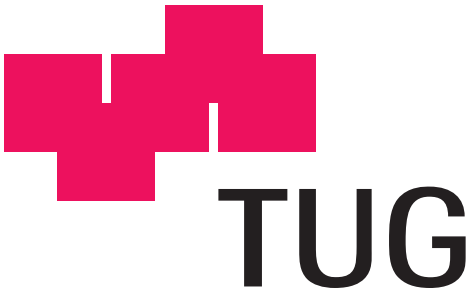
A Digitalization Of A Recurrent Neural Network

Master's Thesis at the
Institute of Theoretical Computer Science
Faculty of Informatics



Graz University of Technology
(Technische Universität Graz)

submitted by

Mark Kröll

Advisor: *o.Univ.-Prof. DI Dr. Wolfgang Maass*

Graz, September 2006

Contents

Abstract

In this thesis a new architecture (see Section 5.2) is introduced to model the function of a cortical microcircuit. It represents a digital form of an already existing framework for computations in neural microcircuits, the Liquid State Machine(LSM) [29]. The LSM, a recurrent neural networks of biologically realistic neurons, incorporates fading memory, temporal integration and task independence. These characteristics of the neural network are translated in the world of kernels, what is implicitly done within the LSM as well. Fading memory is achieved by including a weight vector into the used kernels, temporal integration by concatenation of several time steps and task independence by using unsupervised kernel methods. Exploiting the idea behind using kernel methods, i.e. the projection of the data into a higher dimensional space, simple learning algorithms such as linear regression can be applied.
In Chapter 6, the new architecture is compared to its companion, the LSM, by performing prediction tasks on visual data sets, that are generated artificially and taken from the real-world.

Keywords: Machine Learning, Fading Kernels, Kernel Methods, Liquid State Machine, Fading Memory, Temporal Integration

Gott gebe mir die Gelassenheit, Dinge hinzunehmen, die ich nicht ändern kann, den Mut, Dinge zu ändern, die ich ändern kann, und die Weisheit, das eine vom anderen zu unterscheiden.

Reinhold Niebuhr

Zusammenfassung

In dieser Arbeit wird eine neue Architektur (siehe Sektion 5.2) vorgestellt, die die Funktion eines kortikalen Mikroschaltkreises modelliert. Sie stellt eine digitale Form eines bereits existierenden Frameworks für Berechnungen in einem neuronalen Mikroschaltkreis, der Liquid State Machine(LSM) [29], dar. Die LSM, ein rekurrentes, neuronales Netzwerk, das aus biologisch realistischen Neuronen aufgebaut ist, beinhaltet Fading Memory, zeitliche Integration und Unabhängigkeit von der jeweiligen Fragestellung. Diese Eigenschaften des neuronalen Netzwerks werden in die Welt der Kernels transferiert, was implizit auch innerhalb des Netzwerks geschieht. Fading Memory wird durch das Hinzufügen eines Gewichtvektors in die benutzten Kernel erreicht, zeitliche Integration durch das Vereinigen mehrerer Zeitschritte und die Aufgabenunabhängigkeit durch das Verwenden von unüberwachten Kernelmethoden. Unter Ausnutzung der Projektion der Daten in einen höher dimensionalen Raum, ist es möglich, einfache Lernalgorithmen, wie die lineare Regression, anzuwenden.
In Kapitel 6 wird die neue Architektur mit ihrem Pendant, der LSM, verglichen. Es werden Vorhersageaufgaben auf visuellen Datensätzen ausgeführt, die einerseits künstlich generiert werden und andererseits aus der realen Welt entnommen werden.

Stichwörter: Maschinelles Lernen, Fading Kernels, Kernel Methoden, Liquid State Machine, Fading Memory, zeitliche Integration

Danksagung

Ich möchte mich an dieser Stelle bei meinen Eltern bedanken, die mir stets auf allen meinen Wegen beistanden und mir die Möglichkeit gaben, mein Leben nach meinen Vorstellungen zu gestalten. Ihnen ist diese Arbeit gewidmet.

Mein Dank gilt auch Professor Dr. Maass, der mir die Gelegenheit gab, meine Magisterarbeit im Bereich des maschinellen Lernens zu verfassen.

Ich bedanke mich bei Bernhard Nessler, der mich zu dieser Arbeit animierte und mich einen Grossteil des Weges begleitete. Unsere nächtlichen Gespräche waren mir als Austausch wissenschaftlicher wie philosophischer Ideen und Ansichten stets willkommen.

Herzlichen Dank auch an Harald Burgsteiner, Gerald Steinbauer und vor allem Alexander Leopold. Gemeinsam mit ihnen ist meine erste Publikation entstanden, die ebenfalls Teil dieser Arbeit ist.

Meine letzte Danksagung gilt meinem Arbeitgeber, dem Know-Center, und Michael Granitzer für deren Unterstützung im letzten Abschnitt meiner Arbeit.

List of Figures

List of Tables

Chapter 1

Introduction

When sensory information is processed in living organisms, the complex stream of information has to pass a lot of stages, where it is object to various preprocessing steps such as information compression, offering a sparse representation and segregation of the conceived information into different pathways.

Considering the processes visual information has to undergo, the representation of information that is required by the first cortical area, the striate cortex (V1), is demanding. Let's take a look at these preprocessing steps that are applied to the visual information on its way from the retina to V1.

At first, the received information is segregated into parallel pathways, the magnocellular pathway and parvocellular pathway. The magnocellular pathway carries, among other things, information about the motion of objects in the visual field whereas the parvocellular pathway deals with the shape of objects, depending on the receptors in the retina. Hence, different types of retinal information are kept separate.

Another preprocessing step involving compression takes place at the crossing from the retina, which contains approximately 125 million nerve cells, into the optical nerve that contains only 2 million nerve cells any more. After passing the lateral geniculate nucleus (LGN), the parallel pathways arrive at different layers (mainly layer four) of the visual cortex (V1). There, simplest features such as lines and edges are extracted in order to be further processed by higher hierarchical regions of the cortex.

This hierarchical structure in the cortex, where simple features are extracted in early stages and are then combined in higher layers, partly answers for the mental capabilities living organisms are equipped with. In computational sciences there is the desire to make use of this known structure in order to gain access to the computational power humans can access in their everyday lives.

In [28] it is proposed to model the computational function of a cortical microcircuit as a combination of three basic operations:

1. analog fading memory
2. a non-linear kernel that generates a large number of items at different locations in time and space from the analog fading memory

3. linear readouts that are trained to extract specific features from the output of the kernel

The Liquid State Machine(LSM)[30], a recurrent neural network consisting of biologically realistic neurons, intends to provide the computational function of a cortical microcircuit.

This thesis aims to implement a digitalized form of the computational paradigm LSM. Hence, a stacked architecture is introduced that consists of two layers. The first layer serves as preprocessing unit where salient information is extracted to provide a compact representation of the visual input data. Kernel Principal Component Analysis (kernel PCA) [6] using standard kernels and a greedy algorithm for a training set reduction [27] are applied in the first layer.

The features extracted are combined in that sense that temporal integration is performed and information of a larger time window (see [23]) is available in the second layer. In contrast to the first layer, kernel PCA is used incorporating kernels that possess a fading property, i.e. older information does have less relevance than newer one and eventually *fades* away).

On top of the second layer, a simple linear learning algorithm, linear regression in my case, is placed. This architecture is applied to noisy, real-world video data as well as to an artificially generated data set that was used in [2] with the LSM. The required task encompasses the learning of a prediction up to four time steps ahead. The stacked architecture is compared to its neural network counterpart using the settings from [13]. Video data sets are applied to both learning machines.

The thesis is structured as follows: Chapter 2 introduces kernel functions that represent the crucial component in the kernel methods used throughout the stacked architecture. The kernel methods employed in the two-layered architecture are described in Chapter 3. Chapter 4 gives a proper introduction to the LSM. In Chapter 5 the experimental setup of both approaches as well as the two visual input data sets are presented. The obtained results and a comparison of the results are discussed in Chapter 6. Chapter 7 offers a short summary and considers further experiments that might lead to more insight into processing of sensory information in humans.

Chapter 2

Kernel Functions

Kernels are the basic components shared by all kernel methods. They provide a general framework to represent data and must satisfy some mathematical conditions, e.g., positive definiteness.

Methods and algorithms using kernels provide a new answer to the question of data representation. Data is not represented individually any more, but only through a set of pairwise comparisons (see Figure 2.1).

I denote by $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n), \mathbf{x}_i \in \mathcal{X}$ a set of n objects to be analysed such as a set of images or strings. These objects are usually represented by finite real-valued vectors ($\phi(\mathbf{x}) \in \mathbb{R}^p$). A real-valued "comparison function" $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is used resulting in a representation of the data set $\mathbf{X}$ as $n \times n$ matrix of pairwise comparisons $k_{i,j} = k(\mathbf{x}_i, \mathbf{x}_j)$. All kernel methods are designed to process such square matrices independent of the nature of the objects to be processed. This property is exploited in various scientific fields where data of different nature need to be integrated and analysed in a unified framework. Processing of DNA sequences in computational biology, dealing with images in computer vision and analysing chemical compounds in chemical informatics are representatives for such applications. Another advantage results from the fact that the size of the matrix used to represent a data set of n objects is always $n \times n$. Computationally, this is very attractive in the case when a small number of complex objects are to be processed. However, it should be noted that the size of the square matrix is dependent on the number of objects, acting as an information bottleneck. Hence, when large data sets are to be processed, conventional kernel methods fail and have to be replaced by kernel methods that are capable of handling large input sets.

The comparison function k is the critical component of any kernel method, since it defines how the algorithm "sees" the data. Since a lot of kernel methods can only handle symmetric, positive definite square matrices, there are certain requirements the comparison function k has to satisfy.

Definition 1. *A function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is called a positive definite kernel iff it is symmetric, that is, $k(\mathbf{x}_i, \mathbf{x}_j) = k(\mathbf{x}_j, \mathbf{x}_i)$ for any two objects $\mathbf{x}_i, \mathbf{x}_j \in \mathcal{X}$,*

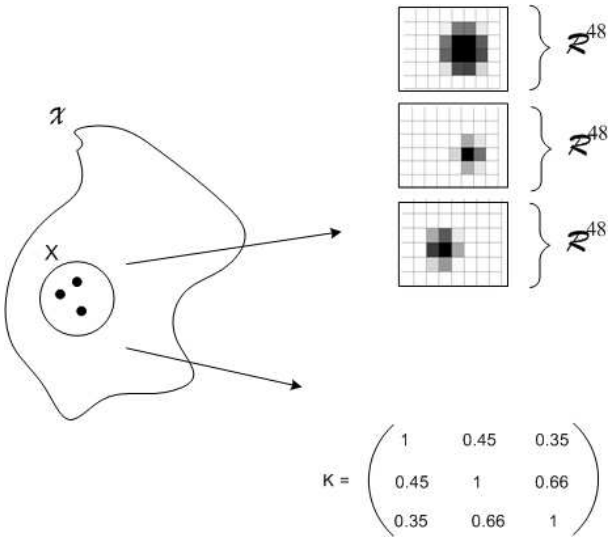


Figure 2.1: Two different representations of the same data set. $\mathcal{X}$ is supposed to be the set of all images, and X is a data set of three particular images depicted on the right side. The set of images denotes different ball positions and serves as data set for the experiments performed in this thesis. The classic way to represent X is to convert the images into vectors and treat each vector individually. Kernel methods are based on a different representation of X, as a matrix of pairwise similarity between its elements.

and positive definite, that is,

$$\sum_{i=1}^{n}\sum_{j=1}^{n} c_i c_j k(\mathbf{x}_i, \mathbf{x}_j) \geq 0 \tag{2.1}$$

for any $n > 0$, any choice of n objects $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathcal{X}$, and any choice of real numbers $c_1, \ldots, c_n \in \mathbb{R}$.

Positive definite kernels are also referred to as *Mercer* kernels. A symmetric matrix is positive definite only if all its eigenvalues are nonnegative, a valuable property that can be checked without great effort.

2.1 Feature Space

Suppose the data to be analysed are real valued vectors ($\mathbf{x} \in \mathbb{R}^p$) and any object is represented as $\mathbf{x} = (x_1, \ldots, x_p)^\mathsf{T}$. One might guess that calculating the inner product between vectors is a way of comparison, i.e., length and direction, and indeed it meets the requirements defined in Definition 1. It is usually referred to as linear kernel (see Equation2.2).

$$k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\mathsf{T}\mathbf{x}' = \sum_{i=1}^{p} x_i x_i' \tag{2.2}$$

Since objects of all kinds, e.g. image data, are admitted, one has to represent each object $\mathbf{x} \in \mathcal{X}$ as a vector $\phi(\mathbf{x}) \in \mathbb{R}^p$ and then defining a kernel for any $\mathbf{x}, \mathbf{x}' \in \mathcal{X}$ by

$$k(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^\mathsf{T} \phi(\mathbf{x}'). \tag{2.3}$$

Function k defined in (2.3) is a valid kernel on the space $\mathcal{X}$. Any mapping $\phi : \mathcal{X} \to \mathbb{R}^p$ for some p$\geq$ 0 results in a valid kernel. Replacing $\mathbb{R}^p$ by an eventually infinite-dimensional Hilbert space[1] leads to the important theorem:

Theorem 1. *For any kernel k on a space $\mathcal{X}$, there exists a Hilbert space $\mathcal{F}$ and a mapping $\phi : \mathcal{X} \to \mathcal{F}$ such that*

$$k(\mathbf{x}, \mathbf{x}') = \langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle \tag{2.4}$$

for any $\mathbf{x}, \mathbf{x}' \in \mathcal{X}$, where $\langle u, v \rangle$ denote the dot product in the Hilbert space between any two points $u, v \in \mathcal{F}$.

Theorem 1 provides a first intuition about kernels: They can be thought of as dot products in some space $\mathcal{F}$, usually called the *feature space*. This intuition of kernels as dot products is useful allowing a geometric interpretation of kernel methods.

Besides the geometric interpretation, Theorem 1 states that calculating dot products in possibly infinite-dimensional spaces reduces to applying kernels in $\mathcal{X}$. It enables us to carry out calculations implicitly without ever dealing with the feature space vectors explicitly.
Along with Theorem 1 goes the following proposition:

Proposition 1. *Any algorithm for vectorial data that can be expressed only in terms of dot products between vectors can be performed implicitly in the feature space associated with any kernel, by replacing each dot product by a kernel evaluation.*

Proposition 1 is usually refered to as the kernel trick. It provides a convenient way to transform linear methods such as principal component analysis, matching pursuit, or the generalized hebbian algorithm, by simply replacing the dot product by a more general kernel. Nonlinearity is then obtained at no computational cost, as the algorithm remains exactly the same. The reformulation of linear algorithms is commonly refered to as *kernelization*.

2.2 Similarity Measurements

A common view of kernels is as a measure of similarity, in the sense that $k(\mathbf{x}, \mathbf{x}')$ is large when $\mathbf{x}$ and $\mathbf{x}'$ are similar. Methods like Support Vector Machines (SVM) [8] for example base their prediction on the assumption that 'similar' points are likely to have the same values. The 'similarity' between points is

[1] a vector space that is endowed with a dot product and complete for the norm induced. R^p with the classic inner product is an example of a finite-dimensional Hilbert space

determined by the kernel.

There are three standard kernels that are introduced in this Section, the polynomial kernel, the Gaussian Radial Basis Function kernel and the sigmoidal kernel. Each of them defines a different measure of similarity.

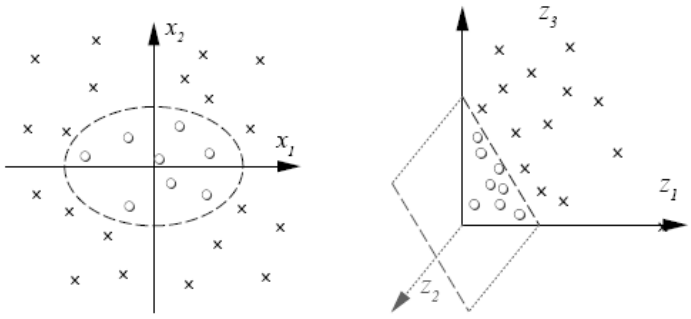


Figure 2.2: Two dimensional classification example. By mapping the data from $\mathbb{R}^2$ to $\mathbb{R}^3$ a linear decision boundary can be found. (Figure from [5])

The polynomial kernel is best introduced by the toy example in Figure 2.2. The task is to separate the circles from the crosses. By mapping the input data in a higher dimensional space (see Equation 2.5), a separation can be found using a linear hyperplane (right), instead of having to construct a non-linear ellipsoidal decision boundary (left). The corresponding feature space is the space of second order monomials.

$$\begin{array}{rcl} \phi : \mathbb{R}^2 & \rightarrow & \mathbb{R}^3 \\ (x_1, x_2) & \mapsto & (z_1, z_2, z_3) := (x_1^2, \sqrt{2}x_1x_2, x_2^2) \end{array} \tag{2.5}$$

The computation of a scalar product between two feature space vectors can be reformulated in terms of a kernel function k

$$\begin{array}{rcl} (\phi(\mathbf{x}) \cdot \phi(\mathbf{x}')) & = & (x_1^2, \sqrt{2}x_1x_2, x_2^2)((x')_1^2, \sqrt{2}(x')_1(x')_2, (x')_2^2) \\ & = & ((x_1, x_2)(x_1', x_2')^\mathsf{T})^2 \\ & = & (\mathbf{x} \cdot \mathbf{x}')^2 \\ & = & k(\mathbf{x}, \mathbf{x}'). \end{array} \tag{2.6}$$

A generalization leads to the polynomial kernel:

$$k(\mathbf{x}, \mathbf{x}') = ((\mathbf{x}, \mathbf{x}') + \theta)^d \tag{2.7}$$

where $\theta \in \mathbb{R}$ and $d \in \mathbb{N}$. The kernel computes a scalar product in the space of all products of d vector entries (monomials) of $\mathbf{x}$ and $\mathbf{x}'$.

Yet one might consider the notion of distance as a measure of similarity rather than the notion of dot product. However, there are cases, where these ideas coincide. For example, the kernel on $\mathcal{X} = \mathbb{R}^p$, called the Gaussian Radial Basis Function (RBF) kernel

$$k(\mathbf{x}, \mathbf{x}') = exp(-\frac{||\mathbf{x} - \mathbf{x}'||^2}{2\sigma^2}) \tag{2.8}$$

where $\sigma > 0$ denotes the width of the Gaussian hill. This is a valid kernel, which can be considered as a polynomial kernel of infinite degree. Its features are all possible monomials of the input features with no restriction placed on the degrees. The Taylor expansion of the exponential function

$$exp(\mathbf{x}) = \sum_{i=0}^{\infty} \frac{1}{i!} x^i \tag{2.9}$$

shows that the weighting of individual monomials falls off as $i!$ with increasing degree.

The RBF kernel can be interpreted as a decreasing function of the Euclidian distance between points. Each point is represented by a bell-shaped function sitting on that point. The larger the kernel $k(\mathbf{x}, \mathbf{x}')$, the closer the points $\mathbf{x}$ and $\mathbf{x}'$ in $\mathcal{X}$.

The last kernel to be considered is the sigmoidal kernel

$$k(\mathbf{x}, \mathbf{x}') = tanh(\kappa(\mathbf{x} \cdot \mathbf{x}') + \theta) \tag{2.10}$$

where $\kappa, \theta \in \mathbb{R}$. The main motivation behind the usage of this kernel is that the decision function learned by a kernel method is a particular type of two-layer sigmoidal neural network. Problems might arise using this type of kernel, since the Mercer conditions are not always assured as it is the case with the former two kernels. So certain parameter combinations of κ and θ do not lead to positive definiteness. Still the sigmoidal kernel has been successfully used in practice.

I conclude this section by the fact that a universal kernel capable of dealing with any kind of data does not exist. Regarding various types of data, the requirements of a learning task on the definition of similarity differ. Often the optimal parameters according to the given data have to be determined in advance, e.g. by parameter sweeps. If standard kernels such as polynomial, RBF or sigmoidal do not suffice, own kernels can be designed that calculate desired, data-dependent measures of similarity.
Thus prior knowledge about the data structure has to be taken into account when kernels are used and designed.

2.3 Function Regularity

In this section I would like to point out that by selecting a particular kernel, the space of functions on $\mathcal{X}$ and the norm on that space are determined as well.

As first example, consider the linear kernel (2.2) on a vector space $\mathcal{X} = \mathbb{R}^p$. The corresponding functional space is the space of linear function $f : \mathbb{R}^d \to \mathbb{R}$:

$$\mathcal{H}_k = \{f(\mathbf{x}) = w^\mathsf{T}\mathbf{x} : w \in \mathbb{R}^p\} \tag{2.11}$$

The associated norm is just the slope of the linear function,

$$\|f\|_{\mathcal{H}_k} = \|w\| \tag{2.12}$$

for $f(\mathbf{x}) = w^\mathsf{T}\mathbf{x}$. As a second example, consider the Gaussian RBF kernel (2.8) on the same vector space $\mathcal{X} \in \mathbb{R}^p$. The associated functional space is the set of functions $f : \mathbb{R}^d \to \mathbb{R}$ with Fourier transform $\hat{f}$ that satisfies:

$$N(f) = \frac{1}{(2\pi\sigma^2)^{\frac{p}{2}}} \int_{R^p} |\hat{f}(\omega)|^2 e^{\frac{\sigma^2}{2}\|\omega\|^2} d\omega < +\infty, \tag{2.13}$$

and the norm in $\mathcal{H}_k$ is precisely this functional: $\|f\|_{\mathcal{H}_k} = N(f)$. Hence $\mathcal{H}_k$ is a set of functions with Fourier transforms that decay rapidly, and the norm $\|.\|_{\mathcal{H}_k}$ quantifies how fast this decay is.

