This is to certify that the

thesis entitled

An "Automatic Animal-Like" Face
And Object Recognition System

presented by

Colin Evans

has been accepted towards fulfillment
of the requirements for

Masters degree in Computer Science

John Weng
Major professor

Date 8/27/99

An "Automatic Animal-Like" Face and Object Recognition System

By

Colin Evans

A Thesis

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

MASTER OF SCIENCE

Department of Computer Science and Engineering

1999

Dr. John Weng

Abstract

An "Automatic Animal-Like" Face and Object Recognition System

By

Colin Evans

This thesis introduces the developmental approach to machine intelligence construction and integration, and then describes a system for Automated Animal-Like Learning. At the core of this system is the HSM Tree, an incrementally generated and real-time decision tree for very high dimensional inputs. The HSM Tree is an early developmental framework, capable of learning online and facilitating a real-time cycle of training and testing. The HSM Tree is a decision tree that learns high-dimensional image data in real time with very fast retrieval rates. Several experiments are described that illustrate the operation and capacity of the HSM Tree. Face recognition is demonstrated using recorded video sequences 33,889 rames in length of 143 different subjects with a correct recognition rate of 95.1%. Real-time online learning at a rate of 5-10 video frames per second is also demonstrated.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. John Weng, who provided advice and motivation, and wisdom at opportune times. I also would like to thank my committee members Dr. Esfahanian and Dr. Stockman for their advice and help in completing this thesis.

Special thanks goes to Wey Hwang, Yilu Zhang, and Yong Lee, all of whom provided assistance with programming, data collection, and with the construction of the HSM tree algorithm. Much of this was a collaborative work, and it would not have been possible without their assistance.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The developmental approach to machine intelligence is presented here as an alternative to prior task-specific approaches. The notion of having a robot system that can be trained in the same manner as an animal or young child is very compelling, and this basis in biological models of incremental development offers a window into problems that are very difficult under current methods. We hope for a system that is able to learn continuously over a lifetime, based on real-world input and training, and without task specificity or complex task decomposition. Such systems could transform our views of technology, as electronic pets or companions become available. This framework also makes the domain of intelligent robotics accessible to people who don't have extensive backgrounds in machine learning or artificial intelligence. The notion of real-world real-time feedback and interaction is a model much closer to biological systems, and so it would be possible for a person to transfer their knowledge of working with small children or animals to instructing a developmental robot.

The core of the developmental framework presented here is the HSM Tree, a real-

time incremental decision tree that deals specifically with high dimensional data. The tree learns online in a real-time interactive context, and is used for appearance-based object recognition here. The experiments here include face recognition using recorded video sequences of 143 different people totaling 33,889 frames in length. The system gave a correct recognition rate of 95.1% for face recognition with a retrieval rate of 0.04 seconds per image. The system also was demonstrated in a real-time framework, training at a rate of 5-10 images per second from a video camera while feedback was provided from the keyboard.

The HSM tree was developed by Dr. John Weng, Wey Shiuan Hwang, Yilu Zhang, Yong Beom Lee, and myself. My work on this project has included programming and testing of the whole HSM tree system alongside the other researchers mentioned, and formulating use of the Neural Gas algorithm, the node and tree freezing mechanism, the sample pruning and retraining mechanisms, and the spawning criteria for the version of the tree described here. All data collection and all experimental results were done by myself as well.

1.1 Task-Specific Paradigm

Current work in robotics and machine intelligence is based on the notion of completing a task. The paradigm for approaching research in these areas is to identify a task or a problem (such as autonomous navigation, face recognition, or Robo-Soccer) and then create a system or agent that "solves" the problem or performs the task. The success of the system largely depends on the decomposition of the problem into sub-problems

and the quality of the knowledge that the scientist is able to impart to the system, either in the form of training samples or in explicit design. This paradigm produces robust and reliable systems that are very good at completing tasks with small scopes that can be formalized and decomposed cleanly and efficiently.

Approach	Species architecture	World knowledge	System behavior	Task specific
Knowledge-based	programming	manual modeling	manual modeling	yes
Behavior-based	programming	avoid modeling	manual modeling	yes
Learning-based	programming	treatment varies	special-purpose learning	yes
Evolutionary	genetic search	treatment varies	genetic search	yes
Developmental	programming	avoid modeling	general-purpose learning	no

Table 1.1: A comparison of several task-specific approaches to machine intelligence alongside the developmental approach.

The task-oriented paradigm works well for producing systems that solve individual problems such as ALVINN [11], a neural-network based autonomous navigation system that is able to drive on roads, AutonoMouse [4], a robot that learns to follow light given reinforcement and using genetic algorithms for learning, or Xavier [14], an autonomous robot navigation system that uses partially observable Markov models, or even broad classes of problems such as SOAR [6], a "general problem solver" that uses axiomatic definitions to define a world, and then performs problem searches through that axiomatic space. Other examples of machine intelligence systems that have been successful include the behavior-based subsumption architecture [3][2], reinforcement methods such as Q-learning [18], [5], and even knowledge-based methods such as CYC [7] and Wordnet [9].

However, all of these systems and methodologies stand firmly within a task-

oriented design paradigm. For any of the above systems to be successful, the designer must know and understand the problem well and be able to give a good decomposition of the problem in order to expose the important points and relevant sub-tasks vital for success. In this kind of a design process, there is a real emphasis on the task space. A good system design recognizes what constraints are present in the environment and how they can be leveraged to provide a compact and coherent representation of the problem that can then be mapped to specific tool or algorithm.

The steps involved in a task-specific approach to machine intelligence are as follows:

1. A scientist identifies a task or problem for study (such as autonomous robot navigation in building hallways, or categorizing web pages).

2. The scientist analyzes the task as best she can, locating key points of the task that are important for success and discarding parts of the task that are irrelevant for good performance of the system. This can involve decomposing the task into tractable sub-problems, identifying what kinds of information needs to be present and processed for a solution to be available, and even deciding what constraints on the problem must be present to allow a good decomposition.

3. The scientist selects a tool (such as POMDP models of the building for navigation, or a "bucket of words" representation for web pages) and encodes the important components of the problem as a task space representation that will allow the tool to operate efficiently and effectively on the problem.

4. Finally, the scientist determines and encodes the parameters of the tool. This can mean incorporating hand-crafted knowledge or manually decomposed behaviors, estimating parameters from training samples using a statistical or computational learning method, or searching for parameters using a manually designed and task-specific objective function.

This paradigm is very good at providing solutions to problems that can be understood well enough to be decomposed by hand. However, it doesn't do well with vague and poorly specified problems, such as general vision-based object recognition, navigation in a variety of unconstrained environments (such as outside environments), and general human-computer centered interaction. Such very general problems have very few reliable constraints, and because of this it is very difficult to construct compact and consistent representations. Without a compact and reliable representation or good task decomposition, the manual labor and domain knowledge required for the task-specific paradigm are very large.

1.2 Developmental Paradigm

We offer an alternative paradigm for approaching robotics and machine intelligence. Instead of trying constructing a complex representation for a general problem, complexity can be placed in the algorithm. Automation and adaptivity can be substituted for domain knowledge and manual decompositions. For many types of machine intelligence problems (such as theorem provers and natural language parsers) a hand-crafted problem decomposition can be essential to the success of the system. However, there

are classes of problems that don't lend themselves well to handmade decompositions (such as some vision problems, and problems of robot control). In these types of problems, the information needed for correct operation is not centralized and easily represented. For example, appearance-based vision methods depend on the combined input of the thousands of pixels in an image, and any small subset of pixels contains much less information for classification than the whole image. The information in a pixel image is not easily decomposed into pieces and semantically labeled.

Similar issues are present in robot control problems (mapping a set of sensor inputs to motor positions with real-time operation). The classical method for performing robot control is to build an analytical model of the inverse kinematics of the robot. The model is usually based on geometric and physical properties of the robot, and maps the final position of, say, a robot arm to the joint angles needed to move the arm to that position. This approach to robot control decomposes each part of the control problem (motors, joints, angles) into a formal model, but such an approach must have extraordinary complexity in order to model things like friction, elasticity of joints, and motor errors. All of these things are gremlins that chip away at the accuracy of an analytical model, and this is because the problem is not totally decomposable into semantic pieces that can be easily recognized and understood.

We believe that problems without clean and easy semantic decompositions can be better decomposed automatically. We take the biological and psychological processes of human and animal development as an inspiration, where over the course of a life-time, from birth to old age a human acquires knowledge and skills in an incremental and autonomous fashion. The learning mechanisms present in humans are very much

automated – young children acquire motor skills and language at a surprising rate. New tasks are learned without the need for reprogramming or a new task decomposition. Representations of knowledge are implicit and distributed in the system, as opposed to being explicitly represented in states, symbols, or data structures. We believe that methods that are developmental in nature offer avenues for exploration into very general and vaguely defined tasks and problems.

The goal of a developmental approach is to study the mechanism, present at the "birth" of a system, that is able to incrementally learn and acquire abilities and behaviors through real-time interaction with the environment. Some important components of such a system are outlined below:

1. Domain-extensibility: The system is able to learn new tasks without the need for reprogramming.

2. Relief from manual task decomposition: The system does not require a programmer to analyze and decompose every task that the system must learn, but instead automatically generates a task space.

3. Developmental mechanism: The system learns incrementally over time in a real-time environment. This means that the system can act on its environment in an unconstrained manner and learns from interaction and real-time feedback from the environment.

