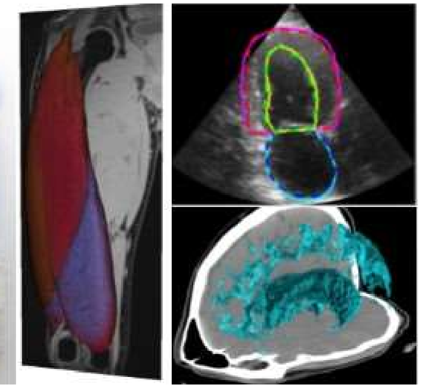
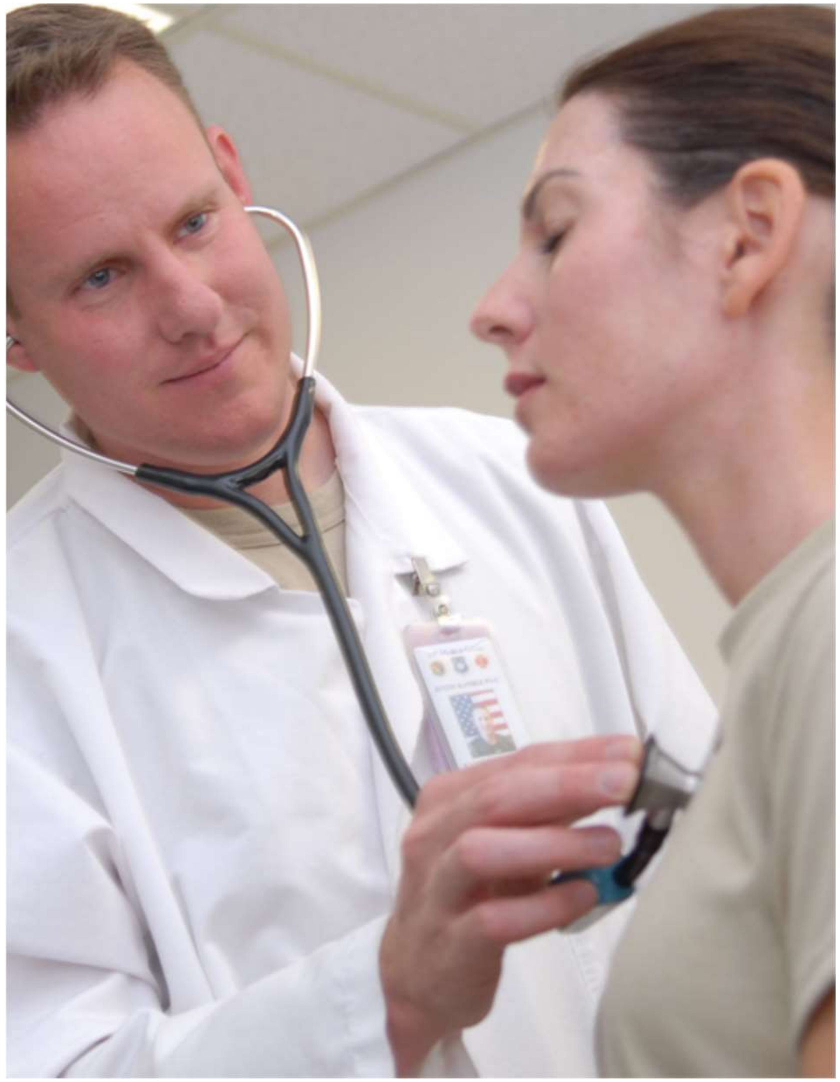


Basics of deep learning

By

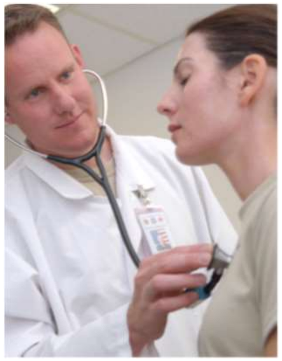
Pierre-Marc Jodoin and Christian Desrosiers