In both examples, the norm $\|f\|_{\mathcal{H}_k}$ decreases if the 'smoothness' of f increases, where the definition of smoothness depends on the kernel. In case of the linear kernel, the smoothness is related to the slope of the function: a smooth function is a flat function. For the Gaussian RBF kernel, the smoothness of a function is measured by its Fourier spectrum: a smooth function has little energy at high frequencies. To state it otherwise - a function is smooth, if it varies slowly between similar points.

2.3.1 Reproducing Kernel Hilbert Space

Given the kernel k, how is the corresponding functional space $\mathcal{H}_k$ constructed? The set $\mathcal{H}_k$ is defined as the set of function $f : \mathcal{X} \to \mathbb{R}$ of the form

$$f(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i k(\mathbf{x}_i, \mathbf{x}), \tag{2.14}$$

for $n > 0$, a finite number of points $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathcal{X}$, and a finite number of weights $\alpha_1, \ldots, \alpha_n \in \mathbb{R}$, together with their limits under the norm:

$$\|f\|^2_{\mathcal{H}_k} = \sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \alpha_j k(\mathbf{x}_i, \mathbf{x}_j) \tag{2.15}$$

$\mathcal{H}_k$ is a Hilbert space, with a dot product defined for two elements $f(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i k(\mathbf{x}_i, \mathbf{x})$ and $g(\mathbf{x}) = \sum_{j=1}^{m} \alpha'_j k(\mathbf{x}'_j, \mathbf{x})$ by

$$\langle f, g \rangle = \sum_{i=1}^{n}\sum_{j=1}^{m} \alpha_i \alpha'_j k(\mathbf{x}_i, \mathbf{x}'_j) \tag{2.16}$$

A property of this construction is that the value $f(\mathbf{x})$ of a function $f \in \mathcal{H}_k$ at a point $\mathbf{x} \in \mathcal{X}$ can be expressed as a dot product in $\mathcal{H}_k$,

$$f(\mathbf{x}) = \langle f, k(\mathbf{x}, .) \rangle \tag{2.17}$$

In particular, taking $f(.) = k(\mathbf{x}', .)$, following reproducing property can be derived valid for any $\mathbf{x}, \mathbf{x}' \in \mathcal{X}$:

$$k(\mathbf{x}, \mathbf{x}') = \langle k(\mathbf{x}, .), k(\mathbf{x}', .) \rangle \tag{2.18}$$

For this reason, the functional space $\mathcal{H}_k$ is usually called the reproducing kernel Hilbert space (RKHS) associated with k.

2.3.2 Regularization

One should keep in mind the connection between kernels and norms on functional spaces. Most kernel methods have an interpretation in terms of functional analysis and can be defined as algorithms that, given a set of objects $\mathbf{X}$, return a function that solves the equation

$$\min_{f \in \mathcal{H}_k} R(f, \mathbf{X}) + c\|f\|_{\mathcal{H}_k}, \tag{2.19}$$

where $R(f, \mathbf{X})$ is small when f 'fits' the data well, and the term $\|f\|_{H_k}$ ensures that the solution of (2.19) is 'smooth'. The parameter $c > 0$ is the regularization parameter which specifies the trade-off between minimization of $R(f, \mathbf{X})$ and the smoothness or simplicity which is enforced by a small norm. It can be viewed as a trade-off between overfitting and generalization as well.

By thinking of kernels as regularization operators, the Representer Theorem can help to see many kernel methods in a different light.

Theorem 2. *Let $\mathcal{X}$ be a set endowed with a kernel k, and $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \mathcal{X}$ a finite set of objects. Let $\Psi : \mathbb{R}^{n+1} \to \mathbb{R}$ be a function of $n+1$ arguments, strictly monotonic increasing in its last argument. Then any solution of the problem*

$$\min_{f \in \mathcal{H}_k} \Psi(f(\mathbf{x}_1), \ldots, \|f\|_{\mathcal{H}_k}), \tag{2.20}$$

where $(\mathcal{H}_k, \|.\|_{\mathcal{H}_k})$ is the RKHS associated with k, admits a representation of the form

$$\forall \mathbf{x} \in \mathcal{X}, f(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i k(\mathbf{x}_i, \mathbf{x}). \tag{2.21}$$

I would like to throw a closer look to the effects of this important Theorem. By including a dependency in $\|f\|_{\mathcal{H}_k}$ in the function to optimize, the solution is forced to be smooth. This penalization is reasonable since demanding smoothness in the solution is usually a powerful protection against overfitting of the data.

Moreover the *representer theorem* states that the solution lies in the span of n particular kernels - those centered on the training points - resulting in

computational advantages. Any solution to (2.19) is known to belong to a subspace of $\mathcal{H}_k$ of dimension at most n, the number of training points, even though the optimization is carried out over a possibly infinite-dimensional space $\mathcal{H}_k$. Hence the problem is reduced to an n-dimensional optimization problem $\rightarrow$ by combining (2.21) and (2.20) and optimizing over $(\alpha_1, \ldots, \alpha_n) \in \mathbb{R}^n$.

To summarize: By selecting a kernel, a space of functions with a corresponding norm is determined. Kernel methods can be understood to return a function that solves Equation 2.19. Theorem 2 entitles us to perform a reduced optimization problem on the functional space by including the norm of that space in the optimization problem as a stability term (regularizer). The resulting function has the property of being smooth (definition of smoothness depends on the kernel) meaning that the solution is optimal in terms of generalization.

These observations can serve as a guide to choose a kernel for practical applications, if one has some prior knowledge about the function the algorithm should output. So, the challenge is to design a kernel such that a priori desirable functions have a small norm.

I recommend books from Schölkopf/Smola [5] and Shawe-Taylor/Cristianini [15] that provide both a thorough introduction into kernels and kernel methods.

Chapter 3

Kernel Methods

In this chapter the algorithms used in the stacked architecture are reviewed. I commence by introducing kernel PCA [6], the *kernelization* result of PCA[18]. Afterwards a greedy derivative of kernel PCA -greedy KPCA [27]- is discussed that aims at reducing the number of training samples by selecting only the most representative ones.

These methods provide a framework for feature extraction in an unsupervised manner. The algorithms attempt to find some structure that is possibly hidden in the data without having access to specified target values as in supervised methods (e.g. Support Vector Machines [8]). A resulting property of unsupervision is that extracted features are not dependent on a particular task specified in advance but can be used by simple learning rules to tackle a variety of tasks.

However, it should be noted that the mentioned kernel methods prove ill when dealing with large data sets is required. Consequently, the size of the input set is limited - otherwise the computations become infeasable because of too large kernel matrices. Different approaches exist (see Section 7.1) that approach this problem in a greedy, sparse or iterative way and thus could be used to enlarge the input set.

3.1 Kernel Principal Component Analysis

Kernel principal component analysis [6] is going to be applied in the first layer (see Section 5.2.1) for dimensionality reduction as well as in the second layer (see Section 5.2.2) as unsupervised learning algorithm in combination with kernels having fading property.

Kernel PCA is a powerful technique for extracting non-linear structure from data, thus capturing part of higher-order statistics. The basic idea is to map the input data into a Reproducing Kernel Hilbert Space (RKHS) and then perform Principal Component Analysis in that space (see Figure 3.1).
Hence, kernel PCA represents a form of non-linear PCA. Before discussing the kernel PCA algorithm, the standard PCA algorithm is introduced.

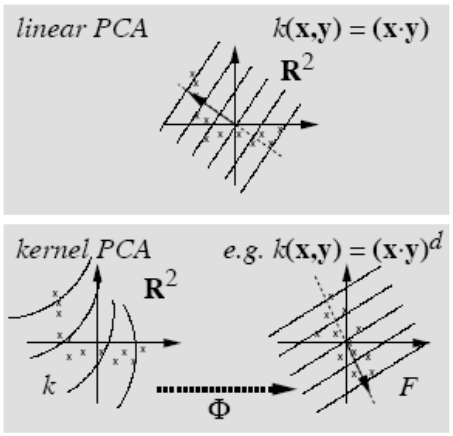


Figure 3.1: A comparison of linear PCA and kernel PCA is depicted. (Figure from [6])

3.1.1 Principal Component Analysis

Being one of the most valuable results from applied linear algebra, PCA [18] is used abundantly in all forms of analysis for it is a simple, non-parametric method of extracting relevant information from confusing data sets. PCA provides the means to reduce a complex data set to a lower dimension to reveal the sometimes hidden, simplified dynamics.

3.1.2 Toy Example

A simple toy example from Shlens [17] will help to understand the mechanisms that are behind PCA. In addition, the obtained results will emphasize the value of this method. The settings are depicted in Figure 3.2. The motion of the physicist's ideal spring shall be studied. This system consists of a ball of mass m attached to a massless, frictionless spring. The ball is released a small distance away from equilibrium. The spring being 'ideal' oscillates indefinitely along the x-axis about its equilibrium at a certain frequency. This is a standard problem in physics in which the motion along the x direction is solved by an explicit function of time. In other words, the underlying dynamics can be expressed as a function of a single variable x.

Experimenters often face situations as described above where phenomenas want to be understood by measuring all available quantities. Hence, more dimensions than actually needed are recorded because experimenters lack the knowledge which measurements best reflect the underlying dynamics of the system in question. Back to the toy example additional measurements are taken into account although only information about the single variable x would suffice. The ball's position in a three-dimensional space is recorded by three cameras A,B and C (see Figure 3.2). Each camera records an image indicating a two-dimensional position of the ball, i.e., camera A records a corresponding po-

sition $(x_A(t), y_A(t))$. Each trial can be expressed as a six dimensional vector $X = (x_A, y_A, x_B, y_B, x_Z, y_Z)$.

PCA aims at finding the most meaningful basis to re-express a noisy data set. The hope is that this new basis will filter out the noise and reveal hidden dynamics. In the example of the spring, the explicit goal of PCA is to determine: "the dynamics are along the x-axis".

PCA makes one stringent but powerful assumption: *linearity*. Assuming linearity simplifies the problem by restricting the set of potential bases and formalizing the implicit assumption of continuity in a data set. With this assumption PCA is now limited to reexpressing the data as a *linear combination* of its basis vectors. Let $\mathbf{X}$ and $\mathbf{Y}$ be $m \times n$ matrices related by linear transformation $\mathbf{P}$. $\mathbf{X}$ is the original recorded data set and $\mathbf{Y}$ is re-representation of that data set.

$$\mathbf{PX} = \mathbf{Y} \tag{3.1}$$

Equation 3.1 represents a change of basis and can be interpreted in various ways:

- $\mathbf{P}$ is a matrix that transforms $\mathbf{X}$ into $\mathbf{Y}$.
- $\mathbf{P}$ can be interpreted geometrically as rotation and a stretch again transforming $\mathbf{X}$ into $\mathbf{Y}$.
- The rows of $\mathbf{P}$ are a set of new basis vectors for expressing the column vectors in $\mathbf{X}$.

Each column $\mathbf{y}_i$ of $\mathbf{Y}$ can be calculated by calculating the dot product between the corresponding column $\mathbf{x}_i$ of $\mathbf{X}$ and the new basis vectors which happen to be the row vectors $\mathbf{p}$ of $\mathbf{P}$.

$$\mathbf{y}_i = \begin{bmatrix} \mathbf{p}_1 \cdot \mathbf{x}_i \\ \vdots \\ \mathbf{p}_m \cdot \mathbf{x}_i \end{bmatrix} \tag{3.2}$$

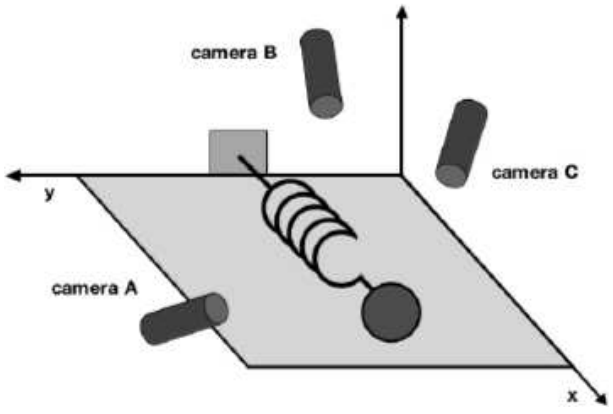


Figure 3.2: Setup of a toy example to demonstrate the effects of Principal Component Analysis. (Figure from [17])

In other words, the j^{th} coefficient of $\mathbf{y}_i$ is a projection of $\mathbf{x}_i$ onto the j^{th} basis vector $\mathbf{p}_j$. The basis vectors become the *principal components* of $\mathbf{X}$.

After the illustration of transforming $\mathbf{X}$ into $\mathbf{Y}$ by means of a new basis, some questions arise such as, What is a good choice of the basis $\mathbf{P}$? and What is the best way to re-express $\mathbf{X}$? These questions can be answered by next asking what features $\mathbf{Y}$ shall exhibit?

When dealing with a linear system, data becomes "garbled" due to noise and redundancy. In order to clean the data, one strives for getting rid of noise and redundancy $\rightarrow$ $\mathbf{Y}$ should be free of noise and redundancy.

Noise

To be able to extract any information about a system, noise in a data set must be low. A common measure is the *signal-to-noise ratio* (SNR), or a ratio of variances σ^2.

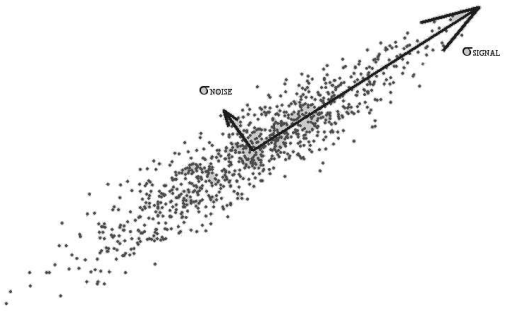


Figure 3.3: The signal and noise variances σ^2_{signal} and σ^2_{noise} of one camera are graphically represented.

$$SNR = \frac{\sigma^2_{signal}}{\sigma^2_{noise}} \tag{3.3}$$

A high SNR ($\gg$1) indicates high precision data, while a low SNR indicates noise comtaminated data. Figure 3.3 shows a simulated plot of the ball's position recorded by camera A. Any camera should record motion in a straight line. Therefore, any spread deviating from straigt-line motion must be noise. The variance due to the signal and noise are indicated graphically. The ratio of the variances, i.e. the SNR, measures how "fat" the oval is - the range of possibilities includes a thin line (SNR $\gg$ 1), a perfect circle (SNR = 1) or even worse.

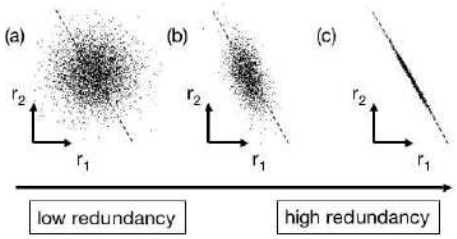


Figure 3.4: A spectrum of possible redundancies in data from two separate recordings r_1 and r_2. The best fit line $r_2 = kr_1$ is indicated by the dashed line. (Figure from [17])

Redundancy

In the example of the spring, redundancy in the measured variables is caused by multiple sensor recordings of the same dynamic information. Consider Figure 3.4 as a range of possible plots between two arbitrary measurement types r_1 and r_2. Panel (a) depicts two recordings with no redundancy. In other words, r_1 is totally uncorrelated with r_2. This situation could occur by plotting two variables such as temperature and income.

However in panel (c) both recordings appear to be strongly related. This extremity might be achieved by plotting the same variable in different scales (celsius, fahrenheit) or related to the toy example, a plot of (x_A, x_B) if the cameras are very nearby.

Consequently, regarding panel (c), it would be more sensible to just have recorded only one variable, the linear combination $r_2 - kr_1$, instead of two variables r_1 and r_2. Recording solely the linear combination would both express the data more concisely and reduce the number of sensor readings. Indeed, this is the very idea behind dimensionality reduction.

Covariance Matrix

The *covariance* represents a generalization of the variance which only takes into account one variable. The covariance matrix of the already centered data $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ where $\sum_{i=1}^{n} \mathbf{x}_i = 0$ is defined in Equation 3.4. Each row of $\mathbf{X}$ corresponds to all measurements of a particular type. Each column of $\mathbf{X}$ corresponds to a set of measurements of particular trial or time step.

$$\mathbf{C}_X = \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i \mathbf{x}_i^{\mathsf{T}}. \tag{3.4}$$

Characteristics of the covariance matrix are enlisted below.

- $\mathbf{C}_X$ is a square matrix having as much dimensions as there are measurement types.
- The diagonal terms of $\mathbf{C}_X$ are the variances of the particular measurement types.

- The off-diagonal terms of $\mathbf{C}_X$ are covariance between measurement types.

Computing $\mathbf{C}_X$ quantifies the correlation between all possible pairs of measurements. Between one pair of measurements, a large covariance corresponds to a situation like panel (c) in Figure 3.4, while zero covariance corresponds to entirely uncorrelated data as in panel (a).

At this point, I refer to the above question - What features $\mathbf{Y}$ and $\mathbf{C}_Y$ respectively shall exhibit? If the goal is to reduce redundancy, then each variable should co-vary as little as possible with other variables. More precisely, to remove redundancy the covariances between separate measurements ought to be zero. Hence an optimal covariance matrix $\mathbf{C}_Y$ looks like a diagonal matrix with all off-diagonal terms zero. In other words, removing redundancy diagonalizes $\mathbf{C}_Y$.

Before deriving an algebraic solution to PCA, I will informally describe what happens during PCA. PCA assumes that the basis vectors $\mathbf{p}_i$ are orthonormal, as a consequence $\mathbf{P}$ is an orthonormal matrix as well. Secondly, PCA assumes that the directions with the largest variances are the most important or most *principal*. PCA first selects a normalized direction along which the variance in $\mathbf{X}$ is maximized - it saves this as $\mathbf{p}_1$. Again it finds another direction along which the variance is maximized, however, because of the orthonormality condition, it restricts its search to all directions perpendicular to all previously selected directions. This could continue until m directions are selected. The resulting ordered set $\mathbf{p}$'s are the *principal components*. The components are rank-ordered according to the corresponding variances.

Eigenvectors of Covariance

The algebraic solution is based on an important property of eigenvector decomposition. Once again, the data set is $\mathbf{X}$, an $m \times n$ matrix, where m is the number of measurement types and n is the number of data trials. The objective is to find some orthonormal matrix $\mathbf{P}$ where $\mathbf{P} \cdot \mathbf{X} = \mathbf{Y}$ such that $\mathbf{C}_Y$ is diagonalized. First, $\mathbf{C}_Y$ is rewritten:

$$\begin{aligned} C_Y &= \tfrac{1}{n}\mathbf{Y}\mathbf{Y}^\mathsf{T} \\ &= \tfrac{1}{n}(\mathbf{P}\mathbf{X})(\mathbf{P}\mathbf{X})^\mathsf{T} \\ &= \tfrac{1}{n}\mathbf{P}(\mathbf{X}\mathbf{X}^\mathsf{T})\mathbf{P}^\mathsf{T} \\ C_Y &= \tfrac{1}{n}\mathbf{P}\mathbf{A}\mathbf{P}^\mathsf{T} \end{aligned}$$

The *symmetric* matrix $\mathbf{A}$ is diagonalized by an orthogonal matrix of its eigenvectors and can be expressed by

$$\mathbf{A} = \mathbf{E}\mathbf{D}\mathbf{E}^\mathsf{T} \tag{3.5}$$

where $\mathbf{D}$ is a diagonal matrix and $\mathbf{E}$ is a matrix of eigenvectors of $\mathbf{A}$ arranged as columns. The matrix $\mathbf{A}$ has $r \leq m$ orthonormal eigenvectors where r denotes the rank of the matrix. The rank of $\mathbf{A}$ is less than m either when $\mathbf{A}$ is degenerative or all data occupy a subspace of dimension $r \leq m$.

The desired transformation matrix $\mathbf{P}$ is now chosen to be a matrix where each row $\mathbf{p}_i$ is an eigenvector of $\mathbf{X}\mathbf{X}^\mathsf{T}$, thus $\mathbf{P} \equiv \mathbf{E}^\mathsf{T}$. Substituting into Equation 3.5, matrix $\mathbf{A}$ reads $\mathbf{A} = \mathbf{P}^\mathsf{T}\mathbf{D}\mathbf{P}$. The covariance matrix $\mathbf{C}_Y$ accordingly takes the following form:

$$\begin{aligned} C_Y &= \tfrac{1}{n}\mathbf{P}\mathbf{A}\mathbf{P}^\mathsf{T} \\ &= \tfrac{1}{n}(\mathbf{P}\mathbf{P}^\mathsf{T})\mathbf{D}(\mathbf{P}\mathbf{P}^\mathsf{T}) \\ &= \tfrac{1}{n}(\mathbf{P}\mathbf{P}^{-1})\mathbf{D}(\mathbf{P}\mathbf{P}^{-1}) \\ C_Y &= \tfrac{1}{n}\mathbf{D} \end{aligned}$$

The choice of $\mathbf{P}$ diagonalizes $\mathbf{C}_Y$.

Singular Value Decomposition(SVD) denotes another algebraic solution for PCA. It is closely related to PCA, in fact, the names of both methods are used interchangeably. In the course of SVD the diagonalization is performed by solving the eigenvalue problem expressed by

$$\lambda\mathbf{e} = \mathbf{C}\mathbf{e} \tag{3.6}$$

for eigenvalues $\lambda \geq 0$ and eigenvectors $\mathbf{e}_i \in \mathbb{R}^p \setminus \{0\}$. The resulting set of mutually orthogonal eigenvectors defines again a new basis along the directions of maximal variance in the data. The orthogonal projection onto the eigenvectors are called *principal components* (PC's) of the data set.

Concluding Remarks on PCA

Beside the fact, that performing PCA is a quite simple procedure, PCA provides useful properties regarding optimality that are enumerated in the following.

1. The first $r (r \in \{1, \ldots, n\})$ extracted features, or projections on the first r eigenvectors (assuming that the eigenvectors are sorted in descending order of their eigenvalues), carry more variance than any other r orthogonal directions.

