5-2012

Deep Machine Learning with Spatio-Temporal Inference

Thomas Paul Karnowski
tkarnowski@gmail.com

To the Graduate Council:

I am submitting herewith a dissertation written by Thomas Paul Karnowski entitled "Deep Machine Learning with Spatio-Temporal Inference." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Electrical Engineering.

Itamar Arel, Major Professor

We have read this dissertation and recommend its acceptance:

Hairong Qi, Xiaobeng Feng, Husheng Li, Michael J. Roberts

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Deep Machine Learning with Spatio-Temporal Inference

A Dissertation Presented for
The Doctor of Philosophy
Degree
The University of Tennessee, Knoxville

Thomas Paul Karnowski
May 2012

Dedicated to my wife Dana Jo. Thank you for your support and understanding. I love you very much. I would also like to dedicate this work to our beloved children Eva, Nelson and Noah.

Acknowledgements

I would like to thank Derek Rose, Bobby Coop and Steven Young of the University of Tennessee - Knoxville Machine Intelligence Laboratory and Chris Boehnen, Ryan Kerekes and Ethan Farquhar of the Measurement Science and Systems Engineering Division of Oak Ridge National Laboratory for computer access to perform network formation experiments and the neural network evaluations. I would also like to thank the management team at Oak Ridge National Laboratory for support during these studies, including the Measurement Science and Systems Engineering Division Directory Kenneth W. Tobin, Jr., and Imaging, Signals and Machine Learning group leaders Shaun Gleason, and Philip Bingham, as well as Deputy Director for Science and Techology Thomas Zacharia: human resources personnel Saylor Hummel; and educational assistance manager Carolyn Ward. Finally, I would like to thank many others at ORNL who have helped through encouragement and anecdotes regarding their personal experiences with the PhD degree, including many who received their degree in abstentia or through ORNL support.

Believe deep down in your heart that you're destined to do great things.
Joe Paterno

A humble knowledge of thyself is a surer way to God than a deep search after learning.
Thomas Kempis

We have deep depth.
Yogi Berra

Abstract

Deep Machine Learning (DML) refers to methods which utilize hierarchies of more than one or two layers of computational elements to achieve learning. DML may draw upon biomemetic models, or may be simply biologically-inspired. Regardless, these architectures seek to employ hierarchical processing as means of mimicking the ability of the human brain to process a myriad of sensory data and make meaningful decisions based on this data. In this dissertation we present a novel DML architecture which is biologically-inspired in that (1) all processing is performed hierarchically; (2) all processing units are identical; and (3) processing captures both spatial and temporal dependencies in the observations to organize and extract features suitable for supervised learning. We call this architecture Deep Spatio-Temporal Inference Network (DeSTIN). In this framework, patterns observed in pixel data at the lowest layer of the hierarchy are organized and fit to generalizations using decomposition algorithms. Subsequent spatial layers draw upon previous layers, their own temporal observations and beliefs, and the observations and beliefs of parent nodes to extract features suitable for supervised learning using standard classifiers such as feedforward neural networks. Hence, DeSTIN is viewed as an unsupervised feature extraction scheme in the sense that rather than relying on human engineering to determine features for a particular problem, DeSTIN naturally constructs features of interest by representing salient regularities in the patterns observed. Detailed discussion and analysis of the DeSTIN framework is provided, including focus on its key components of generalization through online clustering and temporal inference. We present a variety of implementation details, including static and dynamic learning formulations, and function approximation methods. Results on standardized datasets of handwritten digits as well as face and optic nerve detection are presented, illustrating the efficacy of the proposed approach.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Deep Machine Learning

Humans are able to receive and process a myriad of sensory data that is both spatial and temporal in its nature and capture critical aspects of this data in a way that allows for its future use in a concise manner. Over 50 years ago, Richard Bellman, who introduced dynamic programming theory and pioneered the field of optimal control, asserted that high dimensionality of data is a fundamental hurdle in many science and engineering applications. The main difficulty that arises, particularly in the context of pattern classification applications, is that the learning complexity grows exponentially with linear increase in the dimensionality of the data. He coined this phenomenon the curse of dimensionality Bellman (1957). The typical approach of overcoming "the curse" has been to pre-process the data in a manner that would reduce its dimensionality to that which can be effectively processed, for example by a classification engine. This dimensionality reduction scheme is often referred to as feature extraction. As an example in an industrial process, defects in printed material which are inspected by machine vision systems process millions of pixels per second, but the reduction of this high dimensional data into meaningful information such as defective material, or high quality material, is only possible through highly engineered systems which are designed and trained to detect anomalies and their absence through specialized image processing and pattern recognition algorithms and software systems. Thus clearly the intelligence behind many pattern recognition systems is within the human-engineered feature extraction process, which at times can be challenging, highly application-dependent, and does not truly emulate the human decision process Duda et al. (2001). Moreover, if incomplete or erroneous features are extracted, the classification process is inherently limited in performance. In addition to the spatial dimensionality of real-life data, the temporal component also plays a key role. A sequence of patterns that we observe often conveys a meaning to us, whereby independent fragments of this sequence would be hard to decipher in isolation. Since we often infer meaning from events or observations that are received close in time, modeling the temporal component of the observations plays a critical role in effective information representation.

Capturing spatiotemporal dependencies, based on regularities in the observations, is therefore viewed as a fundamental goal for machine learning systems. Recent neuroscience findings have provided insight into the principles governing information representation in the mammalian brain, leading to new ideas for designing systems that represent information. One of the key findings has been that the neocortex, which is associated with many cognitive abilities, does not explicitly pre-process sensory signals, but rather allows them to propagate through a complex hierarchy Lee and Mumford (2003) of modules that, over time, learn to represent observations based on the regularities they exhibit Lee et al. (1998). This discovery motivated the emergence of the subfield of Deep Machine Learning (DML), which focuses on computational models for information representation that exhibit similar characteristics to that of the neocortex.

The very term "Deep Machine Learning" refers to the use of more than a few layers of processing elements in the formulation or implementation of a machine learning algorithm or set of algorithms Arel et al. (2010). From Bengio (2009), a deep architecture is defined as "composed of multiple levels of non-linear operations, such as in neural nets with many hidden layers or in complicated propositional formulae re-using many sub-formulae." The difficulty in truly implementing and developing a DML architecture lies in the learning aspect; solving an optimization problem so that the architecture can actually learn and correctly classify unknown vectors. Many deep learning approaches are biologically inspired, in that they draw upon biological models to create an analogous computational architecture. The degree of biological inspiration at some point becomes more biomimetic, where the computational architecture actually attempts to mimic the biological basis as it is currently understood. Thus there have been a variety of deep learning approaches and research directions proposed over the past decade. It is important to emphasize that each approach has strengths and weaknesses, depending on the application and context in which it is being used. Two leading methods, Convolutional Neural Networks (CNNs) and Deep Belief Networks (DBNs) and their respective variations are well established in the deep learning field. Other methods that draw more upon biomimetic approaches are also exciting fertile areas of current research.

DML has been applied to various domains with success such as handwriting, face, and audio recognition. However, the goal of DML is far beyond such task-specific applications. It should also be noted that despite the great opportunity offered by DML techniques, especially at improving machine learning and machine vision in general, some domain-specific tasks may not be directly improved by such schemes. An example is identifying and reading the routing numbers on the bottom of bank checks. Though these digits are human readable, they are comprised of restricted character sets which specialized readers can recognize flawlessly at very high data rates Rice et al. (1996). Similarly, iris recognition is not a task that humans generally perform; indeed, without training, one iris looks very similar to another at the level of human attention, yet engineered systems can produce matches between candidate iris images and an image database to high precision and accuracy to serve as a unique identifier for biometrics applications Newton and Phillips (2009). Finally, recent developments in facial recognition Osadchy et al. (2007) show equivalent performance relative to humans in their ability to match query images against large numbers of candidates, potentially matching far more than most humans can remember due to the massive memory storage capabilities of computers Adler and Schuckers (2007). Nevertheless, these remain highly specific cases and are the result of a lengthy feature optimization process as well as years of research that does not always map to other, more general applications. Furthermore, in principle deep learning systems can benefit from these same highly engineered features and potentially learn even higher levels of representations which engineered systems lack. Despite the myriad of open research issues and the fact that the field is still in its infancy, it is clear that advancements made with respect to developing DML systems will undoubtedly shape the future of machine learning in general.

1.2 Feature Extraction

Practical pattern recognition has relied on human engineered features since its inception and effective working systems often rely on these mechanisms. As an example, processing of fundus images to locate the optic nerve is an important step in automatic detection of retina disease. Some examples of retina images are shown in Figure 1.1, from the DiaretDB1 database of retina images Kauppi et al. (2007). Another example of a retina image is shown at the top of Figure 1.2; the optic nerve is the bright circular object on the right side of the image. As detailed in Tobin et al. (2007), a human engineered system for detecting the optic nerve takes the following steps, with human-engineered motivation, as illustrated in the middle and bottom of Figure 1.2. First, we note that the vascular tree seems to emanate out from the region of the optic disk. This suggests that regions where there are many vessels could have a high correlation with the presence of the optic nerve; regions where the vessels are thicker are correlated with optic nerve; and regions where the vessels have a strong vertical orientation are correlated with the optic nerve. The second observation pertains to the intensity of the optic nerve region. In many patients, the optic nerve is highly reflective and thus is more intense than other regions in the image. The final observation pertains to the optic nerve size; while variable from patient to patient, it has a definite shape and size which is variable from patient to patient but can be described well by an approximation.

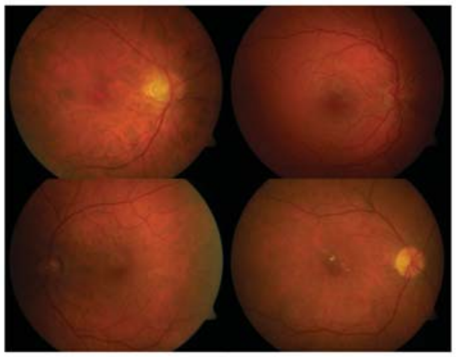


Figure 1.1: Examples of retina images from a public database of retina images (DiaretDB1, Kauppi et al. (2007)).

For true learning, however, autonomous systems must extract their own features and use them to learn the problems at hand with minimal or no human engineering intervention.

1.3 The Deep SpatioTemporal Inference Network (DeSTIN)

The paradigm of partitioning large data structures into smaller, more manageable units, and discovering the dependencies that may or may not exist between such units, is very promising. However, there remains a need for an architecture that can represent temporal information with the same ease in which spatial structure is discovered. Moreover, some key constraints are imposed on the learning schemes driving these architectures, such as the need for layer-by-layer training Erhan et al. (2010). This work proposes a novel architecture for deep learning that combines concepts from unsupervised learning for dynamic pattern representation together with Bayesian inference. We call this architecture "Deep Spatio-Temporal Inference Network" or DeSTIN Arel et al. (2009a), Arel et al. (2009b). The governing architecture yields a highly scalable modeling system which is capable of effectively dealing with high-dimensional signals. Spatiotemporal dependencies that exist within the observations are modeled inherently in an unguided manner. Each node models the inputs by means of clustering and simple dynamics modeling, while it constructs a belief state over the distribution of sequences using inference methods.

The architecture comprises of multiple instantiations of an identical processing node which populate all layers of the system. Each node is tasked with characterizing the sequences of patterns that are presented to it by nodes in the layer that precedes it. At the very lowest layer of the hierarchy nodes receive as input raw data (e.g. pixels of

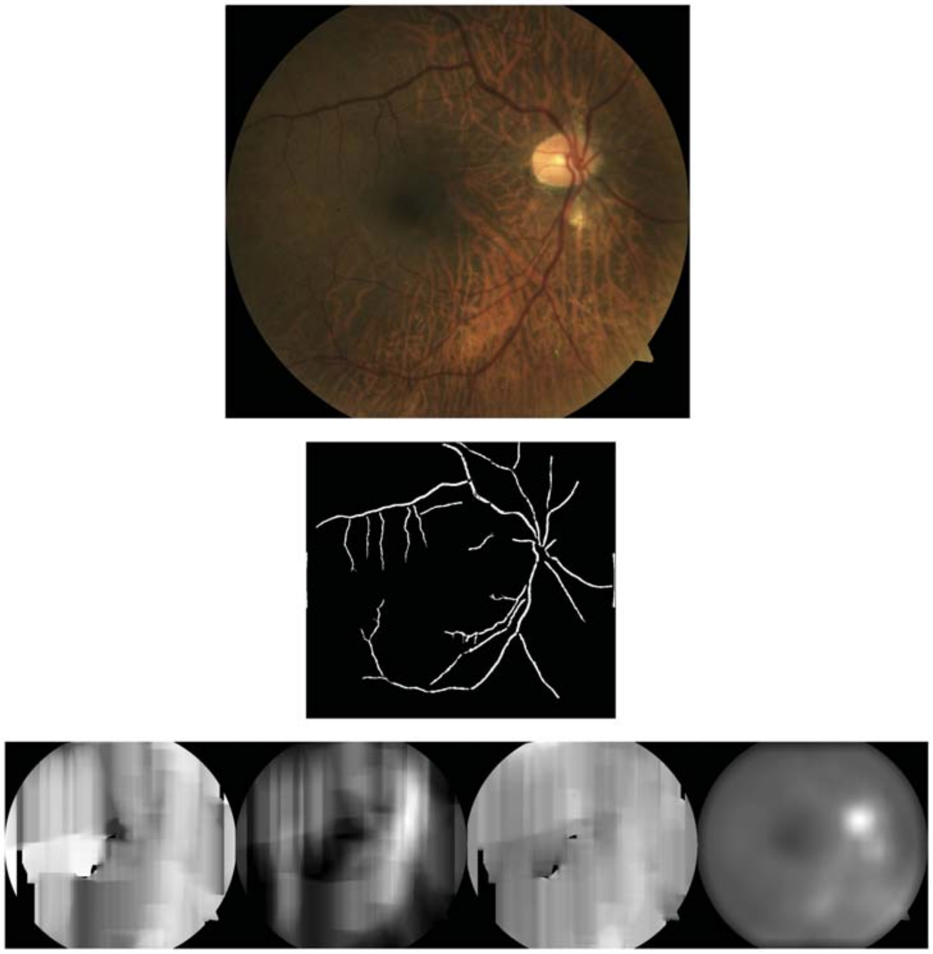


Figure 1.2: Example of hand-engineered features to solve a problem, in this case the location of the optic disk.

the image) and continuously construct a belief state that attempts to characterize the sequences of patterns viewed. The second layer, and all those above it, receive as input the belief states of nodes at their corresponding lower layers, and attempt to construct belief states that capture regularities in their inputs. DeSTIN offers two key attributes that render it attractive. First, the belief space that is formed across the layers of the architecture inherently captures both spatial and temporal regularities in the data. Given that many applications require that temporal information be discovered for robust inference, this is a key advantage over existing schemes. Second, each node is identical suggesting ease of mapping the design to massively parallel platform, such as graphics processing units or other such platforms.

In summary, we may envision DeSTIN as an unsupervised feature extractor. As opposed to a human, hand-engineered system of computer algorithms which seek to highlight and detect features that can be used to solve a specific pattern recognition task, DeSTIN learns in an unsupervised manner, with no training labels, from data examples and generates features that can be used for supervised learning.

1.4 Contributions

We have developed a unique, biologically-inspired, deep architecture for unsupervised feature extraction suitable for a variety of high-dimensional signal analysis problems, including imaging. The research contributions are:

Inherent temporal nature: The DeSTIN belief formulation is inherently temporal in nature, as successive observations are used with the static and dynamic learning to formulate a multiple-observer predictive belief which is suited as an unsupervised feature for later use with a pattern classifier. Each node observes and learns to represent a temporal sequence of patterns, with the lowest layer of the hierarchy accessing temporally changing input data, over time continuously constructing a belief state that attempts to characterize the sequences of patterns viewed. Subsequent layers process the belief states of child nodes, and construct belief states that capture regularities in their inputs. At each node, outputs of hierarchical belief states across all layers capture both spatial and temporal regularities in the data, which is a novel, key advantage over existing deep learning schemes.

Simplified architecture: The second major contribution is the use of a single simplified architecture for the feature extraction process. Deep machine learning has often required a variety of "tunings" which are specific to the problem domain and are often achieved experimentally. Instead DeSTIN has been implemented in this work as a single, specific topology which would change only based on the size of the input field. These principles and guiding philosophy lends DeSTIN to a more natural mapping to simple hardware systems.

Simultaneous layer learning: The third research contribution is simultaneous layer learning, where all layers of the hierarchy learn continuously and simultaneously. While lower layers naturally form their beliefs sooner, the upper layers adapt and change continuously to the response of the lower layers until those layers stabilize, which is both a better emulation of biological learning and more efficient as the convergence of each layer does not depend on complete convergence of previous layers. The pipeline nature of DeSTIN does require some between-layer dependencies, causing the higher layers of the system to learn more slowly than lower layers (with fewer dependencies).

Online learning: The fourth major contribution is the use of online learning throughout the feature extraction process. While many deep machine learning architectures also use online learning, many do not, and emphasize batch processing, largely due to the difficulty of the problem of training multiple layers of processors effectively. Instead DeSTIN has been implemented as an online learning problem, with online clustering and probability distribution function estimation used to generalize static pattern learning, and tabular or function approximation methods used to learn system temporal dynamics. Again, this lends DeSTIN to a more natural mapping to simple hardware systems.

Layered feedback: The final research contribution is the use of layered feedback to focus local attention on data variation. Maintaining good generalization, while capturing sufficient variation to accurately learn to discriminate data categories, is an important goal of artificial general intelligence. In DeSTIN, this goal is accomplished through the feedback used by the cortical algorithm common to each node. This allows the nodes of each layer of the hierarchy a sensitivity to local variations in the data, which can be obscured by the limited capacity or excess generalization of higher nodes, by placing its learning in the context of the "higher picture" provided by its parent nodes. Feedback has been used in deep machine learning in the past, but in DeSTIN it is an inherent part of the formulation and is used in concert with all node beliefs (save perhaps the top node).

1.5 Dissertation Outline

In the next chapters, we present a background section including literature review of work relevant to our general topic. In particular, we formally introduce deep machine learning and its applications. Some discussion of cortical-inspired computing architectures is included for completeness. This chapter presents sufficient background to place DeSTIN in a context with other deep machine learning methods as well as presents the state-of-the-art in deep learning methods. The next chapter contains a detailed description of the components of DeSTIN and proposed methods for analysis and implementation. We then focus on the two important elements of DeSTIN, static learning (Chapter 4), and dynamic learning (Chapter 5), and introduce mechanisms for this work using probabilistic methods and function approximation. In the sixth chapter we discuss results using a variety of different data sets: handwritten digits, face detection, and optic nerve detection from retina images. In the seventh chapter we discuss additional implementation details, including the computational load and noise response of DeSTIN. We conclude in the eighth chapter, where we also discuss future work in this field and with DeSTIN in particular.

Chapter 2

Background and Literature Review

2.1 Overview of Machine Learning

The field of machine learning (ML) is broadly concerned with algorithms that effectively "make computers learn". The concept of computers learning automatically through experience (acquisition of more and more data through time) is a very appealing concept and is the subject of considerable research and development energy over the past 50 years or more. Much of this appeal is based on the features of computing that far exceed human capabilities: memory capacity, computational speed at tasks such as numerical analysis, and the potential for continuous operation. While it is safe to say at this juncture that computers cannot achieve true human levels of general learning and intelligence, there have been many great advances in specialized fields where computers have exceeded human capabilities, including some gaming environments and recognition tasks under controlled environments. These are generally well-defined problems with clear rules, but their accomplishments are truly impressive Mitchell (1997). In this section we will review prominent areas of machine learning which play a relevant part in this dissertation, specifically supervised learning and unsupervised learning. Other areas of machine learning, including semi-supervised learning Zhu (2006), which attempts to solve problems where a large amount of data is available with an incomplete quantity of classified or labeled data, are certainly important but are less relevant to this work.

2.1.1 Supervised Learning Methods

In supervised learning, a training set of labeled data is available Duda et al. (2001) where labels consist of classifications or designations which the ML algorithm should reproduce. The inclusion of the training set allows the pattern recognition algorithm to solve the problem at hand through an optimization process where the algorithm learns how to discriminate between the different categories of interest. Broadly, supervised learning techniques can be divided into two fields: parametric techniques and non-parametric techniques. In parametric techniques, such as Bayesian decision theory Mitchell (1997), a classification engine is created using discriminate functions which are based on some assumed characteristic of the data. A popular assumption is that the data obeys Gaussian probability distributions. Non-parametric techniques, on the other hand, make no assumption about the functional form of the data and instead estimate the probability density function (PDF), or even the a posteriori probability functions, directly from the data rather than estimating the parameters of the PDF from the data. Non-parametric techniques can be very powerful and flexible since they do not require knowledge of the structure of the data. We cannot necessarily conclude that they are "better" than parametric techniques, however, because such a blanket statement does not consider details such as implementation complexity, domain knowledge, and the amount of training data available. Some well-studied and popular nonparametric methods include the k-Nearest Neighbor algorithm and artificial neural networks.

Bayes Decision Theory

A parametric method such as Bayes decision theory uses the laws of probability to determine the most likely category for an unknown vector. The PDFs needed to implement such an approach must be known (or assumed) and their parameters must be determined in order to make effective use of this approach. Bayes' theorem states that

$$p(\omega_i|o) = \frac{p(o|\omega_i)p(w_i)}{p(o)}, \tag{2.1}$$

where ω_i is the 'state of nature', or the actual classification of the observation vector o; $p(w_i)$is the probability that the 'state of nature' ω_i will actually occur (note that ω_i is a discrete value); $p(\omega_i|o)$ is the probability that the 'state of nature' was ω_i given an observed vector o; $p(o|\omega_i)$ is the probability of observing vector o given that the 'state of nature' is ω_i; this is generally a PDF, specifically a conditional probability density function; $p(o)$ is the prior on o. Generally speaking, this latter PDF is not an issue for Bayes' decision theory because it is not a function of the 'state of nature' and is therefore identical for all classes.

In Bayes' decision theory we seek to find the 'state of nature' ω_i that maximizes the probability of producing the observed vector o. There are several ways to compute this, but essentially they can be summarized as choosing the class ω_i that yields the largest value for the observed vector o. As such, the decision rules allow a formal, rigorous methodology for what many of us simply see as common sense: for example, if an unknown vector o "looks more like" a group of vectors labeled ω_i than another group ω_j, that unknown vector most likely belongs to ω_i . From a practical standpoint, the decision rules we derive allow us to build classifiers which can distinguish between vectors from each class and allow us to create trade-offs regarding the risk of miss classifications.

In many cases a reasonable and tractable assumption is a Gaussian probability density given as

$$p(o|\omega_i) = \frac{1}{(2\pi)^{\frac{d}{2}}|\Sigma_i|^{\frac{1}{2}}}\exp(-\frac{1}{2}(o-\mu_i)^T\Sigma_i^{-1}(o-\mu_i)), \tag{2.2}$$

where ω_i is the 'state of nature', or the actual classification / label of vector o, μ_i is the mean of class i, and Σ_i is the covariance matrix, with element σ_{ij} computed as

$$\sigma_{ij} = E\{(o-\mu_i)^T(o-\mu_j)\} \tag{2.3}$$

where $E\{\}$ denotes the expectation. Note that $|\Sigma|$ is the determinant of the covariance matrix. The choice of the Gaussian PDF is motivated by several factors Cooper and McGillem (1999). First, the Gaussian PDF is a good mathematical model for many natural events and processes. A related factor is the results of the Central Limit Theorem, which tells us that the PDF of a sum of many independent random variables will become Gaussian in its form, regardless of the PDF of the original random variables. Another important factor is the simplicity of the Gaussian PDF, which can be completely specified with only its mean and covariance (first and second order statistics); and as we shall see, these parameters are easy to estimated with maximum likelihood estimation techniques. Finally, the Gaussian PDF is mathematically tractable for many problems, which is a powerful motivation for its use as well because of the insight it can offer into different physical situations.

Using the Gaussian models is a powerful tool, but the parameters of the models must be known or computed to effectively tap into their power. Parameter estimation of random processes is a very important area of study in itself, but a powerful simple method, maximum likelihood (ML) estimation, is based on the following concept. Suppose we have a set D of training samples o_1,o_2,o_3, ... o_n which are chosen independently. The probability of picking any one sample o_k is given by $p(o_k|\theta)$ where θ is a vector containing the parameters of the random process. We want to choose the parameters θ that maximizes this probability. When the PDF has a Gaussian form, these parameters (the mean and covariance) are very easy to estimate. Of the training set D we assume there are n_1 elements that "belong" to class ω_1 and n_2 elements that belong to class ω_2. Consider choosing just one vector o_1 the probability of choosing it is $p(o_1|\theta)$. Since the choice of the vectors are assumed to be independent the probability of choosing the first one and the second one together is simply the product $p(o_1|\theta)p(o_2|\theta)$, and extending this to n vectors yields

$$p(D|\theta) = \prod_{k=1}^{n} p(o_k|\theta). \tag{2.4}$$

With maximum likelihood estimation, we want to determine the value of θ that gives the maximum value for these equations. This is a very simple concept, but in practice it can be difficult to determine analytically owing to the complexity of the

PDF model. A common method used in problems of this sort is to take the logarithm of these functions. This reduces the fairly complex product operation into a summing operation which could be simpler to manipulate, and is a valid operation because the logarithm function is monotonically increasing, the x that maximizes $f(x)$ will also maximize $ln(f(x))$. Thus taking the logarithm yields

$$\log(p(D|\theta)) = \log(\prod_{k=1}^{n} p(o_k|\theta)) = \sum_{k=1}^{n} \log p(o_k|\theta) \tag{2.5}$$

For the Gaussian PDF the parameters are the mean μ and the covariance matrix Σ, so substituting the Gaussian form in yields

$$\sum_{k=1}^{n} \log p(o_k|\theta) = -\frac{1}{2}\sum_{k=1}^{n} \left\{\log(2\pi)^d + \log|\Sigma| + (o-\mu)^T\Sigma^{-1}(o-\mu)\right\}. \tag{2.6}$$

To find the value of θ that maximizes this equation, we can take the gradient, which is a set of partial derivatives with respect to the parameters we want to maximize. We then set the gradient to zero and find the θ that solves the equation. In the case of the Gaussian PDF, we are guaranteed that this is indeed the maximum because of the single-modal nature (the function increases, reaches a peak, then declines). The solution for a general, multivariate case is Duda et al. (2001)

$$\mu = \frac{1}{n}\sum_{k=1}^{n} o_k, \tag{2.7}$$

$$\Sigma = \frac{1}{n}\sum_{k=1}^{n}(o_k-\mu)(o_k-\mu)^T. \tag{2.8}$$

This is the "common sense" solution as well; we average the vectors of each class to compute the class mean and the covariance is the average of the covariance computed by each vector individually.

The KNN Algorithm

Most of the work in designing parametric classifiers comes in finding the parameters of $p(o|\omega_i)$. A simple nonparametric method, the K-nearest neighbor (KNN) classifier, cleverly skips much of this work by performing the following approximation. Suppose there are K training vectors in the vicinity of the unknown o, so that there are no other members of the training set that are closer. We can approximate the probability $p(o)$ as

$$p(o) = \frac{K}{nV}, \tag{2.9}$$

where n is the number of vectors in the training set and V is the volume of the region that just envelops all K vectors. This is the fraction of all the vectors that are in our vicinity. In the entire data set, suppose there are n_i vectors that belong to class ϖ_i. We can therefore approximate $p(\omega_i)$ as

$$p(\omega_i) = \frac{n_i}{n} \tag{2.10}$$

Finally, suppose out of the K vectors in our immediate vicinity, K_i of them are labeled ω_i. In that case, we can approximate $p(o|\omega_i)$ as

$$p(o|\omega_i) = \frac{K_i}{n_i V} \tag{2.11}$$

Again, this is the fraction of vectors from class ω_i that are "nearby". We are ignoring all the other classes, and in fact that is what we do when we condition the probability with the “given ω_i" term. If we substitute these approximations into Bayes’ theorem, we find

$$p(\omega_i|o) = \frac{p(o|\omega_i)p(w_i)}{p(o)} = \frac{\frac{K_i}{n_i V}\frac{n_i}{n}}{\frac{K}{nV}} = \frac{K_i}{K}, \tag{2.12}$$

which is merely the fraction of vectors belonging to class ω_i out of the K closest ones. We assign the classification based on which of these probabilities is highest – and since K is constant, basically we just choose the class with the most representatives in our vicinity. The appeal to intuition again makes sense in this case: if most of the vectors that are close to our sample belongs to class ω_i, then our sample probably belongs to class ω_i as well.

The KNN algorithm is very simple to implement. An unknown vector is compared with every member of the training set. The distance is computed between the unknown vector and each member in the training set according to some metric, such as Euclidean distance. Then the distances are sorted. We take the K training members that are the closest to the unknown vector and find the classification that is the most represented. We then assign this class to the unknown vector. The method is straightforward and simple to implement, although it can have some large computational cost since we must compute the distance between our unknown sample and every vector in the training set.

Artificial Neural Networks

Neural networks attempt to model a biological process, the neuron, and deploy it as a computational element. In a neural network a series of input elements or nodes are created which are connected to multiple "hidden" nodes through a weighting matrix. These hidden nodes comprise a "hidden layer" which is may be connected to one or more additional "hidden layers" with large fan-out (connection of nodes in a layer to many nodes in the next layer). At each element the input values are summed and passed through an activation function. A simple neural network is depicted in Figure 2.1. This network has two input nodes, one hidden layer with two nodes, and an output node. Each node has an activation function which operates on the sum of the inputs to the node. Here we depict the input nodes with a linear function (meaning they simply pass their input values with no modification). The hidden and output nodes, however, have a nonlinear activation function which allows for more complex behavior. Each of the nodes may also have a bias added to the values on its input as well which is not depicted in the figure.

A sort of predecessor to neural networks, the Perceptron, originated in the 1950s and 1960s with the work of Rosenblatt Rosenblatt (1958) and MinskyMinsky and Papert (1969). Perceptrons use a binary threshold as its activation function which limited the functional capabilities of the architecture. Indeed, Minksy's analysis has been credited with making neural network research difficult for many years, as the work was perceived as emphasizing the limitations of the approach rather than its strengths Dreyfus and Dreyfus (1988). Neural networks became more popular and influential as more researchers discovered the backpropagation algorithm or variants of it (see Duda et al. (2001) Chapter 6 for description of these contributors), but the paper by Rumelhart, Hinton and Williams Rumelhart et al. (1986) reached a wide audience and led to a resurgence in neural network research and development.

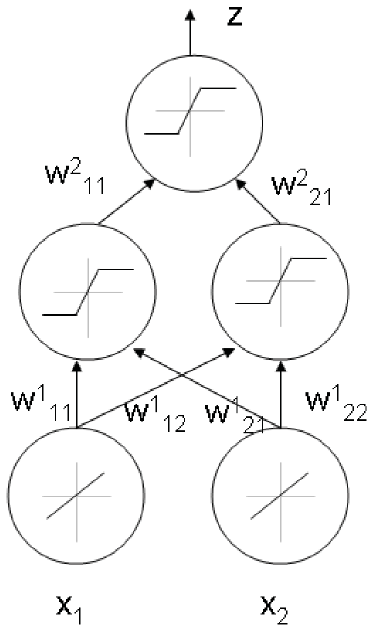


Figure 2.1: Simple neural network for illustration, with a two dimensional input layer, two hidden nodes in one layer, and one output layer.

Gradient descent is at the core of the backpropagation algorithm as it allows us to find the optimal weights. Basic gradient descent consists of choosing an initial value for a weight vector $w^{t=0}$, where t is the iteration through the algorithm, then computing a new value by first finding the value of the gradient of a cost function $J(w^{t=0})$ and adjusting the weight by a small amount in the direction of the gradient. If the gradient is given by ΔJ then the update is simply

$$w^{t+1} = w^t - \eta \Delta J(w^t) \tag{2.13}$$

where η is the learning rate, a scalar value that dictates the degree to which the weight is adjusted. The learning rate can be set by using the Hessian of the cost function $\frac{\delta^2}{\delta w_i \delta w_j} J(w^t)$, but this is not always helpful. Newton descent is also an alternative but it requires an inversion of the Hessian and does not work for nonquadratic error functions. In practice the learning rate is often simply set to small values and decays over time.

The neural network is trained by randomly initializing its weights, often to random values, then repeatedly presenting the network with the inputs, finding the output values, and using the resultant error to train the network. The "credit assignment" problem refers to the problem of properly determining what the output of a hidden node should be to achieve a desired overall target output Duda et al. (2001). Backpropagation defeats the credit assignment problem and allows the neural net training a way to compute the error at each hidden node so that the proper weights from input to hidden nodes can be designed. The back-propagation algorithm works by using gradient descent to adjust the weights after backing the output through each node, requiring a differentiable activation function. From Duda et al. (2001) we see a derivation of the back-propagation algorithm; we simply repeat its main points here. The goal is to minimize the error between the network outputs, z, and the desired target outputs, τ, expressed as

$$J(w) = \frac{1}{2}\sum_{k=1}^{c}(\tau_k - z_k)^2 \tag{2.14}$$

We identify the weighted-and-summed inputs to node k as net_k. Then given an output z_k and a target value τ_k at the k_{th} output, we compute the error as $(z_k - \tau_k)$. We can then adjust each weight between output node k and hidden node j as

$$\Delta w_{kj} = \eta \delta_k y_j = \eta(\tau_k - z_k) f'(net_k) y_j \tag{2.15}$$

where $f\prime(x)$ is the derivative of the activation function $f(x)$ and y_j is the output of the hidden node j. As mentioned, the learning rate η is some heuristic value that scales how we adjust the weights each cycle; if it is too low the network training takes too long, but if it is too high we get the familiar “skip-around” effect of gradient descent where a local minimum is not found. Adjusting the weights between the inputs and the hidden nodes follows a similar form with more complex details:

$$\Delta w_{ji} = \eta \delta_j o_i = \eta(\sum_{k=1}^{c} w_{kj}\delta_k) f'(net_j) x_i \tag{2.16}$$

The neural network training proceeds until a minimum error rate has been achieved or some other stopping criteria (such as performance on a validation set has reached a desired target or a maximum number of training cycles has been attained).

2.1.2 Unsupervised Learning

Another broad area of research in ML is unsupervised learning (UL), where algorithms learn how data of interest is organized. Unsupervised learning also seeks functions that can label input vectors but they begin with no prior knowledge of the desired output label. This kind of problem may seem to be highly impractical but in Duda et al. (2001) five reasons for interest in UL are given

1. Collecting and labeling a large set of sample patterns can be costly.
2. UL can help produce training data for later supervised learning applications.
3. For some applications, the characteristics of the samples can change. UL can actually track these changes due to its greater flexibility.
4. UL can help determine the features of a data set that are most useful for later supervised learning applications. In this case UL acts as a pre-processor.
5. UL can help researchers gain insight into the nature of their data.

In fact, many applications have several of these traits in common. In data mining, databases are analyzed using a variety of techniques including unsupervised learning. These algorithms can identify patterns and connections between data that were previously unknown (5 above) and may continuously refine these connections as the data changes (case 3 above.) A well-studied method of unsupervised learning is clustering Xu and Wunsch (2005), where algorithms attempt to assign labels or classes to data points based on their similarity to other points in the data set. Three well-known clustering algorithms include the K-Means, Winner-take-all (WTA), and Kohonen algorithms.

The algorithm called "K-Means clustering" attempts to estimate the mean value of a given number of clusters K. To describe this algorithm, suppose we know the cluster means. We perform a minimum distance classification of a vector x by simply finding the distance or a similarity between o and every cluster mean. We then classify o as class ω_j where $d(o, \omega_j) \leq d(o, \omega_i)$ for all $i \neq j$, where $d(x, y)$ is a distance measure indicating how dissimilar two vectors x and y are from one another. In essence this is simply approximating the probability $p(\omega_j|o, \theta_j)$ as $p(\omega_j|o, \theta_j) = 1$ for $i = j$ and $p(\omega_j|o, \theta_j) = 0$ for $i = j$. Since the cluster means are not known, typically we can randomly set their initial values and iteratively modify them by assigning each vector to a cluster, then recomputing the cluster means by using the maximum likelihood estimate of the class mean,

$$\hat{\mu} = \frac{1}{n_i}\sum_{k=1}^{n_i} o_k \tag{2.17}$$

where n_i is the number of vectors that were assigned to class i. The vector assignment is performed by computing the distance from the cluster centers and the vector, and assigning the cluster number to the smallest distance. The algorithm stops when the vector assignments to clusters do not change. The K-Means algorithm is simple to implement and works well in practice. As a result, it is commonly applied to a variety of problems and is a very good benchmark for unsupervised learning algorithms Duda et al. (2001).

In a similar method, the Winner-Take-All clustering algorithm Kaski and Kohonen (1994), the cluster centers are iteratively according to the rule

$$\mu_i^{k+1} = \mu_i^k + \eta(o - \mu_i^k) \tag{2.18}$$

This rule applies only to the "winner", the closest cluster center from vector o. The cluster centers are assigned arbitrarily initially, then the vector data is entered and the distance from each cluster center to the vector is computed. The minimum distance center is deemed the 'winner' and its center is updated as shown. After the cluster center is updated, it will be closer to the vector and thus the vector's "cluster number" will not change as a result of the update. The winner-take-all or WTA algorithm has one parameter, the learning rate η. This value can be changed as a function of epoch or iteration number, or it can remain constant. In any event, after several iterations the cluster assignments will not change and the algorithm has converged to a solution.

A final method of interest is the Kohonen self-organizing map or SOM, which is a type of neural network that can perform unsupervised learning Kohonen (1990). The neural network is depicted on the left side of Figure 2.2. As in all neural networks, each input is connected to each node with a system of weights. The weights are used in a slightly different manner than a backpropagation neural network, however. Also, the training of a Kohonen map is a bit simpler than a back-propagation network. The training and operation proceed in the following manner. First, the node weights are initialized through some method. When a new training vector (unlabeled) is presented, each node receives an input computed as

$$y = \sum (o_i - w_{ni})^2, \tag{2.19}$$

which is simply the Euclidean distance between the input and the weights of node n. The node b with the smallest distance y is deemed the "winner". At this point the weights are updated using the following algorithm:

$$w_m^{k+1} = w_m^k + \eta(k)N(k, g_{mb})(o - w_m^k), \tag{2.20}$$

where k is the iteration number, $\eta(k)$ is the learning rate (written here as a function of the iteration number), $N(k,g)$is a neighborhood function (also a function of the iteration number), g_{mb} is the grid distance between the winner node b and the node we are updating m.

An iteration number can be interpreted as either a single training vector submission or a complete epoch through the training set. The learning rate $\eta(k)$ is a function that monotonically decreases as k increases. Typically an exponential or linear decay is chosen, but the function should asymptotically (or absolutely) decay toward zero. The neighborhood function has a maximum when g is 0, and grows smaller as g increases. Furthermore, as k increases $N(k, g_{mb})$ should decrease as well, where g_{mb} is not 0.

The grid distance bears further explanation and illustration (right side of Figure 2.2). The node shown in yellow (node 2,1) has been chosen as the winner. The grid distance to the node (1,1) just to the left of the winner node is 1. The grid distance to the node (1,3) just to the left of the winner node and at the bottom of the grid is 2.23.

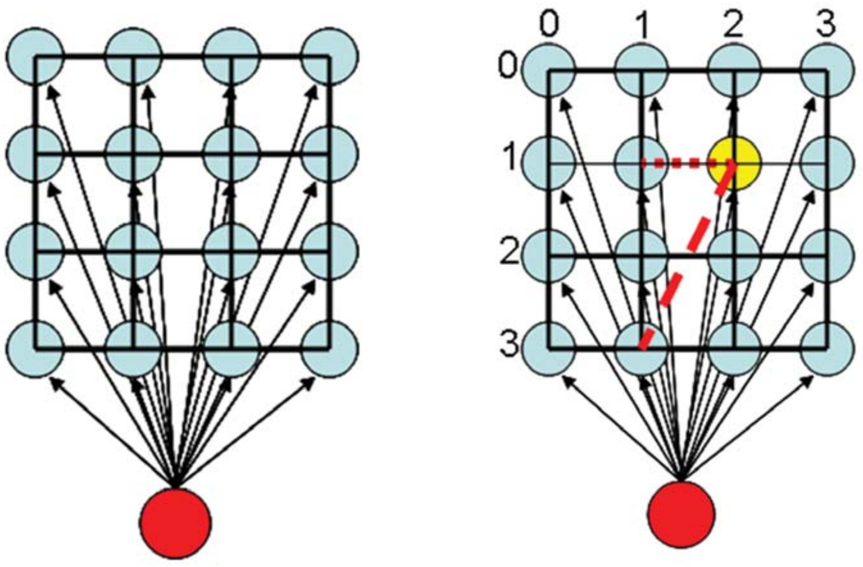


Figure 2.2: Left: Kohonen network. Right: Grid distances for computing the neighborhood training values.

This is a bit counter-intuitive, because the neighborhood function has nothing to do with the weights associated with the node. However, that is the entire point of the grid. Essentially the map "self-organizes" as training data is shown and the weights are trained. The "self-organizing" refers to the algorithms' ability to change the node weights so that adjacent weights on the grid become more like one another. In clustering applications, the weights associated with each node are actually cluster centers. We see that the training process then means we find the most similar cluster to an input vector, then we pull the best cluster's weights toward the input data and modify its neighbors in the same direction (but to a lesser degree). There are K nodes for a target of K clusters. The network performs classification by simply finding the winner node for an input vector and identifying the input as belonging to that cluster. As in the other algorithms, convergence is reached when the data inputs do not change their cluster centers.

2.2 Feature Extraction

In pattern recognition, especially as associated with image processing, the concept of "feature extraction" is a core component for success. A "feature" is essentially a numerical measurement of some characteristics of the image, or regions of an image. Early development of pattern recognition and image processing or machine vision systems usually consisted of a processing phase where, after images were obtained and properly filtered or otherwise pre-processed to isolate desired regions of interest, key elements of the image were measured in an attempt to characterize the image for successful pattern classification Duda et al. (2001), Gonzalez and Richard (2002). For example, in manufacturing the detection of defects in the manufactured material was followed by efforts to determine the type of defect, as this could deliver additional insight into the manufacturing process and flaws that may exist Kumar (2008). In many cases these features consisted of domain-specific values that were easily measured but also indicative of the type of defect, such as the elongation of a region (to identify scratches). In other applications, as we mentioned earlier for the location of the optic nerve Tobin et al. (2007), the features measured were even more specific and depended greatly on the success of earlier image processing steps taken. Efforts to develop more general features, such as invariant moments Wong and Hall (1978), Li (1992) or other such measurements have created a vast body of literature, with much of it very application specific (see for example Hong et al. (1998), Chou et al. (1997), Giancardo et al. (2011)). In Jiang (2009), the author denotes four main classes of features: human-engineered from domain knowledge, image local structure based, image global structure based, and machine learning based "statistical" approaches. The feature extraction method of DeSTIN and CNNs fall into the final category. Techniques such as Fourier analysis or the aforementioned invariant moments yield global structure based features.

In recent years there has been more focus on general feature extraction, especially associated with salient regions or "keypoints" in images. These are examples of the image local structure based approach. This was initially inspired by applications for object tracking and stereo matching, which was initially accomplished by numerically simple, but computationally intensive operations such as normalized correlation Dickey and Romero (1991) and matched filtering Horner and Gianino (1984) where a template was used for a sort of brute-force search using shear matching pixel values. Later work Harris and Stephens (1988), Shi and Tomasi (1994) used image patches and performed computationally efficient searches for patches with two strong spatial eigenvalues, which are indicative of corners. This work still has many important applications and open-source implementations exist Bradski and Kaehler (2008).

A more robust method of great influence is the Scale-Invariant Feature Transform or SIFT Lowe (2004). In essence, there are actually two main components to SIFT. The first is the extraction of keypoints, or regions where the image has a highly identifiable point. These are computed by filtering the image of interest with a variety of scales of difference of Gaussian operations. A keypoint candidate occurs when it is greater than, or less than, its neighbors within a scale and within adjacent scales. A 3-D quadratic model is fit to the point and several additional filtering criteria are applied, to reject points with poor contrast or poor spatial characteristics, then an orientation is assigned to the point using a binning or histogram approach based on the region.

The second main contribution of SIFT is the generation of a high-dimension feature vector that describes each point. A windowed region around the keypoint is used with image gradients and the associated orientation and scale are used to compute a sort of hierarchical descriptor. The method has been empirically proven to be quite robust in the face of some rotation and scale changes, and was the best method in a performance evaluation Mikolajczyk and Schmid (2005). There have been other, related methods Bay et al. (2008) which attempt to make simplifications to save computational cost, and there is an active amount of research on this topic.

Several different general object recognition methods use SIFT as an integral part of their processing, as do "bag of words" type approaches Serre et al. (2005), Ranzato et al. (2007a), Mutch and Lowe (2006). For the purposes of our work, we view the use of SIFT as a complimentary, and competing, method to our work. The method compliments in the fact that the feature values can be used as a proxy for the image content itself, although exploration of this concept has been left for future research. The method competes in that it does not actually learn how to represent image elements, but instead relies on a semi-heuristic, but very general, method which has been empirically proven to "work" in a variety of practical applications.

2.3 Overview of Deep Machine Learning

A comprehensive overview of deep machine learning is presented in Bengio (2009), where it is cast in the light of the goals of general Artificial Intelligence (AI). In this section we will discuss the major findings from that work as they form a basis for deep learning in general. Overall, the goal of AI is to learn complex functions that have much variation, in fact, more variation than training examples. True AI systems must also learn with little human input and should have a computationally manageable scaling with the number of inputs. They should be able to learn in an unsupervised manner, as unsupervised learning is critical for dealing with unknown future tasks or ill-defined tasks; once unsupervised learning is complete (and well-done) it can serve as an input to supervised learning and simplify the tasks. Shallow architectures are limited in their applicability to AI in general because their computational complexity becomes too large for a given problem. An example, which Bengio (2009) references from Hastad (1987), is a parity function which adds d bits together and if the sum is even the output is 1 and if it is odd the sum is -1. This is exponential as a sum-of-products (its on the order of 2^d) but a "deep" tree structure can implement the function with only $d-1$ summing or comparator functions. This anecdotal example is not a formal proof that machine learning problems have the same requirements, but given the difficulty in general AI over the years, clearly there is a good case that complicated functions cannot be adequately represented with 1 or 2-depth shallow architectures. Furthermore, while a given function could be implemented in two ways, one shallow and one deep, the latter should be more efficient with less elements.

Based on current understanding of the hierarchical nature of the neocortex, the many layers of the visual system comprise a deep architecture. Thus it is easy to make an argument that truly bio-inspired or biomemetic approaches should be deep architectures. Of course, examples of shallow architectures abound in machine learning and the literature and can solve many problems. We discussed neural networks earlier; a neural networks with an input layer and a hidden layer is an example of a two layer algorithm. Networks with one or two hidden layers constitute shallow networks and are the most practical implementation of these kinds of algorithms. As another example, a kernel function can be expressed as Scholkopf and Smola (2002)

$$f(o) = \sum_i \alpha_i K(o, o_i) \tag{2.21}$$

where $K(o, o_i)$ is the kernel computation. This function too has two layers; the first being the calculation between the input o and the template vectors o_i, and the second being the linear combination of those initial results. The KNN algorithm is similar; the first layer is the calculation of the distance or similarity between input o and the database vectors, and the second is a counting or voting operation.

The most influential deep architectures in the literature are convolutional neural networks and deep belief networks. There have also been several architectures presented in the literature which we dub "cortical inspired" which are noteworthy as well. In the following sections we review these architectures and describe their operation in some detail.

2.3.1 Convolutional Neural Networks

Convolutional Neural Networks or CNNs LeCun et al. (1998), Huang and LeCun (2006) are a family of multi-layer neural networks particularly designed for use on two-dimensional data, such as images. CNNs are influenced by earlier work in time-delay neural networks (TDNN) which reduce learning computation requirements by sharing weights in a temporal dimension and are intended for speech and time-series processing Waibel et al. (1989), Lang et al. (1990). CNNs are the first truly successful deep learning approach where many layers of a hierarchy were trained in a robust manner. A CNN leverages spatial relationships to reduce the number of parameters which must be learned and thus improves upon general feed-forward back propagation training for image applications. In CNNs, small portions of the image (dubbed a local receptive field) are treated as inputs to the lowest layer of the hierarchical structure. Information propagates through the different layers of the network where at each layer digital filtering is applied in order to obtain salient features of the data observed. The method provides a level of invariance to shift, scale and rotation as the local receptive field allows the neuron or processing unit access to elementary features such as oriented edges or corners.

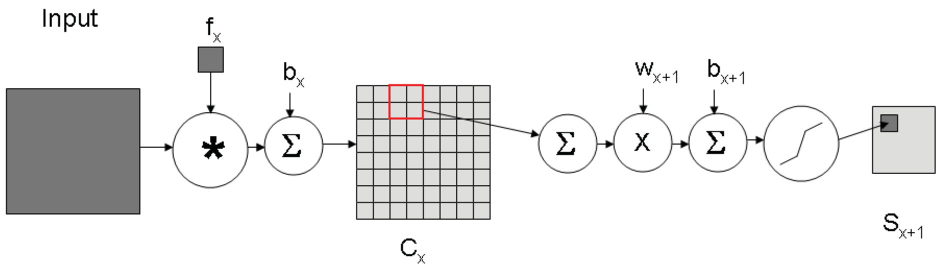


Figure 2.3: The convolution and subsampling process in a CNN. An input is convolved with a trainable filter and biased to produce the convolution layer C_x. The subsampling consists of summing a neighborhood (four pixels) followed by scaling, biasing, and "squashing" to produce a smaller feature map S_{x+1}.

A depiction of a CNN is found in Figure 2.3. Essentially, the input image is convolved with a set of N small filters whose coefficients are either trained or pre-determined using some criteria. Thus, the first (or lowest) layer of the network consists of “feature maps” which are the result of the convolution processes, with an additive bias and possibly a compression or normalization of the features. This is followed by a subsampling (typically a averaging operation on a 2×2 region of pixels) which further reduces the dimensionality and offers some robustness to spatial shifts. The subsampled feature map then receives a weighting and trainable bias and finally propagates through an activation function. When the weighting is small, the activation function is nearly linear and the result is a blurring of the image; other weightings can cause the activation output to resemble an AND or OR function. These outputs form a new feature map that is then passed through another sequence of convolution, sub-sampling and activation function flow. This process can be repeated an arbitrary number of times. It should be noted that subsequent layers can combine one or more of the previous layers; for example, in LeCun et al. (1998) the initial six feature maps are combined to form 16 feature maps in the subsequent layer. Some variants of this exist with as few as one map per layer Chen et al. (2006).

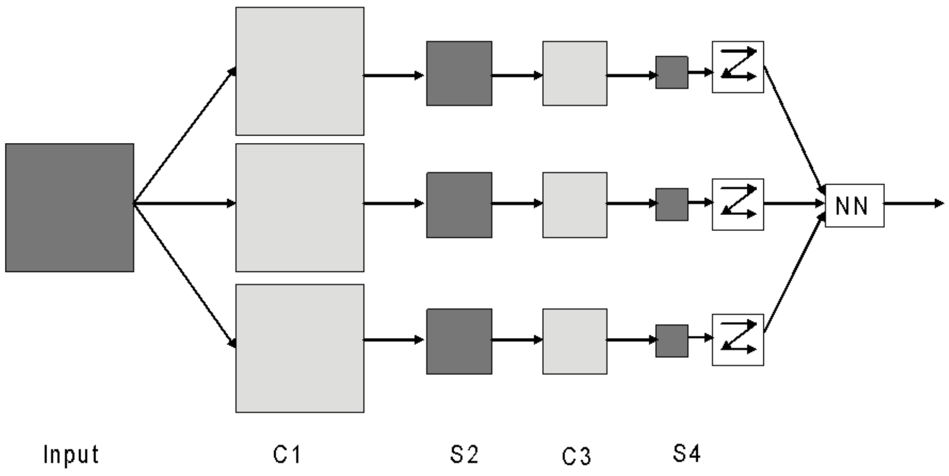


Figure 2.4: Conceptual example of convolutional neural network. The input image is convolved with three trainable filters and biased to produce three feature maps at the C1 level. Each group of four pixels in the feature maps are added, weighted, biased and squashed to produce the feature maps at S2. After repeating the process the outputs are presented as a single vector input to the "conventional" supervised learning system, such as a neural network.

Finally, at the final stage of the process, the activation outputs are forwarded to some supervised learning system, such as a conventional feedforward neural network that produces the final output of the system. This final stage should be capable of generating an error signal during training which can be used to set the weights and biases of the previous layers through backpropagation.

We show the convolutional aspect of the CNN in more detail in Figure 2.5. In this depiction, a single input field (the larger, 10×10 pixel region) is multiplied by a set of weights W_0 and summed, along with a bias value b. These then pass through a nonlinearity element, and a value in the 6×6 output region is produced. The second figure shows a slight shift of a single pixel and the process is repeated; this is the convolutional effect, and the entire image is processed in this way with the same weight and bias used. Multiple input fields can be used, as shown in Figure 2.6. Furthermore multiple output fields can be generated as well, each with their own trainable bias and weighting. From the literature, it appears that the choice of the number of output fields and the number of units that are combined to produce a given field is somewhat tuned by the designer, but the key learning benefit of CNNs - learning the feature extractor for the problem at hand - is completely autonomous. The sampling phase is shown in Figure 2.7. Here we see that a 2×2 windowed region is scaled by a single value, with additive bias and nonlinearity, to produce an output field that is $\frac{1}{2}$ the size of the input. Thus this kind of layer has far fewer training

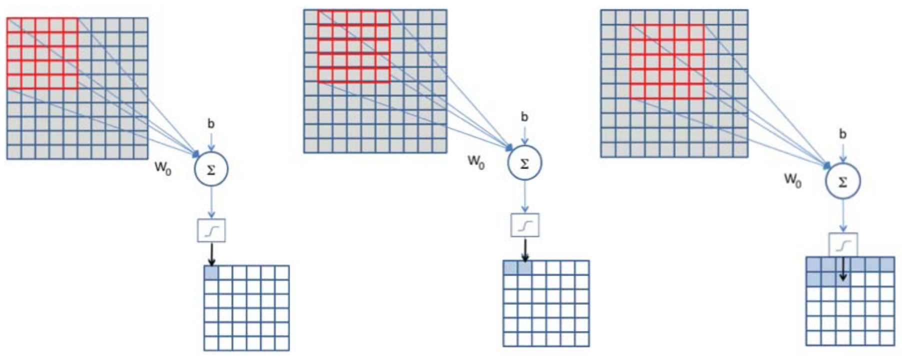


Figure 2.5: The convolution process for a CNN is shown here. At left, the input image or feature map is multiplied in a 5x5 window by a weighting factor W_0. This is summed together and added to a bias, which is then fed to a nonlinearity. In the middle, we see the same process except on a different input field. Finally on the right we have continued with yet another windowed region. This is actually a convolutional process, except with the intervening nonlinearity. The implementation itself is done in parallel, in this case with 36 instantiations of the weighting, summing, and nonlinearity process (for a 6x6 output field). However, the bias value b and the weight W_0 is identical for all 36 processes.

values - two per input field - and far fewer connections as well, since the operation is subsampling blocks as opposed to convolutional.

The intimate relationship between the layers and spatial information in CNNs renders them well-suited for image processing, and they generally perform well at autonomously extracting salient features from images. One of the seminal papers on the topic LeCun et al. (1998) describes an application of CNNs to the problem of handwriting analysis. In some cases Gabor filters have been used as an initial pre-processing step to emulate the human visual response to visual excitation Kwolek (2005). In more recent work, researchers have applied CNNs to various machine learning problems including face detection Tivive and Bouzerdoum (2003), Chen et al. (2006), document analysis Simard et al. (2003) , data fusion Szarvas et al. (2006), and speech detection Sukittanon et al. (2004). CNNs have recently Mobahi et al. (2009) been trained with a temporal coherence objective to leverage the frame to frame coherence found in videos, though this objective need not be specific to CNNs. In Bengio (2009), there is some speculation as to why CNNs can be trained even with multiple layers. One idea is that the small fan-in of each neuron allows the gradients to travel through the layers without much disturbance, thus keeping their significance and allowing multiple layers to focus and converge on meaningful weightings. (The small fan-in refers to the fact that each neuron has fairly few numbers of inputs, just a few pixels.) Another idea is that the entire network begins in a favorable

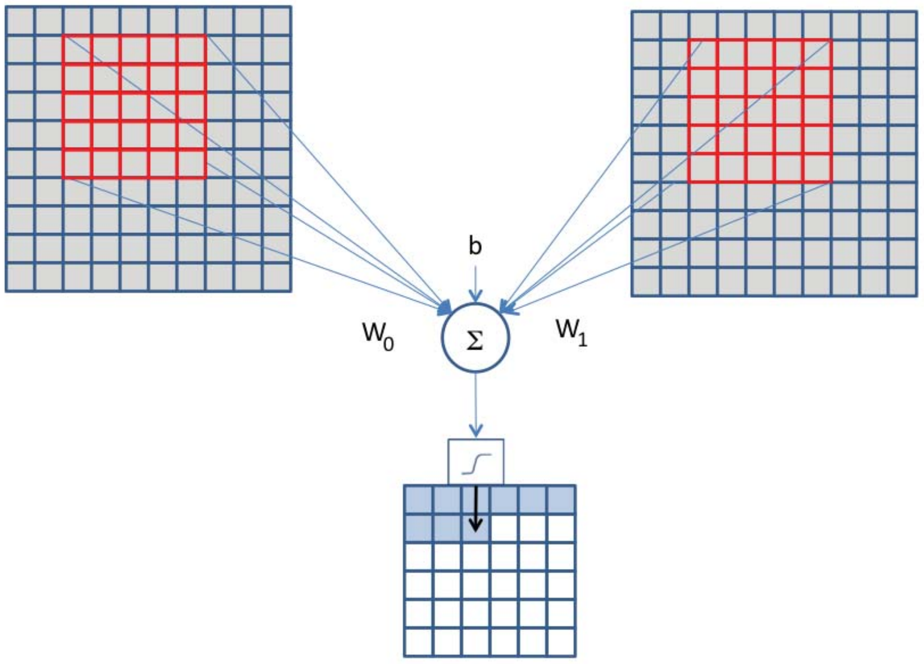


Figure 2.6: Depiction of multiple convolutional fields. In this case, the input field on the left and the field on the right are summed together, with different sets of weights but an identical bias value. In many implementations, such as the handwriting digit application of LeCun et al. (1998), the input fields for this type of processing in a CNN are the "S" fields, which are combined with various combinations to produce the next layer.

starting position due to the hierarchical connectivity structure and thus is well-suited for vision tasks.

