Image & Pervasive Access Lab
A French-Singaporean joint laboratory

Supélec

SCATTERING HIDDEN MARKOV TREES

IMAGE REPRESENTATION AND SCATTERING TRANSFORM MODELING

Jean-Baptiste Regli

2013-2014

RESEARCH REPORT

Academic supervisor : James Nelson
Sponsor : Dstl/UCL Impact studentship

UCL
Department of Statistical Science
London

Contents :

List of figures :

1 Introduction :

1.1 General introduction :

The ***A*STAR Image*** & ***Pervasive Access Laboratory*** (IPAL) is a Franco-Singaporean laboratory concentrating its research on two main themes : Medical Image Understanding and Pervasive Access and Wellbeing Management.

For my final internship I have joined the ***Medical Image Understanding*** team under the supervision of ***Antoine Veillard and Oliver Morère***, both working on theoretical machine learning problems. Thanks to them I had a glimpse over several machine learning methods (Self Organizing Maps, Support Vector Machines, Neural Networks...), before focusing on the ***Deep Neural Networks***.

1.2 Image representation :

Recently, deep learning has drawn the attention of the machine learning community by delivering state-of-the-art results on various tasks such as classification [12,16,18,19,42], regression [20], dimensionality reduction [22,23] or modeling textures [21] in applicative domains as variate as computer vision, robotics [25], natural language processing [26,27] and may others. In addition, those good performances can be obtained with ***no or minor pre-processing on the data*** [8,46], making those networks more convenient to use than those whose performances are highly correlated to the quality of the pre-processing and feature engineering.

To do so, deep architectures learn a new representation of the input data at each stage of the procedure. Those representations start from very basic concepts and progress toward a high level representation of the input. For example, applied to computational vision, this procedure would automatically extract a low-level features (e.g. : presence of edge), then capture more complex but local shapes and end-up to the identification of abstract categories associated with objects -or part of objects- which are parts of the image. This high level representation now capture enough understanding of the scene to perform analyses on it.

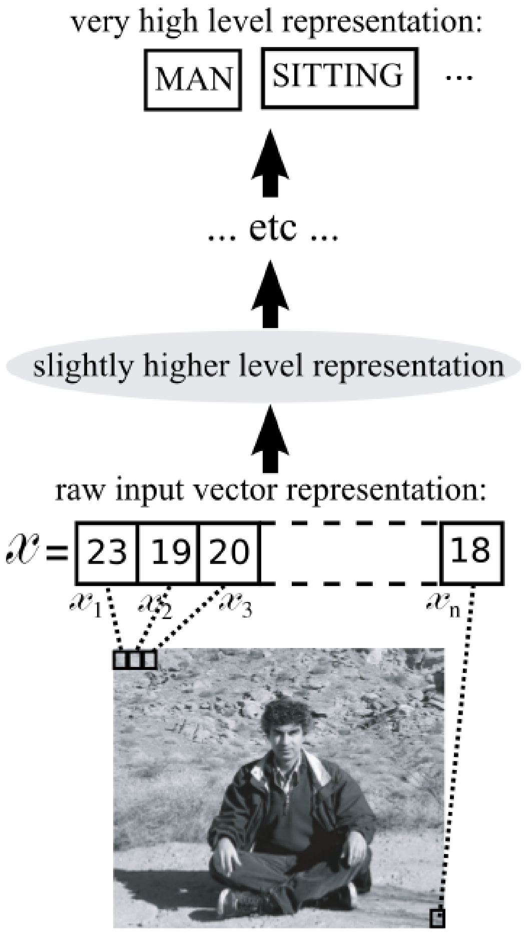

very high level representation:
MAN
SITTING
...
... etc ...
slightly higher level representation
raw input vector representation:
$\mathcal{X}$ =
23
19
20
18
x_1
x_2
x_3
x_n

FIGURE 1.1 – Deep learning procedure applied to a computer vision example

Thus, deep learning methods aim at automatically and recursively learn higher level feature hierarchies among features themselves composed of lower level features. By doing so, we allow the system to learn complex functions mapping "directly" the raw data to the output, ***without relying on any hand-crafted features.***

1.3 Outline of the report :

The aim of this report will be to provide a broad overview on deep neural networks and on the observations realized during my internship. We will start by introducing the concepts required to understand what an artificial neural network is (chapter **??**). We will describe the mathematical concepts it is constructed upon, how it is trained and provides several successful examples of applications.

Second, we will focus on the deep neural networks themselves (chapter 3). We will define more precisely the notion of depth and how it is measured, then explain what the advantages of deep architectures are. We will give an overview of the bottlenecks that have limited the use of deep neural networks before and describe a training procedure adapted to those architectures. Finally we will have a look at how those networks are built.

In a third and fourth chapters, two types of deep neural network will be introduced in more details. First the the deep belief network -chapter 4- and then the stack auto-encoders -chapter 5. We will describe the theoretical background of those networks as well as their training procedure.

In a fifth chapter, we will have a look to how the over-fitting risk is managed for the auto-encoders. Indeed, due to their learning procedure those networks are very likely to over-fit to the training data and several methods exist to prevent this. We will provide there an overview of those regularization methods (chapter **??**).

The sixth chapter will be dedicated to the description of the technical choices made during the internship to implement the stack auto-encoders (chapter **??**). We will have a look at the Python library used to leverage the GPUs for training the deep neural networks and then present the implemented objects.

The final chapter will offer an overview of the experiments run using the deep machines implemented during the internship (chapter 6). We will first assess the performances on the famous MNIST dataset [7] and then present the results obtained on an unknown data analysis problem.

2 Scattering transform :

In this section, basic and general information will be provided about the artificial neural network in order to set the basis of the concepts necessary to study artificial deep neural networks.

When one refers to an -artificial- neural network, one usually narrows the concept down to the feedforward neural networks. This is one of the most basic types of artificial neural networks designed to learn distributed representations of the input data. It consists of a ***series of information processing operations***, whereby information moves strictly in a forward direction (i.e. : no retro-action loop), from the input units, through latent -hidden- units, if any and finally to the output units.

2.1 Convolutional Neural Network :

The simplest form of feedforward neural network is the perceptron [1], built using the mathematical model of a formal neuron [4].

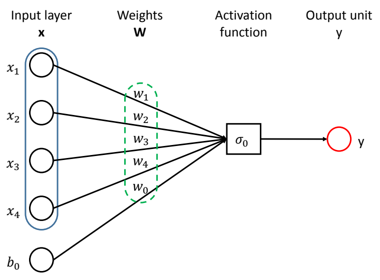


FIGURE 2.1 – The perceptron network : Uses a set of weights $\mathbf{W}$ and a feedforward activation function σ_0 to map a layer of input units $\mathbf{x}$ to the output unit $\hat{\mathbf{y}}$

As illustrated in Figure 2.1, the output value $\hat{\mathbf{y}}$ of an input vector $\mathbf{x}$ is obtained from a set of weights -parameters- $\mathbf{W}$ and a feedforward activation function σ_0. The coefficient w_0 is a parameters known as the bias -or offset- and it is connected to an input unit b_0 that is permanently set to 1. For an input with N dimensions, the mathematical expression of the output may be :

— ***Linearly activated :*** through a weighted sum of the inputs.

$$\hat{\mathbf{y}} = \sigma_0(\mathbf{x}, \mathbf{W}) = \sum_{i=0}^{N} w_i x_i \tag{2.1}$$

— ***Binarized :*** by a threshold t.

$$\hat{\mathbf{y}} = \sigma_0(\mathbf{x}, \mathbf{W}) = \begin{cases} 1 & \text{if } \sum_{i=0}^{N} w_i x_i \geq t \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

Depending on the application and the objective of the learning (discrimination, generalization...), many different activation functions σ_0 can be used. The list bellow is far from exhaustive but introduces the most common ones :

— ***The step function :*** This is the function historically used by the first perceptron model. It has the advantage of offering very straight forward computation. However with the progress done in the domain of computational power, some more complex functions are usually preferred over this one.

$$\hat{\mathbf{y}} = \sigma_0(\mathbf{x}, \mathbf{W}) = \begin{cases} 1 & \text{if } \sum_{i=0}^{N} w_i x_i \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

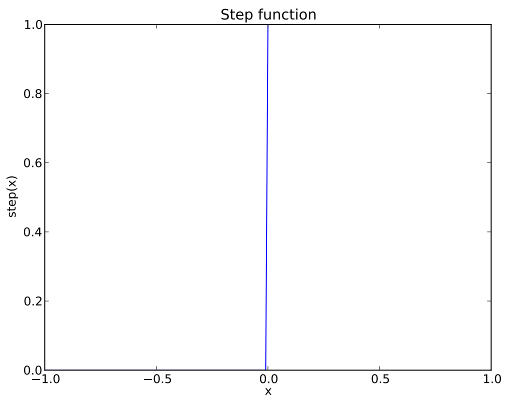


FIGURE 2.2 – The step function : Historically used in the first model of artificial network

— ***The sigmoid function :*** It is the most popular activation function mainly because its derivatives are easy to compute[1]. It can be used for binary classification problems as well as regression ones.

$$\hat{\mathbf{y}} = \sigma_0(\mathbf{x}, \mathbf{W}) = \frac{1}{1 + \exp\left(-\sum_{i=0}^{N} w_i x_i\right)} \tag{2.4}$$