In such an arrangement, the system would learn through active training in the world. For instance, for a robot to learn a new task such as navigation in building

hallways, the robot could be initially led through the building several times, and then allowed to roam autonomously. If the robot appeared to be lost or started to run into a wall, it would be corrected at that point where the error occurred, introducing new training data, and then sent on its way. In this way, a real-time interleaved cycle of training and testing is established, where the system's performance is constantly critiqued and corrected by a teacher. This interleaving of training and testing is markedly different from classical machine learning methods. Because training and testing is done in real time in the environment, errors can be corrected quickly and immediately, as soon as they are identified. Scarcity of training samples is less of a problem, as input to the system is in real-time and so the system is able to acquire data as needed. The training process becomes an active learning process, where the trainer can seek out parts of the task that the system has not learned well and concentrate those weaknesses.

This notion of "interactive learning" allows totally new methods of robot training which are much closer to biological models of learning from instruction and reinforcement. Because of the "hands-on" training methodology, it is possible for people who have no formal background in robotics to train and teach a developmental robot system, simply using their experiences of instructing or training animals and small children.

Additionally, the methodology of a system developing over time similiar to how a biological system would mature allows for the teaching of a wide range of tasks, with prior skills and abilities acquired by the system being used in future tasks. Because the frame work is totally integrated – all of the learning takes place in the same

mechanism – it is simple for skills from one task to cross over to another task. These are things that are difficult to do utilizing the traditional task-specific paradigm of design.

1.3 Learning Mechanism

The mechanism that we have adopted for developmental learning is called "automated animal-like learning". We use the term "animal-like" because the learning process is similar to stimulus-response learning that can be done by an animal in real time. We base our mechanism around a machine agent M – a robot with various sensors (such as vision or sonar) and effectors (such as wheels or an arm). The machine agent views sensory input in discrete time steps starting from its "birth" and is capable of activating effectors at each time step as well. The training phase consists of actions being imposed on M, guiding it through its environment and presenting it with the correct effector output given the sensory input encountered at each time step. Testing of the system involves allowing the system to map sensory input to effector outputs and act in real time. The system updates its internal models immediately, meaning that the gap between training and testing is seamless. At the core of our developmental algorithm is the HSM (Hierarchical Statistical Modeling) tree. This is a high-dimensional incrementally built decision tree that maps sensor inputs to effector output.

The purpose of this work is to describe the HSM tree as a core element to a developmental framework, and to offer several experiments that both show the operation

and capacity of the HSM tree. This is an early work, as the developmental approach is at a fledgling stage.

1.4 Prior Work

No prior work existed about an overall developmental algorithm for an artificial agent until our work here. The HSM tree is based on the SHOSLIF [19] [15] [16] approach to appearance-based vision and robot control. This approach involves constructing a high-dimensional decision tree for mapping image vectors to robot control signals, and was originally inspired by Turk and Pentland's Eigenface [17] approach to face recognition, which involved taking principal component projections of data sets in order to reduce dimensionality and computation while maintaining robust performance. Each node of the SHOSLIF tree finds a subspace projection (such a PCA or LDA projection) of the total image space, and decision splits at each node are found by clustering together classes or by finding splits that reduce variance. However, the SHOSLIF framework is an offline batch generated framework, while the HSM tree is generated online and incrementally.

There are large bodies of work surrounding decision tree methods of classification, with the most successful and well known being the C4.5 [12] [13] and CART [1] methods. Both of these methods use "purity metrics" (such as negative entropy) to find single-axis partitions at each node of the tree. These methods are fast and very successful for low-dimensional problems, but their reliance on single-axis decision boundaries and need for offline batch processing makes them totally inappropriate for

handling high dimensional image data in a real-time training framework. One other system, the OC1 [10] decision tree, offers decision boundaries that are "oblique" and not axis-parallel. However, the OC1 algorithm searches for decision boundaries using monte carlo methods and gradient ascent searches, both of which are very slow and work very poorly on very high dimensional image vectors.

There was no work found on very high dimensional incrementally generated real-time classifiers, which is the job that the HSM tree performs. Most classification methods that are able to operate in real-time (such as decision trees and neural networks) fall prey to the curse of dimensionality in training over 5,000-10,000 dimensional image data in training. These methods are able to produce correct classifications for the training examples, but do not generalize well at all to test cases. Methods that explicitly deal with this kind of high dimensional data (such as PCA and LDA projection methods) require copious amounts of offline processing and are often slow for classification as well.

Chapter 2

The HSM Tree

2.1 Overview of the HSM Tree

The HSM tree allows real-time operation of a nearest neighbor based regression tree by decomposing the input space recursively into a tree structure. The purpose of the algorithm is to give a good decomposition of the input space (and thus good classification performance) while minimizing the number of comparisons and amount of computation that must be done and thus making a nearest neighbor classifier computationally feasible for a very large number of samples. The tree is a regression tree, which mainly means that similarity in the output space is assumed to imply similarity in the input space. A major requirement of the tree is online construction in an incremental fashion that allows training and testing to be interleaved. This means that there are no opportunities for offline batch processing of samples, and the tree must be able to learn and operate in real time. Also, because we are interested in working with high-dimensional images, the tree employs methods to reduce

computational costs, such as using dimensionality-reducing transforms.

The HSM algorithm models the output space of a function in a coarse-to-fine fashion, using some simple incremental data clustering techniques to first break the output space into coarse low-resolution blocks, and then breaking these coarse groups into finer and finer groups, increasing the resolution of the output space at each step. This method of breaking up the output space into nested clusters of similar outputs gives a tree-like formulation of the data. This formulation of the data gives a corresponding coarse-to-fine decomposition of the input space. Because the decomposition of the input space is induced by the decomposition of the output space, the partitionings of the input space are useful for classification purposes. Each of the output space clusters acts as virtual class label, aggregating together similar outputs and thus creating clusters out of the corresponding inputs. The input clusters are then used to derive a discriminant subspace projection of the data, and are modeled by Gaussian curves. At the leaves of this tree, individual samples are stored for actual classification. Classification is done by traversing a path down the tree and then performing a linear nearest-neighbor search on the samples at the leaf.

Given a high dimensional input space $\mathbf{X}$ and a lower-dimensional output space $\mathbf{Y}$, we see a tree-like formulation of data as the basic structure of the HSM algorithm, where each node N of the tree contains at most $\mathbf{q}$ cluster mappings – function mappings from a cluster in the X space to a cluster in the Y-space. These cluster mappings are **virtual classes** that aggregate Y-space classes together and then model the corresponding X-space samples as clusters. Each of these virtual classes is specified by a data cluster $x_i \in X$ and a data cluster $y_i \in Y$. Given that there are $\mathbf{m}$ cluster map-

pings ($1 \leq m \leq q$) in node N, a discriminant subspace $\mathbf{D}$ is generated in N, where D is the $m-1$ dimensional subspace of X that contains the m cluster centroids (cluster means) of the clusters. This means that D is the $m-1$ dimensional hyperspace that is defined by m X-space cluster centers. Each cluster x_i is represented by a Gaussian distribution with a mean in the X-space and a $(m-1) \times (m-1)$ covariance matrix Γ_i in the discriminant space D. Membership of a given pattern vector x_{new} in each of the m virtual classes is determined by projecting x_{new} into D, projecting each of the cluster means x_i into D, and then finding the negative-log-likelihood that x_{new} is a member of each cluster using the covariance matrix Γ_i and the projected mean x_i as parameters to a Gaussian distribution.

Every internal node of the tree can have up to q nodes as children, with each child node corresponding to a virtual class in the parent node. Thus, the X-space is hierarchically decomposed as a nested mixture of Gaussians. The root node breaks the X-space up into m Gaussians, with every Gaussian distribution roughly representing a class corresponding to a cluster in the Y-space. The child node N_i of each virtual class at the the root takes as input every pattern vector that falls into the Gaussian distribution specified by the parent of N_i, and N_i again breaks up its input set into a mixture of Gaussians, with each Gaussian roughly representing a class (again, a cluster in the Y-space.) Assuming that the underlying data is not totally random, such recursive decomposition will eventually lead to a very low level of variation and a high resolution in the Y-space in the leaves of the tree.

An $x_{new} \mapsto y_{new}$ pair that represents a mapping from the X space to the Y space is added to the tree by starting at the root node and finding the Y-space cluster y_i

that is closest to y_{new} and updating y_i, x_i, and Γ_i incrementally. The closest X space cluster x_j to x_{new} is then found using negative-log-likelihood, and the algorithm is repeated for the child node beneath x_j. When the search arrives at a leaf cluster c_{leaf}, a nearest neighbor search is done of the samples at c, and if x_{new} is not contained there, it is added to c_{leaf}. If there are too many samples in c_{leaf}, c_{leaf} spawns a child node that inherits all of the samples from c_{leaf}.

A predicted mapping from a sample input x_{new} can be determined by finding the path from the root to a leaf note by finding the best membership of x_{new} using negative-log-likelihood in the X space at each level of the tree. When the algorithm arrives at a leaf cluster c_{leaf}, a sequential nearest-neighbor search is done over the samples present at the leaf. If c_{leaf} has no samples, then the Y-space cluster centroid of c_{leaf} is used as an output. For greater accuracy in searching the tree, multiple paths can be taken down the tree to find the top k matches to x_{new}, and then the closest leaf sample is taken as an output.

2.2 Nodes

Each node attempts to incrementally build a model that separates classes into different clusters. This is done by first breaking up the Y-space (the output space, or class labels) into at most q separate clusters, and then attempting to model the X-space clusters that result. In each node, the Y-space clusters act as virtual class labels, separating the X-space samples into different virtual classes. The node attempts to model each of these X-space virtual classes for classification.

Each node breaks up the Y-space using unlabeled clustering with most q Y-space clusters. If there are fewer than q classes present in a node, the node will have fewer than q clusters, with one cluster for each class, and each virtual class will correspond to a real class. However, if there are more than q classes present in a node, then the node must aggregate similar classes into the same cluster, forming a virtual class that corresponds to several real classes. The neural gas algorithm [8] – an incremental variant of the k means clustering algorithm – is run over the Y-space vectors in order to find a good way to group classes. Neural gas is less sensitive to local minima than k-means, and that is why it is used in this context.

