GPU Assisted Implementation of Kernel-Based Template Attacks

Schriftliche Prüfungsarbeit
für die Master-Prüfung des Studiengangs Angewandte Informatik
an der Ruhr-Universität Bochum

vorgelegt von

Malysiak, Darius

Abgabedatum
01.06.2011

Name des 1.Prüfers
Prof. Dr.-Ing. Christof Paar

Name des 2.Prüfers
Prof. Dr.-Ing. Tim Güneysu

Erklärung

Ich erkläre, dass das Thema dieser Arbeit nicht identisch ist mit dem Thema einer von mir bereits für ein anderes Examen eingereichten Arbeit.
Ich erkläre weiterhin, dass ich die Arbeit nicht bereits an einer anderen Hochschule zur Erlangung eines akademischen Grades eingereicht habe.

Ich versichere, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen der Entlehnung kenntlich gemacht. Dies gilt sinngemäß auch für gelieferte Zeichnungen, Skizzen und bildliche Darstellungen und dergleichen.

Datum　　　　　　　　　　　　　　　　　Unterschrift

Abstract

This thesis evaluates the use of kernel-based methods in the context of power trace classification/identification.
Most previous works regarding the classification of power traces involve only common methods from statistical learning theory. These methods exploit similarities within power traces in a linear fashion. This work however, represents the first approach to this topic by analyzing nonlinear similarities with a specific technique, more precisely, we utilize kernel methods. With this approach one can construct a more useful representation for power traces, which then can be used for the actual identification of the power trace. Compared to the use of the previously mentioned common methods, kernel methods yield, in general, a visible improvement regarding the success rate of classification. Yet this approach also yields a high computational complexity, within this thesis we present methods to reduce this drawback by utilizing the features of modern GPUs.
We show how different kernel-based methods perform on the classification of power traces and we evaluate the benefit of applying GPUs for data processing. Additionally we developed a method which is capable of improving the results of common kernel methods.

to my grandfather

Contents

Acronyms

$AvEr(U)$	average recognition error on the set U page 44
$Av(U)$	average recognition rate on the set U page 45
w_0, b	bias, offset/translation of an arbitrary hyperplane page 36
$\mathcal{E}$	canonical unit base $\mathcal{E} = \{\mathbf{e}^1,\ ...,\ \mathbf{e}^N\}$ of $\mathbb{R}^N$ page 12
$\mathbf{e}^\mu$	canonical unit vector $\mathbf{e}^\mu = (0, ..., 0, \underbrace{1}_{\mu\text{-th element}}, 0, ..., 0) \in \mathbb{R}^N$ page 12
$\tilde{K}$	centered Gram matrix page 19
$\chi(\lambda)$	characteristic polynomial of an arbitrary matrix A page 23
$\mathcal{C}_i$	class i in a classification task page 35
G	CUDA grid which holds the thread blocks $T_{i,j}$ page 49
$t_{i,j,k}$	single CUDA thread (i, j, k) page 49
N_i	amount of data points inside class i page 35
$det(A)$	determinant of an arbitrary matrix A page 23
$\text{diag}(x_1, ..., x_n)$	diagonal matrix with values $x_1, ..., x_n$ on its diagonal and zeros on the remaining positions page 23

$<,>_{\mathcal{H}}$	dot product in feature space page 17
Λ	eigenvalue matrix, a diagonal matrix with eigenvalues of an arbitrary matrix A on its diagonal and zeros on the remaining positions page 23
$\mathcal{H}$	feature space, an abstract function space with the structure of a Hilbert space page 17
$G(i,k,\theta)$	Givens rotation encoded by three scalar values (i,k,θ) page 30
$\boldsymbol{\alpha}$	eigenvector of the Gram matrix page 18
K	Gram matrix page 18
β	Householder coefficient β which together with the Householder vector $\mathbf{v}$ fully describes a Householder reflection page 24
P_v	Householder matrix for a non-zero vector $\mathbf{v}$ page 24
H	separating hyperplane page 36
${}^{i}\mathbf{x}^{j}$	indexed base vector j of base $\mathcal{B}_i$ page 12
$\mathbf{x}^{j}$	indexed vector page 10
I	input space consisting of real valued data page 11
k	kernel function $k : I \times I \to \mathbb{R}$ page 18
$L(.)$	Lagrange function, formulation of an optimization problem subject to at least one restriction, as one function page 41

$A_{\mathcal{B}}$	matrix form of the linear mapping A regarding a base $\mathcal{B}$ page 15
$E[X]$	mean of random variable X corresponding to the given data distribution, e.g., $E[X] = \frac{1}{M}\sum_{\mu=1}^{M}\mathbf{x}^{\mu}$ page 11
∇	nabla operator, $\nabla := (\frac{\partial}{\partial x_1}, ..., \frac{\partial}{\partial x_n})^T$ page 42
$\|\mathbf{x}\|_{\mathcal{H}}$	2-norm in feature space $\|\mathbf{x}\|_{\mathcal{H}} := \sqrt{<\mathbf{x},\mathbf{x}>_{\mathcal{H}}}$ page 39
$\mathrm{Ker}(Q)$	nullspace $\mathrm{Ker}(Q) := \{x \in \mathbb{R}^N \mid Qx = 0\}$ of the linear mapping $Q : \mathbb{R}^N \to \mathbb{R}^N$ page 16
$[p]$	set $\{1, ..., p\} \subset \mathbb{N}$ page 11
$\mathrm{off}(A)$	off-diagonal norm, euclidean norm of the elements which do not lie on the diagonal of the matrix A page 29
x	scalar value, i.e., $x \in \mathbb{R}$, e.g., a vector element x_1 of a vector $\mathbf{x}$ page 11
ξ_m	slack variable regarding data point $\mathbf{x}_m$ page 42
$< ., . >$	standard euclidean scalar product $< \mathbf{x}, \mathbf{y} >:= \mathbf{x}^T\mathbf{y}$ page 13
$\mathcal{T}$	a template page 44
$T_{i,j}$	CUDA thread block (i, j) page 49
$\mathbb{I}_N$	canonical $N \times N$ unit matrix with ones on its diagonal and zeros on the other positions page 23
$\mathbf{x}$	vector, e.g., $\mathbf{x} \in \mathbb{R}^N$ page 11
$\mathbf{x}_{\mathcal{B}}$	coordinates of a vector $\mathbf{x}$ regarding the base $\mathcal{B}$ page 13

$\mathbf{w}$	weight vector, orthonormal vector on an arbitrary hyperplane page 36

1. Introduction

The term *side channel attack* is very unspecific. It can refer to many different scenarios, e.g., one could measure and record the electromagnetic emissions of an electronic car key and use this recorded information to bypass the car lock. This bypassing can also be done in many different ways, e.g., a simple replay of the emission could be sufficient or one might analyze the recording and reverse engineer the communication protocol, thus gaining the ability to generate new messages.
The scenario which will be addressed in this thesis is as follows. The device to be analyzed is a common PIC microcontroller. During its operation the resulting electrical power is recorded. The goal is to reconstruct the instructions executed by the controller using only the recorded data. Approaching this task can be done in following steps:

- record enough data (power traces),
- learn patterns in the data,
- use the gained knowledge to reconstruct instructions from new data.

This is a pretty abstract view on the problem as it leaves more questions than answers, how should the patterns be *learned* and how can this knowledge afterwards be used on unknown data? The form of a power trace depends on the instructions executed during the measurement, e.g., an *ADD* instruction yields a different power pattern than a *SUB* command. One possibility would be using techniques from statistical learning theory , e.g., a combination of *principal component analysis* and *linear discriminant analysis* (both terms will be explained in later chapters). This approach has been evaluated by [Weg09] who gained first results with the use of *hidden Markov models*.
This thesis evaluates the application of so-called kernel based methods which could be described as improved methods from statistical learning theory. The term *improved* refers to a higher success rate in learning patterns. We will now translate the problem stated above into terms of learning theory. The recorded data is called training data . It is used to *train* an algorithm in terms of determining certain constants within this algorithm. Such an algorithm is called a classifier which, as its name indicates, classifies the given data into specific classes. In our case this corresponds to classifying the unknown data into groups of instruction types, e.g., 25 instruction types as *ADD*,*SUB*,etc., yield 25 corresponding classes.

A *good* classifier could enable us to reverse engineer the program on a microcontroller, it may even expose constant and variable data used within this program. Kernel methods have proven to be very effective in a variety of classification tasks, e.g., image classification [SG10], OCR [TKC07] or noise classification [Tha06]. These methods are able to determine the nonlinear correlation between random variables and thus allow a new approach to classification tasks. Yet this effectiveness comes with a price in the area of computational effort. Kernel methods require a great amount of computation power compared to their non-kernel counterparts. Thus one major task is to find and implement algorithms which solve the underlying problem efficiently. The algorithms used in this thesis have been chosen according to their potential in parallel execution, the implementation utilizes the capabilities of modern GPUs to gain a great speed up in comparison to the standard CPU implementations.

1.1. A Short Outline of this Thesis

As described above the following chapters will gradually introduce/construct the concepts needed for a deeper understanding of the work involved in this thesis. Due to the nature of the topic, these chapters use a precise mathematical notation. Within all the upcoming details, it is easy to loose focus on the main goal or the motivation behind these needed steps. Thus this section will give a brief and abstract outline of the whole thesis and describe the motivation behind the content of each chapter.
Our main goal is to construct an algorithmic solution for the problem of identifying power traces of unknown instructions. A power trace, in it's original form, is nothing more than a series of numbers. Thus to recognize unknown traces, one could compare these numbers to a power trace of a known instruction. Yet this approach contains a certain instability, two traces from the same instruction type do not need to be absolutely identical. There is always a small amount of fluctuations involved, additionally power traces of different instruction types may look very similar, thus variations within the measurement can increase the difficulty to successfully compare and identify those traces.
The solution to this problem is to find a more sophisticated representation for traces, in other words, the numbers representing a single power trace must be exchanged for *something* different. The method within this thesis uses statistical characteristics within a set of recorded power traces to create a meaningful representation for each trace type, i.e., the traces for a certain instruction type. In a nutshell explained, the numbers representing a single trace will be mapped onto a mathematical function. Thus a function will represent a power trace and we will essentially compare functions to recognize the corresponding instruction type.

An example would be the following situation. Let us assume we have recorded many traces for the *ADD* instruction type, i.e., we have measured the power values during the execution of many *ADD* commands (with varying operands). The mentioned statistical properties within all these traces will then be exploited to create a *stable* representation P_{ADD} for the *ADD* type. This representation will be an abstract mathematical function. During the construction of this representation we additionally gain an algorithm which allows us to create representations of the same kind for unknown power traces. Thus we obtain the ability to transform given power traces to abstract functions, which then can be compared against P_{ADD}. Of course this view is a very macroscopic one; the whole process is more complicated. Yet the goal still remains the same, the construction of an alternate, more practicable representation for power traces. Chap. 2 introduces the techniques behind this approach in great detail.
Although the approach from above supports us on the way to reach our main goal, it also brings certain mathematical problems with it. The algorithm which transforms the traces into abstract mathematical functions needs the eigenvectors of a specific matrix. This matrix will be described very detailed in the mentioned chapter. For now let us say that this matrix can reach huge dimensions. Thus determining the eigenvectors becomes a less trivial task, one will face numerical instabilities and long computation times. This problem must be solved in order to increase our chances of recognizing unknown traces. We will introduce two methods from numerical mathematics which both are able to calculate the desired eigenvectors. Each method brings benefits and drawbacks with it, yet both of them have one common problem: the computation time. For this thesis, both algorithms have been implemented with GPU support, increasing their efficiency in that way. These algorithms will be introduced in Chap. 3.
Increasing the efficiency of these two methods is not an easy task, our developed acceleration techniques seek to exploit parallelism within them. Modern GPUs allow the parallel execution of many threads, thus parallel segments within the mentioned algorithms could benefit from the use of a GPU. Yet GPUs also provide an additional obstacle. The architecture of a GPU requires different approaches for implementing these parallel segments. To enable the reader of this thesis to understand our implementation, Chap. 5 provides a short introduction into the structure of modern GPUs and describes important optimization techniques used within this thesis.
Up until now, only the transformation of power traces and the involved problems have been mentioned. The transformation was introduced with the motivation to enhance the recognition of unknown traces, i.e., increase the chance to recognize a trace correctly. But how can one recognize an unknown power trace? How can one assign the correct instruction type to a given trace? We are left with these questions once we have solved all previous problems, e.g., calculating the eigenvectors efficiently with the help of a GPU. The problem of recognizing unknown power traces can be interpreted as a classification problem, where one

tries to assign a power trace to a certain instruction type. Within this thesis we have utilized different classification techniques from statistical learning theory to solve the recognition task. Chap. 4 will introduce each of these methods and explain the theory behind them.
Thus the way to our main goal consists of

1. finding a robust representation for power traces
2. choosing/developing suitable classification methods to recognize power traces with a high success rate
3. developing an efficient implementation of all involved methods (e.g., with the help of a GPU)

The implementation of most algorithms was specifically developed for this thesis, as for most of them currently no previous work exists for a GPU based acceleration. Additionally, regarding the algorithms for recognition, no previous work exists which evaluates their potential for the recognition of power traces. Furthermore we have developed new methods to enhance the recognition rate, these methods will be explained in Sect. 6.6. Chap. 6 describes our implementation of each involved algorithm.
Following the explanation of our implementation, Chap. 7 will present the corresponding results, e.g., computation times or recognition rates.

2. Kernel Principal Component Analysis

This chapter introduces the concept of *feature extraction* and explains *principal component analysis* (PCA) as a common method to perform this extraction. Furthermore we will see how kernel methods work and how they can be applied to enhance PCA. In preparation for chapter 6 we will also address the major practical problems (e.g. algorithm complexity and numerical stability) that arise when considering the actual implementation of PCA.
The following definitions, theorems and algorithms are based on or cited from [SS02],[Wis10].

2.1. Introduction

The general approach to classification tasks can be described in two phases. First the so-called feature extraction and second the actual classification task.
Feature extraction refers to the process of either selecting or creating *significant* values of the data which help to characterize the underlying distribution, irrelevant information is discarded in this process. The term *creating* refers to mapping the given data onto another set.
This chapter introduces the principal component analysis, which represents a standard method for feature extraction. Afterwards, to overcome certain limitations of PCA, the Kernel-PCA (KPCA) will be introduced. Before we begin with the precise mathematical description of PCA, we will look at an example of feature extraction. Let us consider a single power trace x_1 as shown in Fig. 2.1. Additionally we assume that Fig. 2.1 shows a complete power trace. Let Ξ be the set of power traces and $\phi : \Xi \to \mathbb{N}$ a mapping which counts all maxima above the threshold ϑ (as indicated by the dashed line). Thus we have $\phi(x_1) = 2$. Every element of $\phi(\Xi)$ is called a feature of one or more corresponding traces. ϕ is called a feature extractor .

Further we assume that the amount of maxima above the threshold allows us to characterize the power trace, e.g., $\phi(x_i)$ is sufficient enough to distinguish certain trace types. Therefore classification can be carried out on $\phi(\Xi)$ instead of Ξ, i.e., it is computationally easier to work on integer values instead of vectors (a power

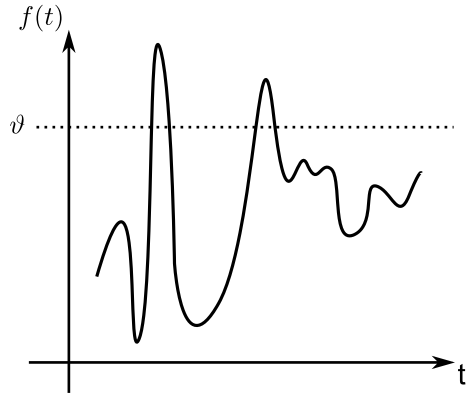


Fig. 2.1.: Possible feature extraction of a power trace

trace can be described as a vector of measurement samples, see Fig. 2.2). In praxis of course such a feature extraction is not a very feasible approach, because the number of peaks will not characterize the trace type, e.g., five peaks above a threshold will not be a very reliable indicator for a specific instruction.

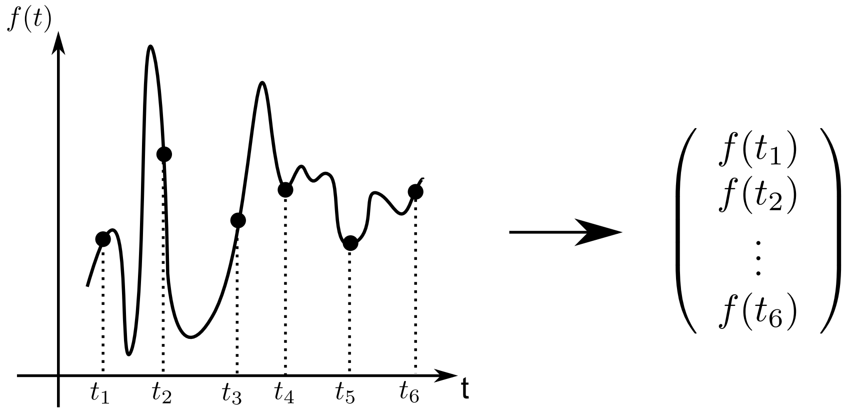


Fig. 2.2.: Representation of a power trace as a vector

2.2. Principal Component Analysis

PCA is a very well-established method for feature extraction. In this chapter we will first precisely define the PCA concept and afterwards develop an intuitive understanding of the mathematical concepts. Roughly described PCA works by mapping the data points $\{\mathbf{x}^\tau\}_{\tau\in\{1,\ldots,m\}}$ into a new coordinate system and zeroing out certain elements of each new coordinate vector $\tilde{\mathbf{x}}^\tau$. The vector $\tilde{\mathbf{x}}^\tau$ represents

the feature(vector) for the data point $\mathbf{x}^\tau$ and can be used for classification instead of $\mathbf{x}^\tau$.
A very precise definition for the PCA problem is given by [Wis10]:

Definition 2.2.1. *Given a set* $\{\mathbf{x}^\mu \mid \mu = 1, ..., M\}$ *of* N*-dimensional data points* $\mathbf{x}^\mu = (x_1^\mu, x_2^\mu, ..., x_N^\mu)^T$ *with zero mean, i.e.,* $E[X] = 0$*, find an orthogonal matrix* $\tilde{Q}$ *with determinant* $det(\tilde{Q}) = +1$ *generating the transformed data points* $\tilde{\mathbf{x}}^\mu := \tilde{Q}x^\mu$ *such that for any given dimensionality* P *the data projected onto the first* P *axes,* $\tilde{\mathbf{x}}_{||}^\mu = (x_1^\mu, x_2^\mu, ..., x_P^\mu, 0, ..., 0)^T$*, have the smallest*

$$\textit{reconstruction error } E := \frac{1}{M}\sum_{\mu=1}^{M} ||\tilde{\mathbf{x}}^\mu - \tilde{\mathbf{x}}_{||}^\mu||^2 \tag{2.1}$$

among all possible projections onto a P*-dimensional subspace.*
The row vectors of $\tilde{Q}$ *define the new axes and are called the principal components.*

Let I be the input space consisting of data points/samples $\mathbf{x}^i \in \mathbb{R}^n$, $i \in [M]$. Furthermore we assume that the underlying data distribution has zero mean, e.g., $E[X] = 0$, with X as the corresponding random variable which indicates the appearance of a data point. Otherwise we center the data by applying the transformation $\tilde{\mathbf{x}}_i = \mathbf{x}_i - E[X]$. We define the covariance matrix

$$C := \frac{1}{M}\sum_{\mu=1}^{M} \mathbf{x}^\mu \mathbf{x}^{\mu T} \in \mathbb{R}^{N\times N} \tag{2.2}$$

For a deeper and intuitive understanding of C one should consider the data distributions visualized in Fig. 2.3.

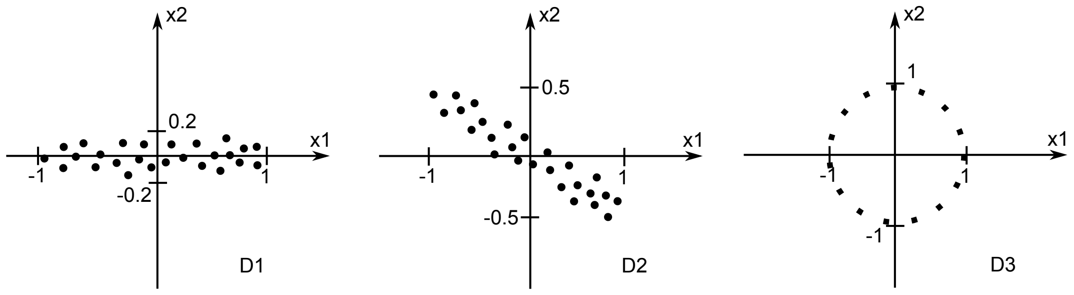


Fig. 2.3.: examples for 2-dimensional data distributions

Definition 2.2 is equivalent to

$$C_{i,j} := \frac{1}{M}\sum_{\mu=1}^{M} x_i^\mu x_j^\mu \tag{2.3}$$

thus we have the variance $Var(x_i) = C_{i,i}$ and the so-called covariance $Cov(x_i, x_j) = C_{i,j}$. The covariance is a measure for the linear correlation between the involved dimensions. On closer inspection of Fig. 2.2, one can verify the following approximated covariance matrices:

$$C_1 := \begin{pmatrix} 0.37 & -0.004 \\ -0.004 & 0.012 \end{pmatrix} C_2 := \begin{pmatrix} 0.37 & -0.2 \\ -0.2 & 0.11 \end{pmatrix} C_3 := \begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix} \tag{2.4}$$

The covariance matrix has diagonal form if and only if the random variables X_i are linearly independent, i.e., $C_{i,j} = 0 \; \forall i \neq j$. The core of the PCA algorithm needs diagonal covariance matrices, yet as shown above, not all covariance matrices are diagonal. It is obvious that C_3 already has diagonal form regarding the base $\mathcal{E}$, thus the question emerges if generally a base transformation can be found which diagonalizes the covariance matrix.

Theorem 2.2.1 (Diagonalization of symmetric matrices).
Let $C \in \mathbb{R}^N$ be a real valued symmetric matrix with eigenvalues λ_i and corresponding eigenvectors $\mathbf{v}^i$, $i = 1, ..., n$. C can be orthogonally diagonalized by

$$C = Q\,diag(\lambda_1, ..., \lambda_n)Q^T \tag{2.5}$$

where $Q := (\mathbf{v}^1 \; \mathbf{v}^2 \; ... \; \mathbf{v}^n)$ is an orthogonal $N \times N$ matrix (e.g. $Q^{-1} = Q^T$) consisting of C's normalized eigenvectors. Furthermore $\mathcal{B} := (\mathbf{v}^1, \mathbf{v}^2, ..., \mathbf{v}^n)$ forms an orthonormal base of $\mathbb{R}^N$.

This theorem (2.2.1) from linear algebra states that every symmetric real matrix can be orthogonally diagonalized by calculating its eigenvalues and eigenvectors. Taking into account that covariance matrices are always symmetric (see Eq. 2.3), we are always able to find a base $\mathcal{B}_2$ so that a covariance Matrix C (currently expressed regarding a base $\mathcal{B}_1$) will take diagonal form. For reasons of simplicity we temporarily assume that the following symmetric matrix C is already diagonalized regarding the base $\mathcal{B}_2 = (\mathbf{v}^1, \; \mathbf{v}^2, ..., \mathbf{v}^n)$ (e.g. $C = (\mathbf{v}^1 \; \mathbf{v}^2 \; ... \; \mathbf{v}^n)\text{diag}(\lambda_1, ..., \lambda_n)(\mathbf{v}^1 \; \mathbf{v}^2 \; ... \; \mathbf{v}^n)^T$). Without loss of generality we additionally assume that $\mathcal{B}_1 = \mathcal{E}$ (where $\mathcal{E}$ denotes the canonical base on $\mathbb{R}^n$) . The theoretical and practical techniques for diagonalization will be discussed in Chap. 3.

PCA projects the given data points $\mathbf{x}^\mu$ into a p-dimensional vector subspace $V \subseteq \mathbb{R}^N$. More precisely $V = \{{}^2\mathbf{b}^i \mid {}^2\mathbf{b}^i \in \mathcal{B}_2, \; i \in \{1, ..., p\}\}$ with $p \leq N$. Fig. 2.4 visualizes this projection for $N = 2$. For example if we want to project the point $\mathbf{x}^\mu$ onto the x_1 axis, we simply set $\mathbf{x}_2^\mu = 0$, the resulting vector $\tilde{\mathbf{x}}^\mu$ has the form $(\mathbf{x}_1^\mu, 0)^T$ and lies in a 1-dimensional subspace of $\mathbb{R}^2$. Of course we need an algorithm to achieve this projection for an arbitrary subspace choice. To achieve this,

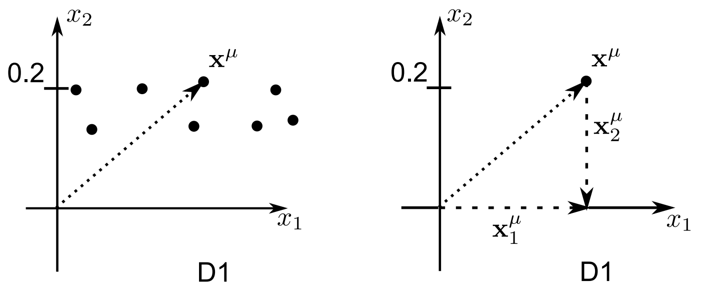


Fig. 2.4.: Projection of 2-dim data onto a 1-dim subspace

the PCA algorithm utilizes the base transformation matrix Q in the following way

$$\begin{aligned} Q^T\mathbf{x}^\mu &= (\mathbf{v}^1\ \mathbf{v}^2\ ...\ \mathbf{v}^n)^T\mathbf{x}^\mu && (2.6)\\ &= \begin{pmatrix} <\mathbf{v}^1,\mathbf{x}^\mu> \\ <\mathbf{v}^2,\mathbf{x}^\mu> \\ ... \\ <\mathbf{v}^n,\mathbf{x}^\mu> \end{pmatrix} && (2.7)\\ &=: \tilde{\mathbf{x}}^\mu && (2.8)\end{aligned}$$

One should remind himself that Qx is an element of span($\mathcal{B}_2$), thus if we leave out several columns from Q, i.e., we construct $\tilde{Q} = (\mathbf{v}_{i^1}\ \mathbf{v}^{i_2}\ ...\ \mathbf{v}^{i_p})$, we get a projection matrix $\tilde{Q}$ which projects $\mathbf{x}^\mu$ onto span($\mathbf{v}^{i_1}\ \mathbf{v}^{i_2}\ ...\ \mathbf{v}^{i_p}$). This holds for arbitrary orthonormal bases $\mathcal{B}_1$, e.g., it is not needed that ${}^1\mathbf{b}^i = \mathbf{e}^i$ (where $\mathbf{e}^i$ denotes the i–th canonical unit vector of $\mathbb{R}^N$).
The solution to the PCA problem can be algorithmically solved through following steps:

1. given the data points $\{\mathbf{x}^\mu\}_{\mu\in[N]}$, form the covariance matrix C
2. diagonalize C yielding $C = Q^T DQ$
3. choose needed dimensions and form $\tilde{Q}$
4. extract features $\tilde{\mathbf{x}}$ for a given test point $\mathbf{x}$ by calculating $\tilde{\mathbf{x}} := \tilde{Q}\mathbf{x}$

With this approach, if $i_p < N$, certain dimensions are discarded and information is lost. We will discuss the resulting error shortly. But first, for a better understanding of this dimensionality reduction, we are going to visualize the PCA concept in a less formal way.
Let us have a look on Fig. 2.5, it is important to distinguish between the terms data point and coordinates. The arbitrary point $\mathbf{x} = (1,2)^T$ has the coordinates $(1,2)$, yet these coordinates have no meaning unless we specify a base. It is a common assumption to consider $\mathcal{E}$ as this base and we denote that by writing $\mathbf{x}_\mathcal{E}$. Now consider another orthonormal base $\mathcal{B} = (\frac{1}{\sqrt{2}}(1,1)^T, \frac{1}{\sqrt{2}}(-1,1)^T)$, the point $\mathbf{x}$ can also be expressed as linear combination of those base vectors. To determine

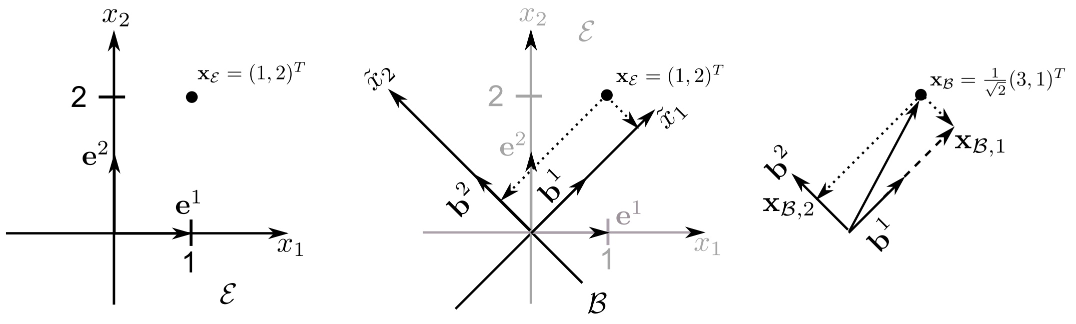

x_2
$\mathbf{x}_\mathcal{E} = (1,2)^T$
2
$\mathbf{e}^2$
$\mathbf{e}^1$
1
x_1
$\mathcal{E}$
$\tilde{x}_2$
$\tilde{x}_1$
$\mathbf{b}^2$
$\mathbf{b}^1$
$\mathcal{B}$
$\mathbf{x}_\mathcal{B} = \frac{1}{\sqrt{2}}(3,1)^T$
$\mathbf{x}_{\mathcal{B},1}$
$\mathbf{x}_{\mathcal{B},2}$

