Artificial Neural Networks (ANN)

Patricia J Riddle

Computer Science 760

Artificial Neural Networks

Robust approach to approximating real-valued, discrete-valued, and vector-valued target functions

For certain types of problems (complex real-world sensor data), ANN are the most effective learning methods currently known

- Learning to recognise handwritten characters, spoken words, or faces

Inspiration

Biological learning systems are built of very complex webs of interconnected neurons

ANNs

- Built of a densely interconnected set of simple units

- Each unit takes a number of real-valued inputs (possibly the outputs of other units)

- And produces a single real-valued output (which may become the input to many other units)

Human Brain

10^{11} densely interconnected neurons

Each neuron connected to 10^4 others

Switching time:

- Human 10^{-3} seconds
- Computer 10^{-4} seconds

10^{-1} seconds to visually recognise your mother

- So only a few hundred steps
- Must be highly parallel computation based on distributed representations

Researchers

Two Groups of Researchers:

Use ANN to study and model biological learning processes

Obtaining highly effective machine learning algorithms

Neural Network Representations

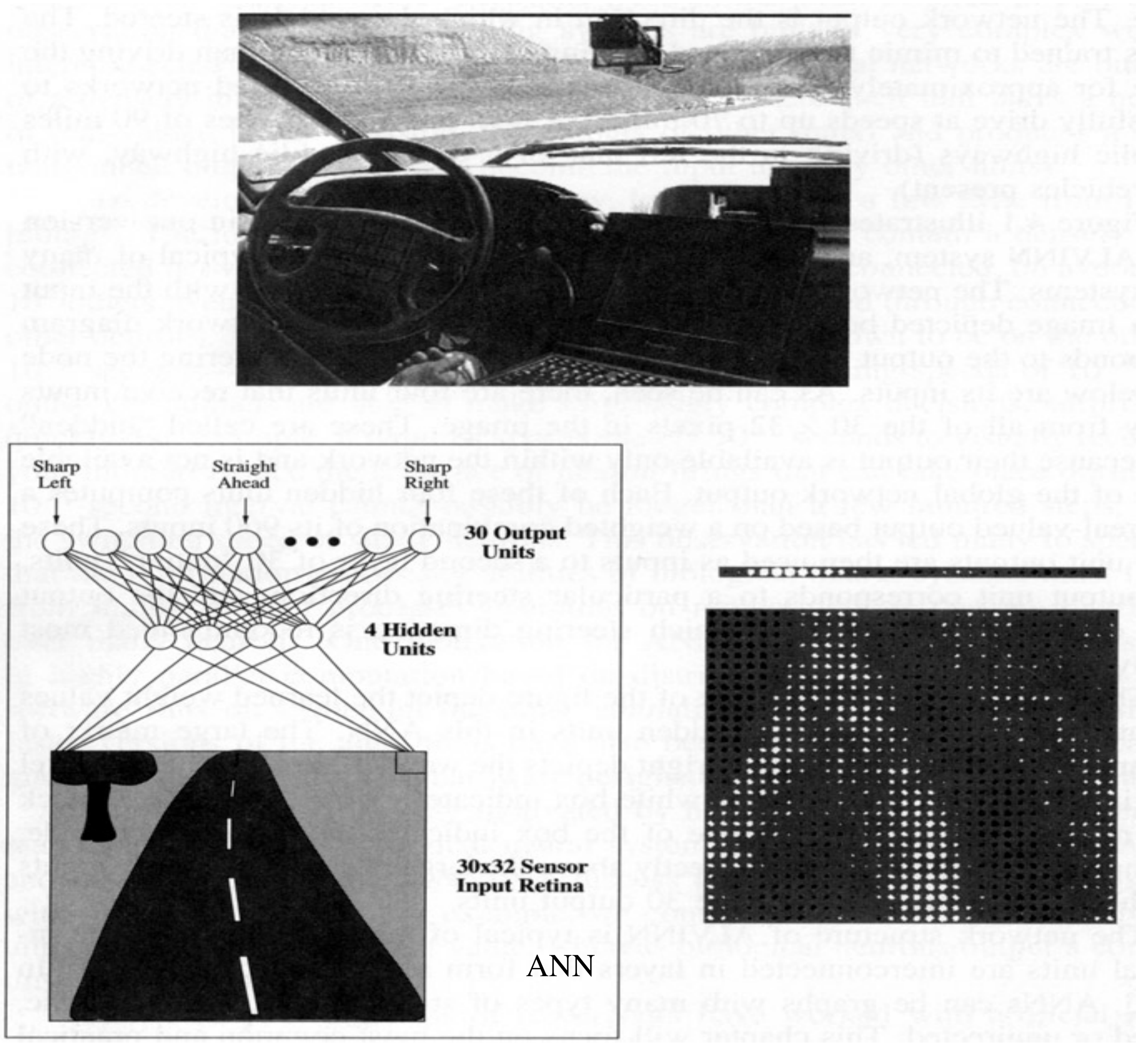
ALVINN - ANN to steer an autonomous vehicle driving at normal speeds on public highways

Input - 30 x 32 grid of pixel intensities from a forward-pointed camera

Output - direction vehicle is steered

Trained to mimic the observed steering commands of a human driving the vehicle for 5 minutes

ALVINN’s ANN


Sharp Left
Straight Ahead
Sharp Right
30 Output Units
4 Hidden Units
30x32 Sensor Input Retina

Backpropagation Network Representations

Individual units interconnected in layers that form a directed graph

Learning corresponds to choosing a weight value for each edge in the graph

Certain types of cycles are allowed

- Vast majority of practical applications are layered acyclic feed-forward networks like ALVINN

Appropriate Problems for ANN

Training data is noisy, complex sensor data

Also problems where symbolic algorithms are used - decision tree learning (DTL)

- ANN and DTL produce results of comparable accuracy

Specifically

Instances are attribute-value pairs

- attributes may be highly correlated or independent,
- values can be any real value

Target function may be

- discrete-valued,
- real-valued or
- vector-valued

Attribute Value Pairs

	P_1	P_2	P_3	P_4	P_5
O_1	1	2	0	1	1
O_2	1	2	0	1	1
O_3	2	0	0	1	0
O_4	0	0	1	2	1
O_5	2	1	0	2	1
O_6	0	0	1	2	2
O_7	2	0	0	1	0

Not really a pair

Ps are attributes

Numbers are values

More Specifically

Training examples may contain errors

Long training times are acceptable

Can Require fast evaluation of the learned target function

Humans do NOT need to understand the learned target function!!!

ANN Solution

[2.8 3.4 1.6
1.2 0.6 3.2
4.5 1.3 0.8]

Perceptrons

Inputs a vector of real-valued inputs

calculates a linear combination and

outputs a 1 if the result is greater than some threshold and -1 otherwise

Hyperplane decision surface in the n-dimensional space of instances

Not all datasets can be separated by a hyperplane, but if they can they are ***linearly separable*** datasets

A Perceptron

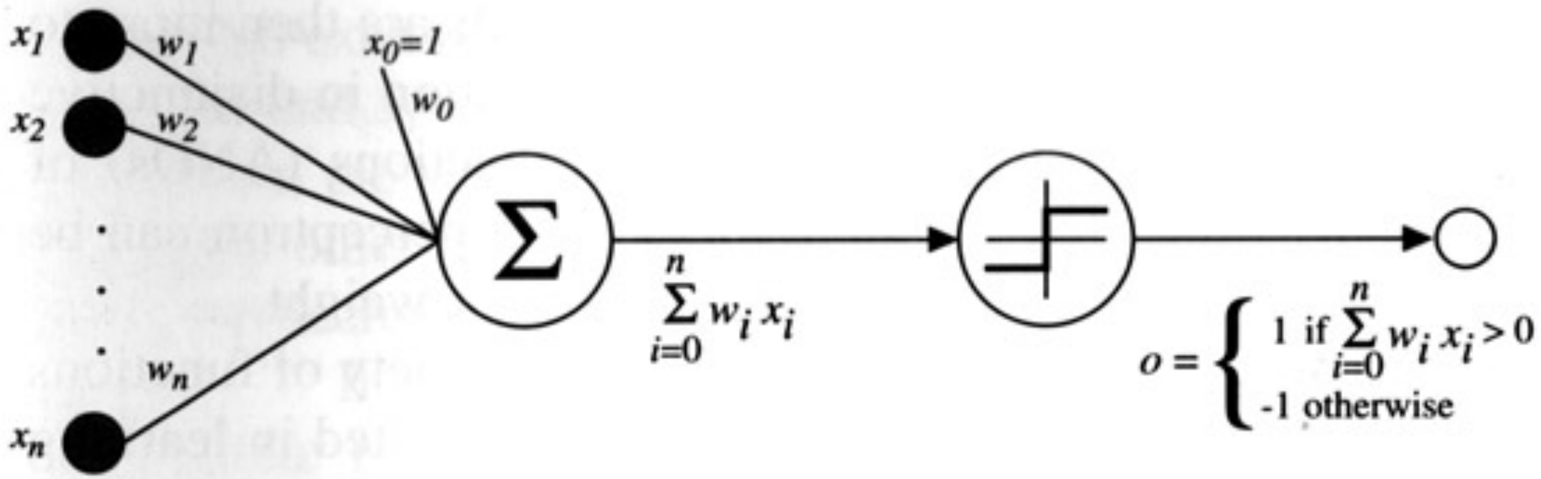

x_1
w_1
$x_0=1$
w_0
x_2
w_2
w_n
x_n
Σ
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

FIGURE 4.2
A perceptron.

Linearly Separable

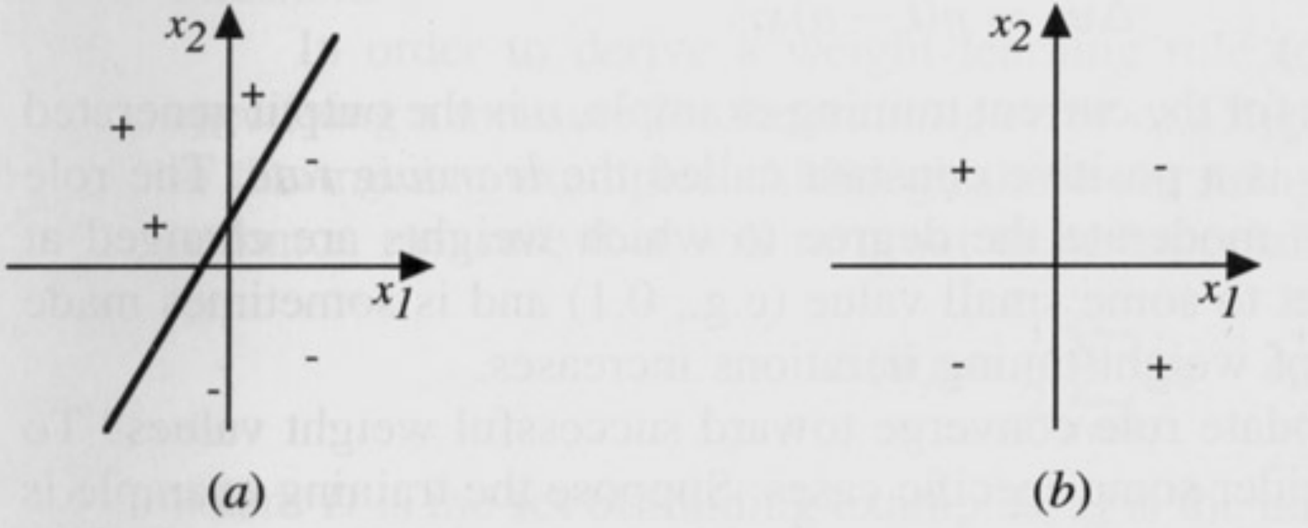


FIGURE 4.3

Even More Specifically

$$o(x_1,..,x_n) = 1 \qquad if\; w_0 + w_1x_1 + w_2x_2 + ... + w_nx_n > 0$$
$$= -1 \quad otherwise$$