2.3.2 Deep Belief Networks

A different DML approach which utilizes a hierarchy of specialized neural networks is the Deep Belief Network or DBN. As opposed to the CNN, which overall takes the form of a special purpose neural network, the DBN Hinton et al. (2006) is composed of several layers of Restricted Boltzman Machines or RBMs. Therefore, to understand RBMs, we begin by discussing associative neural network architectures which learn to encode training vectors in an internal representation. There is a natural progression of variants of these types of networks and we adapt the discussion from Rojas (1996) by discussing basic associative networks including autoassociative and heteroassociative networks, Hopfield networks, Boltzman Machines, and finally Restricted Boltzman Machines.

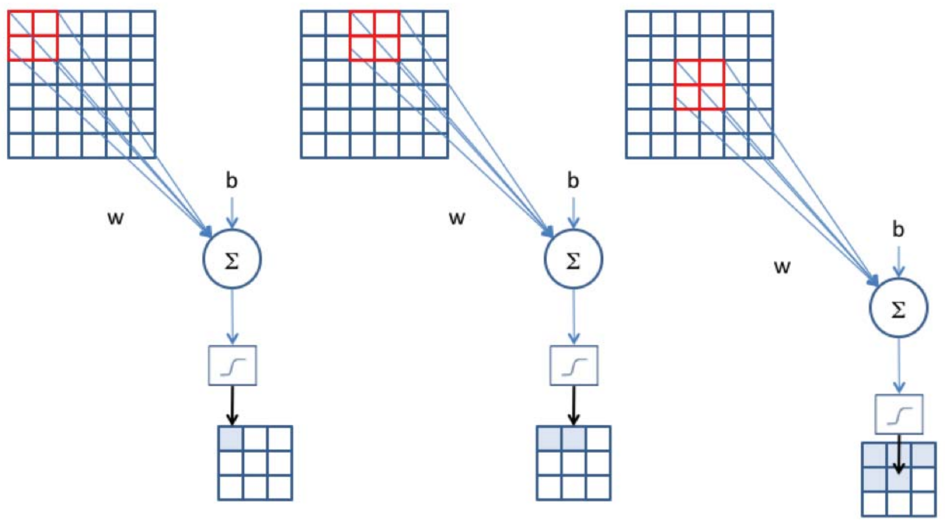


Figure 2.7: Depiction of the "S" layer. The overall effect is to scale the image to $\frac{1}{2}$ the size of the input, through a trainable weighting factor and a bias value. The weighting factor in conjunction with the bias can produce anything from a straight averaging of the data to an AND or OR operation (min and max).

Associative Networks

An associative network consists of a mapping of a set of m input vectors, o, in n-space to m output vectors, y, in k-space which should be robust to noise or distortions. When $o = y$ the network is an autoassociator Rumelhart et al. (1986) or AA; when $o \neq y$ the network is a heteroassociator. The network is composed of an activation from a weighted transformation of the input vector in the form

$$y = f(oW) \tag{2.22}$$

In general W is $n \times k$ but for autoassociators $n = k$. Thus when $f(x)$ is a linear function and there is no feedback the $n \times n$ weight matrix W can be found through least-squares methods or even by exact solution, although the trivial solution of the identity matrix compromises the ability of the network to generalize. More complex behavior can be obtained through the use of feedback, but in the case of linear activation the natural attractors of such a system are the eigenvectors of W and repeated feedback operations will map any input x to the dominant eigenvector. If the input at t is o_t, then the output at $t+1$ is given as

$$o_{t+1} = o_t W \tag{2.23}$$

A fixed point is reached where $o_{t+1} = o_t$ for all t which is the fundamental definition of an eigenvector for an eigenvalue of 1. For some W (such as a rotation matrix) there is a cyclical relationship where no fixed point is ever reached. Clearly the most desirable W are those with a complete set of eigenvectors which therefore should be capable of storing n values. However with feedback the largest magnitude eigenvector dominates and thus restricts the storage utility of this approach. A more powerful solution is to introduce a nonlinearity into the system for the activation function $f(x)$. As a simple example the sign function can be used, defined as

$$sgn(x) = \begin{cases} -1 & x < 0, \\ 1 & x > 0. \end{cases} \tag{2.24}$$

The introduction of even this simple nonlinearity requires a different learning approach than standard linear algebra. Hebbian learning is a method which uses the correlation between desired outputs and inputs to create weight matrices and is inspired by neurological studies Hebb (2002). Hebbian learning updates the weight matrix elements by incrementing each element with a value given by

$$\Delta w_{ij} = \eta o_i y_j \tag{2.25}$$

where η is a constant learning rate. Ideally this produces a weight matrix which is the sum of the correlation matrices for each input-output pair:

$$W = \sum_{i=1}^{m} W^i \tag{2.26}$$

For a set of input vectors x which are orthogonal, this relationship provides the desired output since the product between $o^p$and W is

$$o^p W = \sum_{i=1}^{m} o^p W^i \tag{2.27}$$

$$o^p W = o^p W^1 + o^p W^2 + ... + o^p W^m \tag{2.28}$$

Each individual component in the summation above then can be expressed as the vector

$$o^p W^j = \left[\begin{array}{cccc} o^p y_1^j o^j & o^p y_2^j o^j & ... & o^p y_n^j o^j \end{array} \right] \tag{2.29}$$

$$o^p W^j = \left\langle o^p, o^j \right\rangle y^j \tag{2.30}$$

When $p = j$ the dot product is positive and the activation is simply

$$sgn\left(o^p W^p\right) = sgn\left(\left\langle o^p, o^p \right\rangle y^p\right) \tag{2.31}$$

$$sgn\left(o^p W^p\right) = sgn\left(y^p\right) \tag{2.32}$$

When $p \neq j$ if the o vectors are orthogonal then the vector product is 0 and the function is 0; thus the summation in (2.28) will be 0 for these contributions and the network functions as a memory as desired. Vectors which are not in the input set will have a non-zero residual or crosstalk term, but should converge to the desired output through the feedback recursion. However, generally autoassociative memories have limited capacity due to correlation between patterns which increases these crosstalk terms.

There are other methods for finding the weight matrix which can offer better performance than Hebbian learning Rojas (1996). In Bourlard and Kamp (1988) it is shown that an autoassociator with a hidden linear layer can be trained to create a principle component analysis (PCA) representation of input data, where the N hidden units represent the first N PCA components. The realization that these architectures are performing PCA make linear hidden layers more an academic curiosity rather than a serious machine learning tool; however, when the hidden elements are nonlinear, much different results are obtained as explained in Japkowicz et al. (2000). This ability to extend the autoassociator to nonlinear functions but still easily train the AA suggests that these elements can be part of a powerful deep learning architecture. Past experiments and results have used different approaches to constrain or train AAs to generalize. It is easy to imagine that some training methods could cause overtraining and thus reduce the generalization power of AAs. As an example cited by Bengio (2009), if an AA is given an input of dimension D and the encoding is also D or greater, there is a fear that the AA will simply learn the training examples or the identify function. There are a few strategies that have been deployed to defeat this, such as stochastic gradient descent Bengio et al. (2007a) or constraints such as sparseness Olshausen and Field (1997).

Bidirectional Associative Memories and Energy

We now introduce the concept of a network energy function through a special associative network which uses feedback through shared input and output connection weights, the bidirectional associative memory (BAM) or resonance network Kosko (1988). In this case we consider an input at time t of o which maps to an output value y in the usual manner as in (2.22), except that feedback comes from mapping y_t back through the same weights to produce o_{t+1}. Depending on the weighting of the network, repeated applications of this operation can cause the o and y values to converge to stable values after some finite t. Consider vector o_0 which is introduced to a BAM and produces y_0 through

$$y_0 = sgn\left(o_0 W\right) \tag{2.33}$$

Applying this back through produces

$$o_1 = sgn(W y_0^T) \tag{2.34}$$

If $o_1 = o_0$ then the vector pair(o_0, y_0) produces a stable state for the network. We can see that if this condition is met then there will be high correlation between $W y_0^T$ and o_0 and their product will be large. The negative of this product is used to define the *energy* of the network, expressed here as

$$E(o_i, y_i) = -\frac{1}{2} o_i W y_i^T \tag{2.35}$$

The energy of the recurrent network gives a measure of how close a given pair of values are to stable states for the network, or alternately the "natural" minima of the system. The energy term is different for other activation functions, for example a threshold followed by a step function has an energy of

$$E(o_i, y_i) = -\frac{1}{2}o_i W y_i^T + \frac{1}{2}\theta_r y_i^T + \frac{1}{2}o_i \theta_l^T \tag{2.36}$$

where θ_l are the thresholds of the left units (where the o vectors are applied) and θ_r are the thresholds of the right units (where the y vectors are applied).

Hopfield Networks

The Hopfield network Hopfield and Tank (1985) can be depicted as a BAM where the computing elements do not fire in synchronous fashion but rather are randomly selected, and thus the network energy is recomputed at discrete time intervals in response to the change of a single element (as opposed to an entire application of a new vector stimulus). In the Hopfield network an element has a binary state, typically in the bipolar set $\{-1, 1\}$ or binary set $\{0, 1\}$. An excitation is computed using the current state settings and the weight matrix. Elements are then selected at random and a selected element will change its state according to the excitation and a threshold such that when the excitation is greater than the threshold the state becomes 1, and when less becomes -1. Only one unit fires per time step. All units are connected to all other elements except there is no self-connection. A Hopfield network of n elements thus has a weight matrix which is sized $n \times n$ and the diagonal elements are all 0 (due to the lack of self-connection). The energy function of a Hopfield network is

$$E(o) = -\frac{1}{2}oWo^T + \theta o^T \tag{2.37}$$

where θ is the row vector of the unit thresholds. Hopfield networks have been used to solve optimization problems such as the traveling salesman problem (TSP) Hopfield and Tank (1985), although this approach requires fairly large compute resources and is not guaranteed to find a global minimum. The TSP is a famous optimization problem where a list of cities must each be visited at least one time by a salesman, who must choose a route that has the minimum round trip distance.

Boltzman Machines

In the Hopfield network, the element selected for asynchronous update is chosen at random, but the selected element uses a deterministic method for applying state changes based on the activation value. The Boltzman Machine generalizes the Hopfield network to a case where the state transitions are stochastic. The addition of noise to the network allows the avoidance of shallow local minima and also is a more reasonable approximation of true biological elements which have been shown to have stochastic properties Trappenberg (2009). In the Boltzman machine, a global "temperature" T is chosen which represents the noise in the system and an element is updated according to a probability distribution which is a function of T. When T is small the probability of a transition is mostly a function of the excitation. When T is larger, the updates become more random and less dependent on the activation. Clearly so long as T is non-zero the Boltzman machine cannot be termed to be "stable" in the sense that it can always change in some fashion. With a simulated annealing approach the value of T can be decreased over time to allow the network to settle into a stable state, however. When the probability of transitioning from a state with energy E_j to E_i is given by

$$p_{ij} = \frac{1}{1 + \exp\left(\frac{E_j - E_i}{T}\right)} \tag{2.38}$$

the system will reach an energy equilibrium point which is governed by the Boltzman distribution given as

$$p_i = \frac{\exp(-\frac{E_i}{T})}{Z} \tag{2.39}$$

where the normalizing factor Z is given as

$$Z = \sum_{i=1}^{m} \exp(-\frac{E_i}{T}) \tag{2.40}$$

An element in a Boltzman machine can accept external inputs as well as fed-back values from the other elements. An element which accepts external inputs is called a visible unit and the group of elements with this property is called the visible layer, with hidden inputs (and layers) consisting of the other elements which do not accept external values.

In Boltzman learning Ackley et al. (1985),Hertz et al. (1991),Rojas (1996) the network is trained to create an internal "generative" model. A generative model means an observation which is "near" training examples will cause the network to generate a probability distribution which attempts to explain the observation based on past training examples. Generative models provide a joint probability distribution over observable data and labels, facilitating the estimation of both $p(o|\varpi)$ as well as $p(\omega|o)$, while discriminative models are limited to the latter Bishop et al. (2006). The network has m input units and k hidden units, with a total of $n = m + k$ units overall. Each unit is binary as in the Hopfield model. An input state for a network with 5 units, for example, could therefore be [01011] , or [00011], or [11101], etc. Note that the states of the input units are NOT set by the input themselves; the input is only one factor in their state. With that caveat in mind, the probability that the input units are in a particular state α is given by summing over all the states of the hidden units β, given as

$$P_\alpha = \sum_\beta P_{\alpha\beta} \tag{2.41}$$

The Boltzman distribution itself allows us to write this term as

$$P_\alpha = \frac{1}{Z}\sum_\beta \exp(-\gamma E_{\alpha\beta}) \tag{2.42}$$

where the Z term in the denominator is a normalization factor, and is given as

$$Z = \sum_{\alpha,\beta} \exp(-\gamma E_{\alpha\beta}) \tag{2.43}$$

We see that the probability of entering a particular state thus depends on the energy associated with that state across all the possible states in the hidden layers. This energy term is given as

$$E_{\alpha\beta} = -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\omega_{ij}x_i^{\alpha\beta}x_j^{\alpha\beta} \tag{2.44}$$

Here $x_i^{\alpha\beta}$ refers to the state value of node i when the overall system state is α for the input nodes and β for the hidden nodes. The goal of Boltzman learning is for the probability distribution P_α to match a desired input distribution R_α. This is set by supplying an input vector to the system which provides each input node with an external excitation (which is added to the rest of the self-excitation of the network). A *log* distance between the desired distribution and the current distribution is given as

$$D = \sum_\alpha R_\alpha \log \frac{R_\alpha}{P_\alpha} \tag{2.45}$$

Clearly if $P_\alpha = R_\alpha$ the distance will be 0 thus the learning algorithm should minimize D, which can be done by gradient descent using a weight update rule with learning rate η:

$$\Delta\omega_{ij} = -\eta\frac{\delta D}{\delta\omega_{ij}} = -\eta\frac{\delta}{\delta\omega_{ij}}\left\{\sum_\alpha R_\alpha \log R_\alpha - \sum_\alpha R_\alpha \log P_\alpha\right\} \tag{2.46}$$

Since R_α is not a function of ω_{ij}, we remove that term and have

$$\Delta\omega_{ij} = -\eta\frac{\delta D}{\delta\omega_{ij}} = \eta\frac{\delta}{\delta\omega_{ij}}\left\{\sum_\alpha R_\alpha \log P_\alpha\right\} = \eta\sum_\alpha \frac{R_\alpha}{P_\alpha}\frac{\delta P_\alpha}{\delta\omega_{ij}} \tag{2.47}$$

We can now combine the terms above (2.42),(2.44), and (2.43), focusing on the derivative of P_α with respect to the weight:

$$\frac{\delta P_\alpha}{\delta\omega_{ij}} = \frac{\gamma\sum_\beta \exp(-\gamma E_{\alpha\beta})x_i^{\alpha\beta}x_j^{\alpha\beta}}{Z} - \frac{\gamma\left\{\sum_\beta \exp(-\gamma E_{\alpha\beta})\right\}\sum_\lambda\sum_\mu \exp(-\gamma E_{\lambda\mu})x_i^{\lambda\mu}x_j^{\lambda\mu}}{Z^2} \tag{2.48}$$

This simplifies to the expression

$$\frac{\delta P_\alpha}{\delta \omega_{ij}} = \gamma \left\{ \sum_\beta P_{\alpha\beta} x_i^{\alpha\beta} x_j^{\alpha\beta} - P_\alpha \langle x_i x_j \rangle_{free} \right\} \tag{2.49}$$

where $\langle x_i x_j \rangle_{free}$ denotes the expected value of the product of states x_i and x_j when the network is freely running. Substitute back into (2.47), we get

$$\Delta\omega_{ij} = \eta\gamma \left\{ \sum_\alpha \frac{R_\alpha}{P_\alpha} \sum_\beta P_{\alpha\beta} x_i^{\alpha\beta} x_j^{\alpha\beta} - \sum_\alpha R_\alpha \langle x_i x_j \rangle_{free} \right\} \tag{2.50}$$

For final subtitutions, we note that the joint probability

$$P_{\alpha\beta} = P_{\beta|\alpha} P_\alpha \tag{2.51}$$

can be substituted into the first term in the brackets of (2.50) to yield

$$\sum_\alpha \frac{R_\alpha}{P_\alpha} \sum_\beta P_{\beta|\alpha} P_\alpha x_i^{\alpha\beta} x_j^{\alpha\beta} \tag{2.52}$$

$$\sum_\alpha \sum_\beta R_\alpha P_{\beta|\alpha} x_i^{\alpha\beta} x_j^{\alpha\beta} \tag{2.53}$$

$$\Delta\omega_{ij} = \eta\gamma \left\{ \sum_\alpha R_\alpha \langle x_i x_j \rangle_{fixed} - \sum_\alpha R_\alpha \langle x_i x_j \rangle_{free} \right\} \tag{2.54}$$

The summations weighted by R_α can be removed to get

$$\Delta w_{ij} = \eta\gamma(\langle x_i x_j \rangle_{fixed} - \langle x_i x_j \rangle_{free}) \tag{2.55}$$

We see thus that in Boltzman learning, the weights can be trained by setting a fixed input (also called "clamping" an input) and running the network to compute the value $\langle x_i x_j \rangle_{fixed}$ (the average correlation between the states of two nodes while a fixed input is applied). A second pass then is run without the inputs (the "free" pass) where again the average correlation between states $\langle x_i x_j \rangle_{free}$ is computed. Thus the weight update is based on the correlation between network states generated when inputs are applied but also seeks to measure and therefore ignore the results of noisy, spurious correlations that are generated and are not indicative of the desired direction of the local minimum of error. In summary, in Boltzman learning the application of external inputs is used to stabilize the network at energy states that allow the network to model the probability distribution of the inputs.

Restricted Boltzman Machines and DBNs

Boltzman machines are an interesting academic construct but have not proven useful beyond basic "toy" problems. The issue here is their large number of connections which makes training process very long. A more useful network is the Restricted Boltzman Machine or RBM, which limits layers to a single visible and hidden layer, and does not allow connections within a layer. As shown previously, the hidden units are trained to capture higher-order data correlations that are observed at the visible units.

Training an RBM can be done by applying 2.55 in the following manner Hinton (2002) (also described in simple terms in Trappenberg (2009)). A vector is presented to the visible units that forward values to the hidden units. From 2.55, we can identify the x_i as the visible units, and the x_j as the hidden units. This triggers a probabilistic response based on the current weight values (i.e., w_{ij}), and the states that result are used to form an initial estimate of the correlation between the nodes $\langle x_i x_j \rangle^{t=0}$. The next sampling phase simply uses the hidden layer states to recompute the visible layer states, forming a new correlation estimate $\langle x_i x_j \rangle^{t=1}$, and the process repeats an infinite number of times to yield over time the weight change given in (2.55). Each of these back and forth steps is known as a Gibbs sampling.

While clearly extending the sampling to $t = \infty$ is not practical, it has been shown that only a single step is needed to approximate maximum likelihood learning. This process is improved by the use of multiple training examples, as each step for a particular training example forms an approximation of the correlation, and over the course of many examples a good estimate can be found. We note that there is a trainable bias level applied to each input node, as well as to each hidden node. These weights must be trained in a similar manner. When we seek to train the RBM to perform a classification task, for example supervised learning, we add an additional layer on top which we designate the "label layer'. This layer has an additional set of biases as well as an additional set of weighting factors.

It is beneficial to understand the training of an RBM before proceeding into the DBN discussion. We note that there is an excellent matlab source code for RBM training demonstration purposes Karpathy (2011). Other source of RBM and DBN code are available from Theis et al. (2010) and Hinton and Salakhutdinov (2006). For this discussion, we have an RBM illustrated in Figure 2.8, with an input layer, a hidden layer, and an additional output layer. In this case, we use a database of handwritten digits which are 28×28 in size, so each input is 784 elements (and thus there are 784 units in the input layer). The hidden layer is set to have 90 units. The output layer has 10 units, since we want to classify the digits from 0 to 9. For training, we use the contrastive divergence in the following manner. First, one of the input data samples is placed on the input, and its class label (a vector of $D = 10$, which is all 0 except for element k, i.e., the 0th element for a "0" example image) is placed on the output layer. At the hidden layer, we receive a stimulus that is equal to $oW + b + cW_c$, where o is the observation (a 784-element vector), W is the input to hidden layer weight matrix (784×90), b is the bias level at the hidden layer (a 90×1 vector), c is the class label vector (10×1), and W_c is the weighting factor from class to hidden (10×90). This composite 90-element vector now is applied to the sigmoid function, which gives us the probability that each hidden unit is "on".

We now sample the hidden units; essentially we generate a random value at each unit, and turn the unit on or off stochastically (assuming binary nodes). We thus now have a vector of 90 elements in the hidden layer which tells us which units are "on" and "off", which we will denote as h. We then go downward to the inputs, taking the 90 element vector back through the weights and adding the bias value on the inputs, $hW + d$, where d is the 784×1 dimensional input bias level. We thus have a 784-element vector on the inputs, which after passing through the sigmoid functions gives us the probability that each input node is "on"; we sample this as for the hidden layer. Now, note that we have a signal on the input that was generated by the actual data input, and a signal that is generated by the contrastive divergence, which we will denote as o_{cd}.

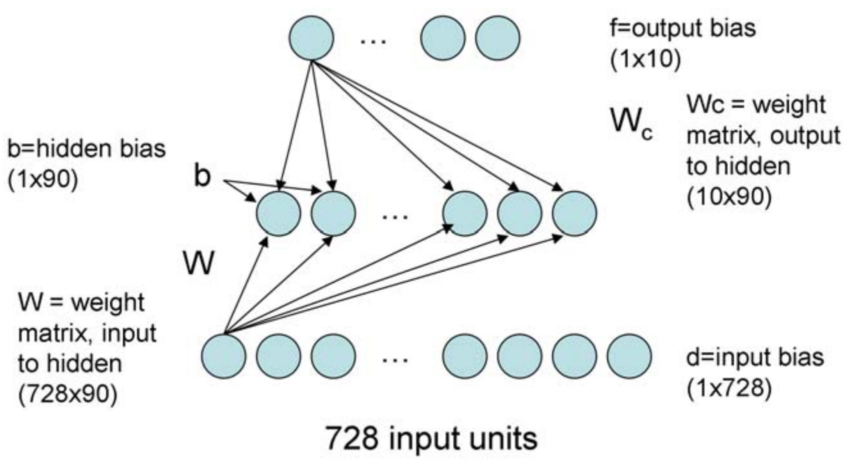
Because we are performing supervised training, we also take the h vector and apply it away from the hidden layer toward the top layer. In this case we create $hW_c' + f$, where f is the trainable bias on the output layer. As with the input, we pass this through a nonlinearity (such as a softmax function) and sample again, constraining to a single element "on" since this is supposed to be a class label. This gives us an output signal c_{cd} generated by the contrastive divergence.

We can now proceed back inward again, toward the hidden layer from the output signal c_{cd} and the input signal o_{cd}, and now generate our "final" hidden vector h_{free}. We use the term "final" here since we are using contrastive divergence, and NOT processing toward ∞ before updating the weight. The weight W is updated based on 2.55, with the update increment weighted by $o_i h_i - o_{cdi} h_{cdi}$ for W; the bias on input d updated by $o_i - o_{cdi}$; the bias on the hidden units b updated by $h_i - h_{cdi}$; the hidden-to-output weight W_c increment weighted by $c_i h_i - c_{cdi} h_{cdi}$; and the output bias f updated by $c_i - c_{cdi}$. These may then be applied by further weight by the learning rate, and or by a variety of other weighting factors such averaging over many samples, or smoothing over several samples. There are often other heuristics applied as well, such as separating the data into batches and doing updates only after each batch. In the absence of a supervised label, we can simply envision this process with the weight matrix W_c forced to 0 for all iterations, so that we perform solely unsupervised learning. The process is depicted graphically by example in Figures 2.9 to 2.10. After training, the response to a particular input can be found by applying all candidate test labels and choosing the classification which gives the lowest average energy in the system based on the sampled values of h.

In a DBN, layers of RBMs as described are linked by using the hidden node of a lower layer as the input or visible layer for the next step in the hierarchy. Each layer added to the network improves the log-probability of the training data, which we can think of as increasing true representational power. The top layers of a DBN are autoassociators which use supervised learning to define the system output, as discussed earlier. The difficult computation cost in DBNs is in the training of the network. The training of the DBN consists of the following steps. First, the network is trained layer-by-layer, just as described for the RBM. The bottom most layer is trained by presenting images of the training set and using the greedy algorithm for contrastive divergence. This can be repeated some number of times. Then, the next layer is trained by again presenting input images to the bottom layer, propagating each one through the bottom layer to the next layer, then using the contrastive divergence algorithm to update the weights and biases for this layer. This is then repeated again and again until the second layer is finished. Then, we repeat, propagating through the bottom, next, and new training layer again. The top layer is trained with the known labels as part of the input. The pseudocode of Bengio (2009) is very useful in attempting to explain DBN greedy layer-by-layer training.

After the greedy layer-by-layer training, the network is "fine tuned" with supervised learning and backpropagation to improve the system response. The "up-down" algorithm of Hinton et al. (2006) starts by taking a sample input and propagating through the network to stochastically set the values in the hidden units. Then the weights are adjusted using multiple iterations of the Gibbs sampling. It is unclear why it was determined that multiple iterations were needed, other than this gives the ability to produce a more accurate answer at the cost of additional training time. Finally, during the down pass the actual label is applied and backpropagated down. However, in this phase only the bottom few layer weights are changed; the top ones are not changed. The Gibbs sampling in this stage seems to require more than a pair of iterations through the data; in the case of Hinton et al. (2006), for 100 epochs an up pass was performed, 3 iterations of Gibbs samples were used, then the down-pass was performed. Then, for the next 100 epochs, six iterations were used, and for the last 100, 10 iterations were used. Other training strategies have been employed to improve the performance of DBNs on purely discriminative tasks Hinton (2007).

It has been shown by Hinton and Salakhutdinov (2006) that the pre-training of DBNs yield better performance than those trained exclusively with back-propagation. This may be intuitively explained by the fact that the pre-training "gets the system close" to the true weights, and then subsequent back-propagation for DBNs is only required to perform a local search on the weight (parameter) space, speeding training and convergence time in relation to traditional feed-forward neural networks alone. One interesting property of DBNs is the ability to generate images of the target values through a top-down procedure. This works by placing the desired target on the output layer and using the activation through sampling to observe the input patterns.



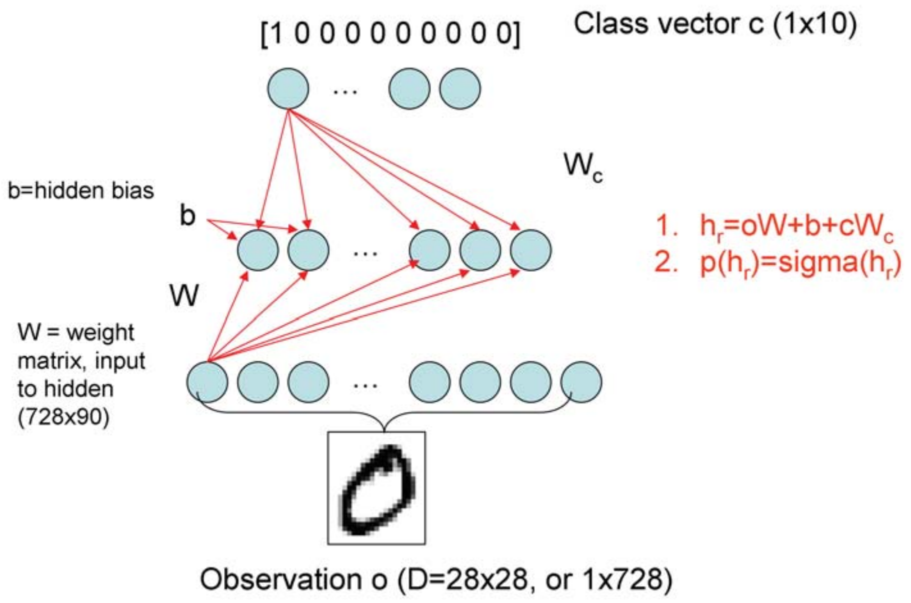


Figure 2.8: Top: An example RBM for supervised learning. This RBM has 728 input layers from a 28x28 pixel image, 90 hidden layers, and an output layer with 10 units to classify data as ten different labels. Bottom: An image is applied (an image of a "0") to the inputs, such that the image is rasterized at the input nodes. The corresponding class label target is applied to the output. The weight matrices are then applied, with additive bias, to produce a summed signal at the hidden layer. Each is passed through the sigma function to yield a probability, $p(h|data)$. Note that the probability is a vector telling us the probability of setting each hidden layer to a 1 or 0.

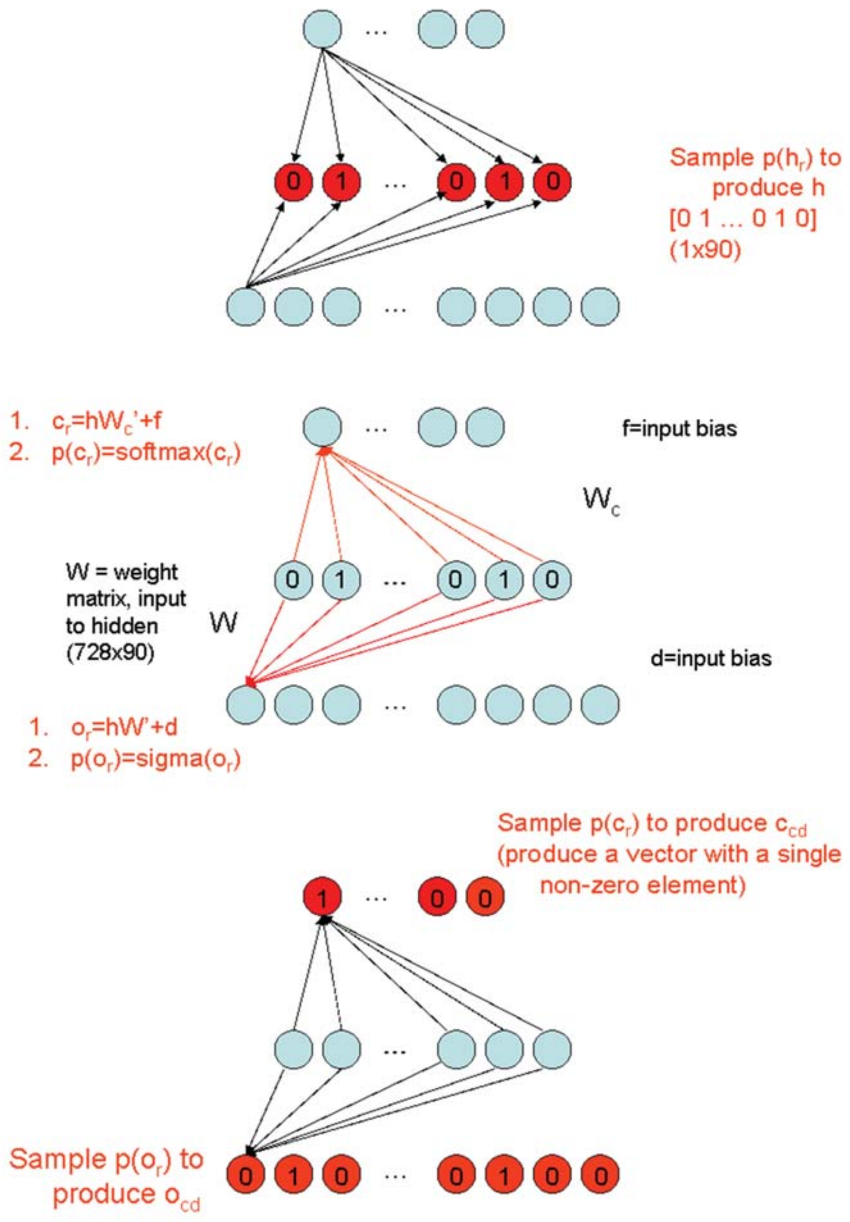
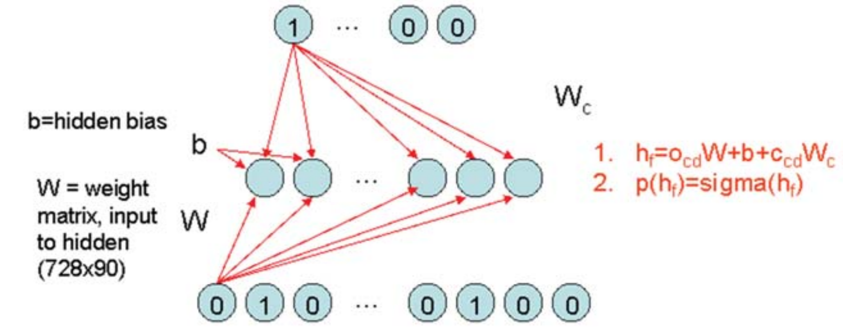

Sample p(h_r) to produce h [0 1 … 0 1 0] (1x90)
1. c_r=hW_c'+f
2. p(c_r)=softmax(c_r)
f=input bias
W_c
W = weight matrix, input to hidden (728x90)
W
d=input bias
1. o_r=hW'+d
2. p(o_r)=sigma(o_r)
Sample p(c_r) to produce c_cd (produce a vector with a single non-zero element)
Sample p(o_r) to produce o_cd

Figure 2.9: Top: The probability is sampled by generating a random 0 or 1 based on the computed probability. Middle: The hidden values are then propagated back to the input and output, generating new probabilities at these nodes. Bottom: The input nodes are then sampled, much as with the hidden nodes. The output sampling is special, since the output is a target vector (all 0 except for a single unit set to 1).



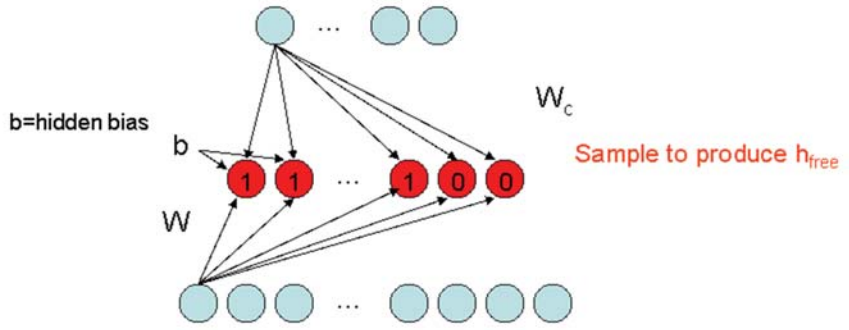


Figure 2.10: Top: As in the image input, the o_{cd} and c_{cd} signals are used to generate a new hidden value. Bottom: The final sampling generates the h_{free} signal. The error signal from h_{free} and h, along with o, o_{cd}, c, and c_{cd}, is used to generate the adjustment signal for the weight matrices and biases as described.

Current Research Thrusts in DBNs

Performance results obtained when applying DBNs to the MNIST handwritten character recognition task LeCun et al. (1998) have demonstrated significant improvement over feedforward networks. Shortly after DBNs were introduced, a more thorough analysis Bengio et al. (2007b) solidified their use with unsupervised tasks as well as continuous valued inputs. Further tests in Ranzato et al. (2007b), Larochelle et al. (2007) illustrated the resilience of DBNs (as well as other deep architectures) on problems with increasing variation. Some more recent work in this field has established the computational hardness of RBMs Long and Servedio (2010).

DBNs do not inherently embed information about the 2D structure of an input image, i.e. inputs are simply vectorized formats of an image matrix. Also, the DBN structure does not scale well to different size images, since each input node is connected to each hidden node. Thus, the number of connections is at least directly proportional to the input image size, and is likely greater than that as the number of hidden layers will increase with increasing image size. We note that Lee et al. (2009) introduces the notion of Convolutional Deep Belief Networks (CDBNs) which are geared towards image processing and utilize the spatial relationship of neighboring pixels with "convolutional RBMs" to provide a translation invariant generative model.

We note also that DBNs do not currently explicitly address learning the temporal relationships between observables, though there has been recent work in stacking temporal RBMs Sutskever and Hinton (2007) or generalizations of these, dubbed temporal convolution machines Lockett and Miikkulainen (2009), for learning sequences.

We may categorize DBNs as one type of a more general architecture where stacks of associators are used to create deep networks. In addition, work has continued on the use of alternate building blocks in deep networks, including stacked auto-encoders Bengio et al. (2007b), Ranzato et al. (2007b), Vincent et al. (2008). These efforts produced deep multi-layer neural network architectures that can be trained with the same principles as DBNs but are less strict in the parameterization of the layers. Unlike DBNs, auto-encoders use discriminative models from which the input sample space cannot be sampled by the architecture, making it more difficult to interpret what the network is capturing in its internal representation. However, it has been shown Vincent et al. (2008) that denoising auto-encoders, which utilize stochastic corruption during training, can be stacked to yield generalization performance that is comparable to (and in some cases better that) traditional DBNs.

2.3.3 Cortical-inspired Deep Learning Architecture

Biomemetics Bar-Cohen (2006) refers to the study and imitation of natural processes and mechanisms in engineering and scientific innovation. A simple example is the use of fins to improve swimming, but more sophisticated examples abound such as Velcro (mimicking spiky, sticky plant seed transport mechanisms) and controlled adhesion based on gecko skin. Clearly artificial intelligence has long been inspired by biological processes, particularly the human and animal ability to rapidly, concisely process real-world information. Indeed the very field of machine learning can be interpreted as bio-inspiration at its heart. As mentioned earlier the neural network computational model is inspired by biology as well, but systems that seek to actually imitate brain and cortical processes are fairly new in artificial intelligence.

There are several computational architectures that attempt to model the neocortex. These models have been inspired by sources such as Marr et al. (2010), which attempted to map various computational phases in image understanding to areas in the cortex. Over time these models have been refined, however the central concept of visual processing over a hierarchical structure has remained. These models invoke the simple-to-complex cell organization of Hubel and Weisel Hubel and Wiesel (1962), which were based on studies of the visual cortical cells of cats. Similar organizations are utilized by CNNs as well as other deep-layered models (such as the Neocognitron Fukushima (2003), Fukushima (2005), Fukushima (1980) and HMAX Serre et al. (2007), Riesenhuber and Poggio (1999)), yet more "explicit" cortical models seek a stronger mapping of their architecture to biologically-inspired models. In particular, they attempt to solve problems of learning and invariance through diverse mechanisms such as temporal analysis, in which time is considered an inseparable element of the learning process. In Nordlie et al. (2009), a framework for biomemetic architectures was posed which can be used to fit these new computing paradigms into a context of brain-emulation.

One prominent example is Hierarchical Temporal Memory (HTM) George (2008). HTMs have a hierarchical structure based on concepts described in Hawkins and Blakeslee (2005) and bear similarities to other work pertaining to the modeling of cortical circuits. With a specific focus on visual information representation, in an HTM the lowest level of the hierarchy receives its inputs from a small region of an input image. Higher levels of the hierarchy correspond to larger regions (or receptive fields) as they incorporate the representation constructs of multiple lower receptive fields. In addition to the scaling change across layers of the hierarchy, there is an important temporal-based aspect to each layer, which is created by translations or scanning of the input image itself. During the learning phase, the first layer compiles the most common input patterns and assigns indices to them. Temporal relationships are modeled as probability transitions from one input sequence to another and are clustered together using graph partitioning techniques. When this stage of learning concludes, the subsequent (second) layer concatenates the indices of the current observed inputs from its children modules and learns the most common concatenations as an alphabet (another group of common input sequences, but at a higher level). The high layer's characterization can then be provided as feedback down to the lower level modules. The lower level, in turn, incorporates this broader representation information into its own inference formulation. This process is repeated at each layer of the hierarchy. After a network is trained, image recognition is performed using the Bayesian belief propagation algorithm Pearl (1988) to identify the most likely input pattern given the beliefs at the highest layer of the hierarchy (which corresponds to the broadest image scope). In later implementations of the HTM Numenta (2011a), the algorithms have undergone substantial changes, with a more biomimetic approach that combines an RBM-like neuron model with a spatial "inhibition" and "excitation" method that is inspired by Kohonen self-organizing maps. The HTM concept seems to be a work-in-progress, with each release featuring additions and modifications over the past development. However, the work is not well documented in the open academic literature, although the Numenta software packages are freely available for download and usage Numenta (2011b). There are other architectures proposed in the literature, which resemble HTMs, include the Hierarchical Quilted SOMs Miller and Lommel (2006) that employ two-stage spatial clustering and temporal clustering using self-organizing maps, and the Neural Abstraction Pyramid Behnke (2003). Although interesting and innovative, most of these and related works are very active areas of research at this point.

Chapter 3

DeSTIN Formulation

3.1 Technical Approach

We introduce a novel DML architecture dubbed the Deep Spatio-Temporal Inference Network or DeSTIN Arel et al. (2009b) Arel et al. (2009a). In the DeSTIN architecture a hierarchy of layers is used, where each layer consists of multiple instantiations of an identical circuit or node which follow a defined spatial orientation for imaging applications. Each node learns to generalize and represent a temporal sequence of observations through unsupervised learning. The lowest layer of the hierarchy processes raw data input, such as image pixels, and continuously constructs a belief state that attempts to characterize the sequences of patterns viewed. The second layer, and all those above it, receive as input the belief states of nodes at their corresponding lower layers, and attempt to construct belief states that capture regularities in their inputs. In addition, feedback from the upper-layer (or parent) node is received and utilized in the formulation of the belief state. The architecture thus forms a hierarchical belief state across its layers which captures both spatial and temporal regularities in the data - a key advantage over existing deep learning schemes. The temporal changes may occur through movement of the imager, movement of the subject, or a combination of both, but for our initial experiments the movement and temporal change is induced through moving the field of view of the imager. Higher layers receive temporally changing data by processing the belief outputs of children nodes. These output beliefs capture regularities in their inputs and thus extract features of the data which can be fed to a supervised learning algorithm to perform classification. In principle, since each node is identical, the architecture can be mapped to parallel computational platforms such as graphics processing units, and since the overall processing is simple it is tractable for hardware-oriented approaches as well. All processing is "online" so that large amounts of memory are not needed during training.

The derivation of the learning / belief update rule for each node in DeSTIN is described as follows. At each node, and during any particular time step, an observation o is received. The node has a belief, b, which is a vector of real numbers with each element corresponding to a state variable, s. The node also receives feedback from a parent node which is denoted by a. Note that the temporal nature of these values are all suppressed, i.e., the observation is an explicit function of time as well. The node seeks to formulate a new belief at the subsequent time step which we will denote as $b\prime$, with the new state value $s\prime$. We therefore write the resulting expression as

$$b'(s'|a) = \Pr(s'|o, b, a) = \frac{\Pr(o, s', b, a)}{\Pr(o, b, a)} \tag{3.1}$$

We then expand the denominator to yield

$$b'(s'|a) = \frac{\Pr(o, s', b, a)}{\Pr(o|b, a)\Pr(b, a)} \tag{3.2}$$

and expand the numerator to yield

$$b'(s'|a) = \frac{\Pr(o|s', b, a)\Pr(s'|b, a)\Pr(b, a)}{\Pr(o|b, a)\Pr(b, a)} \tag{3.3}$$

and, finally, omit the terms common to the denominator and numerator $\Pr(b, a)$ to obtain

$$b'(s'|a) = \frac{\Pr(o|s', b, a)\Pr(s'|b, a)}{\Pr(o|b, a)} \tag{3.4}$$

We assume our observation o is only a function of the (true) state of the environment $s\prime$, or

$$\Pr(o|s', b, a) = \Pr(o|s') \tag{3.5}$$

Applying this simplification yields

$$b'(s'|a) = \frac{\Pr(o|s')\Pr(s'|b, a)}{\Pr(o|b, a)} \tag{3.6}$$

We next note that we can expand the second numerator term with vector b into individual elements $b(s)$ as

$$\Pr(s'|b, a) = \sum_{s \in S} \Pr(s'|s, a)b(s) \tag{3.7}$$

to yield

$$b'(s'|a) = \frac{\Pr(o|s')\sum_{s \in S} \Pr(s'|s, a)b(s)}{\Pr(o|b, a)} \tag{3.8}$$

The denominator term may be expanded using state variable s'' and s''':

$$\Pr(o|b, a) = \sum_{s''' \in S}\sum_{s'' \in S} \Pr(o|a, b, s'', s''')\Pr(s''|s''', a, b)\Pr(s'''|a, b) \tag{3.9}$$

If we assume the current observation is independent of the previous state s''' we can write this as

$$\Pr(o|b,a) = \sum_{s''\in S} \Pr(o|a,b,s'') \sum_{s'''\in S} \Pr(s''|s''',a,b) \Pr(s'''|a,b) \tag{3.10}$$

Next we note that given the state s'', there is no dependence of the observation o on a or b, so

$$\Pr(o|b,a) = \sum_{s''\in S} \Pr(o|s'') \sum_{s'''\in S} \Pr(s''|s''',a,b) \Pr(s'''|a,b) \tag{3.11}$$

Moreover, given the current state, the belief for that state is irrelevant, for which we have

$$\Pr(o|b,a) = \sum_{s''\in S} \Pr(o|s'') \sum_{s'''\in S} \Pr(s''|s''',a) \Pr(s'''|a,b) \tag{3.12}$$

Thus the fundamental belief update rule of DeSTIN is given as Arel et al. (2009b)

$$\boxed{b'(s'|a) = \frac{\Pr(o|s')\left\{\sum_{s\in S} \Pr(s'|s,a) b(s)\right\}}{\sum_{s''\in S}\left\{\Pr(o|s'')\sum_{s'''\in S} \Pr(s''|s''',a) b(s''')\right\}}} \tag{3.13}$$

which maps the current observation o, the belief b (with argument the system state s) and the belief state or advice of a higher-layer node a, to a new (updated) belief and state $b\prime(s\prime)$ at the next time step, with a normalization factor in the denominator.

One interpretation of (3.13) is that the (static) pattern similarity metric, $\Pr(o|s')$, is modulated by a construct that reflects the system dynamics, $\sum_{s'\in S} \Pr(s'|s,a)b(s)$. For shorthand, the latter is denoted as $P^a_{ss'}$. Therefore the belief state captures both spatial and temporal information and these two constructs are the main items which must be learned from the observations. The former is learned using online clustering, while the latter is learned from experience by adjusting of the parameters with each transition from s to $s\prime$ given a (as a tabular method or using function approximation methods). The advice generation method is learned using online clustering. The result is a robust framework that autonomously learns to represent complex data patterns, such as those found in real-life robotics applications and whose output can be used as a generic feature extractor for a supervised learning system.

3.2 Steps of the Learning Process

Here we briefly discuss the different learning steps involved in a DeSTIN implementation. We cover these elements in more detail in Chapters 4 and 5.

3.2.1 Adaptation of the Static Component

The static component of 3.13 is the term $P(o|s')$. This term represents the probability of an observation, given a particular state s'. We elected to pursue an unsupervised learning algorithm, in particular, an online clustering algorithm for this component. The objective here is to learn a concise set of states s that can be used to generalize the data. However, this is only one element for this component, as clustering with associated labeling generally only allows us to estimate $P(s|o)$, and in the hard-threshold method this probability is unity for the selected class and 0 for all others. In our initial work we approximated the $P(o|s)$ with the equation 4.1, which is actually a sort of fuzzy approximation to a probability. After initial study, we determined an additional learned construct must be included in the static learning component: a true (or at least better estimate) of $P(o|s)$. This can be learned from an initial set of cluster centroids by an estimate of a probability distribution. For our work, we experimented with specific models (Rayleigh, Gaussian and exponential), which was motivated by their simplicity and applicability based on an assumption of Gaussian data distributions but measured with cosine similarity (see Chapter 4 for additional details). We therefore estimate the parameters of the given model, then we cycle through the different states s and for each one estimate $P(o|s)$ by integrating around the observation in a closed-form solution. Note that we also replace the observation vector o with a scalar, simply the distance from the cluster s, which reduces dimensionality and has some analogies to nonlinear dimensionality reduction methods that are based on graphical models, as described in Chapter 4.

3.2.2 Adaptation of the Dynamic Component

The dynamic element of the DeSTIN formulation features two main elements. The first, more obvious term, is the $P(s'|s,a)$ term. This simply states the probability of transitioning from state s to s', with a given advice component. There are similarities here to the reinforcement learning concept of the state-action table as discussed briefly in Chapter 2, where the advice takes the place of the action, but here there is no optimization component from the learned a; rather it must be unsupervised, at least in this initial formulation. As in RL methods, the $P(s'|s,a)$ table can be learned from actual transitions. This is memory intensive, depending on the state-advice state space, but is a reasonable approach and thus is our main method for this dissertation, although we explore the use of function approximation methods here as well. We note that we have assumed some small, non-zero element for each entry in the $P^a_{ss'}$ table, so that even unlikely state transitions will not return a response of exactly 0.

The second component is the advice. In early implementations, the advice or belief of the parent node, a, was chosen using the selection rule

$$a = \arg\max_s b_p(s) \tag{3.14}$$

In other words, the parent basically fed back the label of the “winning” centroid to the children nodes. The problem with this in practice was the static learning at the parent level tended to jump repeatedly from state to state, so that there was no clear coherence from observation to observation. In addition, the advice here is more the “hard threshold” method, which tends to work well only in the face of no noise or very clearly defined classes or clusters. Therefore, we have explored instead a more temporally coherent advice generation method which looks across many different observations to determine the parental advice, which is an assessment of the general state of the children’s beliefs (and benefits from a viewpoint one layer up in the hierarchy). This new advice state is based on performing unsupervised learning, clustering, on the concatenation of children belief states across all the movements, up to the final movement. Thus the advice is generated in fairly large temporal “chunks”, albeit with an online system. These advice states are then fed back to the children during learning, and the $P^a_{ss'}$ tables (or function approximators) are built based on that advice. This method is still a “hard threshold”, but when the actual responses are delivered, the children do not actually use parental advice; instead, the cycle through all possible advice states, and deliver a belief in advice vector denoted as $B(a)$. We consider this “passive advice” as a positive step toward true incorporation of an evolving belief with confidence levels that can actually offer a level of control over the observation process. These dynamic processes are described in more detail in Chapter 5.

3.2.3 Supervised Learning

Once DeSTIN has been trained and has produced a set of responses for the given testing and training set, we proceed to create a supervised classifier and test the learned extracted features. In this work we did not seek to exhaustively search through different supervised learning methods. Instead, we did some initial tests with kNN classifier methods, due to their simplicity, and with neural network ensembles which were chosen based on their performance with initial DeSTIN outputs. There are certainly more work that could be done in this field, in particular a more intimate linking of the DeSTIN feature generation process with the supervised learning aspect would be interesting and very likely fruitful, but for the purposes of this dissertation we limited the scope to simple evaluations of existing methods.

3.3 Implementation Details

We next describe the operation of DeSTIN in more pragmatic terms, focusing on image applications. We refer to our main network experiments for clarity but of course most of the arrangements of the hierarchy can be generalized. The network is depicted in Figure 3.1 for a four-layer case. A major question with this architecture pertains to the resource allocation regarding the number of layers and the parameters of each layer's nodes (especially with regard to the number of centroids), which we explore in Chapter 4, 5, and 6. In contrast the spatial topology is straightforward, building on a small clique at the lowest layer (a 4×4 pixel), and combining above that in a 4:1 ratio or spatially a 2×2 set of nodes at the lower layer combining to one node at the layer above it.

The network depicted has an 8×8 arrangement of spatial processing nodes at the lowest layer. This lowest layer actually receives pure image pixels as inputs and each node receives a small $N \times N$ pixel section of the image. A set of discrete time steps are used and at each time step the image is moved slightly (by one pixel either in x, y, or both). At each time step the input data is clustered by the lowest layer nodes by comparing a rasterized version of the input data with its "library" of centroids. The winning centroid is chosen and a relative distance as given by Equation 4.1 is computed. The initial belief state is assumed to be a uniformly distributed vector, and so is updated at each step by the P^a_{ss} table structure given the advice from the parent node. The parent node advice, a, can be formed by many different mechanisms. The P^a_{ss} formulation was built therefore as a group of A tables, where A is the number of states in the parent nodes, using Equation 5.2. As the temporal scanning proceeds the belief values at each node are modified and evolve to produce a set of K values, where K is the number of centroids for the particular node. These belief vectors are saved at the end and input to a supervised learning system such as a neural network to provide a mapping of unsupervised belief states to supervised labels.

3.3.1 Temporal Scanning

The DeSTIN network assumes some level of motion in the field of view which can in principle be obtained through either motion of the subject (as in video sequences) or through a controlled motion of the DeSTIN imager. There is a rough analogy here to saccading, which is the rapid movement of the human eyes while analyzing a scene Deubel and Schneider (1996). Saccading allows the eye to focus the highly resolved central component of the visual system on items of interest in the field of view. The approach taken by DeSTIN here is not a controlled sequence that responds to visual feedback, but rather is a set sequence of scans through the visual field which allows the system to convert a static image into a temporal sequence within each small field of view. The DeSTIN scan motion we have chosen for our initial experiments is shown in Figure 3.2. Some of our earlier experiments Arel et al. (2009b) used different numbers of steps and different patterns. The scan pattern used here is mainly motivated by ensuring a sufficiently long temporal period is used to analyze the image and may not be the optimal scanning strategy. We assume that the scan start and stop are known to the system such that the temporal position of scanning is known at each time step and explicitly returns to the origin at the beginning of the presentation of the next image. We envision that future implementations could pursue a more rigorous strategy of exploration and learned search, motivated by the

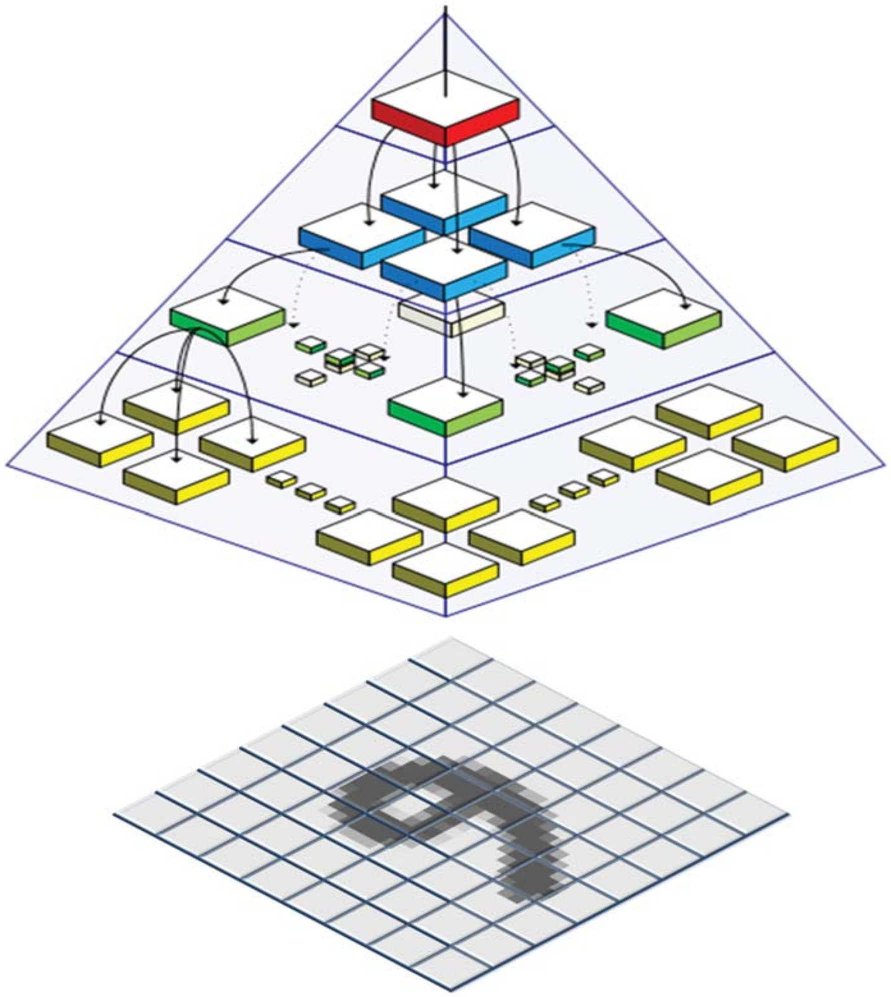


Figure 3.1: Depiction of the DeSTIN hierarchy for image studies. For a 32x32 input image, four layers are configured with 64, 16, 4 and 1 node per layer. At each node the output belief $b(s|a)$ at each temporal step is fed to a parent node. At each temporal step the parent receives input beliefs from four child nodes to generate its own belief (fed to its parent). An advice value is learned by the parent as well, which is used when formulating a set of multiple observations for an estimate of $b(s|a)$

hierarchy learning algorithm and focus-of-attention algorithms Itti and Koch (2000), but this dissertation work will focus on the simple scan of Figure 3.2.

