Developmental Humanoids: Humanoids that Develop Skills Automatically

Juyang Weng, Wey S. Hwang, Yilu Zhang, Changjiang Yang, and Rebecca J. Smith

Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA
{weng, hwangwey, zhangyil, yangcha1,smithr46}@cse.msu.edu
http://www.cse.msu.edu/ weng/research/LM.html

Abstract. It is desirable for humans to control humanoids through high-level commands, but it is too tedious for humans to issue commands for every detailed action for every fraction of a second. However, it is extremely challenging for humans to program a humanoid robot to such a sufficient degree that it acts properly in typical unknown human environments. This is especially true for humanoids due to the very large number of redundant degrees of freedom and a large number of sensors that are required for humanoids to work safely and effectively in the human environment. How can we address this fundamental problem? Motivated by human mental development from infancy to adulthood, we enable robots develop its mind automatically, through online, real time interactions with its environment. Humans mentally "raise" the robot through "robot sitting" and "robot schools" instead of task-specific robot programming. The SAIL developmental robot that has been built at MSU is an early prototype of such new generation of robots.

1 Introduction

The conventional mode of developmental process for a robot is not automatic — the human designer is in the loop. A typical process goes like this: Given a robotic task, it is the human designer to understand the task. Based on his understanding, he comes up with a representation, chooses a computational method, and writes a program that implements his method for the robot. The representation reflects very much the human designer's understanding of the robot task. During this developmental process, some machine learning might be used, during which some parameters are adjusted according to the collected data. However, these parameters are defined by the human designer's representation for the given task. The resulting program is for this task only, not for any other tasks. If the robotic task is complex, the capability of handling variation of environment is very much limited by the human designed task-specific representation. This manual development paradigm has met tremendous difficulties for tasks that require complex cognitive and behavioral capabilities, such as many sensing and behavioral skills that a humanoid must have in order to execute human high-level commands, including autonomous navigation, object manipulation, object delivery, target finding, human-robot interaction through gesture in unknown environment. The high degree of freedom, the redundant manipulators, and the large number of effectors that a humanoid has, plus the multimodal sensing capabilities that are required to work with humans further increase the above difficulties. Humanoid robots have a great potential to work in human environment, but the complex and changing nature of human environment has made the issue of automatic mental development — the way human mind develops — more important than ever.

Many robotics researchers may believe that human brain has an innate representation for the tasks that humans generally do. However, recent studies of brain plasticity have shown that our brain is not as task-specific as commonly believed. There exist rich studies of brain plasticity in neuroscience, from varying extent of sensory input, to redirecting input, to transplanting cortex, to lesion studies, to sensitive periods. Redirecting input seems illuminating in explaining how much task-specific our brain really is. For example, Mriganka Sur and his coworkers rewired visual input to primate auditory cortex early in life. The target tissue in the auditory cortex, which is supposed to take auditory representation, was found to take on *visual* representation instead [10]. Furthermore, they have successfully trained the animals to form visual tasks using the rewired auditory cortex [8]. Why are the self-organization schemes that guide development in our brain so general that they can deal with either speech or vision, depending on what input it takes through the development? Why are robots that are programmed using human designed, task-specific representation do not do well in complex, changing, or partially unknown or totally unknown environment? What are the self-organization schemes that robots can use to automatically develop its mental skills through interactions with the environment? Is it more advantageous to enable robots to automatically develop its mental skills than to program robots using human-specified, task-specific representation?

Since 1996 [13], we have been working on a robotic project called SAIL (short for Self-organizing, Autonomous, Incremental Learner) and SHOSLIF is its predecessor [14]. The goal of the SAIL project is to *automate* the process of mental *development* for robots.

Our developmental algorithm is motivated by automatic human mental development. The primate nervous system may operate at several levels:

1. Knowledge level (e.g., symbolic skills, thinking skills, general understanding of the world around us, learned part of emotions, and rich consciousness).
2. Inborn behavior level (e.g., sucking, breathing, pain avoidance and some primitive emotions in neonates). In neurons, they are related to synapses at the birth time.
3. Representation level (e.g., how neurons grow based on sensory stimuli).
4. Architecture level (corresponding to anatomy of an organism. E.g., a cortex area is prepared for eyes, if everything is developed normally).
5. Timing level (the time schedule of neural growth of each area of the nervous system during development).

Studies in neuroscience seem to show that all of the above 5 levels are experience-dependent [1]. In fact, experience can shape all these levels to a very great extent. But it seems that our gene has specified a lot for Levers 2 through 5. The level 1 is made possible by levels 2 through 5 plus experience; but level 1 is not wired in. Thus, levels 2 through 5 seem what a programmer for a developmental algorithm may want to design — but not rigidly — experience dependent. The designer of a developmental robot may have some information about the ecological condition of the environment in which the robots will operate, very much in a way that we know the ecological condition of typical human environment. However, the designer does not known what particular tasks that the robot will end up learning.

According to the above view, our SAIL developmental algorithm has some "innate" reflexive behaviors built-in. At the "birth" time of the SAIL robot, its developmental algorithm starts to run. This developmental algorithm runs in real time, through the entire "life span" of the robot. In other words, the design of the developmental program cannot be changed once the robot is "born," no matter what tasks that it ends up learning. The robot learns while performing simultaneously. The innate reflexive behaviors enable it to explore the environment while improving its skills. The human trainers train the robot by interacting with it, very much like the way human parents interact with their infant, letting it seeing around, demonstrating how to reaching objects, teaching commands with the required responses, delivering reward or punishment (pressing "good" or "bad" buttons on the robot), etc. The SAIL developmental algorithm updates the robot memory in real-time according to what was sensed by the sensors, what it did, and what it received as feedback from the human trainers.

What is the most basic difference between a traditional learning algorithm and a developmental algorithm? Automatic development does require a capability of learning but it requires something more fundamental. A developmental algorithm must be able to learn tasks that its programmer does not know or even cannot predict. This is because a developmental algorithm, once being designed before robot "birth," must be able to learn new tasks and new skills without requiring re-programming. The representation of a traditional learning algorithm is designed by a human for a given task but that for a developmental algorithm must be automatically generated. This basic capability of human developmental algorithm enables humans to learn more and more new skills without a need to change the design of his brain.

However, the motive of developmental robots is not to make robot more difficult to program, but relatively easier instead. The task nonspecific nature of a developmental program is a blessing. It relieves human programmer from the daunting tasks of programming task-specific visual recognition, speech recognition, autonomous navigation, object manipulation, etc, for unknown environments. The programming task for a developmental algorithm concentrates on self-organization schemes, which are more manageable by human programmers than the above task-specific programming tasks for unknown or partially unknown environments.

Although developmental program for a robot is very much a new concept [13], a lot of well-known self-organization tools can be used in designing a developmental robot. In this paper, we informally describe the theory, method and experimental results of our SAIL-2 developmental algorithm tested on the SAIL robot. In the experiments presented here, our SAIL-2 developmental algorithm was able to automatically develop low-level vision and touch-guided motor behaviors that we humans like the robots to autonomously handle by themselves after receiving human commands.

[1] The literature about this subject is very rich. A good start is "Rethinking innateness" [3] (pages 270-314).

Table 1. Comparison of Approaches

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

2 The Developmental Approach

Automatic development requires a drastic departure from the current task-specific nature of all the existing approaches. Table 1 outlines the major characteristics of existing approaches to constructing an artificial system and the new developmental approach. The developmental approach relieves humans from explicit design of (a) any task-specific representation and knowledge and (b) system behavior representation, behavior modules and their interactions.

2.1 AA-learning

A robot agent *M* may have several sensors. By definition, the *extroceptive*, *proprioceptive* and *interoceptive* sensors are, respectively, those that sense stimuli from external environment (e.g., visual), relative position of internal control (e.g., arm position), and internal events (e.g., internal clock).

The operational mode of automated development can be termed AA-learning (named after *a*utomated, *a*nimal-like learning *without claiming to be complete*) for a robot agent.