2. If observations in $\mathcal{H}$ should be represented by the first r extracted features, the mean-squared approximation error is minimal (over all possible r directions).

3. The extracted features are uncorrelated.

4. The first r features have maximal mutual information.

Hence, the PCA basis , among all basis expansions, minimizes the reconstruction error when the expansion is truncated to a smaller number of basis vectors. That is why PCA is one of the prefered methods when dimensionality is to be reduced.

Furthermore, omitting principal components corresponding to small eigenvalues, i.e. principal components that are associated with low variances, equals to noise reduction since noise is associated with directions of little variance in the data.

Both the strength and weakness of PCA is that it is a non-parametric analysis. There are no parameters to set and no coefficients to adjust. It is not possible to incorporate prior knowledge about the dynamics of the system. A famous example where PCA fails is a ring shaped distribution. Prior knowledge, i.e. converting the data to the appropriately centered polar coordinates, needs to be incorporated before computing the PCA.

Last but not least, it should be noted that PCA tends to fail when it is confronted with non-gaussian data distributions, since the axes with the largest variance do not correspond to the underlying basis. In exponentially distributed data, for example, the largest variance do not correspond to the underlying basis. Under such circumstances, other methods like *Independent Component Analysis* [1] succeed.

3.1.3 The KPCA algorithm

To be able to perform a non-linear form of PCA, the input data $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$, $\mathbf{x}_i \in \mathcal{X}$ has to be mapped into a possibly infinite-dimensional feature space $\mathcal{H}$ (RKHS) as stated above. The non-linear map reads:

$$\Phi : \mathcal{X} \to \mathcal{H}, \mathbf{x} \mapsto \Phi(\mathbf{x}) \tag{3.7}$$

It is assumed that the mapped data is centered, $\sum_{i=1}^{n} \Phi(\mathbf{x}_i) = 0$. In $\mathcal{H}$ the covariance matrix takes the following form:

$$\mathbf{C} = \frac{1}{n} \sum_{i=1}^{n} \Phi(\mathbf{x}_i)\Phi(\mathbf{x}_i)^{\mathsf{T}}. \tag{3.8}$$

In PCA like manner, eigenvalues $\lambda \geq 0$ and nonzero eigenvectors $\mathbf{v} \in \mathcal{H} \setminus \{0\}$ satisfying

$$\lambda \mathbf{v} = \mathbf{C}\mathbf{v} \tag{3.9}$$

All solutions $\mathbf{v}$ with $\lambda \neq 0$ lie in the span of $\Phi(\mathbf{x}_1), \ldots, \Phi(\mathbf{x}_n)$. The consequences imply that one may consider the equivalent system

$$\lambda \langle \Phi(\mathbf{x}_k), \mathbf{v} \rangle = \langle \Phi(\mathbf{x}_k), \mathbf{C}\mathbf{v} \rangle \qquad \text{, for all } k = 1 \ldots n \tag{3.10}$$

and that there exist coefficients α_i $(i = 1 \dots n)$ such that

$$\mathbf{v} = \sum_{i=1}^{n} \alpha_i \Phi(\mathbf{x}_i) \tag{3.11}$$

Combining equations (3.10) and (3.11) leads to

$$\lambda \sum_{i=1}^{n} \alpha_i \langle \Phi(\mathbf{x}_k), \Phi(\mathbf{x}_i) \rangle = \frac{1}{n} \sum_{i=1}^{n} \alpha_i \Big\langle \Phi(\mathbf{x}_k), \sum_{j=1}^{n} \Phi(\mathbf{x}_j) \langle \Phi(\mathbf{x}_j), \Phi(\mathbf{x}_i) \rangle \Big\rangle \tag{3.12}$$

for all $k = 1 \dots n$. In terms of the $n \times n$ Gram matrix $K_{ij} := \langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle$ equation 3.12 reads

$$n\lambda \mathbf{K}\boldsymbol{\alpha} = \mathbf{K}^2 \boldsymbol{\alpha} \tag{3.13}$$

where $\boldsymbol{\alpha}$ denotes the column vector with entries $\alpha_1 \dots \alpha_n$. To find solutions of equation (3.13), one has to solve the dual eigenvalue problem for nonzero eigenvalues.

$$n\lambda \boldsymbol{\alpha} = \mathbf{K}\boldsymbol{\alpha} \tag{3.14}$$

$\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_n$ denote the eigenvalues of the Gram matrix $\mathbf{K}$ and $\boldsymbol{\alpha}^1, \dots, \boldsymbol{\alpha}^n$ the corresponding complete set of eigenvectors.
The solutions $\boldsymbol{\alpha}^1, \dots, \boldsymbol{\alpha}^p$ belonging to nonzero eigenvalues are normalized by requiring that the corresponding vectors $\mathbf{v}$ in the feature space $\mathcal{H}$ be normalized (see equation (3.11)).

$$\langle \mathbf{v}^k, \mathbf{v}^k \rangle = 1 \qquad \text{for all } k = 1 \dots p \tag{3.15}$$

Using equations (3.11) and (3.14) the normalization of $\boldsymbol{\alpha}^1, \dots, \boldsymbol{\alpha}^p$ can be carried out by

$$\begin{aligned} 1 &= \textstyle\sum_{i,j=1}^{n} \alpha_i^k \alpha_j^k \langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle \\ &= \textstyle\sum_{i,j=1}^{n} \alpha_i^k \alpha_j^k K_{i,j} \\ &= \langle \boldsymbol{\alpha}^k, \mathbf{K}\boldsymbol{\alpha}^k \rangle \\ &= \lambda_k \langle \boldsymbol{\alpha}^k, \boldsymbol{\alpha}^k \rangle \end{aligned} \tag{3.16}$$

The data needs to be centered in $\mathcal{H}$ as well. This can be done by simply substituting the matrix $\mathbf{K}$ with

$$\hat{\mathbf{K}} = \mathbf{K} - 1_n \mathbf{K} - \mathbf{K} 1_n + 1_n \mathbf{K} 1_n, \tag{3.17}$$

where $(1_n)_{ij} = 1/n$; for details see [6].

In order to extract principal components, one has to compute projections onto the eigenvectors $\mathbf{v}^k$ in $\mathcal{H}$ (k = 1 ... p). A test point $\mathbf{x}$ is mapped into the feature space at first. Then

$$\langle \mathbf{v}^k, \Phi(\mathbf{x}) \rangle = \sum_{i=1}^{n} \alpha_i^k \langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}) \rangle \tag{3.18}$$

are the non-linear principal components (or features) corresponding to Φ. See Figure 3.5 for the exemplary extraction of one principal component. It can be viewed as projecting the test image, that is mapped into a high dimensional space at first, onto the eigenvector V. The test image $\mathbf{x}$ is compared to the training images $\mathbf{x}_i$ by applying the kernel function k. The feature value results from the linear combination $\sum_{i=1}^{n} \alpha_i k(\mathbf{x}_i, \mathbf{x})$ where the weights are found by solving an eigenvalue problem.

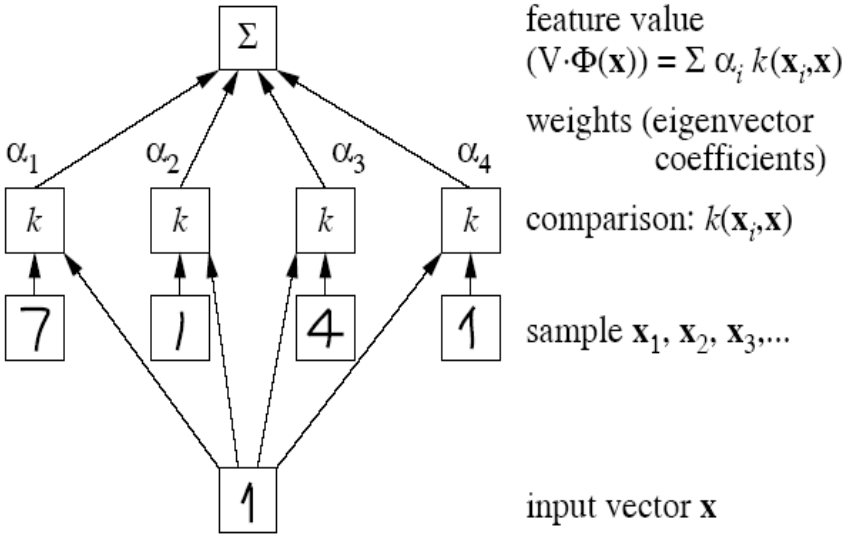


Figure 3.5: The process of feature extraction by kernel PCA is illustrated. A test point x is compared to all other points in the training set by applying the kernel function k. The weights of the linear combination are found by solving an eigenvalue problem. Implicitly the test point is mapped into a high dimensional space and there it is projected onto the eigenvector V. (Figure from [5])

Note that neither computing the entries of the Gram matrix $\mathbf{K}$ nor the feature extraction (3.18) requires $\Phi(\mathbf{x})$ in explicit form. Therefore, applying kernel functions for computing the needed dot-products is feasable without actually performing the map Φ.

Known, optimal properties of kernel PCA that represents similar to PCA an orthogonal basis transformation yet not in the input space but in a possibly infinite-dimensional feature space $\mathcal{H}$ are:

1. The first $r (r \in \{1, \ldots, n\})$ extracted features, or projections on the first r eigenvectors (assuming that the eigenvectors are sorted in descending order of their eigenvalues), carry more variance than any other r orthogonal directions.

2. If observations in $\mathcal{H}$ should be represented by the first r extracted features, the mean-squared approximation error is minimal (over all possible r directions).

3. The extracted features are uncorrelated.

4. The first r features have maximal mutual information.

However, when taking a closer look at the feature extraction process a serious drawback of kernel PCA and kernel methods in general becomes obvious. When an image is projected onto eigenvectors in the feature space, the algorithm needs to have access to all members of the input set $\mathbf{X}$ (see Equation (3.18)). Thus kernel PCA is limited in the number of input samples that can be processed.

Finally I would like to connect kernel PCA with terms as regularization, functional space and an associated norm introduced in Chapter 2.

The feature extractors (see Equation 3.18) are linear functions in the feature space $\mathcal{H}$

$$f_k(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i^k \langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}) \rangle = \sum_{i=1}^{n} \alpha_i^k k(\mathbf{x}_i, \mathbf{x}) \tag{3.19}$$

Regularization properties are closely related to the norm of a functional space. As discussed in Section 2.3, smoothness ensures protection against overfitting, thus leading to low capacity. Smoothness is increased by decreasing the norm, here in case of linear functions this corresponds to keeping the length of the weight vector $\boldsymbol{\alpha}$ short.

When applied to the training data, the kth feature extractor generates a set of outputs with variance λ_k. Dividing each coefficient vector $\boldsymbol{\alpha}^k$ by $\sqrt{\lambda_k}$, a set of non-linear feature extractors with unit variance output is obtained. A direct consequence of this scaling is the property that the kth feature extractor is optimal among all feature extractors of the form of Equation (3.19) in the sense that it has minimal weight vector norm in the RKHS $\mathcal{H}$ that can be obtained combining Equations(3.19) and (2.15)

$$\|f_k\|^2 = \langle \boldsymbol{\alpha}^k, \mathbf{K}\boldsymbol{\alpha}^k \rangle \tag{3.20}$$

Unlike PCA, kernel PCA allows extraction of a number of features which can exceed the input dimensionality though requiring that the number of training samples exceeds the input dimensionality as well.

3.2 Greedy Kernel PCA

Greedy KPCA is going to be applied in the first layer of the stacked architecture (see Section 5.2.1) to reduce the input dimensionality. The algorithm intends to provide answers regarding two problems that occur when performing kernel PCA. The first one concerns the memory capacity. The storage of the Gram

matrix that contains the training data in terms of dot-products becomes expensive since the size of the matrix increases quadratically with the number of training samples. Note that kernel PCA requires access to all training samples.

Secondly, the solution - a linear function in the feature space - is not sparse, i.e. many coefficients α_i are nonzero. The non-sparse solution implies an expensive evaluation. Greedy KPCA circumvents these two problems by proposing a technique to approximate the training set.

Let $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$, $\mathbf{x}_i \in \mathcal{X}$ be the training data that is mapped into a high-dimensional space $\mathcal{H}$ by

$$\Phi : \mathcal{X} \to \mathcal{H} \tag{3.21}$$

The set of training data transformed to $\mathcal{H}$ is denoted as $\mathbf{H} = (\Phi(\mathbf{x}_1), \ldots, \Phi(\mathbf{x}_n))$. The transformed training data $\mathbf{H}$ live in a subspace span$(\mathbf{H}) \subseteq \mathcal{H}$.

It is attempted to find a finite subset $\mathbf{X}_r = (\mathbf{r}_1, \ldots, \mathbf{r}_m)$, $\mathbf{r}_i \in \mathcal{X}$ with corresponding images $\mathbf{H}_r = (\Phi(\mathbf{r}_1), \ldots, \Phi(\mathbf{r}_m))$. Let's suppose that the vectors $\Phi(\mathbf{r}_i)$ are linearly independent and thus forming a basis of linear subspace span$(\mathbf{H}_r) \subseteq \mathcal{H}$. The transformed training data $\mathbf{H}$ is now expressed in a linear basis defined by $\mathbf{H}_r$.

$\mathbf{H}' = (\Phi(\mathbf{x}_1)', \ldots, \Phi(\mathbf{x}_n)')$ denotes a set of approximations of vectors in $\mathbf{H}$ which will be computed as minimal square error projections on the subspace span$(\mathbf{H}_r)$. An approximation $\Phi(\mathbf{x})' \in \mathbf{H}'$ of the vector $\Phi(\mathbf{x}) \in \mathbf{H}$ is accomplished by a linear combination of vectors of $\mathbf{H}_r$.

$$\Phi(\mathbf{x})' = \sum_{i=1}^{m} \beta_i \Phi(\mathbf{r}_i) = \mathbf{H}_r \cdot \boldsymbol{\beta} \tag{3.22}$$

The vector $\boldsymbol{\beta}$ contains real coefficients of linear combination and is computed by an optimization problem that reads

$$\begin{aligned} \boldsymbol{\beta} &= arg\min_{\boldsymbol{\beta}'} \|\Phi(\mathbf{x}) - \Phi(\mathbf{x})'\|^2 \\ &= (\Phi(\mathbf{x}) - \mathbf{H}_r \cdot \boldsymbol{\beta}')^{\mathsf{T}} (\Phi(\mathbf{x}) - \mathbf{H}_r \cdot \boldsymbol{\beta}') \end{aligned} \tag{3.23}$$

The *pseudoinverse* provides a well-known analytical solution to such an inconsistent system

$$\boldsymbol{\beta} = (\mathbf{H}_r{}^{\mathsf{T}} \cdot \mathbf{H}_r)^{-1} \mathbf{H}_r \cdot \Phi(\mathbf{x}) \tag{3.24}$$

Since Equation (3.24) can be represented in terms of dot-products, the coefficient vector $\boldsymbol{\beta}$ can be expressed in the following way

$$\boldsymbol{\beta} = \mathbf{K}_r^{-1} \cdot \sum_{i=1}^{m} k(\mathbf{r}_i, \mathbf{x}) = \mathbf{K}_r^{-1} \cdot k_r(\mathbf{x}) \tag{3.25}$$

where $\mathbf{x} \in \mathbf{X}$ is a vector to be approximated, $\mathbf{K}_r = \mathbf{H}_r{}^{\mathsf{T}} \cdot \mathbf{H}_r$ is a kernel matrix $m \times m$ of vectors from the set $\mathbf{X}_r$.

Let $\boldsymbol{\beta}_i$ denote the coefficient vector for a training sample $\mathbf{x}_i \in \mathbf{X}$. The approximated value of the kernel function of two training samples $\mathbf{x}_i, \mathbf{x}_j \in \mathbf{X}$ is calculated as

$$\begin{aligned} k'(\mathbf{x}_i, \mathbf{x}_j) &= \langle \Phi(\mathbf{x}_i)', \Phi(\mathbf{x}_j)' \rangle \\ &= (\mathbf{H}_r \cdot \boldsymbol{\beta}_i)^\mathsf{T} \cdot (\mathbf{H}_r \cdot \boldsymbol{\beta}_j) \\ &= \boldsymbol{\beta}_i{}^\mathsf{T} \cdot \mathbf{K}_r \cdot \boldsymbol{\beta}_j \end{aligned} \tag{3.26}$$

Being positive definite, the kernel matrix $\mathbf{K}_r$ can be decomposed by the Choleski factorization leading to $\mathbf{K}_r = \mathbf{R}^\mathsf{T} \cdot \mathbf{R}$, where matrix $\mathbf{R}$ is an upper triangular matrix. Thus the computation of the approximated kernel function can be simplified

$$\begin{aligned} k'(\mathbf{x}_i, \mathbf{x}_j) &= \boldsymbol{\beta}_i{}^\mathsf{T} \cdot \mathbf{K}_r \cdot \boldsymbol{\beta}_j \\ &= \boldsymbol{\beta}_i{}^\mathsf{T} \cdot \mathbf{R}^\mathsf{T} \cdot \mathbf{R} \cdot \boldsymbol{\beta}_j \\ &= \langle \boldsymbol{\gamma}_i, \boldsymbol{\gamma}_j \rangle \end{aligned} \tag{3.27}$$

The training set can be represented by a matrix $\boldsymbol{\Gamma} = [\boldsymbol{\gamma}_1, \ldots, \boldsymbol{\gamma}_n]$ of size $m \times n$ instead of the whole kernel matrix $\mathbf{K}$ of size $n \times n$. m is the number of elements in $\mathbf{H}_r$ used to approximate the subspace span($\mathbf{H}$) and n is the number of training samples. Evidently setting $\mathbf{H}_r = \mathbf{H}$, a perfect approximation without errors is obtained. If a perfect approximation is achieved, span($\mathbf{H}_r$) equals span($\mathbf{H}$), but not necessarily $n = m$, if the data is linearly dependent in the space $\mathcal{H}$.

The next subsection deals with the selection process, i.e.which training samples should be contained in $\mathbf{H}_r$?

3.2.1 Selection Procedure

In the case of greedy KPCA, the whole data set has to be accessible during the selection process. The algorithm cycles through the data set in an iterative way, each time selecting that training sample that together with the previously chosen samples leads to the best approximation result in that moment. To prevent the algorithm from looping indefinitely, the iteration procedure stops, if the approximation of the training set yields an acceptable level.

Hence, an approximation error $se(\mathbf{x})$ of the transformed vector $\Phi(\mathbf{x})$ is defined as follows:

$$\begin{aligned} se(\mathbf{x}) &= (\Phi(\mathbf{x}) - \Phi(\mathbf{x})')^\mathsf{T} (\Phi(\mathbf{x}) - \Phi(\mathbf{x})') \\ &= (\Phi(\mathbf{x}) - \mathbf{H}_r \cdot \beta)^\mathsf{T} (\Phi(\mathbf{x}) - \mathbf{H}_r \cdot \boldsymbol{\beta}) \\ &= k(\mathbf{x}, \mathbf{x}) - 2\mathbf{k}_r(\mathbf{x})^\mathsf{T} \cdot \boldsymbol{\beta} + \boldsymbol{\beta}^\mathsf{T} \cdot \mathbf{K}_r \cdot \boldsymbol{\beta} \end{aligned} \tag{3.28}$$

An acceptable level is achieved, if the approximation error crosses a predetermined threshold ϵ, i.e. $se(\mathbf{x}) < \epsilon, \forall \mathbf{x} \in \mathbf{X}$. Another possibility to stop the

iteration is when a certain number of elements in $\mathbf{H}_r$ is reached (this way memory constraints can be met).

Algorithm 1 Training set approximation

1: Initialize the $\mathbf{X}_r = \{\mathbf{r}\}$, $\mathbf{r} = arg\max_{\mathbf{x}\in\mathbf{X}} k(\mathbf{x}, \mathbf{x})$

2: **while** size of $\mathbf{X}_r$ is less than m **do**
3: Compute $se(\mathbf{x})$ for all training vectors not yet included in $\mathbf{X}_r$.
{It is required to compute $\boldsymbol{\beta} = \mathbf{K}_r^{-1} \cdot \mathbf{k}_r(\mathbf{x})$ where $\mathbf{K}_r$ is the kernel matrix of the current set $\mathbf{X}_r$.}

4: **if** $\max_{\mathbf{x}\in\mathbf{X}\setminus\mathbf{X}_r} se(\mathbf{x}) < \epsilon$ **then**
5: Stop the iteration.

6: **else**
7: Add $\mathbf{x} = arg\max_{\mathbf{x}\in\mathbf{X}\setminus\mathbf{X}_r} se(\mathbf{x})$ to the set $\mathbf{X}_r$ and continue the iteration.

The initialization step can be viewed as a selection of the training sample that accounts for the worst approximation error when the subset $\mathbf{X}_r$ is empty, i.e. all samples are projected onto the origin. The algorithm cycles as long as the termination conditions are not met. The termination conditions ensure, that - if met - all remaining training samples can be approximated sufficiently and/or the number of samples to select is reached.

In each cycle the approximation error for all projected training samples that are not included in the set $\mathbf{X}_r$ is computed. The training sample that is approximated worst by a linear combination of the samples in $\mathbf{X}_r$ is added to this set.

After the application of the algorithm a subset $\mathbf{X}_r \subset \mathbf{X}$ is obtained which contains the basis vectors along with the matrix $\mathbf{K}_r^{-1}$. Hence, the coefficient vector $\boldsymbol{\beta}$ can be computed using Equation (3.25), leading to a new representation of the data.

Chapter 4

The Liquid State Machine

4.1 The framework of a Liquid State Machine

The "liquid state machine" (LSM) from [30] is a new framework for computations in neural microcircuits. The term "liquid state" refers to the idea to view the result of a computation of a neural microcircuit not as a stable state like an attractor that is reached. Instead, a neural microcircuit is used as an *online computation tool* that receives a continuous input that drives the state of the neural microcircuit. The result of a computation is again a continuous output generated by readout neurons given the current state of the neural microcircuit.

