An Exercises Supplement

to the

Introduction To The Theory Of Neural Computation

J. Hertz, A. Krogh, and R.G. Palmer

Arun Jagota
University of California, Santa Cruz
email: jagota@cse.ucsc.edu
Includes Contributions from Connectionists

20 April 1995

Preface

This set of exercises and some computer projects is designed for the text book [1]. I thank Professor Palmer for encouraging me to announce it to Connectionists. I thank several people on the Connectionists mailing list who responded positively—and with warm sentiments—to my posting. Additional thanks to those who contributed questions, which I found very useful! Since these were not designed specifically for the HKP book, however, I have included them in a separate chapter at the end, with source indicated.

This supplement may be used any way one wishes to. For example, in a neural networks or related topics course. It may be distributed freely, in postscript or hardcopy form, in its entirety, or in portions. It should not be offered for sale, however. Any set of questions may be taken freely from this list (except from the Contributions chapter) and included in a new text book (or edition) one is writing. (Some people asked me regarding this possibility.) I need not be consulted or informed. For questions from the Contributions chapter, however, one should consult the source.

Most questions I generated have been "classroom tested". (Some of these questions are based on known results in the literature. I have deliberately excluded citing these results, to avoid giving any hints to students.) In the questions I generated, I use notation and refer to equations from the HKP book. Therefore, it will be useful to have the text book around. Contributed questions have been included with little change. Solutions to exercises have been excluded from this list deliberately, except for contributions from Lee Giles, which include pointers to solutions. A separate "Solutions to HKP exercises" document will be created this summer and made available to instructors.

All suggestions and errors should be sent to me at the e-mail or surface mail address given on the title page. I especially welcome new questions, and especially in chapters SIX (Multi-Layer Networks), SEVEN (Recurrent Networks), EIGHT (Unsupervised Hebbian Learning), and NINE (Unsupervised Competitive Learning). I hope to have a second expanded version ready sometime this summer.

Chapter 1

Excercises

ONE: Introduction

1.1 What changes would you propose to make the McCulloch-Pitts neuron more biologically realistic? In what specific ways would they enhance the information processing capabilities of the neuron? Speculate how this might help in computational applications.

1.2 The typical response time of real neurons is in milliseconds, roughly a million times slower than the response time of silicon gates in hardware. Why then has it turned out to be so difficult to build a computer that can compute as quickly and reliably as the brain can, for example in recognizing faces?

TWO: The Hopfield Model

In this chapter, unless stated otherwise, the number of neurons is denoted by n.

2.1 Describe to what extent the Hopfield model contains each of the "brain features desirable in artificial systems" discussed in Chapter ONE.

Hebb Storage Rule

This set of questions deals with storing bipolar vectors in the Hopfield model using Hebb type storage rules. The questions deal with stability of the stored vectors, and formation of spurious memories.

2.1.1 In (2.12), if the magnitude of the second term is less than 1 then unit i is stable for pattern ν. Suppose the constant $1/n$ in the Hebb rule (2.9) is replaced by $1/(np)$. Explain why the absolute value of the second term in (2.12) will then always be less than 1. Does this mean we have found a way to make unit i always stable, regardless of the stored patterns? Explain.

2.1.2 Explain the similarities and differences between the Hopfield pattern storage rule, given by (2.9), and the true Hebb principle.

2.1.3 Construct the smallest example you can (choose n, p, and the p patterns $\xi^1, \ldots, \xi^p$) so that, after storage in the Hopfield model according to (2.9), not all patterns are stable.

2.1.4 Construct the smallest example you can (choose n, p, and the p patterns $\xi^1, \ldots, \xi^p$) so that, after storage in the Hopfield model according to (2.9), there are spurious stable states other than $-\xi^\mu$, for every $\mu = 1, \ldots, p$. You must not choose $p = n$ and $\xi^1, \ldots, \xi^p$ as pairwise orthogonal patterns.

2.1.5 Prove that when $n = 2$, for any collection of patterns stored in the Hopfield model using the Hebb rule (2.9), every spurious stable state, if there is any, is the negative of some stored pattern. (The negative of a vector ξ is $-\xi$.)

2.1.6 Consider storage rule (2.9), and modify it slightly as follows. *After* all the patterns $\xi^1, \ldots, \xi^p$ are stored using (2.9), make the self-weights $w_{ii} = 0$ for all i. We claim that, for all $n \geq 2$, we can find just *two* patterns $\xi^1, \xi^2 \in \{-1, +1\}^n$ such that storing these two patterns using the modified rule causes at least one of them to become unstable. Find a set of two patterns to verify this claim. Give a single description of these patterns that holds for all $n \geq 2$.

Update Rules and Energy Functions

The following questions deal with the asynchronous and synchronous update rules for the Hopfield model, and the role of energy functions. They apply to all instances of a Hopfield network, not just to those arising from storing bipolar vectors using any particular storage rule.

