Using Recurrent Networks for Dimensionality Reduction

by

Michael J. Jones

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE
in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1992

Using Recurrent Networks for Dimensionality Reduction

by

Michael J. Jones

Submitted to the Department of Electrical Engineering and Computer Science
on August 14, 1992 in partial fulfillment of the requirements
for the Degree of Master of Science in Computer Science

ABSTRACT

This thesis explores how recurrent neural networks can be exploited for learning certain high-dimensional mappings. Recurrent networks are shown to be as powerful as Turing machines in terms of the class of functions they can compute. Given this computational power, a natural question to ask is how recurrent networks can be used to simplify the problem of learning from examples. Some researchers have proposed using recurrent networks for learning fixed point mappings that can also be learned on a feedforward network even though learning algorithms for recurrent networks are more complex. An important question is whether recurrent networks provide an advantage over feedforward networks for such learning tasks. The main problem with learning high-dimensional functions is the curse of dimensionality which roughly states that the number of examples needed to learn a function increases exponentially with input dimension. Reducing the dimensionality of the function being learned is therefore extremely advantageous. This thesis proposes a way of avoiding the curse of dimensionality for some problems by using a recurrent network to decompose a high-dimensional function into many lower dimensional functions connected in a feedback loop and then iterating to approximate the high-dimensional function. This idea is then tested on learning a simple image segmentation algorithm given examples of segmented and unsegmented images.

Thesis Supervisor: Professor Tomaso Poggio

Title: Uncas and Helen Whitaker Professor, Dept. of Brain and Cognitive Sciences

ACKNOWLEDGEMENTS

I would like to thank my advisor, Tommy Poggio, for his encouragement and direction while working on this thesis. His energy and enthusiasm kept my research moving forward.

I would also like to thank Federico Girosi, John Harris, Jim Hutchinson and Charles Isbell for helpful discussions during the course of this work.

Finally, thanks to my parents for their support throughout the years.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. This research is sponsored by a grant from the Artificial Intelligence Center of Hughes Aircraft Corporation (LJ90-074) and by Office of Naval Research contract N00014–89–J–3139 under the DARPA Artificial Neural Network Technology Program. Support for the A.I. Laboratory's artificial intelligence research is provided by the Advanced Research Projects Agency of the Department of Defense under Army contract DACA76–85–C–0010, and in part by ONR contract N00014–91–J–4038. The author was sponsored by a graduate fellowship from the National Science Foundation.

Contents

Chapter 1

Introduction

Neural networks can be viewed as a framework for learning. In this framework, learning is a process of associating inputs to outputs. This abstract notion of learning can be made more concrete with a simple example. A child learning to catch a ball must use the visual inputs he is getting of the ball traveling toward him and learn to instruct his arms and hands to move to intercept and grasp the ball. In this case the inputs are visual images and the outputs are motor instructions, and the child must learn to produce the correct motor outputs for the visual inputs he receives. It is also desired that the learning process generalize to handle inputs that are not exactly like the ones already experienced. Under this view, learning is a matter of function approximation - finding a function that maps a given set of inputs to their corresponding outputs. Neural networks then are simply a technique for doing function approximation. A neural network (or simply a network) is a function containing parameters that can be adjusted in order to map inputs that are presented to the network to the desired outputs. The process by which the parameters are adjusted is called the learning procedure. The task for the network is to approximate the function that is partially defined by the input-output examples. Accepting this view of learning allows one to draw on the rich field of approximation theory. This is the viewpoint that will be adopted in this thesis.

There are two main classes of networks, feedforward networks and recurrent networks. Both types will be described in the next chapter. The main focus of this thesis is investigating recurrent networks for function approximation. Recurrent networks are more powerful than their feedforward counterparts, but the learning procedures for recurrent networks are more complex and more difficult to use. The use of recurrent networks has been proposed for problems that can also be solved by feedforward networks ([Pin87]), but there do not seem to be any good arguments for why a recurrent network should be used over a feedforward network. This thesis explores this question and presents an idea for taking advantage of the power of recurrent networks for learning certain high-dimensional mappings.

The rest of this thesis is organized as follows. Chapter 2 describes feedforward networks and recurrent networks in detail. The main idea of this research of using recurrent networks to learn high-dimensional mappings is then presented briefly. Chapter 3 discusses related work and tries to show where this research fits within the greater body of work. Chapter 4 presents the theoretical aspects of recurrent networks and shows that they are equivalent to Turing machines in computational expressiveness. Chapter 5 goes into detail about using recurrent networks over feedforward networks for learning some high-dimensional mappings. Chapter 6 describes two different learning algorithms for training recurrent networks. Chapter 7 presents experiments on using a recurrent network to learn a simple image segmentation algorithm. Lastly, chapter 8 concludes with some final thoughts on learning with recurrent networks.

Chapter 2

Learning and neural networks

2.1 Sigmoidal networks

There are various different types of neural networks. The differences mainly lie in the type of function used at each unit of the network and where the adjustable parameters are. The most common type of network used by researchers is the sigmoidal network or multilayer perceptron. An example of a sigmoidal network is shown in figure 2.1. It consists of a number of layers each containing many computational units. Each unit contains a sigmoidal function such as the logistic function ($\sigma(x) = \frac{1}{1+e^{-x}}$). The units from a lower layer are connected to units in the next higher layer. The connections among units contain weights that determine how one unit affects another. The output of a unit is computed by applying the sigmoidal function to a weighted sum of its inputs. To formalize this, let x_i denote the output of unit i and let c_{ij} be the weight on the connection from unit i to unit j. If units i and j are not connected then $c_{ij} = 0$. The output of unit j can then be written as

$$x_j = \sigma(\sum_i c_{ij} x_i). \qquad (2.1)$$

The topmost units whose outputs do not connect to any other units are called output units. The bottommost units which do not have any units feeding into them are called input units. All other units are known as hidden units. An input vector is presented to the network by initializing each input unit to some value. Each unit then computes its output after all of the units feeding into it have computed their outputs. The output of the network is the vector of outputs from the output units.

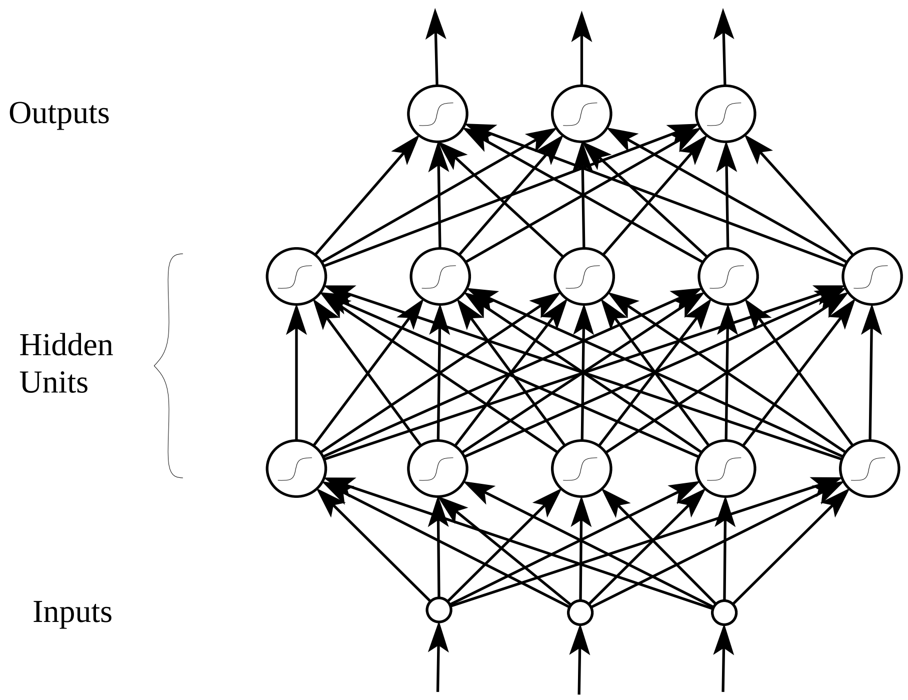

Figure 2.1: Typical sigmoidal network

An important property of sigmoidal networks is that any continuous function can be uniformly approximated arbitrarily well on a finite, compact set by a network with a single hidden layer containing a finite number of units ([Cyb89]). This property guarantees that most functions can be represented accurately by a sigmoidal network, but it does not say anything about how to find a good representation. The problem of finding a good representation for the function being learned is the task of the learning algorithm. The most commonly used learning algorithm is backpropagation ([RHW86]). Backpropagation, as well as many other learning algorithms, attempts to minimize an error function that expresses the distance between the training output examples and the actual outputs of the network. The error function can be thought of as a surface in parameter space. Backpropagation follows the negative gradient of this surface in order to find parameters that minimize the error. The problem with this approach is that the algorithm can get stuck in local minima. Various solutions to the problem of local minima have been used with varying degrees of success ([KGV83]).

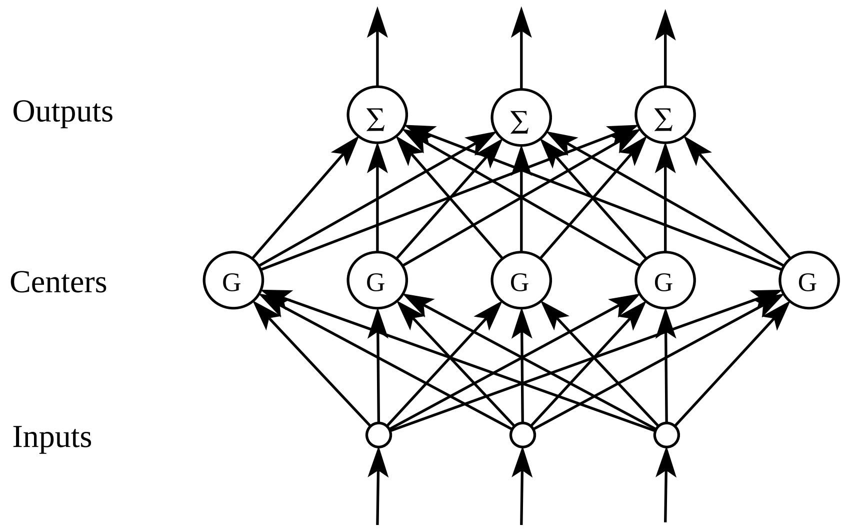

Figure 2.2: Typical radial basis function network with activation function G

2.2 Radial Basis Function networks

Another type of network that has been studied is the radial basis function (RBF) network ([BL88], [PG89], [PG90b]). This three layer network consists of a layer of input units, a layer of radial basis function units and a layer of output units. Each radial basis function unit has a vector of parameters, $\vec{t}_i$, called a center. The connection from RBF unit i to output unit j is weighted by the coefficient c_{ij}. The value of output unit j is given by

$$y_j = \sum_{i=1}^{n} c_{ij} G(\|\vec{x} - \vec{t}_i\|^2), \tag{2.2}$$

where G is a radial basis function and $\|\vec{x}\|$ represents the L_2 norm of $\vec{x}$. Figure 2.2 illustrates a typical RBF network. Examples of radial basis functions include the gaussian, $G(x) = e^{-\sigma x^2}$, and the multiquadric, $G(x) = \sqrt{\gamma^2 + x^2}$.

The number of RBF units is equal to the number of training examples with each center $\vec{t}_i$ being equal to one of the input vectors in the training set. This means that the only parameters in the RBF network that must be learned are the coefficients c_{ij}. Finding the coefficients is a simple linear problem which can be solved by a matrix inversion ([BL88]).

RBF networks can be generalized by allowing fewer centers than training examples. This scheme was named Generalized Radial Basis Functions or GRBFs for short by Poggio and Girosi ([PG89]). The centers can either be fixed or adjustable during learning. If the centers are fixed to some initial values then an overconstrained system of linear equations for the coefficients arises and an exact mapping from input vectors to output vector cannot be found. However, a mapping with the smallest possible L2 error on the training examples can be found by using the pseudo-inverse ([BL88]).

GRBF networks can be further generalized by adding another set of weights to the connections from input units to RBF units. The resulting scheme is called HyperBF networks by Poggio and Girosi ([PG90a]). It corresponds to using a weighted norm $\|\vec{x} - \vec{t}_i\|_W^2 = (\vec{x} - \vec{t}_i)^T W^T W (\vec{x} - \vec{t}_i)$ in place of the L_2 norm. Here, W is the matrix of weights.

Radial basis function networks can be derived from a technique known as regularization theory for changing an ill-posed problem (such as learning a function from input-output examples) into a well-posed problem by imposing smoothness constraints. For details on regularization theory and its relation to RBF networks, see [PG89]. Like sigmoidal networks, RBF networks can also approximate any continuous function arbitrarily well on a finite, compact set. In fact, radial basis functions are equivalent to generalized splines which are a powerful approximation scheme.

2.3 Curse of dimensionality

Sigmoidal networks and radial basis function networks are known to be able to approximate any smooth function well, but this is not the end of the story. The question of how many input-output examples are required to achieve a given degree of accuracy has not been addressed. This problem has been studied and some results are known. In [Sto82], it was shown that in general, the number of examples required to achieve a particular approximation accuracy grows exponentially with the input dimension of the function being approximated, although this effect is mitigated by the smoothness of the function. This presents a major problem in approximating high-dimensional functions. Often the number of examples required can be unreasonably

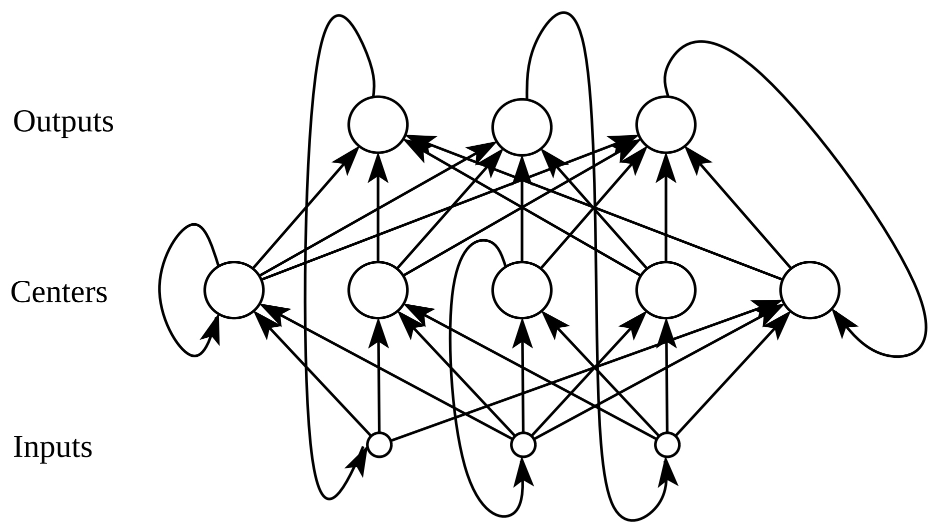

Figure 2.3: Example of a recurrent network

large so that not enough data is available to approximate the function well. Also, the learning procedure often runs in a time dependent on the number of examples which can become impractically long with too many examples. This is a central problem in approximation theory and is known as the curse of dimensionality.

Although the curse of dimensionality is a fundamental problem in approximation theory, there may be methods of avoiding it in practice. It will be shown that recurrent networks may be used to overcome the curse of dimensionality for some problems. Before this idea is presented, recurrent networks in general will be discussed.

