# Chapter 10: Artificial Neural Networks

Dr. Xudong Liu
Assistant Professor
School of Computing
University of North Florida

Monday, 9/30/2019

# Overview

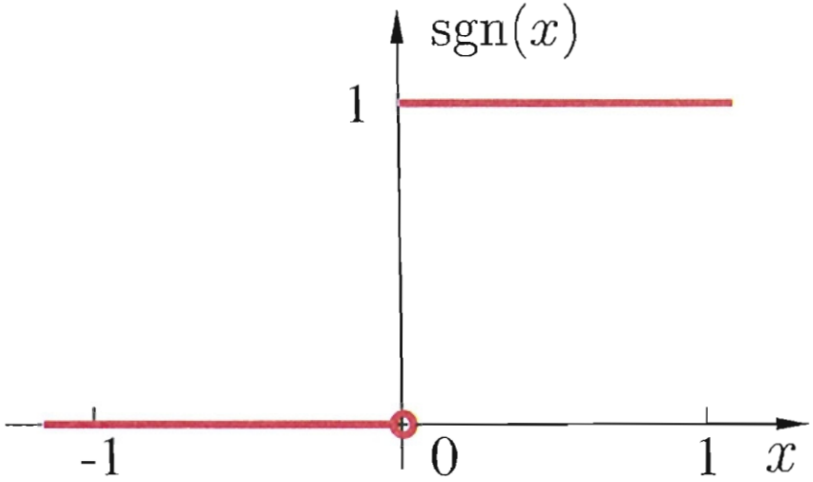
1. Artificial neuron: linear threshold unit (LTU)
2. Perceptron
3. Multi-Layer Perceptron (MLP), Deep Neural Networks (DNN)
4. Learning ANN's: Error Backpropagation

# Differential Calculus

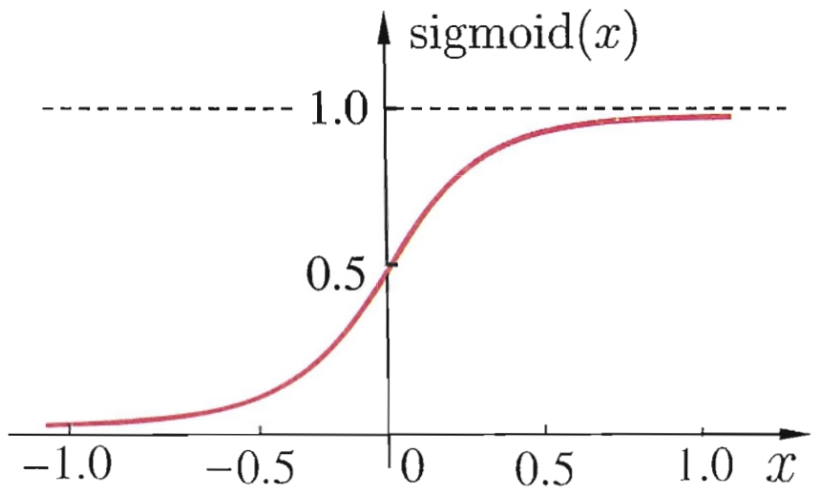
- In this course, I shall stick to Leibniz's notation.
- The *derivative* of a function at an input point, when it exists, is the slope of the tangent line to the graph of the function.
  - Let $y = f(x) = x^2 + 2x + 1$. Then, $f'(x) = \frac{dy}{dx} = 2x + 2$.
- A *partial derivative* of a function of several variables is its derivative with respect to one of those variables, with the others held constant.
  - Let $z = f(x, y) = x^2 + xy + y^2$. Then, $\frac{\partial z}{\partial x} = 2x + y$.
- The *chain rule* is a formula for computing the derivative of the composition of two or more functions.
  - Let $z = f(y)$ and $y = g(x)$. Then, $\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} = f'(y) \cdot g'(x)$.
- Nice thing about the *sigmoid function* $f(x) = \frac{1}{1+e^{-x}}$:
  - $f'(x) = f(x) \cdot (1 - f(x))$.