Because the virtual class labels are decided by similarity of the Y-space outputs, the labeling cannot by random or arbitrary. In regression-type problems, similarity in the output space often implies similarity in the input space, and so this method for aggregating classes works well with the raw output space of regression problems. However, in discrete classification problems where the output corresponds to a name or a symbol, arbitrary or random Y-space vector labels can give very poor partitionings of the X-space. In this case, it is useful to base the Y-space vectors on the X-space data. A method that we used was to set the Y-space label for each class equal to the X-space mean of the class across all of the samples

In the X-space, each virtual class is modeled by a mean and covariance matrix, which are estimated incrementally over the samples using amnesic averaging. Because we wish to take second-order statistics (covariance matrices) for each cluster, a dimensionality-reducing transformation is necessary to make maintaining covariance matrices computationally feasible. If such a transformation was not performed, then

given a $2,000$ dimensional input space (not uncommon for image data), the covariance matrix would need to have $2,000 \times 2,000 = 4,000,000$ elements, which is too large to store or perform any reasonable computations. As a dimensionality-reducing transformation, we take the $m - 1$ dimensional subspace that covers the m cluster centers in the node N. This subspace captures some discriminant information, and the usefulness of each basis vector is captured in the covariance matrix of each virtual class.

Classification at each node is done using negative-log-likelihood for each X-space virtual class. Because we are incrementally approximating the covariance matrix for each class, it is likely that if very few samples have been viewed, the covariance matrix has not yet been reliably estimated, and therefore will give poor classification performance. To solve this problem, we use three different distance metrics in order to give reliable distance measures – Gaussian likelihood, Mahalanobis distance, and Euclidean distance. Gaussian likelihood is based on the covariance matrix for each virtual class and needs a large number of samples for reliable parameter estimation. Mahalanobis distance uses the average of covariance matrices across all of the virtual classes in the node, and so needs fewer samples for good reliability. However, Mahalanobis distance is a less accurate metric than Gaussian likelihood for classification because it is not class dependent and thus carries no discriminant information. Euclidean distance requires no samples and is the naive metric used when few samples have been seen. From these three distance metrics, a weighted average is taken that is based on the number of training samples that have been seen in the node and by each individual cluster.

Updates to the node are done as follows:

Procedure: Update-Node: *Given a node N with m clusters ($1 \leq m \leq q$) and a mapping $x_{new} \mapsto y_{new}$, N is updated with the mapping $x \mapsto y$ as follows:*

1. *If $m < q$ and for each Y-space cluster y_i, $|y_i - y_{new}| > \gamma_y$, add a new virtual class to N with X-space mean x_{new}, Y-space mean y_{new}, a zero covariance matrix, and jump to step 5.*

2. *Find the virtual class c with the closest Y-space cluster centroid y_c to y_{new} using Euclidean distance.*

3. *Update the X-space and Y-space cluster centroids x_c and y_c using amnesic averaging.*

4. *Update the X-space covariance Γ_c matrix of c using amnesic averaging.*

5. *Re-compute the likelihood matrices for N.*

6. *Update the remaining $m - 1$ Y-space clusters with y_{new} using the neural gas algorithm.*

7. *Re-compute the discriminant subspace D_N of the node using Grahm-Schmidt orthogonalization.*

2.2.1 Discriminant Subspace

In order to reduce the dimensionality of the input space, an orthonormal subspace projection D is found. The subspace D is the $m - 1$ dimensional space that contains

the m X-space cluster centers x_1 through x_m. This means that D is the $m-1$ dimensional hyperspace defined by the m X-space cluster means $x_1, ..., x_m$. Distance relative to this set of m points is captured by this projection, and cluster information that is lost in the projection can be recovered in the second order statistics gathered. Because the X-space cluster centers are formed by virtual class groupings in the Y-space, the projection is a discriminant projection that captures information useful for classification, as opposed to a component projection that captures variance across all of the data (such as the well-known principal component projection).

The discriminant subspace D is represented by an $m-1$ dimensional orthonormal vector basis $b_1, ..., b_{m-1}$ with projection matrix $B_D = (b_1/.../b_{m-1})$. The basis is found by finding the mean $x_{mean} = \frac{1}{m}\Sigma_{i=1}^{m} x_i$ of the cluster centers and then taking the m difference vectors $d_i = x_{mean} - x_i$ for $1 \leq i \leq m$. Running the well-known Grahm-Schmidt Orthogonalization procedure on the vectors $d_1, ..., d_m$ will produce at most $m-1$ orthonormal vectors $b_1, ..., b_{m-1}$ that are the basis of the discriminant subspace D. If there are linear dependencies between the set of vectors $d_1, ..., d_m$, then it is possible that the dimensionality of the subspace will be less than $m-1$, but this is handled by padding zero vectors in the basis to get $m-1$ dimensions.

2.2.2 Amnesic Averaging in the X-space and Y-space

Cluster centroids x_c and y_c and the covariance matrix Γ_c are update incrementally using amnesic averaging. Given a set of small update values α_x, α_y, and α_Γ, at time n with inputs x_{new} and y_{new} the updates are done as follows:

$$x_c^{(n)} = (1 - \alpha_x)x_c^{(n-1)} + \alpha_x x_{new}$$

$$y_c^{(n)} = (1 - \alpha_y)y_c^{(n-1)} + \alpha_y y_{new}$$

To compute the covariance matrix, the X-space vectors are projected into the discriminant subspace D using the transformation matrix B_D. This gives the following:

$$\Gamma_c^{(n)} = (1 - \alpha_\Gamma)\Gamma_c^{(n-1)} + \alpha_\Gamma(B_D x_{new} - B_D x_c^{(n)})(B_D x_{new} - B_D x_c^{(n)})^T$$

The update values are decayed over time, allowing for a "freezing" process to occur. This is discussed later.

2.2.3 Neural Gas

For grouping together the Y-space clusters, we use a modified version of the neural gas algorithm [8]. When a node N is updated with a new mapping $x_{new} \mapsto y_{new}$ at time n, the virtual class with a Y-space centroid closest to y_{new} is updated incrementally using amnesic averaging. Then the remaining Y-space centroids for the other virtual classes are incrementally updated with y_{new} according to an update variable α_{ng}. α_{ng} is very small and quickly time-decaying. This means that over time, the update pattern becomes similar to an incremental version of k-means, where the closest cluster is updated and all of the others are ignored.

Assume a new mapping $x_{new} \mapsto y_{new}$ and a node N with m virtual classes $c_1 ..., c_m$ such that $i > j \rightarrow |y_i - y_{new}| \geq |y_j - y_{new}|$, i.e. the clusters are ordered by distance to the vector y_{new} with y_1 being the closest. Vector y_1 is updated using amnesic averaging described above, and the vectors $y_2 ..., y_m$ are updated at time n as follows:

$$y_i^{(n)} = \frac{\alpha_{ng}}{i} y_{new} + (1 - \frac{\alpha_{ng}}{i}) y_i^{(n-1)}$$

This method scales the update α_{ng} by the ranking of the cluster centroid in closeness to y_{new}. Over time, α_{ng} is decayed at a rate faster than α_x or α_y used in amnesic averaging.

2.2.4 Update Values and Freezing

In order for accurate estimation, many of the statistics above depend on other statistics to already be estimated. The X-space means depend on stationary Y-space virtual class vectors, and the X space covariance matrices depend on stationary X-space means. One method for doing this would be to not estimate a variable until its dependencies are accurately estimated, but this approach is not incremental in nature. Instead, we keep the update values α_{ng}, α_y, α_x, and α_Γ constant for different periods of time, and at different schedules we begin to decay the values. All of the update values start off constant, and then at some point the update values for variables with no dependencies begin to decay. After an α decays a significant amount and the update amount gets close to zero, its dependents begin to decay as well. As the

dependencies of a variable become stationary through update value decay, it becomes possible to estimate the variable with accuracy.

Each node keeps a local count T_N of the number of global times steps since the node was created. This is the age of the node. The decay is parameterized by a value η which is the number of local time steps that will pass before an update variable will start to decay. The decay pattern is set up so that as the node passes each specified age, a different update value α begins to decay. After all of the update variables have decayed for a certain time, the node ceases to update at all, and "freezes". This freezing allows for reliable parameter estimation by child nodes which depend on consistent decision boundaries by the parent nodes. The decay pattern used in our experiments goes as follows:

1. The neural gas update value α_{ng} decays first as a part of the neural gas algorithm. It starts to decay at age $T_N = 0.7\eta$.

2. The Y-space update value α_y decays next, fixing the Y-space clusters and thus fixing the virtual classes of the node. It starts to decay when $T_N = 2.0\eta$.

3. The X-space mean update value α_x begins to decay once the virtual classes are fixed. It starts to decay at age $T_N = 2.5\eta$.

4. When the X-space means are fixed, the X-space covariance matrices can be estimated, so α_Γ begins to decay. It starts to decay when $T_N = 3.0\eta$.

