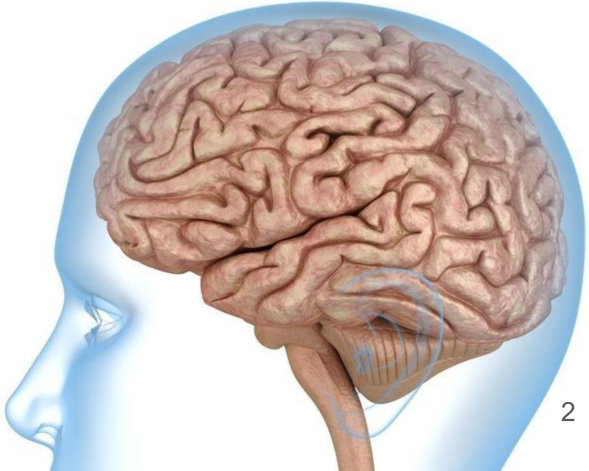
# The Components of Neural Networks

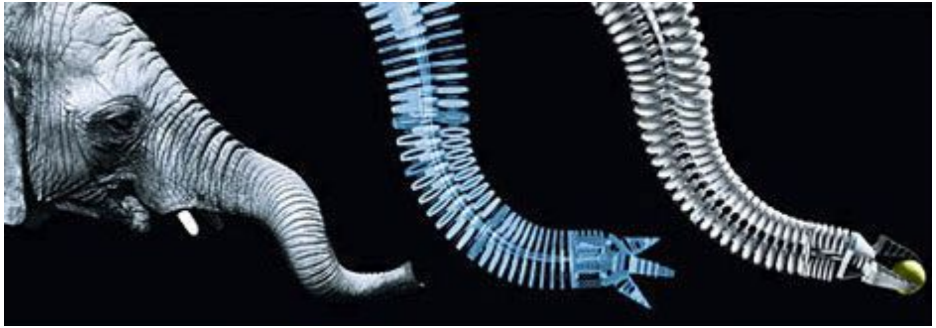
## Neuron, Perceptron, and Multilayer Perceptron

N. Rich Nguyen, PhD

**SYS 6016**

# 0. Inspiration from the human brain

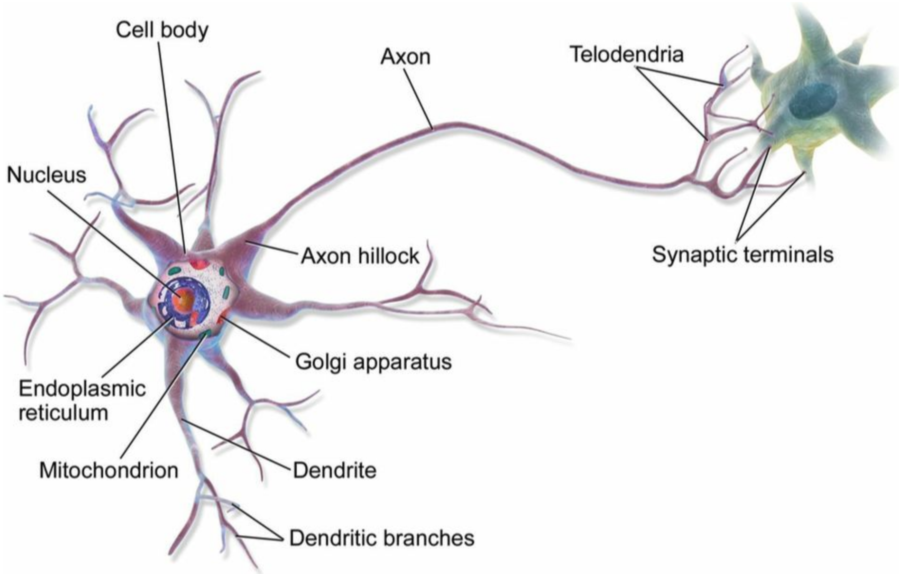


**Nature has inspired many inventions!**

# Biological Neurons

- Perhaps the brain is the nature’s ultimate inspiration for invention?
- It starts with an unusual looking cells found in animal cerebral cortex (brain)
- Neuron receives short **electrical impulses** from other neurons via synapses.
- When receives a **sufficient** number of signals, **fires its own signals**



# Biological Neural Networks (BNNs)

- Individual neurons seems simple
- They are organized in a vast network of billions in consecutive layers
- Each neuron connected to thousands of other neurons.

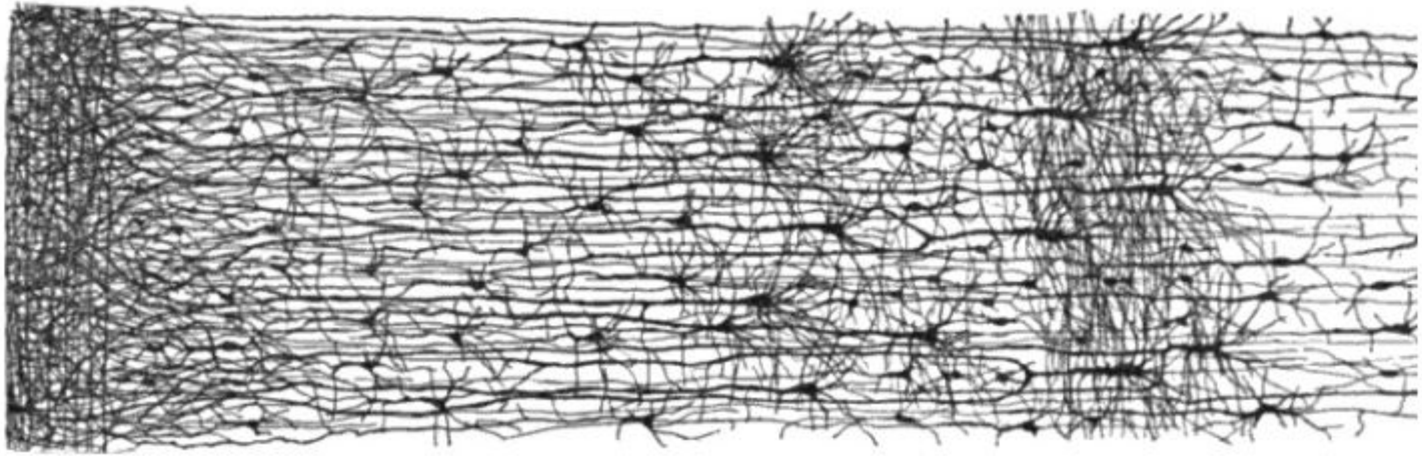
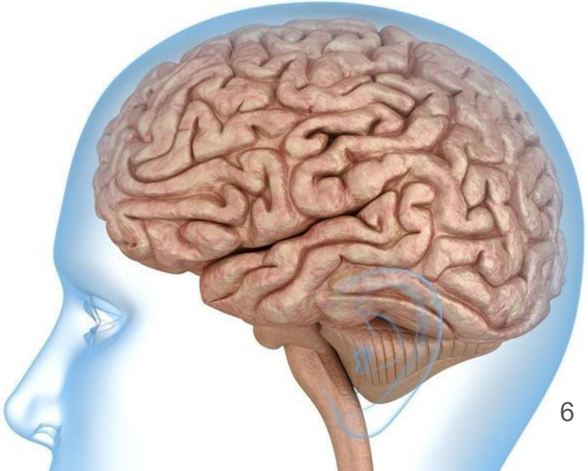


