

Multilayer Perceptrons

Ziping Zhao

School of Information Science and Technology
ShanghaiTech University, Shanghai, China

CS182: Introduction to Machine Learning (Spring 2023)
<http://cs182.sist.shanghaitech.edu.cn>

Ch. 11 of I2ML (Secs. 11.8 – 11.11 excluded)

Outline

Introduction

The Perceptron

Boolean Function Learning

Multilayer Perceptrons

Outline

Introduction

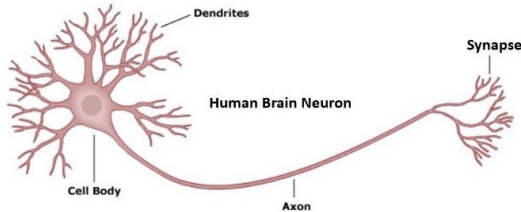
The Perceptron

Boolean Function Learning

Multilayer Perceptrons

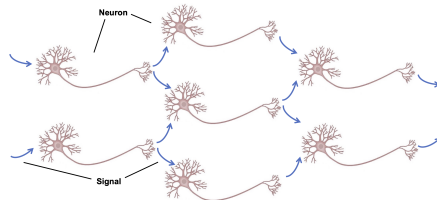
Artificial Neural Networks – I

- ▶ Cognitive scientists and neuroscientists: the aim is to understand the functioning of the brain by building models of the natural neural networks in the brain.
- ▶ Machine learning researchers: the aim (more pragmatic) is to build better computer systems based on inspirations from studying the brain.
- ▶ The human brain is quite different from a computer.
 - a computer generally has one processor
 - the brain is composed of a very large number of processing units, namely, biological neurons (nerve cells)



Artificial Neural Networks – II

- ▶ In the brain, neurons are cells within the nervous system that transmit information to other neurons, muscle, or gland cells.
- ▶ A human brain has:
 - Large number (10^{11}) of **neurons** as **processing** units
 - Large number (10^4) of **synapses** per neuron as **memory** units
 - **Parallel** processing capabilities
 - **Distributed** computation/memory
 - High **robustness** to noise and failure
- ▶ **Biological neural networks:**



- ▶ **Artificial neural networks (ANN)** mimic some characteristics of the human brain, especially with regard to the computational aspects.

Outline

Introduction

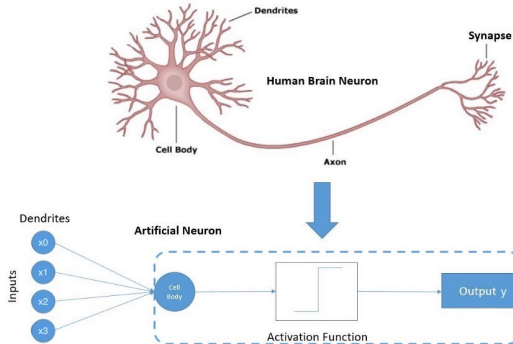
The Perceptron

Boolean Function Learning

Multilayer Perceptrons

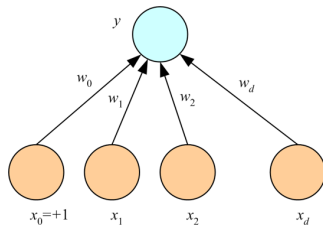
Perceptron – I

- ▶ The **perceptron** (or artificial neuron), a mathematical model of a biological neuron, is the basic processing element in artificial neural networks.



- ▶ The single-layer perceptron forms a **feedforward neural network**, i.e., an ANN where information always moves one direction; it never goes backwards.

Perceptron – II



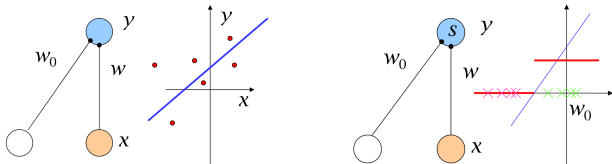
- The **output**, y , in the simplest case is a **weighted sum** of the **inputs** $\mathbf{x} = (x_0, x_1, \dots, x_d)^T$ (may come from the environment or may be the outputs of other perceptrons):

$$y = \sum_{j=1}^d w_j x_j + w_0 = \mathbf{w}^T \mathbf{x}$$

where x_0 is a special **bias unit** with $x_0 = 1$ and $\mathbf{w} = (w_0, w_1, \dots, w_d)^T$ is the **weight vector** with w_0 called the **bias weight** and $w_j, j = 1, \dots, d$, called the **connection weights** or **synaptic weights**.

