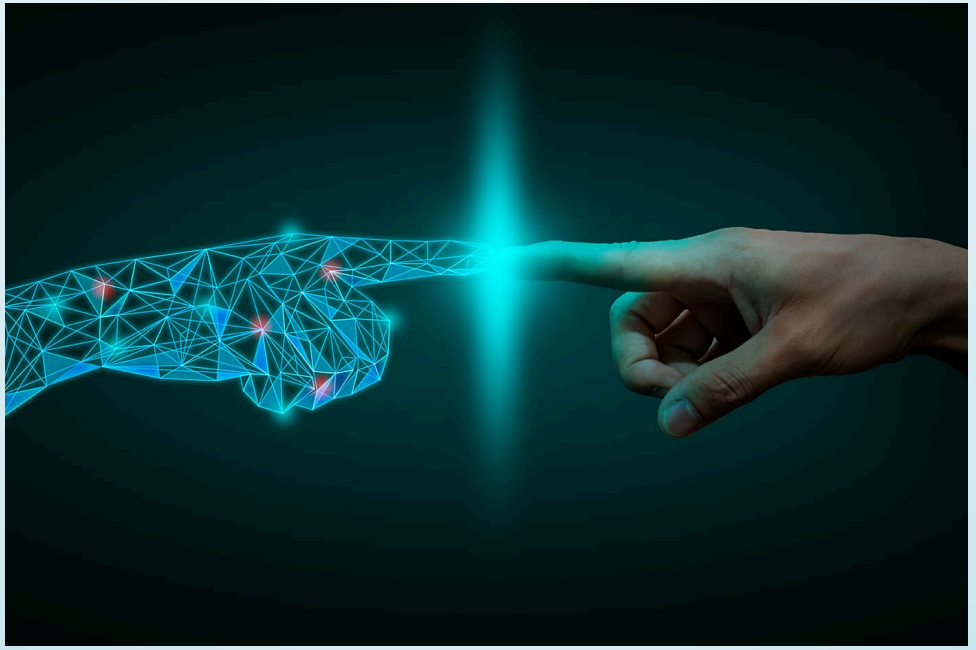
ARTIFICIAL NEURAL NETWORKS

Neural Networks, Deep Learning and Artificial Intelligence

FROM ARTIFICIAL NEURAL NETWORKS TO ARTIFIAL INTELLIGENCE

HISTORICAL BACKGROUND (1)

- In the post-pandemic world, a lightning rise of AI, with a mess of realities and promises is impacting society.
- Since ChatGPT entered the scene everybody has an experience, an opinion, or a fear on the topic.



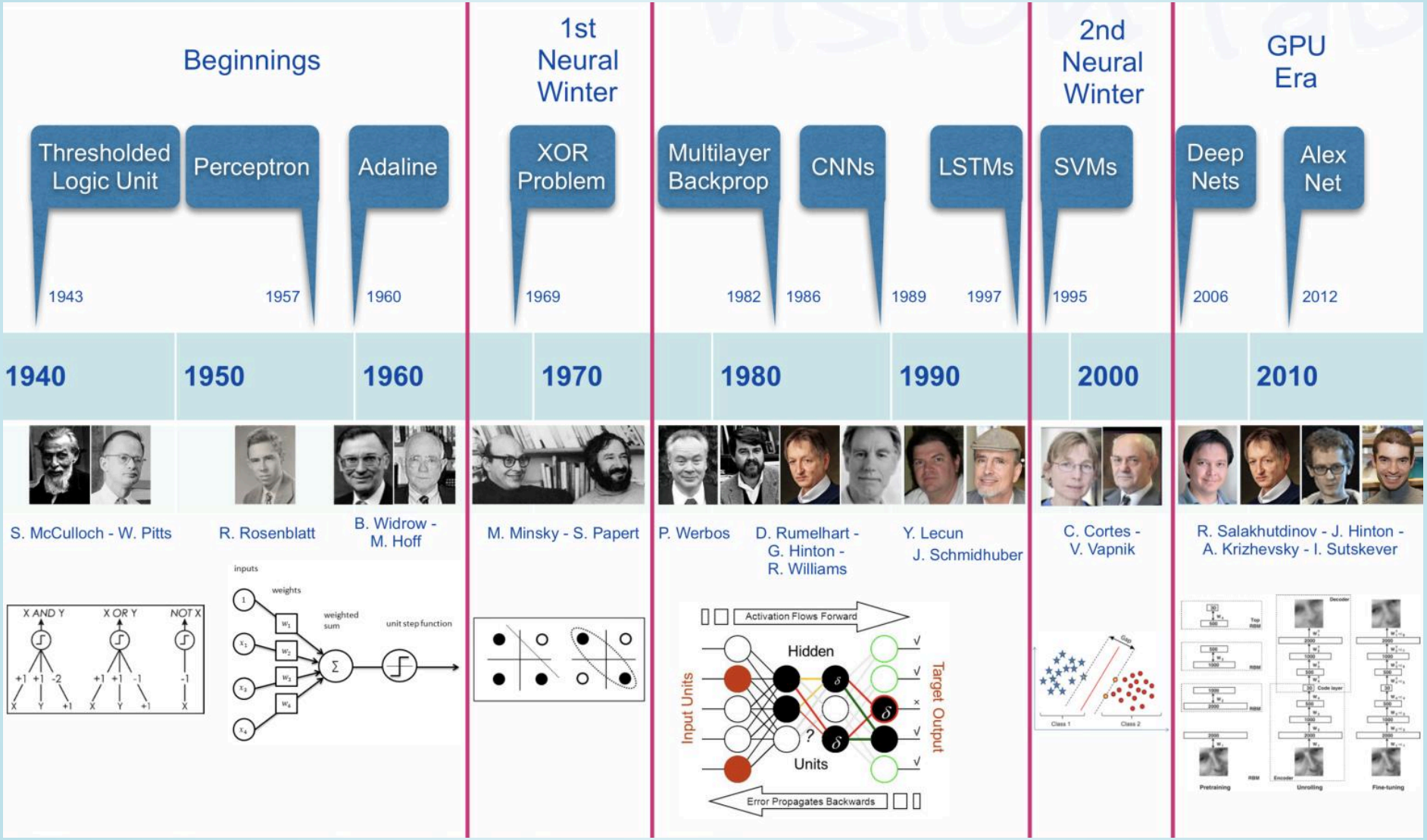
IS IT JUST MACHINE LEARNING?

- Most tasks performed by AI can be described as Classification or Prediction used in applications as:
  - Recommendation systems,
  - Image recognition, Image generation
  - Natural language processing
- AI relies on machine learning algorithms, to make predictions based on large amounts of data.
- AI has far-reaching implications beyond its predictive capabilities, including ethical, social or technological.

AI, ANNS AND DEEP LEARNING

- In many contexts, talking about AI means talking about *Deep Learning (DL)*.
- DL is a successful AI model which has powered many application such as *self-driving cars, voice assistants, and medical diagnosis systems*.
- DL originates in the field of *Artificial Neural Networks*
- But DL extends the basic principles of ANNs by:
  - Adding complex architectures and algorithms and
  - At the same time becoming more automatic

THE EARLY HISTORY OF AI (1)



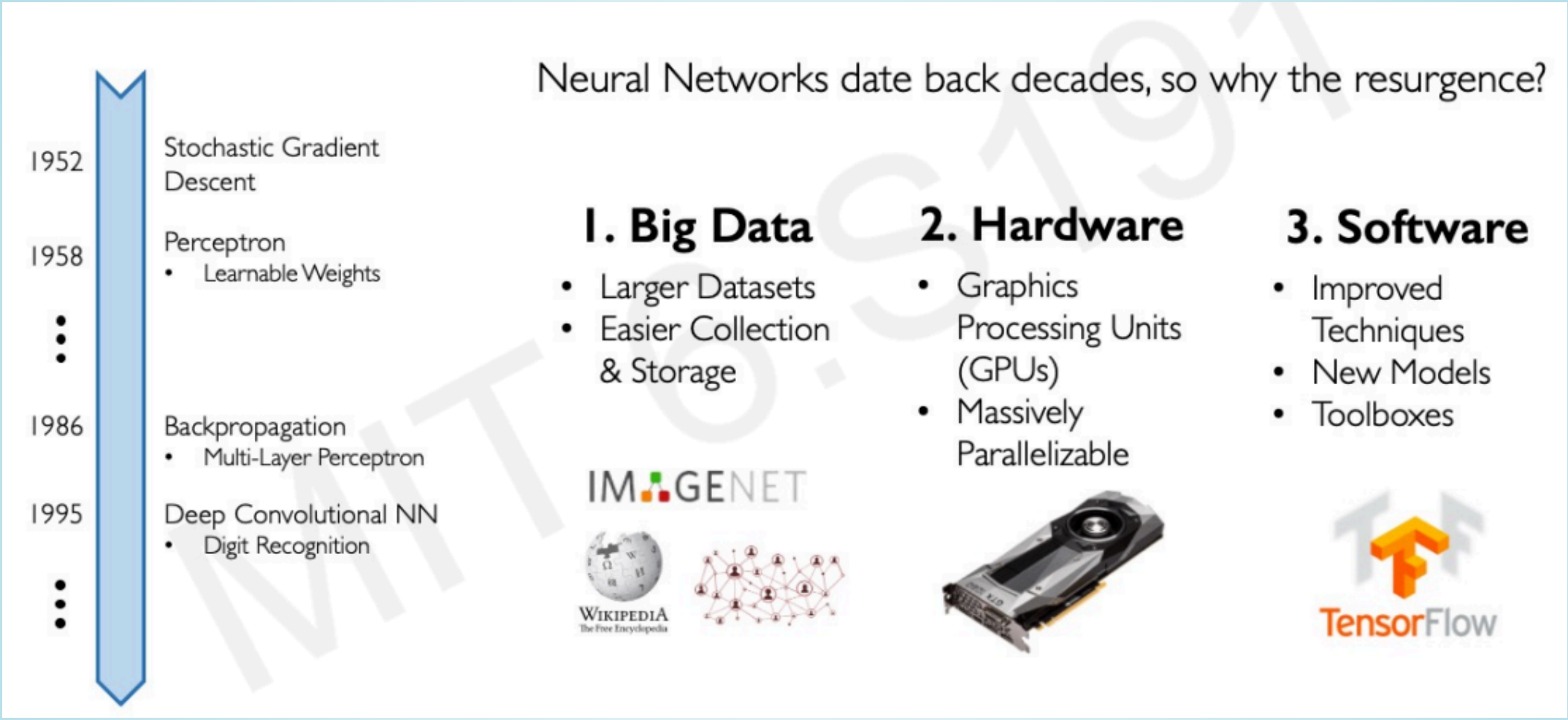
A Quick History of AI, ML and DL

MILESTONES IN THE HISTORY OF DL

We can see several hints worth to account for:

- The **Perceptron** and the first **Artificial Neural Network** where the basic building block was introduced.
- The **Multilayered perceptron** and back-propagation where complex architectures were suggested to improve the capabilities.
- **Deep Neural Networks**, with many hidden layers, and auto-tunability capabilities.

FROM ANN TO DEEP LEARNING



Why Deep Learning Now?

Source: Alex Amini's MIT Introduction to Deep Learning' course

SUCCESS STORIES

Success stories such as

- the development of self-driving cars,
- the use of AI in medical diagnosis, and
- online shopping personalized recommendations

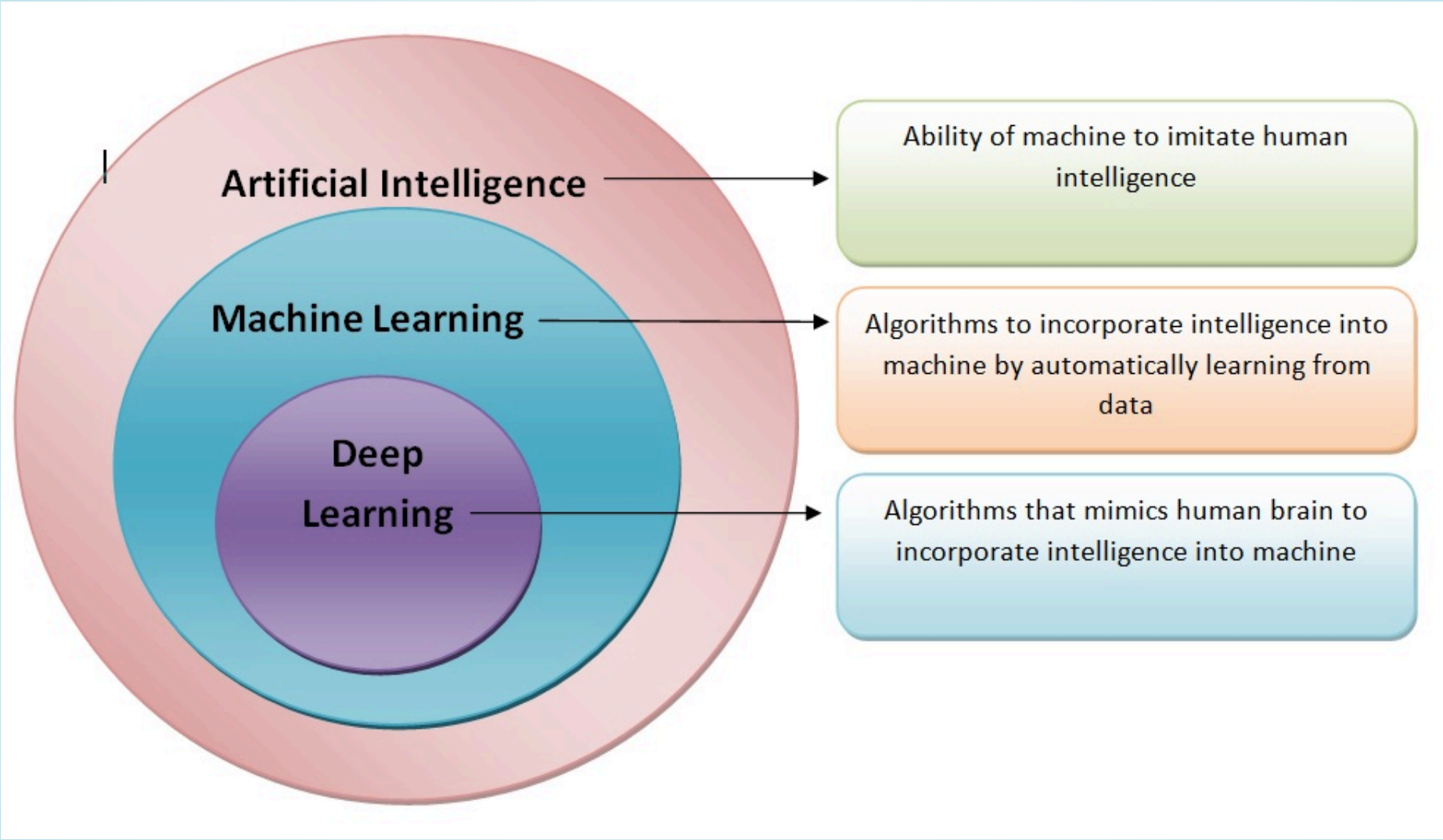
have also contributed to the widespread adoption of AI.