2.2.1 Construct as simple a Hopfield Network as you can (you choose n and the weight matrix W) such that W is asymmetric (i.e., there is at

least one pair $i \neq j$ such that $w_{ij} \neq w_{ji}$) and the network does not always converge to a stable state, under asynchronous updates. For all i, assume that w_{ii} and θ_i are equal to 0.

2.2.2 The energy function H, given by (2.24), is proven to decrease, whenever a unit switches asynchronously, only for the update rule (2.4). Find an energy function for (2.2) and show that it also decreases under asynchronous updates.

2.2.3 Let $f(n)$ be the smallest number $k = k(n)$ such that in every n-neuron Hopfield network with $w_{ii} = 1$ and $\theta_i = 0$ for all i and $w_{ij} = w_{ji} \in \{-1, 1\}$ for all $i \neq j$, from every initial state, every asynchronous energy-decreasing trajectory according to (2.4) terminates in $\leq k(n)$ unit-switches. (In other words, $f(n)$ is the maximum number of unit-switches to convergence in any n-neuron network of the above kind.) Compute the best upper bound on $f(n)$ that you can.

2.2.4 In the same setup as question 2.2.3, try to compute, by construction, a good lower bound on $f(n)$. In particular, try to construct a Hopfield network instance with $w_{ii} = 1$, $\theta_i = 0$ for all i, and $w_{ij} = w_{ji} \in \{-1, 1\}$ for all $i \neq j$ in which there is at least one initial state from which the number of unit-switches to convergence is greater than or equal to $f(n)$ along some energy-decreasing trajectory. Try to maximize $f(n)$. Can you find a network in which $f(n)$ is superlinear? How about just $f(n) > n$?

2.2.5 Consider question 2.2.3 again, but with several restrictions on the weights and thresholds removed. Specifically, $w_{ii} \in \Re^+ \cup \{0\}$ and $\theta_i \in \Re$ for all i, and $w_{ij} = w_{ji} \in \Re$ for all $i \neq j$. Give a naive upper bound on $f(n)$.

2.2.6 Consider the Hopfield model with symmetric weights $w_{ij} = w_{ji}$ for all $i \neq j$ and $w_{ii} \geq 0$ for all i. It is well known that if (2.4) is applied synchronously, with all units updated simultaneously at each time step, the network converges either to a stable state or to an oscillation between two states. The interesting aspect of this result is that it rules out higher order cyclic behavior. Construct the simplest network instance you can which converges to an oscillation between two states, from some initial state.

2.2.7 Prove the fact cited in question 2.2.6, namely that, under synchronous updates, a Hopfield network with symmetric weights and nonnegative

self-weights converges to a fixed point, or to a cycle of period two.

Miscellaneous

2.3.1 (Implementation) Consider the n-neuron Hopfield model with symmetric weights $w_{ij} = w_{ji}$, and $w_{ii} = \theta_i = 0$ for all $i = 1, \ldots, n$. Employ 0/1 neurons instead of the -1/+1 neurons of Section 2.2. The dynamics of the network is given by (2.4) except that sgn means the following instead: $sgn(x) = 1$ if $x \geq 0$; 0 otherwise.

Consider the following sequential algorithm to implement serial energy-descent. (We call it serial energy-descent because it allows asynchronous updates as a special case, but is a little more general.) At one time, one unit is picked such that flipping its state, according to the modified update rule (2.4), would reduce energy, and this unit's state is flipped. If no such unit exists, the network is at a stable state.

Write out this sequential algorithm for serial updates in pseudocode. Store the weight matrix W in a two-dimensional $n \times n$ array, and the initial state $\vec{S}$ in a one-dimensional n-element array. **Make the implementation as efficient as possible**. During each update, you may select the unit to update as you wish, as long as flipping its state reduces energy, whenever possible. (Judicious choice of the unit to pick to switch may impact the efficiency of your implementation.) You may also need to use some clever algorithmic tricks and/or additional data structures to make the implementation most efficient.

2.3.2 Consider two Hopfield networks with the same number of neurons n. Call them *different* if at least one of their stable states is different, *equivalent* otherwise. Consider all possible n-neuron Hopfield networks in which $w_{ii} = \theta_i = 0$ for all i and $w_{ij} = w_{ji} \in \{-1, 1\}$ for all $i \neq j$. What is the total number of such networks? What is the largest collection of pairwise different networks in the total set? An exact answer is perhaps very difficult for arbitrary n. Try to find an exact answer for $n = 2, 3, 4$. Try to find a good lower bound for arbitrary n.

THREE: Extensions of the Hopfield Model

3.1 Review all the dynamical update rules for the discrete and continuous Hopfield model presented in Chapters TWO and THREE. Explain, intuitively, how they differ in their properties, and in their applications.

3.2 In the Willshaw rule (3.29), the neuronal thresholds θ_i are not explicitly given—it is only indicated that they need to be optimized. Consider storing a collection of 0/1 vectors of length n, each containing exactly k ones, using (3.29). What is the best choice for the thresholds θ_i in this case? Justify your answer.