Each w_i is a real-valued weight that determines the contribution of the input x_i to the perceptron output

The quantity $(-w_0)$ is the threshold

Threshold Explained

Remember algebra????

$$w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n > 0$$

$$w_1 x_1 + w_2 x_2 + \ldots + w_n x_n > 0 - w_0$$

$$w_1 x_1 + w_2 x_2 + \ldots + w_n x_n > -w_0$$

A Perceptron

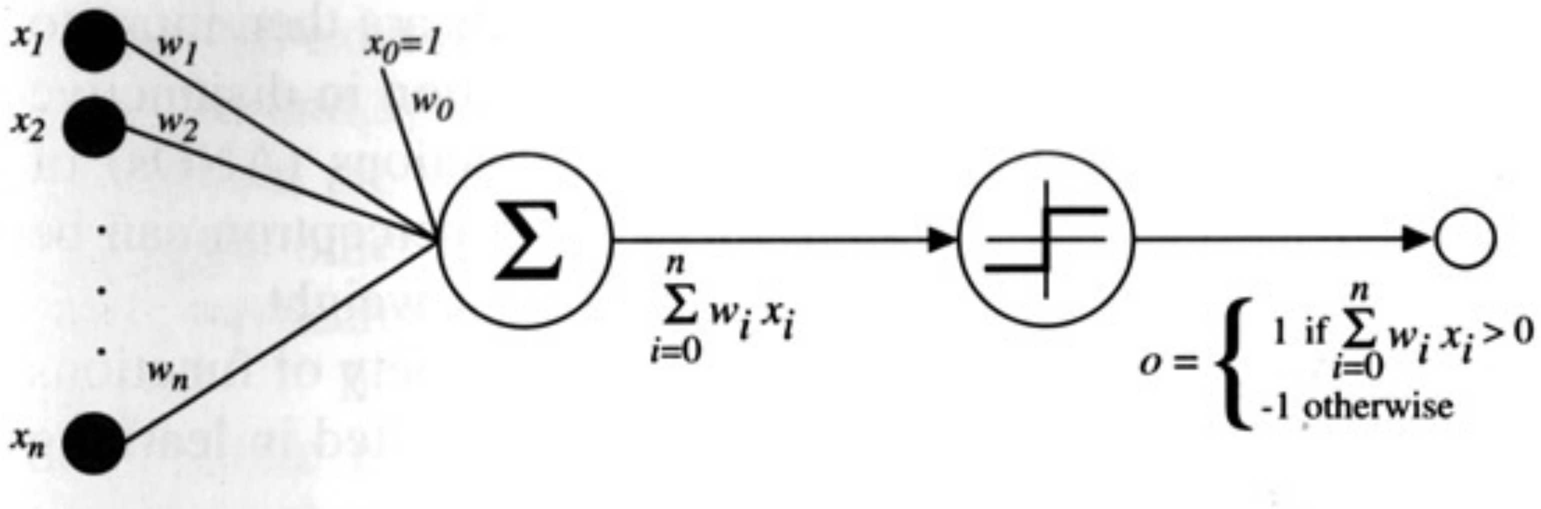

x_1
w_1
$x_0=1$
w_0
x_2
w_2
w_n
x_n
Σ
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

FIGURE 4.2
A perceptron.

Representational Power of Perceptrons

A single perceptron can represent many boolean functions - AND, OR, NAND, and NOR but not XOR!!

If 1 (true) and -1 (false), then to implement an AND function make $w_0 = -0.8$ and $w_1 = w_2 = 0.5$

Every boolean function can be represented by some network of perceptrons only **two** levels deep

Perceptron Learning Algorithms

Determine a weight vector that produces the correct ±1 output for each of the training examples

Several algorithms are known to solve this problem:

- The perceptron rule
- The delta rule

Guaranteed to converge to somewhat different acceptable hypothesis under somewhat different conditions

These are the basis for learning networks of many units - ANNs

The perceptron rule

Basis of Perceptron Training Rule

Begin with random weights

modify them

repeat until the perceptron classifies all training examples correctly

Perceptron Training Rule

Perceptron rule:

$$w_i \leftarrow w_i + \Delta w_i, where\ \Delta w_i = \eta\left(t - o\right)x_i$$

t is the target output,

o is the output generated by the perceptron

η is the ***learning rate***

- moderates the degree to which weights are changed at each step
- usually set to a small value (0.1)

Intuition for Perceptron Training Rule

If the training example is correctly classified (t-o) = 0, making $\Delta w_i = 0$, so no weights are updated

If the perceptron outputs -1 when the target output is +1 and assuming $\eta = 0.1$ and $x_i = 0.8$,
$\Delta w_i = 0.1(1-(-1))0.8 = 0.16$

If the perceptron outputs +1 when the target output is -1, then the weight would be decreased
$\Delta w_i = 0.1(-1-(1))0.8 = - 0.16$

Convergence of Perceptron Training Rule

This learning procedure will **converge**

within **finite number** of applications of the perceptron training rule

to a weight vector that **correctly classifies all** training examples,

provided

1. the training examples are linearly separable and
2. a sufficiently small learning rate, η, is used.

The delta rule

Gradient Descent Algorithm

If training examples are **not linearly separable**,

the delta rule converges toward best-fit approximation

Use ***gradient descent*** to find the weights that best fit the training examples

basis of the Backpropagation Algorithm

Error Definition

Assume an **unthresholded perceptron** (linear unit), then the training error is

$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2$$

Where

D is the set of training examples

$\boldsymbol{t_d}$ is the target output for the training example ***d*** and

$\boldsymbol{o_d}$ is the output of the linear unit for training example ***d***.

Perceptron vs Linear Unit

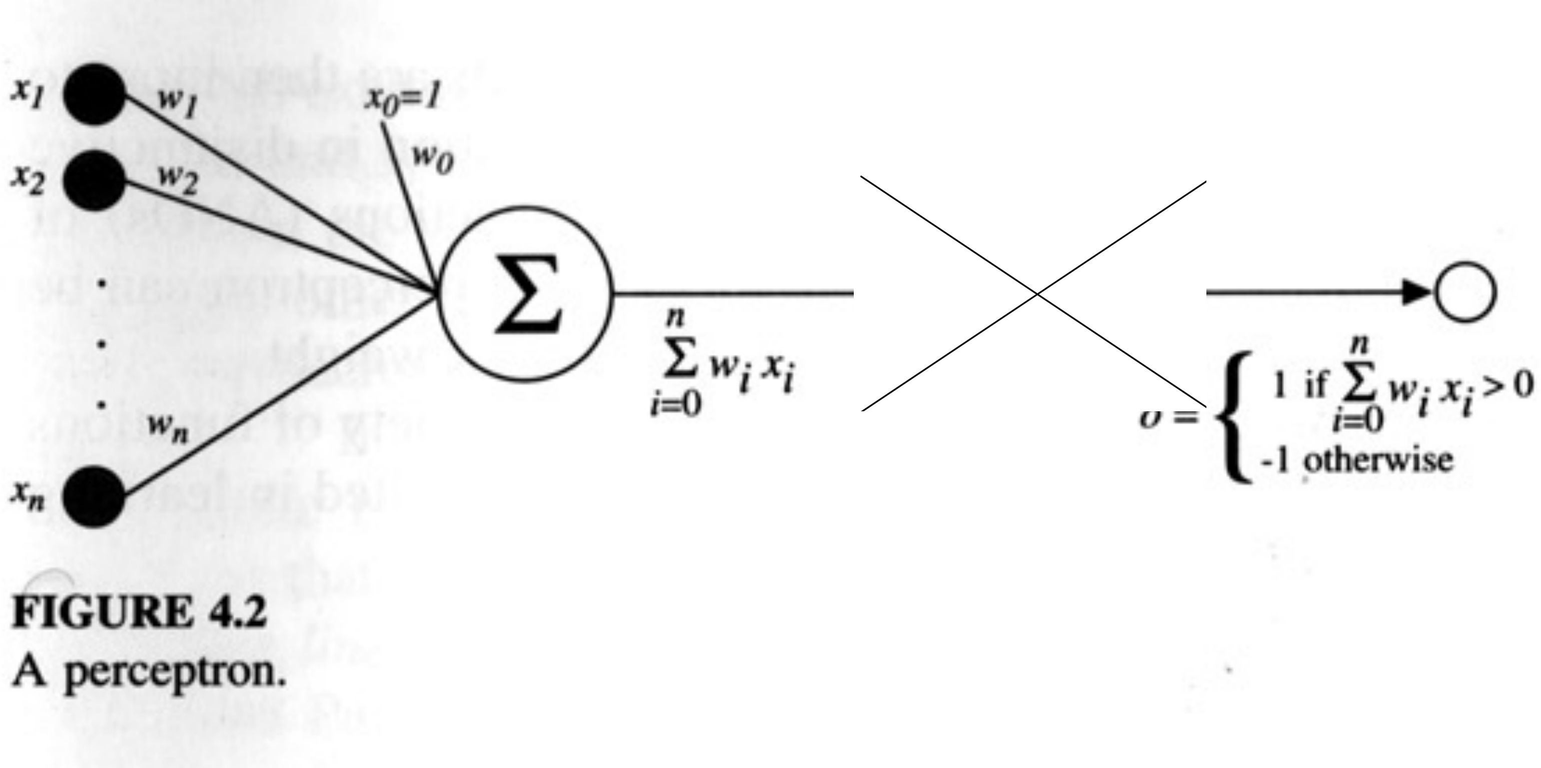

x_1
w_1
$x_0=1$
w_0
x_2
w_2
Σ
w_n
x_n
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

FIGURE 4.2
A perceptron.

Error Surface

Given the above error definition,

the error surface must be parabolic with a single global minimum.