1. For the sigmoid function f : $f' = f(1-f)$

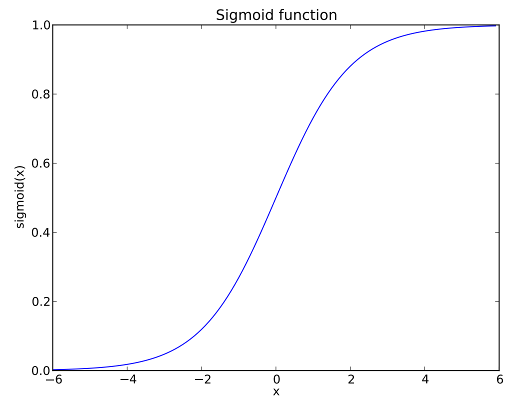


FIGURE 2.3 – The sigmoid function : The most widely used activation function for binary classification and regression problems

— ***The hyperbolic tangent function :*** This activation function is also quite popular for binary classification or regression problems and its derivatives are also quite easy to compute[2]. It can be used on the same type of problem as the sigmoid but offers a stronger discriminative capacities (because of steeper derivatives in the middle of the domains) than the sigmoid which can be an advantage in some situations.

$$\hat{\mathbf{y}} = \sigma_0(\mathbf{x}, \mathbf{W}) = \tanh\left(\sum_{i=0}^{N} w_i x_i\right) \tag{2.5}$$

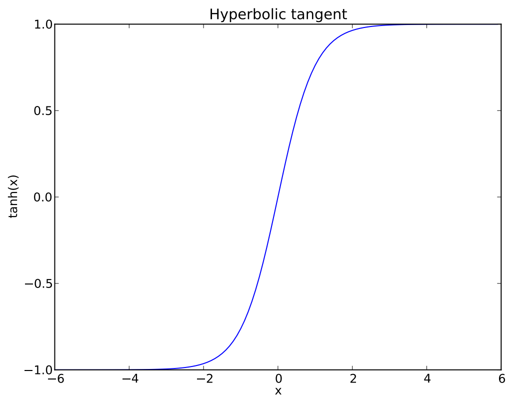


FIGURE 2.4 – The hyperbolic tangent function : Another widely used activation function for binary classification problems as well as regression ones

— ***The rectified linear function :*** This activation function has been highlighted recently as it seems to provide, in some cases, better results than the classic sigmoid for deep neural networks. Again this is due to a higher discriminative performances helpful in some cases.

2. For the hyperbolic tangent function f : $f' = 1 - f^2$

$$\hat{\mathbf{y}} = \sigma_0(\mathbf{x}, \mathbf{W}) = \begin{cases} \alpha \sum_{i=0}^{N} w_i x_i & \text{if } \sum_{i=0}^{N} w_i x_i \geq 0 \\ 0 & \text{otherwise} \end{cases} \\ = \max(0, \sum_{i=0}^{N} w_i x_i) \tag{2.6}$$

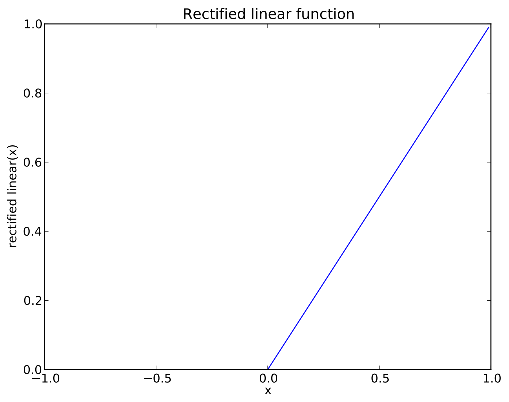


FIGURE 2.5 – The rectified linear function : A fairly new activation function for binary classification problems that seems to have better discrimination capacities compared to the sigmoid function

— ***The softmax function :*** This activation function is the multi-classes version of a sigmoid. It is useful predominantly in the output layer of a clustering system.

$$\hat{\mathbf{y}}_j = f_e(\mathbf{x}, \mathbf{W})_j = \frac{\exp(w_j x_j)}{\sum_{i=0}^{N} \exp(w_i x_i)} \tag{2.7}$$

2.2 Scattering network :

The cost function is used to measure the error done by the learned predictor. It has to measure the difference between an the prediction $\hat{y}$ done by the predictor for an example $\mathbf{x}$ and the effective label of the input y. A cost function should tends to 0 when $\hat{y} \to y$.

While, the cost function used in the training procedure (section 2.4 and the equation 2.14) can be tailored to the problem tackled (custom made cost function leveraging specific prior knowledge on the dataset to be learned), there are also several usual error functions.

2.2.1 0-1 loss :

This is the simplest error function. It is not widely used anymore as it is too simple to allow good learning performances as well as because it does not have interesting derivatives.

$$J(x, y) = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{if } x \neq 0 \end{cases} \tag{2.8}$$

2.2.2 Squared loss :

This is an error function based on the L_2 norm. It is one of the most generic lost function and it can be used for the classification or regression problems whenever there is no special metric tailored o the problem.

$$J(x, y) = \|y - x\|_2 \tag{2.9}$$

2.2.3 Cross-entropy loss :

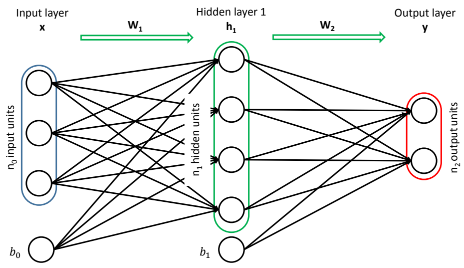
This metric is mostly used with stochastic machines (such as the RBMs - chapter 4) or when the inputs are interpreted as ***bit vectors or as vectors of bit probabilities***. Its general expression is :

$$J(x, y) = -x \log y + (1 - x) \log(1 - y) \tag{2.10}$$

2.3 Spatial Wavelet transform :

A ***Multilayer Perceptron network*** (MLP) consists of multiple stacked perceptron where each layer is fully connected to the next one but where there is no intra-layer connection. Figure 2.6 presents a three-layer MLP. The first layer is a passive (i.e. : no computation done) input layer, followed by an hidden (i.e. : latent) layer where a first level of representation of the data is computed. Finally, the third and output layer performs the computation of a second level of representation of the input data.

Figure 2.6 – Three-layer multi-layer perceptron : Use two successive steps of weighted data information processing (latent layer and output layer) to map the inputs to the outputs



Unlike the perceptron network, the multi-layer perceptron is able to learn a model of data that is not linearly separable. A ***three-layer MLP is considered to be a universal approximator of continuous functions***. This was proven by [5] for sigmoid activation function and by [6] for the multi-layer feedforward network in general, regardless of the choice of activation function

2.4 Roto-translation wavelet transform :

A multi-layer perceptron is trained using the combination of two algorithms :

— ***The Backpropagation algorithm :*** To compute the partial derivatives of the cost function used with regard to each parameters of the network [2,3].

— ***The Gradient descent algorithm :*** To find the optimal parameters that minimize the cost function over the training set.

Let us consider a labeled training set $\mathcal{D}_{train} = (\mathbf{x_k}, y_k)_{k\in[0,|\mathcal{D}_{train}|]}$ used to train a d-layers perceptron and let us define the following notations :

— $J(\mathbf{W}, \mathbf{b}, \mathbf{x_k}, y_k) = J(\mathbf{x_k}, y_k)$: the cost function associated to an example x_k and to the network parameters $\mathbf{W}$ and $\mathbf{b}$.

— $z^{(l+1)} = W^{(l)}x_k + b^{(l)}$: The input of the layer $l+1$.

— $a^{(l+1)} = \sigma_k(z^{(l+1)}$:The value of the activation of the layer l.

The backpropagation procedure for a given tuple of training examples $(\mathbf{x}_k, y_k)$ is described by the algorithm 1 :

```
1) Perform a feedforward pass, computing for each $k \in [1, d-1]$, the activation $a^k$ for
the layer $L_k$.
2) For the output layer $L^{(d)}$ set :
        $\delta^{(d)} = \frac{\partial}{\partial \mathbf{z}^{(\mathbf{l})}} J(\mathbf{x_k}, y_k)$        (2.11)
3)for  $l$ in [d-1,...,1] do
    Set :
        $\delta^{(l)} = (\mathbf{W}^{(\mathbf{l})})^T \delta^{(l+1)}$        (2.12)
end
4) Compute the desired partial derivatives :
        $\nabla_{W^{(l)}} J(\mathbf{W}, \mathbf{b}, \mathbf{x_k}, y_k) = \delta^{(l+1)}(a^{(l)})^T$        (2.13a)
        $\nabla_{b^{(l)}} J(\mathbf{W}, \mathbf{b}, \mathbf{x_k}, y_k) = \delta^{(l+1)}$        (2.13b)
```

Algorithm 1: The backpropagation algorithm on a d-layer perceptron

And then the error backpropagation is used to train a multi-layer perceptron with continuous differentiable activation functions and it can be described as follow :

```
1) Set $\Delta\mathbf{W}^{(l)} := 0$, $\Delta\mathbf{b}^{(l)} := 0$ (matrix/vectors of zeros for all layers $l$).
2) for $k = 1 to |\mathcal{D}_{train}|$ do
    2.a) Use the backpropagation algorithm 1 to compute $\nabla_{W^{(l)}} J(\mathbf{W},\mathbf{b},\mathbf{x_k},y_k)$ and $\nabla_{b^{(l)}} J(\mathbf{W},\mathbf{b},\mathbf{x_k},y_k)$
    2.b) Set : $\Delta\mathbf{W}^{(l)} := \Delta\mathbf{W}^{(l)} + \nabla_{W^{(l)}} J(\mathbf{W},\mathbf{b},\mathbf{x_k},y_k)$
    2.c) Set : $\Delta\mathbf{b}^{(l)} := \Delta\mathbf{b}^{(l)} + \nabla_{b^{(l)}} J(\mathbf{W},\mathbf{b},\mathbf{x_k},y_k)$
end
3) Update the parameters :
```

$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} - \alpha \frac{\Delta\mathbf{W}^{(l)}}{m} \tag{2.14a}$$

$$\mathbf{b}^{(l)} := \mathbf{b}^{(l)} - \alpha \frac{\Delta\mathbf{b}^{(l)}}{m} \tag{2.14b}$$

Algorithm 2: A simple gradient descent algorithm for a learning rate α

Note that those algorithms requires to use activation functions with defined derivatives and that some activation functions introduced in the previous section (section 2.1) are not differentiable (step function, rectified linear function...) over $\mathbb{R}$. However most of the time, the derivatives of the activation function can easily be extended over all $\mathbb{R}$.

