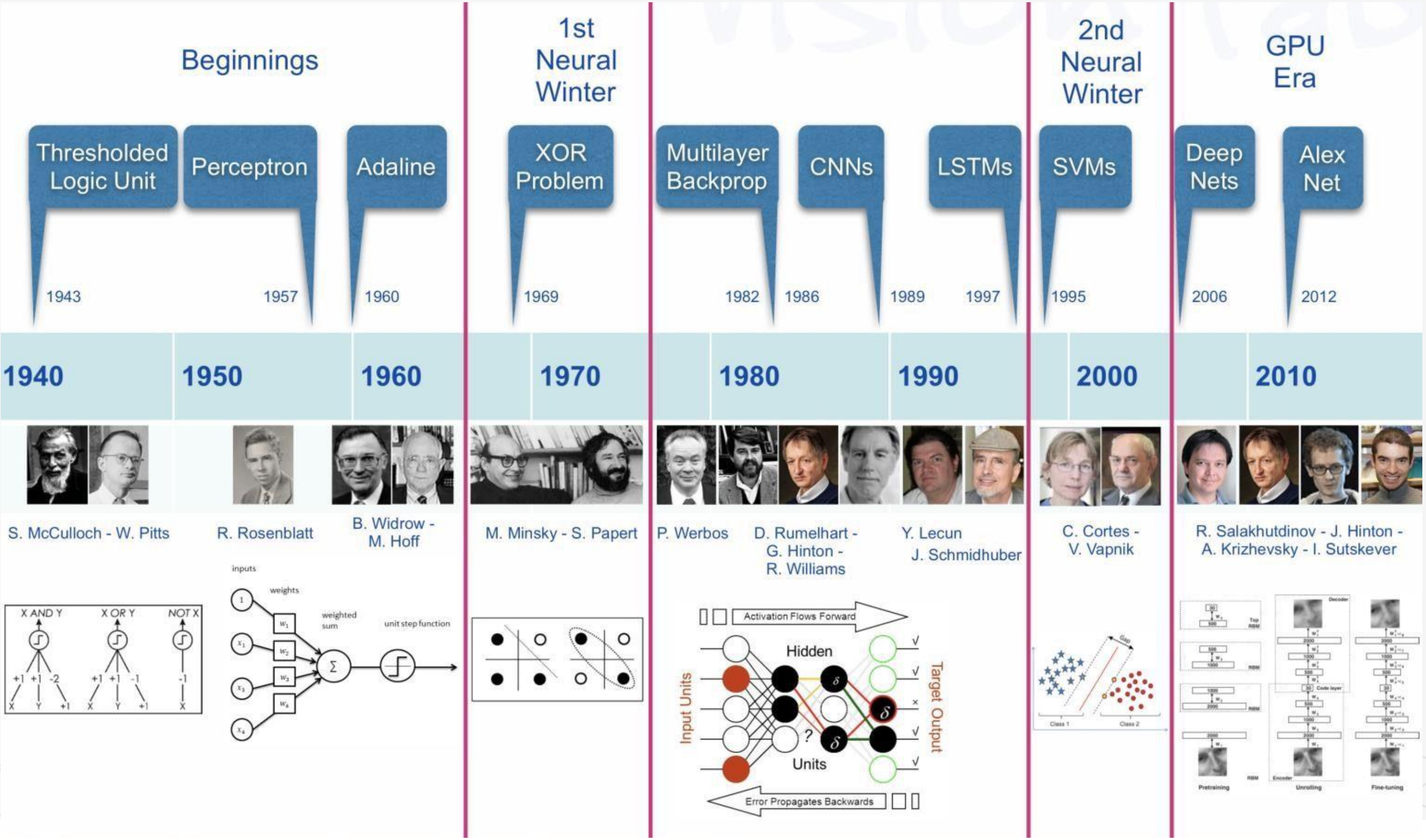
# Introduction to Deep Learning

Neural Networks

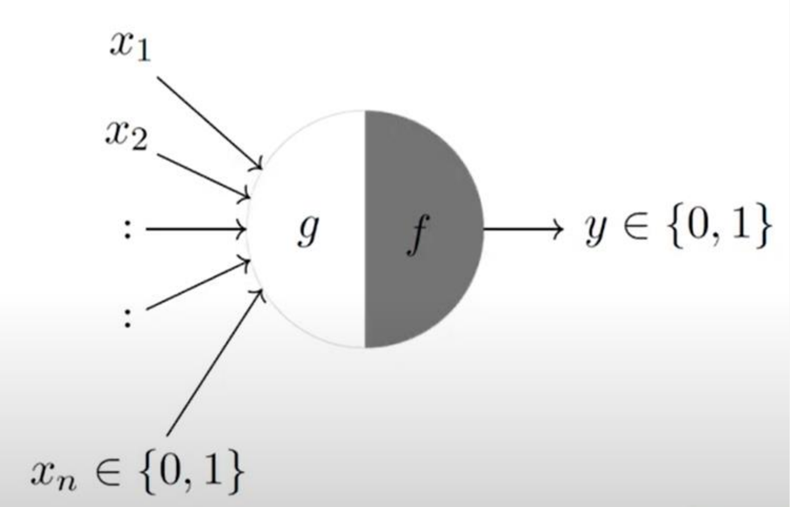
# History of neural networks



# McCulloch Pitts neuron

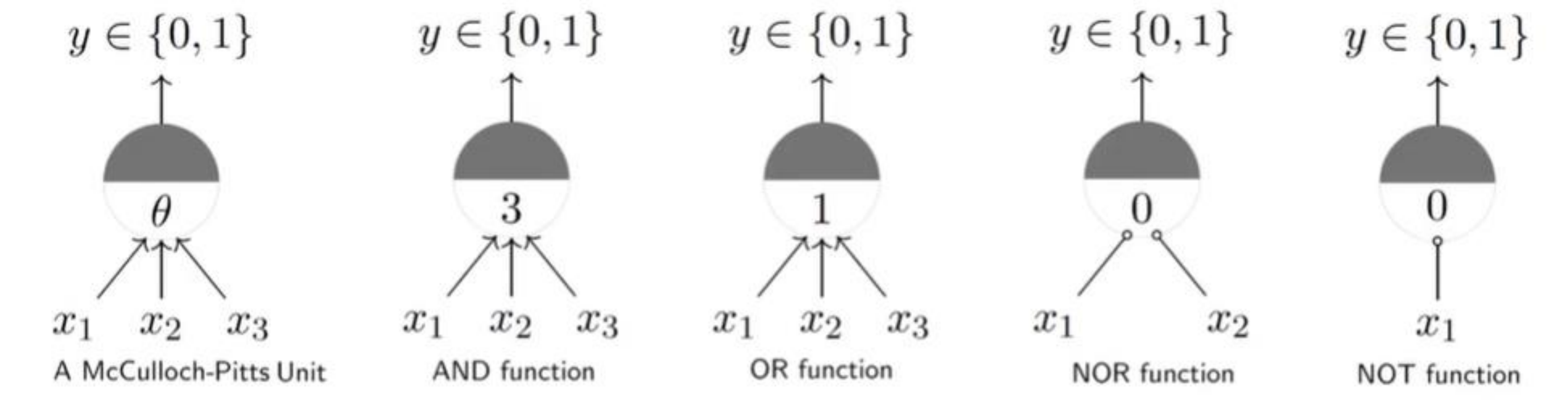
- McCulloch (neuroscientist) and Pitts (logician) proposed a highly simplified computational model of the neuron (1943)
- $g$ aggregates the inputs and the function $f$ takes a decision based on this aggregation
- The inputs can be excitatory or inhibitory
- $y = 0$ if any $x_i$ is inhibitory, else

$$
\begin{aligned}
g(x_1, x_2, \dots, x_n) &= g(\mathbf{x}) = \textstyle\sum_{i=1}^{n} x_i \\
y = f(g(\mathbf{x})) &= 1 \text{ if } g(\mathbf{x}) \geq \theta \\
&= 0 \text{ if } g(\mathbf{x}) < \theta
\end{aligned}
$$