Error Surface Image

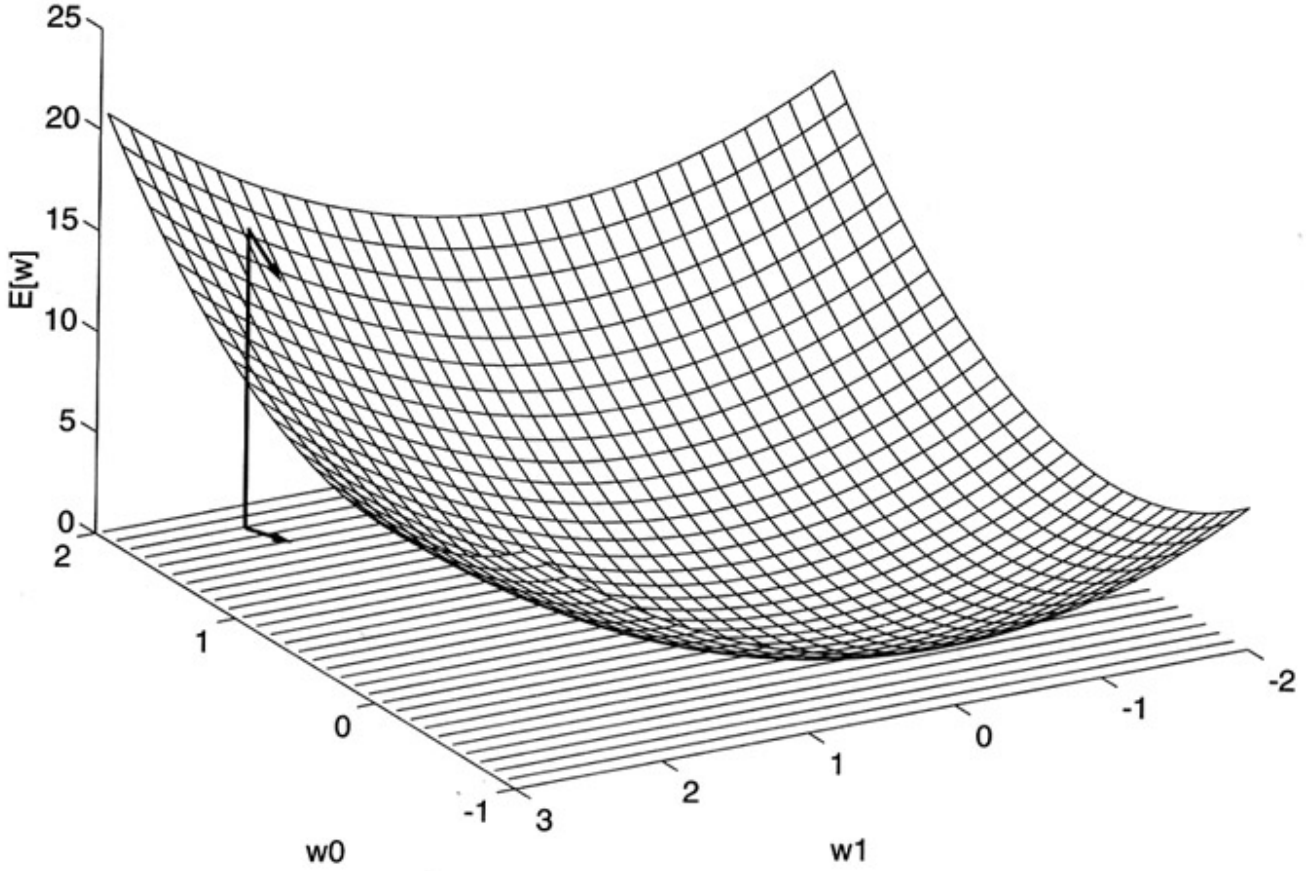
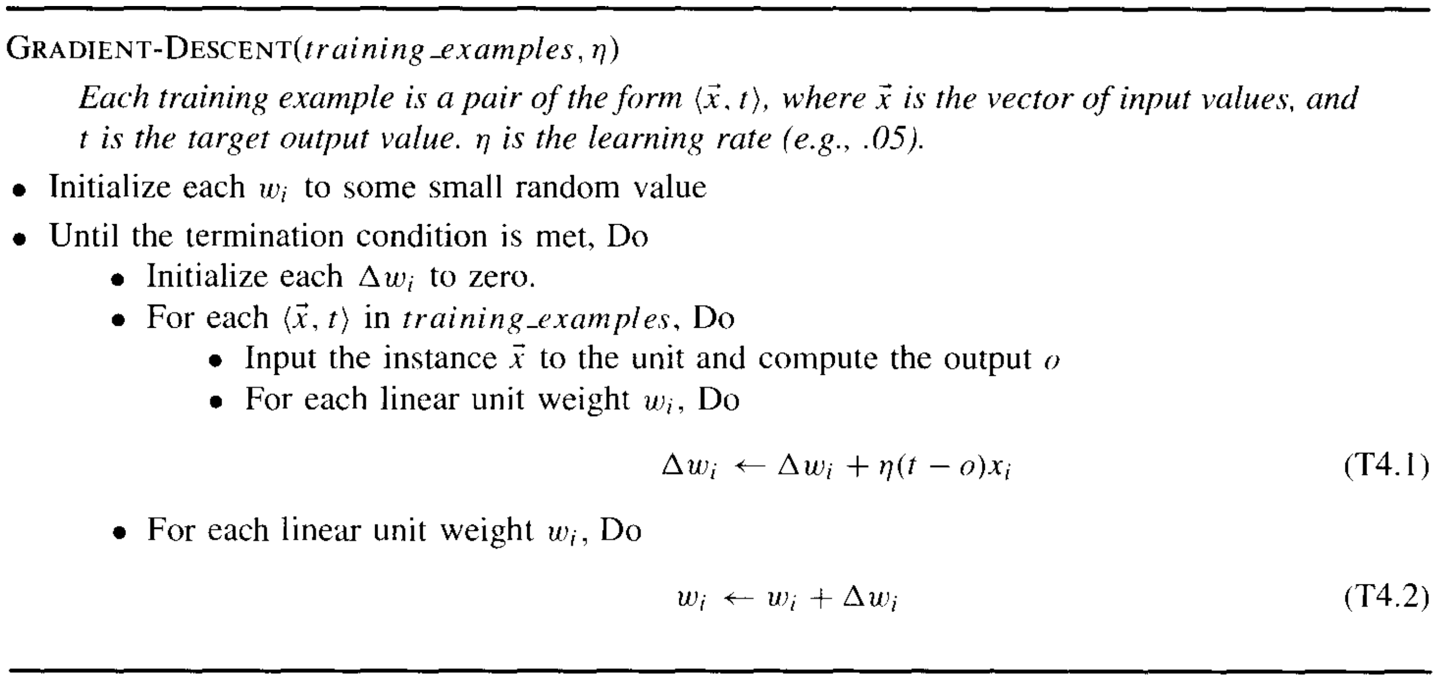


FIGURE 4.4

Gradient-Descent Algorithm



```
GRADIENT-DESCENT(training_examples, $\eta$)
    Each training example is a pair of the form $\langle \vec{x}, t \rangle$, where $\vec{x}$ is the vector of input values, and
    $t$ is the target output value. $\eta$ is the learning rate (e.g., .05).
• Initialize each $w_i$ to some small random value
• Until the termination condition is met, Do
        • Initialize each $\Delta w_i$ to zero.
        • For each $\langle \vec{x}, t \rangle$ in training_examples, Do
                • Input the instance $\vec{x}$ to the unit and compute the output $o$
                • For each linear unit weight $w_i$, Do
                        $\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$        (T4.1)
        • For each linear unit weight $w_i$, Do
                        $w_i \leftarrow w_i + \Delta w_i$        (T4.2)
```

Weight Update Rule Intuition

Gradient descent determines the weight vector that minimizes E.

It starts with an arbitrary weight vector,

modifies it in small steps **in the direction that produces the steepest descent**, and

continues **until the global minimum error** is reached,

Weight Update Rule

Weight update rule:

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d) x_{id}$$

Where

$\boldsymbol{x_{id}}$ denotes input component $\boldsymbol{x_i}$ for training example $\boldsymbol{d}$

Convergence

Because the error surface contains only a **single global minimum**,

the algorithm will **converge** to a weight vector with **minimum error**,

regardless of whether the training examples are **linearly separable**,

given a **sufficiently small learning rate** η is used.

Hence a common modification is to **gradually reduce** the value of η as the number of steps grows.

Gradient descent
Important General Paradigm

When Continuously parameterized hypothesis

i.e., The error can be **differentiated** with respect to the hypothesis parameters

Problems with Gradient descent

1. **Converging** to a local minimum can be quite **slow**

2. If there are **multiple local minima**, then there is no guarantee that the procedure will find the global minimum
   - The error surface will not be parabolic with a single global minima, when training multiple nodes.

Stochastic Gradient Descent

Approximate gradient descent search by updating weights incrementally, following the calculation of the error for ***each*** individual example

Delta rule: $\Delta w_i = \eta(t - o)x_i$

(same as LMS algorithm in 367, but only similar to perceptron training rule because using linear unit)

SGD Error Function

$$E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2$$

If η is sufficiently small, stochastic gradient descent (SGD) can be made to approximate true gradient descent (GD) arbitrarily closely

Difference between GD and SGD

In **GD** the error is **summed over all examples** before updating weights

In **SGD** weights are **updated upon examining each training example**

Summing over multiple examples in **GD** requires **more computation per weight update step**. But since it uses the True gradient, it is often used with a **larger step size** (larger η).

If there are **multiple local minima** with respect to the error function, **SGD** can sometimes **avoid falling** into these local minima.

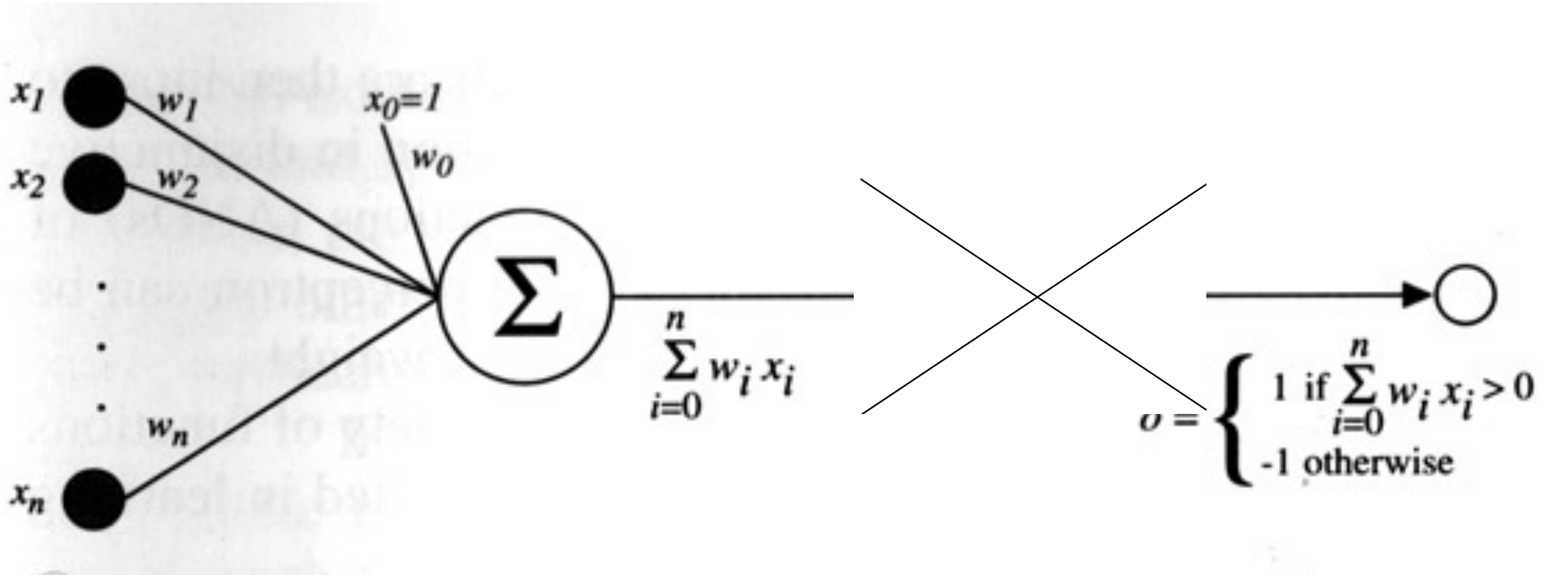
Difference between Delta Rule & Perceptron Training Rule

Appear identical, but

PTR is for **thresholded perceptron** and

DR is for a **linear unit** (or **unthresholded perceptron**).