What a Perceptron Does

- ▶ Regression vs. classification (figures show perceptrons with a univariate input x):



- ▶ By using it to implement a **linear discriminant function**, the perceptron can separate two classes by checking the sign of the output.
 - If we define the **threshold (or step) function**

$$s(a) = \mathbf{1}(a > 0) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases}$$

we can obtain the following decision rule:

$$\text{Choose } \begin{cases} C_1 & \text{if } s(\mathbf{w}^T \mathbf{x}) = 1 \\ C_2 & \text{otherwise} \end{cases}$$

Sigmoid Function

- ▶ Instead of using the threshold function to give a **discrete** output in $\{0, 1\}$, we may use the sigmoid function to give a **continuous** output in $[0, 1]$, i.e.,

$$o = \mathbf{w}^T \mathbf{x}$$

$$y = \text{sigmoid}(o) = \frac{1}{1 + \exp(-o)}$$

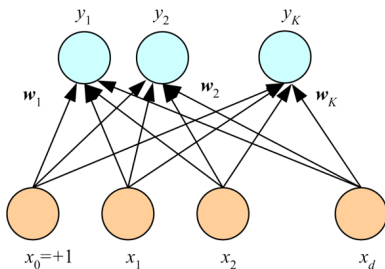
- ▶ The output may be interpreted as the **posterior probability** that the input \mathbf{x} belongs to C_1 , which at a later stage can be used, for example, to calculate the risk.

$K > 2$ Outputs - I

- ▶ K **perceptrons**, each with a weight vector \mathbf{w}_i :

$$y_i = \sum_{j=1}^d w_{ij}x_j + w_{i0} = \mathbf{w}_i^T \mathbf{x} \quad \text{or} \quad \mathbf{y} = \mathbf{W}\mathbf{x}$$

where w_{ij} is the weight from input x_j to output y_i and each row of the $K \times (d + 1)$ matrix \mathbf{W} is the weight vector of one perceptron.



- ▶ The above function performs a **linear transformation** from a d -dimensional space (neglecting x_0 since it is constant) to a K -dimensional space.
 - The network can also be used for dimensionality reduction if $K < d$.
- ▶ By defining auxiliary inputs, the linear perceptron can also be used for polynomial approximation, i.e., regression, as discussed before.

$K > 2$ Outputs – II

- Classification

Choose C_i if $y_i = \max_k y_k$

- If we need the **posterior probabilities** as well, we can use **softmax** to define y_i as:

$$o_i = \mathbf{w}_i^T \mathbf{x}$$
$$y_i = \frac{\exp(o_i)}{\sum_{k=1}^K \exp(o_k)}$$

Perceptron Learning

- ▶ Learning mode:
 - Batch learning: whole sample seen all at once.
 - Online learning: instances seen one by one.
 - Mini-batch learning: between online and batch learning.
- ▶ Advantages of online learning:
 - No need to store the whole sample.
 - Can adapt to changes in sample distribution over time.
 - Can adapt to physical changes in system components.
- ▶ The error function is not defined over the whole sample \mathcal{X} but on individual instances.
- ▶ Starting from randomly initialized weights, the parameters are adjusted a little bit at each iteration to minimize the error, without forgetting what was learned previously.
- ▶ A complete pass over all the patterns in the training set is called an epoch.

