Data Mining

Lesson 8

Neural Networks

MSc in Computer Science

University of New York Tirana

Assoc. Prof. Dr. Marenglen Biba

Data Mining: Content

- Introduction to data mining and machine learning
- Inductive learning
- Decision trees
- Rule induction
- Instance-based learning
- Bayesian learning
- Neural networks
- Support vector machines
- Other machine learning models
- Engineering data mining tasks

Introduction

- Artificial neural networks (ANNs) provide a general, practical method for learning real-valued, discrete-valued, and vector-valued functions from examples.
- ANN learning is robust to errors in the training data and has been successfully applied to important problems.
- For example, the Backpropagation algorithm has proven surprisingly successful in many practical problems such as:
  - learning to recognize handwritten characters (LeCun et al. 1989),
  - learning to recognize spoken words (Lang et al. 1990), and
  - learning to recognize faces (Cottrell 1990).

A long journey

- http://vislab.it/
- http://viac.vislab.it/
  - Prepared and tested to drive with no human intervention from Parma, Italy, to Shanghai, China.
- Some more autonomous cars:
  - http://en.wikipedia.org/wiki/Autonomous_car

- Vehicle classification through multiple neural networks
  - http://www.youtube.com/watch?v=IsNLzhu_BKM

- Neural Network driving
  - http://www.youtube.com/watch?v=DWNtsS2kZWs

Connectionist Models

Consider humans:

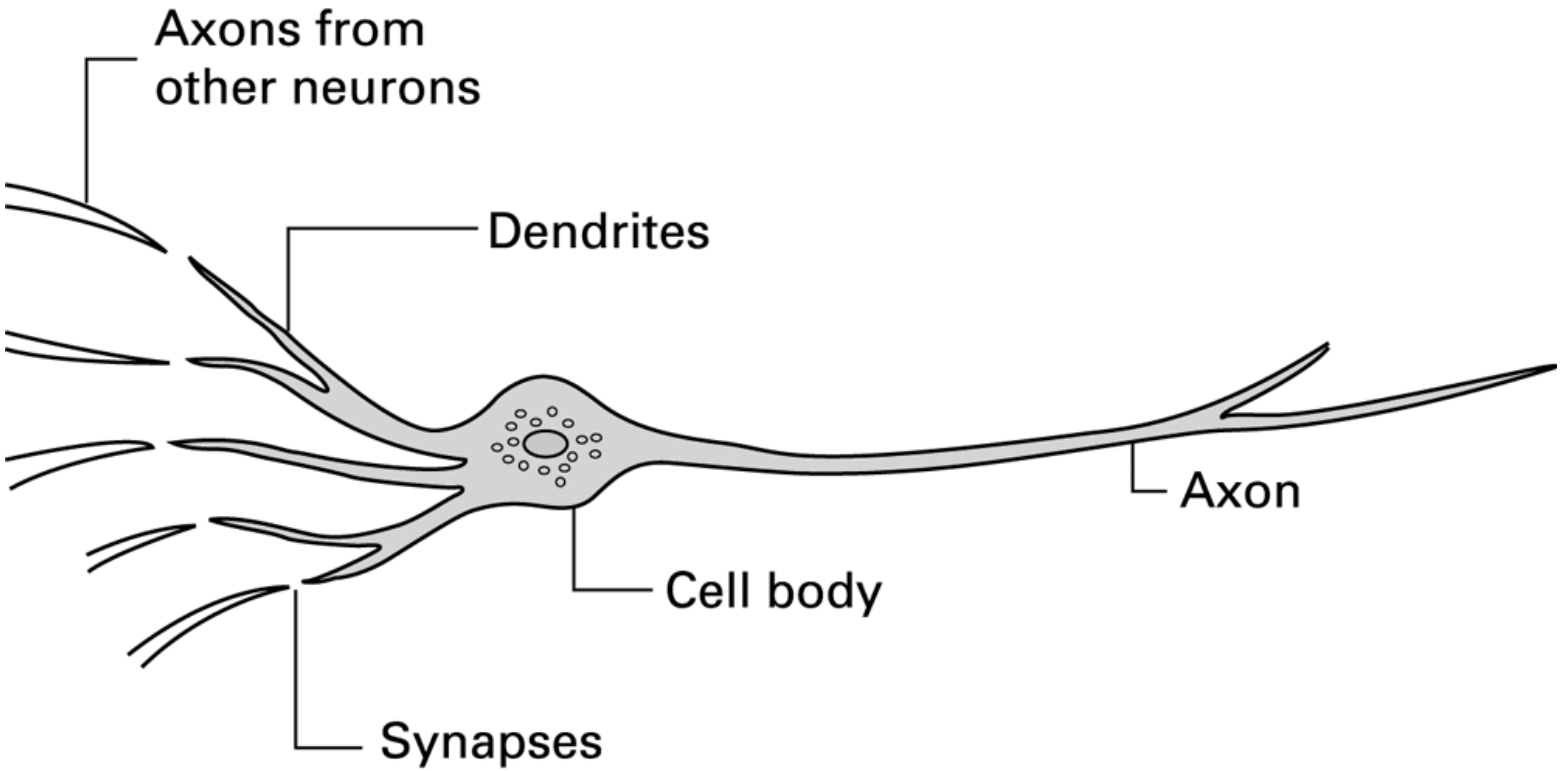
- Neuron switching time $\sim$ .001 second
- Number of neurons $\sim 10^{10}$
- Connections per neuron $\sim 10^{4-5}$
- Scene recognition time $\sim$ .1 second
- 100 inference steps doesn't seem like enough

$\Rightarrow$ Much parallel computation

Properties of neural nets:

- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically

A neuron in a living biological system

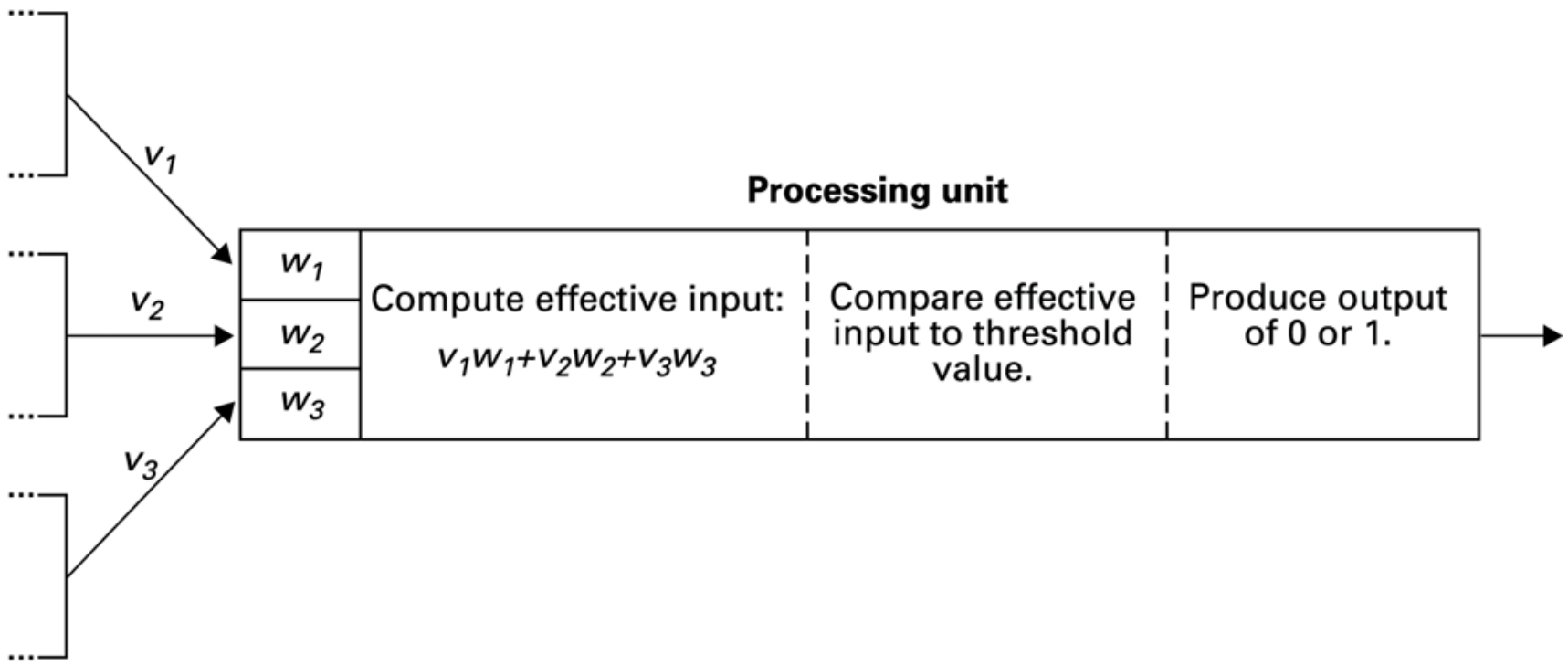


Artificial Neural Networks

- Artificial Neuron
  - Each input is multiplied by a weighting factor.
  - Output is 1 if sum of weighted inputs exceeds the threshold value; 0 otherwise.

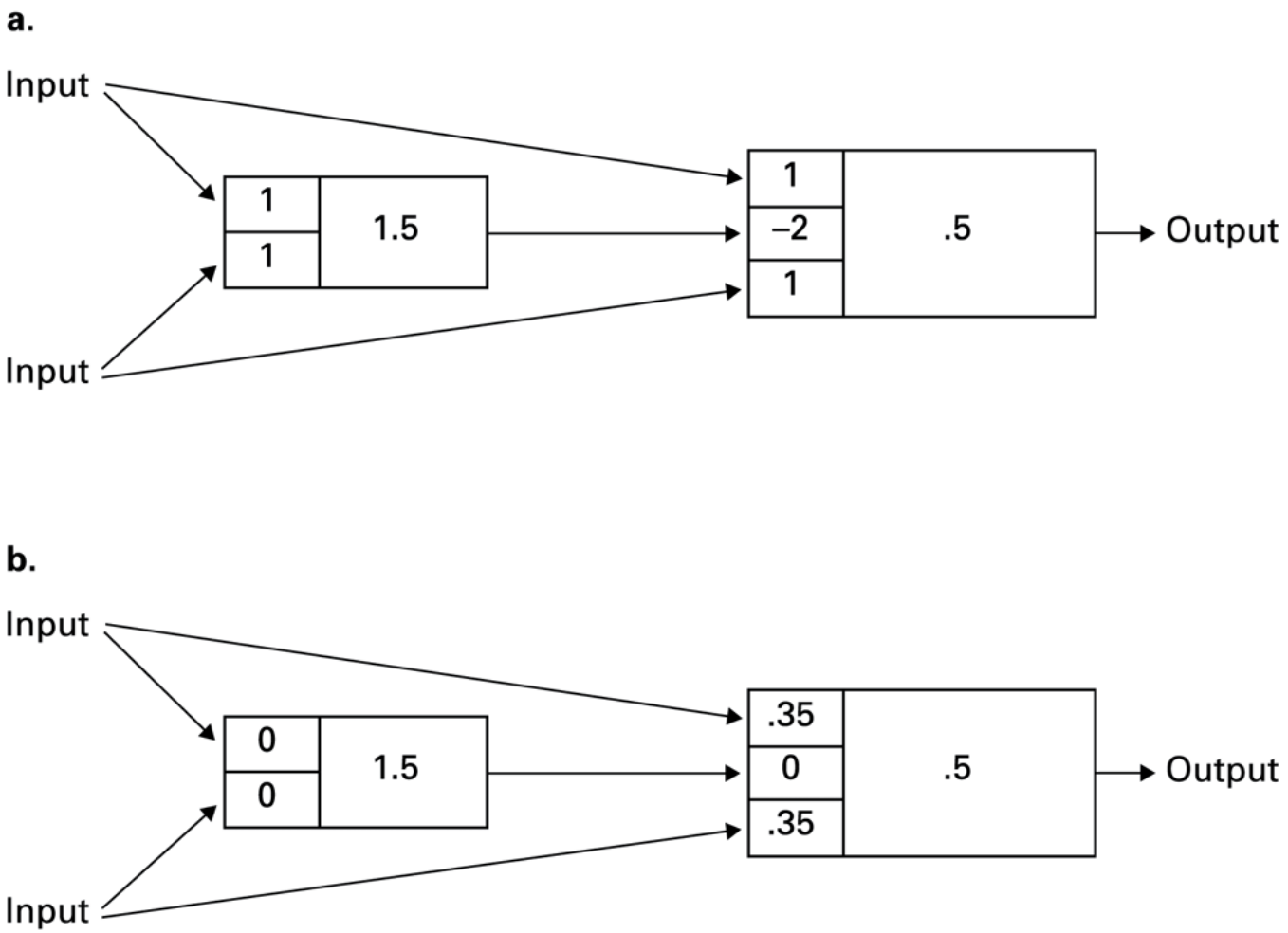
- Network is programmed by adjusting weights using feedback from examples.