Lets start with a simple example



From Wikimedia Commons
the free media repository

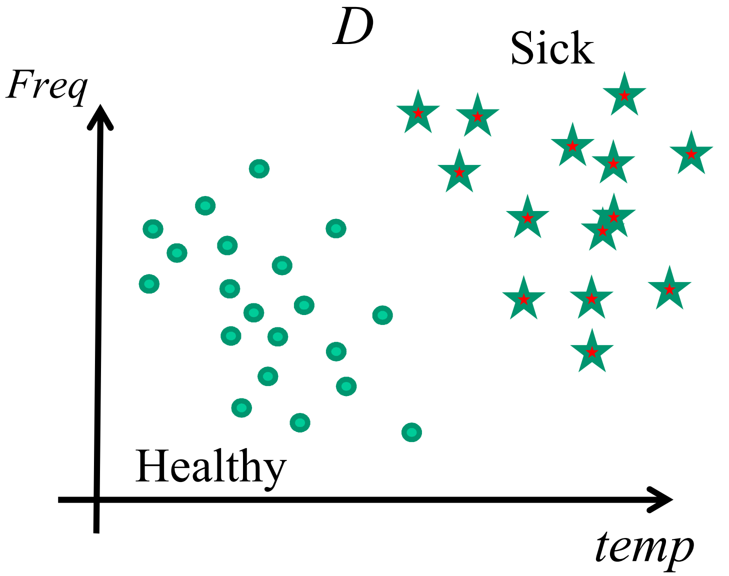
Lets start with a simple example



D

	(temp, freq)	**diagnostic**
Patient 1	(37.5, 72)	Healthy
Patient 2	(39.1, 103)	Sick
Patient 3	(38.3, 100)	Sick
	(…)	…
Patient N	(36.7, 88)	Healthy

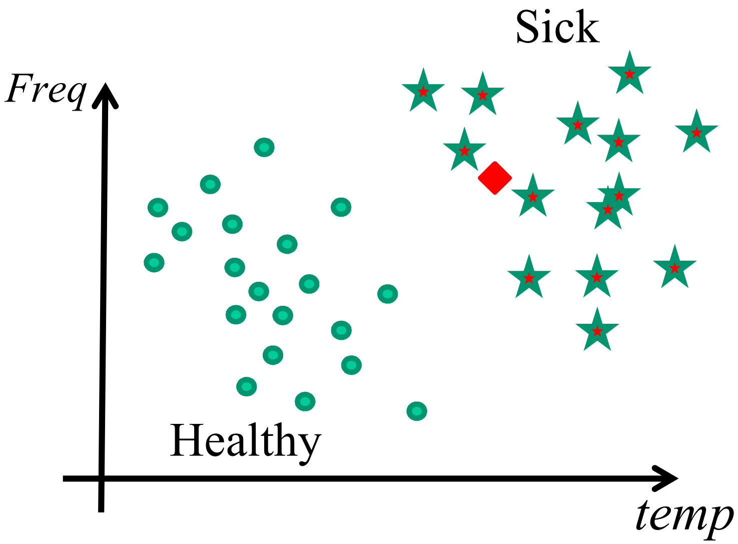
$\vec{x}$ t



Lets start with a simple example

A new patient comes to the hospital
How can we determine if he is sick or not?

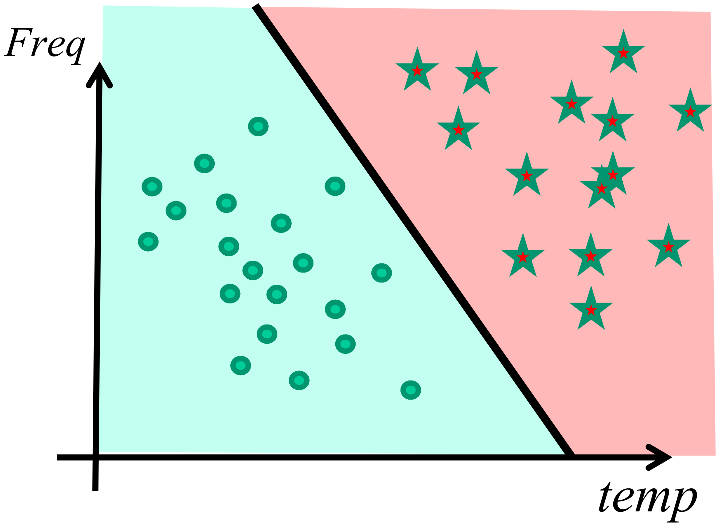
From Wikimedia Commons
the free media repository