Definition 1. *A robot agent M conducts AA-learning at discrete time instances, $t = 0, 1, 2, ...$, if the following conditions are met: (I) M has a number of sensors, whose signal at time t is collectively denoted by $x(t)$. (II) M has a number of effectors, whose control signal at time t is collectively denoted by $a(t)$. (III) M has a "brain" denoted by $b(t)$ at time t. (IV) At each time t, the time-varying state-update function f_t updates the "brain" based on sensory input $x(t)$ and the current "brain" $b(t)$:*

$$b(t+1) = f_t(x(t), b(t)) \tag{1}$$

and the action-generation function g_t generates the effector control signal based on the updated "brain" $b(t+1)$:

$$a(t+1) = g_t(b(t+1)) \tag{2}$$

where $a(t+1)$ can be a part of the next sensory input $x(t+1)$. (V) The "brain" of M is closed in that after the birth (the first operation), $b(t)$ cannot be altered directly by human teachers for teaching purposes. It can only be updated according to Eq. (1).

As can be seen, AA-learning requires that a system should not have two separate phases for learning and performance. An AA-learning agent learns while performing.

2.2 Outline of developmental architecture

Fig. 1 gives a schematic illustration of the implemented architecture of SAIL-2 robot. The current implementation of the SAIL-2 system includes extroceptive sensors and proprioceptive sensors. The color stereo images come from two CCD cameras with wide-angle lens. See the appendix for the vector representation of each image frame of real-time video. 32 touch/switch sensors are equipped for the robot. Each eye can pan and tilt independently and the neck can turn. A six-joint robot arm is the robot's manipulator. In the SAIL-2 system, the "brain" contains a working memory called state $w(t)$ and long-term memory as a tree. The state keeps information about the previous actions (context). If $x(t)$ is the vector of all sensory inputs and action outputs at time t, the state is a long vector $w(t) = (x(t-1), x(t-2), ..., x(t-k))$, where k is the temporal extent of the state. Typically, to save space, we make k small for sensory input but large for action so that action keeps more context. This gives a way of updating the working memory of the brain by function f_t. The updating of long-term memory (part of f_t) as well

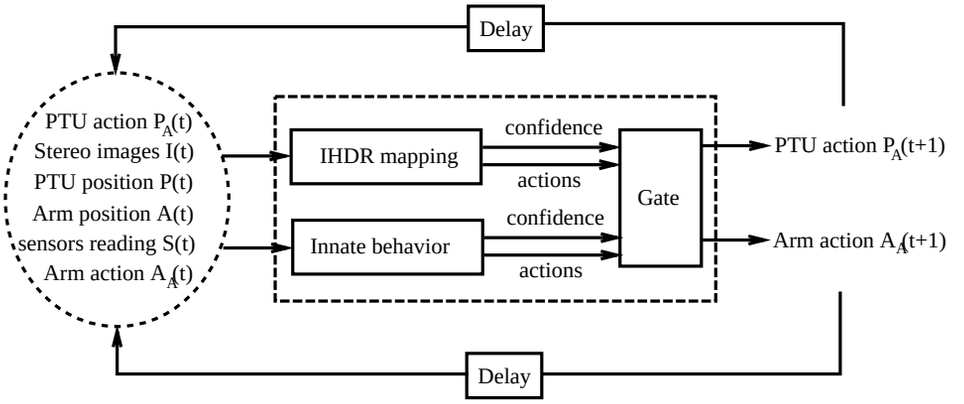


Fig. 1. A schematic illustration of the architecture of SAIL-2 robot. The sensory inputs of the current implementation include stereo images, position of the pan-tilt unit (PTU) for each camera, touch/switch sensors, and the position of arm joints, as well as the action of every effector. The gate is to select an appropriate action from either cognitive mapping (learned) or innate behaviors (programmed in) according to the confidence values.

as the generation of action (what g_t does) are realized by the IHDR mapping in Fig. 1. The IHDR mapping accepts $(x(t), w(t))$ as input and it generates $a(t+1)$ as the output, as well as updating the long term memory of $b(t+1)$, for each time t. The IHDR is a general mapping approximator and will be discussed in the following section.

The innate behavior is programmed before the machine is "born." The current implemented built-in innate behavior is the motion detection and tracking mechanism for vision. When an object is moving in the scene, the absolute difference of each pixel between two consecutive image frame gives another image called intensity-change image, which is directly mapped to the control of PTU of each eye, using also the IHDR mapping technique but this mapping was generated in a "prenatal" offline learning process. In other words, this offline learning generates innate behaviors in the newborns. Our experience indicated that it is much computationally faster and more reliable to generate innate behavior this way than explicitly finding the regions of moved objects through explicit programming.

The online learned IHDR mapping and the innate behavior may generate PTU motion signal at the same time. The resolution of such conflict is performed by the gate system. In the current implementation, the gate system performs subsumption. Namely, the learned behavior takes the higher priority. Only when the learned behavior does not produce actions, an the innate behavior be executed. A more resilient way of conducting subsumption is to use the confidence of each action source, but this subject is beyond the scope of this article.

3 The Mapping Engine

3.1 IHDR algorithm

Therefore, the major technical challenge is to incrementally generate the IHDR mapping. In the work reported here, online training is done by supplying desired action at the right time. When action is not supplied, the system generates its own actions using the IHDR mapping updated so far. In other words, the robot runs in real time. When the trainer likes to teach the robot, he pushes the effector, through the corresponding touch sensor that directly drives the corresponding motor. Otherwise, the robot runs on its own, performing.

Thus, the major problem is to approximate a mapping $h : \mathcal{X} \mapsto \mathcal{Y}$ from a set of training samples $\{(x_i, y_i) \mid x_i \in \mathcal{X},\ y_i \in \mathcal{Y},\ i = 1, 2, \ldots, n\}$, that arrives one pair (x_i, y_i) at a time, where $y_i = *$ if y_i is not given (in this case, the approximator will produce estimated y_i corresponding to x_i). The mapping must be updated for each (x_i, y_i). If y_i was a class label, we could use linear discriminant analysis (LDA) [4] since the within-class scatter and between-class scatter matrices are all defined. However, if y_i is a numerical output, which can take any value for each input component, it is a challenge to figure out an effective discriminant analysis procedure that can disregard input components that are either irrelevant to output or contribute little to the output. We introduce a new hierarchical statistical modeling method. Consider the mapping $h : \mathcal{X} \mapsto \mathcal{Y}$, which is to be approximated by a regression tree, called incremental hierarchical discriminating regression (IHDR) tree, for the high dimensional space $\mathcal{X}$. Our goal is to automatically derive discriminating features although no class label is available (other than the numerical vectors in space $\mathcal{Y}$). In addition, for real-time requirement, we must process each sample (x_i, y_i) to update the IHDR tree using only a minimal amount of computation (e.g., in 0.05 second).

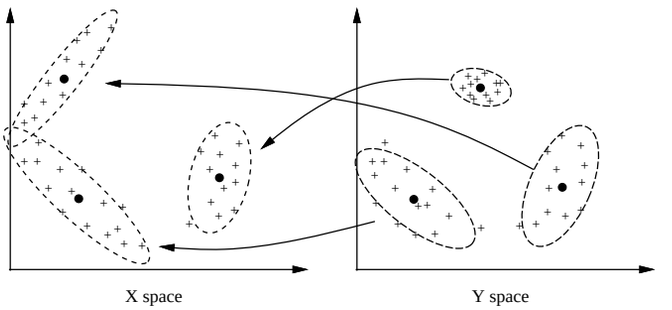


Fig. 2. Y-clusters in space $\mathcal{Y}$ and the corresponding x-clusters in space $\mathcal{X}$. The first and the second order statistics are updated for each cluster.

Two types of clusters are incrementally updated at each node of the IHDR tree — y-clusters and x-clusters, as shown in Fig. 2. The y-clusters are clusters in the output space $\mathcal{Y}$ and x-clusters are those in the input space $\mathcal{X}$. There are a maximum of q (e.g., $q = 10$) clusters of each type at each node. The q y-clusters determine the virtual class label of each arriving sample (x, y) based on its y part. Each x-cluster approximates the sample population in $\mathcal{X}$ space for the samples that belong to it. It may spawn a child node from the current node if a finer approximation is required. At each node, y in (x, y) finds the nearest y-cluster in Euclidean distance and updates (pulling) the center of the y-cluster. This y-cluster indicates which corresponding x-cluster the input (x, y) belongs to. Then, the x part of (x, y) is used to update the statistics of the x-cluster (the mean vector and the covariance matrix). These statistics of every x-cluster are used to estimate the probability for the current sample (x, y) to belong to the x-cluster, whose probability distribution is modeled as a multidimensional Gaussian at this level. In other words, each node models a region of the input space $\mathcal{X}$ using q Gaussians. Each Gaussian will be modeled by more small Gaussians in the next tree level if the current node is not a leaf node. Each x-cluster in the leaf node is linked with the corresponding y-cluster.