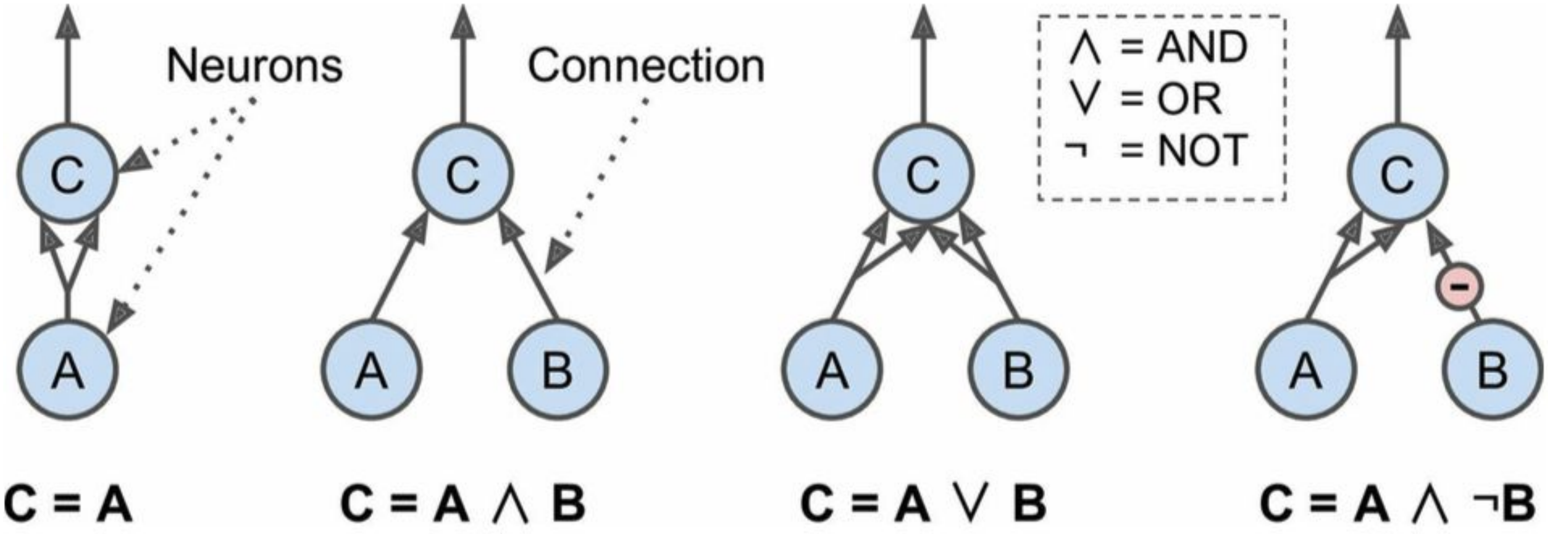
*Figure 10-2. Multiple layers in a biological neural network (human cortex)*[5]

# 1. Neurons and Perceptrons



# Logical Computations with Neurons

- **Artificial Neuron**: has one or more binary (on/off) inputs and one binary output
- A neural network of a few neurons can perform various logical computations
- Assuming a neuron can be activated when both of its input are active



# The Linear Threshold Unit (LTU)

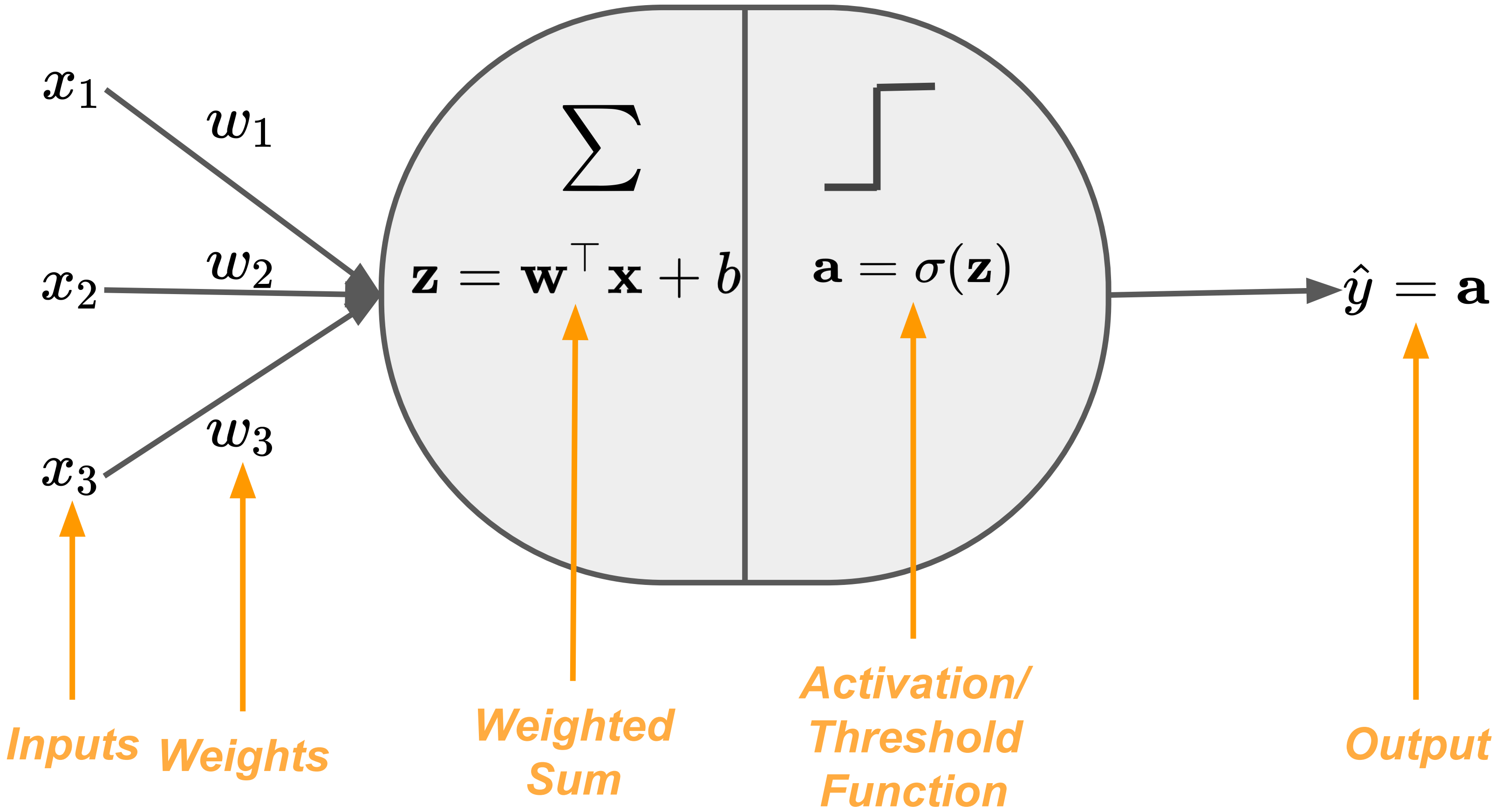
- LTU is based on an **artificial neuron**
- Insteads of binary inputs and output, LTU has numeric ones, and each input connection is associated with a weight.
- Essentially, LTU computes a **weighted sum** of its inputs:

$$z = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n = \mathbf{w}^\top \mathbf{x}$$

- LTU then applied a step function to return an output:

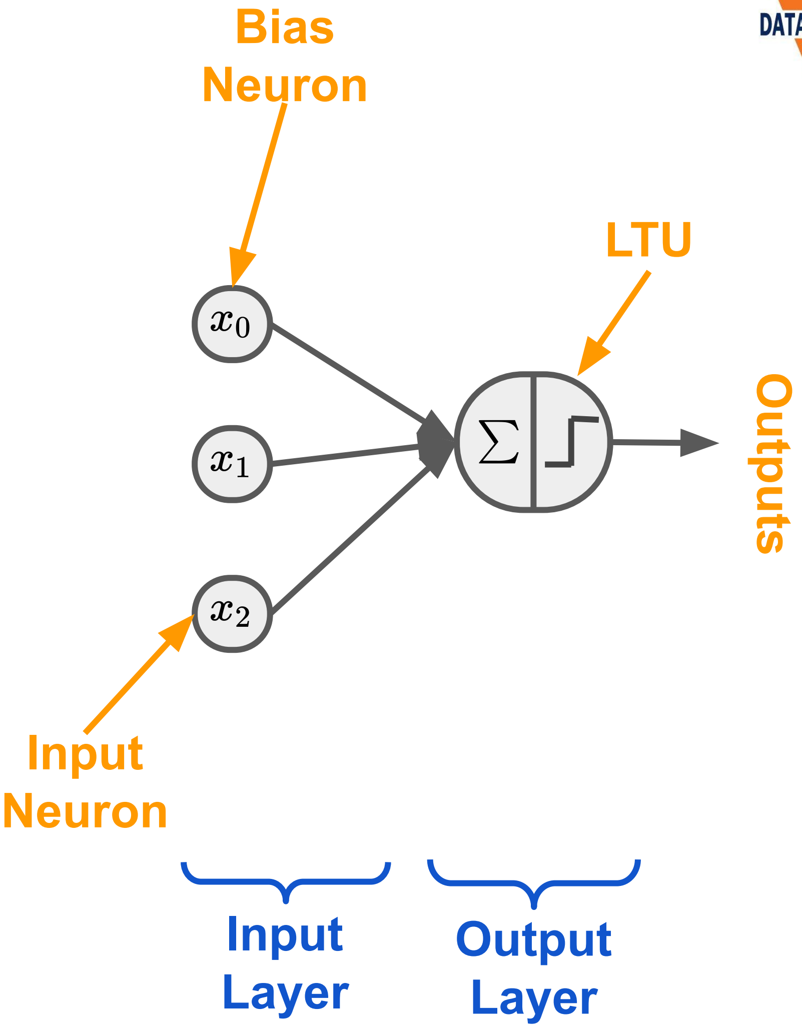
$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \qquad \text{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

# Graphical representation of a LTU



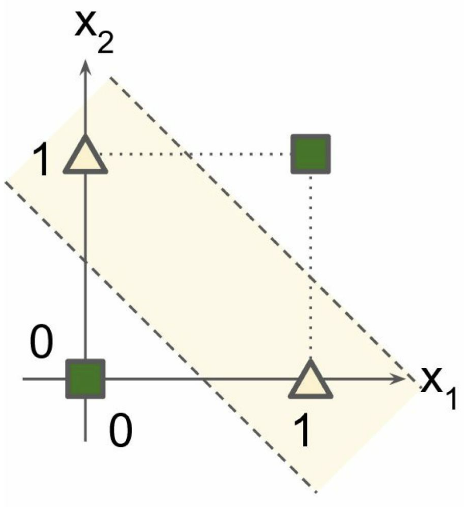
# The Perceptron

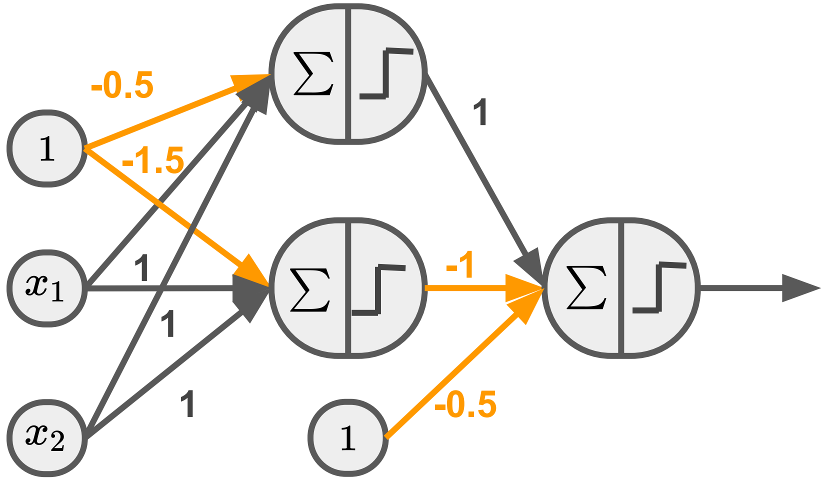
- One of the simplest neural network architecture (invented in 1957 by Frank Rosenblatt)
- Composed of a layer with LTU, and each neuron connected to all the inputs → input neuron
- An extra bias feature is generally added ($x_0$ =1) → bias neuron



