City University of Hong Kong

Master Thesis

A Library for Fast Kernel Expansions with Applications to Computer Vision and Deep Learning

Author:
J. de Curtó i Díaz
curto@cmu.edu
Carnegie Mellon.
curto.2@my.cityu.edu.hk
City University of Hong Kong.

Supervisors:
A. Smola
smola@cs.cmu.edu
Carnegie Mellon.
C. W. Ngo
cwngo@cs.cityu.edu.hk
City University of Hong Kong.

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science

in the

Department of Electrical Engineering

at

City University of Hong Kong.

November 2014.

Carnegie Mellon.
Pittsburgh, 2014.

Carnegie Mellon

“My heart is in the work”.

A. Carnegie. Motto of Carnegie Mellon.

“The brick walls are there for a reason. The brick walls are not there to keep us out. The brick walls are there to give us a chance to show how badly we want something. Because the brick walls are there to stop the people who don’t want it badly enough. They’re there to stop the other people”.

R. Pausch. The Last Lecture.

“Where East meets West”.

Motto of Hong Kong.

CITY UNIVERSITY OF HONG KONG.

Abstract

Master of Science.

A Library for Fast Kernel Expansions with Applications to Computer Vision and Deep Learning

by J. de Curtó i Díaz.

The main contribution of the thesis is the development of a fast library for approximating kernel expansions, which enables the use of Kernel Methods in large-scale datasets. Kernel Methods are computational costly for big data, this library enables the use of non-linear features in log-linear time. This approximation is based on the Walsh Hadamard. A SIMD implementation of the Fast Walsh Hadamard that outperforms current state-of-the-art methods has been developed. The thesis contains interesting applications to Computer Vision and Deep Learning which can serve as guideline for novel researchers in Machine Learning.

Declaration of Authorship

I, DE CURTÓ I DÍAZ Joaquim, declare that this thesis titled, "A Library for Fast Kernel Expansions with Applications to Computer Vision and Deep Learning" and the work presented in it are my own.

Signed: De Curtó i Díaz.

Date: 22 November 2014.

Contents

List of Figures

List of Tables

Dedicated to De Zarza i Cubero.

Chapter 1

Foreword

This thesis is the result of a dream, is the result of a long journey beginning in old continental Europe to the world city of Asia, Hong Kong, and from there, to the new world, the United States of America, in a big but small and cosy town, Pittsburgh.

The thesis started as a summer internship at Carnegie Mellon, in the Robotics. From there, it grows into a dissertation in the ML Department.

This dissertation encompasses the result of a joint work, both my dissertation and de Zarza i Cubero et al. [2014] are complementary, and some concepts are explained in more detail in one or the other.

Enjoy!

Chapter 2

Introduction

Kernel Methods have proven to be successful in many areas, however one of the main drawbacks for large-scale Machine Learning is that they are too expensive in terms of computation and storage. In order to give a solution to this problem, Le et al. [2013] proposed an approximation called Fastfood that speeds up the computation of a large range of kernel functions, allowing us to use it in big data. Fastfood is based on Random Kitchen Sinks by Rahimi and Recht [2007], where instead of using random GAUSSIAN matrices a combination of random matrices diagonal and HADAMARD matrices are used. This approximation will let us use Kernel Methods in applications where large training sets, distributed computation and real time prediction are important.

The main contribution of this thesis is to provide Library McKernel that implements Fastfood for CPU intensive computation, both with distributed and non-distributed versions. McKernel is a SIMD oriented implementation of Fastfood, and it is the first open implementation of the algorithm to be found in the current literature. The key bottleneck of the kernel expansions approximation described in Le et al. [2013] is to use the Walsh Hadamard. We have implemented a SIMD oriented Fast Walsh Hadamard, which is based on the COOLEY TUCKEY algorithm. Our implementation tries to maximize the cache hits and provides superior performance than all the rest FWH implementations out there, being faster than the current state-of-the-art, Spiral (Johnson and Püschel [2000]). Also, SIMD vectorized operations have been used where possible to achieve a compelling performance and a distributed in mind version of the algorithm, where Pseudo-random Numbers are generated by hashing, is provided.

This implementation opens the race to use Kernel Methods in large-scale distributed computation and can be the key contributor to use kernel expansions efficiently in Deep

Neural Networks. Kernel Methods have still something to say in the era of Deep Learning and the approximation Fastfood could enable this new renaissance.

What's more is that Fastfood makes Kernel Methods usable in embedded systems such as your own mobile phone. This is a groundbreaking result that can revolutionize the use of Kernel Methods making it possible to use it in a large range of applications.

The thesis includes applications to Computer Vision and Deep Learning. We go from dataset construction, through extraction of features to classification. First, a conventional handcrafted system of Computer Vision is studied and implemented. Then the implementation of McKernel gives us the way to go from linear classification to kernel based classification. Finally, an introduction to Deep Learning is done and McKernel is embedded in a deep network. Concepts of supervised and unsupervised learning are spread throughout the thesis making it a good read for the novel ML researcher.

Chapter 3

Preliminary Work

In this chapter we build from scratch a face recognition system for estimation of ethnicity. This chapter serves as introduction to Computer Vision and explains the usage of some basic ML algorithms. The material for this chapter was developed in collaboration with the Robotics at the HS Laboratory.

- First we coded in Python and Matlab a program to download massively face images; we retrieve the URLs from API Flickr and store them in a file .txt. We use different search attributes to do a good broad search. Then we use Matlab to download the images massively and filter them by tags so that we reduce noise.
- Once we have the images downloaded (2 million photos), we label 10.000 images (balanced in terms of ethnicity) using MTurk, and we design an interface in HTML and JavaScript to do so.
- With the labeled images, we run an algorithm which is able to extract 49 facial points from face images (Xiong and Torre [2013]). These images are then cropped to extract the face and processed by a normalization and an affine transformation before extraction of features.
- In the step of extraction of features, we are using LBP multi-scale uniform (3 different radius) in HSV color space (for each color channel) for each of the 49 facial points. We also generate a 4th informative channel by using some techniques of preprocessing into the original image (gamma correction, Difference of Gaussians and histogram equalization) and then perform multi-scale LBP Uniform to the preprocessed image.
- Finally, in the classification step, we are using SVM Multiclass Linear (LIBSVM library).

Further comments:

We have tried different handcrafted LBP Uniform configurations. What has resulted to work better is to use LBP Multiscale with three different radius. We use radius 4, 5 and 6. It works better when we sum the three radius rather tan concatenate them. We tried different color spaces and crossvalidated the parameters. For estimation of ethnicity, what worked best is using HSV color space. We do the LBP Multiscale for each of the channels. We added a 4th informative channel using preprocessing Tan and Triggs [2007] using gamma correction, DoG and histogram equalization. Finally, SVM Linear works better than AdaBoost with single stumps.

3.1 Why Build Our Own Dataset? Exploiting Flickr

You can find many datasets publicly available for research in Computer Vision. Why do we want to build our own? Machine Learning is about learning from data, which means literally that you have to care about two things: which algorithms are you going to use for learning and which data are you going to use. Sometimes, it is a common trend to just focus on finding the best possible algorithm (the more complicated and difficult to understand the best, many think), but it is surprising how good you can do by just using a simple algorithm (like the perceptron) if fed it with good data. The idea is that if you don't have good labeled data, no matter how good your algorithm is, you won't be able to learn anything useful from that.

This is why big companies like to use their own data, Facebook or Google have huge amounts of data generated by its own users, but for ML researchers, being able to access good data that fits a particular research interest is not so easy. In our particular problem, there are a lot of face datasets publicly available. However, first and foremost, they do not contain much pose estimation variation and changes of illumination, what means that the face recognition will be not robust against changes of that kind. And second, you cannot build a system of estimation of ethnicity if all the data is from caucasian, and no african american or asians are present. This is a particular problem, but can serve the purpose of understanding why data is so important. Computer Vision and Machine Learning is not just using algorithms in a black box fashion, is about understanding the problem, knowing your data, and using your data properly so that your algorithm is able to generate a strong model.

For our purpose, we exploit the weaknesses of Flickr by using its API and crawl from the web almost two million photos. We download the URL's using Python and then we download the images using Matlab.

3.1.1 Code Highlights

- PYTHON code downloads the URL address of the images using the API access key in Flicker.
- MATLAB code uses the generated text files containing URLs to download the images anonymously from Flickr. We include the following:
  - A filter to avoid noisy images: we use a huge amount of negative tag words so that we are able to discard a huge amount of non-useful photos.
  - The ability to choose which time interval do you want to download, e.g. photos from 2006 to 2008.
  - Selection of country of origin to be able to have a balanced dataset in terms of ethnicity.
  - Selection of multiple tag words to be able to search for faces, selfies, and so on. This enables to download images with different pose and illumination.

Performance time: two weeks to download a two million image dataset, using a server with 32 cores.

3.2 Getting the Labels

Labeling consists on saying for each image, if it is male / female, kid / teenager / adult / senior, smile yes / smile no, ... That is to say, give for each image its attributes to train in a latter step the classifier. When we say we need good data, we mean we need diverse and well-labeled data. This is one of the main drawbacks of supervised learning, you need a huge amount of hand labeled data. You are teaching the computer by examples, but you need to generate those examples properly so that your computer is able to learn what has to do. This process is difficult and tedious and in order to speed it up we use a state-of-the-art tool named MTurk.

MTurk is a crowdsourcing internet marketplace from Amazon that enables individuals to coordinate the use of human intelligence to perform tasks that computers are currently

unable to do. We developed a MTurk interface for labeling the downloaded images. We coded a robust and easy-to-use interface in HTML and JavaScript and upload a subset (10.000) of the downloaded images to the web.

Before uploading the images to MTurk, we need to preprocess the data. We use a system of face recognition based on Xiong and Torre [2013] to extract images of faces, and only upload online those which are properly tracked by the 49-points face tracker. This enables us to generate the landmark points, which are the coordinates of 49-points of the face that we will later use for extracting features.

3.2.1 MTurk

You can see next the HTML interface, Figure 3.1, where the Turkers (people working at MTurk) can select the proper face attributes.

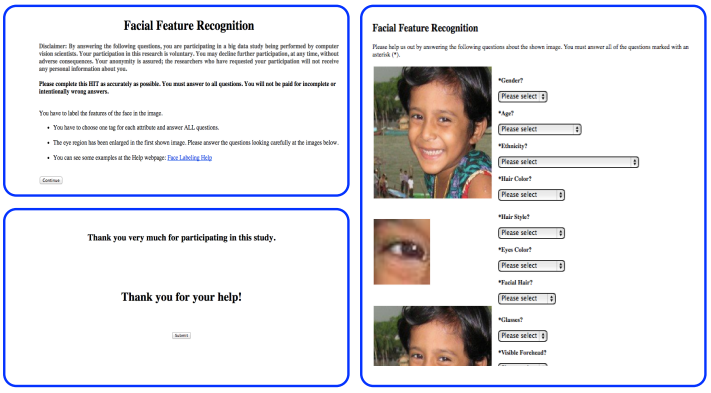


FIGURE 3.1: Interface MTurk.

We use the server at the HS Laboratory and a GOOGLE sites website for the purpose of labeling. We code several bash scripts to be able to upload all the preprocessed images to the server and generate the proper CSV (Comma Separated Value) files needed for MTurk.

3.3 Extraction of Features

We can summarize the procedure as follows:

Using the face tracker in Xiong and Torre [2013] on the original images we extract the landmarks and normalized images (using affine transformation) and save them into different folders. Next, we write a script that using the CSV file that is generated with MTurk, is able to extract the labels for each image and generate a JSON file with the corresponding labels and attributes for each image.

We generate a CSV file that contains the image path, landmark path and label with a row per image which will be the input of the LBP feature extraction. We have coded an optimized LBP Uniform feature extraction in C++. The input parameters are patchsize (we are doing LBP for each of the coordinate landmarks identified by the face tracker), radius (we can use different LBP radius) and number of neighbors. The output is in SVMLight format and it is formed by extracting 49 LBP histograms (one for each landmark point) and concatenating the histograms to form a $49 * 59$ vector of features.

The step of extraction of features in the preliminary work has been done using first the face tracker to recognize 49 key points in the face, called landmark points. Then, we have applied a LBP handcrafted feature in each patch with center a landmark point.

Once the base system was working, we started concatenating LBP Uniform features with different radius. We did simulations and we selected the best three radius according to the accuracy performance of the classifier. Then, we came up with the idea that instead of concatenating, we could just do the sum. This makes sense as we are using a linear classifier and the LBP Uniform just considers a given length of histogram and therefore it is a correct assumption.

