MODELING UNKNOWN DYNAMICAL SYSTEMS USING ADAPTIVE STRUCTURE NETWORKS

Dean Edward Cerrato

S.B. Electrical Science and Engineering
Massachusetts Institute of Technology
(1988)

Submitted to the Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1993

Signature of Author_____
Department of Electrical Engineering and Computer Science
September 1993

Approved by_____
Walter L. Baker
Technical Supervisor, Charles Stark Draper Laboratory

Certified by_____
Sanjoy K. Mitter
Professor of Electrical Engineering

Accepted by_____
Campbell L. Searle
Chair, Department Committee on Graduate Students

MODELING UNKNOWN DYNAMICAL SYSTEMS USING ADAPTIVE STRUCTURE NETWORKS

Dean E. Cerrato

Submitted to the Department of Electrical Engineering
September, 1993, in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Electrical Engineering

Abstract

A dynamical system can be modeled from observations of its behavior over time, assuming that the states are observable in the outputs. This modeling problem can be recast as the on-line approximation of a smooth, bounded, quasi-stationary function, in which the values of a set of free parameters are adjusted to accommodate the observed behavior of the system. However, given little or no information regarding the complexity of the system, it is difficult, if not impossible, to determine *a priori* the number of free parameters required to describe the relevant system dynamics.

To complicate matters, operating conditions may not be conducive to inferring system dynamics from observations of its behavior: measurements of system inputs and outputs may be corrupted by noise; or it may be difficult to drive the system into areas of its operating envelope where its dynamics are of interest.

The class of spatially localized network architectures addresses these difficulties by decomposing the global function approximation problem into local subproblems. Since the local approximations can be adjusted relatively independently of one another, their number can be increased on-line to yield better local approximation capability without disrupting performance in other areas. In addition, this type of architecture tends to be robust to nonuniform coverage of the operating envelope.

Spatially localized networks, in tandem with a novel algorithm for adjusting network structure on-line in the presence of noise, are used to construct predictors for two nonlinear dynamical systems: an aeroelastic oscillator, and the Mackey-Glass chaotic time series. Beginning with networks of small size, the algorithm increases the number of free network parameters on-line, as well as adjusting parameter values, to meet a mean prediction error target.

Technical Supervisor:	Walter L. Baker
Title:	Senior Member of Technical Staff, Draper Laboratory
Thesis Supervisor:	Sanjoy K. Mitter
Title:	Professor of Electrical Engineering

Acknowledgments

FINALLY! The part of this thesis I've been waiting a year to write, and probably the only chunk of this voluminous tome that I will have enjoyed writing... ;)

Now, who should I thank first...well, it all began EXACTLY ten years ago (almost to the day) when my parents put their first-born son on a plane bound for that unknown wasteland known as, well, "The North", to most people down in Arkansas, Birthplace of Presidents. I'd like to thank my mom and dad for all the patience, support, and love they've given me over this last decade, as I've gallivanted around in Boston doing that technology thing with never any prospect of coming home to stay. But don't worry, Mama and Daddy, when I'm rich I'm gonna build you a house in the Ozarks.

Then there are those good friends I've made at Draper, both students (= underpaid) and staff (= overpaid, for the same work) during my tenure there as a fun-loving, over-worked grad student. Ruth, Torsten, Jamie, Steve S., Eric, Jess, Steve A., Bill H., Ching, Matt—the list goes on, but you all know who you are. I've had a great time with you guys. Walt deserves a special line as both friend and advisor, someone who really appreciates creativity and original thought, and who let me pursue my own hare-brained ideas, even when he thought they were hokey. Well, Walt, here's 179 pages of hokey for you! Good luck as a grad student. Pete wins the award for most prolific draft reader: his written comments on anything I asked him to read always seemed to be longer than what I had written. However, he was usually right. Lisa, you win the Coolest Librarian Award—you still owe me a sushi party!

Biggest Time Sync (that's an inside joke) goes to the Chorallaries. I've had so much fun singing and otherwise goofing off with you guys this year, I can't imagine how I got through my first year as a grad student without it. You've really made a big Imprint (another inside joke). Making the album in January (instead of doing my thesis...) was a complete total blast! "Climbin' up on Solsbury Hill...I could see the city light...wind was blowin', time stood still...monkey flew out of my butt...etc." Erin and Tom, you're awesome. This year we're gonna kick big bootay. Japan ho!

Skydivers, you're not forgotten! I haven't been very active this season, but I've been broke and busy...but that will change pretty soon. Anyway, collegiates were fun. Can you think of a better way to spend the New Year than sitting around in south Florida in gale force winds, completely grounded? Well, in any case, the two or three jumps we got off were memorable, especially the one where Meredith kicked me in the face when we funnelled our exit. Jim, thanks for buying groceries for two years. Carl, thanks for keeping your mess mostly confined to your room.

Thanks to my distant friends, Kevin and Steve, who have remained close despite the distance.

Well, it looks like I've scrolled over onto another page, which is what I was shooting for all along, since I'd already compiled the Table of Contents, which allocated two pages for this...sorry for the rambliness, but hey! Walt doesn't have to approve this, so I can write whatever I want!

Whatever.

See yaaaaa!

This thesis is dedicated to the memory of Fish K and Marlene.

This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under Independent Research & Development (IR&D) Project #276: *Connectionist Systems and Learning Control.* Draper Laboratory's generous financial support under this project is greatly appreciated.

Table of Contents

1 Learning and Dynamical Systems

1.1 Introduction

1.1.1 Statement of Problem

A model for a dynamical system can be inferred from observations of its behavior over time, assuming that the system dynamics are observable in the system outputs [17]. For example, one could build a predictor for a system by constructing a mapping from recent operating conditions (i.e., inputs and outputs) to future outputs. This and other methods, which acquire permanent knowledge of system behavior through prolonged observation, can be said to *learn* features of the system.

In recent years, various types of network parameterization structures have been successfully employed in the modeling of dynamical systems and other related applications in control and estimation [2, 4, 12, 15, 18, 19, 21, 22, 30]. Their status as universal approximators [6, 9, 29], as well as the ability to be trained incrementally with streams of data generated in real-time by the dynamical systems of interest, has promoted their use in such learning tasks.

The first step in applying such a network to a modeling task is to initialize the network structure, in particular its size (i.e., number of nodes, or units). The network must possess a sufficient number of free parameters to capture the complex behavior of the actual system; falling short results in the inability of the model to accurately predict or emulate behavior following from observed operating conditions. In contrast, an excessively high number of free parameters, while allowing high model accuracy under observed conditions, can enable the model output to deviate drastically for even

slightly novel conditions—the model does not generalize well. From the standpoint of computation, a larger set of parameters typically means greater CPU time and storage requirements. This trade-off between accuracy on the one hand, and generalization and computational efficiency on the other, is an important factor in the success of learning-based applications.

However, assuming no information is available *a priori* regarding the dynamical system, one cannot hope to appropriately initialize the network before learning begins. A tedious, time-consuming, and possibly unreliable cycle of trial and error with networks of different sizes is the design engineer's only recourse to attaining optimal performance.

This thesis presents a novel learning algorithm that enables networks of a specific class to adaptively alter their structures (i.e., increase their size) while being trained. Such a network may be initialized with as few as a single unit; as training proceeds, new units are allocated to alleviate persistently high errors in the network output. Structure adaptation can occur in the face of noise and adversely ordered training data. In addition, a parameter training algorithm has been developed that complements structure adaptation by facilitating rapid incorporation of newly created units into the existing network model.

The performance of the adaptive structure algorithm is evaluated by using it to construct predictors for two nonlinear dynamical systems from observations of their behavior over time.

1.1.2 Thesis Overview

The remainder of Chapter 1 explores the meaning of and requirements for effective learning as performed in applications involving dynamical systems. Section 1.2 discusses the general topic of learning from examples applied to a variety of potential problems, and introduces the special case of learning as function approximation, in which a mapping is constructed to fit a set of input/output examples. Section 1.3 then addresses particular problems encountered when such learning is implemented on-line with dynamical systems under non-ideal conditions.

Chapter 2 delves into the selection of approximation structure for learning tasks involving on-line function approximation. The properties of different types of network structure are compared/contrasted in light of their desirability for learning applications. A class of structures is introduced, basis/influence function networks, that form approximations of unknown functions by interpolating among local approximations constructed during learning (i.e., approximations that are valid within a particular neighborhood of input space). Two interpolation methods are discussed in detail: the first, based on Gaussian functions, is familiar from the literature; the second, which relies on inverse square functions, is novel and possesses some very interesting properties that are potentially useful for learning.

Chapter 3 presents ideas, both new and old, that deal with methods of training network parameters to form an adequate approximation to the function that describes the observed data. Specifically, various gradient methods are discussed that seek to reduce approximation error for each example. A hybrid approach, combining features that both reduce approximation error directly and organize network resources to better respond to errors that accompany the training examples, serves as the basis

for the learning algorithm developed in this thesis, to which new features are added to improve learning results in on-line scenarios involving dynamical systems.

In Chapter 4, the idea of adaptive structure learning is introduced and an algorithm developed. It is suggested that approximation structures need not be fixed and invariant during learning, but rather that the structure may be augmented or reduced as is deemed necessary to obtain a more accurate or more streamlined approximation. Previous attempts to solve similar problems are summarized and discussed in light of the particular problem posed in this thesis. An algorithm is proposed that operates on the structure of a basis/influence function network during learning, in parallel with the hybrid parameter adjustment algorithm mentioned previously: situations are recognized in which a local approximation is insufficient to provide the desired level of accuracy in its vicinity, and in response additional network resources are allocated there.

The adaptive structure algorithm, along with the improved hybrid training algorithm, are used to train basis/influence function networks to predict future states of two nonlinear dynamical systems. Chapter 5 reports results obtained from networks trained to predict the future position of a free-running aeroelastic oscillator simulation given only its current position and velocity. A similar task is performed in Chapter 6 for a quasi-periodic chaotic system described by the Mackey-Glass equation.

Finally, Chapter 7 summarizes the successes and failure of the adaptive structure hybrid algorithm, and recommends further avenues of relevant research in this area.

1.2 What Is Learning?

Webster's Ninth New Collegiate Dictionary gives the following definitions—

learn: (1) to gain knowledge or understanding of or skill in by study, instruction, or experience;
(2) MEMORIZE

knowledge: the fact or condition of knowing something with familiarity gained through experience or association

Both definitions emphasize *interaction with the environment* as a prerequisite for acquiring knowledge, or learning, about that environment. From a technical viewpoint, the following definition seems appropriate—

learning: to infer properties or features of a system, function, procedure, group, or class from a collection of specific examples

Implicit in this view of learning is the assumption that the potentially complex set of examples is generated or characterized by a comparatively simple set of rules; if we can determine what these rules are, then we essentially know everything there is to know about the domain of interest. In the situation where the examples are arbitrary, the best we can do is to *memorize* the entire set of examples, an inelegant and often impractical recourse.

There are a variety of potential "domains of learning", i.e., things we would like to learn; for example[1]:

---

[1]Thanks to Prof. Ron Rivest for these examples, taken from his class 6.858J Machine Learning.

Type of domain	Example
concept	"zinnia"
device	"VCR"
technique	"juggling"
function	"sonar $\mapsto$ distance"
environment	"floor plan"
language	"Greek"
family of similar phenomena	"face recognition" or "character recognition"

Table 1.1. Learning examples

We may want to distinguish zinnias from other kinds of flowers; teach a robot to juggle or to navigate across a cluttered room; or recognize the face of a friend. In general, the learning system (or learner) constructs an internal representation, or model, of the phenomenon of interest that is capable of capturing or emulating certain salient features or behaviors.

1.2.1 Supervised and Unsupervised Learning

Given a particular learning task, the most natural approach to take will depend on the type of features we must learn, as well as the type of information made available for learning.

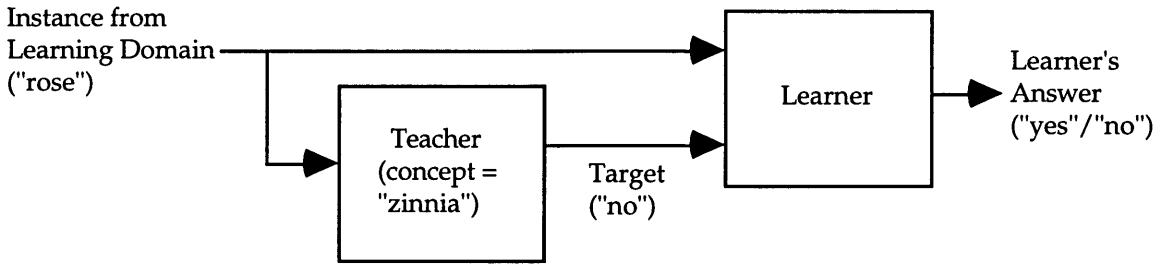


Figure 1.1. Supervised Learning

Perhaps the most straightforward learning method, known as *supervised learning* [7], assumes that each example presented to the learner includes both the input and its corresponding "correct answer", or *target*, which the learner is expected to reproduce or approximate upon future presentations of that input. For instance, if we are learning to distinguish zinnias from other flowers (Figure 1.1), the target for the instance "rose" is "no"; if learning the function f(x) = sin(x), the target for "x = π" is "0.0". In on-line learning scenarios, there may be a *teacher* that receives the same instances from the learning domain as the learner and generates the targets for the learner—it "supervises" the learning.

Supervised learning methods are appropriate for any learning task in which the learner must infer a mapping from inputs to targets, both of which are available for learning. However, in some cases we seek rather to discover interesting features in a set of numerical data, i.e., whether or not a set of data exhibits *self-organization* and in what way.

For example, imagine a pair of random processes, each of which generates vectors in a normal distribution about its own distinct mean vector. A plot of such a set of vectors will reveal two distinct *clusters* centered about the means of the random processes (Figure 1.2). The goal of some learning task might be to determine the means of the two random processes based on the positions of the vectors generated.

This is an *unsupervised learning* problem [7]: since the cluster centers are unknown *a priori*, there can be no clear targets associated with the input vectors. The learner must discover the cluster centers itself solely from vectors generated by the random processes.

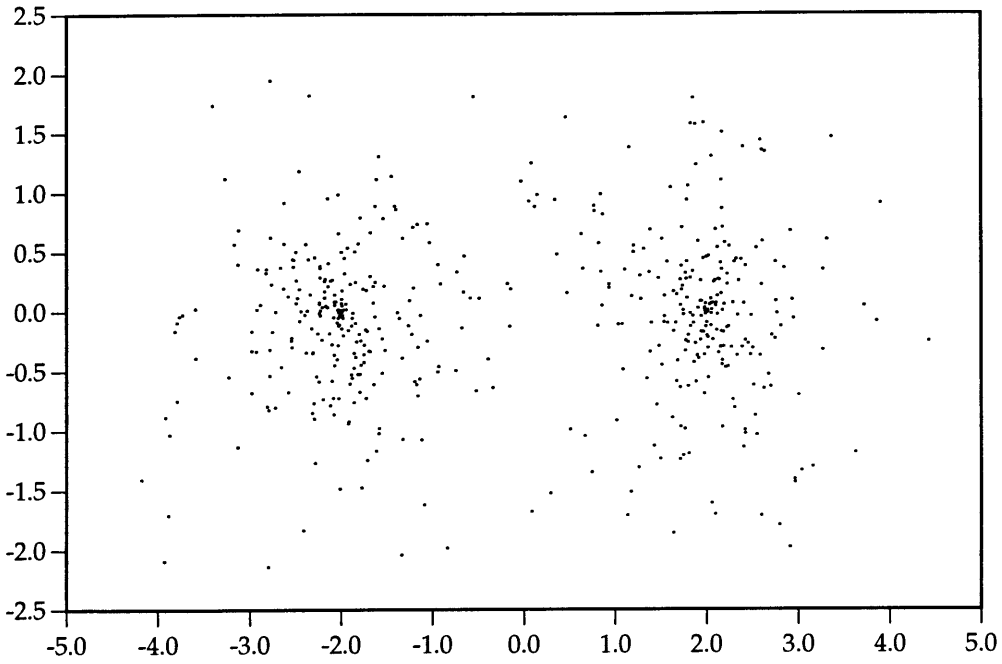


Figure 1.2. Vectors clustered in normal distributions about (-2, 0) and (2, 0)

Another unsupervised learning problem is that of the efficient *encoding* of input vectors [7] (Figure 1.3): representing a set of vectors in fewer dimensions without significant loss of information. The learner is required to project an input vector onto a lower-dimensional space and then to recover a reasonable approximation of the original vector. This is equivalent to equipping the learner with relatively scant resources and then training the learner using each input vector as its own target. Once the learner becomes able to satisfactorily recreate the set of vectors, its internal structure is examined to determine the encoding/decoding mechanism.

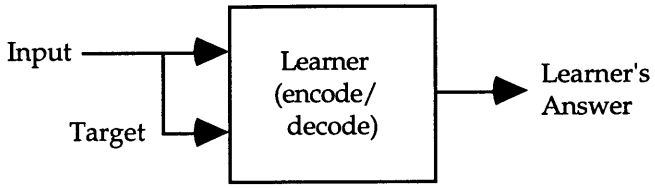


Figure 1.3. Vector Encoding

1.2.2 Batch and Incremental Learning

If the learner has access to the entire set of training examples simultaneously, it is possible to learn in batch mode [7]: updates to the learner's internal model are made only after every example has been presented; thus, performance improvements more likely occur evenly over the entire training set. Alternatively, examples may occur as a stream and appear in any order. In this case, learning may have to be done incrementally [7]: updates are made after each example, improving performance *for that example only*. Hybrid modes are possible, wherein examples are collected from a stream into batches, upon which batch learning is performed; or incremental learning may occur using a stream of individual examples selected arbitrarily from the entire training set.

On-line learning of dynamical systems will be primarily incremental, exploiting real-time (actually simulation time) observations of the system in operation.

1.2.3 Evaluating the Quality of Learning

What does it mean to claim that the learner has been successful in some learning task? Some measures of performance are necessary to gauge the accuracy and dependability of learning. The choice of these will depend on the nature of the learning task and the goals of learning.

We would like the learner to give specific outputs (the targets) in response to specific inputs—we require low error from the learner. In addition, we generally would like unfamiliar yet similar inputs to yield relatively accurate outputs—the learner should exhibit good generalization of its knowledge to unlearned but closely related areas of the learning domain. It is also important that the learner be robust to the order in which training

examples are presented, such that it does not forget information learned from past examples in favor of that from current examples.[2]

1.2.3.1 Training Error and Generalization

During the training phase of supervised learning, the learner adjusts its internal model of the phenomenon of interest so that the outputs it generates in response to inputs chosen from the learning domain are more similar to the targets. The cumulative difference between the targets and the learner outputs over the training set can be represented by a cost function that decreases as the learner outputs become closer to the targets (See Chapter 3). The goal of learning can be recast as a minimization of this cost function.

Depending on the nature of the problem, different types of cost function are appropriate. For example, for the zinnia classification problem presented earlier, the current number of misclassified flowers from the training set could represent the cost associated with the current state of the learner. For learning of numerical information, i.e., function approximation, cost can be measured as some norm of the difference between the target and the learner output over the training set.

But very often the training set does not cover the entire set of possible inputs—after the training phase, the learner may be expected to give accurate outputs to inputs not encountered during training. The *actual* cost associated with the state of the learner is that taken over the entire set of possible inputs, and may differ dramatically with the training cost. The goal of learning, therefore, is to minimize this actual cost, not just the training cost.

---

[2]This assumes that the target function or concept is stationary; in cases where the target is slowly time-varying, some amount of "forgetting" (of out-dated knowledge) may be desirable.

It would be unreasonable to expect the learner to perform well on inputs that are significantly different from those appearing in the training set; for example, by extrapolating far outside the training domain in a function approximation problem. However, it is reasonable to assume, in some cases, that similar inputs should yield similar outputs.[3] (This is especially true for approximation of smooth functions.) A learner is said to generalize well if it is able to maintain a low actual cost in this manner.

To some extent, decreasing the training cost may also decrease the actual cost, but it is possible to minimize the training cost *at the expense of the actual cost.* The learner has at its disposal some amount of representational resources (a.k.a., degrees of freedom, adjustable parameters, weights) with which to construct an internal model of the phenomenon being learned. Some, but perhaps not all, of this representational power will be needed to accommodate the training set examples—that which is leftover is virtually unconstrained and can create wild deviations in the learner output for areas of the learning domain sparsely represented in the training data. The longer training proceeds, the more the training cost may decrease, but the greater the potential for adverse effects caused by unconstrained representational power. The resulting model may represent the training examples well, but may be highly erroneous everywhere else. Actual cost has increased and generalization has suffered. This phenomenon is sometimes known as overfitting to the training data. A simple example appears below of the approximation of a function $f : \Re \mapsto \Re$ with low training error but bad generalization (Figure 1.4a).

---

[3]One notable exception is the class of logical parity functions, where bit strings that differ by only one bit yield different outputs.

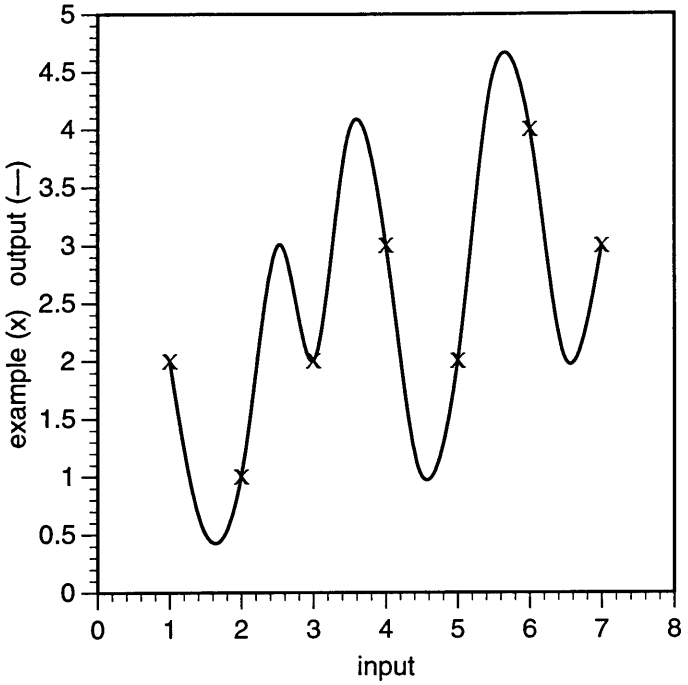


Figure 1.4a. Bad generalization. The training inputs have been learned with very low error, but the approximation for other inputs is unreliable.

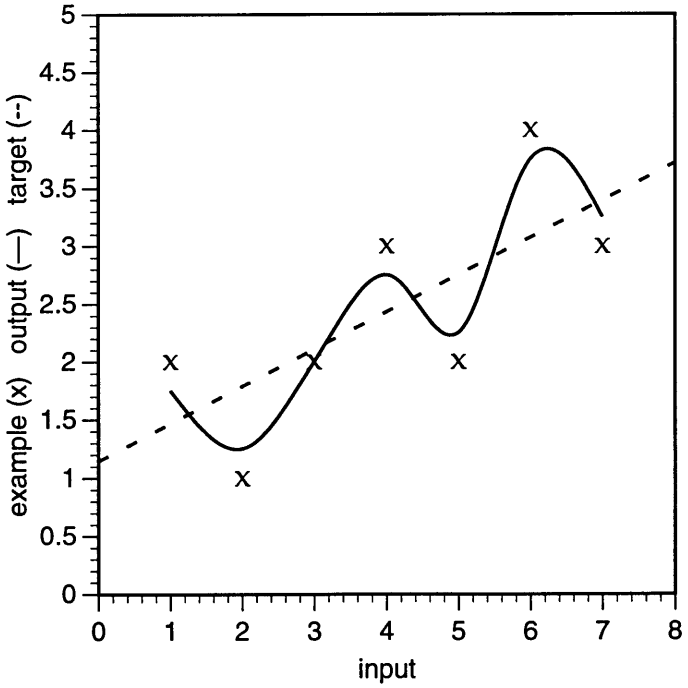


Figure 1.4b. A linear target mapping represented by noisy training examples: the noise has been included in the approximation, resulting in bad generalization.

It should be emphasized that the large swings of Figure 1.4a are a result of unconstrained degrees of freedom. Even if the target mapping contains such large deviations itself, *there is no way to know what those deviations are*—in all likelihood, Figure 1.4a is erroneous for inputs that are between the training examples. The bottom line is that *unconstrained degrees of freedom are to be avoided.*

Generalization can sometimes be improved by smoothing away large deviations, but even this is no guarantee, especially in the case of noisy training data. Figure 1.4b shows an a much smoother approximation for the set of training examples of Figure 1.4a. If the training data were noise-free, then this could be considered an approximation that generalizes well. However, if the target mapping were actually linear (represented by the dashed line), with the training data corrupted by noise, then the approximation would be the result of overfitting to that noise, and generalization would actually be poor.

Overfitting can be combated in either of two ways: empirically, through cross-validation; or by taking steps to reduce the effects of unconstrained degrees of freedom. For cross-validation, a portion of the available data is reserved as the training set before training begins; the remainder will be used to validate the learning as training proceeds. Periodically throughout the training phase, an estimate of the actual cost is calculated over both the training set and the validating set. Training ceases as soon as this estimate begins to increase rather than decrease. Alternatively, the estimate could be calculated only over the validating set, and training stopped when this estimate began to grow significantly larger than the cost taken over the training set. Cross-validation is a useful way to signal the

onset of overfitting, but is only as effective as the validating set is representative of the set of possible examples.

The training cost need not represent only training error—it may also reflect the undesirable contribution of excess learning resources. Thus, it may be possible to improve generalization by minimizing such an augmented training cost. A simple approach that is effective with some kinds of learning architectures (e.g., backpropagation networks [11]) is to add terms to the cost function that are proportional to the squares of the magnitudes of all the adjustable parameters; to minimize the cost, then, the sum of these terms must also be minimized. This is known sometimes as weight decay. In some cases, relatively unimportant or unused resources will tend to be minimized more than resources needed for learning; thus, their contribution to the learner output, and their deleterious effect on generalization, is reduced.[4] Note that weight decay is only appropriate for parameters whose contribution to the output increases with its magnitude.

In the case of approximation of smooth functions, weight decay can enhance generalization by promoting *an increase in the smoothness of the learner's output function* (cf. Figure 1.4) by reducing the magnitudes of swings in the network output that occur between training examples [11]. On the other hand, indiscriminate adjusting of parameters may undermine learning by "erasing" acquired knowledge.

---

[4]Parameters whose magnitudes fall below a certain threshold could be removed completely from the learner's approximation structure. This brings us into the realm of *adaptive structure*, one of the major themes of this thesis.

1.2.3.2 Remembering What Has Been Learned

Incremental learning is the most natural learning mode for a number of scenarios, but significant forgetting can arise as a result of updates being made only to improve performance for the current example. On each update, any of the learner's internal parameters might be adjusted to accommodate the current example, and yet may undo previous adjustments made to accommodate other examples, most likely those most dissimilar to the current example. If examples are presented with enough uniformity over the input space, with sufficiently gradual parameter updates, then forgetting may not be a problem; however, these conditions are by no means guaranteed. Training may remain fixed in a certain region of input space for extended periods, possibly eliminating anything learned in other regions. *Incremental learning minimizes local training error, with the possibility of increasing global training error.*

Forgetting can be reduced by careful selection of learning structure and algorithm. This will be covered more extensively in later chapters.

1.2.4 Learning as Function Approximation

The larger part of this thesis will be concerned with learning tasks in which the target concepts are functions that map numerical inputs into numerical outputs: $f : \Re^n \mapsto \Re^m$ (Figure 1.5). For simplicity, target functions are assumed to be smooth (i.e., once differentiable) and bounded over the domain of interest. The learning mode is assumed to be incremental, performed using streams of training examples (input/output pairs).

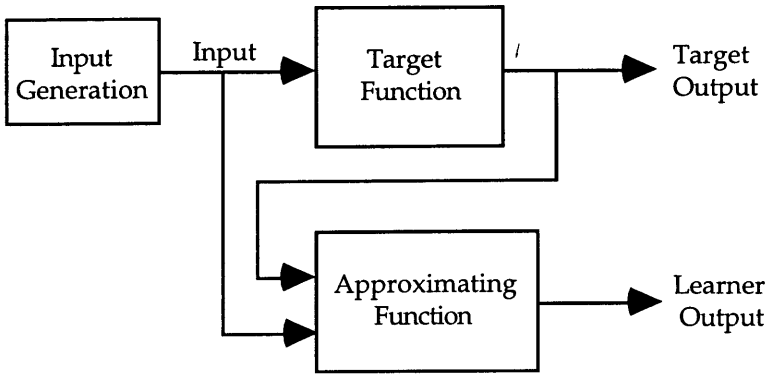


Figure 1.5. Supervised Learning Structure for Function Approximation

From a geometric perspective, each training example is simply a point in the m x n input/output space. The goal of learning is to construct a multidimensional surface that approximately intersects all points closely enough to yield low error over the space, while varying as little as possible between points to preserve generalization. In other words, learning may be viewed as the identification of a manifold that minimizes a cost function based on approximation error and surface energy over a set of vectors.

Since no assumptions are yet made about the unknown target function other than smoothness and boundedness, the choice of approximation structure used by the learner is rather unconstrained—any approximation structure that is known to converge uniformly to any target function as described above is sufficient for learning, e.g., a polynomial approximation. However, there are other considerations that guide the choice of structure, most notably the convenience of incorporating prior knowledge of the otherwise unknown target function into the approximating function before training commences, and robustness to the forgetting that might occur as a result of uneven ordering of examples in the training stream.

1.3 On-Line Learning

The discussion will focus now on scenarios in which incremental learning is carried out under more realistic conditions. Specifically, it will be assumed that training examples are generated in real-time by an actual dynamical system, such that: training examples may appear in the training stream in any order consistent with the (possibly uncontrolled) dynamics of the system of interest; and training data is subject to corruption by noise. Of particular interest is learning in the context of control and modeling of unknown dynamical systems.

Previously, a broad range of functions seemed appropriate for use as learning approximation structures. The assumption of on-line learning, however, further constrains the choice of approximation structure and learning algorithm.

1.3.1 Motivation: Application of Learning to Control

Conventional control methodologies generate control laws based on dynamic models of the plants to which the resulting controllers will be applied. The availability of such models may make it possible to design controllers that meet realistic specifications for performance and stability of the closed-loop systems.

But what is "realistic" is determined in large part by the accuracy of a particular plant model over the operating range of the actual plant. For example, if pole-placement techniques are used to design a controller for a nonlinear plant based on a linearized model of that plant, the accompanying modeling errors may result in instability due to misplaced closed-loop poles. However, as long as performance specifications can be relaxed, such instability can be avoided without changing plant models if the modeling error can be bounded as a function of the operating point of the system and the control

law is then designed according to worst-case scenarios. In the above case, this approach might result in the selection of target poles situated farther than necessary in the left half-plane, yielding a more stable but less responsive closed-loop system.

Model inaccuracy may be unavoidable for practical as well as analytical reasons: more accurate models are often complicated and unwieldy and thus not suitable for practical applications, but simplification of a model inevitably introduces more error. Even when a usable and accurate nominal model is available, actual model parameters may deviate significantly from nominal values—a type of model uncertainty that cannot be eliminated as long as controllers are designed exclusively off-line without reference to the particular system of interest.

Adaptive control strategies have arisen in response to limitations in modeling accuracy, and also as a way to combat the effects of unknown or unpredictable disturbances [1]. As with nonadaptive controllers, an adaptive controller is initially designed off-line using a nominal plant model, but has the added ability to generate *on-line* adjustments to its control law based on the recent observed behavior of the closed-loop system. Adaptive control thus provides some recourse for problems caused by modeling error and plant parameter uncertainty, and may improve system performance.

However, adaptation is essentially a reactive dynamic process; as such, time is needed for an adaptive system to reach a fully adapted state. One can imagine an adaptive controller continually bombarded by extraneous disturbances or unfamiliar dynamics and continually trying to adapt, yet never being completely adapted to the current situation. The effectiveness of the adaptive approach has its limits; but it is possible to improve performance

even further by relieving some of the burden from the adaptive component of the system.

"Disturbances" to the adaptive system may be separated into two types: those that are a function of the operating condition of the plant, i.e., "predictable" disturbances (plant parameter variation, unmodeled time-invariant dynamics); and those that cannot be predicted from the operating condition of the plant (time-varying dynamics, random noise and disturbances). Predictable disturbances can be observed over periods of prolonged interaction with the system and eventually be associated (implicitly) with the operating conditions under which they occur. This enables the controller to *anticipate* predictable disturbances as they happen—rather than simply recognizing them after the fact—and hence respond more quickly than an adaptive system could. By augmenting a purely adaptive control system with a learning component to map operating conditions to appropriate adjustments to the control system, a significant portion of the adaptive burden may be eliminated, along with associated performance limitations and overhead. This approach is known as learning-augmented adaptive control [3].

Learning-augmentation requires the addition of a learning system, consisting of a learner and a facility for generating training data from information within the adaptive system. In learning-augmented direct adaptive control (Figure 1.6a), the learning system supplies the controller with a set of control parameters[5] $\mathbf{k}$ as a function of plant output[6] $\mathbf{y}$. The

---

[5]The learning system could be initialized either with a zero mapping, in which case the adaptive component would provide all initial control parameters in the form of corrections; or with a nominal control law. This will depend on the approximation structure used by the learner and its ability to be so initialized.

[6]Observability of all relevant plant states is conveniently taken for granted.

adaptive component compares plant behavior (in response to a reference input **r**) to that of a reference model, and adjusts the control parameters by $\Delta\mathbf{k}$ for that time step in such a way that the difference in behavior **e** should be decreased for future time steps. The learning system trains the learner to generate control parameters (as a function of plant output) that are more consistent with the observed parameters after adjustment by the adaptive component; posterior learning adjustments $\delta\mathbf{k}$, derived from past (and possibly future) values of $\Delta\mathbf{k}$, are used to construct targets for the learner.

Over time, the learner becomes better able to provide "correct" control parameters, so that the contribution of the adaptive component is reduced. In the limit and under ideal conditions (i.e., no noise or unpredictable disturbances), adjustments by the adaptive component would fall to zero as the learner's mapping of plant output to control parameters more closely approximates the ideal mapping.

Note that the effects of noise and disturbances that are not correlated with plant output can be effectively averaged out by the learner and not necessarily assimilated into the control mapping; the adaptive component will be more likely to attempt to compensate for these. Note also that there are no time constraints imposed on learning that would dictate that learning proceed in sync with adaptation. The learning system may take time out to construct its own posterior estimates of the "correct" control parameters for a particular operating point (e.g., via smoothing) before actually training the learner. The learning system may thus take a view of the system that is more global in time, whereas adaptation is local in time. In addition, to minimize potential interference of the learning system with normal adaptation, the learning system dynamics should be constrained to be much slower than

those of the adaptive component, so that adaptation may operate under quasi-static conditions with respect to learning.

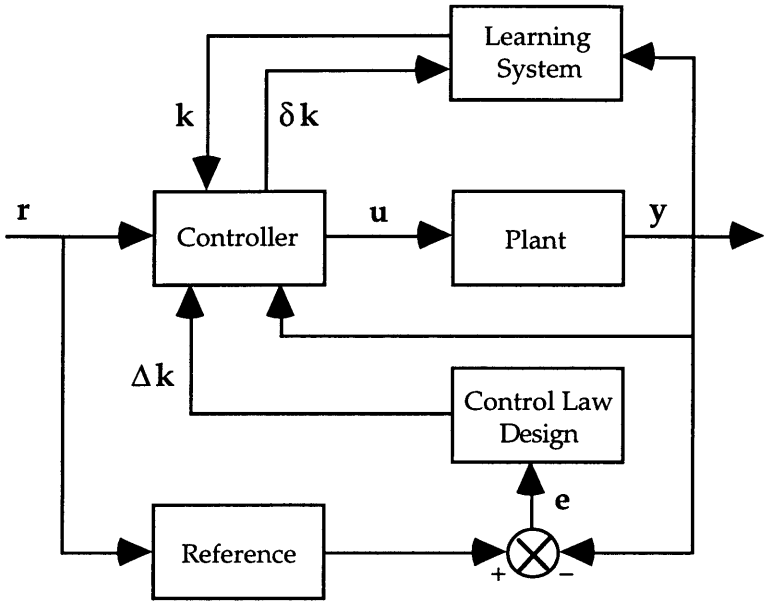


Figure 1.6a. Learning-augmented direct adaptive control

Learning-augmented indirect adaptive control is similar to direct adaptive, but the learner is required to associate plant model parameters rather than control parameters with plant output (Figure 1.6b). In this case, the adaptive component estimates the model parameters **p** based on observed plant inputs **u** and outputs **y**. The learner then must make the association between plant output and model parameter values. Estimates from both the learning system ($\mathbf{p}_l$) and the adaptive estimator ($\mathbf{p}_a$) are made available for on-line design of the control parameters **k**—depending on the assumed reliability of either parameter estimate, the control design may use one or the other or some combination of the two. As with the example of direct adaptive control, the learning process may be delayed relative to adaptation.

The learner is trained on posterior estimates of the model parameters $\mathbf{p}_{post}$ derived from the other estimates.

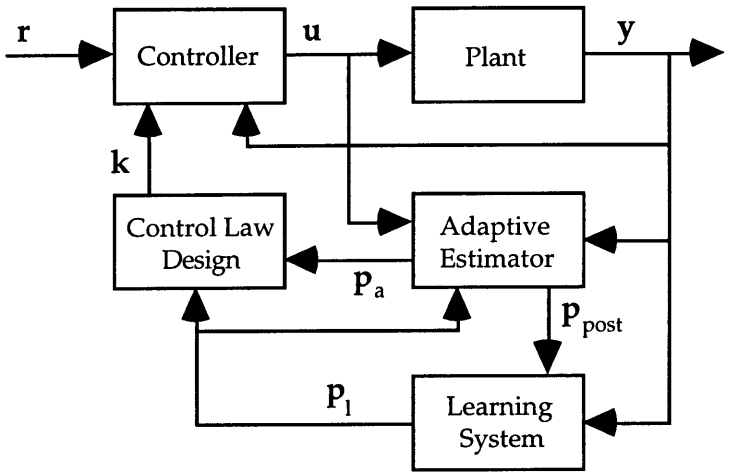


Figure 1.6b. Learning-augmented indirect adaptive control

Learning and adaptation can be viewed as complementary: where learning serves to enable the controller to deal with predictable disturbances, adaptation can be used to handle the remaining unpredictable disturbances; where adaptation reacts immediately to undesirable conditions, learning may proceed gradually and out of sync with adaptation.

Learning-augmented adaptive control is one example of how learning can be used to circumvent common design problems by allowing the controller to mold itself to the actual plant with which it is coupled. In the next section we discuss how learning may be applied to predict the behavior of dynamical systems.

1.3.2 Prediction

Dynamical system behavior can be modeled in different ways. At one extreme, a full-blown dynamic model of the system can be constructed. A

somewhat simpler problem is to predict future outputs of the plant given a set of recent outputs.

A learner can be trained incrementally and on-line in parallel with the plant (or a plant model) (Figure 1.7a).

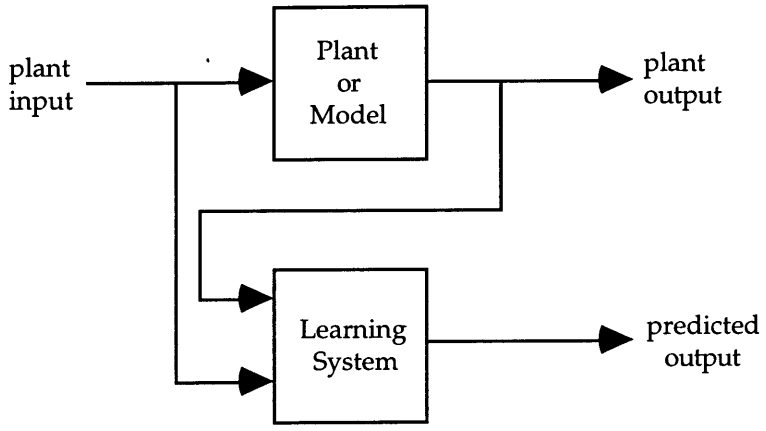


Figure 1.7a: Learning as prediction

At each time step, the learning system (Figure 1.7b) collects past and present inputs and outputs into an information vector that the learner is trained to associate with the plant output some number of time steps in the future. Two conditions are necessary for learning to be successful: i) the plant state must be observable in the outputs used for learning; and ii) the "plant history" carried by the information vector must be sufficiently extensive to convey future plant behavior. If either condition is not met, then the learner will not be provided with enough information to infer a precise relationship between past and future outputs.