Fig. 2.5.: Example for 2-dimensional projection and different coordinate systems

the coordinates of $\mathbf{x}_\mathcal{E}$ regarding this base one simply has to project $\mathbf{x}_\mathcal{E}$ onto the base vectors, i.e., calculate

$$\begin{aligned}
\frac{1}{\sqrt{2}} < \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \mathbf{x}_\mathcal{E} > &= \frac{1}{\sqrt{2}} < \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \end{pmatrix} > & (2.9) \\
&= \frac{3}{\sqrt{2}} & (2.10) \\
&=: \tilde{x}_1 & (2.11) \\
\frac{1}{\sqrt{2}} < \begin{pmatrix} -1 \\ 1 \end{pmatrix}, \mathbf{x}_\mathcal{E} > &= \frac{1}{\sqrt{2}} < \begin{pmatrix} -1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \end{pmatrix} > & (2.12) \\
&= \frac{1}{\sqrt{2}} & (2.13) \\
&=: \tilde{x}_2 & (2.14)
\end{aligned}$$

Thus we have determined the coordinates $(\tilde{x}_1, \tilde{x}_2)^T = \frac{1}{\sqrt{2}}(3,1)^T$ of $\mathbf{x}$ in $\mathcal{B}$. This transformation can also be written in a more compact way

$$\mathbf{x}_\mathcal{B} = \frac{1}{\sqrt{2}} \begin{pmatrix} 3 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \end{pmatrix} = Q^T \mathbf{x}_\mathcal{E} \tag{2.15}$$

where the columns of Q consist of the base vectors of $\mathcal{B}$. Because Q is an orthogonal matrix the transformation back into the original coordinate system can be achieved by

$$\mathbf{x}_\mathcal{E} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix} \frac{1}{\sqrt{2}} \begin{pmatrix} 3 \\ 1 \end{pmatrix} = Q \mathbf{x}_\mathcal{B} \tag{2.16}$$

With this example it should be clear that (Eq. 2.7) actually represents two operations, firstly it transforms $\mathbf{x}_\mathcal{E} = \mathbf{x}_{\mathcal{B}_1}$ into $\mathbf{x}_{\mathcal{B}_2}$ and secondly it reduces the dimensionality of $\mathbf{x}_{\mathcal{B}_2}$. Thus the vector $\tilde{\mathbf{x}}$ is a representation of $\mathbf{x}$ in a p-dimensional subspace of $\mathbb{R}^N$, expressed in coordinates regarding the base $\mathcal{B}_2$. We refer to the transformation back into the original space, i.e., calculating $\tilde{Q}\tilde{\mathbf{x}}$, as reconstruction of the projected data point, with $\tilde{Q}$ as defined in the solution of the PCA

problem.
Similar to points we have to distinguish between a linear mapping $\Phi : \mathbb{R}^M \to \mathbb{R}^N$ and its matrix form Φ, for better readability we will use the same symbol for both. A linear mapping can be described by a matrix if we focus on a specific base, also here we normally consider $\mathcal{E}$ as this base. Let us assume we are given an arbitrary mapping

$$C : \mathbb{R}^2 \to \mathbb{R}^2, \quad C(\mathbf{x}) := \begin{pmatrix} x_1 + 2x_2 \\ 2x_1 + x_1 \end{pmatrix} \tag{2.17}$$

As mentioned before we assume $\mathcal{E}$ as the base for this mapping and denote this by $C_{\mathcal{E}}$ (we use the terms linear mapping and corresponding matrix form interchangeably). Thus C takes the following matrix form

$$C_{\mathcal{E}} = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix} \tag{2.18}$$

which is a symmetric matrix. We now want to find $C_{\mathcal{B}}$, i.e., the matrix form of C regarding the base $\mathcal{B}$. The desired matrix has the form (without proof)

$$C_{\mathcal{B}} = Q^T C_{\mathcal{E}} Q = \begin{pmatrix} 3 & 0 \\ 0 & -1 \end{pmatrix} \tag{2.19}$$

Thus the matrix we have looked for takes diagonal form, we say that $C_{\mathcal{E}}$ has been diagonalized. This procedure can not be applied to arbitrary matrices, yet for symmetric matrices we can find an orthogonal matrix Q such that $C_{\mathcal{B}}$ takes diagonal form (see theorem 2.2.1). To solve the PCA problem we look for a matrix Q such that $C_{\mathcal{B}}$, i.e., the covariance matrix calculated from transformed data points $\tilde{\mathbf{x}}^\mu = Q^T \mathbf{x}^\mu$ takes diagonal form. Furthermore Q needs to have a determinant of +1, i.e., Q represents a transformation into a (positively) rotated coordinate system as depicted in Fig. 2.5.
At this point we can finally analyze the reconstruction error E. To minimize this error, we have to find a new coordinate system such that the loss through projection onto the axes is minimized. This problem is visualized in Fig. 2.6. In the left picture we see the projection of one point $\mathbf{x}^\mu$ onto the first axis $\tilde{x}_1$ of the new (i.e. rotated) coordinate system. The reconstruction error of $\mathbf{x}^\mu$ is denoted by r. One can see that following equation holds

$$r^2 = d^2 - v^2 \tag{2.20}$$

where d stays constant in both cases, i.e., projection onto $\tilde{x}_1$ or $\tilde{x}_2$. The value of v^2 contributes to the variance of the data distribution in the new coordinate system while r^2 contributes to the total reconstruction error E. Thus it seems that minimizing the error is equivalent to maximizing the variance of the projected data, which has been proven in [Bis06].

Following theorem has been proven by [Wis10]:

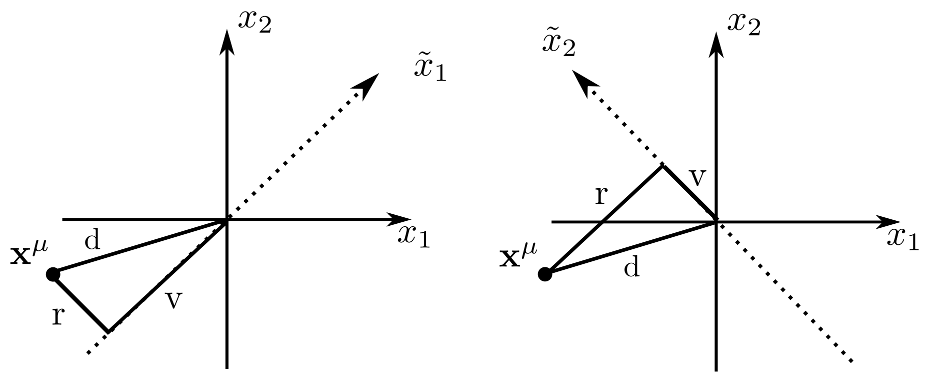


Fig. 2.6.: Reconstruction error r through projection onto 1-dim subspace

Theorem 2.2.2. *Let $\tilde{Q} := (\mathbf{v}_1\ \mathbf{v}_2\ ...\ \mathbf{v}_p)$ be the projection matrix constructed from C's eigenvectors $\mathbf{v}_i$ with corresponding eigenvalues λ_i. Furthermore let the eigenvalues be sorted in descending order. The resulting reconstruction error for the projected data is given by*

$$E = \sum_{i=p+1}^{N} \lambda_i \tag{2.21}$$

Thus the projection onto subspaces of Ker(Q) (where Ker(Q) denotes the nullspace of Q) does not increase the error. As of this, leaving out the eigenvectors $\mathbf{v}^i$ which correspond to an eigenvalue $\lambda_i = 0$, during the construction of $\tilde{Q}$, will not increase the error. Yet, leaving out the eigenvectors $\mathbf{v}^i$ which correspond to an eigenvalue $\lambda_i \neq 0$, will increase the projection error. Although we may have reduced the dimensionality of the input data, it is not a general result in feature extraction. An example would be Kernel-PCA which can increase the dimensionality of the input data, i.e., it can extract more than N features from each data sample.
From a statistical point of view, PCA can be described to find a new coordinate system in which the given data points become nearly uncorrelated (which is indicated by a nearly diagonal covariance matrix). In such a coordinate system the projection on the axis can yield a good characterization of certain data classes, i.e., the variance is maximized along these axis. Although PCA always finds such a system, there exist data distributions for which the computed solution yields no benefit at all, or in other words, for which the extracted features will not help in a classification task. Yet there exists a more powerful variant of PCA for such data distributions, the so-called kernel principal component analysis.

2.3. Kernel Principal Component Analysis

Kernel principal component analysis (KPCA) works in its core just like PCA, the main difference lies in using a nonlinear mapping which maps each given data point onto an abstract function. In other words, KPCA performs PCA with functions. We will see that the actual algorithm completely circumvents the use of abstract data structures for function handling. To understand KPCA one has to understand the principles of kernels or kernel functions.
Let us consider the following mapping

$$\Phi : \mathcal{X} \to \mathcal{H} \tag{2.22}$$

where $\mathcal{X}$ represents the set of given data points and $\mathcal{H}$ an abstract function space. We will refer to them as input and feature space , respectively, where in our case the feature space will always be a Hilbert space. An actual example for this mapping could be

$$\Phi : \mathcal{X} \to \mathcal{H}, \ \mathbf{x} \mapsto \exp\left(\frac{\|\mathbf{x} - .\|_2^2}{2\sigma^2}\right) =: k(\mathbf{x}, .) \tag{2.23}$$

which maps every given vector $\mathbf{x}$ onto a Gaussian function . This is visualized in Fig. 2.7 for a dimensionality of one. As $\mathcal{H}$ is a Hilbert space, it is equipped

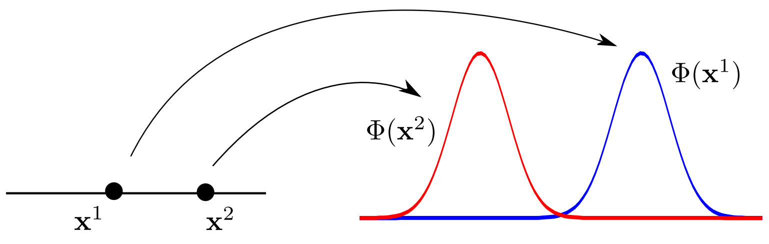


Fig. 2.7.: Mapping data points onto Gaussian functions

with a dot product $< ., . >_{\mathcal{H}}$ which could be defined, for $w_1 = \Phi(\mathbf{x}), w_2 = \Phi(\mathbf{y}) \in \mathcal{H}, \ \mathbf{x}, \mathbf{y} \in \mathcal{X}$ as[1]

$$\begin{aligned} < w_1, w_2 >_{\mathcal{H}} &= \ < k(\mathbf{x}, .), k(\mathbf{y}, .) >_{\mathcal{H}} & (2.24) \\ &:= \ k(\mathbf{x}, \mathbf{y}) & (2.25) \end{aligned}$$

One might recall that the standard euclidean dot product $< \mathbf{x}, \mathbf{y} >:= \mathbf{x}^T\mathbf{y}$ can be interpreted as a similarity measure. If two normed vectors $\mathbf{x}, \mathbf{y}$ are very *similar* regarding their orientation or position, the value $|< \mathbf{x}, \mathbf{y} >|$ will be close to 1. This view holds for arbitrary dot product spaces, e.g., if we restrict Φ to normed

[1]The proof that this definition fulfills the requirements of a dot product can be seen in [Vas09].

functions in $\mathcal{H}$, we receive a way to measure the similarity of $\mathbf{x^1}$ and $\mathbf{x^2}$ in $\mathcal{H}$. For convenience it is possible to define a function

$$k(\mathbf{x^1}, \mathbf{x^2}) =< \Phi(\mathbf{x^1}), \Phi(\mathbf{x^2}) >_{\mathcal{H}} \tag{2.26}$$

which is also called a kernel (function). In other words a kernel measures the similarity of two given data points in an abstract function space. It is important to note that the restriction to normed functions is only one of many possible ways to define a similarity measure. One might ask why this approach should be considered, it would also be possible to measure the similarity in the input space. One benefit of using kernels lies in the fact that a nonlinear mapping/transformation of the given coordinates is applied, we will see an actual example for the practicability of this mapping later in the current section. Another advantage of kernels is the possibility to avoid computation in $\mathcal{H}$, all calculations can be carried out in $\mathcal{X}$.
Kernel PCA utilizes this concept to carry out PCA in $\mathcal{H}$ without actually entering $\mathcal{H}$. Due to the complexity of the derivation of KPCA, this section will only explain the basic concepts required to perform KPCA. Proofs for the following claims can be found in [SS02] and [Bis06]. For now we assume that the given data points $\mathbf{x}^i$ are centered in feature space.

Definition 2.3.1 (Gram matrix).
Let $\Phi : \mathcal{X} \to \mathcal{H}$ be a nonlinear mapping and $\mathcal{X} := \{\mathbf{x}^1, ..., \mathbf{x}^M\} \subset I$. The Gram matrix $K \in \mathbb{R}^{M \times M}$ is defined by

$$K_{i,j} :=< \Phi(\mathbf{x}^i), \Phi(\mathbf{x}^j) >_{\mathcal{H}} \tag{2.27}$$

The Gram matrix can be constructed directly in input space if a corresponding kernel function exists, as with the covariance matrix the Gram matrix is always symmetric. A kernel is called positive definite if it gives rise to a positive definite Gram matrix. To perform KPCA one has to solve the following eigenvalue problem

$$M\lambda K\boldsymbol{\alpha} = K^2\boldsymbol{\alpha} \tag{2.28}$$

which is the analogy to the eigenvalue problem in standard PCA. Yet as shown in [SS02], it is equivalent to solve the following eigenvalue problem

$$M\lambda\boldsymbol{\alpha} = K\boldsymbol{\alpha} \tag{2.29}$$

which is more convenient than (2.28). Let $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_p$ denote all solutions to (2.29), i.e., all solutions $M\lambda$ with λ_p being the last nonzero value and $n = 1, ..., p$. The acquired eigenvectors α_n have to be normalized so they fulfill the equation

$$1 = \lambda_n < \boldsymbol{\alpha}^n, \boldsymbol{\alpha}^n > \tag{2.30}$$

In KPCA we are interested in projecting a data point $\mathbf{x}$ in feature space (i.e. $\Phi(\mathbf{x})$) onto the eigenvectors $\mathbf{v}^n$ of (2.28), i.e., computing $< \mathbf{v}^n, \Phi(\mathbf{x}) >_{\mathcal{H}}$. Also here the computation inside $\mathcal{H}$ can be avoided by

$$< \mathbf{v}^n, \Phi(\mathbf{x}) >_{\mathcal{H}} = \sum_{i=1}^{M} \boldsymbol{\alpha}_i^n k(\mathbf{x}^i, \mathbf{x}) \tag{2.31}$$

where k represents the corresponding kernel. It is important to note that KPCA can extract feature vectors with up to M (the number of data points) elements, thus it is able to extract statistical correlations of higher order, compared to standard PCA where each feature vector has an amount of maximal N (the number of dimensions of each point) elements.
So far we have assumed that the mapped data points are centered in feature space, yet this does not hold in the general case and thus the data points need to be centered after they have been mapped into feature space. Conveniently this can be done after we have used them to create the Gram matrix, this centering is also referred to as centering the Gram matrix .

Theorem 2.3.1. *Let $\Phi : \mathcal{X} \to \mathcal{H}$ be a nonlinear mapping and $\mathcal{X} := \{\mathbf{x}^1, ..., \mathbf{x}^M\} \subset I$. Centering $\Phi(\mathcal{X})$ before creating the Gram matrix K can be avoided and carried out after creation of K by calculating the centered Gram matrix $\tilde{K}$*

$$\tilde{K}_{i,j} := (K - \mathbf{1}_M K - K\mathbf{1}_M + \mathbf{1}_M K \mathbf{1}_M)_{i,j} \tag{2.32}$$

where $\mathbf{1}_M \in \mathbb{R}^{M \times M}$, $(\mathbf{1}_M)_{i,j} := \frac{1}{M}$. The matrix $\tilde{K}$ represents the Gram matrix based on the centered set $\Phi(\mathcal{X})$.

All properties of standard PCA also hold for KPCA especially the error function. For an illustration of the projection onto a principal component $\mathbf{v}^2$ one can refer to Fig. 2.8. Four data points $\mathbf{x}^i$, $i \in [4]$ are being used here for KPCA, each data point represents a single trace in input space.

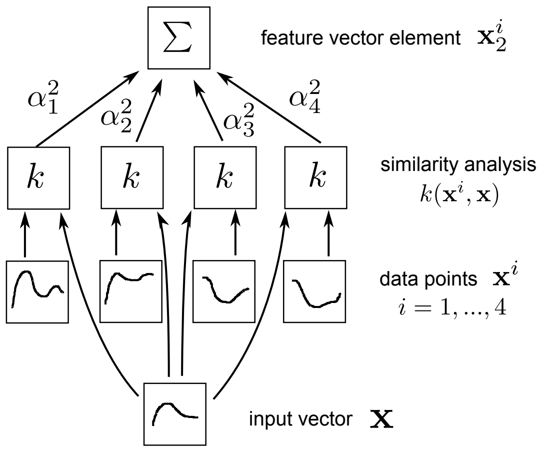

$\sum$
feature vector element $\mathbf{x}_2^i$
α_1^2
α_2^2
α_3^2
α_4^2
k
k
k
k
similarity analysis
$k(\mathbf{x}^i, \mathbf{x})$
data points $\mathbf{x}^i$
$i = 1, ..., 4$
input vector $\mathbf{x}$

Fig. 2.8.: KPCA with 4 data points (traces) $\mathbf{x}^i$ for a given trace $\mathbf{x}$

First the new trace $\mathbf{x}$ will be compared with the existing data points in feature space. The weighted sum of the comparison output, forms the projection onto a principal component in feature space, i.e., the coefficient for a linear combination of functions.
In this thesis following kernels have been used

$$\begin{aligned} k(\mathbf{x},\mathbf{y}) &:= <\mathbf{x},\mathbf{y}>^d,\ d \in \mathbb{N}\backslash\{0\} && \text{polynomial kernel,} \\ k(\mathbf{x},\mathbf{y}) &:= \exp\left(-\frac{\|\mathbf{x}-\mathbf{y}\|_2^2}{2\sigma^2}\right),\ \sigma \in \mathbb{R}\backslash\{0\} && \text{Gaussian kernel,} \\ k(\mathbf{x},\mathbf{y}) &:= \tanh(\kappa <\mathbf{x},\mathbf{y}> + \vartheta),\ \vartheta \in \mathbb{R}\backslash\{0\} && \text{sigmoid kernel,} \end{aligned}$$

where the gaussian and polynomial kernel are positive definite, i.e., yielding only positive eigenvalues. There is no general rule which kernel performs better regarding a specific problem. Polynomial kernels have shown to be very effective in OCR tasks ([SS02]), whereas Gaussian kernels yielded good results in image classification ([SG10]). As mentioned at the beginning of this section, the nonlinear mapping Φ has some interesting properties regarding the alignment of the data in feature space. This can be visualized for a not very abstract mapping. Fig. 2.9 shows a set of given data points on the left side, on the right side we have the mapped datapoints. The mapping used here is

$$\Phi(\mathbf{x}) := (x_1^2, x_2^2, \sqrt{2}x_1^2x_2^2) \tag{2.33}$$

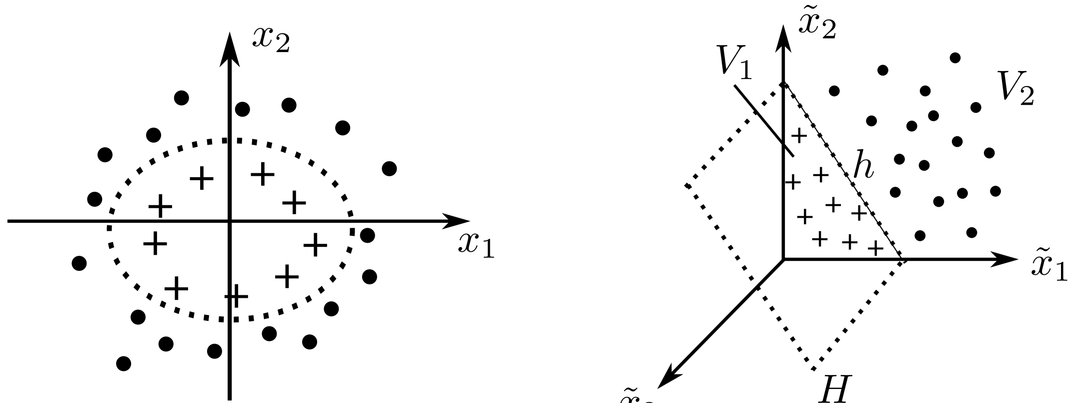


Fig. 2.9.: Alignment of data in feature space and a possible separation plane

The different symbols for the data points represent different classes, i.e., we have data points representing two classes. To classify an unknown data point one might draw a ‘fitting’ separation line through the data and assign the new data point to the class in which region it lies. In the input space this separation line takes the form of an ellipse, while in the feature space this ellipse becomes

a hyperplane. These results can also be deduced by only analyzing the mapping of Eq. 2.33. All points within the ellipse will be mapped to the volume V_1 (due to squaring the coordinates). The outer points of the ellipse will be mapped in the same way to the volume V_2 which lies side by side with the aforementioned. No single point of $Im(\Phi)$ can become negative in his coordinates, the points of the ellipse fulfill $ax^2 + by^2 = c$, in feature space they become linearly depended by $x^2 = c/a - b/ay^2$ and $y^2 = c/b - a/bx^2$, thus they form the line h. This line can be extended to a separating hyperplane H. Such an approach allows one to use linear classifiers in feature space (or transformed input space) instead of nonlinear classifiers in input space, which reduces the computational effort.
Fig. 2.10 and 2.11 show another example of KPCA for artificial data, the kernel used here was a Gaussian one with $\sigma = 1$.

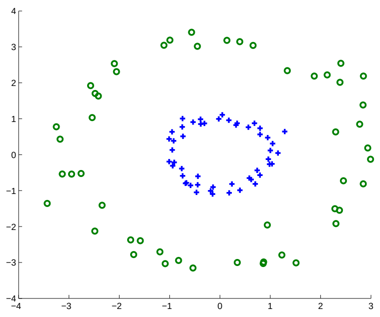


Fig. 2.10.: Data samples of two classes in input space

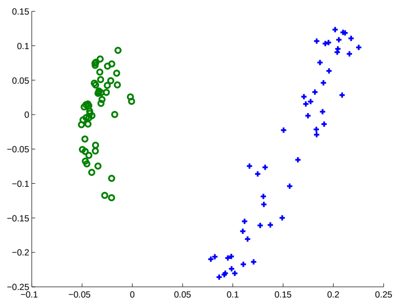


Fig. 2.11.: Data samples of two classes in transformed input space

3. The Symmetric Eigenvalue Problem

If we think about the actual implementation of (K)PCA we will have to solve the symmetric eigenvalue problem (SEVP).

Definition 3.0.2 (Symmetric eigenvalue problem).
Let $Q \in \mathbb{R}^{N \times N}$ be a symmetric matrix, find all eigenvalues λ_i and corresponding eigenvectors $\mathbf{v}_i$.

The SEVP can also be written in one equation

$$\begin{aligned}
(\mathbf{v}_1, ..., \mathbf{v}_p)^T \mathrm{diag}(\lambda_1, ..., \lambda_p) &= Q^T \Lambda && (3.1)\\
&= (\lambda_1 \mathbf{v}_1, ..., \lambda_p \mathbf{v}_p)^T && (3.2)\\
&= \Lambda Q && (3.3)
\end{aligned}$$

The theoretical solution is quite simple, first we need to calculate the zeros of

$$\chi(\lambda) = det(Q - \lambda \mathbb{I}_N) \tag{3.4}$$

which is equivalent to finding the zeros (eigenvalues) of a n-th polynomial function. Yet there exist no algebraic solution for $n > 4$, thus we have to resort to approximation methods. Secondly we have to find the corresponding eigenspaces and their bases (eigenvectors) for each eigenvalue by solving a (possibly underdetermined) system of linear equations. For this thesis two very popular algorithms have been chosen and implemented on GPU as well as on CPU, both of them are iterative methods. Two important aspects arise when considering any of these algorithms, firstly computational complexity (how long will it take to solve the problem regarding the size of the input data) and secondly numerical stability (i.e. how accurate are we regarding a specified/limited machine precision). The computational complexity will be viewed separately for each algorithm when we discuss it in detail. To address the severity of numerical instability one should consider the following example which was encountered during GPU implementation of the QR-algorithm.

Let Q be a symmetric matrix of the form

$$Q = \begin{pmatrix} 1 & 2 & 3 & \dots & N \\ 2 & N+1 & N+2 & \dots & 2N-1 \\ 2 & 3 & 2N & \dots & 3N-3 \\ \vdots & \vdots & \vdots & \dots & \vdots \\ N & 2N-1 & 3N-3 & \dots & NN \end{pmatrix} \tag{3.5}$$

A naive implementation yielded $\lambda_{max,false} \approx 1.462 \cdot 10^6$ while the correct value is $\lambda_{max,correct} \approx 2.708 \cdot 10^8$ for $N = 800$. This miscalculation resulted from calculating

$$PA = (\mathbb{I}_N - \beta \mathbf{v}\mathbf{v}^T)A,\ \mathbb{I}_N = N\text{-dimensional unit matrix} \tag{3.6}$$

instead of

$$PA = A - \mathbf{v}\mathbf{w}^T, \quad \mathbf{w} := \beta A^T \mathbf{v} \tag{3.7}$$

during the implicit QR algorithm. Although both equations are mathematically equivalent, they yield different results due to severe cancellation in Eq. 3.6.
The notation in following sections is borrowed from [GHG96] and basically identical to the matlab language. A matrix element will be addressed by $m(i,j)$, similar for vectors $v(i)$, a complete row of a matrix is addressed via $m(i,:)$ analogously for columns. With respect to a better readability, vectors $\mathbf{x}$ inside a listing will be denoted as x. One important remark regarding indexing: we will start counting from 1! Thus boundaries in the code of the actual implementation in Chap. 6 will be shifted accordingly.

3.1. The Implicit QR Algorithm

This section describes the implicit QR algorithm which basically consists of two other algorithms:

- Tridiagonalization of a matrix (Alg. 2)
- Classic QR algorithm (Alg. 4)

We will briefly discuss the concepts of this algorithm, for a detailed explanation one might refer to [GHG96] or [aHS88]. The following definitions, theorems and algorithms are based on or cited from [GHG96].

Definition 3.1.1. *Let $\mathbf{v} \in \mathbb{R}^N$ be a nonzero vector. The symmetric $N \times N$ Matrix*

$$P_v = \mathbb{I}_N - \beta \mathbf{v}\mathbf{v}^T, \quad \beta := \frac{2}{\mathbf{v}^T\mathbf{v}} \tag{3.8}$$

is called a Householder matrix or Householder reflection, $\mathbf{v}$ is called the corresponding Householder vector and β the corresponding Householder coefficient

A Householder matrix calculated from a vector $\mathbf{v}$ has the ability to mirror $\mathbf{v}$, i.e., $P_v\mathbf{v} = -\mathbf{v}$ thus it is called a reflection matrix. For a given $\mathbf{x} \in \mathbb{R}^N$ with $\mathbf{x} \neq 0$ it is possible to calculate a Householder vector $\mathbf{v} = \mathbf{x} \pm ||\mathbf{x}||_2\mathbf{e}^1$ such that $P_v\mathbf{x} = \mp||\mathbf{x}||_2\mathbf{e}^1$. An example for this would be $\mathbf{x} = (3,1,5,1)^T$ and $\mathbf{v} = (9,1,5,1)^T = \mathbf{x} + 6\mathbf{e}^1$ which yield

$$P_v = \frac{1}{54}\begin{pmatrix} -27 & -9 & -45 & -9 \\ -9 & 53 & -5 & -1 \\ -45 & -5 & 29 & -5 \\ -9 & -1 & -5 & 53 \end{pmatrix}, \quad P_v\mathbf{x} = \begin{pmatrix} -6 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{3.9}$$

Algorithm 1 calculates $\mathbf{v}$ for a given vector $\mathbf{x}$. Additionally this algorithm normalizes $\mathbf{v}$ such that $v_1 = 1$. This reduces the needed memory to store $\mathbf{v}$ by one unit (e.g. one double value). The following algorithms use this fact to to save up $\frac{1}{2}N(N-1)$ units in the representation of the end result.

Algorithm 1 $[v,\beta]$ =**house**(A)

Input: A real vector $x \in \mathbb{R}^N$

```
1: $\sigma = x(2:N)^T x(2:N)$
2: $v = \begin{pmatrix} 1 \\ x(2:N) \end{pmatrix}$
3: if $\sigma = 0$ then
4:     $\beta = 0$
5: else
6:     $\mu = \sqrt{x(1)^2 + \sigma}$
7:     if $x(1) \leq 0$ then
8:         $v(1) = x(1) - \mu$
9:     else
10:        $v(1) = -\frac{\sigma}{x(1) + \mu}$
11:    end if
12:    $\beta = 2\frac{v(1)^2}{\sigma + v(1)^2}$
13:    $v = \frac{v}{v(1)}$
14: end if
```

The application of a Householder matrix to a set of vectors $\mathbf{x}^i$, $i = 1,...,N$, i.e., calculating P_vA where the columns of A consist of $\mathbf{x}_i$ must be done carefully due to the possibility of numerical cancellation. It holds that

$$P_vA = \underbrace{(\mathbb{I} - \beta\mathbf{v}\mathbf{v}^T)A}_{*_1} = \underbrace{A - \mathbf{v}\mathbf{w}^T}_{*_2}, \quad \mathbf{w} := \beta A^T\mathbf{v} \tag{3.10}$$

Yet the first method, i.e., $*_1$, suffers from severe cancellation for bad conditioned matrices whereas the second method, i.e., $*_2$, provides a numerical stable computation.
To enhance the execution speed of the QR method one can simplify the structure of the given problem. Although the QR algorithm has not been presented yet it should be mentioned that the execution speed increases with the number of zero elements in the given matrix A.

Definition 3.1.2. *A matrix $T \in \mathbb{R}^{N\times N}$ of the form*

$$\begin{pmatrix} d_1 & u_1 & 0 & 0 & \dots & 0 \\ l_1 & d_2 & u_2 & 0 & \dots & 0 \\ 0 & l_2 & d_3 & u_3 & \dots & 0 \\ 0 & 0 & l_3 & d_4 & \dots & 0 \\ & & \vdots & & & \\ 0 & 0 & 0 & 0 & \dots & d_N \end{pmatrix} \tag{3.11}$$

is called a tridiagonal matrix .