Note also that the procedures given here describe the simplest gradient descent algorithm but several improvements have been proposed to improve its accuracy. Among those improvements, there is the mini-batch gradient descent [35], which average the update to be made over several training examples. This method speed up the convergence toward the optimal parameters by minimizing the effect of outliers on the descent. Another first order variation of the gradient descent widely used to train -deep- neural networks is the stochastic gradient descent but some researchers start to recommend to use second order optimization methods -such as L-BFGS- to train the neural networks [8, 35].

As an example for the calculation, let us applied those algorithms to a three layers MLP (Figure 2.6), with an input layer $\mathbf{x} \in \mathbb{R}^I$ mapped to a latent layer $\mathbf{h} \in \mathbb{R}^J$ via the weights matrix $\mathbf{W_1}$, itself mapped to an output layer $\hat{\mathbf{y}} \in \mathbb{R}^C$ via the weight matrix $\mathbf{W_2}$. Suppose that we are using a loss function $\mathcal{L}$ -either a tailored one or one described in the section 2.2- then the weights of the network can be updated using the following procedures :

$$\begin{aligned} w_{2,j,c} &= w_{2,j,c} + \epsilon \frac{\partial \mathcal{L}}{\partial w_{2,j,c}} \\ w_{1,i,j} &= w_{1,i,j} + \epsilon \frac{\partial \mathcal{L}}{\partial z_j} \frac{\partial z_j}{\partial w_{1,i,j}} \end{aligned} \tag{2.15}$$

2.5 Application to classification :

Artificial neural networks can be used on a wild range of applications. As seen in section 2.3, neural networks are universal approximators. Thus they are theoretically able to learn any kind of relationship in between the input and the output. As we will see in section 3.1, these results are only theoretical and the performances of classical multi-layers perceptron are limited, however they perform very well for tasks such as capturing associations or discovering regularities within a set of patterns, applications where the volume, the number of variables or diversity of the data is very great (compared to SVMs for example), applications where the relationships between variables are vaguely understood or, application where the relationships are difficult to describe adequately with conventional approaches.

For example, they have been successfully applied for classification problem, such as handwritten digit recognition [7], or for regression problems, such as predict the credit score of a individual [44].

3 Probabilistic graphical models :

In this chapter, we will introduce the concept of deep neural networks more at length by first defining the notion of depth. Once this question solved, we will assess the various motivations to make such important push toward this type of networks. We will then explain the specific training procedure used for deep networks and finally describe the main types of deep neural networks.

3.1 Bayesian Network :

Formally the ***depth is not a intrinsic property of an architecture***. It depends on the ***set of computational elements*** (i.e. : the family of “operators”) considered. Thus, given a set of computational elements, one can define the depth of an architecture as the depth of the graph involved to compute the output from the input. Formally it is the longest path from an input node to an output node.

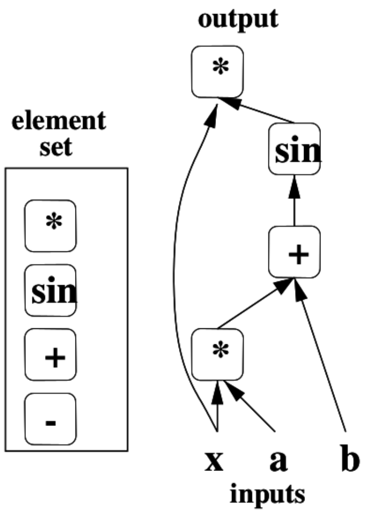


FIGURE 3.1 – Computation of the depth for $f(x) = x.sin(a.x+b)$ for the set of computational elements $(x, +, -, sin)$

For example, let us assess the depth of the representation of the function $f(x) = x.sin(a.x+b)$ given the set of computational elements $(x, +, -, sin)$. The graph of this computation is illustrated in Figure 3.1 and it has a depth of 4. Because the multiplication $a.x$ and the final multiplication by x are computed by two different layers.

For the most classic machine learning applications we have the following depth :

— ***Linear or logistic regression*** have a depth equal to 1 if the set of computational elements includes affine operations as well as their possible compositions with sigmoids.

— ***Kernel machines*** have a depth of 2 if we include a fixed kernel computation $K(\mathbf{u}, \mathbf{v})$ in the family of "operators". The first level has a single element performing the computation of $K(\mathbf{x_i}, \mathbf{c})$ for each training example $\mathbf{x_i}$ and matches the training example $\mathbf{x_i}$ with the input vector $\mathbf{x}$. Then the second level realizes an affine combination $b + \sum_i \alpha_i K(\mathbf{x}, \mathbf{x_i})$ to compute the expected output for an example $\mathbf{x_i}$.

— For ***Neural networks***, the ***depth is equal to the number of layers*** (excluded the input layer as it does not perform any computation) if we define the set of computational elements as the set of computations an artificial can neuron perform. For example a one hidden layer neural network will have, for this set of computational elements, a depth of 2 (the hidden and the output layer).

— ***Decision trees*** can be seen as having a depth of 2 (see [8] for explanations).

— ***Boosting*** adds one level of deepness to the base learner used [9].

As a comparison, according to the current research on brain anatomy [10], the visual cortex itself could be seen as a deep architecture of 5 to 10 levels.

So far, we have define depth as an extrinsic parameters of the learner conditioned by how we define the set of computational elements. But a graph associated with one set can, most of the time, be converted to the graph associated to another set by graph transformation in a way that multiplies the depth. Moreover, theoretical results suggests that ***it is not the absolute value of the depth that matters*** but the ***relative depth*** compared to the depth of a efficient representation of the approximated function with this set of computational elements.

3.2 Hidden Markov Models :

The advantages of deep architecture are of various natures. One can advocate in favor of a deep architecture using computational, statistical or even biological arguments.

3.2.1 Computational advantages :

The expression of a function given by a machine learning algorithm is said ***compact*** when it has only a few degrees of freedom to be tuned by learning (i.e. : a few computational elements). A compact representation of a function will be expected to yield to better generalization than a loose representation for a fixed number of training example. Indeed, the compact expression will have less parameters to be tuned during the learning phase and so is expected to reach a point in the optimization space closer to the actual optimum.

Thus the main idea used to defend deep architecture is that ***some function are simply too complicated to be compactly represented by a too shallow architecture***. In section 2.3 we have seen that a neural network of depth 2 (i.e. : with one hidden layer) is an universal approximator. Thus a network of this depth will be theoretically sufficient in most of the cases (e.g. : logical gates, sigmoid-neurons, Radial Basis Function 'RBF' units like in SVMs) to reach a targeted accuracy on a given problem. Having said this, the number of neurons required to perform the representation may be very large and we are very likely to obtain a loose representation of the objective function. Indeed some theoretical results for logical gates, formal neurons and RBF units have shown that, in the worst case scenario (i.e. : for specific function families) the required number of nodes to realize the compact representation may grow exponentially with the dimension of the input. For the RBF units, Hastad has even described families of functions which can be compactly represented with a linear number of nodes $O(n)$ for an input of dimension n and a network of depth d but required an exponential size $O(2^n)$ for the same input if the network is of a depth $d-1$ [11].

Those results are only proved for a specific family of function and does not proved that the functions to be learned in machine learning fall into this category. They also apply only to a specific kind of circuit and there is no proof for a generalized version of this property. Despite those gaps, this property is often use to advocate in favor of deep architecture [8, 35], as increasing the number of layers is expected to reduce the complexity of our network (i.e. : reduce the number of parameters to be handled) and thus decreasing the computational performances required.

3.2.2 Statistical advantages :

Using the same abusive generalization of the property demonstrated by Hastad for the RBF, one can also advocate in favor of a deep architecture using a statistical argument. Indeed, since the number of parameters that can be tuned/selected during the learning phase depends on the number of training examples available, the use of a loose representation of the objective function, with more free parameters, may lead to a poorer learning quality compared to a learning done on a compact representation on the same number of training examples. Thus a loose representation is expected to lead to poor generalization capacities.

One can see a deep architecture as a kind of factorization of the problem. A completely random function cannot be represented efficiently, whether with a deep or a shallow architecture. But when there is a logic to be learned, many that can be represented efficiently with a deep architecture cannot be represented efficiently with a shallow one (see the polynomials example in [8]). The existence of a compact and deep representation indicates that some kind of structure exists in the underlying function to be represented. If there was no structure whatsoever, it would not be possible to generalize well.

3.2.3 Biological advantages :

One of the goal in machine learning is to allow computers to model our world well enough to exhibit what is called “intelligence”. Many scientists are convinced that this can be achieved only by recreating a brain-like architecture for machine learning procedures. In that sense, deep networks seems very adapted as most of the brain scientists agree to say that the brain works using several successive layers of representation to process the information.