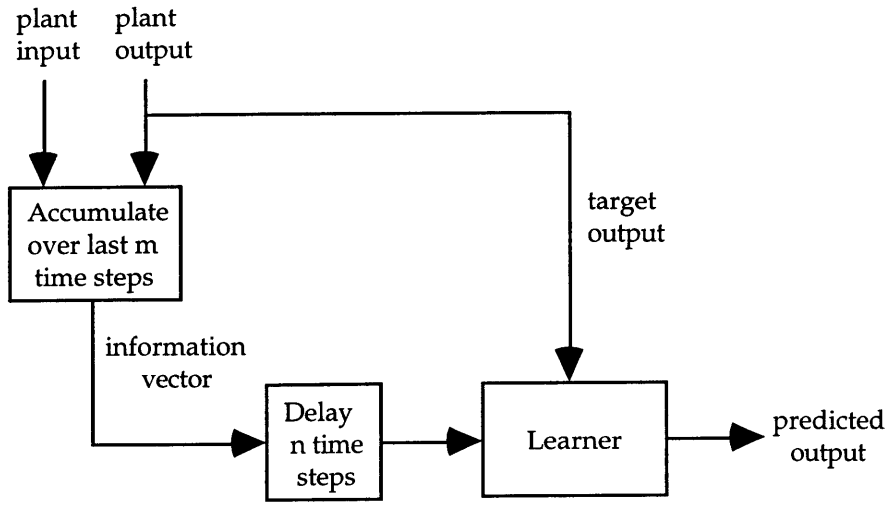


Figure 1.7b. Learning system for prediction

Predicting what the plant outputs will be n time steps hence (or n-step prediction) may be better accomplished via this direct mapping than by calculating from a parameter mapping for the simple reason that the direct mapping subsumes all effects on plant outputs that may be caused by variations in plant parameters in the interval between the present and future times. However, unlike direct n-step mappings, parameter mappings may be used to make predictions for any value of n.

1.3.3 Requirements for Effective On-Line Learning

Crucial to the success of on-line learning applications is the robustness of the learning system to unavoidable real-world conditions. In every example presented, measurements of plant output used as learner input must be assumed to be noisy. Since it is clearly undesirable for the learner to incorporate noise into its mapping, some sort of filtering must occur as a feature of learning.

A stream of training examples is generated by the system as it wanders through its operating envelope. It may not be possible to cause the system to

remain in particular regions of the envelope for extended periods of time, or at all—in fact, in many cases the system will tend to remain near a set point (e.g., regulated systems). This disparity in the amount of training that occurs over the envelope is a potential cause of "forgetting" in the more sparsely covered regions unless a suitable learning structure and algorithm are selected.

We must also allow for the possibility that a reasonable off-line approximation of the target mapping of the learner is available before on-line training begins. The ability to incorporate such prior knowledge into the learner in a straightforward manner provides a "head start" on learning that can significantly decrease learning time. In addition, a learning structure that allows convenient incorporation of explicit knowledge prior to learning may also allow convenient extraction of explicit knowledge after learning.

In light of these issues, the selection of an approximation structure and parameter training algorithm is addressed in Chapter 2 and Chapter 3, respectively. Chapter 4 discusses the idea of learner structure adaptation: adding or deleting resources from the learning structure in order to achieve a better balance between error and generalization. Similar to learner parameter adjustment, learner structure is adjusted when it is determined that the training data are not well represented by the current structure. In contrast to parameter adjustments, which generally occur gradually, structure adjustments are discrete and potentially dramatic. Therefore, it is important that *structure changes not be made in response to noisy individual training examples*, but rather to overall trends in the training data.

1.4 Summary of Chapter 1

It was suggested that more high fidelity models of dynamical systems may be obtained by applying on-line learning methods to the problem. One

of the main contributions of this thesis, adaptive learning structure, was motivated. Special requirements of learning structures and algorithms for training structure parameters were presented. After a discussion of general topics in learning, the focus was narrowed to learning features of real dynamical systems under on-line conditions, accompanied by the presentation of a few examples.

2 The Learning Architecture

Selection of a learning architecture is an important first step in designing a learning system that meets the requirements of on-line learning applications. In addition to possessing a suitably broad approximation capability, the architecture should allow learning from adversely ordered training data, as well as allowing straightforward initialization and subsequent interpretation of the learned mapping.

2.1 Global vs. Localized Representations

Many classes of approximation structures are capable of representing continuous functions with arbitrary accuracy over a finite domain. For example, a polynomial of sufficiently high order can approximate any continuous function arbitrarily well over some interval [28]—

$$f(x) = \sum_j c_j x^j \tag{2.1}$$

Alternatively, a function may be approximated by a weighted sum of Gaussian functions [25, 29]—

$$f(x) = \sum_j c_j \exp\left[-\frac{(x - x_j)^2}{2\sigma_j^2}\right] \tag{2.2}$$

These two types of approximation form representations of functions in distinctly different ways. The polynomial approximation is an example of a global representation of the target function (Figure 2.1a), whereas the Gaussian approximation is a spatially localized representation (Figure 2.1b).

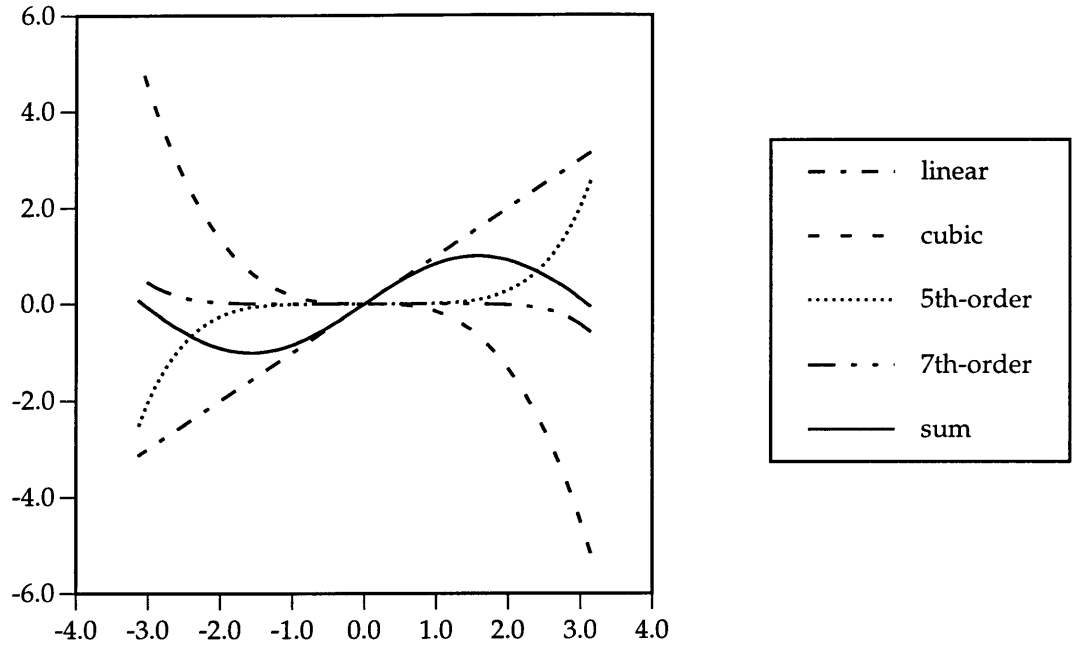


Figure 2.1a: Polynomial approximation of sin(x) on [–π, π]

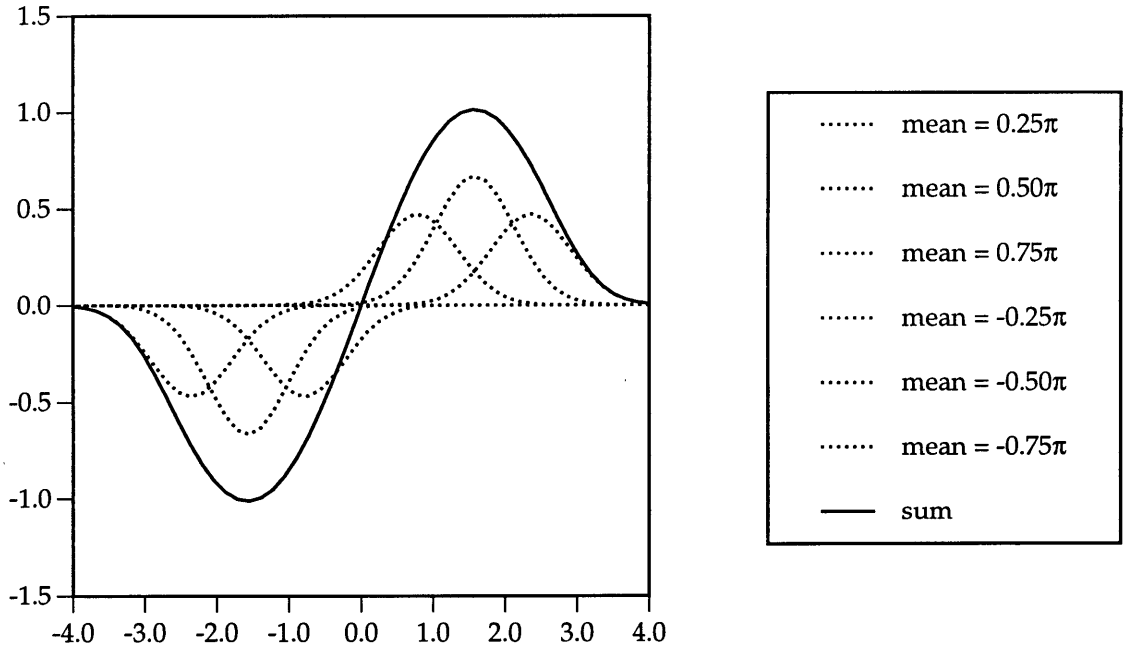


Figure 2.1b: Gaussian approximation of sin(x) on [–4.0, 4.0]

Figure 2.1: Approximation of a sinusoid

The terms "global" and "spatially localized" characterize the scope or "influence" of individual approximation parameters over the input space. If a parameter value of a global representation (e.g., a polynomial coefficient) is changed, the value of the approximation may change significantly over the entire input space. More precisely, for parameter changes Δc_j in response to inputs x from the learning domain X,

$$E\left[\Delta c_j \cdot \Delta f(x)\right] >> 0, \text{ all } x \in X \tag{2.3}$$

In contrast, changing a parameter value of a spatially localized representation (e.g., the magnitude, center, or width of a Gaussian function) will change the approximation significantly only near a specific local region (e.g., near the center of that Gaussian function) and very little far away; thus, *the influence of individual parameters is localized with respect to input space.*

$$E\left[\Delta c_j \cdot \Delta f(x)\right] \approx 0, \ \forall x \in X \quad s.t. \quad \left|x - x_j\right| >> 0 \tag{2.4}$$

Either approach has potential advantages and drawbacks depending on the application. Global representations may be capable of approximating certain functions over large regions of input space using relatively few parameters; for example, three parameters (amplitude, frequency, phase) are sufficient to represent any sinusoid over any interval. Global representations, if implemented in hardware, in some cases offer robustness to individual component failures—sever one wire at random, and the resulting error will be spread over the entire mapping, more likely preserving the functionality of the circuit than if error were concentrated in a local region of input space.

On the other hand, the correspondence of certain parameters to certain input regions that is afforded by spatially localized representations allows the approximation problem to be decomposed to some extent into local

subproblems—this facilitates the incorporation of prior knowledge of the target mapping and subsequent extraction and interpretation of learned knowledge. In the example above, each individual Gaussian function can be set to approximate the target function about its center, such that the superposition of all Gaussian functions then serves as a reasonable initial approximation that can be refined and augmented through subsequent learning.

Localized influence of parameters results in a *decreased tendency to forget during learning*. With many learning algorithms, spatially localized parameters tend to be updated only slightly or not at all in response to "remote" training examples, i.e., examples for which they have small influence. If training becomes fixed in one particular region, only parameters local to that region will be updated significantly—other parameters will remain mostly unchanged until training recommences in their respective regions. This feature can be exploited to defray the cost of computation by training only those parameters that contribute significantly to the current network output.

Whereas global representations may require few parameters, spatially localized representations may require many more parameters due to the limited scope of those parameters. Since every region of the input space may require a set of spatially localized parameters to represent the mapping there, the total number of parameters required increases with the size of the input space. In particular, *the number of parameters generally increases exponentially with the dimension of input space*. Thus, the potential for representational inefficiency may be great, depending on the target mapping.

There are obvious trade-offs that must be considered when deciding between global and spatially localized representations. In the context of on-

line learning applications, however, the advantages of spatial localization are more appealing than those offered by global representations. The next section covers spatial localization in more detail, and introduces a class of approximation structures with this property.

2.2 Spatially Localized Representations

Spatial localization is a feature of a number of architectures that can be viewed as extensions/variations of the Gaussian example presented previously. This section discusses them in detail.

2.2.1 Radial Basis Function Networks

The Gaussian example can be generalized to functions in higher dimensions. Any scalar function of the vector $\mathbf{x}$ whose output is non-negative and monotonically decreasing as the (Euclidean) distance from some center $\mathbf{x}_j^c$ may be used in place of a Gaussian function. Such functions are known as radial basis functions (RBFs), or RBF units, and their weighted superposition yields a radial basis function network (RBFN) [25]:

$$f(\mathbf{x}) = \sum_j c_j R_j\left(\left\|\mathbf{x} - \mathbf{x}_j^c\right\|\right) \tag{2.5}$$

$$\left\|\mathbf{x} - \mathbf{x}_j^c\right\| = \left(\mathbf{x} - \mathbf{x}_j^c\right)^{\mathrm{T}}\left(\mathbf{x} - \mathbf{x}_j^c\right) \tag{2.6}$$

RBFs need not be radially symmetric—distance from center may be calculated with respect to a symmetric positive definite weighting matrix $\mathbf{V}_j$:

$$\left\|\mathbf{x} - \mathbf{x}_j^c\right\| = \left(\mathbf{x} - \mathbf{x}_j^c\right)^{\mathrm{T}} \mathbf{V}_j^{-1}\left(\mathbf{x} - \mathbf{x}_j^c\right) \tag{2.7}$$

As well as having all the desirable features of spatial localization, RBFNs have been shown to be universal approximators of continuous functions [29].

2.2.2 Basis/Influence Function Networks

2.2.2.1 General Description

The idea of local approximations, a feature of spatial localization, can be exploited further if a few changes are made to the simple RBFN architecture, to yield new architectures known as basis/influence function (B/I) networks [3, 12, 30].

Rather than combining network units via superposition, as is done with RBFNs, it is more appropriate (and intuitive) *to interpolate among the local approximations they represent.* The weight c_j of an RBF unit may be thought of as a local approximation that is represented as a constant—refer to this as the basis function; the RBF itself defines the scope of this basis function over the input space—call this the influence function for the local approximation. The network approximation is then the sum of the local approximations *weighted by their relative influences* at each point in space. The relative, or normalized, influence function of a local approximation is simply its fraction of the total cumulative influence of all units over input space; this value varies from 0 to 1.

$$y(\mathbf{x}) = \sum_j B_j\left(\mathbf{x} - \mathbf{x}_j^c\right) \cdot \frac{i_j\left(\left\|\mathbf{x} - \mathbf{x}_j^c\right\|\right)}{\sum_k i_k\left(\left\|\mathbf{x} - \mathbf{x}_k^c\right\|\right)} = \sum_j B_j\left(\mathbf{x} - \mathbf{x}_j^c\right) \cdot I_j\left(\left\|\mathbf{x} - \mathbf{x}_j^c\right\|\right) \tag{2.8}$$

where

$\mathbf{x}$ = input vector
$y(\mathbf{x})$ = scalar output
$\mathbf{x}_j^c$ = center of unit j
B_j = basis function for unit j
i_j = influence function for unit j
I_j = normalized influence function for unit j

(Note that normalization changes the effective influence functions, and in some cases may affect spatial localization. Overall, though, the desirable features are retained with respect to interpolation as opposed to extrapolation. This is discussed further in the next subsection.)

Basis functions need not be constrained to constant functions; indeed, any function may serve as a basis function, as long as spatial localization is preserved (i.e., as distance from input to unit center increases, the influence function decreases faster than the basis function increases). Preferably, each basis function is evaluated about the center of its influence function, i.e., as a function of $\mathbf{x}-\mathbf{x}_j^c$. Constant or affine functions make suitable basis functions, although in some cases a larger set of adjustable parameters may be required to construct good local approximations.

The choice of influence function dramatically affects the nature of the interpolation performed by the network. The next subsection discusses two possibilities.

2.2.2.2 Influence Functions: Gaussian vs. Inverse Square

Gaussian functions are frequently employed as influence functions in B/I networks [3, 12, 30]—

$$i_j(\mathbf{x}) = c_j \exp\left[-\tfrac{1}{2}\left(\mathbf{x}-\mathbf{x}_j^c\right)^{\mathrm{T}} \mathbf{V}_j^{-1}\left(\mathbf{x}-\mathbf{x}_j^c\right)\right] \qquad \mathbf{V}_j \text{ symmetric positive definite} \qquad (2.9)$$

where the weighting matrix $\mathbf{V}$ is symmetric, positive definite. (Unless stated otherwise, assume $c_j = 1.0$ always.) Since Gaussians decrease exponentially with distance squared, there are many possible types of basis function with which they may be paired without losing spatial localization, e.g., polynomial functions of any degree, or sinusoidal functions [3, 30].

This thesis introduces an influence function with interesting properties, the inverse square function—

$$i_j(\mathbf{x}-\mathbf{x}_j^c)=\left[(\mathbf{x}-\mathbf{x}_j^c)^T\mathbf{V}_j^{-1}(\mathbf{x}-\mathbf{x}_j^c)\right]^{-1} \qquad \mathbf{V}_j \text{ symmetric positive definite} \qquad (2.10)$$

A remarkable feature is created by the singularity in this influence function: the normalized influence of a particular unit is always 1 at its own center, and is 0 at the centers of other units. In other words, *at the center of a unit, the network approximation is exactly equivalent to the local approximation (basis function) corresponding to that unit.* In fact, it can be shown that the first derivatives are also equivalent, and that the resulting network mapping is smooth (i.e., once differentiable), assuming smooth basis functions (Appendix A). Thus, the choice of inverse square influence functions allows values and first derivatives of the network approximation to be initialized exactly and arbitrarily as desired.

Consider a 2-input, 1-output network comprised of nine identical units arranged in a 3x3 array. Examples of normalized influences for center, corner, and side units are presented for Gaussian (Figure 2.2) and inverse square influence functions (Figure 2.3).

A few things are readily apparent from the figures: a) the Gaussian influences are smoother than the inverse square influences, whose values are fixed at either 1 or 0 at each unit center; b) influences for both networks are localized (for the most part) within the array of units, but not (as much) outside the array; c) the influences of the corner and edge Gaussian units reach maximum values at inputs far from their centers. These and other key distinguishing features of the two types of influence functions are discussed below.

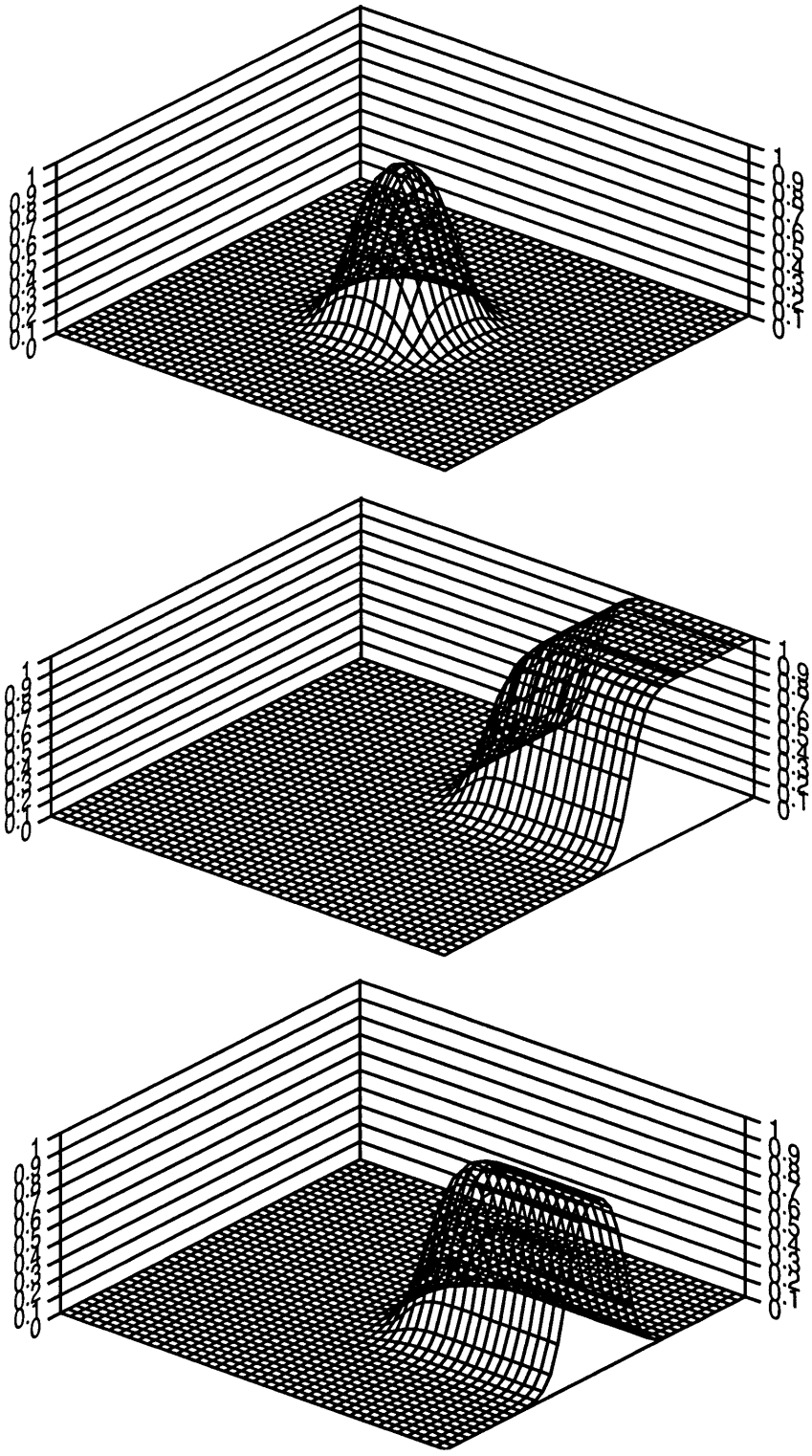


Figure 2.2. Normalized Influences for 3x3 Array of Identical Gaussian Units

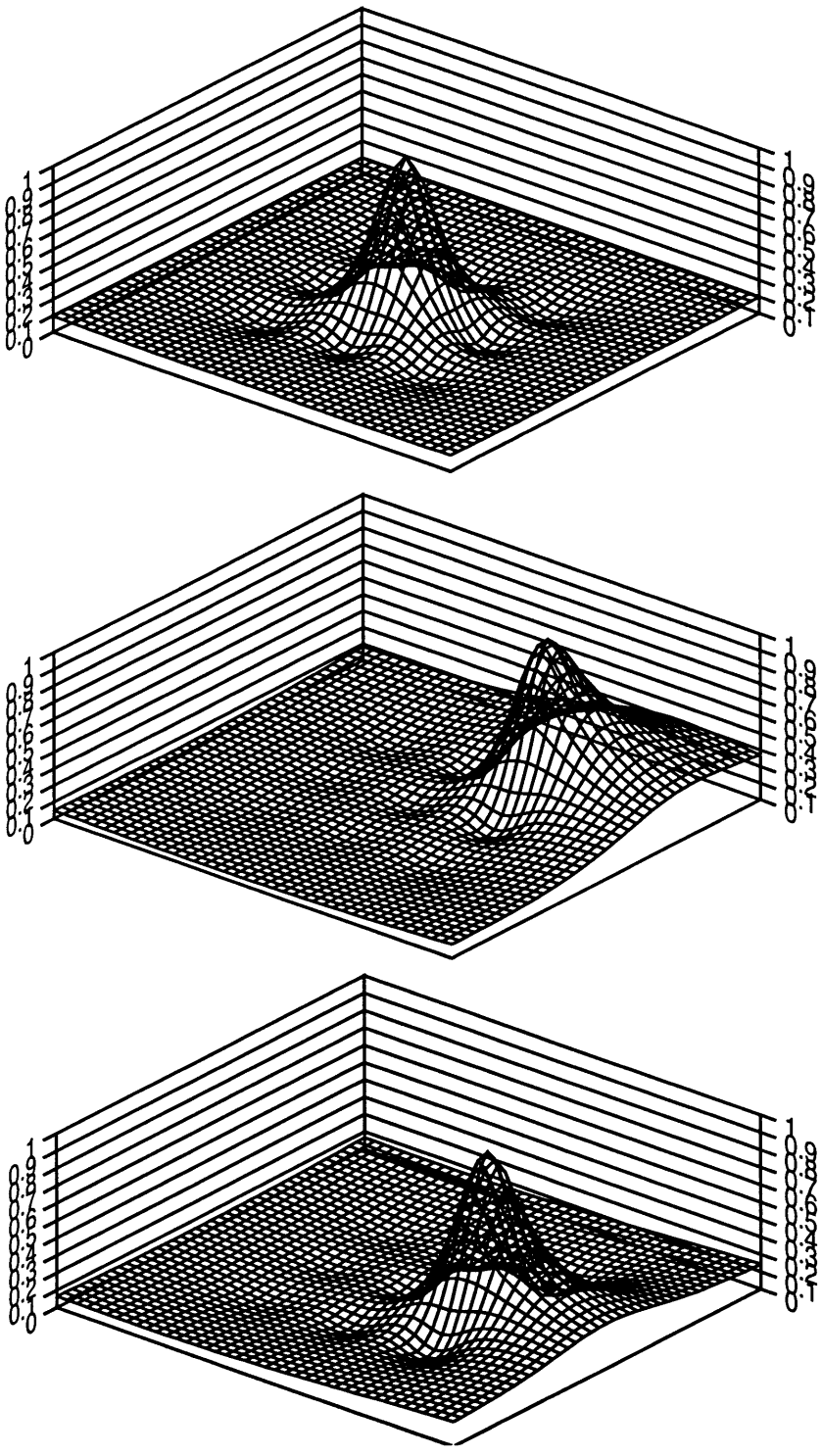


Figure 2.3. Normalized Influences for 3x3 Array of Identical Inverse Square Units

For simplicity, now consider 1-input, 1-output networks of units whose influence functions have identical weighting matrices $\mathbf{V} = v\mathbf{I} = \sigma^2\mathbf{I}$, where σ is unit width. Figure 2.4 shows the normalized influences for Gaussian and inverse square networks, respectively, whose units are centered at -4, 0, and 4, with σ = 1. Similar networks with σ = 3 appear in Figure 2.5. A number of important features are illustrated in these figures.

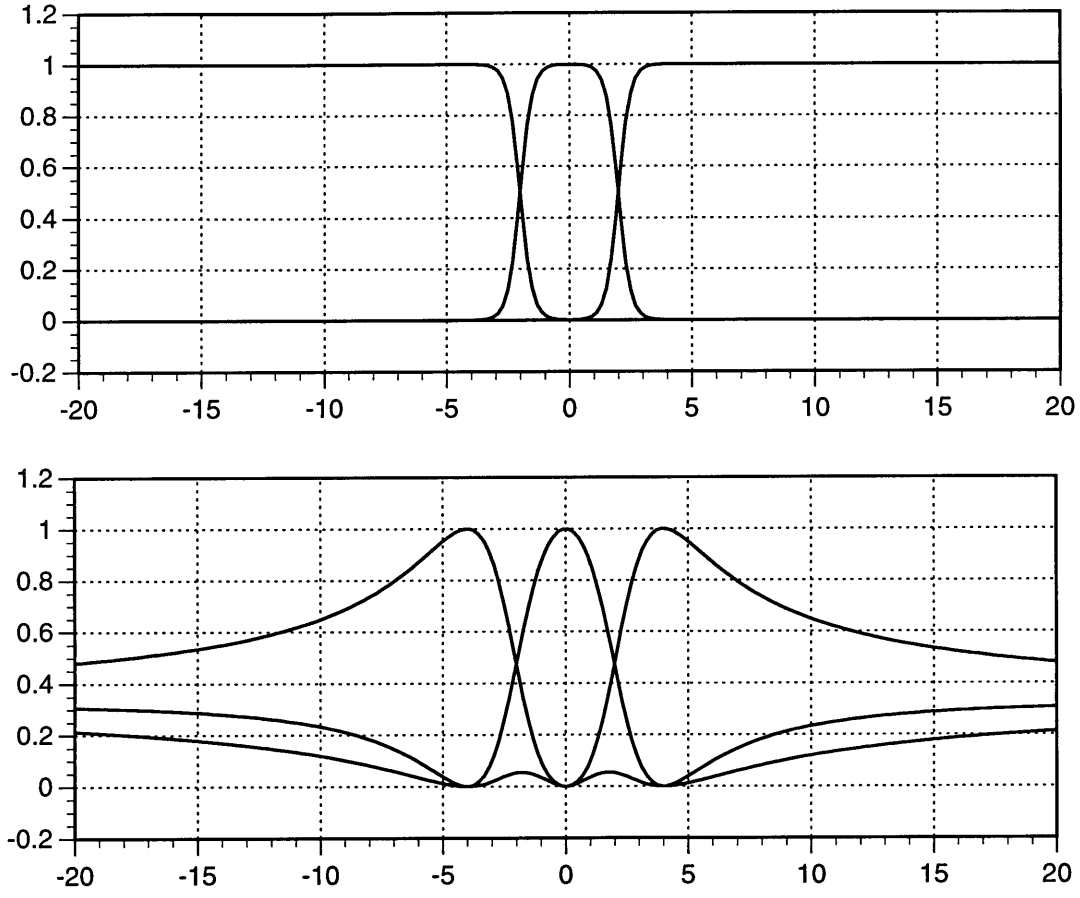


Figure 2.4. Normalized influences for Gaussian (top) and inverse square (bottom) B/I networks with three identical units at –4, 0, and 4. Width $\sigma = 1$.

Increasing the width from 1 to 3 creates more overlap of influence between neighboring units in the Gaussian network. This means that units exert influence more uniformly over the space relative to one another, so

that normalized influences are flatter and smoother. A similar effect can be achieved by moving the unit centers closer together to increase the overlap. Either way, it is clear that the quality of the combination of local approximations depends on the widths of influence functions relative to the proximity of neighboring units. The smaller the overlap between adjacent units, the more abrupt the transition from one local approximation to the next. Figure 2.6 shows influences of units with centers at –7, 1, 5, 7, and 8, where σ = 3 for all units. Scanning from left to right, as the overlap between adjacent units increases, normalized influences flatten and spread out for the Gaussian network.

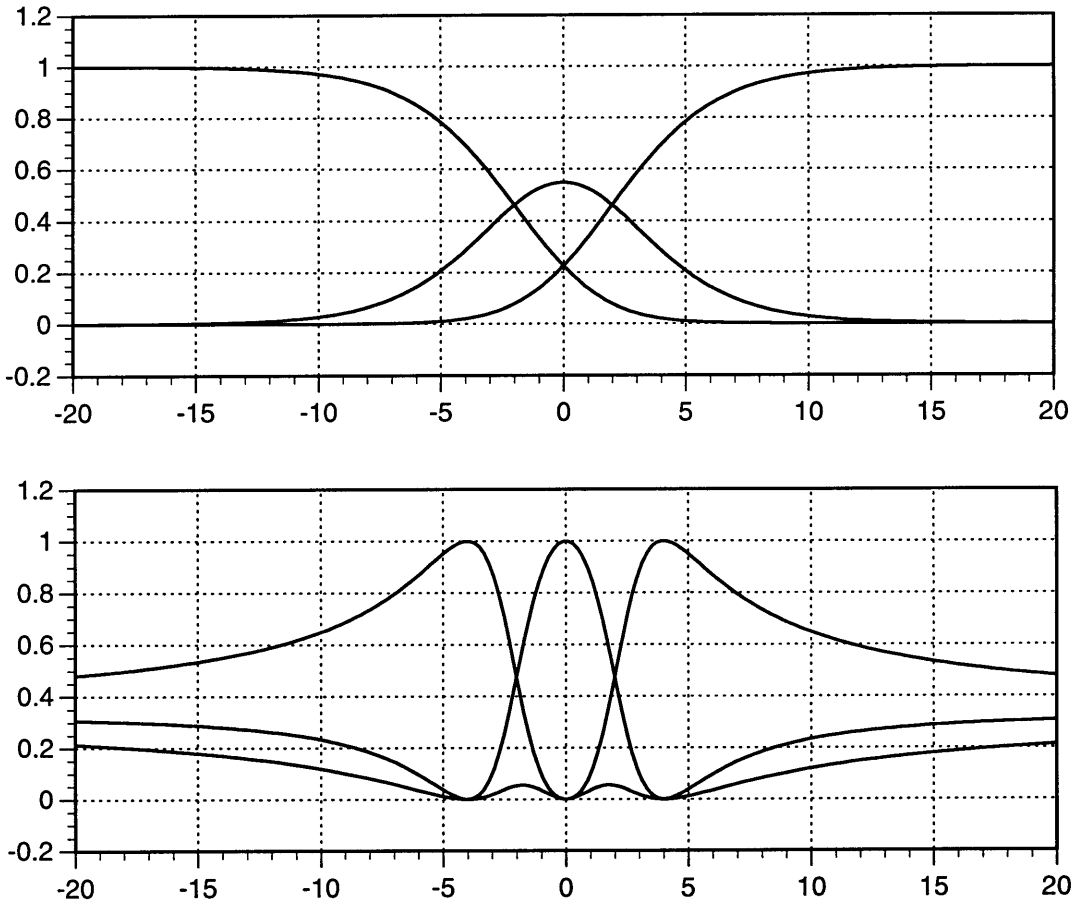


Figure 2.5. Normalized influences for Gaussian (top) and inverse square (bottom) B/I networks with three identical units at –4, 0, and 4. Width σ = 3.

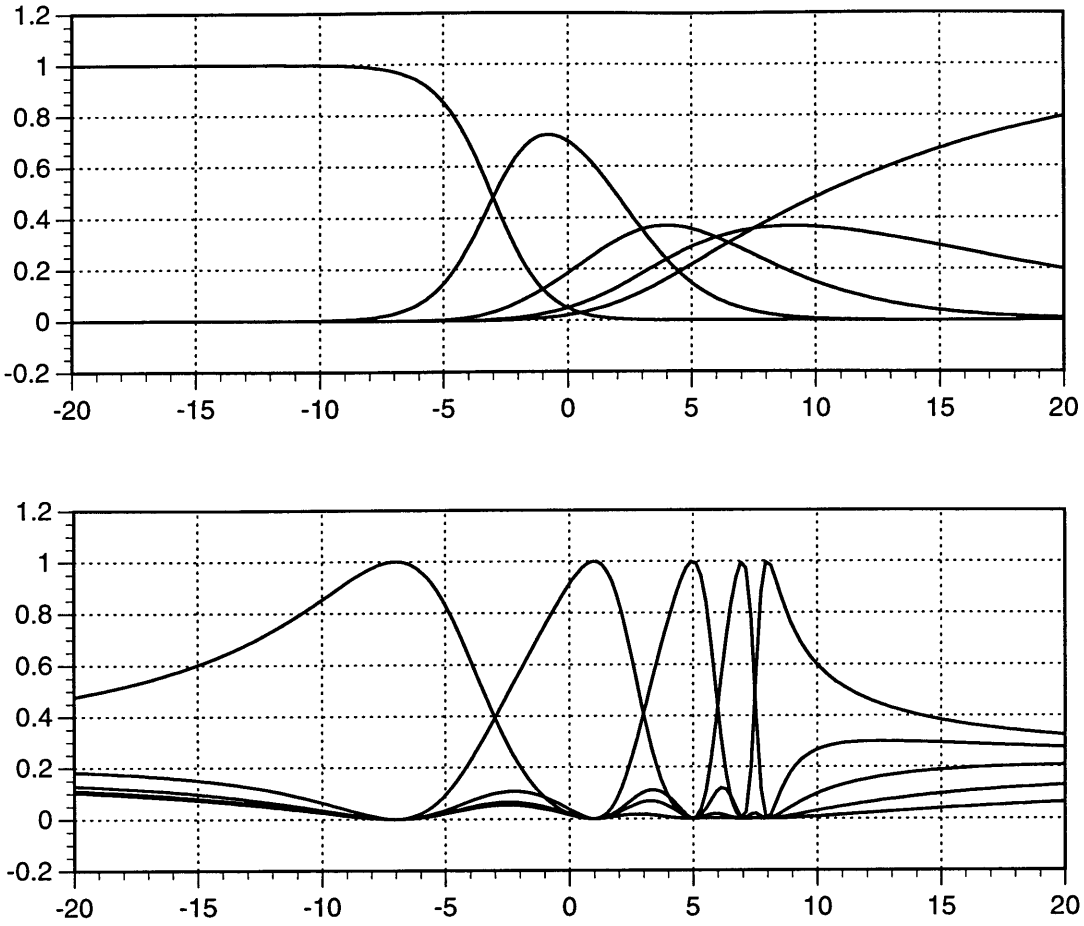


Figure 2.6. Normalized influences for Gaussian (top) and inverse square (bottom) B/I networks with five identical units at –7, 1, 5, 7, and 8. Width $\sigma = 3$.

Transitions between inverse square units, however, depend on the fact that the normalized influence of a unit is fixed at 1 at its center and 0 at other centers, and its first derivative is fixed at 0 at all centers [Appendix A]. By satisfying these conditions, the normalized influence functions adjust themselves to yield smooth transitions between units. In Figure 2.6, the influences transition nicely between units despite the fact that $\sigma = 3$ for them all and despite their uneven spacing. However, notice the small "humps" between distant units in the preceding figures: there is no constraint on the influence to fall to 0 and remain there. As a result, inverse square

normalized influences are somewhat less localized and less smooth overall than those generated by Gaussian units, and may take on significant values at points between units at whose centers their value is 0—i.e., "humps" and "ridges" may exist between/among unit centers.

Inverse square influences are not affected by uniform scale changes in the widths of all units—normalization cancels them out (cf. Figures 2.4 and 2.5). However, changes in the widths of individual units (for both inverse square and Gaussian networks) makes a difference in the relative rate of transition between local approximations; i.e., increasing the width of a single unit expands the region in which its basis function dominates with respect to neighboring basis functions.

At points far away from all unit centers, the Gaussian and inverse square influences behave quite differently. Assuming all units have the same width, normalized influence(s) of the nearest border unit(s) will dominate in the case of Gaussian networks; for inverse square influences, all units will tend to share equally. These effects become more pronounced as distance increases. The preceding figures clearly demonstrate this point.

The weighting matrices of individual units can be allowed to vary independently of one another. This extra representational power makes it possible to achieve better results from either type of network, but may also create undesirable features in the network.

The problem of broader, flatter normalized Gaussian influences (Fig. 2.6) could be remedied by using narrower widths for more closely positioned units, resulting in more uniform heights from the different normalized influence functions.

