Masters' Degree in Informatics Engineering

Dissertation

Final Report

Development of Support Vector Machines (SVMs) in Graphics Processing Units for Pattern Recognition

João Carlos Ferreira Gonçalves

jcgonc@student.dei.uc.pt

Advisors:

Bernardete Ribeiro

Noel Lopes

Date: August 31, 2012

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Acknowledgements

I would like to thank my supervisor, Bernardete Ribeiro, for the invaluable advice and support she has given me. I would also like to thank Noel Lopes for his continuous help in this project's development. Without their suggestions, comments and guidance this work could not have been possible.

I would like to thank my parents who have been highly supportive throughout the time it has taken me to write this report, specially my mother for her encouragement and giving me fortitude when times were difficult.

Finally, I can not forget my little sister's support. Without her graphics card, the project's development would have been slowed down. Thank you, Inês.

Sincerely grateful,
João Carlos Ferreira Gonçalves

Abstract

Machine learning aims to develop algorithms which extract relevant (and meaningful) information from existing data. Or even more exciting would be extraction of the fundamental laws that not only govern all learning processes but as the universe itself. Naturally, this appears to be a complicated task as the more advanced the algorithms are, the heavier requirements are needed from currently available hardware.

Thus, current single-threaded algorithms are often unable to scale with the demanding processing power needed. Among the supervised learning algorithms, Support Vector Machines (SVMs) are the most widely used algorithm due to their generalization properties and regularization capability. SVMs are binary large margin classifiers which have found successful applications in many scientific fields such as engineering, bio-informatics [54], information management [1], finance, business [52] among many others.

The SVM aims to find the optimal decision hyperplane which is equivalent to reach both the smallest generalization and empirical errors. By making use of the Kernel trick, the SVM shows powerful classification and regression performance in complex non-linear problems. Thus, functions which obey Mercer's theorem transform the input vectors into a highly-dimensional space and by learning a linear model in this feature space.

An important and crucial point in the SVM formulation is that it can provide a good generalization independent of the training set's distribution by using the principle of structural risk minimization [49, 11]. However, they usually require significant memory and computational burden for calculating the large Gram matrix [10]. To circumvent this limitation fast learning methods have been proposed [12, 21]. Nevertheless most implementations still do no take advantage of either the multi-core architecture of today Central Processing Units (CPUs) nor the more powerful Graphics Processing Units (GPUs).

In this work we focus on a GPU SVM classifier, developed from a previous studied Multi-Threaded parallel CPU standalone SVM version (MT-SVM)

which builds up from the scratch an implementation of the Sequential Minimal Optimization (SMO) algorithm. Although previous approaches have been developed [8], our implementation includes a new kernel function, the Universal Kernel Function (UKF) [53] which leads to a broad spectrum of the generalization capabilities of the learning machine. Experiments performed on UCI datasets benchmarks [3] yield performance competitive results as compared to state-of-the-art LIBSVM tools while delivering better speedups on large datasets.

Finally, we used our GPU implementation to study a real case problem which consists of offline signature recognition. This is a difficult problem arising in many practical applications. The results achieved for the task of detecting forged signatures are promising although more research is needed because original and faked signatures can be extremely similar. We obtained excellent results on the identification of an individual's signature despite the fact that a generic classifier configuration is complicated to achieve. Future objectives will aim to improve the multi-class learning architecture used in the process of signature identification.

Resumo

A aprendizagem máquina é uma área da ciência que onde se estuda o desenvolvimento de algoritmos cujo alvo é a extracção de informação relevante a partir de dados já existentes. Talvez ainda mais estimulante seja não só a extracção das leis fundamentais que governam os processos de aprendizagem como as que regem o próprio universo. Claro que tal aparenta ser uma tarefa complicada já que quanto mais avançados os algoritmos são, maiores são os requisitos impostos no hardware actual.

Consequentemente, os actuais algoritmos sequenciais são incapazes de escalar com os maiores requisitos a nível de poder de computação. Entre os vários algoritmos de aprendizagem supervisionados encontram-se as SVM, que são talvez o algoritmo mais usado devido às suas capacidades de generalização. As SVM são classificadores de grande margem que são utilizadas em muitas aplicações científicas como engenharia, bio-informática [54], sistemas de informação [1], finanças [52], entre outras áreas.

O objectivo das SVMs é encontrar o hiperplano óptimo de separação, sendo esta procura equivalente a minimizar em simultâneo o erro empírico e de generalização. Recorrendo ao truque do kernel, a SVM apresenta uma forte performance de classificação e regressão em problemas complicados e não lineares. Para tal são usadas funções que obedecem às condições de Mercer e que permitem transformar os atributos de entrada num espaço de dimensão superior, aplicando neste espaço transformado o hiperplano de decisão.

Um ponto importante e crucial na formulação das SVM é que estas permitem uma boa generalização, independentemente da distribuição do conjunto de dados, já que recorrem ao principio da minimização do risco estrutural [49, 11].

Contudo, as SVM incorrem em geral num grande consumo de memória e de cálculo computacional, devido ao cálculo da matriz de Gram [10]. De modo a evitar estas limitações foram propostos métodos eficazes de aprendizagem [12, 21]. Um outro factor a ter em conta é que grande parte das implementações não fazem uso quer dos processadores multi-core quer das GPUs disponíveis na actualidade.

O objectivo desta dissertação é o desenvolvimento de uma SVM recorrendo ao poder das GPUs, a partir de uma implementação multi-threaded para CPUs estudada previamente. Ambas as implementações foram construídas com base no algoritmo SMO.

Embora existam implementações prévias semelhantes [8], a que apresentamos neste trabalho inclui o novo kernel UKF [53], que possibilita à SVM um maior poder de generalização. Resultados efectuados em datasets do UCI apresentam uma performance semelhante à conhecida LIBSVM, com tempos mais rápidos quer de treino quer de classificação para grandes datasets, não esquecendo o facto que a LIBSVM é actualmente o estado da arte da investigação nas SVM.

Por fim, usámos a implementação para a GPU no estudo de um problema que consiste no reconhecimento off-line de assinaturas. Trata-se de um problema complicado com muitas aplicações práticas. Os resultados obtidos na detecção de assinaturas falsas são promissores embora estudos futuros sejam necessários, já que a distinção de assinaturas verdadeiras de forjadas é complicado, devido à grande semelhança entre si. A nível da identificação do autor de uma dada assinatura obtivemos resultados excelentes demonstrando que tal tarefa é actualmente possível e acessível. No futuro esperamos melhorar o suporte para problemas multi-classe e melhorar a arquitectura usada no processo de identificação de assinaturas.

Contents

List of Tables

List of Figures

Acronyms

AI	Artificial Intelligence
ALU	Arithmetic and Logic Unit
ANN	Artificial Neural Network
API	Application Programming Interface
CC	Compute Capability
CGA	Colour Graphics Adapter
CGI	Computer Generated Imagery
CPU	Central Processing Unit
CUDA	Compute United Device Architecture
DAC	Digital-to-Analogue-Converter
DCT	Discrete Cosine Transform
DDR	Double-Data-Rate
FDR	False Discovery Rate
FN	False Negative
FPR	False Positive Rate
FPS	First Person Shooter
FPU	Floating Point Unit
FP	False Positive
GLSL	Graphics Library Shading Language
GPGPU	General-Purpose computing on Graphics Processor Units

GPU	Graphics Processing Unit
HDR	High-Definition-Rendering
HLSL	High Level Shading Language
IDCT	inverse Discrete Cosine Transform
ILP	Instruction-Level-Parallelism
KKT	Karush-Kuhn-Tucker
LDA	Linear Discriminant Analysis
LSB	Least-Significant Bit
MCC	Matthews Correlation Coefficient
MDF	Modified Direction Feature
NN	Neural Network
OPENCL	Open Computing Language
OPENGL	Open Graphics Library
PCA	Principal Component Analysis
QP	Quadratic Programming
RAMDAC	RAM Digital-to-Analogue-Converter
RAM	Random Access Memory
RBF	Radial Basis Function
ROC	Resource Operating Characteristic
SFU	Special Function Unit
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SISD	Single Instruction Single Data
SMO	Sequential Minimal Optimization
SMP	Symmetric-Multi-Processing

SM Streaming Multiprocessor

SPMD Single Process Multiple Data

SP Shading Processor

SVM Support Vector Machine

SV Support Vector

TNR True Negative Rate

TN True Negative

TPC Thread Processing Cluster

TPR True Positive Rate

TP True Positive

UKF Universal Kernel Function

UMA Unified Memory Access

VLIW Very Long Instruction Word

VPU Video Processor Unit

VR Virtual Reality

Notation

In this section we give the mathematical notation used throughout the dissertation. Naturally, a minimum level of knowledge from the areas of calculus, algebra and probability theory is required for the interpretation of this work.

Global symbols

d	number of features
c	number of classes
n	number of samples or patterns
b	bias or offset
χ	pattern or sample set
Ω	class set

Mathematical Symbols

x	a variable (real number)
y	classifier result
c_i	class of sample i
ω_c	set of samples belonging to class c
x_i	sample vector i from the sample set
y_i	classification or training target for sample i
$\mathbf{u}$	sample $\mathbf{u}$ of the feature space
$\mathbf{v}$	sample $\mathbf{v}$ of the feature space
$\mathbf{w}$	weight vector
$\mathbf{x}$	sample or pattern
$f(x)$	function f evaluated at x
$f(\mathbf{x})$	classification f of pattern $\mathbf{x}$
$\mathbb{R}$	the set of real numbers
$\mathbb{R}^d$	the d-dimensional real set
α	a Lagrange multiplier
θ	a real number
a, b	real numbers
$\mathbf{A}$	a matrix
$\mathbf{A}^T$	transpose of matrix $\mathbf{A}$
$\mathbf{w}^T$	transpose of vector $\mathbf{w}$
σ	a real number, usually as the standard deviation variable in a normal distribution
ξ	a slack variable representing a misclassification error
$\langle u, v\rangle$	dot product between vector u and v
e	Euler's number, approximately $2.71828\cdots$
$\mathbf{\Phi}(x)$	projection of x in another space
f_i	Karush-Kuhn-Tucker optimality condition for a given sample x_i

Chapter 1

Introduction

In this chapter we present the main reasons which led to the development of this work and its objectives. Section 1.1 gives the background and context of this project, including current hardware capabilities and software requirements. Section 1.2 summarizes the goals and project planning of the dissertation. Section 1.3 lays out the organization of this report.

1.1 Motivation

The amount of data produced by humans and machines grows at an unparalleled rate, year after year. The challenge is to extract meaningful and relevant information from such data. In this context, Machine Learning systems that can extract relevant and useful information from large repositories of data are extremely important [29, 26]. However, machine learning algorithms often require high-processing capabilities and current CPUs are not able to put up with the demanding processing power needed. Hence, the pressure to shift development towards parallel architectures with high-throughput has been accentuated [29, 26].

The GPU which has become an integral part of today's mainstream computing systems, represents a compelling solution to address the increasing demand for processing power. First used as a processor to accelerate graphics rendering on screen, the enormous computational potential of GPUs has led to research for GPGPU. GPGPU uses GPU for non-graphics computations such as image and signal processing, neural networks, linear algebra, sorting, computational physics and database queries [7, 38, 28].

Over the past few years, the raw computational power of GPUs due to its parallelism has surpassed by far that of top range CPUs. Unlike general-purpose processors, GPUs are optimized to perform floating-point operations on large data sets using the paradigm Single Instruction Multiple Data (SIMD). This is specially important with machine learning algorithms which are often complex, placing high demands on memory and computing resources and CPUs are simply not powerful enough to solve them quickly for use in interactive applications.

To cope with this complexity NVIDIA developed a parallel technology namely CUDA (Compute United Device Architecture) which provides a programming model for its GPUs with an adequate API for non-graphics applications using standard ANSI C, extended with keywords that designate data-parallel functions.

Among the supervised Machine Learning algorithms, SVMs are perhaps the most widely used algorithm due to the exceptional generalization proprieties of the resulting models. SVMs are large margin classifiers which have found successful applications in many scientific fields such as engineering and bioinformatics [54], information management [1], finance and business [52] among many other. However, as many other machine learning algorithms they can be slow especially when large datasets are involved. Thus, a GPU implementation of this particular algorithm is desirable to reduce drastically the time needed to create SVM models.

1.2 Objectives

The goal of this Dissertation is to design, develop and implement a GPU SVM component to integrate/support GPUMLib software[1]. GPUMLib is a high-performance GPU machine learning library, implemented in CUDA, that aims to provide machine learning researchers and practitioners with a high performance library by taking advantage of the GPU enormous computational power [29, 26]. Figure 1.1 presents the main modules of the referred library [29, 26].More specifically, the component to be developed includes the implementation, test and experimentation of Support Vector Machines (SVMs).

Additionally, another important goal is to extend the software component with an advanced kernel which can simplify the kernel selection in a given problem and with stronger mapping ability to deal with large scale pattern recognition problems. This second goal is meant to contribute to the scientific part os this Dissertation.

More detailed, the aim is three fold: first, to test the multi-threaded CPU SVM implementation with benchmark data sets against the state-of-the-art LIBSVM;

[1]GPUMLib is a software component of the Doctoral work of the first author in the papers [29, 26] (in progress).

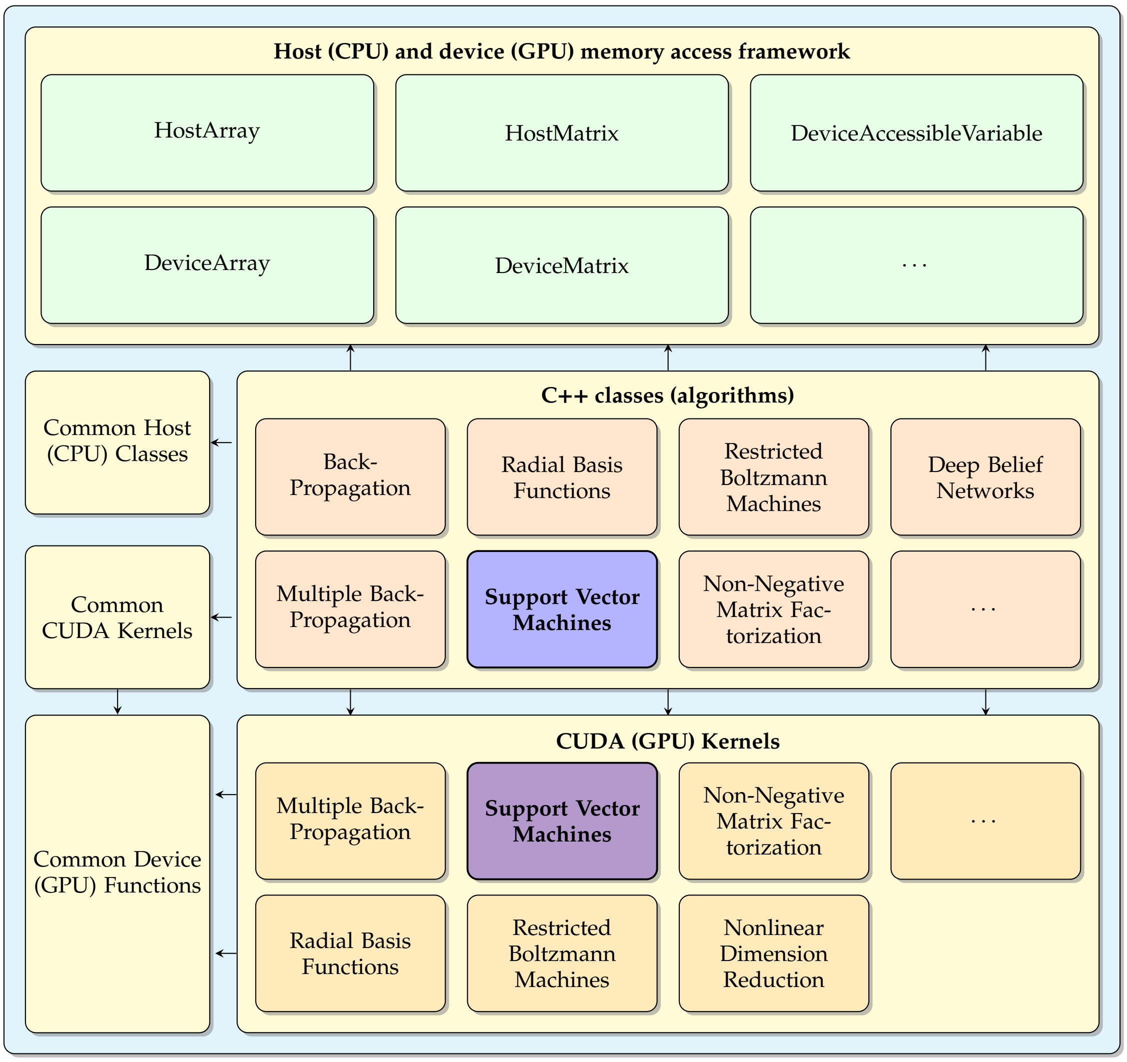

Host (CPU) and device (GPU) memory access framework
HostArray
HostMatrix
DeviceAccessibleVariable
DeviceArray
DeviceMatrix
...
Common Host (CPU) Classes
C++ classes (algorithms)
Back-Propagation
Radial Basis Functions
Restricted Boltzmann Machines
Deep Belief Networks
Common CUDA Kernels
Multiple Back-Propagation
Support Vector Machines
Non-Negative Matrix Factorization
...
Common Device (GPU) Functions
CUDA (GPU) Kernels
Multiple Back-Propagation
Support Vector Machines
Non-Negative Matrix Factorization
...
Radial Basis Functions
Restricted Boltzmann Machines
Nonlinear Dimension Reduction

Figure 1.1: Main components of the GPUMLib [29, 26].

second, to realize the SVM GPU implementation and compare the results with both the standalone CPU version and LIBSVM using the same benchmark datasets while retrieving speed-ups yielded by the implementations; third, to use the GPU SVM implementation in a study of a real life problem, off-line handwritten signature recognition.

1.3 Organization

The organization of this report is as follows. Chapter 2 analyses the state-of-the-art concerning GPU applications and frameworks. First, we summarize the main events that lead to the creation of the first GPU. Then we focus on the GPUs evolution into a fully programmable parallel device, which in turn capture the interest of programmers to use it as general-purpose computation device. To conclude Chapter 2 we analyze the NVIDIA's CUDA (Compute Unified Device Architecture) architecture, which is the platform that we will use to develop the proposed SVM GPU implementation.

Chapter 3 describes the SVMs and focus implementation aspects for both in CPU and GPU. In this chapter, we start by contextualizing SVMs in the broader area of machine learning. Then we define and introduce the concepts of linear classifiers, and describe formally the SVMs as large margin classifiers. Furthermore, we detail the convex optimization problem and in particular we analyze the Sequential Minimal Optimization (SMO) which is crucial to proceed to implement the algorithm. To conclude we survey issues regarding several aspects related both to the CPU and GPU implementations of the SVMs algorithm.

Chapter 4 presents the validation results for both the CPU and GPU implementations. First, we define the metrics to compare our SVMs with LIBSVM – a state-of-the-art SVM implementation. Then we describe the datasets used for that purpose. To conclude the chapter we present and analyse the results obtained with the implementation, which are very promising.

Chapter 5 gives a brief introduction to the problem of off-line handwritten signature recognition, describes the features and the dataset, exposes the results obtained using the GPU SVM and we give some conclusion on the study.

Finally in Chapter 6 we draw some conclusions about the work developed in the framework of this report and present future work guidelines.

Chapter 2

GPU Computing

This chapter gives an overview of GPGPU and the CUDA (Compute Unified Device Architecture) architecture which is the foundation of GPUMLib. We begin by introducing to the Graphics Processing Unit (GPU) predecessors (Section 2.1). Section 2.2 summarizes the main events that lead to the creation of the first GPU. Section 2.3 focus the GPUs evolution into a programmable parallel processor. Section 2.4 describes the events that lead the GPU to be used as general-purpose computation device. Section 2.5 details the NVIDIA's CUDA (Compute Unified Device Architecture) architecture for programming GPUs. Finally, Section 2.6 summarizes and concludes this chapter.

2.1 Introduction

Before the GPU appearance, there have always been specialized co-processors to help with the graphical operations in a computer. Those co-processors, known as Video Processor Units (VPUs), received commands from the CPU and executed them independently, freeing the CPU for other tasks. This was required whenever more graphical power was needed, for example in graphical workstations or gaming consoles. There are many examples of this separate flow of tasks like graphical, audio or I/O, such as the original IBM PC, with its Colour Graphics Adapter (CGA) card and the Commodore Amiga, which was a breakthrough in multimedia home computing.

On economic systems, for instance, the Sinclair's © ZX Spectrum range, the graphics circuitry was largely a system memory area connected to a Digital-to-Analogue-Converter (DAC), which copied the image hold in the Random Access Memory (RAM) to the computer's monitor. This design could, for extreme cases, deny the CPU access to the central RAM. This occurred when the memory area, holding the image, was scanned by the RAMDAC while at the same time the CPU was given HALT/SLEEP states. Because the memory couldn't be accessed by more than one device at the same time and multi-port memories were expensive, the CPU was denied access to central memory [47]. Additionally, it was up to the CPU to manipulate the graphics stored in the RAM. This always resulted in a slower performance because the machine had very little parallelism and too many tasks to perform at the same time.

2.2 The fixed-function GPU

In mid-nineties, the gaming industry was pushing the limits of contemporary computers. The earliest First Person Shooter (FPS) games, for instance *Wolfenstein 3D* © and *Doom* ©, heavily used the Floating Point Unit (FPU) for rendering a 3D perspective view of the game. With the purpose of freeing the heavily loaded CPU, manufacturers like ATI ©, NVIDIA ©, 3DFx! © and S3 ©, among others, began to build dedicated VPUs which offloaded some of the graphical tasks from the CPU.

Initially these VPUs accelerated only some specific tasks, for example filling areas of the screen with some color or moving sprites and blocks on the screen. The more advanced VPU chips also accelerated the stages of 3D rendering and lighting, while few supported the acceleration of geometry processing (vertex manipulation using matrix transformations), which was at the time done on the CPU. The VPUs were usually programmed using one of the two mainstream Application Programming Interfaces (APIs): OpenGL and Microsoft's ®DirectX. These APIs were then in a process of being standardized so that portable applications could be developed. Naturally, they represented a high-level view of the hardware, which was (and still is) forced to evolve constantly, with the release of new and more demanding software (e.g. games).

Later on, in 1997, appeared the first mass produced VPU, the 3DFx! VooDoo ©, capable of accelerating both the rendering and the rasterizing process. This, relieved the CPU from graphical tasks as the VPU could perform them faster, thanks to the parallelism obtained by using multiple pixel pipelines. The same applied to other VPUs like the NVIDIA ©Riva TNT (twin'n'texel') and ATi's Rage. At that time the VPUs were only specific to Computer Generated Imagery (CGI), used almost exclusively for games, Virtual Reality (VR) and computer animation. Nowadays, they are present in most mainstream computing systems.

The first GPU (a term coined by NVIDIA), was unveiled in 1999 when they

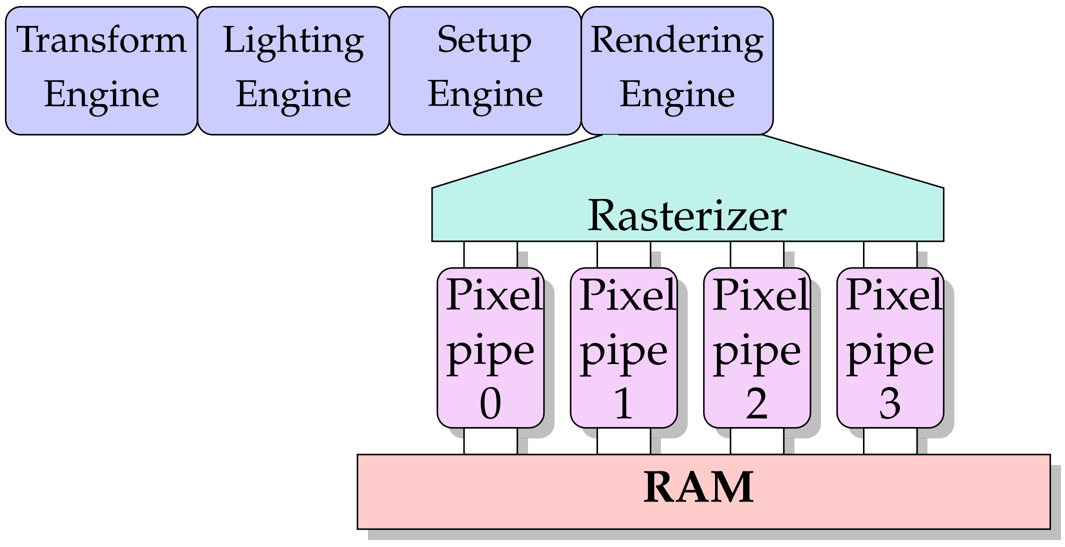


Figure 2.1: GeForce 256 architecture.

presented the GeForce 256 © describing it as *a single-chip processor with integrated transform, lighting, triangle setup & clipping and rendering engines* [32]. A simplified schematic of the GeForce 256 is shown in Figure 2.1. In the same way as the VPUs, additional tasks, like the geometry processing stage, were now offloaded from the CPU to the GPU [31, 38]. However, these first processors were not entirely programmable and were mostly state machines, with very rudimentary programming support. In contrast with current architectures, more evolved, this older technology is now named *Fixed-Function-Pipeline*. Figure 2.2 presents a typical Fixed-Function-Pipeline [7].

The GeForce 256, as well as most contemporary graphics cards of the time, already employed some parallelism (see Figure 2.1). The evident concurrency was easily extracted by rendering multiple pixels per clock cycle [38]. These pixels originate from the polygons composing the scene and this process is named rasterization. Consequently, before the common usage of shading techniques like blur, bloom, field-of-view, etc. the processing of each pixel could be easily done independently, because their color does not depend on neighbouring pixels.

To produce the image shown on a computer monitor, each object in the scene is rendered in a four-step engine consisting of the stages in the given order: *geometry processing*, *rasterization*, *fragment processing*, and *frame buffer processing*, as shown in Figure 2.2 [7]. The geometric data, composed of vertices, lines and triangles (more complex figures are tessellated into triangles) enter the **vertex processing** block, where the specified geometric transformations are performed [7]. Examples of these transformations are the translation, scaling and rotation of geometric objects, and the camera transformation. In the GeForce GPUs this stage is hardware

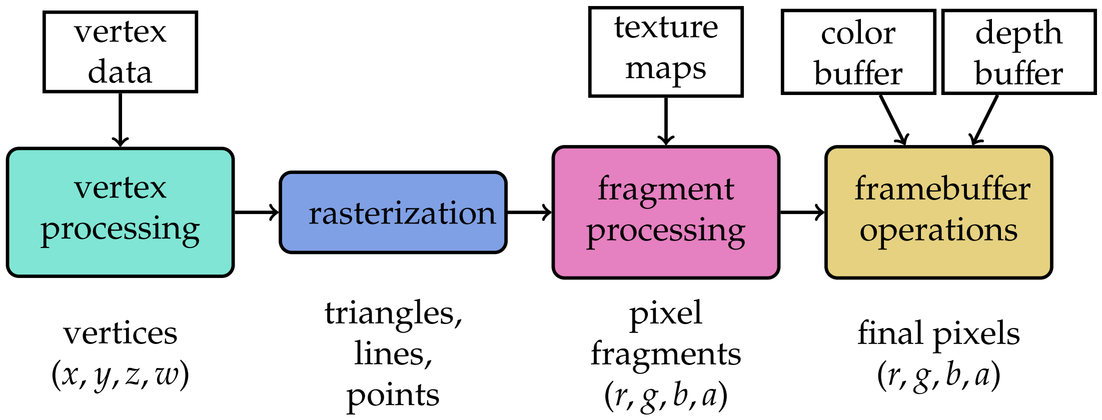


Figure 2.2: Fixed-Function-Pipeline [7].

accelerated by the *transform engine*, relieving the CPU from its processing. In summary, this geometry stage produces a *2D projection* of the scene composed of planar triangles, which in fact corresponds to the scene's projection into the screen.

The next stage, **rasterization**, converts each incoming 2D-triangle into a group of *pixels fragments* – the output of a discrete sampling of the area covered by the triangle [7]. These pixels are then manipulated by the **fragment processing** stage, which takes in account the triangle's lighting properties and its textures [7, 38]. The resulting pixel fragments are written to the *color buffer*, part of the frame-buffer, which stores a single fragment for each pixel composing the final image [7, 38]. This buffer, as well as most graphics data, may be stored in a dedicated memory module or in the CPU's RAM (shared memory) for low-cost systems.

In a typical scenario, a 3D application can have a very large amount of data to process: vertices and pixels. Exploiting the fact that most of the data is independent, the GPU can process it in parallel by pipelining the above fixed-function stages. This quest for maximum performance made the GPU evolve into a complex parallel processor.

