City University of Hong Kong

Master Thesis

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

Author:
H. C. Zarza
zarza.2@my.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 Multimedia Information Technology

in the

Department of Electronic Engineering
City University of Hong Kong

November 2014

Carnegie Mellon
Pittsburgh, 2014

Carnegie Mellon

"A journey of a thousand miles begins with one small step"
Laozi

CITY UNIVERSITY OF HONG KONG

Abstract

Master of Science in Multimedia Information Technology

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

by H. C. Zarza

This thesis provides the first open implementation of the kernel expansion approximation Fastfood. The code is optimized for fast CPU intensive numerical computation. Vectorized code with intrinsic functions Intel are used. Its main contribution is a SIMD implementation of the Fast Walsh Hadamard that performs better than the current state-of-the-art. Moreover, applications to Computer Vision and Deep Learning are enclosed with practical hints on Machine Learning.

Declaration of Authorship

I, DE ZARZA I CUBERO Irene, 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: Zarza

Date: 22 November 2014

Contents

List of Figures

List of Tables

Dedicated to my family for their constant support and unconditional love

Chapter 1

Foreword

This thesis was developed as an exchange student at Carnegie Mellon, ML Department and Robotics. I had the once in a lifetime opportunity to carry out research in one of the best CS Departments in the world. Being able to learn the tricks of the trade from the ML magicians of today has been the most entangling experience in my life.

I hope you enjoy reading this thesis as much as I have enjoyed working in Machine Learning and Computer Vision. What follows next is the description of a six-month adventure.

Chapter 2

Introduction

This dissertation thesis is part of a joint work developed with J. D. Curtó, [1] and this thesis are complementary explanations of the same project.

The work is divided in the following chapters:

- **Chapter 1**: Brief foreword.
- **Chapter 2**: Gives a brief description of the work done in each chapter.
- **Chapter 3**: Describes the basic theory behind kernels, support vector machines and also the algorithms that we need to know in order to understand the theory behind the structure of deep learning used in chapter 8.
- **Chapter 4**: Implements from scratch a whole system of classification. Downloading the images to build a dataset, then using online crowdsourcing resources to get the labels, engineering handcrafted features, doing classification with state-of-the-art libraries and finally crossvalidating the parameters to get the best results.
- **Chapter 5**: Introduces randomized features: a solution to perform non-linear classification using a linear classifier and a previous mapping. In particular, we present all necessary theory to explain an approximating kernel expansions able to be computed in loglinear time, Fastfood [2].
- **Chapter 6**: Introduces the first open implementation of Fastfood: Library McKernel. Explaining vectorized code optimization procedures and implementation details. A SIMD implementation of the Fast Walsh Hadamard that performs better than current state-of-the-art methods is provided.

- **Chapter 7**: Defines and shows the results of applying features Fastfood in the application of computer vision built in chapter 4: classification of ethnicity.
- **Chapter 8**: Gives a brief introduction to neural networks and deep learning and describes a deep network with embedded Fastfood.
- **Chapter 9**: Summarizes the contributions of this thesis and the work in progress.

2.1 How to Read this Thesis?

The dissertation is written with the ML practical user in mind. Theoretical background has been provided when necessary, but long mathematical discussions have been avoided. It is intended to give practical implementation hints putting emphasis on the conceptual understanding.

Chapter 3

Background: Theory Behind the Algorithms

In this chapter we give a brief introduction to some concepts that we will need throughout the thesis.

3.1 Supervised Learning

In the framework of supervised learning, the algorithm uses the labeled training data to produce a function, which will be used to classify new examples. The idea of supervised learning is to train your algorithm using well labeled examples so that is able to generate a model and generalize to new observations.

This idea contrasts with unsupervised learning, where the algorithm is given non labeled data and its goal is to find hidden patterns and extract useful features of the given data.

Supervised learning has proven to be successful in many applications, achieving state-of-the-art performance, however, the need of huge amounts of labeled data is one of its drawbacks. We will see in the first chapters of the thesis a supervised application in computer vision while in the chapters on deep learning, we will also see how unsupervised learning with autoencoders can be used to leverage the usage of labeled data.

3.2 Learning with Kernels

We will introduce two basic concepts: Kernel machines and Support Vector Machines. This section is based on [3].

3.2.1 Kernels

A linear classifier can be expressed as $\sum_c \alpha_c \langle \mathbf{x}, \mathbf{x}' \rangle$, which follows the common trend in machine learning that many algorithms can be expressed in terms of an inner product between observation $\mathbf{x}$ and $\mathbf{x}'$. This can also be extended to inner products between observation matrices, $\mathbf{X}\mathbf{X}'$. The key idea arising from this observation is the kernel trick [4], which can be formulated as follows, whenever inner products are used, they can be replaced by kernel functions. The kernel trick converts a linear algorithm into a non-linear one by using kernels. The non-linear mapping of the kernel transforms observations in low dimensions space into elements in a high dimensional space. Then, we can separate the data linearly in this higher dimensional space being this equivalent to a non-linear separation in the original space. While different kernel functions can be used to incorporate different non-linear behaviors, the ML algorithm remains unchanged. This reusability made kernel methods popular in a wide range of applications.

3.2.2 Support Vector Machines (SVM)

Support Vector Machines try to solve the following problem: given a set of observations $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m\}$, for example being $\mathbf{x}_c$ a set of face images, and $\mathbf{Y} = \{\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_m\}$, being its associated labels, for example if the face image is male or female. The task to solve is to learn a classifier $f : \mathbf{X} \rightarrow \mathbf{Y}$ that is able to give you the labels of a new observation. A SVM linear uses a hyperplane to separate the data linearly, $f(\mathbf{x}) = sign(\langle \mathbf{w}, \mathbf{x} \rangle + b)$. The main goal of the SVM is to separate the data ensuring that the margin is maximized, i.e that the minimum distance between the hyperplane and the observations is maximized.

This can be expressed as a problem of CONVEX optimization:

$$\max_{\mathbf{w},b,\gamma} \gamma \tag{3.1}$$

$$\text{s.t } \forall c, y_c(\langle \mathbf{w}, \mathbf{x} \rangle + b) \geq \gamma \text{ and } ||\mathbf{w}||^2 = 1 \tag{3.2}$$

We can express the dual problem with the Lagrangian as follows

$$L = -\gamma - \sum_c \alpha_c[y_c(\langle \mathbf{w}, \mathbf{x} \rangle + b) - \gamma] - \lambda(||\mathbf{w}||^2 - 1), \forall \alpha_c \geq 0, \lambda \geq 0 \tag{3.3}$$

If we formulate the dual problem and differentiate the primal variables $\mathbf{w}$, b, γ and set the derivative to zero

$$\frac{\partial L}{\partial \mathbf{w}} = -\sum_c \alpha_c y_c \mathbf{x}_c - 2\lambda \mathbf{w} = 0 \tag{3.4}$$

$$\frac{\partial L}{\partial b} = -\sum_c \alpha_c y_c = 0 \tag{3.5}$$

$$\frac{\partial L}{\partial \gamma} = -1 + \sum_c \alpha_c = 0 \tag{3.6}$$

If we substitute those results back in equation 3.3 we get

$$\min_\alpha \sum_c \alpha_c \alpha_z y_c y_z \langle \mathbf{x}_c, \mathbf{x}_z \rangle \tag{3.7}$$

$$s.t. \sum_c y_c \alpha_c = 0, \sum_c \alpha_c = 1, \forall \alpha_c \geq 0 \tag{3.8}$$

This is a CONVEX problem as we can express the dual objective in matrix format as $\boldsymbol{\alpha}^T \mathbf{A} \boldsymbol{\alpha}$, where $\boldsymbol{\alpha} = (\alpha_1, \cdots, \alpha_m)^T$ and $\mathbf{A}_{cz} = y_c y_z \langle \mathbf{x}_c, \mathbf{x}_z \rangle$. Where matrix $\mathbf{A}$ is positive semi-definite and equal to the Hessian of the optimization problem, furthermore all the constraints are affines. Hence we are in front of a problem of CONVEX optimization.

