# Artificial Neural Network

School of Computer Science

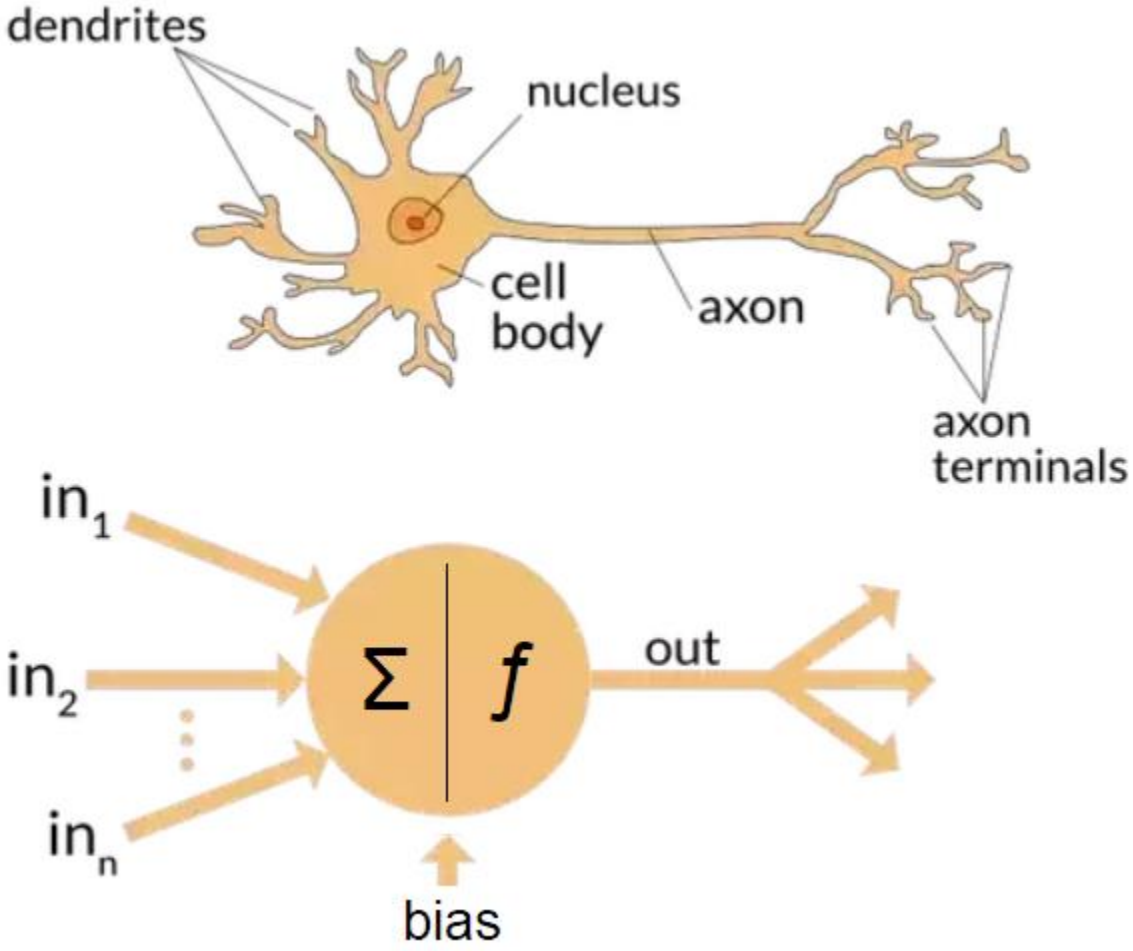
National University of Computing & Emerging Sciences

Karachi Campus

# Artificial Neuron

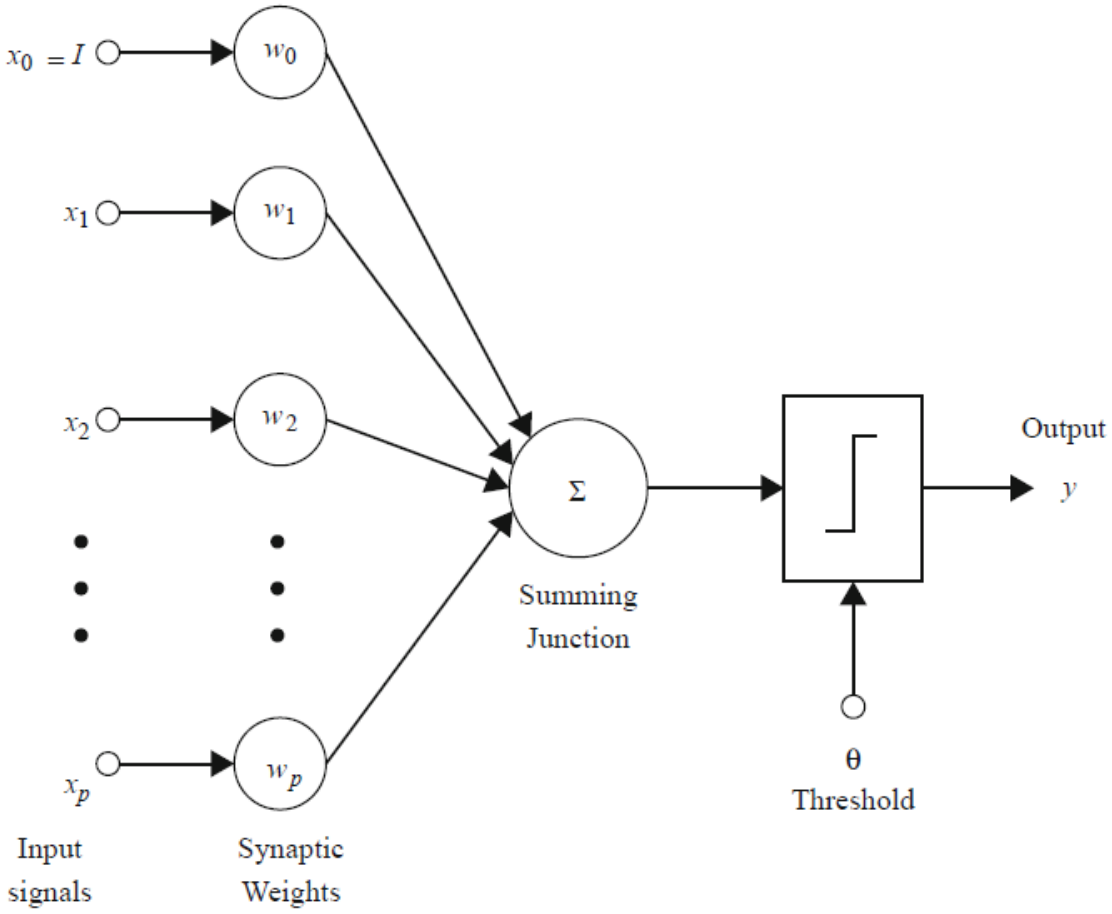
- A human neuron receives a signal, processes it, and propagates the signal (or not)

- A mathematical model of the neuron called the artificial neuron or the perceptron has been used to try and mimic our understanding of the functioning of the human neuron and brain



