This is to certify that the
thesis entitled

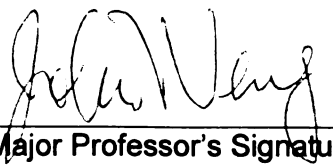
Multi-Layer In-Place Learning
for Autonomous Mental Development

presented by

Matthew D. Luciw

has been accepted towards fulfillment
of the requirements for the

Master of Science degree in Computer Science and Engineering



Major Professor's Signature

8/24/2006

Date

MULTI-LAYER IN-PLACE LEARNING FOR AUTONOMOUS MENTAL DEVELOPMENT

By

Matthew D. Luciw

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Computer Science and Engineering

2006

ABSTRACT

MULTI-LAYER IN-PLACE LEARNING FOR AUTONOMOUS MENTAL DEVELOPMENT

By

Matthew D. Luciw

Cortical self-organization during open-ended development is a core issue for perceptual development. Traditionally, unsupervised learning and supervised learning are two different types of learning conducted by different networks. However, there is no evidence that the biological nervous system treats them in a disintegrated way. The computational model presented here integrates both types of learning using a new biologically inspired network whose learning is ***in-place***. By in-place learning, we mean that each neuron in the network learns on its own while interacting with other neurons. There is no need for a separate learning network. Presented in this thesis is the Multi-layer In-place Learning Network (MILN) for regression and classification. This work concentrates on its two-layer version (without incorporating an attention selection mechanism). The network enables both unsupervised and supervised learning to occur concurrently. Within each layer, the adaptation of each neuron is nearly-optimal in the sense of the least possible estimation error given the observations. MILN is intended to provide memory organization capability and to be used as a general-purpose regressor in Autonomous Mental Development. The theory behind the network is discussed in detail. Experimental results are presented to show the effects of the properties investigated.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Autonomous Mental Development [41] intends to advance the AI field through the developmental approach, based on the mechanisms of perceptual and cognitive learning and development in humans from infancy to adulthood. The program that controls the machine is called the developmental program, inspired by the genetic program that drives the growth of an intelligent human being from the single-celled zygote. This is a push away from traditional, task-specific programs in robotics. In AMD, the developmental program must be general purpose, allowing a robot to be taught new tasks that were not conceived of at programming time. This is not to say the machine's development occurs from zero capability – innate behaviors can be very useful as a starting point. But, the main goal is to allow intelligent capability to be incrementally scaled from the ground up.

The developmental program must provide a broad developmental framework for sensory, cognitive and motivational growth, and thus cannot be task-specific. Additionally, it must be able to fulfill some difficult requirements, such as incremental learning given high-dimensional signals, real-time performance, performing while being taught, and adaptability to unexpected environments.

Recent evidence from neuroscience suggests that developmental mechanisms in the brain might be very similar or identical across different sensing modalities [24]. Thus, the "complexity problem", an argument against a brain-based approach to AI based on the

sheer number of neurons and synapses in the developed brain, might not be as much of an issue as was previously thought.

Past work on a developmental program was heavily dependent on the IHDR technique [38]. IHDR, briefly, is a general purpose regressor that can deal with very high-dimensional input due to its class-discriminating dimensionality reduction capability and perform in real-time due to its automatically imposed tree structure. However, the concept of IHDR is not totally founded in biological plausibility. In particular, the internal representation of the environment was somewhat inefficient, as IHDR stores nearly every sample, making memory an issue. Further, IHDR is not an in-place algorithm. By in-place learning [42] [40], it is meant that the signal processing network itself deals with its own adaptation through its own internal physiological mechanisms and interactions with other connected networks and, thus, there is no need to have an extra network that accomplishes the learning (adaptation). This is different from most classic neural network approaches to learning, which require a mechanism that is not accounted for by neuron behaviors and connections to train the network.

This thesis is concerned with explaining the motivation, theory, and applications of an in-place network for general purpose regression, which can fulfill the difficult requirements of AMD. The Multi-Layer In-Place Learning Network (MILN) was proposed by Weng and Luciw [40] and Weng *et al.* [39] for such a purpose. MILN integrates supervised learning due to teaching with unsupervised learning due to natural correlations within the input and output. Currently there is a lack of biologically inspired networks that integrate these two different learning modes using a single learning network. MILN enables unsupervised and supervised learning to take place at the same time throughout the network.

It is not convincing that biological networks use two different types of networks for unsupervised and supervised learning, which occur in an intertwined way in the process of development. When a child learns to draw, his parent can hold his hand during some

periods to guide his hand movement (i.e., supervised) but leave him practicing on his own during other periods (i.e., unsupervised). Does the brain switch between two totally different networks, one for supervised moments and the other for unsupervised moments? The answer to this type of question is not clear at the current stage of knowledge. However, there is evidence that the cortex has wide-spread projections both bottom-up and top-down [30] (pages 99-103). For example, cells in layer 6 in V1 project back to the lateral geniculate nucleus [21] (page 533). Can projections from later cortical areas be used as supervision signals?

One of the major advantages of supervised learning is the development of certain invariant representations. Some networks have built-in (programmed-in) invariance, either spatial, temporal or some other signal properties. Other networks do not have built-in invariance. The required global invariance then must be learned object-by-object. However, they cannot share invariance of subparts (or locally invariant features) for different objects. Consequently, the number of samples needed to reach the desired global invariance in object recognition is very large.

The general-purpose, multi-layer network discussed herein will learn invariance from experience. The network is biologically inspired. The network has multiple layers; later layers take the response from early layers as their input. This work concentrates on two layers. The network enables supervision from two types of projections: (a) supervision from the succeeding layer; (b) supervision from other cortical regions (e.g., as attention selection signals). The network is self-organized with unsupervised signals (input data) from bottom-up and supervised signals (motor signals, attention selection, etc.) from top-down.

From a mathematical point of view, in each layer of the network, unsupervised learning enables nodes (neurons) to generate a self-organized map that approximates the statistical distribution of the bottom-up signals (input vector space), while supervised learning adjusts the node density in such a map so that those areas in the input space that are

not related (or weakly related) to the output from this layer receive no (or fewer) nodes. Therefore, more nodes in each layer will respond to output-relevant input components. This property leads to increasing invariance from one layer to the next in a multi-layer network. Finally, global invariance emerges at the last (motor) layer.

The network learns to represent the input in a near-optimal way. Theoretically, the stored representations of the environment will be quasi-optimally efficient, in the sense of least estimation error from the observations. But each representation never totally converges – providing capability to adapt to new environments.

The remainder of this thesis is organized as follows. Autonomous Mental Development (AMD) and background information are discussed in Chapter 2. Chapter 3 establishes a AMD-relevant criteria for learning algorithms and discusses some existing methods with respect to that criteria. Chapter 4 explains the theory of a developmental sensory mapping, part of a proposed developmental architecture presented in [47] and [37]. This sensory mapping addresses a very difficult learning problem, which the proposed network is meant to address. Chapter 5 introduces the sensory neuron, the fundamental unit of feature detection, and the theory of optimal representation of the environment is discussed. The next three chapters deal with possible ways of automatic feature detection by groups (layers) of neurons: Principal Component Analysis (Chapter 6), Incremental Component Analysis (Chapter 7), and Lobe Component Analysis (Chapter 8). Chapter 9 presents the Multi-Layer In-Place Learning Network described above, which uses Lobe Component Analysis on each layer. Chapter 10 presents experimental results. Chapter 11 provides concluding remarks.

Chapter 2

Background

2.1 Autonomous Mental Development

Autonomous Mental Development, proposed by Weng *et al.* [41] is a new AI field emphasizing embodiment, autonomy, and a concept called *task-nonspecificity*. In autonomous learning, the emergence of behaviors and skills is a result of the interaction between the learner and the environment; these behaviors and skills are not necessary hard-coded to appear. In an autonomous-learning program, the mechanisms of learning must then be entirely contained within the program itself. Tasks are learned as a combination of "innate", or preprogrammed, drives and values, and a set of algorithms not designed so that a robot learns to perform a particular task, but instead to interact with the environment in such a way that task learning automatically emerges, given a necessary set of environmental constraints conductive to learning that task (such as a teacher and a set of innate abilities that map the teacher's actions to the set of the robot's innate values).

A program that is both embodied and allows autonomous learning is called a *developmental program*. A developmental program contains all the mechanisms necessary for the robot to learn all tasks it might need to learn, even tasks that the programmer might not have thought of. The hope is that this concept will further influence a paradigm shift in programming robots: away from task-specific programs and towards more adaptive

learning-based programs. Task-specific programs are limited by the programmer's choice of task. No matter how sophisticated the robot's physical properties, i.e., sensors and effectors, it can never use them to do anything beyond that task. If a new task becomes necessary, a new program must be written. In AMD, each task is learned as a result of the robot's interactions with the environment, with the teacher becoming the driving force of new task emergence.

2.1.1 Drawbacks of task-specific perception

In order to interact with the complex world in a human-like way, a robot must parse complex, high-dimensional sensory input so that its (non-imposed) current task can be accomplished. The predominant method used to do this is the use of hand-designed features such as oriented edges or colors. These features extract task-specific information and convert the complex sensory stream into a simpler feature vector.

From our perspective, there are several major drawbacks to the use of hand-designed features. First is the limitation of a robot's potential abilities through an imposed, task-specific model of the environment. A set of preselected features can lead to good performance on a preselected task, but might not be applicable to a novel task, or even the same task in a different environment. Second, use of preselected features limits the online learning ability of a robot, as information not represented by the features will be lost from the raw signal, no matter what that signal is. If, in a novel environment, this information becomes important to behavior, it should not be thrown out.

2.1.2 Embodiment

Desirable (from a developmental programmer's point of view) properties of human perception include our ability to learn a large number of different categories and classify an observed object despite a limited range of variations (e.g., size, pose, lighting, etc.). In traditional computer vision, these are hard problems, with no current program per-

forming at a human-level. The theory of AMD posits that invariant categories must be learned through real-time, online interactions with the environment. Researchers such as Edelman [10] and Pfeifer [29] also view this problem as one of learning sensory-motor correlations. In an interesting recent result [23], it was shown that a mobile robot learned to detect invariant object classes from continuous visual input, primarily because of the robot's self-motion. The slow-changing stream of input was also shown to be essential to developing the invariant representations. As for the large number of categories learnable by humans, it is possible this is attributable to the hierarchical nature of object detection [15], in that most objects can be thought of as a collection of parts.

2.2 Link to Biological Development

AMD establishes a link between developmental robotics and biological development through the concept of the developmental program, which is analogous to the genome. In humans and animals, development starts within a single cell (called the zygote) containing the genome of the organism. All physical and mental development thereafter is dependent on the genome's coding. The emergence of behaviors and skills is a process driven by the genes. At latest estimate, the human genome contains 20,000 - 25,000 genes [8]. Compare that to the estimated 10^{11} neurons and 10^{14} synapses in the central nervous system of a mature adult. The discrepancy in number between the two estimates implies that the operations done by many neurons are common.

The hypothesis (strongly put forth in [28]) of the *input-driven self-organization* of the cortical areas of the brain is that fully developed neurons will respond to different stimuli because of adaptation to different types of input over time. The operations done by each neuron are the same, but the input is different. An experiment [24] that was very close to testing exactly this was done on a newborn ferret. Input from the visual sensors was redirected to the auditory cortex, and input from the auditory areas was disconnected.

It was found, after some development, that what would have been the auditory cortex in a normal ferret had altered its representations to function as the visual cortex for the "rewired" ferret. Thus, it is thought that the categorization and recognition ability of the different cortical areas develops as a result of the same mechanism throughout.

Many neurons in the first layer of the visual cortex (known as V1) develop sensitivity to edges at preferred orientations [16]. The neurons in V2, which takes afferent input from V1, become sensitive to more complex shapes, such as corners [15]. By the hypothesis of input-driven self-organization, neurons in both these areas are operationally identical, i.e., if their positions were switched before stimulation they would learn to detect features similar to other neurons in their new areas. If this is the case, then the same self-organizing learning mechanism is active in both areas.

Feature detection by cortical neurons is not only a function of their afferent inputs; it is known that their competitive interactions (e.g., lateral inhibition) with one another play a role. What could neurons be competing for? It is possible that they compete for stimulation. A non-stimulated neuron will be induced to destroy itself through a process called apoptosis [9], so there is, at least on the surface, a motivation for neurons to compete.

An aim in computational neuroscience is to understand and to model the mechanisms of learning at the neural level. This is also a goal for AMD, as it may lead to understanding of how the human perceptual system, with the properties of automatic derivation of many object classes and invariant detection of these classes, develops.

Chapter 3

Criteria and Evaluation of Methods

This chapter discusses some of the challenges of usable programs in AMD, presents a set of criteria (from [42] and [40]) for evaluating algorithms for use in AMD, and reviews a set of algorithms with respect to that criteria.

3.1 Restrictions of a Developmental Program

- **On-line Operation**. In AMD, learning should be done sample-by-sample. Samples arrive in a high-dimensional continuous stream, and cannot be stored for very long.
- **Real-Time Operation**. For each sample, learning must occur within a limited time period. A framerate (e.g. 30 frames per second) can be specified. Any algorithm must finish processing each frame before the next arrives.
- **High-Dimensional Inputs and Outputs**. A developmental robot must have high-dimensional visual sensors, and complex effectors are necessary to manipulate most objects. Thus, any learning algorithms must be able to handle (converge, run at all, etc.) high-dimensional data spaces. This also has an impact on the previous criteria, since usually, more dimensions means that more computations must be done.
- **Plasticity** A non-plastic algorithm has a learning rate that approaches zero as time

goes to infinity. These algorithms cannot adapt to different environments over time, or even variations in the same environment, e.g., different lighting conditions in the same room.

- **Autonomy**. If an agent is to be truly autonomous - a self-directed learner - the mechanisms of learning must be entirely contained within the program itself. There is no off-line tuning of the program. Any necessary distinction between a training phase and a testing phase, and any changing of program parameters must occur in response to sensory input.

3.2 Types of Learning Algorithms

Traditionally, a learning algorithm is called online when it discards each sample after it is observed. An online algorithm learns after each sample, starting with just a single sample. All other algorithms have traditionally been labeled as offline, and learn only once, after the point when all samples have been collected.

For purposes of developmental robotics and AMD, a finer classification system is necessary. Any algorithm that requires all samples be available before learning is useless to us, due to the on-line operation constraint. But if an algorithm only requires a certain number (a block) of samples before updating, and can update more than once, it might be usable. It depends on memory constraints, however. Algorithms that do not need to store any input samples at all are even better, since sensation is incremental as well as high-dimensional. But because of these high-dimensional sensors, the higher-order statistics such as the covariance matrix become infeasible to estimate due to memory limits (and time).

As discussed previously, we desire algorithms for AMD to perform in-place neuron-learning. In addition to the purely practical limits described above, in-place algorithms are biologically-founded. AMD isn't about solving an engineering problem only. It also

seeks to test and validate embodied models of the biological development of intelligence using robot platforms. With an in-place algorithm, learning occurs as a side-effect of low-level unit behaviors and competitive interactions. There is no guiding developmental mechanism overseen by a separate learning network, such as minimizing error rate in gradient-based backpropagation learning, unless it is biologically founded (i.e., using rewards from something similar to the dopaminergic system).

To classify the suitability of learning algorithms for AMD, we define five types:

Type-1 batch: A batch learning algorithm L_1 computes g and $\mathbf{w}$ using a batch of vector inputs $B = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_b\}$, where b is the batch size.

Type-2 block-incremental: A type-2 learning algorithm, L_2, breaks a series of input vectors into blocks of certain size b ($b > 1$) and computes updates incrementally between blocks. Within each block, the processing by L_2 is in a batch fashion.

Type-3 incremental: Type-3 is the extreme case of Type-2 in the sense that block size $b = 1$.

Type-4 covariance-free incremental: A Type-4 learning algorithm L_4 is a Type-3 algorithm, but, it is not allowed to compute the 2nd or higher order statistics of the input $\mathbf{x}$. The CCI PCA algorithm [43] is Type-4 algorithm.

Type-5 in-place neuron learning: A Type-5 learning algorithm L_5 is a Type-4 algorithm, but further, the learner L_5 must be implemented by the signal processing neuron. Learning is a side-effect of the neuron interactions within the system.

It is desirable that a developmental system uses an in-place developmental program due to its simplicity and biological plausibility. Further, biological in-place learning mechanisms can facilitate our understanding of biological systems since there is no evidence that each biological network has a separate network to handle its learning.

The five types of algorithms have progressively more restrictive conditions, with batch (Type-1) being the most general and in-place (Type-5) being the most restrictive.

For an algorithm to be useful in AMD, it must at least be Type-3, due to the require-

ment of real-time, online and incremental processing. Due to the high-dimensionality of the inputs, a Type-4 algorithm is necessary, as storage of the higher order statistics becomes impractical with high-dimensional input. Our goal is to build a system that is usable to fulfill the requirements of AMD through functional biological plausibility, thus we aim for a Type-5, in-place, implementation.

3.3 Review of Existing In-Place Implementations

Recent research involving Type-5 implementations includes the development of computational maps similar to those found in the visual cortex using the LISSOM [28] technique, the Dynamic Field Approach [11], and the "Brain-Based" architecture used in the Darwin VII robot [23].

Our work differs from the others in that the goal is Autonomous Mental Development realized in a developmental robot. The purpose of [28] is a programmed, biologically plausible model of vision development. Thus, real-time, online robot performance is not their concern. The Dynamic Field Approach is founded on a very broad idea – modeling the "dynamics of development" through layered neural systems. But so far the work has not been realized in a robotics application. The Darwin VII's architecture is specialized to fulfill the goal of the study – a real-time system where the robot learns invariant object categories from sensory-motor experience in a controlled "block-world" environment. So, the preprocessing of the visual stream was done in a task-specific way, by detecting only horizontal lines, vertical lines or blobs. All the experiments were done using striped blocks. It is probable that the same result would not have occurred if non-striped, or perhaps circular objects were used, but, of course, that was not their goal. However, in AMD, we cannot use a task-specific early visual representation.

3.4 Unsupervised and Supervised Learning Methods

Well known unsupervised learning algorithms include Self-Organizing Map (SOM) [22], vector quantization, PCA [20], Independent Component Analysis (ICA) [7], Isomap [33], and Non-negative Matrix Factorization (NMF) [25]. Only a few of these algorithms have been expressed by in-place versions (e.g., SOM and PCA [43] – and SOM does not take into account statistical efficiency, making it difficult to tune when given high-dimensional input).

Supervised learning networks include feed-forward networks with back-propagation learning (e.g., [45]), radial-basis functions with iterative model fitting (based on gradient or similar principles) (e.g., [31]), Cascade-Correlation Learning Architecture [12], support vector machines (SVM) [6], and Hierarchical Discriminant Regression (HDR) [17]. None of these have been expressed by in-place versions.

Chapter 4

A Developmental Perceptual Mapping

4.1 The Sensory Mapping

The purpose of what is named the *sensory mapping* by Weng *et al.* [47] [37] is threefold. First, to automatically derive representative features and classes from the experiences of an agent. Second, to provide a source for learning attention selection – learning which part of the image is important for the task at hand. Third, to provide a target for attention itself. The MILN network is meant to address these problems.

It is known that neurons in the early visual pathway detect light at different places on the retina. The area of activity per neuron is called its receptive field. These receptive fields are very small in the early areas, such as V1, thus providing a sparse coding of the image for later areas. Questions that pertain to a developmental sensory mapping for AMD are how these receptive fields are developed, what the features detected in early vision are, and how this organization can help coordinate our attention selection capability.