Figure 11.16 The activities within a processing unit



Representation of a processing unit

A neural network with two different programs



An artificial neural network

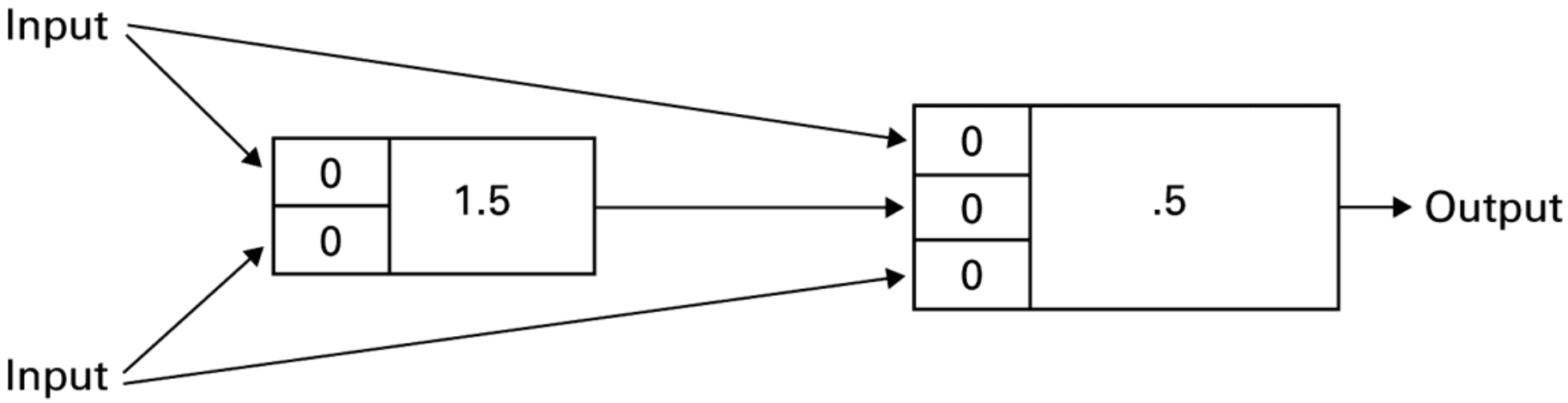
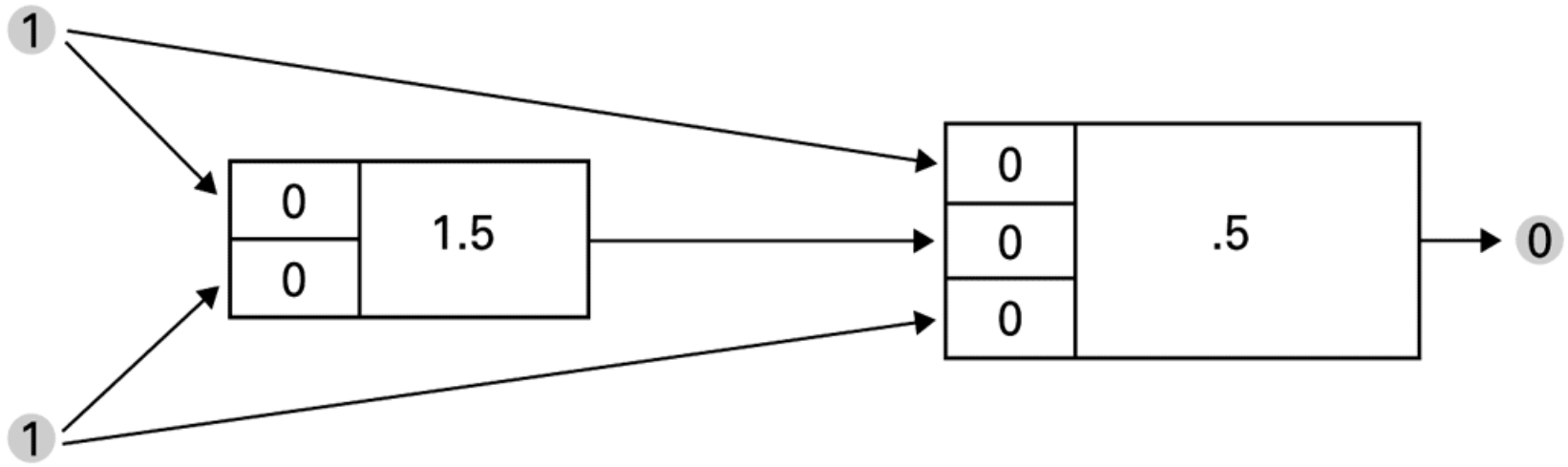
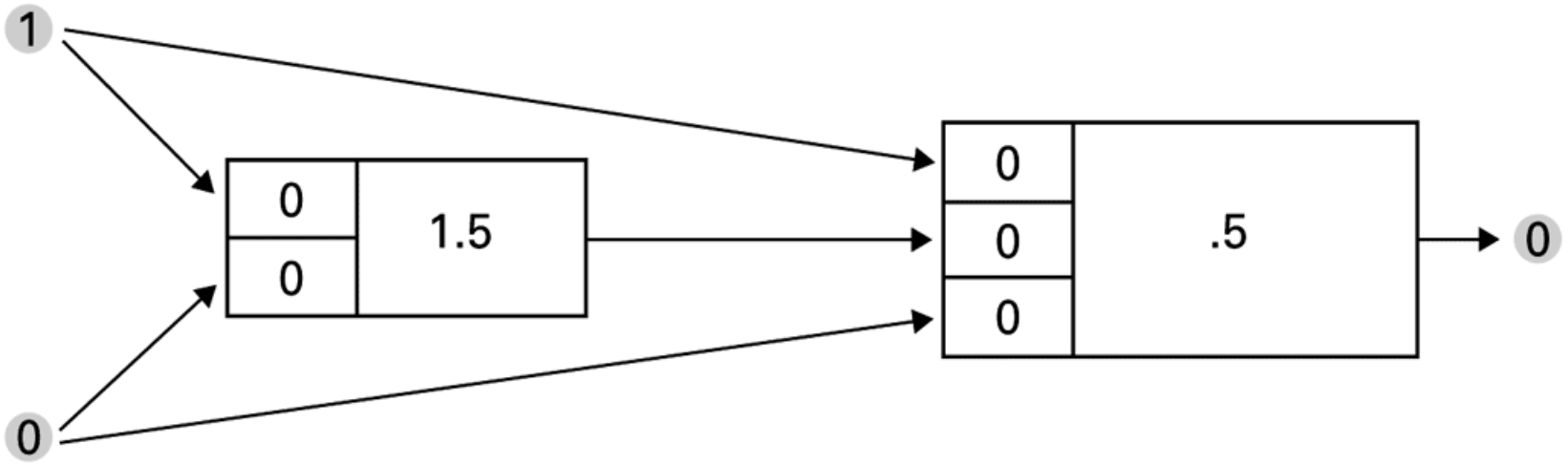


Figure 11.20 Training an artificial neural network



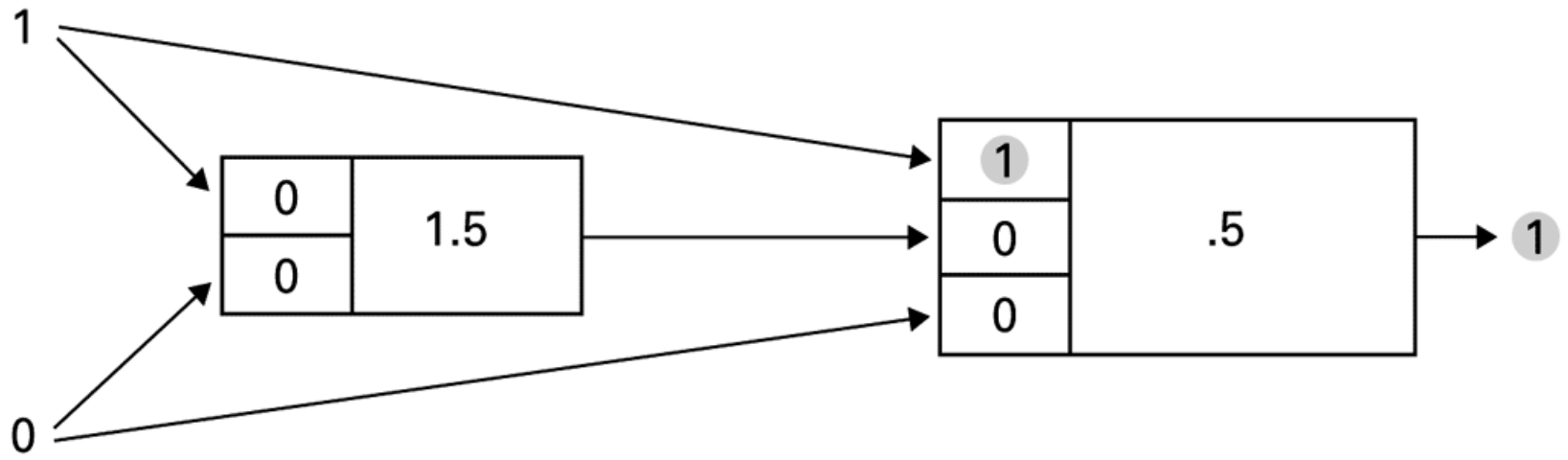
a. The network performs correctly for the input pattern 1, 1.

Figure 11.20 Training an artificial neural network (continued)



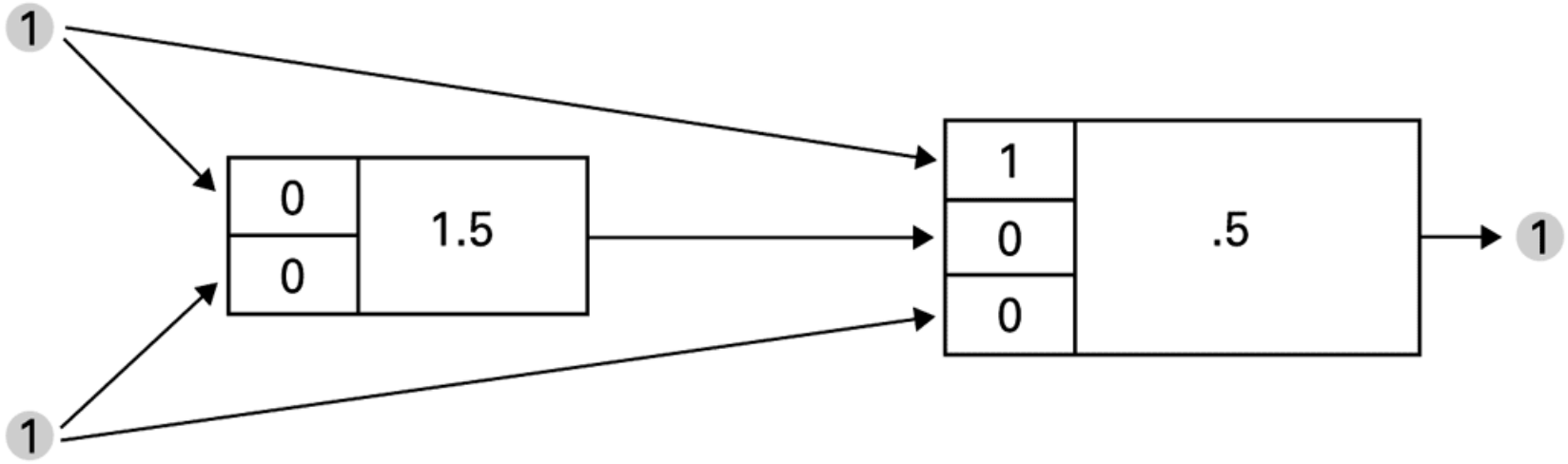
b. The network performs incorrectly for the input pattern 1, 0.

Figure 11.20 Training an artificial neural network (continued)



c. The upper weight in the second processing unit is adjusted.

Figure 11.20 Training an artificial neural network (continued)



d. However, the network no longer performs correctly for the input pattern 1, 1.

We continue to update the weights

Neural Network Representations

- A prototypical example of ANN learning is provided by Pomerleau's (1993) system ALVINN, which uses a learned ANN to steer an autonomous vehicle driving at normal speeds on public highways.

