Hierarchical Slow Feature Analysis on Visual Stimuli and Top-Down Reconstruction

DISSERTATION

zur Erlangung des akademischen Grades

doctor rerum naturalium
(Dr. rer. nat.)
im Fach Biologie

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät I
Humboldt-Universität zu Berlin

von
Dipl.-Phys. Niko Wilbert MSc

eingereicht am: 5.5.2011
Tag der mündlichen Prüfung: 23.4.2012

To my parents.

Abstract

This thesis examines a model of the visual system, which is based on the principle of unsupervised slowness learning and using Slow Feature Analysis (SFA). We apply this model to the task of invariant object recognition and several related problems. The model not only learns to extract the underlying discrete variables of the stimuli (e.g., identity of the shown object) but also to extract continuous variables (e.g., position and rotational angles). It is shown to be capable of dealing with complex transformations like in-depth rotation. The performance of the model is first measured with the help of supervised post-processing methods. We then show that biologically motivated methods like reinforcement learning are also capable of processing the high-level output from the model. This enables reinforcement learning to deal with high-dimensional visual stimuli. In the second part of this thesis we try to extend the model with top-down processes, centered around the task of reconstructing visual stimuli. We utilize the method of vector quantization and combine it with gradient descent. The key components of our simulation software have been intregrated into an open-source software library, the Modular toolkit for Data Processing (MDP). These components are presented in the last part of the thesis.

Zusammenfassung

In dieser Dissertation wird ein Modell des visuellen Systems untersucht, basierend auf dem Prinzip des unüberwachten Langsamkeitslernens und des SFA-Algorithmus (Slow Feature Analysis). Dieses Modell wird hier für die invariante Objekterkennung und verwandte Probleme eingesetzt. Das Modell kann dabei sowohl die zu Grunde liegenden diskreten Variablen der Stimuli extrahieren (z.B. die Identität des gezeigten Objektes) als auch kontinuierliche Variablen (z.B. Position und Rotationswinkel). Dabei ist es in der Lage, mit komplizierten Transformationen umzugehen, wie beispielsweise Tiefenrotation. Die Leistungsfähigkeit des Modells wird zunächst mit Hilfe von überwachten Methoden zur Datenanalyse untersucht. Anschließend wird gezeigt, dass auch die biologisch fundierte Methode des Verstärkenden Lernens (reinforcement learning) die Ausgabedaten unseres Modells erfolgreich verwenden kann. Dies erlaubt die Anwendung des Verstärkenden Lernens auf hochdimensionale visuelle Stimuli. Im zweiten Teil der Arbeit wird versucht, das hierarchische Modell mit Top-down Prozessen zu erweitern, speziell für die Rekonstruktion von visuellen Stimuli. Dabei setzen wir die Methode der Vektorquantisierung ein und verbinden diese mit einem Verfahren zum Gradientenabstieg. Die wesentlichen Komponenten der für unsere Simulationen entwickelten Software wurden in eine quelloffene Programmbibliothek integriert, in das "Modular toolkit for Data Processing" (MDP). Diese Programmkomponenten werden im letzten Teil der Dissertation vorgestellt.

Contents

1 Introduction

Understanding how the human brain works is one of the great challenges in modern science. Current efforts to find answers are covering processes from the subcellular level up to phenomena that involve multiple brain areas. A wide range of experimental methods has consequently been developed, which are, for example, optimized for specific spatial and temporal scales. Not surprisingly there is a corresponding need for a diverse arsenal of theoretical tools. Approaches that target a large scale often require a high level of abstraction from the biological substrate of the brain. Eventually it should be possible to transition between different scales, in order to explain how low-level structures implement high-level objectives.

The model that is explored in this thesis addresses how the brain can learn to process large amounts of information, which it receives from its sensory organs. We focus on the visual system, though some aspects might be transferable to other modalities as well. Due to the size and complexity of the visual cortex the model is formulated at an abstract level and does not attempt to provide a description in terms of individual neurons. Instead it defines high-level functional units and their underlying principles. Such a functional description is not only useful for the scientific understanding of the brain, there is also the possibility of using the same principles for technical applications.

For models that have to deal with large amounts of data (like visual stimuli) it is often inevitable to use numerical simulations in order to evaluate the model. This also means that we are limited by the capabilities of contemporary computers. However, one should not forget that computations in the brain are subject to constraints as well. For example, our primate brain consumes a significant fraction of our overall energy budget. The resulting evolutionary pressure probably led to its remarkable efficiency. One can therefore assume that the brain is cutting corners whenever possible. Efficiency is a major concern for credible models of brain functions.

Most scientists (including us) use heavily simplified data sets to test their computational models. This naturally leads to the question of scalability. For example, it would be problematic if the computational requirements for a visual system grow exponentially with the number of objects that it can recognize. However, one also has to take into account that the brain is fundamentally different from current computers. The connectivity and parallelism in neuron populations are very costly to simulate on standard hardware. A realistic evaluation might ultimately require that the model is formulated in terms of the brain's building blocks (e.g., neurons and synapses). Since this can be very difficult for large-scale models we only aim for the lesser goal of *biological plausibility.*

The model discussed in this thesis seems to be a good compromise in the light of all these different questions and requirements. In its basic form it is elegant and efficient to simulate, thanks to the underlying algorithm of Slow Feature Analysis (SFA). This thesis tries to provide more data points for its evaluation, by analysing both some strengths (e.g., performance for invariant object recognition) and pointing out possible weaknesses (especially with regard to top-down processing).

Overview of the Thesis

This thesis is split into three parts. Part I introduces the SFA algorithm (Chapter 2) and based on it a hierarchical model of the visual system. This model is then applied to the problem of invariant object recognition (Chapter 3 and Chapter 4). In Chapter 5 the hierarchical SFA model is combined with reinforcement learning to demonstrate that it can provide an adequate basis for models of other brain areas (in analogy to the role of the visual system). Overall Part I demonstrates the strength of hierarchical SFA for the biologically relevant task of invariant object recognition.

Part II is centered around extending the feed-forward model in order to enable top-down processes, which are assumed to play an important role in the visual system. In Chapter 6 we start by defining the top-down task of input reconstruction, which is our test case throughout this part. Chapter 7 describes how the required information for top-down processes can be captured in our model with the method of vector quantization. Some results that emerge from that are presented as well. The actual top-down reconstruction is then explored in Chapter 8, together with the method of gradient descent.

Part III focuses on some technical aspects of the simulations from the previous parts. As mentioned earlier, computer simulations are an important part of computational neuroscience. The ever-increasing complexity of these simulations is a problem with regard to the scientific principle of reproducibility. We have therefore integrated several key parts of our model into an open source software library, the Modular toolkit for Data Processing (MDP). This enables other groups to reproduce, modify and reuse our model with minimal effort. A basic introduction to the MDP library is provided in Chapter 9. In Chapter 10 we discuss additions to the library that were created for Part I of this thesis, while Chapter 11 introduces the additions for Part II.

In chapters that are the result of a collaboration we point out the individual contributions at the beginning. This is especially relevant throughout Part I of this thesis and in Part III, Chapter 9.

Part I

Hierarchical Slow Feature Analysis

2 Slow Feature Analysis

2.1 Slowness Principle

Animals and humans are faced with sensory signals that often change very rapidly. For example a simple head movement can completely alter the light pattern that falls on the retina. Receptors that were previously in a bright sport end up in dark patches and vice versa. When an animal or object passes through the visual field this has a similar effect, again causing rapid intensity changes for the rods and cones in the retina. On the other hand the underlying structure of the scene typically changes on a much slower time scale (e.g., which objects are present and where).

A simple object that passes through our visual field can be described by its *pose* (consisting of the position and angle relative to the observer) and *identity*. In principle only this underlying *configuration* is behaviourally relevant, representing the important *features* of the scene. Such a reduced and abstract description is generally called a *high-level representation*. The corresponding visual stimulus is called a *view*, and it is often approximated as a pixel-based image (in turn represented by an input vector $\mathbf{x}$). The visual system has to extract all the relevant high-level features from the stimulus, making them available to other parts of the brain in an appropriate representation.

The postulated difference in the timescale of the changes on the retina and the high-level representation leads to the idea that *slowness* can be used as a learning objective in the brain. If high-level features are expected to be slowly changing then it might be possible to extract them by looking for slowness in the raw visual data. The resulting learning principle does not require an external supervision signal, it is purely based on the statistics of the visual stimuli. This is critical for the plausibility of its use in the brain.

The hierarchical networks that are at the core of this thesis are based on the Slow Feature Analysis (SFA) algorithm [Wiskott, 1998; Wiskott and Sejnowski, 2002], which is one particular implementation of learning based on slowness. There have been various earlier approaches to slowness, e.g., [Hinton, 1989; Földiák, 1991; Mitchison, 1991; Becker and Hinton, 1992]. Slowness has previously been used in some hierarchical models as well [Wallis and Rolls, 1997; Einhäuser et al., 2005; Wyss et al., 2006].

Unsupervised learning based on the slowness principle (i.e., learning that exploits temporal continuity of real-world stimuli) has also attracted the attention of experimentalists [Miyashita, 1988; Li and DiCarlo, 2008]. It has been shown in monkey experiments that features in monkey inferior temporal cortex are adapted in a way that is consistent with the slowness principle [Li and DiCarlo, 2008, 2010].

2.2 SFA Algorithm

SFA as defined in [Wiskott, 1998] solves the following learning task: Given a multidimensional input signal we want to find instantaneous scalar input-output functions that generate output signals that vary as slowly as possible but still carry significant information. To ensure the latter we require the output signals to be uncorrelated and have unit variance. In mathematical terms, this can be stated as follows:

Optimization problem: *Given a function space $\mathcal{F}$ and an I-dimensional input signal* $\mathbf{x}(t)$ *find a set of J real-valued input-output functions* $g_j(\mathbf{x}) \in \mathcal{F}$ *such that the output signals* $y_j(t) := g_j(\mathbf{x}(t))$

$$\textit{minimize } \Delta(y_j) := \langle \dot{y}_j^2 \rangle_t \tag{2.1}$$

under the constraints

$$\begin{aligned} \langle y_j \rangle_t &= 0 \quad \textit{(zero mean)}, && (2.2)\\ \langle y_j^2 \rangle_t &= 1 \quad \textit{(unit variance)}, && (2.3)\\ \forall i < j : \langle y_i y_j \rangle_t &= 0 \quad \textit{(decorrelation and order)}, && (2.4) \end{aligned}$$

with $\langle \cdot \rangle_t$ *and* $\dot{y}$ *indicating temporal averaging and the derivative of* y*, respectively.*

Equation (2.1) introduces the Δ-value, which is a measure of the temporal slowness (or rather fastness) of the signal $y(t)$. It is given by the mean square of the signal's temporal derivative, so that small Δ-values indicate slowly varying signals. The constraints (2.2) and (2.3) avoid the trivial constant solution and constraint (2.4) ensures that different functions g_j code for different aspects of the input. Because of constraint (2.4) the g_j are also ordered according to their slowness, with g_1 having the smallest Δ.

It is important to note that although the objective is slowness, the functions g_j are instantaneous functions of the input, so that slowness cannot be achieved by low-pass filtering. Slow output signals can only be obtained if the input signal contains slowly varying features that can be extracted instantaneously by the functions g_j. Note also that for the same reason, once trained, the system works fast, not slowly.

In the computationally relevant case where $\mathcal{F}$ is finite-dimensional the solution to the optimization problem can be found by means of Slow Feature Analysis (SFA) [Wiskott, 1998; Wiskott and Sejnowski, 2002]. This algorithm, which is based on an eigenvector approach, is guaranteed to find the global optimum. Biologically more plausible learning rules for the optimization problem do exist as well [Hashimoto, 2003; Sprekeler et al., 2007].

If the function space $\mathcal{F}$ is unrestricted the optimization problem can be solved with variational calculus as described in [Wiskott, 2003; Franzius et al., 2007]. In this case the optimization problem for the high-dimensional visual input can be reformulated for the low-dimensional configuration input (e.g., position, orientation, and object identity). Therefore the variational calculus approach becomes tractable and allows making analytical predictions, which are discussed for our specific case in Section 3.3.4.

2.3 Linear SFA and Parallelization

For simplicity we assume that the data has zero mean (i.e., $\langle x_i \rangle_t = 0$). The solutions of the optimization problem for linear SFA have the form

$$g_j(\mathbf{x}) = \mathbf{w}_j^T \mathbf{x} \ , \tag{2.5}$$

where the $\mathbf{w}_j$ are weight vectors. For the nonlinear case a nonlinear expansion of the input signal is performed and afterwards the same linear SFA algorithm is applied (this is discussed in Section 3.2).

The data set that is used for the calculation of the weight vectors $\mathbf{w}_j$ is called the *training data*. Once the weights have been computed, the resulting functions g_j can be applied to new data as well, which in this context is called the *test data*. This distinction between training and testing is a general pattern for learning algorithms (the test data is often used to check for overfitting). It is also possible to formulate SFA as an online learning procedure by periodically recalculating the weight vectors, so that an output $\mathbf{y}$ is produced right from the start. Learning processes in the brain generally have to work online, but since we only want look at the fully trained system we can use the standard SFA algorithm.

We use the SFA implementation from the MDP software library (see Part III), which is based on the formulation of the algorithm in [Berkes and Wiskott, 2005]. A detailed derivation is provided there, and mathematically this is equivalent to the original algorithm in [Wiskott and Sejnowski, 2002]. In this formulation the weight vectors $\mathbf{w}_j$ are the eigenvectors of a generalized eigenvalue problem

$$\mathbf{A}\mathbf{w}_j = \lambda_j \mathbf{B}\mathbf{w}_j \tag{2.6}$$

and the $\lambda_j \in \mathbb{R}$ are their eigenvalues, which are equal to the corresponding Δ-values $\Delta(y_j)$. The symmetric matrices $\mathbf{A}$ and $\mathbf{B}$ are defined as

$$\mathbf{A} := \langle \mathbf{x}^T \mathbf{x} \rangle_t \quad \text{and} \quad \mathbf{B} := \langle \dot{\mathbf{x}}^T \dot{\mathbf{x}} \rangle_t \ . \tag{2.7}$$

The first matrix is the covariance matrix of the data, while the second one is the second moment matrix of the time derivative. In our computer simulations time t is discrete, so it can be replaced with an index $t = 1, \ldots, n$. Correspondingly the temporal averaging $\langle \cdot \rangle_t$ becomes a sum.

The covariance matrix $\mathbf{A}$ requires the product $\mathbf{x}^T \mathbf{x}$ for all t, so its computational complexity is proportional to the number of training samples n. Solving the generalized eigenvalue problem on the other hand is independent of the number of training samples. Beyond a certain n the overall computational cost is therefore dominated by the covariance matrices. The summands $\mathbf{x}^T \mathbf{x}$ for different t can be calculated independently, so the training data can be split up into multiple chunks. For each chunk the covariance matrix is calculated separately, combining them in the end to construct the overall covariance matrix (one also has to keep track of the mean and number of data points in each chunk for normalization). The same method can be used for $\mathbf{B}$. This means that the SFA algorithm is easy to parallelize, reducing the computing time for large sets of training data. This has first been exploited in [Franzius et al., 2007] and we use it throughout this thesis. Further details are discussed in Section 10.2.

3 Invariant Object Recognition with Hierarchical SFA

3.1 Introduction

Object recognition is one task a vision system has to master in order to enable successful interaction with the environment. Naturally this recognition should be independent of object position, relative angle, lighting conditions, and other factors. In other words it should be *invariant*. However, the position and relative rotational angle can be just as crucial (e.g., when faced with a predator), so they too must be extracted. In the previous chapter we called the combination of all these relevant features the high-level representation. The concept of *invariance* can be extended to these other features as well. For example it does not matter if a hunter throws a spear at a zebra or a gazelle. The aiming procedure is only based on the relative position, which is independent from the type of animal (or object identity).

Primates can perform invariant object recognition successfully in most situations of everyday life and behave accordingly, which requires that the information about object identity is available for higher brain areas in a sufficiently explicit format. Since it is hard to answer which specific format is most adequate for later stages of the brain, we just postulate that a simple classifier should be able to distinguish between the objects representations after few training examples and that a simple linear regression should be able to extract pose information. A special case of such a representation would, of course, be an explicit population code where one set of neurons codes for object identity and a different set codes for object pose.

A good vision system should also *generalize* to previously unseen configurations, i.e., it should learn the relevant statistics of transformations rather than just memorizing specific views. It should also generalize to new objects, e.g., it should successfully estimate the position and orientation[1] of an object that was never shown before.

In general, the process of extracting the configuration of an object from a view is very hard to solve, especially in the presence of a cluttered background and many different possible objects. In this chapter we use high-resolution views of complex objects but restrict the problem to the cases of only one object present at a time and a static homogeneous background. Due to the high variability of views caused by changing configuration there is an infinitely large space of possible views of even a single object. The manifold of all possible views of an object is embedded in in a highly complex way within the visual ("pixel") space and two views of distinct objects are often closer to

[1]Generally, there is no canonical "0°"-view of an object, thus a random phase offset of absolute phase for a new object is to be expected.

each other in pixel space than two views of the same object [DiCarlo and Cox, 2007]. One approach to solve such complex computational problems is to break them up into a series of simpler computations. Functionally, each stage or layer should represent views belonging to a single object more compactly and views of distinct objects more separately, thus "untangling the view manifolds". Furthermore, the dimensionality of the object representation should be sufficiently low on the highest layer, such that object-specific behavior (or in the simplest case just an object classifier) needs only few labeled training examples. Both properties, modularized computation and dimensionality reduction, can be implemented by a converging visual hierarchy.

The slowness principle has been applied for object recognition before [e.g., Wallis and Rolls, 1997; Becker, 1999; Stringer and Rolls, 2002; Einhäuser et al., 2005]. However, our model goes beyond these earlier ones by not only extracting translation-invariant and view-invariant representations of object identity but also simultaneously information about position and viewing angles. Furthermore, we show meaningful generalization of the model to previously unseen objects. The structure of the resulting representation solely depends on the statistics of presentation of the training views. We show that in a restricted scenario these representations are encoded independently of each other in an analytically predictable way. In a more complex scenario the solutions tend to mix but are still very simple to decode by linear regression.

To optimize the slowness based objective function we use the Slow Feature Analysis (SFA) algorithm as discussed in the previous chapter. Except for minor changes, the model used here is identical to that used earlier for the modeling of place cells, head direction cells, and spatial view cells in the hippocampal formation [Franzius et al., 2007]. The complete mathematical framework of this publication carries over to the problem of invariant object recognition as presented here (leading to the predictions in Section 3.3.4).

Our model is able to learn both view-invariant object-specific representations and position and rotational information from complex high-dimensional data in a biologically plausible model of the ventral visual system. We therefore speculate that slowness might serve as a general principle for sensory coding in the brain.

The work presented in this chapter has been done in close collaboration with Mathias Franzius and will be published in [Franzius et al., 2011], which is the basis for this chapter. All the simulations and the data analysis presented here were programmed and applied by myself. They are a continuation of our earlier work presented in [Franzius et al., 2008]. In this earlier version the hierarchical network was programmed and trained by Mathias Franzius, while the stimulus generation and the data analysis were done by myself. My rewrite of the model software for [Franzius et al., 2011] led to a significant improvement in performance, thanks to some optimizations in the model architecture and better control over the stimulus generation (eliminating issues like clipping of the objects). All the required building blocks for the hierarchical network are now available in the "Modular toolkit for Data Processing" (MDP) library, which is discussed in Part III of this thesis.

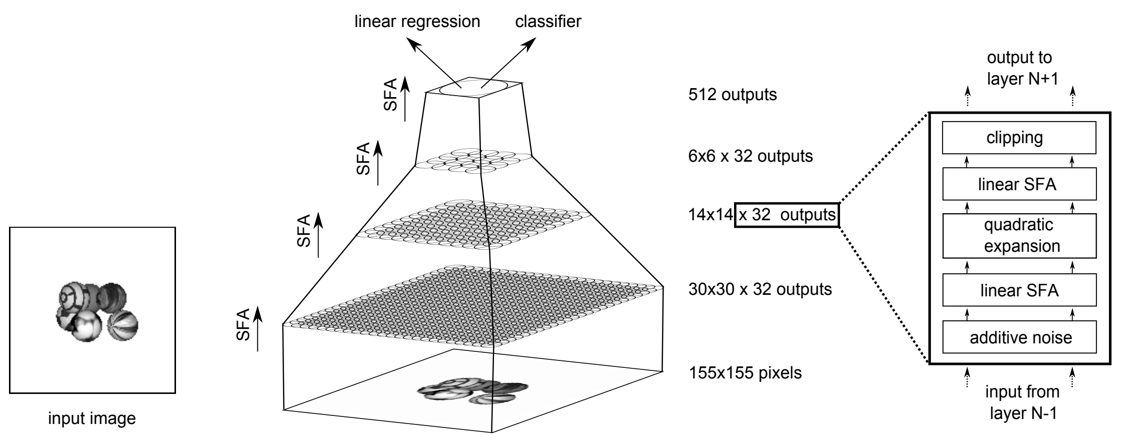


Figure 3.1: **Model architecture and stimuli.** An input image is fed into the hierarchical network. The circles in each layer symbolize the overlapping receptive fields, which converge towards the top layer. The same set of steps is applied on each layer, which is visualized on the right hand side.

3.2 Hierarchical Network Architecture

The visual system is, to a first approximation, structured in a hierarchical fashion, first extracting local features which are then integrated to more and more global and abstract features. We apply SFA in a similar hierarchical manner to the raw visual input data. First, the slow features of small local image patches are extracted. The integration of spatially local information exploits the local correlation structure of visual data. A second layer extracts slow features of these features (again integrating spatially local patches), and so on. Each SFA node in the hierarchical network basically performs the following sequence of operations: linear SFA for dimensionality reduction, quadratic expansion, and another linear SFA step for slow-feature extraction. SFA has been applied successfully to visual data in this hierarchical fashion previously [e.g., Wiskott and Sejnowski, 2002; Franzius et al., 2007].

A hierarchical organization turns out to be crucial for the applicability of the approach for computational reasons, since the application of non-linear SFA on the whole high-dimensional input would be infeasible. Efficient use of resources is also an issue in biological neural circuits. It has been suggested that connectivity is the main constraint there [Chklovskii and Koulakov, 2000; Legenstein and Maass, 2005]. Since a hierarchical organization requires nearly exclusively local communication, it avoids extensive connectivity.

3.2.1 Network Structure

The detailed network structure is shown in Figure 3.1. It consists of four layers of SFA nodes, connected topographically in a feed-forward manner. We first describe the

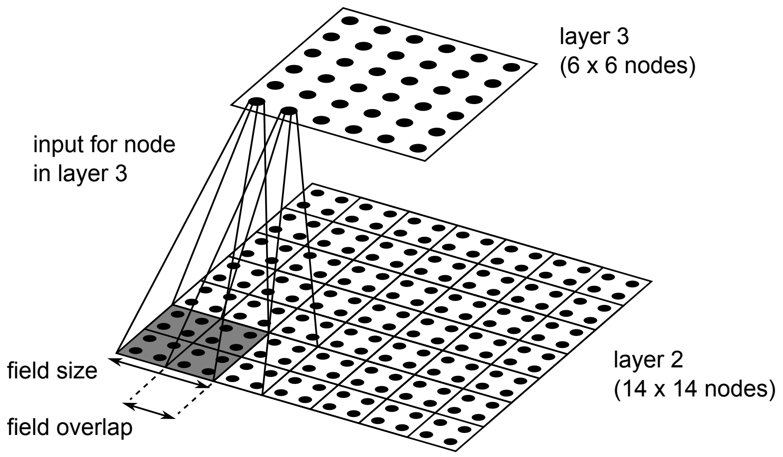


Figure 3.2: **Input fields of nodes in layer 3.** Each dot represents the 32 dimensional SFA output from one node. The field overlap is 2 nodes and the borders of the receptive fields are represented by the black lines between the dots.

internal organization of each individual SFA node before we give a detailed description of the connection architecture below. In each SFA node, first additive Gaussian white noise (with a variance of 10^{-6}) is introduced for numerical reasons, to avoid possible singularities in the subsequent SFA step. Then a linear SFA is performed for a first reduction of the input dimensionality. In a subsequent quadratic expansion, the incoming data $x_1, \ldots, x_n$ is mapped with a basis of the space of polynomials with degree up to two. So in addition to the original data, all quadratic combinations like $(x_1)^2$ or $x_1 x_2$ are concatenated to the data block. Another linear SFA stage is then applied to the expanded data. The solutions of linear SFA on this expanded data is equivalent to those of SFA in the space of polynomials up to degree two. After the second SFA stage we apply a clipping at ± 4. This clipping removes extreme values that can occur on test data due to the divergence of the quadratic functions for larger values. However, both the additive noise and the clipping are mostly just technical safeguards and have typically no effect on the network performance.

The number of SFA components used from the first linear SFA stage in each node depends on the layer in which the SFA node is situated. The first linear SFA stage in each node reduces the dimensionality to 32 in the first two layers, 42 in the third layer, and 52 in the fourth layer (the increase in dimensionality across layers leads to a small performance increase). Accordingly, the quadratic expansion then increases dimensionality to 560, 560, 945 and 1430, in the first, second, third, and fourth layer respectively. The second linear SFA stage reduces the dimensionality of the expanded signal to 32, except for the top layer, where the output is reduced to 512 dimensions.

We now describe how the nodes are connected (see Figure 3.2). We use a layered feed-forward architecture, i.e., the nodes in the first layer receive inputs only from the input image, and nodes in higher layers receive inputs exclusively from their previous

Layer	Number of nodes	Input area of node	Overlap per direction	SFA outputs per node
0 (Image)	155×155	-	-	(1 pixel)
1	30×30	10×10	5	32
2	14×14	4×4	2	32
3	6×6	4×4	2	32
4	1	6×6	-	512

Table 3.1: **Overview over the network architecture.** Layer 0 denotes the input image, a node corresponds to a pixel in that image. The input area denotes the number of nodes in the previous layer from which a node receives input. An example for layer 3 is visualized in Figure 3.2.

layer. Additionally, connections are topographically structured such that a node receives inputs from neighboring nodes in the previous layer. In the following, the part of the input image that influences a node's output is denoted as its receptive field. On the lowest layer, the receptive field of each node consists of an image patch of 10 by 10 grayscale pixels. The receptive fields jointly cover the input image of 155 by 155 pixels. The nodes form a regular (i.e., non-foveated) 30 by 30 grid with partially overlapping receptive fields, resulting in an overlap of five pixels in each direction. The second layer contains 14 by 14 nodes, each receiving input from 4 by 4 layer 1 nodes with neighboring receptive fields, resembling a retinotopic layout (the overlap is two nodes in each direction). The third layer contains 6 by 6 nodes, each receiving input from 4 by 4 layer 2 nodes with neighboring receptive fields, again in a retinotopic layout (with 2 nodes overlap in each direction, as shown in Figure 3.2). All 6 by 6 layer 3 outputs converge onto a single node in layer 4, whose output we call SFA-output. This organization is summarized in Table 3.1.

Thus, the hierarchical organization of the model captures two important aspects of cortical visual processing: increasing receptive field sizes and accumulating computational power at higher layers. The latter is due to the quadratic expansion in each layer, so that each layer computes a subset of higher polynomials than its predecessor. The SFA-outputs at the top layer compute subsets of polynomials of degree $2^4 = 16$.

3.2.2 Network Training

The network layers are trained sequentially from bottom to top. For computational efficiency, we train only one node with stimuli from all node locations in its layer and replicate this node throughout the layer. For example this means that the node in the lowest layer sees $30 \times 30 = 900$ times as much data as if it was only trained at a single location. This mechanism effectively increases the number of training samples and implements a weight-sharing constraint. However, the system performance does not depend on this mechanism. The statistics of the training data are approximately identical for all receptive fields, so individually learned nodes would lead to the same results (but at higher computational cost). While the weight-sharing does ease the emergence of translation invariance it is not at all sufficient.

We used 50,000 time points for the training of the two lower layers and 200,000 for the two top layers. These training sequences were generated with a random walk procedure, which is described in the next section. The random walk parameters of the training data are identical for all layers. The larger training set for the top layers is motivated by the smaller multiplicative effect of the weight-sharing and by the slower time scales towards the top (though one has to combine this factor with the complexity of the data structure). The simulated views are generated from their configuration (position, angles, and object identity) with floating point precision and are not artificially discretized. Therefore it is highly unlikely that any two images in the training set are identical.

3.3 Stimuli

The model was trained and tested with image sequences containing views of different objects. OpenGL was used to render the views of the objects as textured 3D-models in front of a white homogeneous background. We initially used colored stimuli and found color to be a very strong cue especially for object classification. In order to prevent the model from using this simple cue, we only used grayscale views for the results presented here. Two different object classes were used that are described below. For each object class the model was trained with a number of different objects. In the testing phase we also added new objects that the model had never seen during training. The stimulus set in the form of sourcecode or data is available upon request. Note that, in contrast to many other models, the model was trained with image sequences consisting of tens of thousands of distinct views (see Section 3.2.2).

We introduced the two dissimilar fish and sphere stimulus sets to show that our results do not critically depend on one particular kind of stimulus. Using new stimulus sets instead of established ones like the COIL100 database [Nayar et al., 1996] makes it difficult to compare the results with those of other models. Unfortunately, no freely available databases could give us a comparable amount of control over the object pose, which is critical for the evaluation of our model. Our implementation allows us controlled simultaneous rotation around multiple axis and, specifically, allows control over the temporal presentation order.

3.3.1 Stimulus Classes

In the first experiment the objects were clusters of textured spheres as shown in Figure 3.3A, which provide a difficult but generic task. The same six textured spheres were used in different spatial arrangements for all the objects. For each object the spheres were randomly fixed on a lattice (hexagonal close packing) with three layers of size two by two. Five such objects were used for training and five additional were added for testing. As the examples in Figure 3.3A illustrate, identifying the rotation angles and identities for such objects is quite difficult even for human observers.

Using the same building blocks for all objects should force the model into using high-level configural features for object classification since the objects have the same low-level

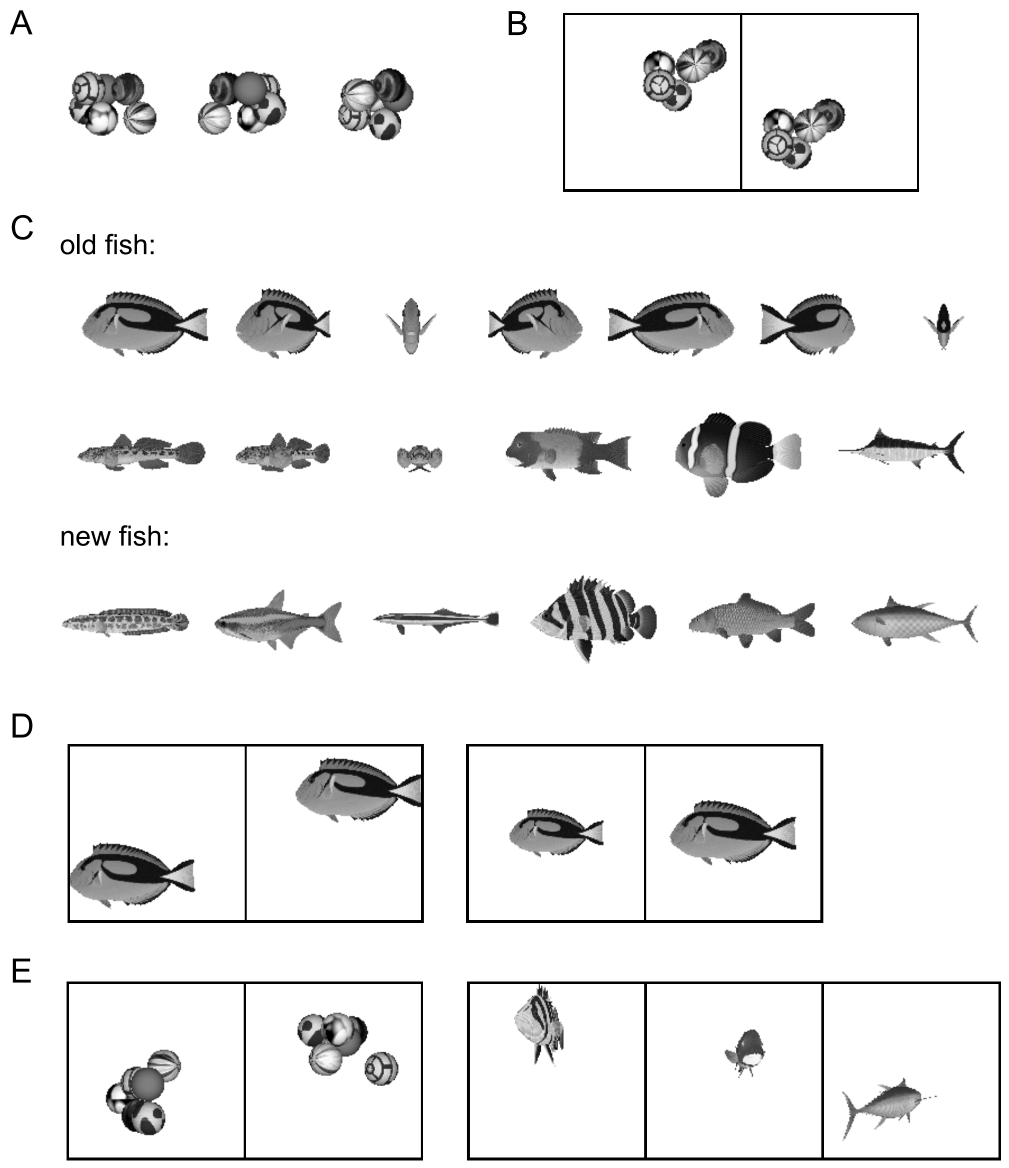


Figure 3.3: **Objects used for stimulus generation.** **A** shows the sphere objects (each cluster of 6 spheres is one object). The first two views show the same object under different in-depth rotation angles, while the third view shows a different object. **B** shows the top right and bottom left corner positions for a single object, to illustrate the translation range. In **C** some of the fish objects used for training and testing are shown with examples for the effect of in-depth rotation. The first row illustrates a rotation from 0° to 270°. **D** shows the corner positions for a single fish and the smallest and largest scale. In **E** we show some of the images which were actually used for training and testing.

features. On the other hand, the common low-level features of the objects help the model to generalize to untrained objects. Using spheres has the advantage that the outline does not give a simple clue for the in-plane rotation angle.

In the second experiment, models of different fish (see Figure 3.3B) were used to provide more natural stimuli from a single object class (all models taken from [Toucan Corporation, 2005] with permission). We used 15 different fish for training and added 10 additional ones for testing, raising the number of different objects to 25.

3.3.2 Object Configurations

For sphere objects, the horizontal x-coordinate, the vertical y-coordinate, the in-depth rotation angle ϕ_y, and the in-plane rotation angle ϕ_z were chosen as configuration variables. x and y range from 0 to 1, ϕ_y and ϕ_z from $0°$ to $360°$. Another configuration variable was the object identity ranging from one to ten, with objects one to five being the training objects. So the transformations the model had to learn consisted of translations in the plane, rotations along the y and z axes (with the in-depth rotation coming first) and changes of object identity. The field of view used in the rendering of the sphere objects was $10°$, so the views look almost like an orthographic projection.

For the fish objects, the configuration variables were x, y, ϕ_y, object identity, and the size. So compared to the sphere objects we added changes in size and removed the in-plane rotations. The maximum size of the objects was 33% larger than the minimum size in each direction (see Figure 3.3D). A greater range of scale variations would have required larger images as well, since even for the smallest scale and difficult angles all configuration variables should be extractable. Introducing scale transformations is also an additional safeguard against the model relying purely on the size of a fish for classification. The field of view was set to $45°$, leading to significant perspective projection distortions. This results in an additional position dependency of the fish object views, which is clearly visible in Figure 3.3E. A size or image area normalization of the different fish objects is not possible due to the configuration dependency of the effective object area in the image (which varies most for the in-depth rotation).

3.3.3 Random Walk Procedure

The configurations for the training sequences were generated by a random walk procedure. To generate a configuration in the sequence we add a random term to the current spatial, angular, and scaling velocities of the object. The random term is drawn from an interval with a homogeneous probability density and zero mean. The velocities are cut off at certain limits and by adjusting these limits one can effectively determine the transformation timescales. The position, angles, and scale are then updated according to these velocities. If an object reaches the position boundary, it is bounced back. All the parameter values are given in Table 3.2. The whole procedure produces flat configuration histograms (given enough time points) and the velocity profiles are independent of the configuration values.

Let us illustrate this procedure with an example for the x-position of a sphere object.

	Spheres			Fish		
	max. vel.	max. acc.	Δ	max. vel.	max. acc.	Δ
x (in $[0;1]$)	0.025	0.005	0.008	0.03	0.005	0.008
y (in $[0;1]$)	0.025	0.005	0.008	0.03	0.005	0.008
size (in $[0.75;1]$)				0.005	0.001	0.014
ϕ_z in-plane)	0.04	0.01	0.007			
ϕ_y rad (in-depth)	0.04	0.01	0.007	0.04	0.01	0.007

Table 3.2: **Parameters for the random walk procedure.** All velocity values are given "per frame" and the angle values are in radiant measure. The "max. vel." parameter is the maximal velocity, "max. acc." is the maximal change in velocity per frame (i.e. acceleration). The Δ's are rounded empirical values from the actual random walk sequences of the training data.

The allowed interval for these position values is $[0, 1]$. Assume that the current position is 0.99 and that the current velocity (change in position per frame) is 0.022, which is still below the maximum value of 0.025. The velocity update is now randomly picked from $[-0.005, 0.005]$, e.g., 0.004. This brings the velocity value to 0.026, so it is cut off at 0.025. The new position would therefore be 1.015, but since this is beyond the boundary, the object bounces back and the final new position is 0.985.

In each step the object identity was changed with low probability ($p = 0.002$). A blank frame was inserted if a switch took place to avoid linking together different objects in identical poses in the stimulus, which would introduce an element of supervised training. The effect of the blank frame on the coding of the configuration variables is described in Section 3.3.4.

Note that the time structure of the stimuli resurfaces in the SFA output of our model. The SFA algorithm does not change the timescales of the features, these timescales are purely determined by the parameters of the random walk.

3.3.4 Theoretical Predictions

In Section 2.2 we mentioned that it is possible to predict the optimal SFA solutions if the available function space is unrestricted. For our model this condition is obviously not met, but if the hierarchical model is computationally powerful enough then it can come close to these optimal predictions. Therefore we briefly describe the theoretical predictions for our stimulus set, which follow directly from the detailed work in [Wiskott, 2003] and [Franzius et al., 2007].

The optimal solution for the slowness optimization problem is generally a function of the slowest configuration variable. For example, let us assume that $c_1(t)$ (e.g., the position x or y from Section 3.3) is the configuration variable that varies on the slowest timescale (i.e., has the smallest Δ-value after normalization to unit variance). The configuration values are mapped to the input signal $\mathbf{x}(t)$, which corresponds to the raw pixel data in our model. Further assume that the values of c_1 are homogeneously distributed in the interval $[0; L]$ (with $L > 0$) and that the velocity profile does not depend on the value of c_1. As shown in [Franzius et al., 2007] the slowest possible output signal $y_1(t) := g_1(\mathbf{x}(t))$ is then given by

$$g_1(\mathbf{x}(t)) = \sqrt{2}\cos\left(\pi\frac{c_1(t)}{L}\right). \tag{3.1}$$

So g_1 is a half cosine over the c_1-interval, i.e., a monotonically increasing representation of c_1-position, which is invariant to all other configuration variables. If the timescale of c_2 is only slightly faster than that of c_1, and both are uncorrelated, then the second slowest solution $g_2(\mathbf{x}(t))$ is a similar half cosine of c_2. However, the decorrelation condition (2.4) is also satisfied by solutions of the form

$$g(\mathbf{x}(t)) = \sqrt{2}\cos\left(\pi\frac{nc_1(t)}{L}\right) \quad \text{with } 1 < n \in \mathbb{N}. \tag{3.2}$$

These higher order harmonics are all uncorrelated, with their timescales being multiples of the original timescale. Similar harmonics exist for c_2. Another class of solutions consists of products of the solutions in c_1 and c_2 mentioned above. If the configuration variables are uncorrelated, then the products of their solutions are uncorrelated as well. All these solutions are naturally ordered by their timescales (i.e., their Δ-value).