4.2 Theory of Sensory Mapping

Here, the sensory mapping is formally defined.

4.2.1 Input formulation

This work mainly has to do with vision. As such, the formulation of the sensory mapping will be presented here with images containing the information to be perceived. The following defines the sensory mapping's input space, originally from [37]. A digital, grayscale image contains a set of pixels organized into r rows and c columns. At each time t, an image is provided, in the following form:

$$X(t) = \begin{bmatrix} x_{1,1}(t) & x_{1,2}(t) & ... & x_{1,c}(t) \\ x_{2,1}(t) & x_{2,2}(t) & ... & x_{2,c}(t) \\ ... & ... & ... & ... \\ x_{r,1}(t) & x_{r,2}(t) & ... & x_{r,c}(t) \end{bmatrix} \tag{4.1}$$

Any image in matrix form must be mapped onto vector space for further processing. Each image matrix X is mapped to a vector x with d elements:

$$x(t) = (x_1(t), x_2(t), ..., x_{d_x}(t))^T \tag{4.2}$$

By concatenating each row onto the end of the previous row, so that

$$x_{(j-1)c+i}(t) = X_{i,j}(t), i = 1, 2, ..., r, j = 1, 2, ..., c \tag{4.3}$$

Then the length of the image vector $x(t)$ gives the dimensionality $d_x = rc$.

Real-world visual input is given as a video stream. A spatiotemporal input vector includes the last h input frames, where h is the length of the short term memory. The spatiotemporal input vector $y(t)$ contains s elements:

$$y(t) = (y_1(t), y_2(t), ..., y_{d_y}(t)) \tag{4.4}$$

This vector is composed by concatenating the vector of each frame in order from the most recent through the last h:

$$y_{(k-1)d+i}(t) = x_i(t-k+1), i = 1, 2, ..., d, k = 1, 2, ..., h \tag{4.5}$$

The length d_y of this vector is the number of pixels from the last h image frames: $d_y = d_x h = rch$.

The images may have multiple components per pixel. E.g. in RGB format, each pixel is associated with a different intensity value for red, green, and blue components. There are other such formats, like YUV, where each pixel is composed of separate (usually three) components. In these cases, each component of each pixel becomes a single value in X. It is useful (and necessary, if receptive fields are to be used) to place the components of each pixel in the same row and in three subsequent columns.

Note that the choice of concatenating row by row in x versus column by column is immaterial, as long as it is done that way consistently, since the possible correlation between every possible pair of elements within y will be considered.

Each vector y gives a spatiotemporal input to the sensory mapping. The space of all possible values for y is given by

$$\mathcal{Y} = \{y(t) \mid t = 0, 1, ..., \infty\} \tag{4.6}$$

the space $\mathcal{Y}$ gives the input space of the sensory mapping. The sensory mapping component of a developmental architecture is a function

$$\mathcal{T} : \mathcal{Y} \longmapsto \mathcal{L} \tag{4.7}$$

where $\mathcal{L}$ is the output space of the sensory mapping, and the mapping itself, $\mathcal{T}$, is developed through experience.

4.2.2 Events

Learning the function $\mathcal{T}$ depends on the existence of repeated *events* within $\mathcal{Y}$.

An event is a set of correlations among a set of dimensions of the input space. The structure of these may be primarily spatial or spatial with a significant temporal component. An event that occurs often should be learned to be detected by $\mathcal{T}$. To do this, a number of different event-detection units (which are commonly called many things: filters, feature detectors, neurons, and lobe components) are used. Each one of these has a set of weights for all dimensions of the input, giving a weight vector w. Each weight vector corresponds to an event which can be detected within the input y by use of vector inner product (or dot product). This operation is denoted by the "$\cdot$" symbol, and is done as follows

$$y(t) \cdot w(t) = \sum_{i=1}^{s} y_i(t) w_i(t) \tag{4.8}$$

From now on, it will be useful to discuss comparisons between weights and inputs geometrically. In the last section, the input vector was introduced as a set of components, each of which corresponds to a dimension. In a Euclidean coordinate space, the vector is best visualized as a line with a certain direction and length (referred to as magnitude), emanating from the origin point.

The inner product operation takes the product of the magnitude of each vector and the angle θ between two vectors, as follows

$$y(t) \cdot w(t) = \|y(t)\| \, \|w(t)\| \, cos(\theta). \tag{4.9}$$

Taking the cosine on the angle gives a larger value when the angle between the two vectors is near 0° or 180° and converges to zero as the angle goes to $\pm 90^\circ$. Considering we take the absolute value of the input, for each neuron, the highest result from the inner product operation will be when θ is smallest, i.e., when the vector directions are most similar. The direction of the weight vector gives the *sensitivity* of the neuron, as it determines the components in the input space where, when they are high, the inner product operation

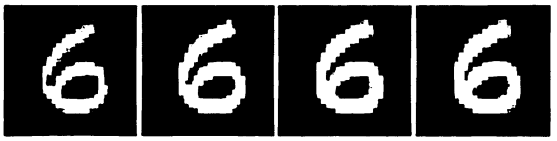


Figure 4.1: Each of the four images represents a vector of equal direction, but different magnitude. There is no need to distinguish between these in a classification task.

gives the highest value.

The length of a vector can be normalized, i.e., set equal to one, by dividing each element by it's length: $w' = \frac{w}{\|w\|}$. Normally, w is normalized before computing the inner product. In order to find the best-matching neuron to an input stimulus, the responses of neurons should be based on their direction, and not length. This has to do with competition and will be explained in more detail in later sections. In any case, the inner product operation is the crucial component in determining what is called the *response* of each neuron, and this response will be high when w and y are similar in direction.

Sensitivity is better defined using vector direction instead of Euclidean distance. Events with the same direction, but with different magnitudes, should be classified as the same, as in Fig. 4.1.

A fundamental problem is how to handle variations of similar events within $\mathcal{Y}$. The same set of relative correlations could occur at two different spatial positions within the visual plane. The system should recognize this event as the same in both cases. For example, consider one of the images in Fig. 4.1. If the digit within that image is translated several pixels to the left, then a event-detector with sensitivity based on the originally placed digit will not respond as highly to the translated image. A system with translation-invariant detectors would respond the same in both cases.

There is another purpose of the sensory mapping besides recognition: providing a source and target for attention. An agent that interacts with the environment must know the position of the objects. So it is not enough to just detect that something is within the image, as in some probabilistic models of visual object recognition. The location (which in some cases may need to be very precise) of the event is also required. How can many

different areas within the image plane be represented internally, so that the input can be automatically segmented?

To handle the above issues, a receptive field architecture is used.

4.2.3 Receptive fields

In the appearance-based approach to recognition, feature extraction, selection and detection are done using the entire input vector. In the input formulation introduced in Section , the space $\mathcal{Y}$ contained the entireties of each image from each time. If features are developed that use this entire vector, invariant detection becomes computationally heavy (a different neuron is needed for every possible variation of the feature) and fundamentally flawed. Additionally, using this (lack of an) architecture to learn attention selection is problematic. This basic approach is *monolithic*. If the program input is a set of same-size objects, each at the exact same position within the image, the features extracted from the input will only be useful for objects at that exact position. The receptive field approach is nonmonolithic - object recognition is eventually handled by detection of subparts, e.g. a face can be recognized as two eyes, a nose, and a mouth in a certain relative configuration.

The receptive field of each neuron is defined as the set of dimensions in $\mathcal{Y}$ for which the neuron has a nonzero weight. For convenience, "nonzero" can be taken to mean "significant" so that dimensions associated with very small weights, which will have no bearing on the developed filters, can be considered to be outside of the receptive field. Receptive fields can be hardcoded, developed, or a combination of both. In the last, each neuron is given an initial receptive field, and the nonzero weights will be further pruned during development.

The following parameters define all receptive fields:

1. The **number** of different receptive field locations n_r, as well as the total number of sensory neurons n_s. Note that these two parameters are not necessarily equal - multiple neurons may have the same receptive field. For each neuron, the following

four parameters must be set:

2. The **position** or center of the field. This is defined as the center of the field in 3D input space - i and j give the spatial center, k gives the temporal center.

3. The **shape** of the field's coverage over the 3D input space. For example, cubic, cylindrical, spherical, multiple cylinders (disconnected), or a combination of several different shapes.

4. The **size** of the field. The parameterized shape function is $s(\theta)$, where s is the shape function chosen, and size is determined by θ (which may represent multiple parameters, such as radius and height for a cylinder).

5. The **cortical area** of the neuron(s) associated with the field. There are L areas $A_1, A_2, ..., A_L$. Afferent input (meaning: not considering the lateral or feedback inputs) to neurons on A_1 is directly from the input space. Neurons with higher-numbered areas take direct afferent inputs are from the outputs of the previous area. However, the classical receptive field of higher-area neurons is still defined in terms of coverage over the input space. See section 4.2.4 for more discussion.

The classical receptive field of each neuron is defined as $y = S(s(\theta), i, j, k)$ where $s(\theta)$ is the parameterized shape function defining what relative inputs to include, and i, j, k determine the center of that shape in the 3D input space.

These parameters define all starting receptive fields in the system They should be further developed so that the system will converge to some quasi-optimal organization of receptive fields for the agent and environment. Exactly how to do this for all five parameters is still an open problem, although it was shown in [48] automatic development in terms of size, position, and shape.

The tradeoff dealt with both manually selecting and automatically developing values for the set of receptive field parameters is that of efficiency versus effectiveness. The

most effective set of receptive fields, in terms of possible attended locations, is one for every possible combination of dimensions. Due to time and memory restrictions and computational load, that is probably not feasible. The most efficient set of receptive fields is just one, at a single pixel. This provides very poor coverage as well as very poor features, obviously.

The issue is one of segmentation: if an object is positioned where it is bisected by two (or more) receptive fields, the recognition will be worse than in the case where it is centered. That is why an overgrowth of receptive field locations may be necessary, although a saccadic-like attention mechanism may be able to center any possible objects.

4.2.4 Areas and layers

According to the well-known "curse of dimensionality", it requires exponentially more samples to cover a higher-dimensional input space than one of lower-dimension. Therefore, convergence of learning algorithms in higher-dimensional space is exponentially slower, and requires exponentially more samples. If the "initial guess" for the algorithm is in a poorly populated area, convergence may not even occur (a problem as real-world input spaces are usually sparsely populated). But a useful input space for a robot that uses vision must be very high-dimensional. If we use a camera, which gives a relatively small 50 x 50 image, our input dimensionality is $50 * 50 = 2500$. If we want binocular vision capability, we'll need two cameras, and the dimension will double to 5000 pixels. Important events might take up a large area of the input space, or cover many pixels. If an event covers just a tenth of all pixels, features will have to converge in a 500-d subspace, containing 256^{500} possible sub-images. We will need receptive fields over large areas of the input space to extract and detect features for these large events. But because of the curse of dimensionality, extracting features using all dimensions for this size receptive field may be too slow, or might not work at all.

To effectively derive features for larger receptive fields, we will organize feature detec-

tors hierarchically into areas. Neurons from the first area perform feature extraction and detection, and thus dimensionality reduction, using lower-dimensional receptive fields and send the dimension-reduced signals for the next area to use as input. A neuron in a higher area will have a receptive field equal to the union of the receptive field elements of the connected neurons in the lowest area. There are p_h (which stands for "pyramid height") areas $A_1, A_2, ..., A_{p_h}$. The afferent inputs of neurons in area $A_i, i > 1$ are from the outputs of the neurons in area A_{i-1}. The inputs to neurons on A_1 are from the external environment.

The assumption we make by using a hierarchical architecture is that all events over a large area can be represented as a set of sub-events, each over a smaller area. E.g., a contour is detected by first detecting a set of smaller edges, arranged in a particular way. There is ample evidence that this is the case - the human visual system is also organized hierarchically - but there are a few issues we must handle to get this working in practice.

1. **Timing**. Operations in area A_i cannot start until area A_{i-1} is done since the input of A_i is the output of A_{i-1}. However, area A_{i-1} can start working on new input while area A_i is working. This is the same as the notion of pipelining in computer architecture. To maximize throughput, we will allow each area to begin processing on next input as soon as it (both the input and the area) is available. This introduces a delay into the system. The outputs of area A_{i-1} will be from one frame ahead of those of area A_i.

2. **Possible Information loss**. Features on each level must derive representations using only (the responses of) the previous area's features, not the direct input. Any information lost after features are detected in the input space can't be recovered. The next area considers an input space using only those features. We want to have a good array of features so that the features extracted by the next area are useful to us. If we extract only one feature at each location, we might be missing out on lower order correlations within the same field that may be useful further on.

We introduce the concept of layers. Each area contains p_d (standing for "pyramid depth") layers $L_l, L_2, ..., L_{p_d}$. This means there are p_d feature detectors on area A_a with a receptive field centered at the same location. Using multiple layers allows different features to be extracted from the same set of inputs. Ideally, the optimal value for p_d should be automatically derived.

3. **Speed.** The system is meant to be run on a developmental robot in real-time. If the processing is at too low of a frame rate, it will not be very useful. Without predicted attention, the system probably will be too slow to use.

4.2.5 Output formulation

In Section 4.2.3, the input space $\mathcal{Y}$ of the sensory mapping was defined. Originally from [37], the output space $\mathcal{L}$ is defined here:

Every neuron can be uniquely identified by its receptive field center i, j, k, area a, and layer within each area l. Each neuron $v_{i,j,k,l,a}$ can be thought of as a function that takes inputs $u_{i,j,k,l,a}$ and produces output $z_{i,j,k,l,a}$. For simplicity, all neurons can be indexed by a single value n so that

$$v_n = v_{i,j,k,l,a}, n = 1, 2, ..., n_s \tag{4.10}$$

From the vector of inputs at time t, each neuron produces output

$$z_n(t) = v_n(u_n(t)) \tag{4.11}$$

The output of all neurons in the sensory mapping can be represented by $z(t)$

$$z(t) = (z_1(t), z_2(t), ..., z_{n_s}(t)) \tag{4.12}$$

All possible outputs give the output space, and the data manifold $\mathcal{L}$ in this space is

given by the value of z at every time

$$\mathcal{L} = (z(t) \mid t = 1, 2, ..., \infty) \tag{4.13}$$

The sensory mapping is a function T that takes input $y(t)$ and produces output $z(t)$

$$z(t) = T(y(t), a(t)) \tag{4.14}$$

T automatically develops over time, and is influenced by the attention vector $a(t)$, described next.

4.3 Architecture

The issues of segmentation and recognition constitute somewhat of a "chicken-egg" problem in developmental vision. Objects in the visual plane can't be recognized until they are separated from the background, but this can't be done until the objects are (at least partially) recognized. A large set of receptive fields at many different positions and sizes can locate objects simply by brute force, but this lacks in terms of efficiency. In natural images, many of the windows will contain only background (non-category) pixels. It is a waste to devote computational resource to the windows where there are no objects. How can the locations of these windows be known?

An attention-selection mechanism predicts the location of receptive fields containing non-background pixels. All other windows are ignored, meaning feature detection is not done in these locations, and computation time per frame is reduced. Attention requires that the input be a continuous video stream. A regressor maps previous attended locations and features detected within to the next attended locations. The core assumption we make is that the positions of known objects over time can be predicted after having seen enough examples of that object's movement (if it has any) or having experienced how the agent's self-movement influences the visual position of objects. The attention vector $a(t)$,

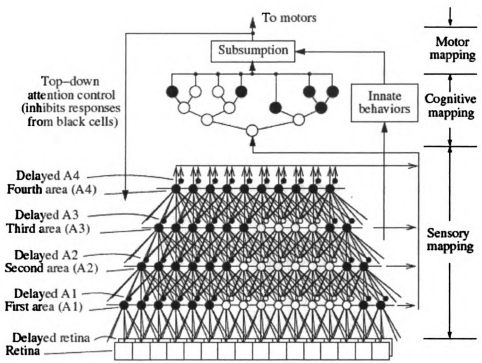


Figure 4.2: Flow diagram of a developmental perceptual system (figure courtesy of [37]).

originating from outside the sensory mapping, inhibits the responses from most neurons at each time. In it's simplest form, the attention vector is a binary vector with dimension equal to the number of receptive field centers. Each component is coupled with a receptive field center. A 1 in the attention vector means that the receptive fields at that location would be attended. The rest (corresponding with 0 values) would be ignored.

Figure 4.2 shows a flow diagram of a developmental vision system. The vision system is separated into three mappings: sensory, cognitive, and motor, along with a block for innate behaviors. The retina and delayed retina give the set of input pixels (the input vector from $\mathcal{Y}$). The black and white circles represent the many feature detectors in different areas and layers. They are connected, either directly or indirectly (through lower numbered areas), to the input. The cognitive mapping [36] learns a function from input $z(t)$

(the sensory output vector) to output $a(t)$. Mimicking a saccadic mechanism, attention is applied by the motor area. White circles in the figure indicate receptive fields within an attended area, while dark circles indicate receptive fields that are not attended. The lower-level innate behaviors are in themselves a complete sensorimotor loop, and give some predefined behaviors for when the higher-level system is foundering. For example, if no objects have been found, perform a random window search. The subsumption (introduced in [4]) mediates the action produced by this module and the action produced by the cognitive mapping, and only allows one (with the highest confidence with preference given to the higher level) to be applied.

It should be noted that a the input to this structure doesn't have to take only visual input. It will work the same given input from a set of any sensory modalities.

Chapter 5

Optimal Representation

5.1 Introduction

The next few chapters of this thesis concern the activities of the system at the neuron level. This section is about neural adaptation mechanisms without considering competition with other neurons.

In feature extraction, the goal is to find and best represent the important sections on the input manifold, i.e., the parts with significant correlation among different signal lines. In AMD, the input space is high-dimensional, and the input to each neuron will contain many signals. The following subsections detail a set of operations each neuron can do on its input, over time, to make it more likely that the features that do exist are detected in the input space of each. Per the restrictions of development, our goal is to find ways each may be done incrementally, without use of a separate storage area or neuron developer outside of the neuron itself.

5.2 A Neuron's Input

A neuron's input comes from a set of d signal lines. Activity on these lines is sampled at time intervals from $t = 1, 2, ...$ The input to the neuron itself is the resulting sample

vector $x(t)$, which has dimension d.

Each neuron does not have any information about how the values of $x(t)$ relate to the (usually very large) world-specific input space. This would involve modeling world-specific knowledge. This is not necessary – an agent's sensors will contain enough information such that it can learn to act appropriately, without explicit knowledge of how the sensory information relates to the environment [5].

5.3 Position Normalization

Having no absolute knowledge, each neuron must compute everything relative to what it has previously experienced. Position normalization places the base of each the neuron's future experience at the location of the sample mean. All future samples are then evaluated by their position relative to the sample mean. Basically, position normalization moves the origin to an "important", highly populated, area of the vast, sparsely populated input space. So, the neuron is closer to any features that may exist, and convergence will be more likely and faster. An in-place way to compute mean is through the amnesic averaging technique, introduced in [43], and discussed here. The amnesic average is so-named since it will "forget" old observations, being useful for an agent experiencing multiple environments, since without forgetting, we only represent the early environments experienced.

5.4 Amnesic Mean

When samples generated by a new process are observed, samples from previous processes should be forgotten. But there is no way to know *a priori* when a new process will occur. A way to do this "forgetting" with no prior knowledge is to increase the learning rate as time increases. The amnesic mean [43] allows forgetting. The incremental, non-amnesic mean is defined as

$$\bar{x}(t) = \frac{t-1}{t}\bar{x}(t-1) + \frac{1}{t}x_t \tag{5.1}$$

where x_t is the current observation and $\bar{x}(t-1)$ is the previously stored average.

