Multilayer Perceptron, Case-based Reasoning, and Unsupervised clustering

Resource(s):

1. ***An Introduction to Neural Networks for Beginners,*** Dr. Andy Thomas (Adventures in Machine Learning).
2. ***Artificial Intelligence: A Guide to Intelligence Systems,*** Michael Negnevitsky, Addison Wesley
3. ***Artificial Intelligence Foundations of Computational Agents***, David L. Poole and Alan K Mackworth, Cambridge University Press, 2019.
4. ***Pytorch Tutorialspoint Simply Easy Learning*** 2019

Multilayer Neural Network

- To cope with problems such as **linearly separable**, we need a **Multi-Layer Perceptron (MLP)** network**,** also called a **backpropagation neural network** (**BPNN**)
  - A multilayer neural network structure consists of an ***input layer***, a ***hidden layer,*** and an ***output layer***
- The layers of a **multilayer NN** are **fully connected**; that is, every neuron in each layer is connected to every other neuron in the adjacent forward layer (see **Figure 4.1**).
- **Links** connect the neurons, and each link has a **numerical weight**.
  - The **weights** are the primary means of long-term memory

Structure of a Multilayer NN

- A multilayer NN learns through repeated **weights adjustments** by the **backpropagation** learning algorithm.

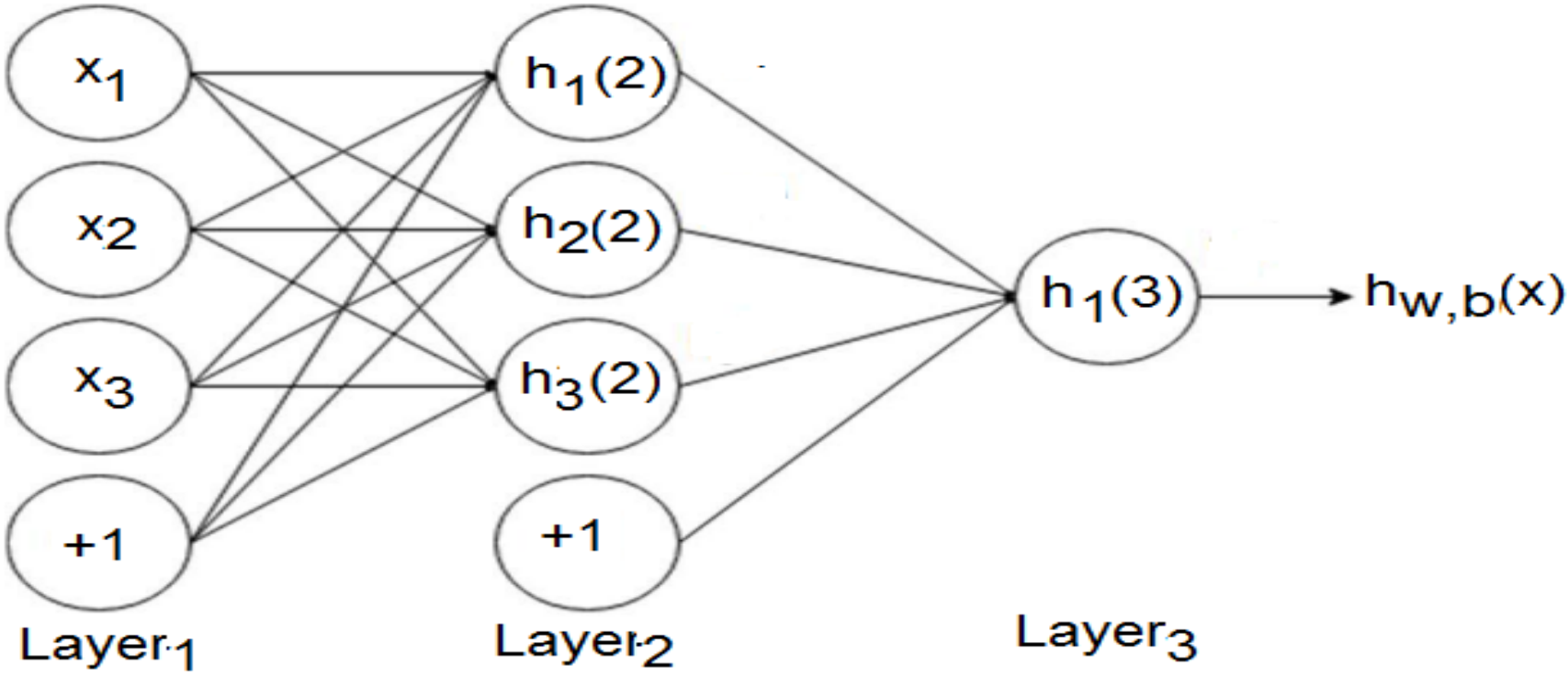


Figure 4.1 A three layer neural network

Structure of a Multilayer NN

- The **three layers** of the network can be seen in **figure 4.1**
- **Layer 1** (**L1**) represents the ***input layer***, where the external input data enters the network.
- **Layer 2** (**L2**) is called the ***hidden layer,*** as this layer is not part of the input or output.
  - **Note:** neural networks can have many hidden layers.
- Finally, **Layer 3** (**L3**) is the ***output layer***, where the output of the entire network is available.
  - You can see many connections among the layers.
  - As can be seen, each node in **L1** connects to all the nodes in **L2**.
  - Likewise for the nodes in **L2** to the single output node **L3**.
  - Each of these connections will have an **associated weight**.

Structure of a Multilayer NN

- As shown in **figure 4.1**, the **bias** is connected to each node in the subsequent layer.
- The **bias** in **layer one** is connected to all the nodes in layer 2.
  - Because the **bias is not a node** with an ***activation function***, **it has no data input but an input value of +1.**
- **Figure 4.2** shows the neural network with its total **weights** and **bias** values:
  - The connection between **node1 (x1)** in **layer1** and **node2** (**h1(2)**) in **layer2**, the weight would be $\mathbf{w21}^{(1)}$.
  - Similarly, the **bias** in **layer1** and the first node in **layer2** is given by **b1(1)** .

Structure of a Multilayer NN

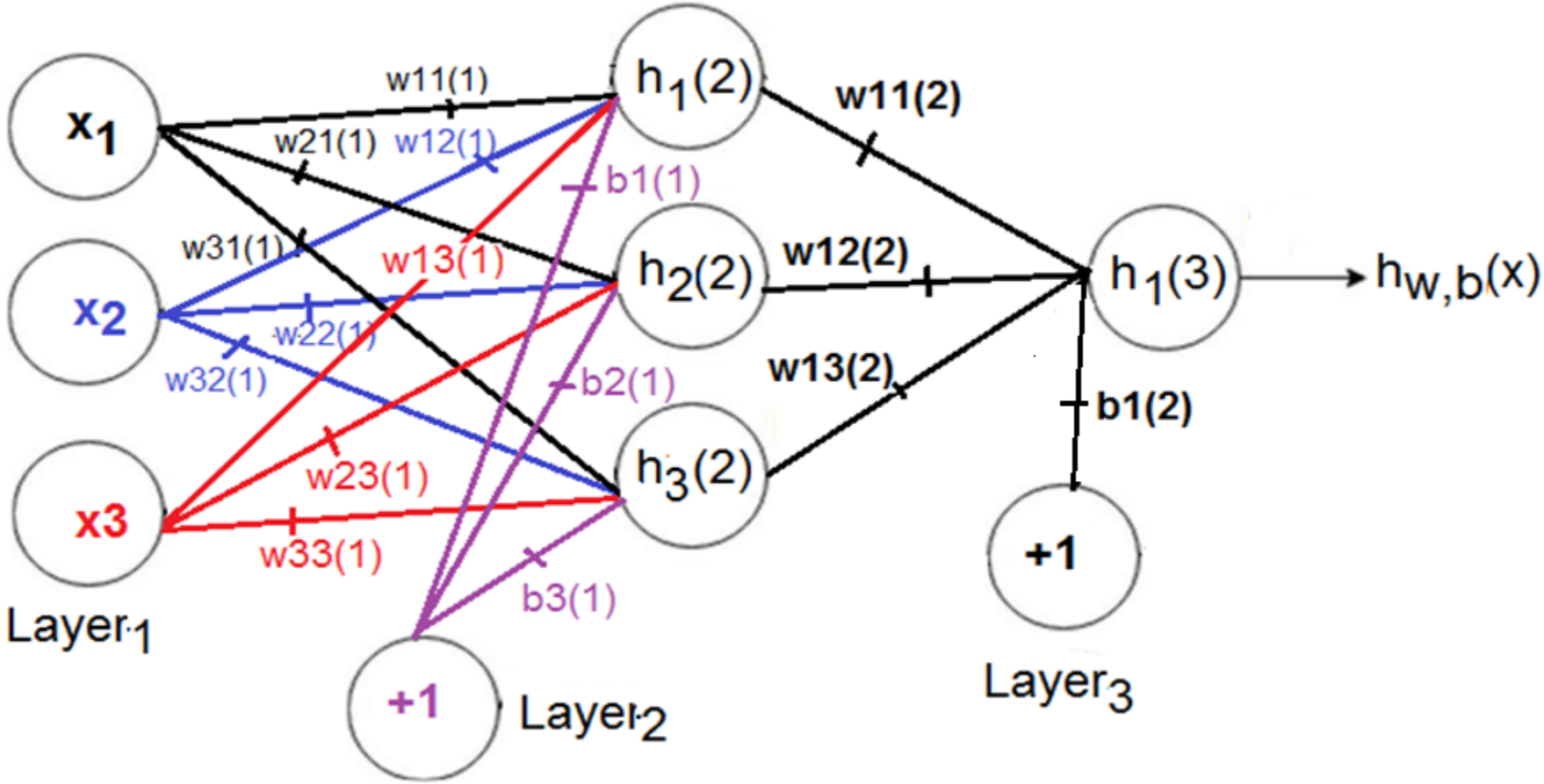


Figure 4.2 The neural network with weight and bias values.

Structure of a Multilayer NN

- Note that we have all the notations for the **3-layer** neural network (see **Figure 4.2**).
- It is now time to look at how to calculate the **output** of the network when the **input, weights,** and **bias** are known.
- The process of calculating the neural network output based on the given values is called the **feed-forward pass** (or **feed-forward**).