- Theta is a thresholding parameter

# McCulloch Pitts neuron



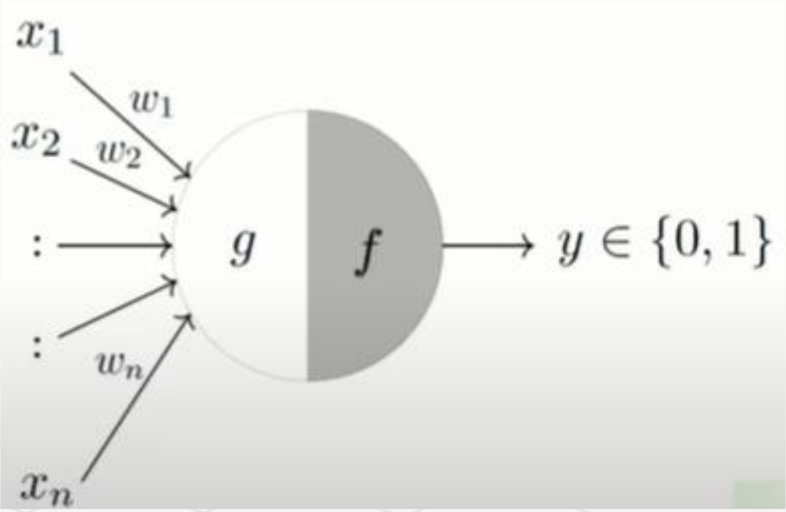

- Feedforward MP networks can compute any Boolean function f:{0,1}^n→{0,1}
- Recursive MP networks can simulate any Deterministic Finite Automaton (DFA) (See this paper below for more information)

Neural Networks: Automata and Formal Models of Computation, A Forcada, M.

# Perceptrons

- Frank Rosenblatt, an American psychologist, proposed the perceptron model (1958)
- Later refined and carefully analyzed by Minsky and Papert (1969)
- A more general computational model than McCulloch-Pitts neurons
- Inputs are no longer limited to Boolean values

$$
\begin{aligned}
g(x_1, x_2, \ldots, x_n) &= g(\mathbf{x}) = \textstyle\sum_{i=1}^{n} w_i * x_i \\
y = f(g(\mathbf{x})) &= 1 \text{ if } g(\mathbf{x}) \geq \theta \\
&= 0 \text{ if } g(\mathbf{x}) < \theta
\end{aligned}
$$



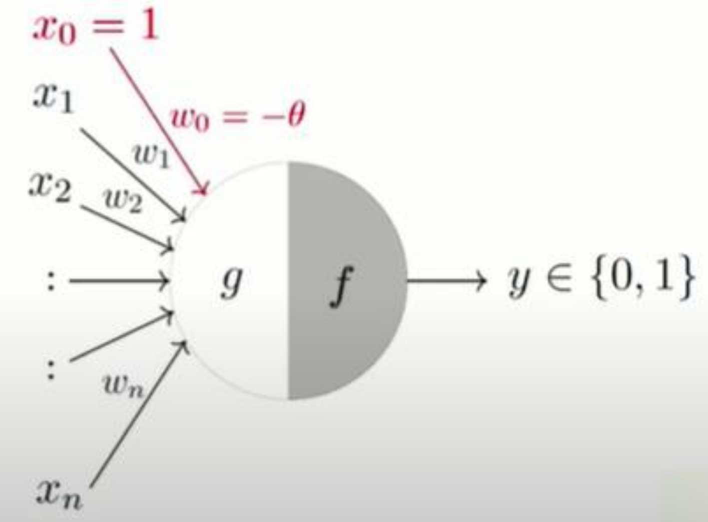
# Perceptrons

- Frank Rosenblatt, an American psychologist, proposed the perceptron model (1958)
- Later refined and carefully analyzed by Minsky and Papert (1969)
- A more general computational model than McCulloch-Pitts neurons
- Inputs are no longer limited to boolean values

$$g(x_1, x_2, \ldots, x_n) = g(\mathbf{x}) = \sum_{i=0}^{n} w_i * x_i$$

$$y = f(g(\mathbf{x})) \quad = 1 \text{ if } g(\mathbf{x}) \geq 0$$
$$= 0 \text{ if } g(\mathbf{x}) < 0$$

where $x_0 = 1$ and $w_0 = -\theta$



# Perceptron learning algorithm

**Algorithm:** Perceptron Learning Algorithm

$P \leftarrow inputs \quad with \quad label \quad 1;$
$N \leftarrow inputs \quad with \quad label \quad 0;$
Initialize $\mathbf{w}$ randomly;
**while** $!convergence$ **do**
  Pick random $\mathbf{x} \in P \cup N$ ;
  **if** $\mathbf{x} \in P \quad and \quad \mathbf{w.x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$ ;
  **end**
  **if** $\mathbf{x} \in N \quad and \quad \mathbf{w.x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$ ;
  **end**
**end**
//the algorithm converges when all the inputs are classified correctly

Why would this work?

We are interested in finding the line $\sum_{i=0}^{n} w_i * x_i = 0$ or $\mathbf{w}^\top \mathbf{x} = 0$ which divides the input space into two halves (binary classifier)

Every point (x) on this line satisfies the equation $\mathbf{w}^\top \mathbf{x} = 0$

What can you tell about the angle ($\alpha$) between $w$ and any point ($x$) which lies on this line?

The angle is $90°$ $\left(\because \cos \alpha = \frac{\mathrm{w}^{\mathrm{T}}\mathrm{x}}{\|\mathbf{w}\|\|\mathbf{x}\|}\right)$

Since the vector $w$ is perpendicular to every point on the line it is actually perpendicular to the line itself

# Perceptron learning algorithm

**Algorithm:** Perceptron Learning Algorithm

$P \leftarrow inputs \quad with \quad label \quad 1;$
$N \leftarrow inputs \quad with \quad label \quad 0;$
Initialize $\mathbf{w}$ randomly;
**while** $!convergence$ **do**
  Pick random $\mathbf{x} \in P \cup N$ ;
  **if** $\mathbf{x} \in P \quad and \quad \mathbf{w}.\mathbf{x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$ ;
  **end**
  **if** $\mathbf{x} \in N \quad and \quad \mathbf{w}.\mathbf{x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$ ;
  **end**
**end**
//the algorithm converges when all the inputs are classified correctly

What will be the angle between vector x $\in P$ and w ? Less than 90°

What will be the angle between vector **x** $\in N$ and w ? Greater than 90°

Ponder and convince yourself this is the case For x $\in P$ if $\mathbf{w}^{\top}\mathbf{x} < 0$

then it means that the angle $(\alpha)$ between this x and the current w is

greater than 90

But we want it to be less than 90

How is adding x to w helping us?

# Perceptron learning algorithm

**Algorithm:** Perceptron Learning Algorithm

$P \leftarrow inputs \quad with \quad label \quad 1;$
$N \leftarrow inputs \quad with \quad label \quad 0;$
Initialize $\mathbf{w}$ randomly;
**while** !*convergence* **do**
  Pick random $\mathbf{x} \in P \cup N$ ;
  **if** $\mathbf{x} \in P$ *and* $\mathbf{w}.\mathbf{x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$ ;
  **end**
  **if** $\mathbf{x} \in N$ *and* $\mathbf{w}.\mathbf{x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$ ;
  **end**
**end**
//the algorithm converges when all the inputs are classified correctly

What happens to the new angle ($\alpha_{\text{new}}$) when

$$\mathrm{w}_{\text{new}} = \mathrm{w} + \mathrm{x}$$

$$\begin{aligned}
\cos\alpha_{new} &\propto \mathbf{w}_{\mathbf{new}}^{\top}\mathbf{x} \\
&\propto (\mathbf{w} + \mathbf{x})^{\top}\mathbf{x} \\
&\propto \mathbf{w}^{\top}\mathbf{x} + \mathbf{x}^{\top}\mathbf{x} \\
&\propto \cos\alpha + \mathbf{x}^{\top}\mathbf{x} \\
\cos\alpha_{\text{new}} &> \cos\alpha
\end{aligned}$$

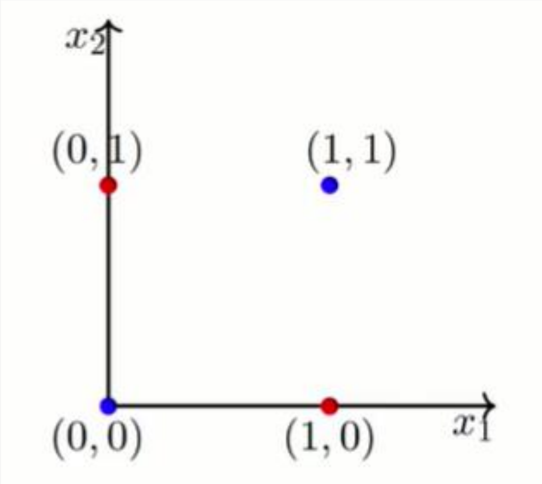
Thus $\alpha_{\text{new}}$ will be less than $\alpha$ and this is exactly what we want

