Neural Network

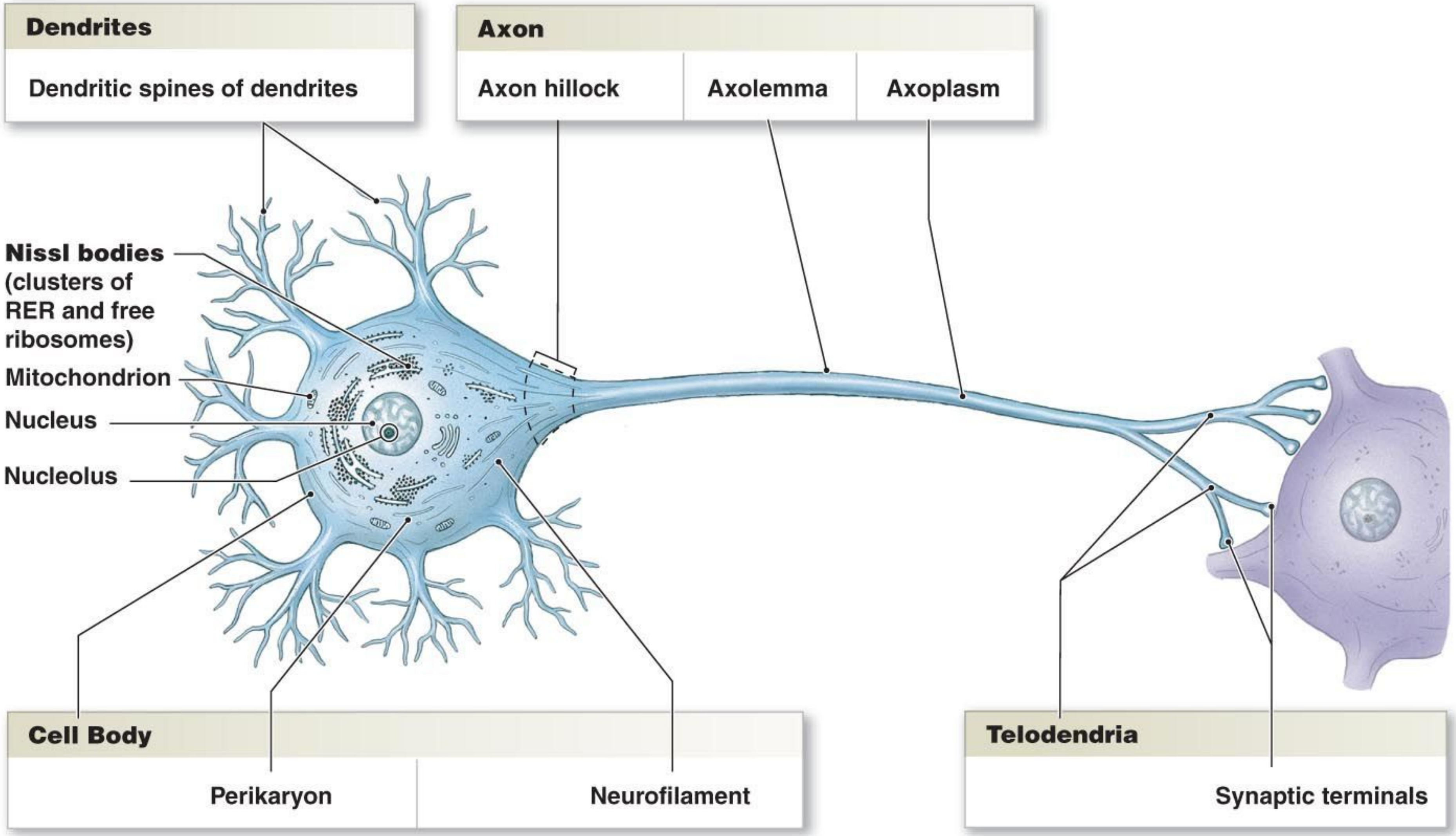
Dr. Dongchul Kim

Neural Network

- A neural network is a method in AI that teaches computers to process data in a way that is inspired by the **human brain**.

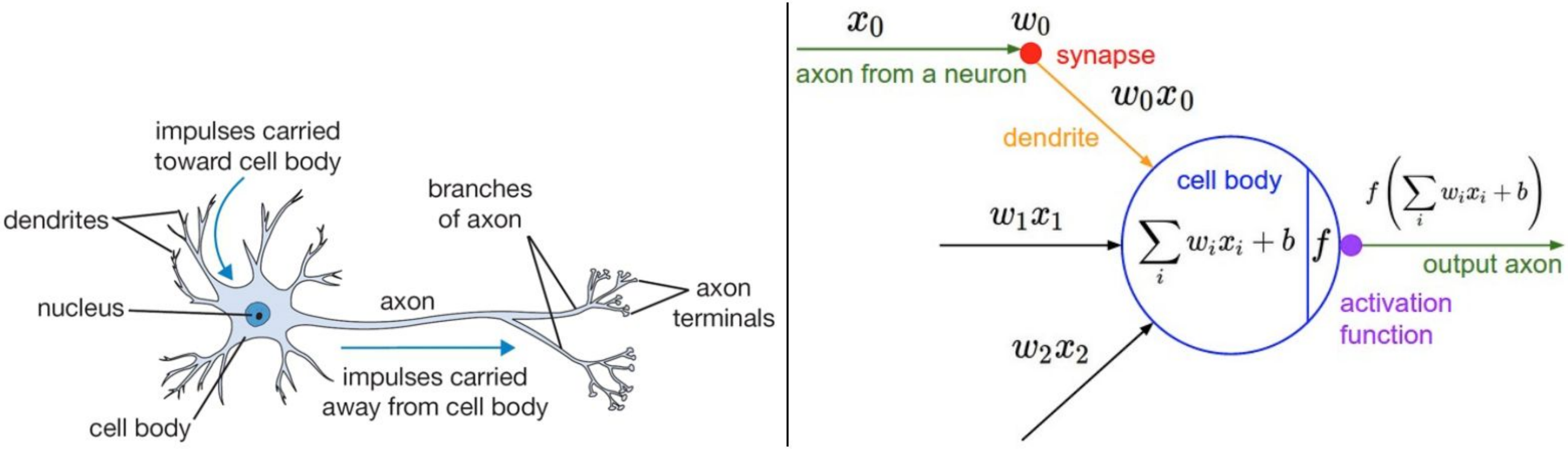
- It is a type of machine learning process, called deep learning, that uses **interconnected nodes (neurons) in a layered structure** that resembles the human brain.

A diagrammatic view of a representative neuron



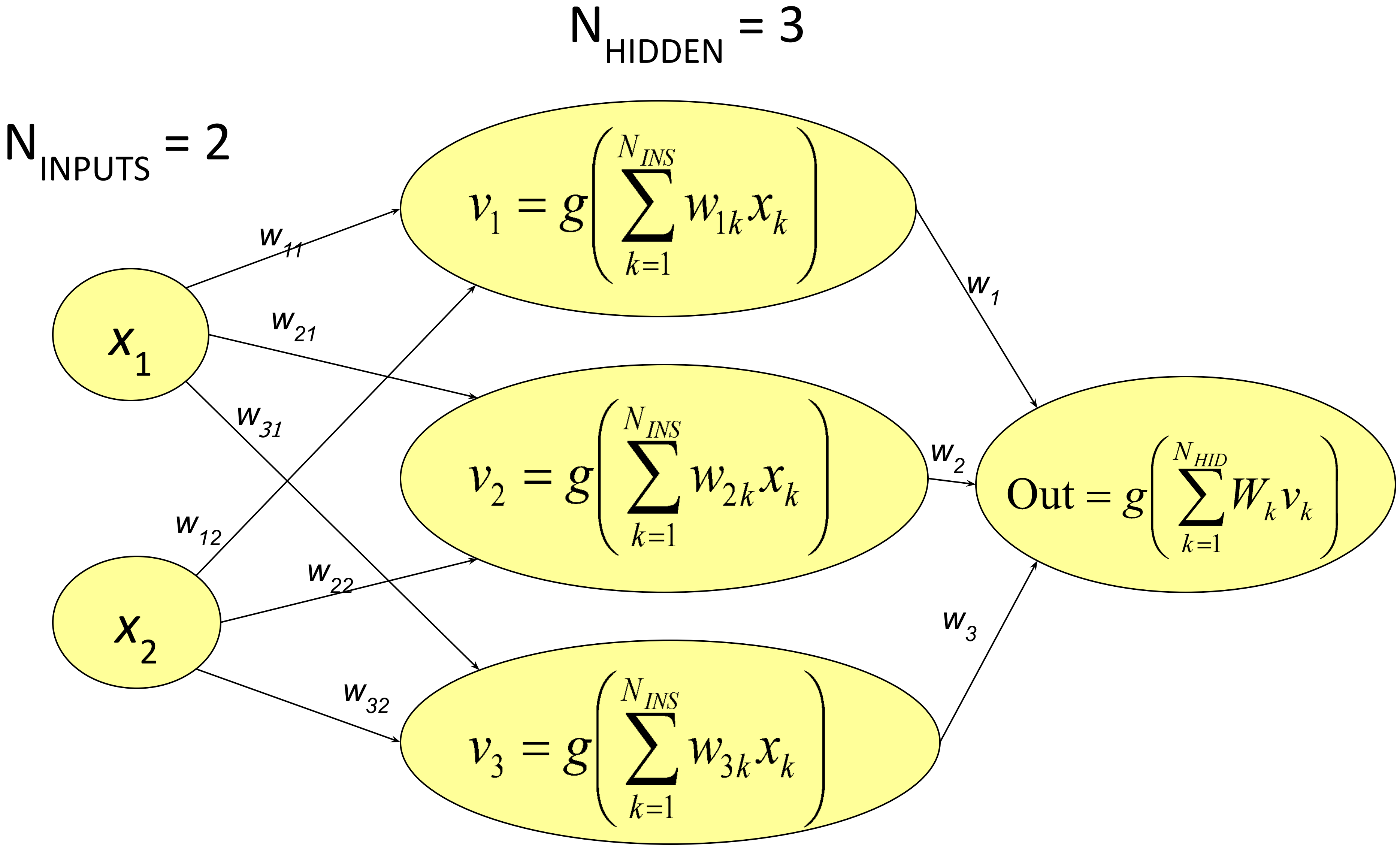
Activation Function

- the activation function of a node defines the output of that node given an input or set of inputs.