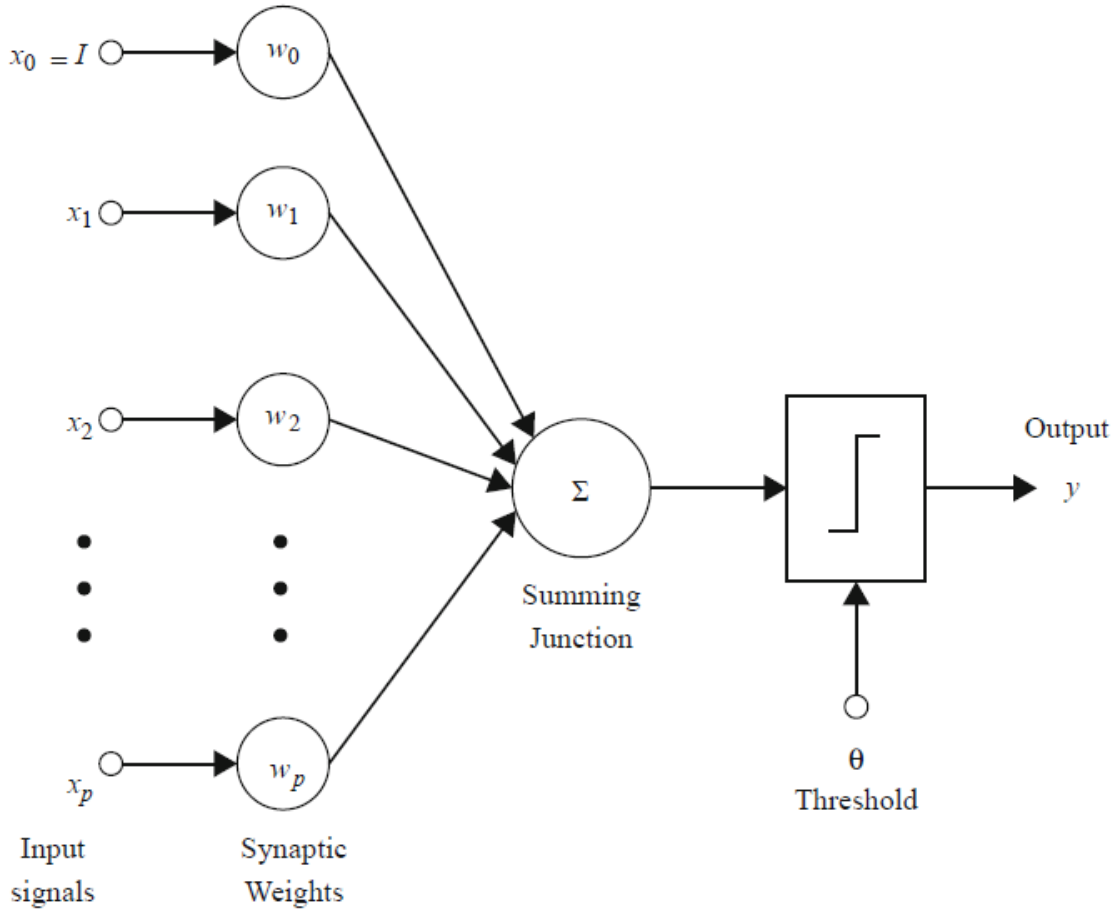
# Artificial Neuron

- Artificial neuron is adaptive i.e., parameters can be changed during operation (training) to suit the problem
- It can have $n$ inputs $x_1, x_2, \ldots, x_n$ that model the signals coming from dendrites (input features or from other neurons)
- Each link between an input and the neuron has a weight (real numbers) associated to it. For example, $x_1$ has weight $w_1$, $x_2$ has weight $w_2$, and so on.
- According to the neurophysiological motivation, some of these synaptic weights may be negative to express their inhibitory character.



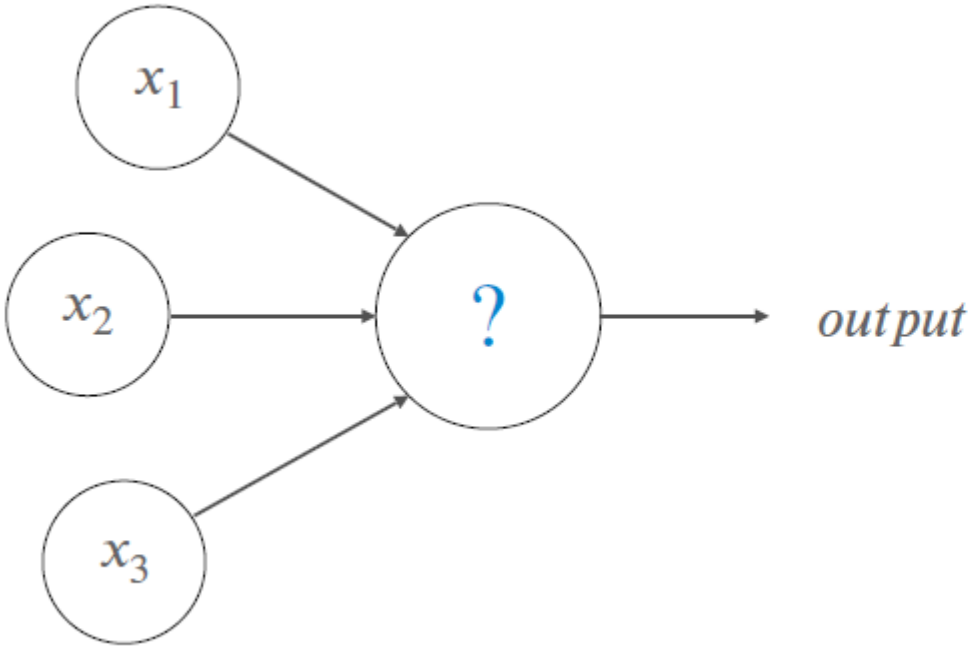
# Artificial Neuron

- It computes weighted sum,
  $= x_0 w_0 + x_1 w_1 + x_2 w_2 + \cdots + x_n w_n$
- $w_0$ is the bias, an intercept value to make the model more general that comes from an extra bias unit x0, which is always +1.
- Sometimes instead of $w_0$ we use $b$ to represent the bias, which is added to the weighted sum,
  $= x_1 w_1 + x_2 w_2 + \cdots + x_n w_n + b$
- If the sum of the weighted inputs is greater than a threshold value, $y$, then the neuron fires, producing an output of +1: otherwise it does not fire, and the output is 0 (activation function)



# Artificial Neuron - Example

- Simple artificial neuron with three inputs and one output




*Do I snowboard this weekend?*

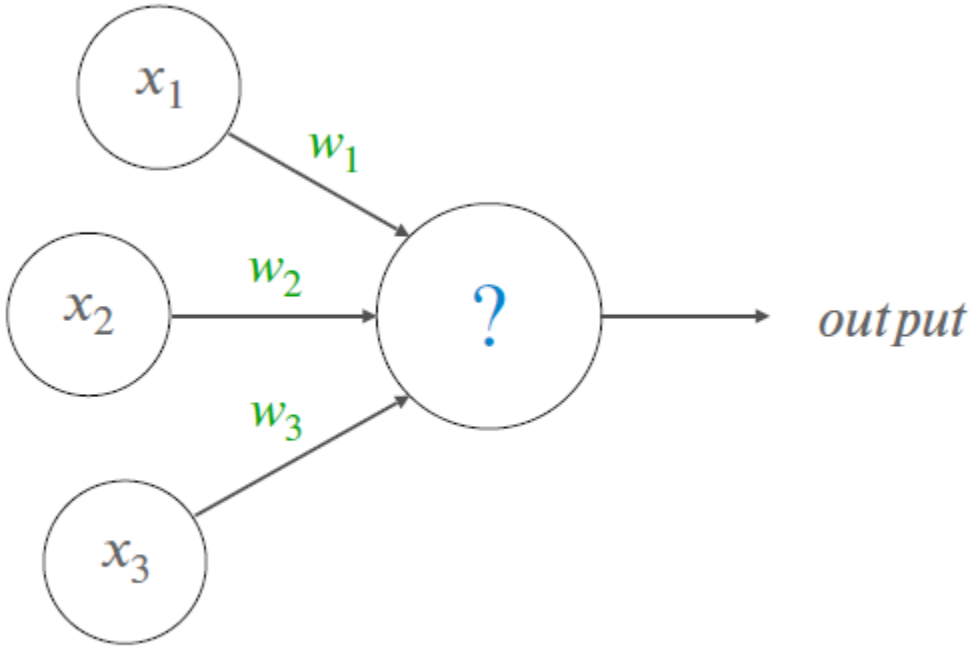
$x_1 \rightarrow$ *Is the weather good?*

$x_2 \rightarrow$ *Is the powder good?*

$x_3 \rightarrow$ *Am I in the mood to drive?*

# Artificial Neuron - Example

- Simple artificial neuron with weights

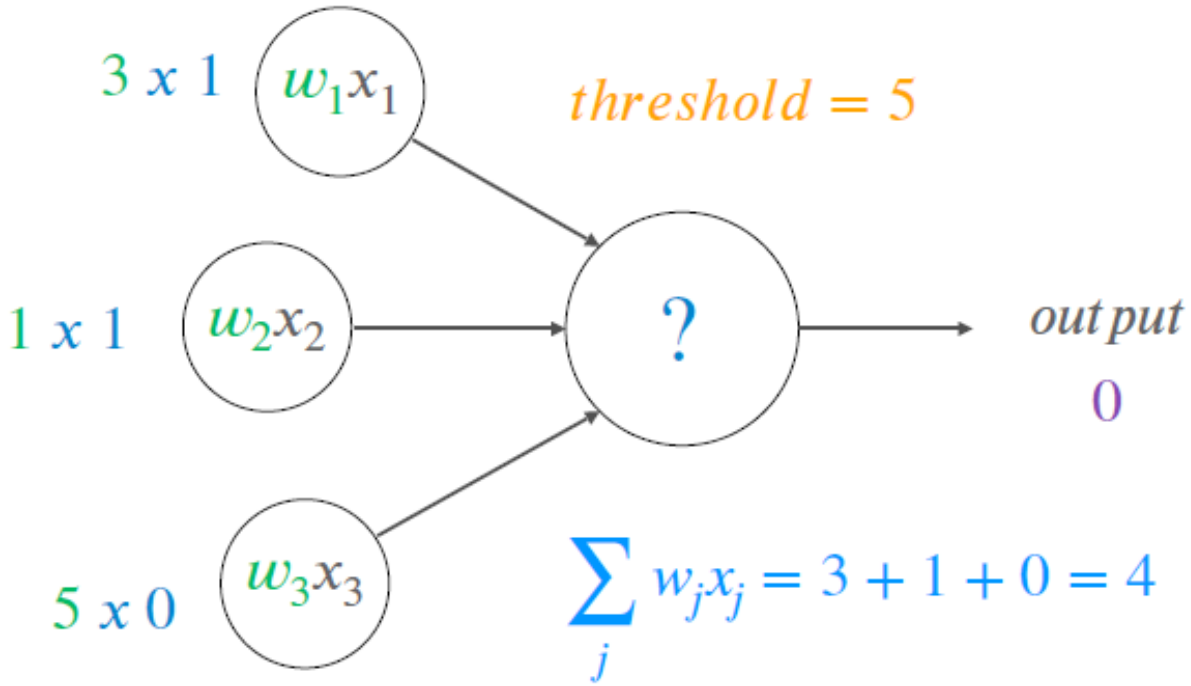



Note: we are not considering bias here

$$output = \begin{cases} 0, & \sum_{j=0}^{n} w_j x_j \leq threshold \\ 1, & \sum_{j=0}^{n} w_j x_j > threshold \end{cases}$$

