# Deep learning and Artificial Neural Network for Object recognition

Dr. Anthony Fleury - HDR
IMT Nord Europe

Dr. Sebastien Ambellouis
Ingénieur de recherche Université Gustave Eiffel
Associé à IMT Nord Europe

*sebastien.ambellouis@univ-eiffel.fr*
*anthony.fleury@imt-nord-europe.fr*

# The perceptron

- **Supervised learning : regression/classification**
- Single neuron model
  - Linear regression
  - Logistic regression
  - Multi-output and softmax regression
- Multi-layer perception ?

# Types of machine learning

We can categorize three types of learning procedures:

1. **Supervised Learning:**

$$\mathbf{y} = f(\mathbf{x})$$

We have a labeled dataset with pairs ($\mathbf{x}$, $\mathbf{y}$), e.g. classify a signal window as containing speech or not:

$\mathbf{x}_1 = [x(1), x(2), \ldots, x(T)]$ $\mathbf{y}_1$ = "no"
$\mathbf{x}_2 = [x(T+1), \ldots, x(2T)]$ $\mathbf{y}_2$ = "yes"
$\mathbf{x}_3 = [x(2T+1), \ldots, x(3T)]$ $\mathbf{y}_3$ = "yes"
...

2. Unsupervised Learning:

$$f(\mathbf{x})$$

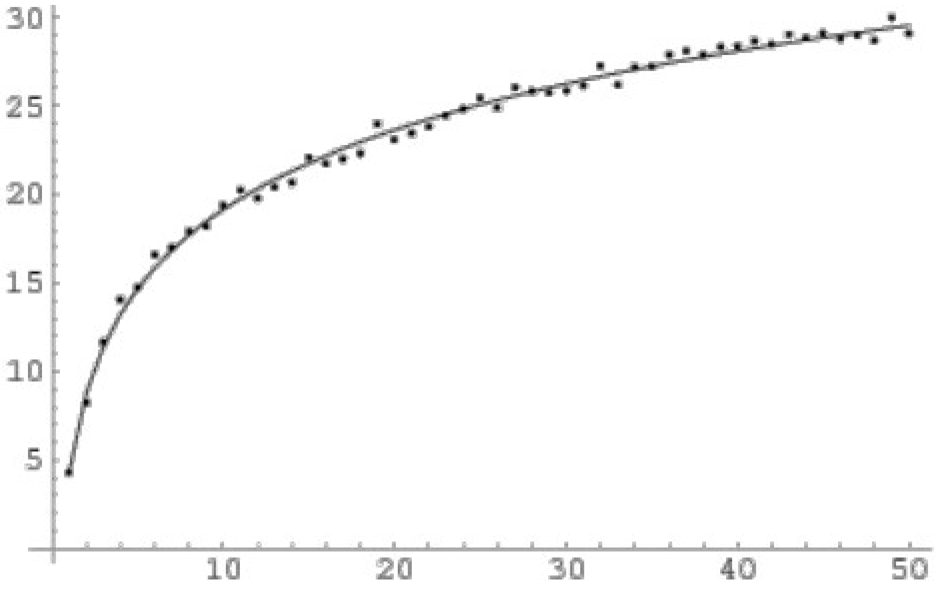
3. Reinforcement Learning:

$$\mathbf{y} = f(\mathbf{x})$$

$$\mathbf{z}$$

# Supervised learning

Fit a function: $\mathbf{y} = f(\mathbf{x}), \quad \mathbf{x} \in \mathbb{R}^m$

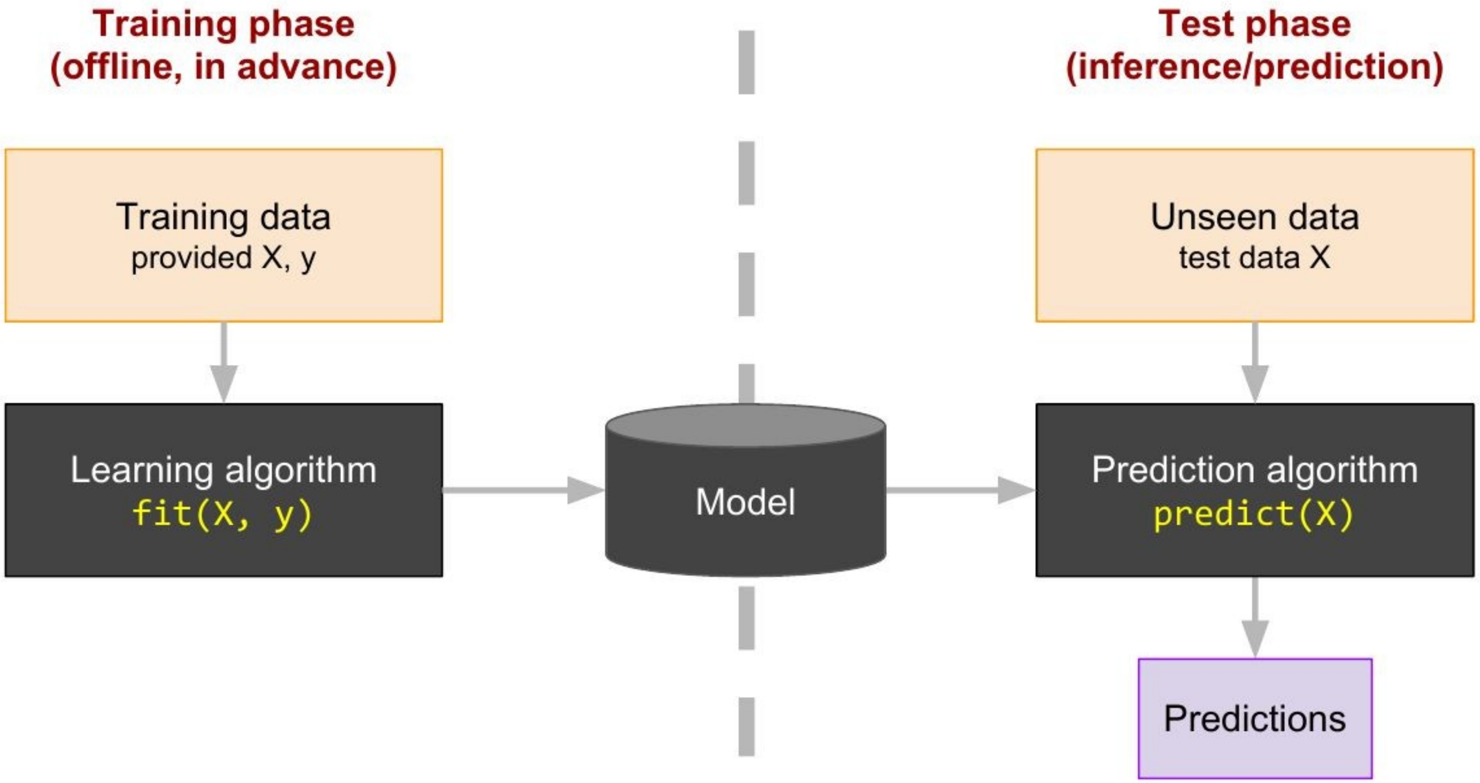


# Supervised learning

Fit a function: $\mathbf{y} = f(\mathbf{x}), \quad \mathbf{x} \in \mathbb{R}^m$

Given paired training examples $\{(\mathbf{x}_i, \mathbf{y}_i)\}$

# Black box abstraction of supervised learning



# Supervised learning

Fit a function: $\mathbf{y} = f(\mathbf{x})$, $\mathbf{x} \in \mathbb{R}^m$

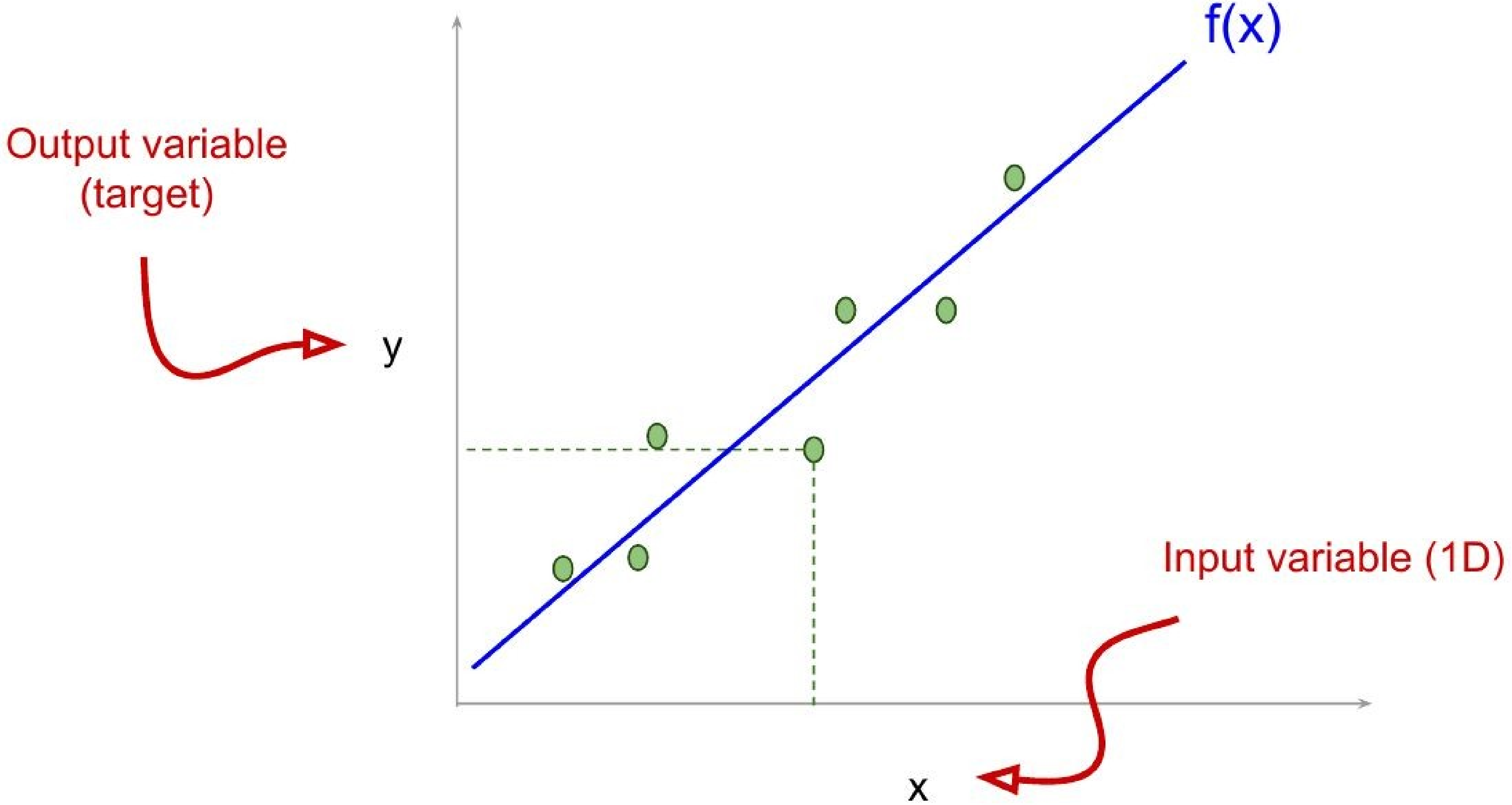
Given paired training examples $\{(\mathbf{x}_i, \mathbf{y}_i)\}$

Key point: **generalize well to unseen examples**

Depending on the type of target $\mathbf{y}$ we get:

- Regression: $\mathbf{y} \in \mathbb{R}^N$ is continuous (e.g. temperatures $\mathbf{y} = \{19°, 23°, 22°\}$)
- Classification: $\mathbf{y}$ is discrete (e.g. $\mathbf{y} = \{1, 2, 5, 2, 2\}$).
  - Beware! These are unordered categories, not numerically meaningful outputs: e.g. code[1] = “dog”, code[2] = “cat”, code[5] = “ostrich”, ...

# Linear Regression (1D input – 1D output)



# Linear Regression (Multi-D input)

Input data can also be M-dimensional with vector **x**:

$$\boxed{y = \mathbf{w}^T \cdot \mathbf{x} + b} = \ w1 \cdot x1 + w2 \cdot x2 + w3 \cdot x3 + \ldots + wM \cdot xM + b$$

e.g. we want to predict the **price of a house (y)** based on:

x1 = square-meters (sqm)

x2,3 = location (lat, lon)

x4 = year of construction (yoc)

y = price = $w1 \cdot$(sqm) + $w2 \cdot$(lat) + $w3 \cdot$(lon) + $w4 \cdot$(yoc) + b

# Supervised learning

Fit a function: $\mathbf{y} = f(\mathbf{x})$, $\mathbf{x} \in \mathbb{R}^m$

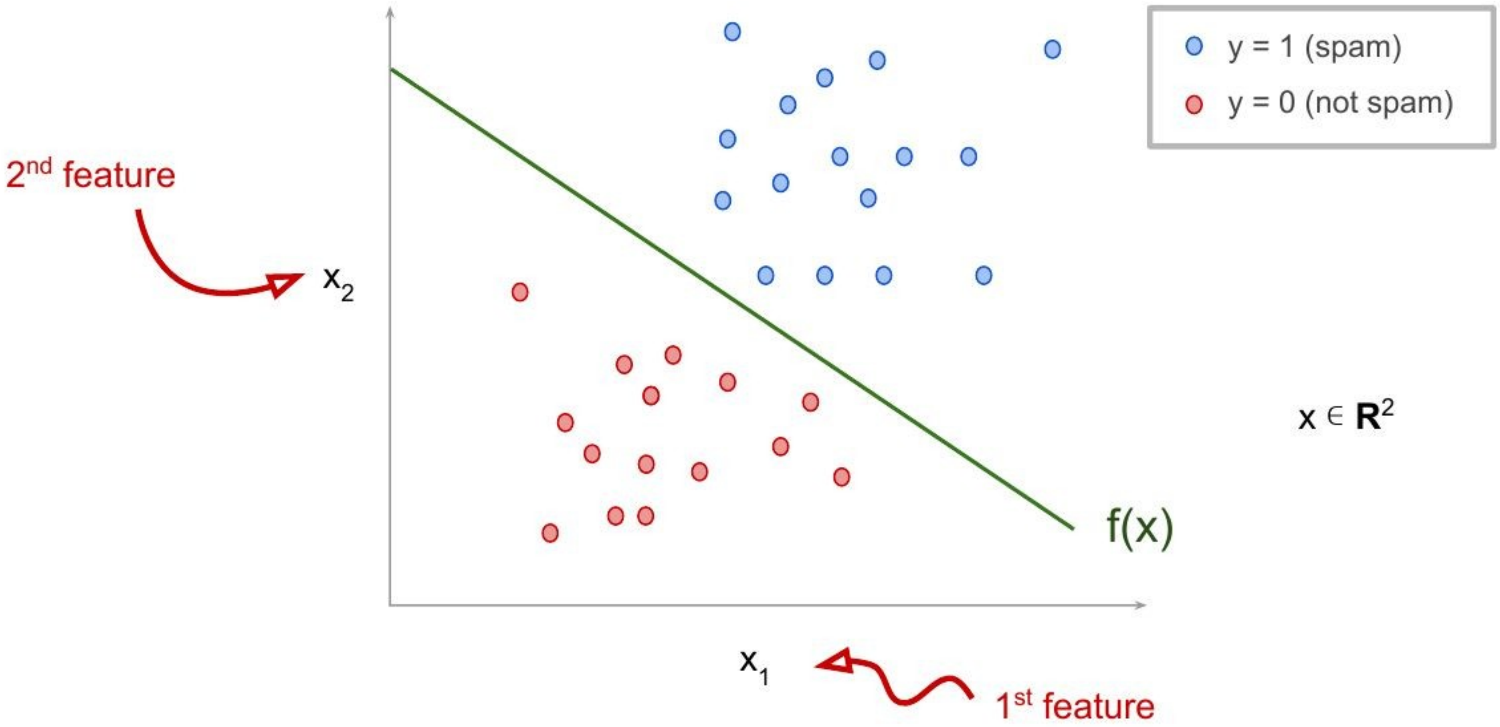
Given paired training examples $\{(\mathbf{x}_i, \mathbf{y}_i)\}$

Key point: **generalize well to unseen examples**

Depending on the type of target $\mathbf{y}$ we get:

- Regression: $\mathbf{y} \in \mathbb{R}^N$ is continuous (e.g. temperatures $\mathbf{y} = \{19°, 23°, 22°\}$)
- Classification: $\mathbf{y}$ is discrete (e.g. $\mathbf{y} = \{1, 2, 5, 2, 2\}$).
    - Beware! These are unordered categories, not numerically meaningful outputs: e.g. code[1] = “dog”, code[2] = “cat”, code[5] = “ostrich”, ...

# Classification (spam/not spam classifier)




$x \in \mathbf{R}^2$

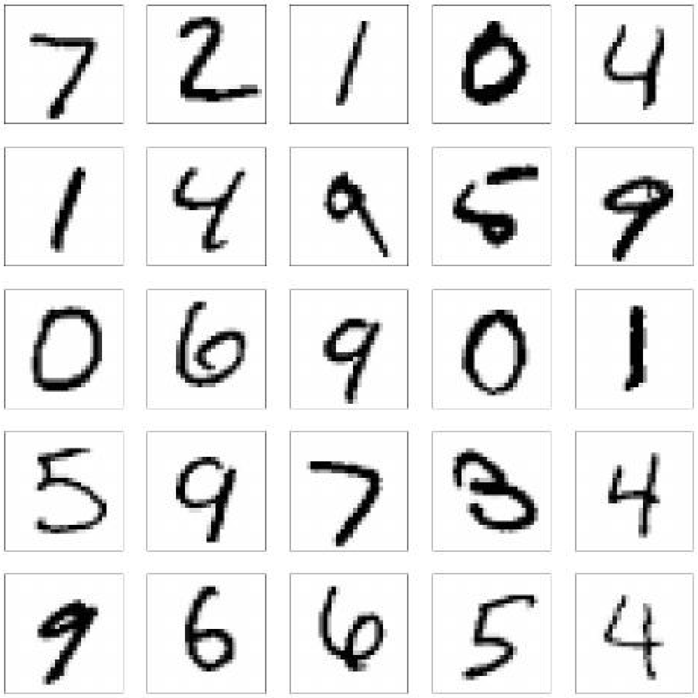
# Classification (digit classifier)

NMIST dataset

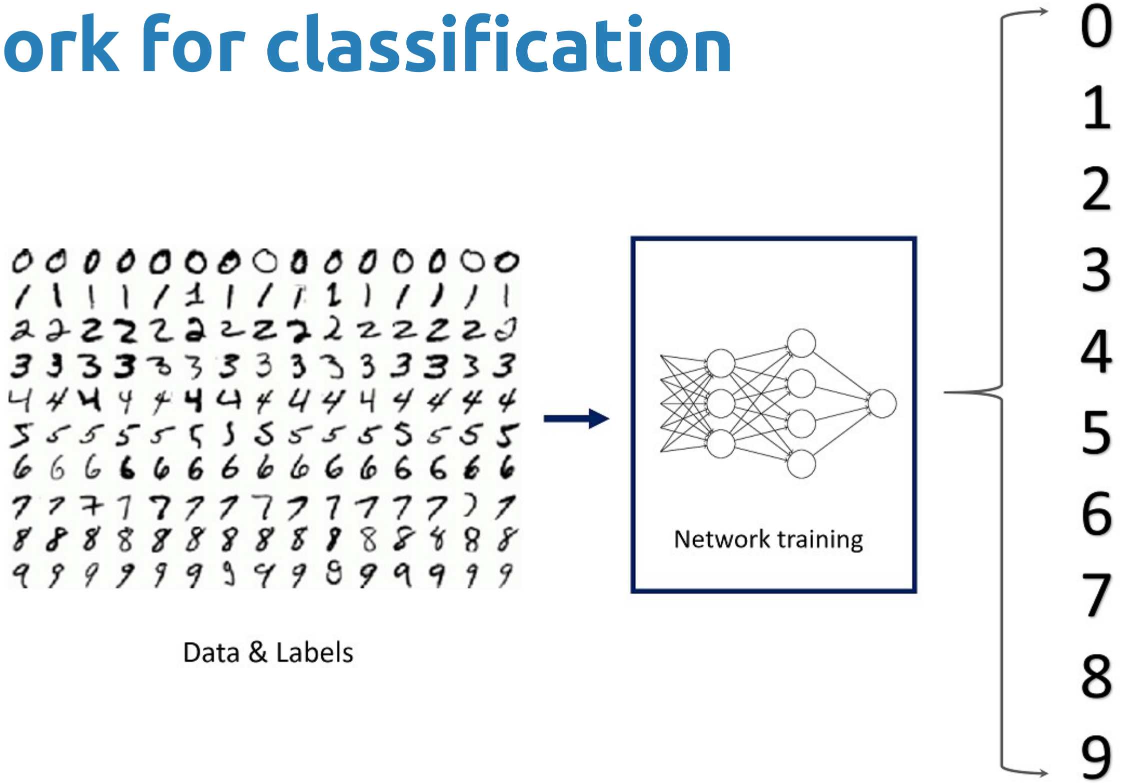
Produce a classifier to map from pixels to the digit.

