NEURAL MODELS OF INCREMENTAL SUPERVISED AND UNSUPERVISED LEARNING

THESE NO 863 (1990)

PRESENTEE AU DEPARTEMENT D'INFORMATIQUE

ECOLE POLYTECHNIQUE FEDERALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ES SCIENCES

PAR

A.I. ETHEM ALPAYDIN

Ingénieur informaticien diplômé de l'Université de Boğaziçi, Istanbul
de nationalité turque

acceptée sur proposition du jury :

Prof. J-D. Nicoud, rapporteur
M. P. Clarke, corapporteur
Prof. Ch. Jutten, corapporteur
Prof. L. Personnaz, corapporteur

Lausanne, EPFL
1990

To the memory of Pumuckl

Acknowledgments

This research is supported by the Fonds National Suisse de la Recherche Scientifique.

I thank Jean-Daniel Nicoud for giving me a chance by engaging me when I had nothing accomplished, and the patient support he provided despite my stubbornness over the three years this work lasted.

Peter Clarke, Christian Jutten, and Léon Personnaz, I thank for stimulating discussions on various occasions which shaped considerably the evolution of my research and their invalueable feedback on this dissertation.

The database of handwritten numerals used throughout the work was kindly supplied by Isabelle Guyon. The tests whose results are given in section 2.10 were made during a short stay at the AT&T Bell Labs at Holmdel, NJ. I would like to thank Dr. L. Jackel and other members of the group for useful discussions and the help they have given.

Marie-Jo Pellaud, a unique combination of warmth and efficiency, I thank, for helping me in all sorts of things, large and small. She did not even say a word when I monopolized her Mac twice each for one month to typeset this dissertation.

Gregory Baratoff wrote the first versions of GAR and GAL simulators of section 3.6 for which I thank him.

I owe a lot to my colleagues at the Laboratoire de Microinformatique, past and present. They showed me understanding, patiently explaining over and over, trying to make some sense out of my babblings in French. The way you get good ideas is by working with talented people you have fun with.

The group Carnac is a wonderful forum to learn and discuss many things connectionist; its members are ready to share their knowledge, listening, correcting, and guiding the novice.

My family and my friends, near and far, new and old, supported and encouraged me. It is due to their confidence in me that I have dared to undertake enterprises which I could not otherwise. Although mere thanks are inadequate, I thank them.

Contents

Contents i
Résumé iii
Abstract vii
Preface xi

1. Introduction And Preliminaries

1.1. System Operation 3
1.2. Decision Theory 8
1.3. Parametric Classification 9
1.4. Non-parametric Classification 10
1.5. Quantifying Success In Classification 13
1.6. Quality Assessment 13

2. Incremental Supervised Learning

2.1. Learning And Adaptation 17
2.2. Designing A Supervised SM Network 18
2.3. Grow And Learn (GAL) 20
2.4. GAL: A Didactic Example 23
2.5. Forgetting In GAL 30
2.6. GAL With Reject 34
2.7. Comparing GAL With Similar Algorithms 37
2.8. Recognition Of Handwritten Numerals Using GAL 49
2.9. Preprocessing Handwritten Numerals Before GAL 50
2.10. Testing GAL With A Big Database 54
2.11. Learning A Mapping With GAL 57
2.12. A Critique Of GAL 61

3. Incremental Unsupervised Learning

3.1. On Features 65
3.2. Incremental Unsupervised Learning 66
3.3. Grow And Represent (GAR) 67
3.4. GAR: A Didactic Example 69
3.5. Comparing GAR With The Self-Organizing Map 74
3.6. Development Of Feature Detector Cells By GAR 75
3.7. A Critique of GAR 79

4. Conclusions

4.1. A Biological View 83
4.2. Incremental Learning 85
4.3. GAL 86
4.4. GAR 87
4.5. Future Directions 87

References 89
Curriculum vitae 93
Publications 94

Modèles neuronaux d'apprentissage incrémental supervisé et non-supervisé

Résumé de la thèse de E. Alpaydın
LAMI-EPFL, Juillet 1990

Mots-clés. Réseaux de neurones artificiels, Parallélisme, Apprentissage incrémental, Apprentissage supervisé, Apprentisage non-supervisé, Reconnaissance de formes, Reconnaissance de caractères.

C'est en détectant les régularités et les invariants que les systèmes, soit naturels, soit artificiels, peuvent réduire la complexité des signaux qu'ils doivent traiter. C'est une opération d'abstraction dont l'une des formes est la classification. Le signal est assigné à une classe générale indépendamment de sa forme particulière.

De nombreuses tâches ne sont pas encore formalisées; par exemple, en reconnaissance de formes, on ne peut pas écrire une définition générale de la lettre 'A' comme une équation mathématique. La seule possibilité est de l'apprendre à partir des exemples. Le critère pour décider si une image correspond à un 'A' ou pas est de calculer si cette image est assez similaire à un des 'A' vu auparavant. Il y a, entre autres, deux problèmes ici pour lesquels deux méthodes sont proposées dans la présente thèse:

[1] Un problème est de pouvoir *extraire la définition d'une classe à partir des exemples.* Une méthode qui s'appelle Grow-and-Learn (GAL) est proposée pour ce but avec la propriété que la définition d'une classe est *extensible.* On commence par la définition la plus simple et on l'étend quand des classes similaires sont rencontrées pour pouvoir extraire les différences plus fines.

[2] Un autre problème est de *trouver un critère de similitude.*

a. Une possibilité est de choisir un ensemble de *caractéristiques importantes* sur lesquelles on se base pour comparer les formes. Une méthode qui s'appelle Grow-and-Represent (GAR) est proposée pour apprendre de telles caractéristiques à partir des exemples. Un réseau de GAR, comme celui de GAL, est un réseau dynamique dont la structure est modifiée pendant l'apprentissage.

b. Une autre possibilité est d'utiliser sa connaissance de l'application pour pouvoir définir des *techniques de pré-traitement* qui faciliteront l'apprentissage de classes. Deux techniques sont proposées pour la reconnaissance de chiffres manuscrits dans ce but.

Toute nouvelle approche doit être évaluée et testée prudemment avec une application réaliste. Les deux méthodes proposées ont été testées avec des chiffres manuscrits dont la reconnaissance présente un intérêt économique évident, mais il faut noter que ces méthodes ne sont pas limitées à la reconnaissance de caractères ou à la vision, mais peuvent être utilisées avec d'autres types d'images ou de signaux.

L'opération effectuée par un système pour générer une certaine sortie, est une fonction de l'entrée et de la structure du système. Cette structure est composée de deux types de composants: ceux qui sont modifiables et ceux qui sont fixes. Les composants modifiables, quand ils existent, s'appellent les *paramètres* et constituent la *mémoire* du système. Lorsqu'un système ne correspond plus aux besoins imposés par l'environnement, il doit être modifié. La modification d'un système, quand elle est faite en modifiant les paramètres, s'appelle *l'adaptation paramétrique.* Parfois elle nécessite la modification de la structure et dans ce cas une *adaptation structurelle* doit être effectuée.

Les règles d'apprentissages actuellement utilisées dans le domaine des réseaux neuronaux, sauf quelques exceptions, sont basées sur la modification paramétrique. La structure du réseau, c'est-à-dire, les nombres d'unités cachées et leurs interconnexions sont définies par le programmeur du réseau et l'algorithme ne peut modifier que les poids des connexions. Il n'y a aucune règle qui permet de décider la structure nécessaire à partir d'une application ou d'un ensemble d'apprentissage donné.

Si la structure choisie ne correspond pas bien à ce qui est nécessaire pour cette tâche, le réseau ne converge pas vers le bon résultat.

Deux algorithmes, dits *incrémentaux*, sont proposées ici; ils s'appellent GAL et GAR pour l'apprentissage supervisé et non-supervisé. Dans ces algorithmes, pendant l'apprentissage, des unités et leurs connexions sont ajoutées lorsque c'est nécessaire. On sait que le fait de pouvoir modifier la structure d'un réseau en lui ajoutant des unités et des liens peut diminuer le temps d'apprentissage et améliore la qualité de généralisation.

GAL convient pour l'apprentissage supervisé. On associe à chaque classe un certain nombre d'*exemplaires* qui sont ses membres typiques ou ses prototypes. Etant donné un certain vecteur d'entrée, le plus proche exemplaire déjà stocké est cherché comme dans la méthode du plus-proche voisin en utilisant une mesure de ressemblance qui dépend de l'application. Pendant l'apprentissage, si un vecteur d'entrée est classifié faux, un nouvel exemplaire est créé dans la position du vecteur d'entrée. Aucune modification n'est faite si la réponse du réseau est déjà juste.

Les vecteurs stockés comme des exemplaires dépendent à l'ordre où ils sont vus pendant l'apprentissage; il peut donc arriver qu'un exemplaire ne soit plus nécessaire à cause d'une addition plus récente. On est seulement interessé à stocker les vecteurs proches des frontières de classes, donc si un vecteur qui est plus proche d'une frontière est ajouté, les exemplaires plus à l'intérieur ne sont plus nécessaires. Pour se débarrasser de tels exemplaires et diminuer le besoin en mémoire (donc la vitesse de reconnaissance sur une machine séquentielle), une phase de *sommeil* est introduite pendant laquelle ces exemplaires sont éliminés. Un exemplaire est détruit si l'exemplaire le plus proche appartient à la même classe. L'erreur qui peut augmenter à cause de ces éliminations, est compensée dans la prochaine phase de *réveil* quand le réseau continue à apprendre avec son ensemble d'apprentissage. Les deux phases de *réveil* et *sommeil* s'alternent et finalement, le réseau converge vers un ensemble d'exemplaires où plus aucune addition ou élimination n'est nécessaire.

GAL est comparé aux autres algorithmes d'apprentissage (RCE, LVQ, la rétro-propagation du gradient) du point de vue des besoins en mémoire, en vitesse d'apprentissage, et en succès de reconnaissance. Il est spécialement intéressant du point de vue du temps d'apprentissage. Il permet particulièrement d'apprendre *on-line*, c'est-à-dire directement pendant l'utilisation, car chaque association est apprise dans une itération alors qu'une centaine d'itérations sont nécessaires dans le cas des algorithmes qui font la descente du gradient. GAL est un bon choix si les formes doivent être apprise en temps-réel, par exemple, dans les applications robotiques. Testé avec des chiffres manuscrits, la comparaison avec d'autres algorithmes est aussi favorable tant du point de vue du taux de succès que de la taille de réseau. Avec une base de données qui contient 9000 exemplaires, AT&T Bell Labs obtiennent un taux de succès de 94.3% avec la méthode du plus-proche voisin quand tout l'ensemble d'apprentissage est stocké. La rétro-propagation du gradient, après un apprentissage qui dure 3 jours, arrive à 95.3%. GAL en stockant 10% de l'ensemble d'apprentissage et après 5 heures d'apprentissage arrive à 92.3%, le succès maximum possible étant 94.3%.

GAR convient pour l'apprentissage non-supervisé; il extrait les caractéristiques statistiquement importantes de la distribution du signal d'entrée. Etant donné un vecteur, l'unité la plus proche est cherchée comme dans GAL. Associé avec chaque unité, un seuil définit une région de dominance ayant la forme d'une hypersphère dont le rayon est une fonction du seuil. Une unité est activée si elle est la plus proche et si son activation est plus grande que son seuil. Pendant l'apprentissage, si le vecteur d'entrée ne peut activer aucune unité, une unité est ajoutée dans la position du vecteur d'entrée. Si une unité est activée, ses poids de connexions sont tirés avec un petit facteur vers le vecteur d'entrée .

Comme l'extraction de caractéristiques est une forme de codage, on doit être sûr que pendant ce processus, l'information perdue est minimisée. La théorie de l'information dit que pour maximiser le transfert d'information, des unités qui codent des événements spéciaux, doivent avoir une probabilité égale d'être activées. On en déduit que dans un réseau compétitif où l'espace d'entrée est divisé entre les unités, les unités qui sont dans les régions où la densité est haute doivent avoir des régions de domination petites et les unités qui sont dans les régions où la densité est basse doivent avoir des plus grandes régions de domination. Dans le cas de GAR, on associe à chaque unité, un *compteur de trophée* dont la valeur est augmentée chaque fois que l'unité gagne la compétition et est activée.

Quand cette valeur arrive à une limite, le seuil d'unité est modifié pour diminuer la région de dominance de cette unité.

GAR, pendant la phase de *sommeil*, contrôle la condition que toutes les unités aient la même probabilité d'être activée. Un certain nombre d'itérations est fait sans modification de poids des connexions et les trophées sont comptés. Les unités dont les compteurs de trophées sont inférieurs à une certaine valeur sont éliminées. C'est seulement quand les unités ont la même probabilité d'être activées que le réseau réalise une bonne approximation de la densité du signal d'entrée. Le pouvoir de détruire des unités ayant un petit nombre de trophées et d'en ajouter quand des formes assez différentes sont rencontrées implique que si le signal d'entrée change dans le temps, le réseau est capable "d'oublier" les anciennes caractéristiques en ajoutant les nouvelles.

La thèse montrè de plus qu'une structure ayant une couche de GAR pour l'extraction de caractéristiques et une couche de GAL pour la classification est une bonne alternative à la rétropropagation du gradient. On peut avoir plusieurs couches de GAR pour extraire des caractéristiqes de plus haut niveau.

Neural models of incremental supervised and unsupervised learning

Thesis summary, E. Alpaydın
LAMI-EPFL, July 1990

Keywords. Artificial neural networks, Parallel processing, Incremental learning, Supervised learning, Unsupervised learning, Pattern recognition, Optical character recognition.

It is by detecting the regularities and invariants that systems, be they natural or artificial, can decrease the complexity of the signal that they should process. This is an abstraction operation, and one type of it is classification where the signal is assigned to a general class independently of its particular apparence.

There are a number of tasks that are not yet formalized; for example in pattern recognition, nobody knows how to write a general definition of the character 'A' as a mathematical equation. The only possibility left is to learn from examples. The criterion to decide whether an image is an 'A' or not, is to compute if it is similar to any of the 'A' seen before. There are two problems, among others, for which two methods are proposed in this thesis:

[1] One problem is to be able to *extract the definition of a class from its examples.* A method named Grow-and-Learn (GAL) is proposed which has the property that the definition of a class is *extensible.* One starts from the simplest definition and extends it when similar classes are encountered to be able to extract finer definitions.

[2] Another problem is that of *finding the similarity criterion.*
  a. The idea is to choose a set of *salient characteristics* based on which patterns are compared for similarity. A method named Grow-and-Represent (GAR) is proposed to learn such characteristics from examples. A GAR network, similar to GAL, is a dynamic network whose structure is modified during learning.
  b. Another possibility is to use one's knowledge of the task to be able to heuristically define application-dependent *preprocessing techniques* that will facilitate learning of classes. Two techniques have been proposed for recognition of handwritten numerals towards this aim.

Any new approach should be evaluated and tested carefully with a real application. Despite the fact that the methods proposed here are not limited to character recognition or vision, but can be used with other types of images or signals, all models proposed have been tested with handwritten numeral recognition which has an evident economical utility.

The operation carried out by a system to give out a certain output, is a function of the input and the system's structure. This structure is composed of two types of components: the modifiables and the fixed ones. The modifiable components, when they exist, are called the *parameters* and make up the *memory* of the system. The modification of a system, when done by modifying the parameters is called *parametric modification.* But sometimes it requires the modification of the structure also, in such a case, a *structural adaptation* should be carried out.

Learning rules currently used with neural networks, a few exceptions aside, are based on parametric modification. The structure of the network, i.e., the number of hidden units and their interconnections, is defined by the programmer and the learning rule can modify *only* the connection weights. There is no rule which allows one to determine the necessary structure from a given application or training set. If the structure chosen does not correspond to what is necessary for that task, the network does not converge to a good result.

Two *incremental* algorithms are proposed here for supervised and unsupervised learning which are called GAL and GAR respectively. With these algorithms, the network has a dynamic structure; units and their connections are added during learning when necessary. It is known that the ability

to modify network structure by adding units and links may decrease learning time and improve generalization quality.

GAL is for supervised learning. Associated with each class is a certain number of *exemplars* which are its typical members or its prototypes. Given a certain input vector, the closest is sought as in nearest-neighbor method using an application-dependent distance measure. During learning, if an input vector is currently classified wrongly, a new exemplar is added at the position of the input vector. No modification is done if the network response is already correct.

As the actual vectors that are stored as exemplars depend on the order they are encountered during learning, it may be the case that an exemplar is no longer necessary due to a recent addition. One is only interested to store the vectors closest to class boundaries thus if a vector closer to the boundary is added, the ones interior are no longer necessary. To get rid of such units and decrease memory requirements (and recognition speed on a sequential machine), a *sleep* phase is introduced during which such units are eliminated. An exemplar unit, if the closest exemplar to it also belongs to the same class is purged. Error that may be increased due to such deletions are compensated for in the next *awake* phase when the network goes on learning with the training set. Succesive *awake* and *sleep* phases, allow the network to converge to a set of exemplar units where no further additions or deletions are necessary.

GAL is compared with other learning algorithms, e.g., RCE, LVQ, and gradient-descent based methods, in terms of memory requirements, recognition speed, and success. It proved itself to be interesting especially in terms of learning time. In particular, it allows *on-line* learning as an association is learned at one-shot, i.e., in one iteration, whereas hundreds are necessary in the case of gradient-descent based algorithms. GAL is a good choice when patterns are to be learned in real time, e.g., in applications in robotics. The computation is not complex and thus can easily be realized in hardware. It also compared well in terms of success and network size when tested with handwritten numerals. With a database containing around 9000 examples, AT&T Bell Labs obtained 94.3% with the nearest-neighbor method when all the training set is stored. Back-propagation of the gradient, after learning for three days, reaches 95.3%. GAL, storing 10% of the training set and after five hours of learning reaches 92.3%, maximum possible success being 94.3%.

Although it is for learning of categories, GAL can easily be extended to learn also continous mappings. Discretizing the range into segments of small size and treating these segments as if they are different classes, GAL can very quickly learn such mappings. The tolerance value chosen during this discretization determines the finesse of the mapping.

GAR is for unsupervised learning; it extracts statistically important features of the input signal distribution. Given an input vector, the closest unit is sought as in GAL. Associated with each unit is a threshold that defines a region of domination having the form of a hypersphere whose radius is a function of the threshold. A unit is activated by the input vector if it is the closest and if its activation is greater than its threshold. During learning, if the input vector cannot activate any unit, a unit is added at the position of the input vector with an initially large domination region. If the input vector activates a unit, that unit's weight vector is moved towards the input vector with a small gain factor.

As feature extraction is a way of encoding, one should make sure that during this process, information loss is minimized. It is known that units that code for special events, to maximize information transfer, need to have equal probability of being activated. This implies that in a competitive mechanism where the input space is divided between units, those units that lie in high density regions should have small region of dominations whereas in low density regions, units should have larger regions of domination. To achieve this in GAR, associated with each unit, is a parameter named a *trophy counter* whose value is increased whenever a unit wins the competition and gets activated. When this counter reaches a certain limit value, the threshold is modified to get a smaller region of domination.

GAR in *sleep* mode, checks if the condition of equal probability of being activated is satisfied or not. A certain number of iterations are performed without any weight vector modification and trophies are counted. Units whose trophies are smaller than a certain value are deleted. Only when

the units have equal probability of being activated, does the network performs a good approximation of the probability density of the input signal. The ability to delete units with low trophies and adding when significantly different patterns are encountered, also implies that when the input signal changes in time, the network is able to "forget" the old features and learn the new ones.

The thesis also shows that having a multi-layer structure made up of a GAR layer for feature extraction and a GAL layer for classification is a good alternative to back-propagation. One, of course, can have several GAR layers to be able to extract higher-order features.

Preface

I have a number in my head
Though I don't know why it's there
When numbers get serious
You see their shape everywhere
—Paul Simon, "When numbers get serious"

A few words of explanation are called for before one proceeds to the following chapters.

This is not an introductory text nor a review of any of the subjects to which this work is related. None of the fields that this work touches, learning, pattern recognition, nor artificial neural networks, need another introductory text. So many of such books have been written in recent years by people who have a lot more knowledge and experience than I do that I have left altogether aside giving an overview of any of these fields.

As time passes, needs of society get ever more complex. Designing and building machines for more and more complicated and detailed tasks is rapidly becoming tiresome, not to mention maintaining them during utilization. General-purpose learning machines or machines that can build, program, or supervise other machines may sound like science-fiction but seems to be the only way out. Self-modifiability, or adaptation during operation is a must, not an extra feature in any application. The idea followed in the work is to have a general-purpose system that can learn to recognize simple forms, be they characters of any alphabet, icons, signs, or whatever. Thus none of the techniques proposed in the text is *limited* to optical character recognition but they are all *tested* with real-world visual forms, namely, handwritten numerals.

There are a number of tasks nobody knows how to formalize, for which there is no complete computational theory. To take an example, nobody knows how to define 'A' as a mathematical equation. The only way left is to learn from examples. The criterion one uses to decide whether a certain image is an A or not, is whether it is similar enough to any of the A seen before. The problem now turns into that of defining an appropriate similarity measure.

Whether just a coincidence or not, but the tasks we cannot formalize are those that we do in everyday life, which, for this very reason, are good candidates to be implemented in machinery. Not only us, but also a great part of the animal kingdom, can accomplish such tasks seemingly very easily.

Maybe I should stress at this point that if one builds a machine to perform a certain task, then the machine should be superior to a human for that particular task. Either it operates in environmental conditions not suitable for a human, or is faster, or cheaper, or does not take coffee breaks. It is prime time that we should get rid of our chauvinism related to brain or any other biological system and stop seeing a struggle of "artificial" versus "natural." A thing is good because it is good, not because it is natural. A thing is bad because it is bad, not because it is artificial. Surrounded by machines whose inner functioning they do not understand, and which in a hundred years or so started to dominate job territories and enterprises previously theirs, people find relief in hearing that they have 10^{10} neurons in their brains which is a far greater number of components than computers have, or that even an amoeba has as much information in its DNA as 1,000 *Encyclopedia Brittanicas.* [1] One unfortunately notes that what once initially started as ignorance finally led to a

[1] *This calculating of numbers, converting them into bits and speculating on information storage capacity is mere fancy. What is important is not really the capacity to store that many states but the machinery that will be able to detect and process such states. Otherwise one is tempted to say: "There are so many thousands of dust particles in one square centimeter of cloth, there are that many millions of atoms in one*

hostility towards a "system" which people identify with science and which they take as destroying the "meaning" of their lives. It is not strange then that these people get good hold of their "mind" and "intelligence" and be unwilling to accept that those may someday be implemented on a machine. There is nothing degrading about thinking of brain as a certain machine; what is degrading is to think of it as unable to understand how something works, even if that something is itself.

Understanding a system implies being able to form a computational theory of it as pointed out by David Marr. However for the same task, one may propose explanations at various levels. One may look at the level of ions moving around, or at the level of individual neurons, at the network level or at the overall system level. For each of these levels, one specifies a certain data representation and an algorithm to perform the required operation on these data, which is then implemented using a certain technique. According to the complexity of the task and implementation constraints, one then chooses a particular level of algorithm and implementation which is formalized independently of the complexities of the underlying layers. For example, if one is at the level of neurons, the fact that a neuron fires is important but the chemical processes that take place to make the neuron fire are not.

The symbolic approach in artificial intelligence dominating since 1960s, concentrates on the highest psychological level in understanding and replicating intelligence. The idea is to simulate directly the cognitive capabilities using a set of procedures in Lisp or Prolog on the pretext that the behavior of the system can be written as a set of symbolic transformations and that the rules of these transformations are independent of how these symbols are represented. The connectionist or subcognitive approach works at a lower level, at the level of neuronal networks. The idea is that what we call intelligence is the behaviour that emerges as a result of the interaction of concurrently operating non-linear units performing simple computations. Learning in these networks correspond to dividing a global task like pattern recognition into a set of simple tasks over a network of processing units which execute in parallel and which constantly exchange information over the links connecting them.

The problem with the symbolic approach, I believe, is that it ignores the physiology of the brain. Whatever the brain does is constrained by what it can or cannot do, so, trying to build artificial correspondants of human abilities without taking into account how humans do them is an error. Similarly there is a danger with the second approach when it concentrates on very low-level details. Say we take a lobster and we are able to record for all neurons in its brain, the input and output relationships. In a huge memory, we store what all neurons should give as output when they are given this or that as input and using a very powerful computer, we are able to do this in real time. There is no doubt that this computer would exactly behave like the lobster it simulates. However this does not give us any knowledge at all about the behaviour of the lobster. Carver Mead's work on artificial retina by replacing synapses with transistors is a good example for this. What we want is a higher-level explanation telling us why the neurons bother to take those values given these inputs. For example, we can say that retinal ganglion cells enhance edges and one way to enhance edges is by applying a Laplacian filter and one way to implement a Laplacian filter is by a cell having an on-center-off-surround receptive field. But there *are* other ways to implement a Laplacian filter and there are other ways to enhance edges. We are in no way bound to use the same implementation technique or algorithm when implementing a human capability *in silico*.

A computer program cannot be more intelligent than its programmer and should not be more intelligent than its user. In the case of adaptive or learning systems, this has a corollary: A learning system cannot be more intelligent than its training set and should not be more sophisticated than what its test set necessitates. Thus the system learns from examples and what it knows is limited to the set of examples shown. The learning process is one of induction and it is required that the system when defining the classes, finds the simplest possible conditions for it to generalize well to patterns unseen during learning.

Another requirement is that in some tasks rapid adaptation is needed. In such cases, the system does not have time to perform thousands of iterations to be able to gradually learn by reinforcement.

dust particle, thus everytime you clean your jacket, you are throwing one volume of Encylopedia Brittanica away.

Think of a robot that is sent to Mars; there is no training set and no quick human remote interference is possible. The robot somehow should be able to make use of what it encounters, learn from them as quickly as possible and thus be able to successfully accomplish its task.

The approach named *incremental learning* implies a dynamic network whose structure gets modified during learning. Unlike most popular learning algorithms where only the connection weights are modified iteratively, in an incremental network, units and links can also be added. Such a strategy satisfies the previously mentioned two conditions, namely:

- One starts from the simplest possible definition of a class. This definition, during learning when necessary is supplemented to get more complicated definitions. When similar patterns that belong to different classes are encountered, units are added to get finer class separations.
- As learning a new pattern necessitates one iteration requiring either network modification or no change at all, this way of learning is very fast. This makes such algorithms quite interesting where on-line learning is necessary, e.g., robotics.

The work involves two methods for incremental learning of classes, i.e., supervised learning, and features, i.e., unsupervised learning. The requirements during supervised learning are already mentioned. In the case of unsupervised learning, the aim is to be able to extract features automatically given examples. One first requires a fast capability to learn features when significantly different patterns are seen; this also implies an incremental strategy. Secondly, one requires that in a changing environment where the input signal changes in time drastically, the features should change also implying forgetting the old ones and learning the new.

To make the idea more clear, let me mention the driving force that led me look for incremental solutions. The task is to learn a visual object as a sequence of eye fixations.

One wants to perform recognition as fast as possible, thus the sequence, i.e., the eye trajectory, should be kept as short as possible. Initially differentiating between quite different objects, just one or two fixations may be sufficient but as similar objects are seen, the learning system should be able to extend automatically the sequences as to be able to differentiate between similar objects. Because one does not have enough time to perform many iterations, a method is needed by which this change in the sequence can be done very rapidly.

An object's definition is not only related to the trajectory followed by the eye but also to what the eye sees, i.e., the small region of the object currently accessible to analysis. Because the whole operation needs to be carried out very rapidly, one also needs a mechanism by which the content of such regions can be learned at one shot. This implies an ability to learn rapidly significantly different features. One also requires that if fine differences exist between commonly occurring features, such differences can be caught to allow a finer definition of objects.

Both of these pointed out a need for incremental learning, supervised and unsupervised.

The following text is organized as follows:

In the first chapter, a general introduction is made to several concepts used throughout the rest of the text. One point important made there is the distinction between what I call structural and parametric adaptation.

The second chapter proposes Grow-and-Learn (GAL); an incremental supervised learning algorithm. It is explained and compared with similar algorithms like RCE, LVQ, and discriminant-based approaches like back-propagation.

Grow-and-Represent (GAR) is an incremental unsupervised learning algorithm. It is explained and compared with Kohonen's self-organizing map in the third chapter.

The final chapter draws the conclusion.

1

Introduction and Preliminaries

(Socrates, narrating his discussion with Glaucon)

Just as in learning to read, I said, we were satisfied when we knew the letters of the alphabet, which are very few, in all their recurring sizes and combinations; not slighting them as unimportant whether they occupy a space large or small, but everywhere eager to make them out; and not thinking ourselves perfect in the art of reading until we recognize them wherever they are found:

True—

Or, as we recognize the reflection of letters in the water, or in a mirror, only when we knew the letters themselves; the same art and study giving us the knowledge of both:

Exactly—

Even so, as I maintain, neither we nor our guardians, whom we have to educate, can ever become musical until we and they know the essential forms, in all their combinations, and can recognize them and their images wherever they are found, not slighting them either in small things or great, but believing them all to be within the sphere of one art and study.

— Plato, "The Republic," Book III, 402.

1.1. SYSTEM OPERATION

A system operates as to give an output (κ) which is a function (ϕ) of the system state (σ) and current input (λ):

$$\kappa = \phi(\lambda, \sigma). \tag{1.1}$$

Associated with each input is a certain required output, γ. The aim of building a system is to have κ equal to γ.

In the context of pattern recognition, λ is the input pattern, e.g., a character image, a speech signal, κ is the class to which the input belongs to, e.g., a character code, identity of the speaker, and $\phi()$ corresponds to the recognition or classification operation.

σ, the system state, covers all we need to know about the operation of the system. There are two ways by which σ may be realized: The machine may be *built* by a higher-level designer, or the machine may *learn* by itself. In a general sense, σ has two parts: those which are fixed and those which are modifiable. The modifiable parts constitute the *memory*, and because the content of the memory affect $\phi()$, memory locations are also called *parameters* whose values can get different values at different times to get different output.

$\phi()$ is a universal function which, given σ and a particular input λ, gives the output κ. For example, $\phi()$ may be a computer and σ its program. Building up a system means:

[1] Finding out what is expected from the system; what output is required given a certain input, i.e., the (λ,γ) pairs.
[2] Defining σ so that these mappings can be accomplished.

symbol	concept
λ	input
κ	actual output
γ	required output
σ	system structure
$\phi()$	operation function
$\pi()$	performance measure
η	performance or cost of the system
$\mu()$	probability density of λ
$\nu()$	joint probability of λ and γ
$\psi()$	adaptation function

It may be the case that we know exactly the function mapping from λ to γ in which case we can write down σ exactly. In such a case, the system is built and when the designer is confident that no significant changes will occur in the environment, it may be fixed.

In general however, what we only have is a set of observations, λ and corresponding γ values, which constitute our empirical knowledge. These pairs make up the so-called *training set*. The aim then, is to define σ based on these values so that when a certain λ is given as input, κ will be equivalent to γ. That λ may or may not be an element of the training set.

The performance of a system may be measured by defining a distance measure between κ and γ, $\pi()$. System performance, η, can then be computed by summing up over all possible λ:

$$\eta = \int \pi(\gamma, \phi(\lambda, \sigma)) d\lambda. \tag{1.2}$$

When λ are not uniformly distributed, we are more interested in "average success" and want to take into account the probability density of λ, $\mu()$. In such a case, η is a weighted sum where weights are densities (White, 1989):

$$\eta = \int \pi(\gamma, \phi(\lambda, \sigma))\mu(d\lambda)d\lambda. \tag{1.3}$$

π is normalized so that $\pi(\gamma, \gamma) = 0$ and as $\pi()$ becomes bigger, performance gets worse. The system should be built, i.e., σ should be specified, so as to minimize η (Fig. 1).

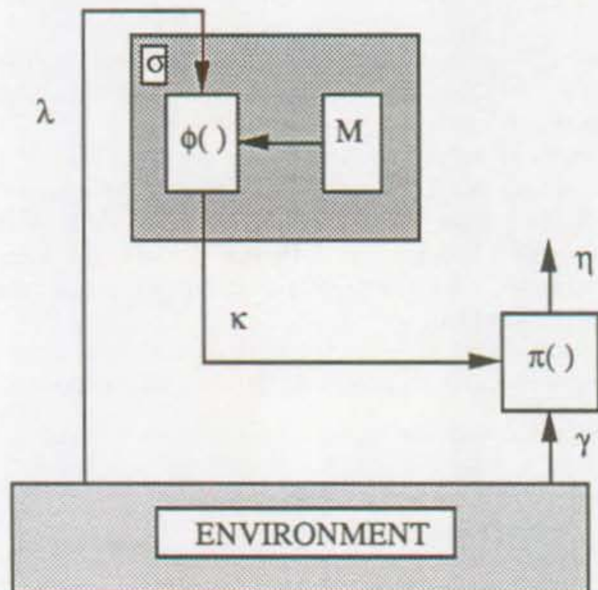

Fig. 1. System operation and parameters. M is the memory which is optional.