SGD for Regression

- ▶ If the error function is differentiable, **gradient descent** may be applied at each iteration to reduce the error.
- ▶ Gradient descent for online learning is also known as **stochastic gradient descent (SGD)** or online gradient descent.
- ▶ For **regression**, the error on a single instance (\mathbf{x}^t, r^t) :

$$E^t(\mathbf{w} \mid \mathbf{x}^t, r^t) = \frac{1}{2}(r^t - y^t)^2 = \frac{1}{2} \left[r^t - (\mathbf{w}^T \mathbf{x}^t) \right]^2$$

which gives the following **online update rule**:

$$\Delta w_j^t = -\eta \frac{\partial E^t}{\partial w_j} = -\eta \frac{\partial E^t}{\partial y^t} \frac{\partial y^t}{\partial w_j} = \eta (r^t - y^t) x_j^t$$

where η is a **step size** (or **learning rate** or **learning factor**) parameter which can be chosen to be decreased gradually over time to facilitate convergence.

SGD for Binary Classification

- ▶ For **classification**, we can use **logistic discrimination**.
- ▶ **Logistic** for a single instance (\mathbf{x}^t, r^t) , where $r^t = 1$ if $\mathbf{x}^t \in C_1$ and $r^t = 0$ if $\mathbf{x}^t \in C_2$, gives the output:

$$y^t = \text{sigmoid}(\mathbf{w}^T \mathbf{x}^t)$$

- ▶ **Likelihood**:

$$L^t(\mathbf{w} \mid \mathbf{x}^t, r^t) = (y^t)^{r^t} (1 - y^t)^{1-r^t}$$

- ▶ **Cross-entropy** error function:

$$E^t(\mathbf{w} \mid \mathbf{x}^t, r^t) = -\log L^t = -r^t \log y^t - (1 - r^t) \log(1 - y^t)$$

- ▶ **Online update rule**:

$$\Delta w_j^t = -\eta \frac{\partial E^t}{\partial w_j} = \eta(r^t - y^t)x_j^t$$

which is the same as the equations we saw in last lecture except that we do not sum over all of the instances but update after a single instance.

SGD for Multi-Class Classification

- **Softmax** for a single instance $(\mathbf{x}^t, \mathbf{r}^t)$, where $r_i^t = 1$ if $\mathbf{x}^t \in C_i$ and 0 otherwise, gives the outputs:

$$y_i^t = \frac{\exp(\mathbf{w}_i^T \mathbf{x}^t)}{\sum_{k=1}^K \exp(\mathbf{w}_k^T \mathbf{x}^t)}$$

- **Likelihood**:

$$L^t(\{\mathbf{w}_i\} \mid \mathbf{x}^t, \mathbf{r}^t) = \prod_i (y_i^t)^{r_i^t}$$

- **Cross-entropy** error function:

$$E^t(\{\mathbf{w}_i\} \mid \mathbf{x}^t, \mathbf{r}^t) = -\log L^t = -\sum_i r_i^t \log y_i^t$$

- **Online update rule**:

$$\Delta w_{ij}^t = -\eta \frac{\partial E^t}{\partial w_{ij}} = \eta (r_i^t - y_i^t) x_j^t$$

- All the update equations have the same form

$$\text{Update} = \text{LearningRate} \times \underbrace{(\text{DesiredOutput} - \text{ActualOutput})}_{\text{the error term for output unit } i} \times \text{Input}$$

Perceptron Learning Algorithm

```
For  $i = 1, \dots, K$ 
  For  $j = 0, \dots, d$ 
     $w_{ij} \leftarrow \text{rand}(-0.01, 0.01)$ 
Repeat
  For all  $(\mathbf{x}^t, r^t) \in \mathcal{X}$  in random order
    For  $i = 1, \dots, K$ 
       $o_i \leftarrow 0$ 
      For  $j = 0, \dots, d$ 
         $o_i \leftarrow o_i + w_{ij} x_j^t$ 
      For  $i = 1, \dots, K$ 
         $y_i \leftarrow \exp(o_i) / \sum_k \exp(o_k)$ 
      For  $i = 1, \dots, K$ 
        For  $j = 0, \dots, d$ 
           $w_{ij} \leftarrow w_{ij} + \eta(r_i^t - y_i) x_j^t$ 
Until convergence
```