We can see that in the dual problem the input just shows up in the objective function in the form a dot product and not depends on $\mathbf{w}$. This provides a basis for the kernel trick: replace the inner products with a non-linear mapping between observations $k(\mathbf{x}, \mathbf{x}') = \langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle$. In this case $\mathbf{A}$ can be expressed as $\mathbf{A}_{cz} = y_c y_z k(\mathbf{x}_c, \mathbf{x}_z)$. As long as $k(\mathbf{x}_c, \mathbf{x}_z)$ maintains the convexity of $\mathbf{A}$, we can replace by any kernel function that satisfies $\mathbf{A}$ is positive definite.

Using kernels the SVM decision rule can be expressed as $f(\mathbf{x}) = \langle \mathbf{w}, \phi(\mathbf{x}) \rangle + b$.

3.3 Learning the Basics

In this section we summarize some key concepts used in Chapter 8.

3.3.1 Gradient Descent

This first-order optimization algorithm to determine a local minimum of a function f relies on the fact that $-\nabla f(x^{(k)})$ is a descent direction. The algorithm is defined as follows:

$$x^{(k+1)} = x^{(k)} - \nu^{(k)} \nabla f(x^{(k)}) \tag{3.9}$$

The main drawback is the parameter of control $\nu^{(k)} > 0$, which for too small values it will converge very slowly and for too large values it will cause the algorithm to overshoot the minima and diverge.

3.3.2 Logistic Regression

Let $\{(x^{(1)}, y^{(1)}), \cdots, (x^{(m)}, y^{(m)})\}$ the training set. The labels are $y^{(c)} \in \{0, 1\}$ and the hypotheses function can be written as follows:

$$h_\theta(x) = \frac{1}{1 + \exp(-\theta^T x)} \tag{3.10}$$

and the model parameters θ are trained to minimized the cost function

$$J(\theta) = -\frac{1}{m} \left[\sum_{c=1}^{m} y^{(c)} \log h_\theta(x^{(c)}) + (1 - y^{(c)}) \log(1 - h_\theta(x^{(c)})) \right] \tag{3.11}$$

3.3.3 Softmax Regression

The labels are $y^{(c)} \in \{1, 2, \cdots, k\}$. This model generalizes logistic regression to classification problems, multiclass classification. In other words, the label y can take more than two possible values, for example, in the classification problem of ethnicity (see Chapter 3) there are four different possible values ($k = 4$), one for each given class; caucasian

($y = 1$), east asian ($y = 2$), south asian ($y = 3$) and african american ($y = 4$).

The hypotheses function can be written as follows:

$$h_\theta(x^{(c)}) = \begin{bmatrix} p(y^{(c)} = 1|x^{(c)};\theta) \\ p(y^{(c)} = 2|x^{(c)};\theta) \\ \vdots \\ p(y^{(c)} = k|x^{(c)};\theta) \end{bmatrix} = \frac{1}{\sum_{z=1}^{k} \exp(\theta_z^T x^{(c)})} \begin{bmatrix} \exp(\theta_1^T x^{(c)}) \\ \exp(\theta_2^T x^{(c)}) \\ \vdots \\ \exp(\theta_k^T x^{(c)}) \end{bmatrix} \tag{3.12}$$

Given a test input x, we want that out hypothesis function estimates the probability $p(y = z|x)$, $\forall z = 1, \cdots, k$. Therefore, the hypothesis function output will be a k dimensional vector, which elements sum one, giving the k estimating probabilities. Let $\frac{1}{\sum_{z=1}^{k} \exp(\theta_z^T x^{(c)})}$ be the normalized factor. Moreover, $\theta_h \in \mathbb{R}^{n+1}$ where $h = 1, \cdots, k$ are the model parameters that train to minimized the cost function:

$$J(\theta) = -\frac{1}{m} \left[\sum_{c=1}^{m} \sum_{z=1}^{k} 1\{y^{(c)} = z\} \log \frac{\exp(\theta_z^T x^{(c)})}{\sum_{l=1}^{k} \exp(\theta_l^T x^{(c)})} \right] \tag{3.13}$$

Which is generalization of the logistic regression cost function in equation 3.11, which can be rewritten as follows:

$$J(\theta) = -\frac{1}{m} \left[\sum_{c=1}^{m} \sum_{z=0}^{1} 1\{y^{(c)} = z\} \log p(y^{(c)} = z|x^{(c)};\theta) \right] \tag{3.14}$$

Chapter 4

Preliminary Work

This chapter represents an introduction to Computer Vision: it will be nice to see the difference between working on a handcrafted design throughout the chapter, which is how conventional CV feature engineering has been done, and how deep learning tries to avoid this with unsupervised learning in chapter 8.

A face recognition system for classification of ethnicity is built from scratch. This work was done in collaboration with the Robotics, at the HS Laboratory. Four different sections divide the chapter: in the first part we download massively images from Flickr, next we create an interface in AMTurk to label a subset of these images. In the third section we extract features and in the fourth section we finally do classification.

4.1 Building our Own Dataset: Exploiting Flickr

There are a lot of public datasets for CV research purposes so, why do we want to create our own? The main reason is because they are pose and illumination controlled as well as poorly balanced; which means that if you want to classify a dataset using gender and the 95% of the images are from males your accuracy in testing will be biased. The second reason is for an academic purpose, just for the sake of learning how to build it and challenge myself to do so.

4.1.1 Description of the Software

The code design has been split into two steps: the first one has been designed in Python to download the metadata information from the images into a file. The second step is

done in Matlab. It uses the output file generated in the previous step to retrieve the image from API Flickr.

- PYTHON code downloads using API Flickr the URL address of the images.
- MATLAB code uses the output .txt file to download the images from the Flickr server avoiding being banned by letting certain time between requests. This code includes different improvements:
  - a filter to avoid noisy images, based on using negative tag words that must be not included in the image tags to be downloaded.
  - a temporal selection option to choose the temporal interval of time when the images have been taken.
  - a multiple word search option, implemented to improve the diversity of the images.

The code took, using a single server machine, two weeks to download a two million image dataset.

4.2 Getting the Labels

Next step in supervised machine learning is to obtain the image labels. To achieve this goal the initial raw dataset from Flickr (2.000.000 image dataset) will be reduced to just 14.000 images, and AMTurk will be used to allow external people label the data.

4.2.1 AMTurk

An HTML interface, where we can upload the images and allow the AMTurk workers to select the proper face attributes, has been designed. In Figure 4.1 it is shown the three step interface: the HTML has a first welcome page with all the instructions you need to follow and a help page link. Then, the second page shows the face image and all the labels to be selected as a display button. Finally, a thanks page it is shown in the third step. Further, to avoid unanswered labels, we have used some JavaScript scripts that prevent users from not completing the fields.

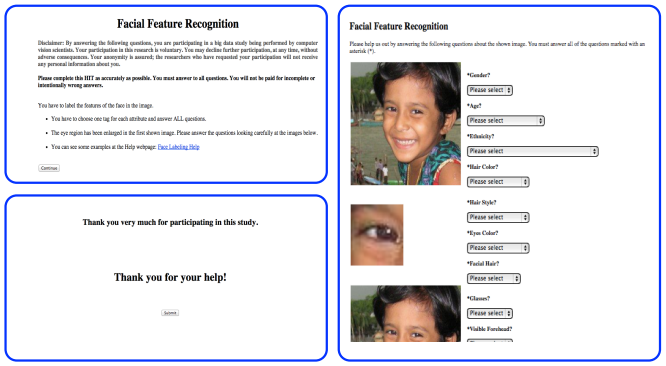

FIGURE 4.1: Interface AMTurk

Both HS Laboratory webserver and a sites from Google have been used to upload the images to AMTurk. Also, we have coded a PYTHON script to create a CSV file to save all the image URLs.

4.3 Feature Extraction

The feature extraction step in the preliminary work has been done using first a face tracker [5] to recognize 49 key points in the face, called landmark points. Then, we have applied a handcrafted Local Binary Pattern (LBP) feature in each patch being centered at a landmark point, Figure 4.3.

4.3.1 Landmarks

In Figure 4.2 we can see an example of the resulting 49 facial points after applying the face tracker to the image. Overlapping between patches will be useful in the performance of the system, this empirical observation was done also by [6].