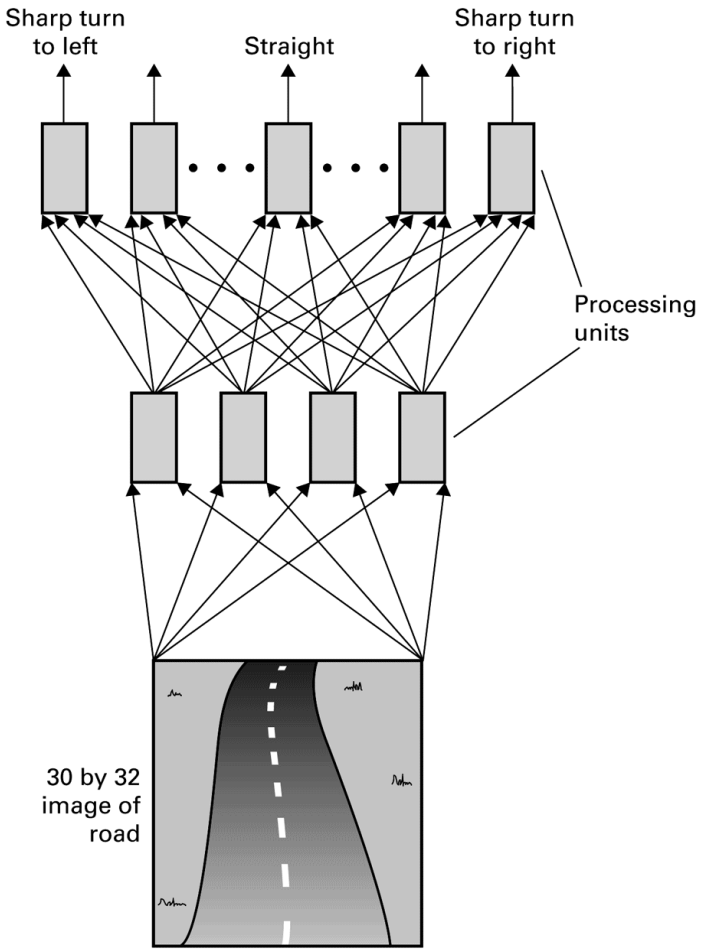
- The input to the neural network is a 30 x 32 grid of pixel intensities obtained from a forward-pointed camera mounted on the vehicle.

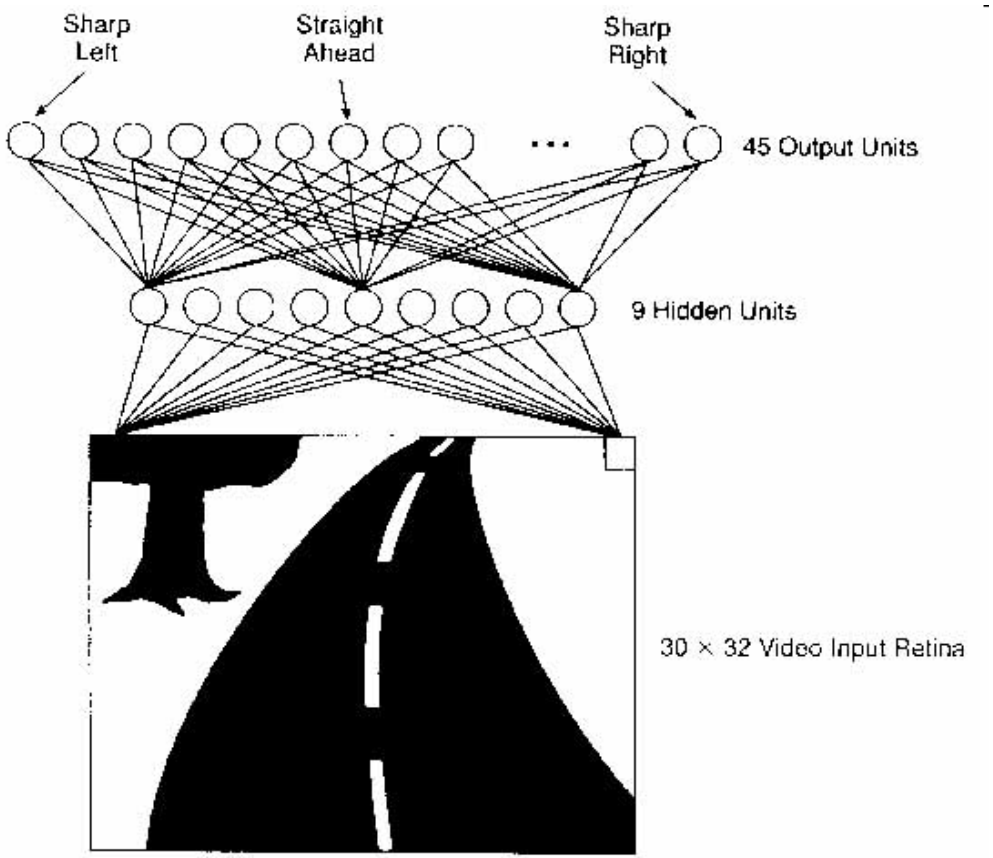
- The network output is the direction in which the vehicle is steered.

- The ANN is trained to mimic the observed steering commands of a human driving the vehicle for approximately 5 minutes.

- ALVINN has used its learned networks to successfully drive at speeds up to 70 miles per hour and for distances of 90 miles on public highways (driving in the left lane of a divided public highway, with other vehicles present).

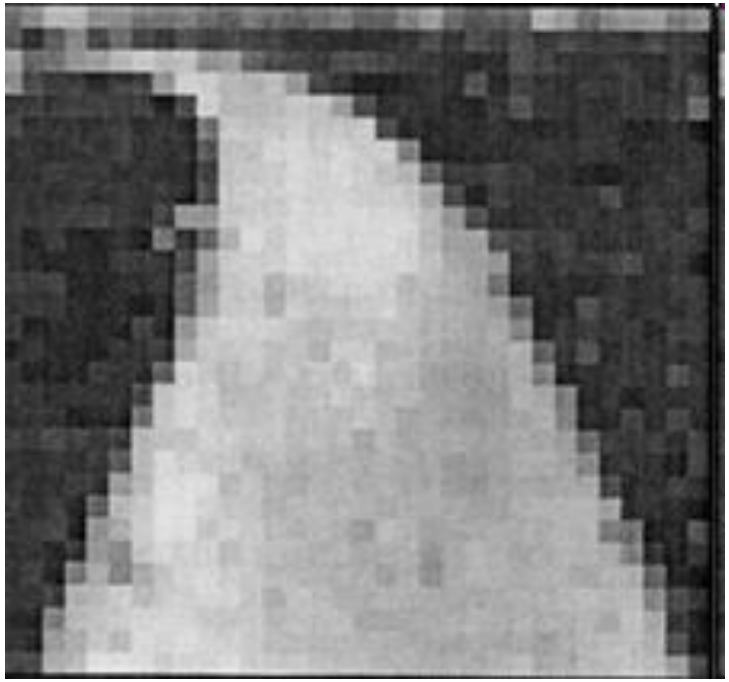
Figure 11.21 The structure of ALVINN

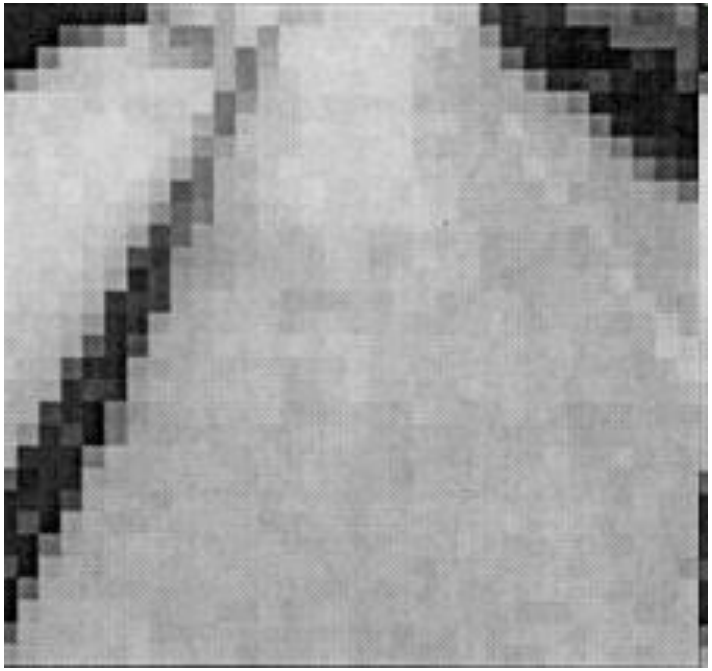




Multilayer Network Control Unit for Navlab
(From Kanade et al., 1994. © 1994 ACM. Courtesy of the authors.)

ALVINN: one and two lane roads





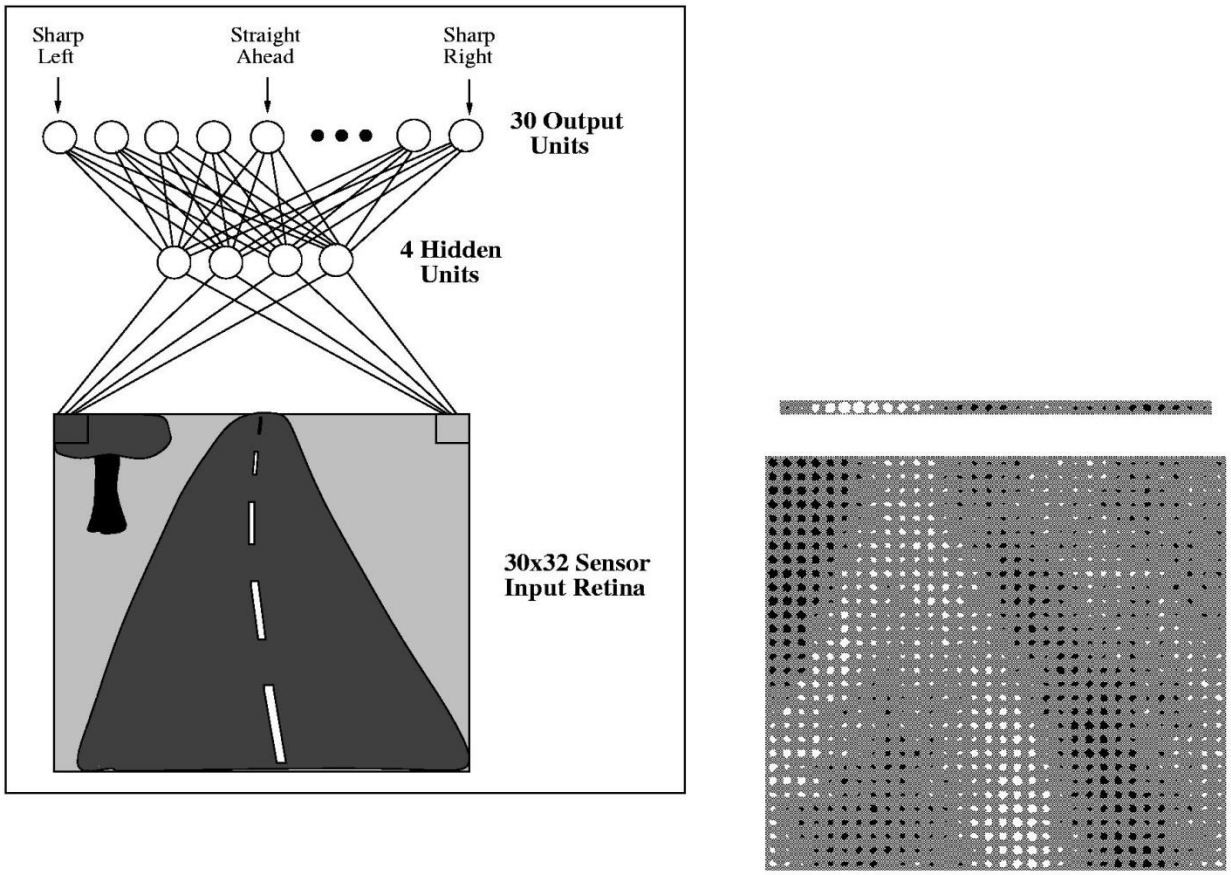
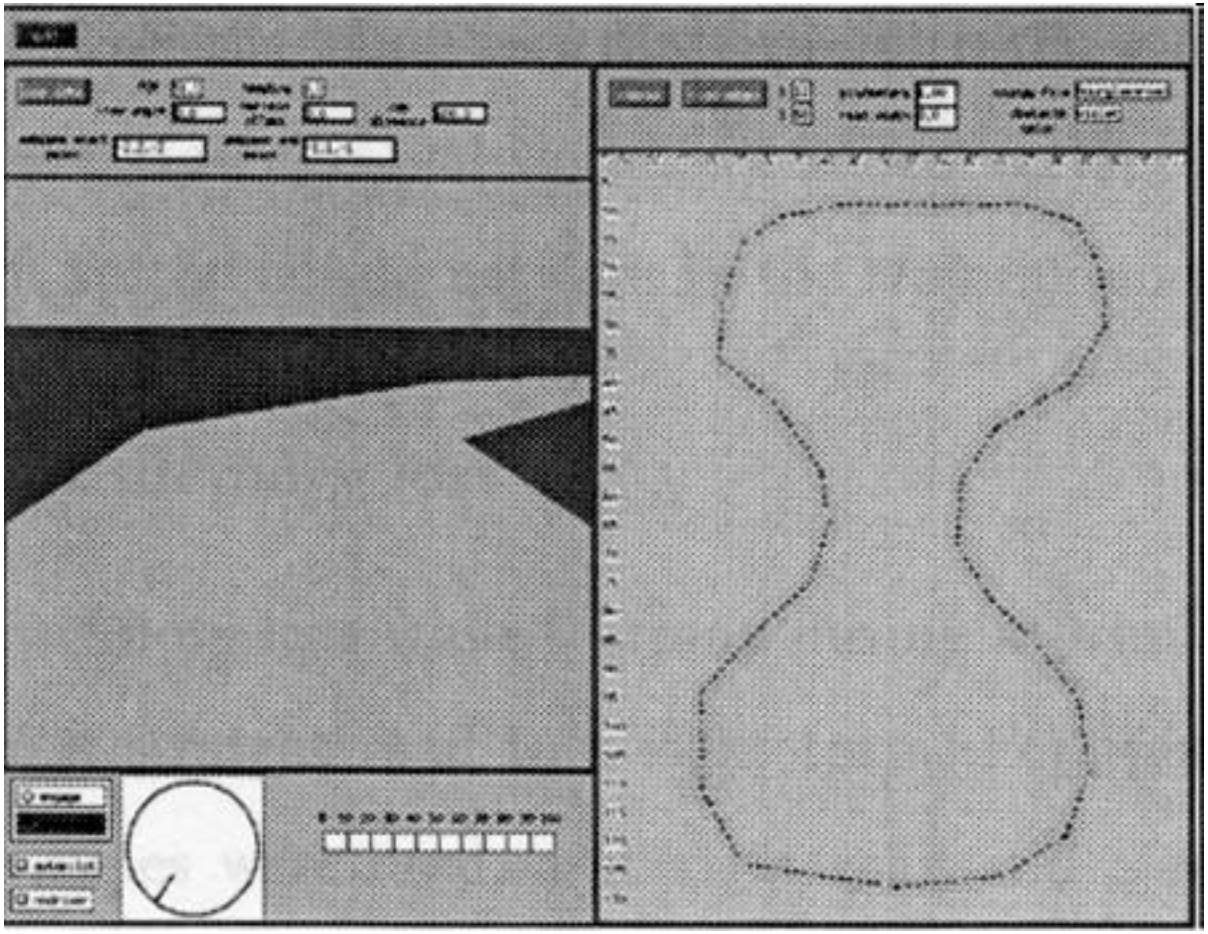


