IITU

Neural Networks

Compiled by
G. Pachshenko

Pachshenko Galina Nikolaevna

Associate Professor of Information System Department,

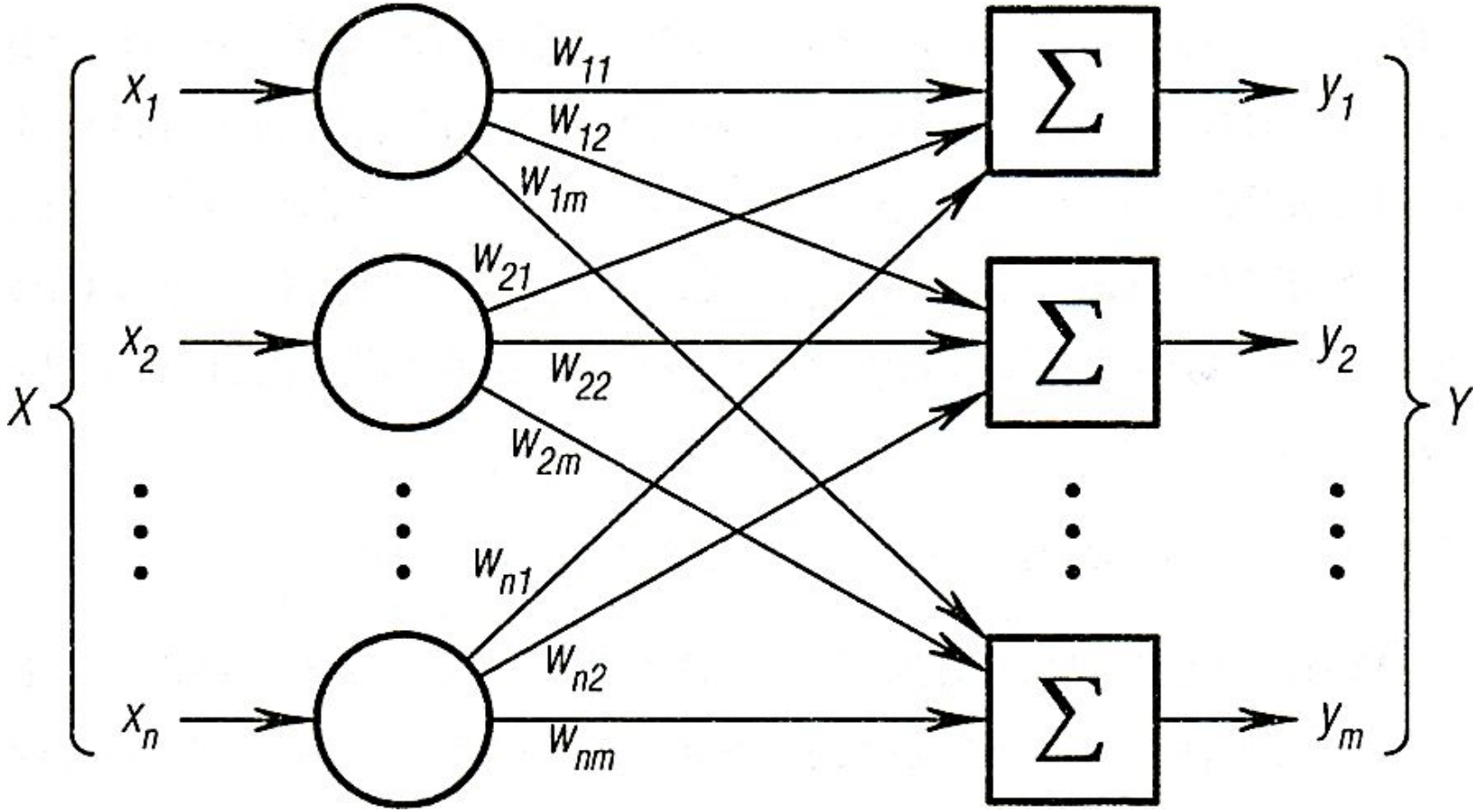
Candidate of Technical Science

Week 4
Lecture 4

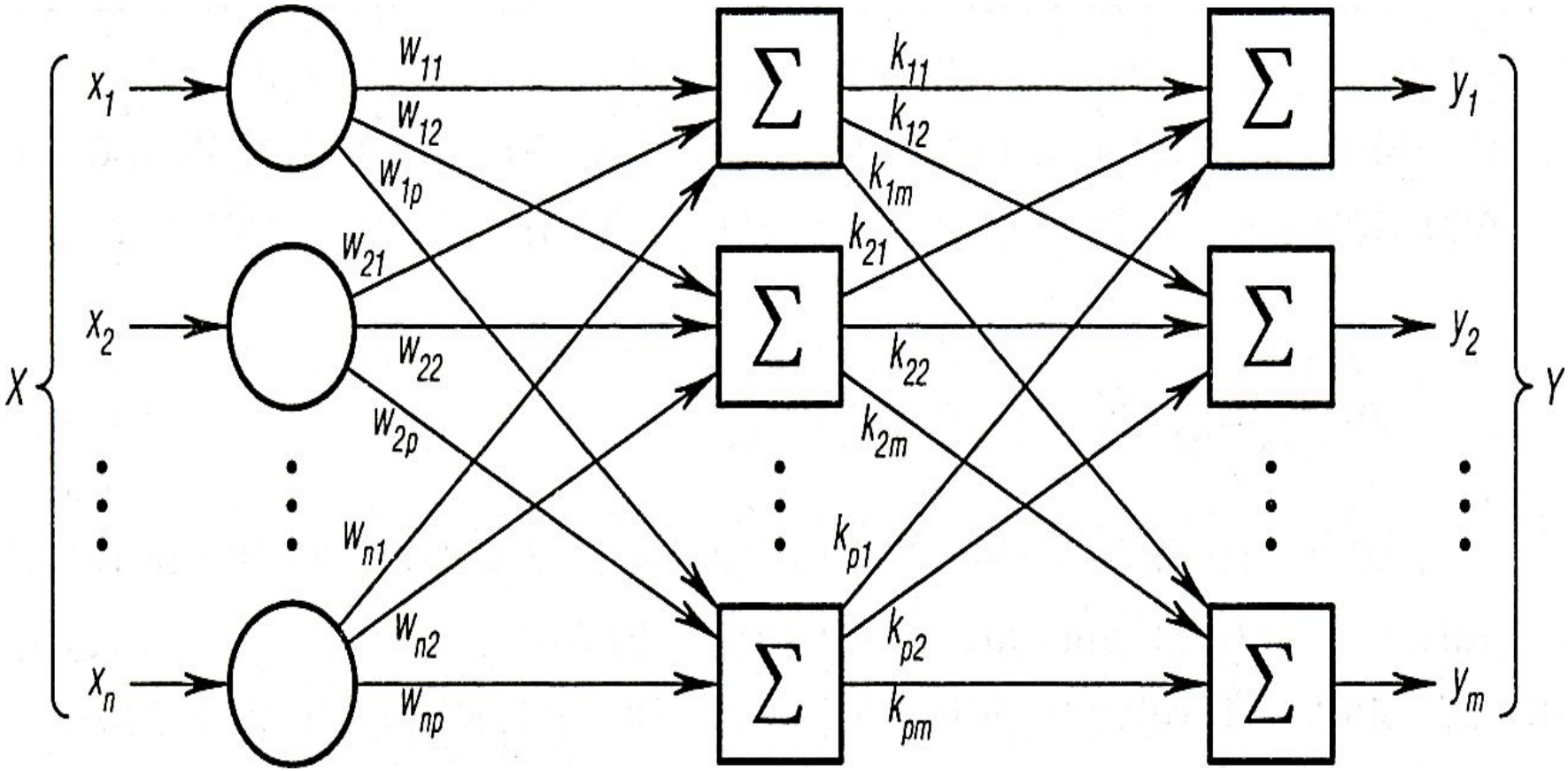
Topics

- **Single-layer neural networks**
- **Multi-layer neural networks**
- **Single perceptron**
- **Multi-layer perceptron**
- **Hebbian Learning Rule**
- **Back propagation**
- **Delta-rule**
- **Weight adjustment**
- **Cost Function**
- **Classification (Independent Work)**