Outline

Introduction

The Perceptron

Boolean Function Learning

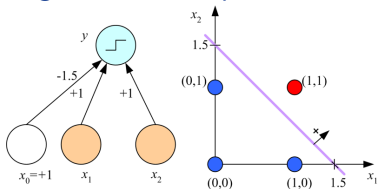
Multilayer Perceptrons

Learning Boolean Function AND

- ▶ In a Boolean function, the inputs are binary and the output is 1 if the corresponding function value is true and 0 otherwise.
- ▶ Learning a Boolean function is a **two-class classification problem**.
- ▶ **AND function** with 2 inputs and 1 output:

x_1	x_2	r
0	0	0
0	1	0
1	0	0
1	1	1

- ▶ Perceptron for AND and its **geometric interpretation**:



Learning Boolean Function XOR – I

- ▶ A simple perceptron can only learn **linearly separable** Boolean functions such as AND and OR but not **linearly nonseparable** functions such as XOR.
- ▶ **XOR function** with 2 inputs and 1 output:

x_1	x_2	r
0	0	0
0	1	1
1	0	1
1	1	0

Learning Boolean Function XOR – II

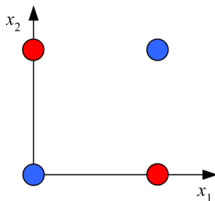
- There do not exist w_0, w_1, w_2 that satisfy the following inequalities:

$$w_0 \leq 0$$

$$w_2 + w_0 > 0$$

$$w_1 + w_0 > 0$$

$$w_1 + w_2 + w_0 \leq 0$$



- This result is not surprising since the VC dimension of a line (in two dimensions) is three. With two binary inputs there are four cases, and thus there exist problems with two inputs that are not solvable using a line; XOR is one of them.

Outline

Introduction

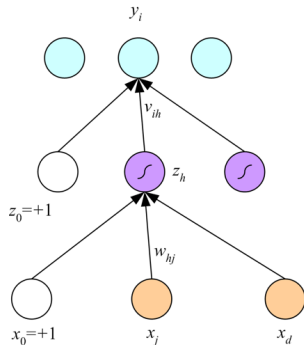
The Perceptron

Boolean Function Learning

Multilayer Perceptrons

Multilayer Perceptrons

- ▶ A single-layer perceptron can only approximate linear functions of the input \mathbf{x} and cannot solve problems like XOR, where the discriminant is nonlinear. Similarly, a perceptron cannot be used for nonlinear regression (note: nonlinear in input \mathbf{x}).
- ▶ A **multilayer perceptron (MLP)** (a.k.a. deep feedforward neural networks) has a **hidden layer** between the input and output layers.
- ▶ MLP can implement **nonlinear discriminants** (for classification) and **nonlinear regression functions** (for regression).
- ▶ We call this a **two-layer network** because the input layer performs no computation.



Forward Propagation – I

- ▶ **Input-to-hidden** (each hidden unit is a perceptron in the first layer):

$$z_h = \sigma(\mathbf{w}_h^T \mathbf{x}), \quad h = 1, \dots, H$$

- The hidden units must implement a **nonlinear function** $\sigma(\cdot)$ called **activation function** (e.g., sigmoid $\text{sigmoid}(\cdot)$ (ranging from 0 to +1), hyperbolic tangent $\tanh(\cdot)$ (ranging from -1 to +1), threshold function $s(\cdot)$, rectified linear unit $\text{ReLU}(\cdot)$, etc.)
 - ▶ If activation function is linear, the whole network is equivalent to a simple perceptron.
- The activation function is an abstraction representing the rate of action potential firing in the neuron cell.
 - ▶ Sigmoid can be seen as a **continuous, differentiable** version of thresholding.