For example, the well-studied visual cortex shows a sequence of areas each of which contains a representation of the input, and signal flow from one to the next. There are also skip connections (a kind of natural dropout - see section **??**) and at some level parallel paths, so the procedure is still way more complex than what is currently done with deep networks. But the main idea remains. At each level of this feature hierarchy, a representation of the input at a different level of abstraction is created with more abstract features further up in the hierarchy, defined in terms of the lower-level ones.

Brain researchers have also noticed that representations in the brain are not dense, neither purely local. They are ***sparse***. For a given task, ***about* 1% *of neurons are active simultaneously in the brain***. Given the huge number of neurons, this is still a very efficient (exponentially efficient) representation. This observation can be use to advocate in favor of deep architecture with a sparse regularization (see section **??**) to mimic the brain behavior.

Finally the cognitive processes themselves seem deep. Indeed, humans organize their ideas and concepts hierarchically.They first learn simpler concepts and then compose them to represent more abstract ones. For example an engineer facing a problem will break-up the solutions into multiple sub-problems and sub-solutions (i.e. : multiple levels of abstraction and processing). And this is what an ideal deep architecture would do facing a problem.

4 Hidden Markov trees :

A ***Deep Belief Network*** is built by stacking together ***Restricted Boltzmann Machines***, a simplified version of the Boltzmann machines (BMs). Thus to explain the training procedure of the RBMs, we will first briefly introduce the Boltzmann machines, then the RBMs and see what are the simplification yielded by the "restriction" done. Finally we will describe the unsupervised training procedure that has to be used in the greedy layer wise training algorithm **??** for RBMs.

4.1 Model :

The Boltzmann machines [32] are a particular type of recurrent neural network with an energy-based model (i.e. : their dynamic is governed by energy functions). Specially, the probability density is given by the Boltzmann (i.e. : Gibbs) distributions. They are built out of I stochastic binary units x_i fully connected to each other, as illustrated in figure 4.1.

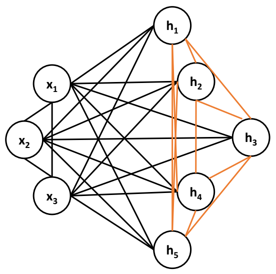


FIGURE 4.1 – A Boltzmann machine : The "input" layer and the "output" layer are fully connected

Each unit x_i, connected to the others with the weights $\mathbf{W}$, has a probability of activation given by the sigmoid function :

$$P(x_i = 1; \mathbf{W}) = \frac{1}{1 + \exp(-\sum_{j=0}^{I} w_{ij} x_j)} \tag{4.1}$$

Thus, the network provide a modeling of the probability of distribution of the input vectors $\mathbf{x}$ based on the Boltzmann/Gibbs distribution governed by the formula :

$$P(\mathbf{x}) = \frac{\exp(-E(\mathbf{x}))}{\sum_{\mathbf{x}} \exp(-E(\mathbf{x}))} \tag{4.2}$$

where the energy function $E(\mathbf{x})$ is given by :

$$E(\mathbf{x}) = -\sum_{i<j}(x_i w_{ij} x_j) \tag{4.3}$$

For a training set $\mathcal{D}_{train} = \{\mathbf{x}_k : k \in [1, |\mathcal{D}_{train}|]\}$, the learning seeks for the set of weights such as having a high average probability under the Boltzmann distribution for an input vector $\mathbf{x}_k$. For computational speed as well as accuracy, it is better to express the optimization of the probability in term of log-likelihood :

$$\begin{aligned}
-\log(P(\mathbf{x})) &= -\log\left(\frac{\exp(-E(\mathbf{x}))}{\sum_{\mathbf{x}}\exp(-E(\mathbf{x})}\right) \\
&= -\log(\exp(-E(\mathbf{x}))) + \log\left(\sum_{\mathbf{x}}\exp(-E(\mathbf{x}))\right)
\end{aligned} \tag{4.4}$$

Indeed, taking the partial derivative of this expression with respect to w_{ij} is simpler and we get :

$$\begin{aligned}
-\left\langle\frac{\partial\log(P(\mathbf{x})}{\partial w_{ij}}\right\rangle_{data} &= -\left\langle\frac{\partial\log(\exp(-E(\mathbf{x})))}{\partial w_{ij}}\right\rangle_{data} + \left\langle\frac{\partial\log\left(\sum_{\mathbf{x}}\exp(-E(\mathbf{x}))\right)}{\partial w_{ij}}\right\rangle_{model} \\
&= \left\langle\frac{\partial E(\mathbf{x})}{\partial w_{ij}}\right\rangle_{data} - \left\langle\frac{\partial E(\mathbf{x})}{\partial w_{ij}}\right\rangle_{model} \\
&= \langle x_i x_j\rangle_{data} - \langle x_i x_j\rangle_{model}
\end{aligned} \tag{4.5}$$

where $\langle.\rangle_{data}$ is the expectation of the input distribution, and $\langle.\rangle_{model}$ is the expectation of the stationary distribution of the model.

Using the simplified derivative expression obtained in 4.5, the weights can be updated using a gradient descent :

$$w_{ij} := w_{ij} + \epsilon\left(\langle x_i h_j\rangle_{data} - \langle x_i h_j\rangle_{model}\right) \tag{4.6}$$

But while, in general, the term $\langle x_i h_j\rangle_{data}$ can easily be computed by sampling from the data distribution, there is ***no efficient way to directly compute the term*** $\langle x_i h_j\rangle_{model}$. A estimate of the average over the sampled distribution $P(\mathbf{x})$ can be produce using a Markov Chain Monte Carlo approach to run the network to the equilibrium distribution. However, for a Boltzmann machine this process is very slow as it requires prolonged Gibbs sampling to explore the distribution $P(\mathbf{x})$ and reach the equilibrium state. Furthermore determining whether the distribution has reached an equilibrium state or not is challenging.

4.2 Learning : Expectation maximization :

The difficulties experienced while training a Boltzmann machine are -at least partly- due to the fully-connected architecture that they present (figure 4.1). Thus the RBMs [33] have been introduced with the objective to ease the sampling problem while still being able to model interesting distributions. They are a particular form of bipartite log-linear Markov Random Field (i.e. :random field for which the energy function is linear in its free parameters). The modeling power is ensure by the bipartite property which suppose that some of the variables are never observed (hidden units). The number of hidden units of the machine is directly linked to its modeling capacities, the more hidden units, the better modeling capacity of the Boltzmann machine will be. Finally the restriction of a BM to an RBM is done by forcing to have ***no connection within the layers*** (i.e. : no connection input-input nor hidden-hidden). The architecture of an RBM is presented figure 4.2.

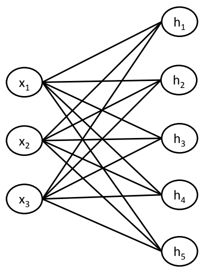


Figure 4.2 – A Restricted Boltzmann machine : Compared to a Boltzmann machines the restriction imposes no within layer connection (i.e. : no input-input nor output-output connection)

Now let us consider a binary RBM with a visible input layer $\mathbf{x}$ of dimension I (i.e. : size of the input vector) and an hidden layer $\mathbf{h}$ of dimension J (i.e. : size of the output vector) as shown figure 4.3. Added to that, there are also offsets (i.e. : bias) units, x_0 and h_0 permanently set to one. The layers are associated by an undirected weight matrix W, such that every input unit i is connected to every hidden variable j via the weight w_{ij}.

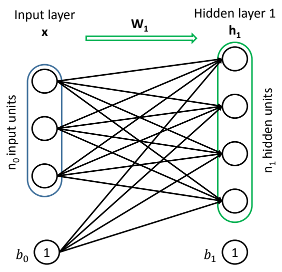


Figure 4.3 – The RBM for machine learning connects an input layer $\mathbf{x}$ to a hidden layer $\mathbf{h}$ via undirected weights W and biases x_0 and h_0

Because we have restricted the machine to a bipartite structure without intra-layer connections, the units of the hidden layer $\mathbf{h}$ are independent given the input layer $\mathbf{x}$. This independence simplify greatly the Gibbs sampling process compared to the Boltzmann machines. Thus, given an input vector, the activation probabilities of the hidden units of an RBM can be sampled by :

$$P(h_j = 1 \mid \mathbf{x}; \mathbf{W}) = \frac{1}{1 + \exp\left(-\sum_{i=0}^{I} w_{ij} x_i\right)} \tag{4.7}$$

Likewise, the input units can be sampled from the hidden vector using a symmetric decoder :

$$P(x_j = 1 \mid \mathbf{h}; \mathbf{W}) = \frac{1}{1 + \exp\left(-\sum_{j=0}^{I} w_{ij} h_j\right)} \tag{4.8}$$

In the case of a binary RBM, the energy of a the joint configuration $(\mathbf{x}, \mathbf{h})$ of activation states n the network defined in equation 4.3 becomes :

$$E(\mathbf{x}, \mathbf{h}) = -\sum_{i=0}^{I} \sum_{j=0}^{J} x_i w_{ij} h_j \tag{4.9}$$

Under the RBMs' assumptions, the joint probability distribution of states corresponding to the energy function $E(\mathbf{x}, \mathbf{h})$ is modeled by a machine as follow :

$$P(\mathbf{x}, \mathbf{h}) = \frac{\exp\left(-E(\mathbf{x}, \mathbf{h})\right)}{\sum_{\mathbf{x},\mathbf{h}} \exp\left(-E(\mathbf{x}, \mathbf{h})\right)} \tag{4.10}$$

