NORTHWESTERN UNIVERSITY

Non-autonomous dynamical systems applicable to Neural Computation

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Mathematics

By

Michael Fiske

EVANSTON, ILLINOIS

December 1996

ABSTRACT

Non-autonomous dynamical systems applicable to Neural Computation

Michael Fiske

This thesis explores properties of classes of non-autonomous and autonomous dynamical systems which are relevant to neural network computation. In the introduction two examples are used to illustrate the critical role played by computation, training, and generalization in neural network computation. Using these examples, we explain why dynamical systems effectively models basic notions that arise in neural network computation.

In Chapter I, we introduce a way to compare two non-autonomous systems. It is interesting that this approach extends the notion of topological conjugacy for autonomous systems. In Chapter II, we discuss stability in non-autonomous systems. In Chapter III, some theorems are offered about non-autonomous systems in which each function is a contraction. In Chapter IV, we prove theorems about topological entropy for non-autonomous systems. In Chapter V we compute a lower bound for the entropy of a chaotic attractor and also characterize the geometry of the attractor. In Chapter VI, we show how to use chaotic non-autonomous systems in optimization algorithms as a method to find the global minima of the error function. In particular, this demonstrates the use of chaotic dynamics in a neural net training algorithm.

TABLE OF CONTENTS

Introduction	1
Chapter 1	12
Chapter 2	24
Chapter 3	38
Chapter 4	45
Chapter 5	82
Chapter 6	115
Bibliography	134

Introduction

Overview

This thesis explores properties of classes of non-autonomous and autonomous dynamical systems which are relevant to neural network computation. In the introduction two examples are used to illustrate the critical role played by computation, training, and generalization in neural network computation. Using these examples, we explain why dynamical systems effectively models basic notions that arise in neural network computation.

In Chapter I, we introduce a way to compare two non-autonomous systems. It is interesting that this approach extends the notion of topological conjugacy for autonomous systems. In Chapter II, we discuss stability in non-autonomous systems. In Chapter III, some theorems are offered about non-autonomous systems in which each function is a contraction. In Chapter IV, we prove theorems about topological entropy for non-autonomous systems. In Chapter V we compute a lower bound for the entropy of a chaotic attractor and also characterize the geometry of the attractor. In Chapter VI, we show how to use chaotic non-autonomous systems in optimization algorithms as a method to find the global minima of the error

function. In particular, this demonstrates the use of chaotic dynamics in a neural net training algorithm.

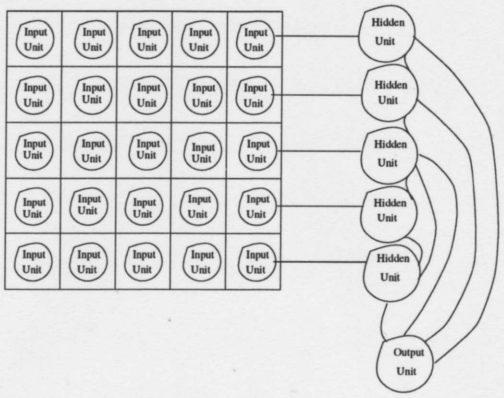
Neural Computation

Neural network computation has proven to be useful in many different applications. This is particularly true in areas such as vision, speech recognition, regulating and controlling production lines in manufacturing environments, and predicting the behavior of financial markets. The three main issues concerned with understanding the behavior and effectively using neural networks are computation, training and generalization. To explain what these issues mean and why they are important we describe two different examples in neural network computation. Then these examples are used to illustrate why the study of dynamics is important in computation, training and generalization.