2.4 Recurrent networks

The networks discussed so far have been feedforward networks. These networks are called feedforward because each layer is only connected to the following layer and there are no loops in the connections. Computing the output of a feedforward network only requires each unit to calculate its output once. When connections are added from higher layers in a network to lower ones, a recurrent network results. An example of a recurrent network is shown in figure 2.3. In a recurrent network, each unit calculates its output whenever its inputs change. Hence, the feedback connections introduce iteration to the network. A recurrent network that is given an initial input and then allowed to iterate continuously can exhibit a number of behaviors. The simplest case is that it can converge to a fixed point, which means that after enough iterations the output of each unit in the network will stop changing. Another possibility is that the network may get into a limit cycle. This means that the outputs of the units go through a repeating pattern. A recurrent network can also become chaotic, meaning that its outputs never get into a pattern. This thesis is concerned with the case in which the network converges to a fixed point. A number of researchers have presented algorithms for teaching recurrent networks to converge to a desired fixed point given an input pattern. An interesting observation is that this task is the same as that for which a feedforward network is used. Very little has been offered in the way of motivation for using recurrent networks over feedforward networks. The obvious question in this case is whether recurrent networks afford an advantage over feedforward networks for this type of problem. Although there have been some examples of recurrent networks outperforming feedforward ones on the same task ([QS88] and [BGHS91]), very little has been published on this subject. This thesis explores an idea of how the iteration inherent in recurrent networks may be taken advantage of for learning fixed points, thus providing some motivation for using a recurrent network over a feedforward network in some cases. This idea will be presented briefly here and then discussed in greater detail later.

2.5 Avoiding the curse of dimensionality

We would like to take advantage of the power gained from iteration in recurrent networks in order to improve the ability to approximate functions in recurrent networks as compared to feedforward networks. Since the major problem in approximation theory is the curse of dimensionality, then a method that uses the characteristics of a recurrent network to reduce the dimensionality of the function being learned would demonstrate a clear advantage to using recurrent networks over feedforward networks. It is in fact possible to use recurrent networks in this manner for some high-dimensional functions. The idea is that a high dimensional function $\vec{F}$ may be expressed in terms of iterating a lower dimensional function f, and then f can be

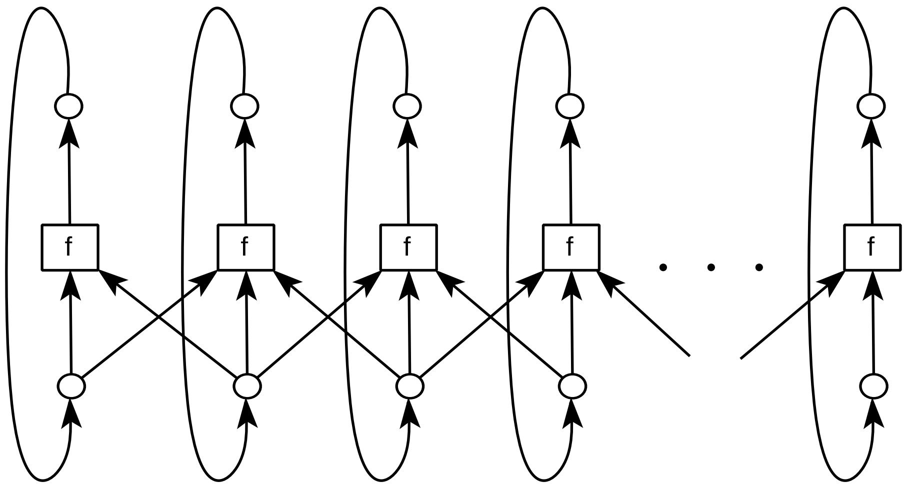

Figure 2.4: Recurrent network for dimensionality reduction. Each box contains a function f which is actually a feedforward network.

learned instead of $\vec{F}$. This approach attempts to avoid the curse of dimensionality by only learning low dimensional mappings. Here we are concerned with vector functions $\vec{F}$ which take a vector as input and return a vector as output. If $\vec{F}$ can be computed by iterating many copies of a simpler function f on pieces of the input vector, then f can be learned instead of $\vec{F}$, in effect reducing the dimensionality of the function that must be learned. This can be understood most easily by looking at figure 2.4. This network consists of many identical copies of the function f, each taking as input a different subset of the whole input. The output from each f forms a vector which is the output of the whole network. This output vector is fed back to the inputs on each iteration until the outputs converge. The function f itself is actually a feedforward network like those discussed earlier. Thus, the free parameters which must be learned in this recurrent network are entirely contained in the representation for f. Because of its lower dimensionality, f should be easier to learn than $\vec{F}$. Once f is learned then F can be calculated by iterating the network containing f. By taking advantage of iteration in a recurrent network, the problem of learning a high dimensional mapping $\vec{F}$ can be reduced to learning a lower dimensional mapping f. This particular recurrent network architecture is intended for learning mappings that act on an array of elements and can be described by local, uniform interactions among the elements. By local, it is meant that the value of an element at time t depends only on neighboring elements at time t-1. By uniform, it is meant that the same local function is used to compute the new values of all elements at each time step. Examples of such mappings include problems in vision such as image segmentation and stereo disparity and many physical phenomena such as modeling gases.

It should be noted that the recurrent network just described is Turing universal given an infinite number of copies of f. It can compute any function computable on a Turing machine, although it is intended for computing functions that are local and uniform. A proof for the Turing universality of this framework is given in chapter 4.

Chapter 3

Related work

It is very informative to briefly review some of the related work by other researchers studying recurrent networks both because it puts the current research in perspective and because it provides some motivation for why this research was carried out in the first place. This research came about largely in response to the question "Why use recurrent networks?" How can the properties of recurrent networks be exploited for approximating functions? It seems that in many cases this question has been left unaddressed.

3.1 Hopfield networks

Hopfield is credited with the idea of associating local minima with memories in a recurrent network. In 1982, Hopfield published a paper which describes how a network of threshold units with feedback connections can be used to implement a content addressable memory ([Hop82]). Each threshold unit outputs either a 1 or a 0 depending on whether the input to the unit is greater than 0 or less than 0, respectively. The set of threshold units in the network are connected by weighted connections with each weight being a real number. Unconnected units have a weight of zero for the connection between them. The input to a unit is computed as in a sigmoidal network, by summing the output multiplied by the connection weight of each of the other units. Hopfield later extended his work to use sigmoidal units in place of threshold units. The resulting network has essentially the same behavior as before ([Hop84]).

A Hopfield network is used to store a number of "memories" which are simply patterns of outputs over the units. The weights on the connections are set according to a simple equation dependent on the particular memories that are to be stored. Setting the weights does not involve an iterative process, but rather is a one-step calculation. Once the weights are set, a memory is recalled by initializing each unit to some value from 0 to 1 and then letting the network iterate until the outputs converge to a fixed point. This fixed point should be one of the stored memories. The idea is that the memory that is output is the one closest to the initial input pattern. The Hopfield net can be viewed as computing a function whose surface contains many local minima. The particular function that the Hopfield network computes depends on the values of the weights on the connections. The weights are picked in such a way that each memory corresponds to a local minima of the network. Hopfield proved that as the network is iterated, the outputs will converge to one of the local minima much like a ball rolling on a hilly surface will eventually come to rest in a valley. Hopfield networks are limited in the number of memories that a network can store at any one time. For a network with N units, approximately $0.15N$ memories can be stored.

The Hopfield network takes advantage of the nature of recurrent networks by having a partial memory pattern converge to the whole pattern as the network iterates. This task would probably be difficult for a feedforward network to emulate. It could be trained to map input patterns to themselves, but this would result in the network learning the identity function. A partial pattern that was presented to the feedforward network would most likely map to itself instead of to the training example that it most closely resembled.

Hopfield's approach is similar to the one that is taken in this thesis in that we are interested in using a recurrent network to converge to a particular output given a particular input. However, the functions considered in this thesis can have an infinite range as opposed to the finite number of memories used in Hopfield networks. Also, the functions considered here do not have to map input patterns to themselves.

3.2 Recurrent backpropagation

A learning algorithm for recurrent networks called recurrent backpropagation has been developed independently by Pineda and Almeida ([Pin87], [Pin89], [Alm87], [Pea88]). As its name suggests, recurrent backpropagation extends the feedforward backpropagation algorithm of [RHW86] to recurrent networks. The algorithm works on networks whose continuous time dynamics are described by the equation

$$dx_i/dt = -x_i + g_i(\sum_j w_{ij}x_j) + I_i \tag{3.1}$$

where $\vec{x}$ is the vector of activation values for each unit of the recurrent network, g_i is a differentiable function which is usually a sigmoid, w_{ij} is a real-valued connection weight and I_i is a constant input ([Pin87]). The network's task is to converge to a particular output vector for each input vector it is given. To train the network to do this, an error measure is defined as

$$E = \frac{1}{2}\sum_{i=1}^{N} J_i^2 \tag{3.2}$$

where $J_i = (T_i - x_i^{\infty})$ and $\vec{T}$ is the target output vector. The recurrent backpropagation procedure attempts to adjust the weights so that the error is minimized. To do this it performs gradient descent in E according to the equation

$$dw_{ij}/dt = -\eta \frac{\partial E}{\partial w_{ij}} \tag{3.3}$$

where η is a constant learning rate. Pineda derives the following simple form for dw_{ij}/dt ([Pin87]):

$$dw_{ij}/dt = \eta y_i^{\infty} x_j^{\infty} \tag{3.4}$$

where x_i^{∞} is a fixed point of equation 3.1 and is obtained by iterating the network until convergence and y_i^{∞} is an error vector which is a fixed point of the dynamical system

$$dy_i/dt = -y_i + g_i'(\sum_j w_{ij}x_j^{\infty})(\sum_k w_{ik}y_k + J_i). \tag{3.5}$$

The above equations are for a single input-output training example. For multiple examples, the total error is defined as

$$E_{TOT} = \sum_{\alpha} E[\alpha] \tag{3.6}$$

which sums over all input-output pairs. The gradient descent equation then becomes

$$dw_{ij}/dt = \eta \sum_{\alpha} y_i^{\infty}[\alpha] x_j^{\infty}[\alpha] \tag{3.7}$$

Using these equations, recurrent backpropagation proceeds as follows. When an input vector is given to the network, x_j^{∞} is approximated by doing a finite number of iterations of equation 3.1. Next, y^{∞} is approximated by iterating equation 3.5 a finite number of times. The weights are then updated according to equation 3.7.

Recurrent backpropagation is distinct from Hopfield networks because arbitrary associations can be made instead of only being able to associate a pattern with itself. With recurrent backpropagation, a network can be taught to converge to a pattern y given an input pattern x that is different from y. After being trained the network could also converge to y given input patterns that were close to but not equal to x. In this sense the recurrent network can act as a content addressable memory. This seems to be the only really useful task that recurrent backpropagation can do that cannot be done easily on a feedforward network. No real motivation has been provided for why one would want to use recurrent backpropagation over feedforward backpropagation on a typical function approximation task.

There are a couple of examples of recurrent backpropagation being used successfully that are worth mentioning. Qian and Sejnowski ([QS88]) used recurrent backpropagation to train a network to learn to compute stereo disparity in random-dot stereograms ([MP76]). Their network was successful, but they used an architecture for the network that was designed especially for the problem. They comment that "using a fully connected network with the hope that the learning algorithm will find a good solution automatically is unlikely to succeed for a large-scale problem. ([QS88])"

Another interesting paper compared the performance of recurrent backpropagation to feedforward backpropagation on a character recognition task ([BGHS91]). The recurrent and feedforward networks used were identical three-layer networks except the recurrent network had feedback connections from the hidden layer to the input layer and from the output layer to the hidden layer. The input to each network was a 16 by 16 array of pixels which contained a representation of a numeral from 0 to 9. The network had 10 outputs. The task of the network was to output a 1 on the output unit corresponding to the numeral represented on the input image. [BGHS91] report that recurrent backpropagation performed slightly better than feedforward backpropagation in terms of the number of epochs needed to reach a certain level of generalization on test images. They do not offer any explanation of how the network took advantage of the iterative nature of the recurrent network.

Despite some success with recurrent backpropagation, it would seem in general that there is no good reason for using recurrent backpropagation over a feedforward learning algorithm for time independent learning tasks except for the case of content addressable memories. Why use the more complicated recurrent backpropagation on a learning task that can be accomplished with a feedforward network? The properties unique to a recurrent network are not being exploited in such a task. However, a technique such as briefly presented in section 2.5 which uses a specific class of recurrent networks in order to exploit iteration for learning high-dimensional functions by reducing the dimensionality of the function being learned may be very useful. This will be the major topic in the remaining chapters of this thesis.

3.3 Learning finite state machines

There is another use of recurrent networks that deserves attention which takes a different approach from those discussed so far. This approach is to use recurrent networks to learn context sensitive mappings that are dependent on past inputs as well as the current input ([SSCM88], [Jor86], [Elm90], [GMC$^+$92]). The standard example of this approach is learning to predict the next letter of a string belonging to a finite state grammar. The basic idea is for the network to receive as input on each time step a representation of the current letter of the string as well as the outputs of some of the other units in the network from the last time step (feedback) and then to use this to predict the next letter of the string. Since there may be multiple letters that follow according to the rules of the grammar, the output of the network is usually an assignment of probabilities to each letter in the grammar indicating the likelihood of that letter to be the next one in the string. In [SSCM88], a recurrent network with three layers that used the network architecture from [Jor86] was employed for learning a finite state grammar. The network's first layer was divided into two parts. The first part was called the context units and were connected in a feedback loop to the hidden units. There was one context unit for each hidden unit. These units could be used by the network to provide an encoding for the current state of a finite state machine that recognized the grammar being learned. The second part of the first layer was called the input units and these units received the current letter of the input string. The current letter was encoded by having one input unit correspond to each letter of the grammar. The second layer in the network contained the hidden units which were connected in a feedforward manner with the input units and in a feedback loop with the context units. Finally, the output layer was connected in a feedforward manner with the hidden layer and contained one unit for each letter of the grammar just like the input units. The training examples were generated by inventing a finite state grammar and selecting a number of strings from this grammar. To train the network, the first letter of the current training string was loaded on the input units and the context units were initialized to zero. The output of the network was then computed and this output was used to generate an error signal by comparing it to the next letter of the string. The error signal was then backpropagated through the network and the weights were adjusted before the next letter was presented. Subsequent time steps proceeded similarly except that the context units were set equal to the activations of the hidden units from the previous time step. [SSCM88] found that such a recurrent network could successfully learn simple finite state grammars.

A similar scheme has been used for learning time series such as stock market prices or sun spot activity ([FS88]). In this case, a series of data points are given to the network as input and trained to predict the next data point. The network can then be iterated in order to produce predictions further into the future.

Using recurrent networks for problems such as learning finite state grammars makes sense because the inputs and outputs of the network vary over time and are dependent on previous inputs and outputs. The need for the network to keep some kind of representation of previous inputs makes the problem context sensitive and hence not suited for a feedforward network which keeps no record of the past. This type of problem makes good use of the properties of the recurrent network.

This framework for using recurrent networks is very different from the approach that is taken in this thesis. In the work by [SSCM88], the recurrent network gets a new input on every time step as well as a target output on every time step that is used to train the network. This thesis is concerned with the problem of learning on a recurrent network which receives a single input and then iterates until it converges to a fixed point which is taken to be the output of the network. No target output is available on intermediate time steps to train the network. This makes the learning task for the network more difficult as compared to learning a finite state grammar.