- ▶ **Hidden-to-output** (each output unit is a perceptron in the second layer):

$$y_i \leftarrow o_i, \quad o_i = \mathbf{v}_i^T \mathbf{z} = \sum_{h=1}^H v_{ih} z_h + v_{i0}, \quad i = 1, \dots, K$$

- **Regression**: linear output units
- **Classification**: a sigmoid unit (for $K = 2$) or K outputs with a softmax (for $K > 2$)

Forward Propagation – II

- ▶ The hidden units make a **nonlinear transformation** from the d -dimensional input space to the H -dimensional space spanned by the hidden units.
 - ▶ In the new H -dimensional space, the output layer implements a **linear function**.
 - ▶ **Multiple hidden layers** may be used for implementing more complex functions of the inputs, but learning the network weights in such deep networks will be more complicated (to be considered later in the topic of Deep Learning Models).
-
- ▶ MLP's are sometimes colloquially referred to as “vanilla” neural networks, especially when they have a single hidden layer.

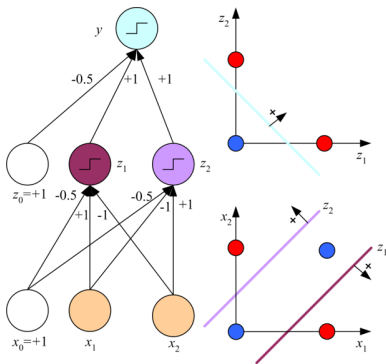
MLP for XOR

- Any Boolean function can be represented as a **disjunction of conjunctions**, e.g.

$$x_1 \text{ XOR } x_2 = \underbrace{(x_1 \text{ AND } \neg x_2)}_{z_1} \text{ OR } \underbrace{(\neg x_1 \text{ AND } x_2)}_{z_2}$$

which can be implemented by an MLP with one hidden layer.

- Two perceptrons can in parallel implement the two AND, and another perceptron on top can OR them together.
 - The hidden units and the output have the threshold activation function $s(\cdot)$ with threshold at 0.



MLP as a Universal Approximator

- ▶ The result for arbitrary Boolean functions can be extended to the **continuous** case.
- ▶ **Universal approximation theorem:**
An MLP with **one hidden layer** can approximate any continuous function on any compact subset of \mathbf{R}^n , under mild assumptions on the activation function, given sufficiently many hidden units.

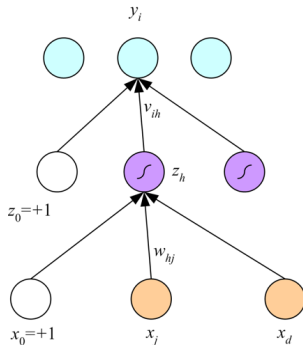
Backpropagation Learning Algorithm

- ▶ Training an MLP is the same as training a simple perceptron; the only difference is that now the output is a nonlinear function of the input due to the nonlinear activation function in the hidden units.
- ▶ Extension of the perceptron learning algorithm to multiple layers by error **backpropagation** (backward propagation) from the outputs back to the inputs.
- ▶ Learning of **hidden-to-output** weights:
Like simple perceptron learning by treating the hidden units as inputs

$$\frac{\partial E}{\partial v_{ih}} = \frac{\partial E}{\partial y_i} \frac{\partial y_i}{\partial v_{ih}}$$

- ▶ Learning of **input-to-hidden** weights:
Applying the **chain rule** to calculate the gradient

$$\frac{\partial E}{\partial w_{hj}} = \frac{\partial E}{\partial y_i} \frac{\partial y_i}{\partial z_h} \frac{\partial z_h}{\partial w_{hj}}$$



MLP Learning for Nonlinear Regression With Single Output – I

- ▶ Assuming a **single output**:

$$y^t = \sum_{h=1}^H v_h z_h^t + v_0$$

where $z_h^t = \text{sigmoid}(\mathbf{w}_h^T \mathbf{x}^t)$.