The next idea was to use the three channels of the image as we were converting the color image to grayscale in the previous simulations. We tried different tests on the dataset to see which color space worked better (YUV, LAB, HSV, HSL, YCrCb,...). After some simulations the best performance was given by HSV color space. This means that now we are using a LBP Multiscale on every channel of the image. This increases the performance around 5%, which is quite good.

In the process of generating a handcrafted powerful extraction of features for estimation of ethnicity, Figure 3.2, we further improve the performance by using a good technique of preprocessing, Tan and Triggs [2007], which consists on applying gamma correction, Difference of Gaussian (DoG) filtering and contrast equalization before applying the LBP

texture descriptor. It gives between 1 and 2 % increase if added as a 4th informative channel to the 3 HSV channels. This preprocessing technique tries to increase the performance by making the system robust against changes of illumination.

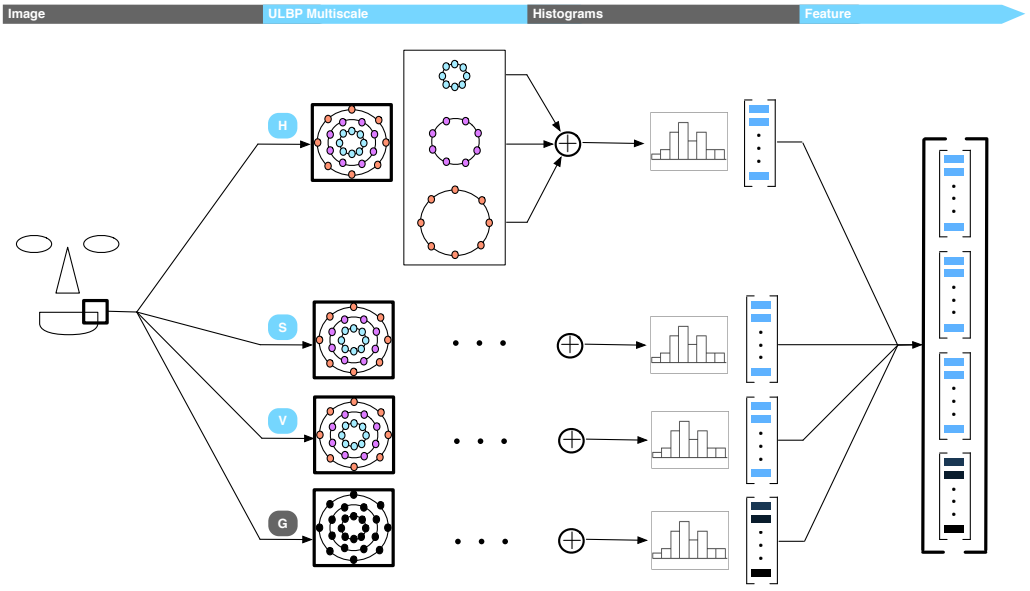


FIGURE 3.2: Implemented System.

3.3.1 LBP Handcrafted Features

We use features Multiscale Uniform Local Binary Pattern in four different channels: the HSV channels (see Table 3.2) and a fourth informative channel in gray scale with preprocessing Tan and Triggs [2007]. But what is LBP?

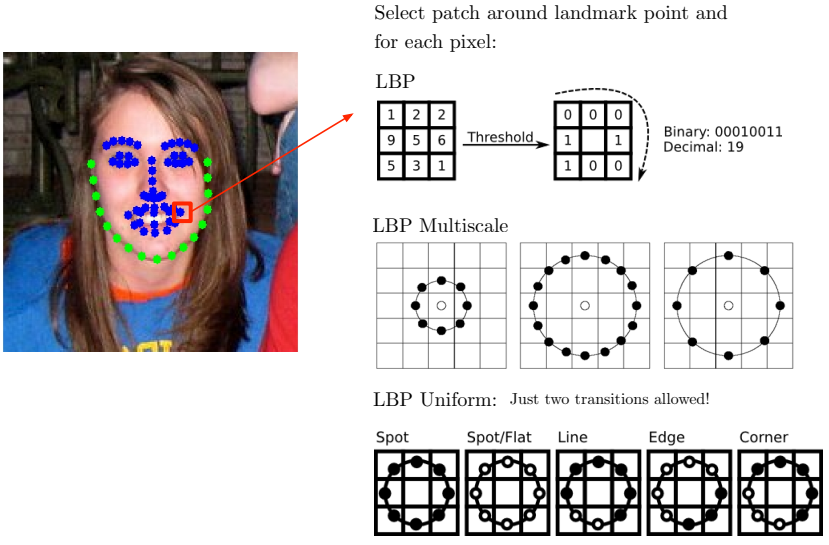


FIGURE 3.3: Local Binary Pattern.

LBP, Figure 3.3, stands for Local Binary Pattern (Ahonen et al. [2011]), consists on a simple yet very efficient texture operator which labels the pixels of an image by thresholding the neighborhood of each pixel. First, subtracting each point with the center pixel, taking its sign and finally converting the result into a binary number. We then do an histogram of the result and use it as vector of features. Due to its discriminative power and computational simplicity, LBP texture operator has become really popular in many real time applications, such as face detection. We implement it using Matlab and C++.

LBP Multiscale generalizes LBP features to different radius (instead of assuming just radius 1) and different number of neighbors, making it possible to encompass both local and global information.

LBP Uniform just considers binary numbers with two transitions (from 0 to 1 and vice versa), so given a fixed number of neighbors all the histograms have the same length (length 59 using 8 neighbors). Using ULBP we compress the dimension of the vectors of features and its performance is as good as plain LBP.

3.4 Classification Stage

We have used Adaboost and Support Vector Machines (LIBSVM library in Chang and Lin [2011]). SVM achieved superior performance.

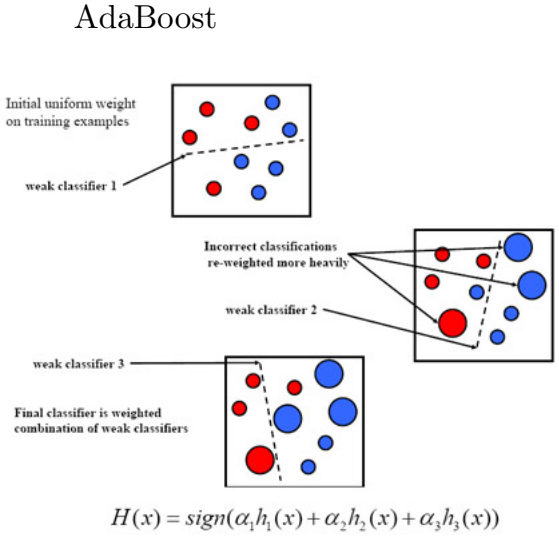


FIGURE 3.4: Adaboost.

The algorithm AdaBoost, Figure 3.4 relies on the idea that using successive classifiers that are just better than random (named as weak learners) in a consecutive way can give you a really good classifier (named as strong learner). The algorithm originated from a mathematical proof and then Freund and Schapire came up with a practical implementation.

Support Vector Machines is a well known classifier that tries to separate the data linearly using a hyperplane that maximizes the margin (average distance between hyperplane and samples).

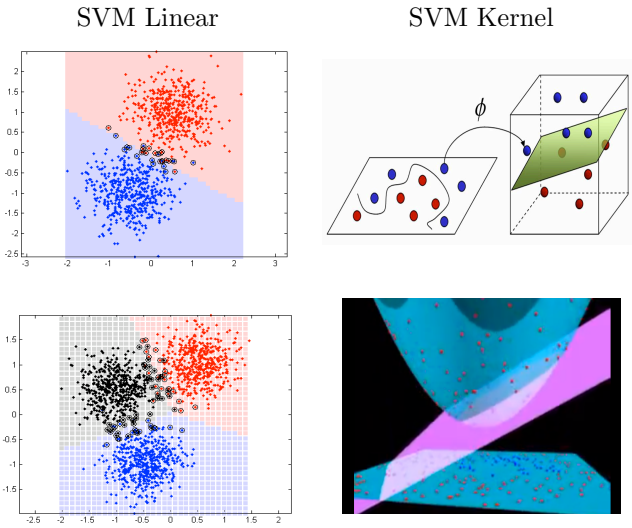


FIGURE 3.5: SVM Linear and SVM Kernel.

A theoretical explanation of Kernels and SVMs is provided in the next section. At this early stage, we have only seen the usage of linear Support Vector Machines. Kernels allow us to generalize SVM to SVM Kernel, where low dimensional observations are mapped into a high dimensional space, Figure 3.5, where the data can be separated linearly. Although in this preliminary work we do not use a non-linear classifier, the need to improve the current system, brings us to explore SVM kernel. In particular, in the thesis we study and implement a library for fast kernel expansions, which will allow us to use kernel based methods in our system easily.

3.5 Experimental Results

3.5.1 Implementation Hints

In the first step, using the original images we save the landmark coordinates in one folder and the normalized images in another. Then, using a bash script, we can save the labels from MTurk CSV to JSON file format. Finally, we use Matlab to generate a file to save the image path, landmark paths and the label of each image in a CSV file format that will allow us to be able to wire the data into the system.

In the second step, we use the output CSV file with all data information to do the LBP feature extraction. We use an optimized LBP Uniform function in C++ with patchsize, radius and number of neighbors as input parameters to perform in the final step the crossvalidation. We wrote the code to generate an output file in SVMLight format.

Once we have the features, we proceed with the classifier. It has train and test files as input data and generate a model and a result file as outputs.

Finally, in the last step a MATLAB script was written to perform the process of cross-validation, calling the binary files of the C++ code and parallelizing the execution in a server with twelve cores. This step helped us choose the best parameters for the LBP and the classifier. The full procedure is illustrated in Figure 3.6.

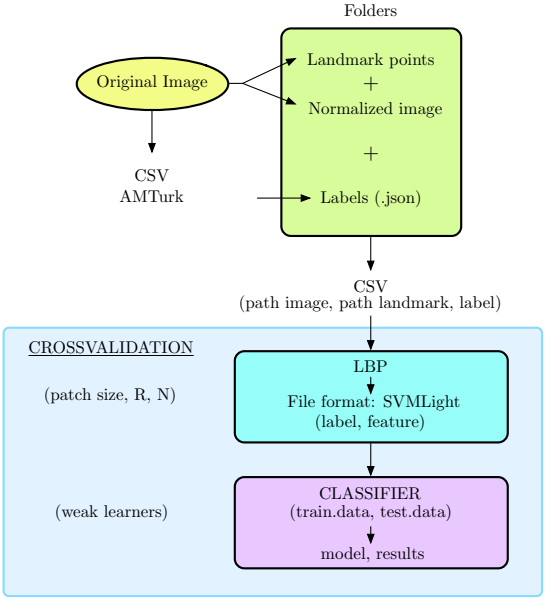


FIGURE 3.6: Implementation Hints.

3.5.2 K-Fold Crossvalidation

Crossvalidation is a technique used to choose the best parameters of your system, Figure 3.7. It is based on grid search. You choose a set of parameters and crossvalidate the system to be able to determine which ones work best. K-Fold Crossvalidation splits your dataset into training, validation and testing. Normally we use around 85% for training and validation, and 15% for testing. Of the 85%, we divide it into k-chunks, in each iteration we use one of the k chunks for validation and the other remaining $k-1$ for training. We average the results by the number of iterations and we compare the results with the different parameters. Those parameters that give better accuracy are the ones that are selected to use in the testing dataset and see the performance.

We used 10-Fold Crossvalidation to choose the best parameters to use for the given problem of ethnicity. We are using 13850 images. We crossvalidate the patch size (which is the region size of the image around the landmark points we are going to use to compute the extraction of features), the number of neighbors and the radius.

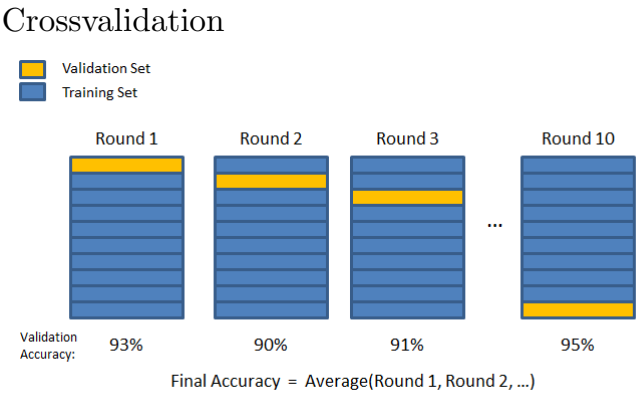


FIGURE 3.7: K-Fold Crossvalidation.