FIGURE 4.1
Neural network learning to steer an autonomous vehicle. The ALVINN system uses BACKPROPAGATION to learn to steer an autonomous vehicle (photo at top) driving at speeds up to 70 miles per hour. The diagram on the left shows how the image of a forward-mounted camera is mapped to 960 neural network inputs, which are fed forward to 4 hidden units, connected to 30 output units. Network outputs encode the commanded steering direction. The figure on the right shows weight values for one of the hidden units in this network. The 30×32 weights into the hidden unit are displayed in the large matrix, with white blocks indicating positive and black indicating negative weights. The weights from this hidden unit to the 30 output units are depicted by the smaller rectangular block directly above the large block. As can be seen from these output weights, activation of this particular hidden unit encourages a turn toward the left.

Simulator for ALVINN



The Structure of ALVINN

- The network structure of ALVINN is typical of many ANNs.
- Here the individual units are interconnected in layers that form a directed acyclic graph.
- In general, ANNs can be graphs with many types of structures — acyclic or cyclic, directed or undirected.
- Learning corresponds to choosing a weight value for each edge in the graph.
- Although certain types of cycles are allowed, the vast majority of practical applications involve acyclic feed-forward networks, similar to the network structure used by ALVINN.

Appropriate Problems For Neural Network Learning

- ANN learning is well-suited to problems in which the training data corresponds to noisy, complex sensor data, such as inputs from cameras and microphones.

- It is also applicable to problems for which more symbolic representations are often used, such as the decision tree learning tasks discussed in Lesson 3.
  - In these cases ANN and decision tree learning often produce results of comparable accuracy.

Appropriate Problems For Neural Network Learning

- The Backpropagation algorithm is the most commonly used ANN learning technique.
- It is appropriate for problems with the following characteristics:

1. *Instances are represented by many attribute-value pairs.*
   - The target function to be learned is defined over instances that can be described by a vector of predefined features, such as the pixel values in the ALVINN example.
   - These input attributes may be highly correlated or independent of one another.
   - Input values can be any real values.

Appropriate Problems For Neural Network Learning

2. *The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes.*
   - For example, in the ALVINN system the output is a vector of 30 attributes, each corresponding to a recommendation regarding the steering direction.
   - The value of each output is some real number between 0 and 1, which in this case corresponds to the confidence in predicting the corresponding steering direction.
   - We can also train a single network to output both the steering command and suggested acceleration, simply by concatenating the vectors that encode these two output predictions.

Appropriate Problems For Neural Network Learning

3. *The training examples may contain errors.*
   - ANN learning methods are quite robust to noise in the training data.
4. *Long training times are acceptable.*
   - Network training algorithms typically require longer training times than, say, decision tree learning algorithms.
   - Training times can range from a few seconds to many hours, depending on factors such as the number of weights in the network, the number of training examples considered, and the settings of various learning algorithm parameters.

Appropriate Problems For Neural Network Learning

5. Fast evaluation of the learned target function may be required.
   - Although ANN learning times are relatively long, evaluating the learned network, in order to apply it to a subsequent instance, is typically very fast.
   - For example, ALVINN applies its neural network several times per second to continually update its steering command as the vehicle drives forward.
6. The ability of humans to understand the learned target function is not important.
   - The weights learned by neural networks are often difficult for humans to interpret.
   - Learned neural networks are less easily communicated to humans than learned rules.

Alternative design for primitive units

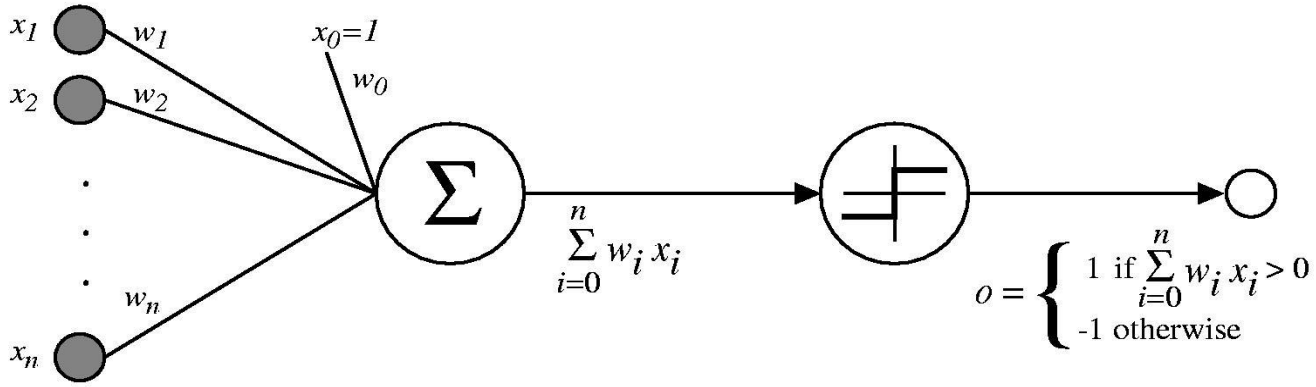
- We consider several alternative designs for the primitive units that make up artificial neural networks:
  - perceptrons
  - linear units
  - sigmoid units
- along with learning algorithms for training single units.

Perceptron

- One type of ANN system is based on a unit called a perceptron.

- A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and -1 otherwise.

Perceptron



$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Sometimes we'll use simpler vector notation:

$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Learning a perceptron

- Learning a perceptron involves choosing values for the weights w_0,..., w_n.

- Therefore, the *space H of candidate hypotheses* considered in perceptron learning is the set of all possible real-valued weight vectors.

Decision surface of a Perceptron

- We can view the perceptron as representing a hyperplane decision surface in the n-dimensional space of instances (i.e., points).

- The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputs a -1 for instances lying on the other side.

- Of course, some sets of positive and negative examples cannot be separated by any hyperplane.
  - Those that can be separated are called linearly separable sets of examples.

Decision surface of a Perceptron

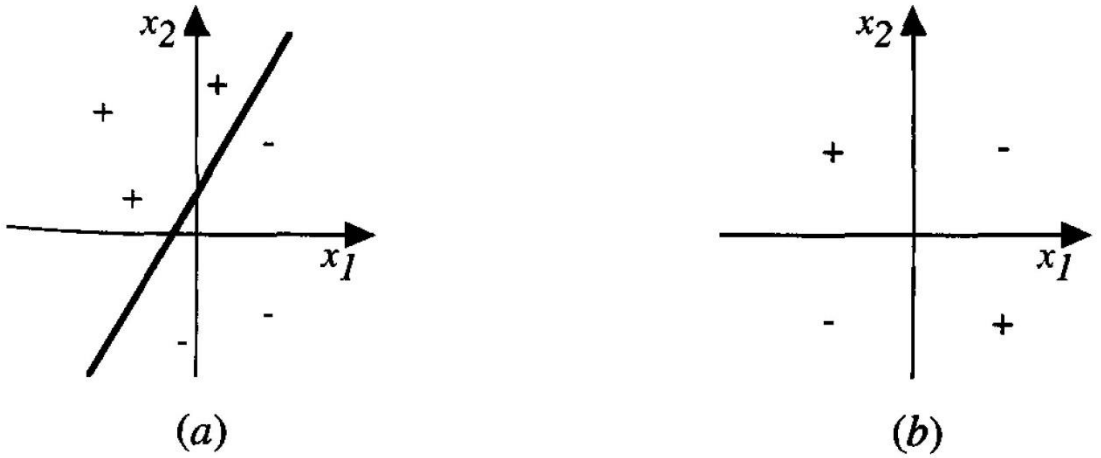


FIGURE 4.3
The decision surface represented by a two-input perceptron. (*a*) A set of training examples and the decision surface of a perceptron that classifies them correctly. (*b*) A set of training examples that is not linearly separable (i.e., that cannot be correctly classified by any straight line). x_1 and x_2 are the perceptron inputs. Positive examples are indicated by "+", negative by "−".

Expressive power of Perceptron

- A single perceptron can be used to represent many boolean functions.

- In fact, AND and OR can be viewed as special cases of m-of-n functions; that is, functions where at least m of the n inputs to the perceptron must be true.

Expressive power of Perceptron

- Perceptrons can represent all of the primitive boolean functions AND, OR, NAND (¬ AND), and NOR (¬ OR).

- Unfortunately, however, some boolean functions cannot be represented by a single perceptron, such as the XOR function.

- Note the set of linearly nonseparable training examples shown in Figure 4.3(b) corresponds to this XOR function.

Expressive power of Perceptron

- The ability of perceptrons to represent AND, OR, NAND, and NOR is important because every boolean function can be represented by some network of interconnected units based on these primitives.

- In fact, every boolean function can be represented by some network of perceptrons only two levels deep, in which the inputs are fed to multiple units, and the outputs of these units are then input to a second, final stage.

- Because networks of threshold units can represent a rich variety of functions and because single units alone cannot, we will generally be interested in learning multilayer networks of threshold units.

Training of Neural Networks

The Perceptron Training Rule

- Although we are interested in learning networks of many interconnected units, let us begin by understanding how to learn the weights for a single perceptron.
- Here the precise learning problem is to determine a weight vector that causes the perceptron to produce the correct ±1 output for each of the given training examples.
- Several algorithms are known to solve this learning problem.
- Here we consider two: the perceptron rule and the delta rule (a variant of the LMS rule used in Lesson 1 for learning evaluation functions).
- These two algorithms are guaranteed to converge to somewhat different acceptable hypotheses, under somewhat different conditions.
- They are important to ANNs because they provide the basis for learning networks of many units.

The Perceptron Training Rule

- One way to learn an acceptable weight vector is to *begin with random weights*, then iteratively apply the perceptron to each training example, modify the perceptron weights whenever it misclassifies an example.
- This process is repeated, iterating through the training examples as many times as needed until the perceptron classifies all training examples correctly.
- Weights are modified at each step according to the perceptron training rule, which revises the weight w, associated with input x_i, according to the rule:

$$w_i \leftarrow w_i + \Delta w_i$$

where

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

- Here t is the target output for the current training example, *o* is the output generated by the perceptron, and η is a positive constant called the learning rate.
- The role of the learning rate is to moderate the degree to which weights are changed at each step. It is usually set to some small value (e.g., 0.1) and is sometimes made to decrease as the number of weight-tuning iterations increases.

Convergence of the learning procedure

- The above learning procedure can be proven to converge within a finite number of applications of the perceptron training rule to a weight vector that correctly classifies all training examples, provided:
  - the training examples are linearly separable and
  - provided a sufficiently small η is used

  (Minsky and Papert 1969).

- If the data are not linearly separable, convergence is not assured.

Gradient Descent and the Delta Rule

- Although the perceptron rule finds a successful weight vector when the training examples are linearly separable, it can *fail to converge if the examples are not linearly separable*.

- A second training rule, called the *delta rule*, is designed to overcome this difficulty.

- If the training examples are not linearly separable, the delta rule converges toward a best-fit approximation to the target concept.

The Delta Rule

- The key idea behind the delta rule is to use gradient descent to search the hypothesis space of possible weight vectors to find the weights that best fit the training examples.

- This rule is important because gradient descent provides the basis for the Backpropagation algorithm, which can learn networks with many interconnected units.

- It is also important because gradient descent can serve as the basis for learning algorithms that must search through hypothesis spaces containing many different types of continuously parameterized hypotheses.