- If images are grayscale and $28 \times 28$ pixels in size, then $\mathbf{x}_i \in \mathbb{R}^{784}$
- $y_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Example of a **multi-class classification** task.



# A network for classification



# The perceptron

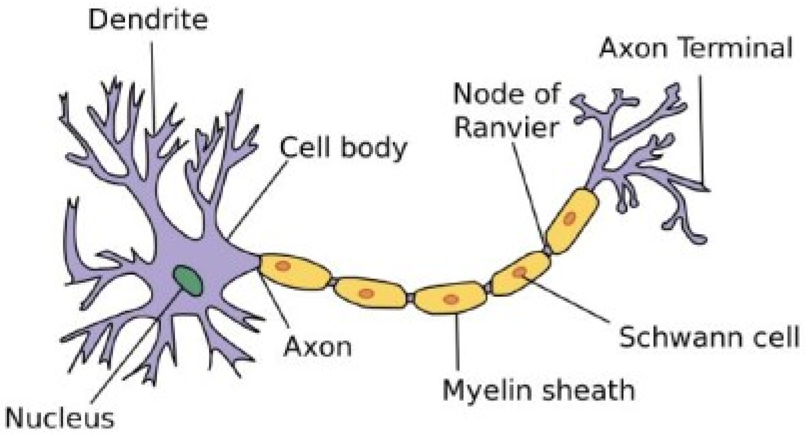
- Supervised learning : regression/classification
- **Single neuron model**
  - Linear regression
  - Logistic regression
  - Multi-output and softmax regression
- Multi-layer perception ?

# Biological inspiration

The Perceptron is seen as an **analogy** to a biological neuron.

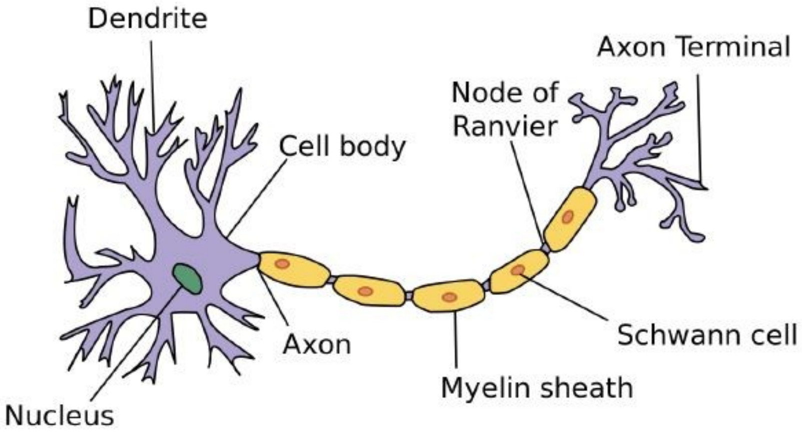
Biological neurons fire an impulse once the sum of all inputs is over a threshold.

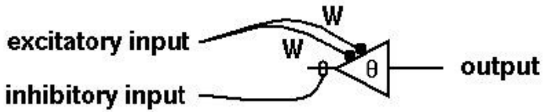
The perceptron acts like a switch (learn how in the next slides...).



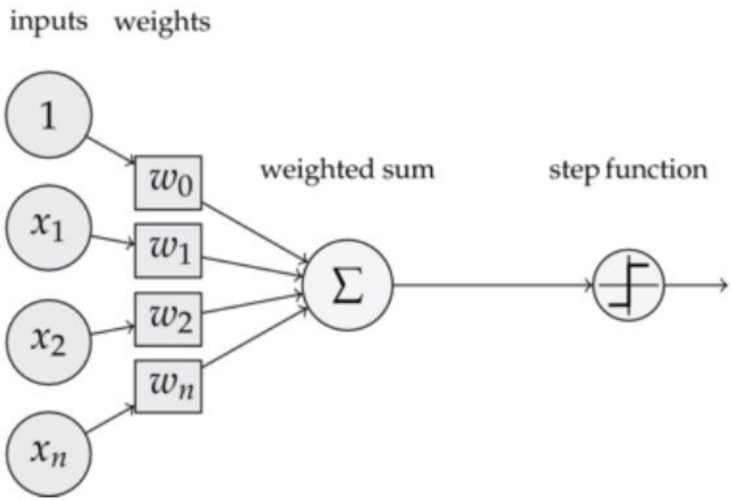

McCullough and Pitts model (1943)

# Biological inspiration




Rosenblatt's Perceptron (1958)

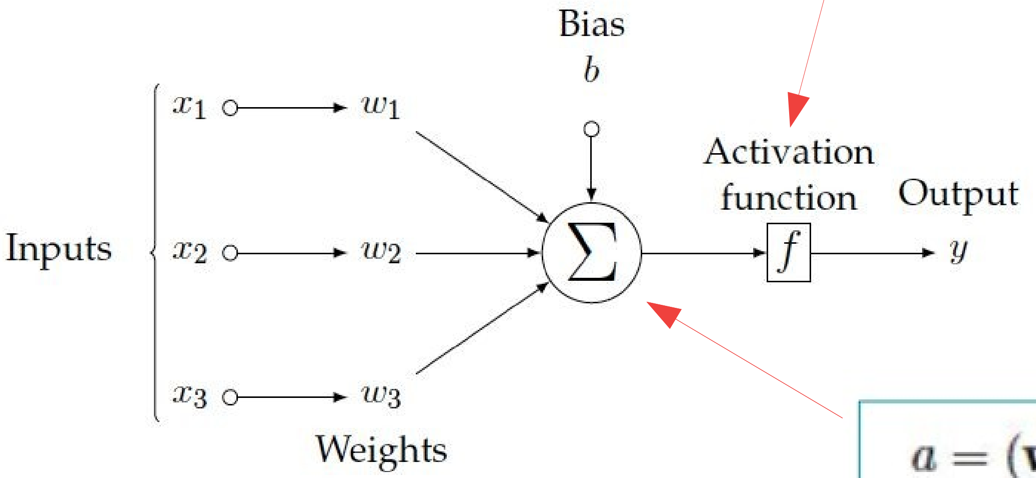


# Single neuron model (perceptron)

The perceptron can address both <u>regression</u> or <u>classification</u> problems, depending on the chosen **activation function**.

**Bias and weights define the behaviour of the perceptron**

**TRAINING**

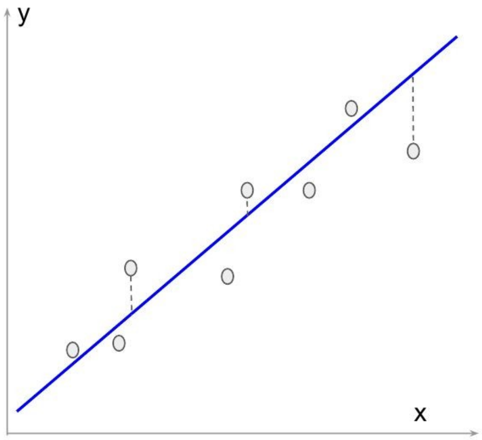



$$a = (\mathbf{w}^T \cdot \mathbf{x} + b)$$

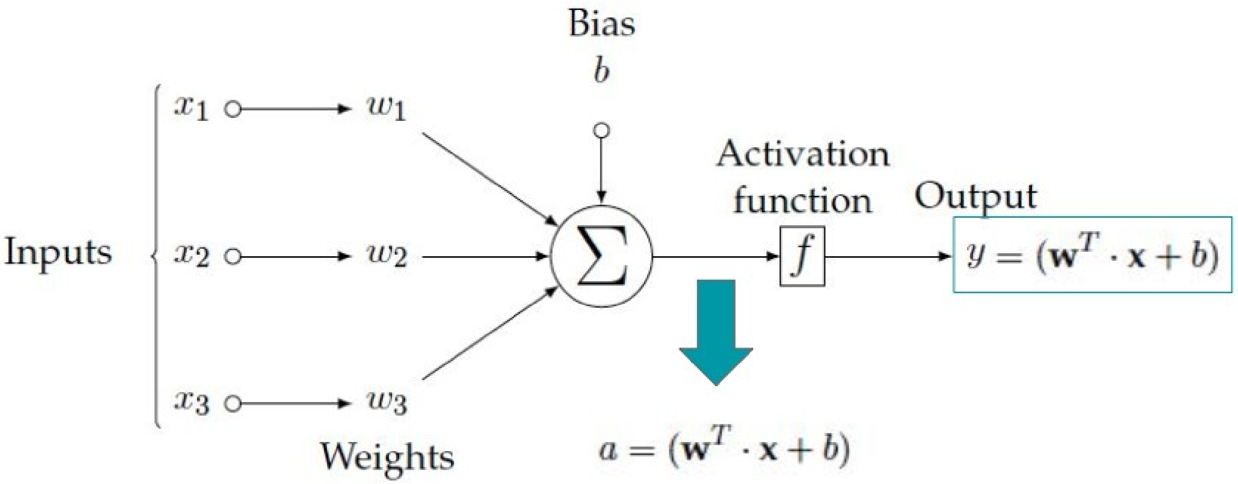
# The perceptron

- Supervised learning : regression/classification
- **Single neuron model**
  - Linear regression
  - Logistic regression
  - Multi-output and softmax regression
- Multi-layer perception ?

# Linear regression

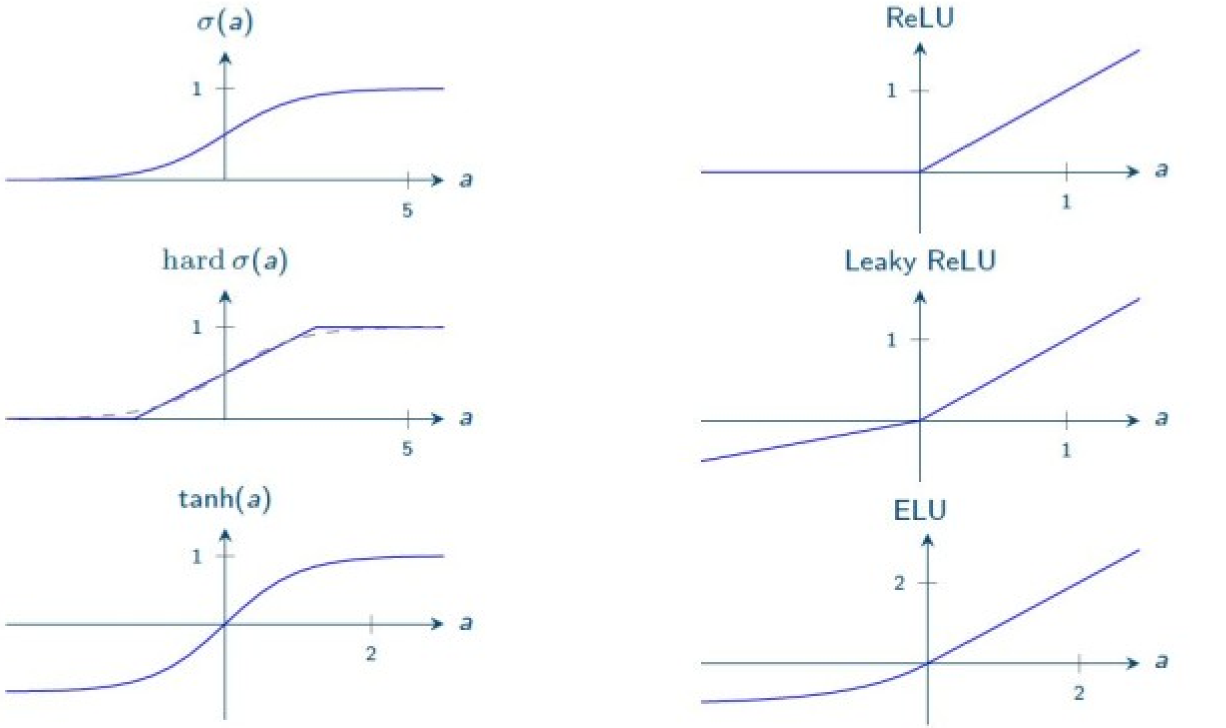



The perceptron can solve <u>linear regression</u> problems when f(a)=a. [identity]



# Other activation function

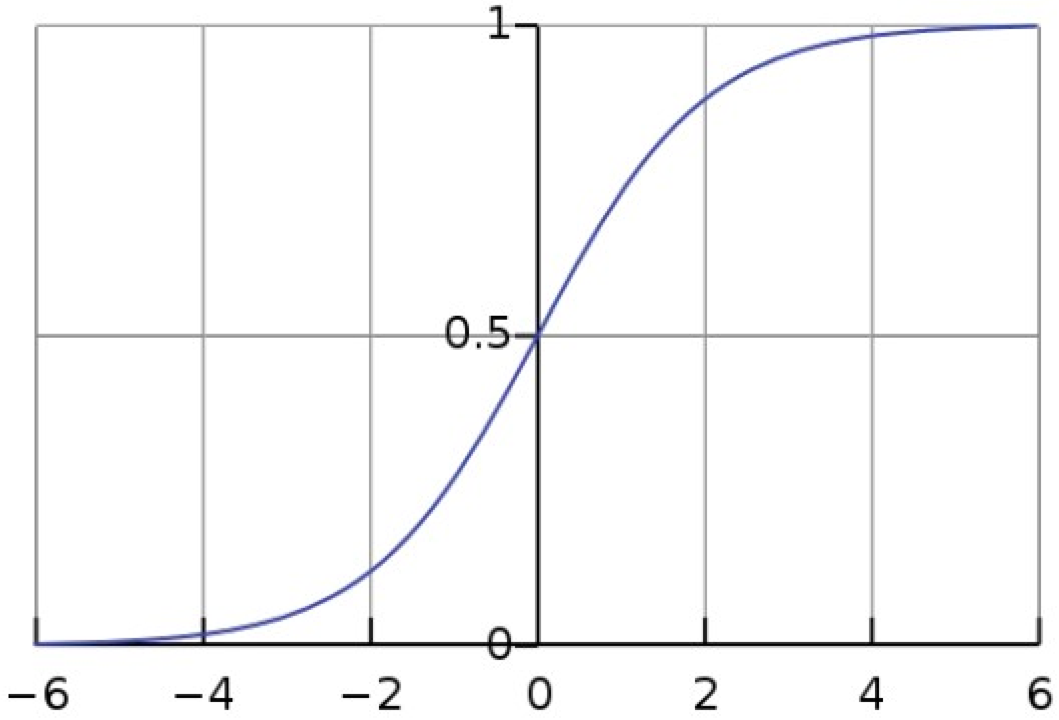
More interesting when the activation function f(a) is not the identity, but:



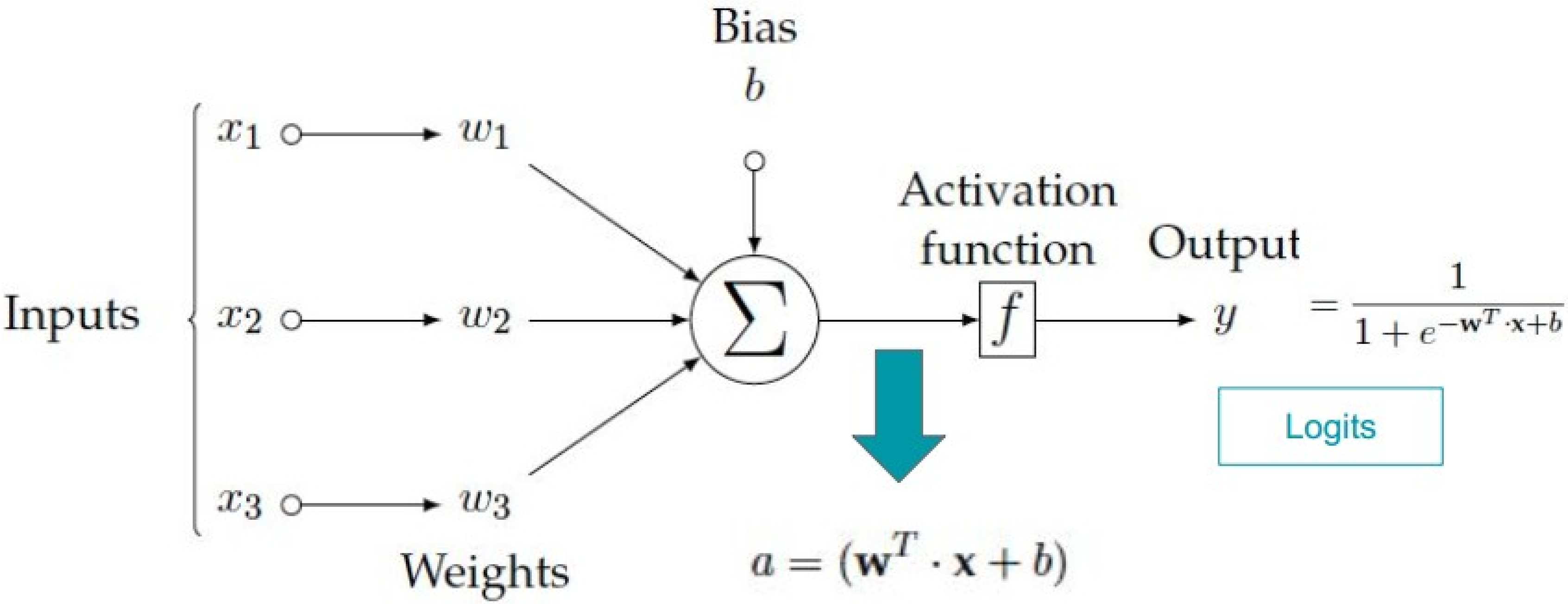
# Other activation function

The **sigmoid function σ(x)** or **logistic curve** maps any input x between [0,1]:

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



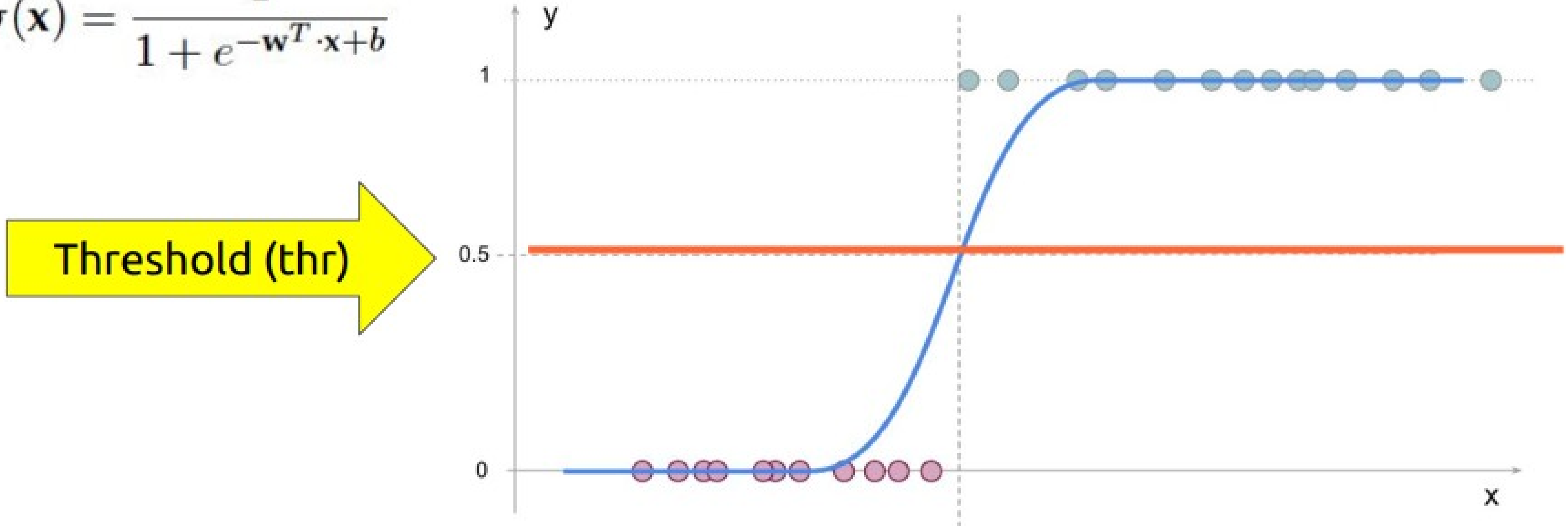
# Logistic regression



# Binary classification

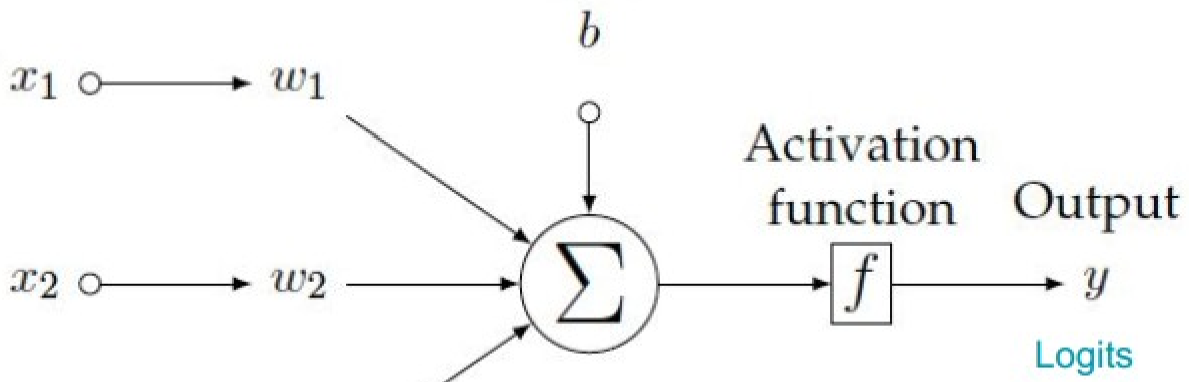
For classification, regressed values should b collapsed into 0 and 1 to quantize the confidence of the predictions ("probabilities").

$$\sigma(\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^T \cdot \mathbf{x} + b}}$$



# Binary classification

Setting a **threshold (thr)** at the output of the perceptron allows solving classification problems between two classes (binary):



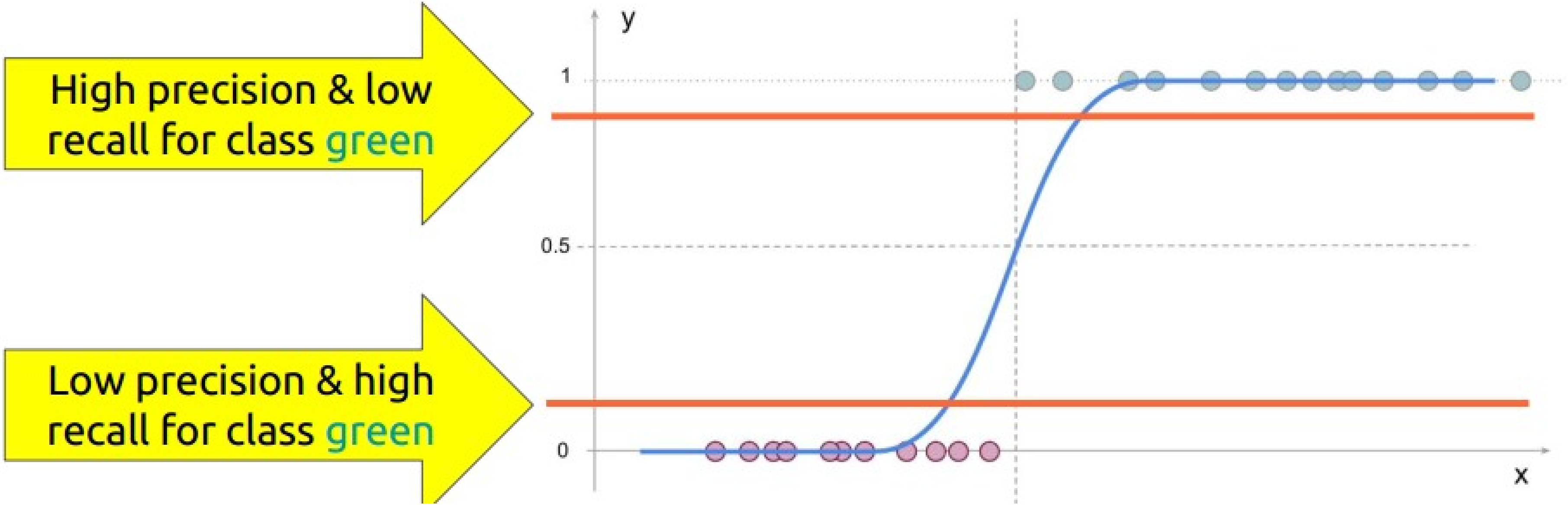

$$\sigma(\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^T \cdot \mathbf{x} + b}}$$

y > thr → class 1 (eg. green)

y < thr → class 2 (eg. red)

# Binary classification

The classification threshold can be adjusted based on the desired precision - recall trade-off:



# Softmax regression (more than 2 classes)

Ideally we would like to predict the *probability* that $y$ takes a particular value given $\mathbf{x}$,

$$P(y = j|\mathbf{x}), \quad j \in \{1, \ldots, K\}$$

with:

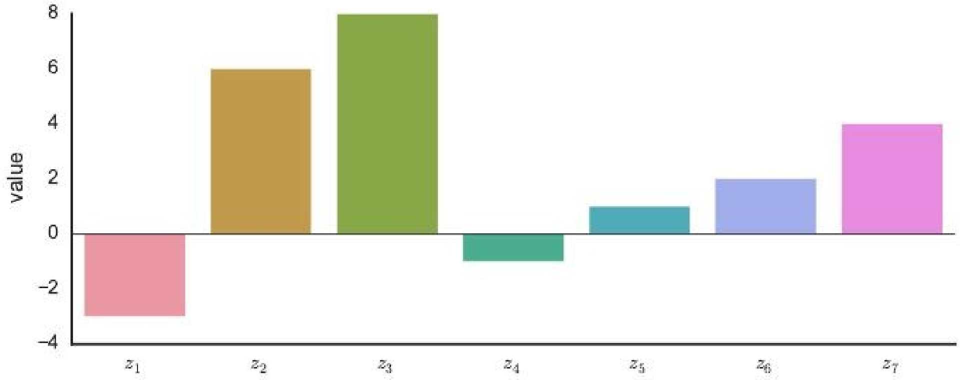
$$\sum_{j=1}^{K} P(y = j|x) = 1$$

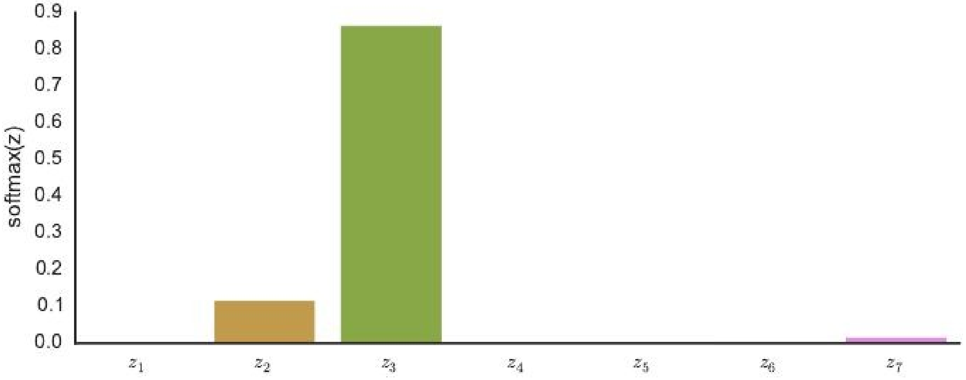
The logistic regression classifier does this for the binary case. The **softmax classifier** extends it to the multiclass case.

$$p(y = j|\mathbf{x}) = \frac{e^{(\mathbf{w}_j^T \mathbf{x} + b_j)}}{\sum_{k \in K} e^{(\mathbf{w}_k^T \mathbf{x} + b_k)}}$$

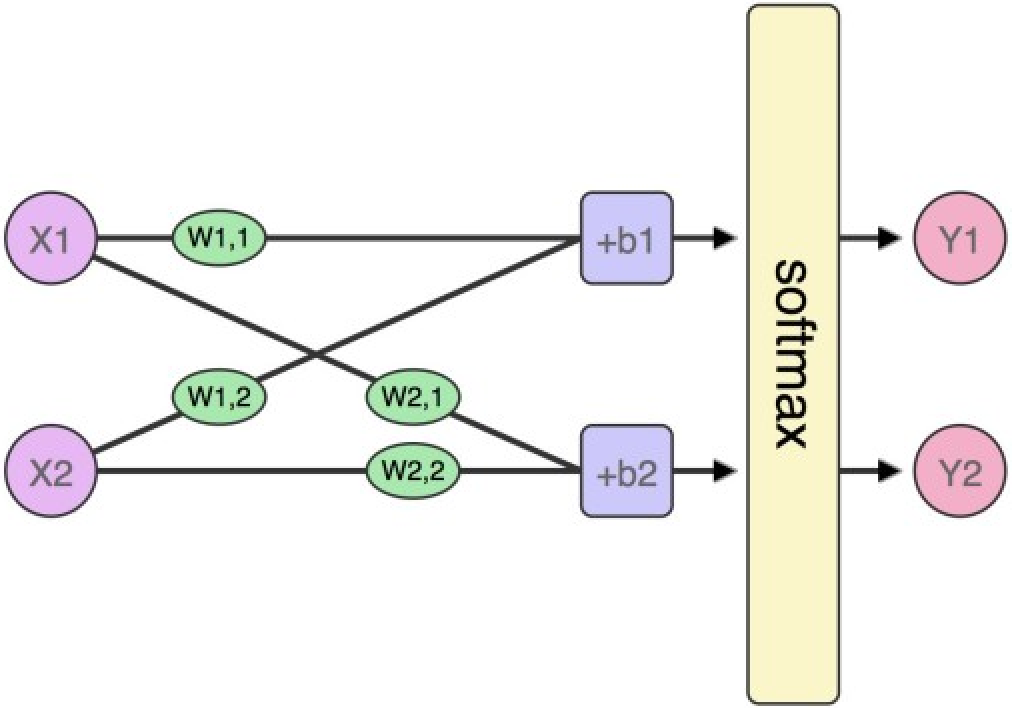
# Effect of the softmax

$$\text{softmax}(\mathbf{x}) = \frac{1}{\sum_{j=1}^{K} \exp(x_j)} \begin{bmatrix} \exp(x_1) \\ \exp(x_2) \\ \vdots \\ \exp(x_K) \end{bmatrix}$$



# Softmax regression: two classes

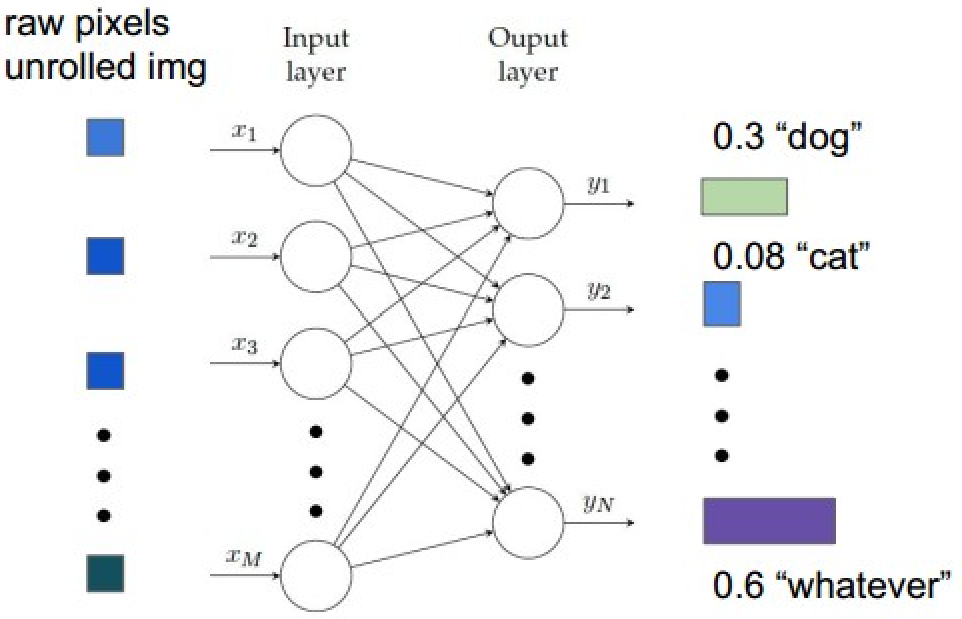



## 2 perceptrons are necessary

$$p(y=j|\mathbf{x}) = \frac{e^{(\mathbf{w}_j^T\mathbf{x}+b_j)}}{\sum_{k\in K} e^{(\mathbf{w}_k^T\mathbf{x}+b_k)}}$$

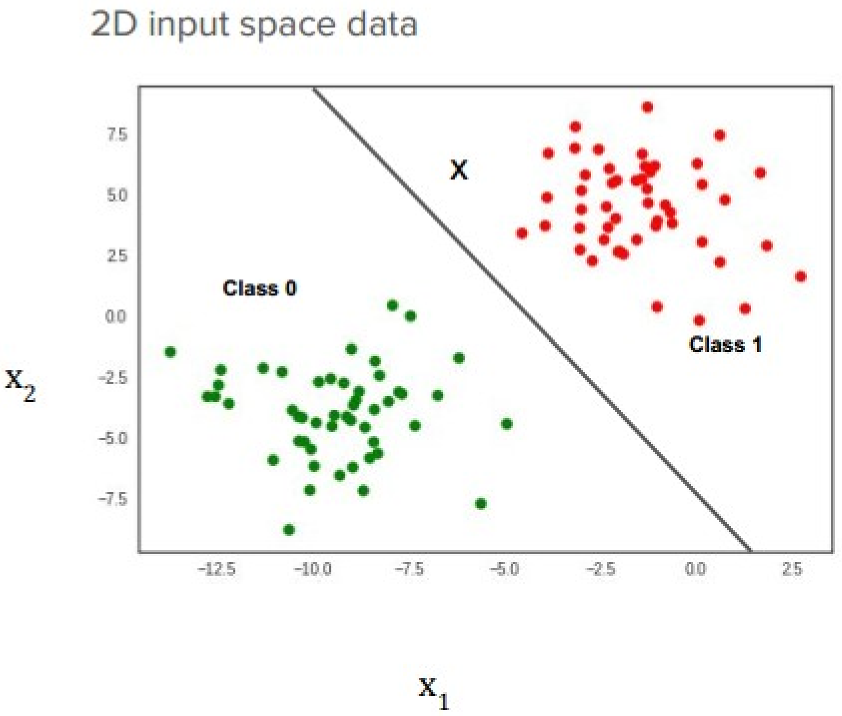
# Softmax regression: more than 2 classes

Multiple classes can be predicted by putting many neurons in parallel, each processing its binary output out of N possible classes.




$$p(y = j|\mathbf{x}) = \frac{e^{(\mathbf{w}_j^T\mathbf{x}+b_j)}}{\sum_{k\in K} e^{(\mathbf{w}_k^T\mathbf{x}+b_k)}}$$

# Limitations : what about the non-linear decision boundaries



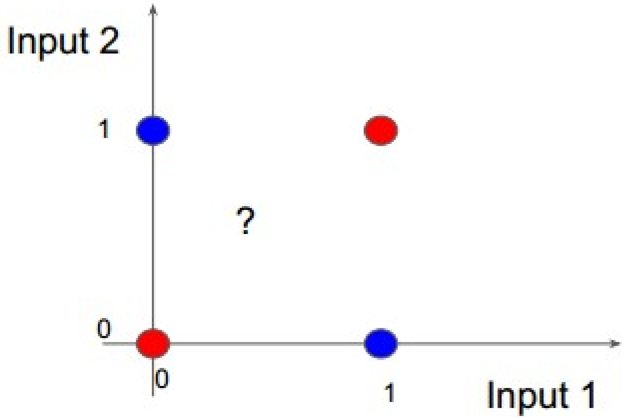
# Limitations : what about the non-linear decision boundaries

XOR logic table

| Input 1 | Input 2 | Desired Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Data might be **non linearly separable**

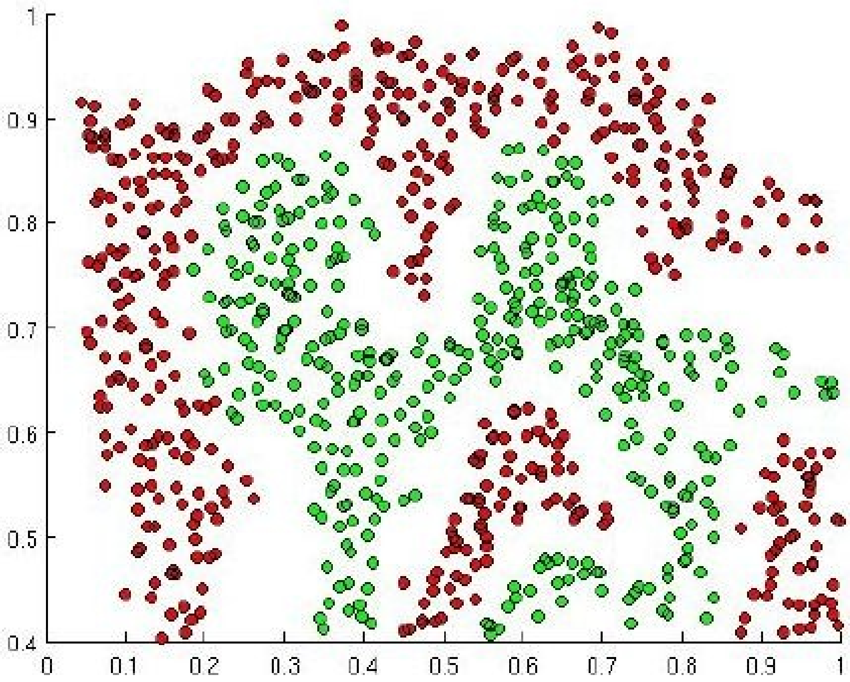
→ One single neuron is not enough



# Limitations : what about the non-linear decision boundaries

Real world problems often need non-linear boundaries

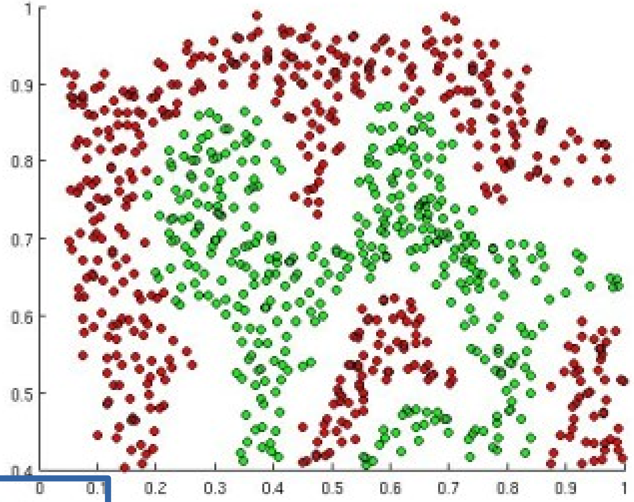
- Images
- Audio
- Text



# Limitations : what about the non-linear decision boundaries

**What can we do?**

1. Use a non-linear classifier
   - Decision trees (and forests)
   - K nearest neighbours
2. Engineer a suitable representation
   - One in which features are more linearly separable
   - Then use a linear model
3. Engineer a kernel
   - Design a kernel **K($x_1$, $x_2$)**
   - Use kernel methods (e.g. SVM)
4. Learn a suitable representation space from the data
   - Deep learning, deep neural networks
   - Boosted cascade classifiers like Viola Jones also take this approach



# The perceptron

- Supervised learning : regression/classification
- Single neuron model
  - Linear regression
  - Logistic regression
  - Multi-output and softmax regression
- **Multi-layer perception**

# Multi layer perceptron

When each node in each layer is a linear combination of **all inputs from the previous layer** then the network is called a **multilayer perceptron** (MLP)

Weights can be organized into matrices.