3.3.2 Processing

DeSTIN training proceeds through the flow chart and pipeline depiction shown in Figures 3.3 through 3.7. The initial processing is simply training DeSTIN in an

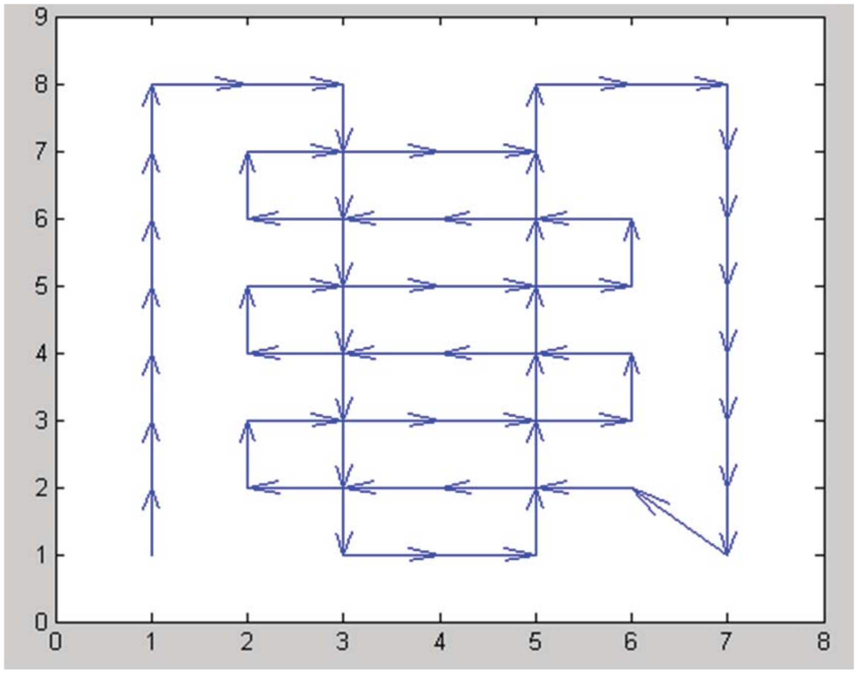


Figure 3.2: Example of scan pattern for 64 "movements" or scans. Each arrow represents the shift taken at each temporal sample.

unsupervised manner using a set of training vectors or images. The processing proceeds in a pipelined fashion, shown in Figure 3.3 for a four-layer hierarchy. Processing proceeds both left to right and right to left in the figure. At the top of the figure we see an input image which is presented to the set of nodes in layer 0 at time $t = 0$ In the figure we show that the lowest layer has 8×8 nodes. At $t = 1$, in the second row of the figure, we look at layer 1 with 4×4 nodes which is receiving the output of layer 0 which was generated at $t = 0$. This is processed and feedback (represented by the red line with the circle on the end) is passed back to layer 0, which is now receiving the second presentation of the image, which has undergone a slight shift or linear translation. This process continues in the third row, where layer 2 processes the output of layer 1 from $t = 0$ and passes feedback to layer 1...which is now processing the $t = 1$ output of layer 0, and passing feedback to layer 0, which is now processing the $t = 2$ presentation of the image. This process continues until time $t = L$, where L is the number of scans (i.e., 64 as show in Figure 3.2).

The actual processing steps for the DeSTIN training and processing are shown in a flow chart for each node in Figures 3.4 through 3.5. During initial training the node is initialized by randomly assigning its centroid locations and setting the P^a_{ss} table to a uniform distribution (meaning any state could follow another with equal likelihood). The processing then consists of making the observation, finding the cluster distances and updating the "winning centroid", then receiving the parent advice and using the

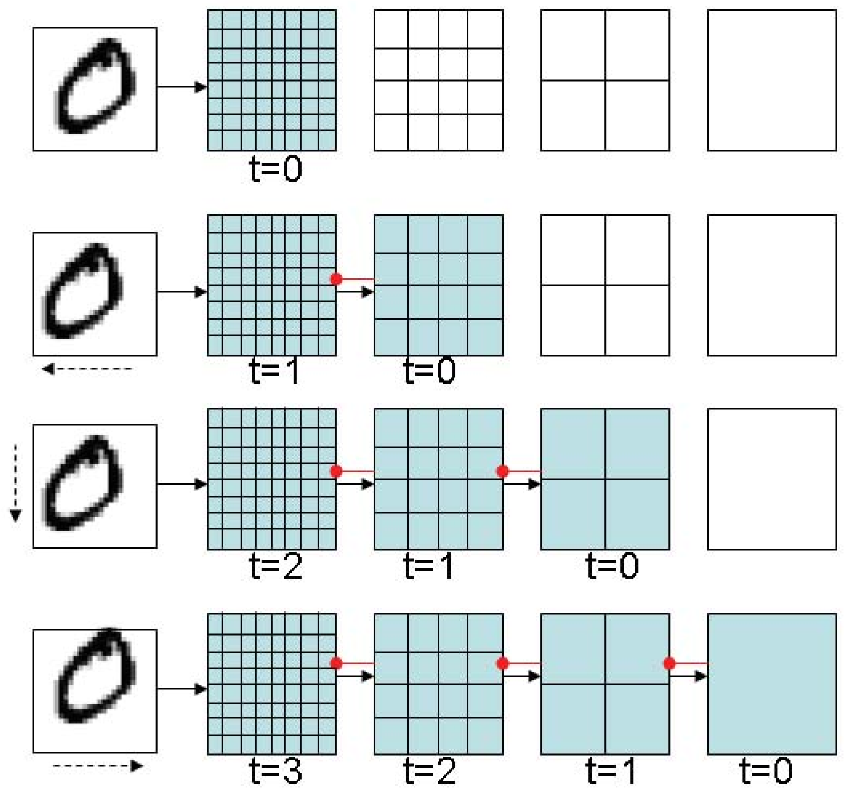


Figure 3.3: Pipeline for DeSTIN processing. At time t=0 an input image is shown and the bottom layer nodes begin processing. At t=1, the outputs of the bottom layer nodes are supplied to layer 1, which performs its processing and generates feedback (the red line) back to layer 0, which processes the shifted input image. This process repeats through the additional steps shown and beyond.

current P^a_{ss} table to compute the expected belief which is passed to the parent. The P^a_{ss} table is updated and the node waits for the next time step. When the number of time steps reaches the scan period T, the node is reset and is prepared to process a new signal. (For the most part, the reset does not alter the node itself but some debugging and diagnostic data which is included in the C++ implementation uses the reset.) Processing after training is virtually the same except the centroids and P^a_{ss} table are not updated as shown in Figure 3.5.

We also note the processing for creating a supervised learning classifier for clarity in Figures 3.6 and 3.7. This is a fairly simple process where the block "Unsupervised DeSTIN Network" which has been conditioned by a sequence of training data, is "frozen" and then its response to another set of training data is found. This training data set may, or may not, contain some of the same vectors used in the

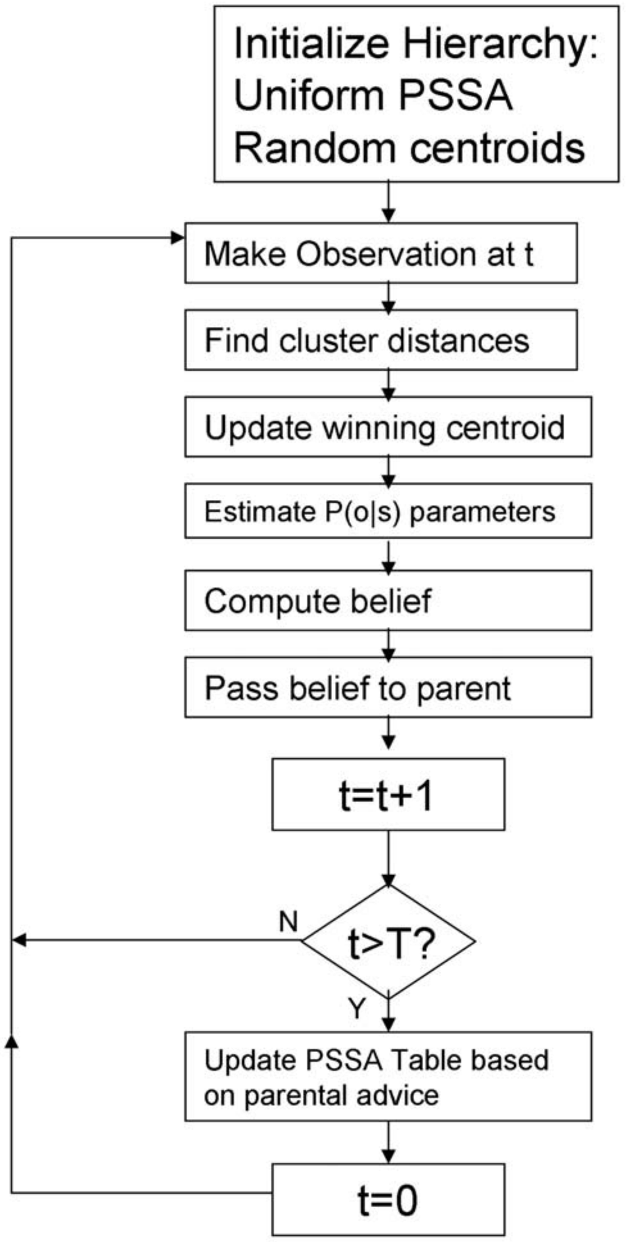

Initialize Hierarchy: Uniform PSSA Random centroids
Make Observation at t
Find cluster distances
Update winning centroid
Estimate P(o|s) parameters
Compute belief
Pass belief to parent
t=t+1
N
t>T?
Y
Update PSSA Table based on parental advice
t=0

Figure 3.4: DeSTIN processing steps for training the DeSTIN network.

unsupervised learning phase. These "Training DeSTIN Responses" are then presented to a supervised learning classifier which is trained to map labels to the DeSTIN response vectors. In the actual classification phase shown in Figure 3.7, a test vector is presented to the DeSTIN network which extracts features by finding the response of DeSTIN, and then this response is mapped to a class label by the supervised learning classifier.

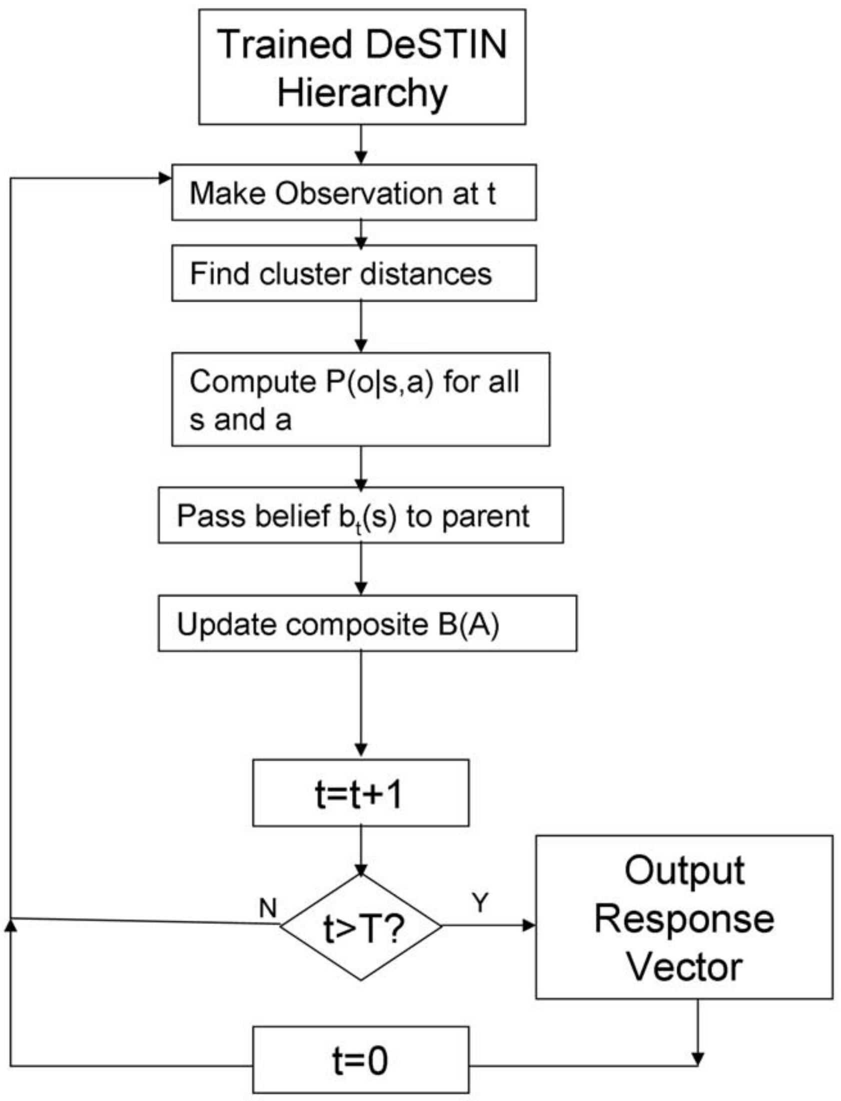


Figure 3.5: DeSTIN processing steps for generating the response of the DeSTIN network to input vectors after unsupervised training has stopped.

3.4 Comparing DeSTIN to Existing DML Schemes

3.4.1 Background

In this section we compare and contrast DeSTIN with other deep machine learning methods with respect to their fundamental functionality and implementation. We should point out that these definitions are not necessarily fixed in the sense that architectures similar to CNNs or DBNs could be developed that attempt to improve these basic approaches by creating, for example, an explicit temporal nature or spatial

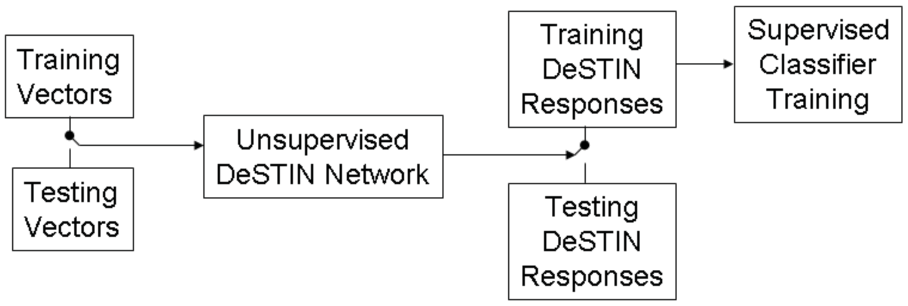

Training Vectors
Testing Vectors
Unsupervised DeSTIN Network
Training DeSTIN Responses
Testing DeSTIN Responses
Supervised Classifier Training

Figure 3.6: Training the supervised learning system with DeSTIN response vectors.

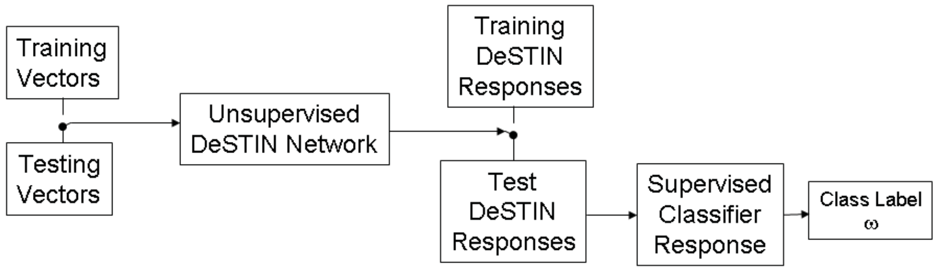

Training Vectors
Testing Vectors
Unsupervised DeSTIN Network
Training DeSTIN Responses
Test DeSTIN Responses
Supervised Classifier Response
Class Label ω

Figure 3.7: Using the supervised learning system to classify input vectors.

component. However, DeSTIN could as well be modified to use other aspects that are similar to the other DML approaches; for example DeSTIN could be modified to use an alternative data representation, but such a representation would not be in keeping with the goals of DeSTIN (an online learning system) unless it is inherently online and evolving. Finally, we discussion the relationship between multiresolution methods and DeSTIN and deep learning in general.

3.4.2 Temporal Mapping

The largest contrast between DeSTIN and conventional CNNs and DBNs pertains to the natural temporal representation of DeSTIN. The temporal component is explicitly present in both the data presentation (through scanning in our cases), the training of the network as beliefs propagate from state s to $s\prime$ which is the next temporal step, and the response which are a set of temporally evolving beliefs. CNNs, although drawing upon early work in TDNNs, are explicitly image-based (spatial only) and do not deal with time inherently. Their predecessor, TDNNs, use a sequence of delays applied to the temporal signal to generate inputs for digital filtering operations Waibel et al. (1989),Lang et al. (1990), which CNNs replace with the spatial dimension. CNNs could be adapted to extend to the temporal dimension in this manner but the cost in terms of additional network complexity would be enormous and also would require a fixed estimate on the size of the temporal window which impacts the topology specification. DeSTIN does not require any specification of the temporal size as the system dynamics functions (the P_{ss}^{a} constructs) adaptively learn the temporal scales of the data. Work in Mobahi et al. (2009) takes advantage of the concept of temporal coherence, or the assumption that adjacent video frames contain the same object, to train a CNN using a modified backpropagation algorithm which attempts to force the output of adjacent time steps to be "close" or "similar" in the internal representation. Two cost functions are used, one for regular supervised learning by gradient descent and a second unsupervised-based algorithm that strives to make adjacent video frames "close" in the internal representation. This is a clever modification but does not truly constitute temporal dependence of the architecture itself: instead, it is a method of incorporating temporal knowledge into a semi-supervised learning environment, while DeSTIN explicitly uses a single learning algorithm to include spatial information spanning adjacent or many frames. DBNs also do not deal with the temporal evolution of the observations as an inherent component of their operation. Work extending the DBN base building block, the RBM, to a "Temporal Restricted Boltzman Machine" or TRBM is described in Sutskever and Hinton (2007) and Lockett and Miikkulainen (2009). The extension involves adding directed connections from the previous states of the visible and hidden units which incorporates the temporal dependence. These designs suffer from extremely long learning times, even using contrastive divergence. In addition the temporal window must also be defined explicitly and the learning time increases exponentially with the size of the window.

3.4.3 Spatial Mapping

The spatial mapping of DeSTIN is very natural due to the structural, spatial topology of the DeSTIN hierarchy. The CNN also has a natural spatial mapping owing to the two-dimensional nature of the CNN structure, and in fact this allows the CNN to capitalize on spatial similarities and reduce the number of connections needed, making the deep nature of the architecture tractable for learning. The DBN, in contrast, is not structured as a natural spatial mapping, instead images are rasterized before presentation and the spatial nature of the vectors is not preserved in this operation. Again ,there is an important exception: the convolutional deep belief network Lee et al. (2009), established a more image-oriented architecture by replacing the basic RBM building block by a "convolutional RBM" where visible units corresponding to image pixels are fed into groups of hidden units or "detectors". The detectors each correspond to a different filter such that the response of a group is a filtered representation of the input image where the filters are generated by contrastive divergence with a CRBM-specific energy function. A sparsity constraint is added as well to prevent trivial fits to the data. An additional step in the CRBM is introduced, a "max pooling" layer which shrinks the detector outputs and is similar to the "S" units in the CNN. Stacks of CRBMs are then employed comparable to the DBN, and training again proceeds in a greedy layer-by-layer fashion.

Another comparison point is the spatial mapping of the features from the hierarchy to the supervised learning layer or system. In DeSTIN, the data available from any node is suitable for inclusion in the supervised learning phase, which means DeSTIN can draw upon both very coarse and find spatial dependencies in the data. This is important when we consider aspects such as the similarity of faces; at a high level, the similarity of faces draws more on spatial characteristics that are clear from a low-resolution aspect such as the roundness of the head, while at lower levels, the similarity depends on more subtle, fine features such as the orientation of the eyes and small details in the face morphology. This aspect can be lost through potential compression of the CNN and DBN as the data propagates through the hierarchy.

3.4.4 Learning Rules and Methodology

The learning algorithm for DeSTIN is simple and explicit from layer to layer, utilizing feedback from the higher (parent) nodes to guide the evolution of the belief vector without losing knowledge or information in the lower nodes. DeSTIN is inherently an unsupervised process for feature extraction. Pattern recognition is performed with DeSTIN by using the DeSTIN output as a set of response vectors which are mapped to a labeled output with supervised ML methods. Conversely, CNNs are explicitly supervised, and use backpropagation for training and feedforward only for classification and formulation of analogous "belief" states. The learning algorithm for DBNs rely on feed forward only and layer-by-layer, greedy unsupervised training. They do have an explicit supervised training aspect for the top layers, after the unsupervised pre-training has been completed.

3.4.5 Comparison with Multiresolution Methods

The hierarchical structure of DeSTIN and some deep learning systems is very reminiscent of multiresolution or other pyramid-like approaches, such as Laplacian or Gaussian pyramids Adelson et al. (1984), Bister et al. (1990), Burt et al. (1981). In pyramids processing proceeds by using a kernel operation and resizing the image, which is equivalent to processing the image with a larger kernel. These kinds of operations emphasize different features of the image or signal in question by changing the spatial content of the image to match that of the kernel (or filter). There are numerous examples of these types of operations; the SIFT features and their ilk are an example Lowe (2004), Bay et al. (2008), as are Gabor filters Gabor (1972) and wavelet processes Strang and Nguyen (1996), with the latter usually imposing orthogonality from scale to scale. (An interesting point for brain emulation is the concept that cortical receptive fields can be modeled as Gabor filtering processes Jones and Palmer (1987), Jain and Farrokhnia (1991)). Other pyramid-like approaches include the pyramid Lucas-Kanade algorithm for point tracking Lucas and Kanade

(1998), which allows the efficient detection or matching of feature points in motion images regardless of the amount of motion (number of pixels moved) from frame to frame.

However, these types of systems or methods are not deep learning, as they do not actually use a hierarchy to learn different levels of details about a signal; instead, they are using the hierarchy to efficiently repeat different detection or decomposition operations rather than performing learning operations such as clustering or recognition on a changing scale of data. Such operations could have a role in deep learning, of course. In the case of DeSTIN, we argue that while the approach is indeed pyramidal in structure, the processing of DeSTIN is not multiresolution in the sense that the same operation is not applied across the board to the entire image; instead, small pieces of the images are processed by the lowest level, and these are then reprocessed or reinterpreted as larger components at the next level of the hierarchy. A DBN has less resemblance to a multiresolution system, since the input at the lowest level is a complete image and the processing proceeds on the entire image at each level without accompanying spatial processing. Finally, for comparison, a CNN on the other hand has an explicit multiresolution component, as the signals processed by the learned filters at each level are followed by a downsampling operation and the next round of processing is thus performed on a lower resolution "image-like" signal. However, the CNN does not impose any kind of constraint from layer to layer regarding the kernels; they can be completely independent and different from layer to layer.

Chapter 4

Learning of Static Components

In this chapter we explore the static learning components of 3.13, which we refer to as the $P(o|s)$ element. We discuss two primary topics: state identification or signal generalization and probability estimation. The formulation of the different states for $P(o|s)$ is achieved through online clustering. We discuss different criteria for measuring the movement of centroids and learning rates which we have developed and explored in this work. We then discuss the evolving cluster method (ECM) Kasabov (2003), which we implemented in an effort to automatically set the number of clusters used.

The second topic is the probability estimation. In our early DeSTIN work we approximated this as the distance between the sample and the cluster centroids given by

$$P(o|s) = 1 - \frac{d_s^{-1}}{\sum\limits_{s'' \in S} d_{s''}^{-1}} \tag{4.1}$$

This expression takes the distance of centroid s to the input vector and normalizes by the sum of the distances to all centroids so when d_s is small (i.e., 0) there is high "belief" that this centroid is the correct one. The relationship is clearly motivated as we would like the vector value to be normalized and each component should represent similarity to each state or centroid. Indeed this expression can be seen in other areas such as fuzzy clustering Duda et al. (2001) but this representation is simplistic and does not consider the variance in the data associated with each centroid. In addition, it very explicitly ties the distance from all centroids together through the normalization factor, which may not be valid. We thus discuss different probabilistic models for $P(o|s)$, including Gaussian, Rayleigh and exponential models which use the distance to the centroids in a probabilistic fashion, borrowing from the concepts of nonlinear dimensionality reduction Kruskal (1964), Tenenbaum et al. (2000) and nearest neighbor distance distributions Ding (1999), Korn et al. (2001).

We conclude with a comparison of batch and online clustering, using the ECM and an online clustering winner-take-all methodology for a variety of test cases.

4.1 Online Clustering

4.1.1 Overview

One goal of DeSTIN is to produce a system that scales efficiently with simple hardware, so we have imposed a constraint that the system cannot iterate through the entire training data set in memory. Consequently, an online clustering algorithm is required. Our fundamental approach is based on the winner-take-all competitive learning and each node of DeSTIN contains a processing engine to perform this function. We have studied a variety of different methods for improving performance and monitoring progress Young et al. (2010), with a goal towards determining a good stopping criteria. In our hierarchy, we adopted a strategy of monitoring the progress of the centroid formation and stopping when some criteria was met, as will be described later. The same strategy applies to the online ECM method as well, which is still a WTA method at its heart, but with extra constraints to adapt the number of clusters.

A summary description of the core algorithm with fixed number of centroids is as follows. The estimated centroids are initialized to random values. A new observation is then assigned to a single estimated centroid based on the minimum distance computed by some similarity value such as Euclidean distance. The WTA centroid selection simply performs $\arg\min_x d_x$ to select the winning centroid for updates, however the algorithm always chooses the labeling centroid as the one with closest distance (without regard to starvation trace). Then the update rule for the winning centroid x is achieved by

$$x^{t+1} = x - \alpha \|x - o\| \tag{4.2}$$

where α represents the learning rate and o is the observation or input vector. The learning rate may be constant or may decrease over time. The clustering changes over time are monitored by a set of metrics of interest derived from the mean and standard deviation of the changes made in the centroids. When some criteria is met, the clustering terminates.

4.1.2 Learning rate

The learning rate in online clustering has been implemented in a variety of ways in the literature, with constant learning rates and decaying rates commonly used Bottou (1995). In competitive learning clustering algorithms the learning rate is often adjusted to allow trade offs between faster learning in early phases of iterations and stability in later phases. Typically the learning rate is adjusted so that it is monotonically decreasing, for example a decaying exponential with the decay as a function of iteration. One methodology for a decaying learning rate which is analogous to k-means clustering is accumulating a count N_s of the samples assigned to a particular centroid and setting the learning rate to its inverse, $\epsilon = \frac{1}{N_s}$. This is an intuitively appealing method, and ensures convergence since the rate decays with each new data point selected for a particular cluster Bottou (1995).

The multi-layer DeSTIN hierarchy has some special features with respect to online learning rates. During the early learning in DeSTIN, the output of each layer is evolving and changing, and therefore the next layer up receives a non-stationary input until its child nodes are stabilized. In this environment, the learning rate could be adjusted based on the data as measured by a few performance and behavioral characteristics Karnowski et al. (2010a). For the DeSTIN hierarchy, during network initialization the learning rate was adaptively modified on a layer by layer basis but learning proceeds in parallel for all layers until a layer is terminated. We monitor the nodes of layer 0, and initialize the learning rate to α_F for layer 0 and α_S for subsequent layers. The mean across all centroids for ρ was computed at each observation. When the mean value was less than a threshold T_ρ, clustering terminates for the node. When half the nodes were terminated the entire layer clustering was stopped. The learning rate for the next layer was then reduced to α_F and the process repeated until the top layer clustering was terminated. This functioned well and we show some results in Chapter 6. For an evolving system with a changing number of centroids, however, a different strategy must be used to enable simultaneous learning. We discuss this in more detail in Section 4.2.3.

4.1.3 Starvation Trace

A problem with clustering in general is that the dynamics of the clustering change depending on the initial centroid selection. This problem has several practical implications, one of which being that in some cases centroids may be underutilized if they are not near the original data itself. Thus we introduce the concept of starvation trace, ψ_x, which is used to include clusters which are initially too far from observations to update. The starvation trace allows idle or starved centroids to accumulate credit over time when they are not the selected centroid (and lose credit when they are the selected centroid). In our case, the starvation trace is initialized to a constant vector of length D where D is the dimensionality of the observation. For each observation where centroid χ is chosen the starvation trace for all non-selected centroids is reduced by some value. The starvation trace is employed to weight the distance calculation and thus render "starved" clusters a chance to make movement towards data samples. A summary description of the application of ψ_x is as follows. The estimated centroids are initialized to random values. A new observation o is then assigned to a single estimated centroid χ based on the minimum distance computed by some similarity value such as Euclidean distance. This distance metric is weighted by the starvation trace as:

$$d_x = dist(x, o) = \|x - o\|\{1 - \psi_x\} \tag{4.3}$$

where ψ_x is the starvation trace. Thus as the starvation trace increases, the distance metric appears to decrease and gives the "starved centroid" an opportunity for updates. The WTA centroid selection simply performs $\arg\min_x d_x$to select the winning centroid for updates, however the algorithm always chooses the labeling centroid as the one with closest distance (without regard to starvation trace).

An example of how the starvation trace works is shown in Figures 4.1 through 4.2. In this illustration there are three actual centroids, all with uniform probability of occurence, and we attempt to fit three cluster centers to the data. For the purposes of illustration, all estimated cluster centers are set to initial values of (0.5,0.5,0.5). The initial configuration is shown at the top of Figure 4.1 where we see the three actual data centroids in red, and the estimated centroids are the blue circle, green triangle, and cyan star. After several samples we see that the estimated centroids have shifted where the green triangle has successfully selected the upper distribution, the blue circle is drawn between the lower distributions, but the cyan star is starved. Using the starvation trace presents a different picture after several iterations (Figure 4.2) where the starvation trace credit for the cyan star has reached the point that the lower right distribution attracts an estimated centroid. After additional iterations all centroids are representing a different data cluster.

4.1.4 Monitoring centroid changes

We may easily accumulate estimates of the standard deviation and mean of the centroid changes for each estimated centroid. These are computed on-line and are given by the following formulas where d_x is the distance between an observation and winning centroid χ with mean μ_x:

$$\mu_\chi^{t+1} = a\mu_\chi^t + (1-a)d_x \tag{4.4}$$

$$\sigma_\chi^{t+1} = b\sigma_\chi^t + (1-b)\|\mu_\chi^t - d_\chi\| \tag{4.5}$$

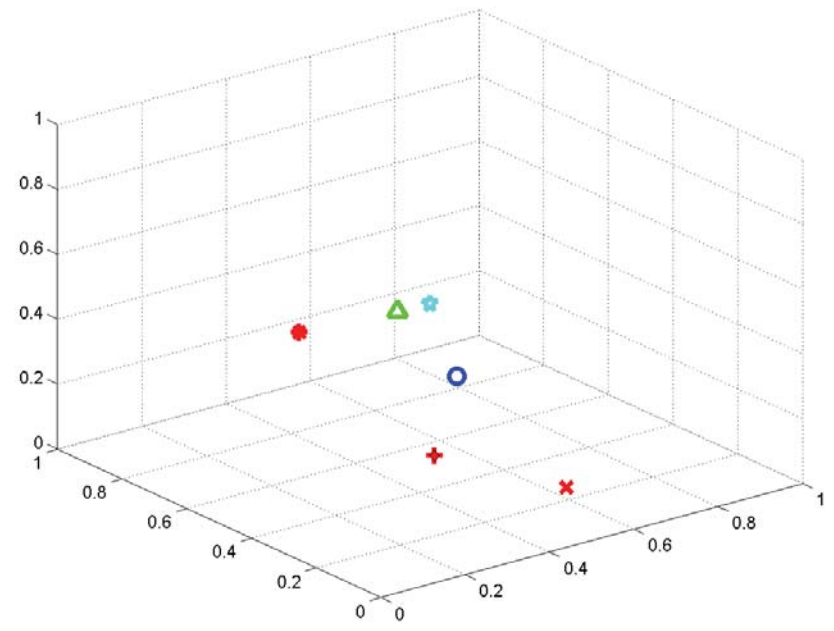
where a and b are constants with $a < 1$ and $b < 1$. Both vectors are initialized to all 1. These estimates are smoothed metrics for the changes made to the centroids over time. Ideally, as the winning centroid comes to represent the actual centroid the value of the mean change estimate μ_χ should approach 0 (since d_χ should become smaller and smaller) and the value of the change standard deviation estimate σ_x should also approach 0 (because both d_x and μ should become smaller and smaller).

To effectively utilize these metrics, we define a functional relationship between μ and σ as

$$\hat{\rho}(\mu_\chi, \sigma_\chi) = \frac{2}{1 + e^{-\gamma \frac{\sigma_\chi}{1+\mu_\chi}}} - 1 \tag{4.6}$$

In this function, we see that when the centroids are not changing much μ and σ should be small and consequently $\hat{\rho}$ should be 0. Periods of large change in centroids will force $\hat{\rho}$ to be nearly unity. As a final note we smooth our estimate of ρ using the following rule with ρ initialized to unity:

$$\rho_\chi^{t+1} = c\rho_\chi^t + (1-c)\hat{\rho} \tag{4.7}$$



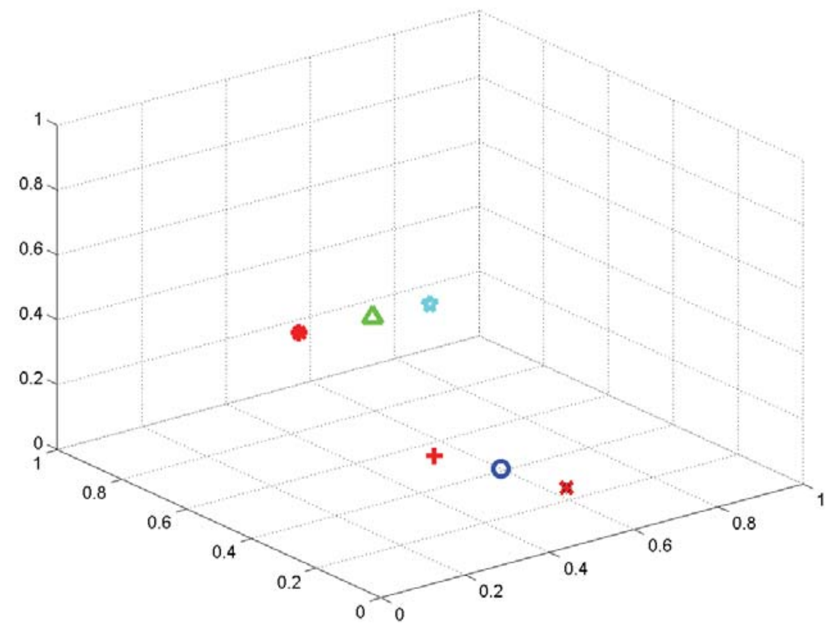
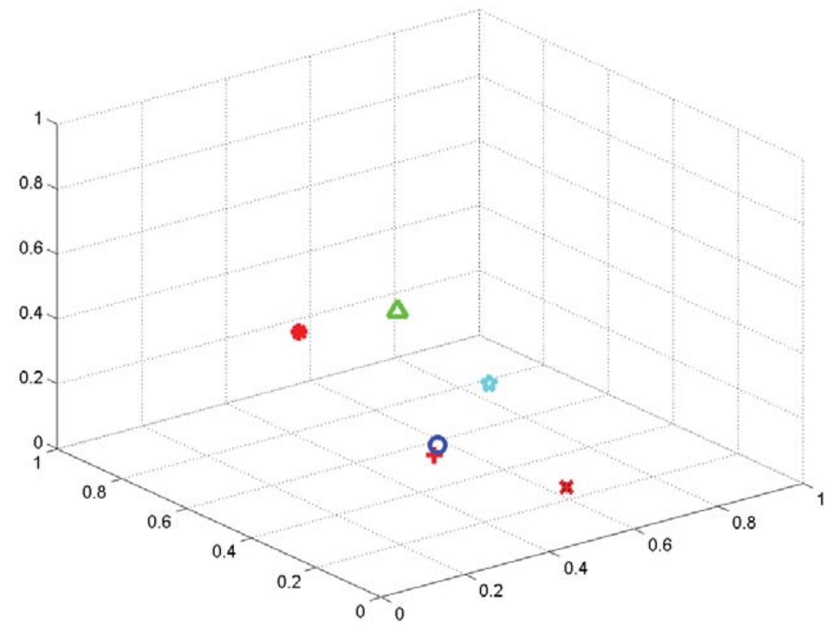


Figure 4.1: Top: No starvation trace, initial iterations. Bottom: the centroid represented by the blue circle is "trapped" between the two actual centroids while the centroid represented by the cyan star is "starved".



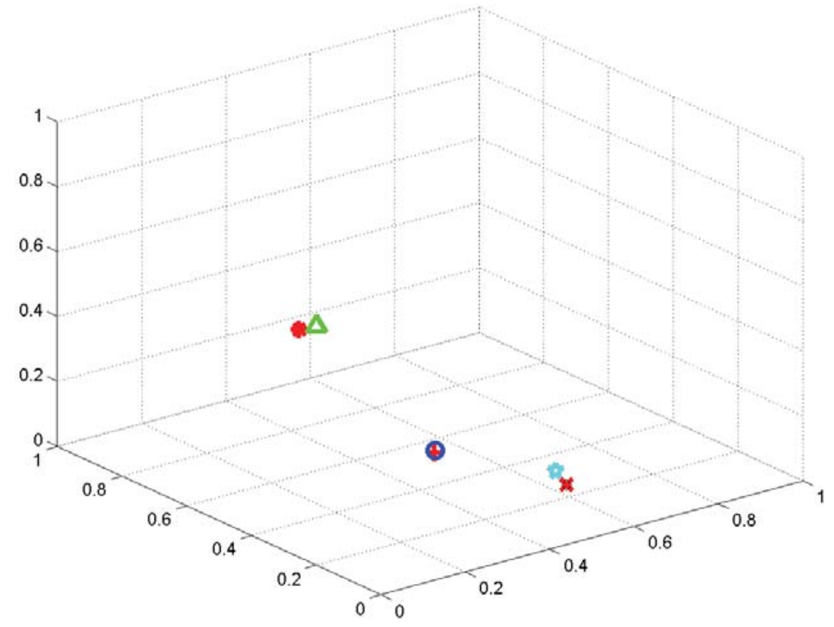


Figure 4.2: Top: With the starvation trace, eventually the green triangle gets "credit" for not being selected and is drawn toward the actual centroids even when it is not the winner. Bottom: After more iterations with the starvation trace all centroids are used.

4.1.5 Momentum

The momentum concept uses the idea that several updates in the same direction indicate future updates will be in the identical direction. The momentum can be used to thus modify the step size or add an extra update in the case of a constant learning rate. If we define the prior centroid update for centroid χ to be $\triangle_\chi$ then the new step size δ_χ is computed as $\delta_\chi = e^{\alpha(\lambda-1)}$ where α is a constant and

$$\lambda = 0.5(1 - \frac{(\chi - o)\triangle_\chi}{\|(\chi - o)\|\|\triangle_\chi\|}) \tag{4.8}$$

In this computation, the scalar value λ is a measure of the cosine of the angle between the prior update center $\triangle_\chi$ and the new direction of the update $(\chi - o)$. This adjustment is intended to weaken the step size when the direction of the update is not close to the vector direction as the prior update, and strengthen it when the vector direction is the same. The idea is that when the direction is similar the estimated centroid is moving consistently to the same general direction, and once it gets close to the true cluster centroid the direction is noisier and thus the step should be scaled back.

As a simple illustration of this concept, consider a clustering algorithm where the step size is the cosine direction scaled by a fixed constant 0.1. The direction is still set based on the difference in the observation and centroid as before. Thus the update rule is given as

$$\chi^{t+1} = \chi - \epsilon\frac{(\chi - o)}{\|\chi - o\|}\frac{(\chi - o)\triangle_\chi}{\|(\chi - o)\|\|\triangle_\chi\|} \tag{4.9}$$

Take a simple case of a single initial estimated centroid at (1,1) and a true centroid at (0.5,0.5). Observations are slightly noisy. The first few iterations of the algorithm produces results such as shown in the top two plots of Figure 4.3, where the red star is the previous estimate, the blue star is the current estimate, and the green star is the next estimate. Note that the step size between is nearly constant. By iteration 14 (bottom of Figure 4.3), we see that the step size is definitely changing in response to the cosine of the angle as described.

We may perform various measures to provide the robustness of this momentum measure against noise. One method is to smooth the step size over time by the relationship

$$\delta_\chi^{t+1} = k\delta_\chi^t + (1-k)\delta_\chi \tag{4.10}$$

Another method to achieve robustness with the momentum concept is to simply count the updates that are in the same consecutive direction and apply them for a single update Young (2011).

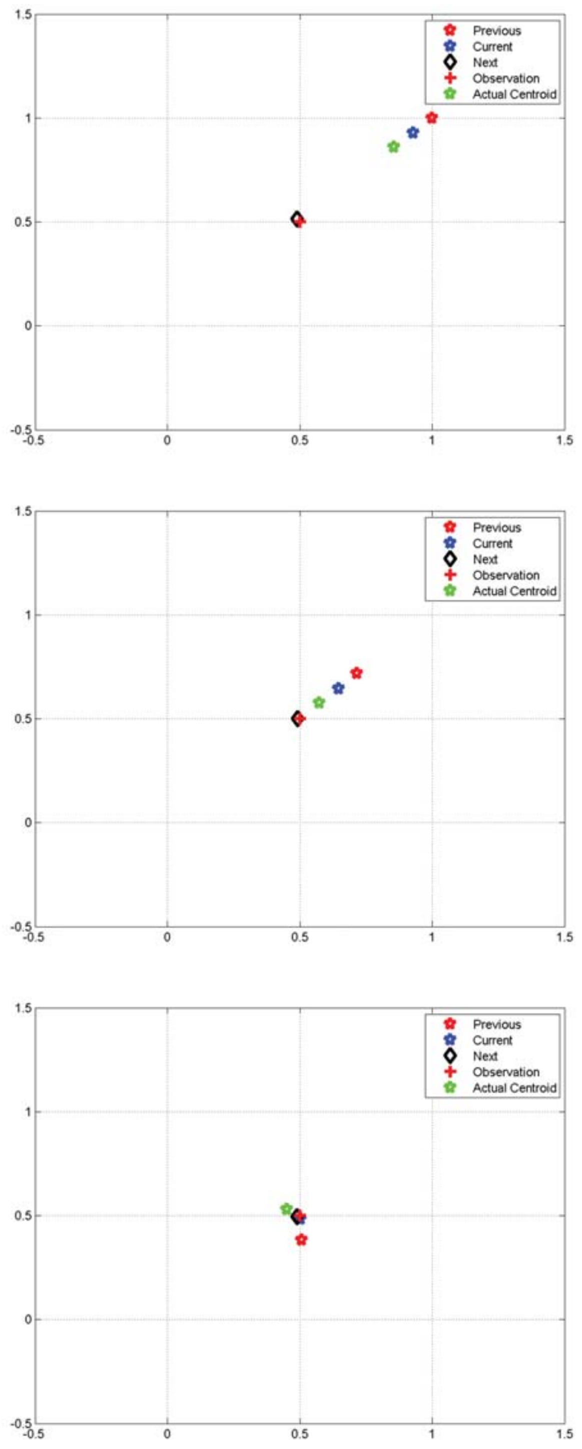


Figure 4.3: Effect of momentum calculation on step size. In all plots the red is the previous estimate; the blue is the current estimate; the green is the next estimate. Top: Iterations 0,1,2. Middle: iterations 3,4,5. Bottom: Iterations 15,16,17.

4.1.6 Pseudo-Entropy Measure

We examine a metric we have named pseudo-entropy or $\Re$ as it is a measurement derived from information entropy Cover and Thomas (2006). A very similar metric has been used in the past in batch clustering Chinrungrueng (1995). The formulation for the $\Re$ of a vector v of length D is given as

$$\Re(v) = 1 - \frac{\sum v_i \log(v_i)}{\log(\frac{1}{D})} \tag{4.11}$$

where $\sum v_i = 1$. From the definition, we see this is clearly related to the entropy of v (assuming it is normalized), but scaled by the maximum entropy possible for a vector of size D. The scaling and subtraction from unity means the $\Re$ is 0 when the elements of v are all equal (a uniform distribution) and is maximum at unity when all elements but one are 0. The value of $\Re$ serves to estimate how well the overall available centroids are adapting to the input data and helps dampen or freeze the clustering but allows the algorithm to remain responsive to changes in the input data stream.

As an illustration of this, see Figure 4.4 where we show a 16-dimensional vector where each element is the count of the times a particular element is chosen. Initially the vector is all 0s with a single 1 when the first element is chosen, which represents maximum $\Re$. As elements are chosen the vector begins to resemble a uniform distribution and the $\Re$ drops with some noise until sample 101, at which time the data probability distribution changes to always select element 1. Thus the $\Re$ begins to rise again over time. As used in Chinrungrueng (1995), the $\Re$ adapts the learning rate based on the within-cluster variation which is computed as the mean distance between the cluster centroid and the data vectors assigned to that cluster.

The $\Re$ measurement can be used on a variety of input vector data. In our formulations we have experimented with its effectiveness as a measurement of the number of vectors assigned to each centroid, which requires a small buffer and a running sum, possibly with some level of "forgetting" by weighting the current value more than past values. Another metric which can be evaluated is the ρ measurement, which would essentially measure how the centroids are deviating with a goal of having all reach a comparable amount (ideally a very small fluctuation).

4.2 Evolving Clustering Method of Kasabov

While the different mechanisms described above can yield learning improvements, a fundamental question about clustering is the choice of K. Clustering algorithms seek to solve the optimization problem of finding the minimum error between a collection of K cluster centers and the data. There has been research into selecting the optimal number of clusters, and generally this involves imposing additional constraints. One example of such an algorithm is the evolving clustering method

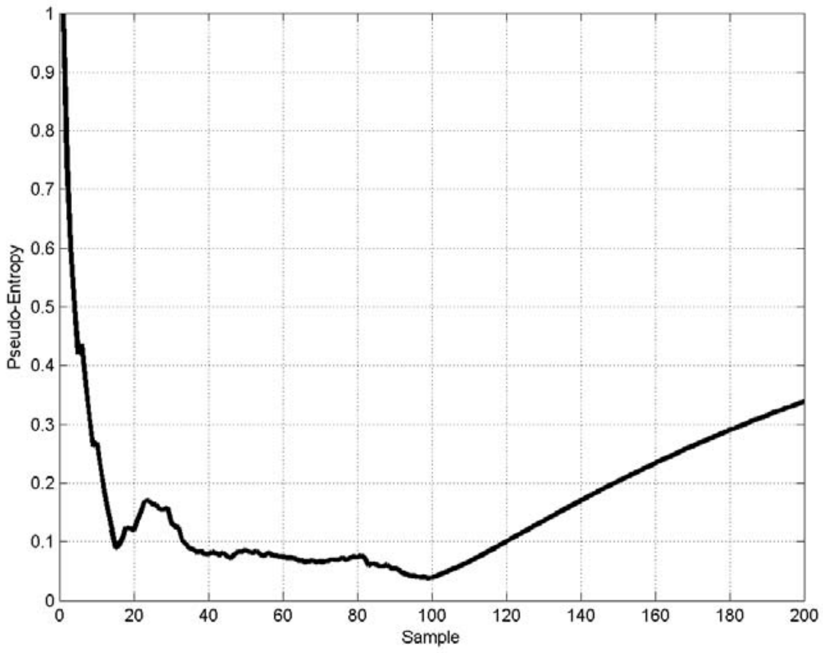


Figure 4.4: Example of $\Re$ over time. At sample 101 the pdf of input vectors changes from a uniform distribution to always choosing a single vector.

(ECM) of Kasabov from Kasabov (2003) and Kasabov and Song (2002), which is inherently online although offline or batch versions have been created. This algorithm can be summarized as follows:

- The first sample of data is used to create the first cluster C_1. An associated cluster radius R_1 is created, with an initial value of 0.
- For each successive sample:
  - Compute the distance of the new sample to each cluster and assign the new sample to the cluster of minimum distance.
  - If this distance is less than the associated cluster radius, then the sample is labeled by that cluster and no updates are performed.
  - Otherwise, for each cluster compute $S_j = D_j + R_j$.
  - Choose the cluster with minimum S and if $S < D_{thresh}$, adjust the selected centroid and increase the radius for that centroid to $S/2$.
  - If the distance is greater than twice a distance threshold D_{thresh}, then a new cluster centroid is added, with a value of the new data point.
- The algorithm terminates when there is no remaining data.

The algorithm ensures that the maximum distance from a cluster centroid to its samples is not greater than D_{thresh}. This is a very simple but elegant algorithm. However, there are some issues with it in the DeSTIN context. The main problem is that when the data is non-stationary, the algorithm will create cluster centroids that are meaningful initially in the data stream, but are wasted later as these values are no longer needed. This is a problem with DeSTIN since the lower layers stabilize first, and until they do the upper layers must work with data whose statistics will likely change. Other related issues include practical issues such as we want to limit the maximum number of clusters obtained since we assume practical hardware limits. Therefore, in our work, we modified the approach slightly by introducing a few additional constraints and implementation details: a monotonicity measurement, limiting the number of clusters, and adaptations to nonstationary data.

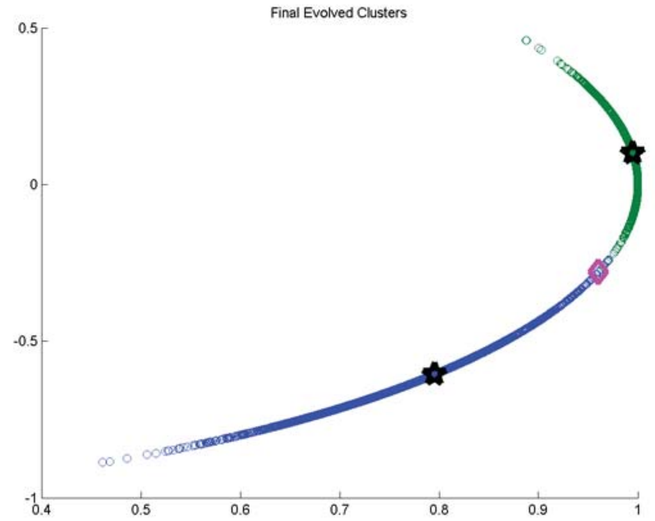
4.2.1 Monotonicity measurement and adaptive thresholds

The monotonicity measurement is intended to avoid cases where two distinct data clusters cannot be properly split due to the hard distance threshold. In this adaptation to the ECM algorithm, each centroid accumulates a measure of the distance of the vectors assigned to it. The monotonicity measurement simply consists of accumulating, in pre-set bins, a histogram of the distance between the selected centroid and the data samples. When a cluster has good cohesion, this histogram should show a monotonic distribution as a simple approximation to the probability distribution function of the data members. Periodically in the processing the monotonicity is evaluated by normalizing the histogram, then computing the sum of differences as

$$\mathcal{M}(h) = \sum_i (h_i - h_{i-1}) \tag{4.12}$$

When this value is above some threshold, the distance threshold D_{thresh} is reduced by some scalar multiple. In principle this distance will change as the centroid changes, but in practice this is not much of an issue since the monotonicity metric induces splitting up to a point; once the clusters have split sufficiently, the centroid will begin to change less and the distance metric will stabilize. It is true that the accumulation of the distance metric does require some measure of memory to the system, but the monotonicity measure is based on a simple counting and binning scheme so this is a minimal detraction.

A case for illustration is shown in Figures 4.5. Here the original centroids are very close together but are very distinct. As the algorithm progresses, the centroid eventually settles in the center of the two centroids, but the distribution of the data for the centroid is not monotonic; rather it has a more uniform value as seen by intuition. When the monotonicity metric is used, and set to a modest value (-0.5), the distance threshold can be reduced until the centroid splits off a new entry. Subsequently, the original centroid chosen will then migrate to the true location.



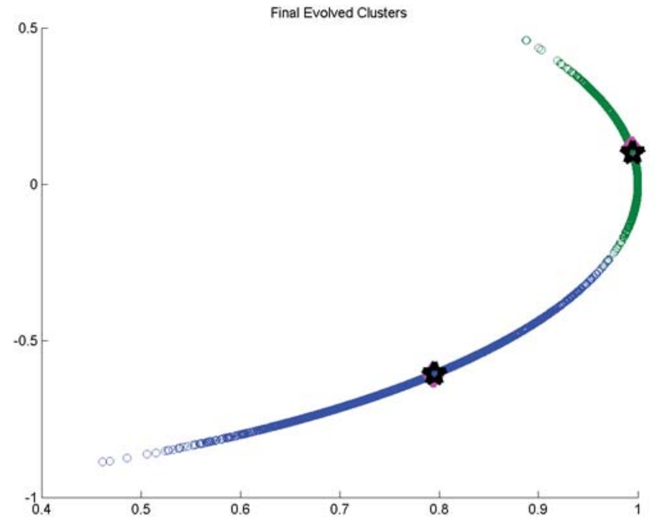


Figure 4.5: Top: Example data transformed to cosine space, where data is normalized to unit vectors and the angle is used to distinguish the vector position. Without the monotonicity criteria the single centroid meets the distance criteria of ECM. Bottom: With monotonicity introduced, we can refine the region and introduce an additional centroid, getting a better representation of the data.

4.2.2 Limited number of clusters

In a real environment, even in software but especially in hardware, it is realistic to impose a limit on the maximum number of cluster centers that can be formed. In our case we imposed this limit for practical implementation aspects. This is a trade-off between the implementation details (fitting each DeSTIN node in a given platform space) and the performance of the selected centroid group in terms of minimum mean-square error. We note that if the data itself does not have cohesive clusters, the ECM algorithm as implemented will likely extend to the maximum number of centroids available. This is illustrated in Figure 4.6 where we show the behavior with an input of uniformly distributed data. The accumulated cluster data histograms never exhibits a good monotonicity property, and consequently the algorithm keeps splitting thresholds until eventually a uniformly distributed set of centroids is achieved. In effect, this is the optimal solution given the maximum number of allowed centroids, so it is not necessarily a problem with the algorithm.

4.2.3 Non-stationary observations at higher layers

Generally, the ECM algorithm assumes the data is nonstationary and simply evolves more centroids as needed. In DeSTIN as the network learns the inputs to the upper layers are indeed nonstationary initially, but over time as the lower layers stabilize the data becomes stationary again. We have already established that DeSTIN need not learn layer-by-layer Karnowski et al. (2010a), but the adaptive nature of the ECM algorithm can cause changes in the observations output from the child nodes over time, until the clustering has stabilized or been halted. For our implementation, we perform this adaptation in the higher layers by starting with the maximum number of clusters allowed by our platform. Each centroid is then adjusted using the winner-take-all method, including the adaptations and metrics discussed previously. This essentially bootstraps the centroids and gives a good approximation to the correct locations. Then, once the lower layer has stabilized sufficiently, the ECM algorithm takes over. The pool of existing centroids is queried and if the winning centroid is within $2D_{thresh}$ it is used and adjusted as in ECM, and is "claimed" as an actively used centroid. Any data points that are outside this threshold, which ECM dictates would create a new centroid, instead capture any currently available centroids in the "unused" pool. When learning is deemed completed, any unused centroids in the pool are deleted. In an implementation with fixed dimension sizes, these values are simply set to 0. This is important because in DeSTIN, the lower layers are initially moving about and therefore ECM may waste resources unnecessarily.

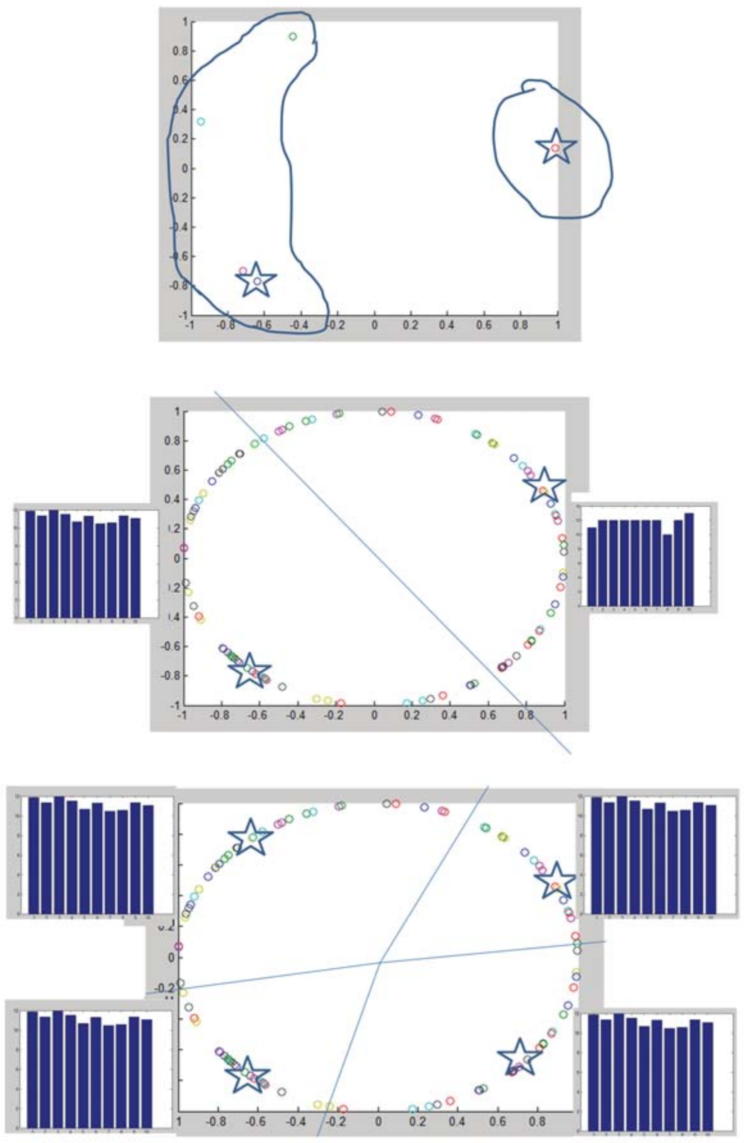


Figure 4.6: ECM with uniformly distributed data on unit circle to illustrate reasoning behind limiting number of clusters. Top: The first few samples create two centroids. Middle: over time, two centroids have evolved, but the distances of members are somewhat uniform, causing additional centroids to be evolved. Bottom: More evolution produces more centroids but each still has a uniform histogram. Over time more centroids will evolve in this situation.

To illustrate this, we performed an experiment which is summarized here. In this experiment four normally distributed processed were used in 2D to create four different classes of data. For each true centroid, we randomly initialized a starting estimated centroid value, then with each iteration "moved" each current estimated centroid toward its final value to simulate a lower layer of DeSTIN adapting centroid values. At each iteration one of the centroids was chosen and then we generated a normally distributed random value with that current centroid value as the mean. The centroids were moved toward the "true" value with an update rate of 0.0001 (using the below Matlab code, with CentroidMovementRate set to 0.0001).

```
Directions = ActualCentroids-CurrentCentroids;
Step = CentroidMovementRate*abs(sum(Directions'));
Step = Step';
Step = repmat(Step,1,size(Directions,2));
CurrentCentroids = CurrentCentroids+Step.*Directions;
CurrentCentroidsNormalized = CurrentCentroids;
```

The experiment ran for 20000 iterations. We illustrate the movements over time with a few "snapshots" of the process in Figure 4.7. The true centroids are shown as diamonds overlaid with crosses, with the moving centroids shown as circles. The evolving "winning" centroids over the current time step and the previous 1000 iterations are shown as the small colored circles. At iteration 3500, we see there are actually the correct number of centroids in play, but by iteration 8500 a new set has arrived in the upper region of the plot. By the final iteration, the noise level of the signals is such that we still have "winning" centroids but they are not well aligned with the true centroids. The main point of this exercise is to illustrate the potential resource waste of the moving centroids, which is problematic for the upper layers of DeSTIN. The proposed solution, of allocating the maximum centroids and letting them track changes as needed, using our online non-evolving WTA method, then transitioning to an evolving system using the current centroids as the pool of values makes intuitive sense here. As further illustration, we show the change in the number of centroids with iteration in Figure 4.8, showing that the ECM method over-estimates the number of centroids significantly due to the nonstationary inputs.