The Feed Forward Pass

- To demonstrate how to calculate the **output** from the **input** in a neural network, consider the of the three layer neural network in **Figure 4.2**.
- Below it is presented in equation form (the ***sigmoid activation*** function ***f*(.)** is used to calculate the output):

$$h_1^{(2)} = f(w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + w_{13}^{(1)}x_3 + b_1^{(1)})$$
$$h_2^{(2)} = f(w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2 + w_{23}^{(1)}x_3 + b_2^{(1)})$$
$$h_3^{(2)} = f(w_{31}^{(1)}x_1 + w_{32}^{(1)}x_2 + w_{33}^{(1)}x_3 + b_3^{(1)})$$
$$h_{W,b}(x) = h_1^{(3)} = f(w_{11}^{(2)}h_1^{(2)} + w_{12}^{(2)}h_2^{(2)} + w_{13}^{(2)}h_3^{(2)} + b_1^{(2)})$$

The Feed Forward Pass

- As can be observed, rather than taking the **weighted input variables** (x_1, x_2, x_3), the final node takes as input the **weighted output** of the nodes of the second layer ($h_1^{(2)}$, $h_2^{(2)}$, $h_3^{(2)}$), plus the **weighted bias**.
- Therefore, you can see in the equation form the hierarchical nature of artificial neural network:
  - **Calculate the output $h_1^{(3)}$ of the given NN based on the following values:**

    $w_{11}(1) = w_{12}(1) = w_{13}(1) = 0.2$, $w_{21}(1) = w_{22}(1) = w_{23}(1) = 0.4$,
    $w_{31}(1) = w_{32}(1) = w_{33}(1) = 0.6$, $w_{11}(2) = w_{12}(2) = w_{13}(2) = 0.5$,
    $b_1(1) = b_2(1) = b_3(1) = 0.8$, $b_1(2) = 0.2$, $x_1 = 1.5$, $x_2 = 2$, $x_3 = 3$

The Feed Forward Pass

- The output of each neuron in the given neural network can be calculated manually as shown below:

$$h_1^{(2)} = f(0.2 * 1.5 + 0.2 * 2.0 + 0.2 * 3.0 + 0.8) = 0.8909$$

$$h_2^{(2)} = f(0.4 * 1.5 + 0.4 * 2.0 + 0.4 * 3.0 + 0.8) = 0.9677$$

$$h_3^{(2)} = f(0.6 * 1.5 + 0.6 * 2.0 + 0.6 * 3.0 + 0.8) = 0.9909$$

$$h_{W,b}(x) = h_1^{(3)} = f(0.5 * 0.8909 + 0.5 * 0.9677 + 0.5 * 0.9909 + 0.2) = 0.8354$$

```
#=======================
def f(x):
    return 1/(1 + np.exp(-x))
import numpy as np
myList1 = [[0.2, 0.2, 0.2], [0.4, 0.4, 0.4], [0.6, 0.6, 0.6]]
w1 = np.array(myList1) #convert list into multi array (numpy array)
w2 = np.zeros((1,3)) # create a 1D list with 1 row and 3 columns filled with zero elements [0,0,0]
mylist2 = [0.5, 0.5, 0.5]
w2[0,:] = np.array(mylist2)
b1 = np.array([0.8, 0.8, 0.8])
b2 = np.array([0.2])
b = [b1, b2]
w = [w1, w2]
h = 0
n_layer = 3
x = [1.5, 2.0, 3.0]
for l in range(n_layer - 1):
    if(l == 0):
        node_in = x
    else:
        node_in = h
    h = np.zeros((w[l].shape[0],))
    for i in range (w[l].shape[0]):
        f_sum = 0
        for j in range(w[l].shape[1]):
            f_sum += w[l][i][j] * node_in[j]
        f_sum += b[l][i]
        h[i] = f(f_sum)
print( "\n The O/P of the final neuron node is %0.4f"% h)
#=======================
'The O/P of the final neuron node is 0.8355
```

Multilayer Neural Network

- A **multilayer NN** is a **feed-forward NN** with one or more hidden layers (shown in **Figure 6.8**)

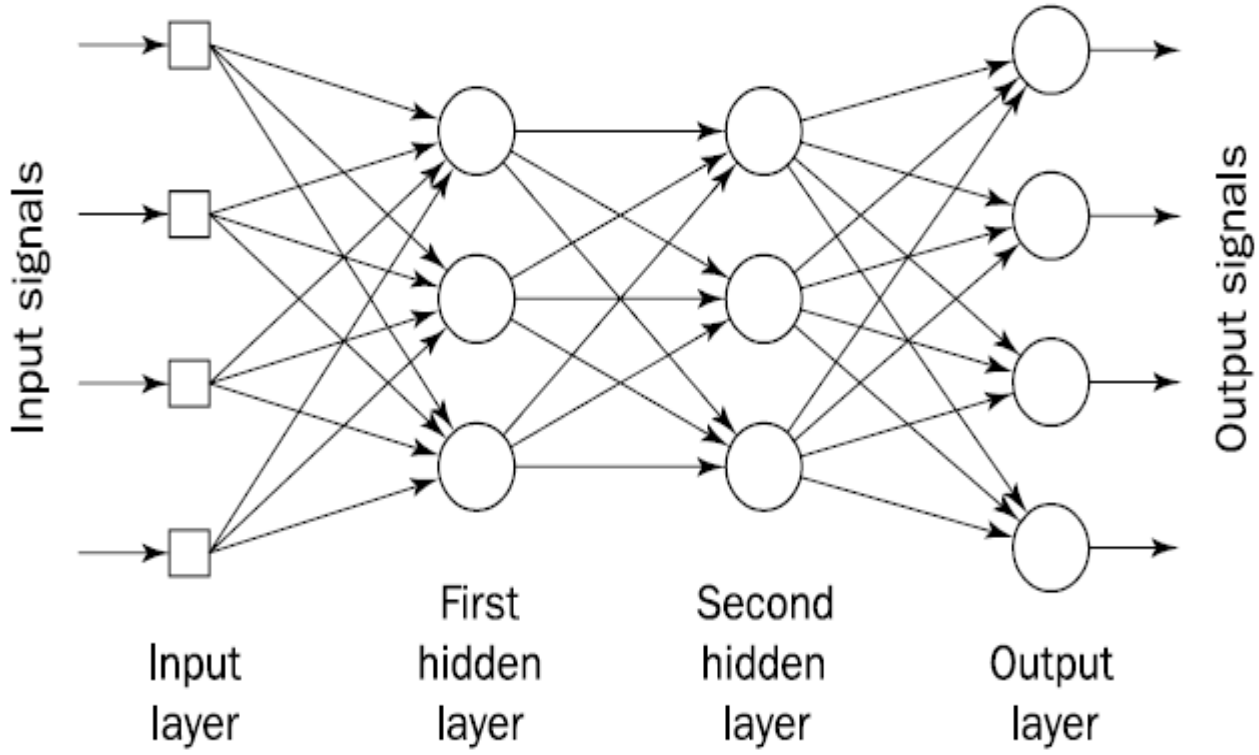


Figure 6.8 Multilayer perceptron with two hidden layers

Multilayer Neural Network

- Each layer in a **multilayer NN** has its specific function
  - **Input layer**: The layer accepts input signals from the outside and distributes them to all neurons in the **hidden** layer
    - **Input layer does not process any signals**
  - **Output layer**: the output layer accepts output signals from the hidden layer and establishes the output pattern
  - Neurons in the **hidden layer** detect the features; the **weights** of the neurons represent **the features** hidden in the input patterns
    - The output layer then uses these features in determining the **output pattern**

Why are middle layers called hidden layers?

- The **hidden layer** 'hides' NN's desired output
- That is, neurons in the **hidden layer** cannot be observed through the input/output behavior of the network
  - **There is no obvious way to know the desired output from the hidden layer**
    - The layer itself determines the **desired output** of the hidden layer
  - The layer topology of commercial ANN has 3 or 4 layers, including one or two hidden layers
  - Each layer can contain from 10 – 1000 neurons
  - But most practical applications use only **3-layer** neural networks because of their computational burden

How Does a Multilayer NN learn?

- Most popular **multilayer NN** training method is the **back-propagation** algorithm [1969 Bryson]
- Learning **multilayer NN** proceeds the same way as for a **perceptron** (**delta** rule)
- But in **multilayer NN**, *a* **training set** of input patterns must be presented to the network
- The network computes its **output** pattern, and if there is any **error** (the difference between **desired** and **actual** outputs), **the weights are adjusted to reduce this error,** also called the **gradient descent** method.

Backpropagation Algorithm

- In a **back-propagation NN**, the learning algorithm has **two phases:**

  Phase 1:

  – A training **input pattern (training dataset** or **seen data)** is presented to the network's **input layer**

  – The network then propagates the **input pattern** from layer to layer until the **output layer** generates the output pattern

  Phase 2:

  – If the **calculated/predicted output** is different from the **desired output**, an **error** is calculated and then **propagated backward through the network from the output layer to the input layer (**is the **backpropagation** operation!**)**

  – **The weights are modified as the error is propagated backward**

Backpropagation Algorithm

- In **BPNNs**, a neuron determines its **sum_of_weight** (***X***) with **bias** (θ) like that of the **Delta rule** for perceptron:

$$X = \sum_{i=1}^{n} x_i * w_i - \theta$$

where ***n*** is the number of **inputs** and θ is the **bias** (or threshold) applied to the neuron

- Neurons in the **back-propagation network** use a **sigmoid activation function** for output:

$$Y_{sigmoid} = 1/(1+e^{-x})$$

Backpropagation Algorithm

- To derive the **back-propagation learning** rule, we need to consider a **3-layer** network as shown in **Figure 6.9**