2.3 The programmable GPU

With the advances in miniaturization and the costs reduction in the manufacturing processes, the early 2000s witness the addition of more advances in the GPUs pipeline [7]. The initial demand from the entertainment industry required more realistic effects and more stylized images [41, 7]. This greater desire of expression demanded more flexibility which could not be addressed by the old fixed-function pipeline [38]. Before new capabilities were added, special effects were generated by doing rendering multiple passes through the scene, combining them with different transforms and texturing operations.

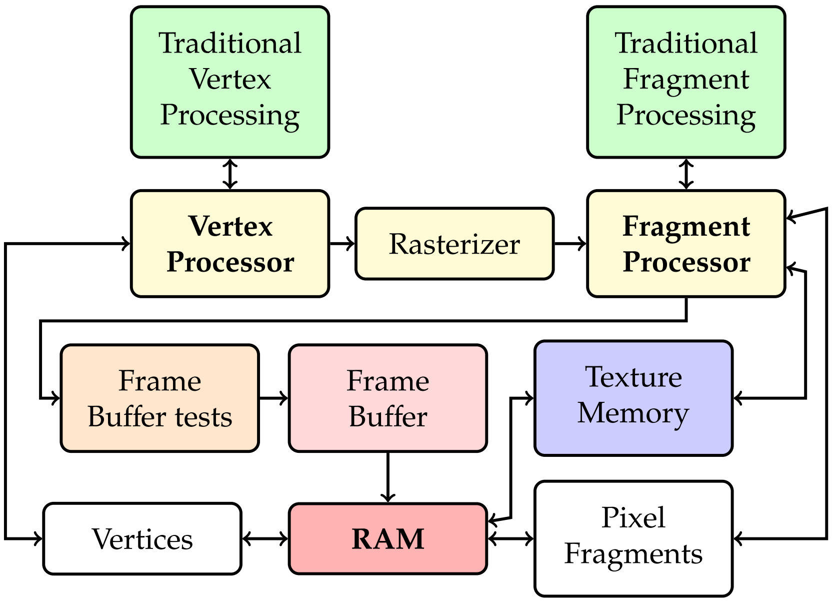
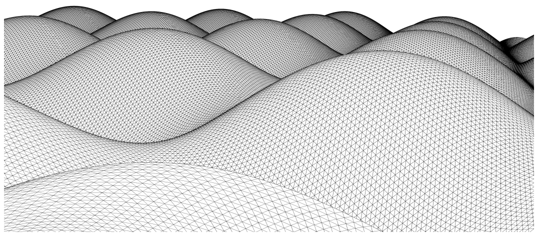


Figure 2.3: More advanced pipeline of a programmable GPU.

These new capabilities were added in the form of small programs, named "*shaders*", invoked for each vertex and pixel fragment [7, 33, 34]. The shaders allow the pipeline to be controlled with a greater degree, beyond what was done before, enabling the subroutines to manipulate data on-the-fly. The new architecture also allowed the rendering results to go to another section of the pipeline or to the ending buffer (frame-buffer or a texture). This can be seen in Figure 2.3. With the new architecture the applications may use both the fixed-function and shader pipelines or disable the old fixed-function pipeline and use only the new rendering engines, by making use of shading programs [35, 41].

To execute the user specified subroutines, the GPU includes two new stages in the pipeline being these the processor phases in Figure 2.3. The new programmable geometry stage is processed at the *vertex processor* and manipulates vertices without the CPU intervention. Consequently, the GPU can apply functions (*deformations* or *morphings*) to a group of vertices, for example to simulate waves of a sea (See Figure 2.4a). The same applies to the *fragment processor*, which handles the pixel fragments coming from the rasterizer using custom shaders. This stage may apply effects like *High-Definition-Rendering (HDR)* or water translucency (see Figure 2.4b), among others [41].

In order to offer programmers the possibility of developing code indepen-


(a) A vertex shader effect, surface deformation.

(b) Two fragment shader effects, HDR and water simulation.

Figure 2.4: Examples of shading effects

dent of the underlying platform, the hardware manufacturers had to support machine-independent shading languages. These languages are very similar to C and function as a form of concurrent execution for their operands (pixels or vertices) [7]. Well known shading languages are HLSL (for Microsoft's DirectX) and GLSL (for the API OpenGL) [7]. Naturally, these dedicated to graphics programming APIs are still in use today and continue to evolve.

2.4 General-Purpose computing on Graphics Processor Units (GPGPU)

In the fall of 2007, NVIDIA released a new GPU, the GeForce 8800 GTX (codename G80). This graphics card made the transition from a GPU with dedicated processing units (Vertex and Pixel processors) to a new unified pipeline paradigm [7]. In reality, the first GPU to support unified shaders was the Xenos chip (Microsoft's ® XBOX360) [38], thus the G80 simply made the shift to the PC architecture. Initially, this served the purpose of supporting applications with different vertex or pixel workloads. If, let's say, a game had extreme geometry detail with little pixel effects, or vice-versa, the new unified architecture should offer greater performance than the older separate Vertex-Pixel approach [7, 38]. In sum, the new unified pipeline allows the GPU to support higher level languages and a greater numerical computation.

Figure 2.5 depicts the G80 architecture [7, 38]. In this architecture the GPU is composed of 8 Thread Processing Clusters (TPCs) (not shown in the figure) segmented in two SMs. Each SM has a simple *in-order* Single Instruction Multiple Data (SIMD) processor containing one *instruction dispatcher* which decodes one

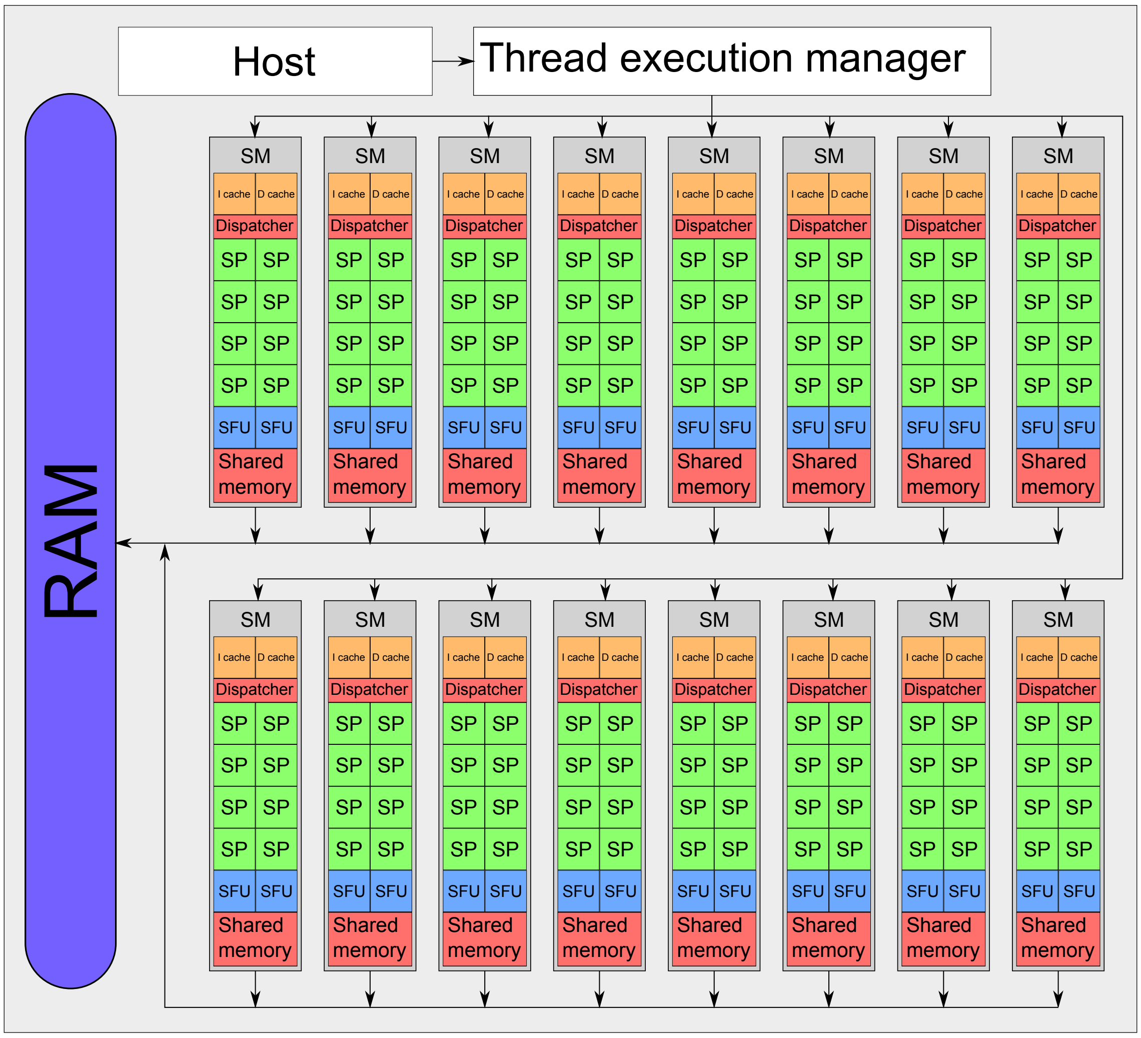
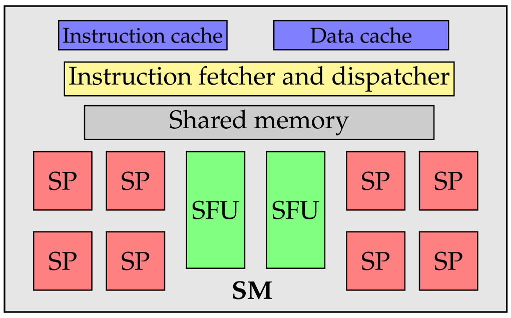


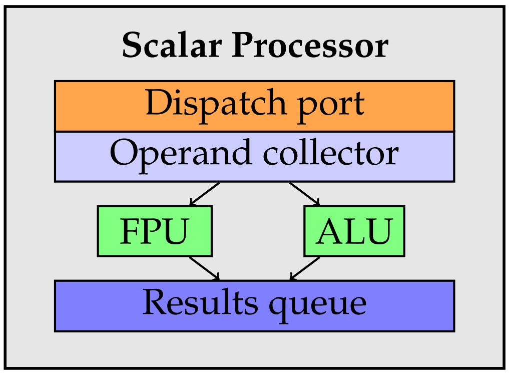
Figure 2.5: NVIDIA GeForce 8800 GTX (G80) architecture [7, 38].

instruction per clock cycle to one block of 8 parallel processing units, designated by Shading Processors (SPs) (see Figure 2.6).

In the GeForce 8800 GTX, each instruction (excluding transcendental operations) takes 4 cycles to complete and is shared by the 8 SPs contained in a SM. Therefore, the SPs executes instructions in groups of $4 \times 8 = 32$ instructions. This number is in CUDA (see Section 2.5) terminology named a *warp* and corresponds to 32 consecutive threads. Because the SPs run in groups, if some of them have to execute different instructions (for instance, due to a condition), threads are divergent in their execution. Basically, if within the same SM the SPs follow a different execution path, some of them execute one of the paths while the others are disabled. The same applies to the other path. Hence while the other SPs are executing their path, the former ones are disabled. For example, if the warp finds a conditional switch, execution is serialized N times, one for each different path taken [38, 36].



(a) A Streaming Multiprocessor (SM).



(b) A Scalar Processor (SP), also known as a CUDA core.

Figure 2.6: A Streaming Multiprocessor in the G80.

Each SPs behaves as a simple scalar CPU with both an Arithmetic and Logic Unit (ALU) and a FPU supporting full *IEEE.754* 32-bit single-precision floating point. Consequently, the SP can be seen as a simple Single Instruction Single Data (SISD) processor, that operates a single instruction on a single destination operand at a time (per clock cycle).

The SM also includes two Special Function Units (SFUs) with the sole purpose of handling complex mathematical operations such as square roots, reciprocals, trigonometric functions and other transcendental (not algebraic) operations. Basically, each SM is responsible by a single and independent flow of execution.

While ATi's GPUs make use of multiple Very Long Instruction Word (VLIW) processors, with greater theoretical performance, the G80, by implementing scalar processors, lowers the burden on the shading compiler. Because ATI's compiler can not make full use of the VLIW units (figure 2.7), the theoretical performance may not be achieved. Consequently, the code produced may not explore the available instruction parallelism, where different operations could be executed by the same instruction.

The G80 and its variants, with their scalar processors and lightweight thread execution, do not require the compiler to expose the program's parallelism. In this case, it is up to the programmer to write multi-threaded code, for what NVIDIA calls a Single Instruction Multiple Thread (SIMT) approach.

With this design, the G80 has definitely more capabilities for massive data processing than the CPU. The greater performance of the GPU, especially in the last years, has sparked a large performance gap between CPUs and GPUs. Figure 2.8 shows the performance of both architectures over the period 2003–2007. The slower performance improvement of the CPU is mainly due to difficulties in

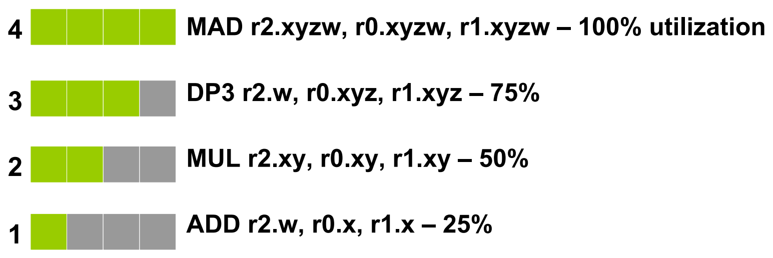

4
MAD r2.xyzw, r0.xyzw, r1.xyzw – 100% utilization
3
DP3 r2.w, r0.xyz, r1.xyz – 75%
2
MUL r2.xy, r0.xy, r1.xy – 50%
1
ADD r2.w, r0.x, r1.x – 25%

Figure 2.7: VLIW efficiency varies with scenario.

achieving higher clock speeds, which led CPU manufacturers to develop multi-core processors. Contrasting with the GPUs oriented data processing approach, the CPUs are tailored to handle both data control and data processing. On the other hand, modern GPUs are designed to run thousands of threads concurrently.

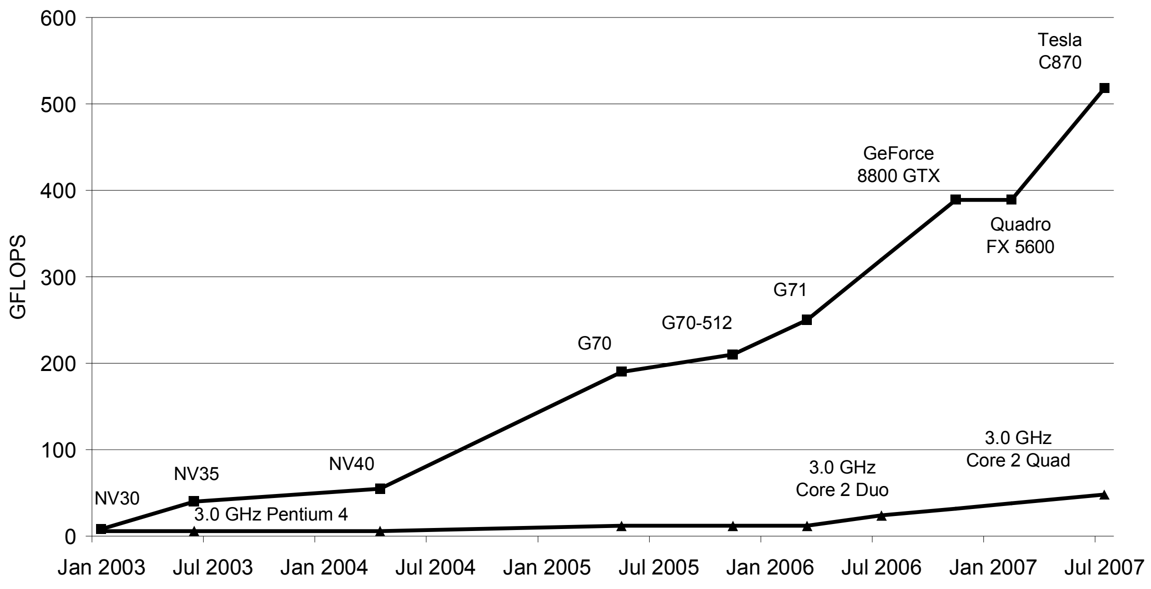
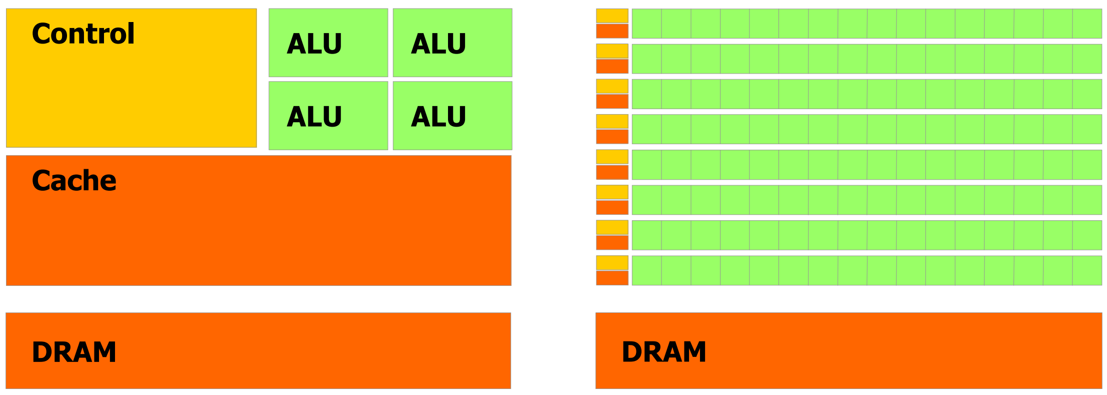


Figure 2.8: CPU and GPU performance comparison between 2003 and 2007 [36].

GPUs have much more transistors dedicated to mathematical operations, as compared against modern CPUs. For example in a GeForce 8800 GTX (with 128 SPs) running at a clock speed of 1.35 GHz, each SP cam execute one FPU and one ALU operation per clock cycle. Thus, the G80 can achieve a peak performance of $1.35 \times 128 \times 2 = 345.6$ GFlops. When compared to current CPUs, it's still very impressive for a GPU designed in 2007.

On the other hand, over time the CPUs have evolved to extract the best possible performance from single threaded operations. These optimizations come mostly from the fact that these processors have *super-scalar* pipelines, allowing instructions, after being decoded, to get executed in parallel functional units on



(a) CPU. (b) GPU.

Figure 2.9: The different philosophies behind CPU and GPU design [36].

the processor, if they're operator independent.

Current GPUs, such as the NVIDIA GeForce 460 GTX, already have a limited form of super-scalar execution. In this GPU, the SM has more groups of SPs than its instruction dispatcher can feed. But it can extract independent instructions from the incoming flow and assign them to those additional groups of SPs, when they belong to the same thread block. Note that this execution is done in-order, that is, the CPU does not try to find independent instructions ahead of time to execute. This contrasts with the out-of-order execution, explained next.

Another reason for the greater control area of the CPU is the support for *out-of-order* execution, where the processor attempts to extract independent instructions on the incoming flow of instructions, executing them on the available functional units, even if their result is irrelevant because those instructions maybe weren't supposed to be executed after all. Naturally, as this technology is quite complex, it isn't supported on modern GPUs, for the lack of silicon area.

To support the locality of data access and to maintain the internal pipeline occupied, the CPUs have much more dedicated on-chip cache than the GPUs. In contrast, the GPUs with a smaller cache per SM, must use software techniques to maintain the SPs occupied. The main approach to this problem is to have the SPs processing thousands of calculations in different threads, in order to hide memory latency.

With better programming support, the GPU has become a new target for applications with significant data parallelism, for instance physics simulation, fluid dynamics, mathematical computations, etc. These tasks were done previously with complex and expensive Symmetric-Multi-Processing (SMP) systems, inaccessible to most people. With the widespread availability of graphics cards and their reduced cost when compared to other systems, like computer clusters, a new programming paradigm has emerged, referred as GPGPU.

2.5 Compute United Device Architecture (CUDA)

The GeForce 8800, with its unified shader pipeline, is capable of executing parallel tasks written in a higher level language, as explained in the last section. In order to support the development of applications for the G80 and its successors, NVIDIA released in November 2006 the CUDA API. This was the first API supporting GPGPU. Initially oriented towards the `C` programming language, CUDA grow to support other languages like `C++` and `FORTRAN`. CUDA runs on UNIX, Windows and MacOS systems. However, it is only supported by NVIDIA GPUs.

Before CUDA, researchers had to resort to the Vertex/Pixel (V/P) approach, which was quite cumbersome to program. One example of an older V/P API still in use today is BrookGPU. This C research project aimed to deliver the higher arithmetic performance of the GPU as an abstract streaming processor. This streaming model, also followed by CUDA and other GPU computing architectures, requires programs to explicitly specify the separate flow of task and consequently, the software's parallelism.

In order to allow applications to make use of massive parallel GPUs, CUDA was designed from the beginning to scale with the available hardware resources. In order to maximize the architecture's performance, the programmer must make use of the following key concepts: hierarchies of thread groups, shared memories, and barrier synchronization [36]. But first, the algorithm to be implemented must have the property where many data operations can be safely performed in parallel.

In CUDA, a single GPU is handled as a *compute device* so that the programmer can run multiple pieces of software in various compute devices [36]. Naturally, each GPU has its own dedicated memory, named *device memory* and in current architectures this may hold a capacity up to 4 GB of high bandwidth Double-Data-Rate (DDR) RAM. There's always at least one host device controlled by the CPU (called the host) which launches parallel executions on the compute device. The CPU has also its memory space, which is in CUDA named *host memory*. Therefore, the software must manage both memories and exchange data between them in order for the GPU to do useful work [36].

Each parallel execution, in CUDA and in Open Computing Language (OPENCL), is named a *kernel*. A kernel is in fact a function, written in `C` with CUDA extensions, which is run N times in parallel, by each individual SP, on the compute device [36]. Consequently, the execution of a kernel corresponds to a variant of SIMD, known as Single Process Multiple Data (SPMD) where a single program spawns multiple lightweight threads on the same process [38].

In turn, each kernel is run across a grid of thread blocks, were a single block is executed on a single SM (see Figure 2.10) [36]. Depending on the GPUs Compute Capability (CC), which specifies the hardware limits imposed on a specific GPU architecture, the amount of threads included in a block may be limited for instance, to 512 elements on CC 1.0. Letting each thread block execute in whatever SM available allows automatic scheduling and scalability, independently of the amount of available SMs on the GPU (see Figure 2.10) [36].

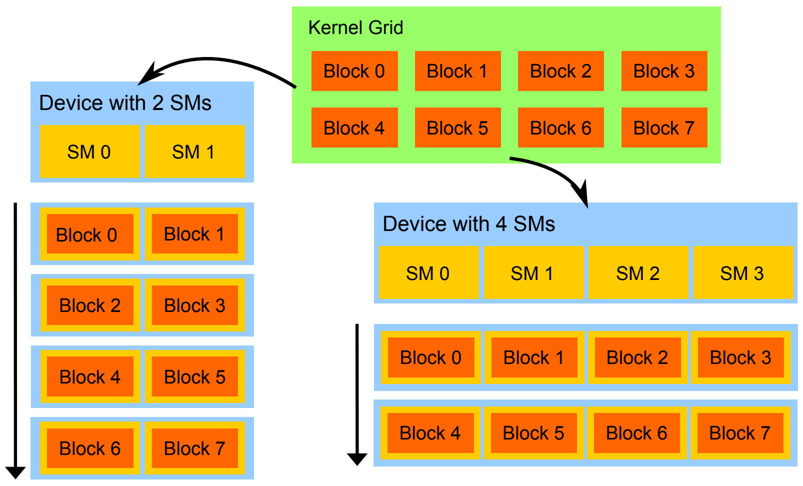


Figure 2.10: The blocks are automatically distributed according to the GPU's amount of SMs [36].

To be executed on the GPU, every kernel declaration requires the identifier `__global__`. Within this function, every thread has its own private variables and may access additional memory, as the shared memory, the constant memory and the global device memory (see Figure 2.11) [36]. Threads must have both a local and global identifiers, so that tasks can be isolated and given different data to work on. These identifiers are calculated from the thread's base ID and the block it belongs to [36]. In turn the GPU manages and executes blocks within a greater group called the grid. For convenience, both threads and blocks may be identified using one to three dimensional indexes, allowing the problem to be easily extended to other domains like a vector, a matrix or a volume [36].

A common example of a kernel given in introductory courses is shown in Algorithm 1. This sample code adds two vectors, A and B both of length N, storing the results into vector C. Note that the initialization of the three arrays and their transfers between host and device's memories are not shown.

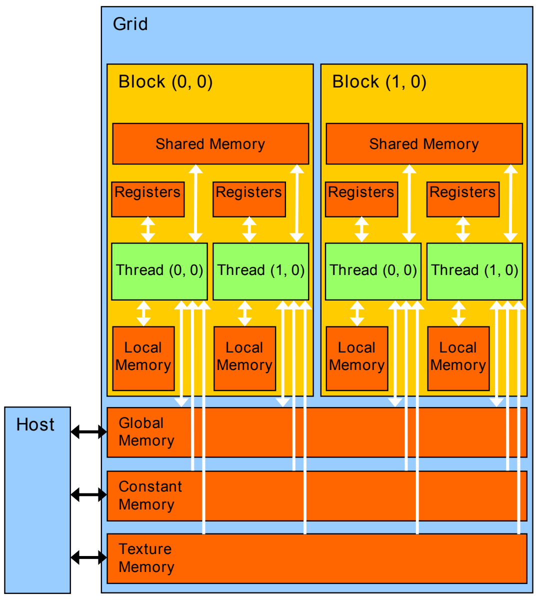


Figure 2.11: Threads, blocks and memory spaces [36].

Algorithm 1 Sample code for a kernel which adds two vectors

```
#define N vector_size
// Kernel definition
__global__ void vec_add(float A[N], float B[N], float C[N])
{
        int i = blockIdx.x * blockDim.x + threadIdx.x; //compute global thread ID
        if (i < N)
                C[i] = A[i] + B[i];
}

int main()
{
        // initialize A, B & C arrays in host...
        // copy arrays to device...
        // etc.

        // Kernel invocation
        // must be careful to respect the device's compute capability
        int threads_per_block = 256;
        //guarantees enough threads & blocks, amount of threads >= N
        int num_blocks = ceil((double)N/(double)threads_per_block);
        vec_add<<<num_blocks, threads_per_block>>>(A, B, C); // run kernel
}
```

On the above code, each thread executing the kernel performs one pair-wise addition. Because there is a hardware limit on the amount of threads per block, these are usually calculated in accordance to the GPU capabilities (CC). In the code, for simplicity, this is statically set to 256 threads. In order to process the arrays of length N, the required amount of blocks is calculated such that the total amount of threads is at least N. The threads in excess do not execute the addition because of the explicit conditional. As said on the beginning of this section, the kernel's execution follows the SIMT design.

Each block of threads may access internally a small shared memory with a

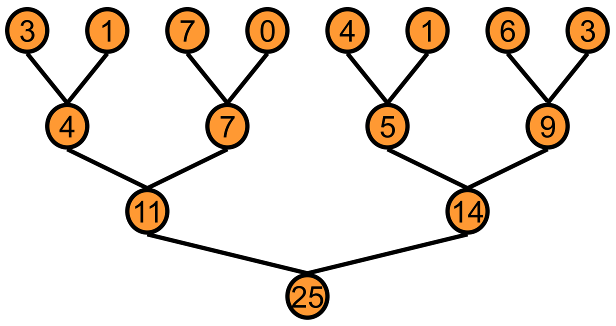


Figure 2.12: The tree structure behind a reduction.

limited amount of capacity - 16 KB for devices with CC below 2.0 and 48 KB for capability 2.0. This small cache is visible to all the threads of a block and its lifetime is as long as a block's, because its contents may be replaced by the next scheduled block to execute on the SM. Additionally, an intelligent usage of this memory is necessary to maximize performance, especially to get above the device's memory bandwidth limit [36].

The reason for this is simple. For instance, the GeForce 8800 GTX has a theoretical processing power of 345.6 GFlops. But the graphics card memory has a bandwidth of 86.4 GB/s, and as each standard floating point and integer datatype require 32 bits (4 bytes) of storage, feeding the SPs directly from the device's memory will hold a throughput of 21.6 GFlops. Consequently, in order to hit a better performance, the SPs must get their data from a nearer and faster memory - the shared memory (see figure 2.11).

With the strong appealing from the GPU's raw performance, researchers and manufacturers began to developed basic parallel building blocks on GPUs, like the operations reduce, scatter, gather and other distributed procedures [38]. An example of a reduction-sum can be seen on Figure 2.12, where the elements of a list are reduced into a final result, the sum. Figure 2.13 represents a graphical view of the algorithm.