# Artificial Neuron

- *Artificial neuron*, also called *linear threshold unit* (LTU), by McCulloch and Pitts, 1943: with one or more numeric inputs, it produces a weighted sum of them, applies an *activation function*, and outputs the result.
- Common activation functions: step function and sigmoid function.



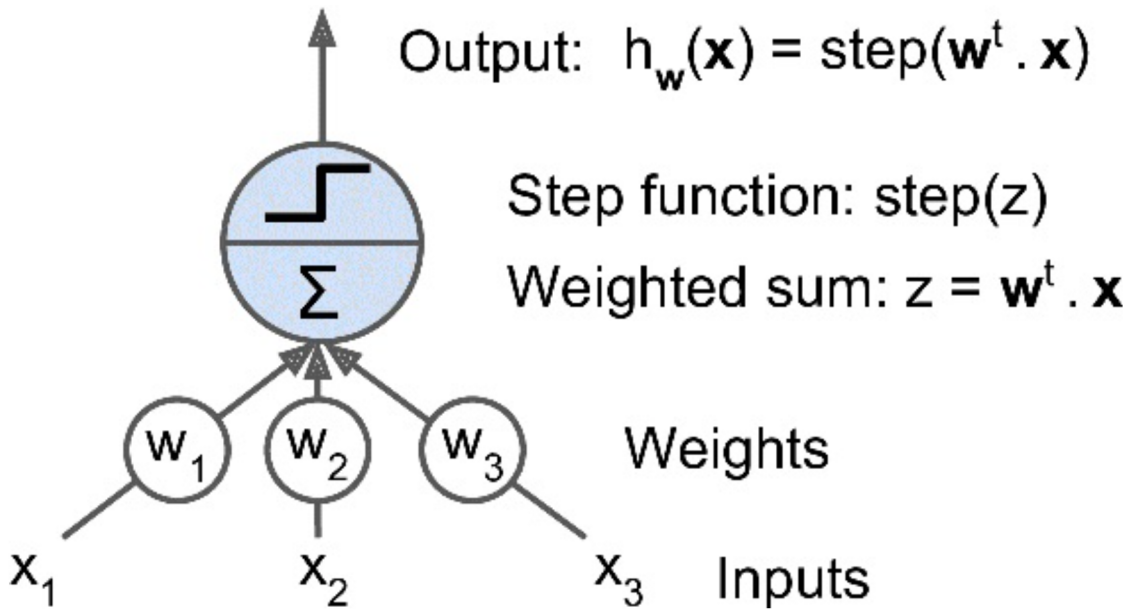

$$\operatorname{sgn}(x) = \begin{cases} 1, & x \geqslant 0; \\ 0, & x < 0. \end{cases}$$