NOT TO TALK ABOU THE FEARS

- AI also comes with fears from multiple sources from science fiction to religion
  - Mass unemployment
  - Loss of privacity
  - AI bias
  - AI fakes
  - Or, simply, AI takeover

BACK TO SCIENCE

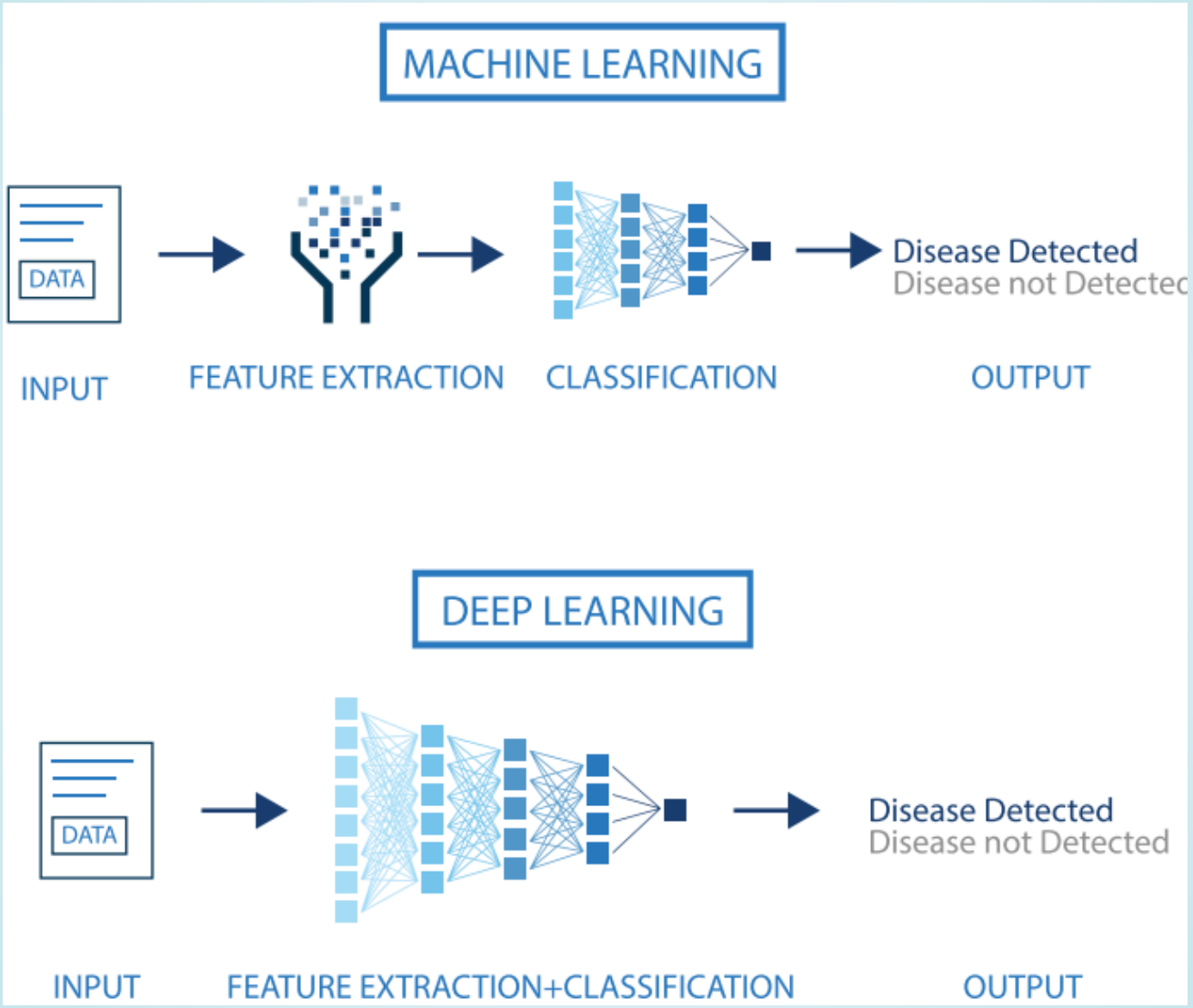
Where/How does it all fit?



AI, ML, DL ...

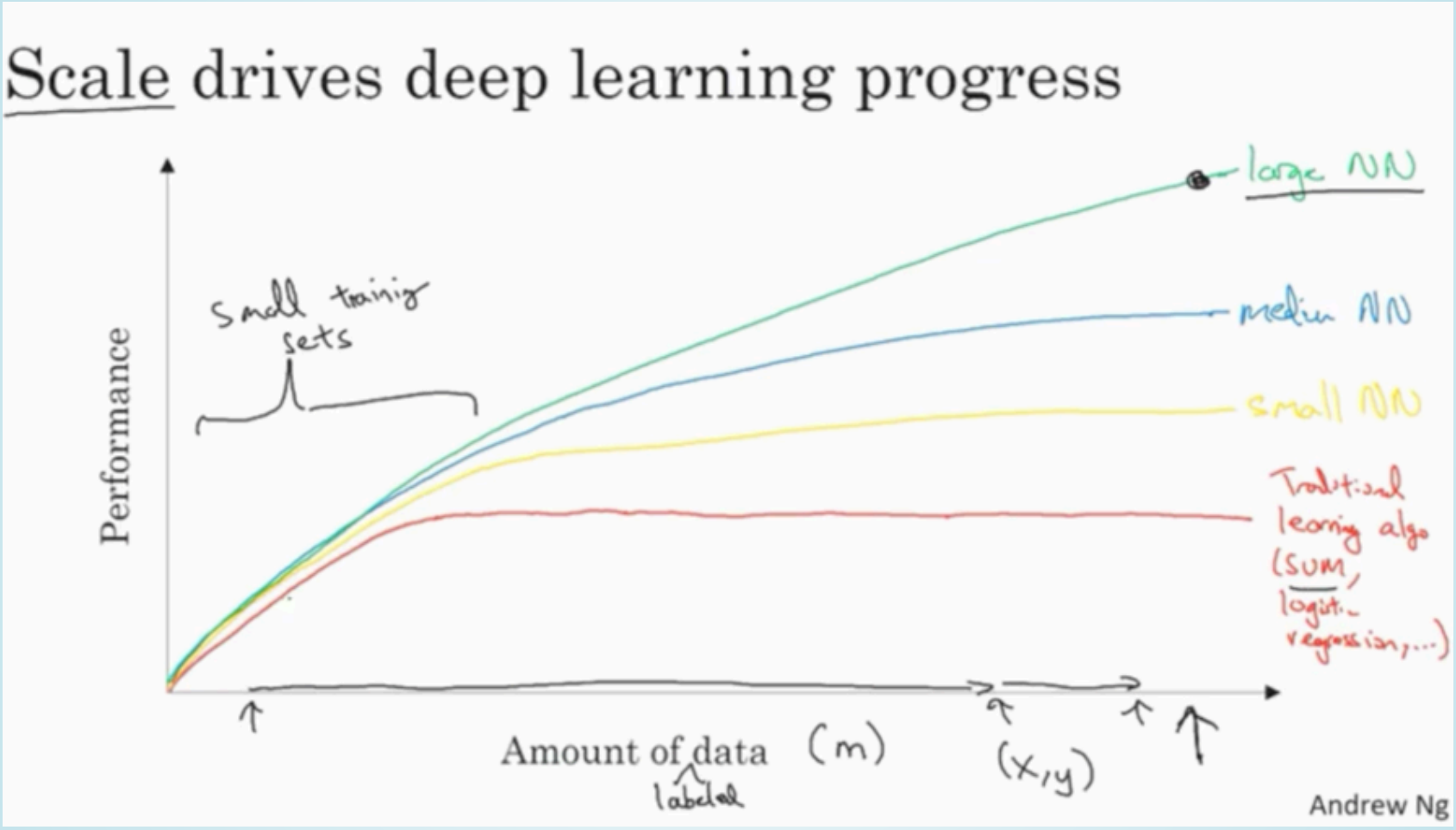
- **Artificial intelligence**: Ability of a computer to perform tasks commonly associated with intelligent beings.
- **Machine learning**: study of algorithms that learn from examples and experience instead of relying on hard-coded rules and make predictions on new data
- **Deep learning**: sub field of ML focusing on learning data representations as successive successive layers of increasingly meaningful representations.

HOW DOES DL IMPROVE ON ML



ML and DL Approaches for Brain Disease Diagnosis

SIZE DOES MATTER!



An illustration of the performance comparison between deep learning (DL) and other machine learning (ML) algorithms, where DL modeling from large amounts of data can increase the performance

THE IMPACT OF DEEP LEARNING

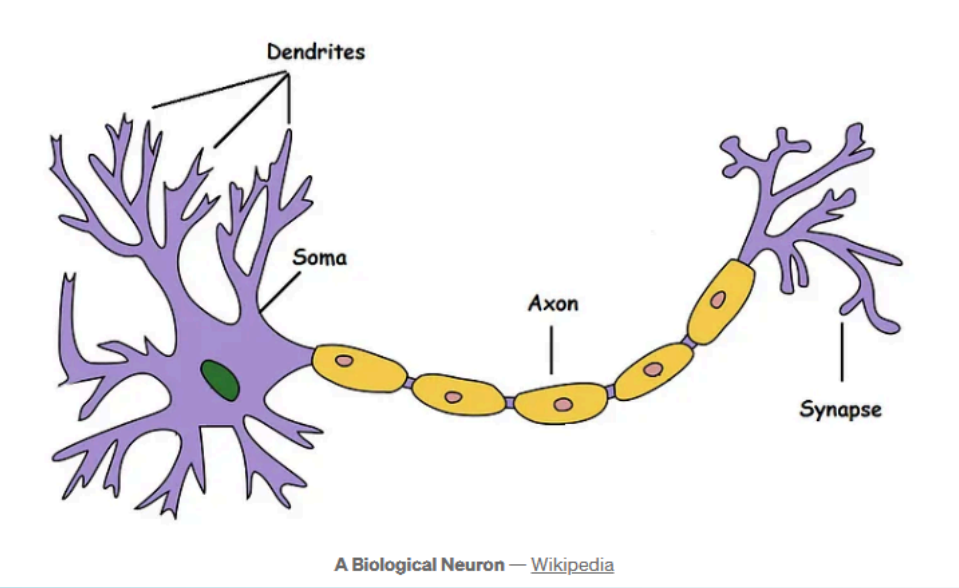
- Near-human-level image classification
- Near-human-level speech transcription
- Near-human-level handwriting transcription
- Dramatically improved machine translation
- Dramatically improved text-to-speech conversion
- Digital assistants such as Google Assistant and Amazon Alexa
- Near-human-level autonomous driving
- Improved ad targeting, as used by Google, Baidu, or Bing
- Improved search results on the web
- Ability to answer natural language questions
- Superhuman Go playing

NOT ALL THAT GLITTERS IS GOLD …

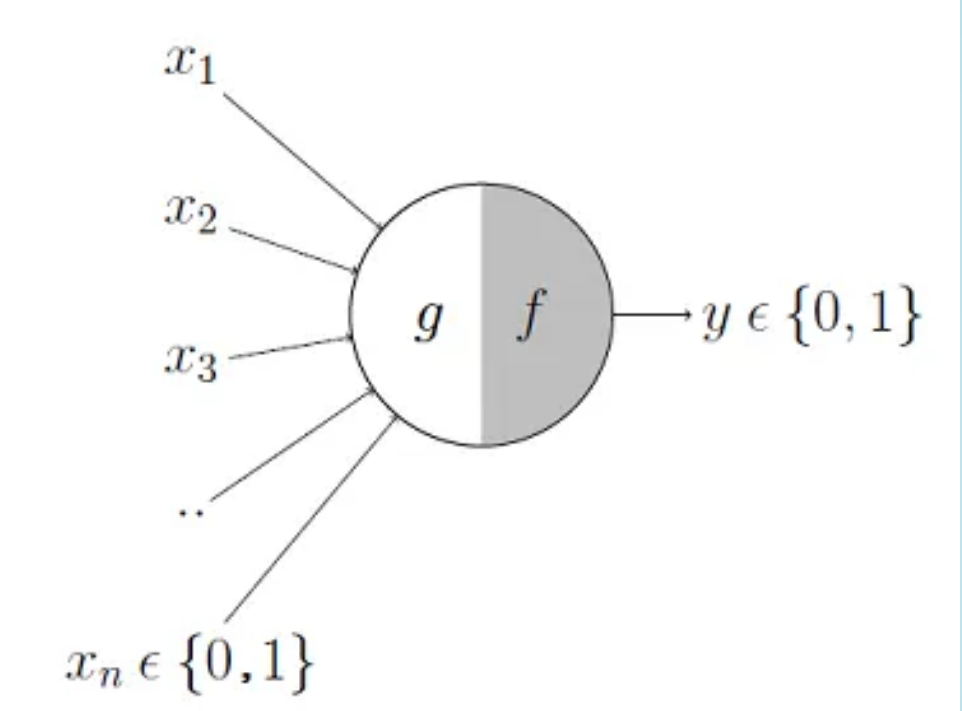
- According to F. Chollet, the developer of Keras,
  - *"we shouldn't believe the short-term hype, but should believe in the long-term vision.*
  - *It may take a while for AI to be deployed to its true potential—a potential the full extent of which no one has yet dared to dream*
  - *but AI is coming, and it will transform our world in a fantastic way*".

