used.

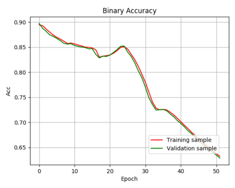
This result seems disappoint our expectations. Nevertheless, it has its relevance: thanks to this study we understand that unsupervised pretraining is a complex matter on which it is not easy drive general conclusions at all. In this case, we can not completely agree with [16] for the following reasons. First of all, we are in front of a completely different kind of input data and inputs ensemble characteristic strictly particularizes the solving strategy for learning optimization. Thus we can not completely rely on results obtained for tests carried on MNIST[3] data to infer property of autoencoder applied to binary classification with kinetic features as inputs. Secondly, to be useful autoencoder networks should learn well and optimizing them networks is not trivial.
Let's give a look to our results. Small differences can be seen inside our tests between training with high-level features and low-level features. As Table 5.6 shows, autoencoders cause some improvements only on high-level processed algorithms.

For a possible explanation we have to consider basically three aspects. First of all, autoencoder pretraining role is to prevent overfitting; if the train is not affected by over fitting then preventing it might be not necessary. In our case, we used a set of 2 million events to training the net; this size should be large enough for overfitting not to be significant (to see accuracy magnitude varying training data size, refer to plots in 5.1.1).

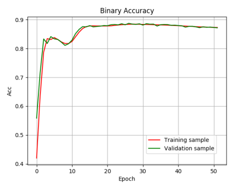
Secondly, autoencoder helps finding an optimal starting configuration for model parameters and this is useful when we do not have idea how to chose a good initialization. In our algorithm, we used a basic uniform initialization and the normal one, described in [13]. They already do a good work without pretraining.

Finally, but most important, the major difficulty resides in first layer pretraining. It is the core of autoencoder pretraining since it abstract the first trait of input data, which is the most salient. In our tests autoencoder network with one hidden layer is not able to generalize well, in particular when raw features are included in inputs. This fact is put in evidence by 5.6a, 5.6b and in 5.6c, where learning process for first layer pretraining is outlined.
On the contrary, pretrain successive layers is not a problem for this kind of architecture. Anyway the first one influences all the rest of the training since next layers pretraining derive from it, using inputs derived from the first layers pretraining. In our tests first layer is better pretrained when high-level features are used and this is why then also the fine tuning performs better and we can see a little improvement. Figures 5.7, 5.8 and 5.9 show the accuracy trend for validation sample during pretraining of hidden layers.

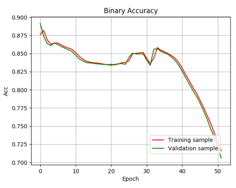
[3]database of handwritten digits available at: `http://yann.lecun.com/exdb/mnist/`



(a) Low-level features

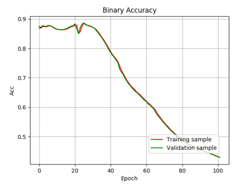


(b) High-level features

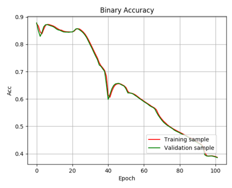


(c) All features

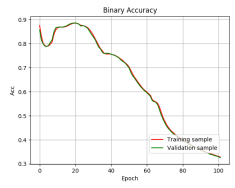
Figure 5.10: **Pretraining accuracy of first layer**. Study of autoencoder pretraining of first layer for a Deep Network of a 3 hidden layers and 300 units per layer. Both input layer and the 3 hidden ones have been pretrained starting from a uniform initialization and for a maximum of 100 epochs. Dropout has not been used.



(a) Learning rate $\eta = 0.05$



(b) Learning rate $\eta = 0.005$



(c) Learning rate $\eta = 0.0005$

Figure 5.11: **Pretraining accuracy of first layer varying the learning rate**. Study of autoencoder pretraining of first layer for raw inputs. First layer has been pretrained starting from a uniform initialization and for a minimum of 100 epochs. Dropout has not been used.

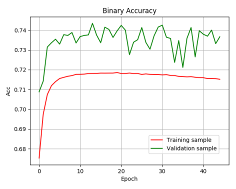
Stated that we need to optimize first layer pretraining, I tried to put a constrain over the minimum number of training epochs, asking the learning process not to stop before having performed at least 50 epochs. Figure 5.10 shows the results for accuracy trend in first layer pretraining: the request did not help and the same even imposing 20 minimum epochs. There are still problems with low-level features generalization. Table 5.7 shows results for the three tests, proving there is not a relevant effect in changing the number of epochs. Nevertheless, accuracy trends in 5.10 show some kind of smooth peaks that let us guess the presence of local minima for the loss function. The algorithm goes through them leaving possible points of convergence. To have a complete outlook on the accuracy shape evolution, I tried to prolong once again the time of learning moving the minimum epochs to 100 and change the value of learning rate to check for those possible local minima which 0.05 learning rate resolution does not distinguish. Lowering the value of learning rate acts as zooming out on accuracy trend in the first epochs, since we are slowing down learning speed. Figure 5.11 shows our results: even using a learning rate two orders of magnitude smaller (5.11c) than the initial one (5.11a), the net is not able to perceive other local minima of loss function.

