Multilayer Neural Networks

- Introduction
- Feedforward Operation
- Backpropagation Algorithm
- Practical Techniques for Improving Backpropagation

Introduction

- So far, we have considered single layer networks for pattern classification and function approximation.

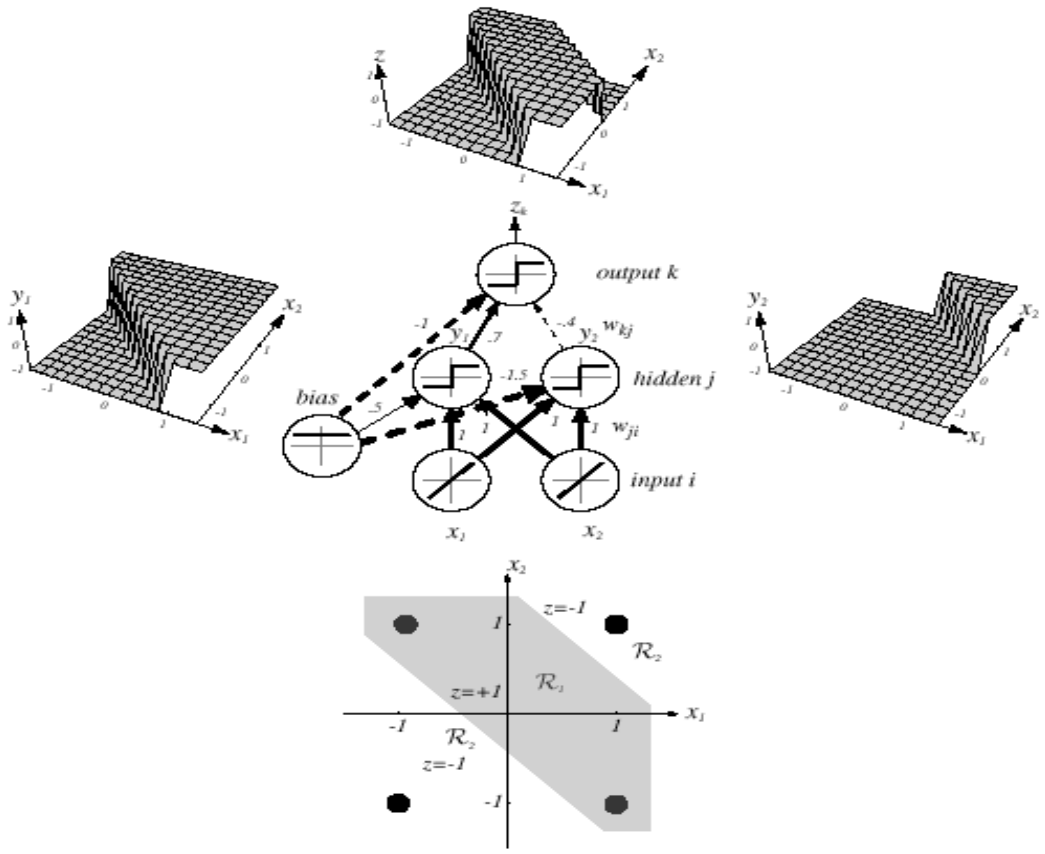
$$y(\mathbf{x}, \mathbf{w}) = f(\sum_j w_j x_j)$$

- We have seen that the computational and function approximation abilities of such networks are fairly limited.
  - For instance, threshold neurons and winner-take-all groups can be trained to classify only linearly separable pattern sets.
  - Linear neurons can only find a linear fit to data on function approximation tasks.
- Extending Linear Classifiers: linear combinations of fixed nonlinear *basis functions*