EXAMPLE 1

COMPUTATION

Suppose the goal is to build a black box that recognizes a photograph of a particular person. We discuss one way of reaching this goal. Consider a black and white photograph, say of Michael Jones. To understand the computational processes inside the black box, partition the photograph into a large, finite number of tiny squares. Assign a computational input unit (input layer) to each square. The following picture illustrates this abstract model for our photograph example:



In this diagram, each unit is labeled as an oval. Units that send messages to each other have a curve connecting the two ovals. (Not all paths are drawn to avoid having a diagram that is overly complicated.) As for the actual mechanics of computation, each input unit sends as its message a real number in the unit interval $[0, 1]$; this value represents the average intensity of light in that square. Each message is sent to a subset of the hidden units. It is worth noting that the messages two different hidden units receive from the same input unit need not be the same. This is because two different hidden units may be assigned to different aspects of information about the photograph; e.g. one might be sensitive to skin color while the other might be sensitive to skin texture. The different wirings can be intricate where the hidden units may send messages to other hidden units, or in some models the hidden units may even send a message to some input units. But in the final analysis, all hidden units send a message to a final output unit. After receiving all messages from the hidden units, the output unit returns a 1 if the image is a photograph of Michael Jones, and 0 if the image is not a photograph of Michael Jones face.

As one might expect from this description, there are a variety of mathematical

models of the actual mechanics of computation. Often, these computational units perform a simple function such as taking a linear combination of the input variables and then applying a sigmoidal function. For example, one choice often used is $F(x_1, \ldots, x_n) = g(\sum w_i x_i)$, where $g(z) = \frac{1}{1+\exp^{-z}}$. When $|z| > 5$, then for most practical purposes, $g(z) \in \{0, 1\}$.

In other models, the computational units mathematically are variables x_j, in a set of differential equations: $\frac{dx_j}{dt} = g(\sum w_i x_i)$. As explained above, to include all of the models in our discussion, we refer to the simple computational entities as units, and we refer to the interactions that these units have with each other as messages. The term messages is appropriate because, in all the models, the computational units send each other information that is needed in order to perform a computational task.

TRAINING

A natural question is how do we specify the messages that the computational units should send each other to obtain the desired outputs? Actually, it is beyond current programming capabilities to directly construct these units and their messages so that the network (i.e. the units functioning as a collection) will recognize if a particular photo is Michael Jones. Instead, the idea is to start the network in a state where the units are sending arbitrary messages to each other, and then train the network on a number of different photographs.

To illustrate, suppose the training starts with a photograph of Michael Jones. If the output unit returns 1, then the rule is to not change any of the messages; otherwise, if the output returns 0, we adjust the messages the input layer sends to

the hidden layer, and we must adjust the messages the hidden layer sends to the output. Clearly, the goal is to reduce the effect of any message coming from the hidden layer that causes the output to be the wrong answer, 0. Similarly, the goal is to increase the effect of any message coming from the hidden layer that causes the output to be the correct answer, 1. The training process makes similar adjustments in the messages sent from the input layer to the hidden layer.

To illustrate these points with our example, suppose for the next training stage the network performs computation on a photograph of David Letterman. If the output unit emits a 0, then do not change anything. If the output is 1, however, we must adjust the messages the input units send to the hidden units; the messages the hidden units send to other hidden units; and the messages the hidden units send to the output unit.

The messages can be adjusted in the following way: If a unit in the input layer sends a message to the hidden layer that influences the output unit to be 0, then either decrease the strength of this message, or change the message that it sends to the hidden layer. Likewise, if a unit in the hidden layer sends a message to the output unit, that influences the output unit to be 0, then decrease the strength of its message to the output unit. On the other hand, if a unit in the hidden layer sends a message to the output unit that influences the output unit to be 1, then increase the strength of its message. This adjustment to the messages sent between units occurs after each training stage.