3.4 Cellular automata

Another area of research that is related to the class of recurrent networks described in section 2.5 is the field of cellular automata ([Wol86], [Gut90]). A cellular automata is a matrix of cells that can assume a finite number of states and a rule for computing the next state of a cell. The next state function is a local function that depends on a cell's current state and the state of certain neighboring cells. The same next state function is used for all cells of the matrix. An example of a cellular automata is the game of Life invented by John Conway ([BCG82]) which simulates the population dynamics of a bacterial colony. Cellular automata are similar to the class of recurrent networks described in section 2.5 in that they both use local, uniform functions to compute the next state for an array of discrete elements. The outputs of each feedforward network contained in the recurrent network can be viewed as states. The two approaches are different in the fact that the cells of a cellular automata can only have a finite number of states while the outputs of the recurrent network can take on a continuous range of values. The recurrent networks of section 2.5 are a generalization of cellular automata since a cellular automata can be implemented by a recurrent network by having the function computed by the feedforward networks approximate the next state function of the cellular automata.

Cellular automata have been used to model a number of physical processes. One example is modeling gas laws. In this application, each cell of the cellular automata represents an area of space which may contain gas particles. The interaction of gas particles is then modeled by the next state function of the cellular automata. The next state function will typically model movement of particles from one area to another and collisions between particles. Such a model may be used to test various theories about gases by comparing the behavior of the model against what is predicted by theory.

Although the mechanisms of cellular automata and recurrent networks are very similar, the intended purpose of each is different. The main emphasis in cellular automata research is on modeling various processes (often physical phenomena) in order to gain a better understanding of the underlying physics involved. The emphasis in recurrent network research is on learning various mappings from examples. The models run on cellular automata are usually designed by people, while the functions that a recurrent network computes are learned by the network. In interesting question is whether a recurrent network can learn the mapping computed by a cellular automata given initial states and final states after some finite number of iterations. This is very similar to the main problem considered in the rest of this thesis. Experience shows that the answer depends on the smoothness of the next state function of the cellular automata as well as the number of examples used to train the recurrent network.

Chapter 4

Computational power of recurrent networks

There are two main questions that arise in the study of recurrent networks. One is the question of their computational power. What set of functions can be represented by a recurrent network? The other is the question of how to go about learning functions on a recurrent network. The question of the computational power of recurrent networks is addressed in this chapter, while the remaining chapters address the problem of learning.

It turns out that recurrent networks are very powerful indeed. A recurrent network can simulate a Turing machine and can thus compute any function computable on a Turing machine. This has been proven for recurrent networks with sigmoidal units by [SCLG91]. Here, the focus is on the class of recurrent networks informally described in section 2.5. To prove this result for this class of recurrent networks, the formal definitions of a recurrent network and a Turing machine will be given and then a constructive proof for simulating a Turing machine on a recurrent network will be presented.

4.1 Formal definition of a recurrent network

Let I be a set of input variables. A *recurrent network* defined on I is a triple $\mathcal{N} = (D, G, f)$ where

- D is the domain of each input variable.

- G = (I, E) is a graph defining the connectivity of the network. E is a set of lists $\{E_{v_i} : v_i \in I\}$ where each list $E_{v_i} = (v_{j1}, v_{j2}, \ldots, v_{jn})$ specifies the variables that v_i depends on. Each list is ordered so that it may be used as a list of arguments to a transition function. The size of each list, n, must be the same.

- $f : D^n \rightarrow D$ is a transition function that maps values of variables at time t to a value at time t+1.

The recurrent network operates by simultaneously updating each variable on each time step according to the transition function. In particular,

$$v_i^{t+1} \leftarrow f(E_{v_i}) \qquad \forall v_i \in I$$

where v_i^t represents the value of v_i at time t. One should keep in mind figure 2.4 to make this definition clear.

In this definition, f is simply an arbitrary function. It is not necessarily a radial basis function. We are not concerned here with learning the function f.

Note: a *cellular automata* is a recurrent network with the restrictions that D is a finite set and the graph G is translation invariant (i.e. $E_{v_i} = (v_{j_1}, v_{j_2}, \ldots, v_{j_n})$ iff $E_{v_{i+k}} = (v_{j_1+k}, v_{j_2+k}, \ldots, v_{j_n+k})$).

4.2 Formal definition of a Turing machine

A Turing machine consists of a one-way infinite tape and a finite control. On each time step, the finite control reads the symbol from the tape under the current position of the read head and then based on its current state and the symbol that is read, writes a new symbol on the tape, enters a new state and moves the read head either left or right on the tape. The components of a Turing machine can be formally described by the following definition which is taken from Hopcroft and Ullman's book, *Introduction to Automata Theory, Languages and Computation* ([HU79]).

A Turing machine is a 7-tuple $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ where

- Q is the finite set of states,

- Γ is the finite set of allowable tape symbols,
- B, a symbol of Γ, is the blank,
- Σ, a subset of Γ not including B, is the set of input symbols,
- δ is the transition function, a mapping from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$,
- q_0 in Q is the start state,
- $F \subset Q$ is the set of final states.

4.3 Simulation of a Turing machine by a recurrent network

Using the above definitions, we can now construct a recurrent network that simulates a particular Turing machine given the 7-tuple describing the Turing machine. This construction is similar to the one in [GM90] showing how a cellular automata can simulate a Turing machine.

Given $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, define $\mathcal{N} = (D, G, f)$ as follows:

- $\mathrm{I} = (v_1, v_2, v_3, \ldots)$
- $D = ((Q \cup *) \times \Gamma)$ where $* \notin Q$.
- $G = (I, E)$ where $E = \{E_{v_1}, E_{v_2}, E_{v_3}, \ldots\}$ and
  $E_{v_i} = (v_{i-1}, v_i, v_{i+1})$ for $2 \leq i \leq l$
  $E_{v_1} = (v_1, v_1, v_2)$
- The initial values of the variables are defined from the Turing machine's initial tape as follows:
  $v_1 = (q_0, t_1)$ and
  $v_i = (*, t_i)$ for $i > 1$
  where q_0 is the start state of $\mathcal{M}$ and t_i is the i^{th} symbol on the tape of the Turing machine.

- The network's transition function is defined as:

$$f((*,x),(q,y),(*,z)) = (*,w) \text{ where } \delta(q,y) = (q',w,d)$$

$$f((q,x),(*,y),(*,z)) = \begin{cases} (q',y) & \text{if } \delta(q,x) = (q',w,R) \\ (*,y) & \text{if } \delta(q,x) = (q',w,L) \end{cases}$$

$$f((*,x),(*,y),(q,z)) = \begin{cases} (*,y) & \text{if } \delta(q,z) = (q',w,R) \\ (q',y) & \text{if } \delta(q,z) = (q',w,L) \end{cases}$$

$$f((*,x),(*,y),(*,z)) = (*,y)$$

$$f((q,x),(q,x),(*,y)) = (*,w) \text{ where } \delta(q,x) = (q',w,d)$$

where $x, y, z, w \in \Gamma$, $q, q' \in Q$, and $d \in \{L, R\}$.

The variables of the recurrent network $\mathcal{N}$ store the contents of the Turing machine's tape as well as keeping track of the state of the finite control and the position of the read head. This is accomplished by having each variable of the recurrent network store an ordered pair. The first member of the ordered pair contains either a state or the symbol $*$. The second member of the ordered pair contains a tape symbol. Variable v_i holds the i^{th} tape symbol. The variable in the recurrent network corresponding to the read head's current position contains the current state while all other variables contain a $*$ in the first position of their ordered pair. The task for the function f in the recurrent network is to keep track of the current state and the read head's position as well as the contents of the tape by mimicking the Turing machine's transition function and updating the values of the recurrent network's variables accordingly. To do this, variable v_i gets the value of f evaluated on v_{i-1}, v_i and v_{i+1}. The definition of f is split into five cases. The first case is for the read head being on v_i which is signified by the first component of the ordered pair containing a state symbol as opposed to a $*$. In this case, v_i gets the value $(*,w)$ where $*$ signifies that the read head is no longer at v_i and w is the symbol written onto the tape. The second case is for when the read head is to the left of v_i at v_{i-1}. In this case, the first component of v_i's ordered pair becomes the new state q' if the read head is instructed to move right, otherwise v_i's value is unchanged. The third case is the same as the second except the read head is to the right of v_i at v_{i+1}. The fourth case covers the situation in which the read head is not on any of v_{i-1}, v_i or v_{i+1}. In this case, v_i is unchanged. The final case is a special case for v_1. Since v_1 has no left neighbor, the connectivity of the recurrent network defined by G causes f to be evaluated on v_1, v_1 and v_2. This case simply covers the possibility that the read head is on v_1.

This construction is able to use a fairly straightforward simulation of a Turing machine on a recurrent network because there is no restriction on the function f in the recurrent network. This allows f to implement δ almost directly. By showing that a recurrent network can simulate a Turing machine, we have proven that a recurrent network is as powerful as a Turing machine in terms of the set of functions it can compute. The next chapters will investigate how the power of a recurrent network can be exploited for learning functions from examples.

Chapter 5

Using recurrent networks for dimensionality reduction

5.1 The curse of dimensionality

As mentioned earlier, the difficulty of learning high dimensional mappings is the major problem in approximation theory. Stone has studied this problem by finding the optimal rate of convergence for approximating functions which is a measure of how accurately a function can be approximated given n samples of its graph ([Sto82]). He showed that using local polynomial regression the optimal rate of convergence of the error is $\epsilon_n = n^{-\frac{p}{2p+d}}$ where n is the number of examples, d is the dimension of the function and p is the degree of smoothness of the function or the number of times it can be differentiated. This means that the number of examples needed to approximate a function grows exponentially with the dimension. To put this in perspective using an example from Poggio and Girosi ([PG89]), for a function of two variables that is twice differentiable, 8000 examples are required to obtain $\epsilon_n = 0.05$. If the function depends on 10 variables, however, then 10^9 examples are needed to obtain the same rate of convergence. Because of the curse of dimensionality, any method for reducing the input dimension of a function is an important tool for learning. The main point of this research is to study how to take advantage of recurrent networks for function approximation by reducing the dimensionality of the function being learned. As previously discussed, the idea that we will explore is that a high-dimensional mapping $\vec{F}$ may be expressed in terms of iterating a lower dimensional function f, and then f

can be learned instead of $\vec{F}$.

5.2 Image segmentation example

To make this idea more concrete, consider the following example. The problem of image segmentation is to take an image consisting of a matrix of pixels and output an image in which each object in the image is separated from surrounding objects. One way to do this is to simply average the values of all pixels within an object and assign each pixel in the object the average value. Different objects can be distinguished from each other by using the assumption that there is a sharp change in the values of pixels (colors) across object boundaries. The problem of training a feedforward network to learn a segmentation mapping that takes as input an image with thousands of pixels and outputs the segmented image is impractical because of the extremely high input dimension. However, a simple image segmentation mapping can be implemented as an iterative procedure which breaks the function into many low dimensional functions. This algorithm taken from [Hur89] works as follows. On each time step, the value of each pixel is replaced by the average of all neighboring pixels whose values are within some threshold of the original pixel's value. This process eventually converges to an image in which each object is colored with its average gray-scale value. Overlapping objects must be of different shades of gray in order to be separated. This iterative procedure only requires a low dimensional function that takes as input a small neighborhood of pixel values and outputs the average of all values that are within some threshold of the central pixel's value. In this case the high-dimensional function for mapping an unsegmented image to a segmented image can be broken up into many low-dimensional functions that work in parallel and are iterated so that they converge to the original high-dimensional function. In a sense, the time to compute the desired segmentation function has been traded off against the dimension of the function.

5.3 A recurrent network architecture for function decomposition

The iterative mapping for image segmentation just described can be implemented on a recurrent network such as in figure 5.1 which uses many copies of a low dimensional function and may take many iterations to converge. This contrasts with the one-shot algorithm that can be implemented on a feedforward network such as in figure 2.2. It is a high-dimensional function, but can be computed in one step.

To formalize this idea, consider a vector function $\vec{F}$ which takes a vector $\vec{x}$ as input and returns a vector $\vec{y}$ as output. Suppose $\vec{F}$ can be expressed as the result of iterating many copies of a simpler function f on pieces of the input vector. In other words, let

$$\vec{u}^{t+1} = < f(P_1\vec{u}^t),\ f(P_2\vec{u}^t),\ \ldots,\ f(P_n\vec{u}^t) > \ \ for\ t \geq 0 \tag{5.1}$$

and

$$\vec{u}^0 = \vec{x} \tag{5.2}$$

where P_j is an $m \times n$ matrix that maps from $\mathcal{R}^n$ to $\mathcal{R}^m$ and $m < n$. (In other words, P_j is a linear operator which extracts a piece of $\vec{u}^t$.) $\vec{F}$ can be expressed as

$$\vec{F}(\vec{x}) = \lim_{t \to \infty} \vec{u}^t. \tag{5.3}$$

The above notation simply means that the output vector $\vec{y} = \vec{u}^\infty$ is computed by first computing a number of intermediate $\vec{u}^t$'s. Element j of the vector $\vec{u}^{t+1}$ is computed by evaluating f on input $P_j\vec{u}^t$. Since P_j is an $m \times n$ matrix, $P_j\vec{u}^t$ is simply a vector that is a piece of the vector $\vec{u}^t$.

Because of its lower dimensionality, many fewer examples are needed to learn f than to learn $\vec{F}$. As discussed earlier, this can be very important for making the learning task tractable. Once f is learned, $\vec{F}$ can be calculated by iterating the network containing f. Thus, by taking advantage of iteration in a recurrent network, the problem of learning a high-dimensional mapping $\vec{F}$ can be reduced to learning a lower dimensional mapping f.

The class of functions defined by equations 5.1-5.3 can be represented directly on a recurrent network such as that shown in figure 5.1. This network consists of many

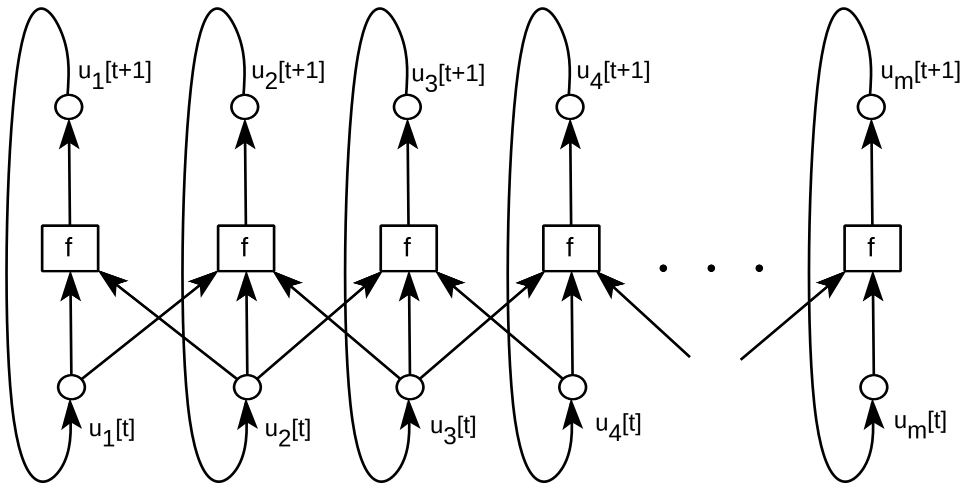

Figure 5.1: Recurrent network for dimensionality reduction. Each box contains a function f which is actually a feedforward network.

identical copies of a feedforward network that approximates f. The P_j matrices are represented by the connections from input units to the inputs of each copy of f. The outputs from each copy of f form the output of the entire recurrent network at each time step. This output is fed back to the inputs at the end of each iteration. The operation of the network proceeds as follows. The input units are first initialized and then on each iteration the outputs are computed by evaluating each copy of f simultaneously on its local input. The outputs are fed back to the inputs and this process repeats until the outputs converge to a fixed point. In practice, the network is iterated for a fixed number of steps so that problems with divergence do not occur.