Where the denominator $\sum_{\mathbf{x},\mathbf{h}} \exp\left(-E(\mathbf{x}, \mathbf{h})\right)$ is a normalization constant called the partition function. Following the maximum likelihood principle, the objective of the network is to maximize the marginal probability of the input vector $\mathbf{x}$ by summing over all possible vectors possible for the hidden layer $\mathbf{h}$:

$$P(\mathbf{x}) = \frac{\sum_{\mathbf{h}} \exp\left(-E(\mathbf{x}, \mathbf{h})\right)}{\sum_{\mathbf{x},\mathbf{h}} \exp\left(-E(\mathbf{x}, \mathbf{h})\right)} \tag{4.11}$$

When given a training set $\mathcal{D}_{train}$ of N input vectors $\{\mathbf{x}_k : k \in [1, N]\}$, the objective of the learning is to find the set of weight parameters $\mathbf{W}^*$ that minimize the average negative log-likelihood of the input data. Which leads to :

$$\begin{aligned} \mathbf{W}^* &= \arg\min_{\mathbf{W}} \; -\left\langle \log P(\mathbf{x}) \right\rangle_{data} \\ &= \arg\min_{\mathbf{W}} \; -\left(\left\langle \log \left(\sum_{\mathbf{h}} \exp(-E(\mathbf{x}, \mathbf{h})) \right) \right\rangle_{data} + \left\langle \log \left(\sum_{\mathbf{x},\mathbf{h}} \exp(-E(\mathbf{x}, \mathbf{h})) \right) \right\rangle_{model} \right) \end{aligned} \tag{4.12}$$

The partial derivative compute for the Boltzmann machines 4.5 still stand for the RBMs and we have the following expression for the weight updates :

$$w_{ij} := w_{ij} + \epsilon \left(\langle x_i h_j \rangle_{data} - \langle x_i h_j \rangle_{model} \right) \tag{4.13}$$

Where $\langle . \rangle_{dist}$ denotes the expectation under the distribution *dist*. The first term increases the probability of data driven activations that are clamped by the environment, while the second term reduces the probability of model driven states are sampled from the equilibrium distribution of a free running network. The weight update expressions are both the same for BMs 4.6 and the RBMs 4.13. However in the case of the RBMs there is an easy way to implement sampling strategy to compute an approximation of the term $\langle x_i h_j \rangle_{model}$.

4.3 Generation : Vitterbi algorithm :

The task of minimizing the average negative log-likelihood of the data distribution is equivalent to ***minimizing the Kullback-Leibler (KL) divergence*** [47] between the data distribution $P_0(\mathbf{x})$ (at time $t = 0$) and the equilibrium distribution $P_\infty(\mathbf{x})$ (at time $t = \infty$) :

$$D_{KL}(P_0 \| P_\infty) = \sum_{(\mathbf{x},\hat{\mathbf{y}})} P_0(\mathbf{x},\hat{\mathbf{y}}) \log \frac{P_0(\mathbf{x},\hat{\mathbf{y}})}{P_\infty(\mathbf{x},\hat{\mathbf{y}})} \geq 0 \tag{4.14}$$

Geoffrey Hinton [48] proposed the ***contrastive divergence learning algorithm***, that approximates the equilibrium distribution with a small finite number of sampling steps (Hinton's original paper use a number of step $N = 1$). The Markov chain is relaxed to run for N sampling steps to generate a reconstruction of the data vectors. An illustration of the sampling approximation is displayed in the figure 4.4. The optimization seeks to minimize the difference between the two divergences :

$$CD_N = D_{KL}(P_0 \| P_\infty) - D_{KL}(P_N \| P_\infty) \geq 0 \tag{4.15}$$

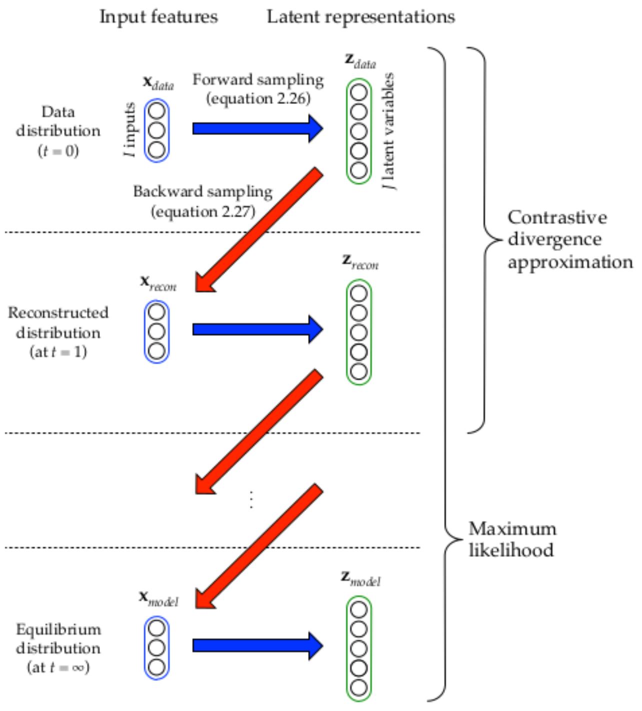

Input features
Latent representations
$\mathbf{x}_{data}$
Forward sampling (equation 2.26)
$\mathbf{z}_{data}$
Data distribution ($t = 0$)
I inputs
J latent variables
Backward sampling (equation 2.27)
Contrastive divergence approximation
$\mathbf{x}_{recon}$
$\mathbf{z}_{recon}$
Reconstructed distribution (at $t = 1$)
Maximum likelihood
$\mathbf{x}_{model}$
$\mathbf{z}_{model}$
Equilibrium distribution (at $t = \infty$)

FIGURE 4.4 – Gibbs sampling relaxation of contrastive divergence learning. The maximum likelihood of the data distribution can be obtained by performing alternating (forward-backward) Gibbs sampling between the input layer x and latent layer h, from the data distribution samples (at $t = 0$) until $t = \infty$. The contrastive divergence approximates the equilibrium distribution using reconstructed samples from a small finite number of sampling steps. One sampling step (N = 1) is used in this example.

The partial derivative with respect to the parameter w_{ij} is :

$$\begin{aligned}\frac{\partial CD_N}{\partial w_{ij}} = \frac{\partial D_{KL}(P_0 \| P_\infty)}{\partial w_{ij}} - \frac{\partial D_{KL}(P_N \| P_\infty)}{\partial w_{ij}} \\ \approx -\left\langle \frac{\partial \log \sum_x \exp\left(-E(\mathbf{x},\mathbf{h})\right)}{\partial w_{ij}} \right\rangle_{data} + \left\langle \frac{\partial \log \sum_x \exp\left(-E(\mathbf{x},\mathbf{h})\right)}{\partial w_{ij}} \right\rangle_{recon} \\ \approx \left\langle \frac{\partial E(\mathbf{x},\mathbf{h})}{\partial w_{ij}} \right\rangle_{data} - \left\langle \frac{\partial E(\mathbf{x},\mathbf{h})}{\partial w_{ij}} \right\rangle_{recon} \\ \approx \langle x_i h_j \rangle_{data} - \langle x_i h_j \rangle_{recon}\end{aligned} \tag{4.16}$$

Where $\langle . \rangle_{recon}$ is the expectation of the reconstructed states after N sampling steps. Using those partial derivative, one can approximates the gradient descent by :

$$w_{ij} := w_{ij} + \epsilon \left(\langle x_i h_j \rangle_{data} - \langle x_i h_j \rangle_{recon} \right) \tag{4.17}$$

Where the energy of samples from the data distribution is decreased, while raising the energy of reconstructed states that the network prefers to real data.

The algorithm 3 presents the iterative procedure for training an RBM with N Gibbs sampling steps followed by the updates of the parameters.

```
1) Initialize $\mathbf{W}$
2) while no convergence : do
    Get $\mathbf{x}_0$ from randomized training batches.
    Sample $P_0(\mathbf{h_0} \mid \mathbf{x_0})$ using 4.7 3) for $n = 1 to N$ do
        Sample $P_n(\mathbf{x_n} \mid \mathbf{h_{n-1}})$ using 4.8
        Sample $P_n(\mathbf{h_n} \mid \mathbf{x_n})$ using 4.8
    end
    Update $w_{ij} := w_{ij} + \Delta w_{ij}$ using 4.17
end
```

Algorithm 3: RBM training with N steps contrastive divergence.

CD_1 is the fastest training procedure for RBMs has it requires the minimal number of sampling steps. Most of the time, it provides a good approximate of the distribution. Although a greater number of sampling steps will produce an estimate closer to the true likelihood of the gradient, it will need more computational time and may results in high estimator variance [49].

5 Scattering hidden Markov tree :

The ***Stack Auto-Encoders*** (SAEs) are another type of deep neural architecture exploiting the ***Auto-Encoders*** (AEs) as building blocks. As we will see in the following section 5.2, the training procedure for AEs is easier to implement than the RBMs (section 4.3) one. So they have also been used as building blocks to train deep neural networks, where each level is associated with an auto-encoder that can be trained separately [12, 19, 42, 51].

5.1 Hypothesis :

An auto-encoder is a multi-layer perceptron (section 2.3) with one hidden layer trained to encode an input $\mathbf{x}$ into some representation $\mathbf{h}(\mathbf{x})$ so that the input can be reconstructed (i.e. : decoded) from that representation to produce the output layer.