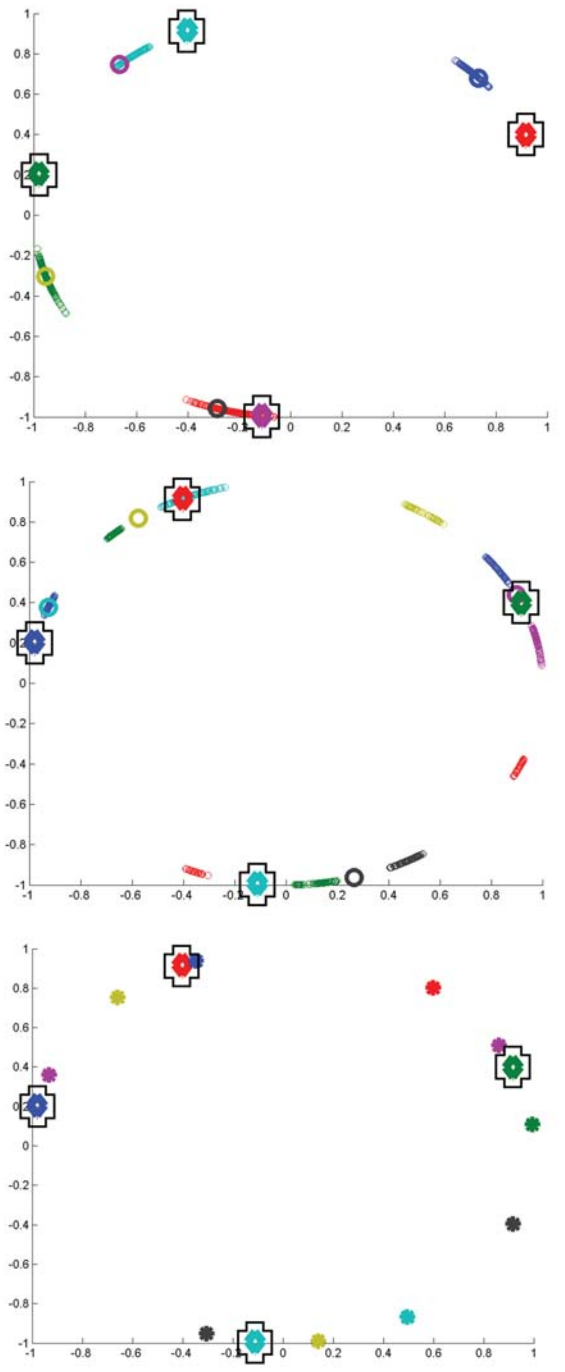


Figure 4.7: Evolution of the centroids over time. The true centroids are shown as diamonds overlaid with crosses. The different centroid values over the past 1000 iterations are shown here. Top: Iteration 2501-3500 There are fewer centroids shown than the final result because we only show those active in the last 1000 iterations. Middle: Iterations 7501-8500, showing ten different labels applied. The moving centroids are close to their final position. Bottom: Final centroids at the end of the data run.

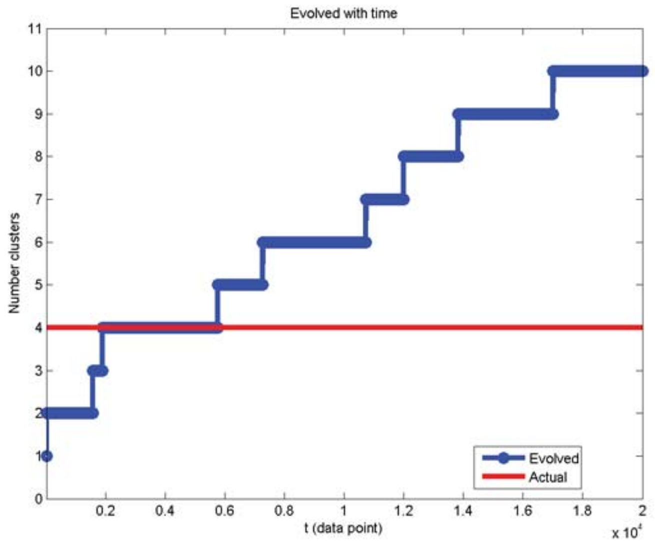


Figure 4.8: The number of centroids used in ECM as a function of time, or data sample shown to the system. We see that the system continues to create new clusters, well above the true number, because ECM algorithm cannot effectively "forget" old clusters which are no longer needed.

4.2.4 Stopping criteria

Ideally the ECM, like the online WTA algorithm, will stop "on its own", in that centroids will stop getting updated due to the data. However this is not guaranteed and highly unlikely except in contrived situations. Instead, we propose two complimentary methods: first, the monotonicity criteria, which in addition to monitoring the distance threshold can also be extended to produce "well behaved" clusterings by canceling clustering when the monotonicity is below some threshold T_M. Another useful criteria is the ρ metric, which again monitors the movement of the centroids over time. A final concept is the use of the k-means-like learning rate, $\frac{1}{N}$, which does not actually stop clustering but eventually will drive the ρ metric to stability.

4.3 Probability Models

Our static modeling using the centroids to represent data states is well motivated, but the true objective given these states is performing an estimate of $P(o|s)$. There are a variety of ways to perform this estimate. One popular method is the expectation-maximization method (EM) Dempster et al. (1977), Majdi-Nasab et al. (2006), which is often seeded or started by performing a k-means clustering very analagous to our method here. However, applying EM using a Gaussian Mixture Model in high-dimensional space often requires some level of manipulation or assumptions, as often the covariance matrix can become singular. As an alternative, other authors have used probabilistic measurements based on the similarity measurement alone, which effectively converts a high-dimensional signal into one-dimensional distance or similarity measurement for the modeling of the stochastic nature of a signal. By drawing upon the distribution of the distance value itself, we can formulate a meaningful estimate of the probability $P(o|s')$. This borrows from the basic concept of most nonlinear feature reduction methods such as Multi-Dimensional Scaling Kruskal (1964) and Isomap Tenenbaum et al. (2000) where distance measurements between members of a dataset are employed to create a meaningful projection space through various methods such as singular value decomposition. In addition, work in Ding (1999) and Korn et al. (2001) have explored mechanisms for this and rely on the actual distance similarity metric, reducing the problem to a simple one-dimensional space. Our approach here does not involve such decompositions, but rather seeks to simply characterize each centroid by the mean and variance of the vectors that are attracted to it in the training phase. This suits our goals of low computational resources as well.

In this section, we study the probability distribution of the distance data when we use a cosine similarity metric by transformation of variables, assuming the data is Gaussian distributed in the initial space. We then fit an exponential, Rayleigh and Gaussian model to the data and show a comparison for these models. Note that clustering in such a cosine similarity space is also known as "spherical" clustering, since the data is constrained to lie on a hypersphere Banerjee and Ghosh (2004).

4.3.1 Probability Model in Cosine Similarity

Since our work focuses largely on the cosine distance, we derive the probability distribution of the cosine distance based on a two-dimensional jointly Gaussian distribution functions with independence between the two dimensions. In this case, an observation consists of a normally-distributed random variable point (x_o, y_o), with the angle Θ from the centroid (mean) uniformly distributed and the distance ρ from the centroid Rayleigh distributed. We can assume the centroid is located at (x_c, y_c), has already been normalized, and we seek to find the probability distribution of the angle ϕ as shown in Figure 4.9. Thus we can write the observation as

$$(x_o, y_o) = (x_c + \rho \cos\Theta, y_c + \rho \sin\Theta) \tag{4.13}$$

The observation must be normalized by the factor

$$R = \sqrt{(x_c + \rho\cos\Theta)^2 + (y_c + \rho\sin\Theta)^2} \tag{4.14}$$

So the normalized observation is

$$(x_n, y_n) = \left(\frac{x_c + \rho\cos\Theta}{\sqrt{(x_c + \rho\cos\Theta)^2 + (y_c + \rho\sin\Theta)^2}}, \frac{y_c + \rho\sin\Theta}{\sqrt{(x_c + \rho\cos\Theta)^2 + (y_c + \rho\sin\Theta)^2}} \right) \tag{4.15}$$

The cosine similarity is simply the dot prodct between (x_n, y_n) and (x_c, y_c), which is equivalent to $\cos(\phi)$.

$$\cos(\phi) = (x_n, y_n).(x_c, y_c) = x_n x_c + y_n y_c \tag{4.16}$$

$$\cos(\phi) = \frac{x_c^2 + \rho x_c \cos\Theta}{\sqrt{(x_c + \rho\cos\Theta)^2 + (y_c + \rho\sin\Theta)^2}} + \frac{y_c^2 + \rho y_c \sin\Theta}{\sqrt{(x_c + \rho\cos\Theta)^2 + (y_c + \rho\sin\Theta)^2}} \tag{4.17}$$

We can make a substitution without loss of generality, by making $x_c = 1$ and $y_c = 0$. This is equivalent to changing the coordinate system and therefore does not cause loss of generality.

$$\cos(\phi) = \frac{1 + \rho\cos\Theta}{\sqrt{(1 + \rho\cos\Theta)^2 + \rho^2\sin^2\Theta}} \tag{4.18}$$

The distance term is then

$$d = 0.5(1 - \cos(\phi)) = 0.5\left(1 - \frac{1 + \rho\cos\Theta}{\sqrt{(1 + \rho\cos\Theta)^2 + \rho^2\sin^2\Theta}}\right) \tag{4.19}$$

To find the probability distribution of d, we must find the cumulative distribution function by integrating the uniform distribution on Θ and the Rayleigh distribution on ρ with limits defined by d above, then differentiating with respect to d.

$$C(d) = \iint\limits_L \left(\frac{1}{2\pi}\right)\left(\frac{\rho}{\sigma^2} e^{-\frac{\rho^2}{2\sigma^2}}\right) d\rho d\Theta, \tag{4.20}$$

where L is defined by

$$0.5\left(1 - \frac{1 + \rho\cos\Theta}{\sqrt{(1 + \rho\cos\Theta)^2 + \rho^2\sin^2\Theta}}\right) < d \tag{4.21}$$

We are simply integrating the joint distribution of the angle and distance of observation to centroid within the limits established by the cosine distance. The integral is simple and is given by

$$f(d) = \left(\frac{\Theta}{2\pi}\right) e^{-\frac{\rho^2}{2\sigma^2}} \tag{4.22}$$

The function value was computed numerically by iterating through values of d and for each value, summing the function (plotted in Figure 4.10) over those regions where the values were d or less. Example regions for $d = 0.05, 0.125, 0.25$ and 0.5 are shown in Figure 4.11. The result is the cumulative distribution function shown in Figure 4.12 which is then fit using linear interpolation over a uniform sampling interval from 0 to 1 and the derivative is calculated to produce the estimate of the PDF of d, depicted in the bottom of Figure 4.12 for $\sigma = 1$. A few different models are considered for fits: Exponential, Rayleigh, and Gaussian. For all models we computed the first and second moments by numerical integration on the numerical PDF. This was performed for $\sigma = 0.5, 1,$ and 2 and the RMS error is shown in Table 4.1.

Exponential Model

The exponential model, while not perfect, serves as the best fit of the candidates and also has the benefit of having a simple cumulative distribution function form.

$$F_E(d|s) = 1 - \exp\left(-\lambda_s d\right) \tag{4.23}$$

Rayleigh Models

While a Rayleigh distribution governs the probability density function for the distance from an observation to the mean for a Gaussian process, this is only true when the distance is measured in Euclidean space. The Rayleigh distribution is the worst fit tested, but also has only a single parameter to compute.

$$F_R(d|s) = 1 - \exp\left(-\frac{d^2}{2\sigma^2}\right) \tag{4.24}$$

Gaussian Models

The Gaussian model is popular but is the second best candidate here. Intuitively a Gaussian model is symmetric about its mean and the numeric PDF is very asymmetric. We can also consider a "truncated Gaussian" which resembles the Gaussian distribution but is bounded (and scaled appropriately); we can compute the variance by reflecting the numeric PDF about the origin and computing the variance assuming the mean is now 0. However, this is not a good fit either as shown in Table 4.1. The likely reason for this is the variance is too large given that the numeric model, after the sharp descent at $d = 0$, does not diminish quickly enough to make a good fit to the Gaussian model. The Gaussian CDF is also more difficult to compute, requiring the evaluation of the erf function as shown below.

Table 4.1: Model Fitting to Cosine Distance

Sigma	Exponential	Rayleigh	Gaussian	TruncGauss
0.5	3.36	5.51	5.08	4.97
1.0	2.80	3.80	3.47	3.40
2.0	1.95	2.63	2.36	2.28

$$F_G(d|s) = \left(\frac{1}{2}\right)\left[1 + \mathrm{erf}\left(\frac{d-\mu_s}{\sigma_s\sqrt{2}}\right)\right] \tag{4.25}$$

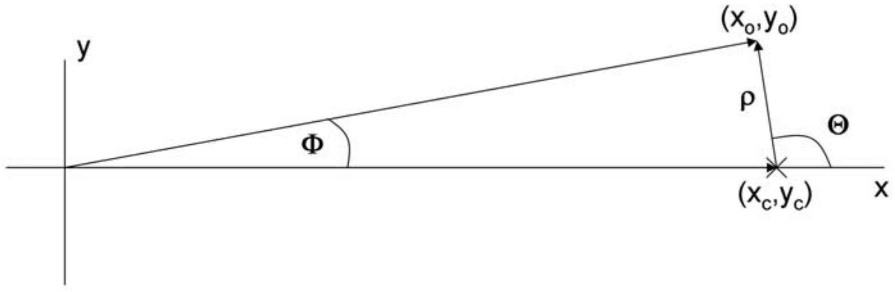


Figure 4.9: Derivation of cosine similarity probability distribution function by transformation from Gaussian distributed data.

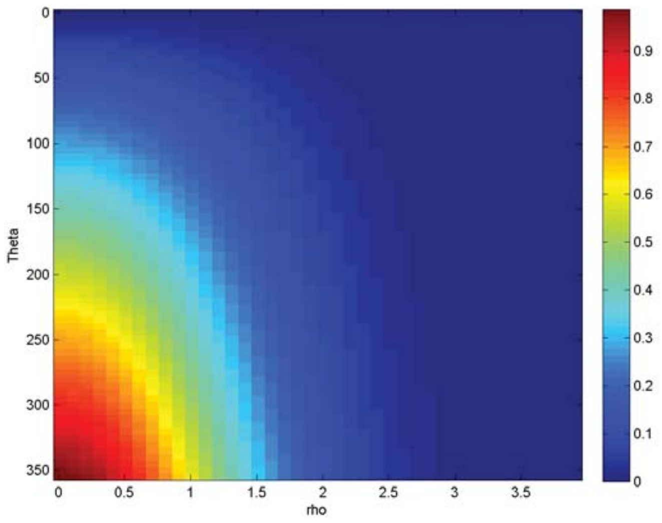


Figure 4.10: Antiderivative of joint pdf for Gaussian process expressed as product of uniform density with angle and Rayleigh distribution of distance.

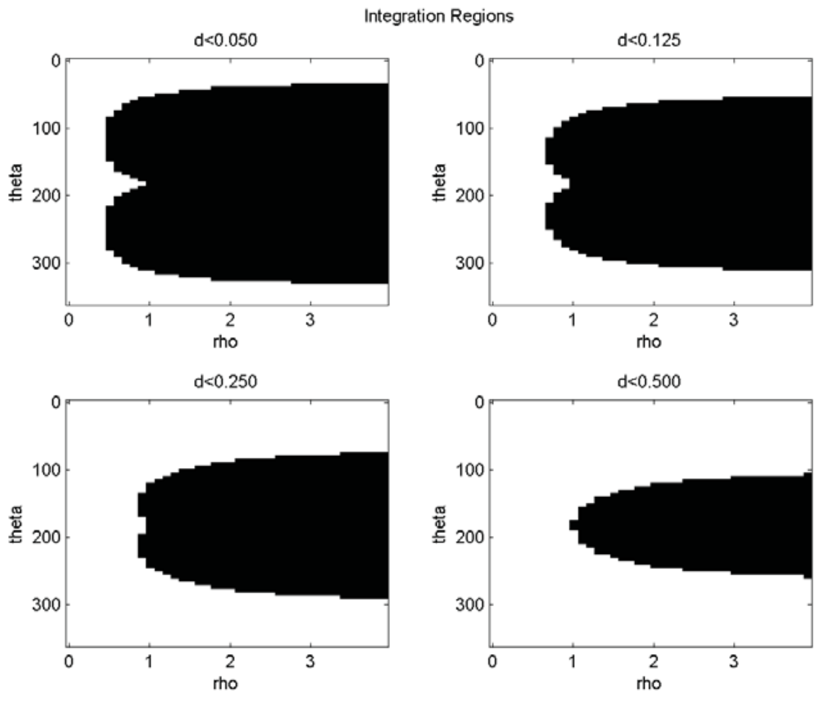
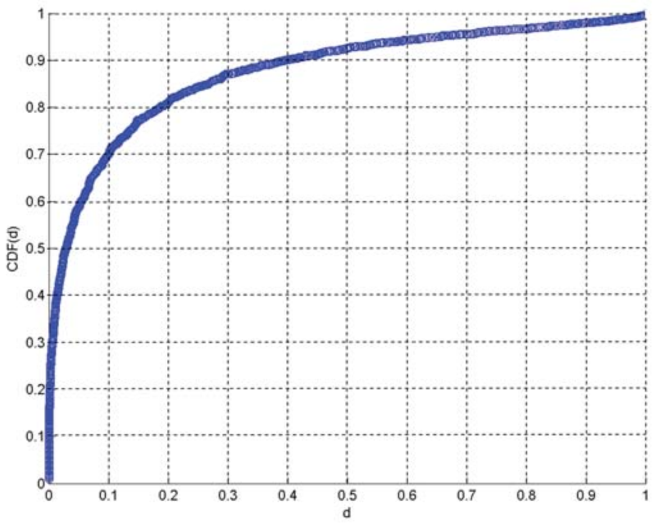


Figure 4.11: Regions of integration for distribution with varying distance d. In these images the white represents the regions of integration, where d is less than (top left) 0.05, (top right) 0.125, (bottom left) 0.25, and (bottom right) 0.50.



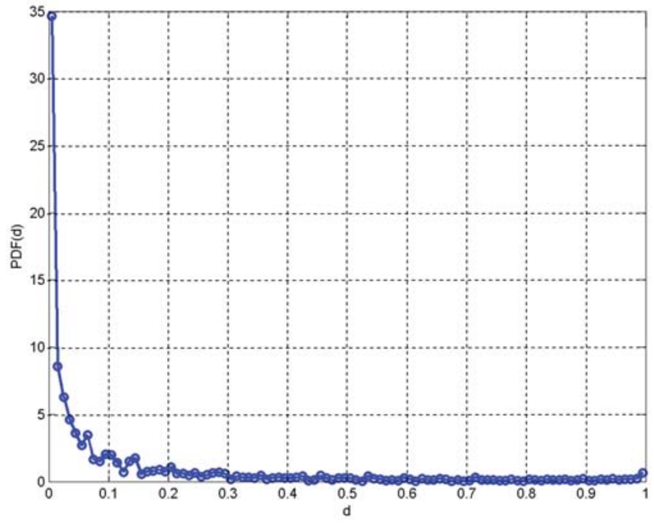


Figure 4.12: Top: CDF from numeric integration. Bottom: PDF after uniform sampling of CDF

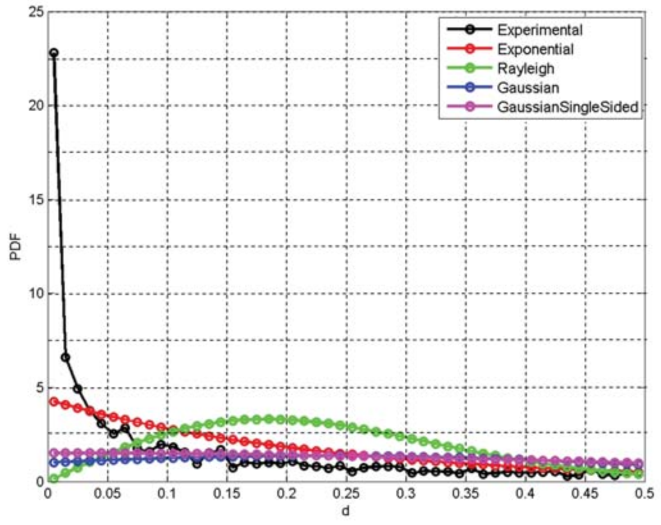


Figure 4.13: Comparison of different PDFs with experimental numerical PDF. Each PDF was computed by calculating the first moment and the second moment (for Gaussian).

Based on this analysis, we believe the exponential distribution is a good choice for the PDF for $P(o|s)$, especially with the cosine similarity metric. In Chapter 6 we compare these with our DeSTIN studies of a handwriting digit database.

In the implementation, the estimate of $P(o|s')$ was therefore made by first computing the parameters of the given distribution from the data online up until the node clustering is terminated, on a centroid-by-centroid basis based on the winning centroid. This represents a set of probabilistic models for observations given each different label and then the $P(o|s')$ can be computed from the cumulative distribution function for each distribution by integrating around a small region around the observation-centroid distance d. For the Gaussian distribution this requires programmatic calls to the erf function, while for the Rayleigh and Exponential models a call to an exponential function is needed.

As a final note, in Korn et al. (2001) the authors use a power law model to fit the distribution of similarities in nearest-neighbor algorithms, which would be an additional option for this probability distribution. Other similarity measures may have different distributions of interest, and some means of actually computing a nonparametric estimate would be of interest as well.

4.4 Online clustering and batch clustering comparison

In this section we compare online clustering and batch clustering performance. We used our online implementation of winner-take-all, with the starvation trace metric and the ρ based stopping criteria, and our implementation of the ECM algorithm. In the WTA and batch algorithms we used the correct number of clusters, which is not necessarily a fair comparison with the ECM algorithm but a reasonable approach for this work.

We note this is not an exhaustive comparison. Instead we have selected several interesting cases and compare the relative performance. We draw conclusions at the end of this section, but note that the scope of this study is necessarily limited.

We show the results graphically with selected representative plots. However, to do a fair comparison between the methods, 20 trials were conducted for each method with different random seeds. This allowed us to compare the methods more thoroughly. These are shown in the tables in the following sections. The final learned centroid locations were compared to the data as a whole to determine the total distance. For each observation, we compute the distance

$$d_j =< o, C_j >, \tag{4.26}$$

where j ranges from 1 to K and gives the distance between observation o and each centroid C_j. Then the total distance D is given as

$$D = \sum_{i=1}^{N} \min(d) \tag{4.27}$$

This is shown in the tables as the median, mean, and standard deviation in the tables under the heading "MedianD", "MeanD", and "STDD". Note that in the tables the learning rate method of 0 is a constant learning rate; 1 is adjusted by $\frac{1}{N}$, which we also call the "k-means learning rate"; and 2 is a constant until $\frac{1}{c}$ are accumulated, at which point we transition to the k-means learning rate. For the ECM methods, we show the median evolved K value across all the trials, but for the WTA and batch methods we used the "correct" number of centroids. A maximum of 25 centroids were used, and the monontonicity threshold was set to -0.7. The value of $D_{thresh} = 0.5$ with adjustments by $\alpha = 0.5$ every 500 samples. The constant learning rate was set to 0.01, which means after 100 labeled data samples for a centroid the rate decayed. As mentioned earlier, each of the five methods (batch, WTA with correct number of centroids, and three different learning rate methods with ECM) was repeated 20 times with different random centroids selected or a different random sequence of data applied. Finally, in each case the data was generated with 4000 points per class, with additive Gaussian random noise of standard deviation σ. The data was generated as Gaussian, with a standard deviation of 0.10, with mean values set by a zero mean, unit standard deviation Gaussian random process. However, we note that in the cosine space we normalize the data.

The iterations required requires a special note. Generally the WTA method used the maximum allowable iterations, especially for smaller numbers of centroids. The ECM methods used a monotonicity criteria, stopping when the monotonicity value was below some threshold (-0.7) after a minimum number of sample evaluations (the last 500 samples evaluated in groups). Note however that we simply noted this stopping point but continued to evaluate the data; this is why the next plots show the performance through the entire data set. The WTA method was also run for the entire data set. From our earlier results, we had set a specification that half the ρ values should be below 0.05 to terminate clustering. Thus, in many of the cases here the clustering was not terminated by the completion of the runs. For the kMeans case, the iterations was computed by taking the number of batch iterations and multiplying by the data size. This is not entirely a fair comparison either, as some strategies could be employed to attempt to reduce the number of data points needed (such as clustering a smaller data set and validating) but the values are included here regardless for scope.

The kmeans algorithm was implemented with Matlab, with a maximum of 100 iterations. According to the Matlab documentation, Mathworks (2012), the algorithm performs batch clustering followed by an online method where each point is updated individually if it lowers the overall mean-square-error. Consequently this is more properly termed a hybrid method.

For our comparison we have selected a variety of particular combinations of K, D, and noise level σ. In particular:

- A low dimensional case with a low number of centroids (D=2, K=4)
- A low dimensional case with a large number of centroids (D=4, K=15)
- A comparison between cases with different levels of noise, in low dimension with large number of clusters (D=2, K=15), with noise levels standard deviation of 0.2, 0.1, and 0.05.
- A high-dimensional, moderate number of clusters (D=100, K=10)

Table 4.2: Online and Batch Clustering Comparison for low dimension, low number of clusters

Method	LRM	K	MedianD	MeanD	STDD	Med Iter
kMeans	NA	4	76.83	182.49	187.76	64000
WTA	0	4	77.37	77.45	0.40	16001
ECM	0	4.00	77.16	77.22	0.29	5708
ECM	1	4.00	76.88	99.09	99.05	1950
ECM	2	4.00	79.54	103.48	98.64	5773

4.4.1 Low dimension with small number of clusters

In this experiment we use a $D = 2$ dimensional space, with $K = 4$ clusters. We see the data plotted for the four clusters as normalized vectors for the cosine similarity metric along with an example (ECM constant learning method) of the resulting centroids in Figure 4.14 . The clustering works well for all online methods, so this plot is representative of all methods. The median number of clusters is correct for all ECM methods. In fact, only a single trial of the kmeans learning rate and constant-kmeans learning rate evolved 3 centroids instead of 4, as shown in Figure 4.15.

The "sum D" measure of the system of centroids is shown in Figure 4.16. The majority of all methods have the minimum total distance of approximately 75, but some of the batch trials had a larger distance, most likely due to an inferior local minimum. The constant learning rate method seems to give the most consistent result, but the most striking trait is the more rapid convergence for the k-means learning rate as shown in Table 4.2.

We plot the ρ metric for the WTA online method in Figure 4.17. We see that the metric gives the desired result of good decay over time and will function as a good stopping criteria, but it does decay more slowly than the other methods.

The monotonicity over time is shown in Figures 4.18 for the k-means ECM method for illustration. In this plot, the monotonicity for a cluster is set to unity for points prior to when it enters the set. We see that the monotonicity drops very rapidly as expected, since the initial sample should be very close, but generally does not change much over time for the ECM methods. We also plot the monotonicity of the WTA method in Figure 4.19, which shows a gradual decay which is expected since the centroids start to navigate toward the true position. Overall the ECM methods converge faster, and all online methods have less error than the batch method and are more consistent for this data.

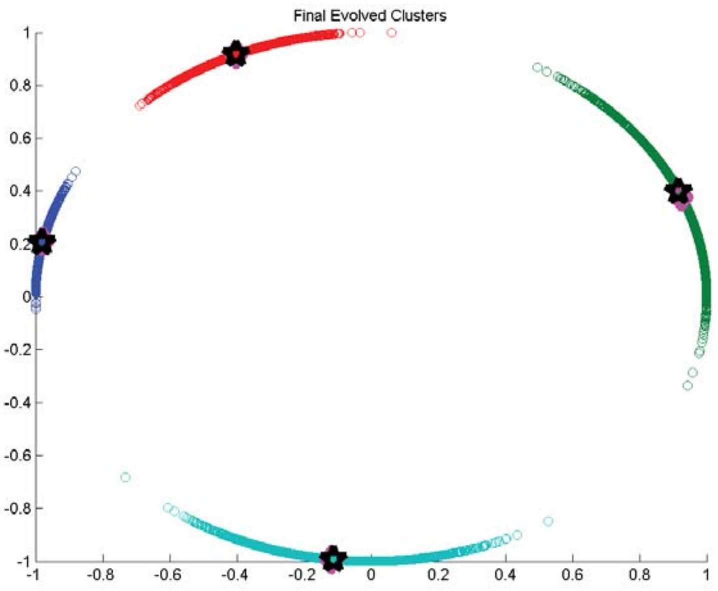


Figure 4.14: Example of clusters for the ECM method with constant learning rate as a representative case for the low-dimensional, low number of centroids case.

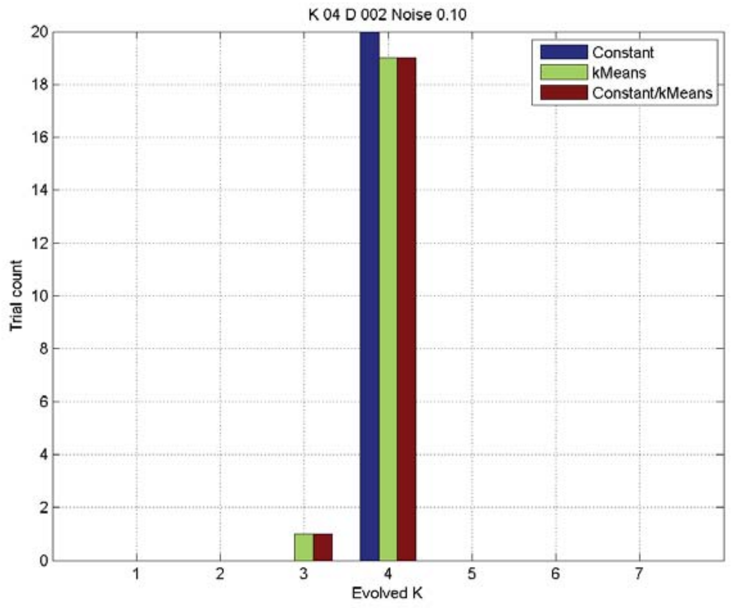


Figure 4.15: Histogram of the evolved number of clusters for ECM methods for the low-dimensional, low-K case, for all trials.

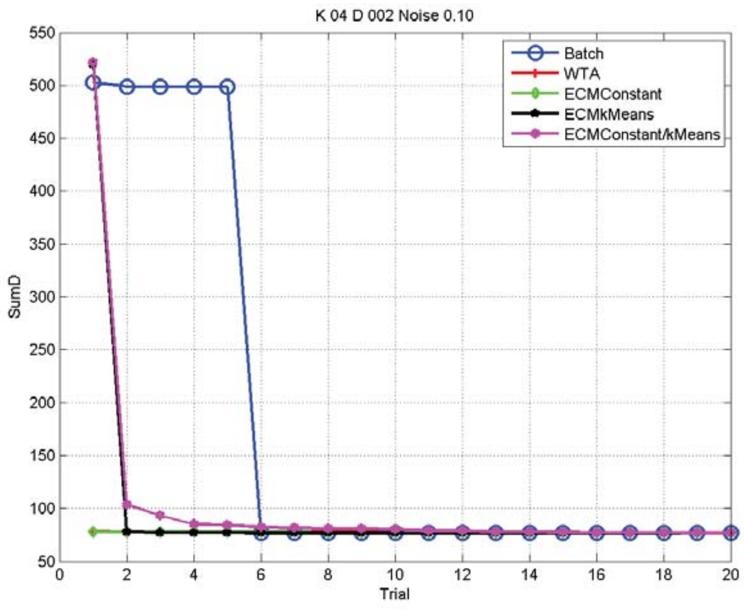


Figure 4.16: Total distance from closest centroid to data points for low dimension, low-K case for all five methods. Each trial is shown but the data is sorted for clarity.

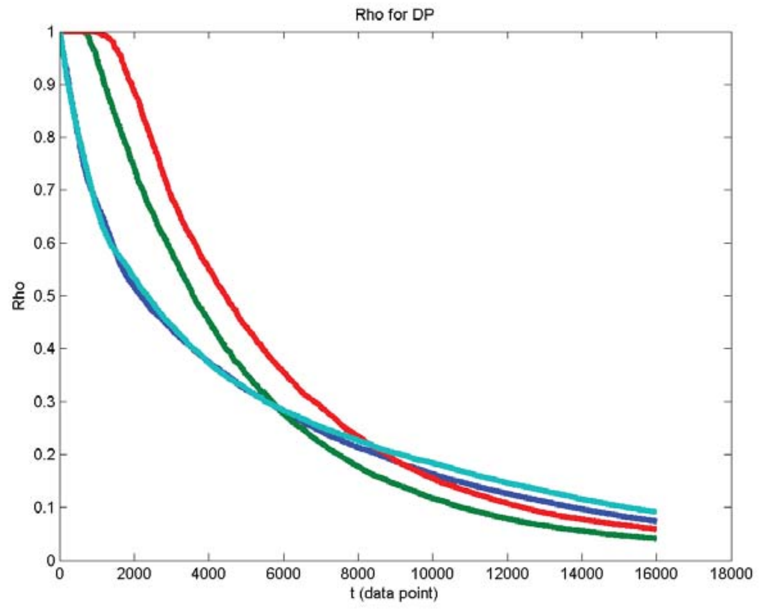


Figure 4.17: The rho metric plotted for the WTA method for the low-dimensional, low-K case.

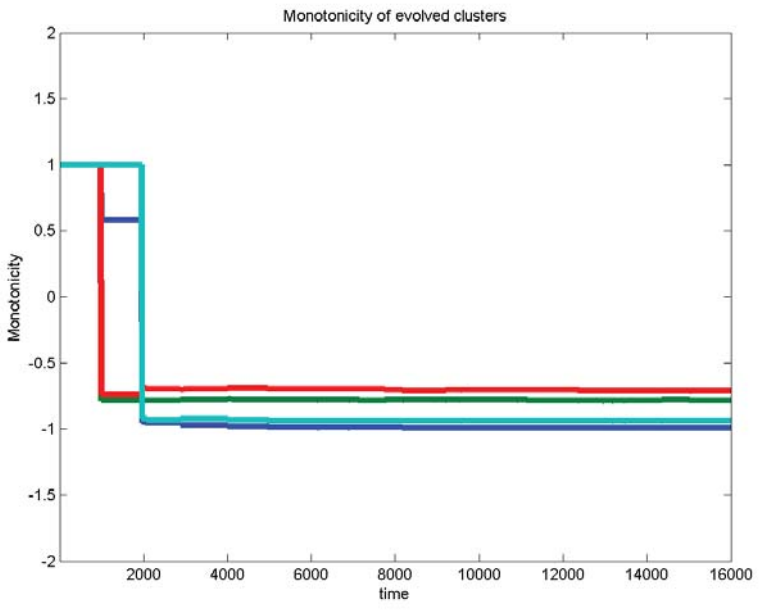


Figure 4.18: Example of monotonicity for the k-means learning rate ECM method for the low-dimensional, low-K case

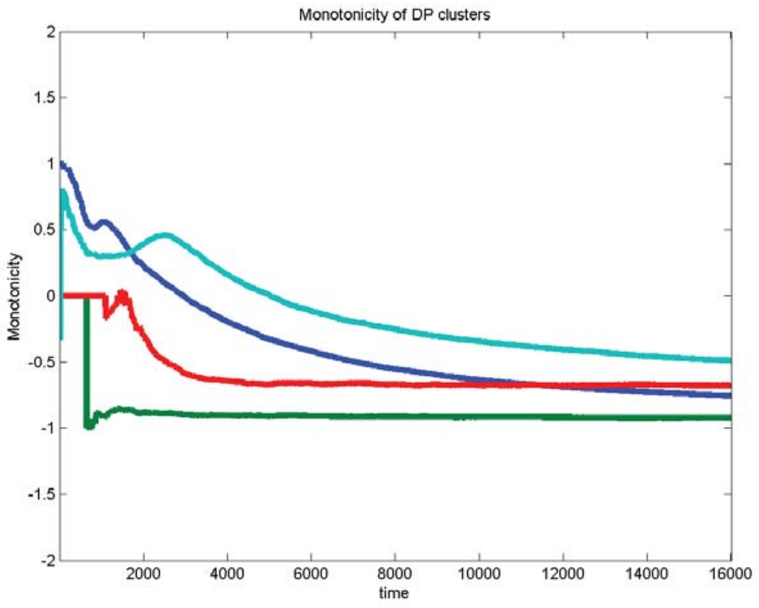


Figure 4.19: Example of monotonicity for the WTA method for comparison for the low-dimensional, low-K case.

Table 4.3: Online and Batch Clustering Comparison for D=4, K=15 case

Method	LRM	K	MedianD	MeanD	STDD	Med Iter
kMeans	NA	15	299.28	529.86	425.10	3450000
WTA	0	15	181.95	184.19	10.79	56523
ECM	0	12.00	263.56	256.50	17.15	60000
ECM	1	12.00	262.56	262.64	0.20	60000
ECM	2	12.00	262.74	257.33	12.99	60000

4.4.2 Higher-dimensional with higher number of clusters

This experiment involved a higher dimension, $D = 4$ with a large number of clusters (K=15) as shown in Figure 4.22 with a clustering results example. For this case the median number of clusters was slightly under the true number (roughly 12 instead of 15) as shown in Figure 4.20. Generally however the online methods perform consistently, and have better or equal error performance than the batch case for most of the trials (Figure 4.21) with the k-means learning rate showing the most consistency. In terms of actual cluster locations, the batch method also does not lead to the necessarily correct clusters either, as shown in Figure 4.22 for example. One disappointing aspect here is that the monotonicity metric does not seem to help us reach a stopping point here; all the evolving methods run for the entire duration of the data set, however most centroids generally do show good monotonicity as shown in Figure 4.25. This suggests some additional criteria could be added to monitor the monotonicity, since the additional threshold splits did not succeed in producing additional centroids or reduce the monotonicity below the threshold for some clusters.

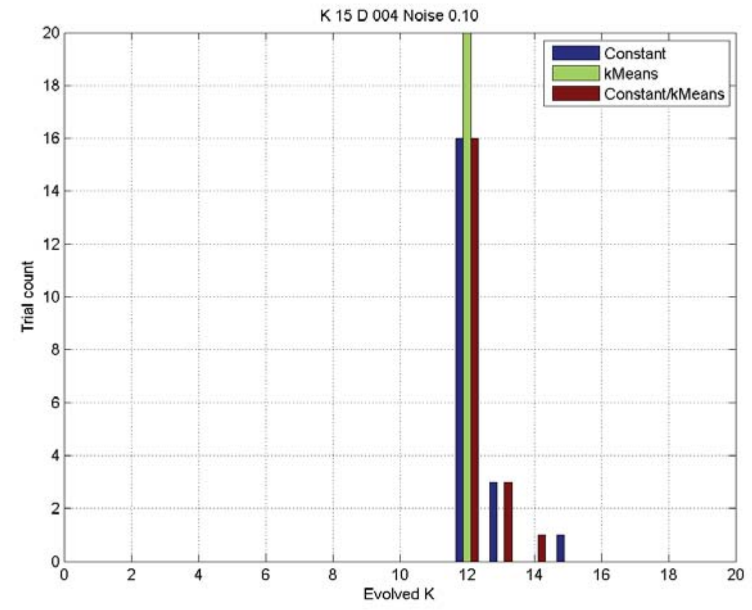


Figure 4.20: Evolved number of clusters for ECM (D=4, K=15 trials).

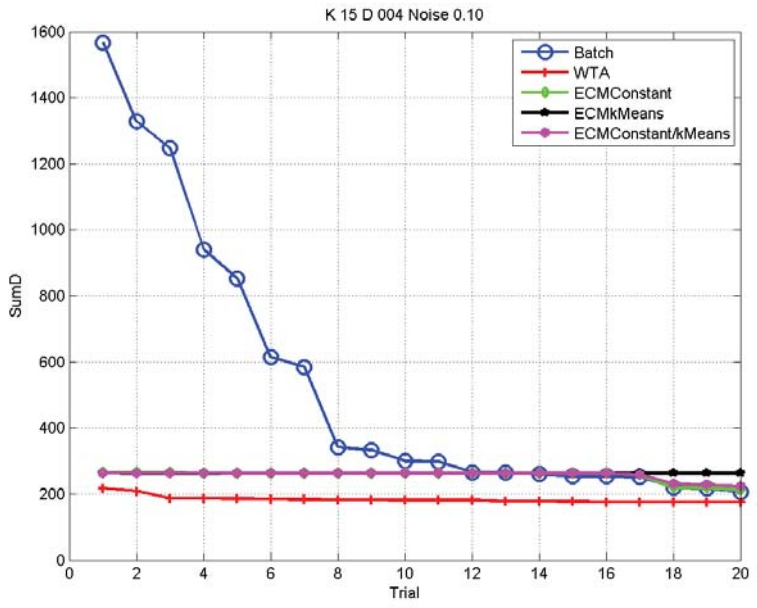


Figure 4.21: Total summed distance from closest centroid to data points for case D=4, K=15.

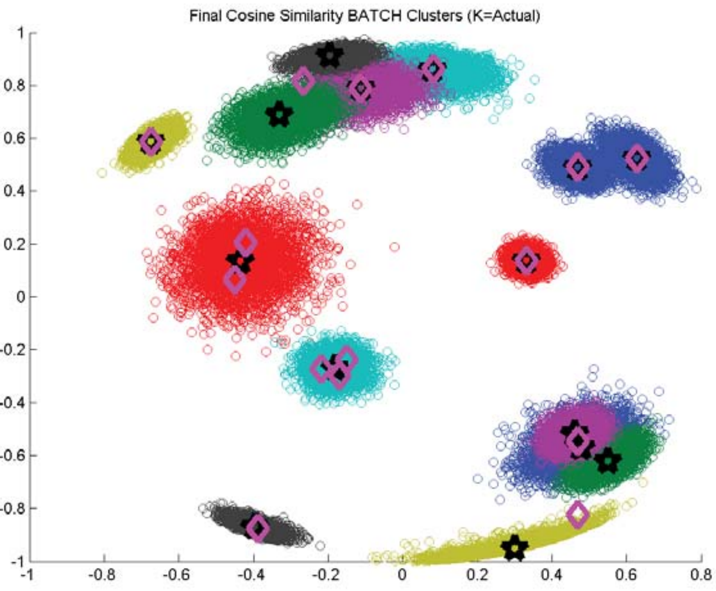


Figure 4.22: Example of the final evolved clusters for the batch method, using D=4 and K=15.

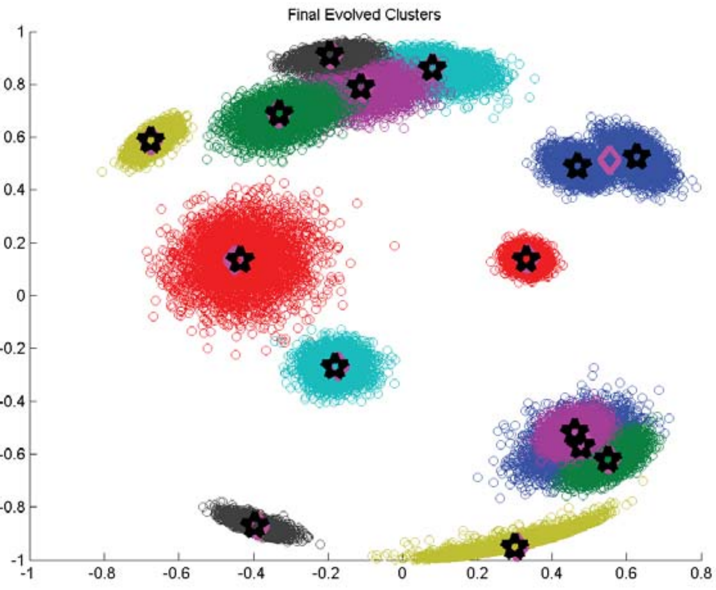


Figure 4.23: Example of the final evolved clusters for the k-means learning rate method, using D=4 and K=15.

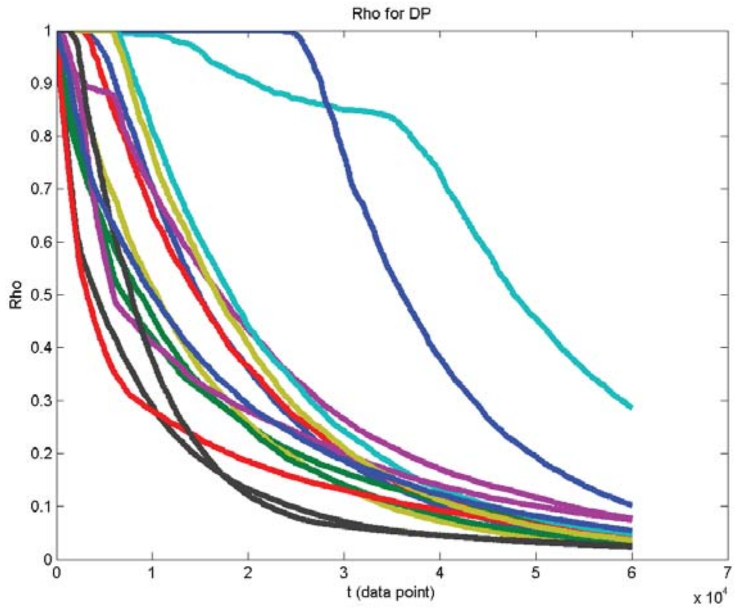


Figure 4.24: Case with D=4, K=15, example of the rho metric for the WTA case.

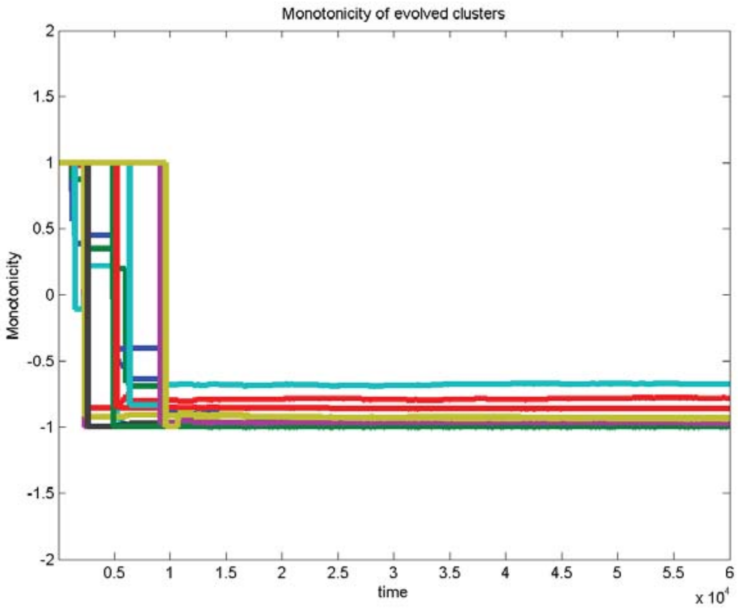


Figure 4.25: Example monotonicity for the ECM method, k-means learning rate, for the case of D=4 and K=15. At least one centroid does not fall below the threshold of -0.7.

4.4.3 Noisy observations with high number of clusters

For these cases, we seek to use a high number of clusters (K=15), in low dimensional space (D=2), changing only the noise level. These cases are summarized in Tables 4.4 through 4.6. The data itself is also illustrated in Figures 4.26 through 4.27. For these cases the comparison is made with respect to each feature we are examining, so we first compare the evolved numbers of clusters. This is shown in Figures 4.28 through 4.30. The median number of clusters evolved does not change much between methods and noise levels, at roughly 8, but the noisiest case does show more variability, evolving more clusters more frequently (although never to the "correct" number).

The batch clustering seems to work the best with respect to minimizing the total distance between data and centroids with the lowest error and variance, although part of this can be attributed to the use of the "correct" number of clusters. This is especially true at the highest noise level, even compared with the WTA method (which also used the "correct" number of centroids). As the noise decreases the WTA method become more comparable, and the other online methods do better as well.

The evolving methods generally did not terminate based on our criteria generally, although the constant followed by k-means approach does show some promise in this regard. As the noise decreases, the ECM methods do terminate, again with the k-means method terminating the fastest with roughly equivalent minimization of the distance metric.

As a final note, although the number of centroids evolved is fairly steady, we do see more variability and a tendency to more centroids with more noise. This is counter-intuitive because we would expect that we would get "more accuracy" as the noise drops. The reason for this behavior is primarily two reasons. First, the clusters are fairly crowded, with 15 centroids in 2D space. Thus, even small levels of noise cause some loss of cohesion and confusion in the algorithm, so less noise does not necessarily correlate to the "right" number of clusters. The second is that the increase in noise caused worse monotonicity scores, forcing the algorithm to lower the D_{thresh} value and increase the number of splitting. With the lower noise cases, the algorithm was not able to distinguish two labeled groups that were close together with little spread; they essentially appeared as the same cluster to the algorithm within the parameters we established. Note that when we say "worse" monotonicity scores we simply mean the values do not drop to the level where the algorithm terminates; instead, they hover at a negative level but not at the desired low range (see Figure 4.34). Certainly setting the value with a less-strict threshold (such as -0.5) or some other criteria could be explored here, but care must be taken to assess the effect of additional heuristic additions to the algorithm.

Table 4.4: Online and Batch Clustering Comparison for low-dimension, high cluster, with highest noise level

Method	LRM	K	MedianD	MeanD	STDD	Med Iter
kMeans	NA	15	153.24	158.01	19.46	6000000
WTA	0	15	209.26	213.72	19.43	60000
ECM	0	9.00	321.04	321.44	27.33	60000
ECM	1	8.00	346.66	345.21	7.77	60000
ECM	2	9.00	319.33	318.41	21.77	52927

Table 4.5: Online and Batch Clustering Comparison for low-dimension, high cluster, with moderate noise level

Method	LRM	K	MedianD	MeanD	STDD	Med Iter
kMeans	NA	15	92.17	109.40	62.91	3570000
WTA	0	15	106.80	112.89	24.02	60000
ECM	0	8.00	195.02	212.17	44.74	10987
ECM	1	8.00	197.95	228.88	77.72	7148
ECM	2	8.00	195.85	196.35	1.82	10456

Table 4.6: Online and Batch Clustering Comparison for low-dimension, high cluster, with lowest noise level

Method	LRM	K	MedianD	MeanD	STDD	Med Iter
kMeans	NA	15	57.60	78.63	65.64	1920000
WTA	0	15	63.98	64.10	20.17	60000
ECM	0	8.00	134.86	163.93	69.71	9565
ECM	1	8.00	136.57	240.60	140.90	6981
ECM	2	8.00	136.33	186.86	90.93	9293

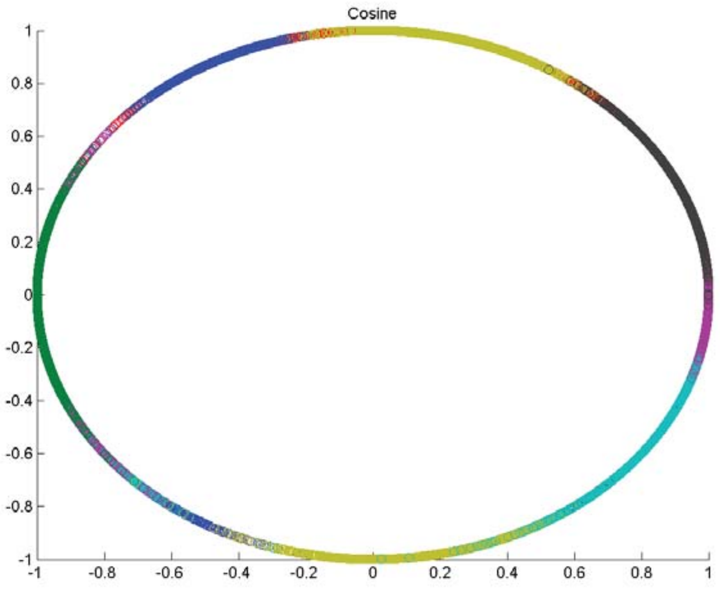


Figure 4.26: Data for D=2, K=15, noise = 0.2, normalized view.

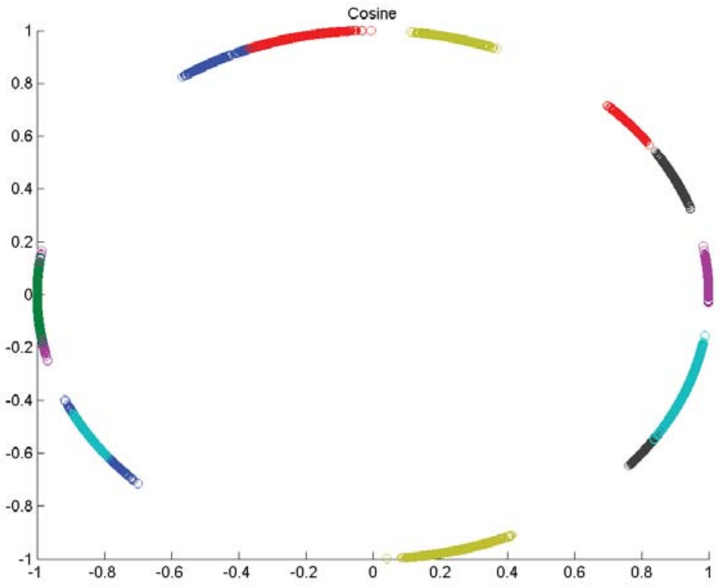


Figure 4.27: Data for D=2, K=15, noise = 0.05, normalized view.

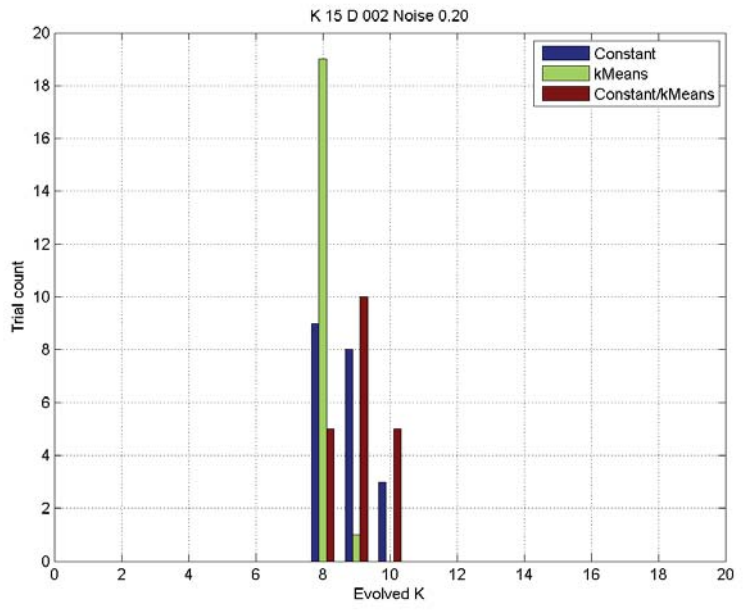


Figure 4.28: Evolved number of clusters for ECM method for highest noise case, for all trials.

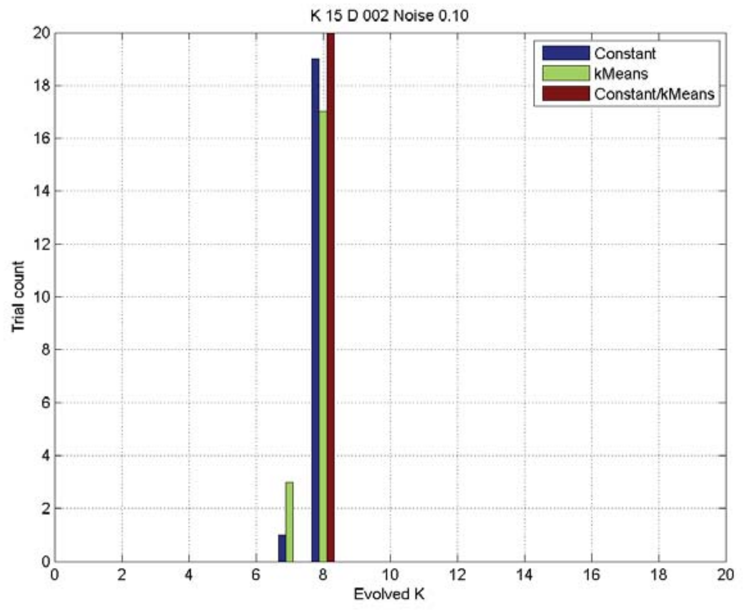


Figure 4.29: Evolved number of clusters for ECM method for moderate noise case, for all trials.

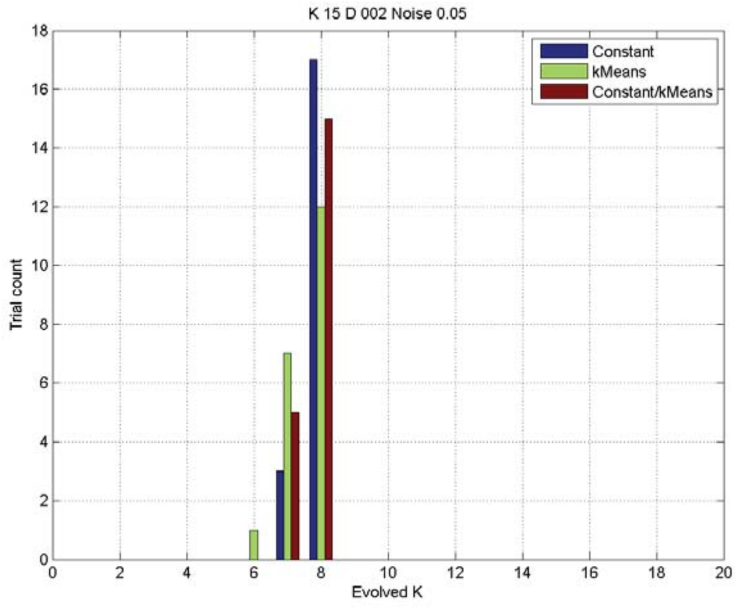


Figure 4.30: Evolved number of clusters for ECM method for lowest noise case, for all trials.

