ALMA MATER STUDIORUM
UNIVERSITY OF BOLOGNA

# Deep Learning for Computer Vision: A comparison between Convolutional Neural Networks and Hierarchical Temporal Memories on object recognition tasks

*Candidate:*
dott. Vincenzo Lomonaco

*Supervisor:*
prof. Davide Maltoni

School of Science
Master Degree in Computer Science

*Academic year 2014-15*
*Session II*

September 2015

*"We may regard the present state of the universe as the effect of the past and the cause of the future. An intellect which at any given moment knew all of the forces that animate nature and the mutual positions of the beings that compose it, if this intellect were vast enough to submit the data to analysis, could condense into a single formula the movement of the greatest bodies of the universe and that of the lightest atom; for such an intellect nothing could be uncertain and the future just like the past would be present before its eyes."*

Pierre Simon Laplace, *A Philosophical Essay on Probabilities*, 1814

ALMA MATHER STUDIORUM
UNIVERSITÀ DI BOLOGNA

# *Sommario*

Scuola di Scienze
Laurea Magistrale in Informatica

**Deep Learning for Computer Vision:
A comparison between Convolutional Neural Networks and Hierarchical Temporal Memories on object recognition tasks**

dott. Vincenzo Lomonaco

Negli ultimi anni, le tecniche di *Deep Learning* si sono dimostrate particolarmente utili ed efficaci nel risolvimento di una grande varietà di problemi, sia nel contesto della visione artificiale che in quello dell' elaborazione del linguaggio naturale, raggiungendo e spesso superando lo stato dell'arte [1] [2] [3]. Il successo del deep learning sta rivoluzionando l'intero campo dell' apprendimento automatico e del riconoscimento di forme avvalendosi di concetti importanti come l' estrazione automatica delle caratteristiche ed apprendimento non supervisionato [4].

Tuttavia, nonostante il grande successo raggiunto sia in ambiti accademici che industriali, anche il deep learning ha cominciato a mostrare delle limitazioni intrinseche. Infatti, la comunità scientifica si domanda se queste tecniche costituiscano solo un sorta di approccio statistico a forza bruta e se possano operare esclusivamente nell'ambito dell' *High Performance Computing* con un enorme quantità di dati [5] [6]. Un' altra questione importante riguarda la possibilità di comprendere quanto questi algoritmi siano biologicamente ispirati e se possano scalare bene in termini di "intelligenza".

L' elaborato si focalizza sul tentativo di fornire nuovi spunti per la risoluzione di questi quesiti chiave nel contesto della visione artificiale e più specificatamente del riconoscimento di oggetti, un compito che è stato completamente rivoluzionato dai recenti sviluppi nel campo.

Dal punto di vista pratico, nuovi spunti potranno emergere sulla base di un' esaustiva comparazione di due algoritmi di deep learning molto differenti tra loro: *Convolutional Neural Network* (CNN) [7] e *Hierarchical Temporal memory* (HTM) [8]. Questi due algoritmi rappresentano due approcci molto differenti seppur all'interno della grande famiglia del deep learning, e costituiscono la scelta migliore per comprendere appieno punti di forza e debolezza reciproci.

L' algoritmo CNN è considerato uno dei metodi supervisionati più potente usato oggi per l'apprendimento automatico e specialmente per il riconoscimento di oggetti. Le reti a convoluzione sono state ben recepite ed accettate dalla comunità scientifica e, ad oggi, sono già adoperate in grandi industrie tecnologiche del calibro di *Google* e *Facebook* per problemi come il riconoscimento del volto [9] e l' auto-tagging di immagini [10].

L'algoritmo HTM, invece, è principalmente conosciuto come un nuovo ed emergente paradigma computazionale biologicamente ispirato che si basa principalmente su tecniche di apprendimento non supervisionate. Esso cerca di integrare più indizi dalla comunità scientifica della neuroscienza computazionale per incorporare concetti come il *tempo*, il *contesto* e l' *attenzione* nei processi di apprendimento che sono tipici del cervello umano.

In ultima analisi, la tesi si presuppone di dimostrare che in certi casi, con una quantità inferiore di dati, L' algoritmo HTM può risultare vantaggioso rispetto a quello CNN [11].

ALMA MATHER STUDIORUM
UNIVERSITY OF BOLOGNA

# *Abstract*

School of Science
Master Degree in Computer Science

**Deep Learning for Computer Vision:
A comparison between Convolutional Neural Networks and Hierarchical Temporal Memories on object recognition tasks**

dott. Vincenzo Lomonaco

In recent years, *Deep Learning* techniques have shown to perform well on a large variety of problems both in *Computer Vision* and *Natural Language Processing*, reaching and often surpassing the state of the art on many tasks [1] [2] [3]. The rise of deep learning is also revolutionizing the entire field of *Machine Learning* and *Pattern Recognition* pushing forward the concepts of automatic feature extraction and unsupervised learning in general [4].

However, despite the strong success both in science and business, deep learning has its own limitations. It is often questioned if such techniques are only some kind of brute-force statistical approaches and if they can only work in the context of *High Performance Computing* with tons of data [5] [6]. Another important question is whether they are really biologically inspired, as claimed in certain cases, and if they can scale well in terms of "intelligence".

The dissertation is focused on trying to answer these key questions in the context of *Computer Vision* and, in particular, *Object Recognition*, a task that has been heavily revolutionized by recent advances in the field.

Practically speaking, these answers are based on an exhaustive comparison between two, very different, deep learning techniques on the aforementioned task: *Convolutional Neural Network* (CNN) [7] and *Hierarchical Temporal memory* (HTM) [8]. They stand for two different approaches and points of view within the big hat of deep learning and are the best choices to understand and point out strengths and weaknesses of each of them.

CNN is considered one of the most classic and powerful supervised methods used today in machine learning and pattern recognition, especially in object recognition. CNNs

are well received and accepted by the scientific community and are already deployed in large corporation like *Google* and *Facebook* for solving face recognition [9] and image auto-tagging problems [10].

HTM, on the other hand, is known as a new emerging paradigm and a new meanly-unsupervised method, that is more biologically inspired. It tries to gain more insights from the *computational neuroscience* community in order to incorporate concepts like *time*, *context* and *attention* during the learning process which are typical of the human brain.

In the end, the thesis is supposed to prove that in certain cases, with a lower quantity of data, HTM can outperform CNN [11].

# *Acknowledgements*

First of all, I would like to thank my supervisor, prof. Davide Maltoni for helping me go through the entire process of the dissertation development, answering thousands of e-mails, giving me incredibly useful insights and introducing me to the huge field of deep learning.
I would also like to express my deep appreciation to all the people who assisted me with this complex project during my graduated years:
My family, who has never constrained me and has always been present in times of need.
Marina Foti, who filled me with enthusiasm, motivation and love even in the hardest moments.
Giovanni Lomonaco, for helping me to develop my ideas on brain, learning and artificial intelligence.
Pierpaolo Del Coco, Ivan Heibi, Antonello Antonacci and many other fellow students, for helping me to develop my current skills through many years of projects, discussions, exams and fun.
Andrea Motisi, Ferdinando Termini and all the people from my student house, for giving me so much strength and comfort also outside the university spaces.
Matteo Ferrara and Federico Fucci, who helped me with the server configurations, their experience and pleasant company.
The many thinkers and friends, whose ideas and efforts have significantly contributed to shape my professional and human personality.

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| **AI** | **A**rtificial **I**ntelligence |
| **ANN** | **A**rtificial **N**eural **N**etwork |
| **BP** | **B**ack **P**ropagation |
| **CAS** | **C**omputer **A**lgebra **S**ystem |
| **CNS** | **C**entral **N**eurvous **S**ystem |
| **CNN** | **C**onvolutional **N**eural **N**etwork |
| **CUDA** | **C**ompute **U**nified **D**evice **A**rchitecture |
| **CV** | **C**omputer **V**ision |
| **DL** | **D**eep **L**earing |
| **GIF** | **G**raphics **I**nterchange **F**ormat |
| **HSR** | **H**TM **S**upervised **R**efinement |
| **HTM** | **H**ierarchical **T**emporal **M**emory |
| **JPEG** | **J**oint **P**hotographic **E**xperts **G**roup |
| **KNN** | **K**-**N**earest **N**eighbor |
| **MHWA** | **M**ulti-stage **H**ubel-**W**iesel **A**rchitectures |
| **ML** | **M**achine **L**earing |
| **MLP** | **M**ulty **L**ayer **P**erceptron |
| **NN** | **N**eural **N**etwork |
| **NORB** | **N**ew York **O**bject **R**ecognition **B**enchmark |
| **PNG** | **P**ortable **N**etwork **G**raphics |
| **PNS** | **P**eripheral **N**eurvous **S**ystem |
| **RGB** | **R**ed **G**reen and **B**lue |
| **RL** | **R**einforcement **L**earning |

*To my dearest love Marina,*
*for helping me go through the most important chapter,*
*the chapter we have written together in the wonderful book of life.*

# Introduction

Since the spread of the first computers, mathematicians, philosophers, psychologists and of course computer scientists have always tried to understand the very nature of computation and how it can be linked to "intelligence". The mechanistic perspective from the eighteenth century and the connectionist approach to cognitive psychology in the '90s have always produced a lot of enthusiasm and hope regarding the possibility of the strong AI [17]. Nowadays, after almost two AI "winters" and new incredible results in the field of machine learning, a regained interest in these matters seems to lead the research community all over the world [18]. A catalyzing factor is, for sure, the unlock of generalized features learning methods, i.e. the ability of automatically discriminate class features without any domain-specific instructions [4]. This is thought to be at the heart of intelligence, and it is concretely helping many business applications. This new machine learning trend that comes with the name of Deep learning, is deeply changing the way machine learning was performed, generally with hand-crafted extraction of salient features [19]. The computer vision community, for example, can work out now, for the first time, methods that are dealing directly with raw images data. Actually, most of the insights that are conducting recent research progress, are dated back in the last century. However, only recent advances in technology (computational speed) and further mathematical tricks have enabled the real usefulness and effectiveness of these approaches.

In fact, deep learning techniques are incredibly computationally intensive and they need a huge quantity of data to work well. Moreover, they are criticized to be too focused on certain mathematical aspects and to ignore fundamental principles of intelligence. A consistent part of the scientific community is working hard to answer these questions. The current dissertation fits exactly on this research track and tries to compare two deep learning algorithms (CNN and HTM) in a computer vision context, specifically in object recognition. It has two main goals. Firstly, pushing object recognition research towards images sequences or video analysis, and secondly proving that with a lower quantity of data HTM can outperform CNN in terms of accuracy while remaining comparable in terms of training times. In Chapter 1 a brief background about machine learning and

artificial neural networks will be covered. In chapter 2 and 3, the two algorithms will be explained in great details. In chapter 4, a new benchmark for image sequences will be introduced and in chapter 5, experiments results will be reported. Eventually, in chapter 6 conclusions will be drawn and future work directions suggested.

# 1

# Background

> *"For generations, scientists and philosophers have tried to explain ordinary reasoning in terms of logical principles with virtually no success. I suspect this enterprise failed because it was looking in the wrong direction: common sense works so well not because it is an approximation of logic; logic is only a small part of our great accumulation of different, useful ways to chain things together."*
>
> – prof. Marvin Lee Minsky, *The Society of Mind (1987)*

In this chapter a brief background about machine learning and artificial neural networks is provided. In the following sections the reader will be introduced to the main concepts behind the work carried out in this dissertation, even with the help of strict mathematical notations when required.

## 1.1 Machine Learning

Learning is a very interesting and articulated phenomenon. Learning processes include the acquisition of new declarative knowledge, the development of motor and cognitive skills through instruction or practice, the organization of new knowledge into general, effective representations, and the discovery of new facts and theories through observation and experimentations. Since the birth of computing, researchers have been striving to implant such capabilities in computers. Solving this problem has been, and still remains, one of the most challenging and fascinating long-term goals in artificial intelligence. The study of computer modeling of learning processes in their multiple manifestations constitutes the subject matter of machine learning.

In 1959, Arthur Samuel defined machine learning as a "*Field of study that gives computers the ability to learn without being explicitly programmed*" [20].

Tom M. Mitchell provided a widely quoted, more formal definition: "*A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E*" [21].

This definition is notable because it defines machine learning in fundamentally operational terms rather than cognitive ones, thus following Alan Turing's proposal in his paper *Computing Machinery and Intelligence* that the question "Can machines think?" be replaced with the question "Can machines do what we (as thinking entities) can do?" [22]

### 1.1.1 Categories and tasks

Usually, machine learning tasks are classified into three broad categories. These depend on the nature of the learning "signal" or "feedback" available to a learning system: [17]

- **Supervised learning**: Is the machine learning approach of inferring a function from *supervised* training data. The training data consist of a set of *training examples* i.e. pairs consisting of an input object (typically a vector) and a desired output value (also called the supervisory signal). A supervised learning algorithm analyzes the training data and produces an *inferred function*, which can generalize from the training data to unseen situations in a "reasonable" way.
- **Unsupervised learning**: Closely related to pattern recognition, unsupervised learning is about analyzing data and looking for patterns. It is an extremely powerful tool for identifying structure in data. Unsupervised learning can be a goal in itself or a means towards an end.
- **Reinforcement learning**: Is learning by interacting with an environment. An RL agent learns from the consequences of its actions, rather than from being explicitly taught and it selects its actions on basis of its past experiences (exploitation) and also by new choices (exploration), which is essentially trial and error learning. The reinforcement signal that the RL-agent receives is a numerical reward, which encodes the success of an action's outcome, and the agent seeks to learn to select actions that maximize the accumulated reward over time.

Between supervised and unsupervised learning another category of learning methods can be found. It is called Semi-supervised learning and it is used in the presence of an

incomplete training signal: a training set with some (often many) of the target outputs missing. *Transduction* is a special case of this principle where the entire set of problem instances is known at learning time, except that part of the targets are missing.

Among other categories of machine learning problems, it is worth pointing out *Multi-task learning* which learns its own inductive bias based on previous experience. On the other hand, *Developmental learning* is elaborated for robot learning and generates its own sequences (also called curriculum) of learning situations to cumulatively acquire repertoires of novel skills through autonomous self-exploration and social interaction with human teachers. It also uses guidance mechanisms such as active learning, maturation, motor synergies, and imitation.

Another categorization of machine learning tasks arises considering the desired output of a machine-learned system: [23]

- In **classification**, inputs are divided into two or more classes, and the learner must produce a model that assigns unseen input patterns to one or more of these classes (fuzzy classification). This is typically tackled in a supervised way. Spam filtering is an example of classification, where the inputs are email (or other) messages and the classes are "spam" and "not spam".
- In **regression**, which is also a supervised problem, the outputs are continuous rather than discrete.
- In **clustering**, a set of input patters have to be divided into groups. Unlike in classification, the groups are not known beforehand, making this typically an unsupervised task. *Topic modeling* is a related problem, where a program is given a list of human language documents and is asked to find out which documents cover similar topics.
- **Density estimation** finds the distribution of input patterns in some space.
- **Dimensionality reduction** simplifies inputs by mapping them into a lower-dimensional space.

## 1.2 Computer Vision

Computer vision is a field which collects methods for acquiring, processing, analyzing, and understanding images and, in general, high-dimensional data from the real world. The aim of the discipline is to elaborate these data to produce numerical or symbolic information in the forms of decisions [24]. A fundamental idea that has always stood

behind this field has been to duplicate the abilities of human vision by electronically perceiving and understanding an image [25]. This image understanding can be seen as the disentangling of symbolic information from image data using models constructed with the aid of geometry, physics, statistics, and learning theory [26]. Computer vision has also been defined as the enterprise of automating and integrating a wide range of processes and representations for vision perception.

Being computer vision a scientific discipline, it is concerned with the theory behind artificial systems extracting information from images. The image data can take many forms, such as image sequences, views from multiple cameras, or multi-dimensional data from a medical scanner. From a technological point of view, computer vision seeks to apply its theories and models to the construction of computer vision systems.

Sub-fields of computer vision include object recognition, scene understanding, video tracking, event detection, object pose estimation, learning, indexing, motion estimation, and image restoration.

### 1.2.1 Object recognition

Object recognition is the task within computer vision which is concerned with the finding and identification of objects in images or video sequences. Humans are able to recognize a multitude of objects without much effort, despite the fact that the objects in the images may vary significantly due to different view points, many different sizes and scales, lighting conditions and poses. Objects can even be recognized when the view is partially obstructed. This task is still a challenge for computer vision systems. Many approaches to the task have been implemented over multiple decades. In this dissertation, this task will be confronted in-depth.

## 1.3 Artificial neural networks

In machine learning, artificial neural networks (ANNs) are a family of statistical learning models inspired by biological neural networks (common in the brains of many mammals) [17]. They can be used to estimate or approximate functions that can depend on a large number of inputs and are generally unknown. Artificial neural networks are generally presented as systems of interconnected "neurons" which send messages to each other. Each of their connection has a numeric weight that can be tuned based on experience, making neural nets adaptive to inputs and capable of learning [27].

For example, a neural network for handwriting recognition could be defined as a set of input neurons which may be activated by the pixels of an input image. After being weighted and transformed by a preconceived function, the activations of these neurons are then passed on to other neurons. This process is repeated until finally, an output neuron is activated. This determines which character has been read.

Along with other machine learning methods, neural networks have been used to solve a wide variety of tasks which would be generally hard to solve using ordinary rule-based programming, including computer vision and speech recognition.

### 1.3.1 From neuron to perceptron

A neuron, also known as "neurone" or "nerve cell", is an electrically excitable cell that processes and transmits information through electrical and chemical signals [28]. These signals between neurons occur via synapses, specialized connections with other cells. Neurons can connect to each other to form neural networks. Neurons are the key components of the brain and spinal cord of the central nervous system (CNS), and of the ganglia of the peripheral nervous system (PNS). Specialized types of neurons include: *sensory neurons* which respond to touch, sound, light and all other stimuli affecting the cells of the sensory organs that then send signals to the spinal cord and brain, *motor neurons* that receive signals from the brain and spinal cord to cause muscle contractions and affect glandular outputs, and *interneurons* which connect neurons among each other within the same region of the brain or the spinal cord.

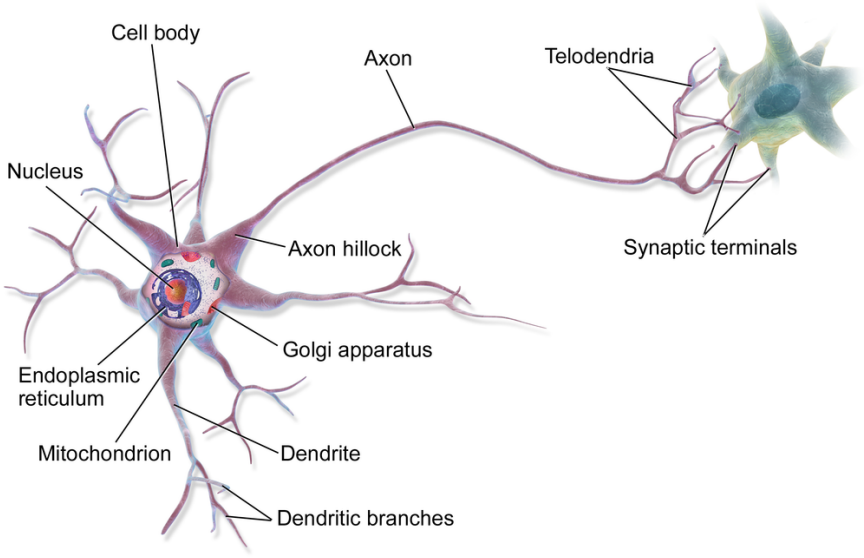


FIGURE 1.1: Anatomy of a multipolar neuron [12].

A typical neuron consists of a cell body (soma), dendrites, and an axon. The term neurite is used to describe either a dendrite or an axon, particularly in its undifferentiated stage. Dendrites are thin structures arising from the cell body, often extending for hundreds of micrometres and branching multiple times, giving rise to a complex "dendritic tree". An axon is a special cellular extension that arises from the cell body at a site called the axon hillock and travels for a distance, as far as 1 meter in humans or even more in other species. The cell body of a neuron frequently brings about multiple dendrites, but never more than one axon, although the axon may branch hundreds of times before it terminates. To the majority of synapses, signals are sent from the axon of one neuron to a dendrite of another.

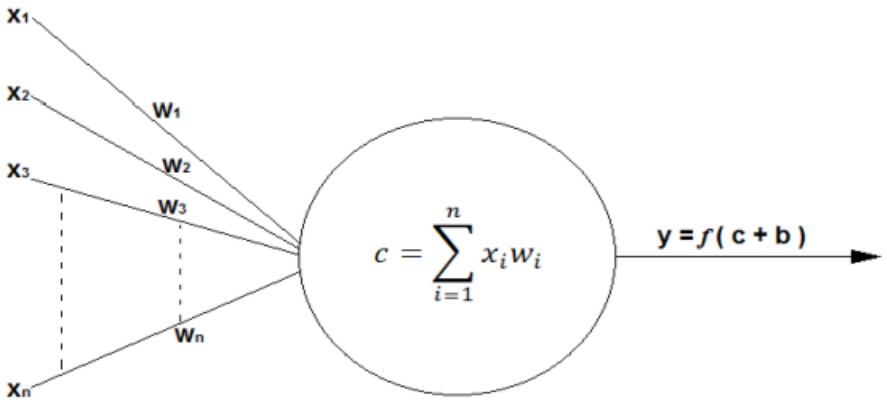


FIGURE 1.2: A graphical representation of the perceptron [13].