3.3 Notice that the energy function of the continuous Hopfield model, (3.33), bears some resemblance to the energy function for the discrete model, (2.24). Are their local minima related in any way?
Hint: In (3.33), replace $g^{-1}(V)$ by $g_\lambda^{-1}(V)$ to make the dependence of (3.33) on λ explicit.

FOUR: Optimization Problems

In each of the following questions, map the problem in question to the discrete Hopfield model, using 0/1 or -1/1 neurons. Show all the steps in achieving the mapping: choose the 0/1 or -1/1 variables, formulate the problem's objective and constraints in terms of these variables, construct the energy function, and, finally, derive the weights and biases of the network. Whenever possible, try to ensure (and prove when possible) that local minima represent only feasible solutions of the mapped problem.

4.1 This is one version of the *map coloring* problem. The number of colors k is fixed in advance.

INPUT: A map on n states and k colors $\{1, 2, \ldots, k\}$.

FEASIBLE SOLUTION: A coloring of the map, satisfying the following constraints. A state is colored with one color from $\{1, 2, \ldots, k\}$, or not colored at all. If two colored states are adjacent, they must not have the same color.

COST OF A FEASIBLE SOLUTION: Negative of the number of states colored.

OBJECTIVE: Find a feasible solution with minimum cost, i.e., with the maximum number of colored states.

4.2 This is the n-queens problem.

INPUT: An $n \times n$ chess board and n queens.

FEASIBLE SOLUTION:A placement of some—not necessarily all—the queens on the chess-board so that no two placed queens are on the

same row, column, left diagonal, or right diagonal. In other words, no queen attacks another.

COST OF A FEASIBLE SOLUTION: Negative of the number of queens placed.

OBJECTIVE: Find a feasible solution with minimum cost, i.e., with the maximum number of queens placed in non-attacking positions.

4.3 This is a *shortest path* problem.

INPUT: An undirected graph $G = (V, E)$ with weights $d_{ij} = d_{ji} \geq 0$ on the edges $\{v_i, v_j\}$, and two vertices s and d.

FEASIBLE SOLUTION: A path from s to d which does not visit any vertex, including s, more than once.

COST: Sum of the edge-weights on the path.

OBJECTIVE: Find a feasible solution with minimum cost.

FIVE: Simple Perceptrons

5.1 Consider all boolean functions $f : \{0, 1\}^2 \rightarrow \{-1, 1\}$. Which of them are linearly separable? For each of the linearly separable ones, hand-construct a simple perceptron with two input units and one output unit with a hard threshold activation function. Use any real-valued threshold θ for the output unit.

5.2 The XOR function (Table 5.2, page 96), cannot be represented in a simple perceptron of two input units and one output unit with a hard threshold activation function, with any threshold θ. What if a linear output unit is used? Can the XOR be represented then? Explain precisely why or why not.

5.3 Consider a simple perceptron with N input units, and M output units, each with a hard threshold activation function (whose range is -1/+1) with threshold $\theta_i = 0$. Restrict each of the weights to have the value -1 or +1. How many functions from $\{-1, 1\}^N$ to $\{-1, 1\}^M$ are representable in this family of networks? Prove your answer.

5.4 Consider a linearly separable set of pairs (ξ^μ, ζ^μ), where $\xi^\mu \in [0, 1]^N$ and $\zeta^\mu \in \{-1, 1\}$. If, along with the pairs (ξ^μ, ζ^μ), D_{max} is known in advance, describe how one might learn the weight vector w^* of

an *optimal perceptron*. D_{max} is defined in (5.25), and the concept of optimal perceptron immediately above it.

5.5 Consider the proof of convergence of the Perceptron Learning Rule in Section 5.3, page 100. Set $\eta = 1$ and $\kappa = 0$. Let $\{w_t\}$ denote the sequence of evolutions of the weight vector. That is, if $x^\mu \equiv \zeta^\mu \xi^\mu$ is misclassified when the weight vector is w_{t-1}, then $w_t := w_{t-1} + x^\mu$.

Construct a geometric visualization of the initial part of this proof, for $x^\mu \in \Re^2$. Choose a set of points x^μ in the plane such that a perceptron solution exists, but is not trivial to find. Draw the vectors w^*, w_t, w_{t-1}, and x^μ. Draw $\langle w_t, w^* \rangle / |w_t|$. Do not carry out the geometric visualization beyond this point, as it gets a little complicated.

5.6 In the example of Figure 5.6, the learning algorithm quickly finds a solution. Try to construct an example in which the learning algorithm takes much longer to find a solution. Choose $\kappa = 0$ and $\eta = 1$, as in Figure 5.6.

5.7 Consider a simple perceptron with one linear output unit (Section 5.4, page 102). Derive the Gradient Descent Learning Rule for the following cost function:

$$E[w] = \sum_\mu \left(\frac{1+\zeta^\mu}{2} \log_e \frac{1+\zeta^\mu}{1+O^\mu} + \frac{1-\zeta^\mu}{2} \log_e \frac{1-\zeta^\mu}{1-O^\mu}\right)$$

Show all steps, upto the online version analogous to (5.42) and (5.43).