Why are we using 49 landmark points? The face tracker in [5] contains 49 points, but no all of them provide the same information. For example, in the classification of ethnicity if we just use 14 (considering just the upper part of the face, i.e nose and eyes) points we have the 90% of the whole facial information, however if we want to achieve the best performance using the given system, empirically we see that we get better results if we use all the given points to extract features. The ones that are giving more importation, though, are the two center points in the eyes and the point on the tip of the nose. This seams logical, eyes are very important to distinguish between different ethnics and also the nose, not only by its form but also because it is a point where we have good access to skin color, which is particularly important in this problem.

4.3.2 LBP Handcrafted Features

Before feature extraction, the original images are cropped to extract the faces and processed by a normalization and an affine transformation. Then, we use Multiscale Uniform Local Binary Pattern (LBP) features in four different channels: the HSV channels (see Table 4.1) and a fourth informative channel in gray scale with preprocessing Tan and Triggs [7].

LBP features vector [8], are created in the following manner: for each pixel in the selected patch, we compute the difference between the selected pixel and its neighbors to create a threshold and obtain a binary number, then for each of the pixels in the patch we have a binary number and then we compute an histogram with these numbers, which is the feature vector.

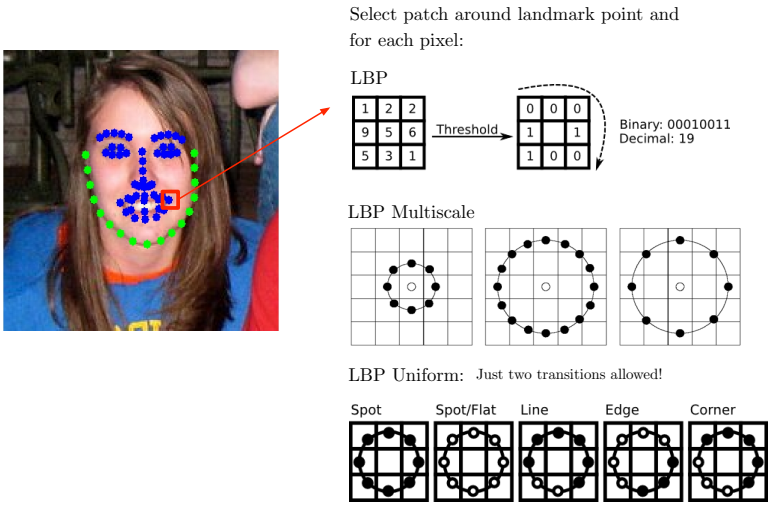

FIGURE 4.2: LBP Features

LBP Multiscale considers neighbors with a given radius instead of assuming radius one.

LBP Uniform considers only binary numbers with two transitions (from 0 to 1 and viceversa), so that all histograms have the same length for a given number of neighbors (for instance, we have a length 59 histogram vector using 8 neighbors).

4.3.3 Preprocessing

The main problems of using a real dataset are non controlled illumination and different pose. To improve the system performance under non controlled light conditions, we

introduce a preprocessing step in the system based on [7].

The preprocessing step consists on a cascade of methods based on the above mentioned paper which are: gamma correction, filtering difference of gaussian (DoG) and contrast equalization.

After applying this preprocessing step (see table 4.2), the performance of the system improves around 2%.

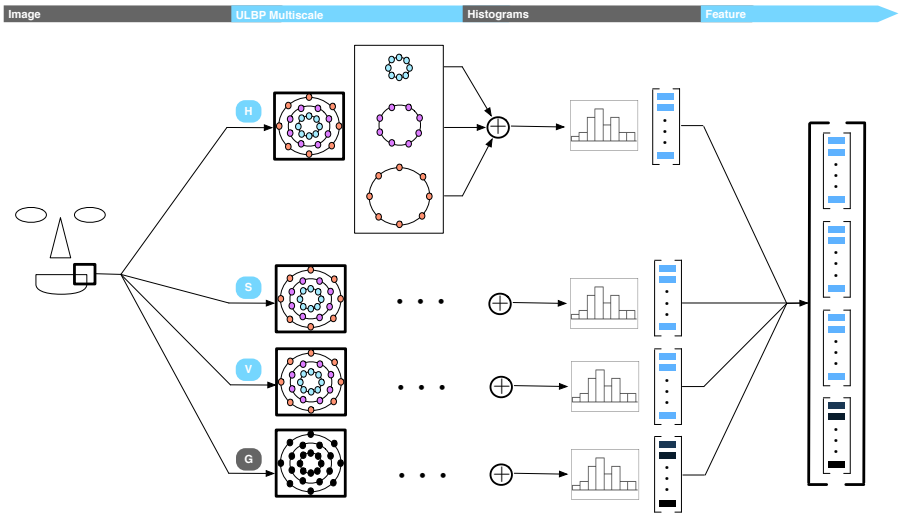

FIGURE 4.3: Feature Extraction System

4.4 Classification Stage

Different classifiers have been used in this part of the work: we started by a simple perceptron, the simplest linear classifier, then we tried different configurations and then moved to more sophisticated algorithms: AdaBoost (using MULTIBOOST Library [9]) and Support Vector Machine (using LIBSVM library [10]). SVM Linear has achieved the best performance.

4.5 System Details and Experimental Results

4.5.1 System Details

In the first step, using the original images we write the code to save the landmark coordinates in one folder and the normalized images in another one. Then, we wrote a

Linux Schell script to save the labels from AMTurk from CSV to a JSON file. Finally, we wrote a script to save the image, landmark paths and the label of each image in a CSV file.

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 output.

Finally, in the last step a MATLAB script was written to perform the crossvalidation process, Figure 4.4, calling the binary files of the C++ code and parallelizing the execution in a twelve core server. This step will help us to choose the best parameters for the LBP and the classifier.

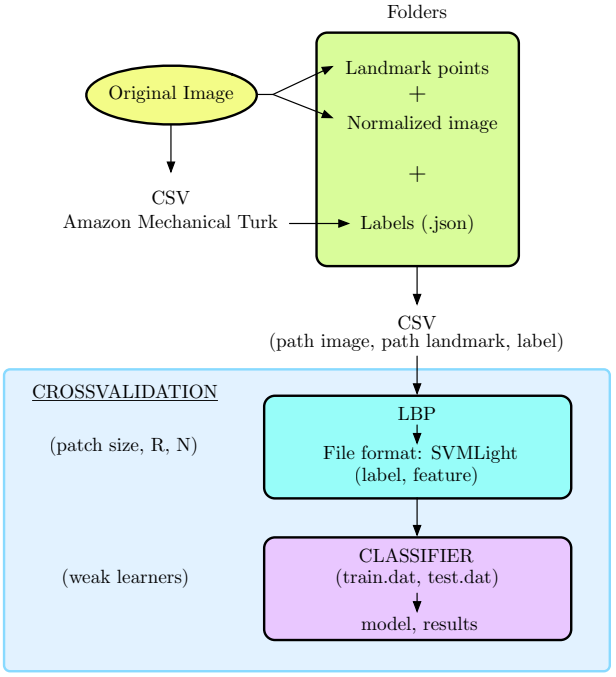

FIGURE 4.4: Classification System Structure

4.5.2 K-Fold Crossvalidation

To perform the crossvalidation step we use K-Fold Crossvalidation, a method consisting on first splitting the dataset into two, one part for training and validation and a second

for testing. Then the first part is splitted for training and validation using the following procedure, we divide the set into chucks of k elements assigning a number from 1 to k to each element. For each round you take all elements with one key, for example 2, to validate and all the other elements to train. Finally we average the results to obtain an accuracy measure. In each one of this settings, we use a different parameter configuration for the system, and we get the results for all possible combinations of the given parameters. Finally, we select the the parameters that gives better accuracy results.

Using this method we have crossvalidated the LBP parameters (patch size, number of neighbors and radius), the color space and in the AdaBoost classifier the number of weak learners.

4.5.3 Benchmarks

In this section we have used LIBSVM library [10] to perform the classification step.

All the following crossvalidation results are done using multi-scale LBP Uniform with SVM Multiclass Linear as a classifier applied to classification of ethnicity (using four classes: caucasian, african american, east asian and south asian).

