Lecture 4

Neural Networks I

This week Tutorial: Thursday 2nd Nov. 6th period

https://www.henrikbachmann.com/mml2023.html

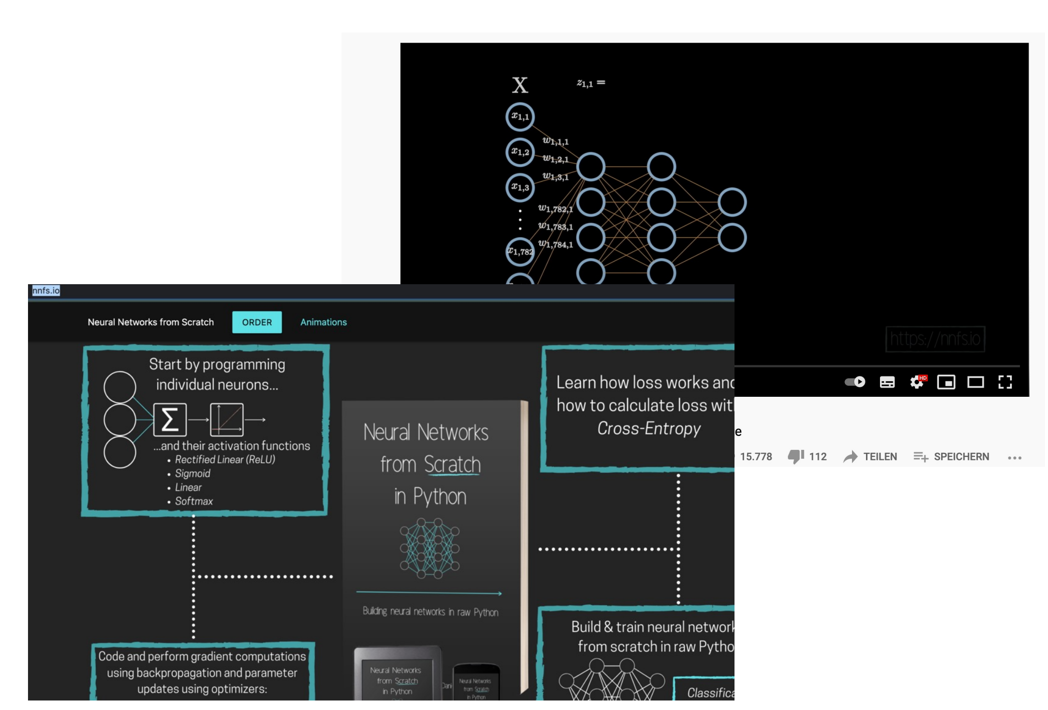
Nice to watch

3blue1brown

https://www.3blue1brown.com/neural-networks

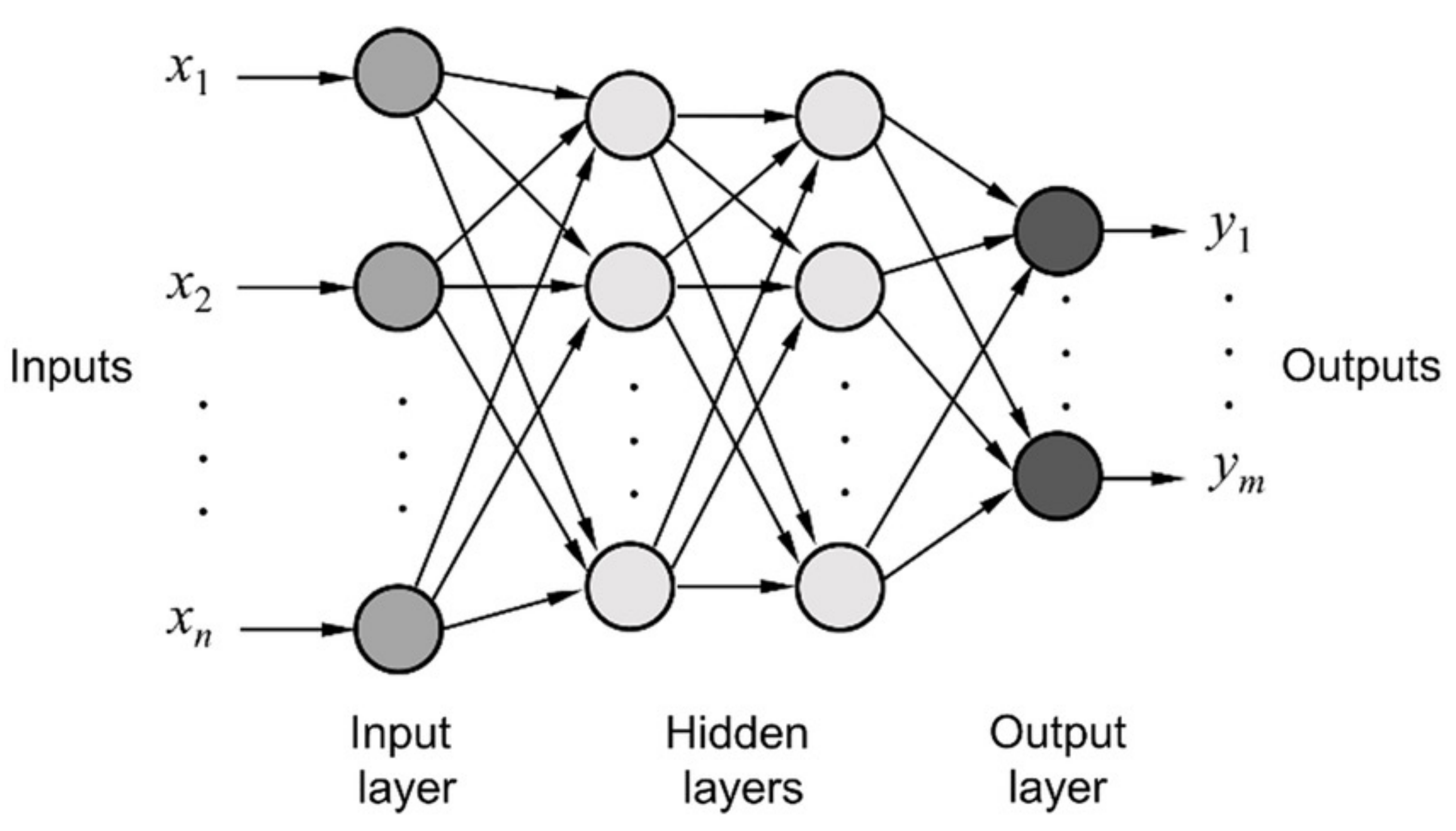
Neural Networks from Scratch

https://nnfs.io/



Neural Networks

We want to approximate/model functions $\mathbb{R}^n \to \mathbb{R}^m$.
In this course we will consider **multi-layer fully-connected feedforward neural networks**.