The delta rule

- The delta training rule is best understood by considering the task of training an unthresholded perceptron; that is, a linear unit for which the output o is given by:

$$o(\vec{x}) = \vec{w} \cdot \vec{x} \qquad (4.1)$$

- Thus, a linear unit corresponds to the first stage of a perceptron, without the threshold.
- In order to derive a weight learning rule for linear units, let us begin by specifying a measure for the training error of a hypothesis (weight vector), relative to the training examples: squared error => next slide

Squared Error

- Although there are many ways to define this error, one common measure that will turn out to be especially convenient is:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \qquad (4.2)$$

- where D is the set of training examples, t_d is the target output for training example d, and o_d is the output of the linear unit for training example *d*.
- By this definition, E(w) is simply half the squared difference between the target output t_d and the linear unit output o_d, summed over all training examples.

Minimizing E and most probable hypothesis

- E also depends on the particular set of training examples, but we assume these are fixed during training, so we do not bother to write E as an explicit function of these.

- In Lesson 6 we provided a Bayesian justification for choosing this particular definition of E.

- In particular, there we showed that under certain conditions the hypothesis that minimizes E is also the most probable hypothesis in H given the training data.

Gradient descent search

- Gradient descent search determines a weight vector that minimizes E by starting with an arbitrary initial weight vector, then repeatedly modifying it in small steps.

- At each step, the weight vector is altered in the direction that produces the steepest descent along the error surface depicted in Figure 4.4.

- This process continues until the global minimum error is reached.

Hypothesis Space

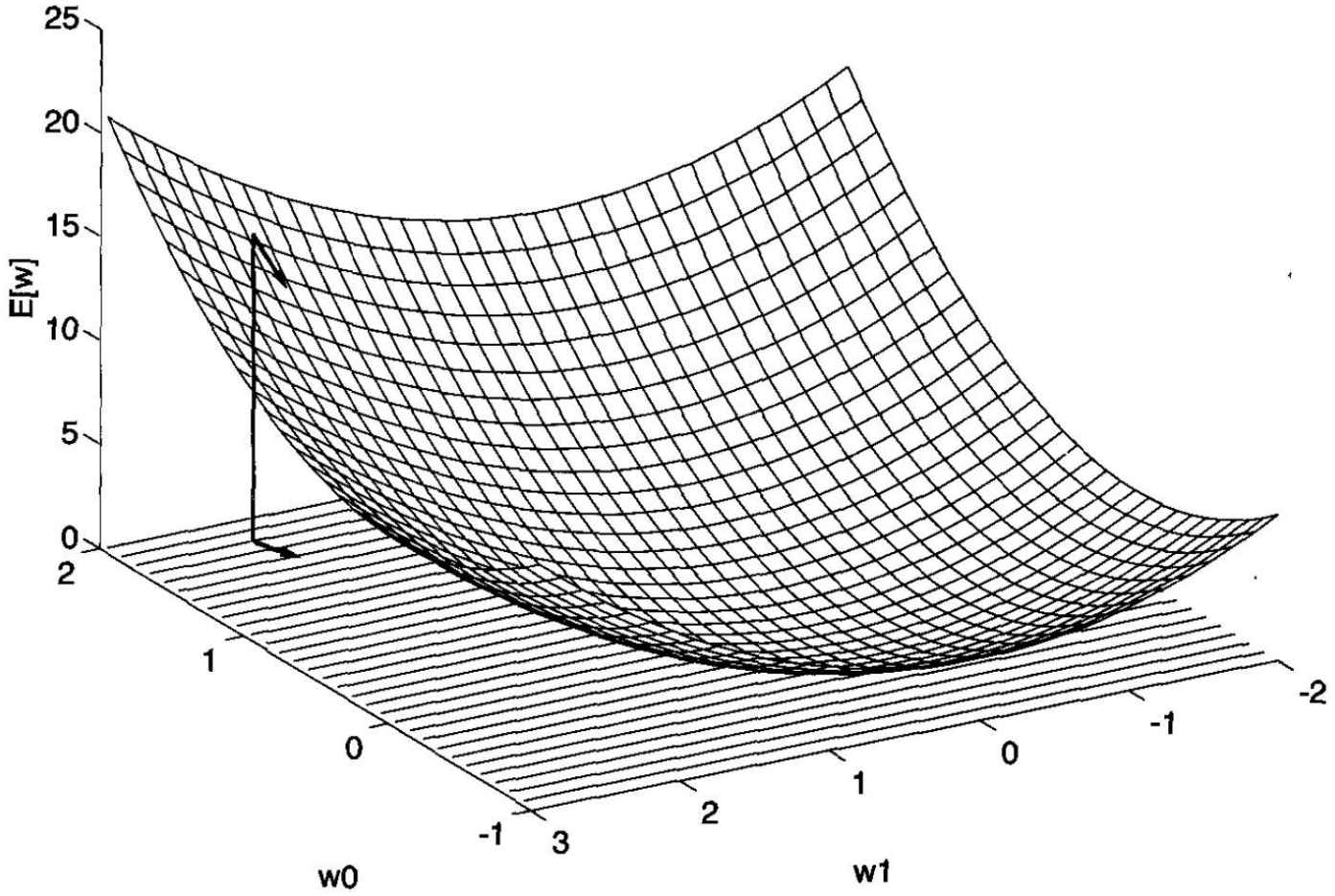


FIGURE 4.4
Error of different hypotheses. For a linear unit with two weights, the hypothesis space H is the w_0, w_1 plane. The vertical axis indicates the error of the corresponding weight vector hypothesis, relative to a fixed set of training examples. The arrow shows the negated gradient at one particular point, indicating the direction in the w_0, w_1 plane producing steepest descent along the error surface.

Derivation Of The Gradient Descent Rule

- How can we calculate the direction of steepest descent along the error surface?
- This direction can be found by computing the derivative of E with respect to each component of the vector $\vec{w}$.
- This vector derivative is called the gradient of E with respect to w, written $\nabla E(\vec{w})$.

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n} \right] \qquad (4.3)$$

- Notice $\nabla E(\vec{w})$ is itself a vector, whose components are the partial derivatives of E with respect to each of the w.
- When interpreted as a vector in weight space, the gradient specifies the direction that produces the steepest increase in E.
- The negative of this vector therefore gives the direction of steepest decrease.
- For example, the arrow in Figure 4.4 shows the negated gradient $-\nabla E(\vec{w})$ for a particular point in the w_0, w_1 plane.

Derivation Of The Gradient Descent Rule

- Since the gradient specifies the direction of steepest increase of E, the training rule for gradient descent is:

$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w}$$

where

$$\Delta\vec{w} = -\eta\nabla E(\vec{w}) \qquad (4.4)$$

- Here η is a positive constant called the learning rate, which determines the step size in the gradient descent search.
- The negative sign is present because we want to move the weight vector in the direction that decreases E. This training rule can also be written in its component form:

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \qquad (4.5)$$

- which makes it clear that steepest descent is achieved by altering each component w_i of $\vec{w}$ in proportion to $\partial E / \partial w_i$

Derivation Of The Gradient Descent Rule

- To construct a practical algorithm for iteratively updating weights according to Equation (4.5), we need an efficient way of calculating the gradient at each step.
- Fortunately, this is not difficult. The vector of derivatives that form the gradient can be obtained by differentiating E from Equation (4.2), as

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\
&= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x}_d) \\
\frac{\partial E}{\partial w_i} &= \sum_{d \in D} (t_d - o_d)(-x_{id}) \qquad (4.6)
\end{aligned}$$

- where x_{id} denotes the single input component x_i, for training example d.

Update rule of gradient descent

- We now have an equation that gives $\partial E / \partial w_i$ in terms of the linear unit inputs x_{id}, outputs O_d and target values t_d associated with the training examples.

- Substituting Equation (4.6) into Equation (4.5) yields the weight update rule for gradient descent:

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d)\, x_{id} \qquad (4.7)$$

The 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. η 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 Δw_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

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i \tag{T4.1}$$

  - For each linear unit weight w_i, Do

$$w_i \leftarrow w_i + \Delta w_i \tag{T4.2}$$

TABLE 4.1

GRADIENT DESCENT algorithm for training a linear unit. To implement the stochastic approximation to gradient descent, Equation (T4.2) is deleted, and Equation (T4.1) replaced by $w_i \leftarrow w_i + \eta(t - o)x_i$.

Summary of gradient descent

- To summarize, the gradient descent algorithm for training linear units is as follows:
- *Pick an initial random weight vector.*
- *Apply the linear unit to all training examples,*
- *Then compute Δw_i for each weight according to Equation (4.7).*
- *Update each weight wi by adding Δwi then repeat this process.*

Convergence of gradient descent

- Because the error surface contains only a *single global minimum*, this 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.
- If η is too large, the gradient descent search runs the risk of overstepping the minimum in the error surface rather than settling into it.
- For this reason, one common modification to the algorithm is to gradually reduce the value of η as the number of gradient descent steps grows => experiments in Weka today

Remarks

- We have considered two similar algorithms for iteratively learning perceptron weights.

- The key difference between these algorithms is that the perceptron training rule updates weights based on the error in the thresholded perceptron output, whereas the delta rule updates weights based on the error in the unthresholded linear combination of inputs.

Remarks

- The difference between these two training rules is reflected in different convergence properties.

- The perceptron training rule converges after a finite number of iterations to a hypothesis that perfectly classifies the training data, provided the training examples are linearly separable.

- The delta rule converges only asymptotically toward the minimum error hypothesis, possibly requiring unbounded time, but converges regardless of whether the training data are linearly separable.

Multilayer Networks and The Backpropagation Algorithm

Multilayer Networks and the Backpropagation Algorithm

- Single perceptrons can only express linear decision surfaces.
- In contrast, the kind of multilayer networks learned by the Backpropagation algorithm are capable of expressing a rich variety of nonlinear decision surfaces.
- For example, a typical multilayer network and decision surface is depicted in Figure 4.5. Here the speech recognition task involves distinguishing among 10 possible vowels, all spoken in the context of "h_d" (i.e., "hid," "had," "head," "hood," etc.).
- The input speech signal is represented by two numerical parameters obtained from a spectral analysis of the sound, allowing us to easily visualize the decision surface over the two-dimensional instance space.
- As shown in the figure, it is possible for the multilayer network to represent highly nonlinear decision surfaces that are much more expressive than the linear decision surfaces of single units shown earlier in Figure 4.3.

Multilayer feedforward Networks

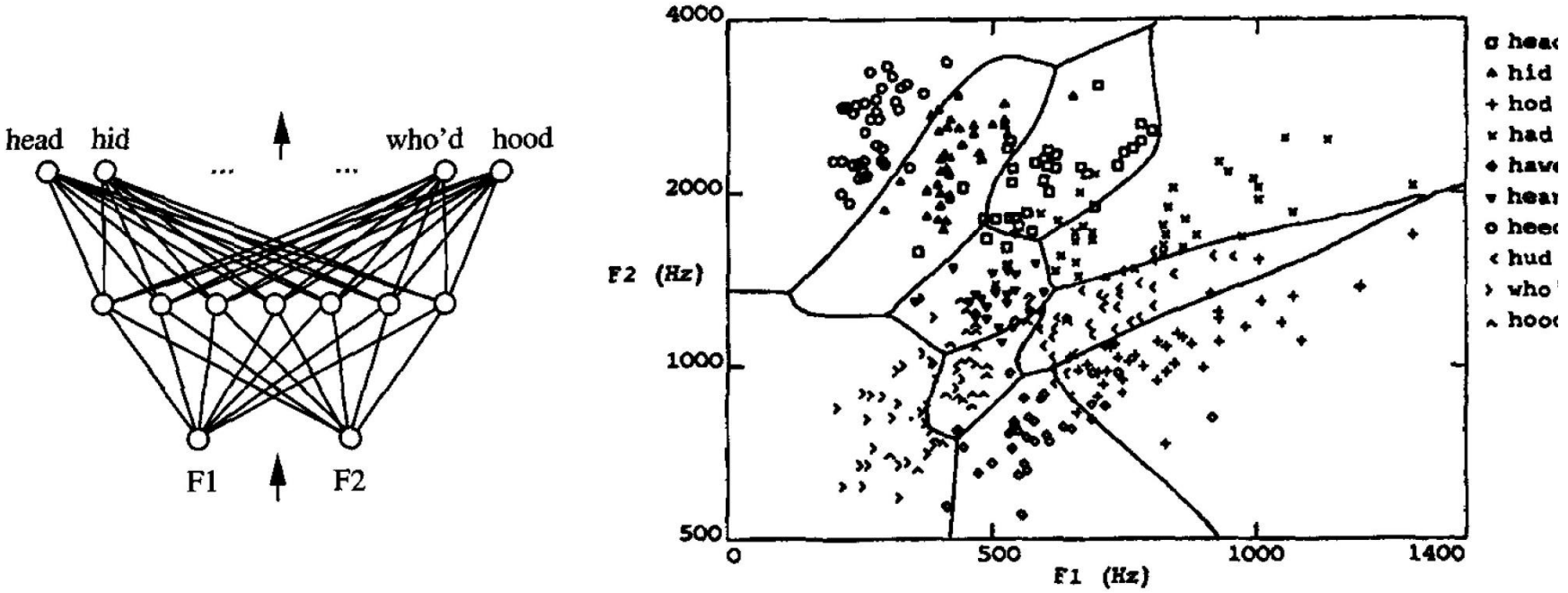


FIGURE 4.5
Decision regions of a multilayer feedforward network. The network shown here was trained to recognize 1 of 10 vowel sounds occurring in the context "h_d" (e.g., "had," "hid"). The network input consists of two parameters, F1 and F2, obtained from a spectral analysis of the sound. The 10 network outputs correspond to the 10 possible vowel sounds. The network prediction is the output whose value is highest. The plot on the right illustrates the highly nonlinear decision surface represented by the learned network. Points shown on the plot are test examples distinct from the examples used to train the network. (Reprinted by permission from Haung and Lippmann (1988).)