# Artificial Neuron - Example

- Compute output using weights and inputs. If the weighted sum is higher than the threshold then the output is 1, otherwise 0 (as in this case)




*Do I snowboard this weekend?*

$x_1 = 1$

$w_1 = 3$

$x_2 = 1$

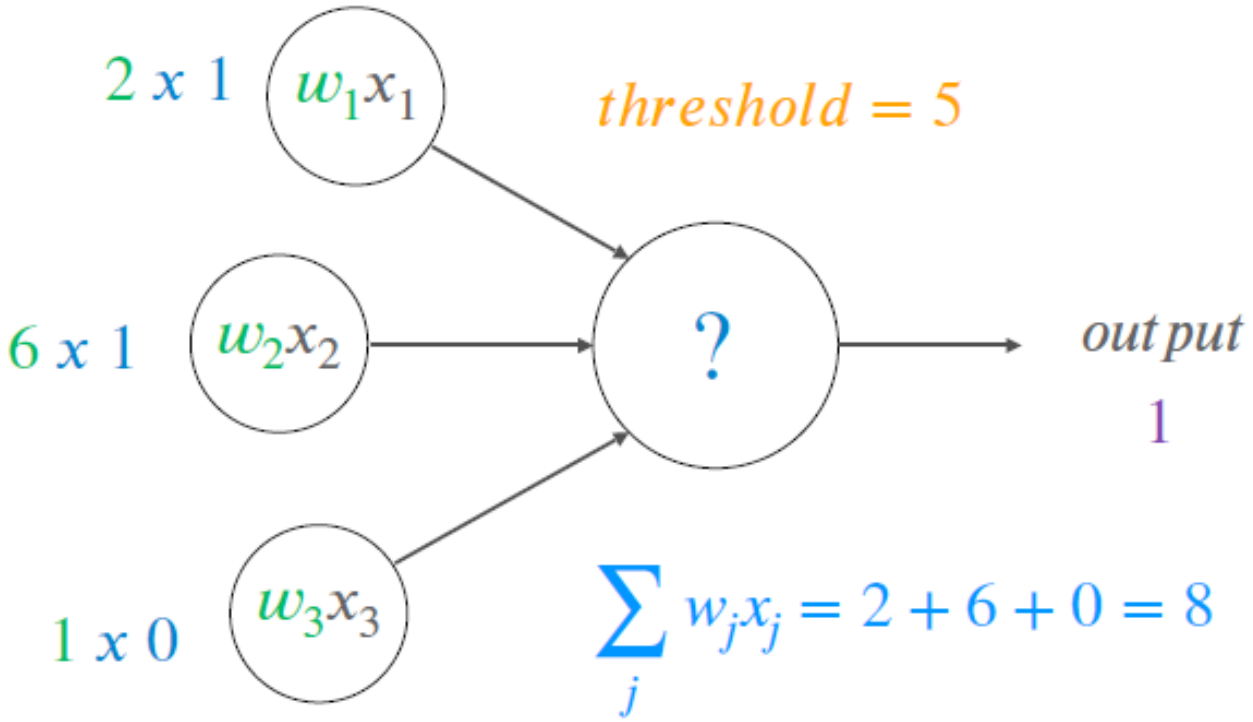
$w_2 = 1$

$x_3 = 0$

$w_3 = 5$

# Artificial Neuron - Example

- Compute output using weights and inputs. If the weighted sum is higher than the threshold then the output is 1 (as in this case), otherwise 0




*Do I snowboard this weekend?*

$x_1 = 1$

$w_1 = 2$

$x_2 = 1$

$w_2 = 6$

$x_3 = 0$

$w_3 = 1$

# Artificial Neuron - Output

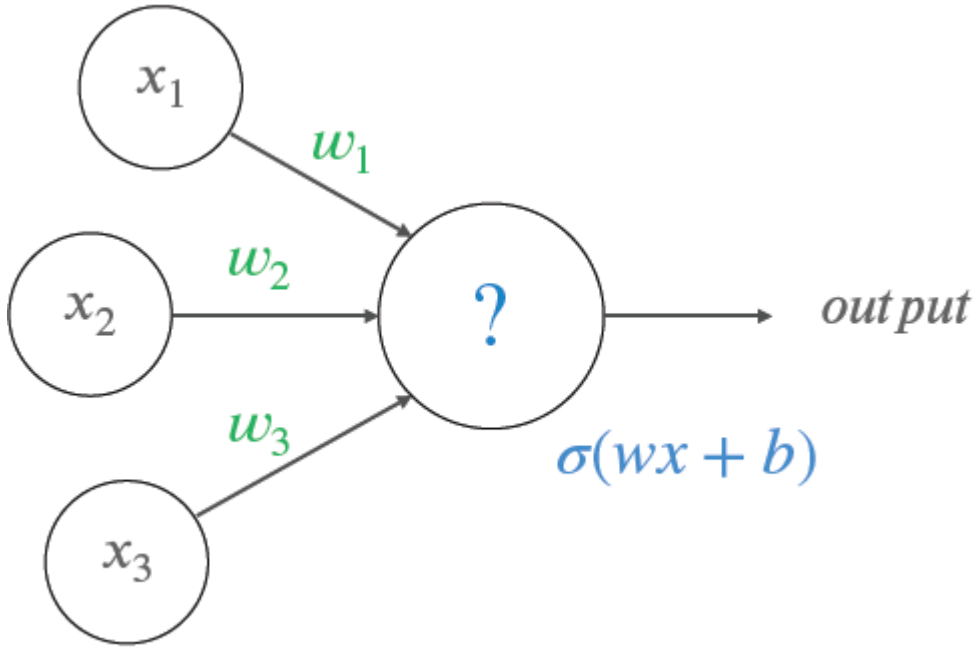
- With the bias factor, the outputs will be computed as the following,

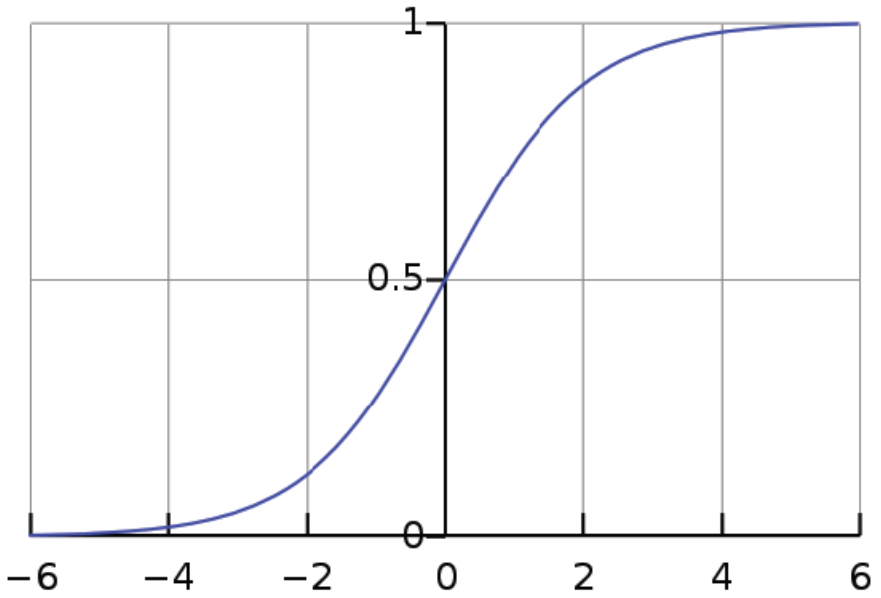
$$output = \begin{cases} 0, & wx + b \leq 0 \\ 1, & wx + b > 0 \end{cases}$$

where b tells how easy it is to get the perceptron to fire

# Artificial Neuron – Activation Function

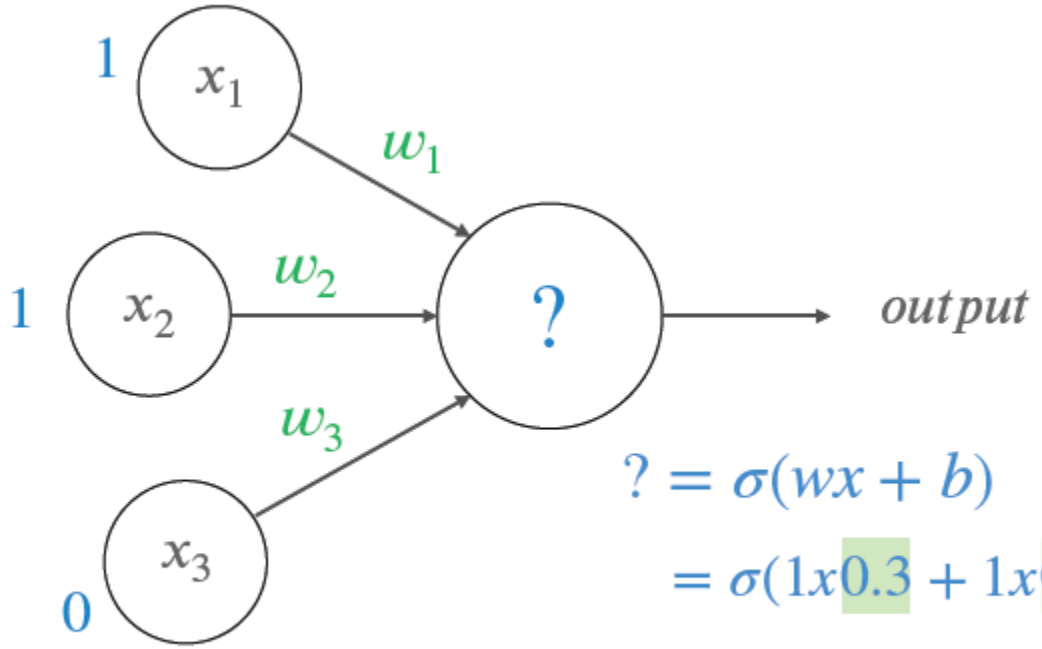
- The weighted sum is passed through an activation function which decides the output of a neuron
- Instead of [0, 1], output is now (0…1) and is defined by $\sigma(wx + b)$