It should be noted that the connectivity of the network shown in figure 5.1 is intended as an example and alternative topologies are possible. Also, there is nothing preventing f from having multiple outputs instead of the single output pictured. The particular function being learned will dictate such details. In this research all of the networks studied use a function with a single output for f. In the remainder of this thesis, the term recurrent network will refer to a network with the particular architecture typified by figure 5.1.

It may be unclear how copies of a low-dimensional function are being used to

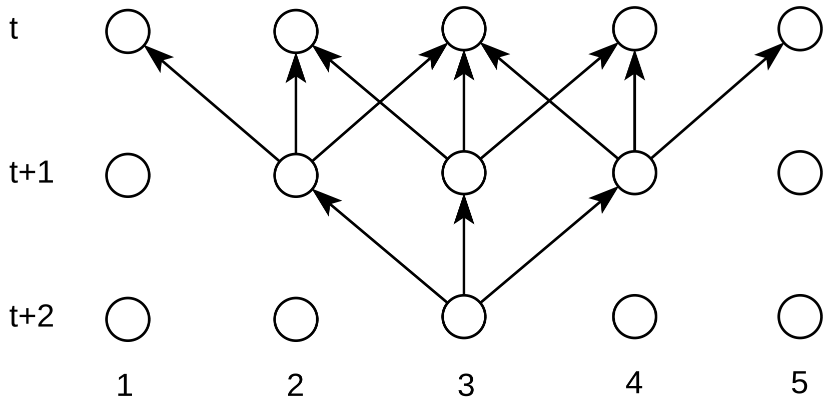

Figure 5.2: See text for explanation.

compute a high-dimensional function that is dependent on all of its inputs. The trick is in the iteration. Consider figure 5.2 which represents the five output units of a recurrent network at time steps t, t+1 and t+2. Assume the network topology is such that the value of an output at time t+1 depends on the values of itself and its two closest neighbors at time t. This means that at time t+2, the value of output 3 depends on the values of outputs 2, 3 and 4 at time t+1. Because of the dependencies of outputs 2, 3 and 4 at time t+1 on the outputs 1, 2, 3, 4 and 5 at time t, output 3 at time t+2 actually depends on outputs 1, 2, 3, 4 and 5 at time t. Hence, the dependencies of each output element in the network spreads as the iteration progresses. This spreading allows the iteration of local functions to compute a global function.

Using such a network, the learning problem is to find the function f such that the recurrent network converges to $\vec{F}$ as it iterates. The training examples given to the learning algorithm are input vectors $\vec{x}_i$ and output vectors $\vec{y}_i$ such that $\vec{y}_i = \vec{F}(\vec{x}_i)$. Examples for f are not given - only examples for $\vec{F}$. Thus, the learning task is to somehow use examples for $\vec{F}$ to learn the lower dimensional function f. The next chapter presents two learning algorithms that solve this problem.

Chapter 6

Learning algorithms for recurrent networks

6.1 Recurrent gradient descent

Many of the algorithms used to train neural networks are based on a gradient descent search. The recurrent networks described in the previous section can also be trained by a straightforward gradient descent procedure similar to the algorithm described in [WZ89]. Let the network be as in figure 5.1 and consist of m inputs and m outputs. Let each feedforward network, f, be an GRBF net with l inputs and n centers along with linear and constant units. (A sigmoidal network could also be used for f, but an GRBF net was chosen since it is used later for running experiments.) The inputs to the recurrent net at time t are denoted by

$$\vec{u}[t] = \{u_1[t], u_2[t], ..., u_m[t]\}. \tag{6.1}$$

The input vector $\vec{u}[t]$ refers to the inputs that are fed back from the outputs at time $t-1$. These are not external inputs which are injected into the network at each time step. The only external input comes from initializing $\vec{u}[0]$ with a given input vector. After that, the network runs until its outputs converge. It does not receive further inputs from the outside. Since the vector of outputs from the feedforward networks at time t is fed back to the inputs at time $t+1$, the ith element of the input vector can be expressed as

$$u_i[t+1] = f(P_i\vec{u}[t]), \; t \geq 0 \tag{6.2}$$

where P_i is an $l \times m$ matrix which picks out a vector of length l from input vector $\vec{u}$ which is of length m. P_i is simply a notational convenience that encodes the connections from the inputs of the whole recurrent network to the inputs of the ith copy of f. Since f has been chosen to be an GRBF network with linear and constant terms, it can be expressed as

$$f(\vec{v}) = \sum_{i=1}^{n} c_i G(\|\vec{v} - \vec{t}_i\|^2) + \sum_{i=1}^{l} c_{n+i} v_i + c_{n+l+1} \tag{6.3}$$

where G is the basis function being used, $\vec{t}_i$ is the ith center, and c_i is the ith coefficient. In this work, the centers are chosen a priori and remain fixed during learning so that the only free parameters to be learned are the coefficients.

Equations 6.1 - 6.3 define the dynamics of the recurrent network. Now an error measure is needed such that the optimum coefficients yield the minimum error. Let $\{\vec{x}^j\}_{j=1}^N$ and $\{\vec{y}^j\}_{j=1}^N$ denote the training set of N input and output examples, respectively. The task that the recurrent network must learn is to converge to output $\vec{y}^j$ given input $\vec{x}^j$. Let E be an error function defined as

$$E = \frac{1}{2} \sum_{j=1}^{N} \sum_{k=1}^{m} (y_k^j - u_k^j[\infty])^2. \tag{6.4}$$

The vector $\vec{u}^j[\infty]$ is computed by iterating the network given the initial condition $\vec{u}^j[0] = \vec{x}^j$. To find a local minimum of E, the negative gradient with respect to the coefficients can be followed. Computing the derivative of E with respect to c_i yields

$$\frac{\partial E}{\partial c_i} = -\sum_{j=1}^{N} \sum_{k=1}^{m} (y_k^j - u_k^j[\infty]) \frac{\partial u_k^j[\infty]}{\partial c_i}. \tag{6.5}$$

This partial derivative can then be used to change the coefficients so that the error, E, decreases. The weight change for an individual coefficient c_i is thus

$$c_i \leftarrow c_i - \alpha \frac{\partial E}{\partial c_i} \tag{6.6}$$

where α is the learning rate which should be chosen to be a small positive real number.

Now it just remains to write an expression for $\frac{\partial u_k^j[t]}{\partial c_i}$. Although a bit messy, it is computed by differentiating equations 6.2 and 6.3.

$$\frac{\partial u_k^j[t+1]}{\partial c_i} = \begin{cases} \sum_{j=1}^{n}[c_j G'(\|\vec{v}-\vec{t}_j\|^2)(2\sum_{p=1}^{l}(v_p - t_{jp})\frac{\partial v_p}{\partial c_i})] \\ \quad +G(\|\vec{v}-\vec{t}_i\|^2) + \sum_{p=1}^{l} c_{n+p}\frac{\partial v_p}{\partial c_i} & \text{if } i \le n \\ \\ \sum_{j=1}^{n}[c_j G'(\|\vec{v}-\vec{t}_j\|^2)(2\sum_{p=1}^{l}(v_p - t_{jp})\frac{\partial v_p}{\partial c_i})] \\ \quad +v_{i-n} + \sum_{p=1}^{l} c_{n+p}\frac{\partial v_p}{\partial c_i} & \text{if } n+1 \le i \le n+l \\ \\ \sum_{j=1}^{n}[c_j G'(\|\vec{v}-\vec{t}_j\|^2)(2\sum_{p=1}^{l}(v_p - t_{jp})\frac{\partial v_p}{\partial c_i})] \\ \quad +\sum_{p=1}^{l} c_{n+p}\frac{\partial v_p}{\partial c_i} + 1 & \text{if } i = n+l+1 \end{cases} \tag{6.7}$$

where $\vec{v} = P_k \vec{u}^j[t]$ and $G'(x)$ denotes the derivative of G with respect to x. The three cases in equation 6.7 correspond to c_i being a coefficient of a basis function term, a linear term or the constant term. The base case for this recurrence equation is

$$\frac{\partial u_k^j[0]}{\partial c_i} = 0. \tag{6.8}$$

Pseudo code for the recurrent gradient descent algorithm can now be written as follows.

```
until the error is low enough do
    for j=1 to the number of training examples N do
        $\vec{u}^j[0] = \vec{x}^j$
        $\frac{\partial \vec{u}^j[0]}{\partial c_i} = \vec{0}\ \forall i$
        for t=1 to the time that the output converges do
            for k=1 to number of outputs m do
                for i=1 to number of coefficients n+l+1 do
                    Compute $\frac{\partial u_k^j[t+1]}{\partial c_i}$ using equation 6.7 given
                      the partial derivatives of $\vec{u}^j$ at time t.
                    Compute $u_k^j[t+1]$ by evaluating the GRBF
                      network for $f$ given $\vec{u}^j[t]$.
        for i=1 to number of coefficients n+l+1 do
            Compute $\frac{\partial E}{\partial c_i}$ using equation 6.5
            Update coefficient $c_i$ using equation 6.6.
```

The pseudo code says that the loop over t continues until the output of the recurrent net has converged. In practice, however, this loop is terminated after some fixed number of steps by which time the network is assumed to have converged.

The main problem with the gradient descent approach is that it requires calculating a complicated derivative each time the coefficients are updated. This calculation is computationally expensive and so the algorithm can be rather slow. Also, as with all gradient descent algorithms, it is vulnerable to getting stuck in a local minima.

6.2 Random step algorithm

In order to avoid the time consuming calculation of the gradient, a simple nondeterministic algorithm has been developed by Caprile and Girosi ([CG90]. Their algorithm can be used to find the optimum parameters for a neural network as follows. To each parameter of the network add a random amount of noise chosen from the interval $(-\omega, \omega)$. If the error of the network on its training set has decreased, then retain the changes to the parameters. Otherwise return the parameters to their previous values. If the change decreases the error then double ω, otherwise halve ω. This changes the magnitude of the noise that will be added to the parameters on the next iteration. Continue this process of randomly perturbing the parameters until the error is suitably low. The idea is to expand the search when the changes cause the error to decrease and to narrow the search when the changes cause it to increase. Note that the error for the network can only decrease with this scheme since any changes that cause it to increase are discarded. If ω gets very small so that the changes to the parameters are too small to affect the error, ω can be reset to some larger number.

The random step algorithm can be used with the recurrent architecture described earlier by using the output of the recurrent network after some fixed number of iterations (instead of infinite iterations) in equation 6.4. Since the feedforward networks, f, that compose $\vec{F}$ are identical, only one set of parameter changes are made, and these changes happen to each copy of f so that they remain identical.

Caprile and Girosi report that the random step algorithm performs comparably to gradient descent on some typical function approximation problems. An iteration of the random step algorithm is much faster than an iteration of the gradient descent algorithm, but many iterations of the random step algorithm result in no improvement to the error since some of the random changes to the parameters are not helpful. So the speed of each iteration is offset somewhat by the need to do many more iterations as compared to gradient descent. The random step algorithm does have the advantage of being able to avoid local minimum which is a problem with straight gradient descent. It is also much easier to implement. Another advantage of the random step algorithm is that there is no learning rate that must be fine tuned in order for the algorithm to work well as with gradient descent. In practice on the experiments run for the work in this thesis, the random step algorithm has worked better than the gradient descent algorithm mainly because of local minima.

The random step algorithm is very simple to parallelize on a SIMD parallel machine such as the Thinking Machines CM 2. One approach to parallelizing this algorithm is as follows. Each processor keeps its own copy of the parameters for the network. On each iteration, each processor picks random changes to its set of parameters in parallel. Each processor then calculates the error of its network with the new parameters. The processor that has the network with the lowest error is found. If this error is less than the previous lowest error then each processor is given the parameters for the best network. Otherwise, all processors keep the old, unchanged parameters. The magnitude of the noise is updated accordingly and the process repeats. This approach in effect tries many random changes to the parameters at once and keeps the best changes. The parallel random step algorithm has been implemented and used for this research and as expected is significantly faster than the sequential version.

6.3 Setting the initial parameters

The problem of how to initially set the parameters of the recurrent network (the centers and coefficients in the case of GRBF networks within the recurrent net) is an important question. Learning procedures for recurrent networks, such as those just presented, are relatively slow and a clever choice of where to start the parameters can result in a large time savings. In this work, the centers of the GRBF are initialized and then kept fixed while the coefficients are learned. To set the centers, it usually suffices to pick them to be a subset of the input examples or to form a regular grid. Picking the coefficients is more difficult. One idea is to first train f as a feedforward net by treating the output vectors for $\vec{F}$ as outputs after one iteration instead of an infinite number of iterations. To do this, an input-output pair $(\vec{x}, \vec{y})$ such that $\vec{F}(\vec{x}) = \vec{y}$ is split up according to the connectivity of the recurrent network into many examples for the feedforward net f. To make this explicit, suppose that $\vec{F}$ has five inputs and five outputs and f has three inputs and one output. Also suppose that the connectivity of the recurrent network is as in figure 5.1. Let $(x_1, x_2, x_3, x_4, x_5)$ be an input example and $(y_1, y_2, y_3, y_4, y_5)$ be the corresponding output example. The following examples could then be created to train f as a feedforward network:

$$(x_1, x_2, x_3) \rightarrow y_2,$$

$$(x_2, x_3, x_4) \rightarrow y_3,$$

$$(x_3, x_4, x_5) \rightarrow y_4.$$

Also, since the output examples for $\vec{F}$ are fixed points, these provide further training examples for the feedforward network f. In other words, since $\vec{F}(\vec{y}) = \vec{y}$, the pair $(\vec{y}, \vec{y})$ can be used as a training example and split up into examples for training f. Training f simply involves computing a pseudo-inverse to solve for the optimal coefficients as described in chapter 2 since the centers are assumed to be fixed.

The idea behind initializing the coefficients of f with this technique is that we are trying to have the network approximate $\vec{F}$ in one shot without iteration. Although it should not be able to do this extremely accurately, it should provide a good starting point in many cases.

Chapter 7

Example: Image segmentation

The previous sections have described a motivation for using recurrent networks over feedforward networks to learn fixed point mappings. It remains to be shown that this motivation is justified in practice. In order to test the performance of a recurrent network for learning high dimensional mappings, the problem of learning a simple image segmentation mapping was chosen. The particular segmentation algorithm used is taken from [Hur89]. In this chapter, the term recurrent network refers specifically to the class of recurrent networks composed of many copies of a feedforward network whose outputs feed back to the inputs as illustrated in figure 5.1.