If two variables vary on equal timescales, then any normalized linear combination of their solutions has the same Δ-value as well. Thus any two uncorrelated solutions from the two-dimensional subspace spanned by the solutions of the individual variables form a valid set of solutions. We refer to this ambiguity as the problem of linear mixing and discuss its practical implications in Section 3.4. Depending on their timescales, higher order harmonics or products of solutions can also take part in linear mixtures.

There are two other special cases that are relevant for our model: cyclic configuration variables like ϕ_y or ϕ_z and variables coding for object identity. Cyclic variables arise if the input data $\mathbf{x}(t)$ consists of periodic functions of a hidden variable $\phi(t) \in [0; 2\pi]$, i.e.,

$$\mathbf{x}(t) = \mathbf{x}(\phi, \ldots) = \mathbf{x}(\phi + 2\pi, \ldots) \tag{3.3}$$

Again we assume a uniform distribution of ϕ and an independent velocity profile. The optimal uncorrelated solutions based on the periodic $\mathbf{x}(t)$ are then

$$g_1(\mathbf{x}(t)) = \sqrt{2}\sin(\phi(t) + \phi_0) \quad \text{and} \quad g_2(\mathbf{x}(t)) = \sqrt{2}\cos(\phi(t) + \phi_0) \tag{3.4}$$

with an arbitrary phase offset ϕ_0. As g_1 and g_2 vary on the same timescale, ϕ_0 simply reflects the linear mixing ambiguity.

The second special case is a variable that takes only discrete[2] values, like the object identity $k(t) \in \{1, 2, \ldots, N\}$. If the transitions between the values are all equal (e.g., switching from object 1 to object 3 is just as likely as switching to object 2), then there

[2]When using a continuous time t this implies non-differentiable changes of the variable, leading to infinities in the analytical analysis. For practical applications of the SFA-algorithm this does not pose a problem, since the time t has to be discrete in any case.

are $N - 1$ uncorrelated optimal solutions. Those solutions are different step functions of k [Berkes, 2005]. Their exact shapes are determined by the decorrelation conditions.

As mentioned earlier we insert a blank frame whenever the object identity changes. Since SFA is instantaneous and deterministic, the SFA outputs always take the same value for this blank frame. Therefore there is no penalty if the SFA outputs become object dependent. It allows the SFA solutions for variables like position and angles to become object dependent without increasing their Δ-value (i.e., reducing their slowness). Therefore the dimension of the space of Δ-optimal solutions increases for all variables apart from the identity. A basis of slowest solutions is now given by functions that are of the form (3.1) or (3.4) for a single object and are zero for all others. So for one particular variable and N objects there are now N uncorrelated optimal solutions instead of just one.

3.4 Post-Processing

In Section 3.3.4 we already presented the form of the expected SFA-output and the problem of linear mixing of solutions for variables with identical timescales. The limited sampling of the training sequence and the function space used by the model generally leads to deviations from the theoretical predictions. Thereby a mixing in the SFA-output can occur even if the underlying timescales are different. The higher order harmonics are also affected by this. Overall this makes it practically impossible to prevent the mixing for complex applications with many different transformations, which complicates the interpretation of the SFA-output.

3.4.1 Feature Extraction with Linear Regression

Our main objective here is to show that pose information is indeed extracted from the raw image data even if linearly mixed. The easiest way to do this is by calculating a multivariate linear regression of the SFA-output against the known configuration values. This gives us a projection vector for each configuration variable (plus a constant offset). To calculate the variable value from a given output one takes the scalar product of the projection vector with the SFA-output vector and adds the offset value.

While the regression procedure is obviously supervised due to the use of the reference configuration values, it nevertheless shows that the relevant signals are easily accessible. Extracting this information from the raw image data linearly is not possible, as shown in Section 3.5.3. One should also note that the dimensionality of the SFA-output is smaller than the raw image data by two orders of magnitude.

For technical convenience, the configuration reference values were binned, which, for example, simplified the calculation of error bars. For each configuration variable (apart from object identity) 60 bins were used, which is sufficient to not influence the results (tests with smaller bin sizes did not show any significant differences). Since the predicted SFA solutions are cosine functions of the position values (see Section 3.3.4), one has to map the reference values correspondingly before calculating the regression. The results from the regression are then mapped with the inverse function. For example one does not calculate the regression for position x, but for the predicted solution $y(x) = \cos(\pi x)$ instead. The result from the linear regression is then mapped with $\arccos(y)/\pi$ (after cutting off the values outside the interval $[-1; 1]$) to get the estimated position values. This method is also used for the size transformations of the fish stimuli.

For those SFA solutions that code for rotational angles, the theory predicts both sine and cosine functions of the angle value (as described in Section 3.3.4). Therefore we calculated regressions for both mappings and then calculated the angles via the arctangent. This automatically matches the phase of the extracted angles to that of the reference values.

If multiple objects are trained and separated with a blank, the solutions for the position and rotational angles are in general object dependent. In theory a global regression for all objects should work nevertheless, since the different solutions can be linearly combined into an object independent solution (i.e., it lies in the space spanned by the solutions). However, for the complicated stimuli used here this generally only works for the "simpler" features like object position (see Section 3.5). For the rotational angles the limited function space of the model leads to complicated dependencies on object identity. The easiest solution to this is to perform an individual regression for every object. While this procedure sacrifices the object invariance it does not affect the invariance under all other transformations (e.g., the extracted angles are still invariant with respect to the position).

3.4.2 Classification

As described in Section 3.3.4, the object identity of N different objects is optimally encoded (under the SFA objective) by up to $N - 1$ uncorrelated step functions, which are invariant under all other transformations. In the SFA-output this should lead to separated clusters for the trained objects. For untrained objects, those SFA-outputs coding for object identity should take new values, which are again invariant under all other transformations. Thus it should be possible to separate old and new objects. Linear mixing of the identity solutions with other features is to be expected.

To explore the potential classification ability of the model we applied two very simple classifiers on the SFA-output: a k-nearest-neighbor and a Gaussian classifier. Since the linear mixing does not affect cluster separation it should not affect the classifiers. On the other hand the classification performance is affected by imperfect transformation invariance of the identity solutions (i.e., dependencies beyond the linear mixing). This can cause overlap between object clusters. Clusters also have to intersect if two distinct objects share an identical or highly similar view. Note that the dimensionality of the regression input also has to be matched with enough reference points to capture the shapes of the clusters.

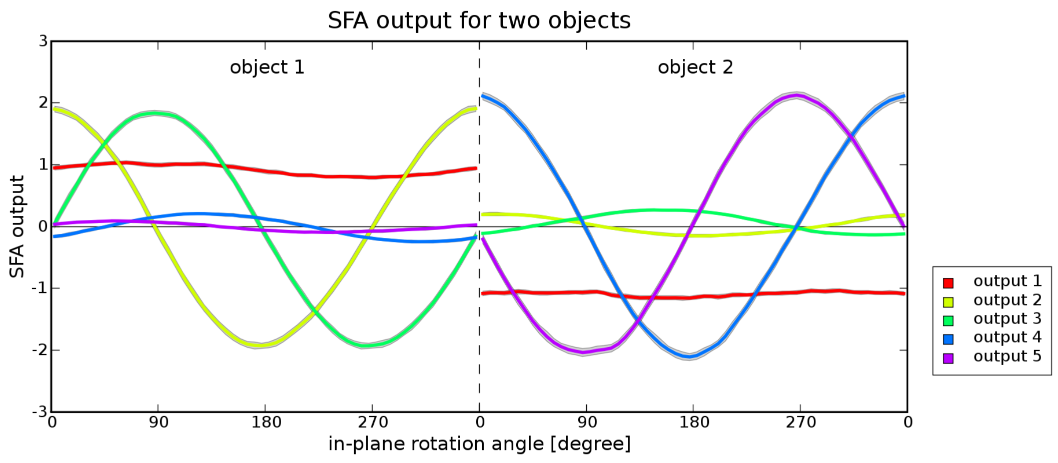


Figure 3.4: **Five slowest SFA-outputs for the simulation with a reduced transformation set.** The in-plane angle dependence of the five slowest SFA-outputs is shown for the two trained sphere objects. The centers of the lines indicate the mean values, the outer line borders represent ± one standard deviation or root mean square error (RMSE). Note that for each rotational angle the objects were shown at many different positions, so the small RMSE illustrates the position invariance of the outputs. The slowest output (output 1) codes for object identity. The other four outputs code for the sine and cosine of the angle. Both the amplitudes and the phases of the solutions are object dependent, as described in Section 3.3.4.

3.5 Results

First, we consider the raw SFA-output for an experiment with a reduced transformation set and clearly separated timescales. Then the experiments with a larger transformation set and similar timescales are analyzed with the methods described above.

3.5.1 Reduced Transformation Set

To illustrate the SFA-outputs of our model we first present a simplified example, with fewer transformations and well separated timescales. This should lead to SFA-outputs that code only for one specific configuration variable per output and are invariant under all other transformations. Two sphere objects with translation and in-plane rotation were used for this example. Both objects were already used for the training of the model. The two lowest layers were, as usually, trained with similar timescales for all transformations. In the training data of the two highest layers very fast translations were used. Identity was the slowest feature and in-depth rotation lay in-between. Note that this manipulation of the timescales was only done in this educational example. By varying the position very fast we are effectively discarding this feature, improving the clarity of the other features.

As predicted in Section 3.3.4, the slowest output channel codes for object identity (Figure 3.4). The output value is almost completely invariant under both rotation and translation. Outputs two to five are the model outputs coding for rotation angle. They nicely fit the theoretical prediction (i.e., the outputs are a basis for the predicted space of the slowest solutions). As expected we get $4 = 2 \times 2$ uncorrelated solutions for the angle, since we trained with two different objects.

In summary, for cases of clearly separated timescales and few simultaneous decorrelated transformations as presented in this section, the model outputs are clearly predictable by variational calculus methods and directly encode the configuration parameters in an invariant manner.

3.5.2 Full Transformation Set

The results in this section are based on the full transformation set (as described in Section 3.3) and the SFA-outputs are subjected to supervised postprocessing (see Section 3.4). The stimulus sets used for testing consisted of 100,000 views in total for the sphere objects (covering both old and new objects) and 100,000 views for the fish objects. Both were generated in the same way as the training data and were then shown to a hierarchical network that was trained with the same object type (as described in Section 3.2). So each stimulus type (fish or sphere objects) has its own hierarchical network.

We refer to objects that were shown during the training phase of the SFA layers in the hierarchical network as *old objects*, while the additional objects that were only shown in the testing phase of the hierarchical network are called *new objects*. So there are 15 old and 10 new fish objects and 5 old and 5 new sphere objects. For the postprocessing step the 100,000 data points were divided into training and test data. From now on the terms *training* and *test data* will therefore refer to the postprocessing, even though they are both test data from the point of view of the fully trained hierarchical SFA-network.

Position and Rotation Angles

To extract the x and y object coordinates from the model SFA-output, we used multivariate linear regression, as described in Section 3.4. Half of the old object data (the training data) was used to calculate the regressions, the other half for testing. The same regression was also tested with the new objects (only half of the data was used for this, to match the number of test samples for the old objects). As one can see in Table 3.3 and Figure 3.5, the regressions work reasonably well for the x and y coordinates, especially given the large size of the objects (which can reach 70% of the image, see Figure 3.3). For the sphere objects we get a root mean square error (RMSE) of 7% (relative to the position range) for old objects and around 9% for new objects. For the old and new fish objects the error in the x-direction increases to 9% and 12% due to the large object size in this direction.

To extract the in-plane and in-depth rotation angles for the spheres, an individual regression for each object was necessary. This still requires invariance of the representa-

	Spheres			Fish		
	old	new	all	old	new	all
x	7%	8%	7%	9%	12%	10%
y	7%	9%	8%	7%	7%	7%
ϕ_z (in-plane)	30°	94°	62°			
ϕ_y (in-depth)	37°	97°	67°	49°	68°	57°
size				13%	12%	12%
ϕ_z (in-plane)	9°	9°	9°			
ϕ_y (in-depth)	10°	13°	11°	15°	19°	17°
ϕ_y no front/back				9°	11°	10°
Gaussian classifier	99.9%	99.0%	98.4%	96.4%	97.8%	95.2%
KNN classifier	99.2%	99.7%	99.0%	97.2%	97.5%	95.2%
NC classifier	89.9%	70.2%	65.9%	83.4%	77.8%	74.0%

Table 3.3: **Results for sphere and fish objects.** The **first part** shows the root mean square error (RMSE) for the global regressions of the x- and y-coordinates given in percent relative to the coordinate range (chance level is 28.9%). For reference we also provide the results of the global regressions for the rotational angles (chance level is 104°). In the **second part** of the table we show the mean RMSE for the individual object regressions. The mean RMSE for the object size is given relative to the size value range (only fish). The next rows show the mean RMSE for the rotational angles (chance level is again 104°). The last row in the second part was calculated by omitting the ambiguous front and back views of the fish objects. In the **third part** of the table the classifier hit rates in percent are shown. The second row refers to the k-nearest-neighbour classifier ($k = 5$), the last row is the nearest center classifier. Chance level is 10% for all sphere objects and 4% for all fish objects.

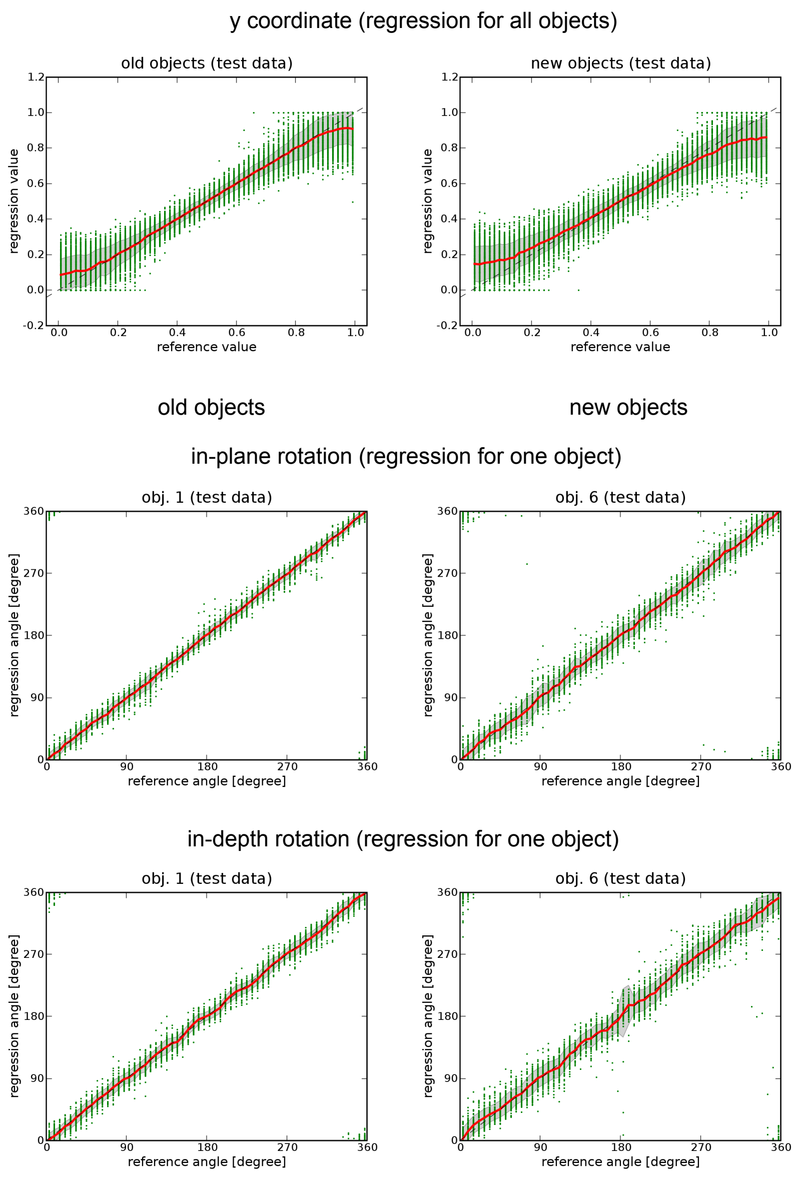


Figure 3.5: **Reconstruction of sphere object position and angle on test data.** The variable values that were calculated with linear regression are plotted against the correct reference values. The gray dots are data points, the black line is the mean and the gray area shows ± one RMSE. The regression of the y coordinate was based on all five training objects. For the rotational angles we show object specific regressions. Object 1 was used during training of the SFA layers, object 6 was not.

tions under the other transformations (including the other rotation type). As the results in Table 3.3 show, the in-plane angles were extracted with a mean RMSE of about 9°, the in-depth angles with about 11°. The RMSEs for individual objects vary, ranging from 9° to 17° for the in-depth rotation. We also verified that calculating the angles with global regressions for all objects did not work well (see Table 3.3).

For the fish, the mean RMSE for the in-depth angles is about 17° for all 25 objects. A large part of this increase compared to the sphere objects is due to systematic errors. The fish models naturally have a very similar front and back view and therefore the model has difficulties to differentiate between those two views, which can be clearly seen in Figure 3.6A. To verify this we did the same analysis after removing all data points within 30° of the front and back view. This brought the error down to the level of the sphere objects (see Table 3.3).

As for the angles, individual regressions for all objects were used to calculate the size for the fish objects (size transformations were not used for the sphere objects). This works with a RMSE of about 12% on average (see Figure 3.6B). The model cannot simply use the object area to infer the size, since the area covered by an object largely depends on the in-depth rotation angle and the object identity.

Classification

To quantify the classification ability of the model, two classifiers were used on the SFA-output. The classifiers were trained with about 5,000 data points for each sphere object and 2,000 data points for each fish object (i.e., half of the data, as for the regressions). The other half of the data points was then used as test data to test the classifier performance. The Gaussian classifier performed with about 95% hit rate for all 25 fish objects (see Table 3.3) and 98% for all 10 sphere objects. Training and testing with only the old or only the new sphere objects resulted in a performance of 99.9% (old) and 99.0% (new). The k-nearest-neighbour classifier ($k = 5$) worked at about the same performance level. In Figure 3.7 some 2D projections of the data clusters are shown. They indicate that the data clusters for different objects are reasonably well separated (as shown by the classifier results). The classification errors for the fish objects mostly occur around the difficult front or back views. Removing all data points within 30° of the front and back view raised the hit rate of the Gaussian classifier to 99.8% on all 25 fish objects.

We also tested the ability of the model to classify based on limited training data for the post-processing step (the hierarchical network was left unchanged). The fish objects were used for these tests and training data was restricted to views where the fish were in the upper left image area. The in-depth rotation angles were restricted to the interval between 0° and 90°. Testing of the classifiers was then done with the fish object in the opposite lower right area and with angles from 120° to 330° (i.e., a distance of at least 30° to the known object views). Since the number of training data points was below the output dimensionality of 512 we used Fisher discriminant analysis (based on the full data set for the old objects only) to project the data down to 14 dimensions. This number matches the theoretically predicted uncorrelated SFA solutions for object identity (as described in Section 3.3.4). The results are shown in Table 3.4.

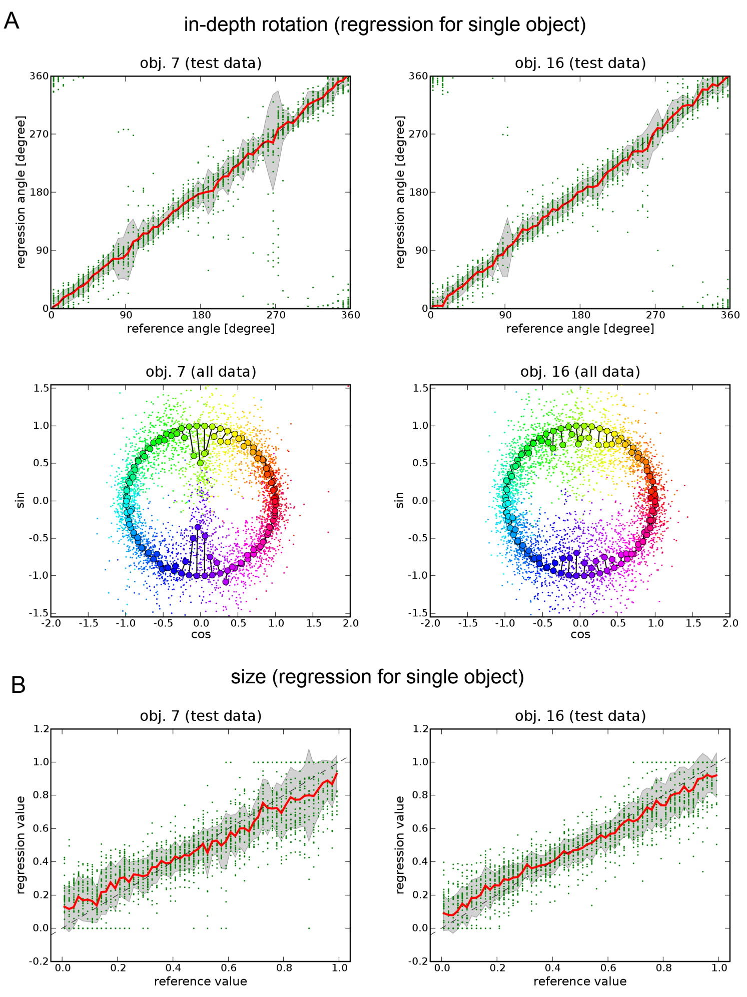


Figure 3.6: **Reconstruction of in-depth rotation angle and size for fish objects on test data.** In **A** we show the regression results for the in-depth rotation angle for one fish used during training of the SFA layers (obj. 7, last one second row Figure 3.3C) and one new object (obj. 16, first one third row Figure 3.3C). The first plots at the top show the regression angle versus the correct reference angle. The plots below show the regression values for the sine and cosine of the in-depth rotation angle. The shades of the points indicate the correct reference angle value. The mean regression values for the 60 different reference angle bins are shown as larger dots connected with a line to ideal values on the circle, which are also indicated by a large dot. The deviations at 90° and 270° for object 7 are a result of the similar front and back view of this fish model. Object 7 was selected to illustrate this effect, which is less of a problem for many of the other objects (including object 16 shown on the right). **B** shows the regression for the size of a single object.

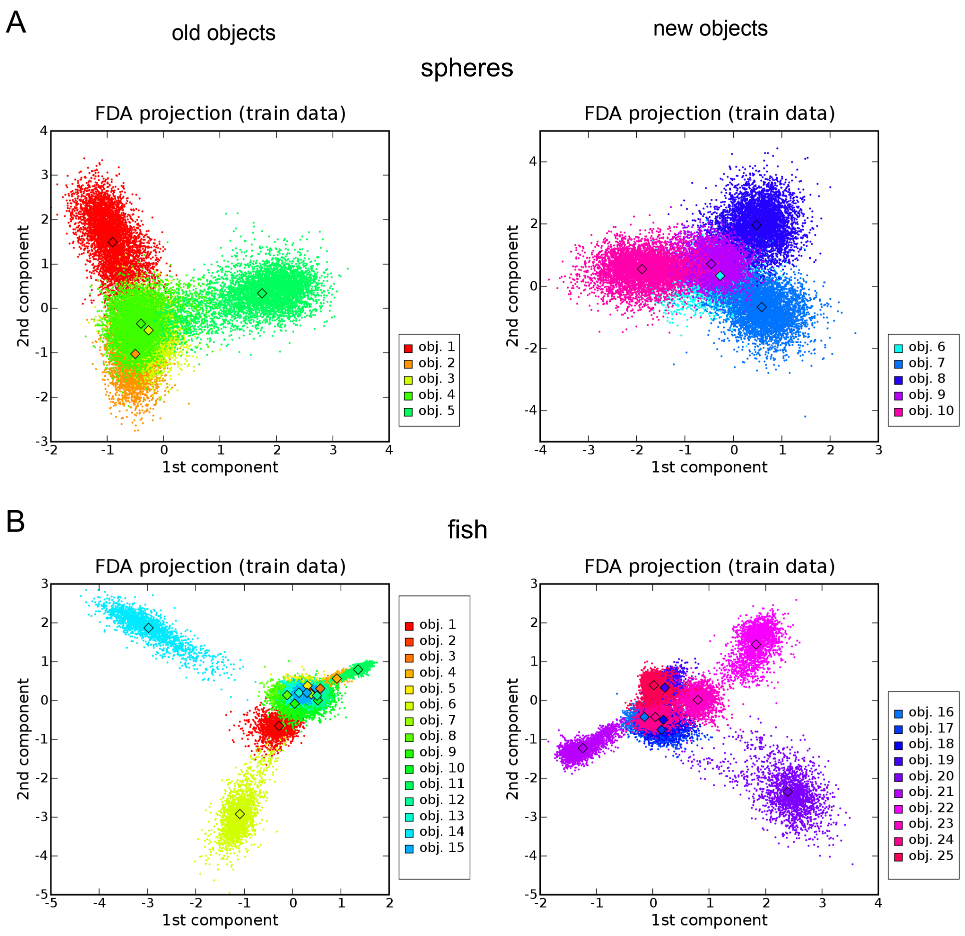


Figure 3.7: **2D projections of the data clusters.** Two-dimensional projections of the data points are shown, which are colored according to object identity. The projection plane was chosen for each plot to maximize cluster separation (using Fisher discriminant analysis). **A** shows the data for the five old sphere objects on the left and the five new ones on the right. **B** shows the projected data for the fifteen old and ten new fish objects.

# of training	old objects		new objects		all objects	
samples per obj.	G	KNN	G	KNN	G	KNN
1		79.9%		57.9%		60.5%
≈ 34	82.2%	90.6%	53.9%	75.6%	58.5%	76.6%
≈ 174	84.5%	90.7%	64.8%	82.3%	63.5%	79.0%

Table 3.4: **Classifier hit rates for reduced training data of fish objects.** Training and test data were taken from different regions of the configuration space to test the generalization ability of the model, see text for details. Columns labeled with "KNN" refer to the k-nearest-neighbor classifier ($k = 1$ in the first line, otherwise $k = 3$), while those with "G" refer to the Gaussian classifier.

For the first test only a single randomly picked view per object was used to train the classifier (so the k of the k-nearest-neighbor classifier had to be set to one as well). This resulted in a hit rate on the 10 old objects of 79.9%. For the new objects this decreased to 57.9% and to 60.5% for all 25 fish objects (which is still well above the 4% chance level). Increasing the number of samples increases the performance, up to 79.0% on all 25 objects for the k-nearest-neighbour classifier ($k = 3$). Note that the training and test data were still drawn from the non-overlapping parameter intervals described above.

An analogous test for the extrapolation of the regression results is not easily possible. Restricting the regression reference values to a limited interval (e.g., 0° to 90° for the angle) leads to large systematic errors in the regression, since features like the sine and cosine of an angle are no longer uncorrelated. Therefore one cannot use the standard procedure for linear regression. Using a more complicated algorithm one could overcome this restriction, but would also make the post-processing part of our model more complex.

3.5.3 Controls

To verify the robustness of our model and to justify the model structure we performed several control experiments.

Normal Transformation Set

The following control experiments were done with the full transformation set for the sphere or fish objects:

Choice of time scales. Our model strongly depends on the timescale of the transformations in the training images. We chose the random walk parameters such that the Δ-values of the transformations were in the same range (e.g., for the spheres the smallest Δ is about 0.0045, the largest 0.0081). Moving away from such balanced timescales generally leads to a decrease of the performance for the more quickly varying configuration variables. On the other hand one can increase the performance for a particular variable by making it slower relative to the other vari-

outputs	32	64	128	256	512
ϕ_y	40°	32°	25°	20°	17°
classifier hit rate	80.7%	87.6%	91.2%	94.0%	95.2%

Table 3.5: **Influence of the number of SFA outputs on the performance (for fish objects).** The table presents results for different channel numbers. The first row shows the mean RMSE for the in-depth rotation angle of all 25 objects. The second row shows the hit rate of the Gaussian classifier on all objects.

ables. These dependencies were to be expected, as a result of the limited function space available in the model.

To verify this we modified the fish random walk and trained a network with the new data. In one example we increased the maximal velocity of the x-position from 0.06 to 0.12 and later 0.18, which increased the Δ-value to 0.017 and then 0.33. The RMSE for x on all 25 fish rose from 10% to 12% and then 22%. At the same time the performance for all other configuration variables improved. For example the Gaussian classifier performance on all 25 fish increased from 95.2% to 98.6% and 99.0%. If the goal was to maximize only the classifier performance one would therefore make all other variables very fast. In this special case SFA approximates nonlinear Fisher discriminant analysis [Berkes, 2005]. Note, however, that maximum transformation speed is limited by receptive field size: if speed is too high, no spatiotemporal structure can be extracted anymore on lower layers with smaller receptive field sizes.

Blank frame. In the training sequences we separated views of different objects with a blank frame. However, we found that the performance of the model is largely unaffected by omitting the blank.

Output dimensionality. The results presented so far are based on 512 SFA-output dimensions i.e., the 512 slowest outputs of the top layer, which may seem excessive. We found that a high number of outputs increased the quality of the results (see Table 3.5). The computational cost of working with 512 outputs is still relatively low (except for the k-nearest neighbor classifier). If the number of transformations is low, one can reduce the output number without sacrificing performance (the results in Figure 3.4 are an extreme case of this).

Nonlinearity. To verify that the nonlinear expansion is really necessary we trained and tested a purely linear network on the same fish object data as before. The hierarchical structure was still used, since otherwise the covariance-matrix sizes would have been prohibitive. So in each layer we omitted the quadratic expansion and the second SFA step. While the position of the objects was still extracted with an RMSE of 16%, in the x-direction (as compared to 10% in the nonlinear network), the mean RMSE for the in-depth rotation angle increased to 60° (as compared to 17°). The hit rate of the Gaussian classifier dropped to 39% (as compared to

95.2%) for all 25 fish objects. These results demonstrate the importance of the nonlinear expansion in the hierarchical network.

We also verified that the performance drops significantly if the linear terms are removed from the quadratic expansion. Adding third degree terms in the expansion (e.g., $x_i x_j x_k$) on the other hand did not significantly improve network performance (due to the large number of terms only a random selection was added).

Nonlinearity without SFA. It was verified that a nonlinear expansion alone is not sufficient to reach the performance level of our full model. This was done by modifying the second SFA step after the quadratic expansion in each layer. Instead of picking the slowest directions, random directions were used. Note that a dimensionality reduction like this is necessary to keep the computational requirements of the next layer in the network under control. The first linear SFA step in each layer was left unchanged. Again the model was trained and tested with the same fish object data as before. This seriously degraded performance, with the mean RMSE for the in-depth rotation angle rising to 77° (as compared to 17%). The hit rate of the Gaussian classifier dropped to 10% (as compared to 95.2%).

Baseline performance on pixel data. To verify that our stimulus set is non-trivial we trained a supervised nearest cluster center classifier on the raw image data. It achieved a hit rate of only 14% on all 25 fish and 13% on the 10 sphere cluster objects. The dimensionality of the data ($155^2 = 24025$ pixel) makes it difficult to test more sophisticated classifiers, and the computational cost would quickly exceed that of our model.

PCA instead of SFA. We compared the performance of SFA with Principal Component Analysis (PCA). This was done by replacing all SFA steps in our network with PCA, leaving everything else unchanged (including the dimensionalities). The PCA networks were then trained and tested exactly like the SFA ones. While the results were better than chance level they were also vastly inferior to the SFA performance. The Gaussian classifier achieved only about 41% hit rate on the 10 spheres and 56% on all 25 fish. For the in-depth angle of the fish the mean RMSE was about 77° (for the spheres the results were even worse). Under the condition that the input data has an appropriate time structure this demonstrates that SFA can clearly outperform PCA.

Modified Transformation Set

In the following control experiments we explore the impact of a reduced or otherwise modified transformation set during training. This helps to understand the ability of the network to actually learn the transformations that were used in the training stimuli. On the other hand this is also important for understanding the contribution of the post-processing steps to the overall performance.

Generalization to new stimulus class. As a test of the generalization capabilities of our model we took a network that had been trained with sphere cluster objects and tested it with fish objects (the original random walk parameters from Table 3.2 were used). Surprisingly the performance of this network was practically identical to that of the dedicated fish network. On the other hand the performance of a fish network when applied to sphere cluster objects was clearly inferior to that of the dedicated sphere cluster network. In this case the performance of the Gaussian classifier dropped to 91% (as compared to 98%) and the mean RMSE for the angles grew by a factor of two. Note that the sphere cluster objects incorporated the additional in-plane rotation, which was not present in the fish training data.

We then used the random walk parameters of the fish for the sphere cluster objects and tested the performance of the fish network on this new stimulus set. The performance was now excellent (mean RMSE for the in-depth angle was 7° and 99.97% hit rate of the Gaussian classifier on all 10 objects). Together with the previous results this suggests that the additional in-plane rotation caused the problems for the fish network, since it was not trained with two simultaneous rotations.

Untrained transformation. As another variant of the previous control experiments we trained networks with reduced transformation sets and then tested their performance on stimuli with a larger transformation set: we trained a network with translations and different identities, but without any rotations (this experiment was performed for both fish and sphere cluster objects). When the resulting network was tested with a stimulus set that included in-depth rotation the results were surprisingly good. In general the mean RMSE for the in-depth angle was a few degrees larger than in the normal case and the classifier performance was just below 90%. However, when the in-plane rotation was added as well (for the sphere cluster objects) the performance took a major hit (the mean RMSE for the angles rose above 40° for both in-plane and in-depth rotation and the Gaussian classifier performance dropped to 77% on all 10 sphere cluster objects). We also did a complementary experiment by training a network with in-depth rotation only and then testing it with a stimulus set that also included translation. In this case the performance was very poor (e.g., only 50% hit rate of the Gaussian classifier).

Random network. One interpretation of the previous results would be that the extraction of the in-depth angle is primarily made possible by the supervised linear regression in combination with the 512 dimensions. To verify this we tested a network in which the two SFA nodes in each layer are replaced with nodes that pick random orthogonal directions (which are fixed throughout the experiment) and added a whitening stage in each layer. On a stimulus set containing only in-depth rotation this produced reasonable results (mean RMSE for the angle was 23°, Gaussian classifier hit rate was 93.3% on all 10 sphere cluster objects). However, as soon as translations were introduced as well the performance went down to chance level. In both cases the whitening stage was trained with the same transformations that were used during testing.

Reduced output dimensionality. To check whether the post-processing results for untrained transformations are purely an artifact of the large number of dimensions we also checked the performance for a reduced number of outputs (by taking only the slowest 128 network outputs, as in one of the earlier control experiments). It turned out that the decrease in performance is comparable to that of a normal network. So the performance gap between a network trained on a reduced transformation set and a fully trained network does not depend on the output dimensionality.

In-depth versus in-plane rotation. The sphere cluster stimuli were also used to determine whether there is a qualitative difference in network learning between stimuli with in-depth and in-plane rotation. One network was trained without in-plane rotation and a second one was trained without in-depth rotation (all other transformations were left unchanged and conformed to Table 3.2). Then both networks were tested with the standard sphere cluster testing stimuli, which include both in-depth and in-plane rotation. As expected the performance in both cases was worse than that of a fully trained network. However, the performance of the network that was trained with in-plane rotation was slightly better than that with in-depth rotation (e.g., 96.7% versus 93.4% Gaussian classifier hit rate and 16.6° versus 20.8° mean RMSE for the in-depth angle). The fact that the network was not able to create an efficient representation of the in-depth rotational angle supports the view that in-depth rotation is harder to learn than in-plane rotation.

Overall, the results suggest that for a single continuous transformation type (e.g., in-depth rotation) the post-processing alone is able to produce reasonable results. But as soon as multiple transformations are present this is no longer true and the SFA network is critical for the performance. A network that has been trained with translations can only deal reasonably well with a single kind of rotation (even though it is still outperformed by a properly trained network). When two rotations are included then at least one of them must have been trained.

Our results also support the assumption that in-depth rotation is harder to extract and learn than in-plane rotation. This is not surprising, since mathematically the effect of in-plane rotation on the input image is simpler to describe than in-depth rotation. Generalization across different objects is especially hard for in-depth rotation, since the effect on the input image can be highly dependent on individual object details like shape. However, for the two object classes that we used it is at least theoretically possible to find general rules for the in-depth rotation (e.g., based on the fact that the two sides of a fish are very similar).

Cluttered Scenes

So far we only used a white background without distractors. In principle the model should be able to deal with changing background to some extent, since those changes typically happen on a different timescale than those of object identity. While the background properties would enter the model output, the relevant object variables should still be present as well. One could also artificially render the model invariant to the background or distractors by rapidly changing them [Einhäuser et al., 2005], thus sacrificing some realism. Unfortunately our tests in that direction were not very promising. The reason might be that the computational power of the network was not sufficient. Furthermore, in our simulations, contrary to those in [Einhäuser et al., 2005], objects were not centered in the input image, such that a suppression of peripheral areas is no solution to cope with clutter influence. In general we believe that recognizing objects in cluttered scenes cannot be solved with the slowness principle in a feed-forward network. It seems necessary to employ top-down mechanisms for that, which will be explored to some extend in Part II of this thesis.

3.6 Discussion

We have demonstrated how the extraction of slowly varying signals leads to relevant latent variables coding for object identity and pose. In Section 3.5.1 we have shown that these representations in principle can encode object identity and pose independently of each other without any supervised postprocessing. As the time structure of object presentation determines the learned representations, behavior determines representations in our model. For instance, in our model rotation invariance increases for an object if it is rotated rather quickly and switched rarely (similar to a child turning an object in its hands).

However, the results in Section 3.5.2 show that the theoretically determined optimal solutions in our model mix for higher numbers of objects. As the stimuli get more complex we find that the SFA outputs start to deviate from the theoretical ideal. A relatively simple supervised postprocessing step can then still lead to good pose estimation and classification rates. With this method the desired invariance can be recovered for most of the features. The only broken invariance is the object identity dependency for the in-depth and in-plane angles, but they are still invariant with respect to position. We have demonstrated supervised classification and regression, but unsupervised steps (e.g., clustering) or reinforcement learning, as demonstrated in Chapter 5, are an alternative. Nevertheless, as a small number of slow features on the top of the hierarchy code for object identity and pose, there are few parameters to fit in a supervised postprocessing step and thus little training data is necessary, which minimizes the number of labeled training examples.

3.6.1 Comparison with Neural Structures

The primate visual system is organized in two hierarchical pathways. Classically, the ventral pathway is assumed to perform invariant object recognition (“what pathway”). This pathway involves the cortical areas V1 (primary visual cortex), V2, V4, and IT (inferotemporal cortex). On the way from V1 to IT, neurons show increasing receptive field size, stimulus specificity and invariance. At the top of this hierarchy, in IT, many neurons code for objects with high invariance to position, angle, scale etc. The dorsal pathway is assumed to perform complementary spatial computation (“where pathway”) [Ungerleider and Mishkin, 1982]. Most models of invariant object recognition are inspired by this interpretation of the ventral path and implement mechanisms for hard-coded position invariance [e.g., Fukushima, 1980; Wersing and Körner, 2003; Rolls and Stringer, 2006], which successfully reduces view-dependence of their representations. There is, however, some more recent evidence against a complete segregation of "what" and "where" information in the macaque brain [e.g., Hung et al., 2005; Lehky et al., 2008], showing that information about position and size of objects is also represented in the inferotemporal cortex of macaques, i.e., in a top layer of the ventral pathway. This latter view is more consistent with our model, which performs invariant object recognition and simultaneous pose estimation. We conjecture that object identity and pose are tightly coupled: pose estimation requires information on object identity (or at least object category) and object identity needs to be inferred from an object's view in a specific pose. Therefore, an early segregation of identity and spatial configuration into separate processing streams would likely require redundant computing hardware.