$$\operatorname{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

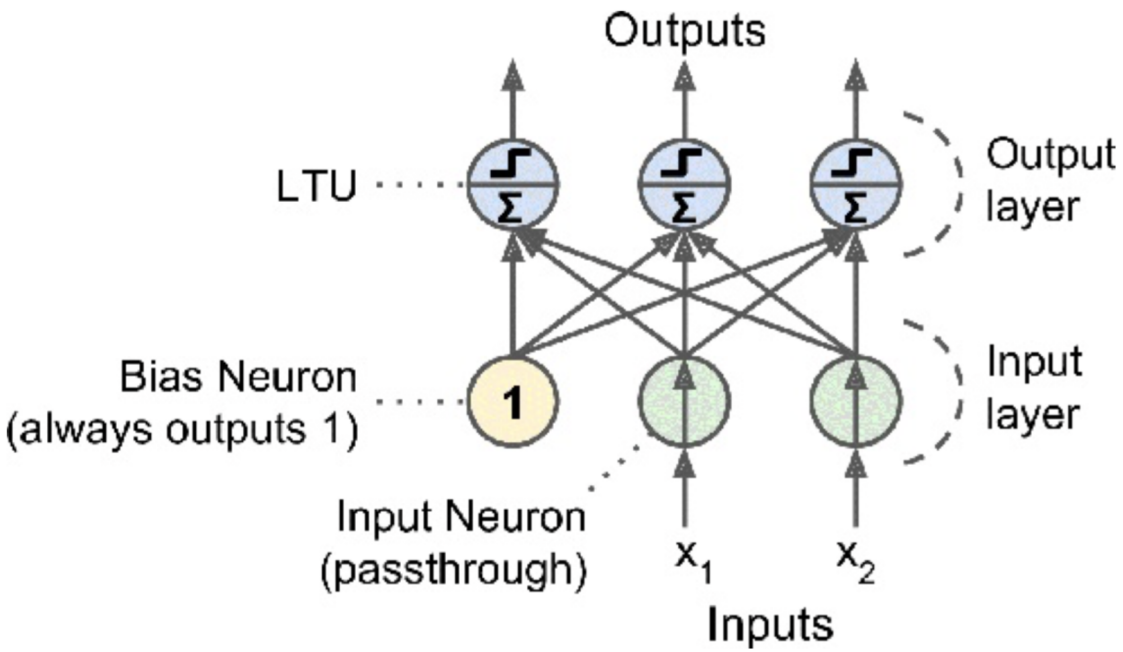
# Linear Threshold Unit (LTU)

- Below is an LTU with the activation function being the step function.



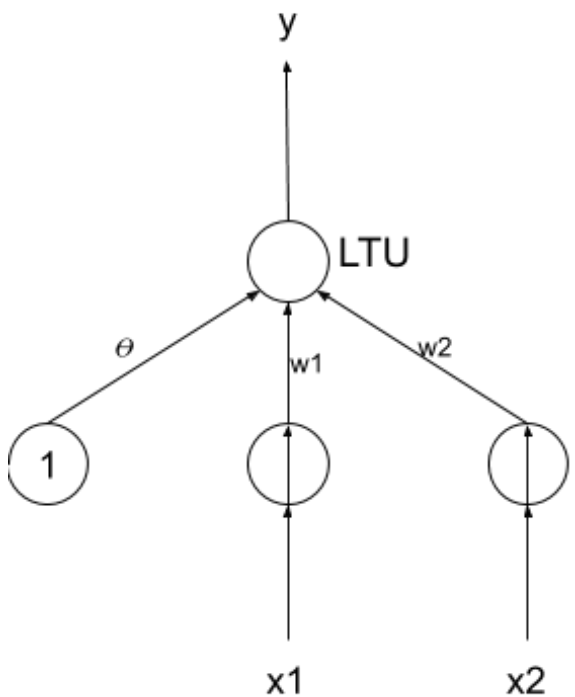
# Perceptrons

- A *perceptron*, by Rosenblatt in 1957, is composed of two layers of neurons: an input layer consisting of special passing through neurons and an output layer of LTU's.
- The bias neuron is added for the completeness of linearity.
- Rosenblatt proved that, if training examples are linearly separable, a perceptron always can be learned to correctly classify all training examples.