7.1 Hurlbert's image segmentation algorithm

As briefly mentioned earlier, the basic idea of the image segmentation algorithm is to take an image which is represented as a table of gray scale pixels and update the value of each pixel with the average of the values of all the immediately surrounding pixels whose gray scale values are close enough to the original pixel's value. This process is repeated until it converges to the segmented image. Simply stated, the algorithm smooths out the values of all pixels within the same object in the image while not smoothing over object boundaries. The algorithm can be written algebraically as follows:

$$u_{x,y}^{t+1} = \frac{1}{n(N^t)} \sum_{l,m \in N(u_{x,y}^t)} u_{l,m}^t \tag{7.1}$$

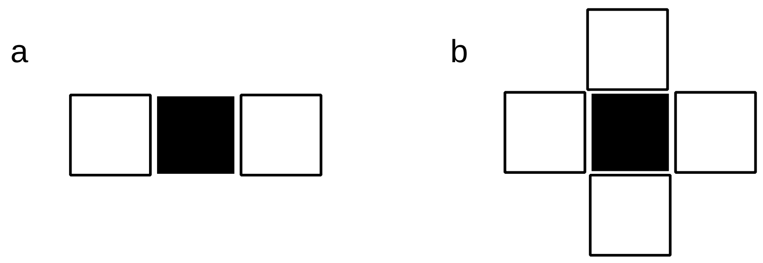

Figure 7.1: a) The white pixels to the left and right of the black pixel comprise the neighborhood of the black pixel for one-dimensional segmentation. b) For two-dimensional segmentation, the four white pixels form the neighborhood of the black pixel.

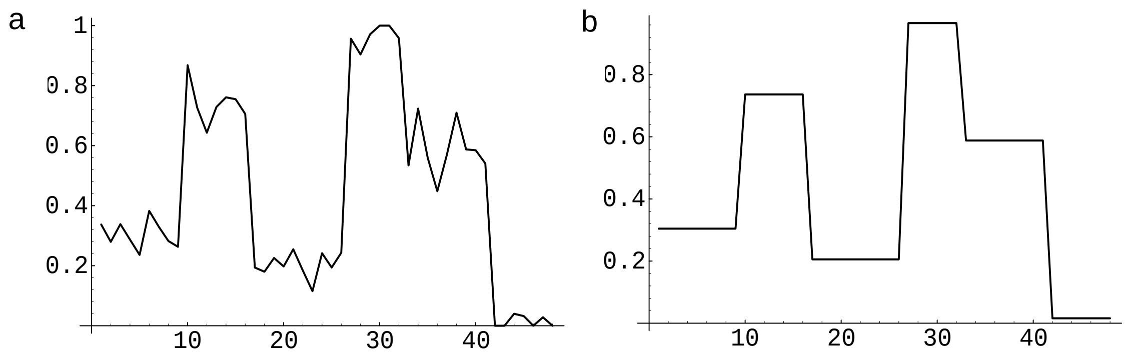

Figure 7.2: Typical training pair for the one-dimensional segmentation problem. Figure (a) is the input vector and (b) is the output obtained by running Hurlbert's segmentation algorithm on the input.

where $u_{x,y}^t$ is the value of pixel x, y at time t, and $N(u_{x,y}^t)$ is the set of $n(N^t)$ pixels among the neighbors of x, y that differ from $u_{x,y}^t$ by less than some threshold. The neighborhood of a pixel in a two dimensional image is simply the pixels above, below, to the left, and to the right. This algorithm can also be applied to a one-dimensional "image" in which the pixels to the left and to the right of a pixel comprise its neighborhood. The neighborhoods for a one and two-dimensional mapping are illustrated in figure 7.1.

7.2 Learning one-dimensional segmentation

The ability of the recurrent network to learn the segmentation mapping, was first tested by training it on the one-dimensional problem. The task that the recurrent network had to learn was to output a segmented one-dimensional image after some number of iterations given an unsegmented image as input. The recurrent network was given a number of unsegmented input images and the corresponding segmented output images as training examples. The training examples were chosen to be 16 pixels long and each pixel had a real value that was scaled to be between 0.0 and 1.0. The input vectors were randomly generated by a simple program and the output vectors were computed by running the actual image segmentation algorithm on the input vectors. This process created as many training examples as needed. Figure 7.2 shows a typical input vector used and the corresponding output vector.

Hurlbert's image segmentation algorithm maps directly into the recurrent network architecture shown in figure 7.3. Each box contains a GRBF network that must learn to approximate the function described by equation 7.1. Since the neighborhood for one-dimensional segmentation contains three pixels, each GRBF net has three inputs. The three input segmentation function can be written explicitly as

$$u_x^{t+1} = f(u_{x-1}^t, u_x^t, u_{x-1}^t) = \begin{cases} (u_{x-1}^t + u_x^t + u_{x+1}^t)/3 & \text{if } |u_x^t - u_{x-1}^t| \leq T \text{ and} \\ & \quad |u_x^t - u_{x+1}^t| \leq T \\ (u_{x-1}^t + u_x^t)/2 & \text{if } |u_x^t - u_{x-1}^t| \leq T \text{ and} \\ & \quad |u_x^t - u_{x+1}^t| > T \\ (u_x^t + u_{x+1}^t)/2 & \text{if } |u_x^t - u_{x-1}^t| > T \text{ and} \\ & \quad |u_x^t - u_{x+1}^t| \leq T \\ u_x^t & \text{if } |u_x^t - u_{x-1}^t| > T \text{ and} \\ & \quad |u_x^t - u_{x+1}^t| > T \end{cases} \tag{7.2}$$

where T is a threshold value which was chosen to be 0.2 in the experiments described later and u_x^t is the value of pixel x at time t. Equation 7.2 is a way of rewriting equation 7.1 given the particular neighborhood for each pixel that has been chosen.

A slice of the graph of this function is shown in figure 7.4. The graph shows that the surface is composed of many different discontinuous planes. These discontinuities make the function much more difficult to learn in terms of the number of examples and GRBF centers required. After many attempts of trying to approxi-

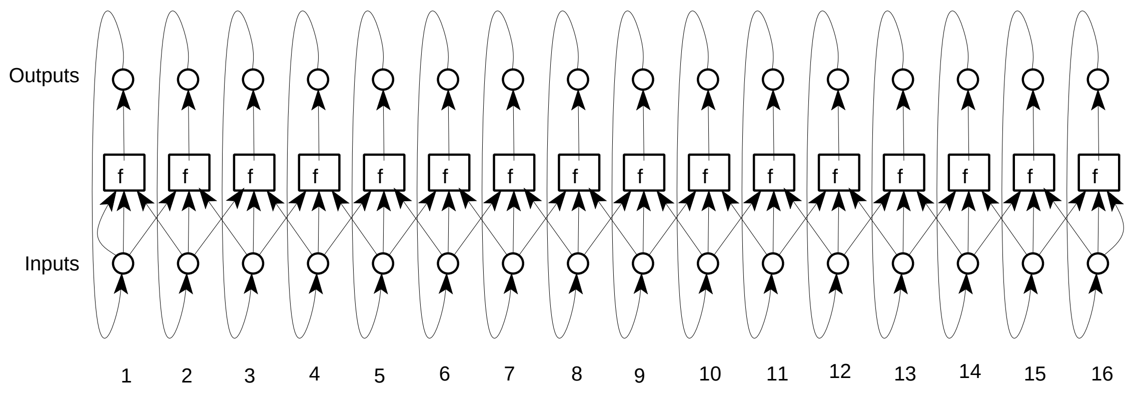

Figure 7.3: This recurrent network was used to learn the one-dimensional segmentation mapping in which the input and output images were 16 pixels long. Each box represents a copy of the same GRBF network which takes a neighborhood of pixels as input and outputs the value of the middle pixel on the next time step.

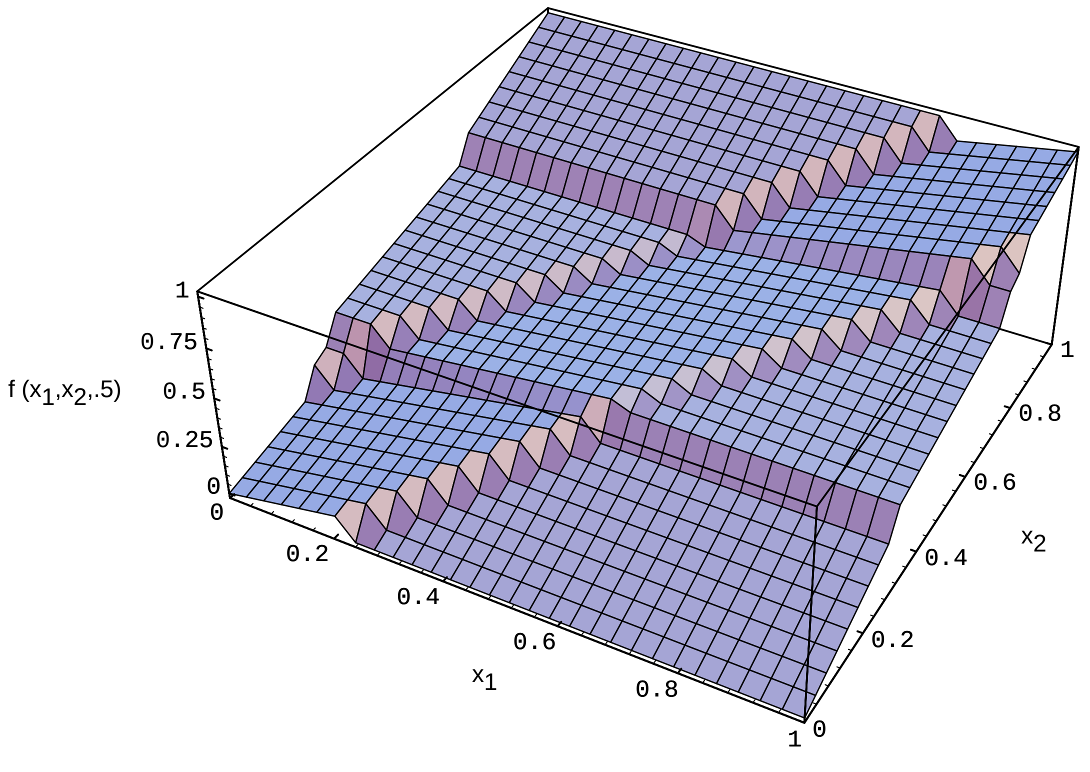

Figure 7.4: A slice of the graph of $f(x_1, x_2, x_3)$ as defined in equation 7.2 with $x_3 = 0.5$.

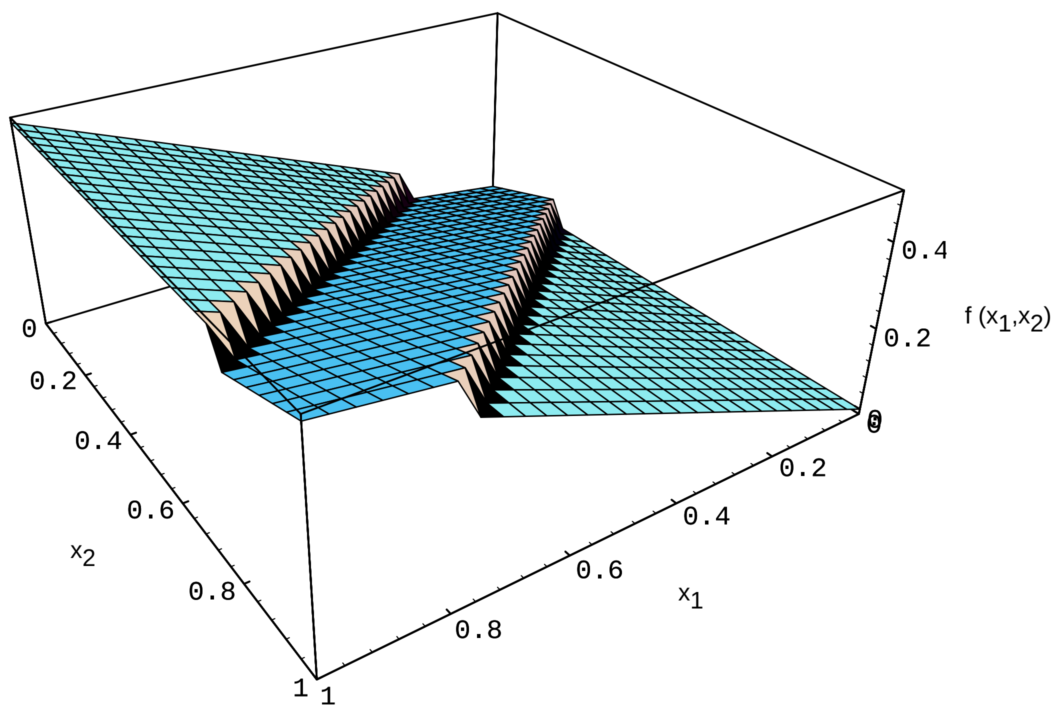

Figure 7.5: Graph of $f'(x_1, x_2)$ defined by equation 7.4.

mate this function using an GRBF network with gaussian units, it was found that the network generalized poorly even with 625 centers. It appears that an extreme number of centers and examples are needed to learn this function because of the lack of smoothness. In order to overcome this problem, the three-input function was split into two two-input functions by taking advantage of the symmetry inherent in the mapping. The function f of equation 7.2 can be approximated by

$$f(u^t_{x-1}, u^t_x, u^t_{x+1}) \approx f'(u^t_x, u^t_{x-1}) + f'(u^t_x, u^t_{x+1}) \tag{7.3}$$

where

$$f'(u^t_x, u^t_y) = \begin{cases} (u^t_x + 2u^t_y)/6 & \text{if } |u^t_x - u^t_y| \leq T \\ u^t_x/2 & \text{if } |u^t_x - u^t_y| > T. \end{cases} \tag{7.4}$$

A graph of this function is shown in figure 7.5. The surface of the graph is composed of three discontinuous planes, and it proved to be much easier for a GRBF network to learn this than the three-input function. A GRBF network with 144

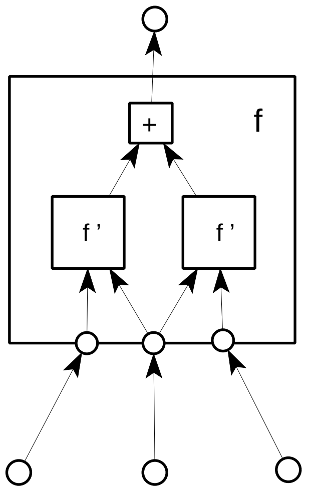

Figure 7.6: This figure shows how f was split into two identical GRBF networks, f', by taking advantage of the symmetry of the image segmentation mapping. The f' boxes contain the same GRBF network; only the inputs differ. The outputs of each f' are summed together to produce the output for f. This construction is used by the recurrent network of figure 7.3.

gaussian centers plus linear and constant terms can do a good job of approximating it.

Although $f' + f'$ is only an approximation to f, the fixed points of the recurrent network that uses $f' + f'$ are the same as the fixed points for one that uses simply f. This is important because the output of the recurrent network is a fixed point.

The recurrent network used to learn the segmentation mapping in this research took advantage of the symmetry by using two feedforward networks whose outputs were added together to approximate f. Figure 7.6 shows how f was represented as the sum of two identical GRBF networks in the recurrent network.

To train the network, good initial parameters were first chosen. It proved very important to start the parameters of the network in a good place as opposed to starting them randomly in order for the segmentation mapping to be successfully learned. This point and the limitations it causes are discussed in section 7.6. The centers of the GRBF network were picked to form a 12 by 12 grid covering the square $[0, 1] \times [0, 1]$. The centers were kept fixed during the learning phase. To set the initial coefficients, 300 input/output examples for one-dimensional segmentation were randomly generated with each example containing 16 pixels. The initial coefficients were then selected using the technique discussed in section 6.3 of treating the output training examples as if they were outputs after one iteration instead of many iterations and then extracting input-output examples for the GRBF network that approximated f'. In effect, the training examples for the recurrent network were used to generate approximate training examples for the GRBF network. The coefficients for the GRBF network were then calculated using the approximate training examples and taking the pseudo-inverse as discussed in section 2.2. The result of this is that the initial recurrent network before training attempted to approximate the segmentation mapping in one step. In practice, the resulting network had good initial values for the coefficients, but left plenty of room for improvement.