5.8 As indicated in Section 5.2, the decision boundary of a simple perceptron with a hard threshold output neuron is a hyperplane (points on one side of the hyperplane are classified +1; points on the other side -1). What is the decision boundary if we allow one hidden layer of neurons with hard threshold activation functions? How about two hidden layers with hard threshold activation functions? In either case, there is a single hard threshold output neuron.

SIX: Multi-Layer Networks

6.1 What are the limitations of one-layer perceptrons with hard threshold output units? How are these limitations overcome with multi-layer feedforward networks, whose units employ continuous activation functions of certain kinds. What are the essential properties of these activation functions?

6.2 Hidden units in multi-layer feedforward networks can be viewed as playing one or more of the following roles:

1. They serve as a set of basis functions.
2. They serve as a means of transforming the input space into a more suitable space.
3. Sigmoidal hidden units serve as approximations to linear threshold functions.
4. Sigmoidal hidden units serve as feature detectors with soft boundaries.

Explain the benefits of the individual roles in helping us understand the theory and applications of multi-layer networks. Which roles help understand which theory, and which roles are insightful for which applications?

6.3 Which of these two-layer feedforward networks is more powerful: one whose hidden units are sigmoidal, or one whose hidden units are linear? The output units are linear in both cases. Explain your answer.

Chapter 2

Computer Projects

The following neural network projects are well-suited to implementation in `Mathematica` or a similar package. Alternately, one might program them in a standard language like C or C++. Or use a simulator, if available.

The Hopfield Model

Hebb Rule and Capacity For $n = 100$ and $n = 200$, conduct the following experiment. For each n, repeat the experiment independently five times, and take the average of the five runs. Implement the Hebb rule (2.9) on page 16. Store $0.15n$ random patterns. Experimentally determine the percentage of bits in the stored patterns that are stable. Compare your experimental results with the theoretical result on pages 18-19.

Asynchronous Retrieval Implement the asynchronous updates version of (2.2), page 13. For $n = 500$, conduct the following experiment. Store $\lfloor n/(4 \log_2 n) \rfloor$ random patterns in the network using the Hebb rule (2.9). ¿From amongst the stored patterns, choose ten patterns ξ^μ, $\mu = 1, \ldots, 10$, arbitrarily, and from each ξ^μ generate a pattern Y^μ by flipping $n/2 - 1$ random bits of ξ^μ. For $\mu = 1, \ldots, 10$, input Y^μ to the network and apply asynchronous updates. Record in what fraction of the cases the correct pattern ξ^μ is retrieved. Also record the number of unit-switches in each case from the initial state to the final state, and report the average, over the ten cases.

Combinatorial Optimization Extend your Hopfield model implementation to allow all the weights and thresholds to be set to any values from

the input. For $\mu = 0, 1/2$, and 2, using (4.15), create three separate Hopfield networks (one by one) for the graph bipartitioning example in Figure 4.6, page 80. Run each network in asynchronous update mode from the initial state $S = \vec{0}$, and record the final state. Record the quality and feasibility of the solution represented by the final state, and note how it changes with μ.

Perceptron

Implement a simple perceptron with one output unit using the hard threshold activation function (with -1/+1 range) of Section 5.2, page 92. There are n input neurons, each taking a 0 or 1 value. The perceptron has n weights $w_1, \ldots, w_n$, and a threshold θ. Implement the simple learning algorithm (5.17) on page 97. (You may use $n + 1$ input neurons and a threshold of zero.) Perform the following experiments, with $n = 2$:

1. Learn each of the linearly separable boolean functions from $\{0, 1\}^2$ to $\{-1, +1\}$. Use the truth table of a function as the training set. Set all weights to zero initially, and the learning rate to $\eta = 1/2$. Record the time it takes to learn each function perfectly, measured by the number of weight adjustments according to (5.17) that occur. Does the training time depend on the function being learnt? Report on any interesting observations in this regard.

2. Repeat the above experiment with $\eta = 1$. Do the results change in any significant way? Elaborate.

3. Compare the networks that were learnt in the computer projects with the networks that were hand-constructed in answering question 5.1.

Back-Propagation

Implement the Back-Propagation algorithm on a one-hidden-layer network, as described on page 120. Choose g to be a sigmoid. Conduct the following experiments.

1. Learn the XOR problem using the network topology of Figure 6.5a, page 131.

2. Learn the 4-Parity problem using the network topology of Figure 6.6, page 131.

3. Learn the encoder problem using the network of Figure 6.7, page 132.

In all the above cases, conduct experiments with different values of η and record the number of epochs it takes to learn the function for each value. How does the number of epochs change with η? Do your experimental results reveal which functions are easy to learn, and which difficult?