# Artificial Neuron – Activation Function

- Compute output using $\sigma(wx + b)$




$$? = \sigma(wx + b)$$
$$= \sigma(1x0.3 + 1x0.6 + 0x0.1)$$
$$= \sigma(0.9)$$
$$= 0.7109$$

*Do I snowboard this weekend?*

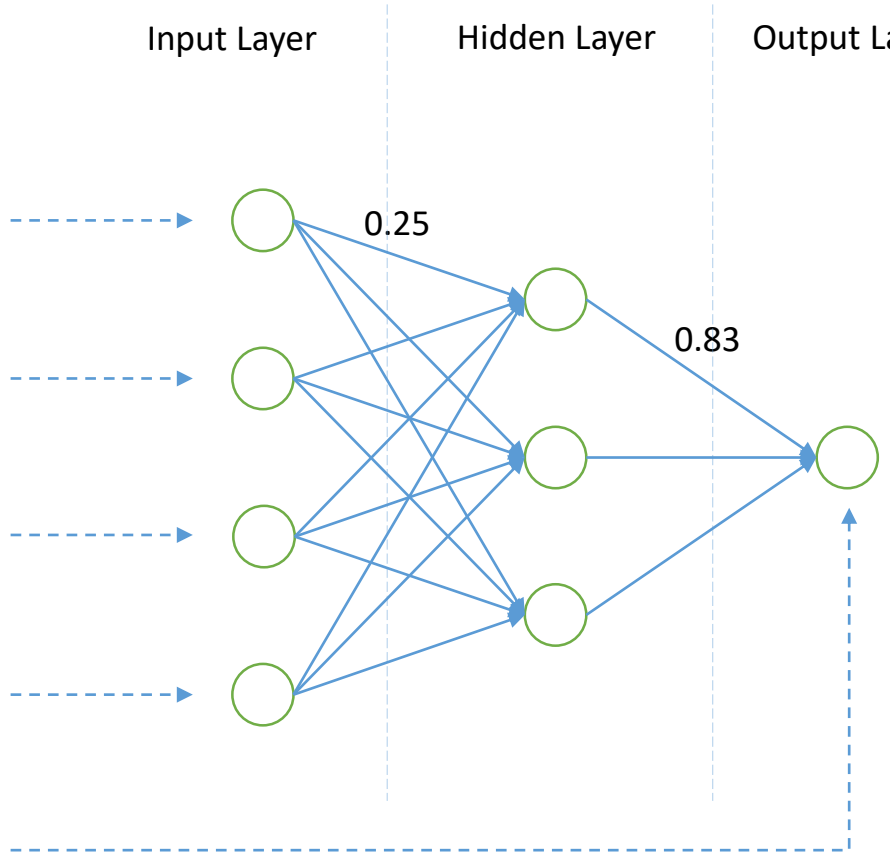
$x_1 = 1$
$w_1 = 0.3$

$x_2 = 1$
$w_2 = 0.6$

$x_3 = 0$
$w_3 = 0.1$

# Artificial Neural Network

| | $x^{(1)}$ | $x^{(2)}$ | | $x^{(m)}$ |
|---|---|---|---|---|
| **Size** | 300 | 200 | .... | 300 |
| **# Bedroom** | 4 | 3 | .... | 5 |
| **Zip code** | 2000 | 4007 | .... | 3300 |
| **Salary** | 120k | 90k | .... | 140k |
| **Price (y)** | 700k | 600k | .... | 750k |




A network consisted of many artificial neurons and the links between them

# Artificial Neuron Network

- Each layer is made up of units (artificial neurons). The inputs to the network correspond to the attributes measured for each training tuple. The inputs are fed simultaneously into the units making up the input layer. These inputs pass through the input layer and are then weighted and fed simultaneously to a second layer of “neuronlike” units, known as a hidden layer. The outputs of the hidden layer units can be input to another hidden layer, and so on.
- The number of hidden layers is arbitrary, although in practice, usually only one is used. The weighted outputs of the last hidden layer are input to units making up the output layer, which emits the network’s prediction for given tuples. The units in the input layer are called input units.

# Logistic Regression – Simplified Neural Network

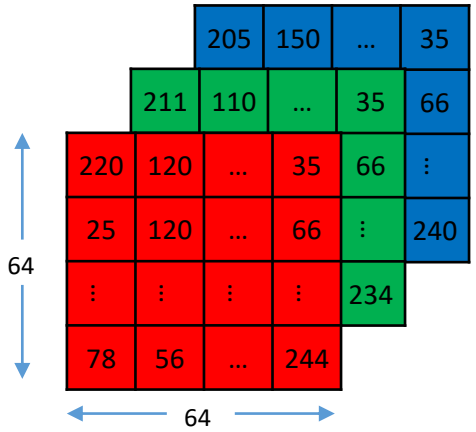
# Introduction to Logistic Regression

- It is a simple form of neural network
- It is a supervised learning algorithm for binary classification
- It will help in understanding the basics ideas of implementing a neural network

# Problem Setting – Binary classification

1 = cat
Or
0 = non cat

i.e., only two class values




$$X = \begin{bmatrix} 220 \\ 120 \\ \vdots \\ 244 \\ 211 \\ \vdots \\ 234 \\ 205 \\ \vdots \\ 240 \end{bmatrix}$$

$$dimensions\ of\ input\ features = n_x = 64 * 64 * 3 = 12288$$

If the input is not represented in linear form then we transform it as in the case of images in RGB

# Problem Setting

- Training example: $(x, y)$ where $x \in \mathbb{R}^{n_x}$ and $y \in \{0,1\}$
- $m$ training examples: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$
- Input features matrix,

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \ldots & x^{(m)} \\ | & | & & | \end{bmatrix} \quad \begin{matrix} \uparrow \\ n_x \\ \downarrow \end{matrix} \qquad X \in \mathbb{R}^{n_x \times m}$$

$$\leftarrow \quad m \quad \rightarrow$$

- Output matrix,

$$\mathrm{Y} = \left[y^{(1)}, y^{(2)}, \ldots, y^{(m)}\right] \qquad Y \in \mathbb{R}^{1 \times m}$$

# Problem Setting

The output $\hat{y}$ is computed using,

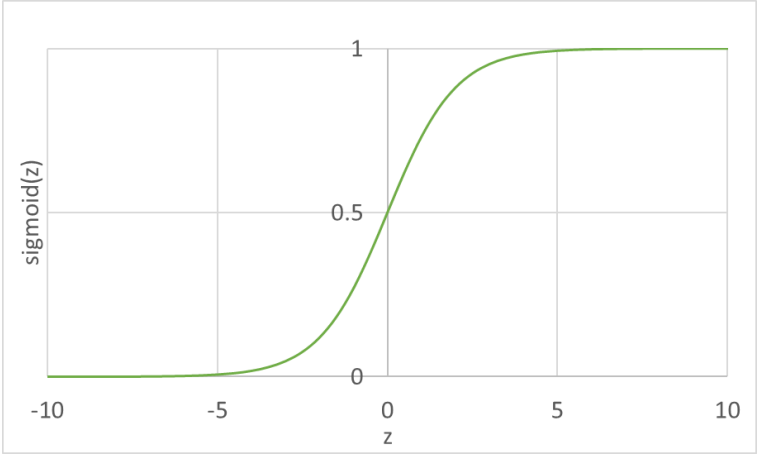
- input features $x \in \mathbb{R}^{n_x}$,
- weight vector w $\in \mathbb{R}^{n_x}$,
- and b $\in \mathbb{R}$