The following equations, originally from [42], increase the weight of new samples by adding the value returned by the amnesic function $\mu(t)$, and define the amnesic mean. Add $\mu(t)$ to the new samples:

$$w_2 = \frac{1+\mu(t)}{t} \tag{5.2}$$

and subtract the same value from the weight of the previous average

$$w_1 = \frac{t-1-\mu(t)}{t} \tag{5.3}$$

Since the same value that is added to w_2 is subtracted from w_1, the two weights still sum to 1

$$\frac{t-1-\mu(t)}{t} + \frac{1+\mu(t)}{t} = 1 \tag{5.4}$$

The amnesic averaging equation, given the previous average $\bar{(x)}(t-1)$ and the new sample x_t is as follows

$$\bar{x}^{(t)} = w_1\bar{x}(t-1) + w_2x_t \tag{5.5}$$

The function $\mu(t)$ is characterized by two "switch-points", t_1 and t_2. The amnesic mean is the same as the incremental mean, i.e., $\mu t = 0$, from $t = 0$ until switch-point t_1. From t_1 until t_2 the learning rate will increase linearly until $\mu(t) = c$. After t_2, $\mu(t)$ increases as a function of time and r.

$$\mu(t) = \begin{cases} 0 & \text{if } t \leq t_1, \\ c(t - t_1)/(t_2 - t_1) & \text{if } t_1 < t \leq t_2, \\ c + (t - t_2)/r & \text{if } t_2 < t, \end{cases} \tag{5.6}$$

Typical values used for the parameters are $t_1 = 20$, $t_2 = 200$, $c = 2$, and $r = 5000$. The reason this function is broken into three intervals has to do with the statistical efficiency (see Chapter 5.6).

5.5 Scatter Vector

For each arriving sample x_t, subtract the incrementally-calculated mean $\bar{x}(t)$. The resulting vector u is commonly called the scatter vector.

$$u(t) = x(t) - \bar{x}(t) \tag{5.7}$$

The distribution of the neuron's input samples $x(t), t = 1, 2, ..., n$ now becomes a distribution $u(t), t = 1, 2, ..., n$ where $\bar{u}(t) = 0$.

5.6 Statistical Efficiency

The theory of statistical efficiency is important in verifying how well each neuron will represent the samples that contributed to it, i.e., the optimality of the neurons can be measured. This chapter formally presents the most useful ideas regarding statistical efficiency for our purpose, and applies them to the amnesic averaging method. It is shown that the amnesic average is a quasi-optimal estimator for the true average of the samples (that is optimal if the amnesic parameter is set to zero – instead, we choose near-optimality so that the parameter can adjust). The theory and equations presented in this section are from [48], [43], and [42], with explanation written by myself.

5.6.1 Efficient estimator

A probability density function (p.d.f.) generates a set of observations S. The p.d.f. is parameterized by vector Θ, and θ is a particular component of Θ. We wish to estimate θ from S. Suppose Γ_1 and Γ_2 are two estimators of parameter θ. Their estimation is based on the observations S. Γ_1 is said to be more statistically efficient than Γ_2 if the expected square error between the estimated parameter and the actual parameter is less for Γ_1 than for Γ_2:

$$E \parallel \Gamma_1 - \theta \parallel^2 < E \parallel \Gamma_2 - \theta \parallel^2 . \tag{5.8}$$

(Equation provided in [48]). For a certain probability distribution, estimator $\hat{\theta}$ is the best estimator for parameter θ if it is unbiased (that is, $E \parallel \hat{\theta} \parallel = \theta$) and it reaches the minimum expected square error for that distribution.

5.6.2 Cramer-Rao lower bound

The Cramer-Rao bound is a well-known way to measure the efficiency of an estimator. It gives the minimum value, i.e., the lower bound, of the expected square error of the estimator $\hat{\theta}$ of parameter θ.

$$E \parallel \hat{\theta} - \theta \parallel^2 = \frac{1}{n \int_{-\infty}^{+\infty} [\frac{\partial log p(x,\theta)}{\partial \theta}]^2 p(x,\theta) dx} \tag{5.9}$$

The Cramer-Rao bound is a useful tool for evaluating the efficiency of an unbiased estimator. More efficient estimators will have expected square error closer to the bound. If the expected square error is equal to the bound, than the estimator is by definition the most efficient estimator of the parameter. It can be proven that taking the sample mean as an estimator for the population mean has mean square error equal to the Cramer-Rao bound if the p.d.f. has a known standard deviation σ and is Gaussian or exponential [27].

Thus, the sample mean is the most efficient estimator for the population mean for these distributions.

5.7 Efficiency Study of Amnesic Average

It must first be proved that the amnesic mean is an unbiased estimator of the population mean. If this is the case, its convergence to the population mean is guaranteed.

First, note that the amnesic mean is a weighted sum of all n samples, for any n

$$\bar{x}(n) = \sum_{t=1}^{n} w_t(n) x_t \tag{5.10}$$

It can be proven using induction on n, that the following (originally provided in [42]) gives the value of the weight of any sample $t = 1, 2, ..., n$, for any n

$$w_t(n) = \frac{1 + \mu(t)}{t} \prod_{j=t+1}^{n} \frac{j - 1 - \mu(t)}{j} \tag{5.11}$$

It can be seen that each weight is nonnegative $w_t >= 0$, and all weights sum to 1, for any n (the following was originally provided in [42]).

$$\sum_{t=1}^{n} w_t(n) = 1 \tag{5.12}$$

except when $n = 1$ - in that case, $\mu(t) = 0$. This relation can be proved by induction. The two induction proofs for equations 5.12 and 5.11 were left as an exercise, and were done by myself, and are contained in Appendix A.

If the samples x_t are independently and identically distributed (i.i.d.) from a random variable x, then the amnesic mean is unbiased. To prove this, $E\bar{x}(n) = Ex$, must be proved. The proof is as follows, originally provided in [42]. First, substitute using equation 5.10

$$E\bar{x}(n) = E\sum_{t=1}^{n} w_t(n)x_t. \tag{5.13}$$

the summation is a constant, so

$$E\sum_{t=1}^{n} w_t(n)x_t = \sum_{t=1}^{n} w_t(n)Ex_t. \tag{5.14}$$

Now, substitute using equation 5.12. And since the expected value of a set of samples that are i.i.d. from a random variable is equal to the expected value of that variable, substitute Ex for Ex_t, and the proof is complete

$$\sum_{t=1}^{n} w_t(n)Ex_t = 1Ex = Ex. \tag{5.15}$$

So, the amnesic mean is unbiased. Now, the efficiency is evaluated. It is expected that the efficiency of the amnesic mean is less than that of the sample mean, and that is the case.

In order to compare the efficiency of the amnesic mean to the sample mean, we derive an error coefficient for each. First note the variance of a parameter is the same as the expected mean square error $\mathbf{Var}(\bar{x}(n)) = E \parallel (\bar{x}(n) - \hat{x}(n))$, where $\hat{x}$ is the population mean. Assume random variable x has known variance σ^2. Now

$$\begin{aligned}
\mathbf{Var}(\bar{x}(n)) &= \mathbf{Var}(\sum_{t=1}^{n} w_t(n)x_t) \\
&= \sum_{t=1}^{n} w_t^2(n)\mathbf{Var}(x_t) \\
&= \sum_{t=1}^{n} w_t^2(n)\sigma^2
\end{aligned}$$

gives the expected mean square error of the amnesic mean. The rule $\mathbf{Var}(cx) = c^2\mathbf{Var}(x)$, where c is a constant and x is a random variable is used to move from the first

to the second step. The rest is substitution.

The expected mean square error of the sample mean, when $\mu(t) = 0$, is

$$\begin{aligned}
\mathbf{Var}(\bar{x}(n)) &= \mathbf{Var}(\sum_{t=1}^{n} w_t(n)x_t) \\
&= \sum_{t=1}^{n} w_t^2(n)\mathbf{Var}(x_t) \\
&= \frac{n}{n^2}\sigma^2 \\
&= \frac{1}{n}\sigma^2
\end{aligned}$$

since each weight in this case equals $1/n$.

The difference in efficiency between sample mean and amnesic mean can now be evaluated by comparing the amnesic mean's error coefficient $\sum_{t=1}^{n} w_t^2(n)$ to the sample mean's error coefficient $\frac{1}{n}$. When $\mu(t) = 0$, the error coefficient is that of the sample mean.

In each of the three cases, the amnesic mean will converge to the population mean. Note the extreme difference in expected error for the early samples. From this, it would seem a lower $\mu(t)$ is more beneficial. But when $\mu(t)$ is smaller, the less of a capability exists to adapt to a changing distribution. This is why the multi-section amnesic mean in equation **??** is used. Until the first switch-point, the sample mean is used for maximum statistical efficiency. It is assumed that these early samples are all from the same process. After the first switch-point, $\mu(t)$ is gradually increased, becoming a function of time after the second switch-point, whereafter it may adapt as $t \rightarrow \infty$.

In conclusion,the multi-sectional amnesic averaging technique will be useful for real experiences, such as robot vision for a large number of frames, where it is probable that the samples will be generated from different processes over time. The multi-sectional amnesic mean is unbiased (so that convergence to population mean is guaranteed for a

stationary process), efficient, but at the same time can adapt to a nonstationary or multiple generative processes. In the remainder of this document, the term mean (or average) should be taken to refer to the amnesic mean presented here.

Chapter 6

Principal Components Analysis

6.1 Introduction

Principal Components Analysis (PCA) [20] is one technique that can be used to automatically derive representation from the input samples. It has been popular in the appearance-based approach to computer vision, e.g., Turk and Pentland's Eigenfaces [34]. The set of principal components of a set of samples gives an optimal linear manifold for any dimensionality less than the original sample dimensionality. The samples can be projected into this smaller space to represent them in an efficient, linear way. Each of this basis' components, known as the principal components, is a feature. Weng *et al.* [43] proposed a fast-converging, candid, incremental algorithm for PCA that will converge in high dimensional sample spaces. This algorithm, known as CCIPCA, can be used to extract the principal components under the constraints of AMD.

The theory of CCIPCA is very important for the CCILCA technique used in MILN, since each lobe component is derived as the first principal component of the samples contributing to it. Additionally, the first principal component gives the optimal representation of the contributing samples.

6.2 Principal Component

A principal component is a vector obtained through eigen-decomposition of the sample covariance matrix. The covariance matrix A of a set of samples $x(t), t = 1, 2, ..., n$ with mean vector $\bar{x}$ is defined as

$$A(x) = \frac{1}{n-1} \sum_{t=1}^{n} (x(t) - \bar{x})(x(t) - \bar{x})^T \tag{6.1}$$

If the samples are already position normalized, equation 6.1 is the same as

$$A(u) = \frac{UU^T}{n-1} \tag{6.2}$$

where each sample $u(t), t = 1, 2, ..., n$ is a column vector placed in n-by-d matrix U, where d is the dimensionality of each samples. The reason the denominator in both equations is set to $n - 1$ instead of n since the sample covariance matrix will then be an unbiased estimator of the population covariance matrix (which is not the case if we divide by n).

The covariance matrix is an extension of the concept of variance to multiple dimensions. Along the diagonal, the value given at row i and column j is a measure of the variance of component i (or j). Visually, this usually (assuming there are no outliers) gives how much the samples are spread out along the axes of the coordinate space. The values that are not along the diagonal indicate the variance between two components. Visually, these values give how much the samples are spread out along dimensions exactly in between two original dimensions.

If the components are first scale-normalized, the value at row r and column c in the covariance matrix gives the correlation among components u_r and u_c of vector u over time. The higher the absolute value, the higher the correlation. A high correlation indicates a high-level of usefulness as a feature. Correlation among the components is graphically indicated as the distribution shape. Note that the direction where the correlations are the

highest might not be along any of the sample coordinate axes. This is where PCA comes in. The first principal component is the direction where the covariance is the largest. This is known as the Most Expressive Feature (MEF) [32].

A principal component is an eigenvector of the covariance matrix. An eigenvector v of the covariance matrix has corresponding eigenvalue λ. v has unit length, as do all principal components. λ gives the variance of the samples along v.

$$A(u)v = \lambda v. \tag{6.3}$$

The eigenvector with the largest eigenvalue is the first principal component v_1. The direction of v_1's is along the axis of largest variance in the sample space. v_2's direction is the axis of largest variance that is orthogonal to v_1. v_3's direction is the axis of largest variance that is orthogonal to both v_1 and v_2... and so on until d vectors are obtained. The corresponding eigenvalues are ordered such that $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_d$. Note that there may be multiple solutions for the principal components. For example, when the data is a sphere, every direction will have equal variance. In that case, any nonzero vector will be a solution for v_1.

All principal components form an orthogonal basis for a d-dimensional vector space, so each sample can be decomposed as a weighted sum of the components

$$u(t) = y_1 v_1 + y_2 v_2 + ... + y_d v_d. \tag{6.4}$$

where $y_1, y_2, ..., y_d$ are scalar coefficients. The principal components are ordered by variance: a higher-indexed component will be in a direction of higher sample variance than its lower-indexed components. Another way to phrase this is to say the samples are most correlated along the first principal axis, then the second principal axis, until the last. So v_1 is the most important component, v_2 is second-most important, and so on. It is expected that the lesser order components will not contain much useful information about

the samples. It will be beneficial to eliminate them. We can select a value k, where $k < d$, and project the samples into a k-dimensional space using the responses to only the first k principal components $\hat{y} = (y_1, y_2, ..., y_k)^T$.

Define the approximate vector $\hat{u}$ as the reconstruction of u using the first k principal components and the projection weights $y_1, ..., y_k$ of u to each

$$\hat{u} = y_1 v_1 + y_2 v_2 + ... + y_k v_k. \tag{6.5}$$

The optimality of PCA is defined as follows. The basis made up of principal components is optimal in the sense that the expected squared error between the approximate sample (in k dimensions) and the original vector (in d dimensions) is the least possible for any set of k orthogonal vectors. This can be formally proved using induction, but just the outline will be presented here, as it is quite intuitive. Think of the case where $k = 1$, i.e., we are using just one principal component. Then the problem is equivalent to fitting a line to the data to minimize the least square errors. From the theory of linear least squares, this line is along the dimension of highest sample variance. As stated above, this is the first principal component, and is the optimal axis to use as a one dimensional basis in terms of minimum information loss.

So, PCA is optimal when $k = 1$. Now, assume we already have an optimal basis for k dimensions and seek an optimal basis in $k + 1$ dimensions. This component must be orthogonal to the previous k. But consider a sample subspace gotten by simply leaving out the axes of any of the first k components. In this space, every vector will be orthogonal to all previous ones. The best axis in terms of least information lost in this space is along the direction of highest variance in this space. The first principal component in the subspace will be the best solution for basis vector $k + 1$ that minimizes the information lost for a basis with $k + 1$ components. Thus, PCA derives an optimal linear basis in the least-squared error sense, for any k.

In many cases, the number of dimensions that can be reduced is very large – especially

when the dimensionality is large to begin with.

For further processing, the samples are represented only by their responses (projections) onto the principal components. In this space, these responses are uncorrelated, meaning their covariance is zero.

6.3 Whitening

An important (see the experiment in Section 10.1.1) preprocessing procedure called whitening can now be introduced. Note that the components $y_1, ..., y_k$ of the responses $y(1), ..., y(t)$ after PCA will be uncorrelated. That is, the covariance of all pairs of different components will be equal to zero. Whitening takes this a step further and normalizes each response component by its standard deviation, so that the variance is unit. After whitening, the covariance matrix of $y(t)$ is the identity matrix.

The variance of component y_i is given by the eigenvalue λ_i of eigenvector v_i. The whitening matrix W is generated by taking the matrix of principal components $V = [v_1 v_2 ... v_k]$ and dividing each by its standard deviation (the square root of variance). The matrix D is a diagonal matrix where component (i, i) (along the diagonal) is $\frac{1}{\sqrt{\lambda_i}}$. Then, $W = VD$. To whiten, multiply the response vector(s) by this matrix.

$$\hat{Y} = WY = VDY. \tag{6.6}$$

A dewhitening matrix Z can be created to undo this procedure. It is defined as

$$Y = Z\hat{Y} = VD^{-1}\hat{Y} \tag{6.7}$$

where D^{-1} is the inverse of D.

Visually, whitening can be thought of as a "sphering" procedure, since afterwards, the data will have unit variance in all directions. It is a useful procedure to do before feature extraction, as extraction procedures based on high variance will otherwise be biased

towards the directions with larger eigenvalues.

It should be noted that an adaptive sigmoid performing scale normalization on the response signal line of each principal component is equivalent to equation 6.6.

6.4 Candid, Covariance-Free, Incremental PCA

How can the principal components be computed given the constraints of the developmental program? Weng *et al.* derived a Type-4 algorithm that computes the principal components. The theory and equations from this section are from that paper [43], but all explanation is as per my understanding. The CCILCA technique used on each layer of the in-place learning network computes the first principal component for each neuron using this technique.

6.4.1 Deriving the first eigenvector

.

Recall the definition of a principal component

$$\lambda v = A(u)v \tag{6.8}$$

where $A(u)$ is the covariance matrix of the samples.

Substitute for $A(u)$ using its definition

$$\lambda v = \frac{1}{n-1}\sum_{t=1}^{n} u(t)u(t)^T v \tag{6.9}$$

and note that $uu^T v = (u \cdot v)u$

$$\lambda v = \frac{1}{n-1}\sum_{t=1}^{n}(u(t) \cdot v)u(t). \tag{6.10}$$

There are multiple solutions to λv with regards to the scaling factor, or eigenvalue, λ. From now on, enforce the length of v to be unit, by normalizing it, so there will be only one eigenvalue per eigenvector direction. When v is unit, the value of λ will equal the variance of observations along axis v. We can now represent the left side simply as v, and the length $\|v\|$ will give λ.

$$v = \frac{1}{n-1}\sum_{t=1}^{n}\frac{u(t)\cdot v}{\|v\|}u(t). \tag{6.11}$$

The right side of the equation gives the parameter to estimate incrementally, notated as $b(t)$

$$v = \frac{1}{n-1}\sum_{t=1}^{n} b(t). \tag{6.12}$$

This is just the batch average of $b(t)$. Convert this to an incremental average:

$$v(t) = \frac{t-2}{t-1}v(t-1) + \frac{1}{t-1}\frac{u(t)\cdot v(t-1)}{\|v(t-1)\|}u(t). \tag{6.13}$$

from samples arriving at $t = 2, 3, \ldots$. Note that the incremental version must start at $t = 2$ since the first scatter vector will be a zero vector: $u(1) = 0$ (because $\bar{u}(1) = u(1)$).

Lastly, instead of incremental average, amnesic average is used to estimate $b(t)$. This is done for two reasons. First, $b(t)$ is a function of $v(t-1)$, which is not stationary. Second, there is no guarantee that the generative process itself will be stationary. This is the same reason we use amnesic mean ro estimate population mean: the developmental program's input will typically be samples from a long-running sensory stream from a changing environment.

$$v(t) \;=\; w_1 v(t-1) + w_2(t)\frac{u(t)\cdot v(t-1)}{\|v(t-1)\|}u(t), \tag{6.14}$$

$$\text{where } w_1(t) = \frac{t-2-\mu(t)}{t-1}, \tag{6.15}$$

$$\text{and } w_2(t) = \frac{1+\mu(t)}{t-1}. \tag{6.16}$$

6.4.2 Geometric explanation

Intuitively, the estimation of principal component vector v will be "pulled" by the direction of the each observation. That is, $v(t)$'s direction will change to be closer to that of each sample.