Using this method we have crossvalidated the LBP parameters (patch size, number of neighbors and radius) and the color space (using RGB, LUV, YCrCB and HSV). In classifier AdaBoost we also needed to crosvalidate the number of weak learners. However, SVM outperformed Adaboost, so we will focus in the SVM results.

In the case of SVM, we also needed to choose the C-parameter of the SVM.

3.5.3 Benchmarks

In this section we have used LIBSVM library to perform the step of classification.

All the following crossvalidation results are done using ULBP Multiscale with SVM Linear Multiclass as a classifier applied to classification of ethnicity (using four classes: Caucasian, African American, East Asian and South Asian).

After crossvalidation, the best patchsize, number of neighbors and radius are 11, 8 and 4 respectively. Moreover, the number of different radius to apply multiscale achieving maximum performance is 3 and the best three values are 4, 5 and 6.

The results for color space crossvalidation are shown below:

Color space	RGB	LUV	YCrCb	HSV
Accuracy (%)	77.4983	78.1971	78.2669	81.4116

TABLE 3.1: Color Space K-Fold Crossvalidation Applied to Classification of Ethnicity.

Next, we can see some more experimental results of the overall system.

	Accuracy (%)
ULBP. SVM Linear.	77.71
ULBP Multiscale(3). SVM Linear.	78.27
ULBP Multiscale(3). SVM Linear. HSV.	81.42
ULBP Multiscale(3). SVM Linear. HSV. Preprocessing.	82.36
ULBP Multiscale(3). SVM Linear. HSV. Optimized preprocessing.	85.02

TABLE 3.2: Results Estimation of Ethnicity.

We have to consider that we are doing our tests in a really challenging dataset, as the images are taken from real photos from Flickr. The accuracy could seem low but the model developed will work really well for real settings like webcams and so on, where image quality is not so good and difference in pose and illumination are important.

We can see first the result when just using one LBP channel with SVM Linear. The accuracy gets better when using 3 different radius, however, if we increase and use 4 or more, the accuracy diminishes. It seems that using 3 different radius enables the system to get both local and global information. Next we can see how using HSV color space gets better performance in the estimation problem of ethnicity. In the next test, we use a preprocessing of the images as a 4th informative channel, which also improves the

performance. In the final result we focus on improving the data used by correcting some wrong labels. We use a script which is able to detect when the classifier is not very sure about the decision (score) and we see if the label is correct or not and get it correct.

Chapter 4

SVM, Kernels and Beyond

A concise introduction to Kernels and Support Vector Machines is given. We formulate SVM as a problem of optimization to understand what is the role of kernels and why the kernel trick is so accepted. Next, a brief description of randomized algorithms is provided. This chapter is intended as a straight to the point introduction that will serve as knowledge background for the rest of the chapters.

4.1 Why Are Kernels Popular?

The key idea at the heart of kernels is that many ML algorithms can be expressed as a function of inner products between observations, that is $\sum_c \alpha_c \langle \mathbf{x}, \mathbf{x}' \rangle$. The point here is that we can use the kernel trick (Schölkopf and Smola [2002]) by replacing these inner products between observations by kernel functions. These kernel functions are feature maps that transform the low dimensional observations into a high dimensional feature space, where the data can be separated linearly. Therefore, we are able to use non-linear methods without even the need to change the algorithms. This versatile behavior made Kernel Methods popular in almost any application of Computer Vision and Machine Learning.

4.2 Support Vector Machines

This section is based on Song [2008].

Let's assume we have a set of labeled images $\{\mathbf{x}_c, \mathbf{y}_c\}$. The idea behind learning is that we want to be able to predict the labels of new examples. A linear Support Vector

Machine uses a hyperplane to separate the data, $f(\mathbf{x}) = \text{sign}(\langle \mathbf{w}, \mathbf{x} \rangle + b)$, such that the margin (minimum distance between the hyperplane and the observations) is maximized.

We can cast SVM as a problem of optimization by

$$\max_{\mathbf{w},b,\gamma} \gamma \tag{4.1}$$

$$\text{s.t. } \forall c, y_c(\langle \mathbf{w}, \mathbf{x} \rangle + b) \geq \gamma \text{ and } ||\mathbf{w}||^2 = 1. \tag{4.2}$$

We can express the dual problem with the Lagrangian as follows

$$L = -\gamma - \sum_c \alpha_c [y_c(\langle \mathbf{w}, \mathbf{x} \rangle + b) - \gamma] - \lambda(||\mathbf{w}||^2 - 1), \forall \alpha_c \geq 0, \lambda \geq 0 \tag{4.3}$$

and then differentiate the primal variables $\mathbf{w}$, b, γ and set them to zero

$$\frac{\partial L}{\partial \mathbf{w}} = -\sum_c \alpha_c y_c \mathbf{x}_c - 2\lambda \mathbf{w} = 0 \tag{4.4}$$

$$\frac{\partial L}{\partial b} = -\sum_c \alpha_c y_c = 0 \tag{4.5}$$

$$\frac{\partial L}{\partial \gamma} = -1 + \sum_c \alpha_c = 0. \tag{4.6}$$

If we then substitute those results back in 4.3 and do some further computation we get

$$\min_\alpha \sum_c \alpha_c \alpha_z y_c y_z \langle \mathbf{x}_c, \mathbf{x}_z \rangle \tag{4.7}$$

$$\text{s.t. } \sum_c y_c \alpha_c = 0, \sum_c \alpha_c = 1, \forall \alpha_c \geq 0. \tag{4.8}$$

We are now with a problem with affine constraints, where the Hessian of the objective is $\mathbf{A}$ and is positive semidefinite. Therefore, it is a CONVEX problem.

What's more, we can see how the objective is just a function of the observations. This is where the kernel trick originates, we can substitute inner products by kernel functions $k(\mathbf{x}, \mathbf{x}') = \langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle$ just having in mind that we have to assure that $\mathbf{A}$ is positive semidefinite, where now $\mathbf{A}_{cz} = y_c y_z k(\mathbf{x}_c, \mathbf{x}_z)$.

The SVM Kernel decision function can be now expressed as $f(\mathbf{x}) = \langle \mathbf{w}, \phi(\mathbf{x}) \rangle + b$.

This basic introduction to the SVM, although very brief, will serve as basis to understand the main concepts throughout the thesis.

4.3 Approximating Kernel Expansions

Here we introduce the idea that we can use random features as an alternative to the kernel trick. The main drawback of SVM kernel based is its high computational cost. Random features will allow us to use them on top of linear classifiers which means that we will be able to use Kernel Methods in large-scale datasets.

When using random features some parameters are randomized and then we optimize over the rest. The idea behind this is that ML algorithms try to optimize parameters that are indeed not necessary to optimize. This smart way of doing, makes old supervised learning scalable, extremely fast and easy to implement.

Kernel machines give really good performance as classifier, but speed is not one of its strengths. In random features we wire the data through a randomized procedure and then into a linear algorithm. This randomization is intelligently thought so that when using this random features in front of a linear algorithm we get the same behavior as we were using a non-linear version of the algorithm.

Kernel Methods are based on the idea that inner products in high dimensional feature spaces can be computed using a kernel function k:

$$k(x, x') = \langle \phi(x), \phi(x') \rangle \tag{4.9}$$

where you do not need to know $\phi(x)$ explicitly, as long as you have access to k. The kernel trick can be employed as follows, being $f(x)$ the decision function of an example x

$$f(x) = \langle w, \phi(x) \rangle = \sum_{c=1}^{N} \alpha_c \langle \phi(x_c), \phi(x) \rangle = \sum_{c=1}^{N} \alpha_c k(x_c, x). \tag{4.10}$$

In large-scale datasets this expansion results to be very costly. In Steinwart and Christmann [2008] it is proved that as the dataset grows, the cost of evaluating f also grows

linearly.

The problem now can be stated as follows: is there a way to scale Kernel Methods to big data? Companies such as Google, Amazon or Facebook use extremely large datasets. Is there a way for them to use Kernel Methods efficiently and effectively in their algorithms?

Random Kitchen Sinks in Rahimi and Recht [2007] is based on the idea that we can approximate the function f in 4.10 using a matrix multiplication with a random matrix Gaussian. For large-scale problems, this is much faster than the kernel trick. The two main cons of this method are the need to store and multiply by a random matrix.

4.4 Random Kitchen Sinks

We will introduce conceptually the key idea behind Fastfood.

The theoretical proof that settles the framework for approximating kernel expansions is MERCER Theorem. It guarantees that kernel functions can be expressed as an inner product in some HILBERT Space (that is a space where we can define an inner product)

Theorem 4.1 (Mercer)**.** *Any kernel* $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ *satisfying* $\int k(x, x')f(x)f(x')dxdx' \geq 0$ *for all* $L_2(\mathcal{X})$ *measurable functions* f *can be expanded into*

$$k(x, x') = \sum_z \lambda_z \phi_z(x)\phi_z(x'). \tag{4.11}$$

Being $\lambda_z > 0$ *and the* ϕ_z *are orthonormal on* $L_2(\mathcal{X})$.

It is here where Rahimi and Recht [2007] suggest to approximate 4.1 as follows

$$\lambda_c \sim p(\lambda) \text{ where } p(\lambda_c) \propto \lambda_c \tag{4.12}$$

$$\text{and } k(x, x') \approx \frac{\sum_z \lambda_z}{n} \sum_{c=1}^{n} \phi_{\lambda_c}(x)\phi_{\lambda_c}(x'). \tag{4.13}$$

This technique attempts to solve the scalability problem in Kernel Methods through the use of randomization.

$$\arg\min \qquad \mathcal{R}_e\left(\sum_z \alpha(w_z)\phi(x; w_z)\right) \tag{4.14}$$

where $\mathcal{R}_e$ denotes the framework of minimization of the empirical risk.

In the above equation, instead of optimizing over α and w, it randomizes w while minimizing over α.

This method approximates the decision function f with

$$\hat{f}(x) = \sum_{z=1}^{n} \alpha_z \hat{\phi}(x; w_z) \tag{4.15}$$

where $\hat{\phi}(x; w_z)$ is a feature function gained through randomization.

This work goes in contrast to previous work as authors try to obtain an explicit function space expansion directly. This can be applied to invariant kernel functions by:

- Generate a random matrix Gaussian G, size $n \times d$.
- For each x compute Gx and apply a nonlinear Φ to each one of the coordinates separately: $\phi_c(x) = \Phi([Gx]_c)$.

This approach is potentially less cheaper than reduced set kernel expansions.

4.5 Fastfood: Kernel Expansions in Log-linear Time

Fastfood follows closely Random Kitchen Sinks but it accelerates from $O(nd)$ to $O(n \log d)$ while requiring only $O(n)$ rather than $O(nd)$ storage. The key is to accelerate the multiplication by a random matrix.

The main idea of the algorithm is that HADAMARD matrices, when combined with scaling matrices Gaussian, behave very much like random matrices Gaussian.

Le et al. [2013] prove that the approximation Fastfood is unbiased, has low variance, and concentrates almost at the same rate as Random Kitchen Sinks while being 100x faster with 1000x less memory.

Let Z be the random matrix Gaussian we want to parameterize by:

$$V := \frac{1}{\sigma\sqrt{d}} SHG\Pi HB \tag{4.16}$$

where

- B is a matrix diagonal with entries i.i.d. $+1$ and -1 from a distribution Uniform.
- H is the Walsh Hadamard computed using FWH in Library McKernel.
- Π is the matrix of permutation generated using algorithm Fisher Yates in Library McKernel.
- G is a matrix diagonal which entries are i.i.d. and follow a random distribution Normal (0,1).
- S is also a matrix diagonal which entries follow a chi distribution with d degrees of freedom and are multiplied by the FROBENIUS norm of matrix G.

As S, G and B are matrices diagonal they can be computed and stored in the worst case in $O(n)$. Π can be computed in linear time using algorithm Fisher Yates and H can be computed using the FWH algorithm in place and in time $O(n \log(d))$. So, the total computational time of Fastfood is carried out in $O(n \log(d))$ and the storage cost is $O(n)$.

4.6 Walsh Hadamard

Walsh Hadamard (WH) can be defined in a recursive way as follows:

If we let $H_0 = 1$, then $\forall m > 0$, H_m becomes

$$H_m = \frac{1}{\sqrt{2}} \begin{pmatrix} H_{m-1} & H_{m-1} \\ H_{m-1} & -H_{m-1} \end{pmatrix}. \tag{4.17}$$

WH can be implemented naively with computational complexity $O(N^2)$. However, we will use an efficient algorithm to compute it in $O(N \log N)$, the Fast Walsh Hadamard (FWH).