$$y(\mathbf{x}, \mathbf{w}) = f(\sum_j w_j \phi_j(x))$$

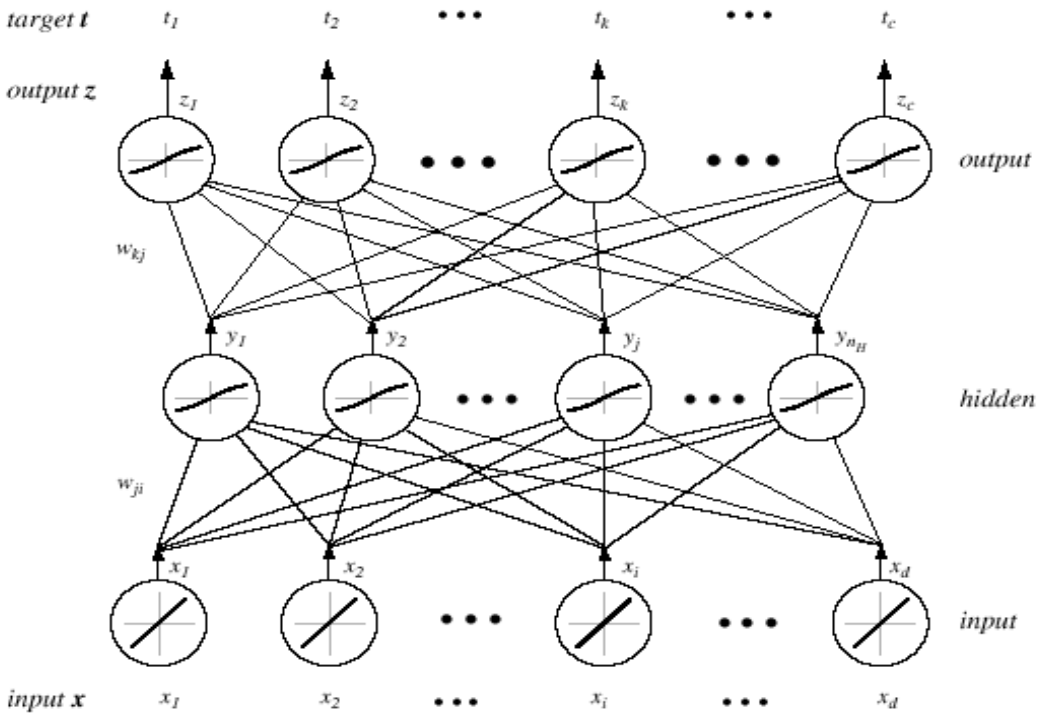
- Allow basis functions themselves to adapt to the data: each basis function is itself a nonlinear function of a linear combination of the inputs → multilayer neurons

- We know that multi-layer networks of threshold neurons can realize arbitrary boolean functions.
- Consider a simple three-layer neural network that solves the exclusive-OR (XOR) problem, shown in the next slide
  - The hidden unit y_1 computes the boundary: $x_1 + x_2 + 0.5 = 0$
    $y_1 = (x_1 \text{ OR } x_2)$
  - The hidden unit y_2 computes the boundary: $x_1 + x_2 - 1.5 = 0$
    $y_2 = x_1 \text{ AND } x_2$
  - The final output unit emits
    $z_1 = y_1$ AND NOT $y_2 = x_1 XOR x_2$
    which provides the nonlinear decision



Feedforward Network

- A three-layer neural network consists of an input layer, a hidden layer and an output layer interconnected by modifiable weights represented by links between layers
- “neuron” or “unit” are used interchangeably
- Multiple layers of cascaded linear neurons still produce linear functions
- The novel computational power provided by multilayer neural nets can be attributed to the nonlinear mapping of the input to the representation at the hidden units, since the hidden-to-output layer leads to a linear discriminant



- A “bias unit” is connected to each unit other than the input units
- Each hidden unit computes its *net activation*:

$$net_j = \sum_{i=1}^{d} x_i w_{ji} + w_{j0} = \sum_{i=0}^{d} x_i w_{ji} \equiv w_j^t x,$$

where the subscript i indexes units in the input layer, *j* in the hidden; w_{ji} denotes the input-to-hidden layer weights at the hidden unit *j*. (In neurobiology, such weights or connections are called “synapses” and the values of the connections the “synaptic weights”)

- Each hidden unit emits an output that is a nonlinear function of its activation, that is: $y_j = f(net_j)$

Threshold neurons use a simple threshold function

$$f(net) = sgn(net) \equiv \begin{cases} 1 \ \ if \ \ net \geq 0 \\ -1 \ \ if \ \ net < 0 \end{cases}$$

- The function *f(.)* is also called the ***activation function*** or “nonlinearity” of a unit. There are more general activation functions with desirable properties such as logistic sigmoid or *tanh*
- Each output unit similarly computes its net activation based on the hidden unit signals as:

$$net_k = \sum_{j=1}^{n_H} y_j w_{kj} + w_{k0} = \sum_{j=0}^{n_H} y_j w_{kj} = w_k^t y,$$

where the subscript *k* indexes units in the ouput layer and n_H denotes the number of hidden units

- An output unit computes the nonlinear function of its net, emitting

$$z_k = h(net_k)$$

e.g., regression h(x) =x, classification h(x) sigmoidal

- The output z_k can be thought of as a function of the input vector ***x:*** $z_k = g_k(x)$