The convergence of $v(t)$ depends on the direction where the "force of pulling" is the most. When the dot product is expanded, the magnitude of $b(t)$ becomes $\|u(t)^2\|cos(\theta)$, where θ is the angle between $u(t)$ and $v(t-1)$. As $\|u(t)^2\|$ is just a measure of variance for the samples, the direction where the force of pulling is the largest is the axis of most variance, or the first principal component.

The maximum amount of pulling changes depending on the weighting factors w_1 and w_2. If the amnesic function $\mu(t)$ is a constant, $v(t)$ will eventually converge in a statistically efficient way. If $\mu(t)$ is a function of time, as it is after the second switch-point, v can continue to shift in direction to adapt to new distributions for any time, and it does this with near, or quasi-optimal efficiency.

6.4.3 Convergence speed

The formula

$$E\|v(n)-v\|^2 = \sum_{t=1}^{n} w_t^2 E\|b\|^2 = \sum_{t=1}^{n} w_t^2 \text{ trace}(\Sigma_b) \tag{6.17}$$

gives expected estimation error as a function of number of samples n. It is a function of the amnesic error coefficient, how spread out the data is, as well as the dimensions, given

by trace(Σ_b), the product of the variances within the covariance matrix of b. Since the amnesic average is used, the right side will never truly reach zero. However, it constantly decreases as n increases, and does converge to zero as n goes to infinity. Its main use is to estimate the number of samples needed to get below an acceptable error bound.

6.4.4 Deriving lower-order eigenvectors

The preceding section only derived the first principal component. What about the other $k-1$? Recall that these vectors compose an orthogonal basis. So each component v_i from $i = 2, ..., k$ must be orthogonal to all previous components.

Orthogonality is enforced by use of residual vectors. Take the derivation of v_2 as an example. In order for v_2 to be orthogonal to v_1, first compute the projection vector of the sample onto v_1. Then subtract that vector from the sample, effectively deriving a new sample in a residual space. In the residual space, every vector will be orthogonal to v_1. Now, the *first* principal component in the residual space is equal to v_2.

For any number k of principal components, $k-1$ residuals are computed at each step. Denote $u_i(t)$ as the sample, possibly in a residual space, used when updating vector v_i. When $i = 1$, $u_1(t) = u(t)$, the original sample. When $i > 1$, u_i gives a residual vector.

First, project u_i onto v_i

$$y_i(t) = u_i(t) \cdot \frac{v_i(t)}{\|v_i(t)\|}. \tag{6.18}$$

Second, subtract the projection vector from u_i to obtain u_{i+1}.

$$u_{i+1}(t) = u_i(t) - y_i \frac{v_i(t)}{\|v_i(t)\|}. \tag{6.19}$$

This is done $k-1$ times per sample.

6.4.5 Link to developmental neuroscience

Hebb's principle of neural adaptation showed that a neuron would strengthen its existing connection to another neuron when both fired at the same time. In Long Term Potentiation, groups of neurons develop sensitivity to input connections corresponding to repetitive (common) input stimuli. Most neuroscientists believe that the adaptation described by Hebb's rule leads to this sensitivity.

In a computational neuron, sensitivity among the input dimensions is defined by its weights. A higher weight means the neuron is more sensitive to that input component. Inner product enforces this sensitivity. The inner product of an input sample and a neuron will be highest when their directions are the same, i.e., the relationships between the sample components are the same as those among the neuron's weights. Each neuron adapts by use of the learning rule in Equation 6.13. An input vector "pulls" the weight vector. By Hebb's rule, when the part of the input space that a neuron is sensitive to is high, the neuron is most likely to fire. The response of a computational neuron can be thought of the firing rate. If the same stimulus is presented over and over again, the firing rate (inner product response) of a single neuron will become larger and larger.

A neuro-computational concept of CCIPCA is as follows. Each neuron is a principal component detector, and each inhibits the neuron of next index (v_i influences v_{i+1}) through use of residuals. The competitive interactions between neurons are strictly enforced by Equation 6.19. Inhibition through subtraction is not based in any observations from neuroscience. It is hypothesized that early neural processing decorrelates the input [1], which CCIPCA does as well. But it is not known exactly how this is done, and there is no evidence of anything like residuals. Thus, CCIPCA is a Type-4 algorithm, since it is online and doesn't use any higher order statistics. It is not, however, a Type-5 (in-place) algorithm, since it requires an "overseer" to enforce the assignment of index to the neurons. However, an in-place, Hebbian-like learning method is used to compute the component in each neuron's input space (which may be a residual space).

6.5 Problems With PCA

There are four problems with PCA that limit its usefulness as the only feature extractor in developmental perception.

1. **Lower-Order Components are Constrained**. PCA derives an optimal basis in the least squared error sense, for any dimension less than the original dimension. As basis components, all vectors must be orthogonal to each other.

   The principal components can also be called the most expressive features (MEFs). Each is the best feature in terms of maximum covariances (components that are often active together), given the constraint of orthogonality to all previous components. The response of a sample to feature v_i will be largest when the sample matches the feature, since $y_i = u \cdot v_i$. So, principal components as features will detect what they represent. This is not necessarily good. As the index decreases, the components become more constrained, and they may not represent anything useful. A major part of PCA's theory (and how its optimality was defined) was minimizing reconstruction error. But we are not necessarily interested in exactly reconstructing the input, since representation does not have to be world-specific.

2. **PCA Extracts Global Features**.

   PCA cannot handle the case where there are multiple concentrations of correlated inputs. As an example, assume the samples fall into two well-separated clusters. The first eigenvector will converge to a direction between them, and may not correspond to any samples so far observed.

3. **PCA is Linear**.

   The basis made up of the principal components in a linear manifold. The most natural shape of the data may be on a non-linear manifold. For example, if the data lies on a spherical surface, the axis on which the data has the most variance is not a

line, but a curve. As a real world example of this: many maps represent the surface of Earth, which is a spherical object in 3D, as a 2D surface. The first component is the equator, the curved line which goes around the planet at its widest point. This leads to an "unfolding" of the surface in the 2D space. Two linear vectors, as used in PCA, would not be able to separate one side of the sphere from the other in 2D.

4. **CCIPCA is Not a Type-5 (In-Place) Algorithm** There is no evidence from neuroscience that anything similar to residual subtraction is used. Only the first principal component can be justified as developed by an in-place algorithm (by the mechanism of Hebbian learning). The lower-order components require a separate developer and are not in-place.

Chapter 7

Independent Component Analysis

This chapter briefly introduces the concepts of Independent Components Analysis, an approach that extracts sparse features. Independent Component Analysis [18] requires for the features to not only be uncorrelated, but also independent.

7.1 ICA Formulation

At every time instance $t = 1, 2, ...,$, a sample vector $\mathbf{s}$ is generated from n sources $\mathbf{s}(t) = (s_1(t), s_2(t), ..., s_n(t))^T$. The sources themselves are unknown, due to their being mixed by the mixing matrix $\mathbf{A}$, which is full-rank. The mixing matrix transforms each source vector into an observed (sensed) vector linearly at each time $\mathbf{x}(t) = \mathbf{A}\mathbf{s}(t)$. $\mathbf{x}(t)$ is an n-dimensional vector of observed signals. The goal of ICA is to find a linear transformation such that the original sources can be recovered.

The core assumption in ICA is that the original signals are independent. However, it is too difficult to extract truly independent components in practice, so a variety of other criteria are used, such as maximizing kurtosis or negentropy (non-Gaussianity), or mutual information (infomax), to name a few – see [18] for a survey.

7.2 ICA and Vision

The ICA model applies very well to blind-source separation, where the number of source signals is typically small and known. But this model is too strong for natural images, where the number of sources is not known, and may be much larger than the number of sensors (pixels) – in fact, the meaning of a "source" in vision is somewhat unclear.

The features derived by non-Gaussianity maximizing ICA methods when given natural images are not independent sources, but simply the most non-Gaussian components. Most of these correspond to localized edge filters [3]. Decomposing an image using these features leads to a sparse coding of the image, as most edges do not overlap. Sparse features are desirable in AMD, since the generalization capability of the system is increased, and the many weights equal to zero can be pruned and ignored, leading to fast performance.

However, most ICA algorithms are rather inefficient. FastICA is a Type-1 algorithm, while Extended Infomax is Type-2. These do not suit AMD implementations.

The CCI LCA technique [42] is a Type-5 ICA algorithm for natural data. It is based on principles of statistical efficiency and quasi-optimal representation. CCILCA functions as an ICA algorithm when the input is a super-Gaussian mixture [48]. It can be thought that the criteria maximized is that of sparseness, through use of a winner-take-all approach. Somewhat surprisingly, it outperforms the popular less restrictive, state-of-the-art ICA algorithms by a very wide margin.

CCI LCA will be the primary method used to derive features in the in-place learning system.

Chapter 8

Lobe Component Analysis

This chapter is adapted from Weng and Zhang [42] and Weng and Luciw [40]. The former introduced the concepts presented here, while the latter concerns the multi-layer version.

8.1 Lobe components

Atick and coworkers [1] proposed that early sensory processing decorrelates inputs. Weng *et al.* [43] proposed an in-place algorithm that develops a network that whitens the input. Therefore, we can assume that prior processing has been done so that its output vector $\mathbf{y}$ is roughly white. By white, we mean its components have unit variance and are pairwise uncorrelated. The sample space of a k-dimensional white input random vector $\mathbf{y}$ can be illustrated by a k-dimensional hypersphere.

A concentration of the probability density of the input space is called a lobe, which may have its own finer structure (e.g., sublobes). The shape of a lobe can be of any type, depending on the distribution. For non-negative input components, the lobe components lie in the section of the hypersphere where every component is non-negative (corresponding to the first octant in 3-D).

Given a limited cortical resource, c cells fully connected to input $\mathbf{y}$, the developing cells divide the sample space $\mathcal{Y}$ into c mutually nonoverlapping regions, called *lobe re-*

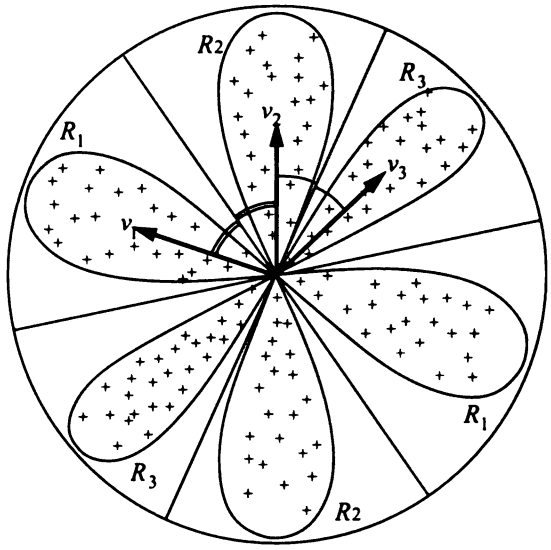


Figure 8.1: Three lobe components converge to the optimal representative prototype within the region of of each, from whitened input. In the unsigned version of LCA, each lobe component represents both positive and negative signals (figure courtesy of [42]).

gions:

$$\mathcal{Y} = R_1 \cup R_2 \cup ... \cup R_c, \tag{8.1}$$

(where $\cup$ denotes the union of two spaces). Each region R_i is represented by a single unit feature vector $\mathbf{v}_i$, called the *lobe component*. Given an input $\mathbf{y}$, many cells, not only $\mathbf{v}_i$, will respond. The response pattern forms a new population representation of $\mathbf{y}$.

Suppose that a unit vector (neuron) $\mathbf{v}_i$ represents a lobe region R_i. If $\mathbf{y}$ belongs to R_i, $\mathbf{y}$ can be approximated by $\mathbf{v}_i$ as the projection onto $\mathbf{v}_i$: $\mathbf{y} \approx \hat{\mathbf{y}} = (\mathbf{y} \cdot \mathbf{v}_i)\mathbf{v}_i$. Suppose the neuron $\mathbf{v}_i$ minimizes the mean square error $E\|\mathbf{y} - \hat{\mathbf{y}}\|^2$ of this representation when $\mathbf{y}$ belongs to R_i.

According to the theory of Principal Component Analysis (PCA) (e.g., see [20]), we know that the best solution of column vector $\mathbf{v}_i$ is the principal component of the conditional covariance matrix $\Sigma_{y,i}$, conditioned on $\mathbf{y}$ belonging to R_i. That is $\mathbf{v}_i$ satisfies $\lambda_{i,1}\mathbf{v}_i = \Sigma_{y,i}\mathbf{v}_i$.

Replacing $\Sigma_{y,i}$ by the estimated sample covariance matrix of column vector y, we

have

$$\lambda_{i,1}\mathbf{v}_i \approx \frac{1}{n}\sum_{t=1}^{n}\mathbf{y}(t)\mathbf{y}(t)^\top\mathbf{v}_i = \frac{1}{n}\sum_{t=1}^{n}(\mathbf{y}(t)\cdot\mathbf{v}_i)\mathbf{y}(t). \tag{8.2}$$

We can see that the best lobe component vector $\mathbf{v}_i$, scaled by "energy estimate" eigenvalue $\lambda_{i,1}$, can be estimated by the *average* of the input vector $\mathbf{y}(t)$ weighted by the linearized (without g) response $\mathbf{y}(t)\cdot\mathbf{v}_i$ whenever $\mathbf{y}(t)$ belongs to R_i. This average expression is crucial for the concept of optimal statistical efficiency discussed below.

8.2 Optimality

Suppose that there are two estimators Γ_1 and Γ_2, for a vector parameter (i.e., synapses or a feature vector) $\theta = (\theta_1, ..., \theta_k)$, which are based on the same set of observations $S = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$. If the expected square error of Γ_1 is smaller than that of Γ_2 (i.e., $E\|\Gamma_1 - \theta\|^2 < E\|\Gamma_2 - \theta\|^2$), the estimator Γ_1 is more statistically efficient than Γ_2. Given the same observations, among all possible estimators, the optimally efficient estimator has the smallest possible error. The challenge is how to convert a nonlinear search problem into an optimal estimation problem using the concept of statistical efficiency.

For in-place development, each neuron does not have extra space to store all the training samples $\mathbf{y}(t)$. Instead, it uses its physiological mechanisms to update synapses incrementally. If the i-th neuron $\mathbf{v}_i(t-1)$ at time $t-1$ has already been computed using previous $t-1$ inputs $\mathbf{y}(1), \mathbf{y}(2), ..., \mathbf{y}(t-1)$, the neuron can be updated into $\mathbf{v}_i(t)$ using the current sample defined from $\mathbf{y}(t)$ as:

$$\mathbf{x}_t = \frac{\mathbf{y}(t)\cdot\mathbf{v}_i(t-1)}{\|\mathbf{v}_i(t-1)\|}\mathbf{y}(t). \tag{8.3}$$

Then Eq. (8.2) states that the lobe component vector is estimated by the average:

$$\lambda_{i,1}\mathbf{v}_i \approx \frac{1}{n}\sum_{t=1}^{n}\mathbf{x}_t. \tag{8.4}$$

Statistical estimation theory reveals that for many distributions (e.g., Gaussian and exponential distributions), the sample mean is the most efficient estimator of the population mean (see, e.g., Theorem 4.1, p. 429-430 of Lehmann [27]). In other words, the estimator in Eq. (8.4) has nearly the minimum error given the observations.

8.3 Lobe components for nonstationary processes

The sensory environment of a developing brain is not stationary. That is the distribution of the environment changes over time. Therefore, the sensory input process is a nonstationary process too. We use the amnesic mean technique below which gradually "forgets" old "observations" (which use bad $\mathbf{x}_t$ when t is small) while keeping the estimator quasi-optimally efficient.

The mean in Eq. (8.4) is a batch method. For incremental estimation, we use what is called an amnesic mean [43].

$$\bar{x}^{(t)} = \frac{t-1-\mu(t)}{t}\bar{x}^{(t-1)} + \frac{1+\mu(t)}{t}x_t \tag{8.5}$$

where $\mu(t)$ is the amnesic function depending on t. If $\mu \equiv 0$, the above gives the straight incremental mean. We adopt a profile of $\mu(t)$:

$$\mu(t) = \begin{cases} 0 & \text{if } t \le t_1, \\ c(t-t_1)/(t_2-t_1) & \text{if } t_1 < t \le t_2, \\ c + (t-t_2)/r & \text{if } t_2 < t, \end{cases} \tag{8.6}$$

in which, e.g., $c = 2, r = 10000$. As can be seen above, $\mu(t)$ has three intervals. When t is small, straight incremental average is computed.

8.4 Single-layer updating algorithm

We model the development (adaptation) of an area of cortical cells (e.g., a cortical column) connected by a common input column vector $\mathbf{y}$ by the following Candid Covariance-free Incremental Lobe Component Analysis (CCI LCA) (Type-5) algorithm, which incrementally updates c such cells (neurons) represented by the column vectors $\mathbf{v}_1^{(t)}, \mathbf{v}_2^{(t)}, ..., \mathbf{v}_c^{(t)}$ from input samples $\mathbf{y}(1), \mathbf{y}(2), ...$ of dimension k without computing the $k \times k$ covariance matrix of $\mathbf{y}$. The length of the estimated $\mathbf{v}_i$, its eigenvalue, is the variance of projections of the vectors $\mathbf{y}(t)$ onto $\mathbf{v}_i$. The output of the layer is the response vector $\mathbf{z} = (z_1, z_2, ..., z_c)$. The quasi-optimally efficient, in-place learning, single layer CCI LCA algorithm $\mathbf{z} = \text{LCA}(\mathbf{y})$ is as follows:

1. Sequentially initialize c cells using first c observations: $\mathbf{v}_t^{(c)} = \mathbf{y}(t)$ and set cell-update age $n(t) = 1$, for $t = 1, 2, ..., c$.

2. For $t = c + 1, c + 2, ...,$ do

   (a) If the output is not given, compute output (response) for all neurons: For all i with $1 \leq i \leq c$, compute response:

   $$z_i = g_i \left(\frac{\mathbf{y}(t) \cdot \mathbf{v}_i^{(t-1)}}{\|\mathbf{v}_i^{(t-1)}\|} \right), \tag{8.7}$$

   where g_i is a neuron-specific sigmoidal function. A simple version of g_i is a linear function with under- and over-saturation points at a distance of a few standard deviations away from the mean of the input.

   (b) Simulating lateral inhibition, decide the winner: $j = \arg\max_{1 \leq i \leq c}\{z_i\}$, using z_i as the belongingness of $y(t)$ to R_i.

   (c) Update only the winner neuron v_j using its temporally scheduled plasticity:

   $$\mathbf{v}_j^{(t)} = w_1 \mathbf{v}_j^{(t-1)} + w_2 l_j \mathbf{y}(t),$$

where the scheduled plasticity is determined by its two age-dependent weights:

$$w_1 = \frac{n(j) - 1 - \mu(n(j))}{n(j)}, w_2 = \frac{1 + \mu(n(j))}{n(j)},$$

with $w_1 + w_2 \equiv 1$. Update the number of hits (cell age) $n(j)$ only for the winner: $n(j) \leftarrow n(j) + 1$.

(d) All other neurons keep their ages and weight unchanged: For all $1 \leq i \leq c$, $i \neq j$, $\mathbf{v}_i^{(t)} = \mathbf{v}_i^{(t-1)}$.