The basic building blocks are then used to design higher level constructs, which can be used to develop more advanced programs. Examples are CUBLAS - a Basic Linear Algebra System and CUFFT - a Fast Fourier Transform interface, both ported by NVIDIA to CUDA.

Building on the above basic blocks, researchers have developed more advanced uses of GPU computing. The solving of differential equations, physic's simulation, tomographic reconstruction algorithms and machine learning, among others, are just some examples of the areas gaining with the usage of the GPU.

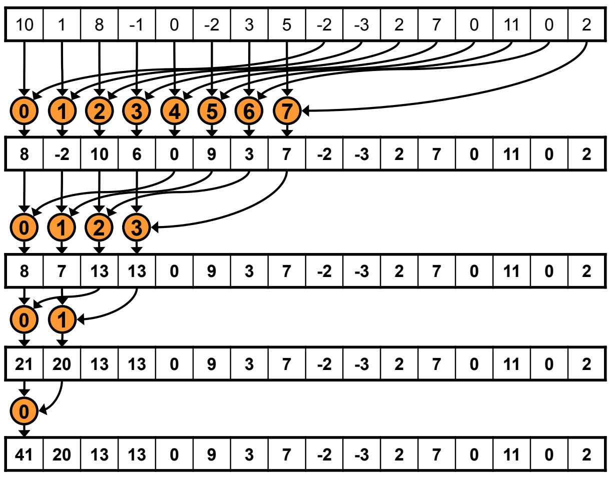


Figure 2.13: An example of a efficient reduction-sum in CUDA [19].

2.6 Conclusion

In this chapter, we gave an overview on the evolution of the GPU, from a fixed-function device into a fully programmable parallel device capable of running general purpose applications. Moreover, we detailed the NVIDIA Compute United Device Architecture (CUDA) (CUDA), which will be used to develop the SVM GPU implementation to be integrated in the GPUMLib library.

Chapter 3

Support Vector Machines (SVMs)

In this chapter a class of learning mechanisms known as the Support Vector Machines (SVMs) which are the main focus of this dissertation are described. We start by giving the machine learning framework, define and introduce the concepts of linear classifiers, and describe formally the SVMs as large margin classifiers. We focus on the convex optimization problem and in particular we deal with the Sequential Minimal Optimization (SMO) which is crucial to proceed to implement the algorithm. Finally we detail issues of the SVMs implementation. Regarding the latter, several aspects related to CPU and GPU implementation are surveyed. Our aim is two fold: first, we implement the CPU version, test it in benchmark data sets; then we proceed with the GPU version. By the end of this work we expect to have results on the GPU platform, in both synthetic and real large scale problems. As one important component of this Dissertation is the implementation and performance surveillance of the SVMs, decisions on how to make SVMs run faster (and better) are taken. We end the chapter with a summary of conclusions.

3.1 Background in Machine learning

Machine learning deals with the development of technologies which allow machines to learn. The challenge is to create algorithms that can take a group of patterns (on a broader range, the existing knowledge) and automatically make new inferences from the initial information, with or without human intervention [30].

Certainly, most algorithms discovered until now are based on the efforts of both psychologists and neurologists to understand not only the human mind, but even the simplest brain of a more humble organism. Helping with this task mathematicians and engineers apply statistical methods, by studying random variables in order to understand their behavior and their properties. It is also very likely that both areas will somehow help on the understanding of the fundamentals behind the learning process [30]. More detailed aspects of Machine Learning can be found in [42].

Comparing with the common computer programming, a program or machine learns whenever it changes its internal structure in accordance to external impulses. It is natural to notice a resemblance to the process of human learning. The learning by the machine itself is a well-known application of **Artificial Intelligence (AI)**, which comprises many learning mechanisms, besides other tasks, such as planning, control, prediction, etc. One typical example of machine learning, which matches closely this dissertation, is to "teach" the machine to distinguish two classes of pictures, let us say, apples from oranges. This falls in a typical problem in Pattern Recognition [13]

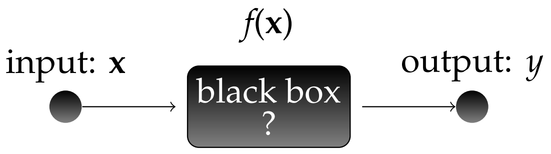


Figure 3.1: A black-box which applies some function to its inputs, presenting the result on its outputs.

The learning process can then be seen as a black-box (see Figure 3.1) which receives the inputs of several samples of each fruit, from the domain χ and the label of each image from the domain Y. The domain usually represents some value associated to each class, or in the above case, the kind of fruit.

Using some learning mechanism hidden inside the black-box, the machine will, hopefully, remember (the technical term is **classify**) each picture of a fruit to its kind. This association and the black box correspond to a function, $f(\mathbf{x})$, which takes an input vector $\mathbf{x}$ and assigns it one class from the domain Y. The learning mechanism relates to how this function is in fact built.

The initial n examples fed to the algorithm, while training, constitute the **train set**:

$$\chi = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\} \tag{3.1}$$

where each $\mathbf{x} \in \mathbb{R}^d$ and d is the size of the feature set. For the current example, d might be the amount of individual pixels that compose each picture and n the number of images that constitute the training set.

The values to be learnt are the **targets** or **classes**:

$$\Omega = \{y_1, y_2, ..., y_n\} \tag{3.2}$$

Additionally, it is sensible to validate the learning process with a different group of samples, the **test set**, so that its classification performance can be measuredand perform correctly inferences from the data [5] . There are various metrics available to assess the results of classification as will be discussed later.

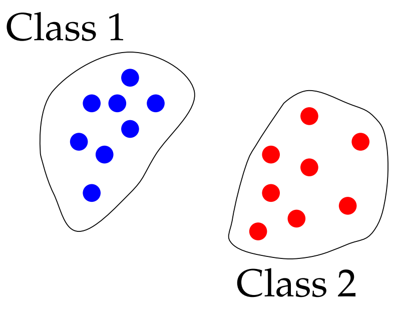


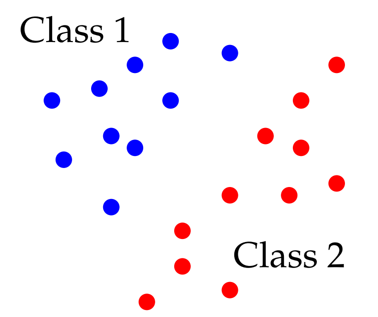
Figure 3.2: An example of unsupervised learning – binary clustering.

This process of giving the machine the value associated with each image is known as **supervised learning**, because there is an external entity - the supervisor - which helps with the learning process, giving training examples and their classes. This supervisor corresponds to what we know as a teacher, who teaches us something to be learnt. Finally, using the above example of the fruits, as its co-domain is comprised of only two classes (the banana and the orange types), it is known as a two-class (**binary**) classification problem ($c = 2$) [30].

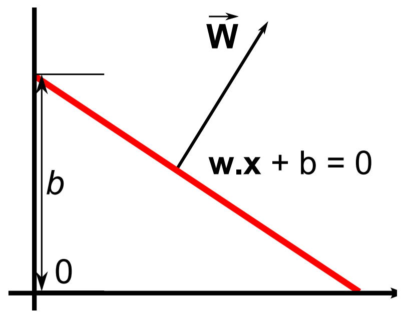
The supervised learning contrasts with the **unsupervised learning**, where it is up to the algorithm to decide to which class each sample belongs. This can be done, for example, using some form of clustering, where nearby samples are clustered in the same class (see Figure 3.2).

3.2 The linear classifier

The theory behind the SVM has its ground on linear binary classifiers. These are, on their simplest form, used as a solution to problems with two linearly separable classes. In these situations, the linear classifier tries to find one boundary between the two training samples, one for each class. Most problems can be broken in the form of boundaries and SVMs are no different. Thus they are easier to understand using a geometric approach, which will be introduced by the use of simpler linear classifiers.



(a) Two classes which can be linearly discriminated.



(b) Geometric interpretation of the hyperplane's variables, the w vector and the bias (b).

Figure 3.3: A hyperplane discriminating two classes.

These classifiers, being linear, discriminate between two classes by using a hyperplane (see Figure 3.3a). Each class of the binary problem has a corresponding value on the set $\Omega = \{-1, 1\}$. Assuming both classes are linearly separable, a decision surface can always be found and the hyperplane discriminating both classes is defined as:

$$0 = \mathbf{w} \cdot \mathbf{x} + b \tag{3.3}$$

Figure 3.3b shows the geometric interpretation of the hyperplane and its parameters. As a consequence the linear classifier discriminating surface is a function of both **w** (also known as **weight** vector) and the bias, b. The vector $\vec{\mathbf{w}}$ is always perpendicular to the hyperplane, that is, the hyperplane's direction is defined through **w**. The variable b represents the plane's exact position in the feature space χ. Both parameters are deduced on the training process, from the training samples [48].

The output of the linear classifier is calculated by using the dot product between the vector **x** and the weight vector **w**, which is the same as taking a *linear combination* between both vectors:

$$y(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b = \langle \mathbf{x}, \mathbf{w} \rangle + b = \sum_{i=1}^{d} (\mathbf{w}_i \cdot \mathbf{x}_i) + b \tag{3.4}$$

This corresponds, geometrically, to check if the projection of a given input **x** in the equation 3.4 is above (or below) the decision hyperplane defined by equation 3.3, which can be accomplished simply by checking the signal of $y(\mathbf{x})$.

One example of a linear classifier, which learns the hyperplane after some iterations, is the **perceptron** (see Figure 3.4). Invented in 1962 by Frank Rosenblatt [30], the perceptron was a major discovery in the field of AI and in machine learning. The perceptron may have different activating functions, from which the *step*, *sigmoid* and *hyperbolic tangent* are the most common examples.

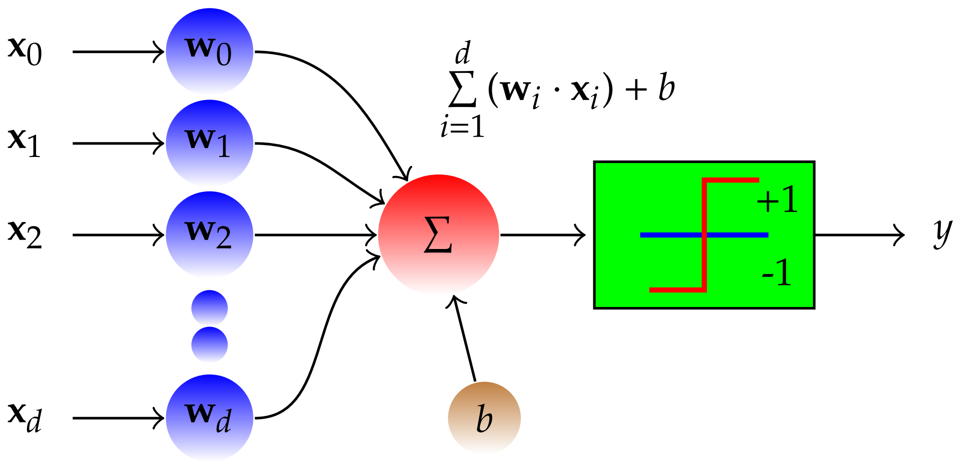


Figure 3.4: The perceptron with a step transfer function.

As a linear classifier, the perceptron takes a weighted sum (linear combination) of its inputs and then applies a step transfer function. In this case, the classification for a binary problem is given by the signal of the function $f(\mathbf{x})$ (see equation 3.5). When its output is above zero, the perceptron identifies it as the class labeled $\{1\}$, otherwise it corresponds to $\{-1\}$ (see Figure 3.5a). The test for zero may be chosen randomly to fall in either one of the classes. Hence, the classification will be the following:

$$y(\mathbf{x}) = \operatorname{sign}\left(\sum_{i=1}^{d}(\mathbf{w}_i \cdot \mathbf{x}_i) + b\right) \tag{3.5}$$

During the perceptron's training, the algorithm updates both the weight vector and the bias with a small percentage of the training error. This error, e, is defined as follows:

$$e = c_i - y(\mathbf{x}) \tag{3.6}$$

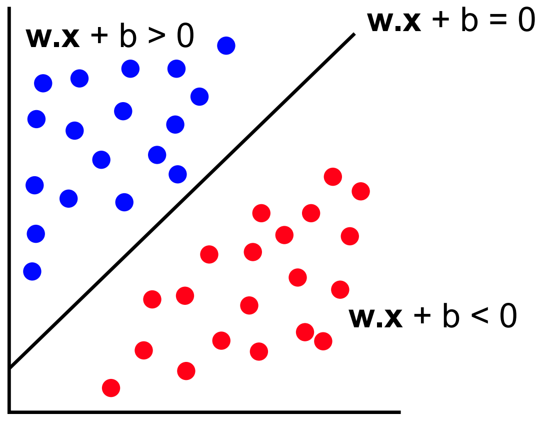
As one can expect, the training error is the difference between the desired output and the perceptron's classification for a given input. When the algorithm starts, both $\mathbf{w}$ and b are usually initialized to zero or to a random configuration, so that different decision hyperplanes may be generated in various runs. In each training iteration, both the weight vector $\mathbf{w}'$ and bias are corrected with a (small) fraction of the classification error. With the mechanism described above the perceptron's learning process can be viewed as a sort of stochastic gradient descent search:

$$\mathbf{w}'_i = \mathbf{w}_i + \theta \cdot e\mathbf{x}_i \tag{3.7a}$$

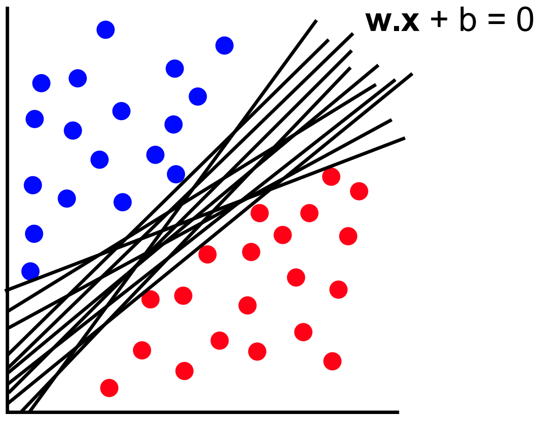
$$b' = b + \theta \cdot e \tag{3.7b}$$

The additional parameter θ controls the convergence rate, where smaller values may converge to a better training error. On the other hand, larger values for θ may force the error to display an oscillatory behavior. In summary, the parameter change is a driver for the rate of convergence for training error in the algorithm [13].

Visually, the perceptron decision surface (hyperplane) changes with the vector's **w** weights and the bias value, b. Minor shifts in the hyperplane may still discriminate all of the training samples (figure 3.5b). The solution that the algorithm eventually converges is dependent on the initial values picked for the vector **w** and the bias b, including the order by which the samples are given.



(a) A linear classifier which uses the function 3.5 to discriminate between the two classes.



(b) Infinite possibilities for the hyperplane when the classes are linearly separable.

Figure 3.5: A hyperplane discriminating two classes.

3.3 The Support Vector Machine

Support Vector Machines (SVMs) are large margin classifiers which have found successful applications in many scientific fields such as engineering and bioinformatics [54], information management [1], finance and business [52] among many other.

From the explanations in above section and also taking into account Figure 3.5b it can be seen that there are countless even infinite possibilities to build the separation hyperplane. Considering the perceptron's formulation we consider

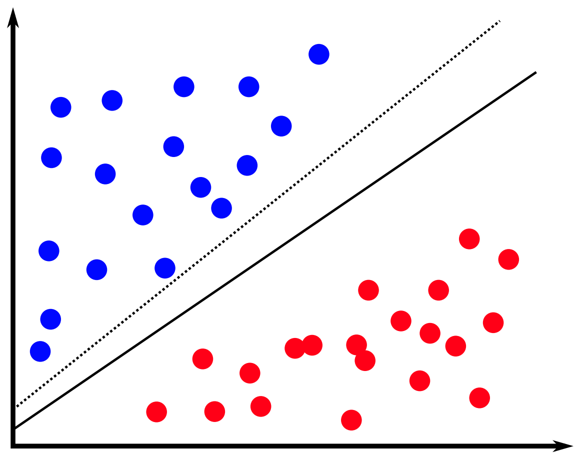


Figure 3.6: Two possible hyperplanes which discriminate both classes.

that there is a boundary between the two classes limited by two margins. The margins delimiting the boundary are defined with the following equation (the canonical representation of both the negative and positive hyperplane):

$$c_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 \tag{3.8}$$

If we aim to reach an optimal hyperplane, we should try to find the one that gives the smallest generalization error, that is, to minimize the empirical error. For instance, given both hyperplanes in Figure 3.6, we should reach for the thick black line, because it leaves more space on either side of the decision plane and creates the widest gap between both classes. In this way, the classifier is best prepared for unknown data [5, 48].

An important and crucial point in the SVM formulation is that it can provide a good generalization independent of the training set's distribution by making use of the principle of *structural risk minimization* (see Figure 3.7) [49, 11]. This principle provides a trade-off between the complexity of the classifier (accuracy in the training set) and the quality of fitting the training data (generalization - empirical error). Therefore, the SVMs belong to a class of algorithms which are known as **Maximum-Margin Classifiers** [50, 46]. The size of the gap is decided upon the training samples which are between the margins. These samples are the so-called **support vectors** and are associated with α_i, the Lagrange multipliers, which are greater than zero as explained further below. As we do not want to give preference to any of the classes, the hyperplane will be at the same distance of each classes' margins. In figure 3.8 we can see a non-linear separable case where the support vectors correspond to the samples with marks, standing exactly on the margin's limits.

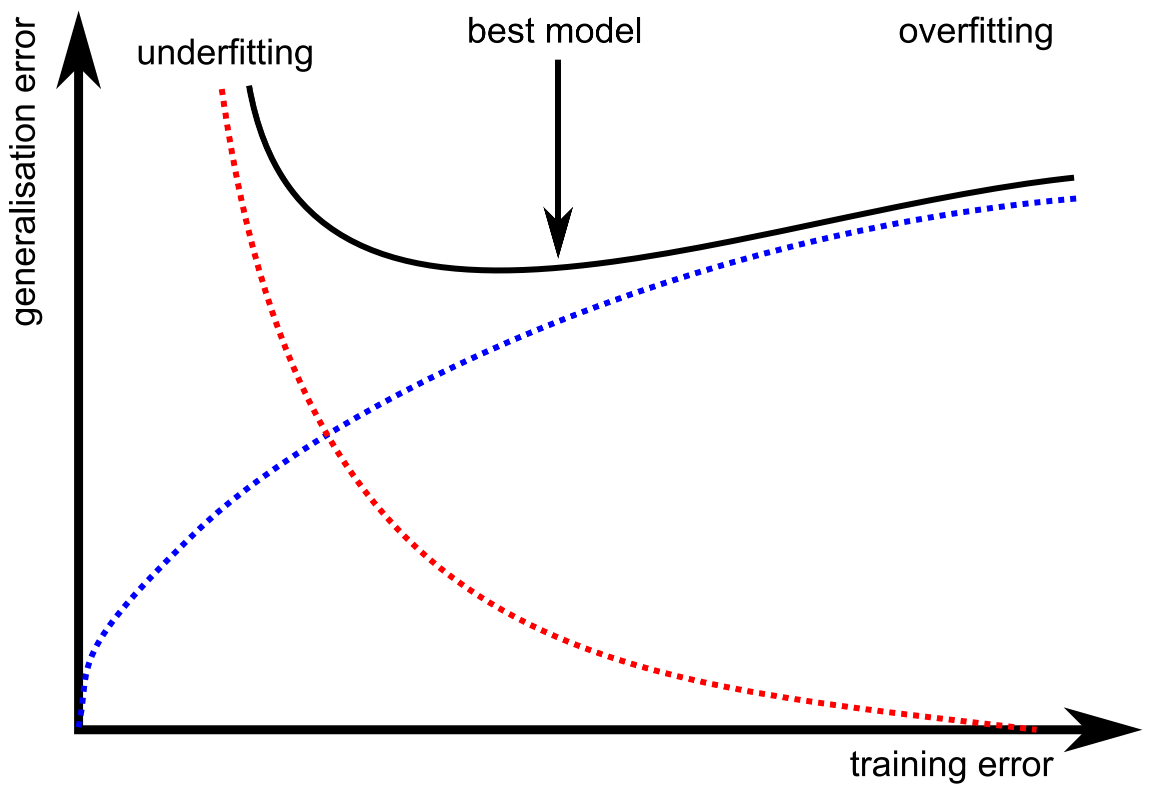


Figure 3.7: Structural risk minimization balances generalization with training error, through the classifier's complexity.

As explained, the goal of the SVM's algorithm is to search for the direction that gives the widest margin. Doing so, we will find both the vector **w** and the offset b. Resorting to geometric definitions, the distance from each class boundaries to the discriminating hyperplane is $||\mathbf{w}||^{-1}$. As a result the size of the margin between both classes will be:

$$\frac{1}{||\mathbf{w}||} + \frac{1}{||\mathbf{w}||} = \frac{2}{||\mathbf{w}||} \tag{3.9}$$

Note that the size of the margin is inversely proportional to **w**. For each training pattern its classification must be on the corresponding side of the margin. This SVM does not assume misclassified samples, consequently is known as a Hard Margin SVM. The canonical form of each classes' boundary is:

$$\mathbf{w} \cdot \mathbf{x} + b \geq 1, \qquad \forall \mathbf{x} \in \omega_1 \qquad \Leftrightarrow c = +1 \tag{3.10a}$$

$$\mathbf{w} \cdot \mathbf{x} + b \leq -1, \qquad \forall \mathbf{x} \in \omega_2 \qquad \Leftrightarrow c = -1 \tag{3.10b}$$

These equations are equivalent to equation 3.8. With the above in hand, we want to maximize the size of the margin while making sure that all the training points are on the correct side of the hyperplane. This formulation is known as the

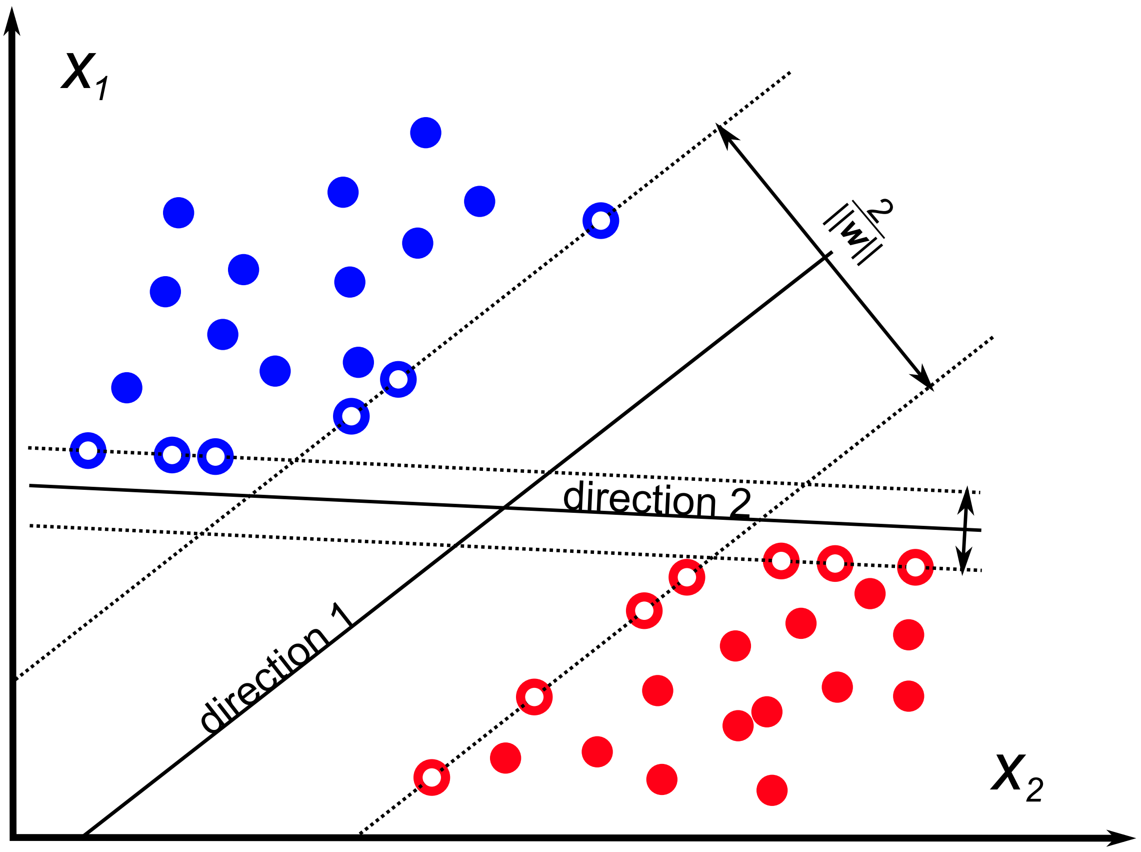


Figure 3.8: Two possible hyperplanes, but the one with the label "direction 1" has a greater margin.

primal problem and can be written as:

$$\text{minimize} \qquad \frac{1}{2}||\mathbf{w}||^2 = \frac{1}{2}\mathbf{w}\cdot\mathbf{w} \tag{3.11a}$$

$$\text{subject to} \qquad c_i(\mathbf{w}\cdot\mathbf{x}_i + b) \geq 1, \qquad i = 1, 2, \ldots n \tag{3.11b}$$

Note that equation 3.11a corresponds to a simplification of the vector's $\mathbf{w}$ norm. In this way there is no need to calculate its exact length (through the square root) while searching for its minimum which is equivalent to find the largest margin. The factor $\frac{1}{2}$ is only present for an easier derivation further ahead (in the development of 3.14). Arriving to this formulation makes it a quadratic optimization task, because of equation 3.11a. The search for the minimum of $||\mathbf{w}||$ is done on a convex surface (a parabola in $\mathbb{R}^d$) and consequently it has a single, optimal solution [48].

The problem can be written as a Lagrangian formula from the above equations, where it can be searched for a minimum [11]. For each constraint in 3.11b there is a corresponding Lagrange multiplier, α_i. The Lagrangian function, $\mathscr{L}$, will then be:

$$\mathscr{L}(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{n} \alpha_i[y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1] \tag{3.12}$$

As a quadratic programming problem with constraints, the Karush-Kuhn-Tucker (KKT) [11] conditions can be applied to function 3.12. As a consequence, the solution represents the hyperplane that gives the maximum margin. The KKT conditions are the following:

$$\frac{\partial}{\partial \mathbf{w}}\mathscr{L}(\mathbf{w}, b, \boldsymbol{\alpha}) = 0 \tag{3.13a}$$

$$\frac{\partial}{\partial b}\mathscr{L}(\mathbf{w}, b, \boldsymbol{\alpha}) = 0 \tag{3.13b}$$

$$\alpha_i \geq 0, \qquad i = 1, 2, \ldots, n \tag{3.13c}$$

$$\alpha_i[y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1] = 0, \qquad i = 1, 2, \ldots, n \tag{3.13d}$$

By applying the KKT conditions to the Lagrangian function $\mathscr{L}$ in 3.12, the composition of vector **w** can be found analytically as:

$$\frac{\partial}{\partial \mathbf{w}}\mathscr{L}(\mathbf{w}, b, \boldsymbol{\alpha}) = 0 \Leftrightarrow \tag{3.14}$$

$$\Leftrightarrow 0 = \frac{\partial}{\partial \mathbf{w}}\frac{1}{2}||\mathbf{w}||^2 - \frac{\partial}{\partial \mathbf{w}}\sum_{i=1}^{n} \alpha_i[y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1] \tag{3.15}$$

$$= \frac{2}{2}\mathbf{w} - \frac{\partial}{\partial \mathbf{w}}\sum_{i=1}^{n} \alpha_i[y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1] \tag{3.16}$$

$$= \mathbf{w} - \sum_{i=1}^{n} \alpha_i[y_i(\mathbf{x}_i + 0) - 0] \tag{3.17}$$

$$= \mathbf{w} - \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i \Leftrightarrow \tag{3.18}$$

$$\Leftrightarrow \mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i \tag{3.19}$$

And the same for b, the offset:

$$\frac{\partial}{\partial b}\mathscr{L}(\mathbf{w}, b, \boldsymbol{\alpha}) = 0 \Leftrightarrow \tag{3.20}$$

$$\Leftrightarrow 0 = \frac{\partial}{\partial b}\frac{1}{2}||\mathbf{w}||^2 - \frac{\partial}{\partial b}\sum_{i=1}^{n}\alpha_i[y_i(\mathbf{w}\cdot\mathbf{x}_i + b) - 1] \Leftrightarrow \tag{3.21}$$

$$0 = 0 - \frac{\partial}{\partial b}\sum_{i=1}^{n}\alpha_i[y_i(\mathbf{w}\cdot\mathbf{x}_i + b) - 1] \Leftrightarrow \tag{3.22}$$

$$0 = 0 - \sum_{i=1}^{n}\alpha_i[y_i(0 + 1) - 0] \Leftrightarrow \tag{3.23}$$

$$0 = -\sum_{i=1}^{n}\alpha_i y_i \Leftrightarrow \tag{3.24}$$

$$\Leftrightarrow \sum_{i=1}^{n}\alpha_i y_i = 0 \tag{3.25}$$

According to both 3.13c and 3.13d there is a Lagrange multiplier for each constraint, so when α_i is greater than zero the corresponding point $\mathbf{x}$ belongs to one of the margin's hyperplanes. As said before, these points are the so-called support vectors. In case of $\alpha_i = 0$, the related points may be outside their class boundary or a degenerate case, as developed in [48].