Recurrent neural networks (see Figure 4.1) with spiking neurons represent a non-linear dynamical system with a high-dimensional internal state, which is driven by the input. The internal state vector $x(t)$ is given as the contributions of all neurons within the LSM to the membrane potential of a readout neuron at the time t. The complete internal state is determined by the current input and all past inputs that the network has seen so far. Hence, a history of (recent) inputs is preserved in such a network and can be used for computation of the current output. The basic idea behind solving tasks with a LSM is that one does *not* try to set the weights of the connections within the pool of neurons but instead reduces learning to setting the weights of the readout neurons. This reduces learning dramatically and much simpler supervised learning algorithms which e.g. only have to minimize the mean square error in relation to a desired output can be applied.
The LSM has several interesting features in comparison to other approaches with recurrent circuits of spiking neural networks:

1. The liquid state machine provides "any-time" computing, i.e. one does not have to wait for a computation to finish before the result is available. Results start emitting from the readout neurons as soon as input is fed into the liquid. Furthermore, different computations can overlap in time. That is, new input can be fed into the liquid and perturb it while the readout still gives answers to past input streams.

2. A single neural microcircuit can not only be used to compute a special

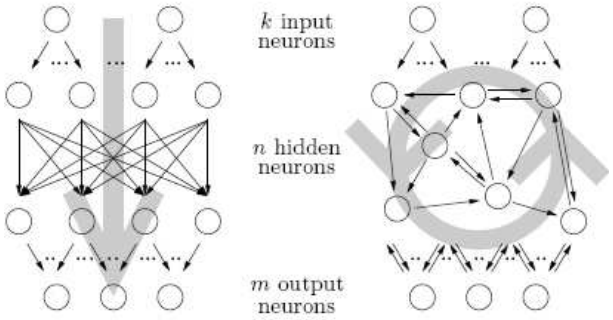


Figure 4.1: Comparison of the architecture of a feed-forward (left hand side) with a recurrent neural network (right hand side); the gray arrows sketch the direction of computation. Figure from [13].

output function via the readout neurons. Because the LSM only serves as a pool for dynamic recurrent computation, one can use many different readout neurons to extract information for several tasks in parallel. So a sort of "multi-tasking" can be incorporated. Figure 4.2 illustrates this and the previous property.

3. In most cases simple learning algorithms can be used to set the weights of the readout neurons. The idea is similar to support vector machines, where one uses a kernel to project input data into a high-dimensional space. In this very high-dimensional space simpler classifiers can be used to separate the data than in the original input data space. The LSM has a similar effect as a kernel: due to the recurrency the input data is also projected to a high-dimensional space. Hence, in almost any case experienced so far simple learning rules like e.g. linear regression suffice.

4. Last but not least it is not only a computational powerful model, but it is also one of the biological most plausible so far. Thus, it provides a hypothesis for computation in biological neural systems.

4.2 Neural microcircuit

The model of a neural microcircuit as it is used in the LSM is based on evidence found in [11] and [4]. Still, it gives only a rough approximation to a real neural microcircuit since many parameters are still unknown. The neural microcircuit is the biggest computational element within the LSM (see Figure 4.3), although multiple neural microcircuits could be placed within a single virtual model.

In a model of a neural microcircuit $N = n_x \cdot n_y \cdot n_z$ neurons are placed on a regular grid in 3D space. The number of neurons along the x, y and z axis,

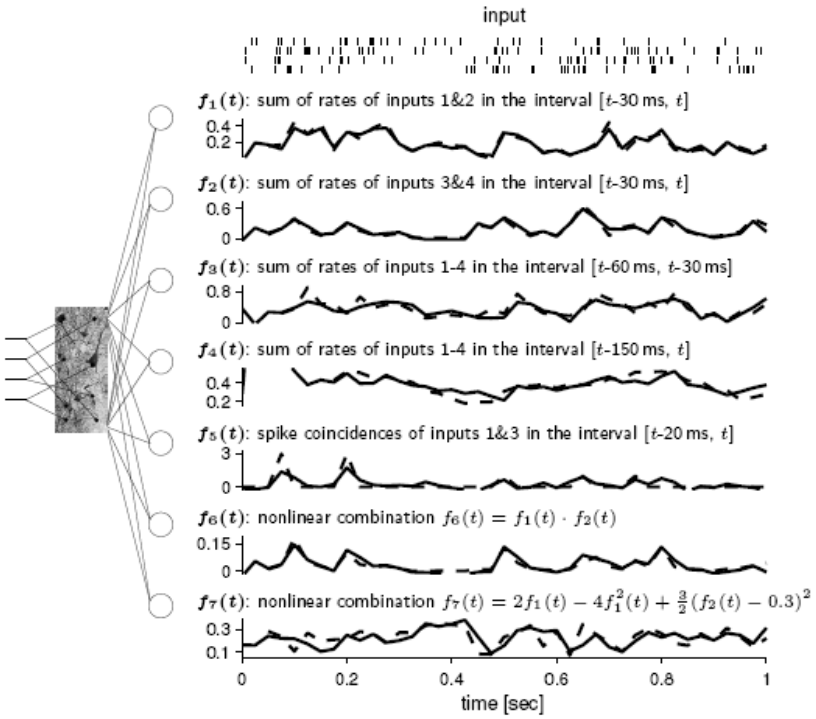


Figure 4.2: Multi-tasking with any-time computing. A single neural microcircuit can be used by different readout-neurons to compute various function in parallel. In this case, based on a Poisson spike train as input to the LSM, 7 different functions were computed by readout neurons (Figure from [25]).

n_x, n_y and n_z respectively, can be chosen freely. One also specifies a factor to determine how many of the N neurons should be inhibitory. Another important parameter in the definition of a neural microcircuit is the parameter λ. Number and range of the connections between the N neurons within the LSM are determined by this parameter λ. The probability of a connection between two neurons i and j is given by

$$p_{(i,j)} = C \cdot exp^{-\frac{D_{(i,j)}}{\lambda^2}}$$

where $D_{(i,j)}$ is the Euclidean distance between those two neurons and C is a parameter depending on the type (excitatory or inhibitory) of each of the two connecting neurons. There exist 4 possible values for C for each connection within a neural microcircuit: C_{EE}, C_{EI},C_{IE} and C_{II} may be used depending on whether the neurons i and j are excitatory (E) or inhibitory (I). In our experiments we used spiking neurons according to the standard leaky-integrate-and-fire (LIF) neuron model that are connected via dynamic synapses. The time course for a postsynaptic current is approximated by the equation:

$$v(t) = w \cdot e^{-\frac{t}{\tau_{syn}}}$$

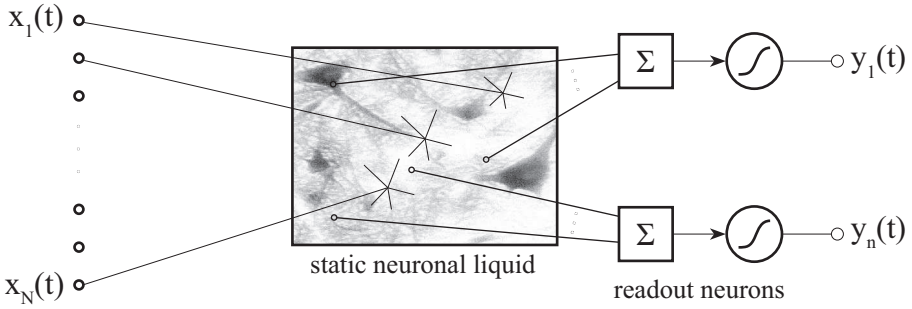


Figure 4.3: **Neural microcircuit with liquid** The input x(t) is applied to the neurons (Figure from [7])

where w is a synaptic weight and τ_{syn} is the synaptic time constant. In case of dynamic synapses the "weight" w depends on the history of the spikes it has seen so far according to the model from [12]. For synapses transmitting analog values (such as the output neurons in our experimental setup) synapses are simply modeled as static synapses with a strength defined by a constant weight w. Additionally, synapses for analog values can have delay lines, modeling the time a potential would need to propagate along an axon.

4.3 Parameterization of the LSM

In this section the LSM that is used to carry out the prediction task is presented. That includes the liquid itself (i.e. how many neurons are contained), the percentage of inhibitory neurons, the type of connections and so on.

To feed input into the liquid, *External Input Neurons* are used that conduct an injection current I_{inject} via *Static Analog Synapses* (parameters are shown in Table 4.1) into the first layer of the liquid pool. Inspired from information processing in living organisms, I set up a cognitive mapping from input layer to liquid pool. The value of I_{inject} depends on the value of the input data. , in this case the activation of each single visual sensor (for further details see Section 5.1).

The liquid consists of 144 *Leaky Integrate And Fire Neurons* (parameters are listed in Table 4.2), grouped in an $8 \cdot 6 \cdot 3$ cuboid, that are randomly connected via *Dynamic Spiking Synapses* (parameters are listed in Table 4.3), as described above. The probability of a connection between every two neurons is modeled by the probability distribution depending on a parameter λ described in the previous section. Various combinations of λ (connection probability) and mean connection weights Ω (connection strength) were used for simulation. 20% of the liquid neurons were randomly chosen to produce inhibitory potentials. Figure 4.4 shows an example for connections within the LSM.

I_{noise}	w_{mean}		$delay_{mean}$	
$[nA]$	-		$[ms]$	
	EE	EI	EE	EI
0	$3 \cdot 10^{-8}$	$6 \cdot 10^{-8}$	1.5	0.8

Table 4.1: Parameters for Static Analog Synapses used to feed the LSM with input. EE and EI denote whether the source and target neurons of a connection emit excitatory or inhibitory action potentials. Co-Variance for $delay_{mean}$ is 0.1.

V_{thresh}	$V_{resting}$	V_{reset}	V_{init}	$T_{refract}$		I_{noise}	I_{inject}
$[mV]$	$[mV]$	$[mV]$	$[mV]$	$[ms]$		$[nA]$	$[nA]$
				E	I		
15	0	$U(13.8, 14.5)$	$U(13.5, 14.9)$	3	2	0	$U(13.5, 14.5)$

Table 4.2: Parameters for Leaky Integrate And Fire Neurons comprising the liquid ($C_m = 30nF, R_m = 1M\Omega$). Letters 'E' and 'I' indicate whether the neurons emit excitatory or inhibitory action potentials. $U(a, b)$ denotes an uniform distribution on the interval $[a, b]$.

	U_{mean}	D_{mean}	F_{mean}	$delay_{mean}$	τ_{syn}	C
connection	-	-	$[s]$	$[ms]$	$[ms]$	-
EE	0.5	1.1	0.05	1.5	3	0.3
EI	0.05	0.125	1.2	0.8	3	0.4
IE	0.25	0.7	0.02	0.8	6	0.2
II	0.32	0.144	0.06	0.8	6	0.1

Table 4.3: Parameters for Dynamic Spiking Synapses connecting neurons inside the liquid. EE, EI, IE and II denote whether the source and target neurons of a connection emit excitatory or inhibitory action potentials. Co-Variance for $delay_{mean}$ is 0.1.

The information provided by the spiking neurons in the liquid pool is processed (read out) by *External Output Neurons* (V_{init}, $V_{resting}$, I_{noise} are the same as for the liquid neurons), each of them connected to all neurons in the liquid pool via *Static Spiking Synapses* (parameters are listed in Table 4.4). The output neurons perform a simple linear combination of inputs that are provided by the liquid pool.
For simulation within the training and evaluation the neural circuit simulator *CSim*[1] was used. Parameterization of the LSM is described below. Names

[1]The software simulator *CSim* and the appropriate documentation for the liquid state machine can be found on the web page *http://www.lsm.tugraz.at/*

τ_{syn}		w	$delay_{mean}$	
$[ms]$		-	$[ms]$	
EE	EI		EE	EI
3	6	$-6.73 \cdot 10^{-5}$	1.5	0.8

Table 4.4: Parameters for Static Spiking Synapses connecting the liquid with each read out neuron. EE and EI denote whether the source and target neurons of a connection fire excitatory (E) or inhibitory (I) action potentials. The value given for w only serves as an example of values set after training. Co-Variance for $delay_{mean}$ is 0.1.

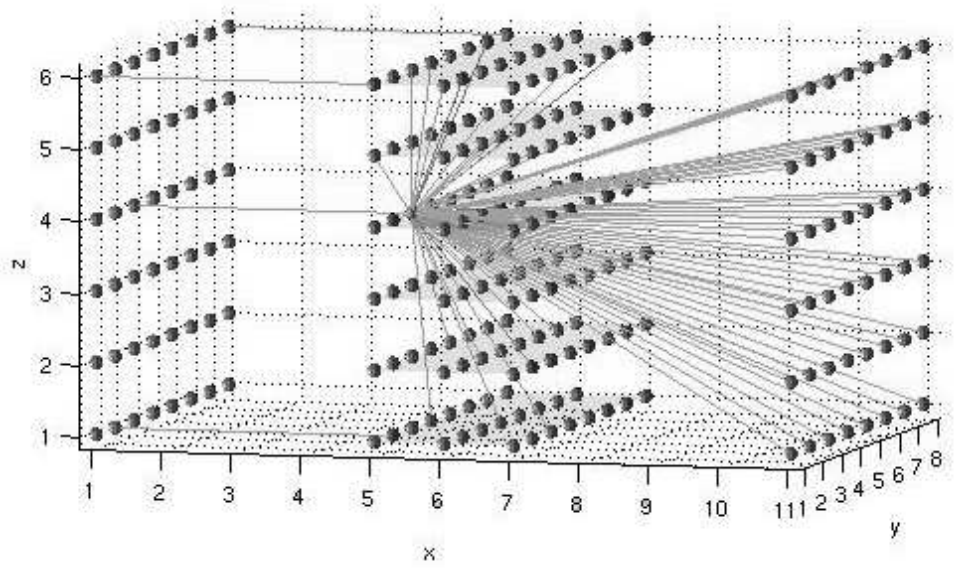


Figure 4.4: An example showing the connections of a single liquid neuron: input is received from the input sensor field on the left hand side and some random connection within the liquid. The output of every liquid neuron is projected onto every output neuron (located on the most right hand side). The 8x6x3 neurons in the middle form the "liquid"

for neuron and synapse types all originate from terms used in the *CSim* environment. Letters I and E denote values for inhibitory and excitatory neurons respectively.

Chapter 5

Experimental Setup

The aim of this chapter is to describe the experimental setup when applying the LSM (see Section 5.1) to perform the prediction task and furthermore to introduce a two-layered architecture (see Section 5.2) using kernels to perform the same task. The results of both approaches are compared and discussed in the next chapter. Furthermore the stacked architecture is applied to another data (see Section 5.3) set that has been artificially generated and resembles the one in [2].

5.1 Applying the Liquid State Machine

The input data was recorded by using a prototype of the middle-size-league RoboCup robot in use (and developed) by the RoboCup team at the Graz University of Technology. The experimental setup can be described as follows: the robot was located on the field and pointed its camera across the field. This robot tracked the movements of a soccer ball and featured a directional firewire camera driven by the *XVision* machine vision software [10], frequently delivering steady state images of 320×240 true color format. Time delays between transmission of two images varied from 70ms to 200ms. Similar to [2], the input to the LSM was provided by 48 sensors arranged in a 2D array(8×6), so the recorded images had to be preprocessed.
For each image, the x and y coordinates as well as the radius of the ball were extracted using an existing tracking package. The ball is detected within an image by simple color-blob-detection leading to a binary image of the ball. This simple image preprocessing is applicable since all objects on the RoboCup-field are color-coded and the ball is the only red one. The segmented image is presented to the 8 times 6 sensor field of the LSM (see Figure 5.1).
The activation of each sensor is equivalent to the percentage of how much of the sensory area is covered by the ball. To group contiguous ball movements from the moment the ball entered the robot's field of view up to the point the ball left it (*movies*), we wrote an add-on for the XVision environment. Tracking information is stored line by line for each image containing coordinates, radius and time elapsed since start. Given this movie recording equipment, it was possible to record several hundreds of raw data movie files.

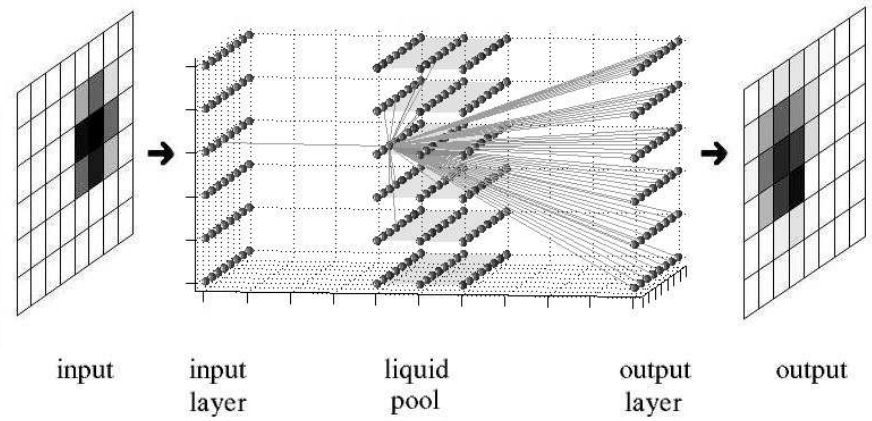


Figure 5.1: Architecture of the experimental setup depicting the three different pools of neurons and a sample input pattern with the data path overview. Example connections of a single liquid neuron are shown: input is received from the input sensor field on the left hand side and some random connection within the liquid. The output of every liquid neuron is projected onto every output neuron (located on the most right hand side). The 8x6x3 neurons in the middle form the "liquid".

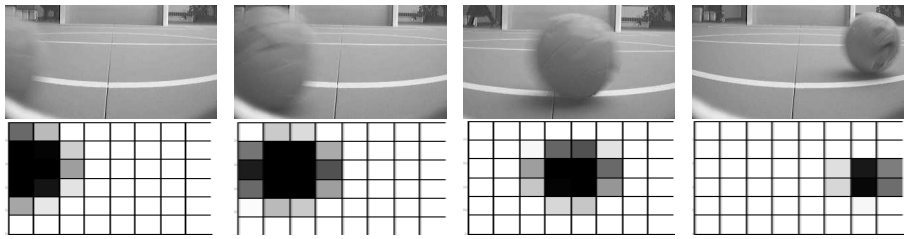


Figure 5.2: Upper Row: Ball movement recorded by the camera. Lower Row: Activation of the sensor field.

A set of 674 video sequences of the ball was recorded rolling with different velocities and directions across the field. The video sequences have different lengths and contain images in 50ms time steps. These video sequences are transfered into the equivalent sequences of activation patterns of the input sensors. Figure 5.2 shows such a sequence. The activation sequences are randomly divided into a training set (85%) and a validation set (15%) used to train and evaluate the prediction. Training and evaluation is conducted for the prediction of 1 time step (50ms), 2 time steps (100ms) and 4 time steps (200ms) ahead. The corresponding target activation sequences are simply obtained by shifting the input activation sequences 1,2 or 4 steps forward in time.

5.1.1 Simulation and Training

Simulation for the training set is carried out sequence-by-sequence: for each collected activation sequence, the neural circuit is reset, input data are assigned to the input layer, recorders are set up to record the liquid's activity, simulation is started, and the corresponding recorded liquid activity is stored for the training part. The training is performed by calculating the weights of all static synapses connecting each liquid neuron with all output layer neurons using linear regression[1]. Let $\{m_{i,j}[n]\}$ be the activation sequence for sensor i out of the 8x6 sensor pool for one sequence j out of the training set. Let $\{x_i[n]\} = \{\{m_{i,1}\}, \{m_{i,2}\}, ..., \{m_{i,N}\}\}$ be the concatenation of all N activation sequences of the training set. With the expected output sequence $\{y_i[n]\} = \{x_i[n+p]\}$ consisting of the input sequence shifted by p prediction time steps, $\underline{w_i}$ as the weights of output neuron i and $\{\underline{psp[n]}\}$ as the sequence of postsynaptic potentials of all liquid neurons recorded during simulation, the regression writes as

$$\underline{w_i} = regress(\{y_i[n]\}, \{\underline{psp[n]}\})$$

for one neuron in the output layer. The least squares method is used for approximation. To get a more robust modeling, white noise was added to $\{\underline{psp[n]}\}$.

Analogous to the simulation with the training set, simulation is then carried out on the validation set of activation sequences. The resulting output neuron activation sequences are stored for evaluating the network's performance.

The quality of the prediction (only the evaluation set is considered) is assessed by computing the correlation coefficient between target and prediction as illustrated in Equation 5.1, where *target* equals one target frame and *prediction* equals the corresponding predicted frame. The variable n denotes the number of all frames considered.

For one simulation (fixed # liquid neurons, λ and Ω) one correlation coefficient is computed that represents the mean correlation coefficient:

$$CC = \frac{1}{n}\sum_{i=1}^{n} cc_i(target(:,i), prediction(:,i)) \tag{5.1}$$

The correlation coefficient measures the *linear* dependency of two random variables. If the value is zero two variables are not correlated. The higher the coefficient the higher the probability of getting a correlation as large as the observed value without coincidence involved.

5.1.2 Finding Optimal Parameter Combination

As stated in Section 4.2 the connection strength w and connection probability λ represent two important parameters regarding the LSM. Various combinations of these two paramters exist. This Subsection aims at finding the optimal ones.

[1] In fact also the injection current I_{inject} for each output layer neuron is calculated. For simplification this bias is treated as the 0^{th} weight.

Findings from [13] provide the necessary calculations to be able to identify the optimal parameter regions. In [13] the same prediction task is carried out applying the LSM to the recorded, noisy real-world data (see Figure 5.1). Each parameter combination yields a certain prediction quality expressed by a correlation coefficient ranging from zero to one or in other words from no prediction to an almost perfect prediction.