5. At age $T_N = 4.0\eta$, the node ceases to update any variables and is frozen.

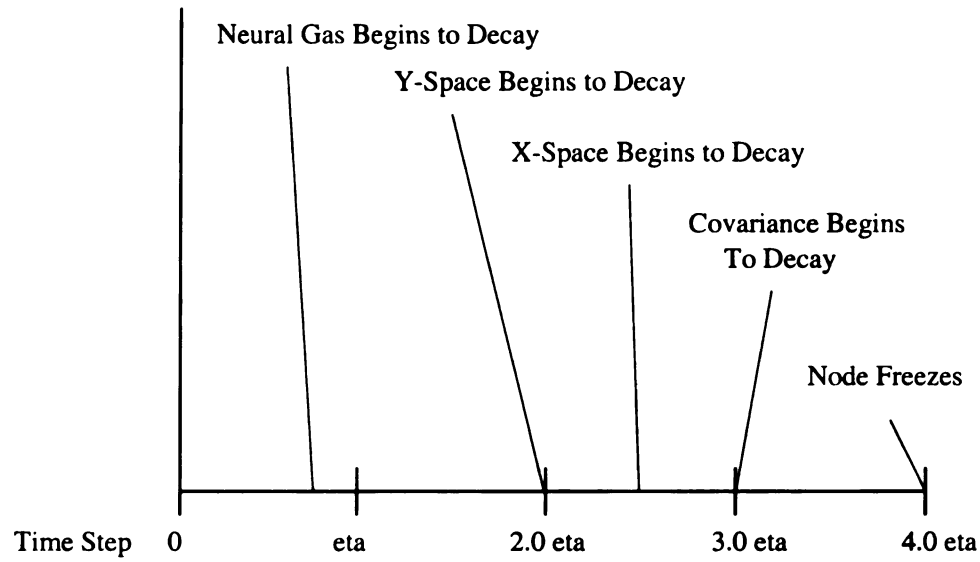

Figure 2.1: This is a timeline of the freezing schedule of the tree across 4η time steps.

Choosing a good value of η depends on knowing something about the types of tasks that must be learned. Generally, the tree should be able to be presented with instances of each task to be learned before η time steps have passed. If η is too small, nodes will freeze too early, damaging classification performance, as variables freeze before seeing samples from all of the classes. If η is too large, then the system will take a very long time to reach reasonable performance.

Assuming that a variable begins to decay at age T_v, the decaying update value α_{decay} is computer as a function of the original α update value as follows:

$$\alpha_{decay} = \alpha e^{T_N - T_v}$$

This means that as the age of the node progresses past age T_v, the update value decays at an exponential rate.

2.2.5 Likelihood Metrics

Negative-log-likelihood given an X-space pattern x_{new}, a cluster mean x_i, an $m-1$ dimensional space, a transformation matrix B_D that projects vectors into the discriminant space, and a scatter matrix S_i is defined as follows:

$$L(x_{new}, x_i) = \tfrac{1}{2} d(x_{new}, x_i) + \tfrac{(m-1)}{2} ln(2\pi) + \tfrac{1}{2} ln(|S_i|)$$

$$d(x_{new}, x_i) = |B_D\, x_{new} - B_D\, x_i|\, S_i^{-1}\, |B_D\, x_{new} - B_D\, x_i|^T$$

If $S_i = \Gamma_i$, using the covariance matrix for cluster i, then the equation above is for Gaussian negative-log-likelihood. If we take the within-class scatter matrix S_w - the weighted average of the covariance matrices in the node - then $L(x_{new}, x_i)$ gives Mahalanobis negative-log-likelihood. If $S_i = I$, the identity matrix, then the equation gives Euclidean distance. Each of these distances has decreasing usefulness for classification, and a decreasing demand for samples for correct estimation as well.

Γ_i is estimated only from samples in virtual class i, and thus gives information useful for discriminating between classes. S_w is estimated from all of the samples that have come through the node, and thus is usually estimated with good accuracy sooner than Γ_i. However, S_w contains no class-dependent discriminant information, and is thus less helpful for classification. Given v_i sample visits at each cluster i in the node, with $v = \Sigma_{i=1}^{m} v_i$, S_w is derived as $S_w = \Sigma_{i=1}^{m} \frac{v_i}{v} \Gamma_i$. Euclidean distance is naive, requiring no samples making no distance-based weightings for class discrimination.

It is the simplest distance metric.

We want to transition between these three distance metrics automatically based on the number of samples that have been seen. If no samples have been seen, we would like to use Euclidean distance as a metric. If there are very few samples in the virtual class i, but the node has seen many samples, then we would like to use Mahalanobis distance with S_w. If the virtual class i has seen many samples, then we would like to use Gaussian likelihood with Γ_i. To facilitate this automatic transition, we set S_i to a weighted average of I, S_w, and Γ_i. The weights are proportional to the number and distribution of samples across the node, and as samples are added, the weighting slips from Euclidean distance to Mahalanobis distance to Gaussian likelihood.

We describe the weighted average using three weights w_e, w_m, and w_g, with $w = w_e + w_m + w_g$. The covariance matrix used for negative-log-likelihood is derived as $S_i = \frac{w_e}{w} I + \frac{w_m}{w} S_w + \frac{w_g}{w} \Gamma_i$. We would like a scheme for assigning w_e, w_m, and w_g so that Euclidean distance is weighted the most when there are very few sample visits to the node, Mahalanobis distance is weighted highly when the total number of sample visits v to the node is high, and Gaussian likelihood is weighted highly when the average number of visits to a single cluster in the node is high. To do this, Euclidean distance is given a static weight v_e, and w_m and w_g are considered to be functions of the number of samples that have been seen coupled with the number of parameters that must be estimated in the covariance matrices.

After v visits to the node, and given that each visit produces an $m-1$ dimensional vector used to estimate one covariance matrices, this means that there have been $v(m-1)$ scalar values so far for estimating the parameters of the covariance matrices.

S_m, the Mahalanobis distance matrix, has $\frac{m(m-1)}{2}$ parameters to be estimated, which means that there has been $\frac{2v(m-1)}{m(m-1)}$ scalars for each parameter of S_m. There are m Gaussian likelihood matrices, which means that there are on average $m(\frac{2v(m-1)}{m(m-1)})$ scalars for each parameter in the m Gaussian likelihood matrices. This system of computing weights based on the number of parameters that need to be estimated gives a basis for assigning w_m and w_g. Given enough samples, we would like to see Gaussian likelihood become the dominant weight, so we set an upper bound v_e for w_m. Given the above formulations, the weightings are as follows:

$$w_e = v_e$$

$$w_m = min\{\frac{2v}{m}, v_e\}$$

$$w_g = \frac{2v}{m^2}$$

The bound v_e is set low – usually to a value of 10 or 20; the tree is not very sensitive to this setting in the long run, and this method makes it possible for a smooth transition to be made into Gaussian likelihood, which is the decision metric that we are really interested in using for the best classification performance.

2.3 Tree

The two major operations on the tree are searching and updating. A search on the tree starting with an X-space vector x_{new} involves starting at the root node and recursively finding the virtual class of x_{new} at each node and then going on to the child node of the class. Individual samples are stored in the leaves of the tree, and when the search arrives at a leaf, x_{new} is compared to the samples there and the

output value of the closest sample is returned.

Because the tree partitions are not perfect, we use a version of the search algorithm that follows down multiple paths of the tree and performs a "redundant search". The algorithm has two parameters - k, the maximum number of paths that can be searched, and ϵ, the threshold ratio of the negative-log-likelihood of the closest virtual class in a node to the other likelihoods. If the likelihood ratio is smaller than ϵ and there are fewer than k paths, the virtual class is added as another path to search.

Procedure Search-Tree: *Let P be the set of virtual classes that designate current paths being followed in the search, with $|P| \leq k$. Assume an X-space vector x_{new} to be classified by tree T with root node r.*

1. *Take the negative-log-likelihood of x_{new} for all virtual classes in r, and add the virtual class with lowest likelihood d_1 to P.*

2. *For all of the other virtual classes with likelihoods d_i, if $\frac{d_i}{d_1} \leq \epsilon$, add the virtual class to P as long as $|P| \leq k$.*

3. *For each member $m \in P$, if m is not a leaf, take the child node N of m and take the likelihood of x_{new} for all virtual classes in N. Add the virtual class with lowest likelihood d_1 to P and remove m.*

4. *For all of the other virtual classes in N with likelihoods d_i, if $\frac{d_i}{d_1} \leq \epsilon$, add the virtual class to P as long as $|P| \leq k$.*

5. *Repeat steps 3 and 4 until P contains only virtual classes that are leaves.*

6. *Perform a linear search through all of the samples contained in each of the leaves in P, finding the sample s with lowest Euclidean distance in the X-space. If a leaf has no samples in it, take the Euclidean distance to the X-space mean of the leaf and treat it as a sample.*

7. *Return the Y-space mapping of the closest sample and the leaf node of the sample.*

Given a new mapping $x_{new} \mapsto y_{new}$, the tree is updated using Add-Pattern, which calls procedure Update-Tree. Update-Tree works by starting at the root node, running Update-Node, and then finding the virtual class of x_{new} and proceeding to the child node of that class. When the algorithm arrives at a leaf cluster, the x_{new} vector is compared to each of the X-space samples present at the leaf, and if the difference between X-space vectors is larger than a small threshold γ_x, $x_{new} \mapsto y_{new}$ sample is added to the leaf cluster.

Procedure Update-Tree: *Given a tree T with root r and a novel mapping $x_{new} \mapsto y_{new}$, the update goes as follows:*

1. *Let the current node N be r.*

2. *Call Update-Node on N with the mapping $x_{new} \mapsto y_{new}$ if N is not frozen.*

3. *Find the virtual class c with lowest negative-log-likelihood for x_{new}.*

4. *If c has a child node, let N be the child node of c and go to step 2.*

5. *Perform a linear search through all of the samples contained in c. If for each sample x_i in c, $|x_i - x_{new}| > \gamma_x$, add x to the leaf c. Return c.*

The Update-Tree procedure is not called directly at each time step, although this is the fundamental procedure for updating the tree. Sample pruning, sample retraining, and spawning are handled within the the procedure Add-Mapping, which is called with a new mapping $x_{new} \mapsto y_{new}$ at each time step and which in turn calls Update-Tree if it is needed.