Solution


From Wikimedia Commons
the free media repository

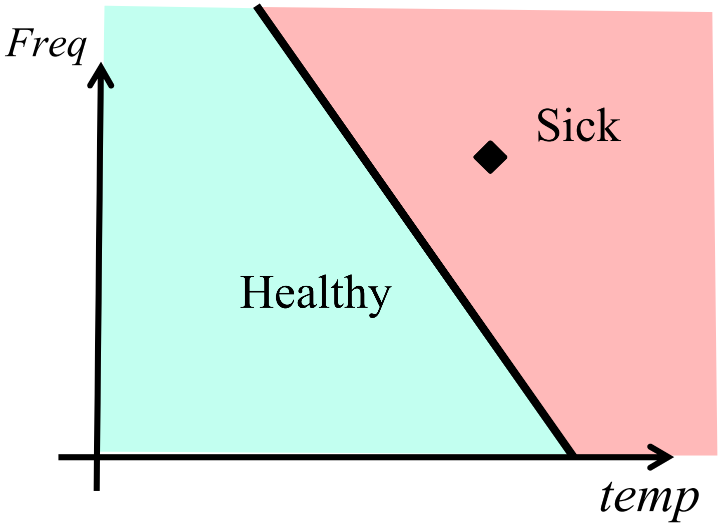
Divide the feature space in 2 regions : sick and healthy



Solution

From Wikimedia Commons
the free media repository

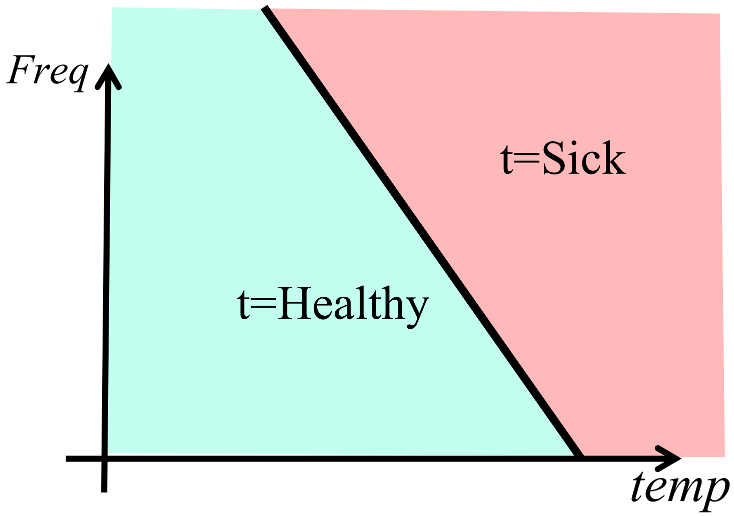
Divide the feature space in 2 regions : sick and healthy



More formally

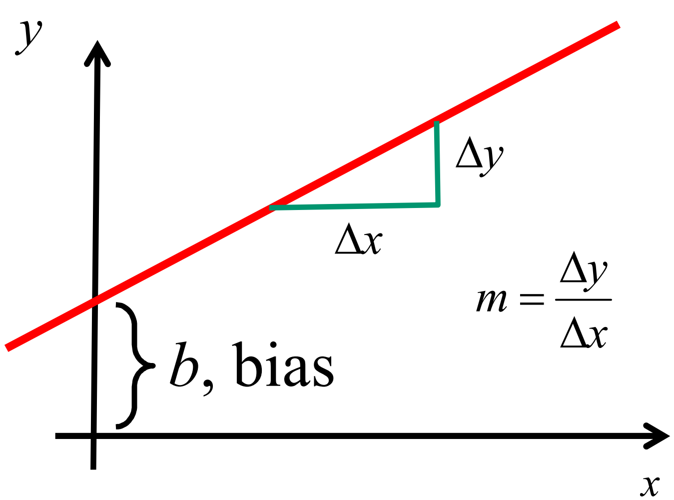
From Wikimedia Commons
the free media repository

$$y(\vec{x}) = \begin{cases} Healthy \text{ if } \vec{x} \text{ is in the blue region} \\ Sick \text{ otherwise} \end{cases}$$



How to split the feature space?

Definition … a line!