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 for the radius 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 4.1: Color Space K-Fold Crossvalidation Applied to Classification of Ethnicity

Table 4.2 shows the experimental system results applying LBP Uniform (ULBP) and SVM Linear in the simulations

We start by just using plain LBP uniform with SVM and subsequently build a more complex system. We see how using Multiscale gives better results, and also that the HSV color space works quite well for estimation of ethnicity. A preprocessing to avoid illumination changes also helps considerably.

Test Explanation	Accuracy achieved (%)
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 4.2: Experimental Results System of Ethnicity

Chapter 5

Fast Kernel Expansions: Randomized Features

In this chapter random features are introduced as an alternative to the kernel trick [4]. The motivation is to face up the main drawback of SVM non-Linear, which is the high cost in term of computation in the training step. In practice it is not recommended to use SVM for a large dataset (more than 50.000 observations in the training step). Therefore, the large datasets used by Google, Amazon and Facebook for example, need a linear version of the classifier to deal with and random features becomes a good solution to this problem.

Random features [11] make supervised learning algorithms scalable so that they can be used in large-scale datasets. The idea is that traditional learning algorithms try to optimize parameters that we do not actually need to optimize. Instead, some parameters are randomized and then we optimize over the others. This recipe opens the line to good learning algorithms that can run extremely fast and are very easy to implement.

Kernel Machines are state-of-the-art architectures for classification, but training them is slow. The idea is to pass the data through random features, and train a linear algorithm on this mapped data. The random features are designed so that the classifier generated by concatenating random features with linear algorithm is the same as using a non-linear algorithm.

5.1 Random Kitchen Sinks

The proof which guarantees that kernel functions can be expressed as an inner product in some HILBERT Space is MERCER Theorem:

Theorem 5.1 (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*

$$k(x, x') = \sum_z \lambda_z \phi_z(x)\phi_z(x') \tag{5.1}$$

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

The Key idea of Rahimi and Recht [11] is to use sampling to approximate the sum in equation 5.1 as

$$\lambda_c \sim p(\lambda) \text{ where } p(\lambda_c) \propto \lambda_c \tag{5.2}$$

$$\text{and } k(x, x') \approx \frac{\sum_z \lambda_z}{n} \sum_{c=1}^{n} \phi_{\lambda_c}(x)\phi_{\lambda_c}(x') \tag{5.3}$$

5.2 Fastfood: Kernel Expansions in Log-linear Time

Fastfood [2] 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. [2] 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 GAUSSIAN random matrix we want to parameterize by:

$$V := \frac{1}{\sigma\sqrt{d}} SHG\Pi HB \tag{5.4}$$

where,

- B is a diagonal matrix with entries i.i.d. $+1$ and -1 from a uniform distribution.
- H is the Walsh Hadamard computed using FWH in Library McKernel.
- Π is the permutation matrix generated using the algorithm Fisher Yates in Library McKernel.
- G is a diagonal matrix which entries are iid and follow a normal random distribution $(0, 1)$.
- S is also a diagonal matrix 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 diagonal matrices 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 $O(n)$.

5.3 Fast Walsh Hadamard

The Walsh Hadamard (WH) H_m can be defined as a $2^m \times 2^m$ matrix scaled by a normalization factor. The more common definition of this transform is given as follows in a recursive way:

Defining the 1×1 Hadamard by the identity $H_0 = 1$, then $\forall m > 0$, H_m is defined as:

$$H_m = \frac{1}{\sqrt{2}} \begin{pmatrix} H_{m-1} & H_{m-1} \\ H_{m-1} & -H_{m-1} \end{pmatrix} \tag{5.5}$$

and for $m > 1$ we have

$$H_m = H_1 \otimes H_{m-1} \tag{5.6}$$

A WH naive implementation has a computational complexity of $O(N^2)$, so for our purpose we will use Fast Walsh Hadamard (FWH), an efficient algorithm to compute the WH in $O(N \log N)$.

FWH is a divide and conquer algorithm, the key idea is that it recursively breaks down a WH of size N into two smaller WHs of size $N/2$. See Figure 5.1.

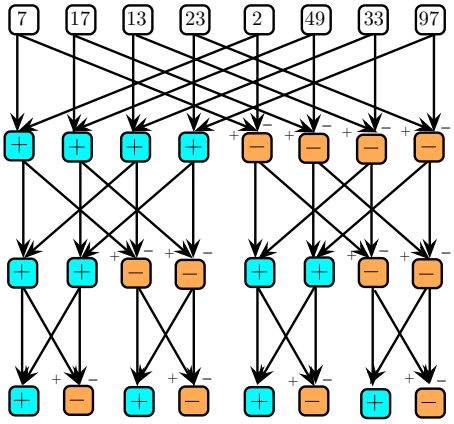

FIGURE 5.1: The Fast Walsh Hadamard Applied to a Vector of Length 8

Chapter 6

Fast Implementation of Fastfood: Library McKernel

The materials of this chapter are jointly written with J. D. Curtó in [1].

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 [2] 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 [12]. 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.

6.1 Software Description

In Random Kitchen Sinks [11] instead of computing RBF GAUSSIAN kernel

$$k(x, x') = \exp(-||x - x'||^2/(2\sigma^2)) \tag{6.1}$$

the method computes

$$k(x, x') = \exp(i[Zx]_c) \tag{6.2}$$

where z_i is drawn from a random normal distribution.

In Fastfood [2] Z is parametrized by V as

$$V := \frac{1}{\sigma\sqrt{d}} SHG\Pi HB \tag{6.3}$$

where,

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

  B's entries are generated by a binomial distribution by drawing random numbers from a uniform distribution. In the single CPU machine version, one McKernel function stores in a vector the diagonal positions with -1 entries and instead of multiplying each element it just flips the sign when necessary, reducing half of the storage and computation. In the distributed version, Pseudo-random Numbers are generated using hash functions. We can generate uniform numbers from a hash function $h(c, z)$ 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 hash function with good probability distribution.

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

  Fast Walsh Hadamard is a divide and conquer approach to solve the Walsh Hadamard. McKernel implementation is based on halving recursively the input vector and doing subsequent sums and subtractions. This idea which is based on the COOLEY TUCKEY algorithm for FFT computation can be visualized in Figure 5.1 and 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 [12]. 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 length one vector. For computational efficiency, it computes from up to bottom, and then the remaining computation is done from bottom to up, being able to maximize cache efficiency and easily being able to use a pre-existing unrolled small routines 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 would damage the performance of a fast implementation. Also, iterative algorithms are

better suited for distributed computation of large-scale data.

- Π is the permutation matrix generated using the 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 diagonal random matrix with iid normal random numbers.

  G's entries are drawn form a standard normal distribution $N(0,1)$. The vector diagonal matrix product is vectorized using SIMD intrinsics to speed up the computation. In the distributed version of the code we use BOX MULLER transform [13] to draw normal random variables from uniform variables using hashing. We generate a random normal number from two hash function values as follows:

$$P_{cz} = (-2 \log h_1(c,z)/N)^{1/2} \cos(2\pi h_2(c,z)/N) \tag{6.4}$$

  There are other methods to generate normal variates from uniform distributions. However, we have to avoid methods with divergent branching / looping (which automatically rules out Ziggurat [14] and rejection sampling methods) in order to allow distributed computation and the use of hash functions. There is also an improved version of BOX MULLER transform, which is called the Polar Method [15], but it is equivalent to a rejection sampling technique, and therefore Box Muller is the best possible option.

- S is a diagonal random matrix with iid 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 diagonal matrix 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.

$$\chi_d^2 = \sum_{c=1}^{d} X_c^2 \tag{6.5}$$

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

A better choice would be to use the BOX MULLER Transform [13] approximation by gaussians and generate it from uniform distribution as

$$\chi_d^2 = -2\log(U_1 U_2 \dots U_\nu) = \sum_{c=1}^{\nu} -2\log(U_c) \tag{6.6}$$

where U_c are uniform random variables $U(0,1)$ 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 in [16].

$$\chi_d^2 = d\left(\sqrt{\frac{2}{9d}}z + \left(1 - \frac{2}{9d}\right)\right)^3 \tag{6.7}$$

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

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

$$z = \frac{(\frac{\chi_d^2}{d})^{\frac{1}{3}} - (1 - \frac{2}{9d})}{\sqrt{\frac{2}{9d}}} \tag{6.8}$$

6.2 Optimizing Code: Vectorization Techniques

This section summarized the way we have optimized Library McKernel to speed up the code performance.