In the following the procedure is described how the correlation coefficient landscape plots are obtained. In conclusion, a theoretical explanation for the optimal region of parameter combinations is given.

Figure 5.3 shows an input frame that is fed into the LSM, the corresponding target needed for the linear regression, the prediction of the LSM and finally the absolute error. A problem which arises if the absolute error (respectively the mean absolute error if several predictions are to evaluated by one value) is used for evaluation is that also networks with nearly no output activation produce a low mean absolute error - because most of the neurons in the target activation pattern are not covered by the ball and therefore they are not activated leading to a low average error per image. Consequently, the mean correlation coefficient (see Equation (5.1)) is calculated.

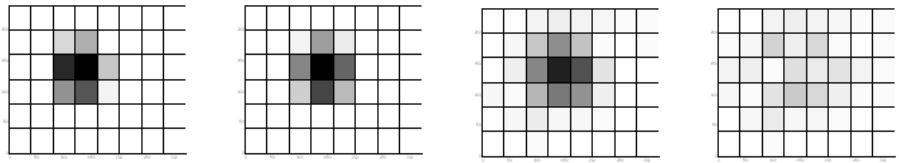


Figure 5.3: Sensor activation for a prediction one timestep ahead. Input activation, target activation, predicted activation and absolute error (left to right).

In Figure 5.4 the quality of the results by means of the correlation coefficients for the prediction of one timestep (50ms) and two timesteps (100ms) ahead are shown for various parameter combinations. The parameter values range for both landscapes from 0.1 to 5.7 for Ω and from 0.5 to 5.7 for λ.

Certain parameter combinations (e.g. the light shaded region in Figure 5.4) yield better results than others. In [22] it is shown that cortical microcircuits do operate at the edge of chaos, a region that is located at the boundary between ordered and chaotic behavior. It turns out that in the study of neural systems this research direction is of special interest, since dynamic systems exhibit enhanced computational power in this region.

At this critical line, the antagonistic effects of the fading memory property [24] and the separation property [29] reach an equilibrium state. The ordered phase is typically characterized by the fading memory property - small differences in the network state tend to decrease rapidly over time. In chaotic networks these differences are highly amplified and do not vanish. In the landscape plots (see

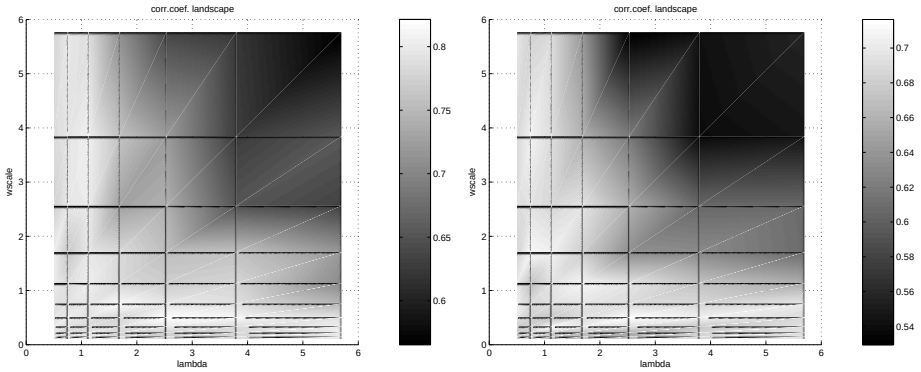


Figure 5.4: Correlation coefficient landscape for a prediction of one timestep (50ms) ahead on the left plot and of two timesteps (100ms) on the right plot. $\Omega(wscale) \in [0.1,5.7]$, $\lambda \in [0.5,5.7]$.

Figure 5.4, regions can be spotted whose parameter combinations yield optimal performance according to the task - the lighter the better.

The regions of good results remain the same throughout the prediction tasks. Therefore it can be assumed that the network operates at the edge of chaos when initialized with the corresponding parameter combinations. Similar parameter regions are obtained in Legenstein et al. [22] when performing a spike train classification task using a LSM.

5.2 Applying a Stacked Architecture

The introduction of a stacked architecture containing two layers intends to provide a digitalized form, i.e. equivalent computational properties, of a recurrent neural network described in Chapter 4. The proposed architecture incorporates the three previously described basic computational operations of a cortical microcircuit translated into the digital world.

The first layer serves mainly as preprocessing unit to provide the second layer with the means to carry out the basic operations. Instead of analog fading memory, a form of a digital tapped delay line in combination with a non-linear kernel is used in the second layer. The kernel generates a large number of non-linear combinations of the input at different locations in time and space.

In the following I will give a detailed description of both layers and their responsibilities. In Figure 5.2.2 the process flow of the proposed architecture is illustrated.

5.2.1 First Layer

As already mentioned the first layer serves as a kind of preprocessing. The dimensionality of the input data is reduced. Kernel PCA and a greedy derivative

are applied which have been described in Sections 3.1.3 and 3.2.1 respectively.

Kernel PCA [6] provides the means to reduce the input dimensionality. Standard kernels (the polynomial kernel, the RBF kernel and the sigmoidal kernel) define the higher dimensional feature space, the input data is first mapped into. Principal Component Analysis is carried out on the new feature space. Resulting eigenvalues and corresponding eigenvectors are then sorted in descending order beginning with the largest. By projecting the transformed input data onto the first eigenvectors, principal components, i.e. features, are obtained in an unsupervised manner. By taking a lower number of eigenvectors as input dimensions into account the number of dimensions can be reduced.

Parameter Setting in the First Layer

The next question is: "How many dimensions suffice to loose as little information as possible while compressing the input data as good as possible?"
A first approach is to analyze the decay of the eigenvalues which is depicted in Figure 5.5 for all three kernels.

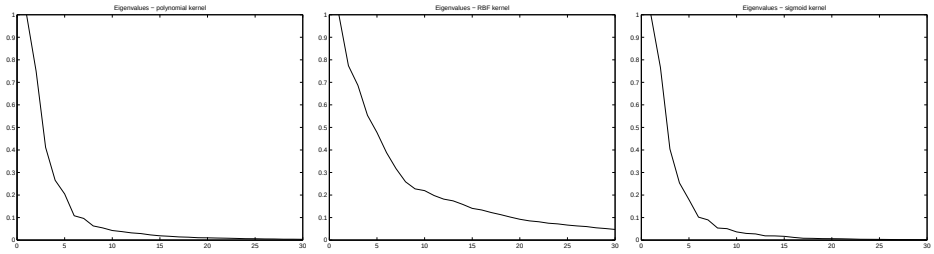


Figure 5.5: The decay of the eigenvalues corresponding to the polynomial kernel (left), the RBF kernel(middle) and the sigmoidal kernel(right). The x-axis is segregated into steps of length 5. So starting at zero the second value on the x-axis denotes the tenth eigenvalue and the fourth value the twentieth eigenvalue.

Instead of estimating the adequate number of dimensions, the reconstruction of the original input data is attempted by applying linear regression to the extracted features. The obtained coefficients from a training data set are then used to reconstruct the evaluation data set. In Table 5.1 the correlation coefficients comparing the original the reconstructed data are listed indicating that the usage of 10 features reconstructs the input data sufficiently.

It should be noted that the correlation coefficients in Table 5.1 correspond to the optimal parameter for each kernel. The optimal kernel parameters are obtained by parameter sweeps over sensible values for each kernel relating to the reconstruction of the evaluation data.

For the polynomial kernel a degree $d = 3$ suffices and the range of θ is $[0.5 : 5]$, for the RBF kernel the range of $\frac{1}{2\sigma}$ is $[0.5 : 2]$ and for the parameter κ of the

Number of Features	Correlation Coefficients		
	polynomial kernel	RBF kernel	sigmoidal kernel
30	0.9972	0.9619	0.9949
20	0.9846	0.9424	0.9834
10	0.9274	0.8635	0.9182

Table 5.1: The correlation coefficients denote the reconstruction quality of the input data after the first layer.

sigmoidal kernel the range is $[0.1 : 1.5]$ (see 2.2 for a kernel overview). Optimal parameter values for each kernel can be taken from Table 5.2.

Optimal Kernel Paramters		
polynomial kernel	RBF kernel	sigmoidal kernel
θ / d	σ	κ/ θ
1.5/ 3	0.75	0.1/ 0

Table 5.2: Optimal parameter values for each kernel regarding the first layer.

As a consequence, in the experiments the polynomial kernel with parameters $d = 3$ and $\theta = 1.5$ is used within the first layer and 10 extracted features are taken along to the second layer.

Remarks on Greedy KPCA

As a representative of a more complex method greedy KPCA (see Section 3.2.1) is applied for dimensionality reduction in the first layer as well. Based on kernel PCA, it aims at approximating the training set by taking only the most significant samples into account. According to the findings from the preceding section, 10 training samples are selected. Experiments show that the reconstruction of the input data is nearly as good as with normal kernel PCA.

Thus greedy KPCA represents an alternative to normal KPCA. In addition greedy KPCA does not require eigenvalue decomposition to select the representative training samples as it is the case with KPCA. Consequently, time complexity reduces from $O(n \times n)$ to $O(n \times m^3)$, where n denotes the number of trainings samples and m the number of samples to select. However, if the proper coverage of the input distribution requires a larger number of selected training samples, the method gradually loses its benefit.

5.2.2 Second Layer

Arranging the input

As already previously mentioned the input to the second layer (no matter if training or evaluation data) is constructed using a time window that slides over a tapped delay line (TDL). When trying to imagine the appearance of the TDL, it may be advantageous to do so the following way:

Consider the extracted feature sets of the first layer $\mathbf{y}_1, \ldots, \mathbf{y}_n$ each containing m features, thus $\mathbf{y}_i \in \mathbb{R}^{10}$, in this case. Now arrange these sets one after another to form a large vector $\in \mathbb{R}^{(n*m)}$, the TDL. In the second layer temporal integration happens by combining feature sets of the first layer, thus feature sets of several time steps form a new input vector for the second layer. For example, I would like that an input vector of the second layer encompasses two time steps. The first (compound) vector **cv** then reads $\mathbf{cv}_1 = ({\mathbf{y}_1}^\mathsf{T}, {\mathbf{y}_2}^\mathsf{T})^\mathsf{T}$. To form a connection with the TDL, imagine a window of size (2*10, i.e. 2 time steps combined, 10 features each) moving along the TDL. The next step of the window translates into shifting the window by an offset (that offset is defined by the size of one feature set, 10 in my case). The second input vector results then from shifting the window by the offset, thus $\mathbf{cv}_2 = ({\mathbf{y}_2}^\mathsf{T}, {\mathbf{y}_3}^\mathsf{T})^\mathsf{T}$. $\mathbf{cv}_3$ would be $({\mathbf{y}_3}^\mathsf{T}, {\mathbf{y}_4}^\mathsf{T})^\mathsf{T}$ and so on (see Figure 5.6 for an illustration).

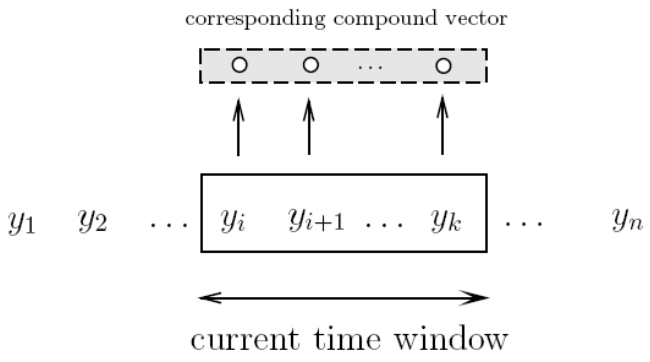


Figure 5.6: By moving a time window along the TDL the corresponding compound vector consisting of feature sets **y** can be built

Note, however, that there is no overlapping of video sequences within the input vectors for the second layer. Similarly, the LSM is reset when a new sequence is fed into the network.

The target vectors for the linear regression on top of the stacked architecture are built the following way. (Their dimensionality remains the same ($\mathbb{R}^{48}$)). Imagine at first, that there is no temporal integration and the number of prediction steps is determined to be two. The corresponding target vector for an

input vector would be the input vector shifted by two steps into the future.

However, it has to be considered that the input vectors of the second layer are compound vectors. Accordingly the targets have to be put together appropriately. Let's assume that a prediction task of two time steps ahead has to be accomplished and one compound vector of the second layer includes two feature sets (hence a time window of two time steps). The corresponding target vector, that is needed for the linear learning rule afterwards, will be the input vector shifted by three time steps into the future (one because the input vector of the second layer has information about that additional time step and another two time steps to fulfil the prediction requirements).

Fading Kernels in the Second Layer

Kernel PCA is applied to the training data that have already been preprocessed in the first layer (see Subsection 5.2.1). As described in Section 3.1.3, kernel PCA implicitly performs PCA in a high-dimensional feature space that is nonlinearly related to the input space. Calculations in that possibly infinite dimensional space (the feature space) are due to the application of kernel functions in the input space that corresponds to computing the dot-product in that feature space. In the second layer RBF kernels as well as polynomial kernels are applied that incorporate a *fading* property. In the LSM paradigm an analog fading memory is responsible for evaluating events in terms of the time of their emergence. Older events become less and less relevant (following an exponential decay) and finally fade away. This analog fading memory is translated into the digital world by temporal integration in combination with a fading kernel. Temporal integration is achieved by concatenation of several time steps. In order to obtain a fading kernel (RBF and polynomial), a weight vector is added to the kernel function. The weight vector equals the speed older parts of one enlarged vector fade away, i.e. becoming more and more insignificant. Different fading speeds are applied within the experiments (see Section 6.4 for the results) , e.g. linear and exponential decays).

The resulting, new kernels have to be valid Mercer kernels to be applicable in the stacked architecture.

Valid Mercer Kernels?

Mercer's Theorem [14] states that if $k(\mathbf{x}, \mathbf{x}')$ is a Mercer Kernel, then there exists a Hilbert space $\mathcal{H}$ of real valued functions defined on $\mathcal{X}$ and a feature map:

$$\Phi : \mathcal{X} \to \mathcal{H}, \mathbf{x} \mapsto \Phi(\mathbf{x}) \tag{5.2}$$

such that

$$\langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle = k(\mathbf{x}, \mathbf{x}') \tag{5.3}$$

where $\langle \cdot, \cdot \rangle$ denotes the inner product.

The new kernels, the fading RBF kernel and the fading polynomial kernel, are defined as follows. The fading RBF kernel reads:

$$k(\mathbf{x}, \mathbf{x}') = exp(-\frac{||\langle(\mathbf{x} - \mathbf{x}'), \mathbf{w}\rangle||^2}{2\sigma^2}) \tag{5.4}$$

where $\sigma > 0$ and the fading polynomial kernel:

$$k(\mathbf{x}, \mathbf{x}') = (\left\langle \langle \mathbf{x}, \mathbf{w} \rangle, \langle \mathbf{x}', \mathbf{w} \rangle \right\rangle) + \Theta)^d \tag{5.5}$$

where d is the degree of the polynomial and Θ is a constant.

The new *fading* kernels have to be valid kernels to be able to exploit Mercers's Theorem.

Proof Normal RBF and polynomial kernels always meet Mercer's conditions[2]. The new, fading kernels (see Equations (5.4) and (5.5) respectively) are extended by a weight vector $\mathbf{w}$ ($w_i \in (0, 1]$) that corresponds to the *fading speed*. The two vectors $\mathbf{x}, \mathbf{x}' \in \mathbb{R}^n$ are multiplied by the constant vector $\mathbf{w} \in \mathbb{R}^n$ corresponding to a weighting of the vector elements. The weighting takes place before the actual translation into a higher dimsional space.
Thus, if the normal RBF and polynomial kernel are Mercer kernels, the new fading kernels are valid Mercer's kernels as well.

Parameter Setting in the Second Layer

The optimal kernel parameters for the second layer are obtained again by parameter sweeps over sensible values for each kernel regarding the prediction task. In other words, parameter values yielding best prediction results are selected. Consequenly, in the stacked architecture, following parameter settings are used:

Optimal Kernel Paramters	
polynomial kernel	RBF kernel
θ / d	σ
1.5/5	0.7

Table 5.3: Optimal parameter values for the polynomial and RBF kernel applied in the second layer.

[2]Mercer's Conditions:

1. $k(\mathbf{x}, \mathbf{x}')$ is continuous.
2. $k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x}', \mathbf{x})$
3. $k(\mathbf{x}, \mathbf{x}')$ is positive definite.

Feature Extraction

By solving the eigenvalue problem stated in Equation (3.14), eigenvalues ($\lambda > 0$) and corresponding eigenvectors ($\neq$ trivial solution) are obtained. Thus the underlying structure of the training data is revealed, lying in the directions of the eigenvectors. Feature extraction is carried out as explained in Equation (3.18). At first the evaluation vector has to be mapped into the feature space $\mathcal{H}$ and is projected onto the eigenvectors.

Learning the Prediction

One of the enormous benefits of applying kernel functions is that the data is mapped into a high dimensional space. The kernel generates a large number of non-linear combinations of the input data, thus enabling the application of simple learning rules, i.e. separating hyperplanes in case of classification.

By exploiting the projection of the data into a higher dimensional space, simple learning algorithms can be applied and are sufficient almost in any case experienced so far. Linear regression serves as the learning algorithms on top of the stacked architecture.

The number of extracted features per second layer input vector varies between 20 and 50 features yielding better results the more features used. Usually it does not make sense to work with more than 50 features, since the results do not become better any more.

After having performed the regression, each target vector from the evaluation set is approximated by the learnt regression coefficients (see Equation 5.6) from the training set and the corresponding features resulting in a prediction vector.

$$coefficients = regress(target, features) \tag{5.6}$$

The quality of the prediction (only the evaluation set is considered) is assessed by computing the correlation coefficient between target and prediction. For one simulation - a simulation encompasses 150 training sequences and 100 evaluation sequences - one correlation coefficient is computed that represents the mean correlation coefficient (in the same way as with the LSM approach):

$$CC = \frac{1}{n}\sum_{i=1}^{n} cc_i(target(:,i), prediction(:,i)) \tag{5.7}$$

Hence, a correlation coefficient is calculated for each predicted frame and the corresponding target frame in the evaluation set. Afterwards the mean correlation coefficient according to Equation 5.7 is computed.

5.3 Artificial Data Set

In order to apply the stacked architecture to another data set, sequences are artificially generated similarly to [2]. The sequences contain balls of varying size and moving at different speeds. For each sequence, the speed, the size and

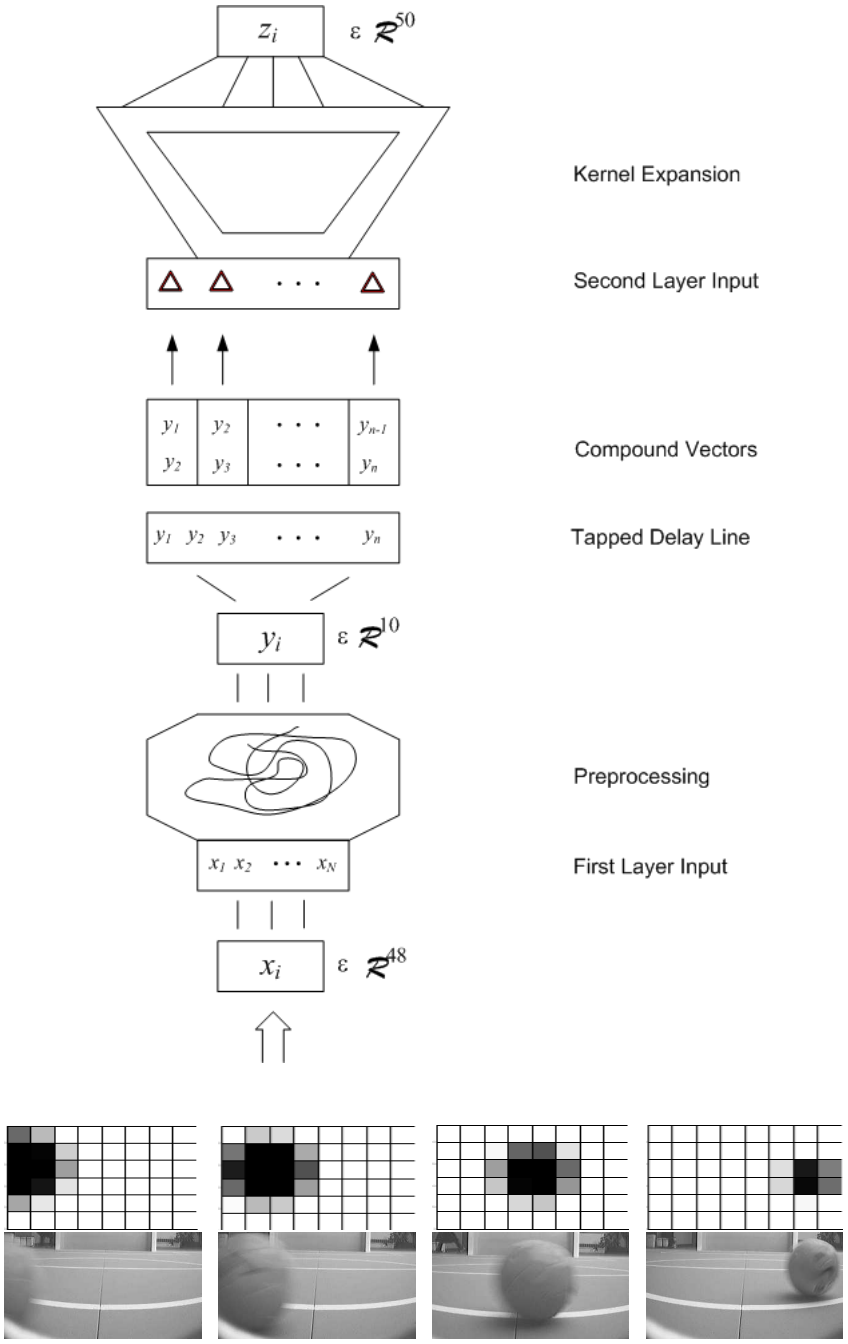


Figure 5.7: The new stacked architecture is illustrated. A pass through starts with the input frames on the bottom. The raster frames are converted into vectors and the dimension of the vectors is reduced in the first layer. Afterwards the resulting reduced vectors are concatenated and fed into the second layer. Kernel PCA using a fading kernel is applied and a feature vector is extracted that is used by the linear regression to perform a prediction task.

the direction of the ball are randomly chosen. In Figure 5.8 some exemplary sequences are shown.