# Linear Systems and XOR Classification Problem

| INPUT | | OUTPUT |
|---|---|---|
| A | B | A XOR B |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

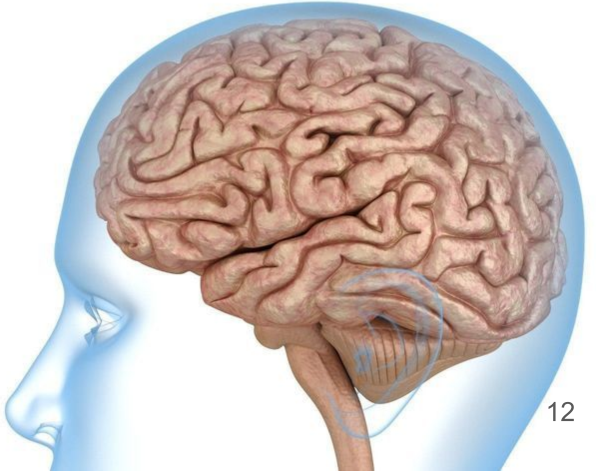



XOR is not linearly separable, cannot be solved with a single layer of LTU

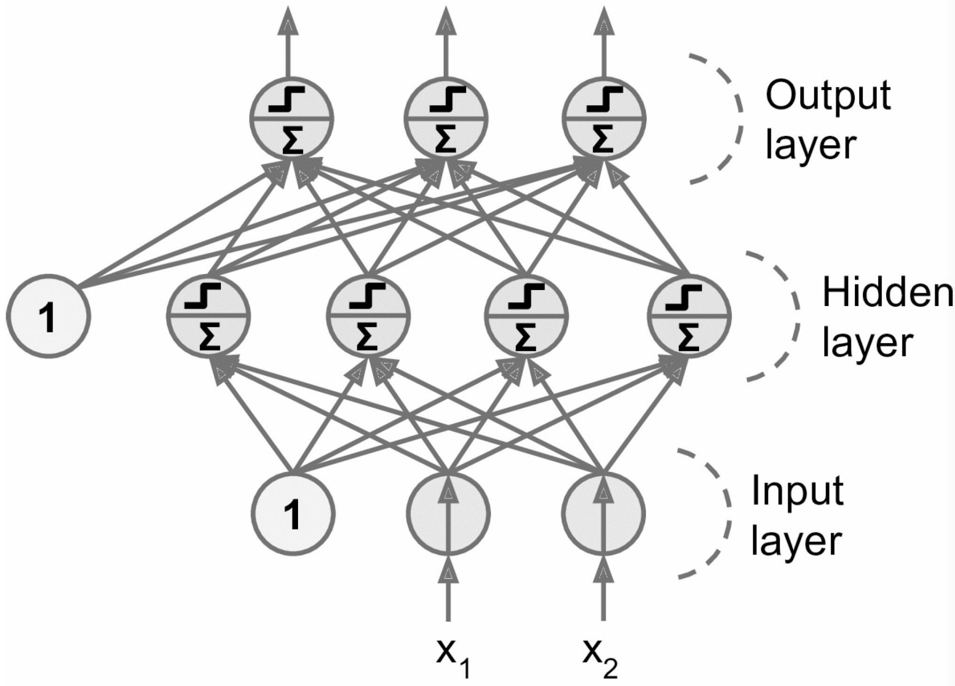
→ **stacking** multiple layers of Perceptrons can overcome this limitation

# 2. Multilayer Perceptron (MLP)

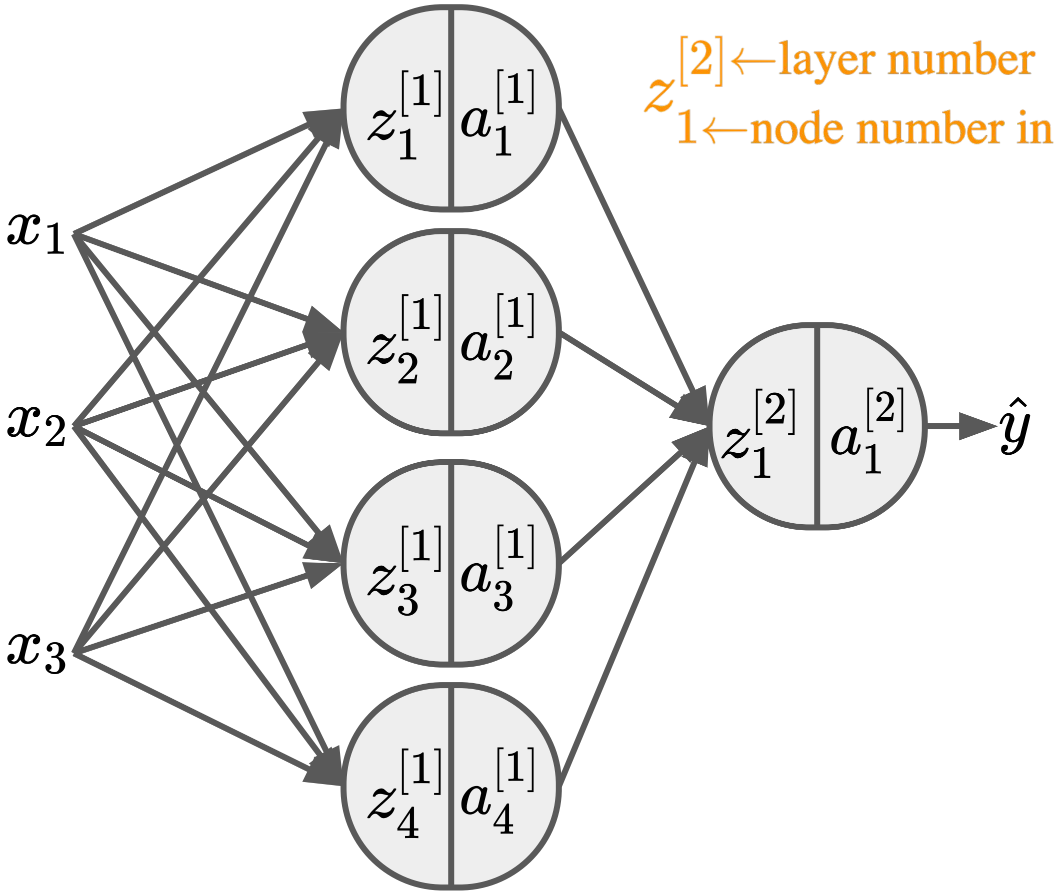


# Multi-Layer Perceptron (MLP)

- Composed of:
  - 1 (passthrough) input layer
  - 1 or more layers of LTUs (called hidden layers)
  - 1 final layer of LTU (output layer).
- When an MLP has **2+** hidden layers, it is called **Deep Neural Network** (DNN)



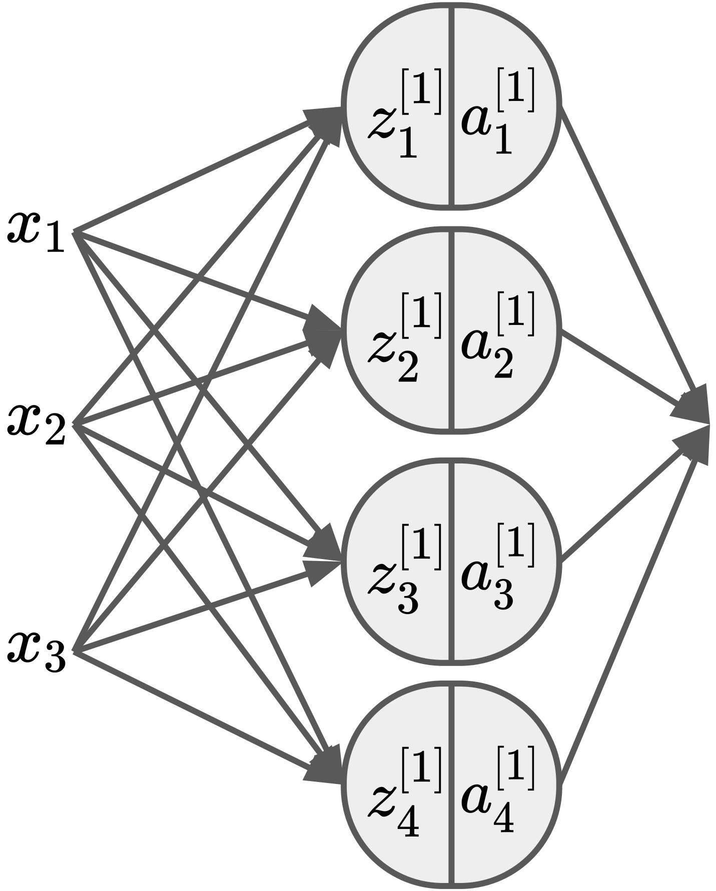
# Representation of a MLP




$z_1^{[2]}$ [2]←layer number, 1←node number in the layer

$$z_1^{[1]} = \mathbf{w}_1^{[1]\top}\mathbf{x} + b_1^{[1]}$$
$$a_1^{[1]} = \sigma\left(z_1^{[1]}\right)$$
$$z_2^{[1]} = \mathbf{w}_2^{[1]\top}\mathbf{x} + b_2^{[1]}$$
$$a_2^{[1]} = \sigma\left(z_2^{[1]}\right)$$
$$z_3^{[1]} = \mathbf{w}_3^{[1]\top}\mathbf{x} + b_3^{[1]}$$
$$a_3^{[1]} = \sigma\left(z_3^{[1]}\right)$$
$$z_4^{[1]} = \mathbf{w}_4^{[1]\top}\mathbf{x} + b_4^{[1]}$$
$$a_4^{[1]} = \sigma\left(z_4^{[1]}\right)$$
$$z_1^{[2]} = \mathbf{w}_1^{[2]\top}\mathbf{a}^{[1]} + b_1^{[2]}$$
$$a_1^{[2]} = \sigma\left(z_1^{[2]}\right)$$
$$\hat{y} = a_1^{[2]}$$