6.2.1 SIMD Intrinsic Functions

SIMD (Single Instructions Multiple Data) is a similar concept to algebraic vector operations. It is based on the fact that the CPU can perform operations in registers containing more than one floating or integer elements. This kind of operations were restricted to GPU code till the appearance of CPU vectorized instruction sets (MMX being the first and subsequent updates being SSE and AVX).

McKernel makes use of two instruction sets, SSE2 for backward compatibility and AVX (starting with Sandy Bridge processors)

6.2.2 Blocks to Vectorize

To obtain good vectorization of a for loop we need to fulfill the next criteria:

- Finite number of iterations: the number of iteration is fixed and does not rely on the input data.
- No break calls inside the loop.
- Function calls: avoid function calls unless they are inline functions or intrinsic math functions.
- Data dependency: all the iterations inside the loop must be independent.
- Contiguous memory access: consecutive addresses in memory must be loaded to a vector register with a single vector instruction (e.g _mm_load_ps() for floats) to achieve an efficient vectorization.

6.2.3 Cache-friendly Code

When one address position is retrieved from main memory, a 64KB chunk of memory is loaded into cache. This means that if you make your code to access contiguous addresses of memory, you are optimizing cache hit rates and therefore the speed of the code is improved.

McKernel uses this idea to improve cache efficiency, solving first the computation of part of the vector from top to bottom and then from bottom to top in small chunks of memory. Using this procedure, cache hits are highly optimized and the performance of

the code is boosted.

Also, recursive functions are avoided and only iterative code is used. This is because recursive functions need to put parameters into the Stack each time a call is done, and so cache locality is lost.

6.2.4 Data Alignment

Iterating over multi-dimensional data may affect alignment if the input data size is not multiple of the cache line size. In McKernel we have avoided this problem just by definition because as the input vector must be a power of 2 and the default registers of SSE and AVX have sizes 128 bits and 256 bits: all possible combinations are a multiple of 64KB (cache line):

1 float = 4 bytes
sizes of the registers in the code: In SSE2 we can store 4 floats (using 128bits register) and in the AVX we can store 8 floats (using 256bits register).

6.3 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 hash function.

Good hash functions have the property that small changes in the input give large changes in the output. For our purposes, we are using a non cryptographic hash function called Murmurhash [17]. It is lighting fast and gives good results. Companies such as Google, Facebook or Amazon use this hash function for problems such as locality sensitive hashing.

Many algorithms require random number generators to work. For instance, Fastfood generates three different random diagonal matrices and a permutation matrix. 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. Good hash functions have the property that small changes in the input give large changes in the output. For our purposes, we are using a non cryptographic hash function called Murmurhash. It is lighting fast and gives good results.

6.3.1 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.

6.3.2 Pseudo-random Numbers Distributed by Hashing

Many algorithms require random number generators to work. For instance, Fastfood needs to generate three different random diagonal matrices and a permutation matrix. 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 to distribute the seed of a conventional 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 random number generation. This means that your code would be very difficult to parallelize and highly dependent on the version of the library of each machine.

6.4 FWH Benchmarks

This section summarizes the performance experiments of McKernel.

Figure 6.1 and Table 6.1 show the performance of McKernel Fast Walsh Hadamard in comparison with the current state-of-the-art [12]. The experiments have been done using an Intel Core i5-4200 CPU 1.60 GHz machine. The results have been computed averaging the time performance of 300 float random vectors for each length.

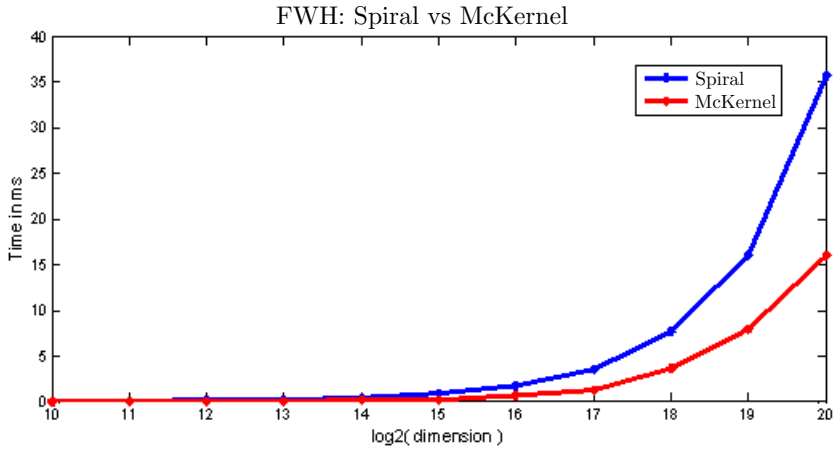

FIGURE 6.1: FWH Comparison between Spiral and McKernel

$\log_2(dimension)$	Time in ms (Spiral)	Time in ms (McKernel)
10	0.0333	0.0000
11	0.0667	0.0333
12	0.1667	0.1000
13	0.2000	0.0667
14	0.4667	0.2000
15	0.9000	0.2000
16	1.6667	0.7000
17	3.5000	1.3000
18	7.6667	3.6000
19	15.9667	7.8667
20	35.7000	15.9667

TABLE 6.1: FWH Results Comparison. Spiral and McKernel

Chapter 7

Using Features McKernel for Recognition of Ethnicity

This chapter serves as example of how McKernel can be used in a CV system to improve the performance and add the possibility to use kernel based methods in large-scale datasets with log linear computational time.

The feature map for Fastfood is defined as:

$$\phi_c(x) = n^{-\frac{1}{2}} \exp(i[Vx]_c). \tag{7.1}$$

which approximates the RBF kernel as it is proven in [2]. Since the kernel values are real numbers, in the implementation we consider the real version of the complex feature map ϕ.

In practice, based on [11] we can replace $\phi \in \mathbb{C}^n$ with $\phi'(x) \in \mathbb{R}^{2n}$, where

$$\phi'_{2c-1}(x) = n^{-\frac{1}{2}} \cos([Vx]_c) \tag{7.2}$$

$$\phi'_{2c}(x) = n^{-\frac{1}{2}} \sin([Vx]_c) \tag{7.3}$$

Therefore, in order to include Fastfood kernel expansion in a system, we need to apply the real feature map. That means we apply to the extracted features fastfood function in the Library McKernel, then compute the cosine for odd component positions and the sine for the even component positions and scale the result using the factor $n^{-\frac{1}{2}}$. Finally,

the output result is the input of a linear classifier, for instance, SVM Linear. See in Figure 7.1 an example of the role of McKernel in a system.

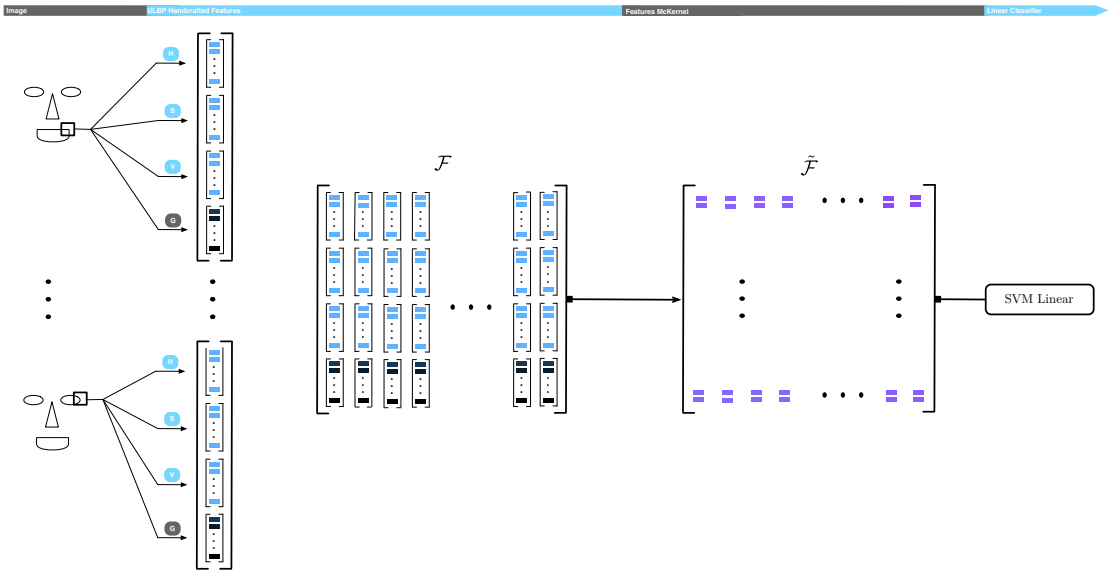

FIGURE 7.1: McKernel Embedded in the Classification System of Ethnicity

Using this configuration we get promising results, obtaining an average improvement of 2% over our original system, which means that we are improving the performance of the system just by adding McKernel before the linear classifier.