- To use the network as a multi-category pattern classifier, in the case of *c* outputs (classes), we can view the network as computing *c* discriminant functions $z_k = g_k(x)$ and classify the input x according to the largest discriminant function $g_k(x)$ $\forall k = 1, \ldots, c$ (winner-take-all approach)

- General Feedforward Operation – case of three-layer network with c output units

$$g_k(x) \equiv z_k = h\left(\sum_{j=1}^{n_H} w_{kj} f\left(\sum_{i=1}^{d} w_{ji} x_i + w_{j0}\right) + w_{k0}\right) \qquad (1)$$

$$(\mathrm{k} = 1,\ldots,\mathrm{c})$$

  - Hidden units enable us to express more complicated nonlinear functions and thus more complicated decision boundaries

  - The activation function does not have to be a sign function, it is often required to be continuous and differentiable

  - We can allow the activation in the output layer to be different from the activation function in the hidden layer, or have different activation for each individual unit

- Network Structures
  - Layered networks: having successive layers of units, with connections running from every unit in one layer to every unit in the next layer.
  - More than one hidden layer; skip-layer connections
  - Restricted to *feed-forward* networks: no feed-back loops in the network
  - Recurrent networks: have internal state
    - Hopfield networks

Expressive Power of multi-layer Networks

- Every boolean function can be represented by a three-layer network, but might require exponential (in number of inputs) hidden units
- Universal Function Approximation Theorem

  "Any bounded *continuous* function from input to output can be approximated with arbitrarily small error by a three-layer network, given sufficient number of hidden units n_H, proper nonlinearities, and weights."
- Any decision boundaries can be implemented as a three-layer neural network
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers.

- The results on the expressive power of networks are of greater theoretical interest than practical, since they do not give an algorithm for finding suitable values for number of hidden units or weights.
- Nor do they imply that a 3-layer network is optimal for approximating a function.

Learning Neural Networks

Given a training set determine

- Network structure – number of hidden units or more generally, network topology
- For a given structure, determine weights
- For now, we focus on the latter

- Network Learning
  - Let t_k be the k-th target (or desired) output and $z_k(x,w)$ be the k-th computed output with *k = 1, …, c* and ***w*** represents all the weights of the network
  - Assume the distribution

$$p(\mathbf{t}|\mathbf{x},\mathbf{w}) = N(\mathbf{t}|\mathbf{z}(\mathbf{x},\mathbf{w}),\sigma^2\mathbf{I})$$

  - Maximization of the likelihood function with respect to **w** is equivalent to minimizing the sum-of-squares error function

$$J(w) = \frac{1}{2}\sum_{i=1}^{n}\sum_{k=1}^{c}(t_{ik} - z_{ik})^2$$

- Function optimization
  - Batch gradient descent

$$w(t+1) = w(t) + \Delta w(t)$$
$$\Delta w = -\eta \nabla J$$

  - Stochastic gradient descent (with momentum)

$$J = \sum_{i=1}^{n} J_i, \quad \Delta w = -\eta \nabla J_i$$

  - More efficient methods: conjugate gradient, quasi-Newton
    - All use gradient information

- Compute gradient

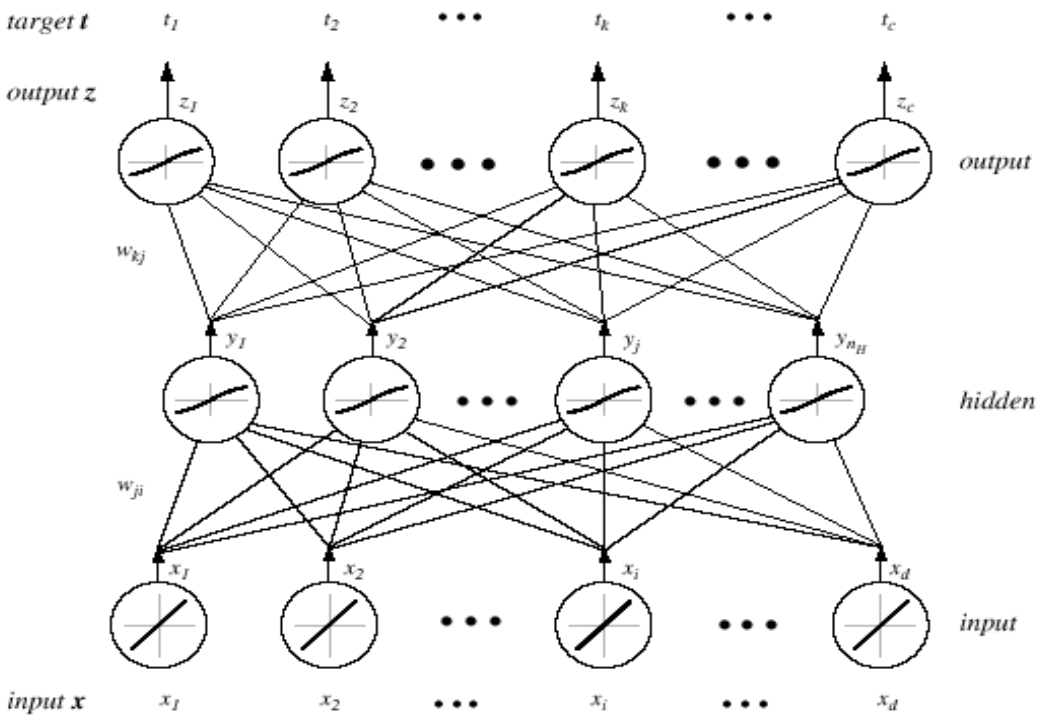
- We first consider the per-pattern learning rule. The per-pattern training error:

$$J(w) = \frac{1}{2}\sum_{k=1}^{c}(t_k - z_k)^2$$

Dropped index i for convenience

$$net_j = \sum_{i=0}^{d} x_i w_{ji}, \quad y_j = f(net_j)$$

$$net_k = \sum_{j=0}^{n_H} y_j w_{kj} = w_k^t y, \quad z_k = h(net_k)$$



Compute gradient

- hidden–to-output weights

$$\frac{\partial J}{\partial w_{kj}} = \frac{\partial J}{\partial net_k} \cdot \frac{\partial net_k}{\partial w_{kj}} = -\delta_k \frac{\partial net_k}{\partial w_{kj}}$$

where the *sensitivity (error) of output unit k* is defined as:

$$\delta_k = -\frac{\partial J}{\partial net_k}$$

$$\frac{\partial net_k}{\partial w_{kj}} = y_j$$

$$\frac{\partial J}{\partial w_{kj}} = -\delta_k y_j$$

$$\delta_k = -\frac{\partial J}{\partial net_k} = -\frac{\partial J}{\partial z_k} \cdot \frac{\partial z_k}{\partial net_k} = (t_k - z_k) h'(net_k)$$

- Linear neuron: $h'(net_k) = 1$
- Logistic sigmoid neuron: $h'(net_k) = z_k (1 - z_k)$

Conclusion:

$$\frac{\partial J}{\partial w_{kj}} = -\delta_k y_j$$

- input-to-hidden units

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial net_j} \cdot \frac{\partial net_j}{\partial w_{ji}} = -\delta_j x_i$$

Similarly as in the preceding case, the *sensitivity (error) for a hidden unit j* is defined as:

$$\delta_j = -\frac{\partial J}{\partial net_j} = -\sum_{k=1}^{c} \frac{\partial J}{\partial net_k} \cdot \frac{\partial net_k}{\partial net_j}$$

$$= +\sum_{k=1}^{c} \delta_k \cdot \frac{\partial net_k}{\partial y_j} \cdot \frac{\partial y_j}{\partial net_j} = \sum_{k=1}^{c} \delta_k \cdot w_{kj} \cdot f'(net_j)$$

$$= f'(net_j) \sum_{k=1}^{c} w_{kj} \delta_k$$

"The sensitivity (error) at a hidden unit is simply the sum of the individual sensitivities at the output units weighted by the hidden-to-output weights w_{kj}, all multipled by *f'(net$_j$)*"

- "Backpropagation" algorithm is so-called because during training an error must be propagated from the output layer back to the hidden layer in order to perform the learning of the input-to-hidden weights

- Backpropagation algorithm provide an efficient method for computing the gradient of error functions

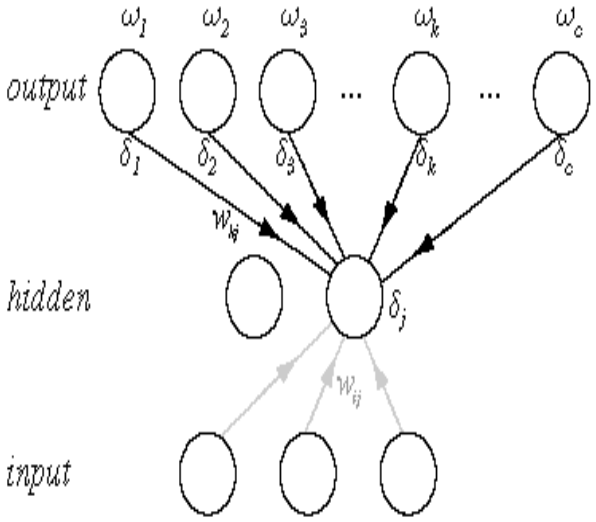


FIGURE 6.5. The sensitivity at a hidden unit is proportional to the weighted sum of the sensitivities at the output units: $\delta_j = f'(net_j) \sum_{k=1}^{c} w_{kj}\delta_k$. The output unit sensitivities are thus propagated "back" to the hidden units. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

Efficiency of back-propagation

One of the most important aspects of back-propagation is its computational efficiency. Let *W* be the total number of weights

- A single evaluation of the error function for a given input pattern would require O(W) operations (The number of weights is typically much larger than the number of units)
- There are *W* derivatives to evaluate.
- Back-propagation allows the *W* derivatives to be evaluated in O(W) operations.

The backpropagation algorithm (also known as the generalized delta rule) is summarized below:

- Select a network architecture.
- Initialize the weights to small random values.
- Present the network with training examples from training set in some order. It helps to randomize the order of presentation of examples in each pass of the training set. For each training example
  - Forward pass: compute the net activations and outputs of each of the neurons in the network, e.g. y_j, z_k
  - Backward pass: compute the errors for each of the neurons in the network, e.g., δ_k, δ_j
  - Update weights
  - If the stopping criterion is satisfied, then stop

- Although our analysis was done for a three-layer network, the backpropagation algorithm can be generalized directly to feed-forward networks in which
  - Skip-layer connections: input units are connected directly to output units as well as hidden units
  - More than three layers of units
  - Different nonlinearities for different layers
  - Different error functions

- The batch backpropagation algorithm
  - So far, we have considered the error on a single pattern.
  - The total training error is the sum over the errors of n individual patterns

$$J = \sum_{i=1}^{n} J_i$$

  - In the batch training, all the training patterns are presented first and their corresponding weight updates summed; only then are the actual weights in the network updated
- Epoch: one epoch corresponds to a single presentation of all patterns in the training set
- We describe the overall amount of pattern presentations by epoch. The number of epochs is an indication of the relative amount of learning.

Choosing network architecture

- Whereas the number of inputs and outputs of a three-layer network are determined by the problem itself, we do not know ahead of time the number of hidden units.
  - The number of hidden units governs the expressive power of the network --- and thus the complexity of the decision boundary
  - If the network has too few hidden units, it does not have enough free parameters to fit the training data well; if too many, it tend to overfit the data
  - A convenient rule of thumb is to choose the number of hidden units such that the total number of weights in the net is roughly n/10?
  - Try different number and use validation set to choose the best one?
  - A more principled method is to adjust the complexity of the network in response to the training data---a few techniques available

- Weight Decay
  - A technique for simplifying network and avoiding over-fitting
  - Start with a network with “too many” weights and “decay” all weights during training
  - After each weight update, every weight is decayed according to

$$w \leftarrow w(1-\varepsilon), \quad 0 < \varepsilon < 1$$

  - Weights that are not needed for reducing the error function become smaller and smaller, possibly to such a small value that they can be eliminated altogether
  - Those weights that *are* needed to solve the problem will not decay indefinitely.
  - It is found in most cases that weight decay helps

- Regularization
  - Look for simpler networks that adequately fit the training data.
  - One general approach of regularization is to make a new criterion function that depends not only on the training error but also on classifier complexity

$$J(w) = J_{err}(w) + \lambda J_{reg}(w)$$

add a term that penalizes network complexity; λ is a parameter that trades off the error against network complexity

  - For example, gradient descent in the following criterion function is equivalent to weight decay

$$J(w) = J_{err}(w) + \frac{\varepsilon}{2\eta}\mathbf{w}^t\mathbf{w}$$

Early Stopping

- One obvious choice for terminating the weight update loop is to continue training until the error *J(w)* (or the change in *J(w)*) falls below some predetermined threshold
- To see the dangers of minimizing the error over the training data, consider how the error varies with the number of weight iterations --*learning curves*
  - Because batch back-propagation performs gradient descent in the error function, if the learning rate is not too high the training error tends to decrease monotonically
  - Typically, the error on the validation set decreases, but then increases, an indication that the classifier may be overfitting the training data
- Since our ultimate goal is low generalization error, we stop training at a minimum of the error on the validation set, since this is the best indicator of classifier performance over unseen examples
- One must be careful to not stop training too soon when the validation set error begins to increase but has not reached its lowest

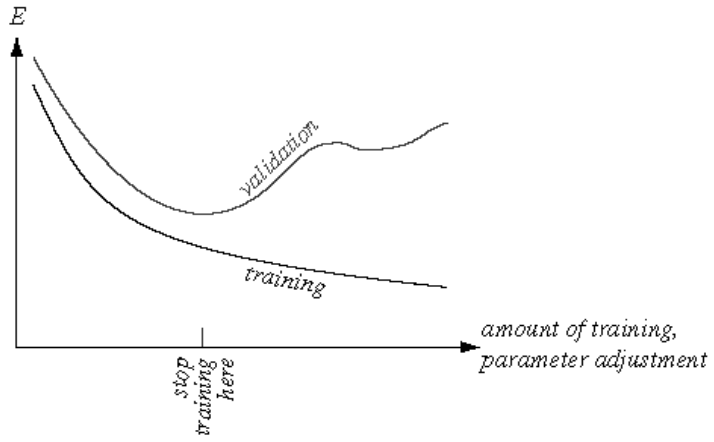


FIGURE 9.9. In validation, the data set $\mathcal{D}$ is split into two parts. The first (e.g., 90% of the patterns) is used as a standard training set for setting free parameters in the classifier model; the other (e.g., 10%) is the validation set and is meant to represent the full generalization task. For most problems, the training error decreases monotonically during training, as shown in black. Typically, the error on the validation set decreases, but then increases, an indication that the classifier may be overfitting the training data. In validation, training or parameter adjustment is stopped at the first minimum of the validation error. In the more general method of cross-validation, the performance is based on multiple independently formed validation sets. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

Cross-validation in training neural networks

- Hold-out method:
  - Evaluate the performance of the network over the validation set every a number of epochs
  - Keep a separate copy of the best-performing weights thus far.
  - Once the trained weights reach a significantly higher error over the validation set than the stored weights, training is terminated and the stored weights are returned as final hypothesis
- A *test set* is used to state/report the performance of the trained network

- *m*-fold cross-validation:
  - The training set is randomly divided into *m* disjoint sets of equal size *n/m*
  - The training procedure is run *m* times, each time with a different set held out as a validation set.
  - On each run the number of epochs is determined that yield the best performance on the validation set.
  - The mean *k* of these estimates is calculated, and a final run of backpropagation algorithm is performed *training on all n examples* for k epochs.

Practical Techniques for Improving Backpropagation

- Activation function, desired properties
  - Nonlinear: otherwise the three-layer network provides no computational power above that of a two-layer net
  - Continuous and differentiable
  - Saturate: have maximum and minimum output value --- This will keep the weights and activations bounded and thus keep training time limited.
  - Monotonic: if not monotonic and has multiple local maxima, additional local extrema in the error surface may become introduced.
  - Linear for a small value of net activation --- enable the system to implement a linear model if adequate

Practical Techniques

- The class of *sigmoid functions* is the most widely used activation function which has all the desired properties
- The logistic sigmoid

$$f(net) = \frac{1}{1+e^{-net}}$$

$$f'(net) = f(net)(1-f(net))$$

- It is best to keep the function centered on zero and anti-symmetric or as an odd function, that is, *f(-net)= -f(net)* --- lead to faster learning together with the data preprocessing

Practical Techniques

- For faster learning, use sigmoid functions of the form of a hyperbolic tangent

$$f(net) = a\tanh(bnet) = a\left[\frac{e^{+bnet} - e^{-bnet}}{e^{+bnet} + e^{-bnet}}\right]$$

- a=1.716, b=1/3 → *f(net)* is nearly linear in the range -1<net<+1.

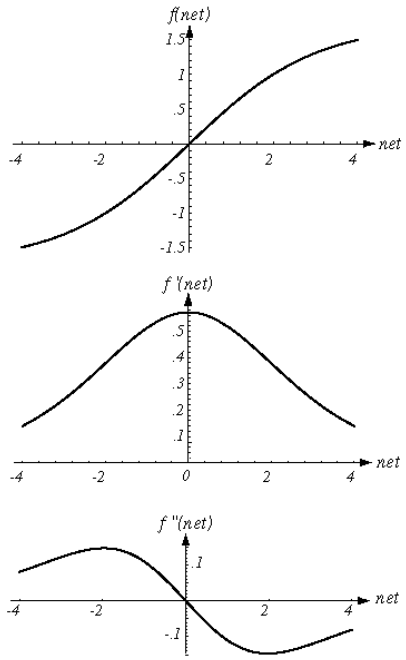


FIGURE 6.14. A useful activation function $f(net)$ is an anti-symmetric sigmoid. For the parameters given in the text, $f(net)$ is nearly linear in the range $-1 < net < +1$ and its second derivative, $f''(net)$, has extrema near $net \simeq \pm 2$. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

Practical Techniques

- Scaling input
  - The values of one feature may be orders of magnitude larger than that of another
  - E.g., classify fish based on the features of mass (measured in grams) and length (measured in meters)
  - During the training, the network will adjust weights from the "mass" input unit far more than for the "length" unit---indeed the error will hardly depend upon the tiny length values
  - If the same information were presented but with mass measured in kilograms and length in millimeters …
  - We do not want our classifier to prefer one feature over the other because they differ solely in the arbitrary representation

Practical Techniques

- Scaling input
  - In order to avoid such difficulties, the input patterns should be preprocessed so that the average over the training set of each feature is zero, and the data set is scaled to have the same variance (chosen to be 1.0) in each feature component
  - *Standardize* the training patterns

$$\hat{\mu}_i = \frac{1}{n}\sum_{k=1}^{k=n} x_i^k \quad ; \quad \hat{\sigma}_i^{\,2} = \frac{\sum_{k=1}^{k=n}(x_i^k - \hat{\mu}_i)^2}{n-1}$$

$$x_i^k \leftarrow \frac{x_i^k - \hat{\mu}_i}{\hat{\sigma}_i}$$

Practical Techniques

- Initializing weights
  - We seek to set the initial weights in order to have fast and uniform learning -- all weights reach their final equilibrium values at about the same time
  - Because data standardization gives positive and negative values equally, on average, we want positive and negative weights as well
  - Initialize weights to small random values according to a uniform distribution $-w<w_{ji}<+w$, that place the neurons in the linear range $-1<net<+1$
  - The net activation from d-dimentional standardized input is distributed roughly between $-w\sqrt{d} < net_j < +w\sqrt{d}$
  - We would like this net activation to be roughly in the range $-1<net<+1$

Practical Techniques

- Initializing weights
  - Initialize input-to-hidden weights to random values according to a uniform distribution in the range

$$-\frac{1}{\sqrt{d}} < w_{ji} < +\frac{1}{\sqrt{d}}$$

  - Initialize hidden-to-output weights to random values according to a uniform distribution in the range

$$-\frac{1}{\sqrt{n_H}} < w_{kj} < +\frac{1}{\sqrt{n_H}}$$

Practical Techniques

- Learning with momentum
  - The modified gradient descent update rule

$$\Delta w(m) = -\eta \frac{\partial J}{\partial w} + \alpha \Delta w(m-1) \quad 0 \le \alpha < 1$$

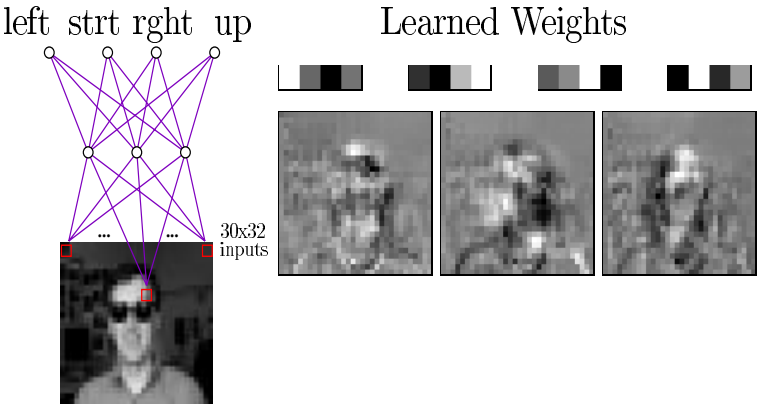
  - Speed up convergence
  - May overcome local minima or plateaus

Practical Techniques

- Stochastic or batch learning?
  - For most applications – especially ones employing large redundant training sets– stochastic gradient learning is typically faster than batch learning
- More than one hidden layer?
  - Unless under special problem requirements (see convolutional networks), since one hidden layer suffices to implement any continuous function
  - Empirically, more prone to getting caught in undesirable local minima
  - The backpropagation algorithm is easily generalized

An Application Example: Face Recognition

- From Tom Mitchell “Machine Learning”
- Data:
  - camera images of faces of various people in various poses
  - 20 people, about 32 images each, varying expressions (happy, sad, …), looking directions (left, right, …), background etc.
  - Collected 624 greyscale images, resolution 120x128, with each image pixel described by a greyscale intensity value between 0 (black) and 255 (white)
- Task: learning the direction in which the person is facing (left, right, straight ahead, upward)