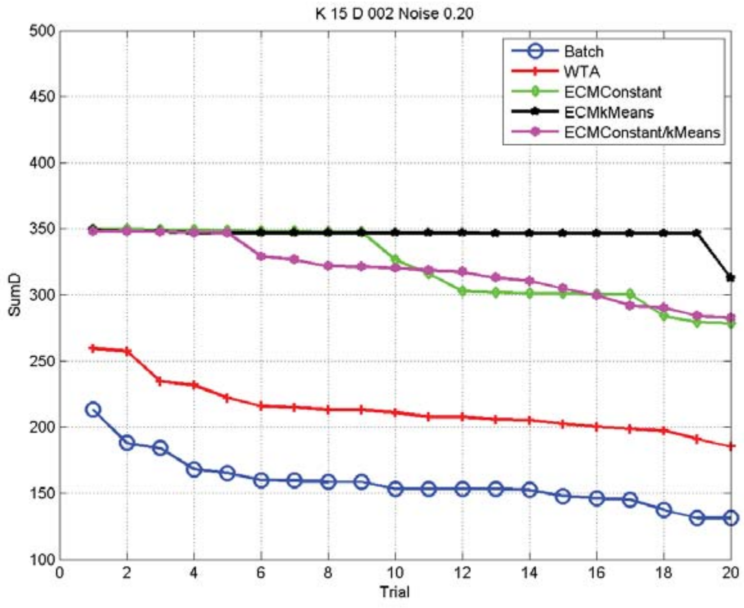


Figure 4.31: Total summed distance from closest centroid to data points for highest noise case.

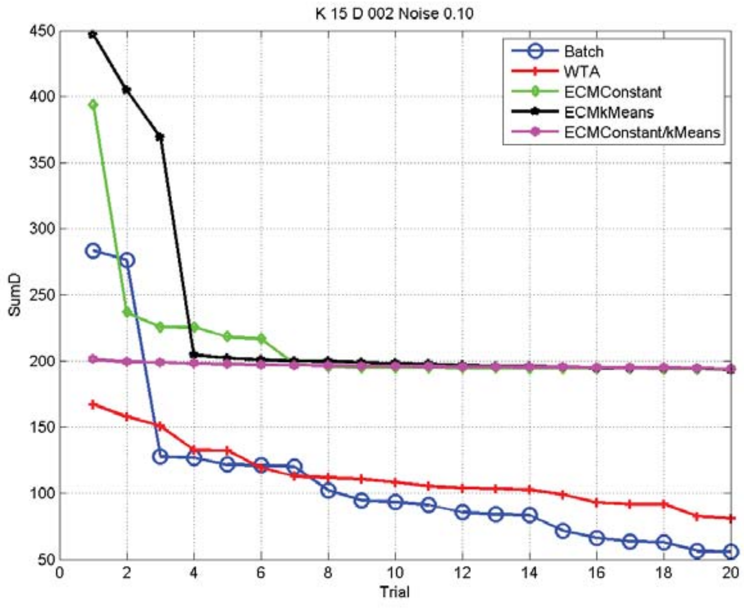


Figure 4.32: Total summed distance from closest centroid to data points for moderate noise case.

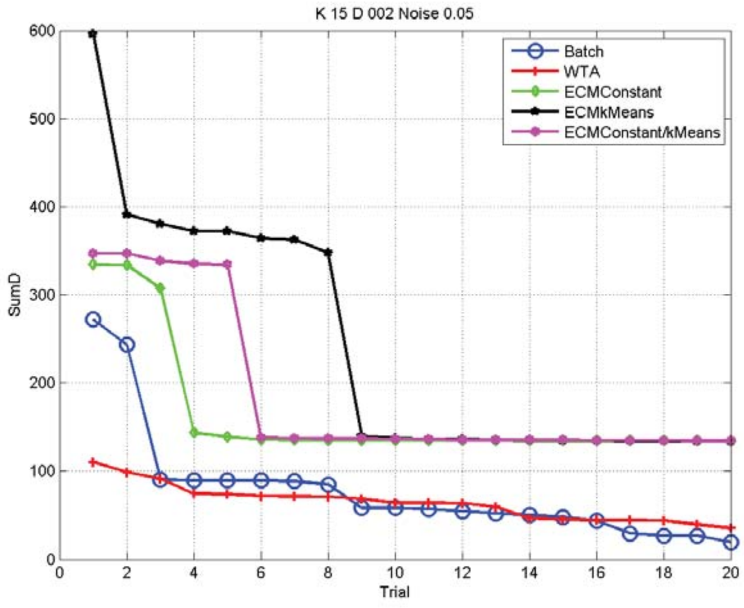


Figure 4.33: Total summed distance from closest centroid to data points for lowest noise case.

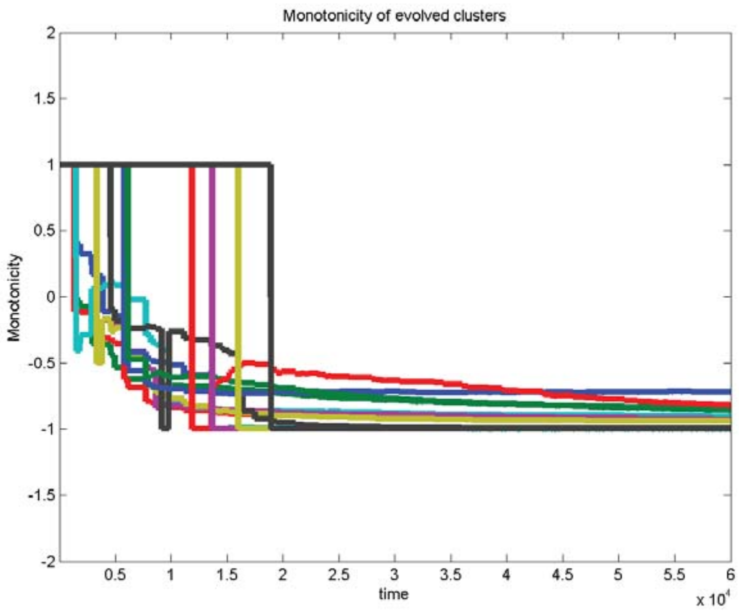


Figure 4.34: Example of monotonicity over time for noisiest case, with D=2, K=15, noise = 0.2.

4.4.4 Comparison with very high dimension and high number of clusters

For a final comparison, we examine a high dimensional experiment (D=100) with a fairly large number of clusters (K=10). This was chosen because this is ideally the type of situation we expect in DeSTIN; high dimensional space, with fewer true clusters relative to the space. In this case the ECM methods perform quite well, as they are able to more quickly focus on the true dimensionality of the space. The batch method has rather large stand deviations and even the median distance value is rather large. One problem here is convergence; in these experiments the batch method failed to converge after 100 iterations. It is certainly feasible that this could be improved by longer iteration times but this is not necessarily our goal here. We note that the ECM methods evolved the correct number of centroids quite easily and had good overall performance, as shown in Figures 4.36 through 4.42.

Table 4.7: Online and Batch Clustering Comparison for case 11

Method	LRM	K	MedianD	MeanD	STDD	Med Iter
kMeans	NA	10	2204.64	1979.01	1130.32	3520000
WTA	0	10	102.62	102.62	0.02	30998
ECM	0	10.00	102.67	102.67	0.05	5067
ECM	1	10.00	102.42	102.44	0.08	5224
ECM	2	10.00	102.99	103.13	0.32	5067

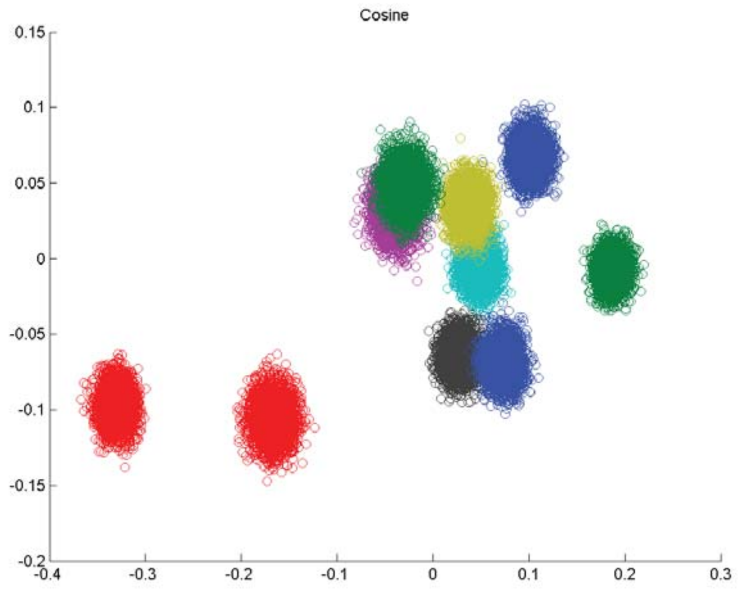


Figure 4.35: High dimensional case (D=100, K=10), view of normalized data.

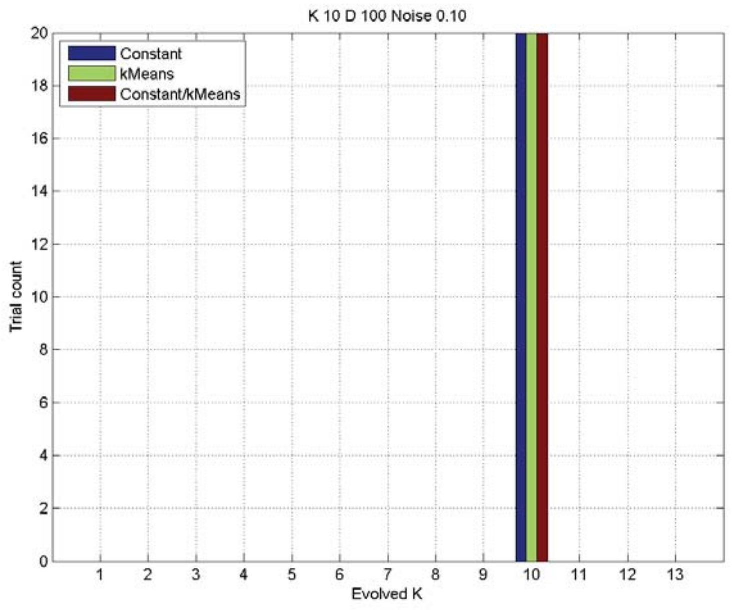


Figure 4.36: Evolved number of clusters for ECM method for high dimensional case, for all trials. The methods consistently evolve the correct number of clusters.

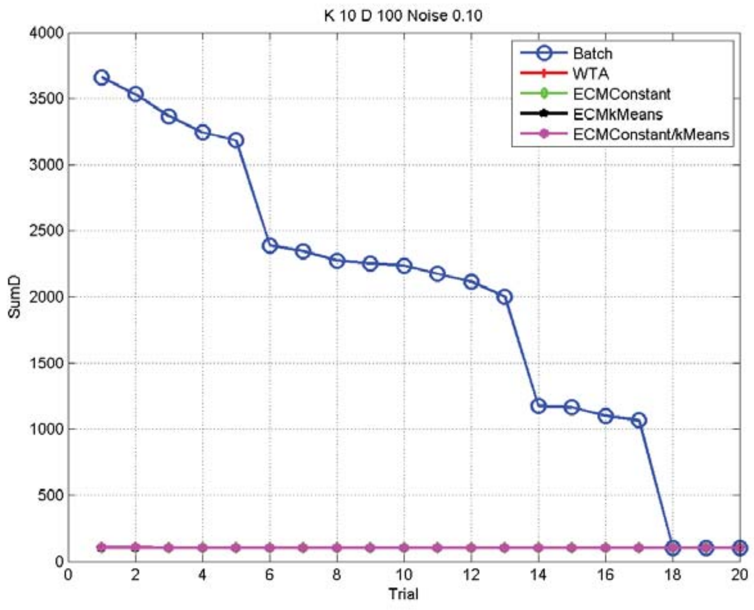


Figure 4.37: Total summed distance from closest centroid to data points for high dimensional case.

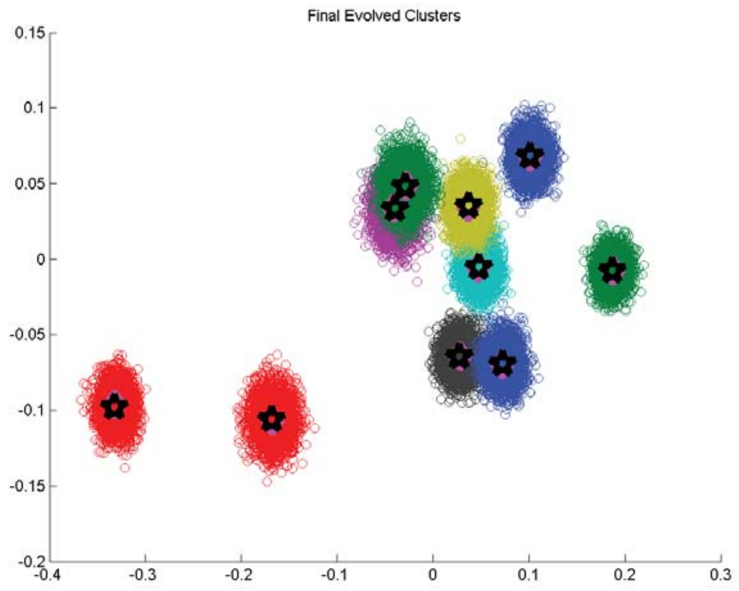


Figure 4.38: An example of the ECM clustering algorithm results, constant learning rate method for the high-dimensional case.

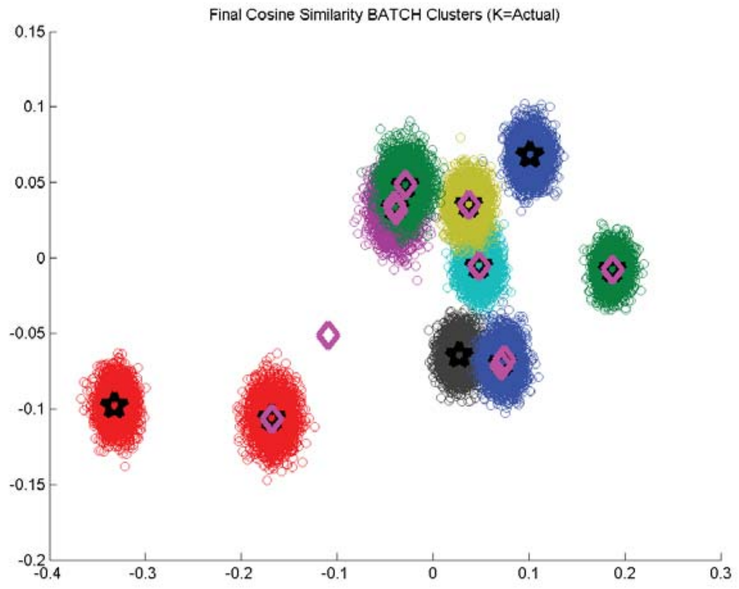


Figure 4.39: An example of the batch clustering algorithm results for the high-dimensional case.

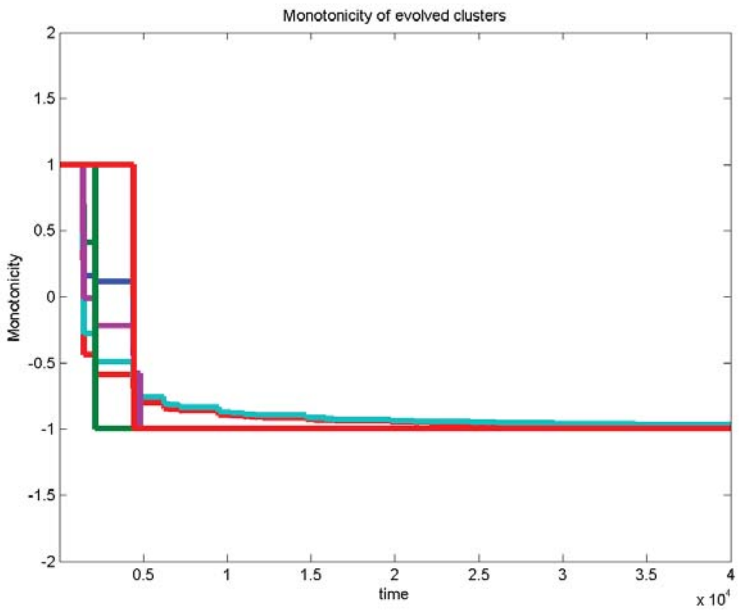


Figure 4.40: An example of the monotonicity with time for ECM, with constant followed by k-means learning rate method for the high-dimensional data.

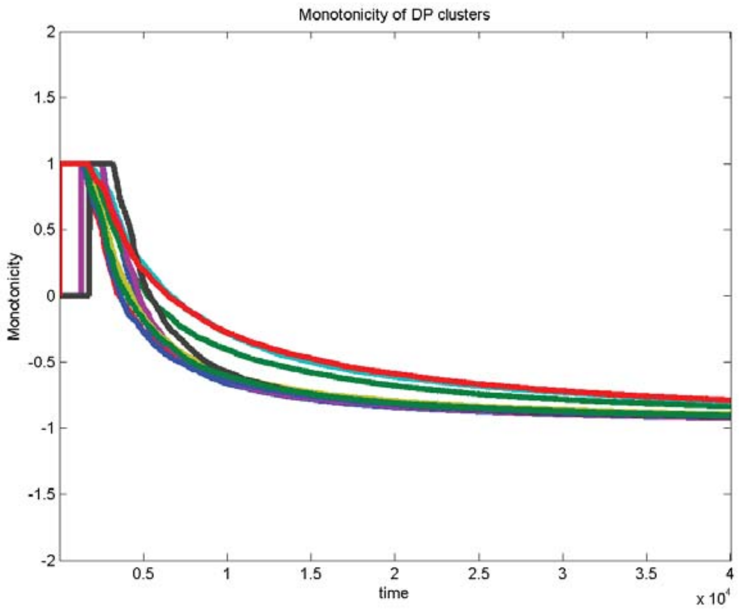


Figure 4.41: An example of the monotonicity with time for WTA method with the high-dimensional data.

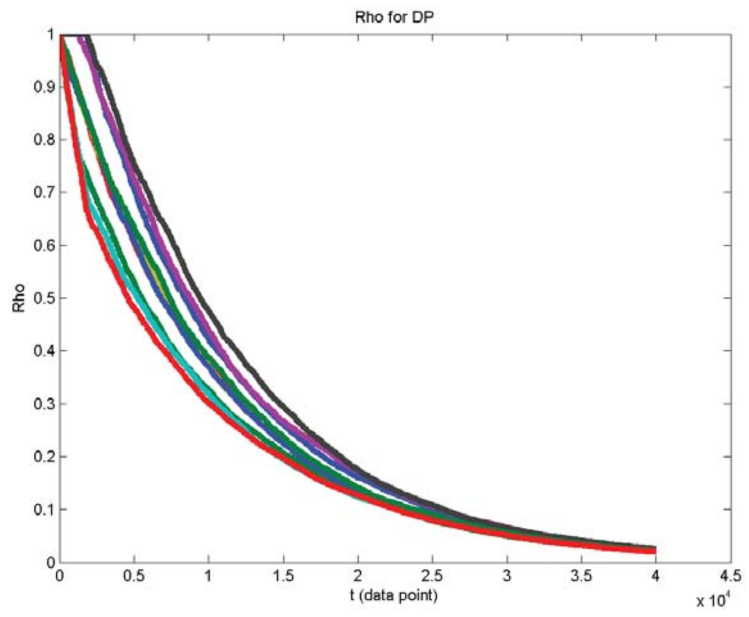


Figure 4.42: An example of the rho metric over time for the WTA method, high dimensional data case.

4.4.5 Summary

We compared clustering algorithms from batch (k-means) and online (ECM, with different variations of the learning rate, and WTA using starvation trace). This comparison was motivated by an attempt to create a static learning module for DeSTIN which featured online learning with an adaptive or evolving number of clusters, along with an interest in comparing online and batch methods. Overall, all algorithms worked well, but the online algorithms seemed to offer better performance with respect to robustness, except possibly for the cases of high numbers of clusters in low dimensional space. Intuitively, we would expect k-means batch to function better since it has access to all the data simultaneously, but there are a few fallacies here. First, the initialization of the method which we used was random choice of centroids, so it can be sensitive to the initialization process. This is a known issue with random initializations and there have been proposed solutions to this Xu and Wunsch (2005), but we limited our comparison to the random initialization. The online WTA method with random initializations often outperformed the batch algorithm, which initially is surprising but this has been reported before in Zhong (2005), Martinetz et al. (1993), Banerjee and Ghosh (2004). Our different variants to the ECM algorithm specifically the use of monotonicity as an adaptive distance threshold and stopping criteria seem to be effective in these cases, both in experiments and from heuristic arguments. Finally, it appears that generally the k-means derived learning rate is the most effective under ideal circumstances, especially with respect to termination, but the constant followed by k-means has merit too, especially in the lower-noise, low-dimension but high-K case.

Chapter 5

Dynamic and Supervised Learning

In this chapter we explore the temporal dynamic aspect of DeSTIN and how it is learned and applied. We start by showing how the temporal process, denoted $P_{ss'}^{a}$, learns and generates a composite belief across multiple time steps. We start with a tabular assumption for the learning - which consists of simply counting states or "winning cluster labels" - due to its simplicity. As a related processing function, we discuss the advice generation process. Since these methods require considerable memory using tabular implementations, we also discuss methods for using function approximation to create these same functions. In this chapter, we also cover the process of selecting nodes for supervised learning. We summarize three main method we call hand-selected for historical comparison, topological, and PCA-based. We seek to use these methods as unsupervised, requiring no advance knowledge of class labels.

5.1 The Temporal State Transition Process

The $P_{ss'}^{a}$ from Equation 3.13 simply states the probability of transitioning from state s to s', with a given advice component. We can represent the $P_{ss'}^{a}$ component as a set of tables which are learned by simply counting the number of transitions from a state s to a state s', when the advice is element a. There are similarities here to the reinforcement learning concept of the state-action table as discussed briefly in Chapter 2, where the advice takes the place of the action, but here there is no optimization component from the learned a; rather it must be unsupervised, at least in this initial formulation. As in RL methods, the $P_{ss'}^{a}$ table can be learned from actual transitions. This is memory intensive, depending on the state-advice state space, but is a reasonable approach and thus is our main method for this dissertation, although we explore the use of function approximation methods here as well. We note that we have assumed some small, non-zero element for each entry in the $P_{ss'}^{a}$ table, so that even unlikely state transitions will not return a response of exactly 0.

For the implementation, each node at a layer has a set of A tables or matrices, each size $S \times S$, initialized to a uniform value of $\frac{1}{S}$ so that each row sums to unity.

After some number of iterations (possibly as few as zero) have passed to allow the clustering and belief state estimates to stabilize, we begin the process of $P^a_{ss'}$ updates. These updates proceed by finding the estimated current belief state as simply the maximum of beliefs,

$$\hat{b} = \arg\max_s b_s, \tag{5.1}$$

which is essentially the index of the winning centroid label. This is followed by updating the estimate of P^a_{ss} using this winning state and the advice a from the parent node. When P^a_{ss} is the matrix associated with advice state a, the update proceeds as

$$P^{a^{(t+1)}}_{ss} = (1-\kappa)P^{a^t}_{ss} + \kappa\delta(\hat{b}) \tag{5.2}$$

where $\delta(\hat{b})$ is the Kronecker delta function. The belief state of each node is then approximated as the following matrix-vector operations, where b is the $S \times 1$ vector of previous beliefs:

$$v_k = b_k P(o|s_k), \tag{5.3}$$

and

$$b' = P^a_{ss} v. \tag{5.4}$$

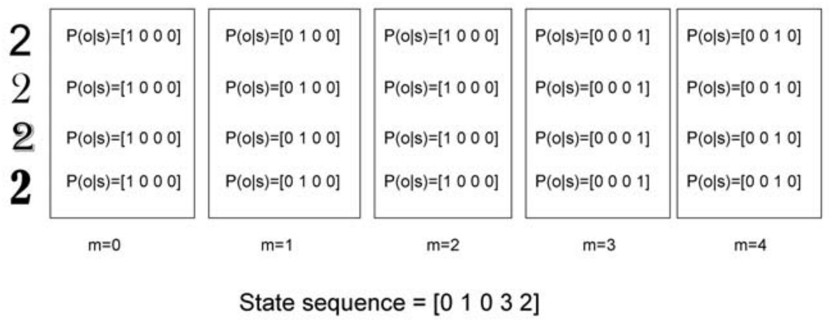
Note that entry r, c in the $P^a_{ss'}$ table is the probability of transitioning from state c to state r given we are in state c. As a final step in the belief state update we normalize b to sum to unity. We show this in matrix form in the following equations.

$$b' = P^a_{ss'} P(o|s) b \tag{5.5}$$

$$b' = \begin{pmatrix} P^a_{ss'(1,1)} & P^a_{ss'(1,2)} & \cdots & P^a_{ss'(1,n)} \\ P^a_{ss'(2,1)} & P^a_{ss'(2,2)} & \cdots & P^a_{ss'(2,n)} \\ \vdots & \vdots & \ddots & \vdots \\ P^a_{ss'(S,1)} & P^a_{ss'(S,2)} & \cdots & P^a_{ss'(S,n)} \end{pmatrix} \begin{pmatrix} POS_1 & 0 & \cdots & 0 \\ 0 & POS_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & POS_S \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_S \end{pmatrix} \tag{5.6}$$

5.1.1 Limitations with Cluster Labels

In our initial DeSTIN work the original intention was that the belief formulation given by Equation 3.13 would temporally evolve a single consistent set of beliefs about a group of states s which represented the underlying data distribution found by unsupervised learning and a probability distribution $P(o|s)$. However, we eventually became aware that this formulation as itself would not function well. The basic concept suffers from a need for consistent labeling from temporal movement to temporal movement. As an example, if the observation is a 4×4 pixel region which is all black except for a single element, a single shift will create a cosine similarity of 0, since the resulting vectors have completely different directions. Other distance metrics could be imposed, but even then the method has issues as illustrated in Figure 5.3, where we show the failure to arrive at a consistent belief in the case of a very simple example.



$$P^a_{ss'} = \begin{bmatrix} 0.0 & 0.5 & 0.0 & 0.5 \\ 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \end{bmatrix}$$

Figure 5.1: Belief propagation with clustering. Whenever a “2” is presented for four movements, the observation clusters to labels 0, 1, 0, 3 with probabilty 1.

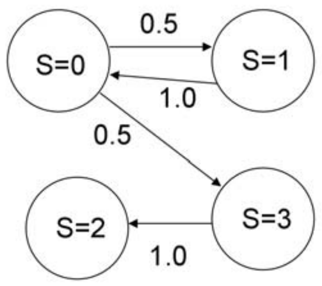


Figure 5.2: State transition diagram. This is deceptive because in reality we never go from state 0 to state 3 without first passing through state 1; thus the system is not Markov.

First movement

$$\underset{P^{a}_{ss'}}{\begin{bmatrix} 0.0 & 0.5 & 0.0 & 0.5 \\ 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \end{bmatrix}} \underset{P(o|s)}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}} \underset{b}{\begin{bmatrix} 0.25 \\ 0.25 \\ 0.25 \\ 0.25 \end{bmatrix}} = \begin{bmatrix} 0.00 \\ 0.125 \\ 0.00 \\ 0.125 \end{bmatrix} \Rightarrow \underset{b'}{\begin{bmatrix} 0.00 \\ 0.50 \\ 0.00 \\ 0.50 \end{bmatrix}}$$

Third movement

$$\underset{P^{a}_{ss'}}{\begin{bmatrix} 0.0 & 0.5 & 0.0 & 0.5 \\ 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \end{bmatrix}} \underset{P(o|s)}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}} \underset{b}{\begin{bmatrix} 1.00 \\ 0.00 \\ 0.00 \\ 0.00 \end{bmatrix}} = \begin{bmatrix} 0.00 \\ 0.50 \\ 0.00 \\ 0.50 \end{bmatrix} \Rightarrow \underset{b'}{\begin{bmatrix} 0.00 \\ 0.50 \\ 0.00 \\ 0.50 \end{bmatrix}}$$

Third movement

$$\underset{P^{a}_{ss'}}{\begin{bmatrix} 0.0 & 0.5 & 0.0 & 0.5 \\ 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \end{bmatrix}} \underset{P(o|s)}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}} \underset{b}{\begin{bmatrix} 1.00 \\ 0.00 \\ 0.00 \\ 0.00 \end{bmatrix}} = \begin{bmatrix} 0.00 \\ 0.50 \\ 0.00 \\ 0.50 \end{bmatrix} \Rightarrow \underset{b'}{\begin{bmatrix} 0.00 \\ 0.50 \\ 0.00 \\ 0.50 \end{bmatrix}}$$

Fourth movement

$$\underset{P^{a}_{ss'}}{\begin{bmatrix} 0.0 & 0.5 & 0.0 & 0.5 \\ 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \end{bmatrix}} \underset{P(o|s)}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}} \underset{b}{\begin{bmatrix} 0.00 \\ 0.50 \\ 0.00 \\ 0.50 \end{bmatrix}} = \begin{bmatrix} 0.00 \\ 0.00 \\ 0.50 \\ 0.00 \end{bmatrix} \Rightarrow \underset{b'}{\begin{bmatrix} 0.00 \\ 0.00 \\ 1.00 \\ 0.00 \end{bmatrix}}$$

Figure 5.3: Example of belief propagation after a new "2" is shown. The first movement is on top; The $P(o|s)$ is on the diagonal, which is then multiplied by the PSSA table and the initial b (set to a uniform distribution). The output relies on the input and essentially validates it. In the second movement we again validate the input; a transition from the previous belief (state 0) with the new $P(o|s)$ (state 1) gives a new belief that we are indeed in state 1. Next we transition from state 1 back to state 0, then finally at the fourth movement we note the belief vector is not stable; it changes with a pattern, but is different after each movement.

5.1.2 Overcoming the limitations with multiple observations

To combat this problem, in Karnowski et al. (2010b) the temporal dynamics module was replaced by simple sampling of the estimated beliefs at intervals T, which are discussed in Chapter 6. Although we achieved good results with this approach, the resulting feature space was much higher than desired and also did not utilize hierarchical feedback in any form. Instead, we propose to use the advice in a multiple-observer model to formulate a set of "belief in advice states" or $B(a)$ which serves as a cumulative estimate of the belief in the advice state of a node. This allows each node to incorporate learning from the parent node over time to generalize to long temporal scales, but retain a level of local knowledge and thus ideally capture sufficient variation to make good supervised learning classifiers. In this mechanism, the system dynamics thus vary from movement to movement and the temporal dependence must be maintained. Each child node retains a brief history of its recent observations and beliefs within a single subject presentation corresponding to the temporal scans discussed in Chapter 3, such as Figure 3.2. Then when the parental advice is available, the $P^a_{ss'}$ is learned across the past observations and movements. We show the basic concept in Figure 5.6. Finally, during testing (when the network has stabilized and responses to presentations are sought, the $B(a)$ is computed using a multiple observer model. Each possible advice state is interpreted as a different observer, with multiple attempts at observation as well (one for each movement), and thus the belief in that advice state is computed as a cumulative prediction given as

$$B(a) = \sum_m \sum_s b(s_m|a) \tag{5.7}$$

where s are all possible states and m are all movements. When there are a total of A advice states, this produces a vector of dimension A for the output of each node. The resulting vector is accumulated over the entire movement sequence and thus results in a single estimate across the entire temporal scope. The advice component is passive, meaning we simply compute the value for $B(a)$ and test for each advice state using the model residing in each node for each different advice state learned during the training process. This approach is more robust in the sense that we do not depend on “good” advice in real-time from the parent, and also occasionally unforeseen observations with zero probability do not cause the entire evolution to grind to a halt.

First movement a=0

$$\underbrace{\begin{bmatrix}0.0&0.0&0.0&0.0\\1.0&0.0&0.0&0.0\\0.0&0.0&0.0&0.0\\0.0&0.0&0.0&0.0\end{bmatrix}}_{P^{a}_{ss'}}\underbrace{\begin{bmatrix}1&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{bmatrix}}_{P(o|s)}\underbrace{\begin{bmatrix}0.25\\0.25\\0.25\\0.25\end{bmatrix}}_{b}=\begin{bmatrix}0.00\\0.25\\0.00\\0.00\end{bmatrix}\Rightarrow\underbrace{\begin{bmatrix}0.00\\1.00\\0.00\\0.00\end{bmatrix}}_{b'}$$

B(a=0)=0.25

Second movement a=0

$$\underbrace{\begin{bmatrix}0.0&1.0&0.0&0.0\\0.0&0.0&0.0&0.0\\0.0&0.0&0.0&0.0\\0.0&0.0&0.0&0.0\end{bmatrix}}_{P^{a}_{ss'}}\underbrace{\begin{bmatrix}0&0&0&0\\0&1&0&0\\0&0&0&0\\0&0&0&0\end{bmatrix}}_{P(o|s)}\underbrace{\begin{bmatrix}0.00\\1.00\\0.00\\0.00\end{bmatrix}}_{b}=\begin{bmatrix}1.00\\0.00\\0.00\\0.00\end{bmatrix}\Rightarrow\underbrace{\begin{bmatrix}1.00\\0.00\\0.00\\0.00\end{bmatrix}}_{b'}$$

B(a=0)=1.25

Third movement a=0

$$\underbrace{\begin{bmatrix}0.0&0.0&0.0&0.0\\0.0&0.0&0.0&0.0\\0.0&0.0&0.0&0.0\\1.0&0.0&0.0&0.0\end{bmatrix}}_{P^{a}_{ss'}}\underbrace{\begin{bmatrix}1&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{bmatrix}}_{P(o|s)}\underbrace{\begin{bmatrix}1.00\\0.00\\0.00\\0.00\end{bmatrix}}_{b}=\begin{bmatrix}0.00\\0.00\\0.00\\1.00\end{bmatrix}\Rightarrow\underbrace{\begin{bmatrix}0.00\\0.00\\0.00\\1.00\end{bmatrix}}_{b'}$$

B(a=0)=2.25

Fourth movement a=0

$$\underbrace{\begin{bmatrix}0.0&0.0&0.0&0.0\\0.0&0.0&0.0&0.0\\0.0&0.0&0.0&1.0\\0.0&0.0&0.0&0.0\end{bmatrix}}_{P^{a}_{ss'}}\underbrace{\begin{bmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&1\end{bmatrix}}_{P(o|s)}\underbrace{\begin{bmatrix}0.00\\0.00\\0.00\\1.00\end{bmatrix}}_{b}=\begin{bmatrix}0.00\\0.00\\1.00\\0.00\end{bmatrix}\Rightarrow\underbrace{\begin{bmatrix}0.00\\0.00\\1.00\\0.00\end{bmatrix}}_{b'}$$

B(a=0)=3.25

Figure 5.4: Cumulating beliefs movement-by-movement given advice state 0. At the top, from the uniform start we see b' which we add to our cumulative belief, initialized as 0. In the second movement we get a new belief which is added to the entire observation sequence, which is then accumulated again for the third and fourth movements for a final $B(a = 0) = 3.25$.

First movement a=1; sequence [0 0 0 0 0]

$$\underset{P^{a}_{ss'}}{\begin{bmatrix}1.0 & 0.0 & 0.0 & 0.0\\ 0.0 & 0.0 & 0.0 & 0.0\\ 0.0 & 0.0 & 0.0 & 0.0\\ 0.0 & 0.0 & 0.0 & 0.0\end{bmatrix}} \underset{P(o|s)}{\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\end{bmatrix}} \underset{b}{\begin{bmatrix}0.25\\ 0.25\\ 0.25\\ 0.25\end{bmatrix}} = \begin{bmatrix}0.25\\ 0.00\\ 0.00\\ 0.00\end{bmatrix} \Rightarrow \underset{b'}{\begin{bmatrix}1.00\\ 0.00\\ 0.00\\ 0.00\end{bmatrix}}$$

B(a=1)=0.25

Second movement a=1

$$\underset{P^{a}_{ss'}}{\begin{bmatrix}1.0 & 0.0 & 0.0 & 0.0\\ 0.0 & 0.0 & 0.0 & 0.0\\ 0.0 & 0.0 & 0.0 & 0.0\\ 0.0 & 0.0 & 0.0 & 0.0\end{bmatrix}} \underset{P(o|s)}{\begin{bmatrix}0 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\end{bmatrix}} \underset{b}{\begin{bmatrix}1.00\\ 0.00\\ 0.00\\ 0.00\end{bmatrix}} = \begin{bmatrix}0.00\\ 0.00\\ 0.00\\ 0.00\end{bmatrix} \Rightarrow \underset{b'}{\begin{bmatrix}0.25\\ 0.25\\ 0.25\\ 0.25\end{bmatrix}}$$

B(a=1)=0.25

Third movement a=1

$$\underset{P^{a}_{ss'}}{\begin{bmatrix}1.0 & 0.0 & 0.0 & 0.0\\ 0.0 & 0.0 & 0.0 & 0.0\\ 0.0 & 0.0 & 0.0 & 0.0\\ 0.0 & 0.0 & 0.0 & 0.0\end{bmatrix}} \underset{P(o|s)}{\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\end{bmatrix}} \underset{b}{\begin{bmatrix}0.25\\ 0.25\\ 0.25\\ 0.25\end{bmatrix}} = \begin{bmatrix}0.25\\ 0.00\\ 0.00\\ 0.00\end{bmatrix} \Rightarrow \underset{b'}{\begin{bmatrix}1.00\\ 0.00\\ 0.00\\ 0.00\end{bmatrix}}$$

B(a=1)=0.50

Fourth movement a=1

$$\underset{P^{a}_{ss'}}{\begin{bmatrix}1.0 & 0.0 & 0.0 & 0.0\\ 0.0 & 0.0 & 0.0 & 0.0\\ 0.0 & 0.0 & 0.0 & 0.0\\ 0.0 & 0.0 & 0.0 & 0.0\end{bmatrix}} \underset{P(o|s)}{\begin{bmatrix}0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}} \underset{b}{\begin{bmatrix}1.00\\ 0.00\\ 0.00\\ 0.00\end{bmatrix}} = \begin{bmatrix}0.00\\ 0.00\\ 0.00\\ 0.00\end{bmatrix} \Rightarrow \underset{b'}{\begin{bmatrix}0.25\\ 0.25\\ 0.25\\ 0.25\end{bmatrix}}$$

B(a=1)=0.50

Figure 5.5: For another advice state, the transitions are always 0. Here we walk the same P(o|s) sequence through, for the first movement at the top through the fourth movement at the bottom. Now the cumulative advice $B(a = 1) = 0.5$, as opposed to $B(a = 0) = 3.25$.

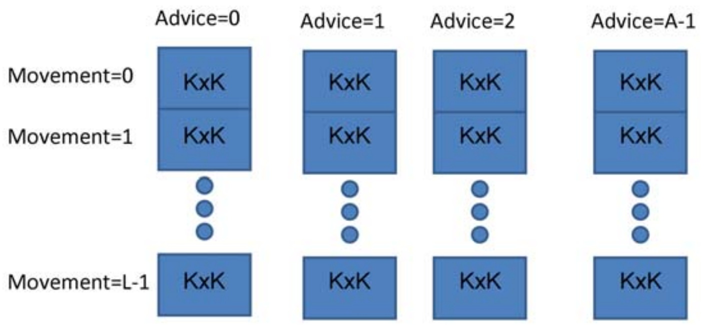


Figure 5.6: Conceptualization of state transition tables. There is one KxK size table for each movement and advice state.

5.2 Advice Generation

We now move to how the advice is actually computed. In early implementations, the advice or belief of the parent node, a, was chosen using the selection rule

$$a = \arg\max_s b_p(s) \tag{5.8}$$

In other words, the parent basically fed back the label of the "winning" centroid to the children nodes. We found that this selection rule was not robust to evolving belief conditions in the online learning process. The problem with this in practice was the static learning at the parent level tended to jump repeatedly from state to state, so that there was no clear coherence from observation to observation. In addition, the advice here is more the "hard threshold" method, which tends to work well only in the face of no noise or very clearly defined classes or clusters.

Instead we use an online advice generation rule where each parental node examines the temporal sequence of input beliefs and performs unsupervised online clustering on the concatenation of children belief states across all the movements, up to the final movement. Thus the advice is generated in fairly large temporal "chunks", albeit with an online system. The resulting label is then passed to the child nodes at the end of the movement observation sequence. The children interpret the advice as a sort of passive learning mechanism in that the advice is used to train the dynamic patterns. Thus instead of a set of A tables, we now have a set of LA tables where L is the number of movements in the observation sequence. The tabulated table is still simply a count of the probability of transitioning from cluster label from s to $s\prime$ given a, except it is tabulated on a temporal basis using the "long time scale" advice provided by the parent during the training phase. We note that there are some similarities here with the temporal pooling method of George (2008).

This is a more temporally coherent advice generation method which looks across many different observations to determine the parental advice, which is an assessment of the general state of the children's beliefs (and benefits from a viewpoint one layer up in the hierarchy). This method is still a "hard threshold", but when the actual responses are delivered, the children do not actually use parental advice; instead, the cycle through all possible advice states, and deliver a belief in advice vector denoted as $B(a)$. One interpretation of this process is that the parents teach the children what situations to "look out for", but then during the response process the children must rely on the parental teachings to actually generate beliefs. We consider this "passive advice" as a positive step toward true incorporation of an evolving belief with confidence levels that can potentially offer a level of control over the observation process.

5.3 Function Approximation

The tabular representation is ill-suited for scaling as it imposes an $O(N^3)$ memory complexity. Even without the temporal dependence discussed above, the $P^a_{ss'}$ tables are still rather large, with AS^2 units required. Furthermore, the temporal observations add a further dependence on the number of movements which are used to formulate the advice. Thus the memory requirements of this scheme using a tabular approach are quite formidable, computed for a single node as

$$M_{tab} = K^2 AL, \tag{5.9}$$

where K is the number of centroids for the node, L is the number of movements or sequence length, and A is the number of advice states from the parental node.

As an alternative to this approach, we seek to create a function approximation method, where the transition probability is estimated by a learned function. This is performed by simply estimating the subsequent state given the current state and advice. In practice this means a very short memory mechanism must be used to train the function approximator, but this is easily achievable with trivial constraints. Furthermore, the advice states and movements can be represented by sparse binary input vectors in the case where the advice data is interpreted as a "hard label". Thus the function approximator must estimate the next state s' given input advice a, movement number m, and the previous state s. There are thus $A + K + L$ inputs, with K outputs. This is best achieved with a nonlinear function approximator such as a neural network Hornik Maxwell and White (1989). The essential idea is depicted in diagram form in Figure 5.7. The memory required to implement the neural network varies with the number of layers and hidden nodes chosen. However, if we assume a single layer with H nodes can achieve the desired result, we can define the memory constraints for the implementation. There are $A + K + L$ inputs with each linked to H hidden nodes, with a weight for each connection. At each hidden node an additive bias is used, then there are K outputs with connections from each hidden node to the output, with another additive bias per output. Thus we can express the number of memory elements as

$$M_{FA} = K + H\left[1 + 2K + A + L\right], \tag{5.10}$$

which is linear in dimension K, A, and L and greatly reduces the memory load, assuming H is not a high-order function of K This should result in substantial memory savings.

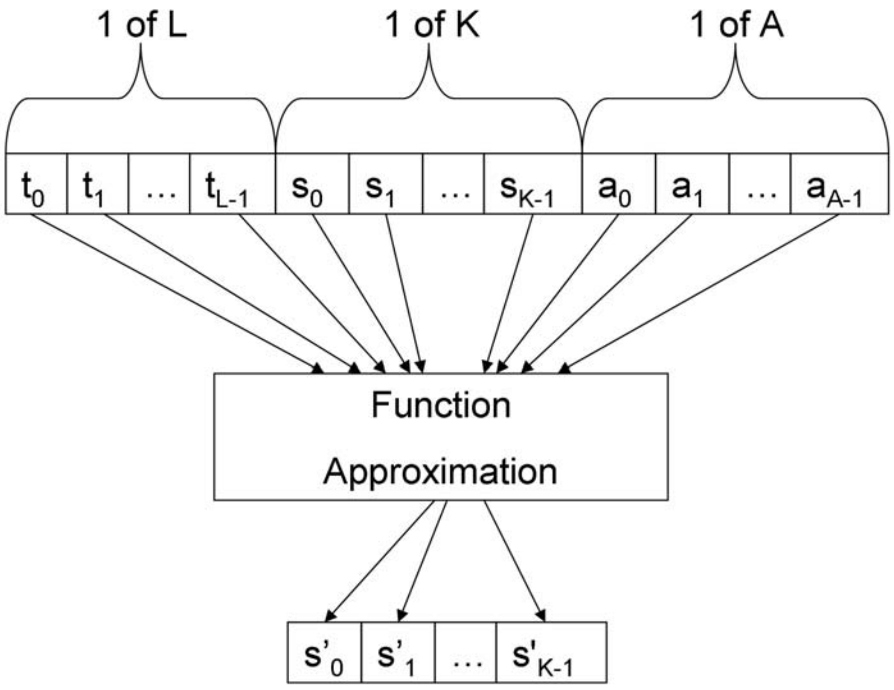


Figure 5.7: Function approximation for state transition tables, mapping $L + K + A$ vector to K outputs to emulated "table".

5.4 Node Selection for Supervised Learning

We discuss the process of node selection for supervised learning in this section. Generally, in pattern recognition the feature selection problem has been address through a variety of methods in supervised and unsupervised learning, including Guyon and Elisseeff (2003), Dy (2008), and Mitra et al. (2002). For DeSTIN our objective was to leave the processing unsupervised through the entire process except the final supervised classifier. Thus, we investigated a few different concepts for node selection. Each node can output either the beliefs as a time-domain signal or

Table 5.1: Number of Unique Vectors in first layer per Row,Column for Cluster Metric Study

	0	1	2	3	4	5	6	7
0	1	46	390	1351	2218	1567	431	29
1	120	2145	11680	29383	41219	32262	12306	1687
2	636	9417	40725	81295	98904	79600	34484	5412
3	982	15821	59799	99964	111033	90396	39300	5818
4	925	18497	63472	99299	109267	86724	35705	5068
5	1205	19782	63053	98565	106894	77893	29735	4255
6	857	13428	47881	82299	83252	49171	14828	1794
7	159	3297	15408	29495	27334	12642	2733	275

the "belief in advice" $B(A)$ values as a single composite output across the entire observation time. For our earlier work, we used a single selection method which we dub "hand selected", which we describe next. We also enumerate methods which we call "topographical", where we used nodes solely based on the DeSTIN topology (layer and node number). Finally, we used specific node selection methods which were based on principle component analysis or (PCA).

5.4.1 Hand picked node selection

The nodes selected for temporal sampling processing mentioned previously are refered to as the "hand picked" nodes. Because this level of work involved a very high feature dimension, where we sampled every T movement from each node, we tried to reduce the numbers of nodes actually used. We used all the top layer nodes, from layer 1 and up, for a total of $16 + 4 + 1 = 21$ nodes. For the bottom layer, we examined the number of unique vectors seen on the input through an search sampling every 25th MNIST digit in the training set. The number of unique vectors is shown in Table 5.1. We clearly omit nodes with less activity, essentially choosing none from the left side of the grid. Most of the nodes in the 3rd and 4th column are chosen, then a random sampling are selected in the final two columns. Thus, the "hand picked" nodes are the following. They consist of all nodes of layers 3, 2, and 1 were used and 18 selected nodes of layer 0 were used. These nodes are (2,6), (3,6), (5,7), (6,6), (7,3), (0,3), (1,4), (2,3), (2,4), (3,3), (3,4), (4,3), (4,4), (5,3), (5,4), (6,3), (6,4), (7,4). They are also shown in Table 5.2.

5.4.2 Topological methods

For these methods we rely on the actual spatial and layer placement of nodes. There are four main methods for performing these node selections and they simply consist

Table 5.2: Nodes used for layer 0

	0	1	2	3	4	5	6	7
0	N	N	N	Y	N	N	N	N
1	N	N	N	N	Y	N	N	N
2	N	N	N	Y	Y	N	Y	N
3	N	N	N	Y	Y	N	Y	N
4	N	N	N	Y	Y	N	N	N
5	N	N	N	Y	Y	N	N	Y
6	N	N	N	Y	Y	N	Y	N
7	N	N	N	Y	Y	N	N	N

Table 5.3: Topographical Node Selection Methods

	N	D
L3	1	25
L23	5	125
L123	21	525
L123Core	30	750
All	85	2125

of choosing each group of layers in combination. We used the top only node, denoted L3; the top two layers denoted L23; the top three, L123; and finally a mode we call L123Core, which uses the top three layers and the 9 inner most nodes of the bottom layer. These were chosen as they are the central nodes in the bottom layer and thus should see as much variation as the outer edges given our movement scans. In Table 5.3 we show the number of nodes for each method, along with the option of using all nodes together (which was not explored). The example assumes we use 25 belief or advice states per node with the $B(a)$ composite observer output.

5.4.3 PCA methods

The PCA methods consist of two main methods. The first, called "PCAAll", simply uses all nodes and performs PCA analysis, keeping some fraction of the total variation (generally 90%) and attempts to remove correlated signals and instead represent them by weightings of an orthogonal basis. The second, called "PCAByNode", performs PCA on each node alone then ranks the nodes according the number of dimensions needed to achieve the specified level of error. The node (or nodes in case of ties) with the most dimensions is chosen first, then the next node, etc. Note, however, that the

PCA projections are not used here, instead we use the original data. This attempts to minimize faults of PCA such as cases where signals which can be discriminatory are nevertheless somewhat correlated. In DeSTIN, we know there is a large degree of correlation between nodes, based on their common parentage (or child nodes), and spatial proximity. Thus the PCAByNode method is an attempt to use PCA to assess variation in a node without limiting ourselves to the attempted projection.

5.5 Summary

In this chapter we further developed two aspects of DeSTIN: the dynamic learning processing, and the node selection process for supervised learning. For the former, we showed how the $P^a_{ss'}$ construct can be used to generate a temporal advice state which is learned through the beliefs created by parent nodes through unsupervised learning over long time scales. As a related processing function, we discussed the advice generation process. Since these methods require considerable memory using tabular implementations, we also discussed methods for using function approximation to create these same functions.

For the node selection process, we discussed three main methods we called hand-selected, topological, and PCA-based. None require advance knowledge of class labels, and the PCA-based can be performed online as well if necessary. These methods will be demonstrated in the results shown in Chapter 6.

Chapter 6

DeSTIN Benchmark Results

In this chapter we apply the DeSTIN architecture and methods to three different image application domains. We first discuss the application of DeSTIN to the classification of handwriting digits and explore a variety of parameters and configurations of DeSTIN, including temporal sampling, the effect of different parameters, and other experiments. We next cover the application of DeSTIN to a face detection problem, and explore different automatically selected collections of nodes. Finally, we apply DeSTIN to a third application domain, the detection of optic nerves in retina images, using a single automatic node selection method.

6.1 Handwriting Digits

During the course of the DeSTIN development, we performed initial experiments using a very simple, three-character training set using alphabetical characters "A", "B", and "C". The results of these experiments have appeared in Arel et al. (2009a) and Arel et al. (2009b), but the toy-like nature of these studies (the set contained only one example of each letter) was designed primarily to demonstrate the DeSTIN concept and viability. The results in Karnowski et al. (2010a) represent a true step beyond such "toy" problems, as these used the MNIST data set. The MNIST data set LeCun and Cortes (2009) is a collection of handwritten digits and has been used extensively in many machine learning algorithms and papers. The dataset has 60,000 training images of digits 0-9 and 10,000 testing images. All images have been roughly centered and are gray-scale. A sampling of 10 examples of each digit is shown in Figure 6.1. This dataset is challenging in the sense that good results can be obtained by very simple machine learning algorithms such as the k-Nearest Neighbor algorithm which obtains 94.6% in our testing (Figure 6.2). The best performance of reported machine learning methods Labusch et al. (2008) achieve over 99% accuracy. A fusion of several different methods and comparison with humans reveals the best possible performance is likely 99.8% Keysers (2007), while LeCun and Cortes (2009) contains a summary of different performance achievements.

For our handwriting analysis, we discuss a variety of experiment using DeSTIN and the MNIST dataset. We begin by reviewing the experimental configuration for DeSTIN, then we discuss some non-DeSTIN results on the image database as a baseline. Next, we discuss the results of DeSTIN with a temporal sampling (omitting the system dynamics) and the use of the fuzzy-distance based estimate of $P(o|s)$. Some discussion of the fixed clustering methods follow as well. We then cover our work with the different probability models, the use of the multiple observer model for $B(a)$ both with tabular and function approximation methods, and then we discuss our newest findings, the use of the ECM algorithm and automatic node selection methods. Finally, we examine the effect of different numbers of maximum clusters for both each node's static representation and the advice states.

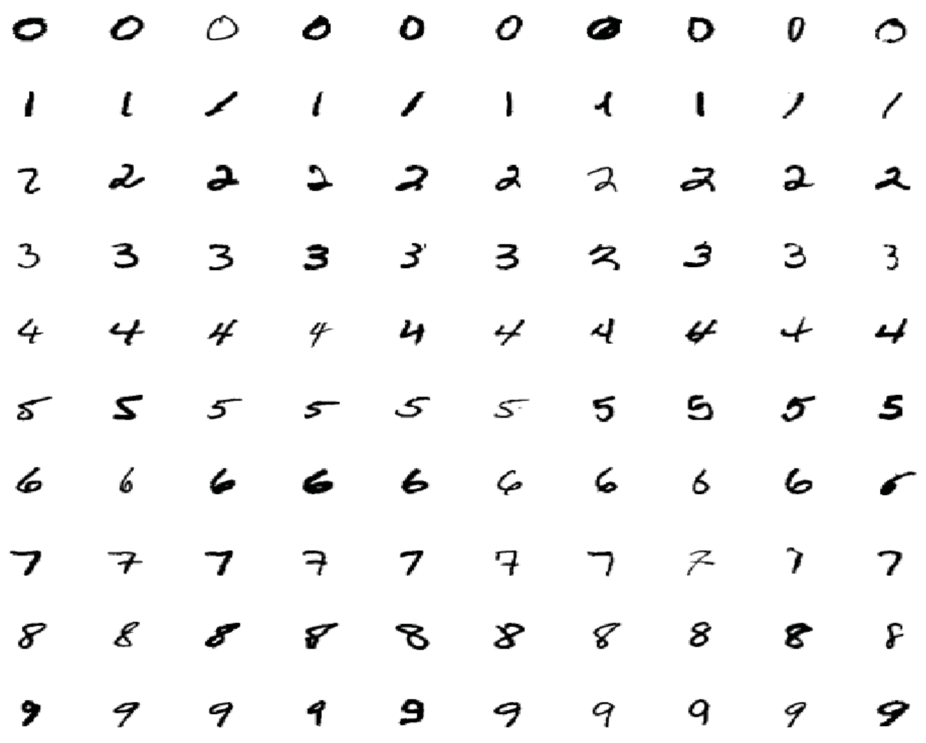


Figure 6.1: Example images from the MNIST training data set.

6.1.1 Experimental Implementations

The experimental architecture topology here is identical to that in Karnowski et al. (2010b). The MNIST dataset images were padded from 28 x 28 pixels in size to 32 x 32 pixels. A hierarchy of 4 layers of sizes 8 x 8 nodes, 4 x 4 nodes, 2 x 2 nodes, and a single node at the top layer was used as depicted in Figure 3.1 At the lowest layer each input node is presented a 4 x 4 pixel region of the input image.

We used two main centroid configurations. In the first, each layer uses a different number of centroids, choosing 25, 16, 12, and 10 for each layer. This was motivated by a study where we took the first layer, middle node (3,3) of the DeSTIN hierarchy and stepped through the training set of data. We then performed a batch clustering using k-means, with a variety of different choices for K. The gain in the mean-square-error reached a "knee" at roughly 25 centroids, so we chose this value for the lowest layer. For the top layer, since we had 10 total classes, we chose 10 centroids (although we should note that there is no optimization strategy, beyond the unsupervised clustering, that would orient or drive the learning toward a 10-class solution). For the intermediate layers we simply reduce the number of centroids by roughly $\frac{2}{3}$. In the second unsupervised learning configuration, we used the ECM method with a maximum of 25 clusters per node.

Regardless of the method, an image presentation is made by taking an MNIST image, and shifting through the sequence of 64 different movements which are offset by a single pixel and form a serpentine pattern as shown in Figure 3.2. The movement ranged from (0,0) pixels to (7,7) total pixels, so the input image was padded to cover boundary regions. The movement pattern was not optimized for this problem, and may not be the best sequence for a complete online system that iteratively derives a best belief for the input image but served as a good case for initial study.

After a DeSTIN network was trained, features were extracted by choosing particular nodes according to some methodology as discussed in Chapter 5. The response of the DeSTIN network was saved and presented to a supervised learning system, in our case a neural network implemented with the MATLAB Neural Network toolbox. In all experiments the output was standardized to zero-mean, unitary standard deviation (omitting features with a standard deviation of 0). The implemented neural network used two hidden layers of 40 nodes each and was trained by using the training set split into a true training set (using 70% of the input images) and a validation set of 30% to prevent overtraining. Ten different network training sequences were used and the results were averaged together.

6.1.2 Basic image dataset testing

The MNIST data set has been used many times in the past as mentioned and is therefore a well-tested dataset. We verified the performance of the kNN algorithm on the MNIST data set, by simply performing the kNN testing across multiple values of k (from 1 to 130). These results are shown in Figure 6.2, where we see that the best performance is with $K = 3$ at 94.6% correct or 5.4% error.

As another experiment, the neural network configuration was used on the raw MNIST data set as well. For one experiment each pixel was normalized by the subtracting the mean and normalizing to zero standard deviation based on the training data. A second experiment was conducted without normalization. We expect this to be a better test of the feature extraction nature of DeSTIN, since the supervised learning method is identical. We achieved composite performance of 96.19% correct on the unnormalized case, with 97.03% correct on the normalized case. These results are summarized in Table 6.1.