With the definition of the problem set initially in 3.11, the next step is the calculation of the α_i values. In this case, the theory behind quadratic (or convex) programming tells us to transform the problem in an equivalent one, called the Lagrangian dual. Continuing from formulation in 3.12 and the conclusions taken in 3.14 and 3.20, the dual will be:

$$\text{maximize} \quad \mathscr{L}(\mathbf{w}, b, \boldsymbol{\alpha}) \tag{3.26}$$

$$\text{subject to} \quad \mathbf{w} = \sum_{i=1}^{n}\alpha_i y_i \mathbf{x}_i \tag{3.27}$$

$$\sum_{i=1}^{n}\alpha_i y_i = 0 \tag{3.28}$$

$$\boldsymbol{\alpha} \geq \mathbf{0} \tag{3.29}$$

Replacing 3.27 and 3.28 into the Lagrangian formula 3.26 to be optimized, we obtain the dual optimization problem:

$$\text{maximize} \quad \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x} \cdot_i \mathbf{x}_j \tag{3.30}$$

$$\text{subject to} \quad \sum_{i=1}^{n} \alpha_i y_i = 0 \tag{3.31}$$

$$\text{and} \quad \alpha_i \geq 0, \qquad i = 1, 2, \ldots n \tag{3.32}$$

To find the optimal Lagrangian multipliers there are various methods which can be employed, like the gradient descent algorithm, the Newton's method or the SMO algorithm [22], this last being used on this work. These "simple" methods can be applied here because, as said before, the function to be optimized is convex (equation 3.11a). After retrieving the solution, the vector **w** can be built using the equation 3.27 and the offset b by the constraint's equation 3.13d.

Note that the formulation in 3.30 contains pairs of variables, similar to dot products. This will be useful to reformulate the problem for non-linearly separable classes, by making use of the kernel trick.

The classification can be done in the same way as for the perceptron, by using equation 3.5:

$$y(\mathbf{x}) = \operatorname{sign}\left(b + \sum_{i=1}^{d} \mathbf{w}_i{}^T \mathbf{x}_i \right) \tag{3.33}$$

or by using the dual and the Lagrange multipliers. This method uses a subset of the training set – the support vectors – which are the samples for which $\alpha_i > 0$. This has the advantage of the less SVs the SVM uses, the faster the classification is. The predicted class is calculated as follows:

$$y(\mathbf{z}) = \operatorname{sign}\left(b + \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i^T \mathbf{z} \right) \tag{3.34}$$

3.4 Soft margin SVM

The last section considered the case where a binary classification problem can be perfectly separated by a hyperplane. However, this does not correspond to the majority of the situations encountered in the real world. When trying to find the largest margin, the SVM must allow some misclassified samples. Naturally, these are penalized because they are on the wrong side of the decision hyperplane and we want to minimize this amount. Consequently, this SVM will try to minimize the influence of difficult samples on the optimization problem.

This kind of SVM is known as the soft margin version, because it "softens" the constraints regarding the misclassification of the patterns considered problematic. The support vectors are not required to be positioned exactly on their canonical hyperplanes, but they are allowed to go beyond the other classes' hyperplane (see Figure 3.9). The decision hyperplane is now built from both correct and incorrect samples, all support vectors [45].

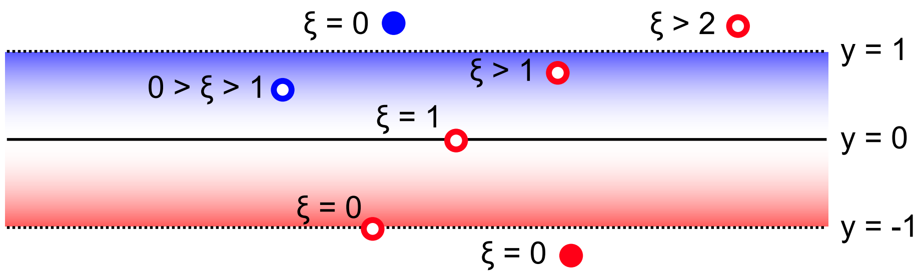


Figure 3.9: An example with slack variables. Samples marked with white circles correspond to support vectors.

The constraints are updated to include some errors in the training process, in the form of slack variables ξ_i. These measure "how far" the sample $\mathbf{x}_i$ is from the correct decision hyperplane's side, or, in another words, how deep they are on the wrong side. A visual example can be seen in Figure 3.9 [49].

Samples which are on the correct side of the boundary and beyond their classes' canonical hyperplane have $\xi_i = 0$, because their misclassification error is zero. The same applies for samples sitting exactly on their classes' canonical hyperplane. Because half of the margin's length is 1, samples which are beyond this hyperplane but not on the other classes' region have $0 > \xi_i > 1$. Samples on the other side of the optimal hyperplane have $\xi_i > 1$.

Note that this penalization is linear (see equation 3.36). In this manner, the constraints are updated to include this error:

$$c_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \xi_i, \quad i = 1, 2, \ldots n \tag{3.35}$$

Naturally, $\xi_i \geq 0$ because both the positive and negative cases are taken into account by c_i. Therefore, the sum of the errors for each sample is given by:

$$\sum_{i=1}^{n} \xi_i \tag{3.36}$$

which represents a measure of the degree of misclassified training patterns. To set a ceiling in the training error, a penalization parameter, C, is added to the problem. The inclusion of the variables ξ in the optimization task turns it into:

$$\text{minimize} \quad \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n} \xi_i \tag{3.37}$$

$$\text{subject to} \quad y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \xi_i, \qquad i = 1, 2, \ldots n \tag{3.38}$$

$$\xi_i \geq 0, \qquad i = 1, 2, \ldots n \tag{3.39}$$

The existence of C is required for controlling the amount of misclassified samples and the margin's length, that is, the structural risk. A larger C allows the training error sum to reach a higher value, which makes the SVM more specific and it could, possibly, overfit the data. In this case, the SVM ends up with a smaller margin. On the other hand, forcing a lower ceiling in the error sum makes the SVM more generic. This may underfit the training data and consequently, the SVM gets a larger margin. Unfortunately, this parameter must be carefully chosen to achieve the best model of the data.

After formulating the optimization task, the same steps used in the Hard Margin SVM are used to add the slack variables - the merge of the KKT conditions in a dual problem and solve the Lagrangian. Naturally, we want to minimize the total sum of these penalties, so they are included in the function to be optimized (see equation 3.37).

These steps are not explained here for the sake of simplicity. More details can be found in [5]. In the end the following optimization task is obtained:

$$\text{maximize} \quad \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i\alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j \tag{3.40}$$

$$\text{subject to} \quad \sum_{i=1}^{n} \alpha_i y_i = 0 \tag{3.41}$$

$$\text{and} \quad 0 \leq \alpha_i \leq C, \qquad i = 1, 2, \ldots n \tag{3.42}$$

There is only one difference between this problem and the one in the hard margin SVM - the Lagrange multipliers are now between a lower bound (zero) and an upper bound given by C. The penalization induced by the slack variables ξ_i is not present in this optimization task but indirectly through C.

Finally, the classification is once again accomplished using equation 3.34.

3.5 The kernel SVM

While the Soft Margin SVM can give a good performance with non-linearly separable classes, it can be further improved by working in a different feature space [49, 50, 45]. This allows the SVM to find a decision boundary which better discriminates both classes. The new feature space is of a higher dimension and created using the kernel trick by means of a projection $\mathbf{\Phi}(\mathbf{x})$. Therefore with a non-linear kernel the margin corresponds to a linear boundary in this new feature space. A geometric interpretation can be seen in figure 3.10.

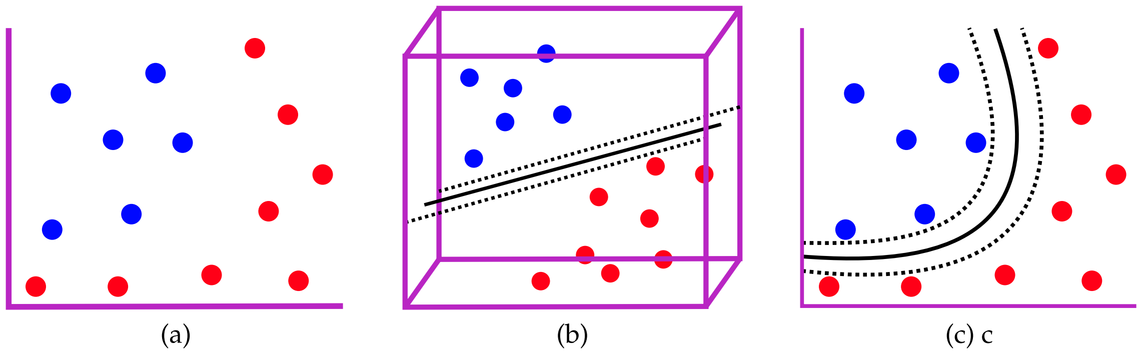


Figure 3.10: The SVM working in a hypothetical higher feature space. (a) A non-linearly separable case. (b) A higher dimensional feature space where the hyperplane can discriminate both classes. (c) The margin projected back into the original feature space.

As said above at the end of section 3.3, the optimization task appears in a form of inner products (see equation 3.30) between samples of indexes i and indexes j. Thus, we only need to compute the dot products and do not require, in the optimization task, the concrete data about samples $\mathbf{x}_i$ and $\mathbf{x}_j$. This allows the SVM to be made non-linear, as shown shortly. The dot products which were previously in the form of $\mathbf{x}_i \cdot \mathbf{x}_j$ can now be written as:

$$K(\mathbf{u},\mathbf{v}) = \langle \mathbf{\Phi}(\mathbf{u}), \mathbf{\Phi}(\mathbf{v}) \rangle \qquad \text{(Mercer's Theorem)} \tag{3.43}$$

$$\text{considering} \quad \mathbf{\Phi}(\mathbf{x}) = \mathbf{x} \tag{3.44}$$

Consequently, the task to be optimized becomes:

$$\text{maximize} \quad \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) \tag{3.45}$$

$$\text{subject to} \quad \sum_{i=1}^{n} \alpha_i y_i = 0 \tag{3.46}$$

$$\text{and} \quad 0 \leq \alpha_i \leq C, \qquad i = 1, 2, \dots n \tag{3.47}$$

Note the presence of C. We are adding the kernel trick to the soft-margin SVM, where we have control over the structural risk. The function to be maximized is a convex one and therefore, a quadratic programming problem like the ones described before. As said above, the inner products are only required to be calculated in the kernel's space. This can be done before the optimization phase and in this case the dot products are stored in matrix known as the Kernel Matrix:

$$K_{n,n} = \begin{bmatrix} \langle \mathbf{\Phi}(\mathbf{x}_1), \mathbf{\Phi}(\mathbf{x}_1) \rangle & \langle \mathbf{\Phi}(\mathbf{x}_1), \mathbf{\Phi}(\mathbf{x}_2) \rangle & \cdots & \langle \mathbf{\Phi}(\mathbf{x}_1), \mathbf{\Phi}(\mathbf{x}_n) \rangle \\ \langle \mathbf{\Phi}(\mathbf{x}_2), \mathbf{\Phi}(\mathbf{x}_1) \rangle & \langle \mathbf{\Phi}(\mathbf{x}_2), \mathbf{\Phi}(\mathbf{x}_2) \rangle & \cdots & \langle \mathbf{\Phi}(\mathbf{x}_2), \mathbf{\Phi}(\mathbf{x}_n) \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle \mathbf{\Phi}(\mathbf{x}_n), \mathbf{\Phi}(\mathbf{x}_1) \rangle & \langle \mathbf{\Phi}(\mathbf{x}_n), \mathbf{\Phi}(\mathbf{x}_2) \rangle & \cdots & \langle \mathbf{\Phi}(\mathbf{x}_n), \mathbf{\Phi}(\mathbf{x}_n) \rangle \end{bmatrix}$$

Each element of the kernel matrix is a dot product between two vectors, $\mathbf{x}_i$ and $\mathbf{x}_j$. Thus, the matrix K is square and with n^2 elements, because there are n vectors $\mathbf{x}$ in the training set. Some examples of kernels are the following:

Linear

$$K(\mathbf{u}, \mathbf{v}) = \langle \mathbf{u}, \mathbf{v} \rangle = \sum_{i=1}^{d} \mathbf{u}_i \mathbf{v}_i \tag{3.48}$$

Polynomial of degree q

$$K(\mathbf{u}, \mathbf{v}) = a \cdot (\langle \mathbf{u}, \mathbf{v} \rangle + b)^q, \qquad q > 0 \tag{3.49}$$

the parameter a represents the function's derivative while b sets the function's offset, i.e., where $a(\cdots)^q = 0$.

RBF

$$K(\mathbf{u}, \mathbf{v}) = e^{-\gamma ||\mathbf{u} - \mathbf{v}||^2} \tag{3.50}$$

the parameter γ controls the bell's aperture. Its length is inversely proportional to the value of γ.

Hyperbolic Tangent or Sigmoid function

$$K(\mathbf{u},\mathbf{v}) = \tanh\left(a \cdot \langle \mathbf{u},\mathbf{v}\rangle + b\right) \tag{3.51}$$

the parameter a represents the variation at which the function increases while b sets the function's offset, i.e., where $\tanh(\cdots) = 0$.

UKF

$$K(\mathbf{u},\mathbf{v}) = L(\|\mathbf{u}-\mathbf{v}\|^2 + \sigma^2)^{-\alpha} \tag{3.52}$$

in the same way as for the RBF kernel, the parameter σ controls the bell's aperture. α affects the decreasing speed around the origin and the constant L is used to normalize the kernel, having usually the value 1 [53]. This kernel aims to gather points near to each other, in a higher dimension space, since they are strongly correlated. Therefore it can provide a smaller number of SVs and thus fasten both the training and classification tasks. Additionally, the UKF kernel can yield better performance generalization [4].

One limitation of the kernel SVM is that there are no practical methods for selecting the best kernel function. Consequently, each classification case must be investigated in order to find an appropriated kernel [48].

The classification is done using a modified version of equation 3.33, because the weight vector $\mathbf{w}$ was calculated in a modified feature space. Hence, the classification is done after projecting vector $\mathbf{x}$ trough the kernel function K. Because we are no longer working on the initial feature space, the classifier is now specified using the Lagrange multipliers α_i. Therefore, the SVM's classification for a given point $\mathbf{z}$ is an extension of the function 3.34 using the kernel trick:

$$y(\mathbf{z}) = \operatorname{sign}\left(b + \sum_{i=1}^{n} \alpha_i y_i K(\mathbf{x}_i, \mathbf{z})\right) \tag{3.53}$$

3.6 The SMO algorithm

Most methods used before the SMO algorithm are slow because they are based on Numerical Optimization libraries, in the form of third-party Quadratic Programming (QP) solvers. Additionally, they require very high amounts of memory to solve the task at hand. There are some better alternatives like working in chunks of training samples [49] or decomposing the problem into a series of smaller QP sub-problems [37, 39]

As a solution for the SVM's existing slow optimization task, Platt developed in 1998 an algorithm which he named Sequential Minimal Optimization(SMO). This algorithm is based on Osuna's decomposition scheme [37] and solves the smallest possible optimization task at each step, updating two α variables. At each step, only two α variables are required to be solved, because the function to be optimized (equation 3.45) has in each instant two Lagrange multipliers in its definition, α_i and α_j. Both multipliers must obey one linear constraint (equation 3.46) and be within the range defined by equation 3.47 [39, 8].

The training process for the soft-margin SMO algorithm with the kernel trick is described below in Algorithm 2, which is a short resume based on Catanzaro's work [8].

Algorithm 2 Sequential Minimum Optimization

```
Require: $\mathbf{x}_i \in \chi$, $y_i \in \Omega$, $i \in \{1 \cdots n\}$
1: Initialize:
   $\alpha_i$=0,
   $f_i$=$-y_i$,
   $i \in \{1 \cdots n\}$
2: Initialize:
   $b_{high} = -1$,
   $b_{low} = 1$,
   $i_{high} = \min\{i : y_i = 1\}$,
   $i_{low} = \min\{i : y_i = -1\}$
3: Update: $\alpha_{ilow}$, $\alpha_{i_{high}}$
4: repeat
5:         Update $f_i$, $i \in \{1 \cdots n\}$
6:         Compute: $b_{high}$, $b_{low}$, $i_{high}$, $i_{low}$
7:         Update $\alpha_{i_{low}}$, $\alpha_{i_{high}}$
8: until $b_{low} \leq b_{high} + 2\tau$
```

Initially, all the α_i are set to 0 as they satisfy the constraints defined in equation 3.47. After choosing i_{high} and i_{low} in each SMO iteration (lines 5 trough 8 in algorithm 2), the new values for the two Lagrange multipliers α_i^{new} are computed as follows:

$$\alpha_{i_{low}}^{\text{new}} = \alpha_{i_{low}} + y_{i_{low}} \frac{b_{high} - b_{low}}{\eta} \tag{3.54}$$

$$\alpha_{i_{high}}^{\text{new}} = \alpha_{i_{high}} + y_{i_{low}} y_{i_{high}} (\alpha_{i_{low}} - \alpha_{i_{low}}^{\text{new}}) \tag{3.55}$$

where η corresponds to the second derivative of the objective function (equation

3.45) and is defined as:

$$\eta = K(x_{i_{high}}, x_{i_{high}}) + K(x_{i_{low}}, x_{i_{low}}) - 2 \cdot K(x_{i_{high}}, x_{i_{low}}) \tag{3.56}$$

η can be non-positive if a given kernel K does not obey Mercer's conditions. As noted by Platt [39] η can be zero if two samples share the same input vector. Because the algorithm always chooses Lagrange multipliers which violate the KKT conditions, the algorithm converges [37, 39].

Both $\alpha_{i_{low}}$ and $\alpha_{i_{high}}$ must be in the range $0 \leq \alpha_i \leq C$. If $\alpha_{i_{low}}$ changes by some amount δ, $\alpha_{i_{high}}$ changes with the same amount on the opposite direction $(-\delta)$, because of the constraint defined in equation 3.46. Next, the KKT conditions must be updated for each sample $\mathbf{x}_i$ using:

$$f_i = \mathbf{w}(\alpha) \cdot \mathbf{z}_i - y_i = \sum_{j=1}^{d} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}_j) - y_i \tag{3.57}$$

which can be simplified to:

$$f_i = f_i^{old} + (\alpha_{i_{high}}^{new} - \alpha_{i_{high}}) y_{i_{high}} K(x_{i_{high}}, x_i) + (\alpha_{i_{low}}^{new} - \alpha_{i_{low}}) y_{i_{low}} K(x_{i_{low}}, x_i) \tag{3.58}$$

Since this algorithm is faster and uses less memory than other QP solving algorithms, the implementations developed in this work use the described method.

The indexes of the next Lagrange multipliers to be updated, i_{low} and i_{high} are chosen from two corresponding sets:

$$I_{low} = \{i : 0 < \alpha_i < C\} \cup \{i : y_i > 0, \alpha_i = C\} \cup \{i : y_i < 0, \alpha_i = 0\} \tag{3.59}$$

$$I_{high} = \{i : 0 < \alpha_i < C\} \cup \{i : y_i > 0, \alpha_i = 0\} \cup \{i : y_i < 0, \alpha_i = C\} \tag{3.60}$$

Naturally, because there is some imprecision involved in the numerical process, these indices are computed using a tolerance ϵ between each Lagrange multiplier and limits 0 and C.

The optimality coefficients b_{low} and b_{high} are calculated as:

$$b_{low} = \max\{f_i : i \in I_{low}\} \tag{3.61}$$

$$b_{high} = \min\{f_i : i \in I_{high}\} \tag{3.62}$$

For simplicity reasons, the mechanism used to chose i_{low} and i_{high} is the first order heuristic from [22]. For the next iteration, these indices are calculated in the following way:

$$i_{low} = \arg\max\{f_i : i \in I_{low}\} \tag{3.63}$$

$$i_{high} = \arg\min\{f_i : i \in I_{high}\} \tag{3.64}$$

The algorithm is executed until the the optimality gap ($b_{low} - b_{high}$) is smaller than threshold 2τ

$$b_{low} \leq b_{high} + 2\tau \Leftrightarrow b_{low} - b_{high} \leq 2\tau \tag{3.65}$$

where τ is a value in the range of $0 < \tau < 1$. After converging, the parameter b can be calculated in two ways: as an arithmetic mean between b_{low} and b_{high} or as a weighted average using the support vectors. As this last method is more precise it is the one used. In this case b is calculated as follows, being n_{NS} the amount of support vectors, i.e., the number of samples having $\alpha_i > 0$:

$$b = \frac{1}{n_{SV}} \sum_{j=1}^{n_{SV}} \left(\sum_{i=1}^{n} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}_j) \right) - y_j \tag{3.66}$$

3.7 Multi-Threaded CPU implementation

In order to study the feasibility of the GPU implementation we first developed a multi-threaded soft-margin kernel SVM. The classifier supports the five kernel functions described in section 3.5 which are the following: linear, polynomial, Gaussian (RBF), sigmoid and UKF kernels.

The program was written in `C++` using OpenMP whose API allows developers to write multi-threaded programs which use shared memory (also named Unified Memory Access (UMA)). Programs written using this approach are in Flynn's taxonomy classified as SPMD because the same program is executed by different threads but each processing a different subset of the data. This corresponds to a modern multi-core computer with a centralized memory (RAM).

The OpenMP library uses the fork-join model of parallel execution where programs begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered. Shortly, the execution is split by multiple threads and in the end of the region the master threads waits for the arrival of the other threads.

Even though the execution of the SMO algorithm is composed of sequential tasks our emphasis is that some of the tasks can be safely parallelized. In fact, these steps match the areas where most of the computation is done and, therefore, where the algorithm consumes more time [24, 8]. One of such steps can be easily identified as the KKT's conditions update (using f_i and the kernel Gram matrix), which is the costliest arithmetic step in the SMO algorithm, as noticed in [24]. This corresponds to what is known as an embarrassingly parallel task, because the calculus of each f_i is independent of each other. Consequently, it fits nicely to the multi-core CPU architecture.

The computation of the next b_{low}, b_{high}, $\imath_{low}$ and $\imath_{high}$ variables are done in the first order heuristic search, which can be executed in parallel using the well-known reduction operations *min* and *max*. Each thread works on a subset of both I_{high} and I_{low} and finds locally the variables specified in the beginning of this paragraph. The master thread waits for the other thread's results and then applies the reduction operators. The calculus of offset b is also implemented in parallel for each SV. The only SMO which are not parallelized are the Lagrange multipliers (α_{ilow} and $\alpha_{i_{high}}$) update and the convergence verification (step 5 on algorithm 2).

From equation 3.53 it can be concluded that the classification is itself inherently parallel as the classification of each sample is independent of the others. In this implementation each thread classifies a group of points, equally distributed between the threads (if possible).

The above parallel tasks are divided into equal parts, each one assigned to a corresponding thread. In theory, if the original single-threaded SMO training process takes T_s time, using a processor with P cores, the multi-threaded SMO would execute in $T_p = \frac{T_s}{P}$ and offer a speedup of $P\times$. However, this theoretical speedup is rarely achieved in practice because part of the code is not parallelized. Even though the algorithm can be fully parallel, the sequential sections always exist (Amdahl's law) due to: (i) synchronization, where some threads must wait for the completion of others, (ii) memory bandwidth, which is shared by all CPU cores, and (iii) mutual exclusion areas, among other reasons.

3.8 GPU implementation

The GPU implementation is based on the multi-threaded CPU version and executes the same SMO steps in parallel. It was written in C++ using both the API CUDA and supporting classes provided by the library GPUMLib.

Being the GPUs massively parallel processors with many cores the multi-threaded algorithm was modified in order to take advantage of both the greater computing power and memory bandwidth available in modern GPUs. Therefore, the implementation has multiple threads running where each thread works on an independent sample and is handled by the SMs. The aim of our work is to have enough threads executing in parallel handled in order to maximize the SMs occupancy, unless there is not enough data to fill all the SMs. The worst case scenario is when the SMs have a minimal amount of calculus to be done and are mostly waiting for their data to be fetched from the shared memory or the GPU RAM. The major example of this bottleneck is the first order heuristic because it cycles trough all the elements of the training data in order to find a minimum and maximum value (constrained to the sets in 3.6). Naturally, this is a memory bandwidth bound step [36].

For the remaining SMO steps and to minimize thread divergence, if possible, each thread follows the same path as the remaining threads of its block (executes the same instruction). In case the thread's result is not needed, it is simply discarded. Also, to maximize the performance of each SMO step this implementation makes use of C++'s templates so that there is an optimized version of each function depending on the program's execution.

The classifier is also parallelized. Unlike the multi-threaded CPU implementation where each thread classifies a equally sized group of samples, in the GPU version each thread classifies a single sample. This is efficient because all the threads in a block (having increasing IDs) classify sequential patterns and therefore work in sync, executing the same instruction in their warp and minimizing thread divergence. For datasets with a higher number of features the speed of the training algorithm will be limited by the SPs computing speed and on the other hand, datasets with a smaller number of features will be limited by the GPUs's memory bandwidth because there is a smaller computation to be done in each SMO step.

In order to minimize the overhead incurred by checking the convergence of the SMO algorithm, the implementation makes use of kernel streams. Streams are fluxes of execution which can be done in parallel by both the GPU driver and the hardware. Because the SMO algorithm is sequential, the execution of its steps is done entirely in one stream (stream 1 in figure 3.11). In parallel, the CPU uses another stream (stream 6 in figure 3.11 which corresponds to the transfer of data from the GPU device to the host) to periodically retrieve some variables from the GPU's memory. These variables are required to check if the learning process has converged. Consequently, the SMO algorithm occupies most of the GPU's time while being minimally disturbed for checking its convergence.

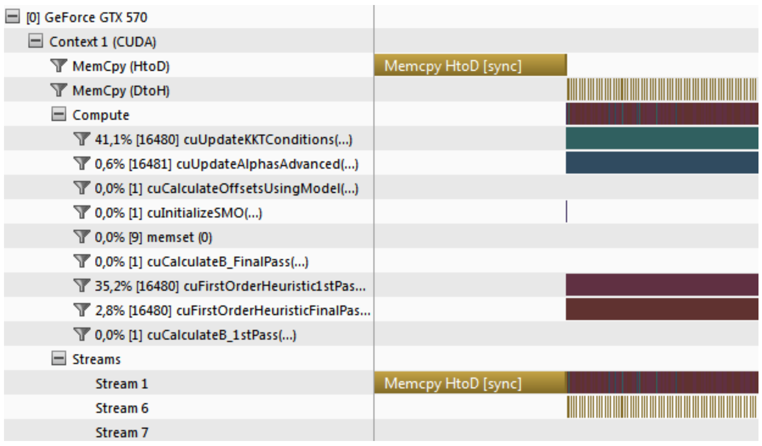

[0] GeForce GTX 570
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
41,1% [16480] cuUpdateKKTConditions(...)
0,6% [16481] cuUpdateAlphasAdvanced(...)
0,0% [1] cuCalculateOffsetsUsingModel(...)
0,0% [1] cuInitializeSMO(...)
0,0% [9] memset (0)
0,0% [1] cuCalculateB_FinalPass(...)
35,2% [16480] cuFirstOrderHeuristic1stPas...
2,8% [16480] cuFirstOrderHeuristicFinalPas...
0,0% [1] cuCalculateB_1stPass(...)
Streams
Stream 1
Stream 6
Stream 7
Memcpy HtoD [sync]
Memcpy HtoD [sync]

Figure 3.11: Profiling of the SMO algorithm executing on the GPU. The algorithm executes on stream 1 while its convergence is verified on stream 6.