The neuron winning mechanism corresponds to the well known mechanism called lateral inhibition (see, e.g., Kandel *et al.* [21] p. 4623). The winner updating rule is a computer simulation of the Hebbian rule (see, e.g., Kandel *et al.* [21] p.1262). Assuming the plasticity scheduling by w_1 and w_2 are realized by the genetic and physiologic mechanisms of the cell, this algorithm is in-place. Alternatively, we use "soft winners" where multiple top (e.g., top-k) winners update.

8.5 LCA is ICA for super-Gaussians

Source components that have a super-Gaussian distribution correspond to lobe components we defined here. Each linear combination of k super-Gaussian independent components correspond to symmetric lobes, as shown in Fig. 8.1. Therefore, if the source components are all super-Gaussian, finding lobe components by CCILCA is equivalent to finding the independent components.

Natural data, such as natural images, were shown to be a mixture of super-Gaussian sources [13]. In AMD, the robot runs in a real-world setting, so the input will be primarily natural data.

LCA can be thought of as a quasi-optimally efficient ICA algorithm for natural data. It is also surprisingly effective using high-dimensional data. In [44], a comparison of

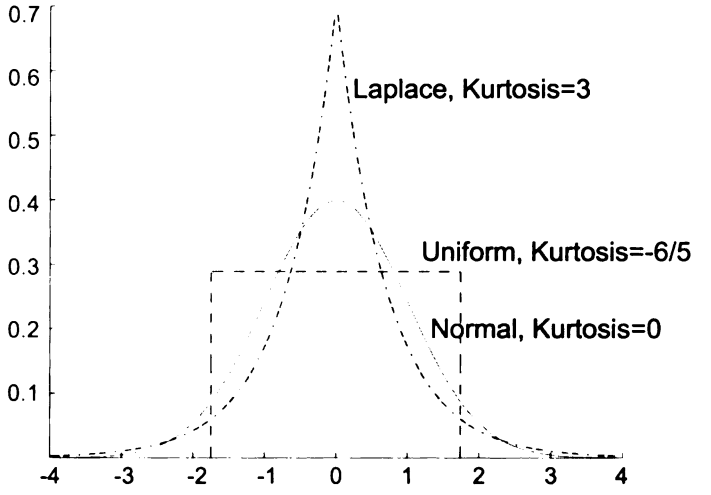


Figure 8.2: Components extracted by LCA are super-Gaussian, having a high kurtosis. The statistics of natural data, such as natural images, have been shown to be mixtures of super-Gaussian edge components. (figure courtesy of [42]).

convergence speed was done between LCA and two of the most popular ICA algorithms (FastICA [19] and Extended Infomax [26]). The original signals were generated from 100-dimensional Laplacian distributions, and were mixed randomly. The three algorithms were then used to attempt recovery of the original signals. The result showed that the Extended Infomax algorithm, which is a Type-2 (block-incremental) algorithm did not even converge in the time allotted (it requires many more samples), and the Type-1 (batch) FastICA converged about 20 times slower than the Type-4 CCI LCA algorithm.

The proposed Type-4 LCA algorithm, operating under the most restrictive condition, out-performs the state-of-the-art Type-3, Type-2, and Type-1 algorithms by a remarkably wide margin. This is thought to be due to the statistical efficiency. The simple nature of the algorithm as compared to other ICA methods, which require optimization of high-order non-linear functions or density estimation, is also very attractive.

Chapter 9

The Multi-Layer In-Place Learning Network

The work for multi-layers of LCA, combined with learning by an imposed action (supervised learning), as presented in this chapter, was published by Weng and Luciw in [40] and Weng *et al.* in [39], with different properties of the network concentrated on in each. The former concerns the study of a limited number of quasi-optimal representations, the effect of sparse response, and the effect of cortical topography in developing invariance detecting neurons. The latter is primarily concerned with the invariance, and goes into more depth on the subject.

9.1 Introduction of the Network

This section presents the architecture of the new Multi-layer In-place Learning Network (MILN), shown in Fig. 9.1. For biological plausibility, assume that the signals through the lines are *non-negative* signals that indicate the firing rate. Two types of synaptic connections are possible, excitatory and inhibitory.

This is a recurrent network. For each neuron i, at layer l, there are three types of weights:

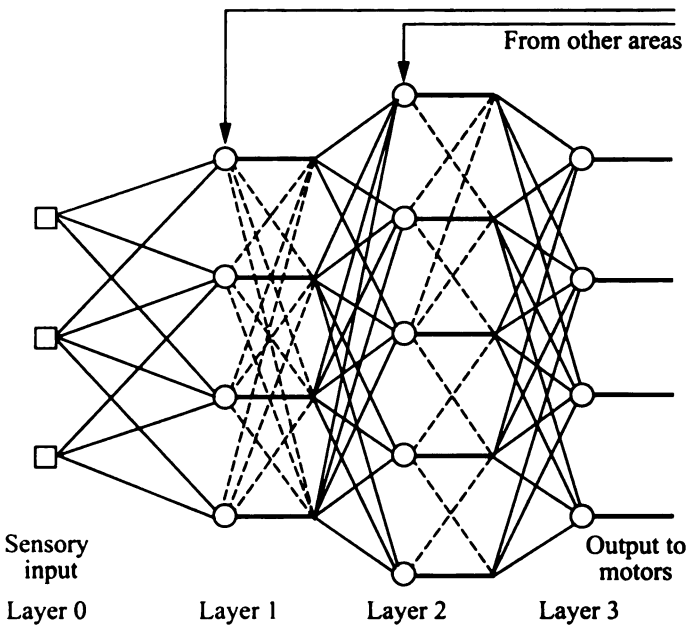


Figure 9.1: The architecture of the Multi-layer In-place Learning Networks. A circle indicates a cell (neuron). The thick segment from each cell indicates its axon. The connection between a solid signal line and a cell indicates an excitatory connection. The connection between a dashed signal line and a cell indicates an inhibitory connection. Projection from other areas indicates excitatory or inhibitory supervision signals (e.g., excitatory attention selection).

1. bottom-up (excitatory) weight vector $\mathbf{w}_b$ that links input lines from the previous layer $l-1$ to this neuron;

2. lateral (inhibitory) weight $\mathbf{w}_h$ that links other neurons in the same layer to this neuron.

3. top-down (excitatory or inhibitory) weight $\mathbf{w}_t$. It consists of two parts: (a) the part that links the output from the neurons in the next layer $l+1$ to this neuron. (b) The part that links the output of other layer processing areas (e.g., other sensing modality) or layers (e.g., the motor layer) to this neuron i. For notational simplicity, we only consider excitatory top-down weight, which selects neurons selected to increase their potential values. Inhibitory top-down connection can be used if the primary purpose is inhibition (e.g., inhibition of neurons that have not been selected by attention selection signals).

Assume that this network computes in discrete times, $t = 0, 1, 2, ...$, as a series of open-ended developmental experience after the "birth" at time $t = 0$. This network incorporates unsupervised learning and supervised learning. For unsupervised learning, the network produces an output vector at the output layer based on this recurrent computation. For supervised learning, the desired output at the output layer at time t is set (imposed) by the external teacher at time t.

9.1.1 Three types of projections

Consider a more detailed computational model of a layer within a multi-layer network. Suppose the input to the cortical layer is $\mathbf{y} \in \mathcal{Y}$, the output from early neuronal processing. However, for a recurrent network, $\mathbf{y}$ is not the only input. All the input to a neuron can be divided into three parts: bottom-up input from the previous layer $\mathbf{y}$ which is weighted by the neuron's bottom-up weight vector $\mathbf{w_b}$, lateral inhibition $\mathbf{h}$ from other neurons of the same layer corresponding which is weighted by the neuron's lateral weight vector $\mathbf{w_h}$, and the top-down input vector $\mathbf{a}$ which is weighted by the neuron's top-down weight vector $\mathbf{w_t}$. Therefore, the response z from this neuron can be written as

$$z = g(\mathbf{w}_b \cdot \mathbf{y} - \mathbf{w}_h \cdot \mathbf{h} + \mathbf{w}_t \cdot \mathbf{a}). \tag{9.1}$$

where g is its nonlinear sigmoidal function, taking into account under-saturation (noise suppression), transition, and over-saturation, and $\cdot$ denotes the dot production.

For digital computer, we simulate this analogue network through discrete times $t = 0, 1, 2, ...$ If the discrete sampling rate is much faster than the change of inputs, such a discrete simulation is expected to be a good approximation of the network behavior.

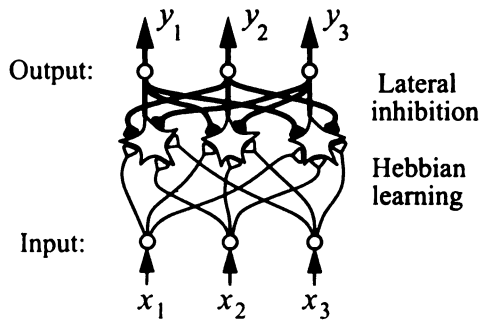


Figure 9.2: In Hebbian Learning, a neuron adapts to become sensitive to the part of the input signal that is present when it fires. In Lateral Inhibition, a neuron with high output suppresses the firing of other nearby neurons (figure courtesy of [39].

9.1.2 Lateral projection

Lateral inhibition is a mechanism of competition among neurons in the same layer. The output of neuron A is used to inhibit the output of neuron B, which shares a part of the receptive field, totally or partially, with A.

The net effect of lateral inhibition is that fewer winners can fire. We use a computationally more effective scheme to simulate lateral inhibition without resorting to iterations: Sort all the responses. Keep top-k responding neurons to have non-zero response. All other neurons have zero response (i.e., does not go beyond the under-saturation point).

9.1.3 Bottom-up projections

A neuron is updated using an input vector only when the (absolute) response of the neuron to the input is high. This is called Hebb's rule.

The rule of Hebbian learning is to update weights when output is strong. And the rule of lateral inhibition is to suppress neighbors when the neuron output is high.

At each layer of MILN, lobe component analysis, a Type-5 ICA algorithm that computes sparse components from natural data, is used.

9.2 Integrating LCA into MILN

As my main original theoretical contribution to this work, the two-layer formulation of MILN for classification and regression is introduced. This is. The input to the network comes from the sensory and motor (action) domains. Each input frame, starting at time $t = 0$, is concatenated into a vector, as in the appearance-based class of algorithms. Denote the sensory input vector at time t as $\mathbf{x}(t)$ and the motor input vector as $\mathbf{y}(t)$.

9.2.1 Input preprocessing

The LCA algorithm presented uses whitened input, which requires a position-normalized signal. Position normalization is done by keeping the means of $\mathbf{x}$ and $\mathbf{y}$ incrementally via amnesic average, then centering each signals at zero through subtraction.

The whitening procedure computes a *scaled* response of each sample along the principle component directions. Afterwards, the responses of each sample will not only be decorrelated, but also scale-normalized. Geometrically, whitening can be thought of a "sphering" process since afterwards the variance will be unit in all directions. Whitening is very beneficial to uncovering the true correlations within the data, since it does not allow derived features to be dominated by the larger components. However, the use of whitening changes the class of the LCA algorithm from Type-5 down to a Type-4 algorithm. The reason is the computation of the principle components. The scaling part is done in-place using a sigmoid, but in order to compute the principal components incrementally, as is done using the CCI PCA algorithm [43]), residual vectors are used to subtract the subspace of the first i principle component vectors from the sample so that the $i - 1_{th}$ component can be derived. Directing the subtraction of these residuals from the signal requires a separate developmental network, and so, if whitening is used, the algorithm is not in-place. We will address this issue further in future work.

After whitening, the input signals are (near) zero-mean and have (near) unit variance

in all directions. Due to the plasticity of amnesic average and a possibility of a changing source distribution, a guarantee of exact zero mean and unit variances is not possible. These signals are MILN's input.

9.2.2 Layers and weights

The first layer is the sensory-embedding layer. It computes n_x lobe components of the preprocessed $\mathbf{x}$, where n_x is the number of the neurons in this layer. The second layer is the motor-embedding layer, which finds n_y lobe components from $\mathbf{y}$. To learn sensory-motor correlations, there are $n_x n_y$ weights (initially) between the two, which are updated using LCA on the responses from sensory space, but driven by the motor control signal.

Denote the weights of the first layer (also called the input-layer) neurons as $\mathbf{W}^{(x)}$, where $\mathbf{w}_i$, the vector at column i is the weight vector of neuron i. Denote the weights of the neurons on the second, or output-layer as $\mathbf{W}^{(y)}$. These two matrices define the sensitivity of the neurons to to the sensors and the motors, respectively.

Denote the weights that connect the two layers as $\mathbf{W}^{(xy)}$ – if we are concerned with afferent input from the input-layer to the output-layer – or $\mathbf{W}^{(yx)}$ – in the context of feedback input from the output-layer to the input-layer. The former case are the "fan-out" weights from a particular input-layer neuron, while the latter are the "fan-in" weights to a particular output-layer neuron. Note that $\mathbf{W}^{(xy)}$ is the transpose of $\mathbf{W}^{(yx)}$.

When only $\mathbf{x}$ or $\mathbf{y}$ is given, the network learns correlations within each of the respective spaces (unsupervised learning) by updating the connections from the X-layer to the input space or the Y-layer to the output space. When both $\mathbf{x}$ and $\mathbf{y}$ are given, the network also learns the correlations between sensory and motor space (supervised learning) by updating the in-between connections.

9.2.3 Layer responses and weight updating

The response of each neuron is used to drive local competition, i.e., the highest responding neurons will update by Hebbian learning. Response is a combination of the neuron's sensitivities (weights) to excitation from the afferent input **a**, inhibition from the lateral input **l**, and excitation from the feedback input **f**. A sigmoid is used to provide too-low and too-high thresholding and response range control capabilities. Conceptually:

$$\mathbf{r}_i = g(\mathbf{W}_i^{(a)} \cdot \mathbf{a} - \mathbf{W}_i^{(l)} \cdot \mathbf{l} + \mathbf{W}_i^{(f)} \cdot \mathbf{f}) \tag{9.2}$$

In this work, lateral inhibition is not computed as a numeric value. Instead, only the neuron(s) with the largest response(s) will update (a "winners-take-all" updating rule). This is done to save computations, and to avoid iterations on the same input while waiting for the response to settle, as in [28].

The response of neuron i in the input-layer at time t is defined as

$$\mathbf{r}_i^{(x)} = g_1(\frac{\mathbf{W}_i^{(x)}}{\|\mathbf{W}_i^{(x)}\|} \cdot \frac{\mathbf{x}(t)}{\|\mathbf{x}(t)\|}(1 - \alpha_s) + \frac{\mathbf{W}_i^{(yx)}}{\|\mathbf{W}_i^{(yx)}\|} \cdot \frac{\mathbf{r}^{(xy)}(t-1)}{\|\mathbf{r}^{(xy)}(t-1)\|}(\alpha_s)). \tag{9.3}$$

where α_s is a parameter controlling the influence of the top-down supervision. Weight vectors will always be normalized by their lengths in competition, since it is undesirable that a weight vector with a larger magnitude dominant others who are more relatively sensitive to the input. The bottom-up and top-down excitations themselves are also normalized here for the same reason.

$$\mathbf{r}_i^{(y)} = g_2(\frac{\mathbf{W}_i^{(y)}}{\|\mathbf{W}_i^{(y)}\|} \cdot \mathbf{y}(t)). \tag{9.4}$$

is the response of neuron i in the output-layer from the motor input at time t and

$$\mathbf{r}_i^{(xy)} = g_3\left(\frac{\mathbf{W}_i^{(xy)}}{\|\mathbf{W}_i^{(xy)}\|} \cdot \hat{\mathbf{r}}^{(x)}(t)\right). \tag{9.5}$$

is the response of a output-layer neuron i from the input-layer response (multi-layer afferent excitation).

The LCA algorithm is used to find lobe components for each of the three weights and response sets. The winners are the neurons with the largest response. They update as in LCA.

9.2.4 Adaptive sigmoid

g_1, g_2, and g_3 are adaptive piecewise linear versions of the standard sigmoidal transfer function. In the linear version, the only parameters are two thresholds, θ_a and θ_b, which are used to control the output as follows. For simplicity, denote the input to each transfer function as x, and the output as y.

$$y = \begin{cases} 0 & \text{if } x \leq \theta_a, \\ (x - \theta_a)/(\theta_b - \theta_a) & \text{if } \theta_a < x < \theta_b, \\ 1 & \text{if } \theta_b \geq x, \end{cases} \tag{9.6}$$

The sigmoid is used to control the range of the responses (which can never be less than zero or greater than one), and to ensure they are non-negative.

The thresholds are set adaptively as follows

$$\theta_a = \bar{x} - T\sigma \tag{9.7}$$

$$\theta_b = \bar{x} + T\sigma \tag{9.8}$$

where $\bar{x}$ is the mean of the input and σ is its standard deviation. T is the tuning factor. Usually setting $T = 2$ is sufficient, but some signals with many components may

require a larger value. For the three response-computing sigmoid functions, there are three parameters to manually set, T_1, T_2, T_3.

9.2.5 Top-down influence

The existence of feedback connections in the cortex is well known, but their role is unclear. We use excitation from higher layers is used to supervise the development of a layer that does not take input directly from the sensors (is not provided a direct supervision signal).

Conceptually, the top-down signal reflects an expectation of what will be observed from the lower layer, and adjusts the response of that layer in order that the expectation is biased to be fulfilled. We conceive of two ways to generate that expectation. The first is by temporal locality, from the fact that similar inputs often occur sequentially in a non-synthetic sensory stream. Here, the top-down signal biases the lower level so that the same experience is more likely to occur next. The second is from an expectation directed by the higher areas, such as an attention selection signal.

The influence of top-down supervision in the current work is controlled by the parameter α_s, which is different for each layer. As α_s increases, the top-down supervision signal has a greater effect. When α_s is zero, the updating on a non-motor layer is totally unsupervised.

9.2.6 Top-"K" parameters for sparse representation

The winner-take-all LCA leads to a sparse coding of the input, meaning that for a given input, very few neurons will be active. It is thought that, due to receptive fields, early visual areas lead to a sparse coding of the input [2] [14]. In [48], it was shown that a winner-take-all updating method leads to the development of the receptive fields themselves. Sparse features are localized, in that they are sensitive to only part of the input, instead of monolithic (as is the standard in the appearance-based approach).

Practically, sparseness is a very desirable property for AMD. There will typically be

a very large number of weights because of the high-dimensional inputs, but most (more than 99% in our preliminary experiments) weights will converge to zero. These can be pruned for much faster performance.

Winner(s)-take-all updating is used wherever LCA is applied. The number of winners to update is set by the programmer. So, we require a parameter k, the number of top winners to update. In the two layer network, LCA updating occurs in multiple places, so multiple k's are needed. Define k as a parameter vector

$$\mathbf{k} = (k_x, k_Y, k_y, k_r, k_o) \tag{9.9}$$

where k_x is the number of winning units to update on the input layer and k_y is the number of winning units to update on the output layer, from the motor input. k_Y is the number of winning units to update on the output layer, from the afferent input.

k_r is a sparse response inducer. All but the top k_r values in the responses from the input layer are set to zero. This leads to sparse weights in $w_{x \to y}$, as the output layer neurons to learn to respond to only a few of their afferent inputs. Define $\hat{\mathbf{r}}^{(\mathbf{x})}$ as the response after only allowing the top k_r responding neurons in the first layer to fire.

k_o controls the smoothness of the output. When $k_o = 1$, only the top-responding output neuron is allowed to fire, generating quantized output, leading to more error. Increasing k_o allows more output neurons to fire, mixed by the strength of the response of each, and generates a more accurate output signal. Define $\hat{\mathbf{r}}^{(\mathbf{xy})}$ as the response after only allowing the top k_o responding neurons in the last layer to fire.