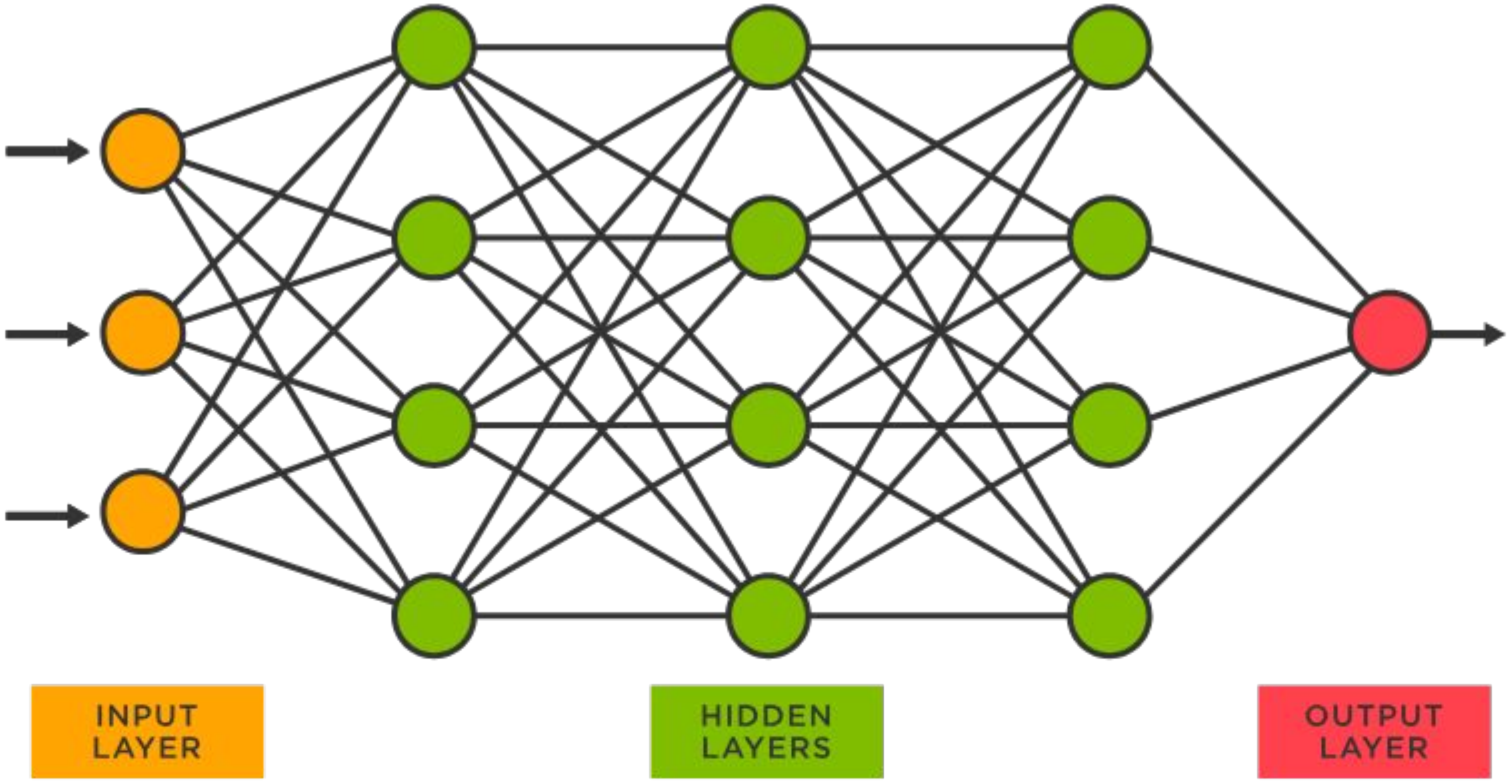


A cartoon drawing of a biological neuron (left) and its mathematical model (right).

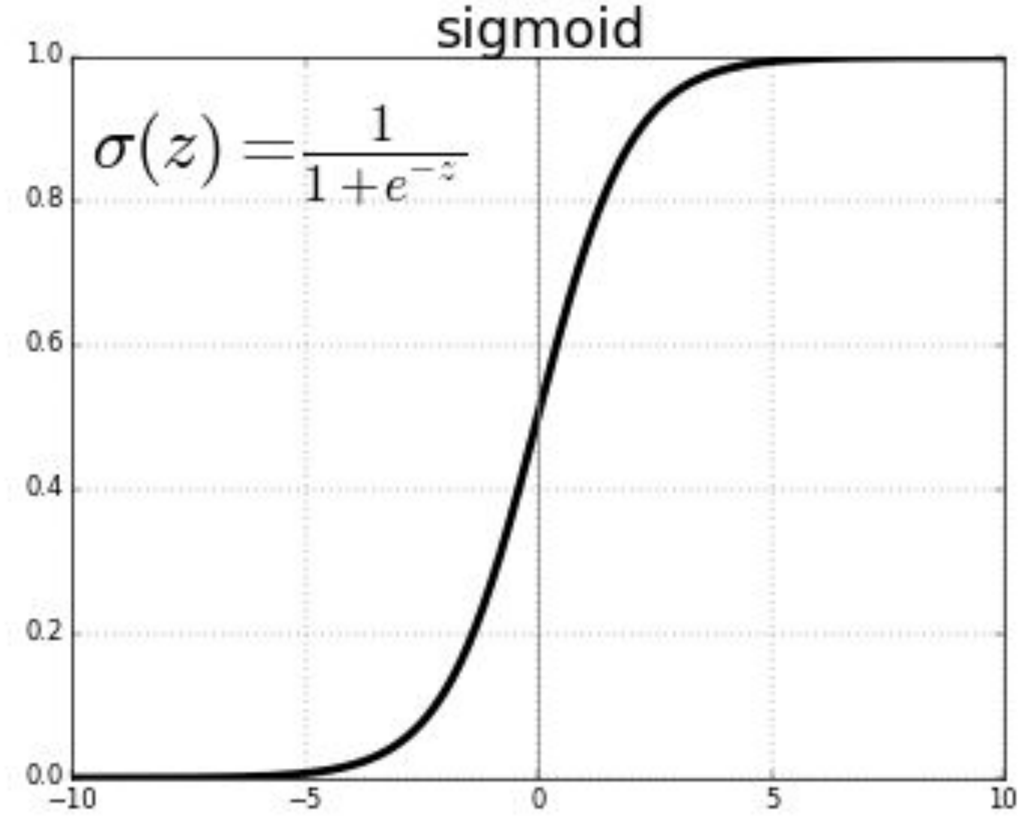
A 1-HIDDEN LAYER NET

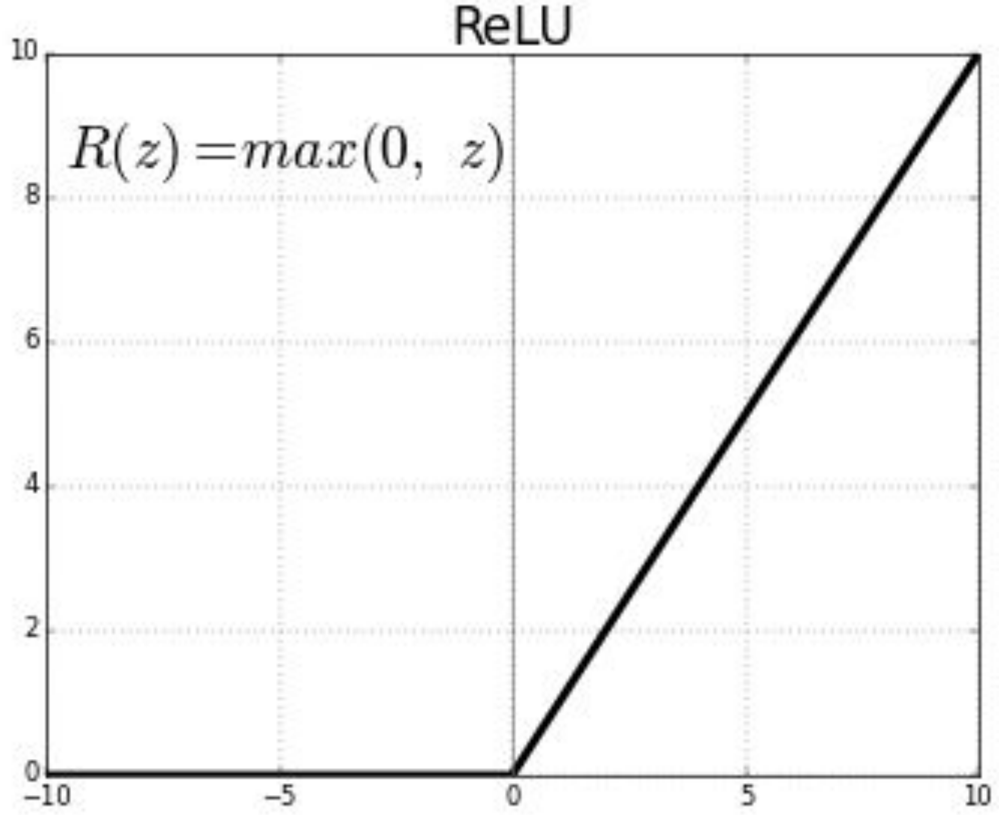

$N_{\text{HIDDEN}} = 3$
$N_{\text{INPUTS}} = 2$
x_1
x_2
w_{11}
w_{21}
w_{31}
w_{12}
w_{22}
w_{32}
$v_1 = g\left(\sum_{k=1}^{N_{INS}} w_{1k} x_k\right)$
$v_2 = g\left(\sum_{k=1}^{N_{INS}} w_{2k} x_k\right)$
$v_3 = g\left(\sum_{k=1}^{N_{INS}} w_{3k} x_k\right)$
W_1
W_2
W_3
$\text{Out} = g\left(\sum_{k=1}^{N_{HID}} W_k v_k\right)$

Neural Networks (Multi-layer Perceptron)



Activation Function





Sigmoid function

The sigmoid function transforms the output of a linear function into a nonlinear form **ranging between 0 and 1**.