FWH is based on an approach divide and conquer that recursively breaks down a WH of size N into two small WH of size $\frac{N}{2}$ as can be seen on Figure 4.1.

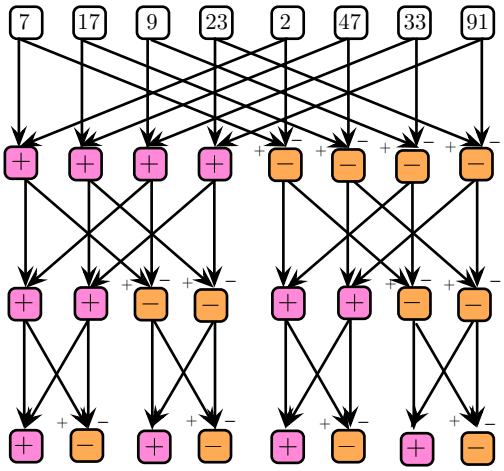


FIGURE 4.1: Fast Walsh Hadamard.

Chapter 5

Writing Fast Numerical Code: Brief Introduction

Intrinsics Intel related questions are typically found when you prepare for IT job interviews, and it is not something strange, big IT companies want good programmers able to write fast code. Intrinsic functions Intel are an example of SIMD (Single Instruction Multiple Data), beginning with the MMX family, through SSE and now with AVX (Advanced Vector Extensions) have become increasingly popular in the latter times. The possibility to vectorize your code to increase its performance is something really useful when dealing with time constrained computing performance.

SIMD are extensions to the CPU instruction set architecture that allow programmers to write fast numerical code for their computing intensive applications. We will see a brief introduction of the main concepts next.

SIMD intrinsic functions allow the programmer to access to assembly like functionality and write vectorized code. The code could seem difficult to read at first, however, the considerable speed up makes them worth to try. Intrinsic functions allow to use fast CPU intensive code, and could be used in applications where some years ago only GPU could be used.

If you are using SSE2 the data type __m128 you are going to use is a 128 bit register. In these registers you can put two doubles or four floats.

In order to load from an array into the proper register, you can use:

```
__mm128 c = _mm_loadu_ps(&data)
```

This instruction would load 4 floats into register *c*. If you are using double precision (doubles) then you need to use:

```
__mm128 c = _mm_loadu_pd(&data)
```

in order to load 2 doubles into register *c*.

Once you have loaded the data into registers, you can use SIMD operations, such as addition or subtraction as

```
__mm128 a = _mm_loadu_ps(&data[0]);
__mm128 b = _mm_loadu_ps(&data[4]);
__mm128 c = _mm_add_ps(a, b);
__mm128 d = _mm_sub_ps(a, b);
```

The first two instructions load the first 4 floats of data into *a*, and the next 4 floats into *b*. The third instruction sums the 4 floats of register *a* to the 4 floats of register *b* and saves the result into *c*. The fourth instruction computes the difference between the 4 floats of each register and saves it into *d*.

There are more complicated expressions such as horizontal add, shuffle inside a register, change of sign or dot product. However, this example ought to be sufficient to get started.

Chapter 6

SIMD Implementation of the Fast Walsh Hadamard

We have implemented an iterative algorithm to compute the FWH. It computes half of the vector going down in deep, and then it goes from bottom to top solving iteratively the remaining computations. Recursions are avoided to decrease stack overhead and cache hits are maximized by using this structure.

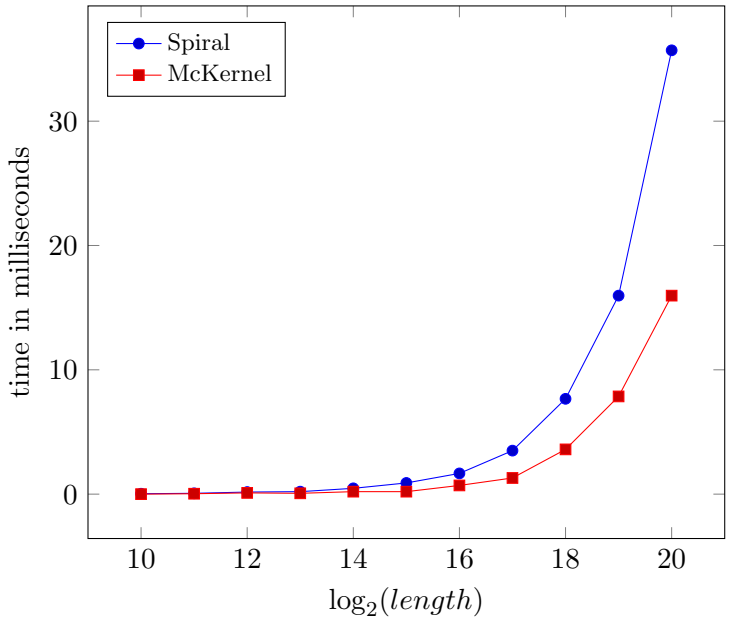
SIMD Intrinsics (SSE2 and SSSE3 using 128 bit registers) are used to improve the performance. A 256 bit register version with AVX instructions is also delivered.

In average, our FWH goes at 1Gflop, and is faster than current state-of-the-art methods.

6.1 Benchmarks

We can see some timing results compared to library Spiral next.

The experiments are done generating 300 different vectors and comparing and computing the average speed of both algorithms. The length of the vector goes from 2^{10} to 2^{20} and we plot the $\log_2$ of the length and the time in milliseconds. We use single CPU comparison in an Intel Core i5-4200 CPU @1.60GHz machine.



We can clearly see that McKernel outperforms Spiral. Moreover, Spiral needs to have precomputed trees and by default can only compute until length 2^{20} whereas our implementation can work for any given size.

Chapter 7

Permutation Using Reservoir Sampling

The matrix of permutation Π could be generated with order $O(n \log n)$ by augmenting the integers $1, \ldots, n$ with random keys, forming value key pairs $(r_1, 1) \cdots (r_n, n)$ where each r_c is a random number Uniform $[0, 1]$. Sort these elements by key using an $O(n \log n)$ algorithm, for instance Quicksort, and then discard the keys to get the permutation. However, we can do this better.

Reservoir sampling is a randomized algorithm for choosing k samples from a list of n items, where $k \leq n$ being n a really large number. This algorithm is highly used by Google and Facebook when they are handling search queries in large arrays of data that do not fit into main memory.

The extreme case where $k = n$ is the shuffle algorithm Fisher Yates, which is the optimum ($O(n)$ operations) algorithm to permute an array of n elements. The idea is to start from the first element of an array $\{1 \ldots n\}$, pick another element uniformly from the remaining set. Swap this new selected element with the current item. Repeat this procedure till you get to the $n - 1$ position to get the desired permutation.

Chapter 8

Fast Implementation of Fastfood: Library McKernel

The material of this chapter is written in collaboration with de Zarza i Cubero et al. [2014].

This chapter provides a description of the Fast Implementation of Fastfood (McKernel) that implements Fastfood for CPU optimized distributed computation and non-distributed computation. McKernel is a SIMD oriented implementation, and it is the first open implementation of the algorithm to be found in the current literature. The key bottleneck of the kernel expansions approximation described in Le et al. [2013] is to use the Walsh Hadamard. We have implemented a SIMD oriented Fast Walsh Hadamard based on the COOLEY TUCKEY algorithm, which is faster than the current state-of-the-art Johnson and Püschel [2000]. Also, SIMD vectorized operations have been used where possible to achieve a superior performance and a distributed in mind version of the algorithm, where Pseudo-random Numbers are generated by hashing, is provided.

8.1 Description of the Software

In Random Kitchen Sinks instead of computing RBF GAUSSIAN kernel

$$k(x, x') = \exp(-||x - x'||^2/(2\sigma^2)) \tag{8.1}$$

the method computes

$$k(x, x') = \exp(i[Zx]_c) \tag{8.2}$$

where z_r is drawn from a random distribution Normal.

In Fastfood Z is parametrized by V as

$$V := \frac{1}{\sigma\sqrt{d}} SHG\Pi HB \tag{8.3}$$

where

- B is a random matrix diagonal with i.i.d. entries $+1$ and -1.

  B's entries are generated by a distribution Binomial by drawing random numbers from a distribution Uniform. In the single CPU machine version, we store in a vector the positions of the elements in the diagonal with -1 entries and instead of multiplying each element we just flip the sign when necessary, reducing half of the storage and computation. In the distributed version, Pseudo-random Numbers are generated using hashing. We can generate numbers Uniform from a function of hashing $h(c, z)$ with range $[0 \dots N]$ just by setting $U_c = h(c, z)/N$. In this way we generate distributed random numbers that can be recomputed on the fly. McKernel uses Murmurhash, it is a fast non cryptographic function of hashing with good probability distribution.

- H is the Walsh Hadamard computed in place with FWH in McKernel.

  Fast Walsh Hadamard is an approach divide and conquer to solve the Walsh Hadamard. Implementation McKernel is based on halving recursively the input vector and doing subsequent sums and subtractions. This idea which is based on the COOLEY TUCKEY algorithm performs extremely well.

  McKernel implements FWH so that it maximizes cache hits and therefore CPU performance, achieving better results than the current state-of-the-art methods. Operations are vectorized using SIMD intrinsic functions. The computation is done iteratively adding and subtracting halves of the input vector until it arrives to the one-length vector. For computational efficiency, it computes from top to bottom, and then the remaining computation is done from bottom to top, being able to maximize cache efficiency and easily being able to use a pre-existing unrolled small routine to improve CPU speed. We use a full iterative algorithm,

avoiding any kind of recursive function. This approach is due to the fact that recursive algorithms need to put parameters to the stack after each call and this damages the performance of a fast implementation. Also, iterative algorithms are better suited for distributed computation of large-scale data.

- Π is the matrix of permutation generated using algorithm Fisher Yates in McKernel.

  Shuffle algorithm Fisher Yates is the optimum algorithm ($O(n)$ operations) to permute an array of n elements. The idea is to start from the first element of an array $\{1 \dots n\}$, pick another element uniformly from the remaining set. Swap this new selected element with the current item. Repeat this procedure till you get to the $n-1$ position to get the desired permutation.

  McKernel permutes row by row the input matrix by using a vector permutation Fisher Yates.

- G is a random matrix diagonal with i.i.d. numbers random Normal.

  G's entries are drawn form a distribution standard Normal $N(0,1)$. The vector matrix diagonal product is vectorized using SIMD intrinsics to speed up the computation. In the distributed version of the code we use BOX MULLER transform in Box and Muller [1958] to draw random variables Normal from variables Uniform using hashing. We generate a random number Normal from two values of the function of hashing as follows:

$$P_{cz} = (-2 \log h_1(c,z)/N)^{1/2} \cos(2\pi h_2(c,z)/N). \tag{8.4}$$

  There are other methods to generate variates Normal from distributions Uniform. However, we have to avoid methods with divergent branching / looping (which automatically rules out Ziggurat (Marsaglia and Tsang [2000]) and methods of rejection sampling) in order to allow distributed computation and the use of hashing. There is also an improved version of BOX MULLER transform, which is called the Polar Method (Marsaglia and Bray [1964]), but it is equivalent to a technique of rejection sampling, and therefore Box Muller is the best possible option.

- S is a random matrix diagonal with i.i.d. chi random numbers.

S's entries are drawn from a chi distribution with d degrees of freedom and are multiplied by the FROBENIUS norm of matrix G. The vector matrix diagonal product is done in the same way as G by the use of intrinsic functions.

To generate the chi distribution using hashes, we could have different approaches. The first one would be to generate it using the definition, by summing squares of GAUSSIAN random variables as follows.

$$\chi_d^2 = \sum_{c=1}^{d} X_c^2 \tag{8.5}$$

where X_c are standard random variables Normal $N(0,1)$.

A better choice would be to use the BOX MULLER Transform approximation by Gaussians and generate it from distribution Uniform as

$$\chi_d^2 = -2\log(U_1 U_2 \dots U_\nu) = \sum_{c=1}^{\nu} -2\log(U_c) \tag{8.6}$$

where U_c are random variables Uniform $U(0,1)$ and ν is equal to $\frac{d}{2}$, being d a power of 2.

However, these two methods are computationally costly. McKernel relies on an asymptotic approximation by Wilson and Hilferty [1931].

$$\chi_d^2 = d\left(\sqrt{\frac{2}{9d}}z + \left(1 - \frac{2}{9d}\right)\right)^3 \tag{8.7}$$

where z is a distribution standard Normal $N(0,1)$.