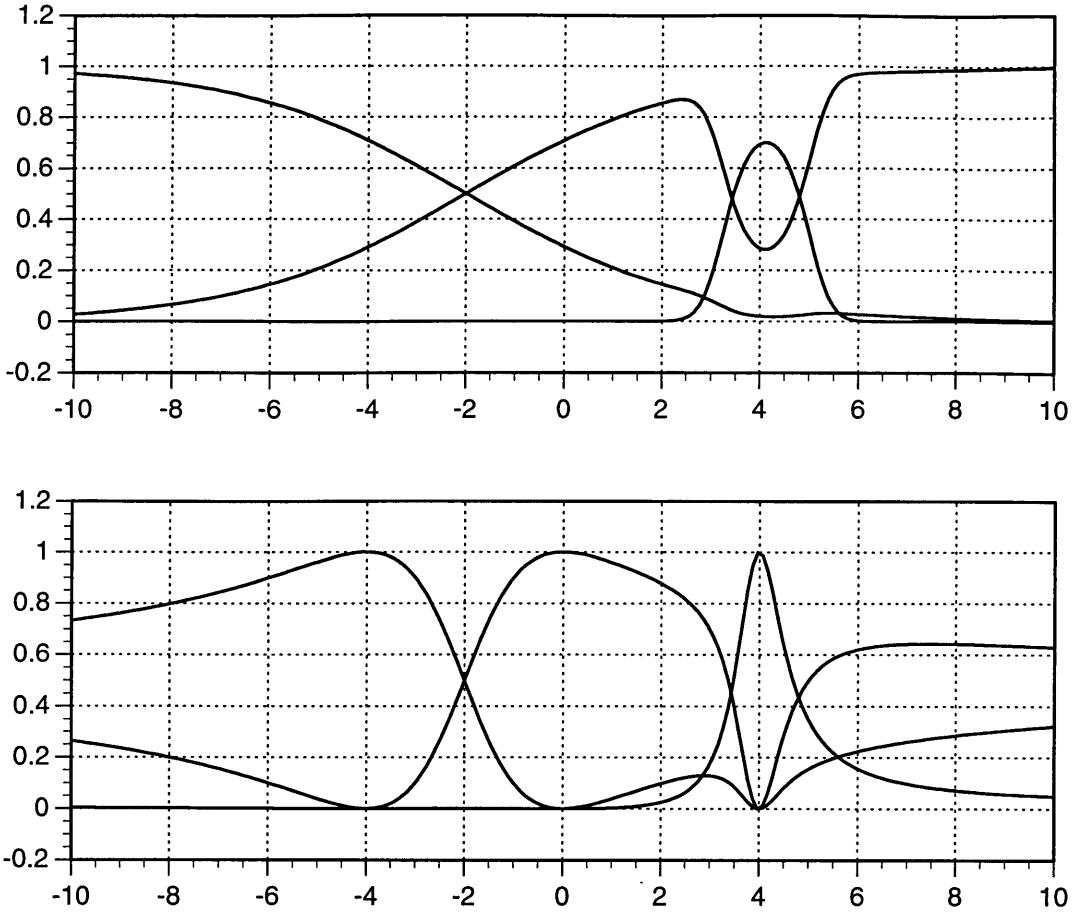


Figure 2.7. Normalized influences for Gaussian (top) and inverse square (bottom) B/I networks with three units at –4, 0, and 4, with widths σ of 3, 3, and 0.25.

However, this disparity of width could also lead to a loss of localization, as illustrated in Figure 2.7, where a "narrow" unit dominates in a small region inside the larger region dominated by a "broader" unit. Similar problems can occur with inverse square networks: in Figure 2.7, the largest normalized influence to the far right belongs not to the nearest unit, but rather to the middle unit, due to the relatively small width of the rightmost unit compared to that of the middle unit.

For distant inputs, Gaussian units with large widths tend to dominate, due to the slow rate of decay of their influence functions. Inverse square units will share in proportion to their variance values.

Ultimately, the choice of influence function depends on what is expected from the network. Normalized Gaussian influence functions tend to be smoother than inverse square influences and thus will create smoother combinations of local approximations. Inverse square influences guarantee that each local approximation predominates in some region of the input space without the possibility of becoming overwhelmed by neighboring local approximations, but at the expense of smoothness. (This trade-off is reminiscent of that between error and generalization discussed in Chapter 1.) One must also weigh the benefits of extra parameters (e.g., fully adjustable weighting matrices for each independent unit) with the responsibilities of training them. To take full advantage of Gaussian units, individual width values must be set to yield good interpolation between arbitrarily positioned units, whereas inverse square units seem to be more robust to these values, and may be able to perform adequately with units of equal width.

A note on normalization: It has been demonstrated how localization may not strictly hold after localized influences have been normalized. However, the proximity of an input to different unit centers remains the chief criterion for computing the network output from the set of basis functions; in this sense, each basis function remains localized for purposes of interpolation. As for extrapolation, it is unclear how to proceed as the distance from all units becomes very large: whether to fall to zero (RBFNs); to combine weighted local approximations (normalized inverse square influences); or to neglect some basis functions in favor of others (normalized Gaussian influences). In any case, this is a secondary issue, since the validity

of the network approximation should be expected to decline outside the region of space populated by units, and under no circumstances should extrapolation be relied upon to produce accurate mappings. Proper utilization of spatially localized networks implies that unit coverage should extend throughout the entire region of space for which a mapping is being constructed.

As a method of knitting together local approximations, normalization appears to perform better than simple superposition (as done with RBFNs). In Figures 2.8a, 2.8b, and 2.8c, a planar surface is approximated using a RBFN, a B/I network with constant basis functions and Gaussian influence functions, and a similar B/I network using inverse square influence functions. The local approximations were simply the values of the target function at the unit centers. Most notable is the difference in smoothness between the RBF and B/I networks, the latter of which yield a much more "planar" surface. The variance of the Gaussian influence functions was chosen so that the transitions between local approximations are rather abrupt, making the local approximations themselves distinguishable. Note that the inverse square network yields a slightly more gradual interpolation.

One point of interest: if the basis functions are upgraded to be affine functions[1], the local approximations can be set to match the target function exactly; subsequent interpolation would yield an exact network approximation. In fact, only one unit is necessary to represent the linear target function. If the target function were of higher order—say, a quadratic

---

[1] A function is said to be *affine* if its value for some input $\mathbf{x}$ minus its value for an input of zero varies linearly with $\mathbf{x}$:

$$\mathbf{f}(\mathbf{x}) = \mathbf{f}(\mathbf{0}) + \mathbf{A}\mathbf{x}, \ \mathbf{A} \text{ is linear} \tag{2.11}$$

function—each such basis function, properly initialized[2], could be regarded as a first-order Taylor approximation of the target function about the center of that unit.

2.2.3 Training the Parameters

Networks using the B/I architecture have performed well in a number of practical applications involving function approximation for control and estimation of nonlinear dynamical systems [12, 16, 19, 20, 24]. But just as important as architecture selection is the choice of a learning algorithm with which to adjust the network parameters to fit the observed training data. Chapter 3 discusses relevant issues and presents an algorithm for on-line training of basis/influence function networks.

[2]Offset value set equal to the value of the target function at that unit center; gradient vector set equal to gradient vector of target function at that unit center.

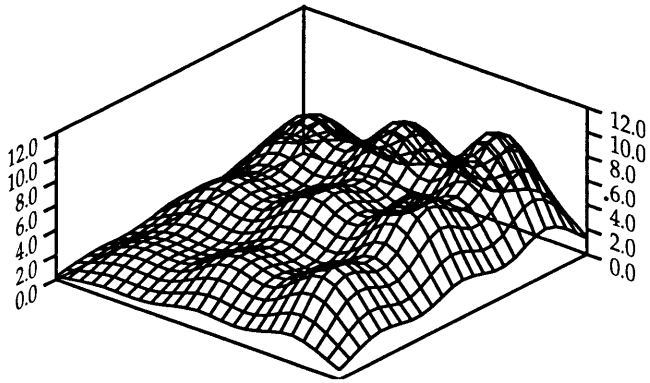


Figure 2.8a: Gaussian RBF

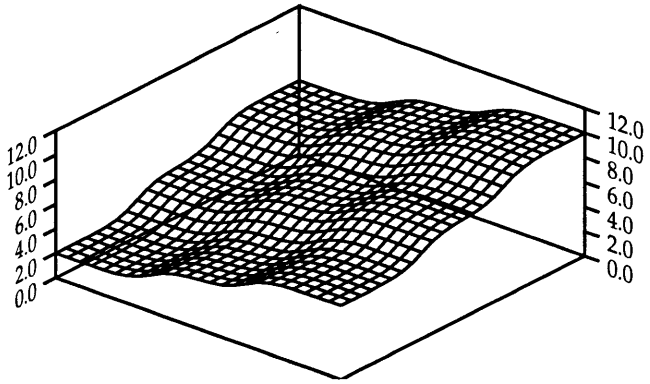


Figure 2.8b: Constant/Gaussian B/I

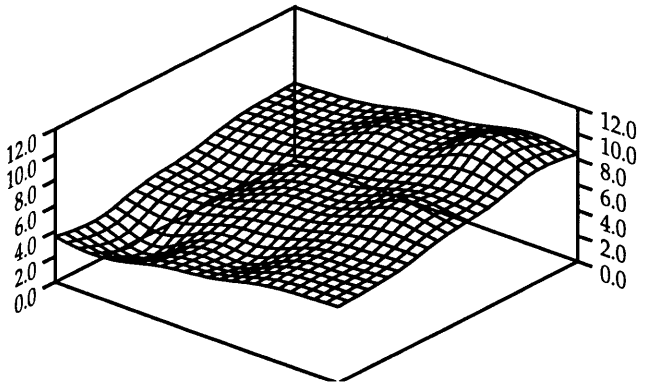


Figure 2.8c: Constant/Inverse Square B/I

Figure 2.8: Representing a plane

3 The Learning Algorithm

Given an adequate approximation structure, an algorithm is required that is able to adjust the values of the free parameters to yield a mapping that closely fits the training data and generalizes well when presented with unfamiliar inputs. The problem presented here is to perform this task on-line, in real-time, from noisy, arbitrarily distributed examples appearing in an order constrained only by the dynamics of the system that generates them.

In general, the quality of the learned mapping is calculated with respect to a cost function, or objective function, typically consisting of some measure of the approximation error over the domain of interest, perhaps with additional constraints that should be met. The goal of the learning algorithm is to minimize this objective function by adjusting the network parameter values.

Learning from a stream, rather than a batch, of training examples is necessarily incremental—minimization of the objective function must be done piecemeal from available training data, rather than minimizing over the entire set of examples all at once. The best that can be done in this case is to adjust the parameter values so that the cost associated with the current example (or set of examples, if a hybrid batch/incremental method is employed) is minimized. Such adjustments for different examples may actually work counter to each other, so that adjustments for later examples may negate adjustments made for previous examples. Hence, if training is not performed correctly, the parameter values may not converge and little learning will occur.

Gradient learning methods, a favorite in the literature and relatively well-understood, will be introduced in the next section as a starting point for further enhancements to this basic approach, which will be outlined in later sections. Finally, the algorithm developed and evaluated in this thesis will be presented.

3.1 Gradient Learning Algorithms

Given a network (parameterized by a vector **p**), a target function, and an objective function J that quantifies how well the network approximates the target function based on available data, a gradient descent algorithm seeks to minimize J by making successive adjustments to **p** in response to training data, such that J decreases after every adjustment [7, 31]. If the objective function is visualized as a surface over **p**-space, the value of J can be seen to move "downhill" along the surface after each adjustment, or along the negative gradient at the current point. Hopefully in this way J eventually reaches its global minimum, as a result of **p** attaining its optimum value, yielding the "best" network.

The next subsection discusses selection of the objective function. Later subsections will cover different approaches to gradient descent and potential problems that must be addressed.

3.1.1 The Objective Function: Batch vs. Incremental

The objective function (or cost function, as it is also known) serves as an indicator of network performance by measuring the extent to which the goals of learning are being satisfied. Typically, some norm of the approximation error serves as the objective function, although additional measures reflecting the desirability of other features (e.g., smoothness of the

mapping, configuration of the units relative to the training data) are often factored in.

Here are some examples of possible error norms:

infinity norm: $$J_y = \sup_{\mathbf{x}} \left| y_{targ}(\mathbf{x}) - y_{net}(\mathbf{x}) \right| \tag{3.1a}$$

2-norm (squared): $$J_y = \int_{\mathbf{X}} \left| y_{targ}(\mathbf{x}) - y_{net}(\mathbf{x}) \right|^2 d\mathbf{x} \tag{3.1b}$$

expected squared error: $$J_y = \int_{\mathbf{X}} p_{\mathbf{x}}(\mathbf{x}_0) \cdot \left| y_{targ}(\mathbf{x}_0) - y_{net}(\mathbf{x}_0) \right|^2 d\mathbf{x}_0 \tag{3.1c}$$

Selection of an error norm depends on the goals of learning. For example, if the trained network is subsequently used in an application in which a large error in response to any single input could have disastrous consequences, then the infinity norm is appropriate. In other scenarios, only average error may be important, allowing the use of one of the other norms.

In order for training to reduce this cost directly, it would be necessary to compute the cost before each parameter adjustment. This is not always possible, due to the unavailability of the target mapping over the entire learning domain at each adjustment, or practical, in light of the huge computational effort entailed.

An alternative to directly minimizing the true cost is to minimize the contribution of the current (*k*th) example to the cost. In effect, the global objective function is traded for an incremental objective function. Instead of the equations above, we would have, respectively:

absolute error: $$J_y(\mathbf{x}^k) = \left| y_{targ}(\mathbf{x}^k) - y_{net}(\mathbf{x}^k) \right| \tag{3.2a}$$

squared error: $$J_y(\mathbf{x}^k) = \left[y_{targ}(\mathbf{x}^k) - y_{net}(\mathbf{x}^k) \right]^2 \tag{3.2b}$$

squared error: $$J_y(\mathbf{x}^k) = \left[y_{targ}(\mathbf{x}^k) - y_{net}(\mathbf{x}^k) \right]^2 \tag{3.2c}$$

A significant drawback with the incremental approach is that parameter updates for individual examples are now decoupled and may work counter to each other. Training on some examples can actually undo previous training for others. This means that the quality of the learning can depend significantly on the ordering of examples in the stream of training data.

Note that incremental training tends to implicitly minimize the expected error of the network over the distribution of training examples, unless steps are taken to alleviate nonuniformities in that distribution.

For certain linear network problems, it can be shown that incremental supervised learning methods yield convergence to the global optimum solution [27]; however, there is no guarantee of convergence for most problems.

3.1.2 The Objective Function Topology

The mapping of the objective function over parameter space may contain hills, valleys, crests, troughs, and saddle points; there may be many local minima in addition to a global minimum that we wish to locate. The manner in which the parameters are adjusted "downhill" can result in convergence that is either slow, oscillatory, or "just right"; parameter adjustments that are too large often result in divergent behavior.

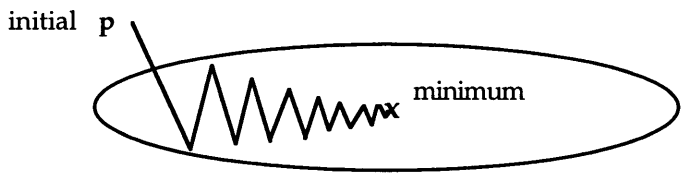


Figure 3.1. Oscillatory convergence of gradient descent to local minimum

Consider a simple update rule that moves the parameter vector **p** by a negative fraction of its gradient at each step. Assume that the initial value of **p** lies near the end of an elongated trough (Figure 3.1).

Adjustments move **p** directly downhill at each step. Due to the elongated shape of the trough, **p** is adjusted more across the trough than down it, so that rather than approaching the minimum point (center "**x**") head on, **p** experiences a fast oscillation across the width of the trough plus slow travel down its length towards the minimum.

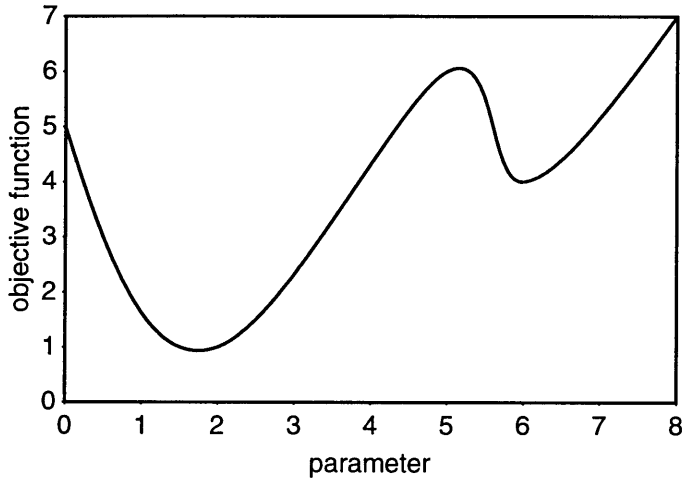


Figure 3.2. Gradient learning can be sensitive to initial conditions. In this example, initial parameter values greater than 5.0 result in convergence to the rightmost minimum, a suboptimal solution.

This example demonstrates the dependence of the final solution on the landscape of the objective function and the initial value of **p**. This local nature of gradient methods makes it impossible to place any guarantees on the quality of the final solution: the algorithm is prone to converge to whatever minimum lies nearby, whether deep and nearly optimal, or

shallow and highly suboptimal (Figure 3.2). This presents practical problems for learning algorithms that must be addressed.

3.1.3 Variations on Gradient Descent

An excellent discussion of gradient learning methods may be found in Chapter 6 of [7]. A brief summary of different methods is presented here.

The simplest way to move down the gradient is to take steps of a constant, predetermined *step size* λ_p in the direction opposite the gradient:

$$\Delta \mathbf{p} = -\lambda_p \cdot \frac{\nabla_p J}{\left\| \nabla_p J \right\|} \tag{3.3}$$

This may eventually find a minimum, if it exists, but will limit cycle about the minimum unless landing exactly on it, since the size of the update is zero only when the gradient is zero.

Another method in common practice instead adjusts $\mathbf{p}$ an amount *proportional to the gradient*:

$$\Delta \mathbf{p} = -\lambda_p \cdot \nabla_p J \tag{3.4}$$

The constant coefficient λ_p is often called the *learning rate* in learning applications. In contrast to the constant step size method, this algorithm is able to converge completely, since the updates shrink in size as the gradient magnitude approaches zero. However, this convergence is not guaranteed.

Better performance can be obtained if the learning rate, rather than being constant, is linked to the *curvature* of the objective function. For small enough adjustments, each step must result in a reduction of the objective function; however, since the objective function can be curved, any adjustment may overshoot and actually increase the objective function for that training step. In fact, subsequent adjustments may likewise overshoot,

causing oscillations in the parameter values and slower convergence. Reducing the learning rate may eliminate oscillations, but convergence time may then be significantly increased.

One alternative is to make the learning rate adaptive by sensing overshoot. If the objective function consistently decreases (say, for the past n steps) in response to an update, then the learning rate is increased, usually linearly; if the objective function increases, the learning rate is decreased, usually exponentially (to prevent learning rate blow-up and subsequent divergent behavior); otherwise, it is unchanged:

$$\Delta\lambda_{\mathbf{p}} = \begin{cases} +a & \text{if } \Delta J < 0 \text{ consistently} \\ -b\lambda_{\mathbf{p}} & \text{if } \Delta J > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.5}$$

Decreasing the learning rate serves to damp out oscillations that accompany persistent overshoot; increasing it should speed convergence when overshoot is not a problem. (Consult [4, 10] for discussions of similar techniques.)

Newton's method is derived from a second-order Taylor series approximation of the objective function with respect to $\mathbf{p}$, which results in this update rule for $\mathbf{p}$:

$$\Delta\mathbf{p} = -\mathbf{H}^{-1} \cdot \nabla_{\mathbf{p}} J \tag{3.6}$$

The scalar learning rate $\lambda_{\mathbf{p}}$ is replaced by the inverse of the square Hessian matrix $\mathbf{H}$, the second derivative of the objective function J with respect to $\mathbf{p}$. This method accounts for the curvature of the objective function in all directions, yielding faster learning rates where curvature is small, and vice versa. Unfortunately, a matrix inversion is required at each step, making this approach computationally expensive and impractical in most cases. One

variation, the pseudo-Newton method, works from approximations of the Hessian (e.g., neglecting off-diagonal elements).

The *steepest descent* method, rather than relying on some *a priori* notion of step size, actually calculates the distance that must be traveled along the gradient direction to achieve the smallest value for J. The related method of *conjugate gradient descent* helps prevent a zig-zag approach to the minimum by calculating the direction of descent as a combination of the gradient and the previous direction followed.

Each method outlined above has advantages and disadvantages. The choice will depend on the problem at hand.

The different update methods presented so far are concerned mainly with speeding convergence to a minimum, any minimum; they are still liable to converge to a local rather than the global minimum. Ironically, though, imperfections in these methods may make the algorithm less prone to converge to relatively shallow local minima—for example, an overshoot of a local minimum may allow the algorithm to escape that region of parameter space. In general, deviating from the exact gradient direction may make shallow local minima avoidable, as may noise in the training data.

The technique of *momentum* is a relatively straightforward way both to avoid local minima and to dampen oscillations in subsequent updates. Rather than adjusting $\mathbf{p}$ along the update calculated, a combination of the calculated update and the previous update is used:

$$\Delta\mathbf{p}^{k+1} = -\lambda_{\mathbf{p}} \cdot (1-\mu) \cdot \nabla_{\mathbf{p}} J^k + \mu \cdot \Delta\mathbf{p}^k \qquad 0 \le \mu < 1 \tag{3.7}$$

This is simply a discrete-time first-order filter applied to the gradient updates. Oscillations will tend to cancel, and persistent updates in the same direction

be preserved. In addition, as **p** reaches the bottom of a shallow basin, momentum may at first cause an overshoot, providing an opportunity for escape from shallow local minima. However, momentum may also slow convergence by enabling escape from *global* minima.

3.1.4 Comments

The previous discussion of various gradient algorithms has assumed that the objective function remains the same for each training step. This is in fact true only for batch learning; during incremental learning, the objective function changes with each new training example. This tends to drive **p** towards an average value that roughly minimizes the objective functions of all examples; it may also aid convergence to the global minimum by creating training "noise" due to the changing objective function. However, if the learning rate is too high, and instantaneous objective functions too dissimilar, the value of **p** will tend not to converge to the batch solution, but rather to adapt itself to the most recent examples presented—the network will forget what it has learned about previous examples.

It has been tacitly assumed that all adjustable parameters affect the network mapping in essentially the same way, and should therefore be viewed as equivalent for training purposes. This assumption is often false and precludes exploiting parameter differences to achieve better training. In the case of B/I networks, influence function parameters affect the output differently than do basis function parameters; potential improvements in learning may be afforded by an algorithm that exploits these differences.

Gradient descent is purely local—it incorporates absolutely no global information, and is prone to converge to locally optimal solutions. Although the primary objective of the learning algorithm is to minimize the error norm, perhaps the objective function can be augmented to encourage the

network to organize itself such that the globally optimal solution is more likely found. For instance, a secondary objective of the algorithm can be to arrange the network units to most efficiently represent the training data, leading eventually to a more accurate approximation.

The next section explores the idea of hybrid learning algorithms that employ both supervised and unsupervised learning techniques to achieve the objectives of low error and optimal unit arrangement, respectively.

3.2 An Introduction to Hybrid Algorithms

The operating philosophy for the remainder of this thesis is that a well-trained network can be more readily obtained if the objective function that guides training incorporates more than just error criteria.

The ability of a B/I network to approximate some function varies with the center locations and widths of the network units. A learning algorithm derived from a purely error-based objective function (say, a pure error gradient algorithm) can adjust centers and widths, and may even result in unit configurations that resemble *a priori* common sense notions of the "correct" final network layout. But why not address these ideas of proper layout and overlap of units directly, rather than hoping that things come out in the wash?

The following discussion presents methods of building additional constraints into the objective function that directly encourage unit positioning and overlap that best match the organization of the training data. Section 3.2.1 presents relevant work by Moody and Darken [20] involving unsupervised training of unit centers; the following section introduces a novel yet analogous method of training unit widths; and finally, the implementation of weight decay as a method of promoting smooth network mappings is discussed.

3.2.1 Center Training via Clustering

Section 1.2.1 introduced the idea of unsupervised learning and presented the learning of input cluster centers as an example. It is exactly this approach that has been adopted by Moody and Darken [20] to train network unit centers, in parallel with gradient training of basis functions. Their approach is summarized below.

The algorithm seeks to maximize coverage of the training inputs by a fixed number of network units. The rationale is that to get the best performance from the network, the unit centers should be placed such that the cumulative squared distance from all training inputs $\mathbf{x}^k$ to their nearest centers $\mathbf{x}_{win}^{c,k}$ is minimized:

$$\text{global cost:} \qquad J_{\mathbf{x}} = \sum_k \tfrac{1}{2} \left\| \mathbf{x}^k - \mathbf{x}_{win}^{c,k} \right\|^2 \tag{3.8a}$$

This equation is incrementalized and used as the objective function of an unsupervised center training algorithm:

$$\text{incremental cost:} \qquad J_{\mathbf{x}} = \tfrac{1}{2} \left\| \mathbf{x} - \mathbf{x}_{win}^{c} \right\|^2 \tag{3.8b}$$

$$\text{incremental update:} \qquad \Delta \mathbf{x}_{win}^{c} = \lambda_c \cdot \left(\mathbf{x} - \mathbf{x}_{win}^{c} \right) \qquad 0 < \lambda_c < 1 \qquad \lambda_c \text{ constant} \tag{3.8c}$$

For each example, a "winner" unit is selected, i.e., that unit whose center $\mathbf{x}_{win}^{c}$ lies nearest the example—only the center of that unit is updated on that example. This algorithm is known as *incremental k-means clustering*, and for small enough learning rate is known to cause the set of centers to converge stochastically to a corresponding set of local "centroids", i.e., each center is placed at the average position of the set of inputs for which

that center is the nearest [13]. Moody and Darken train the networks by cycling through a predetermined batch of examples, rather than using a stream of indefinite length determined on-line. (Note that this is essentially a gradient algorithm that may not yield the globally optimal solution.)

Center locations are initialized with random values within the training domain. Training then proceeds in two phases: centers are trained, and once they have converged, widths are set heuristically to achieve a desirable overlap between neighboring units; basis functions (constant functions) are trained via gradient descent to satisfy a squared error objective.

Moody and Darken report faster convergence with this hybrid algorithm than with a purely supervised algorithm. The improved performance is attributed to the linearity of the unsupervised center update rule, contrasted with the nonlinearity of the supervised rule.

3.2.2 Unsupervised Width Training

The clustering approach to center training, which seeks to locate unit centers near the means of the local sets of inputs, suggests an analogous method for training unit widths: that is, setting unit widths to approximate the standard deviations of the local sets of inputs.

$$\text{global cost:} \qquad J_\sigma = \sum_k \tfrac{1}{2} \left\| r^{2,k} - \sigma_{win}^{2,k} \right\|^2 \tag{3.9a}$$

$$\text{incremental cost:} \qquad J_\sigma = \tfrac{1}{2} \left\| r^2 - \sigma_{win}^2 \right\|^2 \tag{3.9b}$$

$$\text{incremental update:} \qquad \Delta\sigma_{win}^2 = \lambda_c \cdot \left(r^2 - \sigma_{win}^2 \right) \tag{3.9c}$$

Such a method, if successful, would enable the widths of the units to shrink or grow according to the distances between neighboring units, thus accomplishing implicitly what heuristic methods must do explicitly. In

addition, this type of training should be much faster than a supervised approach by virtue of its linearity. It may also be possible to train an entire weighting matrix to approximate the sample covariance of the local inputs, allowing for the creation of influence functions with more complex shapes:

$$\Delta \mathbf{V}_{win} = \lambda_c \cdot \left[\left(\mathbf{x} - \mathbf{x}^c_{win} \right) \cdot \left(\mathbf{x} - \mathbf{x}^c_{win} \right)^{\mathrm{T}} - \mathbf{V}_{win} \right] \tag{3.10}$$

3.2.3 Smoothness Constraints

As discussed in Chapter 1, the ability of a network to generalize to new examples from those experienced during training is related to the extent to which the mapping varies at points away from the training data. Generalization tends to improve as the smoothness of the mapping increases (assuming that the target mapping is smooth).

The desire for maximum smoothness can be expressed in the objective function as a norm of the second derivative (e.g., the Hessian matrix) of the network mapping [14]; for example,

$$J_s = \int_{\mathbf{x}} \left\| \frac{\partial^2 y}{\partial \mathbf{x} \cdot \partial \mathbf{x}^{\mathrm{T}}} \right\|^2 d\mathbf{x} = \int_{\mathbf{x}} \|\mathbf{H}\|^2 d\mathbf{x} \tag{3.11}$$

Minimizing this norm will smooth the mapping. But clearly the analytical evaluation of such a norm is intractable, and even a close numerical approximation would be computationally impractical.

Alternatively, one could compare the linear basis functions of neighboring units and adjust them so that the interpolation from one to the other is more gradual. This approach requires that neighboring units be identified and compared; unless the current set of relevant units is somehow pared down, the computational overhead increases combinatorially with the number of units.

In contrast to creating smoothness constraints based on second derivative information, a more indirect—and computationally cheaper—method increases smoothness by encouraging smaller parameter values. The objective function is augmented with a norm of some subset of the parameter values themselves, resulting in the process of *weight decay* :

$$J_p = \tfrac{1}{2}\mathbf{p}^T \cdot \mathbf{p} \tag{3.12a}$$

$$\Delta\mathbf{p} = -\lambda_p \cdot \mathbf{p} \tag{3.12b}$$

At each training step, these parameter values are decreased slightly; thus, the final mapping may be somewhat higher in error due to this interference, but the smaller parameters should yield a smoother mapping. Ji and Psaltis [11] train a feedforward sigmoidal network using backpropagation and weight decay on all parameters, with favorable results. For implementations with B/I networks, weight decay would make sense only for basis function "slopes" and inverse weighting matrices, since offset and center values do not directly affect the smoothness of the network mapping.

3.3 Learning Rules

The parameter training algorithm developed and evaluated in this thesis is similar to the hybrid algorithm of Moody and Darken, with a few novel enhancements:

- *Adaptive learning rates for influence function parameters*. Learning rates are increased/decreased depending on the error associated with individual examples, and on the relative density of training examples in different regions of the domain, in order to speed convergence of units towards erroneous regions, and to make coverage less sensitive to the distribution of examples.

- *Unsupervised width training*. Rather than setting unit widths heuristically to achieve the desired overlap, widths are adjusted to approximate the size of the area in which corresponding units predominate.
- *Simultaneous training of both basis function and influence function parameters*. All parameters are trained in parallel; however, adjustments to influence parameters are more gradual than to basis function parameters.
- *Heuristic learning rates*. Learning rates are calculated based on an estimate of the number of examples that must appear in the training stream before an accurate picture of their distribution may be inferred.

The algorithm is applied to B/I networks having affine, rather than constant, basis functions [19, 30].

Training examples are assumed to appear as a stream generated by a free-running dynamical system, rather than selected randomly from a predetermined batch. Whereas the Moody and Darken approach presumes a fixed network structure, these modifications anticipate the implementation of adaptive structure (Chapter 4), where the number of units may change at any moment, requiring that the algorithm always be ready to incorporate and accommodate new units into the network arrangement.

3.3.1 Basis Function Training

Basis function parameters are trained using gradient descent to satisfy an incremental error objective. Recall the output equation for a B/I network with affine basis functions:

$$
\begin{aligned}
y_{net}(\mathbf{x}) &= \sum_j B_j\left(\mathbf{x}-\mathbf{x}_j^c\right)\cdot I_j\left(\left\|\mathbf{x}-\mathbf{x}_j^c\right\|\right) \\
&= \sum_j\left[\mathbf{w}_j^{\mathrm{T}}\cdot\left(\mathbf{x}-\mathbf{x}_j^c\right)+w_{0j}\right]\cdot I_j\left(\left\|\mathbf{x}-\mathbf{x}_j^c\right\|\right)
\end{aligned}
\tag{3.13}
$$

$\mathbf{x}$ = input vector
$\mathbf{x}_j^c$ = center of unit j
y_{net} = scalar output
B_j = basis function for unit j
$\mathbf{w}_j^{\mathrm{T}}, w_{0j}$ = basis function parameters for unit j
I_j = normalized influence function for unit j

The incremental cost is simply the squared error associated with the current example:

$$
J_y = \tfrac{1}{2}\left[y_{targ}-y_{net}\right]^2 \tag{3.14a}
$$

Calculating the gradients with respect to the basis function parameters yields the parameter update rules:

$$
\Delta\mathbf{w}_j = \lambda_b\cdot I_j\cdot\left(y_{targ}-y_{net}\right)\cdot\left(\mathbf{x}-\mathbf{x}_j^c\right) \tag{3.14b}
$$

$$
\Delta w_{0j} = \lambda_b\cdot I_j\cdot\left(y_{targ}-y_{net}\right) \tag{3.14c}
$$

$$
0<\lambda_b<1, \quad \lambda_b \text{ constant}
$$

With spatially localized networks, it is usually the case (by design) that only a few units contribute significantly to the output for any particular input. It is therefore not necessary to train every unit on every input. Some useful alternatives are to train only the k nearest neighbors of an input; to train the k most influential neighbors; or to train the set of most influential neighbors, the sum of whose normalized influences exceeds some threshold value less than 1 [19]. For this implementation, the *Heapsort* algorithm is used [26].

Heapsort is fast as sorting algorithms go, arranging a list of N elements into ascending order in $N\log_2 N$ time.

3.3.2 Influence Function Training

Centers are trained using clustering methods in essentially the same manner as Moody and Darken (section 3.2.1), with the exception that the nominal learning rate λ_c is scaled by two parameters, δ_I and δ_n (section 3.3.3), according to the error associated with the current input and the frequency (relative to other units) with which the current unit "wins", respectively.

$$J_{\mathbf{x}} = \tfrac{1}{2}\left\|\mathbf{x} - \mathbf{x}_{win}^{c}\right\|^2 \tag{3.15a}$$

$$\begin{gathered}\Delta\mathbf{x}_{win}^{c} = \lambda_c \cdot \delta_I \cdot \delta_n \cdot \left(\mathbf{x} - \mathbf{x}_{win}^{c}\right) \\ 0 \le \delta_I, \delta_n \le 1 \\ 0 < \lambda_c < 1 \\ \lambda_c \text{ constant}\end{gathered} \tag{3.15b}$$

Clustering (as described herein) is equivalent to setting the center of a particular unit equal to a first-order filtered version of the stream of inputs belonging to that unit:

$$\mathbf{x}_{win}^{c}[k+1] = \left(1 - \lambda_c \cdot \delta_I \cdot \delta_n\right) \cdot \mathbf{x}_{win}^{c}[k] + \lambda_c \cdot \delta_I \cdot \delta_n \cdot \mathbf{x}[k] \tag{3.16}$$

Varying the learning rate for different inputs amounts to weighting some inputs more heavily than others. Since this filtering scheme is stable (for $0 \le \lambda_c \cdot \delta_I \cdot \delta_n \le 1$), centers are guaranteed to lie within the bounds of the training inputs, a feature not guaranteed by a purely supervised, error-based center training algorithm.

Width training proceeds according to similar rules, as described in Section 3.2.2. The same nominal learning rate, λ_c, is used, and is likewise scaled:

$$J_\sigma = \tfrac{1}{2}\left\| r^2 - \sigma_{win}^2 \right\|^2 \tag{3.17a}$$

$$\begin{gathered} \Delta\sigma_{win}^2 = \lambda_c \cdot \delta_J \cdot \delta_n \cdot \left(r^2 - \sigma_{win}^2\right) \\ 0 \le \delta_J \cdot \delta_n \le 1 \\ 0 < \lambda_c < 1 \\ \lambda_c \text{ constant} \end{gathered} \tag{3.17b}$$

By departing from a purely supervised algorithm, where all learning rules are derived from the same error-based objective function, we have decoupled the training of basis function parameters from that of influence function parameters. This makes it less clear that the algorithm will eventually converge; but with prudent selection of learning rates, the likelihood of some foreseeable problems can be reduced.

3.3.3 Adaptive Learning Rates

3.3.3.1 Error weighting

The clustering algorithm of Moody and Darken places units in the regions most densely populated with training inputs. This is a good rationale for arranging the set of units, given no knowledge of how well the network is performing on those examples; but that knowledge is available (incrementally) and can be used to place the units where they are likely to be most able to reduce approximation error.

Some examples will be poorly represented and will generate large errors from the network. Others will generate small errors. It makes little sense to rearrange the network as much for small-error examples as for large-error examples—small errors warrant little or no rearrangement compared to large errors. Therefore, the magnitude of the error associated with each example can be used to determine the current learning rate for that example.

Learning rates for influence function training are scaled by this *error-weighting function* δ_J (Figure 3.3):

$$\delta_J = \frac{J_y}{J_y + J^c_{win}} \tag{3.18}$$

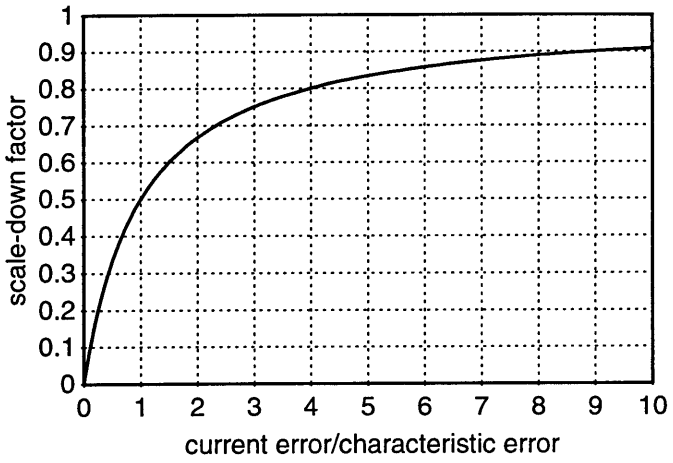


Figure 3.3. Training inputs that generate very large errors relative to the winner's characteristic error are learned at nearly the full rate; inputs with smaller errors are learned at a roughly proportionally scaled down rate.

The magnitude of the current error is judged relative to some characteristic error J^c_{win} for the current winner unit; relatively large errors receive the full learning rate, whereas errors comparable to or less than J^c_{win} are scaled down roughly linearly. The net effect is decreased learning effort applied to examples with relatively small errors.

But how large is large? For the error-weighting function to be able to discriminate meaningfully between large and small errors, the characteristic error should reflect either: i) the magnitude of "typical" error encountered recently by the current winner; or ii) the magnitude of the target error for this learning task. The latter is purely subjective on the part of the engineer directing the learning. For the former, "typical" error translates to filtered error:

$$\langle J_y \rangle_{win}[k+1] = \alpha \cdot \langle J_y \rangle_{win}[k] + (1-\alpha) \cdot J_y[k] \qquad (3.19)$$
$$0 \le \alpha \le 1$$

This equation calculates the average local error $\langle J_y \rangle_{win}$ for the winner unit. The filter constant, or age weight α , should be set small enough to allow the average local error to remain relatively current; if this average lags, convergence could be slowed.

The results appearing later in this thesis were generated using a constant value for the characteristic error, rather than average local error. However, we shall see in Chapter 4 that the calculation of average local error for all network units is an important step in the adaptive structure algorithm.

3.3.3.2 Training density weighting

Uneven distribution of training examples can bias center placement away from more sparsely populated areas. Consider the case where most inputs lie in one particular region (neglect error weighting for the moment). The centroids of all units will be shifted towards that region, at the expense of the approximation capability of the network in other regions.

Training can be made more evenly distributed by reducing the learning rates for examples that occur in over-represented regions. More precisely, learning rates are scaled down in inverse proportion to the amount of overtraining in that region.

Assuming that a "fair" training distribution would divide n_{tot} total examples equally among N_u units, learning rates are scaled down according to the number of examples n_{win} with which the current winner unit has been trained in excess of the "fair" amount. The learning rates of undertrained units are not scaled down (Figure 3.4).

$$\delta_n = \min\left(1.0, \frac{n_{tot}/N_u}{n_{win}}\right) \tag{3.20}$$

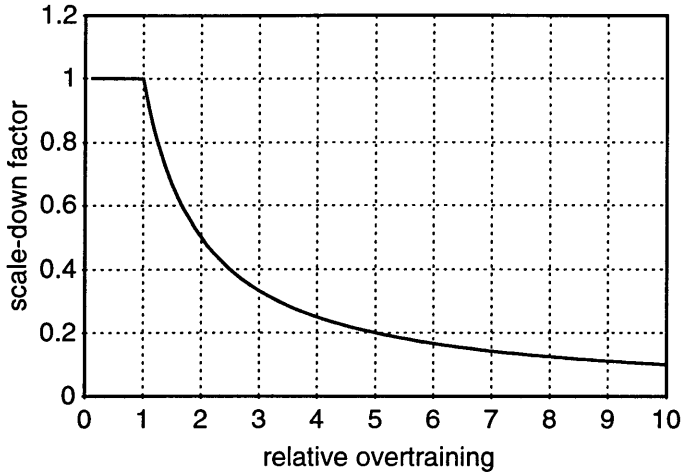


Figure 3.4. The learning rate for the center of a particular unit is scaled down in inverse proportion to the number of updates that unit has undergone relative to the "fair" number of updates, i.e., units sharing equally.