Single-layer neural networks



Multi-layer neural networks



Single perceptron

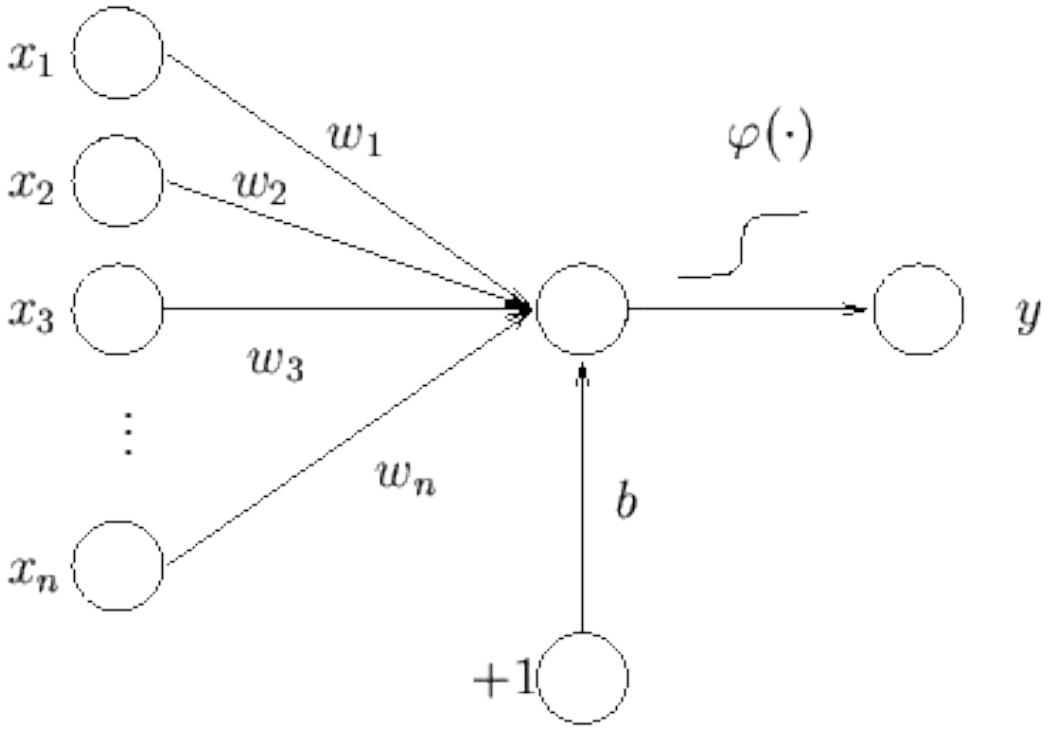
The perceptron computes a single *output* from multiple real-valued *inputs* by forming a linear combination according to its input *weights* and then possibly putting the output through activation function.

Single perceptron.

Mathematically this can be written as

$$y = \varphi(\sum_{i=1}^{n} w_i x_i + b) = \varphi(\mathbf{w}^T \mathbf{x} + b)$$

Single perceptron.



Task 1:

Write a program that finds output of a single perceptron.