Perceptron vs Linear Unit



x_1
w_1
$x_0=1$
w_0
x_2
w_2
Σ
w_n
x_n
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

FIGURE 4.2
A perceptron.

Delta Rule with Thresholded perceptron

If the **unthresholded perceptron** can be trained to **fit these values perfectly** then **so can the thresholded perceptron.**

DR can be used to **train a thresholded perceptron**, by **using ±1 as target values** to a linear unit and having the **thresholded unit return the sign** of the linear unit.

If the target values cannot be perfectly fit, then the **thresholded perceptron** will be **correct** whenever the linear unit has the **right sign**, but this is not guaranteed to happen

Multilayer Networks & Nonlinear Surfaces

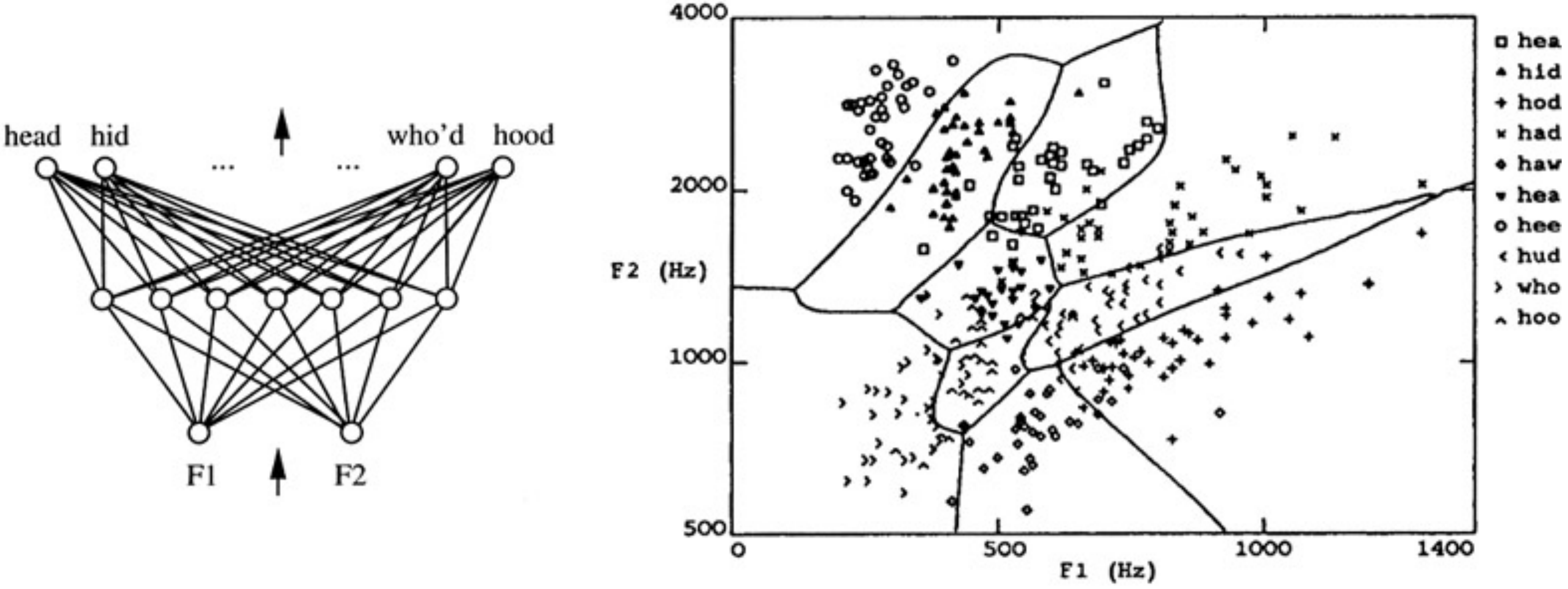


FIGURE 4.5

Multilayer Networks

Multiple layers of linear units still produce only linear functions!!!

Perceptrons have a discontinuous threshold which is undifferentiable and therefore unsuitable for gradient descent

We want a unit whose output is a **nonlinear differentiable function of the inputs**

One solution is a **sigmoid unit**

What is a Sigmoid Unit?

Like perceptrons

it computes a **linear combination** of its inputs and then

applies a threshold to the result.

But the **threshold output** is a **continuous function** of its input which **ranges from 0 to 1**.

If is often referred to as a **squashing function**.

Sigmoid Threshold Unit

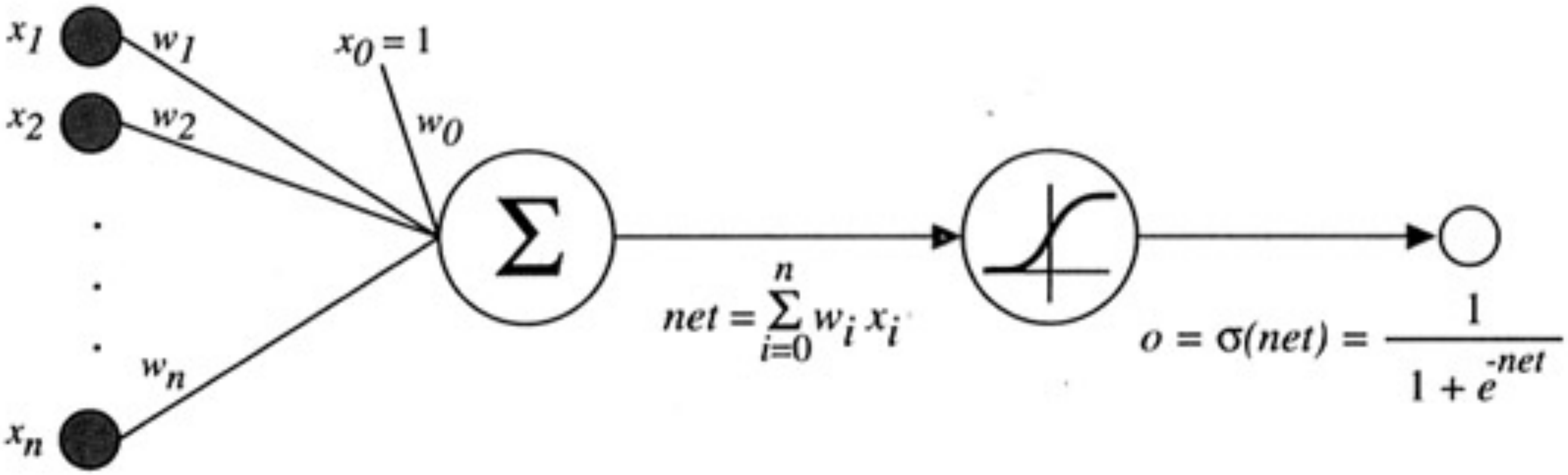

x_1
x_2
x_n
w_1
w_2
w_n
$x_0 = 1$
w_0
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1 + e^{-net}}$

FIGURE 4.6

Properties of the Backpropagation Algorithm

Learns **weights** for a multilayer network, given a **fixed set of units and interconnections**

It uses **gradient descent** to **minimize the squashed error** between the network outputs and the target values for these outputs

Error surface

Error formula:

$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d \in D}\sum_{k \in outputs}(t_{kd} - o_{kd})^2$$

Outputs is the set of output units in the network,

$\boldsymbol{t_{kd}}$ and $\boldsymbol{o_{kd}}$ are the target value and output value associated with the

$\boldsymbol{k}^{\mathrm{th}}$ output unit and the training example ***d***

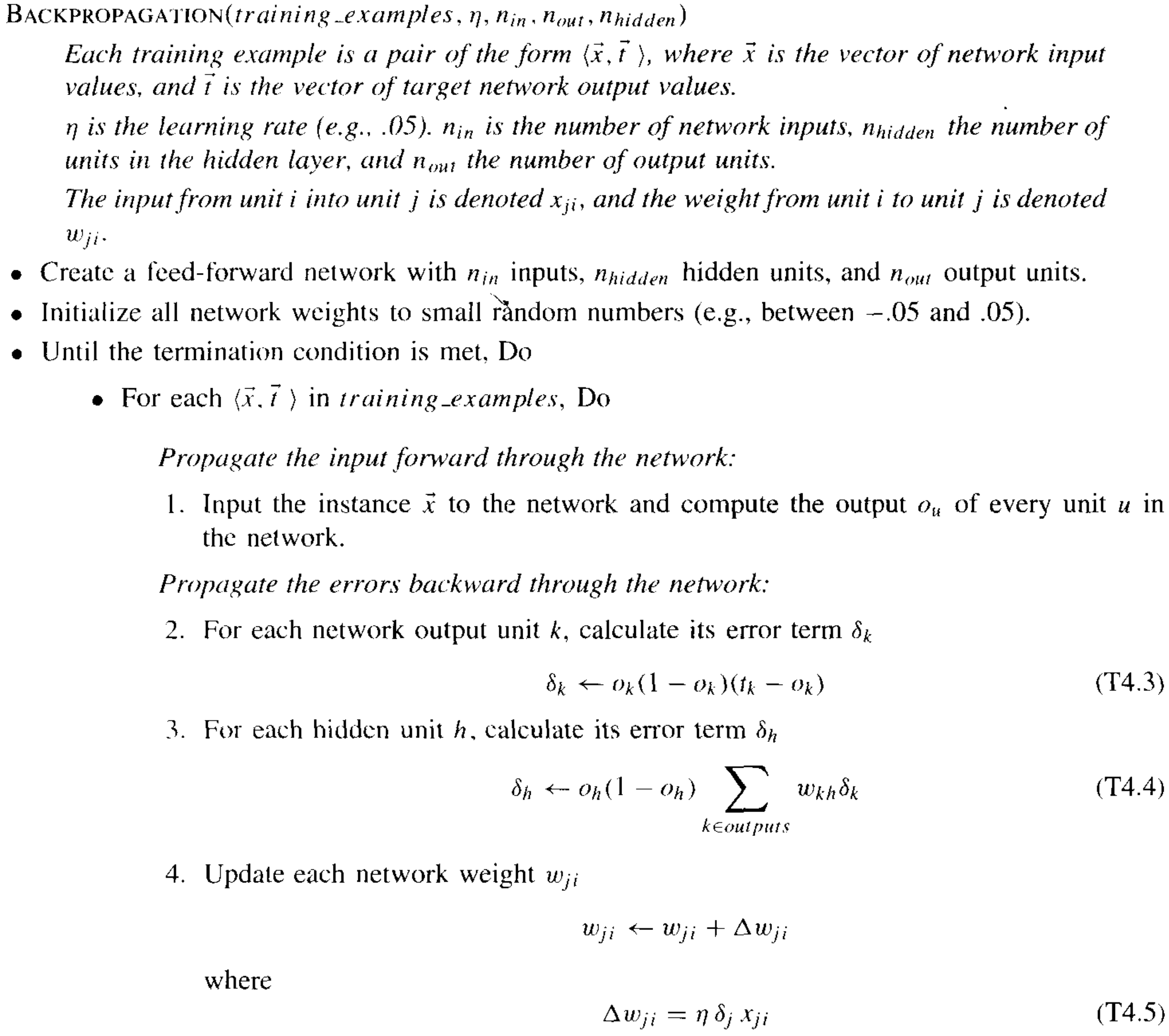
Error Surface

In **multilayer networks** the error surface can have **multiple minima**,

In practice Backpropagation has produced excellent results in many real-world applications.

The algorithm (on the next page) is for two layers of sigmoid units and does stochastic gradient descent.