Chapter 8

Neural Networks and Deep Learning: a Deep Network Using McKernel

In this chapter we give a brief introduction to fundamental concepts in Deep Learning and Neural Networks building upon [18] and [19]. We implement a two hidden layers DL structure that serves as testing framework for kernel expansions approximation McKernel. The results show that embedding Fastfood improves the performance of the system considerably.

8.1 Deep Learning

Supervised learning has proven to give good results in almost any application of computer vision, speech recognition or artificial intelligence. However, as we have seen when we were building our dataset and constructing our own handcrafted engineered features, supervised learning is still severely limited. The fact that we had to come up with some good feature representation of ethnicity, was a slow and hand-engineered work on computer vision. Feature engineering does not scale well to new problems, i.e. what can work extremely well for ethnicity cannot be useful at all for age recognition.

The idea behind deep learning is to be able to learn features, i.e. patterns behind the data, that work better than hand engineered ones, such as our LBP handcrafted in Chapter 1.

We will study next some of the most useful algorithms applied to neural networks. Among them, the autoencoder is an unsupervised learning algorithm that is extremely useful when used in a deep network together with supervised training. Feed forward neural networks and the backpropagation algorithm will be described.

8.2 Neural Networks

Neural Networks are defined by a non-linear mapping $h(\mathbf{x})$, with parameters $\mathbf{W}$, b that we adapt to our labeled observations $(x^{(c)}, y^{(c)})$.

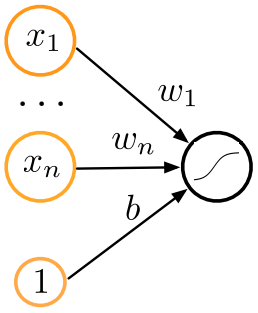

FIGURE 8.1: Example of Simple Neural Network

The simplest form of neural network, is a single neuron (see figure 8.1). This neuron takes as input x_1, x_2, x_3 and a $+1$ bias term, and outputs $h(\mathbf{x}) = g(\mathbf{W}^T\mathbf{x}) = g(\sum_{c=1}^{3} W_c x_c + b)$ where g is called an activation function. There are several famous activation functions, here we will just focus in the sigmoid

$$g(a) = \frac{1}{1 + e^{-a}} \tag{8.1}$$

Our simple neuron network with the sigmoid activation function, becomes the logistic regression mapping (see equation 3.10). There are other activation functions, for instance *tanh* or the *ReLU* function.

A property that will be useful for later derivations, is that if $g(a)$ is a sigmoid function, then its derivative can be computed by $g'(a) = g(a)(1 - g(a))$.

A neural network is constructed by putting together many simple neurons, so that the output of a neuron is the input of another one, Figure 8.2.

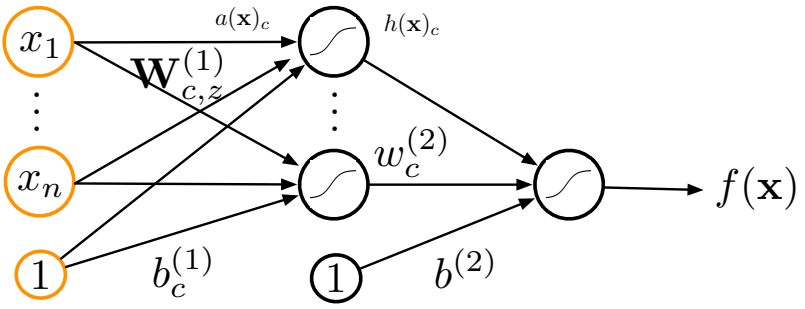

FIGURE 8.2: Example of Three-layer Neural Network

The first layer is called the input layer, the last layer is the output layer. The layers between input and output, are hidden layers. Each network node is called a unit. In this sense, in our particular example we have three input units, three hidden units and one output unit.

Our neural network has parameters $(\mathbf{W}, \mathbf{b}) = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)})$, where we write $W^{(l)}_{c,z}$ to denote the weight associated with the connection between unit z in layer $l-1$, and unit c in layer l. Also, $b^{(l)}_c$ is the bias associated with unit c in layer l.

In the example, $\mathbf{W}^{(1)} \in \mathbb{R}^{3\times 3}$, and $\mathbf{W}^{(2)} \in \mathbb{R}^{1\times 3}$.

We write $h^{(l)}_c$ to denote the output value of unit c in layer l (activation). For the first layer, we will use $h^{(0)}_c = x_i$ to denote the c-th input.

Given a fixed setting of the parameters W, b, our neural network defines a hypothesis $f(\mathbf{x})$ that outputs a real number.

$$h_1^{(1)} = g(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}) \tag{8.2}$$
$$h_2^{(1)} = g(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}) \tag{8.3}$$
$$h_3^{(1)} = g(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}) \tag{8.4}$$
$$f(\mathbf{x}) = h_1^{(2)} = g(W_{11}^{(2)}h_1^{(1)} + W_{12}^{(2)}h_2^{(1)} + W_{13}^{(2)}h_3^{(1)} + b_1^{(2)}) \tag{8.5}$$

Finally, we can set $a^{(l)}_c = \sum_{j=1}^{n} W^{(l)}_{c,z}x_z + b^{(l)}_c$, so that $h^{(l)}_c = g(a^{(l)}_c)$. Now, we can use a more compact notation, if we let the sigmoid function $g()$ apply to vectors element $(g([a_1, a_2, a_3]) = [g(a_1), g(a_2), g(a_3)])$ then we write in vector matrix notation

$$\mathbf{a}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \tag{8.6}$$

$$\mathbf{h}^{(1)} = g(\mathbf{a}^{(1)}) \tag{8.7}$$

$$\mathbf{a}^{(2)} = \mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)} \tag{8.8}$$

$$\mathbf{f}(\mathbf{x}) = \mathbf{h}^{(2)} = g(\mathbf{a}^{(2)}) \tag{8.9}$$

We can now do a more general notation, setting $\mathbf{a}^{(0)} = \mathbf{x}$. Given $\mathbf{h}^{(l)}$, we can compute $\mathbf{h}^{(l+1)}$ as

$$\mathbf{a}^{(l+1)} = \mathbf{W}^{(l+1)}\mathbf{h}^{(l)} + \mathbf{b}^{(l+1)} \tag{8.10}$$

$$\mathbf{h}^{(l+1)} = g(\mathbf{a}^{(l+1)}) \tag{8.11}$$

We have studied a simple yet complete neural network. To build a more complex one, we would just need to stack multiple hidden layers. This type of network is called a feed forward neural network.

8.3 Theory Behind the Code

This section describes the different parts that appear in the implementation.

8.3.1 Backpropagation

Let $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ be a dataset of m training samples.

We consider the loss function for real valued inputs

$$J(W, b; x^{(c)}, y^{(c)}) = l(f(x^{(c)}), y^{(c)}) = \frac{1}{2}||f(x^{(c)}) - y^{(c)}||^2 \tag{8.12}$$

then, we can define the overall cost function of the empirical risk minimization as follows

$$J(W,b) = \left[\frac{1}{m}\sum_{c=1}^{m} J(W,b;x^{(c)},y^{(c)}) + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{c=1}^{s_l}\sum_{z=1}^{s_l+1}(W_{zc}^{(l)})^2\right] \tag{8.13}$$

$$= \left[\frac{1}{m}\sum_{c=1}^{m}\left(\frac{1}{2}||f(x^{(c)}) - y^{(c)}||^2\right) + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{c=1}^{s_l}\sum_{z=1}^{s_l+1}(W_{zc}^{(l)})^2\right] \tag{8.14}$$

The first term is an average sum of squared differences between the estimated value and the given label and the second term is the regularizer term, which penalties certain parameters and helps to prevent overfitting. The λ parameter does a trade off between the importance of the two terms.