Chapter 3

Contributions

Guszti Bartfai, ⟨ Guszti.Bartfai@Comp.VUW.AC.NZ ⟩

1. A perceptron can in general classify patterns of real numbers. The common task of converting a sampled analogue signal, which is a sequence of one-dimensional patterns, into digital can also be viewed as a collection of simultaneous classification tasks. Here we define an N-bit *Analogue-to-Digital Converter* (ADC) that converts an input signal $x \in \mathcal{R}$ in the range of $0 \leq x < 2^N$ into an N-bit unsigned binary number. More precisely, for a given input value x, the output binary vector ($\mathbf{y} = (y_{N-1}, y_{N-2}, ..., y_0)$, $y_i \in \{0, 1\}$) will be such that

$$l \leq x < l + 1,$$

where

$$l = y_{N-1} \cdot 2^{N-1} + y_{N-2} \cdot 2^{N-2} + ... + y_0 \cdot 2^0.$$

In other words, the output will be a quantised, binary coded version of the input.

   (a) Show that the N-bit ADC ($N \geq 2$) function defined above cannot be computed with a *single-layer* perceptron.

   (b) Describe a method of constructing a *multi-layer* perceptron (MLP) that implements an N-bit ADC. Explain the role of each hidden layer in your network. Derive formulas for the number of layers,

number of nodes in each layer, and the total number of connections in your network. Try to find the MLP ADC with the *minimum number of nodes*!

(c) Choose a different binary encoding of the input range $[0, 2^N)$ so that the resulting function – still implementing an ADC essentially – will be computable by a *single-layer* perceptron. Show your solution by constructing a corresponding ADC network.

L.S. Smith, ⟨ l.s.smith@cs.stir.ac.uk ⟩

These questions are offered as a means for stimulating discussion, especially in small classes, rather than as worked out examples.

1. A set of measurements of leaves have been taken, with a view to performing classification of these leaves. There are 40 measurements of each leaf, ranging from maximal length, maximal width, perimeter length, length of longest vein, colour, transparency, etc. Measurements have been taken from a limited number of leaves.

   Discuss the problems such high-dimensional data might present for classification using a backpropagated Delta-rule network. Might PCA techniques be used to alleviate such problems?

   How might one use this data in an LVQ network? Would reducing its dimensionality using PCA be useful in this case?

2. A remote sensing system on a satellite measures incident radiation from a small portion of the Earth's surface in 5 bands, these being near infrared, red, green, blue, and near ultraviolet. The sensors have some degree of overlap. The system must transmit the values back to the Earth-station. However, the link between satellite and station is noisy and of low bandwidth. Interest is not in the signals themselves, but in using the signals to find out whether they originated from sea, tree-cover, vegetation, cities, or bare hillside. Discuss how one might send the data, and process it at the groundstation.

3. The Backpropagation algorithm, and the Boltzmann machine algorithm cover similar ground. Are they equivalent? Are there problems for which one algorithm is superior?

4. Discuss how one might apply feed-forward neural networks to the following problem:

A system is required to attempt to foretell oncoming failures in a large motor inside a gas pumping station. The system has a number of sensors, measuring

```
temperature of the motor rotor
speed of the motor rotor
gas flow rate
vibration level sensor
```

In addition, there is a microphone which can be used to monitor the sound produced by the motor.

In normal operation the motor will run continuously: however, for the purposes of generating the test data, the motor can be started and stopped, and a variety of faults induced.

5. A bank is currently using a rule-based system to decide which of its customers to send sales information about some investment services it is trying to sell. The rules they are using are based on information known to the bank about its customers: namely,

```
customer age and sex
transactions and balance in customer's accounts
```

and the rules they use initially are very simple: if the account holder is over 30, and has a monthly or weekly income exceeding some amount, and keeps his accounts generally in credit, then send the information. This system has been running for two years, and the bank has kept records of which customers have followed up the initial sales information.

Imagine that you are an agent for a company selling neural network based products and services. Explain how you might try to convince the bank that a neural network solution might improve their "hit rate", (i.e. the fraction of those who receive leaflets and then follow them up), and discuss the form of neural network based solution you might suggest.

Steve Gallant, ⟨ sg@belmont.com ⟩

There is a large collection of questions in his book (MIT Press: "Neural Network Learning and Expert Systems") whose copyright is owned by MIT Press.

Ravi Kothari, ⟨ rkothari@ece.uc.edu ⟩

This is a question for Chapter NINE of the HKP text.

1. True color images are stored as 24 bits/pixel (8 bits each for Red, Green, and Blue) allowing for a maximum of 2^{24} colors. While, true color images are photo realistic, they also have large storage requirements. Often it is desirable to reduce the total number of colors to a lesser number of colors (256 for example) so as to produce a minimal amount of visual color distortion.

