ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

School of Science
Master Degree in Computer Science

Deep Incremental Learning for Object Recognition

Supervisor:
prof. Davide Maltoni

Candidate:
dott. Riccardo Monica

Co-supervisor:
dott. Vincenzo Lomonaco

Session II
Academic year 2015-16

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

Sommario

Scuola di Scienze
Laurea Magistrale in Informatica

Deep Incremental Learning for Object Recognition

dott. Riccardo Monica

Recentemente le tecniche di *Deep Learning* hanno ricevuto molta attenzione nel settore informatico. Queste tecniche si sono dimostrate particolarmente utili ed efficaci in contesti quali l'elaborazione del linguaggio naturale, il riconoscimento del parlato e la visione artificiale. In numerose applicazioni del mondo reale è stato raggiunto e/o superato lo stato dell'arte [1] [5] [36]. Nell'ambito dell'*apprendimento automatico* (machine learning) il deep learning è stata una vera rivoluzione e diversi strumenti efficaci sono stati introdotti per il learning supervisionato, non supervisionato e per l'apprendimento automatico di feature [44].

Questa tesi si focalizza su tecniche di deep learning nell'ambito del riconoscimento degli oggetti, e in particolare su tecniche di apprendimento incrementale. Con apprendimento incrementale si intendono tecniche in grado di costruire un modello iniziale a partire da un insieme ridotto di dati; il modello è poi successivamente migliorato quando nuovi dati sono disponibili. Per l'apprendimento incrementale è stato dimostrato che l'impiego di sequenze di immagini temporalmente coerenti può essere molto vantaggioso consentendo anche di operare in modo non supervisionato. Una criticità all'apprendimento incrementale è il cosiddetto *forgetting*, ovvero il rischio di dimenticare concetti appresi precedentemente durante il training successivo di un modello [43].

Nei capitoli introduttivi di questo lavoro (capitolo 1 e 2) introdurremo tematiche di base come le reti neurali (Neural Networks), le Convolutional Neural Networks (CNNs) e tecniche di incremental learning. L'approccio mediante CNN è uno dei più efficaci metodi supervisionati per l'apprendimento automatico e risulta particolarmente idoneo per il riconoscimento degli oggetti [23]. Questa tecnica è ben accettata dalla comunità scientifica e largamente utilizzata anche dai grandi player dell'ICT come Google e Facebook: applicazioni degne di nota sono il riconoscimento dei volti da parte di Facebook [11], e il riconoscimento di immagini da parte di Google [18].

La comunità scientifica dispone di numerosi (grandi) dataset di immagini (es, ImageNet [1]) per lo sviluppo e la valutazione di approcci per il riconoscimento di oggetti. D'altro canto, come descritto nel capitolo 3, esistono pochi dataset per lo studio di approcci incrementali basati su sequenze temporalmente coerenti. Per questo motivo, in questo lavoro è stato collezionato un nuovo dataset, denominato TCD4R (Temporal Coherent Dataset For Robotics). Nel capitolo 4, TCD4R è descritto in dettaglio. Il dataset sarà reso disponibile alla comunità scientifica per consentire nuovi studi e avanzamenti nel settore.

Nel capitolo 5 sono descritti numerosi esperimenti eseguiti applicando CNN (con diverse architetture e parametrizzazioni) al riconoscimento di oggetti su TCD4R. Infine nel capitolo 6 sono riportate le conclusioni e alcuni commenti su sviluppi futuri.

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

Abstract

School of Science
Master Degree in Computer Science

Deep Incremental Learning for Object Recognition

dott. Riccardo Monica

In recent years, *deep learning* techniques received great attention in the field of information technology. These techniques proved to be particularly useful and effective in domains like natural language processing, speech recognition and computer vision. In several real world applications deep learning approaches improved the state-of-the-art [1] [5] [36]. In the field of machine learning, deep learning was a real revolution and a number of effective techniques have been proposed for both supervised and unsupervised learning and for representation learning [44].

This thesis focuses on deep learning for object recognition, and in particular, it addresses incremental learning techniques. With incremental learning we denote approaches able to create an initial model from a small training set and to improve the model as new data are available. Using temporal coherent sequences proved to be useful for incremental learning since temporal coherence also allows to operate in unsupervised manners. A critical point of incremental learning is called *forgetting* which is the risk to forget previously learned patterns as new data are presented [43].

In the first chapters of this work (chapter 1 and 2) we introduce the basic theory on neural networks, Convolutional Neural Networks (CNN) and incremental learning. CNN is today one of the most effective approaches for supervised object recognition [23]; it is well accepted by the scientific community and largely used by ICT big players like Google and Facebook: relevant applications are Facebook face recognition [11] and Google image search [18].

The scientific community has several (large) datasets (e.g., ImageNet [1]) for the development and evaluation of object recognition approaches. However, as described in chapter 3, very few temporally coherent datasets are available to study incremental approaches.

For this reason we decided to collect a new dataset named TCD4R (Temporal Coherent Dataset For Robotics). In chapter 4 TCD4R is described in detail. The dataset will be made available to the scientific community to allow new studies.

In chapter 5 several experiments are reported where different CNNs are used for object recognition on TCD4R. Finally, in chapter 6 we report some conclusions and discuss future works.

Acknowledgements

First of all, I would like to thank my supervisor, professor Davide Maltoni for accepting me as candidate (even though I was coming from a different degree course) and for helping me through the entire process of this thesis development: starting from providing the necessary information about this subject, still new to me, until the final review of this thesis.
I would like to thank my co-supervisor, PhD student Vincenzo Lomonaco, for bearing my pressing questions especially during the experimental phase or the writing of this thesis. I've really appreciated his availability.

I would like to thank my family in all its parts. Starting from my parents, Spartaco and Annamaria, and my grandparents, Lino and Paola, utill all my friends with a special thanks to Matteo Tiscornia which accompanied me through this two long commute years and project works.
I would like to thank also all my fellow students for helping and entertaining me during all my academic years.

I need also to thank all the people who helped me during this last year because their presence and especially patience are starting to spread light inside me.

Contents

List of Figures

List of Tables

A Spartaco, Annamaria, Lino e Paola
i miei porti sicuri anche nelle tempeste più difficili.

Alla vita.

Introduction

Learning is one of the more human and fascinating action which computer science has ever reached. This natural behaviour is studied in the field of machine learning, a branch of artificial intelligence. Already in their early days as an academic discipline, some researchers were interested in having machines which can learn from data. They soon invented *artificial neural networks* for this purpose [35]. Over time machine learning lost some of its interest to the researchers' view primarily due to the advance of other artificial intelligence techniques such as *expert systems*, in the '80s [22]. With the increasing availability of digitalized information, and the possibility to distribute that via the Internet, machine learning restarted its run. Moreover, it took the chance to reorganize its key goals: from achieving artificial intelligence to tackling solvable problems of practical nature. It shifted the focus away from the symbolic approaches it had inherited from artificial intelligence and moved it towards methods and models borrowed from statistics and probability theory [35]. At the present, we have seen a new incredible improvement of such techniques in many machine learning applications such as image recognition or natural language processing systems. These applications will increase their efficacy in a recent future and others will follow their path.

This have been made possible only by the use of new machine learning techniques which came under the name of *Deep Learning*, the topic of this thesis. Its coming is deeply changing the approach to the problem. The computer vision community, for example, can work right now with methods that permits to deal with raw images data. With the increasing of hardware performance such as computational speed or GPU parallel computing the real usability and effectiveness of these techniques is enabled.

Deep learning techniques are really computationally intensive and they need a huge quantity of data to obtain good results; deep learning, indeed, is increasingly associated to the *Big Data* field for its high data request [24]. In this thesis we focus on deep learning techniques for object recognition, with particular emphasis on *incremental learning*. With incremental learning we denote approaches able to create an initial model from a small training set and to improve the model as new data are available, such as in a human brain. Humans typically learn to recognize object in incremental way (after several interactions with them), not all at once such as in machine learning. In incremental learning approach happens the same thing, the network will

learn an object after it has met it several times; its recognition accuracy will increase after the classification model has been exposed to several objects of the same time [9]. We will study both classical (batch) learning and incremental learning.

In chapter 1 a brief background about artificial neural networks and machine learning is introduced. In chapter 2 we explain the convolutional neural networks concepts and in chapter 3 we introduce the existing benchmarks and their limitations for incremental learning studies. In chapter 4 we describe TCD4R, a new dataset that we collected for incremental learning studies. In chapter 5 several experiments are reported where different CNN are used for object recognition on TCD4R. Finally, chapter 6 concludes the thesis with some comments and discussion on future work.

Chapter 1

Background

In this chapter we introduce a brief background about the thesis topic. In the following sections we describe the concept of Machine Learning and its application in the field of Computer Vision and we introduce the theory behind Artificial Neural Networks, Deep Learning and Incremental Learning.

1.1 Machine Learning

Over the past two decades Machine Learning has become one of the mainstays of information technology, and nowadays it is being largely used in several common applications. Machine Learning is present in any computing environment from the Google web ranking [18] to the new processor used in the Apple iPhone 7 [4].
In 1959, Arthur Samuel defined Machine Learning as a *"Field of study that gives computers the ability to learn without being explicitly programmed"* [32].

Indeed, learning mainly means the act of acquiring new, or modifying and reinforcing, existing knowledge and it may involve synthesizing different types of information. It can also be seen as the progress over time, thus defining an actual learning curve [10].
As we will see later in this thesis, plots relating performances to experiences are widely used in machine learning. Performance is the accuracy of the learning system, while experience might be the number of training examples used for learning or the number of iterations used in optimizing the system model parameters. The machine learning curve is useful for many purposes including comparing different algorithms, choosing model parameters during design, adjusting optimization to improve convergence, and determining the amount of data used for training.
Learning, in the common thought, does not happen all at once, but it builds upon and it is shaped by previous knowledge. As we have already said, learning may be viewed as a *process*, rather than a collection of factual and procedural knowledge.
All these backgrounds introduce the concept of learning in the computer science field. An interesting quote that takes its cue from what we have

previously introduced is that of Tom M. Mitchell, currently the Chair of the machine learning Department at Carnegie Mellon University. He said: *"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"* [29].
This definition is notable because it defines machine learning in fundamentally operational rather than cognitive terms, thus following Alan Turing's proposal in his paper *"Computing Machinery and Intelligence"*. Indeed, the question *"Can machines think?"* should be replaced with the question *"Can machines do what we (as thinking entities) can do?"*, or almost the same as us [19].

1.1.1 Three broad categories and tasks

Machine Learning tasks are typically classified into three broad categories. These depend on the nature of the learning *signal* or *feedback* available to a learning system [34].

- **Supervised learning** is the task of inferring a function from labeled training data. *Training examples* is a set of training data, each is a pair 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 be used for mapping new examples. An optimal scenario allows for the algorithm to correctly determine the class labels for unseen instances. This requires the learning algorithm to use the training data to *reasonably* generalize unseen situations.

- **Unsupervised learning** is the machine learning approach of inferring a function to describe hidden structure from unlabelled data. Since the examples given to the learner are unlabelled, there is no error or reward signal to evaluate a potential solution, which distinguishes unsupervised learning from supervised learning and reinforcement learning.

- **Reinforcement learning** is a learning area inspired by behaviourist psychology, concerned with how agents ought to take actions in an environment to maximize some notion of cumulative reward. Game theory links closely to this problem, especially to the notion of maximizing its reward. This machine learning approach differs from standard supervised learning since the correct input/output pairs are never presented, nor are sub-optimal actions explicitly corrected.

Semi-supervised learning stands between supervised and unsupervised learning, in that the teacher gives an incomplete training signal: a training

set with some, or many, of the target outputs missing. The reasoning behind observed, specific training cases to specific test cases is called *trasduction*. In the semi-supervised learning there is a case of this principle where the entire set of problem instances is known as learning time, except that part of the targets is missing.

Among other categories of machine learning problems, *meta learning* is a subfield where automatic learning algorithms are applied on metadata about machine learning experiments. The main goal is to use metadata to understand how automatic learning can become flexible in solving different kinds of learning problems, hence improving the performance of existing learning algorithms.
Flexibility is important because each learning algorithm is based on a set of assumptions about the data: its inductive bias. This means that it will only learn well if the bias matches the data in the learning problem. A learning algorithm may perform very well on one learning problem, but very badly on the next.

Developmental learning, instead, is elaborated for robot learning. Also, called developmental robotics, it is a scientific field which aims at studying the developmental mechanisms, architectures and constraints that allow lifelong and open-ended learning of new skills and new knowledge in embodied machines. As in human children, learning is expected to be cumulative and of progressively increasing complexity, and to result from self-exploration of the world in combination with social interaction. Here the main question is *"Can a robot learn like a child?"* [19].

Another categorization of machine learning tasks arises considering the desired *output* of a machine-learned system.

- In **classification**, inputs are divided into two or more classes, and the learner must produce a model that assigns unseen inputs to one or more of these classes (multi-label 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 patterns have to be divided into groups. Unlike in classification, the groups are not known beforehand. This is typically an unsupervised task.

- **Density estimation** finds the distribution of input patterns in some space.
- **Dimensionality reduction** simplifies inputs by mapping them into a lower-dimensional space. 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.

1.1.2 Computational learning theory

A learner needs to generalize from its experience. Generalization in this context is the ability of a learning machine to perform accurately on new, unseen tasks after having experienced a learning data set. The training examples come from some unknown probability distribution, which is considered representative of the space of occurrences. The goal of a learner is to build a general model about this space, enabling it to produce accurate predictions in new cases in a sufficient number of cases [6] [26].

Computational learning theory is the branch of theoretical computer science, including computational analysis of machine learning algorithms and their performance.
This branch mainly deals with a type of inductive learning called supervised learning, previously introduced. The goal of the supervised learning algorithm is to optimize some measure of performance, such as minimizing the number of mistakes made on new samples.
In addition to performance bounds, computational learning theory studies the time complexity and feasibility of learning. In computational learning theory, a computation is considered feasible if it can be done in polynomial time. There are two different types of time complexity results: positive if a certain class of functions is learnable in polynomial time, negative otherwise.
Negative results often rely with computational complexity $P \neq NP$, which shows that certain classes can not be learned in polynomial time [3].

The complexity of the hypothesis *should match* the complexity of the function underlying the data. The model will *underfit* the data if the hypothesis is less complex than the function. If the complexity of the model is increased in response, then the training error decreases. Instead, the model is subject to *overfitting* and generalization will be poorer if the hypothesis is too complex.

1.2 Computer Vision

Computer vision deals with how computers can understand things from digital image or videos. This field includes several tasks like acquiring, processing, analyzing and understanding digital images, and in general, deals with the extraction of high-dimensional data from the real world in order to produce numerical or symbolic information.
Understanding in this context means the transformation of visual images of the world into descriptions that can interface with other thought processes and elicit appropriate action [8] [20] [28].

Computer vision is sometimes seen as a part of the artificial intelligence field because they share topics such as pattern recognition and learning techniques.
Pattern recognition focuses on the recognition of patterns and regularities in data, although it is in some cases considered to be nearly synonymous with machine learning.
Some characterizations of computer vision appear relevant [41]:

- **Image processing** and **image analysis** focus on 2D and 3D images and how to transform one image to another.
- **Machine vision** applies a range of technologies and methods to provide imaging-based automatic inspection, process control and robot guidance in industrial applications. The focus on applications is mainly tended in manufacturing. Image sensor technologies and control theory are often integrated with the processing of image data to control a robot. That real-time processing is emphasised by means of efficient implementations in hardware and software.
- **Imaging** is a field that primarily focuses on the process of producing images, but also deals with processing and analysing of images. For example medical imaging includes substantial work on the analysis of image data in medical applications.
- **Pattern recognition**, finally, is a field which uses various methods to extract information from signals in general. It is mainly based on statistical approaches such as artificial neural networks. The application of these methods to image data is a significant part of pattern recognition field.

Computer vision takes a lot of computer science fields, but in this thesis we are going to analyze only what comes from pattern recognition.

1.2.1 Recognition

Recognition is a well-known problem in computer vision. The aim of this process is to determine whether or not the image data contains some specific object, feature, or activity. There are different varieties of the recognition problem in literature [2].

- **Object recognition**: an object inside an image or video sequence is found and identified. Humans recognize a multitude of objects in images with little effort, despite the fact that the image of the objects may vary in different view points, in many different sizes and scales or even when they are translated or rotated. Objects can even be recognized when they are partially obstructed from the view. This task is still a challenge for computer vision systems. Many approaches to this task, including machine learning systems, have been implemented over multiple decades and are studied in this thesis.

- **Identification**: an individual instance of an object is recognized. Examples include identification of a specific person's face or fingerprint, identification of handwritten digits, or identification of a specific vehicle.

- **Detection**: the image data are scanned for a specific condition. Examples include detection of possible abnormal cells or tissues in medical images or detection of a vehicle in an automatic road toll system.

Nowadays, the best algorithms for such tasks are based on convolutional neural networks, presented in the following chapter.

1.2.2 Image Classification

In this section we will introduce Image Classification: the problem of assigning an input image labelled from a fixed set of categories. This is one of the core problems in Computer Vision that has a large variety of practical applications.

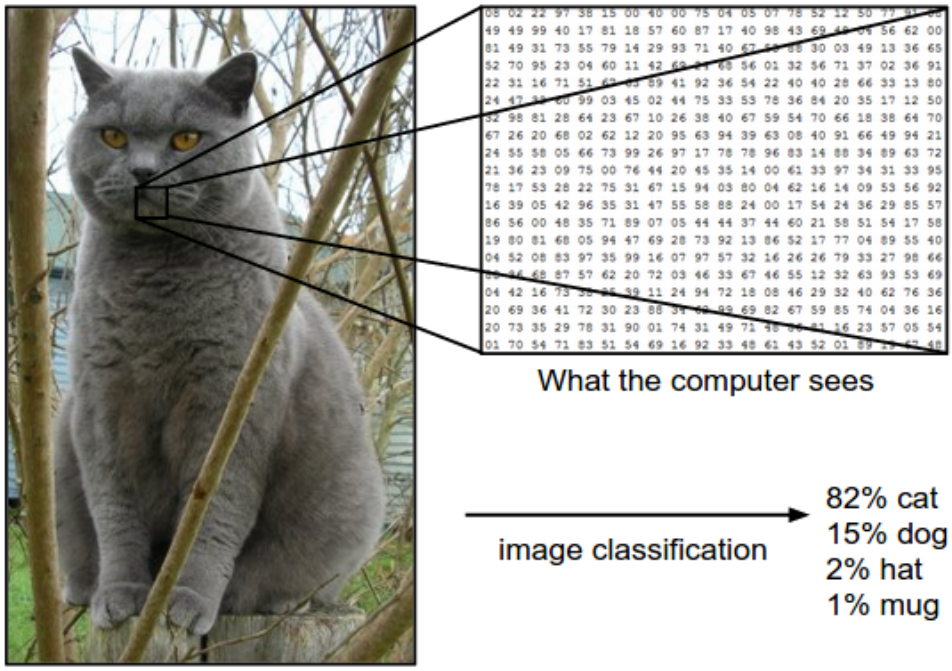


FIGURE 1.1: An image classification model that takes a single image and assigns probabilities to 4 labels: cat, dog, hat and mug [12].