Output: $\hat{y} = \sigma\,(w^{\mathrm{T}} * x + b)$

Let $\mathrm{z} = w^{\mathrm{T}} * x + b$

$\hat{y} = \sigma\,(\mathrm{z})$

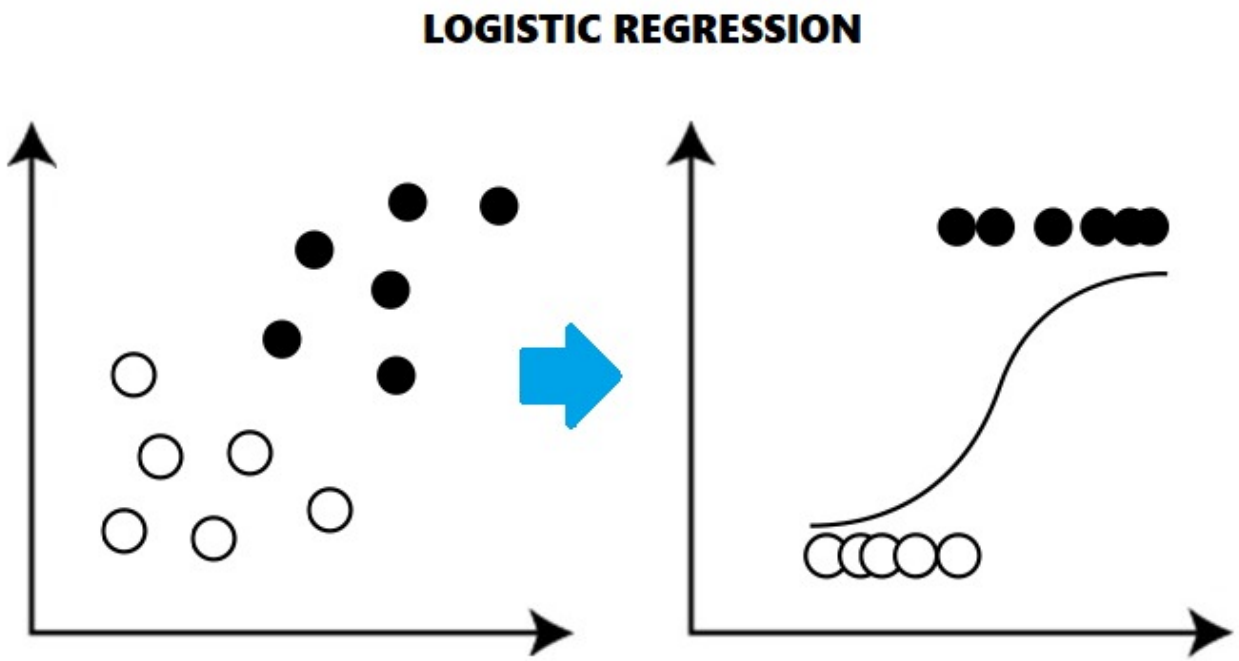
Graph of a sigmoid function




Sigmoid function: $\sigma(z) = \frac{1}{(1+e^{-z})}$

If $z$ becomes large, $\sigma(z)$ tends to become 1

If $z$ becomes small, $\sigma(z)$ tends to become 0


LOGISTIC REGRESSION

# Loss Function

- Loss function is a function that one need to define to measure how good our predicted output is when the true label is y
- As square error seems like it might be a reasonable choice except that it makes gradient descent not work well
- it would be of no use as it would end up being a non-convex function with many local minimums
- it would be very difficult to minimize the cost value and find the global minimum
- So in logistic regression, we will actually define a different loss function that plays a similar role as squared error, that will give us an optimization problem that is convex

# Loss Function

- Loss/error function (w.r.t. single training example):

$$L(y, \hat{y}) = -(y log \hat{y} + (1 - y)\log(1 - \hat{y}))$$

If $y = 1, L(y, \hat{y}) = -log\hat{y}$

- To make it small, $log\hat{y}$ needs to be large, so $\hat{y}$ needs to be large (i.e., close to 1 because of the sigmoid function)

If $y = 0, L(y, \hat{y}) = -\log(1 - \hat{y})$

- To make it small, $\log(1 - \hat{y})$ needs to be large, so $\hat{y}$ needs to be small (i.e., close to 0 because of the sigmoid function

# Cost Function

- Cost function (w.r.t. $m$ training examples):

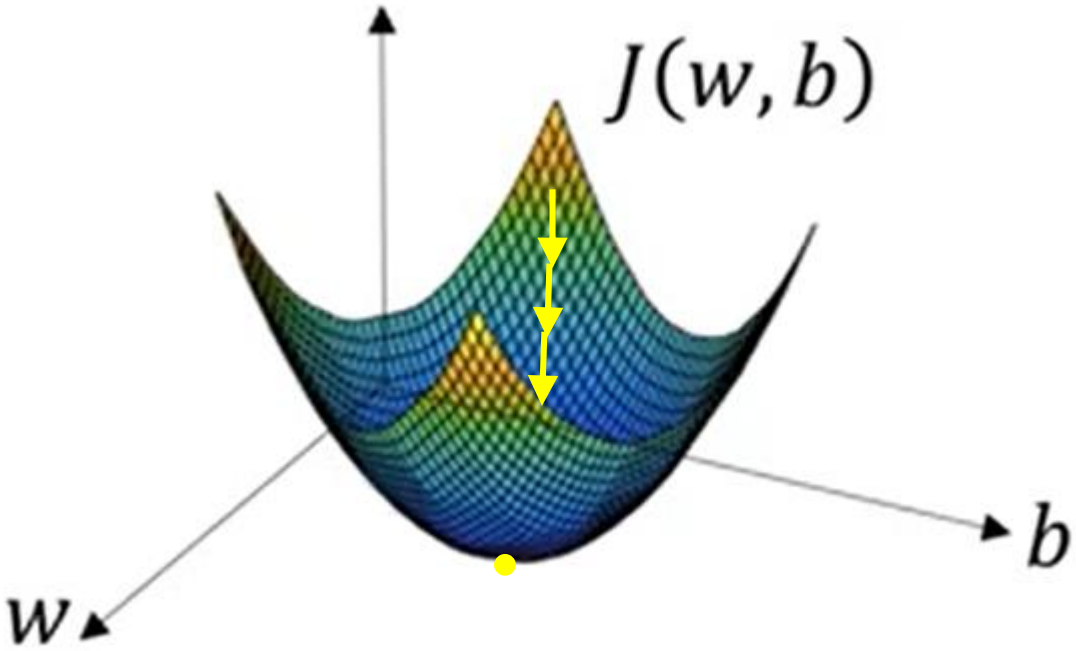
$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L\left(y^{(i)}, \hat{y}^{(i)}\right)$$

$$= -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} log\hat{y}^{(i)} + \left(1 - y^{(i)}\right)\log(1 - \hat{y}^{(i)})\right)$$

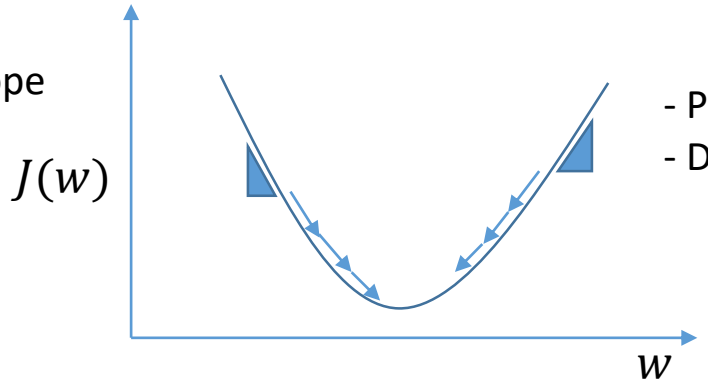
# Gradient Decent

- Gradient descent is a method that finds a minimum of a function by figuring out in which direction the function's slope is rising the most steeply, and moving in the opposite direction
- It finds the gradient of the loss function at the current point and moving in the opposite direction
- The magnitude of the amount to move in gradient descent is the value of the slope $\frac{d}{\mathrm{dw}} f(x; w)$ weighted by a learning rate $\alpha$

# Learning Parameters using Gradient Descent




- Negative slope
- Increases w




- Positive slope
- Decreases w

Repeat{

$$w := w - \alpha \frac{dJ(w)}{dw}$$

}

Here $\alpha$ is the learning rate which controls how big step to take in an iteration

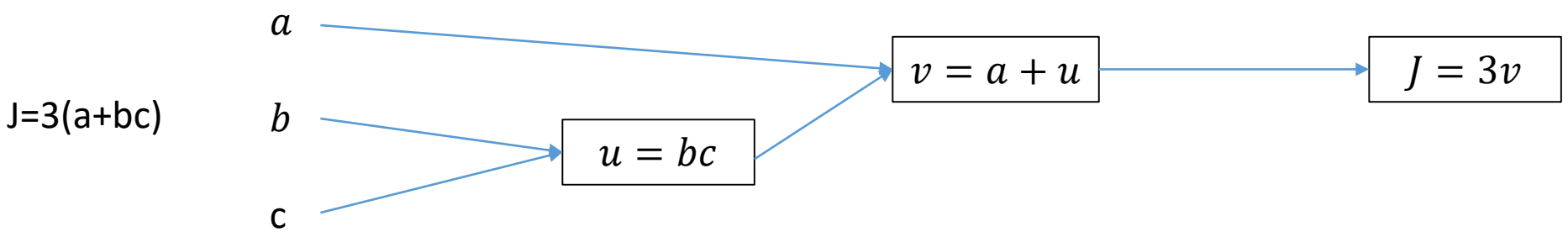
# Learning Parameters using Gradient Descent