It is primarily used to probabilistically represent classification problems like logistic regression.

Despite its popularity in the past, it's not frequently used in deep learning models due to the '**vanishing gradient problem**' that arises as the depth of the model increases.

We'll talk about the details of the vanishing gradient issue at a later slide.

Relu function

The Rectified Linear Unit (ReLU) function, which is actively employed in recent times, **outputs 0 when the input 'x' is negative, and 'x' when the input is positive**.

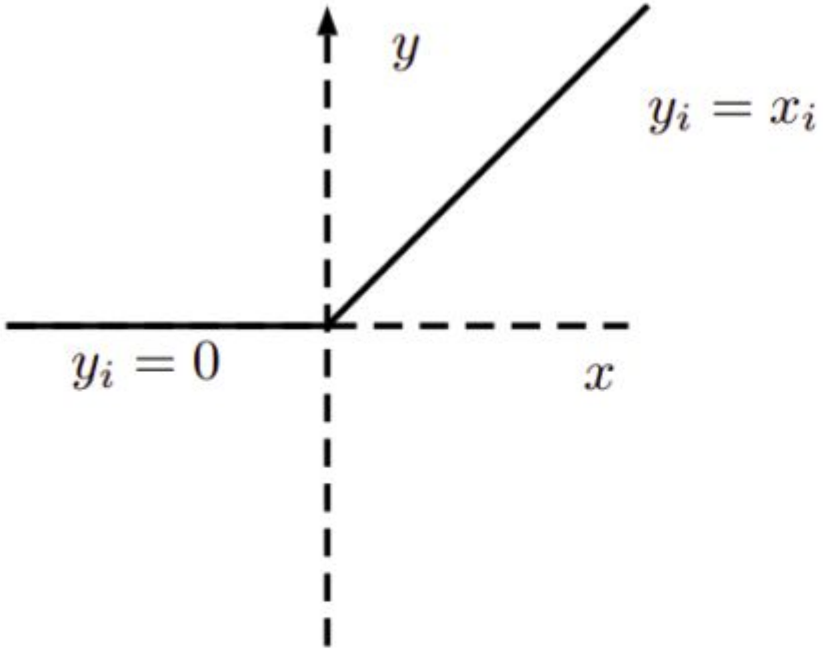
Its attributes of not impacting gradient descent and subsequently leading to faster learning, along with mitigating the vanishing gradient problem, are notable benefits.

Typically, the ReLU function finds its use in hidden layers.

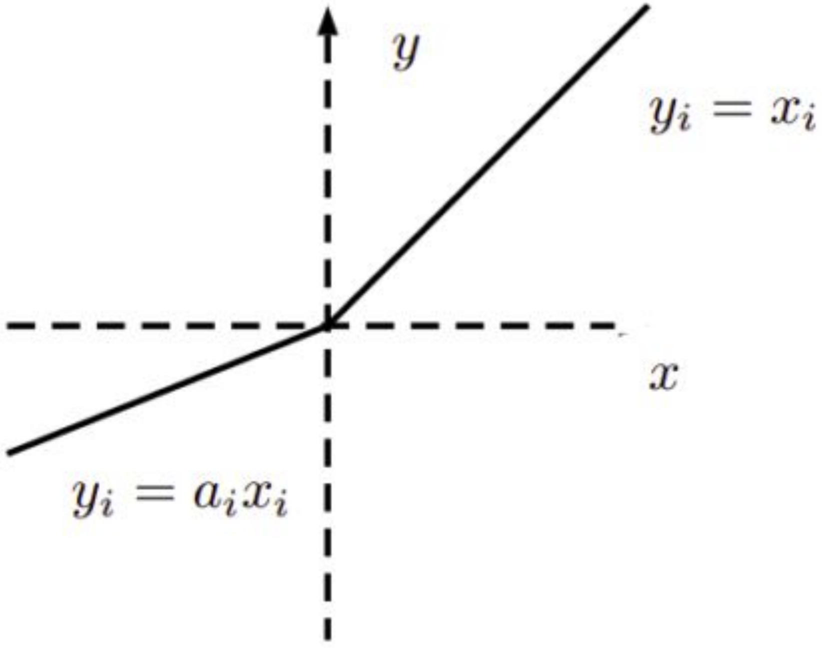
One challenge it presents is that its output is always 0 when receiving negative input values, which could potentially hamper its learning capacity.

To address this issue, the Leaky ReLU function is employed.

Leaky ReLU



ReLU



Leaky ReLU

Leaky ReLU

The Leaky ReLU function is a variation of the ReLU activation function.

For positive input values, it behaves just like the standard ReLU.

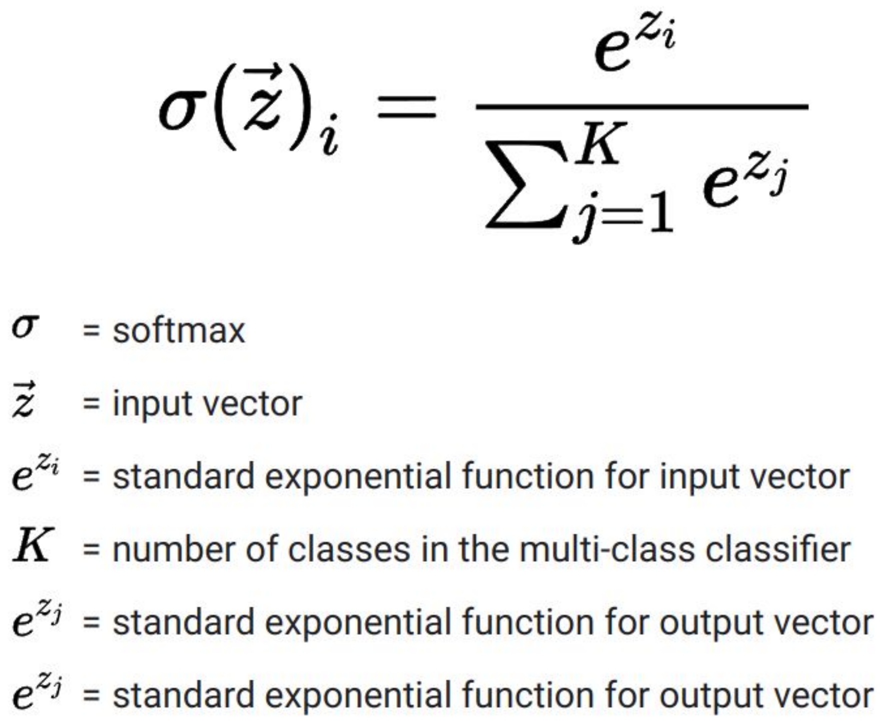
However, for negative inputs, instead of producing a zero output, it returns a small, non-zero output, thereby "leaking" a bit of information and keeping the neurons from 'dying'.

Softmax function

The softmax function normalizes input values so they are outputted within the range of 0 to 1, ensuring that the sum of these outputs always equals 1.

This function is **commonly used as the activation function for output nodes in deep learning.**

Its mathematical formula is as follows.



$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

σ = softmax

$\vec{z}$ = input vector

e^{z_i} = standard exponential function for input vector

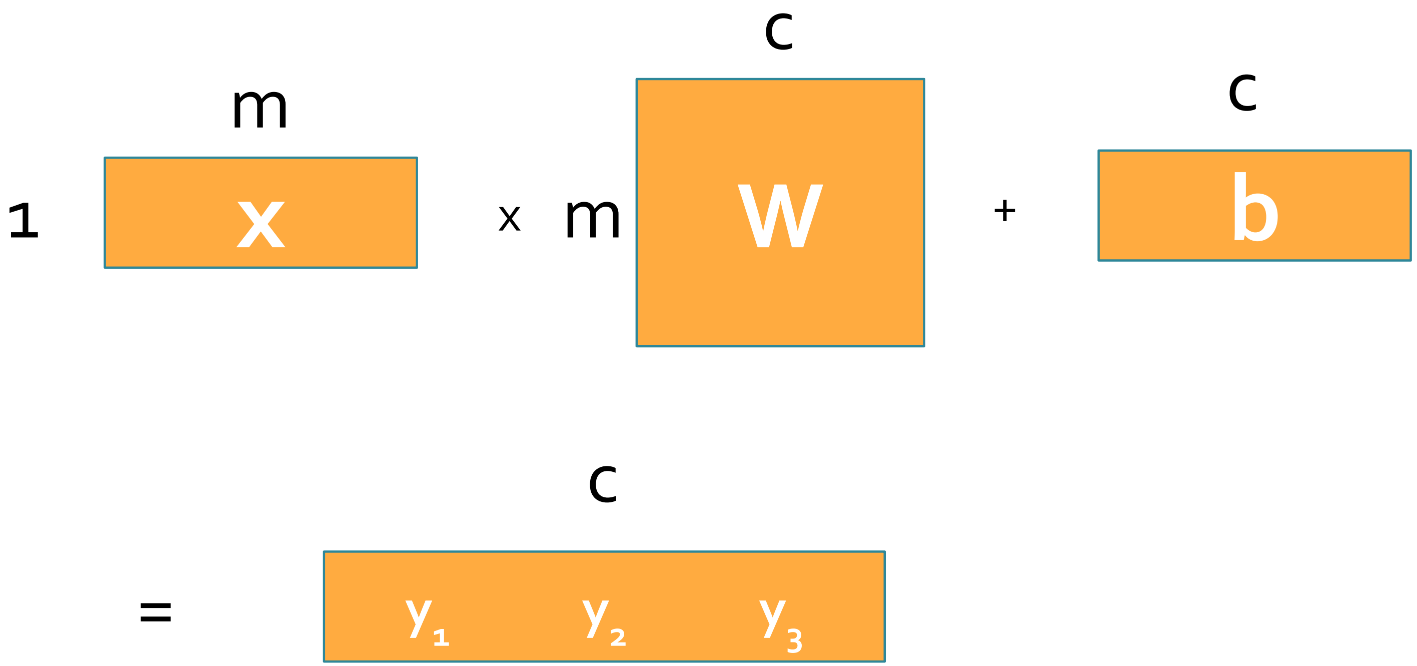
K = number of classes in the multi-class classifier