9.2.7 Network output

When the input is given, but the output is not (unsupervised), the output is computed by the network. The firing rate from the motor layer will give the classification or regression result. Each neuron's sensitivity (weights) to $\mathbf{y}$ is the output to be produced if it alone fires. We also use a population coding method where the responses of multiple motor neurons

are averaged to produce graded (smooth) output over a range of inputs. The output from the network is $\mathbf{z}$, which is defined as

$$\mathbf{z} = \sum_{i=1}^{n_y} \mathbf{W}_i^{(y)} \frac{\hat{\mathbf{r}}_i^{(xy)}}{\sum_{j=1}^{n_y} \hat{\mathbf{r}}_j^{(xy)}} \tag{9.10}$$

See Section 9.2.2 for an explanation of the notation used above.

9.2.8 Multiple-layer in-place learning

The following is the multi-layer in-place learning algorithm $\mathbf{z}(t) = \text{MILN}(\mathbf{x}(t))$. Suppose that the network has l layers.

MILN Learning: Initialize the time $t = 0$. Do the following forever (development), until power is off.

1. Grab the current input frame $\mathbf{x}(t)$. Let $\mathbf{y}_0 = \mathbf{x}(t)$.

2. If the current desired output frame is given, set the output at layer l, $\mathbf{y}_l \leftarrow \mathbf{z}(t)$, as given.

3. For $j = 1, 2, ..., l$, run the LCA algorithm on layer j, $\mathbf{y}_j = \text{LCA}(\mathbf{y}_{j-1})$, where the layer j is also updated.

4. Produce output $\mathbf{z}(t) = \mathbf{y}_l$; $t \leftarrow t + 1$.

Chapter 10

Experimental Results

10.1 Lobe Components of Natural Images

The first set of experiments was done to examine the properties of the LCA algorithm on real-world input. The thirteen images available at `http://www.cis.hut.fi/projects/ica/imageica/` were used as examples of real-world "natural" images, i.e., images whose statistics are representative of the signals we interpret through vision. For the input of each experiment, we incrementally and select a 16 x 16 pixel patch from a random location in a random image, and concatenate it into a column vector. One of the images used in the experiment is shown in Figure 10.1.

Figure 10.1: Example of a natural image used in the experiment.

10.1.1 LCA comparison

Earlier works such as [48] have already shown the result of LCA when given natural image input: localized orientation-selective filters that are functionally similar to those found in V1. By localized, we mean sparse, as most weights converge to zero. Thus, the receptive field of each neuron develops automatically.

In this experiment, we show the importance of the whitening process on the result. Recall that whitening is a preprocessing procedure that transforms the signal by projecting each sample along the principle components, but also adjusting by the scale of each. The whitened sample vector $\hat{\mathbf{x}}$ is computed from the original sample $\mathbf{x}$ as $\hat{\mathbf{x}} = \mathbf{W}\mathbf{x}$, where $\mathbf{W} = \mathbf{V}\mathbf{D}$ is the whitening matrix. $\mathbf{V}$ is the matrix where each principal component $\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_n$ is a column vector, and $\mathbf{D}$ is a diagonal matrix where the matrix element at row and column i is $\frac{1}{\sqrt{\lambda_i}}$, where λ_i is the eigenvalue of v_i.

Figure 10.2 shows the result when using LCA on 256 neurons and 1,500,000 whitened input samples. Each neuron's weight vector is identical in dimension to the input, so it can also be displayed as a 16 row and 16 column image. However, it must first be dewhitened in order to display properly. For example, to restore the original input vector, $\mathbf{x} = \mathbf{V}\mathbf{D}^{-1}\hat{\mathbf{x}}$, is the dewhitening procedure.

The filters in the image are ordered by the number of times each was the winner during the procedure. The component with the most wins is at the top left of the image grid, and it progresses through each row until the one with the least wins, at the bottom right. The number of wins per filter are shown in Figure 10.3. Observe that, after the first few, which won a relatively large amount of times, most filters won nearly the same amount of times. This group of filters are the localized edge detectors. The (small) group that won many times instead respond to global features.

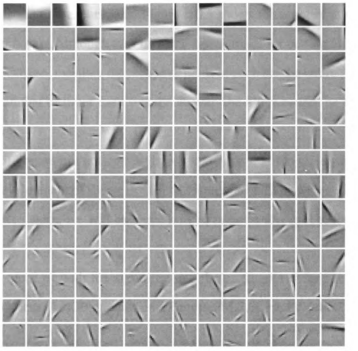


Figure 10.2: Lobe components from natural images (with whitening).

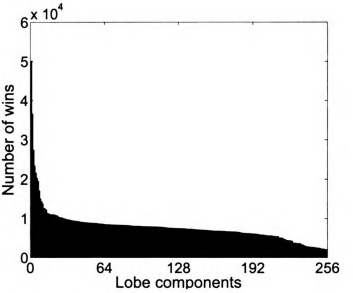


Figure 10.3: Number of wins per component when whitening was used.

Figure 10.4 shows the result when whitening is not used. The filters are ordered in the same way as the first experiment. Figure 10.5 shows the number of times each was the winner. In this case, although they do show a preference for certain orientations, most filters are *not* localized, meaning the receptive fields of each are the entire 16 x 16 window. And from the number of times each was the winner, the filters cannot be broken into two groups as before.

From the difference in the two results, we conclude that the whitening procedure is crucial to the development of sparse, localized features with LCA when using natural data as input. In future work, we will examine how to perform this procedure in an in-place way.

Due to the speed of both the algorithm and the programmed implementation, these experiments took less than 30 minutes on a Pentium M 2.0GHz PC with 1.0GB memory.

10.1.2 Topographic LCA

It is well-known that nearby neurons is the same cortical column respond to similar input [16]. Here, we place all neurons in a two-dimensional "cortical sheet". The updating rule is slightly different. Now, instead of only the winner updating, we model lateral excitation, or a neighborhood updating rule. A simple way of doing this is to update all neurons in a three by three grid, around the winner, weighted by their distance from the center. Note that we enforce the two amnesic averaging weights to still equal one. Figure 10.6 shows the result of 256 neurons after competing for 1,000,000 whitened samples.

Topographic updating leads to local variations of the same type of filter. Note that the unsigned version of LCA was used, where the absolute value of the response was taken before sorting. This is the reason why some nearby filters appear to be inverted.

10.1.3 Principal components as features

A popular approach in the appearance-based approach to computer vision is to use the principle components as the features of the data, e.g., as in [34]. Here we show the principal components of 16 x 16 natural image windows. Figure 10.7 shows the result of the first 48 principal components, displayed as images. Due to the constraints of orthogonality, the lower order components do not appear to represent anything in particular.

We conclude that principal components are mainly useful for linear dimensionality reduction, not as representative features.

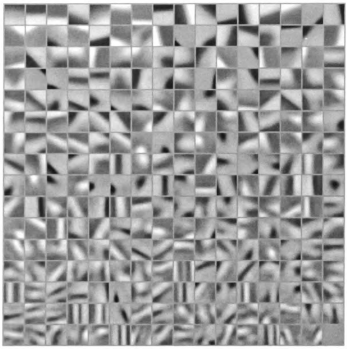


Figure 10.4: Lobe components from natural images (without whitening)

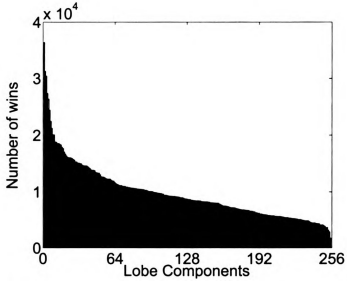


Figure 10.5: Number of wins per component when whitening was not used.

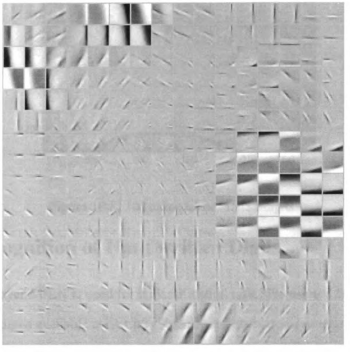


Figure 10.6: Topographic map of lobe components from natural images.

Figure 10.7: Principal components from natural images.

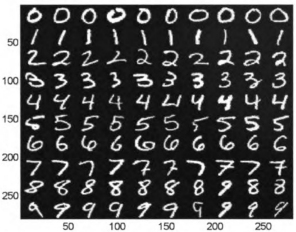


Figure 10.8: 100 examples of MNIST digits.

10.2 Recognition of Handwritten Digits

Here, the two-layer MILN is used for a classification task. We use the MNIST database of handwritten digits available at `http://yann.lecun.com/exdb/mnist/`. Each weight vector after updating is viewable as an image, and thus intuitively shows samples that contributed to it.

An attention selection mechanism, such as that of Zhang & Weng [47] will be incorporated into this network in our future study. Without an attention mechanism, we should not expect that the global match performed by the network will outperform other methods that perform local analysis (e.g., convolution using small templates [**?**]) given the same set of training samples. This is the first framework that we know of for in-place incremental cortical development. It is important to understand the properties of this network before applying attention selection mechanisms.

The MNIST data set consists of 70,000 images of the hand-written digits from 0 to 9, with 60,000 samples for training and 10,000 for testing. The 250 writers represented in the training set are disjoint from the 250 writers represented in the testing set (i.e., writer-disjoint). Each image is composed of $28 \times 28 = 784$ pixel intensity values, with a

grey and white digit over a black background. All images have already been translation-normalized, so that each digit resides at the center of the image. 100 examples of the MNIST digits are viewable in Figure 10.8.

Networks can be constructed with any number of neurons on any number of layers. In the following experiments, networks with only two-layers are used.

10.2.1 Limited cells

In the first experiment of this section, we test global prototyping capabilities of lobe components when the number of cells is limited. All the networks have n layer-1 neurons and 10 layer-2 neurons (for 10 classes of digits). During the supervised training session, we set the layer-2 output vector in the following way: When a sample of class i is given, $i = 0, 1, ..., 9$, the layer-2 output is such that the i-th neuron gives 1 and all other neurons give zero outputs. Therefore, lobe components in layer-1 serve as "prototypes" of a class. We used top-1 response in that only a single winner at each layer can fire. Thus, the layer-2 neurons determine which class the firing layer-1 neuron belongs to. We constructed networks with different numbers of n, to study the effect of varying number of "prototypes" in layer-1.

To compare with the effect of LCA updating, we also tested a simpler network called "Pure Initialization," where the weights of n layer-1 neurons were initialized by the sequentially arriving samples as in the LCA algorithm, but they are not updated any further when later samples arrive. The training set was pre-arranged so that the number of neurons representing each class were nearly equal. In the second network type "updated lobe components," the layer-1 lobe-components were further updated as specified by the LCA algorithm. When $n = 60,000$, both approaches will give the same result, since the number of layer-1 neurons is the same as the number of training samples. Both layers are trained at once using a single cycle through the samples.

A test sample can then be classified using the neuron index with the highest layer-2

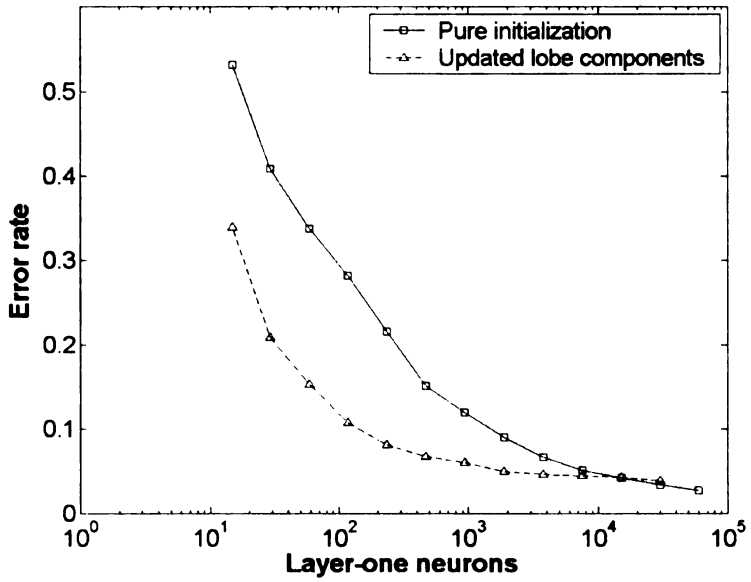


Figure 10.9: The effects of the limited number of layer-1 cells and the update of lobe components. In "Pure initialization," every weight vector in layer-1 is purely initialized by a single training sample and is not updated.

response. Performance is based on the error rate for all 10,000 test samples. Results are summarized in Fig. 10.9. It can be proved that both types of networks perform nearest-neighbor match in the inner-product space, using the lobe components in layer-1 as prototypes. The plot shows that the larger the number of prototypes, the smaller the error rate, in general. When $n = 60,000$, the number of training samples is the same as the number of prototypes, which is excessive for most applications. When n is reduced, the error rate also increases. However, "updated lobe components" give smaller errors, which shows that the initialization followed by updating is more effective than the pure initialization alone when the number of prototypes is smaller than the number of training samples. When n is reduced to $1,800$, about 33 samples per prototype, the error rate only increased slightly from $n = 60,000$.

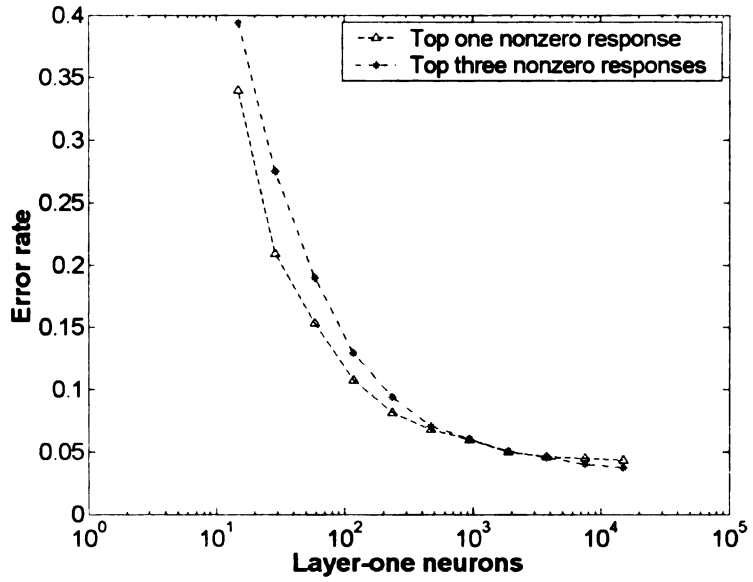


Figure 10.10: The effect of multiple responses versus the neuronal density.

10.2.2 Sparse coding

In this experiment, we study whether it will help to allow more neurons to fire. We used the same network structure as the first experiment, but all the lobe components in layer-1 are updated. In the first network type "top one nonzero response," only the top-one winner in layer-1 is allowed to fire. In the second type called "top three nonzero responses," the top-three responses from layer-1 were multiplied by a factor of 1, 0.66 and 0.33, respectively, and all other layer-1 neurons give zero response. The error rates for different numbers of layer-1 neurons are shown in Fig. 10.10. We know that top-3 responses give more information about the position of the input in relation with the top-three winning neurons. However, Fig. 10.10 showed that multiple responses did not help when the density of the prototypes is low. This is reasonable because bringing in far-away prototypes in decision making may add noise. This situation changed when the number of prototypes is sufficiently large so that the positions of nearby prototypes better predict the shape of the class boundaries indicated by the crossing point in Fig. 10.10.

Figure 10.11: The topographic map of layer-one neurons for the hand-written digit eight.

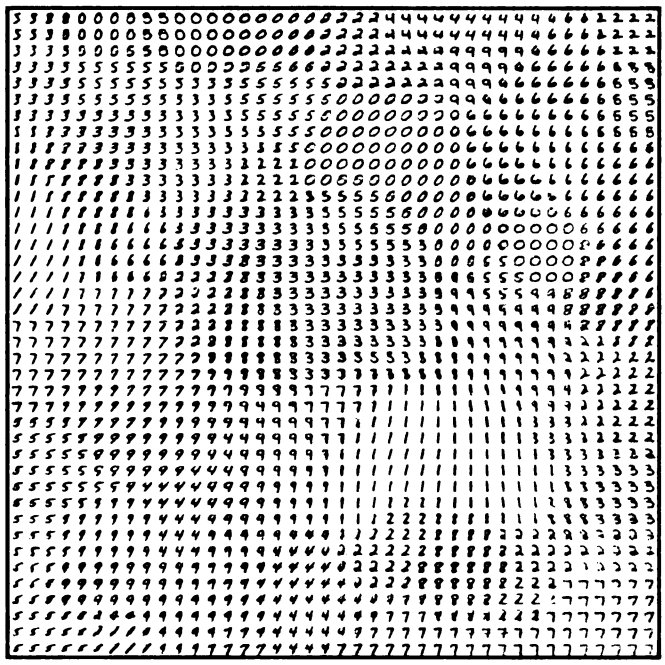
10.2.3 Topography and invariance

The next experiments focus on topographical cortical self-organization and its implication to within-class invariance. As observed in V1 and other cortical areas, nearby neurons have similar representations. This means nearby neurons represent nearby inputs or concepts. We place all layer-1 neurons in a two-dimensional, square grid, simulating a "cortical sheet." The winner neuron that updates for each input will also cause its nearby neurons within a distance of 2 from the winner to update as well, with their gain w_2 in the algorithm weighted by their distance to the winner, but $w_1 + w_2 = 1$ still holds. Fig. 10.11 shows an example with a 10×10 grid and all of the training samples were from the class "8." We can see that within-class variation of the digit 8 is represented by this cortical area, sampled by the discrete number of prototypes. The invariance for such within-class variation is achieved by the positive weights from all of these neurons to the 8-th layer-2 neuron.

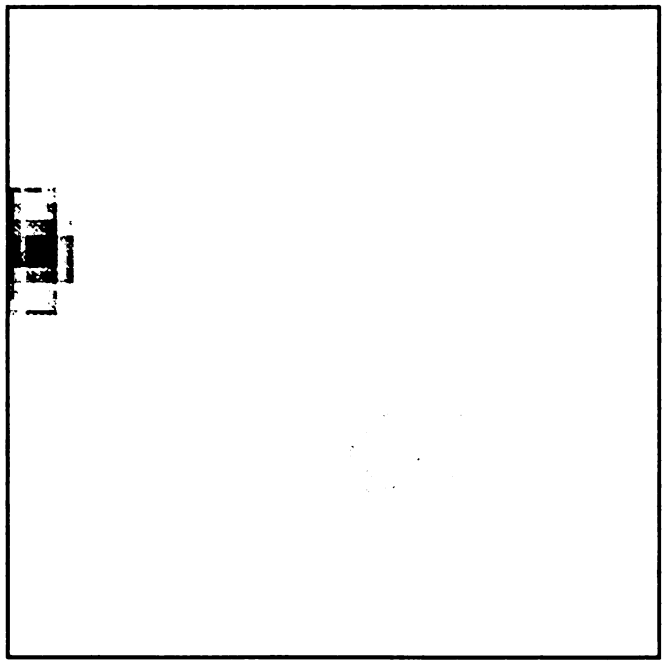
The above case does not include between-class competition for resources (neurons). To show the self-organization of resource for all 10 classes, Fig. 10.12(a) shows the layer-1 neurons in a 40×40 grid trained with all training samples. There are areas which can be seen to correspond to a particular class, since there is no guarantee that the area of a single

class is connected in the topographic map, which is also the case in biological networks (see, e.g., [35]). It depends on the size of neighborhood update, the inputs and degree of input variations.