1.1.1. Adaptation

A system, if it does, may learn in two ways: by *instruction* or by *experience*.

In learning by instruction, the system is given explicit information concerning what is required of it. In learning by experience, the system gives a certain output, κ, receives feedback from the environment, η, changes itself by modifying modifiable parameters of σ and retries. Learning by experience thus is iterative.

One possibility is to store explicitly the entire training set which is the easiest way of learning by instruction. Training sets however tend to be big and highly redundant. Thus an intelligent way would be to decrease memory and computational requirements by summarizing the knowledge lying therein.

If we define $\nu()$ as the joint probability of γ and λ, it is clear that when there is no "noise," knowledge of $\mu()$ implies knowledge of $\nu()$. When there is noise however, as our empirical knowledge, (λ,γ), includes noise, $\nu()$ does not reflect exactly $\mu()$. In such a case, $\phi()$ becomes a probabilistic relationship between λ and γ and is equal to the expected value of κ realized "on average," given λ:

$$\phi(\sigma, \lambda) = E(\kappa|\lambda). \tag{1.4}$$

which may be erroneous. The average of error is by definition 0. When there is no noise:

$$\gamma = E(\kappa|\lambda). \tag{1.5}$$

η is written down as a function of σ given a certain training set. The aim then is to find the global minimum of this function; given a certain initial σ, modify it to σ' so that:

$$\eta(\sigma') = \min_i \eta(\sigma_i). \tag{1.6}$$

η with an analogy to physical systems is named the *energy* function, or with an analogy to optimization problems, the *cost* function. Many algorithms have been proposed in the past to be able to perform a search towards the global minimum in such problems. See (Ackley, 1987) for a review.

Thus, a system when it is modifiable, has its parameters modified in such a way as to minimize η. The adaptation function, $\psi()$, if there is one, modifies system state, σ, as to decrease η (Fig. 2).

$$\sigma(t+\tau) = \psi(\sigma(t), \lambda, \eta, \kappa, \gamma). \tag{1.7}$$

τ is the time constant of the system. The modification of the system state is done generally by modifying the values of memory locations, each one named a *parameter*.

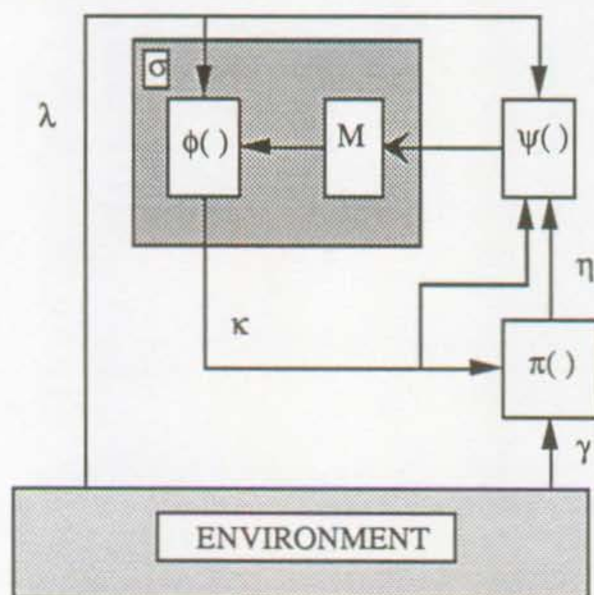

Fig. 2. System operation, parameters, and adaptation. $\psi()$ can also be a part of σ.

In Fig. 2, we may also envisage to put memory before $\pi()$ and $\psi()$ and have them also modifiable in time by a function which we may name $\psi^2()$. In this way, cost function can change in time; the system may learn how to teach or may change what it is teaching.

Note that the final σ achieved at the end which may arrive to a global minimum (but which also risks to get caught in a local minimum), depends heavily on several factors:

[1] The system design parameters, e.g., $\pi()$,
[2] The training set characteristics, e.g., (γ, λ), $\mu()$,
[3] The environmental factors, e.g., $\nu()$.

Testing the system under conditions where any of these different factors are different, will not give good results. The set of (λ,γ) used to test a system's performance is called the *test set*. To be fair, no adaptation occurs during test.

An *adaptive* machine is one that is modified so that its output will be correct when the environment changes (Holland, 1975). This modification can be done by a high-level system or by the system itself. The word "adaptation" generally implies the more interesting latter case. It should be pointed out that *a modifiable machine is superior to a fixed one only if the system requirements,*

i.e., imposed by the environment, change in time. Making adaptation possible, increases system complexity and thus cost.

As biologists emphasize (Dawkins, 1982), adaptedness is recognized as an informational match between organism and environment. An animal that is well adapted to its environment can be regarded as embodying information about its environment, in the way that a key embodies information about the lock it is built to undo. A camouflaged animal has been said to carry a picture of its environment on its back.

Modification may be of two types: In *structural* modification, the structure does not have any modifiable parts and any modification requires restructuring the system. In *parametric* modification, the structure of the system is defined in terms of some parameters; by changing these parameters, different structures, thus different operations are achieved. Adaptation that may be achieved through parametric adaptation is limited to how these parameters are defined. Structural modification, in theory, has no such limitation. In a parametrically modified machine, the existence of parameters whereby information may be stored means that the system has *memory* in which result of past experiences, at least indirectly, are stored. In a structurally modified machine, the machine *itself* is the result of past experiences.[1]

[1] *Evolution by "survival of the fittest" is structural modification, education is parametrical adaptation.*

1.1.2. Example: Learning Logic Array

The system should somewhat be able to store λ and corresponding γ values so that, when a certain input is given, the required output can be given out. One possibility when possible λ values are of a finite size, is to explicitly store all such pairs and look for a match when an input is given. This idea can be used when the dimensionality of λ is small, as memory size increases exponentially. As a simple example to see how a system modifies its parameters during training as a function of the error, let us take a case where a system can learn a logic function from examples, named a *Learning logic array* (LLA) (Alpaydın, 1990a).

When a logic function with some inputs and output is to be implemented, it is generally hard-wired using some, preferrably optimal, number of logic gates. When the specification of the function is modified, the system is halt, the old circuit is removed, and a new circuit is placed in its stead. One approach is to make the gates programmable, thus allowing the modification of the logic function by just modifying the values of some memory elements, i.e., enabling or disabling connections. This second approach is the *soft* approach, as opposed to the former *hard* approach. The third approach proposed here is the *learning* approach, where modification of these memory elements are also performed by the system itself, thus removing the need of a higher-level supervisor completely, who either needs to build the new circuit in the hard approach, or determine and program the connectivities in the soft approach. The higher level informs the system with the required output and the system modifies its parameters in a rather simple way to accommodate it.

The idea is very simple. The diagram for a two-input function is given in Fig. 3. There lies a first layer which is a n to 2^n decoder. The decoder layer gives out unit vectors, i.e., one of the outputs is "on" at a time, others are all "off," thus the actual output of the function can be computed using an OR gate. The connections from the decoder outputs to the actual output are enabled or disabled according to the states of J-K flipflops, each governing one line. The state of the flipflop, *as can also be written as a logic function*, can be determined by the system itself. One gives the required output, r, by which the system checks if there is an error using an XOR gate.

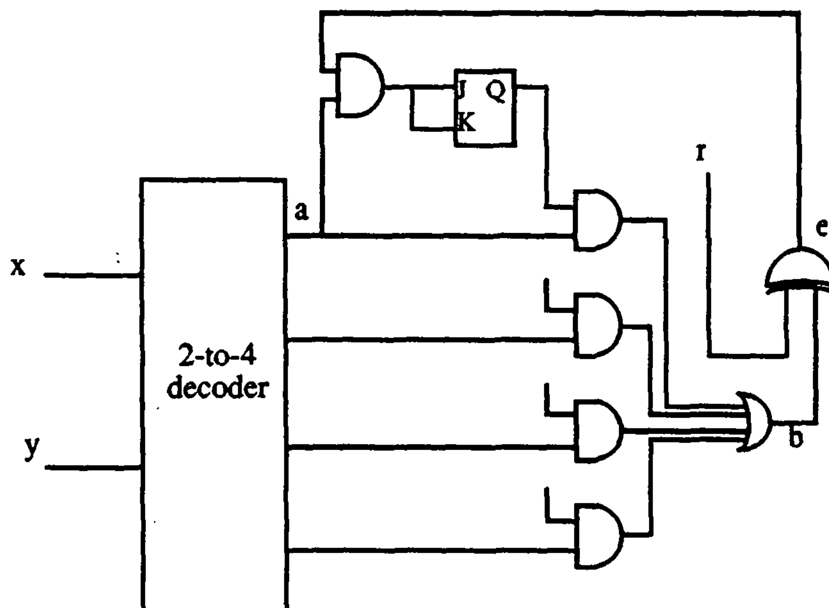

Fig. 3. A two-input, one-output LLA. x and y are inputs, a is the only "on" output of the decoder, b is system's output, r is the required output, e is the error signal when b and r do not match. Only one of the "and gate-flipflop" connection controller are shown, there are altogether four.

First, the connection from the decoder output a to output b is important only when a is "on." Besides, when there is no error, nothing needs to be modified. But when there is an error, the error

should be due the connection connected to the currently "on" output of the decoder. The state of that connection should be toggled, i.e., disabled if currently enabled, and vice versa. A J-K flipflop when both inputs are "on" acts as a toggle. The error signal when the required output is known, is computed by a XOR gate. Another possibility is to remove the XOR gate altogether and feed an external error signal directly to the AND gates preceding the flipflops. No special initialization phase is required.

In this example, the binary input signal x,y corresponds to λ, b is κ, r is γ. $\pi()$ is implemented as an XOR and $\psi()$ is the and gate–flipflop combination. The decoder acts as an associative map to map the given input to any of the (four) stored combinations; in this case, all possible λ are stored. Another possibility, as output is binary, is to store only those λ for which γ is "on," but then one needs to make a search in memory, e.g., by a content-addressable memory (CAM).

LLA: A Critique. In applications where the input signal is high-dimensional and highly redundant and only a small number of output classes exist, LLA is not a good idea because of the n-to-2^n decoder. When n is the number of inputs, and k the number of data pairs stored, the utility of LLA is proportional to $k/2^n$. In the case of a logic function, k corresponds to cases where the output should be "on." The higher is this value, the more LLA becomes interesting. To alleviate the problem of combinatorial explosion, one would like to be able to cascade such structures each having only a small number of inputs. The problem remaining is to propagate the error back to internal layers. Alexander has significant work done related to using RAMs for pattern recognition; see (Alexander, 1990) for a recent review.

1.2. DECISION THEORY

When input λ have discrete values as in the previous case, one can classify by comparing a given input with all those stored. When a match is found, corresponding γ, i.e., class code, is given out. This approach may turn out not to be feasible if one of the following occurs:

[1] When the dimensionality of the input is high, the size of the memory required to store all possible λ,γ pairs grows exponentially—as seen in the LLA above.

[2] Due to noise, one may have input values different from stored patterns. Although differences may be small, this causes a non-match.

These two conditions force one to look for the *closest* stored pattern instead of an exact match. In such a case, one defines a set of *discriminant functions* $g_i(\lambda)$ for each class i where the value of this function denotes our confidence level that the input belongs to that class. One then chooses the most likely class.

$$g_i(\lambda) = \max_j (g_j(\lambda)) . \tag{1.8}$$

In the case of Bayes decision theory (Duda & Hart, 1973), maximum discriminant function corresponds to minimum conditional risk. When the loss terms are equal, this implies maximum a posteriori probability. When $P(\lambda|\gamma_i)$ is the a priori probability that input is λ given that input is of class i and $P(\gamma_i)$, the probability of class i, the a posteriori probability that input is of class i given that it is λ is given as:

$$g_i(\lambda) = P(\gamma_i|\lambda) = P(\lambda|\gamma_i)P(\gamma_i). \tag{1.9}$$

1.3. PARAMETRIC CLASSIFICATION

When the input vectors λ for a given class i are continous valued, mildly corrupted versions of a single prototype vector μ_i, i.e., due to noise, one can assume that patterns belonging to classes obey a multi-variate normal distribution. This is also what one would expect if features, i.e., components of λ, are chosen to extract those features that are different for different classes but as similar as possible for patterns belonging to the same class.

Case 1. The simplest case occurs when features, λ_j, are statistically independent and when each feature has the same variance, σ^2.[2] In this case, class clusters have the shapes of hyperspheres and therefore the probability that a certain pattern λ belongs to a certain class i decreases as distance, e.g., Euclidean, between λ and the class mean, μ_i increases. This is a minimum-distance classifier and the operation is called *template matching*. The boundaries between classes are hyperplanes that pass through medians (orthogonal to the line between the means). In case where a priori probabilities are not equal, one can add logarithm of this to shift the boundary away from the more probable mean.

Case 2. Another simple case occurs when the covariance matrices for all classes are equal. In this case, class densities are hyperellipsoids centered around the mean having equal sizes and shapes for all classes. One cannot use the Euclidean distance in this case, but should take into account the principal axes of the hyperellipsiods which are the eigenvectors of the covariance matrix. One, in this case, either needs the Mahalanobis distance or should perform a linear transformation that rotates and scales the axes so that hyperellipsoids become hyperspheres. The separating hyperplane is not orthogonal to the line between the means but is parallel to the principal axis of the density, i.e., that eigenvector of the covariance matrix having the largest eigenvalue.

Case 3. In the general case, classes have arbitrary shapes and discriminants have quadratic equations.

The above mentioned approaches are valid when our assumption about having normal densities are correct. They are called *parametric*, parameters being those of distributions, namely, mean, the elements of the covariance matrix, and the a priori class probabilities. In cases where these are not known or costly to compute, one looks for non-parametric methods.

---

[2] *This is for example the case when noise is added to binary images. The probability that noise, i.e., pixel inversion, occurs at a pixel, follows a Bernoulli distribution. The overall effect of noise over the image is a sum of independent Bernoulli distributions and thus follow a Binomial distribution. It follows from the "central limit theorem" (Ross, 1987) that for large images and small noise probabilities, this approximates a normal distribution. When n is the number of pixels in the image and p, probability that there is a pixel inversion, mean is np and variance is $np(1-p)$.*

1.4. NON-PARAMETRIC CLASSIFICATION

In non-parametric methods, the aim is to take samples (that form the training set) and use them most efficiently to approximate class separations. There are two approaches by which classes can be represented:

[1] Prototype based methods,
[2] Linear discriminant based methods

In the prototype based methods, associated with each class, one stores a set of prototypes which are also called exemplars or reference vectors, which are the typical members of that class. The criterion to decide whether a given input vector belongs to a certain class is than that of checking if its similar enough to any of its prototypes. In this approach, connection weights are the reference vectors themselves out of which the most similar to the input vector is sought.

In methods based on linear discriminants, one tries to find out the equations of the hypersurfaces that separate a class from all others. Whereas only linear separability can be achieved with one layer, one by having several of such layers with an embedded non-linearity can satisfy more complex criteria. In this approach, the connection weights give the factors of linear discriminant and the order of the hypersurface is a function of the number of non-linear layers.[3]

1.4.1. Prototype based methods

In the *nearest-neighbor* classification, each class is represented by a finite number of samples. A given input vector is compared with all samples using a certain metric, and is assigned to that class whose sample is the closest. To be more precise on borders, some kind of a voting scheme over the k nearest samples may be employed to give rise to *k-nearest-neighbor* methods.

As training sets tend to be big and redundant, one is usually tempted to summarize the empirical information given somewhat as to minimize the cost and operational speed of the classifier. One may, assuming hyperspheric clusters, compute the class means only. By having as many reference vectors as there are classes and performing a time average of labelled vectors of the training set, during a supervised process, one can find out class means (Oja, 1982). Trying to compute the variances also leads to the approach named *Parzen windows*.

The assumption of hyperspheric class shapes rarely hold; one then needs to approximate an arbitrary class shape as a sum of many small hyperspheres. This is called the *k-means* algorithm. A variant of the same approach is the *learning vector quantization* (LVQ) (Kohonen, 1988). In both the k-means and LVQ (and the more recent version LVQ2), the number of reference vectors need to be predefined. The reference vectors are updated during a supervised learning process. During recognition, as in the nearest-neighbor method, the closest reference vector is sought.

In the *restricted coulomb energy* (RCE) model (Reilly *et al.*, 1982), a variant of the Parzen windows, units have hyperspheric domination regions whose radii are defined by thresholds calibrated during a supervised learning process. The number of units is not fixed but they are allocated in an incremental manner when they are necessary.

[3] *The techniques quickly introduced in this section are explained in a more detailed manner with formal equations and using examples and are compared between themselves and the supervised algorithm proposed in this thesis. See section 2.7.*

1.4.2. Distance measures

These approaches require definition of an appropriate distance measure by which patterns can be compared. It should first be pointed out that the measure to be employed depends on the application and also concerns about implementation constraints like time and complexity.

When one wants to compare two vectors A and B, the simplest is to compute their correlation.

$$C(A,B) = \sum_i A_i B_i. \tag{1.10}$$

Correlation has the disadvantage that the result depends on the absolute magnitude of the vectors instead of only the difference between them. If the information lies not in the magnitude but the orientation of the vectors, one can compute the angle between two vectors:

$$\theta = arccos\left(\frac{\sum_i A_i B_i}{\sqrt{\sum_i A_i^2}\sqrt{\sum_i B_i^2}}\right). \tag{1.11}$$

One can use the Euclidean distance.

$$E(A,B) = \sqrt{\sum_i (A_i - B_i)^2}. \tag{1.12}$$

To reduce computational complexity, one can compare vectors in terms of the square of the Euclidean distance:

$$E^2(A,B) = ||A||^2 - 2C(A,B) + ||B||^2. \tag{1.13}$$

When one wants to compare a given vector, P, with a number of vectors, W_i, to choose the closest (for example in nearest-neighbor method where P is the input vector and W_i are the stored sample vectors):

$$E^2(P,W_i) = ||P||^2 - 2C(P,W_i) + ||W_i||^2. \tag{1.14}$$

$||P||^2$ can be dropped out as it is a constant term. Moreover, when W_i are normalized, i.e., all have the same magnitude, the $||W_i||^2$ term can also be ignored and one is left with a correlation.

1.4.3. Linear discriminant functions

Another possibility is to assume that the forms of the discriminant functions are known and look for the parameters of the discriminant function. The most popular case is when they are linear having the form:

$$g(\lambda) = \mathbf{w}^T \lambda + w_0. \tag{1.15}$$

where $\mathbf{w}$ is the *weight vector* and w_0 is the *threshold*. This hyperplane divides the input space defined by the elements of λ into two halfspaces: Those for which $g(\lambda)$ is greater than 0 and those for which it is less than 0. The weight vector is orthogonal to the hyperplane, thus defines its orientation, and the threshold defines the location of the hyperplace with respect to the origin.

The parameters of the discriminant, i.e., $\mathbf{w}$ and w_0, can be determined by an iterative gradient-descent process. In the *perceptron criterion function* (Duda & Hart, 1973), the error is the sum of the distances from the misclassified samples to the decision boundary. In the *LMS rule* (Widrow & Hoff, 1960), the error criterion is the sum of the squared error. In a gradient-descent procedure, during an iterative process, one moves some distance from current $\mathbf{w}$(w_0 also included) in the direction of steepest-descent, i.e., along the negative of the gradient.

When classes are *linearly separable*, the discriminant function of classes can be defined as linear discriminants which can be implemented as a one-layer network. One in this approach tries to define $g_i(\lambda)$ for class i such that:

$$g_i(\lambda) = \begin{cases} > 0, & \text{if } \lambda \text{ belongs to class } i; \\ < 0, & \text{otherwise.} \end{cases} \tag{1.16}$$

When this condition is not satisfied, one needs more complicated decision boundaries than hyperplanes.

[1] One can try to separate classes in a *pairwise* fashion by hyperplanes. With c classes, one needs $c(c-1)/2$ hyperplanes, one for every pair. One then needs another layer of c units to conjunct the results of pairwise separation (Duda & Hart, 1973) (Knerr *et al.*, 1989).

[2] One can add additional terms involving the product of pairs of components to obtain *quadratic discriminant functions*. In the neural network jargon, units implementing such products are called *high-order* or *pi* units (Rumelhart *et al.*, 1986) (Personnaz *et al.*, 1987).

[3] One can have multiple layers of linear discriminants to be able to thus define *hypersurfaces* of unlimited forms. To be able to implement gradient-descent in such networks, the non-linear transfer function should be a differentiable function instead of a thresholding (le Cun, 1985) (Rumelhart *et al.*, 1986).

1.5. QUANTIFYING SUCCESS IN CLASSIFICATION

When a system performs a classification, there are five possibilities:

[1] There is a *success* if the class found by the system is the correct class:

$$\kappa \neq REJECT \wedge \gamma \neq REJECT \wedge \kappa = \gamma \Rightarrow success. \tag{1.17}$$

[2] There is an *erroneous reject* if the system responds with a reject where in fact a class is associated.

$$\kappa = REJECT \wedge \gamma \neq REJECT \Rightarrow ErroneousReject. \tag{1.18}$$

[3] If the system responds with a reject when there is no class associated, this is a normal reject.

$$\kappa = REJECT \wedge \gamma = REJECT \Rightarrow GoodReject. \tag{1.19}$$

[4] If the system assigns it to a class when in fact there is no class associated, this is a nuisance; whether it is considered an error depends on the task. Normally such a situtation will not be encountered.

$$\kappa \neq REJECT \wedge \gamma = REJECT \Rightarrow Nuisance. \tag{1.20}$$

[5] Finally there is an error, if the system responds with a class and that class is not the correct one.

$$\kappa \neq REJECT \wedge \gamma \neq REJECT \wedge \kappa \neq \gamma \Rightarrow Error. \tag{1.21}$$

Throughout this work success, error, and reject are quantified where reject corresponds to the erroneous reject case.

1.6. QUALITY ASSESSMENT

The quality of a network solution is given as:

$$Quality = A * NET_SIZE + B * SWP_TIME * NO_SWP + C * SUC_TEST. \tag{1.22}$$

where

$$A < 0, B < 0, C > 0$$

NET_SIZE is the size of the network which is computed as the size of the memory (in bits) required to store the connection weights; we want to minimize this. SWP_TIME is the time it takes to perform one learning sweep over the training set and NO_SWP is the number of sweeps it takes to learn the training set. The total learning time is $SWP_TIME * NO_SWP$ which we also want to minimize. SUC_TEST is the success on test set that we want to maximize. The application determines which of these three constraints are more important and surely then one network or learning algorithm may be superior to another for one task while worse for another task.

2

Incremental Supervised Learning

It is the object of science to replace, or save*, experiences, by the reproduction and anticipation of facts in thought. Memory is handier than experience and often answers the same purpose.*

In the reproduction of facts in thought, we never reproduce the facts in full, but only that side of them which is important to us, moved to this directly or indirectly by a practical interest. Our reproductions are invariably abstractions.

Sensations are not signs of things; but, on the contrary, a thing is a thought-symbol for a compound sensation of relative fixedness. Properly speaking, the world is not composed of "things" as its elements, but of colors, tones, pressures, spaces, times, in short what we ordinarily call individual sensations.

In the reproduction of facts, we begin with the more durable and familiar compounds, and supplement these later with the unusual by way of corrections. All judgments are such amplifications and corrections of ideas already admitted.

— *Ernst Mach, "The Economy of Science,"* The Science of Mechanics.

2.1. LEARNING AND ADAPTATION

In the previous chapter, I have mentioned the difference between structural and parametric modification. One may arrive to the conclusion that although structural modification (SM) sounds better, it is not feasible to implement in a real world environment, and that parametric modification (PM) is enough. In this and the following chapter, I will discuss two algorithms for supervised and unsupervised learning, both based on SM. In this introductory section, I would like to make the point about relative merits and drawbacks of these two types of modification. Basically, what I will be stressing is the difference between *learning* and *adaptation*, which terms I have used in an interchangeable manner in the previous chapter.

The task we have assigned ourselves is to design a system that, when given examples, will be able to modify its functioning so that, according to a certain criterion of goodness defined, it will become better. Thus, according to the application, we are given the λ, γ pairs and the $\pi()$, and are asked to define σ' that minimizes η—which variables we have defined in the first chapter. To repeat, σ has two parts: those that are modifiable, i.e., parameters, and those that are not. If one wants to perform PM only, one should make sure that the modifiable parts really cover all that may need to be modified. In other words, the unmodifiable parts should be general enough so as to be able to handle all cases, so that achievable σ can always fulfill requirements imposed by the environment.

Turing machine for example, is such a general machine that by loading the tape, i.e., parameters, with different instructions and data, one can perform any computation. Computers are descendants of Turing machines that are more specialized on arithmetic and logic.

As I define the terms for my own usage from now on, learning is finding σ' from no predefined initial structure, and adaptation is finding a $\sigma(t+\tau)$ from $\sigma(t)$; *learning is finding out the required structure from scratch, adaptation is iterative modification of the structure.*

Learning then is clearly SM as initially there is no structure at all. Adaptation can partially be accomplished by PM except in certain cases. As an example let us take curve-fitting. If we know that the data is of second order, we define our structure as $y = cx^2 + bx + a$ and by PM, find out the best values of constant factors like a, b, and c that minimize the difference between the actual values given in the training set and values estimated by the fitted function. If we do not have any idea of the order however, we need SM. We also need SM if our idea of the order is wrong. There are two intertwined problems here; first the order of the polynomial needs to be determined and second, the values of the constant factors should be computed. One approach to carry this out is an *incremental* one by small modifications of the structure.

A computer program cannot be more intelligent than its programmer and should not be more intelligent than its user. Similarly a learning system cannot be more intelligent than its training set and should not be more sophisticated than what its test set necessitates.

As we want to solve a problem as simply as possible, in curve fitting, we favor lower-order polynomials. During learning, we start from the lowest-order and check if we can make a good fit with it. If we cannot, we increase the order by one, and re-try, till we get an error less than a certain tolerance. Hence, *one starts from the simplest description, and passes to more complex descriptions if and when necessary.* It should be stressed that one should always try to look for the *simplest* explanation, otherwise one cannot generalize but risks to specialize too much on the training data. That is why, passing from one structural level to another should be done rather reluctantly. A lower-order function that gives little error should be preferred to a high-order perfect fit (Fig. 1).

In curve fitting, we start with $y = a$ and check for the a value that minimizes error. If error is so big that this cannot be taken as a good fit, we add one more term to get $y = bx + a$ and now look for suitable a and b values, etc.

Passing from one level of complexity to another is SM and determining values of parameters like a, b that minimizes error is PM. One works at a certain level and does one's best at that level; if that is still not sufficient, one passes to the next level of complexity or sophistication (in a manner similar to Kuhn's "paradigm shifts").

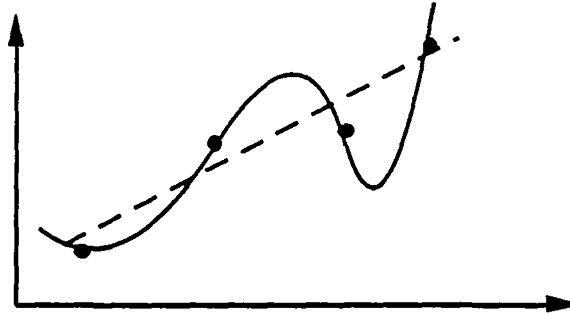

Fig. 1. Two polynomials of different orders are fitted on the same set of points. Lower order deviates more but is to be preferred to the higher order one.

What do these have got to do with neural networks ?

Neural network algorithms perform parametrical modification which, when—as they are generally—based on gradient-descent, require a lot iterations. One pre-defines the network structure, i.e., the number of hidden layers, units, and connectivities. Learning algorithms like back-propagation modify the connection weights *only*. This is clearly PM, and thus is adaptation.

Solution of a problem with a neural network similarly requires, to the first, determination of the network structure, and to the second, computation of the synaptic weights.

Small changes in the environment can be compensated for by a modification of some parameter values, i.e., synaptic weights, but in learning from scratch, the structure of the network should also be determined. Learning by changing connection weights only is time-consuming and does not always work; freedom to modify the network structure is also needed.

It was noted before (Valiant, 1984) (Baum, 1989) that learning algorithms where units and links may also be added may lead to shorter learning times and better generalization. Such algorithms are called *incremental*. In this chapter, I will discuss how one can learn categories incrementally.

2.2. DESIGNING A SUPERVISED SM NETWORK

One should first make sure that the network can learn any discriminant function. This requires at least a three-layer network of threshold units (Lippman, 1987) (Pao, 1989), or something as powerful. The complexity of the mapping ability of such a network depends also on the number of its hidden units. Following the motto of "starting from simple and making more complex when necessary," one would like to start with a small number of units and add when necessary.

However, we cannot do this using, say, back-propagation, or any other learning algorithm that distributes associations on a large number of *shared* connections. In such cases, because the connections are shared and because of the non-linearity, one cannot just add units without disturbing old associations. There are two possibilities:

[1] If one can make sure that when this new unit gets activated, none of the ancient units get activated, there will be no problem. The units thus should somehow be able to suppress other ones when they get control. This implies a competitive strategy and a local representation.

[2] The other possibility is to divide the network into separately trained subnetworks where such subnetworks can be added in an incremental fashion. One needs a certain mechanism whereby addition of a new subnet improves capability instead of corrupting the harmony as one would normally expect. One approach is to have subnets that have different conceptual interpretations where the output of one has priority over another, i.e., a competition between subnets. Another approach is to have each subnet as another hidden layer.

2.2.1. A historical view of supervised incremental learning

This section gives a list of the incremental learning algorithms known to me. They are also classified to belong whether to [1] or [2] above. I will discuss these algorithms in greater detail in a later section after Grow-and-Learn (GAL) is explained.

- The first incremental neural network-based learning algorithm I know of is the Restricted Coulomb Energy (RCE) model proposed by Reilly, Cooper, and Elbaum (1982). This algorithm uses [1]. Associated with each class are a number of prototypes. Each prototype unit performs a dot product and thresholds the activation. This defines a hypersphere of domination around each unit whose radius, which is a function of the threshold, may get modified iteratively during learning. Thus, the space is divided into zones dominated by prototype units.
- The very first version of Grow-and-Learn (GAL) (Alpaydın, 1988) [1] was based on [1]. First, the effect of the already existing units were calculated. If the existing units could not perform the required association, then, a new unit was added. This unit's synaptic weight vector was set equal to the input vector so that it would be the unit that would be activated in that case. The weights from this newly added unit to class units were computed to be just a little bit bigger to undo the effect of the others. Learning was one-shot, i.e., only one iteration was needed to learn an association, but for complicated training sets, many units were allocated unnecessarily.
- One approach is based on [2] but units of somewhat different conceptual interpretations are used (Knerr *et al.*, 1989). In this method, one first tries to train a one-layer network with output units corresponding to classes by the Perceptron learning algorithm, assuming that classes are linearly separable. For the classes which this condition is not satisfied, one adds a subnet and tries to mutually separate classes by adding units to check for these conditions. For cases where this does not work either, one performs a piecewise approximation of boundaries using logical functions using additional subnets.
- Another approach which is also based on [2] is to add the unit always as a new hidden layer and train the connections of this last layer only (Mézard, 1989).
- The Probabilistic Neural Network by Specht (1990) involves assigning units to store each vector in the training set. However, one can hardly call it incremental as all vectors in the training set are stored.
- The Piecewise Linear Network of Kong and Noetzel (1990) is an incremental variant of LVQ. The difference being that instead of having a fixed number of units, when an input vector significantly different from a stored one is seen, it is added as a new unit. The winner-take-all that seeks the closest implies this of type [1].

I will compare in a later section approaches [1] and [2] in greater detail and list their merits and drawbacks.