We provide no layer-by-layer comparison of our model with the ventral stream, because such a comparison would not only require that the computational principle is similar but that also the specific implementation is comparable. Specifically, the computational power of a layer in our model is determined by the employed nonlinearity, which is hard to estimate for neural substrate and we thus did not try to match it against that of the visual system. Nevertheless, some model properties can be matched to the visual system or can serve as predictions for testing the model. The input data for our model consists of a high-dimensional pixel images, which could be associated with a retinal representation. The first layer of our model computes quadratic functions on patches of these images. Earlier work has established that these functions closely approximate V1 complex cell properties [Berkes and Wiskott, 2005]. On the top of the ventral stream, in IT, we predict neurons coding for invariant object representations as well as neurons representing object pose. Although information on object pose might be mixed with object identity information, we predict that a linear regression should be able to extract the 3D rotational angles from a population of IT neurons for well-known objects. Furthermore, in a system optimizing a slowness function, the presentation statistics of newly seen objects should influence object representation. For example, frequent switching of presented views of two distinct objects should make the representations of the two objects in IT more similar. This matches recent experimental findings in monkeys [Li and DiCarlo, 2008, 2010]

A comparison of intermediate layers of our model with the ventral stream is much harder. The interpretation of the exact functional role of the intermediate layers in our model is actually an unsolved problem, which is also discussed in Part II of this thesis. Nevertheless, we predict that the Δ-values of the representations decrease towards the top of the ventral hierarchy. For example, the presentation of a rotating object should elicit more quickly varying responses in V1 than in V4 or in IT.

3.6.2 Feedback

Our model contains no feedback connections, which is in clear contrast to the massive feedback connections present in the primate visual system [Kennedy et al., 2000]. However, the latency results reported in [Nowak and Bullier, 1997] and [Hung et al., 2005] indicate that the visual system can work in a fast feed-forward mode. In this chapter we have shown how a model without feedback can indeed compute view-invariant object-specific representations as well as object poses. Our model operates, however, in a restricted regime without attention, scene clutter or top-down object bias, which might all require a feedback system. We partly address the issue of feedback in Part II of this thesis, where we try to extend our model to accommodate top-down processes.

3.6.3 Related Work

The problem of invariant object recognition has been approached from two sides in the past [Wiskott, 2006]. One approach is mainly motivated by the "biological implementation" in the (primate) brain with a focus on biological realism, generality and unsupervised learning but was so far mostly limited to very simple stimuli. The other approach comes from computer vision, uses sophisticated machine learning approaches, works for complex stimuli, is highly adapted to a specific problem but is often not biologically plausible.

According to the classification by [Rolls and Deco, 2002], our network belongs to the category of "feature hierarchy based computational object recognition devices". Such systems extract increasingly complex features in a hierarchical system. In contrast to "flat" feature space systems that are typically insensitive to scrambled input images [e.g., Mel, 1997], object recognition in primates is highly sensitive to the relative location of features, for example position of eyes and the nose in a face [Rolls et al., 1994; Vogels, 1999; Grill-Spector et al., 1998]. The increasing receptive field size in hierarchical feature systems naturally preserves susceptibility to scrambling, as each layer is sensitive to local arrangements of spatial features.

The slowness principle has been applied in many models of sensory systems in the past [e.g., Földiák, 1991; Stone and Bray, 1995; Bartlett and Sejnowski, 1998; Becker, 1999; Kayser et al., 2001; Rolls and Deco, 2002; Wiskott and Sejnowski, 2002; Berkes and Wiskott, 2005; Hipp et al., 2005; Wyss et al., 2006; Franzius et al., 2007]. Some of these unsupervised models learn on naturalistic input data and generate representations similar to neural codes, like V1 simple cells, V1 complex cells, or place cells, head direction cells and spatial view cells in the hippocampal formation. A number of publications have used the slowness principle for invariant object or face recognition [Wallis and Rolls, 1997; Becker, 1999; Wiskott and Sejnowski, 2002; Stringer and Rolls, 2002; Rolls and Stringer, 2006] but, to our knowledge, pose extraction has not been demonstrated before. Furthermore, most models use simplified stimuli and apply only one transformation, i.e., rotation or translation.

In the remaining part of this chapter we discuss some selected models in a little more detail. Some additional comparisons with other models can be found in [Franzius et al., 2011]. Furthermore, we discuss other approaches in Section 5.4.1 and Section 8.3 (e.g., the HMAX model [Riesenhuber and Poggio, 1999]).

The VisNet Model

VisNet [Rolls and Deco, 2002] is a model based on the trace rule [Földiák, 1991; Rolls, 1992; Wallis and Rolls, 1997], which is closely related to Slow Feature Analysis [Sprekeler et al., 2007]. As in our model, features are learned in an unsupervised manner from the statistics of the network's stimuli. Only the weights on the lowest layer are initialized explicitly to Gabor-like structures. Like our model, VisNet can learn a position-invariant (or view-invariant) but object-specific code. In contrast to our model, only a single invariant representation is extracted (i.e., object identity) and the other parameters (e.g., object position, rotation angles, lighting direction) are discarded. Furthermore, VisNet has a number of extra variables that need to be tuned per layer (e.g., the learning rate α and the trace length η). A further difference comes from the use of sparse coding in VisNet by means of lateral inhibition and the winner-takes-most architecture. In our model, redundancy reduction comes from the decorrelation constraint in SFA, which is performed in each node, but no spatial decorrelation over adjacent nodes in the grid of a layer is enforced.

In [Stringer and Rolls, 2002], VisNet is applied to views of six spheres rotated in-depth over an angle of 120°. The spheres have different surface features consisting of different configurations of three line segments. After learning with the trace rule, a part of the neurons in the top layer of the model code for sphere identity and are invariant to the rotational angle.

[Stringer et al., 2006] uses a variant of VisNet with a purely Hebbian learning rule. Interestingly, in one of their control experiments the trace rule is applied to stimuli that rapidly change object identity and slowly change the angle. The authors find that the resulting representation codes poorly for object identity.

Other Examples of Models based on Slowness

The work by [Becker, 1999] introduces a biologically motivated generative hierarchical model of cortical visual processing. The model performs unsupervised clustering with a Gaussian mixture model gated by temporal context. In one simulation, the network performs viewpoint-independent face categorization on a set of ten faces based on temporal continuity of face presentation during the training phase. In a second simulation, the model learns a pose-specific code after relaxing the temporal context signal. This architecture focuses on a more general "context signal", which might be a temporal one. In contrast to our model, it seems hard to predict the nature of the invariance properties learned by the model for a specific set of architectural design choices.

In [Einhäuser et al., 2005] a similar learning rule as in our model is applied, which is basically a gradient descent on the SFA optimization problem from Chapter 2. Similar to VisNet, the first layer uses static Gabor wavelets to model complex cells in V1, whereas the second layer is optimized according to the slowness principle. As this model uses only two layers, the absolute sum of all outputs of the first layer with the same size and orientation is used as input for the second layer, which introduces a hard-coded translation invariance. After optimization, output units represent object identity independently of viewpoint. Between 10 and 50 training objects from the standard COIL database are used as stimuli in front of a homogeneous or cluttered background. The COIL database contains photos of real-world objects centered and rotating on a turntable. Thus object transformations combine in-depth rotation, changing object identity, and some rescaling, but no translation. The model uses color information (as RGB channels). In our model, we intentionally reduced the input data to gray scale representations because color histograms alone can often be highly informative about object identity.

4 Application to other Stimuli

In this short chapter we want to further explore the performance of hierarchical SFA networks for different kinds of stimuli, presenting some unpublished results. The parameters of the SFA models used here exactly match those from the previous chapter (as described in Section 3.2).

4.1 Leaf Outlines

This stimulus set consists of leaf outlines from different tree species. The images were created by Frank Jäkel (who kindly gave permission to use them here), by scanning dried leafs and converting them to silhouettes. They were used in psychophysical experiments to analyze the classification behavior of human test subjects (not yet published). Overall there are leafs from 30 different tree species. For each kind of tree there are between 21 and 180 individual leaf silhouettes, with an average of 104. The classification task therefore consists of assigning leafs to their tree species, discarding the individual variations of each leaf. This can be interpreted as another kind of invariance, similar to rotation or scale invariance (though the leafs are normalized in size and orientation). Some examples are shown in Figure 4.1. There are several leaf types that look very similar to the untrained observer, so the classification is not an easy task.

4.1.1 Methods

For the network training we generated two different sets of stimuli. The first set is very similar to the fish or sphere stimuli and is used for the lower two network layers. A single leaf is shown and moves around according to the random walk algorithm from Section 3.3. These transformations should enable the lower network layers to develop similar features as in the previous experiments (hopefully roughly matching the characteristics of the visual system). In each time step there is also a 0.2% chance of switching to another tree species and a 5% chance of switching to another leaf from the same tree.

In the stimulus set for the two top layers the translations are dropped, leaving the rotation as the only continuous transformation. The individual leaf is now switched in each frame, while the chance of switching to another tree type remains at 0.2%. The network layers therefore have a strong incentive to learn invariance with respect to individual leaf deviations. The rotation transformation is not really needed at this point, but it might help the network to deal with slight misalignments in the original images.

The original leaf images are very large (from several hundred to several thousand pixels wide), but most features are still clearly visible in our 155×155 pixel training

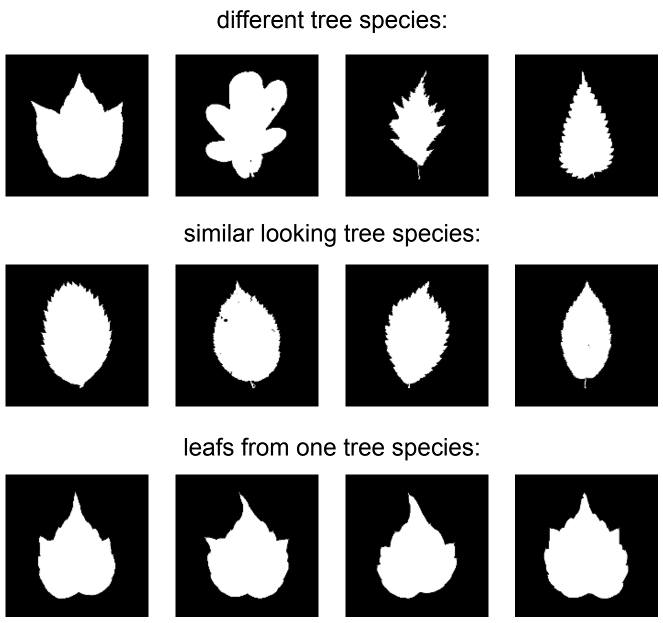


Figure 4.1: **Different leaf stimuli as presented to the SFA network.** The first two rows show leafs that are from different trees. Those in the first row are easy to tell apart, while for those in the second row this is much harder. The third row contains four different leafs from one tree species, illustrating the individual variations.

images. Only half of the 30 tree species are shown during the network training, which enables the testing of its generalization capabilities.

4.1.2 Results

To test the network classification performance each individual leaf is shown once, with no movement or rotation. For the actual classification we use the same method as described in Section 3.4, so half of the test data is used to train a classifier on top of the SFA network. The other half is then used to measure the classification performance, as we did earlier for the fish and sphere stimuli. Due to the limited number of data points (which are not enough to train the Gaussian classifier) we only provide the results for the nearest center classifier. It reaches a hit rate of 95% on the first 15 tree species, which were shown during training. On the untrained second half this drops to 85% and the performance on all 30 species is 86%. Most of the misclassifications happen for trees with leaves that look very similar to a human observer.

4.2 Multiple Objects

In the previous experiments the SFA network was only confronted with single objects, both during training and testing. One way to deal with multiple objects or distracters is to train the SFA network with such multi-object stimuli. If the network is trained with two objects that move around independently then theory predicts that the pose information for both objects is encoded in the network output. This approach is used in the next chapter (see Figure 5.2) and does indeed work. However, the requirement of dedicated training with the specific number of objects is a serious limitation. We therefore also tested how the SFA network reacts to multiple stimuli without this special training. Furthermore, we want to test how the model reacts to transitions between different objects, which could be relevant for the implementation of attentional mechanism (e.g., gain modulation for individual objects).

4.2.1 Methods

For the testing of our model we generated stimuli in which two fish objects are shown simultaneously. The positions of the two fish were chosen such that there is no overlap. We also generated “intermediate” stimuli in which the two objects are shown with varying contrast. Technically this was achieved with alpha blending, for which the so called alpha value defines the transparency of an object. The alpha values range from zero (invisibility) to one (standard view with full contrast).

In the first part of the experiment we used the model from the previous chapter, which had been trained with the standard fish stimuli (only one fish, alpha always one). For the second part we trained a new SFA network with stimuli in which the contrast of the single fish object varies slowly. The variation of the alpha value uses the same random walk procedure and the same parameters (i.e., the same speed) as for the translations. Therefore the model should treat the alpha value similar to the position (but obviously this has to break down when the fish becomes invisible). The same number of training stimuli was used as in the previous chapter.

4.2.2 Results

To test the performance of the network trained with varying contrast we used the same test stimuli as in the previous chapter (alpha always one). The performance of this model is only slightly worse than before, with a Gaussian classifier performance of 93.6% for all 25 fish (compared to 95.2% for the standard model).

For the tests with two objects we used the previously described stimuli in which two non-overlapping fish are shown with different contrast. Both alpha values were varied with a fixed step size, so the value pairs describe a 2D grid. We then recorded the 512 dimensional network output for all these grid points. This effectively creates an embedding of the 2D grid into the 512 dimensional output space. Principal Component Analysis (PCA) was then used to project the grid points onto the first two principal components, allowing us to plot the resulting 2D structure. The results are shown in

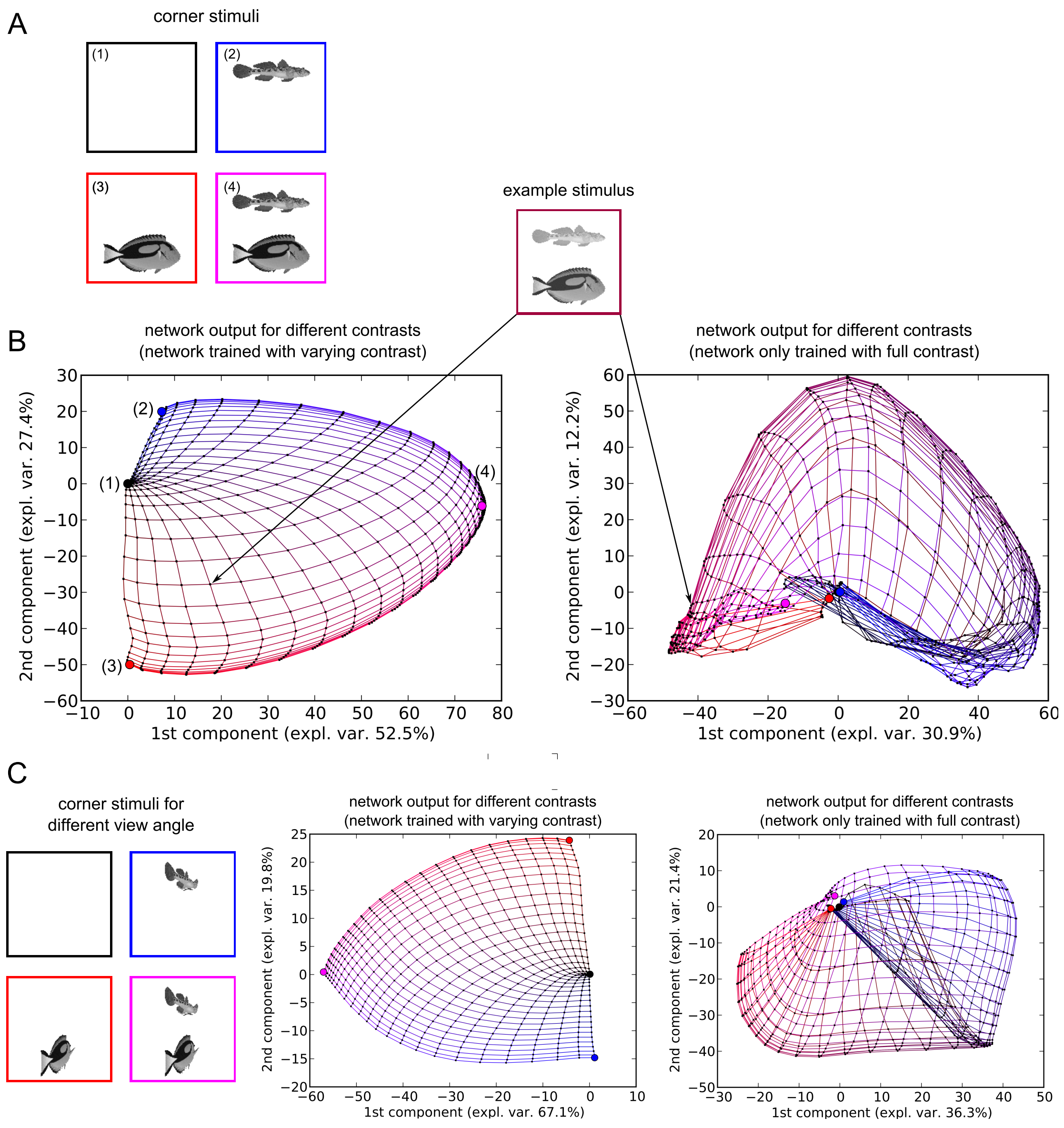


Figure 4.2: **Network output for two fish with different contrast.** **A** shows the corner points of the 2D alpha grid, at which the alpha values are either zero or one. The alpha values are represented by RGB colors, with the alpha value of the upper fish defining the blue channel value and the lower one defining the red channel (the green channel is set to zero). The first plot in **B** shows the 2D PCA projection of the output from the network that was trained with contrast transformations. The second plot is from the standard network. The corner points of the grid are indicated by numbers, corresponding to the numbers in **A**. **C** contains the grid plots for a different fish angle.

Figure 4.2, for two different angles of the fish. They demonstrate that the output of the standard network varies a lot when being confronted with these new stimuli. The resulting alpha grid in Figure 4.2 is therefore quite irregular, even along the grid edges that correspond to a single visible object. The output from the second network (which was trained with varying contrast) on the other hand is fairly stable, with the grid being much closer to its ideal 2D shape.

4.2.3 Conclusion

An SFA network that is trained with varying contrast naturally develops some invariance with respect to this transformation. This even holds if multiple objects are shown, which is certainly helped by the fact that the receptive fields in the lower layers are only large enough to contain parts from a single object. The predictable nature of the network output could be important for attentional mechanisms that are based on gain modulation. By adjusting the effective contrast for different areas of the input image it would be possible to move the focus of attention smoothly between different objects, without generating abnormal network output during the intermediate steps.

A more general conclusion from these results is that the network robustness can be increased by subjecting it to more transformations in the training stimuli. The polynomial nature of our network means that untrained stimuli can result in large variations of the output signals, as seen in Figure 4.2. This seemingly contradicts the generalization capabilities of the network that were demonstrated in the previous chapter. However, the control experiments in Section 3.5.3 already indicated that the SFA network has problems to produce a good output signals (that can be interpreted with a simple linear regression) for untrained continuous transformations.

4.3 Depth Images

Another variation of the fish stimuli is shown in Figure 4.3. These images are based on the depth information of the 3D objects. They were originally intended as a first step towards other sensorial modalities. For example, this data could be used for whisker simulations. Since the fish and sphere stimuli are generated with 3D objects it is technically very simple to extract the depth image from the rendering process.

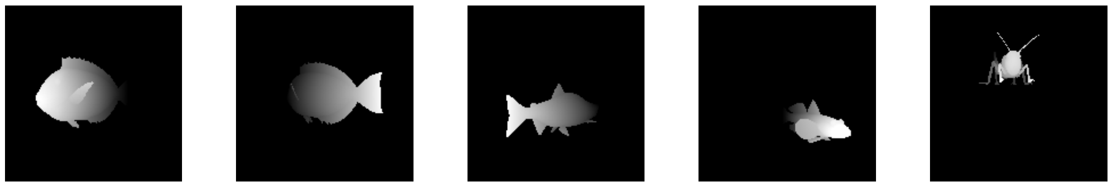


Figure 4.3: **Stimuli rendered from depth information.** The brightness of each pixel represents the distance to the virtual camera. The last image shows the grasshopper object, which was added in this stimulus set.

The training stimuli consisted of 9 fish objects together with a grasshopper (as a tribute to the original whisker idea). Instead of 2D movement the objects are moved around in a 3D box. In addition we applied the standard in-depth rotation, but no scale transformation. Instead of the blank frame between different objects the switch happens each time the object reaches the back side of the 3D confinement box. In addition the depth information is scaled such that the objects also become practically invisible near the back side. This gives the impression of different objects sliding into view and vanishing, which is arguably more elegant than our earlier method.

In the test stimuli the objects were restricted to the front 2D plane, in order to provide the network with a good view. Five new fish objects were introduced during testing (for a total of 15).

The resulting network performance is generally on par with the earlier results from our normal stimuli. For example, the Gaussian classifier achieved a hit rate of 92.3% on all 15 objects. While not surprising this provides an additional verification for the flexibility of hierarchical SFA networks.

5 Combining Hierarchical SFA with Reinforcement Learning

Humans and animals are able to learn complex behaviors based on a massive stream of sensory information from different modalities. Early animal studies have identified learning mechanisms that are based on reward and punishment such that animals tend to avoid actions that lead to punishment whereas rewarded actions are reinforced. However, most algorithms for reward-based learning are only applicable if the dimensionality of the state-space is sufficiently small or its structure is sufficiently simple. Therefore, the question arises how the problem of learning on high-dimensional data is solved in the brain. In this chapter we present a biologically plausible generic two-stage learning system that can directly be applied to raw high-dimensional input streams.

The system is composed of a hierarchical SFA network as described in Section 3.2 for preprocessing and a simple neural network on top that is trained based on rewards. We demonstrate with computer simulations that this generic architecture is able to learn quite demanding reinforcement learning tasks on high-dimensional visual input streams in a time that is comparable to the time needed when an explicit highly informative low-dimensional state-space representation is given instead of the high-dimensional visual input. The learning speed of the proposed architecture in a task similar to the Morris water maze task is comparable to that found in experimental studies with rats. This study thus supports the hypothesis that slowness learning is one important unsupervised learning principle utilized in the brain to form efficient state representations for behavioral learning.

This model is the result of a collaboration with Robert Legenstein, who contributed all the reinforcement learning parts. The results were published in [Legenstein et al., 2010], which is the basis for this chapter.

5.1 Introduction to Reinforcement Learning

The nervous system of vertebrates continuously generates decisions based on a massive stream of complex multimodal sensory input. The strength of this system is based on its ability to adapt and learn suitable decisions in novel situations. Early animal studies have identified learning mechanisms that are based on reward and punishment such that animals tend to avoid actions that lead to punishment whereas rewarded actions are reinforced. The study of such reward-based learning goes back to Thorndikes law of effect [Thorndike, 1911]. Later, the mathematically well-founded theory of reinforcement learning, which describes learning by reward, has been developed [Bertsekas and Tsitsiklis, 1996; Sutton and Barto, 1998].

In a general reinforcement learning problem, an agent senses the environment at time t via a state $s(t) \in \mathcal{S}$, where $\mathcal{S}$ is the state space of the problem. The agent then chooses an action $a(t)$, which leads to state $s(t+1)$ according to some (in general probabilistic) state-transition relation. The agent also receives some reward signal $R(t+1)$, which depends probabilistically on the state $s(t+1)$. By choosing an action $a(t)$ the agent aims at maximizing the expected discounted future reward

$$E\left[\sum_{i=1}^{\infty} \gamma^i R(t+i)\right], \tag{5.1}$$

where $E[\cdot]$ denotes the expectation and $0 \ll \gamma < 1$ is some discount rate. This general theory has a huge influence on psychology, systems neuroscience, machine learning, and engineering, and numerous algorithms have been developed for the reinforcement learning problem. By utilizing these algorithms, many impressive control applications have been developed.

Since physiological experiments are consistent with quite standard reward-based learning schemes, it is reasonable to speculate that the superior learning capabilities of animals is to a high degree based on the ability to autonomously extract relevant features from the input stream such that subsequent reward-based learning is highly simplified (though we note that the distinction between feature extraction and reward-based learning is most likely not so strict in the brain).

In fact, one of the most crucial design questions in the design of a reinforcement learning system is the definition of the state space $\mathcal{S}$. Most reinforcement learning algorithms are only applicable if the state space of the problem is sufficiently small. Thus, if the sensory input to a controller is complex and high-dimensional, the first task of the designer is to extract from this high-dimensional input stream a highly compressed representation that encodes the current state of the environment in a suitable way such that the agent can learn to solve the task. In contrast, the nervous system is able to learn good decisions from high-dimensional visual, auditory, tactile, olfactory, and other sensory inputs autonomously.

The autonomous extraction of relevant features is of course just the problem that was addressed in the previous chapters. After the hierarchical SFA networks have been successfully applied to invariant object recognition it is only natural to test them in other applications. The combination with reinforcement learning is paticaluarly interesting since the supervision in this case only consists of a single reward signal. In terms of biological plausibility this is a significant improvement when compared to the supervised analysis techniques that were discussed in Section 3.4.

In this chapter, we propose a learning system where the state space representation is constituted autonomously by SFA. A subsequent neural circuit is then trained by a reward-based synaptic learning rule that is related to policy gradient methods or Q-learning in classical reinforcement learning. We apply this system to two closed-loop control tasks where the input to the system is high-dimensional raw pixel data and the output are motor commands. We thus show for two control tasks on high-dimensional visual input streams that the representation of the SFA output is well suited to serve as a state-representation for reward-based learning in a subsequent neural circuit.

5.2 Methods

The learning system considered in this chapter consists of two components, a hierarchical SFA network which is identical to that described in Section 3.2 and a subsequent control network, see Figure 5.1. The SFA network reduces the dimensionality of the state-space from 24025 to a small number n. For example, we used $n = 512$ in Chapter 3. In the following experiments the n was generally reduced to 64 or less (by picking only the slowest n components on the top level). The decisions of the subsequent control network are based solely on the features extracted by the SFA network.

For each of the two tasks discussed in this paper (Morris water-maze and variable-targets) we trained a dedicated hierarchical network. The number of training samples and the training itself was done in the same way for both tasks, only the content of the training samples was different.

5.2.1 Environment

We tested this learning system on two different control tasks where an agent (a fish) navigates in a 2D environment with analog state- and action-space: a task similar to the Morris water-maze task [Morris et al., 1982] and a variable-targets task (see Section 5.2.3). The state of the universe at time t (see below for details) was used to render a 155×155 dimensional 2D visual scene that showed the agent (a fish; for one of the tasks two fish-types with different visual appearance were used) at a position $\mathbf{p}(t) \in [-1, 1]^2$ and potentially other objects, see Figure 5.2. This visual scene constituted the input to the learning system. These tasks are to be seen as generic control tasks of reasonable complexity. The bird's eye perspective used here is of course not realistic for animal agents. As demonstrated in [Franzius et al., 2007] our model should also be able to deal with a first-person perspective, especially in the Morris water-maze. For the variable-targets task this would introduce some complications like the target not being in the field of view or being hidden behind the distractor. On the other hand it would simplify the task, since the agent would not need to know its own position and angle (it could simply center its field of view on the target).

For the training of the system, we distinguish two different phases. In a first phase the SFA network is trained. In this phase, the fish, the target, and the distractor are floating slowly over the 2D space of the environment. The type of fish is changed from time to time. In a second phase the control circuit is trained. This phase consists of several learning episodes, an episode being one trial to reach a defined target from the initial fish-position. An episode ends when the target is reached or when a maximum number of T_{max} time-steps is exceeded.

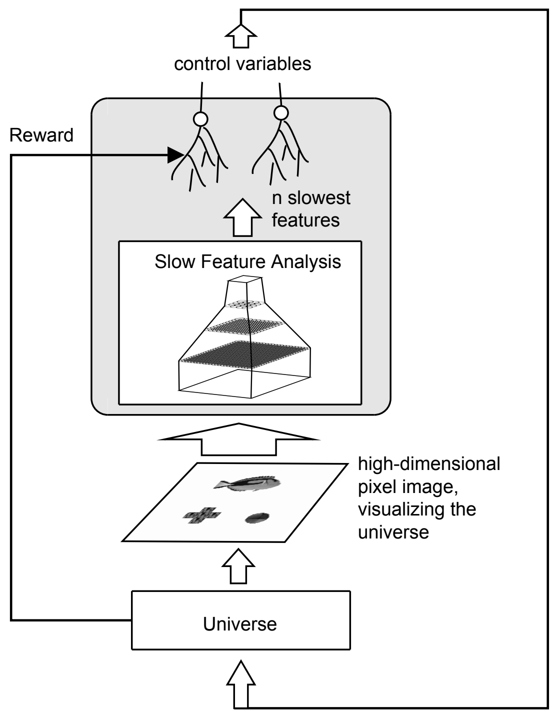


Figure 5.1: **The learning system and the simulation setup.** The learning system (gray box) is based upon a hierarchical slow-feature analysis network, which reduces the dimensionality of the high-dimensional visual input. This reduction is trained in an unsupervised manner. The extracted features from the SFA network serve as inputs for a small neural network that produces the control commands. This network is trained by simple reward-modulated learning. We tested the learning system in a closed-loop setup. The system controlled an agent in an environment (universe). The state of the environment was accessible to the learning system via a visual sensory stream of dimension 155 $\times$ 155. A reward signal was made accessible to the control network for learning.

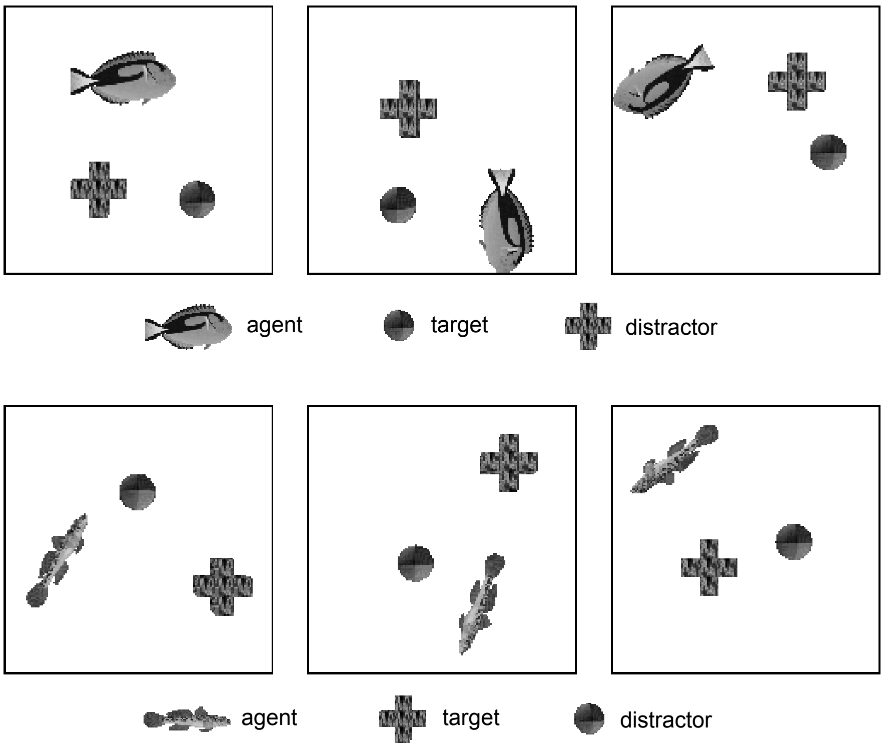


Figure 5.2: **Examples for the visual input to the learning system for the variable-targets task.** The scene consists of three objects, the agent (fish), an object that indicates the location of the target, and a second object that acts as a distractor. As indicated in the figure the target object depends on the fish identity. For the fish identity shown in the upper panels the target is always the disk, whereas for the other fish identity, the target is the cross. In the visual input for the water-maze task the target and the distractor are not present, and the agent representation is the non-rotated image of the fish-type shown in the upper panels.

Training Stimuli of the Hierarchical Network

Training sequences for the two tasks were created with the same random walk procedure that was described in Section 3.3. The velocity distribution was the same for all objects (max. velocity of 0.06 and a max. update of 0.01). For the in-plane angle of the agent the max. velocity was 0.04 with a max. update of 0.01 (in radiant measure).

For the variable-targets task training, the objects were given a radius so that they bounce off each other. The radii were chosen such that there could be only a small visible overlap between any two objects (radius of 0.4 for the agent, 0.2 for target and distractor). In each time step the agent identity was switched with a probability of 0.002.

5.2.2 Neural Circuits for Reward-Based Learning

We employed neural implementations of two reinforcement learning algorithms, one is based on Q-learning and one is a policy-gradient method. Since this part of the simulation was done by Robert Legenstein I refer to [Legenstein et al., 2010] for the detailed description and the parameter values.

Neural versions of Q-learning have been used in various previous works on biological reward-based learning, see e.g., [Foster et al., 2000; Sheynikhovich et al., 2005]. The popularity of Q-learning stems from the finding that the activity of dopaminergic neurons in the ventral tegmental area is related to the reward-prediction error [Montague et al., 1996; Schultz et al., 1997; Schultz, 1998], a signal that is needed in Q-learning [Sheynikhovich et al., 2005]. In Q-learning, decisions are based on a so-called Q-function that maps state-action pairs (s, a) onto values that represent the current estimate of the expected total discounted reward given that action a is executed at state s. For a given state, the action with highest associated Q-value is preferred by the agent. However, to ensure exploration, a random action may be chosen with some probability. We implemented the neural version of Q-learning from [Sheynikhovich et al., 2005] where the Q-function is represented by a small ensemble of neurons and parametrized by the connection weights from the inputs to these neurons. The system learns by adaptation of the Q-function via the network weights. In the implementation used in this work this is achieved by a local synaptic learning rule at the synapses of the neurons in the neuron ensemble. The global signal that modulates local learning is the temporal difference error (TD-error). We do not address the question how this signal is computed by a neuronal network. Several possible mechanisms have been suggested in the literature [Houk et al., 1995; Schultz, 1998; Berns and Sejnowski, 1998]. The Q-function was represented by a set of $N = 360$ linear neurons that receive information about the current state from the output $\mathbf{x}(t)$ of the SFA circuit.

The second learning algorithm employed was a policy gradient method. In this case, the action is directly given by the output of a neural network. Hence, the network (which receives as input the state-representation from the SFA network) represents a policy (i.e., a mapping from a state to an action). Most theoretical studies of such biologically plausible policy-gradient learning algorithms are based on point-neuron models where synaptic inputs are weighted by the synaptic efficacies to obtain the membrane voltage. The output $y_i(t)$ of the neuron i is then essentially obtained by the application of a nonlinear function to the membrane voltage. A particularly simple example of such a neuron model is a simple pseudo-linear rate-based model where a nonlinear activation function $f : \mathbb{R} \to \mathbb{R}$ (commonly sigmoidal) is applied to the weighted sum of inputs $x_1(t), \ldots, x_n(t) \in \mathbb{R}$:

$$y_i(t) = f\left(\sum_{j=1}^{n} w_{ij} x_j(t) + w_{i0} + \xi_i(t)\right). \tag{5.2}$$

Here, w_{ij} denotes the synaptic efficacy (weight) of synapse ij that projects from neuron j to neuron i, w_{i0} is a bias, and $\xi_i(t)$ denotes some noise signal. We assume that a reward signal $R(t)$ indicates the amount of reward that the system receives at time t. Good actions will be rewarded, which will lead to weight changes that in turn make such actions more probable. Reinforcement learning demands exploration of the agent, i.e., the agent has to explore new actions. Thus, any neural system that is subject to reward-based learning needs some kind of stochasticity for exploration. In neuron model (5.2) exploration is implemented via the noise term $\xi_i(t)$.

A single neuron of type (5.2) turns out to be too weak for some of the control tasks considered in this article. The standard way to increase the expressive power is to use networks of such neurons. The learning rule for the network is then unchanged, each neuron tries to optimize the reward independently from the others [Seung, 2003] (but see [Urbanczik and Senn, 2009]). It can be shown that such a greedy strategy still performs gradient ascent on the reward signal. However, the time needed to converge to a good solution is often too long for practical applications. We therefore propose a learning rule that is based on a more complex neuron model with nonlinear dendritic interactions within neurons [Poirazi et al., 2003] and the possibility to adapt dendritic conductance properties [Losonczy et al., 2008].

In our simulations, we needed two control variables, one to control the speed $v(t)$ of the agent and one for its angular velocity $v_\alpha(t)$. Each control variable was computed by a single neuron of this type. The nonlinearity in the branches was the tangens hyperbolicus function $\tanh : \mathbb{R} \to (-1, 1)$. Also a logistic sigmoidal was tested which is a scaled version of the tangens hyperbolicus to the image set $(0, 1)$. Results were similar with a slight increase in learning time. The nonlinearity at the soma was the tangens hyperbolicus for the angular velocity $v_\alpha(t)$ and a logistic sigmoid $\text{logsig} : \mathbb{R} \to (0, 1)$ for the speed $v(t)$. The noise signal $\xi_i(t)$ was drawn independently for each neuron and at each time step from a uniform distribution in $[-0.5, 0.5]$.

5.2.3 Tasks

We tested the system on two different control tasks: a task similar to the Morris water-maze task and a variable-targets task.

Morris Water Maze Task

In the experimental setup of a Morris water maze task [Morris et al., 1982], a rat swims in a milky liquid with a hidden platform underneath the liquid surface. Because the rodent tries to avoid swimming in the liquid, it searches for the platform. This task has been modeled several times [Foster et al., 2000; Sheynikhovich et al., 2005; Vasilaki et al., 2009; Potjans et al., 2009].

In order to be able to compare the results to previous studies, we modeled the Morris water maze task in our standard setup in the following way: We used only a single fish type and a fixed target position at $(0, 0)^T$. Only the fish but not the target was visible in the visual input to the learning system. There was only one control signal which controlled the direction of the next movement. At each time step, the fish was moved by 0.1 length units in the direction given by the controller. The position of the fish was hard-bounded by -1 from below and 1 from above after each update such that it stayed within $[-1, 1]$. In this setup, the fish was always oriented in the same direction (facing to the right), i. e., the fish was not rotated in the visual input. The target was reached by the agent if it was within a radius of 0.2 of the target position.

The reward signal was defined such that reaching the target at time t resulted in a positive reward $R(t) = 1$, hitting the wall at time t resulted in a negative reward $R(t) = -0.1$, and the reward signal was 0 at other times. Hence this is a setup with sparse rewards. An episode ended when the target was reached or after $T_{max} = 450$ time-steps have evolved (this is consistent with [Vasilaki et al., 2009] where a simulation time step was interpreted as a 200 msec time interval).

Variable-Targets Task

In order to explore the general applicability of the system we investigated a more demanding task with several objects in the visual input and two different types of fish of varying orientation.

In this task, the state of the agent at time t was defined by its identity $I(t) \in \{0, 1\}$ (this corresponds to two types of fish, each with a unique visual appearance in the visual input stream, see Figure 5.2), its position $\mathbf{p}(t) \in [-1, 1]^2$, and its orientation $O(t) \in [0, 2\pi)$. Additionally to the agent, there were two objects in the universe, one of them acting as the target and the other as a distractor. One object appeared as a "cross" in the visual scene and the other object as a "disk" (see Figure 5.2). The state of object i was defined by its position $\mathbf{t}_i \in [-1, 1]^2$. The current state of the universe at time t was given by the collection of these variables.

The output of the learning system were two control variables to control the agent in the environment, a speed signal $v(t) \in (0, 1)$ and a signal $v_\alpha(t) \in (-1, 1)$ for angular velocity. These signals were used to update the orientation $O(t)$ and position $\mathbf{p}(t) = (p_1(t), p_2(t))^T$ of the fish

$$O(t) = [O(t-1) + k_a v_\alpha(t)] \bmod 2\pi, \tag{5.3}$$

$$p_1(t) = p_1(t-1) + k_v v(t) \cos O(t), \tag{5.4}$$

$$p_2(t) \quad = \quad p_2(t-1) + k_v v(t) \sin O(t), \tag{5.5}$$

where $k_a = 0.5$ and $k_v = 0.1$ are scaling constants. When the agent hit the boundaries of the environment (i.e., when $p_1(t)$ or $p_2(t)$ were below -1 or above 1), the movement was mirrored. For each training episode the object positions, fish orientation, and fish identity were initially chosen randomly from the uniform distribution in their range. However, when an object was initially less than 0.2 away from the other object or the fish (which likely produced a visual overlap), a new initial state was drawn. The object positions were then fixed. Each fish identity had a different object serving as the target, such that the fish of type A was associated with the "cross" whereas fish-type B was associated with the "disk". The task was to navigate the fish to the target object for the given fish identity. The current episode ended when the fish reached the target location within some predefined radius ($r_{hit} = 0.4$) or after a maximum of $T_{max} = 100$ time steps were exceeded). Although the other object did not influence the outcome of the task (the fish could swim through it), it was still visible as a distracting stimulus.