From figure 1.1, we see an example of the classification task. For a human, recognizing a visual concept is relatively trivial. We have to consider this challenge from the perspective of a Computer Vision algorithm. A good image classification model must be invariant to the cross product of all variations inside the image, while simultaneously retaining sensitivity to the inter-class variations [12].
To obtain this, we are going to provide the computer with many examples of each class and then develop learning algorithms that look at these examples and learn about the visual appearance of each class. This approach is referred to as a data-driven approach, since it relies on first accumulating a training dataset of labeled images.

1.2.2.1 The image classification pipeline

In Image Classification, we take an array of pixels that represents a single image and assigns a label to it. The complete task can be formalized as follows:

- **Input:** we have a set of N images, each labeled with one of K different classes. We refer to these data as the *training set*.
- **Learning:** it consists of a training set to learn what every class looks like. We refer to learning as *training a classifier*.
- **Evaluation:** we need to evaluate the quality of the classifier. The model asks it to predict labels for a new set of images that it has never seen before, usually we call it a *test set*. We will then compare the true labels of these images to the ones predicted by the classifier. We have a true positive result if the match is proper, a false positive result otherwise.

1.2.3 Linear Classification

In the last paragraph we introduced Image Classification where a fixed set of proposed categories are used to assign a label to an image.
We have to introduce two components which we can naturally apply to both Artificial Neural Networks and Convolutional Neural Networks: a *score function* that maps the computed results to class scores, and a *loss function* which quantifies the correctness between the predicted results and the true labels.
A linear score mapping is the simplest possible function [14]:

$$f(x_i, W, b) = Wx_i + b \tag{1.1}$$

where: the image x_i has all of its pixels compressed to a single column vector of shape $[D\text{x}1]$, the matrix W $[K\text{x}D]$ and the vector b $[K\text{x}1]$ are the other function parameters.
In figure 1.2, we compute the score of the input image as a weighted sum of all of its pixel values across all three of its RGB color channels.

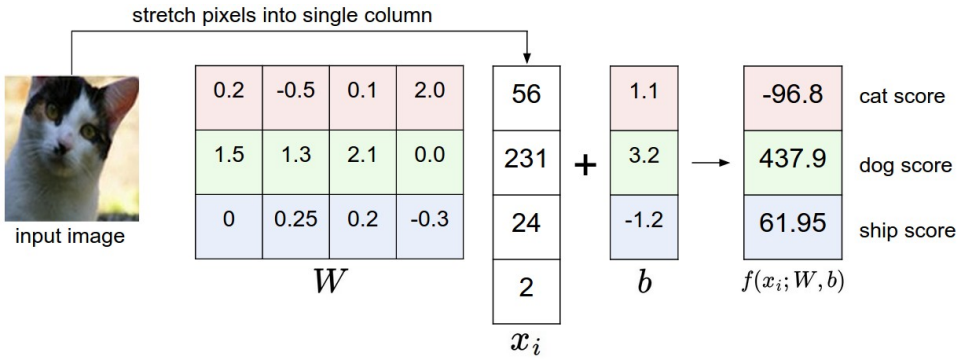


FIGURE 1.2: An example of mapping an image to class scores with four monochrome pixels. The classifier is convinced that the picture represents a dog [14].

The result depends precisely on what values we set for the weights, what

the linear score mapping function can like or dislike, which depends on the sign of each weight and certain colors at certain positions in the image.

We can interpret weights W as a collection of different *templates* one for class. Each template is compared to the input image using an *inner product* one by one to find the one that fits best. This is the score of each class for an image. The linear classifier completes template matching, where the templates are learned.

1.2.3.1 Loss function

The loss function is referred to the *"unhappiness"* with outcomes. The loss will be high if we are doing a poor job of classifying the training data, and it will be low if we are doing well.

One of the most common loss functions is the *Softmax classifier*. It gives an intuitive output, with normalized class probabilities, and also has a probabilistic interpretation. The function mapping does not change, but we now interpret these scores as the unnormalized log probabilities for each class [14].

$$L_i = -f_{y_i} + log \sum_j e^{f_j} \tag{1.2}$$

where the notation f_j mean the j-th element of the vector of class scores f. The full loss for the entire training set is the average of all the L_i for each of the training patterns together with an eventual regularization term $R(W)$ introduced to prevent overfitting.

1.3 Artificial Neural Networks

In machine learning, Artificial Neural Networks are a family of learning models inspired by biological neural networks.
Approximately 86 billion neurons can be found in the human nervous system and they are connected with approximately 10^{14}- 10^{15} synapses. In figure 1.3 a cartoon drawing of a biological neuron is shown. Each neuron receives input signals from its dendrites and produces output signals along its single axon. The axon eventually branches out and connects via synapses to dendrites of other neurons.

In figure 1.4 a computational model of a neuron is shown. We formalize the dendrites as the multiplication, $w_0 x_0$, between the axon x_0, where information is travelling, and the synapse w_0, an input weight of the neuron. The weights w, are learnable, and control the strength of influence, including their direction: excitory (positive weight) or inhibitory (negative

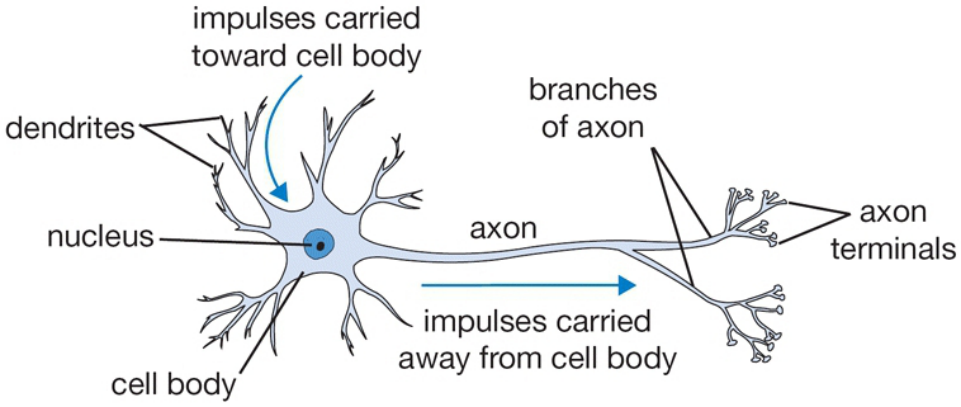


FIGURE 1.3: A cartoon drawing of a biological neuron [15].

weight), of one neuron on another. The signal is carried by the dendrites to the cell body where all inputs are summed. If the final sum is above a certain threshold, the neuron can fire, sending a spike along its output axon. The precise timing of spikes does not matter, but only the frequency of the firing communicates information as needed.

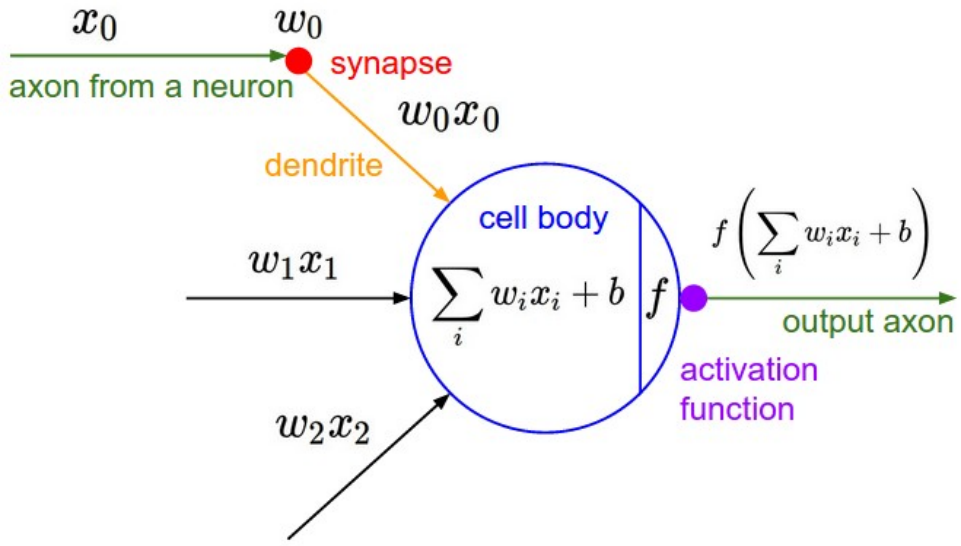


FIGURE 1.4: The mathematical model of the previous neuron [15].

This rate code, or firing rate of a neuron, is modeled by the *activation function f*. This definition represents the frequency of the spikes along the axon. In the end each neuron performs a dot product with the input axons and its weights and adds the bias and applies the activation function. This will be the output fires from a neuron.

$$\sigma(x) = \frac{1}{(1 + e^{-x})} \tag{1.3}$$

In the following paragraph we will describe some of the most common activation functions.

1.3.1 Activation functions

In the previous paragraph we defined activation functions as the firing rate of a neuron; in the following we introduce some of the most common models of this notion.

1.3.1.1 Sigmoid

The sigmoid non-linearity has been previously defined in equation 1.3. It takes a real-valued number and fits it into range between 0 and 1. Historically, the sigmoid function has seen frequent use since it has the interpretation of firing rate previously described: from not firing at all, the zero value, to fully-saturated firing at an assumed maximum frequency, the one value. The sigmoid non-linearity has two disadvantages [15]:

1. *Sigmoids saturate and kill gradients*. A property of the sigmoid neuron is that when the neuron's activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero. This behaviour will *kill* the gradient and returns almost no signal that flows through the neuron to its weights and recursively to its data.

2. *Sigmoid outputs are not zero-centered*. This is an undesirable behaviour since neurons in a Neural Network would be receiving data that is not zero-centered. During gradient descent, this disadvantage has implications on the dynamics, because if the information coming into a neuron is always positive, then the gradient on the weights will become either all positive, or all negative, during backpropagation, presented in the following section. For this purpose the **tanh non-linearity** is preferred.

1.3.1.2 ReLU

The Rectified Linear Unit has become very popular in the last few years. It computes the function

$$f(x) = max(0, x) \tag{1.4}$$

There are several strengths and consequences to using the ReLUs.
The first strength is that the ReLU accelerates the convergence of stochastic gradient descent compared to the sigmoid/tanh functions. It is argued that this is due to its linear, non-saturating form. The ReLU can also be implemented by simply thresholding a matrix of activations at zero; compared to

tanh/sigmoid neurons that involve complicated operations (like exponentials).
On the other hand, ReLU units can be fragile during training. In fact, if a large gradient flows through a ReLU neuron, it could cause the weights to update in such a way that the neuron will never activate on any datapoint again.

1.3.2 Feedforward Neural Network architecture

Feedforward Neural Networks are modeled as collections of neurons that are connected in an acyclic graph, the outputs of some neurons can become inputs to other neurons.

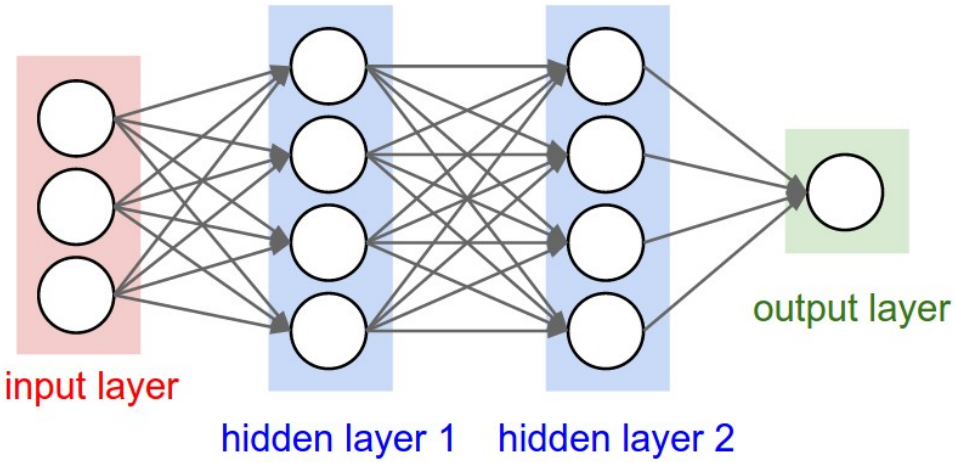


FIGURE 1.5: A 3-layer neural network [15].

Cycles are not allowed since that would imply an infinite loop in the forward pass of a network, here feedforward. These models are often organized into distinct layers of neurons. The most common layer type is the *fully-connected layer* in which neurons between two adjacent layers are fully connected pairwise. Neurons in a single layer share no connections.

1.3.3 Data preprocessing

Data processing techniques can be useful to avoid numerical problems and speed-up convergence.
The first one is called **mean subtraction**. It is the most common form of preprocessing and involves subtracting the mean across every individual feature in the data. This is the geometric interpretation of centering the cloud of data around the origin along every dimension.

$$X- = mean(X, axis = 0) \tag{1.5}$$

The second one is called **normalization** and it refers to normalizing the data dimensions. After this processing they are approximately the same scale. There are two common ways of achieving this normalization. One is to divide each dimension by its standard deviation, once it has been zero-centered.

$$X/ = std(X, axis = 0) \tag{1.6}$$

The second way is by normalizing each dimension. The min and max along the dimension is -1 and 1 respectively. It only makes sense to apply this preprocessing if you have a reason to believe that different input features have different scales, but they should be of approximately equal importance to the learning algorithm. These data preprocessing are used throughout this thesis especially in chapter 5 when we will see the deep learning phase.

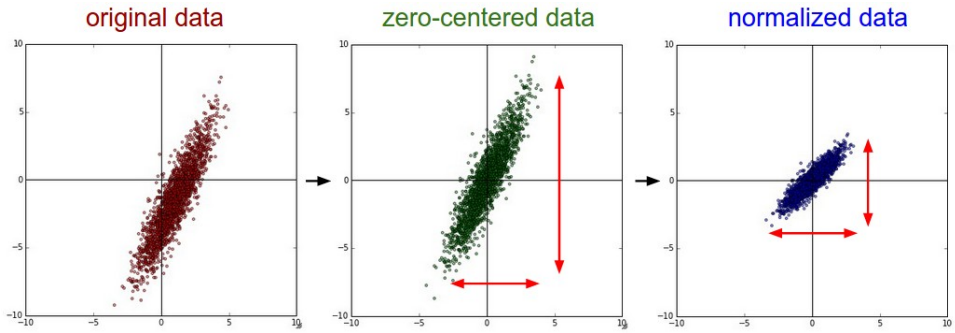


FIGURE 1.6: Common data preprocessing pipeline [16].

1.4 Backpropagation algorithm

In this section we present the concept of backpropagation. The understanding of this process is critical because it is the common algorithm with which Artificial Neural Networks are trained. The problem can be stated as follows: *given some function $f(w, x)$ where x is a vector of inputs, we are interested in computing the gradient of f with respect to w.*

More specifically, we could define N as a neural network with e connections, where $x, x_1, x_2, \ldots, x_k \in \mathbb{R}^n$ are the *input* vectors, $y, y', y_1, y_2, \ldots, y_l \in \mathbb{R}^m$ are the *output* vectors and $w, w_0, w_1, \ldots, w_s \in \mathbb{R}^e$ are the *weights* vectors. The neural network corresponds to a function

$$y = f_N(w, x) \tag{1.7}$$

which, given a set of weights w maps an input x to an output y. We select an error function $E(y, y')$ measuring the difference between two outputs. A typical choice is

$$E(y, y') = |y - y'|^2 \tag{1.8}$$

the square of the *Euclidean distance* between the vectors y and y' where the former is the actual predicted output and the latter is the real label [31].

The backpropagation algorithm takes as input a sequence of NN *training examples* $(x_1, y_1), \ldots, (x_p, y_p)$ and produces a sequence of weights $w_0, w_1, \ldots, w_p$ starting from some initial weight w_0, usually chosen at random. These weights are computed in turn: we compute w_i using only (x_i, y_i, w_{i-1}) for $i = 1, \ldots, p$. The output of the backpropagation algorithm is then w_p, giving us a new function

$$x \mapsto f_N(w_p, x) \tag{1.9}$$

The computation is the same in each step, we describe only the case $i = 1$. We find w_1 from (x_1, y_1, w_0) by considering a variable weight w and applying gradient descent to the function $w \mapsto E(f_N(w, x_1), y_1)$ to find a local minimum, starting at $w = w_0$. We then let w_1 be the minimizing weight found by gradient descent. This iterative optimization algorithm aims to find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient of the function at the current point.

By steps the backpropagation learning algorithm can be divided into two phases [40]:

1. **Propagation**, which involves the following steps:
   (a) Forward propagation of a training pattern's input through the neural network in order to generate the output activations.
   (b) Backward propagation of the output activations through the neural network using the training pattern target in order to generate the deltas of all the output and hidden neurons.
2. **Weight update**, which involves the following steps:
   (a) Multiply its output delta and input activation to get the gradient of the weight.
   (b) Subtract a *percentage* from the gradient of the weight.

This ratio influences the speed and quality of learning; it is called the *learning rate*. The greater the ratio, the faster the neuron trains, but the lower the ratio, the more accurate the training is. The sign of the gradient of a weight indicates where the error is increasing, this is why the weight must be updated in the opposite direction.

1.5 Deep Learning

Deep learning, known as deep structured learning, hierarchical learning or deep machine learning, is a branch of machine learning based on a set of algorithms which attempt to model high level abstractions in data by using a deep graph with multiple processing layers, composed of multiple linear and non-linear transformations [21].
Deep learning is part of a broader family of machine learning methods based on learning representations of data. An observation, an image in our case, can be represented in many ways such as a vector of intensity values per pixel, or in a more abstract way as a set of edges, regions of particular shape. Some representations are better than others at simplifying the learning task. One of the promises of deep learning is replacing handcrafted features with efficient algorithms for unsupervised or semi-supervised feature learning and hierarchical feature extraction.
In this area the researchers attempt to make better representations and create models to learn these representations from large-scale unlabelled data. Some of the representations are inspired by advances in neuroscience and are loosely based on interpretation of information processing and communication patterns in a nervous system, such as neural coding which attempts to define a relationship between various stimuli and associated neuronal responses in the brain [33].
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 [7].