We can work out the math for the other case, when $\mathrm{x} \in N$ and $\mathbf{w}^{\top}\mathbf{x} \geq 0$ quite similarly

For a formal convergence proof, please see this link
[Convergence Proof for the Perceptron Algorithm Michael Collins]

# The XOR Conundrum

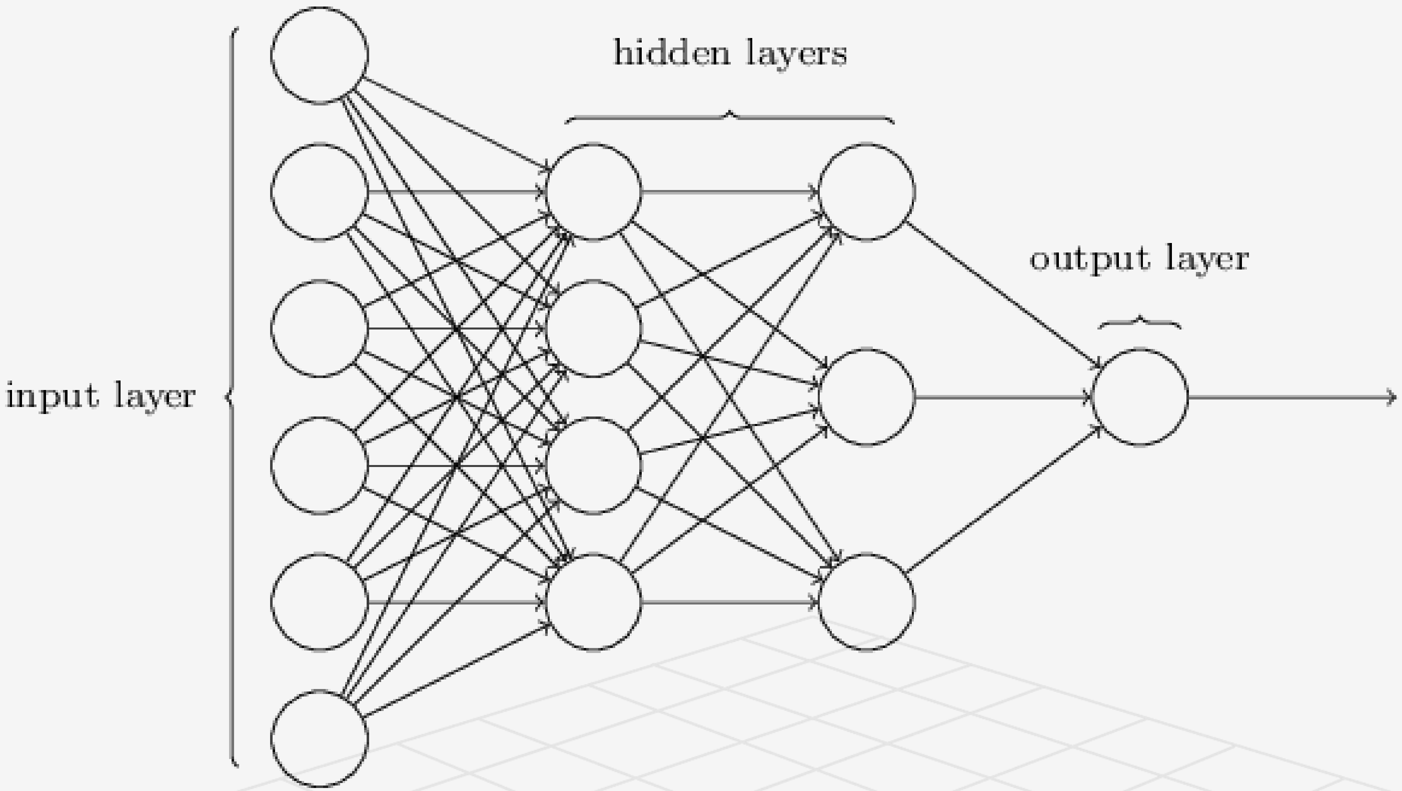
| $x_1$ | $x_2$ | XOR | |
|---|---|---|---|
| 0 | 0 | 0 | $w_0 + \sum_{i=1}^{2} w_i x_i < 0$ |
| 1 | 0 | 1 | $w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$ |
| 0 | 1 | 1 | $w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$ |
| 1 | 1 | 0 | $w_0 + \sum_{i=1}^{2} w_i x_i < 0$ |



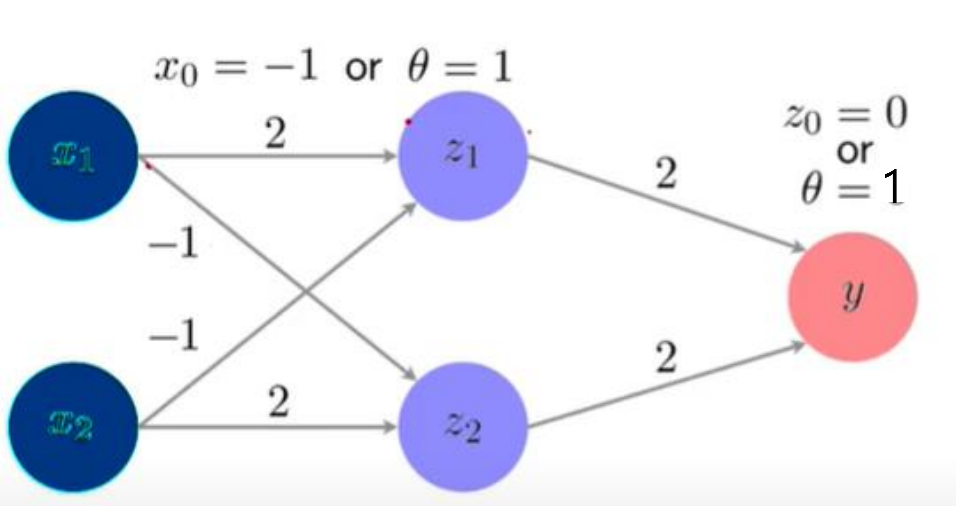

$$
\begin{aligned}
w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0 &\Rightarrow w_0 < 0 \\
w_0 + w_1 \cdot 1 + w_2 \cdot 0 \geq 0 &\Rightarrow w_1 > -w_0 \\
w_0 + w_1 \cdot 0 + w_2 \cdot 1 \geq 0 &\Rightarrow w_2 > -w_0 \\
w_0 + w_1 \cdot 1 + w_2 \cdot 1 < 0 &\Rightarrow w_1 + w_2 < -w_0
\end{aligned}
$$

- The fourth condition contradicts the 2nd and 3rd condition
- No solutions possible satisfying this set of inequalities

# Multi Layer Perceptrons



# Multi Layer Perceptrons




| $(x_1, x_2)$ | $(z_1, z_2)$ | $y$ |
|---|---|---|
| (0,0) | (0,0) | 0 |
| (0,1) | (0,0) | 1 |
| (1,0) | (1,0) | 1 |
| (1,1) | (0,0) | 0 |

# Multi Layer Perceptrons

**Theorem:** Any boolean function of $n$ inputs can be represented exactly by a network of perceptrons containing 1 hidden layer with $2^n$ perceptrons and one output layer containing 1 perceptron

**Proof (Informal):** How? Each of the $2^n$ hidden layer perceptrons can model (or can be fired by) one combination of $n$ inputs.