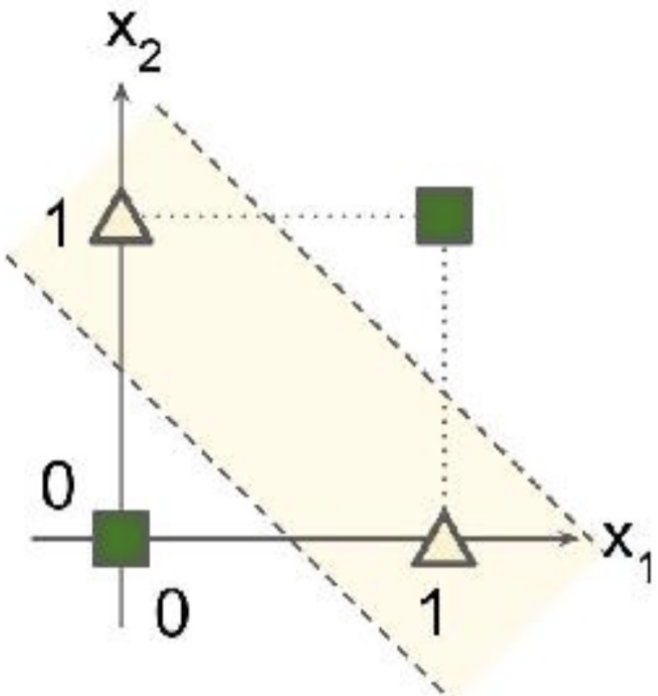
# Perceptrons

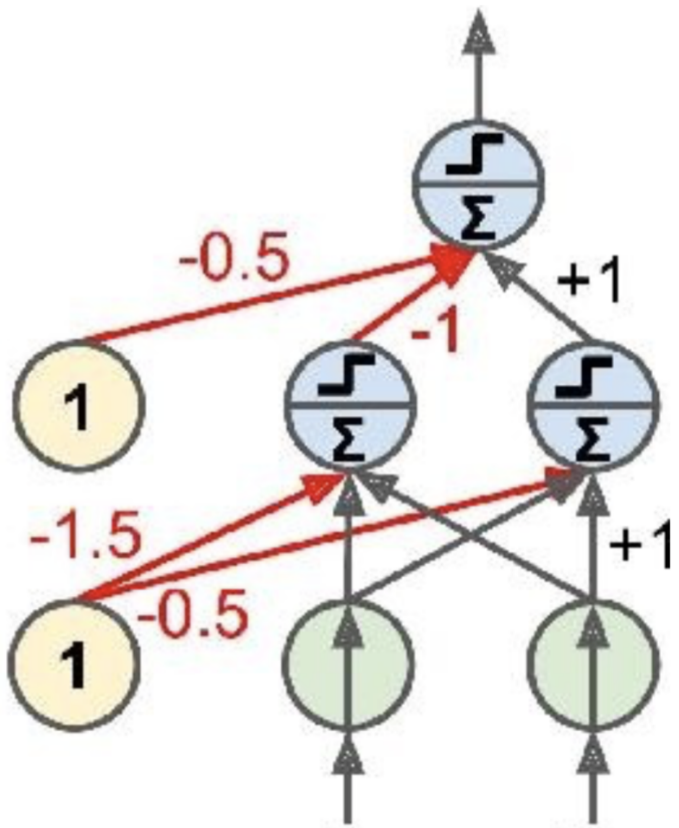
- For instance, perceptrons can implement logical conjunction, disjunction and negation.
- For the following perceptron of one LTU with the step function as the activation function.
  - $x_1 \wedge x_2$: $w_1 = w_2 = 1$, $\theta = -2$
  - $x_1 \vee x_2$: $w_1 = w_2 = 1$, $\theta = -0.5$
  - $\neg x_1$: $w_1 = -0.6$, $w_2 = 0$, $\theta = -0.5$