Backpropagation Algorithm



```
BACKPROPAGATION(training_examples, $\eta$, $n_{in}$, $n_{out}$, $n_{hidden}$)
    Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$, where $\vec{x}$ is the vector of network input
    values, and $\vec{t}$ is the vector of target network output values.
    $\eta$ is the learning rate (e.g., .05). $n_{in}$ is the number of network inputs, $n_{hidden}$ the number of
    units in the hidden layer, and $n_{out}$ the number of output units.
    The input from unit i into unit j is denoted $x_{ji}$, and the weight from unit i to unit j is denoted
    $w_{ji}$.
• Create a feed-forward network with $n_{in}$ inputs, $n_{hidden}$ hidden units, and $n_{out}$ output units.
• Initialize all network weights to small random numbers (e.g., between $-.05$ and $.05$).
• Until the termination condition is met, Do
    • For each $\langle \vec{x}, \vec{t} \rangle$ in training_examples, Do
        Propagate the input forward through the network:
        1. Input the instance $\vec{x}$ to the network and compute the output $o_u$ of every unit $u$ in
           the network.
        Propagate the errors backward through the network:
        2. For each network output unit $k$, calculate its error term $\delta_k$
               $\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$                                (T4.3)
        3. For each hidden unit $h$, calculate its error term $\delta_h$
               $\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh} \delta_k$      (T4.4)
        4. Update each network weight $w_{ji}$
               $w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$
           where
               $\Delta w_{ji} = \eta \, \delta_j \, x_{ji}$                                 (T4.5)
```

Backpropogation Weight Training Rule Output Units

$\boldsymbol{\delta_k \leftarrow o_k(1-o_k)(t_k-o_k)}$

The error $\boldsymbol{(t-o)}$ in the delta rule is replaced by $\boldsymbol{\delta_{j.}}$

The output unit $\boldsymbol{k}$ is the familiar $\boldsymbol{(t_k -o_k)}$ from the delta rule multiplied by $\boldsymbol{o_k(1-o_k)}$

$\boldsymbol{o_k(1-o_k)}$ is the derivative of the sigmoid squashing function.

Backpropogation Weight Training Rule Hidden Units

$$\delta_h \leftarrow o_h(1-o_h) \sum_{k \in outputs} w_{kh}\delta_k$$

For hidden unit ***h,*** the derivative component is the same but there is no target value directly available

so you sum the error terms $\boldsymbol{\delta_k}$ for each output unit influenced by ***h***

weighing each of the $\boldsymbol{\delta_k}$ by the weight, $\boldsymbol{w_{kh}}$, from the hidden unit ***h*** to the output unit ***k***.

This weight characterizes the degree to which each hidden unit ***h*** is responsible for the error in output unit ***k***.

Multilayer Networks & Nonlinear Surfaces

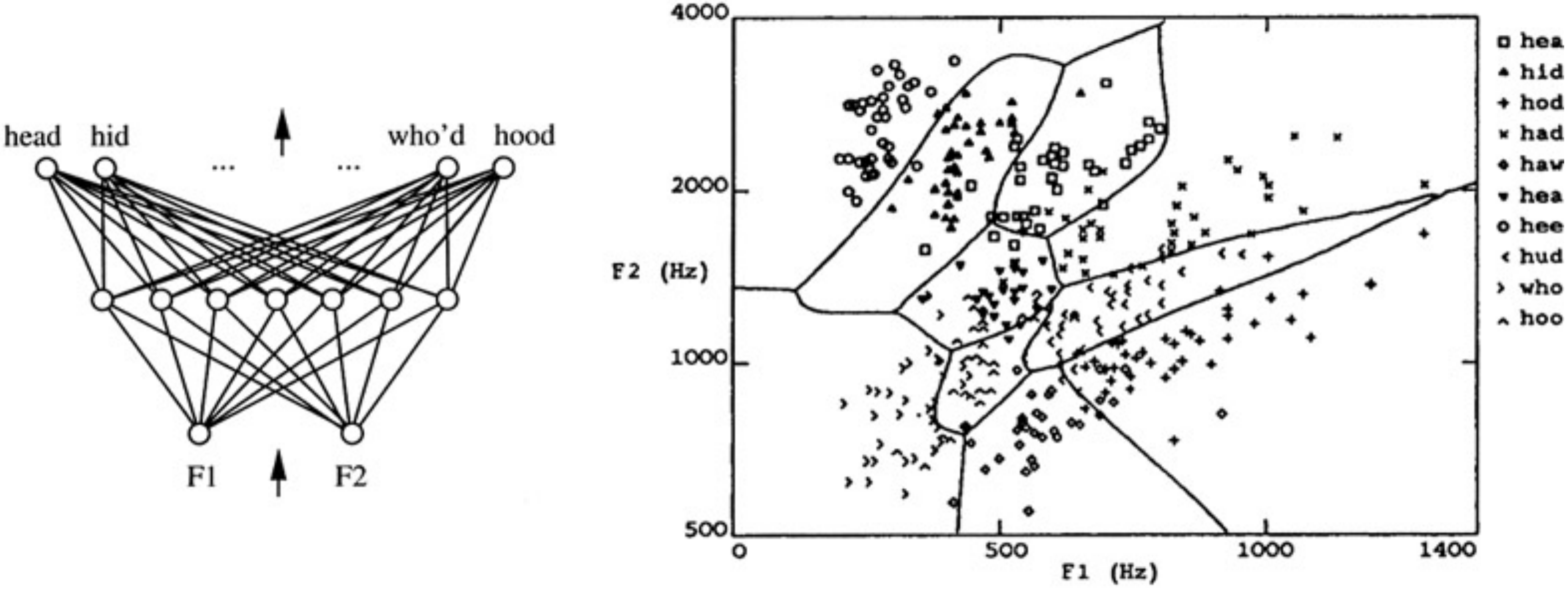


FIGURE 4.5

Termination Conditions for Backpropagation

Halt after a fixed number of iterations.

Halt once the error on the training examples falls below some threshold.

Halt once the error on a separate validation set of examples meets some criterion.

Important:

Too few iterations - fail to reduce error sufficiently

Too many iterations - overfit the data - 367

Momentum

Making the weight in the n^{th} iteration depend partially on the update during the $(n\text{-}1)^{th}$ iteration

$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1)$$

The momentum is represented by ***$0 \leq \alpha < 1$***

Intuition behind Momentum

The **gradient search trajectory** is analogous to a **momentumless ball** rolling down the error surface, the effect of α is to keep the ball rolling in the same direction from one iteration to the next.

The ball can **roll through small local minima** or along flat regions in the surface where the ball would stop without momentum.

It also causes a **gradual increase in the step size** in regions where the gradient is unchanging, thereby **speeding convergence**.

Arbitrary Acyclic Networks

Only equation (T4.4) has to change

Feedforward networks of arbitrary depth

$$\delta_r = o_r(1 - o_r) \sum_{s \in layer\, m+1} w_{sr}\delta_s$$

Directed acyclic graph, not arranged in uniform layers

$$\delta_r = o_r(1 - o_r) \sum_{s \in Downstream(r)} w_{sr}\delta_s$$

Convergence and Local Minima

The error surface in multilayer neural networks may contain many different local minima where gradient descent can become trapped.

But backpropagation is a highly effective function approximation in practice.

Why??

Why it works

Networks with large numbers of weights correspond to error surfaces in very high dimensional spaces.

When gradient descent falls into a local minima with respect to one weight it won't necessarily be with respect to the other weights.

The more weights, the more dimensions that might provide an escape route

- do I believe this???
- more nodes, more outputs, more inputs

Heuristics to Overcome Local Minima

Add momentum

Use stochastic gradient search

New seed (e.g., initial random weights), and choose the one with the best performance on the validation set or treat as a committee or ensemble

Is this cheating??

Add more dimensions (within reason)

Representational Power of Feedforward Networks

Boolean functions: 2 layers of units, but number of hidden nodes grows exponentially in the number of inputs.

Continuous Functions: every bounded continuous function to **arbitrary accuracy** in 2 layers of units.

Arbitrary functions: **arbitrary accuracy** by a network with 3 layers of units - based on linear combination of many localized functions.

Caveats

Arbitrary error???

“Warning Will Robinson”

Network weight variables reachable from the initial weight values may not include all possible weight vectors!!!!!

Hypothesis Space

Every possible assignment of network weights represents a syntactically different hypothesis.

N-dimensional Euclidean space of the n network weights.

This hypothesis space is continuous.

Since E is differentiable with respect to the continuous parameters, we have a well-defined error gradient

Inductive Bias

Inductive Bias depends on interplay between gradient descent search and the way the weight space spans the space of representable functions.

Roughly - smooth interpolation between data points

Given two positive training instances with no negatives between them, Backpropagation will tend to label the points between as positive.

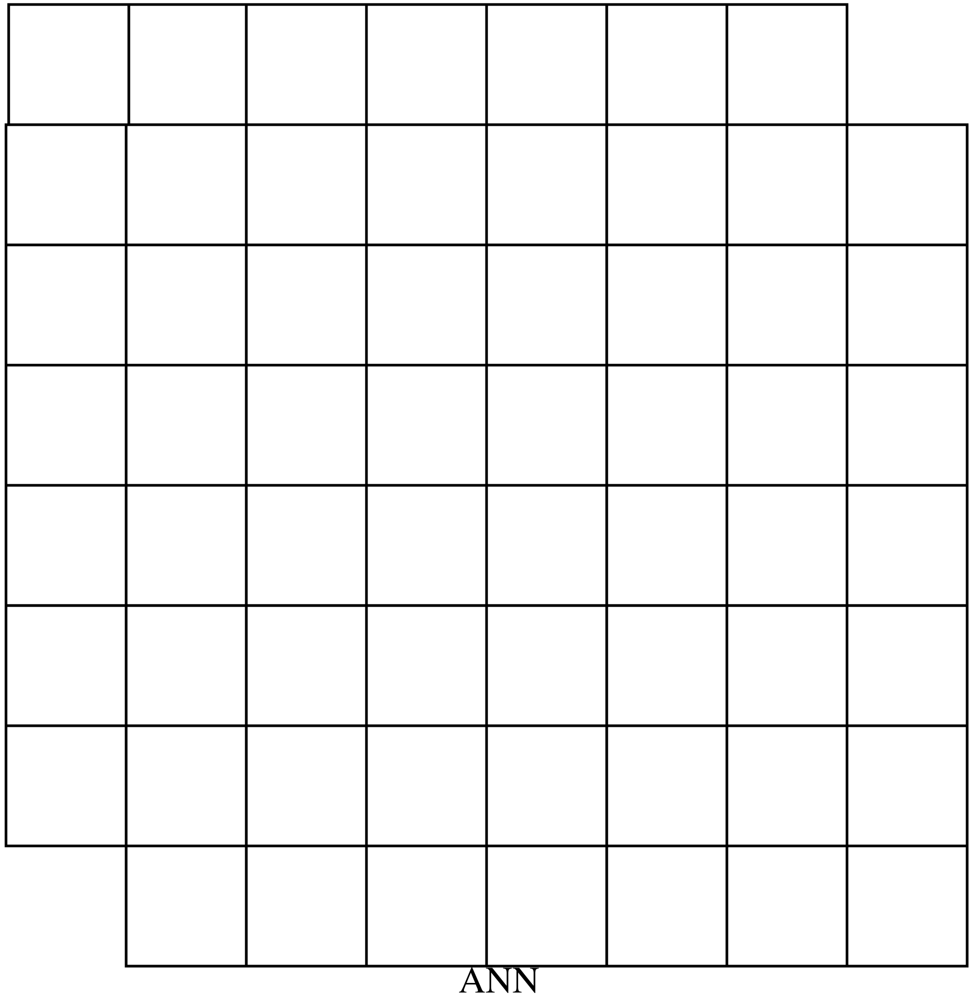
Hidden Layer Representations

Backpropagation can discover useful intermediate representations at the hidden unit layers.