GENERALIZATION

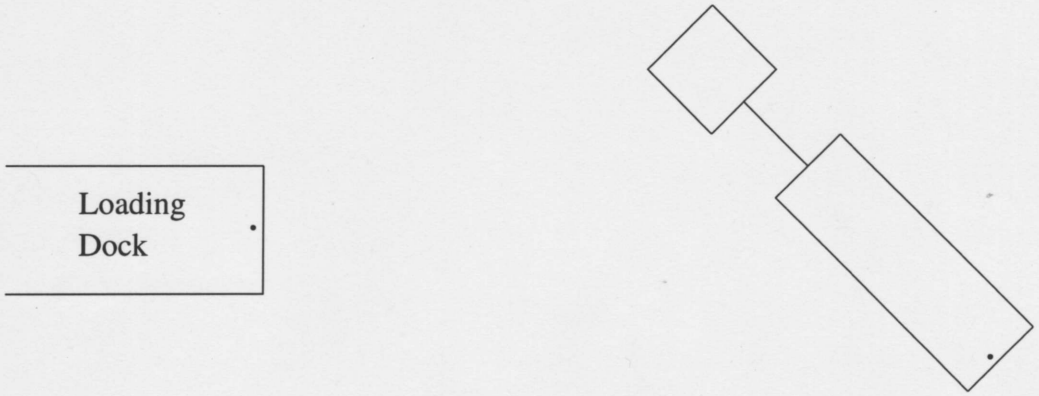
Suppose the network has been trained on 1000 photographs of Michael Jones

and 1000 photographs of people other than Michael Jones. If the trained network receives a photograph of Michael Jones that it has not been trained on, (maybe the new photograph displays Jones as a bald skinhead, rather than the usual spiked hairdo) the correct computation of the network is to return 1 at the output unit. If the network is given a photograph of someone who looks like Michael Jones, but who is not, then we want the network's output unit to return 0. This is what we mean by generalization; it is what the output unit of the network returns when confronted with new examples.

This abstract process can be compared with the training of students. For example, consider a typical freshman calculus course where students are taught, and assigned homework problems. On the exams, we test their ability to generalize by creating new problems they have not worked before. An "easy" problem is one similar to those explained in class; these type of problems do not demand as much generalization as the harder problems. How well students perform on these new problems indicates the level of generalization they have achieved. Ideally, that is how we assign grades for the class.

EXAMPLE 2

A now classical problem [WID1] involves backing an eighteen wheel truck to a loading dock. In the picture below, the goal is to back up the truck so that the dot at the back of the truck is next to the dot on this loading dock and so that the truck is in line with the dock.



Ideally, the goal is to train the driver so that she backs the truck to the loading dock independent of the initial position of the truck with respect to the loading dock. Further, when parking the truck, the driver wants to minimize the forward and backward (truck transmission in reverse) motion. Clearly, it is difficult to construct a function

$$\alpha : [0, b] \times X \longrightarrow X \times Y$$

satisfying the following:

I. X is the space of configurations of the truck,

II. $[0, b]$ represents time,

III. Y represents the parameters of how much the driver is pushing on the gas pedal, and brake, and also the position of the steering wheel,

IV. and $\alpha(b, x)$ corresponds to the truck being parked against the loading dock.

V. If we measure the path length in $\mathbb{R}^2$, defined by the truck while backing up to the loading dock, we want this length to be minimal.

Suppose x_0 is the initial configuration of the truck in the plane and we want to know what state the truck is in at time t. We see that $\alpha(t, x_0) = (x(t, x_0), y(t, x_0))$ where $x(t, x_0)$ is the instantaneous configuration of the truck on its way toward the loading dock and $y(t, x_0)$ is the instantaneous position of the steering wheel, the gas pedal and brake at time t.

Similar to the photography example, the problem is to design a network of simple computational units to compute α. Since α is not explicitly known, the truck starts from different initial configurations. Then the network drives the truck. Depending on how close the network matches the dot on the loading dock to the dot at the back of the truck, we make adjustments to these computational units after each test drive. This is an example of training the network to compute the function α.

SUMMARY

The two examples illustrate the notion of a neural network that is designed to calculate a function F. We call "computation" the actual calculation of the function F. We call "training" the design of the messages that the units of the network send to each other, in order to calculate the function F. We call "generalization" the ability of the network to perform on untrained examples.