Moreover, the center of these x-clusters provide essential information for discriminating subspace, since these x-clusters are formed according to virtual labels in $\mathcal{Y}$ space. We define a discriminating subspace as the linear space that passes through the centers of these x-clusters. A total of q centers of the q x-clusters give $q - 1$ discriminating features which span $(q - 1)$-dimensional discriminating space. A probability-based distance called size-dependent negative-log-likelihood (SNLL) [6] is computed from x to each of the q x-clusters to determine which x-cluster should be further searched. If the probability is high enough, the sample (x, y) should further search the corresponding child (maybe more than one but with an upper bound k) recursively, until the corresponding terminal nodes are found.

The algorithm incrementally builds an IHDR tree from a sequence of training samples. The deeper a node is in the tree, the smaller the variances of its x-clusters are. When the number of samples in a node is too small to give a good estimate of the statistics of q x-clusters, this node is a leaf node. If y is not given in the input, the x part is used to search the tree, until the nearest x-cluster in a leaf node is find. The center of the corresponding y-cluster is the produced estimated y output[2].

Why do we use a tree? Two major reasons: (1) automatically deriving features (instead of human defining features) and (2) fast search. The number of x-clusters in the tree is a very large number. The y-clusters allow the search to disregard input components that are not related to the output. For example, if some sensors are not related to the action of the humanoid, under a context, these sensors are disregarded automatically by the IHDR mapping, since each node partition the samples in $q - 1$ dimensional discriminating subspace, instead of in the original input space. This subspace is the automatically derived feature space for the samples in the subtree. Further, the tree allows a large portion of far-away clusters to disregarded from consideration. This results in the well-known logarithmic time complex for tree retrieval: $O(\log m)$ where m is the number of leaf nodes in the tree.

The algorithm incrementally builds a tree from a sequence of training samples. The deeper a node is in the tree, the smaller the variances of its x-clusters are. When the number of samples in a node is too small to give a good estimate of the statistics of q x-clusters, this node is a leaf node. The following is the outline of the incremental algorithm for tree building (also tree retrieval when y is not given).

Procedure 1 `Update-node`*: Given a node N and (x, y) where y is either given or not given, update the node N using (x, y) recursively. Output: top matched terminal nodes. The parameters include: k which specifies the*

[2] In each leaf node, we allow more than q clusters to fully use the samples available at each leaf node.

upper bound in the width of parallel tree search; δ_x the sensitivity of the IHDR in $\mathcal{X}$ space as a threshold to further explore a branch; and c representing if a node is on the central search path. Each returned node has a flag c. If $c = 1$, the node is a central cluster and $c = 0$ otherwise.

1. *Find the top matched x-cluster in the following way. If $c = 0$ skip to step (2). If y is given, do (a) and (b); otherwise do (b).*
   (a) *Update the mean of the y-cluster nearest y in Euclidean distance by using amnesic averages. Update the mean and the covariance matrix of the x-cluster corresponding to the y-cluster by using amnesic average.*
   (b) *Find the x-cluster nearest x according to the probability-based distances. The central x-cluster is this x-cluster. Update the central x-cluster if it has not been updated in (a). Mark this central x-cluster as active.*
2. *For all the x-clusters of the node N, compute the probability-based distances for x to belong to each x-cluster.*
3. *Rank the distances in increasing order.*
4. *In addition to the central x-cluster, choose peripheral x-clusters according to increasing distances until the distance is larger than δ_x or a total of k x-clusters have been chosen.*
5. *Return the chosen x-clusters as active clusters.*

From the above procedure, we can observe the following points. (a) When y is given, the corresponding x-cluster is updated, although this x-cluster is not necessarily the one on the central path from which the tree is explored. Thus, we may update two x-clusters, one corresponding to the given y, the other being the one used for tree exploration. The update for the former is an attempt to pull it to the right location. The update for the latter is an attempt to record the fact that the central x-cluster has hit this x-cluster once. (b) No matter y is given or not, the x-cluster along the central path is always updated. (c) Only the x-clusters along the central path are updated, other peripheral x-clusters are not. We would like to avoid, as much as possible, storing the same sample in different brother nodes.

Procedure 2 `Update-tree`*: Given the root of the tree and sample (x, y), update the tree using (x, y). If y is not given, estimate y and the corresponding confidence. The parameters include: k which specifies the upper bound in the width of parallel tree search.*

1. *From the root of the tree, update the node by calling* `Update-node` *using (x,y).*
2. *For every active cluster received, check if it points to a child node. If it does, mark it inactive and explore the child node by calling* `Update-node`*. At most q^2 active x-clusters can be returned this way if each node has at most q children.*
3. *The new central x-cluster is marked as active.*
4. *Mark additional active x-clusters according to the smallest probability-based distance d, up to k total if there are that many x-clusters with $d \leq \delta_x$.*
5. *Do the above steps 2 through 4 recursively until all the resulting active x-clusters are all terminal.*
6. *Each leaf node keeps samples (or sample means) $(\hat{x}_i, \hat{y}_i)$ that belong to it. If y is not given, the output is $\hat{y}_i$ if $\hat{x}_i$ is the nearest neighbor among these samples. If y is given, do the following: If $||y - \hat{y}_i||$ is smaller than an error tolerance, (x, y) updates $(\hat{x}_i, \hat{y}_i)$ only. Otherwise, (x, y) is a new sample to keep in the leaf.*
7. *If the current situation satisfies the spawn rule, i.e. the number of samples exceeds the number required for estimating statistics in new child, the top-matched x-cluster in the leaf node along the central path spawns a child which has q new x-clusters. All the internal nodes are fixed in that their clusters do not further update using future samples so that their children do not get temporarily inconsistent assignment of samples.*

The above incrementally constructed tree gives a coarse-to-fine probability model. If we use Gaussian distribution to model each x-cluster, this is a *hierarchical version* of the well-known mixture-of-Gaussian distribution models: the deeper the tree is, the more Gaussians are used and the finer are these Gaussians. At shallow levels, the sample distribution is approximated by a mixture of large Gaussians (with large variances). At deep levels, the sample distribution is approximated by a mixture of many small Gaussians (with small variances). The multiple search paths guided by probability allow a sample x that falls in-between two or more Gaussians at each shallow level to explore the tree branches that contain its neighboring x-clusters. Those x-clusters to which the sample (x, y) has little chance to belong are excluded for further exploration. This results in the well-known logarithmic time complex for tree retrieval: $O(\log m)$ where m is the number of leaf nodes in the tree, assuming that the number of samples in each leaf node is bounded above by a constant.

3.2 Amnesic average

In incremental learning, the initial centers of each state clusters are largely determined by early input data. When more data are available, these centers move to more appropriate locations. If these new locations of the cluster centers are used to judge the boundary of each cluster, the initial input data were typically incorrectly classified. In other words, the center of each cluster contains some earlier data that do not belong to this cluster. To reduce the effect of these earlier data, the amnesic average can be used to compute the center of each cluster. The amnesic average can also track dynamic change of the input environment better than a conventional average.

The average of n input data $x_1, x_2, ..., x_n$ can be recursively computed from the current input data x_n and the previous average $\bar{x}^{(n-1)}$ by equation (3):

$$\bar{x}^{(n)} = \frac{(n-1)\bar{x}^{(n-1)} + x_n}{n} = \frac{n-1}{n}\bar{x}^{(n-1)} + \frac{1}{n}x_n. \tag{3}$$

In other words, the previous average $\bar{x}^{(n)}$ gets a weight $n/(n+1)$ and the new input x_{n+1} gets a weight $1/(n+1)$. These two weights sum to one. The recursive equation Eq. (3) gives an equally weighted average. In amnesic average, the new input gets more weight than old inputs as given in the following expression: $\bar{x}^{(n+1)} = \frac{n-l}{n+1}\bar{x}^{(n)} + \frac{1+l}{n+1}x_{n+1}$, where l is a parameter.