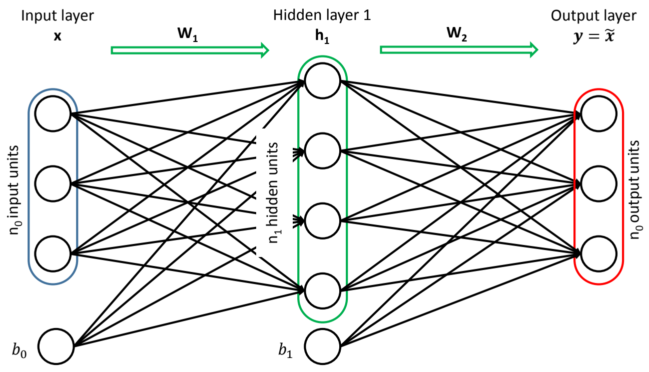


FIGURE 5.1 – An auto-encoder : Maps an input $\mathbf{x}$ to a representation $\mathbf{h_1}$ with the objective to compute the best reconstruction $\hat{\mathbf{y}} = \tilde{\mathbf{x}}$ for the output layer of the network.

Given a train set $\mathcal{D}_{train}$ of m elements living in $[0,1]^d$, an auto-encoder takes an input $\mathbf{x}$ and maps it ***deterministically***, via an encoder, to an hidden representation $\mathbf{h}(\mathbf{x}) \in [0,1]^{d'}$ following the relation :

$$\mathbf{h} = \sigma_1(W_1.\mathbf{x} + b_1) = \sigma_1(W_1, b_1, \mathbf{x}) = a_1(\mathbf{x}) \tag{5.1}$$

Where σ is the activation function for the hidden unit (section 2.1). The latent representation (i.e. : code) $\hat{\mathbf{y}}$ is then mapped back, via a decoder, into a reconstruction $\hat{\mathbf{y}}$ of the same shape as $\mathbf{x}$ by a similar transformation :

$$\hat{\mathbf{y}} = \sigma_2(W_2\mathbf{h} + b_2) = \sigma_2(W_2, b_2, \mathbf{h}) = a_2(\mathbf{h}) \tag{5.2}$$

Thus, $\hat{\mathbf{y}}$ is the prediction of $\mathbf{x}$ given the code $\mathbf{h}$ and we have :

$$\hat{\mathbf{y}} = \sigma_2(W_2, b_2, \sigma_1(W_1, b_1, \mathbf{x})) = a_2(a_1(\mathbf{x})) \tag{5.3}$$

The auto-encoder tries to learn a function $f(W_1, b_1, W_2, b_2, \mathbf{x}) \approx \mathbf{x}$. In other word, it is trying to learn an approximation of the identity function so that the output $\hat{\mathbf{y}} = \tilde{\mathbf{x}}$ is similar to $\mathbf{x}$. Stated like this, one could argue that the identity function appears to be a particularly trivial function to be learned. But by ***placing constraints on the network***, such as reducing the number of neurons for the encoded representation of the inputs (i.e. : number of neurons in the hidden layer), interesting representation can be learned. For example, suppose that the training data are a set of 28 x 28 gray-scale images (784 pixels/image) described by the pixel intensity. Hence, the input layer $L^{(0)}$ of the auto-encoder has $n^{(0)} = 784$ units and suppose we are trying to encode it using a hidden layer $L^{(1)}$ with $n^{(1)} = 196$ units. Note that the AE's properties imposed to have an output layer $L^{(2)}$ with $n^{(3)} = 784$. Since the number of units available to learn the representation of the input is four times smaller than the input size, the network is forced to learn a compressed representation of the input (i.e. : given the vector of hidden units $h(x) \in \mathbb{R}^{196}$, the network has to reconstruct the 784-pixel input x). In the case of random inputs, this compression task is very difficult. But if there is a structure in the data (e.g. : correlated features...), then the AE will be able to learn -at least part of- the structure to perform the compression. In fact, the simple auto-encoder used in this example often ends up learning a low-dimensional representation very similar to PCA's [35].

Even if the above argument relied on the number of neurons $n^{(1)}$ in the hidden unit to be smaller than the number of input neurons $n^{(0)}$, it is still possible to discover interesting structure in the data with a network with more hidden units than input units by imposing other constraints on the auto-encoder (chapter **??** for more details about the regularization methods).

5.2 Results and application :

The goal being to recreate the input, the parameters $\mathbf{W_1}, \mathbf{W_2}, \mathbf{b_1}, \mathbf{b_2}$ of the auto-encoder are optimized in order to minimize the average reconstruction error J. The definition of J can vary depending on the problem assessed :

— ***Cross-entropy :*** When the inputs are interpreted as bit vectors or as vectors of bit probabilities it is better to use the cross-entropy function to measure the cost of the reconstruction. Its expression becomes for an auto-encoder trained on the dataset $\mathcal{D}_{train}$ define earlier in this chapter :

$$\begin{aligned} J(W_1, W_2, b_1, b_2) &= -\frac{1}{m}\sum_{i=0}^{m-1}\sum_{k=1}^{d}(x_{ik}\log\hat{y}_{ik} + (1 - x_{ik})\log(1 - \hat{y}_{ik})) \\ &= -\frac{1}{m}\sum_{i=0}^{m-1}\sum_{k=1}^{d}(x_{ik}\log(\sigma_2(W_2, b_2, \sigma_1(W_1, b_1, x_{ik}))) \\ &\qquad + (1 - x_{ik})\log(1 - \sigma_2(W_2, b_2, \sigma_1(W_1, b_1, x_{ik})))) \end{aligned} \tag{5.4}$$

— ***Squared error :*** The squared error is used for the classification or regression problems.

$$\begin{aligned} J(W_1, W_2, b_1, b_2) &= -\frac{1}{m}\sum_{i=0}^{m-1}\left(\|\mathbf{x_i} - \hat{\mathbf{y}}_\mathbf{i}\|^2\right) \\ &= -\frac{1}{m}\sum_{i=0}^{m-1}\left(\|\mathbf{x_i} - \sigma_2(W_2, b_2, \sigma_1(W_1, b_1, \mathbf{x_i}))\|^2\right) \end{aligned} \tag{5.5}$$

— ***Personalize cost :*** Another possibility is to define a metric tailored to the specific problem assessed. This metrics has to admit derivatives in order to allow its optimization.

The auto-encoder is then trained using the backpropagation algorithm described in the section 2.4.

6 Experimental results :

The aim of this internship was to implement our own deep neural networks and then test it on several datasets. We first decided to test them on MNIST(section 6.1 - [7]) as it is one of the most widely used dataset in the literature on deep neural network. Then we assessed the performances of a deep neural network used as a feature generator in a Kaggle competition (section 6.2 - [43]).

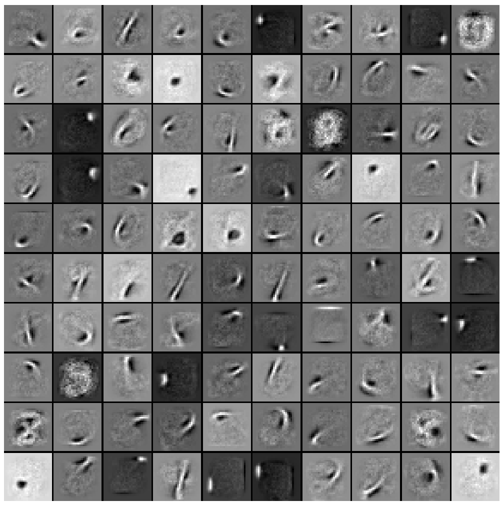
6.1 MNIST :

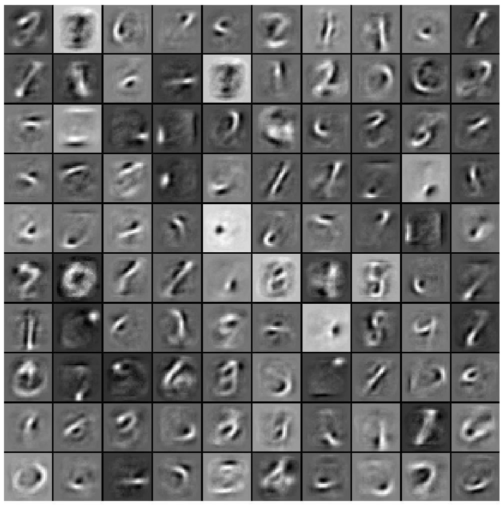
MNIST is a dataset of labeled ***handwritten gray scale digits***. It has a training set of 60 000 examples, and a test set of 10 000 examples. The input images have 28×28 pixels (i.e. : 784 dimensions) an a small pre-processing have been done, as the digits are size-normalized, centered in a fixed-size image and the pixel values have been shifted between 0 and 1.

The advantage with an image dataset is the possibility to plot the representations learned by the network in order to have an idea of the quality of the features learned. But this method is not 100% accurate as what a human may considerate as a “good” intermediate representation may not be the best one for a computer. Despite this, we used this visual argument as well as the classification error to assess the performances of the stack auto-encoders on MNIST.

As the parameter space to be tested is really rich, we decided to restrain to a few number of architectures for the network (3 hidden layers of 500 units, 3 hidden layers of 1 000 units and 4 layers with respectively 1 000, 500, 100 and 20 units). So far our best classification error on the validation set is around 1.1% which is good but still far above the current state of the art for deep neural networks [7].