The *perceptron* is the mathematical abstraction of a neuron and a binary classifier that combined with other counterparts can lead to great results in pattern recognition [29]. A perceptron receives an input vector $x$ consisting of $n$ elements. The linear combination $c$ of the vector input $x$ and a weight vector $w$ is called action potential. The inputs of the perceptron represent signals collected from dendrites, while the weights represent the signal attenuation exercised by the neuron. However, unlike real neurons, the perceptron can also amplify its input. The threshold of a neuron is represented by $b$, which is then added to $c$. Finally, the result of this sum is passed to the activation function $f$ and $y$ is its return value. In fig. 1.2 a graphical explanation of the perceptron is provided. To explain how to interpret the value of $y$, considering that, for example, there are only two classes to distinguish, a value of $y$ less or equal than 0 then could indicate that the input belongs to the first class (-1), and accordingly, a value of y greater than 0 could indicate that the input belongs to the second class (1).

The weights $w_1, w_2, w_3, \ldots, w_n$, the threshold $b$ and the function $f$ are the fundamental characteristics that distinguish a perceptron from another. The activation function is chosen at the design time depending on the data set (e.g. set of patterns in the training

set). The *sigmoid* function is often used for this purpose. The weights $w_1, w_2, w_3, \ldots, w_n$, the threshold $b$, are parameters for which it is necessary to find the optimum combination of values through the minimization of a function that represents the error committed:

$$E(w,b) = \frac{1}{2}\sum_{j=1}^{m}(f(\sum_{i=1}^{n}(w_i, x_i^j) + b) - z_j)^2 \tag{1.1}$$

In the formula above:

- $w$ it is the vector composed of all weights associated with the input of the perceptron.
- $b$ is the threshold associated with the perceptron.
- $m$ is the number of elements in the training set.
- $f$ is the activation function of choice.
- $x_j$ is the j-th example (each example is a vector) of the training set.
- $z_j$ is the desired result that should give the perceptron when it receives ad input the j-th sample of the training set.
- $n$ is the number of weights, one speaks then of the number of elements in the vector w.

The formula 1.1 can be suitably simplified considering $b$ as a weight within $w$ ($w_0 = b$) and associating to it an input $x_0 = 1$. As a result the following equation is obtained:

$$E(w) = \sum_{j=1}^{m} T_{x^j}(w) \tag{1.2}$$

$$T_{x^j}(w) = \frac{1}{2}(f(\sum_{i=0}^{n}(w_i, x_i^j)) - z_j)^2 \tag{1.3}$$

Hence, after having provided the error equations, the objective is to find the combination of values for $w$ that minimizes $E(w)$. For this purpose a minimization method which starts from a random solution can be applied. Indeed, using a gradient descent technique implemented as an iterative method it is possible to get closer and closer to the minimum point that corresponds to the solution of the problem. Frank Rosenblatt, who invented the perceptron, showed that the process just discussed above can be obtained with the following iterative steps:

$$w^{j+1} = w^j + \alpha(z_j - y_j)f'(w^j x^j)x^j \qquad \forall j \in \{1, \ldots, m\} \tag{1.4}$$

In the equation:

- $w^j$ is the weight vector at step $j$.
- $\alpha$ is the *learning rate*, which is the displacement step to minimize the error; If it is too large it can make impossible to reach the minimum, while if it is too small it can greatly slow down the convergence.
- $y_j$ is the output of the perceptron when it processes input $x_j$.
- $(z_j - y_j)f'(w_j x_j)$ can be seen as the mistake of the perceptron prosessing the pattern $x_j$.
- $-(z_j - y_j)f'(w_j x_j)x_j$ is the gradient of $T_{x^j}$ calculated on $w_j$.

The minimum of $E$ can be achieved by repeating the procedure in 1.4 starting each time from the last vector of the weights obtained from the previous computation.

Despite its mathematical elegance and due to its inherent simplicity, a perceptron can only solve binary classification problems. Moreover, it can perform only a linear classification, which can not be accurate when patterns are not linearly separable. In spite of everything, the perceptron is the necessary building block of the entire work carried out and discussed in this dissertation as well as a critical invention for the field of machine learning. Of course, there are many other machine learning techniques that do not rely on neural networks abstractions, but there is no interest in discussing them in this context.

### 1.3.2 Multilayer perceptron

In the light of the strong limitations of the perceptron, trying to link multiple units to create a larger structure appears natural. The first example of an artificial neural network was called *multilayer perceptron* (MLP) [30]. A multilayer perceptron can classify non linearly separable patterns and works well even when there are more than two classes. Theoretically perceptrons can be combined at will, but more than thirty years of research effort have established commonly adopted rules for simple feed-forward networks in order to build efficient and effective architectures for solving useful problems in pattern recognition:

- A MLP should have one or more input perceptrons, i.e. perceptrons that receive as input the original pattern which is not coming from other perceptrons.
- A MLP should have as many output perceptrons as classes (each of these represents a class), where an output perceptron is a simple unit that does not have exit connections.

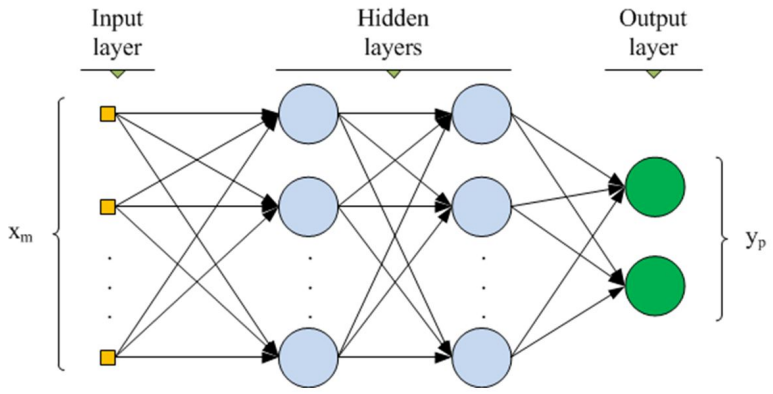


FIGURE 1.3: Multilayer Perceptron commonly used architecture [14].

- Given a pattern $p$ as input, a MLP should associate $p$ to the class represented by the output perceptron computing the higher value among its peers.
- In a MLP, the set of input and output perceptrons should not be necessarily disjoint;
- In a MLP, all the perceptrons should share the same activation function;
- In a MLP, connections among perceptrons should not be cyclic.

In figure 1.3 is shown the most used MLP architecture in machine learning. it is a three-layer architecture composed of an input layer, a hidden layer and an output layer. The input layer is not composed of real perceptrons but simple propagators which provide the same input to all the perceptrons of the hidden layer. On the other hand, in the hidden layer each perceptron is connected to each perceptron in the output layer. Finally, the output layer has no output connections. Broadly speaking, the network showed in fig. 1.3, is a feed-forward artificial neural network in which perceptrons take their input only from the previous layer, and send their output only to the next layer. To train a neural network a powerful technique called *back-propagation* can be used [31]. This algorithm, unknown until the '80s, can adjust the weights of the network propagating the error backwards (starting from the output layer). In the following paragraph, this technique is described in detail. Unlike the perceptron, artificial neural networks are good classifiers even if it is worth pointing out that they are very different from their biological counterparts.

### 1.3.3 The back-propagation algorithm

The back-propagation technique, requires to consider for each iteration a new training pattern $x$. Assuming that the output of the hidden layer are $q_1, q_2, q_3, \ldots, q_m$, it is possible to perform the following calculations:

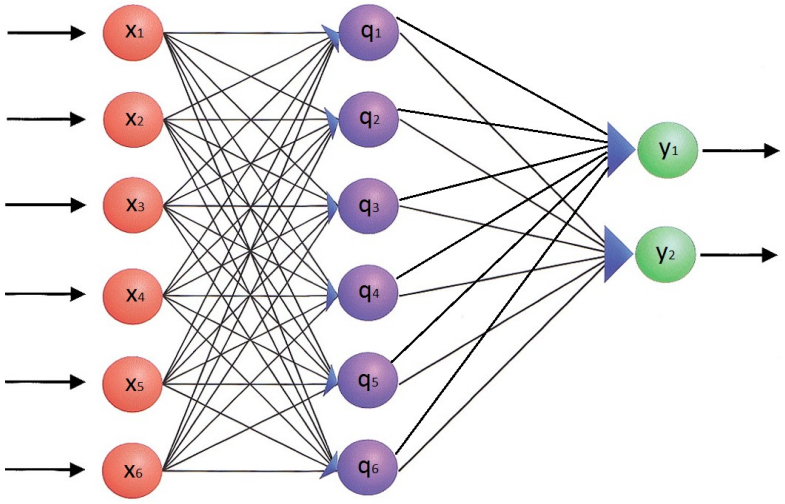


FIGURE 1.4: Another Multilayer Perceptron architecture example [13].

$$q_i = f(h^i x) \qquad \forall i = 1, \ldots, m \tag{1.5}$$

Then, $y_1, y_2, y_3, \ldots, y_n$, the output of the MLP, can be computed as follows:

$$y_i = f(w^i q) \qquad \forall i = 1, \ldots, n \tag{1.6}$$

Where:

- $w^i$ is the weights vector associated with the i-th perceptron in the output layer;
- $h^i$ is the weight vector associated with the i-th perceptron in the hidden layer
- $f$ is the activation function.

The *last square error* on $x$ of the MLP can be calculated as follows:

$$T_x = \frac{1}{2} \sum_{i=1}^{n} (z_i - y_i)^2 \tag{1.7}$$

In the above equation, $z_i$ is the exact result the $i^{th}$ output perceptron should produce and which can be retrieved from the training set. As for the perceptron, we would like to minimize the error by finding the right combination of values for $w_i$ for $i = 1, \ldots, n$ and $h_j$ for $j = 1, \ldots, m$. Also in this case a gradient descent technique can be applied in order to minimize the error function. For the minimization we start calculating the error from the output layer:

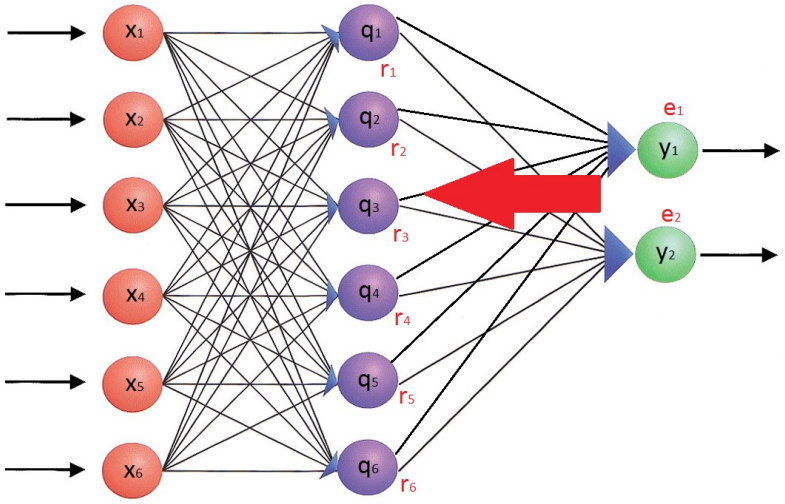


FIGURE 1.5: At every iteration, an error is associated to each perceptron to update the weights accordingly [13].

$$e_i = (z_i - y_i) f'(w^i q) \qquad \forall i \in \{1, \dots, n\} \tag{1.8}$$

Then the error can be propagated backward, based on that computed on the output layer:

$$r_i = f'(h^i x) \sum_{k=i}^{n} e_k w_i^k \qquad \forall i \in \{1, \dots, m\} \tag{1.9}$$

In this case we are considering a neural network with just one hidden layer, but the back-propagation can be easily extended to any type of network by repeating the computation shown in 1.9 by treating the next hidden layer as if it were the output layer. After the error back-propagation the input weights of the output perceptrons can be updated as follows:

$$w^i = w^i + \alpha \cdot e_i \cdot q \qquad \forall i \in \{1, \dots, n\} \tag{1.10}$$

And the input weights of the hidden layer can be adjusted as follows:

$$h^i = h^i + \alpha \cdot r_i \cdot x \qquad \forall i \in \{1, \dots, m\} \tag{1.11}$$

With the last two equations, just a single step ahead (with a *learning rate* of $\alpha$) in the opposite direction to the gradient of $T_x$ (computed on the previous configuration of weights) has been made. Repeating this procedure for each pattern in the training set completes the back-propagation. Now let's consider the corresponding pseudo-code:

**Algorithm 1** back-propagation algorithm

```
1: procedure Back–Propagation
2:     for each input pattern x do
3:         forward propagation
4:         compute error for each output perceptron
5:         for i = K to 1 do //with K number of hidden layer
6:             back-propagate error on the hidden layer i
7:         end for
8:         update weights of the output perceptrons
9:         for i = K to 1 do
10:            update weights of the hidden layer i
11:        end for
12:    end for
13: end procedure
```

Moreover, to minimize the error of all the examples, the back-propagation is repeated several times always starting from the last configuration of weights computed in the previous iteration. The training ends after a predetermined number of iteration or if the error committed by the network does not exceed a certain threshold. It is worth saying that using a constant learning rate could not be that good and we may want to decrease it at each iteration to be more and more accurate as the training goes on.

## 1.4 Deep Learning

Deep learning (deep machine learning, or deep structured learning, or hierarchical learning, or sometimes DL) is a branch of machine learning which comprises a set of algorithms attempting to model high-level abstractions in data through model architectures with complex structures or otherwise, composed of multiple non-linear transformations. [32]
Deep learning is part of a broader family of machine learning methods based on learning representations of data. An observation (like an image, for example) can be coded in many ways such as a vector of intensity values, or more abstractly as a set of edges, regions of particular shape, etc... Some representations make it easier to learn tasks (e.g., face recognition or facial expression recognition) from examples.
The most important and characterizing feature of deep learning is the depth of the network. Until a few years ago, it was thought that an MLP with a single hidden layer would have been sufficient for almost any complex task. With the increasing amounts of data and computing power now available, the advantage of building deep neural networks with a large number of layer (up to 10 or even greater) has been recognized.
As already mentioned, another important feature of deep learning the ability of replacing handcrafted features with efficient algorithms for unsupervised or semi-supervised feature learning and hierarchical feature extraction.

Research in this area attempts to make better representations and create models to learn these representations from large-scale unlabeled data. Some of the representations are inspired by advances in neuroscience and are loosely based on interpretation of information processing and communication patterns in the nervous system, such as *neural coding* which attempts to define a relationship between the stimulus and the neuronal responses and the relationship among the electrical activity of the neurons in the brain [33].

Various deep learning architectures such as deep neural networks, convolutional deep neural networks, deep belief networks and recurrent neural networks have been applied to fields like computer vision, automatic speech recognition, natural language processing, audio recognition and bioinformatics where they have been shown to produce state-of-the-art results on various tasks. Besides, deep learning is often regarded as buzzword, or a simple rebranding of neural networks [34].

# 2

# CNN: State-of-the-art in object recognition

*"If we were magically shrunk and put into someone's brain while she was thinking, we would see all the pumps, pistons, gears and levers working away, and we would be able to describe their workings completely, in mechanical terms, thereby completely describing the thought processes of the brain. But that description would nowhere contain any mention of thought! It would contain nothing but descriptions of pumps, pistons, levers!"*

– G. W. Leibniz (1646–1716)

The current chapter describes in details the Convolutional Neural Network (CNN) approach to pattern recognition starting from the basic concepts of convolution and artificial neural network. The algorithm is introduced in the context of object recognition, focus of this dissertation, even if these networks can be conveniently used on patterns of any type.

## 2.1 Digital images and convolution operations

A digital image can be defined as the numerical representation of a real image. This representation can be coded as a vector or a bitmap (raster). In the first case it describes the primitive elements (lines or polygons) which compose the image, in the second case the image is composed of a matrix of points, called pixels. Their color is defined by one or more numerical values. In coloured bitmap images the color is stored as level of intensity of the basic colors, for example in the RGB model there are three colors: red,

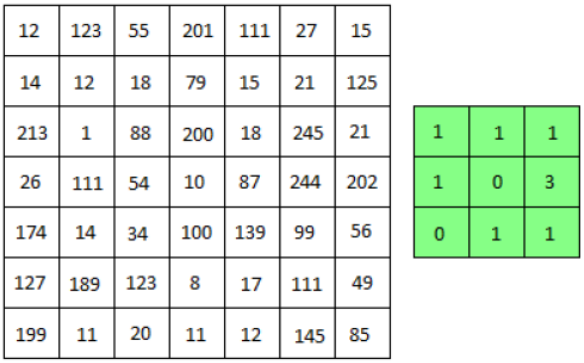


FIGURE 2.1: Examples of a digital image representation and a 3x3 convolution matrix or filter [13].

green and blue. In *grayscale* (improperly called black and white) bitmap images the value indicates different gray intensities ranging from black to white. The images that are further elaborated in this dissertation are grayscale bitmap images. The number of colors or possible gray levels (depth) depends on the amount of bits used to code them: an image with 1 bit per pixel will have a maximum of two possible combinations (0 and 1) and thus may represent only two colors, images with 4 bits per pixel can represent up to 16 colors or 16 levels of gray, an image with 8-bit per pixel can represent 256 colors or gray levels, and so on. Bitmap images can be stored in different formats often based on a compression algorithm. The algorithm can be lossy (JPEG) or lossless, i.e. without loss (GIF, PNG). This type of images is generated by a wide variety of acquisition devices, such as scanners, digital cameras, webcams, but also by radar and electronic microscopes.

### 2.1.1 One-to-one convolution

The convolution [24] is an operation that is performed on a mono-color bitmap image for emphasizing some of its features. In order to do so, a digital image and a convolution matrix are needed, consider the fig. 2.2. The convolution matrix is also called filter. A filter can be thought as a sliding window moving across the original image. At every shift it produces a new value, this value is obtained by summing all the products between the filter elements and the corresponding pixels. The values obtained from all the possible placements of the filter above the image are inserted in an orderly fashion in a new image. With a convolution is therefore obtained an image that highlights the characteristics enhanced by the filter used. Then, as regards the size of the new image we have:

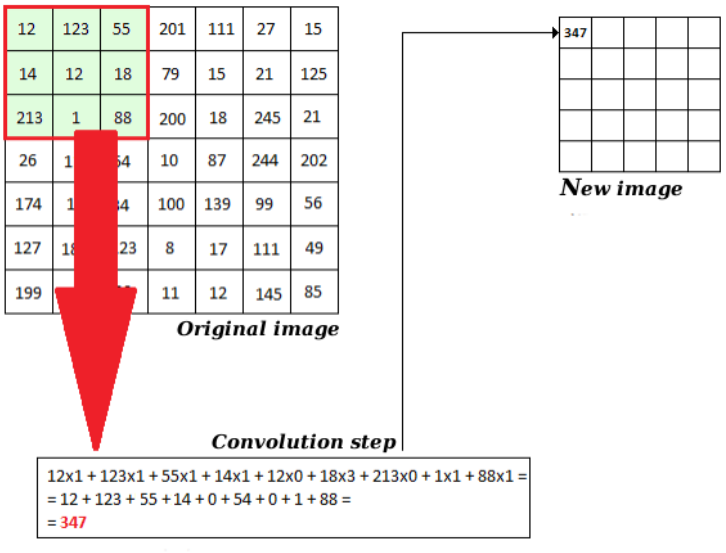


FIGURE 2.2: First step of a convolution performed on a 7x7 image and a 3x3 filter.

$$b_n = b_v - b_f + 1 \quad \textit{and} \quad h_n = h_v - h_f + 1 \tag{2.1}$$

where:

- $b_n$ and $h_n$ are respectively the width and the height of new image resulting from the convolution;
- $b_v$ and $h_v$ are respectively the width and the height of the original image;
- $b_f$ and $h_f$ are respectively the width and the height of the filter used.

In computer vision, several filters are often used and each of them has a particular purpose. The more popular are:

- *Sobel filters*: generally used to highlight edges;
- *Gaussian filters*: generally used to remove noise;
- *High-pass filter*: generally used to increase image details;
- *Emboss filter*: generally used to accentuate brightness differences.

The convolution operation is at the heart of convolutional neural networks where filters are automatically learned in an unsupervised fashion during the training phase.

### 2.1.2 Many-to-many convolution

Until now, we have only discussed *one-to-one convolutions* that take in input a single image and return a single image as the output. Actually, different kind of convolutions

exist and are commonly used in convolutional neural networks. We talk about *one-to-many convolutions* when there is only one input image and $n$ filters; each filter is used to generate a new image (also called *feature map*). Finally, in *many-to-many* convolutions there are $m$ input images and $n$ output images. Each output image can be connected with one or more images from the input. Each connection between an input and an output image is characterized by a filter; For each pixel of the output image, first the corresponding convolution step is performed, then the respective intermediate results are summed together. The higher is the total number of connections and the more exhaustively the images are processed during the learning. Even if, it is clear that the computational time increases sharply with the rise of the connections.

