SEP760 Cyber Physical Systems

Summer 2025

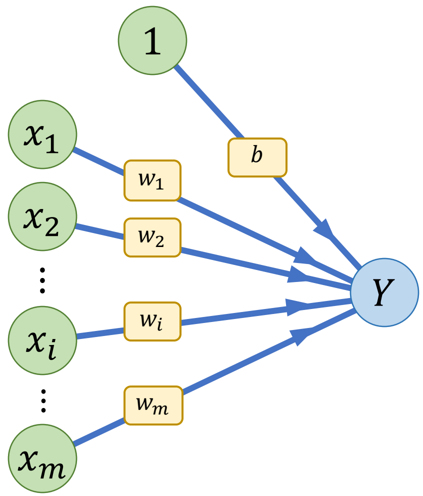
Deep / Machine Learning

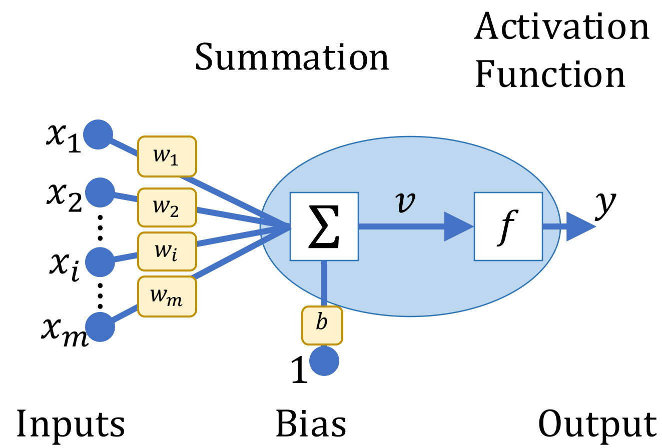
W. Booth School of Engineering, McMaster University

Dr. Anwar Mirza

mirzaa24@mcmaster.ca

2. Feedforward Neural Networks (FNN) and Backpropagation





$$\boldsymbol{x}^T = (x_1, x_2, x_3, \cdots, x_i, \cdots, x_m), \qquad \boldsymbol{w}^T = (w_1, w_2, w_3, \cdots, w_i, \cdots, w_m)$$

v is the weighted sum of inputs:

$$v = b + w_1x_1 + w_2x_2 + \cdots + w_ix_i + \cdots + w_mx_m = b + \sum_{i=1}^{m} w_ix_i = b + w^T\boldsymbol{x}$$

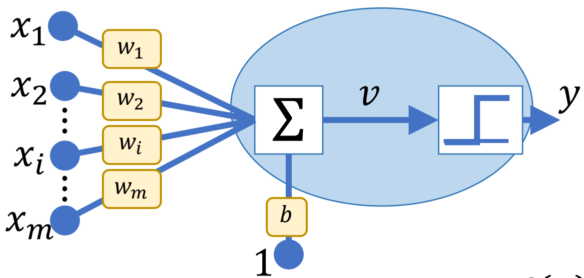
y is the output obtained by applying the activation function f on the weighted sum of inputs v :

$$y = f(v) = f(b + w_1x_1 + w_2x_2 + \cdots + w_ix_i + \cdots + w_mx_m) = f\left(b + \sum_{i=1}^{m} w_ix_i\right)$$

Feedforward Neural Networks

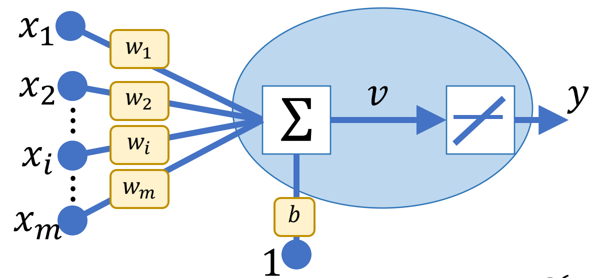
One Output Unit

Perceptron vs. Adaline



Inputs Bias

$$y = f(v) = threshold(v)$$



Inputs Bias

$$y = f(v) = linear(v) = v$$

Error for nth pattern: $e(n) = d(n) - y(n)$

Learning Rules

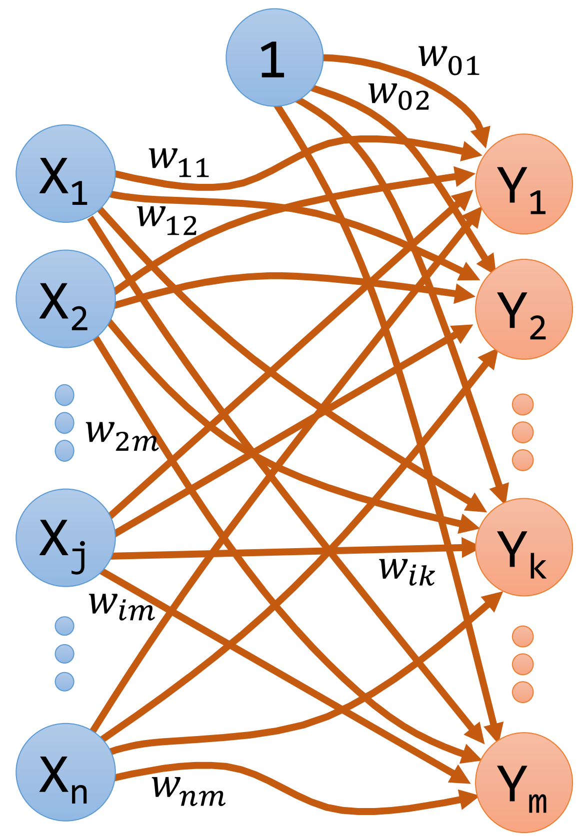
Perceptron Learning Rule

$$w_i(n+1) = w_i(n) + \eta\, d(n)\, x_i(n)$$
$$b(n+1) = b(n) + \eta\, d(n)$$

Adaline / Delta Rule / LMS Learning

$$w_i(n+1) = w_i(n) + \eta\, e(n)\, x_i(n)$$
$$b(n+1) = b(n) + \eta\, e(n)$$

Single-Layer Feedforward Neural Network



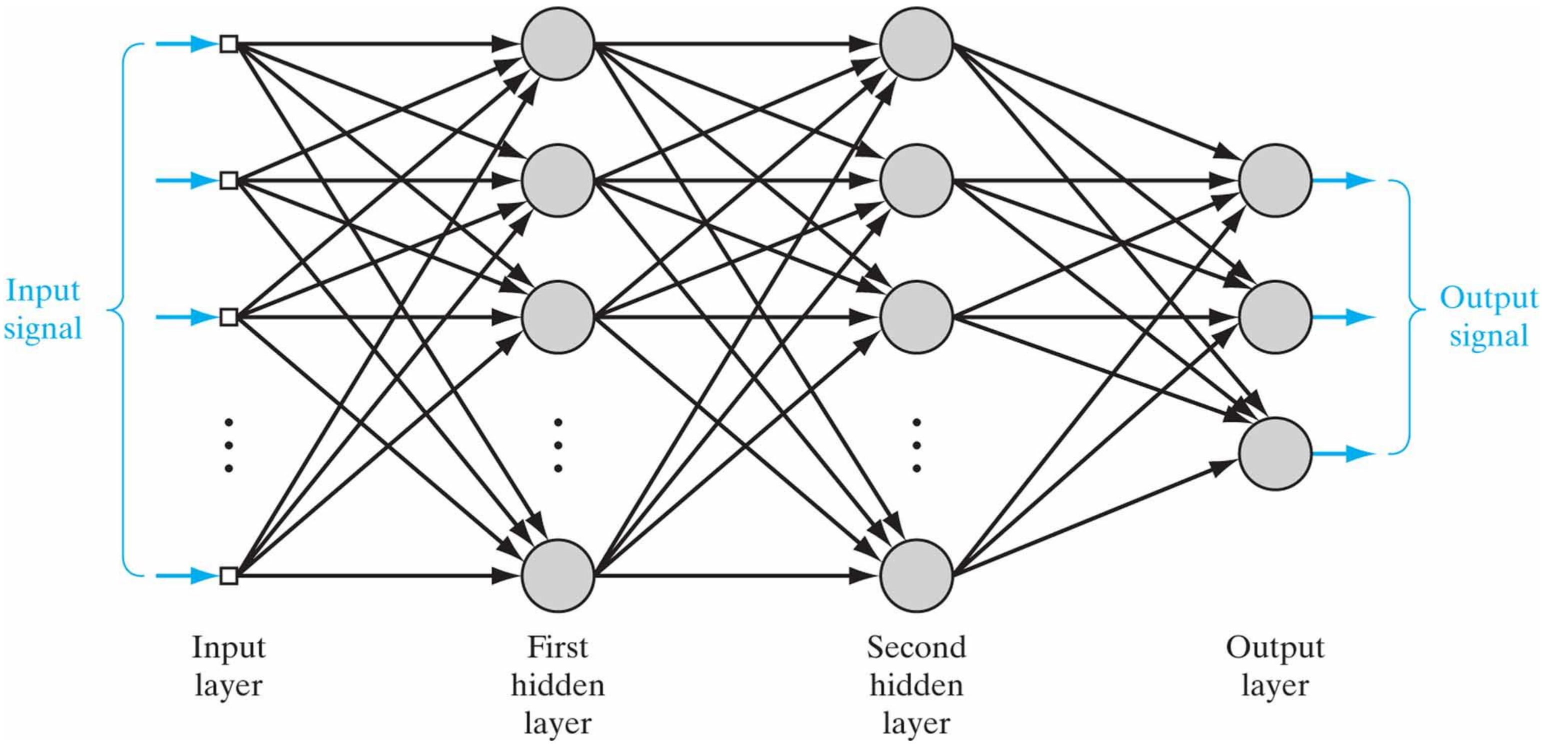
Multi-class Problem

i	s_1	s_2	…	s_n	t_1	t_2	…	t_m
1	0.1	0.14	…	0.71	0.9	0.1	…	0.1
2	0.3	0.2	…	0.34	0.1	0.9	…	0.1
3	0.2	0.9	…	0.62	0.1	0.1	…	0.1
…	…	…	…	…	…	…	…	…
P	0.7	0.3	…	0.93	0.1	0.1	…	0.9

Limitations of a Single-Layer Feedforward Network (SLFN)

- **Cannot Solve Non-Linearly Separable Problems**
  - A single-layer perceptron can only learn **linearly separable** functions (i.e., problems where data can be divided by a straight line/hyperplane).
  - It fails on classic non-linear problems like the **XOR problem**, which requires at least one hidden layer (making it a multi-layer perceptron, MLP).
- **Limited to Linear Decision Boundaries**
  - Since it lacks hidden layers, it can only model linear relationships between inputs and outputs.
  - Real-world data often requires non-linear decision boundaries, which SLFNs cannot represent.
- **No Feature Hierarchy or Abstraction**
  - Deep networks learn hierarchical features (e.g., edges → shapes → objects in images), but SLFNs cannot extract higher-level features since they lack hidden layers.
- **Limited Expressivity (Representational Power)**
  - The Universal Approximation Theorem states that a **single hidden layer** is sufficient to approximate any continuous function, but a **zero-hidden-layer** network (SLFN) cannot.
- **Sensitive to Input Scaling & Preprocessing**
  - Since SLFNs rely on linear transformations, they perform poorly if features are not normalized or are highly correlated.
- **No Ability to Handle Sequential or Temporal Data**
  - Unlike recurrent neural network RNNs or long shot-term memory LSTMs, SLFNs have no memory and cannot process sequences or time-dependent data.
- **Vulnerable to Overfitting if Input Dimensionality is High**
  - If the input features are numerous, a single-layer network may overfit due to limited learning capacity.

Architectural graph of a multilayer perceptron with two hidden layers.

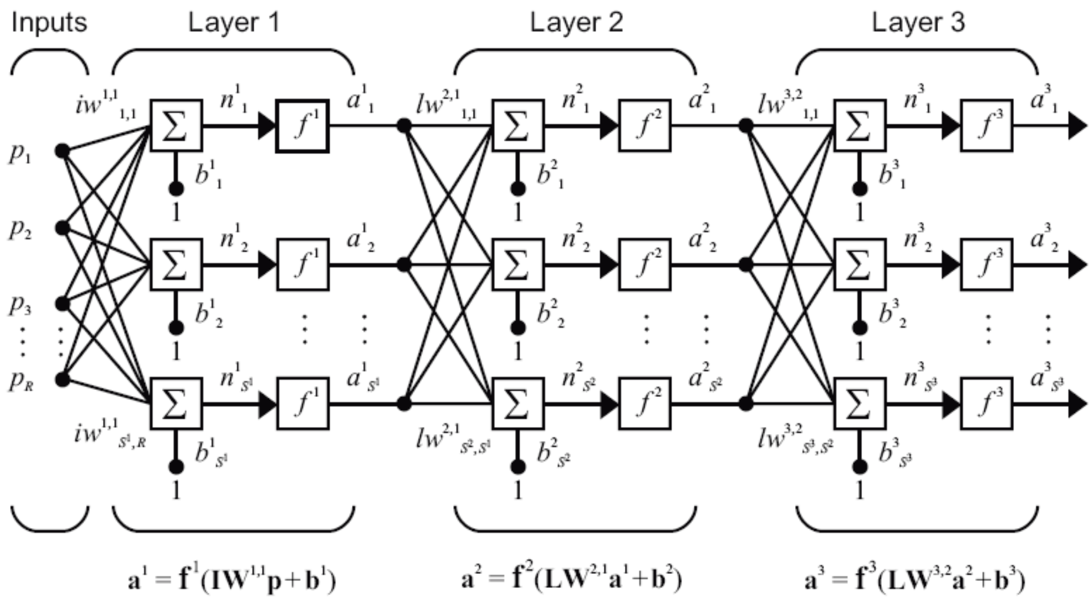


Motivation

- Single-layer networks have severe restrictions
  - Only linearly separable tasks can be solved
- Minsky and Papert (1969)
  - Showed the power of a two-layer feed-forward network
  - But didn't find the solution on how to train the network
- Werbos (1974)
  - Parker (1985), Cun (1985), Rumelhart (1986), Hinton (1987)
  - Solved the problem of training multi-layer networks by back-propagating the output errors through hidden layers of the network
- Backpropagation learning rule

2.1 Multilayer feedforward networks

- Important class of neural networks
  - Input layer (only distributing inputs, without processing)
  - One or more hidden layers
  - Output layer



- Commonly referred to as multilayer perceptron

Properties of multilayer perceptrons

1. Neurons include nonlinear activation function
   - Without nonlinearity, the capacity of the network is reduced to that of a single layer perceptron
   - Nonlinearity must be smooth (differentiable everywhere), not hard-limiting as in the original perceptron
   - Often, a logistic function is used:

$$y = \frac{1}{1 + \exp(-v)}$$

2. One or more layers of hidden neurons
   - Enable learning of complex tasks by extracting features from the input patterns

3. Full connectivity
   - Neurons in successive layers are fully interconnected

About backpropagation

- Multilayer perceptrons can be trained by the backpropagation learning rule
  - Based on the error-correction learning rule
  - Generalization of LMS learning rule (used to train ADALINE)

- Backpropagation consists of two passes through the network

1. Forward pass
   - Input is applied to the network
   - Input is propagated to the output
   - Synaptic weights stay frozen
   - Error signal is calculated

2. Backward pass
   - Error signal is propagated backward
   - Error gradients are calculated
   - Synaptic weights are adjusted

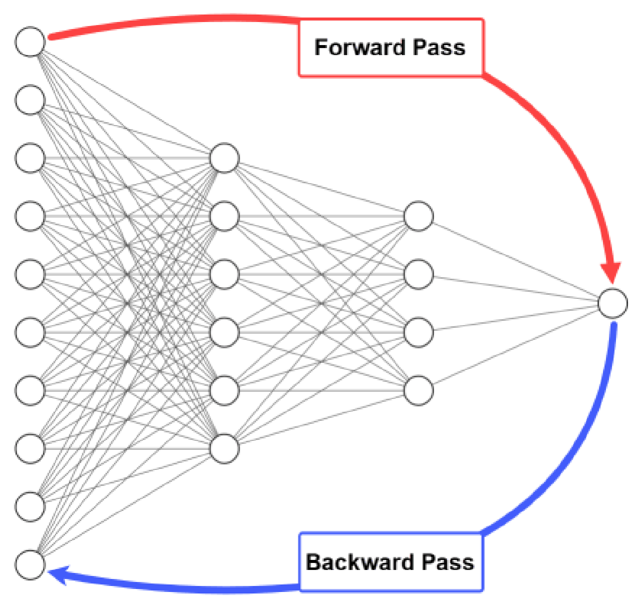
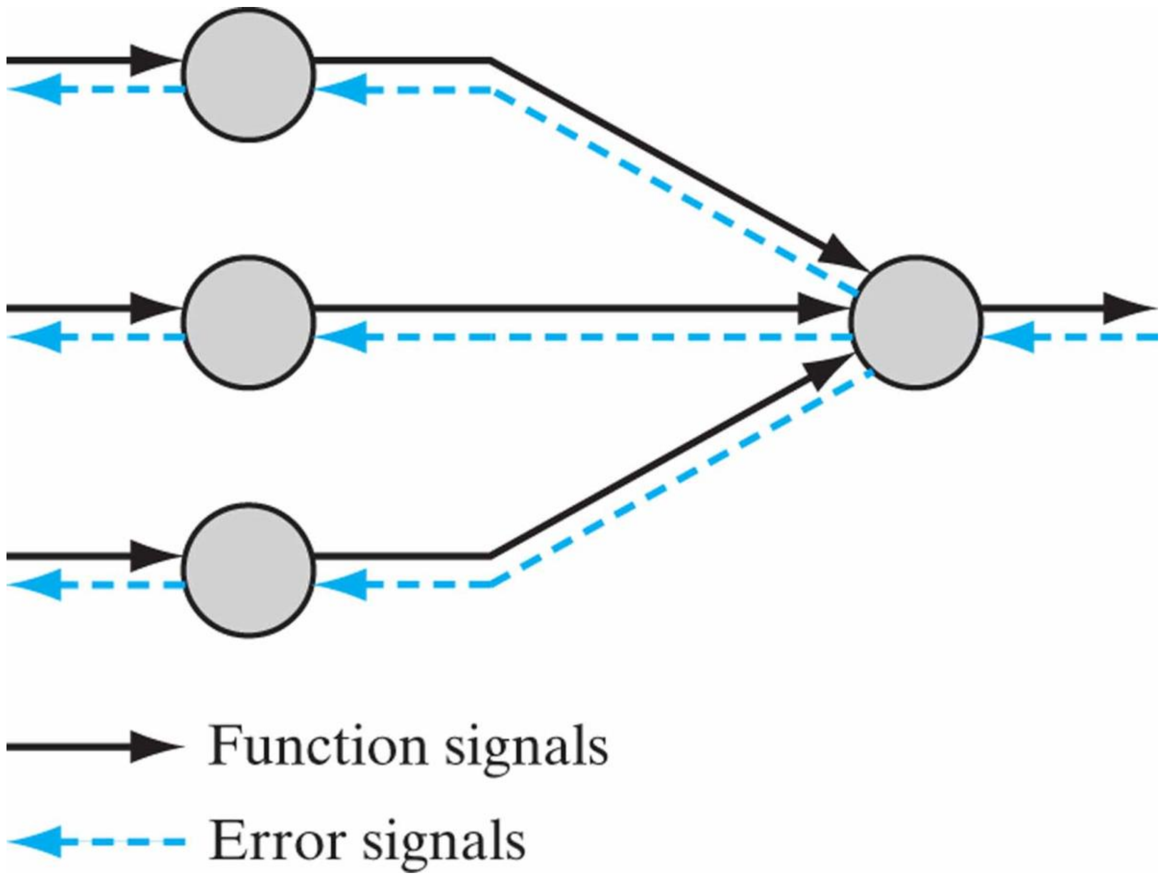


Illustration of the directions of two basic signal flows in a multilayer perceptron: forward propagation of function signals and back propagation of error signals.



2.2 Backpropagation algorithm (1/9)

- A set of learning samples (inputs and target outputs)

$$\{(x_n, d_n)\}_{n=1}^{N} \qquad x_n \in \Re^M, \quad d_n \in \Re^R$$

- Error signal at output layer, neuron *j*, learning iteration *n*

$$e_j(n) = d_j(n) - y_j(n)$$

- Instantaneous error energy of output layer with R neurons

$$E(n) = \frac{1}{2}\sum_{j=1}^{R} e_j(n)^2$$

- Average error energy over all learning set

$$\overline{E} = \frac{1}{N}\sum_{n=1}^{N} E(n)$$

Backpropagation algorithm (2/9)

- Average error energy $\bar{E}$ represents a cost function as a measure of learning performance

- $\bar{E}$ is a function of free network parameters
  - synaptic weights
  - bias levels

- Learning objective is to minimize average error energy $\bar{E}$ by adjusting free network parameters

- Learning results from many presentations of training examples
  - Epoch learning: network parameters are adjusted after presenting the entire training set
  - We use an approximation: pattern-by-pattern learning instead of epoch learning
    - Parameter adjustments are made for each pattern presented to the network
    - Minimizing instantaneous error energy at each step instead of average error energy

Backpropagation algorithm (3/9)

- Similar to the LMS algorithm, backpropagation applies correction of weights proportional to the partial derivative

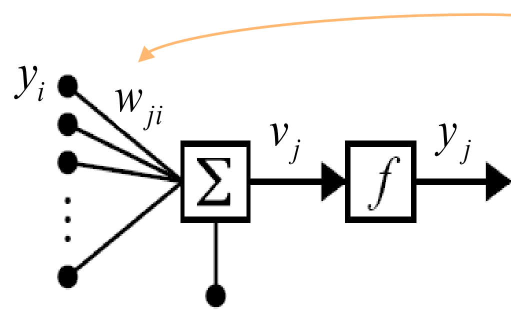
$$\Delta w_{ji}(n) \propto \frac{\partial E(n)}{\partial w_{ji}(n)}$$

Instantaneous error energy

- Expressing this gradient by the chain rule

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = \frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)}$$

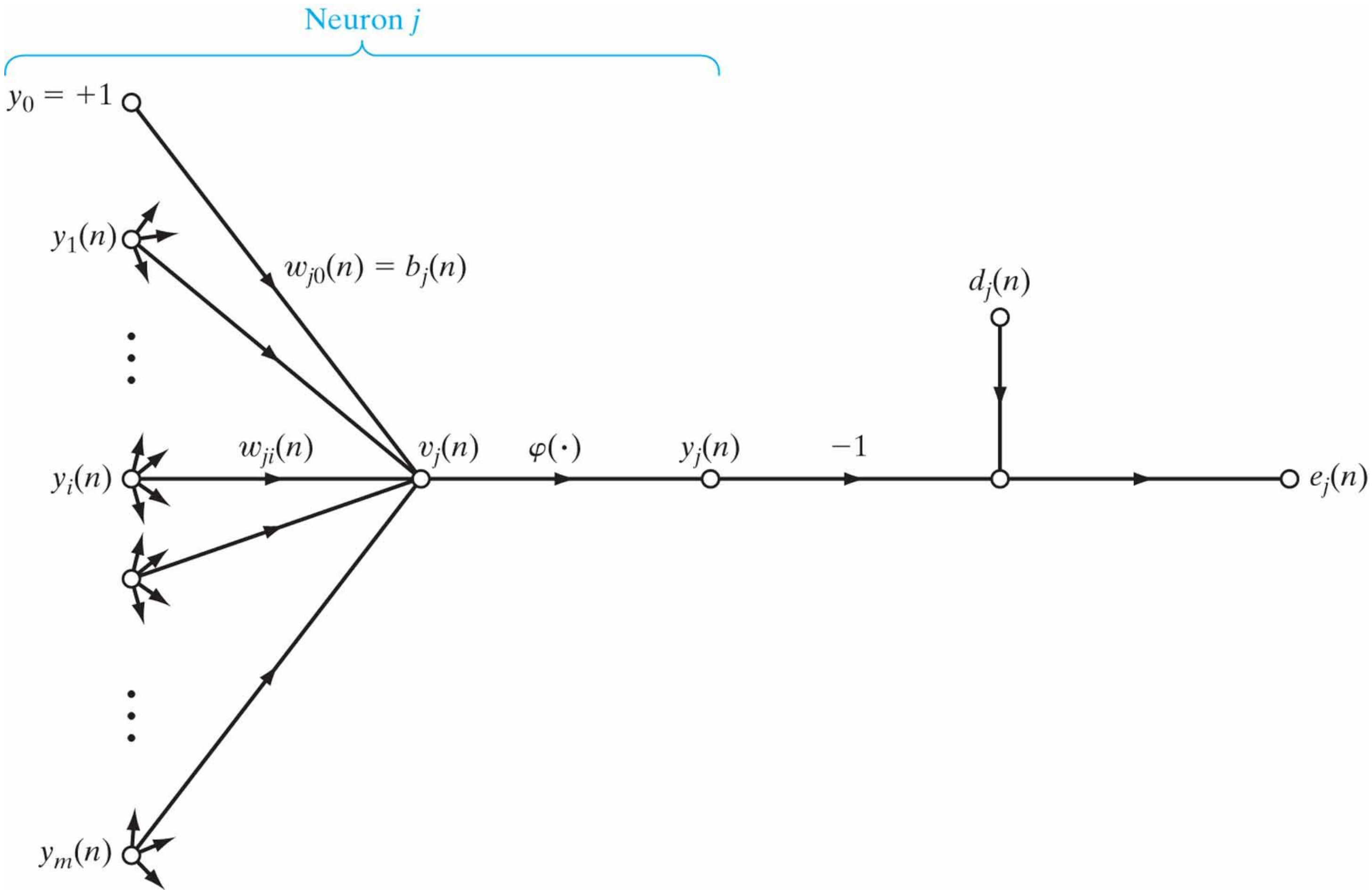
output error
network output
induced local field
synaptic weight



$$e_j = d_j - y_j$$

$$E = \frac{1}{2}\sum_{j=1}^{R} e_j^{\,2}$$

Signal-flow graph highlighting the details of output neuron *j*.



Backpropagation algorithm (4/9)

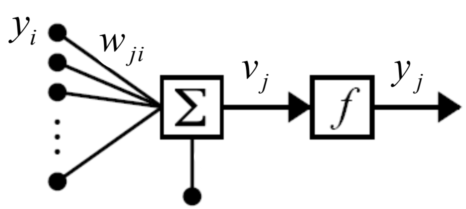
1. Gradient on output error

$$E(n) = \frac{1}{2}\sum_{n=1}^{N} e_j(n)^2 \quad \Longrightarrow \quad \frac{\partial E(n)}{\partial e_j(n)} = e_j(n)$$

2. Gradient on network output

$$e_j(n) = d_j(n) - y_j(n) \quad \Longrightarrow \quad \frac{\partial e_j(n)}{\partial y_j(n)} = -1$$

3. Gradient on induced local field



$$\Longrightarrow \quad \frac{\partial y_j(n)}{\partial v_j(n)} = f'(v_j(n))$$

4. Gradient on synaptic weight

$$v_j(n) = \sum_{j=0}^{R} w_{ji}(n) y_i(n) \quad \Longrightarrow \quad \frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n)$$

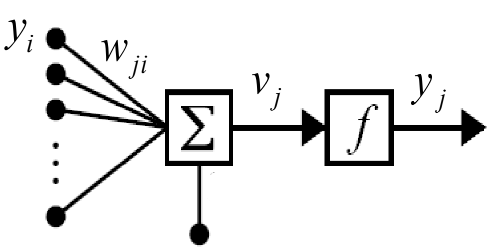
Backpropagation algorithm (5/9)

- Putting gradients together

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = \frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)}$$

$$= e_j(n) \quad (-1) \quad f'(v_j(n)) \; y_i(n)$$

$$= -e_j(n) f'(v_j(n)) y_i(n)$$



- Correction of synaptic weight is defined by the delta rule

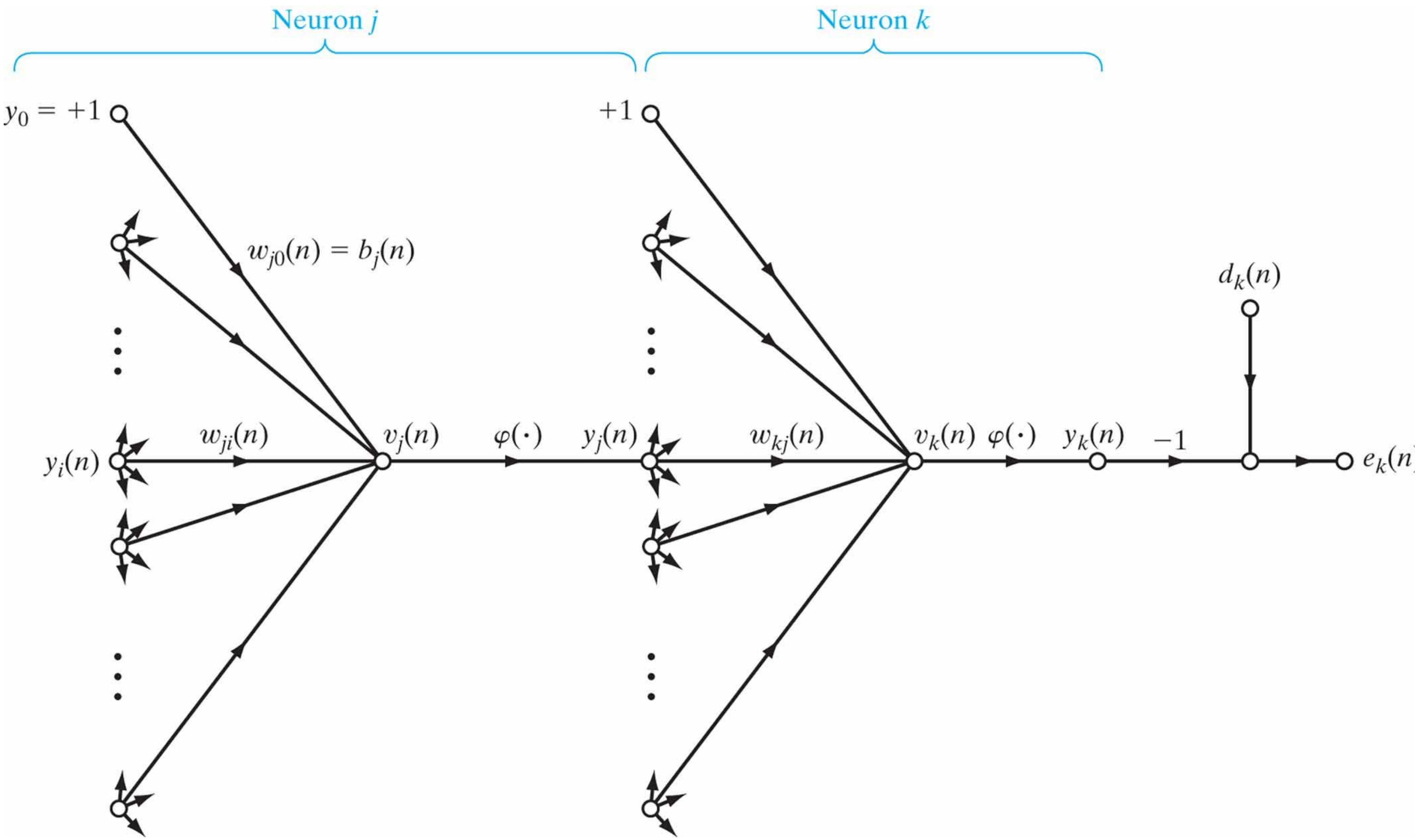
$$\Delta w_{ji}(n) = -\eta \frac{\partial E(n)}{\partial w_{ji}} = \eta \underbrace{e_j(n) f'(v_j(n))}_{\delta_j(n)} \; y_i(n)$$

Learning rate

Local gradient

$$\boxed{\Delta w_{ji}(n) = \eta \, \delta_j(n) \; y_i(n)}$$

Signal-flow graph highlighting the details of output neuron *k* connected to hidden neuron *j*.



Backpropagation algorithm (6/9)

CASE 1 Neuron *j* is an output node

- Output error $e_j(n)$ is available
- Computation of local gradient is straightforward

$$\boxed{\delta_j(n) = e_j(n) f'(v_j(n))}$$

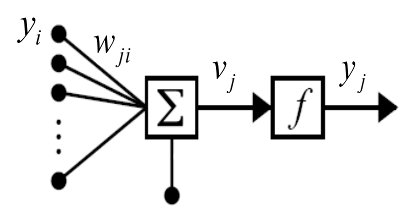
$$f(v_j(n)) = \frac{1}{1+\exp(-av_j(n))}$$

$$f'(v_j(n)) = \frac{a\exp(-av_j(n))}{[1+\exp(-av_j(n))]^2}$$

CASE 2 Neuron *j* is a hidden node

- Hidden error is not available → Credit assignment problem
- Local gradient solved by backpropagating errors through the network

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = \underbrace{\frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)}}_{-\delta_j(n)} \underbrace{\frac{\partial v_j(n)}{\partial w_{ji}(n)}}_{y_i(n)}$$



$$\frac{\partial y_j(n)}{\partial v_j(n)} = f'(v_j(n))$$

$$\delta_j(n) = -\frac{\partial E(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} = -\boxed{\frac{\partial E(n)}{\partial y_j(n)}} f'(v_j(n))$$

How to calculate the derivative of output error energy *E* on hidden layer output y_j ?

Backpropagation algorithm (7/9)

CASE 2 Neuron *j* is a hidden node ...

- Instantaneous error energy of the output layer with *R* neurons

$$E(n) = \frac{1}{2}\sum_{k=1}^{R} e_k(n)^2$$

- Expressing the gradient of output error energy *E* on hidden layer output y_j

$$\frac{\partial E(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial y_j(n)}$$

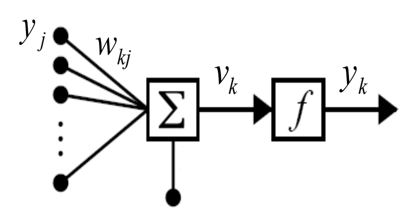
$$e_k(n) = d_k(n) - y_k(n)$$
$$= d_k(n) - f(v_k(n))$$

$$= \sum_k e_k \underbrace{\frac{\partial e_k(n)}{\partial v_k(n)}}_{-f'(v_k(n))} \underbrace{\frac{\partial v_k(n)}{\partial y_j(n)}}_{w_{kj}}$$

$$v_k(n) = \sum_{j=0}^{M} w_{kj}(n) y_j(n)$$

$$= -\sum_k e_k f'(v_k(n))\, w_{kj}$$

$$= -\sum_k \delta_k w_{kj}$$



Backpropagation algorithm (8/9)

CASE 2 Neuron *j* is a hidden node ...

- Finally, combining *ansatz* for hidden layer local gradient

$$\delta_j(n) = -\frac{\partial E(n)}{\partial y_j(n)} f'(v_j(n))$$

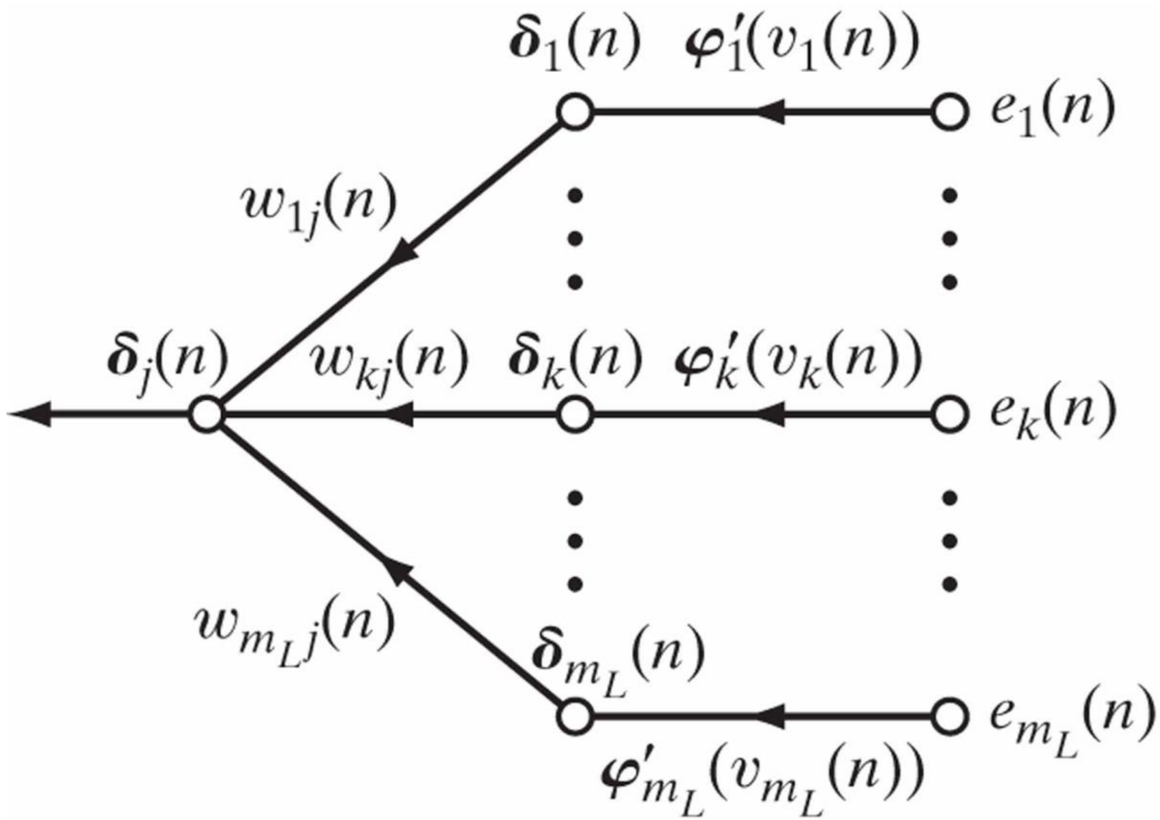
- and gradient of output error energy on hidden layer output

$$\frac{\partial E(n)}{\partial y_j(n)} = -\sum_k \delta_k w_{kj}$$

- gives final result for hidden layer local gradient

$$\boxed{\delta_j(n) = f'(v_j(n)) \sum_k \delta_k w_{kj}}$$

Signal-flow graph of a part of the adjoint system pertaining to back-propagation of error signals.



Backpropagation algorithm (9/9)

- Backpropagation summary

$$\boxed{\Delta w_{ji}(n) = \eta\, \delta_j(n)\, y_i(n)}$$

Weight correction

Learning rate

Local gradient

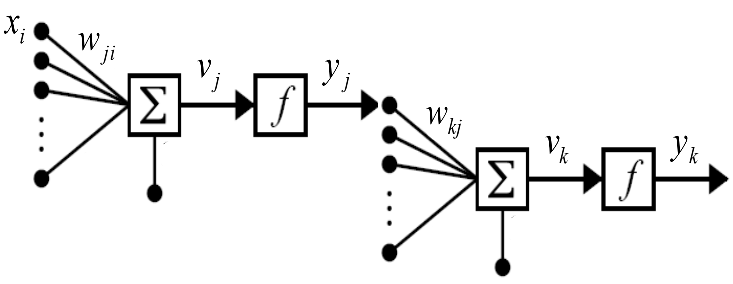
Input of neuron *j*

1. Local gradient of an output node

$$\boxed{\delta_k(n) = e_k(n) f'(v_k(n))}$$

2. Local gradient of a hidden node

$$\boxed{\delta_j(n) = f'(v_j(n)) \sum_k \delta_k w_{kj}}$$



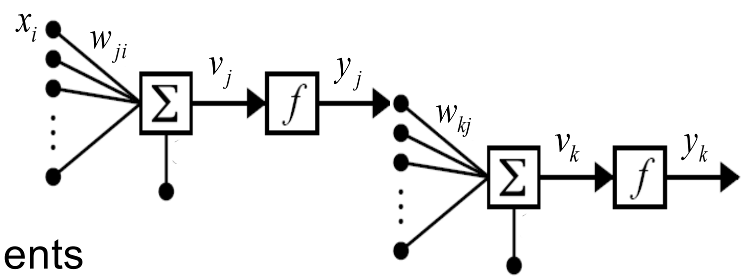
Two passes of computation

1. Forward pass

   Input is applied to the network and propagated to the output

   Inputs → Hidden layer output → Output layer output → Output error

   $x_i(n)$ → $y_j = f\left(\sum w_{ji} x_i\right)$ → $y_k = f\left(\sum w_{kj} y_j\right)$ → $e_k(n) = d_k(n) - y_k(n)$



2. Backward pass
   - Recursive computing of local gradients

     Output local gradients → Hidden layer local gradients

     $\delta_k(n) = e_k(n) f'(v_k(n))$ → $\delta_j(n) = f'(v_j(n)) \sum_k \delta_k w_{kj}$

   - Synaptic weights are adjusted according to local gradients

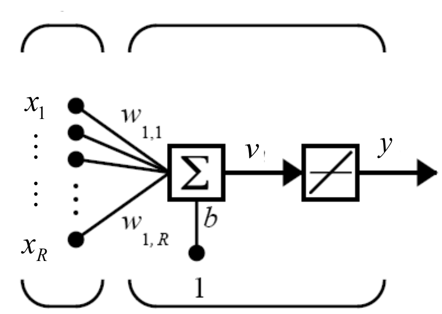
     $\Delta w_{kj}(n) = \eta\, \delta_k(n)\, y_j(n)$ $\Delta w_{ji}(n) = \eta\, \delta_j(n)\, x_i(n)$

Summary of the backpropagation algorithm

1. Initialization
   – Pick weights and biases from the uniform distribution with zero mean and variance that induces local fields between the linear and saturated parts of the logistic function
2. Presentation of training samples
   – For each sample from the epoch, perform forward pass and backward pass
3. Forward pass
   – Propagate training sample from network input to the output
   – Calculate the error signal
4. Backward pass
   – Recursive computation of local gradients from output layer toward input layer
   – Adaptation of synaptic weights according to generalized delta rule
5. Iteration
   – Iterate steps 2-4 until the stopping criterion is met

Backpropagation for ADALINE

- Using backpropagation learning for ADALINE
  - No hidden layers, one output neuron
  - Linear activation function



$$f(v(n)) = v(n) \quad \Rightarrow \quad f'(v(n)) = 1$$

- Backpropagation rule

$$\left.\begin{aligned} \Delta w_i(n) &= \eta\,\delta(n)\,y_i(n), \quad y_i = x_i \\ \delta(n) &= e(n)\,f'(v(n)) = e(n) \end{aligned}\right\} \quad \boxed{\Delta w_i(n) = \eta\,e(n)\,x_i(n)}$$

- Original delta rule

$$\boxed{\Delta w_i(n) = \eta\,e(n)\,x_i(n)}$$

- Backpropagation is a generalization of a delta rule

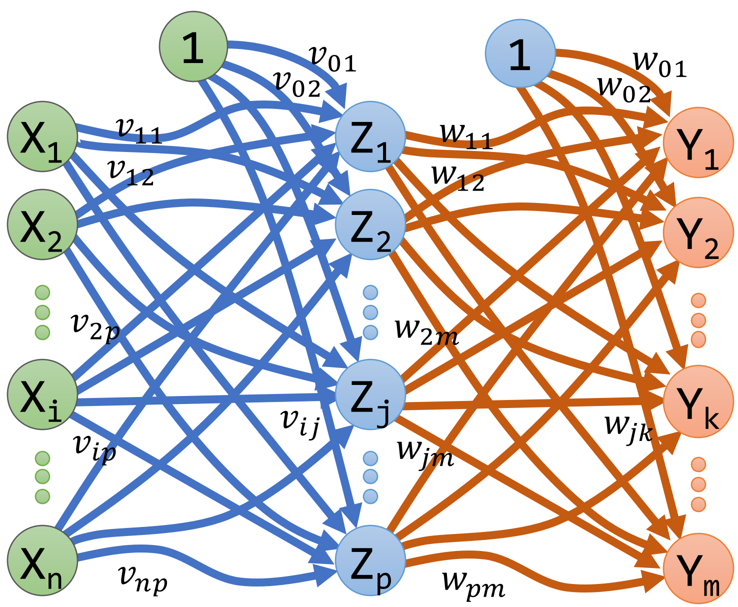
2.3 Working with backpropagation

- Efficient application of backpropagation requires some “fine-tuning”

- Various parameters, functions, and methods should be selected
  - Training mode (sequential / batch)
  - Activation function
  - Learning rate
  - Momentum
  - Stopping criterium
  - Heuristics for efficient backpropagation
  - Methods for improving generalization

Two-Layer Feedforward Neural Network – Backpropagation Algorithm

Architecture



Algorithm

1. Weights and other parameters are initialized.
2. For every pattern in training data of N patterns, do steps 3 to 6.
3. Feedforward Stage
   1. for j=1 to j=p
      a) $z\_in_j = v_{0j} + x_1 v_{1j} + x_2 v_{2j} + \cdots + x_n v_{nj} = \sum_{i=0}^{n} x_i v_{ij}$
      b) $z_j = f(z\_in_j)$
   2. for k=1 to k=m
      a) $y\_in_k = w_{0j} + z_1 w_{1j} + z_2 w_{2j} + \cdots + z_p w_{pk} = \sum_{j=0}^{p} z_j w_{jk}$
      b) $y_k = f(y\_in_k)$
   3. Find out the error $\varepsilon_k = t_k - y_k,\ k = 1,2,\ldots,m$
4. Backpropagation of Error Stage
   1. for k=1 to k=m
      a) $\delta_k = (t_k - y_k) f'(y\_in_k)$
      b) $\Delta w_{jk} = \eta\, \delta_k\, z_j$
   2. for j=1 to j=p
      a) $\delta\_in_j = \delta_1 w_{j1} + \delta_2 w_{j2} + \cdots + \delta_m w_{jm} = \sum_{k=1}^{m} \delta_k w_{jk}$
      b) $\delta_j = \delta\_in_j f'(z\_in_j)$
      c) $\Delta v_{ij} = \eta\, \delta_j\, x_i$
5. Update weights
   1. $w_{jk}(new) = w_{jk}(old) + \Delta w_{jk}, j = 1,2,\ldots,p, k = 1,2,\ldots,m$
   2. $v_{ij}(new) = v_{ij}(old) + \Delta v_{ij}, i = 1,2,\ldots,n, j = 1,2,\ldots,p$
6. Test for Convergence – Mean squared error $MSE = \frac{1}{2N} \sum_{k=1}^{m} \varepsilon_k^2$

Sequential vs. batch training

- Learning results from many presentations of training examples
  - Epoch = presentation of the entire training set

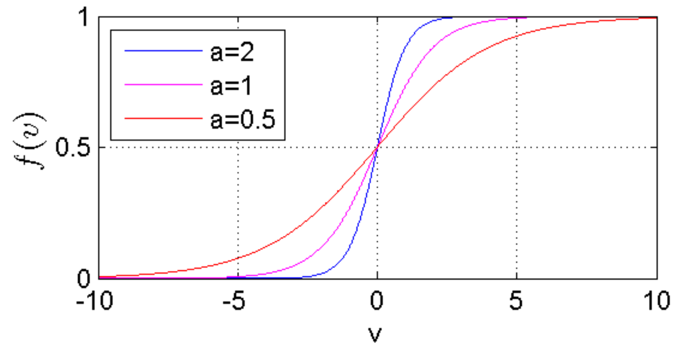
- Batch training (Epoch-by-Epoch Training)
  - Weight updating after the presentation of a complete epoch
  - Training is more accurate but very slow

- Sequential training (Pattern-by-Pattern or Stochastic Training)
  - Weight updating after the presentation of each training example
  - Stochastic nature of learning, faster convergence
  - Important practical reasons for sequential learning:
    - The algorithm is easy to implement
    - Provides an effective solution to large and difficult problems
  - Therefore sequential training is the preferred training mode
  - A good practice is random order of presentation of training examples

Activation function

- Derivative of activation function $f'(v_j(n))$ is required for computation of local gradients
  - The only requirement for activation function: differentiability
  - Commonly used: logistic function

$$f(v_j(n)) = \frac{1}{1+\exp(-av_j(n))} \qquad a > 0, \quad -\infty < v_j(n) < \infty$$



  - Derivative of the logistic function

$$f'(v_j(n)) = \frac{a\exp(-av_j(n))}{[1+\exp(-av_j(n))]^2} \xrightarrow{y_j(n)=f(v_j(n))} \boxed{f'(v_j(n)) = a\,y_j(n)[1-y_j(n)]}$$

Local gradient can be calculated without explicit knowledge of the activation function

Other activation functions

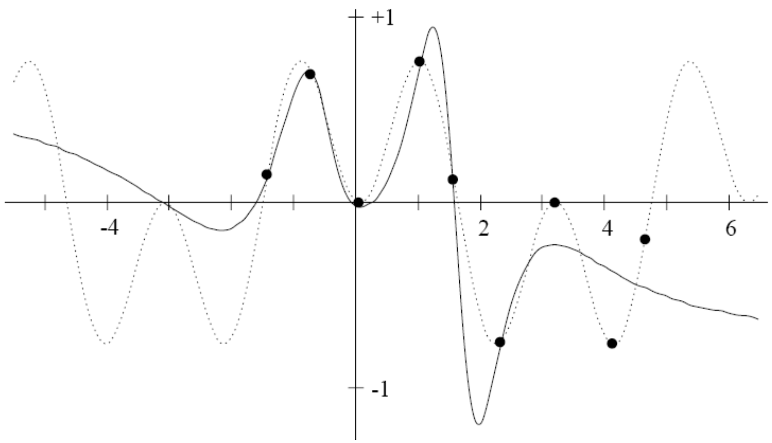
- Using sin() activation functions

$$f(x) = a + \sum_{k=1}^{\infty} c_k \sin(kx + \theta_k)$$

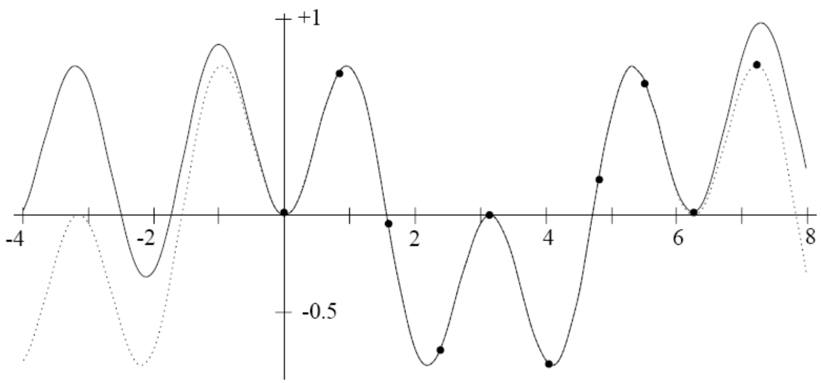
  - Equivalent to traditional Fourier analysis
  - Network with sin() activation functions can be trained by backpropagation

  - Example: Approximating a periodic function by

8 sigmoid hidden neurons

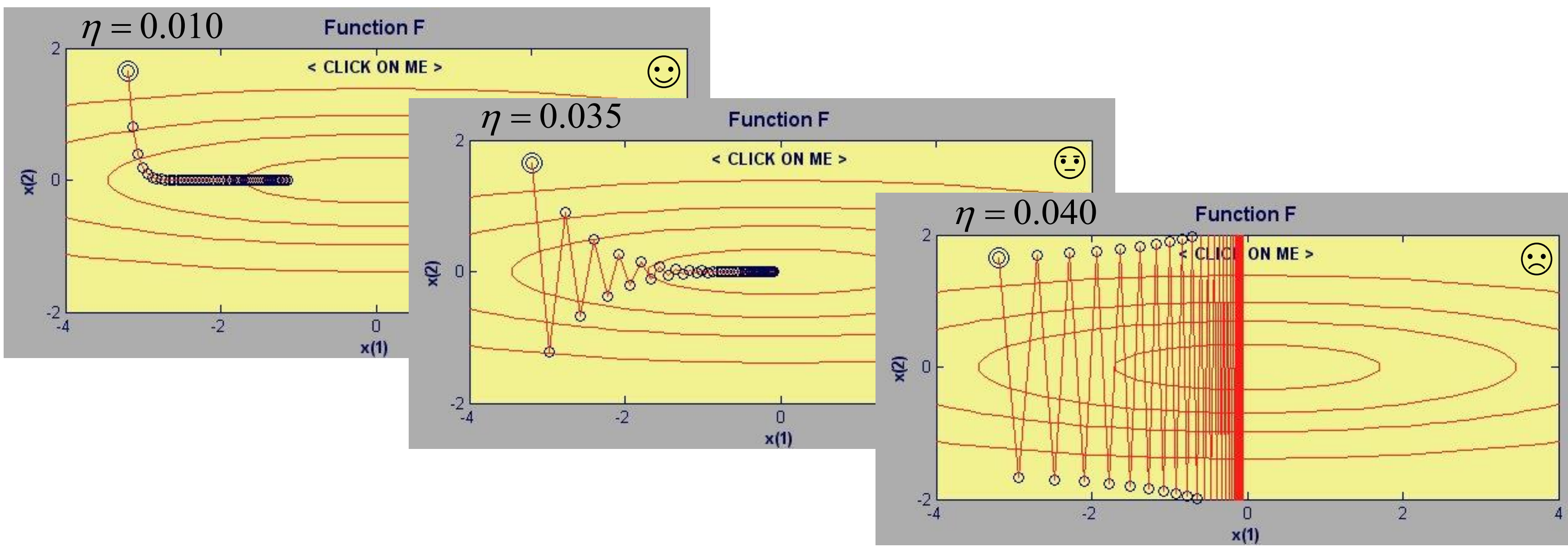


4 sin hidden neurons



Learning rate

- Learning procedure requires
  - Change in the weight space to be proportional to error gradient
  - True gradient descent requires infinitesimal steps
- Learning in practice
  - Factor of proportionality is learning rate η → $\Delta w_{ji}(n) = \eta\, \delta_j(n)\, y_i(n)$
  - Choose a learning rate as large as possible without leading to oscillations



Stopping criteria

- Generally, backpropagation cannot be shown to converge
  - No well-defined criteria for stopping its operation

- Possible stopping criteria

1. Gradient vector
   - Euclidean norm of the gradient vector reaches a sufficiently small gradient
2. Output error
   - Output error is small enough
   - Rate of change in the average squared error per epoch is sufficiently small
3. Generalization performance
   - Generalization performance has peaked or is adequate
4. Max number of iterations
   - We are out of time ...

Heuristics for efficient backpropagation (1/3)

1. Maximizing information content

General rule: every training example presented to the backpropagation algorithm should be chosen on the basis that its information content is the largest possible for the task at hand

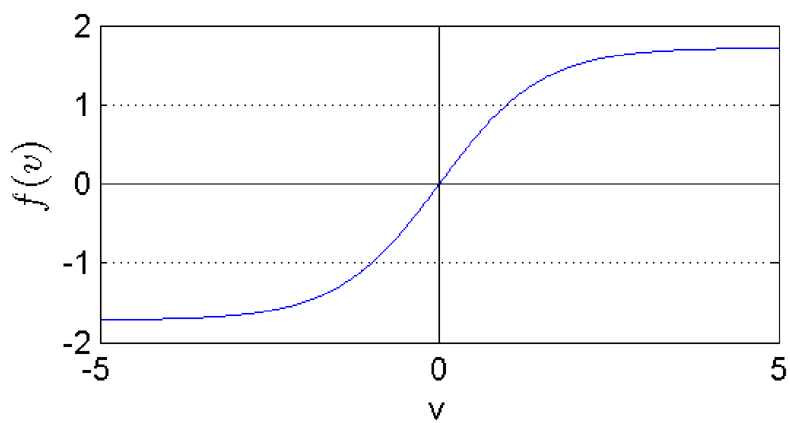
Simple technique: randomize the order in which examples are presented from one epoch to the next

2. Activation function

– Faster learning with antisimetric sigmoid activation functions

– Popular choice is:

$$f(v) = a\tanh(bv)$$
$$a = 1.72$$
$$b = 0.67$$



$f(1) = 1,\ f(-1) = -1$

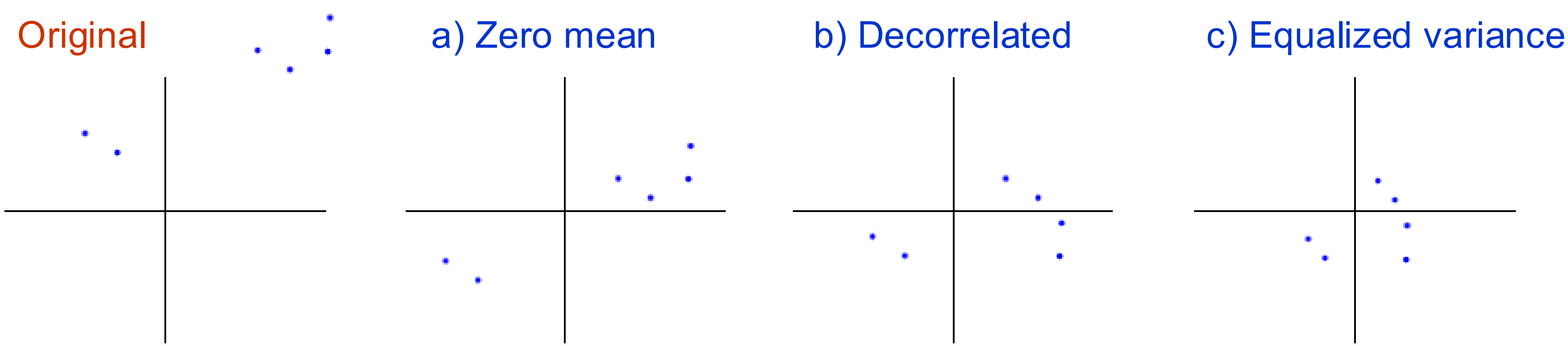
effective gain $f'(0) \approx 1$

max second derivative at $v = 1$

Heuristics for efficient backpropagation (2/3)

3. Target values
   – Must be in the range of the activation function
   – Offset is recommended, otherwise learning is driven into saturation
     • Example: max(target) = 0.9 max(f)

4. Preprocessing inputs
   a) Normalizing mean to zero
   b) Decorrelating input variables (by using principal component analysis)
   c) Scaling input variables (variances should be approx. equal)



Heuristics for efficient backpropagation (3/3)

5. Initialization
   - Choice of initial weights is important for a successful network design
     - Large initial values → saturation
     - Small initial values → slow learning due to operation only in the saddle point near origin
   - Good choice lies between these extreme values
     - Standard deviation of induced local fields should lie between the linear and saturated parts of its sigmoid function
     - *tanh* activation function example (a=1.72, b=0.67):
       synaptic weights should be chosen from a uniform distribution with zero mean and standard deviation

$$\sigma_v = m^{-1/2} \qquad m \text{ ... number of synaptic weights}$$

6. Learning from hints
   - Prior information about the unknown mapping can be included in the learning process
     - Initialization
     - Possible invariance properties, symmetries, ...
     - Choice of activation functions