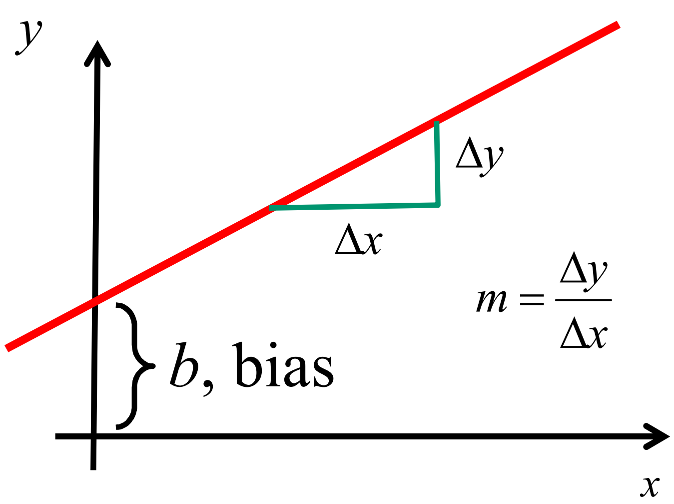


$$y = mx + b$$

slope

bias

Definition … a line!



$$y = mx + b$$

$$y = \frac{\Delta y}{\Delta x} x + b$$

$$y \Delta x = \Delta y x + b \Delta x$$

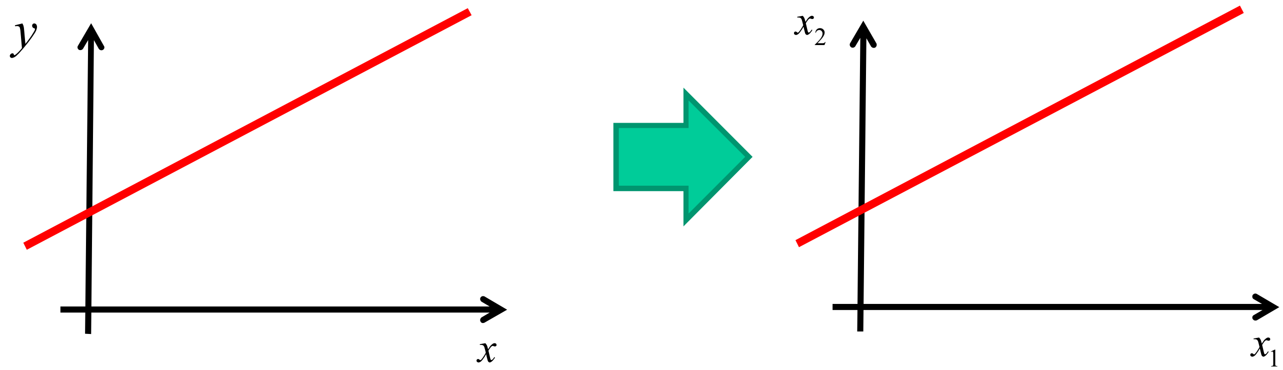
$$0 = \Delta y x - \Delta x y + b \Delta x$$

Rename variables

$$0 = \underbrace{\Delta y}_{w_1} x - \underbrace{\Delta x}_{w_2} y + \underbrace{b \Delta x}_{w_0}$$