Procedure Add-Mapping: *Given a tree T and a novel mapping $x_{new} \mapsto y_{new}$ at time step n, the mapping is processed as follows:*

1. *Test the mapping for pruning. Call Search-Tree with $k = 1$, $\epsilon = 0$, and input x_{new}. If Search-Tree returns a vector y_i such that $|y_i - y_{new}| > \gamma_y$, call Update-Tree with the mapping $x_{new} \mapsto y_{new}$ and go to step 2. Otherwise, discard $x_{new} \mapsto y_{new}$ and jump to step 3.*

2. *Check spawning conditions. If the leaf c returned by Update-Tree is frozen and has more than σ samples, spawn a new child node beneath c and call Update-Node on the new node with $x_{new} \mapsto y_{new}$.*

3. *Retrain one sample. Select an existing sample $x_e \mapsto y_e$ from a node in the tree, delete the sample from the tree, and call Update-Tree with $x_e \mapsto y_e$.*

2.3.1 Sample Pruning

Because the tree is expected to operate in a real-time environment with a continuous real-time input of samples, an incremental pruning algorithm is employed on every new mapping $x_{new} \mapsto y_{new}$ before the tree is updated to keep the tree from being overwhelmed with needless and redundant data. An update is only done if the tree incorrectly classifies the vector x_{new} by a threshold γ_y. If the tree correctly classifies x_{new}, then the new sample is discarded.

This simple incremental pruning algorithm tends to vastly reduce the number of samples needed to perform a classification problem. In some cases, it also will work to normalize prior probabilities of different classes by pruning away samples that provide redundant and inessential information. This is useful because of the unlabeled clustering methods used for creating the Y-space virtual classes. The unlabeled clustering methods used work to incrementally find cluster centers that reduce total squared error. When prior probabilities are dramatically skewed across a set of classes, the classes with highest prior probability will usually be responsible for the most squared error, and so the clusters that result will carry very little information about classes with small prior probability. These sorts of clusters tend to be non-optimal for classification, especially if many of the samples in more frequently viewed classes do not add to classification accuracy. Pruning tackles this problem quickly and effectively.

2.3.2 Retraining Samples

Each X-space mean and Y-space virtual class is incrementally estimated as new samples arrive, and because of this, the decision boundaries induced by the means change over time. As samples are stored in the leaves of the tree, they can be "lost" or "shadowed" as a decision boundary moves and abandons some of the samples that were placed because of that boundary. These samples are useless for classification, and need to be re-inserted into the tree. To do this, a linked list of nodes of the tree is maintained, and at each time step a single sample is selected from the tree, removed from that leaf, and the tree is retrained with the sample. Over time, all of the samples are repeatedly removed and retrained with this process.

This incremental process causes samples to be redistributed through the tree as nodes slowly freeze, and it also prevents redundant samples from being stored in different leaves. A side benefit is that variables in the tree that are being estimated are repeatedly exposed to a randomized mix of samples that are relevant for classification (because of the pruning process). This seems to give better and faster parameter estimation than running repeated epochs of training data – reusing samples offers a speedup in training.

2.3.3 Freezing and Spawning

The primary motivation for spawning child nodes on the tree is to defray computation time. If a leaf cluster collects a large number of samples after many updates, the linear search of samples that must be done at that leaf in order to classify an input

vector can be significant. Thus, if a leaf node collects greater than σ samples, the leaf spawns a child node in order to reduce the liner search length needed for classification. The samples are incrementally redistributed into the child node through the sample retraining process, and the maximum length of linear searches is reduced.

Spawning is only done after a node has frozen completely. This is because a node that is not frozen cannot be counted on to consistently assign a samples to the same virtual classes over time, and this introduces a large amount of noise for the child nodes. This means that the tree grows downward incrementally, only spawning one level of leaves at a time, and freezing each level at the same time as well.

Chapter 3

Experiments

In order to illustrate the operation of the HSM tree, we have three experiments described below. Together, these three experiments demonstrate the correct operation, scalability, and real-world usage of the HSM tree algorithm described above. The first experiment, using very simple synthetic data, demonstrates the tree learning a simple classification problem over time. The decision boundaries generated by the tree are displayed as training goes on, and the experiment illustrates that the tree is able to approximate the decision boundaries of a simple problem.

The second experiment works with real face images, demonstrating face recognition across 143 different people. This experiment illustrates both the performance of this algorithm and the ability for this algorithm to scale up to very large data sets while maintaining very fast retrieval times. A very large amount of very high dimensional data is used, and the algorithm retains its speed and gives a high classification rate.

The third experiment demonstrates this algorithm working in real time, attached

to a camera and taking input from the keyboard as well. The real-time operation of this system is illustrated as the system learns to recognize five people and seven objects over a course of two hours, given training input from the keyboard.

3.1 Synthetic Data

In order to demonstrate the operation of the HSM tree in a manner that can be visualized easily, synthetic data was generated in a two-dimensional problem. Figure 3.1 displays the decision boundaries of a three-class two-dimensional problem. Samples are uniformly drawn from the sample space shown with the training class determined from the sample's location in figure. The tree is given a new sample mapping at each time step. The Y-space class labels are the numerical labels 1, 2, and 3.

The tree is set to have a maximum of 2 clusters at each node ($q = 2$) which means that at each node the tree produces a 1-dimensional discriminant projection. The freeze value η is set to $\eta = 200$, which means that the root node freezes after 800 time steps, and child nodes that are created right after the root node freezes become frozen after 1600 time steps. The freezing schedule is shown in Figure 3.2. This value of η was chosen somewhat arbitrarily – it is a large enough number that each node will see a good amount of samples from each class before it freezes, and that should be good enough for our purposes. The main consideration in choosing a value of η is to be certain that it isn't too small, as too small a value will cause the system to freeze with very poor X-space partitions. The decision boundaries of the tree over different stages of incremental training across 2000 time steps are shown in the figures below.

After 2000 training samples, the tree has a depth of three with seven nodes and 404 samples spread across eight leaf clusters. If the tree had continued to train, it would have continued to spawn leaves, continually refining the decision boundaries. This is because the different regions are adjacent, and so samples are continually added in an attempt to approximate the straight line boundaries.

As the training progresses, the decision boundaries become more refined. Because the HSM tree uses the nearest-neighbor method at each leaf node, the decision boundaries are very jagged, as the tree overfits the low-dimensional data. After steps 800 and 1800, the tree spawns new levels of nodes, and larger refinements in the decision boundaries become apparent because of this. This is the coarse-to-fine growth of the tree – as the tree grows, the boundaries become more accurate and more complex.

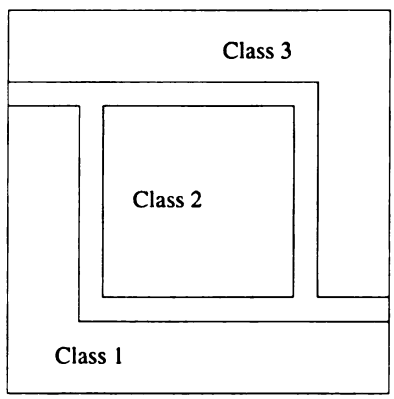

Figure 3.1: The decision boundaries of a 3-class 2-dimensional classification problem. Samples are selected with uniform probability from the space above.

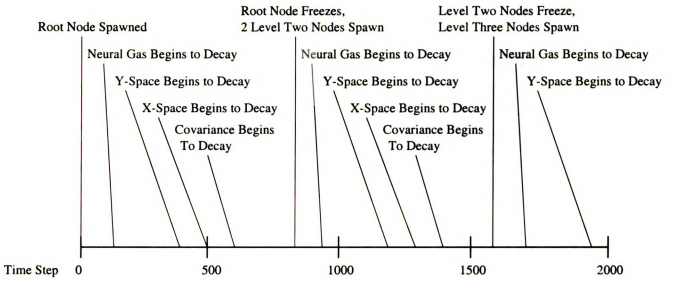

Figure 3.2: This is a timeline of the freezing and spawning patterns of the tree over 2000 time steps. The freezing parameter η is set to 200.

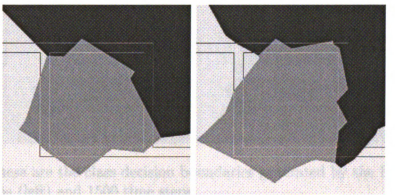

Figure 3.3: These are the class decision boundaries generated by the HSM tree after 100 time steps (left) and 300 time steps.

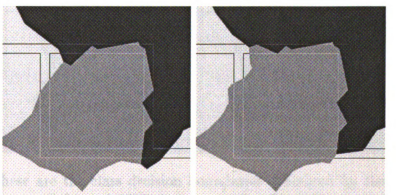

Figure 3.4: These are the class decision boundaries generated by the HSM tree after 500 time steps (left) and 700 time steps.

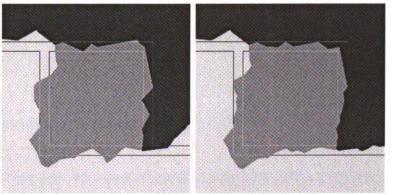

Figure 3.5: These are the class decision boundaries generated by the HSM tree after 900 time steps (left) and 1100 time steps. The decision boundaries are more refined because the root node froze at time step 800, and two child nodes have been spawned.

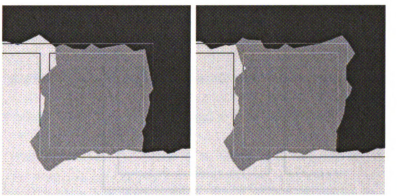

Figure 3.6: These are the class decision boundaries generated by the HSM tree after 1300 time steps (left) and 1500 time steps.