- ▶ **Error function** over entire training sample:

$$E(\mathbf{W}, \mathbf{v} \mid \mathcal{X}) = \frac{1}{2} \sum_{t=1}^N E^t(\mathbf{W}, \mathbf{v} \mid \mathbf{x}^t, r^t) = \frac{1}{2} \sum_{t=1}^N (r^t - y^t)^2$$

- ▶ **Update rule** for **second-layer weights**:

$$\Delta v_h = -\eta \sum_{t=1}^N \frac{\partial E^t}{\partial y^t} \frac{\partial y^t}{\partial v_h} = \eta \sum_{t=1}^N (r^t - y^t) z_h^t$$

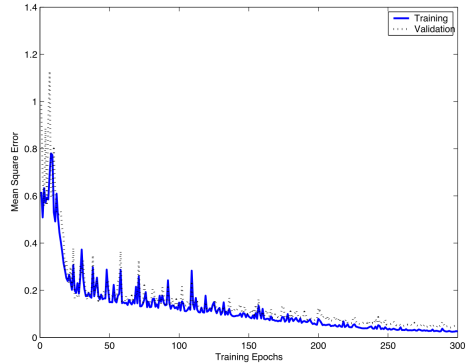
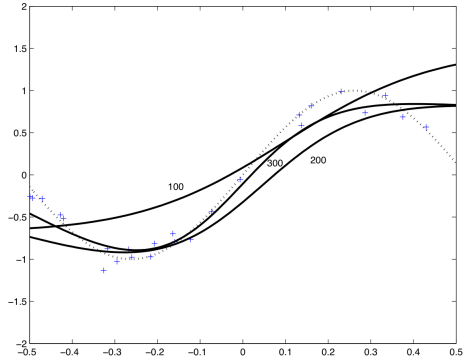
MLP Learning for Nonlinear Regression With Single Output – II

- Update rule for first-layer weights:

$$\begin{aligned}\Delta w_{hj} &= -\eta \frac{\partial E}{\partial w_{hj}} = -\eta \sum_{t=1}^N \frac{\partial E^t}{\partial y^t} \frac{\partial y^t}{\partial z_h^t} \frac{\partial z_h^t}{\partial w_{hj}} \\ &= -\eta \sum_{t=1}^N -(r^t - y^t) \times v_h \times z_h^t (1 - z_h^t) x_j^t \\ &= \eta \sum_{t=1}^N (r^t - y^t) v_h z_h^t (1 - z_h^t) x_j^t\end{aligned}$$

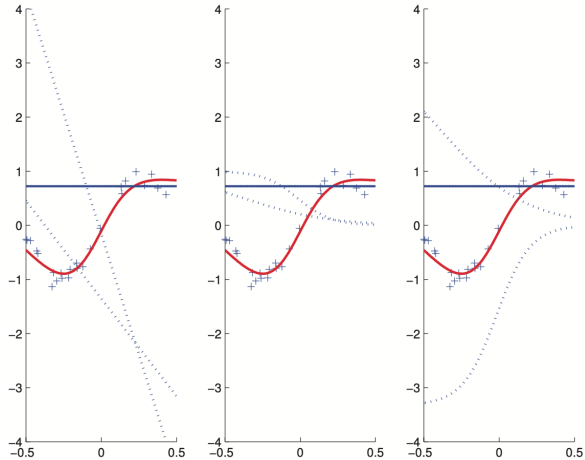
- The $(r^t - y^t)v_h$ acts like the error for hidden unit z_h . $(r^t - y^t)$ is the error in the output which is **backpropagated** from the output to the hidden unit weighted by the “responsibility” of the hidden unit as given by its weight v_h .
- Either **(mini-)batch learning** or **online learning** may be carried out.

Example – I



*Evolution of **regression function** and **error** over epochs in online learning*

Example – II



MLP Learning for Nonlinear Regression With Multiple Outputs – I

- ▶ When there are multiple output units, a number of regression problems are learned at the same time.
- ▶ Outputs:

$$y_i^t = \sum_{h=1}^H v_{ih} z_h^t + v_{i0}, \quad i = 1, \dots, K$$

- ▶ Error function:

$$E(\mathbf{W}, \mathbf{V} \mid \mathcal{X}) = \frac{1}{2} \sum_{t=1}^N \sum_{i=1}^K (r_i^t - y_i^t)^2$$

MLP Learning for Nonlinear Regression With Multiple Outputs – II

- Update rule for second-layer weights:

$$\Delta v_{ih} = \eta \sum_{t=1}^N (r_i^t - y_i^t) z_h^t$$

- Update rule for first-layer weights (sigmoid is used as the nonlinear activation function):

$$\Delta w_{hj} = \eta \sum_{t=1}^N \left[\sum_{i=1}^K (r_i^t - y_i^t) v_{ih} \right] z_h^t (1 - z_h^t) x_j^t$$

The $\sum_{i=1}^K (r_i^t - y_i^t) v_{ih}$ is the accumulated backpropagated error of hidden unit h from all output units.