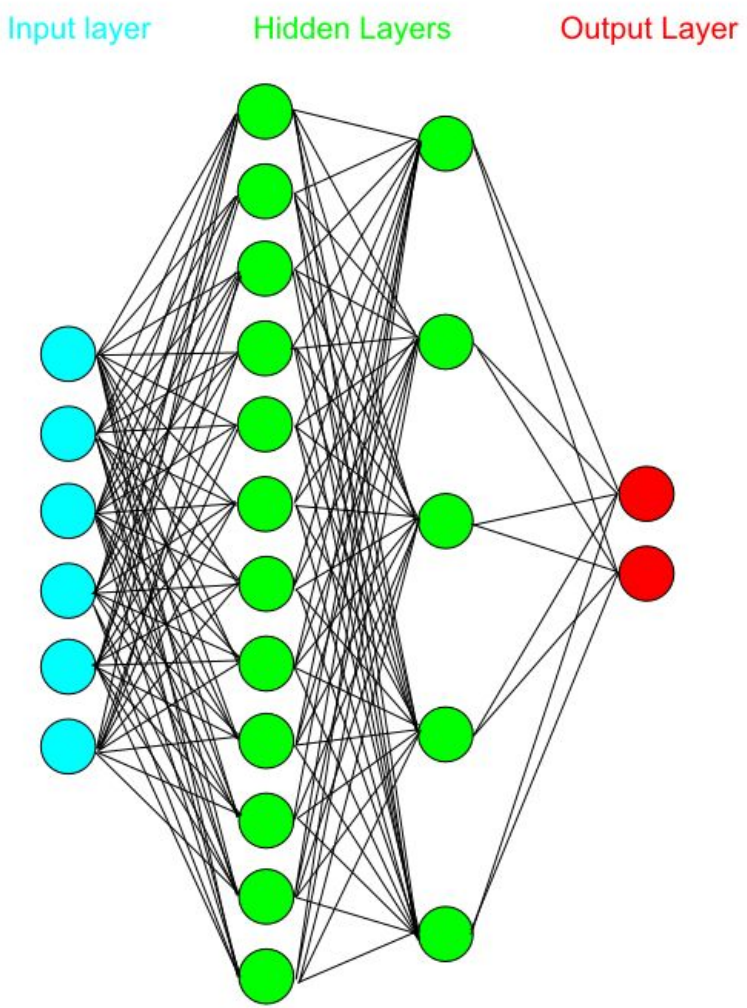
Note:

Use *bias*. The bias shifts the decision boundary away from the origin and does not depend on any input value.

Multilayer perceptron

A **multilayer perceptron** (MLP) is a class of feedforward artificial neural network.

Multilayer perceptron



Structure

- nodes that are no target of any connection are called **input neurons**.

- nodes that are no source of any connection are called **output neuron**s.

A MLP can have more than one output neuron.

The **number of output** neurons depends on the way the target values (desired values) of the training patterns are described.

- all nodes that are neither input neurons nor output neurons are called **hidden neurons**.

- all neurons can be organized in layers, with the set of input layers being the first layer.

The original Rosenblatt's perceptron used a *Heaviside step function* as the activation function.

Nowadays, in multilayer networks, the *activation function* is often chosen to be the *sigmoid function*

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

or the hyperbolic tangent

$$\tanh(x)$$

They are related by

$$(\tanh(x) + 1)/2 = 1/(1 + e^{-2x})$$

These functions are used because they are mathematically convenient.

An MLP consists of at least three layers of nodes.

Except for the input nodes, each node is a neuron that uses a *nonlinear activation function*.

MLP utilizes a supervised learning technique called **backpropagation** for training.

Hebbian Learning Rule

Delta rule

Backpropagation algorithm

$$w_{ij}(k+1) = w_{ij}(k) + \Delta w_{ij}, \quad \Delta w_{ij} = \eta y_i(k) y_j(k)$$

Hebbian Learning Rule (Hebb's rule)

The Hebbian Learning Rule (1949)

is a learning rule that specifies how much the weight of the connection between two units should be increased or decreased in proportion to the product of their activation.