# Vectorizing by stacking them vertically

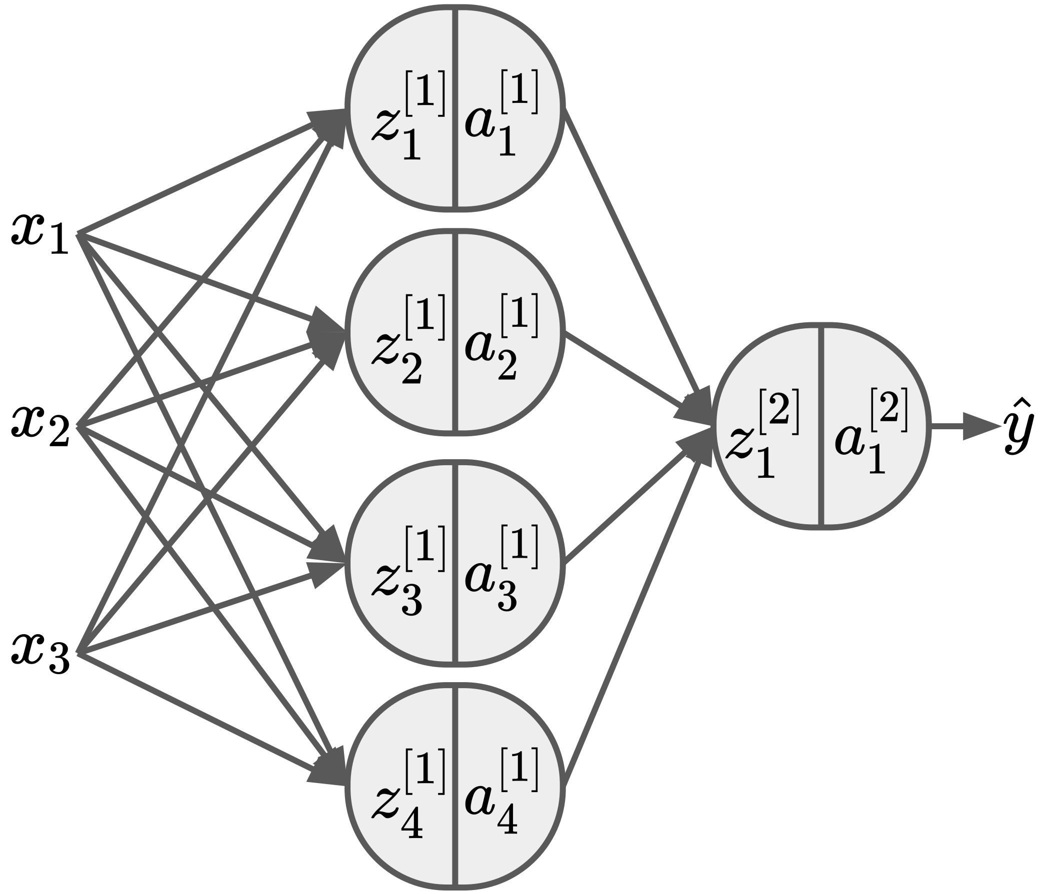



$$\boxed{\begin{matrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{matrix}} \begin{matrix} = \\ = \\ = \\ = \end{matrix} \boxed{\begin{matrix} \mathbf{w}_1^{[1]\top} \\ \mathbf{w}_2^{[1]\top} \\ \mathbf{w}_3^{[1]\top} \\ \mathbf{w}_4^{[1]\top} \end{matrix}} \begin{matrix} \mathbf{x} + \\ \mathbf{x} + \\ \mathbf{x} + \\ \mathbf{x} + \end{matrix} \boxed{\begin{matrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{matrix}} \qquad \boxed{\begin{matrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{matrix}} \begin{matrix} = \\ = \\ = \\ = \end{matrix} \boxed{\begin{matrix} \sigma(z_1^{[1]}) \\ \sigma(z_2^{[1]}) \\ \sigma(z_3^{[1]}) \\ \sigma(z_4^{[1]}) \end{matrix}}$$

$$\mathbf{z}^{[1]} \qquad \mathbf{W}^{[1]} \qquad \mathbf{b}^{[1]} \qquad \mathbf{a}^{[1]} \qquad \sigma(\mathbf{z}^{[1]})$$

$$\mathbf{z}^{[1]} = \mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]} \qquad \mathbf{a}^{[1]} = \sigma(\mathbf{z}^{[1]})$$

# Dimensionality of vectorized components




$$\underset{4\text{x}1}{\mathbf{z}^{[1]}} = \underset{4\text{x}3}{\mathbf{W}^{[1]}}\underset{3\text{x}1}{\mathbf{x}} + \underset{4\text{x}1}{\mathbf{b}^{[1]}}$$

$$\underset{4\text{x}1}{\mathbf{a}^{[1]}} = \sigma(\underset{4\text{x}1}{\mathbf{z}^{[1]}})$$

$$\underset{1\text{x}1}{\mathbf{z}^{[2]}} = \underset{1\text{x}4}{\mathbf{W}^{[2]}}\underset{4\text{x}1}{\mathbf{a}^{[1]}} + \underset{1\text{x}1}{\mathbf{b}^{[2]}}$$

$$\underset{1\text{x}1}{\mathbf{a}^{[2]}} = \sigma(\underset{1\text{x}1}{\mathbf{z}^{[2]}})$$

# Expanding for multiple examples

$$\mathbf{x} \longrightarrow \mathbf{a}^{[2]} = \hat{y}$$

$$\mathbf{x}^{(1)} \longrightarrow \mathbf{a}^{[2](1)} = \hat{y}^{(1)}$$

$$\mathbf{x}^{(2)} \longrightarrow \mathbf{a}^{[2](2)} = \hat{y}^{(2)}$$

$$\vdots$$

$$\mathbf{x}^{(m)} \longrightarrow \mathbf{a}^{[2](m)} = \hat{y}^{(m)}$$

$$\text{for } i = 1 \text{ to } m$$

$$\mathbf{z}^{[1](i)} = \mathbf{W}^{[1]}\mathbf{x}^{(i)} + \mathbf{b}^{[1]}$$

$$\mathbf{a}^{[1](i)} = \sigma(\mathbf{z}^{[1](i)})$$

$$\mathbf{z}^{[2](i)} = \mathbf{W}^{[2]}\mathbf{a}^{[1](i)} + \mathbf{b}^{[2]}$$

$$\mathbf{a}^{[2](i)} = \sigma(\mathbf{z}^{[2](i)})$$

$n^{[1]}$ is number of nodes in layer 1

# Vectorizing for multiple examples

$$\mathbf{X} = \begin{bmatrix} | & | & & | \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \\ | & | & & | \end{bmatrix} \quad \mathbf{Z}^{[1]} = \begin{bmatrix} | & | & & | \\ \mathbf{z}^{[1](1)} & \mathbf{z}^{[1](2)} & \cdots & \mathbf{z}^{[1](m)} \\ | & | & & | \end{bmatrix} \quad \mathbf{A}^{[1]} = \begin{bmatrix} | & | & & | \\ \mathbf{a}^{[1](1)} & \mathbf{a}^{[1](2)} & \cdots & \mathbf{a}^{[1](m)} \\ | & | & & | \end{bmatrix}$$

$n^{[0]}$xm $\qquad$ $n^{[1]}$xm $\qquad$ $n^{[1]}$xm

for $i = 1$ to $m$

$$\mathbf{z}^{[1](i)} = \mathbf{W}^{[1]}\mathbf{x}^{(i)} + \mathbf{b}^{[1]}$$
$$\mathbf{a}^{[1](i)} = \sigma\left(\mathbf{z}^{[1](i)}\right)$$
$$\mathbf{z}^{[2](i)} = \mathbf{W}^{[2]}\mathbf{a}^{[1](i)} + \mathbf{b}^{[2]}$$
$$\mathbf{a}^{[2](i)} = \sigma\left(\mathbf{z}^{[2](i)}\right)$$

$$\Longrightarrow$$

$$\mathbf{Z}^{[1]} = \mathbf{W}^{[1]}\mathbf{X} + \mathbf{B}^{[1]}$$
$$\mathbf{A}^{[1]} = \sigma(\mathbf{Z}^{[1]})$$
$$\mathbf{Z}^{[2]} = \mathbf{W}^{[2]}\mathbf{A}^{[1]} + \mathbf{B}^{[2]}$$
$$\mathbf{A}^{[2]} = \sigma(\mathbf{Z}^{[2]})$$