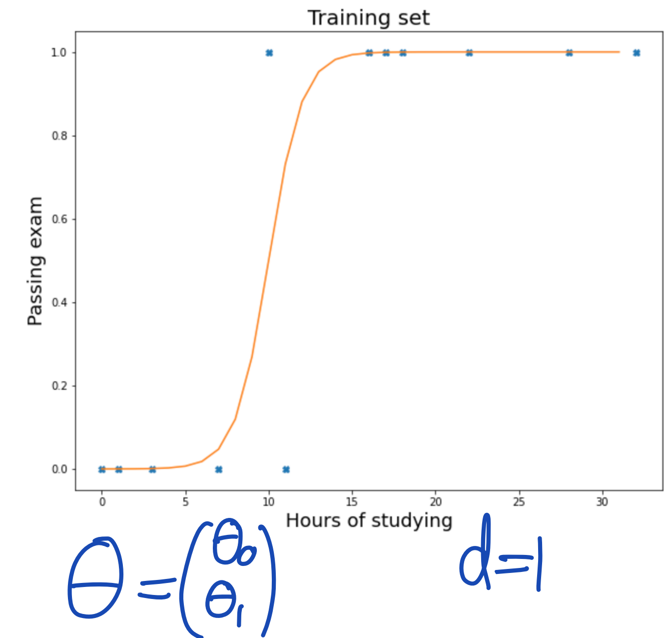


Recall: Logistic regression

Logistic regression

Hypothesis $h_\theta(x) = S(\theta^T x) = \dfrac{1}{1+e^{-\theta^T x}}$

Example: Binary classifier (Pass an exam Yes/No)



We learned the correct parameters by maximizing the log-likelihood (by using gradient ascent)

$$\ell(\theta) = \log L(\theta) = \sum_{j=1}^{n} y^{(j)} \log h_\theta(x^{(j)}) + (1 - y^{(j)}) \log(1 - h_\theta(x^{(j)}))$$

Or mizimizing the negative of the log-likelihood (= cost function)
(by using gradient descent)

Rewriting logistic regression as a neural network

OLD

$$h_\theta(x) = S(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}$$

linear part

$X = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \longrightarrow w^T x + b \longrightarrow S(w^T x + b)$

weight $w \in \mathbb{R}^n$

bias $b \in \mathbb{R}$, θ_0

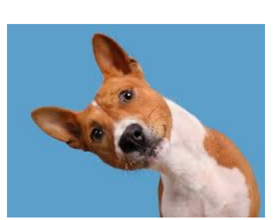
$S : \mathbb{R} \to \mathbb{R}$ activation function

$\begin{pmatrix} \theta_1 \\ \vdots \\ \theta_n \end{pmatrix}$ $\theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n + \theta_0$