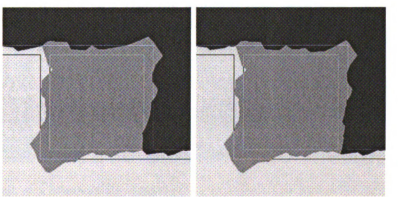

Figure 3.7: These are the class decision boundaries generated by the HSM tree after 1700 time steps (left) and 1900 time steps. Smaller refinements in the decision boundaries are noticeable because the second set level of nodes froze at time step 1800 and a third level of nodes was spawned.

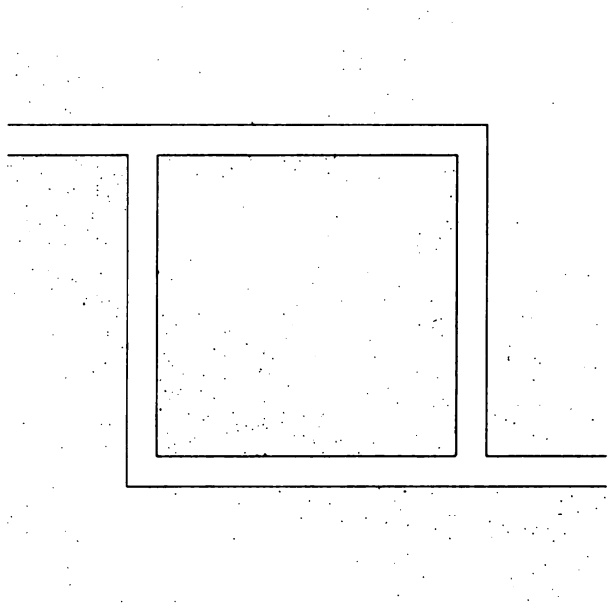

Figure 3.8: These are the 404 training samples that were kept by the tree in leaf nodes after 2000 time steps. They were selected using the sample pruning method described previously.

3.2 Simulated Face Recognition from Video Sequences

3.2.1 Experiment Description

The SAIL tree was used as a part of a simulated robot control system to demonstrate online face recognition and gender recognition using image sequences. This was done to demonstrate the ability of this system to scale up to very long video sequences of very high dimensionality. Data sets of the size used here are typical of what this system might encounter in training in a real-time situation,

Four video sequences of 143 different people were collected and digitized, and the images were cropped close to the individual's faces, and covered with a triangular "foveal mask" that simulates the effect of a fovea and weights the face region of a subject more heavily than the background. Each video sequence was around 50-60 frames in length, with a resolution of 88×64. In total, 33,889 images were collected across all of these sequences.

Figure 3.9: This is a temporally subsampled sequence used in training. The subject's face enters the view and comes to the center of the image, remains there for a while, and then leaves the view. A triangular foveal mask is centered over the image.

Three of the sequences for each individual were used for training for each task,

Figure 3.10: These are a selection of 16 images of the 143 people used for this test. The testing set varied across race, gender and age by a very wide amount.

and the fourth was used for testing. We were interested in training the system online to perform a face recognition task, where the control system is trained online and incrementally to respond with the correct name or gender of an individual if asked. Training involves presenting each sequence to the tree and imposing actions upon the system in a manner of supervised learning. Testing is done by presenting sequences of images and stimuli and recording if the expected responses are displayed.

The two tasks were face recognition and gender recognition. The face recognition task involved asking the system to provide an identifying number that corresponds to a person's identity when it is asked "Who?". The gender recognition task involved providing a number that corresponds to the correct gender of the individual when the system is asked "Gender?". Each sequence is 50-60 images, and at each time step an image was presented to the tree, along with a numerical input sensor which corresponds to "Who?", "Gender?", or to no question. The numerical sensor input was a vector of length 500 with all values set to the same number. These inputs are concatenated into a single vector, which is of length $88 \times 64 + 500 = 6132$, which is the sensory feedback at each time step. The system output at each time step was either a value of 0 which indicates no action, or a value that corresponded to the name of

an individual or to a gender.

Training Session	Image Sequence	Task Trained
1	Sequence 1	Face Recognition
2	Sequence 2	Face Recognition
3	Sequence 3	Face Recognition
4	Sequence 1	Gender Recognition
5	Sequence 2	Gender Recognition
6	Sequence 3	Gender Recognition

Table 3.1: This is an outline of one epoch of training. Three training sequences were collected for each person, and these sequences are recycled between the two tasks trained.

The numerical labels used for this experiment are inappropriate for use as Y-space labels, because the single numerical values provide no information for the the Neural Gas clustering algorithm. We produced Y-space labels for each class by taking an incrementally updated mean of each class. This means that with 143 people to identify, two genders, and an output of no action, there were 146 different classes. Each class had a mean vector which was incrementally updated as samples for that class were received by the system. Each vector was the mean of inputs belonging to the class that the tree was trained on. Each individual sample was also marked with the actual numerical label of the class that it belonged to, so that classification could be done quickly in the Search-Tree procedure.

The system was presented with each sequence of images, and trained sequentially across the whole training set. In each training sequence, the individual's face enters the image, and then "Who?" or "Gender?" was given as an input to the tree. The "Who?" question was represented by setting the numerical input sensor vector to 500. The "Gender?" question was represented by setting the numerical input sensor

to 700. At the time step following the question, the correct response corresponding to identity or gender was imposed. At all other times, the question input was set to 0 (no question), and an output of 0 (no action) was imposed. For testing, the person's face entered the image, and then "Who?" or "Gender?" was given to the numerical sensor. The output at every step of the sequence was recorded, and correct identification of the individual was assessed if the system replied with the correct name or gender at the correct time, and didn't perform any inappropriate actions at other times.

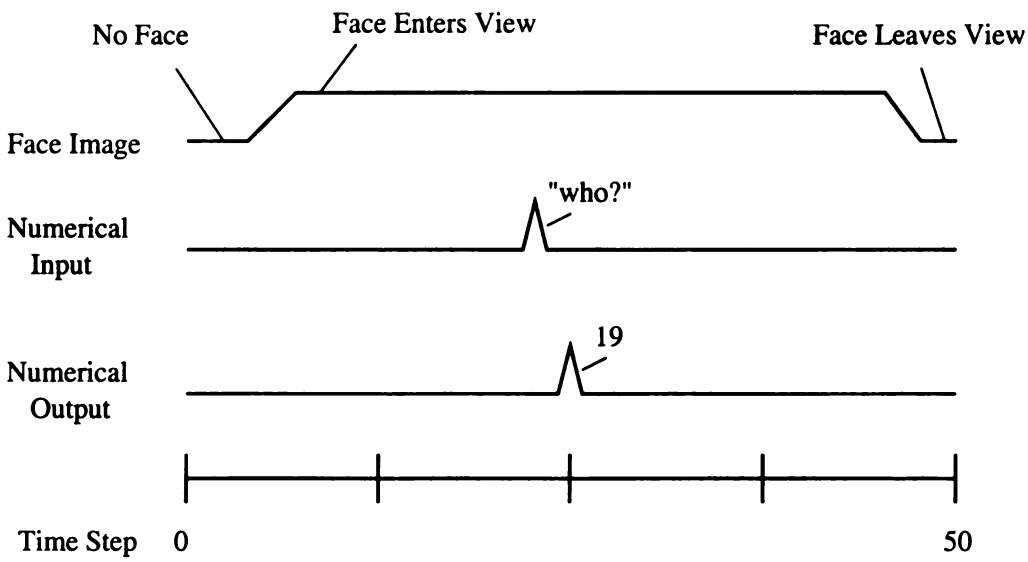

Figure 3.11: A timeline for a typical question-response training session for face identification of person number 19. The face image enters the view, and then a "who?" question is asked on the numerical sensor. The response 19 is trained for the immediately following time step. The two-step memory mechanism means that the system learns to respond when the previous numerical input was a question.

The numerical input sensor vector was given a large size vector size and large responses in order to have equal weighting against the image vector. If the numerical input sensor vector were one-dimensional and had a small response range, then it would not be a salient input for the system.

The HSM tree makes up the core of the simulated control system. When being trained, at each time step the current image and numerical input sensor value is taken as an input along with an output vector. At each time step of testing, the system takes the current image and numerical input sensor value, and outputs a control signal.

Because we wanted to have the system respond one time step after the "Who?" question is given, we equipped the system with a small memory of one time step. This was done by setting the input to the tree to be a vector containing the current image and numerical input sensor concatenated with the previous image and numerical input sensor. This meant that the current state was composed of the current and previous sensor inputs, making the input dimensionality of the tree $2 \times 6132 = 12264$. This "one step memory" meant that the system can respond to stimuli up to one step after it is observed. Thus, when the system is trained to respond one time step after a numerical input of "who?" is given, the system ends up mapping a tree input with a numerical input of "who?" in the previous sensor input to an identifying response. Longer memories would allow more prolonged responses and more complex tasks, but for this experiment, one step was all that was necessary.

This interaction framework makes up a primitive way for a real-time vision-based robot to be trained online. Face recognition was chosen here as a test bed because it is a popular and easily understood area of research right now, but this system is extensible to online training in other kinds of problems. The learning limitations are only what can be represented by the interaction framework, but the real-time operation of the tree makes all kinds of online learning possible.