The reward signal indicated whether the last action was successful in bringing the agent closer to the target:

$$R(t) = \frac{1}{k_v} \sum_i \delta_{I(t)-i} \left(||\mathbf{p}(t-1) - \mathbf{t}_i(t-1)|| - ||\mathbf{p}(t) - \mathbf{t}_i(t)|| \right), \tag{5.6}$$

where $|| \cdot ||$ denotes the Euclidean norm and $\delta_x = 1$ if $x = 0$ and 0 otherwise. This is a relatively informative reward signal.

5.3 Results

5.3.1 Morris Water Maze Task

We implemented this task with our learning system where the decision circuit consisted of the Q-learning circuit described above. In this task, the $n = 16$ slowest components as extracted by the hierarchical SFA network were used by the subsequent decision network. The results of training are shown in Figure 5.3. The performance of the system was measured by the time needed to reach the target (escape latency).

The system learns quite fast with convergence after about 40 training episodes. The results are comparable to previously obtained simulation results [Foster et al., 2000; Sheynikhovich et al., 2005; Vasilaki et al., 2009] that were based on a state representation by neurons with place-cell-like behavior. Figure 5.3B shows the direction the system chooses with high probability at various positions in the water maze (navigation map) after training. Using only the 16 slowest SFA components for reinforcement learning, the system has rapidly learned a near-optimal strategy in this task. This result shows that the use of SFA as preprocessing makes it possible to apply reinforcement learning to raw image data in the Morris water maze task.

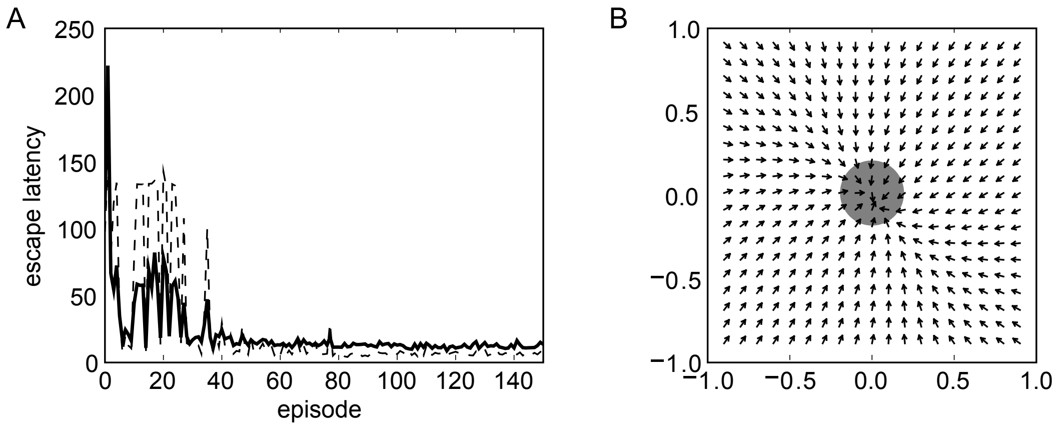


Figure 5.3: **Performance of the learning system in the Morris water maze task with Q-learning.** In **A** the mean escape latency is shown (in simulation time steps) as a function of learning episodes for 10 independent sets of episodes (full thick line). The thin dashed line indicates the standard deviation. **B** shows the navigation map of the system after training. The vectors indicate the movement directions the system would most likely choose at the given positions in the water maze. An episode ended successfully when the center of the fish reached the area indicated by the gray disk.

5.3.2 Variable-Targets Task

The Morris water maze task is relatively simple and does not provide rich visual input. We therefore tested the learning system on the variable-targets task described above, a control task where two types of fish navigate in a 2D environment. In the environment, two object positions were marked by a cross and a disk, and these positions were different in each learning episode. A target object was defined for each fish type and the task was to navigate the current fish to its target by controlling the forward speed and the change in movement direction (angular velocity). The control of angular velocity, the arbitrary target position, and the dependence of the target object on the fish identity complicates the control task such that the Q-learning algorithm used in the water-maze task as well as a simple linear decision neuron like the one of equation (5.2) would not succeed in this task. We therefore trained the leaning system with the more powerful policy gradient algorithm described above on the slowest 32 components extracted by the hierarchical SFA network.

In order to compute the SFA output fast, we had to perform the training of the control network in batches of 100 parallel traces in this task (i.e., 100 training episodes with different initial conditions are simulated in parallel with a given weight vector. After the simulation of a single time step in all 100 episodes, weight changes over these 100 traces are averaged and implemented. Then, the next time step in each of the 100 traces is simulated and weights are updated). When the agent in one of the traces arrived at the target, a new learning episode was initiated in this trace while other traces simply continued. As shown below, the training in batches has no significant influence on the learning dynamics.

Results are shown in Figure 5.4A,B. The reward converges to a mean reward above 0.75 which means that the agent nearly always takes the best step towards the target despite the high amount of noise in the control neurons. Figure 5.5 shows that the trajectories after training were very good. Interestingly, the network does not learn the optimal strategy with respect to the forward speed output. Although it would be beneficial to reduce the forward speed when the agent is directed away from the target, first rotate the agent, and only then move forward, the output of the speed neuron is nearly always close to the maximum value. A possible reason for this is that the agent is directed towards the target most of the time. Thus, the gain in reward is very small and a relatively small fraction of training examples demands low speed.

We compared the results to a learning system with the same control circuit, but with SFA replaced by a vector which directly encodes the state-space in a straight-forward way. For this task with two fish identities and two objects, we encoded the state-space by a vector

$$\mathbf{s}(t) = (p_1(t), p_2(t), \sin O(t), \cos O(t), \delta_{I(t),0}, \delta_{I(t),1}, t_{x,1}, t_{y,1}, t_{x,2}, t_{y,2})^T, \tag{5.7}$$

where $\mathbf{p}(t) = (p_1(t), p_2(t))^T$ is the position of the agent, $O(t)$ is its orientation, $I(t)$ is its identity, and $\mathbf{t}_i(t) = (t_{x,i}(t), t_{y,i}(t))$ is the position of the i^{th} object. Figure 5.4C,D shows the results when the control network was trained with identical parameters but with this state-vector as input. The Performance with the SFA network is comparable to the performance of the system with a highly informative and precise state encoding.

For efficiency reasons, we had to perform the training of the control network in batches of 100 traces (see above). Because no SFA is needed in the setup with the direct state-vector as input, we can compare learning performance of the control network to performance without batches. The result is shown in in Figure 5.4C,D (gray dashed lines). The use of small batches does not influence the learning dynamics significantly.

In the environment considered, movement is mirrored if the agent hits a boundary. Since this helps to avoid getting stuck in corners we performed control experiments where the movement in the direction of the boundary is simply cut off but no reflection happens (i.e., the dynamics of the position $\mathbf{p}(t) = (p_1(t), p_2(t))^T$ of the fish is given by $p_1(t) = \max\{-1, \min\{1, p_1(t-1) + k_v v(t) \cos O(t)\}\}$ and $p_2(t) = \max\{-1, \min\{1, p_2(t-1) + k_v v(t) \sin O(t)\}\}$, compare to equations (5.4),(5.5)). As expected, the system starts with lower performance and convergence takes about twice as long compared to the environment with mirrored movements at boundaries. Interestingly, in this slightly more demanding environment, the SFA network is converging faster than the system with a highly informative and precise state encoding.

In another series of experiments we tested how the performance depends on the number of outputs from the SFA network that are used as input for the reinforcement learning. Since the outputs of the SFA network are naturally ordered by their slowness one can pick only the first n outputs and train the reinforcement learning network on those. For the variable-targets task we tested the performance for 16, 22, 28, 32, and 64 outputs.

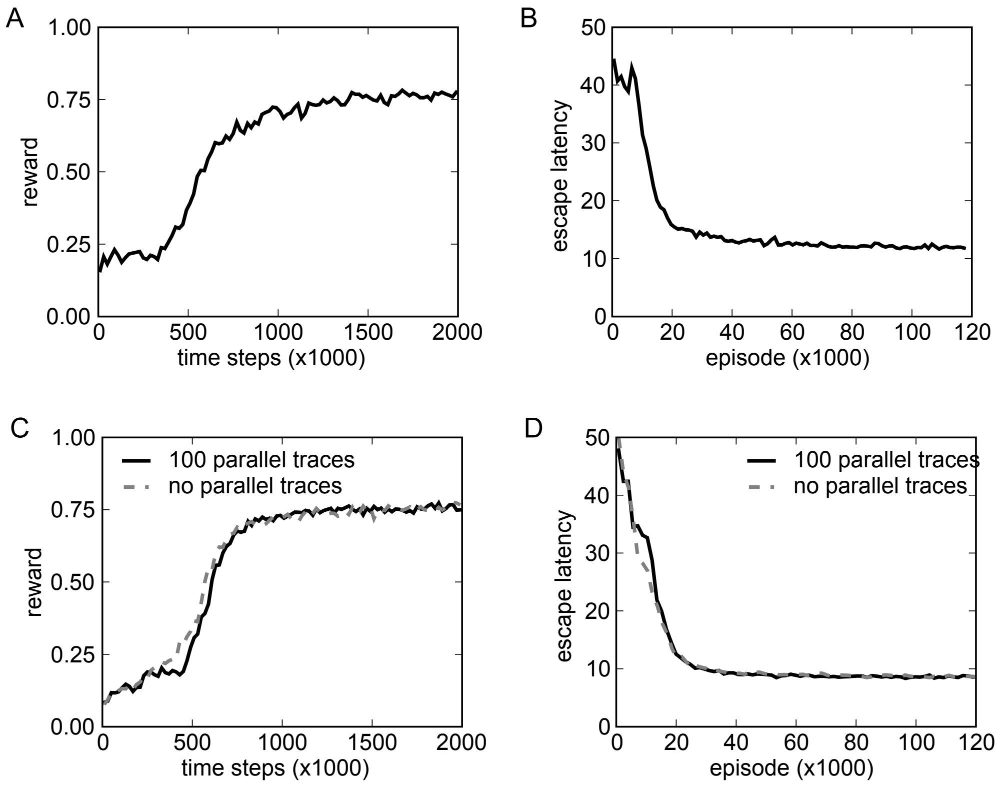


Figure 5.4: **Rewards and escape latencies during training of the control task with target and distractor.** In **A** the evolution of reward during training is shown. A simulation step for all 100 parallel traces corresponds to 100 time-steps at the x-axis. The plotted values are averages over consecutive 20,000 time steps (resulting in 100 data points for the plot). **B** shows the evolution of escape latencies (measured in time steps) during training. The number of episodes on the x-axis is the number of completed traces. The plotted values are averages over 1,200 consecutive episodes. In **C** and **D** the learning was performed on a highly condensed and precise state-encoding instead of the SFA network output. Shown is the performance for learning on 100 parallel traces (black, full line) and without parallel traces (gray, dashed line). Convergence is comparable to learning on SFA outputs. The results without parallel traces are very similar to the results with parallel traces.

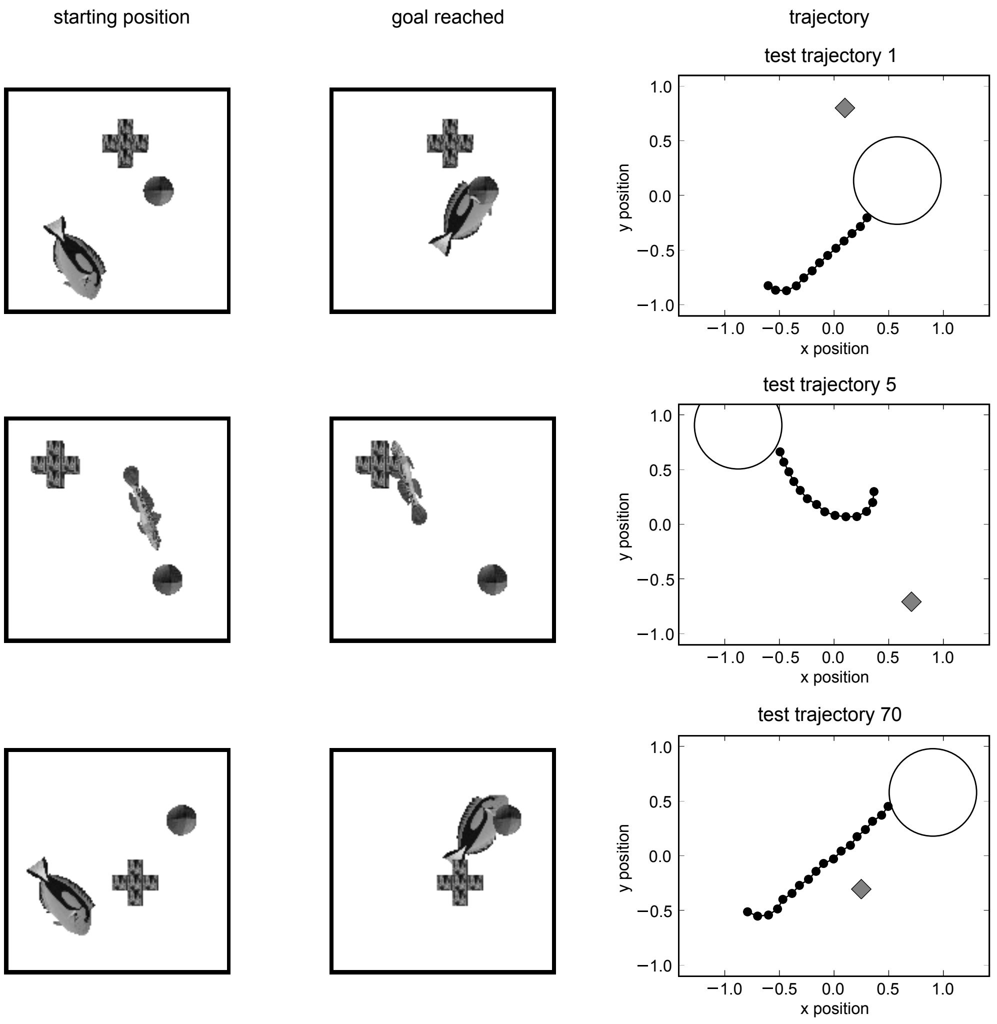


Figure 5.5: **Three representative trajectories after training of the control task with target and distractor.** Each row summarizes one representative learning trial. Shown is the visual input at start position (left column), the visual input when the goal was reached (middle column), and the whole trajectory (right column). In the trajectory, fish positions (small black discs), target region (large circle), and distractor location (gray rhomb) are shown.

For 16 outputs the average reward value always stayed below 0.6 and rose much slower than in the case of 32 outputs. For 28 outputs the performance was already very close to that of the 32 outputs. Going from 32 outputs to 64 did not change the average reward, but in the case of 64 outputs the trajectories of the agent occasionally showed some errors (e.g., the agent initially chose a wrong direction and took therefore longer to reach the target).

We compared performance of the system to a system where the control network is a two-layer feed-forward network of simpler neurons without dendritic branches, see equation (5.2). We used two networks with identical architecture, one for each control variable. Each network consisted of 50 neurons in the first layer connected to one output neuron (increasing the number of neurons in the first layer to 100 did not change the results). Every neuron in the first layer received input from all SFA outputs. The learning rates of all neurons were identical. For details on parameters and results see [Legenstein et al., 2010]. The network of simple neurons can solve the problem in principle, but it converges much slower.

We also compared performance of the system with SFA to systems where the dimensionality of the visual input was reduced by PCA. In one experiment the SFA nodes in the hierarchical network were simply replaced by PCA nodes (like in the earlier control experiment in Section 3.5.3). We then used 64 outputs from the network for the standard reinforcement learning training. As shown in Figure 5.6 the control network was hardly able to learn the control task. This is also evident in the test trajectories, which generally look erratic.

In another experiment we used PCA on the whole images. Because of the high dimensionality we first had to downsample the image data by averaging over two by two pixels (reducing the dimensionality by a factor of four) before using linear PCA. The performance was very similar to the hierarchical PCA experiment (the average reward hovered below 0.16). A direct analysis of the PCA output with linear regression [Franzius et al., 2008] indicates that except for the agent identity, no important features such as position of the agent or the targets can be extracted in a linear way from the reduced state representation. This hints at the possibility that the state representation given by PCA cannot be exploited by the control network because the implicit encoding of relevant variables is either too complex or too much important information has been discarded.

5.4 Discussion

In this chapter, we provided a proof of concept that a learning system with an unsupervised preprocessing and subsequent simple biologically realistic reward-based learning can learn quite complex control tasks on high-dimension visual input streams without the need for hand-design of a reduced state-space. We applied the proposed learning system to two control tasks. In the Morris water maze task, we showed that the system can find an optimal strategy in a number of learning episodes that is comparable to experimental results with rats [Morris et al., 1982]. The application of the learning

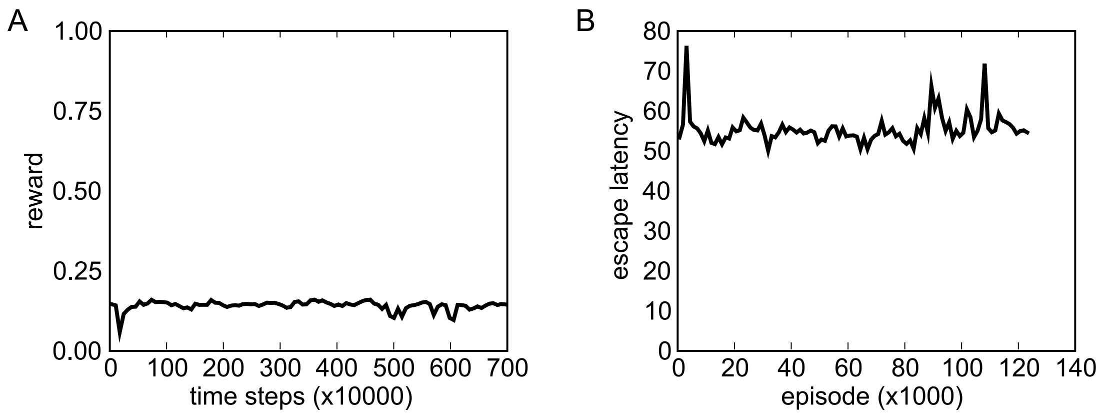


Figure 5.6: **Performance of a PCA based hierarchical network.** The rewards for the variable-targets control experiment with a PCA based hierarchical network are shown in **A** and the escape latencies in **B**. The control network is not able to learn the task based on this state representation. Note the larger scaling factor for the time-axis in **A**.

system to the variable targets task shows that also much more complex tasks with rich visual inputs can be solved by the system. We showed that learning performance of the system in this task is comparable to a system where the state-representation extracted by SFA is replaced by a highly compressed and precise hand-crafted state-space. Finally, our simulation results suggest that performance of the system is quite insensitive to the number of SFA components that is chosen for further processing by the reinforcement learning network as long as enough informative features are chosen.

In the previous chapter we used simple but not biologically motivated post-processing methods to interpret the output of the SFA network. The results in this chapter indicate that the network outputs are also adequate for biologically more plausible processing methods. This further supports that slowness learning could be one important (but not exclusive) unsupervised learning principle utilized in the brain to form efficient state representations of the environment. On the other hand, this work suggests that autonomous learning of state-representations with SFA should be further pursued in the search for autonomous learning systems that do not - or much less - have to rely on expensive tuning by human experts.

5.4.1 Related Work

The Morris water-maze task has been modeled before. In [Potjans et al., 2009], a network of spiking neurons was trained on a relatively small discrete state-space that explicitly coded the current position of the agent on a two-dimensional grid. The authors used a neural implementation of temporal difference learning. In contrast to the algorithms used in this article, their approach demands a discrete state space. This algorithm is therefore not directly applicable to the continuous state-space representation that is achieved through SFA.

In [Foster et al., 2000] and [Vasilaki et al., 2009] the input to the reinforcement learning network was explicitly coded similar to the response of hippocampal place-cells. In [Sheynikhovich et al., 2005], the state-representation was also governed by place-cell-like response that were learned from the input data. This approach was however tailored to the problem at hand, whereas we claim that SFA can be used in a much broader application domain since it is not restricted to visual input. Furthermore, in this article SFA was not only used to extract position of an agent in space but also for position of other objects, for object identity, and for orientation. We thus claim that the learning architecture presented is very general only relying on temporal continuity of important state variables.

Although the variable-targets task considered above is quite demanding, the learning system gets immediate feedback of its performance via the reward signal defined by equation (5.6). By postulating such a reward signal one has to assume that some system can evaluate that "getting closer to the target" is good. Such prior knowledge could have been acquired by earlier learning or it could be encoded genetically. An example of a learning system that probably involves such a circuitry (the critique) is the song-learning system in the songbird. In this system, it is believed that a critique can evaluate similarity between the own song and a memory copy of a tutor song [Troyer and Doupe, 2000]. However, there is no evidence that such higher-level critique is involved for example in navigational learning of rodents. Instead, it is more natural to assume that an internal reward signal is produced for example when some food-reward is delivered to the animal. One experimental setup with sparse rewards is the Morris water maze task [Morris et al., 1982] considered above. In principle, this sparse reward situation could also be learned if the learning rules are amended with eligibility traces [Xie and Seung, 2004]. However, the learning would probably take much longer.

Given the high-dimensional visual encoding of the state-space accessible to the learning system, it is practically impossible that any direct reinforcement learning approach is able to solve the variable-targets task directly on the visually-induced state-space. Additionally, in order to scale down the visual input to viable sizes, a hierarchical approach is most promising. Here, hierarchical SFA is one of the approaches that have been proven to work well. Although it is in general not guaranteed that slowly varying features are also important for the control task, slowly varying features such as object identities and positions are important in many tasks. PCA tries to preserve the variance of the training data, but the resulting features might not be useful for the behavioral task at hand.

We compared the preprocessing with SFA to PCA preprocessing but not to more elaborate techniques since the focus of this chapter is on simple techniques for which some biological evidence exists. One interesting possibility not pursued in this chapter would be to sparsify the SFA output by Independent Component Analysis (ICA). This has led to place-cell like behavior in [Franzius et al., 2007] and might be beneficial for subsequent reward-based learning. Information bottleneck optimization (IB) is another candidate learning mechanism for cortical feature extraction. However, IB is not unsupervised, it needs a relevance signal. It would be interesting to investigate whether a useful relevance signal could be constructed for example from the reward signal. Finally, the problem of state space reduction has also been considered in the reinforcement learning literature. There, the main approach is either to reduce the size of a discrete state space or to discretize a continuous state-space [Moore and Atkeson, 1995; Munos and Moore, 2002].

In contrast, SFA preserves the continuous nature of the state-space by representing it with a few highly informative continuous variables. This circumvents many problems of state-space discretization such as the question of state-space granularity. Thus, there are multiple benefits of SFA in the problem studied: It can be trained in a fully unsupervised manner (as compared to IB). By taking the temporal dimension into account, it is able to compress the state-space significantly without the need to discretize the continuous state-space (as compared to [Moore and Atkeson, 1995; Munos and Moore, 2002]). It provides a highly abstract representation that can be utilized by simple subsequent reward-based learning (compared to PCA). The possibility to apply SFA in a hierarchical fashion renders it computationally efficient even on high-dimensional input streams, both in conventional computers and in biological neural circuits where it allows for mainly local communication and thus avoids extensive connectivity [Legenstein and Maass, 2005; Chklovskii and Koulakov, 2000]. The natural ordering of features based on their slowness implies a simple criterion on the basis of which information can be discarded in each node of the hierarchical network, resulting in a significant reduction of information that has to be processed by higher-level circuits. Finally, SFA is relatively simple, its complexity is comparable to PCA and it is considerably simpler than other approaches for state-space reduction [Antoulas and Sorensen, 2001; Moore and Atkeson, 1995; Munos and Moore, 2002]. functions that generate output signals that vary as slowly as possible but

Part II

Top-Down Processes in Hierarchical SFA Models

6 Top-Down in the Visual System and Hierarchical SFA

In the previous chapters it was established that hierarchical SFA networks can be a good model for some aspects of the visual system. It was also mentioned that this model is purely feed-forward, which is a striking contradiction to the abundance of feedback connections in the visual system [Kennedy et al., 2000]. Feedback connections are thought to mediate *top-down* processes in the brain, transporting information against the flow of input data from our senses, influencing or even shaping the activity of brain areas lower in the hierarchy (e.g., V2 influencing the activity in V1). Not surprisingly our model has been unable to solve tasks that are associated with feedback, like object recognition in cluttered scenes. The importance of these tasks together with experimental evidence has severely impacted the general view of the relation between the brain and the stimuli it receives. This has led to the characterisation of the brain as a "self-sustained machine processing mainly internal information and sampling occasionally the external world" [Bullier, 2006].

Because of the advantages of hierarchical SFA it would be premature to simply discard this model with respect to top-down processes. Instead we will try to extend the model with a basic top-down mechanism. Even if the outcome is negative this will at least provide us with more insights into the limitations of the model. Unfortunately SFA networks are less well suited for top-down processes than other models, which will be discussed throughout the following chapters.

The top-down part of this thesis is split into three chapters. In the current introductory chapter we establish some basic concepts and formulate a concrete top-down task. Chapter 7 concentrates on capturing the input data distribution for each layer, with some analysis of the results for the lowest layer. This information is then applied in Chapter 8 for the actual top-down process, combined with the method of gradient descent. The results in these chapters have not been published (with the exception of a conference abstract [Wilbert and Wiskott, 2010]).

6.1 Obstacles for Top-Down Processing

A general starting point for the modeling of top-down processes is the scalar connection weight w_{ik} for one input x_i of a simple model neuron (or some other functional unit). Typically its output y_k is a monotonous nonlinear function of the weighted input h_k, defined by

$$h_k = \sum_i w_{ik} x_i \ . \tag{6.1}$$

In a classical neural network the connection weights w_{ik} are optimized for the feed-forward processing, but they can just as easily be used to modulate information flow in the other direction (which is actually done during training with error back-propagation [Bryson and Ho, 1969; Rumelhart et al., 1986]). One example for the usefulness of these connections is the famous hypothesis of feature binding via gamma oscillations [Singer and Gray, 1995; Engel et al., 2001], which also introduces an elegant mechanism for covert attention in the visual system. Its practical application in computational models has been demonstrated for relatively simple neural networks in which such a coupling w_{ik} is available (e.g., [Rao et al., 2008]).

Our model on the other hand uses a quadratic expansion and operates with positive and negative real numbers. Sums of mixed terms like $a_{ij}x_i x_j$ for each output component y_k make it impossible to characterize the coupling between a single input x_i and the output y_k in a simple way, let alone with a single scalar coupling. Generally speaking the value of one input x_i can influence the impact of all other inputs (e.g., switching the sign of x_i inverts the influence of x_j on the output value). Since the quadratic expansion is critical for the feed-forward network performance (see Section 3.5.3) there is no easy way around this issue. Instead, other methods must be found for the top-down propagation in this type of model.

6.2 Bayesian Interpretation

For a theoretical description of top-down mechanisms in hierarchical networks it is convenient to formulate a probabilistic "Bayesian" description [Lee and Mumford, 2003; Kersten et al., 2004]. Even though our feed-forward SFA network is strictly deterministic such a formulation is useful.

In a two-layered SFA network there is the input data $\mathbf{x}_0$, and the output of the first and second layer, $\mathbf{x}_1$ and $\mathbf{x}_2$. The probability distribution of the network "state" can be factorized as

$$P(\mathbf{x}_2, \mathbf{x}_1, \mathbf{x}_0) = P(\mathbf{x}_2|\mathbf{x}_1)P(\mathbf{x}_1|\mathbf{x}_0)P(\mathbf{x}_0)\ , \tag{6.2}$$

because the output of a layer is only determined by its direct input. The conditional probabilities $P(\mathbf{x}_{n+1}|\mathbf{x}_n)$ are δ-type distributions. The layer mapping $\mathbf{x}_{n+1} = f_n(\mathbf{x}_n)$ is a well-defined function, so for a given input $\mathbf{x}_n$ the probability distribution is only non-zero for a single value of $\mathbf{x}_{n+1}$, i.e.,

$$P(\mathbf{x}_1|\mathbf{x}_0) \sim \delta(\mathbf{x}_1 - f_1(\mathbf{x}_0)). \tag{6.3}$$

A top-down process could use the prediction about $\mathbf{x}_n$ based on $\mathbf{x}_{n+1}$ and $P(\mathbf{x}_n|\mathbf{x}_{n+1})$. In this context the most straight-forward example for a top-down process is the inversion problem: for a given network layer output $\mathbf{x}_{n+1}$ find the corresponding stimulus $\mathbf{x}_n$. The conditional probability distribution $P(\mathbf{x}_n|\mathbf{x}_{n+1})$ can be calculated with Bayes' formula:

$$P(\mathbf{x}_n|\mathbf{x}_{n+1}) = \frac{P(\mathbf{x}_{n+1}|\mathbf{x}_n)P(\mathbf{x}_n)}{P(\mathbf{x}_{n+1})}\ . \tag{6.4}$$

For the purpose of this argument the $P(\mathbf{x}_n)$ distribution is defined by the probabilities of input patterns in the training data. This distribution cannot be deduced from the trained SFA network alone. Furthermore, the layer function $\mathbf{x}_{n+1} = f_n(\mathbf{x}_n)$ is generally not injective. Therefore $P(\mathbf{x}_n|\mathbf{x}_{n+1})$ can be non-zero for multiple values of $\mathbf{x}_n$ (for a given $\mathbf{x}_{n+1}$). Accordingly we are now faced with a non-trivial probability distribution, even though the starting point was a strictly deterministic network. Because of the dimensionality reduction in our network we can even expect that there are manifolds of inputs that all result in the same wanted output (implicit function theorem). However, because of the constraints in the SFA algorithm (unit variance and decorrelation) the layer functions are typically injective on the set of training samples.

For example, a fish-trained SFA network layer will likely produce an output that is identical to that of a fish for special non-fishy inputs. Being able to "invert" the mapping functions of the SFA layers (which is already a nontrivial problem) is therefore not sufficient to reconstruct likely input images for a given output. The probability distribution $P(\mathbf{x}_n)$ is required to disambiguate the solutions and to pick the most likely candidate.

The goal is of course to solve the inversion problem not only for a single layer but for the whole multi-layered network. What we are ultimately interested in is the input $\mathbf{x}_0$ in image space.

6.3 Input Reconstruction

The previous section introduced the problem of reconstructing the most likely input for a given output, taking into account the distribution of training samples. This is not only an interesting problem in itself, it might also bring us closer to the couplings between input and output from Section 6.1. With a successful input reconstruction for some given output one could compare this to another given input and quantify the deviation. We therefore think that input reconstruction is an interesting task and the efforts in the following two chapter are centered around this.

A successful input reconstruction could also help with a related problem in hierarchical SFA networks: the interpretation of intermediate and top-layer outputs. The SFA outputs in the first layer of our model have been extensively characterized in [Berkes and Wiskott, 2005; Sprekeler and Wiskott, 2010]. For the top-layer outputs there are analytical predictions (see Section 3.3.4), which are accurate if the stimuli are not too complex (Section 3.4 and [Wiskott, 2003; Franzius et al., 2007]). However, they do not reveal which specific features of the input image drive the output (e.g., how relevant the relative positions of local input features are). The situation is worst for outputs in the intermediate layers: neither their input nor output characteristics are well understood (input here referring to the input in image space). This problem of interpreting the trained network is relevant for all deep learning architectures, and recently there has been some progress on this front [Erhan et al., 2010] (other related work is discussed later in Section 8.3.1). Input reconstruction could provide a tool to investigate the relation between input and output. This could also lead to new experimental predictions and further hints about the function of intermediate regions in the visual cortex (like V2).

6.3.1 Related Work

The problem of input reconstruction for hierarchical SFA networks has been addressed before by William Softky (documented in the unpublished [Softky, 2005]). The approach he describes is quite close to the techniques we ended up using. His suggestion was to use some form of vector quantization in order to capture the distribution $P(\mathbf{x}_0)$ of the input data. A local linearization of the SFA network around these points could then be used to modify the input according to given changes in the output. For example, to recreate movement for training stimuli. His suggestions were aimed at the smaller 1D hierarchical SFA model from [Wiskott and Sejnowski, 2002]. Unfortunately his work was in an early stage and no further results are known. Given the difficulties that we encountered (see Section 8.3) it would not be surprising if his implementation ran into similar problems.

A truly probabilistic version of SFA has been presented in [Turner and Sahani, 2007], reformulating slowness learning in the framework of a generative model. However, the resulting model is rather complex and training is slow compared to SFA. Therefore it is not applicable to visual stimuli as we use them (in the publication it is only demonstrated for low-dimensional synthetic stimuli). Overall the paper is focused on theoretical aspects, while the practical application is not really discussed.

We should also point out that our take on top-down processes is not related to the feedback implementation in recurrent neural networks (e.g., see [Rojas, 1996]). These networks are used for data sequences where the interpretation of one input is improved by taking the previous input into account (e.g., for a Markov chain or recognizing successive characters in handwritten text [Graves et al., 2008]). The feedback connections basically provide a short-term memory. Our model on the other hand uses time sequences only during the training phase, but not for the actual object recognition or input reconstruction.

7 Capturing the Input Distribution

7.1 Introduction

In the previous chapter it was argued that implementing top-down processes in a hierarchical SFA network requires that the distribution of input data $P(\mathbf{x}_n)$ is captured for each layer, in order to approximate $P(\mathbf{x}_n|\mathbf{x}_{n+1})$. We pursued a rather simple approach for that, based on different forms of vector quantisation. The specific methods are introduced in Section 7.3. In some cases the results can nicely reveal the underlying structure of the data, which is illustrated in Section 7.4. However, the main objective is to gather data for the top-down input reconstruction, which is discussed in Chapter 8.

7.2 Methods

The methods for top-down processes that we used are quite expensive in terms of computation time. Therefore a smaller three-layered SFA network was used, as well as simplified input stimuli. The vector quantization methods in the model are described in their own Section 7.3.

7.2.1 Network Structure

The new network is very similar to that used for the fish and sphere stimuli in Chapter 3, with the reduced size being the biggest change. All the basic network parameters are provided in Table 7.1.

Instead of the cutoff node with a fixed cutoff (at ± 4) an adaptive cutoff node was used in some simulations, and otherwise the cutoff was completely dropped. It had already been established that the cutoff for the outputs in each layer does not have a significant impact on the network performance. A cutoff also means that the gradient is zero in the affected region, which is problematic for gradient based optimization techniques (like the one used in the next chapter). Therefore the cutoff was only used in simulations together with a vector quantization technique that requires fixed cutoff values (the binned growing neural gas which is presented in Section 7.3.3). The adaptive cutoff node records the histogram of the SFA outputs during training. Afterwards the cutoff values are set for each quadratic SFA output individually, such that a given fraction of the data (in this case 1%) lies above / below the threshold.

Furthermore, no noise node was used in this model. As mentioned before, the noise node does not have a significant impact on network performance and mostly serves as a safeguard against numerical problems with singular eigenvalues. Since these problems

Layer	Number of nodes	Input area of node	Overlap per direction	SFA outputs per node
0 (Image)	80×80	-	-	(1 pixel)
1	13×13	8×8	2	32
2	4×4	4×4	1	32
3	1	4×4	-	200

Table 7.1: **Parameters of the network architecture.** Layer 0 denotes the input image, a node corresponds to a pixel in that image. The input area denotes the number of nodes in the previous layer from which a node receives input.

	max. vel.	max. acc.
x (in $[0;1]$)	0.03	0.007
y (in $[0;1]$)	0.03	0.007
size (in $[0.75;1]$)	0.01	0.002
ϕ_z (in-plane)	0.02	0.004
α (in $[0;1]$)	0.03	0.007

Table 7.2: **Parameters for the letter random walk procedure.** These parameters refer to the same procedure as those in Table 3.2.

only occurred in test runs with minimal training data the noise node was dropped to simplify the model.

7.2.2 Stimuli

Corresponding to the smaller network the input stimuli were also reduced to a size of 80×80 pixels. Two new classes of stimuli were introduced, which are described below. For the first two network layers 20,000 training samples were provided (in chunks of 200), while the top-layer was trained with 200,000 samples (in chunks of length 500). The same random-walk parameters were used for all layers.

Single Letters

These stimuli consist of a single white letter on a black background and are shown in Figure 7.1. They are somewhat similar in style to the black and white leaf stimuli from Section 4.1, so we already knew that the SFA network can deal with silhouette stimuli. The transformations consist of translations, in-plane rotations, scalings, and changes in contrast (as described in Section 4.2 for multiple fish objects). All the random walk parameters are specified in Table 7.2. Five different letters were shown during training (A, B, C, D, and E). One motivation for these stimuli is that their simple structure makes it easier for humans to interpret the results of top-down processes.

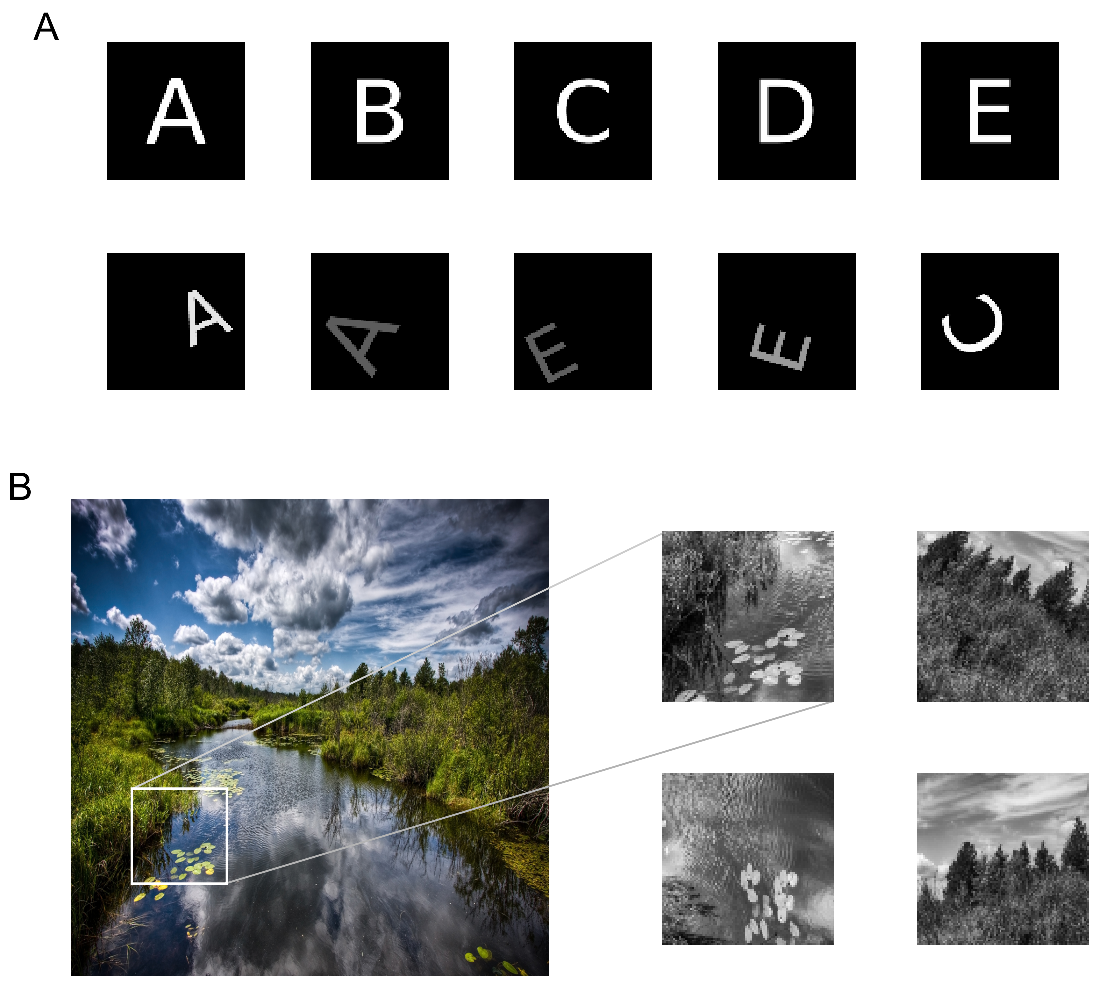


Figure 7.1: **Letter and natural stimuli.** In **A** we show the single letter stimuli. The top row shows the five different training letters (i.e., “objects”). The second row shows more examples from the actual training sequence, which illustrate the translation-, rotation-, size-, and alpha-transformations. **B** shows the natural stimuli, which are crops from a large image of a natural scene. The actual stimulus pictures on the right illustrate the translation- and rotation-transformations of the crop window.