[1] *The RCE model was not known to me when I proposed GAL.*

2.3. GROW-AND-LEARN (GAL)

We have seen in the previous chapter when we discussed discriminants that the biggest problem one encounters when one wants to learn classes is to approximate the shape of classes' distributions. When the distribution is normal and variances on all dimensions are equal, just storing the mean is sufficient as classes have hyperspheric shapes. When it is not, many exemplars of that class need to be stored. In this way, we define the class distribution as a sum of many hyperspheres. How many of them will be necessary is not known, that is why algorithms similar to *k-means*, e.g., Learning Vector Quantization (LVQ) of Kohonen (1988), cannot guarantee finding the best solution as the number of codebook vectors is pre-determined. Of course, one can always store all the vectors in the training set and perform nearest or k-nearest neighbor search. This is a rich man's solution, as training sets are big. One wants to minimize the memory and complexity by somehow summarizing the salient characteristics of the distribution.

Another factor that one wants to minimize, additional to network size, is the time it takes the network to learn the content of the training set. Iterative algorithms based on gradient-descent, par excellence, require many iterations which, for real problems, are counted using multiples of tens of hours. That is why, with iterative algorithms, *off-line learning* is employed.

Another reason for this, except speed, is that, algorithms where associations are distributed over a set of connections during learning, need to be introduced vectors in an unbiased fashion, showing all the categories with their real densities; one cannot guarantee this in a real working environment.

With back-propagation for example, one cannot just perform a number of iterations with one association only to be able to add it to system's memory; as weights are distributed, the network needs to see the whole set In an on-line system, one does not have time to re-learn the whole training set when a new association is to be added. Thus iterative algorithms cannot learn on-line. To be able to learn on-line, two conditions should be satisfied:

(1) Adding a new association should be done very quickly.
(2) Addition of a new association should not disturb associations unrelated.

What one wants, is to be able to extend class definitions if need arises. Grow-and-Learn (GAL) is based on this idea (Alpaydın, 1990b). GAL is an incremental algorithm for supervised learning. The network grows when it learns category definitions, thus the name.

2.3.1. Network structure

A GAL network has the structure seen in Fig. 2. The first layer is the layer of input units which may be binary or analog depending on the input representation. The second layer is the layer of exemplar units and the third is the layer of class units. A class may have more than one exemplar. W_{ie} and T_{ec} denote the connections from input unit i to exemplar unit e and from exemplar unit e to class unit c respectively. W_{ie} values depend on the input representation. T_{ec} are 1 or 0 depending on whether e is an exemplar of class c or not. When P is the input vector, the activation of an exemplar unit, A_e, is the distance between P and the weight vector of unit e, that is W_e^T, computed using a suitable metric, denoted here $D()$. Assume that $D()$ is normalized such that $D(A,A)$ is 0 and $D(A,B)$ increases as A and B get further distant. A winner-take-all layer then chooses the closest, i.e., minimum.

$$A_e = D(P, W_e^T).$$

$$E_e = \begin{cases} 1, & \text{if } A_e = \min_i(A_i); \\ 0, & \text{otherwise.} \end{cases} \tag{2.1}$$

The winner-take-all guarantees that there will be only one unit activated. The unit whose weight vector is closest to the input vector will be the only active unit. For the class layer the response of a class unit is calculated as below after which there will be exactly one active class unit:

$$C_c = \sum_e E_e * T_{ec}. \tag{2.2}$$

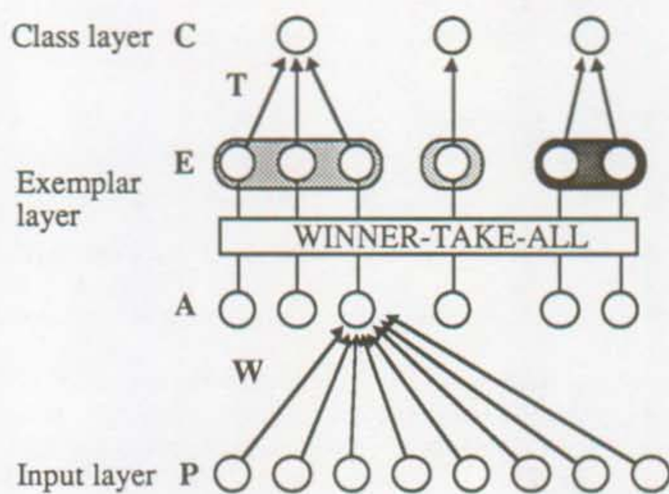

Fig. 2. GAL network structure.

2.3.2. Learning in GAL

Initially there are no exemplar units nor any class units. Learning proceeds as follows (Fig. 3):

When a new input P is given as a member of a certain class c, the system first checks if class c already exists. If not, a new class unit is created and labelled as c. An exemplar unit, e, is created which currently is the only exemplar of that class. Connections are set as follows:

$$\forall i,\ W_{ie} = P_i.$$

$$\forall o,\ T_{eo} = \begin{cases} 1, & \text{if } o = c; \\ 0, & \text{otherwise.} \end{cases} \tag{2.3}$$

If class c already exists, then P is given as input to the network and response of the network, r, is calculated as in (2.1) and (2.2). If the class found by the network, r, is the same as the class desired, c, no modification is done. If it is not, a new exemplar e is created and the W and T connections are set as in (2.3).

The winner-take-all-type non-linearity effectively divides the space into regions of domination among the exemplar units by hyperplanes which pass through points that are of equal distances to two exemplar units (based on the metric used). An exemplar unit's domination region includes all the points in the training set for which that exemplar is the closest. A class's domination region is the sum of its exemplars' domination regions.

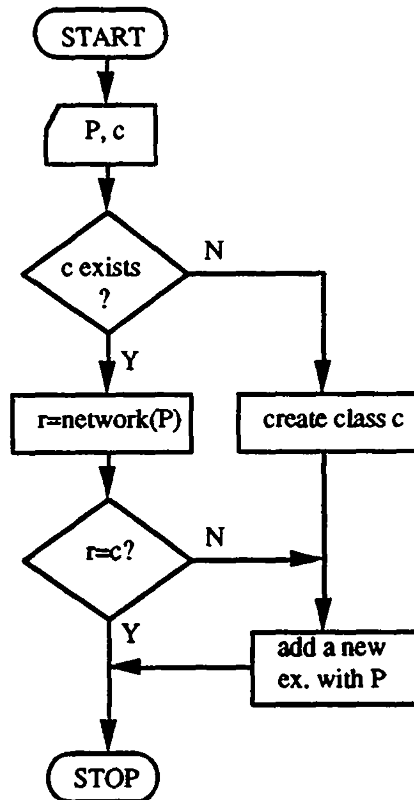

Fig. 3. Flowchart of a learning iteration in GAL. P is the input vector, c is the desired class for P and r is the class found by the network.

2.4. GAL: A DIDACTIC EXAMPLE

As an example to show how GAL works, a two-dimensional input signal is chosen so that the input space and the discriminants can be easily displayed. The simulation program is in C, runs under UNIX on a Sun 4 workstation. Classes are coded using different texture patterns, white implies that no class is associated. Each dimension is discretized into 100 levels; out of 10,000 possibilities, 7,074 are associated with classes (Fig. 4).

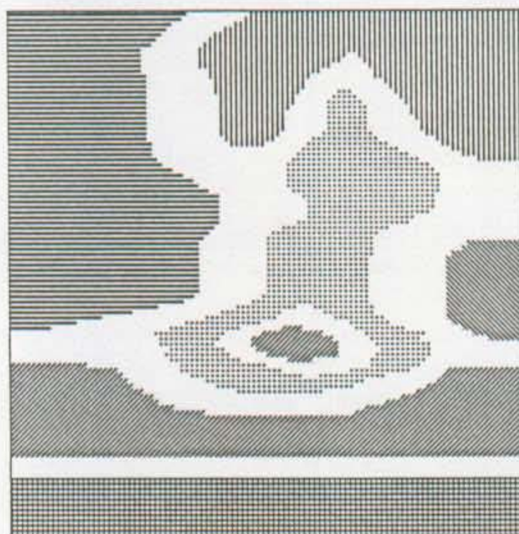

Fig. 4. Input space used as example. Different textures denote different classes, white implies that that point is not a valid input with which no class is associated.

At each iteration, a point is chosen at random using a uniform distribution and if it is associated with a class (not all white), given as input to GAL. The input vector has two dimensions: the x and y coordinates between 0.0 and 1.0. Euclidean distance is used as the metric.

To test GAL, all the 10,000 points are sequentially given as input and the response of the network is displayed. The maximum error is also 7,074.

The network evolves as seen in the following figures. The black lines on the left represent the class boundaries currently known by the network. The black points on the right correspond to the position of exemplar units.

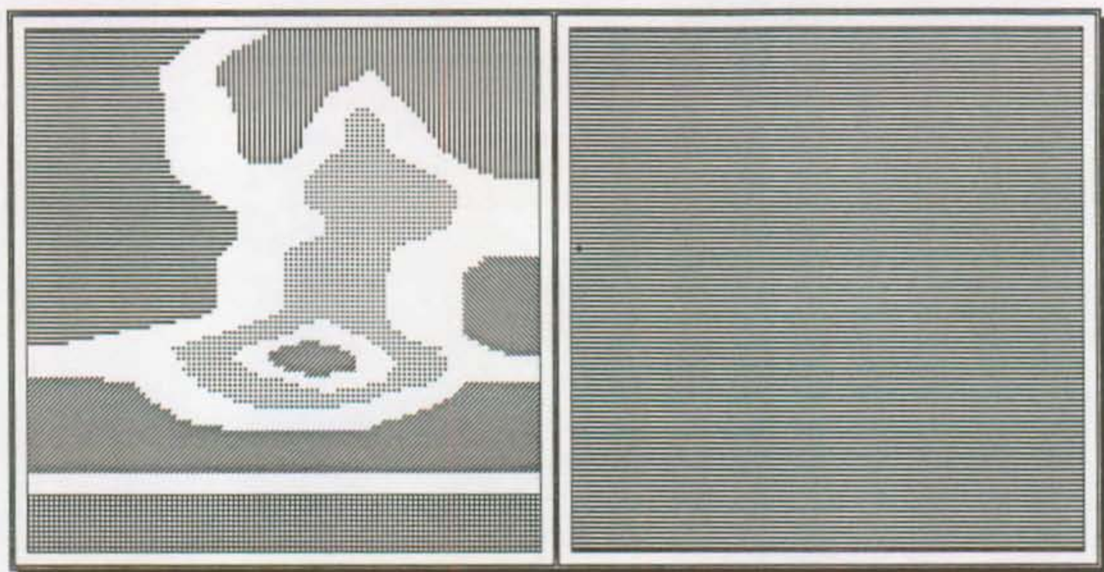

Fig. 5. After 1 iteration, 1 exemplar unit is stored and the error is 5158. To the right is network's current knowledge. The black point shows the position of the exemplar. As only one class is seen by the system, the system knows nothing except that class.

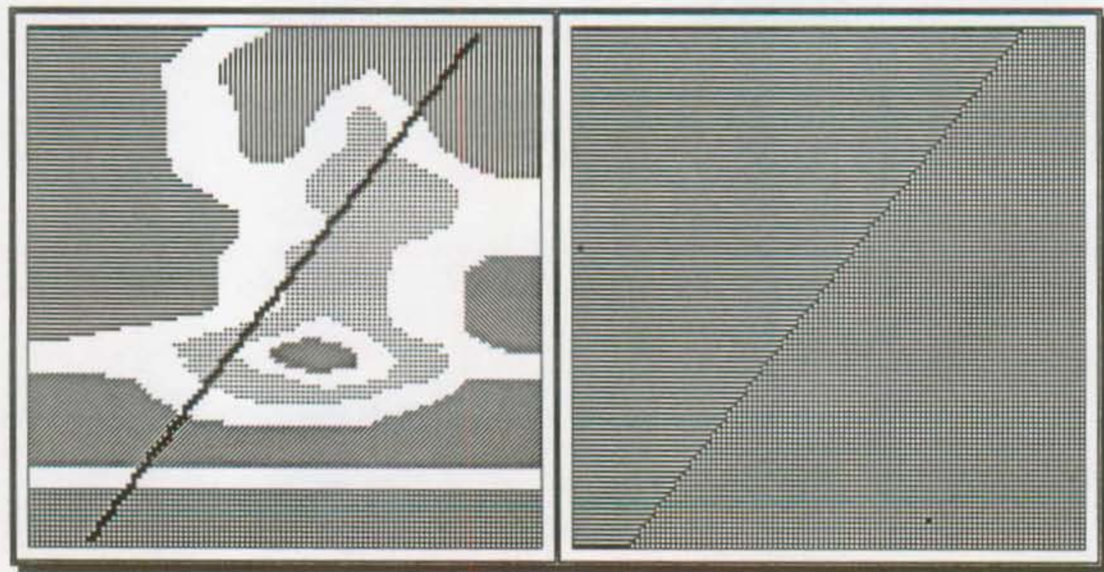

Fig. 6. After 2 iterations, 2 exemplar units are stored, and the error is 4222. The black line to the left is the separating hyperplane.

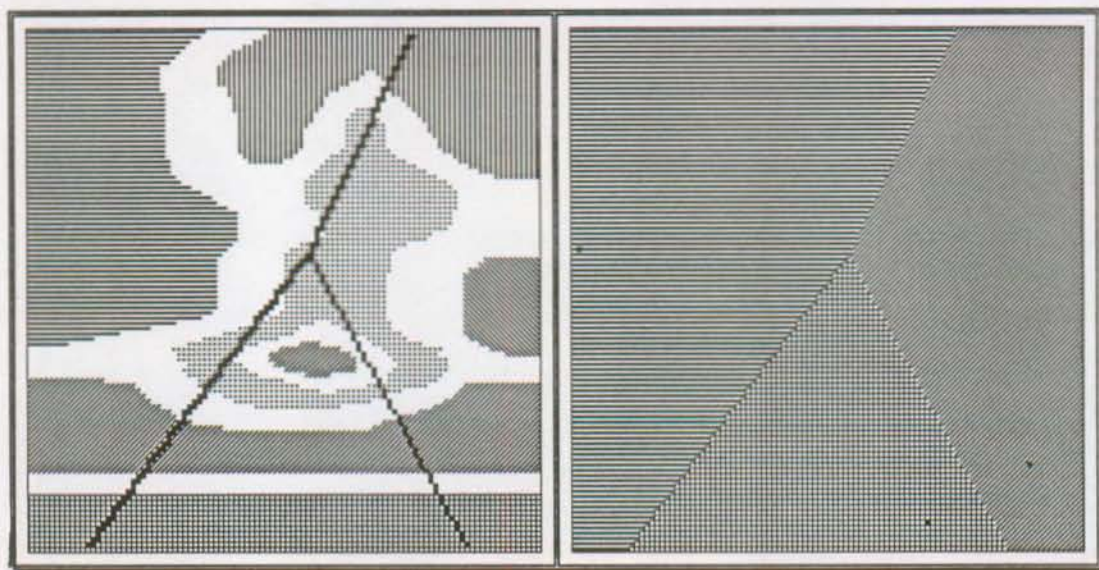

Fig. 7. After 3 iterations, 3 exemplar units are stored, and the error is 4020.

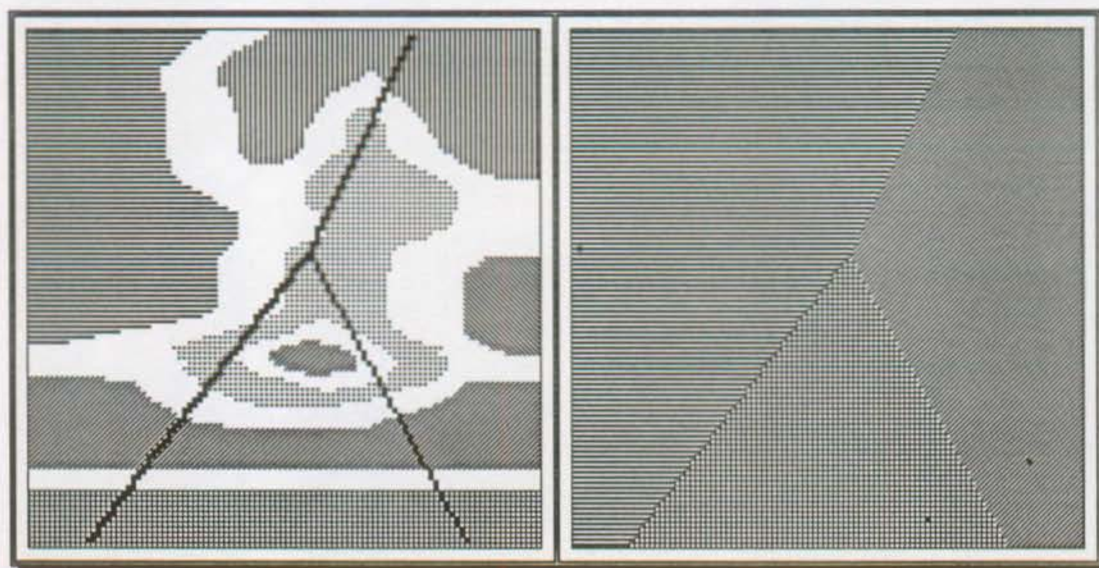

Fig. 8. After 4 iterations, 3 exemplar units are stored, and the error is 4020. The input encountered in this fourth iteration is already correctly classified, so another exemplar is not added.

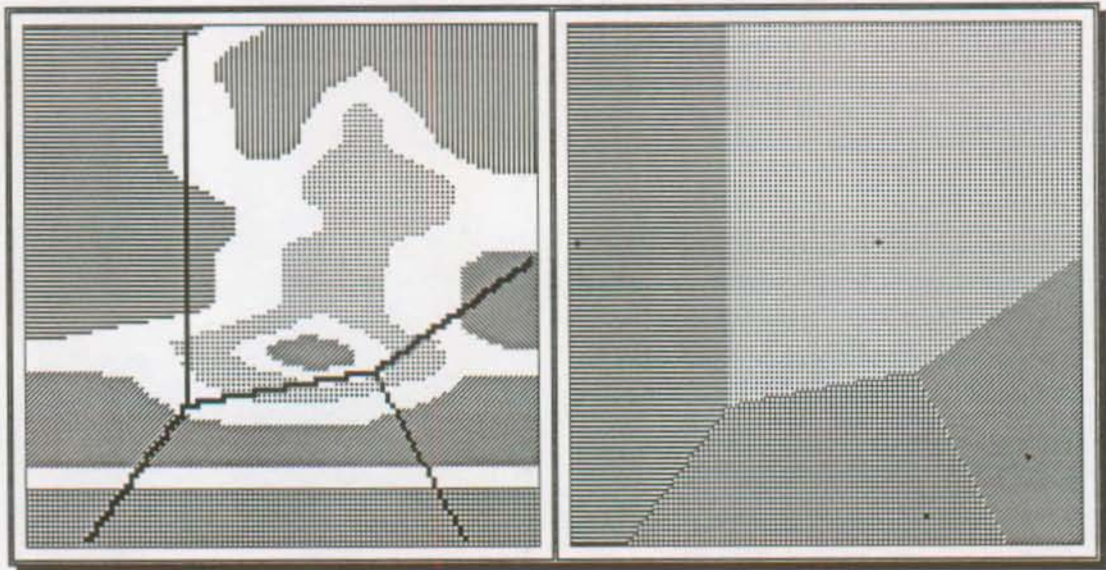

Fig. 9. After 5 iterations, 4 exemplar units are stored, and the error is 3141.

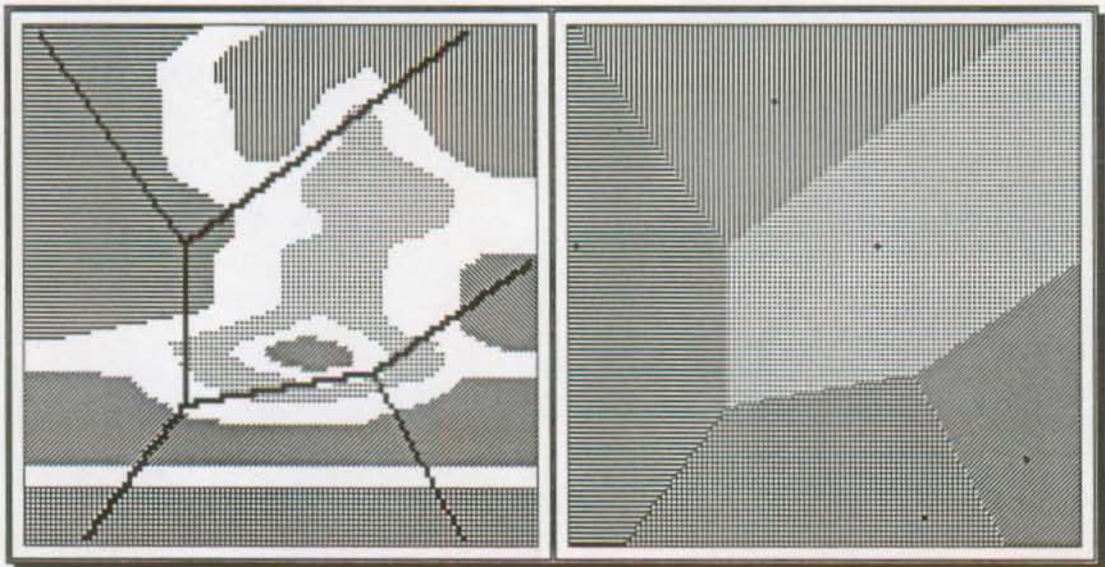

Fig. 10. After 10 iterations, 5 exemplar units are stored, and the error is 3026.

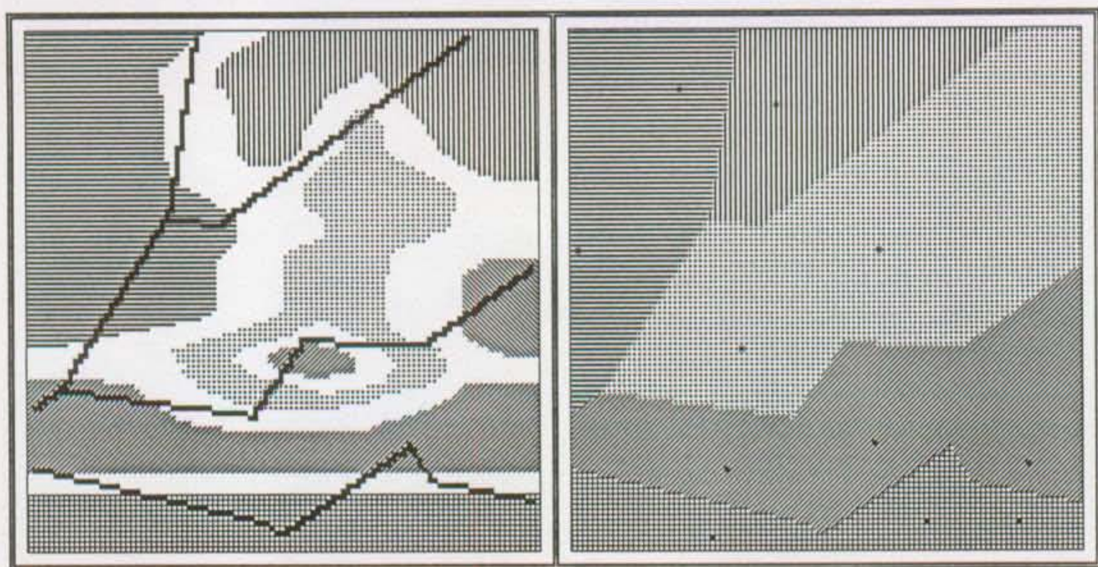

Fig. 11. After 20 iterations, 11 exemplar units are stored, and the error is 1742. Note that some classes have multiple exemplars.

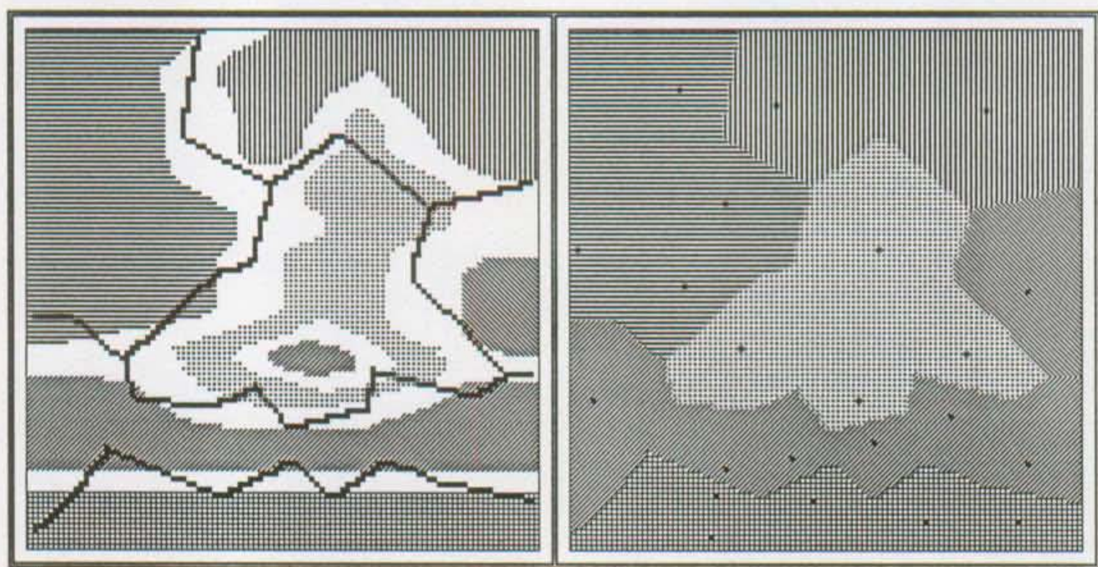

Fig. 12. After 100 iterations, 22 exemplar units are stored, and the error is 499.

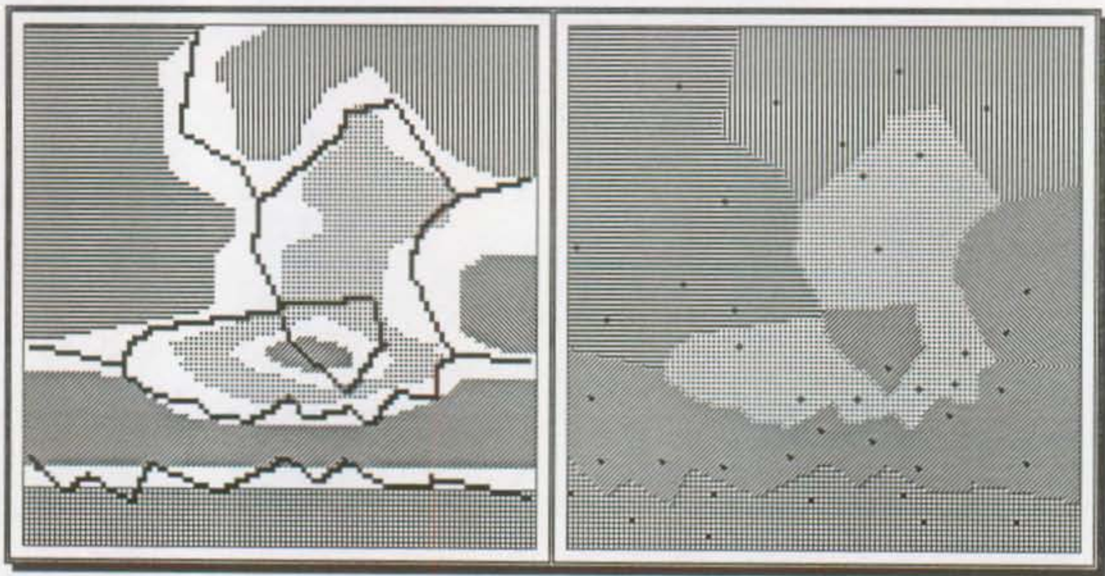

Fig. 13. After 500 iterations, 41 exemplar units are stored, and the error is 190.

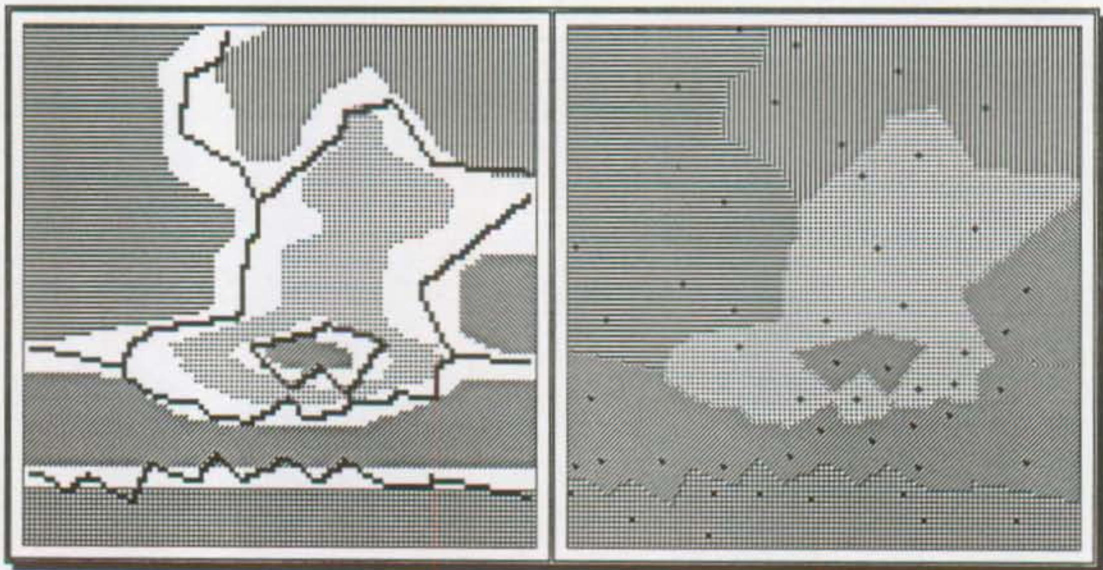

Fig. 14. After 1000 iterations, 50 exemplar units are stored, and the error is 62.

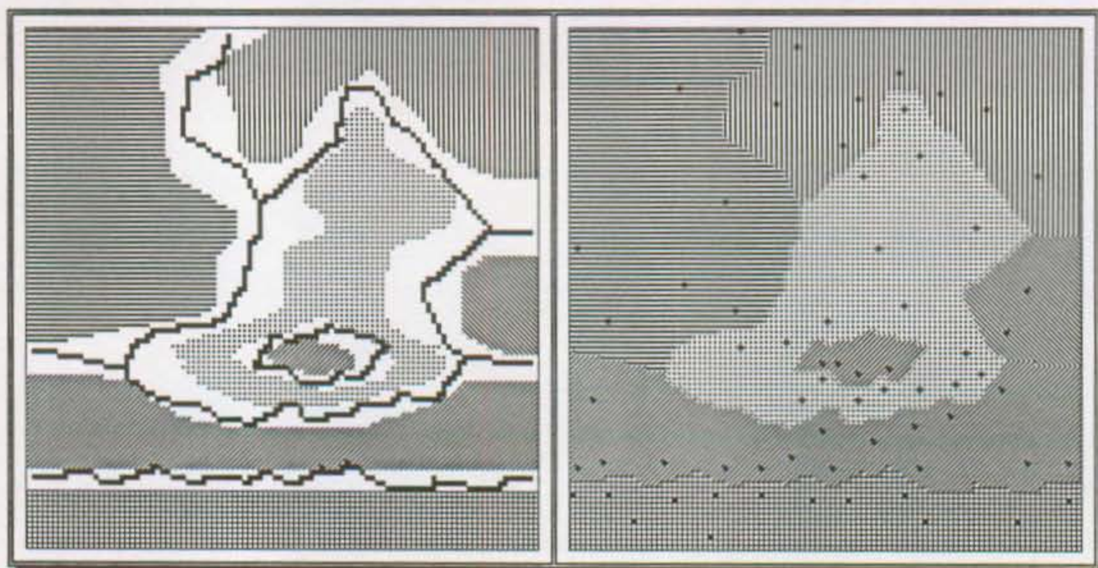

Fig. 15. After 5000 iterations, 67 exemplar units are stored, and the error is 4.

Notice first that the class forms are not important and that convex or concave classes are learned. Note also that when discriminants are low-order, e.g., linear, GAL tries to approximate this in a piecewise manner using a lot of units.

In Fig. 16, the number of exemplar units, i.e., memory required, as a function of the learning iterations made is drawn; as seen, it saturates.