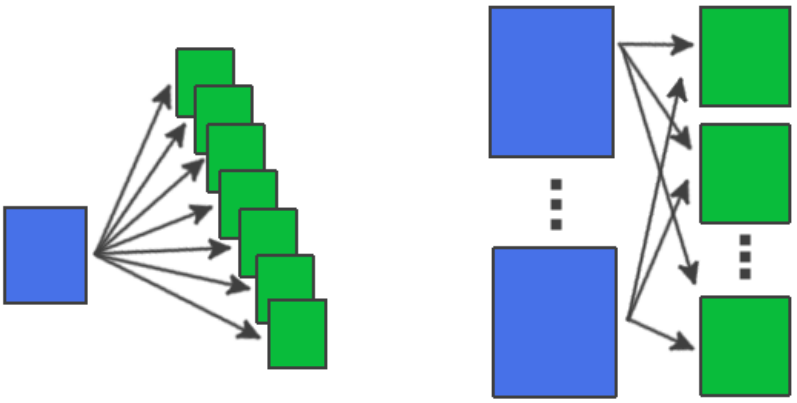

FIGURE 2.3: One-to-many and many-to-many convolution examples.

## 2.2 Unsupervised feature learning

Considering the simplest case, a step of convolution can be expressed with the following formula:

$$c = \sum_{\forall (i,j) \in S} F_{i,j} \cdot S_{i,j} \tag{2.2}$$

where:

- $c$ is the result of the convolution step convolution;
- $S$ is the considered sub-image;
- $F_{i,j}$ is the element of row $i$ and column $j$ of the filter;
- $S_{i,j}$ is the element of row $i$ and column $j$ of $S$.

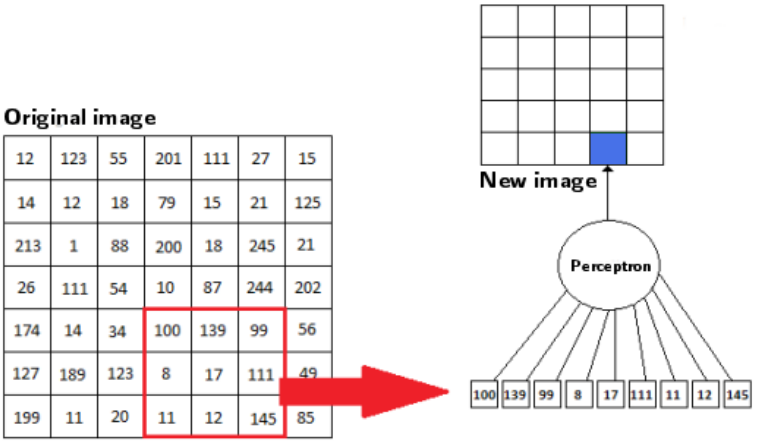


FIGURE 2.4: A convolution step performed with a perceptron.

Now, considering $f$ as an arbitrary activation function and b as a bias, if we substitute the result of the convolution step $c$ with $y$ and we calculate $y$ as follows:

$$y = f(c + b) \tag{2.3}$$

Then we end up with a computation that is pretty similar to what described in the preceding chapter regarding the perceptron. This is commonly referred to as a convolution performed with a perceptron where the weights of the former are the same of the filter. In a convolutional neural network the key element is the *convolutional layer*, or a module that performs many-to-many convolutions with many perceptrons. in this module, each output image has an associated perceptron that takes input from all the related images at every step of convolution. A convolutional layer has therefore a set of perceptrons and their training leads to the ideal filters depending on the cost function that is minimized through the back-propagation algorithm. In a convolutional layer, it is also possible to decrease the computational complexity and the output size by applying the filters every s pixels (where s is called stride). This is generally not recommended as it compromises the quality of the output.

## 2.3 Downsampling

In convolutional neural networks another important module called *subsampling layer* [32] exists: its main task is to carry out a reduction of the input images size in order to give the algorithm more invariance and robustness. A subsampling layer receives $n$ input images, and provides $n$ output; The images to be reduced are divided into blocks

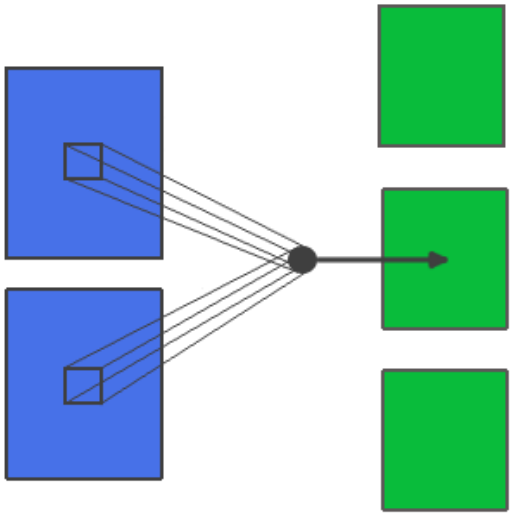

FIGURE 2.5: Two imput images and one perceptron that operates as a filter.

that have all the same size, then, each block is mapped to a single pixel in the following way:

$$y = f(\sum_{\forall(i,j)\in B} B_{i,j} \cdot w + b) \tag{2.4}$$

where:

- $y$ is the result of the subsampling step;
- $f$ is the activation function;
- $B$ is the block considered in the input image;
- $B_{i,j}$ is the value of the pixel $(i, j)$ within the block $B$;
- $w$ is the adjustment coefficient;
- $b$ is the threshold;

The computation performed on an input image from a subsample layer is made by a perceptron that has all equal weights (threshold excluded). Hence, training $n$ perceptrons is sufficient to generate $2n$ parameters.

A subsample layer is less powerful than a convolutional layer. In fact, as previously pointed out, the higher is the number of trainable parameters the higher is the discretionary capacity of our network. Then, why not using only convolutional layers? There are three main reasons:

1. There are fewer parameters and the training phase is faster;

2. A subsampling layer performs its tasks more quickly than a convolutional layer due to a smaller number of steps and products;
3. The subsample layer allow CNN to tolerate translations and rotations among the input patterns. In practice, a single subsampling layer is not enough, but it has been seen that an alternation of feature extraction and subsampling layers can handle different types of invariance which a sequence of only convolutional layers can not manage.

It is also possible to modify a subsampling layer in order to divide the input images into overlapping blocks. Of course, this configuration can slow down the reduction process. Indeed, depending on the task, the temporal aspect could be extremely important both in the training and the test phases that might have to take place within few milliseconds. Sometimes even simpler downsampling layer can be used to speed-up the training. A great example of this approach is the so called *max-pooling* layer. In this layer only the following operation are performed:

- $c_1$ is computed as the max value among the pixels of the block;
- $c_2$ is obtained summing $c_1$ with his bias;
- resulting pixel is obtained computing the activation function on $c_2$

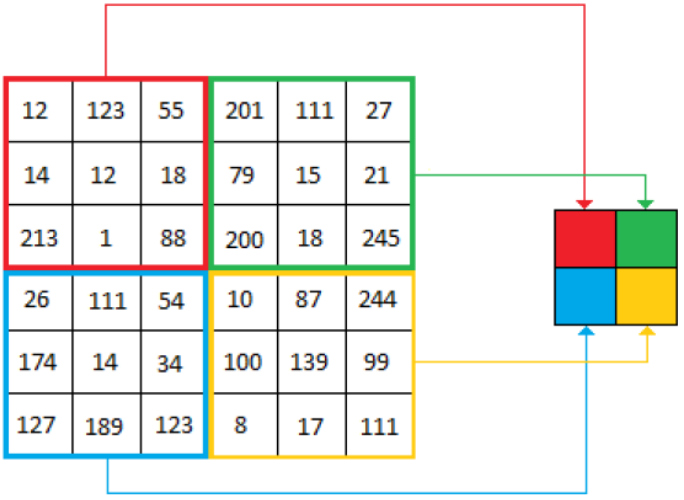


FIGURE 2.6: Image chunking in a downsampling layer [13].

In a CNN each subsampling layer can be replaced with a max-pooling layer. A max-pooling layer has less parameters to train due to the elimination of the adjustment coefficients. With the parameters reduction more speed is obtained during the training phase, but often with lower degree of learning.

## 2.4 CNN architecture

Convolutional Neural Networks (CNN) [35] are powerful classifiers which can be used in many tasks. They are extremely suitable when the number of training pattern and their dimensionality are particularly high. This is why they are often used in computer vision dealing successfully with digital images. A CNN receives in input $p$ mono-color bitmap (for example, can receive the three channels R, G and B of a coloured image); then the input is given to a feature extraction module which releases an array of features consisting of $m$ elements; finally this array is delivered to a full connected neural network that generates the results. The neural network is composed of $n$ perceptron (as many as the classes) all both input and output. The feature module, instead, usually consists of a convolutional layer followed by zero, one or more pairs [subsampling layer, convolutional layer]. Hence, it has an odd number of sub-modules where the first and the last are convolutional layers. The last array of features is the result of the last convolutional layer where the input images are reduced to single values.

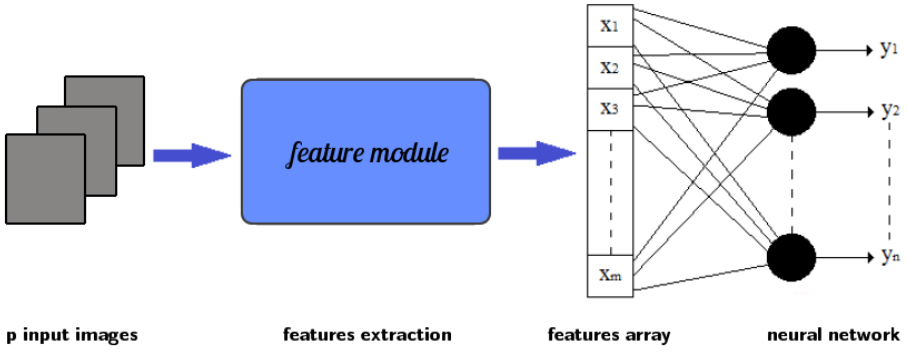


FIGURE 2.7: General CNN architecture composed of a feature module and a neural network of $n$ perceptron.

If $k$ is the number of layers in the CNN, it can be called a k-CNN or simpler, a CNN with $k$ layers. In theory, the higher is $k$, the greater is the degree of learning of the network. However in practice, too many layers can create convergence problems during the training. To train a CNN the classic supervised approach based on back-propagation is used. This approach allows the network to learn how to discern a pattern from another based on the training set. In CNNs the sigmoidal function is commonly used as the activation function of all the layers (including the neural network).

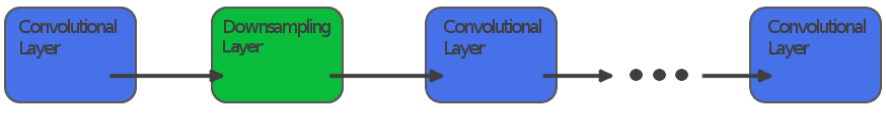


FIGURE 2.8: Layers alternation in a CNN features module.

In this section a general definition of a convolutional neural network has been provided. However, there are many variants depending on the task:

- CNN without final neural network;
- CNN with a final neural network with one or more hidden layers;
- CNN with an even number of layers in their features module.
- CNN with only convolutional layers.

In the dissertation we will continue to talk about CNNs referring to their general definition, even if it is important to understand that there are many researchers who can call with the same right convolutional neural networks slightly different architectures.

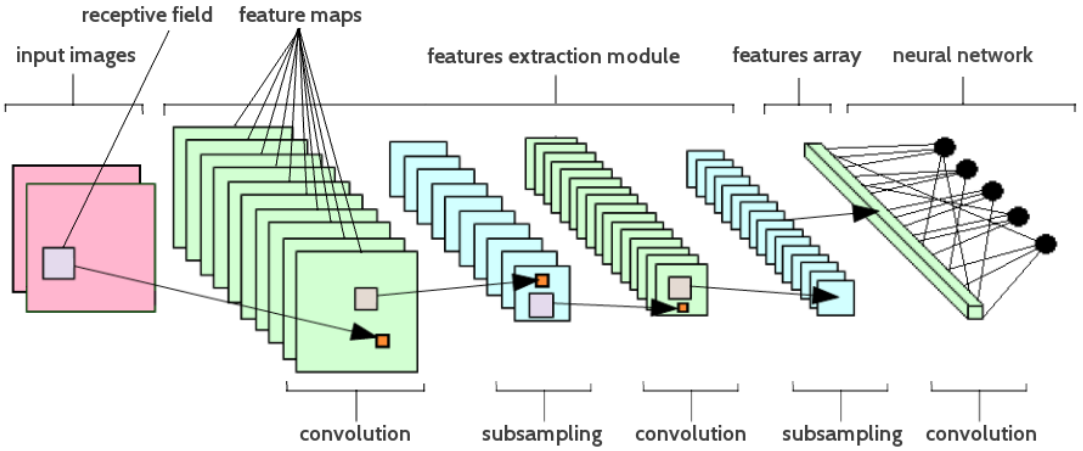


FIGURE 2.9: A complete example of a CNN architecture with seven layer, or commonly called LeNet7 from the name of its author Y. LeCun. All the convolutions are many-to-many, i.e. each feature map has a perceptron that can be connected with two or more feature maps of the previous layer.

## 2.5 CNN training

A CNN is trained using the back-propagation algorithm already discussed in the background chapter. Specifically, the neural network is trained using formulas 1.8 and 1.10. To train the generic module $l$ it is important to consider the following one. If the layer $l+1$ is the final neural network then we calculate the error as follows:

$$\sigma_j^l(0,0) = f'(a_j^l(0,0)) \sum_{k=1}^{n} (e_k w_{k,j}) \tag{2.5}$$

where:

- $\sigma_j^l(0,0)$ is the error of the $j^{th}$ output image of the module $l$ that is composed of a single pixel;
- $f$ is the activation function;
- $a_j^l(0,0)$ is the value passed to $f$ to obtain the value of the pixel with row 0 and column 0 of the $j^{th}$ image;
- $n$ is the number of perceptrons in the final neural network;
- $k$ is the error of the $k^{th}$ perceptron of the CNN output, this error is computed with the formula 1.8;
- $w_{k,j}$ is the weight between the $k^{th}$ output perceptron and the $j^{th}$ image of the module $l$;

If, instead, the module $l+1$ is a convolutional layer the error is calculated as follows:

$$\sigma_j^l(x,y) = f'(a_j^l(x,y)) \sum_{k \in K_j^l} \sum_{\forall (u,v) \in w_{k,j}^{l+1}} (\sigma_k^{l+1}(x-u, y-v) w_{k,j}^{l+1}(u,v)) \tag{2.6}$$

where:

- $\sigma_j^l(x,y)$ is the error of the $j^{th}$ image if the module l, on the pixel with coordinates $(x,y)$ (if $x<0$ or $y<0$ then $\sigma_j^l(x,y)$ is equal to zero)
- $a_j^l(x,y)$ is the value passed to $f$ in order to obtain the value of the pixel with coordinates $(x,y)$ of the $j^{th}$ image of the module $l$;
- $K_j^l$ is the set of indexes of the $l+1$ module images connected with the $j^{th}$ image of the module $l$;
- $w_{k,j}^{l+1}(u,v)$ is the value of column $u$ and row $v$ of the filter associated to the connection between the image $k$ of the module $l+1$ and the image $j$ of the module $l$;

Again, if the module $l+1$ is a subsampling layer the error is calculated as follows:

$$\sigma_j^l(x,y) = f'(a_j^l(x,y)) \sigma_j^{l+1}(\lfloor x/S_x \rfloor, \lfloor y/S_y \rfloor) w_j^{l+1} \tag{2.7}$$

where:

- $S_x$ is the width of the blocks in which the images are divided;
- $S_y$ is the height of the blocks in which the images are divided;
- $w_j^{l+1}$ is the adjustment coefficient of the $j^{th}$ image of the module $l+1$;

Lastly, if the module $l+1$ is a max-pooling layer:

$$\sigma_j^l(x,y) = \begin{cases} 0 & if\ y_j^l(x,y)\ is\ not\ a\ local\ max \\ f'(a_j^l(x,y))\sigma_j^{l+1}(\lfloor x/S_x \rfloor, \lfloor y/S_y \rfloor) & otherwise \end{cases} \tag{2.8}$$

where $y_j^l(x,y)$ stands for the value of the pixel with coordinates $(x,y)$ in the $j^{th}$ image of the module $l$.

Once errors are calculated we need to update the weights accordingly. If the module $l$ is a convolutional layer then:

$$\Delta w_{j,i}^l(u,v) = \alpha \sum_{\forall(x,y)\in\sigma_j^l} (\sigma_j^l(x,y) y_i^{l-1}(x+u, y+v)) \tag{2.9}$$

where $\alpha$ is the learning rate. If, instead, the module $l$ is a subsampling layer:

$$\Delta w_j^l = \alpha \sum_{\forall(x,y)\in\sigma_j^l} \sum_{c=0}^{S_x-1} \sum_{r=0}^{S_y-1} (y_i^{l-1}(x\cdot S_x + c, y\cdot S_y + r)) \tag{2.10}$$

Having computed the gradient, the weights can be updated according to the different type of module as follows. If the module $l$ is a convolutional layer:

$$w_{i,j}^l = w_{i,j}^l + \Delta w_{j,i}^l(u,v) \tag{2.11}$$

If, instead, the module $l$ is a subsampling layer:

$$w_j^l = w_j^l + \Delta w_j^l \tag{2.12}$$

With regard to the thresholds, they can be updated as follows:

$$b_j^l = b_j^l + \alpha \sum_{\forall(x,y)\in\sigma_j^l} (\sigma_j^l(x,y)) \tag{2.13}$$

So far we have considered a single update of the weights. Repeating this procedure for each pattern in the training set completes the back-propagation. Consider the corresponding pseudo-code:

**Algorithm 2** CNN training with back-propagation

```
1: procedure CNN-Back–Propagation
2:     for each input pattern x do
3:         forward propagation
4:         compute error for the final neural network
5:         for i = K to 1 do //with K number of feature modules
6:             compute the error based on the i+1 module i
7:         end for
8:         update weights of the final neural network
9:         for i = K to 1 do
10:            if module i is not a max-pooling layer then
11:                update weights of the module i
12:            end if
13:            update the thresholds of the module i
14:        end for
15:    end for
16: end procedure
```

As a final consideration let us consider the formula 2.8. If the layer l + 1 is a max-pool layer, then the total error for the images of the module l is small compared to the others. Hence the max-pool layer can accelerate the training not only for the smaller number of of parameters, but also because the initial error is lower.

# 3

# HTM: A new bio-inspired approach for Deep Learning

*"Prediction is not just one of the things your brain does. It is the primary function of the neo-cortex, and the foundation of intelligence."*

– Jeffrey Hawkins, *Redwood Center for Theoretical Neuroscience*

The current chapter describes briefly the Hierarchical Temporal Memory (HTM) approach to pattern recognition starting from its basic concepts. A comprehensive description of HTM architecture and learning algorithms is provided in [11].

## 3.1 Biological inspiration

Since its early days, artificial intelligence has always been conditioned by its biological counterpart. Even if, first artificial neural networks had very little in common with biological ones, after a deeper understanding of the human visual system, *multi-stage Hubel-Wiesel architectures* (MHWA) [33] [36] arose, and deep learning sprouted. These kind of architectures base their success on the ability of automatically discovering salient and discriminative features in any pattern. However, they still rely fundamentally on back-propagation, a statistical algorithm that is not exactly biologically inspired. HTM tries to integrate more key elements from biological learning systems like the human brain. Additionally, comparing high-level structures and functionality of the neocortex with HTM is most appropriate. HTM attempts to implement the functionality that is characteristic of a hierarchically related group of cortical regions in the neocortex. A region of the neocortex corresponds to one or more levels in the HTM hierarchy, while

the hippocampus is remotely similar to the highest HTM level. A single HTM node may represent a group of cortical columns within a certain region. Although it is primarily a functional model, several attempts have been made to relate the algorithms of the HTM with the structure of neuronal connections in the layers of the neocortex. [37] [38]

## 3.2 The HTM algorithm

HTM is almost a newborn algorithm that originates from the combination of brilliant engineering and intuitions. In spite of its recent development it could help managing invariance, which is a pivotal issue in computer vision and pattern recognition. Thus, some important properties can be exploited:

- The use of *time as supervisor*. Since minor intra-class variations of a pattern can result in a substantially different spatial representation (e.g., in term of pixel intensities), huge efforts have been done to develop variation-tolerant metrics (e.g., tangent distance [39]) or invariant feature extraction techniques (e.g., SIFT [40]). However, until now, positive outcomes have been achieved only for specific problems. HTM takes advantage of time continuity to assert that two representations, even if spatially dissimilar, originate from the same object if they come close in time. This concept, which lies at the basis of *slow feature analysis* [41], is simple but extremely powerful because it is applicable to any form of invariance (i.e., geometry, pose, lighting). It also enables unsupervised learning: labels are provided by time.

- *Hierarchical organization.* Lately, a great deal of studies furnished theoretical support to the advantages of hierarchical systems in learning invariant representations [33] [42]. Just like the human brain, HTM employs a hierarchy of levels to decompose object recognition complexity: at low levels the network learns basic features, used as building blocks at higher levels to form representations of increasing complexity. These building blocks are crucial as well for efficient coding and generalization because through their combination, HTM can even encode new objects which have never seen before.