The motivation for adaptive scaling of learning rates is the improvement of the clustering algorithm for on-line learning, so it is appropriate to apply adaptive scaling only to the training of influence parameters, i.e., centers and widths.

3.3.3.3 Learning rate heuristics

Basis functions are trained to give the best (smooth) fit to the training data *given the current centers and widths of units.* If centers and widths are trained too quickly, otherwise well-trained basis functions will be dragged through input space more quickly than they can adjust, thereby increasing the error of the mapping in those regions.

In addition, error-weighting assumes that the basis function training has mostly converged, given the current configuration of units, so that the errors generated approximate the lowest errors that can be achieved with the

current arrangement. For these reasons, it is important that the learning rates for centers and widths be smaller than those for basis function parameters. (However, no matter how slowly the centers and widths are trained, some additional error is unavoidable.)

If some knowledge is available regarding the distribution and ordering of training inputs, an effective pseudo-batch size can be determined and provided to the algorithm, from which learning rates for basis functions and influence functions and an age weight (for local error filtering) can be derived. For example, if the distribution of inputs in the training data stream can be described by a probability density function, a suitably large pseudo-batch would contain inputs distributed roughly according to that PDF; the larger the pseudo-batch required, the smaller the learning rates should be to encourage robustness to adverse ordering of inputs in the training data stream.

Learning rates and age weight can be set individually, but it is convenient to link their values heuristically; for example—

influence function learning rate: $$\lambda_c = 1/\text{pseudo-batch size} \quad (3.21a)$$

basis function learning rate: $$\lambda_b = c_{rel} \cdot \lambda_c \quad (3.21b)$$

age weight for average local error: $$\alpha = 1 - \lambda_b \quad (3.21c)$$

Influence function parameters (centers, widths) are trained at the same rate at which a full pseudo-batch is encountered in the data stream; basis functions are trained faster than influence functions by a relative factor c_{rel}, to yield more fully trained basis functions for each instantaneous unit configuration; and averaging of local error occurs at a rate comparable to the rate at which that error is changing through basis function training.

3.4 Adaptive Structure: Automating Network Design

In general, the greater the complexity of the target function, the more units that will be required to construct an accurate approximation. Since this complexity is assumed to be initially unknown, and training data is unavailable until training actually begins, we have no knowledge with which to determine *a priori* a sufficient number of units.

If memory and machine cycles were cheap, and the engineer unconcerned with the elegance of the network design, the training domain could be populated densely enough with units to handle any potential target function. Unfortunately, neither is the case. In addition, over-parameterization can negatively affect generalization. A practical goal of learning, therefore, is to construct an accurate approximation of the target function with as few units as possible.

The optimum number of units can be sought through trial and error experimentation with different networks, but this can be tedious, time-consuming, and in the end ineffective. It is far more desirable to automate the process of network redesign, so that the learning algorithm not only sets parameter values, but also varies the network structure to achieve better results.

Chapter 4 discusses further the idea of adaptive network structure. A method for modifying network structure on-line is presented that meshes nicely with the preceding training algorithm to perform parameter training and network structure adaptation in parallel.

4 The Adaptive Structure Algorithm

4.1 The Adaptive Structure Approach

One could argue that for a particular target function there exists an optimum number of B/I units (of a predetermined type) that maximizes the efficiency of the approximation realized by the network. Given little or no *a priori* knowledge of the target function, this optimum number cannot be known and utilized at the outset of learning when the network is being constructed and initialized.

This thesis investigates and develops an approach that gives a B/I network the ability to modify its own structure on the fly, with the same freedom that parameter values are adjusted to meet the objectives of on-line learning. Beginning training with a modest number of units (perhaps only one), the algorithm essentially identifies local regions of the learning domain that lack the network resources (i.e., units) to construct adequate local approximations, and seeks to increase the population of units in those regions[1]. In effect, the algorithm allocates units over the domain according to the complexity of the target mapping in different regions.

A few attempts at adaptive structure have been made in this vein, and are summarized in the next subsection. Unlike this thesis, however, the success of these algorithms depends on the absence of noise in the training data, and does not consider the effects of adverse ordering and distribution in

[1]Adaptive structure could also be realized by modifying structures of individual units; e.g., the approximating power of a B/I unit can be increased by upgrading its basis function from affine to quadric. It will be assumed here that the structure of an individual unit is fixed; thus, network structure can be changed only by varying the number of units.

the training data stream. This thesis presents an improved algorithm more robust to such conditions, and assesses its performance when applied to learning tasks involving prediction of future states of nonlinear dynamical systems. Specific, novel features of this algorithm include:

- Proliferation of new units via splitting of old units in regions of large average error, rather than allocation of new units in order to correct errors associated with individual examples, which enables training to be done in the presence of both input and target noise.
- A simple quantitative measure of the extent of convergence of the parameters of individual units.
- Operation in parallel with a parameter training algorithm designed to facilitate structure changes.

4.2 Previous Attempts at Adaptive Structure

Three recent attempts to design adaptive structure B/I networks proceed along very similar paths [12, 16, 24]—each approach is presented below. The basic idea is to recognize large error-causing training examples that do not lie squarely within the sphere of influence of any existing network unit, and to improve performance on such poorly represented examples by adding units to the network.

Neither low influence nor large error alone are sufficient to require the addition of new units: errors that accompany examples that lie well within the influence of existing units should be reduced via further parameter adjustments, if possible; and outlying examples for which the network nonetheless yields correct outputs do not warrant additional units.

Although these algorithms perform well when trained with noise-free examples, they may fail in the presence of significant amounts of training noise: at any time, any example could appear to lie outside the influence of

existing units (due to noise at the network input), or to be poorly fit by the current parameter values (due to noise at the network output), or both.

After discussing these three algorithms in the remainder of this subsection, the adaptive structure algorithm developed for this thesis is presented, which addresses the issue of training noise directly.

4.2.1 A Gaussian Potential Function Network With Hierarchically Self-Organizing Learning —[Lee & Kil, 1991]

The network is composed of B/I units with constant basis functions and un-normalized spherical Gaussian influence functions. Training is done incrementally from noiseless examples generated uniformly at random from the input domain. Magnitude, center, and width are adjusted using a gradient update rule to satisfy a squared error cost function.

The adaptive structure algorithm is summarized below—

- start network with no units;
- present training examples to the network;
- if an example is presented that yields large error, and for which there is no unit that has large influence at that point in the input space, then create a new unit centered at that point, with basis function set equal to the target output for that example; otherwise, adjust parameters;
- if the overall network error stops decreasing before it falls below its target value, then increase the likelihood that new units will be created by requiring even greater influence values for each training sample;
- training proceeds until the overall error falls below its target value.

The required influence begins as a small value and is increased every time the network error levels off, in order to allow more units to be created

which will allow network error to decrease even further. The error criterion for new unit creation is a preset constant value.

Good results were obtained for a set of classification and function approximation tasks. In each case, the estimated "ideal" number of required units was compared to the actual number after training, showing that these numbers tended to be similar.

4.2.2 A Resource-Allocating Network for Function Interpolation— [Platt, 1991]

This is essentially a B/I network with constant basis functions, and influence functions that are polynomial approximations to spherical Gaussians. A global bias is also included. All adjustable parameters (basis functions, centers, global bias) are trained via gradient descent to minimize square error. The squared value of influence function width is proportional to the squared distance to the nearest neighboring center at the time a new unit is created; it is not subsequently updated. (The pseudo-Gaussian influence functions are trained using the update rule for actual Gaussians.)

New units are created in a fashion similar to that of Lee & Kil—

- start network with no units;
- present training examples to the network, randomly selected from a batch;
- if an example is presented that has large error, and for which there is no unit whose center is near that example (note: the criterion is *distance*, not influence), then create a new unit at that centered on that example; otherwise, adjust the network parameters.

The nearness criterion is initially set to be relatively lax (i.e., large), so that the network will first form a coarse representation of the training example set. This criterion decays exponentially with training time, so that

the training inputs are required to lie closer and closer to units in the network. Thus, as training progresses, narrower units are added to fill in gaps and thus create a finer representation of the data.

Unlike Lee & Kil, Platt does not pace his network-building with the leveling off of network error: units are created as necessary until the nearness criterion reaches its minimum value, after which time no more units are added. This nearness criterion can be lowered to achieve the required precision in the approximation.

A network was trained to predict the Mackey-Glass chaotic time series, with favorable results.

4.2.3 Representing and Learning Unmodeled Dynamics With Neural Network Memories—[Johansen & Foss, 1992]

This paper focuses on modeling the behavior of dynamical systems using B/I networks; specifically, storing *a priori* knowledge of the system of interest in a "first principles" network, and then training a second network (called the "black box" network) to correct/refine/augment the information stored in the first. The B/I architecture can be viewed as a set of models (basis functions), each paired with a *validity function* (influence function) that describes to what region of the input space that model pertains. Such an architecture readily lends itself to the straightforward incorporation of existing models of a dynamical system into a "first principles" network prior to training.

The networks are trained in parallel on examples generated on-line by the dynamical system. Whenever an example is found to have large error and low influence for both networks, a new unit is created in the "black box" network centered on that example. In general, the adaptive structure

approach here is similar to the previous two, except that there are two networks rather than one.

Johansen & Foss demonstrate in one application how new units are allocated to account for differences between system model behavior and that of the actual system; units tend to be created where the model deviates most from reality. In another application, a self-tuning controller is implemented.

4.3 The Adaptive Structure Algorithm

The above cases pertain to situations in which noise is absent from the training examples, and in which perfectly valid network structure modifications may be made in response to poor performance on individual examples. With the addition of noise, though, decisions of when and how to alter network structure cannot be made this way; rather, trends in network performance in different areas of the input domain must serve as the criterion for structure change.

As a general rule, if the current network performs poorly, on average, for examples that occur in a particular region of the input space even after the relevant network parameters in that region have converged to (quasi)static values, then it will be necessary to augment existing network resources in that region to obtain better performance. Ideally, a complete statistical representation of the current network error over the entire training domain would be constructed as training proceeds, and used to determine precisely where to allocate new units, i.e., which regions contain large persistent error. Realistically, though, this would amount to learning yet another mapping in addition to the target, significantly compounding the time and effort spent to train the network. A viable alternative used in this thesis is to keep track of the recent average error local to each existing unit (recall characteristic error

from Chapter 3). If it is determined that the local error of some unit has not yet decreased to an acceptably low value (after unit parameters converge), then that unit is *split* into two identical, independent units, yielding an increase in local approximating power.

Here is a brief statement of this approach to adaptive structure:

For some unit—

IF	parameter values have CONVERGED
AND	average local error is still TOO LARGE
THEN	SPLIT UNIT

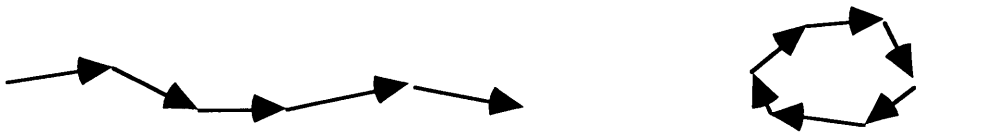
The winner unit of the current training example is tested for the splitting criteria and is split if appropriate. Once a unit has been split, it is up to the training algorithm to rearrange the network in order to accommodate and exploit the added resources.

The following subsections discuss parameter convergence criteria, average local error calculation, and the mechanics of splitting units, respectively.

4.3.1 Measuring Convergence of Unit Parameters

A unit should not be split whose parameters are still converging to steady-state values. However, in on-line, incremental training scenarios, convergence can only occur in a stochastic sense: parameter values migrate to some region of parameter space in which they may vary slightly over time, but without ever really leaving that region. It is this condition that must be achieved and detected before splitting can occur.

To illustrate, Figure 4.1 shows successive updates of a parameter vector $\mathbf{p}$ for some unit. In the first case, $\mathbf{p}$ is actually migrating through parameter space—it cannot be considered to have converged. The following case shows $\mathbf{p}$ after it has become confined to a particular region of space. Updates to $\mathbf{p}$ tend to cancel each other, preventing $\mathbf{p}$ from moving away from this region, on average.



successive updates don't cancel —> not converged

successive updates cancel —> converged

Figure 4.1. Convergence of parameters

The two types of behavior can be distinguished (or, rather, tendencies towards one or the other can be quantified) by measuring the extent of update cancellation. Let

$$d_{net}[n] = \left\|\overline{\Delta \mathbf{p}}_n\right\| = \text{length of time-averaged update vector at time } n \qquad (4.1a)$$

$$d_{tot}[n] = \overline{\left\|\Delta \mathbf{p}\right\|}_n = \text{time-averaged length of update vector at time } n \qquad (4.1b)$$

The quantity $d_{net}[n]$ reflects the size of the average net change in $\mathbf{p}$, *taking cancellations into account*, while $d_{tot}[n]$ measures the average size of the changes made to $\mathbf{p}$. Time-averaging is performed as in Eq. 3.19 for average local error:

$$\overline{\Delta \mathbf{p}}_{k+1} = \alpha_d \cdot \overline{\Delta \mathbf{p}}_k + (1 - \alpha_d) \cdot \Delta \mathbf{p}_k \qquad (4.2a)$$

$$\overline{\|\Delta\mathbf{p}\|}_{k+1} = \alpha_d \cdot \overline{\|\Delta\mathbf{p}\|}_k + (1-\alpha_d)\cdot\|\Delta\mathbf{p}_k\| \tag{4.2b}$$

where $\Delta\mathbf{p}_k$ is the k-th update of $\mathbf{p}$. Solving the difference equations gives

$$\|\overline{\Delta\mathbf{p}}_n\| = \left\|\sum_{k=1}^{n} c_k[n]\cdot\Delta\mathbf{p}_k\right\| = d_{net}[n] \tag{4.3a}$$

$$\overline{\|\Delta\mathbf{p}\|}_n = \sum_{k=1}^{n} c_k[n]\cdot\|\Delta\mathbf{p}_k\| = d_{tot}[n] \tag{4.3b}$$

where

$$\begin{gathered} c_k[n] = \alpha_d\cdot(1-\alpha_d)^{n-k} \\ 0 \le \alpha_d \le 1 \end{gathered} \tag{4.4}$$

From the triangle inequality, we know that

$$0 \le d_{net}[n] = \left\|\sum_{k=1}^{n} c_k[n]\cdot\Delta\mathbf{p}_k\right\| \le \sum_{k=1}^{n}\|c_k[n]\cdot\Delta\mathbf{p}_k\| = \sum_{k=1}^{n} c_k[n]\cdot\|\Delta\mathbf{p}_k\| = d_{tot}[n] \tag{4.4}$$

If there is very little cancellation among the parameter updates,

$$\frac{d_{net}[n]}{d_{tot}[n]} \approx 1.0 \tag{4.5a}$$

which indicates that $\mathbf{p}$ is actively migrating through parameter space. If the amount of cancellation is high,

$$\frac{d_{net}[n]}{d_{tot}[n]} \approx 0.0 \tag{4.5b}$$

and $\mathbf{p}$ has converged. In practice, this ratio is compared to a constant threshold θ_{conv} (selected empirically—0.1 or 0.2 seem to work)—

$$\frac{d_{net}[n]}{d_{tot}[n]} < \theta_{conv} \tag{4.6}$$

When the ratio falls below this threshold, convergence is signaled for that unit.

Convergence depends on scale, in both training time and parameter space. Age-weighting of parameter updates must extend far enough into the past to avoid signaling convergence in response to short-term biases in the training data distribution. This algorithm simply uses the learning rate for influence parameters to compute the time constant of the filter that performs the age-weighting:

$$\alpha_d = 1 - \lambda_c \tag{4.7}$$

This ensures that convergence is measured on a scale comparable to the slowest learning rates.

The nominal magnitudes of individual parameters may vary greatly. Large parameter changes will tend to dominate the values of d_{net} and d_{tot}, so that the failure of smaller parameters to converge may have little effect on this form of convergence detection. One recourse would be to monitor convergence for all parameters individually; another, to rescale each parameter (e.g., by dividing by its maximum observed value) before testing for convergence. Both methods require additional computation.

A simpler, and less expensive, way to counter the ill effects of scaling is to use only the center parameters to detect convergence. (In fact, this is exactly what was done to generate the results in this thesis.) Scaling of the input space, wherein centers reside, is usually known *a priori* to some extent, so that rescaling before training is possible if necessary. In addition, since centers are trained at a very slow rate which allows the basis function parameters to maintain a "mostly trained" state, it is reasonable to assume that when the

center of a unit has converged, so have its remaining parameters (if, in fact, the basis functions remain "mostly trained").

4.3.2 Rating Local Error

A unit should not be split if the local approximation has already achieved an acceptable level of accuracy. That is, if the average local error computed by that unit is sufficiently small, the unit should be left alone.

The simplest approach, which is taken in this thesis, compares average local error to the desired average global error (provided at the outset of training): units that exceed this error are candidates for splitting. This way, regions lacking sufficient resources to meet the error goal receive new units as soon as possible. An alternative approach would be to use the actual, rather than desired, average global error, to split units that perform badly relative to the current overall network performance. This possibly would lead to more gradual, uniform training over the input domain.

4.3.3 Splitting Units

If the current winner unit meets the convergence and local error criteria, that unit is split.

To "split" a unit means to replace it with two similar units, thus doubling the effective number of parameters available to construct and augment the local approximation. Splitting should seek both to preserve information gained through prior training, and to facilitate the assimilation and utilization of the extra approximation capability by the network.

A seamless transition from old network structure to new can be achieved if the network output function after splitting is identical to the previous one. This is the case when the two new units retain the same basis

and influence parameters as the old unit, with the influence functions each scaled down by half:

$$
\begin{aligned}
y(\mathbf{x}) &= \frac{B_1(\mathbf{x}-\mathbf{x}_1^c)\cdot i_1\left(\left\|\mathbf{x}-\mathbf{x}_1^c\right\|\right)+\cdots}{i_1\left(\left\|\mathbf{x}-\mathbf{x}_1^c\right\|\right)+\cdots} \\
\xrightarrow{\text{split}} y(\mathbf{x}) &= \frac{B_1(\mathbf{x}-\mathbf{x}_1^c)\cdot\left[\frac{1}{2}\cdot i_1\left(\left\|\mathbf{x}-\mathbf{x}_1^c\right\|\right)+\frac{1}{2}\cdot i_1\left(\left\|\mathbf{x}-\mathbf{x}_1^c\right\|\right)\right]+\cdots}{\frac{1}{2}\cdot i_1\left(\left\|\mathbf{x}-\mathbf{x}_1^c\right\|\right)+\frac{1}{2}\cdot i_1\left(\left\|\mathbf{x}-\mathbf{x}_1^c\right\|\right)+\cdots} \\
&= \frac{B_{1a}(\mathbf{x}-\mathbf{x}_{1a}^c)\cdot i_{1a}\left(\left\|\mathbf{x}-\mathbf{x}_{1a}^c\right\|\right)+B_{1b}(\mathbf{x}-\mathbf{x}_{1b}^c)\cdot i_{1b}\left(\left\|\mathbf{x}-\mathbf{x}_{1b}^c\right\|\right)+\cdots}{i_{1a}\left(\left\|\mathbf{x}-\mathbf{x}_{1a}^c\right\|\right)+i_{1b}\left(\left\|\mathbf{x}-\mathbf{x}_{1b}^c\right\|\right)+\cdots}
\end{aligned}
\tag{4.8}
$$

The drawback is that the new units are "weaker" (in an influence sense) than their predecessor. Over many generations, a significant disparity in influence would come to exist between older and more recently created units. It may be possible to train influence magnitudes to eliminate this problem. However, avoiding it altogether is preferable.

If inverse square influence functions are used, halving the influence magnitude is equivalent to scaling down the width by the square root of 2—the transition is made seamless by making the new units narrower. Reducing unit widths introduces another potential benefit for training: more units in a given region implies a finer-grained approximation, which in turn implies narrower widths—so automatically narrowing the unit widths can be seen as giving the new units a head start on assimilation into the network. This head start can be given to Gaussian units by a similar narrowing of widths, although without affording a perfectly seamless transition. This algorithm gives the option of halving unit widths during a split. This effectiveness of this width reduction strategy is investigated in this thesis.

Immediately after a split, the clustering algorithm will separate the new units, so that the region that was previously represented by a single basis function is now represented by two independent basis functions. Note that a gradient descent center training algorithm (or any non-"winner-take-all" approach) would not separate the new units, but rather would move them around together—after all, the new units are identical. This illustrates how nicely the training algorithm and adaptive structure algorithm work together.

The error, "win", and convergence statistics of new units are initialized based on those of the old unit. Average local error is passed on unchanged. The number of "wins" (i.e., the number of past examples for which the unit has been selected winner—used for distribution weighting) is divided equally between the new units. Values for net distance and total distance traveled are reinitialized—the new units must converge to new locations.

4.4 Recap: The Complete Learning Algorithm

Now that every aspect of the algorithm has been covered in detail, a concise summary is in order. This section outlines the steps that must be taken to initialize the network and algorithm before training commences, and presents a flow chart of algorithm function during training.

4.4.1 Initialization

Before training begins, the specific learning problem must be stated, the network structure must be initialized, and training parameters must be provided to the algorithm (Figure 4.2).

Information regarding the training domain, as well as any *a priori* knowledge of the target mapping, can help determine the initial structure and parameter values of the network. As a bare minimum, input and output dimensions must be specified. The initial number of units, as well as the type

of influence function used (Gaussian or inverse square) can be specified. Bounds on the training domain can be used to initialize the centers and widths of units so that they cover the domain completely.

Target error, pseudo-batch size, and the basis function scale factor c_{rel} are used to compute learning rates and age weights for the algorithm. The user must provide values for the convergence threshold Θ_{conv}; and the minimum required cumulative normalized influence of the set of most influential units that are actively evaluated and trained on the current example.

4.4.2 Training

Given the above network set-up and a stream of training examples, network parameters and structure are modified on an example-by-example (incremental) basis (Figure 4.3):

1) Distances, and then influences, from the current input to all units are calculated. The winner unit is determined based on either proximity or influence (proximity is the criteria selected for these results).
2) Units are sorted according to influence, and the set of N most influential units is determined such that the cumulative sum of their normalized influences exceeds some threshold value. Only this "active" set of units is subsequently used by the algorithm for network output computation and training.
3) Basis functions are evaluated and network output calculated.
4) Basis function parameters of all active units are updated; influence parameters only of the winner are adjusted.
5) Winner statistics are updated: average local error and convergence statistics.
6) If all splitting criteria are met, the winner unit is split.

7) Average global error (global RMS error) is updated. When this value falls below the target error, the algorithm may be stopped. (Alternatively, training could proceed until all average local errors become less than the target error, which is an even more stringent objective accompanying the cessation of new unit allocation).

8) The resulting network is evaluated on a stream of new examples generated by the same process (having the same distribution) from which it was trained. The true average error (rather than filtered error) is calculated for a large set of examples and is compared with the target error for purposes of rating learning performance. Measuring error in this way yields an estimate of the expected error of the network on the given data stream.

4.4.3 Performance Expectations

The algorithm as presented thus far is not chiseled in stone—there are a number of options available that may or may not enhance learning performance. Subsequent chapters will report results of learning tasks undertaken by the algorithm, and will comment on the effectiveness of these various features.

The algorithm will be considered successful if it consistently demonstrates the ability to achieve target errors through the allocation and training of B/I units in response to a stream of training examples distributed arbitrarily over the training domain.

Assuming that a given target mapping may be approximated within a given target error with a finite number of units, the algorithm should allocate units until a sufficient number have been created, at which point average global error should approach target error, and allocation of new units should level off. This halt in network growth implies that the average local errors of all units have fallen below the target error, in turn implying that the average

global error must have done likewise. Starting from a single unit, the algorithm will require some time to "grow" to a size sufficient to achieve the desired accuracy; the higher the accuracy, the larger the network needed, and the longer the training time. Initializing the network with more than one unit (randomly placed within the envelope of interest) might possibly abbreviate training by beginning with a network nearer the final size required. However, if the training envelope is incorrectly specified, it is possible for some units to lie in areas devoid of training examples, so that these units never "win" and thus are never trained—they may become stranded and superfluous.

Approximation error should decrease on average as training proceeds and the network size increases. However, due to the potentially uneven distribution of training inputs and the unsupervised influence training algorithm, error may actually increase sporadically over small time intervals as the training data wander through space. One can also expect the temporary increase of average network error following the splitting of a unit accompanying the subsequent reordering of the network. If training is finally successful, the expected error of the network, as measured on a set of examples from the training stream, should be less than the target error, although the error for some individual inputs may exceed the target error.

In the absence of any facility for removing units from the network, i.e., "pruning", and with the possibility of creating "stranded" units, it is possible that the final number of units will be greater than necessary in some cases, particularly if the criterion for parameter convergence is too lax. In addition, it may take quite a bit of training before unit allocation ceases; one would expect relatively small target errors to require relatively many units, leading to prolonged training times.

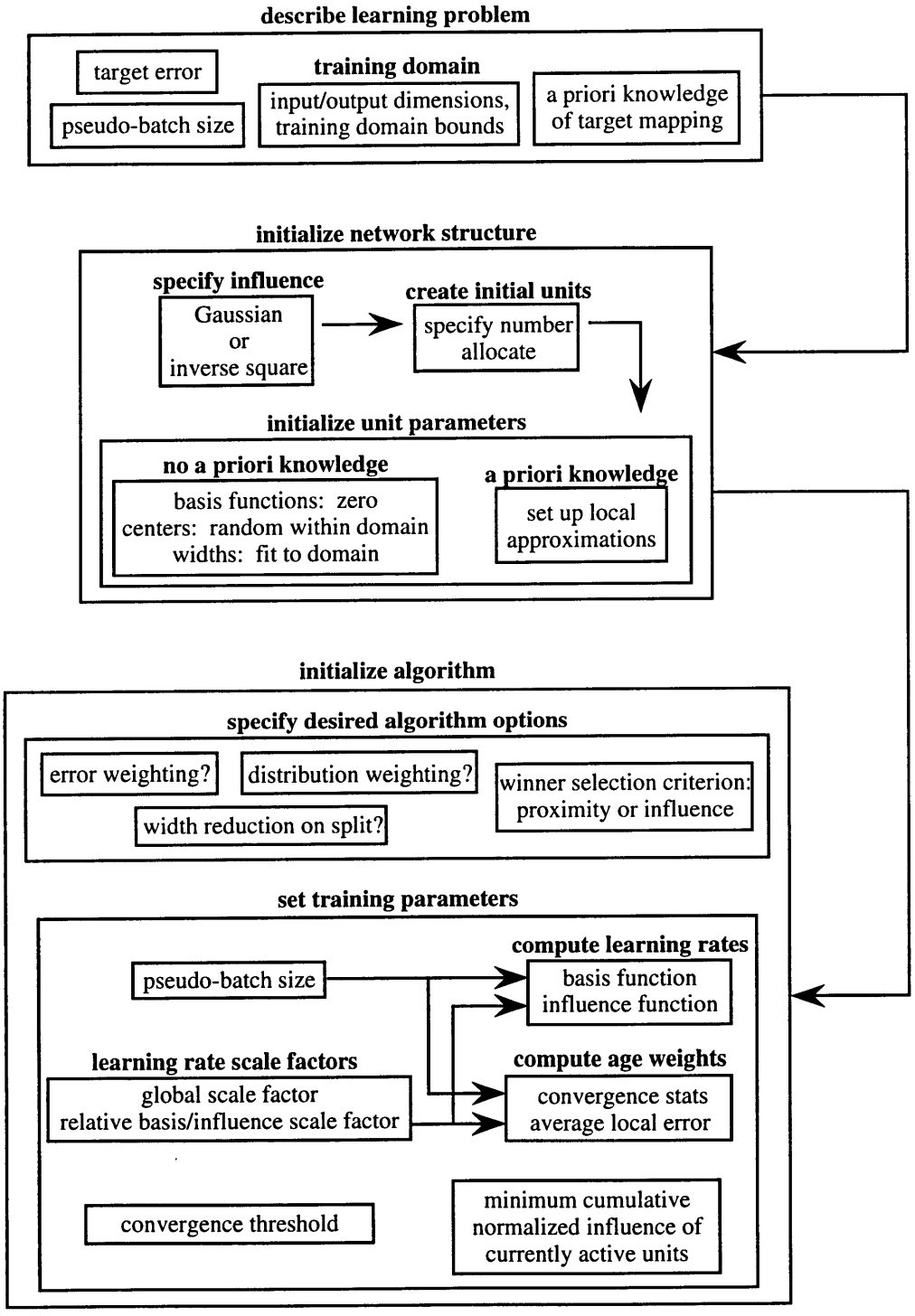


Figure 4.2. Network initialization

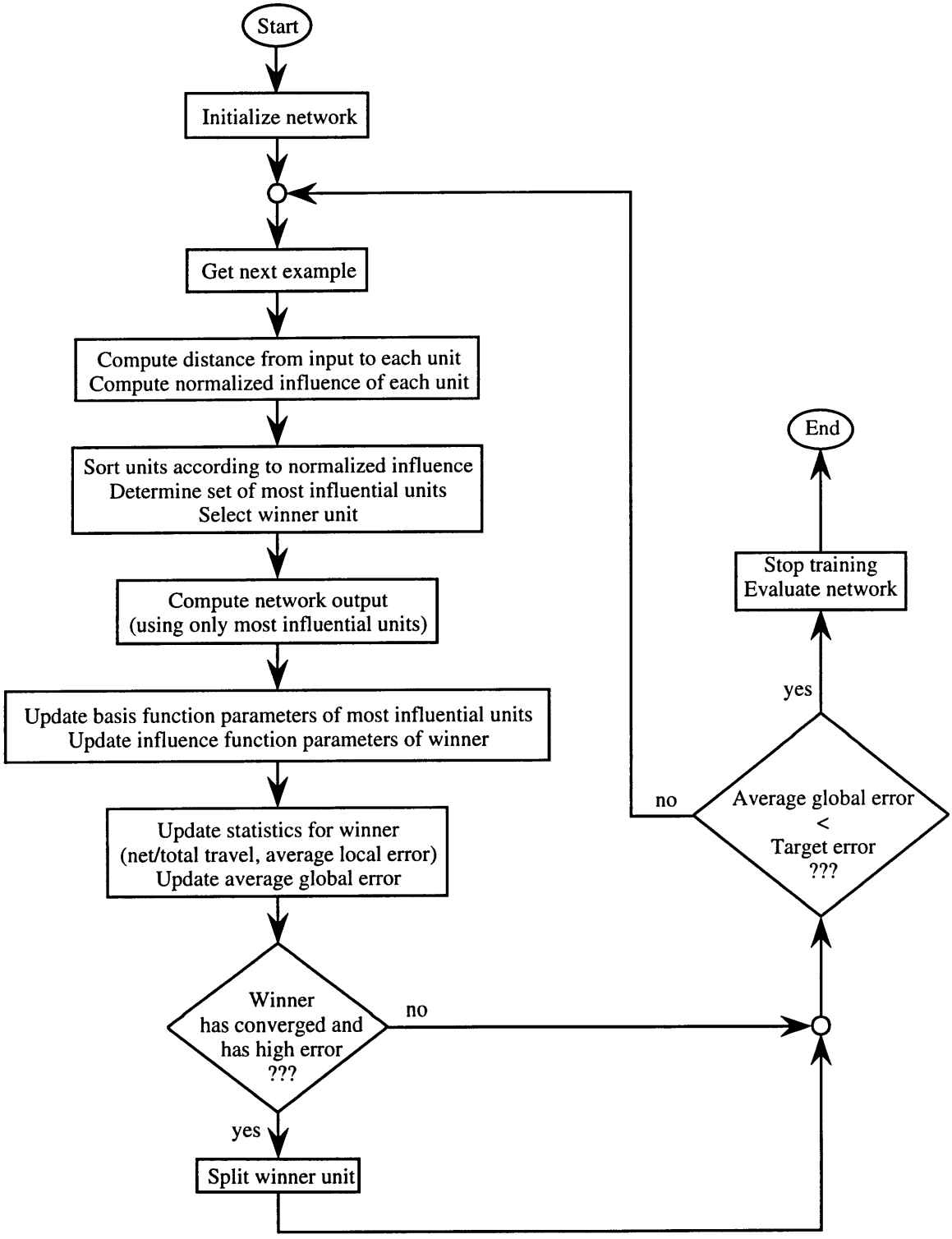


Figure 4.3. Flowchart of Learning Algorithm

5 An *n*-Step Predictor for an Aeroelastic Oscillator

A set of networks are trained to predict future states of a (simulated) nonlinear dynamical system from observations of its behavior over time. Performance of each of the resulting networks is then evaluated by measuring the average error associated with predictions of future state made by the network as it again observes the system in the same manner as during training.

5.1 Position Prediction for an Aeroelastic Oscillator

Specifically, the problem is to predict the future position of an aeroelastic oscillator (see Appendix B for details) from current position and velocity information (Figure 5.1).

The adaptive structure network is trained with a stream of examples (i.e., the current state and the resulting future position) formed from observations of the oscillator as it evolves along state-space trajectories of fixed duration, starting with initial velocities of zero and random positions within the interval [–0.5, 0.5]. Once the oscillator has been given an initial state, no control is exerted over its travel through state-space—the ordering of training examples in the data stream and the distribution over the training envelope are allowed to be non-uniform. Thus, training tends to become fixated along the two stable limit cycles. In a few cases noise is added to the inputs and targets given to the network.

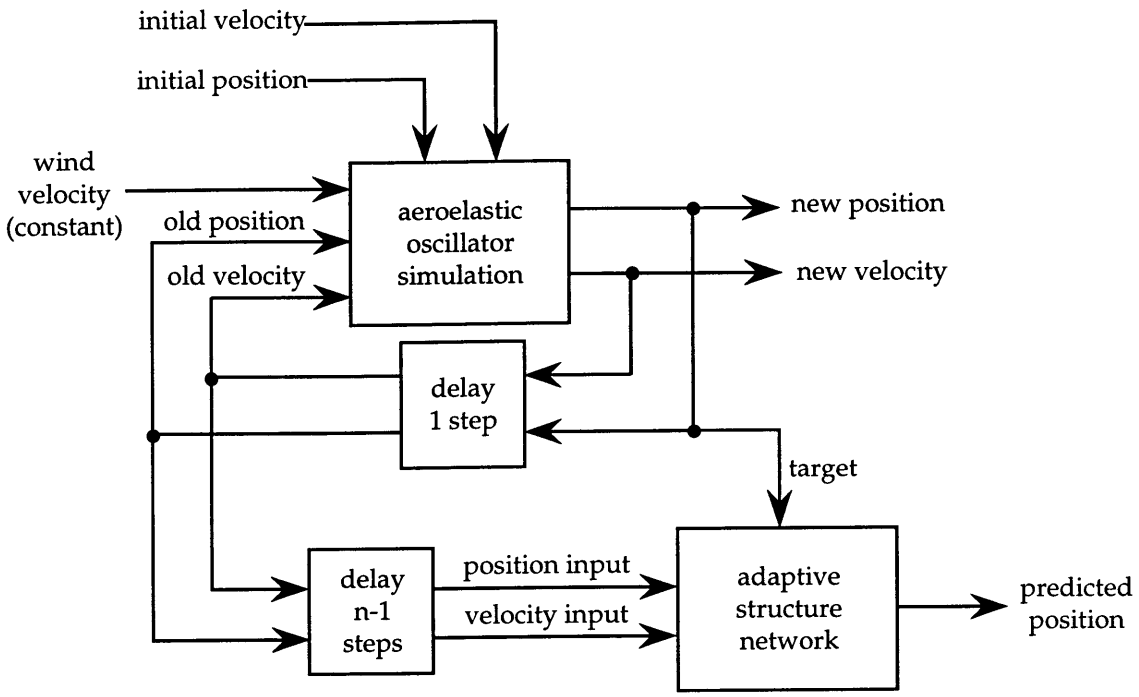


Figure 5.1. Training Set-up for Prediction

The goal is to construct a mapping from the current position and velocity to the position n time steps hence (Figure 5.1). Equivalently, we can predict *the net change in position* over these n time steps:

$$y_{net}[k] = pos[k+n] - pos[k] = f(pos[k], vel[k]) \tag{5.1}$$

Since changes in position may be much smaller in magnitude than absolute position, training the network to predict net position change rather than absolute position eliminates some of the coarse information the network otherwise would have had to learn, and allows it to better focus on the finer details of the mapping.

The specific problem has been fixed at n = 10, yielding a target mapping represented by the surface shown below (Figure 5.2a). Even after restricting the target mapping to position change rather than position, there still exists a significant linear component that predominates over the nonlinearities in

the mapping. It is reasonable to assume that changes in velocity over the 10 time steps are too small to cause position changes that deviate greatly from those that would be observed in a purely linear system. Thus, the target can be analyzed into two components representing linear behavior (Figure 5.2b) and nonlinear behavior (Figure 5.2c), as described by Eq. 5.2:

$$y_{net}[k] = \overbrace{vel[k] \cdot n \cdot \Delta t}^{\text{linear component}} + \overbrace{g(pos[k], vel[k])}^{\text{nonlinear component}} \tag{5.2}$$

The domain for these plots is restricted to the region of input space in which training occurs. Referring to Figure B.3 of Appendix B, the outer limit cycle, within which all training examples are confined, is tightly bounded by positions on [–0.65, 0.65] and velocities on [–0.45, 0.45]; all plotting is performed over the corresponding rectangle.

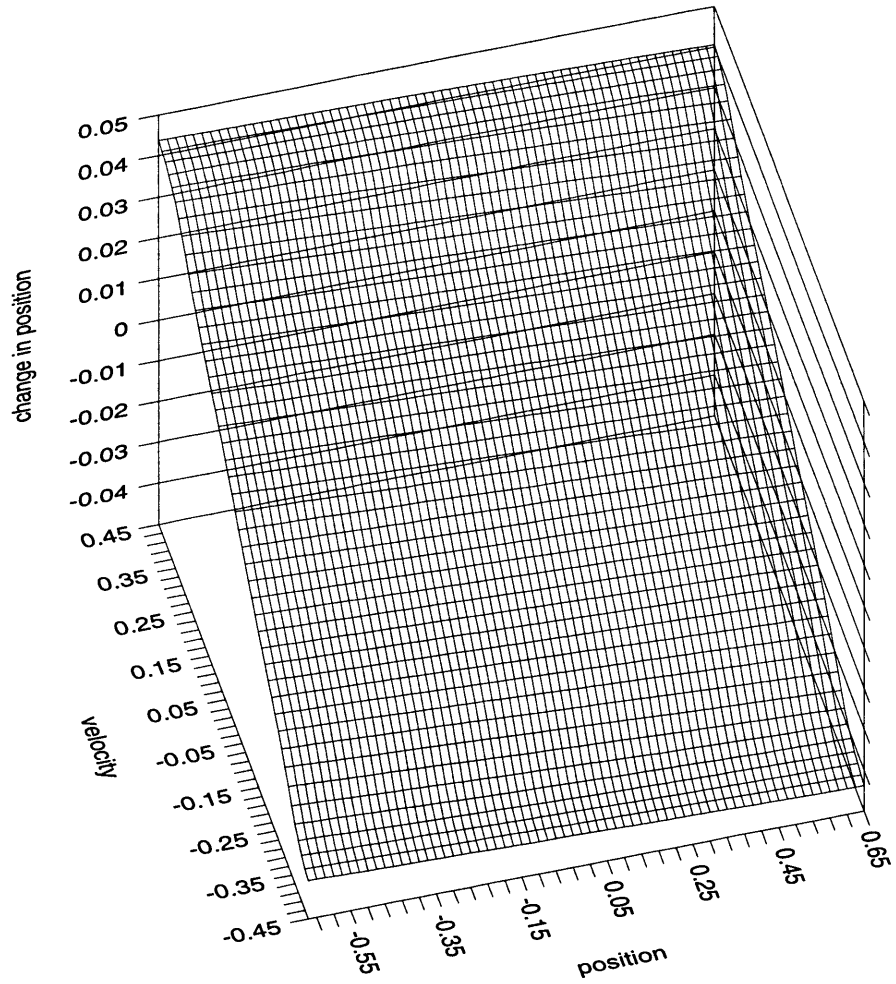


Figure 5.2a. A plot of the change in position after 10 time steps vs. system state. The plot is dominated by the linear behavior of the system, i.e., the change in position that would result if velocity remained constant. Maximum/minimum values are approximately ±0.046.