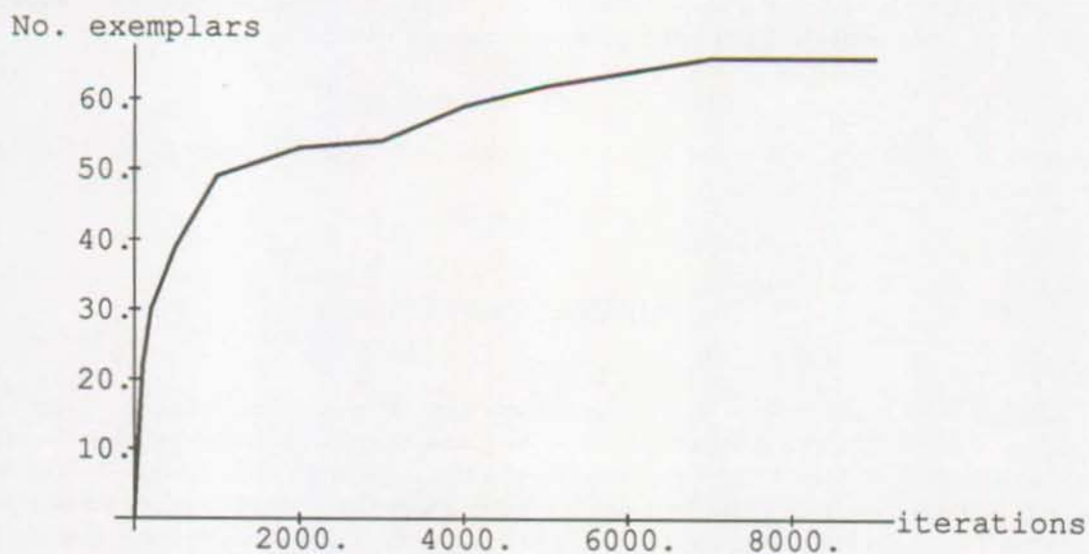

Fig. 16. The number of exemplar units vs. the number of training iterations made.

In Fig. 17, error on test set as a function of the number of training iterations made, is drawn.

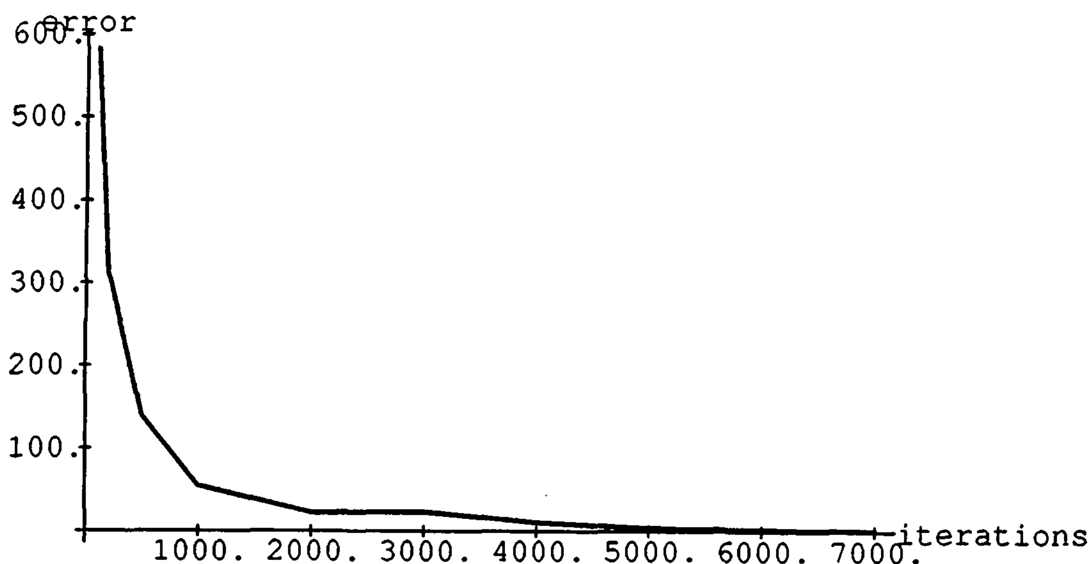

Fig. 17. Error on test set vs. number of learning iterations.

The size of a GAL network is computed as follows (I, E, and C are the number of input, exemplar, and class units respectively. P is the number of bits required to represent the value of an input unit. E to C connections are always binary.):

$$size\ of\ GAL\ net = I * E * P + E * C\ bits. \tag{2.4}$$

For this example, each unit's value is encoded using 7 bits to be able to represent values between 0 and 100. With 67 exemplar units and 6 classes, the network size is:

$$2 * 67 * 7 + 67 * 6 = 1340\ bits.$$

This value is computed so that it can be compared with the size of the memory required for other approaches.

2.5. FORGETTING IN GAL

The algorithm tends to store those input patterns that are closest to the boundaries for finer separation of classes. In such a learning scheme, the actual exemplars stored depend on the order of encountering the vectors. A unit previously stored when an exemplar that is nearer to boundary is stored, becomes useless (Fig. 18). Such exemplar units, as now they are in the domination region of another unit of the same class, are useless and may be eliminated to decrease network size.

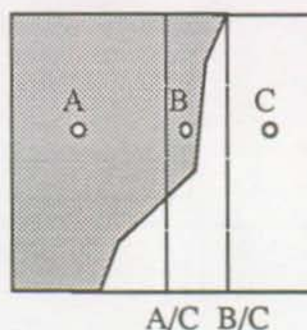

Fig. 18. A and B are associated with the first class and C is associated with the second class. If A is encountered before B, B also needs to be stored; if B is stored first, A is not stored.

To accomplish this, in a so-called *sleep* mode, the following procedure is employed (Fig. 19).

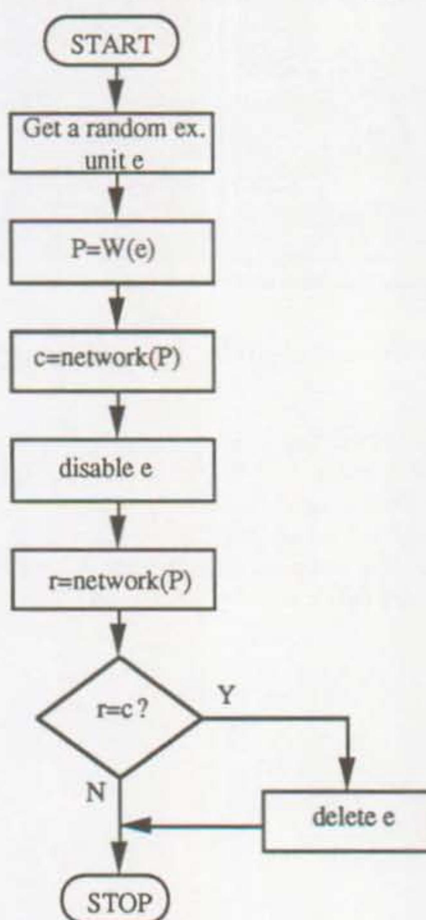

Fig. 19. Flowchart of forgetting during sleep in GAL.

The system is closed to the environment and an exemplar unit e is chosen at random. The input vector is set equal to e's synaptic weight vector.

$$\forall i, \; P_i = W_{ie}. \tag{2.5}$$

The response of the system with this exemplar unit, c, is computed. Then, e is disabled and the system response without e is also computed, r. If r and c are the same, then e is deleted from

the network. Otherwise, it is kept as its elimination will cause a wrong shift of the class boundaries. Note that such an elimination strategy may cause error to increase as we cannot make sure that all the points lying in the domination region of e is dominated by other exemplars of the same class.

When a *sleep* pass is applied, 27 units out of 67 are eliminated but the error goes up to 146 from 4. One can see how the exemplars inside the regions are effectively removed (Fig. 20).

Fig. 20. Following Fig. 15, after sleep, 40 exemplar units are stored with error 146.

To remedy error that may be introduced by *sleep*, one uses alternating *awake* and *sleep* modes. The best strategy I have arrived to is to have *awake* iterations till I get 100% on the training set, and then perform a *sleep*. This is alternated, and finally, GAL settles down to a set of exemplar units where no longer additions are possible because one has already 100%, and no more eliminations are possible because all the units are near the class boundaries. When I switched to *awake* mode and performed 5,000 more iterations, the result is shown in Fig. 21.

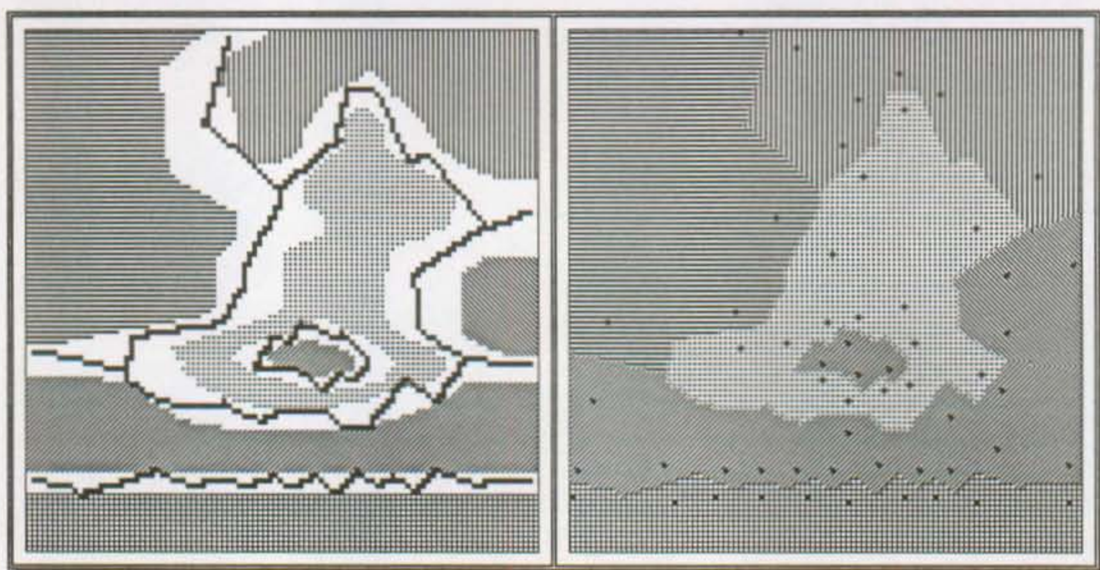

Fig. 21. After 10027 iterations, 57 exemplar units are stored, and the error is 6.

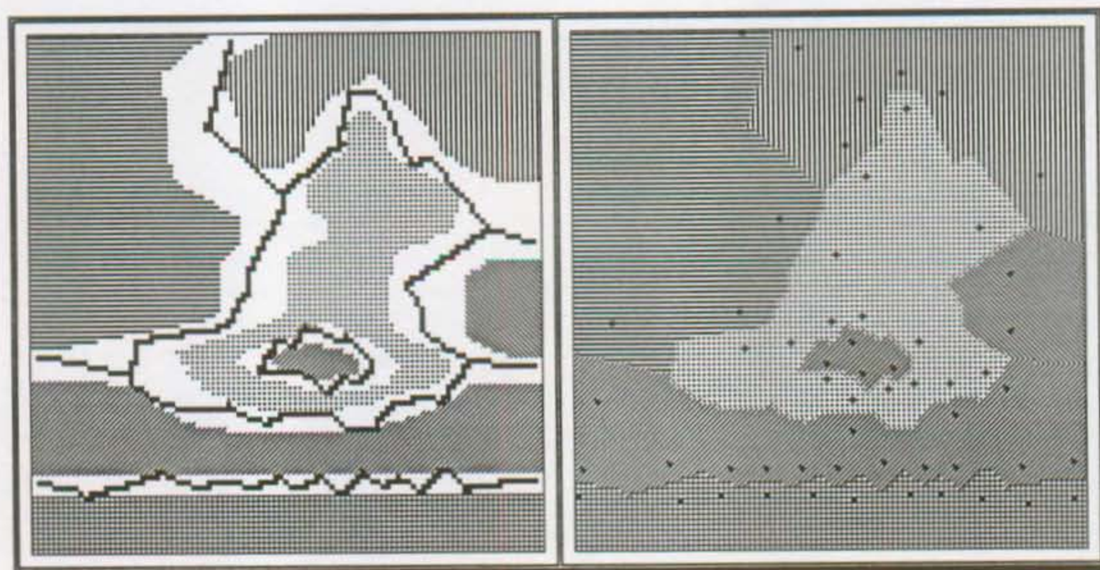

Fig. 22. After 12030 iterations, 57 exemplar units are stored, and the error is 3. This is after one more sleep pass where 3 units are eliminated and 2,000 more awake iterations where 3 more units are added.

Sleep mode thus allows having a network with 10 less units (around 15 %) while not increasing the error rate but of course at the expense of more iterations. The network size is then (using equation (2.4)):

$$2 * 57 * 7 + 57 * 6 = 1140 \ bits.$$

Comparison with the case without reject will be given afterwards so one can judge whether the gain in memory is worth the time one loses during *sleep*.

2.6. GAL WITH REJECT

GAL as explained hitherto, does not reject, i.e., refuse to classify an input saying "I don't know." This in a real-world application is necessary.[2] It is better to refuse to classify if the input is not similar enough to any of the known classes, than just choosing the most similar. The way I propose to handle this is to check the winner and the next highest activated exemplar and refuse to classify if they belong to different classes and if there is no great difference between their activations. If i is the exemplar with the highest activation, and j is the next highest:

$$A_i = \max_e(A_e).$$

$$A_j = \max_{e \neq i}(A_e).$$

one rejects if:

$$class(i) \neq class(j) \ AND \ |A_i - A_j| < \theta_{LIM} \tag{2.6}$$

where θ_{LIM} specifies how large the reject region will be.

After two iterations with two exemplar units, when θ_{LIM} is 0.05, the domination regions differ as seen in figure Fig. 23. As seen, instead of having a separating hyperplane between two exemplars, one now has a region symmetric around it on both sides containing those points that are not much closer to one exemplar with respect to the other.

When the example given above is attempted with this approach, the following results are achieved (Fig. 24):

[2] *To say that you know when you know and to say that you do not know when you do not know, that is knowledge. — Confucius, The Analects.*

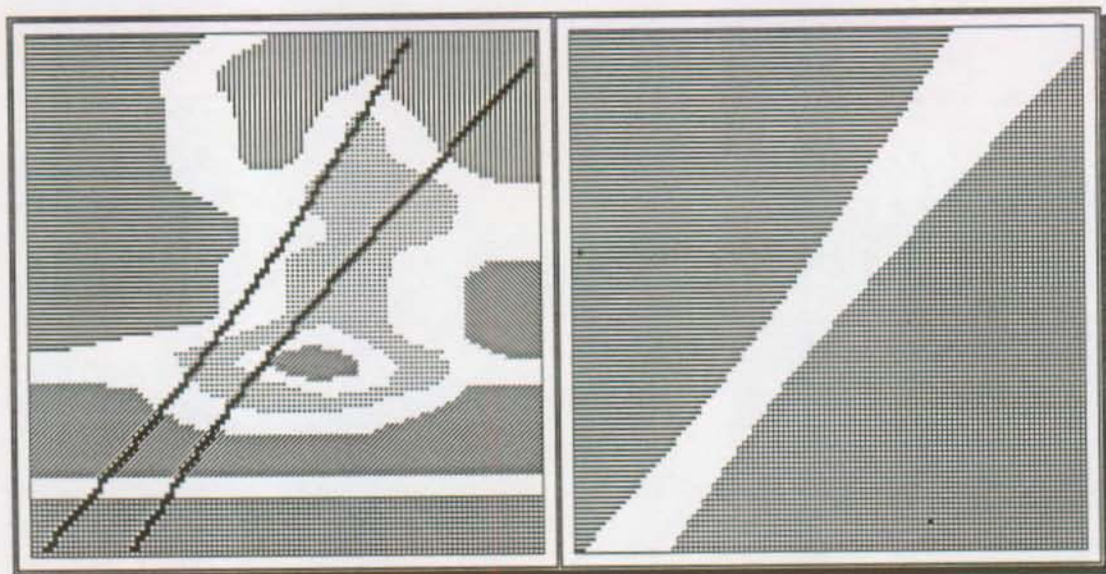

Fig. 23. θ_{LIM} *is 0.05*

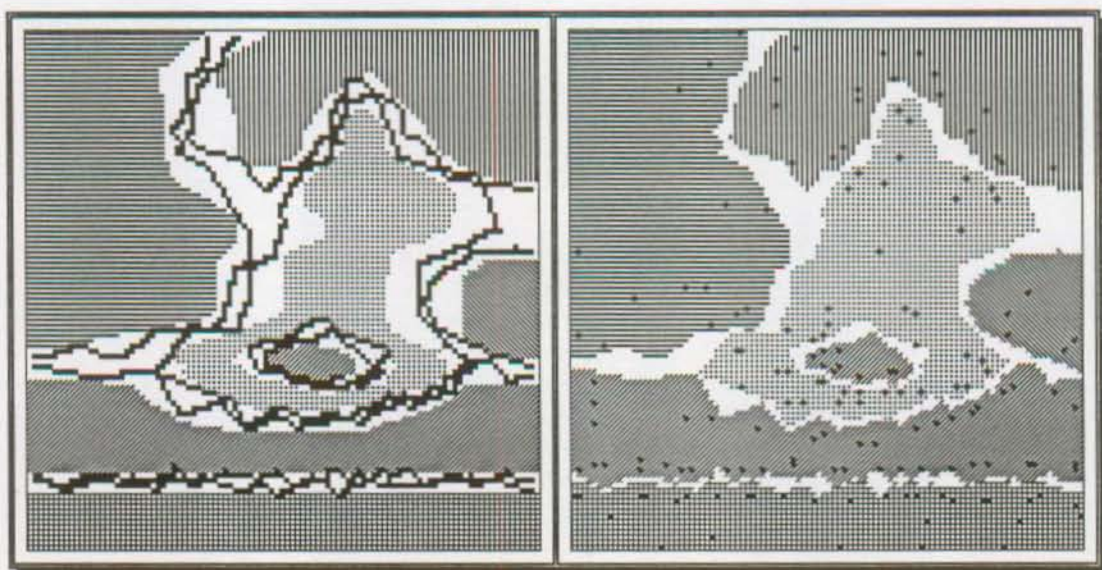

Fig. 24. After 10,000 iterations, 166 exemplar units are stored, and the error is 0, with 6 rejects. $\theta_{LIM} = 0.05$

After one more sleep and one awake passes, the result is as seen in Fig. 25.

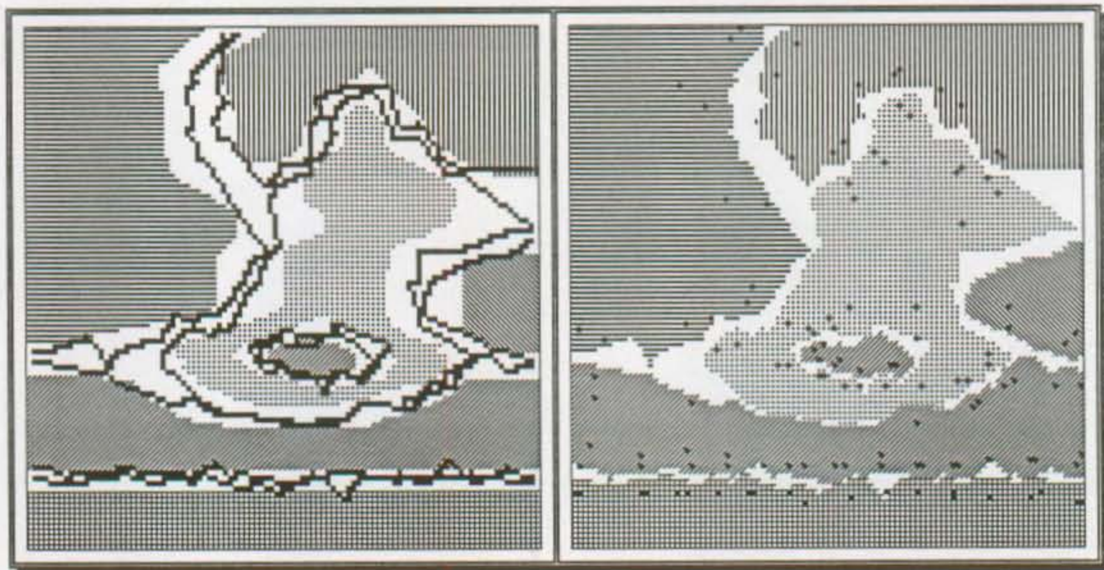

Fig. 25. After 12076 iterations, 132 exemplar units are stored, and the error is 0, with 46 rejects.

The network size is now 2671 bits. As now any input vector is compared with two vectors, to guarantee that a certain region is dominated by a class, two very close exemplars are required. That is why, the number of exemplars are nearly doubled.

Error vs. reject. By modifying the θ_{LIM} value, one can play with reject and error percentages. For example, given a certain error and reject status, to be able to decrease error percentage, one should increase θ_{LIM} value to increase reject percentage which, generally also decreases success. The question as to whether it is worth rejecting or not, depends on the application. It depends on the relative risks of giving wrong answers and rejects. It also depends on how much of the error can be trapped as reject; this depends on the signal and the θ_{LIM} chosen. The ability to reject is especially required in applications where the risk of misclassification is high, in such cases, one wants to be able to reject doubtful inputs instead of possible misclassifying even if that implies also rejecting possibly correctly classifiable inputs. The rejected inputs are then possibly subjected to classification by a more sophisticated—costlier and slower—e.g., manual, procedure.

2.7. COMPARING GAL WITH SIMILAR ALGORITHMS

In this section, the same example will be used with similar learning algorithms commonly used with neural networks to be able to comment on GAL's relative advantage and disadvantages.

2.7.1. Restricted coulomb energy (RCE) model

RCE (Reilly *et al.*, 1982) is probably the most similar algorithm to GAL. Instead of a winner-take-all network that chooses the closest as in GAL, in RCE, "prototype" units have thresholds with which they compare their activations. A prototype unit gets activated if its activation is less than the threshold.[3] This defines a hyperspheric domination region whose radius is equal to the threshold around each unit which contains all the points "close enough" to that prototype. When P is the input vector and W_i^T the prototype vector with index i and T_i its threshold, the output A_i, is computed by a winner-take-all network:

$$A_i = \begin{cases} 1, & \text{if } D(P, W_i^T) < T_i; \\ 0, & \text{otherwise.} \end{cases} \tag{2.7}$$

If an input vector cannot activate any unit, a unit, indexed e, is added at this position with an initially large domination region, i.e., large threshold, T_{INIT}. (Fig. 26).

$$\forall i,\ W_{ie} = P_i.$$
$$T_e = T_{INIT}. \tag{2.8}$$

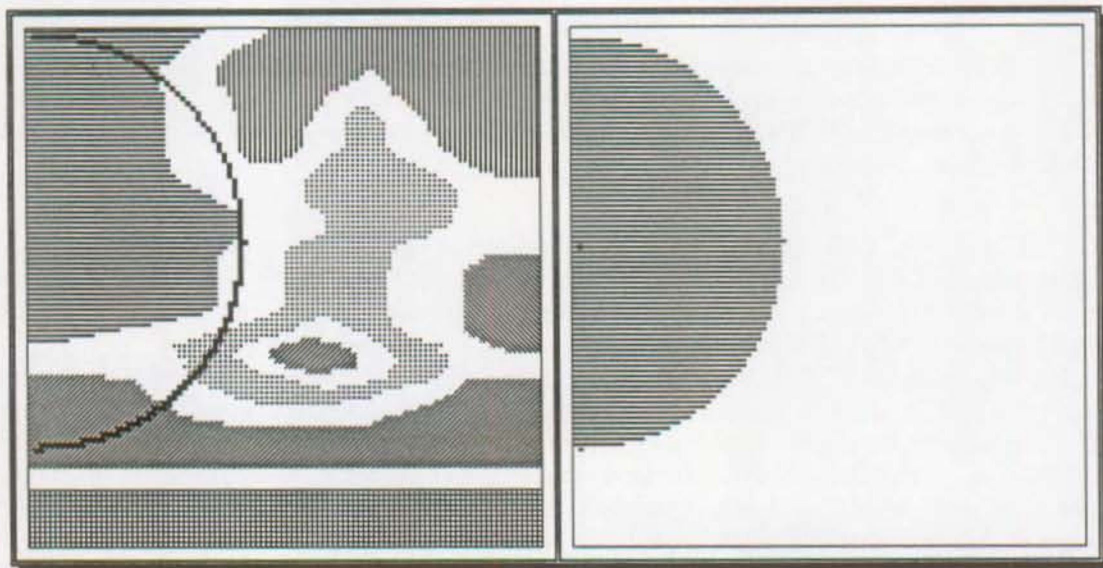

Fig. 26. In the RCE model, a prototype unit's domination region is shown, initial radius is 0.4.

[3] *A distance measure like $D()$ as defined before is assumed where $D(A, A) = 0$ and $D(A, B)$ increases as A and B get further apart. Euclidean distance used here is one such distance. A word of caution: Original article on RCE uses dot product where $D(A, A)$ is maximum and $D(A, B)$ decreases as A and B get further apart.*

During learning, if an input vector activates more than one unit, this implies that that input vector lies in a part of the space which is the intersection of at least two hyperspheres. In such a case, those units which are not of the correct class are penalized and their thresholds are decreased in such a manner so as not to get activated in a future iteration with the same vector—the threshold of the unit is set equal to its activation minus a small epsilon value. If this condition occurs during test, the input vector is rejected (Fig. 27).

$$T_e = D(P, W_e^T) - \epsilon \tag{2.9}$$

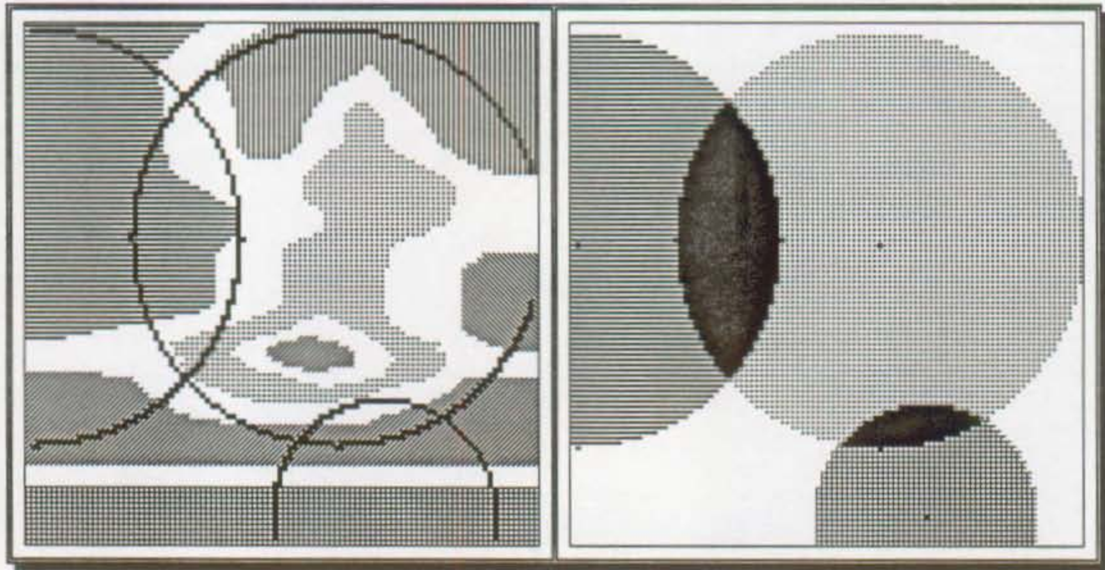

Fig. 27. How the space is divided with the RCE algorithm between 3 units are shown. The unit at the bottom had its radius decreased due to an error. Black regions show points which activate more than one prototype unit. White regions show points which do not activate any unit. Both are reject cases during test.

After 10,000 iterations, RCE with the same problem gives the result shown in Fig. 28.

One notices that units that lie in the interior of the class regions have large thresholds (radii) thus big domination regions. When they get closer to the boundaries, they tend to get more closer to each other to be able to approximate boundaries as a combination of small arcs.

RCE when compared with GAL has two drawbacks: First, calibration of thresholds require iterations, thus an RCE network learns slower than a GAL network. Secondly, as units have symmetric domination regions, to be able to cover a certain region of any shape one tends to need more units using RCE. In GAL, as always the closest is sought, units need not have symmetric domination regions. The merit of RCE with respect to GAL however is that a set of thresholdings is cheaper to implement than a winner-take-all network.

Like GAL, RCE also suffers from a dependency on the order of vectors in the training set.

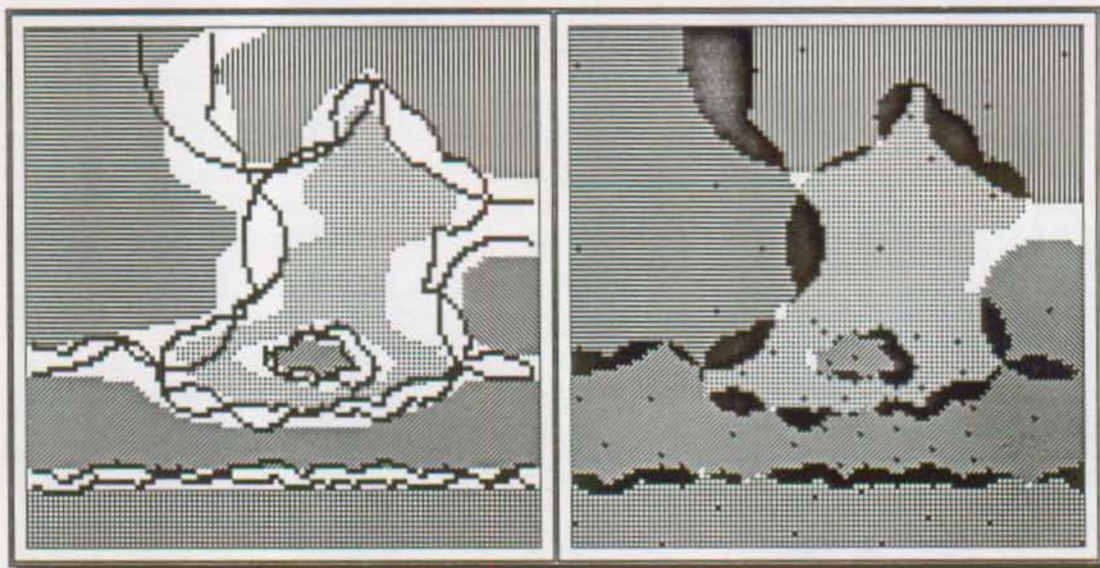

Fig. 28. RCE model after 10,000 iterations. There are 58 units stored and the error and reject values are 3 and 180 respectively.

2.7.2. Learning vector quantization (LVQ) model

The LVQ model (Kohonen, 1988) is similar to GAL in the sense that there are a set of "reference vectors" which together define a class region and when an input is given, the closest is sought. Difference however is that the network structure is fixed so no units are added or eliminated but their number should be predefined. Generally, with n reference vectors for each class, the n first vectors of that class in the training set are taken to initialize the reference vectors (Fig. 29). An LVQ network does not reject; however a strategy as proposed previously for GAL can be employed.

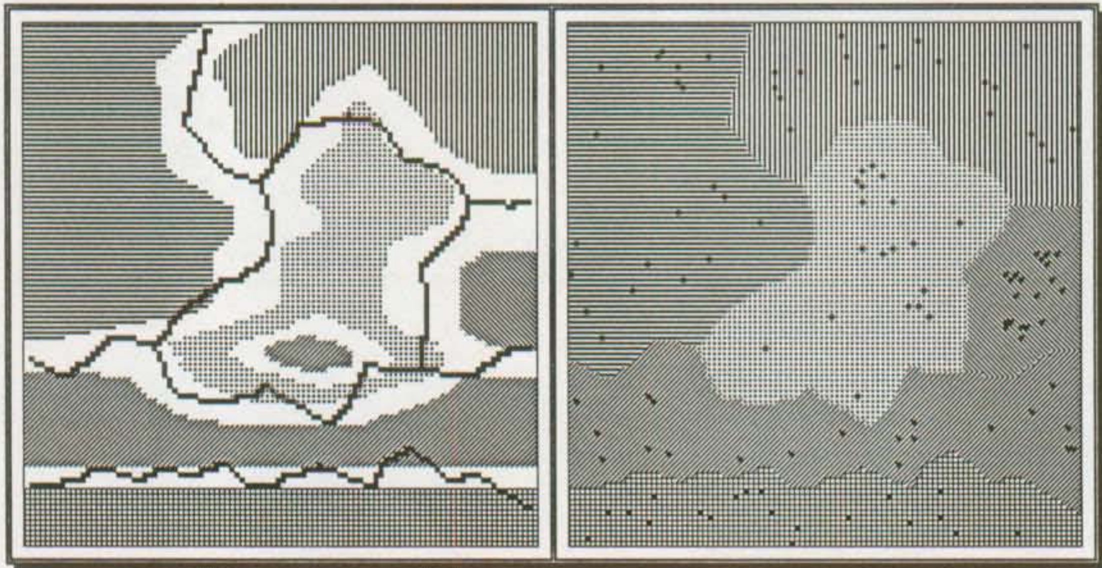

Fig. 29. With 20 units for each class, the LVQ network is initialized as seen. Error is 247.