$X \to$ ($w^T x + b = z$ | S) $\to a = S(z)$

Motivation: Image classification

Goal: Check if a picture contains a cat



100

100

Pixel ~ 3 colors

$n = 100 \cdot 100 \cdot 3$

$x_1, \ldots, x_n$

$w^T x + b$ | σ

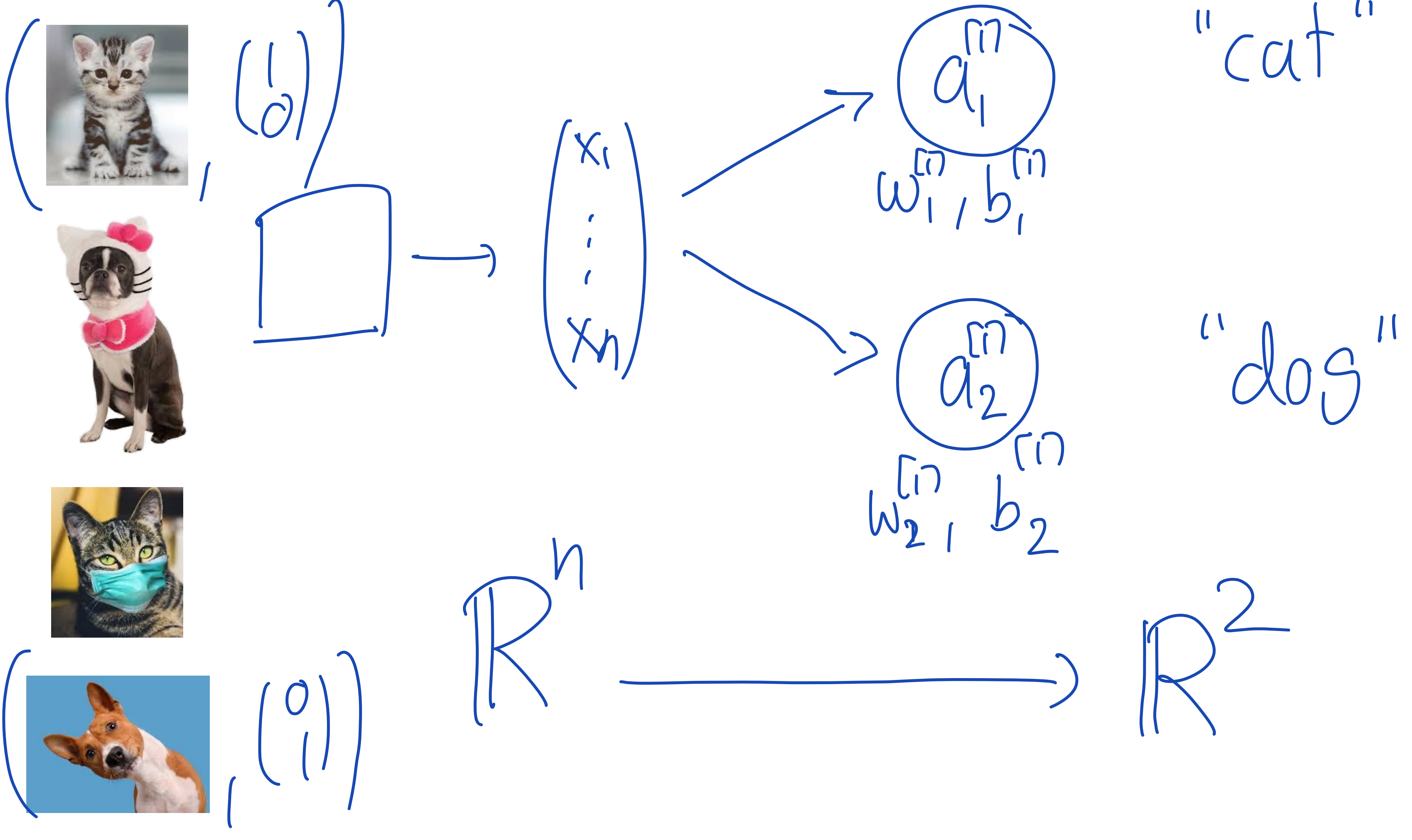
$\sigma(z_1^{[1]}) = a_1^{[1]}$

[1] — layer

1 — neuron in this layer

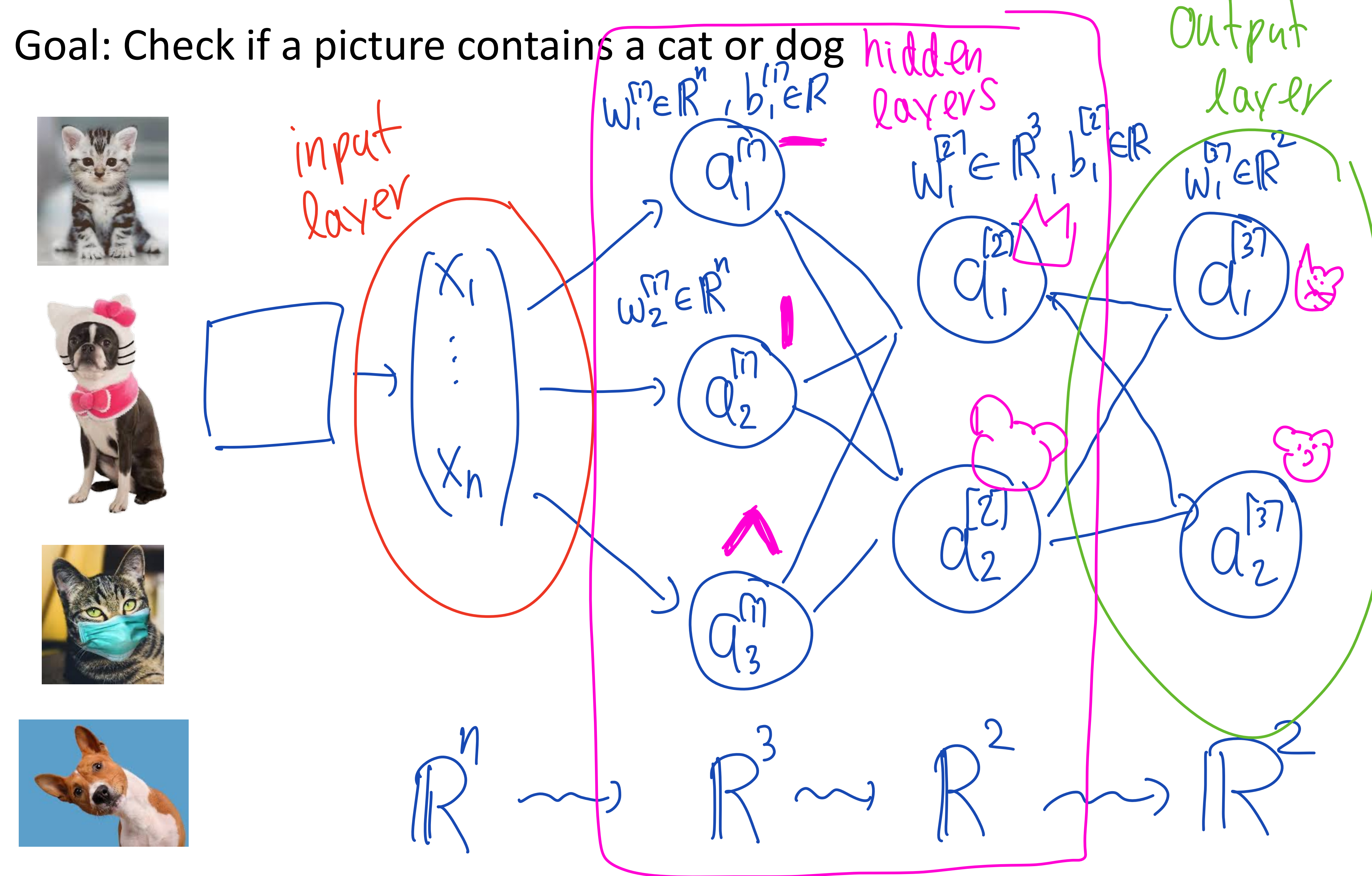
Motivation: Image classification

Goal: Check if a picture contains a cat or dog



Motivation: Image classification, More layers

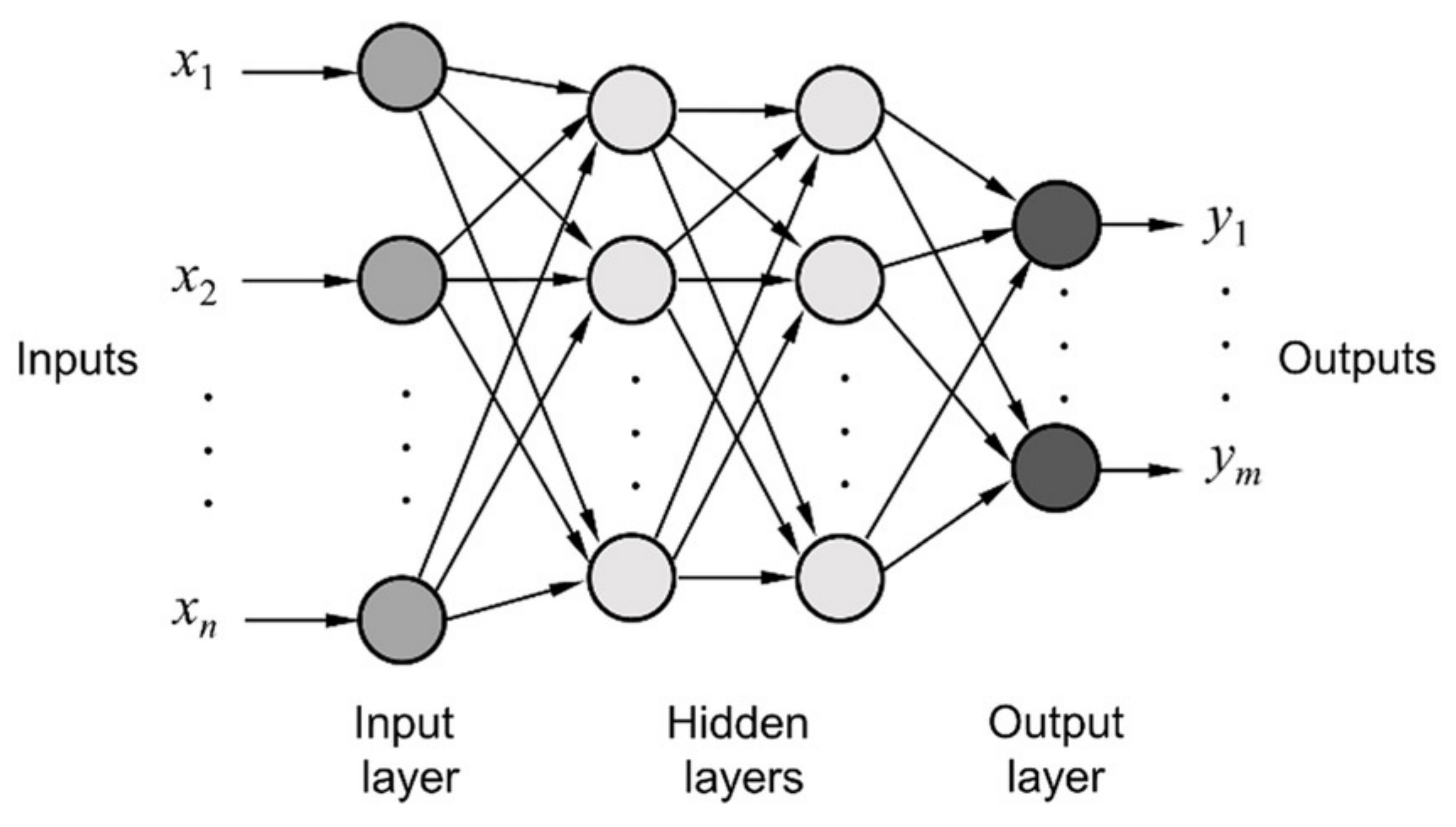
Goal: Check if a picture contains a cat or dog



(rough) Notation

An **activation function** is (in most of the cases) a (often non-linear) function $\mathbb{R} \to \mathbb{R}$.