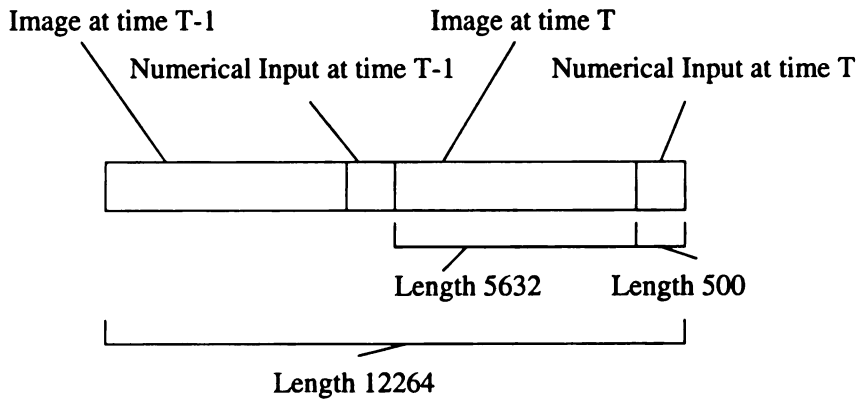

Figure 3.12: This is the makeup of the input vector to the HSM tree at each time step. The previous and current sensory inputs are concatenated into one vector, giving the system a memory of one time step.

3.2.2 Experiment Results

The algorithm was run for five epochs of training, where in each epoch each face identification and gender response was trained on three different image sequences of each person. This gives 50,833 time steps in each epoch. The tree had a maximum of 9 clusters ($q = 9$) for each node, and the freeze weight η was set to 15,000. This value of η is large enough that the system will see an example of each class once before the node freezes. If η is set much higher, performance is not affected, but it takes a much longer time for the tree's decision boundaries to converge.

After the fifth epoch, each of the 14 nodes in the tree was frozen, and the tree had stopped spawning nodes. Resubstitution error for both the face recognition and gender identification portions was zero. At this point, the system was tested using a hold-out set of image sequences with different values of ϵ and k for the Search-Tree procedure. There were 16,945 images in the hold-out set. The results are presented in figures 3.14 and 3.15. The best result for face recognition was 7 errors out of 143 with parameters $k = 4$ and $\epsilon = 1.5$. The best result for gender identification was 4

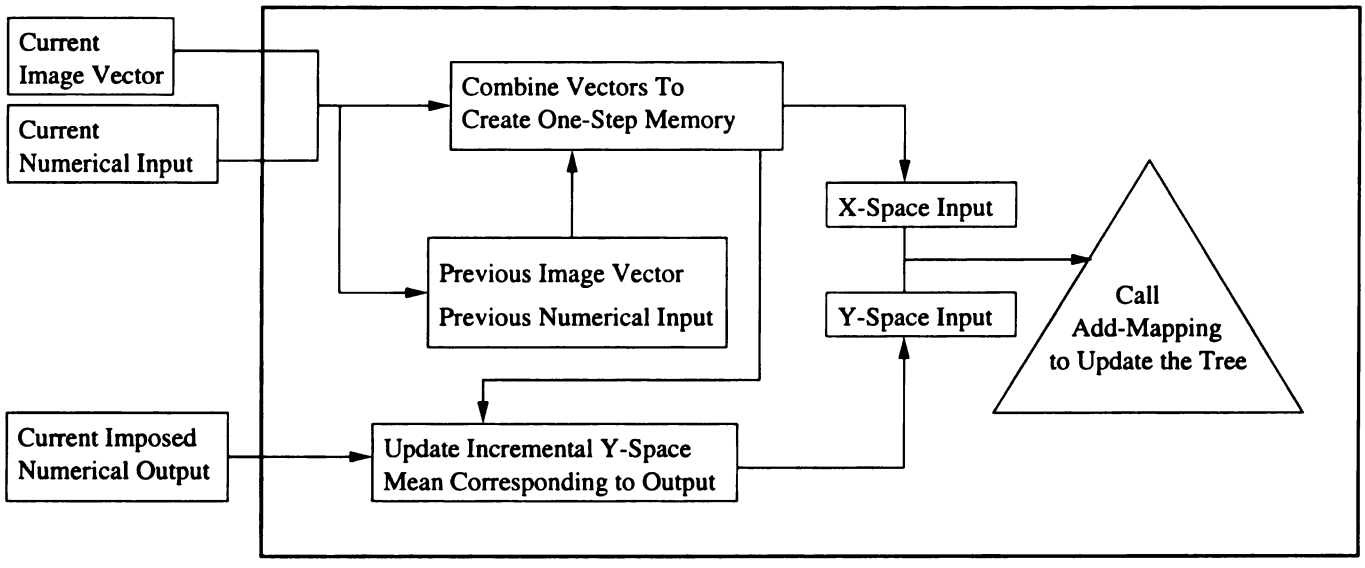

Figure 3.13: This is a data flowchart of how one time step of training on this framework works. The current image and numerical input are combined with the previous image and numerical input to create an X-space input vector for the tree, and the current sensory inputs are stored for the next time step as well. The imposed effector output class is used along with the X-space tree input to incrementally update the class means used for Y-space class labels in the tree. The mean of the imposed effector output class is then used as a Y-space input to the tree. Update-Mapping is then called on the tree.

errors out of 143 with parameters $k = 7$ and $\epsilon = 1.5$. Different values of k and ϵ have a large effect on the performance of the tree, and it can be seen that small values of k with large values of ϵ actually decreases the performance. This is because large ϵ values cause the search to split at higher levels of the tree too quickly, leading to useless parallel searches.

The face identification and gender identification tasks were tested against a nearest neighbor classifier for comparison. The nearest neighbor classifier was provided with the individual training samples used only during the question-and-answer phases of training, and the "no action" images were not used. This made for a training set of 858 images. The nearest neighbor classifier had 4 errors out of 143 for face identification,

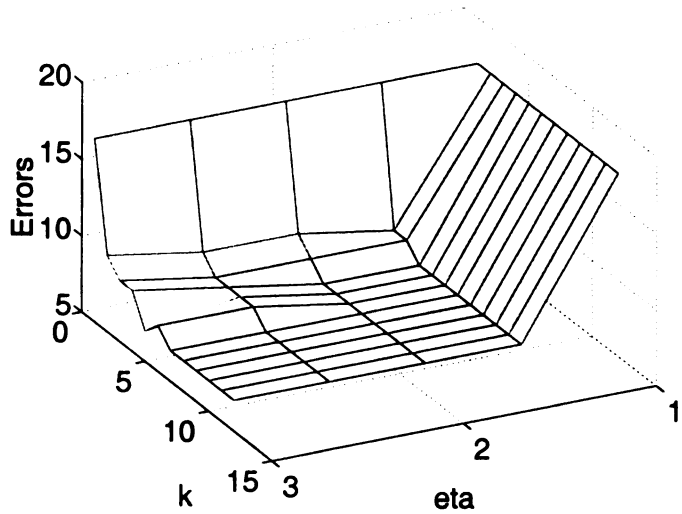

Figure 3.14: Face recognition performance of the HSM tree over different k and ϵ values. As can be seen here, larger k values give consistently better performance with larger ϵ values, but large ϵ values with small k values will decrease performance.

and 2 errors out of 143 for gender recognition. This shows that the HSM tree's performance is comparable to a nearest neighbor classifier for this specific problem of face recognition.

The advantage of the HSM tree over a nearest neighbor classifier is that it is fast, can be trained online, and is scalable to large data sets. The average times for Search-Tree to execute over the 16,945 training images for both tasks is illustrated in 3.16, with a range of times from 0.029 seconds to 0.04 seconds. The average time for Add-Mapping to execute over all of the training samples across all five epochs was 0.15 seconds for one mapping. This compares with an average nearest-neighbor search time over the 858 samples of 0.2 seconds. All of these tests were done on a 400Mhz Intel Pentium II system with 512MB of RAM.

The speed differences are impressive because the HSM tree was generated online, while the nearest neighbor classifier was generated in batch with total knowledge

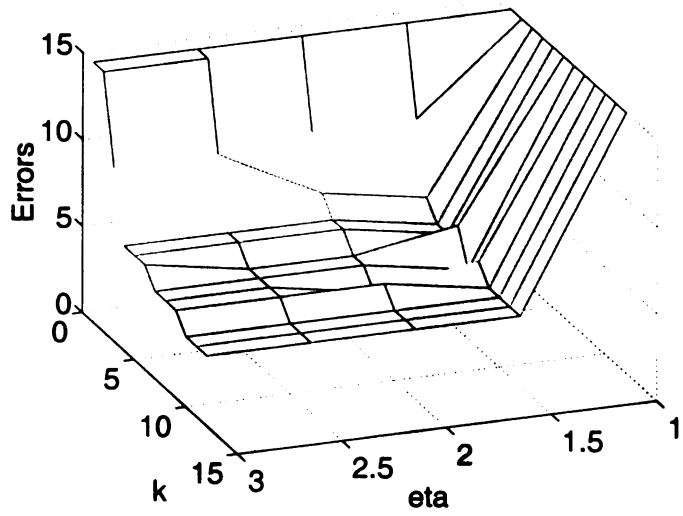

Figure 3.15: Gender recognition performance of the HSM tree over different k and ϵ values.

of which of the 50,833 training samples to exclude. Additionally, the HSM tree maintains a tree structure, which in most cases should give it retrieval times that are logarithmically proportional to the number of samples stored in the tree. For real-time applications, the HSM tree is definitely a feasible method – fast, online, and scalable.