Learning takes place as follows. Given the input vector P as belonging to class c, the closest reference vector, indexed e, is sought:

$$D(P, W_e^T) = \min_i D(P, W_i^T). \tag{2.10}$$

If unit e is of the correct class—C_e is equal to c—then reference vector e is moved towards the input vector with a gain factor decreasing in time, $\alpha(t)$. If $C_e \neq c$, W_e^T is moved away from the input vector:

$$f(C_e, c) = \begin{cases} 1, & C_e = c; \\ -1, & C_e \neq c. \end{cases}$$

$$\forall i,\ W_{ie} = W_{ie} + f(C_e, c)\alpha(t)(P_i - W_{ie}). \tag{2.11}$$

Units finally approximate the probability density of the input signal (Fig. 30).

LVQ also seems to be dependent on initial conditions. With a different set of initial reference vectors (Fig. 31), one obtains a different result despite the fact that weight vectors are tuned during a learning process. This is due to the "gap" in one of the classes.

Increasing the number of units although increases success on the training set, beyond a certain optimal number, *decreases* the success on the test set. To my knowledge, there are no guidelines that help one to compute the optimal number of reference vectors that will be used for each class, except trial-and-error. As it is one of a statistical process, one is advised to carry out a long learning process while decreasing the gain very slowly for better convergence.

Fig. 30. After 10,000 iterations with the LVQ algorithm, error reduces to 100.

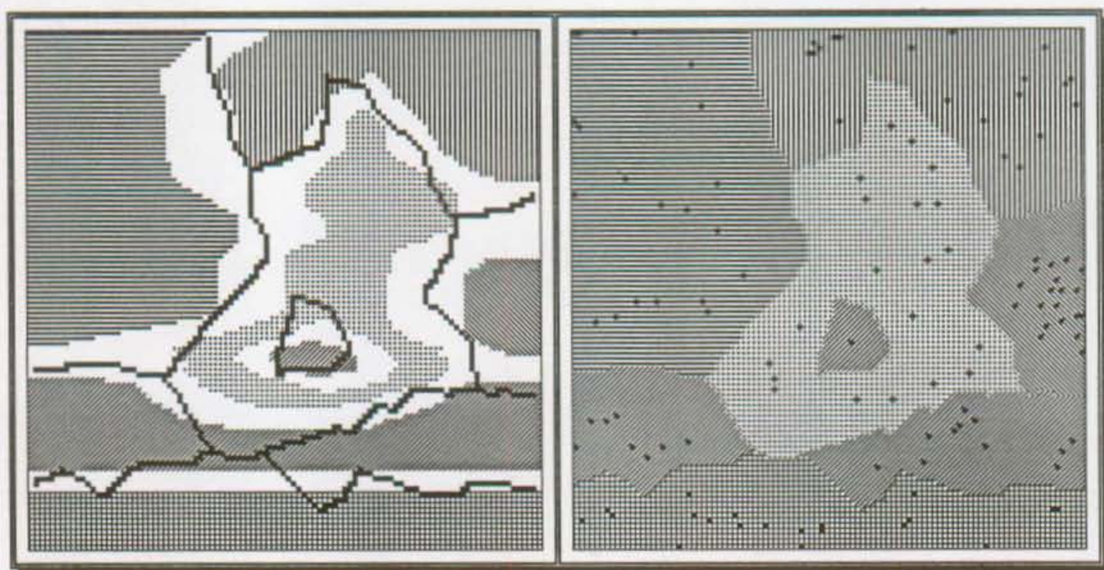

Fig. 31. With also 20 units, a different set of initial reference vectors. Error is 326.

LVQ2. A variant of LVQ, named LVQ2, has recently been proposed (Kohonen *et al.*, 1988) to better converge to asymptotic boundaries in the Bayesian sense. It is one of a finetuning process whereby in conditions where the mid-plane of two reference vectors does not coincide with the real boundary, both reference vectors are modified as to move the mid-plane closer to the real boundary.

When e is the index of the closest reference vector and f the index of the next closest:

$$D(P, W_e^T) = \min_i D(P, W_i^T)$$
$$D(P, W_f^T) = \min_{i \neq e} D(P, W_i^T). \tag{2.12}$$

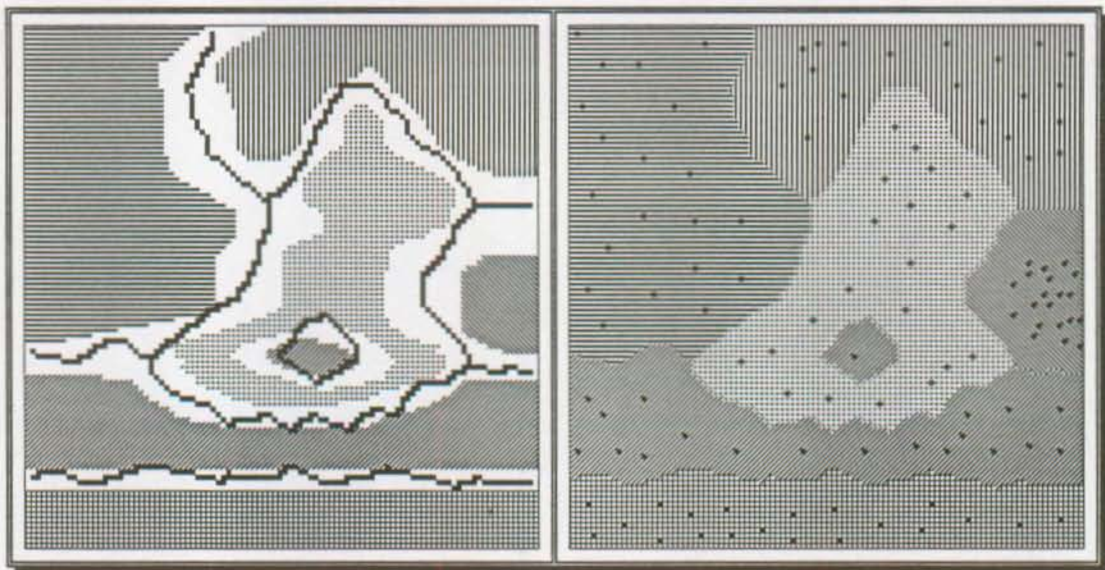

Fig. 32. After 10,000 LVQ iterations, error reduces to 15. The initial reference vector in the "island" helped.

if the following condition is satisfied

$$C_e \neq c \ AND \ C_f = c \ AND \ D(P, M) < \beta \tag{2.13}$$

where M is the mid-point of W_e^T and W_f^T

$$\forall i, \ M_i = \frac{W_{ie} + W_{if}}{2}. \tag{2.14}$$

and β gives the size of the "window" in which corrections are permitted, W_e^T is moved away from the input vector and W_f^T is moved towards it:

$$\forall i, \ W_{ie} = W_{ie} - \alpha(t)(P_i - W_{ie})$$

$$\forall i, \ W_{if} = W_{if} + \alpha(t)(P_i - W_{if}). \tag{2.15}$$

For the last case given above for LVQ, LVQ2 is used to finetune. After 5,000 LVQ iterations, 5,000 LVQ2 iterations are made. The result is shown in Fig. 33.

Both in LVQ and LVQ2, as small modifications are done on vectors, one may need high precision to store them and complex machinery to be able to perform high precision computations.

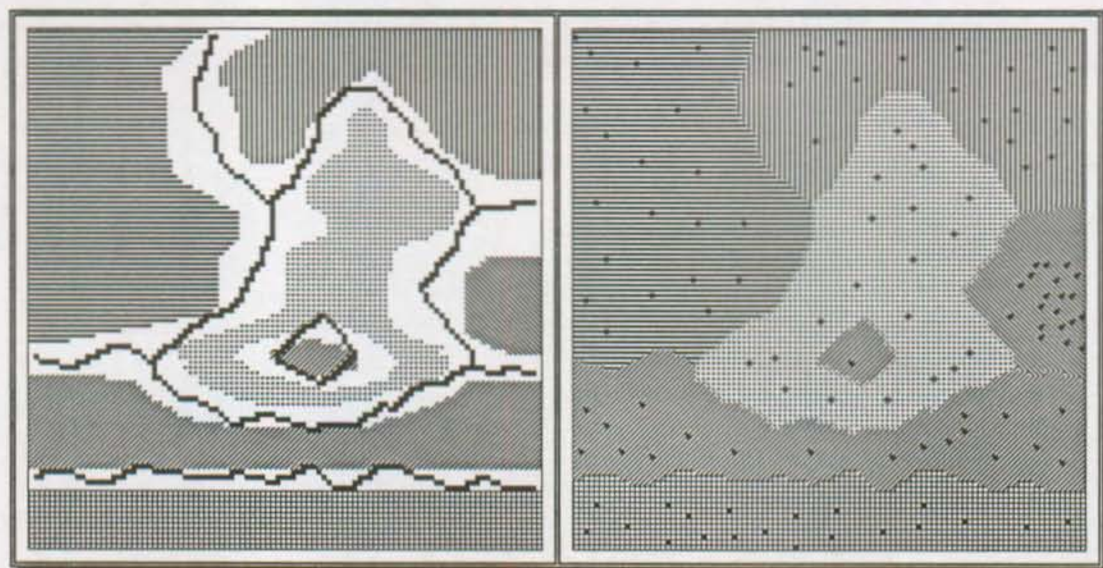

Fig. 33. After 5,000 LVQ and 5,000 LVQ2 iterations, error is 12.

2.7.3. Linear separation of classes (LS)

One can also try to separate classes from each other using linear discriminants. Separating a class from all others with a hyperplane is rarely possible in a real world application, however the advantage of looking for such simple solutions is that, if they work, they give quite good generalization over the test set. Learning the weights is done by an iterative, gradient-descent procedure, the previously mentioned *LMS* rule which is also called the *delta* rule.

In the case of linear discriminant-based methods, weight vectors denote the equation of the line separating a class from all others; W_{ij} is the weight of the connection from the ith input unit to class unit j. Initially connection weights are chosen at random with small magnitudes to break possible symmetries. This rule is not limited to classification but one can learn any mapping from a given input vector to the output units. In the case of classification, one output unit is dedicated to each class and for a correct classification, only that unit should have a high activation (> 0.9), others being inactive (< 0.1). At each presentation of an input vector P belonging to c^{th} class unit, the output of units are computed as a weighted sum filtered through a non-linearity.[4]

$$O_j = \frac{1}{1 + e^{-\sum_i P_i W_{ij}}}. \tag{2.16}$$

The modification done at time $t + 1$ is computed as follows:

$$\Delta W_{ij}(t+1) = \epsilon \delta_j P_i + \alpha \Delta W_{ij}(t). \tag{2.17}$$

where

$$\delta_j = (R_j - O_j) O_j (1 - O_j). \tag{2.18}$$

ϵ is the gain, R_j is the desired output for output unit j which is 1.0 if class of P is j, and 0.0 otherwise. The term $O_j(1 - O_j)$ is the derivative of the non-linear function given in (2.16). The

[4] *Note that this is not exactly thee delta rule; a non-linearity is not in fact necessary. What is employed here, which converges to the same solution with the delta rule is the one-layer case of the generalized delta rule, commonly named "back-propagation."*

first term of (2.17) is thus to advance along the negative of the gradient where the error criterion is the sum of the squared error. The second term is the momentum term, α being its gain, taking into account the modification made in the previous learning iteration to be able to disable oscillations during descent.

During test, if there is no unit active or if there are more than one, the input is rejected. The network size is given by the formula:

$$Network\ size = NI * NO * WP$$

where NI, NO, and WP are the number of input units, number of output units, and the number of bits required to store each connection weight. For this example, it is 74 bits. Discriminants found by the network are shown in Fig. 34:

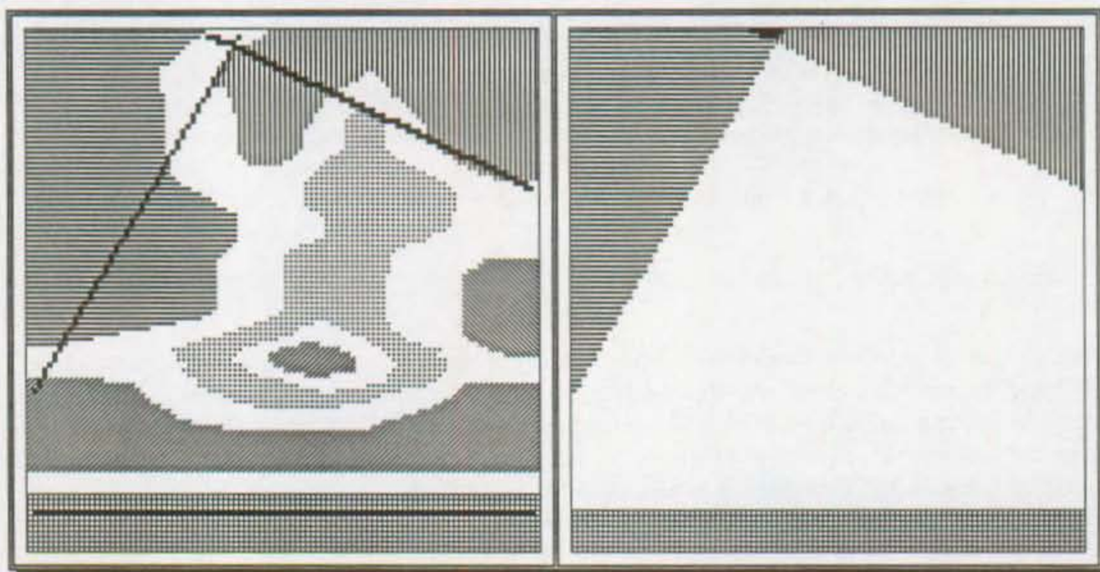

Fig. 34. One layer network trying to seperate classes from each other using linear discriminants. After 10,000 iterations, error and reject values are 3 and 4008 respectively.

2.7.4. Pairwise separation of classes (PS)

Another possible approach is to pairwise separate classes. As it requires $c(c-1)/2$ hyperplanes with c classes for all possible pairs, it is realistic for small c. In a network scheme, this corresponds to defining a hidden layer of $c(c-1)/2$ units, and c units in the last layer (Duda & Hart, 1973) (Knerr *et al.*, 1989). The connections from the input units to the hidden units can be learned by the delta rule; the connections from the hidden units to the class units are predefined and fixed.

The learning procedure is the same as explained in the previous section with the only exception that required outputs are imposed on the hidden units, H_h separating in a piecewise manner. To separate class i from class j, a hidden unit is assigned which has the desired value:

$$H_{i*c+j} = \begin{cases} 1.0, & \text{if class of } P \text{ is } i; \\ -1.0, & \text{if class of } P \text{ is } j; \\ H_{act}; & \text{otherwise.} \end{cases} \tag{2.19}$$

If the desired class is not equal to i or j, the desired output is set equal to the actual output for no modification to be done on the weights connected to that hidden unit. To make sure that there are $c(c-1)/2$ of them, the extra condition that $i > j$ is imposed. The connections from H units to class units C, named T are set as follows:

$$T_{i*c+j,k} = \begin{cases} 1.0, & \text{if } i = k; \\ -1.0, & \text{if } j = k; \\ 0.0, & \text{otherwise.} \end{cases} \tag{2.20}$$

A class unit C_k computes its activation as follows:

$$C_k = \begin{cases} 1.0, & \text{if } \sum_h H_h * T_{hk} = c; \\ 0.0, & \text{otherwise.} \end{cases} \tag{2.21}$$

When applied to the didactic example, the result shown in Fig. 35 is achieved. Network size is given as:

$$Network\ size = NI * \frac{NO * (NO-1)}{2} * WP. \tag{2.22}$$

Here it is 210 bits.

2.7.5. Multi-layer network of discriminants

Still another discriminant based approach is using a multi-layer scheme of linear discriminants. Learning of weights is done by the error back-propagation algorithm, also named the *generalized delta rule* which stands out to be the most popular learning algorithm used with artificial neural networks (Rumelhart *et al.*, 1986). Two different network structures with two, and three-layers of units are used. There are no rules by which one can decide how many hidden units will be necessary, thus it is chosen by the programmer. In what follows, they are chosen arbitrarily.

The connections are learned as previously shown in (2.17). For a connection leading to a unit which is not an output unit, where no required output can be defined, the δ_h are computed using the δ of the units of the following layer (indexed k), thus comes the name back-propagation of error:

$$\delta_h = O_h(1-O_h)\sum_k \delta_k W_{kh}. \tag{2.23}$$

BPN2: Two layer back-prop net. There is one layer of 128 hidden units. Update and momentum factors are 0.5 and 0.9 respectively. The network size is computed according to the formula given in (Guyon *et al.*, 1989). For a network with one hidden layer, the size is computed as:

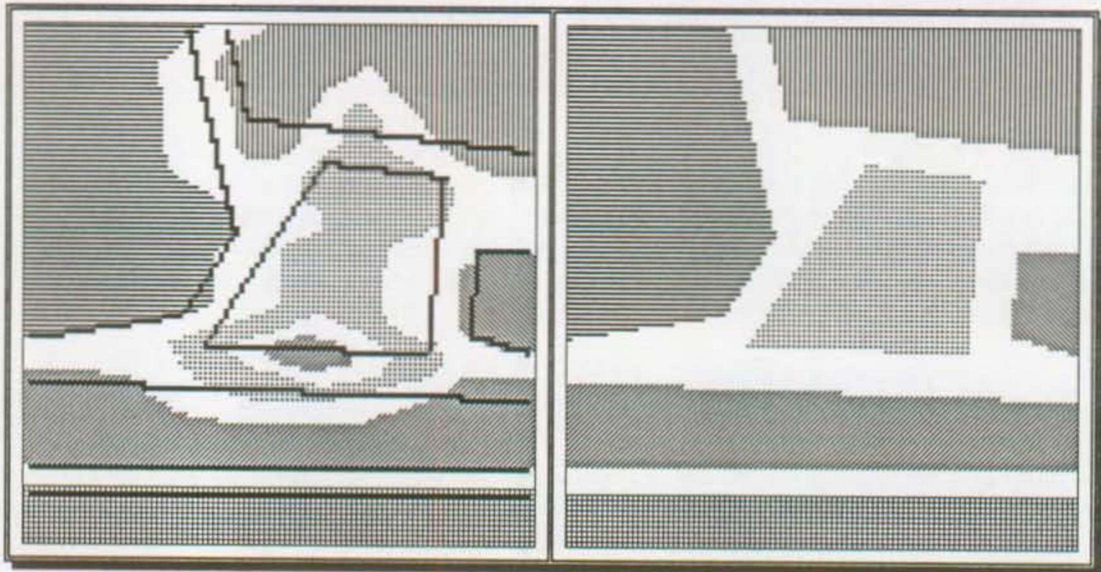

Fig. 35. Pairwise separation of classes using linear discriminants. After 500,000 iterations, error and reject values are 110 and 909 respectively.

$$Network\ size = NI * NH * BPI + NH * NO * log_2(NH). \tag{2.24}$$

where NI, NH, and NO are the number of input, hidden, and output units respectively. BPI is the precision required to store the connection weights from the input units to the hidden units. For the connections leaving a hidden layer, the number of bits required is estimated as $log_2(NH)$. For the example problem, with 128 units, network size is 7168 bits.

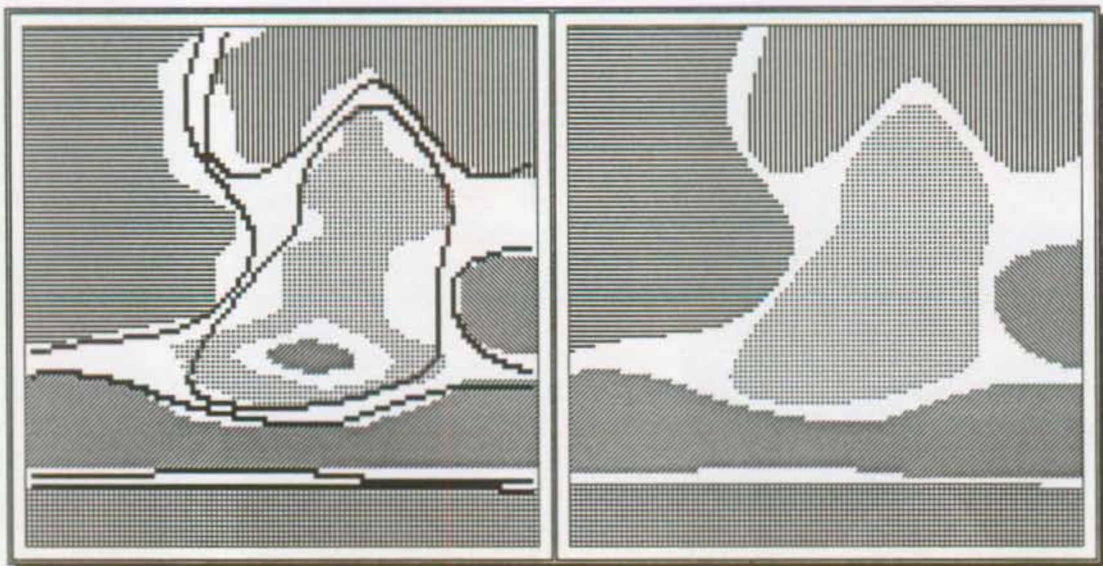

Fig. 36. 2 layer back-prop network with 128 hidden units. After 500,000 iterations, error is 84 and there are 108 rejects.

BPN3: Three layer back-prop net. There are two layers of hidden units with 64 in the first and 32 in the second. Update and momentum factors are 0.1 and 0.2 respectively. With bigger values, the network does not converge. The network size is 14144 bits.

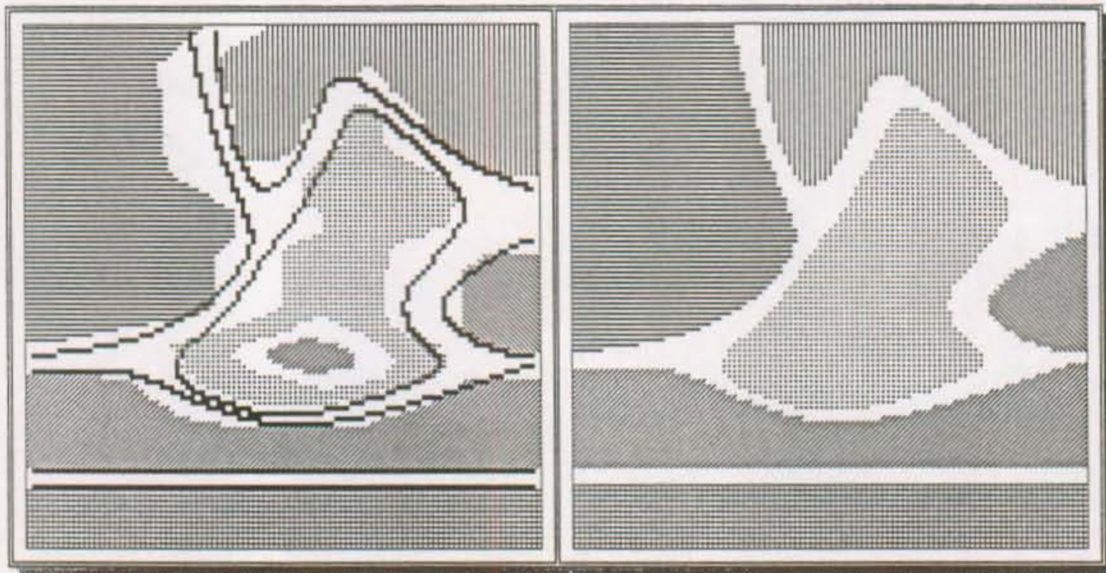

Fig. 37. 3 layer back-prop network with 64 hidden units in the first layer and 32 in the second. After 500,000 iterations, error is 84 and there are 79 rejects.

2.7.6. Summary of results

The table summarizes the results achieved for the same example using different network structures and learning rules.

Network	Learning Iterations[1,2]	Network Units[3]	Network Connections[4]	Error	Reject[5]
GAL	12,030	57	171	3	0
GAL_r	12,076	132	396	0	46
RCE	10,000	58	232	3	180
LVQ	10,000	120	360	15	0
LVQ2	10,000	120	360	12	0
LS	10,000	0	18	3	4,008
PS	500,000	15	135	110	909
BPN2	500,000	128	1158	84	108
BPN3	500,000	64+32	2470	84	79

1. The time it takes to perform that many iterations is not given here as programs have too much graphic I/O and thus given times would not be significant. However, one can roughly say that the more there are connections, the heavier will be the computational load and thus the longer will be the simulation time on a *sequential* computer—like the one used here. Simulation time increases linearly as the number of connections increase.
2. One should also note that one iteration in each model is not computationally equivalent. In incremental models, in the beginning there are less units and thus an iteration takes less and it takes more as the number of units increases. In the case of BPN2 and BPN3 models, there is also complete connectivity after (and between) the hidden units which further increases computational load. In methods based on linear discriminants, all the connections are modified in every learning iteration. LVQ (and LVQ2) takes less time than these models as only one (two in the case of LVQ2) reference vector is updated at one iteration. The computational complexity of RCE is still less which requires either allocation of a unit or modification of the threshold or nothing at all. GAL is still simpler which requires either allocation of a unit or nothing at all.
3. The number of units needed is automatically defined in the case of GAL, GAL_r, and RCE and predefined (without any trial-and-error) in others.
4. In the case of GAL, each exemplar unit is connected to all the input units, here 2, and is connected to only one output unit. RCE is like GAL with the addition of the modifiable threshold needed to be stored with each unit which here is counted as an additional connection. LVQ and LVQ2 are like GAL. In the case of discriminant based schemes, all the units except the input units have an extra connection with a "bias" unit. In the case of the output layer of the PS, this is not necessary. In BPN2, there is complete connectivity between the hidden and output units. In BPN3, there is complete connectivity in all layers. Note that this is not a total indication of the memory requirement of a network; the precision to store the weights should also be taken into account. Such precision is needed in the case of RCE for the threshold, in the case of LVQ, LVQ2 and BPNi for all synaptic weights. In GAL and RCE, as weights are just assigned once and not further modified, the precision required is that of the input and no further. This is a great advantage as will be seen further; only binary weights are required to learn binary images.
5. This is the erroneous reject case implying that the network rejected although the input does belong to a class.

To test and compare GAL in a real-world application it is applied to recognition of handwritten numerals.

The database is made up of 1200 examples of 10 digits written by 12 people where each example is a bitmap normalized to the size 16 by 16 pixels (Guyon *et al.*, 1989). This is a relatively easy database as writers all followed a given writing style. The database is divided into two parts. First 600 make up the training set with which the network is trained. Remaining 600 make up the test set and are used to test how well the network generalizes. Images are not preprocessed. Hamming distance is used as the distance measure.

As results obtained depend on the ordering of vectors in the training set, 30 runs were made each time choosing a different random ordering. Average and best case results were given. Models tested are nearest-neighbor (NN), GAL, GAL_r, RCE, and LVQ.

Network	time/sweep (sec.)[1]	no of sweeps	Average[2]	Best	Worst
NN	740–600	1	600–93.2,6.8[3]		
GAL	96–102	4	117.3–89.8,10.1	119–91.8,8.2	115–88.2,11.8
GAL_r	110–133	4	148.2–86.3,2.5[4]	147–88.7,2.9	142–84.0,2.9
RCE	91–109	6	142.0–74.1,2.9	144–77.7,2.0	131–70.7,2.7
LVQ	36–100[5]	50[6]	100.0–92.9,7.2	100–94.0,6.0	100–91.7,8.3

1. Format is (seconds)–(number of units).
2. Format is (number of units)–(success percentage),(error percentage). In the case of GAL and LVQ there is no reject; in the other models, the rest is reject.
3. With nearest-neighbor as all vectors in the training set are stored, one always gets the same result, thus one cannot talk of best or worst cases.
4. One can play with the threshold θ_{LIM} to decrease error further by getting a higher reject (and possibly lower success) percentages. Typically one is interested how much percent needs to be rejected to get 1% error (le Cun *et al.*, 1989).
5. All models except LVQ use integer arithmetic for distance computation. These simulations are done on a Sun 4/110 with a floating-point processor and accelarators.
6. Results obtained by LVQ can probably be increased by performing more sweeps over the training set with a smaller gain factor. In this simulation, starting from 0.1, it is decreased linearly till 0.0. Increasing the number of reference vectors per class from 10 to 15, increased the success on the training set from 99.2 to 100.0, but decreased the highest success on the test set to 93.7. Having 8 reference vectors per class decreased both.

A few words on LVQ2. With this example I have also used LVQ2, however it seemed to work worse than LVQ. LVQ2 only modifies a reference vector when the following condition is satisfied (see section 2.7.2): When i is the closest reference vector and j is the next closest and the input does not belong to class of i but that of j and when the input vector is less than a certain (small) distance from the mid-point of i and j. When this tight condition is satisfied, i is moved *away* from the input vector and j is moved *towards* the input vector.

One immediately notes that reference vectors should be initialized somehow before LVQ2 can be applied as otherwise some reference vectors (those that lie in interior regions) will never get modified. I use LVQ. In the case of didactic two dimensional example in this way, LVQ2 does seem to improve success. However in the case of handwritten numerals, I rarely got a success with LVQ2

higher than that of LVQ (in 30 runs). One probably needs a special initialization phase which, to my knowledge, was not mentioned anywhere.

Discriminant-based methods. One can improve generalization by employing a method based on discriminants. Results achieved with methods like pairwise separation of classes and a multi-layer network, for the same database is given in (Guyon *et al.*, 1989) and they do increase success on test set to around 98%.

Such models as they are not based on individual "exemplar" or "reference vectors" but find general class boundaries allow better generalization and thus lead to higher success values. This they do at the expense of bigger networks—GAL uses one bit per connection whereas those need more[5]—and slower learning.[6] For the problem of character recognition, as systems can learn off-line and as characters' definitions never change, learning time is not important; success on test set is more important. An algorithm like GAL will probably be more significant in an application where patterns need to be learned very fast, i.e., on-line. The computational complexity of GAL is also less, thus although it has less success on the test set, it is still interesting for many tasks.

Searching the best learning algorithm. As was also pointed out in the first chapter, it does not make much sense to say in a general manner that a learning algorithm is better than another one. An algorithm cannot be compared with another one regardless of what the task is; only with respect to one application, can one compare algorithms between themselves. Success rate may be the most important factor for one application, e.g., character recognition, network size can be important in another one, e.g., when the network is realized in silicon, or learning time can be vital in still another application, e.g., in real-time pattern learning and recognition tasks, e.g., robotics.

2.9. PREPROCESSING HANDWRITTEN NUMERALS BEFORE GAL

The success achieved with GAL in the previous section are not very satisfactory; one then naturally looks for ways to improve success without using a discriminant-based approach that increases learning time. Two techniques are proposed to preprocess digit images in this section that are appropriate to the application and thus allows a better generalization and higher success on the test set.

[5] *4–5 bits for pairwise separation of classes (Personnaz, personal communication).*

[6] *It is reported (le Cun et al., 1989), 23 sweeps over the training set (167,693 pattern presentations) were necessary for back-propagation which took three days on a Sun 4/260, to learn handwritten digits.*

2.9.1. Gaussian filter

A 3x3 Gaussian filter is applied before recognition using GAL. This has the advantage that when the image is displaced less than 3 pixels, the distance between two images is less, thus invariance to small displacements (translations, shifts) is achieved. Note that when bigger filters are used, one risks to lose details as filtering by a Gaussian is a form of "blurring." The Gaussian in the discrete case is approximated as a matrix. Multiplying this matrix with the input values lying in its receptive field to get a scalar is called a *convolution*, the matrix similarly is called a *convolution kernel* or *mask*.

$$G = \begin{pmatrix} 1 & 3 & 1 \\ 3 & 5 & 3 \\ 1 & 3 & 1 \end{pmatrix}$$

$$Pix_{ij} = \begin{cases} 1, & \text{if pixel at position } (i,j) \text{ is } on; \\ 0, & \text{otherwise.} \end{cases} \tag{2.25}$$

$$Input_{i*16+j} = \sum_{a=0}^{2}\sum_{b=0}^{2} G(a,b) * Pix_{(i-1+a)(j-1+a)}. \tag{2.26}$$

The equations given are for the discrete case. For continous functions, double sums should be replaced by double integrals. Matrices are addressed starting from 0 and the first index corresponds to the row index. A character image, in this case a '0', and the result achieved after application of the Gaussian filter is shown in Fig. 38.