A **neuron** consists of a weight vector $w \in \mathbb{R}^n$, a bias $b \in \mathbb{R}$ and an activation function $\sigma : \mathbb{R} \to \mathbb{R}$.



Content of a layer

If we have r layers, then for $i = 1, \ldots, r$ we have a **weight matrix** $W^{[i]} \in \mathbb{R}^{m \times n}$, a **bias vector** $b^{[i]} \in \mathbb{R}^m$ and an activation function $\sigma^{[i]}$.

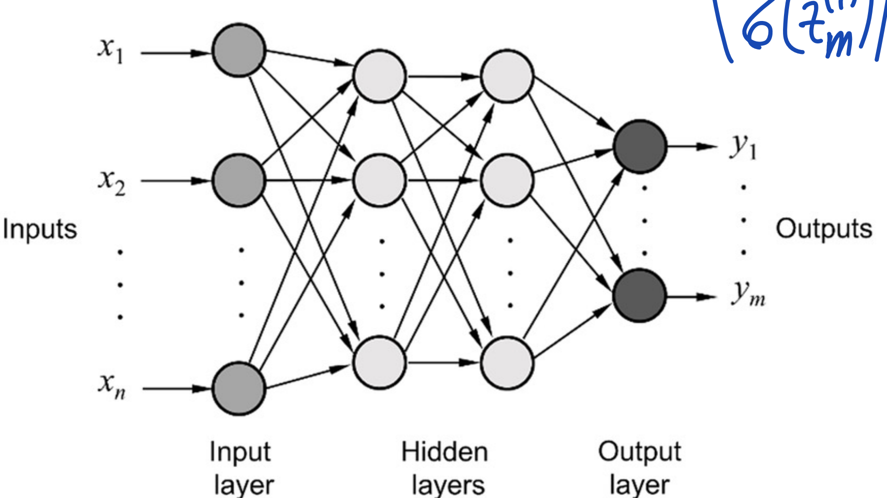
n: the number of inputs of this layer

m: number of neurons in this layer

The linear part of this layer is given by

$$z^{[i]} = W^{[i]} x + b^{[i]},$$

where $x = a^{[i-1]} \in \mathbb{R}^n$ is the output from the previous layer.



Example of activation functions

There are several common activation functions.

https://en.wikipedia.org/wiki/Activation_function

en.wikipedia.org/wiki/Activation_function

autoencoders.

The following table compares the properties of several activation functions that are functions of one fold x from the previous layer or layers:

Name	Plot	Function, $f(x)$	Derivative of f, $f'(x)$	Range	Order of continuity	Mono-tonic
Identity		x	1	$(-\infty, \infty)$	C^∞	Yes
Binary step		$\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$\begin{cases} 0 & \text{if } x \neq 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$\{0, 1\}$	C^{-1}	Yes
Logistic, sigmoid, or soft step		$\sigma(x) = \frac{1}{1+e^{-x}}$ [1]	$f(x)(1 - f(x))$	$(0, 1)$	C^∞	Yes
tanh		$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$1 - f(x)^2$	$(-1, 1)$	C^∞	Yes
Rectified linear unit (ReLU)[11]		$\begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} = \max\{0, x\} = x\mathbf{1}_{x>0}$	$\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$[0, \infty)$	C^0	Yes
Gaussian error linear unit (GELU)[6]		$\frac{1}{2}x\left(1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right) = x\Phi(x)$	$\Phi(x) + x\phi(x)$	$(-0.17\ldots, \infty)$	C^∞	No
Softplus[12]		$\ln(1 + e^x)$	$\frac{1}{1+e^{-x}}$	$(0, \infty)$	C^∞	Yes