# Perceptrons

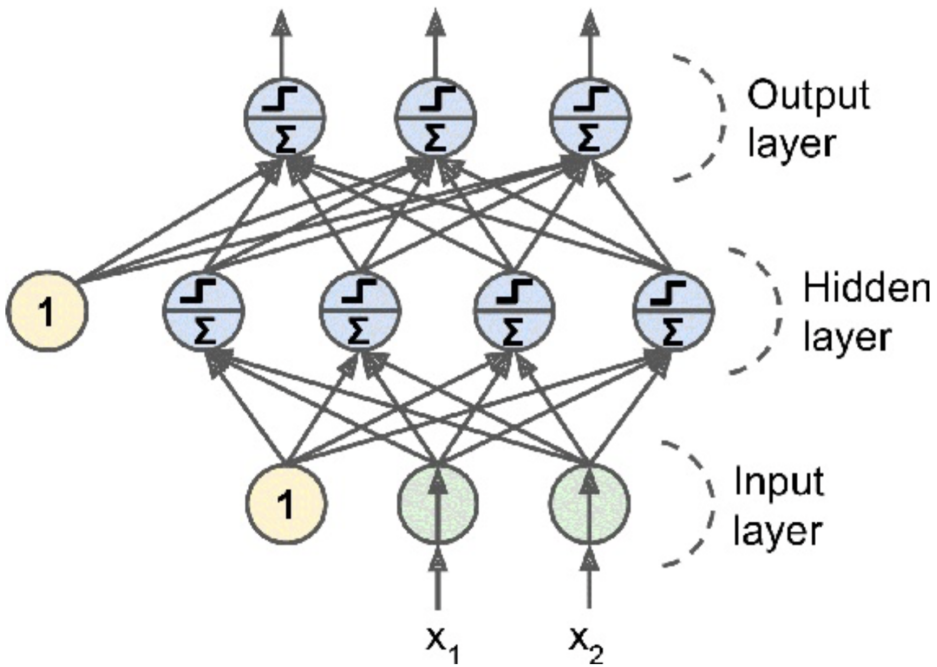
- However, perceptrons cannot solve some trivial non-linear separable problems, such as the Exclusive OR classification problem.
- This is shown by Minsky and Papert in 1969.
- Turned out stacking multiple perceptrons can solve any non-linear problems.