e^{z_j} = standard exponential function for output vector

e^{z_j} = standard exponential function for output vector

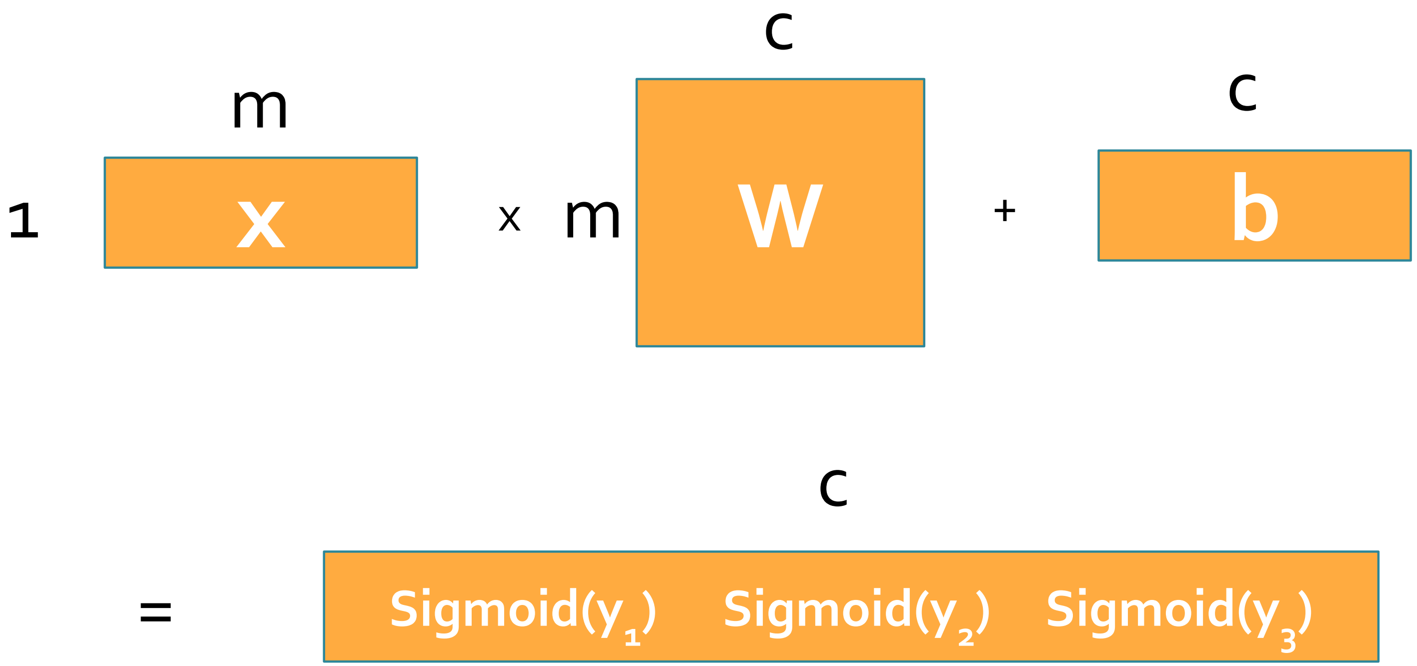
Linear Classifier

- Mutliclass
  - *c* is # of class
  - *m* is # of feature



Logistic Regression

- Mutliclass
    - *c* is # of class
    - *m* is # of feature



m

1 x

x m W

c

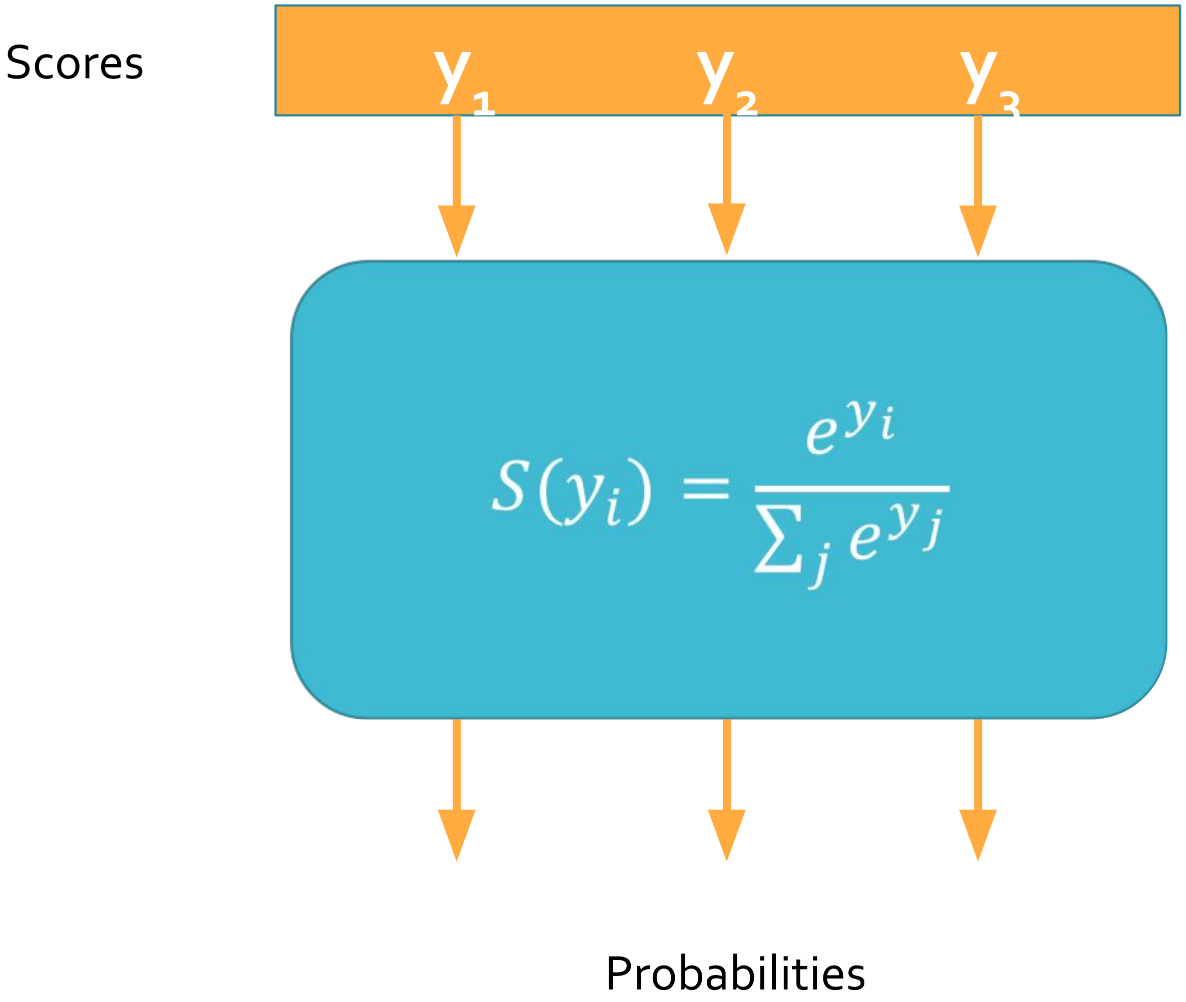
+ b

c

= Sigmoid(y_1) Sigmoid(y_2) Sigmoid(y_3)