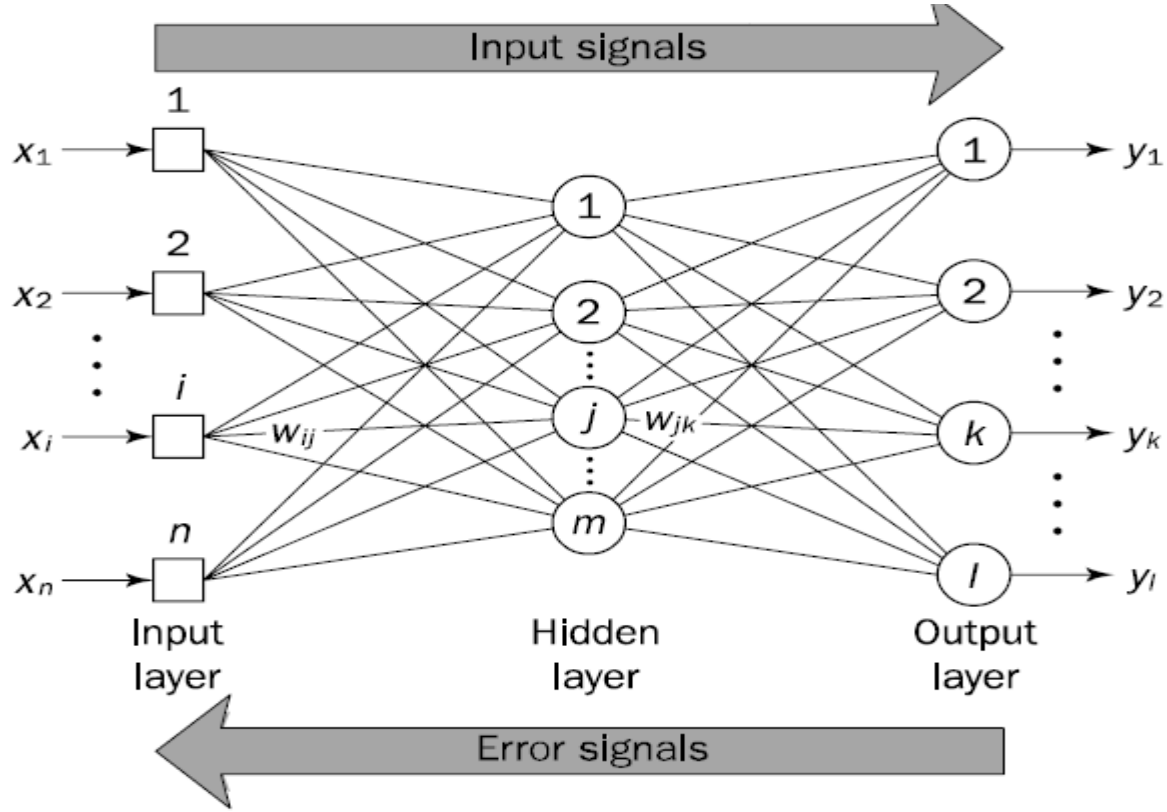


Figure 6.9 Three-layer back-propagation neural network

Back-propagation Algorithm

- From **Figure 6.9**, the indices ***i***, ***j,*** and ***k*** refer to the **input**, **hidden** neuron, and **output** neuron, respectively
- **Input signals**, x_1, x_2,…., x_n are propagated through the network from **left to right**, and **error signals**, e_1, e_2, …., e_i, from **right to left**
  - The symbol w_{ij} denotes the weight for the connection between neuron ***i*** in the input layer and neuron ***j*** in the hidden layer,
  - And the symbol w_{jk} is the weight between neuron ***j*** in the hidden layer and neuron ***k*** in the output layer

Back-propagation Algorithm

- To propagate **error signals**, start at the **output layer** and work **backward** to the **hidden layer**
- The **error signal** at the output of the neuron ***k*** at iteration ***p*** is $e_k(p)$ defined by:

$$e_k(p) = y_{d,k}(p) - y_k(p) \qquad (1)$$

Where $y_{d,k}(p)$ is the **desired output** of neuron ***k*** at iteration *p*

- Use a procedure to **update weight** w_{jk}:

$$w_{jk}(p + 1) = w_{jk}(p) + \Delta w_{jk}(p) \qquad (2)$$

Where $\Delta w_{jk}(p)$ is the **weight correction** of output neuron ***k***

$$\Delta w_{jk}(p) = \alpha * y_j(p) * \delta_k(p) \qquad (3)$$

Back-propagation Algorithm

- Where $\delta_k(p)$ is the **error gradient** at neuron ***k*** in the **output layer** at iteration ***p***
- The **error gradient** is determined as the **derivative of the sigmoid activation function** (***F'***), multiplied by **error $e_k(p)$**. Thus for neuron ***k***, the $\delta_k(p)$ is given as:

$$\delta_k(p) = F'[X_k(p)] * e_k(p) \quad (4)$$

Where $X_k(p)$ is the net **weighted input** to neuron ***k*** at iteration *p*:

$$X_k(p) = \sum_{j=1}^{m} x_{jk}(p) \times w_{jk}(p) - \theta_k \quad (5)$$

The output of neuron ***k*** at iteration ***p*** is, $y_k(p) = 1 / (1 + e^{-X_k(p)})$ (6)

The derivative function, $F'[X_k(p)] = y_k(p) * [1 - y_k(p)]$ (7)

(where ***m*** is the *number of neurons* in the **hidden layer)**

Back-propagation Algorithm

- Thus, from (4) and (7), the **error gradient, $\delta_k(p)$** is given as:

$$\delta_k(p) = y_k(p) * [1 - y_k(p)] * e_k(p) \qquad (8)$$

- Next is to determine **weight correction** for the **neuron *j*** in the **hidden layer**
- Use a procedure (similar to (2) and (3)) to **update weight w_{ij}** :

$$w_{ij}(p + 1) = w_{ij}(p) + \Delta w_{ij}(p)$$

$$\Delta w_{ij}(p) = \alpha * x_i(p) * \delta_j(p) \qquad (9)$$

Where $\delta_j(p)$ represents the **error gradient** at neuron ***j*** in the **hidden** layer, and it can be calculated as (8):

Back-propagation Algorithm

The **error gradient** $\delta_j(p)$ of neuron ***j*** depends on **3** values:

1) The **output** of neuron ***j*** = $y_j(p)$
2) The **error gradient** of neuron ***k*** = $\delta_k(p)$
3) The **input weights** of neuron ***k*** = $w_{jk}(p)$. Hence $\delta_j(p)$ can be given as:

$$\delta_j(p) = y_j(p) \times [1 - y_j(p)] \times \sum_{k=1}^{l} \delta_k(p) \times w_{jk}(p) \qquad (10)$$

Here the error of neuron ***j*** is the **error gradient** of output neurons multiplied by their ***weights***, where ***l*** is the number of neurons in the output layer. Where $y_j(p)$ is

$y_j(p) = 1 / 1 + e^{-Xj(p)}$; Output of the neuron ***J***

$$X_j(p) = \sum_{i=1}^{n} x_i(p) \times w_{ij}(p) - \theta_j$$

where ***n*** is the number of neurons in the **input layer**

Back-propagation Algorithm

Step 1: ***Initialisation***

Set all the weights and threshold levels of the network to random numbers uniformly distributed inside a small range (Haykin, 1999):

$$\left(-\frac{2.4}{F_i}, +\frac{2.4}{F_i}\right),$$

where F_i is the total number of inputs of neuron i in the network. The weight initialisation is done on a neuron-by-neuron basis.

Back-propagation Algorithm

Step 2: ***Activation***

Activate the back-propagation neural network by applying inputs $x_1(p), x_2(p), \ldots, x_n(p)$ and desired outputs $y_{d,1}(p), y_{d,2}(p), \ldots, y_{d,n}(p)$.

(a) Calculate the actual outputs of the neurons in the hidden layer:

$$y_j(p) = sigmoid\left[\sum_{i=1}^{n} x_i(p) \times w_{ij}(p) - \theta_j\right],$$

where n is the number of inputs of neuron j in the hidden layer, and *sigmoid* is the sigmoid activation function.

Back-propagation Algorithm

(b) Calculate the actual outputs of the neurons in the output layer:

$$y_k(p) = sigmoid\left[\sum_{j=1}^{m} x_{jk}(p) \times w_{jk}(p) - \theta_k\right],$$

where m is the number of inputs of neuron k in the output layer.

Back-propagation Algorithm

Step 3: ***Weight training***

Update the weights in the back-propagation network propagating backward the errors associated with output neurons.

(a) Calculate the error gradient for the neurons in the output layer:

$$\delta_k(p) = y_k(p) \times [1 - y_k(p)] \times e_k(p)$$

where

$$e_k(p) = y_{d,k}(p) - y_k(p)$$

Calculate the weight corrections:

$$\Delta w_{jk}(p) = \alpha \times y_j(p) \times \delta_k(p)$$

Update the weights at the output neurons:

$$w_{jk}(p+1) = w_{jk}(p) + \Delta w_{jk}(p)$$

Back-propagation Algorithm

(b) Calculate the error gradient for the neurons in the hidden layer:

$$\delta_j(p) = y_j(p) \times [1 - y_j(p)] \times \sum_{k=1}^{l} \delta_k(p) \times w_{jk}(p)$$

Calculate the weight corrections:

$$\Delta w_{ij}(p) = \alpha \times x_i(p) \times \delta_j(p)$$

Update the weights at the hidden neurons:

$$w_{ij}(p+1) = w_{ij}(p) + \Delta w_{ij}(p)$$

Step 4: ***Iteration***

Increase iteration p by one, go back to Step 2 and repeat the process until the selected error criterion is satisfied.

Back-propagation Algorithm

- Consider a 3-layer **back-propagation neural network** (BPNN) shown in **Figure 6.10**. Suppose that the network is required to perform logical **XOR**.
- Neurons **1** and **2** in the input layer accept inputs $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$, respectively and redistribute these inputs to the neurons in the hidden layer without any processing:

  $x_{13} = x_{14} = x_1$ and $x_{23} = x_{24} = x_2$ **(see Figure 6.10)**
- The effect of the **bias** applied to a neuron in the hidden or output layer is represented by its weight, bias weight θ connected to a fixed input equal to **-1**

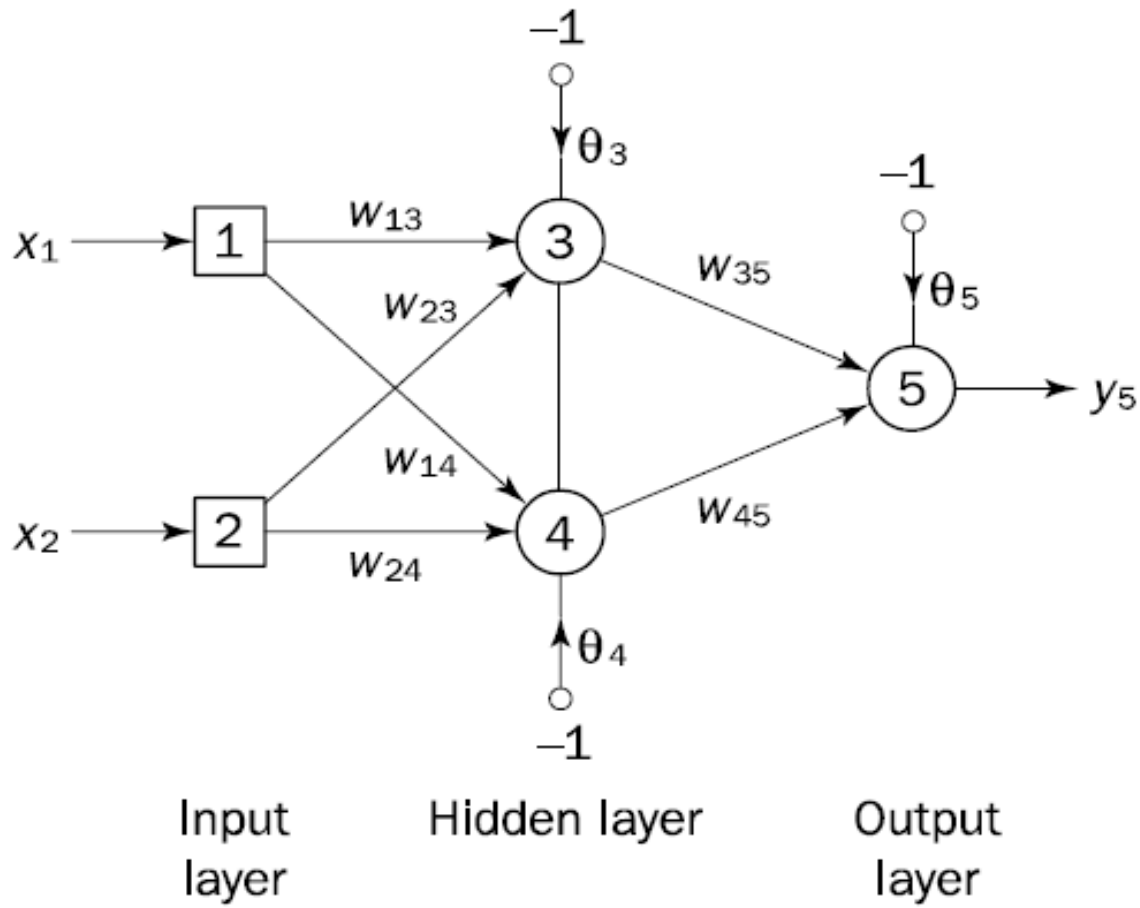


Figure 6.10 Three-layer network for solving the Exclusive-OR operation

Back-propagation Algorithm

- The initial weights and threshold levels are set randomly as follows:

$$w_{13} = 0.5,\ w_{14} = 0.9,\ w_{23} = 0.4,\ w_{24} = 1.0,\ w_{35} = -1.2,\ w_{45} = 1.1,$$
$$\theta_3 = 0.8,\ \theta_4 = -0.1 \text{ and } \theta_5 = 0.3.$$