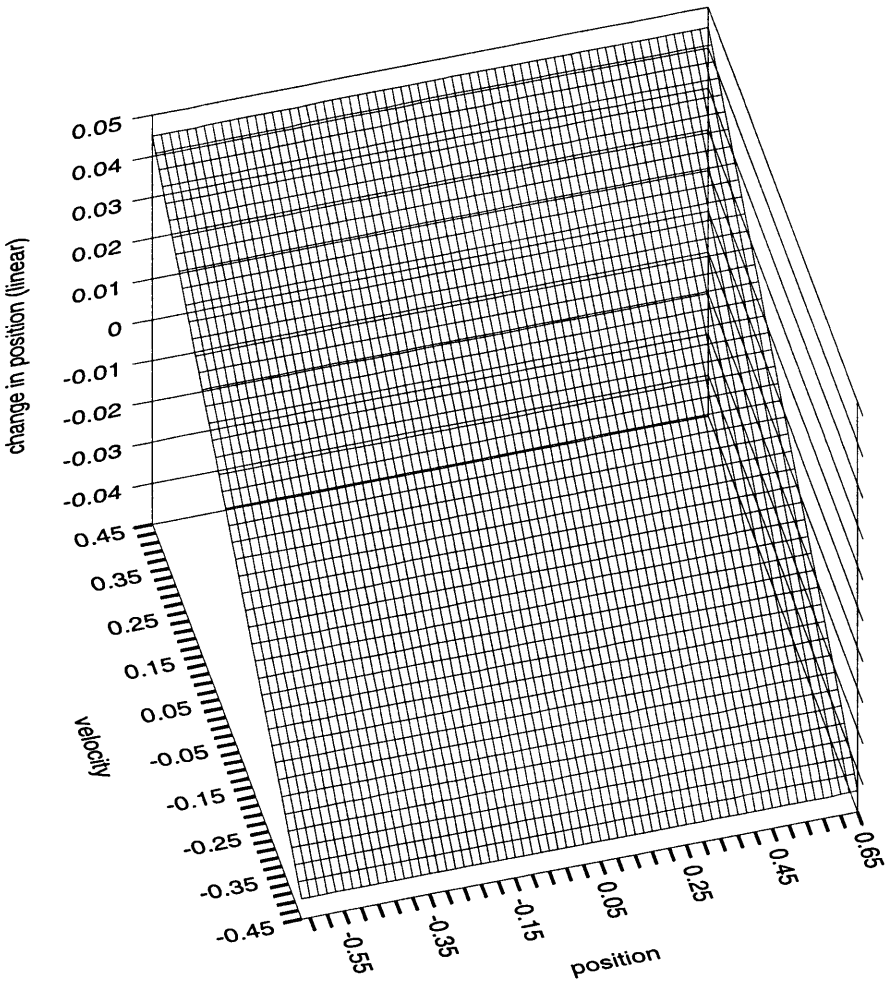


Figure 5.2b. A plot of the dominant linear component of the target mapping. Maximum/minimum values are approximately ±0.045.

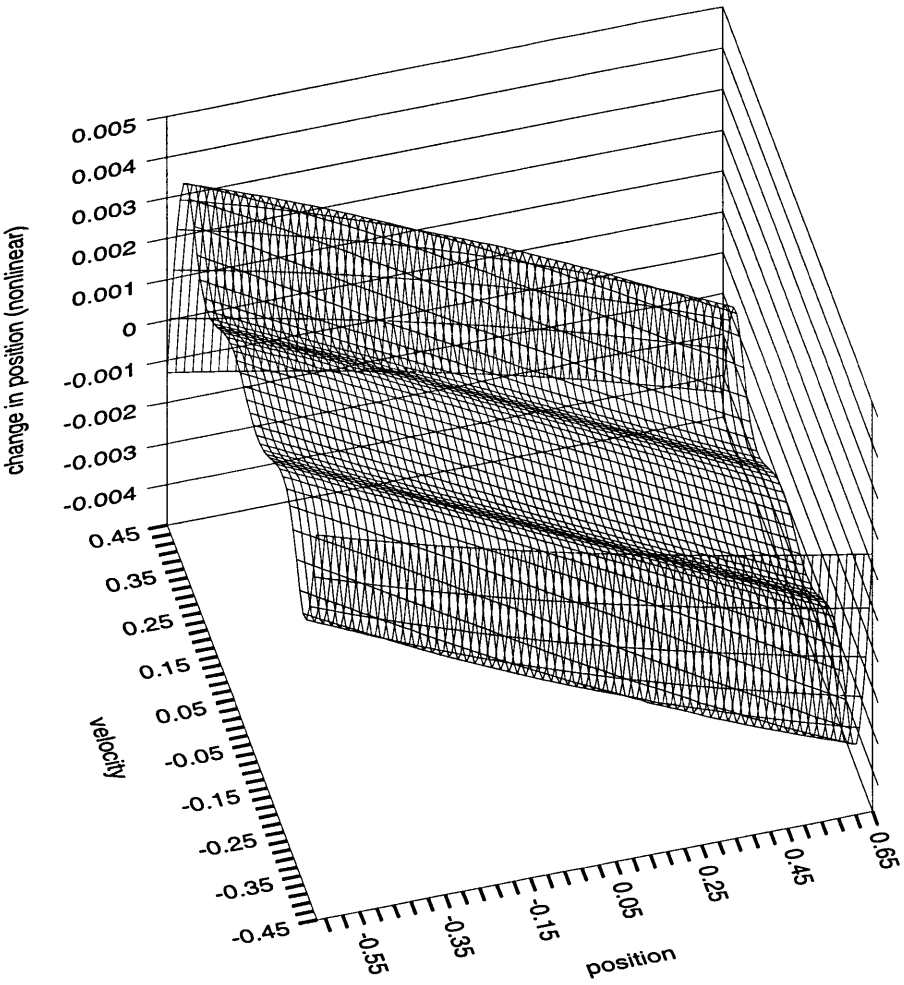


Figure 5.2c. The nonlinear component of the target mapping. Maximum/minimum values are approximately ±0.0046, or 10% of that of the target mapping.

5.2 Evaluation of Adaptive Structure Hybrid Algorithm

The effects of different features of the adaptive structure hybrid algorithm are evaluated. First, benchmark results are obtained by training a network with noise-free examples generated from state-space trajectories, using a version of the algorithm with an arbitrary set of active features (e.g., error weighting). In subsequent runs, features of the algorithm are varied, training parameters are changed, and/or training conditions are altered; the different predictors that result are then evaluated and compared.

5.2.1 Benchmark Run

5.2.1.1 Training set-up

The network was initialized with a single Gaussian unit of width $\sigma = 0.2$ centered at the origin. All added units were also Gaussian.

One pseudo-batch was set to equal roughly the number of examples occurring during two cycles of the aeroelastic oscillator, or 1257 examples. The network was trained with 800,000 examples = 637 pseudo-batches. Network validation used 200,000 examples = 159 pseudo-batches.

The training algorithm ran with these features/parameters:

error weighting:	ON
distribution weighting:	ON
variance halving on split:	ON
input noise variance:	0.0
target noise variance:	0.0
target error:	0.0005
characteristic squared error:	= (target error)^2 = 2.5x10^{-7}
convergence threshold:	0.2
cumulative norm. influence:	0.95
basis function relative learning rate scale:	5

The resulting network was evaluated using a stream of examples generated as during training. Validation error is simply the true RMS error over the set of N validation examples:

$$\varepsilon_{\text{valid}} = \sqrt{\frac{1}{N}\sum_{k=1}^{N}\left(y_{\text{targ}}[k] - y_{\text{net}}[k]\right)^2} \tag{5.3}$$

5.2.1.2 Results

Some important training results are presented for the benchmark run by Figure 5.3 (similar figures will summarize the results of later runs):

1) *Final network configuration*—The set of units at the termination of training is represented by circles whose radii represent unit widths and whose centers coincide with unit centers.
2) *Network growth over time*—The number of units over the course of training is contrasted with the size of the set of "active" units, i.e., those units of greatest influence at the current example, whose cumulative normalized influence meets the desired threshold.
3) *RMS training errors*—RMS error is simply the square-root of low-pass filtered squared error. Maximum and minimum RMS local errors over all units are plotted, as well as RMS global error for the network as a whole.

Referring to the first plot, the most striking feature of the network configuration is the way in which nearly all the units have aligned themselves along the two limit cycles, with just a few units located in areas corresponding to the initial arcs of trajectories (cf. Figure B.3). Widths have been adjusted according to relative distances between neighboring units, so that coverage among these units is fairly efficient. Cases in which a pair of units overlap a great deal can be attributed to a recent splitting—in fact, one

can see pairs of units in various states of post-split separation, where pairs that overlap greatly tend to be very similar. Most of the splitting still in progress appears to occur where "filling in" is required, or in areas along the limit cycles adjacent to underrepresented initial arcs.

Addition of new units to the network initially occurs on the order of every 10,000 examples, later becoming more infrequent as the local errors of individual units begin to fall below the target error, thus making them ineligible for splitting. Transient errors accompanying these structure changes can be observed in the error plots as brief rises or humps. The number of active units stays well below the total number of units for the duration of training, and in fact appears to stabilize to a roughly constant value.

Notice that the error plots initially rise before finally beginning to decrease. This is due to the fact that these errors are derived from the outputs of low-pass filters, which require time to "charge up" from zero. Once this has occurred, the progress of learning becomes apparent. After approximately 13,000 examples, the first unit splits, and distinct maximum and minimum local errors appear. These then separate, with the maximum increasing and then generally leveling off, while the minimum tends to decrease throughout the remainder of training. Global error remains between the two and decreases on the whole.

The final values of local RMS error, when plotted in order of creation of the corresponding units (Figure 5.4), shows that older units tend to be more "well-trained", having lower local errors than newer units. Presumably, if training were to continue for a long enough period, more units would achieve the target error, and unit allocation would level off, eventually plateauing at a constant network size.

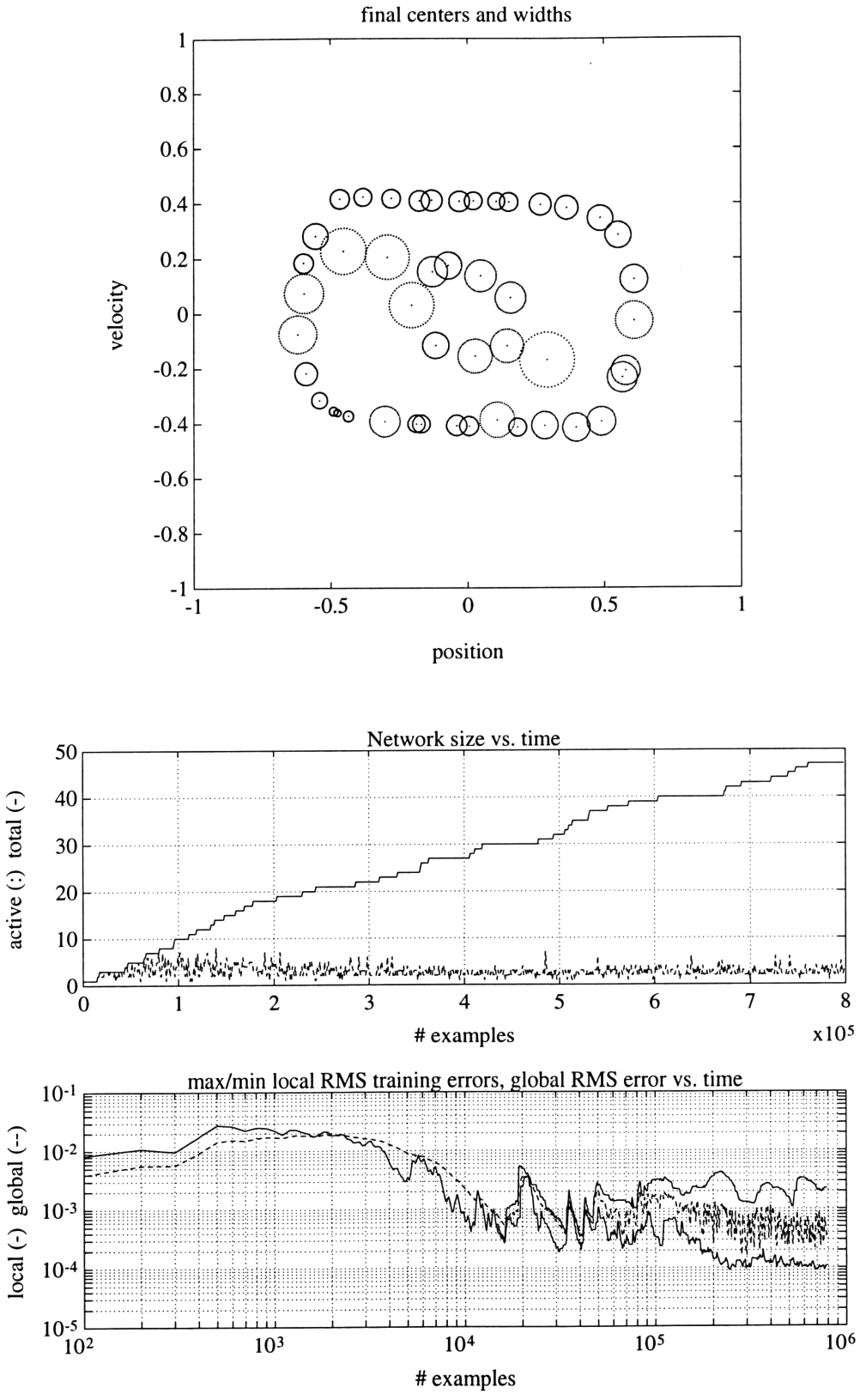


Figure 5.3. Results of benchmark run

The network reaches a size of 47 units by the time training is stopped. The validation error (4.79×10^{-4}) falls within the jitter of the final global training error. A plot of the final network mapping (Figure 5.5), subtracting away the linear portion of the target and compared with the nonlinear component of the target (Figure 5.2c), reveals that training has captured the general tilt of the target mapping, but the two large ripples are not apparent.

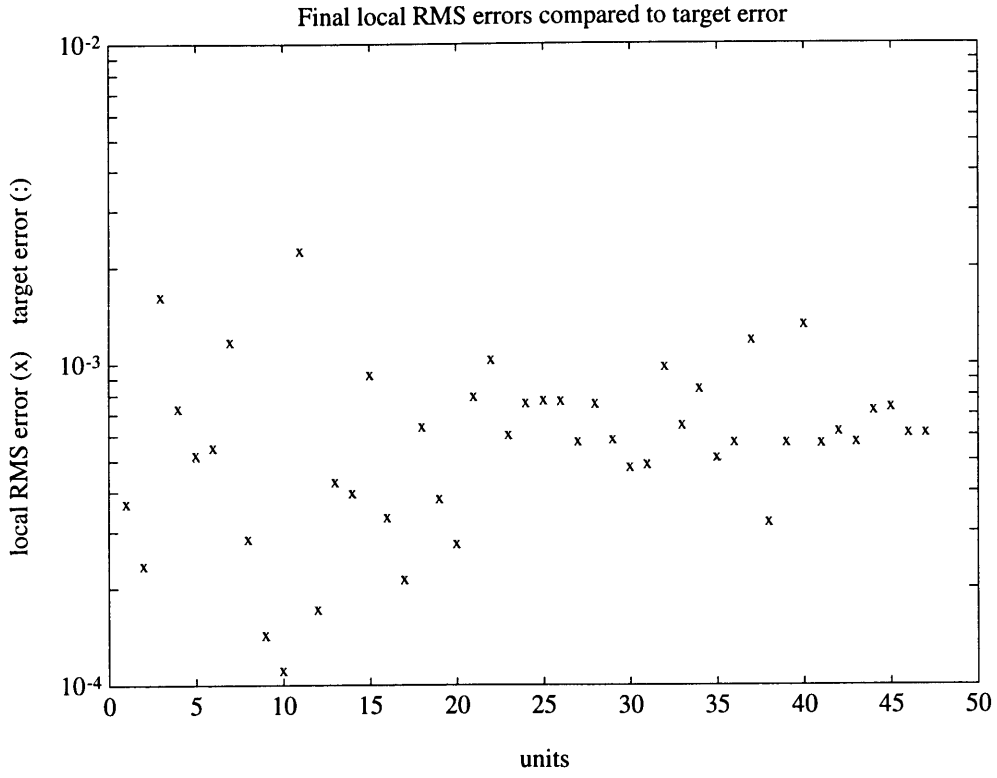


Figure 5.4. Final local errors of individual units of benchmark network (in order of creation)

The mapping in the corners, corresponding to areas that lie outside the training domain, are particularly erroneous; for example, the upper left corner of the network mapping rises steeply when it should fall. (This illustrates the fact that extrapolation cannot and should not be performed with networks such as these.) Overall, the network mapping is much less smooth than the target.

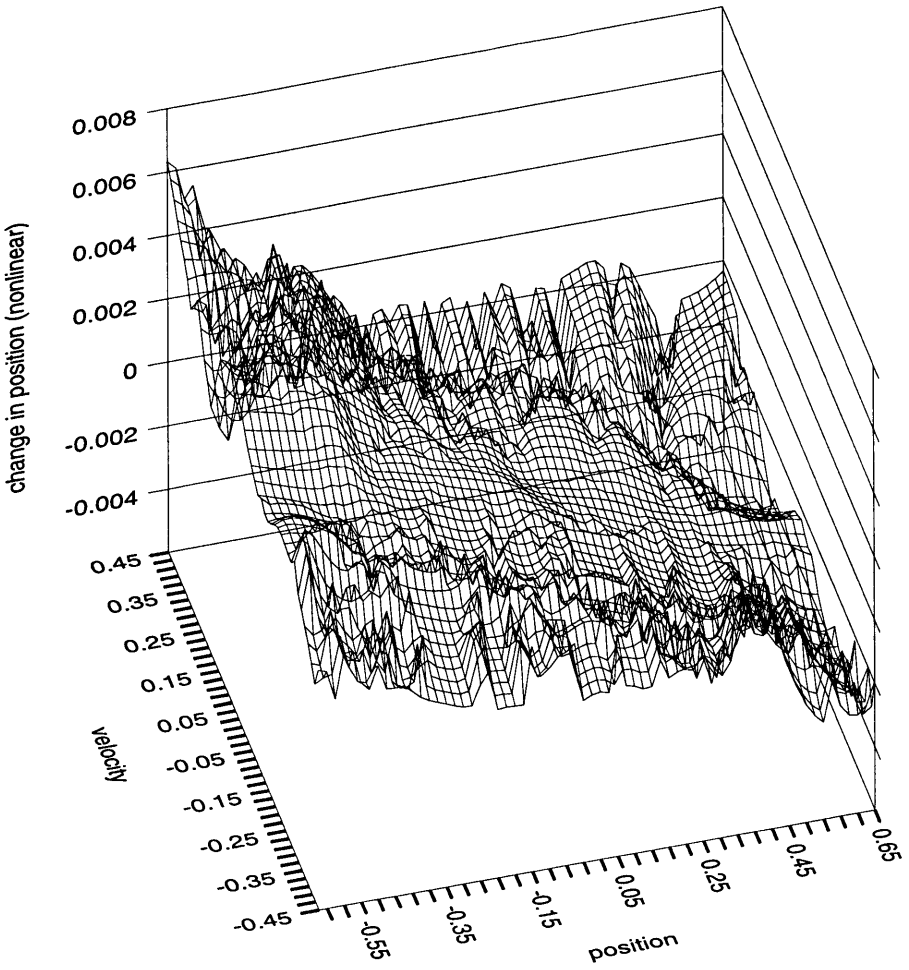


Figure 5.5. Final network mapping obtained from benchmark run. Maximum/minimum values are approximately ±0.0063.

5.2.1.3 Discussion

Consecutive training inputs, as they appear within trajectories of the aeroelastic oscillator, are correlated such that inputs that occur at neighboring positions in the stream are also neighbors in input space. This makes possible the condition in which training inputs remain temporarily confined to a particular region of space, potentially causing forgetting of previously learned information in more distant regions of the input space.

The network units reach a quasi-steady arrangement (allowing for creation of new units) despite uneven ordering of examples, which might be

expected to cause the network to "converge" to a series of different temporary solutions rather than to a single solution. Apparently, the spatially localized architecture, combined with judicious choice of pseudo-batch size (and thus learning rate) is robust to example order.

As well as being unevenly ordered, the majority of training examples occur along limit cycles, so that they are also unevenly distributed over the space—some regions (limit cycles) receiving more training than others (initial arcs of trajectories not along limit cycles). This uneven distribution has a pronounced effect on the eventual unit arrangement, causing units to be more densely populated in regions of heavier training. This gives good coverage of the most frequently visited regions, at the expense of coverage in less frequently visited regions. From the standpoint of minimizing expected error, this may be fine; however, if other error norms are targeted for minimization—say, a 2-norm taken uniformly over the space—failure to cover some regions may make it impossible to meet the learning goals.

Distribution weighting was employed to make training more uniform over those portions of space where examples occur, and thus to yield more even coverage. Clearly this goal has not been met here, although the weighting approach might eventually prove successful in some other form. In no case, however, could distribution weighting account for the complete absence of examples from some region, as in the outlying regions of this training envelope.

Validation error was comparable to final training error, i.e., it lay within the "jitter" of training error at the close of training. This confirms that the network is retaining what it has learned, rather than just quickly adapting to the current training situation at the expense of knowledge gained previously. Since the validation example set was generated according to the

same distribution as the training set, the validation error is an approximation to expected prediction error of the trained network, and thus a meaningful measure of future performance *for operating conditions similar to those under which it was trained.*

In a sense, the network generalizes well, since as long as examples are drawn according to the training distribution, average performance errors are no worse than final average training error. However, in another very important sense, generalization is bad: if the distribution is changed so that inputs are frequently chosen from previously unfamiliar regions, performance error will increase. A successful approach to distribution weighting should help alleviate this problem. From a smoothness standpoint, generalization is not good at all. This may be an indication that unit widths should overlap more, and thus produce smoother transitions among neighboring basis functions.

Restricting training to the set of most influential units has resulted in an enormous decrease in the amount of adjustments that must be done at each time step, the only trade-off being the amount of time necessary to sort the current set of units according to normalized influence. As the network size increases, it appears that the net gain in computational efficiency also increases, since the number of active units remains fairly constant as the total number continues to grow. Thus, there is some relief from the "curse of dimensionality" that can plague spatially localized network architectures.

5.2.2 Variations on Benchmark Run

Except for the noted differences (and different random training trajectories), all of the following runs were identical in every respect to the benchmark.

5.2.2.1 Architecture: Inverse square influence functions

Inverse square influence functions were used in place of Gaussian influence functions (Figure 5.6). The final validation error (3.82×10^{-4}) was less than the benchmark using fewer units (38). However, despite the appearance of more efficient training, the number of active units was much higher than with Gaussian units: as the network size increased, so did the average number of units trained on each example, in contrast to the benchmark run, where the number of active units was limited to two or three, on average. This translates into a potentially enormous computational cost, a big strike against this type of architecture.

5.2.2.2 Algorithm

Pre-allocated units, no splitting (Figure 5.7)

The network was initialized with 47 units (i.e., the final size of the benchmark run), randomly placed within the training envelope with widths of 0.17. Splitting was disabled.

Only 16 units were eventually repositioned and resized by the algorithm, the others lying too far away from the training inputs. This underutilization of units drastically reduced the approximating power of the network, resulting in an especially high validation error (1.97×10^{-3}).

Had the units been initialized in a grid rather than at random, it would be reasonable to expect a similar case of unused units. The point illustrated by this run is that the adaptive structure algorithm facilitates more efficient placement and subsequent utilization of units by creating them only where needed.

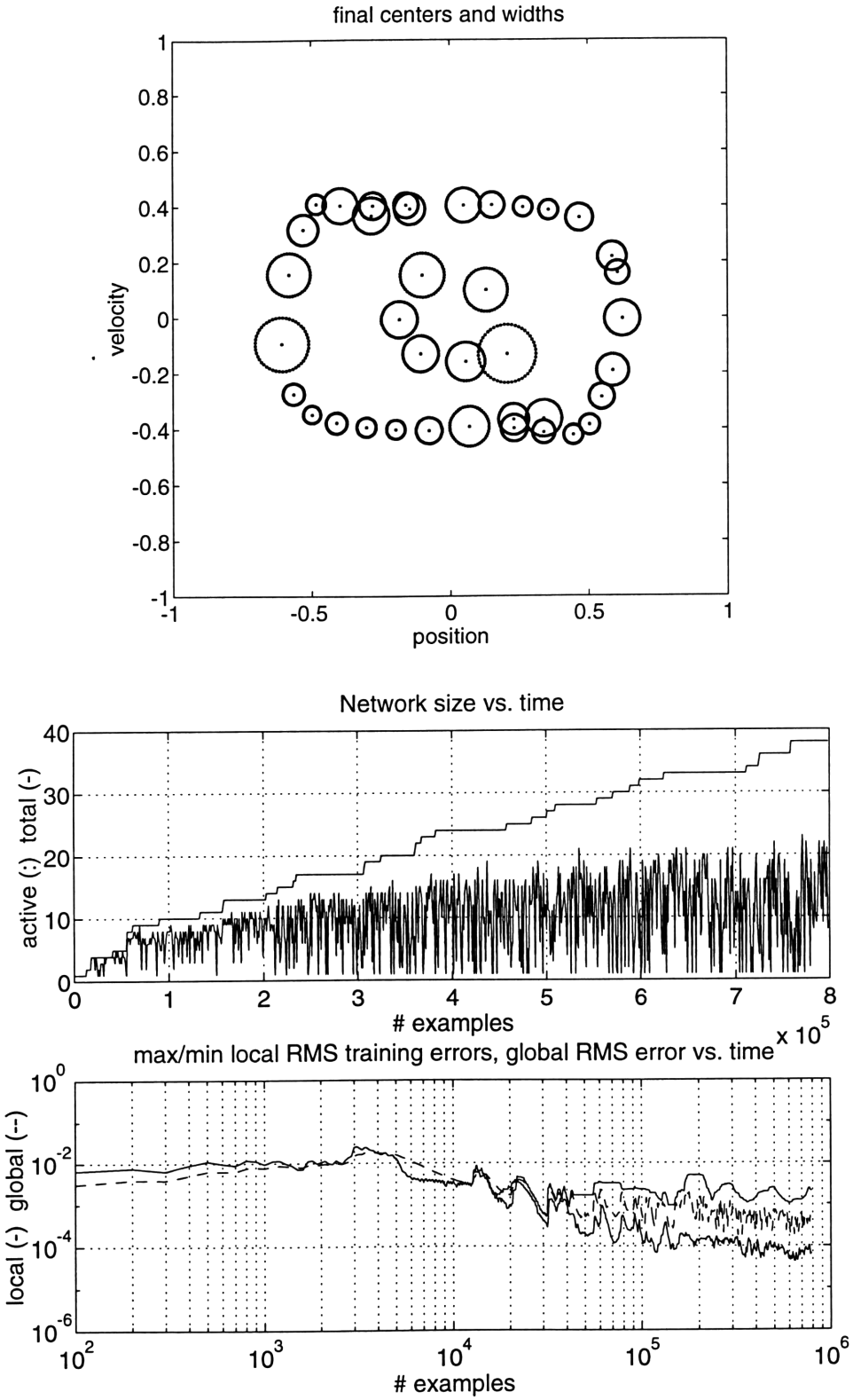


Figure 5.6. Inverse square influence functions

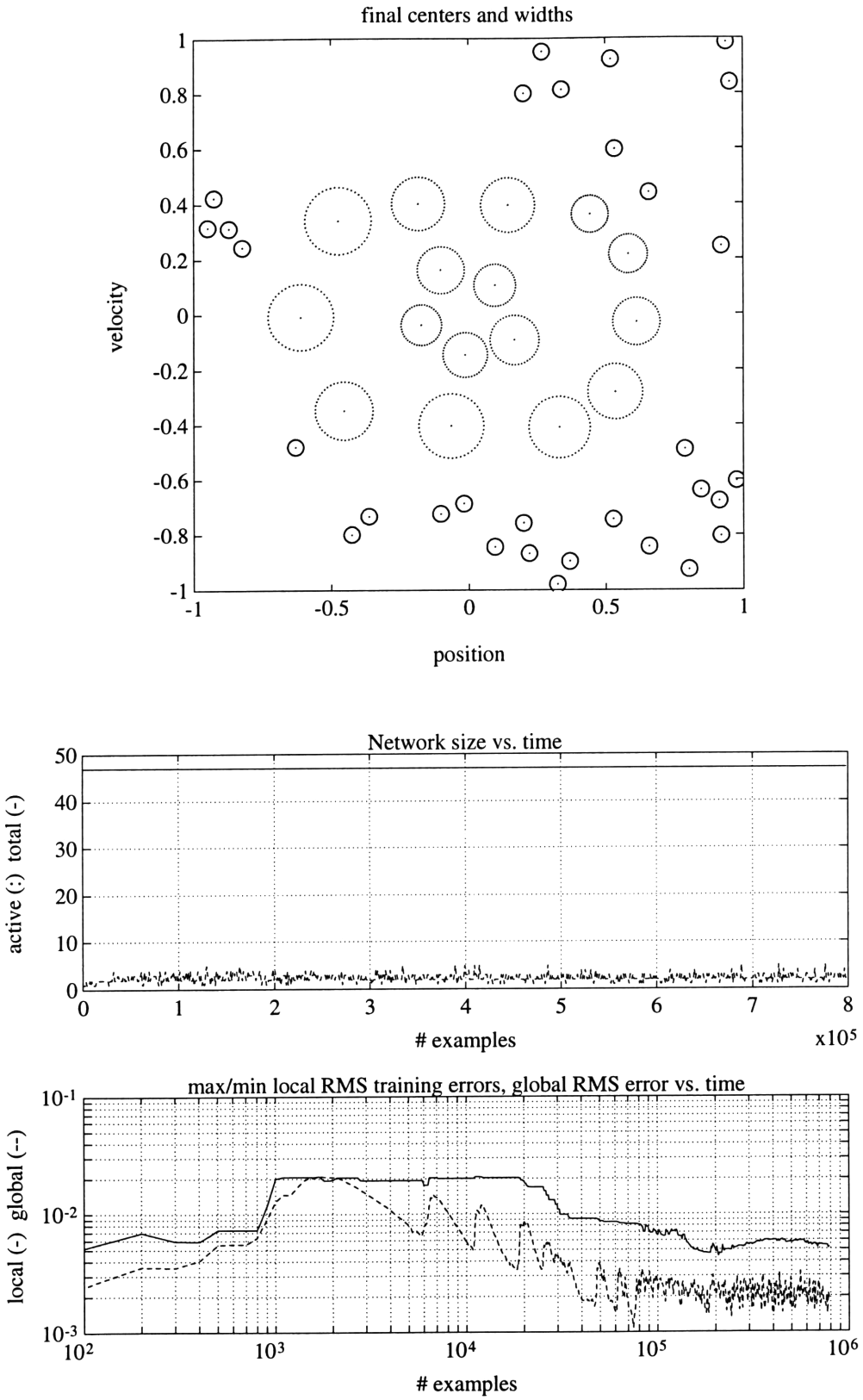


Figure 5.7. Preallocated units/No splitting

Fully supervised training algorithm (Figure 5.8)

A network was initialized with 49 identical Gaussian units arranged in a 7x7 grid to fill the envelope [–1,1] x [–1,1]; width was set equal to one-half the distance between adjacent centers. Centers and widths were trained by making adjustments down the gradient of the error (see Chapter 3); neither error nor distribution weighting were used. Unit splitting was disabled.

After undergoing the same amount of training as the benchmark network with approximately the same number of units, this network failed to achieve a validation error as low (6.44×10^{-4}). Training error decreased more smoothly with less jitter, but did not fall as far. The unit centers hardly moved, and widths changed only slightly, if at all. The number of active units averaged approximately five, as compared to two or three for the benchmark run, indicating that the supervised algorithm required more computation per example than the adaptive structure algorithm. The preallocated grid of units, with its thorough covering of the envelope, was better able to capture the outlying curvature of the nonlinear component of the target than was the benchmark; its mapping was also smoother (Figure 5.9). However, its crude features do not match those of the target mapping as well as do those of the adaptive structure benchmark.

Similar results were obtained for a network composed of 100 units arranged in a 10x10 grid; however, this time the validation error was somewhat better than the benchmark (4.45×10^{-4}).

The adaptive structure hybrid algorithm achieved lower error from a network using a comparable number of units as the supervised algorithm, but required less computation time and minimal preconfiguring of the network units. No training time appears to have been saved by the supervised algorithm by beginning training with a complete set of units;

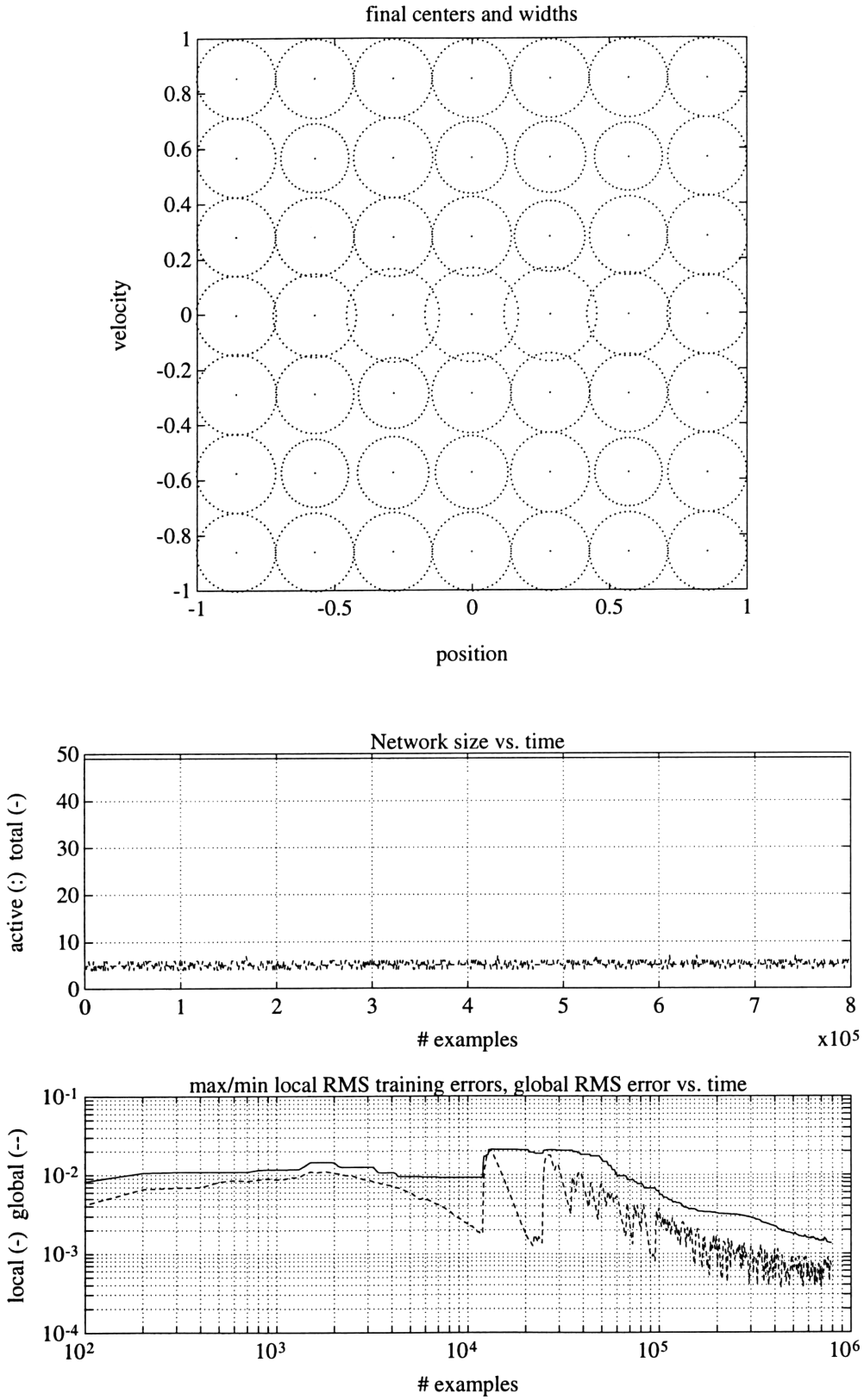


Figure 5.8. Supervised algorithm: 49 units

rather, initializing the structure in such a way seems to have biased the algorithm towards a suboptimal configuration in which many units lie away from the training inputs and thus do not contribute significantly to the solution. Starting with a finer grid of units gave better results, but with a significant increase in network size.

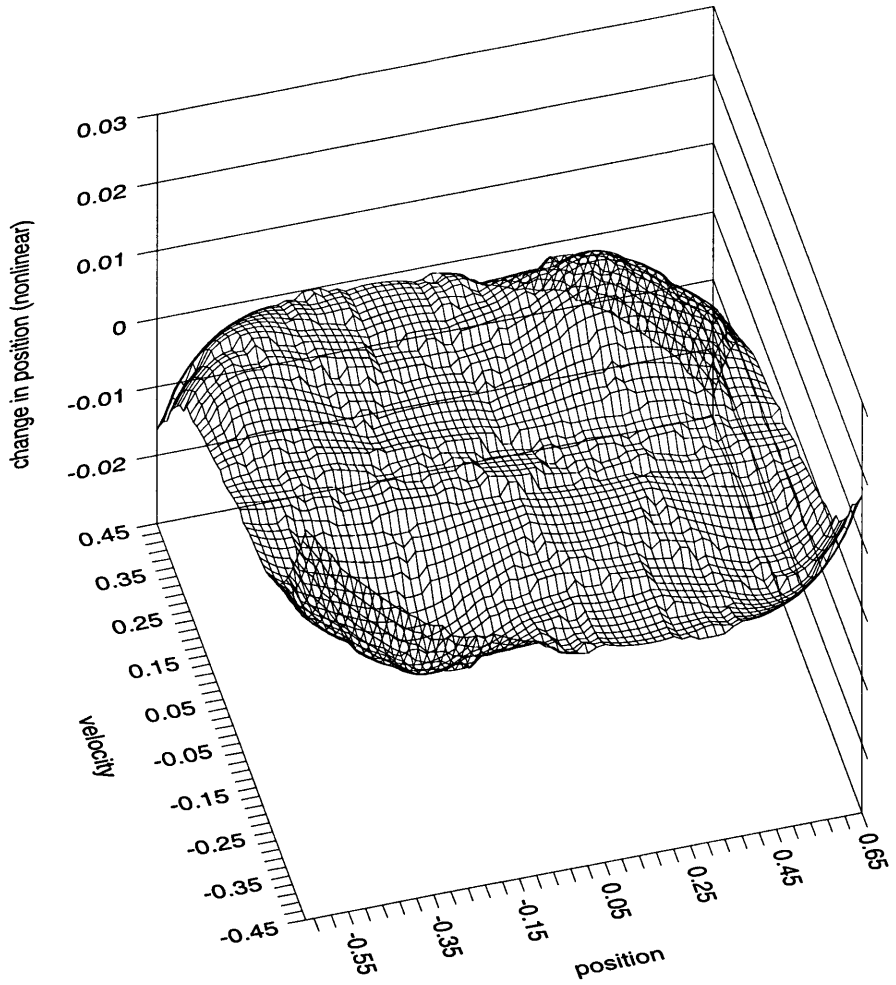


Figure 5.9. Final network mapping after fully supervised training of 49 preallocated units. Maximum/minimum values are approximately ±0.027.