The **Rectified linear unit (ReLU)** function is defined by

$$\text{ReLU}(x) = \max\{0, x\}.$$

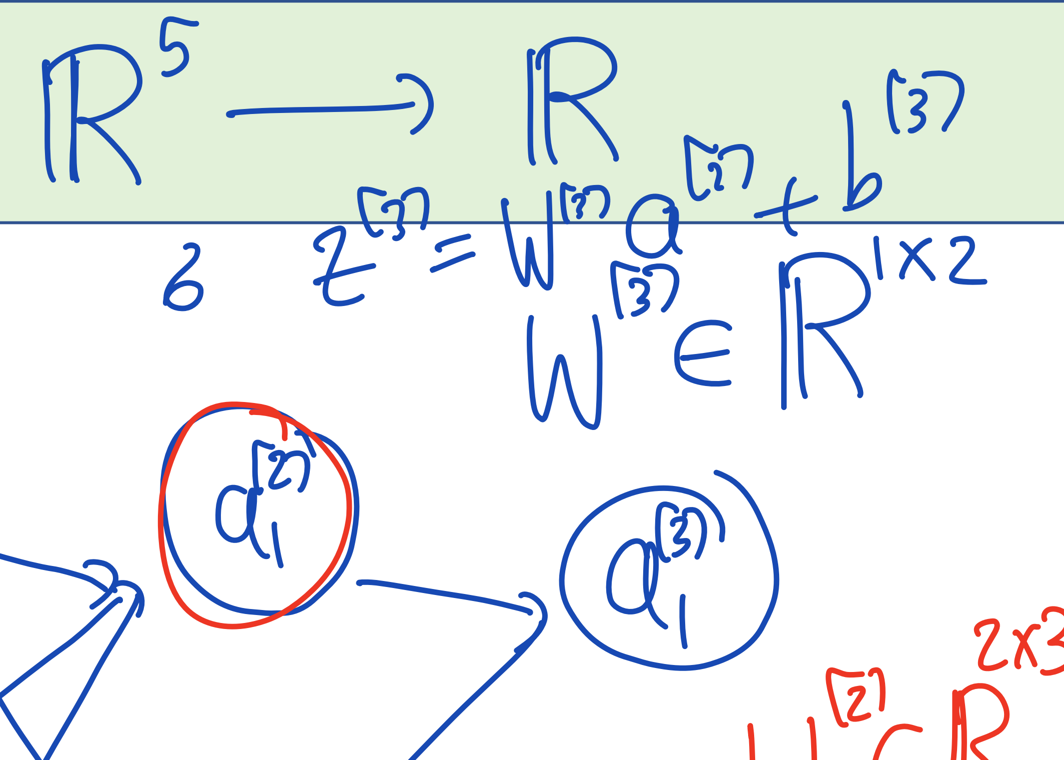
The **sigmoid** function is defined by

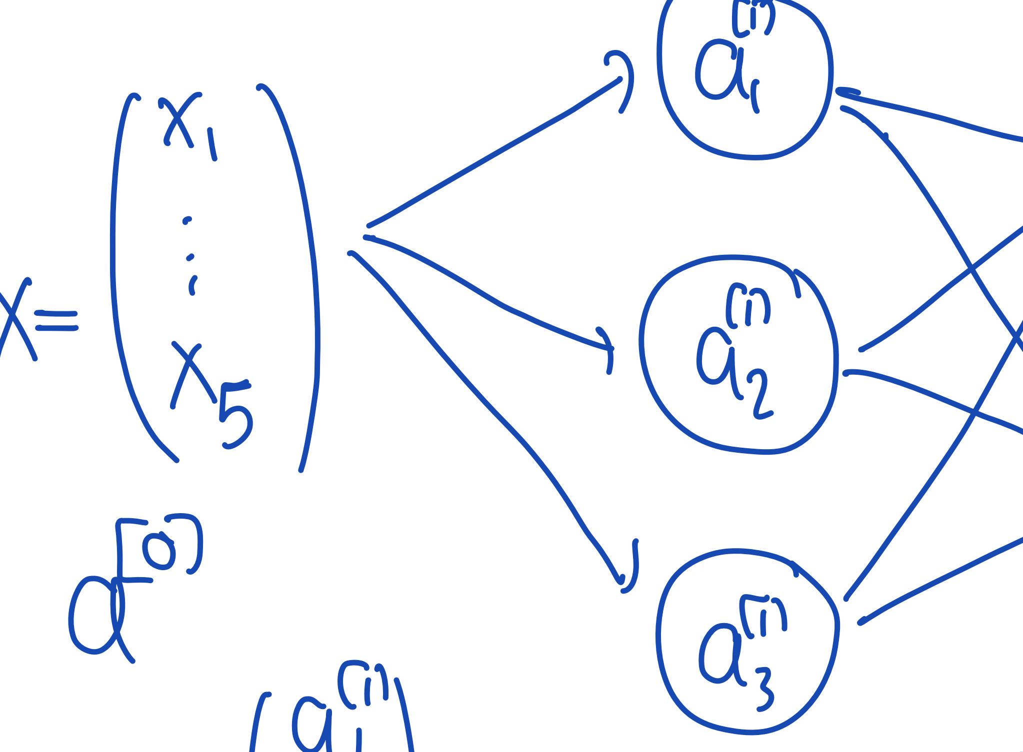
$$S(x) = \frac{1}{1 - e^{-x}}.$$

Question: Why not use the identity map as an activation function?

Example of NN

Let us consider a NN with 3 layers (2 hidden):

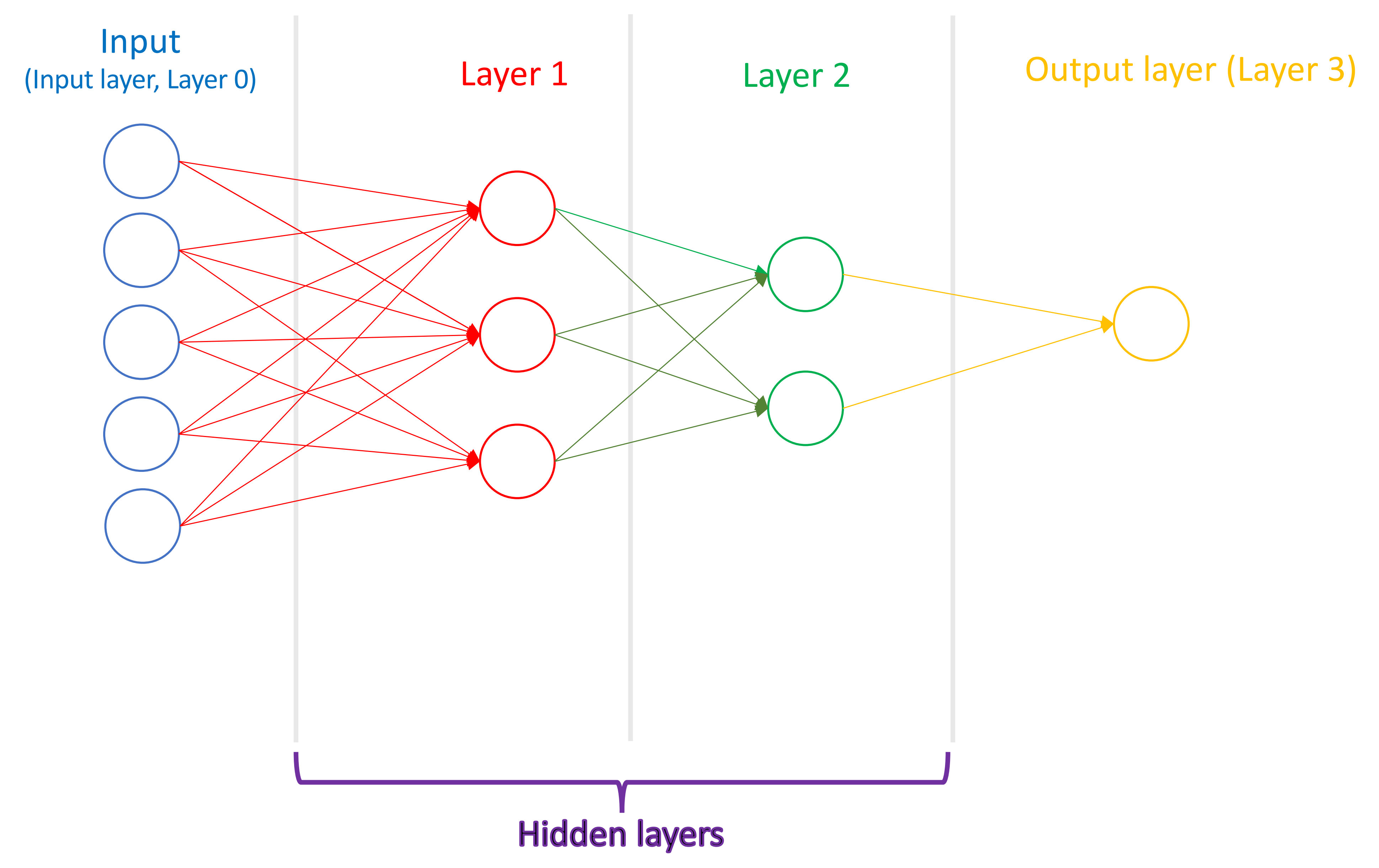




Example of NN in Python

https://colab.research.google.com/drive/1U6banZAzvrmBj3Rd-7JgtmhsIY8Si9aI?usp=sharing

Neural Network with 4 layers



Activation function

Definition 4.1. *(i) An* **activation function** *is a (usually non-linear) function $\sigma : \mathbb{R}^n \to \mathbb{R}^n$. Often activation functions are given as functions $\sigma : \mathbb{R} \to \mathbb{R}$, which are then extended to a function $\sigma : \mathbb{R}^n \to \mathbb{R}^n$ by setting*

$$\sigma \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} := \begin{pmatrix} \sigma(x_1) \\ \vdots \\ \sigma(x_n) \end{pmatrix}.$$

(ii) The **rectified linear unit (ReLU)** *is the activation function defined for $x \in \mathbb{R}$ by*

$$\mathrm{ReLU}(x) = \max\{0, x\}.$$

(iii) The **sigmoid** *function is the activation function defined for $x \in \mathbb{R}$ by*

$$S(x) = \frac{1}{1 - e^{-x}}.$$

(iv) The **softmax** *function is given*

$$\mathrm{softmax} : \mathbb{R}^n \to \mathbb{R}^n,$$

$$\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \longmapsto \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix},$$

where $y_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$ for $i = 1, \ldots, n$.