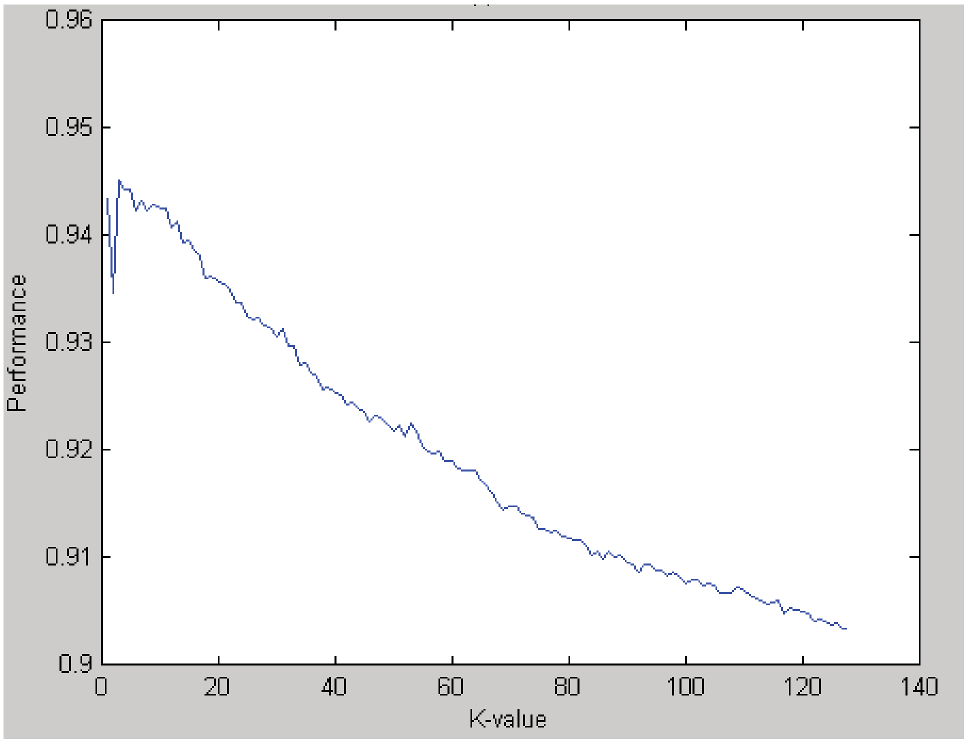


Figure 6.2: Results of KNN processing on MNIST image sets. Best performance was 94.6% at K=3.

Table 6.1: Results of neural network ensemble on raw MNIST image data

Type	Norm	AllPerf	BestPerf	MedPerf
RawImage	No	96.19	96.35	85.86
RawImage	Yes	97.03	97.21	97.05

6.1.3 Temporal sampling experiments

As a starting point, early DeSTIN experiments were performed using the fuzzy-distance based construct given by Equation 4.1. Another "simplification" was the use of temporally sampled data instead of formulating the $P^a_{ss'}$ computation through the evolving $B(a)$ structure. This was done to simplify the initial development, but it caused some issues with very high feature dimensionality. Note that the extracted features are from the clustering performed at different layers in the hierarchy on the beliefs computed by Equation 4.1, and there is still a level of temporal inferencing which is invoked by the use of sampling the temporal outputs of each relevant node which is fed to the neural network in the supervised learning phase. Finally, in these experiments the "hand selected" method of Chapter 5 was used.

Cluster analysis

We examined the clustering characteristics for these tests, noting that the clustering metrics described earlier were simplified greatly. A constant learning rate was used and only the starvation trace mechanism was included. The centroid clustering metrics (μ, σ, and ρ) were retained for monitoring purposes but are not directly tied to the learning rate. The problem with using all the mechanisms is that we had difficulty forcing the clustering to a stable point; generally in online clustering convergence is not guaranteed unless the learning rate monotonically decreases. However, we did use some clustering monitoring based on the earlier described methods to modulate the clustering, albeit in a largely heuristic manner.

The first experiments used a sampling of the MNIST training set (every 25th image) and examined the effect of learning rate on the values of μ and ρ for layers 0 and 3. For layer 0, node (3,4) is reviewed as this node sees the most variation from the input sequence (see Table 5.1). The top layer is chosen also as it gives the highest "overview" of the entire processed sequence. These are plotted in Figures 6.3 and 6.4 with the observation number on the x-axis (where there are 64 movements or observations per input MNIST digit) for learning rates of 0.001 and 0.0001. These plots show that little is gained in the sense of the clustering stability after roughly 6000 observations for the first layer. However the final layer shows that the mean change increases for some centroids, indicating that they are not in a stable position but the change relative to the standard deviation indicates the centroid learning may be reaching a reasonable bounding value. For the slower learning rate, the value of ρ shows that the learning takes longer, as expected, since the value of the largest entry does not reach a comparable level to the faster rate until around 9000 observations. The value of μ at the top layer shows more erratic behavior, reaching a plateau around 6000 then increasing throughout the rest of the sequences before decreasing again after about 10000 observations. This is likely due to the stabilization of most of the lower level nodes around 6000 which causes the highest layer to settle a bit, but later observations cause additional changes that are not as well matched and thus the centroids drift again.

An online error was generated by computing the difference between each input vector and the adjusted, winning centroid. These plots are shown in Figure 6.5 for layers 0 and 3 again. A smoothing window is applied to the plots of size 64. For layer 0, we see that the error decreases fairly rapidly to a roughly constant level. For the top layer, we see that a sort of minimum error is reached rather early in the sequence but the smoothed error increases to a significantly larger amount than layer 0. The minimum error is likely where the online clustering has reached a good match relative

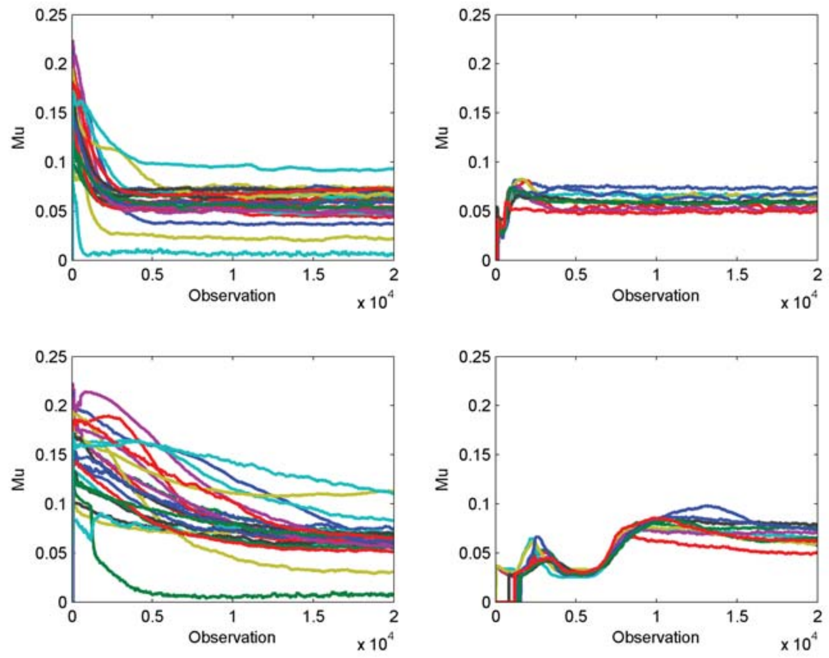


Figure 6.3: Top: μ for learning rate of 0.001. Left is bottom layer, right is top layer. Bottom: μ for learning rate of 0.0001. Left is bottom layer, right is top layer..

to the immature response of the lower layers. The higher upper level error can be partially explained from the smaller number of centroids (10 instead of 25) at this layer. With the lower learning rate, we see a similar behavior for layer 0 although the error decline is slower. The top layer in the slower learning rate case shows an increase in the error after reaching a sort of plateau as seen in plots of μ as well. The error then begins to decline gradually.

In another experiment the learning rate was monitored and heuristically modified to automatically terminate clustering. In this schema, we evaluated the innermost nodes of the initial layer and all nodes of subsequent layers. (As shown in Table 5.1, the edge nodes of layer 0 do not show as much variation.) The learning rate was initialized to 0.001 for layer 0 and 0.0001 for subsequent layers. The mean value of ρ across all centroids was computed at each observation. When the mean value was less than 0.05, clustering terminates for the node. When half the nodes were terminated the entire layer clustering was stopped. The learning rate for the next layer was then reduced to 0.001 and an additional 1000 non-monitored digit presentations were performed followed by renewed monitoring of the value of ρ. This process was repeated until the top layer clustering was terminated. The clustering termination point is shown in Tables 6.2 and 6.3 for each node. Note that "N" denotes a node that either was not included or did not finish clustering before half the candidate notes completed. The top layer stopped at 14050 digits and layer 2 stopped at 10984 and

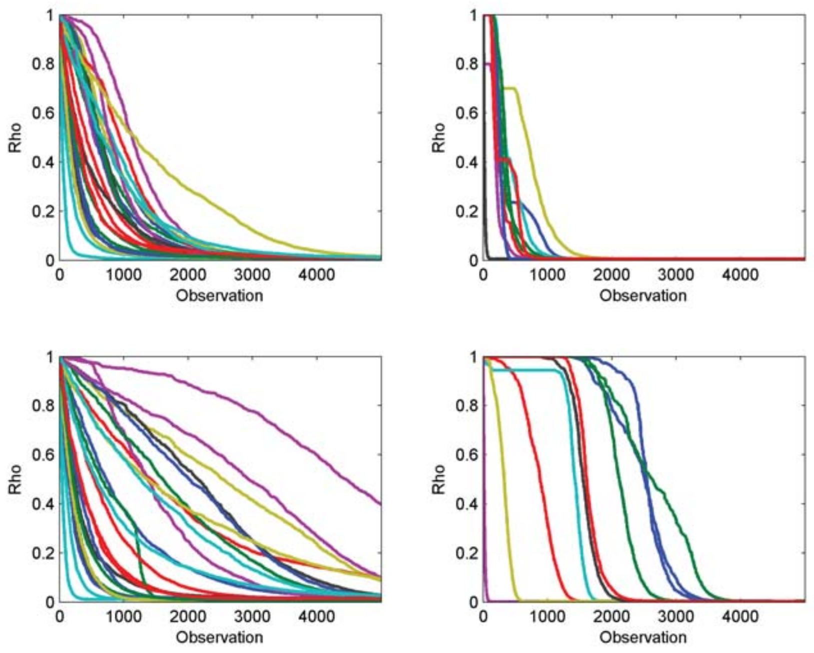


Figure 6.4: Top: ρ for learning rate of 0.001. Left is bottom layer, right is top layer. Bottom: ρ for learning rate of 0.0001. Left is bottom layer, right is top layer.

11043 digits (nodes (0,1) and (1,0)). The resulting error plots are shown in Figure 6.6. The first layer response is as expected, with the error dropping quickly to a fairly constant value. The second layer response is more complex as the error increases initially then slowly drops, settling out around digit presentation 5000. In fact, the clustering of layer 0 stops adapting after digit presentation 4808, so layer 1 adapts slowly with the slow learning rate, then the increase in the learning rate does not seem to change the response much. Layer 1 has its clustering stopped at presentation 8400 and we see that shortly afterward layer 2 reaches a relative constant value. However, the first 4000 or so presentations to layer 2 have a very low error rate. In this case, analysis of the data showed that the response was dominated by a single cluster which closely matched the output of layer 1, but was somewhat meaningless because layer 1 had not begin to adapt. Once layer 1 adapts, we see that the error of layer 2 begins to increase to a peak around presentation 4000 and then it declines as layer 1 stabilizes. A similar phenomena is shown for the top layer, although its adaptation is not as pronounced after approximately presentation 12000.

It is instructive here to observe the cluster centroids formed for some nodes to illustrate the method. In Figure 6.7 we show the centroids formed at one of the inner nodes of the initial layer. The cosine distance metric was used and so consequently the visualization is imperfect, but we clearly see that the centroids are converging to structures that resemble components of handwritten digits such as corners and lines.

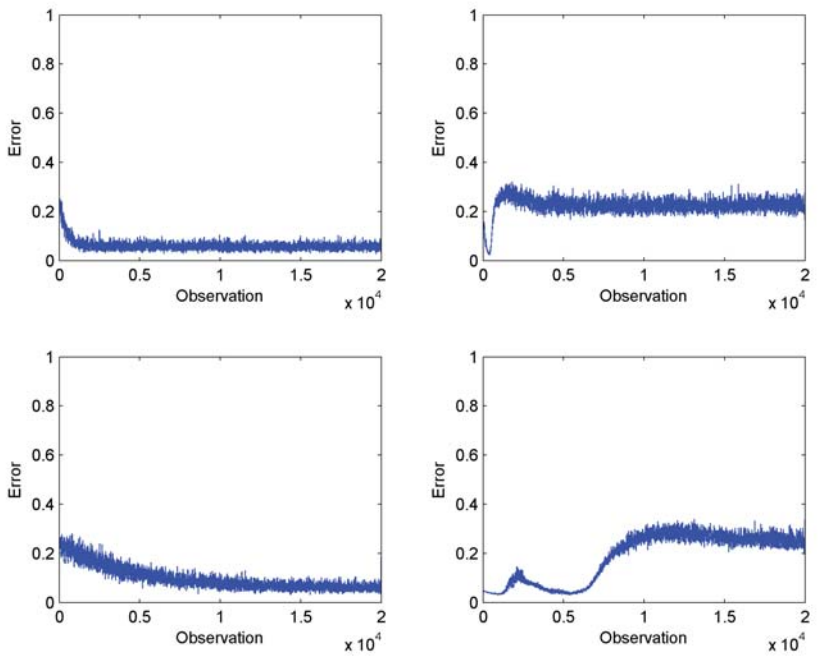


Figure 6.5: Top: error with learning rate of 0.001, with (left) layer 0 and (right) layer 3. Bottom: error with learning rate of 0.0001, with (left) layer 0 and (right) layer 3.

For example, the top right centroid shows a bright diagonal edge. The sample beneath that shows a similar edge, but translated toward the upper left, and the one below that shows yet another diagonal but not translated as much. The second column, first row shows a corner point, and this is also seen in the second and third column of the third row. There are also vertical edges (such as second row, third column) and horizontal edges (top row, third and forth columns for example). One may note that the clusters do not show invariance, but that is perfectly reasonable and even sensible; the overall group achieves invariance by multiple examples. Higher nodes are more difficult to visualize since they are clustering on beliefs and thus do not have the same spatial meaning or context that the centroids of the lowest layer offer.

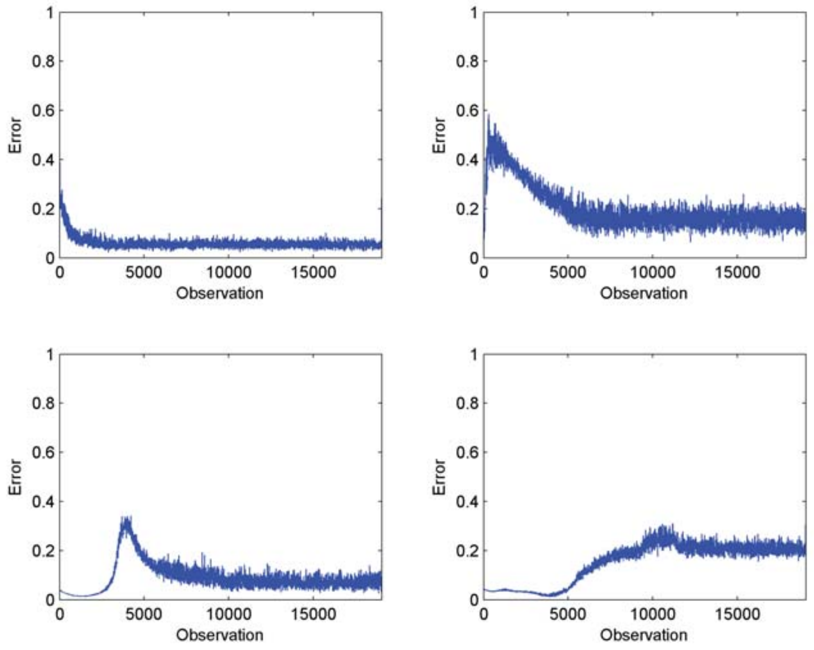


Figure 6.6: Error rates for node (3,4) of layer 0, node (1,2) of layer 1, (0,1) of layer 2, and (0,0) of layer 3 using the adaptive learning rate and automatic termination of clustering.

Table 6.2: Cluster stopping points for nodes of layer 0

	0	1	2	3	4	5	6	7
0	N	N	N	N	N	N	N	N
1	N	N	N	N	N	N	N	N
2	N	N	N	3002	2511	2985	N	N
3	N	N	3883	2598	2423	2739	N	N
4	N	N	4808	2715	2383	2980	N	N
5	N	N	3554	2616	2418	3393	N	N
6	N	N	4663	2949	3203	N	N	N
7	N	N	N	N	N	N	N	N

Table 6.3: Cluster stopping points for nodes of layer 1

	0	1	2	3
0	N	N	9337	N
1	N	6712	6647	8400
2	N	6751	6642	N
3	N	7092	6898	N

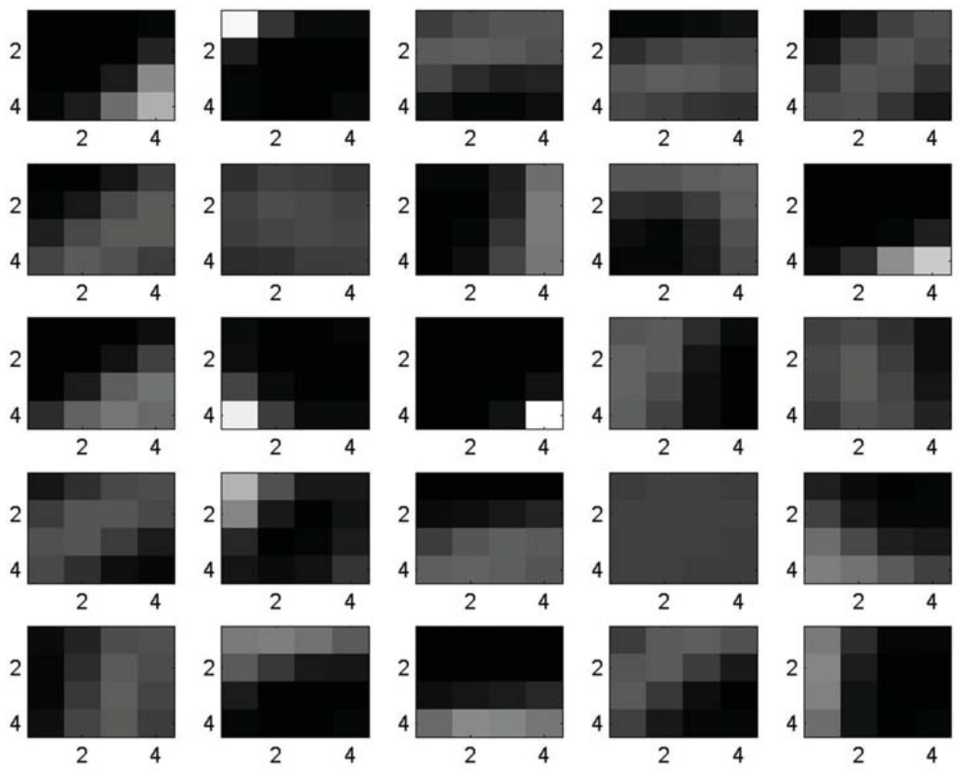


Figure 6.7: Visualization of cluster centroids for bottom layer of DeSTIN hierarchy with MNIST data set, for node (3,3).

Supervised learning

The results of the supervised learning experiments are shown here. Four different sets of experiments were performed. These all use the same configuration of unsupervised clusters at the layers of the hierarchy, the 25-16-12-10 model, with the "hand selected" nodes.

Case 1 represents a fixed number of iterations (20,000) along with sampling the temporal output every 16 movements for 4 samples per digit. Case 2 is also fixed iteration but the temporal output is sampled every 12 movements for a total of 6 samples per digit sequence. In Case 3 we use the adaptive number of iterations and also sample every 12 movements.

Table 6.4: Experiment overview

	T	Iter	D	AllPerf	BestPerf	MedPerfSingle
Case 1	12	20000	4584	97.7	97.82	96.89
Case 2	16	20000	3056	97.43	97.59	96.51
Case 3	12	14050	4584	97.86	97.98	96.94

Table 6.5: Confusion matrix for MNIST data set for best performance of case 3

	0	1	2	3	4	5	6	7	8	9	Perf
0	975	1	0	0	0	0	2	1	1	0	99.49%
1	0	1126	1	1	0	1	2	1	3	0	99.21%
2	5	2	1011	4	1	0	0	8	1	0	97.96%
3	0	0	4	988	0	4	0	4	10	0	97.82%
4	0	0	1	0	963	0	4	1	2	11	98.06%
5	2	0	1	4	0	873	4	2	4	2	97.87%
6	3	2	1	0	1	8	939	0	4	0	98.02%
7	1	3	11	2	0	0	0	1004	2	5	97.66%
8	3	0	4	6	4	4	1	5	941	6	96.61%
9	2	3	1	4	6	2	0	4	9	978	96.93%

The best resulting composite performance was 97.98% accuracy (2.02% error) as shown in Table 6.5. In this table, the sampling period is labeled as column T. The next column, labeled Iter, is the number of iterations performed which is fixed at 20,000 for cases 1 and 2 and was automatically terminated at 14050 for case 3. The data dimensionality is column D. The AllPerf column identifies the performance of all the neural networks combined, while the BestPerf column indicates the best possible performance obtained by selectively using some neural networks in an exhaustive search. Finally the MedPerfSingle columns is the median performance of all the neural networks, which is significantly lower for the raw image case. Overall we note that our experimental results are significantly better than a basic kNN classifier applied to the image data, but are not at the state-of-the-art for this data set. Furthermore, running the same neural network configuration and voting scheme on the raw image pixels produced a best performance of 96.35% or composite performance of 96.19% so we are confident DeSTIN is serving as a beneficial feature extractor.

Mixture Models Comparison

In this experiment, we compared the use of the "true" $\Pr(o|s')$ estimate afforded by the distance-based estimate of 4.1 by that from the Gaussian, Rayleigh and Exponential probability distribution models. In this experiment we used sampled temporal beliefs only, using every 8th movement. Four different trials were performed with each model including network training and supervised learning. These are summarized in Table 6.6.

Table 6.6: Performance with distribution models for P(o|s)

Model	Trial1	Trial2	Trial3	Trial4
Exponential	98.54	98.49	98.48	98.42
Rayleigh	98.15	98.41	98.56	98.20
Gaussian	98.15	98.32	98.24	98.18

The exponential model seems to offer better performance overall, given the relatively equal running conditions, but all offer improvements over the simplistic distance-based estimate; as a comparison, the performance we achieved in Karnowski et al. (2010b) with the supervised learning on the extracted DeSTIN features was 97.98% accuracy (2.02% error). Thus we see that the probabilistic models offer a performance improvement, albeit with some cost in computational load (especially for the Gaussian function). The different models may require different operational conditions to more fully extract their full performance improvements, for example longer training periods may permit better estimates of the Gaussian and Rayleigh functions which will improve their performance.

Tabular and Function Approximation Comparison Experiments

For the next experiment we incorporated the multiple-observer model for $B(a)$ and compared the performance of the supervised learning system on DeSTIN features using ten different DeSTIN networks, with each one trained with both the tabular method and function approximation method for belief formulation. A total of 25 advice states were trained for each parental node, with an additional "oversight" parental node at the top layer. The function approximator was implemented using the Fast Artificial Neural Network Library Nissen (2003). For each node the neural network was implemented with a single hidden layer of 10 nodes, with a learning rate of 0.001 which was not changed through the network training. The memory comparison between the function approximator and the tabular method is shown in Table 6.7. Overall, the function approximation method requires roughly 1/600 of the memory as the tabular method, but does have a slight increase in computational cost. As in the sampled belief experiments, 39 nodal outputs were used making the output feature space 975 in size. While this is still large it is much smaller than the space of the previous experiments, and as shown in Table 6.8, the performance was essentially identical to the more complex case. Indeed, after feature normalization for the tabular method most cases had a reduce feature space of roughly 700 owing to features rejected to a standard deviation of 0. However, the function approximation method did not achieve any reduction in feature space from pathological cases of 0 standard deviation.

Table 6.7: Memory comparisons for function approximation and tabular methods

Layer	Nodes	K	Tab	F.A.
0	64	25	1000000	1425
1	16	16	409600	1236
2	4	12	230400	1152
3	1	10	160000	1110

ECM and Automatic Node Selection Comparison

After these preliminary results, as discussed previously, we return to the tabular method and tested the ECM method from Chapter 4 with the automatic node selection methods discussed in Chapter 5. We also switched from the 25-16-12-9 cluster model to the ECM method, with a maximum of 25 clusters per layer. The advice was also set to a level of 25 clusters per layer, but it was not configured as an ECM, rather as WTA. A set of 10 different experiments were run, with different random initializations of the data presented and initial cluster locations for layers above the initial or bottom layer. The bottom layer was set to the ECM method,

Table 6.8: Performance comparison using tabular and function approximation methods

Trial	Tabular	F.A.
1	98.3	98.43
2	98.27	98.33
3	98.34	98.57
4	98.42	98.38
5	98.32	98.59
6	98.51	98.45
7	98.47	98.42
8	98.45	98.43
9	98.35	98.53
10	98.47	98.41

while layers above this were set to ECM after an initialization period of "cluster pooling" as described in 4. The learning rate for the ECM method was the k-means $\frac{1}{N}$ method. The learning rate for the upper layers was set to a constant $\eta = 0.0001$, with the starvation trace construct used as well, until the layer beneath it stabilized, at which point it adopted the ECM method with the $\frac{1}{N}$ method. Clustering was stopped when either the monotonicity constraint was reached, or half the nodes had a ρ value below the threshold of 0.05. Interestingly, the monotonicity constraint was the only factor for termination, and for the upper layers it always occurred within the window of the forced 1000 initial digit presentations. The exponential model alone was used for these experiments as well, with the λ parameter estimated based on the running average of the sample distances from the winning centroid. Upon the completion of the clustering, exponential parameter estimation, and $P^a_{ss'}$ tabular learning, the response for each network type was created for the testing and training set, then the supervised learning phase was performed as described earlier.

Since there are many "sets" of automatic node selections per experiments, the best way to present these results is in terms of a scatter plot with the dimensionality shown. The hand selected, PCAAll, PCAByNode, L123, L123Core, L3 and L23 are presented as separate symbols at the proper dimension. The best results obtained in this work used the L123Core configuration, which used all the nodes of layers 1, 2 and 3 as well as the "center" 4×4 nodes from the bottom layer, which achieved a maximum performance across 10 different trials of evolving DeSTIN networks of 98.78% correct or an error rate of 1.22 %. However, many of the different network choices achieved very similar performance. If pressed to simply choose a configuration, the Auto selection methods above a dimensionality of 300 are a reasonable choice, as they seem to outperform most methods of comparable dimension (for example, PCA reduced the dimension to roughly 400 with a performance of 98.18%, but the auto processing of similar dimension scored 98.50%).

We also note that the algorithms tended to use the maximum number of allocated clusters for each node. This was slightly surprising, but we believe the experiments and discussion from Chapter 4 showed why this would be the case. Some other means of constraining the clustering to a different number of clusters than the ECM method of the minimum distance could produce different effects, but overall we believe the system has proven itself robust.

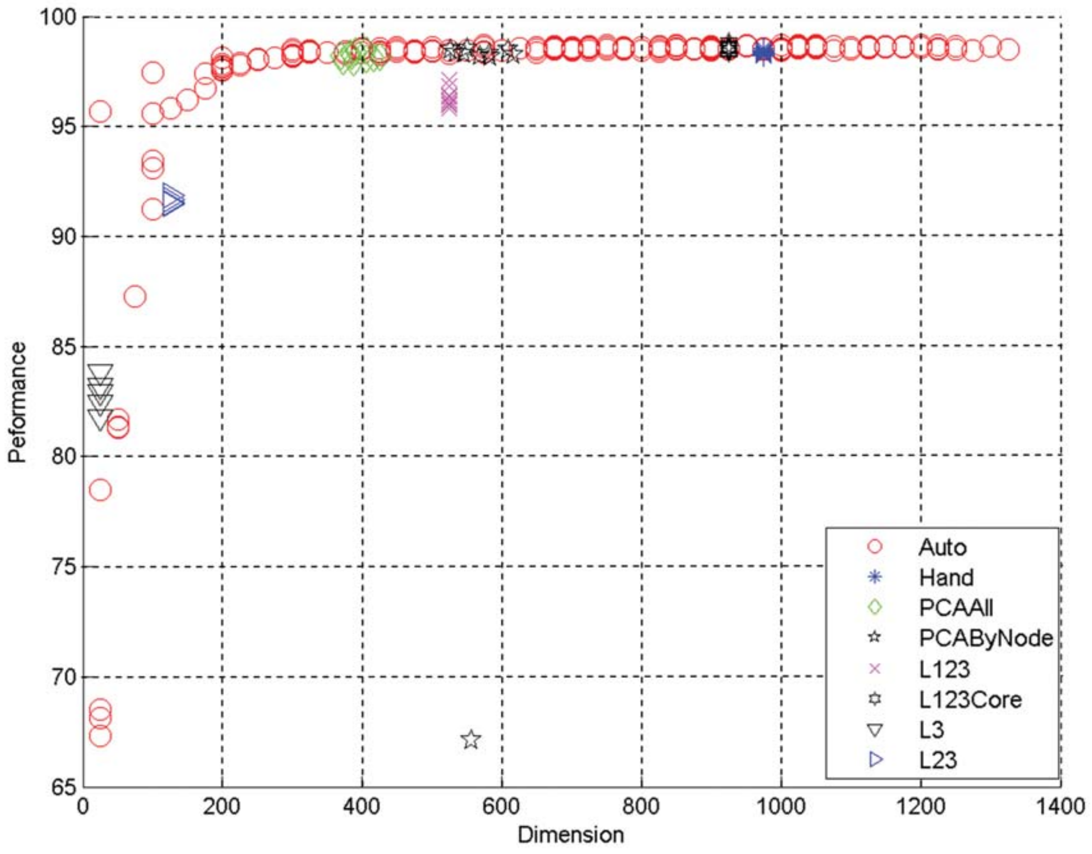


Figure 6.8: Performance of DeSTIN on the MNIST set using different node selection methods.

Table 6.9: Performance comparison example between supervised learning methods with auto selected nodes

Nodes	Number	D	kNN	NN
Auto	1	25	63.13	67.14
Auto	2	100	92.83	95.7
Auto	3	200	95.49	97.44
Auto	4	275	96.81	98.1
Auto	5	300	96.84	98.1
Auto	6	400	97.26	98.5
Auto	7	450	97.41	98.51
Auto	8	500	97.38	98.6
Auto	9	575	97.28	98.61
Auto	10	650	97.08	98.68
Auto	11	675	97.08	98.57
Auto	12	700	97.03	98.62
Auto	13	725	97.05	98.47
Auto	14	850	96.94	98.53
Auto	15	875	96.9	98.67
Auto	16	1050	96.62	98.51
Auto	17	1125	96.59	98.62
Auto	18	1250	96.44	98.62
PCA	NA	390	92.94	98.16
PCAByNode	NA	556	96.13	98.41
Hand	NA	975	96.57	98.26
L3	NA	25	77.15	82.93
L23	NA	125	86.52	91.51
L123	na	525	91.50	96.39
L123Core	na	925	96.83	98.66

Table 6.10: Performance comparison example between supervised learning methods with auto selected nodes

Method	Dim	min	max	mean	median	std
Hand	975	98.21	98.54	98.36	98.38	0.09
PCAAll	394	97.94	98.46	98.18	98.18	0.17
PCAByNode	568	98.16	98.53	98.35	98.30	0.13
L123	525	95.82	97.13	96.30	96.20	0.40
L123Core	925	98.39	98.78	98.56	98.56	0.11
L3	25	81.79	83.84	82.85	82.93	0.78
L23	125	91.47	91.88	91.63	91.65	0.16
Auto	25	67.31	95.70	75.62	68.53	12.11
Auto	50	81.31	81.71	81.45	81.34	0.22
Auto	75	87.29	87.29	87.29	87.29	0.00
Auto	100	91.26	97.44	94.16	93.44	2.39
Auto	125	95.80	95.80	95.80	95.80	0.00
Auto	150	96.22	96.22	96.22	96.22	0.00
Auto	175	96.72	97.41	97.06	97.06	0.49
Auto	200	97.54	98.10	97.74	97.70	0.21
Auto	225	97.79	97.88	97.82	97.80	0.05
Auto	250	98.03	98.08	98.06	98.06	0.04
Auto	275	98.10	98.10	98.10	98.10	0.00
Auto	300	98.14	98.50	98.29	98.28	0.12
Auto	325	98.34	98.45	98.38	98.37	0.05
Auto	350	98.37	98.37	98.37	98.37	0.00
Auto	375	98.32	98.41	98.36	98.36	0.05
Auto	400	98.37	98.53	98.48	98.50	0.07
Auto	425	98.28	98.52	98.38	98.35	0.09
Auto	450	98.38	98.60	98.50	98.51	0.11
Auto	475	98.35	98.45	98.40	98.41	0.05
Auto	500	98.42	98.61	98.50	98.46	0.10

Table 6.11: Performance comparison example between supervised learning methods with auto selected nodes (Continued)

Method	Dim	min	max	mean	median	std
Auto	525	98.29	98.46	98.38	98.38	0.12
Auto	550	98.45	98.47	98.46	98.46	0.01
Auto	575	98.34	98.68	98.51	98.51	0.16
Auto	600	98.44	98.44	98.44	98.44	0.00
Auto	625	98.54	98.54	98.54	98.54	0.00
Auto	650	98.35	98.59	98.50	98.53	0.11
Auto	675	98.50	98.62	98.56	98.56	0.06
Auto	700	98.47	98.58	98.51	98.48	0.05
Auto	725	98.44	98.62	98.54	98.55	0.07
Auto	750	98.50	98.71	98.58	98.55	0.10
Auto	775	98.50	98.57	98.55	98.56	0.03
Auto	800	98.44	98.59	98.49	98.44	0.09
Auto	825	98.38	98.63	98.55	98.61	0.11
Auto	850	98.48	98.67	98.57	98.57	0.08
Auto	875	98.51	98.53	98.52	98.52	0.01
Auto	900	98.44	98.62	98.53	98.52	0.06
Auto	925	98.48	98.56	98.51	98.50	0.04
Auto	950	98.62	98.68	98.65	98.65	0.04
Auto	975	98.50	98.54	98.52	98.52	0.03
Auto	1000	98.45	98.69	98.54	98.51	0.10
Auto	1025	98.53	98.70	98.62	98.63	0.07
Auto	1050	98.56	98.68	98.63	98.62	0.05
Auto	1075	98.47	98.66	98.59	98.63	0.09
Auto	1100	98.51	98.59	98.55	98.55	0.06
Auto	1125	98.53	98.62	98.57	98.56	0.04
Auto	1150	98.57	98.65	98.61	98.61	0.05
Auto	1175	98.58	98.63	98.60	98.60	0.04
Auto	1200	98.60	98.71	98.65	98.64	0.05
Auto	1225	98.50	98.70	98.59	98.56	0.10
Auto	1250	98.51	98.62	98.58	98.62	0.06
Auto	1275	98.46	98.46	98.46	98.46	0.00
Auto	1300	98.62	98.62	98.62	98.62	0.00
Auto	1325	98.49	98.49	98.49	98.49	0.00

We show an example of a confusion matrix from these methods in Table 6.12, and compare the performance of each digit with that of the temporal sampling, case 3 in Table 6.13. The most striking item of the comparison is the improvement in discrimination of digit “9”, which rose from 96.93% to 97.03%, and “8”, which rose from 96.61% to 98.36%, and digit “3” (97.82% to 99.4%). We should note that if we were truly trying to improve performance for this particular task at hand, at this point we would “hand engineer” the system to force better recognition of the worst cases, the digits 9, 8, 7 and 2. An effort such as this would also require an additional validation set, however, to ensure we were not over training on the training and testing set.

Table 6.12: Confusion matrix for MNIST data set for best performance of all

	0	1	2	3	4	5	6	7	8	9	Perf
0	977	0	0	0	0	0	2	1	0	0	99.69%
1	0	1130	1	1	0	0	3	0	0	0	99.56%
2	3	3	1016	2	1	0	1	6	0	0	98.45%
3	0	0	0	1004	0	1	0	2	3	0	99.40%
4	0	0	2	0	972	0	3	0	2	3	98.98%
5	2	0	0	6	0	881	2	1	0	0	98.77%
6	1	1	0	0	3	2	948	0	3	0	98.96%
7	0	2	6	2	0	0	0	1013	1	4	98.54%
8	4	0	1	3	1	2	1	2	958	2	98.36%
9	1	5	1	5	6	3	0	5	4	979	97.03%

Maximum Cluster States Comparison

As a final test, we ran a single sample of the DeSTIN network formulation with a different number of maximum clusters per node (Table 6.14), and also ran a single sample with different number of advice states per node (Table 6.15). This basic test was designed to determine what effect, if any, this fundamental parameter would have on the overall performance. We show a slight dependence on the maximum number of clusters from Table 6.14, with comparable results to our best performance with as few as 5 clusters per node, and possibly even better results with 10 or 20 clusters instead of with the default setting of 25. Similarly, with the number of clusters per node set to 25 but with the number of advice states set lower, we again see similar performance, even with 10 advice states (although below 10 we do see a decline in performance).

Table 6.13: Comparison between Case 3 (temporal sampling) and best overall performing automatic method

	PerfBest	Perf3
0	99.69%	99.49%
1	99.56%	99.21%
2	98.45%	97.96%
3	99.40%	97.82%
4	98.98%	98.06%
5	98.77%	97.87%
6	98.96%	98.02%
7	98.54%	97.66%
8	98.36%	96.61%
9	97.03%	96.93%

Table 6.14: Comparison of performance with different number of maximum clusters per node

MaxK	NN
2	97.64
5	98.39
10	98.65
20	98.63
25	98.54
30	98.54

Table 6.15: Comparison of performance with different number of advice states per node

MaxA	NN
5	96.87
10	98.14
20	98.35
25	98.54

6.2 Face Detection

During the course of this dissertation work, experiments were performed on the MNIST handwriting dataset, although no particular tuning was applied for that dataset (for example, we did not attempt to specifically taylor development towards handwritten character recognition in terms of parameter tuning). Consequently, it is desirable to test the algorithms and work on other datasets. For our tests, we elected to not alter nor tune the networks or evolution parameters, as we believed this would give the best means of comparison since we seek a general architecture which can solve a variety of problems in different application domains. We selected the CBCL Face Database 1 from the MIT Center for Biological and Computational Learning which was recommended as a representative facial dataset for face recognition Moon (2010). The data set consists of a training and testing set drawn from different sources. The training set consists of 2429 face images from Sung (1996) which were obtained from collections of images. In many cases, some subjects were added multiple times with slight rotations of five degrees. The training set also contains a randomly selected 4545 non-faces from the non-face database of Sung (1996), which originally contained 19932 non-face images. These non-face images were obtained by creating 19×19 pixel windows which were used in the detection methods of Sung (1996) across the images, and choosing randomly non-face images. The testing set was drawn from the CMU Image database Vision and Center (2003) and consists of test sets A,B, and C which together comprise"CMU Test Set 1". This is a total of 130 images, with 507 faces, but line-drawn faces and non-frontal faces were removed, so the total number of faces is 479 as described in Heisele et al. (2000). For the non-face dataset, in Heisele et al. (2000), the CMU test set 1 was processed by using 19×19 pixel detection windows. This consists of over 56,000,000 low-resolution "images" (really "windows" from true image files). Of these windows, a subset of 23,573 non-face images were selected for the test set; they were selected because they were the most "face-like" of the windows after classification by a a 2nd-degree polynomial kernel SVM trained on histogram normalized gray level images as described in Heisele et al. (2000). Some examples of the face and non-face images of the training and testing set are shown in Figures 6.9, 6.10, 6.11, and 6.12.

We note that no pre-processing was done on the data set besides a resizing from 19×19 to 32×32 (using bilinear interpolation), and a rescaling of the image intensities using Equation 6.1

$$I(x,y) = 255\frac{I(x,y) - min(I)}{max(I) + min(I)} \tag{6.1}$$

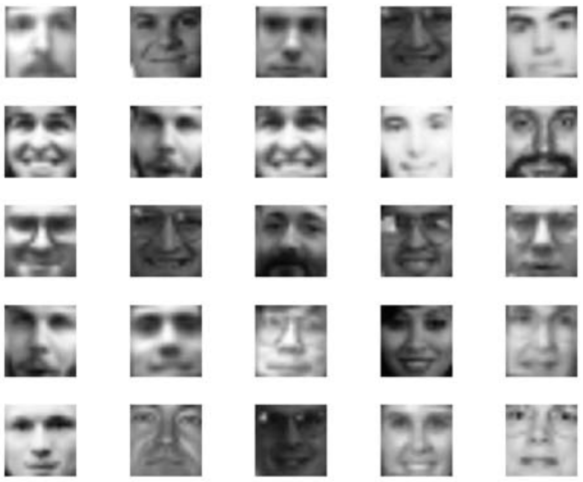


Figure 6.9: Examples of face images from the training set.

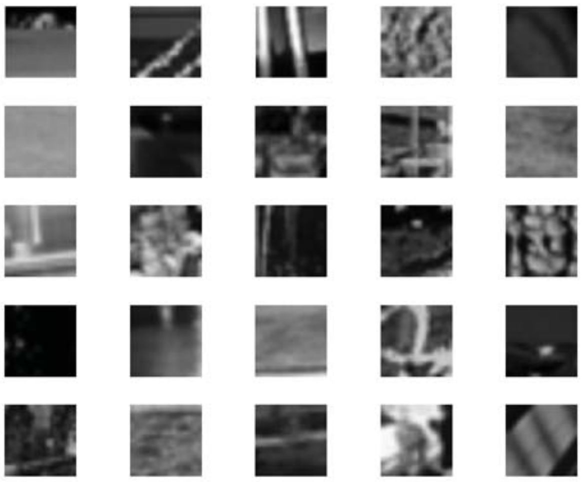


Figure 6.10: Examples of non-face images from the training set.

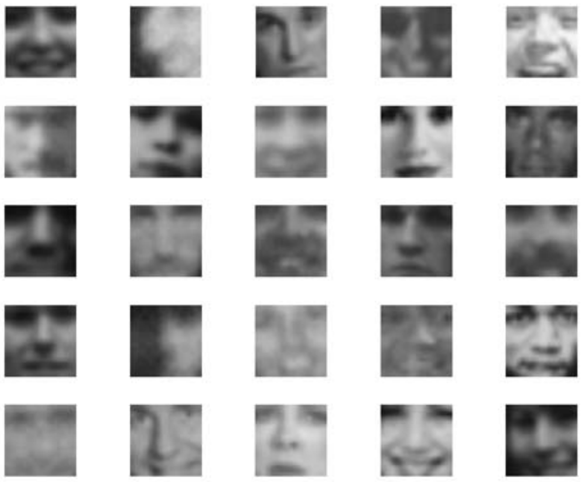


Figure 6.11: Examples of face images from the testing set.

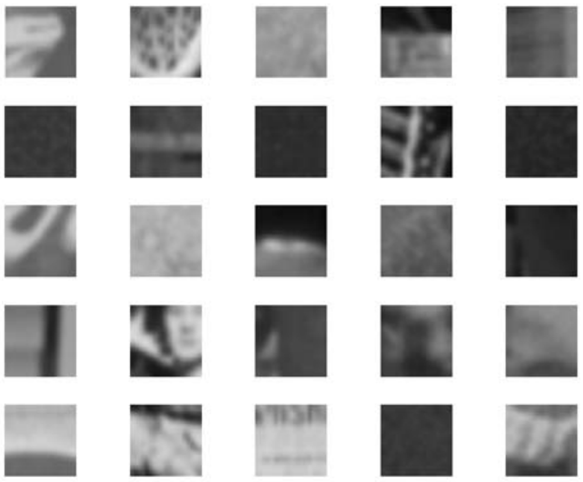


Figure 6.12: Examples of non-face images from the testing set.

6.2.1 Experimental results

Since the data set only uses two classes, with very unequal numbers of each class, we elected to show the results in terms of receiver operating characteristics (ROC) curves. These were generated by using the same supervised learning techniques in the previous work, but the neural network ensemble additive results were then thresholded at various points to generate the ROC curves. The ROC curve comparison also allows easier comparison with Heisele et al. (2000). A set of 10 DeSTIN networks were evolved using the maximum of 25 centroids per node as in the previous chapter, using the exponential probability model. We then compared their results using the L123Core method, the PCAAll method, the PCAByNode method, and the Autoselection methods. The L123Core method used the same nodes as in the previous MNIST results. We see that the best node selection methods and networks perform as well or better than the ROC curves in Heisele et al. (2000), which is superimposed on the sample ROC curves, but without specific design for face detection. Sample ROC curves are shown in Figures 6.13 through 6.18. Overall the hand-selected, L123Core, and PCAAll seem to show fairly good performance, with the exception of one network outlier in both cases. The PCAByNode performance seems to be slightly less; this is confirmed by the summary in Table 6.16. One interesting finding is that the omission of bottom layer nodes, the L123 case, is also competitive with these best methods.

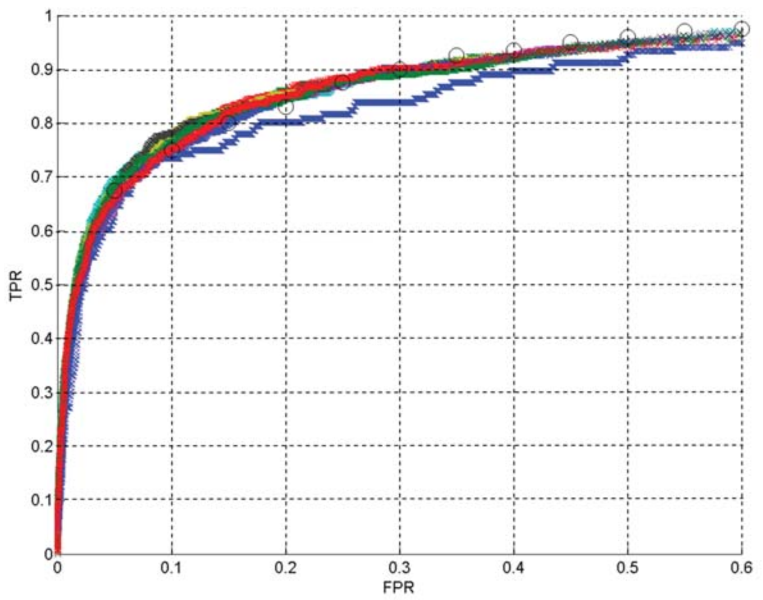


Figure 6.13: Face detection ROC curves for the hand-selected nodes. Each curve is a different trial of 10 total trials, with the results from Heisele et al. (2000) superimposed with circle markers.

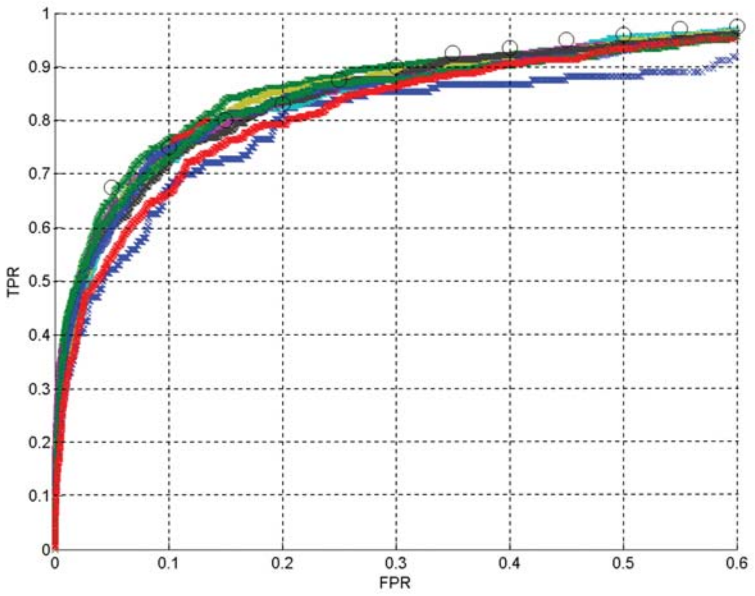


Figure 6.14: Face detection ROC curves for the PCAAll "node selection" method. Each curve is a different trial of 10 total trials, with the results from Heisele et al. (2000) superimposed with circle markers.

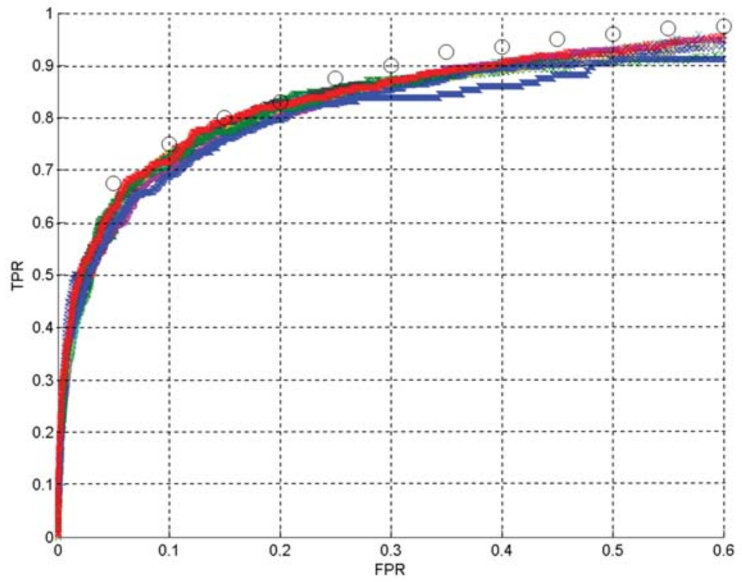


Figure 6.15: Face detection ROC curves for the PCAByNode "node selection" method. Each curve is a different trial of 10 total trials, with the results from Heisele et al. (2000) superimposed with circle markers.

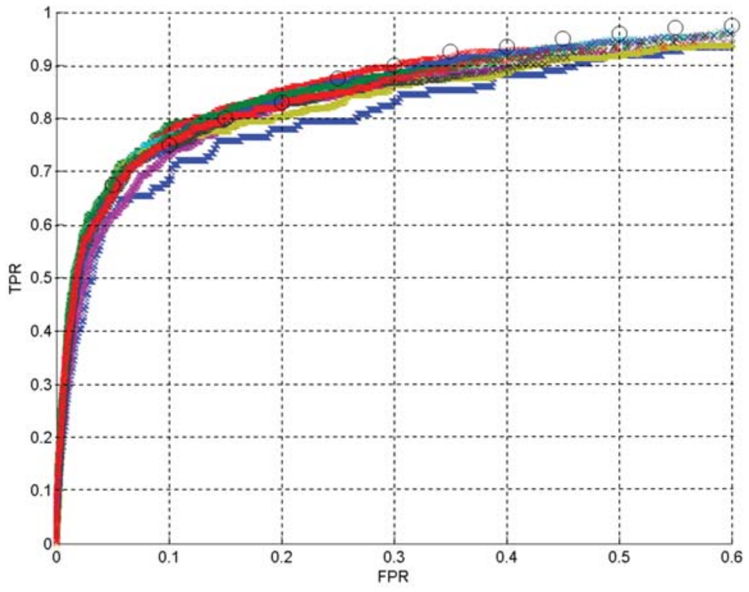


Figure 6.16: Face detection ROC curves for the top three layers only "node selection" method. Each curve is a different trial of 10 total trials, with the results from Heisele et al. (2000) superimposed with circle markers.

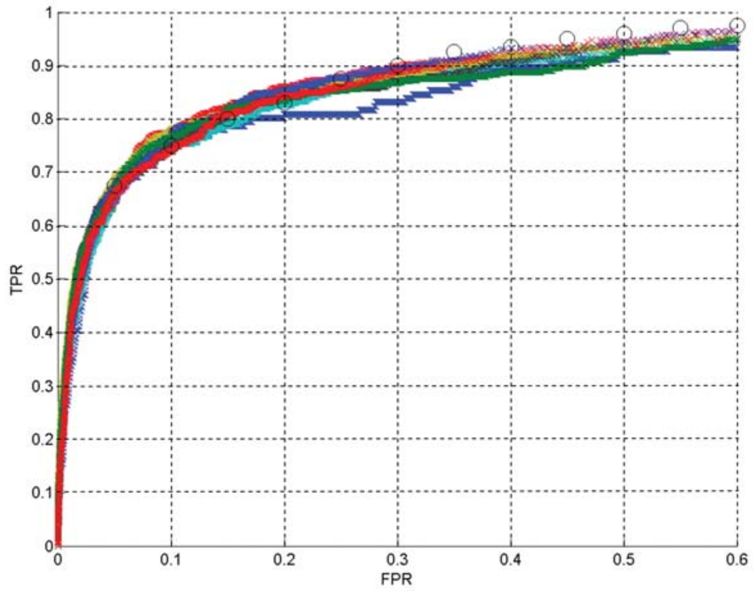


Figure 6.17: Face detection ROC curves for the L123Core "node selection" method. Each curve is a different trial of 10 total trials, with the results from Heisele et al. (2000) superimposed with circle markers.

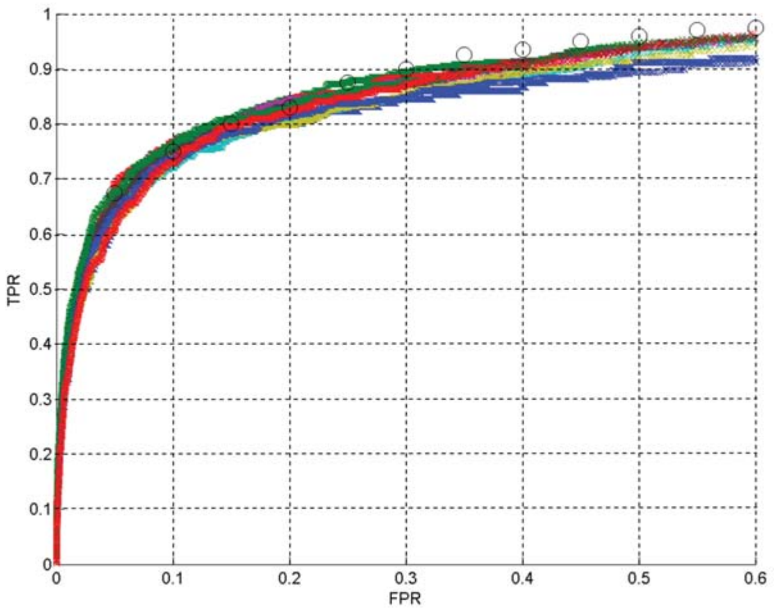


Figure 6.18: Face detection ROC curves for the auto selection method, using the "ranked 14" best nodes. Each curve is a different trial of 10 total trials, with the results from Heisele et al. (2000) superimposed with circle markers.

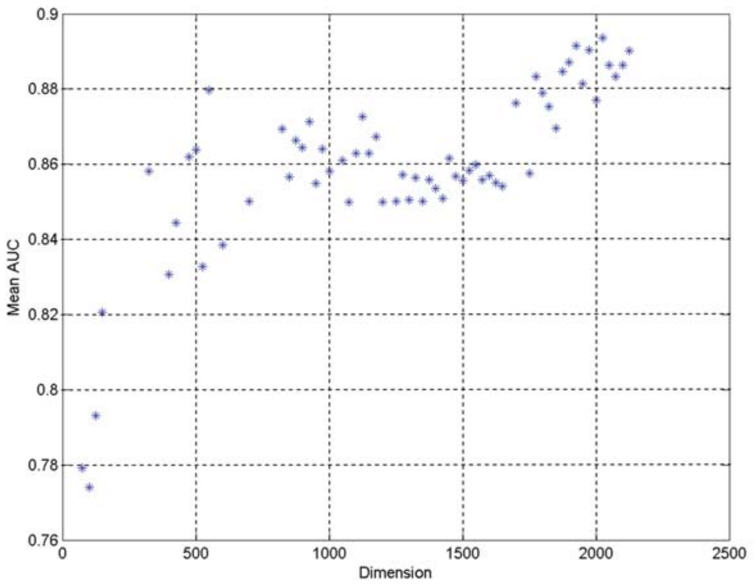


Figure 6.19: AUC as a function of total dimension, for the auto-selected nodes.

Table 6.16: Area under curve (AUC) for different node and feature selection methods across all trials

Method	MeanAUC	MedianAUC	MaxAUC
PCAAll	0.890	0.894	0.905
PCAByNode	0.879	0.881	0.889
Hand	0.900	0.904	0.905
L123	0.889	0.891	0.900
L123Core	0.894	0.897	0.904
AutoSelection01	0.791	0.789	0.838
AutoSelection02	0.848	0.851	0.879
AutoSelection03	0.858	0.856	0.883
AutoSelection04	0.866	0.868	0.873
AutoSelection05	0.859	0.853	0.880
AutoSelection06	0.854	0.852	0.864
AutoSelection07	0.859	0.856	0.870
AutoSelection08	0.850	0.853	0.863
AutoSelection09	0.855	0.859	0.876
AutoSelection10	0.860	0.857	0.879
AutoSelection11	0.875	0.876	0.888
AutoSelection12	0.884	0.886	0.900
AutoSelection13	0.888	0.889	0.901
AutoSelection14	0.887	0.892	0.899
AutoSelection15	0.885	0.889	0.890
AutoSelection16	0.888	0.888	0.888

These results are summarized in terms of the area-under-the-curve or AUC measurement, which allows us to create a single, composite score for performance. Although we believe this method is competitive with the methods discussed in Heisele et al. (2000), we recognize there is room for improvement. We also note, however, that the chosen images were designed to be difficult; thus, running the DeSTIN face detectors on normal images likely would produce much better performance. However, our core purpose here for this work is to show the use of DeSTIN as an unsupervised feature extraction method for demonstration. In particular, we note that nothing was changed from the MNIST experiments, except that the dual-class nature of this problem allowed us to change the supervised learning system output to create ROC curves as opposed to a single classification to more properly compare with Heisele et al. (2000).

6.3 Optic Nerve Detection

In Chapter 1, we discussed "hand engineered approaches to feature extraction in images and specifically focused on an example of optic nerve detection Tobin et al. (2007). While the hand-engineered features in that example are not as specific as in other domains (such as minutiae in fingerprint classification), they are an example of how we can apply domain knowledge to solve an image understanding problem. Since a large part of the motivation for DeSTIN is to avoid such hand-engineered approaches, we close by including a test case for optic nerve detection.

The main cause of blindness in the industrialized world is diabetic retinopathy, a disease associated with diabetes. The retina has a variety of specific anatomical structures which can be easily identified, including the optic nerve (ON) which is a circular spot, often somewhat bright in appearance, where the nerves leave the back of the retina; the vasculature which consists of the blood vessels and veins in the retina that make a sort of parabolic shape emanating from the ON; and the macula or fovea (where most human vision is concentrated), which is a darker spot in the center of the main vessel area. The American Academy of Ophthalmologists recommends individuals with diabetes undergo retina screening annually Panel (2008), but with the growing and projected number of diabetics in the world is making wide spread screening by healthcare personnel intractable Abràmoff et al. (2010). Thus automated retina disease diagnosis can potentially have very high public health impact. For automated screening of the human retina, systems generally attempt to automatically locate these different structures to help establish a sort of coordinate system, then specific lesions are located and diagnosis is performed Teng et al. (2002). For a good overview of this type of processing, see Abràmoff et al. (2010).