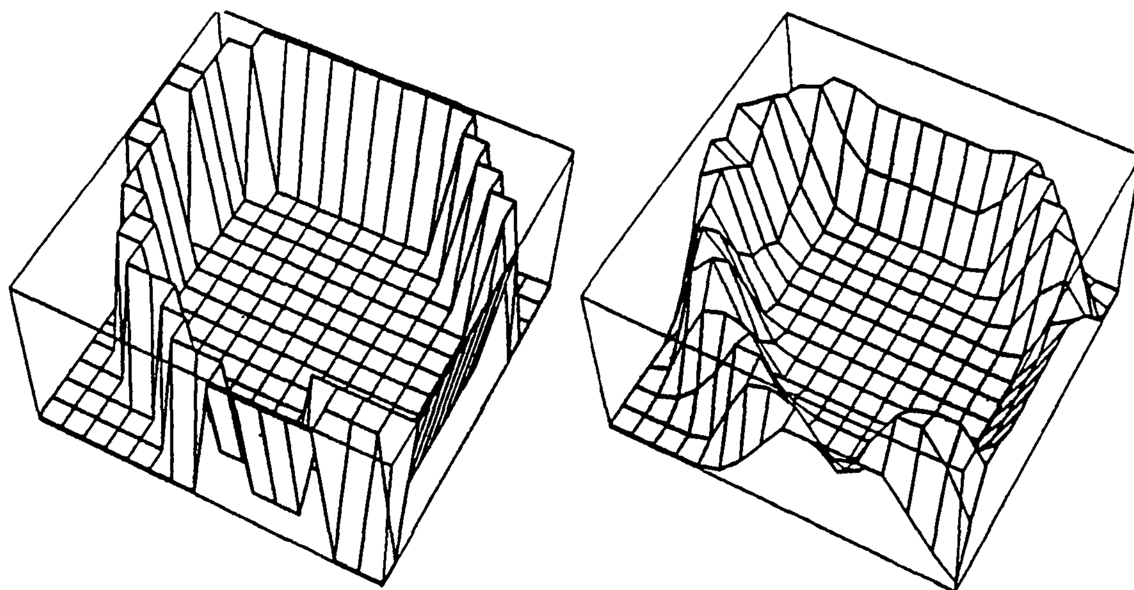

Fig. 38. Character image, to the left before, to the right after application of the Gaussian filter. In the figures, each point where two lines meet corresponds to a pixel. Original image is binary, the filtered one has 21 discrete levels.

Following are the results obtained using Euclidean distance as the similarity measure. Database used is the same. Best and worst cases are chosen according to success only. GAL without reject is used, thus anything which is not a success is an error.

Connection value range	: 0..21, 5 bits
Number of sweeps during training	: 3
Sweep time (during test)	: 51 seconds with 79 units
Nearest-neighbor limit	: 97.5%
Average over 30 runs	
Number of exemplars stored	: 95.9
Size of the network	: $95.9 * 256 * 5 + 95.9 * 10 = 123711$ bits
Success on test set	: 93.4%
Best case:	
Number of exemplars stored	: 98
Success on test set	: 94.8%
Worst case:	
Number of exemplars stored	: 95
Success on test set	: 91.8%

The fact that less exemplars need to be stored with higher success implies that the GAL network generalized better.

Although the Gaussian filter increased processing, because there are now less units stored, there is not a large overhead. Remember that it took 96 seconds to perform a sweep without the Gaussian filter using 102 units.

2.9.2. Centering the image on its center of gravity

By preprocessing one performs a transformation from the input space to an intermediate space in which lower order discriminants can be defined. Thus one, by preprocessing, indends to transform original patterns so that those patterns that belong to the same class will be more similar to each other and those of different classes will be as different from each other as possible. Such an operation is inherently dependent on the application.

When it comes to recognition of handwritten digits, one looks for a way by which different digits can be discriminated from each other. Extracting structural features like lines, or corners necessitate heavy computation. The idea proposed in this subsection is to better center the image according to its pixel density over the matrix. One computes, what can be called the "center of gravity" of the image: A weighted average of pixels where "on" pixels count and where weight decreases as distance increases. Formally, it is a Gaussian filter where the kernel is as big as the bitmap size. The pixel where this value is the maximum is taken as the new center and image surrounding it till a certain distance equal on all sizes is extracted to form the new centered image. It is this image which is fed to the recognizer.

This ability to compute a region of "interest" and extracting its content from its background makes the system *invariant to translation*. Let us define a big, e.g., of size 64 by 64, original image that is named R. We know that somewhere in it is a 16 by 16 image that we need to recognize. This is implemented as follows:

[1] First, R is checked in parallel according to a certain "criterion of being interesting." In this particular task, a region gets more interesting as the number of "on" pixels in it increase. Each pixel takes into account its value and those of its neighbors. The affect of a neighbor decreases as it is further; this is to make sure that images will be centered.

$$R_{i,j} = \begin{cases} 1, & \text{if pixel at position } (i,j) \text{ is } on; \\ 0, & \text{otherwise.} \end{cases} \qquad (2.27)$$

The effect of a pixel at the relative position (a, b) decreases as distance increases. As kernel size is 24 by 24, maximum distance is $12\sqrt{2}$.

$$GaussY_{a,b} = 1 - \frac{\sqrt{a^2+b^2}}{12\sqrt{2}}. \tag{2.28}$$

The effect thus is 1.0 at the center where $a = b = 0$ and 0.0 at the corners where $a = b = 12$.

$$Y_{i,j} = \sum_{a=-12}^{11} \sum_{b=-12}^{11} R_{i+a,j+b} * GaussY_{a,b}. \tag{2.29}$$

[2] The pixel for which this value is highest becomes the new center.

$$Y_{ic,jc} = \max_{i,j} (Y_{i,j}). \tag{2.30}$$

This point of highest pixel density, due to Gaussian which weights further pixels less, is different for different numerals; for 0, it is near the center, for 6, it is below, and for 9 it is above the center.

[3] The region surrounding this point is then extracted, which here is named F. To make sure that this new image covers the whole ancient image, its size should be bigger than 16; in this case, 24 is chosen.

$$F_{fi,fj} = R_{ic-12+fi,jc-12+fj},\ i = 0\ldots23,\ j = 0\ldots23. \tag{2.31}$$

An example image, pixel densities, and the image extracted is given in Fig. 39.

[4] A Gaussian filter with a smaller kernel size, i.e., 5 by 5, is applied as it proved its utility in the previous section.

$$GaussX_{a,b} = 1 - \frac{\sqrt{a^2+b^2}}{2\sqrt{2}}. \tag{2.32}$$

$$X_{i,j} = \sum_{a=-2}^{2} \sum_{b=-2}^{2} F_{i+a,j+b} * GaussX_{a,b}. \tag{2.33}$$

This X vector then is given as input to the GAL network. The results are summarized below. GAL without reject is employed on the same database.

Connection value range (approx. (2.28) and (2.32))	: Y 7 bits, X 5 bits
Size of the preprocessing network	: $40 * 40 * 24 * 24 * 7 + 24 * 24 * 24 * 24 * 5$ $= 8,110,080$ bits
Number of sweeps during training	: 3
Sweep time	: 1 hour 15 minutes
Nearest-neighbor limit (storing 600 exemplars)	: 99.2%
Average over 30 runs	
Number of exemplars stored	: 71.2
Success on test set	: 96.3%
Best case	
Number of exemplars stored	: 79
Success on test set	: 97.8%
Worst case	
Number of exemplars stored	: 63
Success on test set	: 95.0%

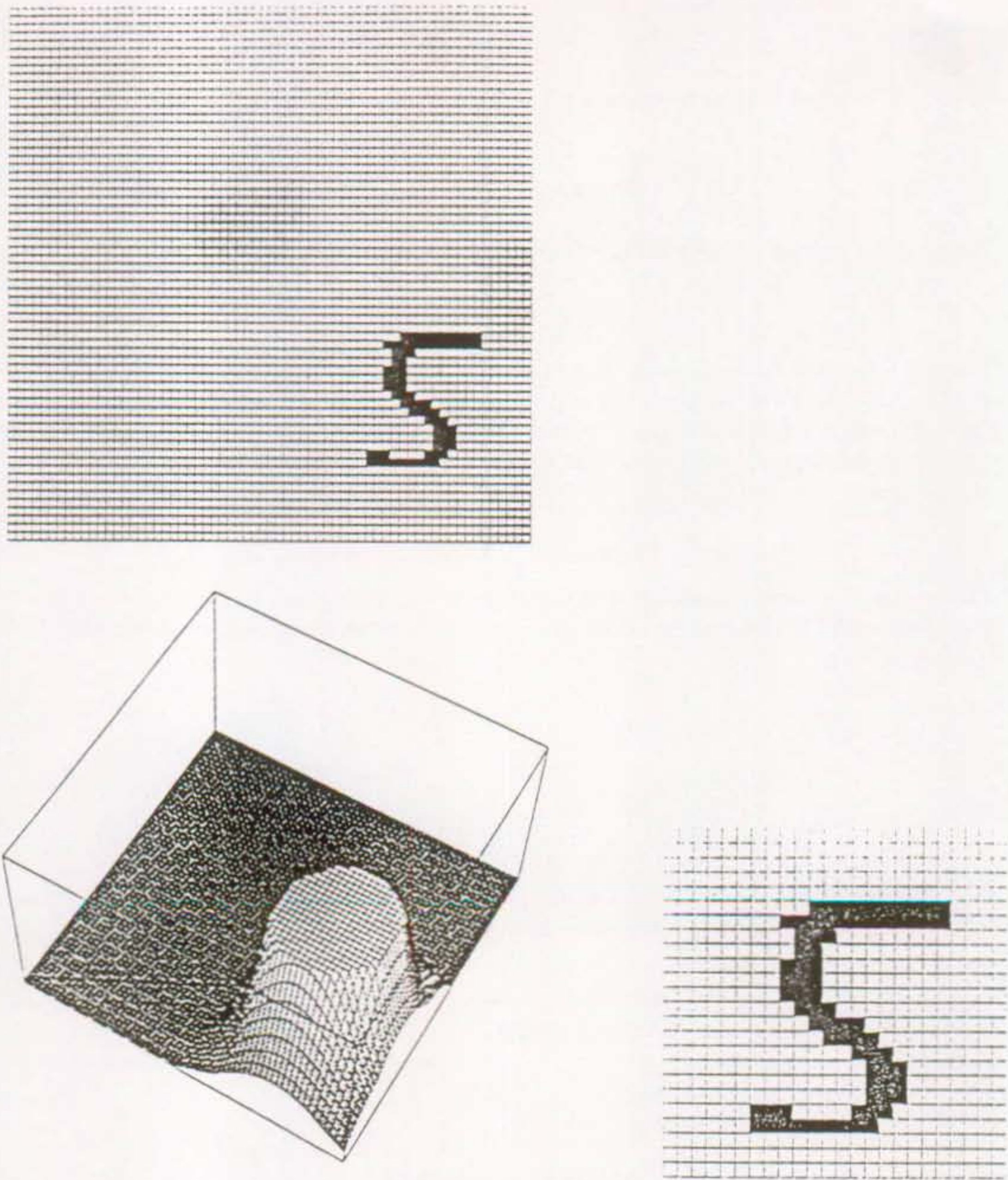

Fig. 39. An example. To the top, 64 by 64 original image, down to the left, density of pixels and to the right the 24 by 24 matrix extracted.

2.10. TESTING GAL WITH A BIG DATABASE

Recently I have tried GAL with a big database of handwritten numerals provided by the US Post Office to the AT&T Bell Labs, Holmdel, NJ. Pixels have values between 0.0 and 1.0 depending on the gray level. The database is divided into two sets; 7291 examples make up the training set and the remaining 2007 are used to test the quality of generalization.

The form of GAL without any reject was employed to be able to get highest success possible, thus anything which is not a success is an error. The results previously achieved (with 0% reject)

with this database are:

- 95.3% Back-propagation and
- 94.3% Nearest-neighbor.

The back-propagation network (le Cun *et al.*, 1990) is a four layer constrained network with weight sharing. Original image is 16 by 16 gray level. In the first hidden layer there are 12 different features with a kernel size of 5 by 5, duplicated on all positions on an 8 by 8 grid. The second hidden layer has also 12 features but placed on a 4 by 4 grid also viewing 5 by 5 neighborhoods and connected to 8 out of the 12 features of the first hidden layer. The third hidden layer has 30 units and is fully connected to the second hidden layer and also gives full connections to the output layer which has 10 units. Summarizing, the network has 1256 units, 64,660 connections, and 9,760 free parameters. It takes three days to learn on a SUN-4/260 with 167,693 pattern presentations. For 1% error, 12.1% should be rejected which leaves 86.9% correct classification. Unfortunately, I did not have time to test GAL_r to see how much do I need to reject to get 1% error.

As a first trial, the pixel images were used without any preprocessing at all. Due to limited amount of time, only two trials were made. Each trial contained three *awake* passes over the training set—which takes around five hours on a Sun 4 Sparc. Successes achieved are 91.6 and 92.3 % by storing 850 and 860 exemplars respectively. The number of exemplars stored as a function of training iterations is given in Fig. 40.

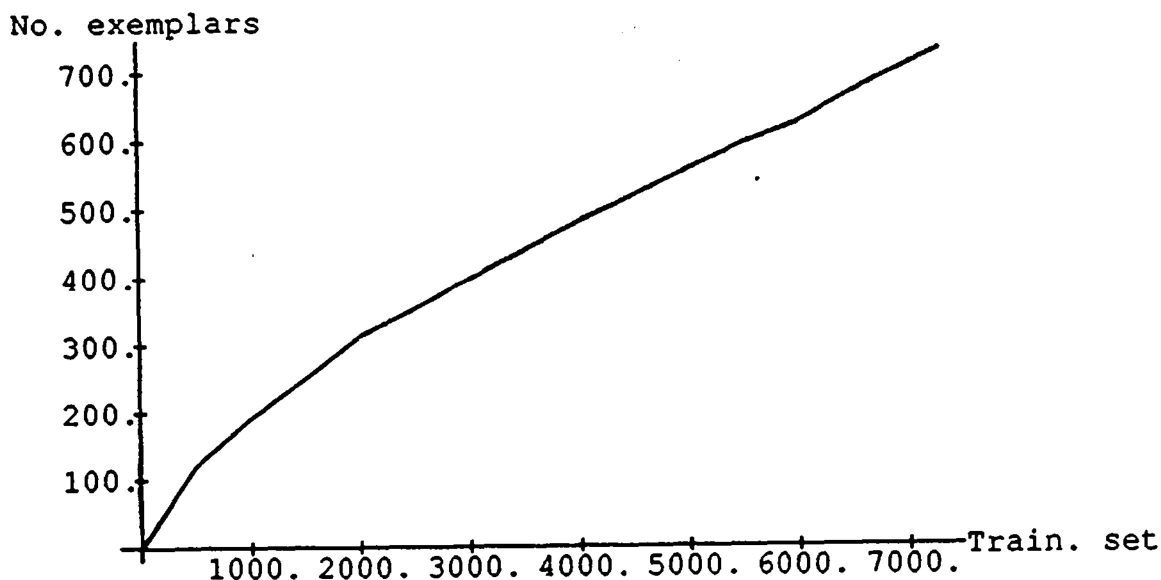

Fig. 40. Number of exemplars stored as a function of the training iterations. 16 by 16 images without any preprocessing were used.

As a preprocessing technique, the "center of gravity" of the image was computed and the image is placed in a 24 by 24 frame centered around the point of highest density, as mentioned in the previous section. Because of time limitation, the city-block distance was employed instead of the Euclidean distance in density computations to decrease pre-processing time. Successes achived are 92.0 and 91.8 storing 864 and 872 exemplars respectively. Using Euclidean distance can increase success. The number of exemplars stored vary as a function of the training iterations as shown in Fig. 41.

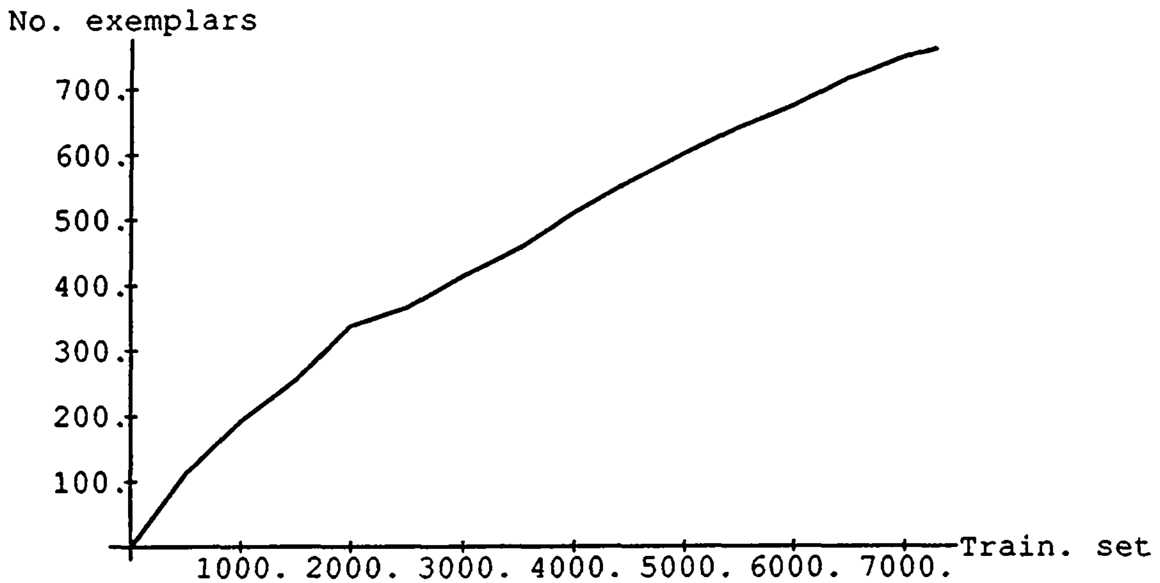

Fig. 41. Number of exemplars stored as a function of the training iterations. 24 by 24 "centered" images were taken as input.

A cheap, fast pre-processing technique that improves success when GAL is used with images is a Gaussian filter. A 3 by 3 such filter, previously shown, was employed and the successes achieved are 91.2 and 91.4 by storing 670 and 643 respectively. Although success did not go up, the fact that less exemplars were stored implies that GAL generalized better. The number of exemplars stored vary as a function of the training iterations as shown in Fig. 42.

Finally note that as success achieved by a GAL network over the test set, i.e., the actual exemplars stored, depends on the order of the vectors in the training set, these results cannot be taken as the highest possible. They do give an idea about the average success.

I thought initially that with such a big database, the number of exemplars would saturate; it does not, it seems to increase linearly. With approaches that are variants of the nearest-neighbor method this is normal; as the dimensionality increase so does the need for more vectors. This is discouraging. What is encouraging is that by storing around 10% of the training set, one can get as close as 2% to the success achieved by the nearest-neighbor method. When compared with back-propagation, success is 3% less but learning time is five hours instead of three days. Possible hardware implementation of a GAL network, due to its simplicity, will also be quite easier than that of a back-propagation network. It should be noted a network that is easy to integrate and build and very fast to train, even if its success is not the best, rests as an attractive alternative.

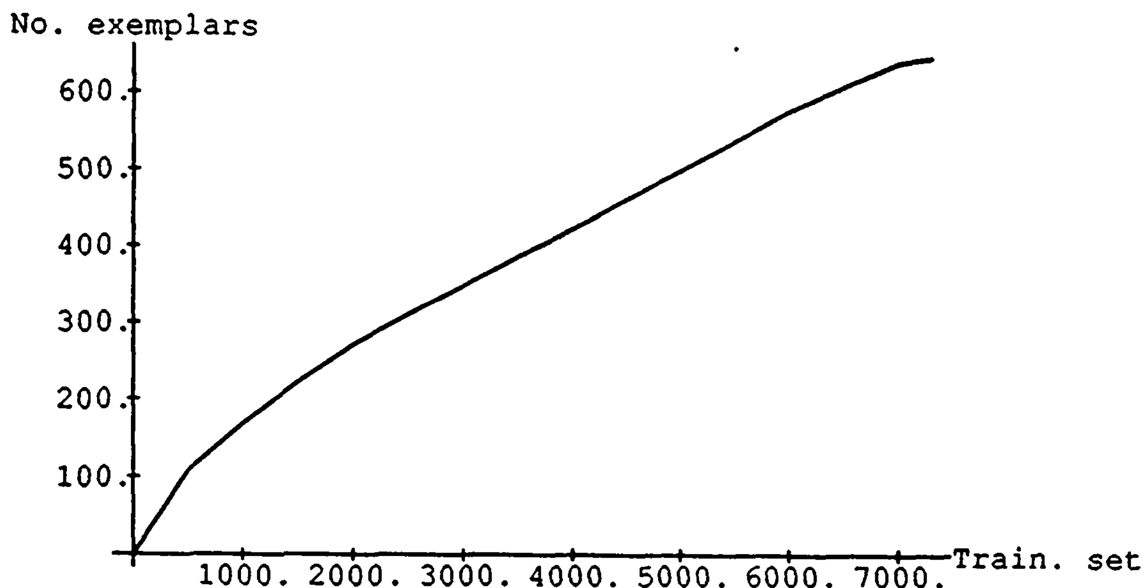

Fig. 42. Number of exemplars stored as a function of the training iterations. 24 by 24 "centered" images with a 3 by 3 Gaussian filter applied on them were taken as input.

2.11. LEARNING A MAPPING WITH GAL

GAL is a supervised algorithm to learn categories thus inherently it performs a mapping from a continous or discrete domain to a *discrete* range. However when one wants to perform *any* mapping, the ability to map to a continous range is also required. Thus using GAL for this task requires discretization of the range.

This is done in the following manner. The continuous output value is divided into discrete values having a certain range named *tolerance.* The system does not discriminate between output values whose difference is smaller than this value. In this way, a continous range is discretized by being divided into small segments. Decreasing this tolerance value leads to a finer mapping. In the case of a category-based scheme, each such segment is taken as a different class.

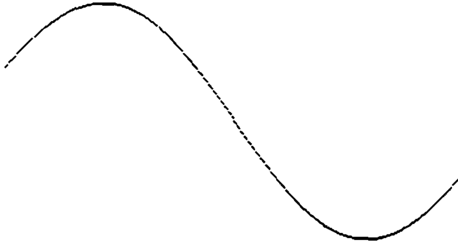

Fig. 43. Function to be learned.

2.11.1. Learning a function with one parameter

It is applied as explained above to learn the function

$$y = -3\sin(-3x) + 0.5$$

seen in Fig. 43.

Learning proceeds as follows. The network is introduced x and y where $y = g(x)$ and $g()$ is the function that we want to learn, i.e., approximate with the network. One looks for the closest exemplar e whose x_e is closest to the given x:

$$||x_e - x|| = \min_i ||x_i - x||. \tag{2.34}$$

A unit is added if

$$||y_e - y|| > TOLERANCE. \tag{2.35}$$

The added unit f has its x and y values set equal to those of the input:

$$x_f = x, \; y_f = y. \tag{2.36}$$

In the sleep mode, one chooses an exemplar unit, e, at random, disables unit e, gives x_e as input and computes the closest unit f. Unit e is eliminated if

$$||y_e - y_f|| < TOLERANCE. \tag{2.37}$$

During test, given a certain x value, the closest x_e is sought and y_e is given out as response. Result achieved using GAL in this manner after 800 learning iterations is seen in Fig. 44. Both functions, original and approximation, superposed are also given there. For this mapping, GAL added 86 units, then eliminated 2, and is left with 84 units. The mean square error (MSE) is 1.533E-5. As seen the approximation function has a ladder-like shape. The difference between two steps is equal to the tolerance value chosen, here 0.01. Each step is the domination region of one unit. The width of a step increases as the derivative of the function at that region decreases; the more horizontal-like is the part of a function, the less number of units are required to approximate it.

Fig. 44. GAL's approximation and its superposition with the original function.

2.11.2. Improving response by interpolation

The behavior of GAL can be improved by adding an interpolation mechanism. Here, instead of choosing the closest one, one chooses the two closest and uses their values to interpolate the function value for the given point. When e and f are the two closest units, the response y is computed as:

$$y = y_e + (x - x_e)\frac{y_f - y_e}{x_f - x_e}. \tag{2.38}$$

With this approach, GAL used only 16 units. Tolerance value was the same, 0.01. The approximation and its superposition with the original function is given in Fig. 45. MSE is 1.986E-5.

Fig. 45. With interpolation, GAL's approximation and its superposition with the original function.

By modifying the tolerance value, one decreases the number of units stored at the expense of a worse approximation. For the same function, when a tolerance value of 0.1 is used, there are 6 units stored and MSE is 285.152E-5. With a tolerance value of 0.001, 40 units are stored and MSE is 0.025E-5 (Fig. 46).

Fig. 46. To the left, tolerance is 0.1 and to the right, tolerance is 0.001.

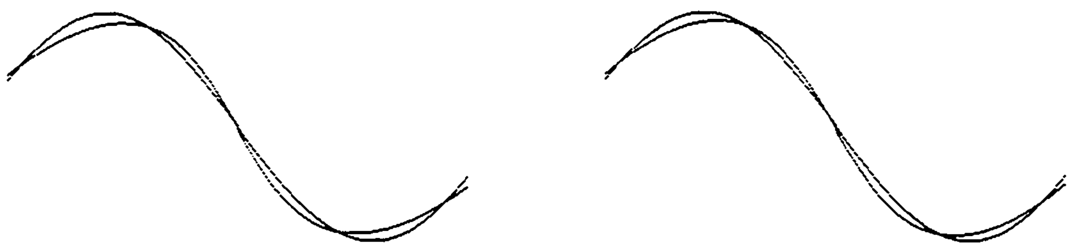

Fig. 47. Two layer back-propagation network learning the same function. To the left with 8 hidden units, to the right with 16 hidden units. Both are after 200,000 learning iterations.

2.11.3. Learning a function with back-propagation

The same function is also learned with a two layer back-propagation network with 8 and 16 hidden units. ϵ and α values are 0.3 and 0.5. Response of networks after 200,000 iterations are given in Fig. 47. MSE values are 47.202E-5 and 35.974E-5 respectively for networks with 8 and 16 hidden units. Both of these values are more than what is achieved by GAL in 800 iterations.

2.11.4. Robustness

As can be seen, GAL learns very well and very fast when the learning environment is perfect. However one rarely encounters such perfect circumstances.

One may have noise in the x and y values during learning which degrades GAL's performance as one may during learning encounter two significantly different y values for x that are very close. Back-propagation, as it is an iterative procedure, will be less affected by such conditions.

The curse of dimensionality exists also for this problem; as the number of parameters of the function to be learned increases, so does the number of units GAL needs to store.

Another shortcoming of GAL with respect to back-propagation is that in some applications one may not have any access to required y values for some range of x. Thus an effective interpolation mechanism may be needed. GAL takes the closest value and gives it out, or takes two and performs a linear interpolation. Back-propagation under such circumstances would interpolate better. However note that when it comes to *extrapolation*, back-propagation does not work well either. Keep also in mind that under such circumstances with the expense of a computational overload, one can perform an interpolation of a higher order with GAL to improve success.

2.12. A CRITIQUE OF GAL

GAL basically can be defined as a variant of the nearest-neighbor method, where one, instead of storing all the vectors in the training set, stores only a subset. The successive *awake* and *sleep* passes over the training set allows the system to choose a good subset, namely those which are closest to class boundaries. One should note that as generally training sets tend to be big and redundant, the advantage of GAL cannot be underestimated in terms of minimizing memory and computational requirements.

One problem with GAL is that the final set of exemplars achieved depends on the ordering of vectors in the training set. For the same training set, when vectors are chosen in different orders, one may get different final set of exemplars. Although all give 100% on the training set, they may give different results on the test set. Neither the number of exemplars nor the success vary that much, but the problem is that *they may vary.* I advise to eventual GAL users, if they have all the training set stored somewhere, to order the set in several ways, i.e., at random, and try GAL several times—as done above while benchmarking. Whichever of the networks arrived at the end is best, when one does not have any access to the test set, is a question one cannot readily answer. For example, one cannot make sure that the one using the smallest number of units to handle the training set will give the highest result with the test set. Note that when one tries to learn on-line, there is only one order and thus only one final set of exemplars is possible.

GAL is very simple to understand and use; there are no parameters for which optimal values need be searched (except the θ_{LIM} value in case of reject that needs to be tuned to get the required reject rate to guarantee a certain maximum error percentage). No a priori knowledge of task is required to tune the network simply because there is nothing that can be tuned.

The idea of simplifying network complexity by assessment of relevance of units and eliminating those that are not very important is not a new idea. It is commonly used in the case of adaptive filtering (Jutten, personal communication). Previously with back-propagation, several researchers have proposed techniques to compute the relevance of units and/or links to be able to delete them if their absence does not contribute too much to system error. Examples are "pruning" by Rumelhart, "skeletonization" by Mozer and Smolensky (1989) and "optimal brain damage" by le Cun (1989). The aim basically is to reduce the number of free variables, i.e., parameters that define the network

structure, namely, the connection weights, and thus be able to improve generalization. If one makes an analogy with curve-fitting, the idea is to eliminate the higher-order terms if their absence does not cause too much deviation.

The idea of "forgetting" or "reverse learning" to improve the quality of association exists in the literature. Hopfield, Feinstein, and Palmer (1983) proposed learning random vectors with a negative factor using the Hebbian learning rule to eliminate local maxima in a Hopfield network (Hopfield, 1982). It was also applied to the Boltzmann machine (Hinton & Sejnowski, 1986). The idea originated from the proposal of Crick and Mitchison (1983) which basically says that during Rapid Eye Movement (REM) sleep when dreaming occurs, an active process of reverse learning or forgetting occurs to increase the capacity of the memory by getting rid of certain unwanted stored associations. This will be further mentioned in the concluding chapter when biological plausibility of incremental learning is discussed.

GAL does not extract any features; there are no hidden units trained to extract features common to many classes. One just assigns patches of the space to classes without taking into account the class densities. There is no limitation on the shape of classes that can be learned. However, if class boundaries are low order, e.g., linear, GAL does not do a good job; it assigns many units to piecewise approximate a line which causes a waste of units as seen in the example shown previously. However, when class separations have strange forms, i.e., concave, GAL can handle them.

In an algorithm based on iteratively modifying the connection weights, one can find good low-order separating boundaries. In GAL, there is local representation; connection weights are not shared to store many associations. The disadvantage is, as I have already mentioned, one cannot find common characteristics shared by many patterns. The advantages are two: To the first, one can add or delete an exemplar unit without disturbing other exemplar units. One can modify the system without disturbing its past knowledge. Adding one more association in GAL is one iteration. In back-propagation for example, one cannot just perform iterations with one vector only; because the weights are shared, modifying the weights in favor of one association only will disturb the others.

The second advantage is that one does not need to make thousands of iterations.

It was proposed (Mézard, 1989) (Knerr *et al.*, 1989) to use an incremental addition of units with an iterative learning process to be able to get the best of both worlds. Although such algorithms may find low-order discriminants, I think, they lose what is best in an incremental learning, namely, *the ability to add an association at one shot, just by one iteration.*

The main advantage of an incremental learning algorithm is that *there is no longer the concept of a training set that needs to be explicitly stored, and over which several passes should be made for its contents to be learned off-line.* In GAL, patterns and the classes with which they are associated are learned as they are encountered. The network slowly builds up, new classes and new exemplars are added when necessary and are eliminated when they are no longer needed. This implies the possibility of *on-line* learning, that is, directly during use.

Relatively low success achieved by GAL can be increased by preprocessing input patterns as shown in the case of handwritten numerals. In the next chapter, I will propose a method, also incremental, to learn features, i.e., characteristics that are shared by many patterns, from examples by which success can further be improved.

Although GAL is an algorithm for learning of categories, it can easily be modified to learn any kind of mapping, including those where the range is continuous. It also proved itself to be a lot faster—1000 times—than back-propagation.