In the following chapter 2 we will explain the architectures of neural networks behind Deep Learning, which is the topic of this thesis.

1.6 Incremental Learning

In recent years, deep learning confirmed its strength in several tasks such as speech recognition, natural language processing and computer vision. The power of Deep Learning can be really exploited if there is a vast amount of data. This is not always the case of real world applications where training data are often only partially available at the beginning and new data keeps coming while the system is already deployed and working. This is exactly the kind of context in which Incremental learning seems the rightful paradigm to use. Of course, we would have two different ways to deal with

this peculiar scenario. The first one is to store all the previously seen past data and retrain the model from scratch as soon as a new batch of data is available. This method is called *cumulative approach*. However, this solution is impractical for many real world systems where memory and computational resources are limited.
The second path is to update the model incrementally based only on the new available batch of data; this is computationally cheaper and storing the entire history of data is not required. it is worth noting that this feature is a very interesting one to posses, especially for a deep learning model where we work with a huge amount of data; and this is the actual incremental learning paradigm. Nevertheless, this method could bring to a substantial loss of accuracy with respect to the cumulative approach.
In the optimal case, we could also deal with semi-supervised learning scenarios where a small set of labelled data and a larger set of unlabelled data from the same classes are available from the beginning. We assume that the unlabelled data are not available initially and become available only at successive stages, once the system has already been trained and the new data are used for incremental tuning. This matches with the human-like scenario previously described.
It has been proved that in incremental learning the stability-plasticity dilemma may arise and dangerous shifts on the model are always possible because of catastrophic forgetting [25]. Forgetting previously learned patterns can be conveniently seen as overfitting the new available training data. As we can understand from this introduction, the incremental scenario is much more complex than overfitting a single, fixed-size training set [37].

Deep Incremental Learning, that is Incremental Learning performed with Deep Learning techniques will be the central subject of this dissertation.

Chapter 2

CNN for Object Recognition

In this chapter we will describe the Convolution Neural Networks algorithm (CNNs) from the underlying theory to the object recognition tasks. After a brief overview where we will introduce the different layers, we will describe different CNN architectures and the concept of training and transfer learning.

2.1 Convolutional Neural Network: an Overview

Convolutional Neural Networks (CNNs) are similar to Feedforward Neural Networks described in chapter 1: they are made up of neurons that have learnable weights and biases. Each neuron gains some inputs, performs a dot product eventually followed by a non-linearity. At the end, the whole network can be expressed as a single differentiable score function, ranging from the raw image pixels on one end to the class scores at the other. They still have a loss function, as it has been previously demonstrated, on the last fully-connected layer.

Historically CNNs architectures explicitly assumed that inputs are images, permitting us to encapsulate specific properties into the architecture. After all, these features make the forward function more efficient decreasing the total number of parameters inside the network.

As we saw in the previous chapter, Neural Networks receive an *input*, a single vector, and transform it through a series of *hidden layers*. Each hidden layer is made up of a set of neurons, where each neuron is fully connected to all neurons in the previous layer, and where neurons in a single layer function completely independently and do not share any connections. The last fully-connected layer is called the *output layer* and in classification settings it represents the class scores [13].

Providing an example would assist in demonstrating this theory. In this thesis, as we will see in the following chapters, we have an image of size 128x128x3 (128 wide, 128 high, 3 color channels). A single fully-connected

neuron in a first hidden layer of a regular Neural Network would have 128x128x3 = 49152 weights. Moreover, several neurons would be desired, meaning that the parameters would add up quickly. This full connectivity can be wasteful and the huge number of parameters would most likely lead to overfitting.

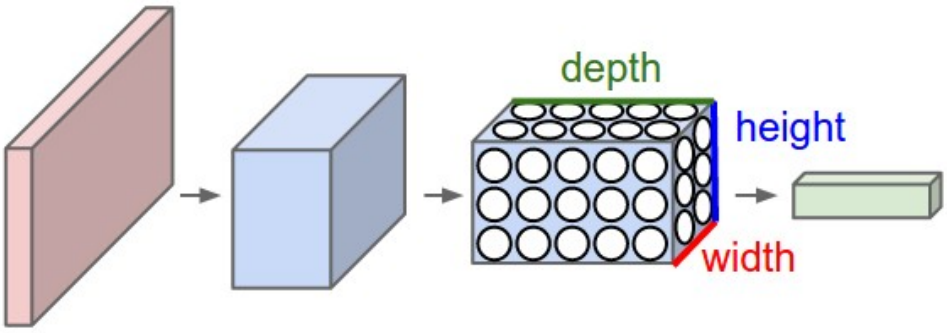


FIGURE 2.1: A CNN who arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers [13].

CNN takes advantage of the fact that the input consists of images, which constrain the architecture in a more practical way. Unlike a regular Neural Network, the layers of a CNN have various neurons that are arranged in 3 dimensions: width, height, and depth (as we can see in the previous image 2.1).

2.1.1 The convolution operations

Convolution is an important operation in signal and image processing. Convolution operates on two signals, *in one-dimension*, or two images, *in two-dimension*. One can be understood as the *input* signal, and the other, which is called the *kernel*, is best understood as a *filter* on the input image. Producing an output image, convolution takes two images as input and produces a third as output. When using a convolution, we obtain an image that highlights the characteristics of the filter used.
We have different filters for the image processing [39]:

- **Blur**: completed by taking the average between the current pixel and its n neighbours, where n determines the filter effectiveness.
- **Find edges**: a filter which finds the horizontal edges.
- **Sharpen**: with this filter, the resulting image will resemble the other, resulting in a new image where the edges are enhanced, making it appear sharper.
- **Emboss**: this filter gives a 3D shadow effect to the image, resulting in a very useful for a bump map of the image.

FIGURE 2.2: Examples of convolution filter, from left to right: blur, find edges, sharpen and emboss.

We can distinguish between three different types of convolution [30]:

- **One to one convolution**: when we could apply a filter to a single image.
- **One to many convolution**: when there is only one input image and n filters, each filter is used to generate a new image, which is labeled the *feature map*.
- **Many to many convolution**: when there are m input images and n output images. Each connection between an input and an output image is distinguished with different filters. The total number of connections increases. The computational time in this case will quickly escalate.

2.2 Layers used to build CNN

A simple CNN is composed of a sequence of layers, and every layer of a CNN transforms one volume of activations to another through a differentiable function. In literature, three main types of layers are described to build a CNN architectures. Let's recall that the ReLU activation function was already introduced in chapter 1.

2.2.1 Convolutional Layer

The Convolutional layer parameters are composed of a set of learnable filters. Every filter is small spatially (along width and height), but extends through the full depth of the input volume. A typical filter on a first layer of a ConvNet will most likely have a size of 5x5x3 (5 pixels wide and high and 3 color channel).
Each filter is convolved across the width and height of the input volume and dot products are computed between the entries of the filter and the input at any position. As the filter is slid over the width and height of the input volume, a 2-dimensional activation map that gives the responses of that filter (at every spatial position) will be produced. Filters that activate

when they see some type of visual feature (starting from edge of a specific orientation or a color blotch and even an entire honeycomb or wheel-like pattern) are then learned by the network.

As we learned in the previous chapter, when dealing with high-dimensional inputs like images, it is not always practical to connect neurons to all neurons in the previous volume. Instead, each neuron should be connected to each neuron with only a local region of the input volume.
The spatial extent of this connectivity is an hyperparameter, the *receptive field* of the neuron (also known as filter size). Along the depth axis, the extent of the connectivity is always equal to the depth of the input volume.

An asymmetry also exists between spatial dimension and the depth dimension. The connections are local in space (width and height), but it is always full along the entire depth of the input volume.

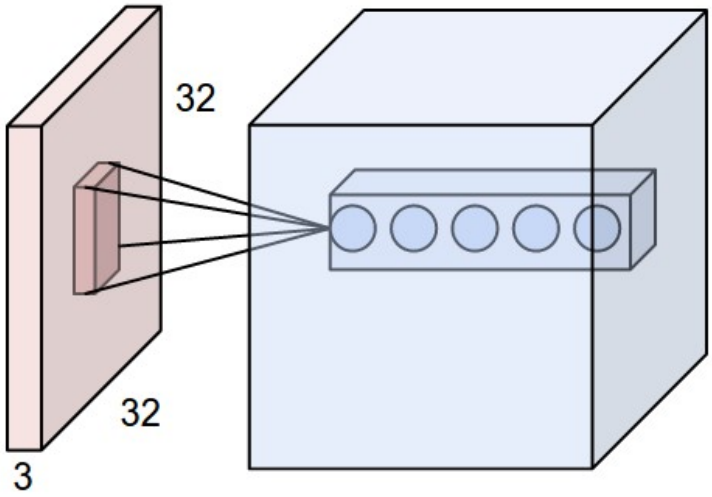


FIGURE 2.3: An example applied to a 32x32x3 image (in red). The volume of neurons in the convolutional layer is connected only to a region in the input volume spatially, but to the full depth [13].

Here are the following hyperparameters, which arrange the output volume:

- **Depth** of the output volume is a hyperparameter: it corresponds to the amount of filters used, each learning to look for different things in the input. For example, if the first Convolutional Layer understands the raw image as input, then different neurons along the depth dimension may activate in the presence of various oriented edged, or blobs of color. The *depth column* will hereby refer to a set of neurons that are all looking at the same region of the input.

- The **stride** is to be understood as how we slide the filter. When the stride is 1, the filters move one pixel at a time. When the stride is 2

then the filters jump 2 pixels at a time, and so forth. This will produce smaller output volumes.

- The size of **zero-padding** is also an hyperparameter. Indeed, sometimes it might be convenient to pad the input volume with zeros along the border. Zero padding allows us to control the spatial size of the output volumes, which is an exciting feature.

The following formula is indicated for calculating how many neurons *fit* is given [13]

$$\frac{(W - F + 2P)}{S} + 1 \tag{2.1}$$

where W is the input volume size, F the receptive size of the convolutional layer neurons, S the stride and P the amount of the zero padding used.

A real world example is the Krizhevsky architecture that won the ImageNet [1] challenge in 2012 accepted images of size [227x227x3]. On the first Convolutional Layer, it used neurons with receptive field size $F = 11$, stride $S = 4$ and no zero padding $P = 0$. Since $(227 - 11)/4 + 1 = 55$, and since the Conv layer had a depth of $K = 96$, the Convolutional layer output volume had size [55x55x96]. Each of the 55x55x96$= 290.400$ neurons in this volume was connected to a region of size [11x11x3] in the input volume. All 96 neurons in each depth column are connected to the same [11x11x3] region of the input, 11x11x3$= 363$ weigths and one bias.
This two adds up to 290400x364$= 105.705.600$ parameters on the first layer.

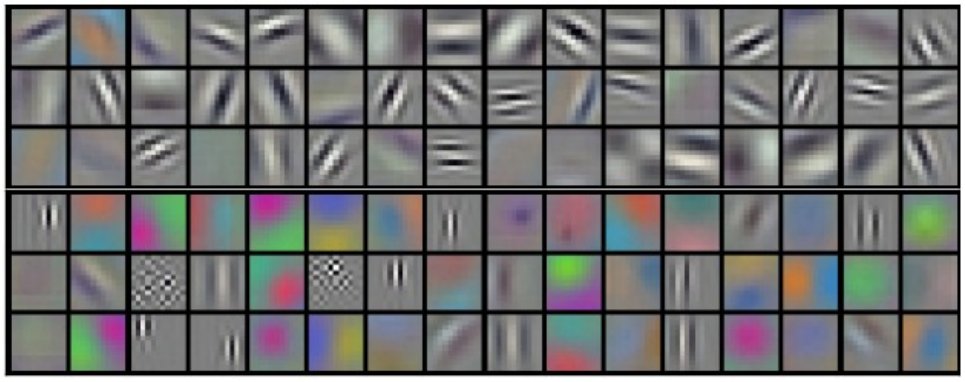


FIGURE 2.4: Example filters learned by Krizhevsky. Each of the 96 filters shown here is of size [11x11x3], and each one is shared by the $55x55$ neurons in one depth slice [13].

This number is very high. To reduce the number of parameters we need to make one assumption.
We compute spatial position (x, y) so we should denote a single 2-dimensional slice of depth as a *depth slice* (in the previous example there are 96 each of size [55x55]). The total parameters number will be lower than before:

96x11x11x3= 34.848 and 96 bias [13].

The forward pass of the Convolutional layer can (in each depth slice) be understood as a convolution of the neuron's weights with the input volume. This is only true if all neurons in a single depth slice are using the same weight vector.
The parameter sharing assumption does not always add up. For example, when faces are used as inputs, some difficulties may arise. This is because input images have some specifically centered structures, such as different eye or hair, for example.
Ultimately, the Convolutional operation performs dot products between the filters and local regions of the input. This is shown in the following example.

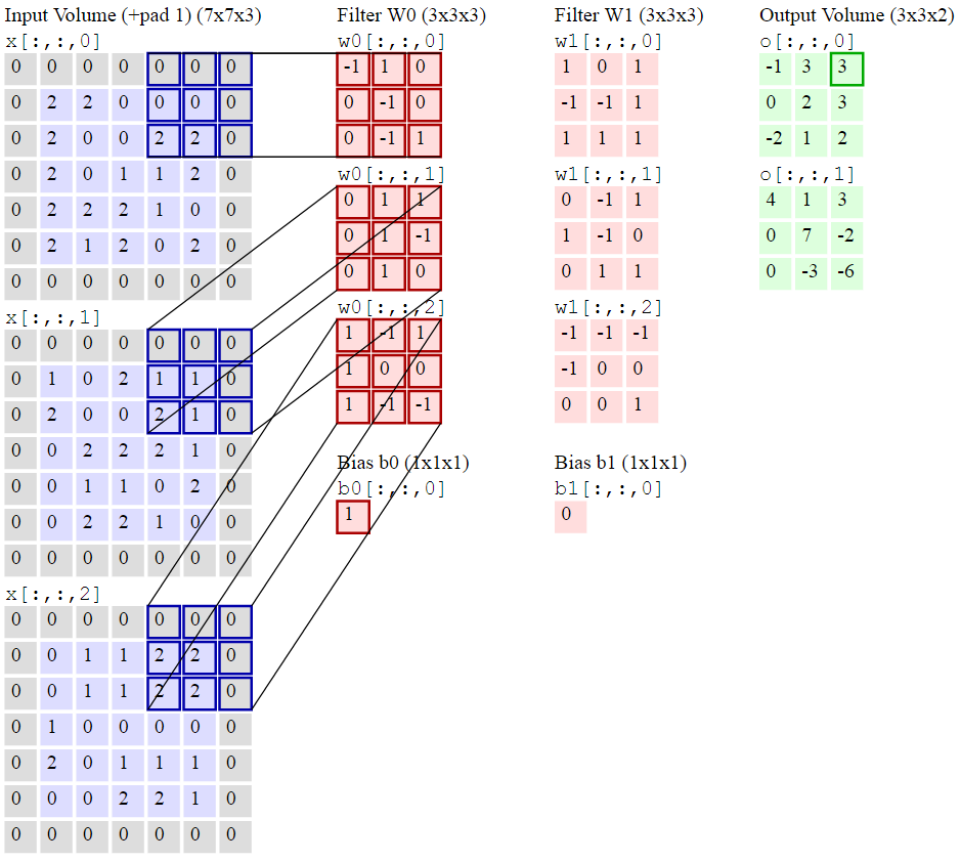


FIGURE 2.5: Dot products example between filters (red) and local regions (blue). The green one are the output activations [13].

A Convolutional layer is formally described as:

- Accepts a volume size of W_1xH_1xD_1, widht, height and depth respectively;
- Requires four hyperparameters: K number of filters, F spatial extent, S stride and P amount of zero padding;
- Produces a volume size of W_2xH_2xD_2 where:
  - $W_2 = (W_1 - F + 2P)/S + 1$;
  - $H_2 = (H_1 - F + 2P)/S + 1$;
  - $D_2 = K$.
- When using parameter sharing, the following is introduced: $FxFxD_1$ weights per filter;
- The d-th depth slice (sized W_2xH_2) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S.

2.2.2 Pooling Layer

A Pooling layer is common to periodically insert after Convolutional layers. The function of the pooling layer is to progressively reduce the spatial size of the representation and to also reduce the amount of parameters and computation in the network. Hence, this controls overfitting. The Pooling Layer independently operates on every depth slice of the input and spatially resizes it, using the maximum operation. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2 down samples. Further, every depth slice is in the input by 2 along both width and height, discarding 75% of the activations.

Formally a Pooling layer is described as:

- Accepts a volume size of W_1xH_1xD_1, widht, height and depth respectively;
- Requires two hyperparameters: F spatial extent and S stride;
- Produces a volume size of W_2xH_2xD_2 where:
  - $W_2 = (W_1 - F)/S + 1$;
  - $H_2 = (H_1 - F)/S + 1$;
  - $D_2 = D_1$.
- Introduces zero parameters since it computes a fixed function of the input;

- Pooling layers does not require the use of the zero padding parameters.

Sometimes in addition to max pooling, the pooling units can also perform other functions, such as *average pooling*, which was used historically but its popularity has since decreased.

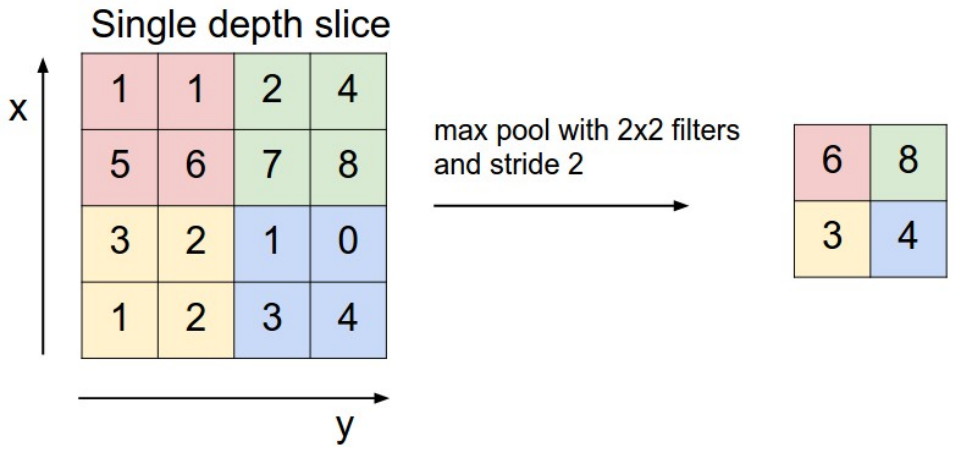


FIGURE 2.6: An example of max pooling with a stride of 2 [13].