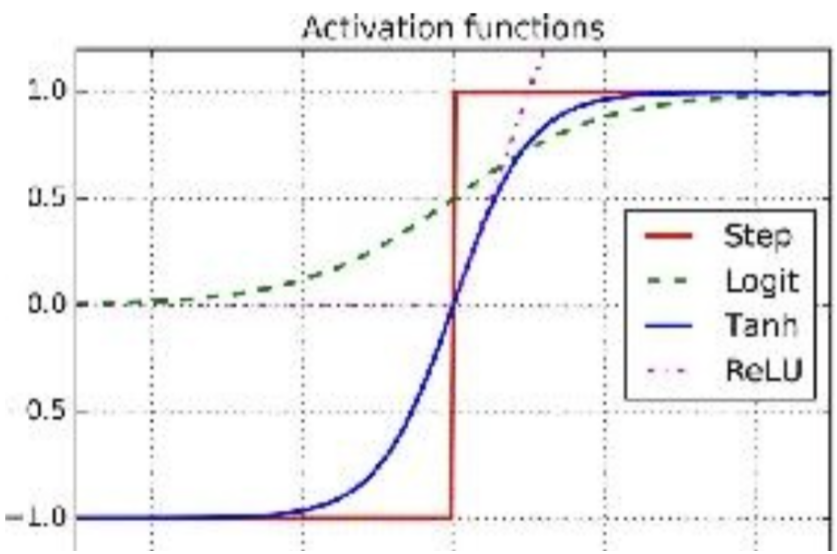
# Multi-Layer Perceptrons

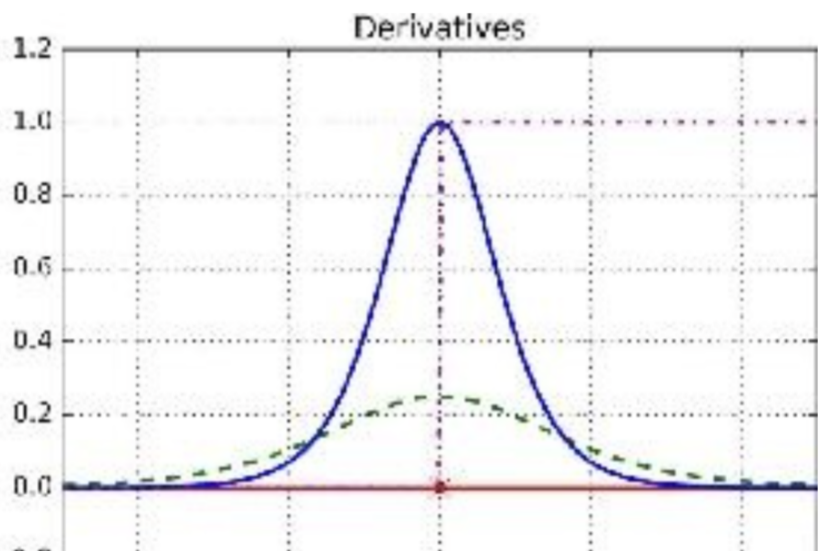
- A *Multi-Layer Perceptrons* (MLP) is composed of one passthrough input layer, one or more layers of LTU's, called *hidden layer*, and one final layer of LTUs, called *output layer*.
- Again, every layer except the output layer includes a bias neuron and is fully connected to the next layer.
- When an MLP has two or more hidden layers, it is called a *deep neural network* (DNN).



# Learning Multi-Layer Perceptrons: Error Backpropagation

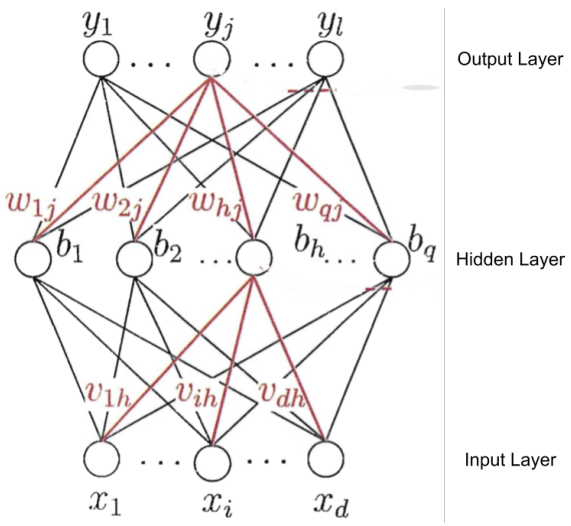
- Error backpropagation so far is the most successful learning algorithm for training MLP's.
- *Learning Internal Representations by Error Propagation*, Rumelhart, Hinton and Williams, 1986.
- Idea: we start with a fix network, then update edge weights using $v \leftarrow v + \Delta v$, where $\Delta v$ is a gradient descent of some error objective function.
- Before learning, let's take a look at a few possible activation functions and their derivatives.



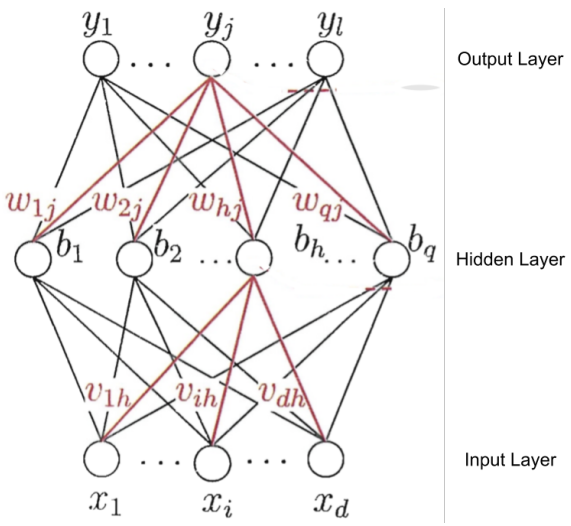
# Error Backpropagation

- Now, I explain the error backpropagation algorithm for the following MLP of $d$ passthrough input neurons, one hidden layer of $q$ LTU's, and output layer of $l$ LTU's.
- I will assume activation functions $f$ at the $l + q$ LTU's all are the sigmoid function.
- Note the bias neuron is gone from the picture, as now the bias is embedded in the LTU's activation function $y_j = f(\sum_i w_i x_i - \theta_j)$.