The amnesic average can also be applied to the recursive computation of a covariance matrix Γ_x from incrementally arriving samples: $x_1, x_2, ..., x_n, ...$ where x_i is a column vector for $i = 1, 2, ...$ The unbiased estimate of the covariance matrix from these n samples $x_1, x_2, ..., x_n$ is given in a batch form as

$$\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})(x_i - \bar{x})^T \tag{4}$$

with $n > 1$, where $\bar{x}$ is the mean vector of the n samples. Using the amnesic average, $\bar{x}^{(n+1)}$, up to the $(n+1)$-th sample, we can compute the amnesic covariance matrix up to the $(n+1)$-th sample as

$$\Gamma_x^{(n+1)} = \frac{n-1-l}{n}\Gamma_x^{(n)} + \frac{1+l}{n}(x_{n+1} - \bar{x}^{(n+1)})(x_{n+1} - \bar{x}^{(n+1)})^T \tag{5}$$

for $n > l + 1$. When $n \leq l + 1$, we may use the batch version as in expression (4). Even with a single sample x_1, the corresponding covariance matrix should not be estimated as a zero vector, since x_1 is never exact if it is measured from a physical event. For example, the initial variance matrix $\Gamma_x^{(1)}$ can be estimated as $\sigma^2 I$, where σ^2 is the expected digitization noise in each component and I is the identity matrix of the appropriate dimensionality.

3.3 Discriminating subspace

Due to a very high input dimensionality (typically at least a few thousands), for computational efficiency, we should not represent data in the original input space $\mathcal{X}$. Further, for better generalization characteristics, we should use discriminating subspaces in which input components that are irrelevant to output are disregarded.

We first consider x-clusters. Each x-cluster is represented by its mean as its center and the covariance matrix as its size. However, since the dimensionality of the space $\mathcal{X}$ is typically very high, it is not practical to directly keep the covariance matrix. If the dimensionality of $\mathcal{X}$ is 3000, for example, each covariance matrix requires $3000 \times 3000 = 9,000,000$ numbers! We adopt a more efficient method that uses subspace representation.

As explained in Section 3.1, each internal node keeps up to q x-clusters. The centers of these q x-clusters are denoted by

$$C = \{c_1, c_2, ..., c_q \mid c_i \in \mathcal{X}, i = 1, 2, ..., q\}. \tag{6}$$

The locations of these q centers tell us the subspace $\mathcal{D}$ in which these q centers lie. $\mathcal{D}$ is a discriminating space since the clusters are formed based on the clusters in output space $\mathcal{Y}$.

The discriminating subspace $\mathcal{D}$ can be computed as follow. Suppose that the number of samples in cluster i is n_i and thus the grand total of samples is $n = \sum_{i=1}^{q} n_i$. Let $\bar{C}$ be the mean of all the q x-cluster centers. $\bar{C} = \frac{1}{n}\sum_{i=1}^{q} n_i c_i$ The set of scatter vectors from their center then can be defined as $s_i = c_i - \bar{C}$, $i = 1, 2, ..., q$. These q scatter vectors are not linearly independent because their sum is equal to a zero vector. Let S be the set that contains these scatter vectors: $S = \{s_i \mid i = 1, 2, ..., q\}$. The subspace spanned by S, denoted by span(S), consists of all the possible linear combinations from the vectors in S, as shown in Fig. 3.3.

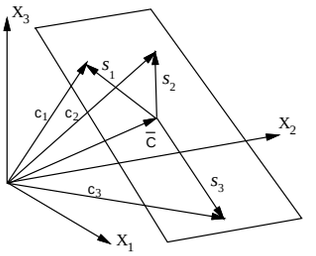


Fig. 3. The linear manifold represented by $\bar{C} + \text{span}(S)$, the spanned space from scatter vectors translated by the center vector $\bar{C}$.

The orthonormal basis $a_1, a_2, ..., a_{q-1}$ of the subspace span(S) can be constructed from the radial vectors $s_1, s_2, ..., s_q$ using the *Gram-Schmidt Orthogonalization* (GSO) procedure. The number of basis vectors that can be computed by the GSO procedure is the number of linearly independent radial vectors in S.

Given a vector $x \in \mathcal{X}$, we can compute its scatter part $s = x - \bar{C}$. Then compute the projection of x onto the linear manifold by $f = M^T s$, where $M = [a_1, a_2, \ldots, a_{q-1}]$. We call the vector f the discriminating features of x in the linear manifold S. The mean and the covariance of the clusters then are computed on the discriminating subspace.

3.4 The probability-based metric

The characteristics of different metrics Let us consider the negative-log-likelihood (NLL) defined from Gaussian density of dimensionality $q-1$:

$$G(x, c_i) = \frac{1}{2}(x - c_i)^T \Gamma_i^{-1}(x - c_i) + \frac{q-1}{2}\ln(2\pi) + \frac{1}{2}\ln(|\Gamma_i|). \tag{7}$$

We call it Gaussian NLL for x to belong to the cluster i. c_i and Γ_i are the cluster sample mean and sample covariance matrix, respectively, computed using the amnesic average in Section 3.2. Similarly, we define Mahalanobis NLL and Euclidean NLL as:

$$M(x, c_i) = \frac{1}{2}(x - c_i)^T \Gamma^{-1}(x - c_i) + \frac{q-1}{2}\ln(2\pi) + \frac{1}{2}\ln(|\Gamma|), \tag{8}$$

$$E(x, c_i) = \frac{1}{2}(x - c_i)^T \rho^2 I^{-1}(x - c_i) + \frac{q-1}{2}\ln(2\pi) + \frac{1}{2}\ln(|\rho^2 I|). \tag{9}$$

where Γ is the within-class scatter matrix of each node — the average of covariance matrices of q clusters:

$$\Gamma = \frac{1}{q-1}\sum_{i=1}^{q-1}\Gamma_i \tag{10}$$

computed using the same technique of the amnesic average.

Suppose that the input space is $\mathcal{X}$ and the discriminating subspace for an internal node is $\mathcal{D}$. The Euclidean NLL treats all the dimensions in the discriminating subspace $\mathcal{D}$ the same way, although some dimensionalities can be more important than others. It has only one parameter ρ to estimate. Thus it is the least demanding among the three NLL in the richness of observation required. When very few samples are available for all the clusters, the Euclidean likelihood is the suited likelihood.

The Mahalanobis NLL uses within-class scatter matrix Γ computed from all the samples in all the q x-clusters. Using Mahalanobis NLL as the weight for subspace $\mathcal{D}$ is equivalent to using Euclidean NLL in the basis computed from Fisher's LDA procedure [4] [11]. It decorrelates all dimensions and weights each dimension using a different weight. The number of parameters in Γ is $q(q-1)/2$, and thus, the Mahalanobis NLL requires more samples than the Euclidean NLL.

The Mahalanobis NLL does not treat different x-lusters differently because it uses a single within-class scatter matrix Γ for all the q x-clusters in each internal node. For Gaussian NLL, $L(x, c_i)$ in Eq.(7) uses the covariance matrix Γ_i of x-cluster i. In other words, Gaussian NLL not only decorrelates the correlations but also applied a different weight at different location along each rotated basis. However, it requires that each x-cluster has enough

samples to estimate the $(q-1) \times (q-1)$ covariance matrix. It thus is the most demanding on the number of observations. Note that the decision boundary of the Euclidean NLL and the Mahalanobis NLL is linear but that by the Gaussian NLL is quadratic.

The transition among different metrics We would like to use the Euclidean NLL when the number of samples in the node is small. Gradually, as the number of samples increases, the within-class scatter matrix of q x-clusters are better estimated. Then, we would like to use the Mahalanobis NLL. When a cluster has very rich observations, we would like to use the full Gaussian NLL for it. We would like to make an automatic transition when the number of samples increases. We define the number of samples n_i as the measurement of maturity for each cluster i. $n = \sum_{i=1}^{q} n_i$ is the total number of samples in a node.