WHY DYNAMICS IS IMPORTANT

Our first assumption, stated above, is that the network of computational units collectively compute a function F. The input group of computational units deter-

mine the domain of F, and the output group of computational units determine the range of F. The input group and output group of units may overlap, or they may be disjoint; the particular case depends on the wiring of the network.

To illustrate this assumption with our examples, the input units in the photograph example receive information about the light intensity in one of the small squares of the photograph. The hidden units collectively process the information coming from the input units, and send their messages to a single output unit. By aggregating the messages coming simultaneously from the intermediate units, the single output unit goes to 1 (Yes, it believes this is a photograph of Michael Jones) or 0 (No, it is not Michael Jone's face.)

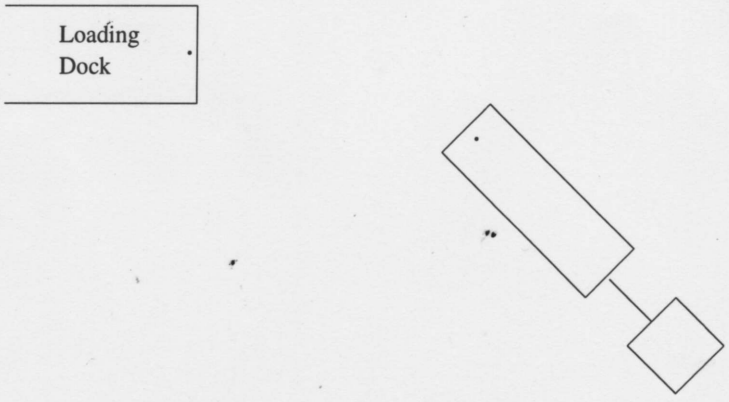
Here the domain of the computed function F is the finite Cartesian product $\prod_{i=1}^{n} X$ where X represents all possible light intensities, and n is the number of input units. The range of F is the two discrete points $\{0, 1\}$. Dynamics plays a critical role in the calculation of the function F. (A few of the many references using dynamics in the study of the computation of F are [HOP82], [HOP84], [HOP85], [HOP86], [HIR93].)

Another area where dynamics plays a critical role is during the training. As mentioned, training involves adjusting the messages sent by the units to each other so that the trained network will compute a function (F in Example 1, and α in Example 2) that effectively performs the desired task. (i.e. recognizing a picture, or driving an 18 wheeler.) Since we can not explicitly determine in advance the appropriate F or α, typically a teacher tells the network how well it has performed on a particular example. The teacher's evaluation is used by the network to adjust the messages that each unit passes to the other units. This adjustment is the

dynamics.

To see the connection with dynamical systems, think about the training in the following way. A unit of time corresponds to the adjustments made to the network during the training on one example. The process of adjustment involves the application of a training function to the network. (Notice that the training function is conceptually a different function from the computation functions F or α.) The image of this training function is the new messages that the units send to each other. By training the network on ten examples, we are applying a sequence of ten training functions. After each example, one training function is applied to the network in order to alter the messages that the units send to each other.

Suppose we train the network on ten different examples. An important question is: Are the training functions applied after each example the same function or different functions? Answers arise by examining Example 2. Suppose we have trained the network to adequately back the truck up to the dock for easy initial positions of the truck such as:



Suppose after the network is trained on the following example, it fails to park the truck even close to the loading dock.

Clearly, we do not want to make a large adjustment which could prohibit the network from being able to back up the truck for the easy initial positions. Therefore, we make a much smaller adjustment.

In summary, we see that the amount of training the driver has had, determines the size of the adjustments even though the initial position of the truck for say the 5th training example may be the same as the 700th example. Consequently, the trainings function applied to alter the messages that the units send to each other are, in general, different. These functions depend on the time, i.e. they depend on the training stage of the network.

We also see that the dynamics of the adjustment process influences the quality of the generalization. We do not want to make adjustments that enable the network to generalize for harder examples, but cause the network to fail on the easier examples. In summary, the adjustment process on the same example, performed at two different times is different, so this can be modeled by a non-autonomous dynamical system.