DN 3 hidden layers (2.5 million ev.) **Autoencoder**	**AUC**		
Test nr.	**Low-level**	**High-level**	**Complete**
10 epochs	0.803	0.7925	0.8360
50 epochs	0.809	0.7930	0.8398
50 epochs	0.806	0.7925	0.8361
20 epochs	0.810	0.7903	0.8371
Variance	0.000007	0.000001	0.000002
σ	0.003	0.001	0.002
Mean	0.807±0.001	0.7919±0.0006	0.8372±0.0008

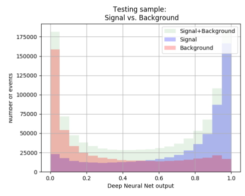
Table 5.7: **Model variance with autoencoder pretraining**

Considering the kind of data we are working with, the conclusion is that using a random initialization is more convenient than applying autoencoder pretraining.
Maybe a more thorough searching for autoencoder optimization would have brought to more successful results, but this goes beyond the scope of this work.

5.3.2 Reducing over-fitting: Dropout

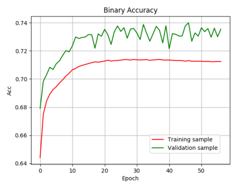


(a) Accuracy vs. epochs

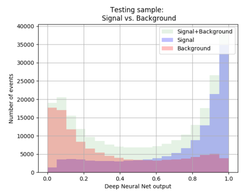


(b) Distribution of signals and background events (testing sample)

Figure 5.12: **Study of dropout techinque of regularization: 11 millions evets**. Application of dropout technique with rate of 0.5 to hidden layers of a Deep Network with 3 hidden layers, each of them with 300 units. All features have been used to train the net. The full dataset has been used, so 8.8 million events as training data and the rest equally divided between validation and testing data. Autoencoder has not been used.

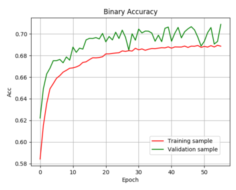


(a) Accuracy vs. epochs

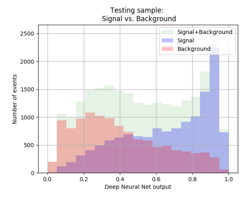


(b) Distribution of signals and background events (testing sample)

Figure 5.13: **Study of dropout techinque of regularization: 2.5 million events**. Application of dropout technique with rate of 0.5 to hidden layers of a Deep Network with 3 hidden layers, each of them with 300 units. All features have been used to train the net. Autoencoder has not been used.



(a) Accuracy vs. epochs



(b) Distribution of signals and background events (testing sample)

Figure 5.14: **Study of dropout technique of regularization: 250 000 events**. Application of dropout technique with rate of 0.5 to hidden layers of a Deep Network with 3 hidden layers, each of them with 300 units. All features have been used to train the net. Autoencoder has not been used.

Let's turn to Dropout. We applied dropout technique to deep networks with 3 hidden layers. Figure 5.12a shows accuracy trends using 8.8 million training events. There is a great difference respect to those shown in Figure 5.1h in subsection 5.1.1: here we find the green line, which represents the accuracy on validation set, above the red one, representing that on training set. This happens thanks to the action of Dropout as regularizer, that is randomly changing every epoch the 50% units per layer which will take part to the training. It is evident, however, that validation accuracy fluctuates around a value which stays quite constant during the epochs. In fact, training with all features and full dataset the value for AUC results 0.814, quite less than 0.859 performed without any regularization. So Dropout technique results useless when large datasets are processed.

If we analyze more in detail how dropout effects change varying training data size, we find that, using low-level features, accuracy oscillaions are more evident for large datasets than for small ones. That is exactly the opposite of what we observed in subsection 5.1.1, where the curve of validation accuracy becomes more stable increasing data size.

Moreover, with the help of dropout technique high-level performances surpass low-level ones; this probably means that also dropout technique validity depends on input features.

Datasize	**AUC**		
	Low-level	**High-level**	**Complete**
8.8 million	0.741	0.781	0.814
4 million	0.733	0.781	0.819
2 million	0.715	0.777	0.811
0.2 million	0.713	0.758	0.782

Table 5.8: **Dropout technique varying data size.** Test on deep network with 3 hidden layers and 300 units per layer.