A Differentiable Threshold Unit

- What type of unit shall we use as the basis for constructing multilayer networks?
- At first we might be tempted to choose the linear units discussed previously, for which we have already derived a gradient descent learning rule.
- However, multiple layers of cascaded linear units still produce only linear functions, and we prefer networks capable of representing highly nonlinear functions.
- The perceptron unit is another possible choice, but its discontinuous threshold makes it *undifferentiable* and hence unsuitable for gradient descent.
- *What we need is a unit whose output is a nonlinear function of its inputs, but whose output is also a differentiable function of its inputs.*
- One solution is the sigmoid unit — a unit very much like a perceptron, but based on a smoothed, differentiable threshold function.

The sigmoid unit

- Like the perceptron, the sigmoid unit first computes a linear combination of its inputs, then applies a threshold to the result.
- In the case of the sigmoid unit, however, the threshold output is a continuous function of its input. More precisely, the sigmoid unit computes its output *o* as:

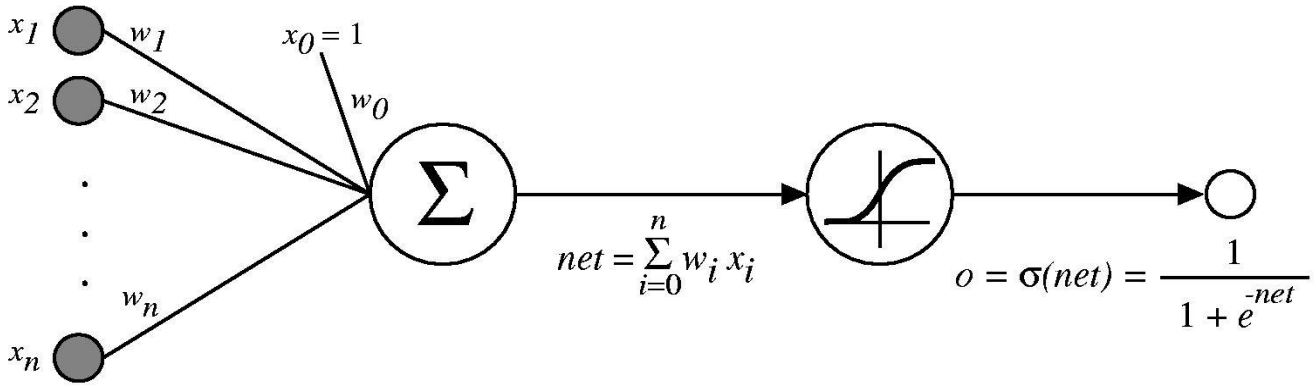
$$o = \sigma(\vec{w} \cdot \vec{x})$$

where

$$\sigma(y) = \frac{1}{1 + e^{-y}} \qquad (4.12)$$

- σ is often called the sigmoid function or, alternatively, the logistic function.
- Note its output ranges between 0 and 1, increasing monotonically with its input.

Sigmoid Unit



$\sigma(x)$ is the sigmoid function

$$\frac{1}{1+e^{-x}}$$

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x))$

As we shall see, the gradient descent learning rule makes use of this derivative.

The Backpropagation Algorithm

- The Backpropagation algorithm learns the weights for a multilayer network, given a network with a fixed set of units and interconnections.

- *It employs gradient descent to attempt to minimize the squared error between the network output values and the target values for these outputs.*

Redefining error

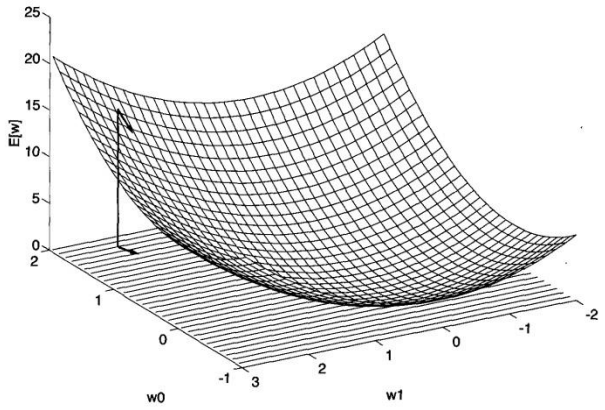
- Because we are considering networks with multiple output units rather than single units as before, we redefine E to sum the errors over all of the network output units:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 \qquad (4.13)$$

- where outputs is the set of output units in the network, and t_{kd} and o_{kd} are the target and output values associated with the *k*th output unit and training example d.

Hypothesis space for Backpropagation

- The learning problem faced by Backpropagation is to search a large hypothesis space defined by all possible weight values for all the units in the network.
- The situation can be visualized in terms of an error surface similar to that shown for linear units in Figure 4.4.
- *The error in that diagram is replaced by our new definition of E, and the other dimensions of the space correspond now to all of the weights associated with all of the units in the network.*



Local minima

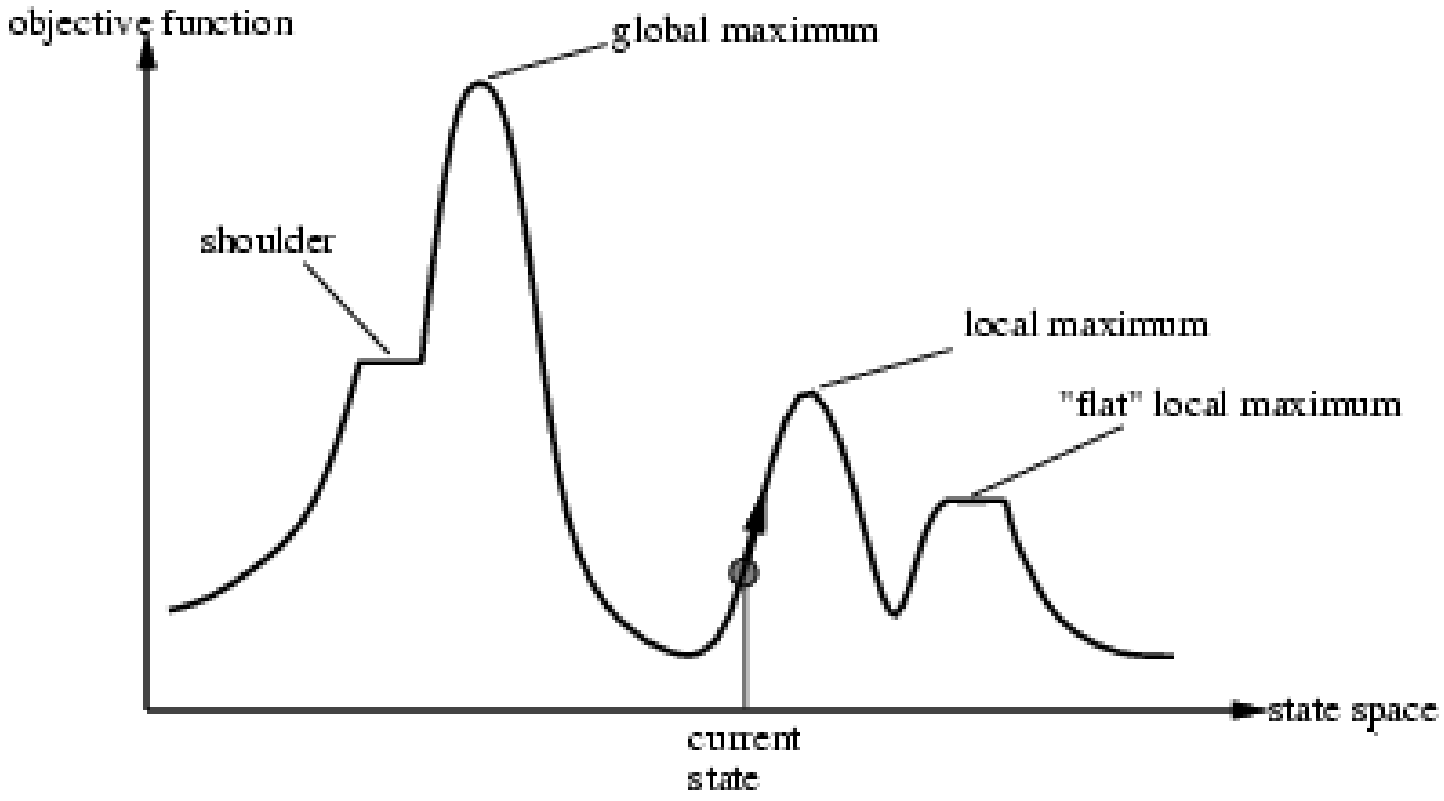
- One major difference in the case of multilayer networks is that the error surface can have multiple local minima, in contrast to the single-minimum parabolic error surface shown in Figure 4.4.

- Unfortunately, this means that gradient descent is *guaranteed only to converge toward some local minimum*, and not necessarily the global minimum error.

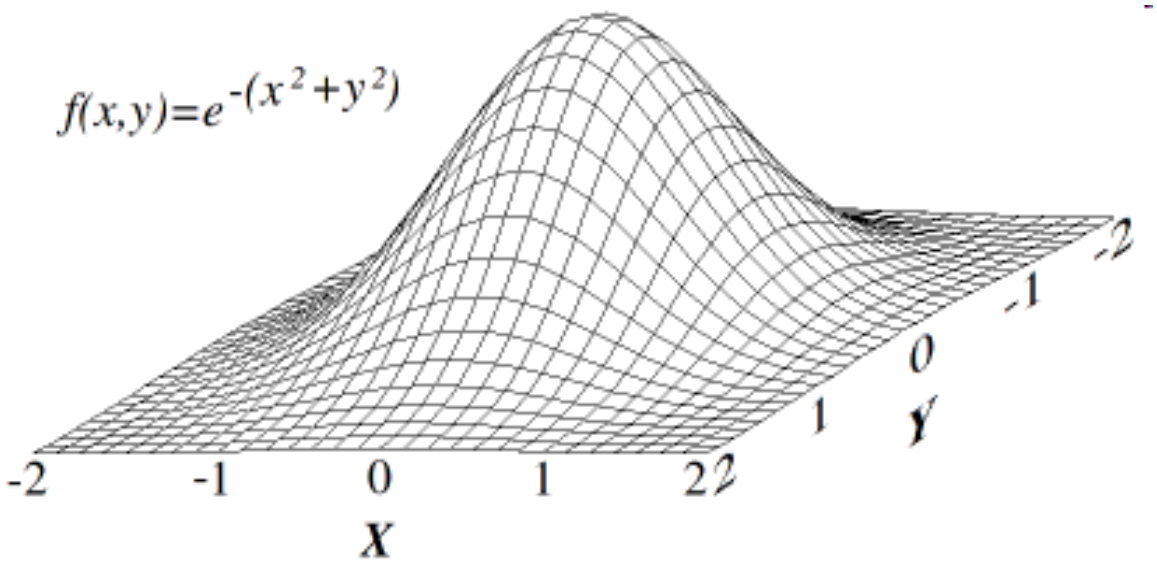
- Despite this obstacle, in practice Backpropagation has been found to produce excellent results in many real-world applications.