Back-propagation Algorithm

Consider a training set where inputs x_1 and x_2 are equal to 1 and desired output $y_{d,5}$ is 0. The actual outputs of neurons 3 and 4 in the hidden layer are calculated as

$$y_3 = sigmoid\,(x_1 w_{13} + x_2 w_{23} - \theta_3) = 1/[1 + e^{-(1\times 0.5 + 1\times 0.4 - 1\times 0.8)}] = 0.5250$$

$$y_4 = sigmoid\,(x_1 w_{14} + x_2 w_{24} - \theta_4) = 1/[1 + e^{-(1\times 0.9 + 1\times 1.0 + 1\times 0.1)}] = 0.8808$$

Now the actual output of neuron 5 in the output layer is determined as

$$y_5 = sigmoid\,(y_3 w_{35} + y_4 w_{45} - \theta_5) = 1/[1 + e^{-(-0.5250\times 1.2 + 0.8808\times 1.1 - 1\times 0.3)}] = 0.5097$$

Thus, the following error is obtained:

$$e = y_{d,5} - y_5 = 0 - 0.5097 = -0.5097$$

Back-propagation Algorithm

The next step is weight training. To update the weights and threshold levels in our network, we propagate the error, e, from the output layer backward to the input layer.

First, we calculate the error gradient for neuron 5 in the output layer:

$$\delta_5 = y_5(1 - y_5)e = 0.5097 \times (1 - 0.5097) \times (-0.5097) = -0.1274$$

Back-propagation Algorithm

Then we determine the weight corrections assuming that the learning rate parameter, α, is equal to 0.1:

$$\Delta w_{35} = \alpha \times y_3 \times \delta_5 = 0.1 \times 0.5250 \times (-0.1274) = -0.0067$$

$$\Delta w_{45} = \alpha \times y_4 \times \delta_5 = 0.1 \times 0.8808 \times (-0.1274) = -0.0112$$

$$\Delta \theta_5 = \alpha \times (-1) \times \delta_5 = 0.1 \times (-1) \times (-0.1274) = 0.0127$$

Next we calculate the error gradients for neurons 3 and 4 in the hidden layer:

$$\delta_3 = y_3(1 - y_3) \times \delta_5 \times w_{35} = 0.5250 \times (1 - 0.5250) \times (-0.1274) \times (-1.2) = 0.0381$$

$$\delta_4 = y_4(1 - y_4) \times \delta_5 \times w_{45} = 0.8808 \times (1 - 0.8808) \times (-0.1274) \times 1.1 = -0.0147$$

Back-propagation Algorithm

We then determine the weight corrections:

$$\Delta w_{13} = \alpha \times x_1 \times \delta_3 = 0.1 \times 1 \times 0.0381 = 0.0038$$

$$\Delta w_{23} = \alpha \times x_2 \times \delta_3 = 0.1 \times 1 \times 0.0381 = 0.0038$$

$$\Delta \theta_3 = \alpha \times (-1) \times \delta_3 = 0.1 \times (-1) \times 0.0381 = -0.0038$$

$$\Delta w_{14} = \alpha \times x_1 \times \delta_4 = 0.1 \times 1 \times (-0.0147) = -0.0015$$

$$\Delta w_{24} = \alpha \times x_2 \times \delta_4 = 0.1 \times 1 \times (-0.0147) = -0.0015$$

$$\Delta \theta_4 = \alpha \times (-1) \times \delta_4 = 0.1 \times (-1) \times (-0.0147) = 0.0015$$

Back-propagation Algorithm

At last, we update all weights and threshold levels in our network:

$$w_{13} = w_{13} + \Delta w_{13} = 0.5 + 0.0038 = 0.5038$$

$$w_{14} = w_{14} + \Delta w_{14} = 0.9 - 0.0015 = 0.8985$$

$$w_{23} = w_{23} + \Delta w_{23} = 0.4 + 0.0038 = 0.4038$$

$$w_{24} = w_{24} + \Delta w_{24} = 1.0 - 0.0015 = 0.9985$$

$$w_{35} = w_{35} + \Delta w_{35} = -1.2 - 0.0067 = -1.2067$$

$$w_{45} = w_{45} + \Delta w_{45} = 1.1 - 0.0112 = 1.0888$$

$$\theta_3 = \theta_3 + \Delta \theta_3 = 0.8 - 0.0038 = 0.7962$$

$$\theta_4 = \theta_4 + \Delta \theta_4 = -0.1 + 0.0015 = -0.0985$$

$$\theta_5 = \theta_5 + \Delta \theta_5 = 0.3 + 0.0127 = 0.3127$$

The training process is repeated until the sum of squared errors is less than 0.001.

Back-propagation Algorithm

- Why do we need to **sum the square errors** (**SOSE**)?
  - The **SOSE** is a helpful indicator of the network's performance
  - **The back-propagation** training algorithm attempts to minimize this criterion
  - When the value of the **SOSE** in an entire pass through all training sets, or **epoch**, is **sufficiently small**, a network is considered to have **converged**- the training is satisfied!
    - Normally, sufficiently small **SOSE** is defined as less than 0.001. **Figure 6.11** represents a **learning curve**
    - The **sum of square errors** plotted versus the **no. of epochs** used in training