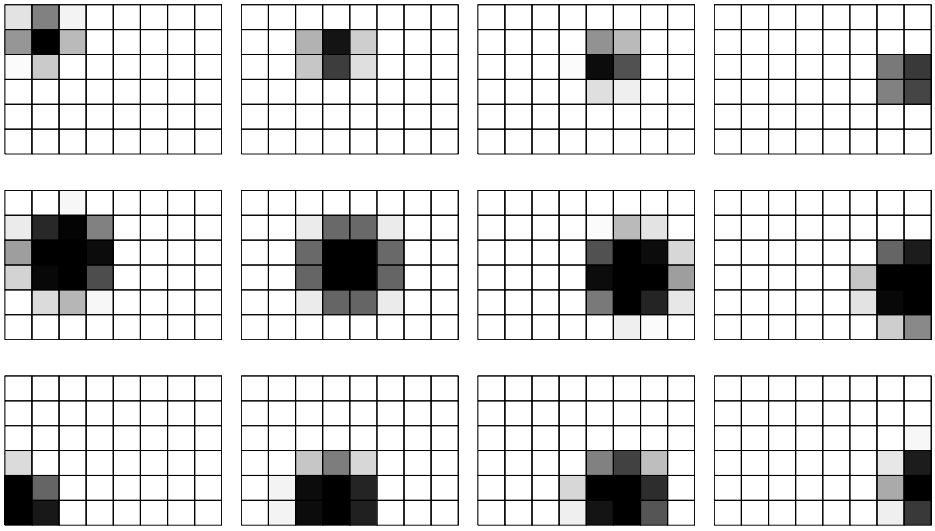


Figure 5.8: Three artificially generated sequences that are fed into the stacked architecture.

Differences to the real-world data set shall be emphasized. The most important difference is the stable radius throughout the sequences. The artificial data is merely two-dimensional, there is no perspective as with the real world data set.

Secondly, there are only three different speeds available when generating the artificial data set. The speed variety within the real world data set is much richer.

Thirdly, the artificially generated data is smooth regarding the curves of the x,y coordinates. Due to the admitting of perspective and the processing of the camera, coordinates and radius are not smooth but tend to be sometimes erratic.

The above mentioned differences are reflected in the prediction results that are discussed in Section 6.3.3.

Chapter 6

Results

In this chapter the results of the prediction tasks are presented. The task requires the prediction of the ball position (see Figure 5.2 for examples of a ball position) one to four time steps ahead by taking into account its trajectory seen so far.

At first, the findings of the LSM approach, applied to the noisy real-world data set, are listed in Section 6.1. In Section 6.2 results of the stacked architecture are presented where the prediction task is performed with both data sets - the real-world and the artificially generated data set. Additionally, experiments are carried out with the fading kernel in the second layer as well as with the normal, non fading version.

Section 6.3 compares some of the approaches that are mentioned above, e.g., fading versus non-fading and LSM versus stacked architecture. This chapter is concluded with Section 6.4 that reports experiences with different parameter settings during experiments with the stacked architecture.

The results are expressed in correlation coefficients that correspond to the mean correlation coefficient as described in Equation 5.1, i.e., the correlation coefficient is calculated for every target frame and its corresponding predicted frame. Afterwards the mean value is output. For the linear regression the dimension of the feature vector, that is output by the second layer per time step, varies between 20 and 50 features in case of the stacked archticture and is fixed at 144 features in case of the LSM.

In all experiments, 10 random splits of the input data are used for each parameter setting. The resulting correlation coefficients are averaged. The size of the input data encompasses 150 training and 100 test sequences in each trial. As a remark, taking 150 sequences corresponds to a kernel matrix of approximately (4000×4000) in the first layer. In Section 7.1 a potential enlargement of the training data size is discussed by taking into account online kernel methods.

The prediction results are presented visually as well. The manner of representation is always the same. On the top there is one sequence of target frames. That sequence is to be predicted given an input sequence that starts one, two,

three or four time steps earlier. Presented this way, the visual quality of the prediction can be compared more easily. The larger the prediction ahead, the worse the result quality.

6.1 LSM Results

In Table 6.1 the obtained correlation coefficients are listed. The parameter combinations of λ and w are selected according to the optimal behaviour of the LSM in this region as stated in Section 5.1.2.

Parameter Combination	Correlation Coefficients			
(λ, w)	One	Two	Three	Four
$\lambda = 2$, $w = 1$	0.9122	0.8716	0.8023	0.7255
$\lambda = 1$, $w = 0.5$	0.9117	0.8662	0.7961	0.7204

Table 6.1: Correlation Coefficients for one, two, three and four predicted time steps ahead corresponding to the optimal parameter combinations of the LSM.

The performance decreases, i.e. lower correlation coefficients are achieved, when the task gets harder, i.e. the number of time steps to predict increases. Mean correlation coefficients for predictions of one (50ms), two (100ms), three(150ms) and four (200ms) time steps ahead are listed in Table 6.1.

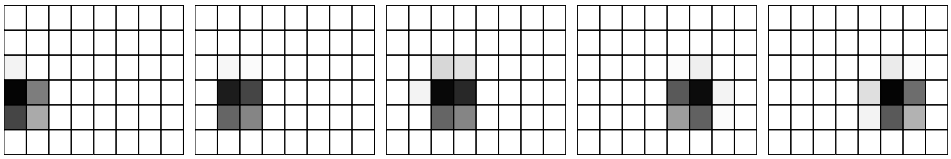


Figure 6.1: Target frames for predictions of one, two, three and four time steps ahead.

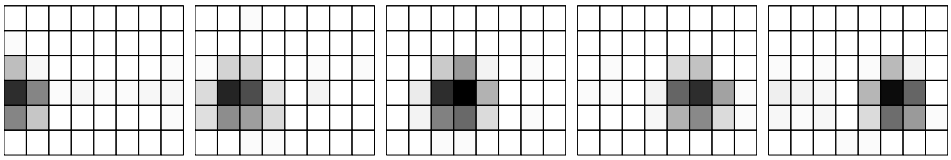


Figure 6.2: Predicted frames for the prediction of one time step ahead.

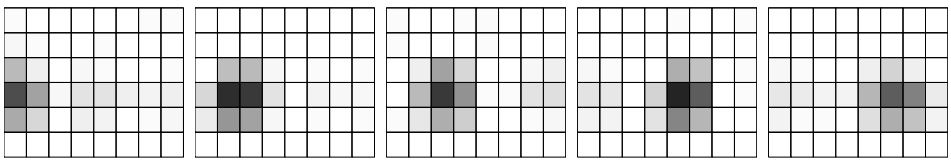


Figure 6.3: Predicted frames for the prediction of two time steps ahead.

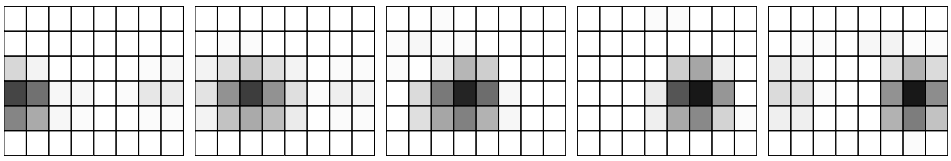


Figure 6.4: Predicted frames for the prediction of three time steps ahead.

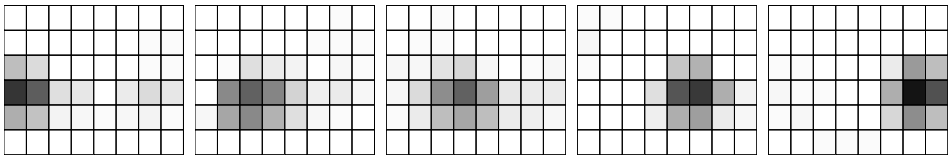


Figure 6.5: Predicted frames for the prediction of four time steps ahead.

In Figures 6.1 to 6.5, prediction results for 50ms, 100ms, 150ms and 200ms are visually presented. For a better comparison, the target frames are the same for all prediction tasks. Accordingly, the corresponding input frames start one, two, three and four time steps earlier. The activation sequence starts at the left side and ends at the left side. The decay of prediction quality that is reflected in the decreasing correlation coefficients is visually confirmed. At first, the activation starts getting weaker. Finally, the position of the ball is not predicted as exactly as it used to be with fewer time steps to predict.

6.2 Stacked Architecture Results

The stacked architecture is first applied to the real-world data set, then to the artificial data set. Experiments are carried out with fading kernels as well as with non-fading kernels in the second layer.

For all experiments regarding the stacked architecture, the polynomial kernel is chosen for the dimensionality reduction procedure in the first layer as stated in Section 5.2.1 and 10 extracted features per time step are forwarded to the second layer. The compound vector encompasses two time steps resulting in 20 elements per vector. The fading speed is set as follows: the older vector (the first ten elements) is weighted half as much as the newer one.

6.2.1 Real World Data Set

Fading in Second Layer

In Table 6.2 the obtained mean correlation coefficients for the polynomial kernel in the second layer are listed. Kernel parameters are selected according to Section 5.2.2 to be $d = 5$ and $\theta = 1.5$.

Not surprisingly, the performance decreases, i.e. lower correlation coefficients are achieved when the task gets harder, i.e. the number of time steps to predict increases. In Figures 6.6 to 6.10, prediction results for 50ms, 100ms, 150ms and

Number of Features	Correlation Coefficients			
	One	Two	Three	Four
50	0.9500	0.8780	0.8038	0.7326
40	0.9479	0.8745	0.7977	0.7235
30	0.9440	0.8696	0.7904	0.7132
20	0.9360	0.8613	0.7799	0.6997

Table 6.2: Correlation Coefficients for one, two, three and four predicted time steps depending on the number of used extraced features during linear regression. The fading polynomial kernel ($d = 5$ and $\theta = 1.5$) was applied to the real data set.

200ms are visually presented using 50 features during linear regression. For a better comparison, the target frames are the same for all prediction tasks. Accordingly, the corresponding input frames start one, two, three and four time steps earlier.

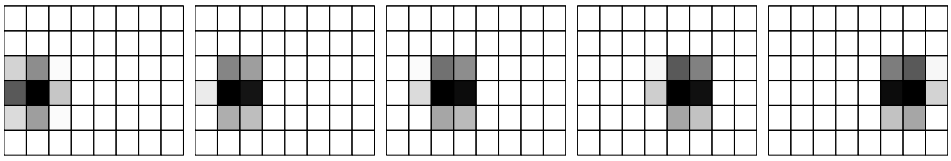


Figure 6.6: Target frames for predictions of one, two, three and four time steps ahead.

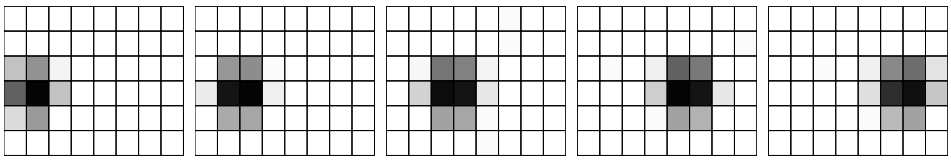


Figure 6.7: Predicted frames for the prediction of one time step ahead.

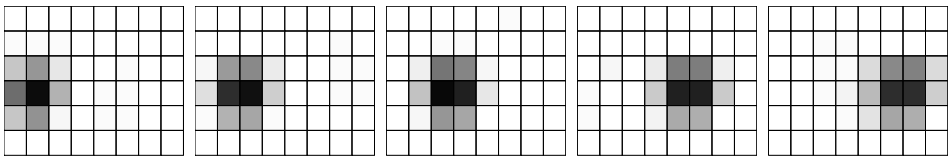


Figure 6.8: Predicted frames for the prediction of two time steps ahead.

In Table 6.3 the obtained mean correlation coefficients for the RBF kernel in the second layer are listed. Kernel parameters are selected according to Section 5.2.2 to be $\sigma = 0.7$.

In Figures 6.11 to 6.15 , prediction results for 50ms, 100ms, 150ms and 200ms are visually presented using 50 features during linear regression. For a better

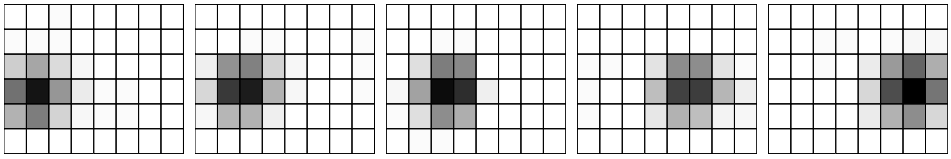


Figure 6.9: Predicted frames for the prediction of three time steps ahead.

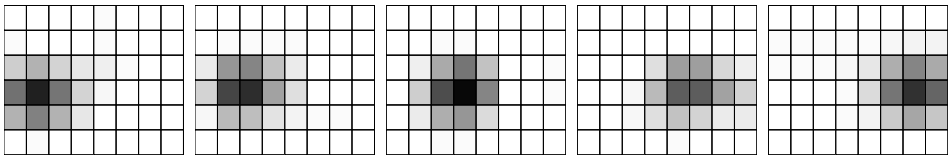


Figure 6.10: Predicted frames for the prediction of four time steps ahead.

Number of Features	Correlation Coefficients			
	One	Two	Three	Four
50	0.9514	0.8820	0.8111	0.7439
40	0.9481	0.8784	0.8070	0.7397
30	0.9434	0.8735	0.8015	0.7327
20	0.9208	0.8316	0.7415	0.6594

Table 6.3: Correlation Coefficients for one, two, three and four predicted time steps depending on the number of used extraced features during linear regression. The fading RBF kernel ($\sigma = 0.7$) was applied to the real data set.

comparison, the target frames are the same for all prediction tasks. Accordingly, the corresponding input frames start one, two, three and four time steps earlier.

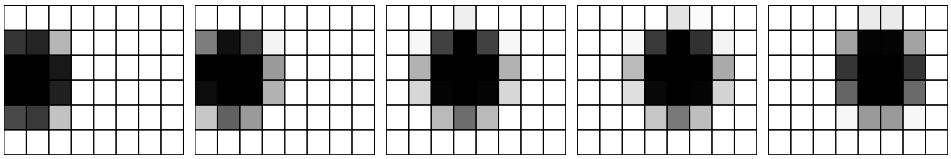


Figure 6.11: Target frames for predictions of one, two, three and four time steps ahead.

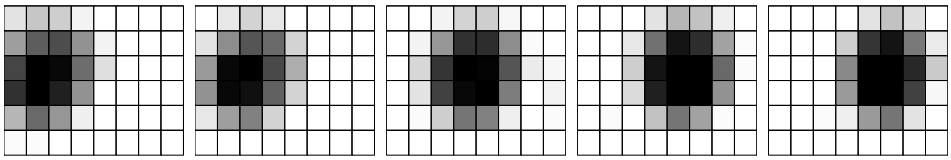


Figure 6.12: Predicted frames for the prediction of one time step ahead.

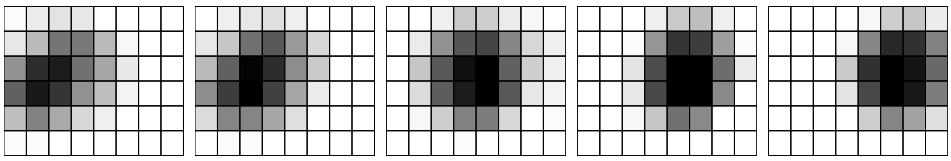


Figure 6.13: Predicted frames for the prediction of two time steps ahead.

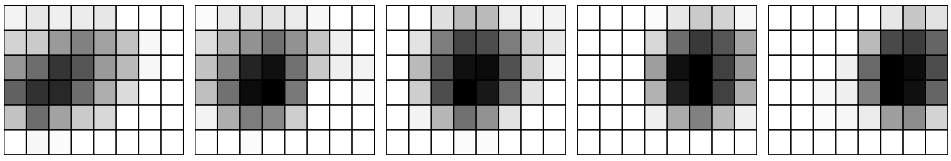


Figure 6.14: Predicted frames for the prediction of three time steps ahead.

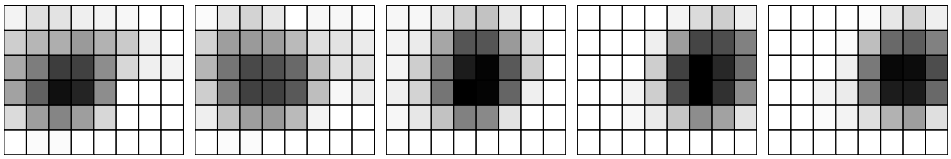


Figure 6.15: Predicted frames for the prediction of four time steps ahead.

The results indicate that both kernels can be used for the prediction task though the RBF kernel seems to yield slightly better results. The good correlation coefficients of both kernels are visually confirmed. The decay in the correlation coefficient quality is reflected by the fact that the predictions get more and more blurred with the growing number of time steps to predict. Especially when four time steps are to predict, the blurring of the activation can be identified clearly.

No Fading in Second Layer

In Table 6.4 the obtained mean correlation coefficients for the polynomial kernel in the second layer are listed. Kernel parameters are selected according to Section 5.2.2 to be $d = 5$ and $\theta = 1.5$. There is no fading in the second layer.

Number of Features	Correlation Coefficients			
	One	Two	Three	Four
50	0.9492	0.8783	0.8047	0.7334
40	0.9467	0.8732	0.7961	0.7214
30	0.9429	0.8685	0.7891	0.7116
20	0.9340	0.8596	0.7786	0.6989

Table 6.4: Correlation Coefficients for one, two, three and four predicted time steps depending on the number of used extraced features during linear regression. The polynomial kernel ($d = 5$ and $\theta = 1.5$) was applied to the real data set.

In Table 6.5 the obtained mean correlation coefficients for the RBF kernel in the second layer are listed. Kernel parameters are selected according to Section 5.2.2 to be $\sigma = 0.7$.

Number of Features	Correlation Coefficients			
	One	Two	Three	Four
50	0.9452	0.8801	0.8125	0.7472
40	0.9399	0.8733	0.8043	0.7388
30	0.9344	0.8684	0.7997	0.7341
20	0.9046	0.8234	0.7418	0.6680

Table 6.5: Correlation Coefficients for one, two, three and four predicted time steps depending on the number of used extraced features during linear regression. The RBF kernel ($\sigma = 0.7$) was applied to the real data set.

The results indicate that both kernels can be used for the prediction task. Again the RBF kernel seems to yield slightly better results. It is curious to note the drop of prediction quality between the two kernels when using 30 features compared to using 20 features. The polynomial kernel succeeds in the task of predicting four time steps ahead when using only 20 features.

6.2.2 Artificial Data Set Results

The prediction task is now performed with artificially generated data (see Section 5.3). The parameter settings are identical to the ones of the real-world data experiments. By performing parameter sweeps in order to identify the optimal parameter settings, it turned out, that the same parameter settings can be applied as with the real-world data set.

Fading in Second Layer

In Table 6.6 the obtained mean correlation coefficients for the polynomial kernel in the second layer are listed. Kernel parameters are $d = 5$ and $\theta = 1.5$.

Number of Features	Correlation Coefficients			
	One	Two	Three	Four
50	0.9508	0.9346	0.9061	0.8675
40	0.9482	0.9331	0.9054	0.8672
30	0.9430	0.9225	0.8881	0.8423
20	0.9313	0.9029	0.8611	0.8097

Table 6.6: Correlation Coefficients for one, two, three and four predicted time steps depending on the number of used extraced features during linear regression. The fading polynomial kernel ($d = 5$ and $\theta = 1.5$) was applied to the artificial data set.

In Table 6.7 the obtained mean correlation coefficients for the RBF kernel in the second layer are listed. The selected kernel parameter is $\sigma = 0.7$.

Number of Features	Correlation Coefficients			
	One	Two	Three	Four
50	0.9440	0.8896	0.8236	0.7552
40	0.9361	0.8784	0.8076	0.7345
30	0.9273	0.8738	0.8061	0.7349
20	0.9163	0.8672	0.8029	0.7343

Table 6.7: Correlation Coefficients for one, two, three and four predicted time steps depending on the number of used extraced features during linear regression. The fading RBF kernel ($\sigma = 0.7$) was applied to the artificial data set.

Again it is not surprising, that the performance decreases, when the task gets harder. In other words, a worse prediction quality is achieved, when the number of time steps to predict increases.
However, in contrast to Section 6.2.1, the results indicate that the polynomial kernel is better suited for the artificial data set. It significantly outperforms the RBF kernel when the prediction encompasses more time steps.

In Figures 6.16 to 6.20 , prediction results for 50ms, 100ms, 150ms and 200ms are visually presented using 50 features during linear regression for the polynomial kernel. For a better comparison, the target frames are the same for all prediction tasks. Accordingly, the corresponding input frames start one, two, three and four time steps earlier.

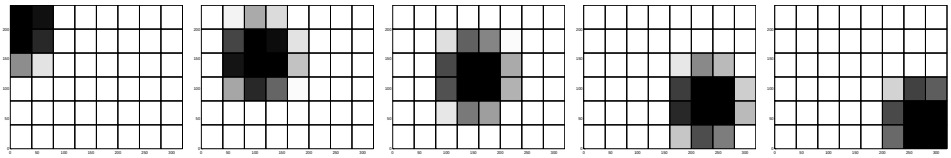


Figure 6.16: Target frames for predictions of one, two, three and four time steps ahead.