This transformation is based on the fact that the cubic root of χ_d^2/d follows closely a distribution standard Normal using

$$z = \frac{\left(\frac{\chi_d^2}{d}\right)^{\frac{1}{3}} - \left(1 - \frac{2}{9d}\right)}{\sqrt{\frac{2}{9d}}}. \tag{8.8}$$

8.2 Pseudo-random Numbers by Hashing

The idea of hashing comes up because of the problem to speed up searching in large arrays of data. Imagine you have to search a huge array for a given value, if it is not sorted, the search may require examining each element. If the array is sorted, we could use binary search with speed $O(\log n)$. However, we can do it even faster, suppose we have a function which maps an index to the given value, with this, the search would be reduced to just one try ($O(1)$). This function is called a function of hashing.

Functions of hashing that are good have the property that small changes in the input give large changes in the output. For our purposes, we are using a non cryptographic function of hashing called Murmurhash (Appleby [2012]). It is lighting fast and gives good results. Companies such as Google, Facebook or Amazon use this function of hashing for problems such as locality sensitive hashing.

8.2.1 Pseudo-random Numbers Mersenne Twister

A generator of Pseudo-random numbers is an algorithm for generating a sequence of numbers whose properties approximate the properties of sequences of random numbers.

There are a lot of Pseudo-random Numbers that we could leverage for this purpose, the most famous being the built in rand() function which implements a Linear Congruential generator. In the CPU single machine version of the code, we use Mersenne Twister, because of its superior statistical properties and considerably fast implementation.

8.2.2 Pseudo-random Numbers Distributed by Hashing

Many algorithms require generators of random numbers to work. For instance, Fastfood needs to generate three different random matrices diagonal and a matrix of permutation. The problem with this is that if we want to compute the operation in many places, we need to distribute these matrices to several machines.

In McKernel instead, we simply generate our Pseudo-random Numbers by hashing and recompute the entries at each machine.

Why not distribute the seed of a conventional generator of Pseudo-random Numbers and use hashes? You could do that, indeed. However, you have to be entirely sure that nowhere in your code you are using random numbers at the same time. Moreover, you have to take care that various versions of your code have the same generation of random numbers. This means that your code would be very difficult to parallelize and highly dependent on the version of the library of each machine.

8.3 How to Use the Library?

McKernel can be embedded in front of your linear classifier. This means you just need to do extraction of features, then McKernel and finally go through a linear classifier, such as SVM Linear or a logistic regression.

Chapter 9

Features Fastfood for Recognition of Ethnicity

The idea is to use McKernel before the linear classifier of our system of estimation of ethnicity and see how it performs.

Fastfood is defined as

$$\phi_c(x) = n^{-\frac{1}{2}} \exp(i[Vx]_c) \tag{9.1}$$

which is an approximation of RBF Kernel (Le et al. [2013]).

In the implementation, we consider the real version of the mapping of complex features ϕ as in Rahimi and Recht [2007]

$$\begin{aligned} \phi'_{2c-1}(x) &= n^{-\frac{1}{2}} \cos([Vx]_c) & (9.2) \\ \phi'_{2c}(x) &= n^{-\frac{1}{2}} \sin([Vx]_c). & (9.3) \end{aligned}$$

We wire the features McKernel into LIBSVM library as described in Figure 9.1.

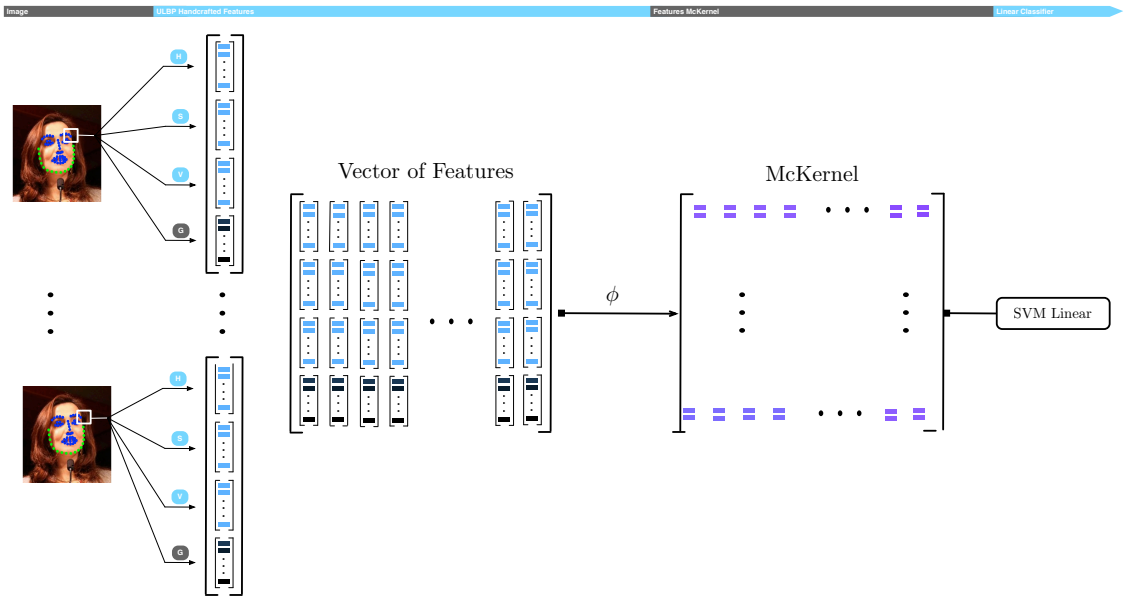


FIGURE 9.1: Estimation of Ethnicity with Features McKernel.

The performance is increased consistently by an average 2% by just wiring McKernel in front of the linear classifier.

Chapter 10

Neural Networks and Deep Learning

Deep Learning and Neural Networks have emerged as a popular field since the appearance of some astonishing results, for example in Le et al. [2011]. Companies such as Google (with Hinton) or Facebook (with LeCun) have turned its head into Deep Learning, at the expectancy of a revolution. The Lisa Laboratory led by Bengio, University of Toronto with Hinton and New York with LeCun, among many others such as Freitas at Oxford, are currently the pioneers in the field.

In this work we expect to give a humble introduction to Neural Networks and Deep Learning. A two hidden layer Deep Neural Network has been implemented with McKernel. We will provide the theoretical background needed to understand the deep network and derive some of the basic formulation.

I want to give thanks to Larochelle for his excellent NN course.

10.1 Introduction to Neural Networks

Neural Networks are based on the idea of artificial neurons. This will be the basic building block to construct more complicated structures. A neuron is a computational unit which is characterized by a pre-activation and an activation.

A pre-activation, which we can also be referred as an input activation is

$$a(\mathbf{x}) = b + \sum_c w_c x_c = b + \mathbf{w}^T \mathbf{x} \tag{10.1}$$

and an activation, which is also referred as output activation is

$$h(\mathbf{x}) = g(a(\mathbf{x})) = g(b + \sum_c w_c x_c) \tag{10.2}$$

where $\mathbf{w}$ can be seen as the connecting weights, b the neuron bias and $g()$ the activation function.

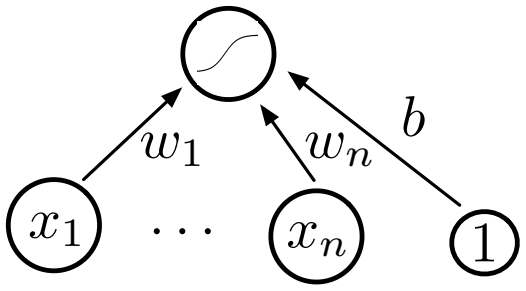


FIGURE 10.1: A Simple Neural Network.

There are many possible activation functions, but the most commonly used is the function sigmoid

$$g(a) = \text{sigmoid}(a) = \frac{1}{1 + e^{-a}}. \tag{10.3}$$

The idea behind this function is that it takes the neuron pre-activation and maps it into the interval 0 and 1, being always positive, bounded and strictly increasing.

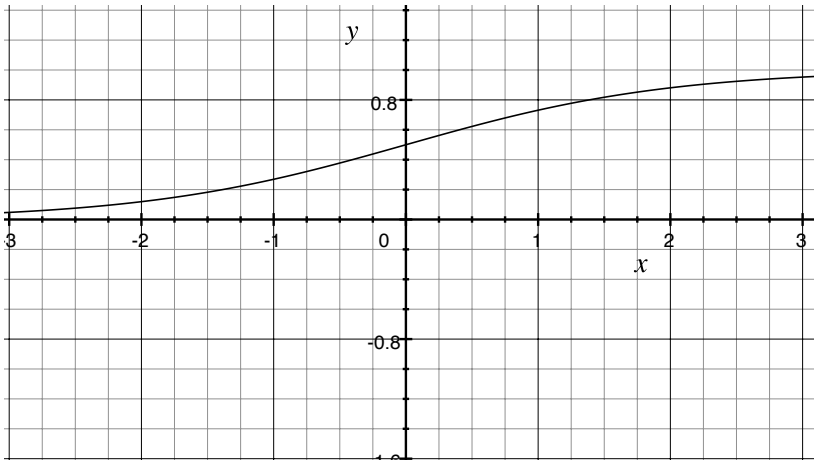


FIGURE 10.2: Activation Function Sigmoid.

Other well known activation functions are the Rectified Linear Unit (see Glorot et al. [2011] for further study) or the hyperbolic tangent. The rectified linear activation function (see Figure 10.3) is bounded below by 0 but not upper bounded. It is an always increasing function.

$$g(a) = \text{relu}(a) = \max(0, a). \tag{10.4}$$

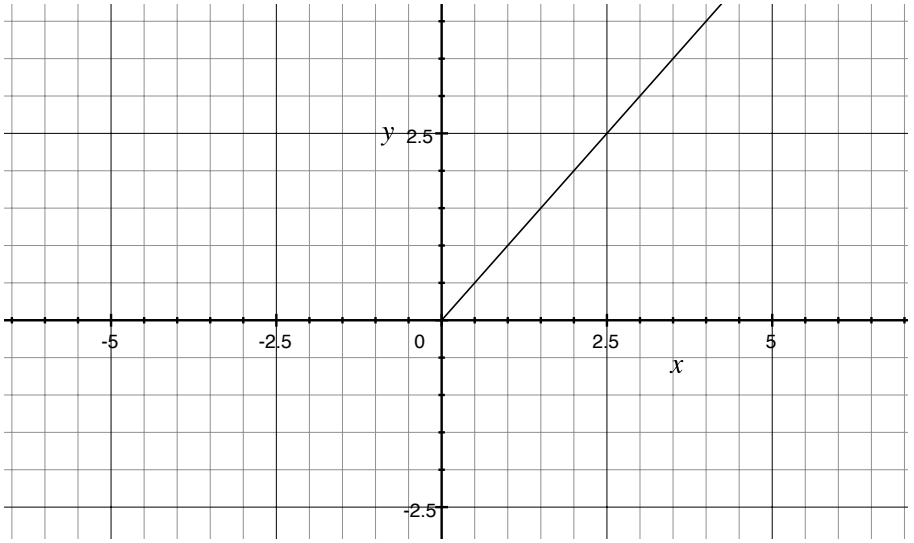


FIGURE 10.3: Rectified Linear Unit.

Using the activation function sigmoid, it is possible to understand an artificial neuron as an estimator of $p(y = 1|\mathbf{x})$, which is also known as a logistic regression classifier. It works like this: if the output is greater than 0.5 the logistic regression outputs class 1,

otherwise it outputs class 0. The decision boundary is a linear function of $\mathbf{x}$.

Using a single neuron one can solve linearly separable problems, but not non-linear problems, unless some transformation is done.

Let's see now a single hidden layer neural network. First we have the hidden layer pre-activation

$$\mathbf{a}(\mathbf{x}) = \mathbf{b}^{(1)} + \mathbf{W}^{(1)}\mathbf{x} \tag{10.5}$$

then we have the hidden layer activation:

$$\mathbf{h}(\mathbf{x}) = g(\mathbf{a}(\mathbf{x})) \tag{10.6}$$

and finally the output layer activation:

$$\mathbf{f}(\mathbf{x}) = \mathbf{o}(\mathbf{b}^{(2)} + (\mathbf{w}^{(2)^T}\mathbf{h}^{(1)}(\mathbf{x}))). \tag{10.7}$$

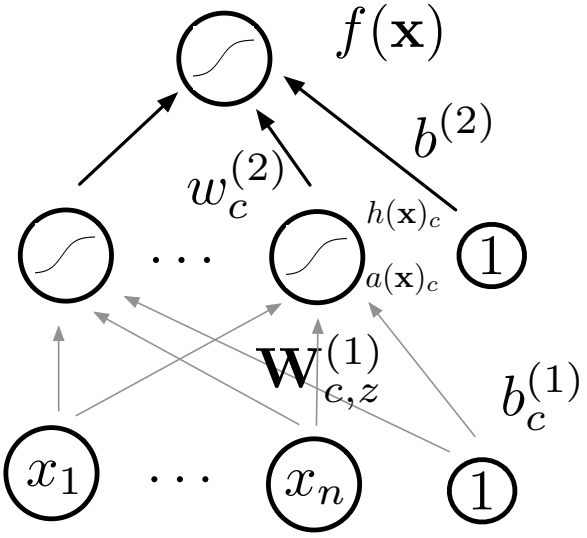


FIGURE 10.4: Single-layer Network.