Test nr.	**AUC** Low-level	**High-level**	**Complete**
1	0.719583968828	0.777	0.811
2	0.715264812438	0.774	0.811
3	0.714817433643	0.768	0.799
Variance	0.000005	0.00001	0.00003
σ	0.002	0.003	0.005
Mean	0.717±0.001	0.773±0.002	0.807±0.003

Table 5.9: **Model variance with dropout technique of regularization** Test on deep network with 3 hidden layers and 300 units per layer using 2 million training events.

In conclusion, we show in Table 5.10 the best results obtained by our tests. In summary, we found that both dropout technique and autoencoders pretraining should be applied to small dataset, otherwise they are useless. In addition, even with few examples, we noticed that autoencoder techniques strictly depends on the complexity of input data, since tuning the autoencoder algorithm can be as difficult as tuning the deep network they should regularize. Also dropout effects changes with the particular kind of input sample.
So, if possible, it is preferable to increase the number of events to train the net or, in alternative, adding to raw inputs one or more high-level features to route the algorithm towards convergence, as we have done when we used all 28 features.

Table 5.10: **Study of regularization techniques.** Comparison of trainings with and without the use of dropout or autoencoders pretraining in term of the Area Under the ROC Curve (AUC). The model used for the tests is a deep network with 3 hidden layers, each of them with 300 units, trained on 2 million events.

Technique	**Architecture**	**AUC** Low-level	**High-level**	**Complete**
Basic	DN 3 layers	0.811±0.001	0.7887±0.0001	0.8385±0.0008
Dropout	DN 3 layers	0.717±0.001	0.773±0.002	0.807±0.003
Autoencoder	DN 3 layers	0.807±0.001	0.7919±0.0006	0.8372±0.0008

5.4 Comparing Shallow and Deep Networks

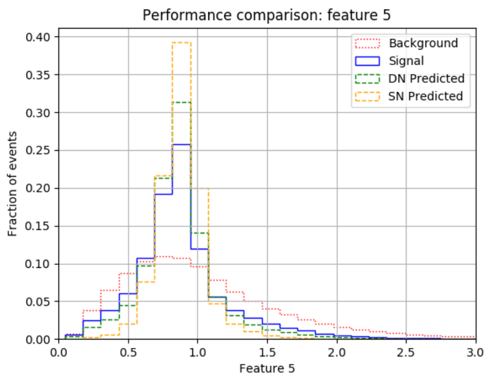
We have already noted in several occasion that shallow networks performances can compete with those of deep networks only if the first ones are trained with the high-level features while the second with the low-level features. That means shallow network need a help from human engine. To better understand the differences between how the two algorithms work, we shall investigate how each model generalize, or rather how they reconstruct the discriminant features necessary to *see* our resonant exotic signal.

Histograms in 5.15 show all the high-level features distribution. Red and blue plots represent in order background and signal expected distributions, those obtained using the already known label 0 and 1 to classify events. Green plots outline the reconstruction of signal events acted by the deep network training with low-level features, while the yellow ones those acted by the shallow network training with high-level features.

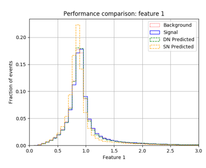
Figure 5.15a is the one which better puts in evidence the different behavior of the two. We can see that the yellow histogram accentuates the peak more than it should do, while the green looks more like the blue one. In virtually, since the shallow network gets a restricted number of inputs, it has less information to extrapolate so it tends to emphasize them. On the other hand, the deep network has not got any prior about which discriminative features to construct in order to delineate the most important traits of the dataset. So it has a conservative behavior.

These results therefore show that handmade features waste part of information; using them to do statistical analysis puts a lower limit to the maximum performance obtainable, leading to the need of more and more data.

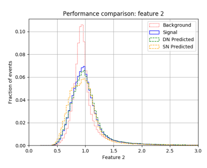
So in high energy physics, deep network techniques reach performances that human engine alone can not.



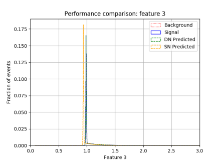
(a) Feature 5: M_{bb}



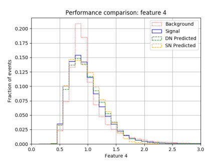
(b) Feature 1: M_{jj}



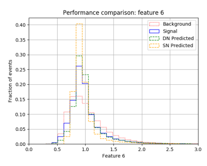
(c) Feature 2: M_{jjj}



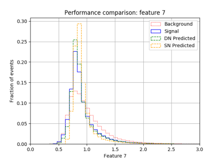
(d) Feature 3: $M_{l\nu}$



(e) Feature 4: $M_{jl\nu}$



(f) Feature 6: M_{Wbb}



(g) Feature 7: M_{WWbb}

Figure 5.15: **Distribution Comparison.** Distribution of signal predictions of a Shallow Network trained with high-level features compared with that of a 3 hidden layers Deep Network trained with low-level features and with the expected distribution. Autoencoder pretraining and dropout have not been used.