Hill-climbing search

- Problem: depending on initial state, can get stuck in local maxima

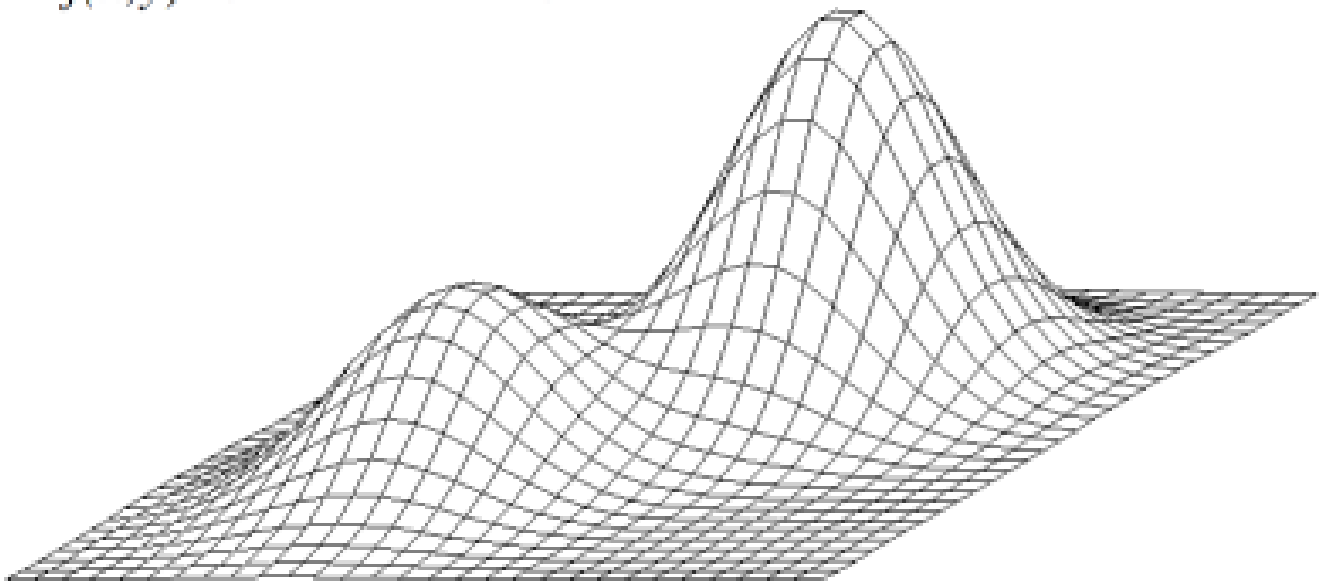


Hill climbing

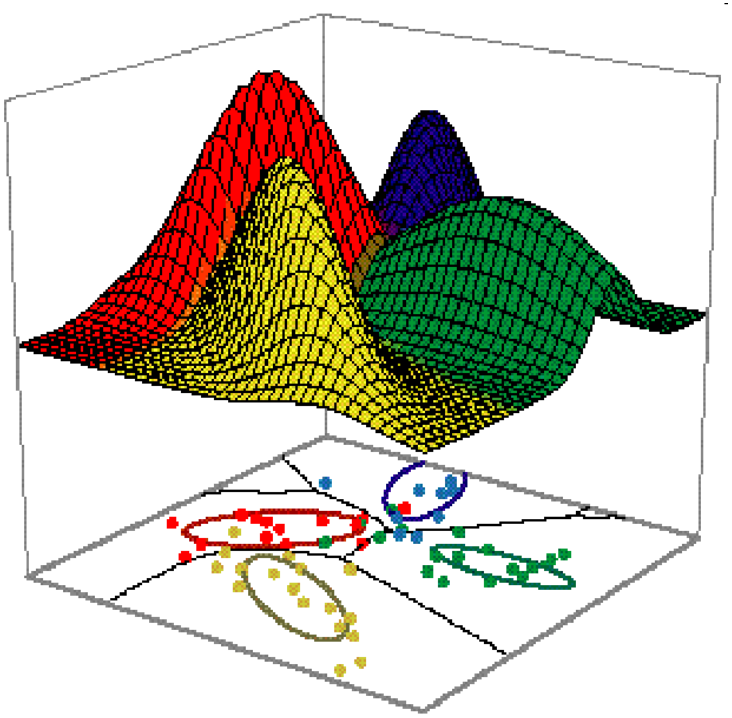


Hill Climbing and Local maxima

$$f(x,y)=e^{-(x^2+y^2)} + 2e^{-((x-1.7)^2+(y-1.7)^2)}$$



Constructing hypothesis and local optima



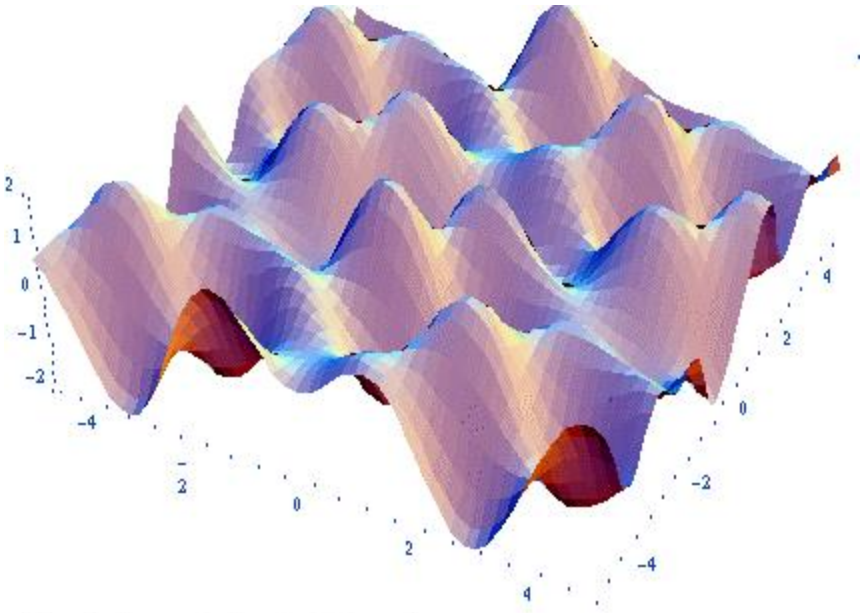


Figure 1. Generated using the Mathematica statement:

Plot3D[-0.2*(Sin[x + 4*y] - 2*Cos[2*x + 3*y] - 3*Sin[2*x - y] + 4*Cos[x - 2*y]), {x, -5, 5}, {y, -5, 5}, PlotPoints->50]

Heuristics for escaping local minima

Common heuristics to attempt to alleviate the problem of local minima include:

1. Add a momentum term to the weight-update rule.
   - Momentum can sometimes carry the gradient descent procedure through narrow local minima (though in principle it can also carry it through narrow global minima into other local minima!).
2. Use *stochastic gradient descent* rather than true gradient descent. The stochastic approximation to gradient descent effectively descends a different error surface for each training example, relying on the average of these to approximate the gradient with respect to the full training set.
   - These different error surfaces typically will have different local minima, making it less likely that the process will get stuck in any one of them.

Heuristics for escaping local minima

3. Train multiple networks using the same data, but initializing each network with different random weights.
   - If the different training efforts lead to different local minima, then the network with the best performance over a separate validation data set can be selected.
   - Alternatively, all networks can be retained and treated as a "committee" of networks whose output is the (possibly weighted) average of the individual network outputs.

Representational Power of Feedforward Networks

What set of functions can be represented by feedforward networks?

1. Boolean functions.
   - Every boolean function can be represented exactly by some network with two layers of units, although the number of hidden units required grows exponentially in the worst case with the number of network inputs.
2. Continuous functions.
   - Every bounded continuous function can be approximated with arbitrarily small error (under a finite norm) by a network with two layers of units (Cybenko 1989; Hornik et al. 1989).

Representational Power of Feedforward Networks

3. Arbitrary functions.
   - Any function can be approximated to arbitrary accuracy by a network with three layers of units (Cybenko 1988).
   - Again, the output layer uses linear units, the two hidden layers use sigmoid units, and the number of units required at each layer is not known in general.

Inductive Bias of Neural Networks

Hypothesis Space Search and Inductive Bias

- It is interesting to compare the hypothesis space search of Backpropagation to the search performed by other learning algorithms.

- For Backpropagation, every possible assignment of network weights represents a syntactically distinct hypothesis that in principle can be considered by the learner.

- In other words, the hypothesis space is the n-dimensional Euclidean space of the n network weights.

Hypothesis Space Search and Inductive Bias

- Notice this hypothesis space is continuous, in contrast to the hypothesis spaces of decision tree learning and other methods based on discrete representations.

- The fact that it is continuous, together with the fact that E is differentiable with respect to the continuous parameters of the hypothesis, results in a well-defined error gradient that provides a very useful *structure for organizing the search for the best hypothesis*.

- This structure is quite different from the general-to-specific ordering used to organize the search for symbolic concept learning algorithms, or the simple-to-complex ordering over decision trees used by the ID3 and C4.5 algorithms.

Inductive bias of Backpropagation

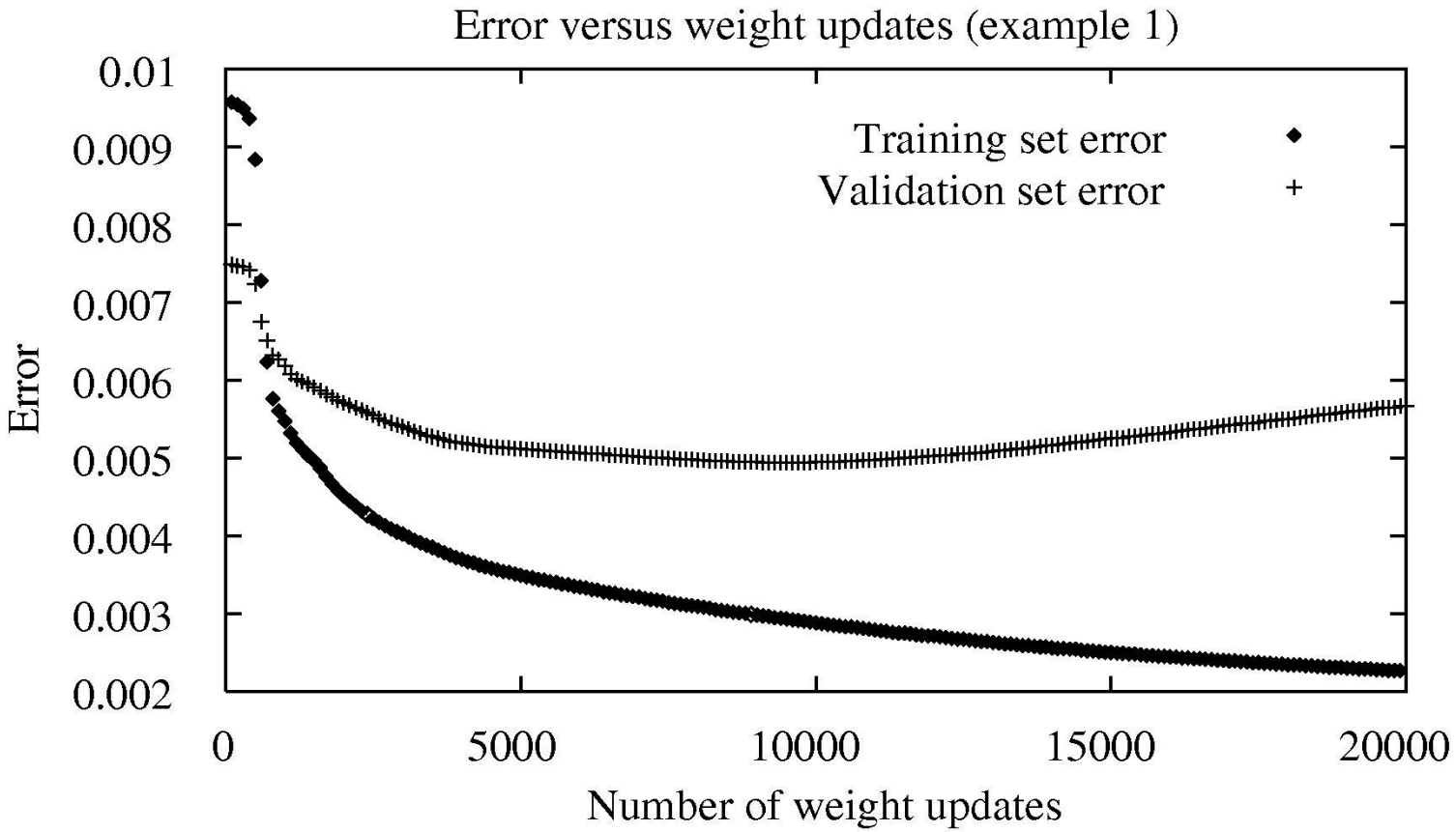
- What is the inductive bias by which Backpropagation generalizes beyond the observed data?
- It is difficult to characterize precisely the inductive bias of Backpropagation learning, because it depends on the interplay between the gradient descent search and the way in which the weight space spans the space of representable functions.
- However, one can roughly characterize it as smooth interpolation between data points:
- *Given two positive training examples with no negative examples between them, Backpropagation will tend to label points in between as positive examples as well.*