For the three types of NLLs, we have three matrices, $\rho^2 I$, Γ, and Γ_i. Since the reliability of estimates are well indicated by the number of samples, we consider the number of scales received to estimate each parameter, called the number of scales per parameter (NSPP), in the matrices. The NSPP for $\rho^2 I$ is $(n-1)(q-1)$, since the first sample does not give any estimate of the variance and each independent vector contains $q-1$ scales. For the Mahalanobis NLL, there are $(q-1)q/2$ parameters to be estimated in the (symmetric) matrix Γ. The number of independent vectors received is $n-q$ because each of the q x-cluster requires a vector to form its mean vector. Thus, there are $(n-q)(q-1)$ independent scalars. The NSPP for the matrix Γ is $\frac{(n-q)(q-1)}{(q-1)q/2} = \frac{2(n-q)}{q}$. To avoid the value to be negative when $n < q$, we take NSPP for Γ to be $\max\left\{\frac{2(n-q)}{q}, 0\right\}$. Similarly, the NSPP for Γ_i for the Gaussian NLL is $\frac{1}{q}\sum_{i=1}^{q} \frac{2(n_i-1)}{q} = \frac{2(n-q)}{q^2}$. Table 2 summarizes the result of the NSPP values of the above derivation.

Table 2. Characteristics of three types of scatter matrices

Type	Euclidean $\rho^2 I$	Mahalanobis Γ	Gaussian Γ_i
NSPP	$(n-1)(q-1)$	$\frac{2(n-q)}{q}$	$\frac{2(n-q)}{q^2}$

A bounded NSPP is defined to limit the growth of NSPP so that other matrices that contain more scalars can take over when there are a sufficient number of samples for them. Thus, the bounded NSPP for $\rho^2 I$ is $b_e = \min\{(n-1)(q-1), n_s\}$, where n_s denotes the soft switch point for the next more complete matrix to take over. To estimate n_s, we consider a serious of random variables drawn independently from a distribution with a variance σ^2, the expected sample mean of n random variables has a expected variance $\sigma^2/(n-1)$. We can choose a switch confidence value α for $1/(n-1)$. When $1/(n-1) = \alpha$, we consider that the estimate can take about a 50% weight. Thus, $n = 1/\alpha + 1$. As an example, let $\alpha = 0.05$ meaning that we trust the estimate with 50% weight when the expected variance of the estimate is reduced to about 5% of that of a single random variable. This is like a confidence value in hypothesis testing except that we do not need an absolute confidence, relative one suffices. We get then $n = 21$, which leads to $n_s = 21$.

The same principle applies to Mahalanobis NLL and its bounded NSPP for Γ is $b_m = \min\left\{\max\left\{\frac{2(n-q)}{q}, 0\right\}, n_s\right\}$. It is worth noting that the NSPP for the Gaussian NLL does not need to be bounded, since among our models it is the best estimate with increasing number of samples beyond. Thus the bounded NSPP for Gaussian NLL is $b_g = \frac{2(n-q)}{q^2}$.

How do we realize automatic transition? We define a *size-dependent scatter matrix* (SDSM) W_i as a weighted sum of three matrices:

$$W_i = w_e \rho^2 I + w_m \Gamma + w_g \Gamma_i \tag{11}$$

where $w_e = b_e/b$, $w_m = b_m/b$, $w_g = b_g/b$ and b is a normalization factor so that these three weights sum to 1: $b = b_e + b_m + b_g$. Using this size-dependent scatter matrix W_i, the *size-dependent negative log likelihood* (SDNLL) for x to belong to the x-cluster with center c_i is defined as

$$L(x, c_i) = \frac{1}{2}(x - c_i)^T W_i^{-1}(x - c_i) + \frac{q-1}{2}\ln(2\pi) + \frac{1}{2}\ln(|W_i|). \tag{12}$$

With b_e, b_m, and b_g change automatically, $(L(x, c_i)$ transit smoothly through the three NLLs. It is worth noting the relation between LDA and SDNLL metric. LDA in space $\mathcal{D}$ with original basis η gives a basis ϵ for a subspace

$\mathcal{D}' \subseteq \mathcal{D}$. This basis ϵ is a properly oriented and scaled version for $\mathcal{D}$ so that the within-cluster scatter in $\mathcal{D}'$ is a unit matrix [4] (Sections 2.3 and 10.2). In other words, all the basis vectors in ϵ for D' are already weighted according to the within-cluster scatter matrix Γ of $\mathcal{D}$. If $\mathcal{D}'$ has the same dimensionality as $\mathcal{D}$, the Euclidean distance in $\mathcal{D}'$ on ϵ is equivalent to the Mahalanobis distance in $\mathcal{D}$ on η, up to a global scale factor. However, if the covariance matrices are very different across different x-clusters and each of them has enough samples to allow a good estimate of individual covariance matrix, LDA in space $\mathcal{D}$ is not as good as Gaussian likelihood because covariance matrices of all X-clusters are treated as the same in LDA while Gaussian likelihood takes into account of such differences. The SDNLL in (12) allows automatic and smooth transition between three different types of likelihood, Euclidean, Mahalanobis and Gaussian, according to the predicted effectiveness of each likelihood.

3.5 Computational considerations

The matrix weighted squared distance from a vector $x \in \mathcal{X}$ to each X-cluster with center c_i is defined by

$$d^2(x, c_i) = (x - c_i)^T W_i^{-1} (x - c_i) \tag{13}$$

which is the first term of Eq.(12).

This distance is computed only in $(q-1)$-dimensional space using the basis M. The SDSM W_i for each x-cluster in then only a $(q-1) \times (q-1)$ square symmetric matrix, of which only $q(q-1)/2$ parameters need to be estimated. When $q = 6$, for example, this number is 15.

Given a column vector v represented in the discriminating subspace with an orthonormal basis whose vectors are the columns of matrix M, the representation of v in the original space $\mathcal{X}$ is $x = Mv$.

To compute the matrix weighted squared distance in Eq.(13), we use a numerically efficient method, Cholesky factorization [5] (Sec. 4.2). The Cholesky decomposition algorithm computes a lower triangular matrix L from W so that W is represented by $W = LL^T$. With the lower triangular matrix L, we first compute the difference vector from the input vector x and each x-cluster center c_i: $v = x - c_i$. The matrix weighted squared distance is given by

$$d^2(x, c_i) = v^T W_i^{-1} v = v^T (LL^T)^{-1} v = (L^{-1}v)^T (L^{-1}v). \tag{14}$$

We solve for y in the linear equation $Ly = v$ and then $y = L^{-1}v$ and $d^2(x, c_i) = (L^{-1}v)^T (L^{-1}v) = \|y\|^2$. Since L is a lower triangular matrix, the solution for y in $Ly = v$ is trivial since we simply use the backsubtitution method as described in [9] (page 42).

4 Experiments

4.1 SAIL robot

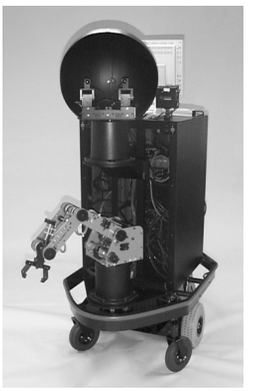


Fig. 4. The SAIL robot built at the Pattern Recognition and Image Processing Laboratory at Michigan State University.

A human-size robot called SAIL was assembled at MSU, as shown in Fig. 4. SAIL robot's "neck" can turn. Each of its two "eyes" is controlled by a fast pan-tilt head. Its torso has 4 pressure sensors to sense push actions and force. It has 28 touch sensors on its arm, neck, head, and bumper to allow human to teach how to act by direct

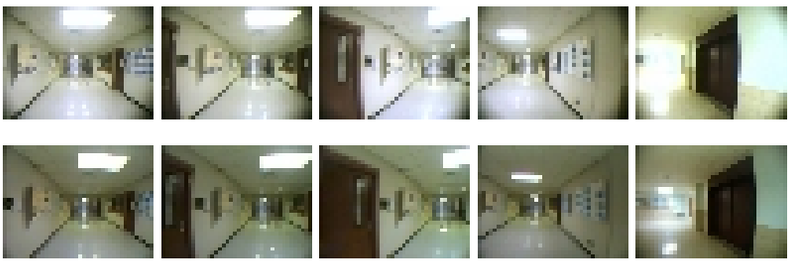


Fig. 5. A subset of images used in autonomous navigation. The number right below the image shows the needed heading direction (in degrees) associated with that image. The first row shows the images from the right camera while the second row shows those from the left camera.

touch. Its drive-base is adapted from a wheelchair and thus the SAIL robot can operate both indoor and outdoor. Its main computer is a high-end dual-processor dual-bus PC workstation with 512MB RAM and an internal 27GB three-drive disk array for real-time sensory information processing, real-time memory recall and update as well as real-time effector controls. This platform is being used to test the architecture and the developmental algorithm outlined here.