Fig. 10.12(b) shows the learned weights of the layer-2 neuron corresponding to the digit-1 class. The darker the intensity, the larger the weight. As shown, invariance is achieved by the positive weights from the corresponding digit "1" neurons to the corresponding output neuron. Therefore, the within-class invariance shown at the output layer can be learned from multiple regions in the previous layer.



(a)



(b)

Figure 10.12: (a) The topographic map of layer-1 neurons. (b) Learned weights of the layer-2 neuron for the digit "1" class.

10.3 Regression Using Single Dimensional Input

In this section of experiments, we move from classification to a simple regression task. We choose a simple problem initially in order to work out the issues that arise in regression, and also to better understand the effect of the tunable parameters on the accuracy and efficient storage of the resulting weights.

The function to be approximated here is $y = sin(x)/x$, where x is in the input space and y is in the output space. x and y are one-dimensional signals. Sampling of x is uniform over the range of the function for all tests. The value (sampling) of samples in the output space is dependant on the output of the function.

The difference between classification and regression is that the values in the output space are no longer discrete (typically). This means that, now every possible output cannot be exactly stored. The output must now be represented in the same way that the input is – using a limited number of neurons as representative "prototypes" of the output space.

In the teaching phase, samples contain both an x and a y component: $S_i = (x_i, y_i)$, where S_i is the i_{th} training sample. We assign n_x layer-one neurons to the input space, and n_y layer-two neurons to the output space. A one-dimensional version of LCA is used to position the neurons from the samples. The closest layer-one neuron to x_i wins, and adapts to better represent x_i. In the same way, the closest neuron to y_i wins, and adapts to better represent y_i. The positions of the neurons are displayed as triangles in this section's figures, as in 10.13.

The function itself is approximated by learning sensory-motor neuron correlations. There are $n_x n_y$ weights from the input layer to the output layer (each neuron in the output space has a n_x dimensional weight vector). When a supervised sample is given, the closest neuron in the motor space to y adapts its weight vector to the layer-one response from x.

Recall that $\mathbf{k} = (k_x, k_y, k_Y, k_r, k_o)$ is the top-"k" parameter vector for a two-layer network. In these experiments, we set $k_x = k_y = k_Y = 1$, meaning winner-take-all updating is used in the three applications of LCA. k_r is the number of neurons in the first

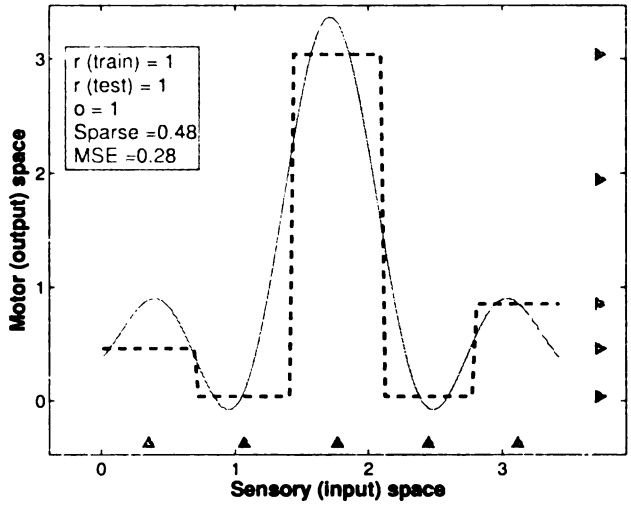


Figure 10.13: Using only 10 neurons to approximate the function $y = sin(x)/x$. The positions of neurons in the input-space are displayed as the lower triangles, and the neurons positioned in the output space are displayed at the triangles on the right. A small number of neurons relative to the complexity of the function leads to large quantization errors.

layer that are allowed to fire – the response of the rest are set to zero. k_o is the number of neurons in the second (output) layer that are allowed to fire. The averaged positions of these neurons in motor space gives z, the network's output.

Figure 10.13 shows a result when $n_x = 5$, $n_y = 5$, $k_r = 1$, and $k_o = 1$. The true function is viewable as the solid line. The approximated result after several epochs of training is the dotted line. Noted on each figure, "MSE" is the mean-square error of the approximation from the true function. And "Sparse" is the sparseness measure of the input-to-output layer weights. It is the percent of these weights that are equal to zero. Sparseness is desirable in AMD for efficient storage and speed – weights that converge to zero do not have to be stored and do not matter in any computations.

The obvious source of error in the first figure is attributable to quantization in both the input and output space. To reduce the x-quantization error, we increase n_x to 33 – on average, three samples per neuron. A result is displayed in Figure 10.14. The error decreases, and sparseness increases since there are more weights. But the y-quantization error remains.

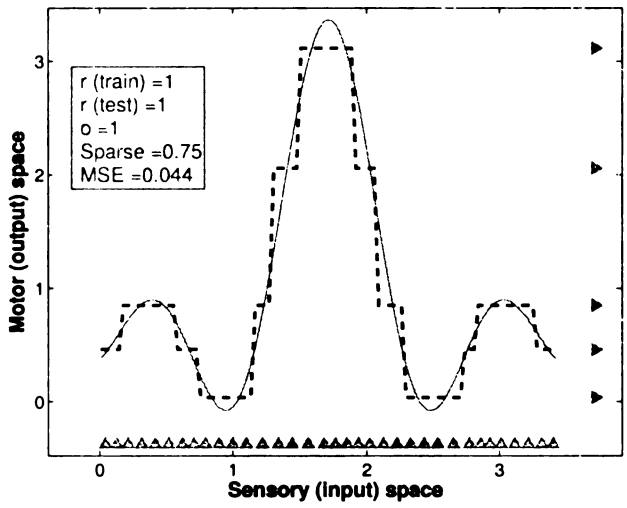


Figure 10.14: Increasing the number of input-layer neurons is helpful, but y-quantization error remains.

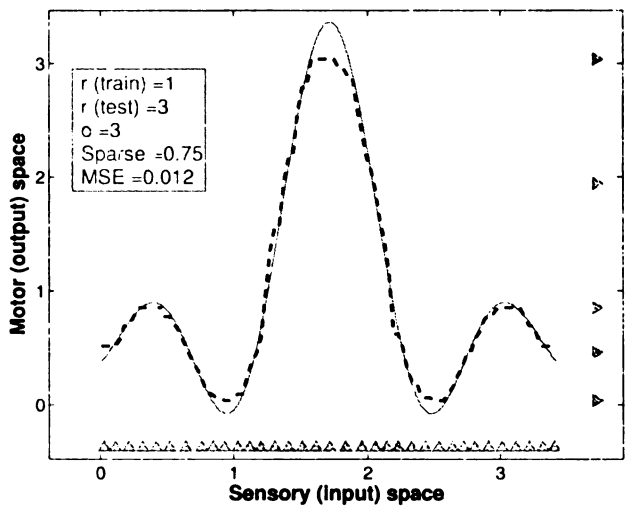


Figure 10.15: Output smoothing helps reduce the y-quantization error, but the number of neurons in the output layer provides insufficient coverage.

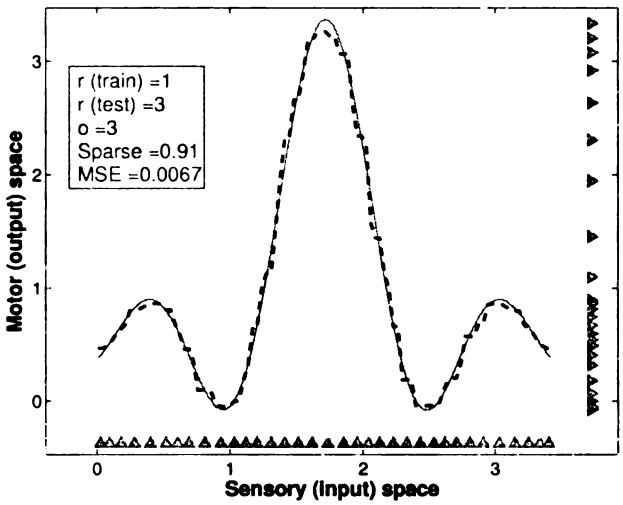


Figure 10.16: Increasing the number of output-layer neurons is helpful in providing greater coverage, e.g., at the center peak of this function.

In the third experiment, we now set the output smoothing parameter k_o to three, and k_r to three during the testing phase. Figure 10.15 shows a result. The error decreases without increasing the number of neurons or decreasing sparseness. However, another source of possible error becomes evident. As averages of samples, the neurons will tend to move away from the edges of the sample space. This leads to errors on the edges, as is the case with the samples that correspond to the center peak in the y-space, and the samples on the far left and right in the x-space. In Figure 10.16, we increase n_y so that enough neurons are available to approximate the function at the peak. Sparseness increases again, since the total number of weights increases.

In general, keeping k_r low during training leads to a sparse solution. But it may be at the cost of accuracy, depending on the complexity of the function. In [40], it was shown that increasing k_r during training in a high-dimensional classification task was beneficial when there were many neurons, so that the close neurons were all members of the same class. Otherwise, the neurons that responded only served as distracter points (noise). This is a very simple function, so increasing k_r during training was not beneficial. Ideally, we would find a solution that has few non-zero weights, but also of acceptable accuracy.

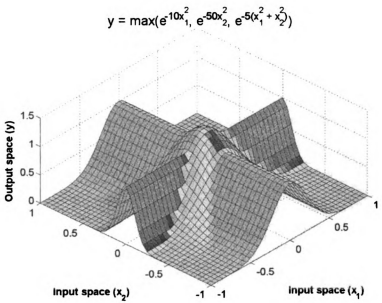


Figure 10.17: The function to be approximated.

10.4 Regression Using Two-Dimensional Input

The next experiments involve the approximation of a more interesting function. The "cross" function involves a two-dimensional input vector $\mathbf{x} = (x_1, x_2)^t$ and a one-dimensional output value y. It is defined as

$$y = \max\{e^{-10x_1}, e^{-50x_2}, e^{-5(x_1+x_2)}\}; \tag{10.1}$$

The range of the input is from [-1 -1] to [1 1]. The range of the output is from -2 to 2.

The details of the different tests are briefly summarized in Table 10.2. Constant parameters for all tests are as follows. In all tests, the neurons in the second layer were set to quantize the output space by .05, so n_y was set to 80.

For the top-k parameters, $k_x = k_y = k_Y = k_r = 1$ for all tests.

Table 10.1: Summary of all tests using the cross function

Experiment number	Training	Testing
(1)	uniform grid (41 x 41)	uniform grid (41 x 41)
(2)	uniform grid (41 x 41)	uniform grid (82 x 82)
(3)	uniform grid (41 x 41)	uniform grid (82 x 82)
(4)	uniform – subset	uniform grid (82 x 82)
(5)	uniform	uniform grid (82 x 82)
(6)	uniform grid (41 x 41)	uniform grid (41 x 41)
(7)	single gaussian	single gaussian

Table 10.2: Summary of parameters and results.

Experiment number	n_x	k_x	k_r	k_o	RMSE	Sparseness
(1)	41^2	1	1	1	$1.4e^{-3}$	0.99
(2)	41^2	1	1	1	$3.7e^{-2}$	0.99
(3)	41^2 1	1	2+ (dynamic)	n_y	$7.0e^{-}3$	0.96
(4)	41^2	1	1	1	$2.1e^{-2}$	0.99
(5)	41^2	5	1	1	$3.4e^{-3}$	0.91
(6)	$\lfloor(41^2)/10\rfloor$	5	2+ (dynamic)	n_y	$1.2e^{-2}$	0.70
(7)	$\lfloor(41^2)/10\rfloor$	5	2+ (dynamic)	n_y	$4.1e^{-3}$	0.73

10.4.1 Approximation with a neuron grid

The first test is meant to provide a starting point. The neurons in the first layer are placed on a 41 by 41 grid, at equal distances from one another. Thus, there are 41^2 neurons. There are also 41^2 training and testing samples, placed in the same locations as the neurons.

The weights were trained over five iterations. The order of the training samples were permuted before each iteration of the main loop. The result is shown in (1) of Figure 10.19. Note that the mesh plot does not show the results at all areas in the input space. Only the values at the testing locations are given, and the plotting function interpolates in-between. In this first simple test, there appears to be no difference from the result and the actual function. And as expected, the only error in the approximation result is attributable to quantization in y-space.

Now, we double the density of the testing samples and test over an 82 by 82 grid. Samples that are in-between neurons in the input space will be given the value associated with the nearest, since k_o was set to one. The result resembles the "stepwise" result in Figure 10.13, but in two dimensions. It is shown in (2) of Figure 10.19.

In the same way as in the one-dimensional case, this error is attributable to quantization in the output space. This error can be reduced by interpolation, i.e., smoothing in the output layer. To interpolate the approximated function for samples in-between neuron locations, k_o must be set to a value greater than one.

In the third test, a method to dynamically change k_r is used. We set a response threshold to the x part of the sample based on the top two responding neurons. If their responses are r_1 and r_2, respectively, we set the threshold at $r_2 - (r_1 - \frac{r_1 + r_2}{2})$. All responses below this threshold are set to zero. The reason this is done is because of symmetry. If a sample falls exactly in between two neurons, and k_r is set to one, then one neuron will be abnormally ignored.

Now, in testing, k_o is set to the number of output neurons. The result is shown in part (3) of Figure 10.19. It looks smooth compared to the previous. In the table it can be seen

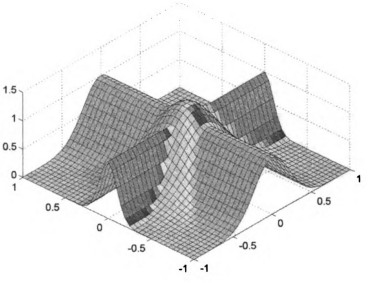


Figure 10.18: Approximation results when the input-layer neurons are placed on a grid. In (1), the training and testing samples both lie exactly on the neuron grid, and the approximation appears flawless.

that the error is nearly halved when smoothness is used, but sparseness (and thus, speed in an AMD implementation) decreases.

10.4.2 Approximation using lobe components

The previous set of tests dealt with the reduction of y-quantization error through the top-k parameters. This experiment shows the effect of using lobe components on the x-quantization error. In the first case, we randomly select the positions of the 41^2 neurons in the first layer, and do not allow these to change. In the second, LCA (winner decided by Euclidean distance instead of inner product) is used to update the neurons over several epochs through the set of training samples. The resulting neuron positions are shown in Figure 10.20, where the top figure is the first result, and the bottom figure is the second. Obviously, using LCA updating will reduce the larger x-quantization errors. The error measured in the two tests reflects this, as the MSE of the second test is more than six

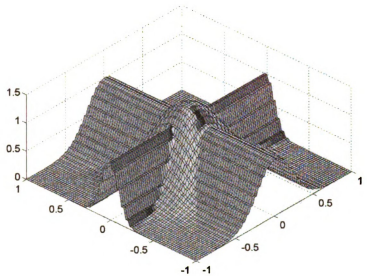
times better than the first.

10.4.3 Concentration of limited neurons

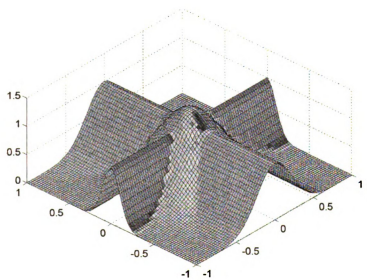
Here, the number of neurons in the input space is limited ($1/10$ of in the previous tests). The entirety of the **x**-space of the function cannot be well-represented using this number of lobe components, as in (6a) of Figure 10.21. However, if the neurons are concentrated densely in a single area, as in Figure 10.22, the approximation in that particular area becomes much better.

In both tests, the number of neurons was set to 168. The difference was the probability distribution of samples in the input space. In the first test, training and testing were done uniformly over the entire surface. In the second, training and testing samples were generated by a gaussian process, centered at $x = (0, 0)$ with a covariance matrix of $\Sigma = .05\ I$, where I is the identity matrix. The MSE of the second result was nearly three times better than that of the first.

In a real context, many places in the world will never be observed by an agent. The latter experiment can be interpreted in this way, where the true function is representative of the environment, and the approximation is representative of an agent's experience.

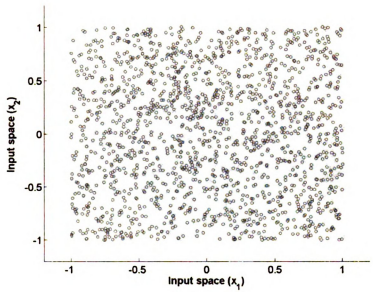


(2)

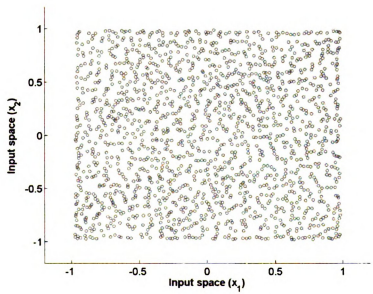


(3)

Figure 10.19: Approximation results when the input-layer neurons are placed on a grid. In (2), the accuracy of (1) is shown to be worse when testing is done in-between the neuron locations. In (3), this is somewhat remedied through use of smoothing in the output space.

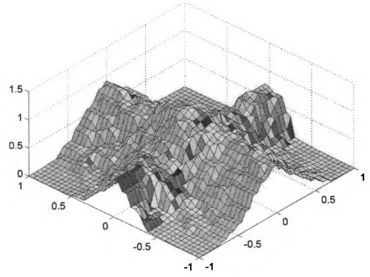


(4)

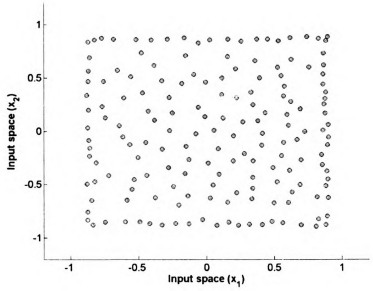


(5)

Figure 10.20: (4): Randomly placing input-layer neurons leads to larger possible x-quantization errors than in (5), where lobe component updating is used to represent the uniform input surface.

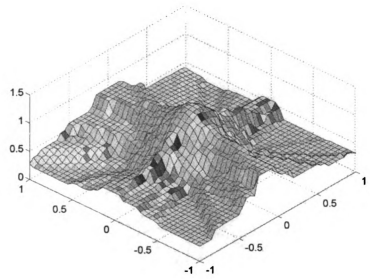


(6a)

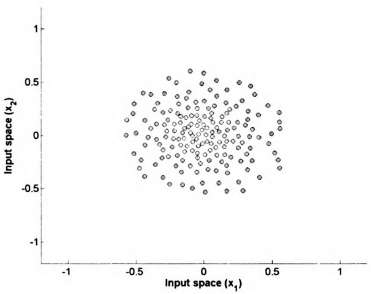


(6b)

Figure 10.21: Relatively few neurons cannot approximate the function well over the entire input space.



(7a)



(7b)

Figure 10.22: A concentration of few neurons can lead to good performance in one particular area. If the other areas are not experienced recently, there is no need to provide any resource to represent them.