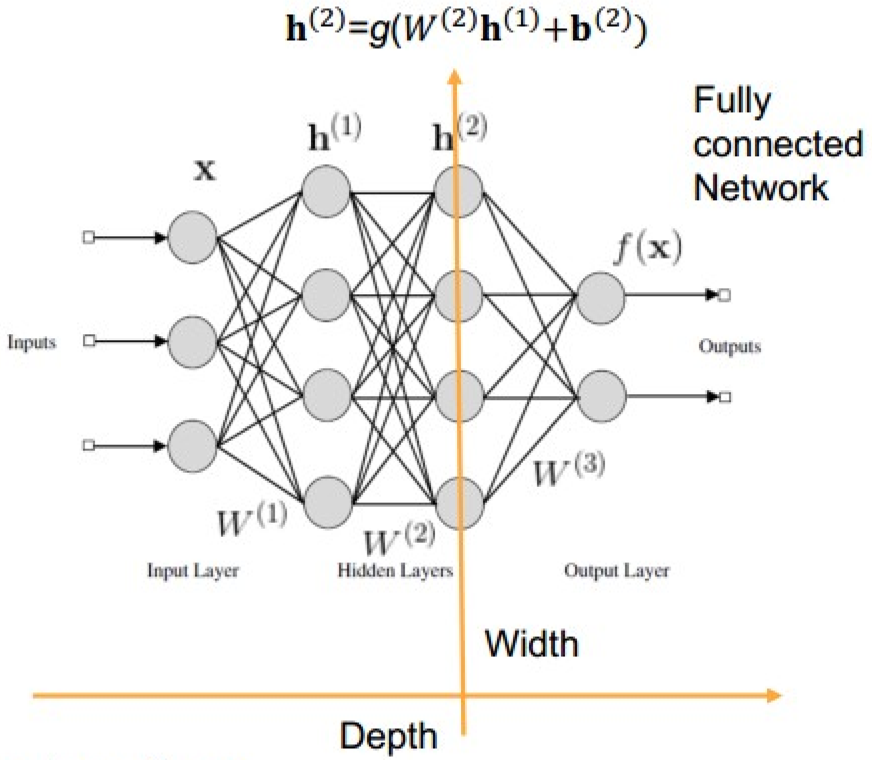
**Forward pass** computes

$$\mathbf{h}_0 = \mathbf{x}$$

$$\mathbf{h}^{(t)} = g(W^{(t)}\mathbf{h}^{(t-1)} + \mathbf{b}^{(t)})$$

$$f(\mathbf{x}) = \mathbf{h}^{(L)}$$

$$y = f(x)$$




$g$: activation function. i.e. sigmoid $f$: target function. i.e. softmax

# Multi layer perceptron (matrix)

$W_1$

| | | | |
|---|---|---|---|
| $w_{11}$ | $w_{12}$ | $w_{13}$ | $w_{14}$ |
| $w_{21}$ | $w_{22}$ | $w_{23}$ | $w_{24}$ |
| $w_{31}$ | $w_{32}$ | $w_{33}$ | $w_{34}$ |
| $w_{41}$ | $w_{42}$ | $w_{43}$ | $w_{44}$ |

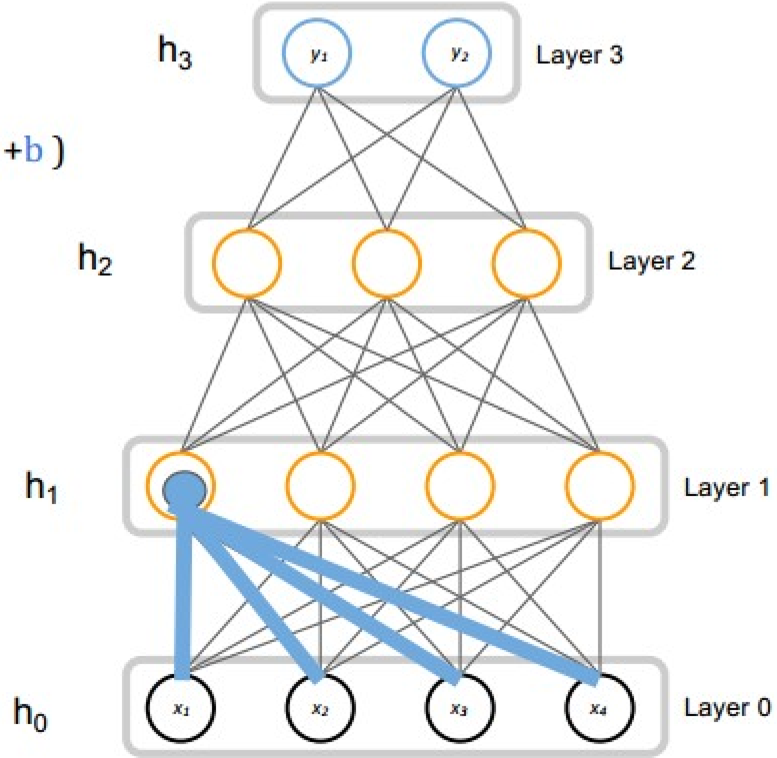
$h_0$

| |
|---|
| $x_1$ |
| $x_2$ |
| $x_3$ |
| $x_4$ |

$b_1$

| |
|---|
| $b_1$ |
| $b_2$ |
| $b_3$ |
| $b_4$ |

$h_{11} = g(\, \mathbf{wx} + b \,)$




**Forward pass** computes

$$\mathbf{h}_0 = \mathbf{x}$$

$$\mathbf{h}^{(t)} = g(W^{(t)}\mathbf{h}^{(t-1)} + \mathbf{b}^{(t)})$$

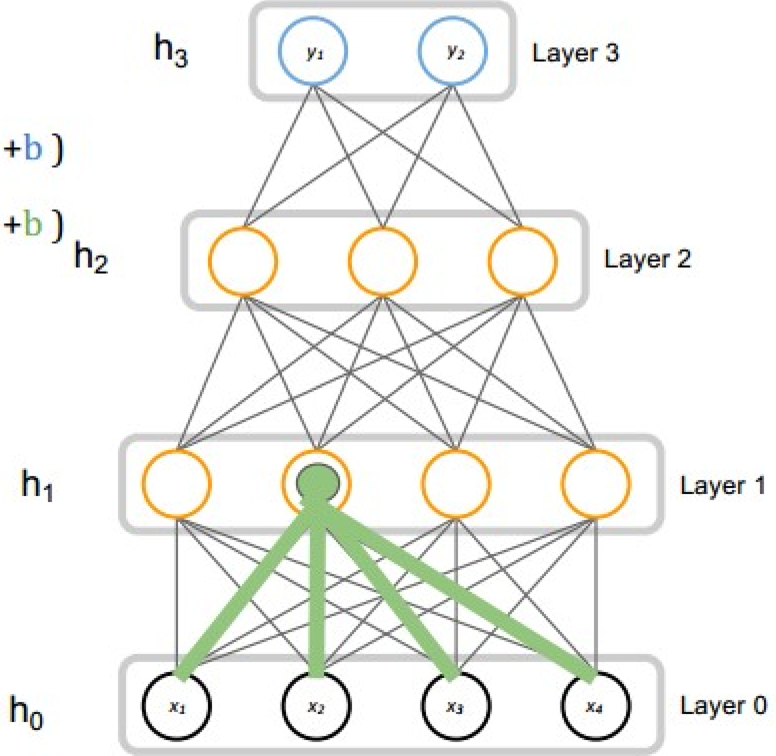
$$f(\mathbf{x}) = \mathbf{h}^{(L)}$$

# Multi layer perceptron (matrix)

$$\underset{W_1}{\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \end{bmatrix}} \quad \underset{h_0}{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}} \quad \underset{b_1}{\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}}$$

$h_{11} = g(\,\mathbf{wx} + b\,)$

$h_{12} = g(\,\mathbf{wx} + b\,)$




**Forward pass** computes

$$\mathbf{h}_0 = \mathbf{x}$$

$$\mathbf{h}^{(t)} = g(W^{(t)}\mathbf{h}^{(t-1)} + \mathbf{b}^{(t)})$$

$$f(\mathbf{x}) = \mathbf{h}^{(L)}$$

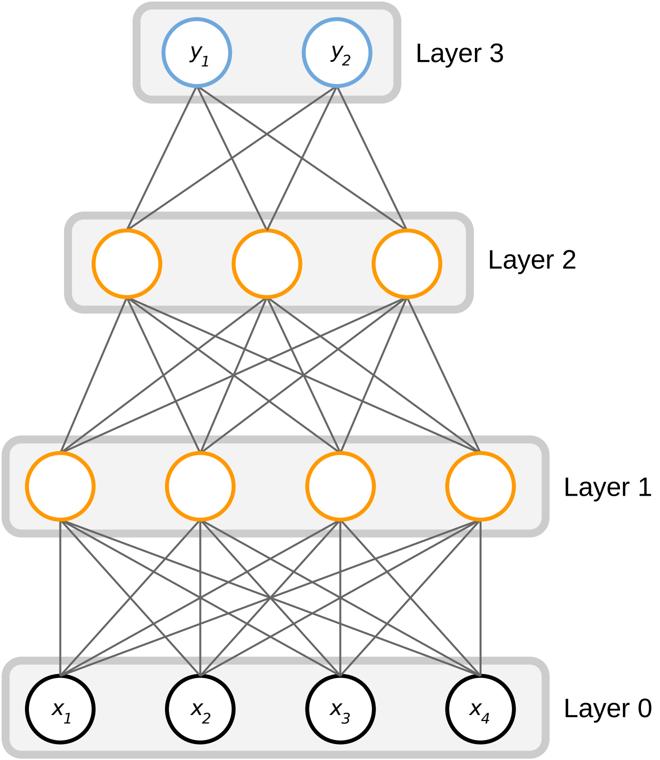
# Deep Neural Network

The i-th layer is defined by a matrix **Wi** and a vector **bi,** and the activation is simply a dot product plus **bi**:

$$h_i = f(W_i \cdot h_{i-1} + b_i)$$

Num parameters to learn at i-th layer:

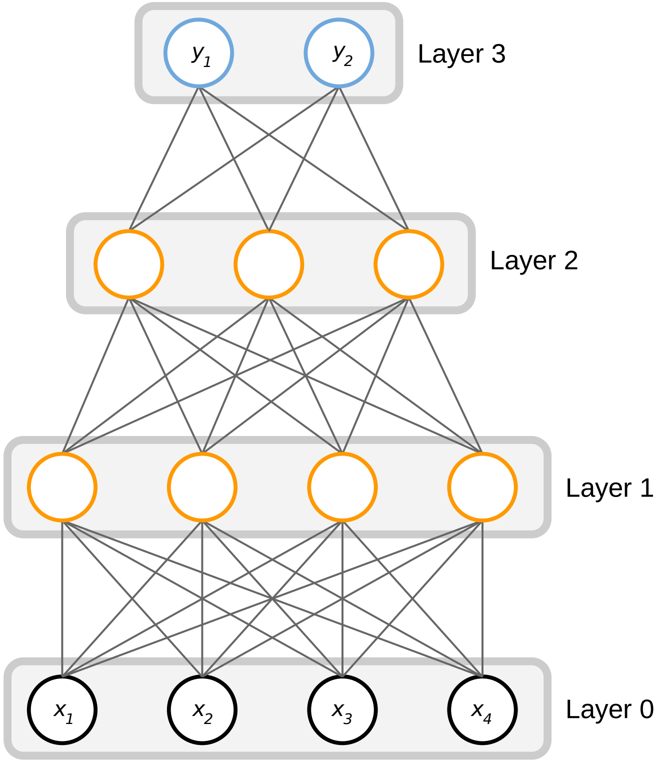
$$N^i_{params} = N^i_{inputs} \times N^i_{units} + N^i_{units}$$



# Deep nets

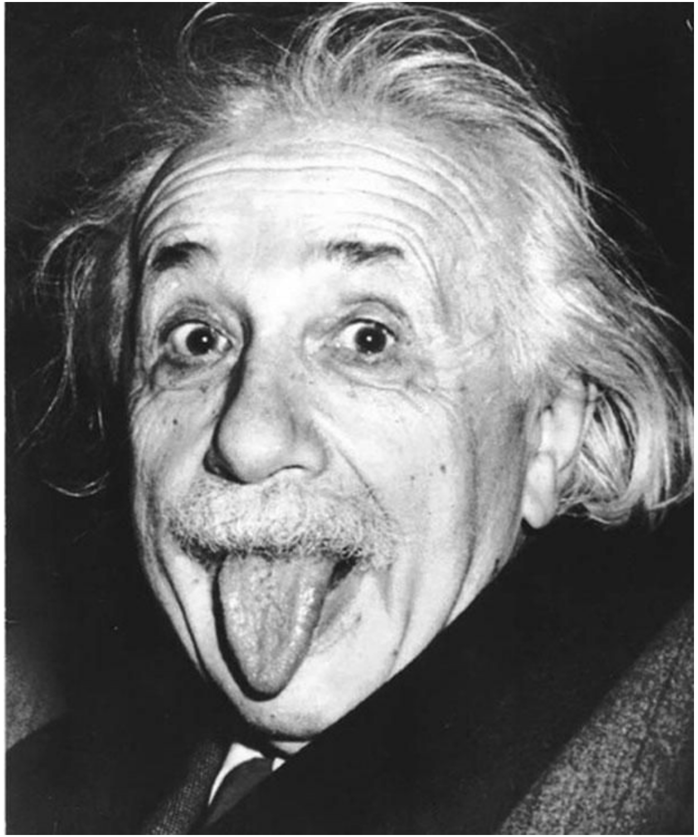
Just a neural network with several hidden layers

- Often uses different types of layers, especially **fully connected**, **convolution,** and **pooling** layers
- Output of each layer can be thought of as a **representation** of the input
- Outputs of the lower layers are more closely related to the input features
- Outputs of the higher layers contain more **abstract features** closer in semantics to the target variable



# From Neurons to Convolutional Neural Networks

What if the Input is an image?



# MNIST Example

**Handwritten digits**

- 60.000 training examples
- 10.000 test examples
- 10 classes (digits 0-9)
- 28x28 grayscale images(784 pixels)
- http://yann.lecun.com/exdb/mnist/

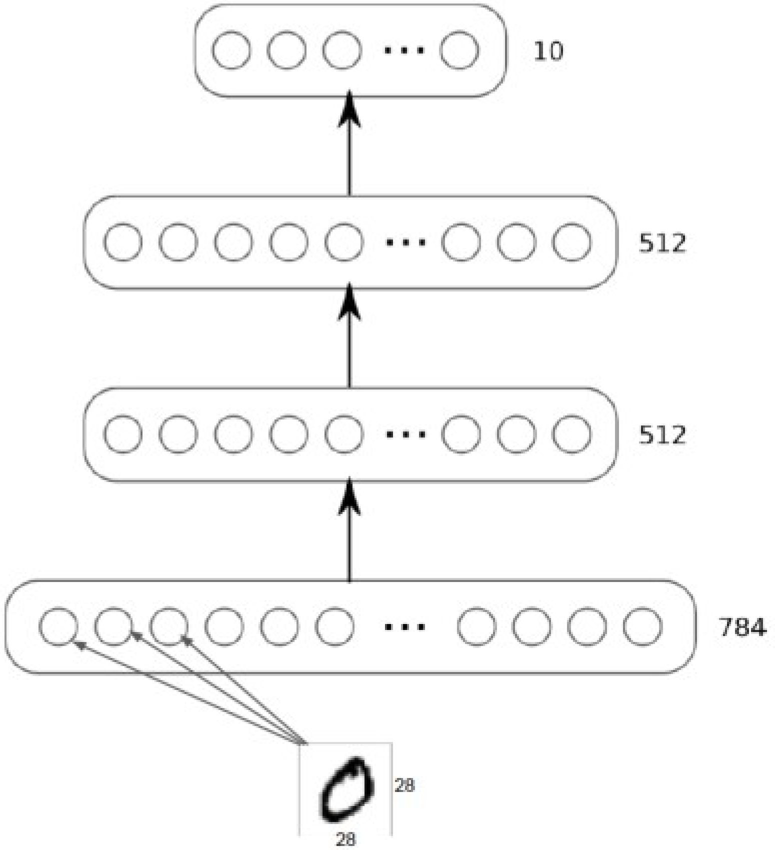
The objective is to learn a function that predicts the digit from the image

# MNIST Example

## Model

- 3 layer neural-network ( 2 hidden layers)
- Tanh units (activation function)
- 512-512-10
- Softmax on top layer
- Cross entropy Loss

| Layer | #Weights | #Biases | Total |
|---|---|---|---|
| 1 | 784 x 512 | 512 | 401,920 |
| 2 | 512 x 512 | 512 | 262,656 |
| 3 | 512 x 10 | 10 | 5,130 |
| | | | **669,706** |



# MNIST Example

**Training**

- 40 epochs using min-batch SGD
- Size of the mini-batch: 128
- Leaning Rate: 0.1 (fixed)
- Takes 5 minutes to train on GPU

**Metrics**

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

there are other metrics….

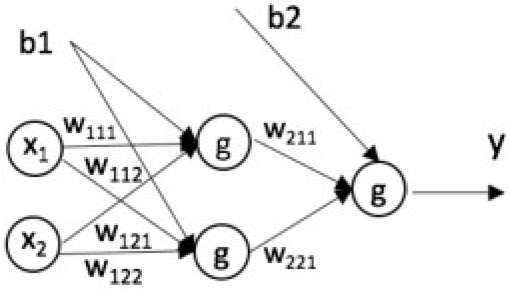
**Accuracy Results**

- 98.12% (188 errors in 10.000 test examples)

there are ways to improve accuracy…

# Exercise...

Given the following network to obtain a XNOR operation, Indicate which parameters are correct:

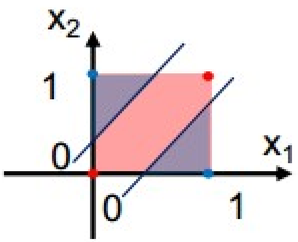


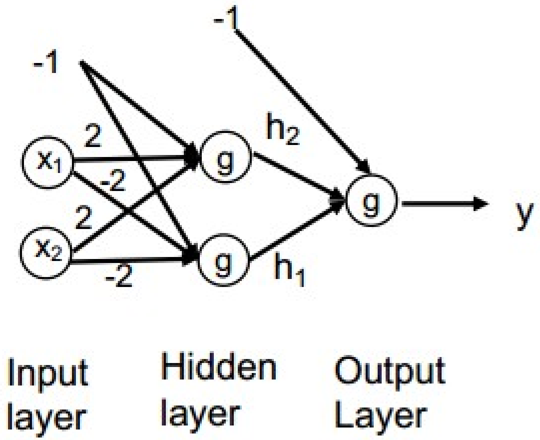

| Input vector (x1,x2) | Class XNOR |
|---|---|
| (0,0) | 1 |
| (0,1) | 0 |
| (1,0) | 0 |
| (1,1) | 1 |

- $w_{111}$=-2, $w_{112}$=2, $w_{121}$=2,$w_{122}$=-2,b1=-1,$w_{211}$=2,$w_{221}$=2,b2=-1
- $w_{111}$=-2, $w_{112}$=2, $w_{121}$=2,$w_{122}$=-2,b1=-1,$w_{211}$=2,$w_{221}$=2,b2=1
- $w_{111}$=-2, $w_{112}$=2, $w_{121}$=2,$w_{122}$=-2,b1=-1,$w_{211}$=-2,$w_{221}$=-2,b2=1
- $w_{111}$=-2, $w_{112}$=2, $w_{121}$=2,$w_{122}$=-2,b1=-1,$w_{211}$=-2,$w_{221}$=-2,b2=-1

# Exercise...

Given the following network to obtain a XNOR operation, Indicate which parameters are correct:




$$h_1 = g\left(\mathbf{w}_{11}^T\, \mathbf{x} + b_{11}\right) = u\left((-2 \quad 2) \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} - 1\right)$$

$$h_2 = g\left(\mathbf{w}_{12}^T\, \mathbf{x} + b_{12}\right) = u\left((2 \quad -2) \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} - 1\right)$$

$$y = g\left(\mathbf{w}_{2}^T\, \mathbf{h} + b_{2}\right) = u\left((2 \quad 2) \cdot \begin{pmatrix} h_1 \\ h_2 \end{pmatrix} - 1\right)$$

# Training ...

With Multiple layers we need to minimize the **loss function** $\mathcal{L}(\boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x}),\boldsymbol{y})$ with respect to all the parameters of the model $\boldsymbol{\theta}(\boldsymbol{W}^{(k)}, \boldsymbol{b}^{(k)})$**:**

$$W^{*} = argmin_{\theta}\mathcal{L}(f_{\theta}(x),y)$$

**Gradient Descent:** Move the parameter $\boldsymbol{\theta}_j$ in small steps in the direction opposite sign of the derivative of the loss with respect j:

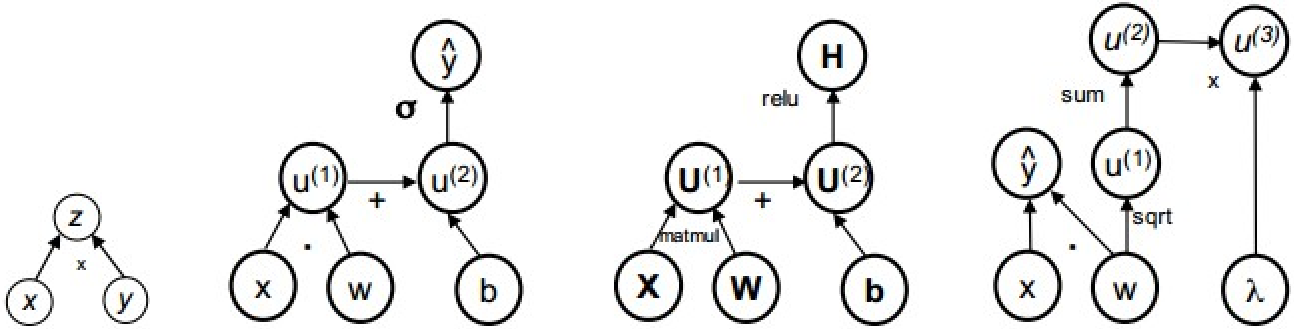
$$\boldsymbol{\theta}_j^{(n)} = \boldsymbol{\theta}_j^{(n-1)} - \boldsymbol{\alpha}^{(n-1)} \cdot \boldsymbol{\nabla}_{\boldsymbol{\theta}_j}\mathcal{L}(\boldsymbol{y},\boldsymbol{f}(\boldsymbol{x}))$$

**Stochastic gradient descent (SGD):** estimate the gradient with one sample, or better, with a **minibatch** of examples.