Natural Image with Transformations

These stimuli closely resemble the ones used in [Berkes and Wiskott, 2005]. They are intended for control experiments, to complement the highly artificial letter stimuli. The stimuli are created by taking a quadratic excerpt from a large natural grayscale picture, as shown in Figure 7.1. The position, orientation and size of the excerpt is subject to the same random walk as used for the fish and letter stimuli (i.e., instead of an object the excerpt window is moved around). The parameters were chosen such that the movement looks reasonably smooth. Only a single underlying picture of size 1024×1024 was used, but control experiments with other pictures were done to verify that the results do not depend on this specific one.

7.3 Vector Quantization

Vector Quantization (VQ) techniques generally approximate data with a limited number n of representatives $\mathbf{w}_i$, minimizing an error measure on a set of training data $\{\mathbf{x}_j \,|\, j = 1, \ldots, m\}$ [Gray, 1984]. Typically the error measure is given by the average distance from the data points to their nearest representative. The standard choice for the distance measure is the squared Euclidean distance, in which case the average error is given by

$$E = \sum_{j=1}^{m} \min\{(\mathbf{w}_i - \mathbf{x}_j)^2 \mid i = 1, \ldots, n\}. \tag{7.1}$$

The representatives $\mathbf{w}_i$ are often called *centroids*, because they typically lie at the center of the data cluster that they represent. Typical vector quantization algorithms are iterative, approximating the optimal solution. Famous examples are k-means clustering [Lloyd, 1982] and the derived LBG algorithm [Linde et al., 1980]. Alternatively one can apply competitive learning techniques, which is described in Section 7.3.3. Related techniques in machine learning and computational neuroscience are for example Self-Organizing Maps [Kohonen, 1984] or Radial Basis Function Networks [Moody and Darken, 1989]. There are numerous technical applications for vector quantization in the area of lossy data compression (e.g., for video or speech), in which the centroids are used as a codeword dictionary.

Vector quantization can be considered a standard option for approximating data distributions. Due to the special characteristics of our data distribution (see Section 7.3.2) we tested different algorithms.

7.3.1 Integration of VQ into the SFA network

The integration of vector quantization into the SFA network is schematically shown for a single node in a layer in Figure 7.2. Only a single instance of the VQ unit is used for each layer and this instance is used for all the input fields in that layer. This was already described in Section 3.2 for SFA.

In Figure 7.2A the vector quantization is performed on the raw input data. This

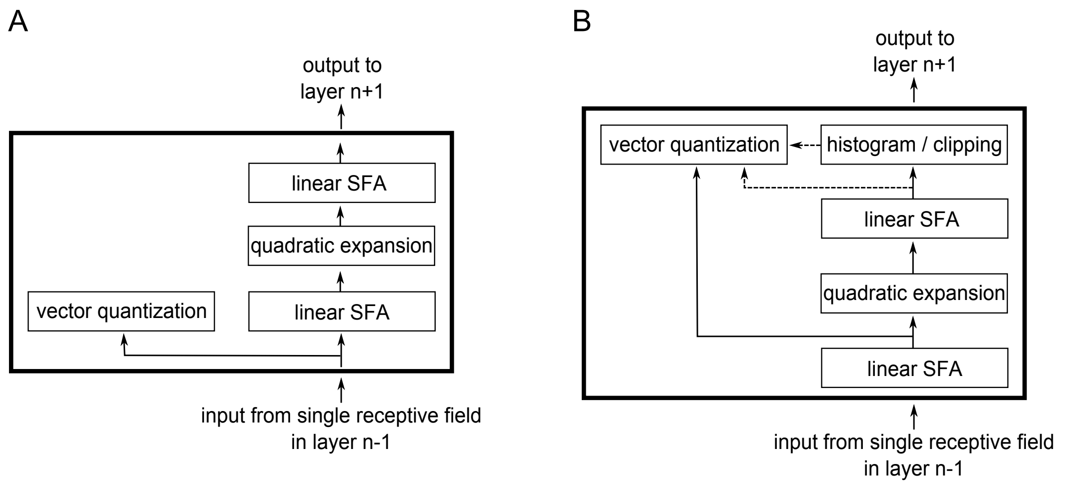


Figure 7.2: **Insertion of vector quantization into the model.** These diagrams depict the processing steps for a single node in a layer (see Figure 3.1). In **A** the vector quantization is performed for the raw input. In **B** the output of the first linear SFA step is used instead. Depending on the algorithm other data might be used as well, indicated by the dotted lines (e.g., the binned growing neural gas quantization requires the cutoff values and the output values for each data point).

variant is mostly used in the next chapter. In Figure 7.2B the output of the first linear SFA stage is used for the vector quantization. This is done for efficiency reasons, taking advantage of the reduced dimensionality. By taking the pseudoinverse[1] of the linear SFA step it is then still possible to map the centroids back into the original input space (e.g., for visualization). Since the linear SFA step on layer 1 behaves very similar to Principal Component Analysis (PCA) the dimensionality reduction on the lowest layer acts like a low-pass filtering. This information loss in the dimensionality reduction is in conflict with the stated goal of top-down processing, so this could negate the efficiency gain. However, this approach (quantization after linear SFA) is typically better suited for illustrating the structure of the data distribution. In this chapter we therefore concentrate on those results. Most of the simulations were performed in both ways and we discuss their differences along the way.

Some of the vector quantization algorithms (like the binned growing neural gas) require additional data during training, for example, the cutoff values for each output. This is illustrated in Figure 7.2B by the dotted lines. The implementation of these data flows was done in the BiMDP framework, which is discussed in Chapter 11.

One potential complication in our model is the technical fact that the training data originated from a continuous random walk. Therefore data points that are close by in the training sequence also tend to be close in the data space, but many VQ algorithms

[1]The pseudoinverse of a matrix A is defined as the matrix that solves the least-squares problem $A\mathbf{x} = \mathbf{b}$ [Strang, 1988]. It can be calculated via a singular value decomposition of A.

expect random samples from the data distribution. Therefore we shuffle all the data points in each training chunk before the VQ training. Due to the limited length of the training chunks this method is not perfect, but seems to be sufficient.

7.3.2 Distribution of Layer Output Data

By capturing the input data distribution for each layer in the hierarchical SFA model we hope to get an approximation of $P(\mathbf{x}_n)$. If the layer input distribution for $\mathbf{x}_n$ is known then the layer output distribution $P(\mathbf{x}_{n+1})$ can be calculated by simply applying the feed-forward layer function. However, the expected complexity of the input distribution suggests that one should optimize the VQ technique with respect to the top-down step as well. Some insight into the distribution can be gained by looking at the histograms of the layer output, some of which are shown in Figure 7.3. These histograms are largely independent of the specific training stimuli. While the histograms shown are based on the artificial letter stimuli (Section 7.2.2), their general shape also holds for both natural stimuli and the fish / sphere objects used earlier. The distribution is densely packed around zero, with high kurtosis (i.e., being very “peaky”). For the lowest layer this result has already been discussed in [Berkes and Wiskott, 2005]. It is a little surprising that we get the same pattern on the top layer, while both the theoretical predictions and the results from Section 3.5.1 suggest a more structured output distribution at the top (i.e., a somewhat homogenous distribution for position encoding outputs, while the object identity outputs should show clusters). However, the simplified example from Section 3.5.1 is our only case in which the top-layer histograms actually display such a clear structure. For any of the more complex training sets the histograms revert back to the peaky distribution around zero. This is apparently caused by the imperfect invariance of the network and the related feature mixing (which was discussed in Section 3.3.4).

It is worth pointing out that the peaky output histograms are “sparse” in the sense of [Olshausen and Field, 1996]. For example, a single output with complex-cell behavior in the lowest layer will rarely be faced with a high-contrast grating, causing a large output value from the long tail of the distribution. Such rare events with large absolute output are naturally the most interesting. This can be formalized with the information theoretical concept of *self-information* (which is also called the *surprisal*). In the context of top-down processes such a large (i.e., unlikely) output value is especially relevant for the deduction of possible input values.

In hierarchical models the output of one layer provides the input for the next one. Therefore the distribution of input values at the higher layers have the same peaky shape. For the raw stimuli at the first layer this is not necessarily true (Figure 7.3B), but if the vector quantisation is performed after the linear SFA step the relevant data on the lowest level is already “sparsified” to some extent (Figure 7.3C).

The data histograms are relevant at this point because standard VQ algorithms match the centroid density to that of the training data. This makes sense as it minimizes the average quantization error. In our case this could potentially be a disadvantage, because centroids might be “wasted” on the dense but relatively uninteresting center region of the data distribution. By using specialized VQ algorithms this can be avoided, but as

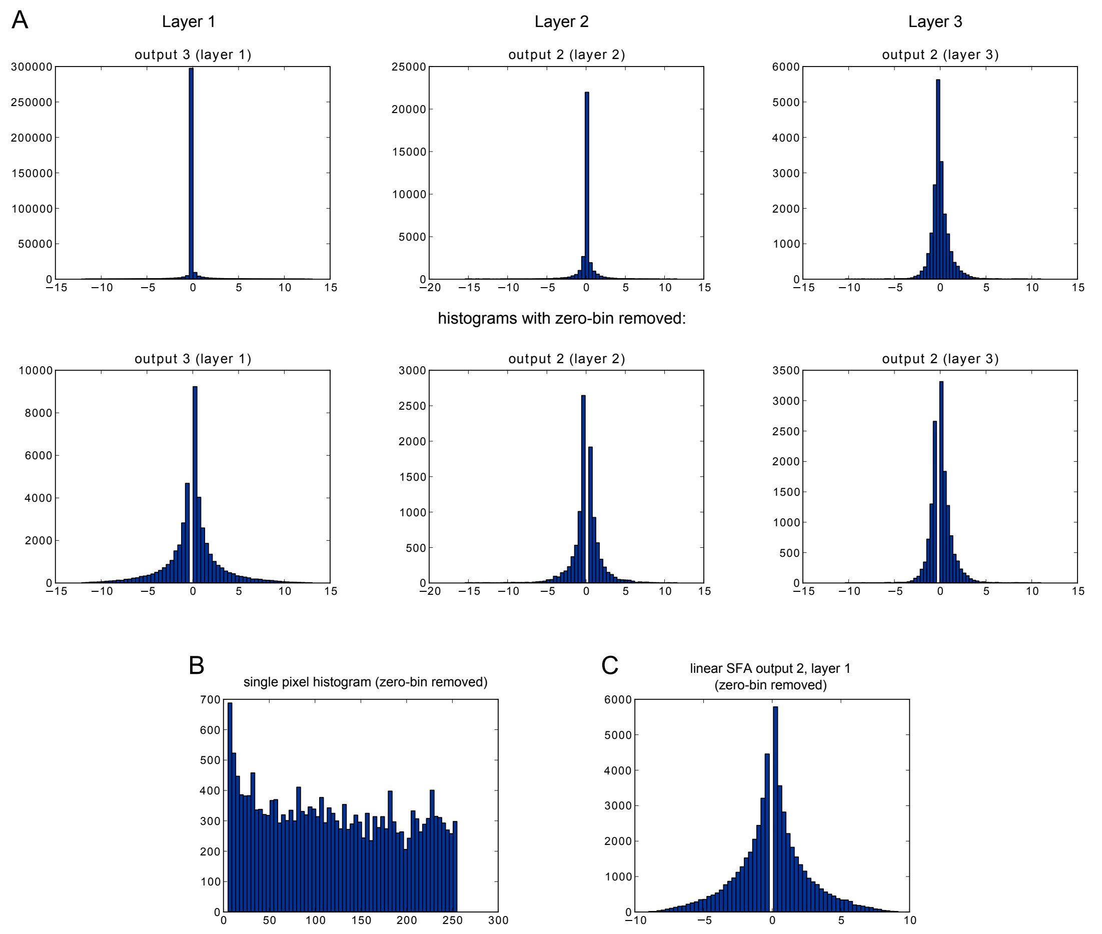


Figure 7.3: **Histograms for SFA outputs.** The histograms where calculated from the letter training data of the network. **A** shows the histogram for one representative output from each layer. Since the bin for the value 0 is so dominant the plots are also shown with this bin removed (second row). In **B** the histogram for a single pixel is shown, with the removed bin corresponding to the black background color (i.e., the value 0). In **C** a single output from the linear SFA step on the first layer is shown (again with the most populated center bin removed). Its distribution is similar to the results in **A** and much sparser than the raw data in **B**.

we will see later this is actually not necessary.

7.3.3 Vector Quantization Methods

We now discuss two different vector quantization techniques, for which the results are presented in Section 7.4.

Competitive Learning

Competitive learning [Grossberg, 1976; Rumelhart and McClelland, 1986] provides a simple but effective algorithm for vector quantization. First the n centroids are initialized by randomly taking n samples from the training data. Then the centroids are refined iteratively with these steps:

1. Pick a random data point $\mathbf{x}$ from the training data.
2. Find the closest centroid $\mathbf{w}_i$ (based on the Euclidean distance).
3. Move the closest centroid towards the data point by updating its value to $\mathbf{w}'_i$, which is defined by
$$\mathbf{w}'_i = \mathbf{w}_i + \varepsilon(\mathbf{x} - \mathbf{w}_i). \tag{7.2}$$

To simplify the implementation we use each training point exactly once (by processing a queue containing the shuffled training data, the queue is only used once). 1000 centroids were used in each layer, with $\varepsilon = 0.1$.

The VQ step represents a significant part of the overall network training time. One standard method for speeding up vector quantization is to use a so called kd-tree for finding the nearest centroid. Unfortunately this is only effective in k dimensions if the number of data points is much bigger than 2^k, so in our case the dimensionality is already too high. Unlike SFA it is also not possible to parallelize this algorithm, because each centroid movement affects the distances for all the following data points (though there are VQ algorithms that are designed for parallelization). More sophisticated algorithms are in principle available (e.g., [Freund et al., 2007]), but were not tested by us.

Binned Growing Neural Gas

As discussed in Section 7.3.2 we were originally worried about the concentration of centroids around the origin and the corresponding under-representation of data in the periphery. To counter this effect we tested an approach in which the input distribution is captured separately for each output component. Furthermore, we split the range between the negative and positive cutoff evenly into m bins. For each output-bin the vector quantization of the input data is performed separately, always using the same number of centroids (regardless of the number of data points for each bin). This ensures that the more frequent bins around zero do not get more centroids than the more interesting bins at the far ends. This is illustrated by the results in Section 7.4.

ϵ_b	0.2
ϵ_n	0.006
a_{max}	10
λ	10
α	0.5
d	0.995

Table 7.3: **Growing Neural Gas parameters.** The parameter table follows the naming convention from [Fritzke, 1995].

For each bin a Growing Neural Gas (GNG) [Fritzke, 1995] was trained with the corresponding input data. The GNG algorithm not only learns the centroid positions, it also creates a graph structure with edges that connect two centroids. Such an edge connection indicates that the two centroids were simultaneously close to some of the training data points. As before, competitive learning is used to move the centroids. Like in a self-organizing map, the graph neighbors of the closest centroid are moved towards the data point as well. The GNG algorithm starts with only two centroids and adds centroids until a predefined limit is reached. New centroids are only inserted on graph edges, hopefully avoiding empty regions between data clusters. Eventually, the graph structure represents the structure of the training data, with graph edges indicating a continuous distribution of data between two centroids. Of course, this depends on the number of centroids being large enough to represent this structure. For details on the GNG algorithm see [Fritzke, 1995].

One advantage of having the graph structure is that it is preserved when the centroids are projected into a 2D subspace for visualisation. The 2D distance between centroids after the projection might be misleading, but the graph edges still represent the topology from the original data space.

We set the maximum number of centroids to 30, the number of bins to 8, and there were 32 outputs from the SFA stage at the lowest layer. The overall number of centroids on the lowest layer is therefore $30 \times 8 \times 32 = 7200$. All the parameter values for the GNG algorithm are provided in Table 7.3. Our implementation of the algorithm in Python is a speed-optimized version of the implementation in MDP (see Part III).

7.4 Results

7.4.1 Network performance

The performance of the simplified network on the letter stimuli is generally worse than that of the larger network in Chapter 3 for the fish and sphere stimuli. For example the average classification performance on the 5 training letters is 92.3%. The most misclassifications happen between the letters B and E, which can be explained with their similar shape.

The overall behavior of the network, however, does not indicate any qualitative differences from the larger network in Chapter 3, so it should be "good enough" for our purposes. If letter stimuli without contrast transformations are used for training then the classification performance rises to 99.2%. Because of the earlier results in Section 4.2 we nevertheless used the network with contrast transformations, in the hope that the added robustness might be an advantage for top-down processes. Control experiments did not show any qualitative difference between the networks (see Section 8.2.3).

For the natural stimulus set no performance measure was recorded, and the analysis was concentrated on the lowest network layer. While the network could theoretically learn to extract the configuration parameters of the excerpt window (in close analogy to the results in [Franzius et al., 2007]) this was not part of the analysis.

7.4.2 Binned Growing Neural Gas

We start with the results for the binned GNG quantization, since these visualisations tend to be the clearest. The GNGs can be used to highlight the different characteristics of layer 1 outputs, some of which behave similar to neurons in the primary visual cortex. This has been discussed extensively in [Berkes and Wiskott, 2005] and [Sprekeler and Wiskott, 2010].

Figure 7.4 visualizes the GNGs for an output with approximate complex-cell behavior. Such an output shows the strongest response (i.e., largest output value) for a stimulus with a grating structure. The output is sensitive to the orientation of the grating ("orientation tuning"), but is invariant with respect to the phase of the grating. When the grating is rotated away from the preferred orientation the output value drops, passes zero, and finally takes the maximal negative value for the orthogonal orientation (which can be compared to maximal inhibition in neurons). However, since the output sign in the SFA algorithm is arbitrary one can also interpret the maximal negative stimulus as the preferred orientation and vice versa. In Figure 7.4D the stimulus with maximal negative output is shown, calculated with the method from [Berkes and Wiskott, 2006].

Figure 7.4A shows the GNGs for the four bins with negative and positive output values in two separate plots. The graph edges are color coded from blue (for the most negative bin) to red (most positive bin). The corresponding output values of all the centroids are shown in Figure 7.4B. Principal Component Analysis (PCA) is used to reduce the dimensionality of the centroids from the 32 dimensional input space down to two dimensions for the plots. Obviously the GNGs have a concentric ring or C shape. The radii of the rings correspond to the contrast and orientation of the input, while the angular position on a ring is given by the phase of the grating (which has no influence on the output value). In the 32 dimensional input space the negative (blue) GNG centroids approximately lie in a plane that is orthogonal to that of the positive GNGs, corresponding to the orthogonal orientations of the optimal grating stimuli.

The open C shape of the GNGs can be understood by looking at a single bin in Figure 7.4C. In addition to the GNG several selected centroids are shown on the right side. As mentioned earlier in Section 7.3.2 the pseudoinverse of the linear SFA step is used to project the centroid vectors back into image space (or more precisely the image patch for a single receptive field). The five centroid samples show a single white bar

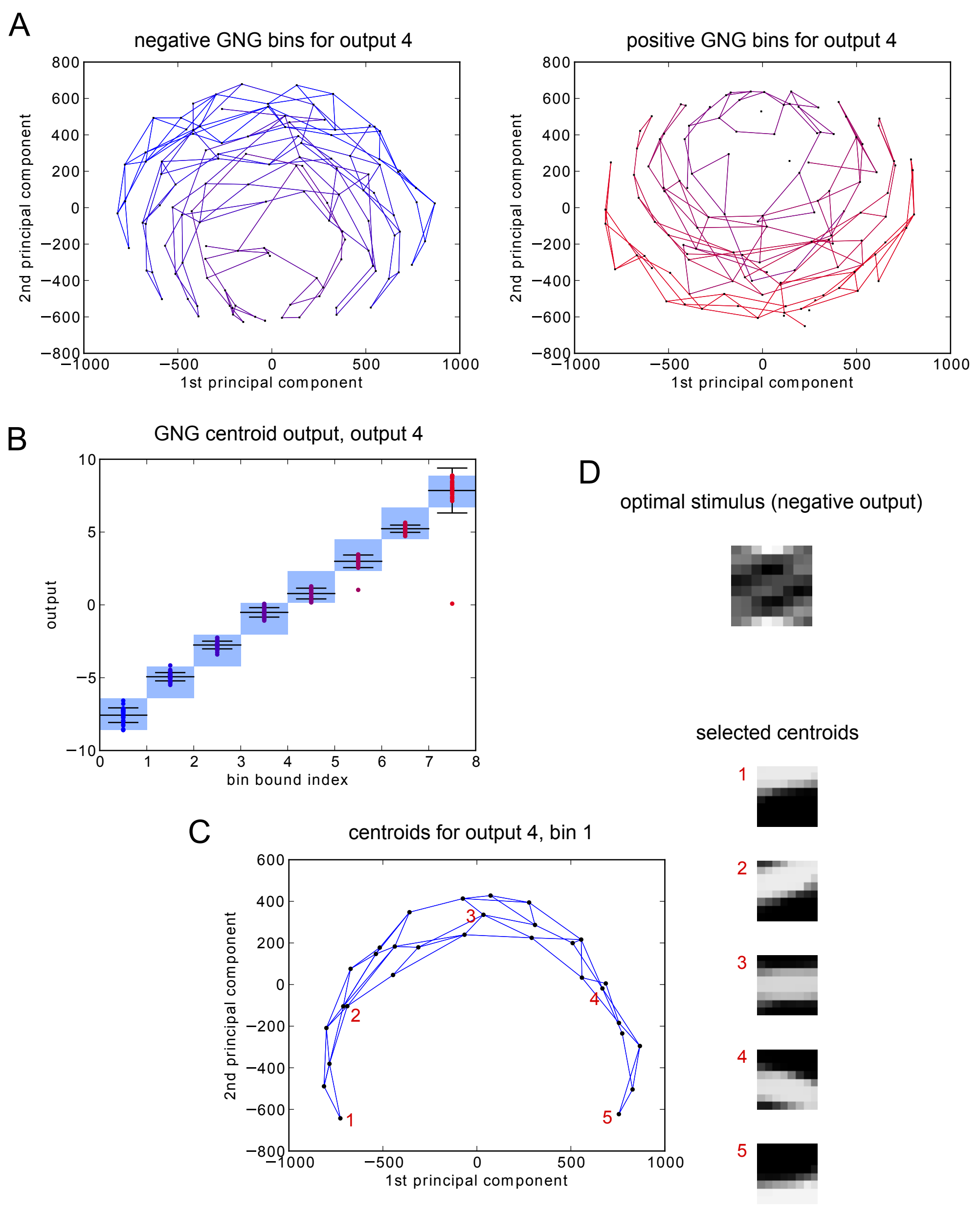


Figure 7.4: **GNG centroid data for a complex-cell type output.** All the data in this figure belong to output 4. In **A** the GNGs for positive and negative output bins are shown separately. They are color-coded corresponding to their bin (from blue to red). **B** shows the output values for each centroid, sorted into bins (black lines indicating the mean and variance for each bin). **C** shows the GNG for the first bin. The corresponding optimal negative stimulus is presented in **D**. On the right side several centroids from the GNG are shown, with their position in the graph indicated by numbers.

that moves through the receptive field. This is clearly different from the continuous grating in the theoretically optimal stimulus, highlighting that the GNG really contains information about the training data. The single white bar also has to vanish on one side before it can reappear on the other side of the receptive field, since there are no parallel bars with the right distance in the training images. This explains why the GNG has a C shape instead of a full ring: there are no data points in the training set for the lower ring section and no corresponding centroids are created.

A GNG for another output is visualized in Figure 7.5. This output shows so called secondary response lobes. This is similar to the complex-cell behavior, but instead of a single preferred orientation there are now two. The preferred orientations are orthogonal, while the maximal negative response is achieved by gratings that are rotated by 45°. Not surprisingly the 2D projection of the GNGs shows two cross structures which also have a relative angle of 45°. The single bin GNG in Figure 7.5B consists of two disconnected parts, each corresponding to one preferred orientation. Each part has the now familiar C shape, but since the second C is seen from “above” it appears as a single line in this plot. In the centroid visualization on the right side the two overlapping ends of the second C correspond to the centroid numbers 3 and 5.

The results presented so far nicely match our expectations, but this is not true for all outputs. As shown by the example in Figure 7.6 there are outputs for which the output values generated by the centroids do not match the output range of the bin that they correspond to. This is a sign that the input data distribution was not captured in enough detail, so the centroids are too far away from the data points that they represent. In principle this can be solved by increasing the number of centroids, but in practice we were not able to increase the number of centroids sufficiently. The problem persisted up to 80 centroids per bin, which was the maximum number we tested. Reducing the number of bins is another possible approach, but this partly counteracts the original intention for the bins.

Originally we had intended to use the binned GNGs for a specific top-down method that would pick the GNG bins according to the desired output value. The results were disappointing, in line with the problem discussed in the previous paragraph. Therefore we abandoned this approach and the binned GNGs are not used in the following chapter.

7.4.3 Competitive Learning Vector Quantization

The results from the competitive VQ are generally not as visually pleasing as the GNG results, probably due to the missing graph structure. However, the structures that are visible in the GNG results can still be identified. It also seems that the centroids provide sufficient coverage for the outer regions of the data distribution.

In Figure 7.7 the centroid analysis for output 3 is shown, which has the characteristics of a complex-cell. In Figure 7.7B one can roughly identify the orthogonal directions of the two optimal grating orientations. After the separation into centroids with positive and negative outputs the characteristic C shape can be found again (though the missing part of the ring is not as prominent as in the GNG results). In Figure 7.7D only the centroids with the smallest (i.e., negative) output values are shown. The C shape of the

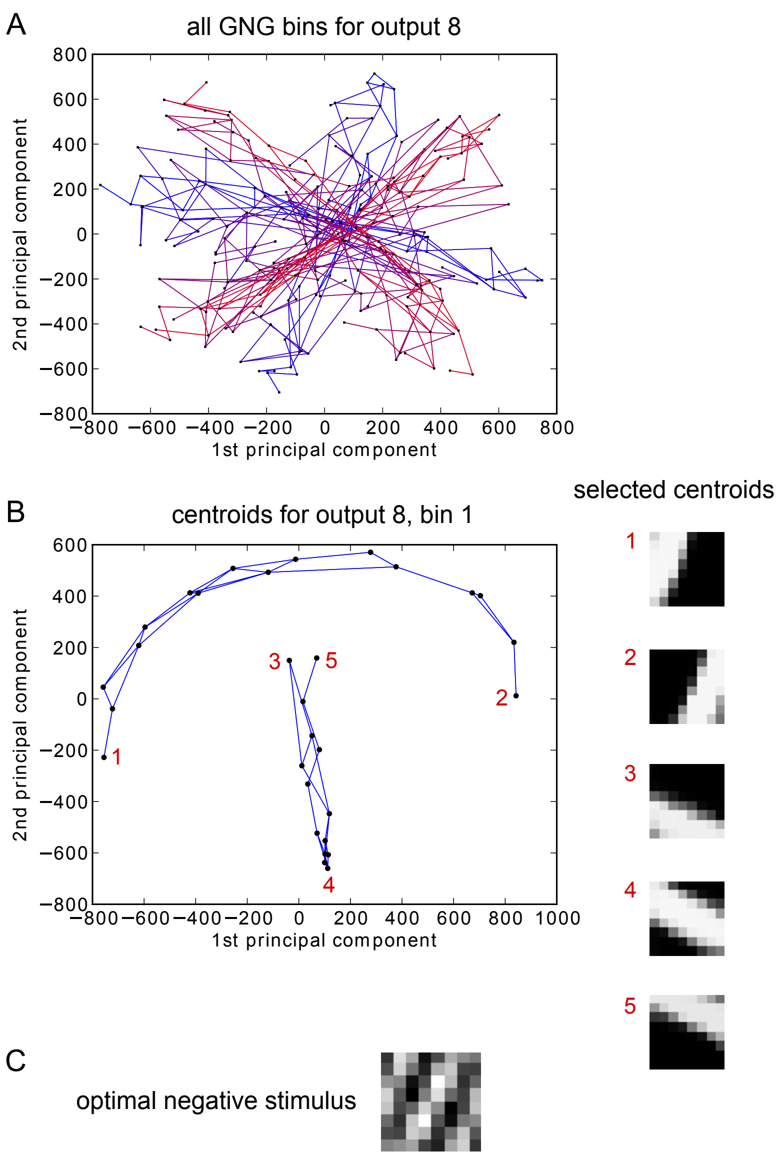


Figure 7.5: **GNGs for an output with secondary response lobes.** The data in this figure belongs to output 8. **A** shows the GNGs for all bins (color coded from blue to red according to the bin). One can see the blue and red cross structure, with each axis of the crosses corresponding to one optimal orientation. In **B** the GNG for the first bin is shown. The two disconnected parts correspond to the two preferred orientations. This is further illustrated by selected centroids on the right side.

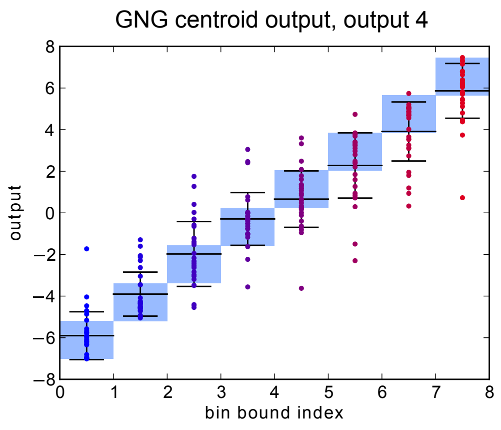


Figure 7.6: **Output values for the centroids in all bins for output 4.** The colored dots indicate the output value for each centroid, sorted into bins (black lines indicating the mean and variance for each bin). Ideally the centroid output values should lie in the same range as the output values for the corresponding data points (defined by the bin borders). This is clearly not the case, indicating that the centroids do not represent the data points adequately in this context.

distribution is not very clear, but can still be identified. The image patches for selected centroids illustrate that this is indeed the expected result for a complex-cell output.

Figure 7.8 again shows some centroids, but this time for a network trained with the natural stimuli. The plot contains only the centroids with the largest output values for output 13, which is an output with secondary response lobes. Unfortunately the structure of the centroids is even less apparent than for the network trained with letter stimuli. However, one can still identify the two orthogonal orientations in the centroid distribution.

7.4.4 Control Experiments

- Control experiments with larger networks were performed for the methods that are introduced in this and the following chapter. These experiments included networks of intermediate size and a training of the smaller networks with fish or sphere stimuli.
- Different parameter values for the GNGs were tested. We picked the final values based on visual inspection. Different numbers of centroids were tested as well. An increase in the number of centroids beyond our chosen value did not result in any significant improvement.
- While we focused on the letter stimuli, the results for the natural stimuli largely verified our results. Due to the more diverse stimulus structure some of the results are less clear than for the letters. However, the output characteristics are also

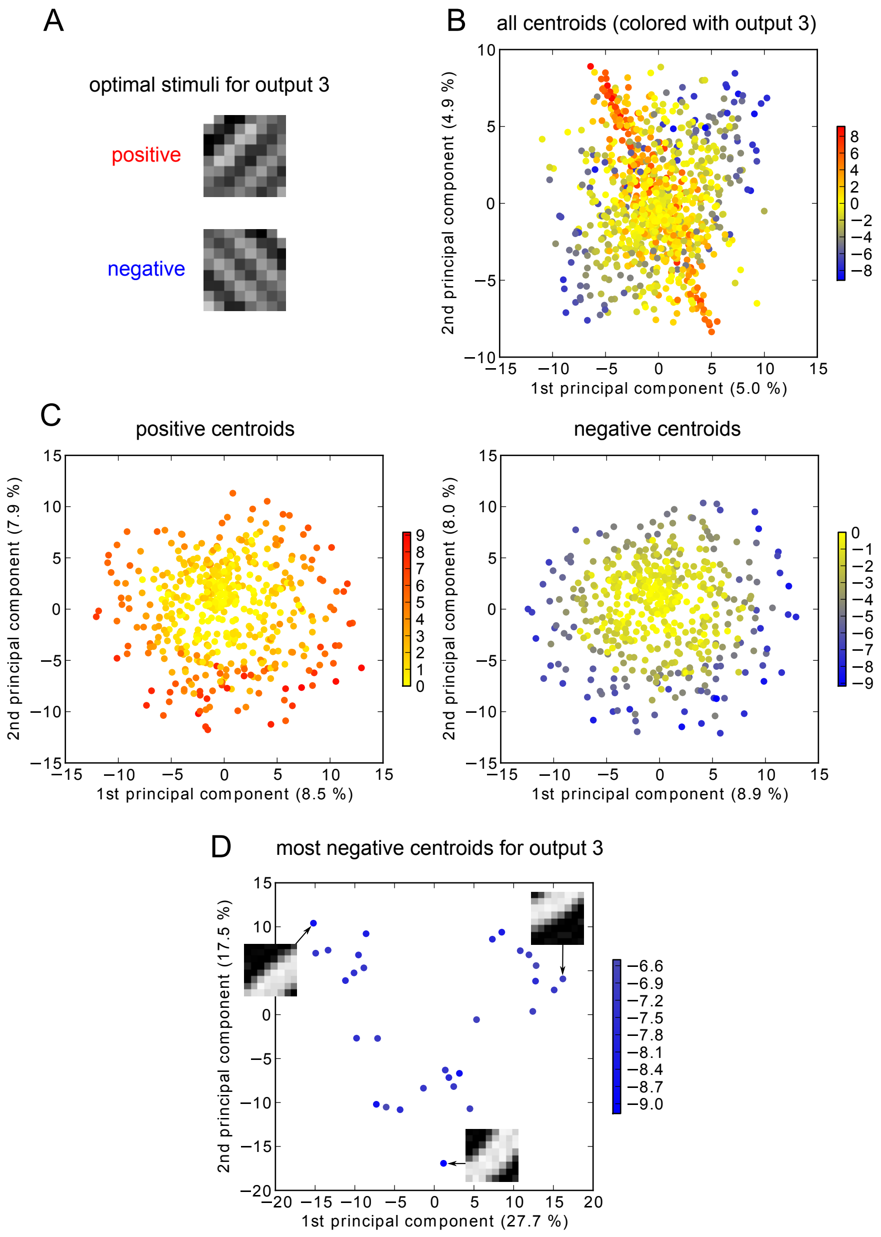


Figure 7.7: **Competitive learning centroids for complex-cell output.** All the data in this figure corresponds to output 3, for which the optimal stimuli are shown in **A**. In **B** all 1000 centroids are shown, after a PCA projection (the percentage numbers along the axes tell the explained variance) and colored according to their output value. **C** shows the same centroids, but split into those with positive and negative output. In **D** only the centroids with the most negative output values are shown, together with the pseudoinverse reconstruction for three selected centroids.

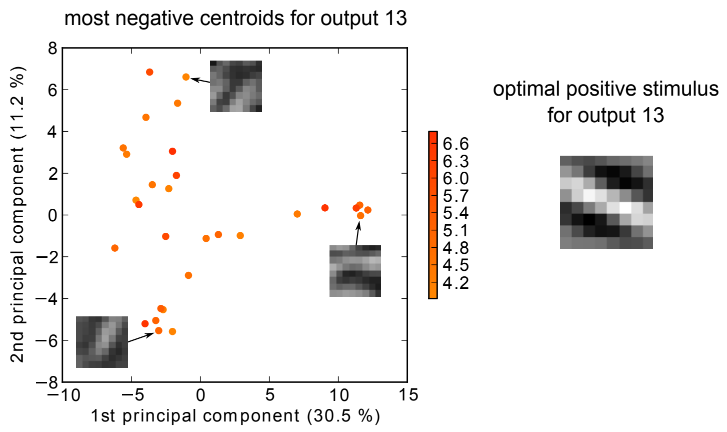


Figure 7.8: **Maximal output centroids for an output with secondary response lobes (natural stimuli).** The optimal positive stimulus is shown on the right side. The two preferred orientations can be roughly identified in the centroid plot, which is shown together with three selected centroid image patches.

slightly different (e.g., the optimal stimuli differ), so no direct comparison is possible. For both types of stimuli the complex-cell outputs and secondary response lobes can be identified.

- As mentioned before we also ran simulations where the vector quantization is trained on the raw input data, before the first linear SFA step. In principle the results were the same as for the vector quantization after the first SFA step. Since the linear SFA step on the lowest layer acts like a low-pass filter the resulting centroid image patches are "edgier". Due to the higher dimensionality the 2D PCA projection plots of the centroids also tend to be noisier.

- The GNG results for natural stimuli are not really convincing. This can be improved by replacing the Euclidean distance measure with a more sophisticated one. Instead of the canonical scalar product (with the Gramian matrix being the unit matrix) we constructed a Gramian matrix from the quadratic form that constitutes the output function. That makes it possible to increase the weight of directions for which the directional derivative of the output is large. The most direct way to get such a Gramian matrix is by taking the square of the quadratic form matrix (making it positive definite).

- Two additional vector quantization algorithms were tested. Since the results did not provide any new insights they are not presented here. These algorithms were also tested in combination with different top-down methods, but showed no improvements over the results with the standard competitive learning quantization (see next chapter).

The first of these vector quantization algorithms used competitive learning, but took the output gradient into account (the vector quantization was performed independently on each output, but without bins). Since the gradient is typically larger in the outer regions of the input data distribution this can be used to concentrate the centroids there, with fewer centroids in the less interesting center region.

In the second algorithm the centroids are directly taken from the data points (no averaging), in a way that maximizes the distances between centroids. This has the effect of spreading the centroids more evenly, away from the center region. However, this did not result in any advantage over the standard competitive learning quantization.

7.5 Discussion

Overall the results in Section 7.4 are not surprising. They merely verify the intuitive picture that follows from [Berkes and Wiskott, 2005] and [Sprekeler and Wiskott, 2010]. In principle the methods in this chapter could provide new insight into the characteristics of the intermediate layers, but 2D centroid plots beyond the first layer did not show any clear structures. Even the comparably simple structures for the layer 1 outputs can be hard to identify (as shown in Figure 7.7D). The 2D projection probably does not leave enough of the structure intact for a simple visual interpretation. The figures in Section 7.4 show the most illustrative cases and do not represent the average visual quality of the results. However, our primary goal is the input reconstruction in the next chapter, for which the aesthetic visualization of the centroids is not relevant. To close this chapter we now briefly discuss related work that deals with capturing the input data distribution.

7.5.1 Manifold Learning Algorithms

Typical manifold learning algorithms like Isomap [Tenenbaum et al., 2000] and Locally-Linear Embedding (LLE) [Roweis and Saul, 2000] take a finite number of sample points and try to uncover their intrinsic manifold structure. The algorithms try to find an embedding of these data points into a space with the intrinsic dimension of the manifold, unwrapping it while preserving the local distance relations between points in the manifold. This can provide some insights into the data distribution, but it would not directly help us with the top-down input reconstruction. The number of data points in our model is also far too large for a direct application of these algorithms. Furthermore, these algorithms solve essentially the same problem that is already addressed by the SFA algorithm: extracting the intrinsic structure of complicated data sets.

The only place where manifold learning algorithms would make sense is the visualization of the centroid set. In Section 7.4 we used PCA to reduce the dimensionality of the centroids for plotting. In principle a manifold learning algorithm could do a much better job, especially in cases like Figure 7.5B (where two principal components are not sufficient). However, our preliminary tests (using the LLE implementation in MDP) only produced results that were inferior to those with PCA.

7.5.2 Other Hierarchical Networks

The method of vector quantization has been used in other models before, but primarily for the feed-forward processing. Models like the Neocognitron [Fukushima, 1980], HMAX [Riesenhuber and Poggio, 1999], or Radial Basis Function Networks [Moody and Darken, 1989] all rely on a set of "centroids" on each layer, which are activated according to their similarity to the stimulus. The signals from all the centroids are then combined in some way to form high-level representations. For example, the centroids in Figure 7.4C could be combined to form a phase invariant edge detector (by using them as linear filters and taking the maximal filter output).

The learned centroids in our model could in principle be used to build a quantized version of the complete SFA network. While the efficiency and practicality of this approach is questionable the result might have some similarity to other slowness based models like VisNet [Wallis and Rolls, 1997] or Hierarchical Temporal Memory [George and Hawkins, 2009].

8 Top-down Input Reconstruction with Gradient Descent

In the previous chapters the goal of input reconstruction in hierarchical SFA networks was established and the input distribution was captured. We now want to conclude this with an actual reconstruction algorithm, which is partly based on the method of gradient descent.