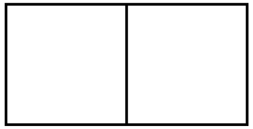
It is a way to make implicit concepts explicit.

Discovering binary encoding

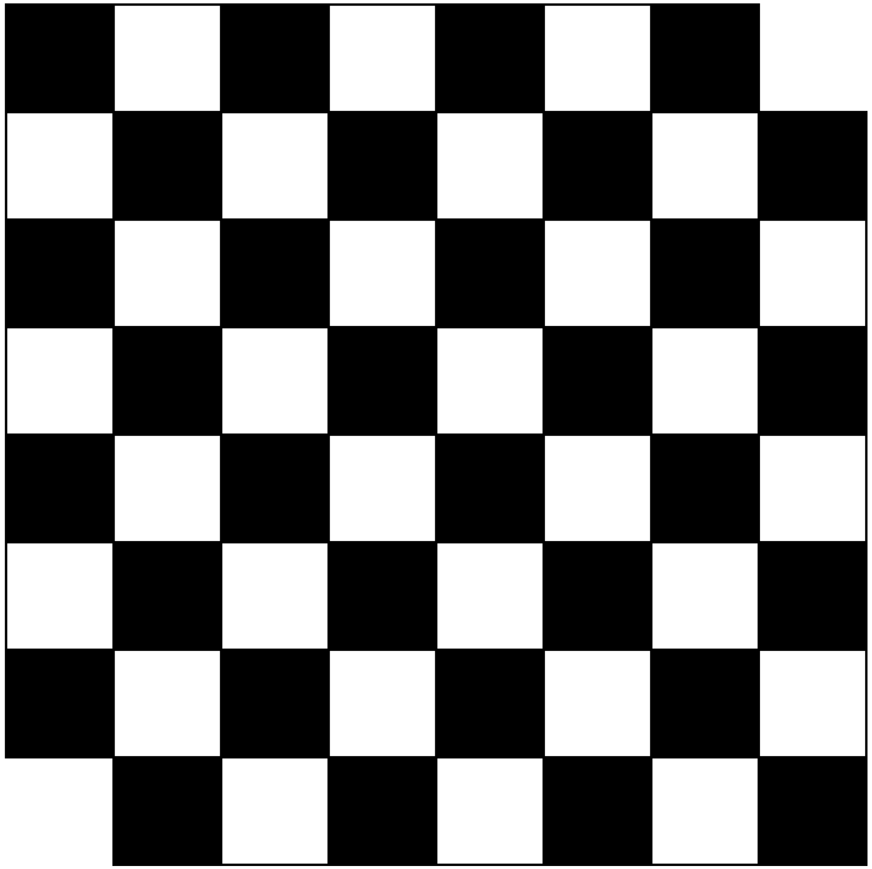
Mutilated Chessboard problem



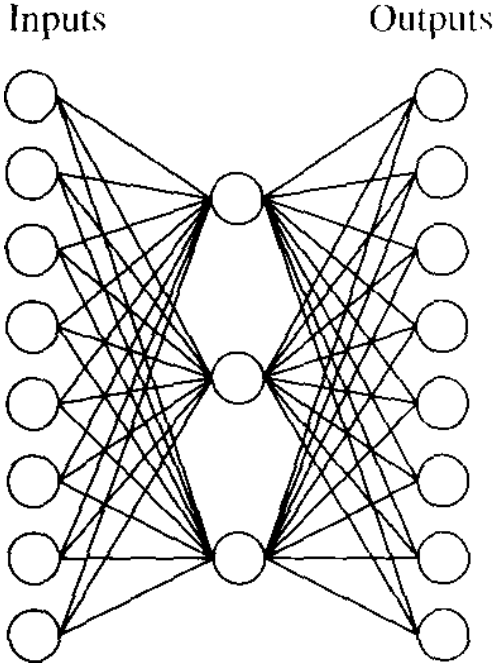
Domino

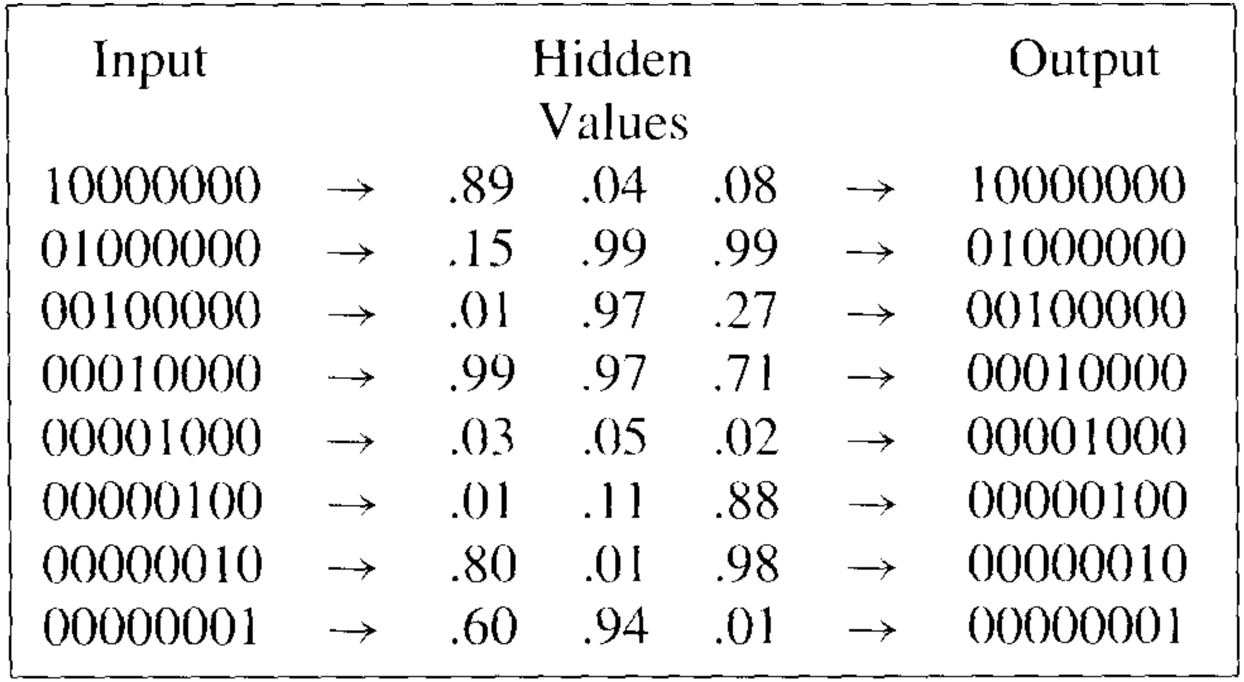


Explicit Representation



Backprop in action





Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

Value of Hidden Units

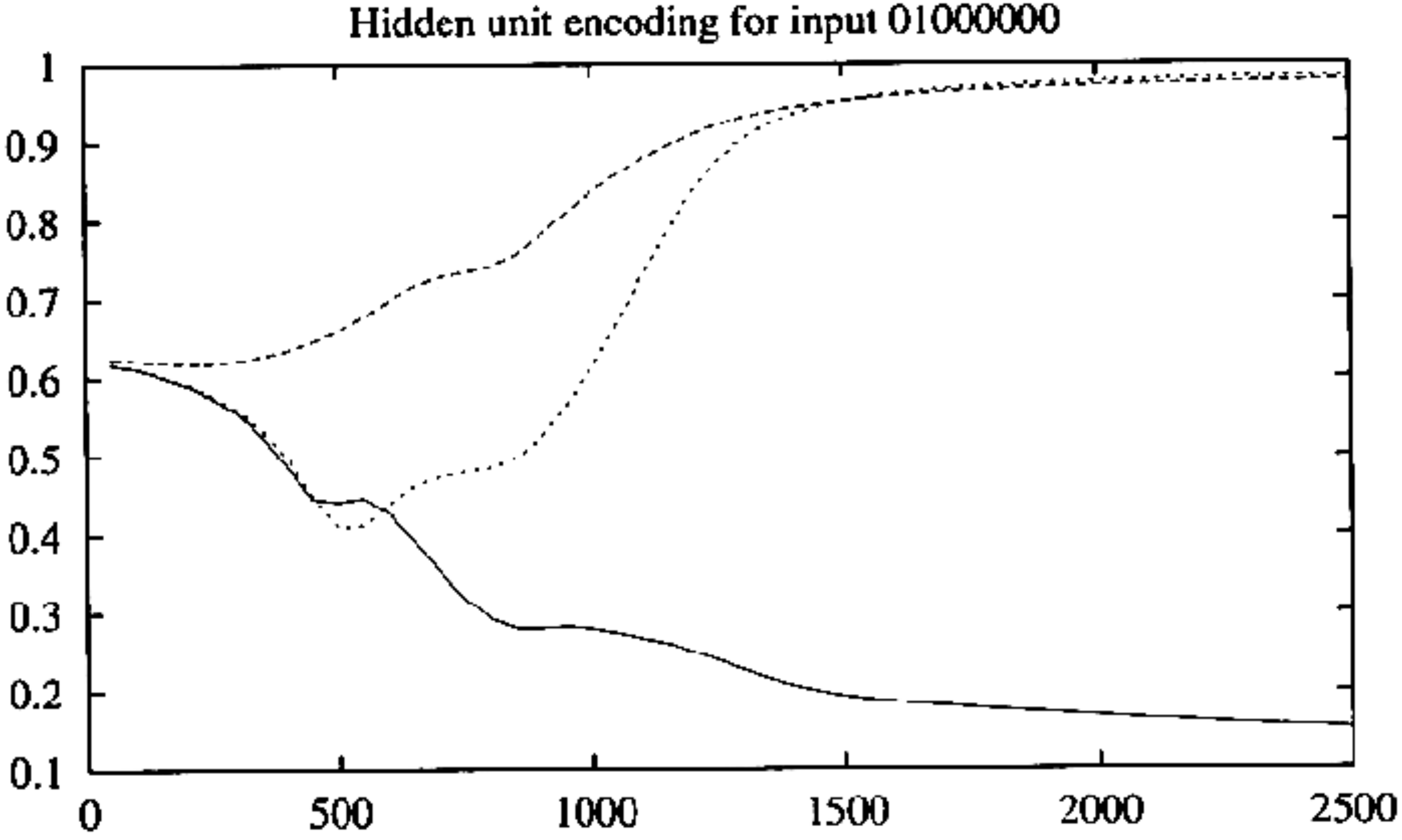
This does not work across rows!

Does not work across multiple entities or copies of the same entity (time stamps).