- For cost function $J(w, b)$, the parameter updates will be,

$$w := w - \alpha \frac{\partial J(w, b)}{\partial w}$$

$$b := b - \alpha \frac{\partial J(w, b)}{\partial b}$$

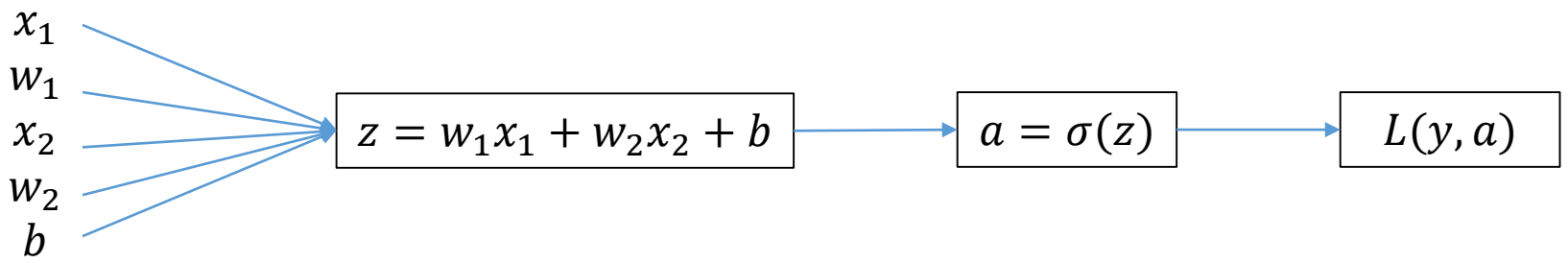
# Computation Graph




- A computation graph organizes the computations
- Following the graph from left to right, the output can be computed using the input values (Forward propagation)
- Following the graph from right to left, the value at the output can be propagated to the inputs (Back propagation)

$a = \hat{y}$, the output

# Logistic Regression Gradient Descent




- To update $w_1$, $w_2$, and $b$, we need to find out their partial derivatives w.r.t $L$
- $\frac{\partial L(y,a)}{\partial a} = da = -\frac{y}{a} + \frac{1-y}{1-a}$
- $\frac{\partial L(y,a)}{\partial z} = dz = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z} = \frac{\partial L}{\partial a}[a(1-a)] = a - y$
- $\frac{\partial L}{\partial w_1} = dw_1 = x_1 * dz,$ $\frac{\partial L}{\partial w_2} = dw_2 = x_2 * dz,$ $\frac{\partial L}{\partial b} = \mathrm{db} = dz$
- $w_1 := w_1 - \alpha * \mathrm{dw}_1,$ $w_2 := w_2 - \alpha * dw_2,$ $\mathrm{b} := b - \alpha * db$

$a = \hat{y}$, the output

# Logistic Regression on $m$ Examples

$J = 0; dw_1 = 0; dw_2 = 0; db = 0$

For $i = 1$ to $m$

$z^{(i)} = w^{\mathrm{T}} x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

$J \mathrel{+}= -\left(y^{(i)} log(a^{(i)}) + \left(1 - y^{(i)}\right)\log(1 - a^{(i)})\right)$

$dz^{(i)} = a^{(i)} - y^{(i)}$

$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$

$dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}$

$db \mathrel{+}= dz^{(i)}$

$J = \frac{J}{m};\ dw_1 = \frac{dw_1}{m}; dw_1 = \frac{dw_1}{m};\ db = \frac{db}{m}$

# Logistic Regression on $m$ Examples

- Note, we can compute gradient of a parameter, say $w_1$, over m examples like $dw_1 = \frac{dw_1}{m}$ because $\frac{\partial J(w,b)}{\partial w_1} = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial}{\partial w_1} L(a^{(i)}, y^{(i)})$

- Finally, the parameters are updated as,

$$w_1 := w_1 - \alpha * \mathrm{dw}_1$$

$$w_2 := w_2 - \alpha * dw_2$$

$$\mathrm{b} := b - \alpha * db$$

- It completes one step of gradient decent. It will be repeated until the cost is minimum

# Summary of the Algorithm

- Calculate the prediction $\hat{y}$ using the current parameters ($W$ and $b$)
- Calculate the loss of the current values
- Calculate the gradients of the loss function with respect to the parameters
- Adjust the weights (optimize) using the gradients
- Repeat for the number of epochs, i.e. the number of times to go through the provided examples (dataset)

# Loss Function Derivation

- We learn weights that maximize the probability of the correct label $p(y|x)$. Since there are only two discrete outcomes (1 or 0), this is a Bernoulli distribution, and we can express the probability $p(y|x)$that our classifier produces for one observation as,

$$p(y|x) = \hat{y}^{y}(1-\hat{y})^{1-y}$$

- If y=1, it simplifies to $\hat{y}$; if y=0, it simplifies to 1-$\hat{y}$
- Take the log of both side,

$$\log p(y|x) = \log \hat{y}^{y}(1-\hat{y})^{1-y}$$
$$= ylog\hat{y} + (1-y)\log(1-\hat{y})$$

- To turn this into loss function (something that we need to minimize), just flip the sign,

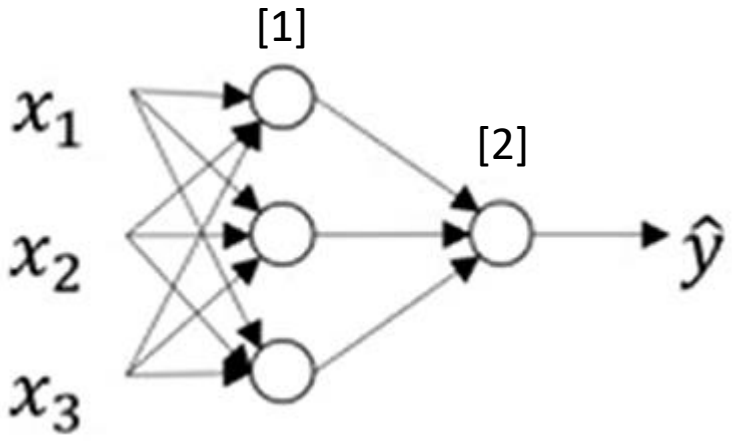
$$L_{CE}(\hat{y}, y) = -\log p(y|x) = -[ylog\hat{y} + (1-y)\log(1-\hat{y})]$$

# Multilayer Feed-forward Neural Network

# Multilayer Feed-forward Neural Network

- An extension of logistic regression
- Comprised of multiple layers and multiple neurons
- Squared error is normally used
- The computation graph will be $z$ calculation followed by $a$ calculation and then these two steps are repeated for each layer

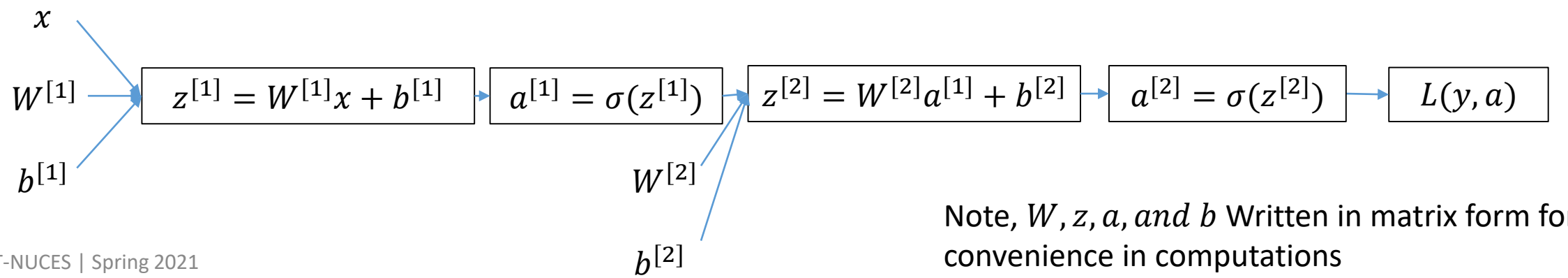
# Multilayer Feed-forward Neural Network – Example




$$W^{[1]} = \begin{bmatrix} w_1^{[1](1)} & w_2^{[1](1)} & w_3^{[1](1)} \\ w_1^{[1](2)} & w_2^{[1](2)} & w_2^{[1](2)} \\ w_1^{[1](3)} & w_2^{[1](3)} & w_3^{[1](3)} \end{bmatrix} \quad b^{[1]} = \begin{bmatrix} b^{[1](1)} \\ b^{[1](2)} \\ b^{[1](3)} \end{bmatrix} \quad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad z^{[1]} = \begin{bmatrix} z^{[1](1)} \\ z^{[1](2)} \\ z^{[1](3)} \end{bmatrix}$$

$$W^{[2]} = \begin{bmatrix} w_1^{[2](1)} & w_2^{[2](1)} & w_3^{[1](1)} \end{bmatrix} \quad b^{[2]} = \begin{bmatrix} b^{[2](1)} \end{bmatrix} \quad a^{[1]} = \begin{bmatrix} a^{[1](1)} \\ a^{[1](2)} \\ a^{[1](3)} \end{bmatrix} \quad z^{[2]} = \begin{bmatrix} z^{[2](1)} \end{bmatrix}$$

[]-> layer number
()->node in a layer
Subscript->node number of previous layer




Note, $W, z, a, and\ b$ Written in matrix form for convenience in computations

# Backpropogation algorithm

- The input data are repeatedly presented to the network.  With each presentation, the output of the network is compared to the desired output and an error is computed.
- This error is then fed back (backpropagated) to the neural network and used to adjust the weights such that the error decreases with each iteration and the neural model gets closer and closer to producing the desired output. This process is known as training.