MNIST was also useful to grasp the influence of each meta-parameters of the learning. We studied different learning rates, different regularizations. We also implemented the ***second order gradient descent algorithm L-BFGS*** to try to improve the quality of the learning and so improve our results. Some examples of the representations learned can be seen on the following figures.


(a) 100 randomly selected sub-representations in the first hidden layer


(b) 100 randomly selected sub-representations in the second hidden layer

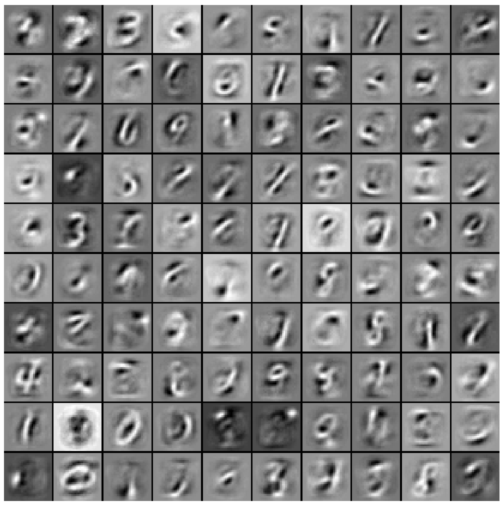
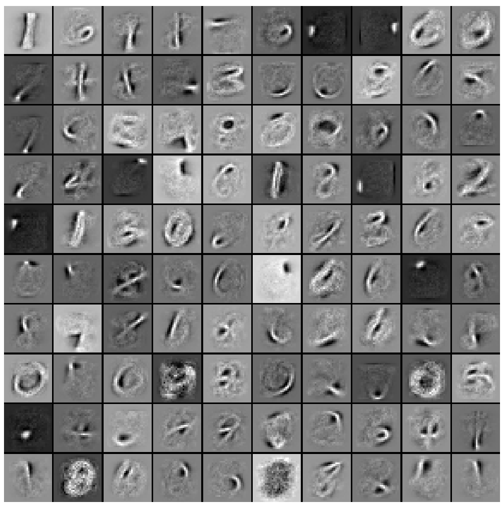
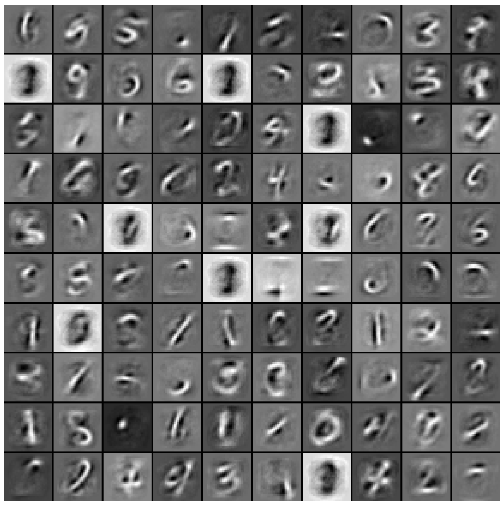

(c) 100 randomly selected sub-representations in the third hidden layer

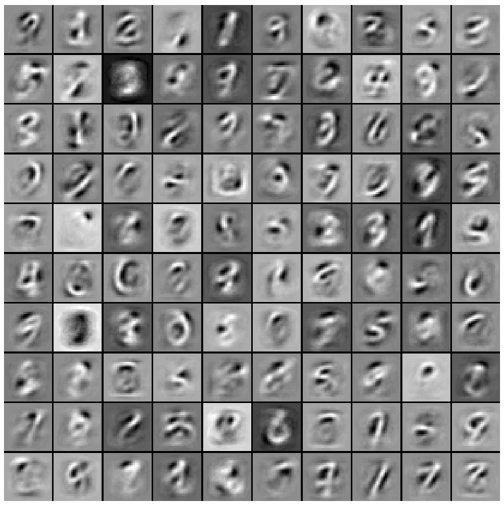
FIGURE 6.1 – A 3 hidden layers 500-500-500 stack auto-encoder with the contractive penalization weighted to $\delta = 0.1$, the sparsity to $\beta = 0.1$ and $\rho = 0.1$, the weight decay to $\lambda = 0.0001$ and a dropout rate of 40% and the auto-encoders' weights are tied. The cost function is the cross-entropy.



(a) 100 randomly selected sub-representations in the first hidden layer



(b) 100 randomly selected sub-representations in the second hidden layer



(c) 100 randomly selected sub-representations in the third hidden layer

FIGURE 6.2 – A 3 hidden layers 1000-1000-1000 stack auto-encoder with the contractive penalization weighted to $\delta = 0.1$, the sparsity to $\beta = 0.1$ and $\rho = 0.1$, the weight decay to $\lambda = 0.0001$ and a dropout rate of 40% and the auto-encoders' weights are tied. The cost function is the cross-entropy.

The two experiments whose results are introduced in figure 6.1 and 6.2, use the same regularization parameters but the second one have a network with twice as many hidden units. We visualize the weights connecting the layer to the previous one projected back into the input space. The first experiences is a stack auto-encoder with 3 hidden layers with 500 units each. As expected the deep network first learns low level features (figure 6.1a : small parts of number, edges) and move to higher level features (figure 6.1c : some numbers can be guessed). Even if the same general behavior can be witnessed for the second experiment (figure 6.2) involving this time a stack auto-encoder with 3 hidden layers with 1 000 units each, one can notice that some of the unit display unidentified shapes (6.2b : second row column 1 and 5 for example). This could be a sign that this network is too big to learn the ideal representation or that the regularization applied is not strong enough.

6.2 Kaggle higgs competition :

Another goal of the internship was to assess the performances of the deep neural networks for exploratory data analyses. To do so we took a data science-like problem, where we have few or no idea on what is the meaning of each feature of the input vector. Our choice has been the dataset provided provided for the ***Kaggle Higgs Boson Challenge*** [43]. The contest is sponsored by the CERN and aim at analyzing the data collected in the Large Hadron Collider. The contestants are asked to classified the events (i.e. : instance of the dataset) between the "signal" and "noise".

The usual procedure to tackle this kind of problems would be to manually transform the given features in order to find a more suitable representation space to perform the classification. But beside being time consuming, this is hard to do when, as here, the input's physical meanings are not easy to understand. However as we have seen in the section 3.2, the deep neural networks should allow to circumvent this issue by performing an unsupervised feature generation.

Hence, the idea is to realize the unsupervised training of a stack auto-encoder on the train set and then use the representation of the dataset learned as an input for a classifier. In order to assess the gain in performances provide by the unsupervised feature generation we will compare to the same classifier trained on the raw input data. Unfortunately so far the results are not very good. Mainly because we have no other way to select an architecture for the network but to try it and see to what performances it yield to, which is a slow process.

7 Conclusion :

Scattering hidden Markov tree : The stack auto-encoders and more generally the deep neural networks are a very powerful tool as they allow to realize machine learning tasks with virtually no human prior or even intervention. However, we have seen during our experiments how hard to optimize where those deep architectures. Indeed the optimization process takes place at two very different levels.

Given a fix architecture, making sure to find the network's parameters minimizing the error is still an open issue as the optimization problem to be solved is non-linear and involves many free parameters. Even if the unsupervised pre-training problem is supposed to be easier to solve, the ***effective minimization is still not certain*** (see [45] for more information on this topic).

Another issue in the learning procedure is to control the over-fitting. Many regularization methods are available (see chapter **??**) and provide encouraging results. However, they also make the optimization problem tackled during the training phase harder to be solved. Thus ***a better understanding of the regularization for deep neural networks is required*** in order to adapt the amount of regularization to the problem.

Finally, there is also an open question on ***how to find the best architecture for the network given a problem***. Today the most widely used method remains a greed search on both the number of layers and the number of units. But such a process is time consuming, not very precise nor efficient. Many researchers are convinced that the next evolution in neural network is to create networks with an adaptive architecture (see the "next steps" section of [8] for more information on the topic).

Next steps : This internship at IPAL has been very interesting. It has been a good opportunity to learn a lot more about machine learning in general, to acquire a good knowledge of the deep architecture and to apply the knowledge acquired from both Supélec and the internship to real world problems.

It was also the perfect opportunity to discover and confirm my taste for the research environment.

8 Acknowledgements :

First, I would like to thank my two supervisors Antoine and Oliver at IPAL for their advices as well as their support. I would also like to express my gratitude to Hanlin Guo for sharing his knowledge of deep learning and for all the tips he gave me.

Second, this internship would not have be the same without the friendly atmosphere of the laboratory. That is why I would like to thanks all the other IPAL's interns as well as the PHD candidates.

Lastly, I would like to thank the IPAL laboratory for the high quality infrastructures and hardwares provided during the internship.

Bibliographie

[1] F. Rosenblatt : *The perceptron : A probabilistic model for information storage and organization in the brain*
Psychological Review, 65 :386 - 408, 1958.

[2] Y. LeCun : *Une procédure d'apprentissage pour réseau à seuil asymmetrique* (A learning scheme for asymmetric threshold networks)
In Cognitiva 85, pages 599 – 604, Paris, France. 1985.

[3] D. E. Rumelhart,G. E. Hinton and R. Williams : *Learning representations by back-propagating errors*
Nature, 323 :533 – 536, 1986.

[4] W. McCulloch and W. Pitts : *A logical calculus of the ideas immanent in nervous activity*
Bulletin of Mathematical Biophysics, 5 :115 – 133, 1943.

[5] G. Cybenko : *Approximations by superpositions of sigmoidal functions*
Mathematics of Control, Signals, and Systems, 2(4) :303 – 314, 1989.