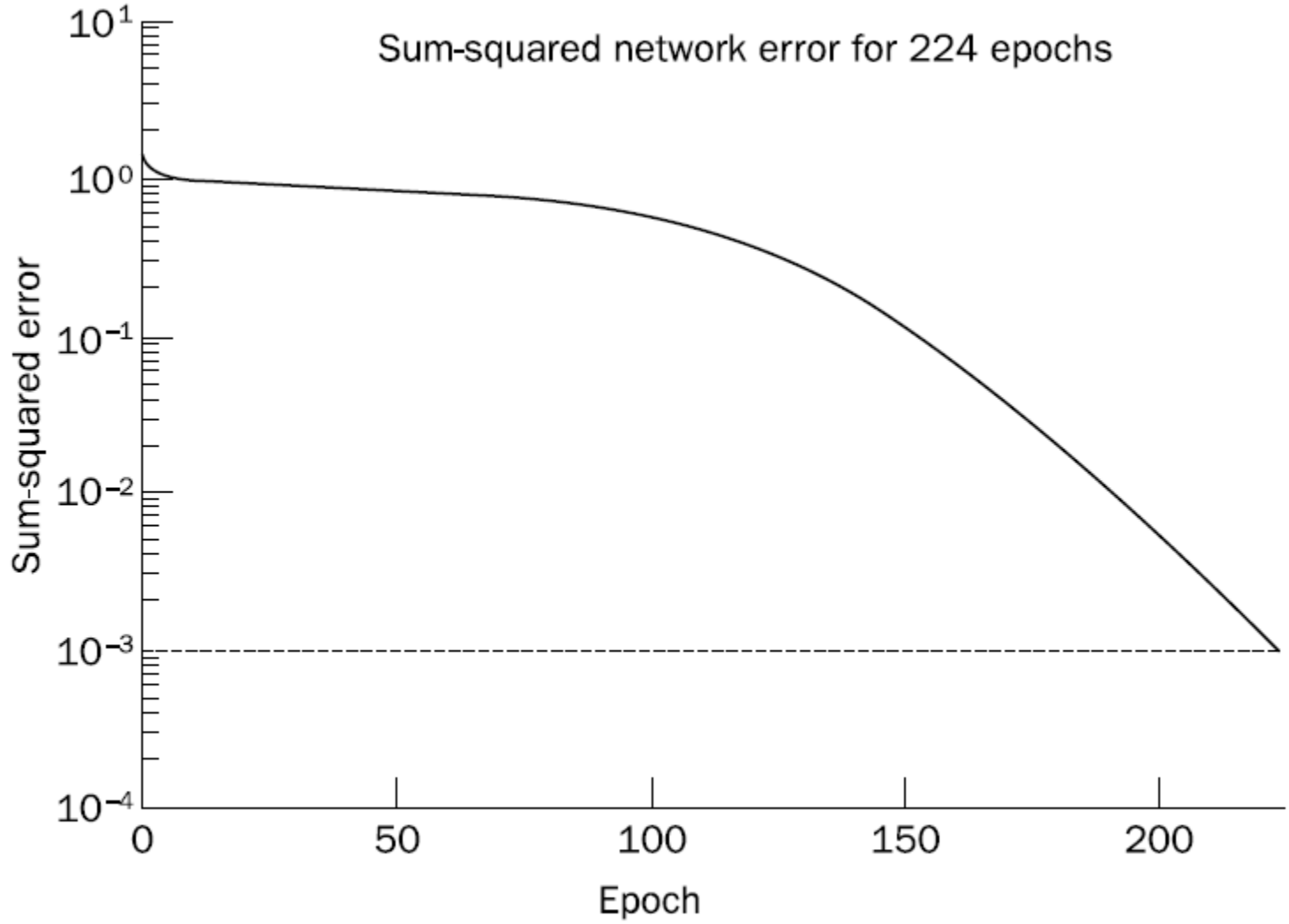


Figure 6.11 Learning curve for operation Exclusive-OR

Back-propagation Algorithm

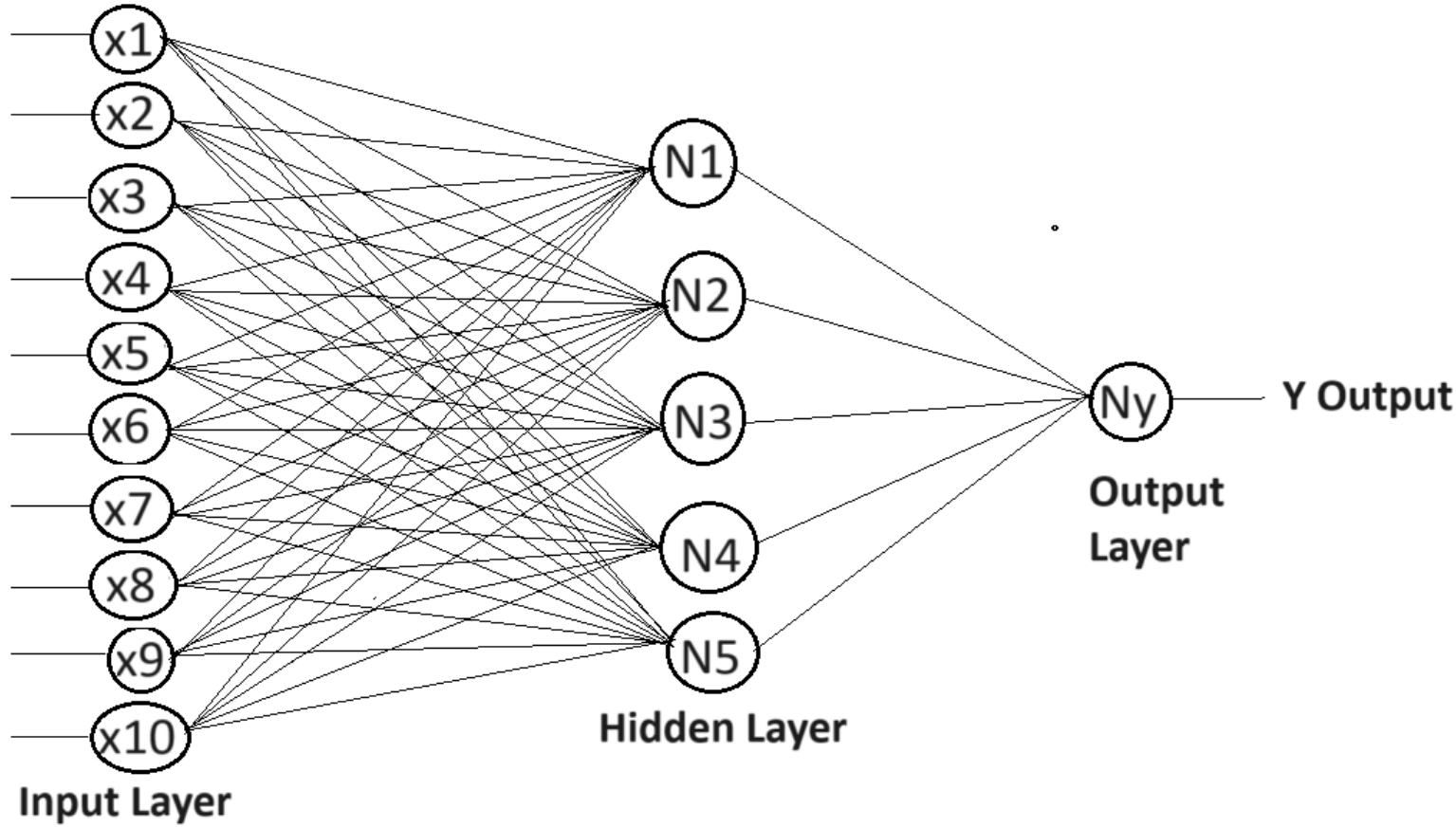
Table 6.4 Final results of three-layer network learning: the logical operation Exclusive-OR

Inputs		Desired output	Actual output	Error	Sum of squared
x_1	x_2	y_d	y_5	e	errors
1	1	0	0.0155	−0.0155	0.0010
0	1	1	0.9849	0.0151	
1	0	1	0.9849	0.0151	
0	0	0	0.0175	−0.0175	

10-5-1 MLP Training with Pytorch

Ref: Pytorch Tutorialspoint Simply Easy Learning 2019
https://machinelearningmastery.com/develop-your-first-neural-network-with-pytorch-step-by-step/

An MLP with **an input layer** (with **ten data inputs**), **one hidden layer** (with **five neurons**), and **an output layer** (with **one output neuron**):



10-5-1 MLP Training with Pytorch

Ref: Pytorch Tutorialspoint Simply Easy Learning 2019
https://machinelearningmastery.com/develop-your-first-neural-network-with-pytorch-step-by-step/

A sample Labeled dataset for the 10-5-1 MLP:

```
        x1      x2      x3      x4      x5      x6      x7      x8      x9      x10              y
tensor([[ 0.2578,  1.1945,  0.3258, -0.3953,  0.1590,  0.3438,  0.9223,  0.2381, -0.0959,  1.3321],    tensor([[1.],
        [-0.7001,  0.1640, -0.0445,  0.0087,  0.8256,  0.0278, -0.3577, -0.2125, -0.9708,  0.5353],            [0.],
        [-1.0306, -0.5569, -0.3449, -1.3263,  1.4548, -0.5716,  0.8582, -0.9023, 1.5266,  1.1466],            [0.],
        [-0.3002, -0.1148,  1.9257, -0.1089, -1.2256,  0.9990,  0.3761,  0.5110, -2.2634,  1.6524],            [1.],
        [-0.9363,  0.4183, -1.8527, -1.8949,  0.7123,  0.7653, -0.1206,  0.0470, 1.0495, -0.4390],            [1.],
        [-0.5451,  0.0839, -1.2980, -0.5948, -0.0399,  0.6705, -0.1995, -1.4762, -0.7573,  0.4255],            [1.],
        [ 0.3538,  1.3610, -0.6983, -0.3181, -0.4332,  0.5897,  0.6721, -0.0380, -0.5999,  1.5046],            [0.],
        [ 0.6471,  0.7389, -1.0197,  0.3321,  0.6532, -1.6685, -0.7545,  0.4889, -0.7317,  2.0207],            [0.],
        [-0.8263,  0.9765,  0.4628, -0.6537, -0.0200, -1.0438, -0.5958,  0.7953, -0.2744,  0.5024],            [1.],
        [ 1.1697,  1.1846,  0.4588,  0.3605, -0.4200,  0.2378,  1.2570,  0.5001, 0.7969, -0.1527]])           [1.]])
```

10-5-1 MLP Training with Pytorch

Ref: Pytorch Tutorialspoint Simply Easy Learning 2019
https://machinelearningmastery.com/develop-your-first-neural-network-with-pytorch-step-by-step/

The Loss after 50 iterations:

```
7
8  #the first neural network model (10-5-1 topology) using PyTorch
9  #import torch import torch.nn as nn
10 import torch
11 import torch.nn as nn
12 # Defining input size, hidden layer size, output size and batch size respectively
13 n_in, n_h, n_out, batch_size = 10, 5, 1, 10
14 # Create dummy input and target tensors (data)
15 x = torch.randn(batch_size, n_in) # x is the input data
16 #y is the output
17 y = torch.tensor([[1.0], [0.0], [0.0], [1.0], [1.0], [1.0], [0.0], [0.0], [1.0], [1.0]])
18 model = nn.Sequential(nn.Linear(n_in, n_h), #Create a nn model
19 nn.ReLU(), nn.Linear(n_h, n_out), nn.Sigmoid()) # network model creation completed
20 criterion = torch.nn.MSELoss() # Next, Construct the loss function
21 # Construct the optimizer (Stochastic Gradient Descent in this case)
22 optimizer = torch.optim.SGD(model.parameters(), lr=0.01) # Lr is the learnoing rate
23 # Gradient Descent
24 for epoch in range(50): # Forward pass: Compute predicted y by passing x to the model
25     y_pred = model(x)
26     loss = criterion(y_pred, y) # Compute and print loss
27     print('epoch: ', epoch,' loss: ', loss.item())
28     optimizer.zero_grad() # Zero gradients, perform a backward pass, and update the weights.
29     loss.backward()  # perform a backward pass (backpropagation)
30     optimizer.step()  # Update the parameters
```

```
epoch:  26  loss:  0.237449169158935
epoch:  27  loss:  0.237313672900199
epoch:  28  loss:  0.237178280949592
epoch:  29  loss:  0.237042948603630
epoch:  30  loss:  0.236907631158828
epoch:  31  loss:  0.236772373318672
epoch:  32  loss:  0.236637666821479
epoch:  33  loss:  0.236504361033439
epoch:  34  loss:  0.236371248960495
epoch:  35  loss:  0.236237600445747
epoch:  36  loss:  0.236105561256408
epoch:  37  loss:  0.235970526933670
epoch:  38  loss:  0.235839933156967
epoch:  39  loss:  0.235705047845840
epoch:  40  loss:  0.235572934150695
epoch:  41  loss:  0.235439658164978
epoch:  42  loss:  0.235306769609451
epoch:  43  loss:  0.235173746943473
epoch:  44  loss:  0.235040754079818
epoch:  45  loss:  0.234908491373062
epoch:  46  loss:  0.234774261713027
epoch:  47  loss:  0.234643489122390
epoch:  48  loss:  0.234508752822875
epoch:  49  loss:  0.234377548098564
```

10-5-1 MLP Training with Pytorch

Ref: Pytorch Tutorialspoint Simply Easy Learning 2019
https://machinelearningmastery.com/develop-your-first-neural-network-with-pytorch-step-by-step/

- **The Loss after 6000 iterations:**

```
#the first neural network model (10-5-1 topology) using PyTorch
#import torch import torch.nn as nn
import torch
import torch.nn as nn
# Defining input size, hidden layer size, output size and batch size respectively
n_in, n_h, n_out, batch_size = 10, 5, 1, 10
# Create dummy input and target tensors (data)
x = torch.randn(batch_size, n_in) # x is the input data
#y is the output
y = torch.tensor([[1.0], [0.0], [0.0], [1.0], [1.0], [1.0], [0.0], [0.0], [1.0], [1.0]])
model = nn.Sequential(nn.Linear(n_in, n_h), #Create a nn model
nn.ReLU(), nn.Linear(n_h, n_out), nn.Sigmoid()) # network model creation completed
criterion = torch.nn.MSELoss() # Next, Construct the loss function
# Construct the optimizer (Stochastic Gradient Descent in this case)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01) # Lr is the learnoing rate
# Gradient Descent
for epoch in range(6000): # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x)
    loss = criterion(y_pred, y) # Compute and print loss
    print('epoch: ', epoch,' loss: ', loss.item())
    optimizer.zero_grad() # Zero gradients, perform a backward pass, and update the weights.
    loss.backward()  # perform a backward pass (backpropagation)
    optimizer.step()  # Update the parameters
```

```
epoch:  5977  loss:  0.011449099518358707
epoch:  5978  loss:  0.01144617609679699
epoch:  5979  loss:  0.011443293653428555
epoch:  5980  loss:  0.011440401896834373
epoch:  5981  loss:  0.011437478475272655
epoch:  5982  loss:  0.01143460813909769
epoch:  5983  loss:  0.011431709863245487
epoch:  5984  loss:  0.01142879668623209
epoch:  5985  loss:  0.011426068842411041
epoch:  5986  loss:  0.01142330002039671
epoch:  5987  loss:  0.011420386843383312
epoch:  5988  loss:  0.011417540721595287
epoch:  5989  loss:  0.011414638720452785
epoch:  5990  loss:  0.011411738581955433
epoch:  5991  loss:  0.011408901773393154
epoch:  5992  loss:  0.011405996046960354
epoch:  5993  loss:  0.01140310987830162
epoch:  5994  loss:  0.011400269344449043
epoch:  5995  loss:  0.011397374793887138
epoch:  5996  loss:  0.011394496075809002
epoch:  5997  loss:  0.01139165461063385
epoch:  5998  loss:  0.011388760060071945
epoch:  5999  loss:  0.011385897174477577
```