This cost function is often used for classification, where in the binary case we let $y = 0$ or $y = 1$.

We want to minimize $J(W,b)$. In order to do that, we will first initialize the parameters of the network, $W_{cz}^{(l)}$ and $b_c^{(l)}$, to a random small number and then apply gradient descent. There are other techniques different than gradient descent, but in this work we will focus on this. It is possible that it falls into local minima, however, experimentally it has a good performance.

Note that the parameters are initialized at random instead of at 0. The most important thing is not to initialize at the same exact value, because if you do, all hidden layers will end up learning the same function, so random initialization tries to break this symmetry.

Gradient Descent can be described as follows

$$\mathbf{W}^{(l)} = \mathbf{W}^{(l)} - \alpha\nabla_{\mathbf{W}^{(l)}}J(W,b) \tag{8.15}$$

$$\mathbf{b}^{(l)} = \mathbf{b}^{(l)} - \alpha\nabla_{\mathbf{b}^{(l)}}J(W,b) \tag{8.16}$$

and each partial derivative term is computed as

$$W_{cz}^{(l)} = W_{cz}^{(l)} - \alpha\frac{\partial}{\partial W_{cz}^{(l)}}J(W,b) \tag{8.17}$$

$$b_c^{(l)} = b_c^{(l)} - \alpha\frac{\partial}{\partial b_c^{(l)}}J(W,b) \tag{8.18}$$

where we can see that the updates depend on the learning rate parameter α.

In order to compute the partial derivates needed for the gradient computation, we need to use the backpropagation algorithm.

From the equation above we can see that

$$\frac{\partial}{\partial W_{cz}^{(l)}} J(W,b) = \left[\frac{1}{m}\sum_{c=1}^{m} \frac{\partial}{\partial W_{cz}^{(l)}} J(W,b;x^{(c)},y^{(c)})\right] + \lambda W_{cz}^{(l)} \tag{8.19}$$

$$\frac{\partial}{\partial b_{c}^{(l)}} J(W,b) = \left[\frac{1}{m}\sum_{c=1}^{m} \frac{\partial}{\partial b_{c}^{(l)}} J(W,b;x^{(c)},y^{(c)})\right] \tag{8.20}$$

We can see that weight decay is only applied to $\mathbf{W}$, not to $\mathbf{b}$.

The idea behind backpropagation is the next. Given (x, y) as example, we do a forward propagation to compute all activation functions $h^{(l)}(\mathbf{x})$, including the decision function $f(\mathbf{x})$. Following, for each node in the layer, we will compute how much the node is affecting the overall error output. This term $\delta_c^{(l)}$ will be computed as a weighted average of error terms.

We can see next a description of the backpropagation algorithm:

- First compute forward pass, compute activation function $h^{(l)}(\mathbf{x})$ till output layer.
- In the output layer we compute

$$\partial_c^{(n_l)} = \frac{\partial}{\partial a_c^{(n_l)}} \frac{1}{2}||y - f(x)||^2 = -(y_c - h_c^{(n_l)}) \cdot g'(a_c^{(n_l)}) \tag{8.21}$$

- For each layer $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$

  - For each node c in layer l, compute

$$\delta_c^{(l)} = \left(\sum_{z=1}^{s_l+1} W_{zc}^{(l+1)} \delta_z^{(l+1)}\right) g'(a_c^{(l)}) \tag{8.22}$$

- Compute partial derivatives as:

$$\frac{\partial}{\partial W_{cz}^{(l)}} J(W,b;x,y) = h_z^{(l-1)} \delta_c^{(l)} \tag{8.23}$$

$$\frac{\partial}{\partial b_c^{(l)}} J(W,b;x,y) = \delta_c^{(l)} \tag{8.24}$$

If we now use vector matrix notation we get

- First compute forward pass, compute activation function $\mathbf{h}^{(l)}(\mathbf{x})$ till output layer.
- In the output layer we compute

$$\delta^{(n_l)} = -(\mathbf{y} - \mathbf{h}^{(n_l)}) \cdot g'(\mathbf{a}^{(n_l)}) \tag{8.25}$$

- For nodes $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$

$$\delta^{(l)} = \left(\mathbf{W}^{(l+1)^T} \delta^{(l+1)} \right) \cdot g'(\mathbf{a}^l) \tag{8.26}$$

- Compute the desired partial derivatives, which are given as:

$$\nabla_{\mathbf{W}^{(l)}} J(W,b;x,y) = \delta^{(l)} (\mathbf{h}^{(l-1)})^T \tag{8.27}$$

$$\nabla_{\mathbf{b}^{(l)}} J(W,b;x,y) = \delta^{(l)} \tag{8.28}$$

Finally, let's sketch the gradient descent algorithm.

- Initialize $\Delta \mathbf{W}^{(l)} = 0$, $\Delta \mathbf{b}^{(l)} = 0$.
- For $c = 1$ to m
  - Use backpropagation to compute $\nabla_{\mathbf{W}^{(l)}} J(W,b;x,y)$ and $\nabla_{\mathbf{b}^{(l)}} J(W,b;x,y)$
  - Do $\Delta \mathbf{W}^{(l)} = \Delta \mathbf{W}^{(l)} + \nabla_{\mathbf{W}^{(l)}} J(W,b;x,y)$
  - Do $\Delta \mathbf{b}^{(l)} = \Delta \mathbf{b}^{(l)} + \nabla_{\mathbf{b}^{(l)}} J(W,b;x,y)$
- Update the parameters
  - $\mathbf{W}^{(l)} = \mathbf{W}^{(l)} - \alpha \left[(\frac{1}{m} \Delta \mathbf{W}^l) + \lambda \mathbf{W}^{(l)} \right]$
  - $\mathbf{b}^{(l)} = \mathbf{b}^{(l)} - \alpha \left[\frac{1}{m} \Delta \mathbf{b}^{(l)} \right]$

8.3.2 Checking Gradients

Backpropagation is a difficult algorithm to implement and is difficult to know if it is actually working correctly or not. There is a common method to check whether the derivatives are computed well or not.

Given a function $J'(W)$ that computes $\frac{\partial}{\partial W}J(W)$, we can use

$$J'(W) \approx \frac{J(W+\epsilon) - J(W-\epsilon)}{2 \times \epsilon} \tag{8.29}$$

The value of ϵ can be chosen arbitrarily small, a good practical value is 10^{-4}.

To extend this formula to vector notation, we can use the following. First we define $\mathbf{W}^{(c+)} = \mathbf{W} + \epsilon \times \vec{e}_c$ where $\vec{e}_c$ is a vector with all zeros and a 1 in the c-th component. Also, define $\mathbf{W}^{(c-)} = \mathbf{W} - \epsilon \times \vec{e}_c$.

So, if we have to approximate a function $J'_c(\mathbf{W})$ that computes $\frac{\partial}{\partial W_c}J(W)$ we can just use

$$J'_c(\mathbf{W}) = \frac{J(\mathbf{W}^{c+}) - J(\mathbf{W}^{c-})}{2 \times \epsilon} \tag{8.30}$$

8.3.3 Autoencoders

An autoencoder tries to extract the internal representation of the data by applying back propagation and setting $y_{(c)} = x_{(c)}$. The idea is to use unsupervised learning to extract meaningful features, Figure 8.3.