- *Top down and bottom-up information flow.* In MHWA information typically goes only one-way from lower levels to upper levels. In the human cortex, both feed-forward and feed-back messages are steadily exchanged between different regions. The precise role of feed-back messages is still a heated debate, but neuroscientists agrees on their fundamental support in the perception of non-trivial patterns

[43]. Memory-prediction theory assumes that feed-back messages from higher levels bring contextual information that can bias the behavior of lower levels. This is crucial to deal with uncertainty: if a node of a given level has to process an ambiguous pattern, its decision could be better taken with the presence of insights from upper levels, whose nodes are probably aware of contextual information the network is operating in (for example, if one step back in time we were recognizing a person, probably we are still processing a crowed scene).

- *Bayesian probabilistic formulation.* When uncertainty is the central issue, it is often better to take probabilistic choices rather than binary ones. In light of this, the state of HTM nodes is encoded in probabilistic terms and Bayesian theory is widely used to process information. HTM can be seen as a Bayesian Network where Bayesian Belief propagation equations are adopted to let information flow across the hierarchy [44]. This formulation is mathematically elegant and allows to solve practical hurdles such as value normalization and threshold selection.

Deepening into details, an HTM has a hierarchical tree structure. The tree is built up by $n_{levels}$ levels (or layers), each composed of one or more nodes. A node in one level is bidirectionally connected to one or more nodes in the level above and the number of nodes in each level decreases as the hierarchy is ascended. The lowest level $\mathcal{L}_l$, is the input level and the highest level, $n_{levels-1}$, which usually contains only one node, is the output level. Those levels and nodes that exist in between the input and output levels are called intermediate. When an HTM is exploited for visual inference, as is the case in this dissertation, the input level typically has a retinotopic mapping of the input. Each input node is connected to one pixel of the input image and spatially close pixels are connected to spatially close nodes. Refer to fig. 3.1 for a graphical example of an HTM.

### 3.2.1 Information Flow

As already mentioned, in an HTM there is a bidirectional information flow. Belief propagation is used to send messages and information both up (feed-forward) and down (feedback) the hierarchy as new evidence is presented to the network. The notation used here for belief propagation (fig. 3.2.a) closely follows Pearl [44] and is adapted to HTMs by George [38]:

- Evidence that comes from below is denoted $e^-$. In visual inference this corresponds to an image or video frame presented to level $\mathcal{L}_0$ of the network.

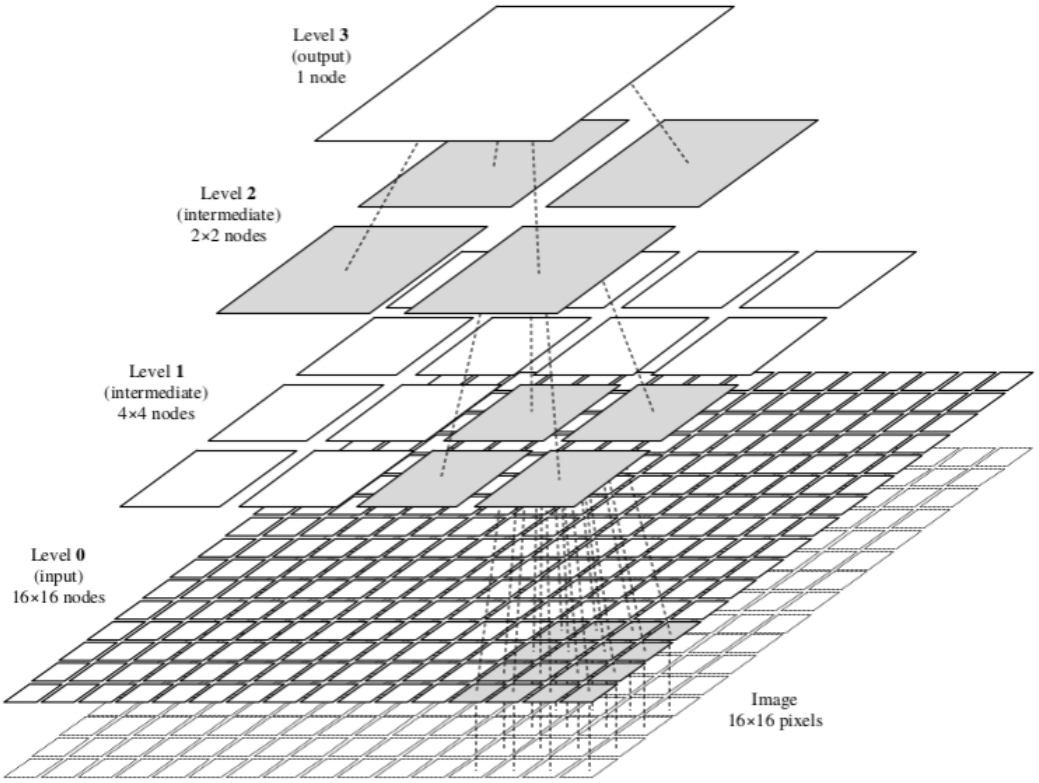


FIGURE 3.1: HTM hierarchical tree-shaped structure. An example architecture for processing 16x16 pixels images [11].

- Evidence coming from the top is denoted $e^+$ and can be seen as contextual information. This can for instance come from another sensor modality or the absolute knowledge of the supervisor training the network.
- Feed-forward messages sent up the hierarchy are denoted $\lambda$ and feedback messages flowing down are denoted $\pi$.
- Messages entering and leaving a node from below are denoted $\lambda^-$ and $\pi^-$ respectively, relative to that node. Following the same notation as for the evidence, messages entering and leaving a node from above are denoted $\lambda^+$ and $\pi^+$.

When an HTM is meant as a classifier, the feed-forward message of the output node is the posterior probability that the input $e^-$ belongs to one of the problem classes. This posterior is denoted $P(w_i|e^-)$ where $w_i$ is one of $n_w$ classes.

### 3.2.2 Internal Node Structure and Pre-training

Since the input level does not need any training and it just forwards the input, HTM training is performed level by level, starting from the first intermediate level. Intermediate levels training is unsupervised and the output level training is supervised. For a detailed description, including algorithm pseudocode, the reader should refer to [11]. For

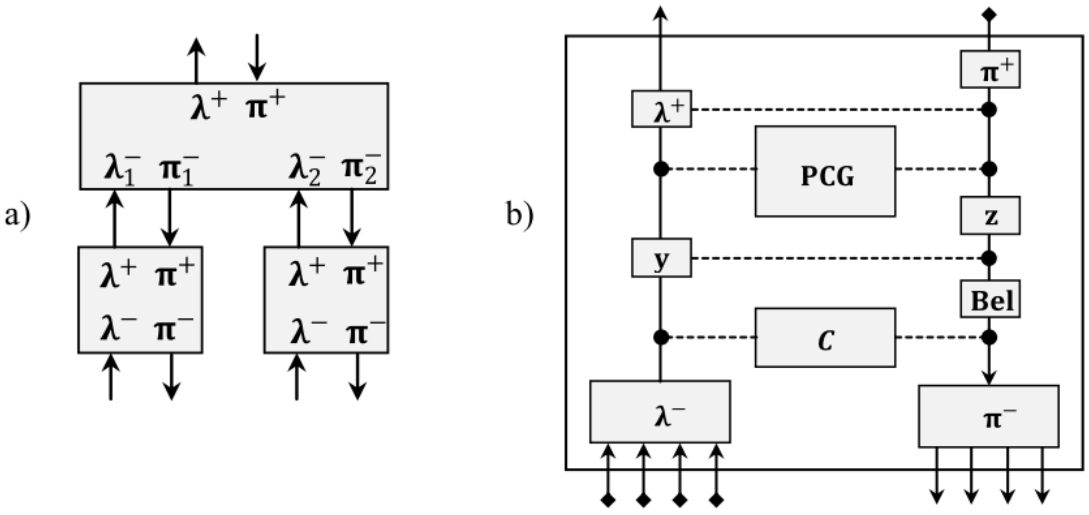


FIGURE 3.2: a) Notation for message passing between HTM nodes. b) Graphical representation of the information processing within an intermediate node [15].

every intermediate node (fig. 3.2.b), a set $C$, of so called coincidences and a set, $G$, of coincidence groups, have to be learned. A *coincidence*, $c_i$, is a vector representing a prototypical activation pattern of the node's children. For a node in $\mathcal{L}_0$, with input nodes as children, this matches to an image patch of the same size as the receptive field of the node. For nodes higher up in the hierarchy, with intermediate nodes as children, each element of a coincidence, $c_i[h]$ is the index of a coincidence group in child $h$. Coincidence groups (or *temporal groups*) are clusters of coincidences that are likely to originate from simple variations of the same input pattern. Coincidences found in the same group can be spatially dissimilar but likely to be found close in time when a pattern is smoothly moved through the receptive field of the node. Exploiting the temporal smoothness of the input and clustering the coincidences accordingly, invariant representations of the input space can be learned [38]. The assignment of coincidences to groups within each node is encoded in a probability matrix $PCG$. Each element $PCG_{ji} = P(c_j|g_i)$ stands for the likelihood that a group, $g_i$, is activated given a coincidence $c_j$. These probability values are the elements that will be manipulated to incrementally train a network whose coincidences and groups have previously been learned and fixed. The output node does not have groups but only coincidences. Instead of memorizing groups and group likelihoods it stores a probability matrix $PCW$, whose elements $PCW_{ji} = P(c_j|w_i)$ represents the likelihood of class $w_i$ given the coincidence $c_j$. This is learned in a supervised fashion by counting how many times every coincidence is the most active one in the context of each class. The output node also keeps a vector of class priors, $P(w_i)$ used to calculate the final class posterior.

### 3.2.3 Feed-Forward Message Passing

Inference in an HTM is conducted through feed-forward belief propagation (see [11]). A degree of certainty over each of the $n_c$ coincidence in the node is computed when a node receives a set of messages from its $m$ children, $\lambda^- = \{\lambda_1^-, \lambda_2^-, ..., \lambda_m^-\}$. This quantity is represented by a vector $y$ and can be seen as the activation of the node coincidences. The degree of certainty over coincidence $i$ is:

$$y[i] = \alpha \cdot p(e^-|c_i) = \begin{cases} e^{-(\|c_i - \lambda\|^2/\sigma^2)} & if\ node\ level = 1 \\ \prod_{j=1}^{m} \lambda_j^-[c_i[j]] & if\ node\ level > 1 \end{cases} \tag{3.1}$$

where $\alpha$ is a normalization constant, and $\sigma$ is a parameter controlling how rapidly the activation level decays when $\lambda^-$ deviates from $c_i$. If the node is an intermediate node, it then computes its feed-forward message $\lambda^+$ (which is a vector of length $n_g$) and is proportional to $p(e^-|G)$ where $G$ is the set of all coincidence groups in the node and $n_g$ the cardinality of $G$. Each component of $\lambda^+$ is

$$\lambda^+[i] = \alpha \cdot p(e^-|g_i) = \sum_{j=1}^{n_c} PCG_{ji} \cdot y[j] \tag{3.2}$$

where $n_c$ is the number of coincidences stored in the node. The feed-forward message from the output node, that is the network output, is the posterior class probability and is computed in the following way:

$$\lambda^+[c] = P(w_c|e^-) = \alpha \sum_{j=1}^{n_c} PCW_{jc} \cdot P(w_c) \cdot y[j] \tag{3.3}$$

where $\alpha$ is a normalization constant such that $\sum_{c=1}^{n_w} \lambda^+[c] = 1$.

### 3.2.4 Feedback Message Passing

The top-down information flow is used to give contextual information about the observed evidence. Each intermediate node combines top-down and bottom-up information to consolidate a posterior belief in its coincidence-patterns [38]. Given a message from the parent $\pi^+$, the top-down activation of each coincidence, $z$, is

$$z[i] = \alpha p(c_i|e^+) = \sum_{k=1}^{n_g} PCG_{ik} \cdot P(w_c) \cdot \frac{\pi^+[k]}{\lambda^+[k]} \tag{3.4}$$

The belief in coincidence $i$ is then given by:

$$Bel[i] = \alpha P(c_i|e^-, e^+) = y[i] \cdot z[i] \tag{3.5}$$

The message sent by an intermediate node (belonging to a level $\mathcal{L}_{\langle}$, $h > 1$) to its children, $\pi^-$, is computed using this belief distribution. The $i^{th}$ component of the message to a specific child node is:

$$\pi^-[i] = \sum_{j=1}^{n_c} I_{c_j}(g_i^{(child)}) \cdot Bel[j] = \sum_{j=1}^{n_c} \sum_{k=1}^{n_g} I_{c_j}(g_i^{(child)}) \cdot y[j] \cdot PCG_{jk} \cdot \frac{\pi^+[k]}{\lambda^+[k]} \tag{3.6}$$

where $I_{c_j}(g_i^{(child)})$ is the indicator function defined as

$$I_{c_j}(g_i^{(child)}) = \begin{cases} 1 & if\ group\ of\ g_i^{(child)}\ is\ part\ of\ c_j \\ 0 & otherwise \end{cases} \tag{3.7}$$

The top-down message sent from the output node is computed in a similar way:

$$\pi^-[i] = \sum_{c=1}^{n_w} \sum_{j=1}^{n_c} I_{c_j}(g_i^{(child)}) \cdot y[j] \cdot PCW_{jc} \cdot P(w_c|e^+) \tag{3.8}$$

Equations 3.6 and 3.8 will be important when, in the next section, it will be shown how to incrementally update the $PCG$ and $PCW$ matrices to produce better estimates of the class posterior given some evidence from above.

### 3.2.5 HTM Supervised Refinement

This section introduces a new way to optimize an already trained HTM originally crafted in [15] [45]. The algorithm, called HSR (Htm Supervised Refinement) shares many features with the traditional back-propagation used to train multilayer perceptrons introduced in chapter 1. It is inspired by weight fine-tuning methods applied to other deep belief architectures [33]. It takes advantage of the belief propagation equations presented above to propagate an error message from the output node back through the network. This enables each node to locally update its internal probability matrix in a way that minimizes the difference between the estimated class posterior of the network and the posterior given from above, by a supervisor. The goal is to minimize the

expected quadratic difference between the network output posterior given the evidence from below, $e^-$, and the posterior given the evidence from above, $e^+$ . For this purpose empirical risk minimization is employed resulting in the following loss function:

$$L(e^-, e^+) = \frac{1}{2}\sum_{c=1}^{n_w}(P(w_c|e^+) - P(w_c|e^-))^2 \tag{3.9}$$

where $n_w$ is the number of classes, $P(w_c|e^+)$ is the class posterior given the evidence from above, $P(w_c|e^-)$ is the posterior produced by the network using the input as evidence (i.e., inference). The loss function is also a function of all network parameters involved in the inference process. In most cases $e^+$ is a supervisor with absolute knowledge about the true class $w_{c^*}$, thus $P(w_{c^*}|e^+) = 1$. To minimize the empirical risk, first of all the direction in which to alter the node probability matrices is found. This is done in order to decrease the loss and then apply gradient descent.

### Output Node Update

For the output node which does not memorize coincidence groups, probability values stored in the $PCW$ matrix are updated through the gradient descent rule:

$$PCW'_{ks} = PCW_{ks} - \eta\frac{\partial L}{\partial PCW_{ks}} \qquad k = 1...n_c, s = 1...n_w \tag{3.10}$$

where $\eta$ is the learning rate. The negative gradient of the loss function is given by:

$$\begin{aligned}\frac{\partial L}{\partial PCW_{ks}} &= \frac{1}{2}\sum_{c=1}^{n_w}\frac{\partial}{\partial PCW_{ks}}(P(w_c|e^+) - P(w_c|e^-))^2 \\ &= \sum_{c=1}^{n_w}(P(w_c|e^+) - P(w_c|e^-))\frac{\partial P(w_c|e^-)}{\partial PCW_{ks}}\end{aligned} \tag{3.11}$$

which can be shown (see Appendix A of [45] for a derivation) to be equivalent to:

$$\frac{\partial L}{\partial PCW_{ks}} = y[k]\cdot Q(w_s) \tag{3.12}$$

$$Q(w_s) = \frac{P(w_s)}{p(e^-)}(P(w_s|e^+) - P(w_s|e^-) - \sum_{i=1}^{n_w} P(w_i|e^-)(P(w_i|e^+) - P(w_i|e^-))) \tag{3.13}$$

where $p(e^-) = \sum_{i=1}^{n_w}\sum_{j=1}^{n_c} \cdot y[j] \cdot PCW_{ji} \cdot P(w_i)$. We call $Q(w_s)$ the error message for class $w_s$ given some top-down and bottom-up evidence.

### Intermediate Nodes Update

For each intermediate node probability values in the $PCG$ matrix are updated through the gradient descent rule:

$$PCG'_{pq} = PCG_{pq} - \eta\frac{\partial L}{\partial PCG_{pq}} \qquad p = 1...n_c, q = 1...n_g \tag{3.14}$$

For intermediate nodes at level $\mathcal{L}_{n_{levels}-2}$ (the last before the output level) it can be shown (Appendix B of [45]) that:

$$-\frac{\partial L}{\partial PCG_{pq}} = y[p] \cdot \frac{\pi_Q^+[q]}{\lambda^+[q]} \tag{3.15}$$

where $\pi_Q^+$ is the child portion of the message $\pi_Q^-$ sent from the output node to its children, but with $Q(w_s)$ replacing the posterior $P(w_s|e^+)$ (compare Eqs. 15 and 8):

$$\pi_Q^+[q] = \sum_{c=1}^{n_w}\sum_{j=1}^{n_c} I_{c_j}(g_q^{(child)}) \cdot y[j] \cdot PCW_{jc} \cdot Q(w_c) \tag{3.16}$$

Finally, it can be shown that this generalizes to all levels of an HTM, and that all intermediate nodes can be updated using messages from their immediate parent. The derivation can be found in Appendix C of [45]. In particular, the error message from an intermediate node (belonging to a level $\mathcal{L}_h, h > 1$) to its child nodes is given by:

$$\pi_Q^+[q] = \sum_{t=1}^{n_c}\sum_{f=1}^{n_g} I_{c_j}(g_q^{(child)}) \cdot PCG_{tf} \cdot \frac{\partial}{\partial PCG_{tf}}$$
$$= \sum_{t=1}^{n_c}\sum_{f=1}^{n_g} I_{c_t}(g_q^{(child)}) \cdot PCG_{tf} \cdot y[t] \cdot \frac{\pi_Q^+[f]}{\lambda^+[f]} \tag{3.17}$$

These results allow us to define an efficient and elegant way to adapt the probabilities in an already trained HTM using belief propagation equations.

### 3.2.6 HSR algorithm

A batch version of HSR algorithm is provided in the pseudo-code in Algorithm 3.
By updating the probability matrices for every training example, instead of at the end of the presentation of a group of patterns, an online version of the algorithm is obtained. In the experimental sections only the batch version of HSR has been used. In many cases it is preferable for the nodes in lower intermediate levels to share memory, so called *node sharing*. This speeds up training and forces all the nodes of the level to respond in the same way when the same stimulus is presented in different places in the receptive field. For a level operating in node sharing, $PCG$ update (eq. 3.14) must be performed only for the master node.

## 3.3 HTM in object recognition

Although HTM is still in its infancy, in the literature, HTM has been already applied to different image dataset for object recognition tasks. In [11] SDIGIT, PICTURE and USPS were used. These three datasets constitute a good benchmark to study invariance, generalization and robustness of a pattern classifier. However, in all the three cases the patterns are small black-and-white or grayscale images (32x32 or smaller). Nonetheless, HTM has been already applied with success to object recognition problems with larger color images see ([46] [47]).

**Algorithm 3** HTM Supervised Refinement

**procedure** HSR($S$)
  **for** each training example in $S$ **do**
    Present the example to the network and do inference (eqs. 1,2,3)
    Accumulate values $(\partial L)/(\partial PCW_{ks})$ for the output node (eqs. 11,12)
    Compute the error message $\pi_Q^-$
    **for** each child of the output node **do**
      BackPropagate(child, $\pi_Q^+$(b) (See function below)
    **end for**
    Update $PCW$ by using accumulated $(\partial L)/(\partial PCW_{ks})$ (Eq. 10)
    Renormalize $PCW$ such that for each class $w_s$, $\sum_{k=1} n_c PCW_{ks} = 1$
    **for** each intermediate node **do**
      Update $PCG$ by using accumulated $(\partial L)/(\partial PCG_{pq})$
      Renormalize $PCG$ such that for each group $g_q$, $\sum_{p=1} n_c PCG_{pq} = 1$
    **end for**
  **end for**
**end procedure**

**procedure** BackPropagate(node, $\pi_Q^+$)
  Accumulate $(\partial L)/(\partial PCG_{pq})$ values for the node (eq. 14)
  **if** *nodelevel* $> 1$ **then**
    Compute the error message $\pi_Q^-$ (eq. 16)
    **for** each child of node **do**
      BackPropagate(child, $\pi_Q^+$)
    **end for**
  **end if**
**end procedure**

# 4

# NORB-Sequences: A new benchmark for object recognition

*"If we want machines to think, we need to teach them to see."*

– Fei-Fei Li, *Stanford Computer Vision Lab*

Over the years different benchmarks arose in order to evaluate the accuracy and the capacity of different pattern recognition algorithms, but, most of them, were not explicit designed for recognizing objects inside image sequences or videos. Investigating pattern recognition algorithms on videos is interesting because it is much more natural and similar to the human visual recognition. Moreover, it is easier to manage ambiguous cases taking advantages of unsupervised learning exploiting temporal continuity. However, as a matter of fact, collecting video that are simple but general enough for the state-of-the-art object recognition algorithms is not straightforward. In this chapter a new benchmark for object recognition in image sequences is proposed. It is based on the New York University Object Recognition Benchmark (NORB) [16]. This is because, instead of creating a new benchmark from scratch, we think it would be better to start from a well-known and commonly accepted dataset. In the following sections, The NORB dataset is summarized and the new benchmark presented.