10-5-1 MLP Training with Pytorch

Ref: Pytorch Tutorialspoint Simply Easy Learning 2019
https://machinelearningmastery.com/develop-your-first-neural-network-with-pytorch-step-by-step/

```
# Gradient Descent
for epoch in range(6000): # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x)
    loss = criterion(y_pred, y) # Compute and print loss
    print('epoch: ', epoch,' loss: ', loss.item())
    optimizer.zero_grad() # Zero gradients, perform a backward pass, and update the weights.
    loss.backward()  # perform a backward pass (backpropagation)
    optimizer.step()  # Update the parameters
# ============Compute accuracy (no_grad is optional)============================
with torch.no_grad():
    y_pred = model(x)
accuracy = (y_pred.round() == y).float().mean()
print(f"Accuracy {accuracy}")
#==================Predictions from the Trained Model============================
```

```
Accuracy 0.8999999761581421
```

10-5-1 MLP Training with Pytorch

Ref: Pytorch Tutorialspoint Simply Easy Learning 2019
https://machinelearningmastery.com/develop-your-first-neural-network-with-pytorch-step-by-step/

```
#==================Predictions from the Trained Model==========================
print(" \nMake predictions with the trained model:\n")
predictions = (model(x) > 0.5).int()
for i in range(5):# put five sets of inputs to see the predicted outputs
    print('%s => %d (expected %d)' % (x[i].tolist(), predictions[i], y[i]))
```

```
Make predictions with the trained model:

[-0.9031498432159424, 0.25740745663642883, 0.6390330791473389, 1.5277125835418701, 0.7614229321479797, -1.0282466411590576,
-0.323503702878952, -1.7030458450317383, -1.1639043092727661, -0.5878746509552002] => 1 (expected 1)
[0.7100346684455872, 0.3341756761074066, -0.7248346209526062, 0.8175957202911377, 0.13197177648544312, -0.6774640083312988,
0.11272069066762924, 2.0795109272003174, 0.13057050108909607, -0.32734739780426025] => 1 (expected 0)
[1.5633020401000977, 0.38462698459625244, 2.2310454845428467, -0.9729742407798767, -0.7570013999938965, -0.3202422559261322,
-0.6343169808387756, -2.1352767944335938, 0.9575261473655701, -1.1325501203536987] => 0 (expected 0)
[0.07438746839761734, -0.03009151481091976, -0.49172094464302063, 0.9121787548065186, 0.2569417953491211, 0.14446085691452026,
1.218924880027771, 0.26921284198760986, -1.4013251066207886, 0.3254304528236389] => 1 (expected 1)
[1.0857501029968262, 1.7694453001022339, 0.18390406668186188, 0.34524303674697876, 1.1150881052017212, 0.4616564214229584,
0.33876678347587585, -0.7609703540802002, -0.07252119481563568, 1.9657427072525024] => 1 (expected 1)
```

Exercise

- Show the completed version of the **2-input XOR** function with the MLP (use **2-x-1** topology).
- Predict the value of **BMI** (Body Mass Index) from the ***Gender***, ***Height,*** and ***Weight*** of a person using an **MLP** with a **3-x-1** topology. Use the ***bmi.csv*** dataset for training the MLP. **Please don't use any Python MLP library***; use the MLP learning steps from this lecture slide. Use the* ***sigmoid activation*** *function and modify your dataset accordingly*.

Case-Based Reasoning

- **Case-based reasoning** is used for *classification* and r*egression*.
- In **case-based reasoning**, the training examples (the cases) are stored and accessed to solve a new problem.
- To get a prediction for a **new example**, those cases that are similar or close to the new example are used to predict the value of the **target features** of the new example.
  - Unlike *decision trees* and *neural networks*, relatively little work must be done offline.

Case-Based Reasoning: *k*-nearest neighbors

- In the **case base reasoning**, if the cases are simple, one algorithm that works well is to use the ***k*-nearest neighbors** for some given number ***k***.
  - Given a new example, the ***k*** training examples with input features closest to that are used to predict the target value for the new example.
- The prediction could be the *mode*, *average*, or some **interpolation** between the prediction of these ***k-training*** examples,
  - perhaps weighting **closer examples** more than **distant examples**.

Case-Based Reasoning: *k*-Nearest Neighbors

- For this method to work, a **distance metric** is required that measures the closeness of two examples:
  - Suppose $\boldsymbol{X_i(e)}$ is a numerical representation of the value of feature $\boldsymbol{X_i}$ for the example $\boldsymbol{e}$.
  - Then $(\boldsymbol{X_i(e_1)} - \boldsymbol{X_i(e_2)})$ is the difference between example $\boldsymbol{e_1}$ and $\boldsymbol{e_2}$ on the dimension defined by feature $\boldsymbol{X_i}$.
  - The **Euclidean distance**, the ***square root of the sum of the squares of the dimension differences***, could be used as the distance between two examples.

$$d(e_1, e_2) = \sqrt{\sum_i \ (x_i(e_1) - x_i(e_2))^2}$$

k-Nearest Neighbors (*k*-NN)

- The ***k*- Nearest Neighbors** (***k*-NN**) algorithm is among the simplest machine learning algorithms.
- The ***k***- Nearest Neighbors (***k*-NN**) belong to the family of **supervised** machine learning algorithms
  - labeled (Target Variable) dataset used to predict the class of new data point.
- Where ***k*** is a positive integer that indicates the **number of neighbors** of a test object in a class
- In pattern recognition, the ***k*-NN** is a non-parametric method used for classification and regression.

k-Nearest Neighbors (*k*-NN)

- ***k*-NN** is an instance or case-based learning technique where the function is only approximated locally, and all computation is deferred until classification.
- The ***k*-NN** algorithm is a robust classifier often used as a benchmark for more complex classifiers such as Artificial Neural Networks (ANN) or Support vector machines (SVM).

k-Nearest Neighbors (*k*-NN)

- A peculiarity of the ***k*-NN** algorithm is that it is sensitive to the local structure of the data (**training data**).
  - *k*-NN could be one of the first choices for classification when there is little or no prior knowledge about the distribution data.
- The technique used with ***k*-NN** is to assign **weight** to the neighbors' contribution so that the nearer neighbors contribute more to the average than the more distant ones.

k-Nearest Neighbors (*k*-NN)

- The training examples (or training data) are vectors in a multi-dimensional feature space, each with a class label.
  - The **algorithm's training phase** consists of storing the training samples' feature vectors and class labels.
  - In the **classification phase**, ***k*** is a user-defined constant, and an unlabeled vector (a query or test point) is classified by assigning the most frequent label among the ***k*** training samples nearest to that query point.

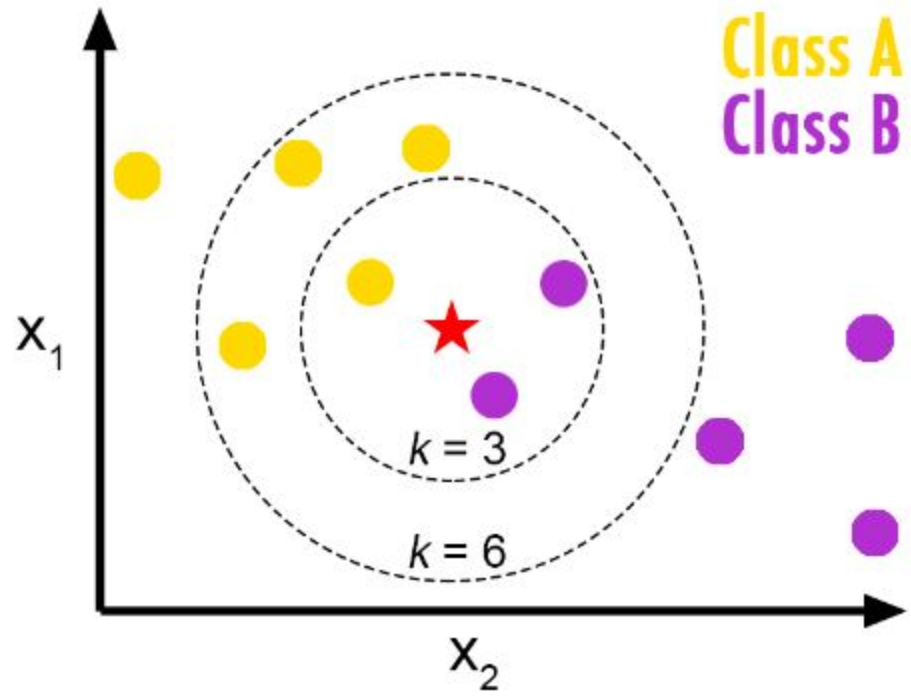
k-Nearest Neighbors (*k*-NN)

- A commonly used distance metric for the *continuous variable* is **Euclidean Distance**.
  - For discrete variables, such as text classification, **Hamming distance** can be used.
- The best choice of ***k*** depends upon the data; generally, larger values of ***k*** reduce the effect of the noise on the classification but make boundaries between classes less distinct.
  - The special case where the class is predicted to be the class of the closest training sample when ***k* = 1**.

k-Nearest Neighbors (*k*-NN)