What happens if we do have to do multi-class classification? For multi-class classification we need multiple outputs and we would like to estimate the conditional probability $p(y = c|\mathbf{x})$. To do that we can use the softmax activation function at the output

$$\mathbf{o}(\boldsymbol{\alpha}) = \text{softmax}(\boldsymbol{\alpha}) = \left[\frac{e^{a_1}}{\sum_c e^{a_c}} \cdots \frac{e^{a_c}}{\sum_c e^{a_c}}\right]^T \tag{10.8}$$

which is strictly positive and adds up to one. The class that will be given as output by the softmax classifier is the one that has highest probability.

Now we are able to see in detail a multilayer neural network. In a network with L hidden layers, the layer pre-activation can be formulated as follows

$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x}) \tag{10.9}$$

where $\mathbf{h}^{(k-1)}(\mathbf{x}) = \mathbf{x}$.

The hidden layer activation is then

$$\mathbf{h}^{(k)}(\mathbf{x}) = g(\mathbf{a}^k(\mathbf{x})) \tag{10.10}$$

and finally the output layer can be written as

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x}). \tag{10.11}$$

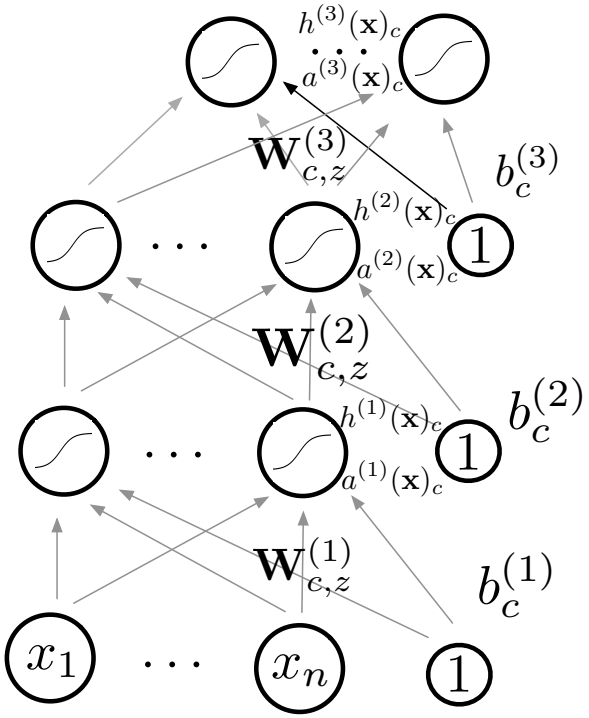


FIGURE 10.5: Multilayer Network.

The Theorem in Hornik et al. [1989] states the following observation, which is the foundation for DL research.

Theorem 10.1 (Universal Approximation)**.** *A single hidden layer neural network with linear output unit can approximate any continuous function arbitrary well, given enough hidden units.*

This result applies for the most popular activation functions: sigmoid, tanh and others. It is a really good result, but it doesn't mean we are actually able to find an algorithm that is able to get the parameters that make it possible. However, it theoretically backs up Neural Networks.

10.1.1 Biological Inspiration

Neural Networks are inspired in how the human brain works. The main idea is that neurons connect through synapses, transmit the information through a cable called axon and process the information in their cell body (called soma). Information from other neurons is collected through their dendrites.

Artificial Neural Networks try to model our neural behavior by first identifying simple forms, like edges and corners, then transform to intermediate forms and groups of features till they get to a high level description of an object, for example a word or face.

10.2 Empirical Risk Minimization

Risk minimization is a framework to design algorithms of learning. The general problem can be stated as follows

$$\arg\min_{\theta} \frac{1}{T} \sum_{t} l(f(\mathbf{x}^{(t);\theta}), y^{(t)}) + \lambda\Omega(\theta) \tag{10.12}$$

where $l(f(\mathbf{x}^{(t)}; y^{(t)}))$ is the loss function and $\Omega(\theta)$ is a regularizer term.

The main idea is that learning is cast as a problem of optimization. However, ideally we would like to optimize the classification error, but the thing here is that it is not a smooth function. Therefore, the loss function is a substitute, for instance, an upper bound, for what we truly want to optimize.

10.2.1 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is an algorithm that performs updates after each example (these kind of algorithms are known as online algorithms). It works as follows.

First we initialize θ, that is the parameters

$$\theta = \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\} \tag{10.13}$$

and then for all iterations we perform the following procedure:

for each $(\mathbf{x}^{(t)}, y^{(t)})$ we update θ as

$$\Delta = -\nabla_{\theta} l(f(\mathbf{x}^{(t)}; \theta), y^{(t)}) - \lambda \nabla_{\Omega} \Omega(\theta) \tag{10.14}$$

$$\theta = \theta + \alpha\Delta. \tag{10.15}$$

In order to apply SGD to Neural Networks we need the loss function $l(f(\mathbf{x}^{(t)};\theta), y^{(t)})$, a procedure to compute its gradient, the regularizer $\Omega(\theta)$ and its gradient and an initialization method.

10.2.2 Loss Function

A Neural Network estimates $f(\mathbf{x})_c = p(y = c|\mathbf{x})$. Therefore, we can try to maximize the probabilities of $y^{(t)}$ given $x^{(t)}$.

To cast the problem as a minimization, we minimize the negative log-likelihood as

$$l(\mathbf{f}(\mathbf{x}), y) = -\sum_c 1_{(y=c)} \log f(\mathbf{x})_c = -\log f(\mathbf{x})_y \tag{10.16}$$

we use the logarithm for mathematical ease.

10.2.2.1 Computing the Gradient

First we will take a look at the gradient at the output layer. We can now compute the partial derivative as

$$\frac{\partial}{\partial f(\mathbf{x})_c} - \log f(\mathbf{x})_y = \frac{-1_{(y=c)}}{f(\mathbf{x})_y}. \tag{10.17}$$

Therefore the gradient is

$$\nabla_{f(\mathbf{x})} - \log f(\mathbf{x})_y = \frac{-1}{f(x)_y} \begin{pmatrix} 1_{(y=0)} \\ \vdots \\ 1_{(y=c-1)} \end{pmatrix}, \tag{10.18}$$

$$= \frac{-\mathbf{e}(y)}{f(x)_y}. \tag{10.19}$$

Now, let's continue with the pre-activation partial derivative

$$
\begin{aligned}
& \frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} - \log f(\mathbf{x})_y = \\
= \quad & \frac{-1}{f(\mathbf{x})_y} \frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} f(\mathbf{x})_y, \\
= \quad & \frac{-1}{f(\mathbf{x})_y} \frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} \text{softmax}(\mathbf{a}^{(L+1)}(\mathbf{x}))_y, \\
= \quad & \frac{-1}{f(\mathbf{x})_y} \frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} \frac{e^{a^{(L+1)}(\mathbf{x})_y}}{\sum_{c'} e^{a^{(L+1)}(\mathbf{x})_{c'}}}, \\
= \quad & \frac{-1}{f(\mathbf{x})_y} \frac{\frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} e^{a^{(L+1)}(\mathbf{x})_y}}{\sum_{c'}, e^{a^{(L+1)}(\mathbf{x})_{c'}}} - \frac{e^{a^{(L+1)}(\mathbf{x})_y} \left(\frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} \sum_{c'} e^{a^{(L+1)}(\mathbf{x})_{c'}} \right)}{(\sum_{c'} e^{a^{(L+1)}(\mathbf{x})_{c'}})^2} \\
= \quad & \frac{-1}{f(\mathbf{x})_y} \left(\frac{1_{y=c} e^{a^{(L+1)}(\mathbf{x})_y}}{\sum_{c'} e^{a^{(L+1)}(\mathbf{x})_{c'}}} - \frac{e^{a^{(L+1)}(\mathbf{x})_y}}{\sum_{c'} e^{a^{(L+1)}(\mathbf{x})_{c'}}} \frac{e^{a^{(L+1)}(\mathbf{x})_y}}{\sum_{c'} e^{a^{(L+1)}(\mathbf{x})_{c'}}} \right), \\
= \quad & \frac{-1}{f(\mathbf{x})_y} \left(1_{(y=c)} \text{softmax}(\mathbf{a}^{(L+1)}(\mathbf{x}))_y - \text{softmax}(\mathbf{a}^{(L+1)}(\mathbf{x}))_y \text{softmax}(\mathbf{a}^{(L+1)}(\mathbf{x}))_c \right), \\
= \quad & \frac{-1}{f(\mathbf{x})_y} \left(1_{(y=c)} f(\mathbf{x})_y - f(\mathbf{x})_y f(\mathbf{x})_c \right), \\
= \quad & \frac{-1}{f(\mathbf{x})_y} \left(1_{(y=c)} - f(\mathbf{x})_c \right).
\end{aligned}
$$

Therefore the gradient becomes

$$\nabla_{\mathbf{a}^{(L+1)}(\mathbf{x})} - \log f(\mathbf{x})_y = -(\mathbf{e}(y) - \mathbf{f}(\mathbf{x})). \tag{10.20}$$

If we try to compute now the gradient at the hidden layers, we will need to use the chain rule

$$\frac{\partial l(h)}{\partial h} = \sum_c \frac{\partial l(h)}{\partial a_c(h)} \frac{\partial a_c(h)}{\partial h} \tag{10.21}$$

being h a unit layer, $l(h)$ the loss function and a_c the pre-activation in the layer above.

Therefore,

$$\frac{\partial}{\partial h^{(k)}(\mathbf{x})_z} - \log f(\mathbf{x})_y = \sum_c \frac{\partial - \log f(\mathbf{x})}{\partial a^{(k+1)}(\mathbf{x})_c} \frac{\partial a^{(k+1)}(\mathbf{x})_c}{\partial h^k(\mathbf{x})_z}, \tag{10.22}$$

$$= \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k+1)}(\mathbf{x})_c} W^{(k+1)}_{c,z}, \tag{10.23}$$

$$= (\mathbf{W}^{(k+1)}_{.,z})^T (\nabla \mathbf{a}^{(k+1)}(\mathbf{x}) - \log f(\mathbf{x})_y) \tag{10.24}$$

where

$$a^{(k)}(\mathbf{x})_c = b^{(k)}_c + \sum_z \mathbf{W}^{(k)}_{c,z} h^{(k-1)}(\mathbf{x})_z. \tag{10.25}$$

Therefore, the gradient becomes

$$\nabla_{\mathbf{h}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y = (\mathbf{W}^{(k+1)})^T (\nabla \mathbf{a}^{(k+1)}(\mathbf{x}) - \log f(\mathbf{x})_y). \tag{10.26}$$

To compute the gradient hidden layer pre-activation we compute the partial derivative

$$\frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k+1)}(\mathbf{x})_c} = \frac{\partial - \log f(\mathbf{x})_y}{\partial h^{(k)}(\mathbf{x})_z} \frac{\partial h^{(k)}(\mathbf{x})_z}{\partial a^{(k)}(\mathbf{x})_z}, \tag{10.27}$$

$$= \frac{-\log f(\mathbf{x})_y}{\partial h^{(k)}(\mathbf{x})_z} g'(a^{(k+1)}(\mathbf{x})_c) \tag{10.28}$$

where

$$h^{(k)}(\mathbf{x})_z = g(a^{(k)}(\mathbf{x})_z) \tag{10.29}$$

and then the gradient can be written as follows

$$\nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y = (\nabla_{\mathbf{h}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y)^T \nabla_{\mathbf{a}^{(k)}} \mathbf{h}^k(\mathbf{x}), \tag{10.30}$$

$$= (\nabla_{\mathbf{h}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y)^T \cdot [\ldots, g'(a^{(k)}(\mathbf{x})_z), \ldots] \tag{10.31}$$

where $\cdot$ denotes product elementwise.