# Tips on Dimensions of Neural Net Components

$$\mathbf{X} = \begin{bmatrix} | & | & & | \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \\ | & | & & | \end{bmatrix} \quad \mathbf{Z}^{[1]} = \begin{bmatrix} | & | & & | \\ \mathbf{z}^{[1](1)} & \mathbf{z}^{[1](2)} & \cdots & \mathbf{z}^{[1](m)} \\ | & | & & | \end{bmatrix} \quad \mathbf{A}^{[1]} = \begin{bmatrix} | & | & & | \\ \mathbf{a}^{[1](1)} & \mathbf{a}^{[1](2)} & \cdots & \mathbf{a}^{[1](m)} \\ | & | & & | \end{bmatrix}$$

$n^{[0]}$xm $n^{[1]}$xm $n^{[1]}$xm

$$\mathbf{W}^{[l]} = (n^{[l]} \times n^{[l-1]})$$

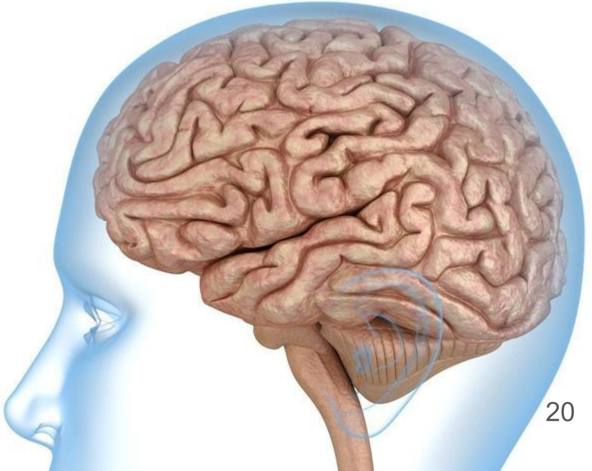
$$\mathbf{Z}^{[l]} = (n^{[l]} \times m)$$

$$\mathbf{A}^{[l]} = (n^{[l]} \times m)$$

$$\mathbf{B}^{[l]} = (n^{[l]} \times m)$$

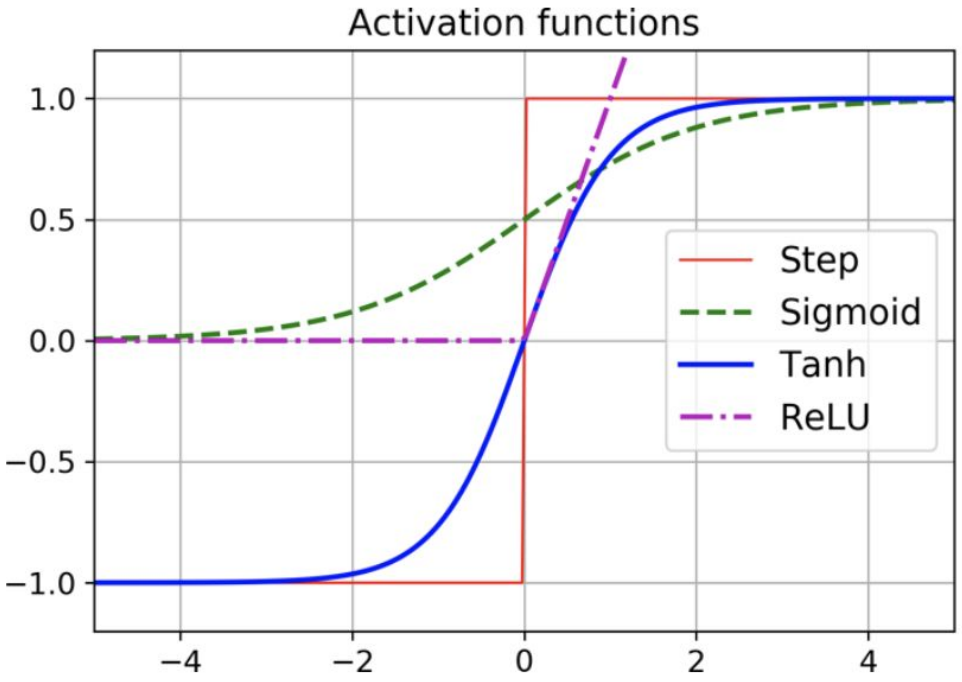
- $n^{[l]}$ is number of nodes in layer $l$
- $m$ is number of examples in a training batch
- Note: $\mathbf{B}^{[l]}$ consists of $m$ replications of vector $\mathbf{b}^{[l]}$ of dimension $n^{[l]}$x1

# 3.  Activation Functions



# Activation function a = $\sigma$(z)

There are a number of activation functions available:




Step: $a = 1 \text{ if } z > 0,$

$0 \text{ if } z < 0$

Sigmoid: $a = \frac{1}{1+e^{-z}}$

Tanh: $a = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

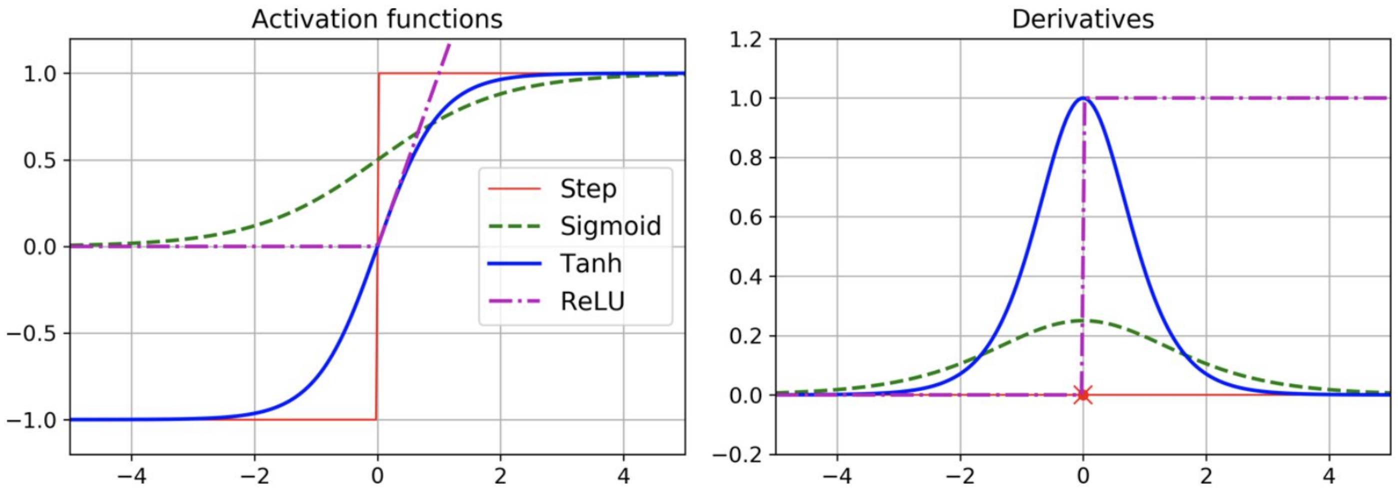
ReLU: $a = \max(0, z)$

# Why activation functions are necessary

- Without activation function, each layer would consist of **linear** operations (a dot product and an addition), so the layer can only learn linear transformations of the input data.
- Adding a deep stack of linear layers would still implement a linear operation ⇒ we need **non-linearity** to gain access to a much richer representation of the input data
- An activation function computes a **non-linear** transformation.
- Most neural networks describe the features using an affine (linear) transformation controlled by the learned parameters, followed by a fixed (non-linear) activation function.

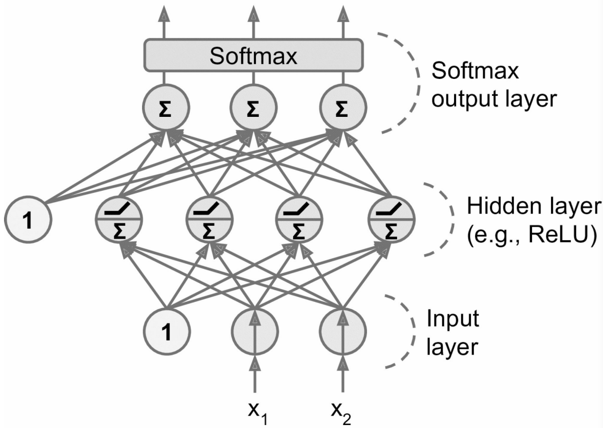
# Learning with activation functions

- Step Function has **zero** derivative → does not work with Gradient Descent
- Use Sigmoid function (and other below) with well-defined non-zero derivative → allow **Gradient Descent** to make progress
- ReLU makes the derivatives **large** and **consistent** whenever it is active (> 0)



# Softmax Function

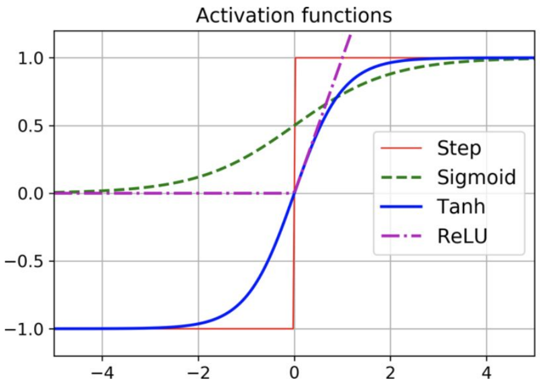
- **Any time** you wish to represent a **probability distribution** over a discrete variable with `n` possible values, you may use the **softmax function**
- Softmax is most used as outputs, which is sum to 1, of a classifier of `n` classes
- Softmax *exponentiates* and *normalizes* the inputs to obtain desired outputs
- Softmax can be seen as a generalization of the sigmoid (for binary variable)
- Softmax provides a “softened” version of the **argmax** (returns a one-hot vector)




$$\mathrm{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \text{where } z_i = \log P(y = i|\mathbf{x})$$

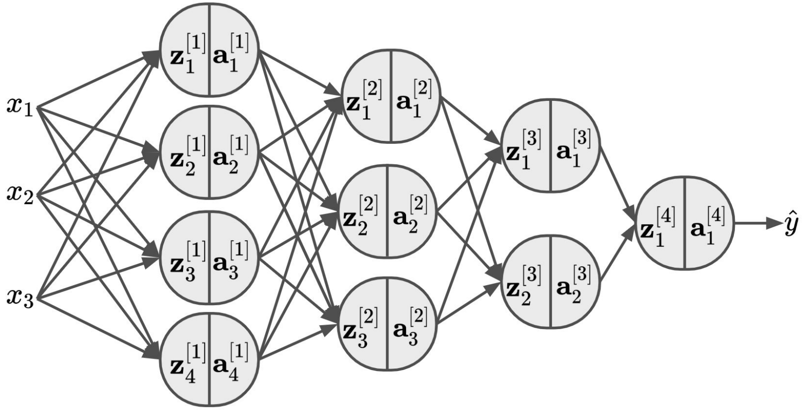
$$\log \mathrm{softmax}(\mathbf{z})_i = z_i - \log \sum_j \exp(z_j)$$
$$\approx z_i - \max_j z_j$$

# Tips on Activation Functions




- **Step** function does not work with Gradient-based Learning
- **ReLU** is faster to compute and easy to optimize as its behavior is closer to linear
- **Sigmoid** and **Hyperbolic Tangent (tanh)** function saturate at 1
- Training with **tanh** is easier than **sigmoid** as it's similar to the identity function near 0
- For classification tasks, **Softmax** function is a good choice
- For regression tasks, **no need** for activation function

# <u>Summary</u>: Network Architecture Design




- In general, most neural network architectures arrange in a *chain* structure, with each layer being a function of the layer that preceded it.
- The main consideration is choosing the **depth** of the network and the **width** of each layer
- Deeper networks often use fewer units per layer, far few parameters, but they also tend to be **harder to optimize**
- The ideal network architecture for a task must be found via *experimentation* guided by monitoring the *validation error*.

# Bonus Content

# Training a Perceptron

"*Cells that fire together, wire together*" - Hebb's rule

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta(\hat{y}_j - y_j)x_i$$

- $w_{i,j}$ is the connection weight between $i^{th}$ input neuron and $j^{th}$ output neuron
- $x_i$ is the $i^{th}$ input value of the current training sample
- $y\hat{}_j$ is the output of the $j^{th}$ neuron
- $y_j$ is the target output of $j^{th}$ output neuron
- $n$ is the learning rate

**What does this learning algorithm remind you of?**

# Code example

Perceptron makes predictions based on a hard threshold (not class probability)

Also, it only works if the training samples are linearly separate.

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris()
X = iris.data[:, (2, 3)]  # petal length, petal width
y = (iris.target == 0).astype(np.int)

per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])
```