## 4.1 The small NORB dataset

Many object detection and recognition systems described in the literature have relied on many different non-shape related clues and various assumptions to achieve their goals. Authors have advocated the use of color, texture, edge information, pose-invariant

feature histograms etc... On the contrary, learning methods operating on raw pixels or low-level local features had been quite successful for such applications as face detection [48] [49], but, until the early 2000s, they had not been applied successfully to shape-based, pose-invariant object recognition. One of the goal addressed in this dissertation is also to endorse methods based on global templates over methods based on local features. The NORB dataset is so valuable because in it the shape of the object is the only useful and reliable clue, while all the other parameters that affect the appearance are subject to variation. These parameters: viewing angles (pose), lighting condition, position in the image plane, scale, image-plane rotation, surrounding objects, background texture, contrast, luminance, and camera settings (gain and white balance). Potential clues whose impact has been eliminated include: color (all images were grayscale), and object texture (objects were painted with a uniform color). For specific object recognition tasks, the color and texture information may be helpful, but for generic shape recognition tasks the color and texture information can be only distracting. The image acquisition setup was deliberately designed to reflect real imaging situations. By preserving natural variabilities and eliminating irrelevant clues and systematic biases, the main aim of the original work was to produce a benchmark in which no hidden regularity can be used, which would unfairly advantage some methods over others. While several datasets of object images have been made available in the past [50] [51] [52], NORB is considerably larger than those datasets, and offers more variability, stereo pairs, and the ability to composite the objects and their cast shadows onto diverse backgrounds. The NORB benchmark has been created in 2004 in the *Computational and Biological Learning Lab* directed by Y. LeCun, one of the greatest pioneers of deep learning and father of the CNN algorithm.

### 4.1.1 Dataset details

The dataset collects images of 50 different toys shown in fig. 4.1. The collection consists of 10 instances of 5 generic categories: four-legged animals, human figures, airplanes, trucks, and cars. All the objects were painted with a uniform bright green. The uniform color ensured that all irrelevant color and texture information was eliminated. 1,944 stereo pairs were collected for each object instance: 9 elevations (30, 35, 40, 45, 50, 55, 60, 65, and 70 degrees from the horizontal), 36 azimuths (from 0 to 350° every 10°), and 6 lighting conditions (various on-off combinations of the four lights). A total of 194,400 RGB images at 640x480 resolution were collected (5 categories, 10 instances, 9 elevations, 36 azimuths, 6 lightings, 2 cameras) for a total of 179 GB of raw data. Note that each object instance was placed in a different initial pose, therefore "0 degree angle" may mean "facing left" for one instance of an animal, and "facing 30 degree

right" for another instance. Then, training and testing samples were generated so as to carefully avoid and remove any potential bias in the data that might make the task easier than it would be in realistic situations. The object masks and their cast shadows were extracted from the raw images. A scaling factor was determined for each of the 50 object instances by computing the bounding box of the union of all the object masks for all the images of that instance. The scaling factor was chosen in such a way that the largest dimension of the bounding box was 80 pixels. This removed the most obvious systematic bias caused by the variety of sizes of the objects (e.g. most airplanes were larger than most human figures in absolute terms). The segmented and normalized objects were then composited, along with their cast shadows, in the center of various 96x96 pixel background images. The original dataset is released and freely downloadable in two main sets: the training set and the test set, both in the *MATLAB*[1] format. They have the same size and are composed of five different instances for each class as reported in fig. 4.1. In this way, in order to obtain a good level of accuracy, the shape of an object has to be well generalized.

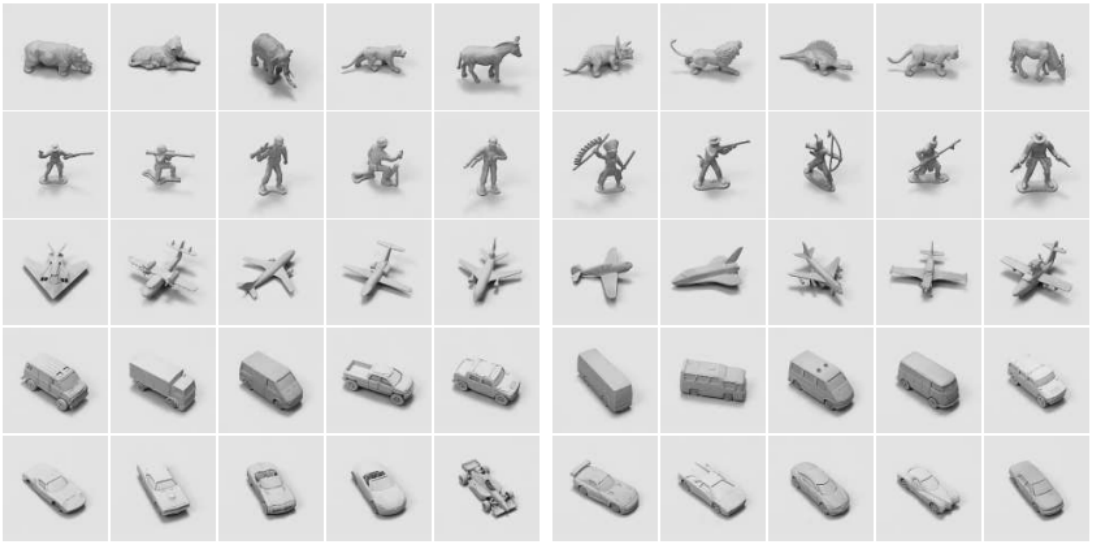

FIGURE 4.1: The 50 object instances in the NORB dataset. The left side contains the training instances and the right side the testing instances for each of the 5 categories originally used in [16].

## 4.2 NORB-Sequences design

The main objective of the new benchmark is to give researchers the possibility to test their algorithm implementations on a standard reference regarding object recognition in image sequences. The design of such a benchmark should be in a way that enables

[1]MATLAB (matrix laboratory) is a multi-paradigm numerical computing environment and fourth-generation programming language

improvements in resolving a task that is neither too much simple or too much difficult. In this section the main design guidelines are provided. Considering the entire pool of images collected in NORB, a sequence is defined as a certain number of images for which two specific properties hold:

- *All the images in the sequence contain the same object instance*
- *Each image inside the sequence differs from the previous and the following ones only for one unit in one single dimension (of course the first and the last images only from the following and the previous ones respectively).*

We consider elevations, azimuth and lighting as different dimensions where elevations has 9 unit (30, 35, 40, 45, 50, 55, 60, 65, and 70 degrees from the horizontal), azimuth has 36 (from 0 to 350° every 10°) and lighting has 6 (various on-off combinations of the four lights) as explained in the previous section. In this way, in the sequence is possible to simulate a camera moving around an object including changes in the surround lighting. However, these specifics are not sufficient to smooth a sequence enough in order to obtain the desired outcome. In this regard, it has been decided to create the sequences based on specific probabilities:

- *elevationProb*: The probability that the variation between one frame and the following one is related to the elevation;
- *azimuthProb*: The probability that the variation between one frame and the following one is related to the azimuth.
- *lightingProb*: The probability that the variation between one frame and the following one is related to the lighting.
- *flipProb*: The probability of changing the direction of the variation (i.e increasing or decreasing of one unit the current selected dimension).

Hence, for each sequence and for each dimension an initial change of direction is chosen, then spatially near images are selected according to these probabilities. Then, the initial change of direction can be inverted only respecting the flip probabilities (used after a dimension change is chosen) or if an upper or lower limit in a particular dimension is reached. Moreover, to make the test set sufficiently different from the training set, a specific *minimum distance* parameter has been introduced. Basically, for each image in the test sequences the following property must hold:

- *The distance between each possible combination (train, test) images must be greater or equal than the fixed minimum distance.*

The distance is computed as the sum of the differences for each dimension between the two images.
In order to create the whole training and test set, for each class and for each object a certain number of sequences are generated. This number can differ between the training and test set.
Finally, It has been decided to enrich the benchmark with a baseline classifier based on the KNN algorithm [53].

FIGURE 4.2: Example sequence of ten images.

## 4.3 Implementation

Even if an already generated dataset of sequences has been proposed, the main purpose of the implementation is to create a rich and flexible benchmark in which it is possible to tune different parameters in order to create image sequences that are suitable for almost any recognition task. The language chosen to develop the software is Java. This is because a great portability, good maintenance properties and fast processing are needed having to deal with images. The current implementation is composed of 10 Java source files listed above:

1. NorbConverter.java
2. NorbSampler.java
3. NorbSamplerTrain.java
4. NorbSamplerTest.java
5. NorbSeqExplorer.java
6. NorbKNN.java
7. KNNTestSeries.java
8. SeqVerifier.java
9. CreateDatasets.java
10. NorbCreator.java

**NorbConverter** *NorbConverter.java* deals with the conversion of the original images from matlab to bitmap format eventually scaling and dividing them in different class subfolders.

**NorbSampler** *NorbSampler.java* is an abstract class that is inherited by two subclasses: *NorbSamplerTrain.java* and *NorbSamplerTest.java*. It implements the basic functionalities offered by a sequences generator, regardless of whether they will be in the train or in the test set.

**NorbSamplerTrain** NorbSamplerTrain.java deals with the sampling of the training set. First, it is assumed that all the NORB images are collected into a single folder containing 5 numbered class subfolders. For each class instance in these directories one or more image sequences can be created. The sequences are generated based on the parameters already discussed above. In the end, all the parameters and image sequences are written in a single configuration file.

```
Config Type: Train
-------------------------
nClass: 2
nObjxClass: 10
nSeqxObj: 1
ElevationProb: 0.3
AzimuthProb: 0.3
LightingProb: 0.2
FlipProb: 0.2
seqLen: 20
seed: 1234
-------------------------
```

FIGURE 4.3: An header example contained in train configuration file.

The output configuration file format is straightforward. In fig. 4.3 you can see an example of header file containing a summary of the parameters used. In this case, only two classes are considered and for each object in each class, only one sequence of length 20 is generated. The last parameter we have not previously discussed is the seed for the random number generator. Using the seed as a parameter is important to be able to generate the same sequences with the same parameters. Eventually, after a blank line, all the sequences with their respective headers are written. These header contains the indices referring the class, the object and sequence, as can be seen in fig. 4.4. All the images in each sequence are listed with their unique (within their class) file name.

```
---------------
Class:    0
Object:   0
Sequence: 0
---------------
00_02_10_05.bmp
00_03_10_05.bmp
00_03_12_05.bmp
00_04_12_05.bmp
00_05_12_05.bmp
00_06_12_05.bmp
00_07_12_05.bmp
...
```

FIGURE 4.4: An example sequence with its own header.

**NorbSamplerTest** *NorbSamplerTest.java* deals with the test sampling. By reading the data written in the configuration file of the training set, it generates for each instance a number of sequences that can differ from the training set. However, each image inside these sequences has to respect the minimum distance parameter we already discussed in the previous section. Also in this case, all the parameters, sequences and their average distance from the training set, are written within a single configuration file. The test configuration file format is exactly the same of the training configuration file. In fig. 4.5 an header example is reported.

```
Config Type: Test
-------------------------
nClass: 2
nObjxClass: 10
nSeqxObj: 1
ElevationProb: 0.3
AzimuthProb: 0.3
LightingProb: 0.2
FlipProb: 0.2
seqLen: 20
seed: 1234
minDistance: 3
-------------------------
```

FIGURE 4.5: An header example contained in the test configuration file

**NorbSeqExplorer** NorbSeqExplorer.java is a image sequence browser that aims to provide an easy graphical interface for reading the config files created from NorbSamplerTrain.java and NorbSamplerTest.java. In Figure 4.6 a screenshot of the browser is reported. Through the GUI it is possible to specify the configuration file to read (i.e. train or test) and the root directory in which all the images are located. After that, it is possible to modify directly the input boxes to choose the class, the instance and the sequence. Initially, the first image of the sequence 0 of the instance 0 and class 0 is shown. Then, it is possible to navigate the sequence using the buttons “prev” and “next”.

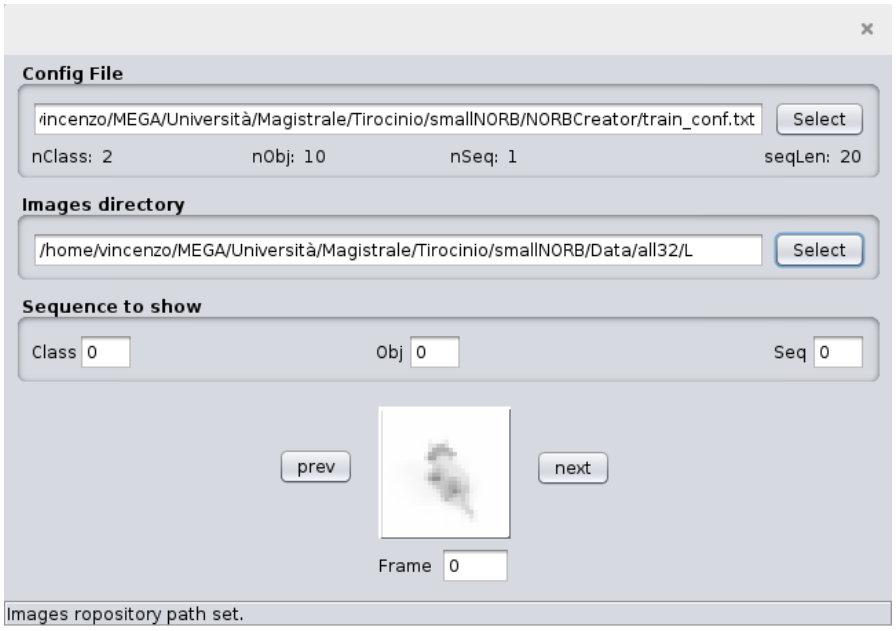


FIGURE 4.6: Sequences browser GUI.

**SeqVerifier** *SeqVerifier.java* is a class that aims to verify the correctness of the generated sequences. In particular it bothers to verify if the sequences are actually spatially sequential and if the distance of the test set from the training set is respected. For each sequence that is generated, the verification is automatically performed, then it is impossible to perform experiments on malformed sequences.

**NorbKNN** *NorbKNN.java* is a basic class that implements a KNN classifier for images contained in a sequence. In particular, it is based on the KNN implementation provided by the well-known *weka*[2] library. For classifying an image in the test sequence, then, It is also possible to choose merging the confidence levels with the images previously classified in the same sequence.

**KNNTestSeries** *KNNTestSeries.java* is a small utility to help automating the accuracy tests by varying several parameters like the sequences length or the min-distance of the test set from the training set.

**CreateDataset** *CreateDataset.java* is a small utility that simply translates the sequences from a text format to an actual collection of images. This tool is useful when you want to perform experiments where the images order inside the sequences is not considered.

[2] *Weka* is a collection of machine learning algorithms for data mining tasks entirely written in Java.

**NorbCreator** *NorbCreator.java* is a command line interface and utility for the automation of all the features offered by the benchmark. Starting from a single and complete configuration file it is possible to convert the images, create the sequences and eventually visualize them with a single command. In fig. 4.7 a configuration file example is reported. It is pretty self explanatory, since the same format and parameters have been already discussed before (in fact, these parameters are then shown again in the training and conf file for completeness and modularity).

```
#########################
#      CONFIG FILE      #
#########################

#########################
#   CONVERSION PARAMS   #
#########################
matlabFile: ../Data/Matlab/smallnorb-5x46789x9x18x6x2x96x96-training-
destDir: ../Data/all32
convert(yes/no): no
inputWidth: 96
inputHeight: 96
scaleFactor: 3
#########################

#########################
#     COMMON PARAMS     #
#########################
imagesRepo: ../Data/all32/L
nClass: 2
nObjxClass: 10
elevationProb: 0.55
azimuthProb: 0.35
lightingProb: 0.1
flipProb: 0.05
seqLen: 40
#########################

#########################
#     TRAIN PARAMS      #
#########################
fileName: train_conf.txt
nSeqxObj: 1
seed: 1
#########################

#########################
#      TEST PARAMS      #
#########################
fileName: test_conf.txt
nSeqxObj: 1
seed: 2
minDistance: 0
#########################

#########################
#      END CONFIG       #
#########################
```

FIGURE 4.7: A complete configuration file example.

| **Conf. Name** | **Type** | **Num. images** | **min-distance** |
|---|---|---|---|
| train_conf.txt | Train | 5x10x10x20 = 10000 | N.D. |
| test_conf_1.txt | Test | 5x10x10x20 = 10000 | 1 |
| test_conf_2.txt | Test | 5x10x10x20 = 10000 | 2 |
| test_conf_3.txt | Test | 5x10x10x20 = 10000 | 3 |
| test_conf_3.txt | Test | 5x10x10x20 = 10000 | 4 |

TABLE 4.1: The different configuration files of the standard distribution prototype.

## 4.4 Standard distribution

Even if the benchmark implementation is flexible and simple enough to generate sequences directly and for any purpose, it has been decided to release in the future a standard distribution for whom doesn't want to download the Java code or the jar package but wants to use an already generated dataset. In this section, a first prototype is discussed. It contains a single training set and 4 different test sets. They differ from each other for the minimum distance used: 1, 2, 3 and 4 respectively. This is the dataset on which first sequences experiments described in the following chapter have been performed.
In order to achieve the smoothest possible sequences, for the standard distribution it has been experimentally agreed about the following probabilities:

- *elevationProb*: 0.35
- *azimuthProb*: 0.55
- *lightingProb*: 0.1
- *flipProb*: 0.05

The first three, of course, need to sum to one. For the standard distribution of the benchmark it has been decided to use for both the training and test sets the following parameters:

- number of classes: 5
- Number of instances for class: 10
- Number of sequence for instance: 10
- Sequences length: 20

Finally, for the training set a seed of 1 has been chosen and for a seed of 2 for the remaining test sets.

## 4.5 KNN baseline: first experiments

In this section first experiments to certify the validity of the baseline are reported. Using *KNNTestSeries.java*, it has been possible to perform several runs and collect averaged results in an automated way. Even if not comprehensive, this has given us an advantageous understanding of the goodness of our approach. The parameters used for these two different batch of experiments are listed below:

- Number of classes: 5
- Number of objects: 10
- Number of sequences for each object: 1
- Number of minimum-distances used: 3 (1, 2, 3)
- Number of sequence lengths: 3 (15, 30, 45)
- Number of tests to average for each parameters set (different seq. seed): 5
- Error: Standard deviation
- K in the KNN algorithm: 1
- KNN distance measure: Euclidean

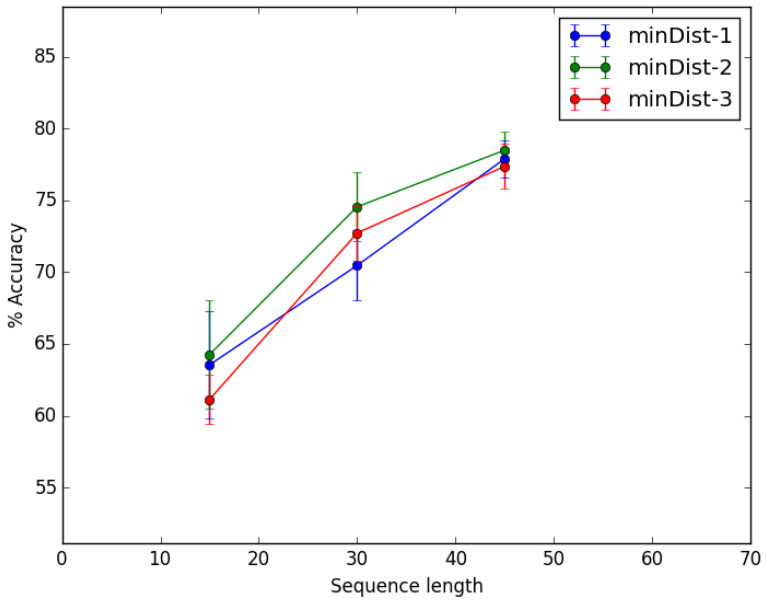


FIGURE 4.8: KNN experiments accuracy results

For the first batch, all the images contained in all the sequences in the training set are considered as equal. In the same way, each image contained in all the test sequences is classified as it was a single image. Practically speaking in this case the sequence factor is completely ignored. The steps to perform the classification with k equals one, ignoring

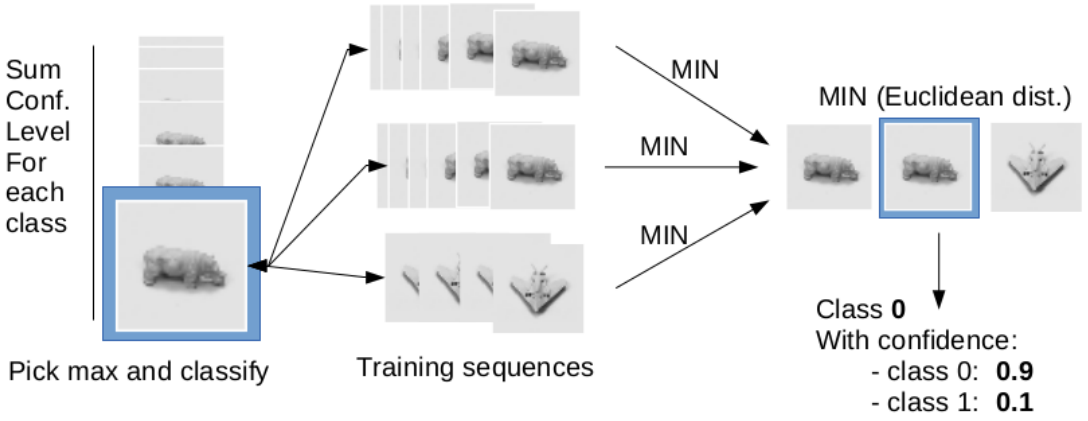


FIGURE 4.9: Explanatory example of how the KNN algorithm works. In this case, K is equal to one and the number of classes is two.

the sequences split are straightforward: first, the euclidean distance between each pair of images is computed, then, for each image in the test set, the class of the image in the training set that minimize this distance is selected. In fig. 4.8 accuracy results along with the standard deviation errors are plotted. Of course, as the length of the sequences increases, the accuracy improves. This is directly and only correlated to the number of the images present in the training set. The minimum distance factor does not influence significantly the results essentially for two reasons. First because the difference between the minimum distances is not substantial and second because there is no upper bound for the distances. The average distance is indeed very similar among the test sets. In the end about 78% of accuracy is reached from each test set.