8.1 Methods

The hierarchical SFA network mainly used in this chapter is the same one as in the previous chapter (see Section 7.2), trained with the letter stimuli. It includes the competitive learning vector quantization, so the centroids are available.

8.1.1 Gradient Descent

Gradient descent is a simple first-order optimization algorithm for finding a local minimum (or maximum) of a real-valued scalar function $f(\mathbf{x})$ [Fletcher, 1980]. It requires a starting point $\mathbf{x}_0$ and then iteratively calculates new points $\mathbf{x}_i$ by repeating the following two steps:

1. Calculate the gradient ∇f at the point $\mathbf{x}_i$.
2. Move along the direction of the gradient to find a better point, i.e.,

   $$\mathbf{x}_{i+1}(t_i) = \mathbf{x}_i + \gamma_i \nabla f(\mathbf{x}_i), \tag{8.1}$$

   where $\gamma_i \in \mathbb{R}$ is chosen such that it minimizes $f(\mathbf{x}_{i+1})$. Depending on f it can be a challenging problem to find the optimal γ_i, but since the search is performed along a single line this is much simpler than the original optimization problem. Various heuristics for this *line-search* problem are available [Fletcher, 1980], but as we will see, those are not needed in our case.

Unfortunately a gradient descent can run into various problems. The most serious issue is that the algorithm often gets stuck in a local minimum, which can be far less optimal than the global minimum.

Single Node Optimization

The objective of top-down stimulus reconstruction is to find an input that produces a given output value $\mathbf{y}_t$ (t for target). We first address this optimization problem for the output and input of a single node in one layer in the SFA network (i.e., the node consisting of the two linear SFA steps and the quadratic expansion in between). To perform a gradient descent we define the error measure $E(\mathbf{y})$ by taking the squared Euclidean distance between the output vector $\mathbf{y}$ and the target vector $\mathbf{y}_t$. The output vector $\mathbf{y}$ is defined by a given input $\mathbf{x}$ for the network node, so the error is

$$E(\mathbf{x}) = (\mathbf{y}_t - \mathbf{y}(\mathbf{x}))^2. \tag{8.2}$$

Each component of $\mathbf{y}$ is defined by a second degree polynomial. The function $E(\mathbf{x})$ is therefore a fourth degree polynomial in the components of $\mathbf{x}$ and the exact gradient ∇E is easy to calculate.

With the exact gradient at the current point $\mathbf{x}_i$ one can move to the second step of the algorithm. We can insert $\mathbf{x}_{i+1}(\gamma_i)$ from (8.1) into the error measure to get $E(\gamma_i)$, a fourth degree polynomial in γ_i. To find the optimal value for γ_i we need the extrema of $E(\gamma_i)$, which are at the roots of the derivative. Since the derivative of $E(\gamma_i)$ is only a third degree polynomial the three roots are again easy to calculate (with the cubic formula or by numeric means). The real-valued root with the lowest error value is then used to calculate x_{i+1} for the next iteration step.

Incorporating the Centroid Information

In Section 6.2 it was noted that typically there are many inputs $\mathbf{x}$ that produce a given target output $\mathbf{y}_t$, but usually not all of them are equally similar to the original training stimuli. Even if we reach a point with zero error for the output this might still not be a likely stimulus. In order to identify the most likely minimum we need the probability distribution of the training data, which was approximated in the previous chapter with a set of centroids.

The easiest way to apply the centroids is to use them as starting points for the gradient decent. In order to find the best starting point $\mathbf{x}$ for a given target $\mathbf{y}_t$ we first pick the centroid $\mathbf{x}_j^c$ that minimizes the output error. The hope is that this centroid is close enough to the correct minimum, so that the gradient descent moves us further towards it. It is somewhat misleading to speak of the distance between centroid and minimum in this context, because the distance in input space is not relevant for our line search. However, a smaller distance has the effect that the minimum and its region of attraction are more likely to be hit by the search line, because its angular cross section (i.e., the range of directions for which it is hit) is larger. Furthermore, a smaller distance generally makes it more likely that the gradient points in the direction of the correct minimum.

8.1.2 Optimization for Multiple Layers

In order to perform input reconstruction beyond a single node there are basically two choices: treating the whole SFA network as a single function that is optimized globally or optimize it layer by layer. In this chapter we use the second approach, but the global alternative is discussed in Section 8.3.1.

Before addressing the optimization across multiple layers we first have to address the issue of receptive field overlap. In the hierarchical SFA networks the *input fields* at each layer (i.e., the areas from which the nodes receive their input from the previous layer) overlap to some extend, which is important to achieve translation invariance and good network performance in general. Unfortunately it also means that the reconstructed input patches for different input fields overlap, and often the overlapping values will not be equal. Lateral connections might be the best solution for this dilemma, as there is experimental evidence for their importance in the visual cortex [Kennedy et al., 2000]. For example, these lateral connections would allow neighbouring nodes to somehow negotiate matching solutions. However, implementing such a mechanism in our model would be nontrivial. To keep things as simple as possible we therefore use a simple winner-take-all mechanism, which is applied after the reconstruction for each input field is finished. The input fields are first sorted according to their reconstruction error $E(\mathbf{y})$. This enables the input values with low error to overwrite results with a larger error. Naturally this increases the error values for the loosing input fields, but we assume that the lower error inputs still lead to an overall better result.

The next step in the reconstruction procedure is the combination of multiple layers. This is actually quite simple: the reconstruction is started at the top layer, and the reconstructed input from that layer is then used as the target output for the layer below. When this procedure reaches the bottom layer it produces a complete reconstruction in image space. A schematic diagram of this procedure for two layers in the network is shown in Figure 8.1.

Network Priming

It is also possible to perform multiple optimization passes with the layer-by-layer optimization technique. After the bottom layer has been reached, the reconstruction result is feed-forward processed through the network, and the resulting input for each node in the network is stored. Then another top-down reconstruction is performed for the original target $\mathbf{y}_t$, but this time the centroid starting points are replaced with the stored input data. Theoretically this might help to resolve conflicts between overlapping input fields, because the loosing input fields get the chance to restart their optimization based on data from the winning fields. Biologically this mechanism could be interpreted as a short term memory at the synaptic level (e.g., as postulated in [von der Malsburg, 1994]). This method of “priming” the network with a given input before the reconstruction can also be used in other contexts (which are discussed in the results).

All the described optimization techniques were implemented with the BiMDP framework (see Chapter 11). A simple example of gradient descent in this framework is available on the MDP homepage.

8.2 Results

Unless noted otherwise we use an SFA network that was trained with the letter stimuli. The same number of iterations for the gradient descent is used across all layers and

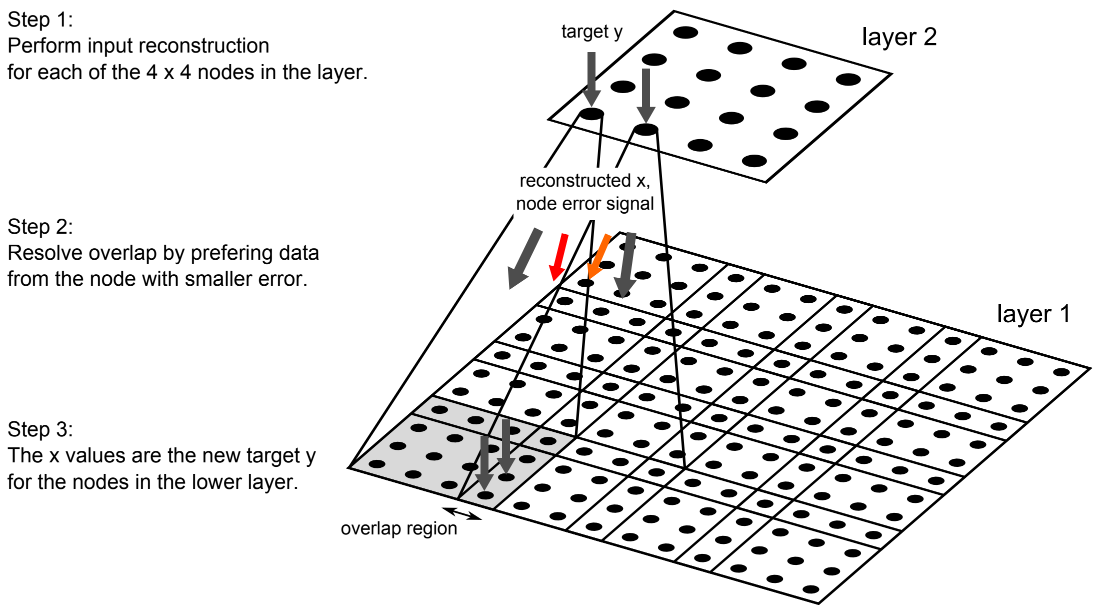

Step 1:
Perform input reconstruction
for each of the 4 x 4 nodes in the layer.
target y
layer 2
reconstructed x,
node error signal
Step 2:
Resolve overlap by prefering data
from the node with smaller error.
layer 1
Step 3:
The x values are the new target y
for the nodes in the lower layer.
overlap region

Figure 8.1: **Top-down reconstruction schematic for two layers.** This figure depicts two layers in the SFA network, in the same way that was already used in Figure 3.2 (but adjusted for the smaller network in this chapter). The flow of information is indicated by the large arrows. Gray arrows symbolize the input and output data. Red arrow symbolize the reconstruction error values from each SFA node. A short description of the procedure is provided on the left side.

all the nodes in each layer (the provided iteration numbers therefore correspond to the description in Section 8.1.2).

The original training images are restricted to the value range between 0 and 255 (the standard 8 bit grayscale range). For the reconstructed input values on the other hand there is no such restriction. To translate the reconstructed data back into an image one can either cut off data values or rescale the data. The rescaled image versions are indicated by the grayscale bar on their right side, while all other reconstruction images in this chapter use the cutoff method.

8.2.1 Stimulus Reconstruction Results

Figure 8.2 shows several input reconstructions. Images from the training data are used as optimization targets for the top-down reconstruction. The results vary depending on the layer that the reconstruction is started from. For example, the reconstruction from layer 2 is started by first calculating the layer 2 target output from the input image. The optimization is then performed in layer 2 to get the reconstructed input values for layer 2. These are then used as the target output for layer 1. The optimization in layer 1 finally produces the reconstructed input, which is translated back into an image.

Not surprisingly, the results in Figure 8.2A show that the reconstructions get worse for the higher layers. The images that were reconstructed from the third layer output are seriously distorted, especially for the tilted A. However, the reconstruction is still localized at the right position and contains several key features of the letter (especially the upper part of the A). The reconstruction error from one layer affects all the layers below, so with each additional layer the quality of the result decreases. There is also the issue of input field overlap: input fields with large errors are partly overwritten, which tends to increase the mismatch between their reconstructed input and target output.

The last row in Figure 8.2A stands out because the target input contains two letters, in contrast to the single letter training data. This results in a third layer reconstruction in which the second letter is completely missing. The likely explanation is the absence of any double letter centroids in layer 3. In this example the closest centroid apparently belongs to the single letter D, and the gradient descent stays in that vicinity. The reconstruction for the lower layers on the other hand performs normally, since their smaller receptive fields mean that each node only sees parts of a single letter.

Figure 8.2B shows reconstruction images without the cutoff method. Their values exceed the training image values, ranging from -50 to about 300. This seems reasonable and subjectively the images do not look much different from the cutoff versions.

The influence of the number of iterations and the centroid impact are illustrated in the next figure. Figure 8.3A shows that the quality of the layer 3 reconstruction does not really increase after about 10 iterations. While the reconstructed input still continues to change it seems that the result only gets nosier. The main benefit seems to be that the lowered error allows the missing segment of the C to be filled in (which was previously overwritten by an overlapping receptive field with lower error). This interpretation is further supported by the error plots in Figure 8.3B. The first plot shows how the Euclidean distance between $\mathbf{y}$ and the target $\mathbf{y}_t$ is successfully reduced to nearly zero

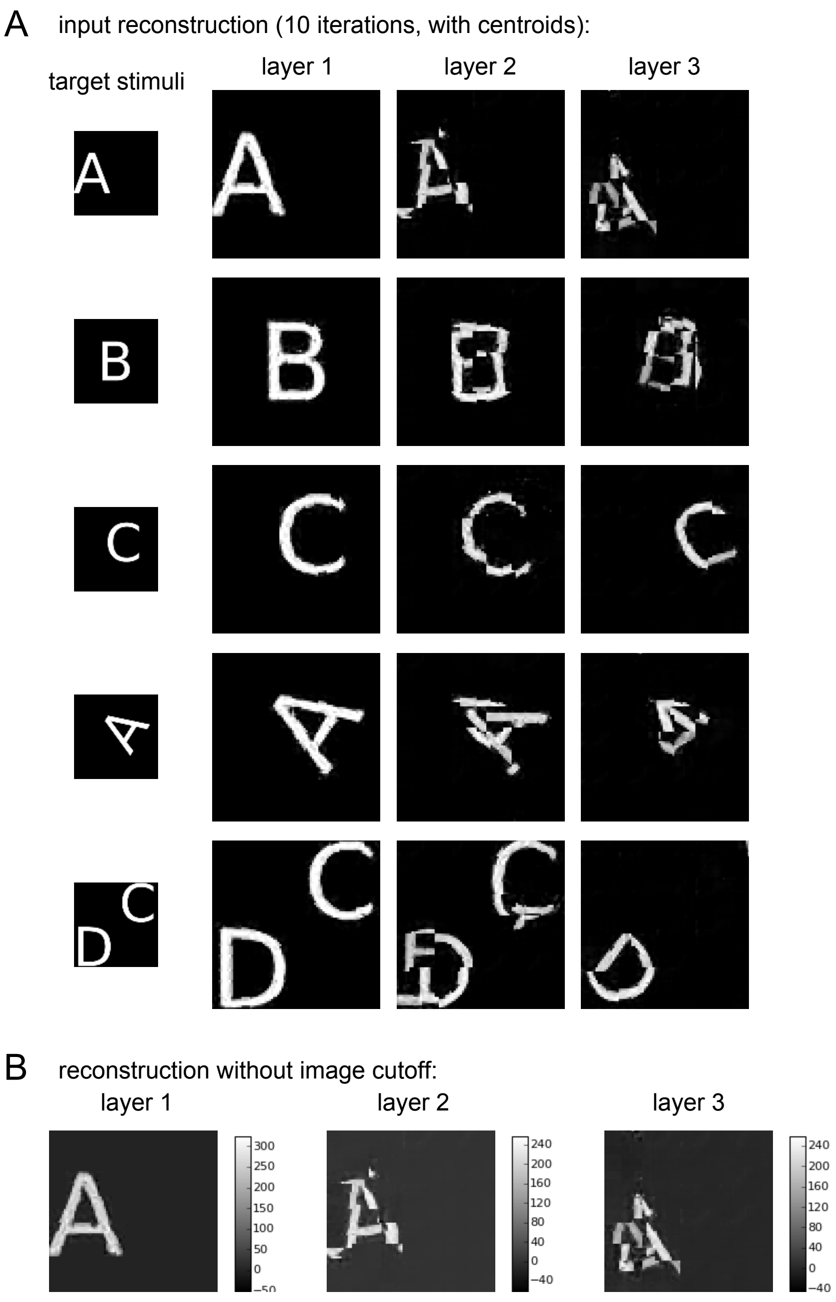


Figure 8.2: **Top-down reconstruction results for different layers.** Several input images are reconstructed, based on the output from different layers of the network (sorted in columns). In **A** the cutoff method was used to force the data into the value range of the training stimuli (0 to 255). In **B** the grayscale range is stretched to match the data range. The grayscale bar next to each image provides the relation between gray value and data value.

(black line). The gray line on the other hand is the error of the reconstructed layer input **x** for each iteration. This is calculated before the input fields are combined, so their overlap does not contribute at this point (the errors for all input fields are simply added up). Surprisingly the input error remains almost constant (though they do change slightly in each iteration).

For the second plot in Figure 8.3B we used a double pass reconstruction (as described in Section 8.1.2): after the first reconstruction this input is used to prime all the layers in the network for a second reconstruction run (in which these inputs are used as starting points instead of the centroids). The second pass starts with practically the same error in **y** as the first pass. From the viewpoint of the third layer the errors that were introduced by the lower layers invalidate any gains from the gradient decent.

Figure 8.3C shows two reconstructions in which random starting points were used instead of the optimal centroids, so the reconstruction fully depends on the gradient descent. Only the lowest layer results show some resemblance to the target stimulus, even though the output error is successfully reduced in all layers (as shown by the error plots).

Overall the results indicate that the input reconstruction is mostly enabled by the centroid information, while the gradient descent is generally unable to further enhance the results. The fact that multi pass reconstruction does not have a positive effect also indicates that the layers are not able to find a globally optimal network state (which is assumed to happen in the brain).

Reconstruction after the Linear SFA Stage

As described in Section 7.3.1 it is possible to perform the vector quantization after the linear SFA stage in each layer. Correspondingly the gradient descent is performed after the linear SFA stage as well. The pseudoinverse of the linear stage is then applied to the reconstruction result to complete the node inversion.

The results for this approach are provided in Figure 8.4. Reconstructions for different layers and different numbers of iteration steps are shown in Figure 8.4A. For the higher layers these results are clearly inferior to those in the previous section. As before they do not improve for a larger number of iterations. For the reconstruction in the last row a random starting point was used instead of the optimal centroid. In this case the result is actually better than the one from the previous section (Figure 8.3C).

Figure 8.4B focuses on the reconstruction error that is introduced by the use of the pseudoinverse. The first layer reconstructions for two receptive field patches are shown together with error plots for the output **y** and input **x**. The high error in **x** is partly a result of the low-pass filtering that is performed by the linear SFA stage. This is also apparent in the reconstructed image patches (which look softer than the target stimuli).

The implications of these results are twofold. On one hand the reconstruction for the higher layers performs poorly due to the information loss in the linear SFA stage. On the other hand it seems that for the first layer the gradient descent does benefit from the reduced number of dimensions (32 versus 64).

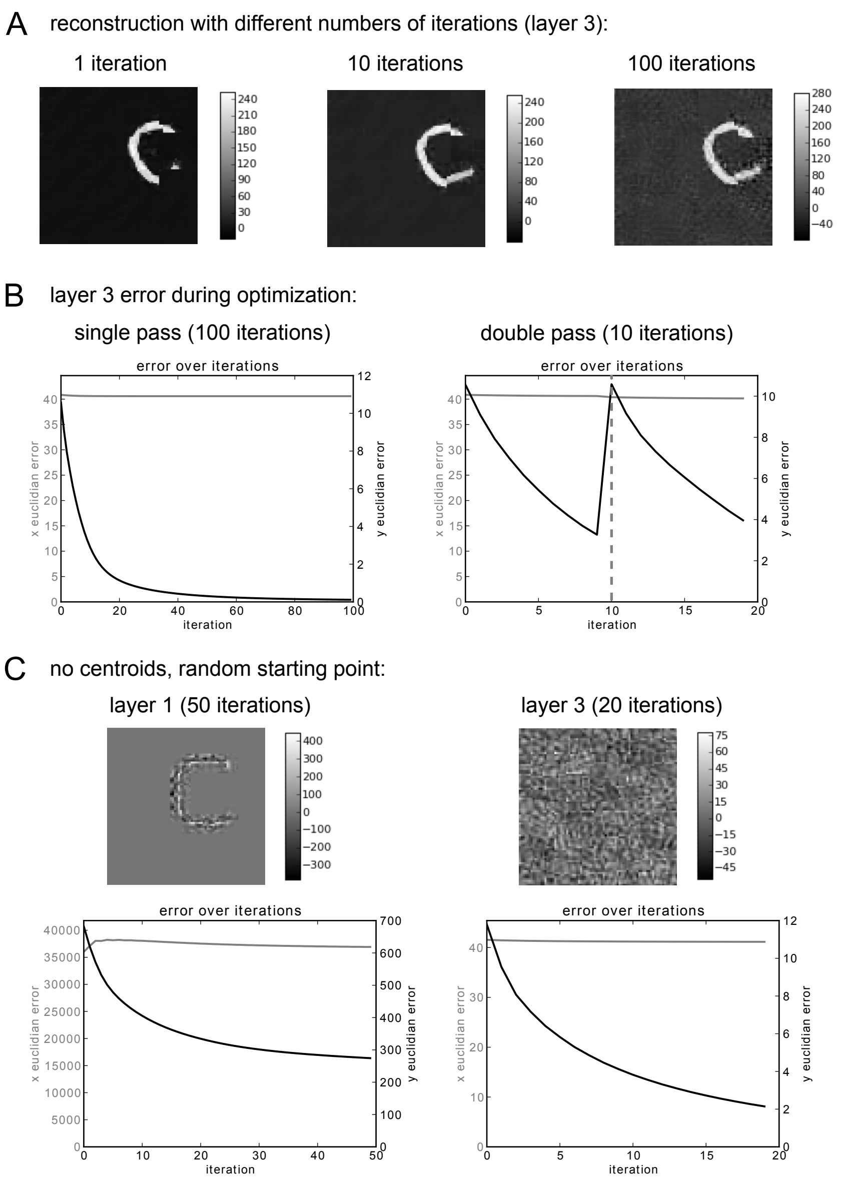
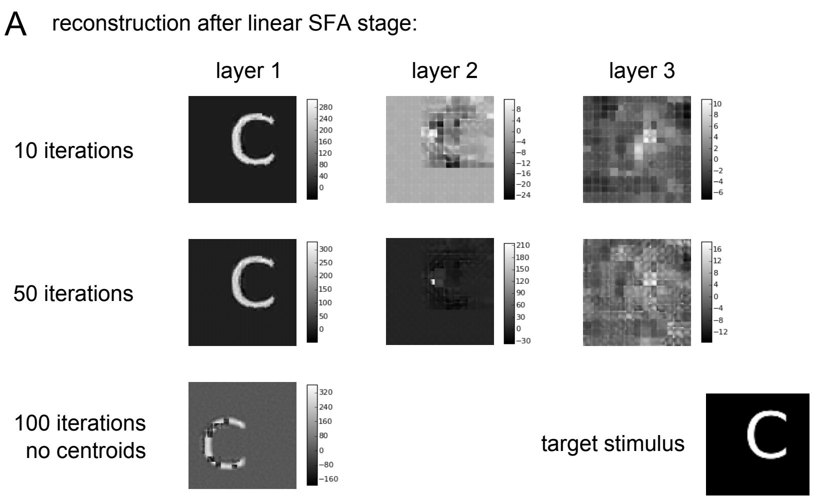


Figure 8.3: **Influence of the number of iterations and centroids.** **A** shows the reconstruction from layer 3 for different numbers of iterations. The first plot in **B** shows the error of the layer input $\mathbf{x}$ (gray line) and output $\mathbf{y}$ (i.e., the Euclidean distance to the target $\mathbf{x}_t$ and $\mathbf{y}_t$) over the number of iterations. In the second plot the same error is shown for a double pass reconstruction (see Section 8.2.1), with the second pass starting at the vertical line (after the first 10 iterations). **C** shows two reconstructions in which random starting points where used instead of the optimal centroid. The corresponding layer errors are shown in the plots below.


A reconstruction after linear SFA stage:
layer 1
layer 2
layer 3
10 iterations
50 iterations
100 iterations no centroids
target stimulus

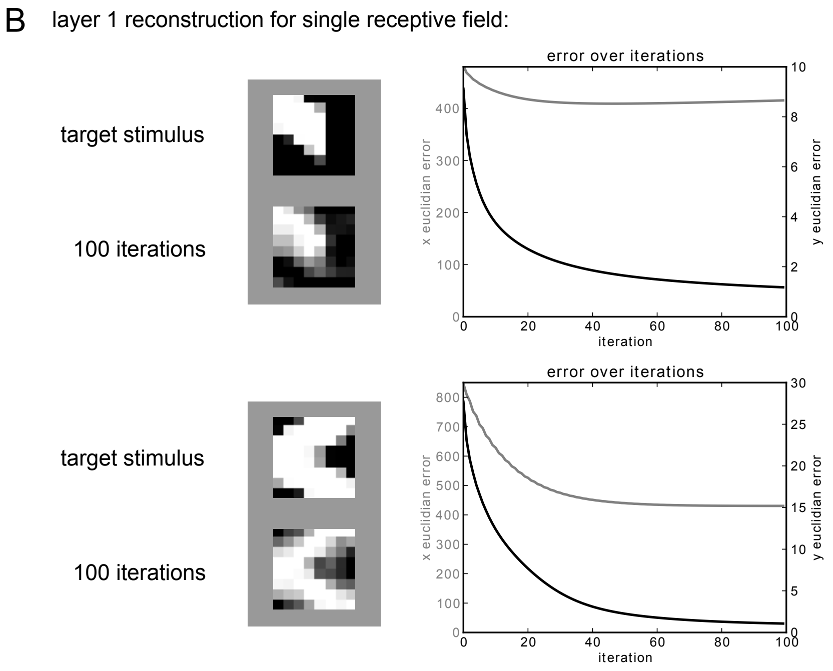


Figure 8.4: **Reconstruction results after the linear SFA stage.** For these examples the vector quantization and gradient descent were performed after the linear SFA stage (and using the pseudoinverse). In **A** the results for different layers and iteration numbers are shown (in the last row with random starting points instead of centroids). **B** shows the layer 1 reconstruction for two receptive field image patches (the gray background is only shown to provide contrast for the white parts of the patches). The error plots on the right contain both the input and the output reconstruction error.

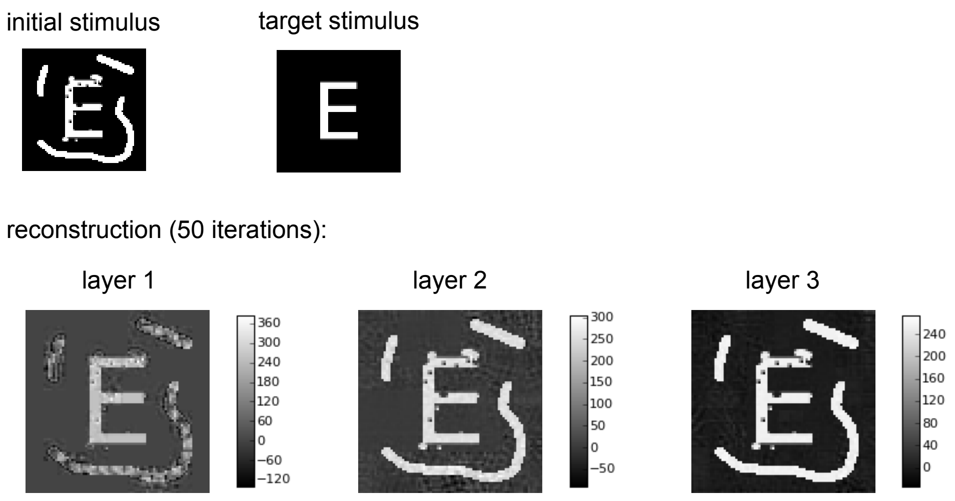


Figure 8.5: **Reconstruction for distorted input stimuli.** The distorted input is shown in the top row, together with the undistorted target stimulus. In the second row the reconstruction results for all three layers are shown. These results were obtained after priming the network with the distorted stimulus, so no centroids were used in the reconstruction.

8.2.2 Reconstruction from Distorted Stimuli

It was shown earlier that priming the network for additional passes does not enhance the performance. The example in Figure 8.5 instead uses a distorted version of the target stimulus to prime the network. However, the results show that the gradient descent fails to improve the reconstruction image.

A similar experiment can be performed by priming the network with a slightly rotated or shifted version of the target stimulus. In principle this should allow us to generate transformations of the input image. Unfortunately this worked just as badly as the priming with distorted stimuli.

We performed multiple experiments in this direction, and they all indicate that the higher layers are stuck in local minima. Even for tiny distortions or transformations the gradient descent is apparently unable to reduce the input error in a meaningful way. The transformation probably would have to be smooth down to the single pixel level. Given the performance of the gradient descent in our earlier experiments this is not surprising. Even the movement of a letter stimulus by a single pixel implies a sizable distance in input space, which is apparently too much for the gradient descent. The invariance of the network with respect to transformations might add to this problem, because the output derivative along these trajectories is small.

8.2.3 Control Experiments

- The reconstruction parameters were varied as widely as possible. This should rule out unlucky parameter choices as the root of the gradient descent problems. For example, we tested the input reconstruction with up to 400 iterations across all three layers, seeing no improvement.

- We also trained and tested an SFA network without the contrast transformations in the letter training set. As mentioned earlier this simplification raised the classification performance of the network to 99.2% (from the original 92.3%). The $\mathbf{y}$ errors for the input reconstructions decreased as well, but otherwise there was no qualitative difference in the results. Due to the less diverse training stimuli the centroid density for the training data is higher, which explains these results.

- We verified that the results from this chapter generalize to other stimulus sets. In Figure 8.6 we present results for a network that was trained with a slightly smaller version of the sphere stimuli from Section 3.3 (matching the smaller network size in this chapter). The reconstruction results are similar in quality to the letter network. We also tested how the sphere network would reconstruct letter stimuli (which are very different from its training data). Figure 8.6 shows how the sphere network fails to reconstruct letter stimuli. This is not surprising, since there are no corresponding centroids.

  Additionally we trained and tested a single layer network with the natural stimuli from Section 7.2.2. The reconstruction performance is comparable to that of the letter network. Without the centroid information the reconstruction performance dropped significantly.

- In an attempt to improve the gradient descent we tested a “shotgun” method, using multiple starting points for the iteration. Instead of only the centroid closest to the target output we picked the nearest 20 centroids and performed the gradient descent for all of them. From the resulting 20 candidates we picked the one with the lowest output error. However, this did not improve the gradient descent performance in any significant way.

- We tested a different method of resolving the overlap of the input fields. Instead of the winner-take-all approach we took the mean of the conflicting values. This did not improve the reconstruction quality.

- In order to get a “chance-level” for the gradient descent we ran simulations in which the gradients were replaced by random vectors (with a Gaussian distribution). The line search was performed in the normal way, based on these random directions. As expected, the output error was still reduced, but much slower than with the correct gradients. The input error increased with the number of iterations, in contrast to the decrease with the real gradients (the error also increased at a faster rate than the decrease). In the input reconstructions for the random gradients it looks like pink noise is added by the gradient descent.

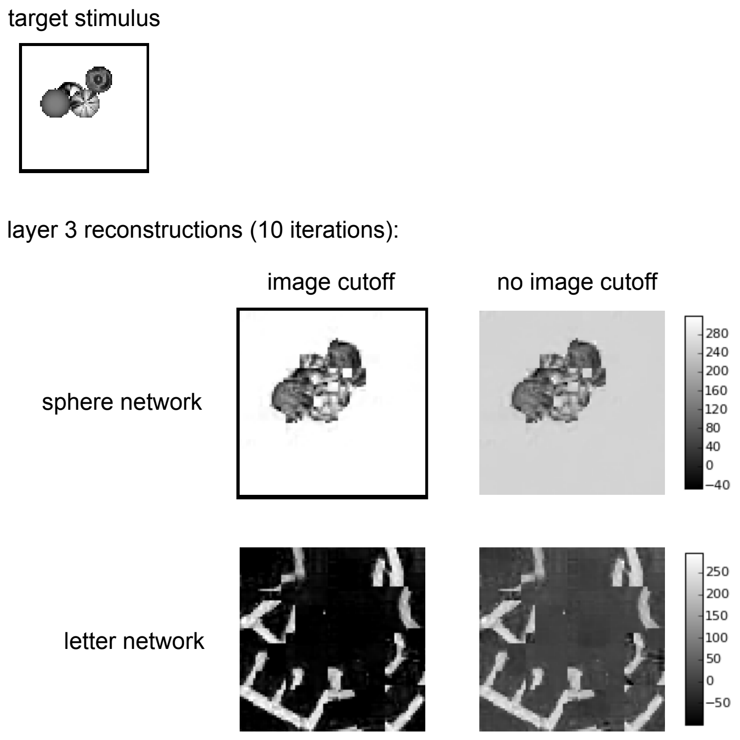


Figure 8.6: **Reconstruction of sphere stimuli.** In this control experiment the reconstruction was performed for sphere object stimuli. The first row shows the third layer reconstructions from an SFA network that was trained with sphere stimuli. The second row was reconstructed by the standard network trained with letters.

8.3 Discussion

The findings in this chapter are twofold:

- All our results indicate that the gradient descent method performs poorly for our hierarchical SFA model. It might be deceiving that the functions representing each node are only second order polynomials. In order to evaluate the difficulty of the problem one has to take the dimensionality into account as well. Even after the linear SFA step the input typically has 32 dimensions, which means that there can be up to 2^{32} local minima in the error function for each node. One possible solution might be to develop optimization techniques specifically for this type of function, similar to what has been done in [Berkes and Wiskott, 2006].
- Only the centroids enable the top-down reconstruction of input stimuli, which raises the question of scalability. In our current implementation the centroids can be interpreted as "grandmother cells" [Gross, 2002], i.e., they correspond to a single very specific stimulus. For a highly restricted stimulus set this might not be a problem, but as the stimuli get more diverse this is likely to fail. The required number of centroids and the computational complexity would explode, at least for our naive approach.

One can conclude that fundamental changes to our model are needed to enable truly general top-down processes. This is unfortunate, since the stimulus reconstruction method could have opened up new ways to analyze our hierarchical networks. The top-down methods could have also been used to implement attentional mechanisms for multiple objects or cluttered scenes.

The method of vector quantization plus gradient descent can be considered as a brute force approach. We also tested multiple other reconstruction methods, in the hope of finding a more elegant approach. However, none of these methods were able to match the reconstruction performance of the brute force solution. We end this discussion by comparing our approach with various other models and techniques.

8.3.1 Global Network Optimization

The optimization layer by layer, one network node at a time, has the advantage that only a very small part of the network is involved in each step of the calculation. It is a natural fit for the local feed-forward structure of the network. Connectivity in the the brain is thought to be of the so called small-world type (e.g., [Humphries et al., 2007]), with the majority of connections being local. This seems to imply that local optimization is biologically more plausible than global approaches (in addition to various more technical reasons).

However, if the goal is only to analyze the SFA network then the biological argument is not relevant. In that case one can perform the gradient descent for the whole network at once. The resulting error function is a polynomial of a much higher degree (e.g., degree 16 for the three layer network). Apart from technical issues with memory consumption this also means that the line-search gets harder.

Christian Hinze tested this global approach for our hierarchical SFA models [Hinze et al., 2009]. Instead of input reconstruction his goal was to find optimal stimuli for higher level network outputs. He used an approximative line-search algorithm. Alternatively it should be possible to calculate the exact error polynomial along the search line and to apply numerical methods for finding the roots (which is no problem for polynomials of this degree).

It turned out that the resulting optimal stimuli for higher network layers are hard to interpret. There is one additional obstacle in this approach, due to the need of a vector norm constraint for the optimal stimuli [Berkes and Wiskott, 2006]. This complicates the evaluation of the gradient descent performance in this setting. Nevertheless the results seem to support our conclusion that gradient descent is problematic for SFA networks. It also suggests that the use of a global gradient descent for input reconstruction would not return better results than our localized approach.

For a completely global approach the vector quantization could be done in the original image space as well, so each centroid would correspond to a full image. However, this would make the distance calculations in the algorithm very costly and the number of centroids might have to be increased as well.

8.3.2 HMAX Model

The HMAX model [Riesenhuber and Poggio, 1999] is similar to the earlier Neocognitron [Fukushima, 1980] (e.g., [Fukushima, 2005] for top-down processes). It is a hierarchical feed-forward network, with each layer consisting of a so called S and C-part. The S-part basically contains centroids (e.g., learned with the k-means algorithm) which act as linear filters. Like in our model each layer consists of multiple input fields with identical centroids ("weight-sharing"). In the C-part a nonlinear maximum operation (MAX) is used to combine the output from identical centroids across multiple input fields, thus achieving basic translation invariance. On the lowest layer the S-part corresponds to simple-cells, while the C-part is associated with phase-invariant complex-cells.

In [Dura-Bernal et al., 2010] a feedback mechanism for the HMAX model has been introduced. The basic idea is to turn the model into a Bayesian network in which belief propagation can be implemented. So called belief units are inserted between the network layers to maintain the current state, consisting of a probability distribution for the input and output of the layers. The discrepancy between beliefs at different layers generates an error signal, which is used to modify one belief unit at a time. This can be done bottom-up (feed-forward) or top-down (feedback), or in any combination thereof. In [Dura-Bernal et al., 2010] this has been applied to an image of the famous Kanizsa square [Kanizsa, 1987], resulting in the emergence of illusionary contours (which is the intended result as it matches the response of the visual system).

While these results are certainly interesting, it is not yet clear if they can be scaled up to more complex stimuli. From the theoretical point of view the ability to work directly in a probabilistic framework is a huge advantage. It is possible that such an approach is absolutely required for implementing general top-down processes. In that case the "shortcuts" that we used in our model must fall short.

8.3.3 Deep Belief Networks

Deep Belief Networks (DBN) are hierarchical networks consisting of restricted Boltzmann machines [Bengio, 2009]. They are probabilistic and generative, i.e., the units in each layer are trained with the objective to reproduce the training stimuli (based on the layer output). This implies that the network has to learn the hidden parameters that underlay the stimulus generation, which is effectively what our SFA network does, too. The recent popularity of DBNs is largely based on the learning algorithm that has been described in [Hinton et al., 2006] (a more detailed description is provided in [Swersky et al., 2010]).

The fact that a DBN is a generative model means that input reconstruction is a natural task for such a network. This has been successfully demonstrated in [Hinton et al., 2006] for the well known MNIST data set of handwritten digits. This technique is basically an iterative sampling procedure for the implicit probability distribution of the network. It can also be used to characterize the units in higher network layers by visualizing their preferred stimuli in the input image space. An alternative method based on gradient descent has been described in [Erhan et al., 2010] and leads to qualitatively different results. This method has been also used to visualize the network invariance, loosely based on an idea from [Berkes and Wiskott, 2006]. While this did not produce any clear results, it might be an approach that could be applied in other situations as well (e.g., for our hierarchical SFA networks).

Overall the DBN results for top-down processes are very interesting and are generally more convincing than the reconstructions in this chapter. However, the computation in a DBN layer is essentially defined by simple scalar weights w_{ij}, which is fairly restricted compared to the quadratic expansion in SFA. The description in [Swersky et al., 2010] also suggest that the training of a DBN is a very intricate process (e.g., the weights start to diverge after some time, so training must be stopped at the right point). This raises questions about the scalability to more complex stimuli.

Interestingly there is recent work on higher-order Boltzmann machines [Memisevic and Hinton, 2010], which are used to learn spatial transformations. In this context higher-order means that three variables (one of them being the "output") are linked in the calculation, instead of the usual two (single input and output). This has some similarity to the quadratic expansion that we use, and which is critical for the transformation invariance in our model. Not surprisingly the results in [Memisevic and Hinton, 2010] are highly similar to those in [Berkes and Wiskott, 2005]. However, this higher-order model is purely focused on transformations and has lost the top-down capabilities of the standard DBN.

Part III

The Modular toolkit for Data Processing

9 Introduction to the Modular toolkit for Data Processing (MDP)

The Modular toolkit for Data Processing (MDP) is an open source library of widely used data processing algorithms, and a framework to combine them according to a pipeline analogy. It is written in the Python[1] programming language and was first released to the public in 2004. In this chapter the basic elements of MDP are introduced, which were written by the project founders Pietro Berkes and Tiziano Zito (who continue to maintain MDP). This introduction closely follows [Zito et al., 2008] and is mostly included to provide the context for the following chapters.

I was invited to join the team of MDP maintainers in 2008 and the parts that I contributed since then are described in the following chapters 10 and 11. These new parts are the foundation on which most of the work presented in this thesis was implemented. They have also been published in [Zito et al., 2008] and the upcoming [Wilbert et al., 2011]. An extensive tutorial and further examples are available on the MDP homepage[2]. At the time of this writing the current version of MDP is 3.0.

9.1 The Python Programming Language

The use of the Python programming language in computational neuroscience and other scientific areas has been growing steadily for the past 10 years. This was made possible by the maturation of open-source libraries like NumPy / SciPy [Jones et al., 2001] (for fast matrix and array operations) and Matplotlib [Hunter, 2007] (for data plotting). While these libraries provide the essential functionality needed by any scientist there is also a large number of more specialized projects like MDP. Today this combination allows Python to rival and often surpass commercial alternatives like Matlab®.