After the initial parameters for the GRBF network were selected, the recurrent network was trained using 14 of the input/output examples. The number of iterations that the recurrent network did each time the output of the net was calculated was arbitrarily chosen to be eight. The major consideration that led to choosing eight iterations was that calculating the output of the net is time consuming and repeatedly done during training. In order to allow the training of the recurrent network to complete in a reasonable amount of time, a relatively small number of iterations had to be used. Both recurrent gradient descent and the random step algorithm were tested, and the random step algorithm proved to work better on this particular problem. Figure 7.7 shows the surface of the best f' function learned by the network after 2000 iterations of the random step algorithm on a Sun 4 and 20 iterations on a CM 2 using 4096 processors. The performance of the recurrent network after training is shown in figures 7.8 and 7.9. These figures show the output of the recurrent network learned for doing one-dimensional image segmentation on two different input vectors each containing 48 pixels. One of these test vectors was created by combining three input examples on which the recurrent network had been trained. The other test vector was created by combining three input examples on which the network had not been trained. Using length 48 vectors instead of length 16 vectors to test the

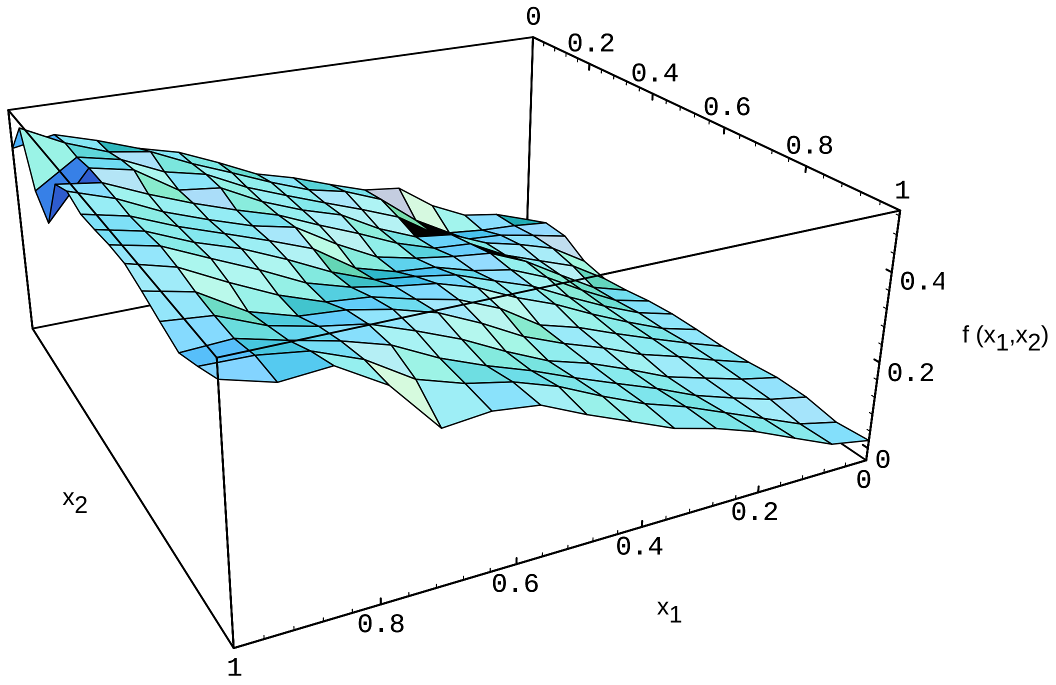

Figure 7.7: Graph of the function f' learned by the recurrent network for one-dimensional segmentation.

performance of the network serves two purposes. First, it allows more information to be displayed at once. Secondly and more importantly, it ilustrates the fact that the recurrent network is scalable. This means that the recurrent network can be used to segment images that are a different size from the training images. This is possible because the recurrent network is made up of identical GRBF networks which can be added as more inputs are desired. The property of scalability is another advantage of this framework over feedforward networks.

To demonstrate that iteration improved the performance of the network, the output of the network is shown in figure 7.10 after one iteration and two iterations given the same input vector used in figure 7.8. By comparing these outputs and the output shown in figure 7.8, one can see that iterating the network improves its approximation to the actual segmentation mapping.

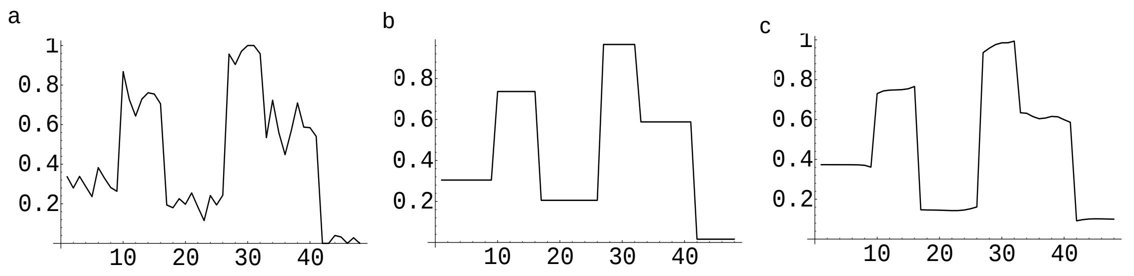

Figure 7.8: a) Input vector obtained by combining three length 16 input vectors which the recurrent network was trained on. b) Target output vector. c) Output of the recurrent network that was trained to learn one-dimensional segmentation. The network was iterated for eight time steps to get this output.

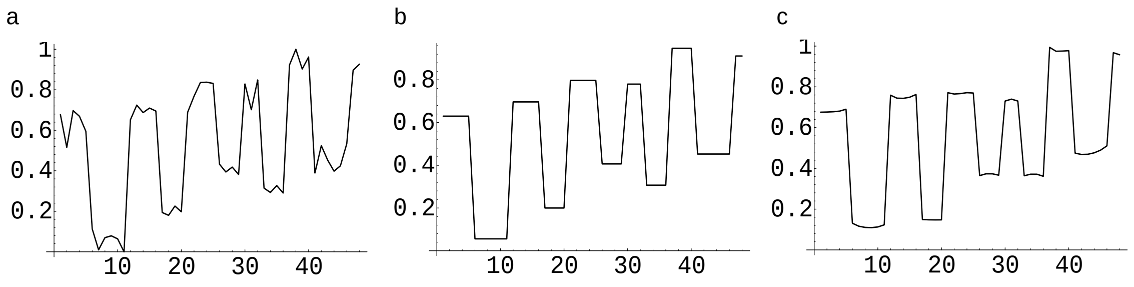

Figure 7.9: Input vector obtained by combining three length 16 vectors which the recurrent network was not trained on. b) Target output vector. c) Output of the recurrent network after iterating for eight time steps.

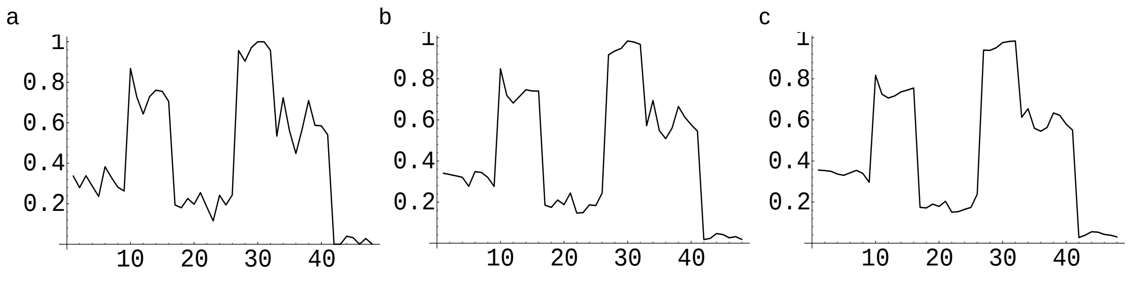

Figure 7.10: a) Input vector given to the trained recurrent network. b) Output of the recurrent network after one iteration. c) Output of the recurrent network after two iterations. This demonstrates that each iteration improves the output and therefore that the network is taking advantage of iteration instead of trying to approximate the function in one step.

7.3 Learning two-dimensional segmentation

After successfully learning the one-dimensional segmentation mapping, another recurrent network was tested on its ability to learn the two-dimensional mapping. The inputs to the recurrent network for the two-dimensional problem were the matrix of pixels composing an image. Each GRBF network contained in the recurrent network computed the output for one pixel at the next time step given the current value of the pixel and its four neighbors. The GRBF network thus had five inputs. As in the one-dimensional case in which the GRBF network had three inputs, this highly non-smooth function that the GRBF networks had to approximate required too many centers and training examples to be practical to learn. However, the same technique of using the symmetry inherent in the function can be employed to circumvent this problem. Thus, the five input function, f, described by equation 7.1 can be broken up so that

$$\begin{aligned} u_{x,y}^{t+1} &= f(u_{x,y}^t, u_{x,y+1}^t, u_{x+1,y}^t, u_{x,y-1}^t, u_{x-1,y}^t) \\ &\approx f'(u_{x,y}^t, u_{x,y+1}^t) + f'(u_{x,y}^t, u_{x+1,y}^t) + f'(u_{x,y}^t, u_{x,y-1}^t) + f'(u_{x,y}^t, u_{x-1,y}^t) \end{aligned} \quad (7.5)$$

where the two input function f' is defined as

$$f'(u_{x,y}^t, u_{z,w}^t) = \begin{cases} (u_{x,y}^t + 4u_{z,w}^t)/20 & \text{if } |u_{x,y}^t - u_{z,w}^t| \le T \\ u_{x,y}^t/4 & \text{if } |u_{x,y}^t - u_{z,w}^t| > T. \end{cases} \quad (7.6)$$

Since f can be approximated by the summation of four simpler functions, the feedforward networks that make up the recurrent network are four identical GRBF networks whose outputs are summed. This scheme was used in the recurrent network and is illustrated in figure 7.11. As with the one-dimensional case, an GRBF network with 144 gaussian centers was used to learn f'. Ten iterations of the recurrent network were used to get its output.

To train the recurrent network to learn the two-dimensional segmentation mapping, six by six unsegmented images were generated for the input training examples and the corresponding segmented images were computed for the output examples. Since each image had 36 pixels, the recurrent network had 36 inputs. Such small

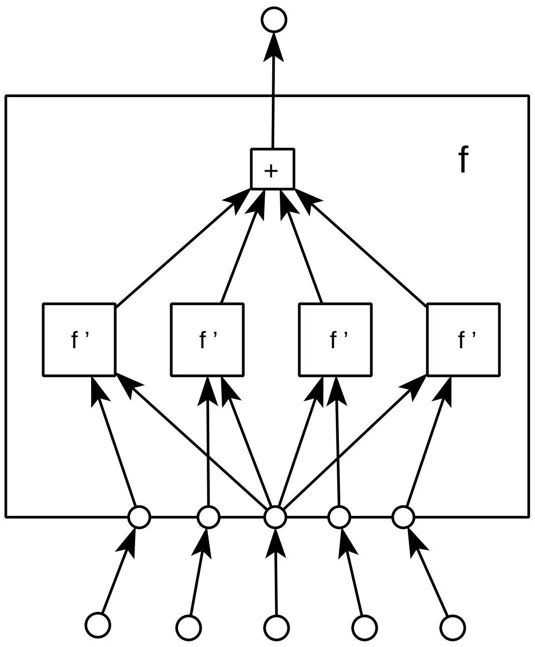

Figure 7.11: This figure is analogous to figure 7.6 and shows how f was split into four identical GRBF networks, f', to be used in the recurrent network for learning two-dimensional segmentation. The outputs of each copy of f' are summed to produce the output for f.

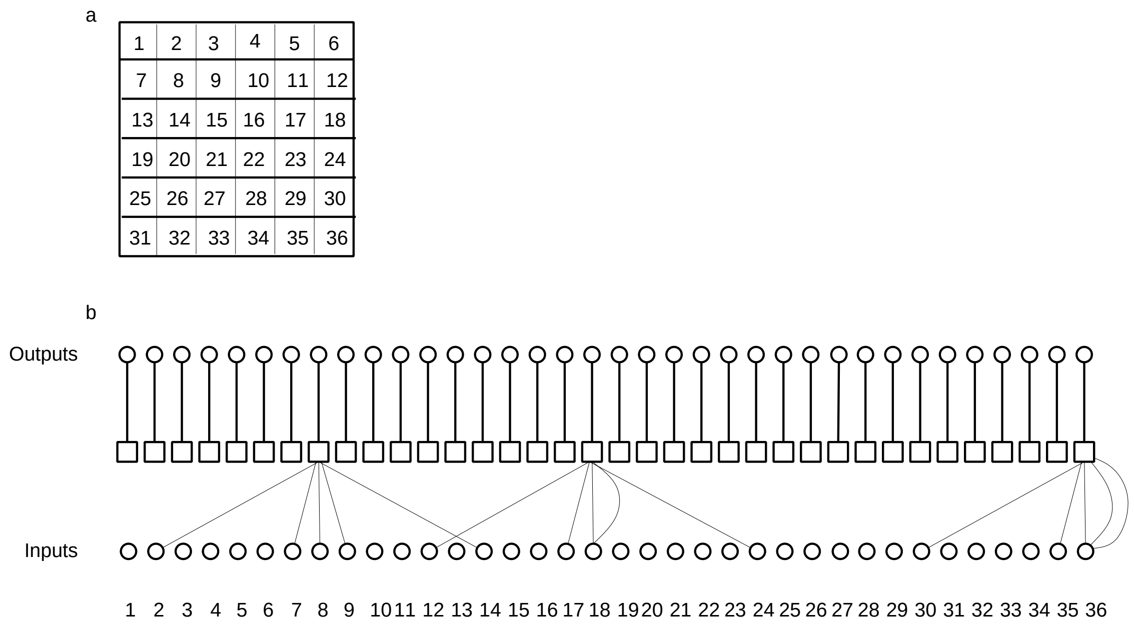

Figure 7.12: a) The pixels of a 6×6 image were numbered as shown so that the images could be given to the recurrent network as one-dimensional vectors. b) This recurrent network was used to learn the two-dimensional segmentation mapping. Each box contains a copy of the function f whose representation is shown in figure 7.11. Only the connections for three of the f boxes are shown so that the figure is less cluttered. The f boxes for boundary pixels such as 18 and 36 that are missing some neighbors compensate by having extra inputs from themselves. Also, the feedback connections from outputs to inputs are not shown.

"images" were chosen in order to make the training time reasonable. The architecture of the recurrent network used is shown in figure 7.12. A typical training example pair is shown in figure 7.13.