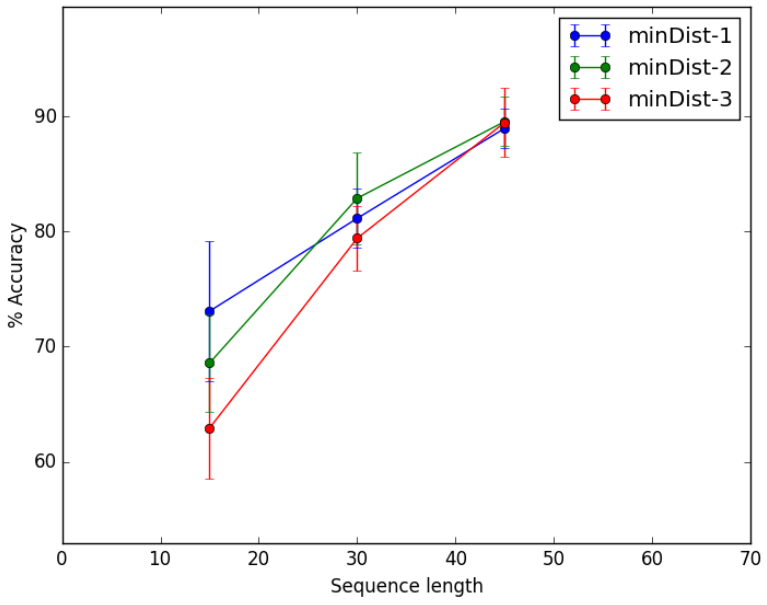


FIGURE 4.10: KNN experiments accuracy results with confidence levels merging

In the second batch of experiments, it has been tried to take advantage of the sequence factor by merging the confidence levels of the previously classified images in the sequence. In fig. 4.9 a simple graphical explanation is pictured. This is not very different to what has been done for the first batch of experiments, but in this case we exploit the confidence levels of each class and we sum them going through the sequence, then we pick the class with the maximum value. In this way the classification is more robust and indeed a better accuracy can be reached. In fig. 4.10 the accuracy results along with the standard deviation errors are plotted. In this case the minimum distance factor plays a good role, especially when the sequence length is small. The highest accuracy result in this case is almost 90% that leaves, however, a good room for improvements with different strategies and algorithms.

# 5

# Comparing CNN and HTM: Experiments and results

*"It doesn't matter how beautiful your theory is, it doesn't matter how smart you are. If it doesn't agree with experiment, it's wrong."*

– Richard P. Feynman, *Nobel Prize in Physics* (1965)

In this chapter, results from several experiments comparing CNNs and HTMs are reported. The main aim is to demonstrate that with a lower quantity of data, HTM can outperform the classical CNN approach for object recognition and can remain comparable when more data are available. In the first sections few reflection about the experiments design are reported. In the following sections, instead, some details about the HTM and CNN implementations are described. Finally, after having introduced the reader to the experiments environment and setups, conclusions are drawn based on the results obtained.

## 5.1 Experiments design

Given the available time, two main experiments have been conducted. The first one is about the plain NORB dataset, the second one about the NORB sequences. In the second experiment the first prototype of the standard distribution reported in the previous chapter has been used, even if not exactly in the task of recognizing objects in sequences, a purpose that will be developed in future works, but not subject of this dissertation. In this way, we can prove the goodness of the HTM approach on a standard benchmark that has consecrated the CNN algorithm and start thinking about the more interesting task of recognize objects inside image sequences or video.

## 5.2 CNN implementation

The CNN implementation wasn't a simple task. These kind of neural networks are difficult to design from scratch and , more importantly, they have a huge quantity of parameters to tune in order to obtain a good accuracy depending on the specific task. Since the first experiment was about the NORB dataset, the details reported in the original paper by Huang and LeCun [16] have been followed:

*"A six-layer net, shown in figure 5, was used in the experiments reported here. The layers are respectively named C1, S2, C3, S4, C5, and output. The C letter indicates a convolutional layer, and the S layer a subsampling layer. C1 has 8 feature maps and uses 5x5 convolution kernels. The first 2 maps take input from the left image, the next two from the right image, and the last 4 from both. S2 is a 4x4 subsampling layer. C3 has 24 feature maps that use 96 convolution kernels of size 6x6. Each C3 map takes input from 2 monocular maps and 2 binocular maps on S2, each with a different combination. S4 is a 3x3 subsampling layer. C5 has a variable number of maps (80 and 100 in the reported results) that combine inputs from all map in S4 through 6x6 kernels. Finally the output layer takes inputs from all C5 maps. The network has a total of 90,575 trainable parameters. A full propagation through the network requires 3,896,920 multiply-adds. The network was trained to minimize the mean squared error with a set of target outputs."*

However, as can be seen, many other details are missing. For example is not clear what activation function was used neither what downsampling type (average, max or sum pooling?) or cost function etc... Hence, to reach the reported accuracy, several experiments have been performed. In the following sections, we will deepen them in great detail. In order to carry out our experiments on a flexible implementation, a *LeNet7*[1] model that can manage binocular or monocular images with two different size (96x96 or 32x32, easily extensible to manage 64x64 too) has been designed. It is a seven layers CNN in which all the feature maps of a particular layer takes input from all the the feature maps of the previous one. This is something different from what has been reported in the original NORB paper but we found no significant differences in terms of accuracy. The first layer is the input layer, then there are C1 and S2, a convolutional layer and a subsampling layer followed by C3 and S4, another convolutional layer and subsampling layer. The sixth layer, C6, is a another convolutional layer and the last one is an output layer. In all the experiments 8, 24 and 100 feature maps for C1, C3 and C5 have been used respectively. For the sake of clearness, let's consider how 32x32 and 96x96 images are processed.

[1] *LeNet7* is the orginal 7-layers CNN architecture which takes its name from whom is condidered the "father" of CNN Yann LeCun.

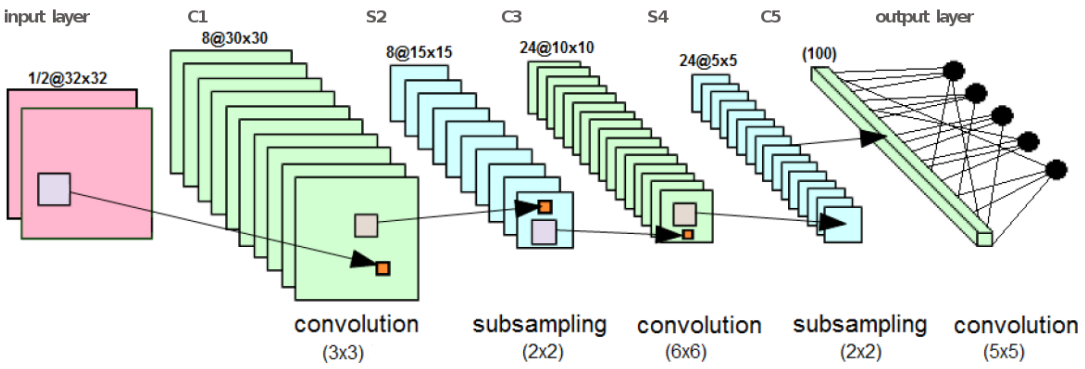


FIGURE 5.1: How (32x32) images are processed in our LeNet7 model. X@YxY stands for X feature maps of size YxY; (ZxZ) stands for the receptive filed or filter of size ZxZ

In fig. 5.1, 32x32 images after a convolution with eight 5x5 filters end up with eight 30x30 images. At level S2, after a 2x2 downsampling they are reduced to 15x15 images. C3 further elaborates them with 24 different kernels to 24 10x10 images. After another downsampling layer they are reduced again by half of their size. The last convolutional layer, eventually, reduces them in 100 values that represent the features selected for the last full-connected layer, the output layer. The same procedure is done for 96x96 images. For futher details the reader can consider fig. 5.2.

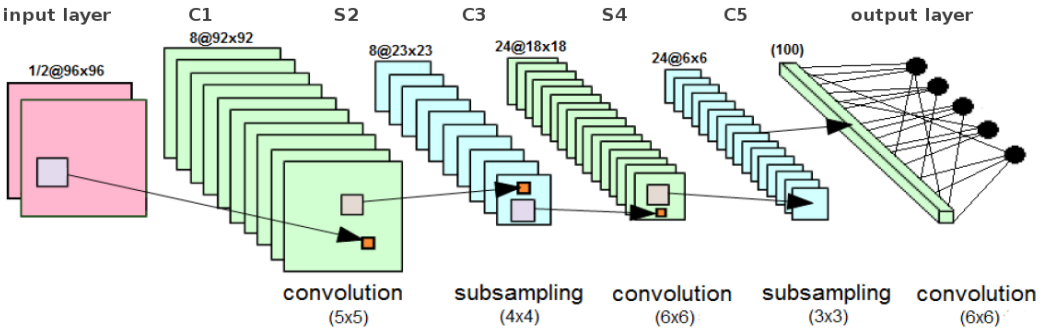


FIGURE 5.2: How (96x96) images are processed in our LeNet7 model. X@YxY stands for X feature maps of size YxY, (ZxZ) stands for the receptive filed or filter of size ZxZ

### 5.2.1 Theano

Theano is a Python library that allows you to define, optimize, and evaluate mathematical expressions, especially the ones with multi-dimensional arrays (numpy.ndarray) [54]. Using Theano it is possible to attain speeds rivaling hand-crafted C implementations for problems involving large amounts of data. It can also surpass C on a CPU by many orders of magnitude by taking advantage of recent GPUs. Theano combines aspects of a computer algebra system (CAS) with aspects of an optimizing compiler. It can also generate customized C code for many mathematical operations. This combination of

CAS with optimizing compilation is particularly useful for tasks in which complicated mathematical expressions are evaluated repeatedly and evaluation speed is critical. For situations where many different expressions are evaluated each once, Theano can minimize the amount of compilation/analysis overhead, but still provide symbolic features such as automatic differentiation.
In Theano there are two ways currently to use a GPUs, one of which only supports NVIDIA cards (CUDA backend) and the other, in development, that should support any OpenCL device as well as NVIDIA cards (GpuArray Backend).
One thing to keep in mind is that the "building blocks" you get in Theano are not ready-made neural network layer classes, but rather symbolic function expressions that are possible to compose into other expressions. The work is made at a slightly lower level of abstraction, but this means there is a lot more flexibility. (That said, if one needs "plug and play" neural networks, he can use *pylearn2* [55] which is built on top of Theano).
Theano was written at the LISA lab to support rapid development of efficient machine learning algorithm and released under a BSD license. For all this features and the great integration with the python scientific Theano seems the natural fit to carry out our deep learning experiments.

### 5.2.2 Lenet7 in Theano

LeNet7 in Theano can be easily implemented using some deep learning libraries already included in the framework or powerful built-in functions. In this dissertation, we certainly can not deepen all the code that has been developed for the purpose but, for the sake of clearness and to show the power of the framework used, let us take a look at the following code:

```
1  import os
2  import sys
3  import time
4  import cPickle
5  import numpy
6  import gzip
7
8  import theano
9  import theano.tensor as T
10 from theano.tensor.signal import downsample
11 from theano.tensor.nnet import conv
```

```
12 from theano.tensor.nnet.neighbours import images2neibs
13
14 from OutputLayer import OutputLayer, load_data
```

As can be seen, after having installed Theano, it is possible to import it as any other library in Python. Then, it is also possible to run the python program as always and Theano will take care of the translation to the C language or CUDA depending on the architecture and its configuration files. All the following code, is based on the assumption that the images are processed in mini-batches. This is because it would be desirable to train the network with stochastic gradient descent, even if no one denies to put all the images in a single batch. A basic convolutional layer, followed by an eventual downsampling layer, then, can be implemented as follows:

```
1  """Conv/Pool Layer of a convolutional network """
2  class LeNetConvPoolLayer(object):
3    def __init__(self, rng, input, filter_shape, image_shape,
      poolsize=(2, 2), pool_type="max"):
4
5    assert image_shape[1] == filter_shape[1]
6    self.input = input
7
8    # there are "num input feature maps * filter height *
      filter width" inputs to each hidden unit
9    fan_in = numpy.prod(filter_shape[1:])
10
11   # each unit in the lower layer receives a gradient from:
      "num output feature maps * filter height * filter width"
      / pooling size
12   fan_out = (filter_shape[0] * numpy.prod(filter_shape[2:])
      / numpy.prod(poolsize))
13
14   # initialize weights with random weights
15   W_bound = numpy.sqrt(6. / (fan_in + fan_out))
16   self.W =
      theano.shared(numpy.asarray(rng.uniform(low=-W_bound,
      high=W_bound, size=filter_shape),
      dtype=theano.config.floatX), borrow=True)
17
```

```
18  # the bias is a 1D tensor -- one bias per output feature
     map
19  b_values = numpy.zeros((filter_shape[0],),
     dtype=theano.config.floatX)
20  self.b = theano.shared(value=b_values, borrow=True)
21
22  # convolve input feature maps with filters
23  conv_out = conv.conv2d(
24   input=input,
25   filters=self.W,
26   filter_shape=filter_shape,
27   image_shape=image_shape
28  )
29
30  #downsample
31  if pool_type == "max":
32  # downsample each feature map individually, using
     maxpooling
33   pooled_out = downsample.max_pool_2d(
34    input=conv_out,
35    ds=poolsize,
36    ignore_border=True
37   )
38
39  elif pool_type == "avg":
40  # downsample using average pooling
41   filter_bank_out = conv_out + self.b.dimshuffle('x', 0,
     'x', 'x')
42   pooled_out = images2neibs(ten4=filter_bank_out,
     neib_shape=poolsize, mode='ignore_borders').mean(axis=-1)
43   new_shape = T.cast(T.join(0,
44     filter_bank_out.shape[:-2],
45     T.as_tensor([filter_bank_out.shape[2]/poolsize[0]]),
46     T.as_tensor([filter_bank_out.shape[3]/poolsize[1]])),
47     'int64')
48   pooled_out = T.reshape(pooled_out, new_shape, ndim=4)
49
50  elif pool_type == 'sum':
51  # downsample using sum pooling
```

```
52      filter_bank_out = conv_out + self.b.dimshuffle('x', 0,
        'x', 'x')
53      pooled_out = images2neibs(ten4=filter_bank_out,
        neib_shape=poolsize, mode='ignore_borders').sum(axis=-1)
54      new_shape = T.cast(T.join(0,
55        filter_bank_out.shape[:-2],
56        T.as_tensor([filter_bank_out.shape[2]/poolsize[0]]),
57        T.as_tensor([filter_bank_out.shape[3]/poolsize[1]])),
58        'int64')
59      pooled_out = T.reshape(pooled_out, new_shape, ndim=4)
60
61    else: #no downsample
62      pooled_out = conv_out
63
64    # add the bias term. Since the bias is a vector (1D
      array), we first reshape it to a tensor of shape (1,
      n_filters, 1, 1). Each bias will thus be broadcasted
      across mini-batches and feature map width & height
65    self.output = T.tanh(pooled_out + self.b.dimshuffle('x',
      0, 'x', 'x'))
66
67    # store parameters of this layer
68    self.params = [self.W, self.b]
```

Hence, a basic many-to-many convolutional layer followed by a subsampling layer has been implemented with few lines of code. Moreover, all three main types of downsampling (max, avg and sum) have been implemented for testing purpose. The activation function used, on the other hand, is always *tanh*. Concerning the cost function, the mean square error is computed as the difference between the actual 5x1 CNN output vectors and the *one-hot* vectors of the training set (i.e. those vectors where all the elements are zero and only the $i^{th}$ element is one, where $i$ is the class of the object inside the image). Then, to build the actual LeNet7 architecture that has been described in the previous section, it is possible to proceed as follows (some variables definition have been omitted for brevity):

```
1  # input layer
2  if(binocular):
3    in_dim = 2
```

```
4  else:
5    in_dim = 1
6
7  layer0_input = x.reshape((batch_size, in_dim, img_dim,
       img_dim))
8
9  # first convolutional/pooling layer:
10 if(img_dim == 32):
11   filter_shape = 3
12   pool_size = (2,2)
13 else:
14   filter_shape = 5
15   pool_size = (4,4)
16
17 c1s2 = LeNetConvPoolLayer(
18         rng,
19         input=layer0_input,
20         image_shape=(batch_size, in_dim, img_dim,
21         img_dim),
22         filter_shape=(nkerns[0], in_dim,
23         filter_shape, filter_shape),
24         poolsize=pool_size,
25         pool_type=pool
26        )
27
28 # second convolutional/pooling layer
29 if(img_dim == 32):
30   img_shape = 15
31   pool_size = (2,2)
32 else:
33   img_shape = 23
34   pool_size = (3,3)
35
36 c3s4 = LeNetConvPoolLayer(
37         rng,
38         input=c1s2.output,
39         image_shape=(batch_size, nkerns[0],
40         img_shape, img_shape),
41         filter_shape=(nkerns[1], nkerns[0], 6, 6),
```

```
42          poolsize=pool_size,
43          pool_type=pool
44         )
45
46 if(img_dim == 32):
47   img_shape = 5
48   filter_shape = 5
49 else:
50   img_shape = 6
51   filter_shape = 6
52
53 # last convolutional layer without pooling
54 c5 = LeNetConvPoolLayer(
55        rng,
56        input=c3s4.output,
57        image_shape=(batch_size, nkerns[1],
58        img_shape, img_shape),
59        filter_shape=(nkerns[2], nkerns[1],
60        filter_shape, filter_shape),
61        poolsize=pool_size,
62        pool_type='no'
63       )
64
65 # the fully-connected output layer, it operates on 2D
     matrices of shape (batch_size, num_pixels) (i.e matrix
     of rasterized images).
66 c5_input = c5.output.flatten(2)
67 output_layer = OutputLayer(input=c5_input, n_in=nkerns[2],
     n_out=n_classes)
68
69 # the cost we minimize during training
70 cost = output_layer.mse(y_matrix)
```

Before dealing with the training procedure we need to implement some functions to evaluate the model and update the weights accordingly:

```
1 # create the functions to compute the mistakes that are
2 # made by the model on the training, test and validation
```

```
3  # sets
4  test_train = theano.function(
5                [index],
6                output_layer.errors(y), givens={
7                x: train_set_x[index * batch_size:
8                (index + 1) * batch_size],
9                y: train_set_y[index * batch_size:
10               (index + 1) * batch_size]
11               }
12              )
13
14 test_model = theano.function(
15               [index],
16               output_layer.errors(y),
17               givens={
18               x: test_set_x[index * batch_size:
19               (index + 1) * batch_size],
20               y: test_set_y[index * batch_size:
21               (index + 1) * batch_size]
22               }
23              )
24
25 validate_model = theano.function(
26                  [index],
27                  output_layer.errors(y),
28                  givens={
29                  x: valid_set_x[index * batch_size:
30                  (index + 1) * batch_size],
31                  y: valid_set_y[index * batch_size:
32                  (index + 1) * batch_size]
33                  }
34                 )
35
36 # create a list of all model parameters to be fit by
37 # gradient descent
38 params = c1s2.params + c3s4.params + c5.params +
39 output_layer.params
40
41 # create a list of gradients for all model parameters
```

```
grads = T.grad(cost, params)

# train_model is a function that updates the model
# parameters by SGD Since this model has many
# parameters, it would be tedious to manually create
# an update rule for each model parameter. We thus
# create the updates list by automatically looping
# over all (params[i], grads[i]) pairs.
l_r = T.scalar('l_r', dtype=theano.config.floatX)

updates = [
           (param_i, param_i - l_r * grad_i)
           for param_i, grad_i in zip(params, grads)
          ]

train_model = theano.function(
               [index, l_r],
               cost,
               updates=updates,
               givens={
               x: train_set_x[index * batch_size:
               (index + 1) * batch_size],
               y_matrix: one_hot_train[index * batch_size:
               (index + 1) * batch_size]
               }
              )
```

As mentioned before, in order to train the network, stochastic gradient descent is used. Without dealing with the entire code base, let us look directly at the heart of the procedure. For each epoch we proceed as follows:

```
for minibatch_index in xrange(n_train_batches):
  iter = (epoch - 1) * n_train_batches + minibatch_index

  if iter % 100 == 0:
    print 'training @ iter = ', iter
    cost_ij = train_model(minibatch_index, learning_rate)

```

```
8   if (iter + 1) % validation_frequency == 0:
9     # compute zero-one loss on validation set
10    validation_losses = [validate_model(i) for i in
11                         xrange(n_valid_batches)]
12    this_validation_loss = numpy.mean(validation_losses)
13
14
15    # if we got the best validation score until now
16    if this_validation_loss < best_validation_loss:
17      #improve patience if loss improvement is good enough
18      if this_validation_loss < best_validation_loss *
19                                  improvement_threshold:
20        patience = max(patience, iter * patience_increase)
21
22        # save best validation score and iteration number
23        best_validation_loss = this_validation_loss
24        best_iter = iter
25
26        # test it on the test set
27        test_losses = [
28                        test_model(i)
29                        for i in xrange(n_test_batches)
30                       ]
31        test_score = numpy.mean(test_losses)
32
33  if patience <= iter:
34    done_looping = True
35    break
```

That is, the model is trained on each mini-batch and, depending on the frequency chosen, it is evaluated on the validation set. Based on the accuracy result and a *patience* paramenter, an early-stopping parameter, *done-looping*, can be set. All the exact parameters used for the CNN experiments will be specified later in section 5.4.

## 5.3 HTM implementation

The HTM implementation was not an objective of the dissertation. An already implemented C# solution was directly e generously provided by Davide Maltoni and the

*Biometric System Lab* (University of Bologna - Dipartimento di Informatica - Scienza e Ingegneria - DISI, Cesena). This implementation includes some of the improvement suggested bu Greg Kochaniack as described in [11], in particular the coincidence buffering techniques.

The HTM architecture which has been used for all the experiments explained in the following sections is composed of 5 layers:

**Input**:

- Nodes: 1024 (32x32)
- Childs: 0x0
- Overlap: 1.00x1.00

**Intermediate 1**:

- Nodes: 169 (13x13)
- Childs: 8x8
- Overlap: 3.50x3.50

**Intermediate 2**:

- Nodes: 169 (13x13)
- Childs: 1x1
- Overlap: 3.25x3.25

**Intermediate 3**:

- Nodes: 9 (3x3)
- Childs: 5x5
- Overlap: 2.00x2.00

**Output**:

- Nodes: 1 (1x1)
- Childs: 3x3
- Overlap: 1.00x1.00

However, intermediate levels 1 and 2 operates in a slightly different way with respect to the classic HTM theory:

At intermediate level 1, 100 prewired coincidences, which operate as 8x8 filters, are used. Each filter is shaped as a dipole (two Gaussians, one positive and one negative, which orientations and centers are random generated) and is able to robustly assess the intensity relationship between two adjacent regions (which one is clearer / darker than the other). As shown in [56] this technique has been already proven to be able

to extract robust and discriminating features. Being the coincidence prewired and the temporal clustering avoided, this level is not subject to training. On the other hand, at Intermediate level 2, coincidences are obtained by merging features from level 1, and calculating their activation through the Hamming distance (as suggested in [56]). Level 2 includes temporal groups and the training is performed in the traditional way.

## 5.4 Validation of the CNN implementation

Before tackling directly the comparison between CNN and HTM on different tasks, it's important to validate the new CNN implementation by comparing the accuracy results obtained on the full NORB dataset with those reported in [16] and [57]. In this way, it is possible to be certain that the CNN will perform at its best when it comes to a comparison. After several exploratory tests, it has been found that accuracy is generally maximized and the training times contained, with the following parameters:

- Initial learning rate: 0.05
- Learning rate decay: 0.9998
- Max num epochs: 10000 (Arbitrarily high)
- Batch size: 200
- Minimum learning rate: 0.01
- Patience: 24300,
- Patience increase: 2
- Improvement threshold: 0.9995
- Downsampling type: Average
- Activation function: Tanh
- Images dim: 32x32

It has been found that changing the activation function or the downsampling type can significantly affect the level of accuracy. In order to carry out a closer evaluation, it has been decided to compare the CNN monocular architecture with the binocular one and a nearest neighbor classifier as baseline. In the following sections, if not specified, the monocular architecture is implied.

| **Training size (per class)** | **NN** | **LeNet7** | **LeNet7 (binocular)** |
|---|---|---|---|
| 20 | 59.44% | 59.72% | 64.72% |
| 50 | 67.62% | 68.76% | 78.32% |
| 100 | 71.26% | 74.34% | 86.58% |
| 200 | 75.96% | 81.62% | 89.9% |
| 500 | 79.16% | 83.54% | 92.76% |
| 1000 | 81.04% | 84.17% | 93.42% |
| 2000 | 82.30% | 86.29% | 93.94% |
| 4860 | 83.34% | 86.00% | 94.40% |

TABLE 5.1: Accuracy results comparison among two different CNN architecture and a NN baseline on different training size

Each of them has been trained on different training sizes to observe how the number of training images affects the accuracy. Accuracy results are reported in tab. 5.1 and plotted in fig. 5.3.

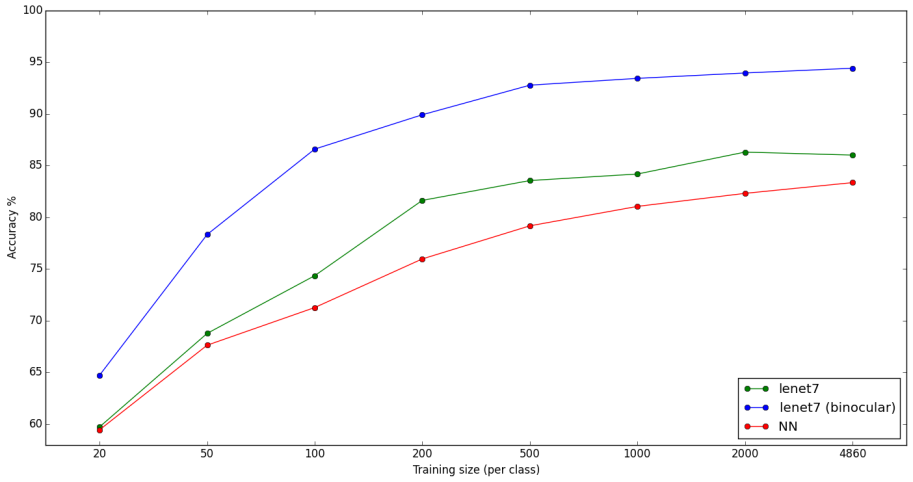


FIGURE 5.3: Plotted accuracy comparison among two different CNN architecture and a NN baseline on different training size. X coordinates are equispaced for an easier understanding.

The best possible result has been obtained with the binocular architecture, with an error rate of 5.6% and the full training set of 24300 images (4860 images for each of the five classes). This is in line with the result obtained in [57] and currently the best result ever achieved on the NORB dataset. This allow us to have more confidence in the goodness of our implementation.

| **Training size (per class)** | **LeNet7** | **LeNet7 (binocular)** |
|---|---|---|
| 20 | 52.96 m | 59.70 m |
| 50 | 55.20 m | 57.01 m |
| 100 | 35.19 m | 35.88 m |
| 200 | 54.53 m | 44.57 m |
| 500 | 57.29 m | 58.74 m |
| 1000 | 51.84 m | 52.58 m |
| 2000 | 94.55 m | 98.11 m |
| 4860 | 74.70 m | 77.32 m |

TABLE 5.2: Training time comparison among two different CNN architecture on different training size

In tab. 5.2, training times are reported (with exception of the NN training times that were negligible). The two CNN architectures have been trained on a PC with two CPUs *Intel Xeon E5-2650 @2.0GHz (8 cores)* and four GPUs *NVIDIA Tesla C2075 @1.15GHz (448 cores)* but using only one GPU.

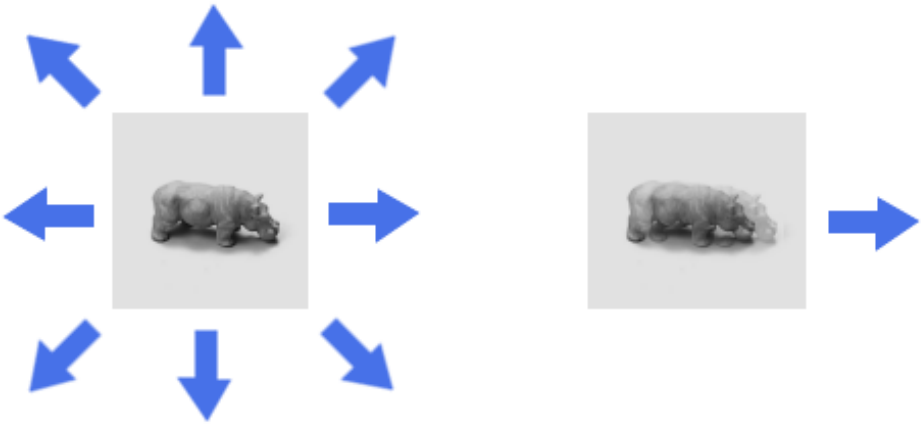

FIGURE 5.4: On the left jitter directions are esemplified, on the right an example of a jittered image is reported (the two images are overlapped).

Although the CNN accuracy is better on binocular images, from here on, we will consider only the monocular case, since it is considered more general and usable in real scenarios. In fact, considering the very nature of the CNN algorithm, it is easy to isolate the object from a uniform background, starting with two slightly different images.
Before dealing with the real comparison between CNN and HTM on the NORB dataset, another important question has to be answered.

| **Training size (per class)** | **LeNet7** | **LeNet7 (jittered)** |
|---|---|---|
| 20 | 59.72% | 60.08% |
| 50 | 68.76% | 70.32% |
| 100 | 74.34% | 76.60% |
| 200 | 81.62% | 83.06% |
| 500 | 83.54% | 83.82% |
| 1000 | 84.17% | 85.86% |
| 2000 | 86.29% | 86.36% |
| 4860 | 86.00% | 85.81% |

TABLE 5.3: Accuracy results of a LeNet7 on different training size, with jittered images or not.

Since the HTM algorithm works on jittered images to learn the coincidences, for a fair comparison, we would like to know if they can also rise the CNN accuracy. Hence, for training size of 20, 50, 100, all the images has been jittered of one pixel in eight direction as illustrated in fig. 5.4. So, for example, when in the training set there are 100x5 = 5000 images, another 500x8 = 4000 are added for a total of 4500 images. For the other sizes 200, 500, 1000, 2000, 4860, only the first 4000 jittered images have been added, since HTM exploits jittered images in this way.

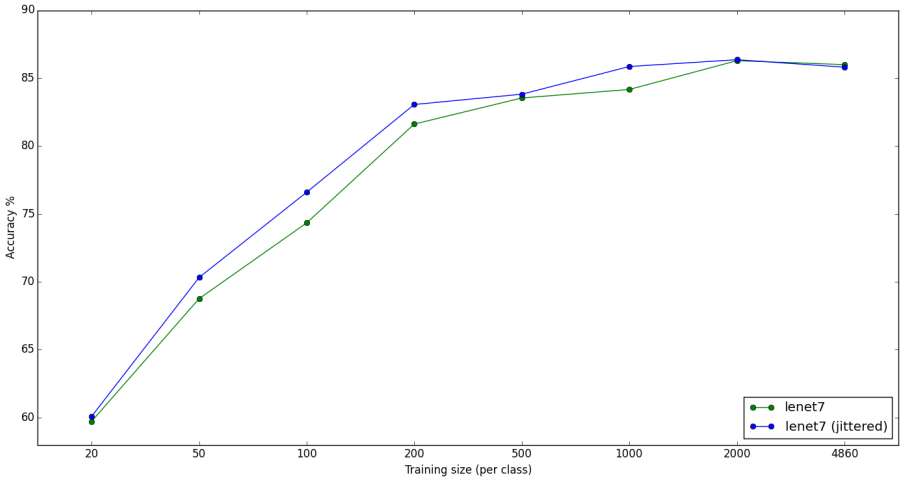


FIGURE 5.5: Plotted accuracy results of a LeNet7 on different training size, with jittered images or not. X coordinates are equispaced for an easier understanding.

In tab. 5.3 accuracy results are reported for each training size. As expected, not impressive improvements have been made. This is in line with what reported in the

| Training size | Epoch | GPU Time | CPU Time | Speedup |
|---|---|---|---|---|
| 1000 | 100 | 11.1 m | 36.36 m | 3.28 |
| 1000 | 200 | 19.43 m | 63.51 m | 3.27 |
| 1000 | 300 | 27.23 m | 88.80 m | 3.26 |
| 1000 | 500 | 42.62 m | 134.41 m | 3.15 |
| 2000 | 100 | 16.47 m | 51.23 m | 3.11 |
| 2000 | 200 | 29.78 m | 93.04 m | 3.14 |
| 2000 | 300 | 42.63 m | 135.38 m | 3.18 |
| 2000 | 500 | 68.21 m | 219.48 m | 3.22 |
| 4860 | 100 | 31.87 m | 102.45 m | 3.22 |
| 4860 | 200 | 61.08 m | 197.21 m | 3.23 |
| 4860 | 300 | 90.37 m | 292.36 m | 3.24 |
| 4860 | 500 | 147.67 m | 477.13 m | 3.23 |

TABLE 5.4: Training time comparison between GPU and CPU implementation with Theano

literature about the CNN invariance with respect to small translations. In fig. 5.5 accuracy results are plotted for a graphical comparation.

As a final consideration regarding the implementation, let us consider the GPU speedup over a single CPU (using a single core). In tab. 5.4 different training times are reported. Since, even with the same seed for the random number generator, a slightly different computation can arise from the GPU and the CPU implementation, only the number of epochs is considered and not the reached level of accuracy. Looking at the results, it is possible to conclude that a x3.2 speedup has been achieved. This is not impressive considering other speedup results reported in the literature [58]. However, with a proper code optimization (along with the updated CUDA libraries) we are confident that better performances can be achieved on this task as well.

## 5.5 On NORB dataset

In this section, a proper comparison between CNN and HTM on the NORB dataset is proposed. The main goal is to prove that HTM can outperform CNN in terms of accuracy when the data are fewer.

### 5.5.1 Setup

Since the original NORB dataset was divided in training and test set, there was no validation set to use. As it has been shown in the section regarding the CNN implementation, a validation set is needed for the stop criterion. The cleanest solution has been

to perform a *k-fold cross-validation*[2] and this is what has been done choosing k as 5. Practically speaking, the test set has been split in five parts. For each of the different training size one fifth of the test set has been used as the validation set and the other four fifths as the test set. Then, we have kept changing the part used for the validation set until all the five parts have been used. Finally, for each of the training size, the averaged accuracy is taken in account. These operations are performed for both the HTM and CNN implementations. Also in this case the CNN is trained and tested on a GPU Tesla C2075 Fermi while the HTM, with an OPEN-MP improved version, on a CPU *Xeon W3550, 4 cores*.

### 5.5.2 Results

In tab. 5.5 all the accuracy results along with the training times are reported for the CNN.

| **Training size** | **Valid. accuracy** | **Test accuracy** | **Time** |
|---|---|---|---|
| 100 + 800jit | 60.64% | 60.58% | 10.94 m |
| 250 + 2000jit | 69.75% | 69.64% | 31.15 m |
| 500 + 4000jit | 77.56% | 77.27% | 38.24 m |
| 1000 + 4000jit | 82.96% | 82.8% | 91.26 m |
| 2500 + 4000jit | 84.17% | 83.87% | 94.90 m |
| 5000 + 4000jit | 85.75% | 85,47% | 124.7 m |
| 10000 + 4000jit | 86.54% | 86.2% | 187.7 m |
| 24300 + 4000jit | 85.35% | 85.01% | 51.31 m |

TABLE 5.5: Averaged accuracy results of a LeNet7 on different training size after a 5-fold cross-validation.

The first consideration, is, of course, that these results are in line with what obtained during the evaluation of the CNN implementation already discussed in a previous section. On the other hand, In tab. 5.6 all the accuracy results on exactly the same tasks but regarding the HTM are reported.

| **Training size** | **Valid. accuracy** | **Test accuracy** | **Time** |
|---|---|---|---|
| 100 + 800jit | 64.29% | 64.21% | 21.19 m |
| 250 + 2000jit | 73.36% | 73.22% | 23.13 m |
| 500 + 4000jit | 78.90% | 78.82% | 22.14 m |
| 1000 + 4000jit | 82.03% | 81.86% | 26.04 m |

[2] *k-fold cross-validation* is a model validation technique for assessing how the results of a statistical analysis will generalize to an independent data set.

| | | | |
|---|---|---|---|
| 2500 + 4000jit | 84.47% | 84.16% | 61.08 m |
| 5000 + 4000jit | 85.75% | 85.37% | 89.58 m |
| 10000 + 4000jit | 85.61% | 85.83% | 143.5 m |
| 24300 + 4000jit | 86.42% | 86.24% | 596.2 m |

TABLE 5.6: HTM averaged accuracy results on different training size after a 5-fold cross-validation. Training times include unsupervised and HSR phases.

Looking at the numbers, we soon realize that using less than 200 examples per class, HTM outperforms CNN of some percentage points. Then, exceeded that threshold, the two algorithms remain comparable. In fig. 5.6 the averaged accuracy results are plotted for each training size. In terms of training times, even if we are dealing with two very different implementation languages, and different hardware platforms, we can also say that they are rather comparable. HTM has been trained on a fixed number of epochs, this is why all the training times within the same training size are equal. For the sizes of 100, 250 and 500 patterns, a number of 30 epochs has been chosen. With the exception of the last size where an exceptional number of 100 epochs has been deployed, for all the other training sizes, a number of 50 epochs have been used.

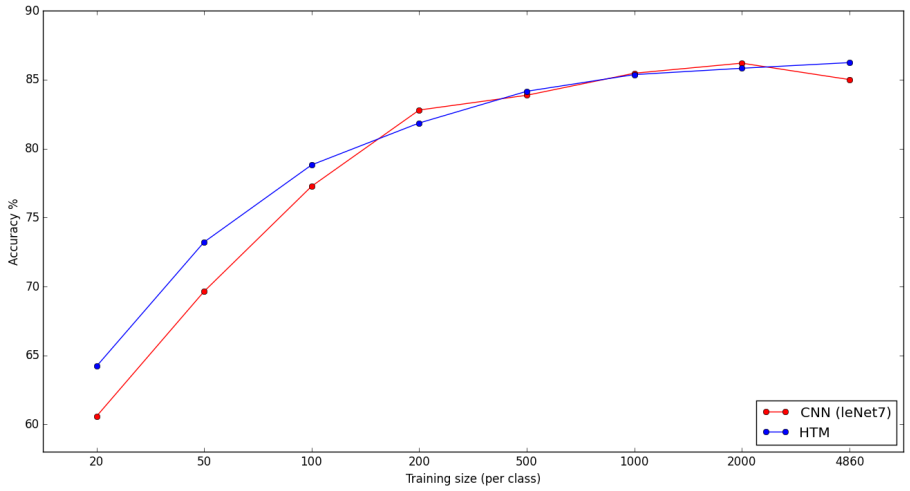


FIGURE 5.6: Averaged 5-fold accuracy comparation between CNN and HTM on different training size. X coordinates are equispaced for an easier understanding.

It is worth pointing out that a great deal of training time is spent with the convolution at level 1. In fact, this procedure inside the HTM implementation is not optimized at the moment. It could be highly improved using integer arithmetic, sparse representation SIMD instructions and GPUs. Further investigations on the subject can not be addressed in this dissertation due to time constraints and will be carried on in the near future.

## 5.6 On NORB-Sequences

In this section several exploratory tests are performed on the NORB-sequences benchmark. These experiments can be considered as preliminary since the sequential factor is not taken into account during the training and the test is about the classification of single images rather than sequences.

### 5.6.1 Setup

The experiments carried out in this section differ from the previous ones essentially for two reasons. First of all, the aim, in this case, is to evaluate the goodness of the new benchmark rather than the quality of the learning algorithms. In this case the training is performed on each instance of each class not respecting the original train/test NORB subdivision. Secondly, these tests are conducted both considering the original class subdivision and considering each instance as a separated class. Hence, in this case we are dealing with a much harder problem with 50 classes. The experiments are run in the same environments and hardwares of the previous ones.