The dynamic nature of Python enables scientific programmers to quickly develop efficient and well structured software, with excellent support for prototyping and reusability. Python is easy to learn while at the the same time offering powerful language features. Among the supported features are, for example, closures, metaprogramming, and coroutines (all of which are used by MDP). Python supports both object oriented and, to a lesser extend, functional programming. It is generally estimated that compared to an implementation in C++ the Python solution requires only about one fifth of the number of lines of code, with a correspondingly reduced development time. One reason for this is that Python is dynamically typed, meaning that the type of a variable is only determined at runtime (as opposed to statically typed languages like C++ or Java, where

[1]`http://python.org/`
[2]`http://mdp-toolkit.sourceforge.net`

the type must be known at compile time). This simplifies the development process and provides enormous flexibility (e.g., as illustrated in Section 10.3).

On the other hand the runtime speed of Python code is typically much lower than that of C, in some cases by a factor of more than 100. The first reason is that the dynamic nature of Python requires more overhead at runtime. Secondly the standard Python implementation runs Python code in a relatively simple interpreter, which cannot match the speed of compiler generated machine code. Fortunately Python has excellent support for other languages like C, C++, or Fortran (thanks to Cython [Seljebotn, 2009] it is even possible to seamlessly mix C and Python code). Therefore it is standard practise to isolate performance critical parts of the code and implement only those in a fast lower-level language. In many cases the critical functionality is already available in libraries like NumPy, which provide a convenient Python interface for highly optimized C and Fortran code (e.g., based on the same linear algebra libraries that are used by Matlab®).

9.2 MDP in the Scientific Python Ecosystem

While MDP has been written in the context of theoretical research in computational neuroscience, it was designed to be useful in any context where trainable data processing algorithms are used. Naturally it could also be described as a machine learning library. MDP relies on NumPy for fast matrix and array calculations and optionally uses SciPy for some further optimizations. Care has been taken to follow best practises in software design, like test coverage (the test suite contains more than 600 automated tests), following established design principles, and having readable and well documented code.

With over 20,000 downloads since its first release in 2004, MDP has become a widely used Python scientific software. It is distributed under an open source BSD license. MDP is also available in the Debian Linux distribution[3], the Python(x,y) distribution[4], and MacPorts[5] for MacOS. Several scientific libraries utilize MDP, like PyMVPA [Hanke et al., 2009] (multivariate pattern analysis), PyMCA [Solé et al., 2007] (X-ray fluorescence), OpenElectrophy [Garcia and Fourcaud-Trocmé, 2009] (analysis of cellular recordings), and the Oger toolbox[6] (language processing and reservoir computing). MDP has been used in various research projects, resulting in more than a dozen citations from other groups.

MDP shares common goals with other Python machine learning libraries, like Orange [Demšar et al., 2004], PyBrain [Schaul et al., 2010], and the recent scikits.learn library[7]. All these libraries differ in their design approach and focus on different sets of algorithms. To maximize code reuse and cooperation, our long-term philosophy is to automatically wrap the algorithms defined in these libraries when they are available. For example, MDP provides wrappers for LIBSVM [Chang and Lin, 2001] and Shogun [Sonnenburg

[3]`http://packages.debian.org/sid/python-mdp`
[4]`http://www.pythonxy.com`
[5]`http://trac.macports.org/browser/trunk/dports/python/py26-mdp-toolkit`
[6]`http://www.reservoir-computing.org/organic/engine`
[7]`http://scikit-learn.sourceforge.net/`

et al., 2010], and recently gained support for scikits.learn. This should enable users to seamlessly combine the algorithms defined in MDP with the functionality in those libraries. There are also ongoing efforts to increase the cooperation with other projects, especially with scikits.learn.

9.3 Basic Elements of MDP

The core of MDP consists in a collection of supervised and unsupervised learning algorithms and other data processing units that are encapsulated in *nodes* with a standardized interface. Multiple nodes can be combined into data processing sequences, which are called *flows*. Given a set of input data, MDP takes care of successively training or executing all nodes in the flow. This allows the user to specify complex algorithms as a series of simpler data processing steps in a natural way.

The base of available algorithms in MDP is steadily increasing and includes signal processing methods (e.g., Principal Component Analysis (PCA), Independent Component Analysis algorithms like FastICA [Hyvärinen, 1999], Slow Feature Analysis (SFA)); manifold learning algorithms (e.g., Locally Linear Embedding [Roweis and Saul, 2000], Growing Neural Gas); various classification methods; and many others.

The functioning of most data processing algorithms can be divided in two distinct phases: in the first phase, a set of training data is used to adjust a set of internal parameters (the *training phase*); in the second phase, the algorithm makes use of the learned parameters to process test data (the *execution phase*).

This basic structure is mirrored in MDP using an object oriented design, where algorithms are represented by classes derived from a `Node` base class. Each node object is characterized by an input dimension (i.e., the dimensionality of the data points), an output dimension, and a data type (i.e., the numerical type of the data, usually single and double precision floating-point numbers). By default, these attributes are inherited from the input data.

9.3.1 Usage of Nodes

Node objects are finite state machines; if the algorithm requires one or more training phases, the node stays in a training state until instructed by the user to proceed to the execution state.

For example, the Principal Component Analysis (PCA) algorithm requires the computation of the mean and covariance matrix of a set of training data from which the principal eigenvectors of the data distribution are estimated. MDP offers an implementation of this algorithm in the class `PCANode`. The node can be trained on the data using the interface common to all nodes: `PCANode.train(x)` analyzes a new batch of data `x`, and updates the estimation of mean and covariance matrix. `PCANode.stop_training()` finalizes the algorithm by computing and selecting the principal eigenvectors. Nodes can also have multiple training phases, in which case `PCANode.stop_training()` starts the next training phase instead of ending the training altogether.

Once the training is finished, new data can be projected onto the principal components by calling the `PCANode.execute(x)` method. If the transformation specified by the underlying algorithm is invertible, the node can also be executed "backwards" using the `PCANode.inverse(y)` method. In the case of PCA this corresponds to projecting a vector in the principal components space back to the original data space. A full example of node usage is given in the following code:

```
import mdp
import numpy as np
# create dummy data: 50 data points, 10-dimensional
data = np.random.random((50,10))
# create PCANode instance; reduce data dimensionality to 5
node = mdp.nodes.PCANode(output_dim=5)
# train algorithm
node.train(data)
node.stop_training()
# project data on the five principal components
proj_data = node.execute(data)
```

The `Node` base class was designed to allow arbitrarily large sets of data by accepting input data in multiple chunks. It is thus possible to perform computations with data that would not fit into memory, or to generate the data on-the-fly as needed:

```
# iterate over training data chunks, which must be arrays
for data_chunk in data_source:
   node.train(data_chunk)
node.stop_training()
```

This works for all data sources that are *iterables*, i.e., data containers that support the Python interface for iteration (like lists and tuples).

9.3.2 Implementation of Nodes

To make writing new nodes as convenient as possible for algorithm implementors, the `Node` base class takes care of issues like error checking, automatic casting, and so on. Typically, developers only have to override the methods `_train`, `_stop_training`, and `_execute` to implement their algorithm[8]. A public version of these methods (without the underscore) is provided by the `Node` base class. These template methods perform all of the tasks related to the framework, and then delegate to the private methods for the actual computations. An example for a simple node implementation is given in Figure 9.1.

[8]In the Python convention, a leading underscore indicates that the method is private, only intended for internal use by the class.

```
class MeanFreeNode(mdp.PreserveDimNode):
    """Node to subtract the mean from the data."""

    def __init__(self, input_dim=None, output_dim=None,
                 dtype=None):
        """Initialize the internal data structure."""
        # call the __init__ method of the parent class
        super(MeanFreeNode, self).__init__(input_dim=input_dim,
                                           output_dim=output_dim,
                                           dtype=dtype)
        self.avg = None  # mean of the data points
        self.tlen = 0  # number of data points

    def _train(self, x):
        """Calculate the mean of the training data."""
        if self.avg is None:
            self.avg = np.zeros(self.input_dim,
                                dtype=self.dtype)
        self.avg += np.sum(x, axis=0)
        self.tlen += x.shape[0]

    def _stop_training(self):
        self.avg /= self.tlen

    def _execute(self, x):
        """Return the data after subtracting the recorded mean."""
        return x - self.avg

    def _inverse(self, y):
        return y + self.avg
```

Figure 9.1: **Implementation of a new node class.** The `MeanFreeNode` calculates the mean of the training data and then subtracts this mean from the data during execution. By deriving from the `PreserveDimNode` base class this node inherits dimensionality checks which ensure that the `input_dim` matches the `ouput_dim`.

9.3.3 Flows

A *flow* is a sequence of nodes that are trained and executed together to form a more complex algorithm. Input data is sent to the first node and is processed by the subsequent nodes along the sequence. The general flow implementation automates the training (including supervised training and multiple training phases), execution, and inverse execution (if defined) of the whole sequence. For example PCA can be used to reduce the dimensionality of some data, after which quadratic SFA is applied (see [Berkes and Wiskott, 2005]). This is done by first creating a `Flow` with the node instances:

```
# create a flow containing two nodes
pca_node = mdp.nodes.PCANode(output_dim=10)
sfa_node = mdp.nodes.SFA2Node()
flow = mdp.Flow([pca_node, sfa_node])

# train the flow with random data
data_source1 = (np.random.random((50,20)) for _ in range(3))
data_source2 = (np.random.random((50,20)) for _ in range(6))
flow.train([data_source1, data_source2])

# pipe some data through the fully trained flow
x = np.random.random((50,20))
y = flow.execute(x)
```

The training and execution are performed similarly to the node class, but this time the iteration over the training chunks is performed automatically by the `flow`. Different data sources were specified for the individual nodes, which is useful when different algorithms require different kinds or different amounts of training data (to demonstrate this the `sfa_node` above is given six data chunks, while the `pca_node` only gets three). Data sources only have to be iterables. The first data source in the list is used for the first node in the flow and so on. Most importantly the flow automatically processes the training data for the second node through the first node. For example the data from `data_source2` are executed by `pca_node`, and the resulting output is then used for the training of `sfa_node`. The flow is responsible for managing the training and execution of nodes, automating these standard tasks.

Nodes with supervised training are a special case, since they require an additional supervision signal during training. For those nodes the iterable can return tuples, in which the first entry is the standard data and the following entries contain the additional arguments that the node requires. However, as shown in Chapter 11 the BiMDP framework provides a more general solution for these cases.

In order to allow easy access to the internal nodes the flow class implements the standard Python container interface and thus supports most of the features of Python lists: one can append new nodes, extract or insert nodes, and concatenate flows. Since the `+` operator is overloaded for nodes and flows one can also create a flow by adding nodes, e.g., `flow = pca_node + sfa_node`.

10 Hierarchical SFA with MDP

The MDP features described in this chapter have been developed for the hierarchical SFA model discussed in Part I and II of this thesis. These parallelized hierarchical SFA models were first implemented in Python by Mathias Franzius [Franzius et al., 2007]. The work presented in this chapter is a completely new implementation, but especially in the area of parallelization it did benefit a lot from his experience. My new implementation is intended to provide a modular design that can be fully integrated into MDP. Furthermore it allows for the inclusion of top-down processes as described in Part II of this thesis (the corresponding features in MDP are described in Chapter 11). All the results of this thesis and the corresponding publications have been obtained with the new MDP code, while the earlier results in [Franzius et al., 2008] are based on the earlier code for hierarchical networks.

10.1 Hierarchical networks

The `hinet` subpackage has been added to MDP for arbitrary feed-forward architectures and in particular hierarchical networks like the one presented in Section 3.2. Following the principle of modularity, it defines three basic classes as building blocks (which are all derived from `Node`): `Layer`, `FlowNode`, and `Switchboard`.

10.1.1 Layer

The first class, `Layer`, works like a horizontal version of a flow. It acts as a wrapper for a list of nodes that are trained and executed in parallel. For example, two nodes with 100-dimensional input can be combined to form a layer with a 200-dimensional input:

```
node1 = mdp.nodes.PCANode(input_dim=100, output_dim=10)
node2 = mdp.nodes.SFANode(input_dim=100, output_dim=20)
layer = mdp.hinet.Layer([node1, node2])
```

The first half of the 200-dimensional input data is automatically assigned to `node1` and the second half to `node2`. In the same way the output of the two nodes is combined and has a dimension of 30. A layer can be trained and executed just like any other node.

10.1.2 FlowNode

In order to to build complex and modular structures `hinet` provides a wrapper class for flows (i.e., vertical stacks of Nodes), called `FlowNode`. While the interface of `Flow` looks similar to that of `Node` it is not compatible and therefore `FlowNode` is needed as an adapter. For example, one can replace `node1` in the previous example with a `FlowNode`:

```
node1_1 = mdp.nodes.PCANode(input_dim=100, output_dim=50)
node1_2 = mdp.nodes.SFANode(input_dim=50, output_dim=10)
node1 = mdp.hinet.FlowNode(node1_1 + node1_2)
node2 = mdp.nodes.SFANode(input_dim=100, output_dim=20)
layer = mdp.hinet.Layer([node1, node2])
```

`node1` has two training phases in this example, one for each internal node. `layer` therefore has two training phases as well and behaves like any other node with two training phases. By combining and nesting `FlowNode` and `Layer`, it is thus possible to build modular node structures.

10.1.3 Switchboard

When implementing networks one generally has to route different parts of the data to different nodes in a layer (e.g., to model a receptive field structure like in the visual system). This functionality is provided by the `Switchboard` node. A basic `Switchboard` is initialized with a 1-D array with one entry for each output dimension, containing the index of the input dimension that it receives its value from, e.g.:

```
switchboard = mdp.hinet.Switchboard(input_dim=6,
                                    connections=[0,1,2,3,4,3,4,5])
x = np.array([[2,4,6,8,10,12]])
print switchboard.execute(x)
# should print: array([[ 2, 4, 6, 8, 10, 8, 10, 12]])
```

The switchboard can then be followed up by a layer containing nodes with corresponding dimensionality, as illustrated in Figure 10.1. In practice the `connections` array is often not specified by hand, which would be tedious. Instead MDP supplies a couple of classes that are derived from `Switchboard`, but which take a higher level description of the connection scheme and construct the connections from that.

The most typical connection scheme in the context of this thesis is the quadratic receptive field structure shown earlier (e.g., in Figure 3.2). This receptive field structure implies that the data has a 2-D structure, which is not apparent in the 1-D data format used by MDP. To define such a switchboard one therefore has to provide information about the underlying 2-D structure, the receptive field size, and the spacing of the receptive fields. One must also consider that a single SFA node has a multi-dimensional output. For example each “pixel” in a higher layer input corresponds to 32 SFA outputs (similar to a color image with three values per pixel). This is abstracted in the `ChannelSwitchboard` node, in which n inputs can be grouped together to form a *channel*. Based on that we define the `Rectangular2dSwitchboard` class, which takes the receptive field parameters and calculates the connections from that. This allows the convenient construction of a complete hierarchical network from the parameters specified in Table 3.1.

In addition to the `Rectangular2dSwitchboard` class there are a couple of other switchboard classes for 2D receptive fields available in MDP, which implement more complicated receptive field structures.

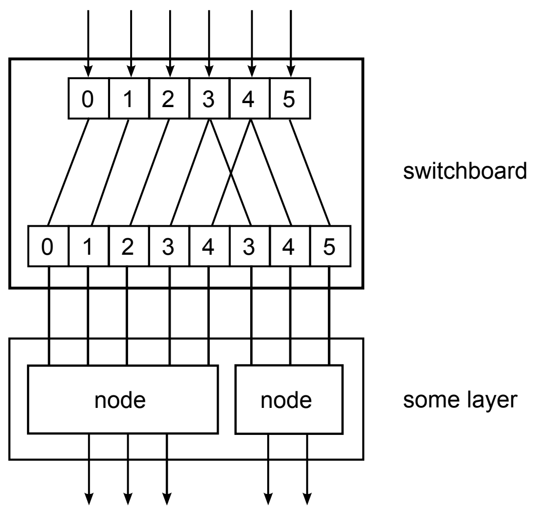


Figure 10.1: **Example of a simple feed-forward network.** The values of input 3 and 4 are duplicated and fed to both nodes, like in the case of overlapping receptive fields. In this figure the data is depicted flowing down from the top (this convention is used throughout the chapter, since it is closer to the technical implementation).

10.1.4 Additional Tools

With the three building blocks that were presented it is now possible to construct arbitrary hierarchical networks, like the ones described in Section 3.2. In Figure 10.2 we present the code for implementing a network layer as it was illustrated in Figure 3.2 (which is the third layer in the hierarchical SFA model). The `hinet` subpackage also includes functionality to create a graphical visualization of any such network. This visualization is provided in the form of HTML code, which can be viewed in a webbrowser. This is as simple as:

```
mdp.hinet.show_flow(flow)
```

An example display for the network from Figure 10.2 is shown in Figure 10.3. It is also possible to specify custom visualizations for specific node types via the extension mechanism that is introduced later (e.g., to define which internal parameters are displayed).

In practice even the task of picking consistent parameters for large networks can be tedious. To help with this, MDP provides a simple graphical user interface for experimenting with the network structure. This application can be found among the MDP examples (under the name `hinetplaner`). Basically this is an interactive version of the passive network display. The user frontend runs in the browser (using HTML and JavaScript) and communicates with a simple Python webserver backend. Originally this was also indented as a framework for interactive analysis of the data processing in a network, but due to the results with gradient descent in Chapter 8 we did not reach that stage.

```
switchboard = mdp.hinet.Rectangular2dSwitchboard(
                                        in_channels_xy=14,
                                        field_channels_xy=4,
                                        field_spacing_xy=2,
                                        in_channel_dim=32)
sfa_node = mdp.nodes.SFANode(input_dim=switchboard.out_channel_dim,
                             output_dim=32)
sfa2_node = mdp.nodes.SFA2Node(output_dim=32)
flownode = mdp.hinet.FlowNode(sfa_node + sfa2_node)
sfa_layer = mdp.hinet.CloneLayer(flownode,
                          n_nodes=switchboard.output_channels)
flow = switchboard + sfa_layer
```

Figure 10.2: **Code example for a single layer in an SFA network.** This corresponds to the network shown in Figure 3.2, which is the third layer in the hierarchical SFA network. Only the cutoff- and noise-nodes are omitted here for simplicity. The receptive field size is 4×4 pixels, with each pixel having dimension 32 (the SFA output dimension in the previous layer). The step size of the receptive fields is 2 channels. The data from one receptive field is considered to form an *output channel*. It is also possible to specify non-quadratic receptive fields (by specifying 2-tuples instead of scalar parameter values).

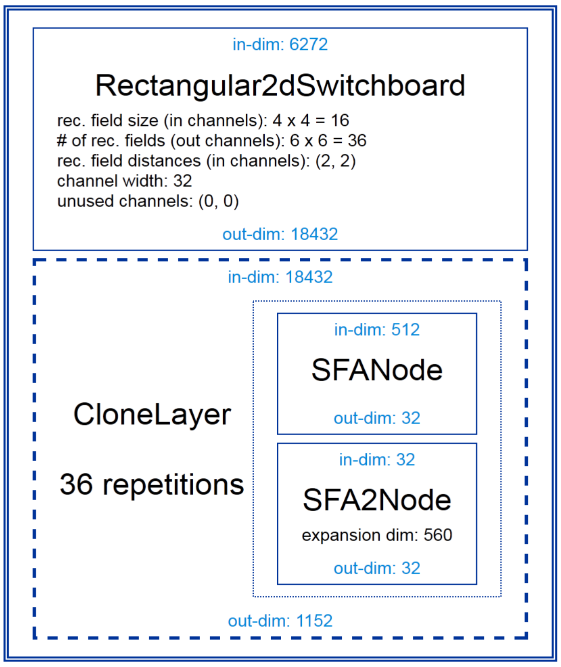


Figure 10.3: **Example of a network visualization.** This is the network created by the code in Figure 10.2. The image is taken from a browser screenshot.

10.2 Parallelization

Some of the algorithms implemented in MDP allow in principle a straightforward parallelization. For example, the PCA algorithm is based on computing the covariance matrix of the data. It is possible to split the task of computing the covariance matrix for a large data set in independent blocks by dividing the data into several subsets and recording the 2nd moment matrix, the mean, and the number of samples individually for each subset. From this data the covariance matrix of the whole data set can easily be reconstructed. The same principle applies to SFA, as discussed in Section 2.3.

The calculations for each block are independent from each other and can therefore be performed in parallel. Problems like this are called *embarrassingly parallel.* An embarrassingly parallel problem is one for which not much effort is needed to segment the problem into a large number of parallel tasks that can be executed independently without communication among tasks [Foster, 1995].

Parallelization is an increasingly important topic, since progress in the speed of a single CPU core has slowed down in recent years. Instead, Moore's law is now perpetuated by providing an increasing number of CPU cores [Sutter, 2005], which require parallelization to be used effectively. The `parallel` package in MDP adds functionality to parallelize the training and execution of MDP flows, thus providing a convenient speedup that can ideally scale with the number of CPU cores (minus the additional overhead).

In the last few years, there has also been a huge trend to use Graphics Processing

Units (GPU) for general numerical applications, due to their highly data-parallel number crunching capabilities (recent applications in computational neuroscience are [Pinto et al., 2009] and [Cireşan et al., 2011]). GPU support in Python is for example provided by PyCUDA [Klöckner et al., 2009] and Theano [Bergstra et al., 2010]. The parallelization approach discussed here is complementary to that approach and could in principle be used to distribute work across multiple GPUs (in a single machine or cluster).

10.2.1 Overview and Example

The `parallel` package in MDP is divided into two weakly coupled parts:

- The first part consists of *schedulers*. A scheduler takes arbitrary "tasks" and processes them in parallel (e.g., in multiple Python processes). A scheduler deals with the more technical aspects of parallelization, but does not need to know about nodes and flows, as it uses a different level of abstraction.
- The second part consists of parallel versions of the familiar MDP nodes and flows. In principle all MDP nodes support parallel execution, since copies of a node can be made and used in parallel. Parallelization of the training on the other hand depends on the specific algorithm. If a node class supports parallel training (like `SFANode` and `PCANode`) it is supposed to provide a `fork` and a `join` method. The `fork` method is used to create multiple instances ("copies") of the node for parallel training. When the training of these instances is done they are combined using the `join` method (e.g., for the `SFANode` this means that the covariance matrices are combined and the generalized eigenvector problem is solved). The forking and joining methods are provided via the extension mechanism of MDP, which is presented in Section 10.3 (how one can parallelize custom nodes will be also shown there).

  For convenient parallel execution or training of a flow the `ParallelFlow` class should be used instead of the normal `Flow`. This class takes complete care of managing the nodes, providing tasks to the scheduler, and combining the results.

In the following example a simple flow consisting of PCA and quadratic SFA is parallelized, so that multiple CPU cores are used:

```
node1 = mdp.nodes.PCANode(output_dim=10)
node2 = mdp.nodes.SFA2Node()
parallel_flow = mdp.parallel.ParallelFlow([node1, node2])
scheduler = mdp.parallel.ProcessScheduler()
parallel_flow.train(data_sources, scheduler=scheduler)
scheduler.shutdown()
```

Only two small changes are needed to parallelize the training of the flow: `ParallelFlow` is used instead of the normal `Flow` and a scheduler must be provided. The `ProcessScheduler` automatically creates as many Python processes as there are CPU cores. The parallel flow creates a training task for each data chunk and hands it over to the scheduler, which distributes them across the available worker processes. The results are then returned to the flow and combined. The `shutdown` method is called at the end to make sure that the resources used by the scheduler are released properly. It is advisable to use a try/finally block to make sure that `shutdown` is called even if an error occured:

```
scheduler = mdp.parallel.ProcessScheduler()
try:
    parallel_flow.train(data_sources, scheduler=scheduler)
finally:
    scheduler.shutdown()
```

The `Scheduler` class also supports the modern Python context manager interface. This enables the use of the `with` statement:

```
with mdp.parallel.ProcessScheduler() as scheduler:
    parallel_flow.train(data_sources, scheduler=scheduler)
```

10.2.2 Schedulers

The scheduler classes in MDP are derived from the `Scheduler` base class, which provides a simple unified interface for all kinds of different schedulers. This enables the `ParallelFlow` class to work with any scheduler, as long as it implements this interface. At the same time this makes it easy to write adapters for the numerous scheduler solutions that are readily available outside of MDP.

The standard scheduler in MDP is the `ProcessScheduler`, which distributes the incoming tasks over multiple Python processes. The use of processes instead of threads is often necessary due to a current limitation of the standard Python virtual machine (VM): it is not thread-safe and therefore a lock is used to prevent threads from running concurrently (the *global interpreter lock* or GIL). However, libraries like NumPy can release the GIL when running numerical code outside of the Python VM. Therefore it is possible to get a speedup even when using Python threads. MDP provides the `ThreadScheduler` class to take advantage of this, but the `ProcessScheduler` is still faster most of the time.

The real-world performance gain is highly dependent on the specific situation. Fortunately the structure of the hierarchical model of Section 3.2 is idealy suited for this, since the quadratic expansion of the data takes place in the worker processes. This means that the data transmission is relatively small compared to the data that arises in the training. For these types of networks the parallelization therefore scales very well with the number of CPU cores (in a real world test the speed-up factor was 4.2 on an Intel Core i7 920 processor with 4 physical / 8 logical cores using 8 processes).

In addition to the process based scheduler MDP has experimental support for the Parallel Python library[1]. This makes it possible to parallelize across multiple machines. A

[1] `http://www.parallelpython.com`

similar scheduler was written by Mathias Franzius, using SSH to turn remote computers into workers. With a few changes it was possible to make this scheduler compatible with the MDP scheduler interface (but it has not been integrated into MDP so far, since it is rather specialized). Originally this was very useful, since at that time the computers were much slower and typically had only a single core. Nowadays CPUs are sufficiently fast to train networks like those in [Franzius et al., 2011] in a couple of hours with the `ProcessScheduler`.

10.3 Node Extensions

The node extension mechanism addresses a problem that is quite common in software design, a problem well illustrated by the situation in the `parallel` package. In order to add the parallel training feature, the new `fork` and `join` methods must be made available. The straight forward solution would be to add these methods in the original class definitions, since they are class-depended and require access to the internal data (e.g., the covariance matrices). But parallelization is not the only node feature that might be added over time. For example the HTML-based visualization in `hinet` (see Section 10.1.4) is supposed create customized HTML code for different nodes. Again this is something that can be naturally solved by adding a new method to the node classes. Whenever a new feature is added this would require that the original code is modified, across many different modules. As more features are added the classes would grow larger, and the dependencies would pile up. This solution does not scale well and violates standard software design principles (e.g., the open-closed and the single-responsibility principles [Freeman et al., 2004]).

An alternative approach would be to add the new features through inheritance, deriving new node classes that implement the feature for the parent node class. This requires that the right class is picked when the node is instantiated, depending on the required feature. For example to use parallelization with the `SFANode`, instances of the `ParallelSFANode` class would have to be used. This quickly leads to problems when the use of multiple new features is required. Classes would have to be created for every combination of features that might be used (e.g., a `Parallel_HTML_SFANode` class). Therefore this approach scales poorly with the number of different features. All these issues motivated the node extension mechanism in MDP.

10.3.1 Node Extension Mechanism

The node extension mechanism makes it possible to add methods (or class attributes in general) to specific node classes during runtime, thereby adding new features. This enables the activation of extensions exactly when they are needed, reducing the risk of interference between different extensions. It is also possible to use multiple extensions at the same time, as long as there are no name collisions. Users are enabled to implement custom extensions, adding new functionality for MDP nodes without modifying any MDP code.

The node extension mechanism enables a mild form of Aspect-Oriented Programming (AOP) [Kickzales et al., 1997], in order to deal with the so-called *cross-cutting concerns*. In the AOP terminology the new features are *aspects*. The problem is that even though these aspects are typically very focused they do affect many node classes, spread all over the MDP code base. The term "cross-cutting" means that they alter a large amount of otherwise separated code (orthogonal to the organization of that code). Following the AOP terminology, the methods that are added contain *advice*. These ideas are not new in the Python world, and solutions that are similar to the MDP extension mechanism have been implemented before[2].

10.3.2 Extension Usage

Extensions in MDP are activated via the `activate_extension` function. For example activating the parallel extension is as simple as:

```
mdp.activate_extension("parallel")
# now the added attributes / methods are available
mdp.deactivate_extension("parallel")
# the additional attributes are no longer available
```

Activating an extension adds the available extension attributes to the supported nodes. Actually the explicit activation of the parallel extension is typically not needed, since this is handled by the `ParallelFlow` class. MDP also provides an extension context manager for the Python `with` statement:

```
with mdp.extension("parallel"):
    pass
```

This ensures that the activated extension is deactivated after the code block, even if there is an error. Finally there is also a function decorator to activate an extension for the time that the function is executed:

```
@mdp.with_extension("parallel")
def f():
    pass
```

Both the context manager and function decorator only deactivate those extensions that were not active when the context was entered, to prevent unintended side effects.

In addition to the mentioned `parallel` and `HTML` extensions there are a couple of other extensions available. The most recent additions in MDP are the `gradient` extension (to calculate the combined gradient for a flow) and the `cache_execute` memoization extension (to cache the node results for greater efficiency). Furthermore some highly specialized extensions have been written by other users and myself, but are not of general interest.

[2]See for example `http://www.cs.tut.fi/~ask/aspects` or `http://www.aspyct.org`.

10.3.3 Extending Nodes

Adding extension methods for a node class can be done in two ways: via multiple inheritance or via a function decorator. For the first approach one has to derive from both the extension base class and from the node class. For example the parallel extension of the SFA node is defined via the following class:

```
class ParallelSFANode(ParallelExtensionNode, mdp.nodes.SFANode):
    def _fork(self):
        # implement the forking for SFANode
        pass
    def _join(self):
        # implement the joining for SFANode
        pass
```

`ParallelExtensionNode` is the base class of the extension (every extension has one). The required methods or class attributes are defined like in a normal class. In principle one could even use the `ParallelSFANode` class like a normal class, but with the extension mechanism the `ParallelSFANode` class never has to be instantiated. Instead the methods defined here are automatically registered with the `SFANode` and the parallel extension (this is implemented with some metaclass magic).

Alternatively one can use the `extension_method` function decorator to create extension methods. The extension method can be defined like a normal function, with the function decorator added on top. This has the same effect as defining the method in an extension class, but this approach is more convenient when only a single extension method is defined. To create the `_fork` method for the `PCANode` one could write:

```
@mdp.extension_method("parallel", mdp.nodes.SFANode)
def _fork(self):
    pass
```

The first decorator argument is the name of the extension, the second is the class to which it belongs.

10.3.4 Creating Extensions

To create a new node extension one simply has to define a new extension base class. For the parallel extension this starts like:

```
class ParallelExtensionNode(mdp.ExtensionNode, mdp.Node):
    extension_name = "parallel"
    def _fork(self):
        raise NotImplementedError()
```

When defining a new extension one always has to define the `extension_name` attribute. This name can then be used to activate or deactivate the extension. When inheriting from `ExtensionNode`, a special metaclass is used for the class creation, which stores all the defined class attributes in a centralized extension dictionary, indexed by the extension name and the node class name. The `mdp.extension_method` decorator works in a similar way, storing a defined function in the same central extension dictionary. Whenever an extension is activated, the extension dictionary is accessed to add the stored extension attributes to the respective node classes at runtime.

Extensions can also override methods and attributes that are defined in a node class. The original attributes can still be accessed by prefixing the name with `"_non_extension_"`. On the other hand one extension is not allowed to override attributes that were defined by another active extension.

11 Bidirectional data flow with BiMDP

The MDP library as described so far executes algorithms in a strictly feed-forward manner. Data passes through the nodes in the order specified at the flow construction time. This has some limitations: for example, there is no built-in support for error back-propagation [Bryson and Ho, 1969; Rumelhart et al., 1986], which is needed for neural networks and other deep learning architectures [Bengio, 2009]. Input and output data in MDP is also restricted to 2D arrays, which can be inconvenient for some applications. BiMDP is a major extension of the MDP framework that addresses these limitations and has been included in MDP with version 2.6.

The `bimdp` package extends the feed-forward flow processing of MDP with the ability to transfer diverse data and execute nodes in arbitrary order. This makes it possible to use the MDP framework for a larger class of algorithms, like gradient descent algorithms or deep belief networks [Hinton et al., 2006] (see Section 8.3.3). The prefix “bi” stands for *bidirectional*, i.e., enabling data to travel both up and down a flow. It provides alternative, extended versions of the `Node` and `Flow` classes and of subpackages like `hinet`.

The development of BiMDP was motivated by the models and simulations described in Part II of this thesis. One primary design goal was to be as compatible with the standard MDP library as possible, including subpackages like `hinet` and `parallel`. This implied some restrictions, but also allowed us to retain the proven modular design of MDP. It would have been risky to change the core of MDP itself, since its stability and simplicity are important for a large number of users. With BiMDP we can provide users with a clear path when they require features beyond the scope of classical nodes and flows. Several example applications for BiMDP are provided on the MDP homepage, including gradient descent, a deep belief network, and error backpropagation in a multi layer perceptron.

11.1 Introduction

BiMDP is imported separately from MDP, with:

```
import bimdp
```

I now provide a brief summary of the most important features in BiMDP, some of which are also illustrated in Figure 11.1:

- Nodes can specify other nodes as jump targets, where the execution will be continued (similar to a goto statement). This is enabled by the new `BiFlow` class. Thereby it is possible to implement loops or backpropagation, in contrast to the

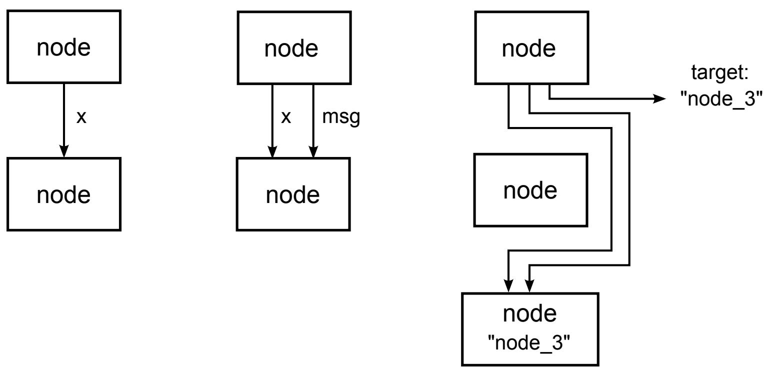


Figure 11.1: **Illustration of key BiMDP features.** The first diagram shows the standard MDP flow pattern, where only the data array `x` is propagated between nodes. In the second diagram the `msg` dictionary is transported alongside `x`, containing arbitrary data. Finally the `target` value `"node_3"` is given as well, causing a direct jump to the third node, based on its matching `node_id`.

strictly linear execution of a normal MDP flow. The new `BiNode` base class adds a `node_id` string attribute, which can be used as a label to target a node. The complexities of the data flow are evenly split up between `BiNode` and `BiFlow`: Nodes specify their data and target (using a standardized interface), which are then interpreted by the flow (like a primitive domain specific language).

- In addition to the standard array data, nodes can transport arbitrary data in a message dictionary (these are just standard Python dictionaries, i.e., `dict` instances). The new `BiNode` base class provides functionality to make this as convenient as possible, automatically extracting the data for each node.
- An interactive HTML-based inspection tool for flow training and execution is provided. This allows users to step through the flow node by node for graphical debugging or analysis purposes. Custom visualizations can be integrated as well (e.g. in the form of data plots for intermediate data).
- BiMDP supports and extends the `hinet` and the `parallel` packages from MDP. BiMDP in general is compatible with MDP, so one can insert standard MDP nodes in a `BiFlow`. It is also possible to use `BiNode` instances in a standard MDP flow, as long as they don not rely on BiMDP features.

The structure of BiMDP closely follows that of MDP, so there are submodules `bimdp.nodes`, `bimdp.parallel`, and `bimdp.hinet`. The module `bimdp.nodes` also contains `BiNode` versions of nearly all MDP nodes (which are automatically generated). For example there is a `bimdp.nodes.PCABiNode` class, which is derived from both `bimdp.BiNode` and `mdp.nodes.PCANode`.

11.2 Targets, id's and Messages

In a standard MDP node the return value of the `execute` method is restricted to a single two dimensional array. A `BiNode` on the other hand can optionally return a tuple containing an additional message dictionary and a target value. So in general the return value is a tuple `(x, msg, target)`, where `x` is the usual 2D data array. For convenience a `BiNode` is also allowed to return only the array `x` or a 2-tuple `(x, msg)`. This also implies that the return value of a standard MDP `Node` is compatible, allowing MDP nodes to be mixed with `BiNode` instances.

The message `msg` is a normal Python dictionary, containing any data that does not fit into the `x` data array. Nodes can take data from the message and add data to it. The message is propagated along with the `x` data. If a normal MDP node is contained in a `BiFlow` then the message is simply passed "around it". A `BiNode` on the other hand can freely decide how to interact with the message, and this is described in detail in Section 11.4.

The `target` value is either a string or a number. A number value gives the relative position of the target node in the flow, so a target value of 1 corresponds to the following node, while -1 is the previous node. The `BiNode` base class also allows the specification of a `node_id` string argument in the initialization. This string can then be used by other nodes as a target value.

The `node_id` string is also useful to access nodes in a `BiFlow` instance. The standard MDP `Flow` class already implements the Python container interface. For example `flow[2]` returns the third node in the flow. `BiFlow` now also enables the use of the `node_id` to index nodes, just like for a dictionary. For example:

```
pca_node = bimdp.nodes.PCABiNode(node_id="pca")
biflow = bimdp.BiFlow([pca_node])
biflow["pca"]  # returns the pca_node
```

11.3 BiFlow

The `BiFlow` class mostly behaves in the same way as the normal `Flow` class. Most of the new features were already mentioned, like support for targets, messages, and the retrieval of nodes based on their `node_id`. Apart from that the only major difference is the way in which one can provide additional arguments for nodes. For example the `FDANode` (for Fisher Discriminant Analysis) in MDP requires class labels in addition to the data array for supervised training, telling the node to which class each data point belongs. In the standard `Flow` class this additional training data is provided by the same iterable as the data (see Section 9.3.3). In a `BiFlow` this functionality is instead provided by the more general message mechanism. A new `msg_iterables` argument has been added in the `train` method to provide the message dictionary. This is demonstrated in the following example:

```
samples = np.random.random((100,10))
labels = np.arange(100)
flow = bimdp.BiFlow([mdp.nodes.PCANode(),
                     bimdp.nodes.FDABiNode()])
flow.train([[samples], [samples]],
           msg_iterables=[None, [{"cl": labels}]])
```

The message dictionary `{"cl": labels}` provides the class labels for the `FDABiNode`. In a normal `Flow` such additional arguments can only be given to the node that is currently in training. This limitation does not apply in a `BiFlow`, where the message can be accessed by all nodes (more on this in Section 11.4.1). Message iterators can be used during execution as well, via the `msg_iterable` argument in `BiFlow.execute`. At the end of the execution the resulting message is returned by `BiFlow.execute`. Therefore the return value has the form `(y, msg)`:

```
biflow = (bimdp.nodes.PCABiNode(output_dim=10) +
          bimdp.nodes.SFABiNode())
x = np.random.random((100,20))
biflow.train(x)
# include a message that is not used
y, msg = biflow.execute(x, msg_iterable={"test": 1})
# the returned msg has the value {"test": 1}
```

In this example only a single data chunk was used for simplicity. When iterables are provided for the execution then the `BiFlow` not only concatenates the `y` result arrays, but also tries to join the `msg` dictionaries into a single one. Arrays in the `msg` are concatenated and for all other types the plus operator is used.

The `train` method of `BiFlow` also has an additional argument called `stop_messages`, which can be used to provide messages for `stop_training` (see Section 11.4.1). The `execute` method on the other hand has the optional argument `target_iterable`, which can be used to specify the initial target at which the flow is entered.

11.4 BiNode

The `BiNode` class is an extension of the `Node` class and API that was described in Section 9.3. We start by describing the possible return values for the different `BiNode` methods:

- `execute` method:
  Return value can be of the form `x` or `(x, msg)` or `(x, msg, target)`. The execution of the flow continues, jumping to the target if that is specified.
- `train` method:
  - `None` terminates the training.

  - x or (x, msg) or (x, msg, target) means that the execution is continued and that this node will be reached again later to terminate the training.