4.2 Autonomous navigation

At each time instance, the vision-based navigation system accepts a pair of stereo images, updates its states which contains past sensory inputs and actions, and then outputs the control signal C to update the heading direction of the vehicle. In the current implementation, the state transition function f_t in Eq. 1 is programmed so that the current state includes a vector that contains the sensory input and past heading direction of last T cycles. The key issue then is to approximate the action generation function g_t in Eq. 2. This is a very challenging approximation task since the function to be approximated is for a very high dimensional input space and the real application requires the navigator to perform in real time.

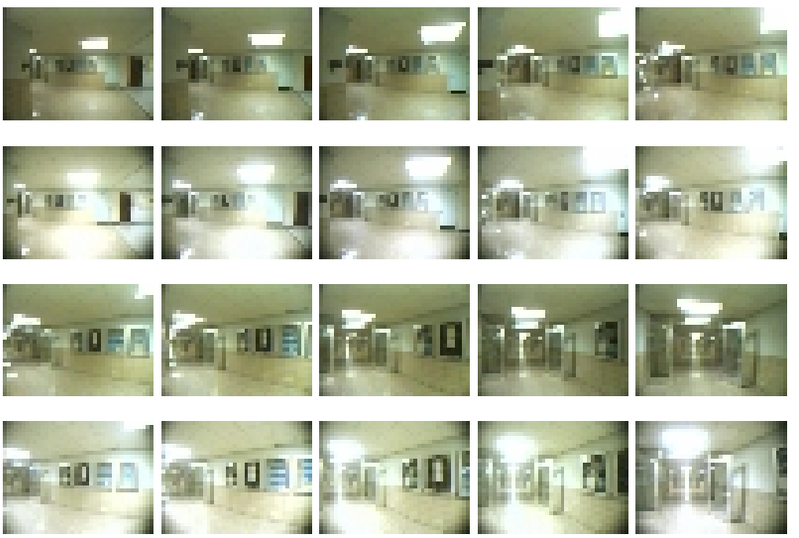


Fig. 6. A subset of images which were inputs to guide the robot turn. Row one and three show the images from left camera. The second and fourth rows show the images taken from right camera.

We applied our IHDR algorithm to this challenging problem. Some of the example input images are shown in Fig 5. We first applied the IHDR algorithm to simulate the actual vision-based navigation problem. Totally 2106 color stereo images with the corresponding heading directions were used for training. The resolution of each image

is 30 by 40. The input dimensionality of the IHDR algorithm is $30 \times 40 \times 3 \times 2 = 7200$, where 3 is the length of history in state. We used the other 2313 stereo images to test the performance of the trained system. Fig 7 shows the error rate versus the number of training epochs, where each epoch corresponds to the feeding of the entire training sensory sequence once. As shown even after the first epoch, the performance of the IHDR tree is already reasonably good. With the increase of the number of epochs, we observed the improvements of the error rate. The error rate for the test set is 9.4% after 16 epochs.

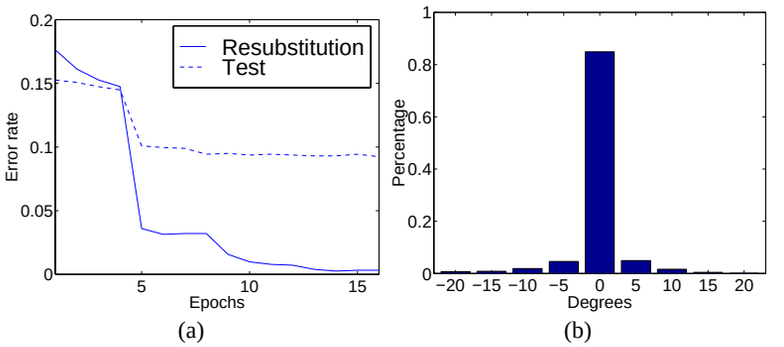


Fig. 7. The performance for autonomous navigation. (a) The plot for the error rates vs. epochs. The solid line represents the error rates for resubstitution test. The dash line represents the error rates for the testing set. (b) The error histogram of the testing set after 16 epochs.

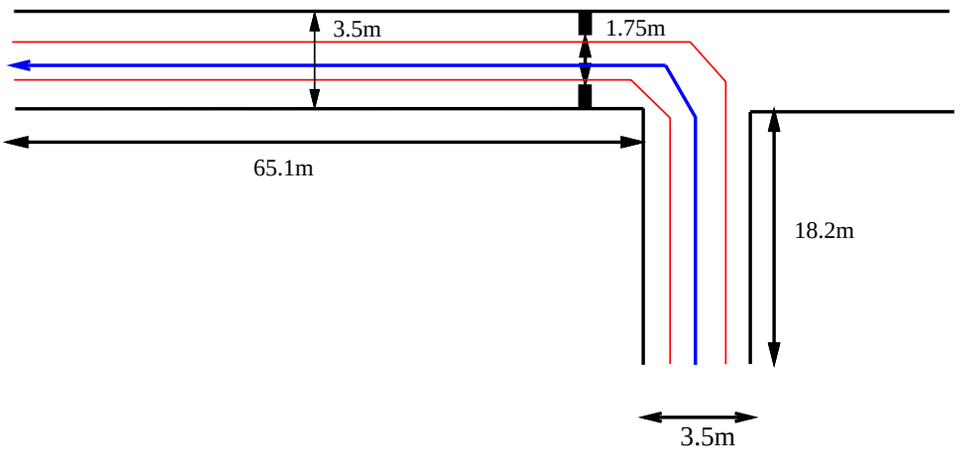


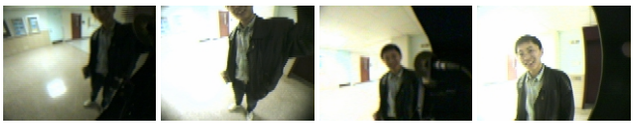
Fig. 8. The navigation path. The blue line is the desired navigation path and the tan lines are the navigation bounds. During the test, the SAIL robot navigated within the boundaries.

The IHDR algorithm then was applied on the real training/testing experiment. The SAIL robot was trained interactively by a human trainer using the force sensors equipped on the body of the robot. The forces sensed by the sensors are translated to the robot heading direction and speed. The training is on-line in real time. The trainer pushed just two force sensors to guide the robot to navigate through the corridor of about 3.5 meter wide in the Engineering building of Michigan State University. The navigation site includes a turn, two straight sections which include a corridor door. Then trips were found sufficient to reach a reliable behavior. During the training, the IHDR algorithm receives both the color stereo images as input and heading direction as output. It rejects samples (not used for learning) if the input images are too similar to samples already learned. We tested the performance by letting the robot go through the corridor 10 times. All the tests were successful. The closest distance between the SAIL robot and the wall is about 40 cm among the 10 tests. The test showed the SAIL robot can successfully navigate in the indoor environment as shown in Fig 8 after the interactive training. We plan to extend the area of navigation in the future work.

4.3 Vision attention using motion

The SAIL robot has embedded some innate behaviors, behaviors either programmed-in or learned off-line. For this behavior we used off-line supervised learning. One such behavior is vision attention driven by the motion. Its goal is to move the eyes so that moving object of interest is moved to the "fovea", the center of the image. With this mechanism, perception and measurement is performed mainly for the "fovea", while the periphery of image frame is used only to find the object of interest.

To implement this mechanism, we first collect a sequence of images with moving objects. The input to the IHDR mapping is an image in which each pixel is the absolute difference of pixels in consecutive images. For training, we acquired the center of moving object and the amount of motion that the pan-tilt unit must perform to bring the position to the image center. We used the IHDR algorithm to build the mapping between the motion (image difference) and pan-tilt control signals. For each training sample point i, $i = 1, \ldots, n$, we have image difference as input and pan and tilt angle increments as output. Some example images are shown in Fig 9.



(a) (b) (c) (d)

Fig. 9. An example of motion tracking, or motion guided visual attention. (a) and (b) are the left and right images when an object moves in. (c) and (d) are the images after desired pan and tilt of the eyes.