If we denote $g(a)$ to be the activation function sigmoid $g(a) = \text{sigmoid}(a) = \frac{1}{1+e^{-a}}$, then we can use the following trick to compute its derivative

$$g'(a) = g(a)(1 - g(a)). \tag{10.32}$$

Let's now compute the gradient of the parameters

$$\frac{\partial}{\partial W_{c,z}^{(k)}} - \log f(\mathbf{x})_y = \frac{\partial - \log f(\mathbf{x})_y}{\partial_{a^{(k)}}(\mathbf{x})_c} \frac{\partial a^{(k)}(\mathbf{x})_c}{\partial W_{c,z}^{(k)}}, \tag{10.33}$$

$$= \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k)}(\mathbf{x})_c} h_z^{(k-1)}(\mathbf{x}). \tag{10.34}$$

Then, the gradient of the parameters becomes

$$\nabla_{\mathbf{W}^k} - \log f(\mathbf{x})_y = (\nabla \mathbf{a}^k(\mathbf{x}) - \log f(\mathbf{x})_y)\mathbf{h}^{(k-1)}(\mathbf{x})^T. \tag{10.35}$$

Let's now compute it for the biases

$$\frac{\partial}{\partial b_c^{(k)}} - \log f(\mathbf{x})_y = \frac{\partial - \log f(\mathbf{x})_y}{\partial_{a^{(k)}}(\mathbf{x})_c} \frac{\partial a^{(k)}(\mathbf{x})_c}{\partial b_c^{(k)}}, \tag{10.36}$$

$$= \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k)}(\mathbf{x})_c}. \tag{10.37}$$

Therefore the gradient becomes

$$\nabla_{\mathbf{b}^{(k)}} - \log f(\mathbf{x})_y = \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y. \tag{10.38}$$

Further details about how to apply SGD can be found in Bengio [2012] and Bottou [2012].

10.2.3 Backpropagation

Backpropagation has been studied thoroughly, you can see a further discussion in LeCun et al. [1998].

Algorithm of Backpropagation assumes a forward propagation has been made before.

The algorithm can be described as follows.

First compute the gradient at the output layer

$$\nabla \mathbf{a}^{(L+1)}(\mathbf{x}) - \log f(\mathbf{x})_y = -(\mathbf{e}(y) - \mathbf{f}(\mathbf{x})). \tag{10.39}$$

Then for k from $L+1$ to 1:

- Compute gradients of hidden layer parameter:

$$\nabla_{\mathbf{W}^{(k)}} - \log f(\mathbf{x})_y = (\nabla \mathbf{a}^{(k)}(\mathbf{x}) - \log f(\mathbf{x})_y) h^{(k-1)}(\mathbf{x})^T \quad (10.40)$$

$$\nabla_{\mathbf{b}^{(k)}} - \log f(\mathbf{x})_y = \nabla_{\mathbf{a}^{(k)}}(\mathbf{x}) - \log f(\mathbf{x})_y. \quad (10.41)$$

- Then compute gradient of hidden layer below:

$$\nabla \mathbf{h}^{(k-1)}(\mathbf{x}) - \log f(\mathbf{x})_y = \mathbf{W}^{(k)T}(\nabla \mathbf{a}^{(k)}(\mathbf{x}) - \log f(\mathbf{x})_y). \quad (10.42)$$

- Then compute gradient of hidden layer below pre-activation as follows

$$\nabla \mathbf{a}^{(k-1)}(\mathbf{x}) - \log f(\mathbf{x})_y = \quad (10.43)$$

$$= (\nabla \mathbf{h}^{(k-1)}(\mathbf{x}) - \log f(\mathbf{x})_y) \cdot [\ldots, g'(a^{(k-1)}(\mathbf{x})_z), \ldots] \quad (10.44)$$

  where $\cdot$ denotes product elementwise here.

How to do the computation?

We can represent forward propagation by the use of a flow graph.

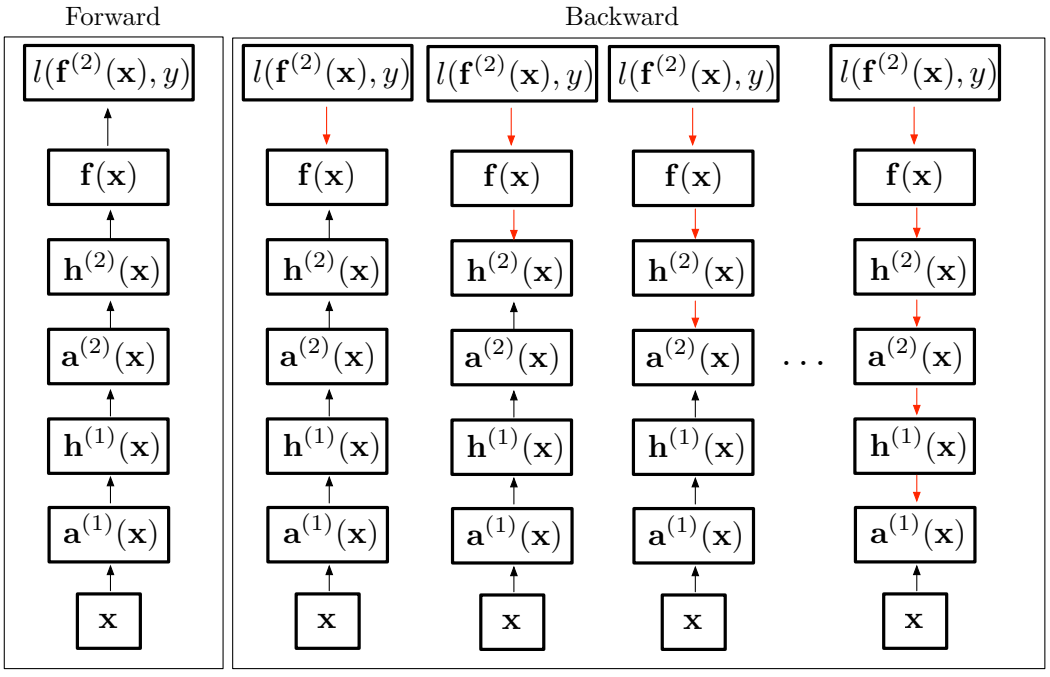


FIGURE 10.6: Forward and Backward Propagation.

Each node has a function of forward propagation which depends on the function of the children. We can compute backpropagation, using a function backpropagation in each node and calling it in the reverse order.

How to check if the gradient is well computed?

We can use a finite difference approximation of the gradient

$$\frac{\partial l(W)}{\partial W} \approx \frac{l(W+\epsilon) - l(W-\epsilon)}{2\epsilon} \tag{10.45}$$

where $l(W)$ represents the loss and W the parameter.

10.2.3.1 Regularization

If we use L2 regularization

$$\Omega(\theta) = \sum_k \sum_c \sum_z (W_{c,z}^{(k)})^2 = \sum_k ||W^{(k)}||_F^2 \tag{10.46}$$

its gradient becomes $\nabla_{\mathbf{W}^{(k)}} \Omega(\theta) = 2\mathbf{W}^{(k)}$.

Note that the regularization is just applied on weights, not on biases and can be interpreted as having a GAUSSIAN prior over the weights.

If we use L1 regularization

$$\Omega(\theta) = \sum_k \sum_c \sum_z |W_{c,z}^{(k)}| \tag{10.47}$$

its gradient is $\nabla_{\mathbf{W}^{(k)}} \Omega(\theta) = \text{sign}(\mathbf{W}^{(k)})$. It can be interpreted as having a prior Laplacian over the weights. As a difference with L2 norm, L1 will result in sparse weight vectors allowing weights with value zero.

10.3 Initializing the Network

For the biases we can just set all of them to zero.

For the weights, do not initialize all weights to the same value to break symmetry. You can sample $\mathbf{W}_{c,z}^{(k)}$ from $U[-b, b]$ where

$$b = \frac{\sqrt{6}}{\sqrt{\text{size}(h^{(k)}(\mathbf{x})) + \text{size}(h^{(k-1)}(\mathbf{x}))}} \tag{10.48}$$

from Glorot and Bengio [2010], the idea is to sample around 0 to break the symmetry.

10.3.1 Optimization

In a Neural Network, there is not a single global optimum, so it means the problem is non-convex. Therefore, optimization can get stuck in local minima.

There are many learning rate decreasing strategies, for example, these are two of the most useful

$$\alpha_t = \frac{\alpha}{1 + \delta t} \tag{10.49}$$

$$\alpha_t = \frac{\alpha}{t^\delta} \text{ with } 0.5 < \delta \leq 1. \tag{10.50}$$

10.4 Autoencoders

Autoencoders are algorithms of unsupervised learning: they rely only on the inputs $\mathbf{x}^{(t)}$ for learning and extracting meaningful features of the data. Restricted Boltzmann Machines and sparse coding models are also examples of unsupervised Neural Networks. They can diminish the need of labeled data and add a data dependent regularizer to training.

The main idea is that autoencoders are feed forward neural networks trained to reproduce its input at the output layer.

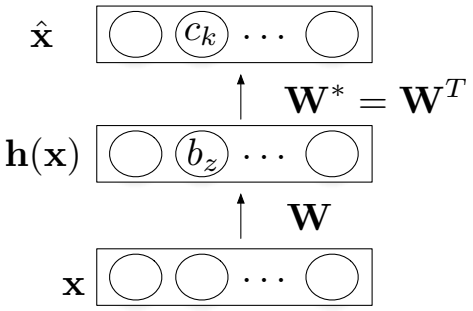


FIGURE 10.7: Autoencoder.

The decoder can be written as

$$\hat{\mathbf{x}} = o(\hat{\mathbf{a}}(\mathbf{x})) = \text{sigmoid}(\mathbf{c} + \mathbf{W}^*\mathbf{h}(\mathbf{x})) \tag{10.51}$$

and the encoder as

$$\hat{\mathbf{x}} = g(\hat{\mathbf{a}}(\mathbf{x})) = \text{sigmoid}(\mathbf{b} + \mathbf{W}\mathbf{h}(\mathbf{x})). \tag{10.52}$$

The loss function for binary inputs can be formulated as

$$l(f(\mathbf{x})) = -\sum_k (x_k \log(\hat{x_k})) + (1 - x_k) \log(1 - \hat{x_k}) \tag{10.53}$$

which can be viewed as the cross-entropy, or a sum of Bernoulli cross entropies.

The idea of an autoencoder is to make the input equal to the output and extract features in the hidden layer $f(\mathbf{x}) \equiv \hat{\mathbf{x}}$.

For real valued inputs, the loss function is

$$l(f(\mathbf{x})) = \frac{1}{2} \sum_k (\hat{\mathbf{x}_k} - x_k)^2 \tag{10.54}$$

which is the sum of squared differences (euclidean distance). We use a linear activation function at the output.

For both cases, the gradient is

$$\nabla_{\hat{\mathbf{a}}(\mathbf{x}^{(t)})} l(f(\mathbf{x}^{(t)})) = \hat{\mathbf{x}}^{(t)} - \mathbf{x}^{(t)}. \tag{10.55}$$

Parameter gradients are obtained by backpropagating the gradient.

In our code, we use sparse autoencoders, where a sparse constraint is used to make the average number of activation functions close to zero.

10.5 Deep Learning

DL research is based on models of learning with multiple layers, such as multilayer feed forward neural networks or multilayer graphical models (for instance, a deep belief network, e.g. Hinton and Osindero [2006]). For a general overview on the subject see Larochelle et al. [2009].

The main question that could arise, is why deep networks are difficult to train? This question could have two possible answers. First, optimization of the deep network is very hard. If this is the problem then we will need to find a better algorithm of optimization. Second, that our model overfits (in other words, it has high variance and low bias, which means that if the training data changes, the model could change a lot), then the solution is to use better regularization, using for example unsupervised learning. There are more advanced techniques that can be used, for instance, dropout, but it is beyond the scope of this introduction.

So, for now, we can use unsupervised pre-training, the idea is to use the network to extract the hidden structure (patterns in the data) of the input.

The idea is to use a greedy, layer by layer, approach, training one layer at a time, from the first to the last, with unsupervised criterion. Then fix the parameters of previous hidden layers, being this previous layers viewed as extracted features. This procedure can be understood by the next figure.

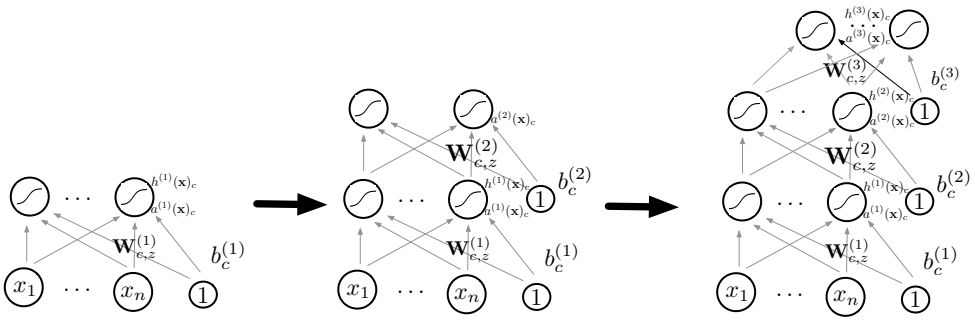


FIGURE 10.8: Pretraining.