**Note:** A network of $2^n + 1$ perceptrons is not necessary but sufficient
But why does this matter?

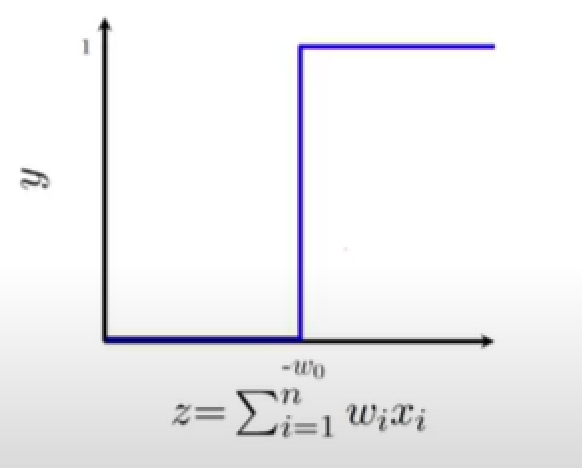
# Going beyond binary inputs and outputs

Question

- What about arbitrary functions of the form $y = f(x)$ where $x \in \mathbf{R}^n$ (instead of $\{0,1\}^n$ ) and $y \in \mathbf{R}$ (instead of $\{0,1\}$) ?
- Can we use the same perceptron model to represent such functions?

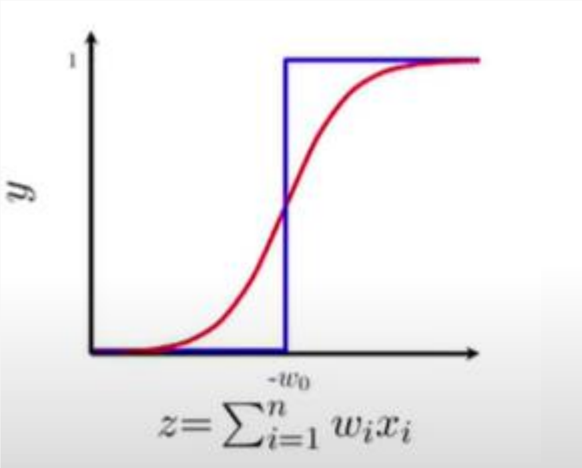
# Need for activation function

- A perceptron only fires if weighted sum of its inputs is greater than threshold $-w_0$
- Thresholding logic could be harsh at times
- E.g., when $-w_0 = 0.5$, though output values $0.49$ and $0.51$ are very close to each other, the perceptron would assign different labels to them.
- This behavior is not a characteristic of the problem, the weights or the threshold; it is a characteristic of the perceptron function itself which behaves like a step function
- There will always be a sudden change in decision (from 0 to 1) when $\sum_{i=1}^{n} w_i x_i$ crosses the threshold $(-w_0)$
- For most real-world applications, we'd expect a smoother decision function which gradually changes from 0 to 1



# Need for activation function

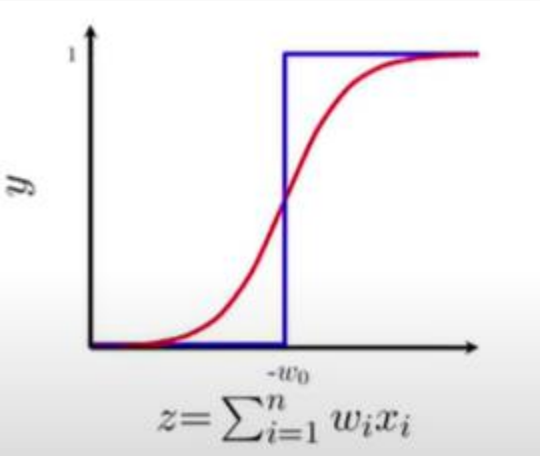
- A perceptron only fires if weighted sum of its inputs is greater than threshold $-w_0$
- Thresholding logic could be harsh at times
- E.g., when $-w_0 = 0.5$, though output values $0.49$ and $0.51$ are very close to each other, the perceptron would assign different labels to them.
- This behavior is not a characteristic of the problem, the weights or the threshold; it is a characteristic of the perceptron function itself which behaves like a step function
- There will always be a sudden change in decision (from 0 to 1) when $\sum_{i=1}^{n} w_i x_i$ crosses the threshold $(-w_0)$
- For most real-world applications, we'd expect a smoother decision function which gradually changes from 0 to 1



# Sigmoid neurons

- We could use any logistic function to obtain a smoother output function than a step function
  One form is the sigmoid function:

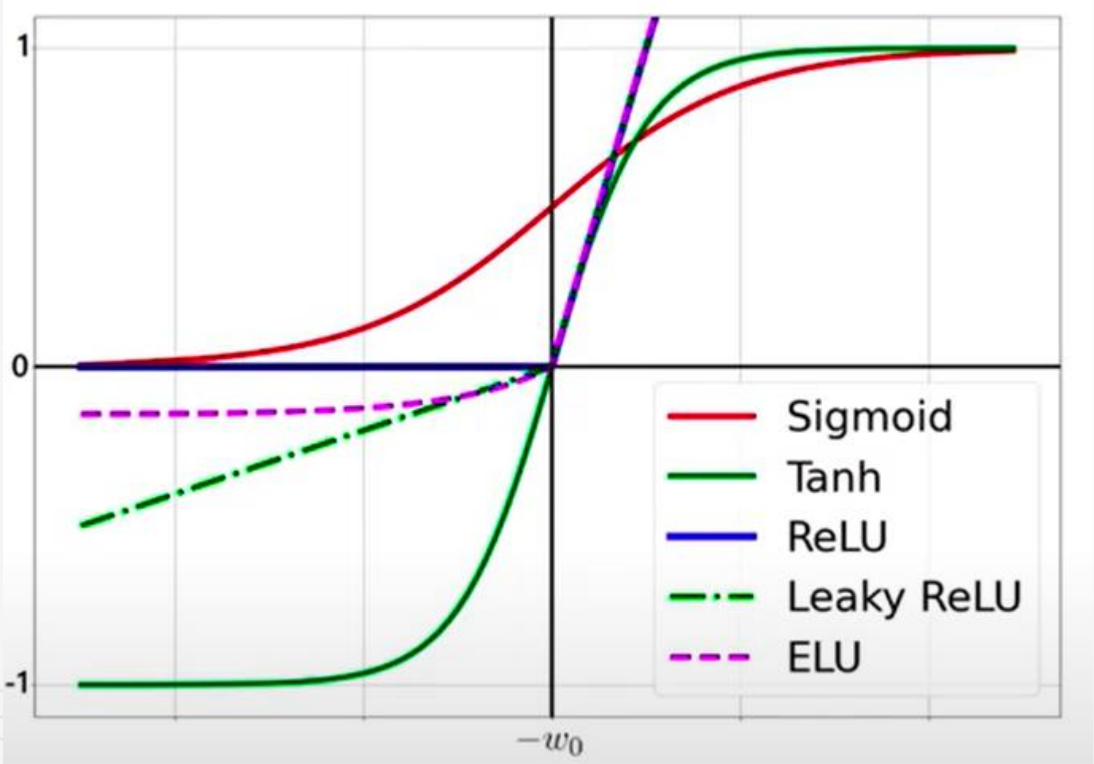
$$y = \frac{1}{1 + e^{-(w_0 + \sum_{i=0}^{n} w_i x_i)}}$$




- No longer a sharp transition at the threshold $-w_0$
- Also, output is no longer binary but a real value between 0 and 1 which can be interpreted as a probability
- Unlike the step function, this one is smooth, continuous at $-w_0$ and most importantly **differentiable**

# Other popular activation functions

- Considering $z = \sum_{i=0}^{n} w_i x_i$
- Sigmoid: $y = \frac{1}{1+e^{-z}}$
- Tanh: $y = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
- Rectified Linear Unit (ReLU): $y = \max(0, z)$
- Leaky ReLU: $y = \max(\alpha z, z), \alpha \in (0,1)$
- Exponential Linear Unit (ELU):
  $y = \max(\alpha(e^z - 1), z)$, where $\alpha > 0$