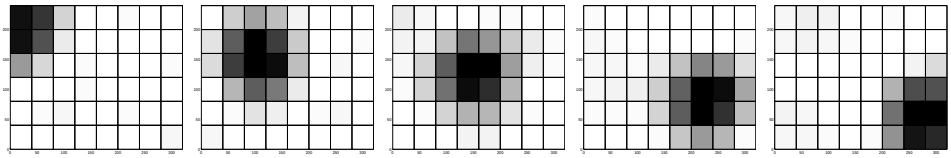


Figure 6.17: Predicted frames for the prediction of one time step ahead.

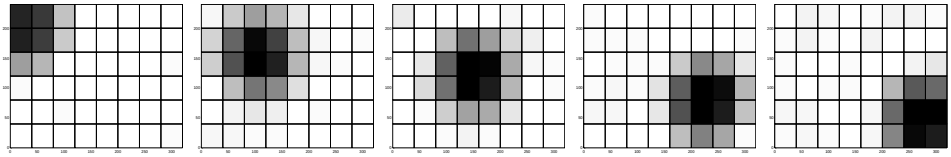


Figure 6.18: Predicted frames for the prediction of two time steps ahead.

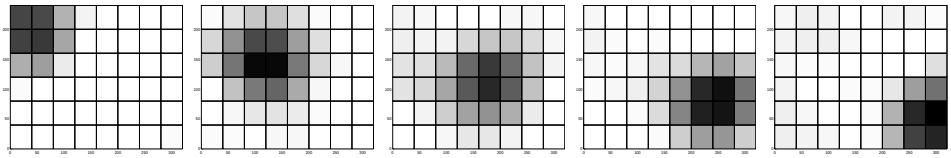


Figure 6.19: Predicted frames for the prediction of three time steps ahead.

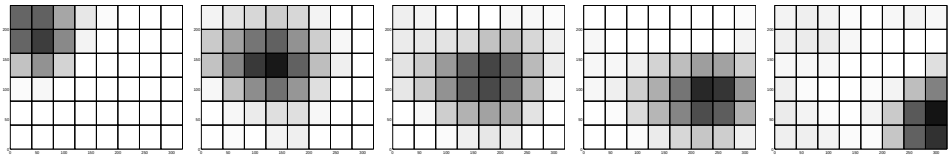


Figure 6.20: Predicted frames for the prediction of four time steps ahead.

No Fading in Second Layer

In Table 6.8 the obtained mean correlation coefficients for the polynomial kernel in the second layer are listed. Kernel parameters are $d = 5$ and $\theta = 1.5$. No fading takes place in the second layer.

Number of Features	Correlation Coefficients			
	One	Two	Three	Four
50	0.9501	0.9339	0.9058	0.8676
40	0.9473	0.9317	0.9040	0.8657
30	0.9410	0.9199	0.8854	0.8397
20	0.9278	0.8981	0.8559	0.8046

Table 6.8: Correlation Coefficients for one, two, three and four predicted time steps depending on the number of used extraced features during linear regression. The polynomial kernel ($d = 5$ and $\theta = 1.5$) was applied to the artificial data set.

In Table 6.9 the obtained mean correlation coefficients for the RBF kernel in the second layer are listed. The selected kernel parameter is $\sigma = 0.7$.
As in the preceding experiment, the results indicate that the polynomial kernel is better suited for the artificial data set. It significantly outperforms the RBF kernel when the prediction encompasses more time steps.

Number of Features	Correlation Coefficients			
	One	Two	Three	Four
50	0.9255	0.8623	0.7908	0.7198
40	0.9187	0.8563	0.7848	0.7134
30	0.9092	0.8511	0.7825	0.7129
20	0.8947	0.8407	0.7752	0.7079

Table 6.9: Correlation Coefficients for one, two, three and four predicted time steps depending on the number of used extraced features during linear regression. The RBF kernel ($\sigma = 0.7$) was applied to the artificial data set.

6.3 Comparison

The results of Sections 6.1 and 6.2 are compared to each other in this section. These comparisons answer questions like "Is the stacked architecture capable of exhibiting an equivalent computational behaviour as a recurrent neural network?, "Does fading in the second layer make sense?" and "Is there a difference between noisy, real-world data and artificially generated data?".

6.3.1 Stacked Architecture vs. LSM

The prediction task using the real-world data set is performed by the stacked architecture as well as by the LSM. In Table 6.10 the resulting correlation coefficients taken from Tables 6.1, 6.2 and 6.3 are shown.

Learning Model	Parameters	Correlation Coefficients			
		One	Two	Three	Four
Stacked Architecture	d =5, $\theta = 1.5$	0.9500	0.8780	0.8038	0.7326
Stacked Architecture	$\sigma = 0.7$	0.9514	0.8820	0.8111	0.7439
LSM	$\lambda = 2$, $w = 1$	0.9122	0.8716	0.8023	0.7255
LSM	$\lambda = 1$, $w = 0.5$	0.9117	0.8662	0.7961	0.7204

Table 6.10: Correlation Coefficients for one, two, three and four predicted time steps corresponding to the optimal parameter combinations of the LSM and the stacked architecture using the polynomial kernel (d =5, $\theta = 1.5$) and RBF kernel ($\sigma = 0.7$).

Comparing the results indicates that the stacked architecture meets the computational capabilities of the LSM. It even exceeds it when considering the correlation coefficients for four time steps ahead.

It is to mention that multi-tasking is feasable with stacked architecture as well. The features are extracted in an unsupervised manner and are thus not task

dependent.

Nevertheless there exist several benefits if using the LSM approach. One major difference concerns the number of training sequences which is limited by the kernel method itself. Because of those computational constraints only 150 sequences can be used for training resulting in matrices of approximately 4000 $\times$ 4000 elements that have to be processed. That limitation could be a disadvantage compared to the LSM, if not all possible input variations (ball trajectories) are covered in the training period. Furthermore the stacked architecture can only be applied offline, whereas the LSM provides *any-time* computing (see Section 4.1), thus as soon as input is fed into the network the corresponding output is available.

In Section 7.1 a potential enlargement of the training data size is discussed by taking into account kernel methods that include a greedy, sparse or iterative approach.

6.3.2 Fading vs. Non Fading

The prediction task is performed by the stacked architecture - one time incorporating the fading property in the second layer and one time without it. First, the results of the real-world data set are considered using 50 features for the linear regression. In Table 6.11 the results from Tables 6.2, 6.3, 6.4 and 6.5 are compared.

Applied Kernel	Fading	Correlation Coefficients			
		One	Two	Three	Four
polynomial kernel	yes	0.9500	0.8780	0.8038	0.7326
RBF kernel	yes	0.9514	0.8820	0.8111	0.7439
polynomial kernel	no	0.9492	0.8783	0.8047	0.7334
RBF kernel	no	0.9452	0.8801	0.8125	0.7472

Table 6.11: Correlation Coefficients for one, two, three and four predicted time steps corresponding to fading/ non-fading RBF kernel ($\sigma = 0.7$ and 50 features used) and polynomial kernel (d =5, $\theta = 1.5$ and 50 features used).

Regarding the real-world data set, it does not care whether the fading property is incorporated in the second layer. The question arises if fading is really needed in the second layer? Or does temporal integration suffice?

The answer is provided by considering the artifical data set. The results of the artificial data set are considered using 50 features for the linear regression. In Table 6.12 the results from Tables 6.6, 6.7, 6.8 and 6.9 are compared.

In case of the polynomial kernel the additional fading does not improve the results significantly whereas in case of the RBF kernel the fading does have an effect on the correlation coefficients.

Applied Kernel	Fading	Correlation Coefficients			
		One	Two	Three	Four
polynomial kernel	yes	0.9508	0.9346	0.9061	0.8675
RBF kernel	yes	0.9440	0.8896	0.8236	0.7552
polynomial kernel	no	0.9501	0.9339	0.9058	0.8676
RBF kernel	no	0.9255	0.8623	0.7908	0.7198

Table 6.12: Correlation Coefficients for one, two, three and four predicted time steps corresponding to fading/ non-fading RBF kernel ($\sigma = 0.7$ and 50 features used) and polynomial kernel (d =5, $\theta = 1.5$ and 50 features used).

In conclusion, including fading in the second layer does not always improve the obtained results and certainly depends on the task and the data set that is used. However, regarding the RBF kernel applied to the artificial data set a significant improvement can be noticed.

6.3.3 Real Data vs. Artificial Data

The last comparison involves both data sets - the real-world and the artificial data set. The following comparisons reflect the significant difference (see 5.3 Section for an overview) between the two data sets.

Data Set	Applied Kernel	Correlation Coefficients			
		One	Two	Three	Four
Real	polynomial kernel	0.9500	0.8780	0.8038	0.7326
Real	RBF kernel	0.9514	0.8820	0.8111	0.7439
Artificial	polynomial kernel	0.9508	0.9346	0.9061	0.8675
Artificial	RBF kernel	0.9440	0.8896	0.8236	0.7552

Table 6.13: Correlation Coefficients for one, two, three and four predicted time steps corresponding to fading RBF kernel ($\sigma = 0.7$ and 50 features used) and polynomial kernel (d =5, $\theta = 1.5$ and 50 features used). Both data sets are used in the experiments.

The results indicate that a much better prediction quality is achievable with the artificially generated data set using the polynomial kernel.

Data Set	Applied Kernel	Correlation Coefficients			
		One	Two	Three	Four
Real	polynomial kernel	0.9492	0.8783	0.8047	0.7334
Real	RBF kernel	0.9452	0.8801	0.8125	0.7472
Artificial	polynomial kernel	0.9501	0.9339	0.9058	0.8676
Artificial	RBF kernel	0.9255	0.8623	0.7908	0.7198

Table 6.14: Correlation Coefficients for one, two, three and four predicted time steps corresponding to RBF kernel ($\sigma = 0.7$ and 50 features used) and polynomial kernel (d =5, $\theta = 1.5$ and 50 features used). Both data sets are used in the experiments.

6.4 Additional Parameter Settings

The question arises if better results are achievable using other parameter settings, e.g. using a larger time window in the second layer or using an other weighting schema.

During the experimental phase a lot of different parameter settings and combinations of these settings were tested. A larger temporal integration was achieved by concatenating up to four input vectors after they were reduced in dimension. Different fading speeds, that correspond to different weighting schemas, were tested as well, including exponential decay.

The correlation coefficients that are reported in Tables 6.2 and 6.3 were not exceeded.

Chapter 7

Conclusion

A stacked architecture consisting of two layers has been presented that incorporates three basic operations, i.e. analog fading memory, non-linear kernel functions and linear readouts, to model the computational function of a cortical microcircuit. The presented approach can be seen as a digitalized form of a recurrent neural network, the LSM, it is compared to . Furthermore the stacked architecture has been applied to two different data sets.

The architecture seems to reflect properties that occur in biological organisms when sensory information is processed. The first layer serves as preprocessing unit that extracts salient features in order to compress the input data as much as possible without loosing essential information.

The second layer implements temporal integration by combining the extracted features of successive time steps. A non-linear kernel equipped with a fading property is applied to the compressed, temporally enlarged data generating non-linear combinations of items at different locations in space and now in time as well. Features are extracted in an unsupervised manner by using kernel PCA. The operation in high-dimensional spaces enables the application of simple learning algorithms such as linear regression in order to obtain the desired findings.

7.1 Next Experiments

According to the results in Chapter 6, the proposed stacked architecture can be applied successfully to computational tasks such as prediction. It is by all means comparable to the computational capabilities of the LSM. It sometimes even outperforms the LSM in the prediction task.

However, the limitations in processing large data sets pose a serious disadvantage. This problem needs to be solved, if the idea of the stacked architecture is to be further pursued in the future.

Possible solutions might be found by contemplating about replacing the standard kernel algorithms by methods that can handle larger data sets. In the following, some existing methods that propose iterative or greedy approaches

are considered.

Iterative Kernel Principal Component Analysis [19] results from kernelizing the General Hebbian Algorithm(GHA)([9], [26]) that represents an iterative method for PCA to solve the eigenvalue problem when the size of the covariance matrix is large. This means that the covariance matrix does not have to be computed and stored directly. Carrying out the resulting Kernel Hebbian Algorithm equals performing GHA in a high dimensional space.

Sparse Greedy Matrix Approximation [3] tries to find a good subset of basis functions, spanning a subspace. Only samples of this subset are used to approximate the original kernel matrix K by a matrix $\tilde{\mathbf{K}} = \mathbf{K} \cdot P_S$, where P_S denotes the projector on the subspace S.

Spectral Greedy Embedding by Ouimet [20], *Greedy Kernel Principal Component* by Franc [27] and *Kernel Matching Pursuit for Large Datasets* [21] follow similar approaches.

7.2 Open Questions

Including a separation into parallel pathways (see Chapter 1) in the stacked architecture as preprocessing step seems promising for further research. Several (preprocessing) units in the first layer can be used in parallel performing different algorithms for feature extraction. Each unit outputs an other representation of the data. This idea reminds of a combination of different kernels as it is proposed by Joachims [16] where a combination of two kernels

$$K = \lambda \cdot K_1 + (1 - \lambda) \cdot K_2 \tag{7.1}$$

enables the utilization of two different measures of similarity in a SVM classification task. A simple rule is infered based on the theoretical results. Combining kernels in a SVM is beneficial, if both kernels yield approximately the same prediction quality individually, while their support vectors are different. In an unsupervised learning environment, that rule could be interpreted as follows. A combination of kernels is advantageous, if the kernels perform equally well as individuals while extracting features that differ in their representation of the data.

The second layer is then capable of making use of these different views of the data structure by combining features of several preprocessing units and time steps.

In another research step, the stacked architecture could be enlarged by several layers, so that the two-layered approach described in this thesis is only a small part of it. Furthermore connections, that reflect backward from higher layers into lower ones, could provide a form of feedback. Such an implementation would certainly be more cortex-like and thus be potentially of advantage.

Appendix A

Source Code

A.1 Generating Data

A.1.1 generateData.m

```
%creates a cell array containing artificial raster data
%parameters are to be set in config.m
%resulting cell array is stored in artificialData.mat File

clear all
config;
designedCoord

numSequences = length(caDesignedCoord);

for j = 1:numSequences
    temp_matrix = caDesignedCoord{j};
    temp_size = size(temp_matrix);
    m_raster = [];

    for i = 1:temp_size(1)
        temp_x = temp_matrix(i,1);
        temp_y = temp_matrix(i,2);
        temp_r = temp_matrix(i,3);

        single_frame = convertInfos(temp_x,temp_y,temp_r);
        m_raster(i, 1:(N_Y*N_X) ) = single_frame;
        clear single_frame;
    end
    caDesignedRaster{j} = m_raster;
end

save artificialData.mat caDesignedRaster caDesignedCoord
```

A.1.2 config.m

```
%parameters for creating artificial raster data are to be set
%it is recommended to alter only NUM_DESIGNED_SESSIONS,
%ROWS_PER_DESIGNED_SESSION and the radius range

global caDesignedCoord
global caDesignedRaster

global RES_X
RES_X = 320;

global RES_Y
RES_Y = 240;

global N_X
N_X = 6;

global N_Y
N_Y = 8;

global D_X
D_X = RES_X/N_Y;

global D_Y
D_Y = RES_Y/N_X;

% amount of sessions to be created by designedSessions.m
global NUM_DESIGNED_SESSIONS
NUM_DESIGNED_SESSIONS = 100;

global R_DESIGNED
R_DESIGNED=50;

global R_DESIGNED_MAX
R_DESIGNED_MAX = 200;

%only starting number equals die fastest velocity
%now three different speeds are feasable [15 30 45]
global ROWS_PER_DESIGNED_SESSION
ROWS_PER_DESIGNED_SESSION = 15;
```

A.1.3 designedCoord.m

```
caDesignedCoord = cell(NUM_DESIGNED_SESSIONS, 1);

n_steps = ROWS_PER_DESIGNED_SESSION;
```

```
y_start = rand(NUM_DESIGNED_SESSIONS, 1) * RES_Y;
y_end = rand(NUM_DESIGNED_SESSIONS, 1) * RES_Y;

x_step =  (RES_X) / (n_steps-1);

%conventional way, where only one radius value is defined
% r = R_DESIGNED*ones(1,n_steps);

radiusArray = floor((rand(1,NUM_DESIGNED_SESSIONS)*80)+20);
velocityArray = ceil(rand(1,NUM_DESIGNED_SESSIONS)*3);

for session = 1:NUM_DESIGNED_SESSIONS
  %assign a radius out of randomly generated radius value array
  r = radiusArray(session) * ones(1,n_steps);

  if(mod(session,2))
    x = [0: x_step: RES_X];
  else
    x = [RES_X : -x_step : 0];
  end

  %y_end = RES_Y-y_start(session);


  y_step = (y_end(session) - y_start(session)) / (n_steps-1);

  tmpMatrix = [];
  tmpMatrix = [x; [y_start(session):y_step:y_end(session)]; r]';
  caDesignedCoord{session} =  [interp(tmpMatrix(:,1),velocityArray(session))
 interp(tmpMatrix(:,2),velocityArray(session))
 interp(tmpMatrix(:,3),velocityArray(session)) ];
end
```

A.1.4 convertInfos.m

```
function raster_record = convertInfos(posX, posY, radius)

%load necessary values from config.m
config

if radius < 5
  radius=5;
end

  width  = D_X/radius;
  height = D_Y/radius;
  myArea = width*height;
```

```
  startX = max(floor(((posX-radius))/D_X),0);
  endX   = min(floor((posX+radius)/D_X),N_Y-1);
  startY = max(floor(((posY-radius))/D_Y),0);
  endY   = min(floor(((posY+radius))/D_Y),N_X-1);

if (startX>endX) | (startY>endY) | (endY>=N_X) | (endX>=N_Y)
  [startX startY; endX endY]
end
raster_local = zeros(1, N_Y*N_X);

  for y = startY:endY
      for x=startX:endX
      xl = (x*D_X-posX)/radius;
      xr = xl+width;
      yb = (y*D_Y-posY)/radius;
      yt = yb+height;
      raster_local( 1, y*N_Y+x+1) = (getSensVal(xl,xr,yb,yt))/myArea;
     end
  end

raster_record = raster_local;
```

A.1.5 convertInfos.m

```
function OUT = getSensVal(xl,xr,yb,yt)

OUT = triCirc(xr,xl,yb)-triCirc(yt,yb,xr)-triCirc(xr,xl,yt)+triCirc(yt,yb,xl);

function OUT = triCirc(r,l,b) %entspricht x=r,y=l,z=b
    if ((b > 1) | (b < (-1)))
           OUT = ((atan2(b,r)-atan2(b,l))*0.5);
    else
            A = sqrt(1-b*b);
            if(((-A) <=l) & (r <= A))
                   OUT = (-0.5*b*(r-l));
            elseif (A <= l)
                    OUT = ((atan2(b,r)-atan2(b,l))*0.5);
            elseif (r <= (-A))
                    OUT = ((atan2(b,r)-atan2(b,l))*0.5);
            elseif ((-A <= l) & (l <= A) & (A <= r))
                    OUT = ((-b*(A-l) + atan2(b,r) - atan2(b,A))*0.5);
            elseif ((l<=-A) & (-A<=r) & (r<=A))
                    OUT = ((-b*(r + A) + atan2(b,-A) - atan2(b,l))*0.5);
            elseif ((l<=-A) & (A<=r))
                    OUT = ((atan2(b,-A) - atan2(b,l) - b*(A+A) + atan2(b,r) -
    atan2(b,A))*0.5);
```

```
            else
                    OUT = (0.0);
            end
        end
```

A.2 Apply Stacked Architecture

A.2.1 kernelArchitecture.m

```
%main file
clear all

initParameters
makeInput01

fprintf('Dimension Reduction using kernel KPCA(%s):\n',firstKernel)
performKpca
featureExtractionKpca

disp('Second Layer:')
makeInput02
disp('Fading Kernel PCA')
fprintf('Second stage kernel KPCA(fading %s):\n', secondKernel)
performKpca02Fading
featexExtractionKpca02Fading
regressionOneTimeStep
regressionTwoTimeSteps
regressionThreeTimeSteps
regressionFourTimeSteps
```

A.2.2 initParameters.m

```
%general parameters

lengthTrain = 150;
lengthTest = 100;

%up to four time steps is to be predicted
predTime = 4;

%how many time steps the compound vector contains
numTsSecond = 2;

%###
%first layer parameter
```

```
%determines the number of eigenvectors used to extract features after the
%first layer
numEvFirst = 30;

%kernel parameters for first stage
%kernel_type 1 -> rbf kernel_type 2 -> poly kernel_type 3 -> sigmoid
kernelTypeFirst = 2;

%kernel PCA
rbfVar01 = 0.9;
polyInh01 = 1.5;
sigVar01 = 0.1;

global firstKernel
switch kernelTypeFirst
    case 1
        firstKernel = 'radial basis function kernel';
    case 2
        firstKernel = 'polynomial kernel';
    case 3
        firstKernel = 'sigmoidal kernel';
    otherwise
        firstKernel = 'no kernel function specified';
end

%###
%second layer kernel parameters
kernelTypeSecond = 2;

%kernel_type 1 -> rbf kernel_type 2 -> poly kernel_type
rbfVar02 = 0.9;
polyInh02 = 1.5;

%determines the number of eigenvectors used to extract features after the
%second layer
numEvSecond = 50;

%defines the number of eigenvectors used
evFirstUsed = 10;

%offset of the sliding window
step = evFirstUsed;

%length of a second layer input vector
lengthVector02 = evFirstUsed * numTsSecond;

%notice the different weight vectors regarding the used kernel
```

```
fadingWeights = ones(lengthVector02,1);
fadingWeights(1:10) = fadingWeights(1:10) * 0.5;
fadingWeights(11:20) = fadingWeights(11:20) * 1;

global secondKernel
switch kernelTypeSecond
    case 1
        secondKernel = 'radial basis function kernel';
    case 2
        secondKernel = 'polynomial kernel';
    otherwise
        secondKernel = 'no kernel function specified';
end
```