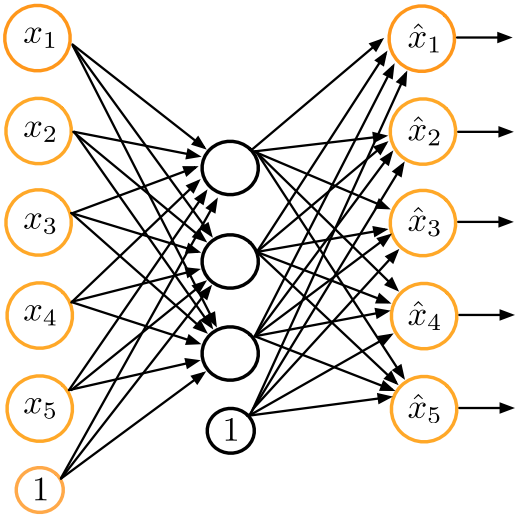

FIGURE 8.3: Autoencoder Example

An autoencoder tries to find the identity function $f(\mathbf{x}) \equiv \mathbf{x}$. By using less hidden units than input and output, we compress the information and extract patterns to represent the internal characteristics of your data. Although this is normally used with a smaller number of hidden units than input and output, it is also possible to have a larger number. For example by imposing a sparsity constraint (i.e using a sparse autoencoder).

We say a neuron is active when its output value is approximately one, whereas it is inactive when it is approximately zero. A sparsity constraint tries to impose that the majority of the neurons are inactive.

First we write $h_z^{(2)}(x)$ to denote the activation of the hidden unit when the network is given a specific input x.

We say that

$$\hat{\rho_z} = \frac{1}{m} \sum_{c=1}^{m} [h_z^{(2)}(x^{(c)})] \tag{8.31}$$

is the average activation of hidden unit z. If we impose

$$\hat{\rho_z} = \rho \tag{8.32}$$

being ρ the sparse parameter, we are trying to make the average activation of each hidden neuron z close to ρ. ρ will be normally a small value.

We can impose a new constraint to the cost function

$$\sum_{z=1}^{s_2} KL(\rho||\hat{\rho}_z) \tag{8.33}$$

where $KL(\rho||\hat{\rho}_z) = \rho \log \frac{\rho}{\hat{\rho}_z} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}}$ is the divergence Kullback-Leibler.

$KL(\rho||\hat{\rho}_z) = 0$ if $\hat{\rho}_z = \rho$, otherwise it increases monotonically.

Our sparse cost function is now

$$J_{sparse}(W,b) = J(W,b) + \beta \sum_{z=1}^{s_2} KL(\rho||\hat{\rho}_z) \tag{8.34}$$

where β gives a tradeoff between the importance of the two terms.

We can easily impose this new constraint to for example the second layer of back propagation as follows:

$$\delta_c^{(2)} = \left(\sum_{z=1}^{s_2} W_{zc}^{(3)} \delta_z^{(3)} + \beta \left(-\frac{\rho}{\hat{\rho}_c} + \frac{1-\rho}{1-\hat{\rho}_c} \right) \right) g'(a_c^{(2)}) \tag{8.35}$$

8.3.4 Stacked Autoencoders

A stacked autoencoder is a neural network consisting of multiple layers of sparse autoencoders in which the outputs of each layer is wired to the inputs of the successive layer.

Formally, consider a stacked autoencoder with n layers. The encoding step for the stacked autoencoder is given by running the encoding step of each layer in forward order:

$$h^{(l)} = g(a^{(l)}) \tag{8.36}$$

$$a^{(l+1)} = W^{l+1,1}h^{(l)} + b^{(l+1,1)} \tag{8.37}$$

We can do the decoding by doing the decoding of each autoencoder in backward order:

$$h^{(n+l)} = g(a^{(n+l)}) \tag{8.38}$$

$$a^{(n+l-1)} = W^{n+l,2}h^{(n+l)} + b^{(n+l,2)} \tag{8.39}$$

The higher order representation of the data is saved in $h^{(n)}$, which is the higher layer of hidden units.

We can use this extracted information, $h^{(n)}$, as input to a softmax classifier for classification.

Stacked autoencoders extract first patterns in the appearance of simple features, then from these features, they extract internal representations, for example which of these features appear together. As higher the layer goes, the representation of the data will be more and more complex to get high order features.

8.4 McKernel in the Deep Network and Implementation

Our implementation relies on the sparse autoencoder and the softmax regression. We will use this system to exemplify the performance of McKernel in a DL structure.

We use MNIST dataset for convenience to see the performance of the system. We are using a stacked autoencoder structure, with two layers of sparse auto encoders. We train layer by layer the autoencoders. In the first layer we input and output the raw data and we get the first activation. We use this to be input and output of the second sparse autoencoder, and extract the activation for the second hidden layer. Finally, we do exactly the same procedure for the third layer to get the parameters. After this pre training we fine tune using backpropagation, where now we are using the labels to finally tune the parameters.

With this system using for training 60000 images and for testing 10000 images we get an average accuracy of 93.79%.

8.4.1 Code Highlights

We can summarize the code implementation as follows

- We read the images from MNIST dataset
- Then implement the function to compute the cost function and gradients for the sparse autoencoder, logistic regression and overall deep network.
- Next, implement the functions to check if all gradients are well computed.
- Train the autoencoder layers and the softmax regression
- Fine-tune the network by backpropagation.

8.4.2 Where Does McKernel Fit in?

We use McKernel as a non-linear mapping of the activation function. This allows the system to extract non-linear features by using this kernel expansion approximation. When combined with a linear classifier like softmax regression, we are able to use kernel methods in a DL structure.

Using the same system as before with training 60000 images and testing 10000 images we get an average accuracy of 96.31%. Which means we have almost an average 3% improvement by just using McKernel.

These results make Fastfood really promising for extracting fast non-linear features in DL structures easily.

Chapter 9

Contributions and Further Work

We can summarize the main contributions of this thesis as follows:

- The Library McKernel which implements kernel expansions in log-linear time (Fastfood).
- A SIMD fast implementation of the Fast Walsh Hadamard, which achieves better results than current state-of-the-art implementations found in the literature.
- A Deep Network with Fastfood embedded showing the performance improvement.
- A working estimation system of ethnicity.

As further work, we will extend the current research to more complex DL architectures, using both kernel expansions and DL architectures to achieve huge performance gains.

Bibliography

[1] J. D. Curtó. A library for fast kernel expansions with applications to computer vision and deep learning. *City University of Hong Kong*, 2014.

[2] Q. Le, T. Sarlós, and A. Smola. Fastfood - approximating kernel expansions in loglinear time. *ICML*, 2013.

[3] L. Song. Learning via hilbert space embedding of distributions. *University of Sydney*, 2008.

[4] B. Schölkopf and A. Smola. *Learning with Kernels*. MIT Press, 2002.

[5] X. Xiong and F. De La Torre. Supervised descent method and its application to face alignment. *CVPR*, 2013.

[6] C. Lu, D. Zhao, and X. Tang. Face recognition using face patch networks. *ICCV*, 2013.

[7] X. Tan and B. Triggs. Enhanced local texture feature sets for face recognition under difficult lighting conditions. *AMFG - 3rd International Workshop Analysis and Modelling of Faces and Gestures*, 2007.

[8] T. Ahonen, A. Hadid, and M. Pietikainen. Face recognition with local binary patterns. *TPAMI*, 2011.

[9] D. Benbouzid, R. Busa-Fekete, N. Casagrande, F. D. Collin, and B. Kégl. MULTIBOOST: A multi-purpose boosting package. *JMLR*, 2012.

[10] C. C. Chang and C. J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2011.

[11] A. Rahimi and B. Recht. Random features for large-scale kernel machines. *NIPS*, 2007.

[12] J. Johnson and M. Püschel. In search of the optimal walsh-hadamard transform. *IEEE*, 2000.

[13] G. E. P. Box and M. E. Muller. A note on the generation of random normal deviates. *The Annals of Mathematical Statistics*, 1958.

[14] G. Marsaglia and W. W. Tsang. The ziggurat method for generating random variables. *Journal of Statistical Software*, 2000.

[15] G. Marsaglia and T. A. Bray. A convenient method for generating normal variables. *SIAM Review*, 1964.

[16] E. B. Wilson and M. M. Hilferty. The distribution of chi-squared. *Proceedings of the National Academy of Sciences of the United States of America*, 1931.

[17] A. Appleby. Murmurhash. 2012. URL https://code.google.com/p/smhasher/.

[18] A. Ng. CS294A lecture notes. 2011. URL http://web.stanford.edu/class/cs294a/sparseAutoencoder.pdf.

[19] A. Ng, J. Ngiam, C. Y. Foo, Y. Mai, and C. Suen. UFLDL tutorial. 2013. URL http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial.