Artificial Neural Networks

Outline

1. Introduction
2. Biological motivation
3. Neural network representation
4. Appropriate problems for neural network learning
5. Perceptrons
6. The perceptron training rule
7. Gradient descent and the Delta Rule
8. Visualizing the hypothesis space
9. Derivation of the gradient descent rule
10. Multilayer networks and the back-propagation algorithm
11. Remarks on the back-propagation algorithm
12. Advanced topics in artificial neural networks

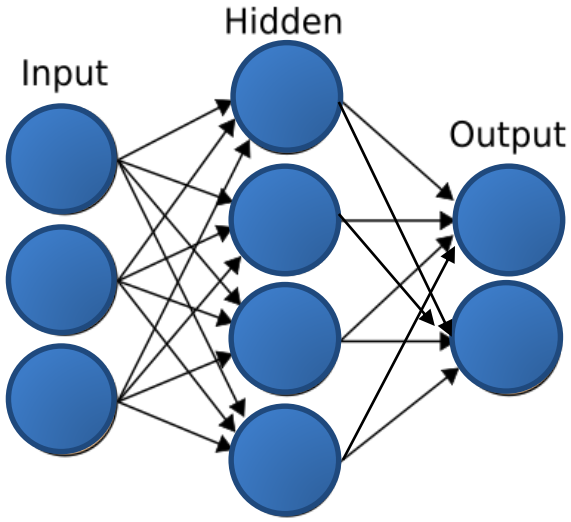
Videos

1. But what is a Neural Network? Deep learning, chapter 1.
2. Gradient descent, how neural networks learn | Deep learning, chapter 2
3. What is backpropagation really doing? | Deep learning, chapter 3
4. Backpropagation calculus | Deep learning, chapter 4

1. Introduction

- Neural network learning methods provide a robust approach to approximating
  - real-valued,
  - discreet-valued, and
  - vector-valued target

  functions.



- For some types of problems, e.g. interpreting complex real-world sensor data, artificial neural networks are very effective.
- **Backpropagation method** has been successfully applied in many practical problems such as learning to recognize handwritten characters and faces.

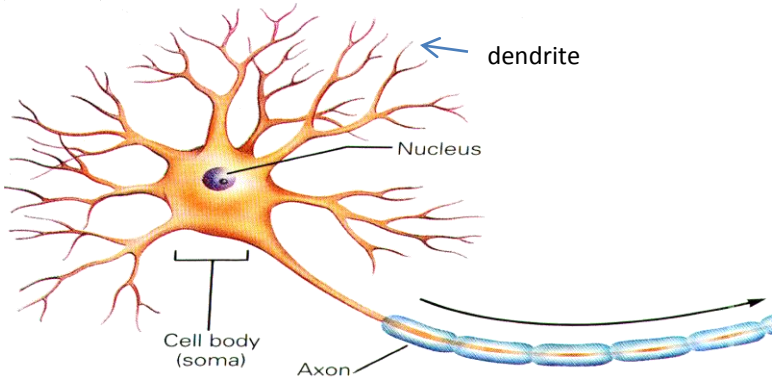
2. Biological motivation

- Human brain contains estimated 10^{11} neurons, each connected on average to 10^4 others.
- Neuron activity is typically excited or inhibited through connections to other neurons.
- The fastest neuron switching times are in the order 10^{-3} seconds.
- It requires approximately 10^{-1} seconds to visually recognize your mother.
- There cannot be more than a few hundred steps in the sequence of neuron firings in 0.1 second interval.
- Speculation: information processing abilities of biological neural systems must follow highly parallel processes operating on representations that are distributed over many neurons.

Biological motivation

Neurons

Cells (processing elements) of a biological or artificial neural network



Nucleus

The central processing portion of a neuron

Dendrite

The part of a biological neuron that provides inputs to the cell

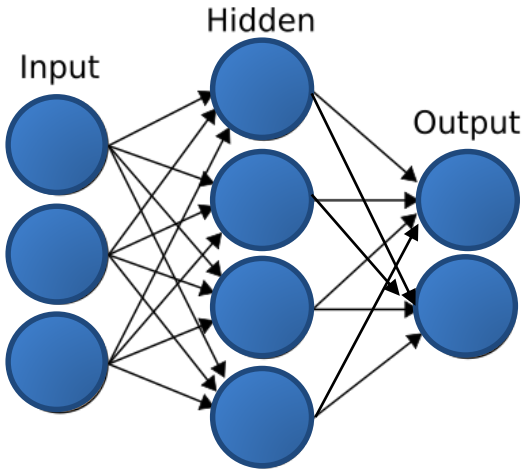
Axon

An outgoing connection (i.e., terminal) from a biological neuron

Synapse

The connection (where the weights are) between processing elements in a neural network

3. Neural network (NN) representation



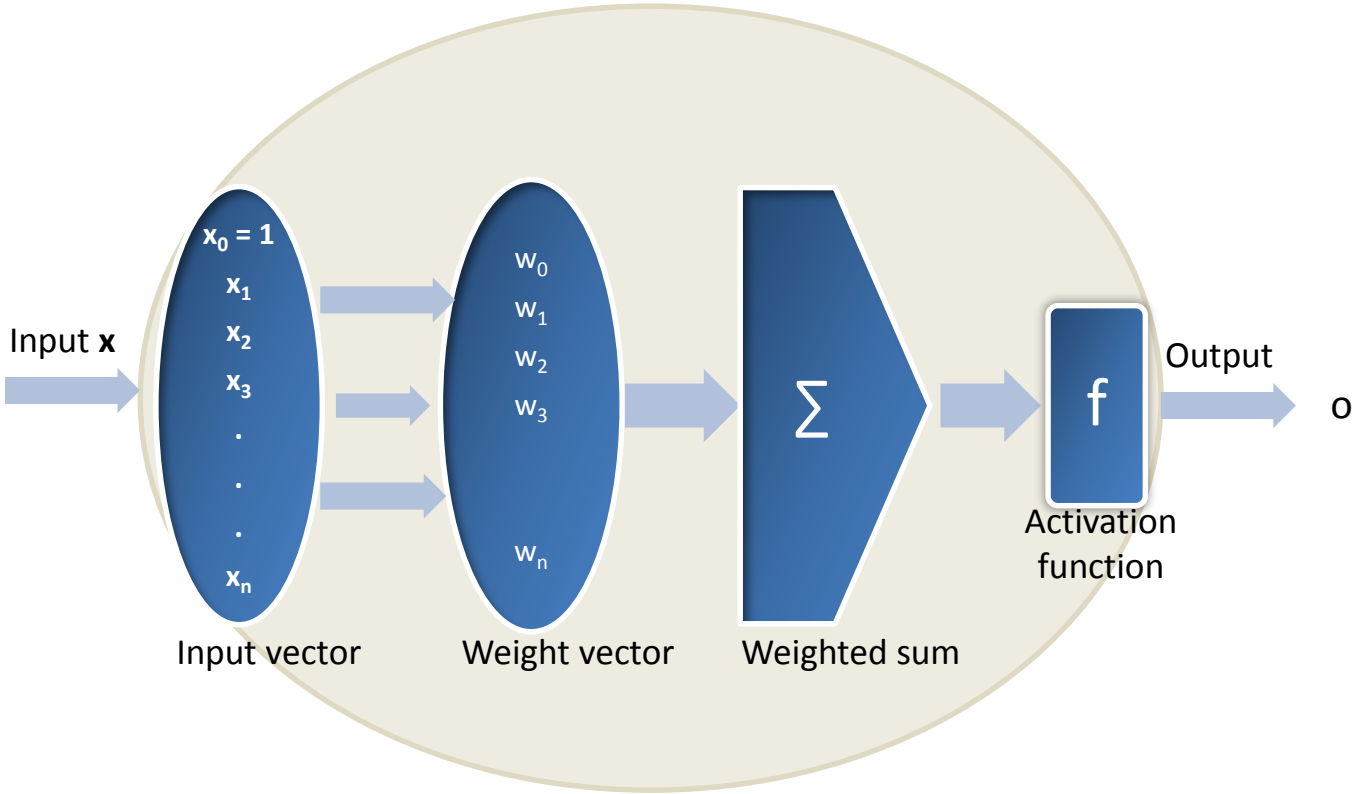
- Each node (circle) corresponds to a single network unit.
- The arrows represent the inputs to a hidden unit or to an output unit.
- The four hidden units are “hidden” because their output is available only within the network and is not available as part of the global network output.
- Each of the hidden units computes a single real-valued output based on the weighted combination of its inputs.
- These hidden units are then used as inputs to the second layer of consisting of 2 output units.
- Each output unit normally has binary target values. For example, the two outputs may have target values (0,1) and (1,0) for a binary classification problem.
- There are 3×4 connections between input layer and hidden layer and 4×2 connections between hidden layer and output layer.
- Network learning corresponds to choosing a weight value for each of these connections.

4. Appropriate problems for NN learning

- Artificial Neural Network (ANN) is well suited to problems in which the training data corresponds to noisy, complex sensor data such as inputs from camera and microphone.
- It is also applicable to problems for which more symbolic representations are often used.
- The backpropagation algorithm is the most commonly used ANN learning technique.
- It is appropriate for problems with the following characteristics:
  - Instances are represented by many attribute-value pairs.
  - The target function output may be discrete-valued, real-valued, or a vector of several real – or discrete-valued attributes.
  - The training examples may contain errors.
  - Long training time are acceptable.
  - Fast evaluation of the learned target function may be required.
  - The ability of humans to understand the learned target is not important.

5. Perceptrons

- One type of ANN system is based on a unit called **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 a threshold, -1 otherwise.



$$o = f(w_0x_0 + w_1x_1 + w_2x_2 + w_3x_3 + \ldots w_nx_n)$$

5. Perceptrons

- Given an observation **x**, the prediction by the perceptron is computed by applying the activation function f to the linear combination of the predictors:

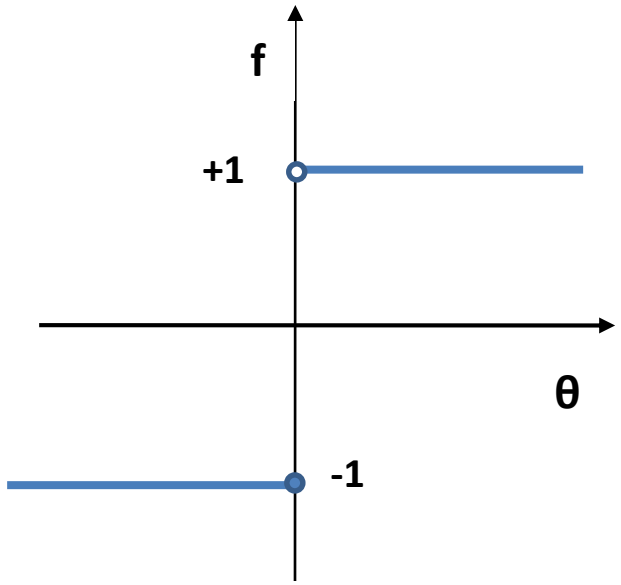
$$o = f(w_0x_0 + w_1x_1 + w_2x_2 + w_3x_3 + \ldots w_nx_n)$$

- The function f maps the linear combination into the set $\mathbb{H} = \{-1,1\}$:
  - $f(\theta) = 1$ if $\theta > 0$

    $= -1$ otherwise

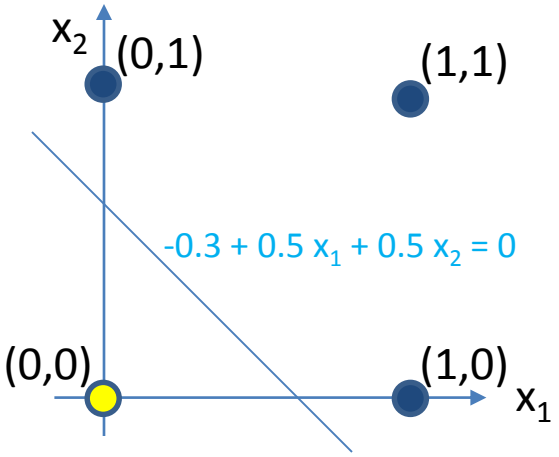
- w_0 is called the *distortion* or *bias* when

  x_0 is set to 1 for all data samples.



Representational power of perceptrons

- A single perceptron can be used to represent many Boolean functions:
  - AND: let $w_0 = -0.8$ and $w_1 = w_2 = 0.5$
  - OR: let $w_0 = -0.3$ and $w_1 = w_2 = 0.5$
  - AND and OR are special cases of m-of-n functions.
  - AND: m = n, OR: m = 1



- m-of-n function: if (m of the following n conditions are true), then ……
- Any m-of-n function is easily represented using a perceptron by setting all input weights to the same value and then setting the threshold accordingly.
- Perceptron can represent all primitive Boolean functions AND, OR, NAND, NOR.
- It cannot represent XOR function.
- Every Boolean function can be represented by some network of perceptrons only two level deep.

6. The perceptron training rule

- **Training**: determine a weight vector that causes the perceptron to produce correct +1 or -1 output for each of the given training examples:
- Begin with a random weight and iteratively apply the perceptron to each training example.
- Modify the weights whenever the output is not correct.
- Weights are corrected as follows:
  - $\mathbf{w}_i \Leftarrow \mathbf{w}_i + \Delta\mathbf{w}$, where
  - $\Delta\mathbf{w} = \eta(t - o)\,\mathbf{x}_i$
- Here:
  - t is the target output of the current training example $\mathbf{x}_i$
  - o is the output generated by the perceptron
  - η is a positive learning constant usually set to a small value, e.g. 0.1.
  - $\mathbf{x}_i$ is the i-th data sample (n-dimensional vector).
  - $\Delta\mathbf{w}$ is the change in weight given the data sample $\mathbf{x}_i$

The perceptron training rule

Example.

- Classify 4 input patterns with known class membership:
  - $\mathbf{x}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ $\mathbf{x}_2 = \begin{bmatrix} 1 \\ -0.5 \end{bmatrix}$ $\mathbf{x}_3 = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$ $\mathbf{x}_4 = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$
  - $t_1 = t_3 = +1$ (Class 1)
  - $t_2 = t_4 = -1$ (Class 2)

- Assume learning rate $\eta = 0.5$, initial weight $\mathbf{w}_1 = \begin{bmatrix} 1.75 \\ -2.5 \end{bmatrix}$

- Input patterns presented sequentially as $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$ and $\mathbf{x}_4$ in a complete training cycle (also called an **epoch**).

The perceptron training rule

Step 1: Pattern $\mathbf{x}_1$ as input

$o_1 = f(w_{1,0}x_{1,0} + w_{1,1}x_{1,1}) = f\{(1.75)(1) + (-2.5)(1)\} = f(-0.75) = -1$

$\Delta\mathbf{w} = \eta(t_1 - o_1)\,\mathbf{x}_1 = 0.5(1 + 1)\,\mathbf{x}_1 = \mathbf{x}_1$

$\mathbf{w}_2 = \mathbf{w}_1 + \Delta\mathbf{w}$

$$= \begin{bmatrix}1.75\\-2.5\end{bmatrix} + \begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}2.75\\-1.5\end{bmatrix}$$

Step 2: Pattern $\mathbf{x}_2$ as input

$o_2 = f(w_{2,0}x_{2,0} + w_{2,1}x_{2,1}) = f\{(2.75)(1) + (-1.5)(-0.5)\} = f(3.5) = +1$

$\Delta\mathbf{w} = \eta(t_2 - o_2)\,\mathbf{x}_2 = 0.5(-1 - 1)\,\mathbf{x}_2 = -\mathbf{x}_2$

$\mathbf{w}_3 = \mathbf{w}_2 + \Delta\mathbf{w}$

$$= \begin{bmatrix}2.75\\-1.5\end{bmatrix} + \begin{bmatrix}-1\\0.5\end{bmatrix} = \begin{bmatrix}1.75\\-1\end{bmatrix}$$

$\mathbf{x}_1 = \begin{bmatrix}1\\1\end{bmatrix}$ $\mathbf{x}_2 = \begin{bmatrix}1\\-0.5\end{bmatrix}$ $\mathbf{x}_3 = \begin{bmatrix}1\\3\end{bmatrix}$ $\mathbf{x}_4 = \begin{bmatrix}1\\-2\end{bmatrix}$

$t_1 = t_3 = +1$ (Class 1)

$t_2 = t_4 = -1$ (Class 2)

The perceptron training rule

Step 3: Pattern $\mathbf{x}_3$ as input

$o_3 = f(w_{3,0}x_{3,0} + w_{3,1}x_{3,1}) = f\{ (1.75)(1) + (-1)(3) \} = f(-1.25) = -1$

$\Delta\mathbf{w} = \eta(t_3 - o_3)\,\mathbf{x}_3 = 0.5(1 + 1)\,\mathbf{x}_3 = \mathbf{x}_3$

$\mathbf{w}_4 = \mathbf{w}_3 + \Delta\mathbf{w}$

$$= \begin{bmatrix} 1.75 \\ -1 \end{bmatrix} + \begin{bmatrix} 1 \\ 3 \end{bmatrix} = \begin{bmatrix} 2.75 \\ 2 \end{bmatrix}$$

Step 4: Pattern $\mathbf{x}_4$ as input

$o_1 = f(w_{2,0}x_{2,0} + w_{2,1}x_{2,1}) = f\{ (2.75)(1) + (2)(-2) \} = f(-1.25) = -1$

$\Delta\mathbf{w} = \eta(t_4 - o_4)\,\mathbf{x}_4 = 0.5(-1+ 1)\,\mathbf{x}_4 = \mathbf{0}$

$\mathbf{w}_5 = \mathbf{w}_4$

$\mathbf{x}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ $\mathbf{x}_2 = \begin{bmatrix} 1 \\ -0.5 \end{bmatrix}$ $\mathbf{x}_3 = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$ $\mathbf{x}_4 = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$

$t_1 = t_3 = +1$ (Class 1)

$t_2 = t_4 = -1$ (Class 2)

End of first epoch and there is no change in weights.

Is training complete? NO

The perceptron training rule

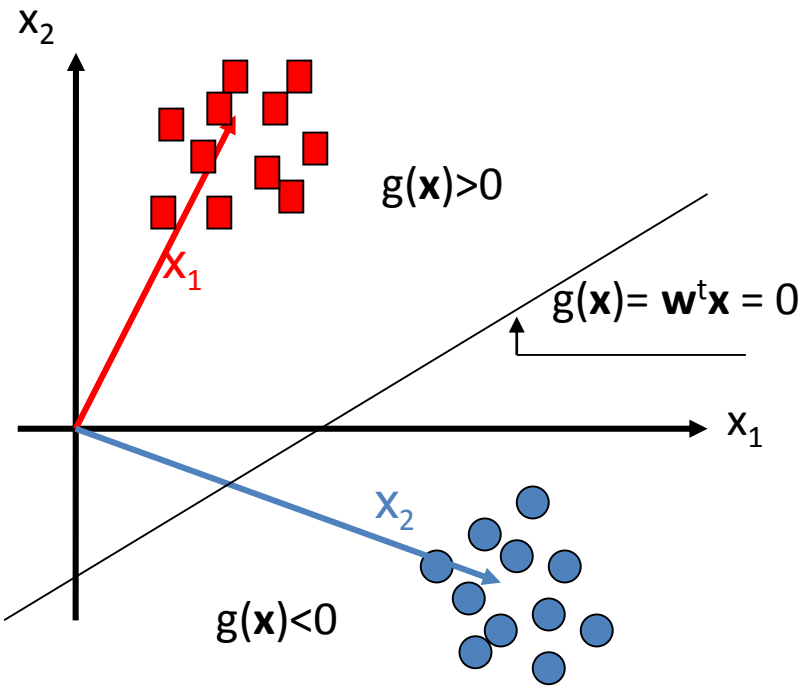
- Step 5 ($\mathbf{x}_1$) : $\mathbf{w}_6 = \mathbf{w}_5$
- Step 6 ($\mathbf{x}_2$) : $\mathbf{w}_7 = \begin{bmatrix} 1.75 \\ 2.5 \end{bmatrix}$
- Step 7-9 ($\mathbf{x}_3$, $\mathbf{x}_4$, $\mathbf{x}_1$) : $\mathbf{w}_{10} = \mathbf{w}_9 = \mathbf{w}_8 = \mathbf{w}_7$ (no adjustment)
- Step 10 ($\mathbf{x}_2$) : $\mathbf{w}_{11} = \begin{bmatrix} 0.75 \\ 3 \end{bmatrix}$ (final weights)
- It took 10 pattern presentations (in practice, 3 epochs)

$$\mathbf{Xw} = \begin{bmatrix} 1 & 1 \\ 1 & -0.5 \\ 1 & 3 \\ 1 & -2 \end{bmatrix} \begin{bmatrix} 0.75 \\ 3 \end{bmatrix} = \begin{bmatrix} 3.75 \\ -0.75 \\ 9.75 \\ -5.25 \end{bmatrix}$$

$$f(\mathbf{Xw}) = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}$$

Perceptron

Perceptron Convergence Theorem: Perceptron learns correct dichotomization of two-class patterns in a finite number of steps, k, regardless of η and $\mathbf{w_0}$, provided classes are linearly separable, i.e., $\mathbf{w}^* = \mathbf{w_{k+1}} = \mathbf{w_{k+2}} = \mathbf{w_{k+3}}$ .....



7. Gradient descent and Delta Rule

- If samples are not linearly separable, perceptron rule may fail to converge.
- Delta rule converges toward a best-fit approximation to the target concept when the samples are not linearly separable.
- Key idea: **Gradient descent** to search the hypothesis space to find the best weights.
- Consider the case of a linear unit with no threshold, the output for input $\mathbf{x_i}$ is computed as

$$o_i = \mathbf{w}^t\mathbf{x_i} = w_1x_{i1} + w_2x_{i2} + ... + w_nx_{in}$$

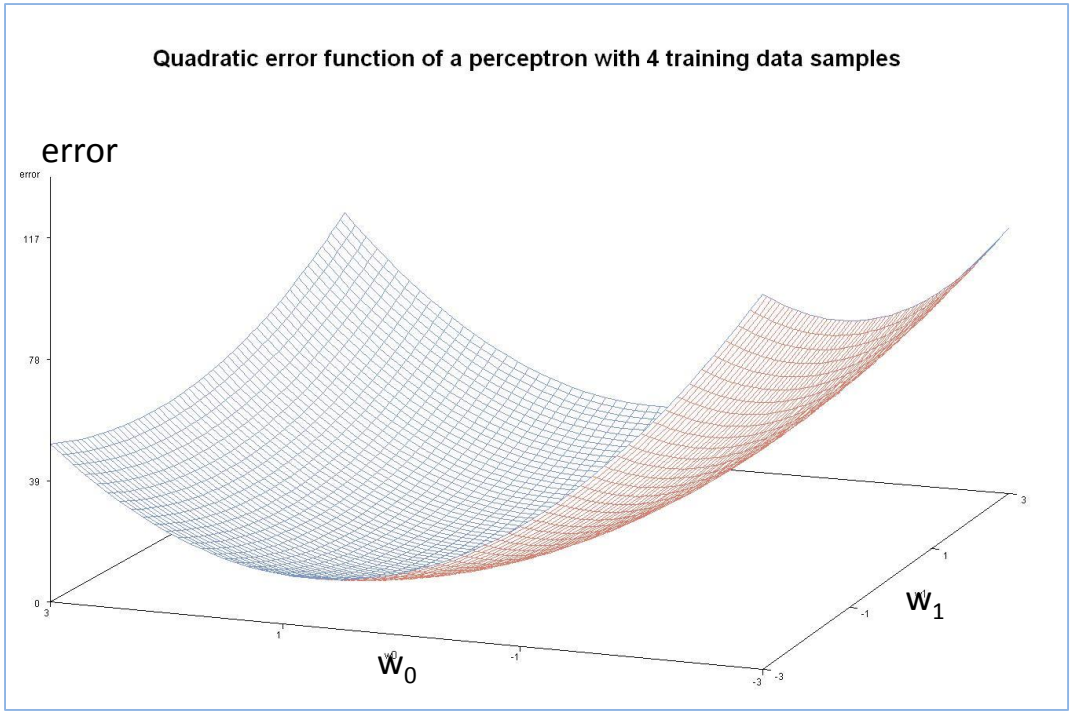
- The training error for the training data set is defined as

$$E(\mathbf{w}) = \tfrac{1}{2} \sum_d (t_d - o_d)^2$$

where t_d is the target value and o_d is the unit's output for input $\mathbf{x_d}$.

8. Visualizing the hypothesis space

- **Visualizing the hypothesis space**



- The axes w_0 and w_1 represent possible values of the two weights of a simple linear unit.
- The w_0,w_1 plane represents the entire hypothesis space.
- The vertical axis indicates the error E relative to the 4 training examples..
- Gradient descent search determines a weight vector that minimizes E by moving along the steepest descent direction along the error surface.

9. Derivation of the gradient descent rule.

Derivation of the gradient descent rule.

- The direction of steepest descent along the hypothesis space can be obtained by computing the derivative of the error function E(**w**).
- The derivative (**gradient**) is

$$\nabla E(\mathbf{w}) \equiv [\partial E/\partial \mathbf{w}_0, \partial E/\partial \mathbf{w}_1, \ldots\ldots, \partial E/\partial \mathbf{w}_n]^t$$

- The negative of this gradient gives the direction of steepest descent.
- The training rule for gradient descent is

$$\mathbf{w} \Leftarrow \mathbf{w} + \Delta \mathbf{w}$$

where: $\Delta \mathbf{w} = - \eta \nabla E(\mathbf{w})$

- Working on each component of the weight vector **w**, the training rule can be written as

$$\mathbf{w}_j \Leftarrow \mathbf{w}_j + \Delta \mathbf{w}_j$$

where: $\Delta \mathbf{w}_j = - \eta \partial E/\partial \mathbf{w}_j$

Derivation of the gradient descent rule.

Efficient gradient calculation.

- At each iteration we need to compute $\partial E/\partial \mathbf{w}_j$
- This can be done as follows:

$$\partial E/\partial \mathbf{w}_j = \partial/\partial \mathbf{w}_j \; \tfrac{1}{2} \sum_d (t_d - o_d)^2$$

$$= \tfrac{1}{2} \sum_d \partial/\partial \mathbf{w}_j \; (t_d - o_d)^2$$

$$= \tfrac{1}{2} \sum_d 2 (t_d - o_d) \; \partial/\partial \mathbf{w}_j \; (t_d - o_d)$$

$$= \tfrac{1}{2} \sum_d 2 (t_d - o_d) \; \partial/\partial \mathbf{w}_j \; (t_d - \mathbf{w}^t \mathbf{x}_d)$$

$$= \sum_d (t_d - o_d) (-\mathbf{x}_{d,j})$$

Eg. $\mathbf{x}_1 = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$ $t_1 = 1$

$$\partial/\partial \mathbf{w}_j \; (t_d - \mathbf{w}^t \mathbf{x}_d)$$
$$= \partial/\partial \mathbf{w}_j \; (1 - 3 w_1 + 2 w_2)$$

For $j = 1$, $\partial/\partial \mathbf{w}_1 \; (t_d - \mathbf{w}^t \mathbf{x}_d) = -3$
$j = 2$, $\partial/\partial \mathbf{w}_2 \; (t_d - \mathbf{w}^t \mathbf{x}_d) = 2$

- The weight update rule for gradient descent is then:

$$\Delta \mathbf{w}_j = -\eta \, \partial E/\partial \mathbf{w}_j$$

$$= \eta \sum_d (t_d - o_d) \, \mathbf{x}_{d,j}$$

Compare to weight update formula on page 11

$$\Delta \mathbf{w} = \eta (t - o) \, \mathbf{x}_i$$

Derivation of the gradient descent rule.

The algorithm.

- Input: $(\mathbf{x}_d, t_d)$, d = 1,2, …. N, $\mathbf{x}_d$ is the vector of input values, t_d is its corresponding target value.
- Learning parameter: η
- Initialize each $\mathbf{w}_j$ to a small random value
- Until termination condition is met, do
  - Initialize each $\Delta\mathbf{w}_j$ to zero.
  - For each $(\mathbf{x}_d, t_d)$ in the data set, do
    - Input the data $\mathbf{x}_d$ to the unit and compute the output o_d
    - For each linear unit weight $\mathbf{w}_j$, do

$$\Delta \mathbf{w}_j \Leftarrow \Delta \mathbf{w}_j + \eta \, (t_d - o_d) \, \mathbf{x}_{d,j}$$

  - Update the weight $\mathbf{w}_j$

$$\mathbf{w}_j \Leftarrow \mathbf{w}_j + \Delta\mathbf{w}_j$$

Derivation of the gradient descent rule.

Example.

- Tasting score of a certain processed cheese.
- The two predictors: scores for fat and salt (0 is the minimum amount and 1 is the maximum amount possible).
- The output: 1 indicates a test panel likes the cheese, 0 otherwise.

Observation	Fat score	Salt score	Acceptance
1	0.2	0.9	1
2	0.1	0.1	0
3	0.2	0.4	0
4	0.2	0.5	0
5	0.4	0.5	1
6	0.3	0.8	1

Derivation of the gradient descent rule.

Example.

- Let $\mathbf{w} = (1 \quad 1 \quad -1)^t$, where the 3rd component $\mathbf{w}_3 = -1$ is the bias
- The predictions and errors with these weights are as follows:

Obs. d	Fat score	Salt score		t_d	$o_d = w^t x_d$	$(t_d - o_d)^2$
	x					
1	0.2	0.9	1	1	(1)(0.2) + (1)(0.9) + (-1)(1) = 0.10	$(1\text{-}0.10)^2 = 0.81$
2	0.1	0.1	1	0	(1)(0.1) + (1)(0.1) + (-1)(1) = -0.80	$(0+0.80)^2 = 0.64$
3	0.2	0.4	1	0	(1)(0.2) + (1)(0.4) + (-1)(1) = -0.40	$(0+0.40)^2 = 0.16$
4	0.2	0.5	1	0	(1)(0.2) + (1)(0.5) + (-1)(1) = -0.30	$(0+0.30)^2 = 0.09$
5	0.4	0.5	1	1	(1)(0.4) + (1)(0.5) + (-1)(1) = -0.10	$(1+0.10)^2 = 1.21$
6	0.3	0.8	1	1	(1)(0.3) + (1)(0.8) + (-1)(1) = 0.10	$(1\text{-}0.10)^2 = 0.81$

$$\tfrac{1}{2} \sum (t_d - o_d)^2 = 1.86$$

Derivation of the gradient descent rule.

Example.

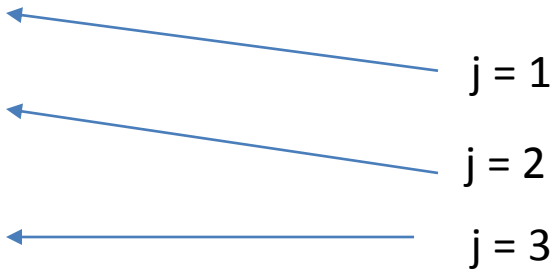
- Compute the weight correction vector $\Delta \mathbf{w}_j \Leftarrow \Delta\, \mathbf{w}_j + \eta\,(\,t_d - o_d\,)x_{d,j}$

Compute the sum:

$$(t_1 - o_1)\,\mathbf{x_1} + (t_2 - o_2)\,\mathbf{x_2} + (t_3 - o_3)\,\mathbf{x_3} + (t_4 - o_4)\,\mathbf{x_4} + (t_5 - o_5)\,\mathbf{x_5} + (t_6 - o_6)\,\mathbf{x_6} =$$

$$0.9\begin{pmatrix}0.2\\0.9\\1\end{pmatrix} + 0.8\begin{pmatrix}0.1\\0.1\\1\end{pmatrix} + 0.4\begin{pmatrix}0.2\\0.4\\1\end{pmatrix} + 0.3\begin{pmatrix}0.2\\0.5\\1\end{pmatrix} + 1.1\begin{pmatrix}0.4\\0.5\\1\end{pmatrix} + 0.9\begin{pmatrix}0.3\\0.8\\1\end{pmatrix}$$

$$= \begin{pmatrix}1.11\\2.47\\4.40\end{pmatrix}$$



Derivation of the gradient descent rule.

Example.

- Let the learning parameter η = 0.1, then $\mathbf{w} \Leftarrow \mathbf{w} + \Delta\mathbf{w}$

$$\mathbf{w}^{new} = \begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix} + 0.1 \begin{pmatrix} 1.11 \\ 2.47 \\ 4.40 \end{pmatrix} = \begin{pmatrix} 1.111 \\ 1.247 \\ -0.56 \end{pmatrix}$$

where: $\Delta\mathbf{w} = -\eta \nabla E(\mathbf{w})$

$$\tfrac{1}{2} \sum (t_d - o_d)^2 = 0.28$$

Obs. d	Fat score	Salt score		t_d	$o_d = w^t x_d$	$(t_d - o_d)^2$
	x					
1	0.2	0.9	1	1	(1.111)(0.2) + (1.247)(0.9) + (-0.56)(1) = 0.78	$(1-0.78)^2 = 0.0464$
2	0.1	0.1	1	0	(1.111)(0.1) + (1.247)(0.1) + (-0.56)(1) = -0.32	$(0+0.32)^2 = 0.1051$
3	0.2	0.4	1	0	(1.111)(0.2) + (1.247)(0.4) + (-0.56)(1) = 0.16	$(0-0.16)^2 = 0.0259$
4	0.2	0.5	1	0	(1.111)(0.2) + (1.247)(0.5) + (-0.56)(1) = 0.29	$(0-0.29)^2 = 0.0816$
5	0.4	0.5	1	1	(1.111)(0.4) + (1.247)(0.5) + (-0.56)(1) = 0.51	$(1-0.51)^2 = 0.2422$
6	0.3	0.8	1	1	(1.111)(0.3) + (1.247)(0.8) + (-0.56)(1) = 0.77	$(1-0.77)^2 = 0.0525$

Gradient descent and Delta Rule

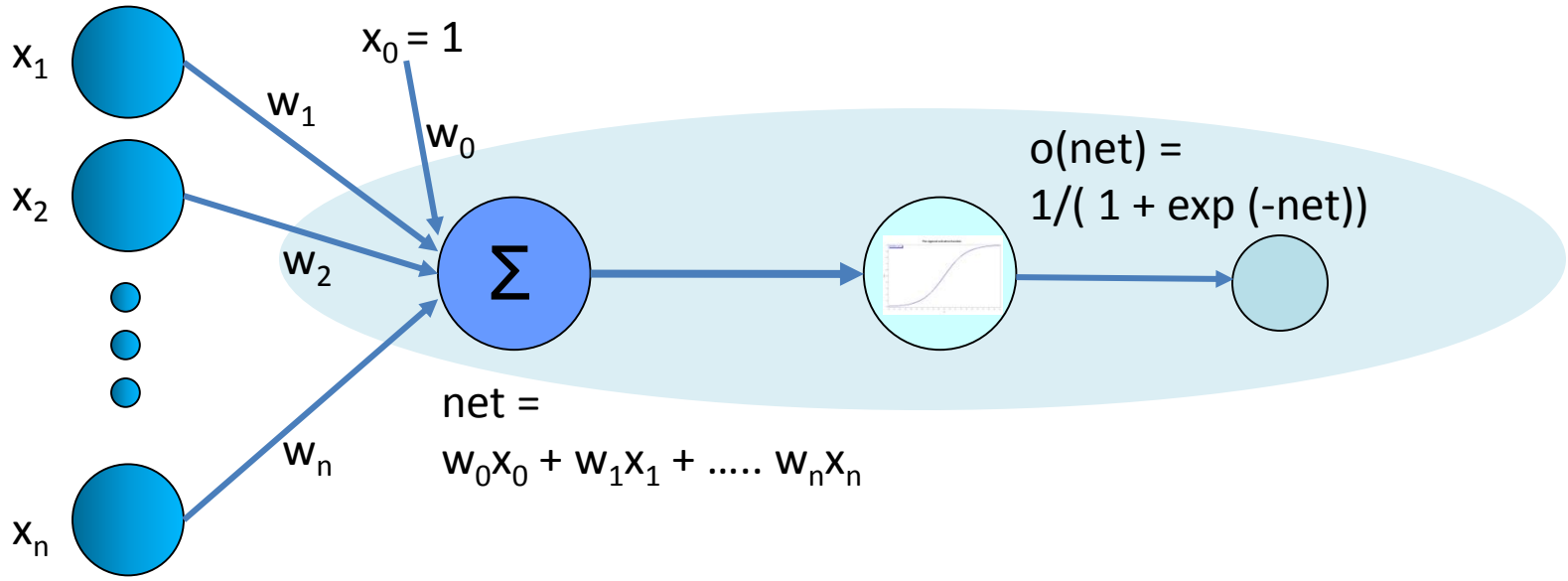
Stochastic approximation to gradient descent.

- Gradient descent can be applied for learning where
  - The hypothesis space contains continuously parameterized hypothesis.
  - The error can be differentiated with respect to the hypothesis parameters.
- Some practical difficulties in gradient descent learning:
  - Slow convergence to a local minimum
  - No guaranteed convergence to a global minimum.
- To alleviate these difficulties, a common variation of gradient descent is used: **incremental gradient descent** or alternatively **stochastic gradient descent**.
- Instead of updating the weights with the sum over all training examples, the stochastic gradient is to approximate the gradient descent search by updating the weights incrementally, following the calculation of the error for each individual sample.

10. Multilayer networks and the backpropagation algorithm

A differentiable threshold unit.

- Multiple layers of cascaded linear units still produce only linear function.
- Networks capable of representing highly nonlinear function are preferred.
- A unit giving a nonlinear function of its inputs as output is the **sigmoid unit**:



Multilayer networks and the back-propagation algorithm

A differentiable threshold unit.

- The sigmoid unit computes its output o as

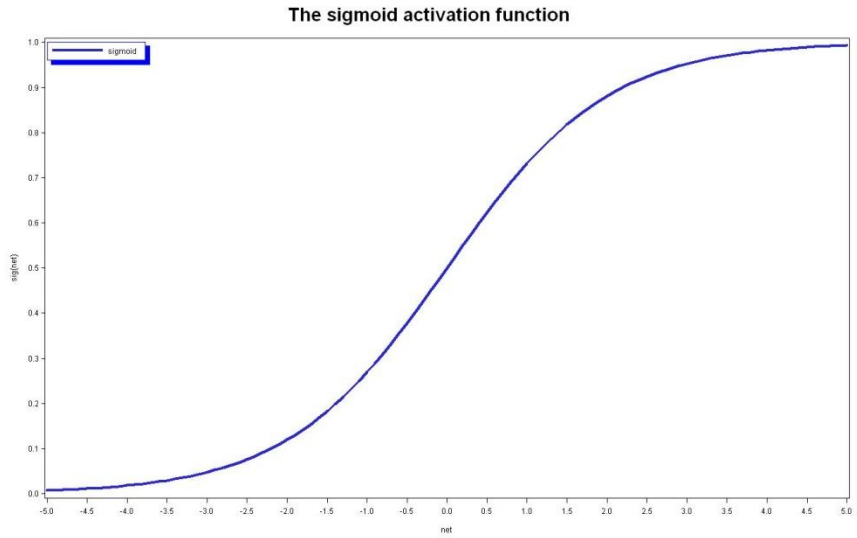
  $o = \sigma(\mathbf{w}^T\mathbf{x})$, where σ is the sigmoid activation function:

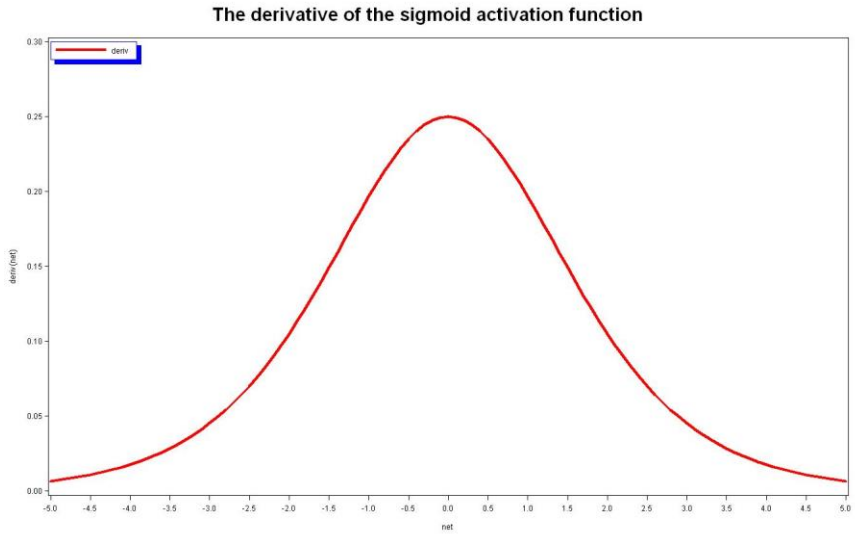
  $\sigma(y) = 1/(1 + \exp(-y)) = 1/(1 + e^{-y})$

- The derivative of the sigmoid function is

  $\sigma'(y) = \sigma(y)(1 - \sigma(y))$

- The function is often referred to as the **squashing function**.





Multilayer networks and the back-propagation algorithm

A differentiable threshold unit.

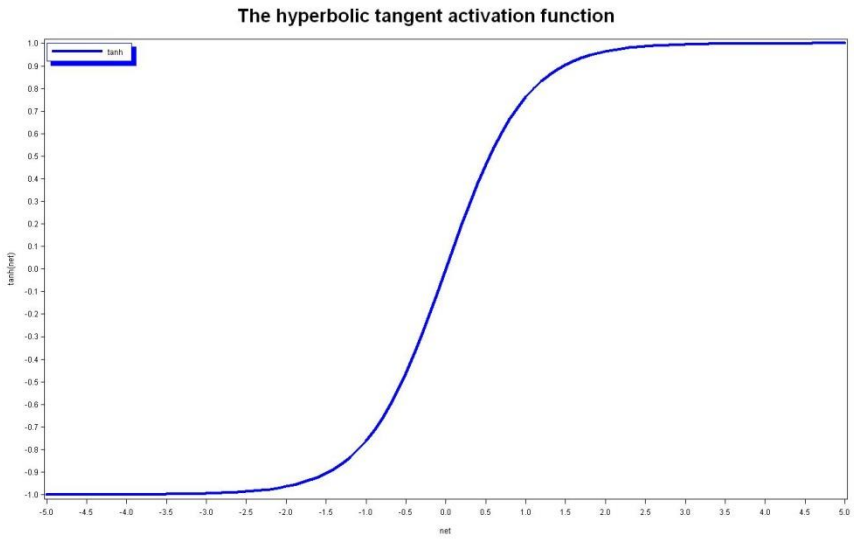
- The function tanh(y) is sometimes used in place of the sigmoid activation function:

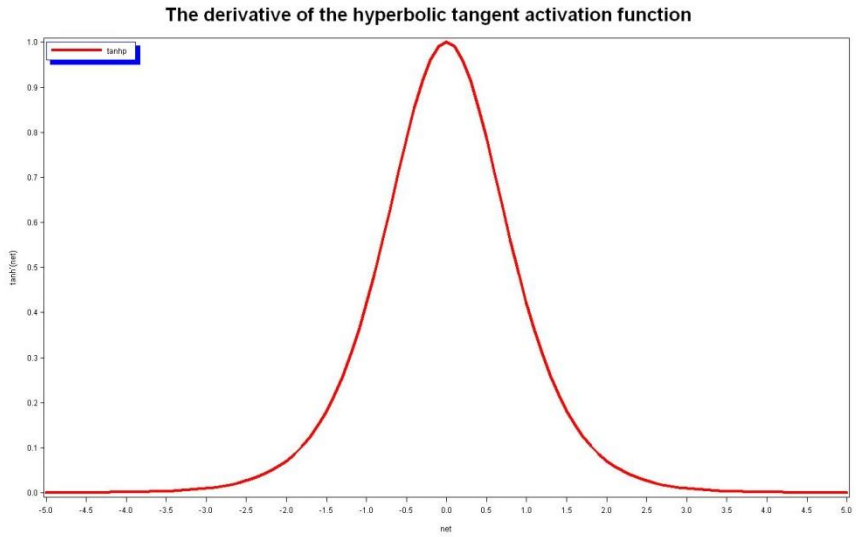
  o = tanh($\mathbf{w}^T\mathbf{x}$) , where tanh(y) is the **hyperbolic tangent** activation function:

  $\tanh(y) = (e^y - e^{-y})/(e^y + e^{-y}) = 2\,\sigma(2y) - 1$

- The derivative of tanh(y) is

  $\tanh'(y) = 1 - \tanh(y)^2$



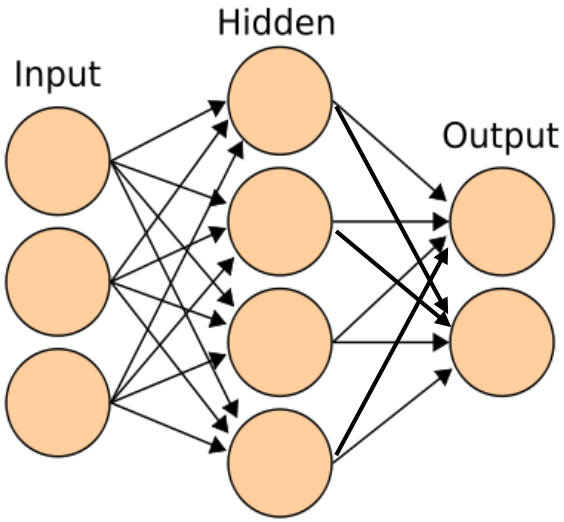


- σ(y) and tanh(y) are examples of activation function/gain function/transfer function/squashing function.

Multilayer networks and the back-propagation algorithm

The back-propagation algorithm.

Each training example is denoted as (**z**,**d**) where **z** is the vector of input values, and **d** is the vector of target network values.



- η is the learning rate (e.g. 0.5)
- I is the number of input units
- J is the number of hidden units
- K is the number of output units
- **W** is the weight matrix for connections from hidden units to output units, with K rows and J columns
- **V** is the weight matrix for connections from input units to hidden units, with J rows and I columns.
- **y** is a vector of hidden unit activations, J rows.
- **o** is a vector of output unit activations, K rows.

Multilayer networks and the back-propagation algorithm

The back-propagation algorithm
(Stochastic gradient descent with sigmoid activation function).

- Step 1. Choose E_{max}, $\eta > 0$. Set $q \Leftarrow 1$, $p \Leftarrow 1$, $E \Leftarrow 0$. Initialize **W**, **V** randomly.
- Step 2. Present the input data: $\mathbf{z} \Leftarrow \mathbf{z}_p$, $\mathbf{d} \Leftarrow \mathbf{d}_p$ and compute

  $y_j \Leftarrow f(\mathbf{v}_j^T \mathbf{z})$ for j = 1, 2, …. J and

  $o_k \Leftarrow f(\mathbf{w}_k^T \mathbf{y})$ for k = 1, 2, …. K
- Step 3. Compute the error: $E \Leftarrow \frac{1}{2} (\mathbf{d}_k - \mathbf{o}_k)^2 + E$ for k = 1,2, … K
- Step 4. Compute error signal vectors δ_o and δ_y for output and hidden layer units:

  $\delta_{ok} = (\mathbf{d}_k - \mathbf{o}_k)(1 - \mathbf{o}_k)\mathbf{o}_k$ for k = 1, 2, … K

  $\delta_{yj} = \mathbf{y}_j(1 - \mathbf{y}_j) \sum_{k=1}^{K} \delta_{ok} \mathbf{w}_{kj}$ for j = 1, 2, … J
- Step 5. Update output layer and hidden layer weights:

  $\mathbf{w}_{kj} \Leftarrow \mathbf{w}_{kj} + \eta\, \delta_{ok}\, \mathbf{y}_j$ for k = 1, 2, … K and j = 1, 2, … J

  $\mathbf{v}_{ji} \Leftarrow \mathbf{v}_{ji} + \eta\, \delta_{yj}\, \mathbf{z}_i$ for j = 1, 2, … J and i = 1, 2, … I
- Step 6. If p < P then $p \Leftarrow p + 1$, $q \Leftarrow q + 1$, go to Step 2.
- Step 7. If E < Emax, stop. Output W, V, q, and E. Else $E \Leftarrow 0$,$p \Leftarrow 1$, go to Step 2.

Multilayer networks and the back-propagation algorithm

The back-propagation algorithm
(Stochastic gradient descent with sigmoid activation function).

- Step 1. Choose E_{max}, $\eta > 0$. Set $q \Leftarrow 1$, $p \Leftarrow 1$, $E \Leftarrow 0$. Initialize **W**, **V** randomly.

  - E_{max} is a stopping condition, stop training if $E < E_{max}$
  - $\eta > 0$ is the learning rate
  - q is a counter: total number of weights updates
  - p is an index, data sample d_p is presented to the network at this iteration.
  - E is the accumulated error at this iteration.

- Step 2. Present the input data: $\mathbf{z} \Leftarrow \mathbf{z}_p$, $\mathbf{d} \Leftarrow \mathbf{d}_p$ and compute

  $y_j \Leftarrow f(\mathbf{v}_j^T \mathbf{z})$ for j = 1, 2, …. J and

  $o_k \Leftarrow f(\mathbf{w}_k^T \mathbf{y})$ for k = 1, 2, …. K

  - d is the target output (target features) and z is the input (descriptive features)
  - v_j is the vector weight from the input to hidden unit j
  - y_j is the activation value at hidden unit j
  - w_k is the vector weight from the hidden units to output unit k
  - o_k is the activation value at output unit j

Multilayer networks and the back-propagation algorithm

The back-propagation algorithm (continued).

- Step 3. Compute the error: $E \Leftarrow \frac{1}{2} (\mathbf{d}_k - \mathbf{o}_k)^2 + E$ for k = 1,2, … K

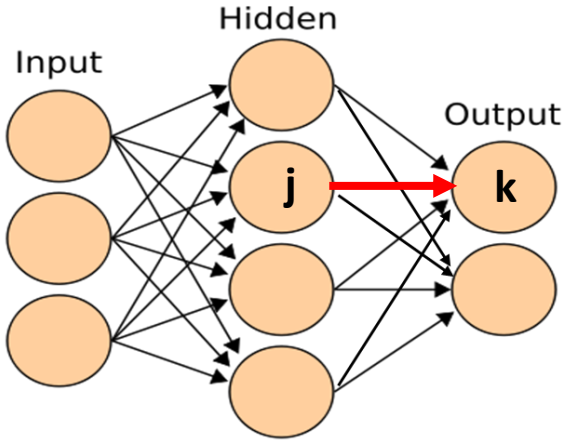
  - The difference between the target output and predicted output at output unit k is error $d_k - o_k$.
  - Accumulate sum of squared errors in E.

- Step 4. Compute error signal vectors δ_o and δ_y for output and hidden layer units:

$$\delta_{ok} = (\mathbf{d}_k - \mathbf{o}_k)(1 - \mathbf{o}_k)\mathbf{o}_k \text{ for } k = 1, 2, \ldots K$$

  - For the output unit, since $o_k \Leftarrow f(\mathbf{w}_k^T\mathbf{y}) = \sigma(\mathbf{w}_k^T\mathbf{y})$, we want to compute the partial derivative of E with respect to $w_{kj,}$ where w_{kj} is the weight from hidden unit j to output unit k

$$\begin{aligned}
\partial E/\partial \mathbf{w}_{kj} &= \partial/\partial \mathbf{w}_{kj} \tfrac{1}{2} (\mathbf{d}_k - \mathbf{o}_k)^2 \\
&= (\mathbf{d}_k - \mathbf{o}_k)\, \partial/\partial \mathbf{w}_{kj} (\mathbf{d}_k - \mathbf{o}_k) \\
&= (-1)(\mathbf{d}_k - \mathbf{o}_k)(1 - \mathbf{o}_k)\mathbf{o}_k\ \partial/\partial \mathbf{w}_{kj} (\mathbf{w}_k^T\mathbf{y}) \\
&= (-1)(\mathbf{d}_k - \mathbf{o}_k)(1 - \mathbf{o}_k)\mathbf{o}_k\ \mathbf{y}_j \\
&= -\delta_{ok}\, \mathbf{y}_j
\end{aligned}$$



Multilayer networks and the back-propagation algorithm

The back-propagation algorithm (continued).

- Step 4. Compute error signal vectors δ_o and δ_y for output and hidden layer units:

$$\delta_{yj} = \mathbf{y}_j(1 - \mathbf{y}_j) \sum_{i=1}^{K} \delta_{ok}\mathbf{w}_{kj} \text{ for } j = 1, 2, \ldots J$$

  - For the hidden unit j, $y_j \Leftarrow f(\mathbf{v}_j^T\mathbf{z}) = \sigma(\mathbf{v}_j^T\mathbf{z})$, we compute the partial derivative of E with respect to $v_{ji,}$ where v_{ji} is the weight from input unit i to output unit j

$$\partial E/\partial \mathbf{v}_{ji} = \partial/\partial \mathbf{v}_{ji} \tfrac{1}{2} (\mathbf{d}_k - \mathbf{o}_k)^2 \text{ for } k = 1, 2 \ldots K$$

$$= \tfrac{1}{2} \sum_{i=1}^{K} \partial/\partial \mathbf{v}_{ji} (\mathbf{d}_k - \mathbf{o}_k)^2$$

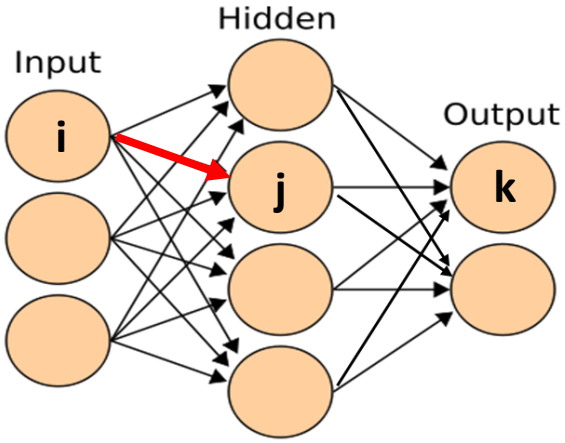
$$= \sum_{i=1}^{K} (\mathbf{d}_k - \mathbf{o}_k) \partial/\partial \mathbf{v}_{kj} (\mathbf{d}_k - \mathbf{o}_k)$$

$$= (-1) \sum_{i=1}^{K} (\mathbf{d}_k - \mathbf{o}_k)(1 - \mathbf{o}_k)\mathbf{o}_k \partial/\partial \mathbf{v}_{ji} (\mathbf{w}_k^T\mathbf{y})$$

$$= (-1) \sum_{i=1}^{K} \delta_{ok} \partial/\partial \mathbf{v}_{ji} (\mathbf{w}_k^T\mathbf{y})$$

$$= (-1) (1 - \mathbf{y}_j)\mathbf{y}_j \sum_{i=1}^{K} \delta_{ok}\mathbf{w}_{kj} \partial/\partial \mathbf{v}_{ji} (\mathbf{v}_j^T\mathbf{z})$$

$$= -\delta_{yj} z_i$$



Recall: $\delta_{ok} = (\mathbf{d}_k - \mathbf{o}_k)(1 - \mathbf{o}_k)\mathbf{o}_k$

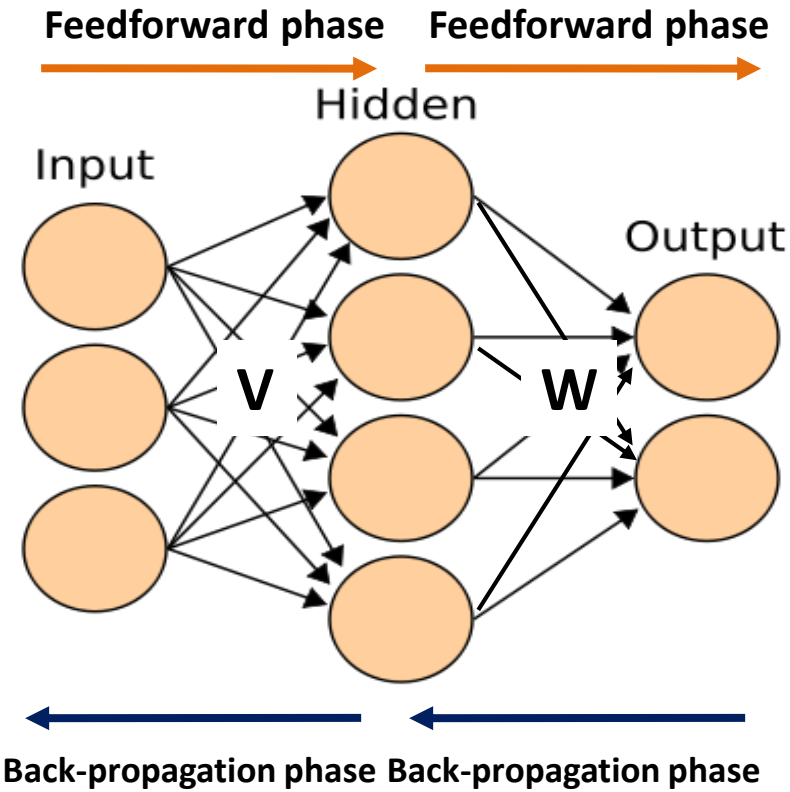
Multilayer networks and the back-propagation algorithm

The back-propagation algorithm (continued).

- Step 5. Update output layer and hidden layer weights:

$$\mathbf{w}_{kj} \Leftarrow \mathbf{w}_{kj} + \eta\, \delta_{ok}\, \mathbf{y}_j \text{ for } k = 1, 2, \ldots K \text{ and } j = 1, 2, \ldots J$$

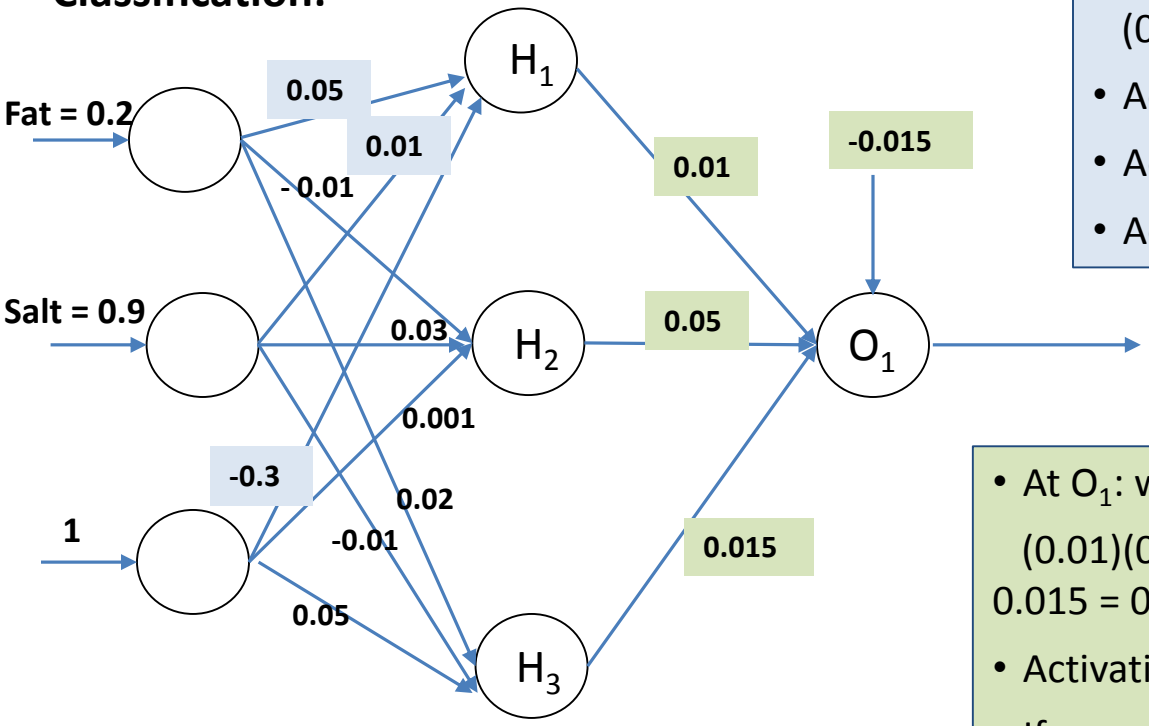
$$\mathbf{v}_{ji} \Leftarrow \mathbf{v}_{ji} + \eta\, \delta_{yj}\, \mathbf{z}_i \text{ for } j = 1, 2, \ldots J \text{ and } i = 1, 2, \ldots I$$



Multilayer networks and the back-propagation algorithm

The back-propagation algorithm (continued).

Classification:

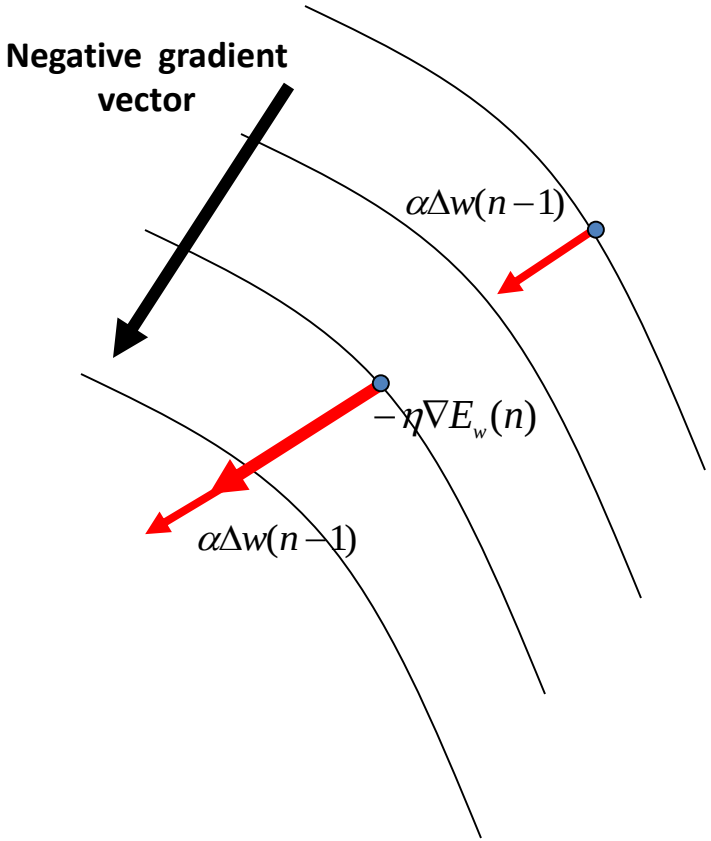


- At H_1: $w_1x_1 + w_2x_2 + w_3x_3$ = $(0.05)(0.2) + (0.01)(0.9) – 0.3 = -0.281$
- Activation value H_1: $1/(1 + e^{0.281}) = 0.430$
- Activation value H_2: $1/(1 + e^{-0.026}) = 0.506$
- Activation value H_3: $1/(1 + e^{-0.045}) = 0.511$

- At O_1: $w_1x_1 + w_2x_2 + w_3x_3$ + bias = $(0.01)(0.430) + (0.05)(0.506) + (0.015)(0.511) – 0.015 = 0.0223$
- Activation value O_1: $1/(1 + e^{-0.0223}) = 0.506$
- If we use a cut-off of 0.5, we classify this sample as Class 1 because 0.506 > 1

Multilayer networks and the back-propagation algorithm

Modification of the back-propagation algorithm: adding momentum



$$\Delta w(n) = -\eta\nabla E_w(n) + \alpha\Delta w(n-1)$$

Figure for monotonic error surfaces contour map E(**w**) in the weight space

α is called the momentum constant, typically of value 0.1 to 0.6

Result: accelerated motion on monotonic error surfaces

Multilayer networks and the back-propagation algorithm

Modification of the back-propagation algorithm: adding momentum

$$\Delta w(n) = -\eta \nabla E_w(n) + \alpha \Delta w(n-1)$$

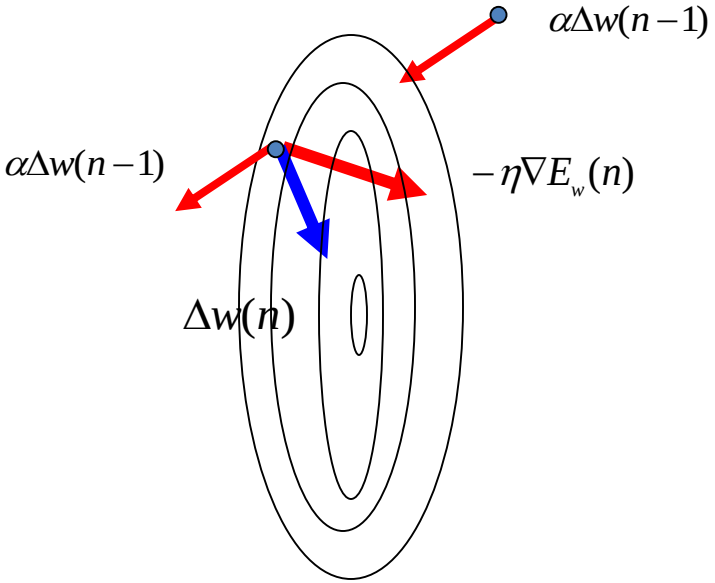


Fig for surface with narrow error valleys

contour map E(**w**) in the weight space

Result: accelerated convergence towards minimum when gradient changes sign from step to step,

thus avoiding futile jumps over narrow valleys

11. Remarks on the backpropagation algorithm

Convergence and local minima

- The BP algorithm implements a gradient descent search through the space of possible weights.
- It iteratively reduces the error E between the training sample target values and the network outputs.
- The error surface may contain many local minima and BP may be trapped in a local minimum.
- BP is a very effective function approximation method.
- A local minimum with respect to one weight will not necessarily be a local minimum with respect to other weights: more weights might provide more “escape routes” for gradient descent.
- More local minima are expected to exist in the region of the weight space representing more complex nonlinear function. It is hoped that when the network weights reach this region, they are already close enough to the global minimum.

Remarks on the backpropagation algorithm

Avoiding local minima

- Add a momentum term to the weight up-date rule.
- Use stochastic gradient instead of true gradient descent:
  - Stochastic approximation descents a different error surface for each training example relying on the average of these to approximate the gradient with respect to full training set.
  - Patterns can be randomized within each training cycle rather then submitted for training as a fixed sequence.
- Train multiple networks using the same data:
  - Initialize each network with different random weights.
  - Select the network with the best performance over a separate validation set, or
  - Form a committee of networks whose output is the (possibly weighted) average of the individual network outputs.

Remarks on the backpropagation algorithm

Representational power of feedforward networks

Set of functions that can be represented by feedforward networks:

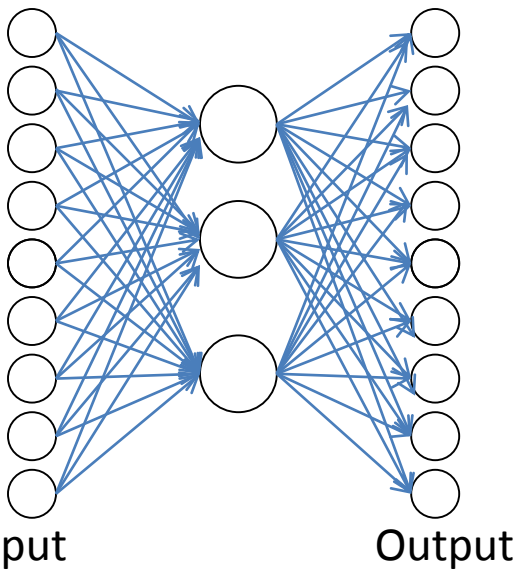
- Boolean function:
  - every Boolean function can be represented exactly by some two layers of units.
  - The number of hidden units needed may be equal to the number of network inputs in the worst case.
- Continuous function:
  - every bounded continuous function can be approximated with arbitrarily small error by a network with two layers of units.
  - Hidden layer with sigmoid units and output layer with linear units.

Remarks on the backpropagation algorithm

Hidden layer representations

- BP is able to discover useful intermediate representations at the hidden unit layers.
- Training samples constrain only the network inputs and outputs: the weight tuning procedure is free to set weights that define the most effective hidden unit representations to minimize E(w).
- BP can capture properties of the input instances that are relevant to learning the target function.
- Example:

Autoencoder network

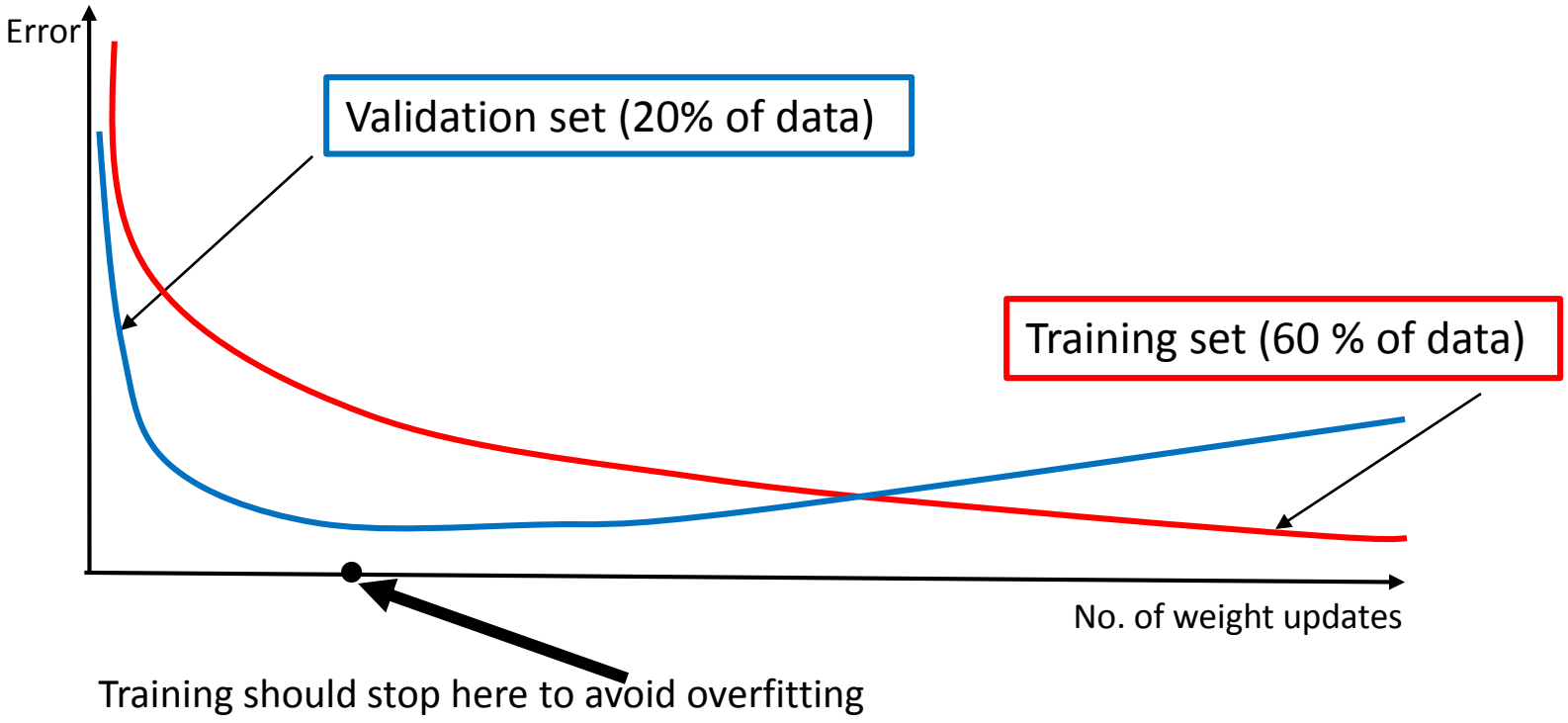


Input		Hidden values				Output
10000000	⇨	1	0	0	⇨	10000000
01000000	⇨	0	1	1	⇨	01000000
00100000	⇨	0	1	0	⇨	00100000
00010000	⇨	1	1	1	⇨	00010000
00001000	⇨	0	0	0	⇨	00001000
00000100	⇨	0	0	1	⇨	00000100
00000010	⇨	1	0	1	⇨	00000010
00000001	⇨	1	1	0	⇨	00000001

Remarks on the backpropagation algorithm

Generalization, overfitting and stopping criteria

- To improve generalization and reduce overfitting, it is recommended that the available data samples be split into three sets: training set, cross validation set and test set. For example: 60%, 20%, 20%.
- Network training is terminated when the network weights produce the lowest total error for the samples in the cross validation set.



12. Advanced topics in artificial neural networks

Alternative error functions

- Adding a penalty term for weight magnitude:

$$E(\mathbf{w}) = \tfrac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} \mathbf{w}_{ji}^2$$

to give network connections tendency to decay to zero.

- Non-useful connections will be removed during training.

- The penalty term penalizes discourages use of large weights: one large weight costs much more than many small ones.

- A different penalty term:

$$E(\mathbf{w}) = \tfrac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} \mathbf{w}_{ji}^2/(1 + \mathbf{w}_{ji}^2)$$

where γ is a positive parameter.

Alternative error functions

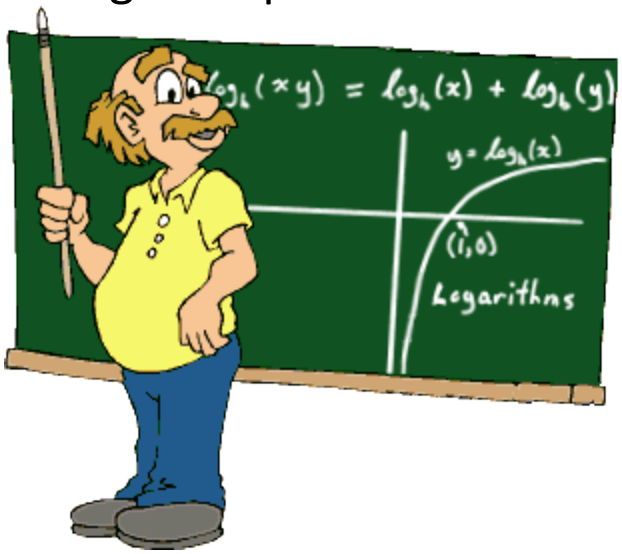
- Minimizing the cross-entropy error function:

$$E(\mathbf{w}) = - \sum_{d \in D} t_d \log o_d + (1 - t_d) \log (1 - o_d)$$

for a network with one output unit with only Boolean target values of 0 or 1.

- Here o_d is the probability estimate output by the network for training example d and t_d is either 0 or 1 target value for this training example.

- Use sigmoid output units.



12. Advanced topics in artificial neural networks

Alternative error minimization procedures

- Back-propagation/Gradient descent is well known to be slow.
- Other optimization methods may be applied to minimize the error function:
  - Line search method: once a direction for update has been specified, the update distance is chosen by finding the minimum of the error function along this direction. At $\mathbf{w}^k$, given a direction $\mathbf{d}$, find $\eta > 0$ such that

    $E(\mathbf{w}^k + \eta\mathbf{d})$ is minimized.
  - Conjugate gradient method.
  - Quasi-Newton method.

Error minimization as an optimization problem

- Using the Taylor series we may expand the error function E(w) about the current point on the error surface $\mathbf{w}^k$:

$$E(\mathbf{w}^k + \mathbf{d}) = E(\mathbf{w}^k) + (\mathbf{g}^k)^T \mathbf{d} + \tfrac{1}{2}\, \mathbf{d}\, \mathbf{H}^k \mathbf{d} + \text{(third and higher order term)}$$

where $\mathbf{g}^k$ is the gradient vector defined by

$$\mathbf{g}^k = \partial/\partial \mathbf{w}_j\, E(\mathbf{w}) \quad \text{computed at } \mathbf{w}^k$$

and $\mathbf{H}^k$ is the Hessian matrix

$$\mathbf{H}^k = \partial^2/\partial \mathbf{w}_j^2\, E(\mathbf{w}) \quad \text{computed at } \mathbf{w}^k$$

- In the steepest descent method: $d = -\eta\, \mathbf{g}^k$
- We can use an quadratic approximation of the error function:

$$E(\mathbf{w}^k + \mathbf{d}) \approx E(\mathbf{w}^k) + (\mathbf{g}^k)^T \mathbf{d} + \tfrac{1}{2}\, \mathbf{d}\, \mathbf{H}^k \mathbf{d}$$

- Taking the derivative of the quadratic function and setting it to 0, we obtain

$$\mathbf{d} = -(\mathbf{H}^k)^{-1} \mathbf{g}^k$$

- Newton's method: $\mathbf{w}^{k+1} = \mathbf{w}^k + \eta \mathbf{d}$

12. Advanced topics in artificial neural networks

Error minimization as an optimization problem

- Newton's method: $\mathbf{w}^{k+1} = \mathbf{w}^k + \eta\mathbf{d}$ where $\mathbf{d} = -(\mathbf{H}^k)^{-1}\mathbf{g}^k$
- Drawback of this approach:
  - It requires the calculation of the inverse of the Hessian matrix which can be computationally very expensive.
  - The Hessian has to be non-singular, which may not be true for the given data.
  - When the error function is nonquadratic, there is no guarantee that it will converge.
- An alternative to the Newton's method is the **Quasi-Newton** method:
  - It approximates the inverse of the Hessian matrix of the error function.
  - At each iteration, this matrix is updated.
  - It can be expected to converge faster than the gradient descent/backpropagation method and the Newton's method.

12. Advanced topics in artificial neural networks

Dynamically modifying network structure

- Three possible ways:
  - Start with very small network, add hidden units as needed to reduce the error.
  - Start with very large network, remove hidden units to improve generalization.
  - Combination of growing and removing hidden units.

12. Advanced topics in artificial neural networks

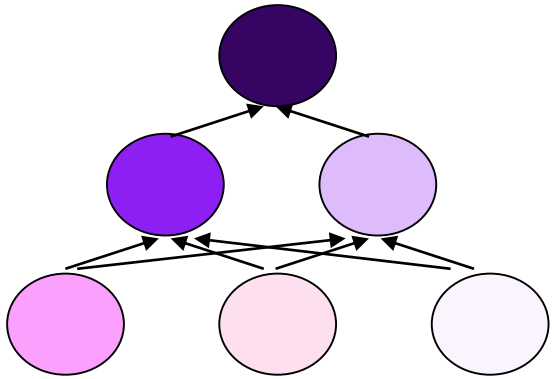
Rule extraction from neural networks with one hidden layer

- It may be interesting to find out exactly how the neural network makes a decision.
- A decomposition approach to neural network rule extraction:

1. Train and prune a network with a single hidden layer
2. Cluster the hidden unit activation values:
   - Original activation values lies in [-1,1]
   - Clustering implies dividing this interval into subintervals, for example [-1,-0.8), [-0.8,0.5), [0.5,1]
   - An algorithm is needed to ensure the network does not lose its accuracy
3. Generate classification rules in terms of clustered activation values
4. Generate rules which explain the clustered activation values in terms of the input data attributes
5. Merge the two sets of rules

Rule extraction from neural networks with one hidden layer

- A decomposition approach to neural network rule extraction:

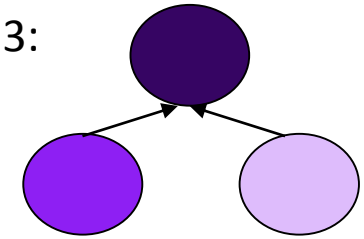


Output layer

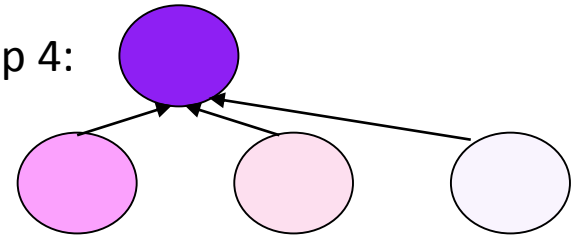
Hidden layer

Input layer

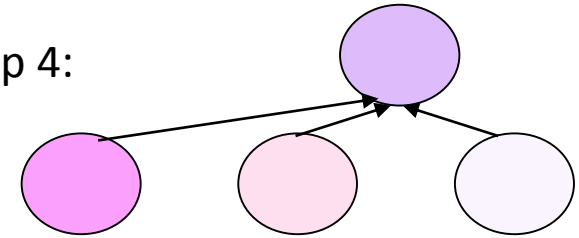
Step 3:



Step 4:



Step 4:



Rule extraction from neural networks with one hidden layer

- Example: Iris data
  - 150 instances.
  - 4 continuous attributes: sepal length, sepal width, petal length, petal width.
  - Three different iris flowers:

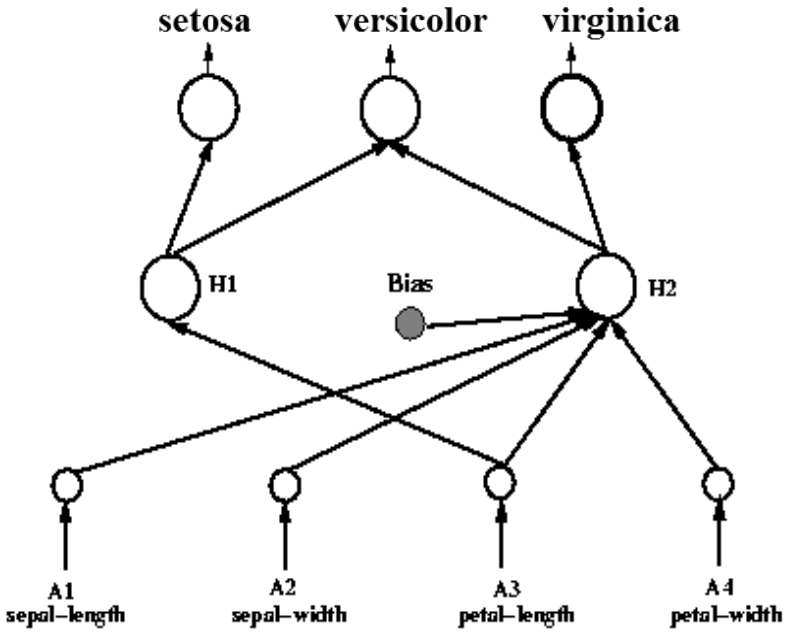
setosa

versicolor

virginica

Rule extraction from neural networks with one hidden layer

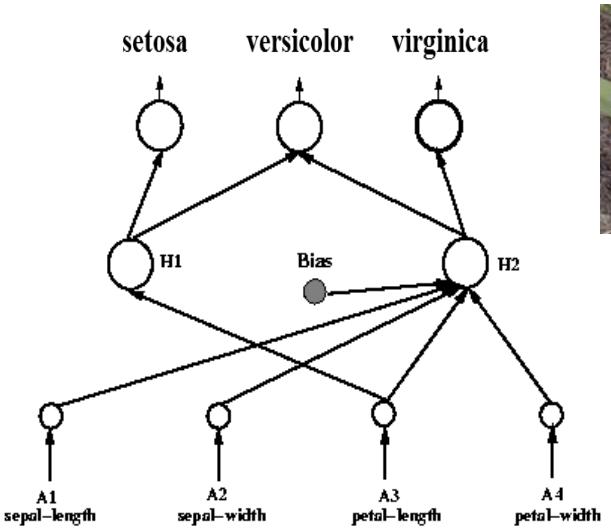
- A pruned neural network for the iris data:

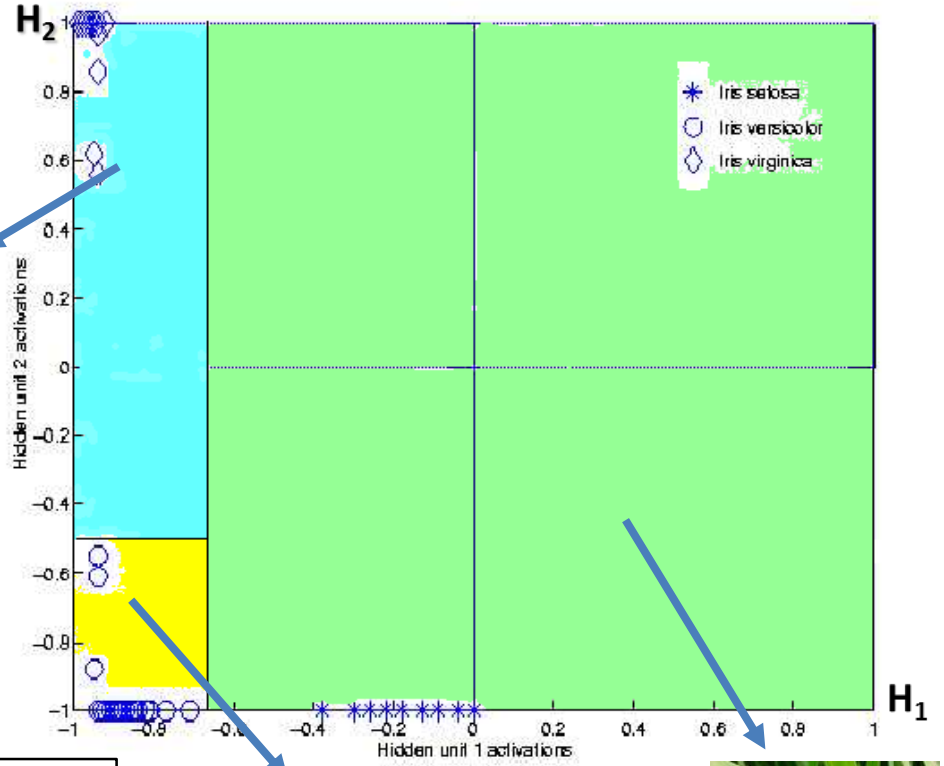


  - Three class problem: 3 output units.
  - Four input attributes: 4 input units + 1 for bias.
  - The network has only 2 hidden units and 10 connections after pruning.
  - It correctly classifies all but one training pattern.
  - 2-dimensional plot of the activation values.

Rule extraction from neural networks with one hidden layer

- A pruned neural network for the iris data and plot of activation values:





Rule in terms of the hidden unit activations:

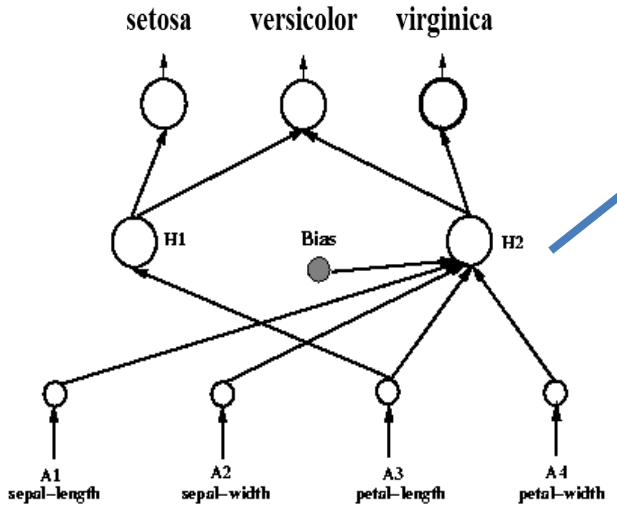
If $H_1 > -0.7$: Iris setosa

Else if $H_2 <= -0.55$: Iris versicolor

Else: Iris virginica

Rule extraction from neural networks with one hidden layer

- A pruned neural network for the iris data and extracted rules:



Step 3

Rule in terms of the hidden unit activations:

If $H_1 > -0.7$: Iris setosa

Else if $H_2 <= -0.55$: Iris versicolor

Else: Iris virginica

Step 4

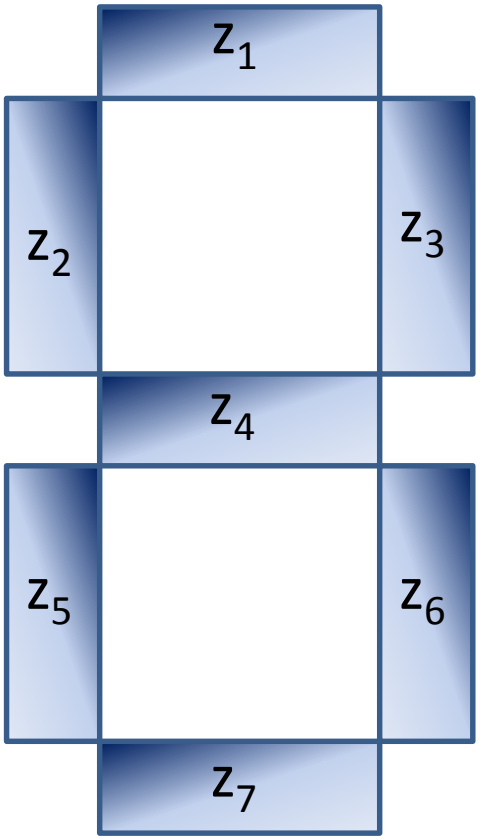
If **petal length > 2.23**, then Iris setosa.

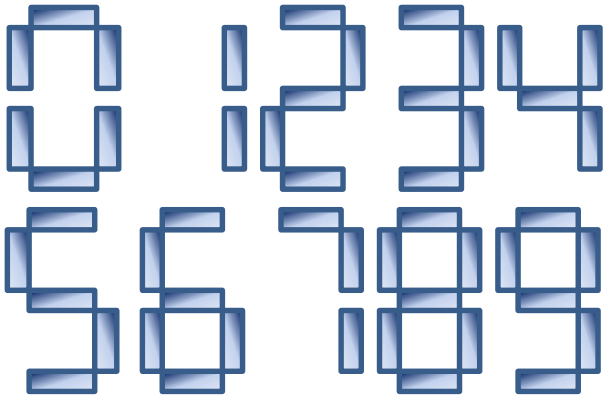
Else if

3.57 petal length + 3.56 petal width - sepal length - 1.57 sepal width > 12.63, then Iris versicolor.

Else Iris virginica.

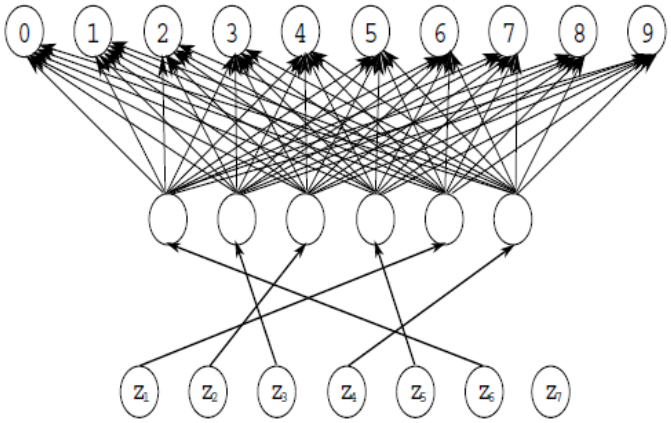
Rule extraction from NN: An LED (Light Emitting Diode) device and digits 0, 1, .. 9

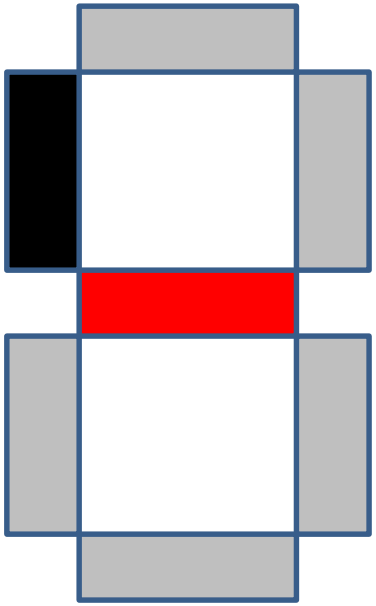




Processed data

z_1	z_2	z_3	z_4	z_5	z_6	z_7	Digit
1	1	1	0	1	1	1	0
0	0	1	0	0	1	0	1
1	0	1	1	1	0	1	2
1	0	1	1	0	1	1	3
0	1	1	1	0	1	0	4
1	1	0	1	0	1	1	5
1	1	0	1	1	1	1	6
1	0	1	0	0	1	0	7
1	1	1	1	1	1	1	8
1	1	1	1	0	1	1	9