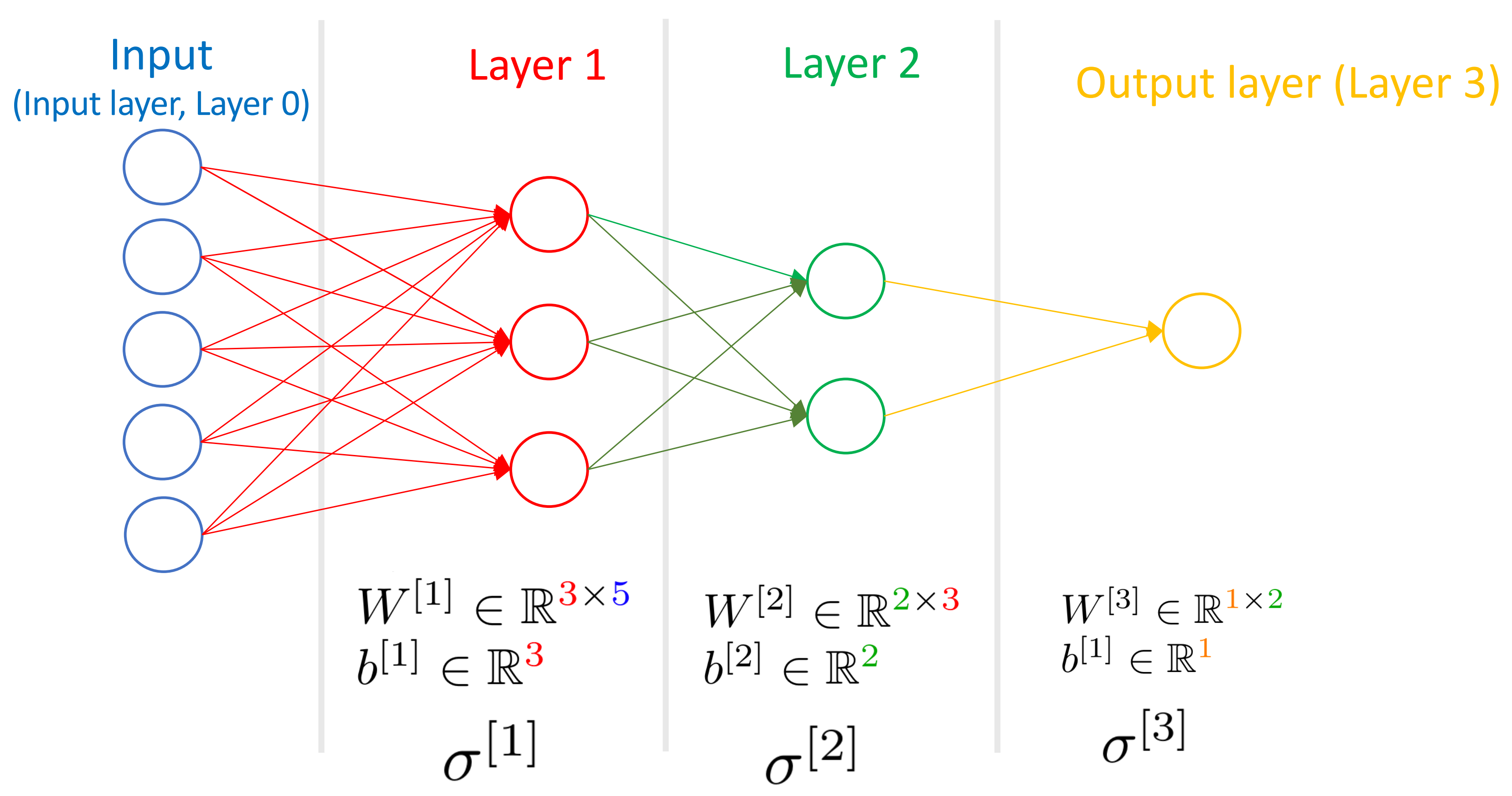
Layer

Definition 4.2. *A* **layer** $L = (W, b, \sigma)$ *of input size* $n \geq 1$ *and (output)* **size** $m \geq 1$ *consists of*

(i) *a* **weight matrix** $W \in \mathbb{R}^{m \times n}$,

(ii) *a* **bias vector** $b \in \mathbb{R}^{m}$,

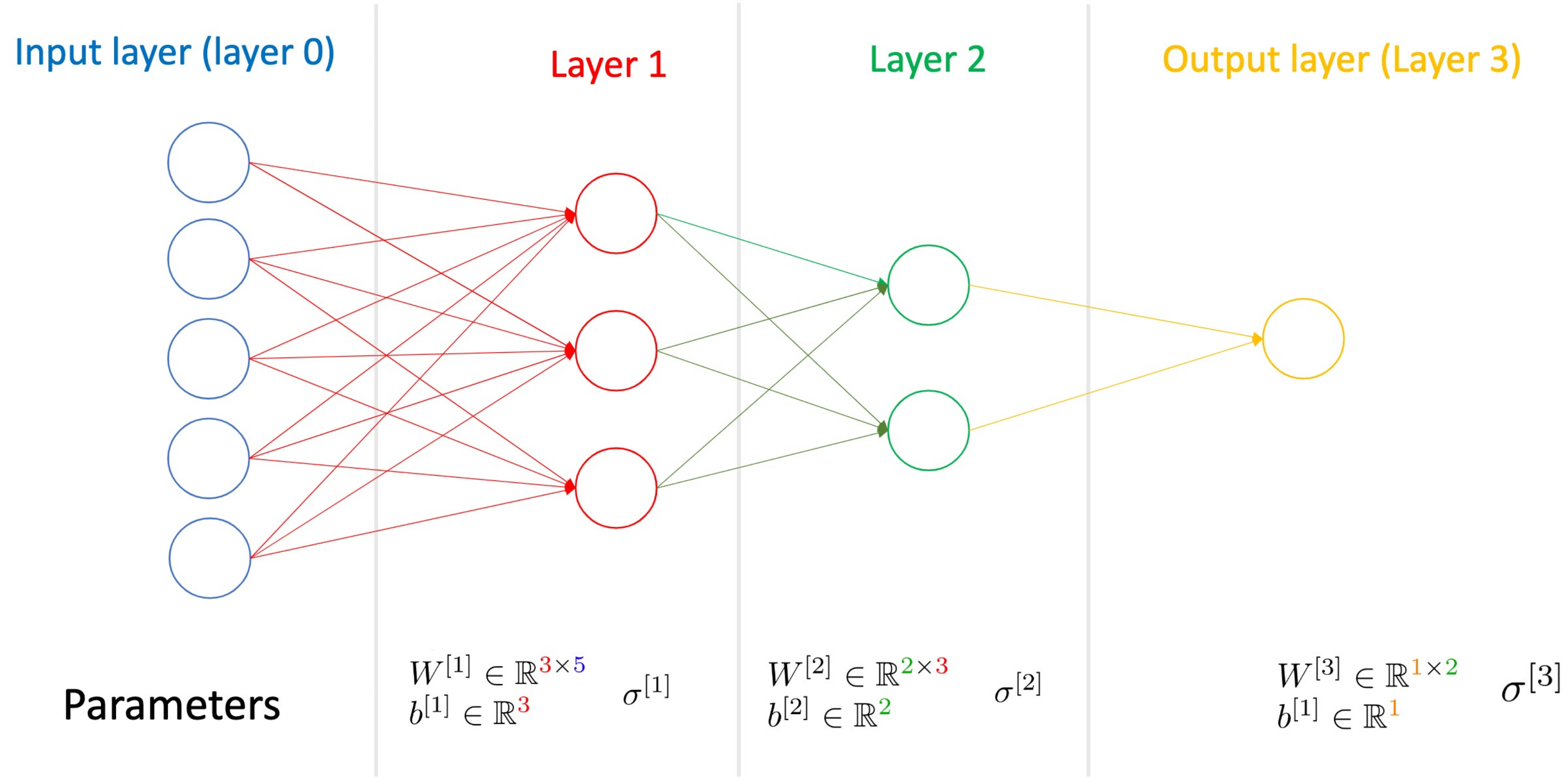
(iii) *and an* **activation function** $\sigma : \mathbb{R}^m \to \mathbb{R}^m$.