The training database was derived from a set of images from the Retina Image Screening and Analysis (RISA) system Li et al. (2011). This dataset consisted of 1391 images which had been screened with a quality detection test Giancardo et al. (2008). The test database was obtained from the Messidor dataset (kindly provided

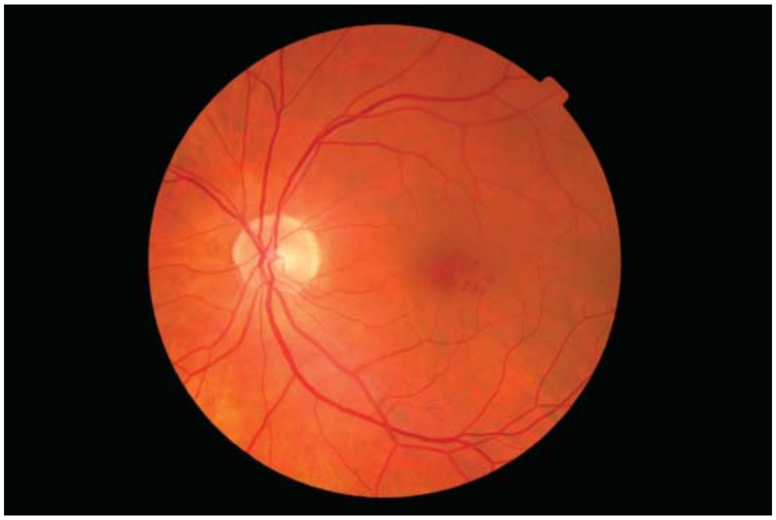


Figure 6.20: Example of a retina image from the Messidor data set. The optic nerve is the circular object on the left; the vascular tree emanates in a parabolic shape from the optic nerve; and the macula is the darker region in the center of the image.

by the Messidor program partners Partners (2008)). This set consists of 1200 images. In both sets, the optic nerve was manually selected for each image and cropped, then resized to fit a 32×32 pixel region. Non-ON regions were then randomly chosen from each image, repeatedly sampling but eliminating previously selected regions. About 25 non-ON regions were selected for each image.

For an interesting comparison, two sources of image data were used. The first was "raw image pixels" where the green channel of each image was used, with normalization as described in Chapter 6.1. In addition, part of the quality measurement process generates a vessel-segmented image using the method of Zana and Klein (2001) which is also used in the optic nerve detection method of Tobin et al. (2007). Thus, we also extracted ON and non-ON regions from these vessel segmented images for comparison. Examples of both are shown in Figures 6.21 through 6.28. The ON regions for the vessel images have a sort of "spiderweb" appearance, generally consisting of vertical and slightly diagonal vessels emanating from the ON center.

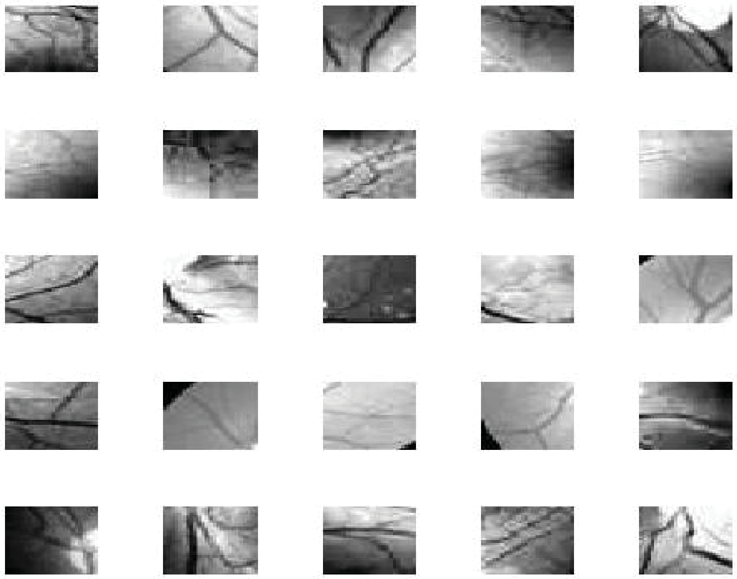


Figure 6.21: Example non-ON images from the training set without vessel segmentation.

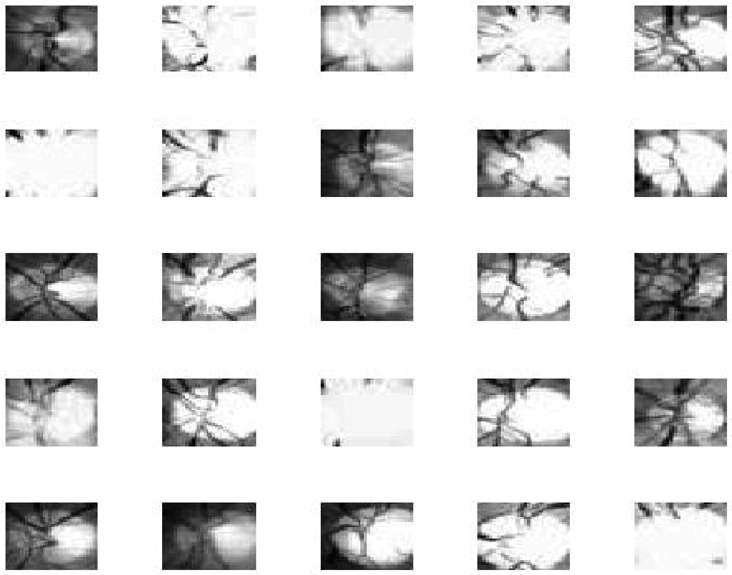


Figure 6.22: Example ON images from the training set without vessel segmentation.

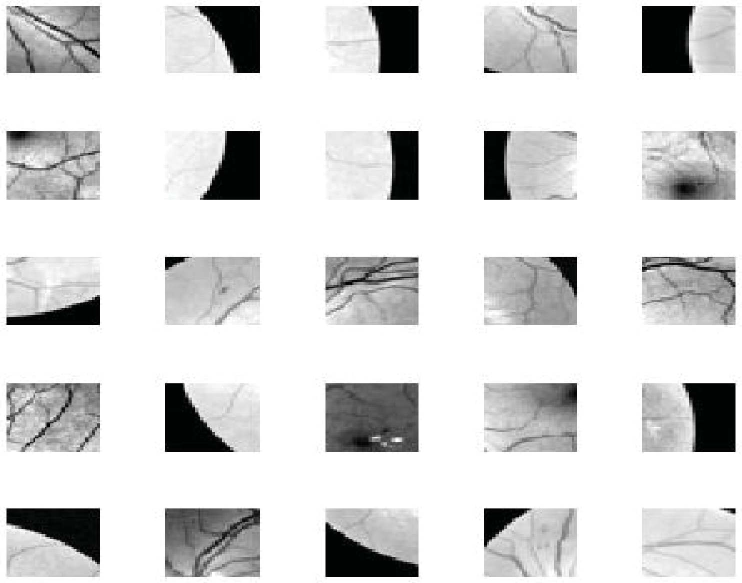


Figure 6.23: Example non-ON images from the testing set without vessel segmentation.

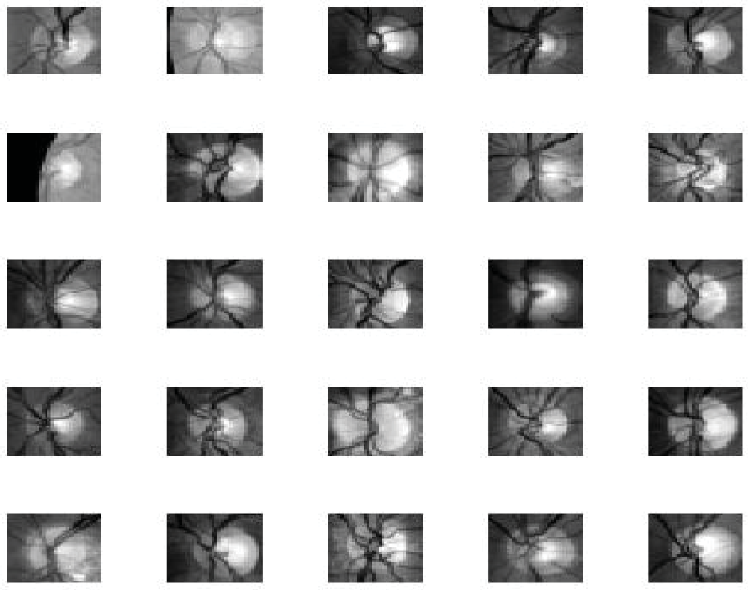


Figure 6.24: Example ON images from the testing set without vessel segmentation.

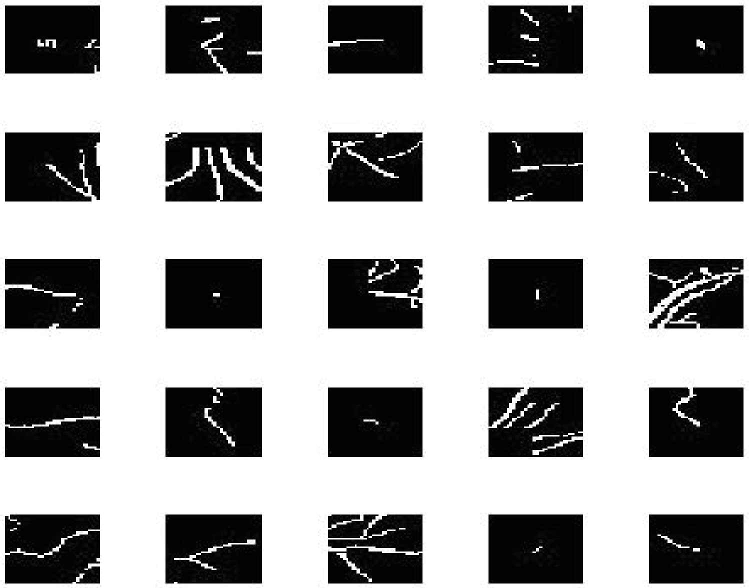


Figure 6.25: Example non-ON images from the training set with vessel segmentation.

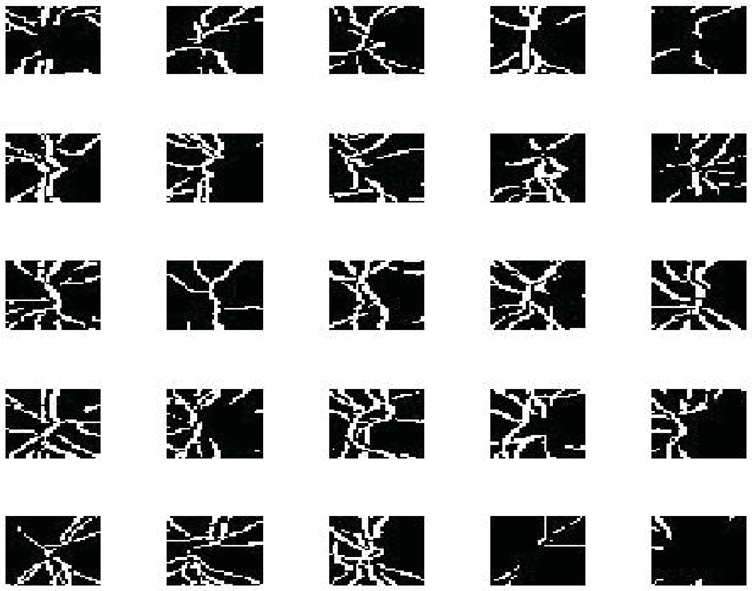


Figure 6.26: Example ON images from the training set with vessel segmentation.

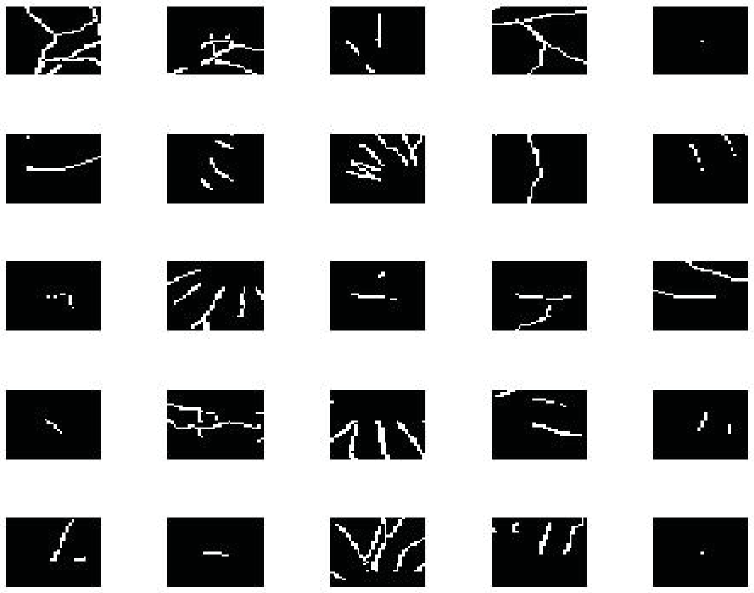


Figure 6.27: Example non-ON images from the testing set with vessel segmentation.

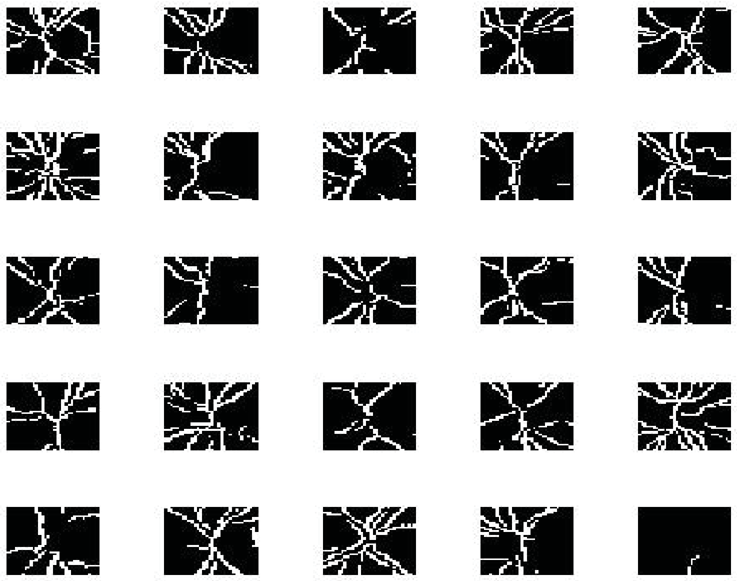


Figure 6.28: Example ON images from the testing set with vessel segmentation.

6.3.1 Optic nerve detection results

Rather than an exhaustive study of the different DeSTIN configurations, we used the same DeSTIN topology as the previous experiments, limiting to the ECM clustering method with 25 maximum states and tabular method for $P^a_{ss'}$. We also used the L123Core nodes for processing. The ROC curve was generated in the same manner as the face detection task and is shown in Figure 6.29. In this case, the two methods performed much better than the face detection set, but we should note that generally optic disk detection is a simpler task Karnowski et al. (2009) especially in normal retina screening environments. The vessel segmentation case had a AUC of 0.951 and the non-segmented case had even better performance, 0.973. The crossover point, where the sensitivity matched the specificity, was approximately 88.3% and 92.3% respectively.

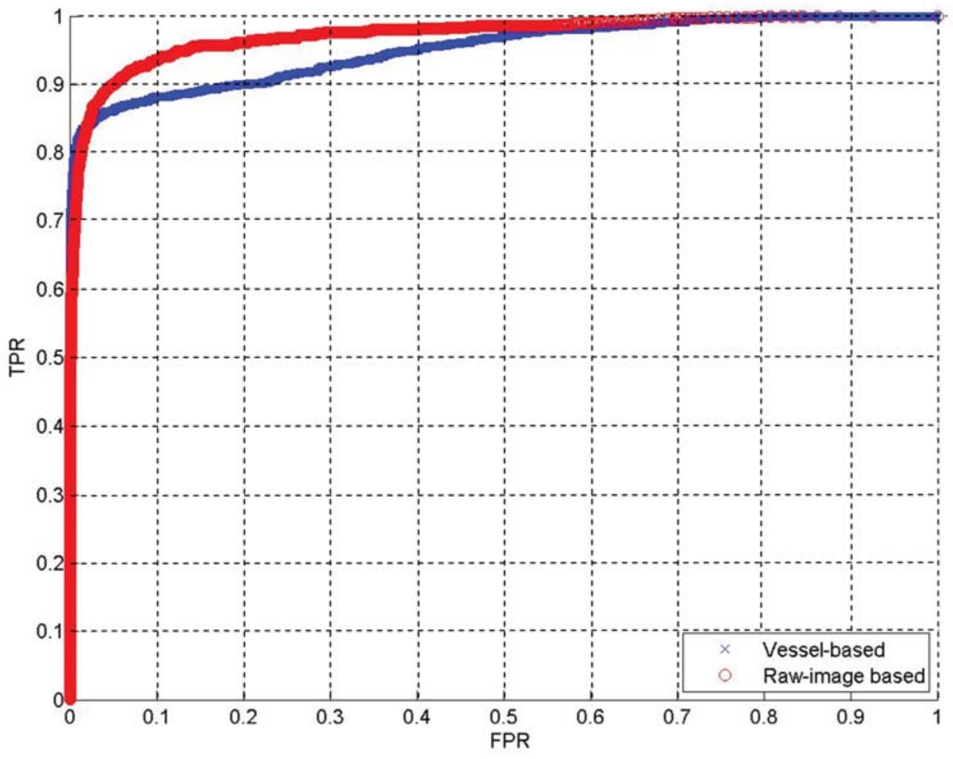


Figure 6.29: ROC curve comparing detection using DeSTIN L123Core on vessel segmented examples and non-vessel-segmented (raw image) examples. The raw image case has better performance especially in the 0.05 to 0.5 FPR range.

6.4 Results Summary

We covered three distinct application areas in this chapter. In the first application area, handwriting digits, we discussed a variety of different experimental configurations for DeSTIN, although they were all based on the same 64-16-4-1 topology as depicted in Figure 3.1. We first discussed the MNIST test set and results of simple experiments on the raw image data, both with a kNN classifier and a neural network ensemble. These initial tests confirmed the relatively high performance of the most basic supervised learning strategy on the dataset, which is a known fact in the literature, and also set a baseline for the performance of DeSTIN.

We then reviewed the effect of temporal sampling of the belief states, including the use of a rough fuzzy-based estimate of the $P(o|s)$ static construct given by Equation 4.1, which yielded top performance of 97.98% or 2.02%. These results were then improved by using a true $P(o|s)$ model; for all tested cases, the exponential, Gaussian and Rayleigh model, performance improved by approximately 0.5

We built on these results, which were flawed in two matters: first, they were very high-dimensional and did not form an evolving consensus belief state for each node, and second, they used a heuristic hand-selected set of nodes. For our next experiments, we added the $P^{a}_{ss'}$ construct which created a multiple observer model and formulated a final overall belief $B(A)$ across a variety of different parental advice states. We compared these results with a function-approximation model which yielded comparable results, at a great memory savings albeit with some additional computational cost both in training and in response evaluation.

For our next experiments, we performed a test of 10 different DeSTIN networks, with evolving cluster formulations using the ECM algorithms described in Chapter 4, and a variety of different selected node configurations, including our previous "Hand" selected nodes, and methods based on PCA. The best results obtained in this work used the L123Core configuration, which used all the nodes of layers 1, 2 and 3 as well as the "center" 4×4 nodes from the bottom layer, which achieved a maximum performance across 10 different trials of evolving DeSTIN networks of 98.78% correct or an error rate of 1.22 %. However, many of the different network choices achieved very similar performance. If pressed to simply choose a configuration, the Auto selection methods above a dimensionality of 300 are a reasonable choice, as they seem to outperform most methods of comparable dimension (for example, PCA reduced the dimension to roughly 400 with a performance of 98.18%, but the auto processing of similar dimension scored 98.50%).

We also note that the algorithms tended to use the maximum number of allocated clusters for each node. This was slightly surprising, but we believe the experiments and discussion from Chapter 4 showed why this would be the case. Some other means of constraining the clustering to a different number of clusters than the ECM method of the minimum distance could produce different effects, but overall we believe the system has proven itself robust.

We then performed a small test of the effect of the different number of maximum clusters and advice states. Relatively small numbers of individual node clusters are needed for success, which seems to validate the notion that the training phase (where we learn the prototypes of the data) is less important than the encoding phase (how we generate features from the functional combination of prototypes and the data), as mentioned in Coates and Ng (2011).

We next applied DeSTIN to a different image recognition problem, face detection. As in the MNIST results, the DeSTIN networks created and tested for this face detection problem generated useful features for supervised learning, with ROC curves that exhibited similar performance to Heisele et al. (2000). We found that the hand-selected nodes seemed to give the best performance, but many of the automatic selection methods worked well, and as with the MNIST set, the L123Core method was generally a good choice. We note that one issue with testing the DeSTIN scheme on additional data sets is the scalability of the method; in our current implementation, we are most well-suited to processing relatively small images (32x32). This can be addressed in the future through more advanced implementations, as discussed in the future work section of the final chapter.

Finally, we tested the DeSTIN processing on a third problem, the detection of optic nerves in retina images. The DeSTIN system performed quite well on this task, with no changes to the algorithm or topology used in the previous two test cases. Note that other domain knowledge could likely improve the detection results; this is simple information such as we expect the ON to be on the side of the image for macula-centered images, and we only expect one ON per image. Overall however we believe these results nicely close the circle on this work and illustrate the utility of automated unsupervised feature extraction using a system such as DeSTIN.

Chapter 7

Resource and Noise Analysis

In this chapter we discuss and analyze two additional aspects of DeSTIN. The first is the relative computational load of DeSTIN, in particular we compare DeSTIN with convolutional neural networks and deep belief nets. The second is a noise analysis of the DeSTIN architecture. For both topics, the analysis is conducted by first attempting to generalize across the different parameters of the DeSTIN topology and node operations, then drawing upon a specific test case, the MNIST images studied in Chapter 6, for further illustration of these points.

7.1 Computational Analysis

The direct comparison of DeSTIN to other deep learning architectures is somewhat problematic for several reasons. First, without an actual implementation there can be aspects of the architecture that are more difficult or computationally intensive than we may learn from reviewing the literature on the subject. Second, when iterative processes are required, as are often needed for unsupervised learning (and some supervised learning algorithms), the comparisons may be difficult to make without experimental analysis over a broad range of parameters. Next, in the specific case of DeSTIN, CNNs and DBNs, the architectures have slightly different objectives. For example DeSTIN is largely an unsupervised feature extraction engine, while CNNs combine feature extraction and supervised learning in an integrated architecture. Finally, from the literature on CNNs and DBNs as applied to the MNIST data set, it would appear that there is some level of tuning - not of the features, but of the architecture and training regiment - which has been undertaken to improve results. Therefore it is difficult to understand exactly what we are comparing in these cases. Nonetheless, we proceed with these caveats in mind and attempt a first-order approximation of the computational load. We will therefore present this discussion in four sections, one for each targeted architecture, then compare their values for the MNIST data set based on published literature and our results.

7.1.1 DeSTIN computational resource requirements

There are two distinct phases of DeSTIN where the computational load is an issue. The first is the training phase, where the network is initialized and presented sequences of target images. The second is the "response" phase, where a trained network is again presented images and an output feature vector is generated. These response vectors are then used for supervised learning. Therefore, we distinguish between these two modes of operation for clarity. We also note that DeSTIN consists of a series of nodes that have some independence to one another, but essentially run the same algorithm. We examine the computational components of a single node. This node is assumed to have K total cluster centroids in its static library, which we will generalize as a maximum requirement. We also assume the node is working on a D-dimensional space for these centroids. For the dynamic component, we assume that the node has A total cluster centroids formulated over L movements for the system advice element, which generally is in a vector space of dimension $L \times C \times K_c$, where C is the number of children, and K_c is the number of centroids from each child's static process. Finally, we assume a single computational element of complexity T is needed for each static centroid to compute the value of $P(o|s)$. We assume we work in cosine similarity space. Thus, in one movement, the following steps must be taken:

- Input data vector o must be normalized, to form vector o_n. This requires D multiplications, the computation of a square root, and a scalar division between D elements, for a total of $2D + T$ operations.
- Vector o_n is compared with each of K centroids by cosine distance. This requires D multiplications for each of the K clusters, followed by a sorting operation of order $K \log(K)$.
- The winning vector is updated. Regardless of the exact methods discussed in our online clustering section, this requires roughly D multiplications and D additions.
- Various metrics regarding the clustering progression are monitored and recorded. These include adaptation of the learning rate requiring a single division, evaluations of the μ, σ, and ρ metrics each requiring a multiplication and addition, and an estimate of the parameters of the chosen probability distribution requiring an additional multiplication and addition. Furthermore, the ρ calculation requires an additional transcendental function, T for each dimension. In our implementation we have also monitored other items of interest, including the online error rate, but these are not included in this comparison. Thus, we estimate these online metrics require roughly $3D + 4$ operations.

This is summarized as

$$\mathcal{N}_{train}^{static} = L\left[(2D+T) + (KD + K\log(K)) + (2D) + (3D + 4 + TD)\right] \tag{7.1}$$

$$\mathcal{N}_{train}^{static} = L\left[7D + T\left(D+1\right) + K\left(D + \log(K)\right)\right] \tag{7.2}$$

(Note that we must multiple the entire process by the number of movements L as shown.)

The advice generation component is very similar except there is only one operation per L movements, and it uses a different number of centroids and dimension as described above. This step thus requires

$$\mathcal{N}_{train}^{advice} = 7LCK_c + T(LCK_c + 1) + A(LCK_c + \log(A)) \tag{7.3}$$

Following the advice generation, the dynamics operation must be estimated. For the tabular method there is minimal cost with respect to computational resources. However, for the function approximation method, there is some issues, and therefore we assume a neural network with H hidden nodes. The function maps LKA values to K values, and we assume a single hidden layer, with T the cost for each activation function. The computational load is approximated as the number of weights to adjust along with a single application of the transcendental function. We also have HK weights for the output layer.

$$\mathcal{N}_{train}^{FA} = LH\left(L + 2K + A + T\right) \tag{7.4}$$

After training, the response generation requires the following at each movement:

- Input data vector o must be normalized, to form vector o_n. This requires D multiplications, the computation of a square root, and a scalar division between D elements, for a total of roughly $2D + T$ operations.

- Vector o_n is compared with each of K centroids by cosine distance. This requires D multiplications for each of K clusters. The sorting operation is not needed in this step, although it is done for debugging purposes in our implementation.

- The probability $P(o|d)$ is computed for each centroid, requiring KT nonlinear functions.

- The belief is computed, requiring a vector-matrix multiplication of K^2 operations along with K additions. Furthermore, these beliefs are computed for each value of A.

- Finally all operations above are performed at each movement.

Thus, the response component is

$$\mathcal{N}^{static}_{response} = L\left[2D + T + DK + KT + A\left(K^2 + K\right)\right] \tag{7.5}$$

$$\mathcal{N}^{static}_{response} = L\left[2D + T + K\left(AK + D + T + 1\right)\right] \tag{7.6}$$

Note that with the function approximation method, an additional calculation is required since the $P^a_{ss'}$ "table" is effectively generated by function calculation instead of simple look-up methods. Thus an extra computational effort at each movement and advice state is needed.

$$\mathcal{N}^{FA}_{response} = LA\left[H\left(L + 2K + A + T\right)\right] \tag{7.7}$$

The memory requirements are also of interest. We discussed the memory requirements of the dynamic component in Chapter 5, where we also compared the neural network function approximation with the tabular method.

The static component requires a much more modest level of memory, in particular largely just the centroid library. The memory requirement is $K(D+6)$ total elements, which includes the online calculations for ρ, μ, σ, and the parameters of the probability model for $P(o|s)$ and its online calculation (including learning rate). There is also a dynamic component which is not included here, but during training the advice generator is required. This requires a measure of memory comparable to the static clustering methods, but with different parameters as described above. Also, we do not require the computation of an equivalent probability distribution, so the additive constant is 4 instead of 6:

$$M_{static} = K(D + 6) + A(CLK_c + 4) \tag{7.8}$$

The total cost is summarized in Tables 7.1, 7.2, and 7.3, along with an estimate for the evolved cases described in the previous chapter. Note that C is 4 for all nodes except layer 0, where it is 0, and for layer 3, where it is 1 (since the top layer generates "self-advice"). Note that the function approximation effort takes roughly 50% longer to run.

As a final note, DeSTIN has a number of free parameters which must be trained. One issue of note here is that the training procedure is fairly simply in all cases, lending itself easily to online adaptive processing. At each node, the process must learn K centroids of size D, a probability distribution for each of the K clusters (with a single parameter), A centroids of size LCK_n, and the dynamics method requiring at most LAK^2. While the latter component is very large, as discussed earlier, it is very simple to learn by counting transitions, so in truth the total number of parameters to learn per node for the static component and the advice generation component is

$$P = K(D + 1) + A(CLK_c) \tag{7.9}$$

- Layer 0: 27200 (425 per node)

Table 7.1: Summary of DeSTIN computational cost during training phase, in MFlops

Layer	Nodes	D	Static	Advice	DynamicsFA	TotalFA	TotalTab
0	64	16	3.27	0	6.1	9.37	3.27
1	16	100	4.43	4.30	1.52	10.26	8.73
2	4	100	1.11	1.08	0.38	2.56	2.18
3	1	100	0.277	0.067	0.095	0.44	0.34
TOTAL	85	--	9.08	5.44	8.10	22.63	14.53

Table 7.2: Summary of DeSTIN computational cost during response, in MFlops

Layer	Nodes	D	Response	Additional FA	TotaWithFAl
0	64	16	66.94	6.10	73.04
1	16	100	19.06	1.52	20.58
2	4	100	4.76	.38	5.14
3	1	100	1.19	0.095	1.29
TOTAL	85	--	91.95	8.10	100.05

- Layer 1: 2600400 (162525 per node)
- Layer 2: 650100 (162525 per node)
- Layer 3: 42525

This is a total of 3320225, or over 3 million parameters which must be learned.

For the final analysis, assuming 20,000 digit presentations for the DeSTIN case, the estimated training operations to obtain features in DeSTIN is 290 GFlops for the tabular method or 452 GFlops for the function approximation method. The additional time to train our default supervised learning method adds additional computational load. This load can be approximated as, for a single sample, $IH_1 + H_1H_2 + H_2O$, where $O = 10$ is the number of output labels, and $H_1 = 40$, $H_2 = 40$. This is a total of 42,000 values. There are $H_1 + H_2 + O$ nonlinear operations required, for a total of 900 operations. Assuming we can still approximate the training requirements as $2(P + C) + NL$, where here $P = C = 42000$, and $I = 1000$ for worst-case, the total load is 169 MFlops for a single pass. However, we used all 60000 training samples, for roughly 150 epochs. This means a single neural network requires 1.5 TFlops; if we use the ensemble average for 10 neural networks, we are now approximately 15 TFlops for training.

Table 7.3: Summary of DeSTIN memory cost in 10^6 units

Layer	Nodes	D	Static	Table	FA	TotalTab	TotalFA
0	64	16	0.04	64	0.091	64.04	0.13
1	16	100	2.60	16	0.023	18.60	2.63
2	4	100	0.65	4	0.006	4.65	0.66
3	1	100	0.043	1	0.0014	1.04	0.044
TOTAL	85	--	3.34	85	0.121	88.3	3.46

7.1.2 CNN computational resource requirements

For this discussion, we focus on the implementation of LeNET 5 from LeCun et al. (1998), which is very thorough with respect to implementation details and computational requirements. First, we note that there are two main classes of layers in CNNs. The first, a convolutional layer, consists of one or more feature maps. Each map has one or more inputs, which are either images of the maps from previous layers. The convolution is performed by taking a windowed region of the input map and applying a multiplicative weighting and adding all values together across all the input maps. A bias is then added and a nonlinearity is invoked. This operation is repeated across each region, which can be interpreted as "sliding" around the input space, so that the overall operation is a convolution (although the convolution is not computed as a true sliding window operation). To summarize:

- The convolutional layer has M feature maps.
- Each map has I inputs, where an input is a map or image from the previous layer. Each input is assumed to be of size $F \times F$.
- Each $N \times N$ region in the input is weighted, summed, and a bias added. This is reproduced for all $N \times N$ regions in the input layers.
- A nonlinearity is applied to each weighted-sum-biased result.

Thus the number of parameters which must be computed in the learning process in a convolutional layer is given by

$$N^{CCNN}_{parameters} = M\left(IN^2 + 1\right). \tag{7.10}$$

The number of connections is given by

$$N^{CCNN}_{connections} = MF^2\left(N^2 + 1\right) \tag{7.11}$$

The number of connections is important because each connection has a weighting factor applied. The number of nonlinearities is given by

$$N^{CCNN}_{nonlinearities} = MF^2 \tag{7.12}$$

The second type of layer is the sampling layer. This, too, consists of one or more feature maps, with one or more inputs from the previous layer, but the difference here is that the output is not a sliding convolution, but rather a sub-sampling to reduce the overall size of the field. This sub-sampling is accomplished through a single weighting factor applied to all pixels in the window. The windows are non-overlapping, as opposed to the convolutional layer. As in the convolutional layer, after the weighting is applied, a bias is added and a nonlinearity is applied. To summarize:

- The sampling layer has M feature maps.
- Each map has I inputs, where an input is a map or image from the previous layer. Each input is assumed to be of size $F \times F$.
- Each non-overlapping 2×2 region in the input is weighted, summed, and a bias added. This results in an output that is size $\frac{F}{2} \times \frac{F}{2}$. (In principle the regions could downsize by a factor that is not 2, but all implementations found in the literature use a factor of 2.)
- A nonlinearity is applied to each weighted-sum-biased result.

Thus the number of parameters which must be computed in the learning process in a sampling layer is given by

$$N^{SCNN}_{parameters} = M\,(I+1) \tag{7.13}$$

The number of connections assuming a 2-to-1 reduction in size is given by

$$N^{SCNN}_{connections} = 5MF^2 \tag{7.14}$$

The number of nonlinearities is given by

$$N^{SCNN}_{nonlinearities} = MF^2 \tag{7.15}$$

The CNN typically has a final output layer that is the mapping to the actual class labels. For our analysis, we can effectively ignore it as it is fairly small, at least in the implementations in the literature. The memory requirements of the CNN is essentially identical to the number of free parameters to compute.

In the case of LeNET-5, there are 6 separate layers, denoted as C_1, S_2, C_3, S_4, C_5, and F_6, as well as an output layer implemented as a radial basis function, with 10 outputs (one per digit). Interestingly, each RBF has a prototype which is a 7×12 or 84 pixel“image” of an ASCII character 0 through 9. Thus the supervision is performed by a physical image which creates a target for the learning algorithm to transform the input data. The F_6 layer contains 84 feature units, and each has as inputs the 120 feature maps of C_5 which are sized 1×1. Each feature of F_6 corresponds to a pixel

Table 7.4: Summary of CNN computational cost (LeNET-5)

Layer	Maps	I	F	Param	Conn	NL	MFlops
C_1	6	1	28	156	122304	4704	0.17
S_2	6	1	14	12	5880	1176	0.02
C_3	16	3 to 6	10	1516	156000	6000	0.22
S_4	16	1	5	32	2000	400	0.006
C_5	120	16	5	48120	48120	1920	0.07
F_6	84	1	120	10164	10164	10080	0.11
TOTAL	--	--	--	60000	344468	24280	0.587

in the prototype for the RBFs, and thus we see that through the learning process the output of F_6 is made to resemble the relevant prototype embedded in the RBF for that digit. In addition, the connection between S_2 and C_3 is a little unusual, in that the 6 maps of S_2 contribute differently to the 16 maps of C_3:

- The first 6 maps of C_3 have, as inputs, all combinations of 3 consecutive maps of S_2
- The next 6 maps of C_3 have, as inputs, all combinations of 4 consecutive maps of S_2
- The first 3 maps of C_3 have, as inputs, some combinations of 4 non-consecutive maps of S_2
- The final map of C_3 has all 6 maps from S_2 as inputs

These peculiarities apparently arose from the development cycle of LeNET-5 and its predecessors. We note that while some small details of the implementation of LeNET-5 were undoubtedly tuned to improve performance, the CNN implemented truly does learn features; the topology may have some small amount of hand-tuning, but the features it learns are not.

Regardless, we summarize the computational resources of LeNET-5 in the following table.

The RBF layer is excluded above, but it consists of 10 RBFs (one per class) with 84 inputs each. Each RBF computes the Euclidean distance between the 84-pixel prototype and the output vectors from F_6 as shown in Equation 7.16. Thus there is a total of 84 differences, 84 multiplications, and 84 additions for each RBF, for a total of 2520 operations which is negligible overall.

$$RBF_i = \sum_{j=0}^{j=83} (x_j - w_{ij})^2 \tag{7.16}$$

For a single digit presentation, then, we expect a total of 587,268 operations (assuming a non-linearity operation takes roughly a factor of 10 longer than a regular operation (this was experimentally calculated using a comparison between multiplication operations and exponential functions)).

A special note is required regarding training, which uses stochastic gradient descent. We can approximate a single training pass with one sample as we did earlier by a set of X nonlinearities and $2(P + C)$ arithmetic operations, where $X = 24280$ is the total nonlinearities, $C = 344468$ is the total number of connections, and $P = 60000$ is the total number of parameters from table 7.4. This is a total of 808936 operations and 24280 non-linearities. The paper describes training as proceeding with 20 iterations through all 60,000 data samples in the MNIST training set, which is a total of approximately 9707 GFlops and 29 GNL (giga-non-linearities). This is very close to 1 TFlop for total training.

7.1.3 DBN computational resource requirements

A deep belief net is comprised of layers of Restricted Boltzman Machines. The computational cost for an DBN is actually fairly small, as each unit is largely a simple weighted sum of inputs, which is used to change the state of the unit in a probabilistic manner. For an RBM, the number of weights is equal to IH, where I is the number of inputs and H is the number of hidden units. A random variable call is assumed to require R operations. Roughly $2IH + R$ operations are needed for each node to compute the activation function:

$$p(s_i = 1) = \frac{1}{1 + \exp\left(-b_i - \sum_j s_j w_{ij}\right)} \tag{7.17}$$

In addition, there are a total of H nonlinearities needed in each RBM as well (the exponential function from 7.17).

The difficult computation cost in DBNs is in the training of the network. The training of the DBN, at least for the MNIST problem for comparison, consists of the following steps. First, the network is trained layer-by-layer. The bottom-most layer is trained by presenting images of the training set and using the greedy algorithm for contrastive divergence. This can be repeated some number of times. Then, the next layer is trained by again presenting input images to the bottom layer, propagating each one through the bottom layer to the next layer, then using the contrastive divergence algorithm to update the weights and biases for this layer. This is then repeated until the second layer is finished. Then, we repeat, propagating through the bottom, next, and new training layer. The top layer, which is an associative memory, is trained with the known labels as part of the input.

After the greedy layer-by-layer training, the network is "fine tuned" with supervised learning and backpropagation to improve the system response. The "up-down" algorithm of Hinton et al. (2006) starts by taking a sample input and propagating through the network to stochastically set the values in the hidden units. Then the weights are adjusted using multiple iterations of the Gibbs sampling. It is unclear why it was determined that multiple iterations were needed, other than this gives the ability to produce a more accurate answer at the cost of speed. Finally, during the down pass the actual label is applied and backpropagated down. However, in this phase only the bottom few layer weights are changed; the top ones are not changed. The Gibbs sampling in this stage seems to require more than a pair of iterations through the data; in the case of Hinton et al. (2006), for 100 epochs an up pass was performed, 3 iterations of Gibbs samples were used, then the down-pass was performed. Then, for the next 100 epochs, six iterations were used, and for the last 100, 10 iterations were used.

The network of Hinton Hinton et al. (2006) which was used to generate their MNIST results consists of an input layer with 768 inputs (for 28×28 sized input images), a layer of 500 units, a layer of 500 units, a layer of 2000 units, and finally a layer of 10 units for the output. This architecture seems to be experimentally determined for this problem, as was LeNET-5. The computational load of the network for system responses is shown in Table 7.10. Their training regiment was somewhat complicated. In the training, they took the 60,000 input samples and used 44,000 of them, and divided them into a balanced set of 440 batches, each with 100 digits (10 examples of each class). Note that the training time for the layer by layer representation was roughly a few hours per layer using Matlab on a 3 GHz processor Hinton et al. (2006). The total time is thus likely about 10 hours, with 30 epochs specified; and this is presumably on all mini-batches. However, it is difficult to tell if that implementation was optimized since Matlab was used. The performance of the network at that point was 2.49% error, or 97.51% correct. In any event, this pre-training time was minimal compared to the fine-tuning training which followed, where a 10,000 sample validation set was taken from the remaining (60000-44000=16,000) images. The up-down algorithm was used for this fine-tuning by running 300 total epochs. At this point the network performance was 1.39% error or 98.61% correct. An additional 59 epochs were then applied using the entire 60,000 training images, by running until the performance on the 60K training set matched the performance on the 44K training set. At this point the error rate was 1.25%, or 98.75% correct. This apparently took "about a week" or 7 days.

If we assume the 30 epochs used for the pre-training took 10 hours, we can assume roughly 3 epochs per hour were needed. Therefore the additional approximately 360 epochs used to fine-tune the weights in the supervised phase likely took approximately 120 hours, which is likely the source of the time period cited. From another perspective, one complete fine-tuning pass was likely not the same as a pass in the pre-training due to the additional Gibbs sampling steps used.

We shall attempt to estimate the time for this work by using an assumption of a single pass, either up or down, through a layer with G Gibbs sampling steps. We assume there are I inputs and H hidden nodes. A random variable call is assumed to require a R operations. Thus a sampling requires $2IH + HR + HT$ operations as before. A Gibbs sampling step requires two samples (to go from h to x and back to h), with the second sampling invoked on the input nodes, and an extra $2IH$ steps to estimate the probability of achieving that final sample; thus, all total, we have the operations given as

$$G\left[4IH + (H+I)(R+T)\right] + 2IH \tag{7.18}$$

To pretrain layer L, we must take an input and generate a sample from layers 1 to $L-1$. Then at layer L we perform operations given by Equation 7.18 with $G = 1$. Finally a weight matrix update of cost $3HI + 4H + 4I$ operations is needed. All total, to perform a single pre-train to layer L requires

$$\begin{aligned}\mathcal{N} = \left[\sum_{l=1}^{l=L-1} (2I_lH_l + H_lR + H_lT)\right] \\ + \left[G\left[4I_LH_L + (H_L + I_L)(R+T)\right] + 2I_LH_L\right] \\ + \left[3H_LI_L + 4H_L + 4I_L\right]\end{aligned}$$

If we are running E epochs of N samples each we must multiply this value by EN.

To run a tuning phase, we perform a single up pass with weight adjustments, assuming $G = 1$. This can be approximated as a summation of operations given by 7.18 summed up from $l = 1$ to L then with the weight adjustments as well

$$\sum_{l=1}^{l=L} \left[\left[4I_lH_l + (H_l + I_l)(R+T)\right] + 2I_lH_l + 3H_lI_l + 4H_l + 4I_l\right] \tag{7.19}$$

In the down pass we have a different value for G, and we also note that only the bottom layers are effected, so the upper bound on the summation is L_u.

$$\sum_{l=1}^{l=L_u} \left[G\left[4I_lH_l + (H_l + I_l)(R+T)\right] + 2I_lH_l + 3H_lI_l + 4H_l + 4I_l\right] \tag{7.20}$$

Experimentally, we found R requires roughly 10 operations, equivalent to a T operation.

Table 7.5: Summary of DBN computational cost pre-training (Hinton MNIST)

Layer	Inputs	Units	MFlops	All Epochs (GFlops)
1-2	768	500	3	460
2-3	500	500	3	403
3-4	500	2000	10	1366
4-5	2000	10	4	469
TOTAL	-	-	-	2698

Table 7.6: Summary of DBN computational cost tuning first 100 epochs (Hinton MNIST)

Layer	Inputs	Units	UpPhaseMF	DownPhaseMF	All Epochs (GFlops)
1-2	768	500	3	7	44421
2-3	500	500	2	4	28987
3-4	500	2000	9	0	39864
4-5	2000	10	0	0	1004
TOTAL	-	-	-	-	114276

To estimate the computational cost for the network described by Hinton et al. (2006), we group the layers as $1-2$, $2-3$, etc for easier understanding. In Table 7.5 we show the cost to pretrain each layer, assuming 30 epochs with 4400 samples. In Tables 7.6, 7.7, and 7.8 we show the cost to do the tuning with 44,000 images, with $G = 3$ for the first 100 epochs, $G = 6$ for the second 100, and $G = 10$ for the final 100. Finally we show the cost for tuning with 60,000 images and 60 epochs for $G = 10$, presumably the value used for the last set of fine-tuning epochs, in Table 7.9. Note that the value of $L_u = 2$.

Finally, we find the cost to actually perform an evaluation of an input sample. There are several ways of performing this evaluation, as described in Hinton et al. (2006), but the method which they indicated worked the best from performance and simplicity was to use the probabilities of activation in each unit instead of a stochastic method (which requires averaging and therefore multiple passes). Thus an evaluation is simply a matter of performing a sampling as before, except with no random component required as this is replaced by multiplications. Thus, for a single layer the cost is $2IH + HT + H$ and this is shown in Table 7.10.

Table 7.7: Summary of DBN computational cost tuning second 100 epochs (Hinton MNIST)

Layer	Inputs	Units	UpPhaseMF	DownPhaseMF	All Epochs (GFlops)
1-2	768	500	3	11	65031
2-3	500	500	2	7	42451
3-4	500	2000	9	0	39864
4-5	2000	10	0	0	1004
TOTAL	-	-	-	-	148350

Table 7.8: Summary of DBN computational cost tuning third 100 epochs (Hinton MNIST)

Layer	Inputs	Units	UpPhaseMF	DownPhaseMF	All Epochs (GFlops)
1-2	768	500	3	18	92510
2-3	500	500	2	11	60403
3-4	500	2000	9	0	39864
4-5	2000	10	0	0	1004
TOTAL	-	-	-	-	193781

Table 7.9: Summary of DBN computational cost tunining, final 60 epochs (Hinton MNIST)

Layer	Inputs	Units	UpPhaseMF	DownPhaseMF	All Epochs (GFlops)
1-2	768	500	3	18	75690
2-3	500	500	2	11	49421
3-4	500	2000	9	0	32616
4-5	2000	10	0	0	822
TOTAL	-	-	-	-	158549

Table 7.10: Summary of DBN computational cost post-training (Hinton MNIST)

Layer	Inputs	Units	MFlops
1-2	768	500	0.774
2-3	500	500	0.5055
3-4	500	2000	2.022
4-5	2000	10	0.040
TOTAL	--	--	3.34

7.1.4 Summary of computational resources

We used our implementation of DeSTIN to provide guidelines regarding the processing time and computational load of the system. We also summarized and compared similar parameters for the CNN and DBN deep learning architectures, based on seminal papers on these technologies which also used the same MNIST data set. In our approach we established "design equations" for these methods, then compared our actual load based on our experimental results and the results reported in the literature.

The supervised learning system we used for our DeSTIN results is the largest computational element here, requiring roughly 15 TFlops, dwarfing the feature extraction training cost which is only 290 GFlops for the tabular method or 452 GFlops for the function approximation method; in contrast, the CNN method requires approximately 1 TFlop. The DBN on the other hand seems to require over 600 TFlops. Once the networks are trained, DeSTIN requires the most computations to produce an output at 100 MFlops, with the DBN requiring 3.34 MFlops and the CNN requiring 0.6 MFlops. Thus, the DeSTIN computational load is larger than the DBN and CNN for running responses after training the feature extractor. However, we note that this is largely because most of our experiments were done using values for A, K, and L which are fairly large. We saw in Chapter 6 that we could achieve good results with smaller values, so it is possible that a more fair comparison can be made, but we preferred to use our "default" setting here.

These results suggest we have not accomplished our goal of creating an architecure suitable for online learning, but in reality we should point out that our main goal here was to focus and constrain ourselves to this domain for the feature generation. Incorporating a supervised learning system more directly could save significant computation, as well as trying to perform other optimizations. We also should point out that the DeSTIN topology used here is very generic, and was used for all the data sets from Chapter 6. By the same token, the training procedure followed by the DBN of Hinton et al. (2006) seems customized for the problem, although in fairness their training could have been terminated at the pre-training phase, which still yielded an error rate of 2.48 % or 97.51 % accuracy.

7.2 Noise analysis

There are three main areas where noise in the system can cause degradation in performance. First, in the clustering itself, a system trained with a certain noise level which is subjected to signals with more noise will began to mis-identify observations as different cluster labels. Second, the exponential distribution of distances to cluster centroids which has a learned standard deviation will began to become "less true", as more observations fall at less likely distances than expected. Third, the $P^a_{ss'}$ table or function approximation will begin to yield less likely beliefs, as the transitions from state to state with each advice become less like the learned values during network training.

There are a few advantages to DeSTIN which combat this degradation. First, the multple observations through different spatial shifts amount to different signal observations and thus can combat noise; if the noise is independent from observation to observation, the noise immunity should fall as $\frac{1}{\sqrt{N}}$. A second strategy employed by DeSTIN is the use of multiple nodes; again, if the noise is independent from node to node, the immunity again gains by the same factor. However, we must note that there is a degree of correlation between observation and node, especially since the observations are mostly spatially shifted versions of one another and the belief of a node is dependent on the beliefs of its children.

Thus for this analysis, we will investigate the degradation at a single node through the three main mechanisms of: clustering errors; the degradation in the integrity of an exponential PDF in the presence of noise; and the degradation in the dynamic system in the presence of noise. We will then use these results and experimental analysis on the MNIST data set to estimate the performance of DeSTIN in the presence of noise, then follow with experimental results with different noise levels on a full DeSTIN and supervised learning stage.

7.2.1 Clustering

The clustering performance is definitely degraded by noise, as the composite mean-square-error becomes larger when more noise is added to a system. Note that the supposition here is that the DeSTIN network is trained without noise to extract features, and then the response to noisy signals is obtained and used in supervised learning. Thus, we expect that added noise will cause observations to drift from the "true" cluster and misclassifications will occur.

However, we note that the DeSTIN mechanism of approximating the distances to the centroid library helps reduce the effect of these errors. In effect this mechanism "fuzzifies" the belief value by using the distance computed to ALL centroids in the library, as opposed to simply taking the winner as a binary label. The exact effect of this quality is difficult to quantify as it depends very highly on the initial data spread, the number of centroids used, the feature dimensionality, and the noise levels. For illustration purposes, however, we performed a simple experiment where a 2D data set of 4 distinct classes was clustered and then data was presented with different noise levels. At each noise level, we score the performance of a binary classification measuring similarity to each library member against a system using an estimate of the similarity as $S = 1 - d$. We use the data set from the first set of experiments from Chapter 4, which is repeated here for convenience. In this case there are 4 clusters in 2D space and the data set uses Gaussian noise with a standard deviation of 0.10, illustrated in Figure 7.1 with the derived exponentials shown in Figure 7.2. The estimated exponential distributions of the distance from this case use a λ of 771, 154, 117, and 343 each.

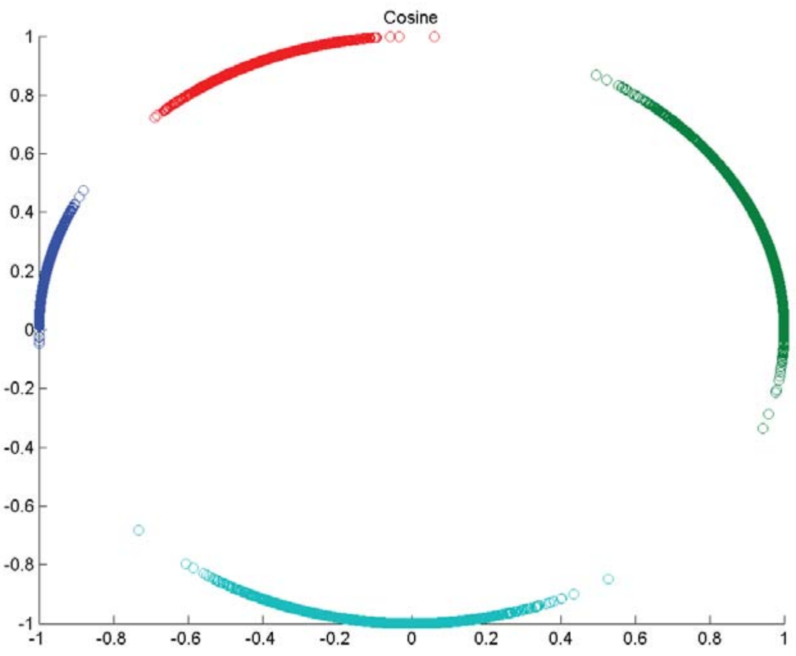


Figure 7.1: Data and centroids of test case to illustrate cluster noise immunity by distance modeling.

For our comparison, at each additive noise level from 0 to 2.0 in steps of 0.1 we evaluated 1000 samples from each centroid class. The basic "winner take all" clustering was performed to achieve a binary result, then for comparison the similarity metric was used and normalized, then the mean-square-error between the output vectors and the ideal case (a vector with a single non-zero element, $v_i = 1$ for $\omega = i$). This was then averaged across all the samples. The result is shown in Figure 7.3. We see that at low noise levels the error from the single non-zero element case is, in fact, smaller as expected since so long as the classifications are correct they will be absolutely correct, with no residual error. At higher noise levels, however, the distance-based case is more robust and yields a better result. The impact of this in DeSTIN is the means of allowing softer decisions which over multiple observations

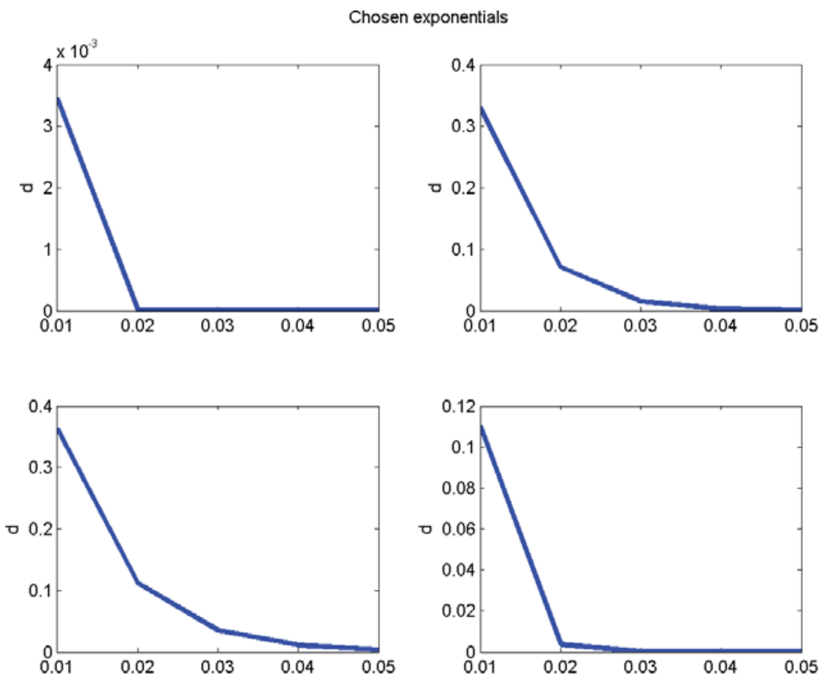


Figure 7.2: Estimated distributions for sample case.

have a higher chance of yielding meaningful features and more reliable classifications at the supervised learning level.

7.2.2 Estimates of $P(o|s)$

The exponential distribution we use to model the $P(o|s)$ component of Equation 3.13 uses a single parameter, λ, which is estimated during the training procedure for the network. As a result, intuitively when the observations are made that are from the original estimate of the exponential distribution, fit that exponential distribution, the PDF of the samples are exactly the distribution of choice. However, when the observations have changed to a noisier situation, the PDF of the new observations changes. Therefore, we would like to compute the PDF of the new observation space. Our original $P(o|s)$ estimate is made by using a small region in the cumulative distribution, such that $P(o|s)$ is given as

$$P(o = d|s) = \int_{d-\delta}^{d+\delta} \lambda_s e^{-\lambda_s x} dx \tag{7.21}$$

$$P(o = d|s) = -e^{-\lambda_s x}\Big|_{d-\delta}^{d+\delta} \tag{7.22}$$

$$P(o|s) = -\exp\left(-\lambda_s(d+\delta)\right) + \exp\left(-\lambda_s(d-\delta)\right) \tag{7.23}$$

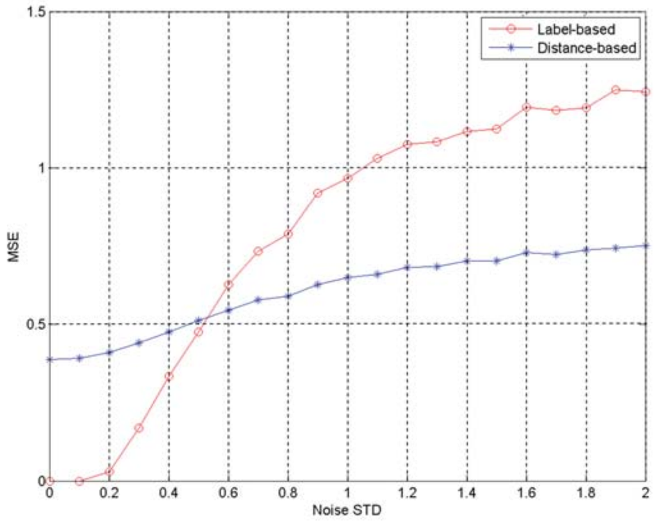


Figure 7.3: Mean-square-error at different noise levels for hard-threshold on distributions for sample case.

$$P(o|s) = \exp\left(-\lambda_s d\right)\left[-\exp\left(-\lambda_s \delta\right) + \exp\left(\lambda_s \delta\right)\right] \tag{7.24}$$

We note that the PDF of this signal is simply a scaled exponential distribution given that we use a fixed δ. We show some examples in Figure 7.4.