   Finding the most frequently occuring colors, does not lead to acceptable results since an infrequently occuring color may be very important to the overall image (e.g., a small potted plant in a peach colored room – green occurs in only a small patch, but is a crucial part of the overall image). A plausible method of finding the set of reduced colors is based on the fact that many of the 2^{24} colors are clustered around a few areas. An appropriate method would sample the reduced number of colors from the Red-Green-Blue (RGB) space to reflect the probability density function in the RGB space – a task well performed by Self Organizing Feature Maps.

   In this exercise, write a Self Organizing Feature Map (SOFM) program to find a reduced color representation of an image ($2^n colors \rightarrow 2^m$ colors, $m << n$).

   Hint: The SOFM will have a total of m neurons, and will accept three 8 bit inputs – the Red, Green and Blue components of the color of a pixel. Denote the weights of neuron m, as $W^m = [w_r^m w_g^m w_b^m]'$, and the pixel color as $I^i = [I_r^i I_g^i I_b^i]'$. Training of the network consists of sampling each pixel from the original image, finding the winner, and adjusting the weights of the winner and the neighborhood. At the end of the training process, the m weight vectors have a number

density reflecting the probability density function of the original image colors. At the end of the training process, each pixel color can be represented by the index of the winning neuron when it is presented as input. Assuming $n = 24$, and $m = 256$, each pixel now needs 8 bits as oppposed to the original 24 for representation.

Geoff Goodhill, ⟨ geoff@salk.edu ⟩

Tutorial exercise: Linsker's model

Background

Linsker's model for the development of certain types of receptive fields in the visual system has generated much interest in the neural networks literature. This is because it shows how a series of layers of simple units, evolving with a simple learning rule, can generate progressively more complicated sets of feature detectors - given entirely *random* activity in the first layer.

The model was originally presented in a series of rather technical papers [7]. A later article [8] reviewed this work, and went on to discuss optimization and information-theoretic properties of the model, as well as a number of more general issues. The aim of this exercise is to implement a simple version of Linsker's model, in order to get a feel for its properties and its sensitivity to parameter values.

The original model has several layers. However, this isn't strictly necessary: being a *linear* network, actually just two layers are sufficient to generate all the interesting behaviour. Linsker suggests that a good reason for using more than two layers is that structures such as oriented receptive fields are more stable to parameter variations when there are several intermediate layers. (However, another reason he adopted the multilayer approach could be that he understood very well the publicity value of a system that so closely matches the architecture of the natural vision system.) For simplicity, we will implement the model with two layers, assuming gaussian correlations in the first layer (i.e. we just consider Linsker's layers B and C).

We will also make the same assumption as Linsker used to speed up learning: rather than simulating a whole layer of developing units, we will just consider one unit. This is reasonable because there are no interactions between units in the same layer, and all units in a layer evolve simliar receptive fields.

The Program

Here are some hints about how to write the program. See [7, 8] if anything is unclear.

Write the program with the following main data structures:

- A **weight array** w containing the weights from each input unit to the single output unit. Assume a single type of synapse which can be both excitatory and inhibitory, bounded between upper and lower limits $[-0.5, 0.5]$.

- A **correlation matrix** Q that gives the correlations between pairs of units in the input layer. Assume these correlations are gaussian, with "spread" σ_c (as in layer B of Linsker's original model). That is, for two input units with positions (indices) (i, j) and (k, l) in the input array, the correlation Q_{ijkl} between them is

  $$Q_{ijkl} = e^{-\sigma_c\{(i-k)^2+(j-l)^2\}}$$

  (Remember Q is "really" two dimensional: it's just easier to use 4 indices since the inputs form a square array).

- An **arbor matrix**. This defines the receptive field of the output cell by multiplying each weight by a number ≤ 1 that decreases to zero with distance from the centre of the receptive field. Assume this matrix has value 1 in the centre of the input array, and drops off as a gaussian with "spread" σ_a away from the centre.

Firstly, initialize the correlation matrix and the arbor matrix. Set the initial weights to be small random values (both positive and negative - e.g. between -0.1 and $+0.1$). Then iterate the equation for the development of the weights:

$$\dot{w} = (\mathsf{Q} + k_2\mathsf{J})w + k_1 n$$

where J is the matrix $J_{ij} = 1,\ \forall\ i, j$; n is the vector $n_i = 1,\ \forall\ i$, and k_1 and k_2 are constants.

(I used a simple Euler-type iteration with step size of 0.025. This means I iterated the following:

- Calculate the value of the right hand side of the equation for the current weight values.

- Add in 0.025 times this value to the current weight vector.

See a textbook on Numerical Analysis if you want further information on this, and more sophisticated methods).

I printed out the weight matrix every 250 iterations. Don't forget to multiply weights by the arbor function when calculating the right hand side of the above equation, and to limit the weights between the upper and lower bounds. Start with an input layer of size 7 by 7: I found this took a couple of minutes to run (on a Sparc1). To start with try parameter values $\sigma_c = 0.15$, $\sigma_a = 0.1$.