r-layer Neural Network

Definition 4.3. *For $r \geq 1$ a **r-layer neural network** $N = (L^{[1]}, \ldots, L^{[r]})$ of **input size** n and **output size** m, consists of a collection of layers $L^{[i]} = (W^{[i]}, b^{[i]}, \sigma^{[i]})$ for $i = 1, \ldots, r$, such that $W^{[i]} \in \mathbb{R}^{m_i \times m_{i-1}}$ with $m_0 = n$ and $m_r = m$.*

The following shows an example of a 3-layer neural network with input size 5 and output size 1 and $m_0 = 5, m_1 = 3, m_2 = 2$, and $m_3 = 1$:



Neural Network: forward pass

Definition 4.4. *Let $N = (L^{[1]}, \dots, L^{[r]})$ be a r-layer neural network of input size n and output size m. We want to view it as a function $N : \mathbb{R}^n \to \mathbb{R}^m$ by defining $N(x)$ for an* **input** *$x \in \mathbb{R}^n$ by the output of its last layer $N(x) = a^{[r]}$. Here we define for $i = 1, \dots, r$ the following:*

(i) The **linear part** *of the layer $L^{[i]}$ is defined by*

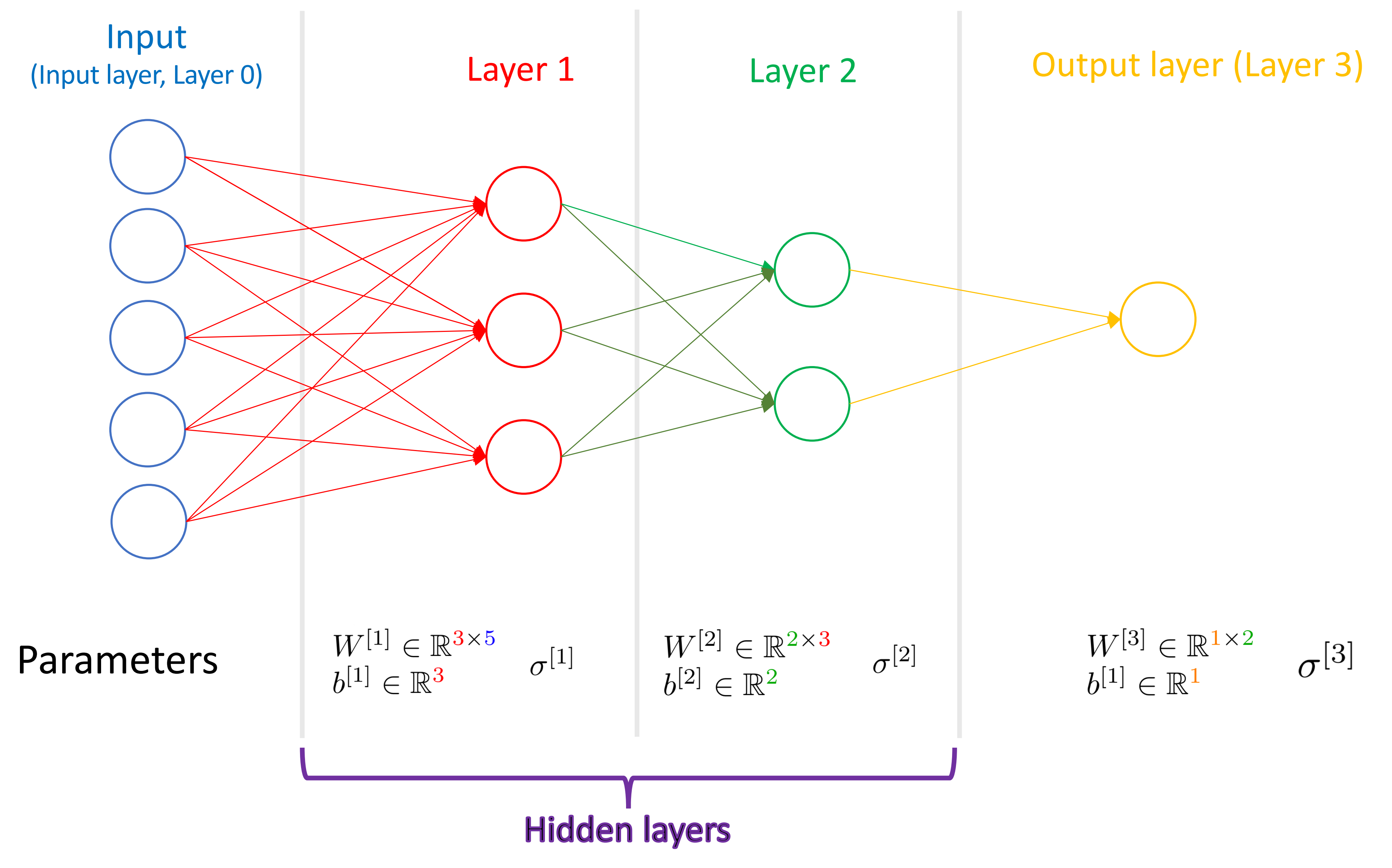
$$z^{[i]} = W^{[i]} a^{[i-1]} + b^{[i]},$$

where $a^{[i-1]} \in \mathbb{R}^{m_{i-1}}$ is the output from the previous layer. In the case $i = 1$ we set $a^{[0]} = x$.

(ii) The **output** *of the layer $L^{[i]}$ is defined by applying the activation function to the linear part, i.e.*