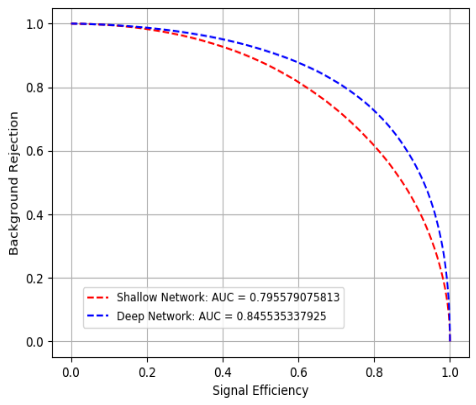


Figure 5.16: **ROC curves comparison.** Comparison between shallow network trained on 8.8 million events using high-level features and deep network trained on 8.8 million events using low-level features.

5.5 Comparing with Baldi, Sadowski and Whiteson results

Now, at the end of this essay, we just have to compare our best results for signal discrimination to those presented in [13].In the following table there are placed side by side the AUC final results found in [13] and that of our tests.

In previous sections we concluded that the best solution is to use the 8.8 million events as training sample. With such a large training sample our virtual machines are not able to compute autoencoder pretraining while dropout does not causes any improvement in AUC results.

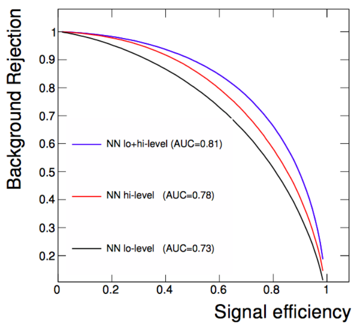
Therefore our best results are the ones obtained without the use of regularization techniques.

Table 5.11: **Study of network size and depth.** Comparison of shallow networks with different number of hidden units (single hidden layer), and deep networks with varying hidden layers in term of the Area Under the ROC Curve (AUC). The deep networks have 300 units in each hidden layer. On the left side AUC resulting in [13] and on the right those resulting from our tests.

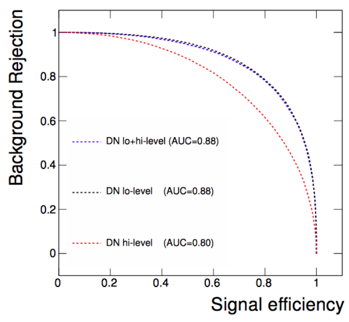
		AUC			**AUC**	
Technique	**Low-level**	**High-level**	**Complete**	**Low-level**	**High-level**	**Complete**
NN 300-hidden	0.733	0.777	0.816	0.675	0.763	0.812
NN 1000-hidden	0.788	0.783	0.841	0.731	0.778	0.812
NN 2000-hidden	0.787	0.788	0.842	0.739	0.780	0.814
NN 10000-hidden	0.790	0.789	0.841	0.746	0.777	0.845
DN 3 layers	0.836	0.791	0.850	0.840	0.793	0.859
DN 4 layers	0.868	0.797	0.872	0.845	0.794	0.860
DN 5 layers	0.880	0.800	0.885	0.846	0.796	0.858
DN 6 layers	0.888	0.799	0.893	0.847	0.794	0.859

Our results do not achieve the performances in [13]. Our algorithms, indeed, run for about 40-60 epochs before reaching the condition of overfitting, while those implemented by Baldi, Sadowski and Whiteson run for 200-1000 epochs. This is a great difference. It means they were able to regularize the algorithms such that overfitting was pulled away.

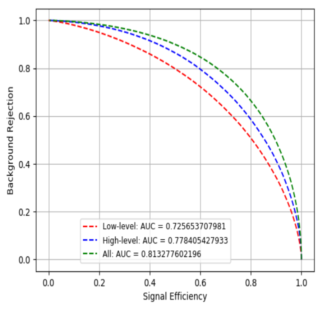
Even if we did not achieve the same results as in previous work, we are able to prove that using low-level features in deep networks leads to better performance than using high-level. Conversely, for shallow network it does not happen.



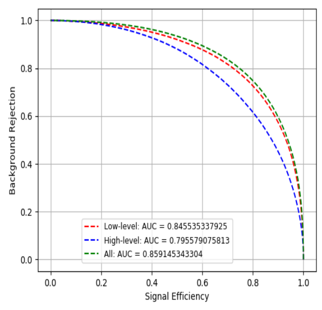
(a) NN performances in [13].