The 144 centers of the GRBF network were chosen exactly as in the one-dimensional case: they formed a grid covering the square $[0,1]\times[0,1]$. The initial coefficients of the network were selected by extracting two-input, one-output examples from the training images as described in section 6.3 in order to train the GRBF network to give a rough approximation to f'. Once the initial parameters of the network were set, the recurrent network was trained on 10 input-output image pairs using the random step algorithm on a Sun 4 for 1700 iterations and then on a CM 2 with 4096 processors for

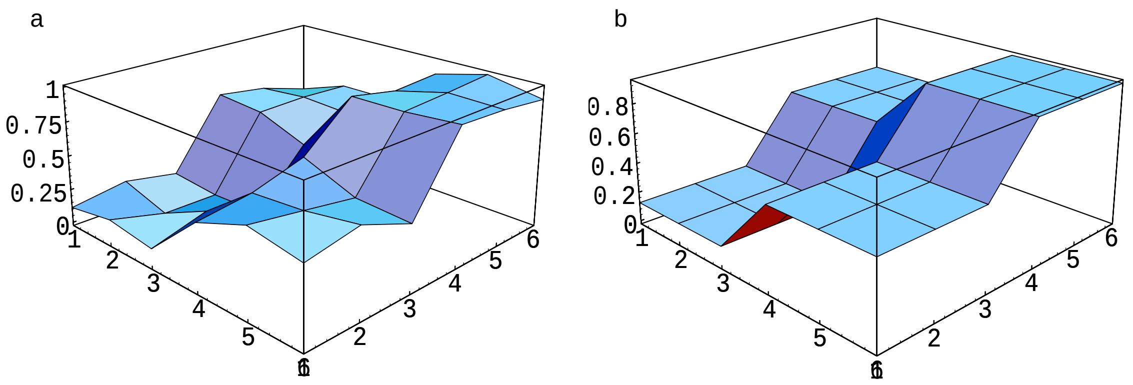

Figure 7.13: Typical two-dimensional training example. Figure (a) is the input image and (b) is the output image obtained by running Hurlbert's algorithm on the input.

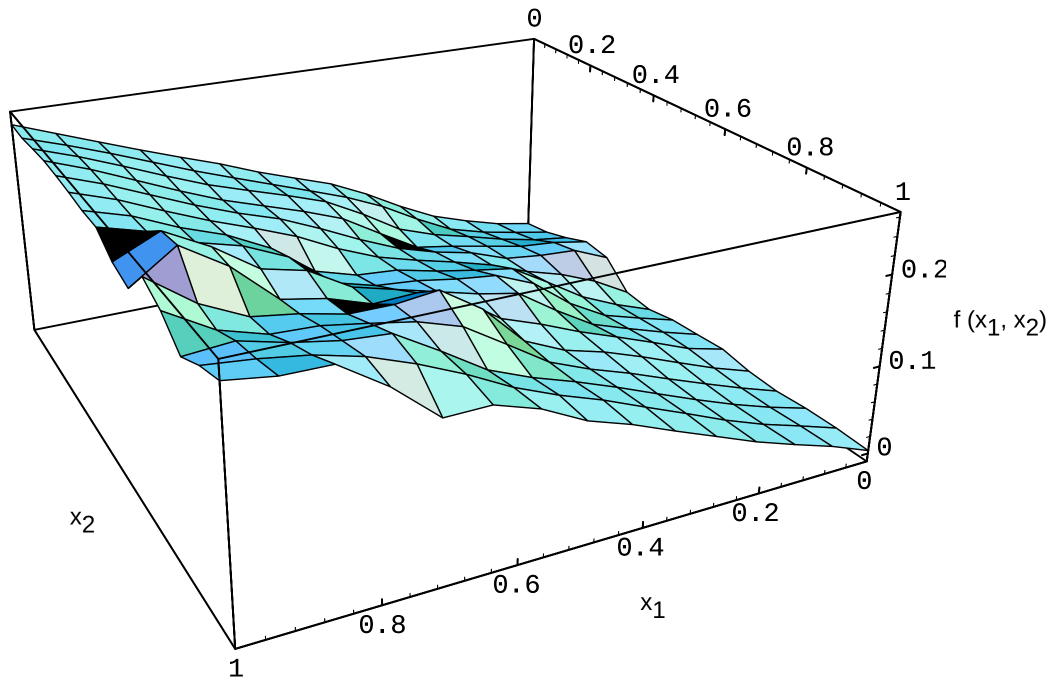

Figure 7.14: Graph of the function f' learned by the recurrent network.

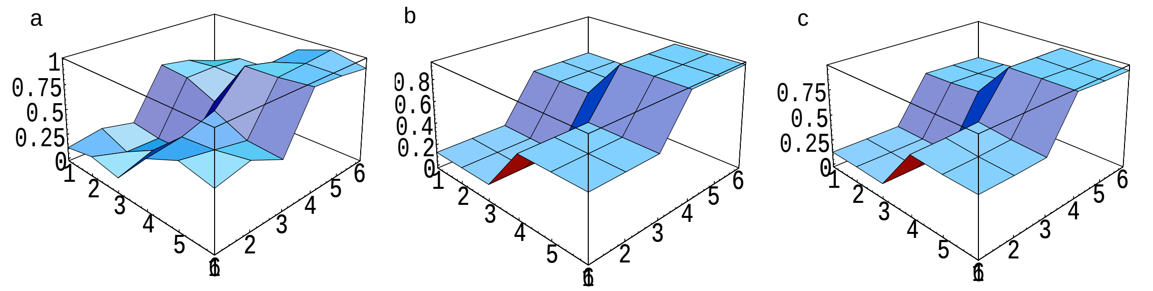

Figure 7.15: a) Input image used in training the recurrent network. b) Target output image. c) Output of the recurrent network trained to learn two-dimensional segmentation. The network was iterated for ten time steps.

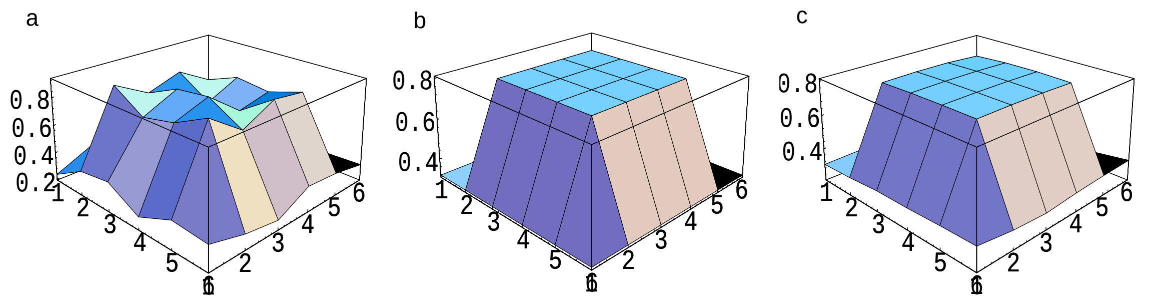

Figure 7.16: a) Input image not used in training the recurrent network. b) Target output image. c) Output of the recurrent network trained to learn two-dimensional segmentation. The network was iterated for ten time steps.

31 iterations. The surface of the function learned by the GRBF networks is shown in figure 7.14. The performance of the recurrent network on two different training images is shown in figures 7.15 and 7.16. These figures show that the recurrent network has done a good job of learning the two-dimensional segmentation algorithm.

As in the one-dimensional case, the output of the recurrent network approximates the desired segmentation mapping more closely as the network is iterated. The improvement of the network's output as it iterates is shown in figure 7.17. This shows that the recurrent network is taking advantage of iteration in order to approximate the image segmentation mapping.

The trained recurrent network was also tested on an actual image that was 320

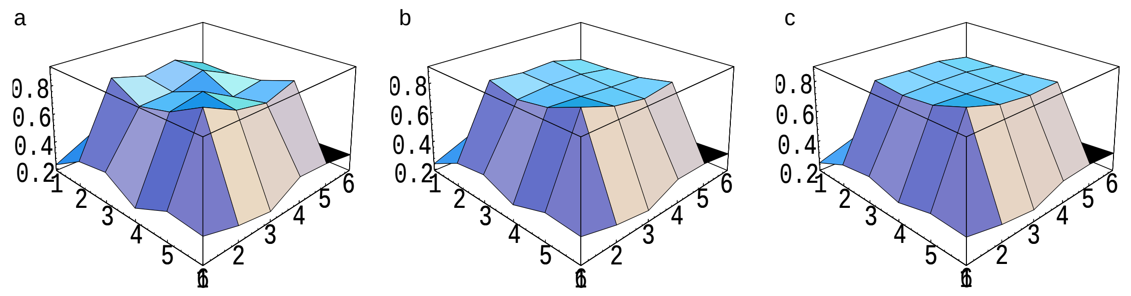

Figure 7.17: a) Input image b) Output of the recurrent network after one iteration. c) Output of the recurrent network after two iterations.

by 200 pixels in size. To do this, the GRBF network which was previously learned was put into a new recurrent network with 64000 ($= 320 \times 200$) inputs and hence 64000 copies of the GRBF network. The same pattern of connectivity was used as in the 36 input recurrent network. This illustrates once again that the recurrent network is scalable. The original image given to this recurrent network as input is shown in figure 7.18. The output of the recurrent network after 300 iterations is shown in figure 7.19. Figure 7.20 shows the result of running Hurlbert's actual segmentation algorithm on the original image. Comparing the network's output to the original image shows that the network has done a descent job of segmenting the image. The rightmost three quarters of the face are mostly one shade of gray while the shadow on the left quarter has been kept separate. The face has retained the nose and mouth features. Looking at the output from Hurlbert's algorithm in figure 7.20 reveals more smoothing than the output from the recurrent network. The nose and mouth have been mostly smoothed over. It appears that the threshold learned by the recurrent net is smaller than the one used by Hurlbert's algorithm which results in more features remaining distinct from others. This indicates that the function learned by the recurrent network is not exactly the one it was being trained on. However, the network still performs reasonably well.

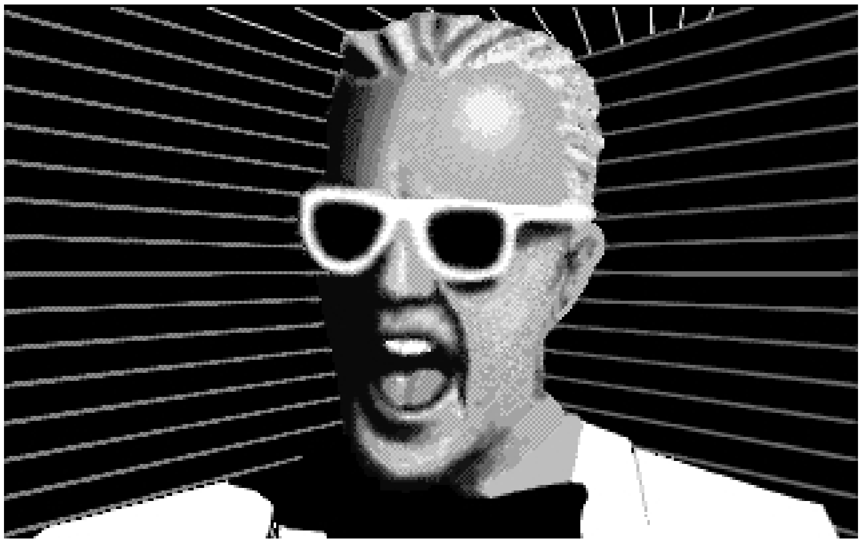

Figure 7.18: Real image given as input to the trained recurrent network.

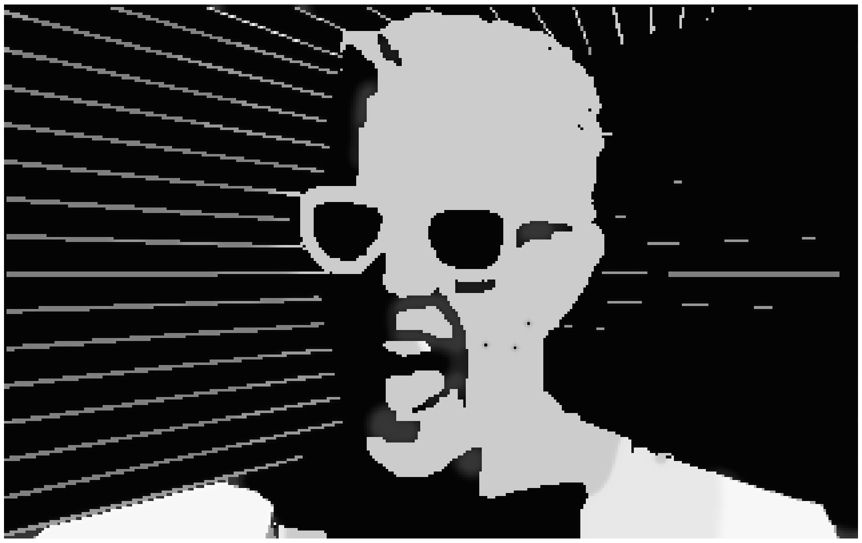

Figure 7.19: Output of the recurrent network after 300 iterations.

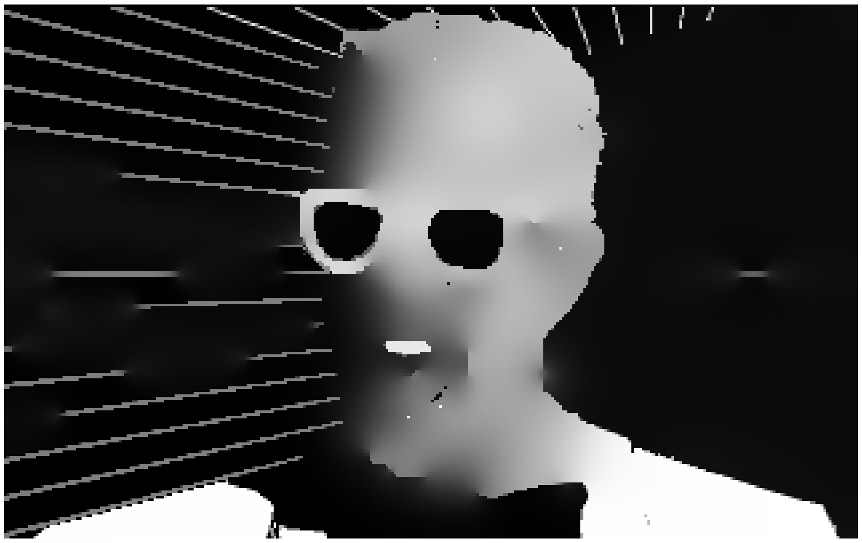

Figure 7.20: Output of Hurlbert's algorithm after 300 iterations.

7.4 Learning image segmentation on a feedforward network

The claim has been made that the recurrent network for learning image segmentation is superior to a feedforward network. This claim is supported by the theoretical results of [Sto82] concerning the curse of dimensionality since the recurrent network need only learn a low-dimensional function. This claim can also be verified by a simple experiment.

To test the ability of a feedforward network to learn the image segmentation mapping and compare it against the recurrent network, an RBF network with 16 inputs, 16 outputs and 283 gaussian centers along with linear and constant units was used. The same 300 training examples used with the recurrent network for learning one-dimensional image segmentation were used to train the RBF network. The centers were chosen to be the first 283 examples (each of which was a vector of length 16). With the centers fixed, the equation for the optimal coefficients is linear and thus, a matrix inversion was used to find the best coefficients. This method resulted in the RBF network learning the training examples perfectly. However, the network exhibited very poor generalization. An example of a typical output for the

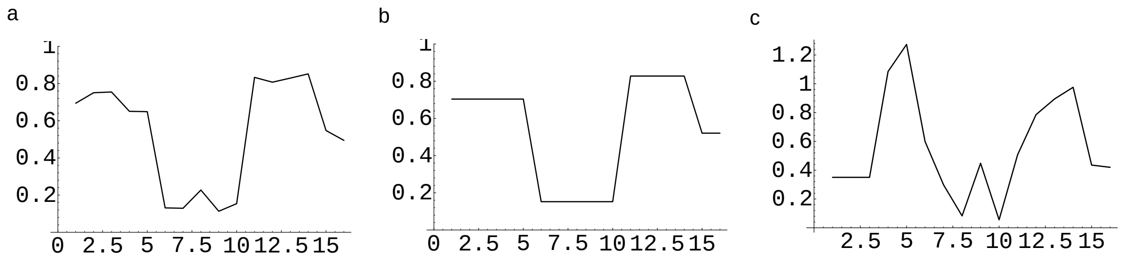

Figure 7.21: a) Input vector given to the trained feedforward network. b) Desired output of the network. c) Actual output of the network. This shows very poor generalization by the trained RBF network.