THE ARTIFICIAL NEURONE

EMULATING BIOLOGICAL NEURONS



A biological Neuron



MuCulloch & Pitts proposal

- The first model of an artifial neurone was proposed by Mc Cullough & Pitts in 1943

MC CULLOUGH'S NEURON

- It may be divided into 2 parts.
  - The first part, ,takes an input (as the dendrites of a neuron would do),
  - It performs an aggregation and
  - based on the aggregated value the second part, , f makes a decision.

- See this link for an explanation on how it can emulate logical operations such as AND, OR or NOT, but not XOR.
- See this link for a nice clarifying example

LIMITATIONS

This first attempt to emulate neurons succeeded but with limitations:

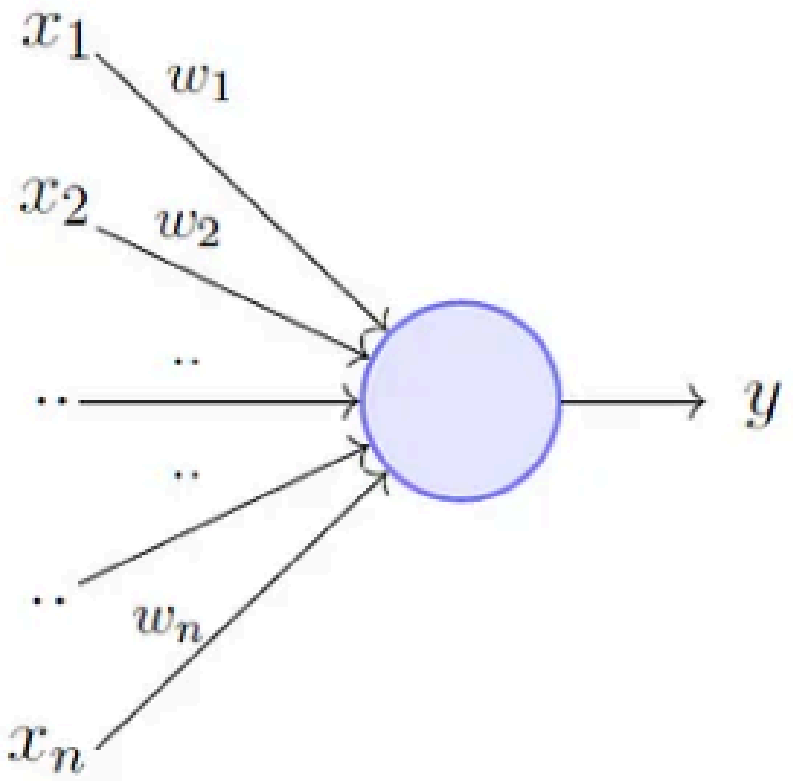
- What about non-Boolean (say, real) inputs?
- What if all inputs are not equal?
- What if we want to assign more importance to some inputs?
- What about functions which are not linearly separable? Say XOR function

OVERCOMING THE LIMITATIONS

- To overcome these limitations Rosenblatt, proposed the perceptron model, or *artificial neuron*, in 1958.
- Generalizes McCullough-Pitts neuron in that *weights and thresholds can be learnt over time*.
  - It takes a weighted sum of the inputs and
  - It sets the output to iff the sum is more than an arbitrary threshold ().

$$\theta$$

ROSENBLATT'S PERCEPTRON



$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i \geq \theta$$

$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i < \theta$$

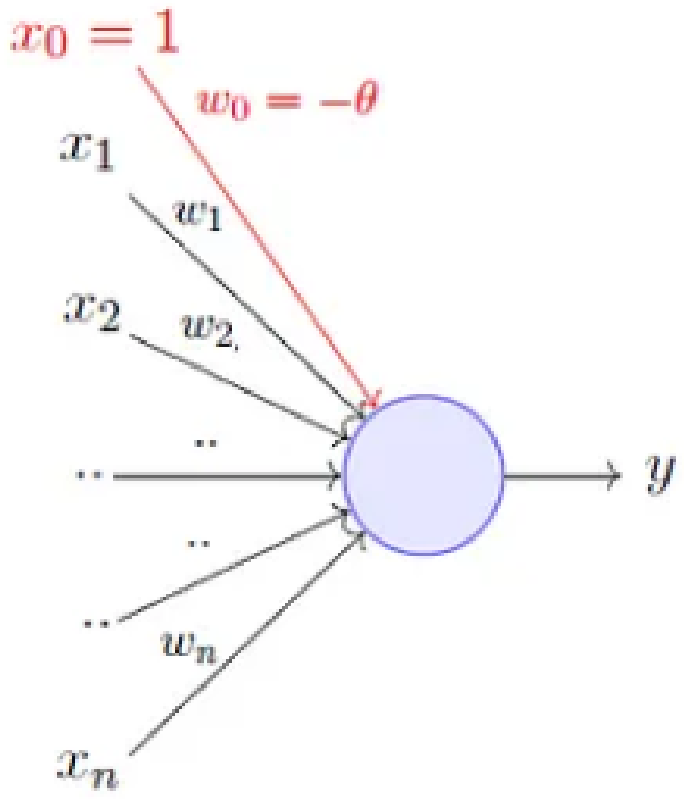
Rewriting the above,

$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i - \theta \geq 0$$

$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i - \theta < 0$$

ROSENBLATT'S PERCEPTRON

- Instead of hand coding the thresholding parameter , θ
- It is added as one of the inputs, with the weight . $w_0 = -\theta$



A more accepted convention,

$$y = 1 \quad if \sum_{i=0}^{n} w_i * x_i \geq 0$$

$$= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0$$

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

COMPARISON BETWEEN THE TWO

McCulloch Pitts Neuron
(assuming no inhibitory inputs)

$$y = 1 \quad if \sum_{i=0}^{n} x_i \geq 0$$

$$= 0 \quad if \sum_{i=0}^{n} x_i < 0$$

Perceptron

$$y = 1 \quad if \sum_{i=0}^{n} w_i * x_i \geq 0$$

$$= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0$$

COMPARISON BETWEEN THE TWO

- This is an improvement because
  - both, weights and threshold, can be learned and
  - the inputs can be real values
- But there is still a drawback in that a single perceptron can only be used to implement linearly separable functions.
- Artificial Neural Networks improve on this by introducing *Activation Functions*

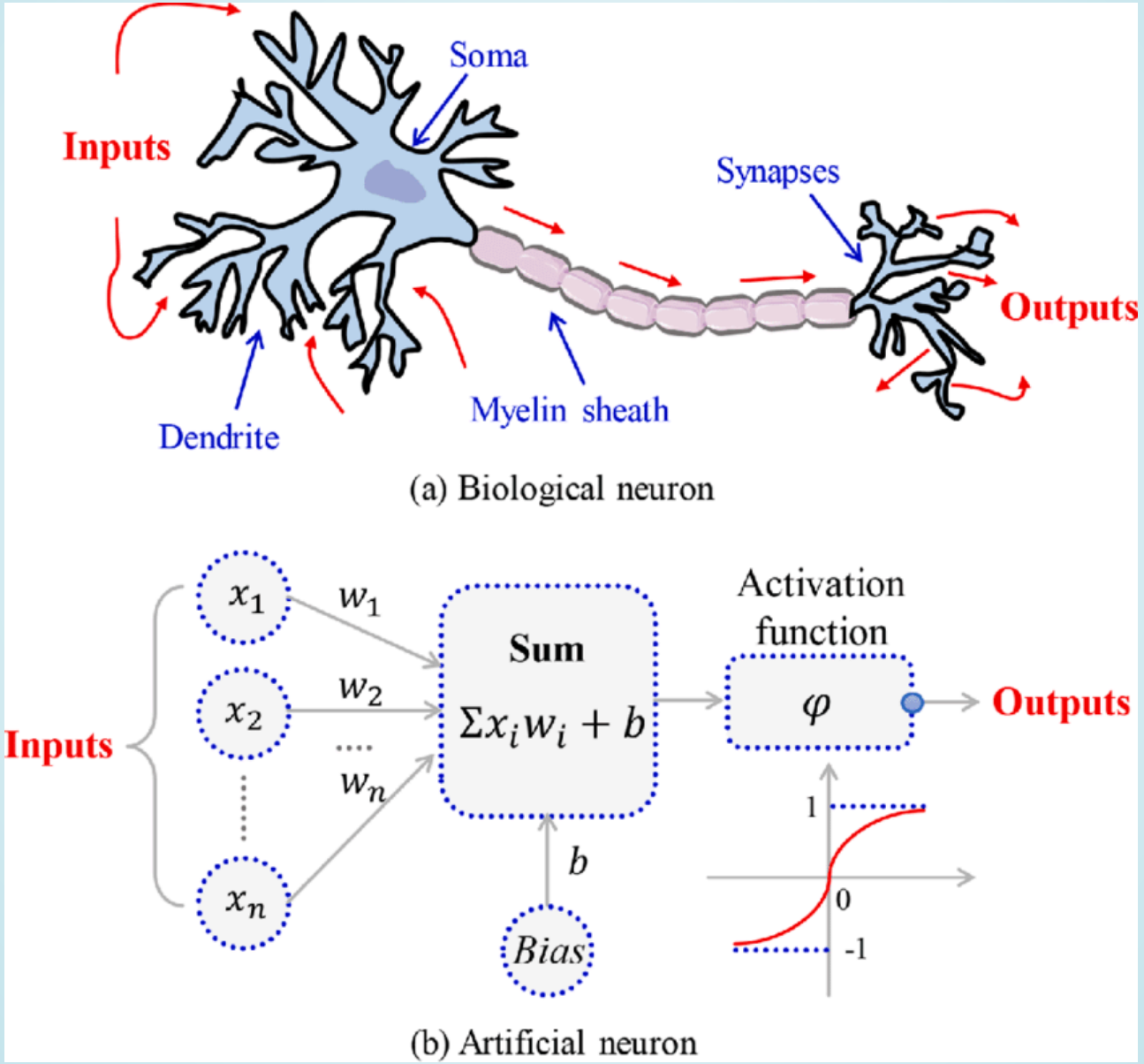
ACTIVATION IN BIOLOGICAL NEURONS

- Biological neurons are specialized cells that transmit signals to communicate with each other.
- Neuron's activation is based on releasing *neurotransmitters*, chemicals that transmit signals between nerve cells.
  - When the signal reaching the neuron exceeds a certain threshold, it releases neurotransmitters to continue the communication process.

ACTIVATION FUNCTIONS IN AN

- Analogously, *activation functions* in AN are functions to decide if the AN it is activated or not.
- AN's activation function is a mathematical function applied to the neuron's input to produce an output.
  - In practice it extends to complicated functions that can learn complex patterns in the data.
  - Activation functions can incorporate non-linearity, improving over linear classifiers.

ACTIVATION FUNCTION



ARTIFICIAL NEURON

With all these ideas in mind we can now define an Artificial Neuron as a *computational unit* that :

- takes as input ,
- outputs ,
- where is called the **activation function**.

$$x = (x_0, x_1, x_2, x_3), \ (x_0 = +1 \equiv bi$$

$$h_\theta(x) = f(\theta^\mathsf{T} x) = f(\sum_i \theta_i x_i)$$

$$f : \mathbb{R} \mapsto \mathbb{R}$$

ACTIVATION FUNCTIONS

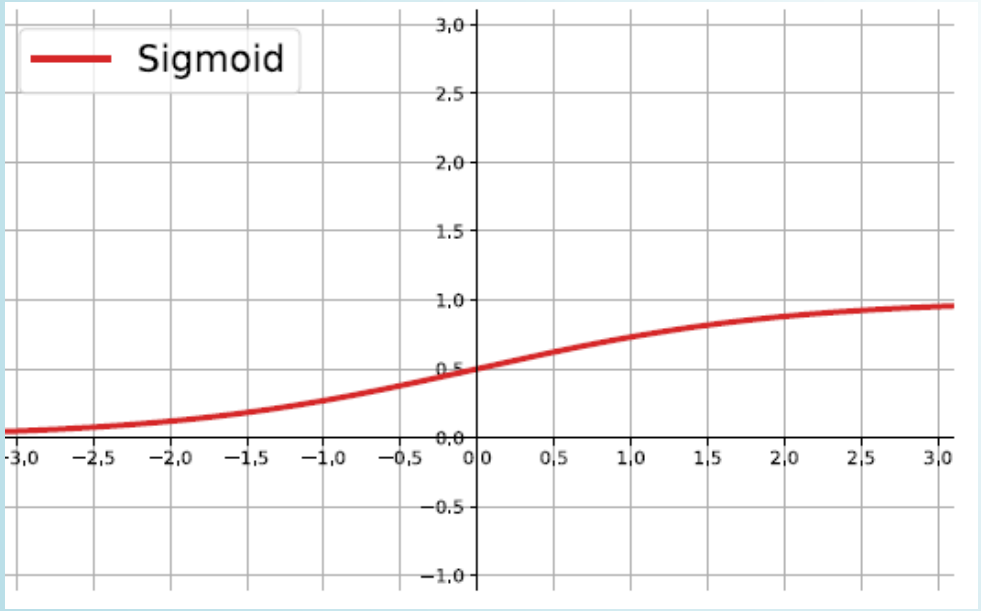
- Goal of activation function is to provide the neuron with *the capability of producing the required outputs*.
- Flexible enough to produce
  - Either linear or non-linear transformations.
  - Output in the desired range ([0,1], {-1,1}, ...)
- Usually chosen from a (small) set of possibilities.
  - Sigmoid function
  - Hyperbolic tangent, or `tanh`, function
  - ReLU

$\mathbb{R}^+$

THE SIGMOID FUNCTION

$$f(z) = \frac{1}{1 + e^{-z}}$$

- Output real values . $\in (0, 1)$
- Natural interpretations as *probability*

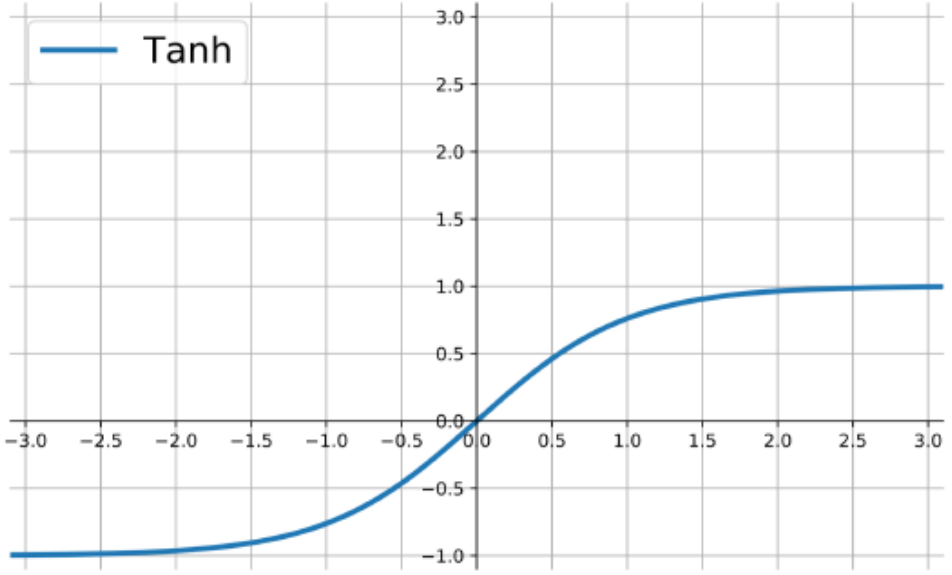


$$\begin{aligned}
\frac{d}{dx}\sigma(x) &= \frac{d}{dx}\left[\frac{1}{1+e^{-x}}\right] \qquad \frac{d}{dx}e^{x} = e^{x} \\
&= \frac{d}{dx}\left(1+e^{-x}\right)^{-1} \\
&= -(1+e^{-x})^{-2}(-e^{-x}) \\
&= \frac{e^{-x}}{(1+e^{-x})^{2}} \\
&= \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}} \\
&= \frac{1}{1+e^{-x}} \cdot \frac{(1+e^{-x})-1}{1+e^{-x}} \\
&= \frac{1}{1+e^{-x}} \cdot \left(\frac{1+e^{-x}}{1+e^{-x}} - \frac{1}{1+e^{-x}}\right) \\
&= \frac{1}{1+e^{-x}} \cdot \left(1 - \frac{1}{1+e^{-x}}\right) \\
&= \sigma(x) \cdot (1 - \sigma(x))
\end{aligned}$$

THE HYPERBOLIC TANGENT

Also called `tanh`, function:

$$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



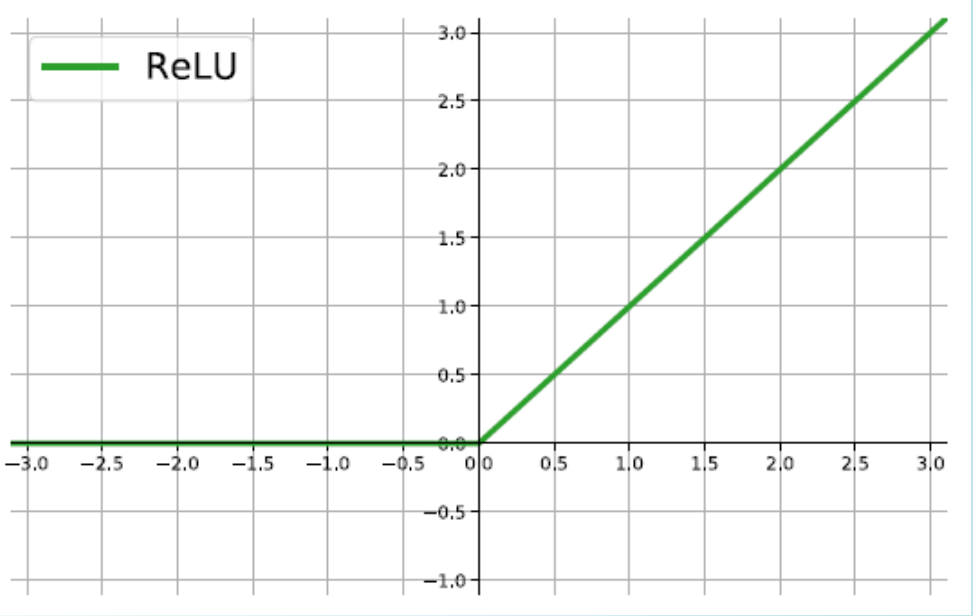
- outputs are zero-centered and bounded in −1,1
- scaled and shifted Sigmoid
- stronger gradient but still has vanishing gradient problem
- Its derivative is .

$$f'(z) = 1 - (f(z))^2$$

THE RELU

- *rectified linear unit*: .
- Close to a linear: piece-wise *linear* function with two linear pieces.
- Outputs are in %(0,)$, thus not bounded
- Half rectified: activation threshold at 0
- No vanishing gradient problem

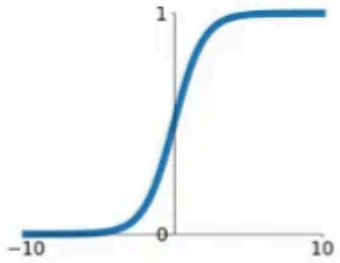
$$f(z) = \max\{0, z\}$$



MORE ACTIVATION FUNCTIONS

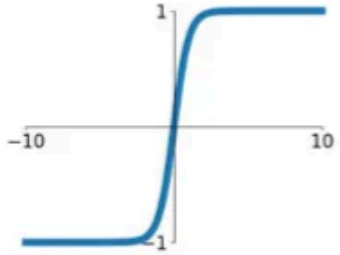
Sigmoid

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



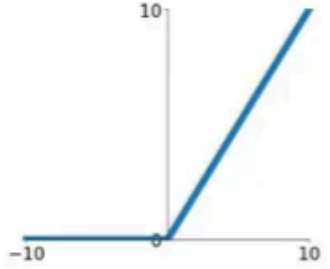
tanh

$$\tanh(x)$$



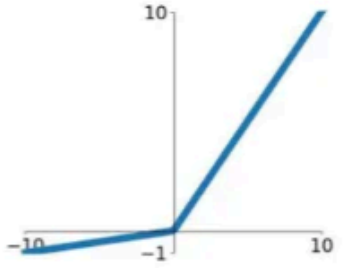
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

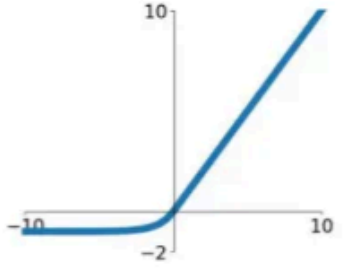


Maxout

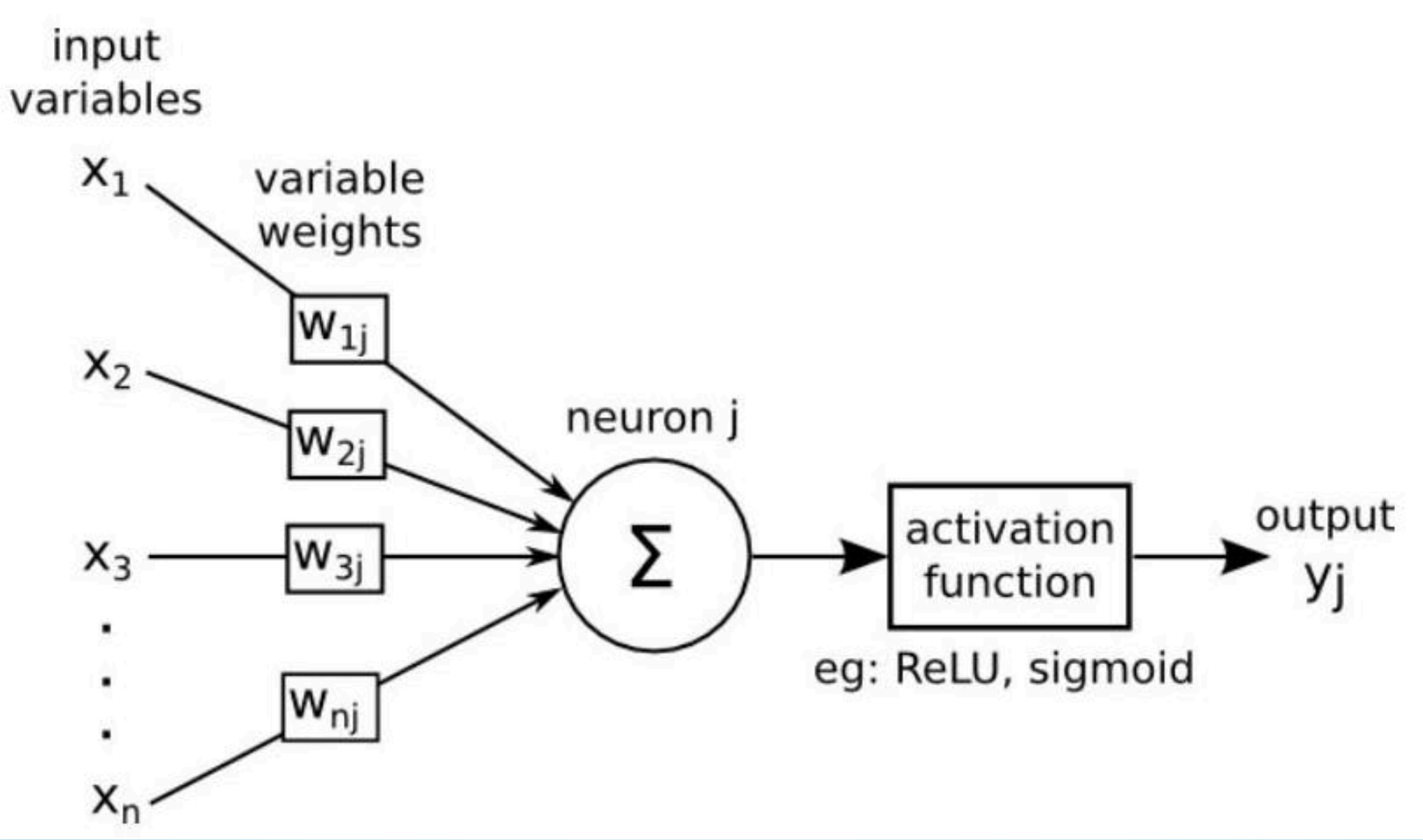
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



PUTTING IT ALL TOGETHER



IN WORDS

- An ANN takes a vector of input values and combines it with some weights that are local to the neuron to compute a net input, $x_1, \ldots, x_d$
- To compute its output, it then passes the net input through a possibly non-linear univariate activation function , usually vchosen from a set of options such as *Sigmoid*, *Tanh* or *ReLU* functions $(w_0, w_1, \ldots, w_d)$ $w_0 + \sum_{i=1}^{d} w_i \cdot x_i$ $g(\cdot)$
- To deal with the *bias*, we create an extra input variable with value always equal to 1 , and so the function computed by a single artificial neuron (parameterized by its weights) is: x_0 $\mathbf{w}$

$$y(\mathbf{x}) = g\left(w_0 + \sum_{i=1}^{d} w_i x_i\right) = g\left(\sum_{i=}\right.$$

FROM NEURONS TO NEURAL NETWORKS

THE BASIC NEURAL NETWORK

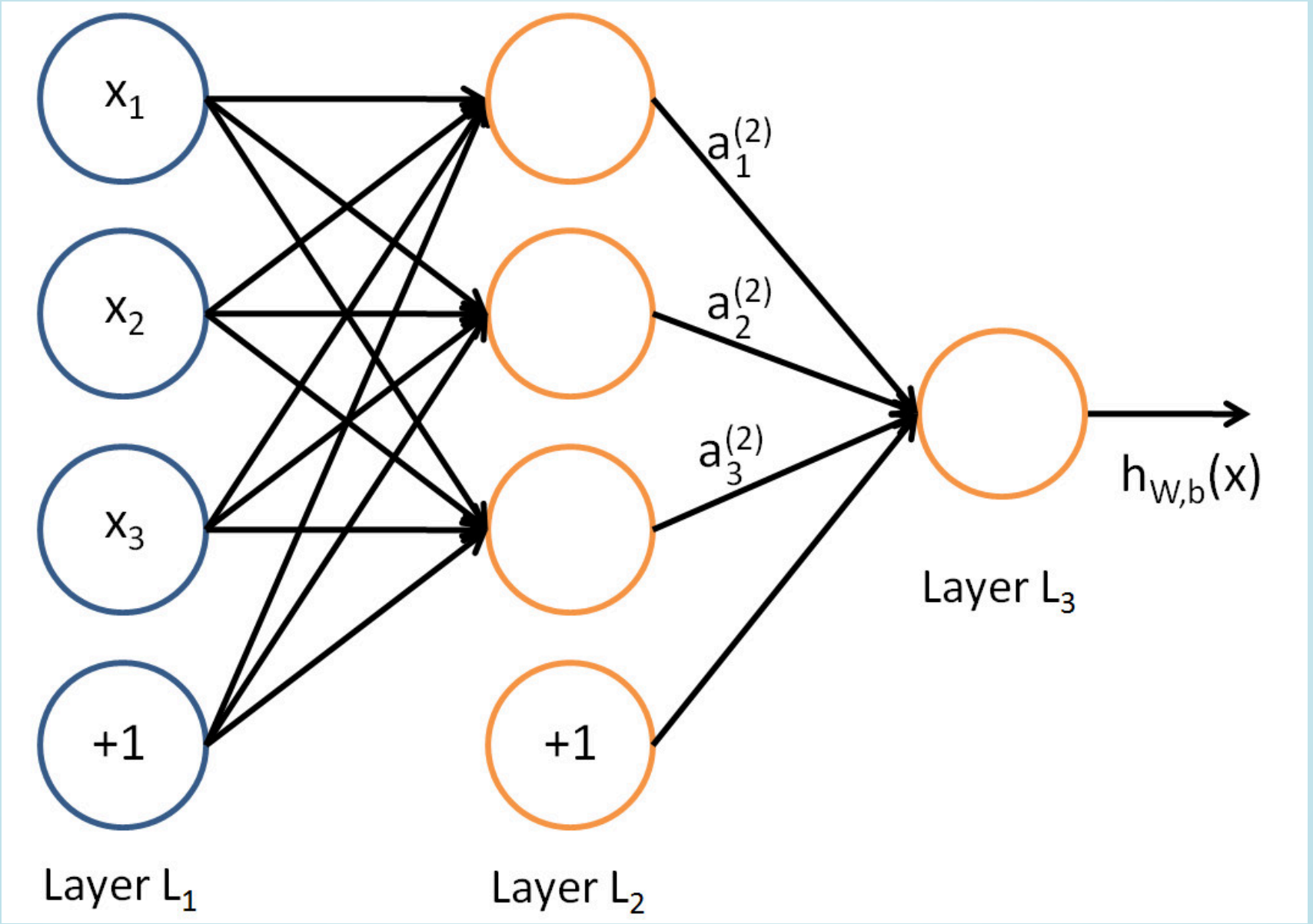
Following with the brain analogy one can combine (artificial) neurons to create better learners.

A simple artificial neural network is usually created by combining two types of modifications to the basic perceptron (AN).

- Stacking several neurons insteads of just one.
- Adding an additional layer of neurons, which is call a *hidden* layer,

This yields a system where the output of a can be the input of another in many different ways.

AN ARTIFICIAL NEURAL NETWORK



THE ARCHITECTURE OF ANN

In this figure, we have used circles to also denote the inputs to the network.

- Circles labeled +1 are *bias units*, and correspond to the intercept term.
- The leftmost layer of the network is called the *input layer*.
- The rightmost layer of the network is called the *output* layer.
- The middle layer of nodes is called the *hidden layer*, because its values are not observed in the training set.

Bias nodes are not counted when stating the neuron size.

With all this in mind our example neural network has three layers with:

- 3 input units (not counting the bias unit),
- 3 hidden units,
- 1 output unit.

HOW AN ANN WORKS

An ANN is a predictive model (a *learner*) whose properties and behaviour can be well characterized.

- It operates through a process known as *forward propagation*, which encompasses the information flow from the input layer to the output layer.
- Forward propagation is performed by composing a series of linear and non-linear (activation) functions.
- These are characterized (parametrized) by their *weights* and *biases*, that need to be *learnt*.
  - This is done by *training the ANN*.

TRAINING THE ANN

- In order for the ANN to perform well, the training process aims at finding the best possible parameter values for the learning task defined by the fnctions. This is done by
  - Selecting an appropriate (convex) loss function,
  - Finding those weights that minimize a the total *cost* function (avg. loss).
- This is usually done using some iterative optimization procedure such as *gradient descent*.
  - This requires evaluating derivatives in a huge number of points.
  - Such high number may be reduced by *Stochastic Gradient Descent*.
  - The evaluation of derivatives is simplified thanks to *Backpropagation*.

FORWARD PROPAGATION

As described above the process that encompasses the computations required to go from the input values to the final output is known as *forward propagation*.

The weights are combined with the input to produce the final output.

Each node, $a_i^{(2)}$ of the hidden layer opperates on all nodes of the input values

$$a_1^{(2)} = f(\theta_{10}^{(1)} + \theta_{11}^{(1)} x_1 + \theta_{12}^{(1)} x_2 + \theta_{13}^{(1}$$

$$a_2^{(2)} = f(\theta_{20}^{(1)} + \theta_{21}^{(1)} x_1 + \theta_{22}^{(1)} x_2 + \theta_{23}^{(1}$$

The output of the hidden layer is transformed through the activation function:

$$a_3^{(2)} = f(\theta_{30}^{(1)} + \theta_{31}^{(1)} x_1 + \theta_{32}^{(1)} x_2 + \theta_{33}^{(1}$$

$$h_\Theta(x) = a_1^{(3)} = f(\theta_{10}^{(2)} + \theta_{11}^{(2)} a_1^{(2)} + \theta^{(}$$

A COMPACT NOTATION (1)

Let $z_i^{(l)}$ denote the total weighted sum of inputs to unit i in layer l:

$$z_i^{(2)} = \theta_{i0}^{(1)} + \theta_{i1}^{(1)} x_1 + \theta_{i2}^{(1)} x_2 + \theta_{i3}^{(1)} x_3$$

the output becomes: $a_i^{(l)} = f(z_i^{(l)})$.

Extending the activation function $f(\cdot)$ to apply elementwise to vectors:

$$f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)],$$

we can write the previous equations more compactly as:

$$z^{(2)} = \Theta^{(1)} x$$
$$a^{(2)} = f(z^{(2)})$$
$$z^{(3)} = \Theta^{(2)} a^{(2)}$$
$$h_\Theta(x) = a^{(3)} = f(z^{(3)})$$

A COMPACT NOTATION (2)

More generally, recalling that we also use $a^{(1)} = x$ to also denote the values from the input layer,

Given layer l's activations $a^{(l)}$, we can compute layer $l+1$'s activations $a^{(l+1)}$ as:

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = f(z^{(l+1)})$$

A COMPACT NOTATION (3)

This can be used to provide a matrix representation for the weighted sum of inputs of all neurons:

$$z^{(l+1)} = \begin{bmatrix} z_1^{(l+1)} \\ z_2^{(l+1)} \\ \vdots \\ z_{s_{l+1}}^{(l)} \end{bmatrix} = \begin{bmatrix} \theta_{10}^{(l)} & \theta_{11}^{(l)} \\ \theta_{20}^{(l)} & \theta_{21}^{(l)} \\ \vdots & \vdots \\ \theta_{s_{l+1}0}^{(l)} & \theta_{s_{l+1}1}^{(l)} \end{bmatrix}$$

A COMPACT NOTATION (4)

So that, the activation is then:

$$a^{(l+1)} = \begin{bmatrix} a_1^{(l+1)} \\ a_2^{(l+1)} \\ \vdots \\ a_{s_{l+1}}^{(l)} \end{bmatrix} = f(z^{(l+1)}) = \begin{bmatrix} f \\ f \\ \\ \\ . \end{bmatrix}$$

EFICIENT FORWARD PROPAGATION

- The way input data is transformed, through a series of weightings and transformations, until the ouput layer is called *forward propagation*.
- By organizing parameters in matrices, and using matrix-vector operations, fast linear algebra routines can be used to perform the required calculations in a fast efficent way.

MULTIPLE ARCHITECTURES FOR ANN

- We have so far focused on a single hidden layer neural network of the example.
- One can. however build neural networks with many distinct architectures (meaning patterns of connectivity between neurons), including ones with multiple hidden layers.
- See here the Neural Network Zoo.

MULTIPLE ARCHITECTURES FOR ANN

- We have so far focused on a single hidden layer neural network of the example
- One can build neural networks with many distinct architectures (meaning patterns of connectivity between neurons), including ones with multiple hidden layers.

MULTIPLE LAYER DENSE NETWORKS

- Most common choice is a -layered network:
  - layer 1 is the input layer, n_l
  - layer is the output layer,
  - and each layer is densely connected to layer .
- In this setting, to compute the output of the network, we can compute all the activations in layer , then layer , and so on, up to layer , using equations seen previously.

L
n_l
l
$l+1$
L_2
L_3
L_{nl}

FEED FORWARD NNS

- The type of NN described is called feed-forward *neural network (FFNN)*, since
  - All computations are done by Forward propagation
  - The connectivity graph does not have any directed loops or cycles.

TRAINING NEURAL NETWORKS

TRAINING AN ANN

- An ANN is a predictive model whose properties and behaviour can be mathematically characterized.
- In practice this means:
  - The ANN acts by composing a series of linear and non-linear (activation) functions.
  - These are characterized by their *weights* and *biases*, that need to be *learnt* .
- *Training* the network is done by
  - Selecting an appropriate (convex) loss function,
  - Finding those weights that minimize a the total *cost* function (avg loss).

THE TOOLS FOR TRAINING

- Training an ANN is usually done using some iterative optimization procedure such as *Gradient Descent*.
- This requires evaluating derivatives in a huge number of points.
  - Such high number may be reduced by *Stochastic Gradient Descent*.
  - The evaluation of derivatives is simplified thanks to *Backpropagation*.

A LOSS FUNCTION FOR OPTIMIZATION

Depending on the form of the activation function we may decide to use one or another form of loss function.

- At first an idea may be to use *squared error loss*:

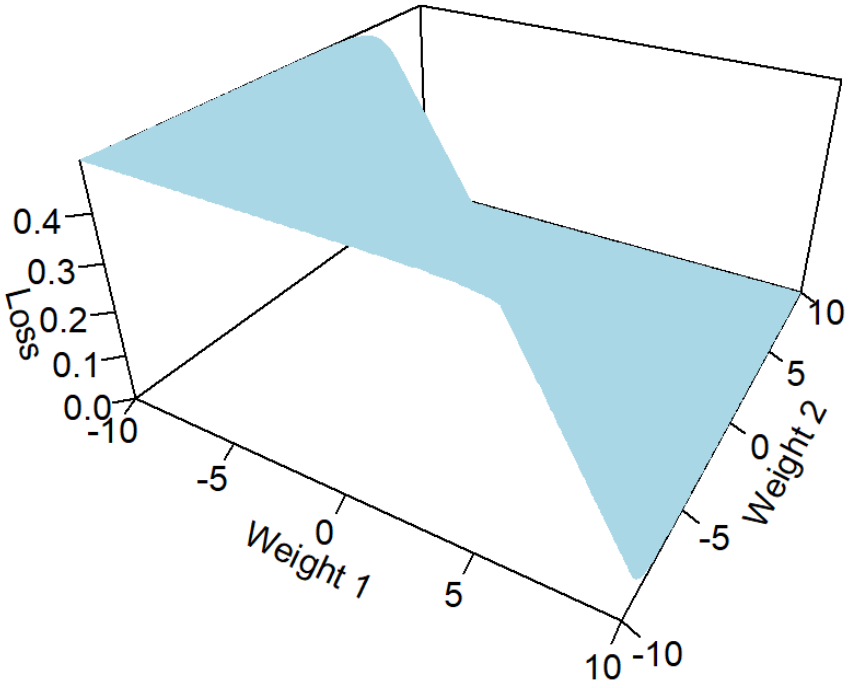
$$l(h_\theta(x), y) = \left(y - \frac{1}{1+e^{-\theta^T x}} \right)^2$$

- However it happens to be that, given a sigmoid activation function, the resulting loss function * is not a convex problem* which means that MSE is not appropriate.
- Quadratic loss may be used, for instance, with ReLu activation.

ILLUSTRATING NON-CONVEXITY

```r
library(plot3D)

# Define the squared error loss function
squared_error <- function(y, y_hat) {
  return(0.5 * (y - y_hat)^2)
}

# Define the logistic activation function
logistic <- function(z) {
  return(1 / (1 + exp(-z)))
}

# Generate data
x <- seq(-10, 10, length.out = 200)
y <- seq(-10, 10, length.out = 200)
z <- outer(x, y, FUN = function(x, y) squared_error(1, logistic(x + y)))

# Plot the exaggerated loss surface
persp3D(x, y, z, theta = 30, phi = 30, expand = 0.5, col = "lightblue",
        ticktype = "detailed", xlab = "Weight 1", ylab = "Weight 2", zlab = "Loss")
```

ILLUSTRATING NON-CONVEXITY



CROSS-ENTROPY LOSS FUNCTION

This function can also be written as:

$$l(h_\theta(x), y) = \begin{cases} -\log h_\theta(x) & \text{if} \\ -\log(1 - h_\theta(x)) & \text{if} \end{cases}$$

$$l(h_\theta(x), y) = -y \log h_\theta(x) - (1 - y)$$

Using cross-entropy loss, the cost function is of the form:

$$J(\theta) = -\frac{1}{n}\Big[\sum_{i=1}^{n} (y^{(i)} \log h_\theta(x^{(i)}) +$$

Now, this is a convex optimization problem.

$$(1 - y^{(i)}) \log(1 - h$$

REGULARIZED CROSS ENTROPY

In practice we often work with a *regularized version* of the cost function (we don't regularize the bias units)

$$J(\Theta) = -\frac{1}{n}\left[\sum_{i=1}^{n}\sum_{k=1}^{K} y_k^{(i)}\right.$$

$$+ (1 - y_k^{(i)}) \log(1 -$$

$$+ \lambda \sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}} (\theta_{ji}^{(l)}$$

GRADIENT DESCENT

- Training a network corresponds to choosing the parameters, that is, the weights and biases, that minimize the cost function.
- The weights and biases take the form of matrices and vectors, but at this stage it is convenient to imagine them stored as a single vector that we call .
- Generally, we will suppose , and write the cost function as to emphasize its dependence on the parameters. So Cost .

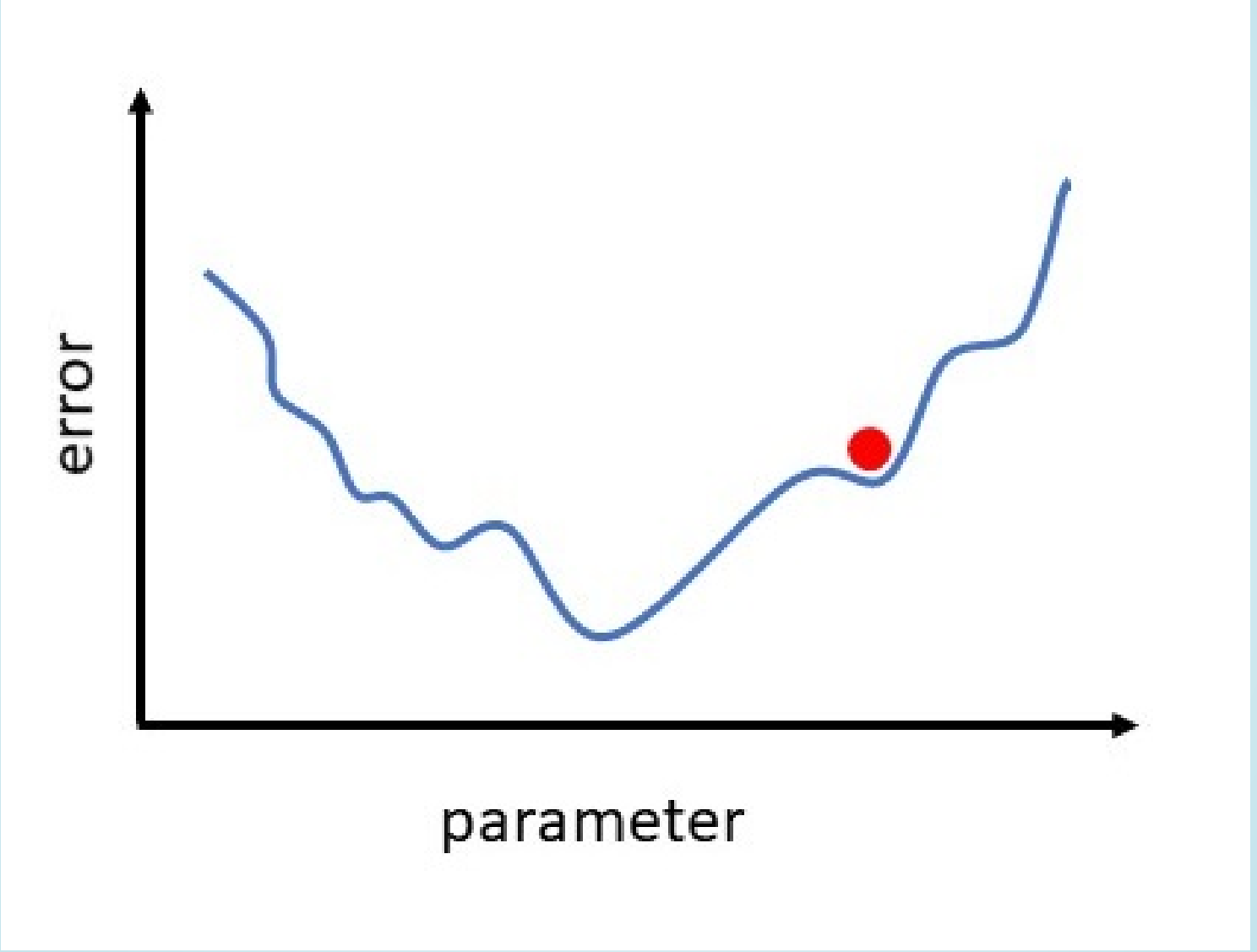
θ

$\theta \in \mathbb{R}^p$

$J(\theta)$

$$J : \mathbb{R}^p \to \mathbb{R}$$

GRADIENT DESCENT



Error hyper-surface

GRADIENT DESCENT

- *Gradient Descent* is a classical method in optimization that is often referred to as *steepest descent*.
- Given a function $J(\theta)$ to be minimized, the method proceeds iteratively, computing a sequence of vectors $\theta^1, \theta^2, \ldots, \theta^n$ in $\mathbb{R}^p$ with the aim of converging to a vector that minimizes the cost function.
- Suppose that our current vector is θ. *How should we choose a perturbation, $\Delta\theta$, so that the next vector, $\theta + \Delta\theta$, represents an improvement, that is:* $J(\theta + \Delta\theta) < J(\theta)$?

GRADIENT DESCENT

- We proceed by linearization of the cost function using a Taylor approximation
- If $\Delta\theta$ is small, then ignoring terms of order $||\Delta\theta||^2$ or higher:

$$J(\theta + \Delta\theta) \approx J(\theta) + \sum_{i=1}^{p} \frac{\partial J(\theta)}{\partial \theta_i} \Delta\theta_i$$

- where $\frac{\partial J(\theta)}{\partial \theta_i}$ denotes the partial derivative of the cost function with respect to the i-th weight

GRADIENT DESCENT

- Let denote the *gradient*, i.e. the vector of partial derivatives.
  $\nabla J(\theta) \in \mathbb{R}^p$

$$\nabla J(\theta) = \left(\frac{\partial J(\theta)}{\partial \theta_1}, \ldots, \frac{\partial J(\theta)}{\partial \theta_p} \right)^\intercal$$

- Now the approximation can be written as:

$$J(\theta + \Delta\theta) \approx J(\theta) + \nabla J(\theta)^\intercal \Delta\theta$$

GRADIENT DESCENT

- Recalling that our aim is to reduce the value of the cost function,
- Taylor approximation above motivates the idea of choosing $\Delta\theta$ to *make* $\nabla_\theta J(\theta)^\mathsf{T}\Delta\theta$ *negative*,
- because this will make the value of $J(\theta + \Delta\theta)$ smaller.
- The bigger in absolute value we can make this negative expression, the smaller will be the value of the cost function.

THE CAUCHY-SCHWARZ INEQUALITY

- The Cauchy-Schwarz inequality, states that for any , we have: f, g

$$|f^\mathsf{T} g| \leq ||f|| \cdot ||g||.$$

- Moreover, the two sides are equal if and only if and are linearly dependent (meaning they are parallel). f g

BACK TO GRADIENT DESCENT

- How much can decrease? $\nabla J(\theta)^{\mathsf{T}} \Delta\theta$
- By Cauchy-Schwarz, biggest possible value for is the upper bound, . $\nabla J(\theta)^{\mathsf{T}}$ $||\nabla J(\theta)|| \cdot ||\Delta\theta||$
  - The equality is only reached when $||\nabla J(\theta)|| = ||\Delta$
  - The highest possible negative value will come out when $-\nabla J(\theta) = \Delta\theta$
- That is, *we should choose to lie in the direction of* . $\Delta\theta$ $-\nabla J$

GRADIENT DESCENT

- Keeping in mind that the Taylor linearization of $J(\theta)$ is an approximation that is relevant only for small $\Delta\theta$,
- We will limit ourselves to a small step in that direction, defined by the *learning rate* η. .
- This leads to the update formula that defines the steepest descent method.

$$\theta \rightarrow \theta - \eta \nabla J(\theta)$$

GRADIENT DESCENT

In summary, givent a cost function $J(\theta)$ to be optimized the gradient descent optimization proceeds as follows:

1. **Initialize** θ_0 randomly or with some predetermined values
2. **Repeat until convergence:**

$$\theta_{t+1} = \theta_t - \eta \nabla J(\theta_t)$$

3. **Stop when:**

$$|J(\theta_{t+1}) - J(\theta_t)| < \epsilon$$

- θ_0 is the initial parameter vector,
- θ_t is the parameter vector at iteration t,
- η is the learning rate,
- $\nabla J(\theta_t)$ is the gradient of the loss function with respect to θ at iteration t,
- ϵ is a small positive value indicating the desired level of convergence.

COMPUTING GRADIENTS

- The gradient method provides a way to optimize weights and biases by minimizing the cost function, . $(\theta = \{W$
- This minimization requires, of course, the computation of an important number of partial derivatives $J(\theta)$

$$\frac{\partial}{\partial \theta_j} J(\theta)$$

- The algorithm used to perform these computation is known as the *backpropagation algorithm*

A SHORT HISTORY

- The backpropagation algorithm was originally introduced in the 1970s in a MSc thesis.
- In 1986 a paper by Rumelhart, Hinton, and Williams describes several neural networks where backpropagation works far faster than earlier approaches to learning.
- This improvement in efficiency made it possible to use neural nets to solve problems which had previously been insoluble.

ERROR BACKPROPAGATION

$$\Delta w = \eta(c - y)d \quad (3.21)$$

- La regla delta es la base de la retropropagación:
  - c es el valor que indica la clase en el ejemplo de entrenamiento,
  - y el valor de la función de salida para la instancia , y
  - d la tasa o velocidad de aprendizaje (*learning rate*).
- El método utilizado para entrenar este tipo de redes, y basado en la regla delta, se muestra en el algoritmo siguiente (Algoritmo 1).
  - Para que el algoritmo converja es necesario que las clases de los datos sean linealmente separables.

ALGORITMO 1 (GRADIENTE)

- **Entrada**: W (conjunto de vectores de pesos) y D (conjunto de instancias de entrenamiento)
  - mientras $(y \not\approx c_i \forall (d_i, c_i) \in D)$ hacer
    - para todo $((d_i, c_i) \in D)$ hacer
      - Calcular la salida y de la red cuando la entrada es d_i
      - si $(y \not\approx c_i)$ entonces
        - Modificar el vector de pesos $w' = w + \eta(c - y)d_i$
      - fin si
    - fin para
  - fin mientras

Devolver: El conjunto de vectores de pesos W

RETROPROPAGACIÓN DEL ERROR

- En las redes multicapa no podemos aplicar el algoritmo de entrenamiento visto en la sección anterior.
- El problema aparece con los nodos de las capas ocultas: no podemos saber a priori cuáles son los valores de salida correctos.
- En el caso de una neurona j con función sigmoide, la regla delta es:

$$\Delta w_i^j = \eta \sigma' \left(z^j \right) \left(c^j - y^j \right) x_i^j$$

donde:

- $\sigma' \left(z^j \right)$ indica la pendiente (derivada) de la función sigmoide, que representa el factor con que el nodo j puede afectar al error.

AJUSTANDO LOS PESOS

$$\Delta w_i^j = \eta \sigma' \left(z^j \right) \left(c^j - y^j \right) x_i^j$$

- Si el valor de $\sigma' \left(z^j \right)$ es pequeño, nos encontramos en los extremos de la función, donde los cambios no afectan demasiado a la salida.
- Por el contrario, si el valor es grande, nos encontramos en el centro de la función, donde pequeñas variaciones pueden alterar considerablemente la salida.
  - representa la medida del error que se produce en la neurona .
  - indica la *responsabilidad* de la entrada de la neurona en el error.
- Cuando este valor es igual a cero, no se modifica el peso, mientras que si es superior a cero, se modifica proporcionalmente a este valor.

$\left(c^j - y^j \right)$ j

x_i^j i j

EXPRESION GENERAL DE *delta*

- La δ que corresponde a la neurona j puede expresarse de forma general para toda neurona, simplificando la notación anterior:

$$\delta^j = \sigma'\left(z^j\right)\left(c^j - y^j\right) \quad (3.25$$

$$\Delta w_i^j = \eta \delta^j x_i^j \quad (3.26$$

- A partir de aquí debemos determinar qué parte del error total se asigna a cada una de las neuronas de las capas ocultas.
- Es decir, debemos definir *cómo modificar los pesos y tasas de aprendizaje de las neuronas de las capas ocultas a partir del error observado en la capa de salida*.

LA RETROPROPAGACIÓN

- El método de retropropagación (backpropagation) se basa en un esquema general de dos pasos:

1. Propagación hacia adelante (*feedforward*), que consiste en introducir una instancia de entrenamiento y obtener la salida de la red neuronal.
2. Propagación hacia atrás (*backpropagation*), que consiste en calcular el error cometido en la capa de salida y propagarlo hacia atrás para calcular los valores delta de las neuronas de las capas ocultas.

- La idea que subyace a este método es relativamente sencilla, y se basa en *propagar el error de forma proporcional a la influencia que ha tenido cada nodo de las capas ocultas en el error final producido por cada una de las neuronas de la capa de salida*.

EL ALGORITMO (1)

1. El primer paso, que es la propagación hacia adelante, consiste en aplicar el ejemplo a la red y obtener los valores de salida (línea 3).
2. A continuación, el método inicia la propagación hacia atrás, empezando por la capa de salida. Para cada neurona j de la capa de salida:

2.1 Se calcula, en primera instancia, el valor δ^j basado en el valor de salida de la red para la neurona j $\left(a^j\right)$, el valor de la clase de la instancia $\left(c^j\right)$ y la derivada de la función sigmoide $\left(\sigma'\left(z^j\right)\right)$ (línea 5).

2.2 A continuación, se modifica el vector de pesos de la neurona de la capa de salida, a partir de la tasa de aprendizaje (η), el valor delta de la neurona calculado en el paso anterior $\left(\delta^j\right)$ y el factor x_i^j que indica la responsabilidad de la entrada i de la neurona j en el error (línea 6).

EL ALGORITMO (2)

3. Finalmente, la propagación hacia atrás se aplica a las capas ocultas de la red. Para cada neurona k de las capas ocultas:

3.1 En primer lugar, se calcula el valor δ_k basado en la derivada de la función sigmoide $\left(\sigma'\left(x_k^j\right)\right)$ y el sumatorio del producto de la delta calculada en el paso anterior $\left(\delta_l\right)$ por el valor w_{lk}^j que indica el peso de la conexión entre la neurona k y la neurona l (línea 9). El conjunto S_k está formado por todos los nodos de salida a los que se encuentra conectada la neurona k.

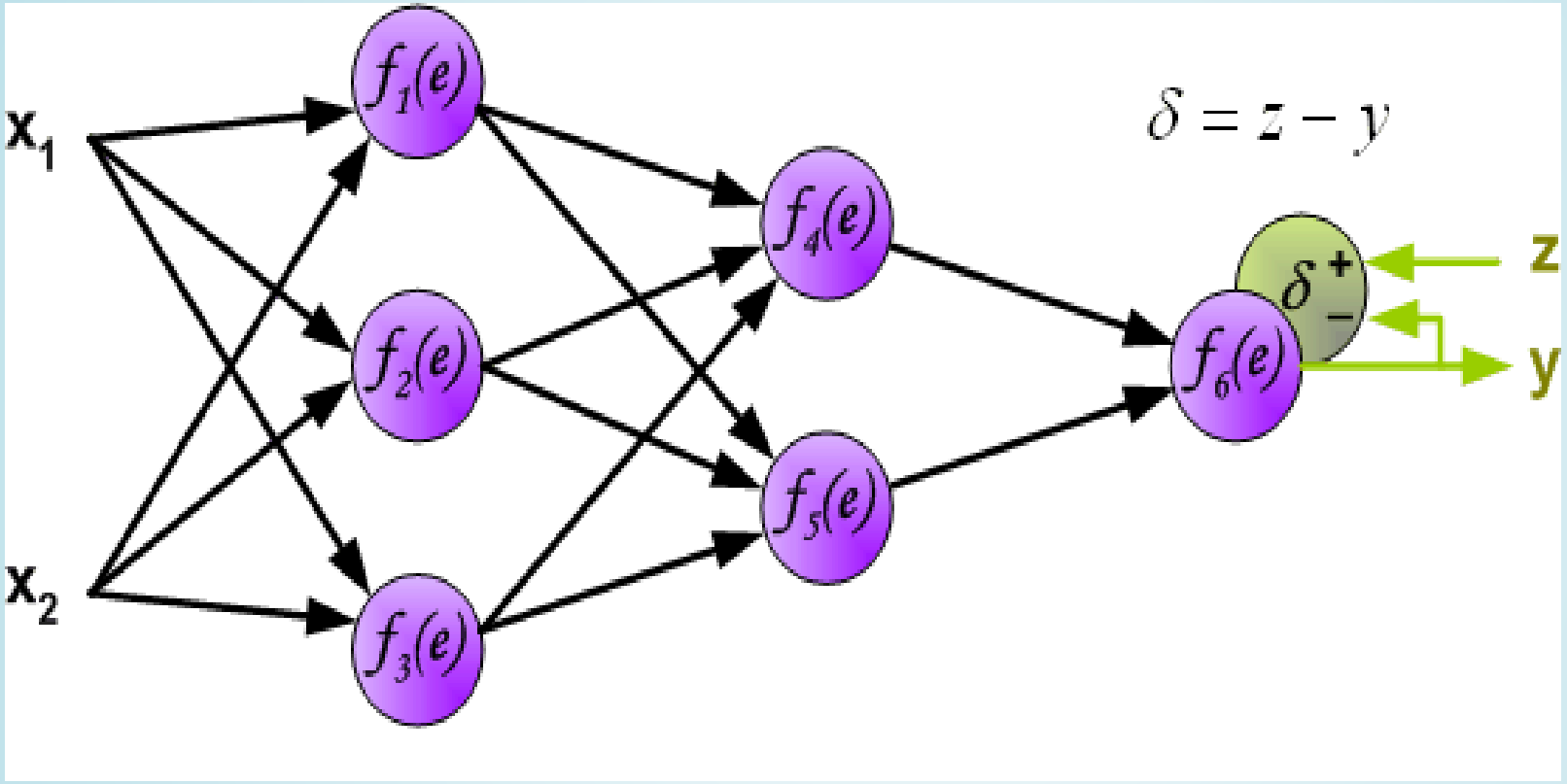
3.2 En el último paso de la iteración, se modifica el vector de pesos de la neurona k, a partir de la tasa de aprendizaje η, el valor delta de la neurona calculado en el paso anterior $\left(\delta_k^j\right)$ y el factor x_i^k que indica la responsabilidad de la entrada i de la neurona k en el error (línea 10).

EL ALGORITMO

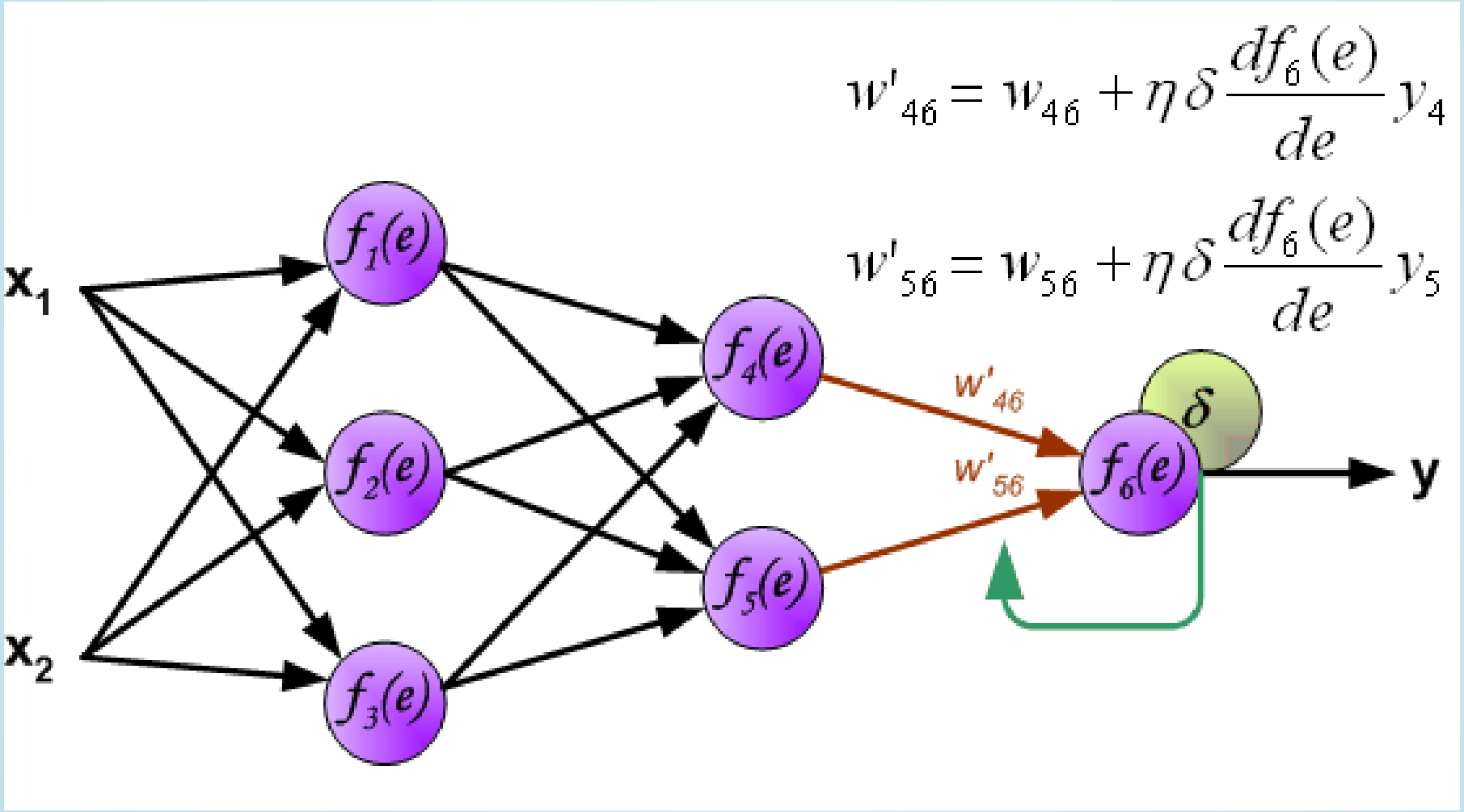
Entrada: W (conjunto de vectores de pesos) y D (conjunto de instancias de entrenamiento)

1: **mientras** (error de la red $>\varepsilon$) **hacer**
2: **para todo** $((d_i, c_i) \in D)$ **hacer**
3: Calcular el valor de salida de la red para la entrada d_i
4: **para todo** (neurona j en la capa de salida) **hacer**
5: Calcular el valor δ^j para esta neurona: $\delta^j = \sigma'(z^j)(c^j - y^j)$
6: Modificar los pesos de la neurona siguiendo el método del gradiente: $\Delta w_i^j = \eta \delta^j x_i^j$
7: **fin para**
8: **para todo** (neurona k en las capas ocultas) **hacer**
9: Calcular el valor δ^k para esta neurona: $\delta^k = \sigma'(z^k) \sum_{j \in S_k} \delta^k w_k^j$
10: Modificar los pesos de la neurona siguiendo el método del gradiente: $\Delta w_i^k = \eta \delta^k x_i^k$
11: **fin para**
12: **fin para**
13: **fin mientras**

FORWARD PROPAGATION EXAMPLE



BACKPROPAGATION ILLUSTRATED



IMPROVING THE LEARNING PROCESS

LEARNING OPTIMIZATION

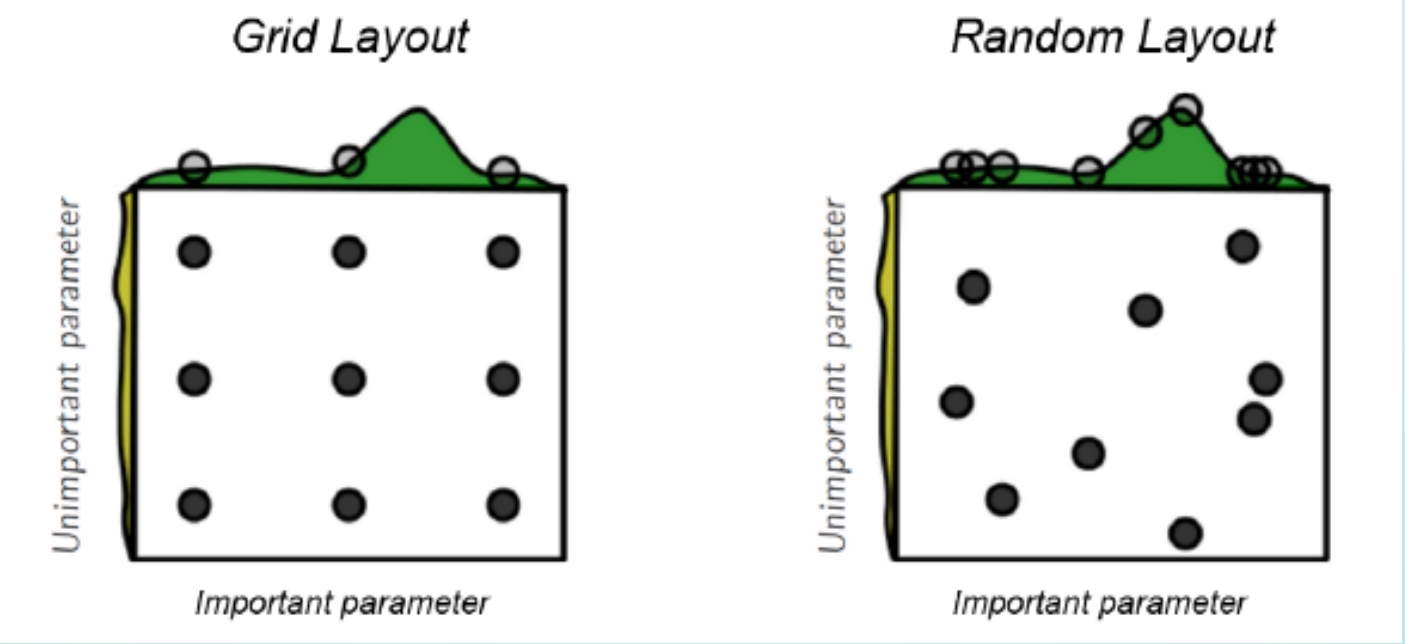
- The learning porocess such as it has been derived may be improved in different ways.
  - Predictions can be be bad and require improvement.
  - Computations may be inefficent or slow.
  - The network may overfit and lack generalizability.
- This can be partially soved applying distinct approaches.

NETWORK ARCHITECHTURE

- Network performance is affected by many hyperparameters
  - Network topology
  - Number of layers
  - Number of neurons per layer
  - Activation function(s)
  - Weights initialization procedure
  - etc.

HYPERPARAMETER TUNING

- Hyperparameters selection and tuning may be hard, due simply to dimensionality.
- Standard approaches to search for best parameters combinations are used.



HOW MANY (HIDDEN) LAYERS

- Traditionally considered that one layer may be enough
  - *Shallow Networks*
- Posterior research showed that adding more layers increases efficency
  - Number of neurons per layer decreases exponentially
- Although there is also risk of overfitting

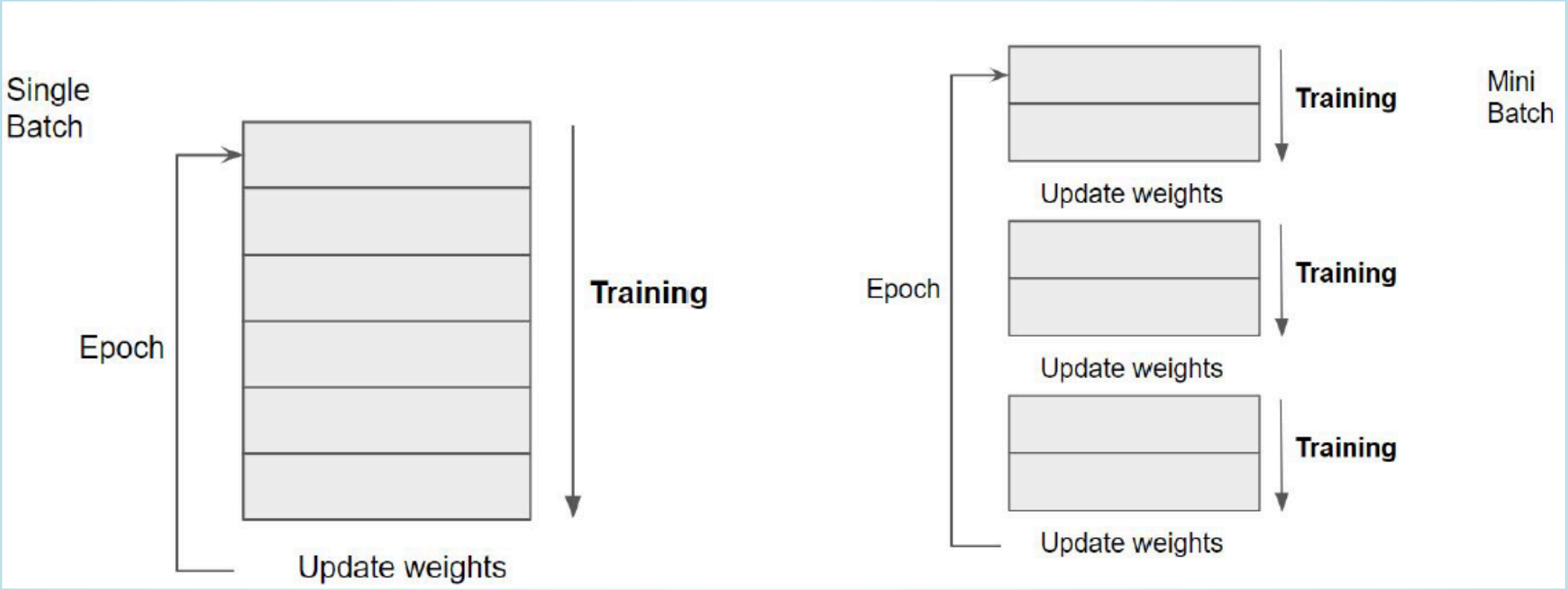
EPOCHS AND ITERATIONS

- It has been shown that using the whole training set only once may not be enough for training an ANN.
- One iteration of the training set is known as an *epoch*.
- The number of epochs , defines how many times we iterate along the whole training set.
- N_E can be fixed, determined by cross-validation or left open and stop the training when it does not improve anymore.

ITERATIONS AND BATCHES

- A complementary strategy to increasing the number of epochs is decreasing the number of instances in each iteration.
- That is, the training set is broken in a number of *batches* that are trained separately.
  - Batch learning allows weights to be updated more frequently per epoch.
  - The advantage of batch learning is related to the gradient descent approach used.

TRAINING IN BATCHES



GRADIENT DESCENT DRAWBACKS

- With many parameters and many training points, computing the gradient vector at every iteration of the steepest descent method can be time consuming.
  - It is mainly due to that we have *to sum across all training points*.
  - With big data this becomes prohibitive.
- Several alternatives exist (many indeed).

STOCHASTIC GRADIENT

- A much cheaper alternative is to replace the mean of the individual gradients over all training points *by the gradient at a single, randomly chosen, point*.
- This leads to the simplest form of the *stochastic gradient method*:
- Choose an integer i uniformly at random from $\{1, \ldots, n\}$ and update

$$\theta_j = \theta_j - \eta \frac{\partial}{\partial \theta_j} J(\theta; x^{(i)})$$

- incuded in the notation of $x^{(i)}$ to remark the dependence: $J(\theta; x^{(i)})$

RATIONALE FOR SGD

- At each step, the SGD method uses one randomly chosen training point to represent the full training set.
- As the iteration proceeds, the method sees more training points.
- So *there is some hope* that this dramatic reduction in cost-per-iteration will be worthwhile overall.
- Note that, even for very small , the update is not guaranteed to reduce the overall cost function because we traded the mean for a single sample.
- Hence, although the phrase stochastic gradient descent is widely used, we prefer to use **stochastic gradient**.

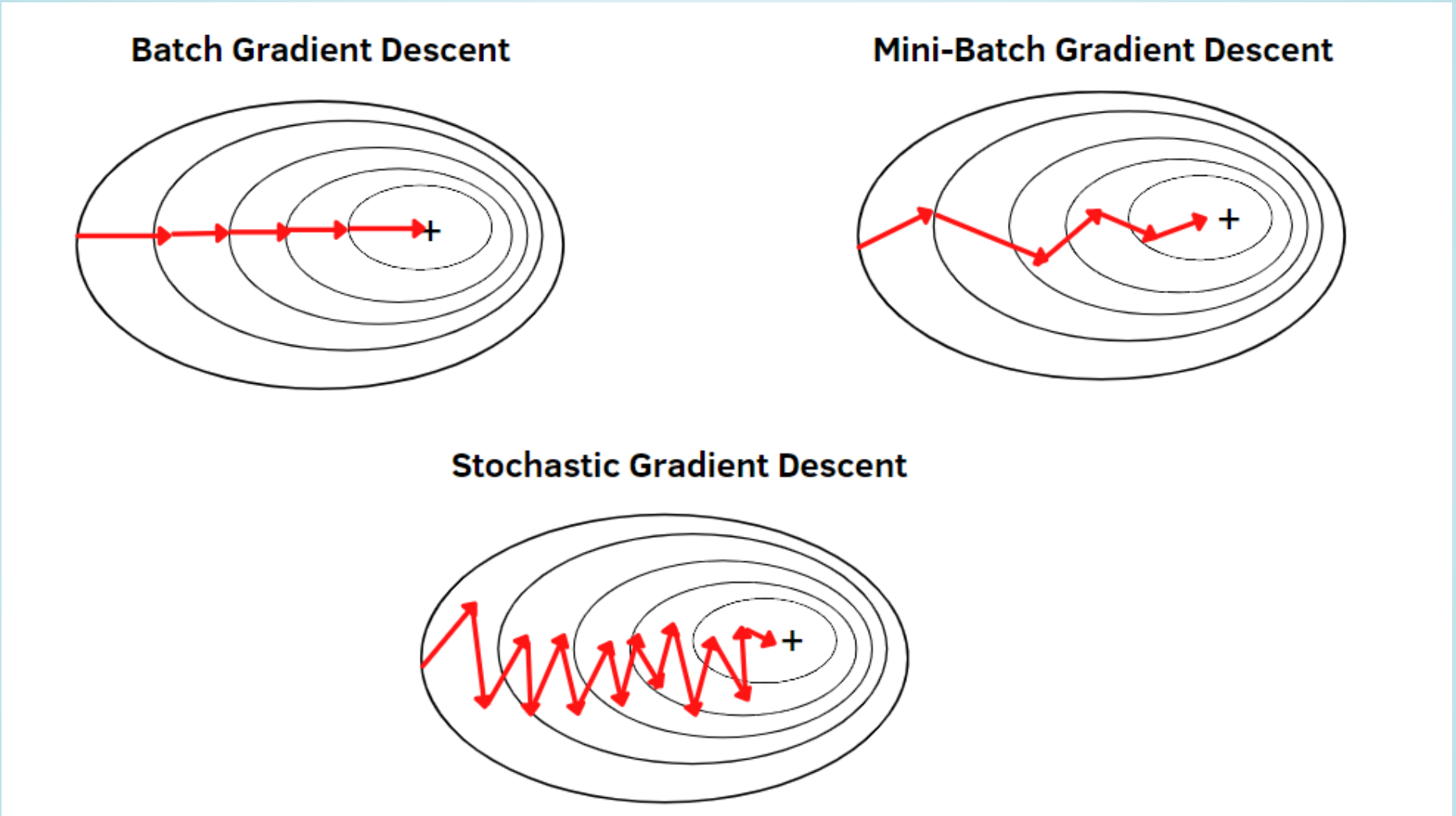
BATCH GRADIENT DESCENT

- Batch Gradient Descent computes the error for each example in the training dataset but the model is updated only after evaluating all examples.
- As a benefits there is computational efficiency, which produces a stable error gradient and a stable convergence.
- As drawbacks, the stable error gradient can sometimes result in a state of convergence that isn't the best the model can achieve. It also requires the entire training dataset to be in memory and available to the algorithm.

MINI-BATCH GD

- Mini-batch Gradient Descent (MBGD) divides the training dataset into small batches to compute error and update model coefficients.
- MBGD balances the robustness of stochastic gradient descent with the efficiency of batch gradient descent, making it a prevalent implementation in deep learning.
- A common batch size for MBGD is 32, although it can vary based on the specific problem and data.

TYPES OF GRADIENT DESCENT



Source: Gradient Descent and its Types

ALTERNATIVE OPTIMIZERS

- Gradient descent can be improved adopting different strategies
- Other optimizers exist that may make the training faster with strategies such as accumulating velocity in relevant directions or adjusting learning rates based on parameter frequency.
- Overall these optimizers enable quicker convergence and better handling of various data characteristics, making them favorable choices over traditional gradient descent.

MOMENTUM

- Accelerates SGD by accumulating momentum in relevant directions, akin to a rolling ball gaining speed downhill.
- Increases acceleration for dimensions with consistent gradients, reducing updates for changing gradients, leading to faster convergence and reduced oscillation.

ADAGRAD

- Adapts learning rate based on parameter frequency, making smaller updates for frequent features and larger updates for rare ones, suitable for sparse data.
- Efficient for handling sparse data due to its adaptive learning rates.

ADADELTA

- An extension of Adagrad aiming to mitigate its aggressive, monotonically decreasing learning rate.
- Restricts accumulated past gradients to a fixed-size window, avoiding over-accumulation.

ADAM

- Combines momentum optimization with adaptive learning rates, tracking both past gradients' exponential decay and a moving average of gradients.
- Recent studies suggest caution with adaptive optimization methods like Adam due to potential generalization issues, prompting exploration of alternatives such as Momentum optimization.

OPTIMIZING THE TRAINING SPEED

- Training speed optimization can be based on
  - Weight inizialization
  - Changing learning rate
  - Using efficient cost functions

OPTIMIZING TO AVOID OVERFITTING

In order to fight overfitting distinct strategies are common

- L2 regularization
- Early stopping
- Dropout
- Data augmentation

SUMMARIZING OPTIMIZATION

Tabla 4.1. Principales objetivos de las técnicas de optimización

Técnicas	Mejoras rendimiento	Velocidad aprendizaje	*Overfitting*
Arquitectura de la red	X	X	X
Épocas, iteraciones y *batch*		X	
Softmax	X		
Algoritmos de entrenamiento	X	X	
Inicialización pesos de la red		X	
Velocidad de aprendizaje	X	X	
Función de entropía cruzada		X	
Regularización L2			X
Early stopping			X
Dropout			X
Expansión conjunto de datos	X		X

Fuente: elaboración propia

Source: Introducción al Deep Learning (UOC)

AN EXAMPLE USING R

A PREDICTIVE ANN

We use the `neuralnet` package to build a simple neural network to predict if a type of stock pays dividends or not.

```
1 if (!require(neuralnet))
2   install.packages("neuralnet", dep=TRUE)
```

DATA FOR THE EXAMPLE

And use the `dividendinfo.csv` dataset from https://github.com/MGCodesandStats/datasets

```
1 mydata <- read.csv("https://raw.githubusercontent.com/MGCodesandStats/datas
2 str(mydata)
```

```
'data.frame':   200 obs. of  6 variables:
 $ dividend       : int  0 1 1 0 1 1 1 0 1 1 ...
 $ fcfps          : num  2.75 4.96 2.78 0.43 2.94 3.9 1.09 2.32 2.5 4.46 ...
 $ earnings_growth: num  -19.25 0.83 1.09 12.97 2.44 ...
 $ de             : num  1.11 1.09 0.19 1.7 1.83 0.46 2.32 3.34 3.15 3.33 ...
 $ mcap           : int  545 630 562 388 684 621 656 351 658 330 ...
 $ current_ratio  : num  0.924 1.469 1.976 1.942 2.487 ...
```

DATA PRE-PROCESSING

```
1 normalize <- function(x) {
2   return ((x - min(x)) / (max(x) - min(x)))
3 }
4 normData <- as.data.frame(lapply(mydata, normalize))
```

TEST AND TRAINING SETS

Finally we break our data in a test and a training set:

```
perc2Train <- 2/3
ssize <- nrow(normData)
set.seed(12345)
data_rows <- floor(perc2Train *ssize)
train_indices <- sample(c(1:ssize), data_rows)
trainset <- normData[train_indices,]
testset <- normData[-train_indices,]
```

TRAINING A NEURAL NETWORK

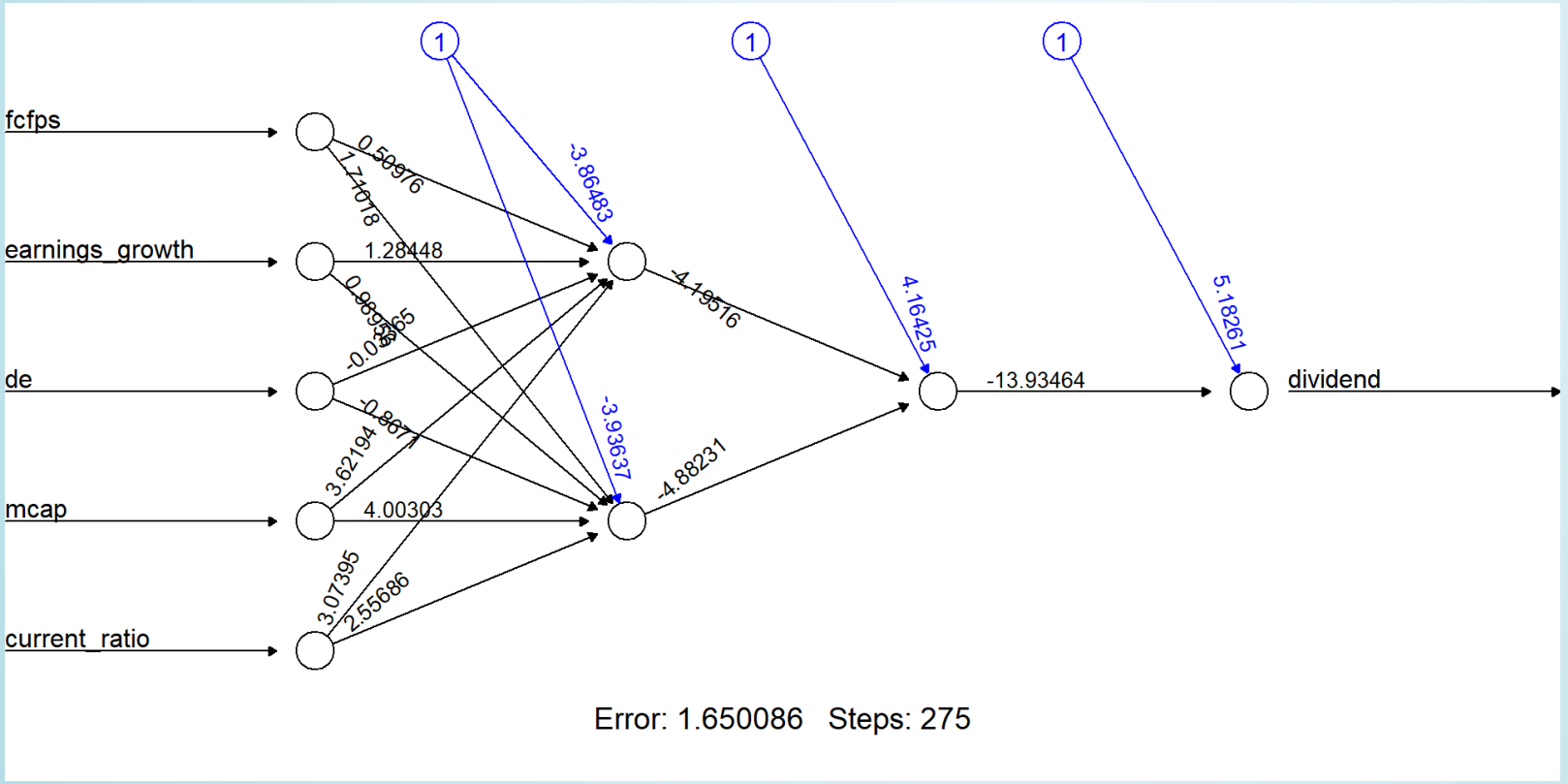
We train a simple NN with two hidden layers, with 4 and 2 neurons respectively.

```
#Neural Network
library(neuralnet)
nn <- neuralnet(dividend ~ fcfps + earnings_growth + de + mcap + current_ra
                data=trainset,
                hidden=c(2,1),
                linear.output=FALSE,
                threshold=0.01)
```

NETWORK PLOT

The output of the procedure is a neural network with estimated weights

```
1 plot(nn, rep = "best")
```



PREDICTIONS

```
temp_test <- subset(testset, select =
                        c("fcfps","earnings_growth",
                          "de", "mcap", "current_ratio"))
nn.results <- compute(nn, temp_test)
results <- data.frame(actual =
                      testset$dividend,
                      prediction = nn.results$net.result)
head(results)
```

```
   actual   prediction
9       1 0.9919213885
19      1 0.9769206123
22      0 0.0002187144
26      0 0.6093330933
27      1 0.7454164893
29      1 0.9515431416
```

MODEL EVALUATION

```
1 roundedresults<-sapply(results,round,digits=0)
2 roundedresultsdf=data.frame(roundedresults)
3 attach(roundedresultsdf)
4 table(actual,prediction)
```

```
      prediction
actual  0  1
     0 33  3
     1  4 27
```

REFERENCES AND RESOURCES