Rename variables

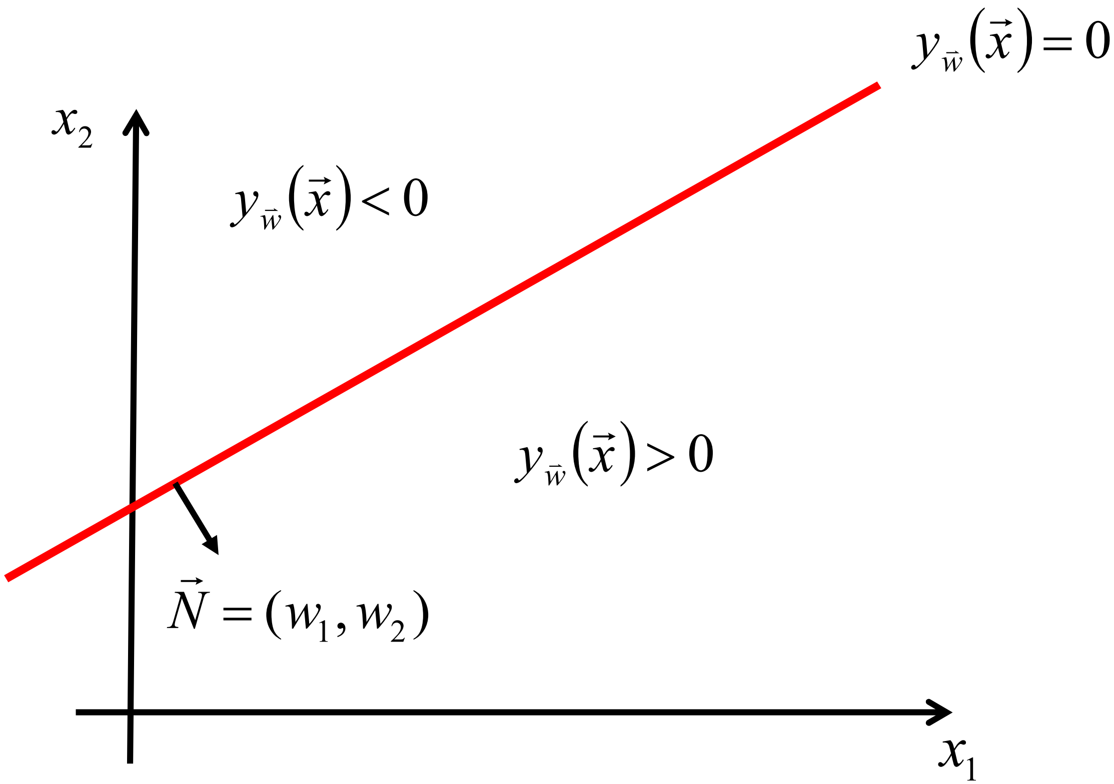
$$0 = w_1 x + w_2 y + w_0$$



$$0 = w_1 x_1 + w_2 x_2 + w_0$$

Classification function

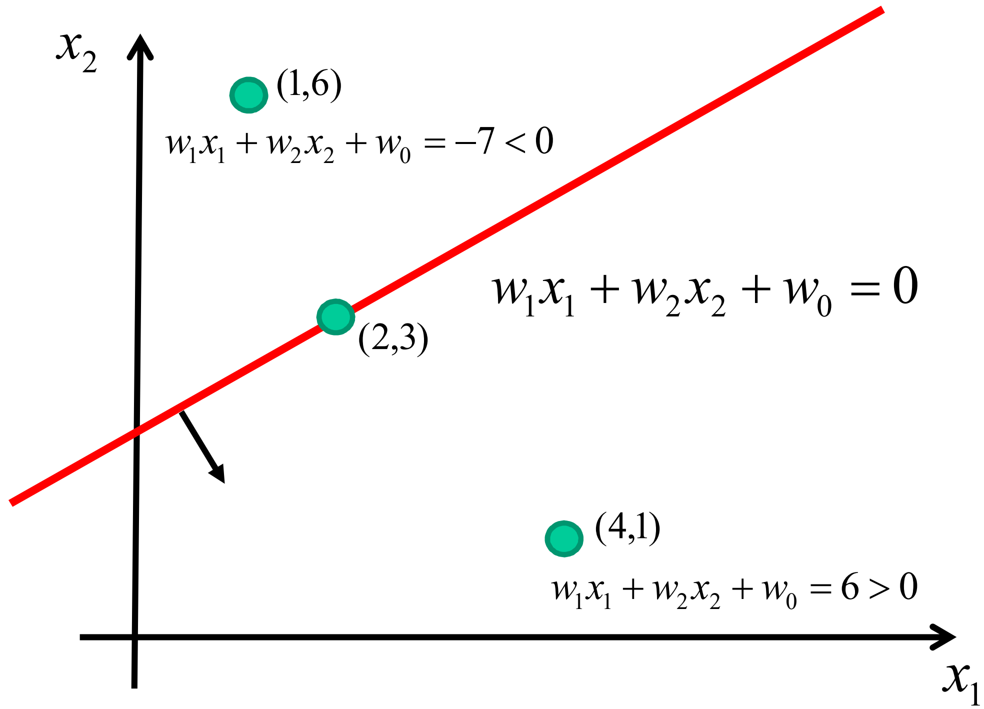
$$y_{\vec{w}}(\vec{x}) = w_1 x_1 + w_2 x_2 + w_0$$