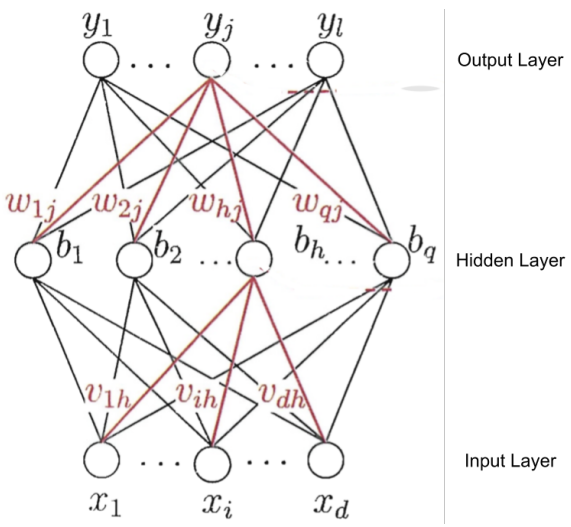
# Error Backpropagation

- So our training set is $\{(\boldsymbol{x_1}, \boldsymbol{y_1}), \ldots, (\boldsymbol{x_m}, \boldsymbol{y_m})\}$, where $\boldsymbol{x_i} \in \mathbb{R}^d$ and $\boldsymbol{y_i} \in \mathbb{R}^l$.
- We want to have an MLP like below to fit this data set; namely, to compute the $l \cdot q + d \cdot q$ weights and the $l + q$ biases.
- To predict on a new example $\boldsymbol{x}$, we feed it to the input neurons and collect the outputs. The predicted class could be the $y_j$ with the highest score. If you want class probabilities, consider softmax.
- MLP can be used for regression too, when there only is one output neuron.



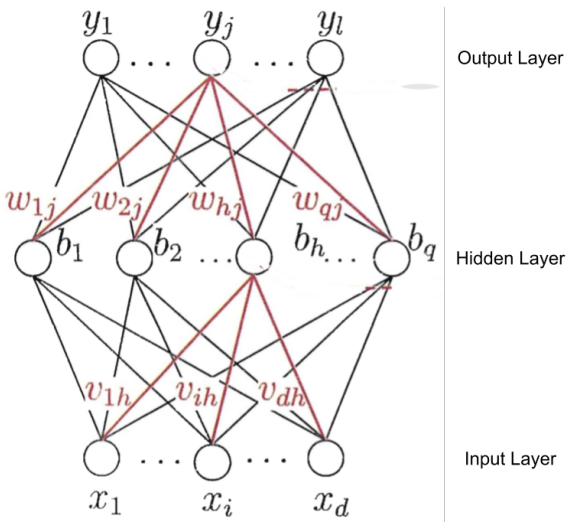
# Error Backpropagation

- I use $\theta_j$ to mean the bias in the $j$-th neuron in the output layer, and $\gamma_h$ the bias in the $h$-th neuron in the hidden layer.
- I use $\beta_j = \sum_{h=1}^{q} w_{hj} b_h$ to mean the input to the $j$-th neuron in the output layer, and $\alpha_h = \sum_{i=1}^{d} v_{ih} x_i$.
- Take a training example $(\boldsymbol{x}, \boldsymbol{y})$, I use $\hat{y} = (\hat{y}_1, \ldots, \hat{y}_l)$ to mean the predicted output, where each $\hat{y}_j = f(\beta_j - \theta_j)$.