- Pseudo Code of *k*-NN:
  1. Load the dataset
  2. Initialize the value of ***k***
  3. For getting the predicted class:
     1. Calculate the distance between **test data** and each row of **training data**.
     2. Sort the calculated distances in ascending order based on distance values.
     3. Get the top ***k*** rows from the sorted array
     4. Get the most frequent class of these rows
     5. Return the predicted class

k-Nearest Neighbors (*k*-NN)



k-NN Distance functions

Distance functions

Euclidean

$$\sqrt{\sum_{i=1}^{k}(x_i - y_i)^2}$$

Manhattan

$$\sum_{i=1}^{k}|x_i - y_i|$$

Minkowski

$$\left(\sum_{i=1}^{k}\left(|x_i - y_i|\right)^q\right)^{1/q}$$

Hamming Distance

$$D_H = \sum_{i=1}^{k}|x_i - y_i|$$

$$x = y \Rightarrow D = 0$$

$$x \neq y \Rightarrow D = 1$$

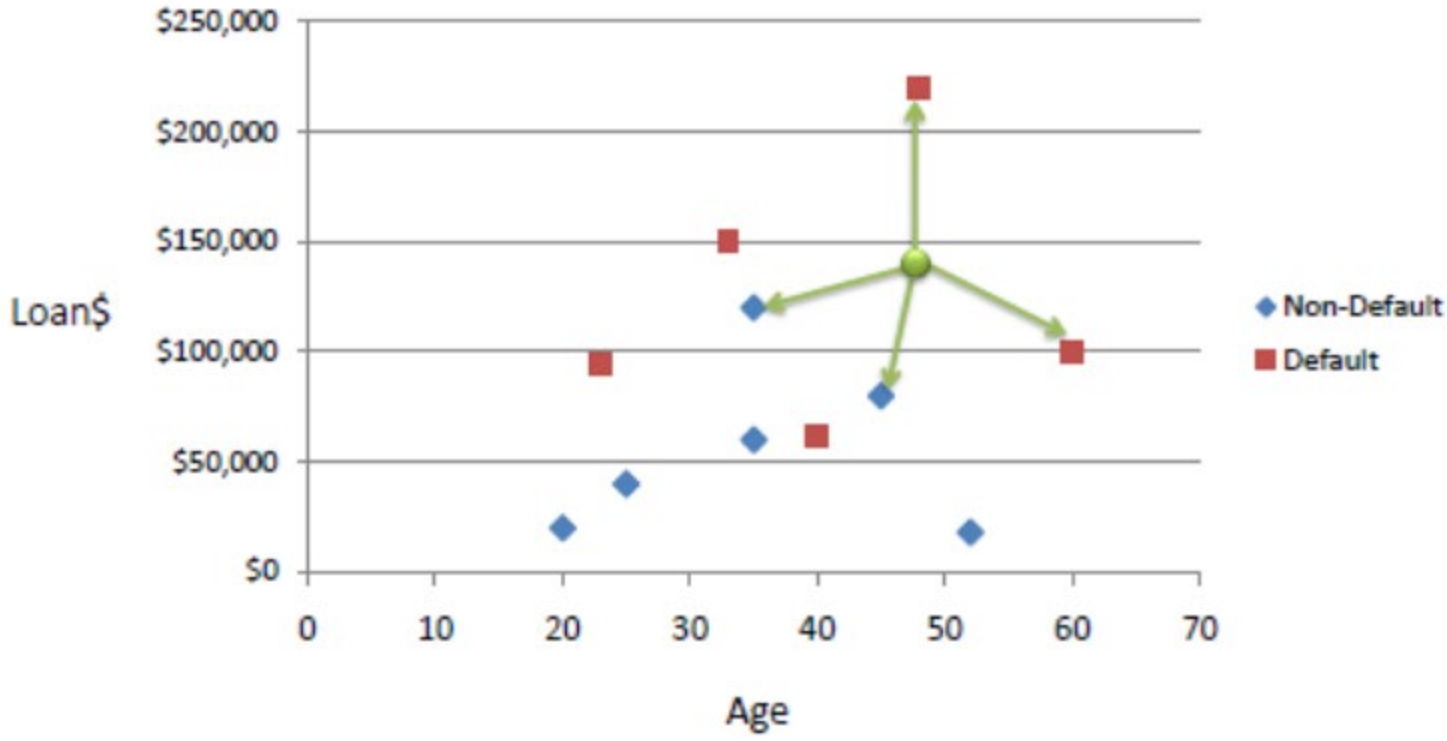
X	Y	Distance
Male	Male	0
Male	Female	1

k-NN Distance functions

- Choosing the optimal value for ***k*** is best done by inspecting the data.
- In general, a considerable ***k*** value is more precise as it reduces the overall noise, but there is no guarantee.
- **Cross-validation** is another way to retrospectively determine a good ***k*** value using an independent dataset to validate the ***k*** value.
- Historically, the optimal ***k*** for most datasets has been between 3-10.

k-NN Example

- Consider the following data concerning credit default. Age and Loan are two numerical variables (predictors) and Default is the target.



k-NN Example

- We can now use the training set to classify an **unknown** case (Age = 48 and Loan = $142,000) using **Euclidean distance**.
- If ***k* = 1,** the nearest neighbor is the last case in the training set with **Default ⇒ Y**.
  - D = Sqrt[(48-33)^2 + (142000-150000)^2] = 8000.01 >> Default ⇒ Y
- With ***k* = 3**, there are two Default=Y and one Default=N out of three closest neighbors. The prediction for the unknown case is again Default =Y.

k-NN Example

Age	Loan	Default	Distance	
25	$40,000	N	102000	
35	$60,000	N	82000	
45	$80,000	N	62000	
20	$20,000	N	122000	
35	$120,000	N	22000	2
52	$18,000	N	124000	
23	$95,000	Y	47000	
40	$62,000	Y	80000	
60	$100,000	Y	42000	3
48	$220,000	Y	78000	
33	$150,000	Y	8000	1
48	**$142,000**	**?**		

(Euclidean Distance) $D = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$

Features of *k*-NN

- **K-NN is pretty intuitive and simple**
  - To classify the new data point, the *k*-NN algorithm reads the whole dataset to find *k* nearest neighbors.
- **K-NN has no assumptions**
  - *K*-NN is a non-parametric algorithm, which means there are no assumptions to be met to implement the *k*-NN.
- **No Training Step**
  - *k*-NN does not explicitly build any model; it simply tags the new data entry-based learning from historical data.
- **Very easy to implement for multi-class problem**
- **Can be used both for Classification and Regression**
- **Variety of distance criteria to be chosen from**
  - *K*-NN algorithm gives the user the flexibility to choose distance

Pros and Cons of *k*-NN

- **Pros:**
  - No assumptions about data needed
  - Simple algorithm
  - High accuracy (relatively) but not accurate as better supervised learning models.
  - Versatile – useful for classification or regression.
- **Cons:**
  - Prediction stage might be slow for a large size of neighbors

References

- https://www.fromthegenesis.com/pros-and-cons-of-k-nearest-neighbors/
- https://www.geeksforgeeks.org/k-nearest-neighbours/
- https://www.saedsayad.com/k_nearest_neighbors.htm

Clustering

- In **supervised learning**, the target features that must be predicted from input features are observed in the training data.
- In **clustering** or **unsupervised learning**, the **training examples do not give the target (output) features**.
  - The aim is to construct a natural classification that can be used to **cluster** the data.
    - An **intelligent tutoring** system may want to cluster students' learning behavior so that strategies that work for one class member may work for others.

Clustering

- The general idea behind **clustering** is to partition the examples into **clusters** or **classes.**
  - Each class predicts feature values for the examples in the class.
  - Each clustering has a prediction error on the predictions.
  - The best clustering is the one that minimizes the error.

Clustering

- In **hard clustering**, each example is placed definitively in a class.
  - The class is then used to predict the feature values of the example.
- In **soft clustering**, each example has a probability distribution over its class.
  - The prediction of the values for the features of an example is the weighted average of the predictions of the classes the example is in, weighted by the probability of the example being in the class.

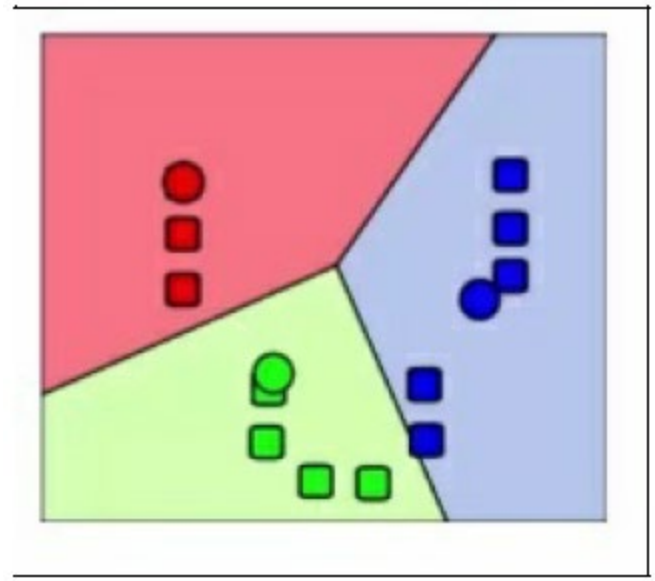
K-Means Clustering

- The ***K*-Means clustering** algorithm is used for **hard clustering**
  - This is an **unsupervised learning** technique where you have a collection of stuff that you want to group into various clusters.
  - This is a widespread technique in **machine learning** where you try to take a bunch of data and find interesting clusters of things based on the **attributes** of the data itself.
  - The **training examples** (**data set**) and the number of classes, ***k***, is given as **input**.
  - The **output** is a set of ***k*** classes, a prediction of a value for each feature for each class, and an assignment of examples to classes.

K-Means Clustering

- Here, all we do in **K-means** clustering is trying to split our data into ***k*** groups - that's where the ***k*** comes from; it’s how many different groups you're trying to break your data into - and it does this by finding ***k*** centroids (means).

So, basically, what group a given data point belongs to is defined by which of these centroid points it's closest to in your scatter plot. You can visualize this in the following image:



K-Means Clustering

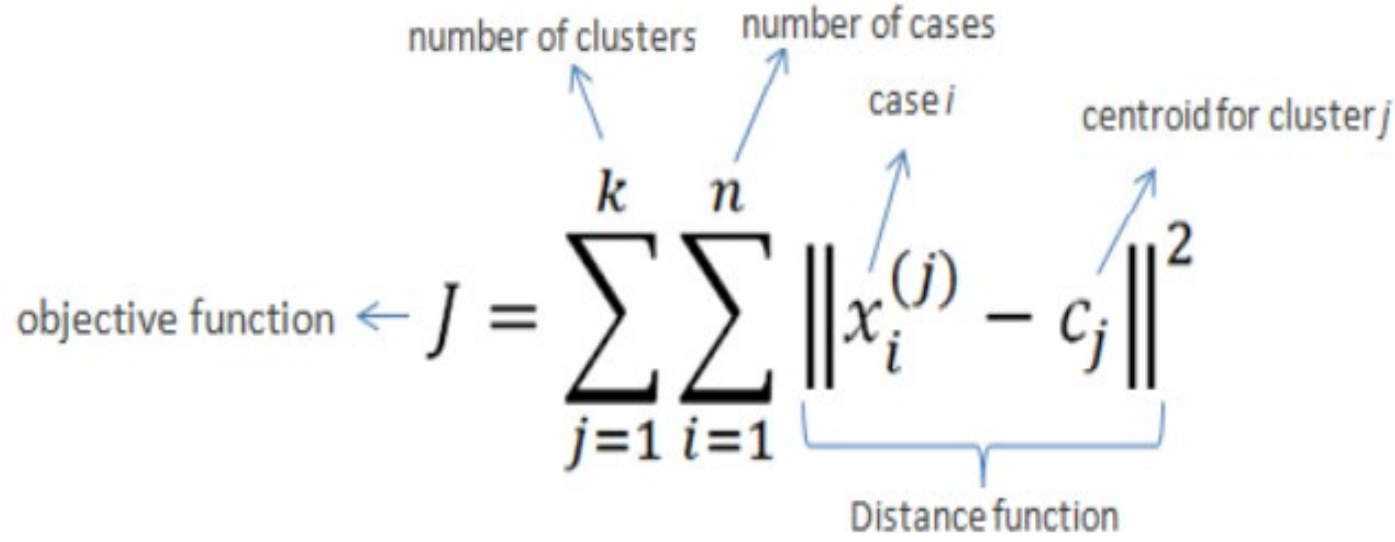
- This is showing an example of **K-means clustering** with ***k*** of 3 (***k*** = 3), and the squares represent data points in a scatter plot.
  - The **circles** represent the **centroids** that the ***k***-means clustering algorithm came up with, and each point is assigned a cluster based on which centroid it's **closest to**.
  - So that's all there is to it, really. It's an example of **unsupervised learning**.
  - Instead of clustering a given dataset based on it’s training data; rather, you're just given the data itself and it tries to converge on these clusters naturally just based on the attributes of the data alone.