c

Softmax



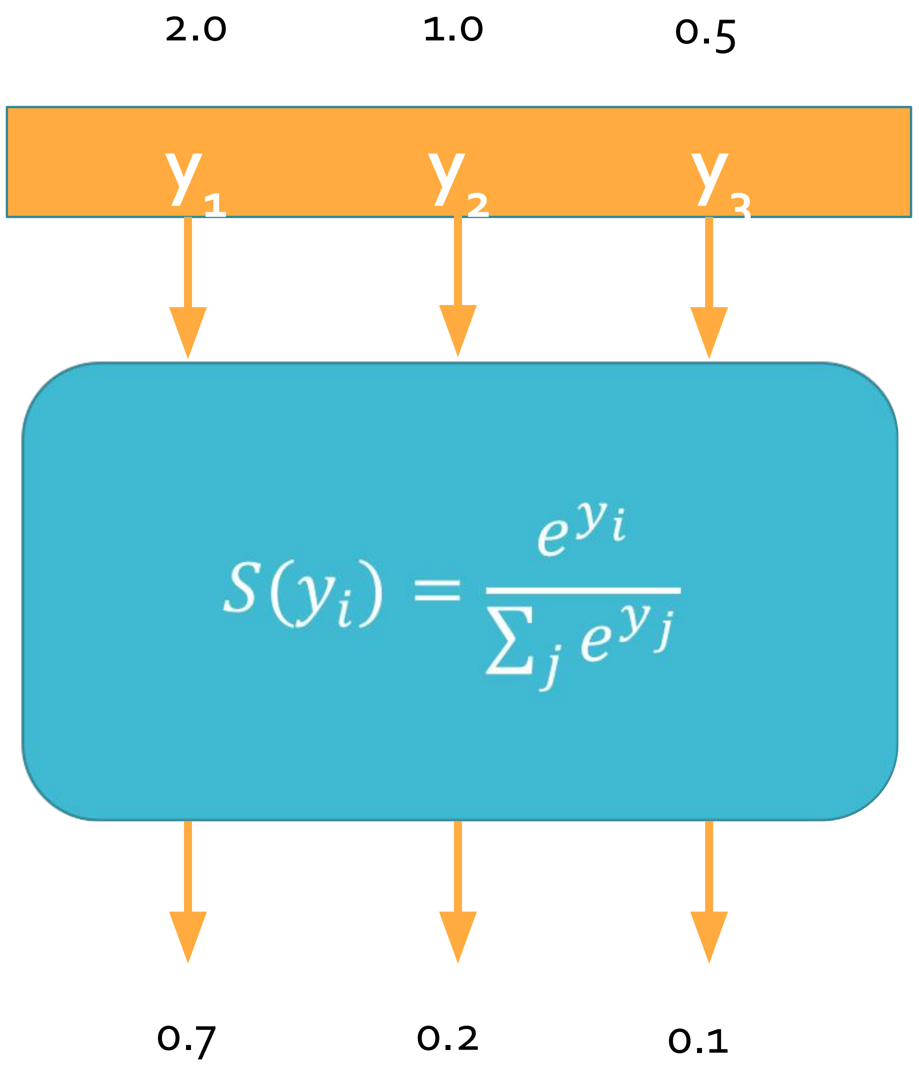
Scores

y_1 y_2 y_3

$$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

Probabilities

Softmax



$$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

Cost function

Cost(Loss) function

A loss function quantifies the **disparity between the actual data and the predictions** rendered by a model, with the weights of the model fine-tuned through training. A larger value of this loss function indicates a less accurate prediction, while a value close to zero implies a minimal difference between the predicted and actual data.

As previously discussed, the weight updates in the model are guided by the method of *gradient descent*, which leverages the instantaneous gradient of the loss function. The **resultant adjusted weights enable the model to yield more precise predictions**. Consequently, the magnitude of the loss function diminishes with the progression of the learning process.

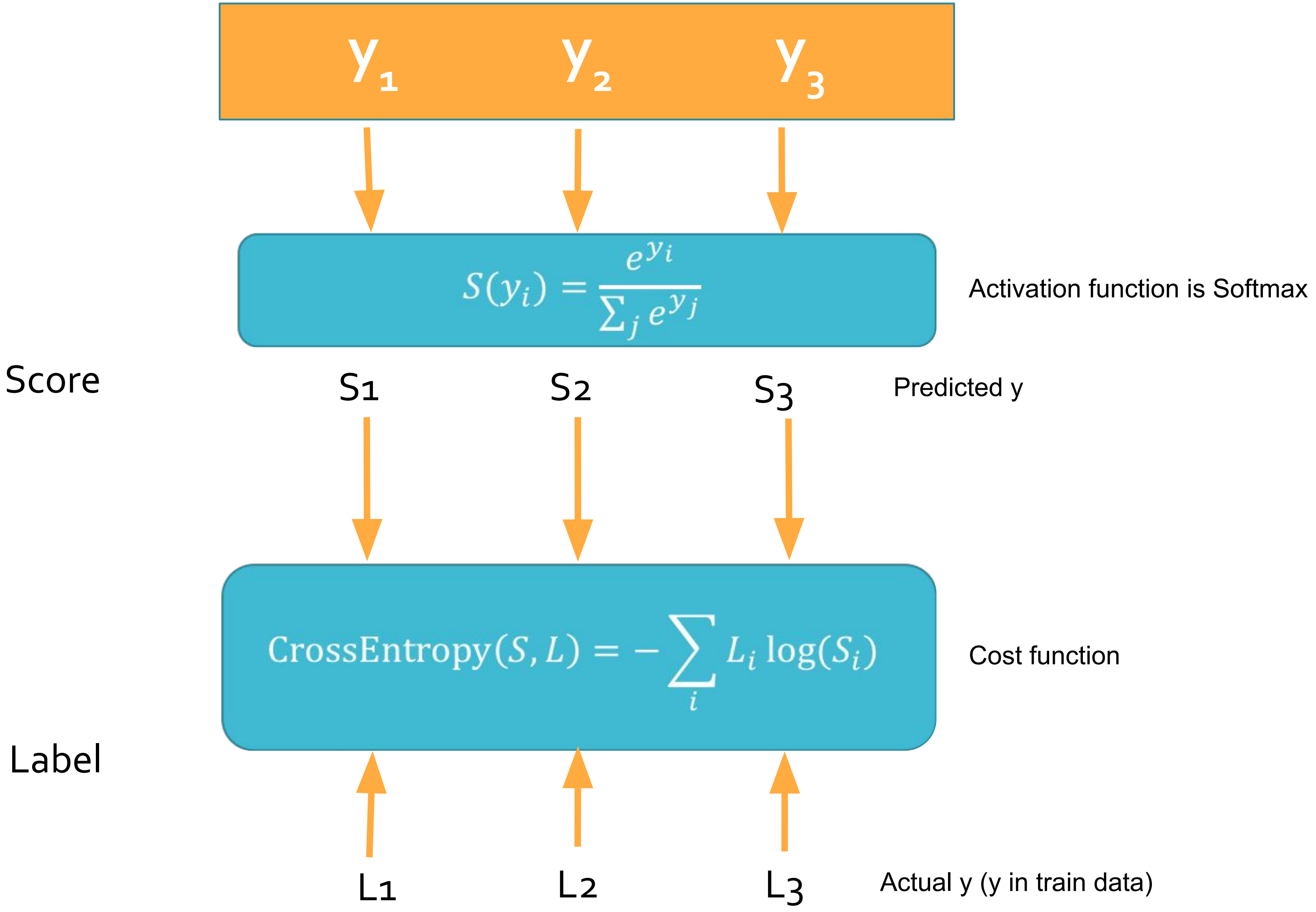
In practice, we utilize two specific loss functions: **Mean Squared Error** (MSE) and **Cross-Entropy**.

Cost function for **Binary class**

When activation function is sigmoid function,

$$Cost(W,b) = -y\log(\underbrace{H(x)}_{\text{Predicted y}}) \quad -(1-\underbrace{y}_{\text{Actual y (y in train data)}})\log(1-H(x))$$

Cost function for Softmax (Multiclass)



y_1 y_2 y_3

$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$

Activation function is Softmax

Score

S1 S2 S3

Predicted y

$\text{CrossEntropy}(S, L) = -\sum_i L_i \log(S_i)$

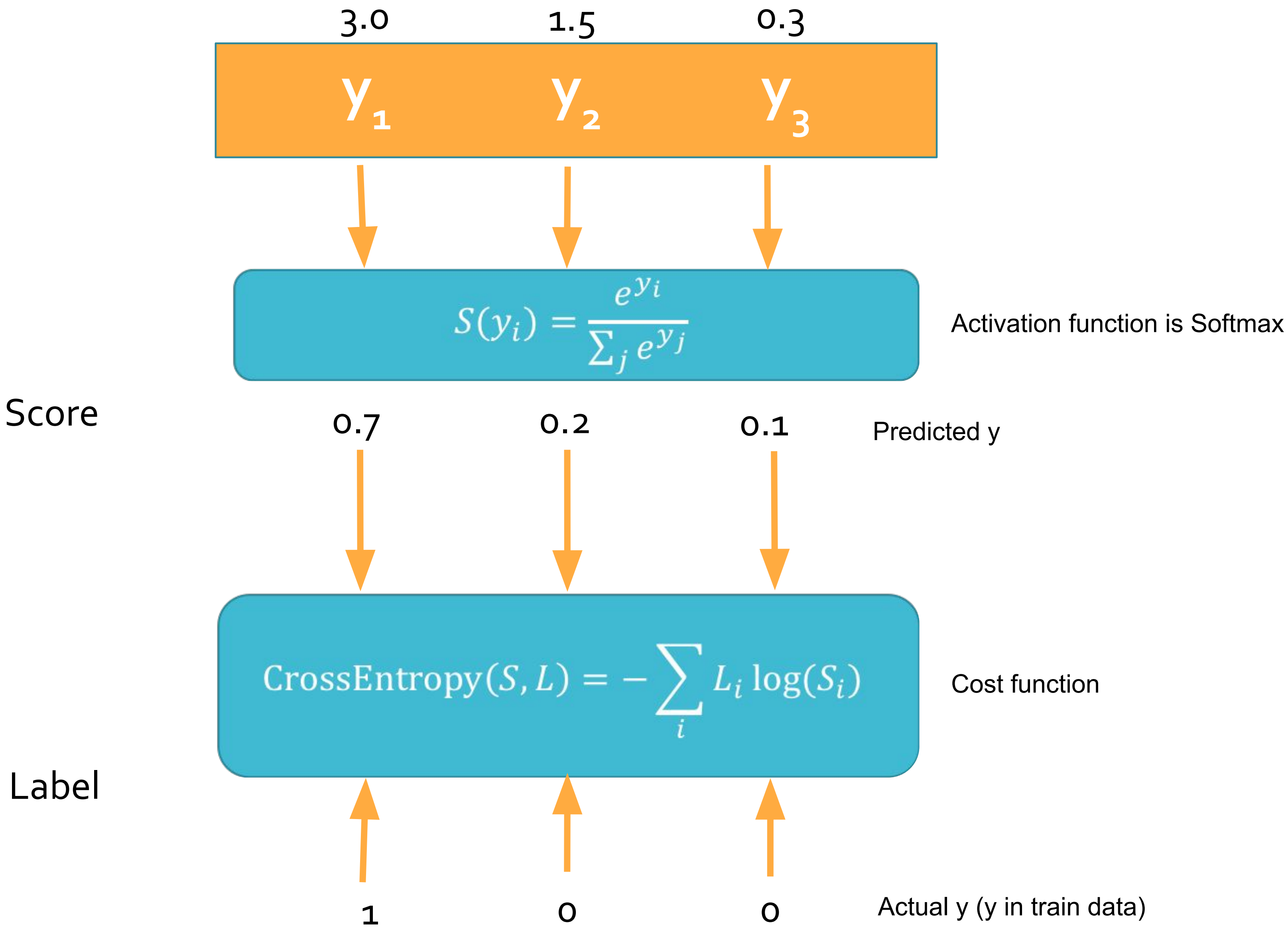
Cost function

Label

L1 L2 L3

Actual y (y in train data)