Typical input images

Design Choices

- Input encoding
  - Encode the image as 30x32 pixel intensity values, calculated as the mean of the corresponding high-resolution values, with one network input per pixel
  - Using this coarse-resolution image reduces the number of inputs and weights to a much more manageable size, thereby reducing computational demands, while maintaining sufficient resolution to correctly classify the images
  - The pixel intensity values 0 to 255 were linearly scaled to range from 0 to 1 (match the logistic sigmoid output values)

Design Choices

- Network structure
  - One hidden layer, three hidden units (experimented with 30 hidden units, yielding a test set accuracy one to two percent higher)
  - Logistic sigmoid units for hidden and output layer
- Output encoding
  - 1-of-c encoding: use four output units, each representing one of the four possible face directions, with the highest-valued output taken as the network prediction
  - Target values: e.g., (0.9, 0.1, 0.1, 0.1), avoid (1, 0, 0, 0)

- Learning algorithm parameters
  - Batch gradient descent algorithm
  - Learning rate η 0.3, momentum α 0.3
  - Initial weights: input-to-hidden 0 (because this yields more intelligible visualizations of the learned weights), hidden-to-output small random values
  - Stopping criterion: validation method, after every 50 epochs 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: after training on 260 images, achieves an accuracy of 90% over a separate test set

Error Functions for Classification

Sum-of-squares error function may not be the most appropriate for classification task

Two-category classification problem

- Use a single output unit
- Arrange the output of the network to correspond to the posterior probability

$$P(\omega_1|\vec{x}) = z(\vec{x};w) = \sigma(net)$$

- That is, the output unit should use the logistic sigmoid activation function

Posterior probability

The posterior probability for two-class problem

$$p(\omega_1|\vec{x}) = \frac{p(\vec{x}|\omega_1)P(\omega_1)}{p(\vec{x}|\omega_1)P(\omega_1)+p(\vec{x}|\omega_2)P(\omega_2)}$$
$$= \frac{1}{1+\exp(-g(\vec{x}))} = \sigma(g(\vec{x}))$$

where

$$g(\vec{x}) = \ln \frac{p(x|\omega_1)P(\omega_1)}{p(x|\omega_2)P(\omega_2)}$$

Use a neural network to represent the discriminant function

Error Functions for Classification

- Maximizing conditional likelihood leads to cross-entropy error function

$$J(w) = -\sum_{p=1}^{n} [t_p \ln z_p + (1-t_p)\ln(1-z_p)]$$

  where t_p and z_p are the target and output for input pattern $\vec{x}_p$
- Target coding: $t_p = 1$ if $x_p \in \omega_1$ and $t_p = 0$ if $x_p \in \omega_2$
- Computing gradient ...

$$\frac{\partial J}{\partial net_p} = z_p - t_p$$

Multiclass Problem

The posterior probability for multiple-category classification problem

$$p(\omega_k|\vec{x}) = \frac{p(\vec{x}|\omega_k)P(\omega_k)}{\sum_j p(\vec{x}|\omega_j)P(\omega_j)} = \frac{\exp(g_k(\vec{x}))}{\sum_j \exp(g_j(\vec{x}))}$$

known as the *softmax function*, where

$$g_j(\vec{x}) = \ln(p(\vec{x}|\omega_j)P(\omega_j))$$

Use a neural network to represent the discriminant functions $g_j(\vec{x})$

Multiclass Problem

Multiple-category classification problem

- 1-of-c encoding
- The outputs of the network correspond to the posterior probabilities

$$P(\omega_k|\vec{x}) = z_k(\vec{x}; w) = \frac{\exp(net_k)}{\sum_{m=1}^{c} \exp(net_m)}$$

- The output units should use the softmax activation function

Error Functions for Classification

- Cross-entropy error function

$$J(w) = -\sum_{i=1}^{n}\sum_{k=1}^{c} t_{ik} \ln z_{ik}$$

  where t_{ik} and z_{ik} are the target and output of the output unit k for input pattern $\vec{x}_i$

- Target coding: $t_{ik} = 1$ if $x_i \in \omega_k$ and $t_{ik} = 0$ otherwise
- Computing gradient ...

$$\frac{\partial z_k}{\partial net_j} = z_k(\delta_{kj} - z_j), \qquad \frac{\partial J}{\partial net_{ik}} = z_{ik} - t_{ik}$$