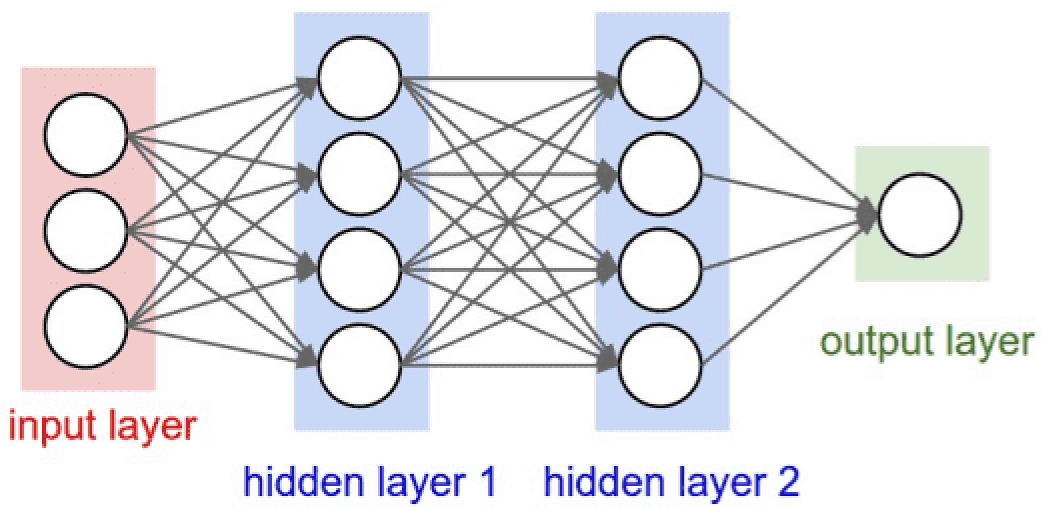


# Homework

- Solve XOR using an MLP with 4 hidden units
- Universal approximation theorem - Deep Learning e-book

# Feedforward Networks

- A feedforward neural network, also called a multi-layer perceptron, is a collection of neurons, organized in layers.

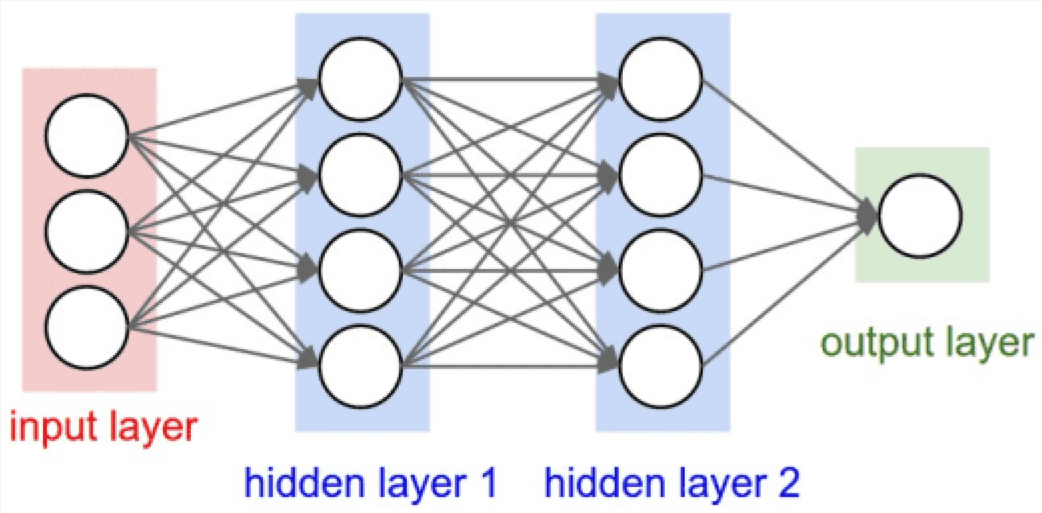



- It is used to approximate some function $\boldsymbol{f}^*$. For instance, $\boldsymbol{f}^*$ could be a classifier that maps an input vector $\boldsymbol{x}$ to a category $\boldsymbol{y}$.

- The neurons are arranged in the form of a directed acyclic graph i.e., the information only flows in one direction - input $\boldsymbol{x}$ to output $\boldsymbol{y}$. Hence the term feedforward.

# Feedforward Networks

- The number of layers in the network (excluding the input layer) is known as depth




- Each neuron can be seen as a vector-to-scalar function which takes a vector of inputs from the previous layer and computes a scalar value.

- Above network can be seen as a composition of functions $y = f^{(3)}\left(f^{(2)}\left(f^{(1)}(x)\right)\right), f^{(1)}$ being the first hidden layer, $f^{(2)}$ being the second and $f^{(3)}$ being the final output layer.

# Feedforward Networks

- To approximate some function $f^*$, we are generally given noisy estimates of $f^*(\boldsymbol{x})$ at different points, in the form of a dataset $\{\boldsymbol{x}_i, y_i\}_{i=1}^{M}$
- Our neural network defines a function $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$. Our goal is to learn the parameters (weights and biases) $\boldsymbol{\theta}$ such that $f$ best approximates $f^*$.
- How to find the values of the parameters i.e., train the network?
- In this lecture, we introduce Gradient Descent, the go-to method to train neural networks

# Gradient Descent: 1D example

- Neural networks are usually trained by minimizing a loss function, such as mean square error:

$$\mathrm{Loss}_{MSE} = \frac{1}{M}\sum_{i=1}^{M}\left(f^*(\boldsymbol{x}) - f(\boldsymbol{x};\boldsymbol{\theta})\right)^2$$

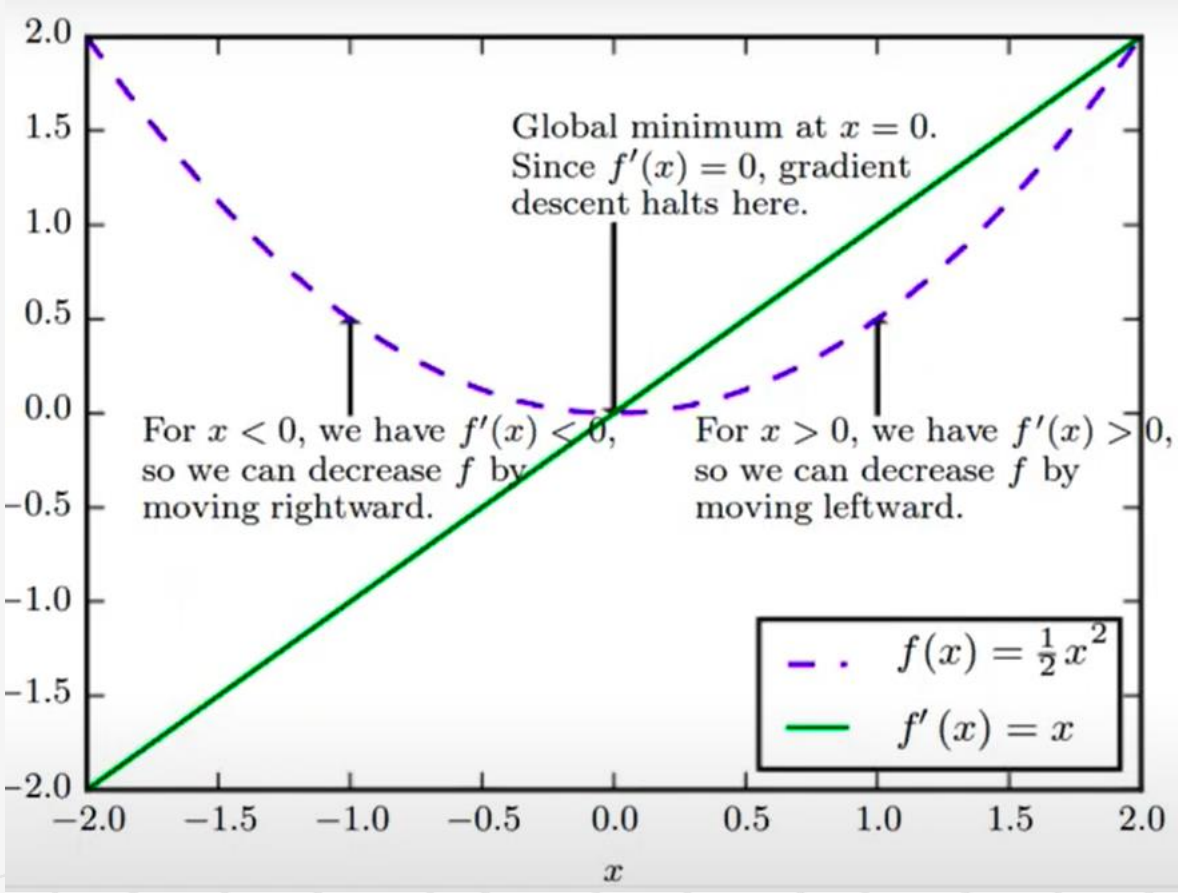
- Let us consider a simple **1D** example, where we try to minimize the function $f(x) = x^2$ Specifically, we find out the value $\boldsymbol{x}^*$ gives the smallest value for $f(x)$ i.e., $f(x^*)$

- $x^* = \arg\min_x f(x)$

# Gradient Descent: 1D example

- We can obtain the slope of the function $f(x)$ at $x$ by taking its derivative i.e., $f'(x)$
- This means, if we give a very small push to $x$ in the direction (sign) of the slope, we're sure that the function will increase.
  - $f\big(x + p \cdot \mathrm{sign}(f'(x))\big) > f(x)$ for an infinitesimally small $p$
- The reverse is also true i.e.,
  - $f\big(x - p \cdot \mathrm{sign}(f'(x))\big) < f(x)$ for an infinitesimally small $p$
- This forms the basis for gradient descent - we start off at a random $x$, and take small steps in the direction of the **negative** gradient.

# Gradient Descent: 1D example



# Why Negative Gradient

- Consider the multivariate case, since while training neural networks, the loss function we minimize is parametrized by multiple weights, $\boldsymbol{\theta}$

- For simplicity, we denote our loss function as $\boldsymbol{L(\theta)}$**.** Our aim is to find the weight vector $\boldsymbol{\theta}$ which minimizes $\boldsymbol{L(\theta)}$

- Let $\mathbf{u}$, a unit vector, be the direction that takes us to the minimum, i.e.:

$$\min_{\mathbf{u},\mathbf{u}^T\mathbf{u}=1} \mathbf{u}^T \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$$

$$= \min_{\mathbf{u},\mathbf{u}^T\mathbf{u}=1} \| \mathbf{u} \|_2 \, \|\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})\|_2 \cos\beta$$

- Since $\| \boldsymbol{u} \|_2 = 1$, we can minimize the above function when $\beta = 180^0$, i.e. when $\mathbf{u}$ is the direction of **negative** gradient

# How to Use Gradient Descent

- We can use Gradient Descent to train neural networks as follows:

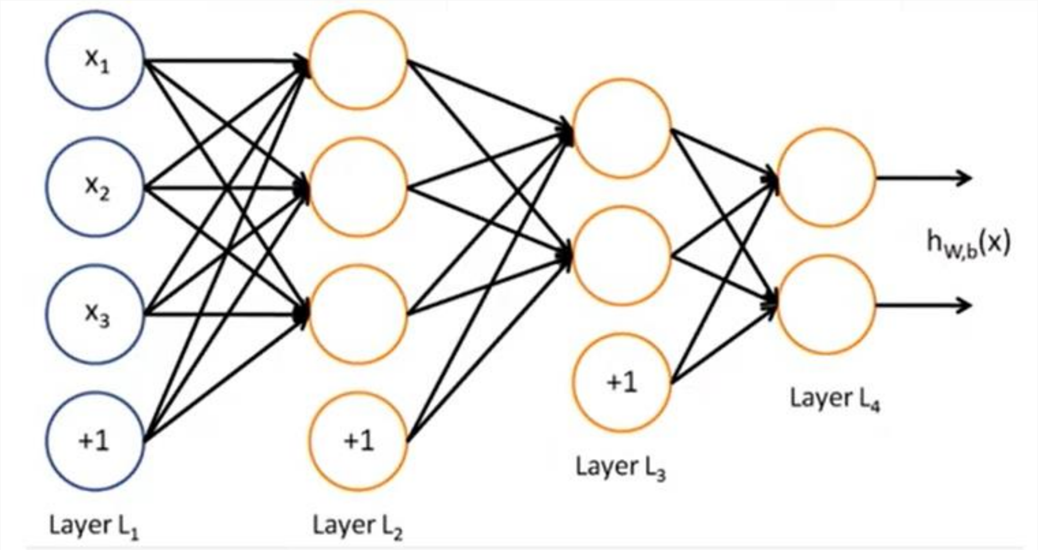
- Start with a random weight vector $\boldsymbol{\theta}$.
- Compute the loss function over the dataset, i.e., $L(\theta)$ with the current network, using a suitable loss function such as mean-squared error
- Compute the gradients of the loss function with respect to each weight value $\frac{\delta L}{\delta \theta_i}$.
- Update the weights as follows, where $\boldsymbol{\eta}$ is the learning rate i.e., the amount by which the weight is changed in each step:

$$\boldsymbol{\theta}_i^{\text{next}} = \boldsymbol{\theta}_i^{\text{curr}} - \eta \frac{\delta L}{\delta \boldsymbol{\theta}_i^{\text{curr}}}$$

- We can repeat the above steps until the gradient is zero.

# Gradient Descent

- A feedforward neural network is a composition of multiple functions, organized as layers



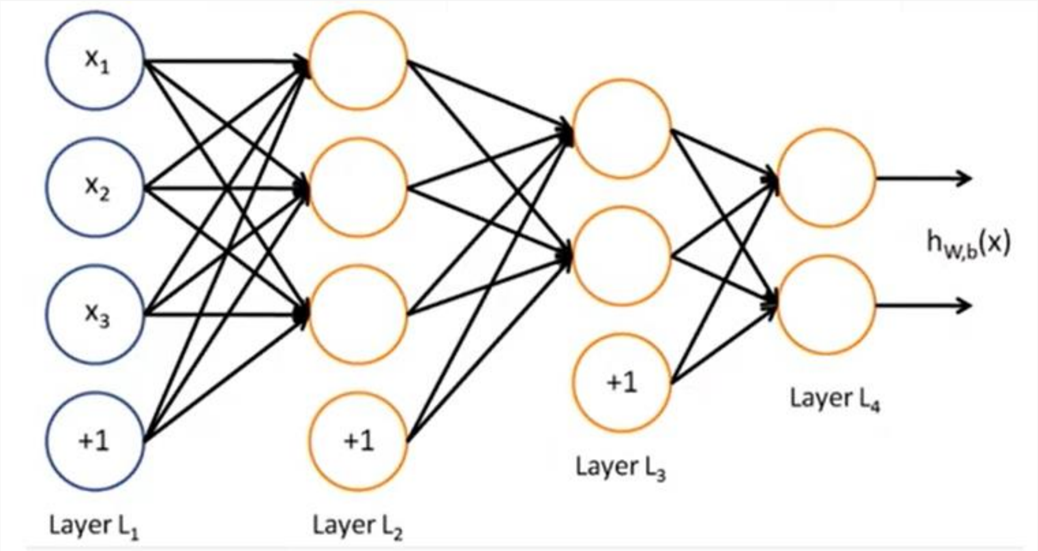

- What do we need to implement gradient descent? Compute gradient of loss function w.r.t. each weight in the network. How to do this?
- Using the chain rule in calculus
  - Computing gradient w.r.to a weight in layer $i$ requires computation of gradients with respect to outputs which involve that weight i.e., all activations from layer $i+1$ to last layer, $n_l$

# Backpropagation

- In the next few slides, we introduce **backpropagation**, a procedure which combines gradient computation using chain rule and parameter updation using Gradient Descent, thus fully describing the neural network training algorithm.

# Backpropagation

- Consider a simple FFNN (or MLP)



# Backpropagation

- A fixed training set $\{x^{(i)}, y^{(i)}\}_{i=1}^{M}$ of $M$ training samples
- Parameters $\theta = \{W, b\}$, weights and biases
- Mean square cost function for a single example:

$$L(\theta; x, y) = \frac{1}{2} \|h_\theta(x) - y\|^2$$

- Overall cost function is given by:

$$\begin{aligned} L(\theta) \quad &= \frac{1}{M} \sum_{i=1}^{M} L\left(\theta; x^{(i)}, y^{(i)}\right) \\ &= \frac{1}{2M} \sum_{i=1}^{M} \| h_\theta\left(x^{(i)}\right) - y^{(i)} \|^2 \end{aligned}$$

# Backpropagation: Notations

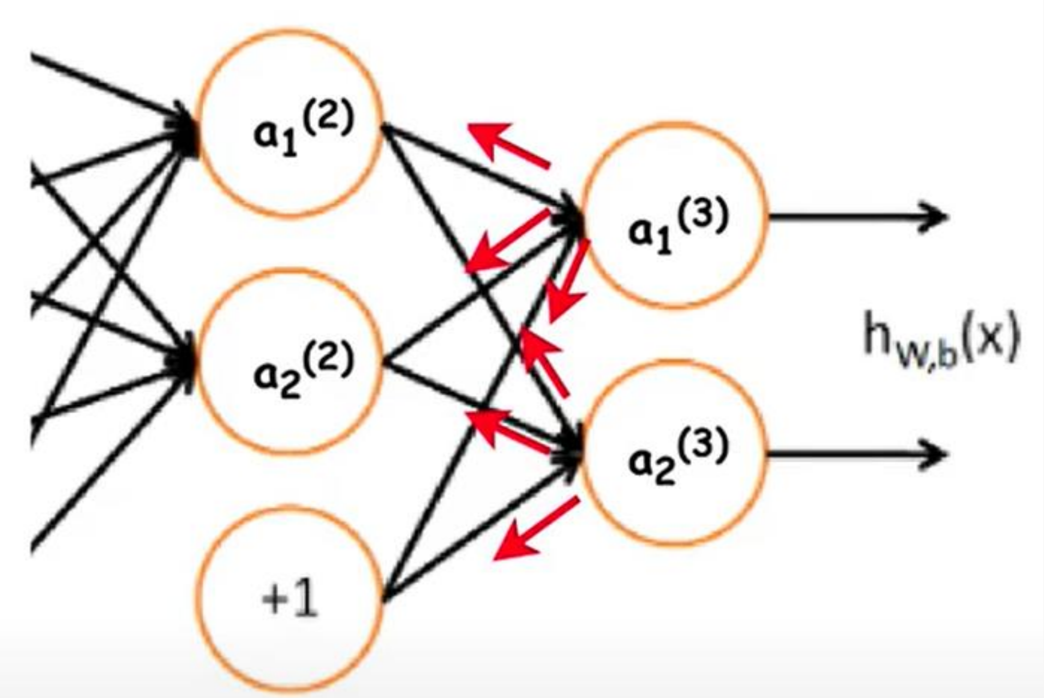
- We have $n_l$ layers in the network, $l = 1,2,..,n_l$
- We denote activation of node $i$ at layer $l$ as $a_i^{(l)}$
- We denote weight connecting node $i$ in layer $l$ and node $j$ in layer $l+1$ as $W_{ij}^{(l)}$. The weight matrix between layer $l$ and layer $l+1$ is denoted as $W^{(l)}$
- For a 3-layer network shown earlier, compact vectorized form of a forward pass to compute neural network's output is shown below:

$$
\begin{aligned}
z^{(2)} &= W^{(1)}x + b^{(1)} \\
a^{(2)} &= f\left(z^{(2)}\right) \\
z^{(3)} &= W^{(2)}a^{(2)} + b^{(2)} \\
h(x) &= \underline{a^{(3)}} = \underline{f}\left(z^{(3)}\right)
\end{aligned}
$$

- Function $f$ can denote any activation function such as sigmoid, ReLU, identity, etc.

# Backpropagation: Backward Pass

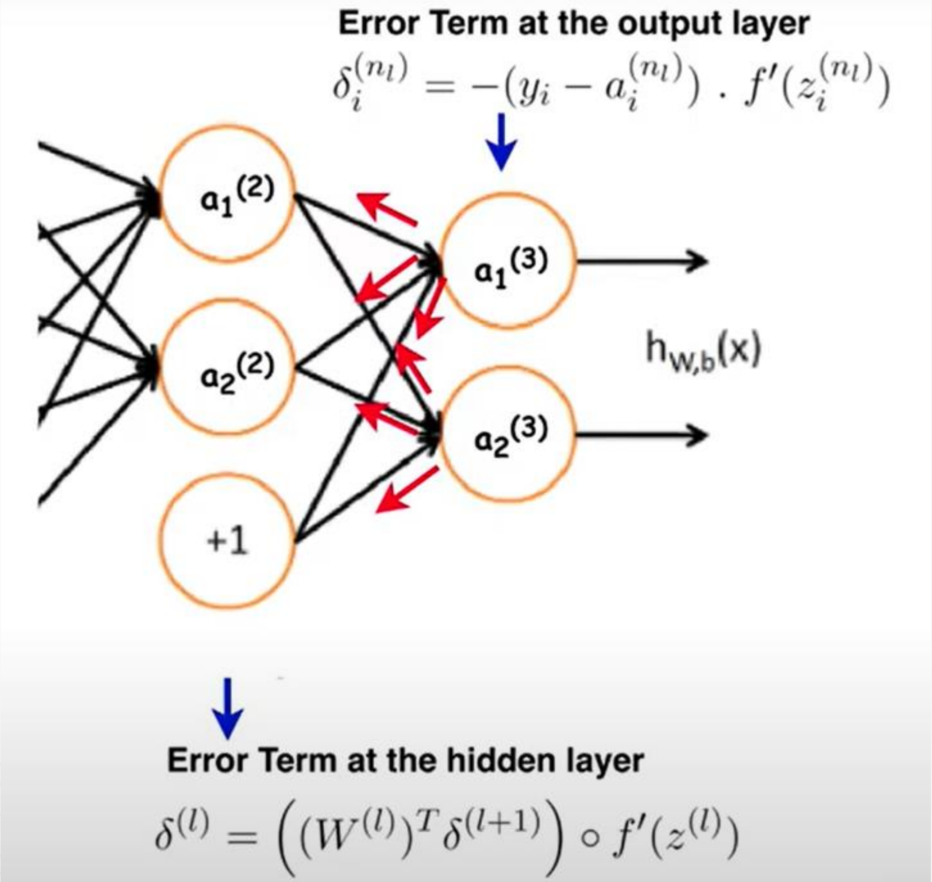
- During the forward pass, we successively compute each layer's outputs from left to right.
- During backward pass, we aim to compute derivatives of each parameter starting from the right most layer to the left most one i.e., layer $n_l, n_l - 1, \ldots, 1$.
- Once the derivatives are computed, we use Gradient Descent to update the parameters,



# Backpropagation: Backward Pass

- For each node, we define an error term $\delta_i^{(l)}$ to denote how much the node was responsible for the loss computed
- If $l = n_l$ i.e., last layer, error term computation is straightforward, since we directly take derivative of loss function (MSE, in this case, between output and target values)

$$\delta_i^{(n_l)} = -\left(y_i - a_i^{(n_l)}\right) \cdot f'\left(z_i^{(n_l)}\right)$$

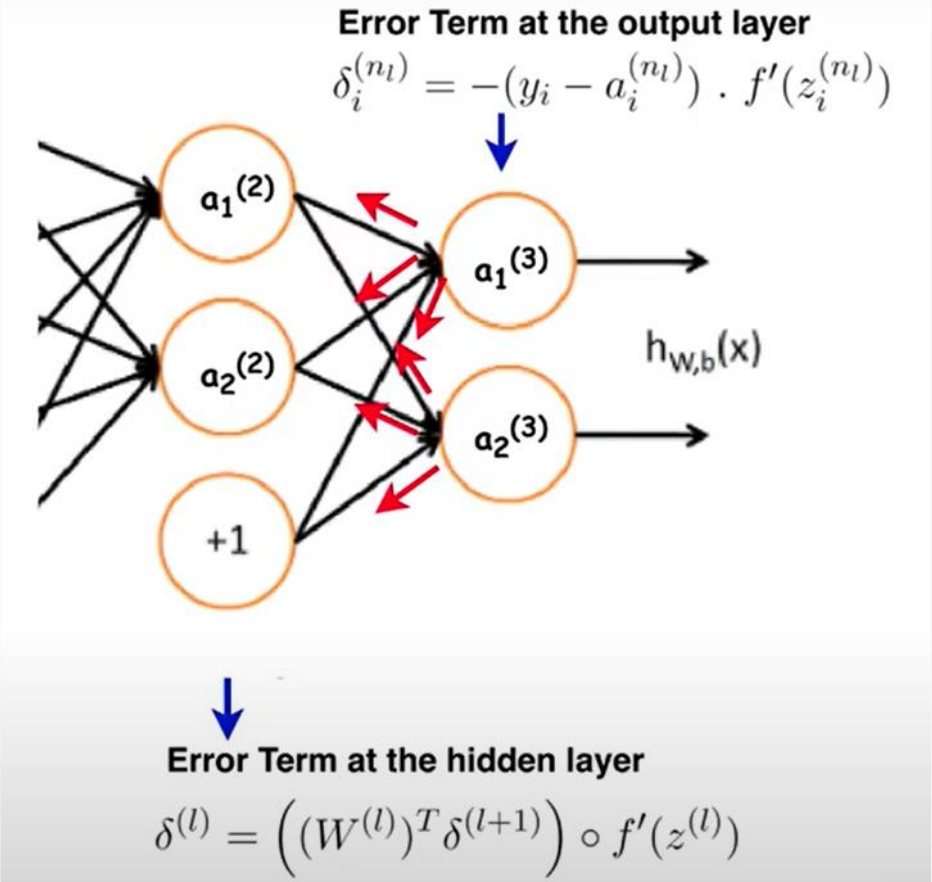


# Backpropagation: Backward Pass

- To compute error term **for hidden layers**, $l = n_l - 1, n_l - 2, \ldots$, we rely on error terms from subsequent layers

- In particular, we compute error term as sum of error terms in next layer, weighted by weights along connections to next layer:

$$\delta_i^{(l)} = \left( \sum_{j=1}^{s_{l+1}} W_{ij}^{(l)} \delta_j^{(l+1)} \right) f'\left(z_i^{(l)}\right)$$



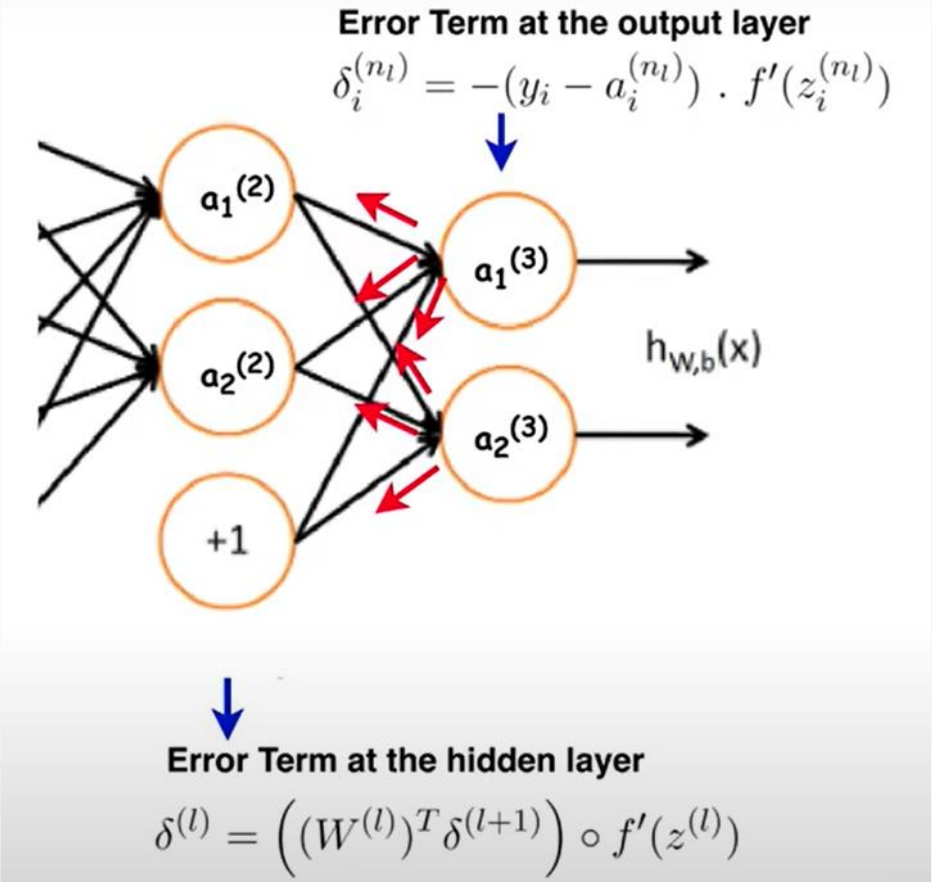
# Backpropagation: Backward Pass

- Perform a feedforward pass, computing the activations for layers $1, 2, \ldots n_l$.
- For each output unit $i$ in layer $n_l$ set,

$$\delta^{(n_l)} = -\left(y - a^{(n_l)}\right) \circ f'\left(z^{(n_l)}\right)$$

- For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$ For each node in layer $l$ set,

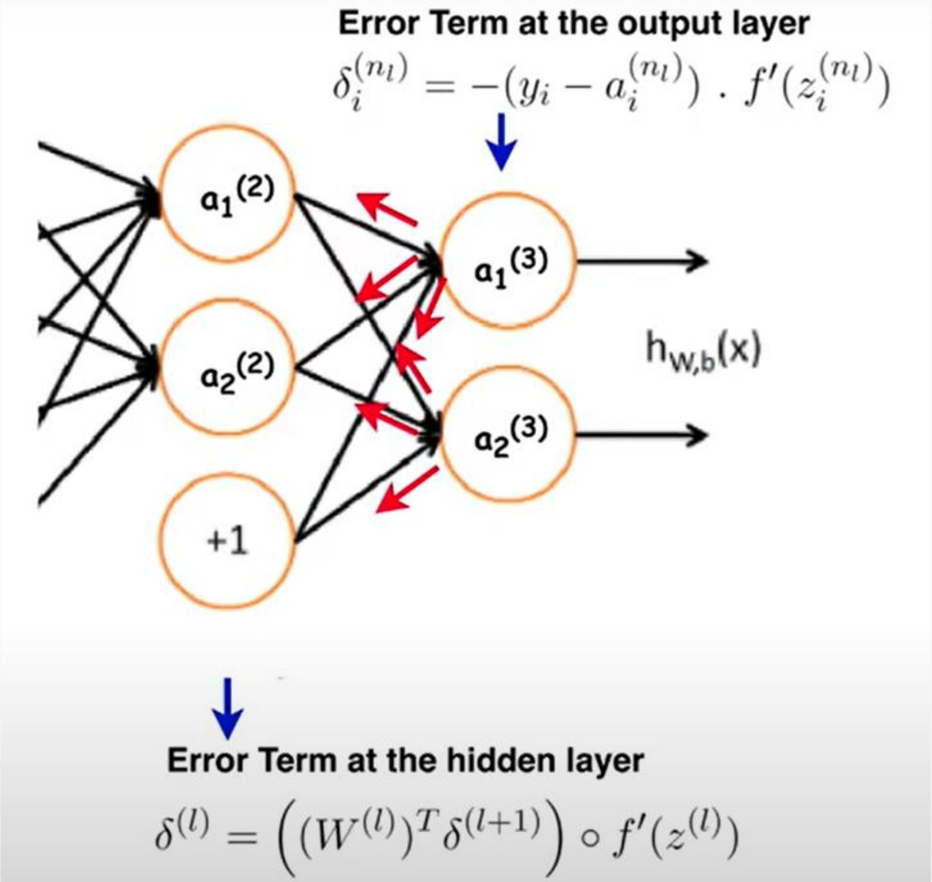
$$\delta^{(l)} = \left(\left(W^{(l)}\right)^T \delta^{(l+1)}\right) \circ f'\left(z^{(l)}\right)$$



# Backpropagation: Backward Pass

- Compute the desired partial derivatives, as:

$$\nabla_{W^{(l)}} L(W, b; x, y) = \delta^{l+1} \left(a^{(l)}\right)^T$$
$$\nabla_{b^{(l)}} L(W, b; x, y) = \delta^{l+1}$$



# Gradient Descent Using Backpropagation

- Set $\Delta W^{(l)} := \mathbf{0}, \Delta b^{(l)} = \mathbf{0}$ (matrix/vector of zeros) for all $l$.
- For $i = 1$ to $M$
    - Use backpropagation to compute $\nabla_{\theta^{(l)}} L(\theta; x, y)$.
    - Set $\Delta\theta^{(l)} := \Delta\theta^{(l)} + \nabla_{\theta^{(l)}} L(\theta; x, y)$
- Update the parameters:

$$
\begin{aligned}
W^{(l)} &= W^{(l)} - \eta\left[\frac{1}{M}\Delta W^{(l)}\right] \\
b^{(l)} &= b^{(l)} - \eta\left[\frac{1}{M}\Delta b^{(l)}\right]
\end{aligned}
$$

- Repeat for all points until convergence.

# Questions?

- Sources for this lecture include materials from works by Mitesh K, Vineeth N B