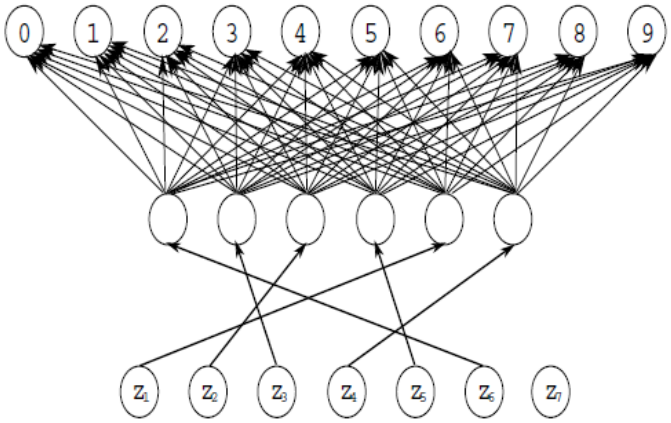


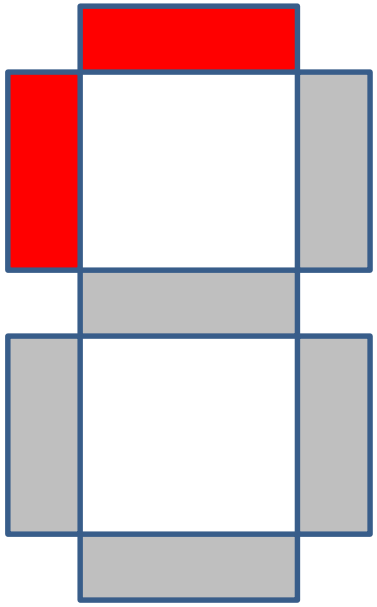
= 0

Must be on

Must be off

Doesn’t matter



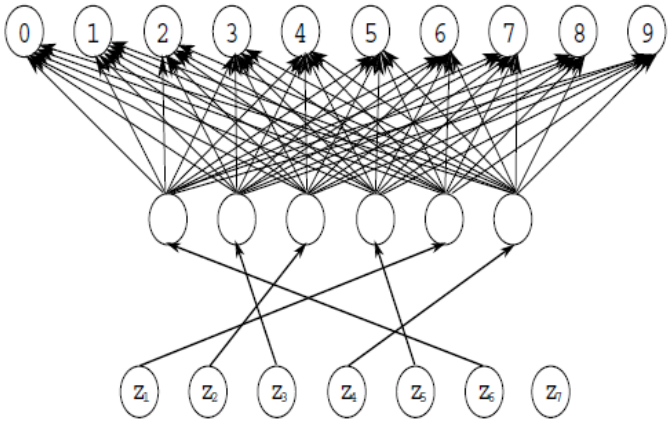


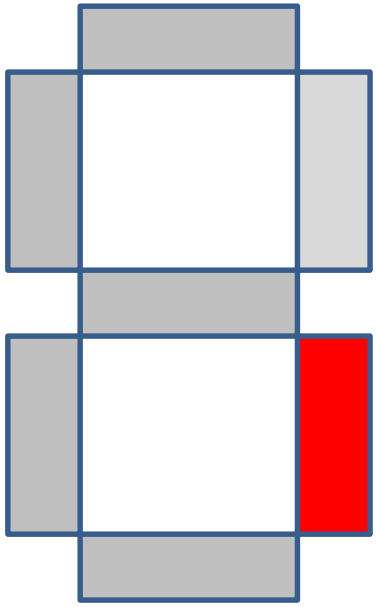
= 1

Must be on

Must be off

Doesn't matter





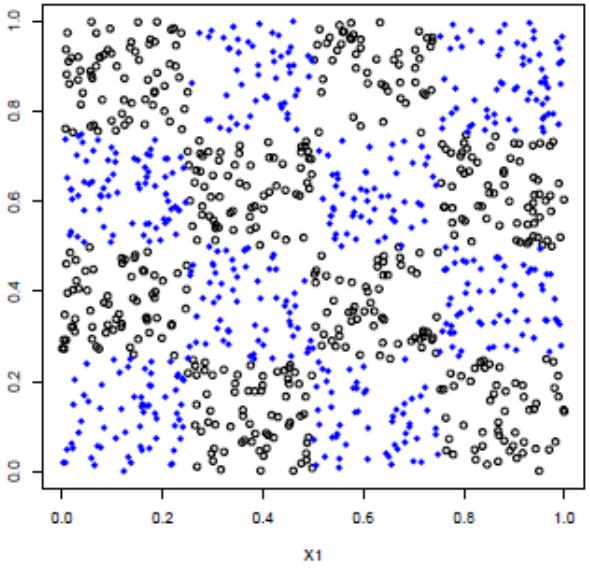
= 2

Must be on

Must be off

Doesn't matter

12. Advanced topics in artificial neural networks

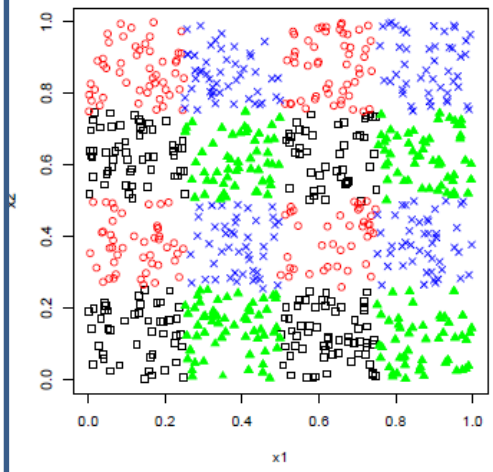


(a) Chess data set.

- If 3 of $\{\overline{I}_9, I_{14}, \overline{I}_{19}, I_{29}, \overline{I}_{34}, I_{39}\}$ are equal to $+1$, then Class 1,
- else Class 0.

I_9	I_{14}	I_{19}	x_1	I_{29}	I_{34}	I_{39}	x_2
0	0	0	$x_1 < 0.25$	0	0	0	$x_2 < 0.25$
0	0	1	$0.25 \leq x_1 < 0.5$	0	0	1	$0.25 \leq x_2 < 0.5$
0	1	1	$0.5 \leq x_1 < 0.75$	0	1	1	$0.5 \leq x_2 < 0.75$
1	1	1	$0.75 \leq x_1$	1	1	1	$0.75 \leq x_2$

More complex patterns:



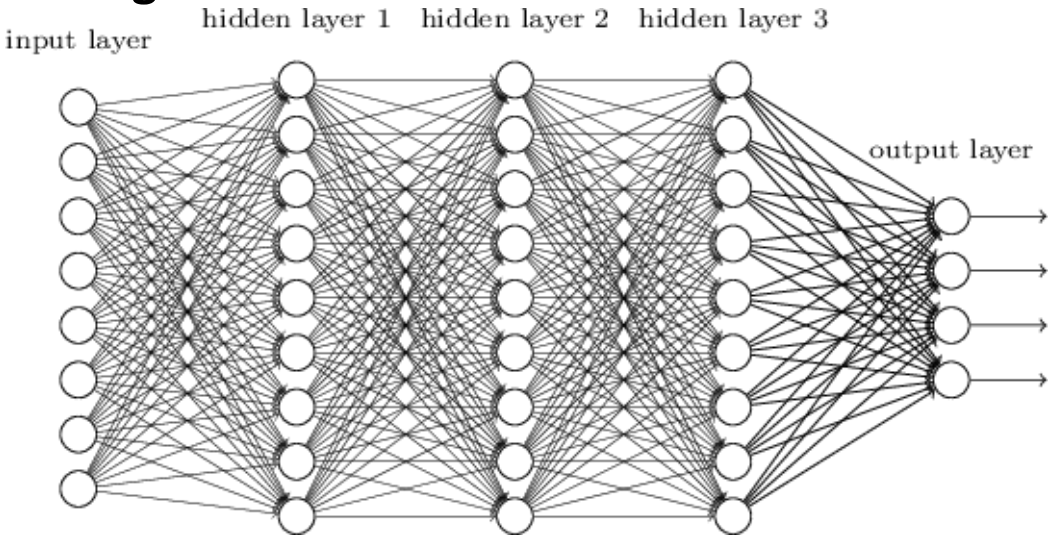
(b) Four-by-four data set.

Other applications of NN rule extraction:

- Bankruptcy prediction.
- Distinguishing between organisations that adopt IT from those that do not.
- Predicting gene sequences.
- Predicting protein secondary structures.
- Analysis of marketing survey data.

12. Advanced topics in artificial neural networks

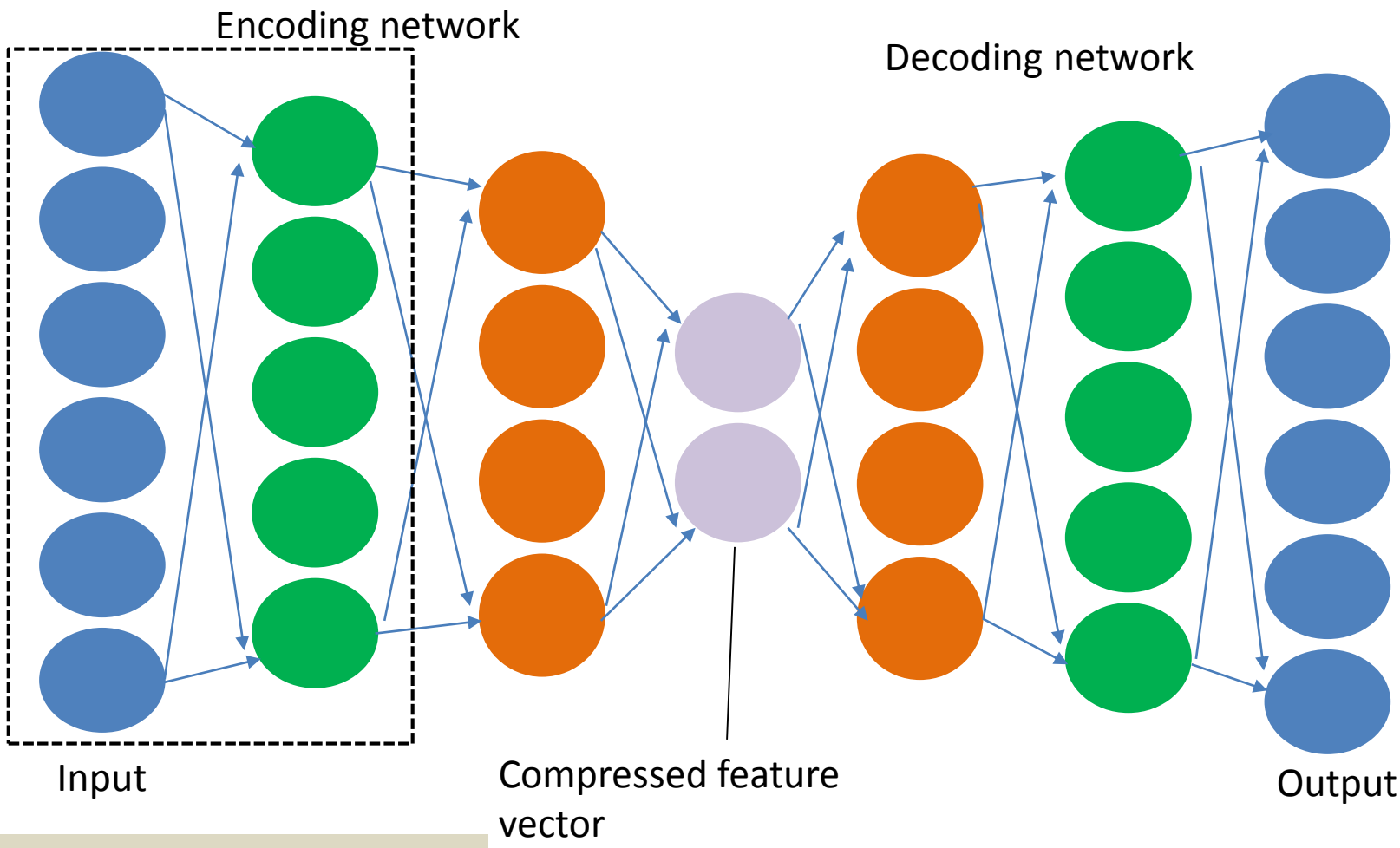
Deep learning neural networks:



- Many layers between the input and output layers
- Difficulty with training using error back-propagation: vanishing gradient problem.
- The gradient diminishes as the error is back-propagated.
- Weights near the input train very slowly.
- Possible remedy: pretrain some weights using unsupervised learning, stack auto encoder neural networks such that each layer of the learns an encoding of the layer below it.

12. Advanced topics in artificial neural networks

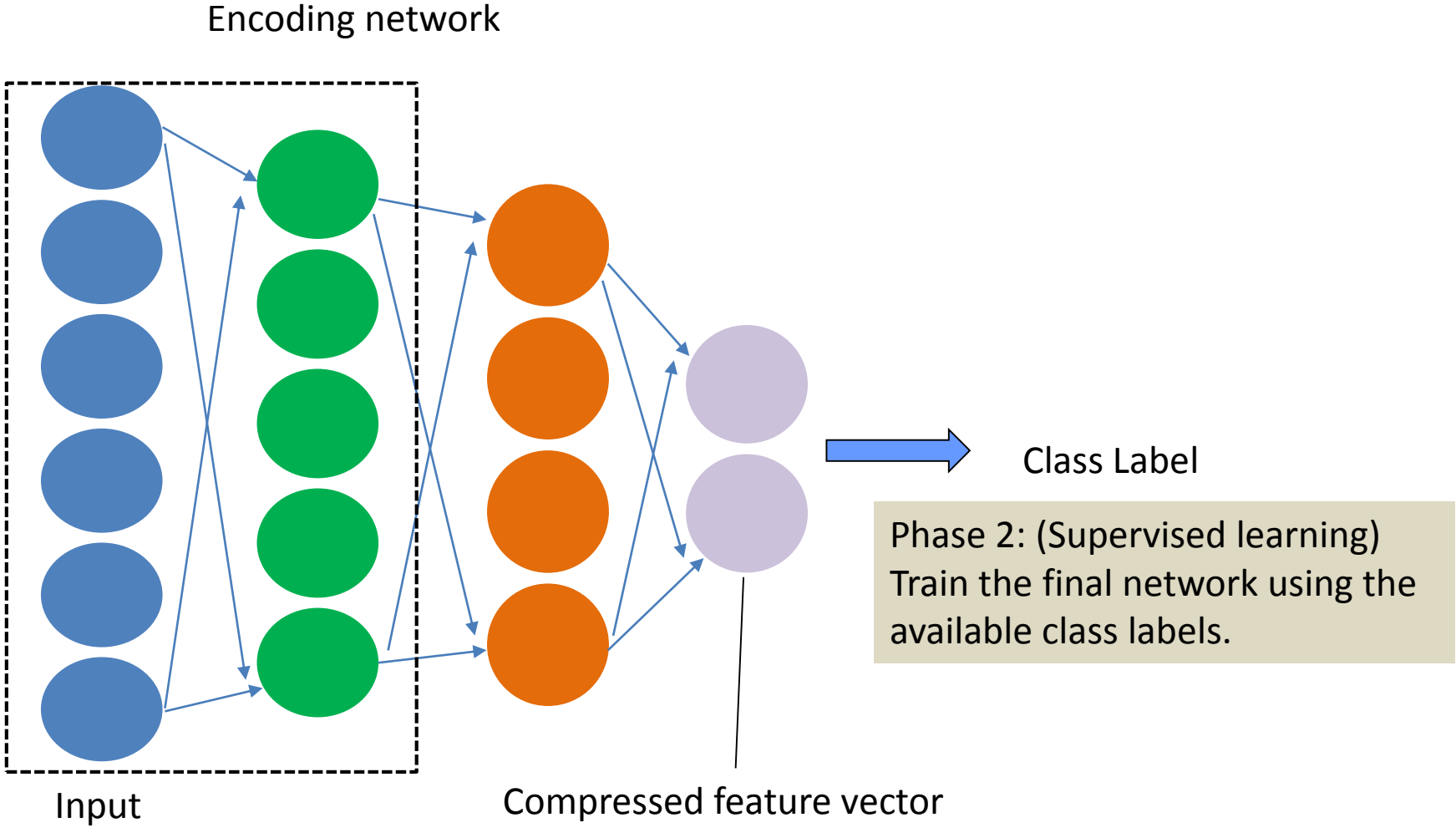
A stack of autoencoder networks:



Phase 1: Train one layer at a time and stack them

12. Advanced topics in artificial neural networks

A stack of auto encoder networks:



References.

- Machine Learning, Tom M. Mitchell, McGraw-Hill International Editions, Chapter 4.
- Neural Networks, Simon Haykin, Prentice Hall, Chapter 4.
- Introduction to Artificial Neural Systems, Jacek M. Zurada, PWS Publishing Company, Chapter 4.