(b) DN performances in [13].



(c) NN performances from our work.

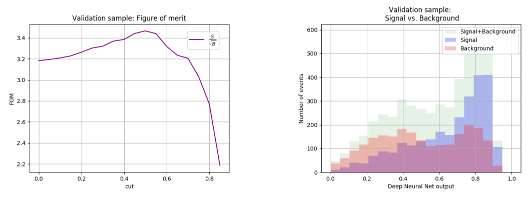


(d) DN performances from our work.

Chapter 6

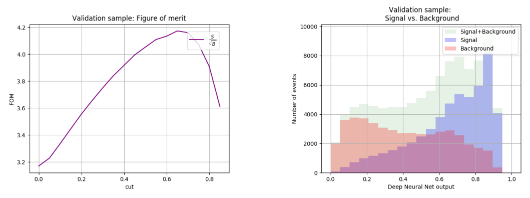
Applying results to discover new Physics

6.1 Figure of merits: the physical meaning of our work



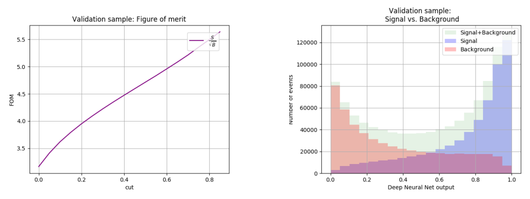
(a) FOM (b) Event distributions.

Figure 6.1: **40 000 events.**Test on deep network with 3 hidden layers and 300 units er layer using raw features.



(a) FOM (b) Events distribution.

Figure 6.2: **800 000 events.**Test on deep network with 3 hidden layers and 300 units er layer using raw features.



(a) FOM (b) Events distribution.

Figure 6.3: **8 800 000 events.** Test on deep network with 3 hidden layers and 300 units er layer using raw features.

Up to now, we have not discussed yet physical implications of our results. We compared different networks via AUC in order to select the best neural architecture and the best choice of hyper-parameters. Now we aim to translate AUC in a more suitable metric for physics scope, in order to get a measure of *discovery significance.*
Figures 6.1a, 6.2a and 6.3a show how FOM value changes varying the cut point for labeling of signal predictions. In the first example FOM trend shows a peak at about 0.5 while increasing the number of training events, it gradually disappeared till it completely vanish thus the maximum value for $\frac{S}{\sqrt{B}}$ is obtained when the cut point tends to 1.This because output predictions get nearer and nearer to a binary classification. A part from the trends, such FOMs say nothing about a concrete significance value: as Figures 6.1b, 6.2b and 6.3b show, our dataset is composed equally of signal and background, but that does not correspond to physical reality. As we have already said in chapter 3, signal events of new physics are rare, so their cross section is smaller than that of background. To understand if obtained results can help discriminating new signals we need to renormalize histograms such that they correspond more or less to real experiments.

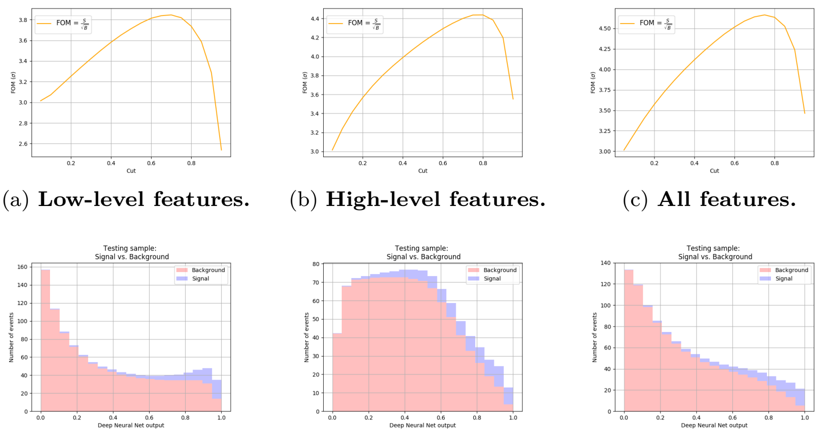
To compare significance results to those calculated by Baldi, Sadowski and Whiteson (see TABLE I in [13]) we have to assume 100 signal events and 1000 background events and compute our FOM. [1]

[1]In [13] significance has been computed in a more sophisticated way, evaluating the Likelihood ratio between different hypothesis of events distribution based on a *signal strenght* parameter (μ). They assume distribution to be background-only like if $\mu = 0$ and background plus signal if $\mu = 1$, such that probability distribution for n events is a *marked Poisson model*:

$$P(\{x_1, .., x_n\}|\mu) = Pois(n|\mu S + B)[\prod_{e=1}^{n} \frac{\mu S f_S(x_e) + B f_B(x_e)}{\mu S + B}]$$

The analysis have been done using `HistFactory` on Root. More information at [21].