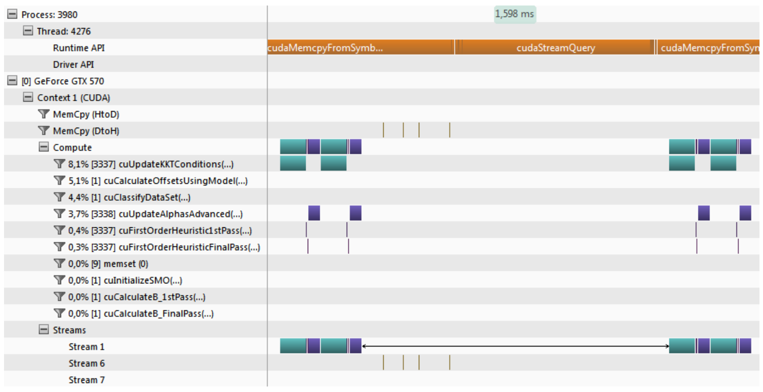

Process: 3980
Thread: 4276
Runtime API
Driver API
[0] GeForce GTX 570
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
8,1% [3337] cuUpdateKKTConditions(...)
5,1% [1] cuCalculateOffsetsUsingModel(...)
4,4% [1] cuClassifyDataSet(...)
3,7% [3338] cuUpdateAlphasAdvanced(...)
0,4% [3337] cuFirstOrderHeuristic1stPass(...)
0,3% [3337] cuFirstOrderHeuristicFinalPass(...)
0,0% [9] memset (0)
0,0% [1] cuInitializeSMO(...)
0,0% [1] cuCalculateB_1stPass(...)
0,0% [1] cuCalculateB_FinalPass(...)
Streams
Stream 1
Stream 6
Stream 7
1,598 ms
cudaMemcpyFromSymb...
cudaStreamQuery
cudaMemcpyFromSyn

Figure 3.12: Overhead of both the data requests and query status from the GPU.

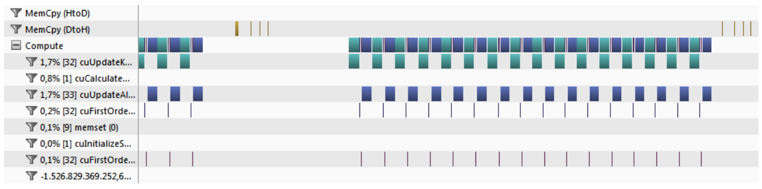


Figure 3.13: Using a batch of 16 iterations for the SMO algorithm.

An additional improvement was done after noting that while still using multiple streams the algorithm did not fully fill the GPU usage. As can be seen in figure 3.12 either when the CPU (host) requests asynchronously the transfer of the required variables or queries their delivery the next SMO step is executed on average 1.6 ms later (using a NVIDIA GeForce GTX 570 on a Windows 7 x64 operating system). Naturally, when compared with the time spent on each step of the algorithm this unused time may be much larger, depending on the dataset. Because the algorithm's convergence must be verified and its execution is done on the GPU device, the solution used to maximize the GPU occupancy is to execute more steps of the SMO algorithm while requesting less data from the GPU. Consequently, we execute a "batch" n of iterations and only then the convergence is verified. After some experiments we concluded that an acceptable value for n is 16 (figure 3.13). Naturally, this forces the number of iterations done in our GPU implementation to be a multiple of n which may increase the time spent by the algorithm. However, doing more iterations of the algorithm decreases the optimality gap and improves the classifier performance. The parameter n can be specified by the user.

3.9 Existing GPU SVM implementations

To date, there are four implementations of SVM for the GPU: Catanzaro's gpuSVM [8], Herrero's multiSV [20], Carpenter's cuSVM and Lin's sparse SVM [25]. All of these are written in CUDA for NVIDIA's GPUs.

This work follows the first implementation of a SVM classifier using programmable GPUs, named "gpuSVM" and actually a binary classifier. This implementation was largely the work of Catanzaro, Sundaram and Keutzer and was developed targeting the first programmable GPU, the G80 (GeForce 8800 GTX). It explores the heaviest compute bound step of the SMO algorithm, the update of KKT conditions. Also done on the GPU is the search for the next Lagrange multipliers to be updated, using a first order heuristic. Their implementation also makes use of a second order heuristic from [14] which tries to choose the next Lagrange multipliers so that these may cause a higher change in the objective function. However, this heuristic's computation may take some time and consequently, the algorithm may take a performance hit. Finally, their algorithm caches the most calculated kernel dot-products in RAM in order to minimize the amount of calculus. Compared to the well known LIBSVM software, they had a performance increase in the range of 9× to 35× for the training implementation. For the classifier, they parallelized the kernel dot product between the support vectors and the testing samples using the CUBLAS library, all in a single step. Then a sum-reduce operation is done for each testing sample (equation 3.53). Using this approach, their speedup was in average about 110× [8].

Another implementation, this one supporting multiple classes, is Herrero et al's "multiSVM". Their work is very similar to Catanzaro's, but it executes different binary classifiers at the same time on the GPU. Additionaly, i uses NVIDIA's CUBLAS algebra library to help calculating the kernel matrix.

An implementation which is also largely based upon "gpuSVM" is "cuSVM" by Carpenter. The major improvement to Catanzaro's work is the usage of mixed precision floating point arithmetic. In "cuSVM" most computations are done in 32-bit precision (float) but some computations like the sum of dot products are done and stored in double precision floating point (64-bit double). According to the author, this can be of extreme importance for some data sets, like the "Forest" data set, which in fact doesn't appear in Catanzaro's results. However, this requires the support of double precision floating point from the GPU, something which is not available in older GPUs.

Finally, Lin et al proposed the usage of sparse matrices for cache the kernel matrix. Their work is also very similar to Catanzaro's SVM. They claim an speedup over "gpuSVM" of 1.9× to 2.41×. However, in one dataset, Usps, they had a slower result than Catanzaro's implementation. It seems that on their case, the dataset has an impact on the program's training time. As they state on their article, the sparse matrix structure is affected by the dataset. Also, the non-existence of efficient sparse matrix-matrix multiplication algorithms means their implementation could be better.

3.10 Conclusions

In this Chapter we thoroughly reviewed Support Vector Machines (SVMs) from the standpoint of making possible an implementation. There are many implementations of SVMs both in CPU and GPU. However, one main driver of this Dissertation is to incorporate the GPU-SVM component software in the GPUMLib Software which is Open Source available. The main advantage of our implementation will be its open source architecture allowing researchers and practitioners to use and extend it for research or any other suitable purposes. Besides we added a new kernel which brings an increased value to this component from the scientific point of view. As expected, it will be possible to run with improved performance large scale problems such as those occurring in bioinformatics, biometrics, image processing and web mining.

Chapter 4

Experimental results

In this chapter we present the validation results obtained with our multi-threaded CPU and GPU SVM implementations. In section 4.1 we define the metrics used to assess both the classifier's speed and performance. The datasets used in the experiments are described in section 4.2. Eight of the datasets were taken from the UCI Machine Learning repository, available at `http://archive.ics.uci.edu/ml/`. The "spiral" dataset was produced exclusively for this dissertation, in order to test the UKF kernel performance. The MP3 Steganalysis dataset was extracted from a real problem using the four methods described in [40]. This dataset is composed of two classes: normal MP3 audio files (cover) and the same MP3 files with hidden information (stego). The Peptidases dataset is also a real problem consisting in the prediction of peptidases and non-peptidases proteins [27]. Section 4.3 contains the configuration used for each dataset. In section 4.4 we present the results in the form of tables and graphs. Finally, in section 4.5 the results are discussed and we give a short conclusion about our GPU implementation.

4.1 Evaluation metrics

In this section we define the metrics used in Machine Learning to assess the classifier's performance.

Cross validation is an evaluation method which helps to decide the optimal parameters for a given classifier and estimate how well the classifier will respond when it is required to make new predictions for data which it was not trained for. Because in real applications we only have access to a limited number of samples, the methods used to assess the classifier's performance will have an impact on its future performance [17, 23]. There are many kinds of validation algorithms, some of which are summarized in the following sections.

Holdout

The holdout is the simplest validation method. From the initial dataset, two subsets are created, one for training and another for testing. It has the problem of in the case we have a small dataset, it may be difficult to put aside a significant part of the data exclusively for validating the classifier.

Random Sub-sampling

This method, also known as Monte Carlo cross-validation, is a repeated version of the holdout. It randomly selects individual samples to build the training set and without replacement, it assigns different samples to the test set. After the training and testing set are build, the classifier is evaluated. The final classifier performance is obtained from the average results obtained with each run.

K-Fold

The K-Fold cross-validation is similar to the random sub-sampling validation. Being k the number of folds in each training phase, the training dataset has $\frac{K-1}{K}$ elements from the initial dataset. The remaining $\frac{1}{K}$ of the dataset are used as the testing set so that the classifier's performance can be measured. This process is repeated K times while the folds rotate and get attributed to either the training or the testing sets (figure 4.1). The classification error is averaged through all K runs and consequently its standard deviation decreases as K is increased. In short, cross-validation allows us to figure how the classifier will generalize in a new dataset [44]. When compared to random sub-sampling, K-Fold eventually uses all the samples for both the training and testing sets, which doesn't happen with sub-sampling because of its random nature.

The choice of the number of folds K naturally has an impact on the classifiers' performance. A larger number of folds will allow a good estimative of the classifiers performance on the real world. However this estimative will have a larger variance because many test sets may fall outside of the model learned by the classifier and be incorrectly predicted. And naturally, more folds imply more executions of the training and classification tasks which could make the validation procedure costly.

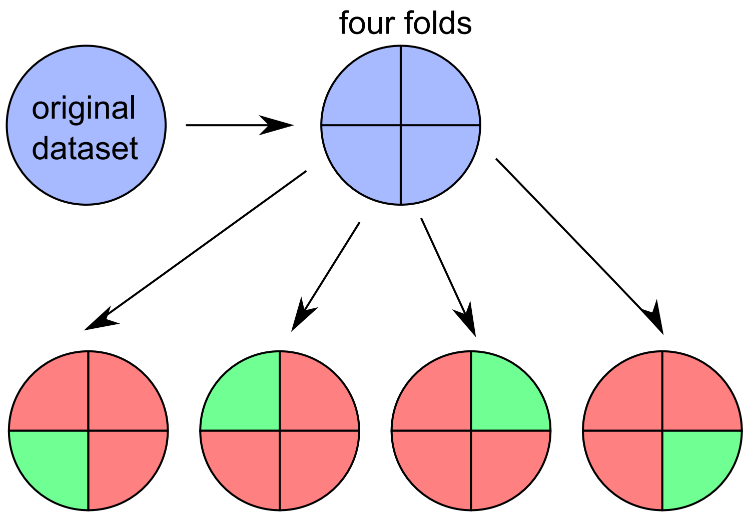


Figure 4.1: Example of “K-Fold” cross-validation using four folds. The green folds correspond to the testing set while the red folds correspond to the training set.

On the other hand, a smaller number of folds presents opposing advantages and disadvantages. The computation time done by the validation procedure and the variance of the estimator (performance) will be both smaller when compared to using a higher number of folds. However, the variance of the testing error will also be smaller and therefore more conservative, which may not reflect what would happen in a real problem.

In practice, the number of folds depends on the size of the problem, including the complexity of the dataset and training. According to literature, typical values for K are 5 and 10 [17, 23].

Leave-one-out

This method can be seen as an extreme version of the K-Fold validation, when K is equal to the number of samples n. Thus, the training set is always comprised of $n-1$ samples and the test set composed of a single sample. As for the K-Fold technique, the process is repeated n times. This validation method has the clearly disadvantage of requiring the most training time and therefore of running the experiments.

Performance metrics

The performance metrics are defined using the well known Confusion Matrix, shown in table 4.1. To create this matrix, the classifier is trained with the training set and later is given the testing set, where it predicts a classification based on the testing set. The first row of the matrix lists the possible values for the actual (desired) class and in the first column we have the same values for the prediction. Therefore, the elements of the confusion matrix compare each positive or negative prediction against the same two possibilities of the actual class. When the classifier's result is predicted correctly, the outcome is either a true positive or a true negative, depending on the class. On the other hand, if the classifier misclassifies the actual class, we have a false outcome.

		Actual class	
		Negative	Positive
Predicted class	Negative	True Negative (TN)	False Negative (FN)
	Positive	False Positive (FP)	True Positive (TP)

Table 4.1: Confusion Matrix.

The performance metrics, calculated from the confusion matrix are presented next.

- Precision

  Precision shows the proportion of the correctly predicted positive cases relative to all the positives:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{4.1}$$

- Recall

  The recall, sensitivity or True Positive Rate (TPR) represents how many positives the classifier did correctly predict. That is, a high recall means the classifier could have given some false positives, but classified correctly most of the positives.

$$\text{Recall} = \frac{\text{TP}}{\text{FN} + \text{TP}} \tag{4.2}$$

- False Positive Rate (FPR)

FPR is the proportion of how many incorrect positives did the classifier predict in all positive predicted cases:

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TP}} \tag{4.3}$$

- Specificity

  The Specificity or True Negative Rate (TNR) refers to how accurately the classifier did predict the positives without giving false positives.

$$\text{Sensitivity} = \frac{\text{TN}}{\text{TN} + \text{FP}} = 1 - \text{FPR} \tag{4.4}$$

- False Discovery Rate (FDR)

  The FDR is the expected proportion of false positives among all predicted positive cases:

$$\text{FDR} = \frac{\text{FP}}{\text{FP} + \text{TP}} \tag{4.5}$$

- Accuracy

  The accuracy represents how many predictions were in fact correct:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{4.6}$$

- F-score

  The F-score is a harmonic mean of the precision and recall. It is similar to accuracy.

$$F1 = \frac{2 \cdot \text{Recall} \cdot \text{Precision}}{\text{Recall} + \text{Precision}} \tag{4.7}$$

A different metric, shown in equation 4.8 is needed to measure the performance of the parallel implementation in relation to the sequential version. This metric, the speedup, which is basically the reduction in time done by the parallel version against the sequential:

$$\text{Speedup} = \frac{\text{T}_{\text{sequential}}}{\text{T}_{\text{parallel}}} \tag{4.8}$$

Usually all algorithms which focus in improving speed over an existing version have their speedup shown using the above formula.

When comparing SVM implementations it is also useful to compare the amount support vectors (SVs) generated. The least the implementation uses while still having a good classification performance, the better.

4.2 Datasets

Both our multi-threaded CPU implementation and the GPU version are currently binary. Therefore, we use two-class datasets for the experiments. Table 4.2 presents the main characteristics of the chosen datasets, where the last two rows correspond to the real datasets from [27, 40]. Before running the experiments, all of the datasets were normalized using standard score normalization.

Dataset	#Samples	#Features	#Classes
Adult	32561	14	2
Breast Cancer	569	30	2
German	1000	59	2
Haberman	306	3	2
Heart	270	20	2
Ionosphere	351	34	2
Sonar	208	30	2
Tic-tac-toe	958	9	2
Spiral	2097152	2	2
Peptidases	20778	24	2
MP3 Steganalysis (stego)	1994	742	2

Table 4.2: Datasets used in the validation of the GPU implementation.

With the exception of the MP3 Steganalysis, the Spiral and the Peptidases, the remaining datasets were obtained at the UCI repository. The UCI datasets are summarized below:

- The task of the adult dataset consists of predicting if a person as an annual income greater than 50,000 dollars per year based on attributes such as age, sex, job, education, etc. This dataset was extracted from the 1994 census database.
- The Breast cancer dataset aims to discriminate between benign and malignant diagnosis. It is composed of samples taken from pictures of breast

masses. Each feature corresponds to a characteristic of the cell nuclei present in the pictures.

- The German dataset classifies credits attributed to individuals as good or bad (risky) credits, using characteristics as the person's salary, employment, possessed goods, age, type of housing, etc. The name (german) comes from the fact it is a study done in Germany.
- The Haberman dataset contains cases from study conducted on the survival of patients who had undergone surgery for breast cancer.
- The Heart dataset is similar to the "Breast Cancer" dataset but related to the diagnosis of heart problems.
- The Ionosphere dataset relates the evidence of some kind of structures in the ionosphere using data retrieved from 16 high-frequency antennas.
- The Sonar dataset contains patterns composed of signals extracted from the reflection of radio waves on two types of objects: metal cylinders and rocks. Each individual signal corresponds to the intensity of a frequency present in the received radio wave after being reflected by the target. The aim of the dataset is to detect the type of the target.
- The Tic-tac-toe dataset encodes the complete set of possible board configurations at the end of various games when the "X" player starts the game and wins in the end. Naturally, the game finishes when player "X" makes three-in-a-row.

The Two-Spiral dataset was produced in order to assess the UKF kernel efficiency. The main task consists of learning to discriminate between data distributed on two distinct spirals that coil around each other in the x-y plane. As this dataset has two features it can be easily understood using a graphical plot (see Figure 4.2).

The peptidases dataset consists of discriminating between peptidases and non-peptidases. Protein classification into functional and structured groups is an important task in the understanding of the inner biological cell functioning. Peptidases are a class of enzymes that catalyze chemical reactions, allowing the decomposition of protein substances into smaller molecules. The data composing this dataset was extracted using the mechanisms described in Lopes et al. [27] and kindly provided by the authors.

Lastly, we used a real dataset produced by the authors for a study regarding stegalaysis - the MP3 Steganalysis dataset. This field comprises techniques

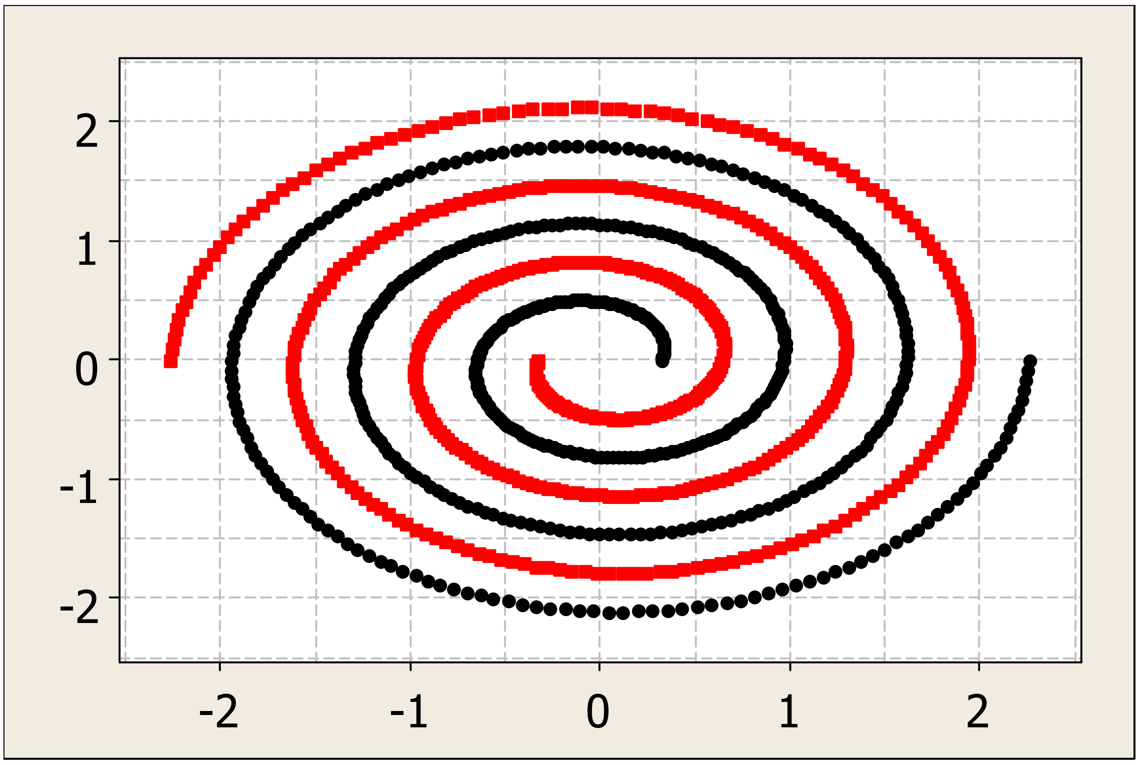


Figure 4.2: Scatter plot of the dataset "Spiral".

used to hide and detected confidential information behind innocent data. As the dataset name suggests, the dataset was extracted from compressed MP3 audio files divided in two groups: original MP3 audio files (cover) and files with hidden information (stego). The data was extracted from a real problem using the methods described in [40].

The datasets were normalized before being used in the experiments using the standard score procedure which normalizes each sample ($\mathbf{x}$) according to the population's mean ($\bar{\mathbf{x}}$) and standard deviation (σ) as follows:

$$\mathbf{x}_i = \frac{\mathbf{x}_i - \bar{\mathbf{x}}}{\sigma} \tag{4.9}$$

In machine learning, normalization is very important in order to minimize the impact that features, with a wide range of values may have upon the learning algorithm. Using normalized data, SVMs are able to achieve a better performance, because the kernel's parameters are independent of the features. For example, when using the RBF kernel, the user cannot specify a different σ for each feature. Therefore all features should have the same range. This can be accomplished by normalizing.

Dataset	C	γ	τ
Adult	1.0	0.100	0.01
Breast Cancer	3.0	0.050	0.01
German	1.0	0.050	0.01
Haberman	1.0	1.000	0.01
Heart	0.1	0.050	0.01
Ionosphere	1.0	0.500	0.01
Sonar	3.0	0.050	0.01
Tic-tac-toe	3.0	0.250	0.01
Spiral	0.56	11.30	0.01
Peptidases	0.1	0.250	0.01
MP3 Steganalysis	1.0	0.001	0.01

Table 4.3: Configuration used in the RBF kernel.

4.3 Experimental setup

We ran the experiments 10 times for each dataset which were normalized using standard score normalization. We used a 5 K-Fold cross validation. Consequently, each classifier was run 50 times.

As our implementation doesn't make use of a kernel cache we set LIBSVM's cache to one MegaByte. This was the only way to minimize LIBSVM's cache usage as it does not have an option to completely disable it. However, one MegaByte of cache can still be a significant improvement. LIBSVM is a purely sequential program, consequently it has no options to change the amount of threads. The LIBSVM version we used was 3.12. For our CPU implementation, we set the number of threads to 4 so that each thread can be assigned to one of the four cores of the system's CPU.

Unless otherwise specified, the three classifiers used the RBF (Gaussian) kernel. All the parameters, including those used in the UKF kernel, were chosen using a grid search in order to obtain the best classification performance from the classifiers. This test was done to compare the generalization capability of the UKF kernel. These are shown in tables 4.3 and 4.4. The numerical tolerance ϵ used in the first order heuristic was set to 1×10^{-5} and the optimality gap τ to 0.01. For comparing the UKF against the RBF kernel we used our Multi-Threaded CPU SVM in order to have the same heuristic search and algorithms for both kernels.

The system used to extract the results had 12 GB of RAM and an Intel Quad Core i5-750 (3.33 GHz) processor. As a result of this CPU having Intel's Turbo Boost technology, which changes the processor's speed according to the proces-

Dataset	C	L	σ	α
Adult	0.16	0.64	0.64	0.01
Breast Cancer	6.10	6.10	6.10	0.16
German	0.16	10.24	0.02	0.16
Haberman	6.10	0.06	0.01	0.16
Heart	6.10	0.97	0.06	0.01
Ionosphere	5.25	3.50	5.25	0.04
Sonar	5.25	1.50	5.25	0.53
Tic-Tac-Toe	2.44	6.10	2.44	0.39
Spiral	0.10	5.15	0.50	7.78
Peptidases	2.56	10.24	0.16	0.04
MP3 Steganalysis	2.56	10.24	10.24	0.64

Table 4.4: Settings used in the UKF experiments.

sor's load, that technology was disabled in order to have the same clock speed throughout all the experiments. The machine has a NVIDIA GeForce 570 GTX GPU. Its characteristics are described on table 4.5.

Table 4.5: NVIDIA GeForce 570 GTX characteristics

Number of SPs	480
IEEE single precision (float) performance	748.8 GFlops
Number of SMs	15
Shading clock speed	1.56 GHz
Memory size	1.25 GB
Memory bandwidth	152.0 GB/s
Shared memory per block	48 KB

One important difference between the GPU-SVM and the other classifiers (CPU-SVM and LIBSVM) is that the GPU version only uses single precision (32-bit) floating point data types because of architectural limitations.

4.4 Results

In this section we present the classification performance obtained using the three classifiers for all the datasets. We also compare differences in performance between the GPU version and LIBSVM training times for the three heaviest datasets (adult, spiral and mp3 steganalysis).

The next tables present the classification performance for each classifier and dataset followed by a table comparing the number of SVs, F-Score and iterations between the GPU implementation and LIBSVM. The two last results compare the speedup obtained by our GPU implementation against LIBSVM in both classification and training tasks.

Adult dataset

	Accuracy (%)	Precision (%)	Recall (%)	F-Score (%)	nSVs
CPU-SVM	84.65±0.39	88.01±0.42	92.37±0.42	90.13±0.28	**9781.76** ±56.38
GPU-SVM	80.97±0.80	**91.89**±0.64	82.19±1.45	86.76±0.67	9801.94 ±56.43
LIBSVM	**84.72**±0.38	86.62±0.30	**94.47**±0.36	**90.38**±0.24	9788.40 ±48.90

Table 4.6: Performance results for the Adult dataset.

Breast Cancer dataset

	Accuracy (%)	Precision (%)	Recall (%)	F-Score (%)	nSVs
CPU-SVM	97.48±2.26	96.95 ±2.68	**96.32**±2.92	96.59±1.90	**113.28**±5.18
GPU-SVM	95.42±2.02	**100.00**±0.00	87.64±5.45	93.32±3.12	115.46±4.85
LIBSVM	**97.76**±1.49	97.93 ±2.26	96.09±3.10	**96.96**±2.02	114.44±4.89

Table 4.7: Performance results for the Breast Cancer dataset.

German dataset

	Accuracy (%)	Precision (%)	Recall (%)	F-Score (%)	nSVs
CPU-SVM	**73.61**±1.65	74.40±0.98	95.00±1.94	83.44±1.08	**713.70**±5.53
GPU-SVM	73.28±2.90	**80.04**±1.73	82.41±4.14	81.15±2.36	718.94±5.04
LIBSVM	73.03±1.32	73.11±0.76	**97.24**±1.27	**83.46**±0.82	718.74±5.14

Table 4.8: Performance results for the German dataset.

Haberman dataset

	Accuracy (%)	Precision (%)	Recall (%)	F-Score (%)	nSVs
CPU-SVM	71.85±4.42	**77.17**±2.59	87.99±5.64	82.14±3.13	**149.14**±4.84
GPU-SVM	72.72±4.06	76.51±2.31	**91.05**±4.12	83.11±2.66	152.42±4.55
LIBSVM	**72.92**±3.50	75.55±1.99	93.39±4.21	**83.49**±2.31	151.20±4.41

Table 4.9: Performance results for the Haberman dataset.

Heart dataset

	Accuracy (%)	Precision (%)	Recall (%)	F-Score (%)	nSVs
CPU-SVM	83.18±4.64	83.33±5.04	88.00±5.97	**85.44**±4.09	**175.74**±3.09
GPU-SVM	**83.33**±4.92	**87.15**±5.01	82.74±6.61	84.73±4.70	178.28±2.98
LIBSVM	82.37±4.96	79.72±5.33	**91.95**±4.38	85.30±4.00	177.18±3.10

Table 4.10: Performance results for the Heart dataset.

Ionosphere dataset

	Accuracy (%)	Precision (%)	Recall (%)	F-Score (%)	nSVs
CPU-SVM	**89.66**±3.38	98.49±1.57	85.02±5.37	**91.16**±3.13	**215.10**±3.08
GPU-SVM	67.52±2.00	66.23±1.35	**99.60**±0.85	79.55±1.06	218.72±3.20
LIBSVM	89.06±3.58	**98.58**±1.51	84.22±5.64	90.72±3.29	217.74±3.23

Table 4.11: Performance results for the Ionosphere dataset.

Sonar dataset

	Accuracy (%)	Precision (%)	Recall (%)	F-Score (%)	nSVs
CPU-SVM	85.77±4.90	82.32±5.67	93.47±6.84	87.30±4.39	**151.10**±2.57
GPU-SVM	**88.16**±4.57	**92.20**±5.15	85.07±9.42	**88.06**±5.18	154.82±2.39
LIBSVM	84.65±4.78	80.93±5.65	**94.18**±6.15	86.83±4.00	153.74±2.36

Table 4.12: Performance results for the Sonar dataset.

Tic-Tac-Toe dataset

	Accuracy (%)	Precision (%)	Recall (%)	F-Score (%)	nSVs
CPU-SVM	97.70±1.22	96.72±1.72	**99.90**±0.26	98.28±0.90	**548.36**±10.08
GPU-SVM	**98.98**±0.88	**98.98**±1.10	99.47±0.74	**99.22**±0.67	553.18±10.58
LIBSVM	97.72±1.26	96.74±1.77	**99.90**±0.26	98.29±0.93	551.78±10.89

Table 4.13: Performance results for the Tic-Tac-Toe dataset.

Spiral dataset

	Accuracy (%)	Precision (%)	Recall (%)	F-Score (%)	nSVs
LIBSVM	100.00±0.0	100.00±0.0	100.00±0.0	100.00±0.0	939.86±58.23
CPU-SVM	100.00±0.0	100.00±0.0	100.00±0.0	100.00±0.0	**698.80**±28.65
GPU-SVM	100.00±0.0	100.00±0.0	100.00±0.0	100.00±0.0	1053.10±67.49

Table 4.14: Performance results for the Spiral dataset.