Classification function

$$y_{\vec{w}}(\vec{x}) = w_1 x_1 + w_2 x_2 + w_0$$

$w_1 = 1.0$
$w_2 = -2.0$
$w_0 = 4.0$

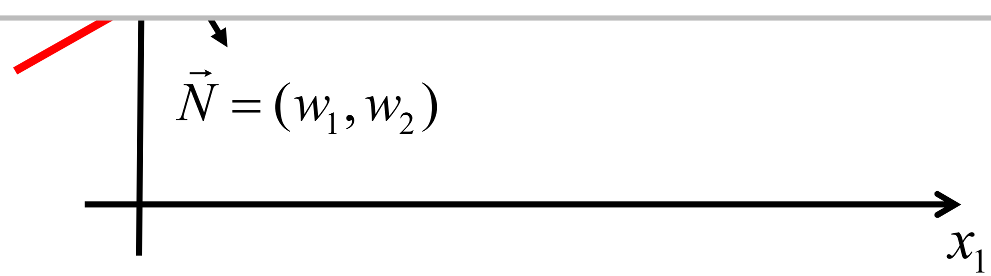


Classification function

$$y_{\vec{w}}\left(\vec{x}\right) = w_1 x_1 + w_2 x_2 + w_0$$

$$= \underbrace{\left(w_0, w_1, w_2\right)}_{\vec{w}} \cdot \underbrace{\left(1, x_1, x_2\right)}_{\vec{x}\,'}$$

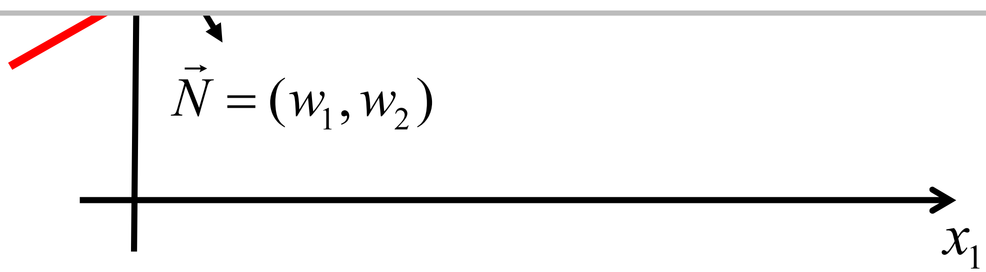
DOT product



Classification function

$$
\begin{aligned}
y_{\vec{w}}(\vec{x}) &= w_1 x_1 + w_2 x_2 + w_0 \\
&= (w_0, w_1, w_2) \cdot (1, x_1, x_2) \\
&= \vec{w}^T \vec{x}\,'
\end{aligned}
$$

DOT product



To simplify notation

linear classification = dot product with bias included

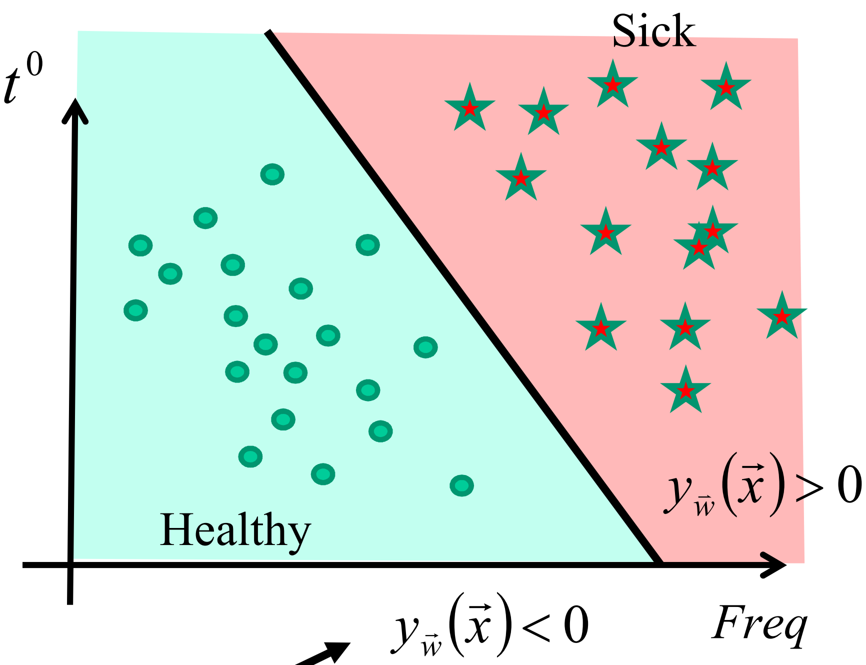
$$y_{\vec{w}}\left(\vec{x}\right) = \vec{w}^{T}\vec{x}$$