2.2.3 Fully-connected Layer

Neurons in a fully connected layer have complete connections to all activations in the previous layer, as was described in the previous chapter, Artificial Neural Networks. Therefore, their activations can be computed with a matrix multiplication followed by a bias offset.

2.3 CNN Architecture

As previously described, there are four layer types: CONV (Convolutional), POOL (Max Pooling), FC (Fully-Connected) and RELU (seen in the previous chapter). In this section we introduce how these are commonly used together to form an entire Convolutional network.

2.3.1 Layer Patterns

The most common form of a ConvNet architecture stacks a few CONV-RELU layers and follows them with POOL layers. This pattern is repeated until the image has been merged spatially to a small size. Towards the end of the network, it is common to use fully-connected layers, instead. The last fully-connected layer holds the output, such as the class scores. In other words, the most common ConvNet architecture follows the pattern:

IN -> [[CONV -> RELU]*N -> POOL?]*M -> [FC -> RELU]*K -> FC

where $*$ indicates repetition and POOL? indicates an optional Pooling layer. Usually $0 \leq N \leq 3$ and $0 \leq K < 3$.

2.3.2 Layer Sizing Patterns

In this section the common hyperparameters used in each of the layers in a Convolutional network are discussed.

- The *input layer* should be divisible by two many times. Common numbers include 32, 64, 96, 128 and 224.
- The *convolutional layer* should be using small filters. Usually $S = 1$ and padding the input volume with zeros in such a way that the convolutional layer does not alter the spatial dimensions of the input. For a generall F, $P = (F - 1)/2$, for small filter size.
- The *pooling layer* deals with down sampling the spatial dimensions of the input. The most common setting is to use max-pooling with 2x2 receptive fields ($F = 2$), and with a stride of 2 ($S = 2$).

2.3.3 Case studies

In the field of Convolutional Networks there are several successful and well known architectures that have reached a high accuracy score level. The most common are [13]:

- **LeNet**: the first successful applications of Convolutional Networks were developed by Yann LeCun in the 1990's. In these applications the LeNet architecture was used to read zip codes, digits, etc.

- **AlexNet**: the first work that popularized Convolutional Networks in Computer Vision was the AlexNet, developed by Alex Krizhevsky, Ilya Sutskever and Geoff Hinton. The AlexNet had a very similar architecture to LeNet, but was deeper, bigger, and featured Convolutional Layers stacked on top of each other.
- **ZF Net**: the ILSVRC 2013 winner was a Convolutional Network from Matthew Zeiler and Rob Fergus, which became known as ZFNet (short for Zeiler & Fergus Net). This improved on AlexNet's features with architecture hyperparameters enhancements.
- **GoogleLeNet**: the ILSVRC 2014 winner was a Convolutional Network from Szegedy from Google. Its main contribution was the development of an Inception Module that dramatically reduced the number of parameters in the network. This paper uses Average Pooling instead of Fully Connected layers at the top of the ConvNet, eliminating a large amount of needless parameters.
- **VGGNet**: the runner-up in ILSVRC2014 was the network from Karen Simonyan and Andrew Zisserman, later known as VG- GNet. Its main contribution was that it showed the critical component of the depth of the network for good performance. Their final best network contains 16 CONV/FC layers and, interestingly, features an extremely homogeneous architecture that only performs 3x3 convolutions and 2x2 pooling from start to finish. Their pretrained model is available for plug and play use in Caffe. One negative aspect of the VGGNet is that it is more expensive to evaluate and uses a higher amount of memory and parameters (140M). Most of these parameters are in the first fully connected layer, and it was since found that these FC layers can be removed with no performance downgrade, significantly reducing the number of necessary parameters.

2.4 CNN Training

In the previous chapter, the backpropagation algorithm (often called **backprop**) was described. The same algorithm is used for training Convolutional Neural Networks. This allows the information from the Loss to flow backwards through the network, in order to update its weights accordingly.

2.4.0.1 Loops for Convolution

The following rules are often used to implement the forward and backward propagation:

$$x_j^{L+1} = \sum_i w_{j,i}^{L+1} x_i^L \tag{2.2}$$

$$g_i^L = \sum_j w_{j,i}^{L+1} g_j^{L+1} \tag{2.3}$$

where x_i^L and g_i^L are respectively the activation and the gradient of unit i at layer L, and $w_{j,i}^{L+1}$ is the weight connecting unit i at layer L to unit j at layer $L+1$.

This can be understood as the activation units of the higher layer *pulling* the activations of all the units to which they are connected. The pulling strategy is complex and difficult to implement when computing the gradients of a convolutional network, because of the number of connections. Basically, in a convolution layer, the number of connections leaving each unit is not constant due to border effects. To simplify this computation, instead of pulling the gradient from the lower layer, the gradient from the upper layer can be *pushed*. The resulting equation is:

$$g_{j+1}^L = w_i^{L+1} + g_j^{L+1} \tag{2.4}$$

For each unit j in the upper layer, fixed number of (incoming) units i from the lower layer is updated. Since weights are not shared in convolution, w does not depend on j.

Chapter 3

Benchmarks for Object Recognition

In the previous chapters we described the necessary background for the introduction of the thesis topic. In this chapter we introduce the concept of benchmarking related to Object Recognition described in chapter 1, the existing benchmarks for Incremental Learning (along with their limitations) and in the last section we describe the improvements introduced in this thesis aimed to improve these deficiencies.

3.1 Performances evaluation: why

A benchmark is the act of running a computer program, a set of programs, or other operations, in order to assess the relative performance of an object, normally by running a number of standard tests and trials against it.
In this thesis the benchmark concept is almost the same, but it is applied to the performance accuracy of a trained CNNs. The accuracy is chosen as *score function* introduced in chapter 2 and it is based on the model predictions. The Deep Learning framework Caffe, used to run experiments, prints this value every time it has scheduled a test on the network within the solver. We will introduce Caffe in the next chapter and also the *loss function*. The details of the experiments will be described in the following chapter 5.

Benchmarking a Deep Learning experiment is needed to compare different strategies like new CNN architectures, tuning of hyperparamenters or testing the same network on different datasets.
In this thesis we test CNNs on existing benchmarks, and on a new dataset specific for incremental learning.
We will compare the new dataset for incremental learning with other datasets which were previously used for incremental learning studies by Maltoni and Lomonaco [37]. This paper computes the benchmark of these datasets.

A relevant contribution of this thesis is to propose a new benchmark for deep incremental learning. The benchmarks have not to be too hard, but at the same time not too easy. They must therefore be a good compromise between this two extremes and they must ensure a good starting point for new experiments. This is specifically discussed in the last chapter.

3.2 Existing benchmarks for Incremental Learning

In this section we present the incremental tuning strategies and the existing dataset described in [37].
There are three different strategies to deal with an incremental tuning/learning scenario.

- Training or tuning an ad-hoc CNN architecture suitable for the problem;
- Fine-tuning an already trained CNN;
- Using an already trained CNN as a fixed feature extractor in conjunction with an incremental classifier.

By *fine-tuning*, we mean taking an already learned model, adapting the architecture, and resuming the training from the already learned model weights.
An incremental classifier can be trained starting from the features extracted from a predefined level of a CNN.
As we will see in the following sections, we will use all the listed strategies.

In the next paragraphs we will describe the labeled image datasets suitable for incremental learning, which means that the objects have been acquired in a number of successive sessions where the environmental condition, like background or lighting, can change among or also during any session.

3.2.1 iCubWorld28

iCubWorld28 is a relevant dataset to study incremental learning in the robotics field. iCub is a 1 meter high humanoid robot suitable for research into human cognition and artificial intelligence.

The resultant *iCubWorld28* dataset is nothing more than the perspective *robot-side* of the object to recognize hand held by a human supervisor. These captured images are used for training the CNN and allows iCub to recognize the object.
The dataset [17] is composed by 28 objects organised into 7 categories 3.1.

FIGURE 3.1: The 28 objects from one of the 4 datasets [17].

Each image, sized 128x128 pixels in RGB format, is a frame of the acquisition session of a single object. Each video recording session lasts approximately 20 seconds where the object is slowly moved and rotated in front of the iCub camera. Each set (train and test) included 220 images, respectively. This acquisition run over four days and results in 4 datasets of more than 12K images each.
In the following table we report results from [17] where the accuracy of predictors trained on a single day compared with a predictor trained on all days together are related to the *iCubWorld28* dataset.

		TEST Accuracy (%)				
		Day 1	**Day 2**	**Day 3**	**Day 4**	**Average**
TRAIN	**Day 1**	67.7	41.9	37.2	67.2	53.5
	Day 2	40.1	67.8	35.4	66.8	57.5
	Day 3	62.0	63.5	66.4	64.9	64.2
	Day 4	62.9	64.1	65.3	67.1	64.8
	All Days	73.4	71.0	68.1	68.9	70.3

As we will introduce in the following chapter the new TCDR4 was inspired by this dataset.

3.2.2 BigBrother

The *BigBrother* dataset [37] has been created starting from 2 DVDs made commercially available at the end of the 2006 edition of the *Grande Fratello* Italian reality show where 20 participants' 99 day stay in a large house is documented.

This dataset consists of 23.842 gray-scale images of the faces of 19 competitors (the first one was immediately eliminated at beginning of the reality show). Each image is 70x70 pixels.

As with the previous dataset, training and test sets are defined. An additional large set of images (*the updating set*), was also created. This set is provided for incremental learning and tuning purposes. A restricted version of the dataset is also available to focus on the subset of participants who remained in the house for a long time and for whom we can expect better precision.

3.2.3 NORB dataset

NORB stands for *New York University Object Recognition Benchmark*, it is a well-known and largely used datasets proposed by LeCun in 2004. This is one of the best dataset to study invariant object recognition and best fits our purposes because it contains 50 objects and 972 variations for each objects. The 50 objects belong to 5 classes, 10 objects per class, and the 972 variations are produced by systematically varying the camera elevation (9 steps), the object azimuth with respect to the camera (18 steps) and the lighting condition (6 steps). A total of 194.400 RGB images at 640x480 pixels resolution. The collection consists of 10 instances of 5 generic categories, figure 3.2: four-legged animals, human figures, air planes, lorry, and cars.

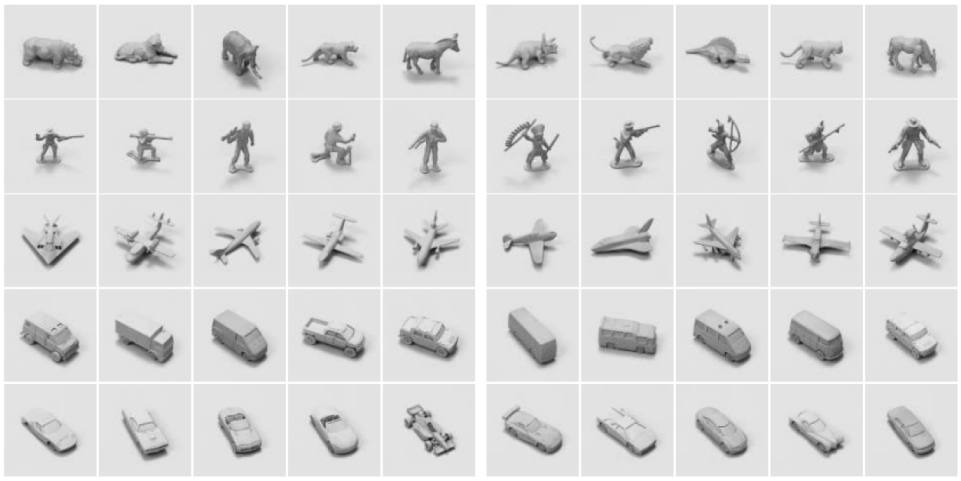


FIGURE 3.2: The 50 different object instances in NORB dataset [38].

All the objects were painted with a uniform bright green. The uniform color ensured that all irrelevant color and texture information was eliminated.

Temporally coherent video sequences can be generated from NORB [38] by randomly walking the 3D (elevations, azimuth, lighting) variation space,

where consecutive frames are characterized by a single step along one dimension. In the standard NORB benchmark for each of the 5 classes, 5 objects are included in the training set and 5 objects in the test set.

3.3 Limitation of existing benchmark

In this section we will introduce the latest results in literature built up on the previously described datasets and using these results will explain the limitation of these existing datasets and benchmarks for incremental learning studies.

3.3.1 Experiment on existing benchmark

The first experiment reported is related to the iCubWorld28 dataset. As we could analyze from the following graph 3.3; all three of the previously described strategies have been used.

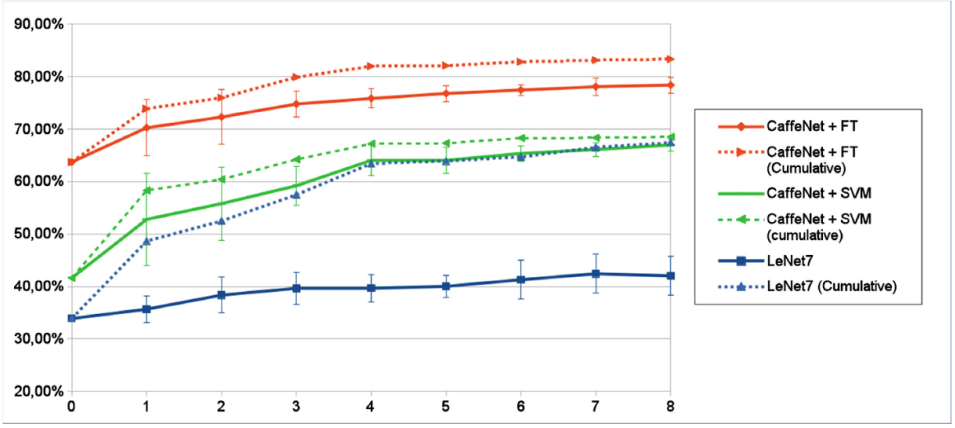


FIGURE 3.3: Averaged accuracy results for all three strategies [37].

- **LeNet7**: proposed by Yan LeCun in 2004. The simple CNN and the small amount of training data do not allow to learn complex features (such as rotations, partial occlusions, and the removal of the object from the camera).
- **CaffeNet + SVM**: CaffeNet is a Caffe library called *Model Zoo* (*BVLC Reference CaffeNet*) which is based on *AlexNet* (as we have seen previously). SVM classifier can be trained incrementally. This strategy has quite a good recognition rate increment along the batches, as we can see from the graph.

- **CaffeNet + FT**: this is the most effective strategy for this dataset because the features originally learned on the ImageNet dataset are general and this dataset can be thought as a specific sub-domain where feature fine-tuning can help pattern classification.

The second experiment is related to the BigBrother dataset. The following graph shows CaffeNet accuracy on this challenging dataset where control of forgetting is not easy. In fact, in BigBrother dataset there is a high variation in the number of patterns in different incremental batches.

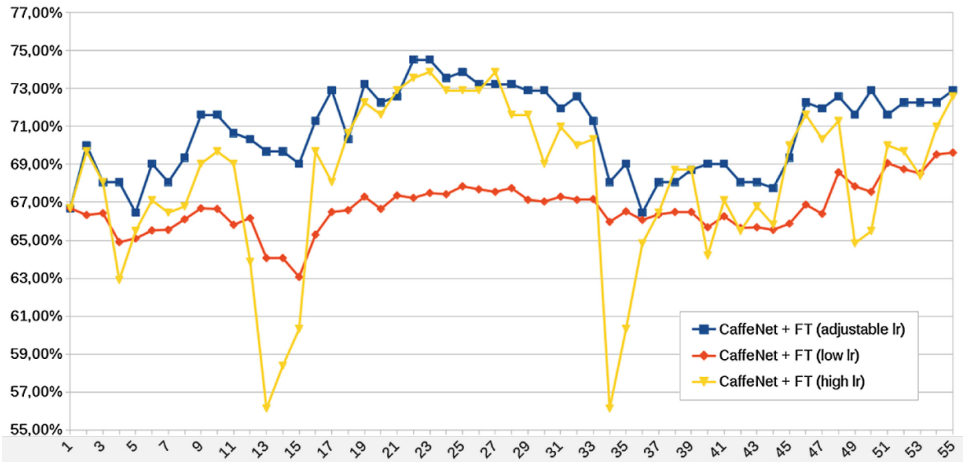


FIGURE 3.4: Accuracy results in BigBrother dataset for different learning rates [37].

Graph 3.4 compares three different strategies.

1. *Low learning rate*, with this value of *lr* there are no peaks and the incremental learning trend is quite smooth. It has maintained an average growth.

2. *High learning rate*, with this value of *lr* there are peaks in the learning trend (especially for batches 13 and 34). In these cases the network forgot what it previously learned.

3. *Adjustable learning rate*, with an adjustable value of *lr* there are some peaks in the variation of learning trend, but not abrupt as the case with high learning rate. This behaviour can be explained by the variable size of incoming batches and the ability for the implemented network to adjust its *lr* dynamically up on this size.

Another result is shown in figure 3.5, where different CNN architectures are evaluated on BigBrother dataset.

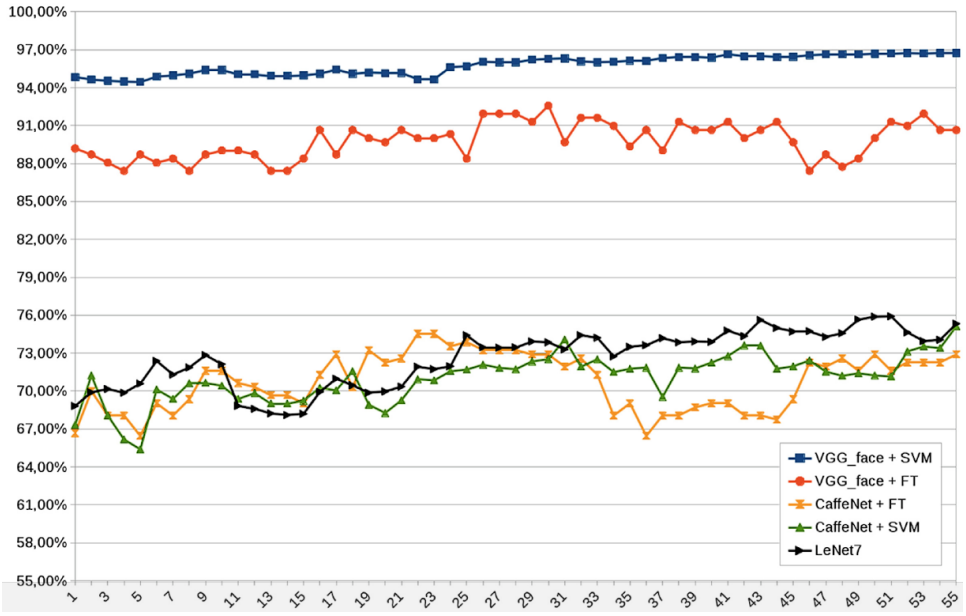


FIGURE 3.5: Accuracy results in BigBrother dataset about all different strategies tested [37].