[6] K. Hornik : *Approximation capabilities of multilayer feedforward networks*
Neural Networks, 4(2) :251–257, 1991.

[7] Y. LeCun : *Personal web page gathering the results of various algorithms on the MNIST classification problem*
`http://yann.lecun.com/exdb/mnist/`

[8] Y. Bengio : *Learning Deep Architecture for AI*
Fondation and Trends in Machine Learning Vol. 2, No. 1 1 - 127, 2009.

[9] Y. Freund and R. E. Schapire : *Experiments with a new boosting algorithm*
Machine Learning : Proceedings of Thirteenth International Conference, p.148 - 156, USA : ACM, 1996.

[10] T. Serre, G. Kreiman, M. Kouh, C. Cadieu, U. Knoblich and T.Poggio : *A quantitative theory of immediate visual recognition*
Progress in Brain Research, Computational Neuroscience : Theoretical Insights into Brain Function, vol. 165, p.33 – 56, 2007.

[11] J. Hastad : *Almost Optimal Lower Bounds for Small Depth Circuits*
STOC, 1986.

[12] Y. Bengio, P. Lamblin, D. Popovici and H. Larochelle : *Greedy layer-wise training of deep networks*
Advances in Neural Information Processing Systems 19 (NIPS'06), (B. Schlkopf, J. Platt, and T. Hoffman, eds.), p.153 – 160, MIT Press, 2007.

[13] S. Geman and D. Geman : *Stochastic relaxation, gibbs distributions, and the Bayesian restoration of images*
IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 6, p.721 – 741, November 1984.

[14] H. Larochelle, Y. Bengio, J. Louradour and P. Lamblin : *Exploring strategies for training deep neural networks*
Journal of Machine Learning Research, vol. 10, p.1 – 40, 2009.

[15] D. Erhan, P-A. Manzagol, Y. Bengio, S. Bengio and P. Vincent : *The difficulty of training deep architectures and the effect of unsupervised pre-training*
International Conference on Artificial Intelligence and Statistics (AISTATS), p.153 – 160, 2009

[16] A. Ahmed, K. Yu, W. Xu, Y. Gong and E. P. Xing : *Training hierarchical feed-forward visual recognition models using transfer learning from pseudo tasks*
Proceedings of the 10th European Conference on Computer Vision (ECCV'08), p.69 – 82, 2008.

[17] H. Larochelle, D. Erhan, A. Courville, J. Bergstra and Y. Bengio : *An empirical evaluation of deep architectures on problems with many factors of variation*
Proceedings of the Twenty-fourth International Conferenceon Machine Learning (ICML'07), (Z. Ghahramani, ed.), p.473 – 480, ACM, 2007.

[18] M. Ranzato, Y.-L. Boureau and Y. LeCun : *Sparse feature learning for deep belief networks*
Advances in Neural Information Processing Systems 20 (NIPS'07), (J. Platt, D. Koller, Y. Singer and S. Roweis, eds.), p. 1185 – 1192, Cambridge, MA : MIT Press, 2008.

[19] P. Vincent, H. Larochelle, Y. Bengip and P.-A. Manzagol : *Extracting and composing robust features with denoising autoencoders*
Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08), (W. W. Cohen, A. McCallum, and S. T. Roweis, eds.), p. 1096 – 1103, ACM, 2008.

[20] R. Salakhutdinov and G. E. Hinton : *Deep Boltzmann machines*
Proceedings of The Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS'09), vol. 5, p. 448 – 455, 2009.

[21] S. Osindero and G. E. Hinton : *Modeling image patches with a directed hierarchy of Markov random field*
Advances in Neural Information Processing Systems 20 (NIPS'07), (J. Platt, D. Koller, Y. Singer, and S. Roweis, eds.), p.1121 – 1128, Cambridge, MA : MIT Press, 2008.

[22] R. Salakhutdinov and G. E. Hinton : *Learning a nonlinear embedding by preserving class neighbourhood structure*
Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics (AISTATS'07), San Juan, Porto Rico : Omnipress, 2007.

[23] G. E. Hinton and R. Salakhutdinov : *Reducing the dimensionality of data with neural networks*
Science, vol. 313, no. 5786, p.504 – 507, 2006.

*[24] G. E. Hinton, S. Osindero and Y.-M. Teh : [*A fast learning algorithm for deep belief networks
Neural Computation, 18(7) :1527 – 1, 2006.

[25] R. Hadsell, A. Erkan, P. Sermanet, M. Scoffier, U. Muller, Y. LeCun : Deep belief net learning in a long-range vision system for autonomous off-road driving
Proc. Intelligent Robots and Systems (IROS'08), p.628 – 633, 2008.

[26] R. Collobert and J. Weston : A unified architecture for natural language processing : Deep neural networks with multitask learning
Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08), (W. W. Cohen, A. McCallum, and S. T. Roweis, eds.), p.160 – 167, ACM, 2008.

[27] A. Mnih and G. E. Hinton : A scalable hierarchical distributed language model *Advances in Neural Information Processing Systems 21 (NIPS'08), (D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, eds.), p.1081 – 1088, 2009.*

[28] M. Ranzato, C. Poultney, S. Chopra and Y. LeCun : Efficient learning of sparse representations with an energy-based model
Advances in Neural Information Processing Systems (NIPS), p.1137 – 1144, 2006.

[29] Y. Bengio, P. Lamblin, D. Popovici and H.Larochelle : *Greedy layer-wise training of deep networks*
Advances in Neural Information Processing Systems (NIPS), 2006.

[30] G. E. Hinton : *To recognize shapes, first learn to generate images*
Paul Cisek, T. D. and Kalaska, J. F., editors, Computational Neuroscience : Theoretical Insights into Brain Function, volume 165 of Progress in Brain Research, p.535 – 547. Elsevier, 2007.

[31] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio : *Why does unsupervised pre-training help deep learning ?*
Journal of Machine Learning Research, 11 :625 – 660, 2010

[32] G. E. Hinton and T. Sejnowski : *Learning and relearning in Boltzmann machines*
D. E. Rumelhart, J. L. McClelland and the PDP Research Group, editors, Parallel Distributed Processing : Volume 1 : Foundations, chapter 7, p.282 – 317. MIT Press, Cambridge, 1986.

[33] P. Smolensky : *Information processing in dynamical systems : Foundations of harmony theory*
D. E. Rumelhart, J. L. McClelland and the PDP Research Group, editors, Parallel Distributed Processing : Volume 1 : Foundations, chapter 6, p.194 – 281. MIT Press, Cambridge, 1986.

[34] S. Geman, E. Bienenstock and R. Doursat : *Neural networks and the bias/variance dilemma*
Neural Computation, Volume 4, p.67 - 79, 1992.

[35] A. Ng : *Sparse autoencoder*
CS294A Lecture notes.

[36] S. Rifai, P. Vincent, X. Muller, X. Glorot and Y. Bengio : *Contractive Auto-Encoders : Explicit Invariance During Feature Extraction*
Proceedings of the 28th International Conference on Machine Learning (ICML-11), 2011.

[37] K. Matsuoka : *Noise injection into inputs in back-propagation learning*
Systems, Man and Cybernetics, IEEE Transactions on, 22(3), p.436 – 440, 1992.

[38] C. M. Bishop : *Training with noise is equivalent to Tikhonov regularization* Neural computation, 7(1), p.108 – 116, 1995.

[39] S. Rifai, X. Glorot, Y. Bengio and P. Vincent : *Adding noise to the input of a model trained with a regularized objective*
arXiv preprint arXiv : 1104.3250, 2011.

[40] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever and R. R. Salakhutdinov : *Improving neural networks by preventing co-adaptation of feature detectors*
arXiv :1207.0580v1 [cs.NE], 2012

[41] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley and Y. Bengio : *Theano : A CPU and GPU Math Expression Compiler*
Proceedings of the Python for Scientific Computing Conference (SciPy) 2010. June 30 - July 3, Austin, TX.

[42] H. Larochelle, D. Erhan, A. Courville, J. Bergstra and Y. Bengio : *An empirical evaluation of deep architectures on problems with many factors of variation*
Proceedings of the Twenty-fourth International Conference on Machine Learning (ICML'07), (Z. Ghahramani, ed.), p.473 – 480, ACM, 2007.

[43] `https://www.kaggle.com/c/higgs-boson`

[44] D. West *Neural network credit scoring models*
Computers & Operations Research 27.11 (2000) : 1131-1152.

[45] G. Xavier and Y Bengio : *Understanding the difficulty of training deep feedforward neural networks*
International Conference on Artificial Intelligence and Statistics. 2010.

[46] Saxe, Andrew M., James L. McClelland and Surya Ganguli : *Learning hierarchical category structure in deep neural networks*
Proceedings of the 35th Annual Conference of the Cognitive Science Society. 2013.

[47] S. Kullback and R. Leibler : *On information and sufficiency*
Annals of Mathematical Statistics, 22(1) :p.79 – 86, 1951.

[48] G. E. Hinton : *Training products of experts by minimizing contrastive divergence*
Neural Computation, 14(8) : p.1771 – 1800, 2002.

[49] G. Hanlin *Deep Learning and Visual Representations*
Thesis review, 2013.

[50] N. Le Roux and Y. Bengio : *Representational power of restricted Boltzmann machines and deep belief networks*
Neural Computation, 20(6) : p.1631 – 1649, 2008.

[51] M. Ranzato, Y. Boureau, S. Chopra, Y. LeCun : *A unified energy-based framework for unsupervised learning*
Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics (AISTATS'07), San Juan, Porto Rico : Omnipress, 2007.