3

Incremental Unsupervised Learning

To detect regularities in the relations of objects and so construct theoretical physics requires the disciplines of logic and mathematics. In these fundamentally tautological endeavors we invent surprising regularities, complicated transformations which conserve whatever truth may lie in the propositions they transform. This is invariance, many steps removed from simple sensation but not essentially different. It is these regularities, or invariants, which I call ideas, whether they are theorems of great abstraction or qualities simply sensed.

—Warren S. McCulloch, "Why the Mind Is in the Head."

3.1. ON FEATURES

In the previous chapter when talking of preprocessing, I have mentioned the need of defining an intermediate space which would facilitate learning discriminants, or in the general case, learning of any mapping. It was also mentioned that the definition of this space is independent of how it is actually divided between classes. One, during this process, does not need a supervisor to actually label patterns as belonging to this or that class; learning to define such an intermediate space is called *unsupervised learning.* Dimensions of this space, as also mentioned, are called features and transforming an input vector to the feature space is called feature extraction. The feature space is also referred to as the *representation.* Representation A is said to be better than another one, B, when in A discriminants are defined using lower-order polynomials which thus allows better generalization.

Definition of the feature space is that of extracting the statistically salient properties of the input signal. The aim is to define "a model of 'what usually happens' with which incoming messages are automatically compared, enabling unexpected discrepancies to be immediately identified" (Barlow, 1989). As it is that of finding the regularities, it is the redundancy of the signal, i.e., the signal distribution, that is important. It allows one to find out where for example, the high-density regions, i.e., very frequent events, are. It naturally follows that many iterations are necessary for a statistical process to converge to real values.

Oja has shown (1982) that a unit whose weight vector is modified by being pulled towards the current input vector with a gain factor decreasing in time, acts as a principal component analyzer, i.e., finds the data components having maximum variance. When more than one such unit is used, a mechanism is needed to forbid them to specialize all on the same set of components but have them maximally sensible to *different* set of signals. One thus should penalize correlated output between feature-detector units.

Sanger (1989) achieves this with an explicit Gram-Schmidt orthogonalization and finally gets the n principal axes of the distribution, i.e., the first n eigenvectors of the covariance matrix.

In competitive algorithms, one implements a competition between units using a winner-take-all type non-linearity after which only one unit is modified: the closest (Kohonen, 1982) (Rumelhart & Zipser, 1986). Given the input space and N units, due to competitition between units, the network acts as if the input space is divided into N boxes and a unit becomes active when current input lies in its box. Learning in this sense means deciding on the size and position of boxes. In such methods, the weight vectors converge to the eigenvectors of *subsets* of the distribution.

In self-supervised backpropagation or "encoder" problem, a two-layer network is trained to perform the identity mapping, yet the number of hidden units is set to be fewer than the number of inputs. The hidden units must therefore discover an efficient encoding of the input data (Cottrell & Fleming, 1990, see also references therein). The hidden units do not find the eigenvectors themselves, but find a set of vectors that have equal variance and span the same space as the eigenvectors do.

The quality of the transformation is proportional to how well the density of the weight vectors, W_i^T, approximate the probability density of the input signal. When N is the number of units, regardless of the distribution of the input signal, we want each unit to win the competition, i.e., activated, with an equal probability of $1/N$. Information conveyed is maximized in this case. The reason for this is the following (Barlow, 1963).

The capacity of a channel carrying in an interval t_i, one member of a set (an "alphabet") of n alternative mutually exclusive symbols of probabilities $P_1, P_2, \ldots, P_n$ is:

$$I = \sum_{k=1}^{n} P_k log_D \frac{1}{P_k}. \tag{3.1}$$

This capacity is maximum when all the probabilities are equal to each other and each therefore has the value $1/n$. In this case,

$$I = log_D n. \tag{3.2}$$

In a pure competitive learning method, when the distribution of the input signal is not uniform, there will be units which gain the competition more than other units and thus will have more chance of being modified. There may even be units which never win the competition. There are two ways to get around this problem:

[1] Units are modified even if they do not win the competition. The update factor for such units is less than that for the winner. Kohonen (1982) proposed using a Mexican-hat function which has the effect that when a unit is pulled towards the input, its neighbors are moved also. This will help these neighbors to get closer to the region of high density and will give them a chance to win the competition and get modified eventually.

[2] The second possibility is to make the frequent winners less susceptible to win by imposing additional constraints. Rumelhart and Zipser (1986) use a threshold which defines a hyperspheric domination region (volume) around each unit. Winners increase their threshold (when dot product is used as the similarity measure), i.e., their regions get smaller, and losers decrease it. DeSieno (1988) proposes a "conscience" factor for each unit which affect the result of the competition; units that have won frequently in the past, "feel guilty" and tend to lose competition afterwards.

The result of such modifications is that there will be more units in a higher density region. Units lying in low density regions have larger boxes and box size decreases as density increases. One therefore makes sure that units have equal probability of being activated.

3.2. INCREMENTAL UNSUPERVISED LEARNING

The algorithms mentioned above for unsupervised learning use a static network structure whose parameters, i.e., synaptic weights, are modified during learning. That is to say, there are a fixed number of boxes whose positions and sizes change during learning. After learning, the synaptic weight vectors of the units approximate the distribution of the input signal.

The idea of adding new feature detector units, when a sufficiently different input is encountered, is not new. ART (Carpenter & Grossberg, 1987) has it by a vigilance parameter which is used as the threhold. In ART however, this parameter is the same for all the units and is fixed, thus information transfer is not maximized; one assumes uniform distribution.

As variants of the self-organizing map, methods have been proposed by which units are added in an incremental fashion to a map. Angéniol *et al.* (1989) have proposed it for solving the travelling salesman problem. Jockush (1990) adds one unit at a time near the closest unit; because position with respect to closest is chosen randomly, map topology is not completely conserved. Rodrigues & Almeida (1990) propose an algorithm by which a number of units are added at once on all sides, their positions being interpolated from the ancient units as to conserve the existing map organization.

GAR, following the idea of structural modification mentioned in chapter 1, is an incremental algorithm. The network structure is dynamic; units and links are added and removed during learning if and when necessary.

3.3.1. Learning in GAR

In the case of an incremental unsupervised learning scheme, initially there are no boxes; they are added when and where necessary and removed when no longer relevant. This capability implies that even if the input signal characteristics changes in time, the network is able to modify itself to accommodate it. The network structure is given in Fig. 1. The first layer is the layer of input units which can have binary or analog values. The second layer is the layer of units that interest us which are linear, where each unit computes the distance between its synaptic weight vector and the input.

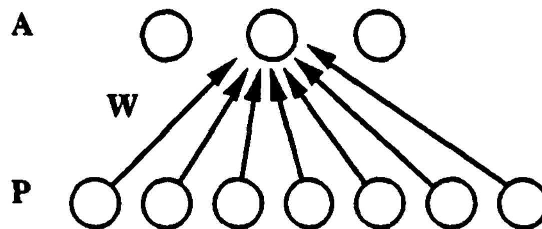

Fig. 1. GAR network structure

In GAR, the distance between current input P and synaptic weight vector of unit i, W_i^T is computed using a distance measure $D()$. It is assumed here, as previously, that $D(A, A) = 0$ and $D(A, B)$ increases as A and B get further apart.

$$A_i = D(P, W_i^T). \tag{3.3}$$

Associated with each unit is a threshold value T_u; unit u is selected during learning if:

$$A_u = \min_i(A_i) \ AND \ A_u < T_u \tag{3.4}$$

Such a thresholding implies a hypersphere around each unit with radius T_u; a vector P is in the hypersphere of unit u if the distance between them is smaller than this radius. Therefore, input vector P activates unit u if u is the closest unit and if P lies in the hyperspheric domination region of u, i.e., $D(P, W_u^T) < T_u$.

Originally, only the units of the input layer exists. Learning proceeds as follows:

[1] Input P is given and the unit responses, A_i, are computed.

[2] If the input does not lie in the domination region of any unit, i.e., no unit is selected, a unit u is added and its weight vector is set equal to P. Originally, a unit has a certain initial T_{INIT}, which as we will see afterwards may change in time:

$$\forall p, \ W_{pu} = P_p.$$

$$T_u = T_{INIT}. \tag{3.5}$$

[3] If the input *does* lie in the domination region of a unit, say u, the weight vector of u is pulled towards the input vector with a gain equal to $a * T_u$, where a, the gain, is between 0 and 1:

$$\forall p, \; W_{pu} = W_{pu} + a * T_u * (P_p - W_{pu}). \tag{3.6}$$

Scaling the update factor with T_u is to make sure that in big regions the modifications will be big, and in small regions they will be small. Other units' weight vectors are not modified. a, gain, need not be decreased in time; when T_u is decreased during learning, so does the update factor.

Above it is mentioned that the size of the domination region of a unit should be inversely proportional to the input density in that region. This implies in GAR that units lying in a high density region should have smaller radii, T. For this purpose, a so-called *trophy count*, τ_u, is associated with each unit. When a unit is selected as closest, i.e., wins the competition, its trophy count is incremented by 1. When it reaches a limit value, τ_{MAX}, the unit's T is decreased to decrease the size of the region of domination. A certain T_{MIN} is defined as the minimum T value. The process proceeds as follows:

When unit u is selected:

$\tau_u = \tau_u + 1$
if $(\tau_u > \tau_{MAX})$
$T_u = T_u - \gamma * (T_u - T_{MIN})$
$\tau_u = 0$

γ is the gain denoting step size in modifying T towards T_{MIN} and naturally is between 0 and 1.

To converge better to the mean of a region, one can divide the update factor with the trophy count:

$$\forall p, \; W_{pu} = W_{pu} + \frac{a * T_u}{1 + \tau_u} * (P_p - W_{pu}). \tag{3.7}$$

This has the effect that as a unit gets higher trophy, the modification will be less and final W_u^T will converge to the mean. This is analogical to the idea of decreasing the update factor with time, as proposed by Oja (1982).

What I call boxes or regions are sometimes called *clusters*. The idea is that similar enough patterns will cause the same unit to get activated, and thus the weight vectors of these units can be thought of acting as cluster prototypes for grouping similar patterns under the same cluster index.

3.3.2. Forgetting

We have said previously that what we want is to approximate as well as possible the probability density of our input signal with the density of the synaptic weight vectors of the units. If P_i is the probability that unit i is selected as the responding unit, we want P_i to be equal to $1/N$ where N is the number of units. The information transfered, as mentioned before, is maximized when as here, the variances of units are equal. As units when they are added have initial, not so low T values, it may be the case that a unit does not have a large effective domination region due to overlapping hyperspheres; the distance between two units is smaller than the sum of their radii. Units for which these overlappings are big, may be eliminated in a *sleep* mode.

In *sleep* mode, inputs are generated as usual and trophies are counted. In this phase, thresholds, T, are not taken into account, the closest is chosen; no T modification is done either. Finally, a unit is eliminated if its trophy count, τ, is less than a certain limit value. When M is the number of iterations and N is the number of units, unit u is eliminated if:

$$\tau_u < (1 - k)\frac{M}{N} \tag{3.8}$$

where k is between 0 and 1; a typical value is 0.5.

The *sleep* phase is important to have the units converge to good results. Experience has shown that for every approximately 5 *awake* sweeps over the training set, a *sleep* phase should be made.

Elimination of units with low trophies also allows the system to handle the case when the probability density of the input signal changes in time, if ever this happens. For example, a unit which originally dominated a high density region when the signal changes as to have low density there, will have small trophy count in that region and will be eliminated in the next *sleep* pass. Next *awake* pass during learning, as there will be no unit there, will add a unit with a large T, radius, i.e., big domination region.

This ability implies that when the signal changes in time, GAR is able to forget the old features and learn the new ones.

3.4. GAR: A DIDACTIC EXAMPLE

As examples to show how GAR works, two-dimensional input signals are chosen to simplify displaying of results. Two example distributions are used: Uniform and exponential. Distance measure is Euclidean (Fig. 2).

When u is a random variable uniformly distributed between 0 and 1, x, random variable obeying an exponential distribution is computed as:

$$x = \frac{e^{\frac{e^u - 1}{1.71828}} - 1}{1.71828}. \tag{3.9}$$

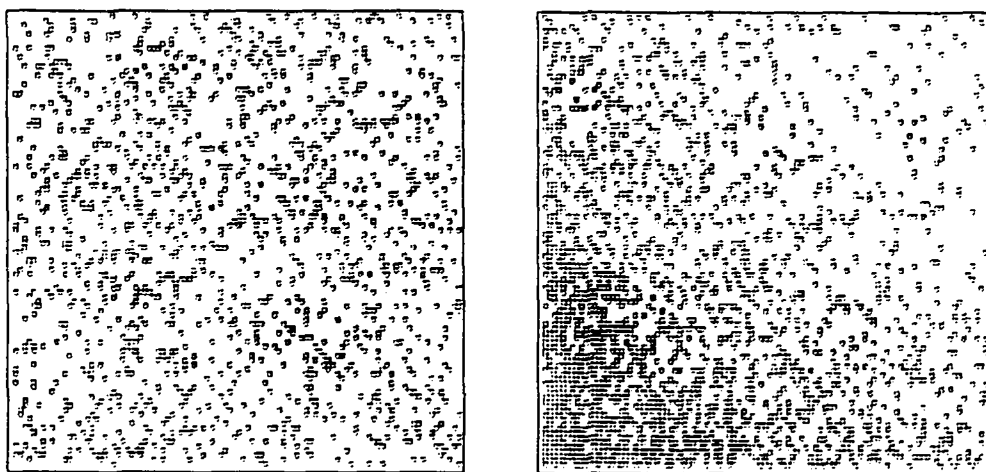

Fig. 2. Uniform and exponential distributions used in examples.

On choosing the parameters. In an unsupervised learning method, be it GAR, or the self-organizing map or another, there is one subtle task, which is choosing the parameters of the process. There are a number of parameters that are interdependent and that need to be tuned to get good results and they depend on the application. They do not have very wide range of possible values; some can even be fixed, for example, k and a are always fixed to 0.5 and 0.8 respectively. By decreasing T_{MIN}, one changes the number of features extracted. M, duration of sleep, is equal to the size of the training set. τ_{MAX}, trophy limit, and γ, threshold gain factor, determine the number of iterations to be made. Having a large γ and small τ_{MAX} one can converge very fast. However note that as feature extraction is a statistical process, one is advised to carry out a large number of iterations. The approach taken in the examples in this chapter is to use a relatively large trophy limit and small gain for good convergence. This prudent approach by making a lot of iterations will always work. One can play with the parameters to achieve fast convergence however such special

parameter combinations will be application dependent. In the case of one or two dimensional signals, one can display the input signal and the features on screen and by comparing them, evaluate if a good approximation has been made. When one passes to high dimensions however, such a control is no longer possible and one can only be advised to be patient.

The parameters are set as follows.

k	0.5	Elimination factor during sleep
a	0.8	Update factor
M	10,000	Duration of sleep
T_{INIT}	0.8	Initial threshold
T_{MIN}	0.2	Final threshold
τ_{MAX}	100	Trophy limit
γ	0.02	Threshold increase factor

The regions of domination evolve as shown in Fig. 3. Each texture denotes domination region of a different unit; 'x's denote unit positions. White are the points which are not dominated by any unit; if they are encountered, units will be added there. Notice the hyperplane boundary when units are close, and circular regions when they are sufficiently apart, i.e., distance between two units is greater than the sum of their radii.

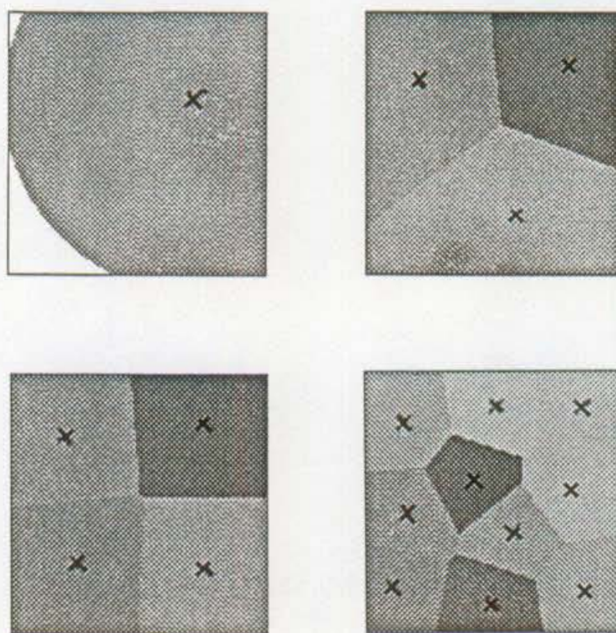

Fig. 3. Evolution of unit positions and domination regions with GAR for uniform distribution. From top-left to bottom-right, after 500, 1,000, 6,500, and 77,000 iterations.

For the exponential distribution, regions change in time as in Fig. 4.

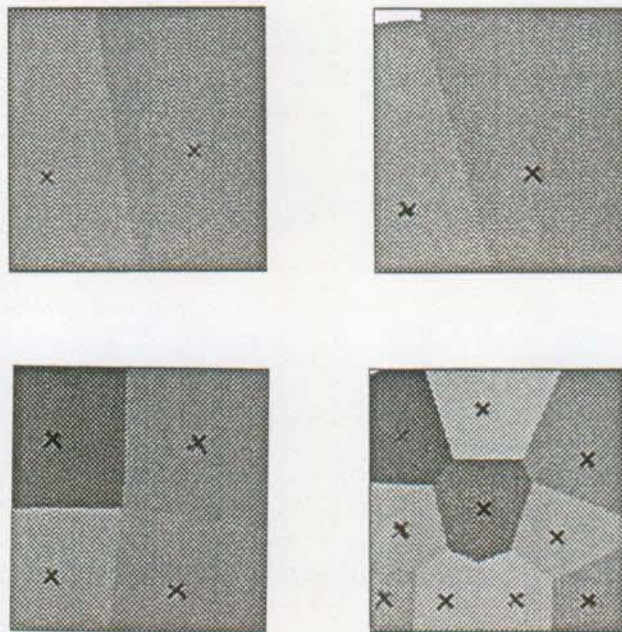

Fig. 4. Evolution of regions GAR for exponential distribution. After 500, 1,000, 6,500, and 77,000 iterations.

When distributions have different, even concave, shapes, GAR is able to learn them. Starting from Fig. 5, the evolution of GAR for square and cross-shaped uniform distributions with uniform and exponential densities are shown; parameters used are given below.

k	0.5	Elimination factor during sleep
a	0.8	Update factor
M	4,000	Duration of sleep
T_{INIT}	0.5	Initial threshold
T_{MIN}	0.05	Final threshold
τ_{MAX}	100	Trophy limit
γ	0.02	Threshold increase factor

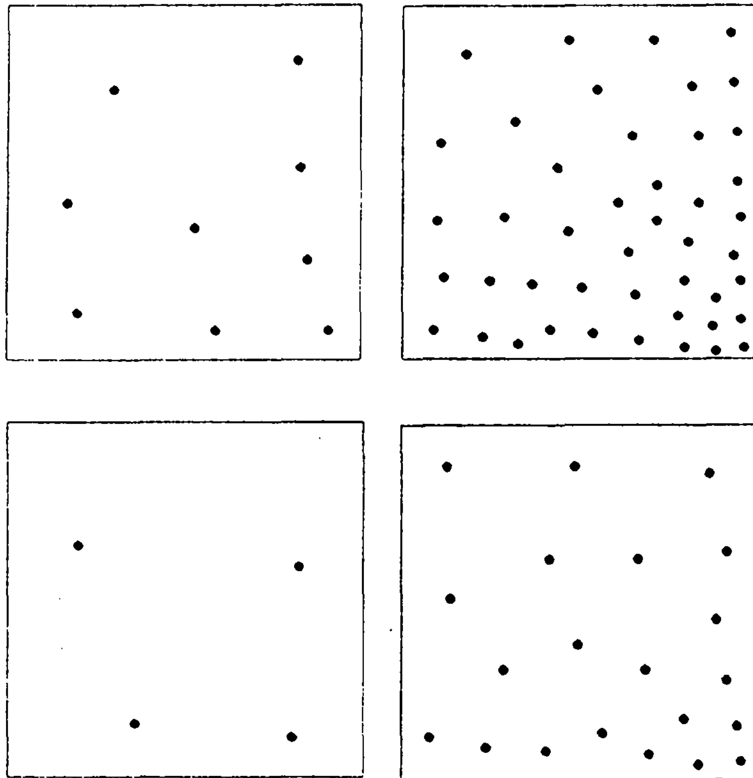

Fig. 6. Exponential square-shaped distribution. After 1,500, 20,000, 70,000, and 220,000 iterations.

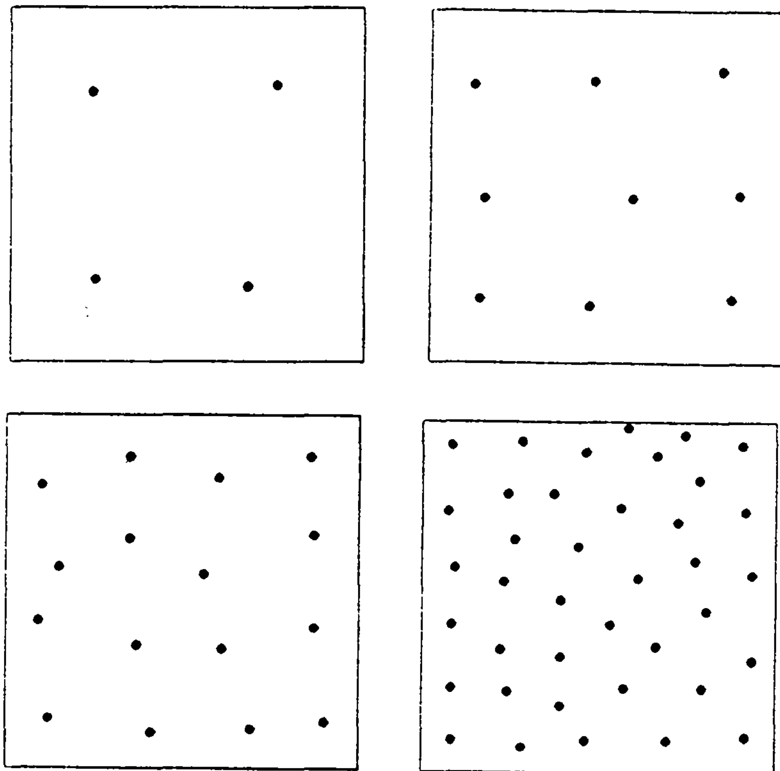

Fig. 5. Uniform square-shaped distribution. After 4,000, 20,000, 60,000, and 150,000 iterations.

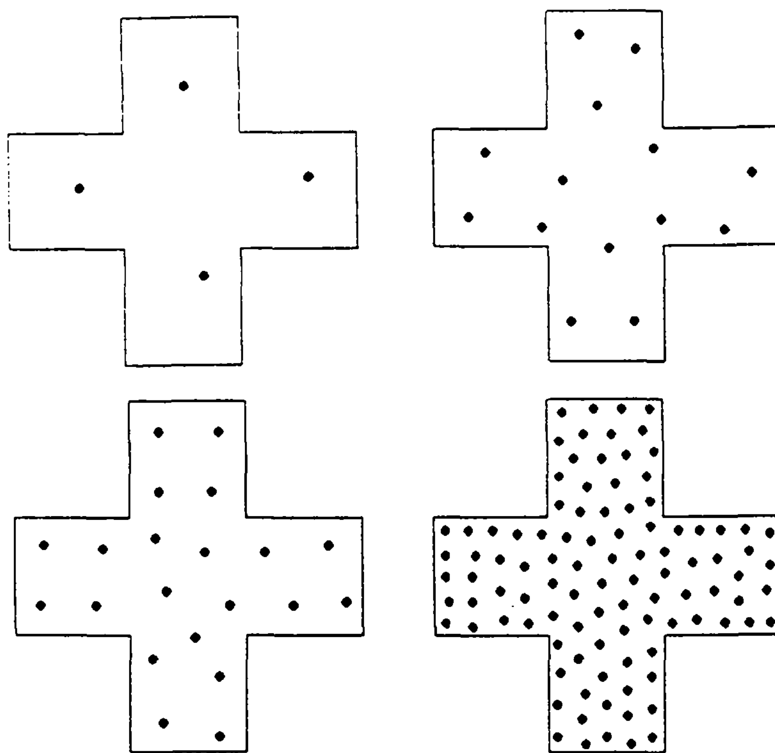

Fig. 7. Uniform cross-shaped distribution. After 1,500, 38,000, 90,000 and 500,000 iterations.

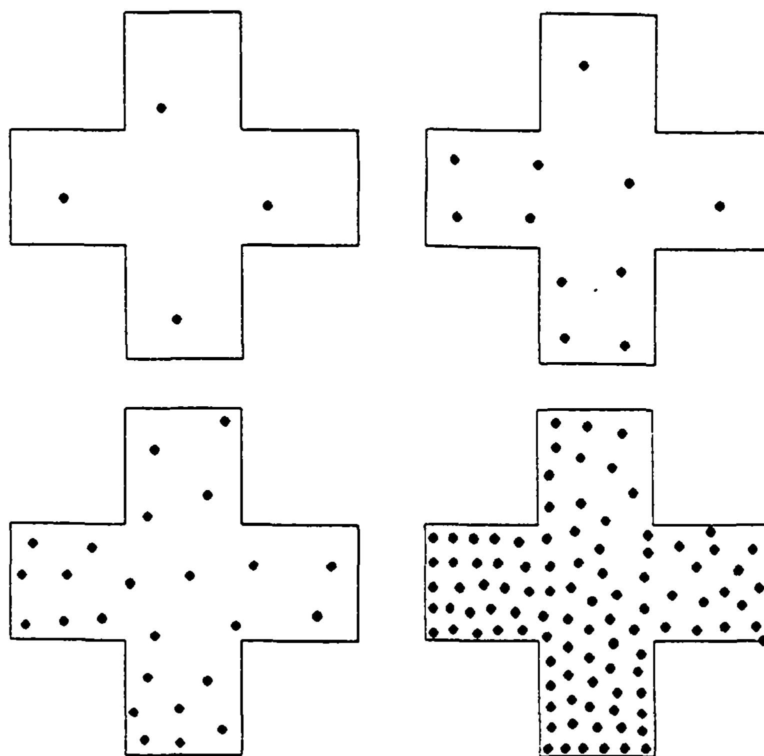

Fig. 8. Exponential cross-shaped distribution. After 1,500, 36,000, 97,000 and 500,000 iterations.

3.5. COMPARING GAR WITH THE SELF-ORGANIZING MAP

GAR is compared with Kohonen's self-organizing map (Kohonen, 1982), the most popular unsupervised learning algorithm actually. Input signal is two dimensional, distributions are uniform or exponential like in the above case. Distributions have two different shapes: square and cross. The number of units are manually defined. Parameters of the self-organizing map are set as follows:

Size of map	Two dimensional: 12 by 12
Initial update factor	0.99
Rate of decrease of update factor	0.9999
Initial neighborhood	12
Rate of decrease of neighborhood	0.9999
Mexican-hat function	Linearly decreasing proportional to distance

For the distributions and parameters above, one achieves the following results with the self-organizing map.

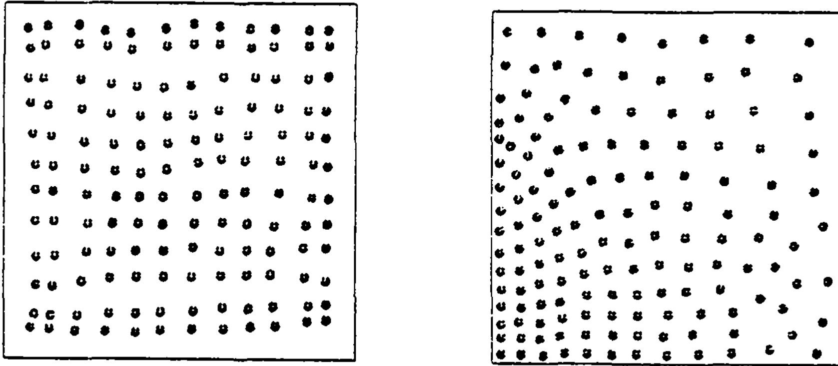

Fig. 9. For uniform and exponential square-shaped distributions, results by Kohonen's self-organizing map. After 50,000 iterations.

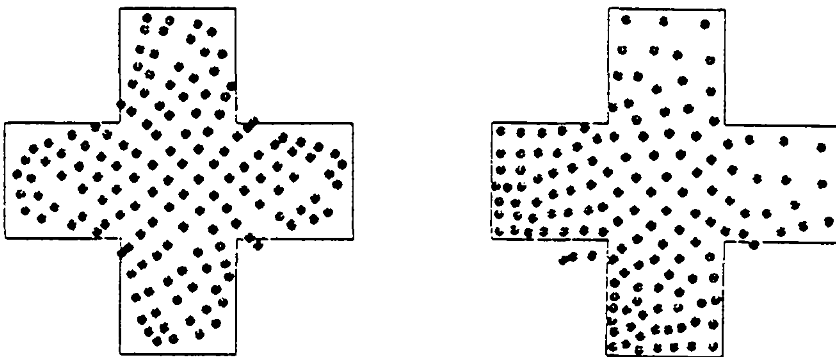

Fig. 10. For uniform and exponential cross-shaped distributions, results by Kohonen's self-organizing map. After 50,000 iterations.

One notices with the cross-shaped distribution that to be able to conserve topology, the self-organizing map needs to allocate practically unnecessary units where there are "gaps." GAR does not has this disadvantage as no initial space is defined. In a map an initial space is defined, here having a rectangular shape, which is later distorted to fit the actual space. There is no such predefinition in GAR.

3.6. DEVELOPMENT OF FEATURE DETECTOR CELLS BY GAR

In this section, I will talk about development of feature detector cells using GAR. The network after a process of unsupervised learning during which it is shown examples, extracts a set of features. These features are then tested by adding a Grow-and-Learn (GAL) layer and their quality is quantified in terms of:

[1] success achieved on the test set, and
[2] memory requirements, i.e., number of exemplars stored.

The self-organizing map is also used to extract features and compared with GAR. Various post-processing techniques are tested before classification.

The application is recognition of handwritten numerals with the same database used in the previous chapter. Input to GAR network is 5 by 5 windows extracted from 24 by 24 binary images, "centered" versions of 16 by 16 images as explained previously. Distance measure used is the Euclidean distance.

The parameters of GAR are set as follows:

Parameter	Value	
T_{INIT}	0.150	Initial threshold value
T_{MIN}	0.125	Final threshold value
a	0.5	Update factor
γ	0.01	Factor for increasing the threshold
τ_{MAX}	400	Trophy limit
k	0.5	Factor used in sleep

A relatively big threshold limit, T_{MIN}, is used to get a small number of masks. The masks shown in Fig. 11 are generated after 5 epochs where each epoch includes 5 awake and 1 sleep passes over the data set.

Thus 4 units were generated and each unit effectively becomes a detector of one of the features shown above. When an input vector is given, each unit computes the Euclidean distance between its weight vector and the input vector. With the same database, one gets the following features using Kohonen's self-organizing map (Fig. 12).

These features are replicated for all positions in the original 24 by 24 character image. Any 5 by 5 region is given as input to 4 feature detectors and thus for each pixel in the original image, 4 values are computed:

$$Act_f = 1.0 - \frac{\sqrt{\sum_{a=0}^{4}\sum_{b=0}^{4}(Pix_{ab} - W_{fab})^2}}{5}, \quad f = 1\ldots4. \tag{3.10}$$

Pix_{ab} is the 5 by 5 region extracted from character image, W_f^T is the weight vector of the fth feature detector. Act_f is 1.0 when they are the same and decreases till 0.0 as they get further apart. Thus the total 24 by 24 image is then represented by $24 * 24 * 4 = 2304$ numbers. It is this vector that is given as input to a supervised learning algorithm, here GAL.

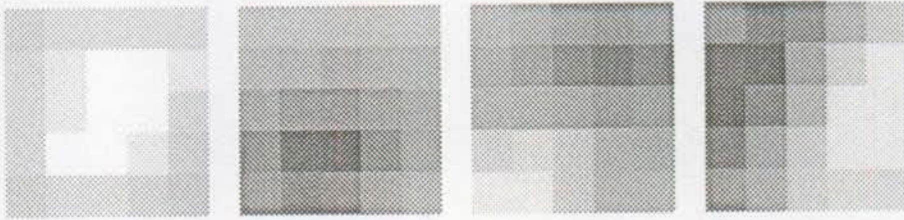

Fig. 11. Features learned by GAR.