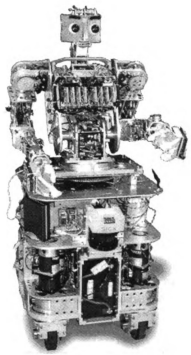


Figure 10.23. The SICK PLS laser range finder is mounted on the front of the Dav robot.

10.5 High-Dimensional Regression In a Robotic Application

This experiment is adapted from Zeng and Weng's work in [46], in which it was done using IHDR. Here, several two-layer in-place networks are used to judge the performance.

The Dav robot (Figure 10.23) is equipped with a SICK PLS laser range finder. It is mounted on the front of the robot, and is angled 3.8° downwards.

The range finder returns a vector of values corresponding to the distance to the closest objects 180° around the front of the robot. This vector $\mathbf{x} \in \mathcal{R}^{361}$, which is the input space, and $\mathbf{x}$ is quantized to set two values per degree. As in [46], we wish to use $\mathbf{x}$ to direct the reactive behavior of the robot so that it will avoid possible collisions when moving.

The movement of the robot can be controlled by a two-dimensional vector $\mathbf{y} = (y_1, y_2)$ where y_1 gives the heading direction in radians and y_2 gives the movement veloc-

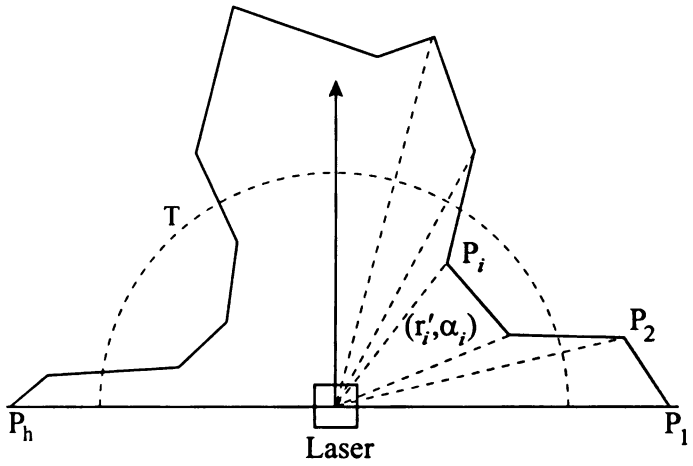


Figure 10.24: The pre-attentional map is approximated by a polygon P, represented by h endpoints. The post-attentional laser threshold is given by the half-circle T. Points that lie beyond this threshold are capped to the threshold distance. Points that lie within the threshold are returned normally. Figure courtesy of [46].

ity. The regression task in this problem is to learn the function $\mathbf{y}(t+1) = f(\mathbf{x}(t))$, where the action generated at time $t+1$ is best suited to avoid collisions with nearby obstacles.

The network was trained using supervised examples. Supervision was done in real-time on the robot using a training program. The current range map from the range finder was interpreted as an image, and the teacher's mouse click location on the image was interpreted as a velocity and direction. The $\mathbf{x}$ and $\mathbf{y}$ parts were coupled and saved as a training sample. In general, the robot was trained to move at a moderate speed and straight when no object was nearby. When an object was close, it was trained to slowly turn in a direction away from the object. This type of local reactive behavior can be used in parallel with purpose-driven movement, and can take over when an obstacle is nearby.

A two-layer network was trained and tested using 3926 samples previously collected from teaching, done by S. Zeng for the study in [46]. In each run, 75% of the samples were randomly selected for training and the remaining 25% were set aside for a disjoint test. The data contains samples from many different scenarios including interactions with single and multiple stationary obstacles, single and multiple moving obstacles (people), moving in a hallway (where the robot must avoid running into the walls), and moving in

a wider space.

As in [46], a preprocessing attention module was used on the range vector when an obstacle was nearby. The attentional mechanism implements a threshold on the distance returned by the range finder (think of it as a synthetic enclosure) when a small value is returned, meaning an object is nearby that should be attended to. This is to reduce the number of training samples needed for obstacle avoidance, since the surface beyond the threshold no impact on the robot's action when an obstacle is close – these cases should be classified the same.

We tested the performance using different numbers of input-layer neurons. The number of output-layer neurons was set to the number of unique actions (around 1000) in the dataset. The top-k parameters were $k_x = k_y = k_r = k_o = 1$ and $k_Y = 10$ (the number of sensory-motor weights to adjust) since there were a large number of output-layer neurons. Results are presented in Figures 10.25 and 10.26.

In testing, the program ran at nearly 30 frames per second, without pruning. Due to the sparseness of the solution, we expect after pruning, it will run much faster.

As of now, there is no way to tell whether the observed errors will lead to collisions in practice, as the system may quickly adjust when tested on a future sample before the collision occurs. There is no way to truly judge the collision avoidance capability without actually testing it on the robot, which will be done very soon.

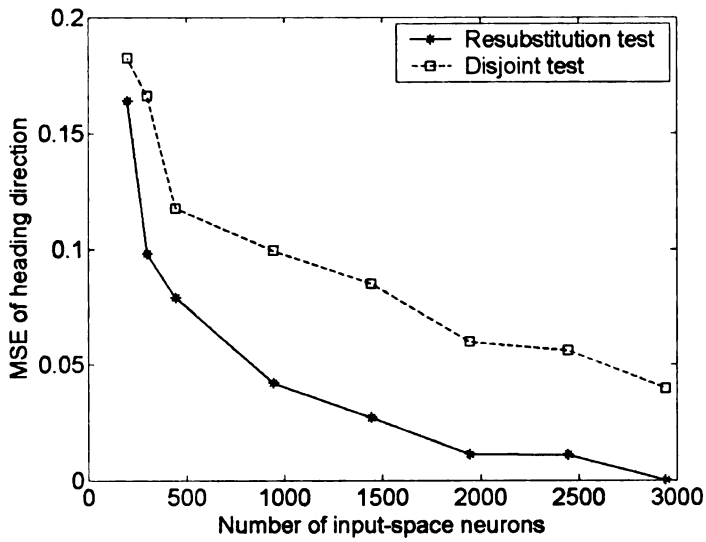


Figure 10.25: Error of the trained networks on the heading direction.

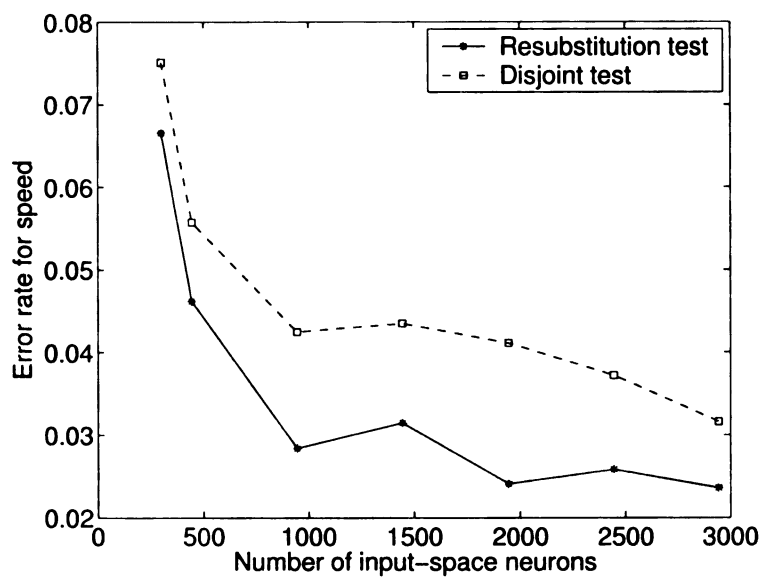


Figure 10.26: Error of the trained networks on the speed.

Chapter 11

Conclusions

11.1 Concluding Remarks

This thesis concerns the motivating factors and theory of a multi-layer, in-place network that integrates bottom-up unsupervised learning and top-down supervised learning. A two-layer version of this network was built and tested with a variety of experiments to empirically validate the theoretical properties, and to better understand the effect of the parameters.

This is the first multi-layer in-place learning network for general-purpose regression and classification with a near-optimal within-layer statistical efficiency. The experimental networks showed that the lobe components on each layer can automatically develop oriented edge filters with localized receptive fields from whitened natural input, and can learn the complex shape of the class manifolds in the layer's input (inner-product) space. Other experiments showed the two-layer networks can learn to perform regression by adapting to correlations within sensory-motor input, with lobe components representing each layer in a near-optimal way. Application for AMD was shown by verifying that a two-layer network could learn a real-world, high-dimensional regression problem accurately. Due to the sparseness of the solution, it is expected that the speed will be suitable for real-time use in a robotics application.

A plan for future work is to (1): study the extension of MILN to more than two layers, (2): compare the real-time capabilities of the program as with IHDR, (3): examine MILN's use for developmental vision in Dav and (4): study the possibility of its use as both a source and target for learning attention.

Appendix A

Proofs

In [42], in proving the near-optimality of amnesic average, the following two equations were presented. But the proofs were not done at the time, instead being left as an exercise. Here, I prove each.

A.1 Proof of Equation 5.11

The formula to calculate any amnesic weight w_t, $t = 1, 2, ..., n$ for any n

$$w_t(n) = \frac{1 + \mu(t)}{t} \prod_{j=t+1}^{n} \frac{j - 1 - \mu(t)}{j} \tag{A.1}$$

is here proved using induction on n.

- **Base Case**. $n = 1$, meaning only one sample has been observed. From equation A.1

$$w_1(1) = \frac{1 + 0}{1} = 1 \tag{A.2}$$

Since $\mu(1) = 0$ and the product on the right is not applicable when $t = n$.

- **Induction Hypothesis**. Now $n = k$. Assume that

$$w_t(k) = \frac{1+\mu(t)}{t} \prod_{j=t+1}^{k} \frac{j-1-\mu(t)}{j} \tag{A.3}$$

- **Induction Step**. Another sample is observed, so $n = k+1$. By equation 5.2, the weight of the newest sample is $(1+\mu(k+1))/(k+1)$. And from equation 5.3, the previous average $\bar{x}(k)$ is multiplied by $((k+1)-1-\mu(k+1)/(k+1)$. For purposes of this proof, denote this quantity as w_m.

  Since $\bar{x}(t-1)$ is a weighted sum of all samples observed at time 1 to k, each term can be multiplied by w_m as follows

$$\bar{x}(k) = w_m(w_1x_1 + w_2x_2 + ... + w_kx_k) = w_mw_1x_1 + w_mw_2x_2 + ... + w_mw_kx_k \tag{A.4}$$

  Finally, we substitute using the induction hypothesis. For any $t = 1, 2, ..., (k+1)$

$$\begin{aligned} w_t(k+1) &= w_t(k)w_m \\ &= \frac{1+\mu(t)}{t} \prod_{j=t+1}^{k} \frac{j-1-\mu(t)}{j} w_m \\ &= \frac{1+\mu(t)}{t} \prod_{j=t+1}^{k} \frac{j-1-\mu(t)}{j} \frac{(k+1)-1-\mu(k+1)}{k+1} \\ &= \frac{1+\mu(t)}{t} \prod_{j=t+1}^{k+1} \frac{j-1-\mu(t)}{j} \end{aligned}$$

A.2 Proof of Equation 5.12

This section proves that for any number of samples n, the sum of all amnesic weights w_t, $t = 1, 2, ..., n$ will be equal to 1

$$\sum_{t=1}^{n} w_t(n) = 1 \tag{A.5}$$

using induction on n.

- **Base Case.**

$$\sum_{t=1}^{1} w_t(1) = w_1(1) = \frac{1+\mu(1)}{1} = 1 \tag{A.6}$$

Using equation 5.11, and the fact that $\mu(1) = 0$.

- **Induction Hypothesis**

Assume that, for some number of samples $n = k$

$$\sum_{t=1}^{k} w_t(k) = 1. \tag{A.7}$$

- **Induction Step**

Another sample is observed and $n = k+1$. By equations 5.2 and 5.3, the weight of sample $k+1$ is $(1+\mu(k+1))/(k+1)$, and all the previous weights are multiplied by $((k+1)-1-\mu(k+1))/(k+1)$. Using the induction hypothesis, we can prove that the sum of the weights is still equal to one as follows

$$\begin{aligned}
\sum_{t=1}^{k+1} w_t(k+1) &= \frac{(k+1)-1-\mu(k+1)}{k+1}\left(\sum_{t=1}^{k} w_t(k)\right) + \frac{1+\mu(k+1)}{k+1} \\
&= \frac{(k+1)-1-\mu(k+1)}{k+1} + \frac{1+\mu(k+1)}{k+1} \\
&= \frac{k+1}{k+1}.
\end{aligned}$$

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] J. J. Atick and A. N. Redlich. Towards a theory of early visual processing. *Neural Computation*, 2:308–320, 1990.

[2] H.B. Barlow. Single units and sensation: A neuron doctrine for perceptual psychology? *Perception*, 1:371–394, 1972.

[3] A. J. Bell and T. J. Sejnowski. The 'independent components' of natural scenes are edge filters. *Vision Research*, 37(23):3327–3338, 1997.

[4] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.

[5] R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–160, 1991.

[6] N. Christianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods.* Cambridge University Press, Cambridge, UK, 2000.

[7] P. Comon. Independent component analysis, A new concept? *Signal Processing*, 36:287–314, 1994.

[8] International Human Genome Sequencing Consortium. Finishing the euchromatic sequence of the human genome. *Nature*, 431:931–945, 2004.

[9] M. J. Crowe, J.C. Bresnahan, S.L. Shuman, J.N. Masters, and M.S. Beattle. *Nature Medicine*, 3:73–76, 1997.

[10] G.E. Edelman. *Neural Darwinism. The theory of neuronal group selection.* New York, 1987.

[11] W. Erlhagen and G. Schoner. Dynamic field theory of movement preparation. *Psychological Review*, 3:545–572, 2002.

[12] S.E. Fahlman. The recurrent cascade-correlation architecture. In J. E. Moody and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems*, pages 190–196. Morgan Kaufmann, San Mateom CA, 1991.

[13] D.J. Field. Relations between the statistics of natural images and the response properties of cortical cells. *Journal of the Optical Society of America*, 4:2379–2394, 1987.

[14] D.J. Field. What is the goal of sensory coding? *Neural Computation*, 6:559–601, 1994.

[15] J. Hegde and D. C. van Essen. Selectivity for complex shapes in primate visual area v2. *Neuroscience*, 20:RC61, 2000.

[16] D. H. Hubel and T. N. Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *Journal of Physiology*, 160(1):107–155, 1962.

[17] W. S. Hwang and J. Weng. Hierarchical discriminant regression. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 22(11):1277–1293, 2000.

[18] A. Hyvarinen, J. Karhunen, and E. Oja. *Independent Component Analysis*. Wiley, New York, 2001.

[19] A. Hyvarinen and E. Oja. A fast fixed-point algorithm for independent component analysis. *Neural Computation*, 9(7):1483–1492, 1997.

[20] I. T. Jolliffe. *Principal Component Analysis*. Springer-Verlag, New York, 1986.

[21] E. R. Kandel, J. H. Schwartz, and T. M. Jessell, editors. *Principles of Neural Science*. McGraw-Hill, New York, 4th edition, 2000.

[22] T. Kohonen. *Self-Organizing Maps*. Springer-Verlag, Berlin, 3rd edition, 2001.

[23] J. Krichmar and G. Edelman. Machine psychology: autonomous behavior, perceptual categorization and conditioning in a brain-based device. *Cerebral Cortex*, 2002.

[24] S. L. Pallas L. von Melchner and M. Sur. Visual behavior mediated by retinal projections directed to the auditory pathway. *Nature*, 404:871–876, 2000.

[25] D Lee and S Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401:788–791, 1999.

[26] T. W. Lee, M. Girolami, and T. J. Sejnowski. Independent component analysis using an extended infomax algorithm for mixed sub-gaussian and super-gaussian sources. *Neural Computation*, 11(2):417–441, 1999.

[27] E. L. Lehmann. *Theory of Point Estimation*. John Wiley and Sons, Inc., New York, 1983.

[28] R. Miikkulainen, J. A. Bednar, Y. Choe, and J. Sirosh. *Computational Maps in the Visual Cortex*. Springer, Berlin, 2005.

[29] R. Pfeifer and C. Scheier. Sensory-motor coordination: the metaphor and beyond. *Autonomous Systems*, 20:157C178, 1997.

[30] M. I. Posner and M. E. Raichle. *Images of Mind*. Scientific American Library, New York, 1994.

[31] M. Rosenblum and Larry S. Davis. An improved radial basis function network for visual autonomous road following. *IEEE Trans. on Neural Networks*, 7(5):1111–1120, 1996.

[32] D. L. Swets and J. Weng. Using discriminant eigenfeatures for image retrieval. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 18(8):831–836, 1996.

[33] J. B. Tenenbaum, V. de Silva, and J. C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290:2319–2323, December 22 2000.

[34] M. Turk and A. Pentland. Eigenfaces for recognition. *Journal of Cognitive Neuroscience*, 3(1):71–86, 1991.

[35] X. Wang, M. M. Merzenich, K. Sameshima, and W. M. Jenkins. Remodeling of hand representation in adult cortex determined by timing of tactile stimulation. *Nature*, 378(2):13–14, 1995.

[36] J. Weng. Developmental robotics: Theory and experiments. *International Journal of Humanoid Robotics*, 1(2):199–235, 2004.

[37] J. Weng, G. Abramovich, and D. Dutta. Adaptive part inspection through developmental vision. *Journal of Manufacturing Science and Engineering*, 127(4):846–856, 2005.

[38] J. Weng and W.S. Hwang. Incremental hierarchical discriminant regression. *IEEE Trans. on Neural Networks*, 2006. accepted and to appear.

[39] J. Weng, H. Lu, T. Luwang, and X. Xue. In-place learning for positional and scale invariance. In *Proc. World Congress on Computational Intelligence*, Vancouver, Canada, July 16-21 2006.

[40] J. Weng and M.D. Luciw. Optimal in-place self-organization for cortical development: Limited cells, sparse coding and cortical topography. In *Proc. IEEE 5th International Conf. on Development and Learning (ICDL 2006)*, Bloomington, IN, May 30 - June 3 2006.

[41] J. Weng, J. McClelland, A. Pentland, O. Sporns, I. Stockman, M. Sur, and E. Thelen. Autonomous mental development by robots and animals. *Science*, 291(5504):599–600, 2001.

[42] J. Weng and N. Zhang. Optimal in-place learning and the lobe component analysis. In *Proc. World Congress on Computational Intelligence*, Vancouver, Canada, July 16-21 2006.

[43] J. Weng, Y. Zhang, and W. Hwang. Candid covariance-free incremental principal component analysis. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 25(8):1034–1040, 2003.

[44] Juyang Weng, Nan Zhang, and Raja Ganjikunta. Distribution approximation: An in-place developmental algorithm for sensory cortices and a hypothesis. Technical Report MSU-CSE-04-40, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, September 2004.

[45] L. Bengio Y. LeCun and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86:2278C2324., 1998.

[46] S. Zeng and J. Weng. Obstacle avoidance through incremental learning with attention selection. In *Proc. IEEE Conf. on Robotics and Automation*, New Orleans, Louisiana, April 26 - May 1 2004.

[47] N. Zhang and J. Weng. A developing sensory mapping for robots. In *Proc. IEEE 2nd International Conf. on Development and Learning (ICDL 2002)*, pages 13–20, MIT, Cambridge, Massachusetts, June 12-15 2002.

[48] N. Zhang and J. Weng. Sparse representation from a winner-take-all neural network. In *Proc. of International Joint Confernce on Neural Networks*, Budapest, Hungary, 2004.