A function not displayed in this system is a "reject" option – an ability to reject faces that are not recognized, or an ability of the system to express low confidence in the response given. If presented with a novel face and asked to identify, the system would simply return the closest answer that it could find. Traditionally, such a capability is arrived at either by building an "imposter model" of imposter faces, and then training them against the system, or else using a manually tuned threshold found with an ROC curve or a similar method. Both of these methods represent very task-specific approaches to face recognition, and are not compatible with the developmental paradigm with which this algorithm is being developed. A

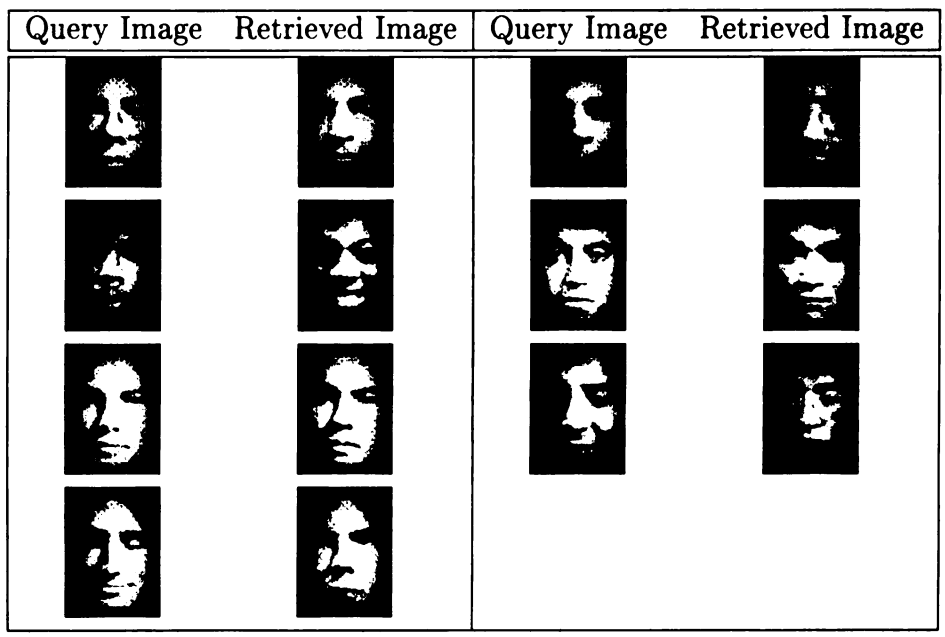

Table 3.2: These are the seven face identification errors made by the HSM tree. The queried images are on the left, and the images that the tree returned are on the right.

Method	Face Results	Gender Results
HSM Tree	95.1% (Best)	97.2% (Best)
Nearest Neighbor	97.2%	98.6%

Table 3.3: These are the best results of the HSM tree for face and gender recognition compared to the nearest neighbor method.

reject option is a valid and important function, but it will be a future work, trained to the system as a behavior by a trainer, and not constructed intrinsically in the algorithm.

Method	Search Time	Training Time
HSM Tree	0.04s (Worst)	0.15s
Nearest Neighbor	0.2s	0s

Table 3.4: These are the worst average time results for searching and training the HSM tree for one sample compared with times for the Nearest Neighbor classifier with 858 samples. Training the Nearest Neighbor classifier for one sample takes no time because a new sample is simply kept. Note that the HSM tree's combined training and search times are less than the search time alone of the Nearest Neighbor classifier.

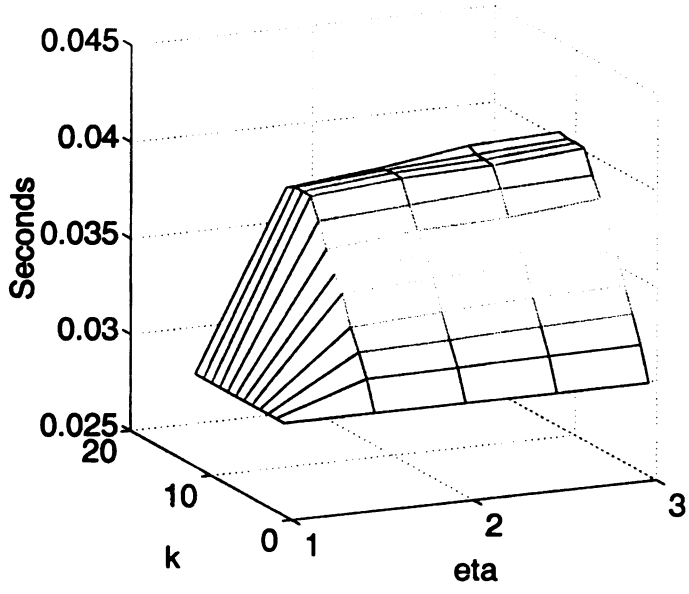

Figure 3.16: The time average in seconds that it would take to perform a search on the tree, given different k and ϵ values. Times range from 0.029 seconds to 0.04 seconds for a search.

3.3 Real Time Face and Object Recognition

In order to supplement the simulated recognition tests described above, the above system was used for real-time online incremental training for recognition of five people and seven objects using a camera and frame grabber. The purpose of this system was to demonstrate an actual proof-of-concept system that is trained and is tested in real time given real interaction with the trainer. The objects and faces trained were easily discriminable, but the purpose of this last experiment was not to demonstrate classification capability, but instead only the real-time operation of this system.

The same question-response model of training outlined above was used here, with the only change to the system being that RGB color images with a superimposed foveal mask at a resolution of 80×60 were used. This makes the input dimensionality of the tree $((80 \times 60) \times 3 + 500) \times 2 = 29800$.

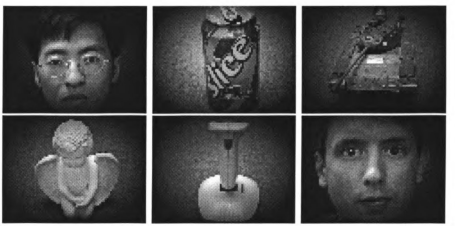

Figure 3.17: Several of the objects used in real-time online training of the algorithm.

The system was trained over the course of two hours, with three training sessions for each person or object. In sessions for face recognition, individual's eyes were lined up with two marks on the computer screen and the correct output for the individual's

identity was imposed from 5 to 10 times. For object recognition, the object was placed on a uniform brown background in the center of the field of view and the correct response was trained 5 to 10 times. Care was taken to make certain that the objects were in roughly the same positions for training and testing, but explicit hand-tuned image registration was not done for either the faces or the objects. Training for each face and object was interleaved, with three consecutive sessions of training the 12 classes.

After the three training sessions, each recognition of each person and object was tested once, with a correct recognition in each case. The frame rate of the system ran around 5-10 frames per second on a 400 Mhz Intel Pentium II system with 512MB of RAM. Training was a simple process, and this final experiment exhibits a functioning real-time system operating on a simple problem.

Chapter 4

Conclusions

The HSM tree and framework presented here represent a very simple developmental learning system. The above experiments demonstrate that the HSM tree is scalable to very large data sets and is capable of running in real time with very high dimensional data. Face recognition from video sequences across 143 different subjects and with 33,889 frames of video is demonstrated with a correct identification rate of 95.1% and an average retrieval time of 0.04 seconds per image. Real-time operation is demonstrated as well, with the system processing and being trained at a rate of 5-10 frames per second.

The next steps in this work are bringing this software to a robot system and designing a training framework for the system. This is an early work, but it a first step on designing a truly reactive and flexible machine intelligence system. Many new problems and a rich array of research topics lie ahead of this project, including finding new ways to automate parameter selection in the HSM tree, integration of this work into a temporal learning model, and adaptation of reinforcement learning

methods to our framework.

Bibliography

[1] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees.* Chapman & Hall, New York, 1993.

[2] R. Brooks. Cog, a humanoid robot. Technical report, MIT Artificial Intelligence Laboratory, 1997. private communication.

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

[4] M. Dorigo and M. Colombetti. Robot shaping: Developing autonomous agents through learning. *Artificial Intelligence*, 71(2):321–370, 1994.

[5] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[6] J. E. Laird, E. S. Yager, M. Hucka, and C. M. Tuck. Robo-soar: An integration of external interaction, planning, and learning using Soar. *Robotics and Autonomous Systems*, 8:113–129, 1991.

[7] D. B. Lenat. CYC: A large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11):33–38, 1995.

[8] T. M. Martinetz, S. G. Berkovich, and K. J. Schulten. 'neural-gas' network for vector quantization and its application to time-series prediction. *IEEE Trans. on Neural Networks*, 4(4):558–569, 1993.

[9] G. A. Miller. Worknet: A lexical database for English. *Communications of the ACM*, 38(11):39–41, 1995.

[10] Sreerama K. Murthy, Simon Kasif, and Steven Salzberg. A system for induction of oblique decision trees. *Journal of Artificial Intelligence Research*, 2:1–33, 1994.

[11] D. A. Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In D. Touretzky, editor, *Advances in Neural Information Processing*, volume 1, pages 305–313. Morgran-Kaufmann Publishers, San Mateo, CA, 1989.

[12] J. Quinlan. Introduction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[13] J. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Mateo, CA, 1993.

[14] Reid Simmons, Richard Goodwin, Karen Zita Haigh, Sven Koenig, and Joseph O'Sullivan. A modular architecture for office delivery robots. *Autonomous Agents*, pages 245–252, Feb. 1997.

[15] D. Swets and J. Weng. SHOSLIF-O: SHOSLIF for object recognition (Phase I). Technical Report CPS 94-64, Department of Computer Science, Michigan State University, East Lansing, MI, Dec. 1994.

[16] D. Swets and J. Weng. SHOSLIF-O: SHOSLIF for object recognition (Phase II). Technical Report CPS 95-39, Department of Computer Science, Michigan State University, East Lansing, MI, Oct. 1995.

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

[18] C. Watkins. Learning from delayed rewards. Technical report, PhD thesis, King's College, Cambridge, England, 1989.

[19] J. Weng and S. Chen. Incremental learning for vision-based navigation. In *Proc. Int'l Conf. Pattern Recognition*, volume IV, pages 45–49, Vienna, Austria, Aug. 25-30 1996.