# Error Backpropagation

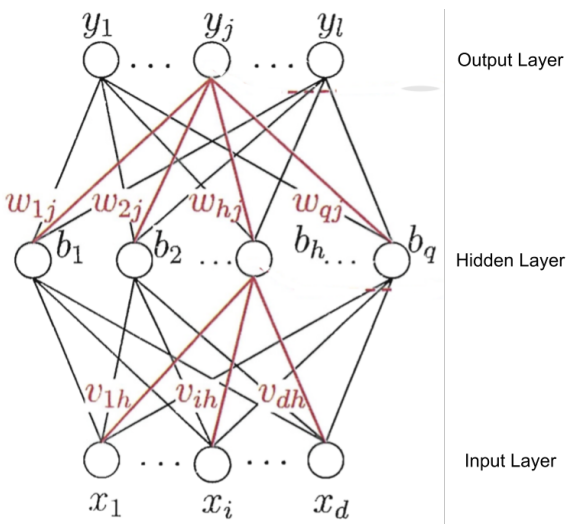
- Error function (objective function): $E = \frac{1}{2}\sum_{j=1}^{l}(\hat{y}_j - y_j)^2$.
- This error function is a composition function of many parameters and it is differentiable, so we can compute the gradient descents to be used to update the weights and biases.



# The Error Backpropagation Algorithm

Given a training set $D = \{(\boldsymbol{x}, \boldsymbol{y})\}$, and learning rate $\eta$, we want to finalize the weights and biases in the MLP.

1. Initialize all the weights and biases in the MLP using random values from interval $[0, 1]$;
2. Repeat the following until some stopping criteron:
   1. For every $(\boldsymbol{x}, \boldsymbol{y}) \in D$, do
      1. Calculate $\hat{\boldsymbol{y}}$;
      2. Calculate gradient descents $\Delta w_{hj}$ and $\Delta \theta_j$ for the neurons in the output layer;
      3. Calculate gradient descents $\Delta v_{ih}$ and $\Delta \gamma_h$ for the neurons in the hidden layer;
      4. Update weights $w_{hj}$ and $v_{ih}$, and biases $\theta_j$ and $\gamma_h$;



# The Error Backpropagation Algorithm: $\Delta w_{hj}$

1. $\Delta w_{hj} = -\eta \frac{\partial E}{\partial w_{hj}}$
2. $\frac{\partial E}{\partial w_{hj}} = \frac{\partial E}{\partial \hat{y}_j} \cdot \frac{\partial \hat{y}_j}{\partial \beta_j} \cdot \frac{\partial \beta_j}{\partial w_{hj}}$
3. We know $\frac{\partial \beta_j}{\partial w_{hj}} = b_h$, for we have $\beta_j = \sum_{h=1}^{q} w_{hj} b_h$
4. We know $\frac{\partial E}{\partial \hat{y}_j} = \hat{y}_j - y_j$, for we have $E = \frac{1}{2} \sum_{j=1}^{l} (\hat{y}_j - y_j)^2$
5. We also know $\frac{\partial \hat{y}_j}{\partial \beta_j} = f'(\beta_j - \theta_j) = \hat{y}_j(1 - \hat{y}_j)$, for we know $f$ is sigmoid
6. Bullets 3, 4 and 5 together can solve $\Delta w_{hj}$ in bullet 1.
7. Computing $\Delta \theta_j$ is similar.

# The Error Backpropagation Algorithm: $\Delta v_{ih}$

1. $\Delta v_{ih} = -\eta \frac{\partial E}{\partial v_{ih}}$
2. $\frac{\partial E}{\partial v_{ih}} = \frac{\partial E}{\partial b_h} \cdot \frac{\partial b_h}{\partial \alpha_h} \cdot \frac{\partial \alpha_h}{\partial v_{ih}}$
3. (Long story short)
4. $\Delta v_{ih} = \eta x_i b_h (1 - b_h) \sum_{j=1}^{l} w_{hj} g_j$, where $g_j = (y_j - \hat{y}_j)\hat{y}_j(1 - \hat{y}_j)$
5. So we update $\Delta w_{hj}$ and $\Delta \theta_j$ for the output layer first, then $\Delta v_{ih}$ and $\Delta \gamma_h$ for the hidden layer.
6. This is why it is called *backpropagation*.