Using this efficiency rate, histograms of events distribution and FOM functions assume the following shapes:



(a) **Low-level features.** (b) **High-level features.** (c) **All features.**

(d) **Low-level features.** (e) **High-level features.** (f) **All features.**

Figure 6.4: **FOM and histograms after renormalization.** Performances obtained after normalization for deep network with 3 hidden layers and 300 units per layer, using 8.8 million training events assuming 100 signal events and 1000 background events.

Relative results for significance are:

Model	Features	Cut	Sgnificance	Sgnificance
NN	Low-level	0.45	3.34 σ	2.5 σ
	High-level	0.65	3.69 σ	3.1 σ
	All	0.7	4.11 σ	3.7 σ
DN	Low-level	0.75	4.44 σ	4.9 σ
	High-level	0.7	3.85 σ	3.6 σ
	All	0.75	4.66 σ	5.0 σ

Table 6.1: **Significance results**. Results obtained after histogram renormalization assuming 100 signal events and 1000 background. Training on shallow network (NN) with 300 hidden units and on deep network (DN) with 3 hidden layers and 300 units per layer. On the left side significances resulting by our tests, on the right those resulting in [13].

As we could foresee from AUC comparison in previous chapter, our significances do not achieve those in [13]. We can anyway find the minimum number of signal events and a correspondent cross section, to achieve a discovery.

To do it we can, for instance, compute an hypothesis test based on χ^2 distribution. We can define χ^2 as follows:

$$\chi^2 = \sum_{i=1}^{n_{bins}} \frac{(O_i - A_i)^2}{A_i} \tag{6.1}$$

where the sum is over histogram bins, O_i is the observed i-th bin height and A_i the expected one. We assume as observed distribution that of signal plus background while as expected distribution that of background only, which is also called null hypothesis (H_0). Distribution of signal and background events over a single bin is poissonian like, thus the variance for a number of expected events A_i in a single bin is A_i. We can then rewrite equation 6.1 as

$$\chi^2 = \sum_{i=1}^{n_{bins}} \frac{((s_i + b_i) - b_i)^2}{b_i} = \sum_{i=1}^{n_{bins}} \frac{s_i^2}{b_i}$$

where s_i is the number of signal events in the i-th bin, b_i that of background events and denominator is the poissonian variance of b_i. From statistics we know that the expectation value for a quantity distributed as χ^2 is equal to the number of degree of freedom (ν). In our histograms we have 49 total bins between 0 and 1, 0.02 width, but we decided to restrict the χ^2 computation to a smaller region, choosing as cut point 0.94, the value for which FOM is maximum and discrepancies between signal and background distributions are highlighted

as best as possible. In this way the number of considered bins is only 3 and this is also our ν. Assuming a confidence level (CL) of 95%, the cut point for a χ^2 with $\nu = 3$ is about 7.81, thus we can discard H_0 hypothesis if the computed χ^2 is major of 7.81.
To procede in computing χ^2, we first need to renormalize background distribution such that its integral coincides with the total number of expected events. The number of events (N) for a process with cross section σ is given by

$$N = L_{int} \cdot \sigma$$

where L_{int} is the textitintegrated luminosity with respect to time. L_{int} is a parameter used to characterize the performance of particle accelerator. Here we assume $L_{int} = 20.3\, fb^{-1}$, as in [22]. For what concerns background cross section we reference to simulations run with Madgraph5 [2]:

$$\sigma_{bkg} = (27.92 \pm 0.08)\, pb$$

Number of background events becomes

$$N_{bkg} = L_{int} \cdot \sigma_{bkg} \approx 566776$$

Fixed N_{bkg}, we varied number of signal events till we found the minimum value for which χ^2 allows to reject H_0. We found that the minimum number of signal events is $N_{sign} \approx 886$. Normalizing signal distribution respect to 886 and considering as cut point 0.94 we found $FOM \approx 2.53$ in unit of σ and $\chi^2 \approx 7.91$, which allow to reject H_0 with $95\%\, CL$. We can also convert the result in term of minimum cross section for signal evidence. Assuming the same integrated luminosity used for background ($20.3\, fb^{-1}$), minimum cross section is equal to

$$\sigma_{sign} = \frac{N_{sign}}{L_{int}} \approx 43.6\, fb$$

Thus the ratio between signal and background cross section is

$$\frac{\sigma_{sign}}{\sigma_{bkg}} \approx \frac{43.6\, fb}{27.92\, pb} \approx 1.6 \cdot 10^{-3}$$

[2]See Appendix A.