In this experiment a new architecture is introduced: *VGG_Face* is a very deep architecture than that has been trained directly on a very large dataset of faces. Due to the specific pre-training on faces, VGG-based approaches highly outperformes the other approaches.

3.3.2 Limitations

The results described in the previous paragraph can be considered satisfactory for specific applications, but the related datasets are quite hard to use in order to show a more general applicability.
However, the results are satisfactory at the most. In fact, in both cases the accuracy results are good enough. Also in the iCubWorld28 with *CaffeNet + fine-tuning* the averaged accuracy reaches the 75%.

The main limitations of these datasets are in the quality and the number of frames. More specifically:

- *iCubWorld28*: the main critical points of this dataset are:
  1. the small number of sessions: 4;
  2. the small number of objects: 28;
  3. the limited background variation: only indoor acquisition with controlled lighting;
  4. the maximun resultion of 128x128.
- The *BigBrother* dataset contains grayscale frames and the resolution of 70x70 pixels is almost half of iCubWorld28 dataset. This dataset

furthermore contains only faces and it is not useful for generic object recognition which is the main target of this work, like those addressed in this thesis.

- The *NORB* dataset contains a relevant number of frames and variations, but the images recorded are not natural objects but small untextured toys. Furthermore, there is no natural background.

Due to the above limitation we decided to collect a new dataset that will be described in chapter 4.

Chapter 4

TCD4R

In this chapter we introduce the new dataset and its design, we describe the hardware used and the software application implemented to support the dataset acquisition.

4.1 TCD4R Overview

In this section, as we mentioned before, we introduce an overview of the dataset built from scratch. The dataset name stands for **TCD4R: Temporal Coherent Dataset For Robotics**. It shares some features with *iCubWorld28 dataset* described in the previous chapter.
The purpose of TCD4R is to overcome the limitations presented in the chapter 3 especially in terms of limited number of sessions and objects, and background variation.

4.1.1 Design

TCD4R is an RGB + D dataset (the depth information are not used in this work but can be very useful in future studies). Each color frame is 350x350 pixels and the depth frame is 512x424 pixels. The two frames can be registered by using Microsoft libraries in the following.
The dataset contains relevant lighting and background variations. It consists of 11 sessions: 8 indoor and 3 outdoor. The background, especially in the outside sessions, changes significantly. Each session lasts about 15 seconds with a framerate of 20 fps, resulting in total of 300 frames.
As for *iCubWorld28* during acquisition, each object is slowly moved and rotated in front of the camera which is a Microsoft Kinect 2.0.

A specific purpose of this dataset is to increase the number of registered objects. TCD4R contains 50 different objects divided in 10 different classes. An object can be partially occluded by the hand holding in front of the camera; this clearly complicates object recognition and makes the dataset

particularly suited for robotic applications. The objects considered are typical of a domestic environment and their size is almost the same. Here is a list divided by classes.

1. **Adapters**;
2. **Remote controls**;
3. **Light bulbs**;
4. **Highlighters**;
5. **Smartphones**;
6. **Balls**;
7. **Glasses**;
8. **Cans**;
9. **Scissors**;
10. **Cups**.

The total size of TCD4R is 165.000 RGB+D frames (50 objects x 11 sessions x 300 frames per session). In the following picture we can see one frame per each object.

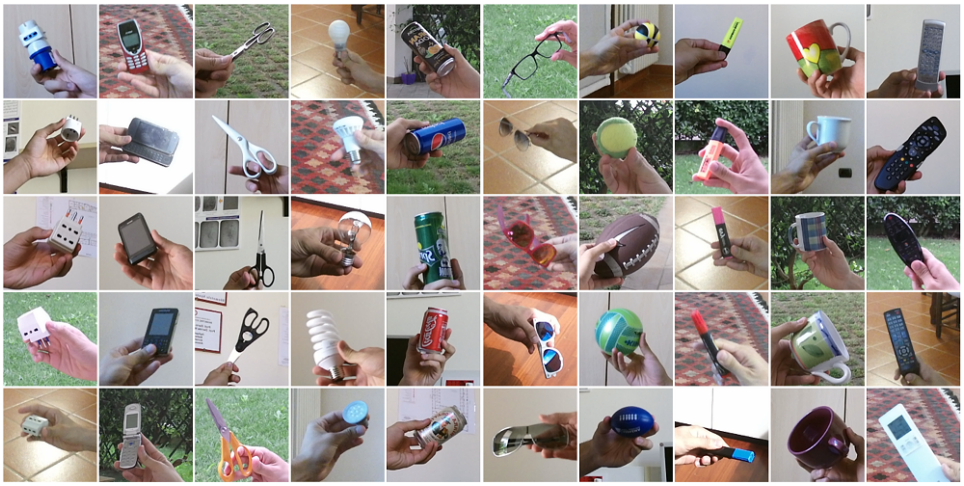


FIGURE 4.1: The 50 different objects divided in 10 different classes (in columns). We can appreciate in this figure the large lighting and background changes characterizing different sessions.

4.2 Hardware and Software

In this section we describe the hardware and software that had to be used for data collection.

4.2.1 Hardware

The hardware used consists of a Microsoft Kinect 2.0 and a laptop running Microsoft Windows 8.1 64 bit. This operating system is recommended to use the APIs to develop an application for Microsoft Kinect 2.0.
A USB 3.0 port is necessary to connect the camera to the PC. The laptop features are:

- Intel Core i7-3632QM CPU with a 2.20 GHz frequency.
- 8 GB of primary memory and a 1 TB of secondary storage.
- AMD Radeon HD 7700M Series video card.

This configuration does not permit the use of the Caffe framework in GPU mode because CUDA (the parallel computing platform and programming model invented by Nvidia) is required. The minimum requirement to use the GPU mode is therefore an Nvidia video card.
In the following paragraph we describe the Microsoft Kinect 2.0 and its SDK called *Kinect for Windows*.

4.2.1.1 Microsoft Kinect and Kinect for Windows SDK

Kinect is a line of motion sensing input devices by Microsoft for Xbox 360 and Xbox One video game consoles and Windows PCs. Based on a webcam-style add-on peripheral, it enables users to control and interact with their console/computer without the need of a game controller, through a natural user interface using gestures and spoken commands.
The first-generation Kinect was introduced in November 2010 in an attempt to broaden Xbox 360's audience beyond its typical gamer base.
Kinect sensor is a horizontal bar connected to a small base with a motorized pivot and is designed to be positioned lengthwise above or below the video display. The device features an RGB camera, depth sensor and multi-array microphone running proprietary software, which provide full-body 3D motion capture, facial recognition and voice recognition capabilities.
The depth sensor consists of an infrared laser projector combined with a monochrome CMOS sensor, which captures video data in 3D under any ambient light conditions. The sensing range of the depth sensor is adjustable, and Kinect software is capable of automatically calibrating the sensor based on gameplay and the player's physical environment, accommodating for the presence of furniture or other obstacles.

Described by Microsoft personnel as the primary innovation of Kinect, the software technology enables advanced gesture recognition, facial recognition and voice recognition. According to information supplied to retailers, Kinect is capable of simultaneously tracking up to six people, including two active players for motion analysis with a feature extraction of 20 joints per player.
A second generation of Kinect was released on 2013. The main hardware difference between the two versions of Microsoft Kinect lies in the color and depth cameras resolution. The first version has a resolution of 640x480@30 fps for the color camera and 320x240 for the depth camera. The second version instead has a resolution of 1920x1080@30 fps for the color camera and 512x424 for the depth camera. Both versions have got a maximum and a minimum depth distance. The maximum is about 4.5 for both versions, while the minimum is 40 cm for the first version and 50 cm for the second. To acquire depth information the first version uses infrared structured light and a conventional optical sensor, while the second version uses a more accurate Time of Flight optical sensor with an infrared illuminator. Figure 4.2 describes the layout of color camera, depth sensors and microphone multi-array.
In 2011 Microsoft released the Kinect software development kit. This SDK was meant to allow developers to write Kinect apps in C++/CLI, C#, or Visual Basic .NET. The Kinect for Windows SDK 2.0 was released in July 2014. It is intended for those developing Kinect-enabled software for Windows 8 and Windows 8.1, and is optimized to operate at a closer range than the Xbox One version.

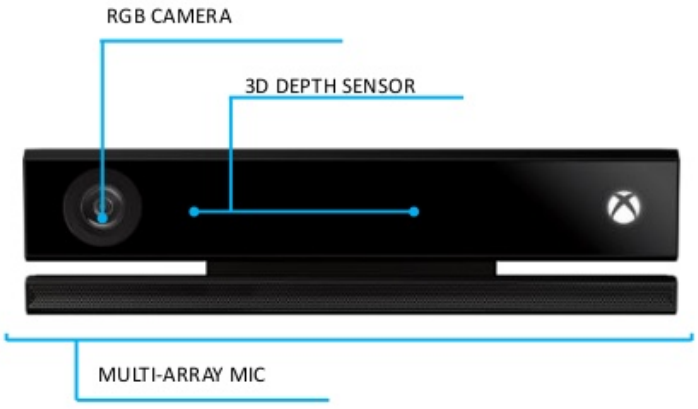


FIGURE 4.2: The layout of color camera, depth sensors and microphone in Microsoft Kinect 2.0.

The Kinect for Windows SDK 2.0 needs Visual Studio IDE where a specific extension need to be installed. The SDK provides the tools and APIs, both native and managed, that are needed to develop Kinect-enabled applications for Microsoft Windows. Developing Kinect-enabled applications is essentially the same as developing other Windows applications, except that the Kinect SDK provides support for the features of the Kinect, including color images, depth images, audio input, and skeletal data. The SDK includes [27]:

- Drivers and technical documentation for implementing Kinect-enabled applications using a Kinect for Windows sensor.
- Samples that demonstrate good practices for using a Kinect sensor.
- Example code that demonstrates the most useful tasks.

The SDK also includes the reference APIs and documentation for programming in managed and unmanaged code. The APIs deliver multiple media streams with minimal software latency across various video, CPU, and device variables.
The Kinect for Windows SDK 2.0 provides three different API sets that can be used to create Kinect-enabled applications. A set of Windows Runtime APIs is provided to support the development of Windows Store applications. A set of .NET APIs is provided to support the development of WPF (Windows Presentation Foundation, a graphical subsystem for rendering user interfaces in Windows-based applications) applications. A set of native APIs is provided to support applications that require the performance advantages of native code. Through the APIs, we can develop different features such as:

- **Lean tracking**: incorporate player lean, how much their body is leaning from vertical, into their experience. For example, it is used when a player leans around an obstacle to see something.
- **Body tracking**: track up to six player's bodies for each frame.
- **Face tracking**: recognize a face in a frame by a training face recognition neural network that has already been put in place.

The APIs also include the **Coordinate Mapper** allows to perform two important tasks: project and unproject depth from 2D image space to 3D camera space and map between locations on the depth image and their corresponding locations on the color image.
The second task was used in this work to prove the feasibility of correctly aligning color and depth data (useful for future extensions).
The term *Depth space*is used to denote the domain of 2D locations in the

depth images. The location $x = 0$, $y = 0$ corresponds to the top left corner of the image and $x = 511$, $y = 423$ (width-1, height-1) is the bottom right corner of the image. In some cases, a z value is needed to map out of depth space. For these cases, simply sample the depth image at the row/column in question, use that value (which is depth in millimetres) directly as z.
The color sensor on Kinect is located at a little distance from the depth sensor. As a result, the depth sensor and the color sensor see a slightly different view of the world. If we want to find the color that corresponds to given pixel on the depth image, we will have to convert its position to color space. To color space describes a 2D point on the color image, just like depth space does for the depth image. A position in color space is a row/column location of a pixel on the image, where $x = 0$, $y = 0$ is the pixel at the top left of the color image, and $x = 1919$, $y = 1079$ (width-1, height-1) corresponds to the bottom right. In figure 4.3 we show an example of depth frame and corresponding color frame.

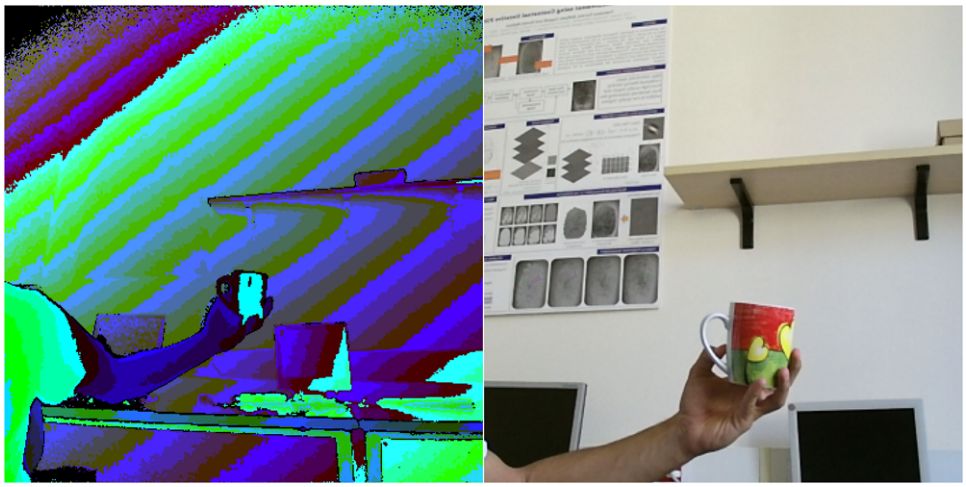


FIGURE 4.3: The same frame acquired from depth sensor, at the left side, and color camera, at the right side. Depth information are visualized with false colors.

4.2.2 Software

In this paragraph we describe the software developed and used to build TCD4R.

4.2.2.1 Data capture

In this section we present the data capture step. The application developed is called *Video Recording* and its purpose is to capture RGB+D frames at al least 20 fps. Images are then saved in png format.
This application is developed through Visual Studio 2015 IDE.
Here we present the *ScreenshotButton_Click* function that save a color frame in a png file.

```
private void ScreenshotButton_Click(object s, Event e)
{
  if (this.colorBitmap != null)
  {
    //png bitmap encoder who will save a .png file
    BitmapEncoder encoder = new PngBitmapEncoder();
    //create frame from bitmap and add to encoder
    encoder.Frames.Add(BitmapFrame.Create(colorBitmap));
    string time = System.DateTime.ToString();
    string myPhotos = Environment.GetFolderPath
      (Environment.SpecialFolder.MyPictures);
    string path = Path.Combine
      (myPhotos, "KinectScreenshot-Color-" + time + ".png");
    // write the new file to disk
    try
    {
      using (FileStream fs =
        new FileStream(path, FileMode.Create))
      { encoder.Save(fs); }
      this.StatusText = string.Format(Properties.Resources.
        SavedScreenshotStatusTextFormat, path);
    }
    catch (IOException)
    {
      this.StatusText = string.Format(Properties.Resources.
        FailedScreenshotStatusTextFormat, path);
    }
  }
}
```

The *colorBitmap* is the color frame coming from Kinect, it is saved by the *BitmapEncoder* which passes the path of the picture that will be created.

As described in the previous paragraph the coordinate mapper allow to register color and depth information. The following code describes how to map color frame to depth frame; in the loop the code associates each color point to the equivalent depth point, as we can see from the instruction *float colorMappedToDepthX = colorMappedToDepthPointsPointer[colorIndex].X;*.

```
//Treat the color data as 4-byte pixels
uint* bitmapPixelsPointer = (uint*)this.bitmap.BackBuffer;

// Loop over each row and column of the color image
// Zero out any pixels that don't correspond to a body index
for (int colorIndex = 0;
  colorIndex < colorMappedToDepthPointCount; ++colorIndex)
{
  float colorMappedToDepthX =
    colorMappedToDepthPointsPointer[colorIndex].X;
  float colorMappedToDepthY =
    colorMappedToDepthPointsPointer[colorIndex].Y;

  // The sentinel value is -inf, -inf, meaning
  // that no depth pixel corresponds to this color pixel.
  if (!float.IsNegativeInfinity(colorMappedToDepthX) &&
    !float.IsNegativeInfinity(colorMappedToDepthY))
  {
    // Make sure the depth pixel maps to color space
    int depthX = (int)(colorMappedToDepthX + 0.5f);
    int depthY = (int)(colorMappedToDepthY + 0.5f);

     // If the point is not valid, there is no index there.
     if ((depthX >= 0) && (depthX < depthWidth) &&
       (depthY >= 0) && (depthY < depthHeight))
     {
       int depthIndex = (depthY * depthWidth) + depthX;

       // If we are tracking a body for the current pixel,
       // do not zero out the pixel
       if (bodyIndexDataPointer[depthIndex] != 0xff)
       { continue; }
     }
  }
  bitmapPixelsPointer[colorIndex] = 0;
}
```

Figure 4.4 shows the user interface of the Video Recording application during a capture session.
In the central box there is the view from the color camera of the Microsoft Kinect 2.0. The full frame displayed has a 1920x1080 pixels resolution and

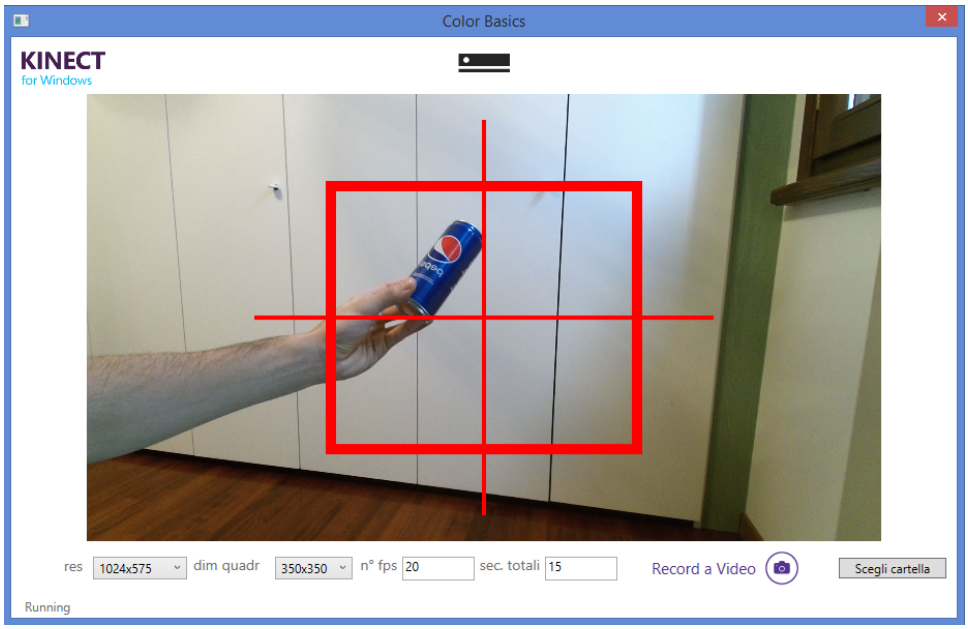


FIGURE 4.4: A screenshot of the Video Recording application during a recording session.

its output goes to the *colorBuffer* introducted in the first code example.
The user can perform the following settings:

- **Frame resolution**: this parameter denotes the acquisition resoltion. An aspect ratio of 16:9 is always used, but the user can reduce the resolution from 1920x1080 to 320x180.
  The default for TCD4R frame resolution is 1024x768 pixels.