Algorithm

```
Initialize all  $v_{ih}$  and  $w_{hj}$  to  $\text{rand}(-0.01, 0.01)$   
Repeat  
  For all  $(\mathbf{x}^t, r^t) \in \mathcal{X}$  in random order  
    For  $h = 1, \dots, H$   
       $z_h \leftarrow \text{sigmoid}(\mathbf{w}_h^T \mathbf{x}^t)$   
    For  $i = 1, \dots, K$   
       $y_i = \mathbf{v}_i^T \mathbf{z}$   
    For  $i = 1, \dots, K$   
       $\Delta \mathbf{v}_i = \eta(r_i^t - y_i^t) \mathbf{z}$   
    For  $h = 1, \dots, H$   
       $\Delta \mathbf{w}_h = \eta(\sum_i (r_i^t - y_i^t) v_{ih}) z_h (1 - z_h) \mathbf{x}^t$   
    For  $i = 1, \dots, K$   
       $\mathbf{v}_i \leftarrow \mathbf{v}_i + \Delta \mathbf{v}_i$   
    For  $h = 1, \dots, H$   
       $\mathbf{w}_h \leftarrow \mathbf{w}_h + \Delta \mathbf{w}_h$   
Until convergence
```

The weights are initialized to small random values, e.g., in the range $[-0.01, 0.01]$, so as not to saturate the sigmoids.

MLP Learning for Nonlinear Two-Class Discrimination

- Output:

$$y^t = \text{sigmoid}(o^t), \quad \text{with } o^t = \sum_{h=1}^H v_h z_h^t + v_0$$

which approximate the **posterior probabilities** $P(C_1 | \mathbf{x}^t)$

- Error function:

$$E(\mathbf{W}, \mathbf{v} | \mathcal{X}) = - \sum_{t=1}^N \left[r^t \log y^t + (1 - r^t) \log(1 - y^t) \right]$$

- Update rules (sigmoid is used as the nonlinear activation function):

$$\Delta v_h = \eta \sum_{t=1}^N (r^t - y^t) z_h^t$$
$$\Delta w_{hj} = \eta \sum_{t=1}^N (r^t - y^t) v_h z_h^t (1 - z_h^t) x_j^t$$

- As in the simple perceptron, the update equations for reg. and class. are identical.
- Multilayer Perceptrons

MLP Learning for Nonlinear Multi-Class Discrimination

► Outputs:

$$y_i^t = \frac{\exp(o_i^t)}{\sum_k \exp(o_k^t)}, \quad \text{with } o_i^t = \sum_{h=1}^H v_{ih} z_h^t + v_{i0}, \quad i = 1, \dots, K$$

which approximate the **posterior probabilities** $P(C_i | \mathbf{x}^t)$.

► Error function:

$$E(\mathbf{W}, \mathbf{V} | \mathcal{X}) = - \sum_{t=1}^N \sum_{i=1}^K r_i^t \log y_i^t$$

► Update rules (sigmoid is used as the nonlinear activation function):

$$\begin{aligned} \Delta v_{ih} &= \eta \sum_{t=1}^N (r_i^t - y_i^t) z_h^t \\ \Delta w_{hj} &= \eta \sum_{t=1}^N \left[\sum_{i=1}^K (r_i^t - y_i^t) v_{ih} \right] z_h^t (1 - z_h^t) x_j^t \end{aligned}$$

MLP With Multiple Hidden Layers

- ▶ It is possible to have multiple hidden layers each with its own weights and applying the sigmoid function to its weighted sum.
- ▶ It constitutes a deep **feedforward neural network**.
- ▶ Training such a network can be implemented in a similar way.