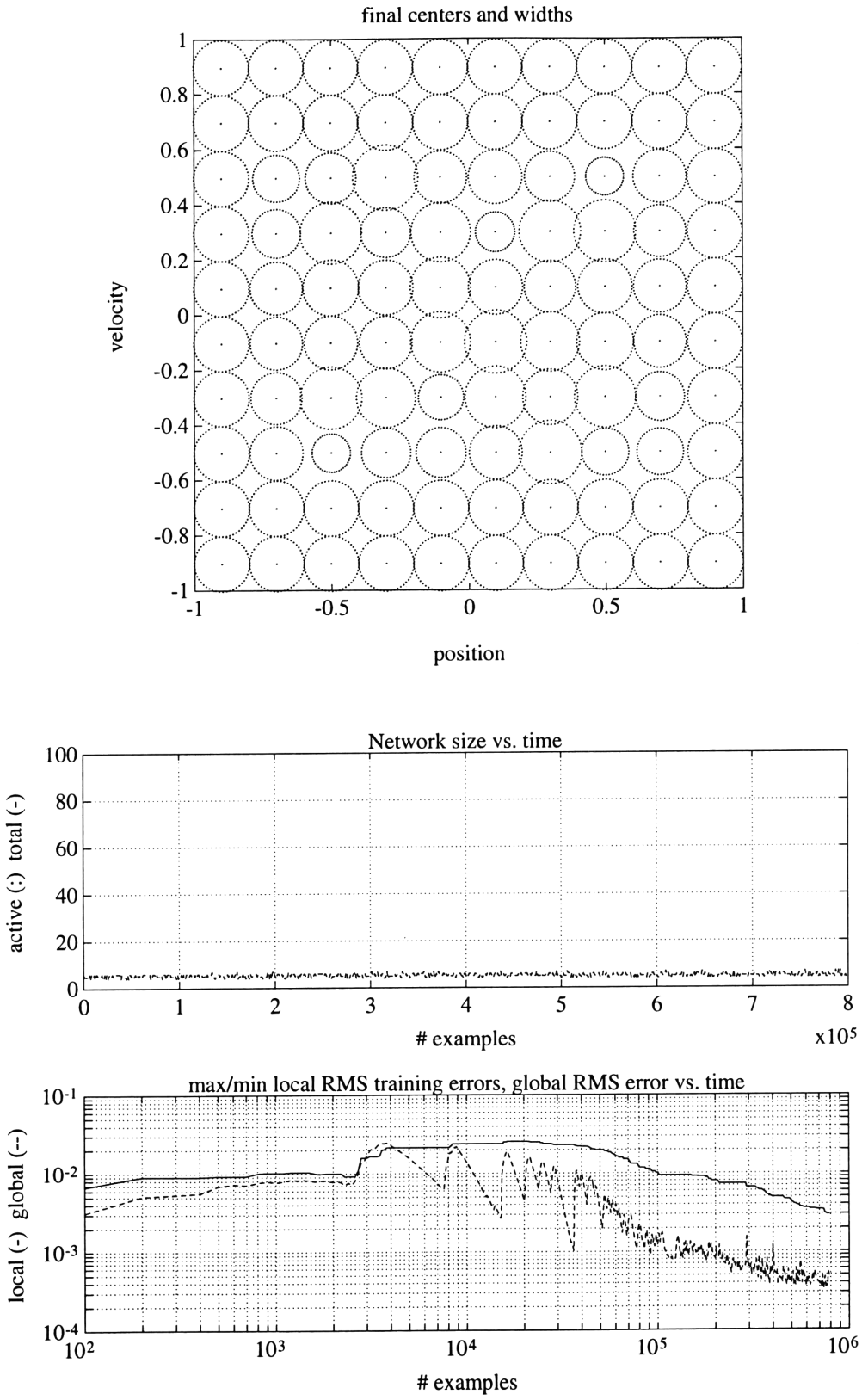


Figure 5.10. Supervised algorithm: 100 units

No error weighting (Figure 5.11)

The absence of error weighting resulted in a network with significantly more units than the benchmark run (73), and smaller validation error (3.94×10^{-4}). It is unclear whether the decrease in validation error accompanying the larger number of units was a net improvement, i.e. whether the resulting network is a more efficient approximation in terms of resources used.

The number of units along the outer limit cycle was slightly greater than in the benchmark, whereas there were almost four times as many units allocated in the inner region without error weighting than with it. The simplest explanation for the difference in unit configuration is that errors generated by inputs in the inner region were low enough for error weighting to effectively reduce the learning rate there; with the elimination of error weighting, the nominal maximum learning rate was used, leading to the creation of more units.

No distribution weighting (Figure 5.12)

Results were very similar to those of the benchmark. Validation error was slightly higher (5.01×10^{-4}) and network size slightly lower (45 units). Distribution weighting seems to have speeded convergence marginally, but the difference is not significant enough to warrant a strong conclusion.

No width reduction during split (Figure 5.13)

The pair of units resulting from a split are identical to the parent—unit widths are not reduced as in the benchmark run. The effect is quite pronounced: whereas the size (51 units) and configuration of the network is very similar to the benchmark, validation error (6.40×10^{-4}) is significantly higher.

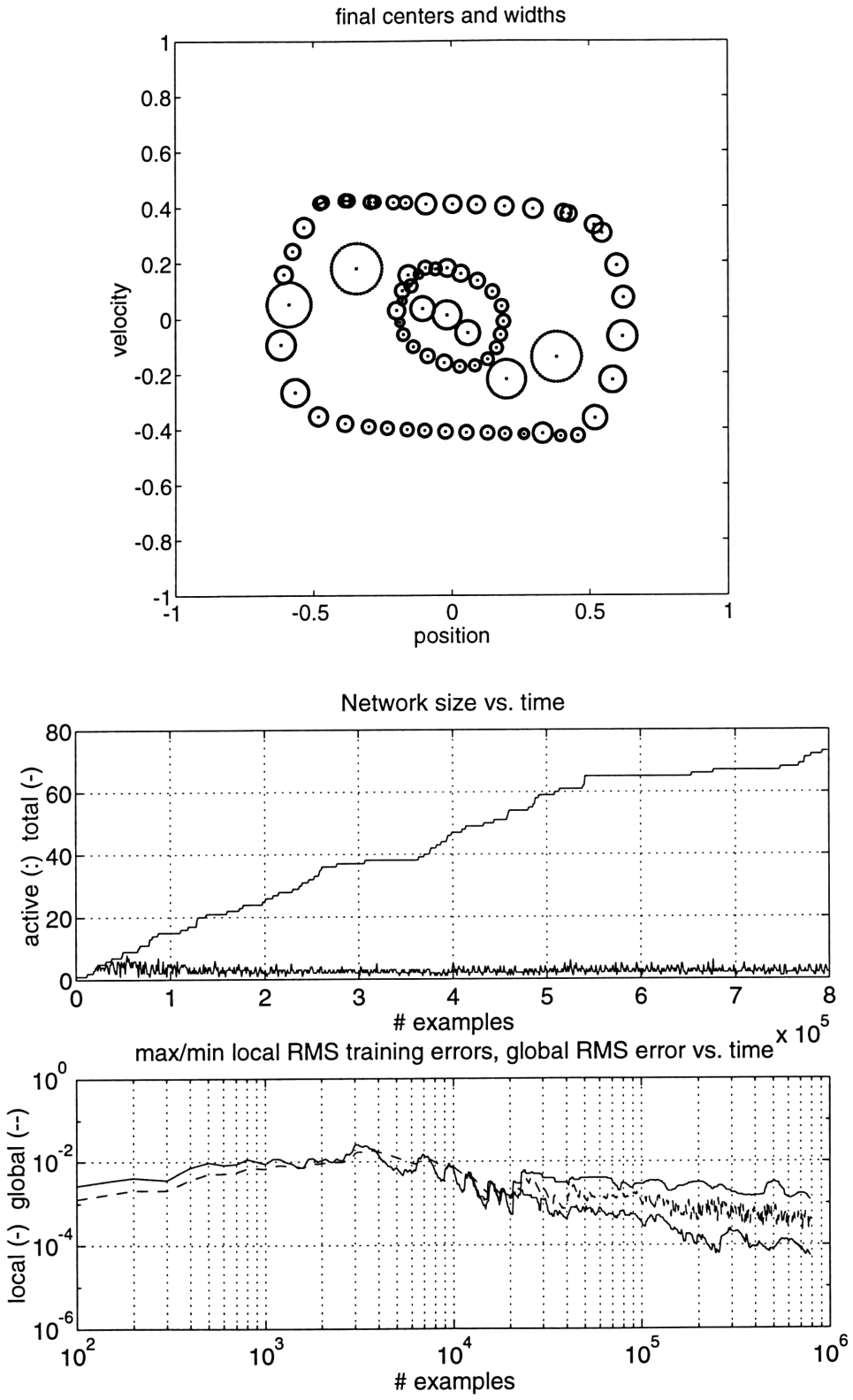


Figure 5.11. No error weighting

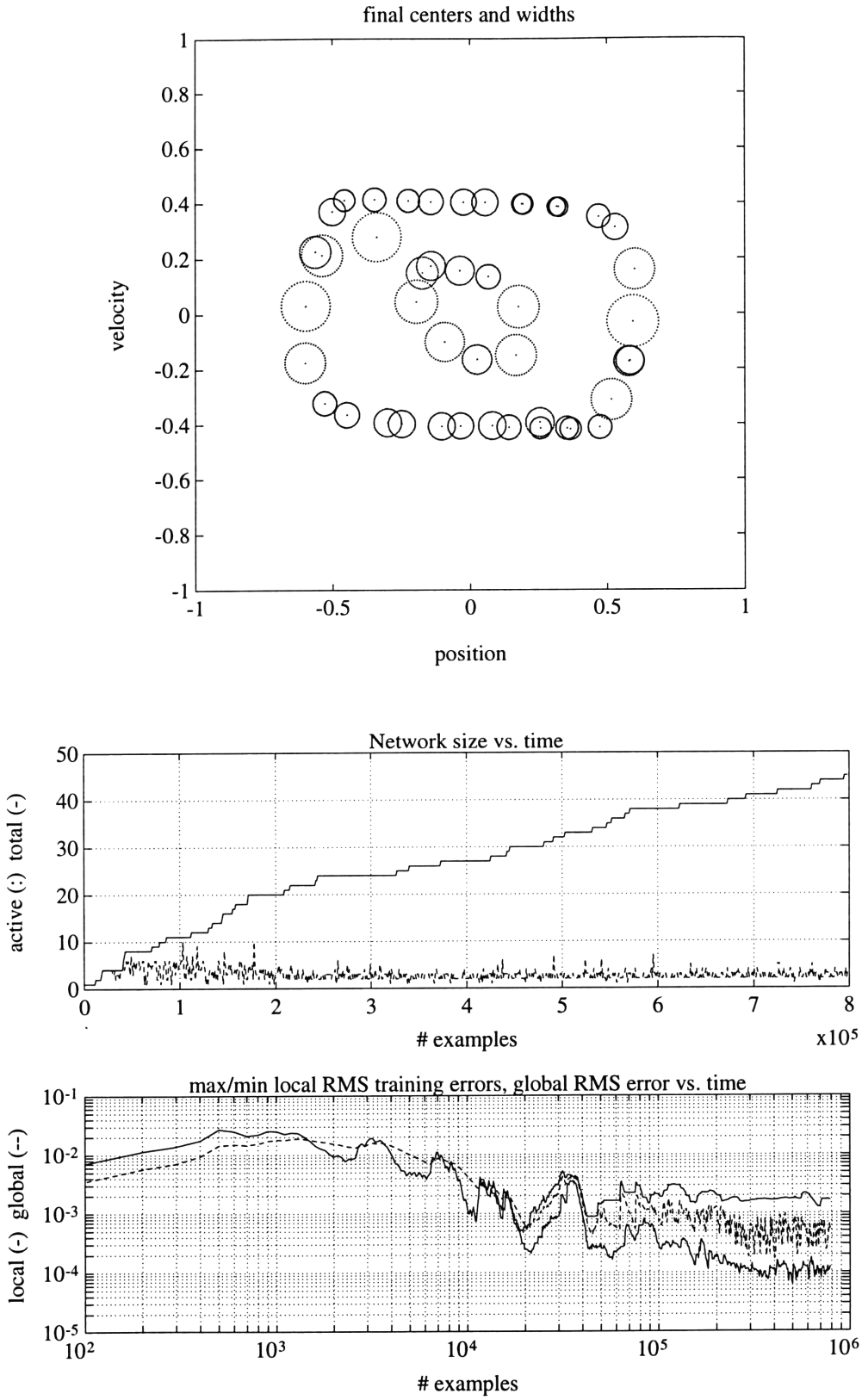


Figure 5.12. No distribution weighting

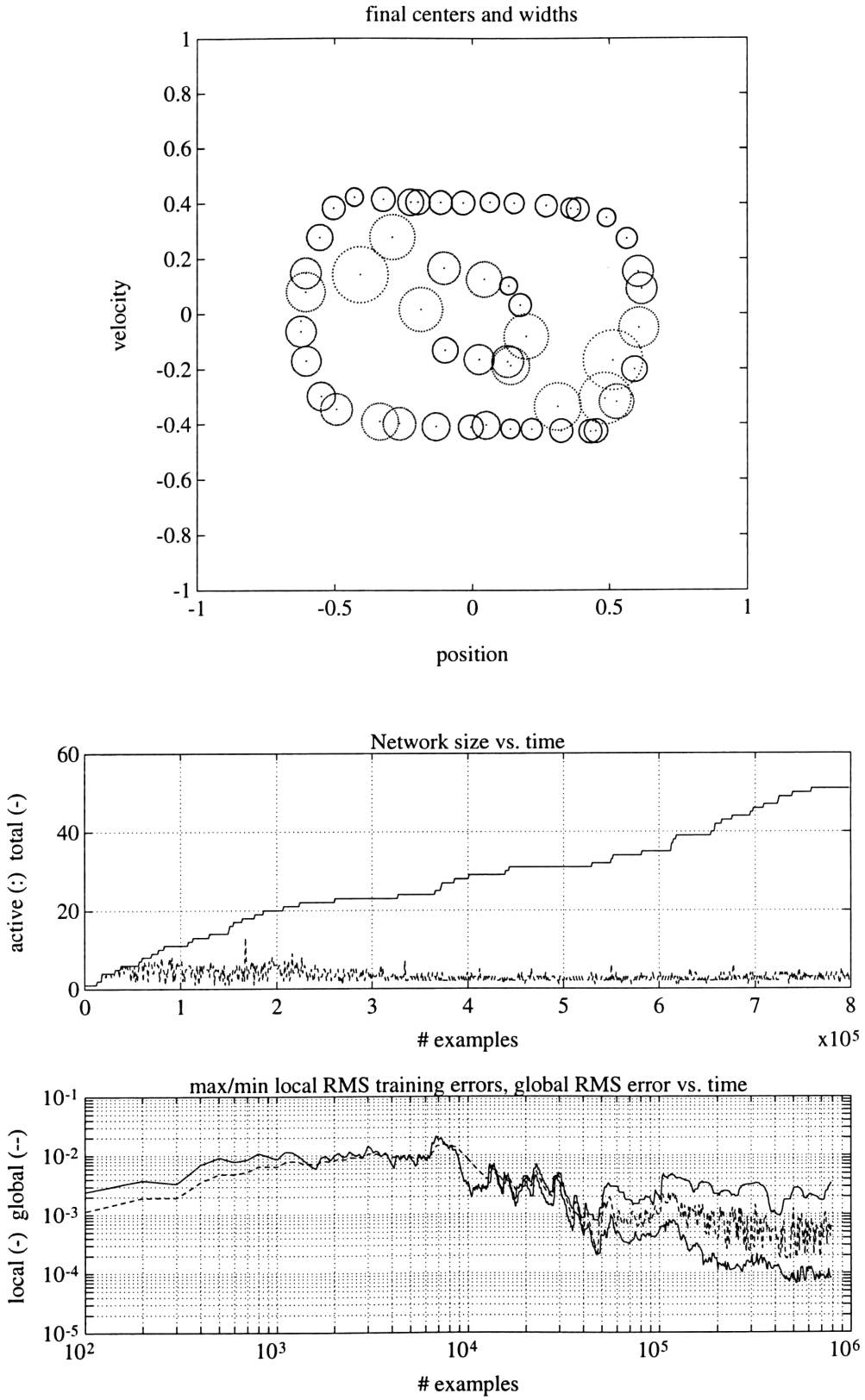


Figure 5.13. No width reduction on split

Width reduction appears to cause a less drastic modification to the network than simply duplicating the parent unit. Whenever a unit splits, some transient error should be expected; reducing the widths of the new units eliminates some of that transient error.

5.2.2.3 Training parameters

Larger pseudo-batch size/Extended training time (Figure 5.14)

Doubling the size of the pseudo-batch is equivalent to halving the learning rates for all adjustable parameters. In order to do a more meaningful comparison, the training time (i.e., number of examples) was also doubled.

The network achieved lower validation error (4.29×10^{-4}) with a network of similar size (48 units) and configuration of units. More gradual training has yielded more efficient learning in terms of network size vs. final validation error.

Smaller basis function relative learning rate (Figure 5.15)

The learning rate for basis function parameters was reduced from five times to twice the influence learning rate. The results were similar to those obtained by eliminating error weighting, but worse: validation error was higher (7.39×10^{-4}); number of units was greater (65); and network configuration was similar to the benchmark, except that there were three times as many units in the region inside the outer limit cycle.

As with the elimination of error weighting, the increased number of units is probably the result of higher influence function learning rates; in this case, the lower basis function learning rate caused an increase in local errors, from which error weighting in turn increased the influence function learning rates.

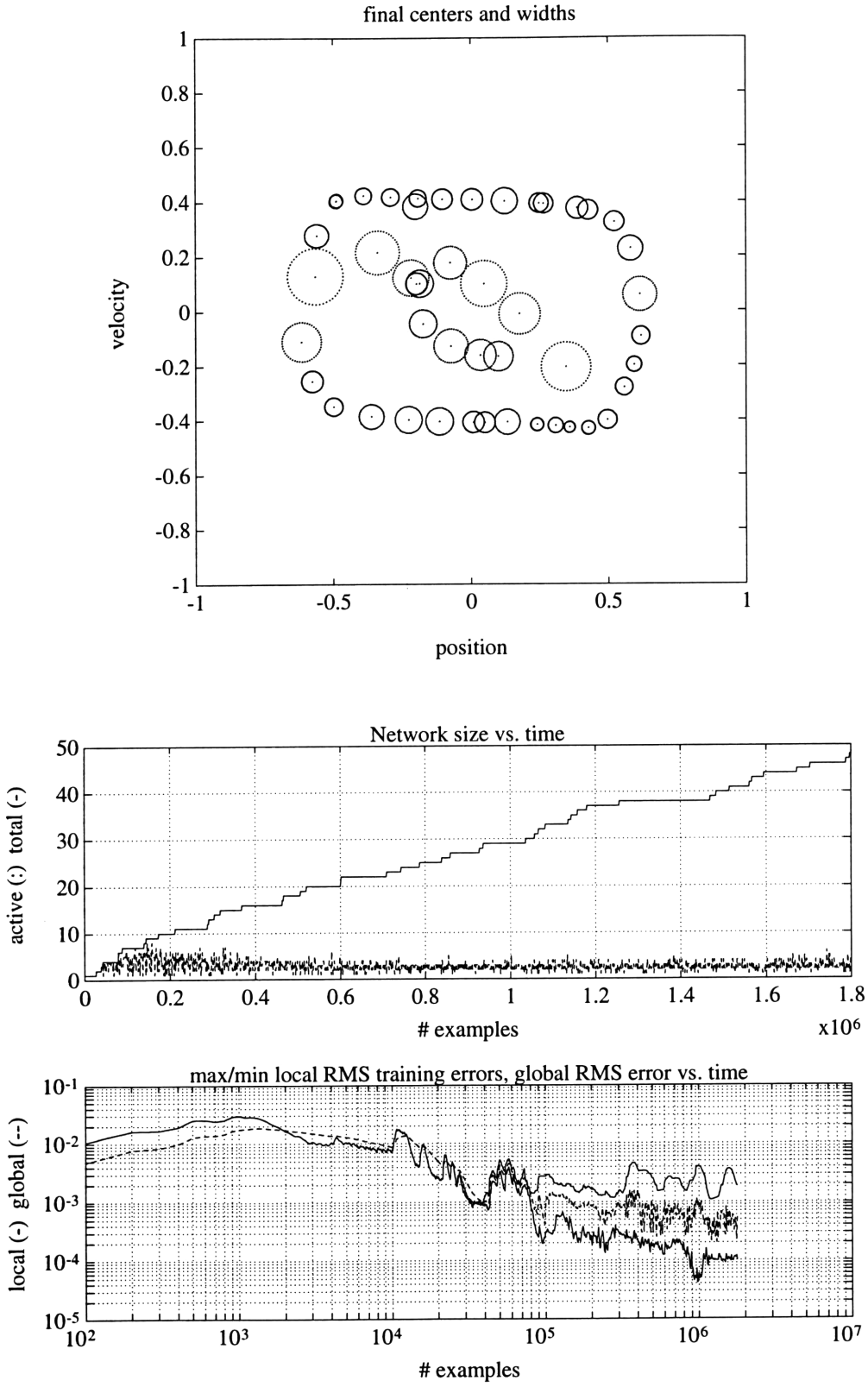


Figure 5.14. Increased pseudo-batch size

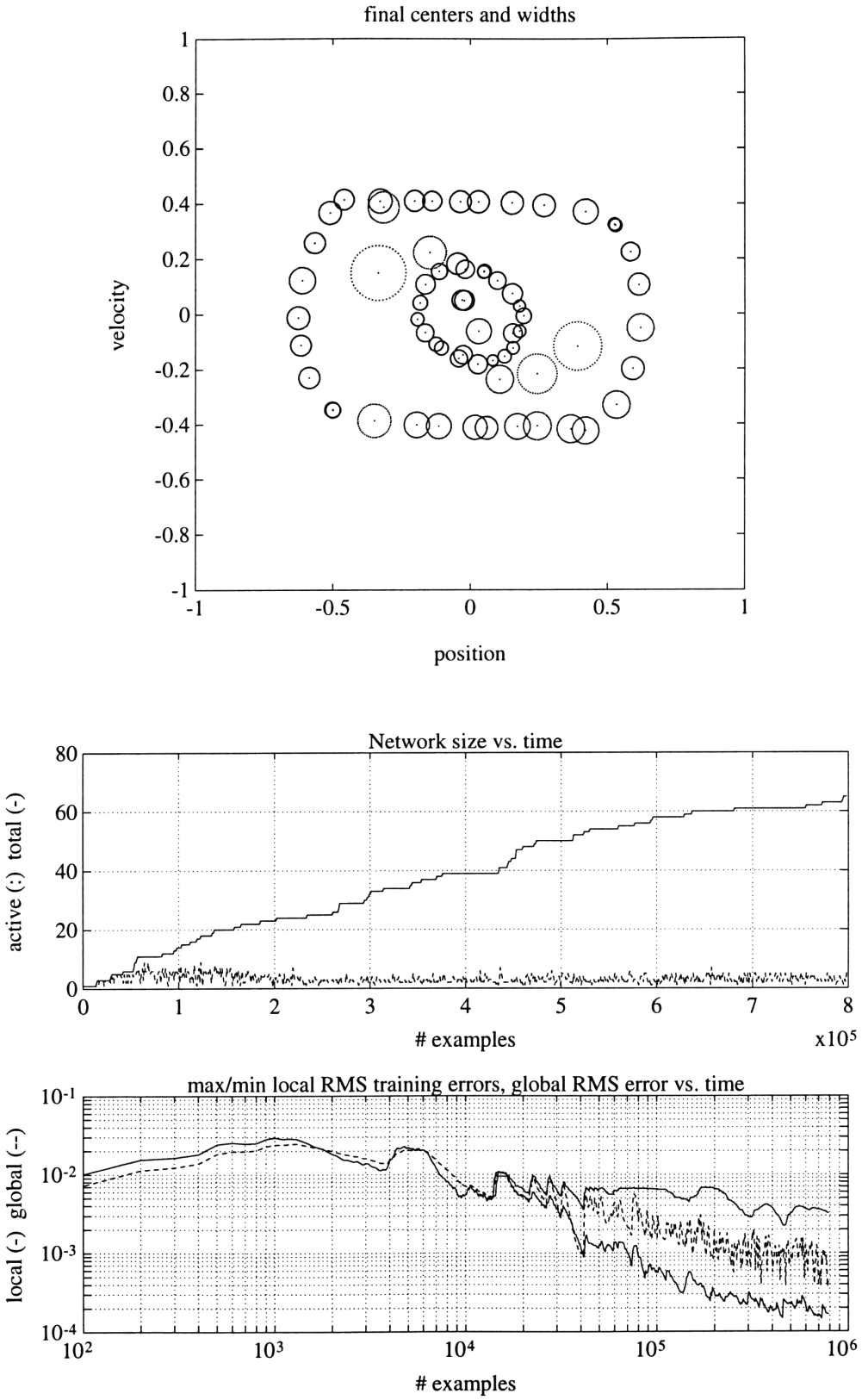


Figure 5.15. Smaller basis function relative learning rate

Stricter convergence parameter threshold (Figure 5.16)

Decreasing the convergence threshold served to decrease the number of units allocated at the end of training, but since the resulting validation error was also higher (5.64×10^{-4}) it is unclear whether anything was gained. Certainly training was slowed, but unless a much more efficient network is eventually obtained, the previous threshold value is preferable.

5.2.2.4 Training conditions

Additive noise at inputs and targets (Figures 5.17, 5.18, 5.19)

Three networks were trained, where random zero-mean training noise was added to both inputs and targets: the standard deviation of the input noise was approximately 10% of the envelope width, or 0.2; that of the target noise was approximately 10% of the maximum target value, or 0.0045. The only difference among the networks was pseudo-batch size: 1 trajectory, 5 trajectories, and 10 trajectories, respectively. The networks were evaluated using noise-free examples.

Since the training noise is random and thus unlearnable, it dominates the plot of training error and maintains it at a value slightly higher than the standard deviation of the target noise. The validation error (1.86×10^{-3}, 1.10×10^{-3}, 1.08×10^{-3}, respectively), however, is free of the extra error introduced by noise and is therefore much less than the training error.

The addition of noise has led to an increased unit creation rate (final number = 131 units), which indicates that the splitting criteria are being satisfied after shorter periods of time. This in turn suggests that the unit centers are behaving sooner as if they have converged, i.e., center updates are cancelling to a greater degree earlier in the lifetime of a unit. This makes sense when one considers that noise added to the training examples should

cause successive parameter values to become more sporadic and less directed, increasing the likelihood that centers appear to have converged, and allowing the unit to split sooner.

Decreasing the learning rate by increasing the pseudo-batch size yields a roughly proportional decrease in the number of units created (24 units and 14 units), but the validation errors achieved with the smaller networks are significantly lower. Since the noise is zero-mean, and therefore cancels on average, it is possible to learn at least a course representation of the target mapping at some scale; but the greater the variance of the noise, the more pronounced the training jitter in the parameter values, and thus the less detail that can be distinguished and eventually learned in the target. In short, noise obscures the finer details of the target. But lowering the learning rate decreases the noisy parameter jitter, enabling the network to converge to a more precise, and more "correct", solution.

Despite the fact that unit creation and basis function training are sensitive to noise, center and width training are much more robust. The algorithm produces unit configurations similar to those obtained during noise-free runs. This is quite an improvement over the expected behavior of a unit creation algorithm which makes structure modifications based on the performance of the network on a single example: every apparently outlying example that appeared to yield high error would elicit a new unit, and the network would grow out of control. In this case, turning down the learning rate certainly would not help, and further restricting the criteria for unit creation would hamper normal network growth.

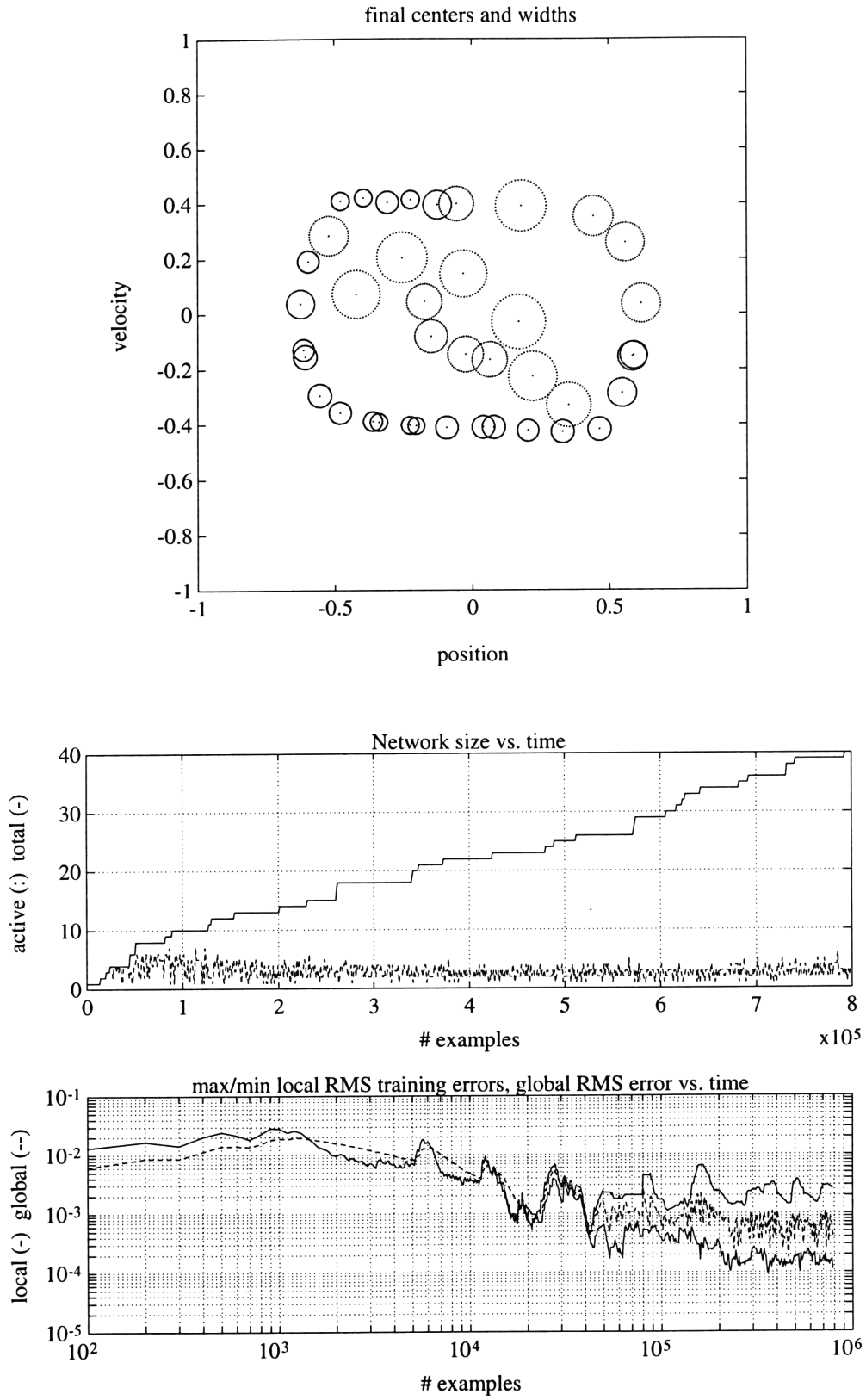


Figure 5.16. Stricter convergence threshold

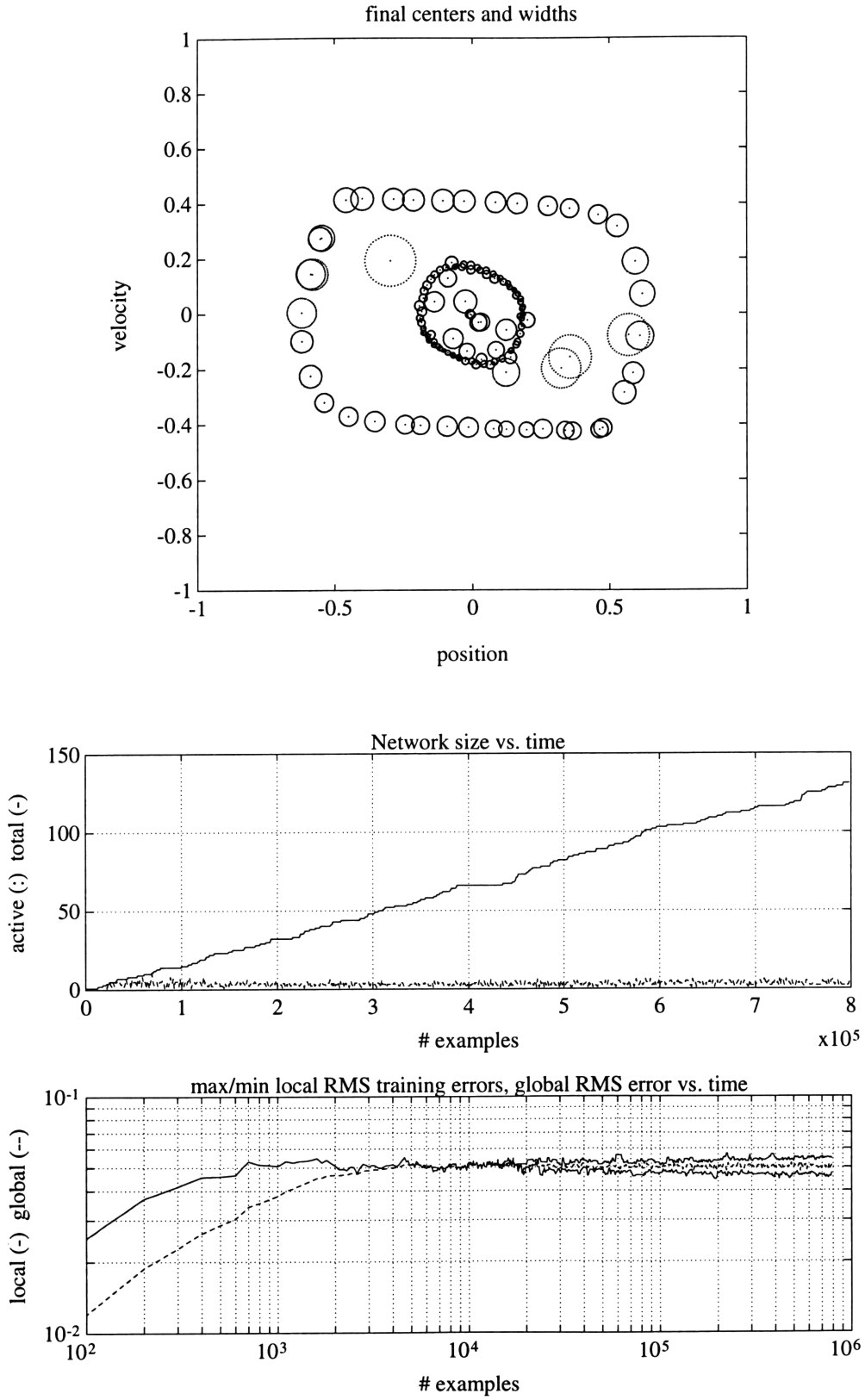


Figure 5.17. Training noise: pseudo-batch size = 1 trajectory

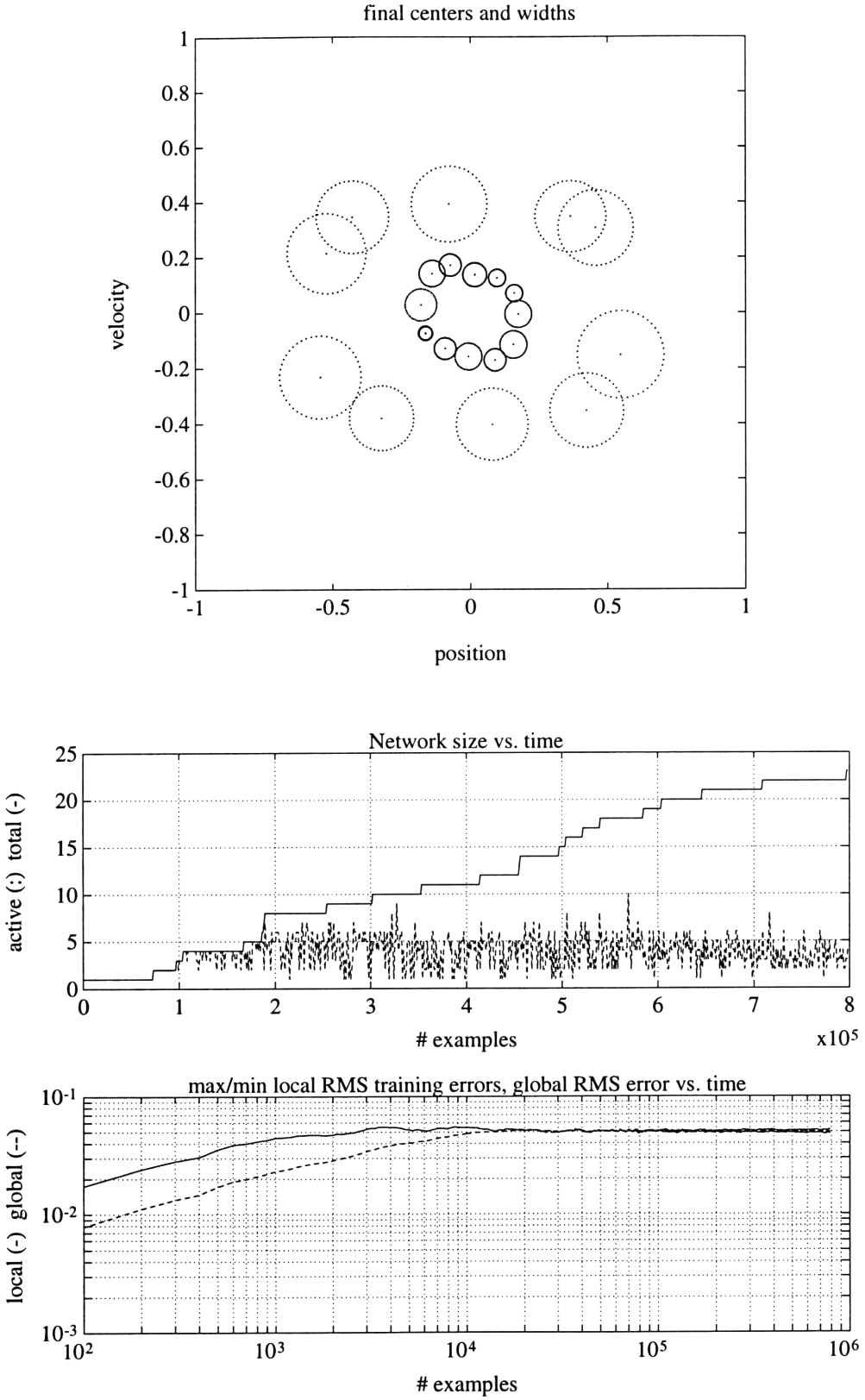


Figure 5.18. Training noise: pseudo-batch size = 5 trajectories

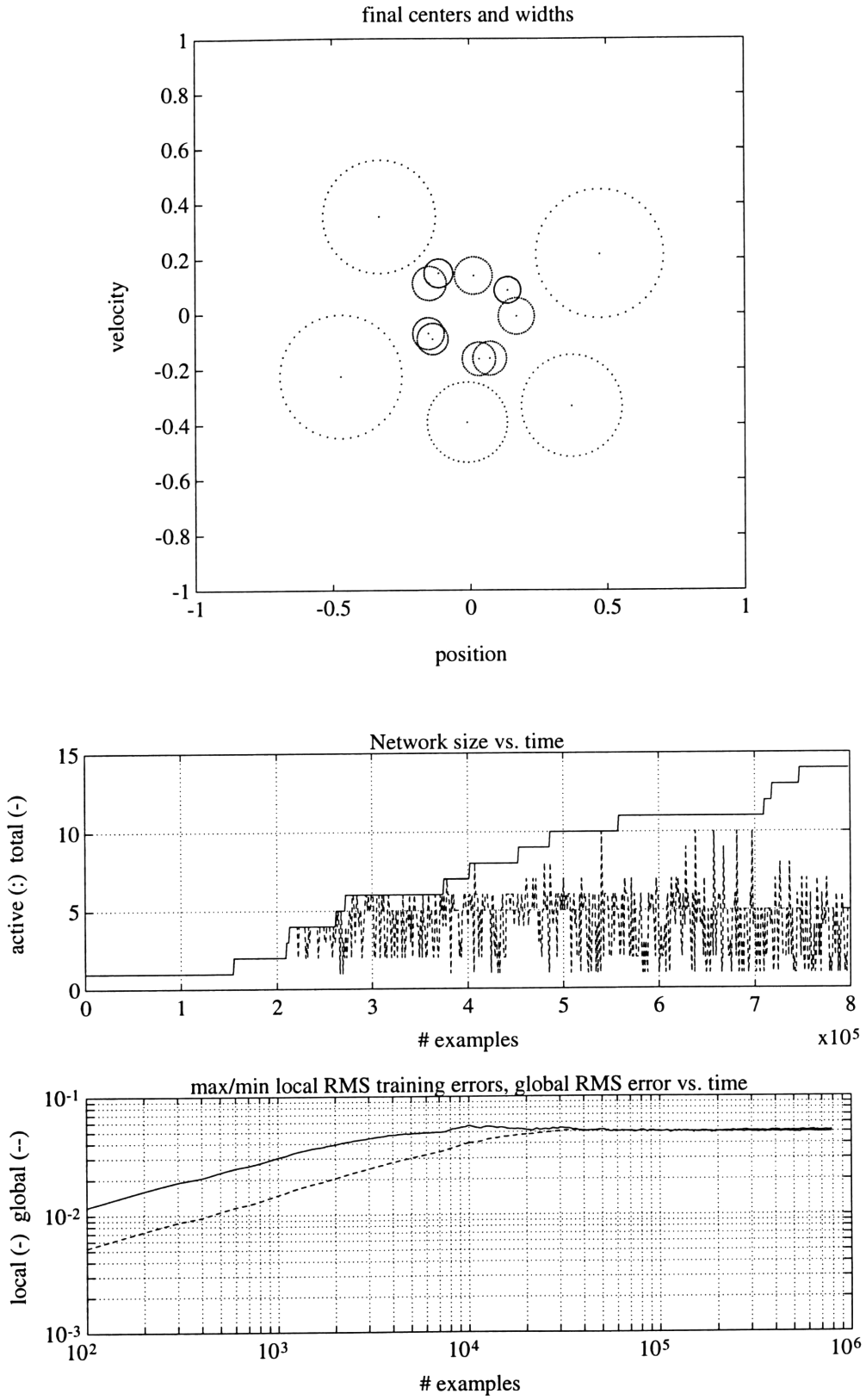


Figure 5.19. Training noise: pseudo-batch size = 10 trajectories

Shorter trajectories (Figure 5.20)

This run illustrates the effect of training distribution on the network configuration. By cutting the length of the state-space trajectories in half, the training distribution is biased away from the limit cycles, in favor of other areas of space.