The following theorem gives rise to an algorithm which tridiagonalizes an arbitrary symmetric matrix.

Theorem 3.1.1. *Let A be a symmetric matrix $A \in \mathbb{R}^{N\times N}$. Then A can be tridiagonalized by applying Householder transformations Q_i, $i = 1, ..., N-2$*

$$(Q_1Q_2...Q_{N-2})^T A(Q_1Q_2...Q_{N-2}) = T \tag{3.12}$$

The resulting matrix T is symmetric and tridiagonal. Additionally it holds that

$$Q_i...Q_{N-2} = \begin{pmatrix} \mathbb{I}_i & 0 \\ 0 & \tilde{Q}_i \end{pmatrix} \tag{3.13}$$

where $\tilde{Q}_i$ is a $(N-i)\times(N-i)$ Householder matrix.

As stated in the theorem, exactly $N-2$ Householder transformations are applied. Each Householder matrix can be represented by a single vector $\mathbf{v}$. The product of these matrices also has a specific structure which can be exploited to represent each matrix Q_i by a $(N-2-i)$-dimensional vector (one should be reminded that the first element of a Householder vector is always 1 and thus needs not to be saved).

Algorithm 2 Tridiagonalize(A)

Input: A symmetric Matrix $A \in \mathbb{R}^{N \times N}$

```
1: for $k = 1$ to $N - 2$ do
2:     $[v,\beta]$ = house($A(k+1:n,k)$)
3:     $p = \beta A(k+1:n, k+1:n)v$
4:     $w = p - (\frac{1}{2}\beta p^T v)v$
5:     $A(k+1,k) = \|A(k+1:n,k)\|_2$
6:     $A(k,k+1) = A(k+1,k)$
7:     $A(k+1:n,k+1:n) = A(k+1:n,k+1:n) - vw^T - wv^T$
8: end for
```

Alg. 2 stores the Householder vectors in the subdiagonal part of the tridiagonal output matrix T. This approach saves half the space one would need if every Householder vector would be stored separately.

For initialization the final version of the QR algorithm demands one single tridiagonalization matrix Q (i.e. $QA = T$). Yet the algorithms above never calculated such a representation for the applied Householder matrices, in theory the desired matrix Q can be calculated from all single Householder vectors which are stored in the output of Alg. 2. One simply has to calculate

$$Q := Q_1 Q_2 ... Q_N \tag{3.14}$$

However, this approach yields a complexity of $\mathcal{O}(N^4)$, thus it is not very useful for a practical implementation. Therefore Alg. 3 should be considered.

Algorithm 3 Backward_Accumulate(A, β)

Input: Output matrix $A \in \mathbb{R}^{N \times N}$ and coefficient vector β from Tridiagonalize(A)

```
1: $Q := \mathbb{I}_N$, $\mathbf{v} := 0$
2: for $j = N$ to $1$ do
3:     $\mathbf{v}(j:N) = \begin{pmatrix} 1 \\ A(j+1:N,j) \end{pmatrix}$
4:     $Q(j:N,j:N) = (\mathbb{I}_{N-j} - \beta_j \mathbf{v}(j:N)\mathbf{v}(j:N)^T)Q(j:N,j:N)$
5: end for
```

This algorithm offers a complexity of $\mathcal{O}(N^3)$ and thus is a more realistic approach to compute Q.
To get a basic understanding of the operations within the QR algorithm we define the QR decomposition

Definition 3.1.3. *Let $A \in \mathbb{R}^{N \times N}$ be an arbitrary matrix. A can be decomposed into*

$$A = QR \tag{3.15}$$

where Q is an orthogonal matrix and R is an upper-right matrix.

All the previous algorithms have been exact solutions to the given problems. The QR algorithm is an approximation method that iteratively creates a matrix $D \in \mathbb{R}^{N \times N}$ which, in the end, holds all desired eigenvalues on its diagonal. Most algorithms regarding the SEVP are unable to efficiently compute the corresponding eigenvectors. The computation of eigenvectors often involves the solving of a system of linear equations. The QR method inherently allows the computation of the corresponding eigenvectors. We will now give the most basic and abstract version of the QR algorithm and enhance it with the concepts we have defined before.

Algorithm 4 Classic_QR(A)

Input: Real matrix $A \in \mathbb{R}^{NxN}$

```
1: $k := 1$
2: repeat
3:     $Z_k := AQ_{k-1}$
4:     $Q_k R_k = Z_k$ (QR factorization of $Z_k$)
5: until result is accurate enough
```

Algorithm 4 has a complexity of $\mathcal{O}(N^4)$, additionally its convergence rate is at most linear. Therefore this algorithm is not suitable for practical considerations. The QR factorization ($\mathcal{O}(N^3)$) represents the major performance brake in this approach. To overcome this obstacle we introduce the concept of Givens rotations. A Givens rotation $G(j,k)$ is a rotation matrix which allows to zero the element (k,j) in an arbitrary matrix A.
Givens rotation will play a fundamental role in Sect. 3.2, where we discuss them in greater detail. Regarding the QR factorization it is sufficient just to note that for tridiagonal matrices this decomposition can be done in $\mathcal{O}(N^2)$ using Givens rotations. The standalone QR factorization has not been implemented in this thesis. The implemented algorithms will be explained in Chap. 6. To conclude this section we take a look at the final form of the QR-algorithm which has been used in this thesis. This algorithm needs about $9N^3$ flops to solve the SEVP and uses a subroutine 'QR_Sub(D, Q, q)' (see [GHG96] Alg. 8.3.3).

Algorithm 5 QR(A)

Input: Real matrix $A \in \mathbb{R}^{N \times N}$

```
1:  compute tridiagonalization of $A$
2:  $T := (Q_1 Q_2 ... Q_{N-2})^T A (Q_1 Q_2 ... Q_{N-2})$
3:  $D := T$
4:  $Q := (Q_1 Q_2 ... Q_{N-2})$
5:  $q := 0$
6:  repeat
7:      for $i = 1$ to $N - 1$ do
8:          if $|D(i+1,i) = D(i,i+1)|$ & $|D(i+1,i)| \le tol \cdot (|D(i,i)| + |D(i+1,i+1)|)$
            then
9:              $D(i+1,i) := 0$    set values to zero if below threshold $tol$
10:             $D(i,i+1) := 0$
11:         end if
12:     end for
13:     QR_Sub($D, Q, q$);
14: until q=n
```

The choice of the termination value *tol* will be discussed in Chap. 6. The subroutine performs the actual QR decomposition using Givens rotations and several other subroutines. Because of the complexity of this method and the fact that it was not implemented in this thesis we will not analyze it any further. Very detailed information regarding this routine can be found in [GHG96] and [aHS88].

3.2. The Two-Sided Jacobi Method

The two-sided Jacobi method (TSJM) represents an algorithm which is inherently parallel and works by applying orthogonal transformation matrices Q onto a given matrix A in the following form:

$$A_1 = Q^T A Q \tag{3.16}$$

The orthogonal transformations are applied on both sides of A. In [JD92] it has been shown that the Jacobi methods in general yield a higher numerical accuracy than QR methods. Although there exist one-sided versions of this algorithm, for this thesis the two-sided approach has been chosen due to higher numerical accuracy for a given amount of iterations ([JD92]).
The goal of TSJM is to reduce the following quantity

$$\mathrm{off}(A) := \sqrt{\sum_{i=1}^{N} \sum_{j=1, j \neq i}^{N} a_{i,j}^2} \tag{3.17}$$

i.e., the 'norm' of the the off-diagonal elements. The reduction is done through so-called Givens rotations.

Definition 3.2.1. *A matrix $G(i,k,\theta) \in \mathbb{R}^{N\times N}$, $i,k \in \mathbb{N}$, $\theta \in \mathbb{R}$ is called a Givens rotation if it satisfies the form*

$$G(i,k,\theta) = \begin{pmatrix} 1 & \dots & 0 & \dots & 0 & \dots & 0 & \\ \vdots & \ddots & \vdots & & \vdots & & \vdots & \\ 0 & \dots & c & \dots & s & \dots & 0 & \leftarrow i \\ \vdots & & \vdots & \ddots & \vdots & & \vdots & \\ 0 & \dots & -s & \dots & c & \dots & 0 & \leftarrow k \\ \vdots & & \vdots & & \vdots & \ddots & \vdots & \\ 0 & \dots & 0 & \dots & 0 & \dots & 1 & \\ & & \uparrow & & \uparrow & & & \\ & & i & & k & & & \end{pmatrix} \tag{3.18}$$

with $c := \cos(\theta), s := \sin(\theta)$

A Givens rotation can be completely represented by the tuples (i,k,θ) or (i,k,c,s). Algorithm 6 computes a Givens rotation $G(i,k,\theta)$ for $1 \le i < k \le N$ with the property that $b_{i,k} = b_{k,i} = 0$ with $B := G(i,k,\theta)^T A G(i,k,\theta)$. For a given input of (A,i,k) the output is a Givens rotation represented by (c,s). The re-

Algorithm 6 Givens(A,i,k)

Input: Real symmetric matrix $A \in \mathbb{R}^{N\times N}$

```
1: if $A(i,k) \neq 0$ then
2:     $\tau := \frac{(A(k,k) - A(i,i))}{2A(i,k)}$
3:     if $\tau \geq 0$ then
4:         $t := \frac{1}{\tau + \sqrt{1+\tau^2}}$
5:     else
6:         $t := -\frac{1}{-\tau + \sqrt{1+\tau^2}}$
7:     end if
8:     $c := \frac{1}{\sqrt{1+t^2}}$
9:     $s := tc$
10: else
11:     $c := 1$
12:     $s := 0$
13: end if
```

sulting matrix enables one to zero out specific elements of A. This leads to the classic Jacobi method which is described by Alg. 7. The parameter *tol* represents the termination criteria and determines the accuracy of the approximated eigenvalues and eigenvectors. We will discuss its selection in Chap. 6. The clas-

Algorithm 7 ClassicJacobi(A, tol)

```
Input: Real symmetric matrix $A \in \mathbb{R}^{N\times N}$
1: $V := \mathbb{I}_N$
2: $eps := tol \cdot ||A||_F$
3: while off($A$) > $eps$ do
4:     choose $(i,k)$ so that $|a_{i,k}| = \max_{p\neq q} |a_{p,q}|$
5:     $(c,s) :=$ Givens$(A,i,k)$
6:     $A := G(i,k,\theta)^T A G(i,k,\theta)$
7:     $V := VG(i,k,\theta)$
8: end while
```

sic Jacobi algorithm basically thins A out until it reaches nearly diagonal form. The off-diagonal elements will be regarded as zeros while the diagonal elements represent the desired eigenvalues. Additionally we get an approximation V of the corresponding eigenvectors.
Yet the search for i, k has not been specified, a naive approach would be to check every element of A against all others which results in a complexity of $\mathcal{O}(N^2)$. This problem will be handled by the cyclic-row extension of the classic Jacobi algorithm. But before that we take a look on both matrix updates. These updates do not involve complete matrix multiplications, a Givens rotation can be applied in $6N$ flops. Analyzing the form of a Givens rotation one can see that it only affects two rows or two columns regarding $G(i,k,\theta)^T A$ or $AG(i,k,\theta)$, respectively.

$$A_1 = G(i,k,\theta)^T A \Leftrightarrow \begin{array}{l} A_1(:,:) = A(:,:) \\ A_1((i,k),:) = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^T A((i,k),:) \end{array} \tag{3.19}$$

$$A_1 = AG(i,k,\theta) \Leftrightarrow \begin{array}{l} A_1(:,:) = A(:,:) \\ A_1(:,(i,k)) = A(:,(i,k)) \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \end{array} \tag{3.20}$$

[GHG96] shows that the convergence rate is linear. Additionally [GHG96] shows that it can reach quadratic convergence after enough iterations. The amount of N while-iterations in Alg. 7 is called a (Jacobi) sweep.
The main problem with Alg. 7 is the search-complexity for appropriate indices i, k. One solution to this problem is the so-called cyclic-row extension where the symmetry of A and the structure of Givens rotations is exploited. Instead of finding the maximal value of A, one can try to zero out the upper diagonal part

(which also affects the lower diagonal part in the same way). Thus one can cycle the elements row by row, e.g., for $N = 4$

$$(i,k) = (1,2),(1,3),(1,4),(2,3),(2,4),(3,4),(1,2),... \tag{3.21}$$

until the terminations criteria is satisfied. [GHG96] shows that this method reduces $\text{off}(A)$ with every iteration. The complete extension is described by Alg. 8 and will be further extended to the final form of the Jacobi algorithm used in this thesis. As mentioned before, Jacobi methods are inherently parallel, yet

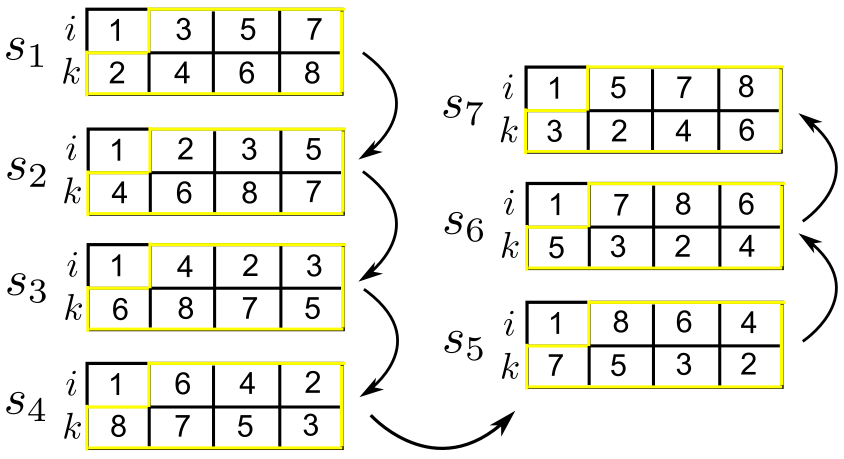
Algorithm 8 CyclicRowJacobi(A, tol)

```
Input: Real symmetric matrix $A \in \mathbb{R}^{N\times N}$
 1: $V := \mathbb{I}_N$
 2: $eps := \text{tol}||A||_F$
 3: while $\text{off}(A) > eps$ do
 4:    for $i = 1$ to $N-1$ do
 5:       for $k = i+1$ to $N$ do
 6:          $(c,s) := \text{Givens}(A,i,k)$
 7:          $A := G(i,k,\theta)^T A G(i,k,\theta)$
 8:          $V := VG(i,k,\theta)$
 9:       end for
10:    end for
11: end while
```

the listed algorithms do not exploit this parallelism. Let us take a look on a single sweep for $N = 4$. From now on we will refer to each choice of (i,k) as a subproblem. If we partition all subproblems in a single sweep into 3 sets

$$s_1 = \{(1,2),(3,4)\} \tag{3.22}$$
$$s_2 = \{(1,3),(2,4)\} \tag{3.23}$$
$$s_3 = \{(1,4),(2,3)\} \tag{3.24}$$

we can see that all subproblems within each of these sets can be executed in parallel, e.g., we can update A simultaneously for $(i,k) = (1,2)$ and $(i,k) = (3,4)$ (the same holds for V). Subproblems which can be executed simultaneously are called independent subproblems. From now on we assume that N is even (the case where N is odd will be handled later).
It is possible to create such a partition for all values of N. The procedure is visualized in Fig. 3.1 for $N = 8$. We start with s_1, s_2 is created by shifting all values except 1 clockwise, this procedure is repeated 5 times. In the general case this procedure yields a total of $N-1$ sets each containing $\frac{N}{2}$ pairs (i,k) of indices.



s_1 i 1 3 5 7 k 2 4 6 8
s_2 i 1 2 3 5 k 4 6 8 7
s_3 i 1 4 2 3 k 6 8 7 5
s_4 i 1 6 4 2 k 8 7 5 3
s_5 i 1 8 6 4 k 7 5 3 2
s_6 i 1 7 8 6 k 5 3 2 4
s_7 i 1 5 7 8 k 3 2 4 6

Fig. 3.1.: construction of subsets containing independent subproblems

We now define the complete version of the two-sided Jacobi algorithm, as it has been implemented.

Algorithm 9 TwoSidedJacobi(A, tol)

Input: Real symmetric matrix $A \in \mathbb{R}^{N \times N}$

```
1: $V := \mathbb{I}_N$
2: $eps := \text{tol}||A||_F$
3: while $\text{off}(A) > eps$ do
4:   for $set = 1$ to $N - 1$ do
5:     for $sub = 1$ to $\frac{N}{2}$ do
6:       $(i,k) := PROBLEM\_LUT(set, sub)$
7:       $(c,s) := \text{Givens}(A,i,k)$
8:       $A := G(i,k,\theta)^T A G(i,k,\theta)$
9:       $V := VG(i,k,\theta)$
10:    end for
11:  end for
12: end while
```

Alg. 9 utilizes a lookup table '$PROBLEM\_LUT$' to determine the needed indices, the creation of this table and its structure will be explained in detail in Chap. 6. The inner for-loop can be executed in parallel as all subproblems in each set are independent, this fact will used in a later chapter. So far we have assumed that N is an even number, this assumption is very convenient for the algorithms above, yet it represents a practical restriction for them. The case where N is odd can be handled by adding a new zero filled row and a zero filled column to the given matrix A.

4. Classification

So far we have only explained coordinate transformations of the input space, which yield a more feasible representation in the context of statistics. Yet our goal is the classification of unknown traces. For this task we will use three popular concepts and corresponding algorithms. These concepts are those of nearest neighbour classification (KNN), linear discriminant analysis (LDA) and support vector machines (SVM). The KNN and LDA will be explained in their non-kernelized version, i.e., their formulation in terms of input space. As [aAFF05] points out, the kernelized version of both algorithms can be formulated in terms of a transformed input space.

4.1. Kernel k-nearest Neighbors Classification

The k-nearest neighbors approach to classification is one of the most simple solutions to this task, as it makes no assumptions about underlying distributions. The following derivation is based on [Bis06]. Let us suppose we are given N_i data points of class $\mathcal{C}_i$ for $i = 1, .., K$ and have to construct a classifier $f : I \to \{1, ..., K\}$. For a given point $\mathbf{x}$ one could draw a sphere around it in way that it contains k other points (disregarding their class membership) of the given data samples and exactly k_i data points of the class which $\mathbf{x}$ belongs to. The conditional density of each class can be estimated with

$$p(\mathbf{x}|\mathcal{C}_i) = \frac{k_i}{N_i V} \tag{4.1}$$

where V denotes the volume of the sphere. The unconditional density takes the form

$$p(\mathbf{x}) = \frac{K}{NV}, \ N = \sum_{i=1}^{K} N_i \tag{4.2}$$

with prior estimations

$$p(\mathcal{C}_i) = \frac{N_i}{N} \tag{4.3}$$

Combining these equations one can formulate the posterior probabilities

$$p(\mathcal{C}_i|\mathbf{x}) = \frac{k_i}{k} \tag{4.4}$$

Minimizing the risk of misclassification can be done by simply assigning $\mathbf{x}$ to the class which minimizes the posterior probability. In other words, $\mathbf{x}$ will be assigned to the class with the majority of points inside the sphere (as depicted in Fig. 4.1). In algorithmic terms, this corresponds to analyzing the k nearest samples to $\mathbf{x}$ and assigning $\mathbf{x}$ to the class with the most samples under those k neighbors.

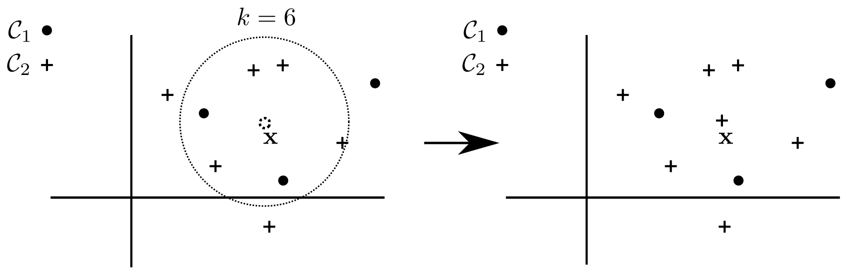


Fig. 4.1.: KNN algorithm for $k = 6$

4.2. Kernel Linear (Fisher) Discriminant Analysis

In linear discriminant analysis one tries to find a so called discriminant function of following form

$$f : I \to \mathbb{R},\ f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0 \tag{4.5}$$

The vector $\mathbf{w}$ is called a weight vector and the number w_0 is the bias. For now, let us assume that we only have to distinguish between two classes $\mathcal{C}_1$ and $\mathcal{C}_2$. Furthermore we assume that $f(\mathbf{x}) \geq 0$ if $f(\mathbf{x})$ belongs to $\mathcal{C}_1$ and $f(\mathbf{x}) < 0$ if $\mathbf{x}$ lies in $\mathcal{C}_2$. This discriminant is visualized in Fig. 4.2, one can see that the decision boundary between the two classes is nothing more than a $(N-1)$-dimensional hyperplane $H = \{\mathbf{x}|f(\mathbf{x}) = 0\}$ (hessian normal form). It is not always possible to

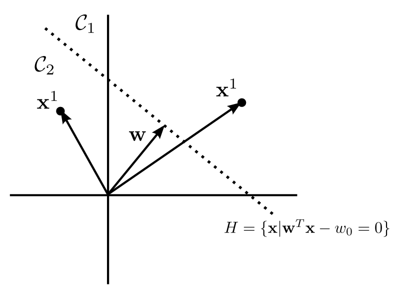


Fig. 4.2.: Example of separating hyperplane in input space

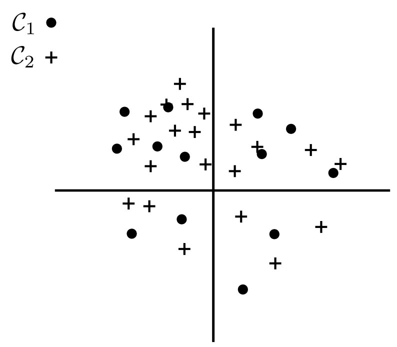


Fig. 4.3.: Non-separable data set

create a perfect class separation. This case is visualized in Fig. 4.3. Additionally Figs. 4.4 and 4.5 show the resulting overlap of the projected values, i.e., the projection of each data point onto the weight vector (exemplified by W). We will refer to this overlap as class overlap. But how do we choose the right weight vector and bias to get the best possible class separation. Before we address the problem of finding these parameters, we shall ask the question what characterizes the separability of two given data sets. If we take a look at Fig. 4.4 it is obvious that a small variance within each dataset and a large variance between both data sets yields a smaller class overlap. Thus one should take these properties into account when looking for optimal values $\mathbf{w}$ and w_0.
This strategy is applied in a specific variant of LDA called Fisher LDA (FLDA), where it is proposed to carry out the optimization in the hyperplane H with fixed $w_0 = 0$. First a few words about notation. Let $\mathbf{m}^1, \mathbf{m}^2 \in \mathbb{R}^N$ be the class means for $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively. Furthermore we have $\mathbf{x}^\mu \in \mathcal{C}_1,\ \mathbf{y}^\tau \in \mathcal{C}_2,\ \mu \in \{1, ..., M\},\ \tau \in \{1, ..., P\}$. To maximize the global variance one has to maximize the value

$$(\frac{\mathbf{w}^T}{||\mathbf{w}||_2}(\mathbf{m}^2 - \mathbf{m}^1))^2 = (m_2 - m_1)^2 \tag{4.6}$$

For mathematical convenience the weight vector shall be normed. In other words, the global variance will be maximized for the projected points x_μ and y_τ. To minimize the within-class variance, the following value has to be minimized.

$$\begin{aligned} \sigma_1^2 + \sigma_2^2 &= \sum_{i=1}^{M}(x_i - m_1)^2 + \sum_{i=1}^{P}(y_i - m_2)^2 && (4.7) \\ &= \sum_{i=1}^{M}((\frac{\mathbf{w}^T}{||\mathbf{w}||_2}\mathbf{x}^i - w_0) - m_1)^2 + \sum_{i=1}^{P}((\frac{\mathbf{w}^T}{||\mathbf{w}||_2}\mathbf{y}^i - w_0) - m_2)^2 && (4.8) \end{aligned}$$

Both problems can be formulated in one equation

$$\max_{\mathbf{w}} \frac{(m_2 - m_1)^2}{\sigma_1^2 + \sigma_2^2} = \max_{\mathbf{w}} \frac{\mathbf{w}^T S_b \mathbf{w}}{\mathbf{w}^T S_w \mathbf{w}} \tag{4.9}$$

where

$$S_b = (\mathbf{m}^2 - \mathbf{m}^1)(\mathbf{m}^2 - \mathbf{m}^1)^T \tag{4.10}$$

and

$$S_w = \sum_{i=1}^{M}(\mathbf{x}^i - \mathbf{m}^1)(\mathbf{x}^i - \mathbf{m}^1)^T + \sum_{i=1}^{P}(\mathbf{y}^i - \mathbf{m}^2)(\mathbf{y}^i - \mathbf{m}^2)^T \tag{4.11}$$

are symmetric positive definite matrices. As noted in [Bis06] the solution $\mathbf{w}$ is proportional to $S_w^{-1}(\mathbf{m}_2 - \mathbf{m}_1)$, i.e.,

$$\mathbf{w} \propto S_w^{-1}(\mathbf{m}^2 - \mathbf{m}^1) \tag{4.12}$$

Thus only a matrix inversion (in praxis, the solution of a linear equation system with LR/Cholesky decomposition) is needed to compute the direction of $\mathbf{w}$.

[Wis10] gives an alternative derivation using a diagonalized average covariance matrix, yet this approach is computationally too expensive.
We now have determined the optimal direction for the weight vector with fixed bias $w_0 = 0$, in general this bias does not have to be the optimal value. To construct an appropriate bias [Bis06] suggests the use of class-conditional densities

$$p(f(.)|\mathcal{C}_i) \tag{4.13}$$

while [Kuh06] proposes

$$w_0 = \frac{\mathbf{w}(\mathbf{m}^2 - \mathbf{m}^1)}{2} \tag{4.14}$$

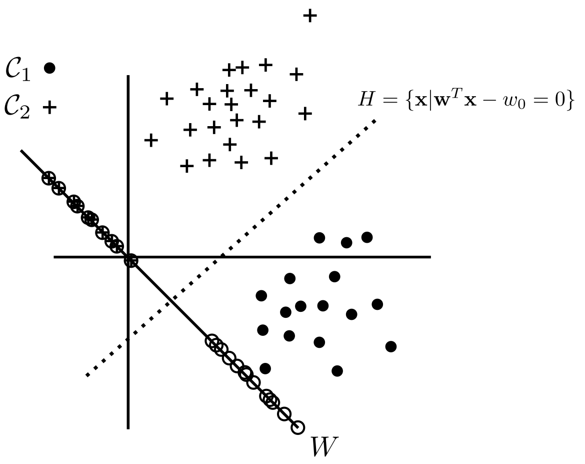


Fig. 4.4.: Projected data points in case of perfect separable classes

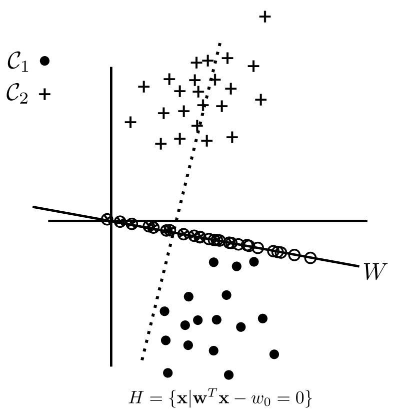


Fig. 4.5.: Class overlap in the case of non-optimal parameters

For better accuracy we will use the first proposition. The derivation of following bias can be found in [Bis06], [Li08] and [Ige10]

$$w_0 = \log\frac{\pi_1}{\pi_2} - \frac{1}{2}(\mathbf{m}^1 + \mathbf{m}^2)^T\Sigma^{-1}(\mathbf{m}^1 - \mathbf{m}^2) \tag{4.15}$$

Where π_1, π_2 represent the prior probabilities (i.e. $\pi_i = p(\mathcal{C}_i)$) for class $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively. Σ is the covariance matrix of both classes, i.e., we assume that both data sets have an underlying normal distribution with an indentical covariance matrices. The prior probabilities can be approximated with

$$\pi_i = \frac{\#\text{ of samples in class } i}{\text{total }\#\text{ of samples}} \tag{4.16}$$

Approximating the gaussian distribution can be done via

$$\Sigma = \frac{1}{M + P - 2}\sum_{i=1}^{2}\sum_{\mathbf{x}\in\mathcal{C}_i}(\mathbf{x} - \mathbf{m}^i)(\mathbf{x} - \mathbf{m}^i)^T \tag{4.17}$$

So far we only considered two classes of data points, in praxis one often has to classify with more than two classes. As mentioned in [Bis06] a feasible way to classify with K classes is the construction of multiple discriminants f_k. Which yield a measure of class membership. Classification can be done by assigning $\mathbf{x}$ to class j through

$$j = \arg\max_k f_k(\mathbf{x}) \tag{4.18}$$

The above derivation for a two class problem can be generalized for K classes. Yet at this point, only the final derived discriminants $f_k,\ k \in \{1, ..., K\}$ should be given. The reader interested in the complete proof should refer to [Bis06], [Li08] and [Ige10].

$$f_k(\mathbf{x}) = \mathbf{x}^T\Sigma^{-1}\mathbf{m}^k - \frac{1}{2}\mathbf{m}^{k^T}\Sigma^{-1}\mathbf{m}^k + \log\pi_k \tag{4.19}$$

with estimations

$$\Sigma = \frac{1}{l-K}\sum_{k=1}^{K}\sum_{\mathbf{x}\in\mathcal{C}_k}(\mathbf{x}-\mathbf{m}^k)(\mathbf{x}-\mathbf{m}^k)^T,\ l = \sum_{k=1}^{K}\#\text{ samples in class } k \tag{4.20}$$

In this thesis, the kernelized version of LDA consists of applying LDA to the KPCA-transformed space $\mathbb{R}^{M\times M}$ (as motivated and researched by [aAFF05]). As mentioned above, the KNN classifier does not take probability distributions into account. Thus the above made assumption of normal distributed data sets will proof wrong, if KLDA produces worse results than KKNN.