RBF network given an input vector on which the network was not trained is shown in figure 7.21. The figure shows that the RBF network performs very poorly on this non-training example. The poor performance of the feedforward network trained on 300 examples indicates that many more training examples are needed for this network to successfully learn the one-dimensional image segmentation mapping. Hence, the recurrent network has a clear advantage for this problem. It should also be noted that the feedforward network is not scalable. If one wanted to use inputs vectors of length 48, a new feedforward network would have to be trained.

7.5 The effect of smoothness on learning with recurrent networks

One question that the experiments using recurrent networks raise is how much of the difficulty of learning image segmentation is due to the fact that the mapping is discontinuous and how much is due to the fact that the network is iterated many times to get an output. In order to explore this question, a smooth version of the segmentation mapping was used to generate smooth output examples for one-dimensional segmentation given the same input vectors used before. To accomplish this, the function f' shown in figure 7.5 was smoothed according to the equation

$$f'_s(x,y) = \sigma(-|x-y|+T)\,(x+2y)/6 + \sigma(|x-y|-T)\,(x/2) \tag{7.7}$$

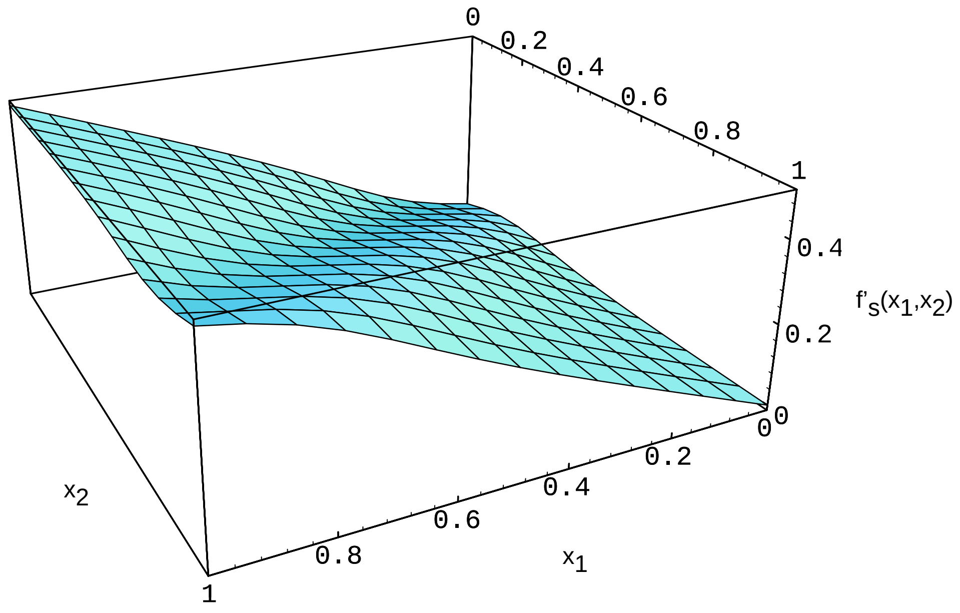

Figure 7.22: Graph of the function f_s' used for the smooth segmentation mapping.

where T is a threshold (picked to be 0.2) as before and σ is the logistic function,

$$\sigma(x) = \frac{1}{1 + e^{-10x}}. \tag{7.8}$$

This function is pictured in figure 7.22. The three input, one output function, f_s that calculates a pixel's next value given the neighborhood of pixels around it can then be expressed as

$$f_s(u_{x-1}^t, u_x^t, u_{x+1}^t) = f_s'(u_x^t, u_{x-1}^t) + f_s'(u_x^t, u_{x+1}^t) \tag{7.9}$$

which is the smooth analog of the function f in equation 7.2. The function f_s was then used as f was before to calculate output vectors for smooth segmentation. A typical training example is shown in figure 7.23. The output vectors for smooth segmentation were generated after iterating the smooth segmentation mapping for eight iterations as opposed to iterating until convergence in the case of normal segmentation. This is due to the fact that smooth segmentation will smooth over object boundaries if iterated enough.

Two identical recurrent networks, each starting with the same initial parameters, were trained concurrently. Both networks were given the same input vectors, but

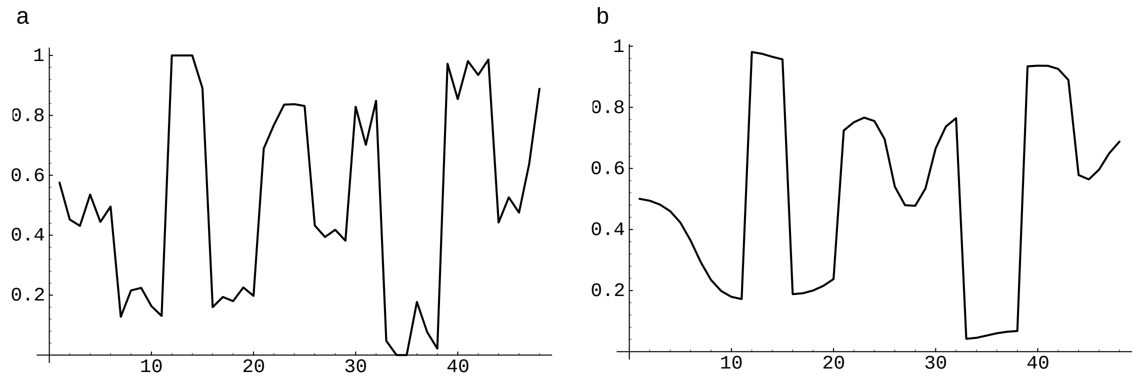

Figure 7.23: Typical training pair for smooth segmentation. Figure (a) is the input vector and (b) is the corresponding output vector.

the output vectors for the two networks were different. Network 1 was given output vectors segmented normally while network 2 was given output vectors created using the smooth segmentation mapping. The initial L2 error for network 1 was 0.30 while the initial error for network 2 was 0.85. This means that the initial parameters were closer to approximating the normal segmentation mapping than the smooth one. After training each network for the same number of iterations using the random step algorithm, the final error for network 1 was 0.09 and for network 2 it was 0.11. The error for network 2 improved much more than network 1 in the same amount of time, but network 1 still had a lower final error. This probably means that learning to approximate a smooth mapping is easier, but the iterative nature of the recurrent network also adds to the difficulty of the learning task. More experiments are needed to explore this issue further.

7.6 Discussion of learning with recurrent networks

The previous sections have shown that a recurrent network using the architecture of figure 5.1 can learn the image segmentation mapping given examples of unsegmented and segmented images. This provides some motivation for using recurrent networks over feedforward networks for learning high dimensional fixed point mappings. In order to learn the image segmentation mapping, however, a number of a priori assumptions were made. First of all, the connectivity of the recurrent network and the number of centers to use in the GRBF nets was predetermined instead of learned. The problem of learning the network architecture and number of units to use is not particular to this research, and has been studied by other researchers in other contexts ([Pla91], [MS89]). Currently, there does not seem to be a principled way of overcoming this difficulty. Other a priori knowledge that was used to learn segmentation was the method of initializing the parameters of the network intelligently and the use of symmetry to break up f into lower dimensional functions. The effectiveness of the method of initializing the parameters presented in section 6.3 depends on the particular fixed point mapping being learned. In the case of image segmentation it provided a good starting point although in other problems this method of initializing the parameters may not result in a good starting point. The use of output to output examples as well as input to output examples to generate the training examples for initializing the GRBF net will be valid in general for learning any fixed point function since the output vectors by definition map to themselves. One should keep in mind that finding good initial parameters can greatly simplify the learning task, but not starting with good initial parameters does not make the problem impossible. It just makes the learning process much slower.

The use of symmetry, on the other hand, was specific to the image segmentation mapping and will not apply to all functions. It is important to realize the difficulty of the task that the recurrent network is being asked to perform. The network is given an input vector from which to start and is told by way of the output vector where it should end up, but it is not given any hint of the intermediate steps that it must go through to get there. The network must discover for itself a way of getting from the starting point to the ending point and the way it discovers must work for all input-output pairs it is given as well as generalizing to ones it has not seen. One might compare this to learning to build an engine by being given all the parts of an engine along with an already constructed engine and being told to learn all the steps needed to build the engine oneself. It is not too surprising that some a priori knowledge is important for succeeding at this task.

Chapter 8

Conclusion

This thesis has studied the ability of recurrent networks to learn fixed point mappings. Recurrent networks have been proposed for learning fixed point mappings by a number of researchers, however, since this task can also be accomplished using a simpler feedforward network, the question of why a recurrent network should be used over a feedforward network arises. This question has received little attention in the literature. This thesis addresses this question by proposing a way of using recurrent networks that takes advantage of their properties in order to ease the problem of learning high-dimensional fixed point mappings. The idea is to decompose a high-dimensional function into many identical low-dimensional functions and then use iteration to converge to the original high-dimensional mapping. Two learning algorithms were presented for training such a recurrent network and experiments were run to test a recurrent network on the problem of learning an image segmentation mapping. The recurrent network learned this mapping reasonably well by taking advantage of some a priori knowledge. The use of a priori knowledge such as assumptions about symmetries was found to be important and may limit the usefulness of using recurrent networks for learning some high-dimensional mappings.

This work can be extended in a number of ways. One extension would be to try learning a different mapping such as stereo disparity to see if the recurrent network performs similarly. Also, different types of feedforward networks such as sigmoidal networks could be tested in the recurrent network in place of the RBF networks that were used in this research. A longer range and more ambitious idea is to study how any function (not just local and uniform functions) can be represented and learned by a recurrent network. Since recurrent networks have been shown to be able to compute any function computable by a Turing machine, it should be possible for a recurrent network to learn any such function.

Bibliography

[Alm87] Luis Almeida. A learning rule for asynchronous perceptrons with feedback in a combinatorial environment. In *Proceedings of the IEEE First International Conference on Neural Networks*, volume 2, pages 609–618, 1987.

[BCG82] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning ways, for your mathematical plays.* Academic Press, 1982.

[BGHS91] H. Behrens, D. Gawronska, J. Hollatz, and B. Schurmann. Recurrent and feedforward backpropagation for time independent pattern recognition. In *International Joint Conference on Neural Networks*, pages II:591–II:596, 1991.

[BL88] D.S. Broomhead and David Lowe. Multivariable functional interpolation and adaptive networks. *Complex Systems*, (2):321–355, 1988.

[CG90] Bruno Caprile and Federico Girosi. A nondeterministic minimization algorithm. A.I. Memo 1254, MIT, 1990.

[Cyb89] G. Cybenko. Approximation of superpositions of a sigmoidal function. *Neural Networks*, 1989.

[Elm90] J.L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.

[FS88] J. Doyne Farmer and John J. Sidorowich. Exploiting chaos to predict the future and reduce noise. Technical report, Los Alamos National Laboratory, 1988.

[GM90] Eric Goles and Servet Martinez. *Neural and Automata Networks: dynamical behavior and applications.* Kluwer Academic, 1990.

[GMC$^+$92] C.L. Giles, C.B. Miller, D. Chen, G.Z. Sun, H.H. Chen, and Y.C. Lee. Extracting and learning an unknown grammar with recurrent neural networks. In J.E. Moody, S.J. Hanson, and R.P. Lippmann, editors, *Advances in Neural Information Processing Systems*, pages 317–324, San Mateo, CA, 1992. Morgan Kaufmann.

[Gut90] Howard Gutowitz. *Cellular Automata: theory and experiment.* MIT Press, 1990.

[Hop82] J.J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79:2554–2558, April 1982.

[Hop84] J.J. Hopfield. Neurons with graded response have collective computational properties like those of two-state neurons. *Proceedings of the National Academy of Sciences*, 81:3088–3092, 1984.

[HU79] John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[Hur89] Anya C. Hurlbert. The computation of color. A.I. Technical Report 1154, MIT, 1989.

[Jor86] Michael I. Jordan. Attractor dynamics and parallelism in a connectionist sequential machine. In *Proceedings of the Eight Annual Conference of the Cognitive Science Society*, pages 531–546, Hillsdale, NJ, 1986. Erlbaum.

[KGV83] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, May 1983.

[MP76] David Marr and Tomaso Poggio. Cooperative computation of stereo disparity. *Science*, 194:283–287, 1976.

[MS89] Michael C. Mozer and Paul Smolensky. Skeletonization: A technique for trimming the fat from a network via relevance assessment. In D.S. Touretzky, editor, *Advances in Neural Information Processing Systems*, pages 107–115, New York, 1989. AIP.

[Pea88] Barak A Pearlmutter. Dynamic recurrent neural networks. Technical Report CMU-CS-88-191, Carnegie Mellon University, 1988.

[PG89] Tomaso Poggio and Federico Girosi. A theory of networks for approximation and learning. A.I. Memo 1140, MIT, 1989.

[PG90a] Tomaso Poggio and Federico Girosi. Extensions of a theory of networks for approximation and learning: dimensionality reduction and clustering. A.I. Memo 1167, MIT, 1990.

[PG90b] Tomaso Poggio and Federico Girosi. Regularization algorithms for learning that are equivalent to multilayer networks. *Science*, 247:978–982, February 1990.

[Pin87] Fernando J. Pineda. Generalization of backpropagation to recurrent neural networks. memo S1A-63-87, Johns Hopkins Universitu Applied Physics Laboratory, 1987.

[Pin89] Fernando J. Pineda. Recurrent backpropagation and the dynamical approach to adaptive neural computation. *Neural Computation*, 1:161–172, 1989.

[Pla91] John Platt. A resource-allocating network for function interpolation. *Neural Computation*, 3:213–225, 1991.

[QS88] Ning Qian and Terrence J. Sejnowski. Learning to solve random-dot stereograms of dense and transparent surfaces with recurrent backpropagation. In *Proceedings of Spring Cold Harbor Summer School on Neural Networks*, pages 435–442, 1988.

[RHW86] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing, Vol I.* MIT Press, 1986.

[SCLG91] Guo-Zheng Sun, Hsing-Hen Chen, Yee-Chun Lee, and C. Lee Giles. Turing equivalence of neural networks with second order connection weights. In *International Joint Conference on Neural Networks*, pages II:357–II:362, 1991.

[SSCM88] David Servan-Schreiber, Axel Cleeremans, and James McClelland. Encoding sequential structure in simple recurrent networks. Tech Report CMU-CS-88-183, Carnegie Mellon University, 1988.

[Sto82] Charles J. Stone. Optimal global rates of convergence for nonparametric regression. *The Annals of Statistics*, 10(4):1040–1053, 1982.

[Wol86] Stephen Wolfram. *Theory and applications of cellular automata.* World Scientific Publishing, Singapore, 1986.

[WZ89] Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1:270–280, 1989.