We can model additive noise through a very simple mechanism here to a first-order approximation. Let the noise be denoted as an additive value to the true distance which we will denote here as ω. Thus, the probability is now

$$P(d+\omega|s) = \exp\left(-\lambda_s d + \omega\right)\left[-\exp\left(-\lambda_s \delta\right) + \exp\left(\lambda_s \delta\right)\right] \tag{7.25}$$

$$P(d+\omega|s) = \exp\left(-\lambda_s \omega\right) P(d|s) \tag{7.26}$$

Thus, the noisy probability is simply the original probablity scaled by a factor that is a function of the original estimated model parameter λ_s. Note that the additive factor ω is the result of the original additive noise, and its exact form again depends on the centroids used. We also note the dependence on the original model parameter λ_s is such that when λ_s is larger, the scaling factor gets smaller, and thus reduces the overall probability of seeing a particular observation; the noise corrupts the system into thinking common occurrences are actually rarities, so the overall probability goes down. To some degree this is mitigated by some of the normalization factors.

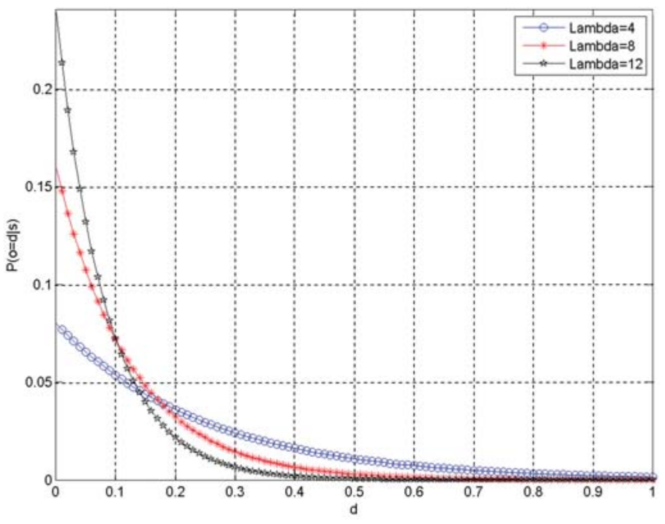


Figure 7.4: Example of the probability of a particular distance with different lambda values and delta set to 0.01.

We can attempt to get a bound on this additive noise by performing a transformation of random variables. We effectively have a relationship given as

$$y = \exp\left(-\lambda_s x\right) \tag{7.27}$$

In this equation x is a random variable which has been transformed through an exponential function. Solve this for x to get

$$x = -\frac{\ln\left(y\right)}{\lambda_s} \tag{7.28}$$

To find the $P(Y \leq y)$ we integrate over the distribution of x, which we will assume is an exponential distribution with parameter λ_2. Since y is decreasing, we establish the lower bound with our definition and the upper bound as ∞. We therefore see that

$$P\left(Y \leq y\right) = P\left(x \geq -\frac{\ln y}{\lambda_s}\right) \tag{7.29}$$

This is written in integral form as

$$P\left(Y \leq y\right) = \int_{-\frac{\ln y}{\lambda_s}}^{\infty} \lambda_2 e^{-\lambda_2 x} dx \tag{7.30}$$

Solve the integral, and noting that λ_s is always positive, to get

$$P\left(Y \le y\right) = -e^{-\lambda_2 x}\Big|_{-\frac{\ln y}{\lambda_s}}^{\infty} \tag{7.31}$$

$$P\left(Y \le y\right) = \exp\left(\frac{\lambda_2}{\lambda_s}\ln y\right) \tag{7.32}$$

The PDF is found by differentiating:

$$p\left(y\right) = \frac{\lambda_2}{y\lambda_s}\exp\left(\frac{\lambda_2}{\lambda_s}\ln y\right) \tag{7.33}$$

Note that the exponent is always negative, since $0 \le y \le 1$. We can find the expected value also by integrating:

$$E\left(y\right) = \int_0^1 y\left(\frac{\lambda_2}{y\lambda_s}\exp\left(\frac{\lambda_2}{\lambda_s}\ln y\right)\right)dy \tag{7.34}$$

$$E\left(y\right) = \int_0^1 y\left(\frac{\lambda_2}{y\lambda_s}\exp\left(\ln y^{\frac{\lambda_2}{\lambda_s}}\right)\right)dy \tag{7.35}$$

$$E\left(y\right) = \int_0^1 y\left(\frac{\lambda_2}{y\lambda_s}y^{\frac{\lambda_2}{\lambda_s}}\right)dy \tag{7.36}$$

$$E\left(y\right) = \frac{\lambda_2}{\lambda_s}\int_0^1 y^{\frac{\lambda_2}{\lambda_s}}dy \tag{7.37}$$

$$E\left(y\right) = \frac{\lambda_2}{\lambda_s}\frac{1}{1+\frac{\lambda_2}{\lambda_s}}\,y^{\frac{\lambda_2}{\lambda_s}}\Big|_0^1 \tag{7.38}$$

$$E\left(y\right) = \frac{\lambda_2}{\lambda_2+\lambda_s} \tag{7.39}$$

When $\lambda_2 = \lambda_s$ the result is simply $P\left(Y \le y\right) = y$, which means the PDF of y is a uniform distribution and overall the mean of the distribution is 0.5. When λ_2 is less than λ_s, the mean is closer to 0 which makes sense since the noise is greater (recall the mean of an exponential distribution with parameter λ is $\frac{1}{\lambda}$). Finally, when $\lambda_2 > \lambda_s$, the expectation tends to be greater than 0.5, up to a maximum of 1.0 which means the noise is negligible.

7.2.3 Propagation through $P_{ss'}^{a}$ mechanisms

In this section we consider the system dynamics and normalization factors, and summarize their susceptibility to noise processes. As shown in Chapter 5, the belief in each state vector $b(s)$ is then propagated through a probability transition matrix or equivalent function which we have termed $P_{ss'}^{a}$ for brevity. The $P_{ss'}^{a}$ calculation is denoted in the following manner

$$b = \mathbf{P_{os}} \mathbf{P^{a}_{ss'}} b' \tag{7.40}$$

where $\mathbf{P_{os}}$ is the vector of probabilities $P(o|s)$ calculated as above with a scalar "noise" factor for each element. The overall vector with noise has the form

$$\mathbf{P_{d|s}} = \begin{pmatrix} \exp(-\lambda_1\omega_1)P(d_1|s_1) \\ \exp(-\lambda_2\omega_2)P(d_2|s_2) \\ \vdots \\ \exp(-\lambda_m\omega_m)P(d_m|s_m) \end{pmatrix} \tag{7.41}$$

As the final analysis step, we compile the composite expression $B(a)$ by the summation of the beliefs found for a given advice a across all possible states at movement m as stated in Equation 5.7. The effect here is again a composite scaling, but the factor is a weighted summation (with weights given by the noise terms $\exp(-\lambda_s\omega_s)$ and thus cannot be generalized with any added insight. However, we can consider extreme cases, where the belief is noise-free and simplistic compared to with a noisy situation. In the case of a noise-free, clear belief, the state-to-state transition is absolute, with a probability of 1 for the transition from state s_{m-1} to state s_m and probability of 0 (or a small non-zero value ϵ for regularization) for all other possible transitions. The new belief will therefore be essentially equal to the previous belief, with $P(d|s_m)$ as the dominant belief. As this accumulates for all movements, the $B(a)$ term for the "correct" advice becomes dominant. However, with noise, the value of $P(d|s_m)$ is reduced by the exponential scaling factor we found earlier. If no other beliefs are possible, based on past learning for the $P^a_{ss'}$ transition matrix, then the normalization effect will effectively remove the noise for this particular advice state! In this case, the past learned experience is the dominant factor, and can be regarded as a learned prior with 100% probability. Unfortunately, this will only be effective so long as all other advice states have a zero probability of occuring for this set of $P(d|s)$ values. For other advice states, the normalization actually works against us, as the beliefs tend to become smaller (since the exponential scaling factor reduces the value of $P(d|s)$ for all states) and thus the normalization tends them toward uniformity. Thus the resulting features have less value as they no longer embody the learned states associated with the sequence of observations, and performance is degraded.

7.2.4 Experimental Results

The MNIST images were used for these experiments. For each image, Gaussian-distributed noise was added with zero mean and a standard deviation ranging from 0.3 to 4.0, then the images were thresholded to a range of 0 to 1. There are not really very robust means of specifying the true signal to noise ratio in images, since image interpretation is highly subjective Winkler and Mohandas (2008), but the use of mean-square-error or MSE and associated power-signal noise ratio (PSNR) are commonly used. The MSE was computed on a per-image basis using Equation 7.42.

$$y = \sum_{j=1}^{m} \sum_{i=1}^{n} (o_i - w_{ni})^2 \tag{7.42}$$

We then averaged the MSE across the entire training set for the composite MSE value. Note that for additive noise of 0 standard deviation, the MSE will also be 0.

For noise analysis, a network was created using evolving clusters with 25 maximum as discussed earlier. We then performed a variety of tests where the source images from the MNIST dataset were corrupted with additive noise applied to both the training and testing images. The noise was added as Gaussian noise, zero mean, with varying values of sigma. The values were thresholded so that the signal ranged from 0 to 1, where values less than 0 were set to 0 and values greater than 1 were set to 1. The resulting noise level was characterized by the mean-square-error from the original image level, with the average taken across all images in the MNIST training set. The results are shown in Table 7.11 for a sample DeSTIN network with the neural network ensemble supervised learning system. In addition, we show the results of a kNN approach using the same DeSTIN features, and also the results using the raw image data alone for comparison. The results are plotted in Figure 7.10.

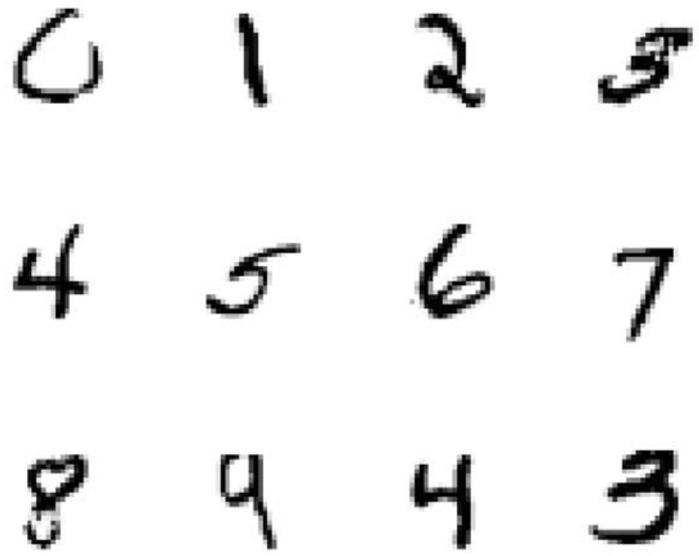


Figure 7.5: Example MNIST images with no added noise.

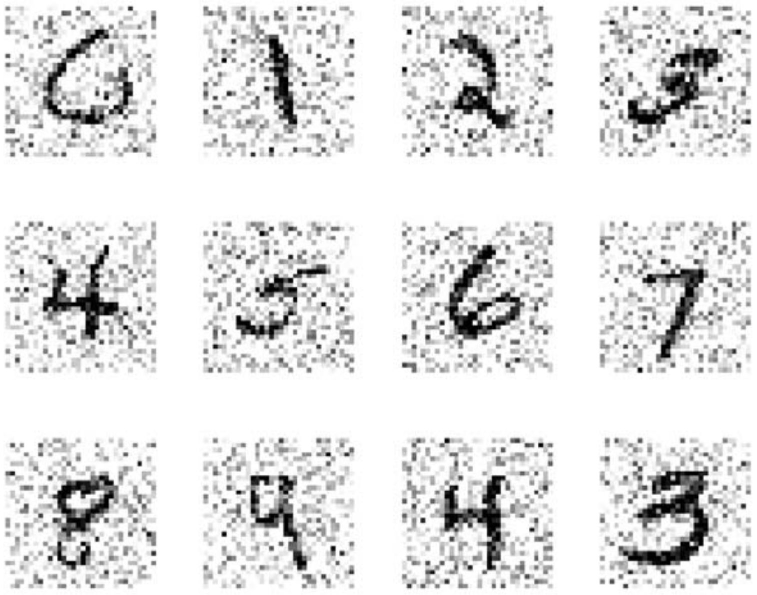


Figure 7.6: Example MNIST images with added noise STD 0.3.

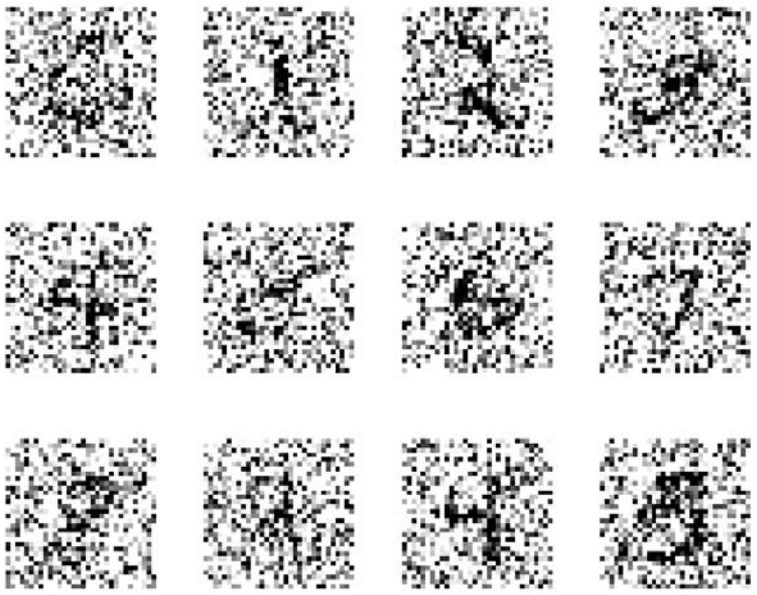


Figure 7.7: Example MNIST images with added noise STD 0.7.

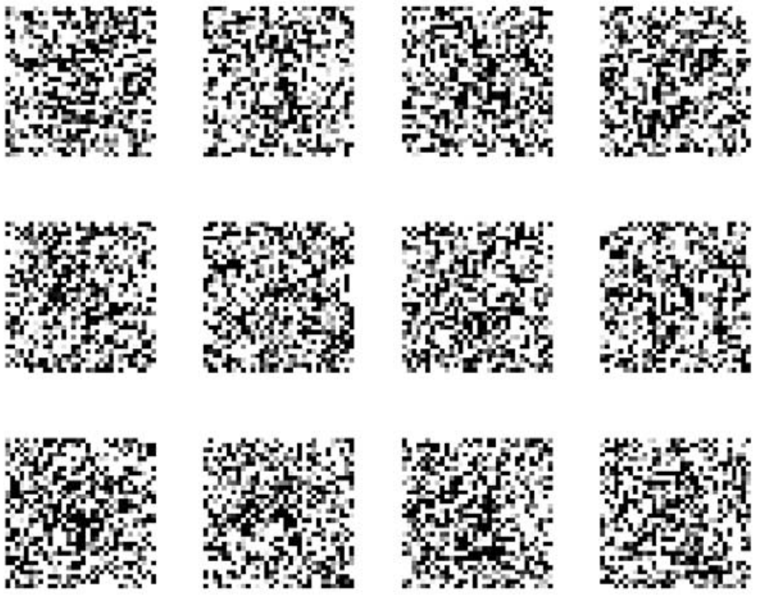


Figure 7.8: Example MNIST images with added noise STD 1.6.

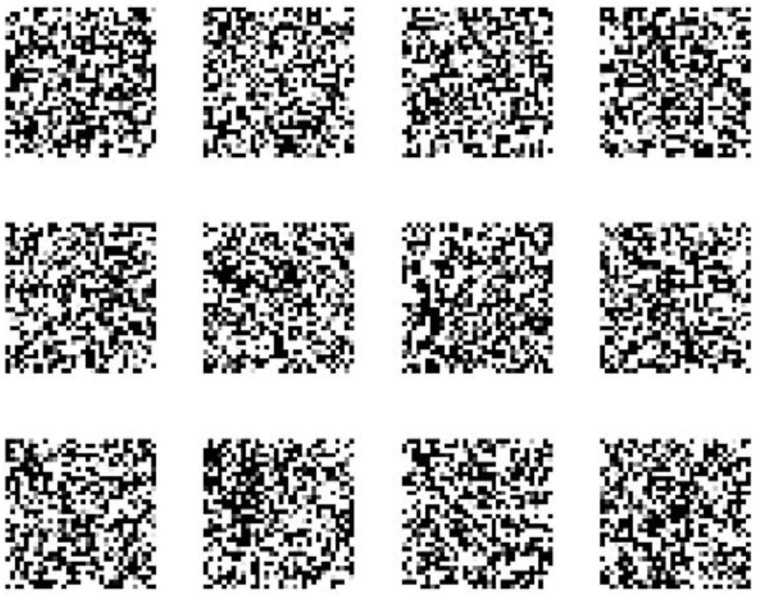


Figure 7.9: Example MNIST images with added noise STD 4.0.

Table 7.11: Summary of noise analysis study

STD	DeSTIN NN	DeSTIN kNN	Image kNN	MSE
0	98.26	96.574	94.527	0
0.3	96.68	93.1233	88.9	0.0434905
0.5	93.17	88.735	80.78	0.10842
0.6	89.9	85.765	72.5	0.14104
0.7	87.4	81.67	68.1	0.170584
0.8	83.78	77.4667	60.25	0.196356
0.9	80.14	72.6733	54.8	0.21869
1	75.48	67.9433	48.6	0.237925
1.2	66.71	58.2892	42	0.269062
1.4	59.26	50.3512	34.8	0.292909
1.6	52.49	43.965	29.3	0.31163
1.8	46.76	38.3475	25.2	0.326546
2	42.15	33.4323	21.4	0.338739
4	23.4	17.238	14.1	0.395787

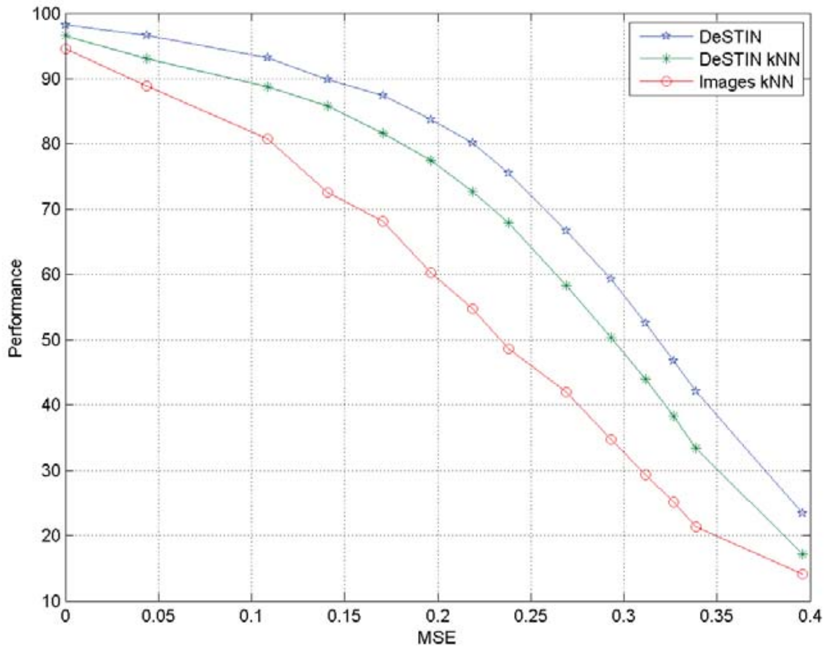


Figure 7.10: Performance of DeSTIN supervised learning with increasing image noise.

7.2.5 Summary of Noise Analysis

From our analysis and results, we see that there is some susceptibility to noise in the DeSTIN architecture. We traced the response of a single node in the presence of noise through the belief formulation process, and found the main effect is in the reduction of the $P(o|s)$ term which propagates to the $B(a)$ output value for each node. The multiple observer model, through the multiple movements at a single node and then through the use of multiple nodes, should provide some level of noise resistance and our experimental results seem to validate this, where the kNN classification of DeSTIN features outperformed the kNN on raw image data especially in the moderate noise levels (where the MSE was between 0.15 to 0.25). Visually, noise levels above this made a classification by human observer very difficult. However, the noise reduction obtained was not nearly as robust as predicted by a $\frac{1}{\sqrt{MN}}$ result, which is largely due to correlations in the noise and signal from observation to observation.

The analysis performed here was fairly rudimentary, but it highlights the mechanisms where noise in the DeSTIN architecture can cause issues and means by which DeSTIN mitigates these effects. We note that further experiments and a more rigorous analysis are certainly possible, but we also note that there are additional mechanisms which could be undertaken to help reduce the effect of noise even further, in particular a non-fixed number of movements which would allow longer "staring" times which could permit better feature extraction and decision making, especially when coupled with a supervised learning system providing online feedback on the quality of the classification.

Chapter 8

Conclusions

8.1 Summary

In this work we have presented the Deep Spatio-Temporal INference Network or DeSTIN, a biologically-inspired, unsupervised feature extraction deep learning architecture, that uses multiple layers of identical circuits or nodes. DeSTIN learns feature representations from repeated observations of spatial patterns that are changed over time either through active scanning (for the results shown in this dissertation) or through other forms of motion. We introduced the basic conceptual formulation of DeSTIN, along with background on other deep learning architectures in the literature. We presented a variety of details on DeSTIN, including its basic implementation details, the static elements of the DeSTIN method including online clustering and the use of probabilistic mixture models in our belief formulation process, the dynamic elements of the DeSTIN process from both a tabular and function approximation approach, the use of online clustering for advice formulation from parental nodes, and a multiple observer model for the creating of a single temporal representation for a node's belief states. We then presented methods for the selection of nodes for supervised learning, and showed the results of the DeSTIN architecture on the MNIST data set (for handwritten digit recognition), the CBCL Face Database 1 from the MIT Center for Biological and Computational Learning (for face detection), and an optic nerve detection task (using the Messidor data set and a private set) . We found that DeSTIN compared favorably to state-of-the-art results in the literature, with little or no tuning for these domains. Finally, we performed an analysis of the effects of noise on the DeSTIN architecture and compared the computational load of DeSTIN to other deep learning methods.

The main DeSTIN software is written in C++ and can be compiled under both Microsoft Windows Visual Studio and Linux environments. The core software consists of a series of classes which include a clustering engine, a network node, a network layer, a raw data file format, and various other utility classes needed to implement a DeSTIN network. The function of the software is to produce a trained DeSTIN network with a given hierarchical structure using a training data set, and generate the network output responses to a testing data set. These output responses are then used with a group of MATLAB scripts to generate supervised learning performance results. Much of the software is being vetted and developed for more general use by the OpenCOG software community Foundation (2012), with a broad release targeted at the end of 2012.

8.2 Future Directions

Overall, DeSTIN is a fertile ground for further research efforts in deep machine learning. In addition, there are many areas of general machine learning that could be valuable research topics in DeSTIN, both advancing the architecture itself and the body of research for these areas.

8.2.1 Similarity measures and data representation

In this work we used the basic cosine similarity measurement to establish the similarity of observations to one another and to our basic prototypes of our clustering representation of the data. Other metrics could be used, and indeed in conjunction with different data representation methods besides clustering may prove beneficial. We envision a system where the actual concept of similarity can be dynamically learned from some family of possible functions or kernel methods, in particular where the concept of temporal closeness can be used to constrain an optimization process. In addition, while we chose to represent our belief "state" using online clustering, there are other data representation methods that could be utilized as well, including various factor analysis methods and even other clustering mechanisms with different objective functions. This is a fertile field for additional improvements to the DeSTIN architecture and methods.

8.2.2 System Dynamics

The system dynamics of the DeSTIN method were explored through initial temporal sampling and later through a multiple-observer model where the belief in an "advice state" was created over multiple movements or observations. In our work we utilized a fixed pattern of temporal scans, and used the entire pre-known scan sequence to create the advice which was generated by a parent and projected down to child nodes after each set of scans was completed. An interesting alternative could be to generate the scan sequence on-the-fly and create the advice on-the-fly, as we originally intended in our earlier DeSTIN formulations. The evolving belief formulation could then attempt to modify the scan itself using some kind of "action" rule as in reinforcement learning, with a goal of reaching a conclusion as early as possible, or with a certain level of confidence.

8.2.3 Supervised learning

We believe the supervised learning phase of DeSTIN can be directly incorporated into the architecture, much as it is done for CNNs and DBNs. This would be a very valuable contribution as our current approach is essentially an unsupervised feature extraction method that does not draw from all known characteristics of the data to optimize the feature extraction process. There are a few easily obvious methods for incorporating supervised learning. One method is to use the supervision as labels that replace or augment the advice during the training process. The advice formulation in our work here could then be used for a joint probability where unsupervised learning generates state labels as we do here, but the supervised label represents an additional "dimension" in a joint probability. Another method could be to use the labels as an additional constraint in the data representation process, where the clustering process distance or similarity measurement could be influenced by supervised learning. The use of partially labeled data (for semi-supervised learning) would be a fruitful area of research as well.

8.2.4 Control problems

We have investigated the use of DeSTIN in the context of image classification, but the signals and learning methods used here could also be adapted for control. In this operation the decisions made from the features learned could be used to guide a control process, and thus DeSTIN could fit into a robotics or other RL scenario as a feature extractor. This type of application could utilize the other future directions mentioned here as well.

8.2.5 Other inputs

While there are advantages to working with images from the "raw pixel" level, other representations may be useful. The SIFT method and its relatives have been shown to have good generalization properties; we envision that DeSTIN could be constructed to work with these representations instead, or in addition to, the raw pixel data itself. Other image processing algorithms such as edge detection and thresholding could be simple, generic early pre-processing operations that could be combined with DeSTIN to produce better results or extend the capabilities of the architecture. This may actually represent a more biologically-inspired approach, since there is evidence from neuroscience that the human visual system performs some basic processing steps which are inherent to the mammalian sensory process (i.e., they are an evolved product of biology as opposed to a learned process).

Finally, DeSTIN itself need not be limited to image processing. Other types of input data, such as acoustic or hyperspectral signals, or sensory arrays featuring a variety of different signal types, could be interesting fields of application for DeSTIN as well. There may be some required changes in the topology, since some of these fields may not actually have a “spatial” nature, but this general topic is worth future investigation.

Bibliography

Bibliography

Abràmoff, M., Garvin, M., and Sonka, M. (2010). Retinal imaging and image analysis. *Biomedical Engineering, IEEE Reviews in*, 3:169–208.

Ackley, D., Hinton, G., and Sejnowski, T. (1985). A learning algorithm for boltzmann machines*. *Cognitive science*, 9(1):147–169.

Adelson, E., Anderson, C., Bergen, J., Burt, P., and Ogden, J. (1984). Pyramid methods in image processing. *RCA engineer*, 29(6):33–41.

Adler, A. and Schuckers, M. (2007). Comparing human and automatic face recognition performance. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 37(5):1248–1255.

Arel, I., Rose, D., and Coop, R. (2009a). DeSTIN: A Scalable Deep Learning Architecture with Application to High-Dimensional Robust Pattern Recognition. In *Proc. of the AAAI 2009 Fall Symposium on Biologically Inspired Cognitive Architectures (BICA)*.

Arel, I., Rose, D., and Karnowski, T. (2009b). A Deep Learning Architecture Comprising Homogeneous Cortical Circuits for Scalable Spatiotemporal Pattern Inference. In *NIPS 2009 Workshop on Deep Learning for Speech Recognition and Related Applications*.

Arel, I., Rose, D., and Karnowski, T. (2010). Deep Machine Learning - A New Frontier in Artificial Intelligence Research. *IEEE Computational Intelligence Magazine*, 5(6):15–43.

Banerjee, A. and Ghosh, J. (2004). Frequency-sensitive competitive learning for scalable balanced clustering on high-dimensional hyperspheres. *Neural Networks, IEEE Transactions on*, 15(3):702–719.

Bar-Cohen, Y. (2006). BiomimeticsâĂŤusing nature to inspire human innovation. *Bioinspiration & Biomimetics*, 1:P1.

Bay, H., Ess, A., Tuytelaars, T., and Van Gool, L. (2008). Speeded-up robust features (surf). *Computer Vision and Image Understanding*, 110(3):346–359.

Behnke, S. (2003). *Hierarchical neural networks for image interpretation.* Springer-Verlag New York Inc.

Bellman, R. (1957). *Dynamic Programming.* Princeton University Press.

Bengio, Y. (2009). *Learning deep architectures for AI.* Now Publishers Inc. 2, 19, 24, 26, 33, 198

Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. (2007a). Greedy layer-wise training of deep networks. In *Advances in Neural Information Processing Systems 19: Proceedings of the 2006 Conference*, page 153. The MIT Press.

Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. (2007b). Greedy layer-wise training of deep networks. In *Advances in Neural Information Processing Systems 19: Proceedings of the 2006 Conference*, page 153. The MIT Press.

Bishop, C. et al. (2006). *Pattern recognition and machine learning.* Springer New York:.

Bister, M., Cornelis, J., and Rosenfeld, A. (1990). A critical view of pyramid segmentation algorithms. *Pattern Recognition Letters*, 11(9):605–617.

Bottou, L. (1995). Convergence properties of the K-means algorithm. *Advances in Neural Information Processing Systems*, 7.

Bourlard, H. and Kamp, Y. (1988). Auto-association by multilayer perceptrons and singular value decomposition. *Biological cybernetics*, 59(4):291–294.

Bradski, G. and Kaehler, A. (2008). *Learning OpenCV: Computer vision with the OpenCV library.* O'Reilly Media.

Burt, P., Hong, T., and Rosenfeld, A. (1981). Segmentation and estimation of image region properties through cooperative hierarchial computation. *Systems, Man and Cybernetics, IEEE Transactions on*, 11(12):802–809.

Chen, Y., Han, C., Wang, C., Jeng, B., and Fan, K. (2006). The application of a convolution neural network on face and license plate detection. In *Pattern Recognition, 2006. ICPR 2006. 18th International Conference on*, volume 3.

Chinrungrueng, C. (1995). Optimal adaptive k-means algorithm with dynamic adjustment of learning rate. *IEEE Transactions on Neural Networks*, 6(1):157.

Chou, P., Rao, A., Sturzenbecker, M., Wu, F., and Brecher, V. (1997). Automatic defect classification for semiconductor manufacturing. *Machine Vision and Applications*, 9(4):201–214.

Coates, A. and Ng, A. (2011). The importance of encoding versus training with sparse coding and vector quantization. In *Proc. of the 28th Int. Conf. on Machine Learning*.

Cooper, G. and McGillem, C. (1999). *Probabilistic methods of signal and system analysis*. Oxford University Press, USA.

Cover, T. and Thomas, J. (2006). *Elements of information theory*. John Wiley and sons.

Dempster, A., Laird, N., and Rubin, D. (1977). Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 1–38.

Deubel, H. and Schneider, W. (1996). Saccade target selection and object recognition: Evidence for a common attentional mechanism. *Vision Research*, 36(12):1827–1837.

Dickey, F. and Romero, L. (1991). Normalized correlation for pattern recognition. *Optics letters*, 16(15):1186–1188.

Ding, C. (1999). A similarity-based probability model for latent semantic indexing. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 58–65. ACM.

Dreyfus, H. and Dreyfus, S. (1988). Making a mind versus modeling the brain: Artificial intelligence back at a branchpoint. *Daedalus*, 117(1):15–43.

Duda, R., Hart, P., and Stork, D. (2001). *Pattern classification*. Citeseer.

Dy, J. (2008). Unsupervised feature selection. *Computational Methods of Feature Selection*, pages 19–39.

Erhan, D., Bengio, Y., Courville, A., Manzagol, P., Vincent, P., and Bengio, S. (2010). Why Does Unsupervised Pre-training Help Deep Learning? *The Journal of Machine Learning Research*, 11:625–660.

Foundation, O. (2012). Opencog foundation. http://opencog.org.

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

Fukushima, K. (2003). Neocognitron for handwritten digit recognition. *Neurocomputing*, 51(1):161–180.

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

Gabor, D. (1972). Holography, 1948-1971. *Proceedings of the IEEE*, 60(6):655–668.

George, D. (2008). *How the brain might work: A hierarchical and temporal model for learning and recognition*. PhD thesis, Stanford University.

Giancardo, L., Abramoff, M., Chaum, E., Karnowski, T., Meriaudeau, F., and Tobin, K. (2008). Elliptical local vessel density: a fast and robust quality metric for retinal images. In *Engineering in Medicine and Biology Society, 2008. EMBS 2008. 30th Annual International Conference of the IEEE*, pages 3534–3537. IEEE.

Giancardo, L., Meriaudeau, F., Karnowski, T., Li, Y., Tobin, K., and Chaum, E. (2011). Microaneurysm detection with radon transform-based classification on retina images. In *Engineering in Medicine and Biology Society, EMBC, 2011 Annual International Conference of the IEEE*, pages 5939–5942. IEEE.

Gonzalez, R. and Richard, E. (2002). Woods, digital image processing.

Guyon, I. and Elisseeff, A. (2003). An introduction to variable and feature selection. *The Journal of Machine Learning Research*, 3:1157–1182.

Harris, C. and Stephens, M. (1988). A combined corner and edge detector. In *Alvey vision conference*, volume 15, page 50. Manchester, UK.

Hastad, J. (1987). *Computational limitations for small depth circuits*. MIT Press.

Hawkins, J. and Blakeslee, S. (2005). *On intelligence*. Owl Books.

Hebb, D. (2002). *The organization of behavior: A neuropsychological theory*. Lawrence Erlbaum.

Heisele, B., Poggio, T., and Pontil, M. (2000). Face detection in still gray images. A.I. memo 1687, Center for Biological and Computational Learning, MIT, Cambridge, MA. xvii, 136, 139, 140, 141, 142, 144, 152

Hertz, J., Krogh, A., and Palmer, R. (1991). *Introduction to the theory of neural computation*. Westview Press.

Hinton, G. (2002). Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800.

Hinton, G. (2007). To recognize shapes, first learn to generate images. *Progress in brain research*, 165:535–547.

Hinton, G., Osindero, S., and Teh, Y. (2006). A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554. 24, 33, 34, 162, 164, 166

Hinton, G. and Salakhutdinov, R. (2006). Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504.

Hong, L., Wan, Y., and Jain, A. (1998). Fingerprint image enhancement: algorithm and performance evaluation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(8):777–789.

Hopfield, J. and Tank, D. (1985). Neural computation of decisions in optimization problems. *Biological cybernetics*, 52(3):141–152.

Horner, J. and Gianino, P. (1984). Phase-only matched filtering. *Applied optics*, 23(6):812–816.

Hornik Maxwell, K. and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366.

Huang, F. and LeCun, Y. (2006). Large-scale learning with svm and convolutional for generic object categorization. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 1.

Hubel, D. and Wiesel, T. (1962). Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of Physiology*, 160(1):106.

Itti, L. and Koch, C. (2000). A saliency-based search mechanism for overt and covert shifts of visual attention. *Vision research*, 40(10-12):1489–1506.

Jain, A. and Farrokhnia, F. (1991). Unsupervised texture segmentation using gabor filters. *Pattern recognition*, 24(12):1167–1186.

Japkowicz, N., Hanson, S., and Gluck, M. (2000). Nonlinear autoassociation is not equivalent to PCA. *Neural Computation*, 12(3):531–545.

Jiang, X. (2009). Feature extraction for image recognition and computer vision. In *Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on*, pages 1–15. IEEE.

Jones, J. and Palmer, L. (1987). An evaluation of the two-dimensional gabor filter model of simple receptive fields in cat striate cortex. *Journal of Neurophysiology*, 58(6):1233–1258.

Karnowski, T., Arel, I., and Rose, D. (2010a). Deep Spatiotemporal Feature Learning with Application to Image Classification. In *2010 9th IEEE International Conference on Machine Learning Applications*, pages 645–650. IEEE.

Karnowski, T., Arel, I., and Rose, D. (2010b). Deep spatiotemporal feature learning with application to image classification. In *Machine Learning and Applications (ICMLA), 2010 Ninth International Conference on*, pages 883–888. IEEE.

Karnowski, T., Aykac, D., Chaum, E., Giancardo, L., Li, Y., Tobin, K., and Abramoff, M. (2009). Practical considerations for optic nerve location in telemedicine. In *Engineering in Medicine and Biology Society, 2009. EMBC 2009. Annual International Conference of the IEEE*, pages 6205–6209. IEEE.

Karpathy, A. (2011). Code for training restricted boltzmann machines and deep belief networks in matlab. http://code.google.com/p/matrbm/.

Kasabov, N. (2003). *Evolving connectionist systems: Methods and applications in bioinformatics, brain study and intelligent machines*. Springer Verlag.

Kasabov, N. and Song, Q. (2002). Denfis: dynamic evolving neural-fuzzy inference system and its application for time-series prediction. *Fuzzy Systems, IEEE Transactions on*, 10(2):144–154.

Kaski, S. and Kohonen, T. (1994). Winner-take-all networks for physiological models of competitive learning. *Neural Networks*, 7(6-7):973–984.

Kauppi, T., Kalesnykiene, V., Kamarainen, J., Lensu, L., Sorri, I., Raninen, A., Voutilainen, R., Uusitalo, H., Kalviainen, H., and Pietila, J. (2007). Diaretdb1 diabetic retinopathy database and evaluation protocol. *Proc. Medical Image Understanding and Analysis (MIUA)*, pages 61–65. xi, 3, 4

Keysers, D. (2007). Comparison and Combination of State-of-the-art Techniques for Handwritten Character Recognition: Topping the MNIST Benchmark. *Arxiv preprint arXiv:0710.2231*.

Kohonen, T. (1990). The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480.

Korn, F., Pagel, B., and Faloutsos, C. (2001). On the "dimensionality curse" and the "self-similarity blessing". *Knowledge and Data Engineering, IEEE Transactions on*, 13(1):96–111.

Kosko, B. (1988). Bidirectional associative memories. *IEEE Transactions on Systems, Man, and Cybernetics*, 18(1):49–60.

Kruskal, J. (1964). Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27.

Kumar, A. (2008). Computer-vision-based fabric defect detection: A survey. *Industrial Electronics, IEEE Transactions on*, 55(1):348–363.

Kwolek, B. (2005). Face Detection Using Convolutional Neural Networks and Gabor Filters. *Artificial Neural Networks: Biological Inspirations–ICANN 2005*, pages 551–556.

Labusch, K., Barth, E., and Martinetz, T. (2008). Simple method for high-performance digit recognition based on sparse coding. *IEEE Transactions on Neural Networks*, 19(11):1985–1989.

Lang, K., Waibel, A., and Hinton, G. (1990). A time-delay neural network architecture for isolated word recognition. *Neural networks*, 3(1):23–43.

Larochelle, H., Erhan, D., Courville, A., Bergstra, J., and Bengio, Y. (2007). An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th International Conference on Machine learning*, page 480. ACM.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324. xii, 20, 22, 23, 38, 158

LeCun, Y. and Cortes, C. (2009). The MNIST database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`.

Lee, H., Grosse, R., Ranganath, R., and Ng, A. (2009). Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 609–616. ACM.

Lee, T. and Mumford, D. (2003). Hierarchical Bayesian inference in the visual cortex. *Journal of the Optical Society of America A*, 20(7):1434–1448.

Lee, T., Mumford, D., Romero, R., and Lamme, V. (1998). The role of the primary visual cortex in higher level vision. *Vision research*, 38(15-16):2429–2454.

Li, Y. (1992). Reforming the theory of invariant moments for pattern recognition. *Pattern Recognition*, 25(7):723–730.

Li, Y., Karnowski, T., Tobin, K., Giancardo, L., Morris, S., Sparrow, S., Garg, S., Fox, K., and Chaum, E. (2011). A health insurance portability and accountability act–compliant ocular telehealth network for the remote diagnosis and management of diabetic retinopathy. *Telemedicine and e-Health*.

Lockett, A. and Miikkulainen, R. (2009). Temporal Convolution Machines for Sequence Learning. *U. Texas Technical Report*.

Long, P. and Servedio, R. (2010). Restricted boltzmann machines are hard to approximately evaluate or simulate. In *Proceedings of the 27th International Conference on Machine Learning*, pages 703–710.

Lowe, D. (2004). Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110.

Lucas, B. and Kanade, T. (1998). with an application to stereo vision. *Proceedings DARPA Image Understanding Workrhop*, pages 121–130.

Majdi-Nasab, N., Analoui, M., and Delp, E. (2006). Decomposing parameters of mixture gaussian model using genetic and maximum likelihood algorithms on dental images. *Pattern recognition letters*, 27(13):1522–1536.

Marr, D. et al. (2010). *Vision: A computational investigation into the human representation and processing of visual information*. MIT Press.

Martinetz, T., Berkovich, S., and Schulten, K. (1993). Neural-gas' network for vector quantization and its application to time-series prediction. *Neural Networks, IEEE Transactions on*, 4(4):558–569.

Mathworks (2012). Matlab kmeans documentation. http://www.mathworks.com/help/toolbox/stats/kmeans.

Mikolajczyk, K. and Schmid, C. (2005). A performance evaluation of local descriptors. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 27(10):1615–1630.

Miller, J. and Lommel, P. (2006). Biomimetic sensory abstraction using hierarchical quilted self-organizing maps. In *Proceedings-SPIE the International Society for Optical Engineering*, volume 6384, page 638.

Minsky, M. and Papert, S. (1969). Perceptrons MIT Press. *Cambridge, Ma*, 10.

Mitchell, T. (1997). Machine learning. WCB. *Mac Graw Hill*, page 368.

Mitra, P., Murthy, C., and Pal, S. (2002). Unsupervised feature selection using feature similarity. *IEEE transactions on pattern analysis and machine intelligence*, 24(3):301–312.

Mobahi, H., Collobert, R., and Weston, J. (2009). Deep learning from temporal coherence in video. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 737–744. ACM.

Moon, S. (2010). personal communication.

Mutch, J. and Lowe, D. (2006). Multiclass object recognition with sparse, localized features. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 1, pages 11–18. IEEE.

Newton, E. and Phillips, P. (2009). Meta-analysis of third-party evaluations of iris recognition. *IEEE Transactions on Systems, Man and Cybernetics, Part A*, 39(1):4–11.

Nissen, S. (2003). Implementation of a fast artificial neural network library (fann). *Report, Department of Computer Science University of Copenhagen (DIKU)*, 31.

Nordlie, E., Gewaltig, M., and Plesser, H. (2009). Towards reproducible descriptions of neuronal network models. *PLoS Comput Biol*.

Numenta (2011a). Hierarchical temporal memory including htm cortical learning algorithms. `http://www.numenta.com/htmcla.php`.

Numenta (2011b). Numenta htm software. `http://www.numenta.com`.

Olshausen, B. and Field, D. (1997). Sparse coding with an overcomplete basis set: A strategy employed by v1? *Vision research*, 37(23):3311–3325.

Osadchy, M., Le Cun, Y., and Miller, M. (2007). Synergistic face detection and pose estimation with energy-based models. *Toward Category-Level Object Recognition*, pages 196–206.

Panel, A. A. O. R. (2008). Preferred practice pattern: Diabetic retinopathy. `http://www.aao.org/ppp/`.

Partners, M. P. (2008). Messidor retina database. `http://www.mathworks.com/help/toolbox/stats/kmeans`.

Pearl, J. (1988). *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann.

Ranzato, M., Huang, F., Boureau, Y., and LeCun, Y. (2007a). Unsupervised learning of invariant feature hierarchies with applications to object recognition. In *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*, pages 1–8. Ieee.

Ranzato, M., Huang, F., Boureau, Y., and LeCun, Y. (2007b). Unsupervised learning of invariant feature hierarchies with applications to object recognition. In *IEEE Conference on Computer Vision and Pattern Recognition, 2007. CVPR'07*, pages 1–8.

Rice, S., Jenkins, F., and Nartker, T. (1996). The fifth annual test of OCR accuracy. *Information Sciences Research Institute TR-96-01, Las Vegas, NV.*

Riesenhuber, M. and Poggio, T. (1999). Hierarchical models of object recognition in cortex. *nature neuroscience*, 2:1019–1025.

Rojas, R. (1996). *Neural networks: a systematic introduction*. Springer.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386–408.

Rumelhart, D., Hintont, G., and Williams, R. (1986). Learning representations by back-propagating errors. *NATURE*, 323:9.

Scholkopf, B. and Smola, A. (2002). *Learning with kernels*. Citeseer.

Serre, T., Wolf, L., Bileschi, S., Riesenhuber, M., and Poggio, T. (2007). Robust object recognition with cortex-like mechanisms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(3):411–426.

Serre, T., Wolf, L., and Poggio, T. (2005). Object recognition with features inspired by visual cortex. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 2, pages 994–1000. Ieee.

Shi, J. and Tomasi, C. (1994). Good features to track. In *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR'94., 1994 IEEE Computer Society Conference on*, pages 593–600. IEEE.

Simard, P., Steinkraus, D., and Platt, J. (2003). Best practices for convolutional neural networks applied to visual document analysis. In *International Conference on Document Analysis and Recogntion (ICDAR), IEEE Computer Society, Los Alamitos*, pages 958–962. Citeseer.

Strang, G. and Nguyen, T. (1996). *Wavelets and filter banks*. Cambridge University Press.

Sukittanon, S., Surendran, A., Platt, J., and Burges, C. (2004). Convolutional networks for speech detection. In *Eighth International Conference on Spoken Language Processing*. Citeseer.

Sung, K.-K. (1996). *Learning and Example Selection for Object and Pattern Recognition*. PhD thesis, MIT, Artificial Intelligence Laboratory and Center for Biological and Computational Learning, Cambridge, MA.

Sutskever, I. and Hinton, G. (2007). Learning multilevel distributed representations for high-dimensional sequences. In *Proceeding of the Eleventh International Conference on Artificial Intelligence and Statistics*, pages 544–551. Citeseer.

Szarvas, M., Sakai, U., and Ogata, J. (2006). Real-time pedestrian detection using LIDAR and convolutional neural networks. In *2006 IEEE Intelligent Vehicles Symposium*, pages 213–218.

Tenenbaum, J., Silva, V., and Langford, J. (2000). A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319.

Teng, T., Lefley, M., and Claremont, D. (2002). Progress towards automated diabetic ocular screening: a review of image analysis and intelligent systems for diabetic retinopathy. *Medical and Biological Engineering and Computing*, 40(1):2–13.

Theis, L., Gerwinn, S., Sinz, F., and Bethge, M. (2010). In all likelihood, deep belief is not enough. *Arxiv preprint arXiv:1011.6086*.

Tivive, F. and Bouzerdoum, A. (2003). A new class of convolutional neural networks (SICoNNets) and their application of face detection. In *Neural Networks, 2003. Proceedings of the International Joint Conference on*, volume 3.

Tobin, K., Chaum, E., Govindasamy, V., and Karnowski, T. (2007). Detection of anatomic structures in human retinal imagery. *Medical Imaging, IEEE Transactions on*, 26(12):1729–1739.

Trappenberg, T. (2009). *Fundamentals of Computational Neuroscience*. Oxford Press.

Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P. (2008). Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM.

Vision, C. and Center, A. S. (2003). Cmu image data base. `http://vasc.ri.cmu.edu/idb/html/face/frontal_images/`.

Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., and Lang, K. (1989). Phoneme recognition using time-delay neural networks. *Readings in Speech Recognition*, pages 393–404.

Winkler, S. and Mohandas, P. (2008). The evolution of video quality measurement: From psnr to hybrid metrics. *Broadcasting, IEEE Transactions on*, 54(3):660 –668. 173

Wong, R. and Hall, E. (1978). Scene matching with invariant moments*. *Computer Graphics and Image Processing*, 8(1):16–24.

Xu, R. and Wunsch, D. (2005). Survey of clustering algorithms. *IEEE Transactions on neural networks*, 16(3):645–678.

Young, S. (2011). personal communication.

Young, S., Arel, I., Karnowski, T., and Rose, D. (2010). A Fast and Stable Incremental Clustering Algorithm. In *2010 Seventh International Conference on Information Technology*, pages 204–209. IEEE.

Zana, F. and Klein, J. (2001). Segmentation of vessel-like patterns using mathematical morphology and curvature evaluation. *Image Processing, IEEE Transactions on*, 10(7):1010–1019.

Zhong, S. (2005). Efficient online spherical k-means clustering. In *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on*, volume 5, pages 3180–3185. Ieee.

Zhu, X. (2006). Semi-supervised learning literature survey. *Computer Science, University of Wisconsin-Madison*.

Appendix

Appendix A

Appendix: DBN PseudoCode

In this section we show matlab implementations of the pseudocode for DBN greedy layer-by-layer training from Bengio (2009). This is not intended to be a full implementation of a DBN, but rather is a means of clarifying the process. We show functions for the implementation and then a screen print of the code itself.

```
function d = sigm( x )
    d =exp(x)./(1+exp(x));
end
```

```
function h1 = SampleFromRBM( x1, W, b )

    h1 = zeros(size(W,1),1);
    for i=1:size(W,1)
        Sm=0;
        for j=1:size(W,2)
            Sm = Sm+W(i,j)*x1(j);
        end;
        Sm=Sm+b(i);
        Qof1=sigm(Sm);
        %I think Qof1 is a probability of being 1.  So do a rand()
        %and set it...
        r=rand(1);
        if ( r<Qof1 )
            h1(i)=1;
        else
            h1(i)=-1;
        end;
    end;

end
```

```
function [ Wout bout cout ] = RBMUpdate( x1,eps,W,b,c )

%from Bengiio's RBMUpdate pseudo-code
%     W = zeros(3,2);
%     x1=[1 -1]';
%     b=zeros(size(W,1),1);
%     c=zeros(size(W,2),1);
%     eps = 0.01;

    NumberOfHiddenUnits = size(W,1);
    NumberOfInputs = size(W,2);

    h1=SampleFromRBM(x1,W,b);


    %now go through the visible units
    x2=SampleFromRBM(h1,W',c);


    %finally go back through the hidden units...
    % this generates the update
    %rule, and is NOT a sampling!!
    Qofh2Equals1 = zeros(size(W,1),1);
    for i=1:size(W,1)
        Sm=0;
        for j=1:size(W,2)
            Sm = Sm+W(i,j)*x2(j);
        end;
        Sm=Sm+b(i);
        Qofh2Equals1(i)=sigm(Sm);
    end;

    Wout = W+eps*(h1*x1'-Qofh2Equals1*x2');
    bout = b+eps*(h1-Qofh2Equals1);
    cout = c+eps*(x1-x2);

end



clc;
```

```
clear all;

%loop through...
InputSize = 2;
HiddenSize = [3 3 3 3 3];
L=5;
jj=0;
Wcell=cell(L,1);
bcell=cell(L,1);
hcell=cell(L,1);
eps = 0.01;

b0=zeros(InputSize,1); %this is the bias level applied to
% the input, which is h0


for l=1:L
    %initialize the weights and biases
    if ( l>1 )
        W=zeros(HiddenSize(l),HiddenSize(l-1));
        b=zeros(HiddenSize(l),1);
    else
        W=zeros(HiddenSize(l),InputSize);
        b=zeros(HiddenSize(l),1);
    end;

    Wcell{l}=W;
    bcell{l}=b;

end;

for l=1:L

    fprintf(1,'On layer %d\n',l);

    StopCriteria=0;
    SamplesShown=0;
    while ( StopCriteria==0 )

        SamplesShown = SamplesShown+1;
        %Get an input vector and put it for h0
        x=zeros(InputSize,1);
        h0=x;
        fprintf(1,'Got sample %d from the inputs\n',
SamplesShown);
        for k=1:l-1
            if ( k==1 )
                h=h0;
                InputLayerIndex=0;
            else
                h=hcell{k-1};
                InputLayerIndex=k-1;
            end;
            W=Wcell{k};
            b=bcell{k};
            hNext=SampleFromRBM(h,W,b);
            hcell{k}=hNext;
            fprintf(1,'Generated a sample from RBM layer %d using
sample from
%d as input\n',k,InputLayerIndex);
        end;

        if (l==1 )
            bPrev=b0;
        else
            bPrev=bcell{l-1};
        end;

        if ( l==1 )
            hPrev=h0;
        else
            hPrev=hcell{l-1};
        end;

        W=Wcell{l};
        b=bcell{l};

        [ Wout bout cout ] = RBMUpdate( hPrev,eps,W,b,bPrev );

        Wcell{l}=Wout;
        bcell{l}=bout;
        fprintf(1,'Updated weights and bias for layer %d\n',l);

        if ( l==1 )
            b0=cout;
        else
```

```
            bcell{l-1}=cout;
        end;

        if ( SamplesShown == 10 )
            StopCriteria=1;
        end;

    end;  %while (not stop)


end;
```

```
On layer 1
Got sample 1 from the inputs
Updated weights and bias for layer 1
Got sample 2 from the inputs
Updated weights and bias for layer 1
Got sample 3 from the inputs
Updated weights and bias for layer 1
Got sample 4 from the inputs
Updated weights and bias for layer 1
Got sample 5 from the inputs
Updated weights and bias for layer 1
Got sample 6 from the inputs
Updated weights and bias for layer 1
Got sample 7 from the inputs
Updated weights and bias for layer 1
Got sample 8 from the inputs
Updated weights and bias for layer 1
Got sample 9 from the inputs
Updated weights and bias for layer 1
Got sample 10 from the inputs
Updated weights and bias for layer 1
On layer 2
Got sample 1 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Updated weights and bias for layer 2
Got sample 2 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Updated weights and bias for layer 2
Got sample 3 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
```

```
Updated weights and bias for layer 2
Got sample 4 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Updated weights and bias for layer 2
Got sample 5 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Updated weights and bias for layer 2
Got sample 6 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Updated weights and bias for layer 2
Got sample 7 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Updated weights and bias for layer 2
Got sample 8 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Updated weights and bias for layer 2
Got sample 9 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Updated weights and bias for layer 2
Got sample 10 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Updated weights and bias for layer 2
On layer 3
Got sample 1 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Updated weights and bias for layer 3
Got sample 2 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Updated weights and bias for layer 3
Got sample 3 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Updated weights and bias for layer 3
Got sample 4 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Updated weights and bias for layer 3
Got sample 5 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Updated weights and bias for layer 3
```

```
Got sample 6 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Updated weights and bias for layer 3
Got sample 7 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Updated weights and bias for layer 3
Got sample 8 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Updated weights and bias for layer 3
Got sample 9 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Updated weights and bias for layer 3
Got sample 10 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Updated weights and bias for layer 3
On layer 4
Got sample 1 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Generated a sample from RBM layer 3 using sample from 2 as input
Updated weights and bias for layer 4
Got sample 2 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Generated a sample from RBM layer 3 using sample from 2 as input
Updated weights and bias for layer 4
Got sample 3 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Generated a sample from RBM layer 3 using sample from 2 as input
Updated weights and bias for layer 4
Got sample 4 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Generated a sample from RBM layer 3 using sample from 2 as input
Updated weights and bias for layer 4
Got sample 5 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Generated a sample from RBM layer 3 using sample from 2 as input
Updated weights and bias for layer 4
Got sample 6 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Generated a sample from RBM layer 3 using sample from 2 as input
Updated weights and bias for layer 4
Got sample 7 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Generated a sample from RBM layer 3 using sample from 2 as input
Updated weights and bias for layer 4
Got sample 8 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Generated a sample from RBM layer 3 using sample from 2 as input
Updated weights and bias for layer 4
Got sample 9 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Generated a sample from RBM layer 3 using sample from 2 as input
Updated weights and bias for layer 4
Got sample 10 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Generated a sample from RBM layer 3 using sample from 2 as input
Updated weights and bias for layer 4
On layer 5
Got sample 1 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Generated a sample from RBM layer 3 using sample from 2 as input
Generated a sample from RBM layer 4 using sample from 3 as input
Updated weights and bias for layer 5
Got sample 2 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Generated a sample from RBM layer 3 using sample from 2 as input
Generated a sample from RBM layer 4 using sample from 3 as input
Updated weights and bias for layer 5
Got sample 3 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Generated a sample from RBM layer 3 using sample from 2 as input
Generated a sample from RBM layer 4 using sample from 3 as input
Updated weights and bias for layer 5
Got sample 4 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Generated a sample from RBM layer 3 using sample from 2 as input
Generated a sample from RBM layer 4 using sample from 3 as input
Updated weights and bias for layer 5
Got sample 5 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Generated a sample from RBM layer 3 using sample from 2 as input
Generated a sample from RBM layer 4 using sample from 3 as input
Updated weights and bias for layer 5
Got sample 6 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Generated a sample from RBM layer 3 using sample from 2 as input
Generated a sample from RBM layer 4 using sample from 3 as input
Updated weights and bias for layer 5
Got sample 7 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Generated a sample from RBM layer 3 using sample from 2 as input
Generated a sample from RBM layer 4 using sample from 3 as input
Updated weights and bias for layer 5
Got sample 8 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Generated a sample from RBM layer 3 using sample from 2 as input
Generated a sample from RBM layer 4 using sample from 3 as input
Updated weights and bias for layer 5
Got sample 9 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Generated a sample from RBM layer 3 using sample from 2 as input
Generated a sample from RBM layer 4 using sample from 3 as input
Updated weights and bias for layer 5
Got sample 10 from the inputs
Generated a sample from RBM layer 1 using sample from 0 as input
Generated a sample from RBM layer 2 using sample from 1 as input
Generated a sample from RBM layer 3 using sample from 2 as input
Generated a sample from RBM layer 4 using sample from 3 as input
Updated weights and bias for layer 5
```

Vita

Thomas P. Karnowski was born in 1966 and received his Bachelor of Science in Electrical and Computer Engineering from the University of Tennessee-Knoxville in 1988. He went on to graduate school at North Carolina State University, receiving a Master's Degree in Electrical Engineering in 1990 with a Master's Thesis titled "Generalized filtering in acoustoptic systems using area modulation" under the direction of A. Vanderlugt. He then began a long career at Oak Ridge National Laboratory with the Real-Time Systems Group of the Instrumentation and Controls Division, where he worked on a variety of projects relating to signal processing and computer system integration and software development. He became a member of the Image Science and Machine Vision Group in 1995 and continued his work there, focusing on imaging and pattern recognition problems in industrial, medical, and national security applications. He has been on three R&D 100 Award teams from ORNL and has received numerous other awards for technology transfer and engineering development. He restarted his academic pursuits in 2005, taking courses part time and pursuing the PhD degree in Computer Engineering at the University of Tennessee Knoxville. He was very fortunate to receive sabbatical support from ORNL for 2009-2010 to complete coursework and perform dissertation research in earnest, and expects to graduate in 2012. He has been married to Dana Jo Karnowski since 1996 and has three children Eva, Nelson and Noah.