4.3. Support Vector Machines

In this section a different approach to classification will be presented. So far we have used KPCA as a tool to enhance standard classification methods (i.e. KNN and LDA). We have reduced or increased the dimensionality of the space in which we carried out the classification. We will now present a way of performing linear classification entirely in the feature space, i.e., without changing the dimensionality or approaching classification in a transformed input space. From now on we again assume that only two classes $\mathcal{C}_1, \mathcal{C}_2$ have to be distinguished. The technique described next, consists of separating data points in the feature space $\mathcal{H}$ by a hyperplane.
Let

$$H = \{\mathbf{x} \mid\ <\mathbf{w}, \mathbf{x}>_{\mathcal{H}} + b = 0\} \tag{4.21}$$

be a hyperplane in $\mathcal{H}$, with weight vector $\mathbf{w} \in \mathcal{H}$ and bias $b \in \mathbb{R}$. We will classify according to decision functions

$$f(\mathbf{x}) = sgn(<\mathbf{w}, \mathbf{x}>_{\mathcal{H}} + b) \tag{4.22}$$

Let $\{(\mathbf{x}^i, y_i)\}$ be a family of M data points with corresponding class labels $y_i \in \{-1, 1\}$, which we will refer to as training points. To find the best separating hyperplane we assume that the training points are linearly separable, i.e., a separating hyperplane exists. The not linearly separable case will be handled later. The optimal hyperplane maximizes the margin (i.e. the shortest distance of a point to the hyperplane) of each training point. In mathematical terms

$$\max_{\mathbf{w}\in\mathcal{H}, b\in\mathbb{R}} \min\{||\mathbf{x} - \mathbf{x}^i||_{\mathcal{H}} \mid \mathbf{x} \in \mathcal{H}, \ <\mathbf{w}, \ \mathbf{x}>_{\mathcal{H}} + b = 0, \ i = 1, ..., M\} \tag{4.23}$$

This concept is visualized in Fig. 4.6. In the following derivation (which bases on [SS02] and [Bis06]) we require $|<\mathbf{w}, \mathbf{x}^i>_{\mathcal{H}} + b| = 1$ for the closest points to H. With this, the minimal margin becomes $1/||\mathbf{w}||_{\mathcal{H}}$. This can be seen by considering two points, e.g., $\mathbf{x}^1, \mathbf{x}^2$ with minimal distance to H

$$\begin{aligned} <\mathbf{w}, \mathbf{x}^1>_{\mathcal{H}} &= 1 && (4.24)\\ <\mathbf{w}, \mathbf{x}^2>_{\mathcal{H}} &= -1 && (4.25)\\ \Rightarrow \ <\mathbf{w}, (\mathbf{x}^1 - \mathbf{x}^2)>_{\mathcal{H}} &= 2 && (4.26)\\ \Rightarrow \ <\frac{\mathbf{w}}{||\mathbf{w}||_{\mathcal{H}}}, (\mathbf{x}^1 - \mathbf{x}^2)>_{\mathcal{H}} &= \frac{2}{||\mathbf{w}||_{\mathcal{H}}} && (4.27) \end{aligned}$$

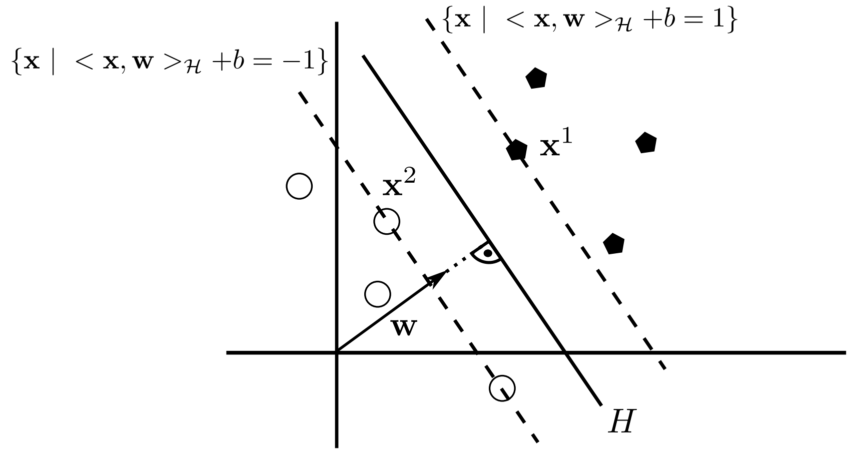

$\{\mathbf{x} \mid \ <\mathbf{x}, \mathbf{w}>_{\mathcal{H}} + b = 1\}$
$\{\mathbf{x} \mid \ <\mathbf{x}, \mathbf{w}>_{\mathcal{H}} + b = -1\}$
$\mathbf{x}^1$
$\mathbf{x}^2$
$\mathbf{w}$
H

Fig. 4.6.: Visualization of a hyperplane in feature space

We now define

Definition 4.3.1 (Canonical hyperplane).
Let $T = \{(\mathbf{x}^i, y_i)\}$ be a training set. The pair $(\mathbf{w}, b) \in \mathcal{H} \times \mathbb{R}$ is called a canonical form of a hyperplane H with respect to T if

$$\min_{i=1,..,m} | < \mathbf{w}, \mathbf{x}^i >_{\mathcal{H}} + b| \geq 1 \tag{4.28}$$

and which yields

$$y_i < \mathbf{w}, \mathbf{x}^i >_{\mathcal{H}} + b \geq 1 \tag{4.29}$$

Thus a canonical hyperplane is optimal in the sense that it classifies all training points correctly and has a lower bound $1/||w||_{\mathcal{H}}$ for its margin. Maximizing this lower bound corresponds to minimizing $||w||_{\mathcal{H}}$. For mathematical convenience we formulate this optimization problem as finding $(\mathbf{w}, b)$ with

$$(\mathbf{w}, b) = \arg\min_{\tilde{\mathbf{w}}, \tilde{b}} \frac{1}{2}||\tilde{\mathbf{w}}||^2_{\mathcal{H}} \tag{4.30}$$

$$\text{subject to } y_i < \mathbf{w}, \mathbf{x}^i >_{\mathcal{H}} + b \geq 1 \quad \forall i = 1, ..., M \tag{4.31}$$

This is a typical problem of quadratic programming, its solution can be obtained by the minimization of following Lagrangian

$$L(\mathbf{w}, b, \mathbf{a}) = \frac{1}{2}||\mathbf{w}||^2_{\mathcal{H}} - \sum_{m=1}^{M} a_m(< \mathbf{w}, \Phi(\mathbf{x}) >_{\mathcal{H}} + b) - 1, \quad \mathbf{x} \in I, a_m \geq 0 \tag{4.32}$$

This formulation is called the primal problem. To calculate (i.e. approximate) the solution another formulation is needed, the so-called dual problem . The dual problem arises from the above Lagrangian and requires the maximization of

$$\tilde{L}(\mathbf{a}) = \sum_{m=1}^{M} a_m - \frac{1}{2}\sum_{n=1}^{M}\sum_{m=1}^{M} a_n a_m t_n t_m k(\mathbf{x}^n, \mathbf{x}^m) \tag{4.33}$$

subject to the constraints

$$a_m \geq 0, \quad m = 1, ..., M \tag{4.34}$$

$$\sum_{m=1}^{M} y_m a_m = 0 \tag{4.35}$$

Although this represents again a quadratic programming problem, the amount of variables has been reduced and we obtained a formulation in terms of kernel functions. The training points $\mathbf{x}^m$ with $a_m > 0$ are called support vectors. After solving above the optimization problem, one can formulate a classifier f by

$$f(\mathbf{x}) = sgn(\sum_{m=1}^{M} a_m y_m k(\mathbf{x}, \mathbf{x}^m) + b) \tag{4.36}$$

This kind of classifiers are called support vector machines (SVM). At this point it should be noted that by calculating $\nabla L(\mathbf{w}, b, \mathbf{a})$, it can be shown that the weight vector can be expressed as

$$\mathbf{w} = \sum_{m=1}^{M} \alpha_m y_m \mathbf{x}^m \tag{4.37}$$

Thus the weight vector is a linear combination of the support vectors.
In praxis however, most problems are not linearly separable. This requires a relaxation of the constraints we have made above. To accommodate this fact one can introduce so called slack variables

$$\xi_m \in \mathbb{R},\ m = 1, ..., M \tag{4.38}$$

with following property

$$y_m(< \mathbf{w}, \phi(\mathbf{x}^m) >_{\mathcal{H}} + b) \geq 1 - \xi_m \tag{4.39}$$

SVMs that use these variables are called soft-margin SVMs. The geometric interpretation of the slack variables can be summarized as:

- if $-\infty < \xi^m < 0$, we demand that the sample $\phi(\mathbf{x}^m)$ has a hyperplane margin greater than the minimal distance of 1, i.e., it still has to be classified correctly.
- if $0 < \xi^m < 1$, we allow the sample $\phi(\mathbf{x}^m)$ to have a hyperplane margin smaller than 1. Yet it still has to be classified correctly.
- if $0 < \xi^m \leq 1$, we allow the sample $\phi(\mathbf{x}^m)$ to have a hyperplane margin smaller than 1. Yet it can lie on the hyperplane itself, yielding no clear classification.
- if $0 < \xi^m < \infty$, we allow the sample $\phi(\mathbf{x}^m)$ to be misclassified, i.e., we tolerate an error regarding $\phi(\mathbf{x}^m)$ during training of the SVM.

The introduction of slack variables changes the optimization problem which was described above. One possible way to formulate a new objective function (i.e. a function which has to be minimized to solve an optimization problem) is described in [SS02]. Where the following function has to be minimized

$$f(\mathbf{x}, \xi) = \frac{1}{2}||\mathbf{w}||^2_{\mathcal{H}} + \frac{C}{M} \sum_{m=1}^{M} \xi_m \tag{4.40}$$

The last term in this function describes a trade-off between margin maximization and training error. The constant C regulates this ratio. As in the separable case, the soft-margin problem yields a weight vector which can be expressed as a linear combination of support vectors (see Eq. 4.37). To obtain the coefficients α_m one has to solve

$$\alpha = \arg\min_{\tilde{\alpha}} \sum_{i=m}^{M} \tilde{\alpha}_m - \frac{1}{2} \sum_{m,n=1}^{M} \tilde{\alpha}_i \tilde{\alpha}_j y_i y_j k(\mathbf{x}^i, \mathbf{x}^j) \tag{4.41}$$

subject to the constraints

$$0 \leq \alpha_m \leq \frac{C}{M}, \quad m = 1, ..., M \tag{4.42}$$

$$\sum_{m=1}^{M} y_m a_m = 0 \tag{4.43}$$

The proof for this statement can be found in [SS02] and [Bis06]. One important note should be taken regarding the constant C, which is no variable in the optimization problem. Yet C is a variable in the overall training of a SVM. After determining α one can calculate the hyperplane offset via

$$b = \frac{1}{\# \text{ of SVs}} \left(\sum_{j:\alpha_j > 0} y_j - \sum_{m=1}^{M} y_m \alpha_m k(\mathbf{x}^m, \mathbf{x}^j) \right) \tag{4.44}$$

We will not cover the numerical solution to this problem as it has not been implemented during this thesis. The reader might consider [SS02] and [aCJL11] for implementation details.
A SVM is a binary classifier, using the above formulation it is impossible to distinguish between more than two classes. Yet there exist many approaches to extend the use of SVMs to multi-class problems, i.e., classification tasks with $p \geq 2$ classes. The most simple one is the so called one-vs-one (OvO) approach. Where training data is splitted regarding the classes, before the actual training begins. Let us assume that we are given training samples from p classes. In OvO, the data is segmented into p parts. Each part consists only of data from the corresponding class. Afterwards $p(p-1)/2$ SVMs are trained on all possible pairs of classes, e.g., $(1,2), (3,p), (12,5)$. To classify a given test point $\mathbf{x}^\mu$ each SVM will classify it and make a vote for the correct class. In the end $\mathbf{x}^\mu$ will be assigned the class with the most votes. The case where two or more classes have the same vote can be handled by assigning $\mathbf{x}^\mu$ to the first class encountered. [aCJL11] points out that this method is a competitive approach regarding other solutions like, e.g., one-vs-all. The OvO method has been used in this thesis.

4.4. Grid Searching

Let us assume that we have a method to solve all optimization problems which were mentioned in the last section. We still need to tune several *external* parameters, like multiple kernel parameters or the trade-off constant C. A common method to locate the optimal range of parameters is the so-called *grid search*. During a grid search the whole possible parameter range is discretized into an exponential grid. For example, the range of $C \in \mathbb{R}$ would be $(-\infty, \infty)$ which could be *sampled* via $..., b^{-2}, b^{-1}, b^0, b^1, b^2, ...$ with $b \in \mathbb{N}$. Of course one will not sample

the complete range. Instead a subset of the range will be chosen and discretized, e.g., $[2, 10^4]$ for C. Once a subregion has been sampled one can train a SVM for each sample point and afterwards evaluate the overall classification performance. According to the results afterwards, a more specific subset of the previous subset can be chosen. On which in turn a grid search can be performed. This technique narrows the optimal parameters down.
The training of a SVM involves at least two parameters: at least one kernel parameter and the trade-off constant. Thus the grid search requires a sampling of the euclidean plane (as depicted in Fig. 4.7). The SVMs used in this thesis involve kernel functions with only one parameter.

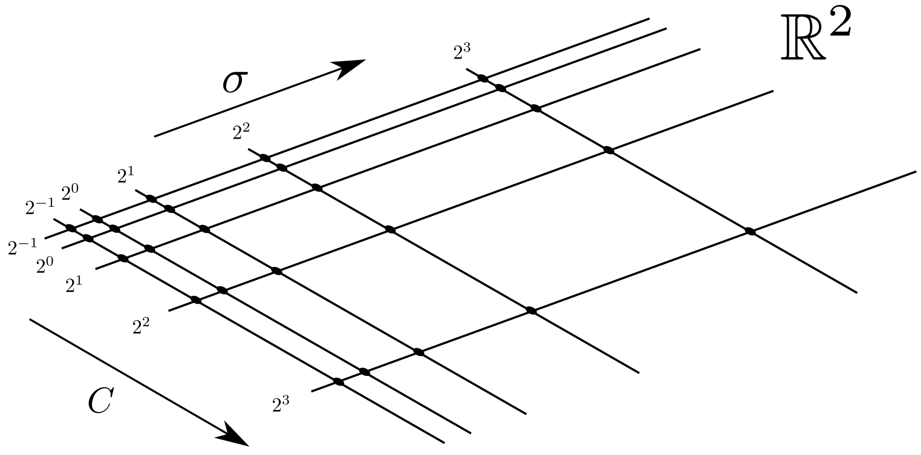


Fig. 4.7.: Sampling $\mathbb{R}^2$ for a local grid search

At this point we can precisely define the goal of this thesis: the construction of a so-called template

Definition 4.4.1 (Template).
Let $\Xi : I \to [p]$ be a classifier working on $\mathfrak{C}$ and T, with $\mathfrak{C} := \{\mathcal{C}_1, ..., \mathcal{C}_p\}$ a set of classes, $T \subset I$ a set of training data. Furthermore it should hold that $\forall x \in T \exists! i \in [p] : x \in \mathcal{C}_i$. The tuple $\mathcal{T} = (\Xi, \mathfrak{C}, T, \Psi)$ is called a template where Ψ denotes the classifier parameters that are independent of $\mathfrak{C}$.

which minimizes

$$AvEr(U) := \frac{1}{|U|} \sum_{\mathbf{x} \in U} \Delta(|\Xi(x) - k_x|),\ U \subset I,\ U \cap T = \varnothing,\ \Delta(x) := \begin{cases} 0 & x = 0 \\ 1 & else \end{cases} \tag{4.45}$$

for an arbitrary choice of U, where k_x denotes the class index of $\mathbf{x}$. Regarding above definition the classifier Ξ must not be of a specific type, e.g., a single SVM. It would be also possible to use a classifier array, consisting of a KLDA, a k-nearest neighbors and a SVM classifier. The set Ψ holds the parameters which

are required to optimize Ξ, e.g., the trade-off constant C within a SVM.
The function $AvEr$ represents the average recognition error, thus $AvEr(U)$ stands for the classification error on a set U of data points. Furthermore we define the actual recognition rate on U.

Definition 4.4.2 (averate recognition rate)**.**

$$Av(U) := 1 - AvEr(U) \tag{4.46}$$

Thus a template attack, in the context of this thesis, involves the construction of a template $\mathcal{T}$, using a set T of power traces and an arbitrary classifier Ξ, which has been trained on T. This classifier has to be fine-tuned with an appropriate choice of Ψ. Enabling it to correctly identify many instruction types from unknown power traces.

5. The CUDA architecture

The last chapters covered the key concepts for this thesis. Most methods described in Chap. 2 and 4 have a complexity of $\mathcal{O}(N^3)$, where N stands for the dimensionality of the input space I. One method to approach this complexity is the use of heuristics or approximation methods (e.g. construction of kernel classifiers by approximating the gram matrix). Yet the focus in this thesis lies on the direct solution of the described problems. Accelerating the algorithms can be done by utilizing parallel computation in terms of symmetric multiprocessor (SMP) architectures. The CUDA architecture provides massively parallel computation power, e.g., as of today up to 448 computation cores per CUDA device. This chapter will provide a very brief introduction into CUDA, including important hardware specifications and implementation aspects.

5.1. A Brief Introduction to CUDA

Modern graphic cards utilize so-called shader units which allow a graphics designer to implement custom computation routines. For example, enhancing the look of a texture by adding special effects after it has been applied to the surface of a polygon. A shader unit can be described as a restricted (in terms of functionality) CPU core. A GPU features a high number of such units, thus they are also called a shader array. NVIDIA cards provide another abstraction layer, called CUDA, to this shader array. CUDA enables the developer to use the GPU nearly in the same way as a SMP system. Yet there are several important restrictions to this access. Let us first take a look onto the CUDA architecture and the developers view onto it.
Fig. 5.1 shows the view onto a CUDA enabled device. The GPU is segmented into so called streaming multiprocessors (SMs), which in turn consist of streaming processors (SPs). A SP is also called a CUDA core. The SP is one atomic computation unit in CUDA, i.e., one processor which executes a program. The GPU used in this thesis is a GTX470 which provides a total of 448 SPs and belongs to NVidias Fermi GPU generation. Those CUDA cores are grouped into 14 SMs, each of them incorporating 32 cores. The developer can not access these cores directly, the execution of a CUDA program will be managed by the GPU itself. The management of program execution is clearly defined and the programmer must use this to his advantage. In most literature (e.g. [aEK10] or [aWmH10])

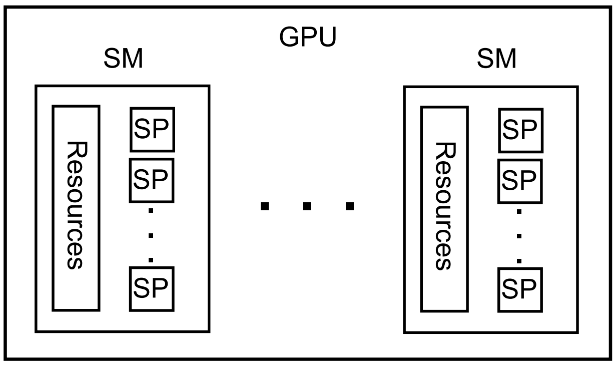


Fig. 5.1.: CUDA architecture of a GPU

this programming model is introduced by an implementation of the well-known matrix multiplication. For reasons of simplicity we will introduce the concepts with an example of computing the sum of two real valued quadratic matrices $A, B \in \mathbb{R}^{n \times n}$. This operation has a complexity of $\mathcal{O}(n^2)$.
The sum of two matrices is defined as

$$(A + B)_{i,j} := A_{i,j} + B_{i,j} \tag{5.1}$$

An ordinary single threaded implementation would have a form like

```
void main()
{
        for(int i=0;i<n;i++)
        {
                for(int j=0;j<n;j++)
                {
                        C[i,j] = A[i,j]+B[i,j];
                }
        }
}
```

One might notice that the instructions inside the inner for-loop are independent of each other, thus they can be executed in parallel. The optimal case would be if we had n^2 cores available, such a system would reduce the complexity to $\mathcal{O}(1)$. Yet for arbitrary matrix sizes, no such system exists. Even though one might not have the required number of cores to execute all instructions simultaneously, the usage of more than one core is still a major benefit, as it reduces computation time by at least half.
The CUDA approach to this algorithm consists of launching n^2 threads, each computing one element of C. At this point thread scheduling becomes relevant. Before we take a look on scheduling we shall explore how threads are ordered and executed in general. CUDA groups threads into three dimensional thread

blocks $T_{i,j}$. Those blocks are in turn ordered in a so-called two dimensional grid G. This concept is visualized in Fig. 5.2. Let us further simplify the situation by

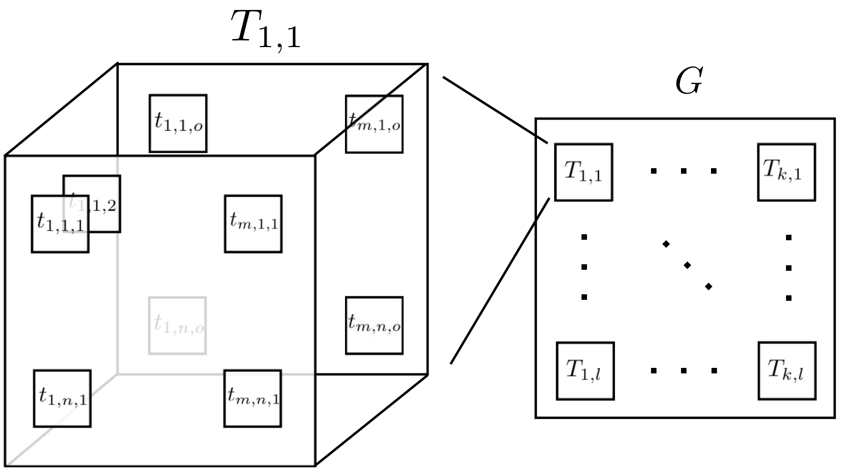


Fig. 5.2.: Thread grouping in CUDA

assuming that each block $T_{i,j}$ is quadratic and its size divides n. The program which will be executed by every thread is called a kernel (which should not be confused with kernel functions). The kernel in our case could have following form, which is similar to, e.g., an OpenMP approach.

```
__global__ kernelForMatrixAddition(double* A, double* B,
    double* C, int blockSize)
{
 //block coordinates
 int blockX = blockIdx.x;
 int blockY = blockIdx.y;
 //thread coordinates inside the block
 int threadX = threadIdx.x;
 int threadY = threadIdx.y;

 //actual indices for memory access
 int i = blockX*blockSize+threadX;
 int j = blockY*blockSize+threadY;

 C[i,j] = A[i,j]+B[i,j];

}
```

Every thread entering the kernel computes its indices i, j and accesses the corresponding matrix elements. A kernel launch is always asynchronous, i.e., once the GPU has received the command to launch a kernel, it returns control to the host

thread. Thus the host must take care of synchronization between GPU and host application segments. GPU threads can communicate with each other in terms of data exchange and synchronization. Yet this is limited to threads within a block. The synchronization of all GPU threads can be done by a method called 'host synchronization' which holds the host thread until all GPU threads have finished execution. This method can be explained with the following example. Let us assume that we want to synchronize all threads on the GPU during the execution of a kernel. To achieve this, we have to split the kernel into two new kernels. The first one will hold all instructions before the synchronization point, whereas the second one holds all instructions after the synchronization point. The first kernel will be launched and the host thread will wait until the GPU has finished computation. In other words, until all GPU threads have reached the end of the first kernel. Afterwards the host will launch the second kernel, thus we achieved the desired synchronization of all GPU threads.
Only a specific (GPU dependent) number of blocks can be assigned to a SM, e.g., up to 8 on a GTX470. It should be noted that this number depends on the total count of threads residing in these blocks, e.g., a GTX470 can assign up to 1536 threads to a single SM. Each SM also has a limited number of internal resources, which could further reduce the maximal number of assignable threads during runtime. Each SM divides its blocks into so called warps of threads. This segmentation of blocks into warps is visualized in the Figs. 5.3 to 5.5 for a warp size of 32 threads within quadratic blocks. For reasons of simplicity these figures do not take into account GPU specific limitations regarding the maximal thread number. On a GTX470 a warp contains 32 threads, which will be executed in parallel by the SPs of the SM.

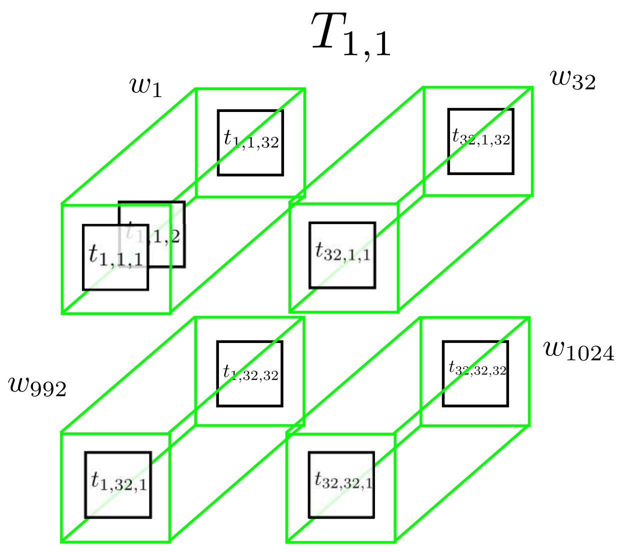


Fig. 5.3.: Warps inside a three dimensional thread block

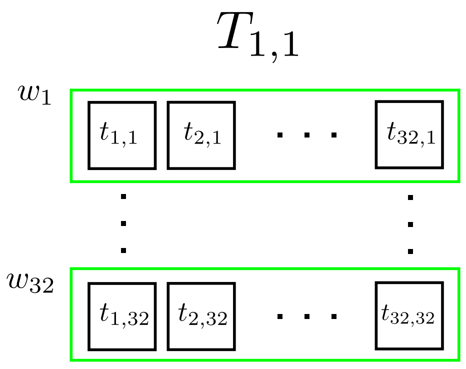


Fig. 5.4.: Warps inside a two dimensional thread block

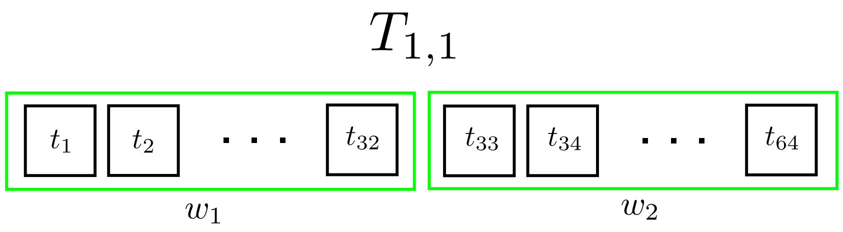


Fig. 5.5.: Warps inside a one dimensional thread block

The GPU uses these warps as an atomic schedule unit. Up to 48 warps may be active on a SM for the mentioned GPU. This means that the GPU schedules actively between those 48 warps, which are chosen arbitrarily from the assigned thread blocks.
The limitations for, e.g., assignable threads per SM or the global thread limit for the GPU, are specified in the CUDA capability levels . These levels are defined in [NVI10b]. For a GTX470 (CUDA capability level 2.0) the limitations are summarized in Table 5.1

Table 5.1.: Important limitations for the GTX470 GPU

Maximum x- or y- dimensions of a grid of thread blocks	65535
Maximum number of threads per block	1024
Maximum x- or y- dimension of a block	1024
Maximum z- dimension of a block	64
Warp size	32
Maximum number of resident blocks per multiprocessor	8
Maximum number of resident warps per multiprocessor	48
Maximum number of resident threads per multiprocessor	1536
Maximum number of instructions per kernel	$2 \cdot 10^6$

Up until now only the computational aspect of CUDA has been discussed. The actual data was considered a given constant in the explanations above. A GPU can access data in several ways, the most important for us are

1. copying data from host memory to GPU memory
2. accessing data on the host directly via the PCIexpress bus

Before analyzing both methods it should be stressed out that none of both methods is superior to the other. For every given problem is must be carefully decided which approach would yield the best performance.
The GPU memory, also called global or device memory, is considerably faster than the host memory . Yet its size is often much smaller than the host memory. It might not be possible to fit all the data into GPU memory at once and incremental copying might be necessary. Many copy operations with huge data chunks between host and GPU can reduce the benefits of using CUDA considerably. The use of GPU memory is recommended when many operations will be executed on a relative small amount of data, i.e., data that fits completely into GPU memory.
It is also possible for the GPU to access the hosts memory directly. This may reduce the size restrictions considering GPU memory. The downside to this lies in the transfer speed of the data. If a kernel accesses the hosts memory many times during his execution, the algorithm will be slowed down to a large extend.

5.2. Optimization Methods

In the previous section we have developed an understanding about the basic CUDA concepts. We will now briefly review important optimization strategies which have been considered in our implementation.

5.2.1. Coalesced Memory Access

Under certain conditions the GPU can optimize the memory access to device memory. The requirement for such an optimization is a coalesced memory access by the kernel. Coalesced memory access refers to a specific access pattern by the kernel. Again for reasons of simplicity, let us assume the kernel will be executed by 128 threads $t_1, ..., t_{128}$ in a one dimensional block, inside a 1x1 grid. Additionally the kernel shall work on a 128 byte array, residing in global memory. This memory can be linearly split into segments $s_1, ..., s_4$ of size 32. The GPU threads will be linearly scheduled into warps $w_1, ..., w_4$ of size 32. For coalesced memory access, each thread $t_i,\ i = 1, ..., 16$ in a half warp $\tilde{w}_{k,j},\ k = 1, .., 4\ ,\ j = 1, 2$ has to access the $(j-1) \cdot 16 + i$-th byte in segment s_k. This concept is visualized in Figs. 5.6 and 5.7.