4.4 Test for the developmental algorithm SAIL-2

We ran the developmental algorithm on the SAIL robot. Since tracking objects and reaching objects are sensori-motor behaviors first developed in early infants, we trained our SAIL robot for two tasks. In the first task, called finding-ball task, we trained the SAIL robot to find a nearby ball and then turn eyes to it so that the ball is located on the center of sensed image. In the second task, called pre-reaching task, we trained the SAIL robot to reach for the object once it has been located and the eyes fixate on it.

Existing studies on visual attention selection are typically based on low-level saliency measures, such as edges and texture. [1] In Birnbaum's work [2], the visual attention is based on the need to explore geometrical structure in the scene. In our case, the visual attention selection is a result of past learning experience. Thus, we do not need to define any task-specific saliency features. It is the SAIL robot that automatically derives the most discriminating features for the tasks being learned. At the time of learning, the ball was presented in the region of interest (ROI) inside the stereo images. The human trainer interactively pulls the robot's eyes toward the ball (through the touch sensors for the pan-tilt heads) so that the ball is located on the center of the region of ROI (fixating the eyes on the ball)[3]. The inputs to the developmental algorithm are the continuous sequence of stereo images and the sequence of the pan-tilt head control signal. Three actions are defined for the pan-tilt head in pan direction: 0 (stop), 1 (move to the left), or -1 (move to the right). The size of ROI we chose for this experiment is defined as 120×320. In the mind of trainer, the ROI is divided into five regions so that each region is of size 120×64. The goal of the finding-ball task, is to turn the pan-tilt head so that the ball is at the center region. Fig. 10 shows some example images for the tracking task.

The transitions during the training session are described below:

1. The task input is initiated by pushing a pressure sensor of the robot (or typing in a letter via keyboard) before imposing action to pan the camera. The action of the pan is zero at this time since no action is imposed.
2. The action of the pan is imposed at time t. The initialization flag is on at the same time. The main program issues a control signal to pan the camera.
3. The PTU starts to pan. The pan position as well as the image changes. Note that at time $t + 1$ the previous pan action is zero.

[3] This is not typically done with human infants, since we cannot pull infant's eye. However, this makes robot learning much faster than what a human baby can. This is in fact an advantage of robot over humans in that the robot can be built to fascinate training.

Fig. 10. A subset of images used in the tracking problem. The number right below the image shows the PTU position associated with that image. From left to right, one image sequence of ball-tracking is shown.

4. When the ball is at the fixation of the view at time T, we stops the imposition of action of pan, and the initialization flag is off.
5. At time $T+1$, the PTU stopped moving and the image does not change any more. It is worth noting that the pan action is all zero after time $T-1$.

Similarly, the testing session can be explained as follows:

1. The human tester pushes a pressure sensor to simulate a task command and the initialization flag is on at time t.
2. The action of the pan is automatically generated by the IHDR tree. A non-zero action is expected according to the training process.
3. The PTU starts to move automatically and the image changes.
4. When the ball is at the fixation of the view at time T, the query result of the IHDR is a zero action. This zero action (stop) is sent to the PTU and the initialization flag is off.
5. At time $T+1$, the PTU stops moving and the image does not change any more.

Why is the state important here? If the state, which keeps the previous pan action is not used, as input to the IHDR tree, the image and the pan position will be very similar at the point where the action should stop. This will make the PTU stop and go in a random fashion at this boundary point. The context (direction from which the arm is from) resolves the ambiguity.

The online training and testing were performed successfully and the robot can perform finding-ball task and pre-reaching task successfully, after interactive training, although the developmental algorithm was not written particularly for these two tasks.

To quantitatively evaluate the performance of the online learning and performance, we recorded recorded the sensory data and studied the performance off-line. Since the developmental algorithm runs indefinitely, does its memory grow without bound? Fig. 11(a) shows the memory usage of the program. In the first stage, the tree grows since the samples are accumulated in the shallow nodes. When the performance of the updated tree is consistent to the desired action, the tree does not grow and thus the memory curve becomes flat. The tree will grow only when the imposed action is significantly different from what the tree comes up with. Otherwise, the new inputs only participate in the average of the corresponding cluster, simulating sensorimotor refinement of repeated practice, but there is not need for additional memory. This is a kind of forgetting — without remembering every detail of repeated practice. How fast the developmental algorithm learn? Fig. 11(b) shows the accuracy of the PTU action in terms of the percentage of field of view. After the 3-rd epoch (repeated training), the systems can reliably move the eye so that the ball is at the center of ROI. Does the developmental algorithm slow down when it has learned more? Fig. 11(c) gives the plot of the average CPU time for each sensory-action update. The average CPU time for update is within 100 millisecond, meaning that the system runs at about 10 Hertz, 10 refresh of sensory input and 10 updated actions per second. Since the IHDR tree is dynamically updated, all the updating and forgetting are performed in each cycle. This relatively stable time profile is due to the use of the tree structure. The depth of the tree is stable.

4.5 Speech recognition

Speech recognition has achieved significant progress in the past ten years. It still faces, however, many difficulties, one of which is the training mode. Before training any acoustic models, such as HMM, the human trainer

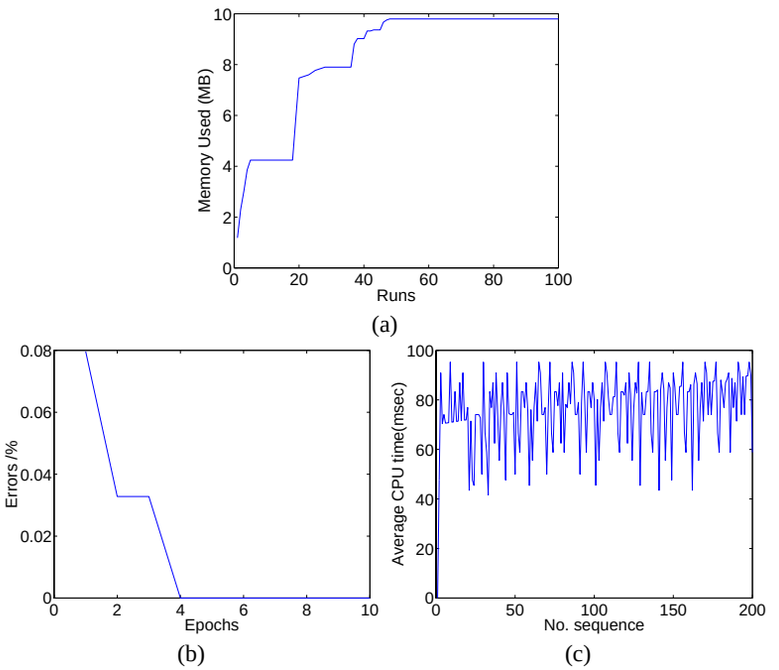


Fig. 11. (a) The memory usage for the off-line simulation of finding-ball task. (b) The accuracy of finding-ball task versus the number of training cases. (c) The CPU time for each update.

must do data transcription, a procedure of translating a speech waveform into a string of symbols representing the acoustic unit, like phonemes. In other words, the training data must be organized manually according to the acoustic characteristics. This procedure requires the expertise of linguistics and is very labor-intensive. Moreover, inherently, this training can only be done off-line, making on-line experience learning not possible. We used our SAIL-2 developmental algorithm to realize online learning based on supervised learning.

Experiment procedure Once SAIL starts running, the microphone keeps collecting the environment sound. A SoundBlast card digitizes the signals from microphone at 10 kHz. For every segment of 256 speech data points, which is roughly 25 ms of data, cepstrum analysis gives a 16-dimensional Mel-Cepstrum feature vector. There are 56-point overlap between two consecutive segments.

When teaching SAIL, the trainer says the word first and then imposes actions through the touch sensors to generate control signal vectors.

The control signal sequence together with the 16 cepstrum feature vector stream goes into the IHDR mapping engine. As speech patterns are temporal patterns, a piece of 20 ms segment does not include much pattern information. In other words, we need longer working memory or state. In effect, the designed speech state covers 32 time steps which amounts to 640 ms, while the control signal state covers 16 time steps which is of 320 ms long.

After training, the trainer can test it by saying the word again and see whether SAIL repeats the action. Each sound corresponds to a verbal command of a different action.