# Backpropogation algorithm

- Backpropagation learns by iteratively processing a data set of training tuples, comparing the network's prediction for each tuple with the actual known *target value. The target value may be the known class label of the training* tuple (for classification problems) or a continuous value (for numeric prediction).

# Backpropogation algorithm

- For each training tuple, the weights are modified so as to minimize the mean-squared error between the network's prediction and the actual target value. These modifications are made in the "backwards" direction (i.e., from the output layer) through each hidden layer down to the first hidden layer (hence the name *backpropagation). Although it is* not guaranteed, in general the weights will eventually converge, and the learning process stops.

# Training phase

- During training, the weights are adjusted until the outputs of the perceptron become consistent with the true outputs of the training examples, $y_i$.
- The weights are initially assigned to small random values and the training examples are used one after another to tweak the weights in the network, all the examples are used, and then the whole process is iterated until all examples are correctly classified by the output. This constitutes learning.

# Multilayer backpropogation Algorithm

**Algorithm: Backpropagation.** Neural network learning for classification or numeric prediction, using the backpropagation algorithm.

**Input:**

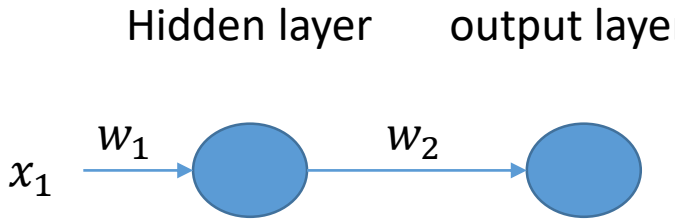
- *D*, a data set consisting of the training tuples and their associated target values;
- *l*, the learning rate;
- *network*, a multilayer feed-forward network.

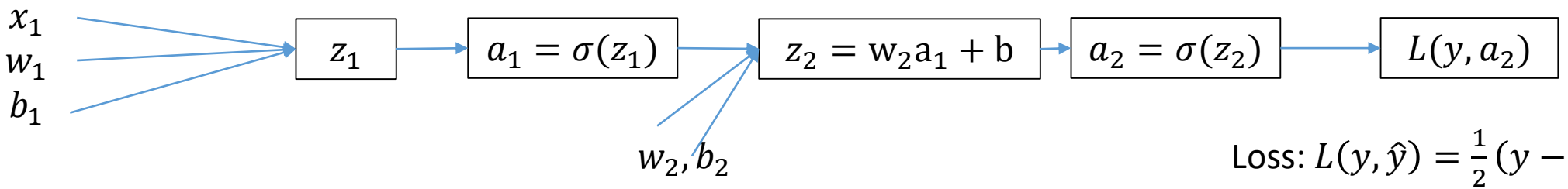
**Output:** A trained neural network.

Line 3 to 20 is one epoch

(1) Initialize all weights and biases in *network*;
(2) **while** terminating condition is not satisfied {
(3) **for** each training tuple $\boldsymbol{X}$ in $D$ {
(4) // Propagate the inputs forward:
(5) **for** each input layer unit $j$ {
(6) $O_j = I_j$; // output of an input unit is its actual input value
(7) **for** each hidden or output layer unit $j$ {
(8) $I_j = \sum_i w_{ij} O_i + \theta_j$; //compute the net input of unit $j$ with respect to the previous layer, $i$
(9) $O_j = \frac{1}{1+e^{-I_j}}$; } // compute the output of each unit $j$
(10) // Backpropagate the errors:
(11) **for** each unit $j$ in the output layer
(12) $Err_j = O_j(1 - O_j)(T_j - O_j)$; // compute the error
(13) **for** each unit $j$ in the hidden layers, from the last to the first hidden layer
(14) $Err_j = O_j(1 - O_j) \sum_k Err_k w_{jk}$; // compute the error with respect to the next higher layer, $k$
(15) **for** each weight $w_{ij}$ in *network* {
(16) $\Delta w_{ij} = (l) Err_j O_i$; // weight increment
(17) $w_{ij} = w_{ij} + \Delta w_{ij}$; } // weight update
(18) **for** each bias $\theta_j$ in *network* {
(19) $\Delta \theta_j = (l) Err_j$; // bias increment
(20) $\theta_j = \theta_j + \Delta \theta_j$; } // bias update
(21) } }

# Neural Network Gradient Descent




Loss: $L(y, \hat{y}) = \frac{1}{2}(y - a)^2$

- To update $w_1$, $w_2$, $b_1$ and $b_2$, find out their partial derivatives w.r.t $L$
- $\frac{\partial L}{\partial a_2} = da_2 = a_2 - y$
- $\frac{\partial L}{\partial z_2} = dz_2 = \frac{\partial L}{\partial a_2}\frac{\partial a_2}{\partial z_2} = \frac{\partial L}{\partial a_2}[a_2(1 - a_2)] = a_2(1 - a_2)(a_2 - y)$
- $\frac{\partial L}{\partial w_2} = dw_2 = a_1 * dz_2, \quad \frac{\partial L}{\partial b_2} = \mathrm{db}_2 = dz_2$

$Err_j = O_j(1 - O_j)(T_j - O_j);$

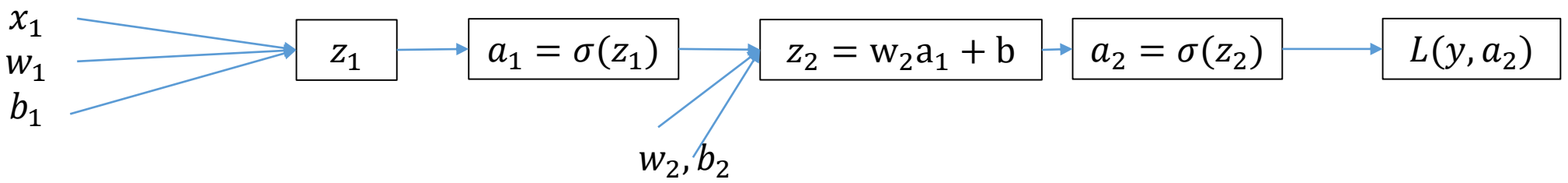
Note, it is y-a instead of a-y because we use $w: w - \alpha\Delta w$ and in the algorithm $w: w + \alpha\Delta w$ is used

$\Delta w_{ij} = (l) Err_j O_i;$

$a = \hat{y}$, the output

Note, derivative of $a = \sigma(z) = a(1 - a)$

# Neural Network Gradient Descent

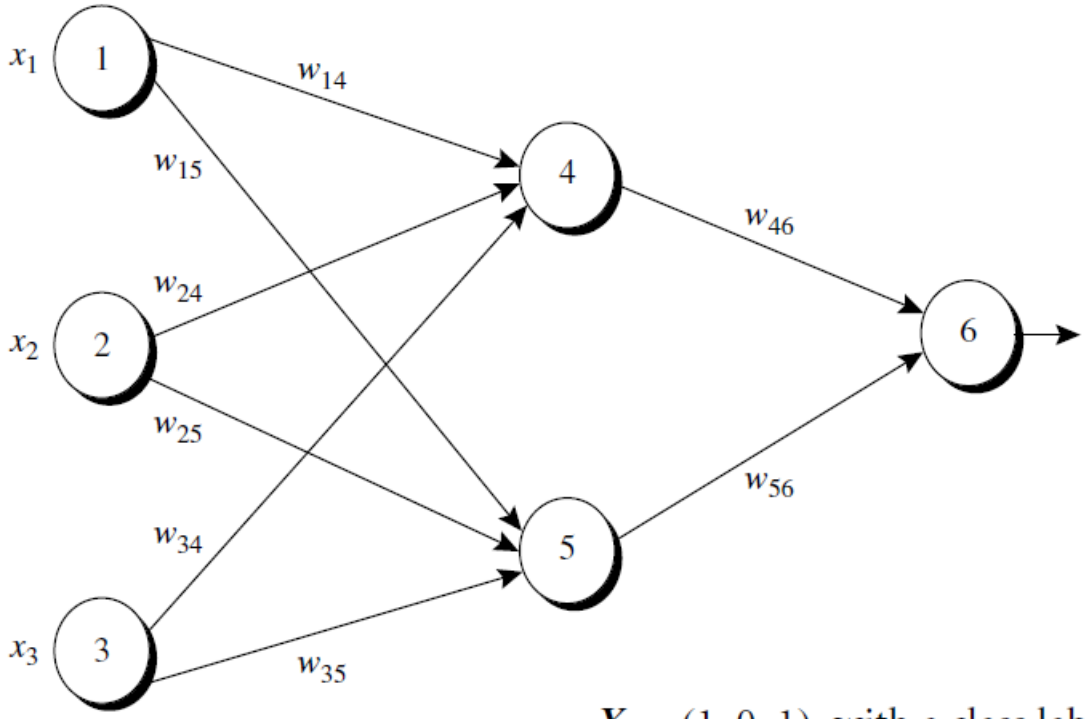



- $\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial a_1}\frac{\partial a_1}{\partial z_1}\frac{\partial z_1}{\partial w_1}$ ← Apply chain rule
- Where, $\frac{\partial L}{\partial z_2} = \frac{\partial L}{\partial a_2}\frac{\partial a_2}{\partial z_2} = (a_2 - y)[a_2(1 - a_2)], \frac{\partial z_2}{\partial a_1} = w_2$ *and* $\frac{\partial z_1}{\partial w_1} = x_1$
- $\frac{\partial L}{\partial w_1} = a_2(1 - a_2)(a_2 - y)\, w_2\, a_1(1 - a_1)x_1$
- $= a_1(1 - a_1)\ \ a_2(1 - a_2)(a_2 - y)\ \ w_2\ \ x_1$