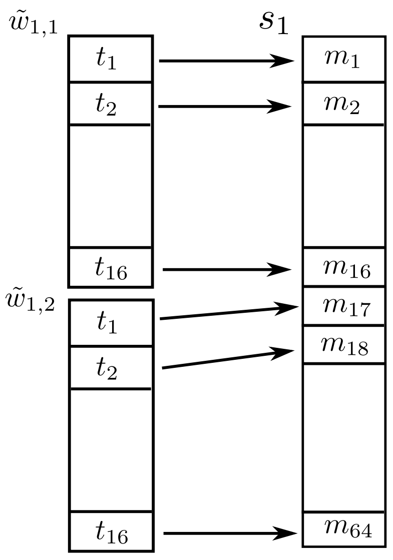


Fig. 5.6.: example of coalesced memory access

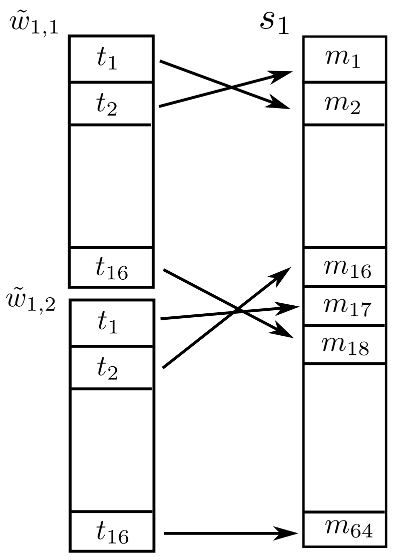


Fig. 5.7.: example of non-coalesced memory access

The explanation above refers to the basic principle as introduced in the specification of CUDA computability level 1.0 (see [NVI10a]). The requirements are relaxed with increasing level and one should refer to the appropriate documentation regarding the device's level.

5.2.2. Pinned Memory & Zero-copy

Pinned memory refers to page locked host memory, i.e., memory which will not be swapped by the operating system. The use of pinned memory increases the transfer rates for data copying between host and GPU. This speed up results from the fact that the basic data copying from host to device memory involves the following stages: copy data from non-pinned host memory to pinned host memory and afterwards to device memory. The usage of pinned memory does not require an additional transfer to pinned host memory. Yet one should use the allocation of pinned memory with caution as is reduces the overall memory available to the system.
The GPU can also access host memory directly, this is referred to as zero-copy memory access. The GPU can only access pinned memory on host side. Zero-copy memory access allows a GPU to overcome the size restrictions of its own memory. In other words, the GPU can access large data amounts within the hosts memory. Yet this access is a high latency operation and should be used with caution.

5.2.3. Block dimensions & Thread Resources

The choice of the thread block dimension is very important, as it can be used to control the thread scheduling and thus the performance of the algorithm. As described above, each SM consists of multiple SPs. Every SM has a limited amount of resources, this includes limited memory to execute a kernel using the assigned threads. Each thread incorporates his own copy of the kernels data, e.g., if the kernel declares 10 double variables, each the size of 64 bits and the SM will be assigned 1000 threads, the SM has to provide 80 kB of memory. If the requirements are not met, the kernel will not be launched at all. Thus the developer has to be careful when designing kernels.

5.2.4. Thread Divergence

Another crucial aspect in kernel design is the problem of thread divergence. The CUDA architecture is not driven by the SIMD (single instruction, multiple data) paradigm. Where data is prefetched and afterwards used by a specific instruction, e.g., instructions in SSE in Intel CPUs. The GPU executes kernels by following the SIMT (single instruction, multiple-thread) approach. It executes a (common) instruction for all threads within the same warp before continuing with the next instruction. Let us now consider the kernel in Listing 5.1. This kernel obviously splits the control flow during execution. The GPU will require multiple execution passes to execute each warp involved in this kernel. One pass for the threads which handle the if-part and one pass for the else-part, thus parallelism will be reduced. The homogeneity of a kernel, i.e., the amount of control flow splits, is a measure for the divergence of the control flow during execution.

Listing 5.1: Kernel yielding thread divergence

```
__global__ inefficientKernel(double* someArray)
{
 //thread coordinates inside the block
 int threadX = threadIdx.x;
 int threadY = threadIdx.y;

 if(threadX>=2 && threadY<=15)
 {
 someArray[i,j] = i+j;
 }
 else
 {
 someArray[i,j] = i-j;
 }
}
```

Control structures as if-else blocks should be avoided as much as possible. A kernel yielding no thread divergence at all, will be called a homogeneous kernel. Otherwise we will call the kernel inhomogeneous.

6. Implementation

This chapter explains the implementation of the algorithms from Chaps. 2 and 4. At the time this thesis was written, many frameworks for numerical linear algebra existed. Yet most of them represented simple wrapper APIs for the well-established framework LAPACK. This library provides a wide variety of numerical methods from linear algebra, yet it is written in Fortran and does not include SMP or even GPU support. Regarding an implementation in the C/C++ language, the *GNU scientific library* (GSL) represents a popular framework. NVidia also provides its own version of the LAPACK libraries, yet this framework is closed source and only commercially available. Thus it is unclear what algorithms have been used and how efficient their implementation is. Additionally it provides only a tiny fragment of the LAPACK functionality. For this thesis the GSL has been choosen due to following reasons:

- It provides access to the source code, thus fragments can be used and enhanced for, e.g., SMP
- It is still in development, thus new features might be available in time, e.g., new algorithms

Whenever possible, methods which were already present in the GSL, have been used to solve subproblems in our implementation. In other cases, GSL routines have been enhanced by replacing subroutines with CUDA counterparts. Not every such replacement was implemented from scratch, i.e., by writing a custom kernel for it. CUDA also includes basic functions for computations in linear algebra. These functions are bundled under the so called CUBLAS portion of CUDA. In specific cases, the usage of CUBLAS can yield a major performance gain. To understand these specific cases one has to consider the complexity of basic calculations from linear algebra. These are grouped in three so-called BLAS levels. The first level includes all scalar-vector operations, e.g., the scaling of a vector by multiplying each element with a real number. Level two consists of vector-vector operations ,e.g., adding two vectors or the euclidean scalar product. Level three incorporates all matrix-matrix operations, e.g., the matrix multiplication. As shown in [aWmH10] it is unwise to move all BLAS calculations to the GPU. The reason for this is quite simple, the overhead of computing small problems on a GPU outweighs its benefits. Small calculations, like the computation of a scalar product with two 5-dimensional vectors, are carried out faster on the host side. Additionally Chap. 7 will verify these facts through benchmarks of basic

CUBLAS and GSL methods for different problem sizes.
Our implementation consists of so-called hybrid-algorithms, which execute subroutines on the GPU only if it yields a massive speedup compared to the host side. The functions in level three can reach a complexity of, e.g., $\mathcal{O}(N^3)$, thus if a parallel approach exists for them, the speed-up will be immense. Especially the ordinary matrix multiplication benefits greatly for huge values N. Thus certain parts of the algorithms from the chapters above will be reformulated to use level three routines.
In the following sections we will omit code fragments as far as possible, listings will be given only if they are self-explanatory or intuitively to grasp. The major implementation concepts and their benefits as well as drawbacks, will be explained. Additionally the encountered problems and their solutions will be addressed. If not mentioned otherwise, all given numbers refer to the GTX470 GPU.
Before we start, an important aspect must be mentioned. The host and device memory is organized in a one dimensional fashion, i.e., by numbering the memory *slots*. Yet a matrix is a two dimensional structure, thus the indices must be mapped onto the corresponding memory addresses. This mapping can be done in two ways, either as column- or row-major mapping. The difference between both methods is visualized in Fig. 6.1. The column-major format sequentially aligns the columns of a matrix in memory, while the row-major format does the same for the rows of the matrix. The conversion between both formats can not be done by a simple call to *memcpy*. The data needs to be copied by a single for-loop, i.e., regarding a $N \times N$ matrix, N^2 for-loop iterations are required for this.

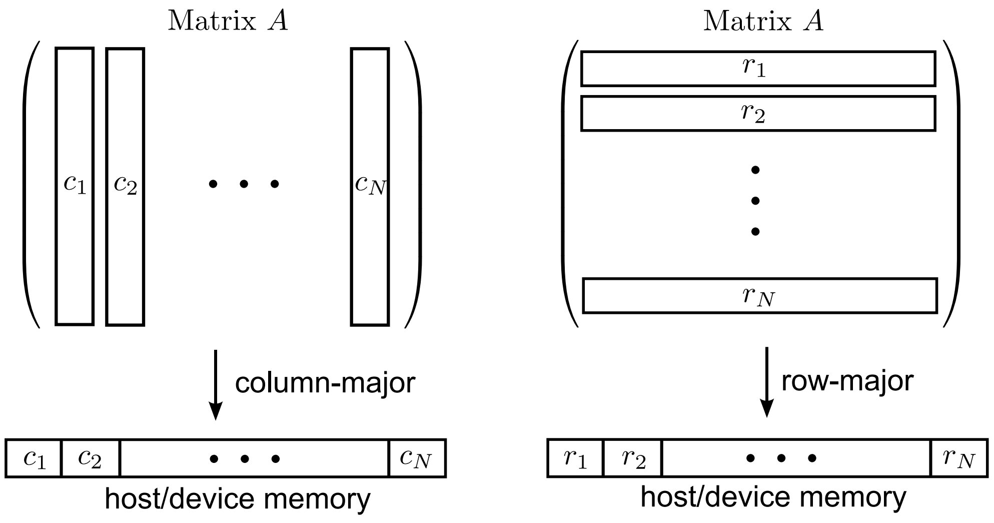


Fig. 6.1.: mapping a matrix onto host/device memory

One problem lies in the different philosophies of CUDA and GSL, while CUDA (and Matlab) uses the column-major alignment, GSL uses row-major alignment. This requires some overhead during data exchange between both frameworks, yet this poses no major problem in the following implementations. The reason for this is the fact that such a data exchange (e.g. the conversion of a CUBLAS matrix to a GSL matrix) only occurs in the initialization or the end phase of an algorithm. The conversion times (up to 20ms) are relatively small compared to the involved computation time (up to 5 minutes).
Additionally an interface for Matlab was required, thus a flexible matrix class (Listing 6.1) was developed. It can be used for GSL and CUDA (format conversion is performed in the background) and provides methods for data import/export from/to Matlab.

Listing 6.1: the main data structure used in this thesis

```
template<class T> class Matrix {
public:
        Matrix(int type);
        Matrix(Matrix<T>* matrix, int type);
        virtual ~Matrix();
        void initMatrix(int rows, int columns, bool pinned
           =false);
        void setData(int i, int j, T value);
        T getData(int i, int j);
        T* getDataPtr();
        int getRowCount();
        int getColumnCount();
        void readMatrixFromFile(std::string filename);
        void writeMatrixToFile(std::string filename);
        void exportMatrix(std::string filename);
        void readMatrixFromMatlabFile(std::string filename
           ,std::string varName);
        void writeMatrixToMatlabFile(std::string filename,
           std::string varName, bool compressed);
        int getType();

private:
        int type;
        int rows;
        int columns;
        bool pinned;
        T* data;
        bool readCSVValue(FILE** pFile, std::string* s);
};
```

Additionally this data structure provides pinned memory if desired, the details of each method can be viewed in the documented source files.

6.1. QR Algorithm

The implementation of the QR algorithm involved the following tasks:

1. extending the GSL routines with an interface which allows communication with CUDA methods
2. implementing Alg. 2 and the required subroutines with CUDA enhancement
3. ensure the correctness of the implementation

The GSL already provides the QR algorithm as described in Chap. 2, yet it does not utilize SMP. For this thesis the GSL implementation has been modified. More precisely, the matrix tridiagonalization has been ported to the GPU. An interface has been created which allows injection of an already tridiagonalized matrix into the QR routines within GSL. This concept is visualized in Figure 6.2.

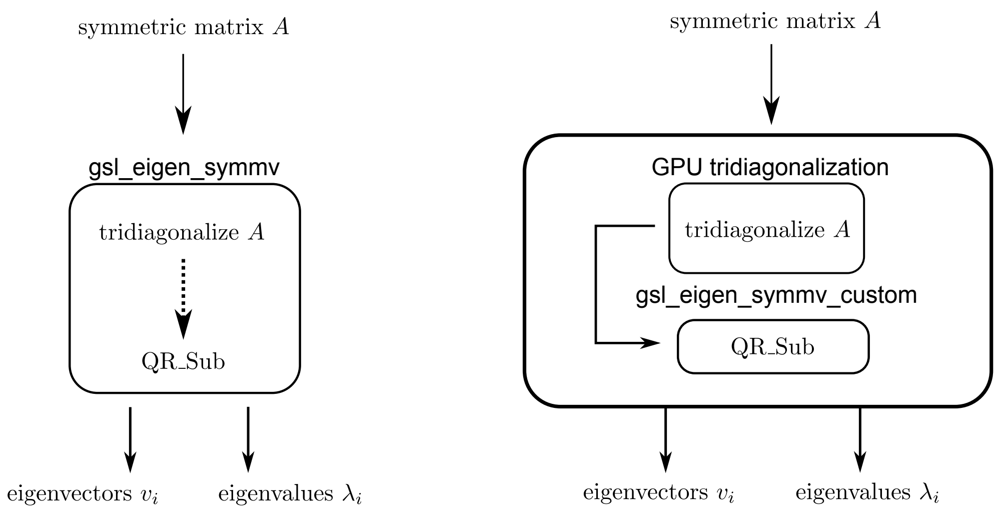


Fig. 6.2.: Modification of the GSL to allow injection of an already tridiagonalized matrix

The original GSL function *gsl_eigen_symmv* consists of the matrix tridiagonalization and QR steps. In our implementation this procedure has been reduced to only incorporate the QR steps, yet this requires the input of an already tridiagonal matrix. This matrix will be provided by a subroutine which utilizes the GPU to tridiagonalize the given matrix. The reason for not porting the *QR_Sub* procedure to the GPU as well, lies the algorithms internal structure, which does

not allow parallelization.
We will now look in detail onto the implementation of the matrix tridiagonalization. The *tridiagonalize* function follows the GSL convention, i.e., it splits the actual algorithm into two parts. One handles the tridiagonalization (algorithm 2) and creates a compact representation of the matrices T, Q in just one matrix. The other part expands this compact representation into two full size matrices. This parts are handled by the functions *symmtd_decomp* and *symmtd_unpack*, respectively. Before we continue with the details of those functions, we shall look on the compact representation $\aleph \in \mathbb{R}^{N \times N}$ of the tridiagonalized matrix T and the corresponding tridiagonalization matrix Q. Fig. 6.3 shows the form of such a matrix $\aleph$.

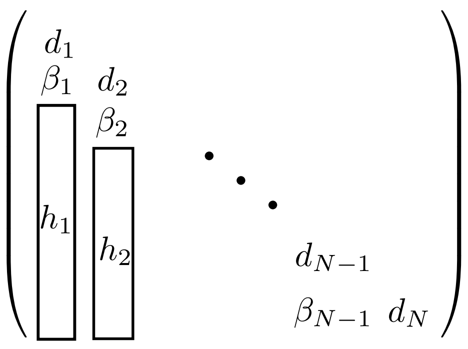


Fig. 6.3.: Representation of a tridiagonalization $A = Q^T T Q$ with just one matrix $\aleph$

The diagonal elements d_i represent the matrix T, under each of these entries lies one Householder coefficient and the corresponding Householder vector. Tridiagonalization of a $N \times N$ matrix requires $N - 1$ Householder reflections. At this point is should become more clear what was meant in chapter 2 by '... saves half the space ...'. The first element of the Householder vector is always 1. h_1 is of dimension $N - 1$. Leaving out the first element, we need to save only the remaining $N - 2$ elements. Yet the first column has $N - 1$ *slots* left under the diagonal element, this allows us to save even β_1 between h_1 and d_1. Additionally it should be noted that $h_{N-1} = 1$ thus we only store β_{N-1} under d_{N-1}.
Now to the methods *symmtd_decomp* and *symmtd_unpack*. *symmtd_decomp* computes the tridiagonalization and creates $\aleph$. It is basically Alg. 2 which is already rich in BLAS-3 routines and thus needs no reformulation. Yet it involves the sub routine *house* which only incorporates the euclidean dot product. This operation (taking the remaining structure of the algorithm into account) can not be extended to a higher BLAS level. Our implementation works nearly completely on the GPU, only the scalar routines in *house* are executed on the host. Thus only $(N - 1)$ double values (i.e. the β values) are exchanged between host and device within the inner part (i.e. omitting initialization) of one tridiagonization.

Once the matrix $\aleph$ has been created, *symmtd_unpack* can create the matrices T and Q. The creation of T is straight forward as is only consists of copying the diagonal values of $\aleph$ into an empty matrix, this requires $N-1$ copy operations. The important step is the creation of Q, this was done completely on the GPU by using Alg. 3 which consists of level 3 routines. No custom kernels have been developed for these functions, only routines from CUBLAS have been used. It would be difficult to create a homogeneous kernel which provides the computation of a Householder vector and corresponding coefficient.
Testing the implementation of numerical methods can be a difficult task, cancellation can stay undetected for small matrix dimensions. This was also the case in this thesis. Using the matrix (Eq. 3.5) cancellation became evident at $N \approx 70$. Verifying the correctness of the implementation was done by comparing the results with reference implementations, i.e., Matlab and GSL. The results do not have to be identical as Matlab uses different algorithms. Additionally the GPU uses different representations of floating point numbers (e.g. different rounding and cut-off policies). Thus comparison was done by calculating the Frobenius norm $|||.|||_F$ of the difference matrices $(A_{\text{gsl, matlab}} - A_{\text{CUDA}})$.

6.2. Two-Sided Jacobi Method

The two-sided Jacobi method was implemented without the usage of CUBLAS routines. The reason for this lies in the fact that the involved 2-dimensional Givens rotations are not suitable (regarding their performance) for CUBLAS functions. Our implementation is inspired by the work of [GS10]. The possibly easiest method to understand the program structure of our implementation, is by looking at Figure 6.4. We will omit the details of initialization here, one might refer to the source code to see the details. Let A be a $(n-2) \times (n-2)$ matrix. The following explanations will refer to a dimensionality of n, i.e., it will seem as we try to diagonalize a $n \times n$ matrix. The reason for that will be discussed later, at this point it is sufficient to consider A temporarily as a $n \times n$ matrix.
First the matrix V will be set to $\mathbb{I}_n$, this matrix will be holding our eigenvectors $\mathbf{v}^i$. Additionally a matrix $\tilde{A}$ is being initialized, this matrix will be explained later. Afterwards the actual algorithm begins his work, $3(n-1)\frac{n}{2}$ Givens rotations are carried out on the GPU. We will now take a more detailed look on where this number of rotations originates from. As described in Chap. 2 the cyclic-row extension works by grouping the rotations in a single sweep into so called subproblems. Each subproblem-set consists of $\frac{n}{2}$ Givens rotations which can be executed in parallel. There are exactly $n-1$ subproblem-sets in a single sweep. This translates into Fig. 6.4 as follows. $\frac{n}{2}$ rotations are applied on the rows of $\tilde{A}$, yet these rotations also need to be applied on the columns of $\tilde{A}$. Obviously this can not be done simultaneously. One has to wait until all row operations

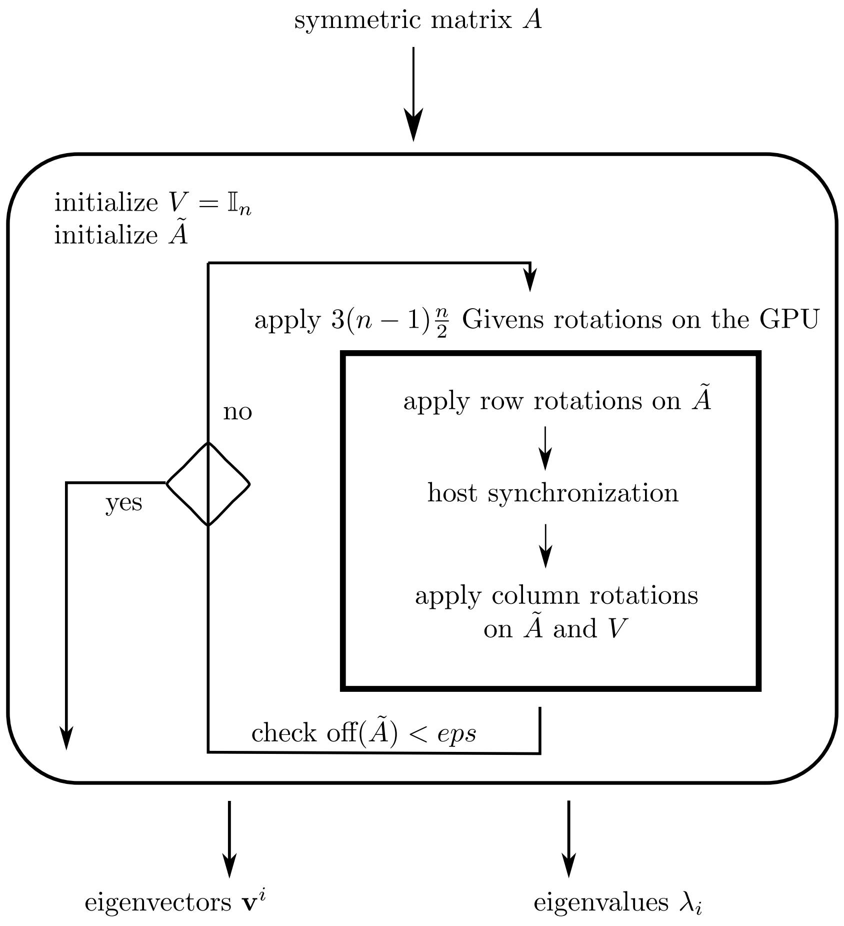

symmetric matrix A
initialize $V = \mathbb{I}_n$
initialize $\tilde{A}$
apply $3(n-1)\frac{n}{2}$ Givens rotations on the GPU
apply row rotations on $\tilde{A}$
host synchronization
apply column rotations on $\tilde{A}$ and V
no
yes
check off$(\tilde{A}) <$ *eps*
eigenvectors $\mathbf{v}^i$
eigenvalues λ_i

Fig. 6.4.: Implementation outline of the two-sided Jacobi method

have finished before continuing with the columns. This step can only be done by synchronizing all threads on the GPU (e.g. by a thread barrier). As described in Chap. 5, the technique for this is host synchronization. Thus we have 3 applications of $\frac{n}{2}$ Givens rotation for a singe subproblem-set. There are $n-1$ subproblem-sets, which implies a total of $3(n-1)\frac{n}{2}$ Givens rotations in a single sweep.
So far we have described the implementation in a macroscopic view, now we take closer look on the developed kernels for executing these rotations. Again a visualization, given by Fig. 6.5, will accompany us. We used column-major alignment on the device memory. Every thread on the GPU handles a single column of the matrix, e.g., t_{k+1} executes $\frac{n}{2}$ Givens rotations on column c_i, which consists of values $\tau_j,\ j \in [n]$. At this point the question arises about the amount of threads to be launched and the dimensionality of the grid G and the thread blocks. The answer to this question lies completely in the structure of the look-up table LUT for the subproblems. Each column in LUT represents a Givens rotation, thus we split the table into segments of length 32 (chosen to be same

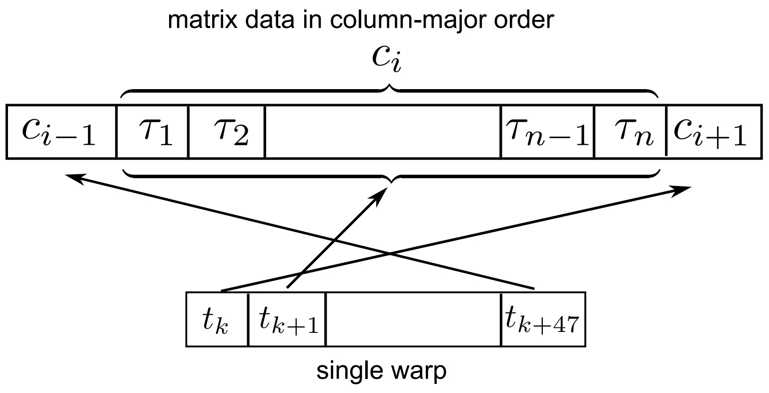


Fig. 6.5.: Threads executing Givens rotations

length as a warp). If the size of *LUT* is no multiple of 32 we pad the table with enough additional columns. Each of them containing the numbers $n+1$ and $n+2$ in the first and second row respectively (see Fig. 6.6).

1	3	5		$n-1$	$n+1$	$\cdots$	$n+1$
2	4	6		n	$n+2$	$\cdots$	$n+2$

Fig. 6.6.: Padded look-up table for subproblem-set 1

This approach allows us to launch $\frac{n+z}{2\cdot 32}$ 1-dimensional thread blocks of size 32 whose threads execute a homogeneous kernel ($z := 32 - [\frac{n}{2}\ mod\ 32]$ stands for the amount of padding). The choice for the content of the additional columns is motivated by the access onto the matrix data. The kernel does not work on the matrix A, for the strategy described so far to work, the matrix A must be padded with two additional columns and rows. Thus we work on the matrix $\tilde{A}$ given by

$$\tilde{A} := \begin{pmatrix} A & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \in \mathbb{R}^{n \times n} \tag{6.1}$$

At this point is should be visible that our additional columns in *LUT* represent *dummy rotations* on the added rows and columns of $\tilde{A}$.
A naive way of implementation would be to pre-calculate this look-up table and store it in device memory. Each thread could then access the appropriate indices from global memory and start executing the rotation. Which essentially, is nothing more than a for-loop with n iterations. Yet, according to [NVI10a] a single access to global memory takes ≈ 400 – 600 clock cycles. Thus this approach would yield a severe overhead. To overcome this obstacle each thread can compute the appropriate table elements *on-the-fly* during execution. [GS10] presented a similar approach, for this thesis a slightly modified version has been developed. First

it should be pointed out that the method to create a certain subproblem-set is equivalent to the method depicted in Fig. 6.7.

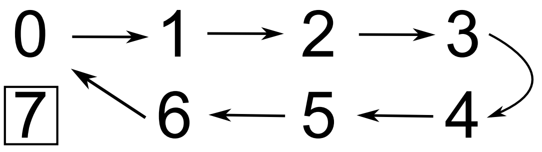


Fig. 6.7.: Another approach to create a subproblem-set

The main differences to Chap. 2 are the counting order and the fixed element. Additionally we start counting at 0. A table as the one in Fig. 6.7 can be linearly mapped to the 1-dimensional structure depicted in Fig. 6.8.

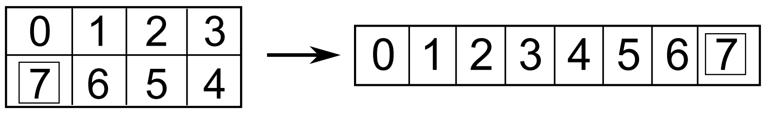


Fig. 6.8.: Mapping a subproblem-set onto a 1-dimensional array

The shifting is needed only for the first $n-1$ elements. It can be carried out by simply calculating $x+(n-2)\ mod\ (n-1)$ where $x \in \{1,...,n-1\}$ is the desired position in our 1-dimensional structure. Because each look-up table is a multiple of 32, each 1-dimensional mapping will be one as well. Thus each thread can access the mapping as depicted in Fig. 6.9. The numbers next to the arrows indicate corresponding elements, e.g., $(2,5)$ indicated by the number 3 represent one single subproblem.

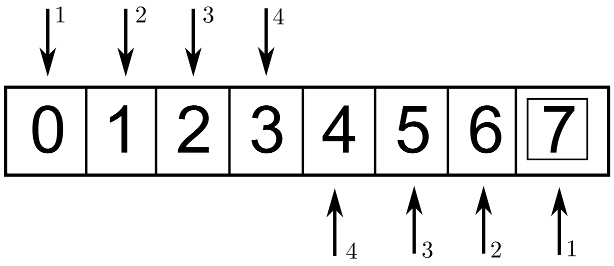


Fig. 6.9.: Thread accessing a mapped first lookup-table i.e. subproblem-set 1, example for $n = 8$

To understand the final formula, one should first refer to Fig. 6.10, where the *on-the-fly* computation of the look-up table is visualized. To access a pair from the subproblem-set 2 for $n = 8$, one just has to carry out the depicted calculations. Only the basic array from Fig. 6.7 is needed.