Chapter 7

Future prospectives

In this work several tests have been done. We studied how performances are influenced by a specific network architecture, especially by the number of hidden layers; we tried new methods for algorithm regularization with poor results and we observed variance of predictions. During this work we encountered troubles related to handling large dataset size: first of all, we needed moving to virtual machines useful to upload all data at a time; nevertheless there were still limits in the choice of algorithm complexity due to the RAM of these machines which does not allow to apply Autoencoder pretraining to the full dataset. Moreover, the long training time stretched also the time needed to execute the all optimization tests. Possible solutions to these drawbacks already exist.

To speed up the computation and optimize memory management we can procede in two different ways: on one hand, we can operate on hardware level strengthening the capacity of processors, especially moving from CPUs to GPUs; on the other hand, we can upgrade software going from single thread to multi thread executions.

Clustering: distributed Keras packages As for software development, next step in our work should be, for instance, moving from Keras packages to that of Distributed Keras, a deep learning framework still under construction based on distributed optimization algorithms and data parallel methods [1]. Distributed Keras is built on Keras libraries for machine learning and Apache Spark, a general engine to process large sets of data using cluster computing [2]. The idea is to exploit all the power of virtual machines cluster created into in the Cloud to make all them contribute to run the same single learning algorithm using only one shared training datasets.

Autoencoder potentiality: building latent spaces For what concerns future prospectives on deep neural networks in High Energy Physics, more in-depth studies should be carry on about unsupervised training algorithms as autoencoders. Their capability in finding the principal features of a data sample will be optimized and exploited to recognize new topologies on phase space which human engine and theoretical physics have not reached yet.

In any case, development in machine learning with deep networks for High Energy Physics is still in progress and algorithm optimization has not been concluded. During this work we have been able to notice the sensitivity of neural networks to the specific input features besides network architecture. We have seen that having a large training dataset could solve part of regularization problems without looking for more sophisticated algorithms, but at the same time this means facing new troubles in handling big data, as infrastructures and software development. This work seeks to take only a first look at this subject. Even if our results do not achieve the best performances already experimented in [13], we were able to prove the overcoming of deep neural networks on shallow neural networks. This result make us hopeful that getting on with machine learning algorithms would bring to improvements in analysis performances at LHC.

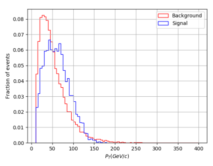
[1]More information about Distributed Keras at: `https://github.com/cerndb/dist-keras`
[2]Apache Spark: `https://spark.apache.org`

Appendices

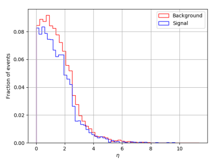
Appendix A

Data simulation

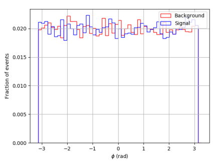
Dataset used as training sample was standardize such that gaussian and uniform distributions, namely that of η and ϕ, have mean 0 and standard deviation 1, while exponential distributions (invariant mass and transverse momentum) have mean 1. To reconstruct the effective features distributions we simulated 10000 signal and background events using MADGRAPH5 with DELPHES and PYTHIA at $\sqrt{s} = 8\,TeV$ at leading order. Thus we could see the real scale of each quantity. The distributions obtained due to simulations are shown below.



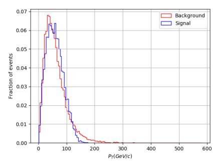
(a) Lepton p_T (GeV)



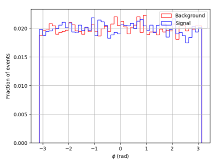
(b) Lepton η



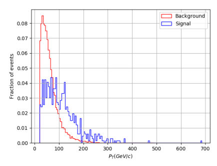
(c) Lepton ϕ (rad)



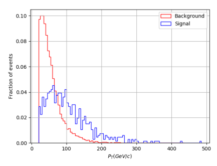
(d) Missing momentum p_T (GeV)



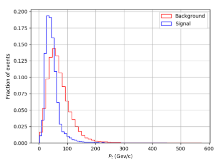
(e) Missing momentum ϕ (rad)



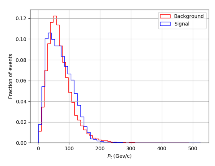
(f) Jet 1 p_T (GeV)



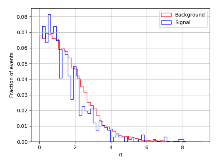
(g) Jet 2 p_T (GeV)



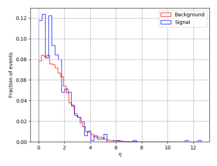
(h) Jet 3 p_T (GeV)