Cost function for Softmax (Multiclass)



3.0 1.5 0.3

y_1 y_2 y_3

$$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

Activation function is Softmax

Score

0.7 0.2 0.1

Predicted y

$$\text{CrossEntropy}(S, L) = -\sum_i L_i \log(S_i)$$

Cost function

Label

1 0 0

Actual y (y in train data)

Training - Feedforward and Backpropagation

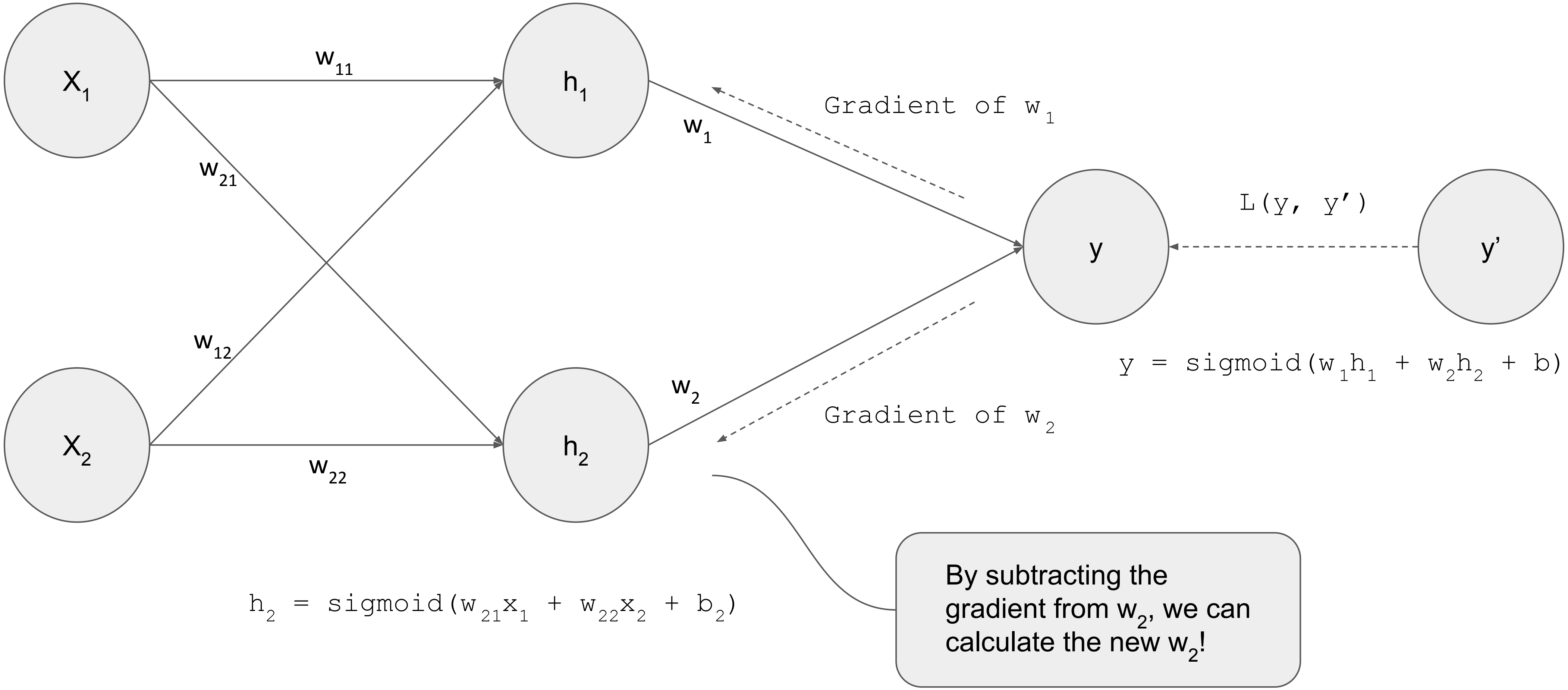
The process of training in deep learning can be broadly divided into two steps: feedforward and backpropagation.

During the **feedforward** step, the training data is input into the network, and it traverses through the entire architecture to compute the predictions based on the given data. Specifically, each neuron applies transformations (weighted sums and activation functions) to the information received from the previous layer's neurons and sends this output to the neurons in the next layer. Once the values have been propagated through all the layers, the output value is calculated at the final output layer.

Training - Feedforward and Backpropagation

The **backpropagation** step follows feedforward. Here, the difference (or error) between the computed output value (the predicted value) and the actual target value, y, is calculated through a loss function. The goal is to minimize this error. To achieve this, the weights (w) in the network are updated using an optimization algorithm, such as gradient descent. The changes made to the weights in the output layer propagate back to the previous hidden layers, with the weights in each hidden layer being adjusted in turn. This error propagation continues until the input layer, leading to an adjustment of the weight values across all layers in the network.

h_1 = sigmoid($w_{11}x_1$ + $w_{12}x_2$ + b_1)



y = sigmoid(w_1h_1 + w_2h_2 + b)

h_2 = sigmoid($w_{21}x_1$ + $w_{22}x_2$ + b_2)

By subtracting the gradient from w_2, we can calculate the new w_2!

Backpropagation

Before we dive into differentiating within the context of neural networks, let's take a moment to review some basic principles of differentiation. The derivative of the following expressions are as follows, correct?

$$f(x) = 3 \qquad \frac{\partial f}{\partial x} = 0$$

$$f(x) = x \qquad \frac{\partial f}{\partial x} = 1$$

$$f(x) = 2x \qquad \frac{\partial f}{\partial x} = 2$$

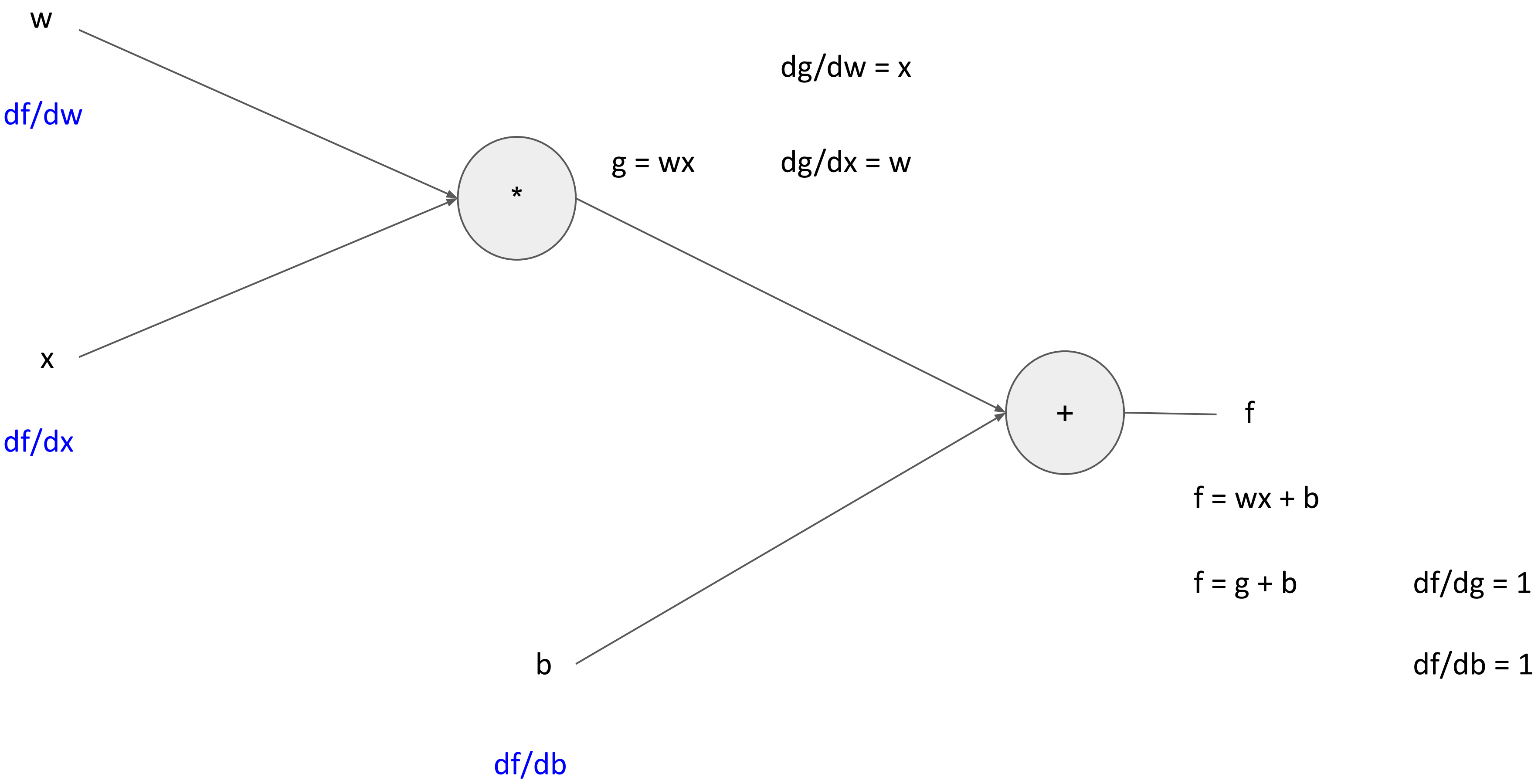
Backpropagation

Let's also review the chain rule.