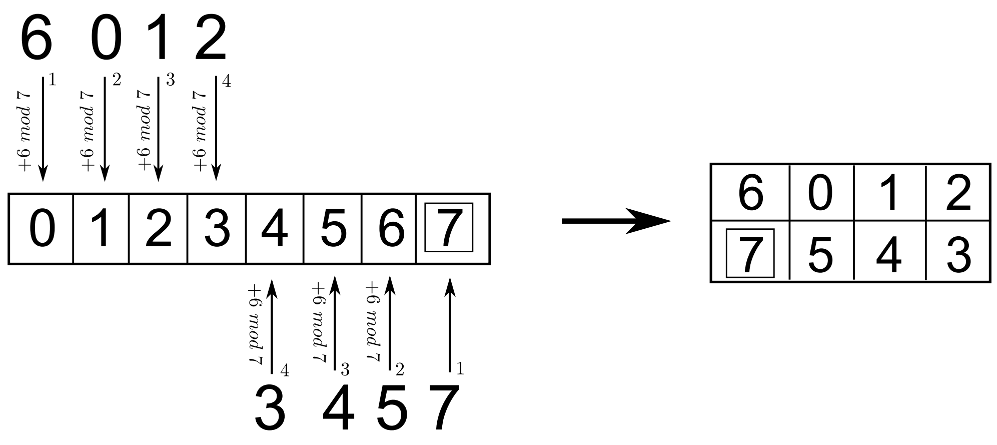


Fig. 6.10.: Computing the subproblems for subproblem-set 2, $n = 8$

Theorem 6.2.1 (On-the-fly calculation of subproblem-sets).
Each subproblem $p_{j,l} = (i,k) \in \{0,...,n-1\}^2$, $j \in \{1,...,\frac{n}{2}\}$ *of subproblem-set* $l \in \{0,...,n-1\}$ *can be calculated by*

$$p_{j,l} = \begin{pmatrix} (j-1) + l \cdot (n-2) \ mod \ (n-1) \\ [(n-1)-(j-1)] + l \cdot (n-2) \ mod \ (n-1) \end{pmatrix} \tag{6.2}$$

with the exception of

$$p_{0,l} = \begin{pmatrix} (j-1) + l \cdot (n-2) \ mod \ (n-1) \\ n-1 \end{pmatrix} \tag{6.3}$$

Both equations in a single kernel need an if-else structure, thus a homogeneous kernel becomes impossible.
Additionally one might argue that this approach yields no speed-up by using coalesced memory access. It would be possible to avoid the for-loop inside the kernel and instead use much more threads, i.e., each thread would perform one iteration of the for-loop approach. This suggestion must be analyzed carefully, firstly it still would yield the same amount of accesses to the device memory. Secondly only one kind of Givens rotations would benefit from coalesced memory access, either the row rotations or the column rotations (depending on how one aligns the matrix in memory, either in row-major or in column-major order). Thus only one half of the algorithm would benefit from coalescing, yet this is not the major disadvantage of this approach. The kernel would need a control-structure, e.g., an if-else block, to finalize the computation. Thus in the worst case the execution speed would be reduced by another half. Another inherent drawback is the subroutine Givens(A, i, k) (Chap. 3) as it splits up the control flow twice.
Summarized it can be said that our approach allows up to 448 Givens rotations to be executed in parallel while minimizing the thread divergence. A major bottleneck in this solution is the host synchronization which contributes a noticeable

amount to the overall running time.
This bottleneck will become more evident in the extension to the multi-GPU version, depicted in Fig. 6.11. In the case of 2 GPUs, one might handle the

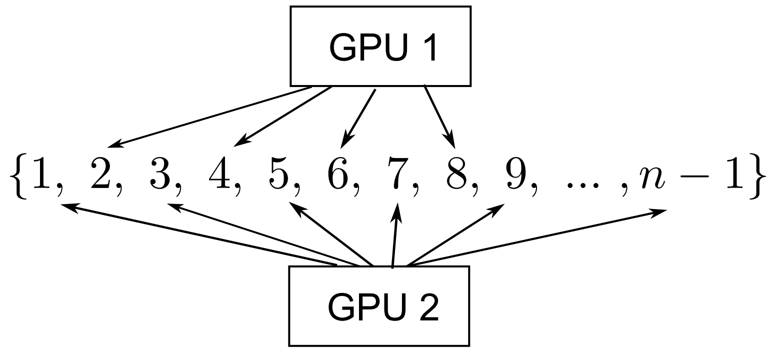


Fig. 6.11.: Distributing subproblem-sets onto two GPUs

odd numbered subproblem-sets while the other one handles the even numbered. Yet the host synchronization from above, has to be extended onto multiple devices. Every GPU needs the results of all the other ones. The most reasonable approach would be to either use a common host memory space (e.g. zero-copy memory access) or copy the result back to the host memory, wait until all devices have copied their results and then copy the complete data back into every device. Both approaches have benefits and disadvantages. Using a common host memory space allows the handling of very large matrices but slows the kernel execution down due to high latencies. The use of a temporary cache on host side does not slow the execution down. Yet the matrix size is limited by the lowest memory capacity of the GPUs involved. Additionally the synchronization procedure becomes more complex, as each GPU must copy only specific rows and columns back to the host.
The correctness has been verified in the same manor as with the QR algorithm.

6.3. Kernel Principal Component Analysis

In this section we will describe our implementation of the GPU-enhanced KPCA. Before describing the complete algorithm we will explain the major improvements and differences compared to the standard (i.e. CPU based) KPCA. The execution of the KPCA has been enhanced by the usage of a GPU in multiple ways, which are specifically:

- creating the Gram matrix on GPU
- performing matrix centering on the GPU

- reformulating the projection of all data points in a single matrix multiplication

We will first take a look on how the Gram matrix was created on the GPU, a new kernel has been developed for this. Considering a set of p n-dimensional data points $\mathbf{x}^i$, $i \in [p]$, where p is a multiple of 32, the strategy is as follows. Every launched thread computes one single element of the Gram matrix. This computation is basically a simple for-loop with n iterations. In general the assumption regarding p will not hold, thus we will require some data padding. Before diving into the details of this padding, we shall take a look on Fig. 6.12, which visualizes the concept described so far. The thread $t_{1,1}$ in block $T_{1,1}$ will compute

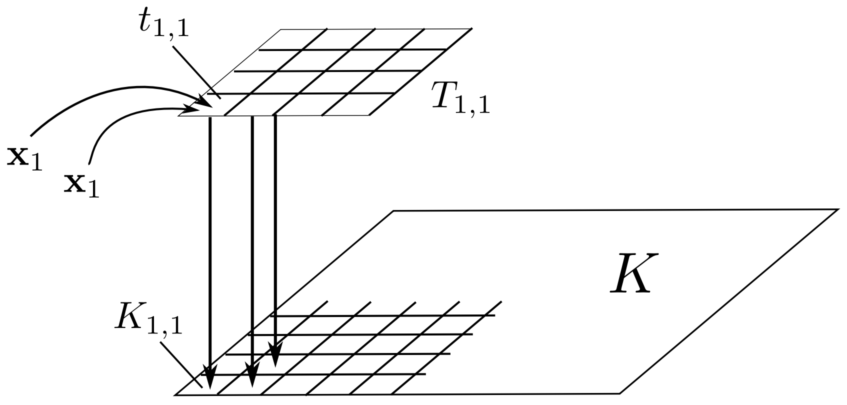


Fig. 6.12.: Computing the Gram matrix by a kernel on the GPU

element $K_{1,1}$ of the Gram matrix K, the needed data for this consists of the vector $\mathbf{x}^1 \in I$. Each thread block has a dimension of 32×32, as each thread computes a single element of K. If p is not a multiple of 32, the described concept has to be extended as follows. The working space, i.e., where the computed values will be stored, has to be padded accordingly. To avoid control flow splits e.g. if-else segments, the number of input vectors $\mathbf{x}^i$ has to be increased as well, Fig. 6.13 shows this approach. It depicts the case where the memory space has to be padded by only 1 unit, thus only one row and one column is added to K. Every cell of the memory(-grid) holding K will be handled by a unique thread, thus all thread blocks cover this grid without an overlap. The thick black line represents the border between elements of K and dummy entries in the added rows/columns. These dummy entries will be discarded after the computation has finished. Their purpose is nothing more than the avoidance of if-else parts within the kernel. As mentioned before, each thread computes a single element $K_{i,j}$ by using the input vectors $\mathbf{x}^i, \mathbf{x}^j$.
Once the matrix has been extended with one row and one column we will need one additional vector $\mathbf{x}^{p+1}$. The threads at the border of the extended matrix can then use $\mathbf{x}^{p+1}$ to compute the elements $K_{p+1,j} = k(\mathbf{x}^{p+1}, \mathbf{x}^j)$. Once the gram

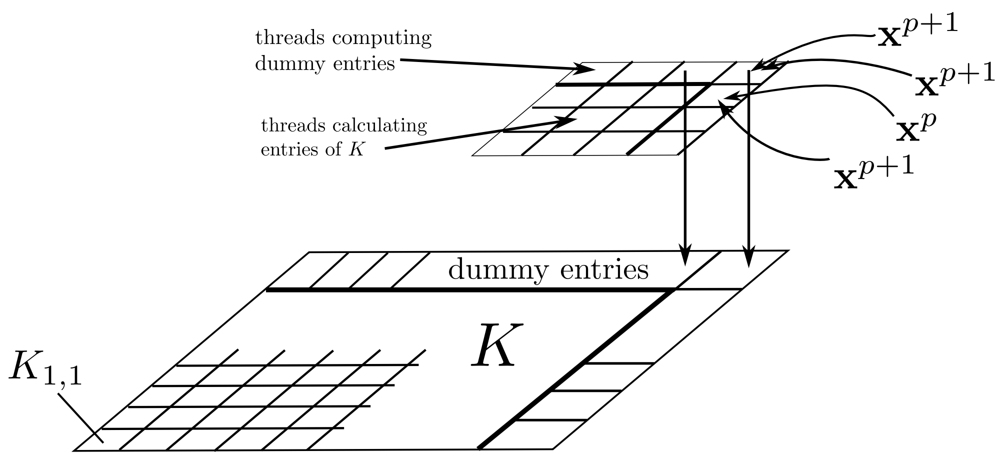


Fig. 6.13.: Padding the device memory according to the dimension of the thread blocks, here for the case where only one column and row have been added

matrix has been computed and before solving the SEVP, it must be centered. During implementation it became evident that this centering represents a major bottleneck during the KPCA, as it requires 4 matrix multiplications. Considering the matrix dimensions encountered in this thesis (up to 6000), it is unfeasible to execute these multiplications on the host side (as will be shown in Chap. 7). Thus the centering has been transfered to the GPU.
In chapter 2 it was explained how a single vector $\mathbf{x} \in I$ is being projected onto the principal components $\boldsymbol{\alpha}^i$ in feature space. This was formulated by a single sum for each input vector, for this thesis another formulation was developed. We will give a projection formula that projects all desired vectors onto a set of principal components, in terms of a single matrix multiplication.

Theorem 6.3.1 (Projecting multiple input vectors)**.**
Let $\boldsymbol{x}^l \in I,\ l \in [w]$ be the training set for the KPCA. Furthermore let $\boldsymbol{y}^j \in I,\ j \in [n]$ be the set of vectors to be projected onto the principal components $\boldsymbol{\alpha}^i,\ i \in [m],\ m < w$ in feature space. The projections $\tilde{\boldsymbol{y}}^j,\ j \in [n]$ are given by

$$\begin{pmatrix} k(\boldsymbol{x}^1, \boldsymbol{y}^1) & k(\boldsymbol{x}^2, \boldsymbol{y}^1) & \dots & k(\boldsymbol{x}^w, \boldsymbol{y}^1) \\ k(\boldsymbol{x}^1, \boldsymbol{y}^2) & k(\boldsymbol{x}^2, \boldsymbol{y}^2) & \dots & k(\boldsymbol{x}^w, \boldsymbol{y}^2) \\ \vdots & \vdots & \dots & \vdots \\ k(\boldsymbol{x}^1, \boldsymbol{y}^n) & k(\boldsymbol{x}^2, \boldsymbol{y}^n) & \dots & k(\boldsymbol{x}^w, \boldsymbol{y}^n) \end{pmatrix} (\boldsymbol{\alpha}_1 \dots \boldsymbol{\alpha}_m) = (\tilde{\boldsymbol{y}}_1 \dots \tilde{\boldsymbol{y}}_n)^T \tag{6.4}$$

The actual implementation precomputes the kernel matrix (i.e. the matrix with kernel calls inside) using the same method as for the Gram matrix. Afterwards the multiplication is again carried out through the CUBLAS framework. The described projection method yielded significant speed improvements compared to other implementations (especially Matlab). We will now show the complete implementation with the most relevant points, for more details (e.g. initialization procedures) one should refer to the provided source code.

Algorithm 10 KPCA

```
Input: Training set $(\mathbf{x}^i)_{i\in[w]}$
       Set of vectors to be projected $(\mathbf{y}^i)_{i\in[n]}$
       Dimension count $m$
1: init memory for $K$, kernel matrix,
   eigenvector matrix $Q$, eigenvalue vector $\Lambda$
   temporary buffer $\tilde{Q}$
2: compute $K$ on GPU
3: compute kernel matrix on GPU
4: center Gram matrix on GPU
5: compute eigenvectors and eigenvalues of $K$ using the GPU
6: sort eigenvectors according by their eigenvalues in descending order
7: compute projection of $(\mathbf{y}^i)$ onto first $m$ eigenvectors using the GPU
```

The sorting of the eigenvectors is a non critical task regarding the execution time. Only the eigenvalues have to be sorted, e.g., for w data points, w numbers have to be sorted in descending order. As our experiments *only* involve up to 6000 eigenvalues, this sorting will be carried out on the host side. Afterwards the corresponding eigenvectors are sorted by copying them into the right position in a temporary buffer $\tilde{Q}$. From there the sorted vectors will be copied back into the original matrix Q. This requires at most $2w$ *memcpy* calls, which in turn only occupies a small fragment of the overall execution time (diagonalization can take up to 5 minutes for a 5000×5000 matrix).
One might argue that this copying is not necessary. It would also be possible to use a simple two dimensional look-up table which connects the sorted eigenvalues and the corresponding eigenvectors. Yet we require the eigenvectors to be aligned in memory according to their eigenvalues, as we need to perform an efficient matrix multiplication.

6.4. k-nearest Neighbors & Linear Discriminant Analysis

For this thesis only the KNN algorithm has been ported to the GPU. The reason for this is simply that only this algorithm can benefit significantly from parallel computation. The LDA was carried out through Matlab, as its implementation utilizes SMP and the algorithm itself *only* involves a single simple matrix inversion. The results from a KPCA have been exported to Matlab and used there by

the mentioned routines.
We will now explain our implementation of the KNN algorithm, it can be divided into two parts:

1. compute the distances of every test point to all training points and create a distance matrix $\mathcal{D}$
2. use $\mathcal{D}$ to classify all test points according to a given parameter k

Let $(\mathbf{x}^i)_{i\in[w]}$ be the set of training points and $(\mathbf{y}^i)_{i\in[m]}$ the set of test points. The distance matrix $\mathcal{D}$ is defined by

$$\mathcal{D} := \begin{pmatrix} ||\mathbf{x}^1 - \mathbf{y}^1|| & ||\mathbf{x}^1 - \mathbf{y}^2|| & \dots & ||\mathbf{x}^1 - \mathbf{y}^m|| \\ ||\mathbf{x}^2 - \mathbf{y}^1|| & ||\mathbf{x}^2 - \mathbf{y}^2|| & \dots & ||\mathbf{x}^2 - \mathbf{y}^m|| \\ \vdots & \vdots & \dots & \vdots \\ ||\mathbf{x}^w - \mathbf{y}^1|| & ||\mathbf{x}^w - \mathbf{y}^2|| & \dots & ||\mathbf{x}^w - \mathbf{y}^m|| \end{pmatrix} \tag{6.5}$$

The computation is carried out in exactly the same way as for the Gram matrix. Thus up to 448 elements will be computed in parallel. Considering this matrix, getting the k nearest neighbors of a test point $\mathbf{y}^i$ is equivalent with sorting the i-th column of $\mathcal{D}$ in ascending order and get the test points $\mathbf{x}^j$ corresponding to the first k entries. This procedure represents again a candidate for a host side implementation, as only up to $m \cdot w$ distance values need to be sorted. We do not need to sort any vectors in this case.
The classification was carried out on host side, yet it utilizes SMP by using the OpenMP framework. Our implementation classifies up to r given test points simultaneously, where r represents the number of available CPUs on the host. We now summarize the complete algorithm

Algorithm 11 KNN

```
Input: Labeled Training set $(\mathbf{x}^i)_{i\in[w]}$
       Set of unlabeled test vectors $(\mathbf{y}^i)_{i\in[m]}$
       Neighbors count $k$
1: init memory for $\mathcal{D}$, temporary buffers
2: compute $\mathcal{D}$ on GPU
3: DO IN PARALLEL
4: sort columns of $\mathcal{D}$ in ascending order
5: get first $k$ values and corresponding vectors $(\mathbf{x}^j)$ for each column $j$
6: assign each test point $\mathbf{y}^i$ to the class with most elements among
   the last $k$ corresponding vectors
7: OD IN PARALLEL
```

6.5. Support Vector Machines

The implementation complexity of the underlying algorithms for SVMs is comparable to the task of implementing QR or TSJM. Thus due to reasons of remaining time for this thesis, it became unfeasible to further pursue a GPU enhanced implementation of SVMs. As a result of this fact, two already existing SVM frameworks have been evaluated. These were the *Shark* and the *LibSVM* library. Although both frameworks claim to provide SMP support, only LibSVM provides it for SVM-based classification. Experiments with the Shark library yielded very poor performance for the OvO method (as it doesn't utilize SMP in this context). On the other side, LibSVM showed an acceptable performance for even large classification problems. Thus it was decided to use LibSVM for evaluating the OvO SVM classification.
Additionally the authors of LibSVM provide a Matlab-interface (in form of mex-assemblies). For this thesis an interface has been developed which provides Matlab with test and training data for the aforementioned LibSVM interface.

6.6. Boosting

The term boosting refers to a powerful technique which can increase recognition results by utilizing multiple classifiers (e.g. multiple SVMs), these classifiers are also called *base classifiers*. [Bis06] introduces boosting algorithms like the ADA-Boost, yet most of these approaches require a deep modification of existing classification methods. For this thesis a new approach has been developed and successfully evaluated. We will introduce a committee of multiple SVMs, which utilize a voting system to further increase the recognition rate of the optimal SVM (i.e. the SVM, optimal in the sense of definition 4.4.1, which has been determined through a grid search).
Before introducing our approach, we will explore the motivation behind it. Let us assume we have found a SVM S_{opt} that produces good recognition results. The term *good* refers to an acceptable average recognition rate (i.e. the optimization criteria in our definition of a template). S_{opt} has been determined by a grid search, meaning that several SVMs have been analyzed regarding their average recognition rates. Yet this process may skip SVMs with good local results, i.e., SVMs able to classify only certain patterns with high results. The following gedankenexperiment illustrates this problem very directly. Let us assume we are classifying power traces of our microcontroller. The optimal SVM S_{opt} yields an average rate of 65%, the instruction types *ADD*,*MUL* and *SUB* can be recognized with a success rate of 90% each, all other rates are below 20%. Thus the remaining commands have a very low rate, yet if we are only interested in the three instruction types from above, this SVM represents an acceptable choice.

Let us further assume that the process of finding S_{opt} involved a grid search with 30 vertices, i.e., 30 SVMs $S_1, ..., S_{30}$ have been evaluated. The classifier S_4 had an average rate of 48%, yet it showed a local rate of 80% for the *ADDWF* command. A similar situation happened for several other of these SVMs. Thus it may be wise, not to use only S_{opt} alone for our classification task.
We will now explain our boosting approach, yet another new term must be explained before that.

Definition 6.6.1 (Confusion matrix).
Let $\Xi : I \to [p]$ be a classifier working on $\mathfrak{C}$ and T, with $\mathfrak{C} := \{\mathfrak{C}_1, ..., \mathfrak{C}_p\}$ a set of classes, $T \subset I$ a set of training data, $U \subset I$ a set of test data. Furthermore it should hold that $\forall x \in T \exists! i \in [p] : x \in \mathfrak{C}_i$ as well as $\forall x \in U \exists! i \in [p] : x \in \mathfrak{C}_i$.
The matrix $Conf$ is defined by

$$Conf := \begin{pmatrix} g_1(U_1) & g_2(U_1) & \cdots & g_p(U_1) \\ g_1(U_2) & g_2(U_2) & \cdots & g_p(U_2) \\ \vdots & \vdots & \cdots & \vdots \\ g_1(U_p) & g_2(U_p) & \cdots & g_p(U_p) \end{pmatrix} \in \mathbb{R}^{p \times p} \tag{6.6}$$

where $U_i \subset U$ represents the set containing only test samples from class i, the functions $g_i : (U_j)_{j \in [p]} \to \mathbb{R}$ are defined through

$$g_i(U_j) := \frac{1}{|U_i|} \sum_{\mathbf{x} \in U_j} \Delta(|\Xi(\boldsymbol{x}) - i|) \tag{6.7}$$

Thus the entry $Conf_{i,j}$ represents the percentage of how many samples from class i have been recognized as samples from class j. It should be noted that

$$\dot{\bigcup_{j \in [p]}} U_j = U \tag{6.8}$$

and that

$$AvEr(U) = 1 - \frac{1}{p \cdot 100} \sum_{i=1}^{p} Conf_{i,i} \tag{6.9}$$

and thus

$$Av(U) = \frac{1}{p \cdot 100} \sum_{i=1}^{p} Conf_{i,i} \tag{6.10}$$

Our boost concept can now be introduced. First we do not discard any SVM obtained during the grid search. Every classifier and the corresponding confusion matrix will be aligned in an sequence to form the aforementioned committee. Secondly we need a test set to obtain the confusion matrices. All these SVMs will then be used as a new classifier in the following way. An unknown test point $\mathbf{x}$ will be presented to all aligned classifiers, each of them will then give his opinion i for the right class. The details of this voting are presented in Fig. 6.14.

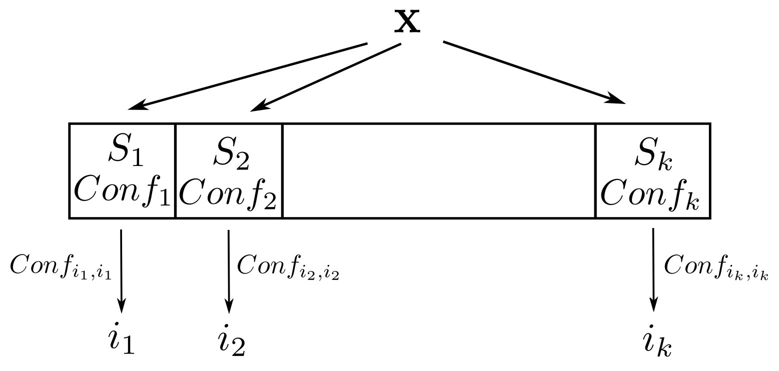


Fig. 6.14.: A committee of SVMs

Every SVM S_l will classify the point $\mathbf{x}$ and tell us its opinion i_l about the right class. Once we have all opinions, we need to decide which classifier tells the *truth*. This will be done by considering the confusion matrix of each SVM, depending on its answer to the question after the right class. The entry $Conf_{i_l,i_l}$ will now be used as a measure for the probability that l tells the *truth* (as $Conf_{i_l,i_l}$ stands for the recognition rate of l for this particular class i_l).
Yet this approach yields following problem, what if two classifiers tell the *truth* with the same probability? A decision has to be made in that case, we propose the following strategy:

- choose the classifier l with the highest average rate regarding the corresponding confusion matrix $Conf_l$

Additionally we propose another way of finding the right classifier in the committee. For this we lift the decision problem into the 2-dimensional realm. One dimension is made up of the average recognition rate $Av_l(U)$, with l being the number of the corresponding SVM, while the other dimension represents the recognition rate $Conf_{i_l,i_l}$ regarding SVM ls prediction i_l. The SVM with the largest distance to the origin $(0,0)$, i.e., the largest euclidean norm, will be chosen. Figure 6.15 visualizes this concept for 4 SVMs, i.e., $l \in [4]$. The classifier S_2 will be chosen as it has the largest distance to the origin.
We added both strategies to a template consisting of one *optimal* SVM, the corresponding results will be given in Chap. 6.
Our boosting concept has been implemented and evaluated only for SVMs, yet it can be applied to any array of classifiers (these classifiers even do not need to be of the same kind). An additional benefit of this approach is the fact, that it requires no adjustment of any involved classifier. It must be mentioned again, that this technique represents an add-on for a given template. The committee should be considered only for classes with a *low* recognition rate regarding the template. Considering our previous gedankenexperiment, the usage of this boosting technique can be indicated through the following situation. The SVM S_{opt} classifies a point $\mathbf{x}$ as belonging to class l, yet from $Conf_{opt}$ we know that this

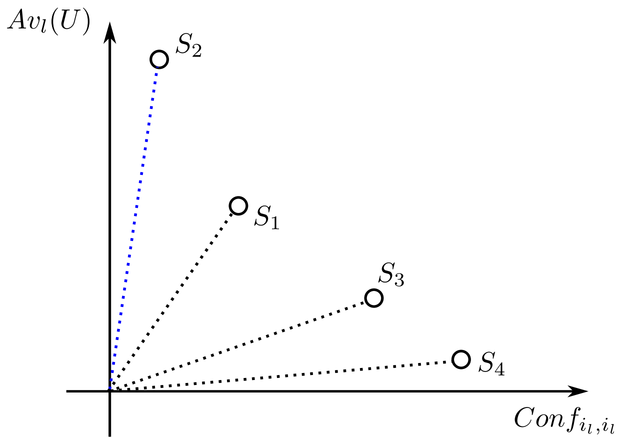


Fig. 6.15.: Two dimensional boosting

is true with a probability of 19%. It would be unwise to trust S_{opt} in this case, thus one could refer to the committee of multiple SVMs.

7. Results

This chapter presents and discusses the final results. Which have been obtained by the methods described in Chap. 6. We will first show the performance of the numerical algorithms, i.e., the QR and Jacobi algorithm. Afterwards the environment for testing will be described, i.e., the setup of power traces. The last section finally presents the recognition rates for various instruction types of our microcontroller.
Some of the plots which will be referenced in this chapter, can be found in the appendix. Only the most informative plots have been included into this chapter, i.e., comparison of recognition rates and confusion matrices.

7.1. BLAS & LAPACK Benchmark

We begin this section by listing all involved soft- and hardware. The important hardware specs are as follows

- Intel Core I7 950 3.07Ghz (4 Cores, enabled Hyper-Threading, disabled Turbo-Boost)
- 24GB DDR3 RAM (1066MHz)
- X58 Chipset
- 2x GTX470 GPU (each 1.2 GB RAM) using 16x PCIe 2.0, these cards were used exclusively for computation, i.e., the display image was provided by a separate card

The software setup can be summarized by

- Ubuntu 10.10 x64
- GSL 1.14
- Matlab 2009b
- GCC 4.1
- CUDA 3.2

In Chap. 6 it has been mentioned that only problems of a certain size should be computed on the GPU. Additionally the claim was made that BLAS3 routines are good candidates for GPU computation. Fig. A.1 verifies all these statements

by showing the computation times for different implementations of the common matrix multiplication. The upper curve shows the performance of the naive matrix multiplication (i.e. the school method). Our classification setup will handle matrices with a dimensionality up to 5250. It can be seen that the computation time would reach up to 200 seconds for a single mutliplication, if we would stick with this method. Another option would be to use Matlab, which provides a SMP implementation. The computation time would be reduced 20 times to ≈ 10 seconds. Yet the GPU implementation provided by the CUBLAS library reduces the time to at most 2 seconds. Which corresponds to a factor of 100 compared to the naive implementation. One should note that below a dimensionality of 500, the GPU implementation should be avoided.
Thus if an algorithm can be formulated in terms of BLAS3 routines, a major performance gain can be expected when execution of the appropriate segments is ported to the GPU.
Let's take a look on Fig. A.2, which shows the computation times for two implementations of the QR algorithm. The ordinary implementation provided by the GSL library needs twice the time compared to our GPU implementation. As described before, we have only ported the tridiagonalization to the GPU, which is rich in BLAS3 methods. Thus again, one can see that our implementation yields a gain only for matrices with a dimensionality above 256.
Fig. A.3 shows execution times of three implementations of the TSJM. As GSL provides no implementation of this method, a corresponding Matlab version represents the naive (host) implementation. The first observation would be that our implementation is up to 5 times faster than the ordinary Matlab version. Yet this only holds if we use the GPU memory during computation. The zero-copy version of our implementation yields a tremendous slow-down. As of this a zero-copy multi-GPU version was no longer pursued. We implemented the multi-GPU version without the use of zero-copy memory access. The synchronization between all involved GPUs has been done in the following way. After all GPUs have finished their computation, each GPU copies all its processed rows or columns to a memory block on host side. Once all GPUs have finished this process, the complete memory block is copied to every GPU. Which then can carry on with the next computation step. Theoretically the execution speed should linearly increase with each additional GPU. Yet the required synchronization yields a serious slow-down. We used two GPUs for the benchmark depicted in Fig. A.3. The aforementioned synchronization procedure slows the algorithm down to host level, i.e., all benefits of our GPU implementation disappear.

7.2. Classification Setup

The power traces were recorded with a *Lecroy SDA 735Zi* scope, at a sampling rate of 1GHz and a shunt of 27Ω. The involved microcontroller was a *PIC16F54*, which provides 25 one-cycle instruction types, each with up to two operands. Our classification experiments will use only these types. The traces for each type had been recorded in the following way. Let us take the *CLRF* instruction type as an example, only one operand is required in this case. The microcontroller executed a test series with following structure

1. CLRF(randomized_value); randomized_instruction(randomized_value);
2. CLRF(randomized_value); randomized_instruction(randomized_value);
3. ...

For each element in this series, the corresponding power trace had been recorded. We will now describe the structure of training- and test-sets. In the following experiments each set consist of 6250 traces, for each instruction type 200. Following this method, every type has the same amount of traces in our sets. These 200 traces have been chosen in the following way. Again, we shall explain it by using the *CLRF* instruction type. The corresponding 200 traces can be described by the following series

1. CLRF(randomized_value); instruction$_1$(randomized_value);

...

8. CLRF(randomized_value); instruction$_1$(randomized_value);

9. CLRF(randomized_value); instruction$_2$(randomized_value);

...

16. CLRF(randomized_value); instruction$_2$(randomized_value);

...

193. CLRF(randomized_value); instruction$_{25}$(randomized_value);

...

200. CLRF(randomized_value); instruction$_{25}$(randomized_value);

Where instruction$_i$ is an element of the enumerated 25 instructions, i.e., instruction$_{17}$ stands for instruction type 17. In that way the 200 traces have chosen for every type.

7.3. Classification Results

The sets in the following sections have been chosen as described above. The sets within each classification are disjoint, e.g., for the SVM classification the test- and training-set have no common traces.