Learning

With the **training dataset** D

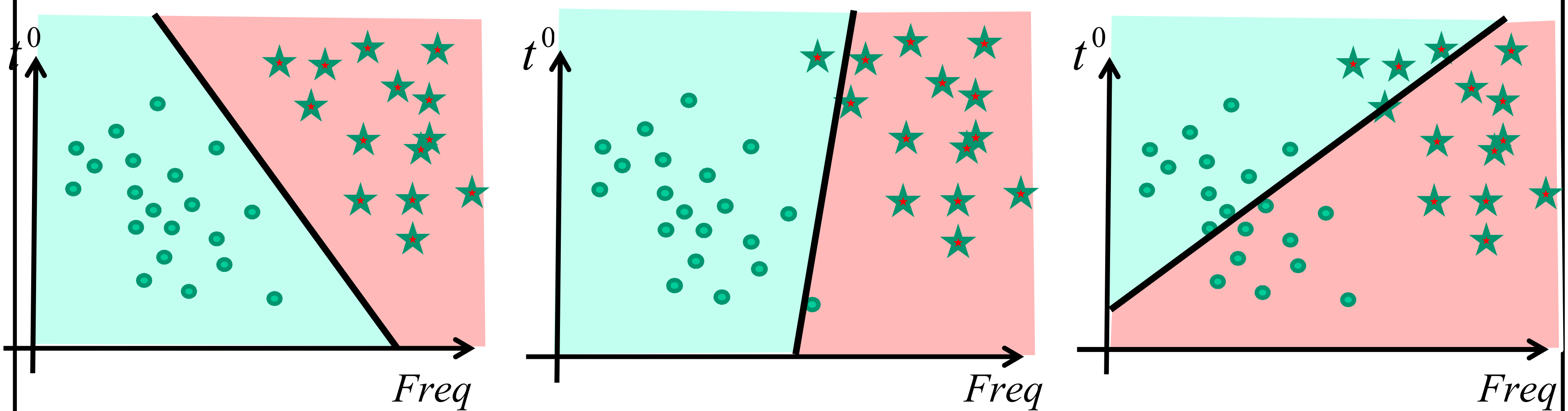
the GOAL is to

find the parameters (w_0, w_1, w_2) that would best separate the two classes.



How do we know if a model is good?

Loss function



$L\left(y_{\vec{w}}(\vec{x}), D\right) \approx 0$ $L\left(y_{\vec{w}}(\vec{x}), D\right) > 0$ $L\left(y_{\vec{w}}(\vec{x}), D\right) >> 0$

Good! **Medium** **BAD!**

So far…

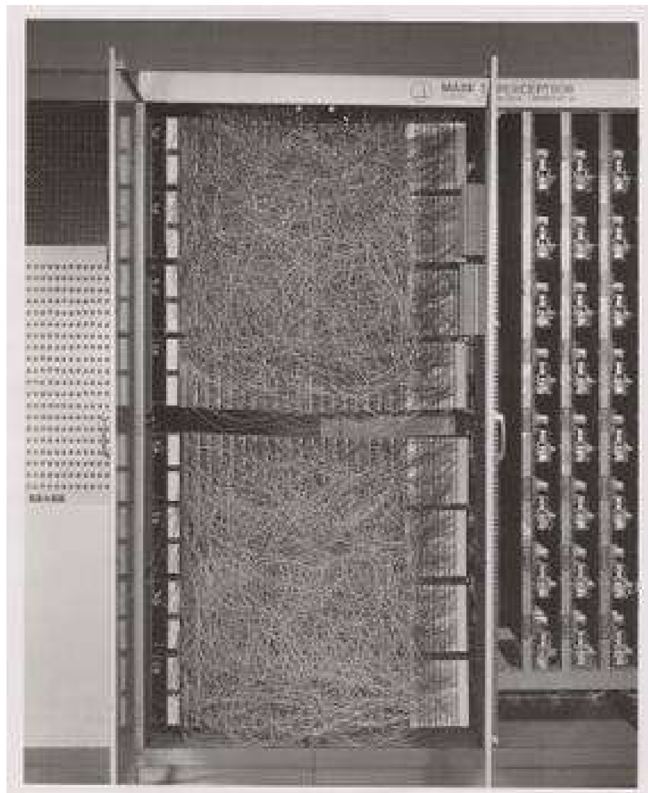
1. Training dataset: D
2. Classification function (a line in 2D) : $y_{\vec{w}}(\vec{x}) = w_1 x_1 + w_2 x_2 + w_0$
3. Loss function: $L(y_{\vec{w}}(\vec{x}), D)$