Results

The crucial parameters for controlling the receptive fields produced are k_1 and k_2. Explore the parameter space. Here are some examples of the weight-matrices I produced:

$k_1 = 0.8$, $k_2 = -3.0$ gives an all-excitatory receptive field:

```
Iterations = 0
 -0.15  0.12  0.02 -0.20 -0.04  0.05 -0.19
  0.17 -0.06  0.14  0.02 -0.06  0.04 -0.10
  0.19  0.08  0.08 -0.04  0.16  0.11  0.16
 -0.02  0.04  0.13 -0.09 -0.08  0.07 -0.18
 -0.14 -0.03 -0.00  0.19  0.14  0.03  0.15
  0.03 -0.01  0.03  0.18 -0.07  0.18  0.13
 -0.04 -0.14 -0.17 -0.10  0.00  0.01 -0.01

Iterations = 250
  0.50  0.50  0.50  0.50  0.50  0.50  0.50
  0.50  0.50  0.50  0.50  0.50  0.50  0.50
  0.50  0.50  0.50  0.50  0.50  0.50  0.50
  0.50  0.50  0.50  0.50  0.50  0.50  0.50
  0.50  0.50  0.50  0.50  0.50  0.50  0.50
  0.50  0.50  0.50  0.50  0.50  0.50  0.50
  0.50  0.50  0.50  0.50  0.50  0.50  0.50
```

$k_1 = -0.8$, $k_2 = -3.0$ gives an all-inhibitory receptive field:

```
Iterations = 250
 -0.50 -0.50 -0.50 -0.50 -0.50 -0.50 -0.50
 -0.50 -0.50 -0.50 -0.50 -0.50 -0.50 -0.50
 -0.50 -0.50 -0.50 -0.50 -0.50 -0.50 -0.50
 -0.50 -0.50 -0.50 -0.50 -0.50 -0.50 -0.50
 -0.50 -0.50 -0.50 -0.50 -0.50 -0.50 -0.50
 -0.50 -0.50 -0.50 -0.50 -0.50 -0.50 -0.50
 -0.50 -0.50 -0.50 -0.50 -0.50 -0.50 -0.50
```

$k_1 = 0.0$, $k_2 = -3.0$ gives an oriented receptive field:

```
Iterations = 15000
 -0.50 -0.50 -0.50 -0.50  0.14  0.50  0.50
 -0.50 -0.50 -0.50 -0.50  0.50  0.50  0.50
```

```
-0.50 -0.50 -0.50 -0.49  0.50  0.50  0.50
-0.50 -0.50 -0.50  0.01  0.50  0.50  0.50
-0.50 -0.50 -0.50  0.49  0.50  0.50  0.50
-0.50 -0.50 -0.50  0.50  0.50  0.50  0.50
-0.50 -0.50 -0.15  0.50  0.50  0.50  0.50
```

Centre-surround receptive fields are much harder to produce. Here's the best I could get ($k_1 = 0.45$, $k_2 = -3.0$). Do let me know if you can do any better!

```
Iterations = 37500
 -0.50 -0.50 -0.50 -0.50 -0.50 -0.50 -0.50
 -0.50 -0.50 -0.50  0.50  0.50 -0.50 -0.50
 -0.50 -0.50  0.50  0.50  0.50  0.50 -0.50
 -0.50  0.50  0.50  0.50  0.50  0.50 -0.50
 -0.50  0.50  0.50  0.50  0.50  0.50 -0.50
 -0.50  0.50  0.50  0.50  0.50 -0.50 -0.50
 -0.50 -0.50 -0.50  0.09 -0.50 -0.50 -0.50
```

A detailed theoretical analysis of the effects of k_1 and k_2 on the final results can be found in [9, 10]: read these if you want an explanation in terms of competing eigenvectors.

Linsker's system has sometimes been criticised for being very sensitive to it's parameters. Do you think this is a valid criticism? That is, (1) is it true, and (2) would that be a good reason for having doubts about the usefulness of Linsker's model?

Lee Giles,⟨ giles@research.nj.nec.com ⟩

This is a set of problems for Recurrent Networks (not relaxation nets, but dynamically driven ones). Each problem is followed by a pointer to its solution. This exception has been made because of the nature of the questions, and because the pointers came with the questions.

1. Derive and contrast the complexity both in space and time for back-propagation through time and real-time recurrent learning training algorithms. Make sure the complexity considers whether or not the networks are fully or partially connected.

   (Solution: see [6]).

2. Derive how a recurrent network can be trained with an extended Kalman estimator. What is the space and time complexity of this training algorithm?

   (Solution: see [5, 3])

3. How do recurrent networks compare to FIR and IIR filters? Show architectural examples. Derive appropriate training algorithms.

   (Solution: see [4])

4. Learn the parity grammar (all strings with an odd number of ones are accepted, all others rejected) with a recurrent network and an appropriate training algorithm. Discuss how the strings of 0s and 1s will be encoded and how the network will do the training. Discuss the minimum training set needed. How well will the trained network perform on the unseen strings? Is the network stable?