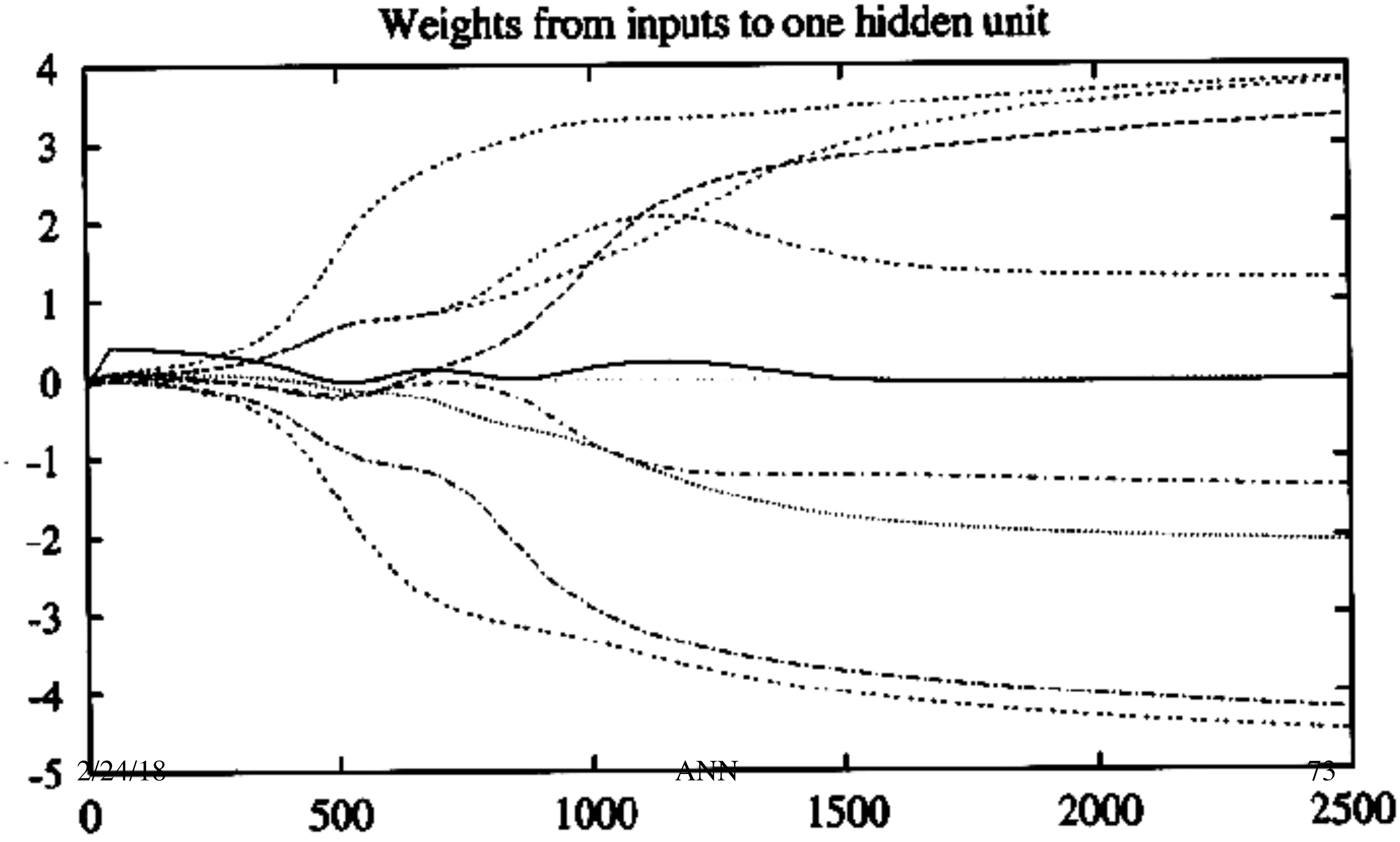
Important degree of flexibility

More layers of units - more complex features can be invented

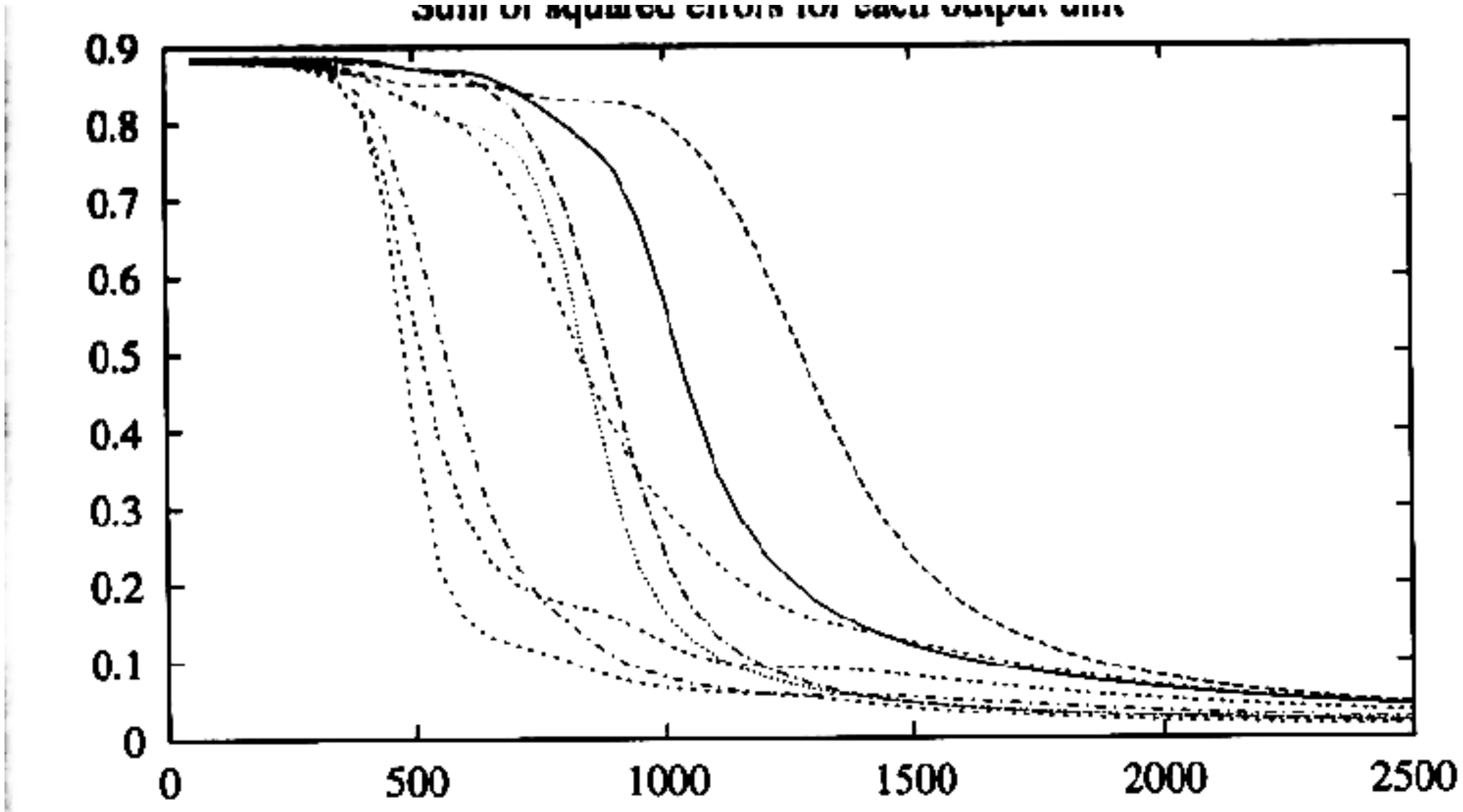
Hidden Unit Encoding



Weights from inputs



Sum of squares Error



Overfitting and Stopping Criteria

1. Train until the Error on the training examples falls below some level

Why does overfitting tend to occur in later iterations?

Initially weights are set to small random numbers, as training proceeds weights change to reduce error over the training data & complexity of the decision surface increases, given enough iterations can overfit.

Weight decay - decreases each weight by a small factor on each iteration - intuition keep weight values small to bias against complex decision surfaces

do complex decision surfaces need to have high weights???

Continued

2. Stop when you reach the lowest error on the validation set.

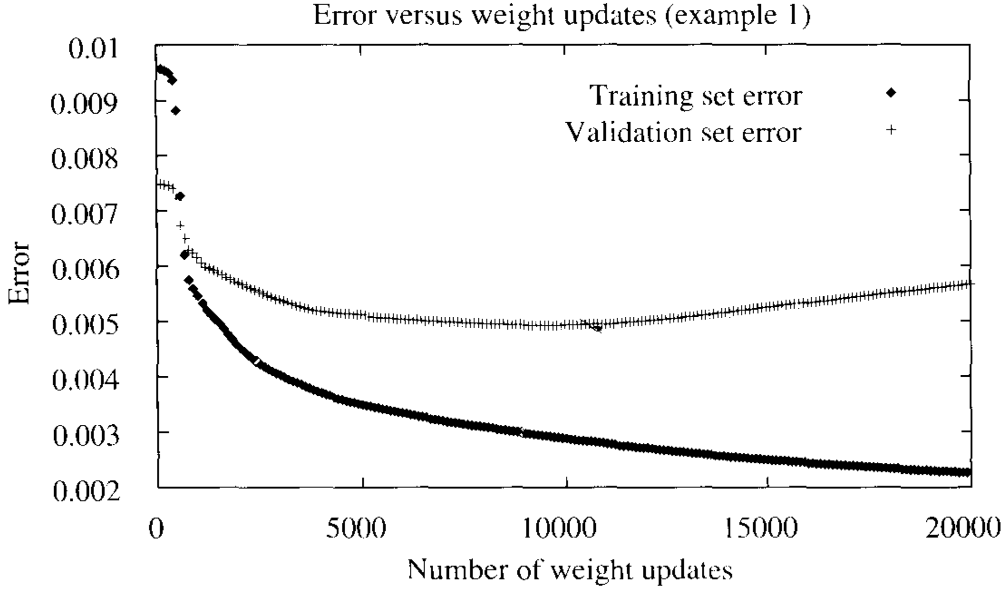
Keep current ANN weights and the best-performing weights thus far measured by error over the validation set

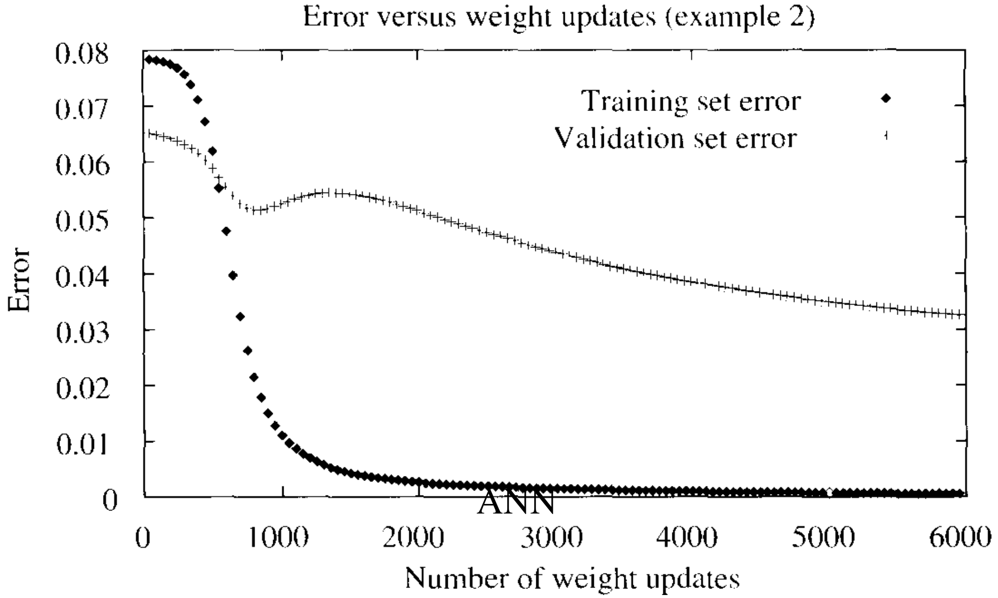
Training is terminated once the current weights reach a significantly higher error over the validation set

Care must be taken to avoid stopping too soon!!