- **Crop window size**: this parameter denotes the size of the saved frames (a subwindow cropped from the center of the acquired full frame). An aspect ratio of 1:1 is always used. The default for TCD4R pointer size is 350x350 pixels.

- **Fps**: number of frame per second that the application captures from the *colorBuffer* and *depthBuffer*. The default for TCD4R is 20 fps.

- **Recording length**. The default for TCD4R is 15 seconds.

The red pointer in figure 4.4 denotes the position of the crop windows with respect to the full frame, and is useful for the user during image acquisition to keep objects within the saved image portion. This pointer shows where in the *colorBuffer* where the final part of the current frame is captured, making it possible for the user to keep the object on which to build the capture inside the marked pane. The size of this red pointed changes according to the current resolution settings.
To speed up acquisition the cropping is performed at the end of data capture just before image storage.

When the user clicks on the *Record a video* button, the application runs the following loop and stores *fps∗recording length* different frames in the *chosen directory*.

```
//total number of frames fps*rec_length
int recordlength =
  Int32.Parse(textboxSec.Text)*Int32.Parse(textboxFps.Text);

//setting delay 1 sec / fps
int delay = 1000 / Int32.Parse(textboxFps.Text);
for (int i = 0; i < recordlength; ++i)
{
  Console.WriteLine(i);

  WriteableBitmap display = this.colorBitmap.Clone();

  //async call of captureFrame function
  AsyncMethodCaller caller =
    new AsyncMethodCaller(captureFrame);
  caller.BeginInvoke(i, display,
    this.rectangleColor, null,null);

  //wait delay tick
  await Task.Delay((int)delay);
}
```

Using .NET C# language, asynchronous methods had to be used. Delay instruction requires an *await*, an operator applied to a task in a method to suspend the execution of the method until the awaited task completes. This kind of asynchronous programming is typical of .NET C# in order to build multi-thread application.

At the end of recording there will be a object recording; there will be several frames, both color and depth stored in the *chosen directory* named like *ColorNumberOfFrame.png* or *DepthNumberOfFrame.png* where *NumberOfFrame* is the number of frames stored as 000 regular expression.

4.2.2.2 Data preprocessing

We have already mentioned the first data processing operation related to resizing and cropping.

The second step concerns the proper file organization: all frames are divided by recording sections (11 directories) and for each section 50 subdirectories (one for each object) are populated. Each object directory initially is named with the object name, but to simplify following processes they will be then renamed with an incremental number from 0 to 49.

Another processing step that we implemented to simplify the CNN training is a further crop (to a final size of 128x128 pixel). As we mentioned in the chapter 2, there are several standards size that could be used with popular CNN architectures. The aim of this cropping is to isolate as much as possible the object of interest from the background and place the cropping box around it. Since object are moving in front of the camera we cannot use a fixed crop position but we must track the object.
To implement this step we used an existing application performing object tracking by: frame difference, thresholding, and connected component labelling. The cropping box is then placed around the largest connected component. In figure 4.5 we show two consecutive frames and their binarized difference denoting motion.

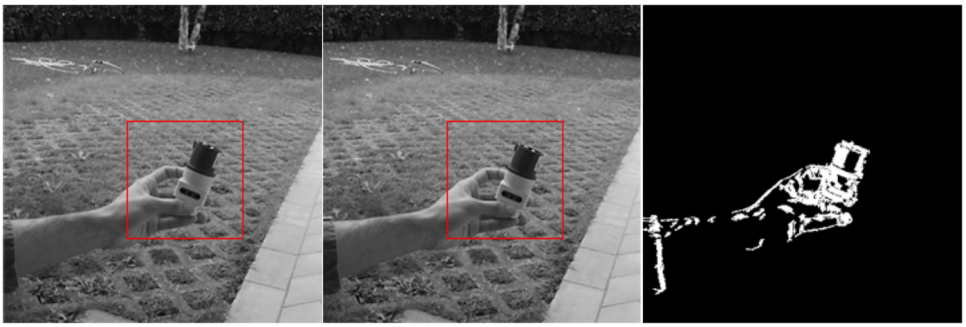


FIGURE 4.5: Two subsequent frames with the 128x128 pixels crop box and their binarized difference.

After this step, we noted that in some cases the 128x128 crop did not contain the object of interest (or only a small amount of images). See for example figure 4.6 where the plug adapter is not present in the first frame and only partially present in the second one. In this specific outdoor session the leaves in the background moved by the wind probably fooled the motion detector. This behaviour means that there are some other relevant variations between two following frames.
We need to correct these errors because they could bring some unnecessary complexity to the recognition task by CNN. The correction was performed

by automatic detection of anomalous frames (characterized by abrupt crop position changes) then manually corrected. After this data pre-processing the TCD4R is ready for a Deep Learning application.

FIGURE 4.6: Three subsequent frames with a cropping error.

Chapter 5

CNN Training

In the first section of this chapter we introduce the Deep Leaning framework used, *Caffe*; in the second section we present the data partitioning (in the three typical subsets: *train*, *validation* and *test*), and in the third part there is a description of the network used. In the last section we report the result of our experiments.

5.1 Caffe

Caffe is a deep learning framework developed by the Berkeley Vision and Learning Center (BVLC) and by community contributors. Caffe has some interesting features like [42]:

- **Expressive architecture**: it encourages application and innovation. Models and optimization are defined by configuration without hard-coding. Switch between CPU and GPU by setting a single flag to train on a GPU machine.
- **Extensible code**: the framework tracks the state-of-the-art in code and models.
- **Speed**: Caffe can process over 60M images per day with a single Nvidia K40 GPU (with the ILSVRC2012 winning network). It's 1 ms/image for inference and 4 ms/image for learning.
- **Community**: there is a rich community ranging from researchers to professionals who use Caffe.

For this thesis we tried to install Caffe on two different operating systems: Microsoft Windows 8.1 and Ubuntu 16.04, however since the Windows version is not stable we decided to continue working under Linux OS. The Linux server used for the experiments has two CPU (Intel Xeon) and four GPU (Nvidia Tesla C2075).
In the following subsection we provide some details about CNN training in Caffe.

5.1.1 Training walkthrough

The first step is to configure the dataset and split the data in three subsets:

- **Train**: this subset of images contains all the sessions that will be used to train the CNN during the *train* step.
- **Valid**:this subset of images contains all the sessions that will be used to validate the CNN during the *train* step and tune hyperparameters.
- **Test**: this subset of images contains all the sessions that will be used to test the CNN during the *test* step. It is computed after the *train* step.

The network architecture and training are defined in files: *solver.prototxt* and *network.prototxt*. The first file orchestrates the model optimization by coordinating the network's forward inference and gradient backward to update parameters with the objective of reducing the loss. The second file contains the CNN definition (layer by layer).

For each data subset (train, valid, test) it is necessary to define a list of images (and corresponding classes) in the following format:

```
/path/to/folder/image1.png 0
/path/to/folder/image2.png 3
/path/to/folder/image3.png 1
/path/to/folder/image4.png 2
/path/to/folder/image5.png 1
...
...
/path/to/folder/imageN.png N-1
```

Another step that can be useful for the CNN training is normalizing data by subtracting the image mean (over the training set). Caffe provides a way to compute the image mean directly. We need to generate the lmdb database for our training images so that Caffe can use it to generate the mean image. The following command to generate (*train_lmdb*) is presented below:

```
$ GLOG_logtostderr=1 convert_imageset --resize_height=32
    --resize_width=32 --shuffle / /path/to/train.txt
    /path/to/train_lmdb
```

where the setting *resize* is used to resize the image loaded and used in lmdb.

The *shuffle* option is used, instead, to shuffle the frames included in lmdb. It is important that the images are shuffled in lmdb. We want images from random classes to appear in a sequence.
After the lmdb creation we can compute the mean image by the following command.

```
$ compute_image_mean /path/to/train_lmdb
    /path/to/mean_image.binaryproto
```

Now we are ready to run the training command and wait until the end of iterations or exit before if the error is lower than the threshold defined in *solver.prototxt*.

```
$ caffe train --solver /full/path/to/solver.prototxt
```

The last step is testing the network resulting from the training. The following command scores models by running them in the test phase and reports the net output as its score. The network architecture must be properly defined to output an accuracy measure or loss as its output. The per-batch score is reported and then the grand average is reported last.

```
caffe test -model path/to/network.prototxt
    -weights path/to/weights.caffemodel -iterations 100
```

where the parameter *weights* represents the weights learned during training and iteration the number of batch considered during the test.

5.2 Data partitioning and experiment introduction

First of all, we need to define the notation which will be used here on to introduce the data partitioning and experiments.

- n_s is the number of recorded sessions.
- n_w is the number of considered classes.
- The dataset D will be splitted in training, validation and test set D_{train}, D_{valid} and D_{test} where $|D_{\text{train}}| + |D_{\text{valid}}| + |D_{\text{test}}| = n_s$
- S_i^w, $i = 1, \ldots, n_s$ is the temporal coherent frame sequence of class w in the session i.
- $v^{(t)}$, $t = 1, \ldots, \text{len}(S_i^w)$ denotes the frame at (discrete) time t in a sequence.
- N a generic classifier (a CNN in our case).
- $N(v^{(t)})$ the classifier output for the frame at time t, or the probability $P(w|v^{(t)}), w = 1, \ldots, n_w$.
- $d(v^{(t)})$ the *desideratum*, the frame label at time t.

Now we define the three different sets D_{train}, D_{valid} and D_{test}. TCD4R has 11 sessions, $n_s = 11$, with 3 outdoor sessions highlighted with $\bar{m}$ where m is the session number. The sets are defined as:

- $D_{\text{train}} = \{S_i^w | w \in \{0, \ldots, n_w\}, i \in \{1, 3, \bar{4}, 6, 7, 8, 9, \bar{10}\}\}$.
- $D_{\text{test}} = \{S_i^w | w \in \{0, \ldots, n_w\}, i \in \{2, 5, \bar{11}\}\}$.
- D_{val} is a *K-fold cross validation* with $K = 4$ from D_{train} set.

We define the *task policy* of an experiment E, $\pi(E)$:

1. Calculate the hyperparamenters and evaluate the validation set accuracy with *K-fold cross validation*;
2. Train the model on D_{train};
3. Evaluate the accuracy on D_{test}.

We define four experiments $E = \{e_{\text{frame}}^{w_{10}}, e_{\text{frame}}^{w_{50}}, e_{\text{seq}}^{w_{10}}, e_{\text{seq}}^{w_{50}}\}$ where $e_{\text{frame}}^{w_n}$ is the classification task with n classes on single frames and with $e_{\text{seq}}^{w_n}$ is the classification task with n classes on full test sequences S_i^w. In TCD4R we can switch from $n = 10$ to $n = 50$ classes. In particular, when $n = 50$ each object is a class, while when $n = 10$ the 5 objects of the same category are considered together as a unique class.
In the experiments $e_{\text{frame}}^{w_n}$ the classification task is much more difficult since we need to decide the object class based on a single frame, while in the $e_{\text{seq}}^{w_n}$ case we can consolidate the decision on more frames of the same objects. Given a temporal window size c in sequence classification we can fuse $P(w|v^{(t)}), P(w|v^{(t-1)}), \ldots, P(w|v^{(t-c)})$, based for example on the simple sum fusion criterion.

5.3 Network Setup

In this paragraph we introduce the network architecture and its setup. In particular, we describe the two files: *solver.prototxt* and *network.prototxt*.
The network used is the CIFAR-10, built by Alex Krizhevsky. The motivation to use a quite simple network such as CIFAR-10 is due to the fact that training from scratch a more complex network would require much more examples than those available in TCD4R. It works on 32x32 RGB images. Note that to use this network on TCD4R input images 128x128 need to be heavily down sampled to 32x32. CIFAR-10 is a Caffe built-in network, but as we will see in this paragraph in order to run it over our dataset some changes are necessary.

In the following we describe the *solver.prototxt*.

```
net: "/home/riccardo/models/cifar10/network.prototxt"

test_iter: 240
test_interval: 1000
base_lr: 0.01
momentum: 0.9
weight_decay: 0.004
lr_policy: "fixed"

display: 200

max_iter: 60000

snapshot: 10000
snapshot_format: HDF5
snapshot_prefix: "/home/riccardo/models/cifar10/cifar10"

solver_mode: GPU
```

The solver contains the settings of the training task, where:

- The *train-test* network protocol buffer definition by the *net* parameter.
- *test_iter* specifies how many forward passes the test should carry out. In the case of CIFAR-10, we have test batch size 240 and 250 test iterations, covering the full 60.000 testing images.
- *test_interval* the framework performs one evaluation (on the specified validation set) every 1.000 training iterations.
- The *base learning rate, momentum, weight decay* and *learning rate policy* of the network.
- The *display rate*, in this network every 200 iterations.
- The maximum number of iterations, in this network 60.000.
- The snapshot intermediate result settings.
- The solver mode: target hardware used by the framework: CPU or GPU (if available).

The file *network.prototxt* describes the CNN architecture layer by layer. In this file a *top-down* layer disposition is described, where you could find all the previously described CNN layers. In the following, starting from *data layer*, we will introduce all types of layers inserted in CIFAR-10 network.

```
name: "CIFAR10_full"
layer {
  name: "cifar"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    mean_file: "models/cifar10/mean_image_train.binaryproto"
  }
  data_param {
    source: "/home/riccardo/train32_lmdb"
    batch_size: 250
    backend: LMDB
  }
}
```

In this part, the *data layer* is described where we can find the *phase* (train or test), the location of the precalculated mean file, the input LMDB and the *batch size*. All Caffe layers have a common layer description with the *name, type, top* and *bottom* parameters which identify the position of the current layer inside the network.

Now we show and example of definition of a convolutional layer:

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1
  convolution_param {
    num_output: 32
    pad: 2
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "gaussian"
      std: 0.0001
    }
    bias_filler {
      type: "constant"
    }
  }
}
```

The main parameters are: *output number, padding, kernel size* and *stride* are introduced.

The following layers are *Pooling* and *ReLU* whose main parameters are: *max pooling* definition, *kernel size* and *stride*.

```
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 3
    stride: 2
  }
}
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "pool1"
  top: "pool1"
}
```

The last layer is a fully connected layer whose Caffe type is *InnerProduct*. Here *number of outputs* denotes the number of classes n of the problem, 10 in the example.