For MLP gradients can be found using the **chain rule** of differentiation.

The calculations reveal that the gradient wrt. the parameters in layer k only depends on the error from the above layer and the output from the layer below. This means that the gradients for each layer can be computed iteratively, starting at the last layer and propagating the error back through the network. This is known as the **backpropagation** algorithm.

# Training : based on computational graphs




$z = xy$

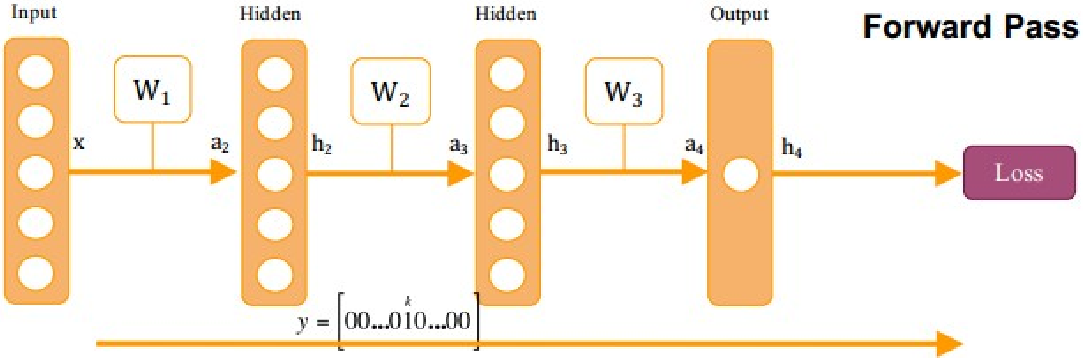
$\hat{y} = \sigma(\mathrm{x}^T \mathrm{w} + \mathrm{b})$

$\boldsymbol{H} = \max\{0, \boldsymbol{XW} + \boldsymbol{b}\}$

$\hat{y} = \mathrm{x}^T \mathrm{w}$

$\lambda \sum_i w_i^2$

# Training: two pass ...




**Probability Class given an input (softmax)**

$$p(c_k = 1|\mathbf{x}) = \frac{\exp(a_k)}{\sum_c \exp(a_c)}$$

*Figure Credit: Kevin McGuiness*

**Loss function; e.g., negative log-likelihood (good for classification)**

$$L(\mathbf{x}, y; \mathbf{W}) = -\sum_j y_j \log(p(c_j|\mathbf{x}))$$

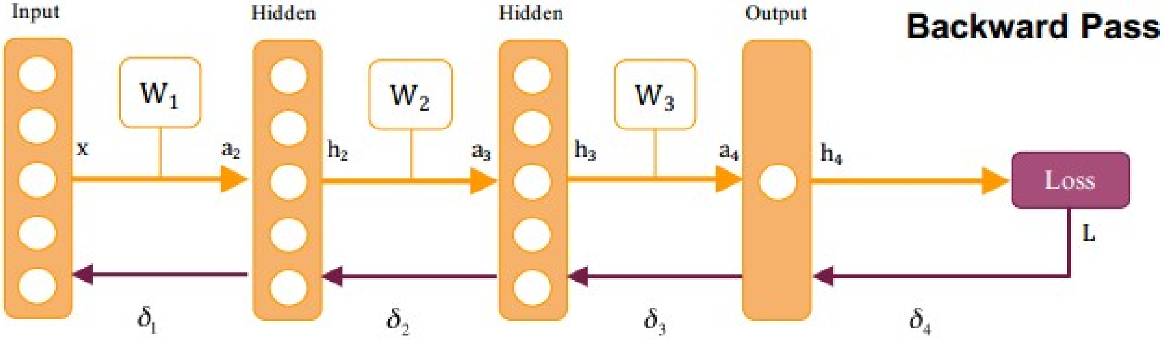
**Regularization term (L2 Norm) aka as weight decay**

$$L(\mathbf{x}, y; \mathbf{W}) = -\sum_j y_j \log(p(c_j|\mathbf{x})) + \frac{\lambda}{2}\|\mathbf{W}\|_2^2$$

**Minimize the loss (plus some regularization term) w.r.t. Parameters over the whole training set.**

$$\mathbf{W}^* = argmin_\theta \sum_j L(\mathbf{x}^n, y^n; \mathbf{W})$$

# Training: two pass ...




**1. Find the error in the top layer:**

$$\delta_K = \frac{\partial L}{\partial a_K}$$

$$\delta_K = \frac{\partial L}{\partial h_K}\frac{\partial h_K}{\partial a_K}$$

$$\delta_K = \frac{\partial L}{\partial h_K} \bullet g'(a_K)$$

*Figure Credit: Kevin McGuiness*

**2. Compute weight updates**

$$\frac{\partial L}{\partial W_k} = \frac{\partial L}{\partial a_{k+1}}\frac{\partial a_{k+1}}{\partial W_k}$$

$$\frac{\partial L}{\partial W_k} = \frac{\partial L}{\partial a_{k+1}} \bullet h_k$$

$$\frac{\partial L}{\partial W_k} = \delta_{k+1} \bullet h_k$$

**To simplify we don't consider the biass**

**3. Backpropagate error to layer below**

$$\delta_k = \frac{\partial L}{\partial a_k}$$

$$\delta_k = \frac{\partial L}{\partial a_{k+1}}\frac{\partial a_{k+1}}{\partial h_k}\frac{\partial h_k}{\partial a_k}$$

$$\delta_k = W_k^T \frac{\partial L}{\partial a_{k+1}} \bullet g'(a_k)$$

$$\delta_k = W_k^T \delta_{k+1} \bullet g'(a_k)$$

# Training: iterative estimation

**Gradient Descent:** Move the parameter $\theta_j$ in small steps in the direction opposite sign of the derivative of the loss with respect j.

$$\theta^{(n)} = \theta^{(n-1)} - \alpha^{(n-1)} \cdot \nabla_\theta \mathcal{L}\big(y, f(x)\big) - \lambda\theta^{(n-1)}$$

**Weight Decay:** Penalizes large weights, distributes values among all the parameters

**Stochastic gradient descent (SGD):** estimate the gradient with one sample, or better, with a **minibatch** of examples.

**Momentum:** the movement direction of parameters averages the gradient estimation with previous ones.

Several strategies have been proposed to update the weights: **optimizers**

# Training: weight initialization

Need to pick a starting point for gradient descent: an initial set of weights

Zero is a very **bad idea**!

- Zero is a **critical point**
- Error signal will not propagate
- Gradients will be zero: no progress

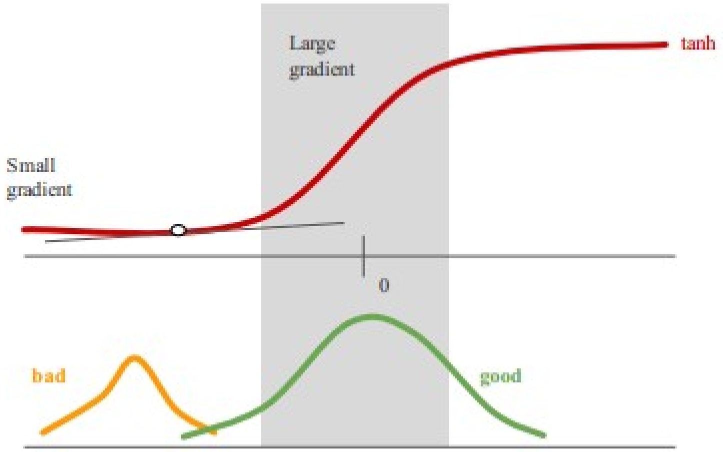
Constant value also bad idea:

- Need to break symmetry

Use **small random values:**

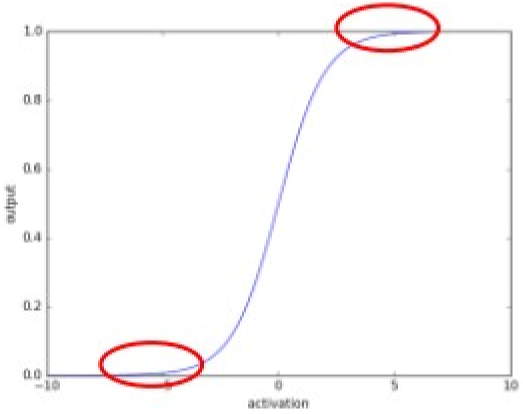
- E.g. zero mean Gaussian noise with constant variance

Ideally we want inputs to activation functions (e.g. sigmoid, tanh, ReLU) to be mostly **in the linear area** to allow larger gradients to propagate and converge faster.



# Training: vanishing gradients

In the backward pass you might be in the flat part of the sigmoid (or any other activation function like tanh) so derivative tends to zero and your training loss will not go down

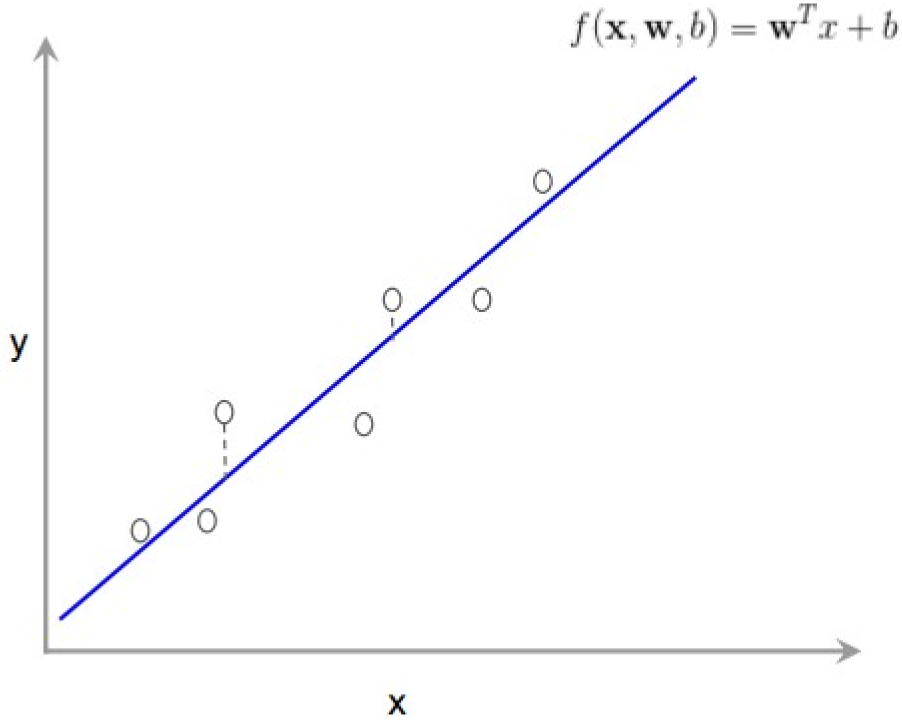


# Training: loss function (linear regression)

$$X = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_N^T \end{bmatrix} \in \mathbb{R}^{N \times D} \qquad \mathbf{y} \in \mathbb{R}^N$$

Loss function is square (Euclidean) loss

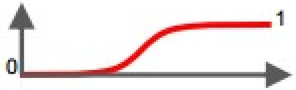
$$L(X, y, \mathbf{w}, b) = \frac{1}{2N} \sum_{i=1}^{N} (y_i - f(x_i, \mathbf{w}, b))^2$$



# Training: loss function (logistic regression)

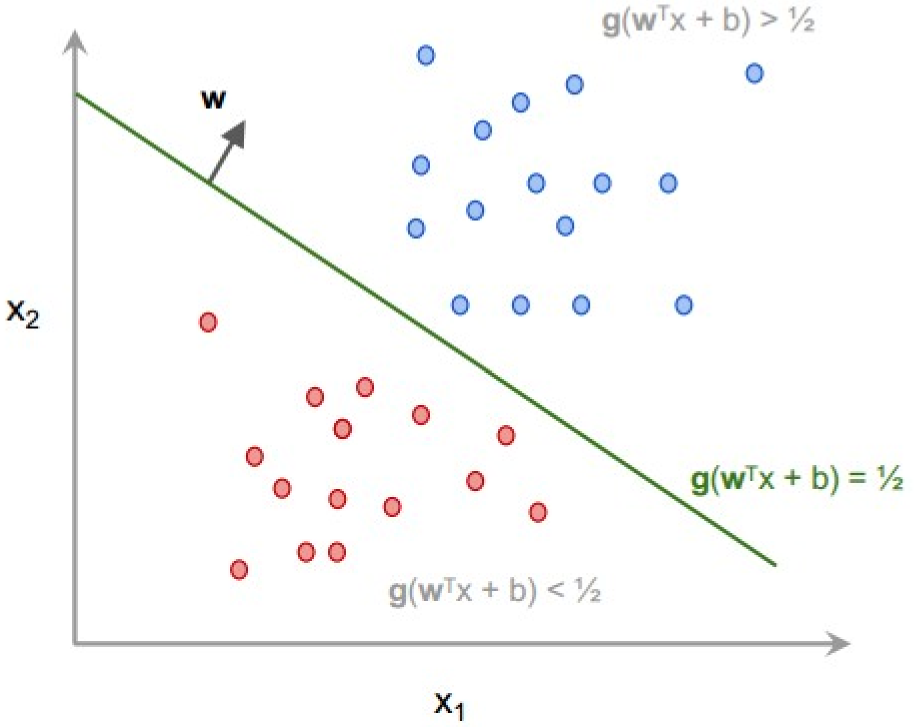
Activation function is the sigmoid

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




Loss function is cross entropy

$$L = -\frac{1}{N}\sum_{i=1}^{N} y_i \log f(\mathbf{x}_i) + (1-y_i)\log(1-f(\mathbf{x}_i))$$



# Training: gradient descent

E.g. linear regression

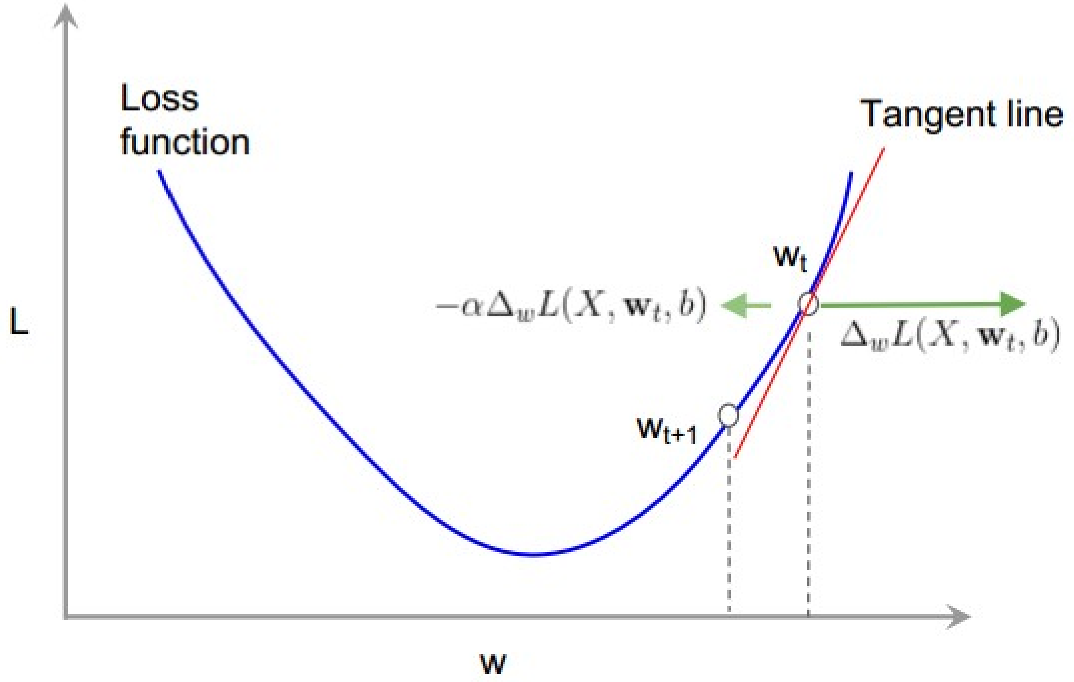
$$L(X, y, \mathbf{w}, b) = \frac{1}{2N} \sum_{i=1}^{N} (y_i - f(x_i, \mathbf{w}, b))^2$$

Need to **optimize** L

Gradient descent

$$\Delta_w L = \frac{1}{N} \sum_{i=1}^{N} (f(x_i) - y_i) x_i$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \Delta_w L(X, \mathbf{w}_t, b)$$




**α** : learning rate (aka step size)

# Training: all the hyperparameters

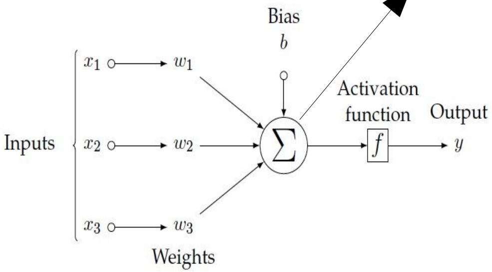
So far we have lots of **hyperparameters** to choose:

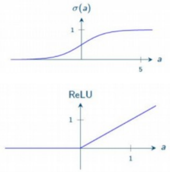
1. Learning rate ($\alpha$)
2. Regularization constant ($\lambda$)
3. Number of epochs
4. Number of hidden layers
5. Nodes in each hidden layer
6. Weight initialization strategy
7. Loss function
8. Activation functions
9. …

# Summary

## Perceptron

$$a = (\mathbf{w}^T \cdot \mathbf{x} + b)$$

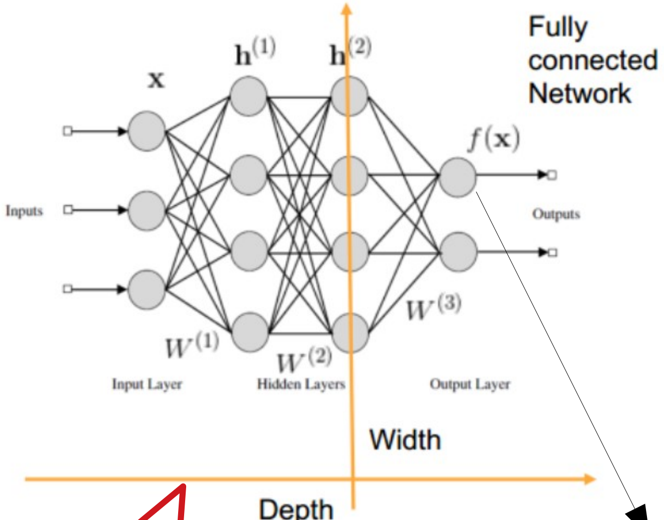


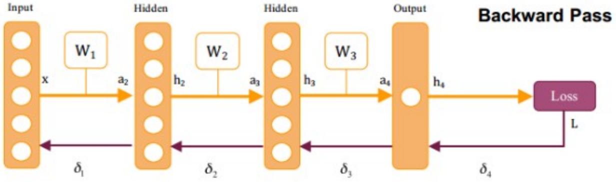

## Activation function

## Multi-layer perceptron

$$\mathbf{h}^{(2)}=g(W^{(2)}\mathbf{h}^{(1)}+\mathbf{b}^{(2)})$$




## Two steps training : backpropagation algorithm

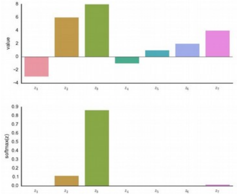



## #Parameters per layer

$$N^i_{params} = N^i_{inputs} \times N^i_{units} + N^i_{units}$$

## SoftMax function

$$p(y=j|\mathbf{x}) = \frac{e^{(\mathbf{w}_j^T\mathbf{x}+b_j)}}{\sum_{k\in K} e^{(\mathbf{w}_k^T\mathbf{x}+b_k)}}$$



# From Neurons to Convolutional Neural Networks

For a 200x200 image, we have $4x10^4$ neurons each one with $4x10^4$ inputs, that is $16x10^8$ parameters, only for one layer (not counting the bias)!!!