If data is too small can do k-fold cross validation (remember to use just to determine the number of iterations!) then train over whole dataset (same in decision trees)

Error Plots


Error versus weight updates (example 1)


Error versus weight updates (example 2)

Face Recognition Task

20 people
32 images per person

Varying expression, direction looking, wearing sunglasses, background, clothing worn, position of face in image.

624 greyscale images, resolution 120x128, each pixel intensity ranges from 0 to 255.

Could be many targets: identity, direction, gender, whether wearing sunglasses - All can be learned to high accuracy.

We consider direction looking.

ANN Architecture

- Number of layers
- Number of hidden units per level
- Input and Output encoding

Input Encoding

How encode the image?

Use 102x128 inputs?

Extract local features: edges, regions of uniform intensity - how handle a varying number of features per image?

Actual Input Encoding Used

Encode image in 30x32 pixel intensity values,

one network input per pixel

pixel intensities were linearly scaled from 0 to 255 down to 0 to 1.

Why?

Coarse Resolution Summary

The 30x32 grid is a coarse resolution summary.

Coarse pixel intensity is a mean of the corresponding high pixel intensities

Reduces computational demands while maintaining sufficient resolution to correctly classify images

Same as ALVINN except there a represented pixel was chosen randomly for efficiency

Output Encoding

A single output unit,

assigning 0.2, 0.4, 0.6, 0.8 as being left, right, up and straight

OR

4 output nodes and choose the highest-valued output as the prediction (1-of-n output encoding)

OR

4 separate neural networks with 1 output each

Output Encoding Issues

Is “left” really closer to “right” then it is to “up”?

4 outputs gives more degrees of freedom to the network for representing the target function (4x #hidden units instead of 1x)

The difference between the highest valued output and the second highest valued output gives a measure of the confidence in the network prediction

Target Values

What should the target values be?

Could use <1,0,0,0> but we use <0.9,0.1,0.1,0.1>

sigmoid units can't produce 0 and 1 exactly with finite weights so gradient descent will force the weights to grow without bound

Network Graph Structure

How many units and how to interconnect?

Most common - layer units with feedforward connections from every unit in one layer to every unit in the next.

The more layers the longer the training time.

We choose 1 hidden layer and 1 output layer.

Hidden Units

How many hidden units?

3 units = 90% accuracy, 5 minutes learning time

30 units = 92% accuracy, 1 hr learning time

In general there is a minimum number of hidden units needed

above that the extra hidden units do not dramatically effect the accuracy,

provided cross-validation is used to determine how many gradient descent iterations should be performed, otherwise increasing the number of hidden units often increases the tendency to overfit the training data.

Other Algorithm Parameters

Learning rate, η, 0.3

Momentum, α, 0.3

Lower values produced equivalent generalization but longer training times, if set too high training fails to converge to a network with acceptable error.

Full gradient descent was used (not the stochastic approximation).

More Parameters

Network weights in the output units were initialized to small random values, but the input unit weights were initialized to zero.

It yields a more intelligible visualization of the learned weights without noticeable impact on generalization accuracy.

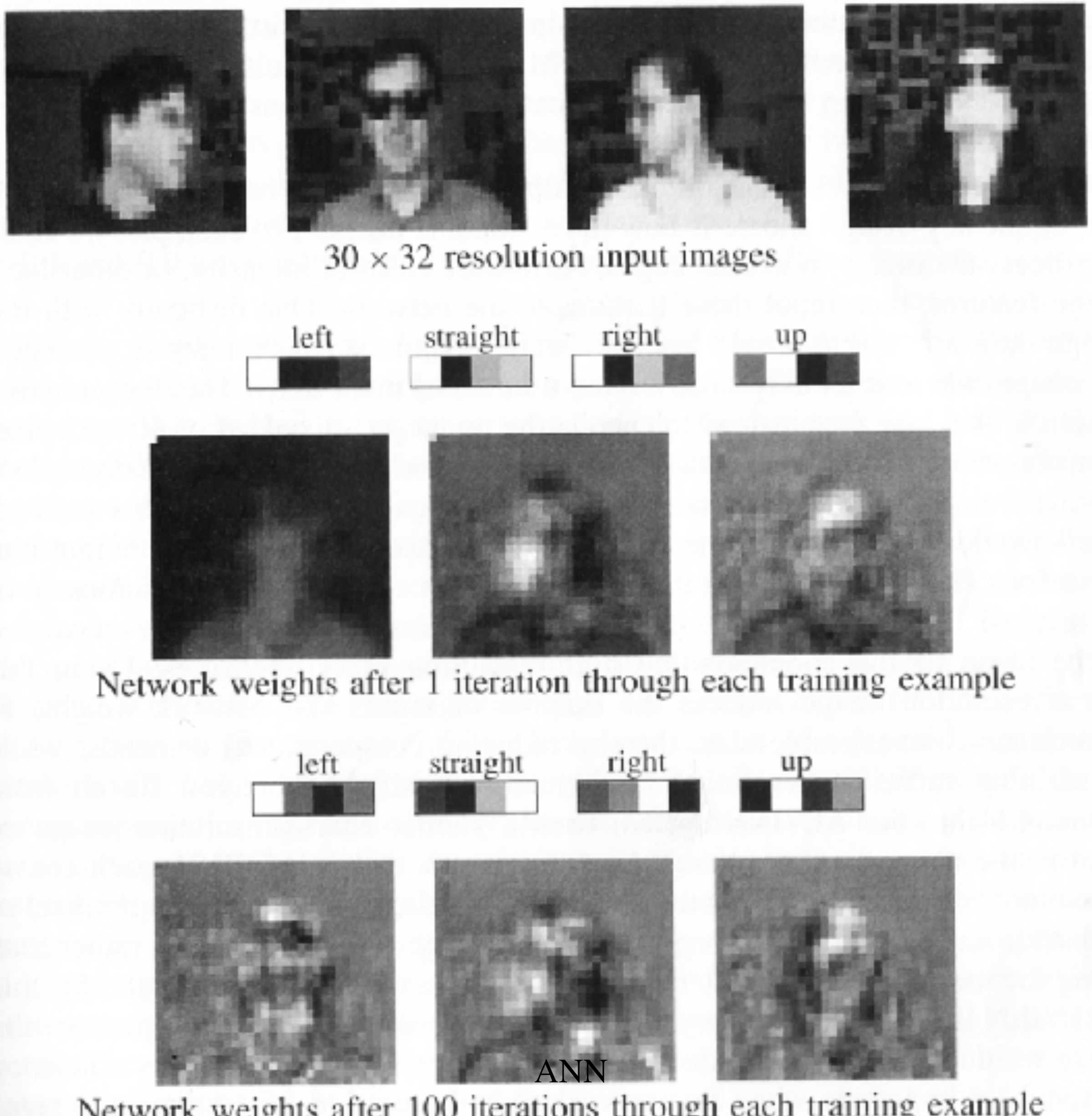
Still More Parameters

The number of training iterations was selected by partitioning the available data into a training set and a separate validation set.

GD was used to minimize the error over the training set and after every 50 gradient descent steps the network performance was evaluated over the validation set.

The final reported accuracy was measured over yet a third set of test examples that were not use to influence the training.

Learned Hidden Representations



30 × 32 resolution input images

Network weights after 1 iteration through each training example

Network weights after 100 iterations through each training example

Advanced Topics

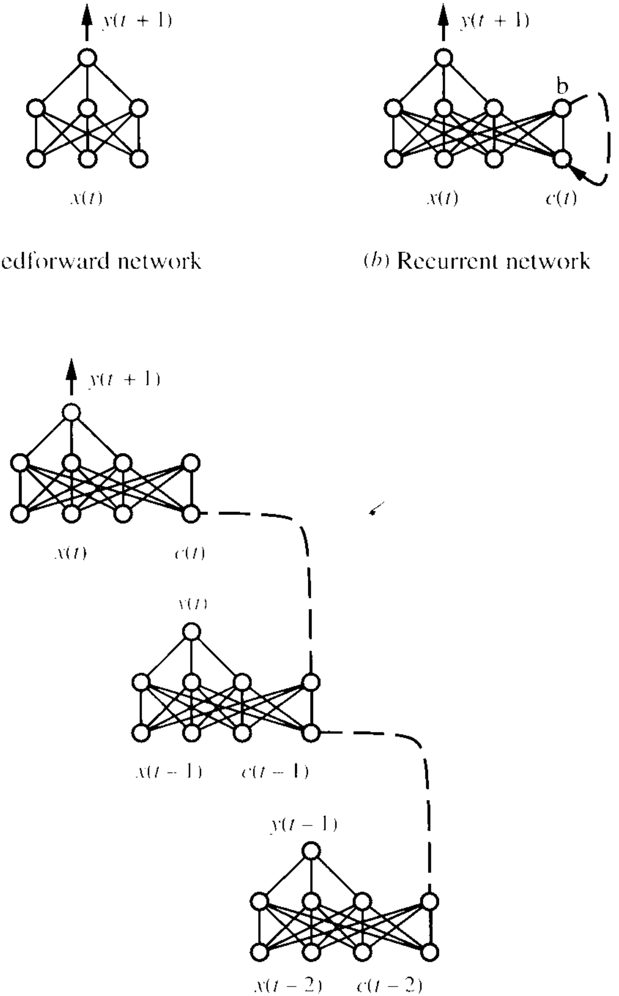
Alternative Error Functions

Alternative Error Minimization Procedures

Recurrent Networks

Dynamically Modifying Network Structure

Recurrent Neural Networks



(a) Feedforward network

(b) Recurrent network

(c) Recurrent network
unfolded in time

Problems with NN

A lot of parameters

Adding random input variables or output variables makes it work better

Humans can’t understand the results

Summary

Practical method for learning real-valued and vector-valued functions over continuous and discrete-valued attributes

Robust to noise in the training data

Backpropagation algorithm is the most common

Hypothesis space: all functions that can be represented by assigning weights to the fixed network of interconnected units

Feedforward networks containing 3 layers can approximate any function to arbitrary accuracy given sufficient number of units in each layer

Summary II

Networks of practical size are capable of representing a rich space of highly nonlinear functions

Backpropagation searches the space of possible hypotheses using gradient descent (GD) to iteratively reduce the error in the network to fit the training data.

GD converges to a local minimum in the training error with respect to the network weights

Backpropagation has the ability to invent new features that are not explicit in the input

Summary III

Hidden units of multilayer networks learn to represent intermediate features (e.g., face recognition)

Overfitting is an important issue (caused by overuse of accuracy IMHO)

Cross-validation can be used to estimate an appropriate stopping point for gradient descent

Many other algorithms and extensions.

Questions you should be able to answer

- What does the hypothesis space of an ANN look like?
- The difference between Gradient Descent and Stochastic Gradient Descent?
- Why don’t ANNs get stuck in local maxima?
- What is the inductive bias of backprop?
- What is the main advantage of the hidden units?
- What aspect makes an ANN overfit?
- How important is the architecture?
- How do you determine a “good” architecture?

References

- Machine Learning, Tom Mitchell, chapter 4