### 5.6.2 Results

In tab. 5.7 first accuracy results on different training size are reported. The first prototype of the NORB-sequences benchmark has been used. Using train_conf, for each instance a different number of sequence of length 20 has been taken in account (2, 3 and 5). As a validation set, *test_conf_1* has been chosen, using only the first 5 images of each sequence. Finally, regarding the test, every test_conf (minimum distance of 1, 2, 3 and 4) has been used. In this case, eventually duplicated images (images that appear more than one time in the sequences) are included only once. This has been found not compromising the achievable accuracy results as reported in tab. 5.8.

| CNN | | | | | | | |
|---|---|---|---|---|---|---|---|
| Name | | | Size | | | Accuracy | |
| **Train** | **Valid.** | **Test** | **Train** | **Test** | **Valid.** | **Valid.** | **Test** |
| 2x20 | test_conf_1 | test_conf_1 | 1847 | 2316 | 8039 | 83% | 84.1% |
| 2x20 | test_conf_1 | test_conf_2 | 1847 | 2316 | 6401 | 83% | 82.96% |
| 2x20 | test_conf_1 | test_conf_3 | 1847 | 2316 | 4448 | 83% | 82.26% |
| 2x20 | test_conf_1 | test_conf_4 | 1847 | 2316 | 2375 | 83% | 81.35% |
| 3x20 | test_conf_1 | test_conf_1 | 2732 | 2316 | 8039 | 89.05% | 89.27% |
| 3x20 | test_conf_1 | test_conf_2 | 2732 | 2316 | 6401 | 89.05% | 88.46% |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3x20 | test_conf_1 | test_conf_3 | 2732 | 2316 | 4448 | 89.05% | 88.42% |
| 3x20 | test_conf_1 | test_conf_4 | 2732 | 2316 | 2375 | 89.05% | 88.00% |
| 5x20 | test_conf_1 | test_conf_1 | 4496 | 2316 | 8039 | 95.1% | 95.11% |
| 5x20 | test_conf_1 | test_conf_2 | 4496 | 2316 | 6401 | 95.1% | 94.4% |
| 5x20 | test_conf_1 | test_conf_3 | 4496 | 2316 | 4448 | 95.1% | 93.1% |
| 5x20 | test_conf_1 | test_conf_4 | 4496 | 2316 | 2375 | 95.1% | 92.13% |

TABLE 5.7: Accuracy results of the CNN trained on different training size and tested on different test sets collected in the NORB-sequences benchmark. Note that the accuracy is high because the CNN is trained on all the instances of the five classes.

| CNN | | | | | | | |
|---|---|---|---|---|---|---|---|
| Name | | | Size | | | Accuracy | |
| **Train** | **Valid.** | **Test** | **Train** | **Test** | **Valid.** | **Valid.** | **Test** |
| 5x20 | test_conf_1 | test_conf_1 | 5000 | 2500 | 10000 | 94.84% | 94.89% |
| 5x20 | test_conf_1 | test_conf_2 | 5000 | 2500 | 10000 | 94.84% | 94.55% |
| 5x20 | test_conf_1 | test_conf_3 | 5000 | 2500 | 10000 | 94.84% | 93.21% |
| 5x20 | test_conf_1 | test_conf_4 | 5000 | 2500 | 10000 | 94.84% | 92.00% |

TABLE 5.8: Accuracy results of the CNN trained on 5 sequences of 20 images for each class and tested on different test sets collected in the NORB-sequences benchmark. In this case eventually duplicated images are included in the training, validation and test sets.

The achieved accuracy is not surprisingly higher than what reported on the test on the NORB dataset. This is of course because the training is performed on all the instances as has been mentioned before. In tab. 5.9 accuracy results of the 50-classes classification task is reported.

| CNN | | | | | | | |
|---|---|---|---|---|---|---|---|
| Name | | | Size | | | Accuracy | |
| **Train** | **Valid.** | **Test** | **Train** | **Test** | **Valid.** | **Valid.** | **Test** |
| 2x20 | test_conf_1 | test_conf_1 | 1847 | 2316 | 8039 | 38.18% | 37.57% |
| 2x20 | test_conf_1 | test_conf_2 | 1847 | 2316 | 6401 | 38.18% | 34.38% |
| 2x20 | test_conf_1 | test_conf_3 | 1847 | 2316 | 4448 | 38.18% | 30.71% |
| 2x20 | test_conf_1 | test_conf_4 | 1847 | 2316 | 2375 | 38.18% | 25.47% |
| 3x20 | test_conf_1 | test_conf_1 | 2732 | 2316 | 8039 | 49.87% | 49.89% |
| 3x20 | test_conf_1 | test_conf_2 | 2732 | 2316 | 6401 | 49.87% | 48.01% |
| 3x20 | test_conf_1 | test_conf_3 | 2732 | 2316 | 4448 | 49.87% | 40.56% |
| 3x20 | test_conf_1 | test_conf_4 | 2732 | 2316 | 2375 | 49.87% | 33.77% |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5x20 | test_conf_1 | test_conf_1 | 4496 | 2316 | 8039 | 54.86% | 55.17% |
| 5x20 | test_conf_1 | test_conf_2 | 4496 | 2316 | 6401 | 54.86% | 52.55% |
| 5x20 | test_conf_1 | test_conf_3 | 4496 | 2316 | 4448 | 54.86% | 45.86% |
| 5x20 | test_conf_1 | test_conf_4 | 4496 | 2316 | 2375 | 54.86% | 40.08% |

TABLE 5.9: Accuracy results of the CNN trained on different training size and tested on different test sets collected in the NORB-sequences benchmark. Note that the accuracy is low because 50 different classes are considered (one for each instance).

In line with our expectations, the achieved accuracy is much lower, and the gap introduced by the minimum distance parameter seems to have a legit leverage. Concerning the HTM algorithm, preliminary tests are reported below for both the 5-classes and 50-classes experiments. In tab. 5.10, accuracy results (only on the test sets) are shown. With the 2x20 and 3x20 sets, the network have been trained only in an unsupervised fashion, while with the 5x20 an additional 50 epochs of HSR refinement have been used.

| HTM | | | | | | |
|---|---|---|---|---|---|---|
| Name | | | Size | | | Accuracy |
| **Train** | **Valid.** | **Test** | **Train** | **Test** | **Valid.** | **Test** |
| 2x20 | test_conf_1 | test_conf_1 | 1847 | 2316 | 8039 | 86.36% |
| 2x20 | test_conf_1 | test_conf_2 | 1847 | 2316 | 6401 | 86.16% |
| 2x20 | test_conf_1 | test_conf_3 | 1847 | 2316 | 4448 | 86.16% |
| 2x20 | test_conf_1 | test_conf_4 | 1847 | 2316 | 2375 | 84.20% |
| 3x20 | test_conf_1 | test_conf_1 | 2732 | 2316 | 8039 | 91.00% |
| 3x20 | test_conf_1 | test_conf_2 | 2732 | 2316 | 6401 | 89.52% |
| 3x20 | test_conf_1 | test_conf_3 | 2732 | 2316 | 4448 | 88.68% |
| 3x20 | test_conf_1 | test_conf_4 | 2732 | 2316 | 2375 | 85.88% |
| 5x20 | test_conf_1 | test_conf_1 | 4496 | 2316 | 8039 | 92.69% |
| 5x20 | test_conf_1 | test_conf_2 | 4496 | 2316 | 6401 | 91.62% |
| 5x20 | test_conf_1 | test_conf_3 | 4496 | 2316 | 4448 | 91.86% |
| 5x20 | test_conf_1 | test_conf_4 | 4496 | 2316 | 2375 | 91.23% |

TABLE 5.10: Accuracy results of the HTM trained on different training size and tested on different test sets collected in the NORB-sequences benchmark. In red, the accuracy results that are better than what reported for the CNN are highlighted.

Also in this case, HTM outperforms CNN when there are few data. Moreover, it is worth pointing out that giving the preliminary nature of these results, there might be a good room for improvement. In tab. 5.11, the 50-classes experiment results are reported.

| HTM | | | | | | |
|---|---|---|---|---|---|---|
| Name | | | Size | | | Accuracy |
| **Train** | **Valid.** | **Test** | **Train** | **Test** | **Valid.** | **Test** |
| 2x20 | test_conf_1 | test_conf_1 | 1847 | 2316 | 8039 | 37.08% |
| 2x20 | test_conf_1 | test_conf_2 | 1847 | 2316 | 6401 | 37.82% |
| 2x20 | test_conf_1 | test_conf_3 | 1847 | 2316 | 4448 | 33.17% |
| 2x20 | test_conf_1 | test_conf_4 | 1847 | 2316 | 2375 | 28.89% |
| 3x20 | test_conf_1 | test_conf_1 | 2732 | 2316 | 8039 | 43.68% |
| 3x20 | test_conf_1 | test_conf_2 | 2732 | 2316 | 6401 | 44.08% |
| 3x20 | test_conf_1 | test_conf_3 | 2732 | 2316 | 4448 | 37.93% |
| 3x20 | test_conf_1 | test_conf_4 | 2732 | 2316 | 2375 | 34.93% |
| 5x20 | test_conf_1 | test_conf_1 | 4496 | 2316 | 8039 | 52.57% |
| 5x20 | test_conf_1 | test_conf_2 | 4496 | 2316 | 6401 | 49.74% |
| 5x20 | test_conf_1 | test_conf_3 | 4496 | 2316 | 4448 | 45.52% |
| 5x20 | test_conf_1 | test_conf_4 | 4496 | 2316 | 2375 | 41.30% |

TABLE 5.11: Accuracy results of the HTM trained on different training size and tested on different test sets collected in the NORB-sequences benchmark. 50 different classes are considered. In red, the accuracy results that are better than what reported for the CNN are highlighted.

HTM seems to perform better than the CNN also when the minimum distance parameter is higher. This is an interesting observation that will be certainly deepen in the future.

# 6

# Conclusions and future work

*"If I have seen further, it is by standing on the shoulders of giants."*

– Isaac Newton

In this chapter, some conclusions about the entire work which has been carried out during the dissertation will be drawn. Then, a number of future improvement paths will be proposed.

## 6.1 Conclusions

During the dissertation a number of ambitious tasks have been undertaken. Firstly, a new benchmark based on the well-known and receipt NORB benchmark has been created along with a robust and flexible java application. This has been done on the shared feeling that working with image sequences is a critical step towards the ambitious goal of teaching machines how to see. Secondly, several experiments comparing two deep learning algorithms (HTM and CNN) for object recognition have been conducted. The implementation of a flexible CNN architecture with Theano has been a crucial point of the entire dissertation. This has not been an easy task due to server configurations, few architectural details, and a huge number of parameters to tune. Indeed, a significant amount of time has been spent trying to reach the proper level of accuracy on the NORB dataset, in line with what reported in the literature. The main objective of the dissertation was to prove the quality of the HTM algorithm, a new biologically inspired model with an elegant mathematical model incorporating basic concepts of the brain functionalities. The goal has been reached having demonstrated that, with a lower quantity of data, HTM can outperform CNN while remaining comparable with more data

in terms of both accuracy and training times. Of course, these conclusions have to be corroborated by further and more accurate experimentations, but now we have another evidence that, even scaling up the number of features (now 32x32 images), the considerations formulated in [11] are holding. These observations, lead us to conclude that incorporating further insights from existing biological learning systems can be helpful for deep learning, a field that has been always thought as closely related to the biological aspects of learning, but that, instead, could be significantly improved following this path. The main expectation would be to incorporate new concepts like context and attention during the learning process, moving the focus towards unsupervised learning and taking advantage of more flexible and dynamic architectures. This would allow us to reshape deep learning making it more computationally feasible and able to generalize well on almost any task.

## 6.2 Future work

In this section a number of possible improvements to the work carried out during the dissertation will be outlined. They can be conveniently framed in three main parts: the first one about the NORB-sequences benchmark, the second one about the CNN and HTM implementations and finally the third one about a new set of interesting experiments to be conducted.

**The NORB-sequences benchmark** The experiments reported in the previous chapters have already proved the benchmark quality and usability. However, few tweaks can be added. In particular regarding the smoothness of the sequence, that for now is acceptable but improvable. Simple image processing operations, commonly used to increase the number of pattern in machine learning tasks can be performed. For example, it is possible to add slightly rotations and shifts for each image to exponentially increase the total number of images. Generally, these operations, are different to compute without corrupting images quality, but since in this case the images are grayscale, centered and with a uniform background this is not the case. The Author already plans to add these improvements in upcoming software releases.

**The CNN and HTM implementations** Concerning the two algorithms implementations, it would be interesting to let them converge towards a common framework in order to evaluate their performance in a more accurate way. Since Theano has been chosen for the CNN implementation after a meticulous search about the most powerful and research oriented framework, it would be natural to convert the HTM to this framework

too. In this way, we could compare the algorithms performance both on CPU and GPU with minimal effort. Another important point is the simplification of the experiments environment, dealing with less languages and tools that can always alter results of one algorithm rather than the other.

**A new set of experiments** Of course, the first thing to do would be testing the two algorithms accuracy on the NORB sequences merging their confidence levels as has been done with the KNN apporach in chapter 4. Then, in order to compare and evaluate the online learning proficiency of the two algorithms, a completely new set of experiments could be designed. This is a pretty interesting quality for a learning algorithm to have, especially when it comes to real applications. These experiments could be based on two main training steps:

1. Train the model in a batch way (supervised, eventually divided in mini-batch).
    - For example using two sequences for each instance of the training set (2x20x10x 5 = 2000 images) of the NORB-Sequences benchmark.
2. Train the model incrementally.
    - For example using only one sequence (20x10x5 = 1000 images).

Step 2 could be repeated 8 times (sequences from 3 to 10 not used in step 1), always starting from the previously obtained model. The hypothesis, however, is that you no longer have access to the previous patterns (even those of the initial batch). Thus, the incremental training can have a negative impact if not properly addressed. The entire experiment could be repeated on the 50-classes problem too, testing the accuracy progress after each incremental training.

# Bibliography


[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[2] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. Deepspeech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

[3] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*, pages 3111–3119, 2013.

[4] Yoshua Bengio, Aaron C Courville, and Pascal Vincent. Unsupervised feature learning and deep learning: A review and new perspectives. *CoRR, abs/1206.5538*, 1, 2012.

[5] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with cots hpc systems. In *Proceedings of The 30th International Conference on Machine Learning*, pages 1337–1345, 2013.

[6] Quoc V Le. Building high-level features using large scale unsupervised learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8595–8598. IEEE, 2013.

[7] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[8] Jeff Hawkins and Dileep George. Hierarchical temporal memory: Concepts, theory and terminology. Technical report, Numenta, 2006.

[9] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Computer Vision*

*and Pattern Recognition (CVPR), 2014 IEEE Conference on*, pages 1701–1708. IEEE, 2014.

[10] Chuck Rosenberg Google Research Blog. Improving photo search: A step across the semantic gap, 2015. URL http://googleresearch.blogspot.it/2013/06/improving-photo-search-step-across.html.

[11] Davide Maltoni. Pattern recognition by hierarchical temporal memory. Technical report, DEIS - University of Bologna, April 2011. URL http://cogprints.org/9187/1/HTM_TR_v1.0.pdf.

[12] Wikimedia Commons. Anatomy of a multipolar neuron, 2013. URL https://upload.wikimedia.org/wikipedia/commons/1/10/Blausen_0657_MultipolarNeuron.png.

[13] Luca Nicolini. Convolutional neural network, 2012.

[14] Thiago M Geronimo, Carlos ED Cruz, Eduardo C Bianchi, Fernando de Souza Campos, and Paulo R Aguiar. *MLP and ANFIS Applied to the Prediction of Hole Diameters in the Drilling Process*. INTECH Open Access Publisher, 2013.

[15] Davide Maltoni and Erik M Rehn. Incremental learning by message passing in hierarchical temporal memory. In *Artificial Neural Networks in Pattern Recognition*, pages 24–35. Springer, 2012.

[16] Yann LeCun, Fu Jie Huang, and Leon Bottou. Learning methods for generic object recognition with invariance to pose and lighting. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, volume 2, pages II–97. IEEE, 2004.

[17] Stuart Russell, Peter Norvig, and Artificial Intelligence. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25, 1995.

[18] Ryszard S Michalski, Jaime G Carbonell, and Tom M Mitchell. *Machine learning: An artificial intelligence approach*. Springer Science & Business Media, 2013.

[19] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[20] Phil Simon. *Too Big to Ignore: The Business Case for Big Data*. John Wiley & Sons, 2013.

[21] Tom M Mitchell. Machine learning. 1997. *Burr Ridge, IL: McGraw Hill*, 45, 1997.

[22] Stevan Harnad. The annotation game: On turing (1950) on computing, machinery, and intelligence. *The Turing Test Sourcebook: Philosophical and Methodological Issues in the Quest for the Thinking Computer*, 2006.

[23] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[24] Tim Morris. *Computer vision and image processing*. Palgrave Macmillan, 2004.

[25] Milan Sonka, Vaclav Hlavac, and Roger Boyle. *Image processing, analysis, and machine vision*. Cengage Learning, 2014.

[26] David A Forsyth and Jean Ponce. A modern approach. *Computer Vision: A Modern Approach*, 2003.

[27] Ajith Abraham. Artificial neural networks. *handbook of measuring system design*, 2005.

[28] Eric R Kandel, James H Schwartz, Thomas M Jessell, et al. *Principles of neural science*, volume 4. McGraw-Hill New York, 2000.

[29] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[30] Sankar K Pal and Sushmita Mitra. Multilayer perceptron, fuzzy sets, and classification. *IEEE Transactions on Neural Networks*, 3(5):683–697, 1992.

[31] Yves Chauvin and David E Rumelhart. *Backpropagation: theory, architectures, and applications*. Psychology Press, 1995.

[32] Yoshua Bengio, Ian J. Goodfellow, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2015. URL http://www.iro.umontreal.ca/~bengioy/dlbook.

[33] Yoshua Bengio. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[34] Lee Gomes IEEE Spectrum. Machine-learning maestro michael jordan on the delusions of big data and other huge engineering efforts, 2014. URL http://spectrum.ieee.org/robotics/artificial-intelligence/machinelearning-maestro-michael-jordan-on-the-delusions-of-big-data/and-other-huge-engineering-efforts.

[35] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[36] David H Hubel and Torsten N Wiesel. Receptive fields of single neurones in the cat's striate cortex. *The Journal of physiology*, 148(3):574–591, 1959.

[37] Jeff Hawkins and Sandra Blakeslee. *On intelligence*. Macmillan, 2007.

[38] Dileep George. *How the brain might work: A hierarchical and temporal model for learning and recognition*. PhD thesis, Stanford University, 2008.

[39] Patrice Simard, Yann LeCun, and John S Denker. Efficient pattern recognition using a new transformation distance. In *Advances in neural information processing systems*, pages 50–58, 1993.

[40] David G Lowe. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, pages 1150–1157. Ieee, 1999.

[41] Laurenz Wiskott and Terrence J Sejnowski. Slow feature analysis: Unsupervised learning of invariances. *Neural computation*, 14(4):715–770, 2002.

[42] Jake Bouvrie, Lorenzo Rosasco, and Tomaso Poggio. On invariance in hierarchical models. In *Advances in Neural Information Processing Systems*, pages 162–170, 2009.

[43] Christof Koch. *The Quest for consciousness: a neurobiological approach*. Roberts and Company Publishers, Englewood, CO, 2004. ISBN 0-9747077-0-8. URL http://opac.inria.fr/record=b1101880.

[44] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 2014.

[45] Erik M Rehn and Davide Maltoni. Incremental learning by message passing in hierarchical temporal memory. *Neural computation*, 26(8):1763–1809, 2014.

[46] Dileep George and Jeff Hawkins. Towards a mathematical theory of cortical micro-circuits. *PLoS Comput Biol*, 5(10):e1000532, 2009.

[47] Á Dám B Csapó, Péter Baranyi, and Domonkos Tikk. Object categorization using vfa-generated nodemaps and hierarchical temporal memories. In *Computational Cybernetics, 2007. ICCC 2007. IEEE International Conference on*, pages 257–262. IEEE, 2007.

[48] Régis Vaillant, Christophe Monrocq, and Yann Le Cun. Original approach for the localisation of objects in images. *IEE Proceedings-Vision, Image and Signal Processing*, 141(4):245–250, 1994.

[49] Henry Rowley, Shumeet Baluja, Takeo Kanade, et al. Neural network-based face detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20 (1):23–38, 1998.

[50] Hiroshi Murase and Shree K Nayar. Visual learning and recognition of 3-d objects from appearance. *International journal of computer vision*, 14(1):5–24, 1995.

[51] Andrea Selinger and Randal C Nelson. Appearance-based object recognition using multiple views. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–905. IEEE, 2001.

[52] Bastian Leibe and Bernt Schiele. Analyzing appearance and contour based methods for object categorization. In *Computer Vision and Pattern Recognition, 2003. Proceedings. 2003 IEEE Computer Society Conference on*, volume 2, pages II–409. IEEE, 2003.

[53] Naomi S Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992.

[54] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4, page 3. Austin, TX, 2010.

[55] Ian J Goodfellow, David Warde-Farley, Pascal Lamblin, Vincent Dumoulin, Mehdi Mirza, Razvan Pascanu, James Bergstra, Frédéric Bastien, and Yoshua Bengio. Pylearn2: a machine learning research library. *arXiv preprint arXiv:1308.4214*, 2013.

[56] Zhenan Sun and Tieniu Tan. Ordinal measures for iris recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 31(12):2211–2226, 2009.

[57] Kevin Jarrett, Koray Kavukcuoglu, Marc'Aurelio Ranzato, and Yann LeCun. What is the best multi-stage architecture for object recognition? In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 2146–2153. IEEE, 2009.

[58] Rajat Raina, Anand Madhavan, and Andrew Y Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning*, pages 873–880. ACM, 2009.