Overfitting in Neural Networks

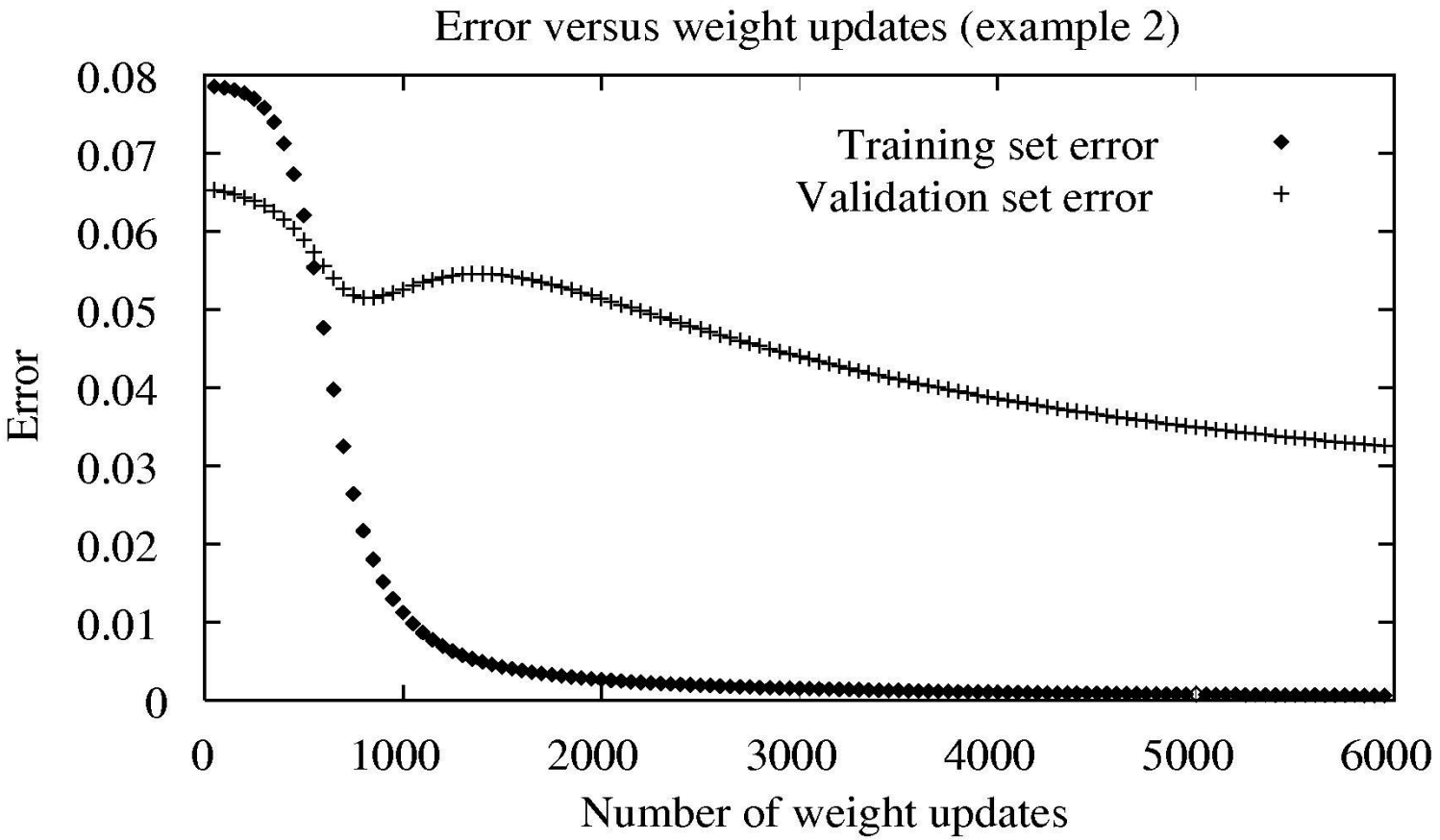
Generalization, Overfitting, and Stopping Criterion

- In the description of the Backpropagation algorithm, the termination condition for the algorithm has been left unspecified.
- What is an appropriate condition for terminating the weight update loop?
- One obvious choice is to continue training until the error E on the training examples falls below some predetermined threshold.
- In fact, this is a poor strategy because Backpropagation is susceptible to overfitting the training examples at the cost of decreasing generalization accuracy over other unseen examples.

Overfitting in Neural Nets



Overfitting in Neural Networks



Overfitting in NNs: Ex.2

- Notice the generalization accuracy measured over the validation examples first decreases, then increases, even as the error over the training examples continues to decrease.
- How can this occur?
  - This occurs because the weights are being tuned to fit peculiarities of the training examples that are not representative of the general distribution of examples.
  - The large number of weight parameters in ANNs provides many degrees of freedom for fitting such peculiarities.

Overfitting in NNs

- Why does overfitting tend to occur during later iterations, but not during earlier iterations?
- The effective complexity of the hypotheses that can be reached by Backpropagation increases with the number of weight-tuning iterations.
- Given enough weight-tuning iterations, Backpropagation will often be able to create overly complex decision surfaces that fit noise in the training data or unrepresentative characteristics of the particular training sample.
- This overfitting problem is analogous to the overfitting problem in decision trees.

Overfitting: Weight decay

- Several techniques are available to address the overfitting problem for Backpropagation learning.
- One approach, known as *weight decay*, is to decrease each weight by some small factor during each iteration.
- This is equivalent to modifying the definition of E to include a penalty term corresponding to the total magnitude of the network weights.
- The motivation for this approach is to keep weight values small, to bias learning against complex decision surfaces.

Overfitting: use of validation set

- One of the most successful methods for overcoming the overfitting problem is to simply provide a set of validation data to the algorithm in addition to the training data.
- The algorithm monitors the error with respect to this validation set, while using the training set to drive the gradient descent search.
- How many weight-tuning iterations should the algorithm perform?
- Clearly, it should use the number of iterations that produces the *lowest error over the validation set*, since this is the best indicator of network performance over unseen examples.

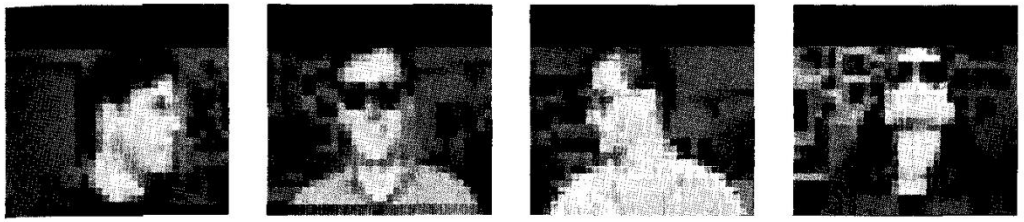
k-fold cross-validation

- In general, the issue of overfitting and how to overcome it is a subtle one.
- The above cross-validation approach works best when extra data are available to provide a validation set.
- Unfortunately, however, the problem of overfitting is most severe for *small training sets*.
- In these cases, a k-fold cross-validation approach is sometimes used, in which cross validation is performed k different times, each time using a different partitioning of the data into training and validation sets, and the results are then averaged.

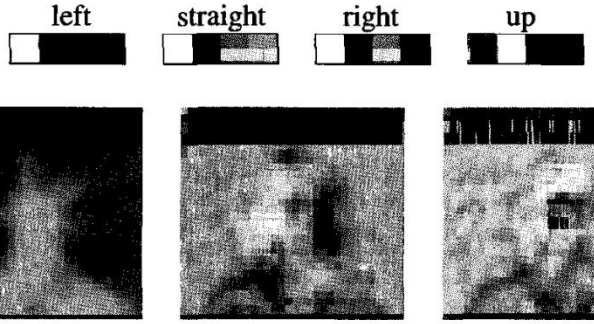
Artificial Neural Networks in Action: Face Recognition

An Illustrative Example: Face Recognition

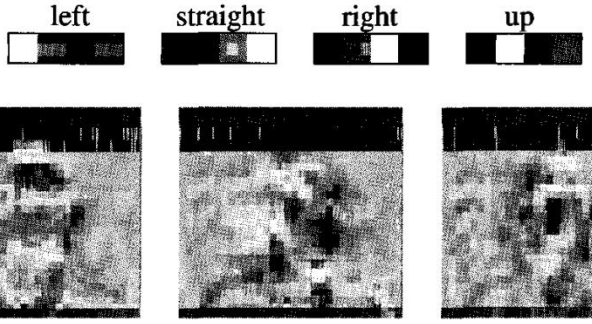
- The learning task here involves classifying camera images of faces of various people in various poses.
- Images of 20 different people were collected, including approximately 32 images per person, varying the person's expression (happy, sad, angry, neutral), the direction in which they were looking (left, right, straight ahead, up), and whether or not they were wearing sunglasses.
- There is also variation in the background behind the person, the clothing worn by the person, and the position of the person's face within the image.
- In total, 624 greyscale images were collected, each with a resolution of 120 x 128, with each image pixel described by a greyscale intensity value between 0 (black) and 255 (white).



30 × 32 resolution input images



Network weights after 1 iteration through each training example



Network weights after 100 iterations through each training example

FIGURE 4.10
Learning an artificial neural network to recognize face pose. Here a 960 × 3 × 4 network is trained on grey-level images of faces (see top), to predict whether a person is looking to their left, right, ahead, or up. After training on 260 such images, the network achieves an accuracy of 90% over a separate test set. The learned network weights are shown after one weight-tuning iteration through the training examples and after 100 iterations. Each output unit (left, straight, right, up) has four weights, shown by dark (negative) and light (positive) blocks. The leftmost block corresponds to the weight w_0, which determines the unit threshold, and the three blocks to the right correspond to weights on inputs from the three hidden units. The weights from the image pixels into each hidden unit are also shown, with each weight plotted in the position of the corresponding image pixel.

Input encoding

- Given that the ANN input is to be some representation of the image, one key design choice is how to encode the image.
  - For example, we could preprocess the image to extract edges, regions of uniform intensity, or other local image features, then input these features to the network.
  - One difficulty with this design option is that it would lead to a variable number of features (e.g., edges) per image, whereas the ANN has a fixed number of input units.
- The design option chosen in this case was instead to encode the image as a fixed set of 30 x 32 pixel intensity values, with one network input per pixel.
- The pixel intensity values ranging from 0 to 255 were linearly scaled to range from 0 to 1 so that network inputs would have values in the same interval as the hidden unit and output unit activations.

Output encoding

- The ANN must output one of four values indicating the direction in which the person is looking (left, right, up, or straight).
- We use four distinct output units, each representing one of the four possible face directions, with the highest-valued output taken as the network prediction.
- This is often called a 1-of-n output encoding.

Other learning algorithm parameters

- In these learning experiments the learning rate η was set to 0.3, and the momentum α was set to 0.3.
- Lower values for both parameters produced roughly equivalent generalization accuracy, but longer training times.
- If these values are set too high, training fails to converge to a network with acceptable error over the training set.
- Full gradient descent was used in all these experiments (in contrast to the stochastic approximation to gradient descent.
- *The number of training iterations* was selected by partitioning the available data into a training set and a separate validation set.
- Gradient descent was used to minimize the error over the training set, and after every 50 gradient descent steps the performance of the network was evaluated over the validation set.
- The final selected network was the one with the highest accuracy over the validation set.

Results

- In the results reported in Figure 4.10, only three hidden units were used, yielding a test set accuracy of 90%.

- In other experiments 30 hidden units were used, yielding a test set accuracy one to two percent higher.

Readings for this part

- Machine Learning. T. Mitchell
  - Chapter 4

Neural Networks in Weka

- Discrete-valued functions
- Continuous-valued functions

- Parameters
  - Learning rate
  - Momentum

Discrete-valued functions

- Iris
  - 0.3 and 0.2 (97.3% of accuracy, better than any previous model, but be careful with overfitting)
  - 0.2 and 0.1 96.667%
  - 0.1 and 0.1 96.667%
  - 0.05 and 0.05 96%
- Diabetes
  - 0.3 and 0.2 75.39%
  - 0.2 and 0.1 75.26%
  - 0.1 and 0.1 74.47%
  - 0.005 and 0.005 76.43%
  - 0.3 and 0.2 and decay = true 77.34%
  - 0.3 and 0.2 and decay = true, training time= 5000, 77.08%

- Use of GUI

Discrete-valued functions

- Hepatitis
  - 0.3 and 0.2 83.22%
  - 0.2 and 0.1 82.58%
  - 0.1 and 0.1 79.35%
  - 0.05 and 0.05 79.35%
  - …

Continuous-valued functions

- Auto price
  - 0.3 and 0.2
    - Correlation coefficient 0.8994
    - Mean absolute error 1746.4859
    - Root mean squared error 2672.8403
    - Relative absolute error 37.7849 %
    - Root relative squared error 45.2067 %
  - 0.2 and 0.1
    - Correlation coefficient 0.9196
    - Mean absolute error 1585.8407
    - Root mean squared error 2301.8609
    - Relative absolute error 34.3094 %
    - Root relative squared error 38.9322 %
  - 0.1 and 0.1
    - Correlation coefficient 0.9167
    - Mean absolute error 1579.5605
    - Root mean squared error 2349.37
    - Relative absolute error 34.1735 %
    - Root relative squared error 39.7358 %

Continuous-valued functions

- Cholesterol
  - 0.3 and 0.2
    - Correlation coefficient 0.031
    - Mean absolute error 71.2998
    - Root mean squared error 98.278
    - Relative absolute error 180.7066 %
    - Root relative squared error 189.5804 %
  - 0.1 and 0.1
    - Correlation coefficient 0.0015
    - Mean absolute error 59.5206
    - Root mean squared error 86.8512
    - Relative absolute error 150.8527 %
    - Root relative squared error 167.5377 %
  - 0.005 and 0.005 (better)
    - Correlation coefficient 0.1334
    - Mean absolute error 40.1213
    - Root mean squared error 53.8912
    - Relative absolute error 101.686 %
    - Root relative squared error 103.9572 %

Continuous-valued functions

- 0.0001 and 0.0001 and training time = 5000
  - Correlation coefficient 0.2058
  - Mean absolute error 38.0398
  - Root mean squared error 50.6644
  - Relative absolute error 96.4103 %
  - Root relative squared error 97.7326 %

End of class