7.3.1. Kernel k-nearest Neighbors

First experiments with sigmoid and polynomial kernels yielded very low recognition rates. Thus they have not been pursued any further. Our experiments base on Gaussian kernels. The Gram matrix K was created using a training set of 6250 traces. Afterwards this matrix was used for KPCA as described in chapter 2. Two test-sets U_1, U_2 have been created, the traces in U_1 were projected onto the principal components of K. While the elements in U_2 were used as unknown traces in our classification task. Figure A.4 shows the best average recognition rate. This result was obtained by projecting onto the first 5000 principal components and considering only 1 neighbor. One can see that the recognition rate peaks at 46.4% for $\sigma = 2^{-3}$. The experiment was repeated with the same sets for the KPCA using the TSJM.
No differences between both algorithms have been found. In other words, the TSJM yielded no benefit during the principal component analysis. The confusion matrices are also identical, thus we only list one of them in table 7.1

Table 7.1.: Confusion matrix for kernel k-nearest neighbors

	addwf	andlw	andwf	bcf	bsf	btfss	clrf	clrw	comf	decf	decfsz	incf
addwf	22	0	2	1	5	6	3	0,5	0	2	2,5	2,5
andlw	2	59,5	1,5	0	6,5	0,5	0	0	0	0,5	1	10,5
andwf	3,5	1	51	17,5	1,5	4,5	5,5	0	1	1	0	2,5
bcf	1,5	0	10	68	0,5	0,5	0	0	0	2,5	1,5	0
bsf	2	2	0,5	0,5	48	1	2,5	0,5	0	0,5	0	4
btfss	3	0	3,5	1	0,5	71	0	0	0	2	5	1,5
clrf	8	1	1	0	1,5	0,5	41	0,5	2	1	0	1,5
clrw	0	0	1	0,5	0	0	2	84,5	0,5	0,5	0	0
comf	0	0	3,5	0	0	0	4,5	0,5	68,5	17,5	3	1,5
decf	2	1	2,5	2,5	0,5	6	0	0	10,5	43,5	21	2,5
decfsz	1	1,5	1,5	1,5	0	14	0,5	0	6,5	22,5	40	2
incf	2	5,5	4,5	0	13	3,5	4,5	0,5	2	1	2,5	27,5
incfsz	2	5,5	2	0	9,5	2	2	0	1	2	3	16
iorlw	1	0,5	1,5	0,5	3	1	1	0	0	0	0	0
iorwf	6	0	3,5	3,5	1,5	0,5	2,5	2,5	0	0,5	0	1
movf	2,5	0,5	1,5	1	1,5	0	1,5	1	0	0	0	0,5
movlw	1	0	0	0	1	0	1	0	0	0	0	0
movwf	6,5	2	1,5	0,5	1,5	0	18	3,5	1	0	0	6
nop	0	0	0	0	0	0	0	7,5	0	0	0	0
rlf	6,5	3	0	0	4,5	0	1	1,5	1	0	0	1,5
rrf	4	5	0	0	0,5	0	2	2	1,5	1	0,5	3
subwf	1	0	0	7,5	0	0,5	0	0	0	4,5	6	0
swapf	4,5	3,5	0	0	2,5	0	0,5	1,5	2,5	0	1	0,5
xorlw	2	0,5	0,5	0	5,5	0	0	0	0	0	0	0,5
xorwf	10,5	0	0,5	2	2	2	3,5	1,5	0	0	0	1,5

	incfsz	iorlw	iorwf	movf	movlw	movwf	nop	rlf	rrf	subwf	swapf	xorlw	xorwf
addwf	3	4	7,5	3,5	0,5	2,5	0	9	5,5	0,5	6,5	2	9
andlw	6	0,5	0,5	0	0,5	2,5	0	1,5	3	0	2,5	0,5	0,5
andwf	0,5	1,5	2	1,5	1,5	0,5	0	1	0,5	0	0,5	0	1,5
bcf	0	0,5	4,5	2,5	0,5	0	0	0	0	6	0	0	1,5
bsf	13	6	2	1	3	0	0	2,5	0,5	0	0	7	3,5
btfss	0,5	2,5	1	1	0,5	0	0	1,5	0	1	1	0	3,5
clrf	0,5	0,5	0,5	4	0	16,5	0	2,5	8	0	1,5	0	8
clrw	0	0	1,5	1	0	5	0	0	0,5	0	0,5	0	2,5
comf	0,5	0	0	0	0	0	0	0,5	0	0	0	0	0
decf	1	0	2	0,5	0	0	0	0	1	2,5	0	0	1
decfsz	0,5	0	3	1,5	0	0	0	0,5	0	3	0	0	0,5
incf	12,5	1	1	1,5	0,5	4,5	0	0,5	2	0	3	1	6
incfsz	31	3,5	1	0,5	0	1	0	4,5	3	0	4	5,5	1
iorlw	2,5	44,5	3	2	17	0,5	0,5	1,5	0,5	0	1,5	15,5	2,5
iorwf	0,5	2,5	27,5	18,5	0,5	3	0	2,5	4	4	3	0,5	12
movf	0	4	16	31,5	7,5	3	0	3	9,5	0,5	6,5	0	8,5
movlw	0	16	1,5	4,5	64	0,5	1	1	3	0	1	3,5	1
movwf	0,5	1	5	2,5	0,5	30,5	0	3	8	0	3,5	0	5
nop	0	0	0	0	0	0	92,5	0	0	0	0	0	0
rlf	2,5	1	1	4	1,5	1,5	0	20,5	20	0	22	2	5
rrf	3,5	0,5	2,5	2,5	0,5	3,5	0	21	19,5	0	17,5	0,5	9
subwf	0	0	4,5	0	0	0	0	0	0	74,5	0	0	1,5
swapf	4,5	0,5	2,5	3,5	1,5	1,5	0	17	17,5	0,5	26	1	7,5
xorlw	5	21	1	1	1,5	0	0,5	1,5	1,5	0	1,5	56,5	0
xorwf	1	2,5	12	10,5	3	1,5	0	10,5	9	1	7,5	0	18

7.3.2. Kernel Linear Discriminant Analysis

Also for the kernelized version of LDA, a training set was created which was used for KPCA. The LDA utilized a training- and test-set U_1, U_2. respectively. The traces in U_1 have been projected onto the principal components and used for training within the LDA. The data in U_2 was again used as unknown traces. Fig. A.5 shows the best results, which have been obtained by the projection onto the first 500 principal components.
The highest average recognition rate of 57.1% was obtained for $\sigma = 2^{-3}$. The confusion matrix for these parameters is listed in Table 7.2. Studying this table, one can see that the *NOP* instruction type can be recognized with a probability of 100%. Additionally the *CLRW* instruction type can be indentified with a success rate of 97%.

Table 7.2.: Confusion matrix for kernel LDA

	addwf	andlw	andwf	bcf	bsf	btfss	clrf	clrw	comf	decf	decfsz	incf
addwf	23,5	0	1,5	2,5	8	1	2,5	0	0	0,5	2	4
andlw	0	77,5	2	0	5,5	0	0,5	0	0	0	0,5	2,5
andwf	4,5	0	45,5	26	0	0,5	3	0	0,5	1	1	4,5
bcf	1,5	0	20	60,5	1	0,5	0	0	0	2,5	1,5	0,5
bsf	7	1	0,5	0,5	42	0	1,5	0	0	0	0	4,5
btfss	5	0	2,5	2,5	0,5	72	0	0	0	0	1	2
clrf	6	0	1	0	2	0	41,5	0	2,5	0	0	4
clrw	0	0	0	0	0	0	0,5	97,5	0	0	0	0
comf	1	0,5	3	0	0	0	8,5	0	57	17	7,5	2,5
decf	0	0	1,5	0	0	0,5	0	0	9,5	55,5	26	3
decfsz	0,5	0,5	0	1	0	2	0	0	7	21,5	62,5	3
incf	4,5	4	1	0	12	1	3,5	0	2,5	0	1,5	34,5
incfsz	4	3,5	0,5	0	14	1	1	0	0	0	1,5	14
iorlw	2,5	0	1	1,5	2	0,5	0	0	0	0	0	0
iorwf	10	0	3	4	0,5	0	2	1	0	0	1	2
movf	4,5	0	4,5	0,5	0	0	4	0	0	0	0	1
movlw	0	0	0	0	0,5	0	0,5	0	0	0	0	0
movwf	2,5	1	1,5	0	0,5	0	20,5	1,5	0,5	0	0	1,5
nop	0	0	0	0	0	0	0	0	0	0	0	0
rlf	7,5	2	1	0	6,5	0	0	0	1	0	0	2,5
rrf	5	0,5	0,5	0	2,5	0	1,5	0	0,5	0	0	5
subwf	0	0	0	1	0	0	0	0	0	7,5	4,5	0
swapf	3	2,5	0	0	2	0	1	0	0,5	0,5	0,5	1,5
xorlw	0,5	0	0	0	1,5	0	0	0	0	0	0	0
xorwf	9,5	0	1	3	0	0,5	3,5	0	0	0	0	2,5

	incfsz	iorlw	iorwf	movf	movlw	movwf	nop	rlf	rrf	subwf	swapf	xorlw	xorwf
addwf	1,5	1,5	7,5	3	0	0	0	11,5	5,5	0	7	0,5	16,5
andlw	1	1,5	1	0,5	0	1,5	0	0,5	1	0	2	0	2,5
andwf	0	0,5	6,5	0,5	0	3	0	1	0	0	0	0	2
bcf	0,5	0	4,5	3	0	0,5	0	0,5	0	0	0,5	0	2,5
bsf	11	4	2,5	2,5	0	1	0	2	2,5	0	4,5	5	8
btfss	0,5	3	1,5	1	1	0	0	1	1	0	1	0,5	4
clrf	0	0,5	5	3	0	20,5	0	5,5	4,5	0	0	0	4
clrw	0	0	0	0,5	0	1,5	0	0	0	0	0	0	0
comf	0	0	1	0	0	1,5	0	0	0	0	0	0	0,5
decf	0	0	1	0,5	0	0	0	0	0,5	1,5	0,5	0	0
decfsz	1,5	0	0	0	0	0	0	0	0	0,5	0	0	0
incf	12,5	0	5	1,5	1	3	0	1	2,5	0	1	0,5	7,5
incfsz	43,5	2,5	2,5	1	0	0	0	0,5	1,5	0	3,5	3	2,5
iorlw	1,5	52	1	0,5	13,5	0	0	1	1	0	1	20,5	0,5
iorwf	0,5	0	28,5	25	0,5	5	0	1,5	1,5	1	2	0	11
movf	0,5	1	16,5	40,5	1,5	6	0	6,5	2	0,5	2,5	0	8
movlw	0	18	0	0,5	71	0,5	0	2	3	0	1	1	2
movwf	0,5	0	7	1	0	48,5	0	2,5	3,5	0	1	0	6,5
nop	0	0	0	0	0	0	100	0	0	0	0	0	0
rlf	3,5	0	3,5	4	0,5	0,5	0	27	16	0	15	0	9,5
rrf	3	1	3	2,5	0	3	0	21,5	24	0	16	0	10,5
subwf	0	0	0	0	0	0	0	0	0	87	0	0	0
swapf	5,5	0,5	1	2	0,5	0	0	19,5	18	0	32	2	7,5
xorlw	2,5	17,5	0	0	2,5	0	0	1	1,5	0	1	72	0
xorwf	0	0,5	11	13	0,5	3,5	0	14,5	10,5	0	7	0	19,5

7.3.3. Support Vector Machines

The SVM approach only involves two sets, a training-set T and a test-set U. Also in this case Gaussian kernels yielded the highest average recognition rates. Polynomial kernels peaked around 14% and sigmoid kernels at 4%. Thus a grid search was executed for C and σ, the results are depicted in Fig. A.6. The maximal recognition rate was further increased to 64.1% with the use of $\sigma = 2^5$ and $C = 2^4$. This result was also the motivation for developing and applying the boosting technique onto the SVM approach. A look on the corresponding confusion matrix in Table 7.3, shows that one can now classify the *CLRW* instruction type with a probybility of 100%. Compared to the LDA approach, the rate for the *NOP* command was slightly reduced to 99.2%. Yet this is a negligible descrease. Allthough the average rate peaks at only 64.1%, the corresponding SVM enables us to classify 4 instruction types with at least 90% certainty.

Table 7.3.: Results for a SVM with a Gaussian kernel

	addwf	andlw	andwf	bcf	bsf	btfss	clrf	clrw	comf	decf	decfsz	incf
addwf	44,4	0	1,6	2,4	2,4	1,6	0,4	0,8	0	0	4,8	2,8
andlw	0	92	0,8	0	0,8	0	0	0	0	0	0	0,4
andwf	3,2	0,4	73,2	7,2	0	0,4	1,2	0	3,2	1,2	0,4	4
bcf	0,4	0	4,8	85,2	0	1,2	0	0	0	2,8	2	0
bsf	2,4	1,6	0	1,2	68	0	0	0	0	0	0	3,6
btfss	3,6	0	2,4	2,4	0,4	82	0	0	0	1,6	0,4	2,4
clrf	5,2	0	1,6	0	0,8	0	48,8	0,4	1,6	0	0	1,6
clrw	0	0	0	0	0	0	0	100	0	0	0	0
comf	0	0	2,4	0	0	0	6,4	0	75,6	9,2	2,8	0
decf	0	0,4	1,2	0	0	0,8	0	0	20,8	56,4	16,4	1,2
decfsz	1,6	0,4	1,2	0	0	3,2	0	0	8,4	8,8	69,2	4
incf	2	6	4,8	0	8,8	0	3,2	0	2,8	1,6	2	47,6
incfsz	2,8	2,8	0	0	16	0,4	0,8	0	1,2	0	2,4	7,2
iorlw	6,8	0	0,4	0,8	0	1,6	0	0	0	0	0	0
iorwf	8	0	2,4	4	0	0	2,4	0,8	0	0,8	2,4	0,4
movf	4	0	2,4	1,6	0	0	2,4	1,2	0	0	0,8	0
movlw	0,8	0	0	0,4	0	1,6	0	0	0	0	0	0
movwf	3,2	1,2	3,2	0	0,8	0	21,2	3,2	0,4	0	0	0,4
nop	0	0	0	0	0	0	0	0,8	0	0	0	0
rlf	8,8	2,4	0	0	2	0,4	0	0	0,8	0,4	0,8	1,6
rrf	5,2	3,6	0	0	1,2	0	0,4	1,2	1,6	0,4	0,4	4,4
subwf	0	0	0	2,8	0	0	0	0	0	4,8	0,8	0
swapf	8	2	0	0	1,6	0	0	0	1,6	0,4	0,8	1,6
xorlw	2	0	0	0	1,2	0	0	0	0	0	0	0
xorwf	17,2	0	2	2	0,8	0,4	1,6	1,2	0	0,8	1,6	2

	incfsz	iorlw	iorwf	movf	movlw	movwf	nop	rlf	rrf	subwf	swapf	xorlw	xorwf
addwf	1,2	2,4	4	2,4	0	0,8	0	6,4	4,4	0,4	4	0,4	12,4
andlw	1,2	0	0	0,8	0,4	1,2	0	0,4	1,2	0	0	0	0,8
andwf	0	0	2,4	0,8	0	2	0	0	0	0	0	0	0,4
bcf	0	0	0,8	0,4	0	0	0	0	0	2,4	0	0	0
bsf	8,4	3,2	1,6	3,2	0	1,2	0	0,8	0	0	1,2	3,2	0,4
btfss	1,2	0,8	1,2	0,4	0,4	0	0	0	0	0	0	0,4	0,4
clrf	0	0	0	6,8	0	23,6	0	0,8	4,8	0	0,4	0	3,6
clrw	0	0	0	0	0	0	0	0	0	0	0	0	0
comf	0	0	0	0	0	3,6	0	0	0	0	0	0	0
decf	0	0	0,8	0	0	0	0	0	0	1,6	0	0	0,4
decfsz	0	0	0,8	0	0	0	0	0,4	0	2	0	0	0
incf	13,6	0	0	0	0	2	0	0,8	2	0	0,4	0	2,4
incfsz	57,2	2	0	0	0	0	0	2	0,8	0	0	3,2	1,2
iorlw	2	80,4	0	0	6,4	0	0	0,8	0,4	0	0,4	0	0
iorwf	0	2	36	20	1,6	2,8	0	1,6	4	2	2,8	0	6
movf	0	1,2	18,8	47,2	2,4	1,2	0	2	7,2	0,8	2	0	4,8
movlw	0	8,8	0	1,2	85,6	0	0	0,4	0,4	0	0,4	0	0,4
movwf	0	0	0,8	8	0	54,4	0	0,4	0,8	0	0	0	2
nop	0	0	0	0	0	0	99,2	0	0	0	0	0	0
rlf	2,4	0	0,4	3,2	0,4	0,4	0	28,8	22,8	0	22	0	2,4
rrf	2,4	0	0,4	5,2	0,4	0	0	18,4	33,2	0	16	0	5,6
subwf	0	0	0	0	0	0	0	0	0	91,6	0	0	0
swapf	5,2	0,4	0,8	1,6	0,8	0	0	20,8	14,8	0	38,4	0	1,2
xorlw	2,4	1,2	0	0	0	0	0	1,6	0,4	0	2,4	88,8	0
xorwf	0	0,4	8	9,6	2	2	0	8,4	17,2	0,4	3,2	0	19,2

7.3.4. Comparison

We conclude this chapter with a comparison of all three classifiers. Fig. 7.1 shows the average recognition rates of the previously mentioned KKNN, LDA and SVM classifier. The common parameter, on the x-axis, is the exponent of σ the *sigma range* during the grid search. In other words, the grid search sampled the range $[2^{-4}, 2^{10}]$ of σ via $2^{-4}, 2^{-3}, 2^{-2}, ..., 2^{10}$. The x-axis shows the corresponding exponent of the base 2.
One can see that the KKNN and KLDA results are very similar, but only for higher values of σ. For lower values, the KLDA shows improvements of up to 10%. This indicates that the traces could be normally distributed in the feature space.
Regarding the best average recognition rate, the SVM approach shows the best results. As it improves the highest KLDA rate by nearly 10%.

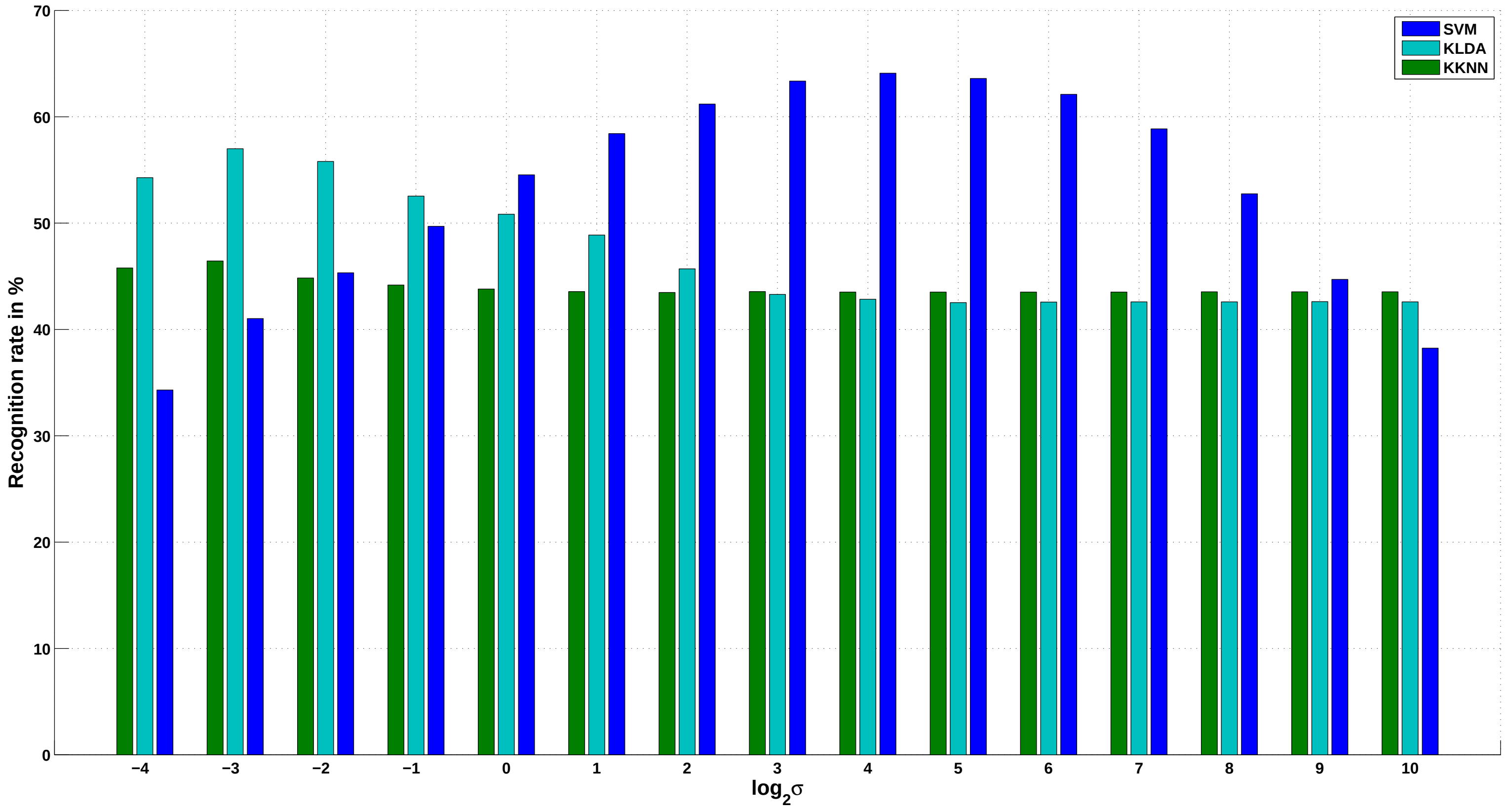


Fig. 7.1.: Comparison of the recognition rates for the classifiers used in this thesis. Only the best classifiers are depicted in this comparison.

7.3.5. Boosting

We will now take a look on the results after applying the boosting method onto the SVMs from the previous section. Fig. A.7 shows the average recognition rate of a SVM committee with the one dimensional decision process. By studying Fig. A.7 one can see that the success rate is dramatically reduced for several instructions, e.g., the committee has a chance of 0% to recognize the *ADDWF* instruction type. Yet the chance for the *DECFSZ* is increased by more than 20% from 69.2% to 95%. This enables us to find another instruction type with almost perfect certainty.
Fig. A.8 shows the boosting results for the 2-dimensional decision process. This method boosts the rate for the *BCF* type from 85.2% to 91%. Regarding the 'COMF' type, the rate has been increased from 75.6% to 90%. Additionally several other types are boosted by a small amount. Thus the *answers* of the best SVM, i.e., the one presented in the previous section, should not be considered for the instruction types *BCF* and *COMF*. Instead the committee should be asked for its opinion. Either it verifies the SVMs answer or it falsifies it. Additionally classification should be carried out in parallel by the committee and the best SVM. This approach enables one to classify 8 instruction types with at least 90% success rate.

8. Conclusion

The goal of this thesis was to evaluate the usage of kernel-based methods for the classification of power traces. A *good* classification method/result would give rise to a variety of side channel attacks, e.g., a disassembler for recorded (i.e. unknown) power traces. It was uncertain if kernel methods would outperform the first results of [Weg09], who used a Markov-chain approach. Additionally the usage of kernel methods always yields a high computational effort. Thus two major problems had been addressed with this thesis: finding a suitable kernel-based method for classification and an efficient implementation.
As this thesis represents the first approach on this field, i.e., kernel methods for power trace classification, there was no last point to start from. Thus the decision was made to start with the most simple methods available, implement them as efficient as possible (with respect to the available time) and evaluate their results. The first method of choice was, as always in applied statistical learning theory, the principal component analysis. Yet in our context its main purpose was not the dimensionality reduction, it was the extraction of good features. In other words, the creation of a better description for power traces. We have shown that a kernelized PCA (i.e. KPCA) in combination with LDA, yields results comparable to the Markov-chain approach.
There was also no prior knowledge about the appropriate kernel choice. Thus we evaluated the most popular types of kernel functions. Gaussian kernels have yielded superior results compared to other kernel types, i.e., sigmoid or polynomial. The results from KNN classification suggest that the KNN approach should not be considered for power trace classification at all. Additionally it has been shown that the assumption of a Gaussian distribution in feature space is appropriate.
Before looking on support vector machines we shall discuss the efficiency of our implementations. All implementations showed a significant benefit through the use of a GPU. Yet it has been shown, that the use of the GPU should be restricted to problems above a certain size. Our hope that the TSJM would yield better recognition rates, as it has better numerical properties, was disproved. In general, the QR algorithm has shown to be considerably faster than the TSJM. We have shown that its speed can be doubled by the use of a hybrid algorithm. Both algorithms address the SEVP within KPCA, yet there are other parts of KPCA which also benefit greatly from the use of a GPU. The projection onto the principal components and the creation, as well as the centering of the Gram

matrix, have shown significant improvements regarding their execution speed. The GPU port of the KNN algorithm also showed a major increase in execution speed. Additionally the LDA could be further accelerated. Its speed could be further improved by porting not the matrix inversion to the GPU, but by carrying out the actual classification in parallel.
Support vector machines represent one of the most powerful methods on the field of kernel-based methods. Yet they also involve a high computational complexity. We have evaluated the use of a specific SVM type, the OvO SVMs. Due to reasons of time, considering the algorithmic complexity of a GPU port, we have used an existing implementation. Yet the algorithms behind the SVM classification contain segments which are good candidates for a GPU port.
The use of SVMs yielded further improved results, outperforming even the Markov-chain approach in terms of average recognition rates. Motivated by these results and the concept of the grid search, a new boosting technique was developed. The main goal of our boosting technique was not to increase the average recognition rate, but to increase the recognition rate for single instruction types. The computational effort of our boosting technique is pretty much non-existing, as we only need to keep the SVMs from a concluded grid search. Yet the results have shown that this approach yields satisfying results, i.e., boosting a small amount of instructions beyond the 90% barrier.
The perfect classifier would yield an average recognition rate of 100%, i.e., 100% for each class. The results of this thesis indicate that this goal can not be achieved by a single kernel-based classifier, e.g., a single SVM or LDA. A side channel attack on a microcontroller requires a template, which is nothing more than a setup of trained classifiers. These classifiers do not need to be of the same kind. Our results have shown that a SVM in combination with the aforementioned boosting methods can yield a major benefit for a side channel attack. As it provides us with more instruction types which we can recognize with nearly perfect certainty, i.e., above 90%.
Thus one may speculate, with good reasons, that techniques like the ADA-Boost could further increase the amount of recognizable instruction types. A choice of different kernel functions may also improve the results, as well as a different setup of classifiers, i.e., a different template.
We would like to end our conclusion with a sentence from [SS02] motivated by the work of [DW96], which fits the problems encountered during this thesis pretty well.

> ...there is no free lunch in learning and there is no free lunch in kernel choice...