```
layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "pool3"
  top: "ip1"
  inner_product_param {
    num_output: 10
    ...
  }
}
```

5.4 Experimental Phase

In this section we report the experimental results. We will start from the tests made right after TCD4R was built and we continue with the study carried out in order to improve the benchmark and finally we will introduce the experiments conducted with respect to Incremental Learning.

Our error analysis also include the confusion matrix. This is a square matrix whose size corresponds to the number of classes. On the column we have the predicted classed and on the row the real classes. The name stems from the fact that it makes it easy to see if the system is confusing two classes.

5.4.1 Exploratory analysis

The first exploratory tests are related to $e^{w_{10}}_{\text{frame}}$ and $e^{w_{50}}_{\text{frame}}$. Before testing the entire TCD4R on the previously described tasks, we build a trial dataset with only five objects. The purpose of this step is to set up the network parameters in order to increase the accuracy to the highest value.

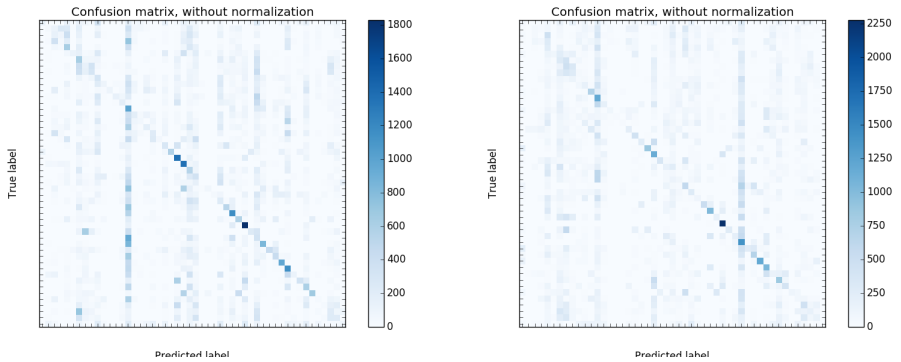
After the network set up step we make the study on TCD4R sessions. We executed a Caffe training on a *4-fold* validation of the train test. We choose the session that seemed a good compromise in terms of recognition difficulty within the train set. This difference between the sessions are highlighted by different backgrounds (e.g. outdoor session or indoor session), different lighting or objects occlusion. We make this training test with three indoor sessions and one outdoor session for each set (train and valid) with 10 and 50 different classes of objects. In the following table 5.1 we report the results reached after 30.000 iterations on the two experiments.

Classes	Accuracy Reached
$e^{w_{10}}_{\text{frame}}$	30%
$e^{w_{50}}_{\text{frame}}$	17%

TABLE 5.1: Accuracy after 30.000 iterations on the two classification problems.

After this first test we decided to study the contribution that each single object (class) gives to the total error. To this purpose we use confusion matrices.

In figure 5.1 we compare two different confusion matrices reached from two different training tests. The figure 5.1a represents the standard case



(A) 50 different classes of objects (B) Object 14 removed

FIGURE 5.1: A comparison between the two confusion matrices in case of two test on 50 an 49 object respectively.

with 50 objects, while in figure 5.1b we plot a confusion matrix obtained in a test with only 49 objects.In the second case we removed object 14 which, as we can see from 5.1a, is the object with the highest number of recognition errors. A recognition error in a confusion matrix is a coloured point not in the matrix main diagonal. Darker is the coloured point more the network recognize that *true label* with a *predicted label*.
The result shows in the figure 5.1b is a confusion matrix with another column which contains an high number of false positive recognitions, object 35. The accuracy does not increase (17%) because the errors (e.g. objects occlusion and different background) do not depend from an object but to a single recorded session with a few percentages points difference.

FIGURE 5.2: Some frame examples of the deleted session.

Another experiment we tried was removing from the train set a session that appered to be particularly complex. We made this evaluation looking to the contrast between the main object color and the background color. In figure 5.2 we show an example of some recorded frames from the deleted session. However, also in this case the resulting accuracy did not vary significantly.

Then a more systematical analysis has been perfomed by varing the sessions included in validation set and test set.

Classes	Accuracy Reached	Test Notes
2-5-11	19.2%	Standard test set
2-8-9	24.38%	Apparently the easiest lighting and backgrounds
2-10-11	18.46%	Two outdoor sessions (10 and 11)
2-6-11	19.85%	Test on the apparently most difficult session (6)
6-8-11	18.48%	Test on session 8
6-8-9	19.34%	Cross test between sessions 6, 8 and 9

TABLE 5.2: Obtained accuracies by changing test set with valid and test set.

The results shown in table 5.2 demonstrate that the accuracy depends on the session configuration. The session complexity, as expected, is influenced by environment (indoor sessions are simpler), background uniformity and lighting. The seven sessions used as test ordered by increasing complexity are: 2, 8, 9, 6, 11 (outdoor), 5 and 10 (outdoor). In the following figure we show them in the above order.

FIGURE 5.3: The seven sessions order by complexity, increasing from left to right: 2, 8, 9, 6, 11, 5 and 10. For each session we only show one object.

Another test made before the experiments run is the developing of a *ad-hoc viewer*, developed in .NET C#, which displays for each class inside the session tested, we used the easiest sessions possible (2, 8 and 9) calculated in the previous test, each frame ordered ascending by score on the true class. The first frames will be the more incorrect compared to the true class. In this test we will study the characteristics of the most erroneous frames.

The first step is to calculate the recognition percentage on test set. In the following table we can see the worst ten classes in term of recognition rate.

Class	Recognition Percentage
Adapter2	0%
Adapter3	0%
Bulb5	0.5%
Adapter5	0.8%
Adapter1	1.6%
Smartphone3	1.8%
Remote Control3	2.8%
Can1	3.17%
Scissors1	3.4%
Glasses2	4.6%

TABLE 5.3: The ten worst ordered ascending by recognition percentage classes.

The following figure shows an example image of the object belonging to each of the ten worst classes to be recognized.

FIGURE 5.4: The ten ordered form top-left to bottomright worst classes.

When we will introduce the experiments result some of these miss-classifications will be explained by the confusion matrices of the frame independent task with 50 different classes of objects.

After this step we implement the previously described *ad-hoc viewer* and with a *csv* file for each class we can pass to this software the probabilities calculated in the previous step. When we analyze a class we can deduce that there are not difference in object occlusion or not show part of an object. This behaviour is a positive feature of TCD4R because all classes have been registered through the same movements.

The *ad-hoc viewer* shows some particular features that are listed in the following.

- Despite the session 2 appears to be the simples one, some frames contain a complex and wall poster in the background. This features greatly complicate the recognition of a generalized object.
- Some objects have moving parts such as sidepiece of a glasses or the scissors blades. This feature complicates the recognition of these objects.

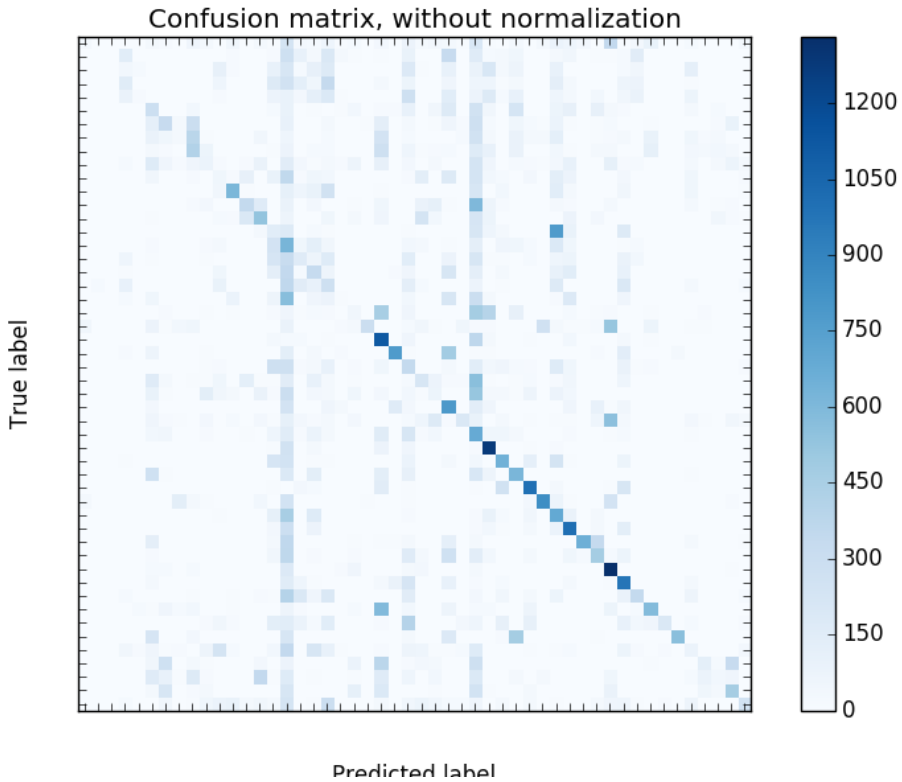


FIGURE 5.5: Confusion matrix of the easiest test set (2, 8 and 9 sessions are included).

- In figure 5.5 shows that objects 15 and 29 are responsible of the highest number of recognition errors. Both objects are white and this color can be confused with shining metallic parts. In pictures of very small resolution such as this test sets, 32x32 pixels, the main object color could confuse the network.
- The confusion matrix 5.5 shows that in some cases the network confuses the objects inside the same main class of objects such as: bulbs (partially hand occluded because its small dimension; the bulb1 and the bulb5, referenced to object 15 and 19, are confused as object 15),

smartphones or remote controllers (they present also the lighting problem described in the previous point; remote controllers are partially confused with objects 15 and 29).

- The confusion matrix 5.5 shows that adapters are confused with bulbs. This behaviour is due to the similar shape of these two classes of objects. It is explained by very low recognition probability previously described.

5.4.2 Detailed training results

The results which are presented in this section are divided in four groups: frame classification, sequence classifications, *8-sequences* train set and, test with pretrained networks.

5.4.2.1 Frame classification

In this paragraph we provide details on the experiments e^{w10}_{frame} and e^{w50}_{frame}. As we have seen, the D_{val} is built with a 4-fold cross validation from the D_{test} set. We run this experiment by implementing a simple Python script which automate the cross-validation on Caffe. We collect all accuracy and loss value from each Caffe training print and we plot the following graphics and confusion matrices.

The first experiment refers to e^{w10}_{frame}. In the figure 5.6 we plot the accuracy over the training iterations for two different learning rates: 0.01 and 0.001.

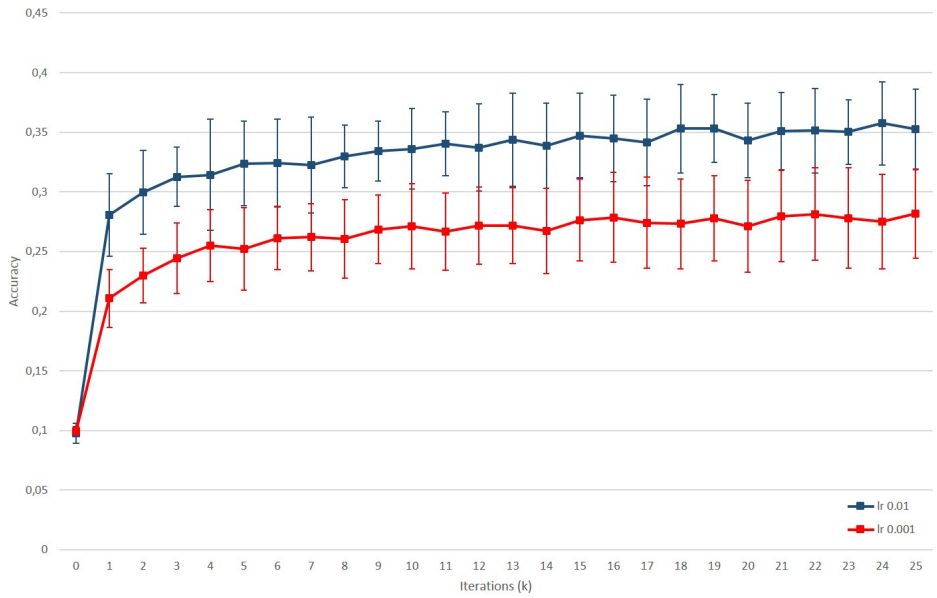


FIGURE 5.6: Frame classification task with 10 classes: accuracy over iterations. The bars denote the error variation over different runs.

As we see from figure 5.6 the $lr = 0.01$ returns an average accuracy of $35\% \pm 4\%$ against the $28\% \pm 3\%$ for $lr = 0.001$. With an higher learning rate the network learns more from the current iteration. For lower value of learning rate the network learns less and this explains the respond to $lr = 0.001$.

Figure 5.7 shows th loss over iterations. For both values of the learning rate loss tends to the asymptote 0.1. The $lr = 0.001$ loss function is slightly more stable than $lr = 0.01$.

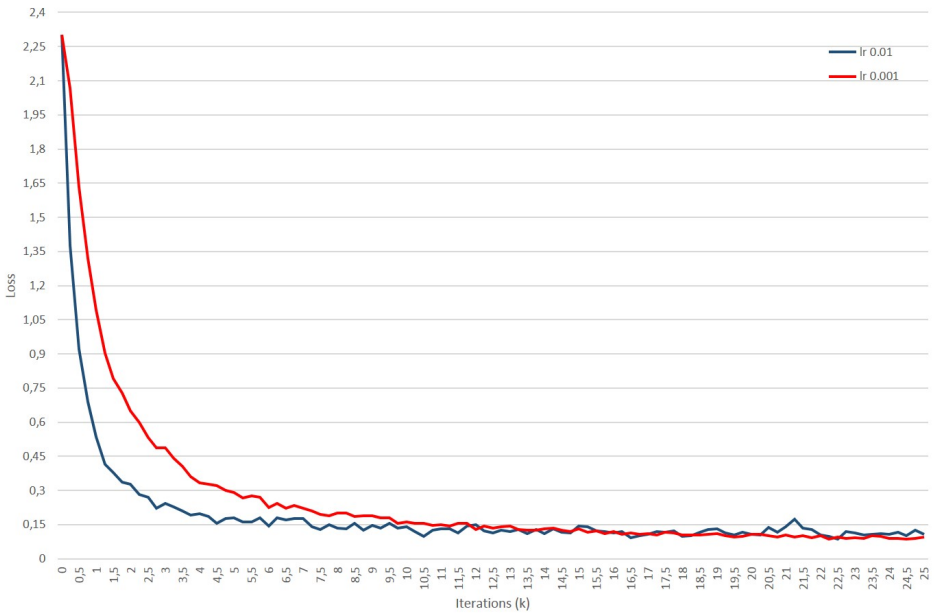
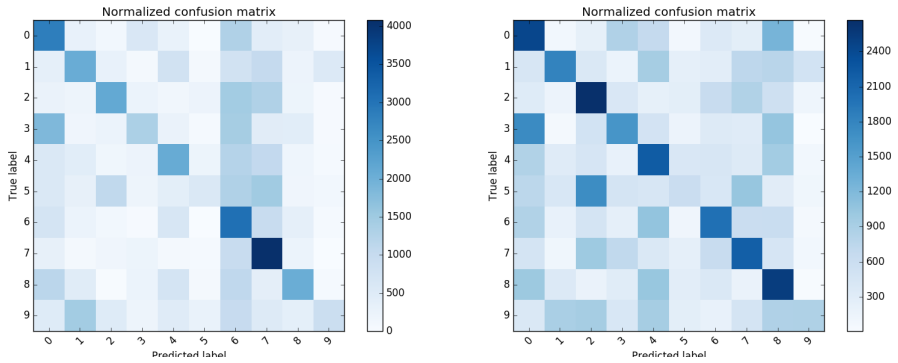


FIGURE 5.7: Frame classification task with 10 classes: loss over iterations.

Figure 5.8 shows the two confusion matrices obtained from this experiment (single run). We print the resulting confusion matrices of a single run in 4-fold cross validation set. The matrix in 5.8a with $lr = 0.01$ is better than matrix in 5.8b with $lr = 0.001$.



(A) $lr = 0.01$ (B) $lr = 0.001$

FIGURE 5.8: A comparison between the two confusion matrices obtained with different learning rates.

With $lr = 0.001$ the network recognizes a huge number of false positive objects in almost all cases between true label and predicted label. In the $lr = 0.01$ case we have a well defined main diagonal (true positive) regard

the false positives in confusion matrix. We have this behaviour because the network does not learn enough from current interation in $lr = 0.001$ case.

The network, in $lr = 0.001$ test, confuses as before the adapters class with bulbs class, this behaviour is due to the same shape. Similarly, it is exchanged the scissors class with glasses class. The network, in $lr = 0.01$ test, does not confuse a class with another but in almost two cases we have a massive recognition of other classes different from true positive recognition such as highlighters and balls cases.

The second experiment refers to e^{w50}_{frame}. In figure 5.9 we report accuracy over iterations for two different learing rate: 0.01 and 0.001.

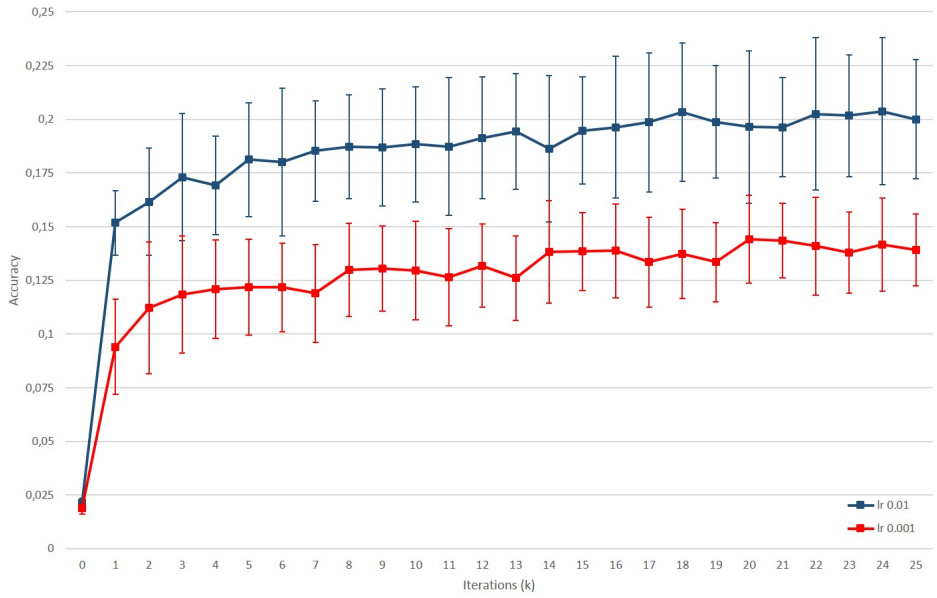


FIGURE 5.9: Frame independent task with 50 classes: accuracy over iterations.

As we saw in the previous experiment with an high value of learning rate the network learns better. With $lr = 0.01$ the accuracy reaches $20\% \pm 5\%$ while with $lr = 0.001$ the accuracy reaches $14\% \pm 2\%$. The accuracy value in both cases is much lower than the previous experiment because in this task we have a higher number of $n_w = 50$. It is more difficult recognize an huge number of classes (with *five times* less frames) than a low number of classes $n_w = 10$ (with 300x4x5 = 6.000 frames per class) even the intra-class invariance is less than the previous experiment because there are five different classes in each category.

The following figure 5.10 shows the loss function over iterations. As

introduced in the previous experiment, we have the same behaviour between the two different values of learning rate. The value of loss function for $lr = 0.001$ tends to 0.2 after 25.000 iterations; this value is lower than $lr = 0.01$ case (0.3) because the network does not learn from current iteration in $lr = 0.01$ case.
In $lr = 0.001$ case the loss function is more precise and quantifies the amount by which the prediction deviates from the actual values.

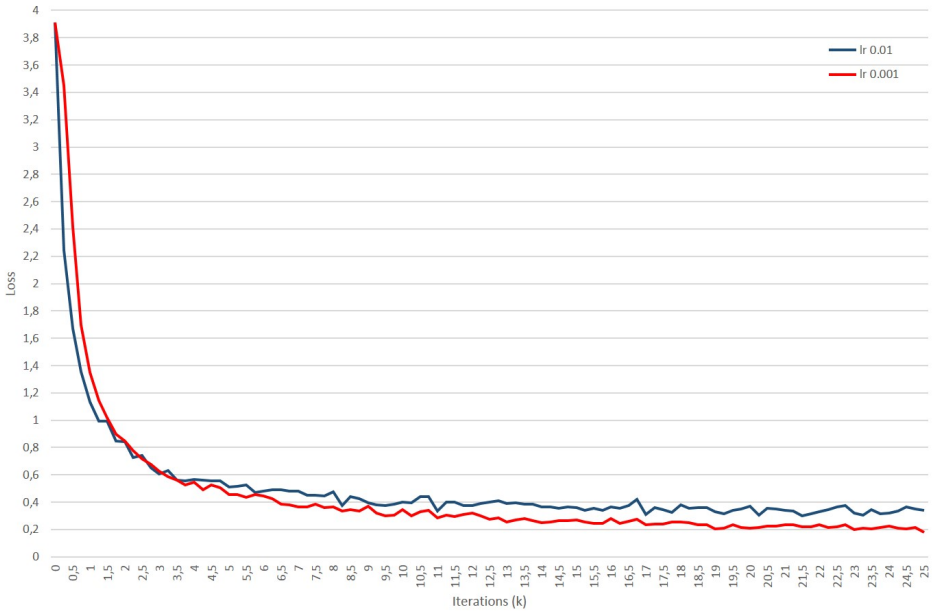
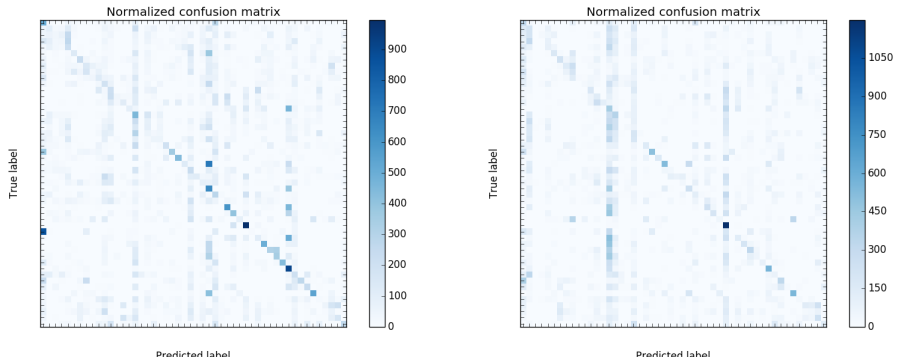


FIGURE 5.10: Frame independent task with 50 classes: loss over iterations.

Finally, figure 5.11 show the confusion matrices for the two learning rates. The figure 5.11a with $lr = 0.01$ is more precise than figure 5.11b with $lr = 0.001$. As described in the previous paragraph in both cases we have some objects that lower the final accuracy, with an high number of false positive. In this experiment we have not enough accuracy to recognize significantly a relevant number of objects. In $lr = 0.01$ the network recognize better some classes of objects such as glasses (objects 25-29), balls (objects 30-34), highlighters (objects 35-39) and cups (objects 40-44). These two different confusion matrices have almost the same features of the previously presented matrices (figure 5.1).

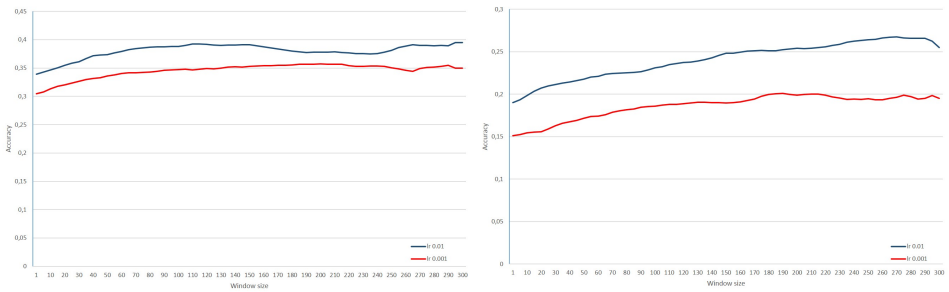


(A) $lr = 0.01$ (B) $lr = 0.001$

FIGURE 5.11: A comparison between the two confusion matrices for different learning rates.

5.4.2.2 Sequence classification