$$a^{[i]} = \sigma^{[i]}(z^{[i]}) \in \mathbb{R}^{m_i}$$

Neural Network with 3 layers



Training set

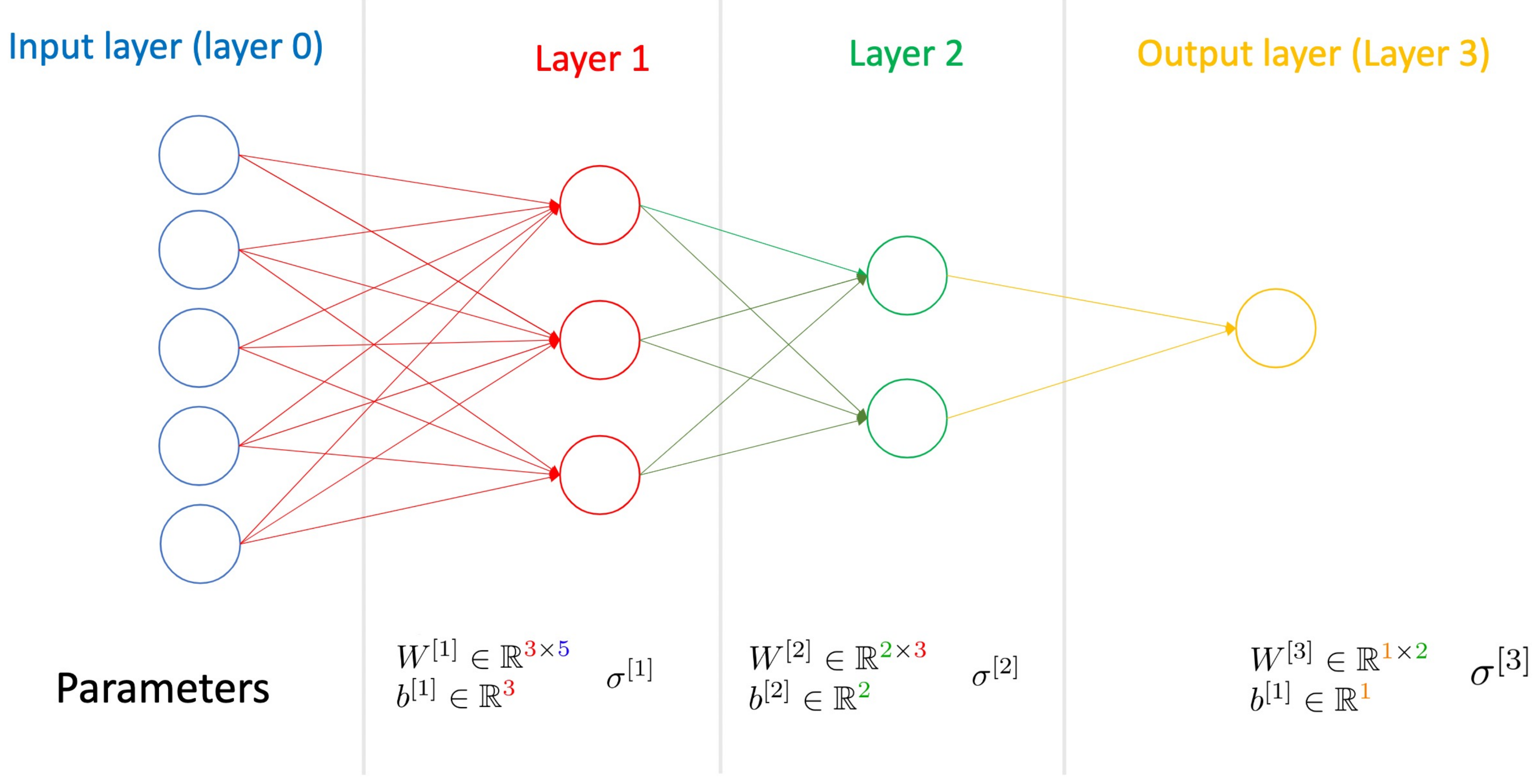
Our neural network is supposed to approximate a function $f : \mathbb{R}^n \to \mathbb{R}^m$(In the example $n = 5$ and $m = 1$). So we want to find the best possible choices of paramters (i.e. weight matrices and biases) such that $N(x)$ is a good approximation for $f(x)$.

- Input values (Feature space): $\mathcal{X} = \mathbb{R}^n$
- Output value (Label space): $\mathcal{Y} = \mathbb{R}^m$
- Training example: $(x, y) \in \mathcal{X} \times \mathcal{Y}$.
- Training set (with t training examples): $\mathcal{T} = \big((x^{(1)}, y^{(1)}), \ldots, (x^{(t)}, y^{(t)})\big) \in (\mathcal{X} \times \mathcal{Y})^t$.

Goal: Find a neural network N, such that N(x) is a good approximation for a given training set.

Strategy: Define a cost function and minimize it

Cost function in our example



An example of a neural network with $(3 \cdot 5 + 3) + (2 \cdot 3 + 2) + (1 \cdot 2 + 1) = 29$ parameters.

In our example we have 29 parameters, i.e. a cost function $J : \mathbb{R}^{29} \to \mathbb{R}$, which we want to minimize.

Recall: Cost functions

Given a training set $\mathcal{T}$, a **cost function** is a map from the space of parameters to $\mathbb{R}$, which measures how good the current parameters are with respect to the training set.

For linear regression we used the sum of squares:

$$J(\theta) = \frac{1}{2}\sum_{j=1}^{n}(h_\theta(x^{(j)}) - y^{(j)})^2 \,.$$

For logistic regression we used the log likelihood (logistic cost function)

$$J(\theta) = -\sum_{j=1}^{n} y^{(j)} \log h_\theta(x^{(j)}) + (1 - y^{(j)}) \log(1 - h_\theta(x^{(j)}))$$

Recall: Gradient descent

Gradient descent algorithm (rough version).

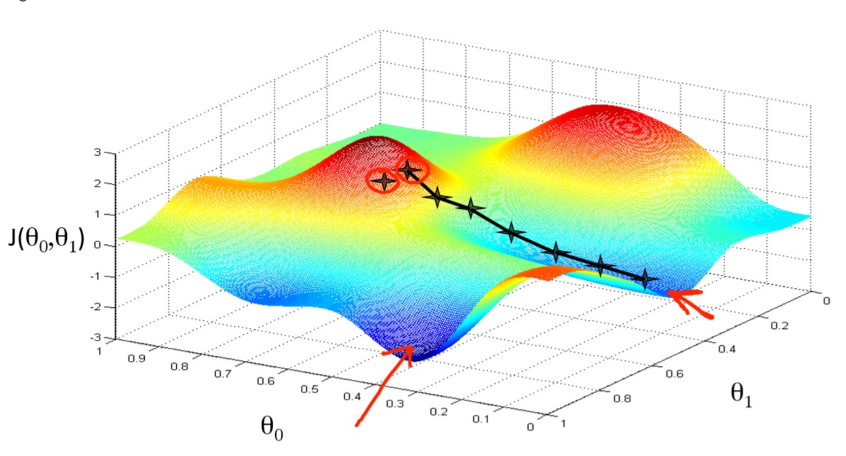
i) Start with a random starting value for the parameters, e.g. $\theta = 0 = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}$.

ii) Change the parameters in the opposite direction of the steepest ascent, i.e. opposite direction of the gradient. This means we want to subtract the gradient from the current parameters, weighted by a factor $\alpha \in \mathbb{R}$, the **learning rate**.

The new parameters θ are therefore given by:

$$\theta := \theta - \alpha \nabla J(\theta).$$

iii) Repeat step ii) until the value $J(\theta)$ does not change anymore.



Gradient descent in our example

In our example we have 29 parameters, i.e. a cost function $J : \mathbb{R}^{29} \to \mathbb{R}$, which we want to minimize.

We need to calculate the **gradient of** J

$$\nabla J = \begin{pmatrix} \frac{\partial}{\partial \theta_1} J \\ \frac{\partial}{\partial \theta_2} J \\ \vdots \\ \frac{\partial}{\partial \theta_{29}} J \end{pmatrix}$$

Main idea (**Backpropagation**): Calculate the gradient layer by layer using the chain rule