Hebbian Learning Rule (Hebb's rule)

$$w_{ij}(k+1) = w_{ij}(k) + \Delta w_{ij}, \quad \Delta w_{ij} = \eta y_i(k) y_j(k)$$

The Hebb Rule

$$\Delta w_{i,j} = \mu a_i a_j$$

- The change in weight from unit *i* to *j* is the product of the two units' activities, and a scaling factor *u*
  - If both units' activities are positive, or both negative, weight goes up
  - If signs are opposite, weight goes down

Delta rule
(proposed in 1960)

$$\delta = T - OUT$$

$$w_N(i+1) = w_N(i) + \eta \delta x_N$$

The **backpropagation algorithm** was originally introduced in the 1970s, but its importance wasn't fully appreciated until a famous 1986 paper by David Rumelhart, Geoffrey Hinton, and Ronald Williams.

That paper describes several neural networks where backpropagation works far faster than earlier approaches to learning, making it possible to use neural nets to solve problems which had previously been insoluble.

Supervised **Backpropagation** – The mechanism of backward error transmission (**delta learning rule**) is used to modify the weights of the internal (hidden) and output layers

Back propagation

The back propagation learning algorithm uses the **delta-rule**.

What this does is that it computes the **deltas**, (local gradients) of each neuron starting from the output neurons and going backwards until it reaches the input layer.

The delta rule is derived by attempting *to minimize the error* in the output of the neural network through gradient descent.

To compute the *deltas* of the output neurons though we first have to get the *error of each output* neuron.

That’s pretty simple, since the multi-layer perceptron is a supervised training network so the error is the difference between the network’s output and the desired output.

$$\mathbf{e_j(n) = d_j(n) - o_j(n)}$$

where e(n) is the error vector, d(n) is the desired output vector and o(n) is the actual output vector.

Now to compute the deltas:

$$\mathbf{delta_j^{(L)}(n) = e_j^{(L)}(n) * f'(u_j^{(L)}(n))} \ ,$$

for neuron j in the *output* layer L

where $f'(u_j^{(L)}(n))$ is the *derivative* of the *value* of the jth neuron of layer L

The same formula:

$$\delta_q = OUT_q \left(1 - OUT_q\right)\left(T_q - OUT_q\right)$$

Weight adjustment

Having calculated the deltas for all the neurons we are now ready for the third and final pass of the network, this time to *adjust the weights* according to the generalized delta rule:

Weight adjustment

$$w_{p\text{-}q}(i+1) = w_{p\text{-}q}(i) + \eta \delta_q OUT_p$$

For $w_{B1\text{-}C1}$

$$\delta_{C1} = OUT_{C1}\left(1 - OUT_{C1}\right)\left(T_1 - OUT_{C1}\right)$$

$$w_{B1\text{-}C1}\left(i+1\right) = w_{B1\text{-}C1}\left(i\right) + \eta\delta_{C1}OUT_{B1}$$

Note: For sigmoid activation function

Derivative of the function:

$$S'(x) = S(x)*(1 - S(x))$$

Derivative of Sigmoid function

$$y = \frac{1}{1+e^{-x}}$$

$$\frac{dy}{dx} = -\frac{1}{(1+e^{-x})^2}(-e^{-x}) = \frac{e^{-x}}{(1+e^{-x})^2}$$

$$= \frac{1}{1+e^{-x}}\left(1-\frac{1}{1+e^{-x}}\right) = y(1-y)$$

Cost Function

We need a function that will minimize the parameters over our dataset. One common function that is often used is mean squared error

$$E(W) = \frac{1}{2}\sum_{k=1}^{K}(d_k - y_k)^2$$

- Squared Error: which we can minimize using gradient descent
- A cost function is something you want to minimize. For example, your cost function might be the *sum of squared errors over your training set*. Gradient descent is a method for finding the minimum of a function of multiple variables. So you can use gradient descent to minimize your cost function.

Back-propagation is a gradient descent over the entire networks weight vectors.

In practice, it often works well and can run multiple times. It minimizes error over all training samples.

Task 2:

Write a program that can update weights of neural network using backpropagation.

Thank you for your attention!