A.2.3 makeInput01.m

```
load ./trainingSet.mat
load ./testSet.mat

disp('building training/test/target sets for
predictions of 1/2/3/4 time steps ahead');

input01Train = [];
coInput01Train = [];
caInput01Train = [];
pred1FirstTrain = [];
pred2FirstTrain = [];
pred3FirstTrain = [];
pred4FirstTrain = [];

actual02Train = [];
coActual02Train = [];
pred1SecondTrain = [];
pred2SecondTrain = [];
pred3SecondTrain = [];
pred4SecondTrain = [];
coPred1SecondTrain = [];
coPred2SecondTrain = [];
coPred3SecondTrain = [];
coPred4SecondTrain = [];

for i=1:lengthTrain
    tmp = caTrainingSet{i}';
    input01Train = [input01Train tmp(:,1:end-predTime)];
    caInput01Train{i} = tmp(:,1:end-predTime);
    pred1FirstTrain =  [pred1FirstTrain tmp(:, 2 : end-(predTime-1))];
```

```
    pred2FirstTrain =  [pred2FirstTrain tmp(:, 3 : end-(predTime-2))];
    pred3FirstTrain =  [pred3FirstTrain tmp(:, 4 : end-(predTime-3))];
    pred4FirstTrain =  [pred4FirstTrain tmp(:, 5 : end-(predTime-4))];

    clear inputLength; inputLength = size(tmp,2) - predTime;
    clear targetLength; targetLength = inputLength-(numTsSecond-1);

    actual02Train = [actual02Train tmp(:, numTsSecond:
                                       numTsSecond + targetLength-1 )];
    pred1SecondTrain = [pred1SecondTrain tmp(:,1 + numTsSecond :
                        1 + numTsSecond+targetLength-1)];
    pred2SecondTrain = [pred2SecondTrain tmp(:,2 + numTsSecond :
                        2 + numTsSecond+targetLength-1)];
    pred3SecondTrain = [pred3SecondTrain tmp(:,3 + numTsSecond :
                        3 + numTsSecond+targetLength-1)];
    pred4SecondTrain = [pred4SecondTrain tmp(:,4 + numTsSecond :
                        4 + numTsSecond+targetLength-1)];

    tmp01 = caCoordTrain{i};
    coInput01Train{i} = tmp01(1:end-predTime,1:3);
    coActual02Train{i} = tmp01(numTsSecond:numTsSecond + targetLength-1,1:3);

    coPred1SecondTrain{i} = tmp01(1 + numTsSecond:1 +
                                    numTsSecond+targetLength-1,1:3);
    coPred2SecondTrain{i} = tmp01(2 + numTsSecond:2 +
                                    numTsSecond+targetLength-1,1:3);
    coPred3SecondTrain{i} = tmp01(3 + numTsSecond:3 +
                                    numTsSecond+targetLength-1,1:3);
    coPred4SecondTrain{i} = tmp01(4 + numTsSecond:4 +
                                    numTsSecond+targetLength-1,1:3);

    clear tmp tmp01
end

input01Test = [];
coInput01Test = [];
caInput01Test = [];
pred1FirstTest = [];
pred2FirstTest = [];
pred3FirstTest = [];
pred4FirstTest = [];

actual02Test = [];
coActual02Test = [];
pred1SecondTest = [];
pred2SecondTest = [];
pred3SecondTest = [];
```

```
pred4SecondTest = [];
coPred1SecondTest = [];
coPred2SecondTest = [];
coPred3SecondTest = [];
coPred4SecondTest = [];

for i=1:lengthTest
    tmp = caTestSet{i}';
    input01Test = [input01Test tmp(:,1:end-predTime)];
    caInput01Test{i} = tmp(:,1:end-predTime);
    pred1FirstTest =  [pred1FirstTest tmp(:, 2 : end-(predTime-1))];
    pred2FirstTest =  [pred2FirstTest tmp(:, 3 : end-(predTime-2))];
    pred3FirstTest =  [pred3FirstTest tmp(:, 4 : end-(predTime-3))];
    pred4FirstTest =  [pred4FirstTest tmp(:, 5 : end-(predTime-4))];

    clear inputLength; inputLength = size(tmp,2) - predTime;
    clear targetLength; targetLength = inputLength-(numTsSecond-1);

    actual02Test = [actual02Test tmp(:, numTsSecond :
                                        numTsSecond + targetLength-1)];
    pred1SecondTest = [pred1SecondTest tmp(:,1 + numTsSecond :
                                            1 + numTsSecond+targetLength-1)];
    pred2SecondTest = [pred2SecondTest tmp(:,2 + numTsSecond :
                                            2 + numTsSecond+targetLength-1)];
    pred3SecondTest = [pred3SecondTest tmp(:,3 + numTsSecond :
                                            3 + numTsSecond+targetLength-1)];
    pred4SecondTest = [pred4SecondTest tmp(:,4 + numTsSecond :
                                            4 + numTsSecond+targetLength-1)];

    tmp01 = caCoordTest{i};
    coInput01Test{i} = tmp01(1:end-predTime,1:3);
    coActual02Test{i} = tmp01(numTsSecond:numTsSecond + targetLength-1,1:3);

    coPred1SecondTest{i} = tmp01(1 + numTsSecond:
                                 1 + numTsSecond+targetLength-1,1:3);
    coPred2SecondTest{i} = tmp01(2 + numTsSecond:
                                 2 + numTsSecond+targetLength-1,1:3);
    coPred3SecondTest{i} = tmp01(3 + numTsSecond:
                                 3 + numTsSecond+targetLength-1,1:3);
    coPred4SecondTest{i} = tmp01(4 + numTsSecond:
                                 4 + numTsSecond+targetLength-1,1:3);

    clear tmp tmp01
end
```

A.2.4 performKpca.m

```
%clear all
%initializing parameter

%initializing variables
X_train = input01Train;

disp('Start calculating the kernel matrix');
%carry out Kernel PCA

KmTrain = myKernel(X_train, X_train, kernelTypeFirst,
                   rbfVar01, polyInh01, sigVar01);

sizeKmTrain = size(KmTrain,2);
UmTrain = ones(sizeKmTrain, sizeKmTrain)/sizeKmTrain;

% centering in feature space!
KmTrainN = KmTrain - UmTrain*KmTrain - KmTrain*UmTrain
           + UmTrain*KmTrain*UmTrain;

disp('Start diagonalizing the kernel matrix');
[evecsFirstLayer, evalsFirstLayer] = eig(KmTrainN);
evalsFirstLayer = real(diag(evalsFirstLayer));
[dummy,esort] = sort(evalsFirstLayer);
esort=flipdim(esort,1);
evalsFirstLayer = evalsFirstLayer(esort);
evecsFirstLayer = evecsFirstLayer(:,esort);

for i=1:sizeKmTrain,
  evecsFirstLayer(:,i) = evecsFirstLayer(:,i)/(sqrt(evalsFirstLayer(i)));
end
```

A.2.5 featureExtractionKpca.m

```
disp('Start extracting features');

 numberOfTrainSamples = size(X_train,2);
 KmProjN = KmTrainN;

 features01Train = zeros(numEvFirst,numberOfTrainSamples);
 features01Train = evecsFirstLayer(:,1:numEvFirst)' * KmProjN; %'

 X_train = input01Train;
 X_test =  input01Test;
 numberOfTestSamples = size(X_test,2);
 UmTrain=ones(numberOfTrainSamples,numberOfTrainSamples)/numberOfTrainSamples;
```

```
 UmTest=ones(numberOfTrainSamples,numberOfTestSamples)/numberOfTrainSamples;

 KmProj = myKernel(X_train, X_test, kernelTypeFirst,
                 rbfVar01, polyInh01, sigVar01);

 clear KmProjN
 KmProjN = KmProj - KmTrain * UmTest - UmTrain * KmProj
           + UmTrain * KmTrain * UmTest;

 features01Test = evecsFirstLayer(:,1:numEvFirst)' * KmProjN;
```

A.2.6 makeInput02.m

```
%building compound vectors for second layer

features = features01Train(1:evFirstUsed,:);
features = features./max(max(features));

input02Train = [];

for i = 1:length(coInput01Train)
    tmp_input = [];

    for j = 1:(size(coActual02Train{i},1))
      tmp = features(:,j:(j+numTsSecond-1));
      tmp = reshape(tmp,size(tmp,1)*size(tmp,2),1);
      tmp_input = [tmp_input tmp];
    end

    features = features(:,1+size(coInput01Train{i},1):end);
    input02Train = [input02Train tmp_input];
    clear tmp
end
clear feat_vec tmp_input features

input02Test = [];
features = features01Test(1:evFirstUsed,:);
features = features./max(max(features));

for i = 1:length(coInput01Test)
    tmp_input = [];
    for j = 1:(size(coActual02Test{i},1))
      tmp = features(:,j:(j+numTsSecond-1));
      tmp = reshape(tmp,size(tmp,1)*size(tmp,2),1);
      tmp_input = [tmp_input tmp];
    end
```

```
    features = features(:,1+size(coInput01Test{i},1):end);
    input02Test = [input02Test tmp_input];
    clear tmp
end
```

A.2.7 performKpca02Fading.m

```
kernelType = kernelTypeSecond;
if(kernelType == 1)
  fprintf('Second Layer: using rbf kernel with parameter: %2.1f\n',rbfVar02)
elseif(kernelType == 2)
  fprintf('Second Layer: using poly kernel with parameter: %2.2f\n',polyInh02)
end

%initializing variables
X_train = input02Train;

disp('Start calculating the kernel matrix');
KmTrain = myKernelFading(X_train, X_train, kernelType,
                         rbfVar02, polyInh02, fadingWeights);

sizeKmTrain = size(KmTrain,2);
UmTrain = ones(sizeKmTrain, sizeKmTrain)/sizeKmTrain;

% centering in feature space!
KmTrainN = KmTrain - UmTrain*KmTrain - KmTrain*UmTrain
           + UmTrain*KmTrain*UmTrain;

disp('Start diagonalizing the kernel matrix');
%diagonalizing first kernel matrix and plotting the biggest eigenvalues
[evecsSecondLayer, evalsSecondLayer] = eig(KmTrainN);
evalsSecondLayer = real(diag(evalsSecondLayer));
[dummy,esort] = sort(evalsSecondLayer);
esort = flipdim(esort,1);
evalsSecondLayer = evalsSecondLayer(esort);
evecsSecondLayer = evecsSecondLayer(:,esort);

for i=1:sizeKmTrain,
  evecsSecondLayer(:,i) = evecsSecondLayer(:,i)/(sqrt(evalsSecondLayer(i)));
end
```

A.2.8 featexExtractionKpca02Fading

```
disp('Start extracting features');

numberOfTrainSamples = size(X_train,2);
```

```
KmProjSecondN = KmTrainN;

features02Train = zeros(numEvSecond,numberOfTrainSamples);
features02Train = evecsSecondLayer(:,1:numEvSecond)' * KmProjSecondN; %'

X_test =  input02Test;
numberOfTestSamples = size(X_test,2);

UmTrain = ones(numberOfTrainSamples,numberOfTrainSamples)/numberOfTrainSamples;
UmTest = ones(numberOfTrainSamples,numberOfTestSamples)/numberOfTrainSamples;

KmProj = myKernelFading(X_train, X_test, kernelType,
                        rbfVar02, polyInh02, fadingWeights);

clear KmProjSecondN
KmProjSecondN = KmProj - KmTrain*UmTest - UmTrain*KmProj
                + UmTrain*KmTrain*UmTest;

features02Test = evecsSecondLayer(:,1:numEvSecond)' * KmProjSecondN; %'
```

A.2.9 regressionOneTimeStep

```
prediction = [];
coefficients02Train = [];
target = [];

disp('Performing regression - One Time Step');
trainingData = pred1SecondTrain;
testingData = pred1SecondTest;

ccsTrain2nd = [];
features = features02Train;
target = trainingData;

for j = (numEvSecond/10):-1:2
    numberOfUsedFeatures = j*10;
    clear coefficients

    for i = 1:48
        coefficients(:,i) = regress(target(i,:)',
                            [ones(size(features(1:numberOfUsedFeatures,:),2),1)
                             features(1:numberOfUsedFeatures,:)'] );
        prediction(i,:) =  ([ones(size(features(1:numberOfUsedFeatures,:),2),1)
                            features(1:numberOfUsedFeatures,:)']
                            * coefficients(:,i))';
    end
```

```
    coefficients02Train{j} = coefficients;
    [r,p] = corrcoef(target, prediction);
    cc = r(2);
    ccsTrain2nd = [ccsTrain2nd; [numberOfUsedFeatures cc]];
end

features = features02Test;
ccsTest2nd = []; prediction = [];

target = testingData;

for j = (numEvSecond/10):-1:2
    numberOfUsedFeatures = j*10;

    for i = 1:48
        prediction(i,:) =  ([ones(size(features(1:numberOfUsedFeatures,:),2),1)
                             features(1:numberOfUsedFeatures,:)']
                             * coefficients02Train{j}(:,i))';
    end

    [r,p] = corrcoef(target, prediction);
    cc = r(2);
    ccsTest2nd = [ccsTest2nd; [numberOfUsedFeatures cc]];
    if (numberOfUsedFeatures == numEvSecond)
        predictionOneAhead = prediction;
    end
end

CcsTrain{1} = ccsTrain2nd;
CcsTest{1} = ccsTest2nd;
```

A.2.10 myKernel.m

```
function K = myKernel(x, y, kernelTypeFirst, rbfVar01, polyInh01, sigVar01)

if (kernelTypeFirst == 1)
for i=1:size(x,2)
   for j=1:size(y,2)
     K(i,j) = exp(-norm(x(:,i)-y(:,j))^2/rbfVar01);
   end
 end
elseif(kernelTypeFirst == 2)
     K = (((x' * y)/(size(x,1))) + polyInh01).^3;
else
     K = tanh(sigVar01*((x'*y)));
end
```

A.2.11 myKernelFading.m

```
function K = myKernelFading(x, y, kernelType, rbfVar01, polyInh01, fadingWeights)

if (kernelType == 1)
for i=1:size(x,2)
   for j=1:size(y,2)
     K(i,j) = exp(-norm(((x(:,i)-y(:,j)).* fadingWeights))^2/rbfVar01);
   end
 end
elseif(kernelType == 2)
    K = (((x.*repmat(fadingWeights,1, size(x,2)))' *
          (y.*repmat(fadingWeights,1, size(y,2))))/(size(x,1)) + polyInh01).^5;
end
```

A.3 Analyzing Data

A.3.1 analyzeData.m

```
plotEvFeatures
config
plotRasterSession(input01Train')
plotRasterSession2(Pred1SecondTest', predictionOneAhead')
```

A.3.2 plotEvFeatures.m

```
%plots the decay of eigenvalues from first and second layer
%in addition, the extracted features from both layers are depicted

numberOfEvals = 30;
figure;
subplot(2,3,1);
plot(evalsFirstLayer(1:numberOfEvals));
title('EW - first layer');

subplot(2,3,2)
imagesc(features01Train(1:numberOfEvals,:)')
title('1st - training features');

subplot(2,3,3)
imagesc(features01Test(1:numberOfEvals,:)')
title('1st - test features');

subplot(2,3,4);
plot(evalsSecondLayer(1:numberOfEvals));
title('EW - second layer');

subplot(2,3,5)
```

```
imagesc(features02Train(1:numberOfEvals,:)')
title('2nd - training features');

subplot(2,3,6)
imagesc(features02Test(1:numberOfEvals,:)')
title('2nd - test features');
```

A.3.3 plotRasterSession.m

```
function plotRasterSession(m_session)
%plots two sequences of raster records one after another
%Configuration parameters concerning number of slices in x and y
%dimension as well as x and y resolution are taken from config.m

if ~exist('N_Y', 'var')
  config
end

%checking range
m_session = m_session .* (m_session>=0);
m_session(m_session>1) = 1;

for i = 1:size(m_session, 1)
     figure(1);
     %figure;
  plotRaster ( m_session(i,:), N_Y, N_X, RES_X, RES_Y );
end
```

A.3.4 plotRasterSession2.m

```
function plotRasterSession2(m_session1, m_session2)
if ~exist('N_Y', 'var')
  config
end

%checking range
m_session1 = m_session1 .* (m_session1>=0); m_session1(m_session1>1) = 1;
m_session2 = m_session2 .* (m_session2>=0); m_session2(m_session2>1) = 1;

for i = 1:size(m_session1, 1)
  figure(1); plotRaster ( m_session1(i,:), N_Y, N_X, RES_X, RES_Y );
  figure(2); plotRaster ( m_session2(i,:), N_Y, N_X, RES_X, RES_Y );
end
```

Bibliography

[1] Hyvärinen A. and Pajunen P. Nonlinear independent component analysis: Existence and uniqueness results. *Neural Networks*, 12(3):429–439, 1999.

[2] Legenstein R. A., Markram H., and Maass W. Input prediction and autonomous movement analysis in recurrent circuits of spiking neurons. *Reviews in the Neurosciences (Special Issue on Neuroinformatics of Neural and Artificial Computation)*, 14(1–2):5–19, 2003.

[3] Smola A. and Schölkopf B. Sparse greedy matrix approximation for machine learning. *International Conference on Machine Learning*, 2000.

[4] Thomson A.M., West D.C., Wang Y., and Bannister A.P. Synaptic connections and small circuits involving excitatory and inhibitory neurons in layers 2-5 of adult rat and cat neocortex: Triple intracellular recordings and biocytin labelling in vitro. *Cerebral Cortex*, 12(9):936–953, 2002.

[5] Schölkopf B and Smola A. *Learning with Kernels*. Cambridge, MA: MIT Press, 2002.

[6] Schölkopf B., Smola A., and Müller K.R. Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation*, 10(5):1299–1319, 1998.

[7] Nessler Bernhard. A brain-like architekture for real-time computing. Master's thesis, Technical University Graz, 2004.

[8] Cortes C. and Vapnik V. Support vector networks. *Machine Learning, Vol20, pp 1-25*, 1995.

[9] Oja E. A simplified neuron model as a principal component analyzer. *Journal of Mathematical Biology*, 15:267–273, 1982.

[10] Hager G.D. and Toyama K. The XVision system: a general purpose substrate for portable real-time vision applications. *Computer Vision and Image Understanding*, 69:23–37, 1998.

[11] A. Gupta, Y. Wang, and H. Markram. Organizing principles for a diversity of gabaergic interneurons and synapses in the neocortex. *Science*, 287:273–278, 2000.

[12] Markram H., Wang Y., and Tsodyks M. Differential signaling via the same axon of neocortical pyramidal neurons. *Proceedings of the National Academy of Science*, 95(9):5323–5328, 1998.

[13] Burgsteiner Harald, Kröll Mark, Leopold Alexander, and Steinbauer Gerald. Movement prediction from real-world images using a liquid state machine. In *18th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems.*, pages 121–130, Bari, Italy, 2005. Springer.

[14] Mercer J. Functions of positive and negative type, and their connection with the theory of integral equations. *Royal Society of London Proceedings Series A*, 83:69–70, Nov 1909.

[15] Shawe-Taylor J. and Cristianini N. *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004.

[16] Thorsten Joachims, Nello Cristianini, and John Shawe-Taylor. Composite kernels for hypertext categorisation. In Carla Brodley and Andrea Danyluk, editors, *Proceedings of ICML-01, 18th International Conference on Machine Learning*, pages 250–257, Williams College, US, 2001. Morgan Kaufmann Publishers, San Francisco, US.

[17] Shlens Jonathan. Tutorial on principal component analysis, 2005.

[18] Pearson K. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 2:559–572, 1901.

[19] Schölkopf B. Kim K.I., Franz M.O. Kernel hebbian algorithm for iterative kernel principal component analysis. Technical Report Tech. Report No.109, Max-Planck-Institut fuer biologische Kybernetik, 2003.

[20] Ouimet Marie and Bengio Yoshua. Greedy spectral embedding. *International Conference on Machine Learning*, 2005.

[21] Bengio S. Popovici V. and Thiran J.P. Kernel matching pursuit for large datasets. *Pattern Recognition*, 38(12):2385–2390, 2005.

[22] Legenstein R. and Maass W. What makes a dynamical system computationally powerful? In S. Haykin, J. C. Principe, T.J. Sejnowski, and J.G. McWhirter, editors, *New Directions in Statistical Signal Processing: From Systems to Brain*. MIT Press, 2005. to appear.

[23] Stefan Rüping and Katharina Morik. Support vector machines and learning about time.

[24] S.Boyd and L.O.Chua. Fading memory and the problem of approximating nonlinear operators with volterra series. *IEEE Trans. on Circuits and Systems*, pages 1150–1161, 1985.

[25] Natschläger T., Markram H., and Maass W. Computational models for generic cortical microcircuits. *Computational Neuroscience: A Comprehensive Approach*, pages 575–605, 2004.

[26] Sanger T.D. Optimal unsupervised learning in a single-layer feedforward neural network. *Neural Networks*, 12:459–473, 1989.

[27] Franc V. and Hlavac V. Greedy algorithm for a training set reduction in the kernel methods. *Computer Analysis of Images and Patterns: Proceedings of the 10th International Conference*, pages 426–433, 2003.

[28] Maass W. and Markram H. On the computational power of recurrent circuits of spiking neurons. *Journal of Computer and System Sciences*, 69(4):593–616, 2004.

[29] Maass W., Legenstein R.A., and Markram H. A new approach towards vision suggested by biologically realistic neural microcircuit models. In H. H. Buelthoff, S. W. Lee, T. A. Poggio, and C. Wallraven, editors, *Biologically Motivated Computer Vision. Proc. of the Second International Workshop, BMCV 2002*, volume 2525 of *Lecture Notes in Computer Science*, pages 282–293. Springer (Berlin), 2002.

[30] Maass W., Natschläger T., and Markram H. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14(11):2531–2560, 2002.