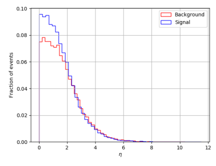
(i) Jet 4 p_T (GeV)



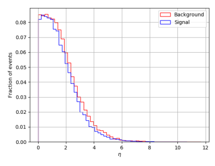
(j) Jet 1 η



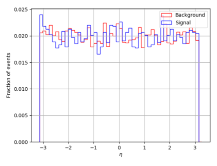
(k) Jet 2 η



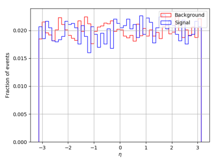
(l) Jet 3 η



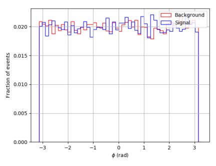
(m) Jet 4 η



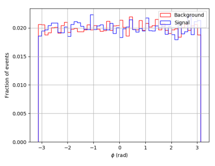
(n) Jet 1 ϕ (rad)



(o) Jet 2 ϕ (rad)



(p) Jet 3 ϕ (rad)



(q) Jet 4 ϕ (rad)

Figure A.1: **Low-level features.** Distributions obtained from MADGRAPH5 simulations.

Bibliography

[1] Jeff Byers (July 3rd, 2017)
The physics of data.
`www.nature.com/naturephysics`

[2] Michael Nielsen
Neural networks and deep learning
`http://neuralnetworksanddeeplearning.com/index.html`

[3] Recurrent neural networks (RNNs)
`https://en.wikipedia.org/wiki/Recurrent_neural_network`

[4] CS231n Convolutional Neural Networks for Visual Recognition
`http://cs231n.github.io/neural-networks-2/`

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton (2012)
ImageNet Classification with Deep Convolutional Neural Networks.

[6] Yoshua Bengio (Version 2, September 16th, 2012)
Practical Recommendations for Gradient-Based Training of Deep Architectures

[7] Alessandro Bettini (Second Edition, 2014)
Introduction to Elementary Particle Physics

[8] The Large Hadron Collider (LHC)
`https://home.cern/topics/large-hadron-collider`

[9] LHC complex
`http://atlas.kek.jp/sub/photos/Accelerator/0107024_01.jpg`

[10] Inside a proton
`http://www.quantumdiaries.org/2016/02/01/spun-out-of-proportion-the-proton-spin-crisis/`

[11] Inside a detector (CMS slice)
`https://cms-docdb.cern.ch/cgi-bin/PublicDocDB/RetrieveFile?docid=4172&filename=CMS_Slice.gif&version=2`

[12] K. Hornik, M. Stinchcombe, H. White (1989)
Multilayer Feedforward Networks are Universal Approximators

[13] P. Baldi, P. Sadowski, and D. Whiteson (June 5th, 2014)
Searching for Exotic Particles in High-Energy Physics with Deep Learning
`https://arxiv.org/pdf/1402.4735.pdf`

[14] M. Bianchini, F. Scarselli (January 6th, 2014)
On the complexity of shallow and deep neural network classifiers
IEEE Transactions on Neural Networks and Learning Systems (Volume: 25, Issue: 8, Aug. 2014)
`https://pdfs.semanticscholar.org/2786/feab5c644bf0bde98fb6f9d1dbd0b58ca80c.pdf`

[15] Building Autoencoders in Keras - The Keras Blog
`https://blog.keras.io/building-autoencoders-in-keras.html`

[16] D. Erhan, Y. Bengio, A. Courville, P.A. Manzagol, S. Bengio P. Vincent
Why Does Unsupervised Pre-training Help Deep Learning?
(Journal of Machine Learning Research 11 (2010))

[17] Example of autoencoder pretraining.
`https://www.researchgate.net/figure/304249651_fig3_Fig-3-Pretraining-procedure-for-autoencoder-The-weights-of-a-5-layer-autoencoder`

[18] Keras Documentation
`https://blog.keras.io/building-autoencoders-in-keras.html`

[19] Theano Documentation
`http://deeplearning.net/software/theano/`

[20] Tensorflow Documentation
`https://www.tensorflow.org`

[21] K. Cranmer, G. Lewis, L. Moneta, A. Shibata, W. Verkerke
HistFactory: A tool for creating statistical models for use with RooFit and RooStats
`http://cds.cern.ch/record/1456844/files/CERN-OPEN-2012-016.pdf?subformat=pdfa&version=1`

[22] ATLAS Collaboration (December 6th, 2013)
Search for a multi-Higgs-boson cascade in $W + W - b\bar{b}$ *events with the ATLAS detector in pp collisions at* $\sqrt{s} = 8TeV$
`https://arxiv.org/pdf/1312.1956.pdf`

[23] Distributed Keras
`https://github.com/cerndb/dist-keras`