Given $f(g(x))$, $\frac{\partial f}{\partial x} = \frac{\partial g}{\partial x} \cdot \frac{\partial f}{\partial g}$

Backpropagation

To explain backpropagation, let's take a look at a simple example.



Backpropagation

Here is an explanation of a neural network that models the function f = wx + b.

This network is composed of input nodes w, x, b, where '*' is the activation function in the hidden layer, and '+' is in the output layer.

Our focus is on determining the impact of the input values, namely w, x, and b, on the final output of the function.

Understanding the gradient is crucial for adjusting these input values.

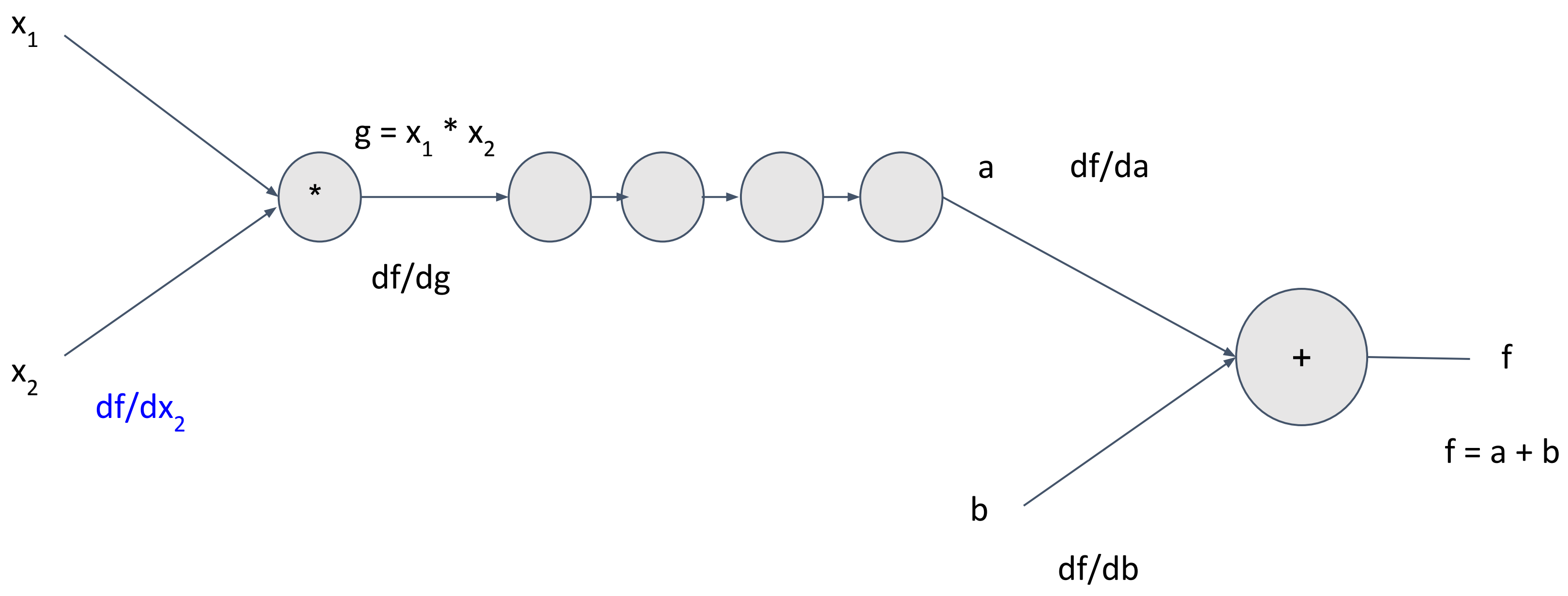
To find the derivative of f with respect to w, denoted as df/dw, we utilize the chain rule.

This involves calculating df/dw as the product of dg/dw and df/dg.

Given that dg/dw is x and df/dg is 1, it follows that df/dw is equal to x.

Backpropagation

Let's take a look at a slightly more complex case, similar to a deep neural network.



$df/dx_1 = ?$ We want to know how much x1 affects to f

$df/dx_1 = dg/dx_1 * df/dg$

Backpropagation

In this example, our objective is to comprehend how x_1 influences the output of the final layer.

Specifically, we aim to determine the derivative with respect to x_1.

Even in a lengthy network, by knowing the derivative relative to the subsequent value (denoted as g), we can efficiently compute df/dx_1 by applying the chain rule.

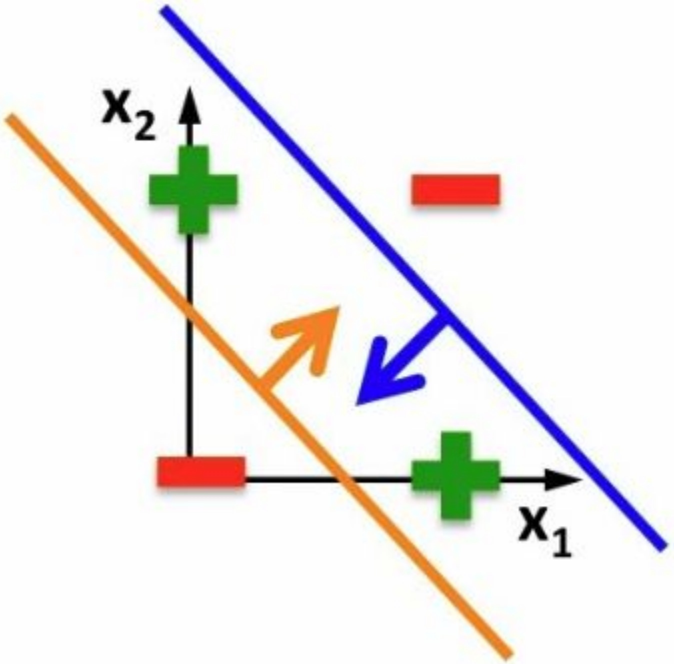
Indeed, the derivative concerning g can be determined from the derivative of the following hidden layer.

Utilizing the chain rule in this manner allows for the calculation of the derivative with respect to not only x_1 but also any weight w within the network.

Consequently, the process of derivative calculation begins at the output layer and progresses backward towards the input layer, sequentially.

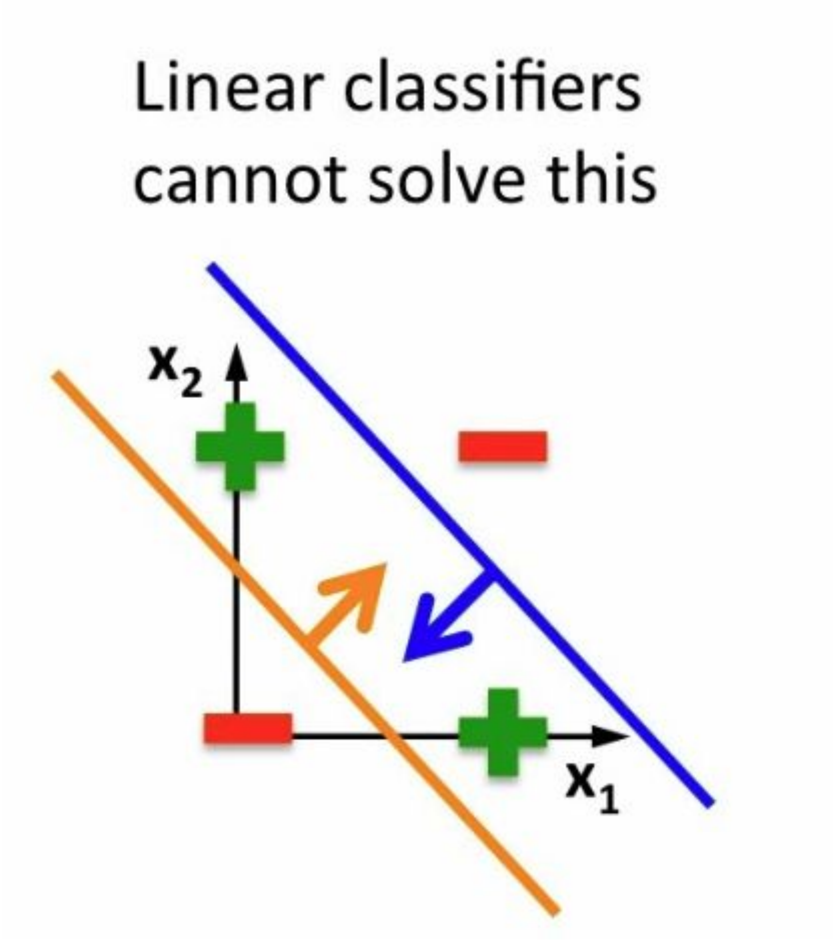
This systematic approach of working backwards through the network for derivative calculation is what gives the technique its name: backpropagation.