K-Means Clustering Algorithm

- Here's the algorithm for ***K*-Means clustering**:
  - **Randomly pick *K* centroids (means):** Start with a randomly chosen set of centroids. So if we have a ***K*** of three, we'll look for three clusters in our group.
  - **Assign each data point to the centroid it is closest to:** We then assign each point to the randomly assigned centroid it is closest to.
  - **Recompute the centroids based on the average position of each centroid's points:** This means re-compute the centroid for each cluster we come up with.
  - **Iterate until points stop changing assignment to centroids:** We will do it all again until those centroids stop moving, we hit some threshold value, and we have converged on something here.
  - **Predict the cluster for new points:** To predict the clusters for further points.

K-Means Clustering Algorithm

- **K-Means** clustering intends to partition ***n*** objects into ***k*** clusters in which each object belongs to the cluster with the **nearest mean**.
- This method produces exactly ***k*** different clusters of the greatest possible distinction.
- The best number of clusters ***k*** leading to the greatest separation (distance) is not known as a prior and must be computed from the data.
- The objective of ***K*-Means** clustering is to minimize total intra-cluster variance, or, the squared error function:



Where x_i is a data point in dataset ***i***, C_j is the cluster mean(the center of cluster ***j***)

K-Means Clustering Algorithm

- ***K*-Means clustering algorithm**:
  1. Choose a value of ***k***, the number of clusters to be formed.
  2. Randomly select ***k*** data points from the data set as the initial cluster centroids/centers.
  3. For each data point:
     3.1. Compute the distance (**Euclidean distance**) between the data point and the cluster centroid
     3.2. Assign the data point to the **closest** centroid.
  4. For each cluster, calculate the **new mean** based on the cluster's data points.
  5. Repeat 3 & 4 steps until the mean of the **clusters stop changing** or the maximum number of iterations is reached.

K-Means Clustering Algorithm - Example

- Apply *k*-means on the following 1 dimensional dataset for ***k* = 2** (for two clusters) and **dataset** = {2, 4, 10, 12, 3, 20, 30, 11, 25}
- **<u>Iteration 1</u>**

  Let M1, and M2 are the two randomly selected centroids (means) where M1= 4, M2=11 and the initial clusters are C1= {4}, C2= {11}.

  Calculate the Euclidean distance as:

  Distance D = [x,a]=$\sqrt{(x-a)^2}$

  D1 is the distance from M1 and D2 is the distance from M2

K-Means Clustering Algorithm - Example

As we can see in the above table, 2 data-points are added to cluster C1 and other data-points are added to cluster C2:

C1 = {2, 4, 3} and

C2 = {10, 12, 20, 30, 11, 25}

Datapoint	D1	D2	Cluster
2	2	9	C1
4	0	7	C1
10	6	1	C2
12	8	1	C2
3	1	8	C1
20	16	9	C2
30	26	19	C2
11	7	0	C2
25	21	14	C2

K-Means Clustering Algorithm - Example

- **Iteration 2**

Calculate the new mean of data-points in C1 and C2:

M1 = (2 + 4 + 3)/3 = 3 and

M2 = (10 + 12 + 20 + 30 + 11 + 25)/6 = 18

Calculating distance D1 and D2:

$$D1 = \sqrt{(data - M1)^2}$$

$$D2 = \sqrt{(data - M2)^2}$$

Calculated distances (D1 and D2) and updated clusters are shown in the Table below:

K-Means Clustering Algorithm - Example

As we can see in the above table, there are 4 datapoints are added in cluster C1 and other datapoints are in cluster C2:

C1 = {2, 4, 3, 10} and

C2 = {12, 20, 30, 11, 25}

Datapoint	D1	D2	Cluster
2	1	16	C1
4	1	14	C1
3	0	15	C1
10	7	8	C1
12	9	6	C2
20	17	2	C2
30	27	12	C2
11	8	7	C2
25	22	7	C2

K-Means Clustering Algorithm - Example

- **Iteration 3**

Calculate new mean of datapoints in C1 and C2:

M1 = (2 + 4 + 3 + 10)/4 = 4.75 and

M2 = (12 + 20 + 30 + 11 + 25)/5 = 19.6

Calculating distance D1 and D2:

$$D1 = \sqrt{(data - M1)^2}$$

$$D2 = \sqrt{(data - M2)^2}$$

Calculated distances (D1 and D2) and updated clusters are shown in the Table below:

K-Means Clustering Algorithm - Example

As we can see in the above table, there are 6 datapoints are added in cluster C1 and other datapoints are in cluster C2:

C1 = {2, 4, 3, 10, 12, 11} and

C2 = {20, 30, 25}

Datapoint	D1	D2	Cluster
2	2.75	17.6	C1
4	0.75	15.6	C1
3	1.75	16.6	C1
10	5.25	9.6	C1
12	7.25	7.6	C1
20	15.25	0.4	C2
30	25.25	10.4	C2
11	6.25	8.6	C1
25	20.25	5.4	C2

K-Means Clustering Algorithm - Example

- **Iteration 4**

Calculate new mean of datapoints in C1 and C2:

M1 = (2 + 4 + 3 + 10 +12 +11)/6 = 7 and

M2 = (20 + 30 + 25)/3 = 25

Calculating distance D1 and D2:

$$D1 = \sqrt{(data - M1)^2}$$

$$D2 = \sqrt{(data - M2)^2}$$

Calculated distances (D1 and D2) and updated clusters are shown in the Table below:

K-Means Clustering Algorithm - Example

As we can see in the above table, there are 6 datapoints are added in cluster C1 and other datapoints are in cluster C2:

C1 = {2, 4, 3, 10, 12, 11} and

C2 = {20, 30, 25}

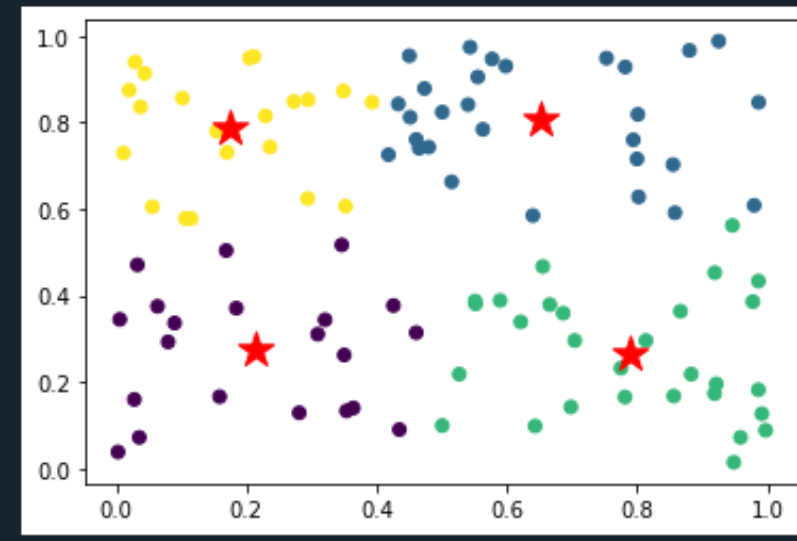
- There are same datapoints in both clusters after the iterations 3 and 4. This becomes the stopping criterion of this algorithm.

Datapoint	D1	D2	Cluster
2	5	23	C1
4	3	21	C1
3	4	22	C1
10	3	15	C1
12	5	13	C1
11	4	14	C1
20	13	5	C2
30	23	5	C2
25	18	0	C2

K-Means Clustering: Python

(PDF) The Fundamentals of Machine Learning (researchgate.net)

```python
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import numpy as np
# Generate random data points
X = np.random.rand(100, 2)
# Specify number of clusters
k = 4
# Initialize k-means algorithm
kmeans = KMeans(n_clusters=k)
# Fit the algorithm to the data
kmeans.fit(X)
# Get the cluster labels and centroids
labels = kmeans.labels_
centroids = kmeans.cluster_centers_
# Visualize the clusters
plt.scatter(X[:,0], X[:,1], c=labels)
plt.scatter(centroids[:,0], centroids[:,1], marker='*', s=300, c='r')
plt.show()
```



K-Means Clustering Algorithm – Home Work

- Suppose we want to group the visitors to a website using just their age (one-dimensional space) as follows (k = 2):
  - Ages: 5,15,16, 19, 19, 20, 20, 21, 22, 28, 35, 40, 41, 42, 43, 44, 60, 61, 65
  - Initial clusters are (randomly selected centroids) are M1, and M2 (where M1 < M2). means that the initial clusters are C1 = {M1} and C2 = {M2}.
  - Based on the above parameters, show the iterations used by the k-means clustering algorithm (without using its Python library.)

- Modify the about program for **k = 3**.

Rules: If (D1 <= D2) and (D1 <= D3), then **C1 = data**
Else if (D2 <= D1) and (D2 <= D3), then **C2 = data**
Else if (D3 <= D1) and (D3 <= D2), then **C3 = data**

K-Means Clustering- Performance

- **K-Means** is a relatively efficient method.
- However, we need to specify the number of **clusters** in advance, and the final results are sensitive to initialization and often terminate at a **local optimum**.
  - Unfortunately, no global theoretical method exists to find the optimal number of clusters.
- A practical approach is to compare the outcomes of multiple runs with different ***k*** and choose the best one based on a predefined criterion.
  - In general, a large ***k*** probably decreases the error but increases the risk of overfitting.

Drawbacks of *K*-Means algorithm

- **The result might not be globally optimal:** We can't assure that this algorithm will lead to the best global solution. Selecting different random seeds at the beginning affects the final results.
- **Value of *K* needs to be specified beforehand:** We can expect this value only if we have a good idea about our dataset, and if we are working with a new dataset, then the **elbow method** can be used to determine the value of ***K***.
- **Works only for linear boundaries: *K*-means** makes this assumption that the boundaries will always be linear. Hence it fails when it comes to complicated boundaries.
- **Slow for large samples:** As this algorithm accesses each dataset point, it becomes slow when the sample size grows.