A. Appendix

A.1. Benchmark Results

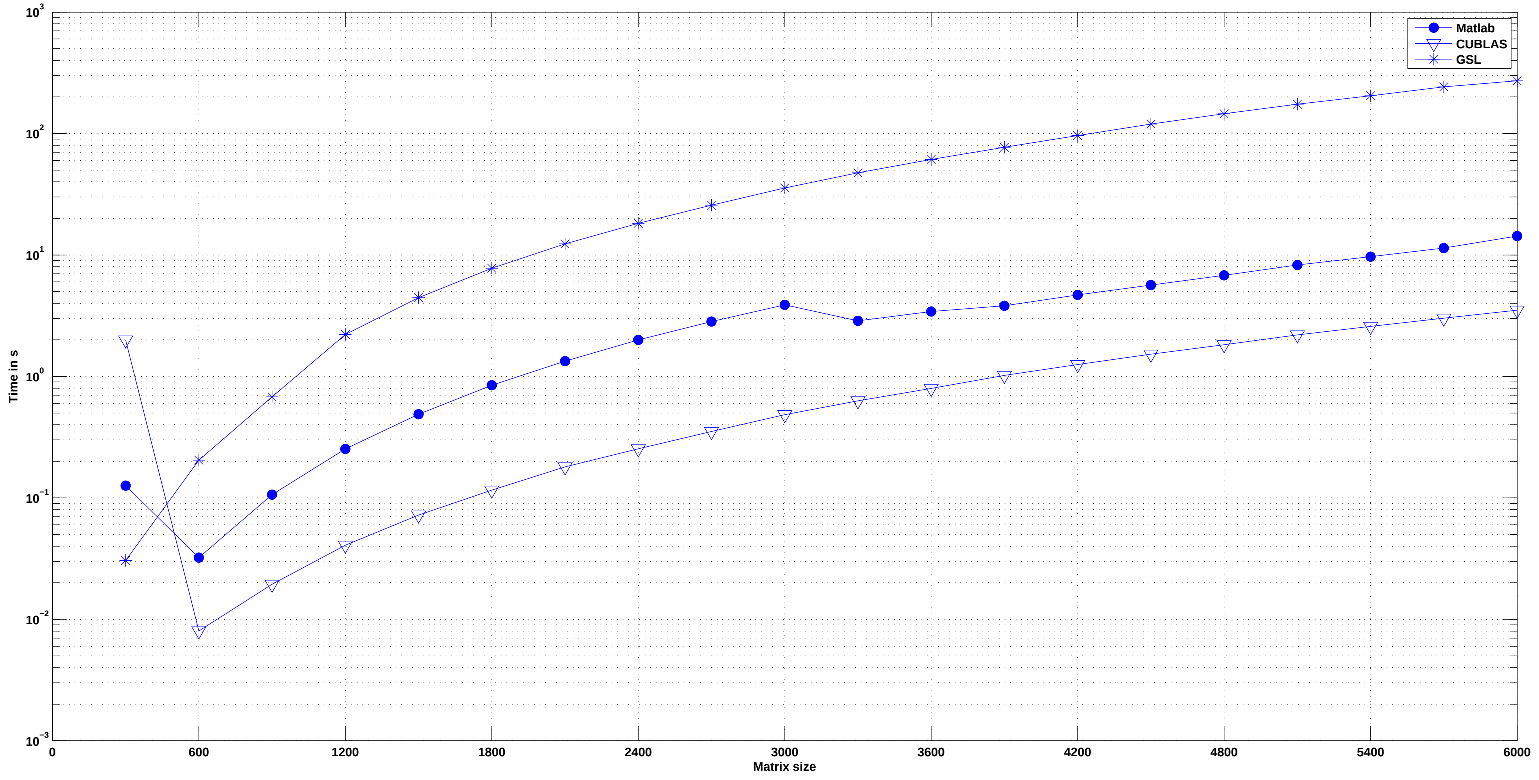


Fig. A.1.: Comparison of matrix multiplication speed on GPU and host

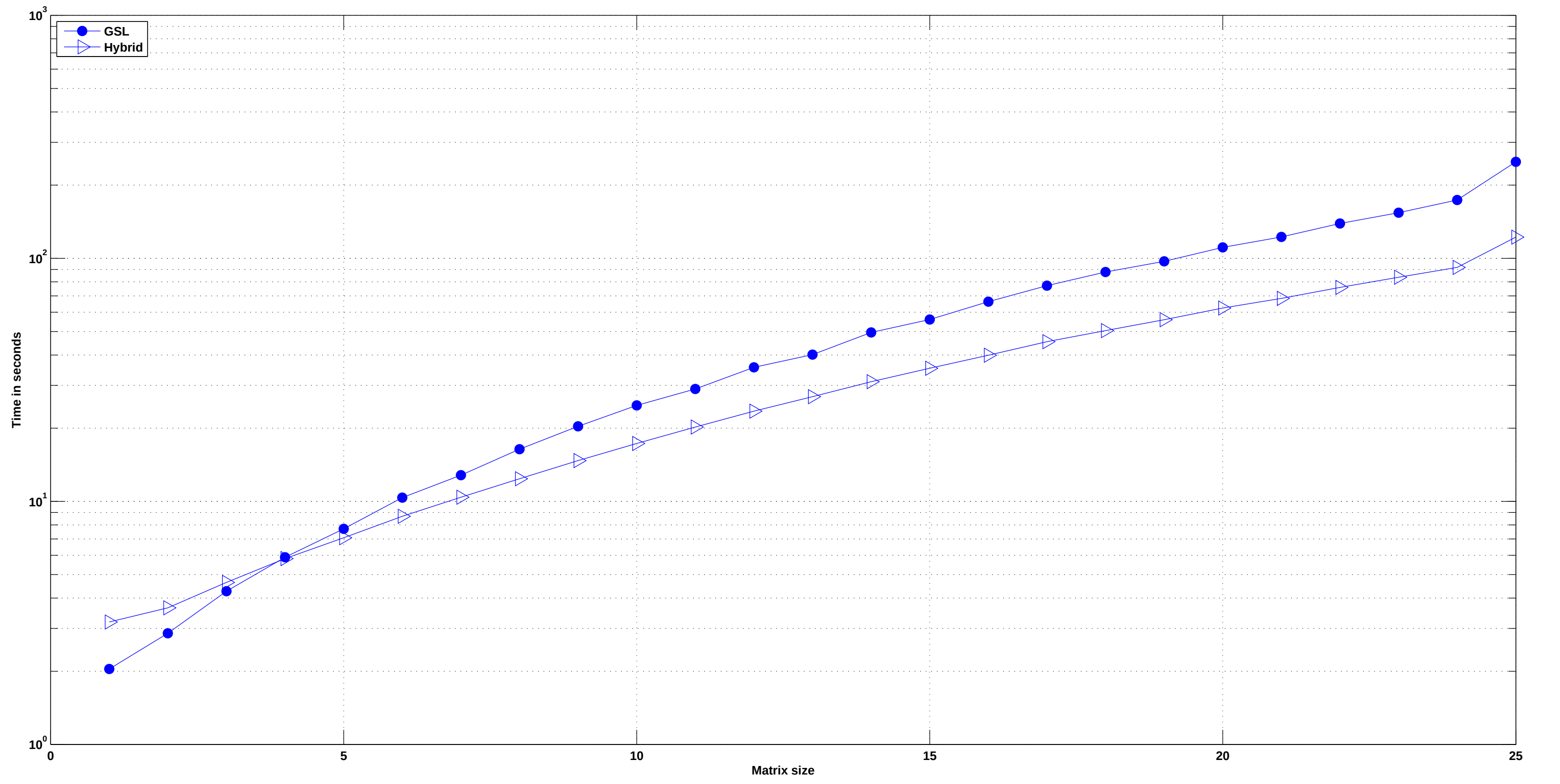


Fig. A.2.: Comparison of the QR algorithm on GPU and host

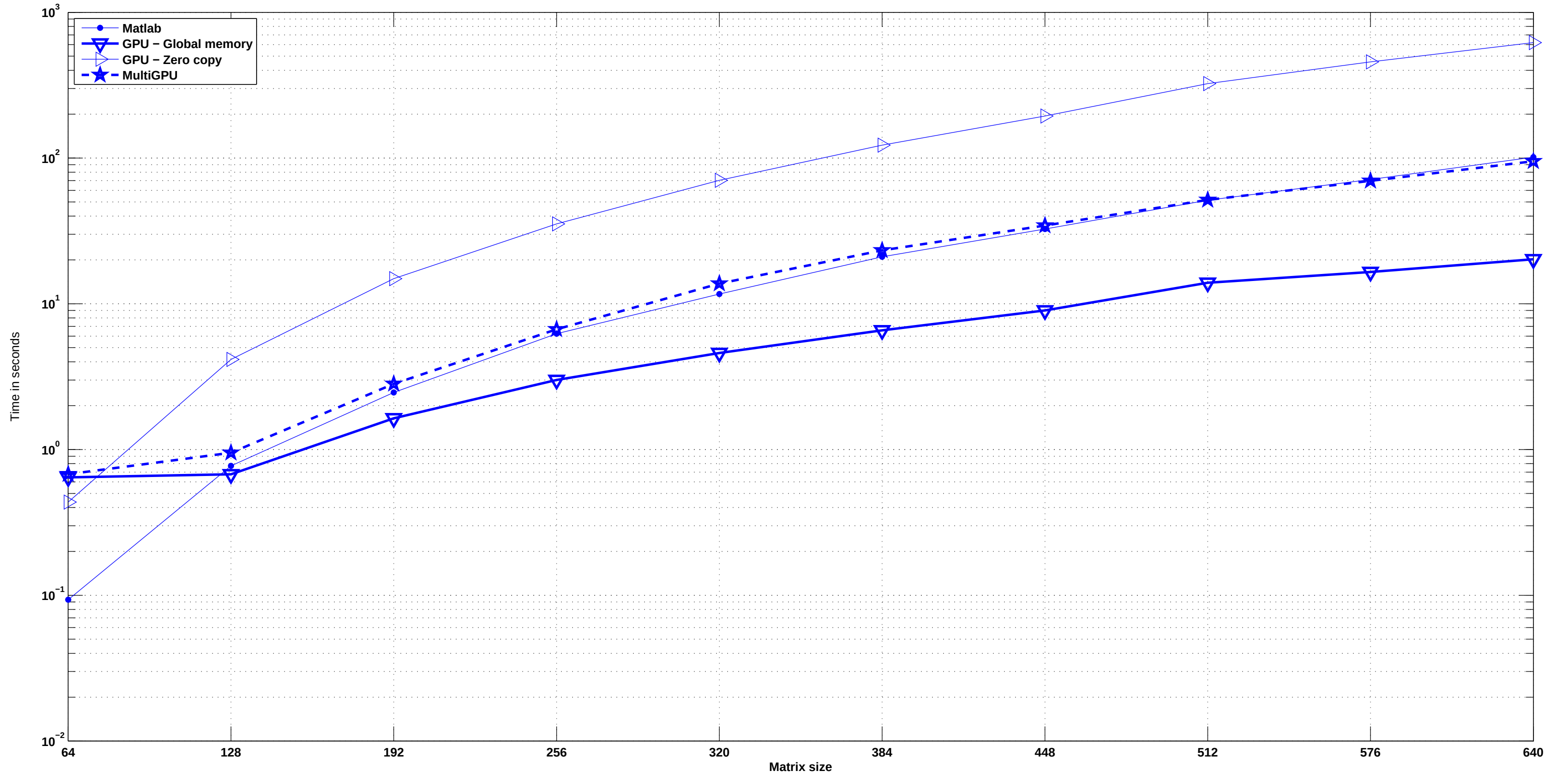


Fig. A.3.: Comparison of TSJM on GPU and host

A.2. Classification Results

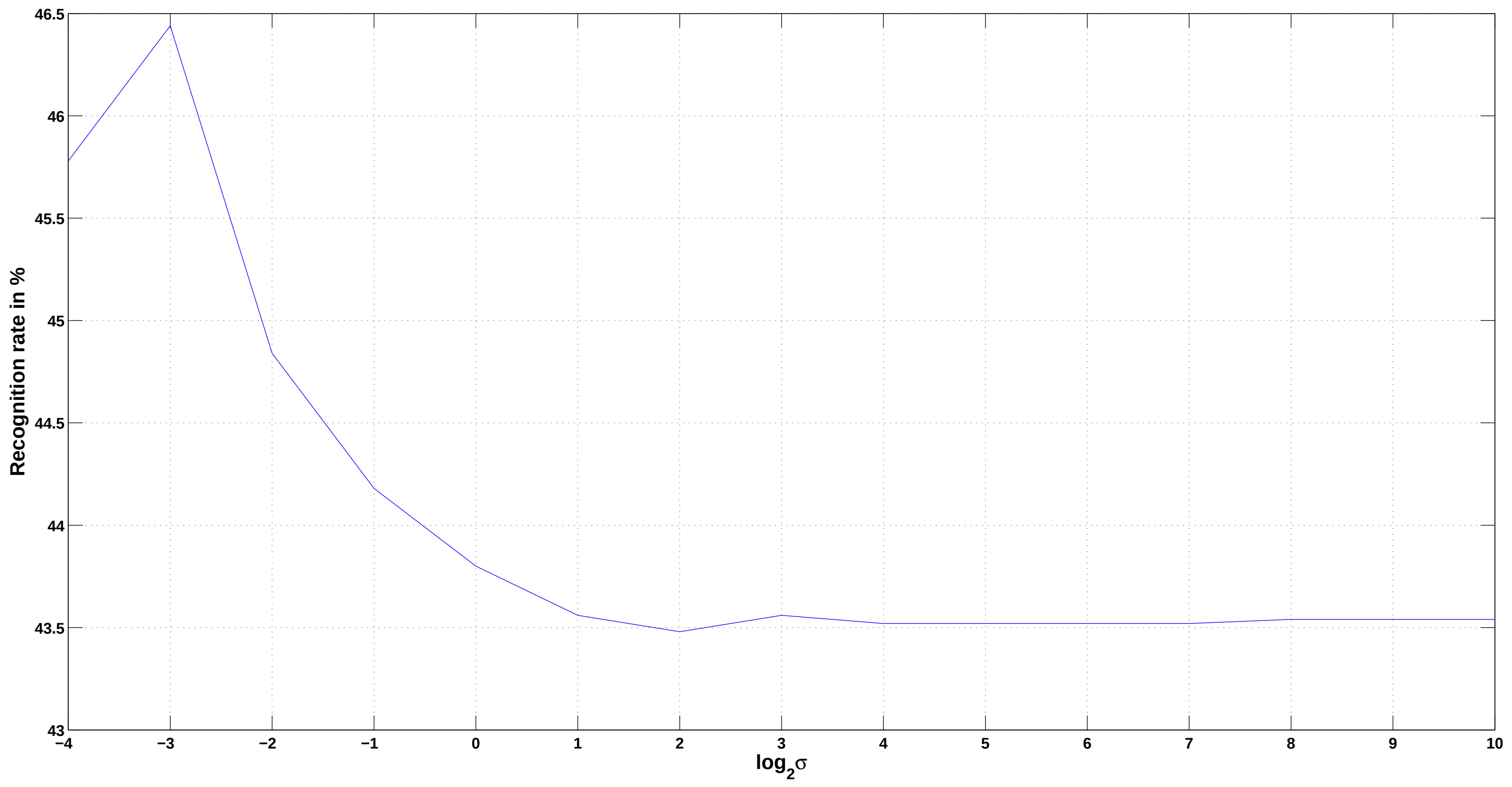


Fig. A.4.: Recognition rate using the kernel k-nearest neighbors classifier, with $k = 1$

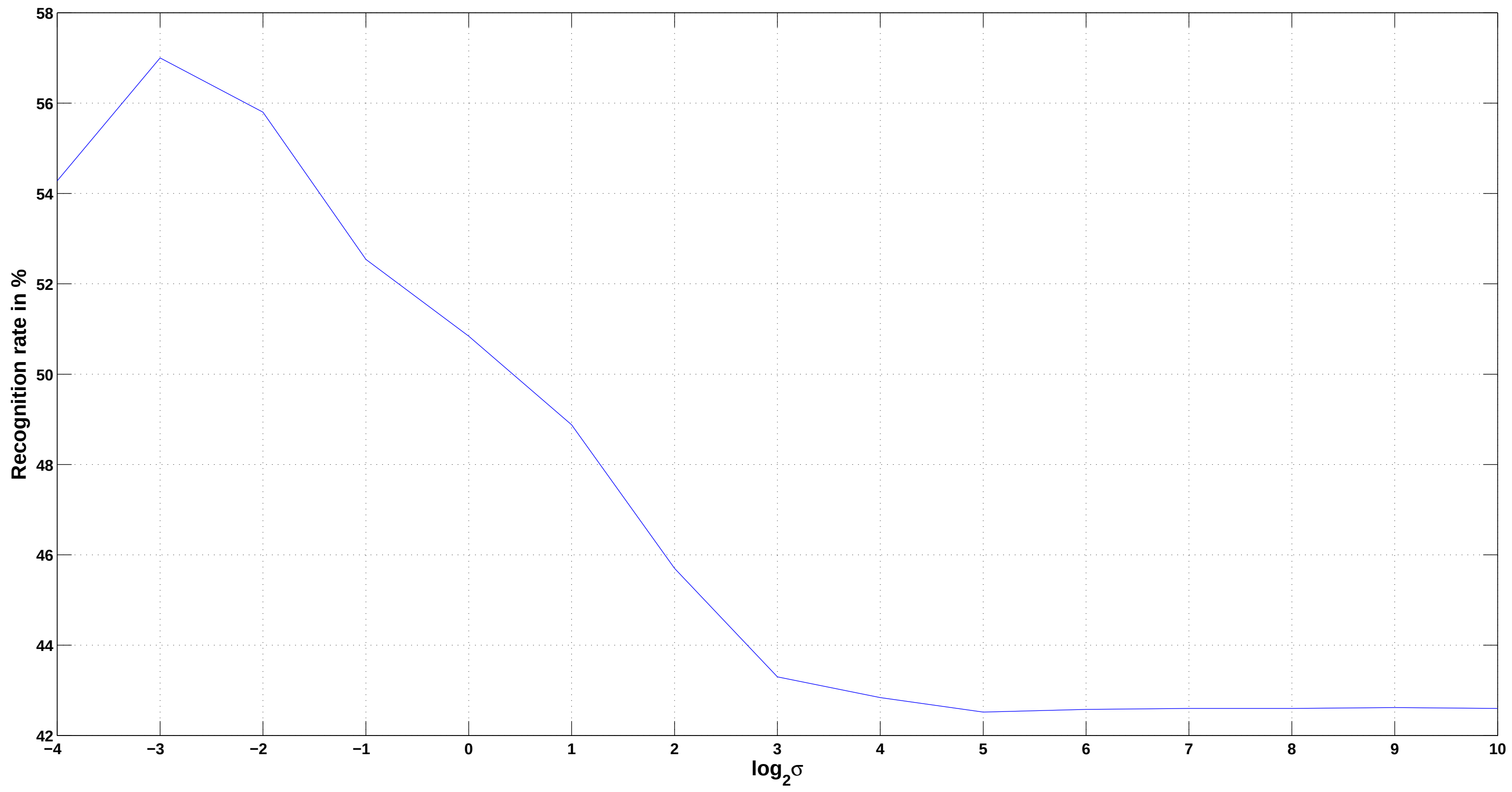


Fig. A.5.: Recognition rate using the kernel linear discriminant analysis. This recognition rate has been obtained via a projection onto the first 500 principal components.

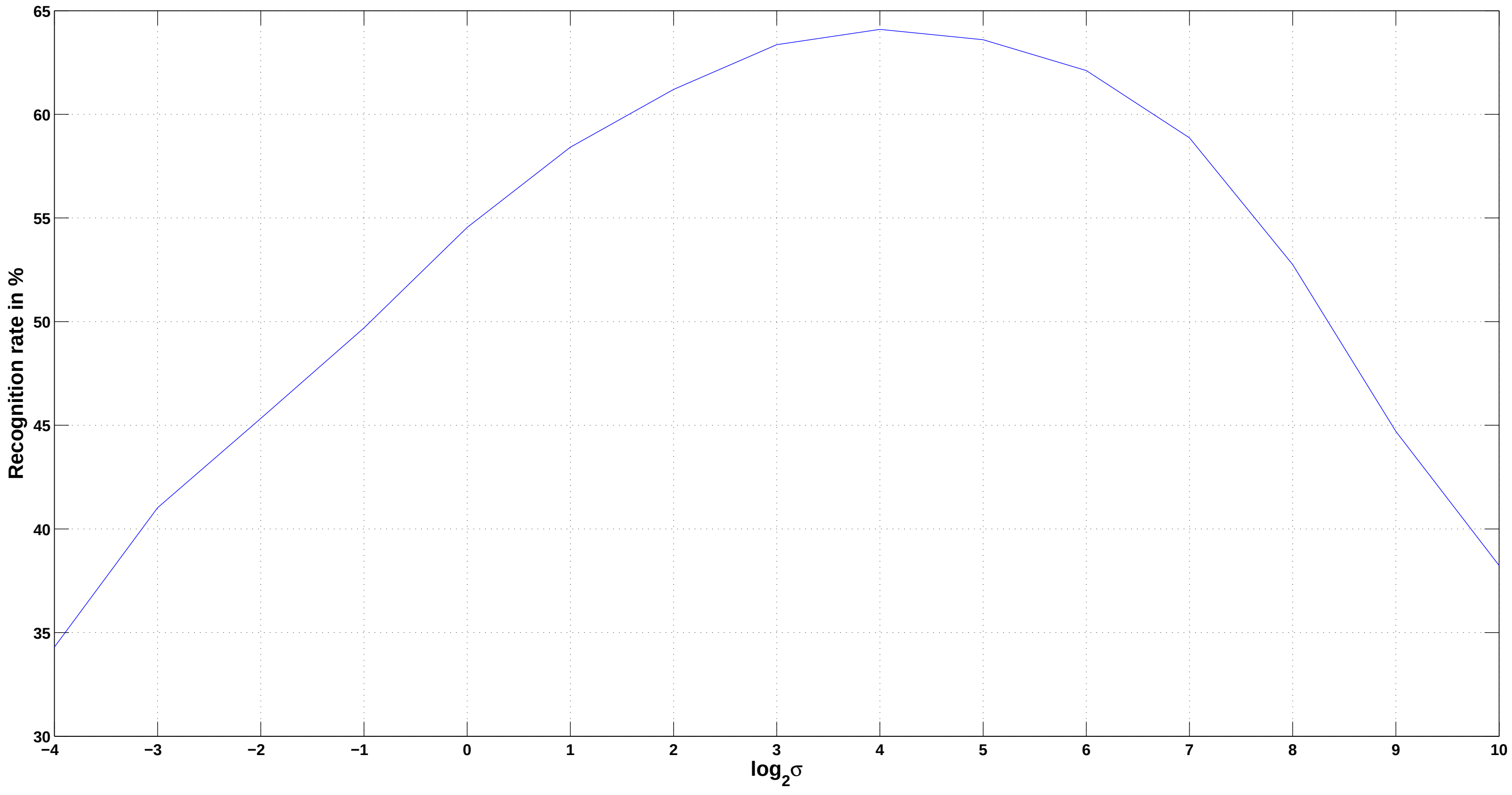


Fig. A.6.: Recognition rate for a support vector machine. This result has been obtained with $C = 2^4$

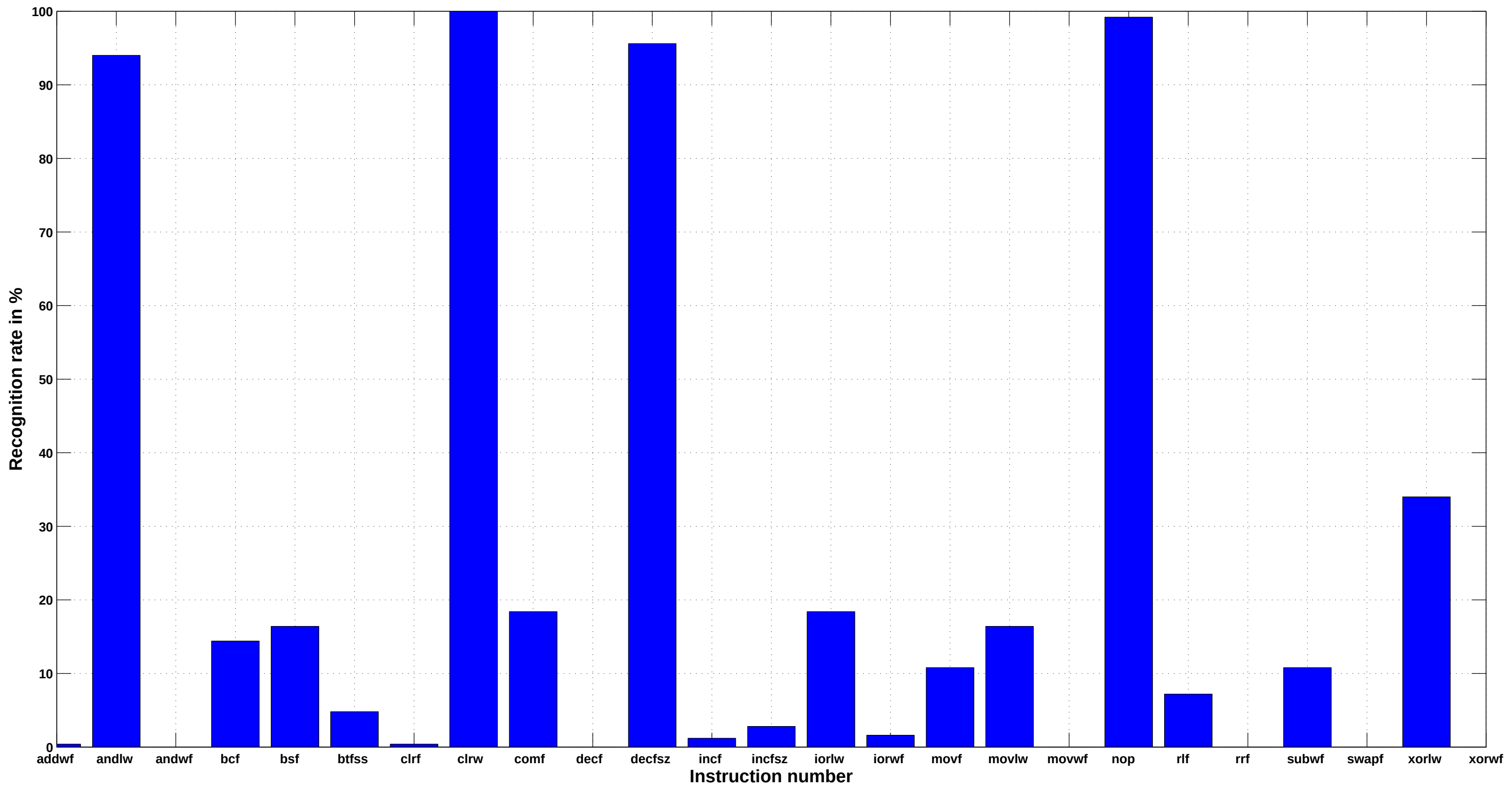


Fig. A.7.: Results of a one dimensional boosting

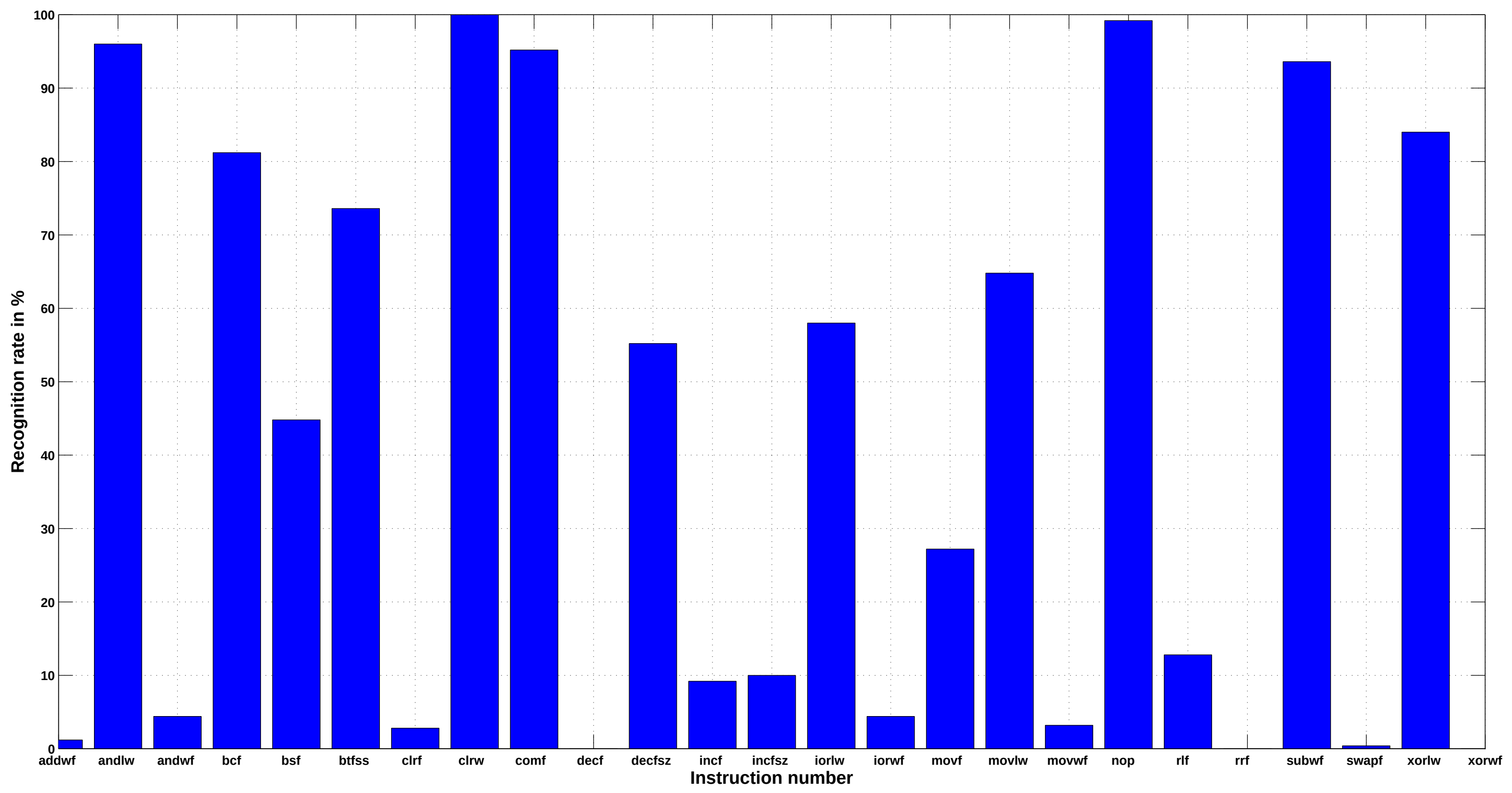


Fig. A.8.: Results of a two dimensional boosting

List of Figures

List of Tables

Index

A. Bibliography

[aAFF05] Jian Yang 'and' Alejandro F. Frangi. Kpca plus lda: A complete kernel fisher discriminant framework for feature extraction and recognition. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, 27:230–245, 2005.

[aCJL11] Chih-Chung Chang 'and' Chih-Jen Lin. Libsvm: a library for support vector machines. Technical report, 2011.

[aEK10] Jason Sanders 'and' Edward Kandrot. *CUDA by example*. Addison Wesley, 2010.

[aHS88] A. Kielbasunski 'and' H. Schwetlick. *Numerische lineare Algebra*. VEB, 1988.

[aWmH10] David Kirk 'and' Wen-mei Hwu. *Programming Massively Parallel Processors*. Morgan Kaufmann, 2010.

[Bis06] Christopher M. Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[DW96] W. Macready D. Wolpert. No free lunch theorems for optimization. Technical report, 1996.

[GHG96] Charles F. van Loan Gene H. Golub. *Matrix Computations*. Johns Hopkins, 1996.

[GS10] Mary W. Hall Gagandeep Sachdev, Vishay Vanjani. Takagi factorization on gpu using cuda. Technical report, School of Computing, University of Utah, 2010.

[Ige10] Christian Igel. Kernel-based learning methods - linear regression and discrimination. Technical report, Institut für Neuroinformatik - Ruhr-Universität Bochum, 2010.

[JD92] Kresimir Veselic James Demmel. Jacobi's method is more accurate than qr. Technical report, Fernuniversität Hagen, 1992.

[Kuh06] Anthony Kuh. Neural networks and learning theory - lecture notes. Technical report, University of Hawaii, 2006.

[Li08] Jia Li. Linear discriminant analysis. Technical report, Department of Statistics - Pennsylvania State University, 2008.

[NVI10a] NVIDIA. *CUDA C best practices guide 3.2*. NVIDIA, 2010.

[NVI10b] NVIDIA. *CUDA C programming guide 3.2.* NVIDIA, 2010.

[SG10] Ivor Wai-Hung Tsang Shenghua Gao. Kernel sparse representation for image classification and face recognition. Technical report, School of Computer Engineering, Nanyang Technological Univertiy, Singapore, 2010.

[SS02] Schölkopf and Smola. *Learning with Kernels.* MIT Press, 2002.

[Tha06] Nattanum Thatphithakkul. Robust speech recognition using kpca-based noise classification. Technical report, 2006.

[TKC07] C.V. Jawahar Tejo Krishna Chalasani, Anoop M. Namboodiri. Support vector machine based hierarchical classifiers for large class problems. Technical report, Center for Visual Information Technology, International Institute of Information Technology, Hyderabad, India, 2007.

[Vas09] Nuno Vasconcelos. Dot-product kernels. Technical report, ECE Department, UCSD, 2009.

[Weg09] Björn Weghenkel. Statistical methods for side-channel based reverse engeneering. Technical report, 2009.

[Wis10] Laurenz Wiskott. Machine learning - lecture notes. Technical report, Ruhr-Universität Bochum, 2010.