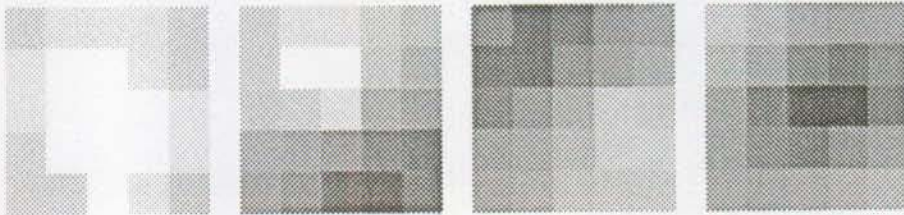

Fig. 12. Features learned by a one dimensional self-organizing map.

3.6.1. Testing learned features with GAL

Using GAL, one performs a number of *awake* passes, generally two or three, till no more units are added then a *sleep* pass is performed to eliminate unnecessary exemplar units. One iterates this process till no more additions or deletions are possible. This is done 20 times as GAL depends on the order of vectors in the training set by using a different random order at each trial. Best and worst cases and the average is given together for the number of exemplars stored by GAL, and the success on test set. Success on test set with the nearest-neighbor method (NN) is also given. This is done both for features extracted by GAR and those extracted by the self-organizing map.

[1] With masks extracted by GAR.

	NN	Best	Average	Worst
Units	600	68	67.7	65
Success	99.2	97.8	96.3	93.2

[2] With masks extracted by the self-organizing map.

	NN	Best	Average	Worst
Units	600	75	70.1	71
Success	99.2	97.7	96.1	95.2

Success with the nearest-neighbor method without any features is 96.7; the fact that this value increased imply the utility of the feature extraction process.

3.6.2. Post-processing the output of feature detectors

In the approach mentioned above, the input vector to GAL, as mentioned, has 3456 dimensions with each element requiring many bits to be represented. This is too costly both in terms of memory and computation. One can perform some post-processing to simplify the representation of the input vector. There are various possibilities in calculating the output of units, O_f, from their activations, A_f.

A. The quite commonly used one is to choose the highest activation and give to it the output value 1 and 0 to all others. Thus feature detectors between themselves are organized as a winner-take-all network and the feature closest, m, becomes the winner.

$$A_m = \max_i A_i. \tag{3.11}$$

$$\begin{aligned} O_m &= 1, \\ O_i &= 0, \quad \forall i \neq m \end{aligned} \tag{3.12}$$

This has the advantage that less precision, i.e., 1 bit, is required to code each output. Thus the output for each position is 4 bits and the total input vector is 2304 bits long. The problem however is that one loses information because wherever the vector is in that unit's domination region, the same output vector is given out. One can no longer do any interpolation. This problem may partially be solved by using a lot of units thus decreasing the sizes of domination regions.

[1] With masks extracted by GAR.

	NN	Best	Average	Worst
Units	600	119	123.2	122
Success	95.3	94.7	92.4	90.5

[2] With masks extracted by the self-organizing map.

	NN	Best	Average	Worst
Units	600	138	132.5	126
Success	94.8	92.2	89.7	87.7

B. Another possibility is to choose a small number of units that have the highest activation, use their values and set all others to 0 (Hecht-Nielsen, 1988). For example, when one takes only the two highest, m and n, the vector can be computed as follows:

$$\begin{aligned} A_m &= \max_i A_i, \\ A_n &= \max_{i \neq m} A_i. \end{aligned} \tag{3.13}$$

$$\begin{aligned} O_m &= 1.0, \\ O_n &= 0.5, \\ O_i &= 0.0, \quad \forall i \neq m \wedge i \neq n. \end{aligned} \tag{3.14}$$

The highest takes the value 1.0, and the second highest 0.5, all the rest are set to 0.0. Now as any feature output can take one of three different states, two bits are necessary. Thus total GAL input requires a vector $2403 * 2 = 4806$ bits long.

[1] With masks extracted by GAR.

	NN	Best	Average	Worst
Units	600	90	85.4	81
Success	97.8	97.2	95.0	93.5

[2] With masks extracted by the self-organizing map.

	NN	Best	Average	Worst
Units	600	90	80.9	79
Success	98.8	96.7	95.4	93.8

Overall, the two algorithms, GAL and the self-organizing map, seem to have performed equally well.

3.7. A CRITIQUE OF GAR

As a result of using GAR, the distribution of the synaptic weight vectors of the units approximate the distribution of the input signal. Input values may be binary or analog. In "tie-cases" only one unit should be modified.

GAR has too many parameters that need to be tuned to get good features. This, however, seems to be a general property of unsupervised learning algorithms as, for example, the self-organizing map also has a lot.

The dynamically growing structure of the network allows to have a network whose size is a function of the signal. The number of units used is not fixed and pre-defined by the programmer but depends on the T_{MIN} value which indicates how close the units are allowed to be with respect to each other. Thus the network size depends on the signal and not on the programmer's intuition. The idea of adding a new unit when an input is different from the stored, also exists in the ART. In their model, a vigilance parameter fixed and same for all units, controls how large clusters, i.e., boxes or domination regions, are allowed to be. The problem with ART however is that it does not take into account densities, thus there will be information loss during feature extraction.

As there is no modification of neighbors, there is no topology defined between units, thus GAR's output is not a topological map. This may be a disadvantage if a final topology is required. One can use an approach as proposed in (Rodrigues & Almeida, 1990) to have an incremental learning of a topological map.

However, the advantages of not having a topology are that, first, as no neighbors are modified, less computation is done and second, no border effects are encountered. What is more, when concave shapes are used, i.e., gaps in the distribution, a topological map wastes units in trying to "bridge" these gaps, which is not the case with GAR. This phenomenon can also be noticed in the cross-shaped distribution given in the above example. In the case of GAR, there is no initial space defined which needs to be modified to fit the input signal.

One is of course free to have multi-layer GAR networks to be able to extract still higher-order features. In this case, one should perform training of the first layer and once it is over, can pass to the training of the second layer, etc. One *can* train the two layers at the same time, however, the second order features learned before the first order features have converged will not be of any use so it is faster to train layers one at a time.

The idea of eliminating units with low trophies allows the network to re-organize itself when the distribution of the signal changes in time. In a Kohonen self-organizing map, this requires increasing the gain and if the Mexican-hat shrinks in time, re-enlarging it. The novel property of GAR is that if the signal changes in time so that the salient features change, GAR is able to forget automatically the old ones learning the new.

A multi-layer network made of a layer of GAR for feature extraction followed by a layer of GAL for classification gives interesting results in terms of success. Such a scheme is rather similar to the counter-propagation network (Hecht-Nielsen, 1988) and can be a good alternative to the back-propagation of the gradient. Several GAR layers may be used before GAL to extract higher-order features. To be able to decrease memory and computational cost, one may post-process output of feature detectors to code feature activations using less bits.

4

Conclusions

i mean that the blond absence of any program
except last and always and first to live
makes unimportant what you and i believe;
nor for philosophy does this rose give a damn ...

since the thing perhaps is
to eat flowers and not to be afraid.

— *e.e. cummings.*

4.1. A BIOLOGICAL VIEW

There is no evidence that neurones are generated post-natally. So, dynamically adding new units has no biological plausibility. The brain is built according to some genetic program with an abundant number of cells. The neurons are generated at an average rate of 250,000 per minute *in utero* (Cowan, 1979). This proliferation may be succeeded by a migration process where young neurons migrate from one part of the brain to another. Finally, they settle down, maturate, specialize, and form synapses (Kandel & Schwartz, 1985).

This initial redundant structure loses then between 15% and 85% of its components. This phenomenon of "cell death" takes place both during embryonic and post-natal days. Cell death before birth is an intrinsic phenomenon; the criteria to decide which one to eliminate are genetic if we are to assume that an embryo cannot experience anything. Post-natal cell death depends on experience where the structure during a *critical development period* is tested against the environment and fine-tuned or polished—the word "sculptured" is also used (Cowan, 1979)—to better match the environment. It seems like the genetic plans foresee an environment with some variance and according to what is really encountered during this short period just after birth, the unnecessary parts are eliminated. This is not that much different from starting from nothing and adding when necessary. To make a statue, one, when using clay, adds clay; when using marble, one removes.

Before cell death starts, the majority of the axons have reached their target fields and have just started establishing connections. The fact that these two phenemona overlap, suggests that there is some sort of a feedback process "back-propagated" from the axons to the soma—a retrograde transport of a "trophic," i.e., nourishing, substance which probably is glia-derived when the axons are growing (before birth) and driven by the activity of the target cells once the synapses are formed.

In the case of post-natal development, the utility criterion by which relevance of neurones are assessed is related to the functional activity of the cells in the target field on which synapses were formed—retrograde maintenance modulated by activity (Clarke, 1985). If the target field is destroyed, the cell death increases to around 100% and if it is artificially extended, death proportion decreases (Cowan *et al.*, 1984).

Not only the cells die, but also the synapses are eliminated during development. Although the dendritic branching of a neuron is determined genetically, most neurons seem to generate many more processes than are needed or than they are subsequently able to maintain. There is a phase of process elimination during which many (and in some cases all but one) of the initial group of connections are withdrawn (Cowan, 1979).

The creation of many and then elimination of most, naturally brings into mind the idea of "competition" and "selection of the fittest." As early as 1881 Roux "suggested that cells, including neurons, may be involved in a Darwinian struggle for survival during development" (Clarke, 1985).

Dawkins proposed (1971) selective neuron death as a possible memory mechanism. My convinction is that cell death during the critical period is to form the *central* pathways, which thus require structural modification. Memory, achieved through everyday learning, however, can better be formed through parametric modification, i.e., of synaptic efficacies.

Pondering on this, one may relate post-natal synapse elimination and cell death to each other:

- There should be a tendency for a cell to have as small a number of synapses on its dendrites as possible. This corresponds to saying that *the condition that should be satisfied to make a cell fire is tried to be kept as simple as possible.*
- A synapse's fate is a function of the activities of the pre- and post-synaptic cells. A neuron by itself cannot fire another cell; a cell needs coincident activity from many cells to fire (Crick & Asanuma, 1986). For a synapse to be considered "useful," the pre- and post-synaptic cells activities should in time be similar, their firing should coincide, i.e., be correlated. It was proposed that (Schmidt & Tieman, 1989) it is not the activity level *per se*, but the correlation of pre- and post-synaptic activity that leads to synaptic stabilization—the so called Hebbian law. A synapse for which this correlation is not high, is susceptible to be eliminated.

- A cell which loses most of the synapses on its axons, is also likely to die. Although no experiments have yet addressed the relationship between activity dependent branch elimination and cell death, it was proposed that (Schmidt & Tieman,1989) branch elimination might occur first and lead to cell death in those cases where the number of branches falls below a critical number.

The synaptic learning method used both in GAL and GAR, is related to the proposal of Hebb (1949).

It was initially proposed by Crick and Mitchison (1983) that some sort of a "reverse learning" to get rid of parasitic memory traces occurs during Rapid Eye Movement (REM) sleep when dreaming occurs. The idea basically is that, the system is closed to its environment, e.g., sleep, inputs are generated by the system itself, e.g., dreaming, and unwanted modes of behaviour emerging due to accumulation of experience are eliminated during an active process of unlearning. It is said that "we dream in order to forget."

When applied to artificial neural networks, their proposal is based on the assumption that in networks where many associations are distributed, parasitic confabulation stable states, i.e., local minima that do not correspond to any vector in the training set, exist which decrease the quality of the network as an associative memory. When applied to Hopfield's model, (Hopfield *et al.*, 1983), random vectors were used as inputs to the system and a Hebbian learning with a negative factor was applied to increase the energy of such states. The same idea later with Boltzmann machine was applied by Hinton and Sejnowski (1986) which they call the *phase*$^-$.

In GAL and GAR, forgetting can be done more efficiently than these two models. First, because the vectors are stored locally, the system knows exactly what to unlearn. The second advantage is that, because the weights of units are not distributed, the system can "forget" at one shot; no iterative procedure is necessary. I believe that if some sort of an unlearning *does* occur during sleep, it cannot be with random inputs. Dreams although not perfectly logical, are not completely random either; dreams are not hallucinations. Jouvet's suggestion that "species-specific behaviours are rehearsed during sleep governed by a genetical preprogram" (in Kandel & Schwartz, 1985)[1] seems more convincing to me. We make "simulations" of real life during sleep, whose aim is rather similar to military manoeuvres. This we can also think of as a sort of off-line learning.

[1] *Jouvet, M. (1983) "Neurophysiology of dreaming," in M. Monnier and M. Meulders (eds.) Functions of the nervous system, 4, Psycho-Neurobiology, Amsterdam:Elsevier, pp. 227-248.*

4.2. INCREMENTAL LEARNING

Incremental learning implies having a strategy which includes not only the modification of the system parameters but also the system structure by which one has a larger scope of adaptation. Parametric adaptation whereby only parameters can be modified is limited to how these parameters are defined, structural modification which can modify the system structure as well has no such limitation.

In the case of neural networks, parametric adaptation is the usual approach which involves modification of the synaptic weights only. As there is no rule whereby one can compute the necessary network structure given a training set or application, the usual approach is one of trial-and-error. A more recent approach is to start with a large one and "prune" those units that are not very relevant based on a certain criterion of relevance.

Structural adaptation, named incremental learning, implies a dynamic network whose structure can also be modified by adding or deleting units and links. As such a learning method has a larger scope than weight modification of a static network, it is more promising as growing recent interest in such networks point out.

To make the idea clearer, let me take the common analogy made between curve fitting and learning. In curve fitting, one is given a set of points and is asked the equation of the polynomial that gives a good fit to these set of points. The aim then is to be able to interpolate for points whose values are not previously given. In classification, the problem is similar, one is given a set of patterns labelled as members of this or that class and then is asked to somehow find class definitions based on these examples so that, when later unlabelled patterns are presented, the system will be able to say to which class they belong.

There are two problems in curve fitting, the first is that of finding the order of the polynomial and second the constant factors once the order is defined. For example, when the polynomial is $y = ax^2 + bx + c$, the order is 2 and the factors are a, b, and c. In parametric modification, one assumes a certain order and computes the best values of these factors that minimizes error. In structural modification, one can also play with the order of the polynomial by adding or deleting terms.

As in any scientific endeavour, one wants to find out the simplest definition possible. Thus in structural modification, one starts with the lowest order polynomial and checks if a good fit can be made with it, if not the order of the polynomial is increased and a good fit is checked for, etc. till a reasonable fit is reached.

In the case of learning classes, one learns the discriminants that seperate the members of one class from all others. When a fixed network structure is chosen where only the synaptic weights are modified, one assumes a certain order of the hypersurface where only the factors of the hypersurface are modified. In the case of a network, the order is given by the number of hidden layers. The factors correspond to the synaptic weights. The necessity to perform thousands of iterations to learn a set of associations by tuning weights is generally accepted to be the main drawback of neural models. In the case of structural learning, the complexity of the class discriminants also get modified in time. It is the environment that shapes the network.

GAL and GAR have their sections where they are compared with other learning algorithms to get an idea of their relative advantage and disadvantages. In the following sections, I will just stress a few general points.

4.3. GAL

GAL is an incremental algorithm to learn classes from examples. It learns very fast; each pattern takes one iteration to learn. This is its basic advantage and makes it interesting for applications where patterns need to be learned on-line, in real-time.

Basically what is envisaged is to have a *dynamic* network structure where the structure gets modified as a function of the error. As is said, it is the environment, i.e., the training set, that shapes the network. This opposes sharply to learning algorithms where the network has a *static* structure whose synaptic weights are modified as a function of the error. The problem with having a static structure is that of being able to correctly guess the best structure before learning. To my knowledge, there is no method by which one can compute the required network structure, i.e., number of hidden layers, units and their connectivities, from a given data set. There *are* some very general rules related to the shape of class separations one can achieve. For example, if there is no hidden layer, classes need to be linearly separable. With one hidden layer, one can learn convex shapes. One can separate classes two by two using one hidden unit for each pair. With two hidden layers, one can also learn concave shapes. The problem however is that when one is given a data set, one does not know the shapes of classes and the approach is that of trial-and-error which depends to a great extent on designer's knowledge of the task and his experience.

In the case of GAL, there is only one layer because a winner-take-all type of non-linearity is powerful enough to perform any kind of separation. It is the number of units in this layer that is important and that changes.

Learning in GAL involves either allocating a unit or not, and once a unit is committed its weight vector is no longer modified. Thus there is no parameter finetuned during a statistical learning process. The first advantage of this is that learning time is less. Second, to store synaptic weights one needs as much precision as needed for the input vector. For example as seen in chapter 2, one needs binary weights to learn binary images. Although it is a variant of the nearest-neighbor method, it is in this regard, namely by requiring a lot less memory, better than the nearest-neighbor. Of course, as the network gets smaller, the computational complexity decreases, and, on a sequential computer, so does the recognition time. The *sleep* mode allows getting rid of units that are no longer necessary which allows a further decrease of the network size.

In the case of RCE, although weight vectors are not modified either, there is a modifiable threshold associated with each unit which needs to be calibrated as a function of the error. This threshold needs also to be stored and operated on with high enough precision. The domination regions in RCE are hyperspheric and thus is symmetric around each unit which may be disturbing. GAL, due to the winner-take-all non-linearity, can have regions that are not symmetric.

In LVQ, the weight vector of the closest reference vector is modified with a small factor. This implies higher precision for the weight vectors and longer learning time. Another nuisance of LVQ is to determine the number of reference vectors needed as it is pre-determined and fixed.

In discriminant-based methods, the first problem is the determination of the necessary network structure, i.e., number of hidden layers, hidden units, connectivities, etc. once a problem is given. This is generally done based on trial-and-error as the programmer gains more intuition related to the task. Because all the weights are modified at each learning iteration, such methods are slow. As the update factor is also less than 1, one also needs high precision to store weight vectors and to perform computations with them.

In algorithms that are similar to nearest-neighbor, the percentage of the training set that needs to be stored as exemplars depend on the dimensionality of the input vector, the so-called "curse of dimensionality." As the number of dimensions of the input vector goes high, more exemplars are stored.

The second factor that influences the percentage of the patterns stored is the quality of the representation. The better is a representation, the less are the exemplars stored. Better representations can be achieved by preprocessing or feature extraction. Especially preprocessing improves success and minimizes memory required without slowing down learning time.

From the point of view of success, GAL is interesting. Although its success is not as high as discriminant-based methods, when preceded with a preprocessing or feature extraction layer, GAL compares easily with such methods also in terms of success as experiences have shown. Preprocessing or feature extraction also gives GAL immunity to noise which it does not have otherwise as exemplars once committed, are not modified afterwards.

Despite the fact that GAL is for learning of categories, it can easily be modified to learn also continous mappings. One needs to discretize the range into small segments with a required precision and then use GAL as if each of these segments is a different class. It is shown that this leads to a very rapid learning of such mappings.

4.4. GAR

GAR is an incremental algorithm to learn features from examples. As feature extraction is a statistical process, to converge to good results, one needs to perform a large number of iterations. The aim is to approximate as well as possible the probability density of the input signal.

The advantage of GAR with respect to Kohonen's self-organizing map is that there is no initial space defined that needs to be fit to the input space. There is no topology defined between units, thus GAR's output is not a topological map. The advantages are that, to the first, no border effects are encountered and to the second, when the input space is not convex, no units are wasted in trying to bridge the gap.

One interesting property that GAR has, thanks to its *sleep* mode, is that it allows the probability density of the input signal to change in time. That is, when the salient features change in time, GAR is able to pass to a new set eliminating automatically the old.

GAR and GAL can be combined to perform feature extraction and classification. In such a case, first the GAR layer is trained to extract features, GAL layer is then trained based on the output of the GAR feature detectors. One can have more than one GAR layer before GAL to be able to extract higher order features. The advantage of post-processing the output of feature detectors before classification has also been shown. Such a combination, namely a GAR layer for feature extraction followed by a GAL layer for classification is similar to the counter-propagation network as proposed by Hecht-Nielsen (1988) and stands out to be a good alternative to the currently most popular back-propagation algorithm.

4.5. FUTURE DIRECTIONS

GAL and GAR in this dissertation are applied to the recognition of handwritten numerals only. They should now be used for other tasks, with signals of other modalities, so that one will be able to better evaluate their scope of usage. For example in storing and recognizing temporal relations, a time delayed version of GAL can be used, although how well, we do not yet know. Similarly in applications related to robotics where real-time learning is necessary, it will be interesting to try and use a GAL, or GAR+GAL, and see what it brings at what cost.

One possible research direction is to extend GAR to the level of individual connections. When a GAR network is trained, one predefines the size and position of the receptive fields. It will be rather useful if one can start from highly redundant, big receptive fields and then somehow compute relevance of individual connections and then eliminate those who are not necessary, in a way as to automatically find good receptive field sizes *and* good features. Whether the usual Hebbian dynamics, in terms of the covariance of the pre- and post-synaptic signals, is sufficient, I do not know.

References

[0] Ackley, D.H. (1987) *A connectionist machine for genetic hillclimbing*, Kluwer Academic Publishers.

[1] Alexander, I. (1990) "Ideal neurons for neural computers," in R. Eckmiller, G. Hartmann, G. Hauske (eds.) *Parallel Processing in Neural systems and Computers*, North-Holland.

[2] Alpaydın, E. (1988) "Grow-and-Learn," *EPFL-LAMI, Internal note.*

[3] Alpaydın, E. (1990a) "Learning logic array," *Int. Joint Conf. on Neural Networks*, January, Washington, USA.

[4] Alpaydın, E. (1990b) "Grow-and-Learn: An incremental method for category learning," *Int. Neural Network Conf.*, July, Paris, France.

[5] Angéniol, B., de La Croix Vaubois, G., Le Texier, J.-Y. (1989). "Self-organizing feature maps and the travelling salesman problem," *Neural networks*, **1**, 289–294.

[6] Barlow, H.B. (1963) "The information capacity of nervous transmission," *Kybernetik*, **2**, 1 also in *Brain theory: Reprint volume*, G.L. Shaw, G. Palm, (eds.) (1988) World Scientific, 683.

[7] Barlow, H.B. (1989) "Unsupervised learning," *Neural Computation*, **1**, 295–311.

[8] Baum, E.B. (1989) "A proposal for more powerful learning algorithms," *Neural Computation*, **1**, 201–207.

[9] Carpenter, G.A., Grossberg, S. (1987) "ART2: Self-organization of stable category recognition codes for analog input patterns," *Applied optics*, **26**, 4919–4930.

[10] Clarke, P.G.H. (1985) "Neuronal death in the development of the vertebrate nervous system," *Trends in Neuroscience*, **8**, 345–349.

[11] Cottrell, G., Fleming, M. (1990) "Face recognition using unsupervised feature extraction," *Int. Neural Network Conf.*, Paris, July 1990.

[12] Cowan, W.M. (1979) "The development of the brain," *Scientific American*, **241(3)**, 106–117.

[13] Cowan, W.M., Fawcett, J.W., O'Leary, D.D.M., Stanfield, B.B. (1984) "Regressive events in neurogenesis," *Science*, **225**, 1258–1265.

[14] Crick, F.H.C., Asanuma, C. (1986) "Certain aspects of the anatomy and physiology of the cerebral cortex," in *Parallel distributed processing*, J.L. McClelland, D.E. Rumelhart (eds.) **2**, MIT Press, 333–371.

[15] Crick, F. Mitchison, G. (1983) "The function of dream sleep," *Nature*, **304**, 111–114.

[16] Dawkins, R. (1971) "Selective neurone death as a possible memory mechanism," *Nature*, **229**, 118–119.

[17] Dawkins, R. (1982) *The extended phenotype*, Oxford University Press.

[18] Denker, J., Schwartz, D., Wittner, B., Solla, S., Howard, R., Jackel, L., and Hopfield, J. (1987) "Large automatic learning, rule extraction, and generalization," *Complex Systems*, **1**, 877–922.

[19] DeSieno, D. (1988) "Adding a conscience to competitive learning," *IEEE Int. Conf. on Neural Networks*, San Diego, USA.

[20] Duda, R.O., Hart, P.E. (1973) *Pattern classification and scene analysis*, John Wiley and sons.

[21] Guyon, I., Poujoud, I., Personnaz, L., Dreyfus, G., Denker, J., and le Cun, Y. (1989) "Comparing different neural architectures for classifying handwritten digits," *Int. Joint Conf. on Neural Networks*, Washington, USA.

[22] Hebb, D.O. (1949) *The organization of behaviour*, Wiley. Chapter 4: "The first stage of perception: Growth of the assembly," Chapter 5: "Perception of a complex: The phase sequence," in *Brain theory: Reprint volume* G.L. Shaw, G. Palm, (eds.) (1988) World Scientific, 224–270..

[23] Hecht-Nielsen, R. (1988) "Applications of counterpropagation networks," *Neural Networks*, **1**, 131–140.

[24] Hinton, G.E., Sejnowski, T.J. (1986) "Learning and relearning in Boltzmann machines," in *Parallel distributed processing*, D.E. Rumelhart, J.L. McClelland (eds.) **1**, MIT Press, 282–317.

[25] Holland, J.H. (1975) *Adaptation in natural and artificial systems*, University of Michigan Press.

[26] Hopfield, J.J. (1982) "Neural networks and physical systems with emergent collective computational abilities," *Proc. Nat'l Acad. Sci.*, **79**, 2554–2558.

[27] Hopfield, J.J., Feinstein, D.I., Palmer, R.G. (1983) "'Unlearning' has a stabilizing effect in collective memories," *Nature*, **304**, 158–159.

[28] Jockush, S. (1990) "A neural network which adapts its structure to a given set of patterns," in *Parallel processing in neural systems and computers*, R. Eckmiller, G. Hartmann, G. Hauske (eds.), North-Holland.

[29] Kandel, E.R., Schwartz, J.H. (1985) *Principles of neural science*, 2nd edition, Elsevier.

[30] Knerr, S., Personnaz, L., Dreyfus, G. (1989) "Single-layer learning revisited: A stepwise procedure for building and training a neural network," *Neurocomputing: Algorithms, architectures, and applications*, Fogelman-Soulié, F. (ed.), NATO ASI Series, Springer.

[31] Kohonen, T. (1982) "Self-organized formation of topologically correct feature maps," *Biological Cybernetics*, **43**, 59–69.

[32] Kohonen, T. (1988) *Self-organization and associative memory*, 2nd edition, Springer.

[33] Kohonen, T., Barna, G., Chrisley, R. (1988) "Statistical pattern recognition with neural networks: Benchmarking studies," *IEEE Int. Conf. on Neural Networks*, San Diego, July.

[34] Kong, Y., Noetzel, A. (1990) "A training algorithm for a piecewise linear neural network," *International Neural Network Conference*, Paris, July 1990.

[35] le Cun, Y. (1985) "Une procedure d'apprentissage pour reseau à seuil," *Proceedings of Cognitiva 85*, 599–604, Paris.

[36] le Cun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W., Jackel, L.D. (1989) "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, **1**, 541-551.

[37] Lippman, R.P. (1987) "An introduction to computing with neural nets," *IEEE ASSP mag.*, **4**, 4-22.

[38] Mach, E. (1883) "The economy of science," *The science of mechanics* also in *The world of mathematics*, J.R. Newman (ed.) (1988) New edition, Tempus books, **3**, 1759–1767.

[39] Marr, D. (1982) *Vision*, Freeman.

[40] McCulloch, W.S. (1965) *Embodiments of mind*, MIT Press.

[41] Mézard, M. (1989) "Learning algorithms in layered networks," *NATO ARW on Neurocomputing*, February, Les Arcs, France.

[42] Mozer, M.C., Smolensky, P. (1989) "Skeletonization: A technique for trimming the fat from a network via relevance assessment," *Connection Science*, **1**, 3–26.

[43] Oja, E. (1982) "A simplified neuron model as a principal component analyzer," *Journal of mathematical biology*, **15**, 267–273.

[44] Pao, Y.-H. (1989) *Adaptive pattern recognition and neural networks*, Addison-Wesley.

[45] Personnaz, L., Guyon, I., and Dreyfus, G. (1987) "High-order neural networks: Information storage without errors," *Europhysics Letters*, **4**, 863–867.

[46] Reilly, D.L., Cooper, L.N., and Elbaum, C. (1982) "A neural model for category learning," *Biological Cybernetics*, **45**, 35–41.

[47] Rodrigues, J., Almeida, L. (1990) "Improving the learning speed in topological maps of patterns," *International Neural Network Conference*, Paris, July 1990.

[48] Ross, S.M. (1987) *Introduction to probability and statistics for engineers and scientists*, Wiley.

[49] Rumelhart, D.E., Hinton, G.E., Williams, R.J. (1986) "Learning internal representations by error propagation," in *Parallel distributed processing*, D.E. Rumelhart, J.L. McClelland (eds.), **1**, (1986) MIT Press, 318–362.

[50] Rumelhart, D.E. Zipser, D. (1986) "Feature discovery by competitive learning," in *Parallel distributed processing*, D.E. Rumelhart, J.L. McClelland (eds.) **1**, (1986) MIT Press, 151–193.

[51] Sanger, T.D. (1989) "Optimal unsupervised learning in a single-layer linear feedforward neural network," *Neural Networks*, **2**, 459–473.

[52] Schmidt, J., Tieman, S.B. (1989) "Activity, growth cones and the selectivity of visual connections," *Comments on Developmental Neurobiology*, **1**, 11–28.

[53] Specht, D. (1990). "Probabilistic neural networks," **3**, 109-118.

[54] Valiant, L.G. (1984) "The theory of the learnable," *Communications of the ACM*, **27**, 1134–1142.

[55] White, H. (1989) "Learning in artificial neural networks," *Neural Computation*, **1**, 425–464.

[56] Widrow, B., Hoff, M.E. (1960) "Adaptive switching circuits," *1960 IRE WESCON convention record*, 96–104 also in *Neurocomputing: Foundations of research*, J.A. Anderson, E. Rosenfeld (eds.), (1988) MIT Press, 126–134.

Curriculum Vitae

Name	A. I. Ethem Alpaydın
Sex	M
Date of birth	23/6/1966
Nationality	Turkish
Marital status	Single
Professional address	Laboratoire de microinformatique Ecole Polytechnique Fédérale de Lausanne IN-F 1015 Lausanne Switzerland
Private address	Av. du Mont d'or 15, 1007 Lausanne Switzerland
Scholar curriculum	
	(1983-1987) B.S. degree in Computer Engineering from Boğaziçi University Istanbul, Turkey
	(1988) Postgraduate course in Computer Science from EPF Lausanne, Switzerland
Professional Curriculum	
	(1984-1986) Part-time programmer in Bilpaz A.Ş. Istanbul, Turkey
	(1986) Part-time programmer in Eltek A.Ş. Istanbul, Turkey
	(1987-) Assistant in LAMI–EPFL Lausanne, Switzerland

Publications

[0] Alpaydın, E. (1988). "Distributed representation and associative storage of knowledge," *Annales du groupe CARNAC, EPFL*, **1**, 41-50.

[1] Alpaydın, E. (1988). "Optical character recognition using artificial neural networks," unpublished postgraduate course project report, *Cours postgrade en informatique technique*, Département d'Informatique, EPFL.

[2] Alpaydın, E., Marchal, P. (1989). "Snark: A neural optical character reader," in Pfeifer, R., Schreter, Z., Fogelman-Soulié, F., and Steels, L. (eds.) *Connectionism in Perspective*, 309-315, Amsterdam: North Holland.

[3] Alpaydın, E. (1989). "What is a feature that it may define a character, and a character that it may be defined by a feature ?" *NATO Advanced Research Workshop on Neurocomputing*, February, Les Arcs, France.

[4] Alpaydın, E. (1989). "Why an '**A**' is an '*A*' ? " presented at *European Congress on System Science*, October, Lausanne, Switzerland, (1989) at *Journées d'Electronique EPFL*, October, Lausanne, Switzerland (Invited Paper), and (1990) abstract to appear in *Artificial Intelligence Abstracts*.

[5] Alpaydın, E. (1989). "Optical character recognition using artificial neural networks," *IEE Int. Conf. on Artificial Neural Networks*, October, London, UK.

[6] Alpaydın, E. (1990). "Learning logic array," *Int. Joint Conf. on Neural Networks*, January, Washington, USA.

[7] Alpaydın, E. (1990). "Why are neural networks ? Will we have Neural Instruction Multiple Data (NIMD) machines ?" 4th *Annual Symposium on Parallel Processing*, April, Fullerton, USA.

[8] Alpaydın, E. (1990). "Grow-and-Learn: An incremental method for category learning," *Int. Neural Network Conf.* July, Paris, France.