Peptidases dataset

	Accuracy (%)	Precision (%)	Recall (%)	F-Score (%)	nSVs
CPU-SVM	88.93±0.15	88.71±0.13	**99.99**±0.01	94.02±0.07	6467.36±18.49
GPU-SVM	**96.25**±0.24	**97.57**±0.23	98.13±0.31	**97.85**±0.14	6849.42±23.73
LIBSVM	96.04±0.23	96.63±0.26	98.90±0.17	97.75±0.13	**3829.62**±17.31

Table 4.15: Performance results for the Peptidases dataset.

MP3 Steganalysis dataset

	Accuracy (%)	Precision (%)	Recall (%)	F-Score (%)	nSVs
CPU-SVM	**97.05**±0.87	**96.87**±1.17	97.26±1.28	**97.06**±0.88	**346.16**±7.11
GPU-SVM	96.07±1.29	93.75±2.23	**98.80**±0.74	96.19±1.21	348.82±7.06
LIBSVM	96.92±1.00	96.40±1.42	97.50±1.24	96.94±0.99	348.08±7.16

Table 4.16: Performance results for the MP3 Steganalysis dataset.

Number of Support Vectors (SVs), F-Score and SMO iterations comparison

	nSVs (%)	F-Score (%)	Iterations (%)
Adult	-0.89↓	-3.90↓	-29.98↓
Breast	-0.14↓	-4.17↓	-40.66↓
German	-0.03↓	-2.85↓	-5.55↓
Haberman	-0.81↓	-0.45↓	-50.49↓
Heart	-0.62↓	-0.67↓	-1.69↓
Ionosphere	-0.45↓	-14.05↓	-14.96↓
Sonar	-0.70↓	1.40↑	-11.87↓
Tic-tac-toe	-0.25↓	0.94↑	-17.12↓
Spiral	-33.64↓	0.0=	-54.05↓
Peptidases	-78.85↓	0.10↑	-49.03↓
Steganalysis	-0.21↓	-0.78↓	-33.17↓
Mean	-10.60↓	-2.21↓	-28.052↓

Table 4.17: Improvements on the amount of Support Vectors, F-Score and SMO iterations of the GPU version compared to LIBSVM. A negative value indicates that our GPU obtained lower results which is also graphically shown with a decreasing arrow.

Adult dataset speedup

Classifier	Classification (s)	Training (s)
CPU-SVM	0.84±0.12	14.83±0.42
GPU-SVM	**0.03**±0.01	**2.24** ±0.22
LIBSVM	2.02±0.07	30.49±0.72
CPU improvement over LIBSVM	↑2.27×	↑2.06×
GPU improvement over LIBSVM	↑67.33×	↑13.61×

Table 4.18: Speedup and iterations taken by the classifiers for the Adult dataset.

MP3 Steganalysis dataset speedup

	Classification (s)	Training (s)
CPU-SVM	**0.02**±0.01	**0.34**±0.02
GPU-SVM	0.06±0.01	0.68±0.07
LIBSVM	0.57±0.02	2.37±0.05
CPU improvement over LIBSVM	↑29.53×	↑6.88×
GPU improvement over LIBSVM	↑9.5×	↑3.48×

Table 4.19: Speedup and iterations taken by the classifiers for the MP3 Steganalysis dataset.

Spiral dataset speedup

	Classification (s)	Training (s)
CPU-SVM	3.02±0.50	21.26±0.19
GPU-SVM	**0.04**±0.11	**0.89**±0.01
LIBSVM	11.72±0.19	146.72±0.42
CPU improvement over LIBSVM	↑3.88×	↑6.90×
GPU improvement over LIBSVM	↑265.85×	↑165.15×

Table 4.20: Speedup and iterations taken by the classifiers for the Spiral dataset.

UKF kernel results

Dataset	Kernel	Accuracy	Precision	Recall	F-Score	nSVs
adult	RBF	**84.65**±0.39	**88.01**±0.42	92.37±0.42	**90.13**±0.28	**9781.76**±56.38
adult	UKF	83.36±0.37	86.12±0.33	**93.09**±0.33	89.47±0.26	12543.60±56.63
german	RBF	**73.61**±1.65	**74.40**±0.98	95.00±1.94	**83.44**±1.08	**713.70**± 5.53
german	UKF	71.79±3.15	71.62±3.41	**98.96**±1.08	83.04±2.08	795.60±13.89
breast	RBF	97.48±1.41	96.95±2.68	96.32±2.92	96.59±1.90	113.28±5.18
breast	UKF	**98.11**±1.00	**98.33**±2.11	**96.51**±1.91	**97.39**±1.46	**63.08**±3.03
haberman	RBF	71.85±4.42	**77.17**±2.59	87.99±5.64	82.14±3.13	**149.14**±4.84
haberman	UKF	**72.45**±4.91	76.06±4.67	**91.31**±5.13	**82.85**±3.58	231.08±4.03
heart	RBF	**83.18**±4.64	83.33±5.04	**88.00**±5.97	**85.44**±4.09	**175.74**±3.09
heart	UKF	82.93±3.59	**83.57**±6.35	86.83±6.00	84.85±3.41	181.52±5.52
ionosphere	RBF	89.66±3.38	**98.49**±1.57	85.02±5.37	91.16±3.13	215.10±3.08
ionosphere	UKF	**94.28**±3.04	93.68±3.59	**97.70**±2.20	**95.61**±2.30	**101.40**±3.99
sonar	RBF	**85.77**±4.90	82.32±5.67	**93.47**±6.84	**87.30**±4.39	151.10±2.57
sonar	UKF	85.10±4.81	**84.83**±6.86	88.07±6.14	86.19±4.69	**129.76**±3.84
tic-tac-toe	RBF	97.70±1.22	96.72±1.72	**99.90**±0.26	98.28±0.90	548.36±10.08
tic-tac-toe	UKF	**98.17**±0.94	**97.60**±1.30	99.61±0.78	**98.59**±0.76	**475.64**±10.95
spiral	RBF	100.00±0.00	100.00±0.00	100.00±0.00	100.00±0.00	698.80±28.65
spiral	UKF	100.00±0.00	100.00±0.00	100.00±0.00	100.00±0.00	**324.00**±3.66
peptidases	RBF	88.93±0.15	88.71±0.13	**99.99**±0.01	94.02±0.07	6467.36±18.49
peptidases	UKF	**96.73**±0.25	**97.18**±0.28	99.11±0.17	**98.14**±0.14	**5097.64**±57.52
steganalysis	RBF	**97.05**±0.87	**96.87**±1.17	**97.26**±1.28	**97.06**±0.88	**346.16**±7.11
steganalysis	UKF	93.02±0.78	91.67±1.87	94.68±1.54	93.12±0.83	1019.44±8.51

Table 4.21: UKF vs RBF kernel results for the eleven datasets using our Multi-Threaded CPU SVM.

UKF kernel times and iterations versus RBF kernel

Dataset	Kernel	Classification (s)	Training (s)	Training iterations
adult	RBF	**1.03**±0.06	17.73±0.59	15316.02±325.10
adult	UKF	2.21±0.03	**15.52**±0.16	**6866.16**± 32.37
german	RBF	0.01±0.00	**0.09**±0.00	**755.60**±16.25
german	UKF	0.01±0.00	0.14±0.02	1032.76±21.83
breast	RBF	0.01±0.00	0.03±0.00	273.20±26.07
breast	UKF	**0.00**±0.00	0.03±0.01	**225.84**±24.54
haberman	RBF	0.01±0.00	0.03±0.00	337.42±48.24
haberman	UKF	**0.00**±0.00	0.03±0.01	**281.56**± 8.23
heart	RBF	0.01±0.00	0.01±0.00	**91.14**±2.37
heart	UKF	**0.00**±0.00	0.01±0.00	135.72±4.47
ionosphere	RBF	0.01±0.00	0.03±0.00	313.30±16.30
ionosphere	UKF	**0.00**±0.00	**0.01**±0.00	**141.08**±18.43
sonar	RBF	0.01±0.00	0.02±0.00	213.58± 9.09
sonar	UKF	**0.00**±0.00	**0.01**±0.00	**155.08**±11.65
tic-tac-toe	RBF	0.01±0.00	**0.15**±0.01	**1649.22**±64.93
tic-tac-toe	UKF	**0.00**±0.00	0.21±0.02	1970.84±66.02
spiral	RBF	3.02±0.12	21.26±2.03	495.50±39.09
spiral	UKF	**2.94**±0.34	**13.21**±0.52	**165.60**± 1.69
peptidases	RBF	**0.42**±0.03	**3.97**±0.20	**4430.64**±20.59
peptidases	UKF	0.64±0.06	7.80±0.83	6040.76±64.32
steganalysis	RBF	**0.02**±0.00	**0.34**±0.02	**465.18**±22.06
steganalysis	UKF	0.05±0.01	0.92±0.04	1148.24±11.62

Table 4.22: Number of training iterations and both classification and training times when using the RBF and UKF kernels. The classifier used was the Multi-Threaded CPU SVM.

Speedup vs number of threads (adult dataset)

	Training		Classification	
Threads	Time (s)	Speedup	Time (s)	Speedup
5	21.79 ± 0.36	2.44	1.03 ± 0.14	3.20
4	**14.83** ± 0.42	**3.58**	**0.84** ± 0.12	**3.93**
3	19.04 ± 0.41	2.79	1.10 ± 0.08	2.99
2	27.94 ± 0.66	1.90	1.66 ± 0.03	1.99
1	53.13 ± 1.06	1.00	3.30 ± 0.03	1.00
LIBSVM	30.49 ± 0.72	1.74	2.02 ± 0.07	1.63

Table 4.23: Speedup achieved by the multi-threaded CPU SVM over the sequential CPU version (one thread) using the "adult" dataset. For comparison, we include the LIBSVM times.

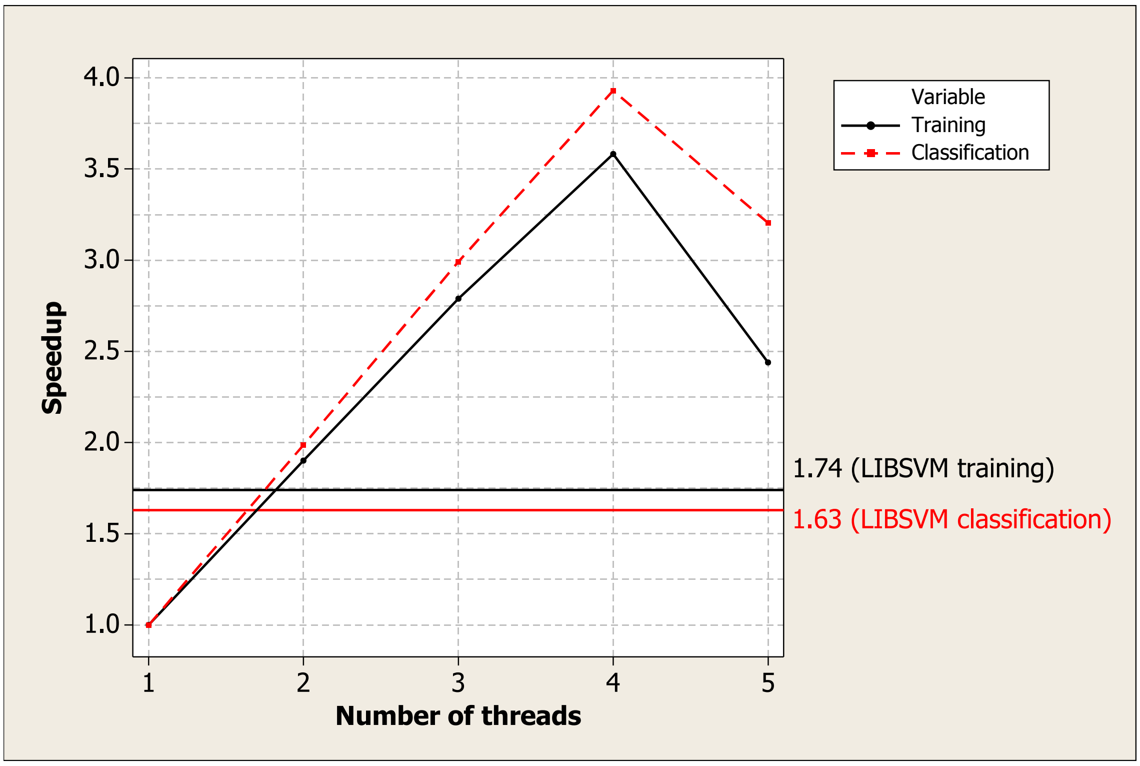


Figure 4.3: Speedup achieved by the CPU version when increasing the number of threads. Using the results shown in table 4.23.

4.5 Discussion

Using the Gaussian (RBF) kernel the performance and amount of support vectors used by our classifiers is within 1% tolerance of the same number used by LIBSVM (table 4.17). In most datasets the GPU implementation has a similar performance to the other classifiers. The only exception is with the dataset "Ionosphere" (table 4.11. The inferior performance in this case can only be explained by the usage of a lower floating precision arithmetic (float data type) which is the only significant difference between our implementations.

The speedups achieved with the multi-threaded CPU SVM (shown in table 4.23) are promising and prove that a GPU SVM was feasible and can provide faster execution times. Using four threads in the quad-core CPU of the test machine gave the fastest times so we conclude that in that case all the threads are given work fully occupied. As can be seen in table 4.23 and figure 4.3 when using two or more threads our simple implementation outperforms LIBSVM using on average 68.5 % less time in both classification and training. Naturally, the classification task scales better with increasing number of threads when compared to the training task. The main reason behind this is that the classification process is essentially parallel. More details regarding the results we obtained with our Multi-Threaded CPU SVM can be seen in Gonçalves et al. [16].

LIBSVM makes use of two important optimizations our implementation does not: second order shrinking heuristic and caching of kernel computations [9]. Caching can be very useful, as we did verify in the "adult" dataset while debugging the calculated kernel dot products. If using caching, the classifier could have reduced the total amount of kernel dot products in about 33 %. If our implementation could cache these repeated calculus, that alone could in theory provide a speedup of $\frac{100}{100-33} \approx 1.5\times$, for the given dataset. The second order heuristic [14] is also reported in the literature to usually reduce the training time although sometimes it can increase it [8]. Another advantage of the LIBSVM is the use of optimized matrix manipulation operations which our implementation does not make use of [9]. Therefore, our CPU implementation is only faster when using two or more threads. It is comprehensible as LIBSVM has many years of research and optimizations behind the training algorithm [9].

The GPU implementation used, in average, 28% more iterations in the SMO algorithm than those of LIBSVM. This is caused by the usage of a second order heuristic by LIBSVM and therefore chooses optimal lagrange multipliers to fasten the algorithm's convergence. However, the raw computation power of a modern GPU allowed our implementation to achieve both faster times in training (up to 13.61×) and classification (up to 265.85×). However, it can be seen that our implementations have an impressive speedup for the "Spiral" dataset. Although the classification speedup is easily explained by the embarrassingly parallel nature of the classification task, the same does not apply to the training process, where it can be seen that LIBSVM has problems with the training task. Therefore our GPU implementation had a training improvement of 165.15× and the CPU version a speedup of 6.9×, being this last value greater than the amount of used threads (4). Naturally, as we achieved a faster than linear speedup with our CPU version against LIBSVM, the result is related to algorithmic differences.

Naturally, the speedups were greater with more complex datasets. For the Multi-Threaded CPU SVM there is always an overhead of creating (forking) and merging (synchronizing) threads. In the GPU SVM, this overhead comes from the initialization of the GPU context, copying data between the GPU's memory and the host's RAM and controlling the CUDA's kernels, which is done by the local CPU.

In table 4.21 we show that the UKF kernel yields similar performance to the one obtained the RBF kernel. Each kernel can give a smaller improvement when compared to the other, depending on the dataset. Using the Wilcoxon signed ranked test [51] with a significance level of 5% we found no statistical evidence of the UKF kernel performing worse than the RBF kernel and vice-versa. It is interesting to note that, with the exception of the Sonar dataset, UKF yields better F-Score results in the datasets that present a smaller number of SVs than the corresponding number for the RBF kernel. This seems to indicate, as the "spiral" dataset is an example of, that UKF presents better classification performance when it is able to gather points near to each other, in a higher dimension space, as intended.

Regarding the classification and training times between both RBF and UKF kernels, shown in table 4.22, we can not conclude that one kernel is faster than the other. When compared with the RBF kernel, the only issue we found with the UKF kernel is the increased grid search complexity caused by the greater number of parameters. Therefore, the choice of the ideal kernel depends on the problem being considered.

4.6 Conclusions

In this chapter we presented a study consisting of the feasibility of both our CPU and GPU SVM classifiers. We analysed various datasets, the performance of all classifiers and LIBSVM and the speedup achieved by our versions.

After analysing the results obtained with our CPU and GPU classifiers it can be concluded that both implementations are correct. The faster classification and training times available using the GPU SVM make it an interesting choice for complicated datasets or when choosing the best SVM and kernel parameters (grid searching). Thus, these tasks can be now executed in less time using the GPU of a modern computer and laptop.

In case there is no GPU available, our CPU implementation, although simple, can cope with the described tasks by making fully use of modern multi-core CPUs.

We also shown that in addition to the usual kernels, the new UKF kernel has good generalization properties in the high-dimensional feature space and gives an added value for our SVM classifiers.

Chapter 5

Signature recognition using the GPU

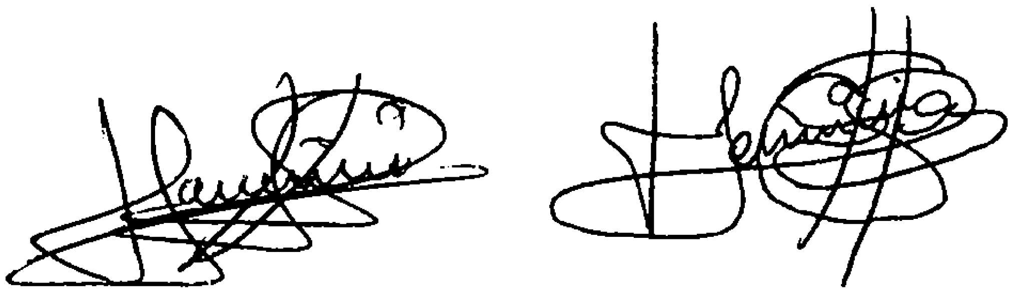
In this chapter we study the problem of off-line signature recognition. The chapter is organized as follows: a brief introduction is done in section 5.1 and we describe the features in section 5.2. The experimental setup and dataset characteristics are shown in section 5.3 and we present the results obtained in section 5.4. In section 5.5 we discuss the results we obtained and in section 5.5 we conclude our study regarding the off-line signature recognition problem.

5.1 Introduction

The problem of handwritten signature recognition is a challenging one that plays an important role in many official documents. The idea consists of creating an off-line classifier to discriminate between fake signatures (forgery) and genuine ones in a database of digitalized signatures, after identifying the author. The database was obtained from the GPDS (Grupo de Procesado Digital de Señales), available at `http://www.gpds.ulpgc.es/download/`. Figure 5.1 shows two signatures from the referred database.

5.2 Dataset

The database contains data from 300 individuals. For each individual there are 24 genuine signatures, plus 30 forgeries of his/her signature making 54 images per individual and a total of 16200 images. The 24 genuine specimens of each signer were collected in single day writing sessions. The forgeries were produced under the following conditions: The forger imitates a genuine signature from the static image of the genuine signature (scanned at 300 DPI) and the forger is allowed to practice writing the signature for as long as s/he wishes. Each forger



(a) Genuine Signature. (b) Forged Signature.

Figure 5.1: Two signatures from the GPDS database [15].

has to imitate three signatures of five signers in a single day writing session. The genuine signature shown to each forger is chosen randomly from the 24 genuine ones. Therefore, for each genuine signature, there are 30 simple forgeries made by 10 forgers from 10 different genuine specimens. The dataset used consists of 16200 handwritten off-line signature recognition (each signature is a 649×462 pixels image). Additional information on this database can be found in Ferrer et al. [15]. Although the image dataset could be used directly its size in memory would make it impractical to use as it would be composed of $16200 \times 649 \times 642 = 6.749.859.600$ pixels (roughly 6.75 Gigapixels). As each pixel would be translated to a single precision floating type (four bytes per pixel) its storage would require 25.15 GB of RAM. One solution could be the use of a Principal Component Analysis (PCA) or LDA feature reduction technique however the use of raw data does not give acceptable results. Therefore, feature extraction constitute an essential step of a signature verification system in order to achieve a good performance [43].

Previously, research was by done by Ribeiro et al [43], Armand et al [2], Blumenstein et al. [6] and Ferrer et al [15] in order to study better features from the original dataset. More information regarding the extraction algorithms are referenced in their paper. The authors in [43] describe fourteen algorithms, but only twelve were used as the other two were not useful for the learning process. These were the width and height of each image. All images were centred and added borders to make them the same size, thus these two attributes would not help with the learning process. The other twelve attributes are summarized in the next sub-sections.

Gravity Center Angle

This feature consists in dividing an image using a centred vertical axis, cropping two equally sized sections of the image. The "Centroid" of each section is calcu-

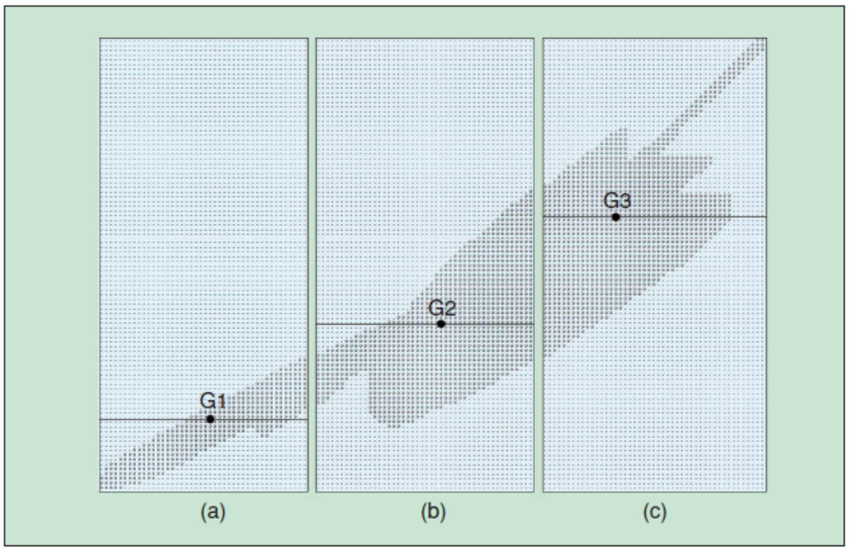


Figure 5.2: Six-fold surface feature extraction.

lated and the angle of the vector between the two centroids is used as the final feature.

Maximum Intensity Points

This feature returns either the line or the column of the image containing the greatest number of black pixels.

Tri-fold surface

This feature corresponds on the proportion of pixels contained in three vertical divisions of the image. Therefore it represents the dispersion of pixels within the three sections.

Six-fold surface

Following the last feature this extends the algorithm by additionally dividing the image in three horizontal sections. However the horizontal divisions done on each vertical section are done using the gravity center of the vertical sections, as can be seen in image 5.2.

Best Fit

This feature represents the angle between the signature and the horizontal axis. Therefore, two lines are interpolated using the two centroids, one on the left side

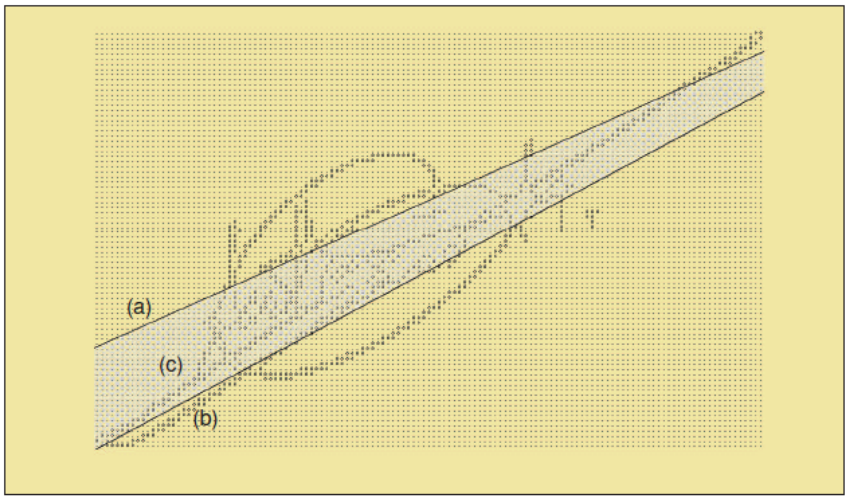


Figure 5.3: Best Fit feature extraction.

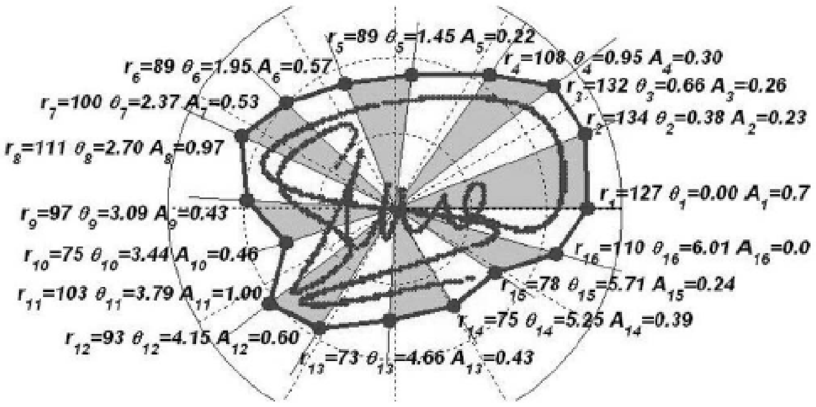

r_5=89 θ_5=1.45 A_5=0.22
r_6=89 θ_6=1.95 A_6=0.57
r_4=108 θ_4=0.95 A_4=0.30
r_3=132 θ_3=0.66 A_3=0.26
r_7=100 θ_7=2.37 A_7=0.53
r_2=134 θ_2=0.38 A_2=0.23
r_8=111 θ_8=2.70 A_8=0.97
r_1=127 θ_1=0.00 A_1=0.7
r_9=97 θ_9=3.09 A_9=0.43
r_{16}=110 θ_{16}=6.01 A_{16}=0.0
r_{10}=75 θ_{10}=3.44 A_{10}=0.46
r_{15}=78 θ_{15}=5.71 A_{15}=0.24
r_{11}=103 θ_{11}=3.79 A_{11}=1.00
r_{14}=75 θ_{14}=5.25 A_{14}=0.39
r_{12}=93 θ_{12}=4.15 A_{12}=0.60
r_{13}=73 θ_{13}=4.66 A_{13}=0.43

Figure 5.4: Geometric Parameters (Polar) feature extraction.

of the image and the other on the right side (Figure 5.3). Additionally, the authors added the proportion of pixels inside each centroid to represent the dimension of the centroids.

Geometric Parameters (Polar)

This feature characterizes the distribution of the image radially, starting on its gravity center. Therefore, the image is equally divided into equally sized angular sections using equidistant points on the outer edge of the image (figure 5.4). Extracted features are the distance of each point to the center, its angle with the center and the proportion of black pixels contained in each section.

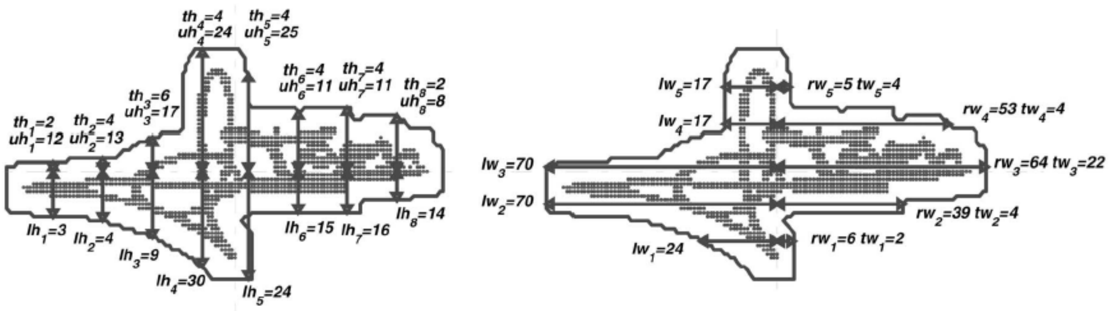


Figure 5.5: Geometric Parameters (Cartesian) feature extraction.

Geometric Parameters (Cartesian)

This feature identifies the image's morphology by doing an analysis using two cartesian axis (figure 5.5). The same principle behind the extraction of the polar features is applied, however using sections evenly distributed in a rectangle centred in the image.

Modified Direction Feature (MDF)

This feature identifies the direction of the different segments composing the signature's line and the location of the areas were pixels change from white to black. This last method is done either vertically either horizontally.