   (See for example [2])

Daniel Nikovski, ⟨ danieln@cs.siu.edu ⟩

Here is an excerpt from his message, explaining his contribution:

> I wrote a small part of a book, for which I prepared six simple exercises. As they were not included in the final version of the book, there would not be any copyright violation if I send them to you. Two of the problems are on Bernoulli units, introduced by Ron Williams, and not discussed in HKP. The last problem is on Oja's rule.

1. Consider the A_{RP} algorithm with a 5 dimensional output vector $\mathbf{y} = (y_1, y_2, y_3, y_4, y_5)$, where positive reinforcement is given only when $y_1 = 1, y_2 = -1, y_4 = 1, y_5 = 1$ (the value of y_3 does not matter). What is the best ratio between the learning rates for positive reinforcement $\eta(+1)$ and negative reinforcement $\eta(-1)$? (Hint: calculate the probability of positive reinforcement by random guessing.)

2. Design a dynamic reinforcement function for Problem 1 that initially returns positive rewards for approximately half of the output vectors, and subsequently shrinks to the region, described in Problem 1. Update the learning rates $\eta(+1)$ and $\eta(-1)$ accordingly. (Hint: Start with a function that monitors the value of y_1 only, and then add y_2, y_4 and y_5 later.)

3. Implement the algorithms from Problems 1 and 2 and compare the convergence speed. Consider the algorithms to have converged when the average reinforcement of the latest 100 iterations exceeds 0.95, i.e. positive reinforcement is obtained in at least 95 trials out of 100.

4. Implement a single Bernoulli unit network with output y and no inputs. Positive reinforcement $r = 1$ will be given for $y = 1$, while negative reinforcement $r = -1$ will be given for $y = 0$. Calculate the reinforcement baseline b as a moving average of the last 5 reinforcement values.

5. Add a second Bernoulli output unit to the one in Problem 4. This time use the following function for the reinforcement signal:

$$r = 2(\alpha y_1 + (1 - \alpha)y_2) - 1$$

$$\alpha \in [0, 1]$$

   Here the parameter α models the relative importance of each of the outputs. Note that for each value of α, only four values for the reinforcement signal are possible: $1, 2\alpha-1, 1-2\alpha, -1$. Plot the dependence between the convergence speed and α. Compare the results from Problem 4 with simulations for α close to 1 and 0. Why is the difference in convergence speed small?

6. Compute the covariance matrix C of the following two vectors:

$$\mathbf{x^1} = (3, 4)$$
$$\mathbf{x^2} = (12, 5)$$

   Compute the eigenvalues and corresponding eigenvectors of the covariance matrix. Why is only one of them not zero? What is the direction of the first principal component? How many principal components are there for only those two vectors $\mathbf{x^1}$ and $\mathbf{x^2}$?

7. Implement Oja's rule for the two vectors from Problem 6. Normalize the two vectors to unit length and zero-center them by subtracting the average vector (μ_1, μ_2) from each of them. Start with a weight vector (ω_1, ω_2) of unit length too. To what value does the weight vector converge? Compare the results with the first principal component, discovered analytically in Problem 6.

Bibliography

[1] J. Hertz, A. Krogh, and R.G. Palmer. *Introduction to the Theory of Neural Computation*. Addison-Wesley, 1991.

[2] C.B. Miller and C.L. Giles. Experimental comparison of the effect of order in recurrent neural networks. *International Journal of Pattern Recognition and Artifical Intelligence*, 7(4):849–872, 1993. Special Issue on Neural Networks and Pattern Recognition, editors: I. Guyon , P.S.P. Wang.

[3] G.V. Puskorius and L.A. Feldkamp. Neurocontrol of nonlinear dynamical systems with kalman filter-trained recurrent networks. *IEEE Transactions on Neural Networks*, 5(2):279–297, 1994.

[4] A.C. Tsoi and A. Back. Locally recurrent globally feedforward networks, a critical review of architectures. *IEEE Transactions on Neural Networks*, 5(2):229–239, 1994.

[5] R.J. Williams. Some observations on the use of the extended kalman filter as a recurrent network learning algorithm. Technical Report NU-CCS-92-1, Computer Science, Northeastern University, Boston, MA, 1992.

[6] R.J. Williams and D. Zipser. Gradient-based learning algorithms for recurrent connectionist networks. Technical Report NU-CCS-90-9, Computer Science, Northeastern University, Boston, MA, 1990.

[7] Linsker, R. (1986). From basic network principles to neural architecture (series). *Proc. Nat. Acad. Sci., USA*, **83**, 7508-7512, 8390-8394, 8779-8783.

[8] Linsker, R. (1988). Self-organization in a perceptual network. *Computer*, March, 105-117.

[9] MacKay, D.J.C. & Miller, K.D. (1990). Analysis of Linsker's simulations of Hebbian rules. *Neural Computation,* **2**, 169-182.

[10] MacKay, D.J.C. & Miller, K.D. (1990). Analysis of Linsker's application of Hebbian rules to linear networks. *Network*, **1**, 257-297.