In the final network configuration, the outer limit cycle is still very well defined, but units in the inner region are distributed more evenly—the inner limit cycle is barely distinguishable. The network achieves a low validation error (3.80×10^{-4}), but at the expense of many additional units (final size = 96), which are required to construct better local approximations in those areas that are now more heavily represented in the training data.

5.2.3 Summary of Aeroelastic Oscillator Results

A network composed of B/I units having affine basis functions and normalized Gaussian influence functions was trained to predict future positions of an aeroelastic oscillator simulation, using the adaptive structure hybrid algorithm and a stream of examples generated by the free-running simulation. The first such network served as a benchmark to which the slightly modified training of subsequent networks could be compared and evaluated.

Gaussian influence functions resulted in a somewhat higher validation error with more units than inverse square influence functions; however, with training restricted to only the most influential units, the greater localization (i.e., decay away from center) of Gaussian influences yielded fewer units trained and therefore significantly less computation.

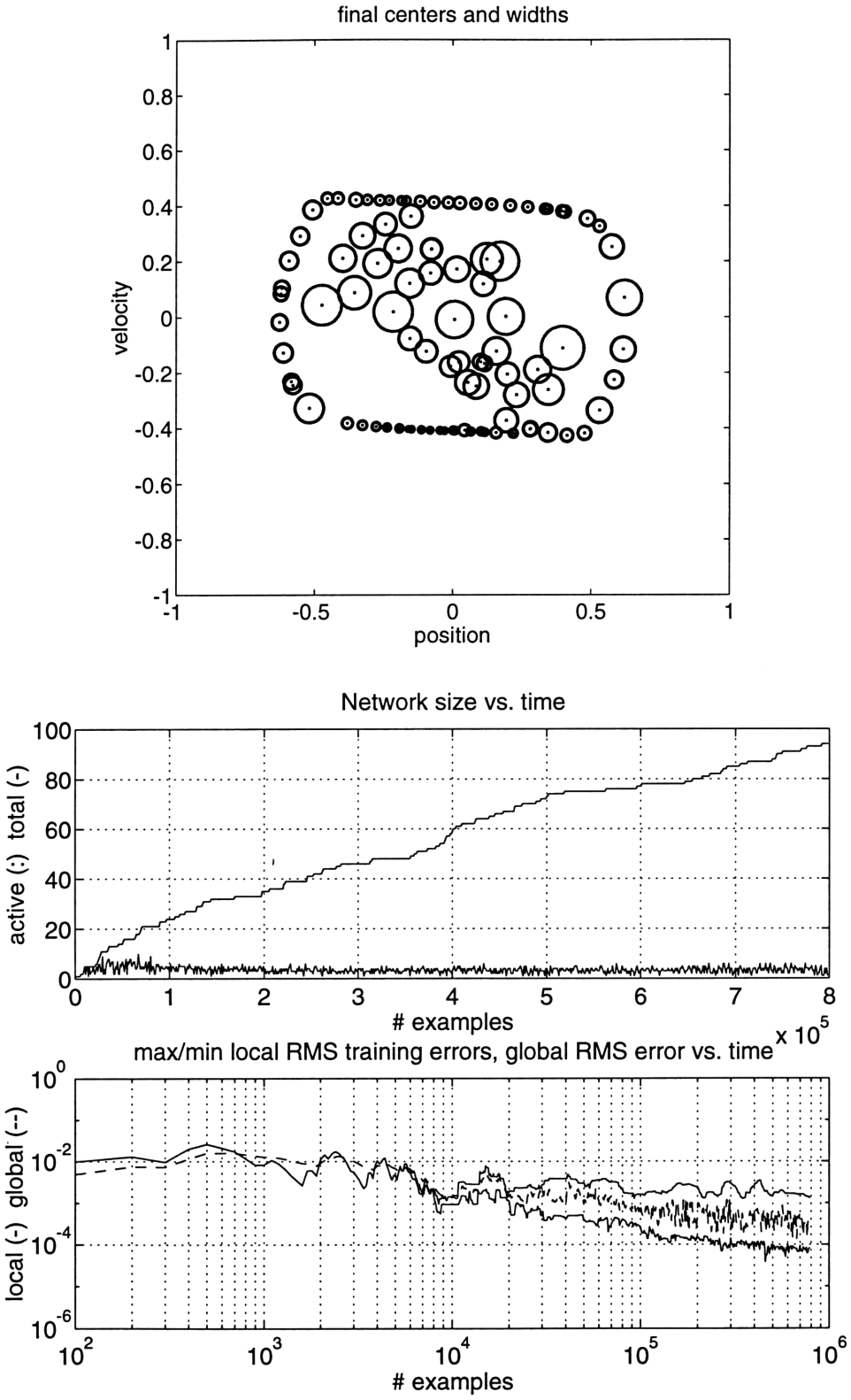


Figure 5.20. Shorter training trajectories

Allocating all network units up front (with random centers and zero basis functions) and then training without subsequent splitting was not sufficient to achieve validation error comparable to the benchmark—some units lay too far away from the training examples to be utilized efficiently. A similar phenomenon was observed when networks were initialized with square grids of units and then trained using a fully supervised algorithm; in this case, the unit configurations hardly changed, and it was necessary to increase the number of units allocated to decrease final error. However, the networks with initial grid configurations were better able to capture features of the target mapping in regions where the distribution of training examples was sparse.

Training without error weighting resulted in increased proliferation of units overall, particularly along the inner limit cycle, with a large increase in final network size and a decrease in validation error. It is unclear whether the final network is more efficient than the benchmark in terms of size vs. error, but in any case convergence was faster. Distribution weighting seemed to speed convergence somewhat, but the effects were minimal. Reduction of unit widths during splitting led to lower validation error after the same amount of training with a modest increase in network size.

Lower error was achieved when the pseudo-batch size was larger (and thus the learning rates smaller) and training time longer. A large basis function learning rate relative to the influence function learning rate performed better than a smaller rate. A stricter convergence threshold slowed learning without any apparent benefit.

In the presence of noise, the adaptive structure algorithm was able to place units in the more heavily trained areas, but overproliferation of units and high error were problems. Increasing the pseudo-batch size offered more

robustness to noise, enabling the network to capture more detail in the target mapping, but with increased training time.

Training the network with shorter trajectories significantly changed the distribution of the examples, decreasing the probability that an example lay along a limit cycle. Consequently, the network configuration created by the adaptive structure algorithm shifted to reflect this change in distribution.

Final network sizes and validation errors are summarized in Figure 5.21.

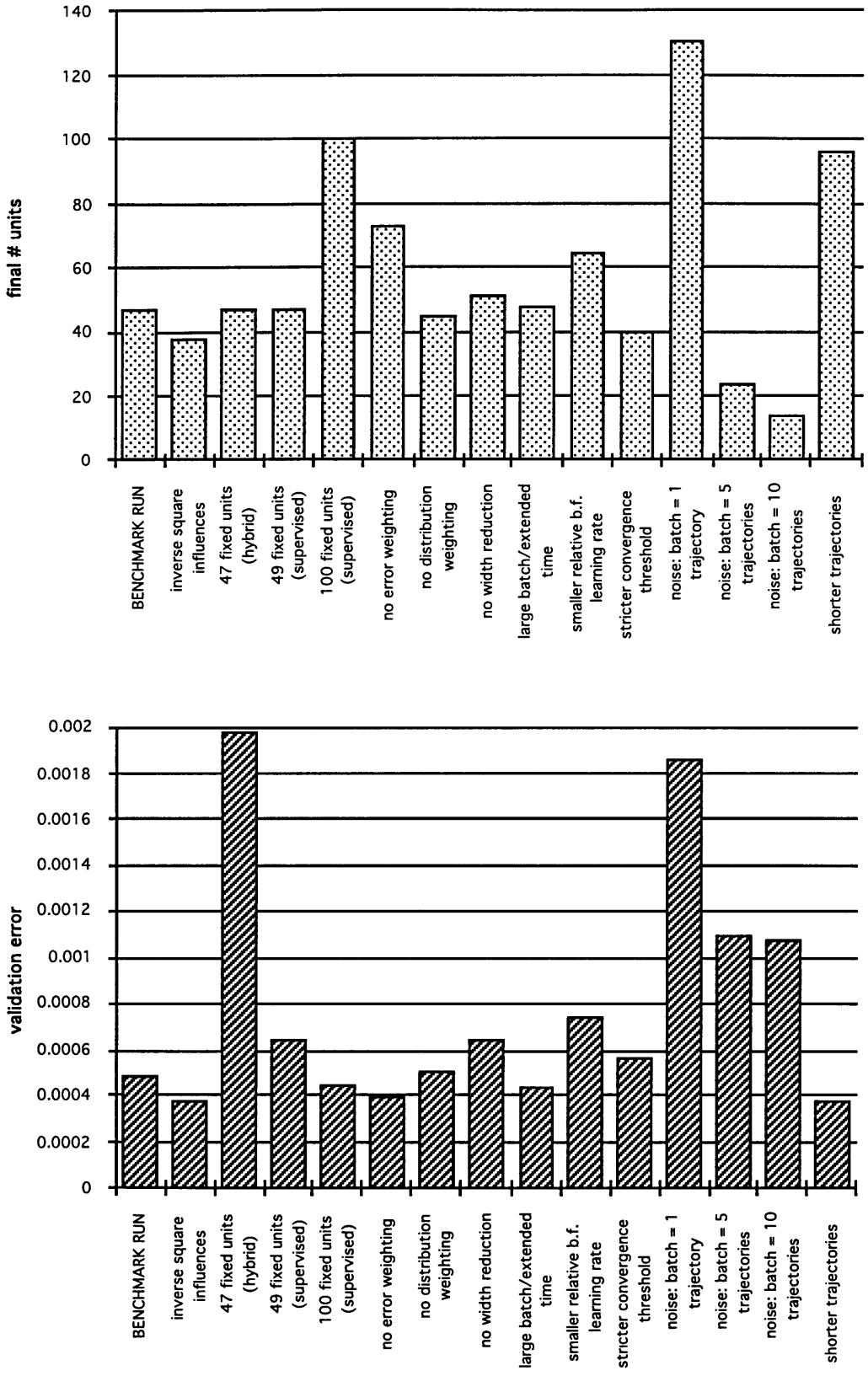


Figure 5.21. Final Sizes and Errors of Networks

6 Prediction of a Chaotic Time Series

We now measure the performance of the adaptive structure hybrid algorithm at training a B/I network to predict future values of a chaotic time series generated by the Mackey-Glass equation (see Appendix C) from a set of past values. Results obtained here are then compared with those from a previous such attempt (reported in the literature) that uses a similar, fixed-structure B/I network and an off-line hybrid training algorithm.

6.1 Previous Work

Prediction of time series generated by the Mackey-Glass equation has been used to measure the performance of several other algorithms [5, 15, 20, 30]. The approximation structure selected and evaluated by Stokbro et al is exactly the B/I network (affine basis functions, normalized Gaussian influence functions) that appears in this thesis; for this reason, the results of the adaptive structure hybrid algorithm are compared with those of Stokbro.

The parameter training algorithm of Stokbro is similar to the hybrid algorithm used here, in that influence function training is unsupervised and decoupled from basis function training. As with Moody and Darken, the network is trained on a fixed set of examples, from which centers and widths are determined prior to basis function training. However, no "training" is done per se; rather, clusters and their centers are identified explicitly, and widths computed to achieve a suitable overlap among N adjacent units.

Stokbro continues to exploit the off-line nature of the problem as fully as possible. Once influence parameters have been computed, basis functions are initialized to give best local fits to examples that lie within the

corresponding clusters. Finally, genuine "training" of basis function parameters occurs, using a supervised gradient algorithm that minimizes a batch cost that has been revised to reduce redundant computations.

In Table 6.1, the algorithm of Stokbro is compared to the adaptive structure hybrid algorithm developed in this thesis.

	Thesis	Stokbro
Network architecture	linear basis functions normalized Gaussian influence functions number of units varies to meet target error	linear basis functions normalized Gaussian influence functions number of units chosen based on number of training examples
Training examples	stream, generated on-line	batch, generated off-line
Center determination	trained on-line to minimize cumulative squared distance from examples to nearest center	computed off-line to minimize cumulative squared distance from examples to nearest center
Width determination	trained on-line to reflect "typical" distance of a center from examples in region in which it predominates	computed off-line for each center based on proximities of N nearest centers
Basis function parameter determination	supervised gradient updates to minimize incremental error	supervised gradient updates to minimize batch error
Network initialization	center of first unit initialized randomly within training domain width set based on size of training domain basis function parameters set to zero subsequent units inherit parameters from parent units	calculation of all centers and widths basis functions set to yield best local fit

Table 6.1. Comparison of Stokbro, Thesis Algorithms

The off-line results obtained by Stokbro should serve as a reasonable example of the performance attainable by this particular type of B/I network on this particular problem, despite the major differences in approach from the adaptive structure hybrid algorithm.

6.2 Prediction Results

The adaptive structure hybrid algorithm is used in two different scenarios to train a B/I network (affine basis functions, normalized Gaussian influence functions) to predict future values of the Mackey-Glass chaotic time series. First, training is performed off-line by iterating through a fixed set of examples generated prior to training. Second, the network is trained on-line using a stream of examples generated by the free-running Mackey-Glass system (see Section C.3 and Figure 5.1). Prediction error in both cases is measured over a stream of examples from the free-running system.

The resulting networks are compared to those of Stokbro in terms of prediction error, the number of training iterations, the number of distinct training examples used, and the size of the final network.

6.2.1 Off-line Training

A set of 500 examples (10 quasi-periods) was generated and stored prior to training. The adaptive structure network was trained by iterating randomly through the training set a total of 1600 times. The training algorithm and parameters were the same as for the aeroelastic oscillator benchmark run, except that distribution weighting was turned off, c_{rel} = 10, and target error = 0.01. The normalized prediction error, $\overline{E}^2$ (Eq. 6.1), was measured on a new set of examples (200 quasi-periods) in the manner of Stokbro. (The evaluation set was much larger than the training set so that generalization to novel inputs could be measured.)

$$\overline{E}^2 = \frac{\left\langle \left(y_{targ}[k] - y_{net}[k] \right)^2 \right\rangle}{\left\langle \left(y_{targ}[k] - \left\langle y_{targ}[k] \right\rangle \right)^2 \right\rangle} = \frac{\text{mean squared error}}{\text{variance of target set}} \tag{6.1}$$

Referring to Figure 6.2, global training error and both maximum and minimum local error decrease fairly smoothly (compared with that observed previously for the aeroelastic oscillator). A steady proliferation of units generates a thorough covering of regions of the attractor (Figure 6.1) represented in the training set.[1] The number of active units maintains a very low relative value throughout training. When evaluated on a stream of examples, RMS validation error (i.e., unnormalized prediction error) was found to be quite a bit higher (0.0208) than the final RMS training error (0.0115), indicating mediocre generalization.

The final size of the network, as well as the training effort required to achieve the resulting normalized prediction error, was compared with a network from Stokbro that yielded similar prediction error (Table 6.2).

	Off-line Results	Stokbro Results
Normalized prediction error	0.0083	~0.007
Size of final network	216 units	25 units
Size of training set	500 examples = 10 quasi-periods	500 examples = 10 quasi-periods
Number of training examples	800,000 examples = 16,000 quasi-periods	100,000 examples = 2000 quasi-periods

Table 6.2. Results of off-line training

[1] If it appears that the overlap among units is excessive, keep in mind that the figure shows a projection of a four-dimensional space onto two dimensions; thus, units that appear to overlap in this subspace do not necessarily overlap in the full space.

It has taken the adaptive structure network roughly 8 times as many units and training examples to achieve results approximating those of Stokbro. It may be that premature proliferation of units, i.e., splitting units that are not yet fully trained, has caused significant inefficiency in the final network. Otherwise, training has proceeded nicely, giving a sensible covering of the training domain and steadily decreasing error.

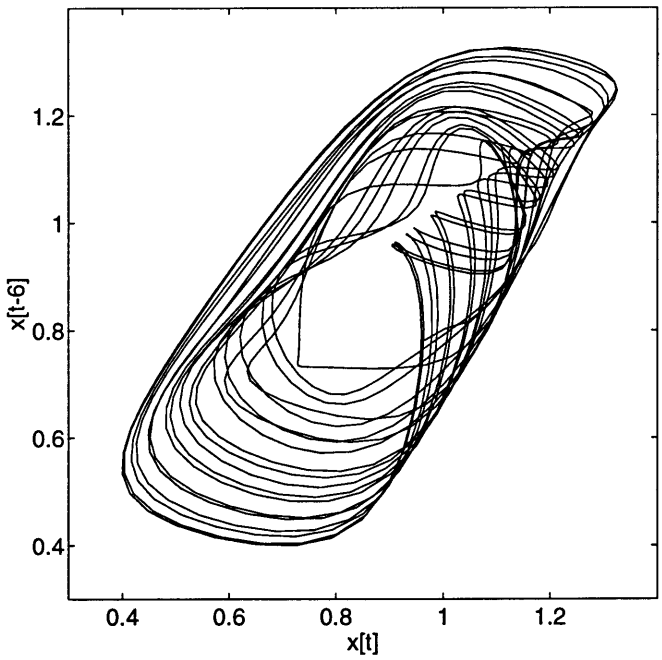


Figure 6.1. A two-dimensional Poincaré map of the time series generated by the Mackey-Glass equation. The current state is plotted vs. the state 6 time steps in the past, over a total of 10 quasi-periods.

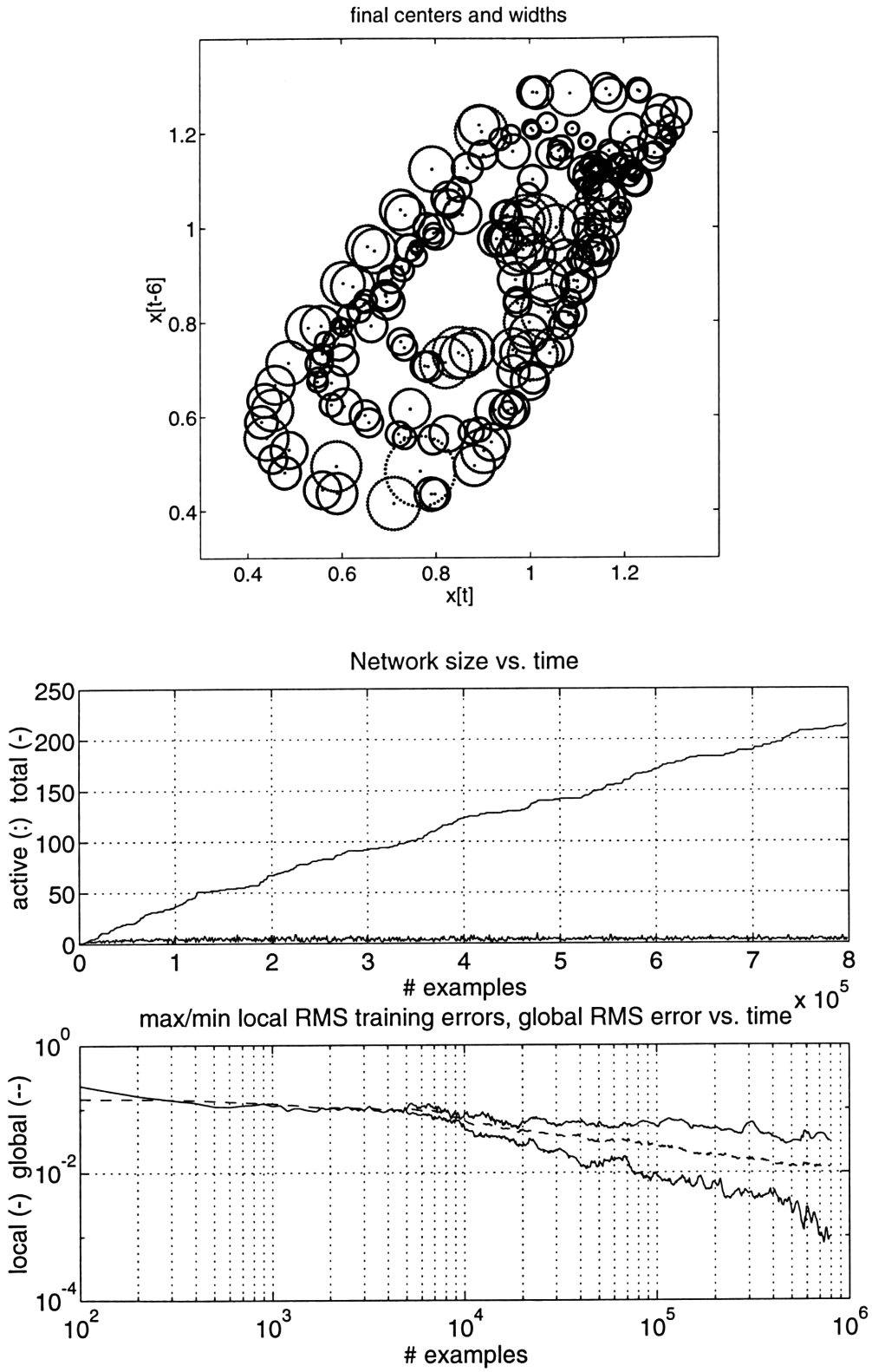


Figure 6.2. Off-line Training

6.2.2 On-line Training

Another network was trained, but this time using examples generated on-line. The total number of examples presented to the network was the same (800,000 examples = 1600 quasi-periods), but rather than repeated presentations of the same set of quasi-periods, each quasi-period was different, providing a more thorough exploration of the Mackey-Glass attractor. The convergence threshold θ_{conv} was changed to 0.1 (from 0.2 in the previous run) to possibly reduce any overproliferation of units.

On-line training has made it possible to construct a more thorough covering of the attractor than was done off-line with an example set of fixed size (Figure 6.3). The quality of training as manifested by the smoothly decreasing error plots has been preserved. Prediction error was measured in the same manner as before. Generalization is excellent: RMS training error (0.0142) is practically the same as unnormalized prediction error (0.0144), which has decreased significantly from the off-line case despite a reduction in network size (159 units).

A network from Stokbro having similar normalized prediction error was chosen for comparison (Table 6.3).

	On-line Results	Stokbro Results
Normalized prediction error	0.0039	~0.0045
Size of final network	159 units	100 units
Size of training set	800,000 examples = 16,000 quasi-periods	2000 examples = 40 quasi-periods
Number of training examples	800,000 examples = 16,000 quasi-periods	400,000 examples = 8000 quasi-periods

Table 6.3. Results of on-line training

The gap in performance seems to have narrowed somewhat. A 59% increase in network size, accompanied by twice as much training, has yielded a modest decrease in prediction error, as contrasted with the previous eight-fold disparity resulting in higher prediction error.

6.3 Discussion

On-line training has produced a better predictor than off-line training performed for the same length of time. The unit configuration is more efficient (more thorough covering, fewer units) and prediction error is lower. This outcome makes sense when one considers that the set of on-line training examples is virtually identical (for a large enough set) to the set of examples eventually used to measure prediction performance. In contrast, the fixed set of off-line examples represents only a segment of possible prediction inputs; hence, the network is biased towards those examples to the detriment of performance on other examples, leading to poor generalization.

On-line training with a stricter convergence threshold significantly improved network performance relative to the Stokbro results, though this performance remained inferior. The excessive network size might be caused by a convergence threshold that is still too lax, allowing units to split before they have been fully trained, and yielding an inefficient use of network resources. Overproliferation can also account for the increased number of training examples required, since the winner-take-all influence training algorithm effectively divides up the training set among all units, requiring more examples to achieve the same level of training for a larger network.

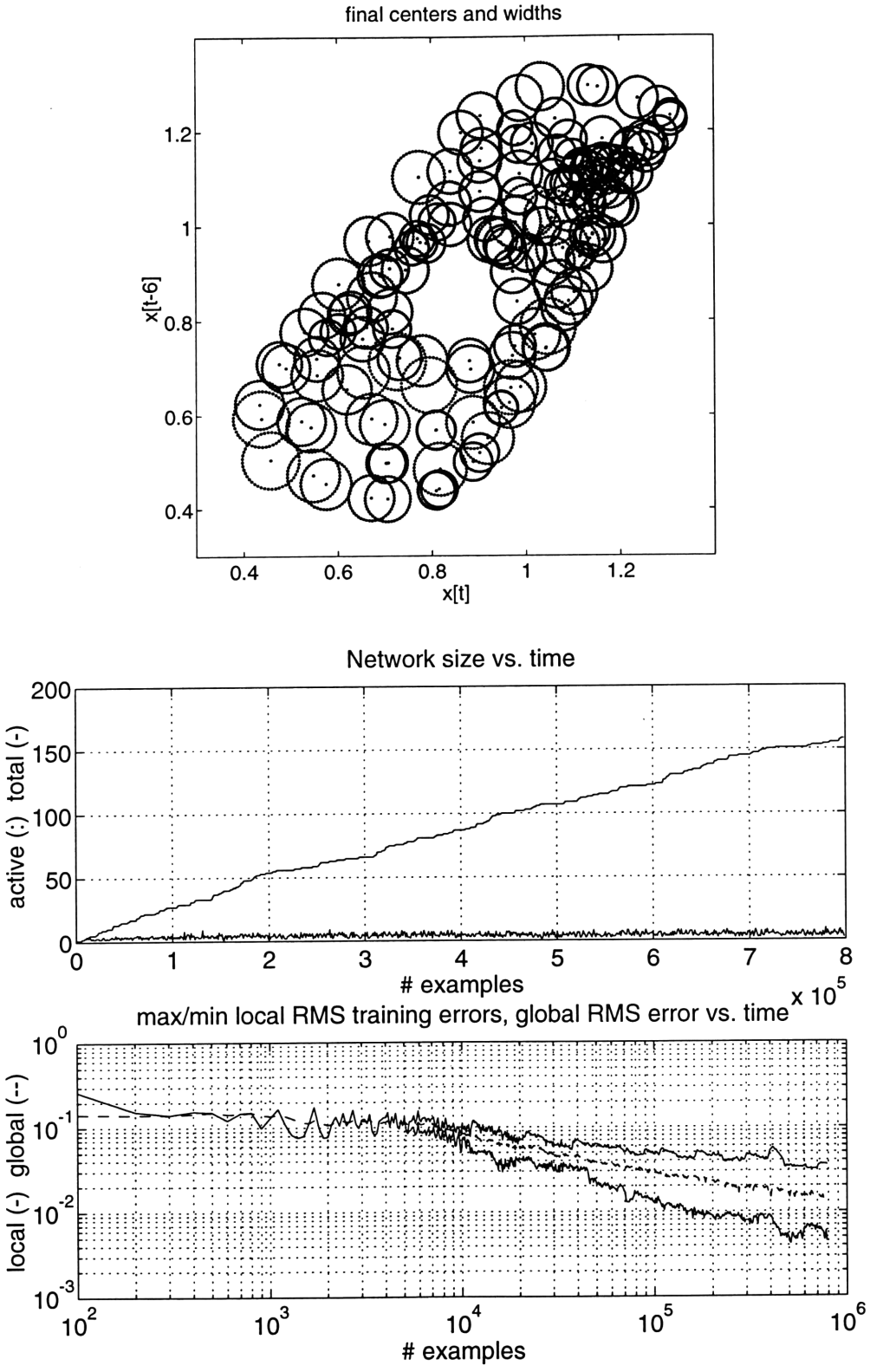


Figure 6.3. On-line Training

7 Conclusion

7.1 Overall Performance of the Network and Algorithm

The objectives of this thesis were to develop a network approximation structure capable of increasing its size in order to achieve a closer fit to on-line training examples generated in accordance with the dynamics of some unknown system, while being robust to noise, adverse ordering, and nonuniform distribution associated with the training data stream. In light of these goals, the successes and failures of this thesis are assessed below.

7.1.1 Evaluation of Results

The unsupervised center training algorithm performed well under on-line conditions, as distinct from the off-line training investigated by Moody and Darken. Unit center organization corresponded very closely with the training input distribution, even in the presence of significant amounts of input noise, and did not appear to be affected by the ordering of examples. Distribution weighting did not appear to affect center placement substantially—the unit configuration was very sensitive to local frequencies of examples, leading to sparse covering of areas where examples occurred less frequently. The addition of error weighting led to the proliferation of fewer units in some regions but did not necessarily improve final network performance. The new incremental width training algorithm was very successful at adjusting the overlap of neighboring units.

Simultaneous training of basis functions and influence functions (centers and widths) yielded adequate results: training error decreased overall despite possible interference from the unsupervised influence training.

However, the addition of training noise to the targets substantially increased the validation error of the final network.

The adaptive structure algorithm meshed very well with the hybrid parameter training algorithm. Newly created units were quickly incorporated into the network structure without introducing devastating transient errors. Reducing the widths of new units relative to the old unit served to facilitate their speedy incorporation. Structure adaptation yielded more flexible and complete utilization of available units than did fixed structure networks trained with either the hybrid or fully supervised algorithm. In addition, structure adaptation did not appear to increase the training time required to achieve a particular error, compared to training times and errors of the fixed structure networks.

There seemed to be a problem with overproliferation of units, perhaps due to the frequent splitting of units before basis functions had achieved minimum local errors. Not only did this cause inefficient use of network resources, but generalization may have been compromised as well as a result of a decrease in smoothness of the network mapping.

By evaluating and training only those units ("active" units) that contributed significantly to the current network output (according to the desired cumulative normalized influence), the computational cost of training was *dramatically* reduced for networks constructed of Gaussian units. Not only was the set of active units much smaller than the network as a whole, the number of active units actually stabilized to a small constant value. This reduction in computation is an extremely important feature of the algorithm, allowing "the curse of dimensionality" inherent in spatially localized networks to be circumvented to a large extent.

Inverse square influence functions were shown to be a viable alternative to Gaussian functions for use in basis/influence function networks. They afford the advantages of exact interpolation and sensible, automatic transitions between neighboring units, and can be trained to yield networks of similar quality to those constructed from Gaussian units. However, since inverse square functions decrease polynomially away from center rather than exponentially, they are essentially less localized than Gaussian functions; this accounts for the higher number of active units observed for the inverse square network and the corresponding rise in computational cost, which introduces an important trade-off between representational convenience and computational efficiency.

7.1.2 Overall Assessment

In comparison to the fully supervised algorithm and that of Stokbro et al, the adaptive structure hybrid algorithm performed adequately for a first effort. As discussed in the next section, there are a number of additions and modifications that can be made to the algorithm that should both decrease convergence time and lead to more efficient usage of network resources.

A future, improved adaptive structure algorithm will be useful in the design of systems for control, estimation, and prediction of dynamical systems, by effectively automating network design with a subsequent reduction in development cost.

7.2 Recommendations for Further Work

7.2.1 Parameter Training Algorithm

- *Learning rate decay*. The variance of a parameter about its "ideal" value is related to the size of its learning rate, and places a lower limit on the average error that can be attained through

training. As a parameter reaches the vicinity of its "ideal" value, its learning rate should be decreased to reduce the parameter variance, likewise allowing the mean approximation error to be reduced. This learning rate decay could be linked to the convergence measure currently used to indicate when unit splitting should occur.

- *Recursive Least Squares for basis function training*. RLS is known to converge quickly for quasi-stationary target mappings [1, 31]. Therefore, basis function training might be improved by replacing the current Least Mean Squares algorithm with an age-weighted version of RLS, to allow for unit migration and corresponding changes in the local target mapping.

- *Improved distribution weighting*. It may still be possible to achieve a more uniform mapping over those areas of input space in which examples appear by adjusting learning rates for centers and widths to account for relative densities of examples in different areas. In particular, center and width training could be turned off for overtrained units. Hopefully, the disproportionate effects of limit cycles, set points, etc., on the unit configuration could be reduced.

- *Smoother network mappings*. Better generalization might be attained by increasing the overlap among neighboring units, i.e., training the unit width to extend beyond the standard deviation of the local set of inputs, which would yield smoother interpolation. In addition, basis functions of the current set of active units could be compared and adjusted to increase local smoothness.

- *Confidence measure for network outputs*. In addition to providing an approximation to the target mapping, the network could provide an additional output that indicates the estimated accuracy or validity of the network output as a function of the current input. For instance, outputs generated primarily by units having large local errors should be accompanied by a correspondingly low measure of confidence; outputs given in

response to relatively novel or unfamiliar inputs should likewise be identified as unreliable.

7.2.2 Adaptive Structure Algorithm

- *Improved measure of parameter convergence*. The current measure of convergence of a unit's parameter vector is prone to domination by individual parameters that receive relatively large adjustments, so that convergence is signaled if and only if these dominant parameters have converged. This disparity might be avoided or reduced by: i) computing convergence separately for individual parameters, and then combining the results in a total convergence measure; ii) weighting individual parameter updates (according to prior knowledge of typical parameter magnitudes, or observed past magnitudes) before calculating the convergence measure.
- *More efficient basis function training before splitting units*. Splitting undertrained units leads to increased network size, inefficient utilization of available network resources, and increased computation costs. Basis functions must be guaranteed to be fully trained (i.e., have nearly the lowest possible local error) before units are split. This may be achieved by using learning rate decay or by improving the convergence measure (as described previously); it may also be helpful to measure local error convergence, and to split only when parameters and local error have all converged.
- *Removal of redundant/underutilized units (i.e., pruning)*. There will always be the possibility of adding units to the network that are not subsequently needed or used. By eliminating these extra resources, network representational and computational efficiency can be increased. Redundancy might be detected by comparing the basis functions of neighboring (currently active) units to determine whether their contribution to the network output could be adequately approximated using a smaller number. Under-utilized units could be identified

simply by setting thresholds for training activity, and those units that are seldom trained or evaluated would be removed.

7.2.3 Meta-Learning

On-line structure adaptation is fundamentally distinct from on-line parameter training, and there is no reason why the two should be performed in synchrony, especially given the time constraints imposed on the learning algorithm when the network is evaluated and updated at each time step. It may be more appropriate to view and to treat structure adaptation as a process that is performed independently of parameter training, on its own time scale and according to its own objectives. As parameter training proceeds as usual, a meta-learning system would observe and evaluate the progress of learning given the current network structure, making modifications along the way to improve the ability of the network and algorithm to meet the learning objectives of low error and good generalization.

The list of additional tasks that might be undertaken by a meta-learning system is extensive. In addition to structure augmentation and reduction, meta-learning might involve the selection of alternative basis functions to achieve better local approximations (e.g., sinusoidal, quadratic). Learning rates could be regulated, or learning simply turned on and off in different regions of space. Graduated learning (i.e., the sequential construction of crude approximations followed by finer-grained corrections) could be implemented. Networks could be constructed of global as well as spatially localized components, where appropriate.

The idea of meta-learning is simply another way of thinking about solutions to the same set of problems. As novel paradigms often do, it may lead to new, improved methods of solving those problems.

A Inverse Square Networks

In this appendix, we will show the following properties for networks that are constructed from units composed of pairs of inverse square influence functions and affine basis functions that share the same center:

- **The network performs exact interpolation.** For inputs that coincide exactly with the center of any unit, the network output is equivalent to that unit's basis function value at its center, which in the case of affine basis functions is just the offset w_0—

$$y(\mathbf{x}_j^c) = B_j(\mathbf{x}_j^c) = \mathbf{w}_j^{\mathrm{T}} \cdot (\mathbf{x}_j^c - \mathbf{x}_j^c) + w_{0j} = w_{0j} \tag{A.1}$$

- **Basis functions are local 1st-order approximations.** Likewise, the first derivative of the network output is equivalent to the first derivative of the local basis function, i.e., the weight vector **w**—

$$\text{for } \mathbf{x} = \mathbf{x}_j^c: \qquad \frac{\partial y}{\partial \mathbf{x}} = \frac{\partial B_j}{\partial \mathbf{x}} = \mathbf{w}_j^{\mathrm{T}} \tag{A.2}$$

  Thus, *each basis function is the 1st-order Taylor approximation* to the network mapping about the corresponding unit center—

$$B_j(\mathbf{x}) = y(\mathbf{x}_j^c) + \left.\frac{\partial y}{\partial \mathbf{x}}\right|_{\mathbf{x}_j^c} \cdot (\mathbf{x} - \mathbf{x}_j^c) \tag{A.3}$$

- **The resulting network mapping is smooth.** Basis functions, normalized influence functions, and their first derivatives exist and are bounded over bounded regions—

$$\frac{\partial y}{\partial \mathbf{x}} = \sum_k \left[\frac{\partial B_k}{\partial \mathbf{x}} \cdot I_k(\mathbf{x}) + B_k(\mathbf{x}) \cdot \frac{\partial I_k}{\partial \mathbf{x}} \right] \tag{A.4}$$

  Therefore, the first derivative of the network output exists over the entire training domain, and the mapping is smooth.

A.1 Network Definition

A.1.1 Network equations

Below are the equations necessary to evaluate the network output given an input $\mathbf{x}$ and a set of basis function and influence function parameters, all of which are assumed to be bounded. Unit widths are assumed to be greater than zero. For analytical convenience, all unit centers are assumed to be distinct; however, this assumption is later shown not to be necessary.

Where it is convenient and unlikely to result in confusion in the derivations, the dependency on $\mathbf{x}$ is omitted from expressions.

Network output: $$y(\mathbf{x}) = \sum_k B_k(\mathbf{x}) \cdot I_k(\mathbf{x}) \tag{A.5}$$

Basis function: $$B_j(\mathbf{x}) = \mathbf{w}_j^{\mathrm{T}} \cdot \left(\mathbf{x} - \mathbf{x}_j^c\right) + w_{0j} \tag{A.6}$$

Normalized influence: $$I_j(\mathbf{x}) = \frac{i_j(\mathbf{x})}{\sum_k i_k(\mathbf{x})} \tag{A.7}$$

Unnormalized influence: $$i_j(\mathbf{x}) = \frac{\sigma_j^2}{r_j^2(\mathbf{x})} \tag{A.8}$$

Squared distance from input to center of j-th unit: $$r_j^2(\mathbf{x}) = \left\|\mathbf{x} - \mathbf{x}_j^c\right\| = \left(\mathbf{x} - \mathbf{x}_j^c\right)^{\mathrm{T}} \cdot \left(\mathbf{x} - \mathbf{x}_j^c\right) \tag{A.9}$$

Units have nonzero widths: $$\sigma_j^2 > 0, \ \forall j \tag{A.10}$$

Assume all centers are distinct (for now): $$\mathbf{x}_j^c \neq \mathbf{x}_k^c, \ \forall j,k, \ j \neq k \tag{A.11}$$

A.1.2 Useful Relations

The following three relations will be used to simplify complicated expressions and to eliminate infinite quantities from equations.