K-Means

Another feature which identifies the position of the main elements of the signature is K-Means clustering using the image's black pixels. The number of clusters is fixed for all the images and set to five [43]. An example can be seen in figure 5.6.

Histogram Frequencies

In order to characterize the signature's intensity variations either vertically and horizontally this feature calculates the vertical and horizontal frequency histograms of the pixels in each image. These frequencies are calculated using the Fast Fourier Transform.

Discrete Cosine Transform (DCT) Frequencies

This feature uses the two dimensional Discrete Cosine Transform (DCT) to change the initial amplitude-time space to a new amplitude-frequency space, therefore

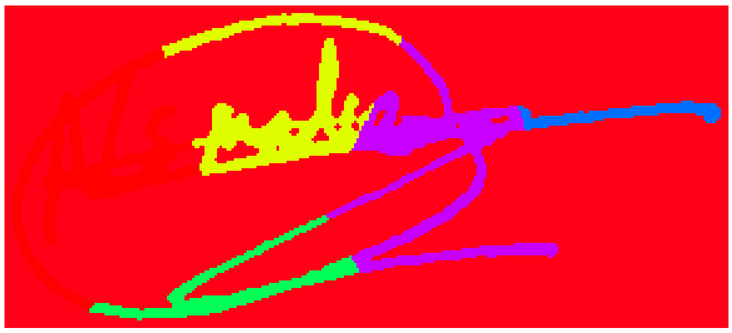


Figure 5.6: K-Means feature of a signature.

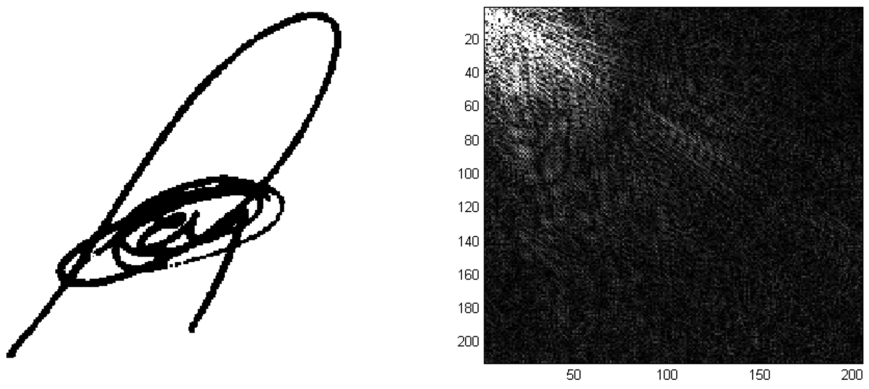


Figure 5.7: Discrete Cosine Transform of a signature.

representing the intensity in frequencies of the initial image (figure 5.7). It is the same algorithm behind JPEG and some video compression codecs.

Wavelet Transform feature

This feature is similar to the DCT but it extracts temporal resolution by capturing both frequency and time information. The base (mother) wavelet used to extract the features is the Haar (rectangular) wavelet. Figure 5.8 shows an example of a two level wavelet decomposition.

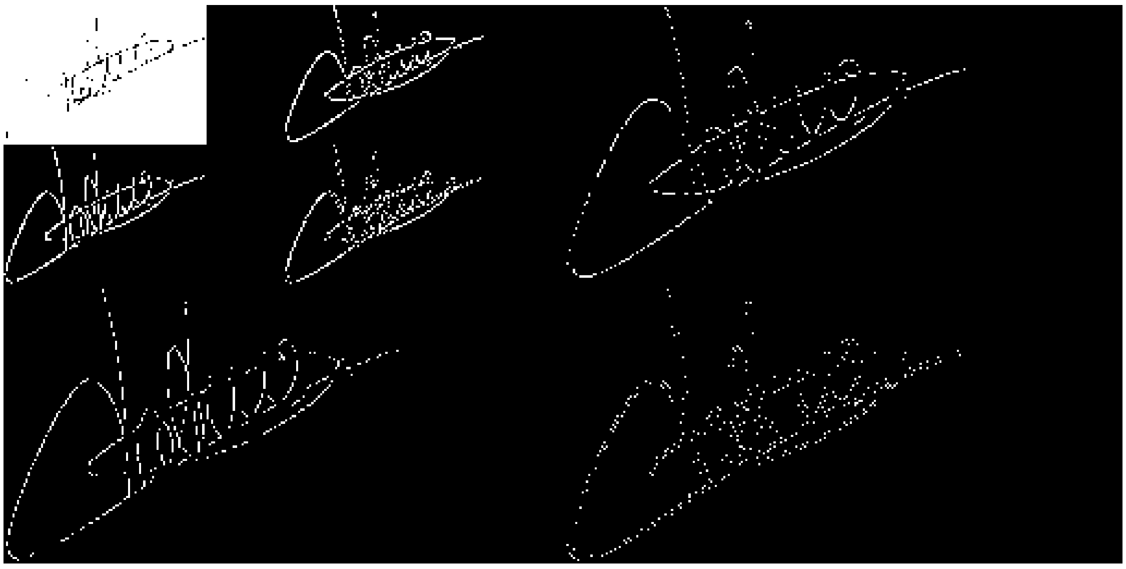


Figure 5.8: Wavelet Transform of a signature.

5.3 Experimental setup

For extracting the results we used 9 images for the testing set and the remaining 45 for the training set. Both training and testing sets were randomly generated from the initial data, being the test set composed of 4 genuine signatures and 5 forged. The experiments were run 10 times per configuration.

Feature	Number of attributes
Best Fit	4
Discrete Cosine Transform (DCT)	5
Geometric Parameters (Cartesian)	180
Geometric Parameters (Polar)	192
Gravity Center	1
Histogram Frequencies (hist)	6
K-Means	10
Max Intensity Points (maxint)	1
Modified Direction Feature (MDF)	160
Six-fold-Surface	6
Three-fold-Surface	3
Wavelet Transform Feature	12

Table 5.1: Number of attributes for each feature of the signature dataset.

In table 5.1 we present the number of attributes for each extracted feature from the image dataset. For that purpose we used a tool developed by Ivo Gonçalves and Sérgio Santos, also used in [43].

The system's configuration is the same as the last chapter, composed of an Intel

Core i5 running at 3.33 GHz with 12 GB of RAM and an NVIDIA Geforce GTX 570 with its characteristics described in table 4.5. Naturally, all of the datasets were normalized using standard score normalization.

Being the GPU SVM currently binary, the multi-class functionality is not supported. Therefore the identification of an individual by using its signature is done only by comparing against another individual. We also developed an external program (driver) which controls the GPU SVM allowing for fine tune of the grid search algorithm, K-Fold cross-validation or a custom validation scheme and supporting multiple parallel executions of either the Multi-Threaded CPU and GPU SVMs. This is useful as in most experiments the training process is faster than the dataset loading time. Therefore it is possible to run multiple training instances at the same time in order to minimize the total execution time.

We did three kinds of experiments. The first was simply the identification of original and forged signatures. For that purpose, we used all the 300 individuals and studied each group of features. We run two sub-experiments where in the first we only used the RBF kernel with the configuration given in table 5.2. The second sub-experiment we used the feature combination DCT + MDF and tested all the supported kernel functions by our GPU implementation, with the configuration given in table 5.3. We only tested the given feature combination because the time taken by the grid-search algorithm for all the kernels was expensive. For this first experiment we used 5 fold K-Fold cross-validation.

For the second experiment, instead of using all the 300 groups of signatures as the first we studied groups or combinations of features for each individual. Therefore, this experiment consisted of identifying, for each person, if a signature either original or forged. We only used the RBF kernel and as a validation method we used the custom 4 original plus 5 forged signatures composing the test set, as specified above.

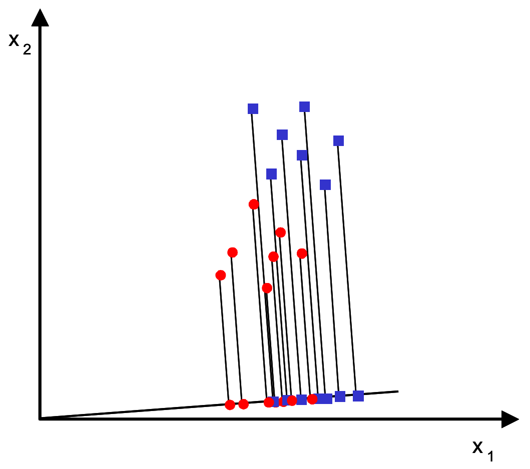
The third experiment consisted on identifying a signature according to the related individual, using both the original and forged signatures. As we described before, since our classifier is currently binary we simulated a One-Against-One multi-class classifier, that is, we train and test each individual class against one of the others. Thus, for a dataset with c classes, $(c \times (c - 1))$ training tasks are required. In this third experiment we performed $300 \times 299 = 89700$ trainings, excluding the validation procedure for each sub-experiment. As the amount of time involved in the training process is high, we only used the RBF kernel and a 5 K-Fold cross-validation.

Features	C	γ
polar	0.08	0.04
cartesian	0.64	0.08
mdf	10.24	0.04
wavelet	0.01	64.00
bestfit	0.01	8.19
dct	0.08	2.56
gravitycenter	0.01	0.00
hist	0.01	8.19
kmeans	0.04	0.00
maxint	0.01	0.01
sixfold	0.02	3.78
tri-fold-surface	1.28	0.02
dct+mdf	11.71	0.02
dct+mdf+cart	8.00	0.01

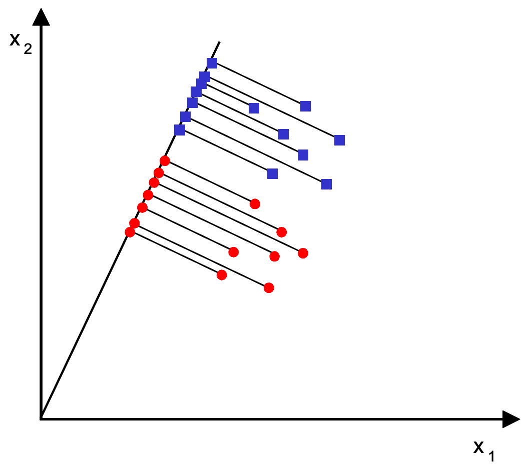
Table 5.2: RBF kernel configuration used in the first experiment, the generic identification of original and forged signatures.

Kernel	C	γ	L	σ	α	b	q
RBF	11.71	0.02					
UKF	10.24		1.00	2.56	0.25		
Linear	0.001						
Polynomial	0.10				0.99	13.40	0.04
Sigmoid	0.10				0.08	10.24	

Table 5.3: The configuration of all kernel functions used in the first experiment, when using the combination of features DCT + MDF.



(a) Initial feature space.



(b) Projected feature space.

Figure 5.9: Two-dimensional LDA reducing an initial feature space (left) to a new feature space (right) where the separation of the classes is improved. Images taken from [18].

5.4 Results

In this section we present the results for the three experiments explained above. Tables 5.4, 5.5, 5.6 and 5.7 contain the results for the first experiment where the single objective was to identify if a given signature is original or forged. Tables 5.4 and 5.5 show the results we obtained using only the RBF kernel while tables 5.6 and 5.7 show the performance using all the available kernels in our SVM implementation. For this last sub-experiment we only used both features DCT and MDF.

LDA is a supervised feature reduction algorithm which aims to reduce input space dimensionality while maximizing the classe's separation [18]. This is done by projecting the initial feature space using a linear combination of the input features. Figure 5.9 presents a hypothetical two-dimensional LDA feature projection from an original feature space (left) to a new feature space (right) where the class separation is higher. In figure 5.10 we show a LDA projection of this experiment using features DCT and MDF, in order to visualize the level of the task's complexity at hand. Regarding the second experiment, the same performance metrics described above and for a given set of features can be seen in tables 5.8 5.9. Figures 5.13 and 5.14 show the grid-search contour map for the individual number 23, using either DCT or MDF features. In figures 5.11 5.12 we present a graphical statistical summary for both FPR and F-Score obtained using the features DCT + Cartesian + MDF + Polar.

The third experiment's results are shown in tables 5.10, 5.11 and figure 5.15.

The two tables show the same performance metrics as shown in the other experiments while figure 5.15 has a graphical summary for the F-Score obtained using the MDF + DCT features.

Features	Accuracy	Precision	Recall	FPR
mdf	77.46 ±0.72	78.35 ±1.23	68.14 ±1.67	15.08 ±1.08
dct	67.18 ±0.92	59.20 ±0.98	84.24 ±1.54	46.48 ±1.91
cart	66.71 ±0.65	58.82 ±0.99	83.66 ±1.22	46.85 ±1.19
polar	57.77 ±1.21	51.65 ±1.42	79.94 ±2.50	59.93 ±3.70
six-fold-surface	52.80 ±1.16	48.26 ±0.68	85.02 ±2.35	72.98 ±3.75
tri-fold-surface	44.50 ±0.07	44.47 ±0.03	99.92 ±0.11	99.83 ±0.16
gravity	44.44 ±0.68	44.44 ±0.68	100.00 ±0.00	100.00 ±0.00
kmeans	44.45 ±0.89	44.45 ±0.89	100.00 ±0.00	100.00 ±0.00
maxint	44.54 ±0.81	44.47 ±0.68	99.50 ±2.63	99.44 ±2.97
wavelet	45.08 ±0.87	44.58 ±0.85	97.08 ±0.52	96.52 ±0.63
bestfit	53.95 ±0.86	48.84 ±0.88	75.14 ±1.97	63.01 ±3.02
hist	55.97 ±0.89	50.40 ±1.16	59.78 ±1.60	47.08 ±1.69
dct+mdf	**79.42** ±1.13	72.31 ±1.85	87.23 ±2.12	26.83 ±2.89
dct+mdf+cart	80.53 ±0.73	**78.99** ±2.86	76.99 ±4.30	**16.63** ±3.90

Table 5.4: Accuracy, Precision, Recall and False Positive Rate for the first identification experiment, using the RBF kernel.

Features	FDR	F-Score	Specificity
mdf	21.65 ±1.23	72.87 ±1.04	**84.92** ±1.08
dct	40.80 ±0.98	69.52 ±0.87	53.52 ±1.91
cart	41.18 ±0.99	69.07 ±0.77	53.15 ±1.19
polar	48.35 ±1.42	62.71 ±0.81	40.07 ±3.70
six-fold-surface	51.74 ±0.68	61.55 ±0.42	27.02 ±3.75
tri-fold-surface	55.53 ±0.03	61.54 ±0.04	0.17 ±0.16
gravity	55.56 ±0.68	61.54 ±0.65	0.00 ±0.00
kmeans	55.55 ±0.89	61.53 ±0.86	0.00 ±0.00
maxint	55.53 ±0.68	61.45 ±0.83	0.56 ±2.97
wavelet	55.42 ±0.85	61.10 ±0.82	3.48 ±0.63
bestfit	51.16 ±0.88	59.18 ±0.62	36.99 ±3.02
hist	49.60 ±1.16	54.68 ±1.04	52.92 ±1.69
dct + mdf	27.69 ±1.85	**79.03** ±0.92	73.17 ±2.89
dct + mdf + cart	**21.01** ±2.86	77.82 ±1.07	83.37 ±3.90

Table 5.5: False Discovery Rate, F-Score and Specificity for the first identification experiment using the RBF kernel.

Kernel	Accuracy	Precision	Recall	FPR
Linear	60.33±7.21	51.60±16.07	62.55±28.85	41.45±25.65
Polynomial	54.33±9.22	50.55±6.14	84.81±12.71	70.06±25.19
RBF	79.42±1.13	72.31± 1.85	87.23± 2.12	26.83± 2.89
Sigmoid	44.41±0.14	44.40± 0.05	**99.49**± 0.93	99.65± 0.91
UKF	**79.97**±1.63	**73.75**± 3.76	86.25± 4.87	**25.05**± 6.04

Table 5.6: Accuracy, Precision, Recall and False Positive Rate for the first experiment using the all kernels and the combination features MDF + DCT.

Kernel	FDR	F-Score	Specificity	nSVs
Linear	40.40±12.98	53.34±22.30	58.55±25.65	6696.70±4721.16
Polynomial	49.45± 6.14	62.45± 3.42	29.94±25.19	4605.20±4491.62
RBF	27.69± 1.85	79.03± 0.92	73.17± 2.89	8159.68± 563.64
Sigmoid	55.60± 0.05	61.40± 0.20	0.35± 0.91	1424.80± 298.35
UKF	**26.25**± 3.76	**79.29**± 1.16	**74.95**± 6.04	10731.16± 377.94

Table 5.7: False Discovery Rate, F-Score and Specificity for the first experiment using the all kernels and the combination features MDF + DCT.

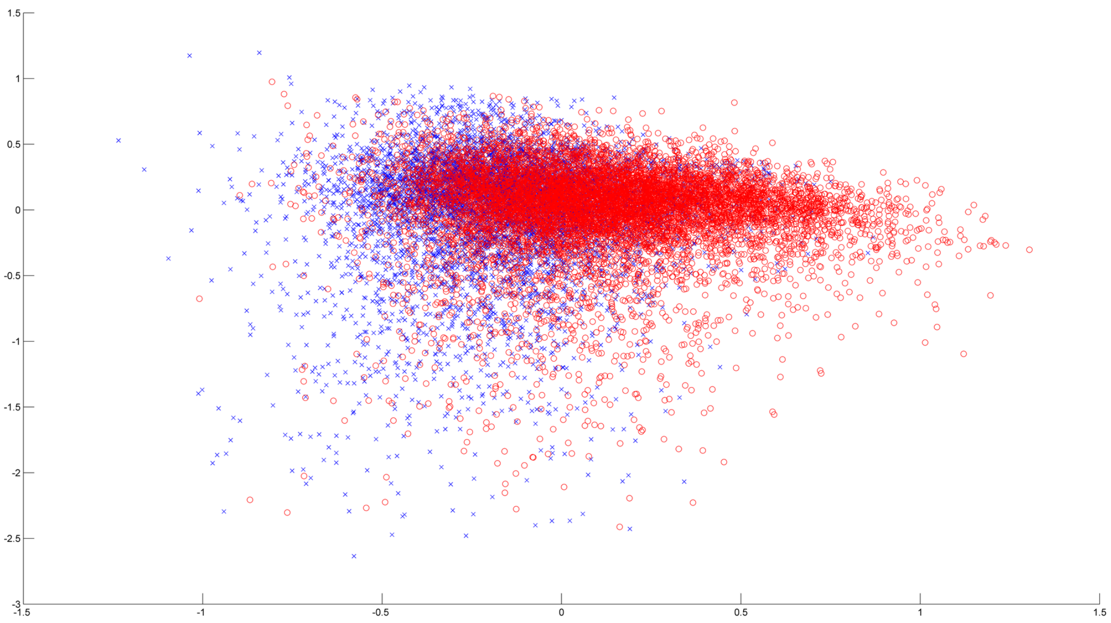


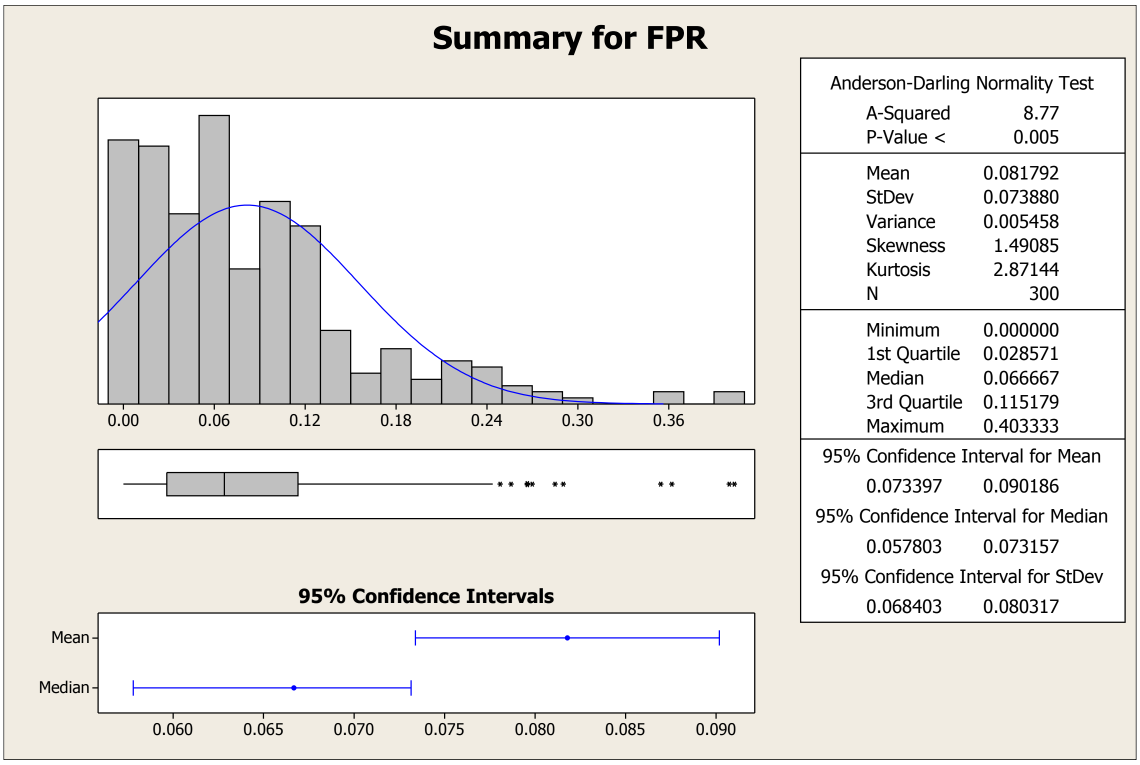
Figure 5.10: Two dimensional projection of LDA for the first experiment using features MDF and DCT.

Features	Accuracy	F-Score	FDR	FPR
cartesian	77.23±11.39	67.54±16.92	15.98±18.38	10.64±12.24
mdf	76.74±11.56	66.77±18.05	15.82±17.96	10.62±12.20
polar	71.13±12.84	59.45±18.78	21.44±21.88	13.84±14.39
wavelet	61.63±13.09	42.06±19.90	28.96±27.52	16.36±16.66
kmeans	54.00±13.19	31.32±19.58	38.84±30.81	22.71±20.11
six-fold-surface	67.19±12.58	50.33±18.91	22.83±23.70	13.10±14.32
dct	79.16±10.55	69.06±16.00	13.06±15.54	**7.97**±9.50
histogram	64.33±12.68	45.16±19.64	26.92±25.40	14.44±14.57
bestfit	66.18±12.43	48.54±19.08	25.30±24.72	14.08±14.60
tri-fold-surface	65.73±11.96	57.87±16.45	35.76±21.37	29.02±18.19
gravity center	59.34±12.17	52.23±16.19	42.92±21.96	35.92±19.97
max intensity angle	55.93±12.30	46.86±16.99	46.93±22.38	37.66±20.30
mdf + dct	78.76±11.01	69.74±16.67	12.87±15.59	8.60±10.47
polar + cartesian	73.71±12.65	61.02±18.95	15.55±18.57	10.36±12.59
mdf + cartesian	78.33±11.07	69.28±16.58	13.45±16.30	9.29±11.08
dct + cartesian	78.70±11.55	69.81±17.46	14.00±16.44	9.45±10.98
dct + polar + mdf	75.75±12.02	64.87±18.35	15.02±17.58	10.24±12.12
dct + cart + mdf	79.45±11.50	71.16±16.71	12.96±15.74	8.78±10.80
dct + cart + mdf + polar	**79.92**±11.07	**71.62**±16.59	**12.33**±14.93	8.18±9.78

Table 5.8: Forged/original signature identification per individual (second experiment). Shown performance metrics are Accuracy, F-Score, False Discovery Rate and False Positive Rate.

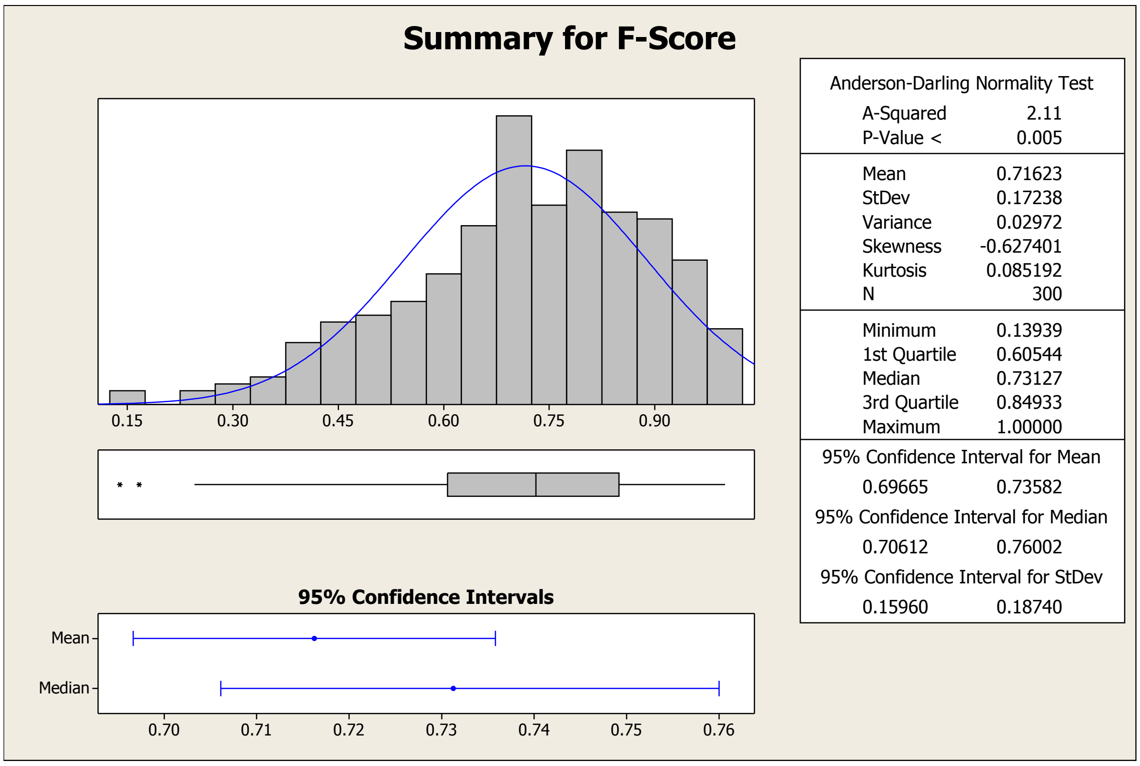
Features	Precision	Recall	Specificity
cartesian	79.68±20.62	65.62±21.40	89.36±12.24
mdf	79.64±21.19	63.86±22.22	89.38±12.20
polar	73.30±24.25	57.46±23.63	86.16±14.39
wavelet	60.38±29.20	39.21±23.48	83.64±16.66
kmeans	45.02±30.18	31.71±24.98	77.29±20.11
six-fold-surface	68.17±27.00	47.07±22.34	86.90±14.32
dct	82.74±18.40	65.85±19.23	**92.03**± 9.50
histogram	61.88±28.84	42.36±23.18	85.56±14.57
bestfit	65.70±27.64	45.10±22.08	85.92±14.60
trisurface	61.90±21.83	61.27±20.79	70.98±18.19
gravity center	54.88±21.49	57.15±21.76	64.08±19.97
max intensity angle	51.00±22.40	50.66±22.46	62.34±20.30
mdf + dct	83.00±18.94	66.32±20.88	91.40±10.47
polar + cartesian	74.52±22.46	58.87±24.31	89.64±12.59
mdf + cartesian	82.35±18.68	66.80±21.73	90.71±11.08
dct + cartesian	81.80±19.10	67.27±21.89	90.55±10.98
dct + polar mdf	78.58±22.20	62.35±23.43	89.76±12.12
dct + cart + mdf	83.91±18.25	**68.44**±21.50	91.22±10.80
dct + cart + mdf + polar	**84.47**±17.32	68.35±21.42	91.82± 9.78

Table 5.9: Forged/original signature identification per individual (second experiment). Shown performance metrics are Precision, Recall and Specificity.



Anderson-Darling Normality Test	
A-Squared	8.77
P-Value <	0.005
Mean	0.081792
StDev	0.073880
Variance	0.005458
Skewness	1.49085
Kurtosis	2.87144
N	300
Minimum	0.000000
1st Quartile	0.028571
Median	0.066667
3rd Quartile	0.115179
Maximum	0.403333
95% Confidence Interval for Mean	
0.073397	0.090186
95% Confidence Interval for Median	
0.057803	0.073157
95% Confidence Interval for StDev	
0.068403	0.080317

Figure 5.11: False Positive Rate statistical summary for the forged/original signature identification per individual (second experiment). The features used were DCT + MDF + Cartesian and Polar coordinates.