The idea behind pretraining is to initialize the parameters in a region such that we do not fall into a near local optima and so overfit less the data. See Larochelle et al. [2009] for more information.

10.5.1 Fine-tuning

Once all layers are pretrained using the latter procedure, we add the output layer, and train the whole network using supervised learning. In other words, we are going to use the labels now.

Supervised learning is performed as in a regular feed-forward network:
First forward propagation, then backpropagation and finally update.

We can call this procedure fine-tuning, because all the parameters are adjusted to be more discriminative.

10.5.2 Pseudocode

- For $l = 1$ to L:
  - Build unsupervised training set $\mathcal{D}$.
  - Train autoencoder on $\mathcal{D}$.
  - Use hidden layer weights and biases of autoencoder to initialize the deep network parameters $\mathbf{W}^{(l)}$, $\mathbf{b}^{(l)}$.
- Initialize $\mathbf{W}^{(L+1)}$, $\mathbf{b}^{(L+1)}$ randomly.

- Train the whole Neural Network using stochastic gradient descend with backpropagation.

10.6 McKernel in the Deep Network

We have done a general introduction to Deep Learning and Neural Networks. The code is explained in more detail in de Zarza i Cubero et al. [2014], where a more rigorous explanation is done. The implementation builds on this theory and is based on a two-layer stacked autoencoder, with sparse autoencoders as hidden layers.

We have embedded McKernel in a Neural Network as non-linear mapping to the activation function. We have obtained improved performance of 3% just by embedding this kernel expansion.

Chapter 11

Conclusions and Further Work

In this chapter we summarize the main contribution of the current project and highlight possible new research directions.

11.1 Contributions

- SIMD FWH implementation faster than current state-of-the-art methods.
- Library McKernel for fast kernel expansions in log-linear time.
- Implementation of a simple DL architecture to test our library for approximating kernel expansions, it will be the ground for future research.
- Implementation from scratch of a system for estimation of ethnicity, including the tools to crawl massively images from the net.

11.2 What's Next?

We will continue doing experiments with kernel expansions Fastfood inside DL architectures. Here we have just explored conventional feed forward neural networks with DL structure. The idea is to continue and extend these experiments to even more complex settings.

Bibliography

T. Ahonen, A. Hadid, and M. Pietikainen. 2011. Face Recognition with Local Binary Patterns. *TPAMI* (2011). 11

A. Appleby. 2012. Murmurhash. (2012). https://code.google.com/p/smhasher/ 35

Y. Bengio. 2012. Practical recommendations for gradient-based training of deep architectures. *Neural networks: Tricks of the trade. Springer* (2012). 49

L. Bottou. 2012. Stochastic Gradient Descent Tricks. *Neural networks: Tricks of the trade. Springer* (2012). 49

G. E. P. Box and M. E. Muller. 1958. A Note on the Generation of Random Normal Deviates. *The Annals of Mathematical Statistics* (1958). 33

C. C. Chang and C. J. Lin. 2011. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology* (2011). 11

I. de Zarza i Cubero, A. Smola, and C. W. Ngo. 2014. A Library for Fast Kernel Expansions with Applications to Computer Vision and Deep Learning. *City University of Hong Kong* (2014). 1, 31, 56

X. Glorot and Y. Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. *AISTATS* (2010). 52

X. Glorot, A. Bordes, and Y. Bengio. 2011. Deep Sparse Rectifier Neural Networks. *AISTATS* (2011). 41

G. E. Hinton and S. Osindero. 2006. A fast learning algorithm for deep belief nets. *Neural Computation* (2006). 54

K. Hornik, M. Stinchcombe, and H. White. 1989. Multilayer feedforward networks are universal approximators. *Neural Networks* (1989). 44

J. Johnson and M. Püschel. 2000. In search of the optimal Walsh-Hadamard Transform. *IEEE International Conference on Acoustics, Speech, and Signal Processing* (2000). 3, 31

H. Larochelle, Y. Bengio, J. Louradour, and P. Lamblin. 2009. Exploring Strategies for Training Deep Neural Networks. *JMLR* (2009). 54, 55

Q. Le, T. Sarlós, and A. Smola. 2013. Fastfood - Approximating Kernel Expansions in Loglinear Time. *ICML* (2013). 3, 21, 31, 37

Q. V. Le, R. Monga, M. Devin, G. Corrado, K. Chen, M. Ranzato, J. Dean, and A. Y. Ng. 2011. Building high-level features using large scale unsupervised learning. *ICML* (2011). 39

Y. LeCun, L. Bottou, G. Orr, and K. Müller. 1998. Efficient BackProp. *Neural networks: Tricks of the trade. Springer* (1998). 49

G. Marsaglia and T. A. Bray. 1964. A Convenient Method for Generating Normal Variables. *SIAM Rev.* (1964). 33

G. Marsaglia and W. W. Tsang. 2000. The Ziggurat Method for Generating Random Variables. *Journal of Statistical Software* (2000). 33

A. Rahimi and B. Recht. 2007. Random Features for Large-Scale Kernel Machines. *NIPS* (2007). 3, 20, 37

B. Schölkopf and A. Smola. 2002. *Learning with Kernels*. MIT Press. 17

L. Song. 2008. Learning via Hilbert Space Embedding of Distributions. *University of Sydney* (2008). 17

I. Steinwart and A. Christmann. 2008. *Support Vector Machines*. Springer. 19

X. Tan and B. Triggs. 2007. Enhanced Local Texture Feature Sets for Face Recognition under Difficult Lighting Conditions. *AMFG - 3rd International Workshop Analysis and Modelling of Faces and Gestures* (2007). 6, 9, 10

E. B. Wilson and M. M. Hilferty. 1931. The Distribution of Chi-Squared. *Proceedings of the National Academy of Sciences of the United States of America* (1931). 34

X. Xiong and F. Torre. 2013. Supervised Descent Method and its Application to Face Alignment. *CVPR* (2013). 5, 8, 9

J. de Curtó i Díaz

General information

E-mail: c@decurto.tw
Webpage: https://www.decurto.tw

Career

Carnegie Mellon. Pittsburgh.
Research Associate I. June 2014 - August 2014.
School of Computer Science. Robotics.

Education

Carnegie Mellon. Pittsburgh.

Master of Science (exchange). May 2014 - February 2015.
School of Computer Science. ML Department and Robotics.

Thesis: A Library for Fast Kernel Expansions with Applications to Computer Vision and Deep Learning.

City University of Hong Kong. Hong Kong.

Master of Science. September 2013 - February 2015.
Department of Electrical Engineering.
GPA: 4.12 (0-4.3 scale).
Classification of Award: Distinction (ranked 1st in the program).

Academic Distinctions:

- Top Achiever 2015.
  Award for being the first with regard to academic performance.
- MS Internship Sponsorship 2014.
  Award for top performing students. Robotics. Carnegie Mellon. Pittsburgh.
- MS Entrance Scholarship 2013/2014.
  This award is given to the 3 most outstanding students.

Universitat Autònoma de Barcelona. Cerdanyola del Vallès (Barcelona).

5-year Degree in Engineering of Telecommunication, Second Cycle. 2011 - 2013.
Specialization in Communications, Signal Processing and Microwave Engineering.
School of Engineering.

Thesis: Construction and Performance of Network Codes.
Grade: Excellent. First Class with Distinction.

Universitat Politècnica de Catalunya (UPC). Barcelona.

5-year Degree in Engineering of Telecommunication, First Cycle. 2006 - 2009.

University Entrance Examination.
Average Grade: 9.22/10. First Class with Distinction.

Academic Distinctions:

- First year scholarship for university studies. Ministry of Education.
  This award is given to the top nationwide first year university students.
- First year scholarship for university studies. Caixa Manresa.
  This award is given to the top university entrance examination average grades in the region of Catalunya.
- University scholarship for an outstanding academic performance. Technological Baccalaureate. Government of Salou. This award is given to the top students in each graduation year by the local authority.

Technological Baccalaureate. 2004 - 2006.
Average Grade: 9.7/10. First Class Degree and Honorary Scholarship.

Academic Distinctions:

- Outstanding Thesis of Research. Development and Design of a Virtual Shop in Visual Basic on the .NET Framework.
- Outstanding Curriculum.

Publications

Curtó, Zarza, Torre, King and Lyu.
High-resolution Deep Convolutional Generative Adversarial Networks.
https://www.curto.hk/c/hdcgan.pdf

Curtó, Zarza, Yang, Smola, Torre, Ngo and Gool.
McKernel: A Library for Approximate Kernel Expansions in Log-linear Time.
https://www.curto.hk/c/mckernel.pdf

De Curtó i Díaz, De Zarza i Cubero and Vázquez.
Secure Network Coding: Overview and State-of-the-art.
Universitat Autònoma de Barcelona. Cerdanyola del Vallès (Barcelona). 2012.
https://blogs.uab.cat/curto/files/2019/05/nc_decurto12.pdf

De Curtó i Díaz, Moreno, Torrellas, Bofill and Muñoz.
Dear New Student: A Comparison between a Frontal and an Active Approach.
ALE 2007. Toulouse.

Dissertations

Master of Science.
A Library for Fast Kernel Expansions with Applications to Computer Vision and Deep Learning.
Supervisors: Ngo and Smola.
Carnegie Mellon. Pittsburgh. 2014.
https://www.curto.hk/c/decurto.pdf
https://www.curto.hk/c/slides_decurto.pdf

5-year Degree in Engineering of Telecommunication.
Construction and Performance of Network Codes.
Supervisor: Vázquez.
Universitat Autònoma de Barcelona. Bellaterra. 2013.
https://blogs.uab.cat/curto/files/2019/05/pfc_decurto.pdf
https://blogs.uab.cat/curto/files/2019/05/slides_pfc_decurto.pdf

Languages

English -

TOEFL Internet Based test. 26-11-2016. **Score 114/120.**

First Certificate in English. December 2005. **Grade: A.**

Work Experience

CELLS ALBA Synchrotron Facility. Cerdanyola del Vallès (Barcelona).
Research Scientist. April 2010 - June 2010.

CELLS ALBA Synchrotron Facility. Cerdanyola del Vallès (Barcelona).
Internship. January 2010 - February 2010.

Universitat Politècnica de Catalunya (UPC). Barcelona.
Teaching Assistant. Departament de Teoria del Senyal i Comunicacions. September 2009 - December 2009.

CELLS ALBA Synchrotron Facility. Cerdanyola del Vallès (Barcelona).
Internship. July 2009 - September 2009.

Universitat Politècnica de Catalunya (UPC). Barcelona.
Teaching Assistant. Departament d'Arquitectura de Computadors. 2008 - 2009.

Universitat Politècnica de Catalunya (UPC). Barcelona.
Teaching Assistant. Departament d'Arquitectura de Computadors. 2006 - 2007.

Services

First European Training School in Network Coding: Random Network Coding and Designs over $GF(q)$. IEEE Information Theory Society. Universitat Autònoma de Barcelona. Cerdanyola del Vallès (Barcelona). 4 - 8 February 2013.

From designs over $GF(q)$ to applications of networking: a cross-road for mathematics, computer science and engineering.

Attendee and Volunteer.

ESOF 2008. Barcelona. 18 - 22 July 2008.

Scientific Volunteer.

Extracurricular Activities

Program of Open Mentoring. Department of Computer Science. The University of Hong Kong. 2014 - 2018.

MS Class Representative, cohort 2013. Department of Electrical Engineering. City University of Hong Kong. 2013 - 2014.

Course in Investment and Financial Markets. Technical Analysis and Risk Management. Barcelona. 23 May 2011 - 26 May 2011.

Course in Investment and Financial Markets. Barcelona. 18 April 2011 - 21 April 2011.

Competition of Entrepreneurship. EMPRÈN UPC. 1st Edition. Universitat Politècnica de Catalunya (UPC). Finalist project awarded with honorable mention and 1000 euros. Barcelona. 14 March 2011 - 14 June 2011.

Programming

C, C++, Java, MATLAB, Python, HTML, VHDL and Assembly.

Software

LaTeX, Maple, PSpice and ADS.

DE CURTÓ I DÍAZ Joaquim.
Pittsburgh, 2014.