Sum of normalized influences:

$$\sum_k I_k = \sum_k \left(\frac{i_k}{\sum_m i_m} \right) = \frac{\sum_k i_k}{\sum_m i_m} = 1 \tag{A.12}$$

Relations between normalized and unnormalized influences:

$$\frac{1}{\sum_k i_k} = \frac{I_j}{i_j} \quad \forall j, i_j \neq 0 \tag{A.13}$$

$$I_m \cdot i_j = \frac{i_m \cdot i_j}{\sum_k i_k} = i_m \cdot I_j \quad \forall j, m \tag{A.14}$$

A.2 Exact Interpolation

The values of normalized influences are computed for inputs that exactly "hit" unit centers. Finally, it is shown that in these cases, the network output equals the basis function offset value of the "hit" unit.

A.2.1 Normalized influence of unit *j* at its center—

$$\begin{aligned} I_j\left(\mathbf{x}_j^c\right) &= \frac{i_j\left(\mathbf{x}_j^c\right)}{\sum_k i_k\left(\mathbf{x}_j^c\right)} = \frac{1}{1+\sum_{k\neq j} \frac{i_k\left(\mathbf{x}_j^c\right)}{i_j\left(\mathbf{x}_j^c\right)}} \\ &= \frac{1}{1+\sum_{k\neq j} \frac{\sigma_k^2}{\sigma_j^2} \cdot \frac{r_j^2}{r_k^2}} = \frac{1}{1+\sum_{k\neq j} \frac{\sigma_k^2}{\sigma_j^2} \cdot \frac{0}{r_k^2}} \\ &= \frac{1}{1+0} = 1 \end{aligned} \tag{A.15}$$

A.2.2 Normalized influence of unit j at center of unit m, $m \neq j$—

$$
\begin{aligned}
I_j\left(\mathbf{x}_m^c\right) &= \frac{i_j\left(\mathbf{x}_m^c\right)}{\sum_k i_k\left(\mathbf{x}_m^c\right)} = \frac{\dfrac{i_j\left(\mathbf{x}_m^c\right)}{i_m\left(\mathbf{x}_m^c\right)}}{1+\sum_{k\neq m}\dfrac{i_k\left(\mathbf{x}_m^c\right)}{i_m\left(\mathbf{x}_m^c\right)}} \\
&= \frac{\dfrac{\sigma_j^2}{\sigma_m^2}\cdot\dfrac{r_m^2}{r_j^2}}{1+\sum_{k\neq m}\dfrac{\sigma_k^2}{\sigma_m^2}\cdot\dfrac{r_m^2}{r_k^2}} = \frac{\dfrac{\sigma_j^2}{\sigma_m^2}\cdot\dfrac{0}{r_j^2}}{1+\sum_{k\neq m}\dfrac{\sigma_k^2}{\sigma_m^2}\cdot\dfrac{0}{r_k^2}} \\
&= \frac{0}{1+0} = 0
\end{aligned}
\tag{A.16}
$$

A.2.3 Network output at center of any unit j—

$$
\begin{aligned}
y\left(\mathbf{x}_j^c\right) &= \sum_k B_k\left(\mathbf{x}_j^c\right)\cdot I_k\left(\mathbf{x}_j^c\right) = B_j\left(\mathbf{x}_j^c\right) \\
&= \mathbf{w}_j^{\mathrm{T}}\cdot\left(\mathbf{x}_j^c-\mathbf{x}_j^c\right)+w_{0j} \\
&= w_{0j}
\end{aligned}
\tag{A.17}
$$

A.3 Local 1st-order Approximations

It is shown that basis function slope vectors $\mathbf{w}_j$ are equivalent to the first derivatives of the network output evaluated at corresponding unit centers.

A.3.1 First Derivatives w.r.t. Input

Network output: $$\frac{\partial y}{\partial \mathbf{x}} = \sum_k\left[\frac{\partial B_k}{\partial \mathbf{x}}\cdot I_k(\mathbf{x})+B_k(\mathbf{x})\cdot\frac{\partial I_k}{\partial \mathbf{x}}\right] \tag{A.18}$$

Basis function:
$$\frac{\partial B_j}{\partial \mathbf{x}} = \mathbf{w}_j^{\mathrm{T}} \tag{A.19}$$

Unnormalized influence:
$$\frac{\partial i_j}{\partial \mathbf{x}} = \frac{\partial i_j}{\partial r_j^2} \cdot \frac{\partial r_j^2}{\partial \mathbf{x}} = -\frac{2}{\sigma_j^2} \cdot i_j^2 \cdot \left(\mathbf{x} - \mathbf{x}_j^c\right)^{\mathrm{T}} \tag{A.20}$$

Squared distance:
$$\frac{\partial r_j^2}{\partial \mathbf{x}} = 2\left(\mathbf{x} - \mathbf{x}_j^c\right)^{\mathrm{T}} \tag{A.21}$$

Normalized influence:

$$\begin{aligned}
\frac{\partial I_j}{\partial \mathbf{x}} &= \frac{1}{\left(\sum_k i_k\right)^2} \cdot \left(\frac{\partial i_j}{\partial \mathbf{x}} \cdot \sum_k i_k - i_j \cdot \sum_k \frac{\partial i_k}{\partial \mathbf{x}}\right) \\
&= \frac{1}{\sum_k i_k} \cdot \left(\frac{\partial i_j}{\partial \mathbf{x}} - I_j \cdot \sum_k \frac{\partial i_k}{\partial \mathbf{x}}\right) \\
&= \frac{1}{\sum_k i_k} \cdot \left[\left(1 - I_j\right) \cdot \frac{\partial i_j}{\partial \mathbf{x}} - \sum_{k \neq j} I_j \cdot \frac{\partial i_k}{\partial \mathbf{x}}\right] \\
&= \frac{1}{\sum_k i_k} \cdot \left[\sum_{k \neq j} I_k \cdot \frac{\partial i_j}{\partial \mathbf{x}} - \sum_{k \neq j} I_j \cdot \frac{\partial i_k}{\partial \mathbf{x}}\right] \\
&= \frac{-2}{\sum_k i_k} \cdot \sum_{k \neq j} \left[\frac{1}{\sigma_j^2} \cdot I_k \cdot i_j^2 \cdot \left(\mathbf{x} - \mathbf{x}_j^c\right)^{\mathrm{T}} - \frac{1}{\sigma_k^2} \cdot I_j \cdot i_k^2 \cdot \left(\mathbf{x} - \mathbf{x}_k^c\right)^{\mathrm{T}}\right] \\
&= -2 \cdot \sum_{k \neq j} \left[\frac{1}{\sigma_j^2} \cdot I_j \cdot I_k \cdot i_j \cdot \left(\mathbf{x} - \mathbf{x}_j^c\right)^{\mathrm{T}} - \frac{1}{\sigma_k^2} \cdot I_k \cdot I_j \cdot i_k \cdot \left(\mathbf{x} - \mathbf{x}_k^c\right)^{\mathrm{T}}\right]
\end{aligned} \tag{A.22}$$

A.3.2 Derivatives Evaluated at Unit Centers

Substitutions are made to eliminate infinite-valued influences resulting from inputs at the centers of units. It is shown that the first derivatives of all normalized influence functions evaluate to zero at the

center of every unit. Consequently, the first derivative of the network mapping equals the slope vector of the "hit" unit.

for $\mathbf{x} = \mathbf{x}_j^c$:

$$\begin{aligned}\frac{\partial I_j}{\partial \mathbf{x}} &= -2 \cdot \sum_{k \neq j} \left[\frac{1}{\sigma_j^2} \cdot I_j \cdot i_k \cdot I_j \cdot \left(\mathbf{x}_j^c - \mathbf{x}_j^c\right)^{\mathrm{T}} - \frac{1}{\sigma_k^2} \cdot I_k \cdot I_j \cdot i_k \cdot \left(\mathbf{x}_j^c - \mathbf{x}_k^c\right)^{\mathrm{T}} \right] \\ &= -2 \cdot \sum_{k \neq j} \left[\frac{1}{\sigma_j^2} \cdot 1 \cdot i_k \cdot 1 \cdot \mathbf{0}^{\mathrm{T}} - \frac{1}{\sigma_k^2} \cdot 0 \cdot 1 \cdot i_k \cdot \left(\mathbf{x}_j^c - \mathbf{x}_k^c\right)^{\mathrm{T}} \right] \\ &= \mathbf{0}^{\mathrm{T}}\end{aligned} \tag{A.23}$$

for $\mathbf{x} = \mathbf{x}_m^c$, $m \neq j$:

$$\begin{aligned}\frac{\partial I_j}{\partial \mathbf{x}} &= -2 \cdot \sum_{k \neq j} \left[\frac{1}{\sigma_j^2} \cdot I_j \cdot I_k \cdot i_j \cdot \left(\mathbf{x}_m^c - \mathbf{x}_j^c\right)^{\mathrm{T}} - \frac{1}{\sigma_k^2} \cdot I_k \cdot i_j \cdot I_k \cdot \left(\mathbf{x}_m^c - \mathbf{x}_k^c\right)^{\mathrm{T}} \right] \\ &= -2 \cdot \left\{ \left[\frac{1}{\sigma_j^2} \cdot I_j \cdot I_m \cdot i_j \cdot \left(\mathbf{x}_m^c - \mathbf{x}_j^c\right)^{\mathrm{T}} - \frac{1}{\sigma_m^2} \cdot I_m \cdot i_j \cdot I_m \cdot \left(\mathbf{x}_m^c - \mathbf{x}_m^c\right)^{\mathrm{T}} \right] \right. \\ &\qquad \left. + \sum_{\substack{k \neq j, \\ k \neq m}} \left[\frac{1}{\sigma_j^2} \cdot I_j \cdot I_k \cdot i_j \cdot \left(\mathbf{x}_m^c - \mathbf{x}_j^c\right)^{\mathrm{T}} - \frac{1}{\sigma_k^2} \cdot I_k \cdot i_j \cdot I_k \cdot \left(\mathbf{x}_m^c - \mathbf{x}_k^c\right)^{\mathrm{T}} \right] \right\} \\ &= -2 \cdot \left\{ \left[\frac{1}{\sigma_j^2} \cdot 0 \cdot 1 \cdot i_j \cdot \left(\mathbf{x}_m^c - \mathbf{x}_j^c\right)^{\mathrm{T}} - \frac{1}{\sigma_m^2} \cdot 1 \cdot i_j \cdot 1 \cdot \mathbf{0}^{\mathrm{T}} \right] \right. \\ &\qquad \left. + \sum_{\substack{k \neq j, \\ k \neq m}} \left[\frac{1}{\sigma_j^2} \cdot 0 \cdot 0 \cdot i_j \cdot \left(\mathbf{x}_m^c - \mathbf{x}_j^c\right)^{\mathrm{T}} - \frac{1}{\sigma_k^2} \cdot 0 \cdot i_j \cdot 0 \cdot \left(\mathbf{x}_m^c - \mathbf{x}_k^c\right)^{\mathrm{T}} \right] \right\} \\ &= \mathbf{0}^{\mathrm{T}}\end{aligned} \tag{A.24}$$

for $\mathbf{x} = \mathbf{x}_j^c$:

$$\frac{\partial y}{\partial \mathbf{x}} = \sum_k \left[\frac{\partial B_k}{\partial \mathbf{x}} \cdot I_k + B_k \cdot \frac{\partial I_k}{\partial \mathbf{x}} \right] = \frac{\partial B_j}{\partial \mathbf{x}} = \mathbf{w}_j^{\mathrm{T}} \tag{A.25}$$

A.3.3 Smoothness of Network Mapping

For inputs that do not coincide with unit centers, unnormalized influences have finite non-zero values, and normalized influences fall in the open interval (0,1). Since all widths are non-zero, all other parameters are finite, and all inputs are bounded, the first derivative of the network mapping exists over the entire (bounded) learning domain. Therefore, the mapping is smooth (i.e., all components of Eq. A.18 are well-behaved).

A.4 Distinctness of Unit Centers

The previous analysis assumed that all unit centers were distinct—in fact, this is an unnecessary condition. A similar analysis may be performed for networks containing concentric units if the unnormalized influences of units that share the same center are grouped together.

For example, imagine that some network contains a subset of units, p, that share a common center; call the remaining set of non-concentric units q. When calculating normalized influences, the members of p may be grouped together, effectively forming a new unit whose variance is the sum of the variance of all members of p. Members of q are dealt with as usual. A similar analysis may now be performed using this slight modification.

$$I_j = \frac{i_j}{\sum_k i_k} = \frac{\frac{\sigma_j^2}{r_j^2}}{\sum_k \frac{\sigma_k^2}{r_k^2}} = \frac{\frac{\sigma_j^2}{r_j^2}}{\sum_p \frac{\sigma_p^2}{r_p^2} + \sum_q \frac{\sigma_q^2}{r_q^2}} = \frac{\frac{\sigma_j^2}{r_j^2}}{\frac{1}{r_p^2} \cdot \sum_p \sigma_p^2 + \sum_q \frac{\sigma_q^2}{r_q^2}} = \cdots \qquad \text{(A.26)}$$

Note that the normalized influence of a unit at its own center is no longer equal to 1—influence must be "shared" among the concentric units according to variance:

$$I_j\left(\mathbf{x}_j^c\right) = \frac{\sigma_j^2}{\sum_p \sigma_p^2} \quad \text{if } j \in p \tag{A.27}$$

However, the normalized influences still sum to 1.

B The Aeroelastic Oscillator

B.1 Aeroelastic Galloping

(A thorough treatment of aeroelastic galloping can be found in the paper by Parkinson and Smith [23]. A brief description is presented here.)

When a steady wind flows across a flexible elastic structure, small oscillations of the structure in the direction transverse to air flow may be amplified, resulting in the phenomenon of *aeroelastic galloping*. Such behavior can be observed in airplane wings, bridges, or power lines on a windy day.[1]

The oscillation reaches a steady-state amplitude that varies nonlinearly with the velocity of the incident wind—i.e., the wind determines limit cycles in system state to which motion becomes constrained over time. Very large or very small wind velocities create limit cycles whose amplitudes generally increase with wind speed. However, bifurcations occur for wind velocities in between, yielding multiple limit cycles, any of which may capture the system states; in such cases, the steady-state behavior of the oscillator depends on both wind velocity and initial state.

The aeroelastic oscillator presents an interesting problem for prediction of future states from current state and wind velocity. The following sections describe the dynamics of the system and the method of constructing a predictor.

---

[1] Perhaps the most notorious example of aeroelastic galloping is the violent shaking, bucking, and eventual collapse of the Tacoma Narrows Bridge in the state of Washington one blustery day.

B.2 System Model

B.2.1 Equation of Motion

The aeroelastic oscillator is simply a flexible structure with incident air flow transverse to the direction of oscillatory motion. The physical system is modeled as a mass-spring-dashpot with an aerodynamic forcing term—

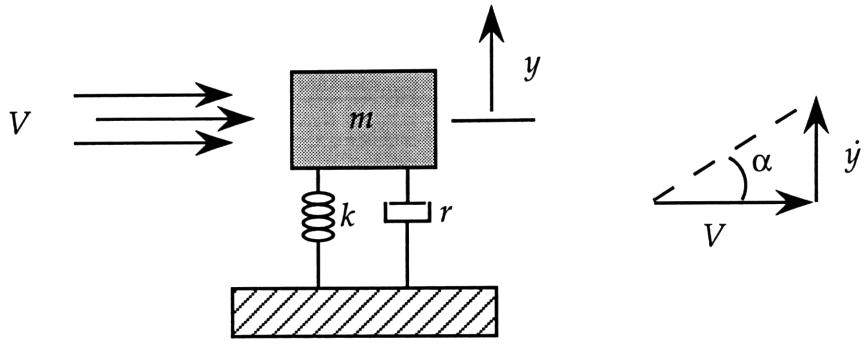


Figure B.1. Aeroelastic Oscillator Model

The equation of motion is

$$m\ddot{y} + r\dot{y} + ky = \tfrac{1}{2}C_{F_y}\rho V^2 hl \qquad \text{(B.1)}$$

where

m = mass of beam
k = spring constant
r = damping constant
y = displacement of beam from equilibrium
V = velocity of incident wind
ρ = density of air
h,l = dimensions of beam (square cross - section)
C_{F_y} = aerodynamic force coefficient

The coefficient C_{F_y} determines the magnitude of the aerodynamic force (in the direction of y) which results from the interaction of the wind with the moving object. This coefficient is an odd function of the angle-of-attack, α:

$$\alpha = \tan\left(\frac{\dot{y}}{V}\right) \tag{B.2}$$

For suitably small values of α

$$|\alpha| < \alpha_{max} = 16° \tag{B.3}$$

C_{F_y} can be approximated by a seventh-degree polynomial:

$$C_{F_y} \approx A\left(\frac{\dot{y}}{V}\right) - B\left(\frac{\dot{y}}{V}\right)^3 + C\left(\frac{\dot{y}}{V}\right)^5 - D\left(\frac{\dot{y}}{V}\right)^7 \tag{B.4}$$

$$A = 2.69 \quad B = 168 \quad C = 6,270 \quad D = 59,900$$

For more convenient nonlinear analysis, Parkinson and Smith make equation B.4 dimensionless by dividing through by kh and redefining terms:

$$\ddot{Y} + Y = nA \cdot \left[(U - U_0)\dot{Y} - \left(\frac{B}{AU}\right)\dot{Y}^3 + \left(\frac{C}{AU^3}\right) - \left(\frac{D}{AU^5}\right)\dot{Y}^7\right] \tag{B.5}$$

Y = nondimensional position

U = nondimensional wind velocity

U_0 = critical wind velocity = $\dfrac{r}{nAm\omega}$

n = mass parameter = $\dfrac{\rho h^2 l}{2m}$

$$\omega = \sqrt{\frac{k}{m}}$$

This nondimensionalized model is completely and conveniently specified by assigning arbitrary values to the critical wind velocity and mass parameter.

We shall therefore opt to implement simulations using this model rather than the original, dimensional version.

B.2.2 Simulation

Integration of the equation of motion is performed using a fourth-order Runge-Kutta algorithm with a step size of 0.01 seconds. In nondimensional time units, a single period of the oscillator has a length of roughly 2π, approximately equal to 628 simulation time steps.

The two free oscillator parameters—critical wind velocity and mass parameter—are both arbitrarily set to 1.0. This choice results in a system that converges quickly (within a few periods) from initial states to odd-shaped limit cycles (Figure B.3). The limit cycles themselves are determined by the velocity of the incident wind (Figure B.2).

For wind velocities less than critical, no galloping behavior is observed—the system always converges to the stable equilibrium point at the origin. As wind velocity increases, the system progresses through three distinct types of behavior:

1) one small stable limit cycle—The equilibrium point at the origin becomes unstable, and a limit cycle forms around it, whose amplitude increases with wind velocity.

2) two stable limit cycles—The system bifurcates, and the small stable limit cycle is joined by a larger stable limit cycle and an unstable limit cycle in between. The unstable equilibrium at the origin remains. The stable limit cycle amplitudes increase with wind velocity.

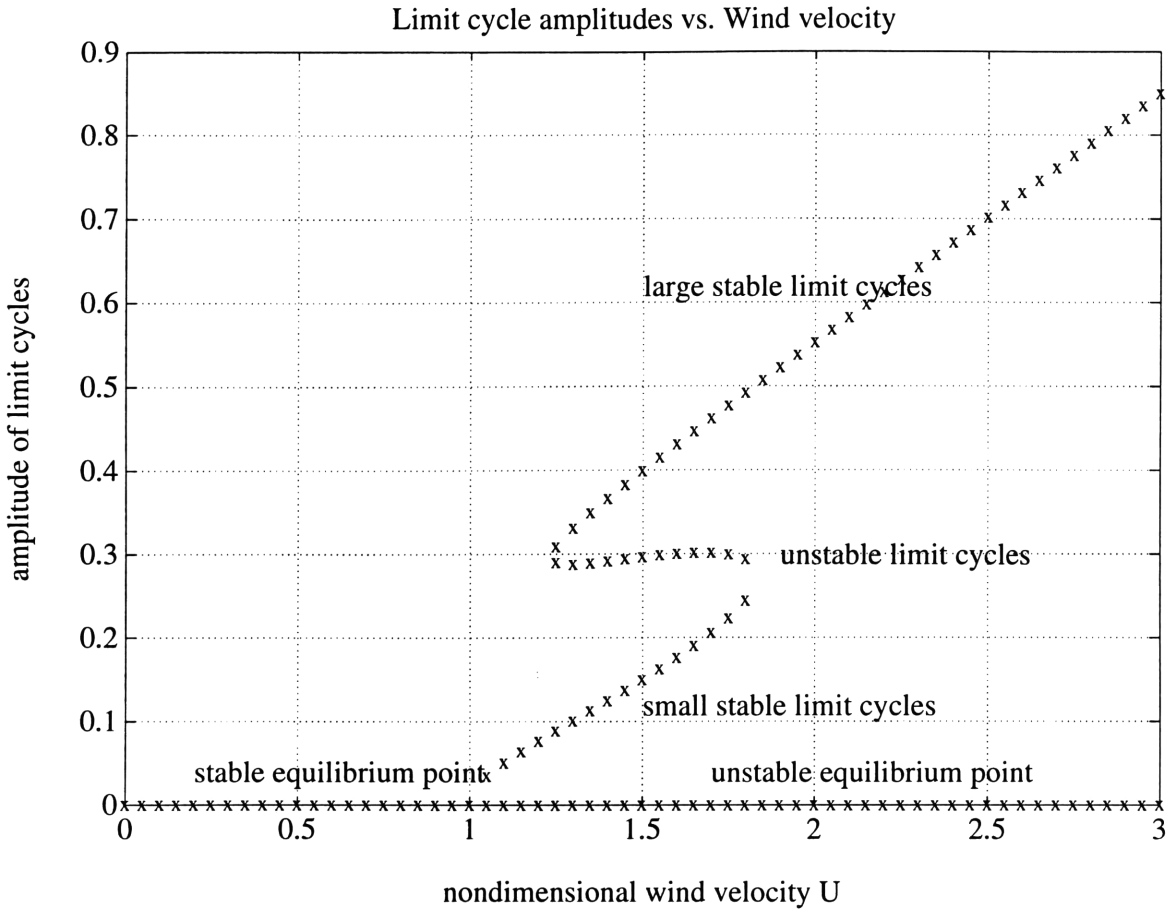


Figure B.2. Limit cycles of aeroelastic oscillator

3) one large stable limit cycle—The small stable limit cycle and the unstable limit cycle merge and cancel, leaving only the large stable limit cycle and the unstable equilibrium. The amplitude of the remaining limit cycle grows as wind velocity increases.

For this thesis, the incident wind velocity is set to a constant value of $U = 1.6$, which places the oscillator in the mode of most complex behavior.

Because the polynomial approximation to the aerodynamic force is valid only for small values of the angle-of-attack, initial states must be chosen

such that the velocity always remains small relative to the incident wind velocity; namely, that

$$\left|\frac{\dot{Y}}{U}\right| < \tan\alpha_{max} = \tan 16° \approx 0.29 \tag{B.6}$$

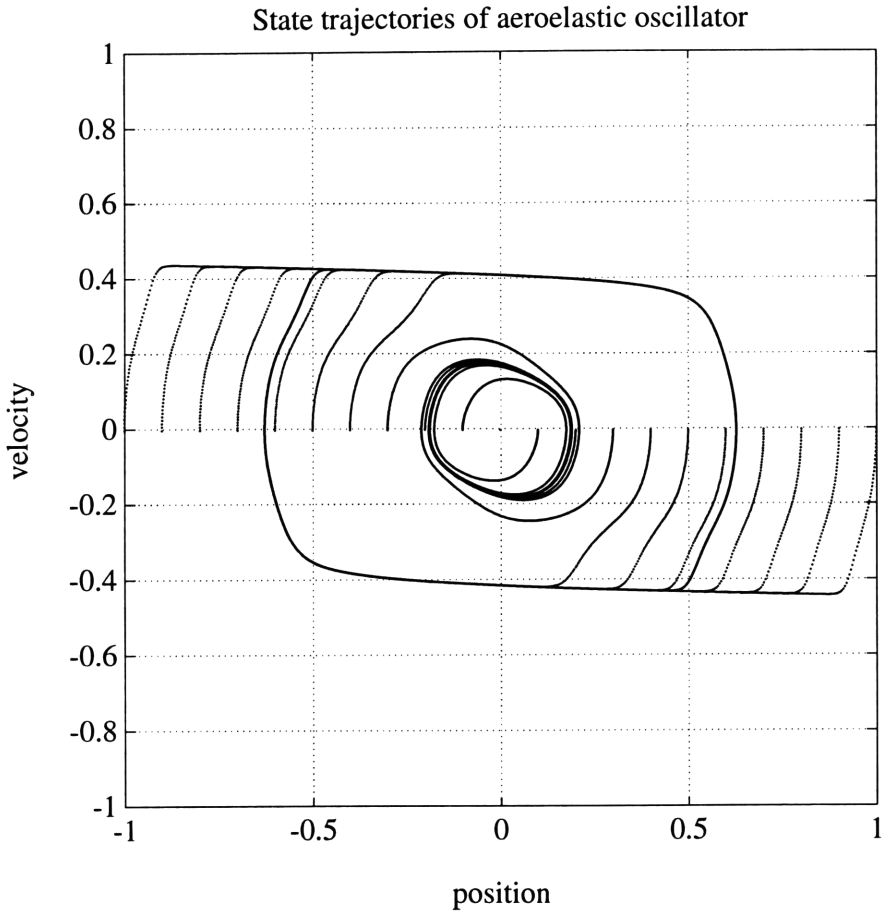


Figure B.3. State-space trajectories

Perhaps the safest approach is to set initial velocities equal to 0.0 while selecting initial positions never much larger than the amplitude of the largest limit cycle. In fact, this is the manner in which initial states shall be chosen

when using the simulation to provide training examples for learning prediction.

B.3 Constructing a Predictor

The aeroelastic oscillator simulation is used in the configuration below (see Chapter 1) to train a network to predict future positions y given the current position and velocity of the oscillator, and assuming an invariant incident wind velocity.

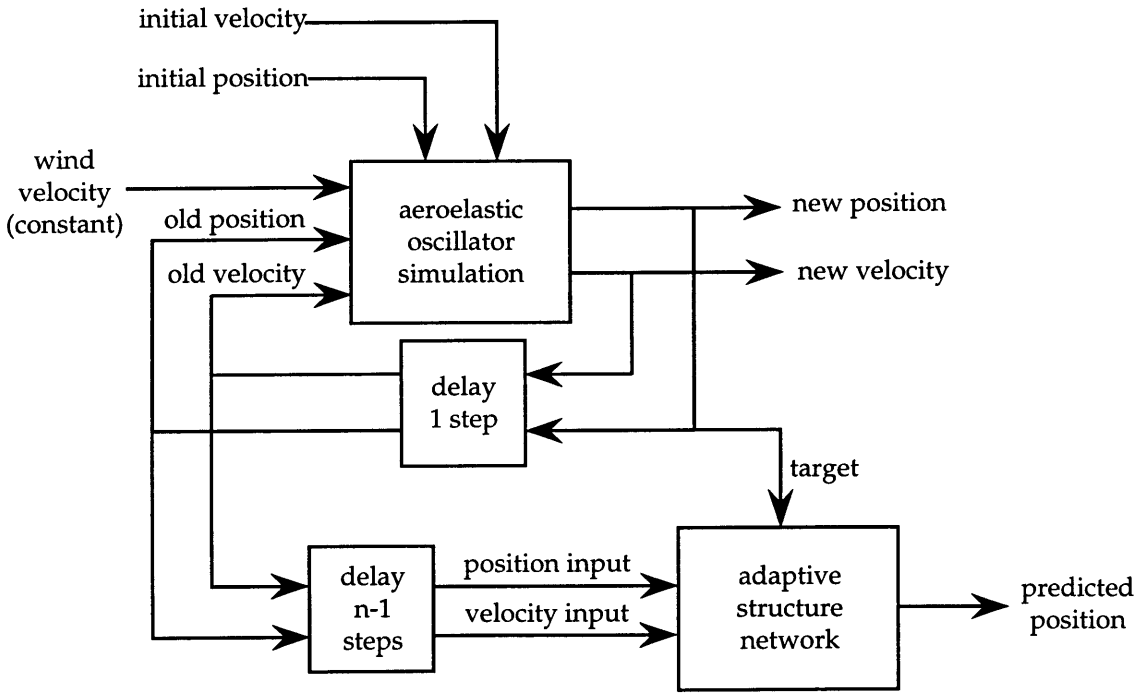


Figure B.4. Training an n-step predictor

The stream of training examples is constructed from observations of the state-space trajectories followed by the oscillator simulation after being initialized with random initial positions (and zero initial velocities). A single training examples consists of a system state (position and velocity) and the corresponding position n steps in the future. After setting the initial state of

the oscillator, the system is allowed to evolve along the resulting trajectory for a fixed length of time before the state is reset to another random value. (Obviously, for n-step prediction, each trajectory must evolve for n+1 time steps before training examples begin to be generated.)

Initial positions are confined to the interval [–0.5, 0.5], which lies entirely within the outer limit cycle—hence, the system state remains bounded, and instability due to excessively large velocities is not a concern. By choosing initial states for which position is random and velocity is zero, we are doing the equivalent of pulling up (or pressing down) on the beam a random distance, holding it there, and then letting go, allowing the beam in this case to converge to a steady-state oscillation corresponding to one of the two limit cycles.

Note that as trajectory lengths increase, the amount of time the system spends traveling along limit cycles becomes significantly greater than time spent in other regions, especially regions near the unstable limit cycle and unstable equilibrium point. Long trajectories produce a stream of examples biased toward the limit cycles, and tend to focus training, and thus increase unit population density, in those regions.

C The Mackey-Glass Equation

(Material contained in this summary is based on information taken from [20, 30]. For a more thorough presentation of chaotic systems, please check your local library for literature on nonlinear dynamical systems [32].)

C.1 The Mackey-Glass Delay Differential Equation

A challenging prediction problem would be to attempt to predict future behavior of an "unpredictable" system. So, a fitting method of evaluating the quality of a learning algorithm is to measure its success at predicting future states of chaotic systems.

The Mackey-Glass delay differential equation gives rise to such a chaotic system. That is, when fully initialized and allowed to evolve over time, future states becomes increasingly difficult to predict solely from observations of past states.

The Mackey-Glass equation is fully deterministic: its unpredictability relies not on any "randomness", but rather on its infinite dimensionality and the potentially large influence past states may exert on future states.

$$\dot{x}(t) = -b \cdot x(t) + a \cdot \frac{x(t-\tau)}{1 + x(t-\tau)^{10}} \qquad \text{(C.1)}$$

To determine the value of $x(t)$ for $t > t_0$, one must specify exactly the value of $x(t)$ on the interval $[t_0 - \tau, t_0]$. This amounts to having perfect knowledge of each member of an infinite set. Unless these values are somehow known exactly by some means other than observation, e.g., by explicitly setting the initial conditions, $x(t)$ cannot be calculated very far into the future with accuracy greater than chance.

C.2 Simulation

The Mackey-Glass equation is integrated using a 4th-order Runge-Kutta algorithm with a step size of 1 and the same parameter values as in [30]: $a = 0.2$, $b = 0.1$, $\tau = 17$. Initial values of $x[t]$ for integer values of t on [–17, 0] were chosen at random from [0, 1]. The resulting chaotic series is quasi-periodic with a characteristic period of $t_{char} \approx 50$ (i.e., the reciprocal of the mean of the power spectral density). Both the quasi-periodicity and the sensitivity of the system to initial conditions are manifested in the following plot of two such series (Figure C.1).

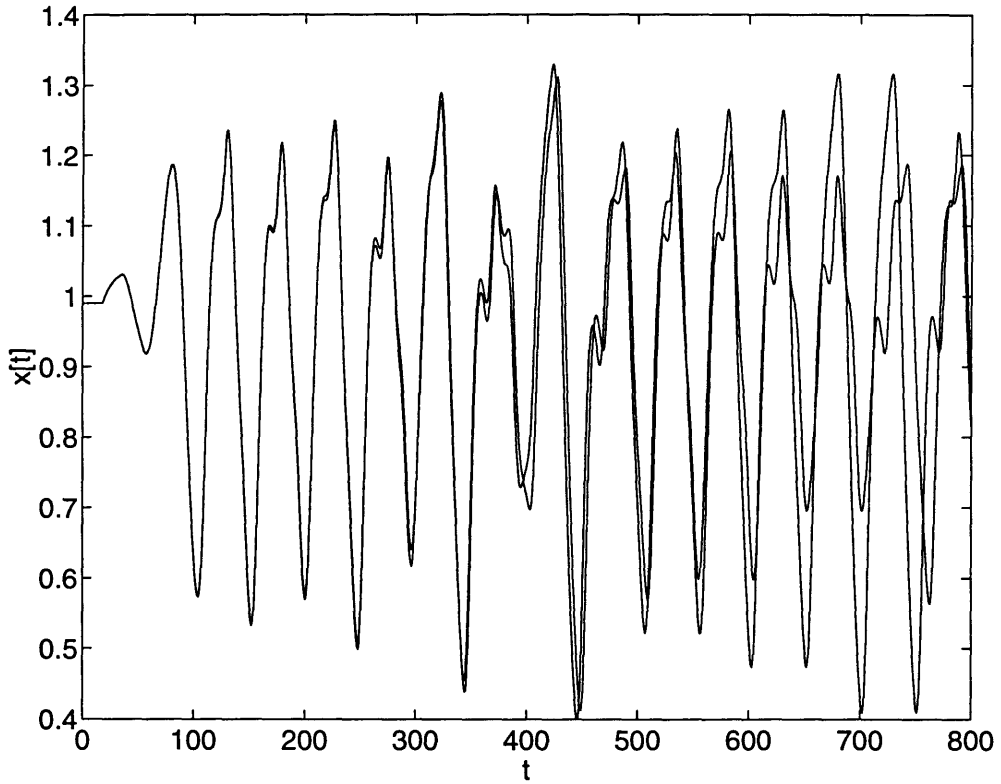


Figure C.1. Two series generated by the Mackey-Glass equation from slightly different initial conditions. Divergence becomes apparent after approximately four characteristic time periods.

C.3 Prediction

Using the Mackey-Glass simulation to generate training data in a fashion similar to that of the aeroelastic oscillator problem (see Appendix B), an adaptive structure network is trained to predict the state of the system T steps in the future, where

$$T = 6 \tag{C.2}$$

Four past state values are provided as inputs to the network:

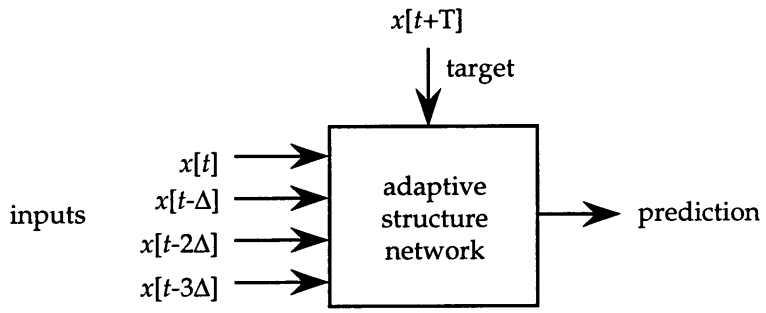


Figure C.2. Training an n-step predictor

where $\Delta = 6$. The simulation is initialized once and allowed to run for the duration of training.

Bibliography

[1] Åström, K. & Wittenmark, B. (1989). *Adaptive Control,* Addison-Wesley.

[2] Baird, L. (1991). *Learning and Adaptive Hybrid Systems for Nonlinear Control,* CSDL Report T-1099, M.S. Thesis, Department of Computer Science, Northeastern University.

[3] Baker, W. & Farrell, J. (1992). "An Introduction to Connectionist Learning Control Systems", in White, D. & Sofge, D., eds., *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches,* Van Nostrand Reinhold.

[4] Berger, T. (1992). *Control of Unmanned Underwater Vehicles Using Spatially Localized Learning Methods,* CSDL Report T-1142, M.S. Thesis, Department of Mechanical Engineering, M.I.T.

[5] Farmer, J. & Sidorowich, J. (1987). "Predicting Chaotic Time Series", *Physical Review Letters,* Vol. 59, Number 8, pp. 845-848.

[6] Funahashi, K. (1989). "On the Approximate Realization of Continuous Mappings by Neural Networks", *Neural Networks,* Vol. 2, pp. 183-192.

[7] Hertz, J., Krogh, A., & Palmer, R. (1991). *Introduction to the Theory of Neural Computation,* Vol. I, Addison-Wesley.

[8] Hirose, Y., Yamashita, K., & Hijiya, S. (1991). "Back-Propagation Algorithm Which Varies the Number of Hidden Units", *Neural Networks,* Vol. 4, pp. 61-66.

[9] Hornik, K. (1989). "Multilayer Feedforward Networks are Universal Approximators", *Neural Networks,* Vol. 2, pp. 359-366.

[10] Jacobs, Robert A. (1988). "Increased Rates of Convergence Through Learning Rate Adaptation", *Neural Networks,* Vol. 1, No. 4, pp. 295-307.

[11] Ji, C., Snapp, R., & Psaltis, D. (1990). "Generalizing Smoothness Constraints from Discrete Samples", *Neural Computation,* Vol. 2, pp. 188-197.

[12] Johansen, T. & Foss, B. (1992). "Representing and Learning Unmodeled Dynamics With Neural Network Memories", ACC/FM2, pp. 3037-3043.

[13] Kohonen, T. (1982). *Self-organization and Associative Memory*, Springer-Verlag.

[14] Lancaster, P. & Salkauskas, K. (1986). *Curve and Surface Fitting: An Introduction*, Harcourt Brace Jovanovich.

[15] Lapedes, A. & Farber, R. (1987). *Nonlinear signal processing using neural networks: Prediction and system modeling*. Technical Report, Los Alamos National Laboratory, Los Alamos, New Mexico.

[16] Lee, S. & Kil, R. (1991). "A Gaussian Potential Function Network With Hierarchically Self-Organizing Learning", *Neural Networks*, Vol. 4, pp. 207-224.

[17] Ljung, L. (1987). *System Identification: Theory for the User*, Prentice Hall.

[18] Millington, P. (1991). *Associative Reinforcement Learning for Optimal Control*, CSDL Report T-1070, M.S. Thesis, Department of Aeronautics and Astronautics, M.I.T.

[19] Millington, P. & Baker, W. (1992). *Learning Augmented Flight Control for High Performance Aircraft*, Final Report, USN Contract No. N62269-91-C-0033, Draper Laboratory, Cambridge, MA.

[20] Moody, J. & Darken, C. (1989). "Fast Learning in Networks of Locally-Tuned Processing Units", *Neural Computation*, Vol. 1, pp. 281-294.

[21] Narendra, K. & Parthasarathy, K. (1990). "Identification and Control of Dynamical Systems Using Neural Networks", *IEEE Transactions on Neural Networks*, Vol. 1, No. 1.

[22] Nistler, N. (1992). *A Learning Enhanced Flight Control System for High Performance Aircraft*, CSDL Report T-1127, M.S. Thesis, Department of Aeronautics and Astronautics, M.I.T.

[23] Parkinson, G. V. & Smith, J. D. (1964). "The Square Prism as an Aeroelastic Nonlinear Oscillator", *Quarterly Journal of Mechanics and Applied Mathematics*, Vol. XVII, Pt. 2.

[24] Platt, J. (1991). "A Resource-Allocating Network for Function Interpolation", *Neural Computation*, Vol. 3, pp. 213-225.

[25] Powell, M. (1985). *Radial Basis Functions for Multivariate Interpolation: A Review.* Technical Report DAMPT 1985/NA12, Department of Applied Mathematics and Theoretical Physics, Cambridge University.

[26] Press, W., Flannery, B., Teukolsky, S., & Vetterling, W. (1988). *Numerical Recipes in C: The Art of Scientific Computing,* Press Syndicate of the University of Cambridge.

[27] Rosenblatt, F. (1962). *Principles of Neurodynamics,* Spartan.

[28] Rudin, W. (1976). *Principles of Mathematical Analysis,* p.59, McGraw-Hill.

[29] Stinchcombe, M. & White, H. (1989). "Universal Approximation using Feedforward Networks with Non-sigmoid Hidden Layer Activation Functions", *IEEE International Joint Conference on Neural Networks,* SOS Printing, pp. I-613 - I-617.

[30] Stokbro, K., Umberger, D., & Hertz, J. (1990). "Exploiting Neurons with Localized Receptive Fields to Learn Chaos", *Complex Systems,* Vol. 4, pp. 603-622.

[31] Strang, G. (1986). *Introduction to Applied Mathematics,* Wellesley-Cambridge Press.

[32] Wiggins, S. (1990). *Introduction to Applied Nonlinear Dynamical Systems and Chaos,* Springer-Verlag.