- stop_training method:
  - None means that no further action is required (just like a normal MDP stop_training).
  - x or (x, msg) or (x, msg, target) starts an execution phase, which terminates when the end of the flow is reached or when the target value is the EXIT_TARGET constant (which will be explained later).

Naturally these methods also accept incoming messages: compared to Node methods they all have an additional msg argument. The target value on the other hand is only intended to be used by the encapsulating BiFlow.

As mentioned in the train description, training does not have to stop when the training node is reached. Instead it is possible to continue with the execution to return to the node later on (which is for example used for backpropagation). Similarly there are the stop_training result options that start an execution phase to propagate results from the completed node training or to prepare nodes for their upcoming training phase.

All these options taken together might seem confusing, but it is perfectly fine to ignore most of them for any given task. For example messages can be used productively without ever specifying target values. Finally there are two more additions in the BiNode API:

- node_id property:
  This is a read-only property, which returns the node id value (which by default is None). The __init__ method of a BiNode accepts a node_id argument to set this value.

- bi_reset method:
  This method is called by the containing BiFlow before and after training or execution, in order to reset possible internal state variables (this is relevant for stateful nodes as described in Section 11.4.2).

11.4.1 Extending BiNode and Message Handling

Developers are welcome to derive custom nodes from the BiNode base class. As for the MDP Node class they should not overwrite the public execute or train methods, but instead the private versions with an underscore in front. In addition to the dimensionality checks performed on x by the Node class this also provides developers with convenient message handling features.

The automatic message handling is a central feature of BiNode and relies on the dynamic nature of Python. In the FDABiNode example of Section 11.3 it is implicitly demonstrated how a value of a message is automatically passed to the _train method. This works because the key for the message value ("cl" for the class labels) is also the name of the _train keyword argument for the class label information. We now describe the mechanism behind this.

The public execute and train methods in BiNode not only accept a data array x but also a message dictionary msg. When given a message they perform introspection to determine the arguments of the corresponding private method like _train. If there is a matching key in the message for an argument like "cl" then the value from the msg dictionary is provided for this argument. Therefore the fact that FDABiNode._train has the signature (x, cl) is sufficient to automatically extract the "cl" value from the message. The message itself remains unchanged because the "cl" key stays in msg.

Private methods like _execute or _train have the same return options as their public BiNode methods, so, for example, one can return a tuple (x, msg). The msg in the return value from _execute is then used by execute to update the original msg. With this mechanism _execute can overwrite or add new values to the message.

There are also several additional features to make the message handling more powerful and convenient. For example, a message key "method" can be used to direct execute to call a method with the specified name instead of the standard _execute. This is used by the gradient extension (which was briefly mentioned in Section 10.3) to divert node execution to a special _gradient method. All these special message features are covered in the MDP online documentation.

11.4.2 Coroutines for Stateful Nodes

Sometimes it is required to visit a node multiple times during a single flow execution (so this should not be confused with multiple training phases), requiring a node that keeps track of the current "position" in the algorithm. For example a deep belief network typically requires multiple up and down passes. The standard solution for this is to implement a state machine in the node, which is repetitive and obfuscates the actual algorithm. Fortunately Python does support *continuations* via so called *coroutines*. A coroutine is basically a function that can be entered and exited multiple times, continuing execution at the previous position in the code. Coroutines are most popular in the context of concurrency, but they are generally useful as a light-weight replacement for state-machines.

Using a coroutine in a BiNode method to maintain a state would still require tedious boilerplate code. Therefore BiMDP provides a function decorator to perform the required steps automatically, making coroutines highly convenient in this situation. This has been used for the implementation of the top-down optimization in Chapter 8. The coroutine decorator can also be found in both the gradient descent and the deep belief network example on the MDP homepage.

11.5 Inspection

Using jumps and messages can result in complex data flows. BiMDP includes a graphical inspection tool to help with debugging and analyzing such a flow. This tool is based on the static HTML view from the mdp.hinet module, described in Section 10.3. Instead of a static view of the flow it creates an animated slideshow of the flow training or execution. The execution inspection can be performed with

```
bimdp.show_execution(flow, x, ...)
```

instead of the normal `flow.execute(x, ...)`. This also opens the inspection automatically in the webbrowser. The same functionality is available for training:

```
bimdp.show_training(flow, data_iterables, ...)
```

This performs the training of the flow and creates a single inspection page for all training phases. Figure 11.2 shows an example of such an inspection.

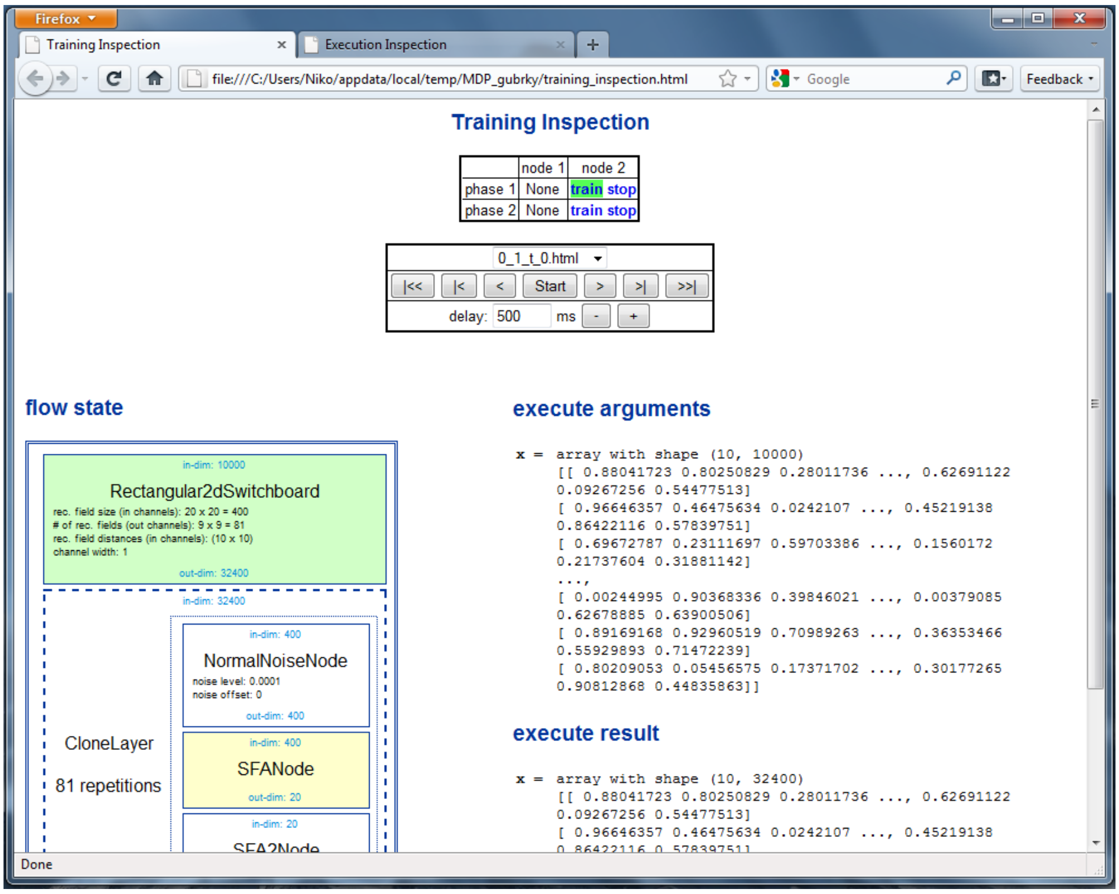
The BiMDP inspection is also useful to visualize the data processing that is happening inside a flow. This is especially handy for building or understanding new algorithms. Therefore it was one objective to allow easy customization of the HTML views in the inspection. One simple example is provided on the MDP homepage, where a plot of the node result is added to the HTML data presentation. In addition one can bypass the `bimdp.show_training` and `bimdp.show_execution` helper functions if even more flexibility is required.

Technically the inspection only consists of static HTML files that are animated with JavaScript. Therefore an inspection can be viewed at any time by opening the main HTML file in a browser. One advantage of this is that inspections can be stored and integrated into the simulation management workflow. This way they can document how the scientific results were achieved and ensure reproducibility. The inspection tool was also critical for debugging the complex top-down flow patterns from Chapter 8.

11.6 Hierarchical Networks and Parallelization

BiMDP is mostly compatible with the hierarchical networks in `mdp.hinet` (see Section 10.1). For the full BiMDP functionality in such a network it is required to use the BiMDP version `bimdp.hinet`. This module contains a `BiFlowNode` that offers the same functionality as the `FlowNode`, but with the added capability of handling messages, targets, and other BiMDP concepts. Similarly there is a new `BiSwitchboard` base class and a BiMDP compatible layer class. All the required building blocks for hierarchical networks are therefore present.

The `mdp.parallel` module has been adapted for BiMDP as well, in the form of the `bimdp.parallel` module. It contains a `ParallelBiFlow` class which can be used like the normal `ParallelFlow`. No changes to schedulers are required, so all the scheduler classes in `mdp.parallel` can be used.



Training Inspection

	node 1	node 2
phase 1	None	train stop
phase 2	None	train stop

flow state

in-dim: 10000
Rectangular2dSwitchboard
rec. field size (in channels): 20 x 20 = 400
# of rec. fields (out channels): 9 x 9 = 81
rec. field distances (in channels): (10 x 10)
channel width: 1
out-dim: 32400

in-dim: 32400
CloneLayer
81 repetitions

in-dim: 400
NormalNoiseNode
noise level: 0.0001
noise offset: 0
out-dim: 400

in-dim: 400
SFANode
out-dim: 20

in-dim: 20
SFA2Node

execute arguments

```
x =  array with shape (10, 10000)
     [[ 0.88041723 0.80250829 0.28011736 ..., 0.62691122
     0.09267256 0.54477513]
     [ 0.96646357 0.46475634 0.0242107 ..., 0.45219138
     0.86422116 0.57839751]
     [ 0.69672787 0.23111697 0.59703386 ..., 0.1560172
     0.21737604 0.31881142]
     ...,
     [ 0.00244995 0.90368336 0.39846021 ..., 0.00379085
     0.62678885 0.63900506]
     [ 0.89169168 0.92960519 0.70989263 ..., 0.36353466
     0.55929893 0.71472239]
     [ 0.80209053 0.05456575 0.17371702 ..., 0.30177265
     0.90812868 0.44835863]]
```

execute result

```
x =  array with shape (10, 32400)
     [[ 0.88041723 0.80250829 0.28011736 ..., 0.62691122
     0.09267256 0.54477513]
     [ 0.96646357 0.46475634 0.0242107 ..., 0.45219138
     0.86422116 0.57839751]
```

Figure 11.2: **Example view for BiMDP training inspection.** This figure shows the browser window with the inspection. On top are controls for the user to select a training phase. Below that are controls to step through the flow training, node by node. The lower part of the view contains the visualisation of the current flow state, consisting of a graphical view on the left and a display of the current node input and output values on the right. The current node in the flow is highlighted in green, while the node currently in training is colored yellow. Clicking on a node makes the inspection jump forward to the step in which that node is reached.

12 Conclusion

The slowness principle is an elegant model for unsupervised learning in the brain. Its realization in the form of Slow Feature Analysis (SFA) provides an analytical framework and an efficient numerical implementation for computer simulations. The simplicity of SFA is comparable to the established method of Principal Component Analysis (PCA) and could therefore become a standard method for data processing (when data has an appropriate time structure). For biologically inspired models it is especially valuable that, unlike PCA, SFA works very well in a hierarchical network structure. Since SFA is still a relatively young method its capabilities are not as well explored as those of older methods like, for example, Independent Component Analysis (ICA). For slowness learning in general this is also true to some extend, even though the slowness principle has been used before the discovery of SFA. This thesis had the goal of thoroughly evaluating the performance of hierarchical SFA for the biologically relevant task of invariant object recognition. Furthermore we have tried to explore how this model can be extended for top-down processes, which would be an important next step.

In the first part of this thesis we have shown that hierarchical SFA indeed provides a powerful model for invariant object recognition. We hope that this can help to close the performance gap between biologically inspired models and purely artificial methods from computer science. Chapter 3 was primarily aimed at measuring different aspects of the performance of our model, using supervised post-processing. Chapter 5 on the other hand demonstrated that this performance is also achievable in combination with methods that are closer to the neural substrate of the brain, like in this case reinforcement learning.

The extension of our model with top-down processes in Part II lead to mixed results. Capturing the input distributions in Chapter 7 worked reasonably well, based on the standard method of vector quantization with competitive learning. The top-down reconstruction of input stimuli in Chapter 8 did work to some extend, but only thanks to the centroid information from the previous chapter. Applying gradient descent on the other hand turned out to be an almost complete failure, which is a problem for the scalability and further development of top-down processes in this model. This suggests that more sophisticated methods might be needed, probably in combination with fundamental changes in the model (e.g., a probabilistic formulation of SFA or a different kind of nonlinear expansion).

The importance of computational methods in neuroscience has been growing steadily. This is partly a result of the ever increasing availability of cheap computational resources, which enables work like that presented in this thesis. The increasing complexity of computational methods also implies that good software engineering is becoming increasingly relevant. This is also connected to the issue of reproducibility of scientific results. Integrating the central parts of our model into the MDP open source library enables others to easily verify and adapt the presented methods. Cooperation in the scientific computing ecosystem and the reuse of software ideally makes it easier for researchers to concentrate on the innovative aspects of their work.

Acknowledgements

First of all, I would like to thank Laurenz Wiskott for his continued and extensive support. I very much enjoyed the constructive scientific discussions with him, and his feedback was always very valuable for me. His scientific thoroughness has left a positive mark on me.

I am indebted to Mathias Franzius and Robert Legenstein. Not only did I enjoy our collaboration, I also learned from both of them. This is actually true for all the members of the group, especially Henning Sprekeler, Susanne Lezius, Sven Dähne, Christian Hinze, and Minh Ha Quang. I would also like to thank the group of Stephan Brandt, especially Sein Schmidt, for interesting insights into the experimental and medical aspects of neuroscience. One highlight for me was the cooperation with the other MDP developers. I am especially thankful to the founders of MDP, Tiziani Zito and Pietro Berkes.

Finally, I would like to thank everyone at the ITB and the BCCN for creating such a friendly and productive atmosphere. Of course, this would not be possible without the support from the administrative staff, especially Margreth Franke, Elvira Lauterbach, Karin Winklhöfer, and Andreas Hantschmann.

Bibliography

A. C. Antoulas and D. C. Sorensen. Approximation of large-scale dynamical systems: An overview. *Int J Appl Math Comp*, 11(5):1093–1121, 2001.

M. S. Bartlett and T. J. Sejnowski. Learning viewpoint-invariant face representations from visual experience in an attractor network. *Network: Computation in neural systems*, 9(3):399–417, 1998.

S. Becker. Implicit learning in 3d object recognition: The importance of temporal context. *Neural Comput*, 11(2):347–374, 1999.

S. Becker and G. E. Hinton. Self-organizing neural network that discovers surfaces in random-dot stereograms. *Nature*, 355(6356):161–163, 1992.

Y. Bengio. *Learning deep architectures for AI.* Now Publishers Inc, 2009. ISBN 1601982941.

J. Bergstra, F. Bastien, J. Turian, R. Pascanu, O. Delalleau, O. Breuleux, P. Lamblin, G. Desjardins, D. Erhan, and Y. Bengio. Deep Learning on GPUs with Theano, 2010.

P. Berkes. Handwritten digit recognition with nonlinear fisher discriminant analysis. *Proceedings of ICANN 2005*, 2(LNCS 3696):285–287, 2005.

P. Berkes and L. Wiskott. Slow feature analysis yields a rich repertoire of complex cell properties. *J Vision*, 5(6):579–602, 2005.

P. Berkes and L. Wiskott. On the analysis and interpretation of inhomogeneous quadratic forms as receptive fields. *Neural computation*, 18(8):1868–1895, 2006. ISSN 0899-7667.

G. S. Berns and T. J. Sejnowski. A computational model of how the basal ganglia produces sequences. *J Cognitive Neurosci*, 10:108–121, 1998.

D. P. Bertsekas and J.N. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, 1996.

A. E. Bryson and Yu-Chi Ho. *Applied optimal control: optimization, estimation, and control.* Blaisdell Publishing Company, 1969.

J. Bullier. What is fed back? In J. L. van Hemmen and T. J. Sejnowski, editors, *23 Problems in Systems Neuroscience.* Oxford University Press, New York, 2006. ISBN 978-0-19-514822-0.

Chih-Chung Chang and Chih-Jen Lin. *LIBSVM: a library for support vector machines*, 2001. `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

D. B. Chklovskii and A. A. Koulakov. A wire length minimization approach to ocular dominance patterns in mammalian visual cortex. *Physica A*, 284(1–4):318–334, 2000.

D. C. Cireşan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber. High-Performance Neural Networks for Visual Object Classification. *Arxiv preprint arXiv:1102.0183*, 2011.

J. Demšar, B. Zupan, G. Leban, and T. Curk. Orange: From experimental machine learning to interactive data mining. *Knowledge Discovery in Databases: PKDD 2004*, pages 537–539, 2004.

J. J. DiCarlo and D. Cox. Untangling invariant object recognition. *Trends in Cognitive Science*, 11:333–341, 2007.

S. Dura-Bernal, T. Wennekers, and S.L. Denham. The role of feedback in a hierarchical model of object perception. *BICS 2010 - Brain Inspired Cognitive Systems*, 2010.

W. Einhäuser, J. Hipp, J. Eggert, E. Körner, and P. König. Learning viewpoint invariant object representations using a temporal coherence principle. *Biol Cybern*, 93:79–90, 2005.

A. K. Engel, P. Fries, and W. Singer. Dynamic predictions: oscillations and synchrony in top–down processing. *Nature Reviews Neuroscience*, 2(10):704–716, 2001.

D. Erhan, A. Courville, and Y. Bengio. Understanding Representations Learned in Deep Architectures. *Technical Report*, 2010. URL `http://www.dumitru.ca/files/publications/invariances_techreport.pdf`.

R. Fletcher. Practical Methods of Optimization, vol. 1 and 2, 1980.

P. Földiák. Learning invariance from transformation sequences. *Neural Comput*, 3(2): 194–200, 1991.

D. J. Foster, R. G. M. Morris, and P. Dayan. Models of hippocampally dependent navigation, using the temporal difference learning rule. *Hippocampus*, 10:1–16, 2000.

I. Foster. *Designing and building parallel programs.* Addison-Wesley, 1995.

M. Franzius, H. Sprekeler, and L. Wiskott. Slowness and sparseness lead to place, head-direction and spatial-view cells. *Public Library of Science (PLoS) Computational Biology*, 3(8):e166, 2007.

M. Franzius, N. Wilbert, and L. Wiskott. Invariant object recognition with slow feature analysis. In Vera Kurková, Roman Neruda, and Jan Koutník, editors, *Proc. 18th Intl. Conf. on Artificial Neural Networks, ICANN'08, Prague*, volume 5163 of *Lecture Notes in Computer Science*, pages 961–970. Springer, September 2008. ISBN 978-3-540-87535-2.

M. Franzius, N. Wilbert, and L. Wiskott. Invariant object recognition and pose estimation with slow feature analysis. *Neural Comput*, 2011. accepted for publication.

E. Freeman, E. Freeman, B. Bates, and K. Sierra. *Head first design patterns.* O'Reilly & Associates, Inc., 2004.

Y. Freund, S. Dasgupta, M. Kabra, and N. Verma. Learning the structure of manifolds using random projections. *Advances in Neural Information Processing Systems*, 20, 2007.

B. Fritzke. A growing neural gas network learns topologies. In *Advances in Neural Information Processing Systems 7*, pages 625–632. MIT Press, 1995.

K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biol Cybern*, 36(4):193–202, 1980.

K. Fukushima. Restoring partly occluded patterns: a neural network model. *Neural Networks*, 18(1):33–43, 2005.

S. Garcia and N. Fourcaud-Trocmé. OpenElectrophy: an electrophysiological data-and analysis-sharing framework. *Frontiers in Neuroinformatics*, 3, 2009.

D. George and J. Hawkins. Towards a mathematical theory of cortical micro-circuits. *PLoS Comput Biol*, 5(10):e1000532, 10 2009.

A. Graves, M. Liwicki, S. Fernández, R. Bertolami, H. Bunke, and J. Schmidhuber. A novel connectionist system for unconstrained handwriting recognition. *IEEE transactions on pattern analysis and machine intelligence*, pages 855–868, 2008.

R. Gray. Vector quantization. *IEEE Assp Magazine*, 1(2):4–29, 1984.

K. Grill-Spector, T. Kushnir, T. Hendler, S. Edelman, Y. Itzchak, and R. Malach. A sequence of object-processing stages revealed by fMRI in human occipital lobe. *Human Brain Mapping*, 6:316–328, 1998.

C. G. Gross. Genealogy of the "grandmother cell". *Neuroscientist*, 8(5):512–518, 2002.

S. Grossberg. Adaptive pattern classification and universal recoding: I. Parallel development and coding of neural feature detectors. *Biological cybernetics*, 23(3):121–134, 1976.

M. Hanke, Y. O. Halchenko, P. B. Sederberg, E. Olivetti, I. Fründ, J. W. Rieger, C. S. Herrmann, J. V. Haxby, S. J. Hanson, and S. Pollmann. PyMVPA: a unifying approach to the analysis of neuroscientific data. *Frontiers in neuroinformatics*, 3, 2009.

W. Hashimoto. Quadratic forms in natural images. *Network-Comp Neural*, 14(4):765–788, 2003.

G. E. Hinton. Connectionist learning procedures. *Artif Intell*, 40(1-3):185–234, 1989.

G. E. Hinton, S. Osindero, and Y. W. Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

C. Hinze, N. Wilbert, and L. Wiskott. Visualization of higher-level receptive fields in a hierarchical model of the visual system. *BMC Neuroscience*, 10(Suppl 1):P158, 2009. URL `http://www.biomedcentral.com/1471-2202/10/S1/P158`.

J. Hipp, W. Einhäuser, J. Conradt, and P. König. Learning of somatosensory representations for texture discrimination using a temporal coherence principle. *Network-Comp Neural*, 16(2/3):223–238, 2005.

J. C. Houk, J. L. Adams, and A. G. Barto. A model of how the basal ganglia generate and use neural signals that predict reinforcement. In *Models of information processing in the basal ganglia*, pages 249–270. MIT Press, Cambridge, MA, 1995.

M. D. Humphries, K. Gurney, and T. J. Prescott. Is there a brainstem substrate for action selection? *Philosophical Transactions of the Royal Society B: Biological Sciences*, 362(1485):1627, 2007.

C. P. Hung, G. Kreiman, T. Poggio, and J. J. DiCarlo. Fast readout of object identity from macaque inferior temporal cortex. *Science*, 310(5749):863–866, 2005.

J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, pages 90–95, 2007.

A. Hyvärinen. Fast and robust fixed-point algorithms for independent component analysis. *IEEE Transactions on Neural Networks*, 10:626–634, 1999.

E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001. URL `http://www.scipy.org/`.

G. Kanizsa. Quasi-perceptual margins in homogeneously stimulated fields. *The Perception of Illusory Contours, Petry S. & Meyer, GE (eds) Springer Verlag, New York*, pages 40–49, 1987.

C. Kayser, W. Einhäuser, O. Dümmer, P. König, and K. Körding. Extracting slow subspaces from natural videos leads to complex cells. *Artificial Neural Networks - ICANN 2001 Proceedings*, pages 1075–1080, 2001.

H. Kennedy, P. Barone, and A. Falchier. Relative contributions of feedforward and feedback inputs to individual areas. *Eur J Neurosc*, 12(Suppl 11):489, 2000.

D. Kersten, P. Mamassian, and A. Yuille. Object perception as Bayesian inference. *Psychology*, 55(1):271, 2004.

G. Kickzales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. *Proceedings of ECOOP, IEEE, Finland*, pages 220–242, 1997.

A. Klöckner, N. Pinto, Yunsup Lee, B. C. Catanzaro, P. Ivanov, and A. Fasih. Pycuda: Gpu run-time code generation for high-performance computing. *CoRR*, abs/0911.3456, 2009.

T. Kohonen. *Self-organization and associative memories.* Springer Verlag, 1984.

T. S. Lee and D. Mumford. Hierarchical Bayesian inference in the visual cortex. *JOSA A*, 20(7):1434–1448, 2003.

R. Legenstein, N. Wilbert, and L. Wiskott. Reinforcement Learning on Slow Features of High-Dimensional Input Streams. *PLoS Comput Biol*, 6(8):e1000894, 2010.

R. A. Legenstein and W. Maass. Wire length as a circuit complexity measure. *J Comput Syst Sci*, 70:53–72, 2005.

S. Lehky, X. Peng, C. McAdams, and A. Sereno. Spatial modulation of primate inferotemporal responses by eye position. *PLoS ONE*, 3(10):e3492, 10 2008.

N. Li and J. J. DiCarlo. Unsupervised natural experience rapidly alters invariant object representation in visual cortex. *Science*, 321:1502–1507, 2008.

N. Li and J. J. DiCarlo. Unsupervised natural visual experience rapidly reshapes size-invariant object representation in inferior temporal cortex. *Neuron*, 67(6):1062–1075, 2010.

Y. Linde, A. Buzo, and R. Gray. An algorithm for vector quantizer design. *IEEE Transactions on communications*, 28(1):84–95, 1980.

S. P. Lloyd. Last square quantization in PCMs. Technical Note (1957), Bell Laboratories. *IEEE Trans. Inform. Theory*, 28:129–137, 1982.

A. Losonczy, J. K. Makara, and J. C. Magee. Compartmentalized dendritic plasticity and input feature storage in neurons. *Nature*, 452(7186):436–441, 2008.

B. W. Mel. SEEMORE: Combining color, shape and texture histogramming in a neurally-inspired approach to visual object recognition. *Neural Comput*, 9(4):777–804, 1997.

R. Memisevic and G. E. Hinton. Learning to represent spatial transformations with factored higher-order boltzmann machines. *Neural Comput*, 22(6):1473–1492, 2010.

G. Mitchison. Removing time variation with the anti-Hebbian differential synapse. *Neural Comput*, 3:312–320, 1991.

Y. Miyashita. Neuronal correlate of visual associative long-term memory in the primate temporal cortex. *Nature*, 335(6193):817–820, 1988.

P. R. Montague, P. Dayan, and Sejnowski T. J. A framework for mesencephalic dopamine systems based on predictive Hebbian learning. *J Neurosci*, 16:1936–1947, 1996.

J. Moody and C. J. Darken. Fast learning in networks of locally-tuned processing units. *Neural Comput*, 1(2):281–294, 1989.

A. Moore and C. Atkeson. The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Mach Learn*, 21, 1995.

R. G. Morris, P. Garrud, J. N. Rawlins, and J. O'Keefe. Place navigation impaired in rats with hippocampal lesions. *Nature*, 297(5868):681–683, 1982.

R. Munos and A. Moore. Variable resolution discretization in optimal control. *Mach Learn*, 49(2-3):291–323, 2002.

S. K. Nayar, S. A. Nene, and H. Murase. Real-time 100 object recognition system. In *Proc. of ARPA Image Understanding Workshop*, Palm Springs, 1996.

L. G. Nowak and J. Bullier. The timing of information transfer in the visual system. *Cerebral cortex: Extrastriate cortex in primate*, 12, 1997.

B. A. Olshausen and D. J. Field. Natural image statistics and efficient coding. *Network-Comp Neural*, 7:333–339, 1996.

N. Pinto, D. Doukhan, J. J. DiCarlo, and D. D. Cox. A high-throughput screening approach to discovering good forms of biologically inspired visual representation. *PloS computational biology*, 5(11):e1000579, 2009.

P. Poirazi, T. Brannon, and B. W. Mel. Pyramidal neuron as two-layer neural network. *Neuron*, 37(6):989–999, March 2003.

W. Potjans, A. Morrison, and M. Diesmann. A spiking neural network model of an actor-critic learning agent. *Neural Comp.*, 21:1–39, 2009.

R. Rao, G. A. Cecchi, C. C. Peck, and J. R. Kozloski. Unsupervised segmentation with dynamical units. *Neural Networks, IEEE Transactions on*, 19(1):168–182, 2008.

M. Riesenhuber and T. Poggio. Hierarchical models of object recognition in cortex. *Nat Neurosci*, 2:1019–1025, 1999.

R. Rojas. *Neural networks: a systematic introduction*. Springer, 1996. ISBN 3540605053.

E. T. Rolls. Neurophysiological mechanisms underlying face processing within and beyond the temporal cortical visual areas. *Philosophical Transactions of the Royal Society*, 335:11–21, 1992.

E. T. Rolls and G. Deco. *Computational Neuroscience of Vision*. Oxford University Press, New York, 2002.

E. T. Rolls and S. M. Stringer. Invariant visual object recognition: A model, with lighting invariance. *Journal of Physiology - Paris*, 100:43–62, 2006.

E. T. Rolls, M. J. Tovee, D. G. Purcell, A. L. Stewart, and P. Azzopardi. The responses of neurons in the temporal cortex of primates, and face identification and detection. *Experimental Brain Research*, 101(3):473–484, 1994.

S. T. Roweis and L. K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323, 2000.

D. E. Rumelhart and J. L. McClelland. *Parallel distributed processing: explorations in the microstructure of cognition. Volume 1. Foundations.* MIT Press, Cambridge, Ma, 1986.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

T. Schaul, J. Bayer, D. Wierstra, Y. Sun, M. Felder, F. Sehnke, T. Rückstieß, and J. Schmidhuber. PyBrain. *Journal of Machine Learning Research*, 2010.

W. Schultz. Predictive reward signal of dopamine neurons. *J. Neurophys.*, 80:1–27, 1998.

W. Schultz, P. Dayan, and P. R. Montague. A neural substrate of prediction and reward. *Science*, 275:1593–9, 1997.

D. S. Seljebotn. Fast numerical computations with Cython. In *Proceedings of the 8th Python in Science Conference*, 2009.

H. D. Seung. Learning in spiking neural networks by reinforcement of stochastic synaptic transmission. *Neuron*, 40:1063–1073, 2003.

D. Sheynikhovich, R. Chavarriaga, T. Strösslin, and W. Gerstner. Spatial Representation and Navigation in a Bio-inspired Robot. In *Biomimetic Neural Learning for Intelligent Robots: Intelligent Systems, Cognitive Robotics, and Neuroscience*, pages 245–264, 2005.

W. Singer and C. M. Gray. Visual feature integration and the temporal correlation hypothesis. *Annual review of neuroscience*, 18(1):555–586, 1995.

W. Softky. Learning to make specific predictions using slow feature analysis. *unpublished presentation*, 2005. unpublished presentation.

V. A. Solé, E. Papillon, M. Cotte, P. Walter, and J. Susini. A multiplatform code for the analysis of energy-dispersive X-ray fluorescence spectra. *Spectrochimica Acta Part B: Atomic Spectroscopy*, 62(1):63–68, 2007.

S. Sonnenburg, G. Ratsch, and F. De Bona. The SHOGUN Machine Learning Toolbox. *Journal of Machine Learning Research*, 11:1799–1802, 2010.

H. Sprekeler and L. Wiskott. A theory of slow feature analysis for transformation-based input signals with an application to complex cells. *Neural Comput*, pages 1–33, 2010.

H. Sprekeler, C. Michaelis, and L. Wiskott. Slowness: An objective for spike-timing-plasticity? *PLoS Comput Biol*, 3(6):e112, 2007.

J. V. Stone and A. Bray. A learning rule for extracting spatio-temporal invariances. *Network-Comp Neural*, 6:429–436, 1995.

G. Strang. Linear algebra and its applications. *Saunders, Philadelphia, PA*, 1988.

S. M. Stringer and E. T. Rolls. Invariant object recognition in the visual system with novel views of 3D objects. *Neural Comput*, 14:2585–2596, 2002.

S. M. Stringer, G. Perry, E. T. Rolls, and J. H. Proske. Learning invariant object recognition in the visul system with continuous transformations. *Biological Cbernetics*, 94:128–142, 2006.

H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):16–20, 2005.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1998.

K. Swersky, B. Chen, B. Marlin, and N. de Freitas. A tutorial on stochastic approximation algorithms for training Restricted Boltzmann Machines and Deep Belief Nets. In *Information Theory and Applications Workshop (ITA), 2010*, pages 1–10. IEEE, 2010.

J. B. Tenenbaum, V. de Silva, and J. C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290:2319–2323, 2000.

E. L. Thorndike. *Animal Intelligence.* Hafner, Darien, 1911.

Toucan Corporation. Toucan virtual museum. `http://toucan.web.infoseek.co.jp/3DCG/3ds/FishModelsE.html`, 2005.

T. W. Troyer and A. J. Doupe. An associational model of birdsong sensorimotor learning ii. temporal hierarchies and the learning of song sequence. *J Neurophysiol*, 84(3):1224–1239, September 2000.

R. Turner and M. Sahani. A maximum-likelihood interpretation for slow feature analysis. *Neural Comput*, 19(4):1022–1038, 2007.

L. G. Ungerleider and M. Mishkin. *Two Cortical Visual Systems*, chapter 18, pages 549–586. BMJ Publishing Group Ltd., 1982.

R. Urbanczik and W. Senn. Reinforcement learning in populations of spiking neurons. *Nat Neurosci*, 12(3):250–252, 2009. URL `urbanczik_senn_2009a.pdf`.

E. Vasilaki, N. Frémaux, R. Urbanczik, W. Senn, and W. Gerstner. Spike-based reinforcement learning in continuous state and action space: When policy gradient methods fail. *PLoS Comput Biol*, 5(12):e1000586, 2009.

R. Vogels. Effect of image scrambling on inferior temporal cortical responses. *NeuroReport*, 10:1811–1816, 1999.

C. von der Malsburg. The correlation theory of brain function. *Models of Neural Networks II: Temporal Aspects of Coding and Information Processing in Biological Systems*, pages 95–119, 1994.

G. Wallis and E. T. Rolls. Invariant face and object recognition in the visual system. *Prog Neurobiol*, 51(2):167–194, 1997.

H. Wersing and E. Körner. Learning optimized features for hierarchical models of object recognition. *Neural Comput*, 15(7):1559–1588, 2003.

N. Wilbert and L. Wiskott. Hierarchical slow feature analysis and top-down processes, 2010.

N. Wilbert, T. Zito, R. B. Schuppner, Z. J. Szmek, L. Wiskott, and P. Berkes. Building extensible frameworks for data processing: the case of MDP, Modular Toolkit for Data Processing. *under review for publication in the Journal of Computational Science*, 2011.

L. Wiskott. Learning invariance manifolds. In L. Niklasson, M. Bodén, and T. Ziemke, editors, *Proceedings of the 8th International Conference on Artificial Neural Networks, ICANN'98, Skövde*, Perspectives in Neural Computing, pages 555–560, London, 1998. Springer. ISBN 3-540-76263-9.

L. Wiskott. Slow feature analysis: A theoretical analysis of optimal free responses. *Neural Comput*, 15(9):2147–2177, 2003.

L. Wiskott. How does our visual system achieve shift and size invariance? In J. L. van Hemmen and T. J. Sejnowski, editors, *23 Problems in Systems Neuroscience*, chapter 16, pages 322–340. Oxford University Press, New York, 2006.

L. Wiskott and T. Sejnowski. Slow feature analysis: Unsupervised learning of invariances. *Neural Comput*, 14(4):715–770, 2002.

R. Wyss, P. König, and P. F. M. J. Verschure. A Model of the Ventral Visual System Based on Temporal Stability and Local Memory. *PLoS Biol*, 4(5), 2006.

X. Xie and H. S. Seung. Learning in neural networks by reinforcement of irregular spiking. *Phys Rev E*, 69(041909), 2004.

T. Zito, N. Wilbert, L. Wiskott, and P. Berkes. Modular toolkit for Data Processing (MDP): a Python data processing framework. *Front Neuroinf*, 2, 2008.

List of Figures

List of Tables

Selbständigkeitserklärung

Hiermit erkläre ich, die Dissertation selbstständig und nur unter Verwendung der angegebenen Hilfen und Hilfsmittel angefertigt zu haben.

Ich habe mich anderwärts nicht um einen Doktorgrad beworben und besitze einen entsprechenden Doktorgrad nicht.

Ich erkläre die Kenntnisnahme der dem Verfahren zugrunde liegenden Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät I der Humboldt-Universität zu Berlin vom 01. September 2005.

München, den 06.05.2011

Niko Wilbert

Veröffentlichungsliste

Artikel in Fachzeitschriften

- N. Wilbert, T. Zito, R. B. Schuppner, Z. J. Szmek, L. Wiskott, P. Berkes (2011).
  Building extensible frameworks for data processing: the case of MDP, Modular Toolkit for Data Processing.
  Journal of Computational Science, doi: 10.1016/j.jocs.2011.10.005.

- M. Franzius, N. Wilbert, L. Wiskott (2011).
  Invariant Object Recognition and Pose Estimation with Slow Feature Analysis.
  Neural Computation, doi:10.1162/NECO_a_00171.

- R. Legenstein, N. Wilbert, L. Wiskott (2010).
  Reinforcement Learning on Slow Features of High-Dimensional Input Streams.
  PLoS Computional Biology, doi:10.1371/journal.pcbi.1000894

- T. Zito, N. Wilbert, L. Wiskott, P. Berkes (2008).
  Modular Toolkit for Data Processing (MDP): A Python Data Processing Framework.
  Frontiers in Neuroinformatics, doi: 10.3389/neuro.11.008.2008.

Konferenzbeiträge

Proceedings

- M. Franzius, N. Wilbert, L. Wiskott (2008).
  Invariant Object Recognition with Slow Feature Analysis
  Proc. 18th Int'l Conf. on Artificial Neural Networks, ICANN'08, Prague, September 3-6, eds. Vera Kurková, Roman Neruda, and Jan Koutník, publ. Springer-Verlag, pp. 961-970.

Posters / Abstracts

- N. Wilbert, L. Wiskott (2010).
  Hierarchical Slow Feature Analysis and Top-Down Processes.
  Bernstein Conference on Computational Neuroscience, TU Berlin, 2010.

- S. Dähne, N. Wilbert, L. Wiskott (2010).
  Self-organization of V1 Complex Cells Based On Slow Feature Analysis And Retinal Waves.
  Bernstein Conference on Computational Neuroscience, TU Berlin, 2010.

- N. Wilbert, R. Legenstein, L. Wiskott (2009).
  Slowness in hierarchical networks for visual processing.
  Sloan-Swartz Centers for Theoretical Neurobiology Annual Meeting 2009, Harvard University, July 25-28, 2009.

- N. Wilbert, M. Franzius, R. Legenstein, L. Wiskott (2009).
  Reinforcement learning on complex visual stimuli.
  Eighteenth Annual Computational Neuroscience Meeting: CNS*2009, Berlin, Germany, 18-23 July 2009, publ. BMC Neuroscience 2009, 10(Suppl 1):P90.

- S. Dähne, N. Wilbert, L. Wiskott (2009).
  Learning complex cell units from simulated prenatal retinal waves with slow feature analysis.
  Eighteenth Annual Computational Neuroscience Meeting: CNS*2009, Berlin, Germany, 18-23 July 2009, publ. BMC Neuroscience 2009, 10(Suppl 1):P129.

- C. Hinze, N. Wilbert, L. Wiskott (2009).
  Visualization of higher-level receptive fields in a hierarchical model of the visual system.
  Eighteenth Annual Computational Neuroscience Meeting: CNS*2009, Berlin, Germany, 18-23 July 2009, publ. BMC Neuroscience 2009, 10(Suppl 1):P158.

- M. Franzius, N. Wilbert, L. Wiskott (2008).
  Unsupervised learning of invariant 3D-object and pose representations with slow feature analysis.
  Proc. Federation of European Neuroscience Societies (FENS) Forum 2008, Geneva, July 12-16.

- M. Franzius, N. Wilbert, L. Wiskott (2007).
  Unsupervised learning of invariant 3D-object representations with slow feature analysis.
  Proc. 3rd Bernstein Symposium for Computational Neuroscience, Göttingen, September 24-27, p. 105.

- N. Wilbert, M. Franzius, R. Cichy, S. Schmidt, S. Brandt, L. Wiskott (2007).
  Towards a model of visual attention.
  Proc. Midterm Evaluation of the German National Network for Computational Neuroscience, Berlin, Germany, December 3–4, p. 30.

München, den 06.05.2011

Niko Wilbert