In this paragraph we describe the experiments $e_{\text{seq}}^{w_{10}}$ and $e_{\text{seq}}^{w_{50}}$. We selected a run extracted from *4-fold* cross validation set and we add this sequence experiment over this trained model. Accuracy here is computed by fusing the single frames output probabilities through sum rule. The graphics in figure 5.12 shows the accuracy over window size c; we remember that $c = 20$ is equal to a second of a recording.



(A) 10 classes (B) 50 classes

FIGURE 5.12: A comparison between the accuracy over window size for sequence classification with 10 and 50 classes.

From both graphs we can note that increasing the window size leads to better accuracy. This is an expected result since, fusing information from multiple frames gives to the network the chance to see the same object under different poses.

5.4.2.3 8-sequences test set

The relatively low accuracy obtained in previous experiments could be due to the intrinsic difficulty of the recognition task, or the insufficient number of training examples. We designed a new experiment where we trained the network on the full training set (8 sessions) without extracting any session for validation. We run this experiment over 10 and 50 classes.

In figure 5.13 we show results. We note that accuracy of 50 classes problem is now much better (nearly twice) than in the previous experiments, thus indicating that increasing the number of training samples is very helpful. On the other hand accuracy for 10 classes problem remain almost the same. We argue that discriminating object categories remains a difficult task even if more training samples are available.

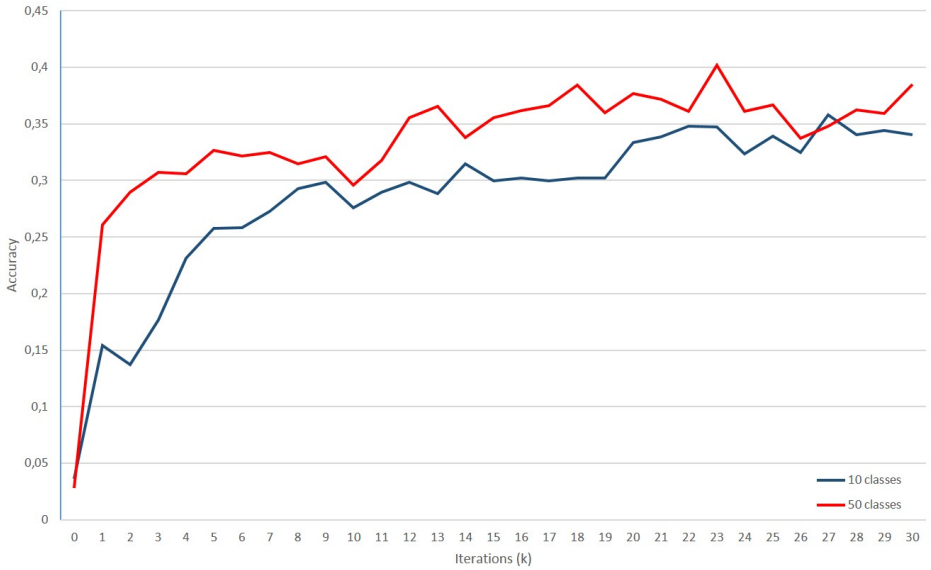


FIGURE 5.13: Frame classification task with 10 and 50 classes: accuracy over iterations.

In figure 5.14 we plot the accuracy of the sequence classification task. Here too accuracy increases with the window size. For large windows size we achieve a remarkable accuracy (over than 50%) for both classification problem. We believe this is a very good result if we consider the small resolution (32x32) of the input images.

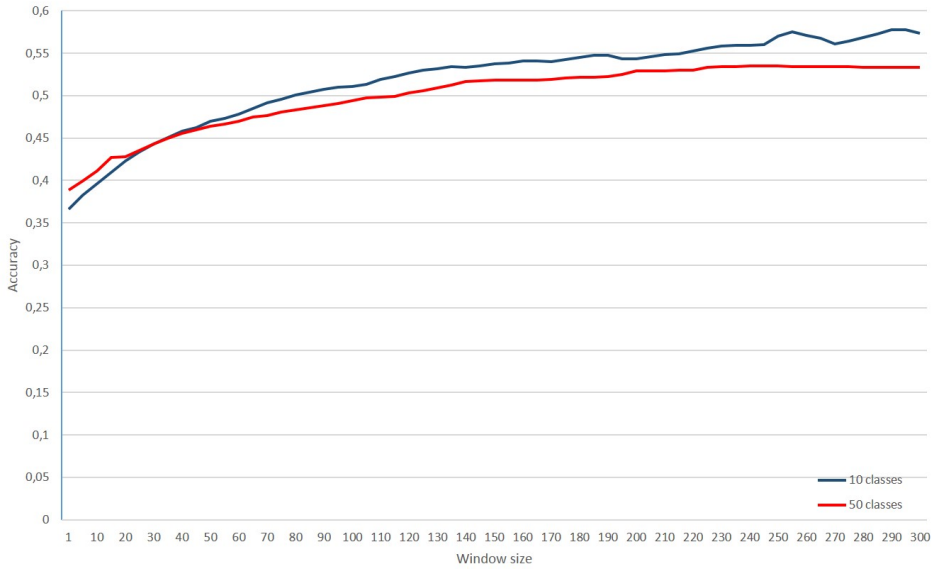


FIGURE 5.14: Sequence classification task with 10 and 50 classes: accuracy over window size.

5.4.2.4 Pretraining and finetuning

All the experiments reported until now have been perfomed with CIFAR-10 network. As explained before training larger CNN from scratch on a relatively small dataset as TCD4R might not be that beneficial.

A different possibility is starting from a pretrained network and tuning it on our dataset. To this purpose we used a CaffeNet (a Caffe implementation of Alexnet described in chapter 2) pretrained on ImageNet (1 milion images, 1000 classes). We tried two different techniques, both described in the chapter 2, *pretraining* and *finetuning*.

We use the standard partition with four sessions in train and valid sets and three sessions in test set. In the table 5.4 we report the archived results.

	10 Classes	50 Classes
CaffeNet Pretrained	67.06%	58.90%
CaffeNet Finetuned	81.06%	71.51%

TABLE 5.4: Task results with pretraining and finetuning of *CaffeNet*.

The accuracy obtained is very high if compared with the CIFAR-10 network trained from scratch. The main reasons are explained in the following. There are some different features from previous experiments. First, the

different image size of the input images used; *CaffeNet* accepts 227x227 pixels resolution frames. This parameter is very important because a 128x128 pixels image, even if it is warped to fit the 227x227 input dimensions), is extremely more detailed than a 32x32 pixels image.
Another difference is the more deeper network with five convolutional layer, in *CaffeNet*, against the three convolutional layer in *LeNet*. The last remark is the different techniques used; in the pretrained case we pass to the original *CaffeNet* our TCD4R, in the finetuned case we made a finetuning with our recorded session over a network already able to recognize objects and not from scratch as in the previous experiments.
It is therefore evident that to manage complex invariants of TCD4R such as different lighting in the same session or different backgrounds a deeper pretrained network is extremely useful.

Chapter 6

Conclusions and Future Works

In this section we present some conclusions about the entire thesis work described in the previous chapters and propose some possible improvements.

6.1 Conclusions

The main contribution of this work is the collection of a new dataset (TCD4R) which allows to overcome the limitation of existing datasets for incremental object recognition studies. TCD4R main features are:

- 50 different objects;
- 10 categories;
- 11 acquisition sessions;
- both indoor and outdoor sessions;
- complex lighting and background;
- very well suited for robot vision applications.

The second part of the thesis focused on training and testing CNN on the new dataset to provide baseline performance.
In chapter 5 we reported several experiments and we observed interesting behaviours.
As expected:

- A simple network as CIFAR-10 trained from scratch on this difficult dataset performs poorly unless a sufficient number of sample are provided in the training phase.
- Classifying sequences instead of single frames is also very helpful to improve accuracy.
- Finetuning a pretrained deeper network such as CaffeNet leads to a very relevant accuracy improvement.

The detailed analysis included in chapter 5 allowed us to understand the main difficulties intrinsic in the new dataset; this analysis could be very useful in the future to search for specific solutions.

In our opinion this dataset, as we already mentioned, is a good starting point for other experiments because it is not too difficult nor too easy, as proof in the previous chapter 5, to learn it.

6.2 Future Works

In this paragraph we provide some ideas for future improvements.

- **TCD4R improvements.** This dataset contains five objects per category and eleven sessions. Further expanding the number of objects and sessions would be highly desiderable.
- **Deep Learning algorithms improvement.** In this work we tested only some CNN architectures and we did not use depth information available in TCD4R. More sophistcated architectures could be used to improve accuracy and fuse RGB with depth information.
- **Incremental Learing experiments.** TCD4R dataset was built with some specific features such as the time coherent recorded sessions. However in this first study we only partially addressed incremental learning scenarios. An interesting future study would be training CNN incrementally over the eleven TCD4R sessions. We can start from the first session and, at each step, train the network on a new session without using the previous patterns. Of course this is a challenging task because of the risk of forgetting previously learned patterns.

Bibliography

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

[2] Abdullah Algarni. "Parallel Image Processing". In: *Presentation Report* (2009), pp. 21–23.

[3] Ethem Alpaydin. "Introduction to Machine Learning". In: *MIT Press* (2010), pp. 35–37.

[4] Apple. "Apple iPhone7 Machine Learning Processor". In: *Apple* (2016). URL: http://www.apple.com/iphone-7/.

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

[6] C. M. Bishop. "Pattern Recognition and Machine Learning". In: *Springer* (2006), pp. 103–106.

[7] R. Collobert. "Deep Learning for Efficient Discriminative Parsing". In: *VideoLectures.net* (). URL: videolectures.net/aistats2011_collobert_deep/.

[8] Christopher M. Brown Dana H. Ballard. "Computer Vision". In: *Prentice Hall* (1982).

[9] Dongheui Lee Junichi Ishikawa Yoshihiko Nakamura Dana Kulic Christian Ott. "Incremental learning of full body motion primitives and their sequencing through human motion observation". In: *The International Journal of Robotics Research* (2011). URL: http://mediatum.ub.tum.de/doc/1160735/file.pdf.

[10] Daniel M. Wegner Daniel L. Schacter Daniel T. Gilbert. "Psychology, 2nd edition". In: *Worth Publishers* (2011), p. 264.

[11] Facebook. "Facial Recognition App". In: *Facebook* (2016). URL: https://www.facebook.com/facialrecognitionapp.

[12] Justin Johnson Fei-Fei Li Andrej Karpathy. "CS231n: Convolutional Neural Networks for Visual Recognition: Classification". In: *Standford University* (2015). URL: http://cs231n.github.io/classification/.

[13] Justin Johnson Fei-Fei Li Andrej Karpathy. "CS231n: Convolutional Neural Networks for Visual Recognition: Convolutional Networks". In: *Standford University* (2015). URL: http://cs231n.github.io/convolutional-networks/.

[14] Justin Johnson Fei-Fei Li Andrej Karpathy. "CS231n: Convolutional Neural Networks for Visual Recognition: Linear Classify". In: *Standford University* (2015). URL: http://cs231n.github.io/linear-classify/.

[15] Justin Johnson Fei-Fei Li Andrej Karpathy. "CS231n: Convolutional Neural Networks for Visual Recognition: Neural Networks 1". In: *Standford University* (2015). URL: http://cs231n.github.io/neural-networks-1/.

[16] Justin Johnson Fei-Fei Li Andrej Karpathy. "CS231n: Convolutional Neural Networks for Visual Recognition: Neural Networks 2". In: *Standford University* (2015). URL: http://cs231n.github.io/neural-networks-2/.

[17] F. Odone L. Rosasco L. Natale G. Pasquale C. Ciliberto. "Teaching iCub to recognize objects using deep Convolutional Neural Networks". In: *4th Workshop on Machine Learning for Interactive Systems* (2015).

[18] Google. "Search by Image". In: *Google* (2016). URL: https://images.google.com/imghp.

[19] Steven Harnad. "The Annotation Game: On Turing (1950) on Computing, Machinery, and Intelligence". In: *The Turing Test Sourcebook: Philosophical and Methodological Issues in the Quest for the Thinking Computer* (2008).

[20] T. Huang. "Computer Vision : Evolution And Promise". In: *19th CERN School of Computing* (1996), pp. 21–25.

[21] Aaron Courville Ian Goodfellow Yoshua Bengio. "Deep Learning". In: *MIT Press* (). URL: http://www.deeplearningbook.org/.

[22] Pat Langley. "The changing science of machine learning". In: *Machine Learning* (2011), pp. 275–279.

[23] Bastian Leibe. "Understanding Convolutional Neural Networks". In: *Seminar Report* (2014). URL: http://davidstutz.de/wordpress/wp-content/uploads/2014/07/seminar.pdf.

[24] Taghi M Khoshgoftaar Naeem Seliya Randall Wald Edin Muharemagic Maryam M Najafabadi Flavio Villanustre. "Deep learning applications and challenges in big data analytics". In: *Journal of Big Data* (2015). URL: https://journalofbigdata.springeropen.com/articles/101186.

[25] M. McCloskey and N. J. Cohen. "Catastrophic interference in connectionist networks: The sequential learning problem". In: *The psychology of learning and motivation* (1989), pp. 109–165.

[26] Ameet Talwalkar Mehryar Mohri Afshin Rostamizadeh. "Foundations of Machine Learning". In: *MIT Press* (2012), pp. 320–332.

[27] Microsoft. "Microsoft MSDN". In: *Microsoft MSDN* (). URL: https://msdn.microsoft.com/en-gb/.

[28] Roger Boyle Milan Sonka Vaclav Hlavac. "Image Processing, Analysis, and Machine Vision". In: *Thomson* (2008).

[29] T. Mitchell. "Machine Learning". In: *McGraw Hill* (1997), p. 2.

[30] Tim Morris. "Computer vision and image processing". In: *Palgrave Macmillan* (2004).

[31] Geoffrey E.; Williams Ronald J Rumelhart David E.; Hinton. "Learning representations by back-propagating errors". In: *Nature* (1986), pp. 533–536.

[32] Phil Simon. "Too Big to Ignore: The Business Case for Big Data". In: *Wiley* (2013), p. 89.

[33] S. Y Song H.A.; Lee. "Hierarchical Representation Using NMF". In: *Springer Berlin Heidelberg* (), pp. 466–473.

[34] Peter Norvig Stuart Russel. "Artificial Intelligence: A Modern Approach". In: *Prentice Hall* (1995), p. 25.

[35] Peter Norvig Stuart Russel. "Artificial Intelligence: A Modern Approach". In: *Prentice Hall* (2003).

[36] Kai Chen Greg S Corrado Tomas Mikolov Ilya Sutskever and Jeff Dean. "Distributed representations of words and phrases and their compositionality". In: *Advances in neural information processing systems* (2012), pp. 3111–3119.

[37] D. Maltoni V. Lomonaco. "Comparing Incremental Learning Strategies for Convolutional Neural Networks". In: *Artificial Neural Networks in Pattern Recognition* (2016), pp. 175–184.

[38] D. Maltoni V. Lomonaco. "Semi-supervised Tuning from Temporal Coherence". In: *Under review as a conference paper at ICLR 2016* (2016).

[39] Lode Vandevenne. "Image Filtering". In: *Lode's Computer Graphics Tutorial* (2004). URL: http://lodev.org/cgtutor/filtering.html.

[40] Paul J. Werbos. "The Roots of Backpropagation. From Ordered Derivatives to Neural Networks and Political Forecasting". In: *John Wiley & Sons* (1994), pp. 493–498.

[41] Nicole Wong. "Computer Graphics, Computer vision and Image processing". In: *CSDN* (2013). URL: m.blog.csdn.net/article/details?id=9255483.

[42] Evan Shelhamer Yangqing Jia. "Caffe framework". In: *Berkeley Artificial Intelligence Research Lab* (). URL: http://caffe.berkeleyvision.org/.

[43] Aaron C Courville Yoshua Bengio and Pascal Vincent. "Incremental Learning". In: *Encyclopedia of Biometrics* (2009), pp. 731–735.

[44] Aaron C Courville Yoshua Bengio and Pascal Vincent. "Unsupervised feature learning and deep learning: A review and new perspectives". In: *CoRR, abs/1206.5538* (2012).