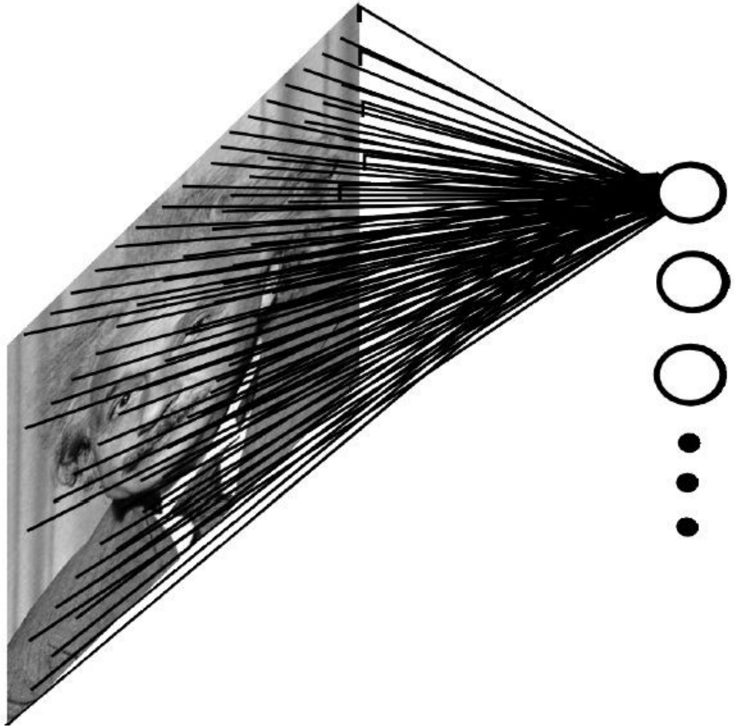


*Figure Credit: Ranzatto*

# From Neurons to Convolutional Neural Networks

For a 200x200 image, we have
$4x10^4$ neurons.

Each neuron is connected to a
local n X n pixels : for example 10X10.

***#weigths: $4X10^6$***

***A set of weights depends on the pixel location (not invariant to translation)***

What else can we do to
reduce the number
of parameters ?

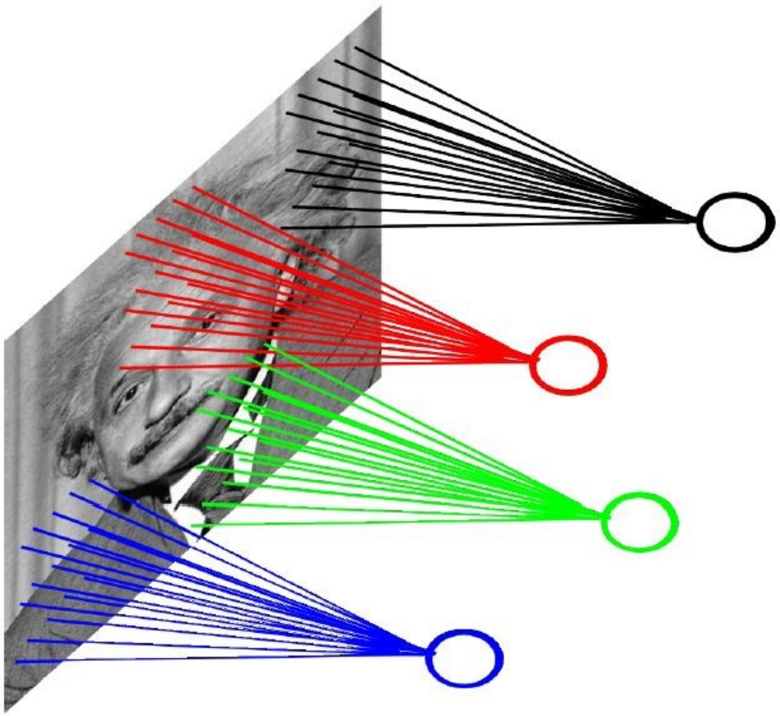
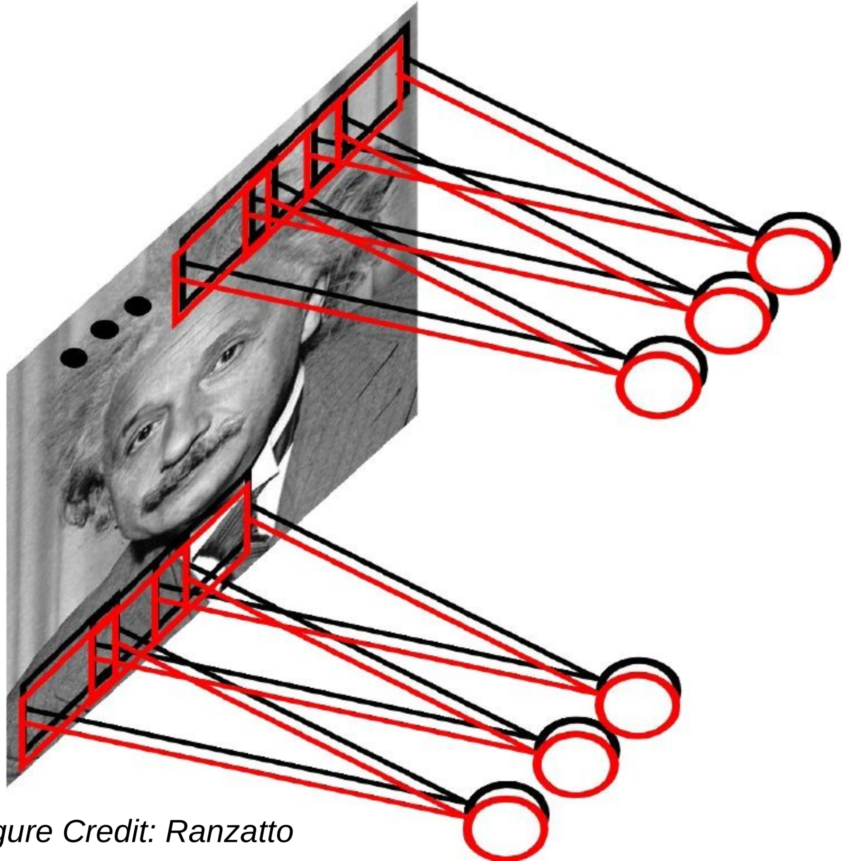


*Figure Credit: Ranzatto*

# From Neurons to Convolutional Neural Networks



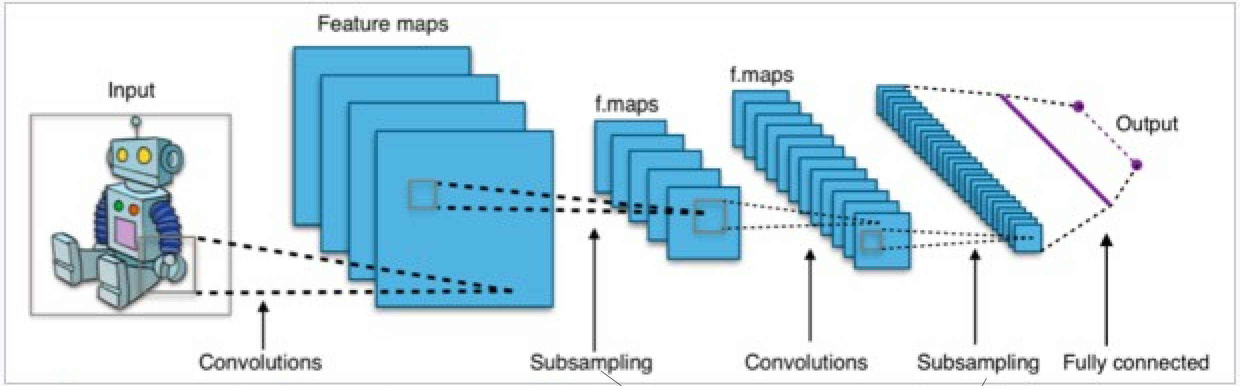
**Translation invariance:** we can use same parameters to capture a specific “feature” in any area of the image. We can try different sets of parameters to capture different features.

These operations are equivalent to perform **convolutions** with different filters.

*Ex: With 100 different filters (or feature extractors) of size 10 x 10 the number of parameters is $10^4$ (before : $10^8 \rightarrow 10^6 \rightarrow 10^4$)*

*Figure Credit: Ranzatto*

# Common form of CNN




Architecture standard d'un réseau à convolutions.

Pooling

- `INPUT -> FC` implémente un classifieur linéaire
- `INPUT -> CONV -> RELU -> FC`
- `INPUT -> [CONV -> RELU -> POOL] * 2 -> FC -> RELU -> FC` Ici, il y a une couche de CONV unique entre chaque couche POOL
- `INPUT -> [CONV -> RELU -> CONV -> RELU -> POOL] * 3 -> [FC -> RELU] * 2 -> FC` Ici, il y a deux couches CONV empilées avant chaque couche POOL.

# The convolutional layer

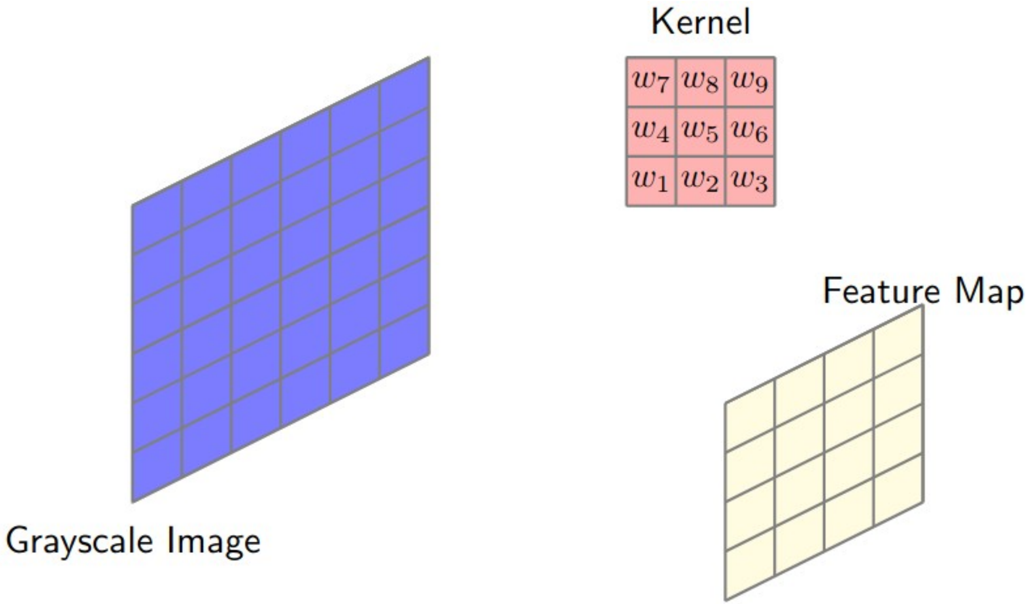
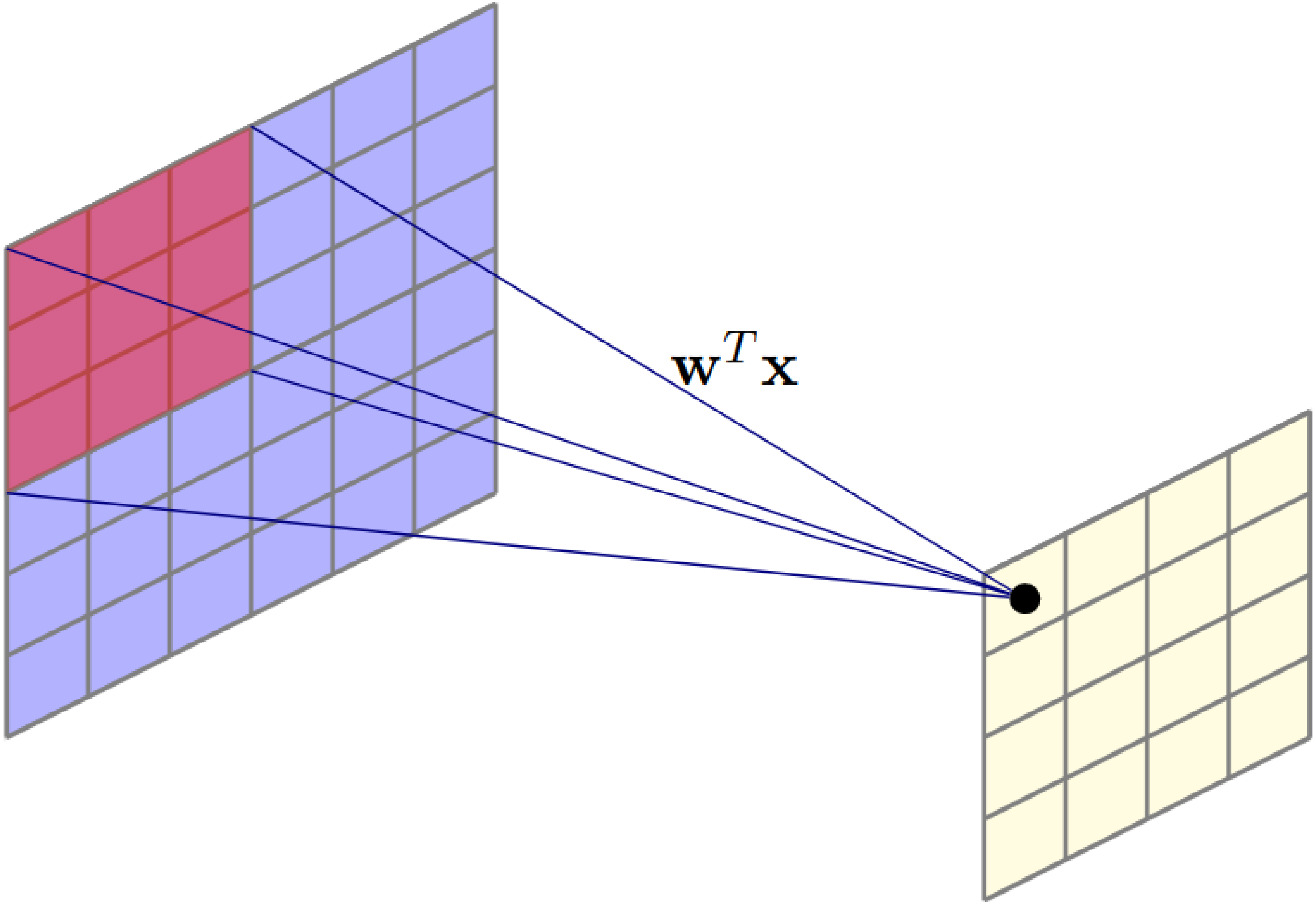
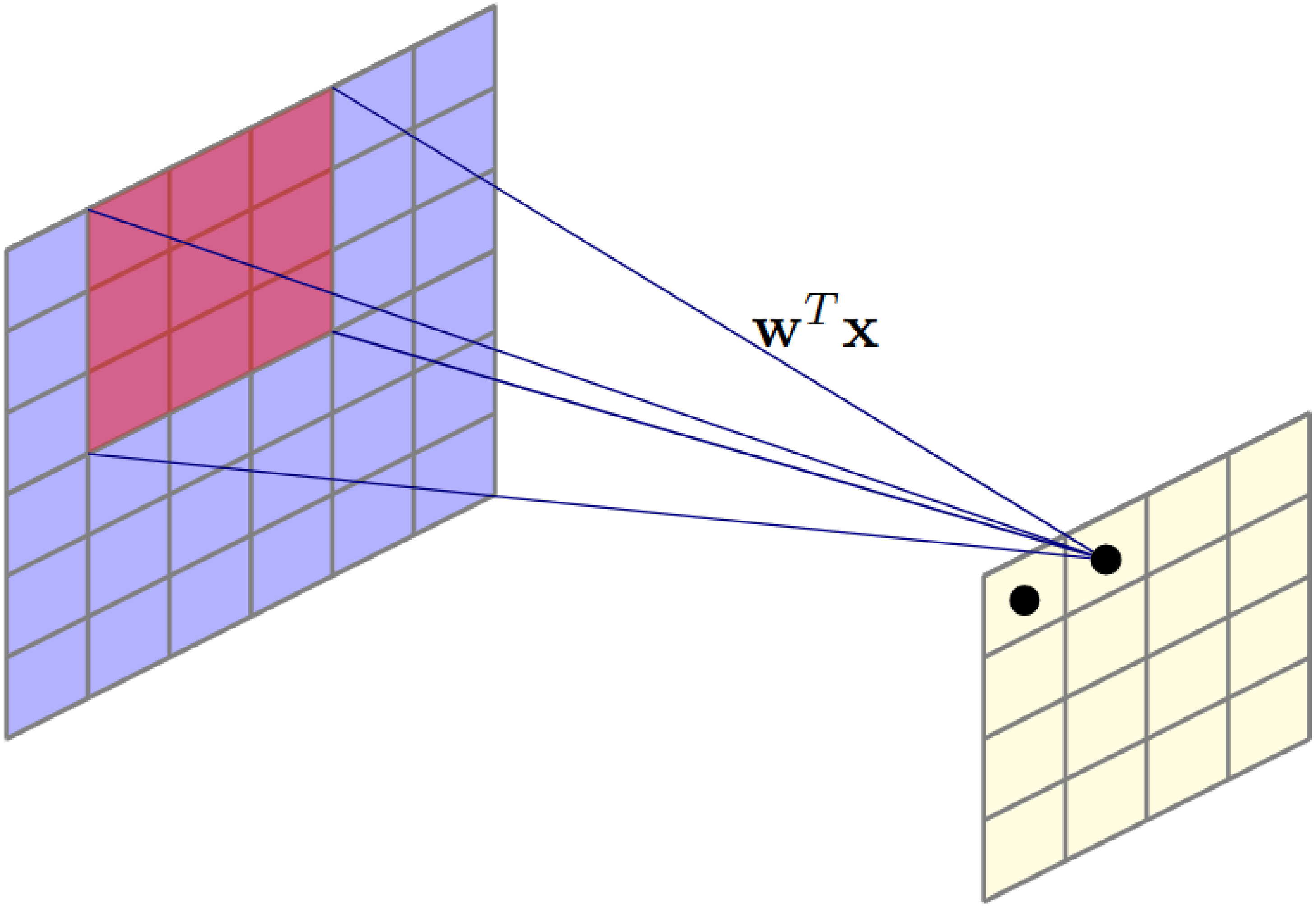
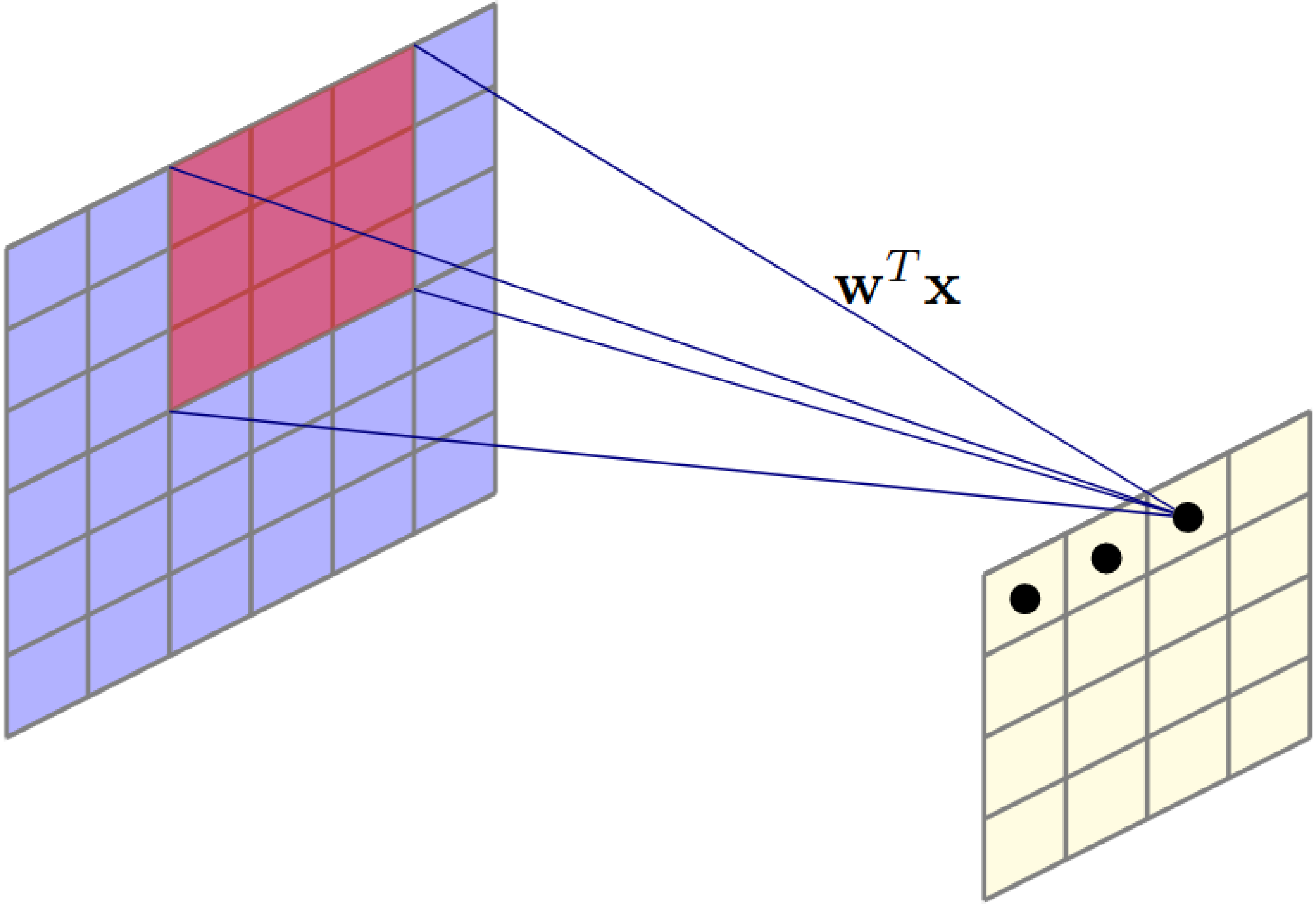
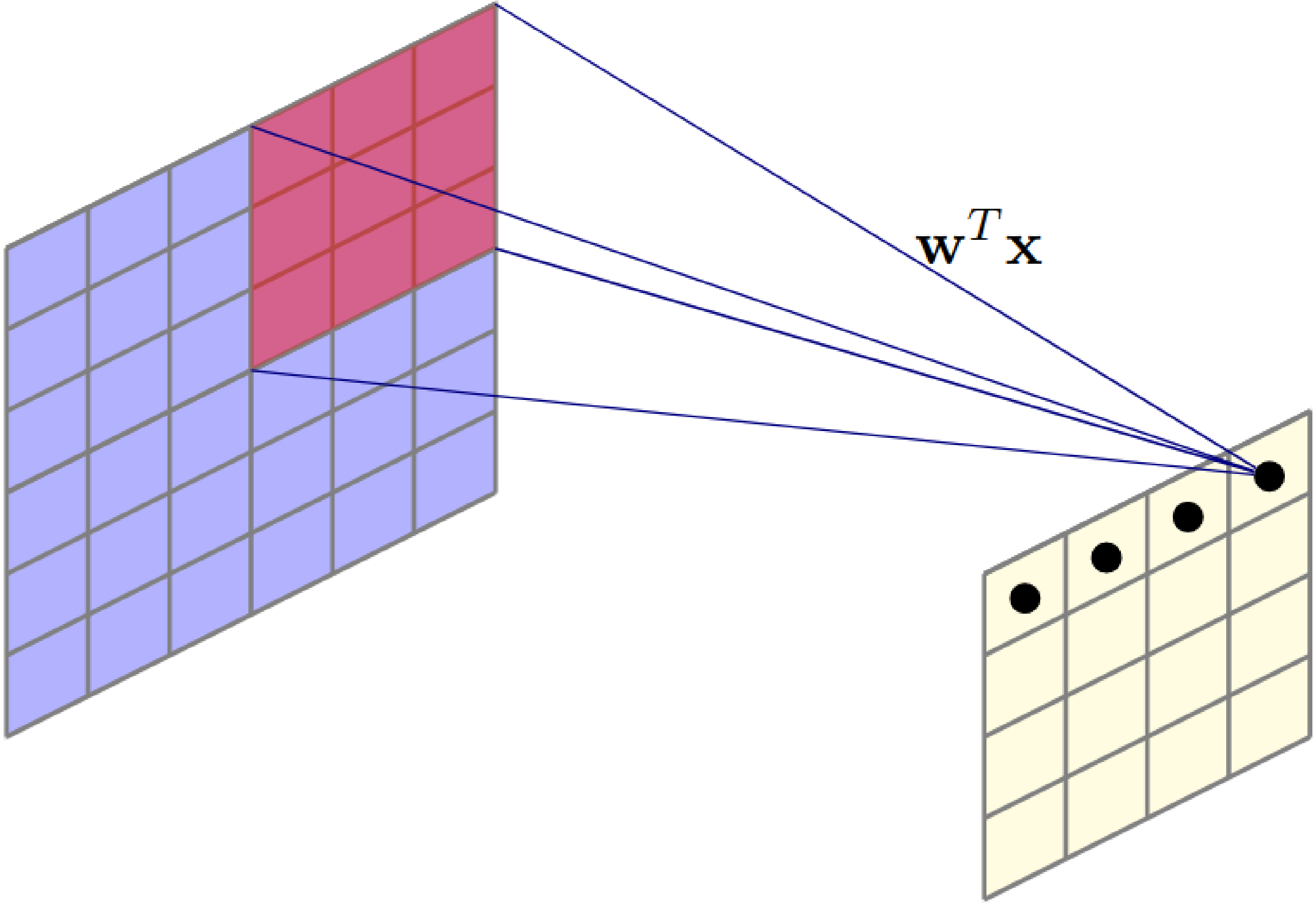


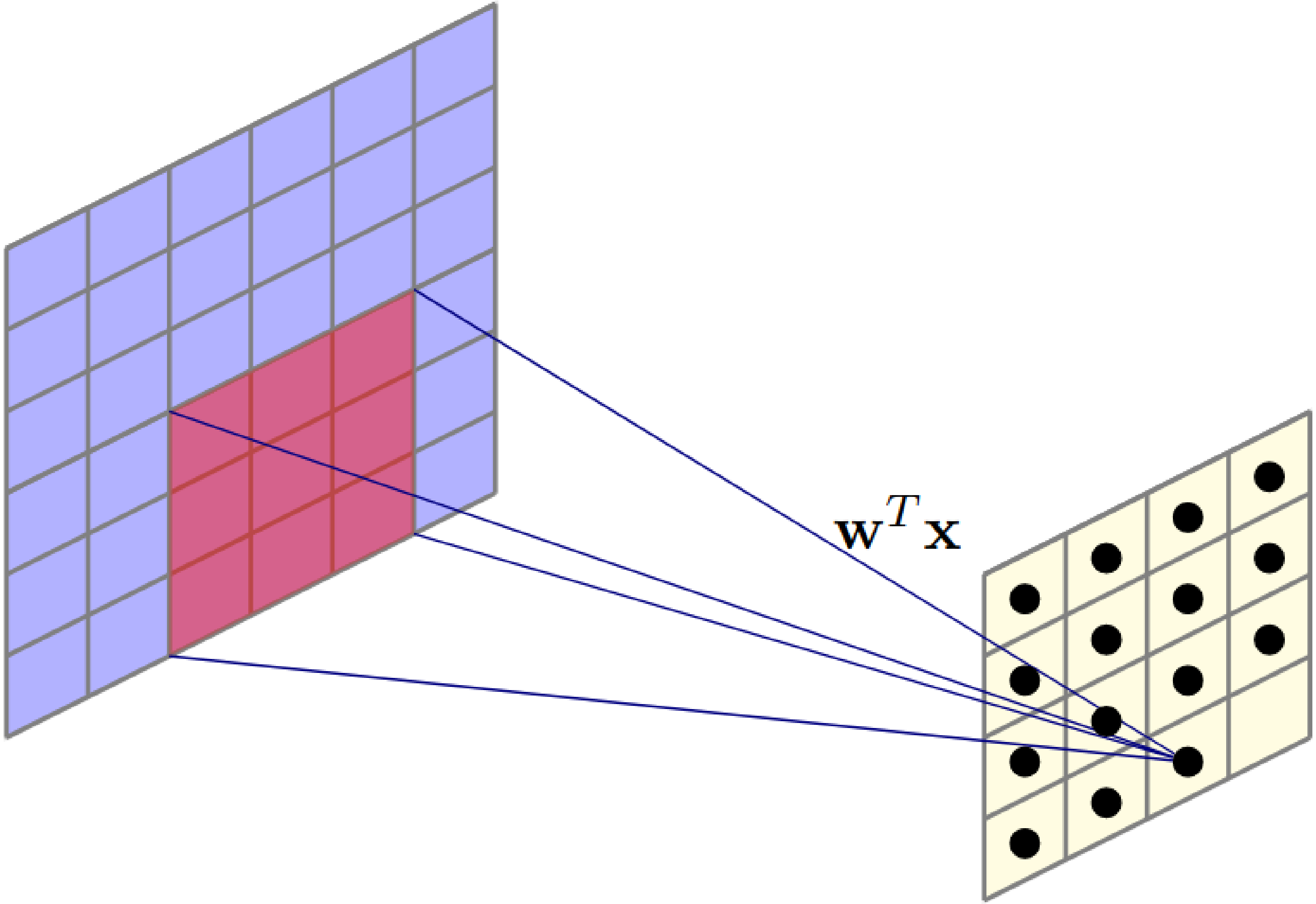

*Figure Credit: Ranzatto*

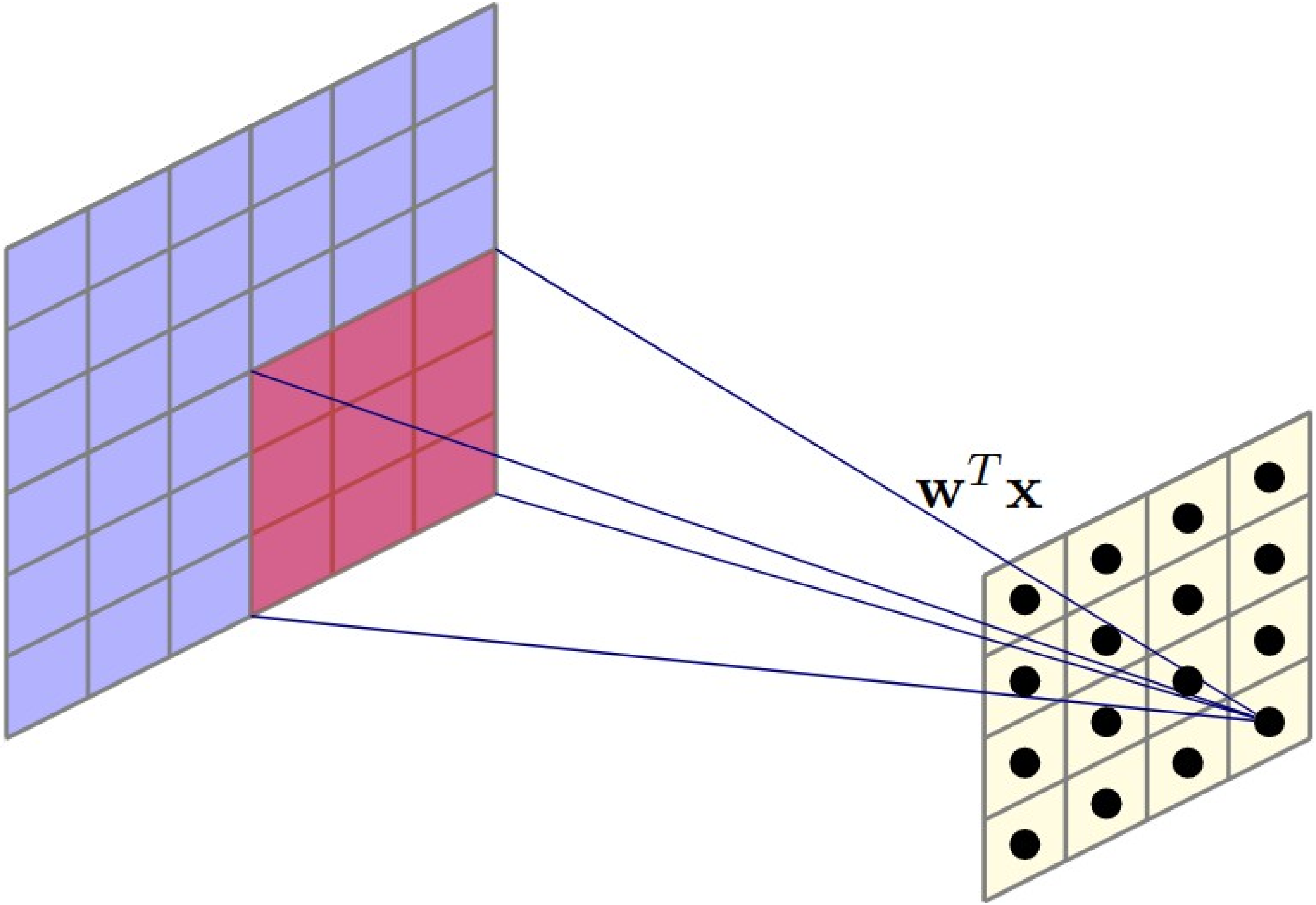

$\mathbf{w}^T\mathbf{x}$


$\mathbf{w}^T\mathbf{x}$


$\mathbf{w}^T\mathbf{x}$


$\mathbf{w}^T\mathbf{x}$


$\mathbf{w}^T\mathbf{x}$


$\mathbf{w}^T\mathbf{x}$

# The convolutional layer: output size

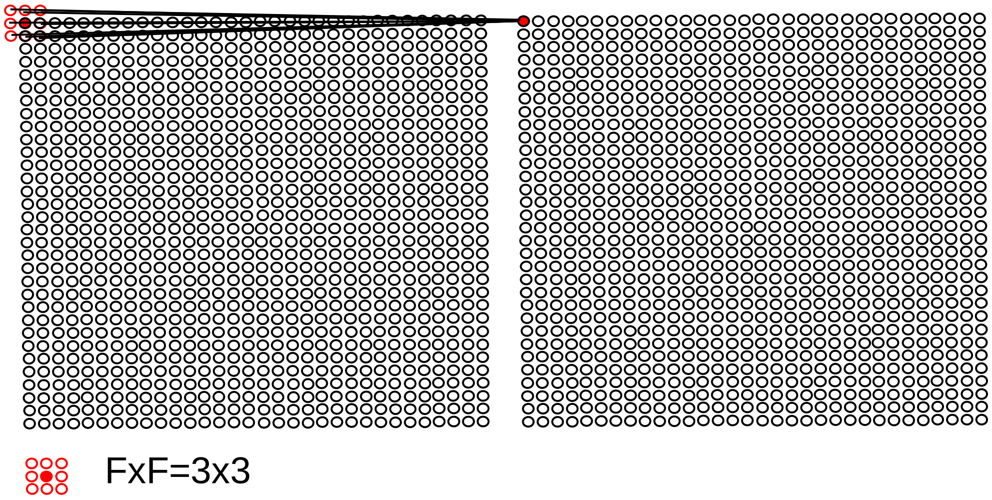
- We used stride of 1, kernel with receptive field of size 3 by 3
- Output size:

$$\frac{N-K}{S}+1$$

**Stride (S):** When doing the convolution or another operation like pooling, we may decide to slide not pixel by pixel but every s pixel. S is called stride. It is used to reduce the dimensionality of the ouput.

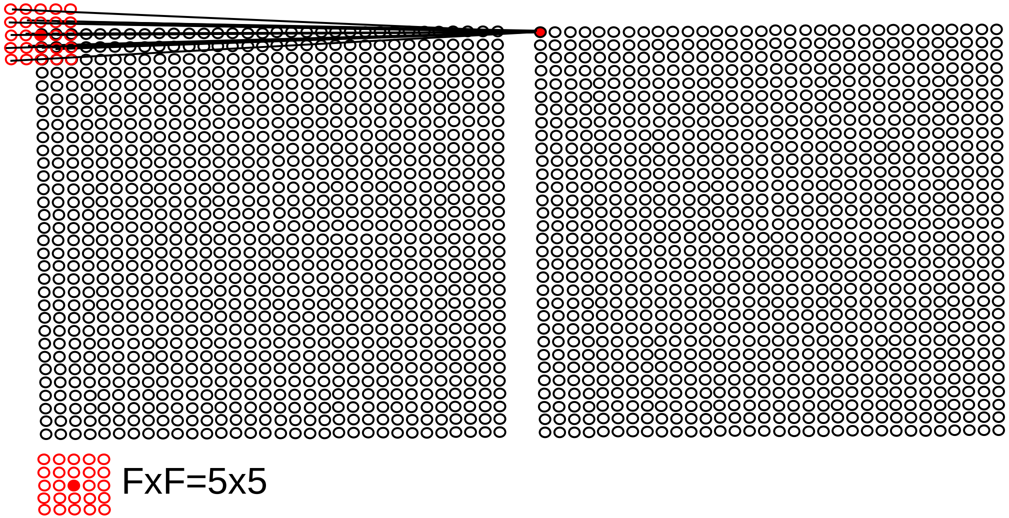
- In previous example: $N = 6, K = 3, S = 1$, Output size $= 4$
- For $N = 8, K = 3, S = 1$, output size is $6$

# From Neurons to Convolutional Neural Networks




**Padding (P):** When doing the convolution in the borders, you may add values to compute the convolution.
When the values are zero, that is quite common, the technique is called zero-padding.
When padding is not used the output size is reduced.

# From Neurons to Convolutional Neural Networks




**Padding (P):** When doing the convolution in the borders, you may add values to compute the convolution.
When the values are zero, that is quite common, the technique is called zero-padding.
When padding is not used the output size is reduced.