Anderson-Darling Normality Test	
A-Squared	2.11
P-Value <	0.005
Mean	0.71623
StDev	0.17238
Variance	0.02972
Skewness	-0.627401
Kurtosis	0.085192
N	300
Minimum	0.13939
1st Quartile	0.60544
Median	0.73127
3rd Quartile	0.84933
Maximum	1.00000
95% Confidence Interval for Mean	
0.69665	0.73582
95% Confidence Interval for Median	
0.70612	0.76002
95% Confidence Interval for StDev	
0.15960	0.18740

Figure 5.12: F-Score statistical summary for the forged/original signature identification per individual (second experiment). The features used were DCT + MDF + Cartesian and Polar coordinates.

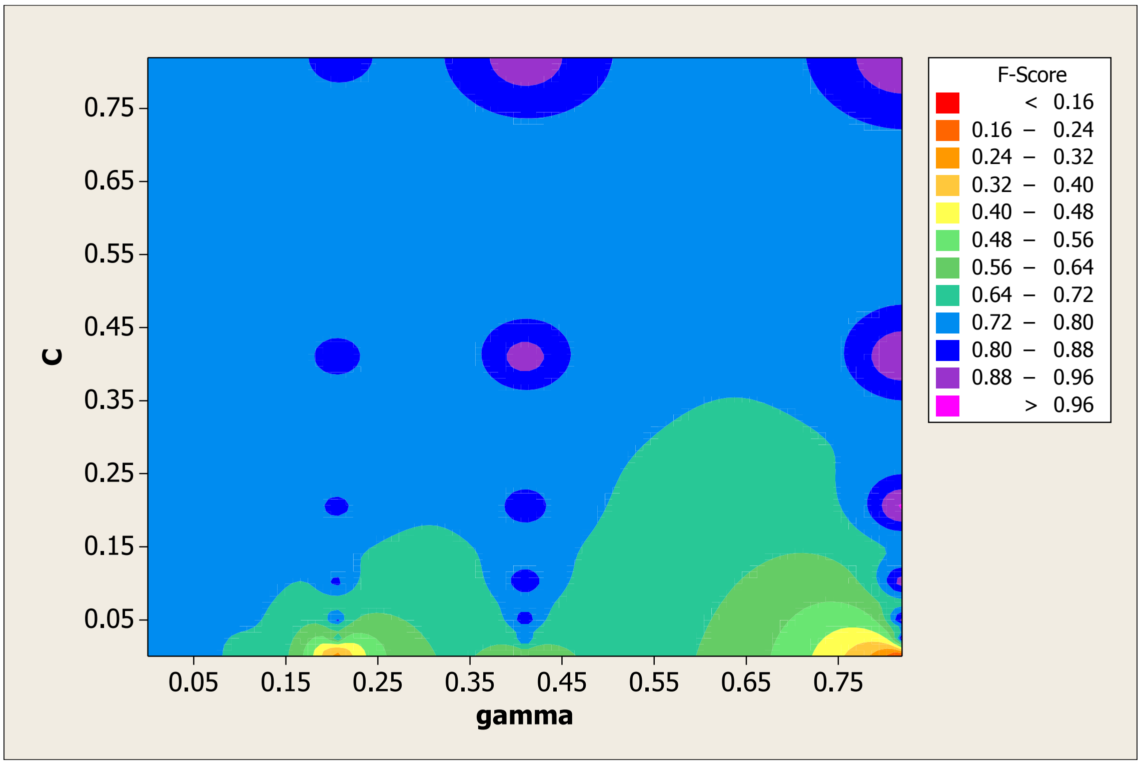


Figure 5.13: F-Score RBF grid search using the DCT features for the detection of forged/original signature identification, author number 23 (second experiment).

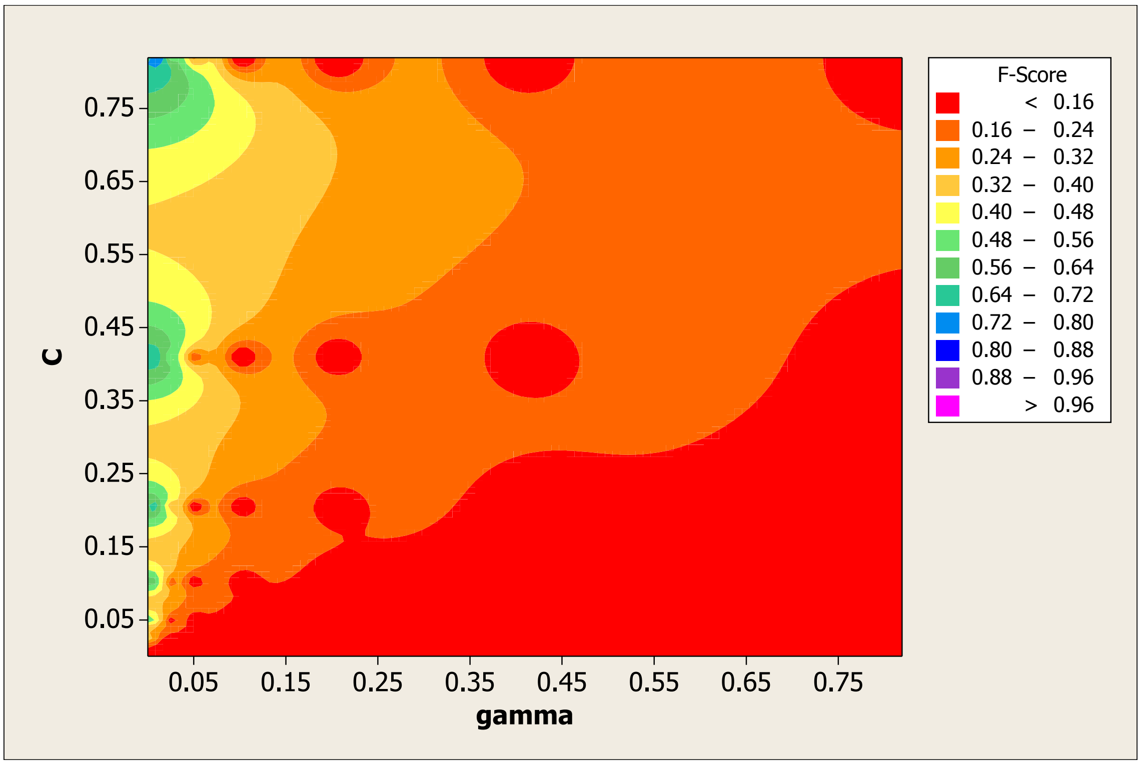


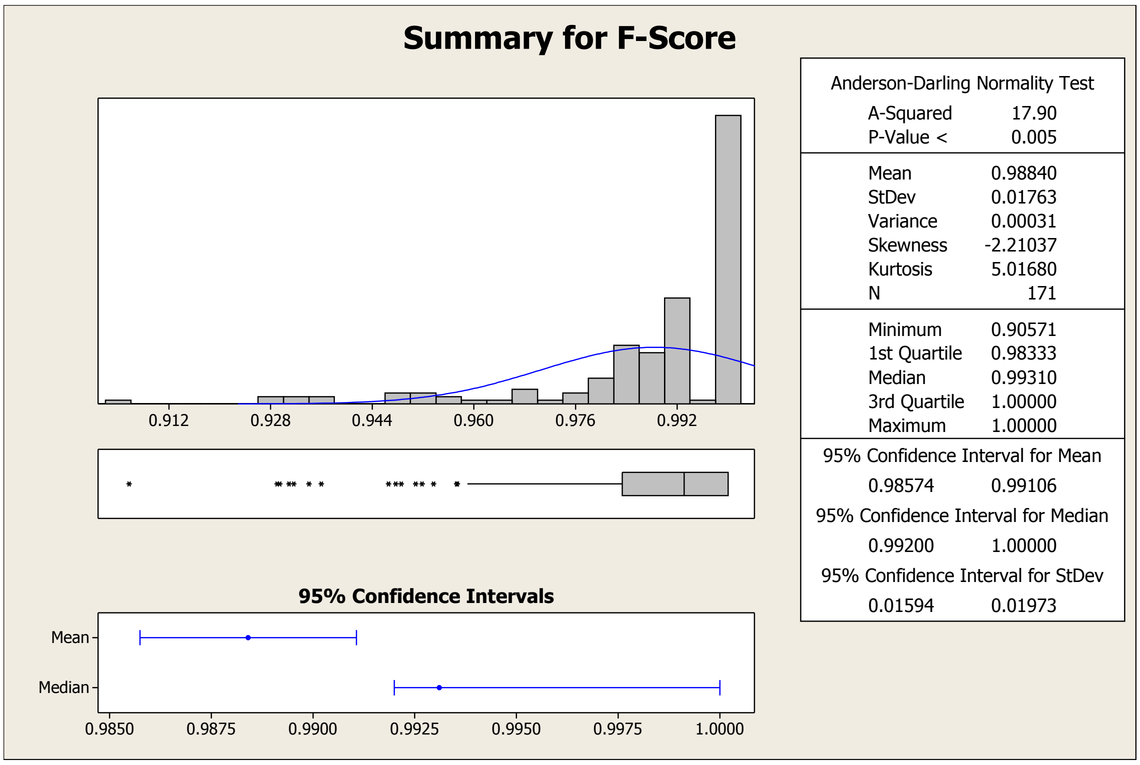
Figure 5.14: F-Score RBF grid search using the MDF features for the detection of forged/original signature identification, author number 23 (second experiment).

Features	Accuracy	F-Score	FDR	FPR
cartesian	97.62±phantom02.12	97.58±phantom02.16	2.43± 2.91	2.62± 3.05
mdf	97.94± 1.90	97.80± 2.17	2.11± 2.67	**2.28**± 2.77
polar	91.97± 5.25	91.56± 5.99	7.73± 7.79	8.71± 8.62
wavelet	82.87± 6.26	82.06± 8.53	14.05±10.40	17.79±13.72
kmeans	69.50± 8.92	68.47±12.51	25.72±15.48	31.87±21.25
six-fold-surface	89.60± 4.63	89.31± 5.53	8.69± 7.31	10.62± 8.91
dct	81.54± 6.62	80.65± 7.64	14.55±10.41	15.69±12.38
histogram	74.55± 7.59	73.12±10.23	20.82±13.69	24.25±17.48
bestfit	90.42± 4.22	89.88± 4.82	8.09± 6.33	8.83± 7.18
tri-fold-surface	85.51± 5.20	85.08± 5.97	14.30± 7.69	14.47± 8.05
gravity center	74.92± 7.07	73.92± 8.38	23.43±10.55	23.29±10.87
max intensity angle	66.99± 7.99	66.16± 9.50	32.28±11.93	33.12±12.44
mdf + dct	**98.13**± 1.75	**98.14**± 1.83	**2.05**± 2.48	2.36± 2.70
polar + cartesian	90.46± 6.27	90.08± 7.29	9.81± 8.46	11.81± 9.57
mdf + cartesian	96.53± 2.65	96.28± 3.05	3.76± 4.03	4.35± 4.52
dct + cartesian	97.50± 2.29	97.30± 2.60	2.56± 3.15	2.81± 3.43
dct + polar + mdf	91.72± 5.97	91.68± 6.43	8.81± 8.03	10.78± 9.67

Table 5.10: Results related to the One-Against-One (binary) signature author identification (third experiment) using the given features. Shown performance metrics are Accuracy, F-Score, False Discovery Rate and False Positive Rate.

Features	Precision	Recall	Specificity
cartesian	97.57± 2.91	98.08± 2.35	97.38± 3.05
mdf	97.76± 2.74	98.41± 2.11	**97.72**± 2.77
polar	91.67± 8.03	93.69± 7.01	91.29± 8.62
wavelet	84.62±10.76	86.64±13.44	82.21±13.72
kmeans	71.55±15.86	76.28±20.59	68.13±21.25
six-fold-surface	91.05± 7.50	91.62± 8.31	89.38± 8.91
dct	85.45±10.41	81.86±12.75	84.31±12.38
histogram	78.32±13.92	77.55±17.09	75.75±17.48
bestfit	91.91± 6.33	90.79± 7.45	91.17± 7.18
tri-fold-surface	85.70± 7.69	85.70± 7.93	85.53± 8.05
gravity center	76.57±10.55	73.74±11.29	76.71±10.87
max intensity angle	67.72±11.93	67.89±13.06	66.88±12.44
mdf + dct	**97.88**± 2.49	**98.83**± 1.64	97.64± 2.70
polar + cartesian	88.99± 9.08	94.06± 6.68	88.19± 9.57
mdf + cartesian	95.84± 4.33	97.70± 2.54	95.65± 4.52
dct + cartesian	97.04± 3.45	98.02± 2.54	97.19± 3.43
dct + polar + mdf	90.32± 8.41	95.44± 5.54	89.22± 9.67

Table 5.11: Results related to the One-Against-One (binary) signature author identification (third experiment) using the given features. Shown performance metrics are Precision, Recall and Specificity.



Anderson-Darling Normality Test	
A-Squared	17.90
P-Value <	0.005
Mean	0.98840
StDev	0.01763
Variance	0.00031
Skewness	-2.21037
Kurtosis	5.01680
N	171
Minimum	0.90571
1st Quartile	0.98333
Median	0.99310
3rd Quartile	1.00000
Maximum	1.00000
95% Confidence Interval for Mean	
0.98574	0.99106
95% Confidence Interval for Median	
0.99200	1.00000
95% Confidence Interval for StDev	
0.01594	0.01973

Figure 5.15: F-Score histogram for the One-Against-One (binary) signature author identification (third experiment) using both the MDF and DCT features.

5.5 Discussion

As can be seen in tables 5.4 and 5.5 the global original/forged signature identification task is somewhat complex. Most features are confusing and have a high ($\geq 70\%$) False Positive Rate. This can be intuitively understood as identifying forged signatures, regardless of the individual they represent is problematic. Thus the performance of the resulting classifier is disappointing (as shown in the results).

To help understand the first experiment's complexity, the help of the LDA tool allowed us to confirm the complexity of the task, shown in figure 5.10. However, the feature MDF alone gave promising results and when combined with either the DCT or the Cartesian coordinates resulted in higher F-Scores (approximately 80%) and lower False Positive Rate (FPR). Using the features MDF + DCT we improved the results by a minimal percentage, by making use of the UKF kernel. However, this kernel used more SVs than the RBF kernel, reflecting the complexity of the problem at hand. The remaining kernels gave worst results (F-Score $< 63\%$).

Similar conclusions taken for the first experiment can be applied to the second, the identification of original vs forged signatures for each person. Again, we found confusing features as the K-Means, Histogram, Best-Fit, among others. Although these gave smaller False Positives (FPR), the remaining performance metrics, mostly Recall demonstrate that we obtained False Negatives. Both MDF and DCT features gave good results and combining them with Cartesian and Polar coordinates resulted in the best original/forged identification. According to figures 5.11 and 5.12 there are individuals whose original/forged signatures were almost perfectly identified which makes us believe that with further research, the results can be improved. We were expecting better results when compared to the first experiment, thus we conclude that the identification of original and forged signatures with current features requires further research. However, excluding the confusing features described in the beginning of this paragraph, we did obtain a lower FPR when compared with the first experiment.

Another observation is that figures 5.13 and 5.14 clearly demonstrate a problem when combining both features MDF and DCT. When using the DCT feature, the best results are obtained with increased γ and penalization constant C. On the other hand, using the MDF feature requires a smaller γ and a higher C in order to achieve better performance. Thus, it is hard to conciliate both features unless some transformation is applied to one of the features, in order to make both compatible with the same parameters. Another and simpler solution could be the use of a specialized SVM to separate the kernel parameters for each feature.

Regarding the last experiment, the One-Against-One person identification task, we obtained in general excellent results. Excluding some confusing features of which K-Means and Max-Intensity-Angle are an example, most features resulted in Accuracy and F-Score above 80%. Some features like MDF, Cartesian and Polar Coordinates gave F-Scores above 90%, especially the MDF with a F-Score of 97.80%. Combining both MDF and DCT features allowed an Accuracy of 98.13%, a F-Score of 98.14% and the highest Precision and Recall, 97.88% and 98.83%, respectively. The statistical summary of this combination can be seen in figure 5.15.

5.6 Conclusion

From the results we obtained in both the first and second experiments, the identification of original and forged signatures, we can conclude that although these are promising, for a better identification system further research is required. We obtained similar results to the ones acquired by Ribeiro et al. in [43] despite the fact that in their article they used a different classifier, Deep Belief Networks. These networks employ deep learning techniques which allow them a higher level of abstraction, specially in visual recognition tasks [43], possibly more oriented for the problem of off-line signature recognition. In accordance with their article, we confirm that both DCT and MDF features gave the best results.

When faced with the problem of identifying a person's signature, our study demonstrated that this task is accessible and provides excellent results, even with a single feature (MDF). Therefore, the construction of a multi-class signature identification system for identifying the author is possible and should give promising results.

Chapter 6

Conclusions and future work

As the amount of data produced by humans and machines grows at an unparalleled rate, fast machine learning algorithms that can extract relevant and useful information from large repositories of data become extremely important. To cope with the increasingly computational performance demands, the challenge consists of building multi-core implementations of machine learning algorithms. Recently the Graphics Processing Unit (GPU) has become a major player in this context. The rapid evolution of the GPU from a fixed-function device into a fully programmable one and its inherent parallelism made this very attractive architecture capable of accelerating applications by one or more orders of magnitude.

In particular, the realization of Support Vector Machines (SVMs) in the GPU is specially attractive. SVMs are perhaps the most widely used algorithm, possessing state-of-the-art generalization characteristics. Moreover, SVMs have been successfully in many scientific domains proving their usefulness.

In this Dissertation we aimed at developing a GPU implementation of the SVMs algorithm as part of the effort to develop an open-source GPU Machine Learning library (GPUMLib). To this end, the NVIDIA CUDA (Compute United Device Architecture) was used. This architecture provides a programming model for its GPUs with an adequate API for non-graphics applications using standard ANSI C, extended with keywords that designate data-parallel functions.

The results we obtained with both our CPU and GPU classifiers are promising and bring faster execution times to either desktop or mobile computers while offering similar results to the well known LIBSVM classifier. With training speedups up to 13.61× and classification speedups up to 265.85× using our GPU SVM, not only more complex tasks can be solved in less time as fine tuning of the SVM parameters using grid-search can be done. Even if there may be no GPU available, our Multi-Threaded CPU SVM achieves faster times than LIBSVM up to

6.90× in training and 3.58× in classification (excluding smaller datasets). Thus, our Multi-Threaded CPU allows complicated problems to be solved on modern multi-core CPUs.

However, in the course of this thesis we did not implement advanced features present in another SVM classifiers, as kernel caching and advanced heuristics, which can drastically reduce the kernel dot-products calculus and fasten the convergence. Another important factor is the support for double precision floating point arithmetic in the GPU, if supported by the device. The increased mathematical precision should, in theory, improve the SVM's training and classification performance. The technical improvements we referred can be added to our classifiers as future work, including vectorization support for our CPU classifier.

The inclusion of the UKF kernel gives additional value to both our CPU and GPU SVMs as it offers good generalization properties when compared to the well known RBF kernel. The use of this generic kernel can make the use of the other kernels superfluous, however, with the increased cost of fine-tuning the additional kernel parameters.

The GPU SVM was used to study the problem of handwritten off-line signature recognition. By using our GPU SVM we were able to study a complicated dataset, composed of 16200 signatures and draw conclusions for further investigation. Our study allowed us to identify the best features and their combination in order to identify if a given signature is original or forged and the person affiliated with a given signature.

We obtained interesting results in the identification of forged signatures but we give attention to the fact that better feature extraction is needed to improve the current detection performance. This fact is emphasised by the F-Scores of 71.62% and FPR of 7.97% we achieved. Additionally, better performance can also be achieved by future work in the form of specialized classifiers, supporting fine-tune of kernel parameters for each feature.

Finally, we highlight the excellent results obtained in the identification of the signature's author. In our opinion, the achievement of 98.14% F-Score clearly demonstrate that this task, excluding the computation time, is simple and accessible. Intuitively, this is natural as the signature of each individual is practically unique and distinguishable for another person. Thus, current classifiers, from which the SVMs are a good example of, can easily cope with the task and give excellent results.

Bibliography

[1] Rie Kubota Ando and Tong Zhang. A framework for learning predictive structures from multiple tasks and unlabeled data. *Journal of Machine Learning Research*, 6:1817–1853, 2005.

[2] S. Armand, M. Blumenstein, and V. Muthukkumarasamy. Off-line signature verification using an enhanced modified direction feature with single and multi-classifier approaches. *Computational Intelligence Magazine, IEEE*, 2(2):18 –25, may 2007.

[3] A. Asuncion and D.J. Newman. UCI machine learning repository, 2010. Available at http://archive.ics.uci.edu/ml/index.html.

[4] N.E. Ayat, M. Cheriet, C.Y. Suen, and M. Cheriet C. Y. Suen. Kmod - a two-parameter svm kernel for pattern recognition. In *In ICPR*, pages 30331–30334, 2002.

[5] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, New York, Inc. Secaucus, NJ, USA, 2006.

[6] M. Blumenstein, X.Y. Liu, and B. Verma. A modified direction feature for cursive character recognition. In *Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on*, volume 4, pages 2983 – 2987 vol.4, july 2004.

[7] David Blythe. Rise of the graphics processor. *Vol. 96, No. 5, May 2008 — Proceedings of the IEEE*, 96:761–778, 2008.

[8] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. Fast support vector machine training and classification on graphics processors. In *Proceedings of the 25th international conference on Machine learning*, ICML '08, pages 104–111, New York, NY, USA, 2008. ACM.

[9] Chih-Chung Chang and Chih-Jen Lin. *LIBSVM: A Library for Support Vector Machines*. Department of Computer Science National Taiwan University, Taipei, Taiwan, May 2011.

[10] Badong Chen, Songlin Zhao, Pingping Zhu, and José Carlos Príncipe. Quantized kernel least mean square algorithm. *IEEE Trans. Neural Netw. Learning Syst.*, 23(1):22–32, 2012.

[11] Corinna Cortes and Vladimir Vapnik. Support-vector networks. In *Machine Learning*, pages 273–297, 1995.

[12] K. De Brabanter, J. De Brabanter, J. A. K. Suykens, and B. De Moor. Optimized fixed-size kernel models for large data sets. *Comput. Stat. Data Anal.*, 54(6):1484–1504, June 2010.

[13] Joaquim Marques de Sá. *Pattern recognition: concepts, methods, and applications*. Springer, 2001.

[14] Rong-En Fan, Pai-Hsuen Chen, and Chih-Jen Lin. Working set selection using second order information for training support vector machines. *Journal of Machine Learning Research*, 6:1889–1918, December 2005.

[15] M. Ferrer, J. Alonso, and C. Travieso. Offline geometric parameters for automatic signature verification using fixed-point arithmetic. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 27(6):993–997, 2005.

[16] J. Gonçalves, Noel Lopes, and B. Ribeiro. Multi-thread support vector machines for pattern recognition. In *International Conference on Neural Information Processing*. Springer, November 2012.

[17] Ricardo Gutierrez. Intelligent sensor systems - lecture 13: Validation, March 2001.

[18] Ricardo Gutierrez-Osuna. Pattern analysis - lesson 10: Linear discriminants analysis, November 2011.

[19] Mark Harris. Optimizing parallel reduction in cuda. Internet, 2007.

[20] Sergio Herrero-Lopez, John R. Williams, and Abel Sanchez. Parallel multi-class classification using SVMs on GPUs. In *GPGPU 10*, pages 2–11, 2010.

[21] L. Hoegaerts, J. A. K. Suykens, J. Vandewalle, and B. De Moor. Subset based least squares subspace regression in rkhs. *Neurocomput.*, 63:293–323, January 2005.

[22] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy. Improvements to platt's smo algorithm for svm classifier design. *Neural Comput.*, 13:637–649, March 2001.

[23] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. pages 1137–1143. Morgan Kaufmann, 1995.

[24] Chong-Jin Ong J. Q. Zhang Uvaraj Periyathamby Xiu Ju Fu H. P. Lee L. J. Cao, S. S. Keerthi. Parallel sequential minimal optimization for the training of support vector machines. In *IEEE Transactions on neural networks*, volume 17, pages 1039–1049, 2006.

[25] Tsung-Kai Lin and Shao-Yi Chien. Support vector machines on gpu with sparse matrix format. In *Proceedings Ninth International Conference on Machine Learning and Applications (ICMLA 2010)*, pages 313–318. IEEE Computer Society, 2010.

[26] N. Lopes, B. Ribeiro, and R. Quintas. GPUMLib: A new library to combine machine learning algorithms with graphics processing units. In *Hybrid Intelligent Systems (HIS), 2010 10th International Conference on*, pages 229–232, August 2010.

[27] Noel Lopes, Daniel Correia, Carlos Pereira, Bernardete Ribeiro, and Ant'onio Dourado. An incremental hypersphere learning framework for protein membership prediction. In *Int. Conf. on Hybrid Artificial Intelligence Systems*, LNCS 7208, pages 429–439, 2012.

[28] Noel Lopes, B. Ribeiro, and João Gonçalves. Restricted boltzmann machines and deep belief networks on multi-core processors. In *IEEE World Congress on Computational Intelligence (WCCI 2012)*, pages 1 – 7, Brisbane, Australia, June 2012.

[29] Noel Lopes and Bernardete Ribeiro. GPUMLib: An efficient open-source GPU machine learning library. *International Journal of Computer Information Systems and Industrial Management Applications*, 3:355–362, 2011.

[30] Tom Mitchell. *Machine Learning*. McGraw-Hill, Columbus, OH, 1997.

[31] W.J.; Nickolls, J.; Dally. The gpu computing era. *IEEE Micro*, 30:56 – 69, 2010.

[32] NVIDIA. Geforce 256 - the world's first gpu. Internet, August 1999.

[33] NVIDIA. Pixel shaders - a facet of the nfinitefx engine. Internet, March 2001.

[34] NVIDIA. Vertex shaders - a facet of the nfinitefx engine. Internet, March 2001.

[35] NVIDIA. Nvidia geforce 8800 gpu architecture overview. Internet, November 2006.

[36] NVIDIA. *CUDA Programming Guide Version 4.0*, August 2011.

[37] E. Osuna, R. Freund, and F. Girosi. An improved training algorithm for support vector machines. In *Proceedings IEEE Neural Networks in Signal Processing NNSP'97*, pages 276–285. IEEE Computer Society, 1997.

[38] John D. Owens, Mike Houston, David Luebke, Simon Green, John Stone, and James Phillips. GPU computing. *Vol. 96, No. 5, May 2008 — Proceedings of the IEEE*, 96:879 – 899, 2008.

[39] John C Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. *Advances in Kernel Methods - Support Vector Learning*, 208(MSR-TR-98-14):1–21, 1998.

[40] Mengyu Qiao, Andrew H. Sung, and Qingzhong Liu. Feature mining and intelligent computing for mp3 steganalysis. In *Proceedings of the 2009 International Joint Conference on Bioinformatics, Systems Biology and Intelligent Computing*, IJCBS '09, pages 627–630, Washington, DC, USA, 2009. IEEE Computer Society.

[41] Ashu Rege. An introduction to modern gpu architecture, 2008.

[42] Bernardete Ribeiro. *Master in Informatics Engineering - Pattern Recognition Techniques*. September 2009.

[43] Bernardete Ribeiro, Ivo Gonçalves, Sérgio Santos, and Alexander Kovacec. Deep learning networks for off-line handwritten signature recognition. In *Proceedings of the 16th Iberoamerican Congress conference on Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, CIARP'11, pages 523–532, Berlin, Heidelberg, 2011. Springer-Verlag.

[44] Jeff Schneider. Cross validation, February 1997.

[45] B. Schölkopf, S. Mika, C.J.C. Burges, P. Knirsch, K.-R. Müller, G. Rätsch, and A. Smola. Input space vs. feature space in kernel-based methods. *IEEE Transactions on Neural Networks*, 10:1000–1017, 1999. IEEE Transactions on Neural Networks.

[46] Alexander J. Smola, Peter Bartlett, Bernhard Schölkopf, and Dale Schuurmans (Eds.). *Advances in Large Margin Classifiers*. The MIT Press, first edition, October 2000.

[47] Ivor Spital. *Sinclair ZX Spectrum +2A 128K Manual*. AMSTRAD Plc, 1987.

[48] Sergios Theodoridis and Konstantinos Koutroumbas. *Pattern recognition*. Academic Press, 2009.

[49] Vladimir Vapnik. *The nature of statistical learning theory*. Springer-Verlag, 1995.

[50] Vladimir Vapnik. *Statistical Learning Theory*. Wiley New York, Inc., 1998.

[51] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[52] Wei Wen Wu. Beyond business failure prediction. *Expert Systems with Applications*, 37:2371–2376, 2010.

[53] Rui Zhang and Wenjian Wang. Facilitating the applications of support vector machine by using a new kernel. *Expert Systems with Applications*, 38:14225–14230, 2011.

[54] A. Zien, G. Rätsch, S. Mika, B. Schölkopf, T. Lengauer, and K.-R. Müller. Engineering support vector machine kernels that recognize translation initiation sites. *BioInformatics*, 16(9):799–807, 2000.