Experiment results To evaluate the performance more conveniently, we first did the experiment in simulation.

We record the voice of 141 persons with a variety of nationalities, including American, Chinese, French, India, Malaysian and Spanish, and ages, from 18 to 50. Each person made 5 utterances for each of the 5 vowels, a, e, i, o, u. There is silence of 0.5s between two consecutive utterances. Thus, we got a one-hour speech dataset of isolated vowel utterances. There are totally 3525 utterances. The control signal vectors sequence is generated so that there are different control signals after different vowels.

We used, in training session, 4 out of 5 utterances of each of the 5 vowels of each person and the remaining utterance of each vowel was used for test. The data were fed to the developmental algorithm in the way described above.

The performance is evaluated as followings. Within 10 time steps (200ms) before and after the position the system is supposed to react, if there is one wrong reaction or if the system keeps quiet by doing nothing, we mark it as doing wrong. If the system reacts correctly once or more than once within the time window we are interested, we mark it as doing correct. The whole experiment was done with a 5-fold cross-validation. The average error rate was 0.99%.

We also ran the experiment on SAIL robot. Its performance varied, very much depending on the trainer. In the simulation mode, the time of the imposed actions can be given pretty consistently for different utterances. In real test, however, it is not easy for a trainer to impose the action precisely at the same time instant after each utterance. If he/she is not consistent, SAIL will be confused and in many cases keeps doing nothing. We are currently working on two ways to resolve this issue, one is attention selection, the other is reinforcement learning.

A video demonstration is attached to this paper submission.

5 Conclusions

We introduced here a new kind of robots: robots that can develop their mental skills automatically through real-time interactions with the environment. The representation of the system is automatically generated through interaction between the developmental mechanism and the experience.

A technical challenge for the developmental algorithm is that the mapping engine must be scalable — keeping real-time speed and a stable performance for a very large number of high dimensional sensory and effector data. With our IHDR mapping engine, the developmental algorithm is able to operate in real time. The SAIL-2 developmental algorithm has successfully run on the SAIL robot for real-time interactive training and real-time testing for two sensorimotor tasks: finding ball and reaching the centered ball, two early tasks that infants learn to perform. These two tasks do not seem very difficult judged by a human layman, but they mark a significant technical advance since the program has little to do the task. First, the same developmental program can be continuously used to train other tasks. This marks a significant paradigm change. Second, if a task-specific program was used for the two tasks that the SAIL robot infant has learned, it cannot run in real-time without special image process hardware, due to the extensive computation required for image analysis. Apart from the appearance-based methods, almost no other image analysis methods can run in real time without special-purpose image processing hardware. Third, detecting an arbitrary object from arbitrary background is one of the most challenge tasks for a robot. The main reason that our developmental algorithm can learn to do this challenging task is that it does not rely on human to pre-define representation. The same is true for our autonomous navigation experiment — the amount of scene variation along the hallways of our engineering building is beyond hand programming.

The automatically generated representation is able to use context very intimately. Every action is tightly dependent on the rich information available in the sensory input and the state. In other words, every action is context dependent. The complexity of the rules of such context dependence is beyond human programming. A human defined representation is not be able to keep such rich information, without making the hand-designed representation too complicated to design any effective rules.

Since the developmental algorithm is not task specific, we plan to train the SAIL robot for other more tasks to study the limitation of the current SAIL-2 developmental algorithm as well as the SAIL robot design.

Acknowledgement

The work is supported in part by National Science Foundation under grant No. IIS 9815191, DARPA ETO under contract No. DAAN02-98-C-4025, DARPA ITO under grant No. DABT63-99-1-0014, and research gifts from Siemens Corporate Research and Zyvex.

A Vector representation for an image

A digital image with r pixel rows and c pixel columns can be represented by a vector in (rc)-dimensional space S without loss of any information. For example, the set of image pixels $\{I(i,j) \mid 0 \leq i < r, 0 \leq j < c\}$ can be written as a vector $\mathbf{X} = (x_1, x_2, \cdots, x_d)^t$ where $x_{ri+j+1} = I(i,j)$ and $d = rc$. The actual mapping from the 2-D position of every pixel to a component in the d-dimensional vector $\mathbf{X}$ is not essential but is fixed once it is selected. Since the pixels of all the practical images can only take values in a finite range, we can view S as bounded. If we consider $\mathbf{X}$ as a random vector in S, the cross-pixel covariance is represented by the corresponding element of the covariance matrix Σ_x of the random vector $\mathbf{X}$. This representation, used early by [7] and [12], have been widely used by what is now called *appearance-based* methods in the computer vision literature. Using this new representation, the correlation between any two pixels are considered in the covariance matrix Σ_x, not just between neighboring pixels.

B Principal component analysis

The principal component analysis [4] computes an orthonormal basis from a set of sample vectors. Computationally, it computes the eigenvectors and the associated eigenvalues of the sample covariance matrix Γ of the samples. When the number of samples n is smaller than the dimensionality d of the sample vectors, one can compute the eigenvectors and eigenvalues of a smaller $n \times n$ matrix instead of a large $d \times d$ matrix.

C Linear discriminant analysis

Originally, Fisher's linear discriminant analysis (LDA) was developed to discriminate two classes. The result is later extended to more than two classes. Two matrices are defined from the pre-classified training samples, within-class scatter matrix W and between-class scatter matrix B. The LDA computes a basis of a linear subspace of a given dimensionality in S so that in the subspace, the ratio of the between-class scatter over the within-class scatter in maximized. The basis of this linear subspace defines the MDF feature vectors. Computationally, these basis vectors are the eigenvectors of $W^{-1}B$ associated with the largest eigenvalues [4]. When W is a degenerate matrix, LDA can be performed in the subspace of PCA.

References

1. Martin Bichsel. *Strategies of Robust Object Recognition for the Automatic Identification of Human Faces*. Swiss Federal Institute of Technology, Zurich, Switzerland, 1991.
2. Lawernce Birnbaum, Matthew Brand, and Paul Cooper. Looking for Trouble: Using Causal Semantics to Direct Focus of Attention. In *Proc of the IEEE Int'l Conf on Computer Vision*, pages 49–56, Berlin, Germany, May 1993. IEEE Computer Press.
3. J.L. Elman, E. A. Bates, M. H. Johnson, A. Karmiloff-Smith, D. Parisi, and K. Plunkett. *Rethinking Innateness: A connectionist perspective on development*. MIT Press, Cambridge, MA, 1997.
4. K. Fukunaga. *Introduction to Statistical Pattern Recognition*. Academic Press, New York, NY, second edition, 1990.
5. G. H. Golub and C. F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, 1989.
6. W. Hwang, J. Weng, M. Fang, and J. Qian. A fast image retrieval algorithm with automatically extracted discriminant features. In *Proc. IEEE Workshop on Content-based Access of Image and Video Libraries*, pages 8–15, Fort Collins, Colorado, June 1999.
7. M. Kirby and L. Sirovich. Application of the karhunen-loève procedure for the characterization of human faces. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 12(1):103–108, Jan. 1990.
8. S. L. Pallas L. von Melchner and M. Sur. Visual behavior mediated by retinal projections directed to the auditory pathway. *Nature*, 404:871–876, 2000.
9. W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes*. Cambridge University Press, New York, 1986.
10. M. Sur, A. Angelucci, and J. Sharm. Rewiring cortex: The role of patterned activity in development and plasticity of neocortical circuits. *Journal of Neurobiology*, 41:33–43, 1999.
11. D. L. Swets and J. Weng. Hierarchical discriminant analysis for image retrieval. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 21(5):386–401, 1999.
12. M. Turk and A. Pentland. Eigenfaces for recognition. *Journal of Cognitive Neuroscience*, 3(1):71–86, 1991.
13. J. Weng. The living machine initiative. Technical Report CPS 96-60, Department of Computer Science, Michigan State University, East Lansing, MI, Dec. 1996. A revised version appeared in J. Weng, "Learning in Computer Vision and Beyond: Development," in C. W. Chen and Y. Q. Zhang (eds.), Visual Communication and Image Processing, Marcel Dekker Publisher, New York, NY, 1999.
14. J. Weng and S. Chen. Vision-guided navigation using SHOSLIF. *Neural Networks*, 11:1511–1529, 1998.