# The convolutional layer: Padding

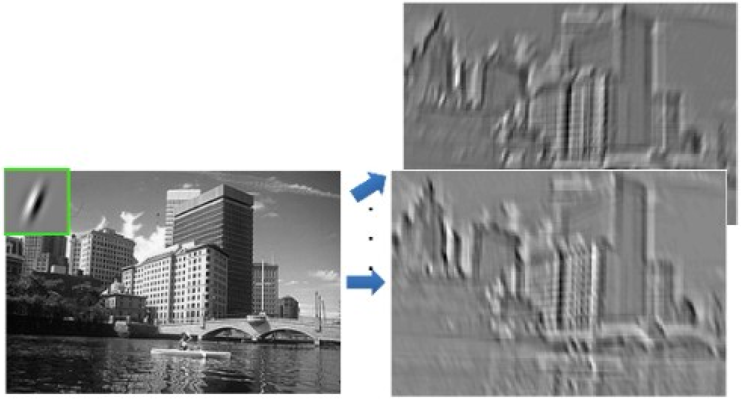
- Often, we want the output of a convolution to have the same size as the input. Solution: Zero padding.
- In our previous example:

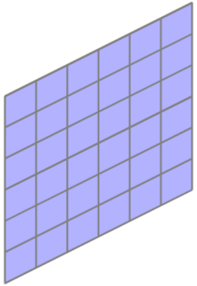
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | | | | | | | 0 |
| 0 | | | | | | | 0 |
| 0 | | | | | | | 0 |
| 0 | | | | | | | 0 |
| 0 | | | | | | | 0 |
| 0 | | | | | | | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

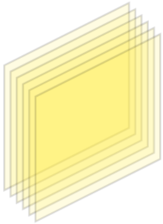
- Common to see convolution layers with stride of 1, filters of size $K$, and zero padding with $\frac{K-1}{2}$ to preserve size

# The convolutional layer: Multiple filters

- If we use 100 filters, we get 100 feature maps

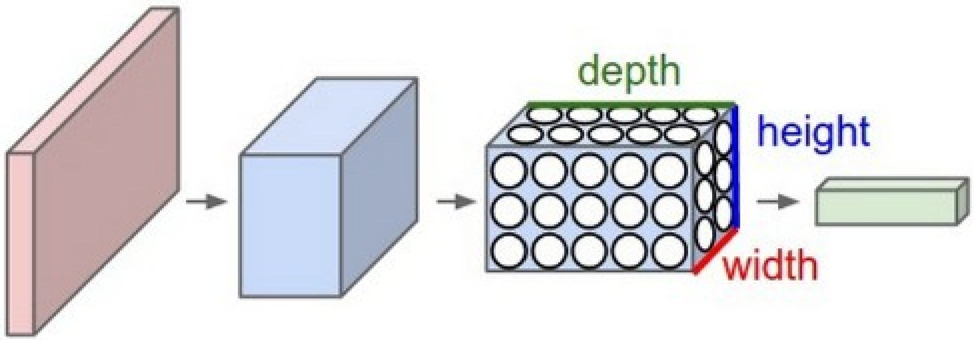






# The convolutional layer: output size

- We have only considered a 2-D image as a running example
- But we could operate on volumes (e.g. RGB Images would be depth 3 input, filter would have same depth)



# The convolutional layer: output size

- For convolutional layer:
  - Suppose input is of size $W_1 \times H_1 \times D_1$
  - Filter size is $K$ and stride $S$
  - We obtain another volume of dimensions $W_2 \times H_2 \times D_2$
  - As before:

$$W_2 = \frac{W_1 - K}{S} + 1 \text{ and } H_2 = \frac{H_1 - K}{S} + 1$$

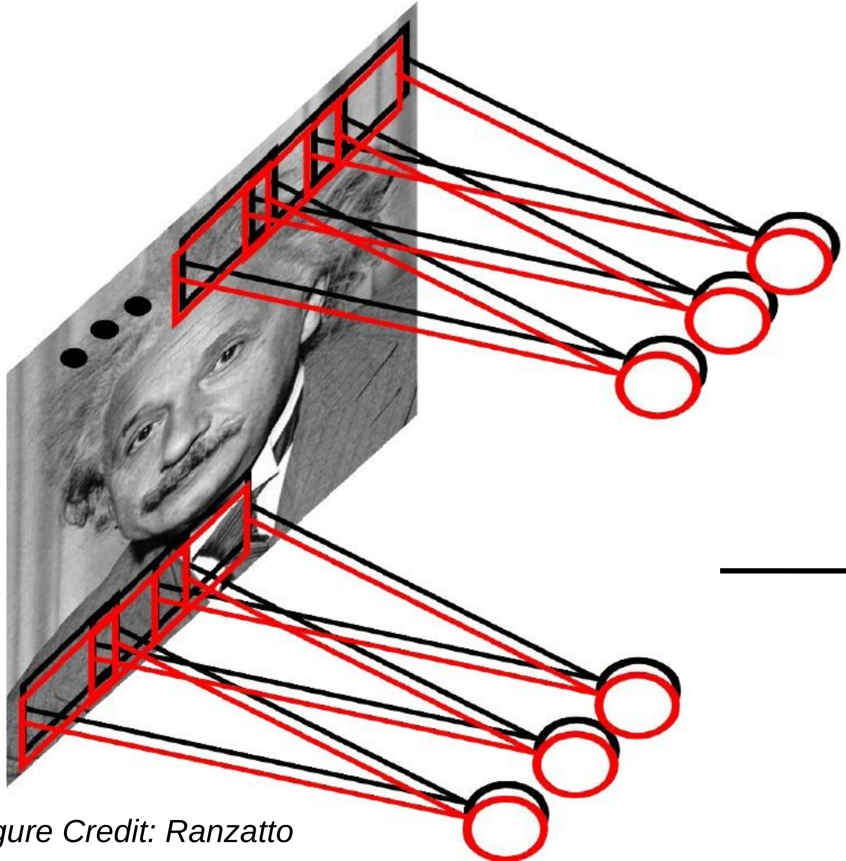
  - Depths will be equal

# The convolutional layer: output size

## Example

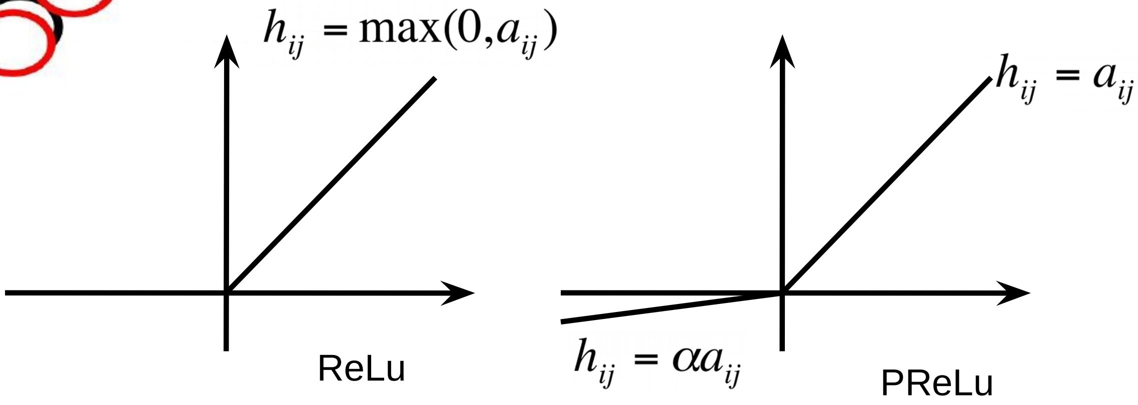
Example volume: $28 \times 28 \times 3$ (RGB Image)
100 $3 \times 3$ filters, stride 1
What is the zero padding needed to preserve size?
Number of parameters in this layer?
For every filter: $3 \times 3 \times 3 + 1 = 28$ parameters
Total parameters: $100 \times 28 = 2800$

# The convolutional layer: Activation (non linearity)



… and don’t forget the activation function!

$$a_{ij} = \sum_{k,l} w_{kl} x_{k-i,l-j} + b$$




$$g(a) = sigm(a) = \frac{1}{1+\exp(-a)}$$

$$g(a) = \tanh(a)$$

*Figure Credit: Ranzatto*

# The convolutional layer: Pooling layer

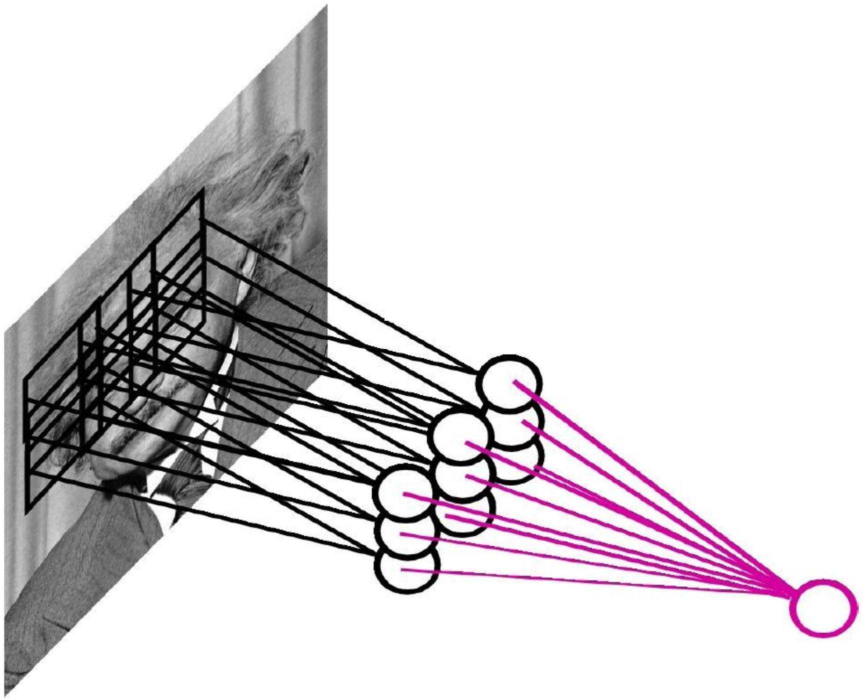


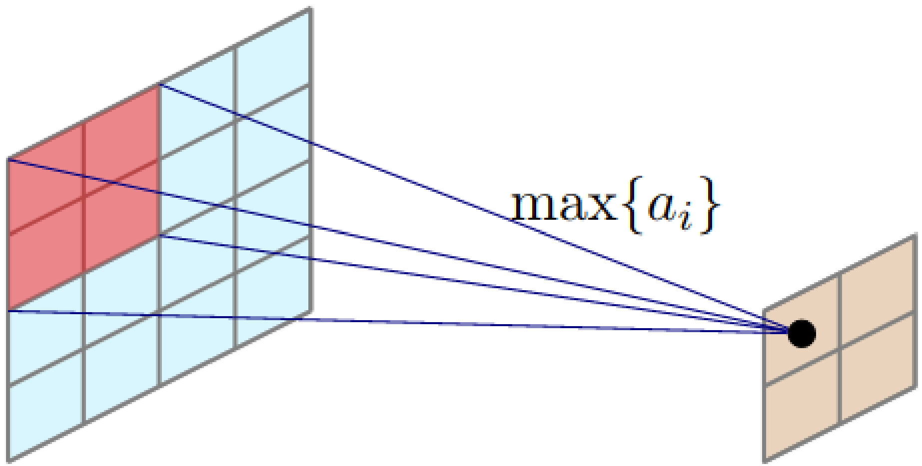
Figure Credit: Ranzatto

Most  ConvNets use **Pooling** (or subsampling) to reduce dimensionality and provide invariance to small local changes.

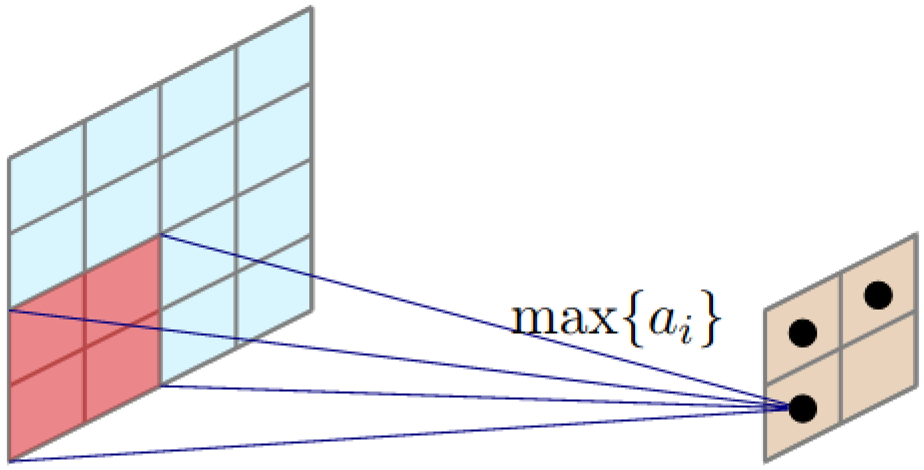
Pooling options:

- **Max**
- Average
- Stochastic pooling

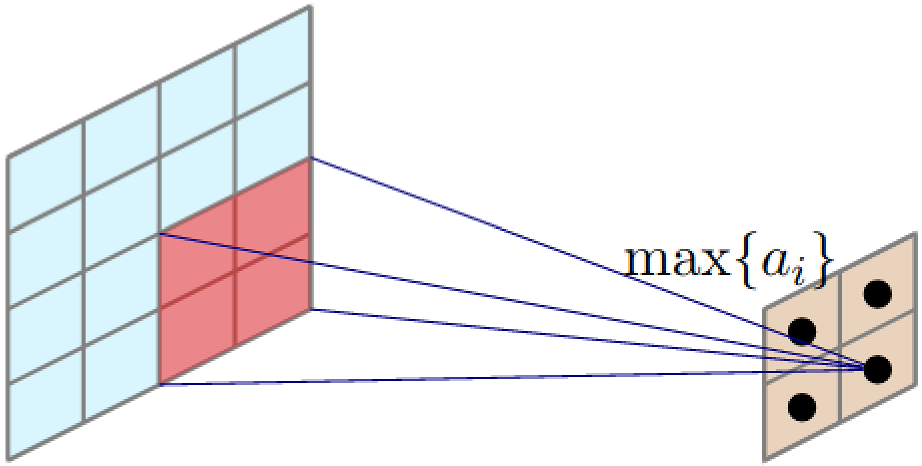
# The convolutional layer: Pooling layer



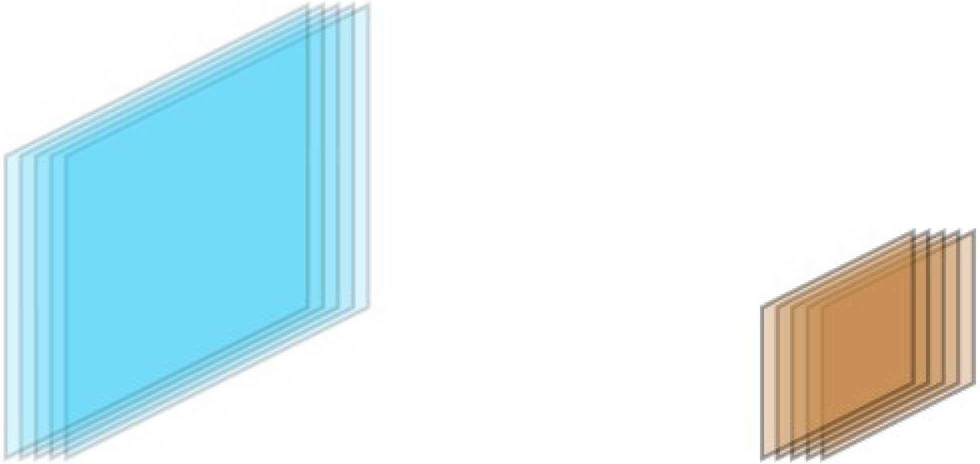
# The convolutional layer: Pooling layer



# The convolutional layer: Pooling layer

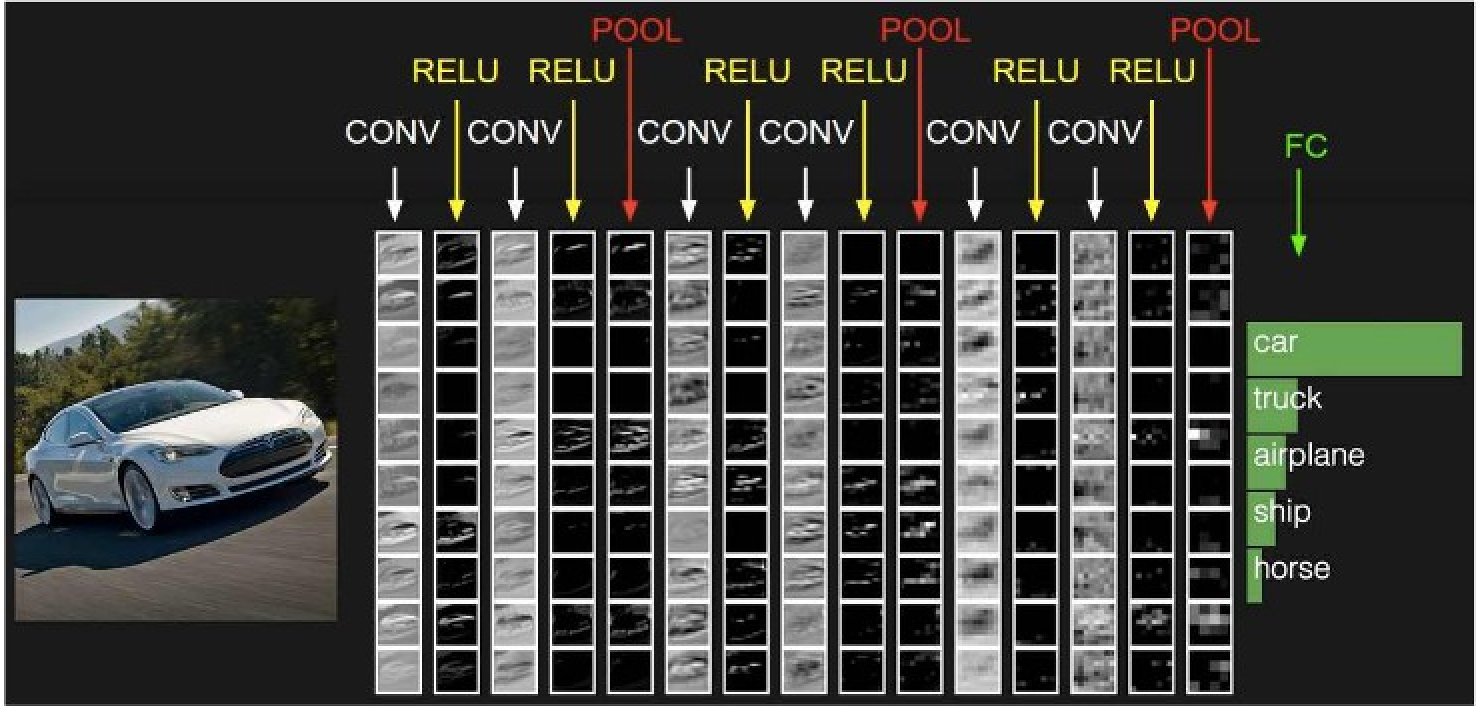


# The convolutional layer: Pooling layer



- We have multiple feature maps, and get an equal number of subsampled maps

# The convolutional layer: output size

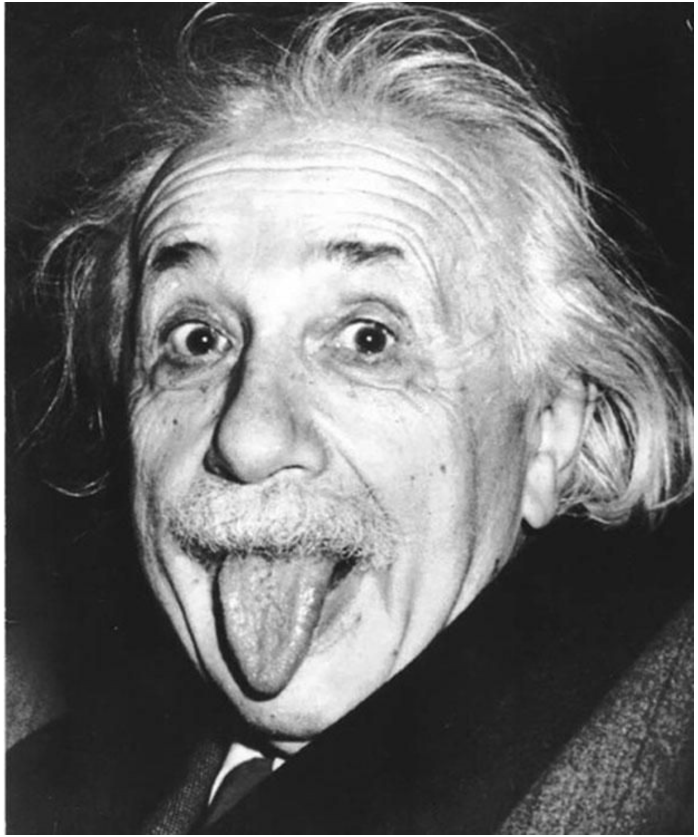


# References

Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position,
Kunihiko Fukushima
NHK BroadcastingScienceResearchLaboratories,Kinuta, Setagaya, Tokyo, Japan, Bio Cybernetics, (4) 1980, 193-202

Deep Learning
An MIT Press book
Ian Goodfellow and Yoshua Bengio and Aaron Courville
http://www.deeplearningbook.org/contents/convnets.html

Stanford Course in Convolutional NN for Visual Representation (2017)
http://cs231n.github.io/convolutional-networks/
2016 course: https://youtu.be/LxfUGhug-iQ



# Questions?