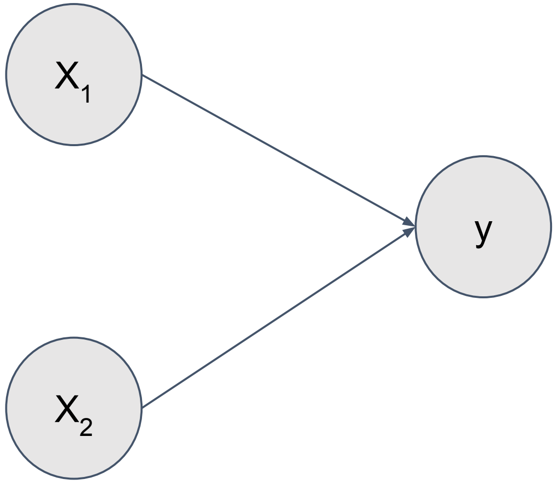
XOR problem



Linear classifiers cannot solve this

Logistic Regression (Non-linear classifier)

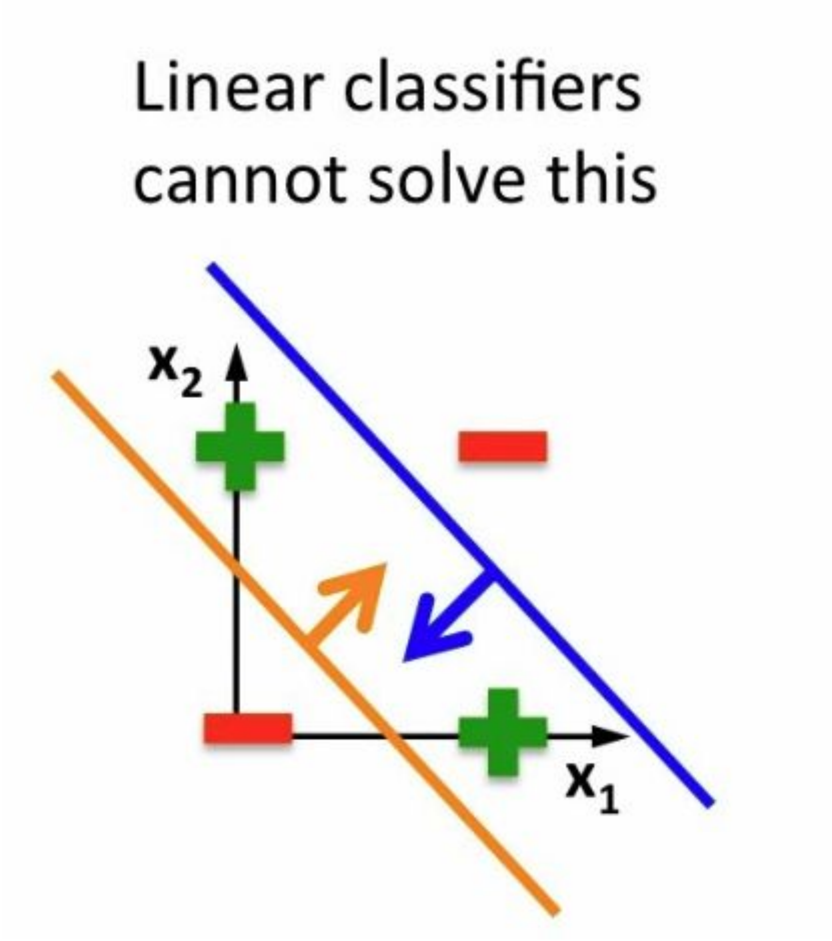




$y = \text{sigmoid}(w_1x_1 + w_2x_2 + b)$

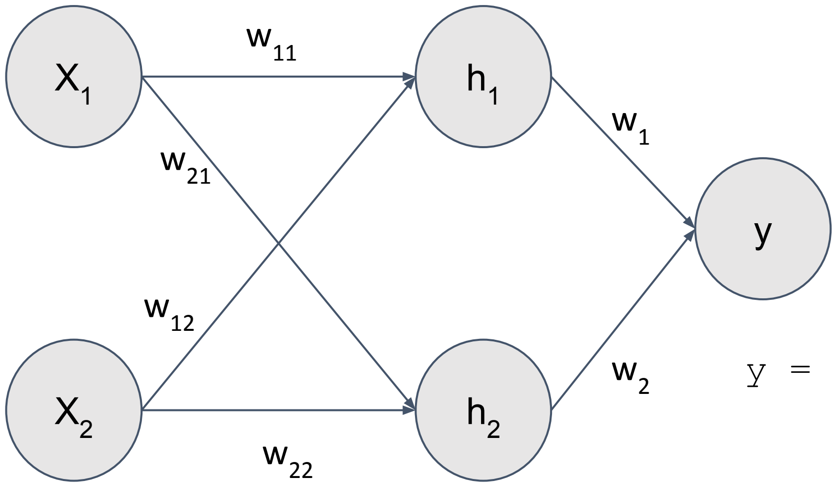
Equivalent to Neural net **without any hidden layers**

Neural Net (Non-linear classifier)



Linear classifiers cannot solve this

$h_1 = \text{sigmoid}(w_{11}x_1 + w_{12}x_2 + b_1)$



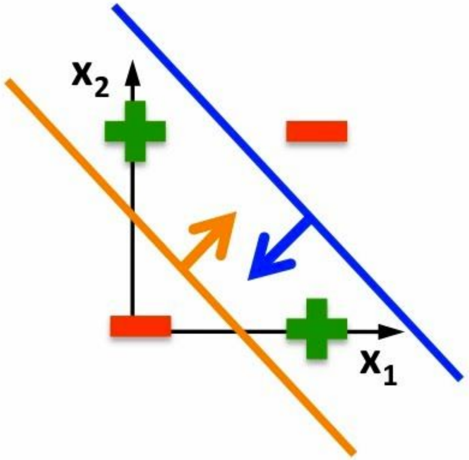
$y = \text{sigmoid}(w_1h_1 + w_2h_2 + b)$

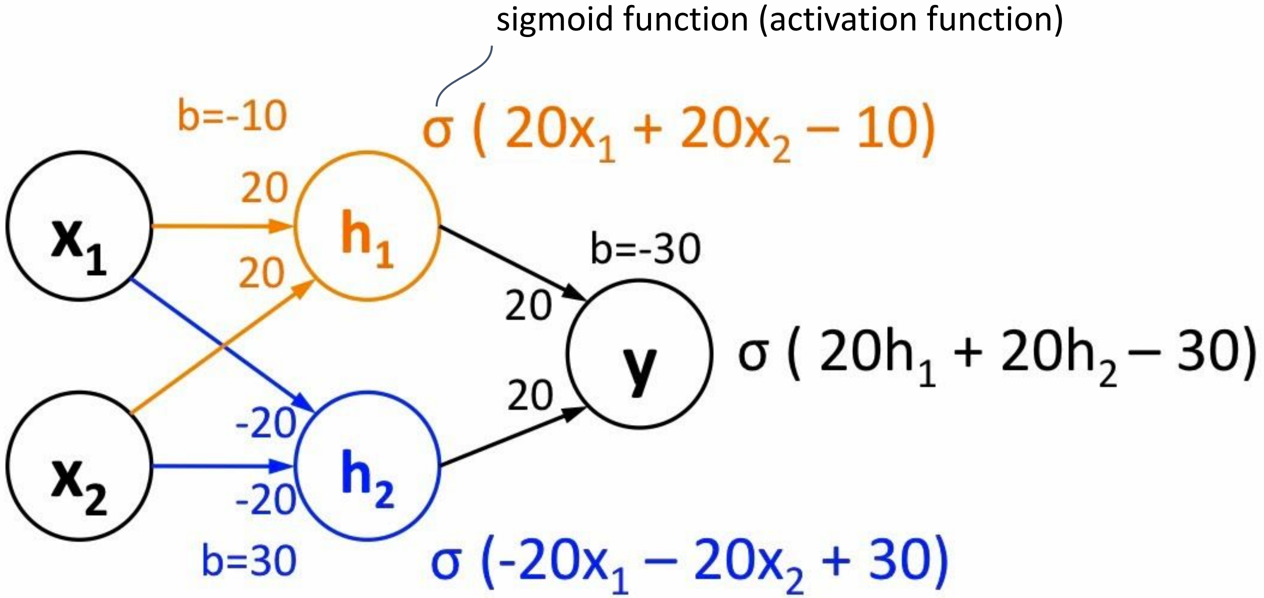
$h_2 = \text{sigmoid}(w_{21}x_1 + w_{22}x_2 + b_2)$

Neural net with a hidden layer

Solving XOR with a Neural Net

Linear classifiers cannot solve this





sigmoid function (activation function)
b=-10
$\sigma\ (\ 20x_1 + 20x_2 - 10)$
20
20
x_1
h_1
b=-30
20
y
$\sigma\ (\ 20h_1 + 20h_2 - 30)$
20
x_2
-20
h_2
-20
b=30
$\sigma\ (-20x_1 - 20x_2 + 30)$

$\sigma(20*0 + 20*0 - 10) \approx 0$

$\sigma(20*1 + 20*1 - 10) \approx 1$

$\sigma(20*0 + 20*1 - 10) \approx 1$

$\sigma(20*1 + 20*0 - 10) \approx 1$

$\sigma\ (-20*0 - 20*0 + 30) \approx 1$

$\sigma\ (-20*1 - 20*1 + 30) \approx 0$

$\sigma\ (-20*0 - 20*1 + 30) \approx 1$

$\sigma\ (-20*1 - 20*0 + 30) \approx 1$

$\sigma\ (20*0 + 20*1 - 30) \approx 0$

$\sigma\ (20*1 + 20*0 - 30) \approx 0$

$\sigma\ (20*1 + 20*1 - 30) \approx 1$

$\sigma\ (20*1 + 20*1 - 30) \approx 1$

NN in Pytorch

Pytorch codes

Review these codes.

16_Pytorch_basics.ipynb

16_Pytorch_Data_Loading.ipynb

16_Pytorch_Define_Model.ipynb

16_Pytorch_Model_parameter_settings_and_training.ipynb

16_Pytorch_MNIST_Lab_full.ipynb

Lab15

Part 1

Before you start this lab, review the Pytorch example.

16_Pytorch_MNIST_Lab_full.ipynb

Part 2

Implement a neural network and train with **Titanic data** (train data).

https://www.kaggle.com/c/titanic/data

Measure an accuracy with test data.

Submit your code (.ipynb or .py) and captured accuracy in blackboard.