$O_j(1 - O_j)\sum_k Err_k w_{jk}$ $\qquad$ $\Delta w_{ij} = (l)\, Err_j O_i;$

$\frac{\partial L}{\partial b_2}$
$= a_1(1 - a_1)\ \ a_2(1 - a_2)(a_2$

# Worked example




$\boldsymbol{X} = (1, 0, 1)$, with a class label of 1.

Initial Input, Weight, and Bias Values

| $x_1$ | $x_2$ | $x_3$ | $w_{14}$ | $w_{15}$ | $w_{24}$ | $w_{25}$ | $w_{34}$ | $w_{35}$ | $w_{46}$ | $w_{56}$ | $\theta_4$ | $\theta_5$ | $\theta_6$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0.2 | −0.3 | 0.4 | 0.1 | −0.5 | 0.2 | −0.3 | −0.2 | −0.4 | 0.2 | 0.1 |

Net Input and Output Calculations

| *Unit*, $j$ | *Net Input*, $I_j$ | *Output*, $O_j$ |
|---|---|---|
| 4 | $0.2 + 0 - 0.5 - 0.4 = -0.7$ | $1/(1 + e^{0.7}) = 0.332$ |
| 5 | $-0.3 + 0 + 0.2 + 0.2 = 0.1$ | $1/(1 + e^{-0.1}) = 0.525$ |
| 6 | $(-0.3)(0.332) - (0.2)(0.525) + 0.1 = -0.105$ | $1/(1 + e^{0.105}) = 0.474$ |

Calculation of the Error at Each Node

| *Unit*, $j$ | $Err_j$ |
|---|---|
| 6 | $(0.474)(1 - 0.474)(1 - 0.474) = 0.1311$ |
| 5 | $(0.525)(1 - 0.525)(0.1311)(-0.2) = -0.0065$ |
| 4 | $(0.332)(1 - 0.332)(0.1311)(-0.3) = -0.0087$ |

Calculations for Weight and Bias Updating

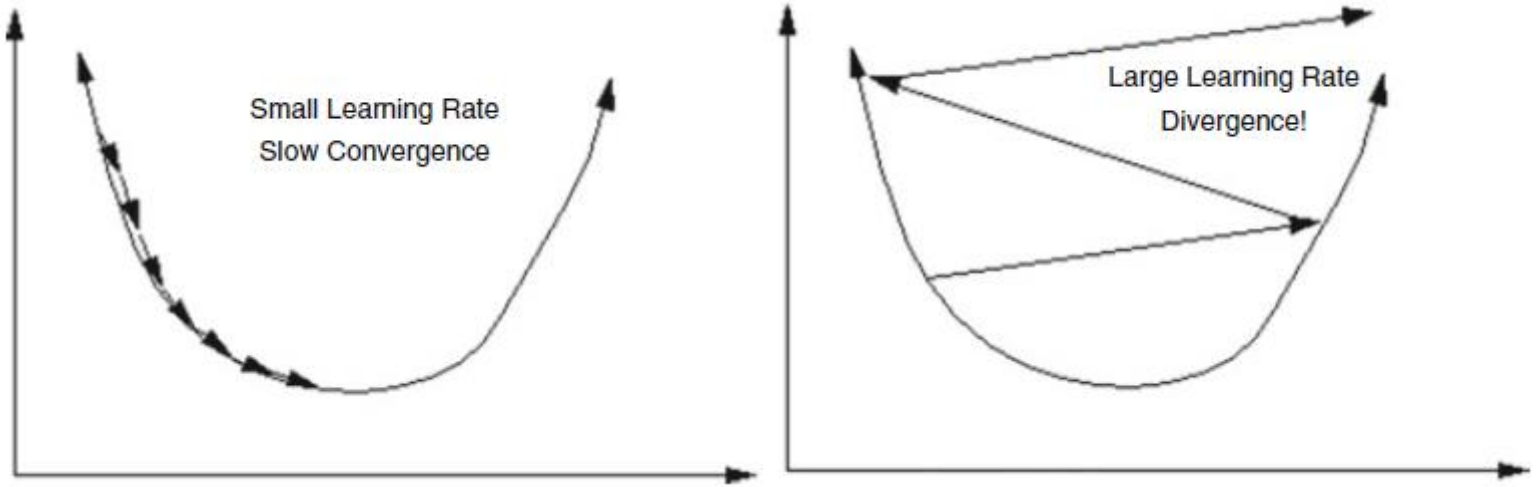
| *Weight or Bias* | *New Value* |
|---|---|
| $w_{46}$ | $-0.3 + (0.9)(0.1311)(0.332) = -0.261$ |
| $w_{56}$ | $-0.2 + (0.9)(0.1311)(0.525) = -0.138$ |
| $w_{14}$ | $0.2 + (0.9)(-0.0087)(1) = 0.192$ |
| $w_{15}$ | $-0.3 + (0.9)(-0.0065)(1) = -0.306$ |
| $w_{24}$ | $0.4 + (0.9)(-0.0087)(0) = 0.4$ |
| $w_{25}$ | $0.1 + (0.9)(-0.0065)(0) = 0.1$ |
| $w_{34}$ | $-0.5 + (0.9)(-0.0087)(1) = -0.508$ |
| $w_{35}$ | $0.2 + (0.9)(-0.0065)(1) = 0.194$ |
| $\theta_6$ | $0.1 + (0.9)(0.1311) = 0.218$ |
| $\theta_5$ | $0.2 + (0.9)(-0.0065) = 0.194$ |
| $\theta_4$ | $-0.4 + (0.9)(-0.0087) = -0.408$ |

# Learning rate

- The weight learning problem can be seen as finding the global minimum error, calculated as the proportion of misclassified training examples, over a space where all the input values can vary.
- The learning rate (between 0 and 1, but typically set to $0.1 < \alpha < 0.4$) is used to control the amount of adjustments made in each iteration.
- If $\alpha \sim 0$, then the new weight is mostly influenced by the old weight. If $\alpha \sim 1$, then the new weight is sensitive to the adjustment in the current iteration.

# Learning rate

- Sometimes, $\alpha$ is set to decay as the number of such iterations through the whole set of training examples increases, so that it can move more slowly toward the global minimum in order not to overshoot in one direction.



# Weights initialization

- **Initialize the weights:** The weights in the network are initialized to small random numbers (e.g., ranging from -1.0 to 1.0, or -0.5 to 0.5). Each unit has a *bias associated with*. The biases are similarly initialized to small random numbers.

# Convergence condition

- **Terminating condition:** Training stops when
- All $\Delta wij$ *in the previous epoch are so small as to be below some specified* threshold, or
- The percentage of tuples misclassified in the previous epoch is below some threshold, or
- A prespecified number of epochs has expired.

# Classification of unknown datapoint

- To classify an unknown tuple, X, the tuple is input to the trained network, and the net input and output of each unit are computed. (There is no need for computation and/or backpropagation of the error.)
- If there is one output node per class, then the output node with the highest value determines the predicted class label for X. If there is only one output node, then output values greater than or equal to 0.5 may be considered as belonging to the positive class, while values less than 0.5 may be considered negative.

# Critique of ANN

- Neural networks involve long training times and are therefore more suitable for applications where this is feasible.
- They require a number of parameters that are typically best determined empirically such as the network topology or “structure.” Neural networks have been criticized for their poor interpretability. For example, it is difficult for humans to interpret the symbolic meaning behind the learned weights and of “hidden units” in the network. These features initially made neural networks less desirable for data mining.

# Advantage

- Advantages of neural networks, however, include their high tolerance of noisy data as well as their ability to classify patterns on which they have not been trained. They can be used when you may have little knowledge of the relationships between attributes and classes.
- They are well suited for continuous-valued inputs *and outputs, unlike most* decision tree algorithms. They have been successful on a wide array of real-world data, including handwritten character recognition, pathology and laboratory medicine, and training a computer to pronounce English text.

- Neural network algorithms are inherently parallel; parallelization techniques can be used to speed up the computation process.
- It can perform tasks which a linear classifier cannot.
- Multilayer feed-forward networks, given enough hidden units and enough training samples, can closely approximate any function.