4. Training : find (w_0, w_1, w_2) that minimize $L(y_{\vec{w}}(\vec{x}), D)$

Today

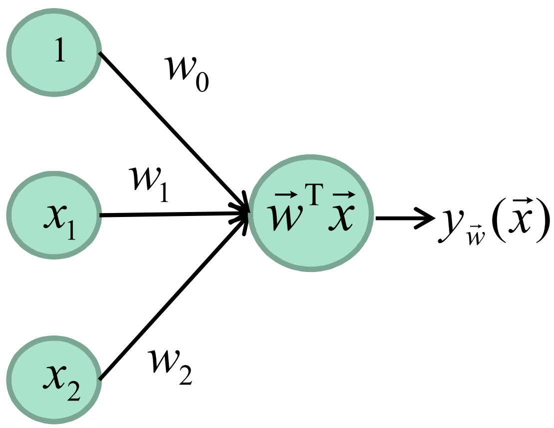
Perceptron
Logistic regression
Multi-layer perceptron
Conv Nets

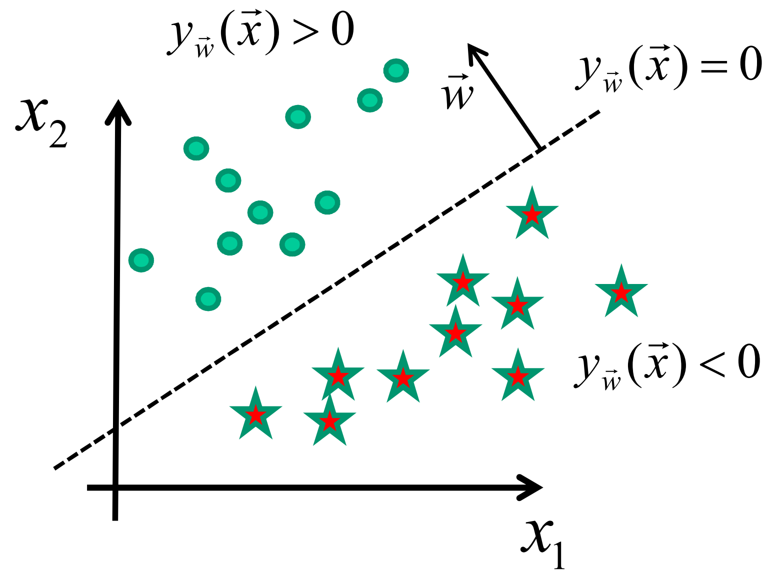
Perceptron



Rosenblatt, Frank (1958), **The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain**, Psychological Review, v65, No. 6, pp. 386–408

Perceptron
(2D and 2 classes)

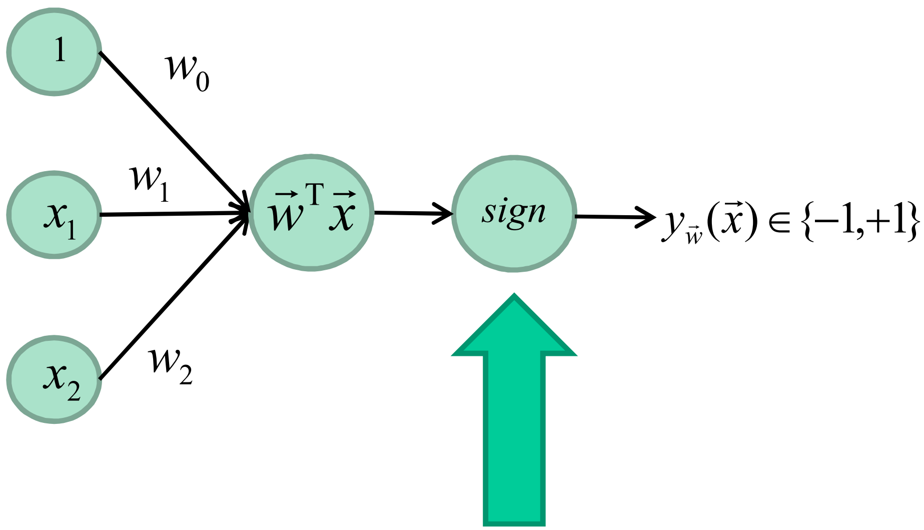




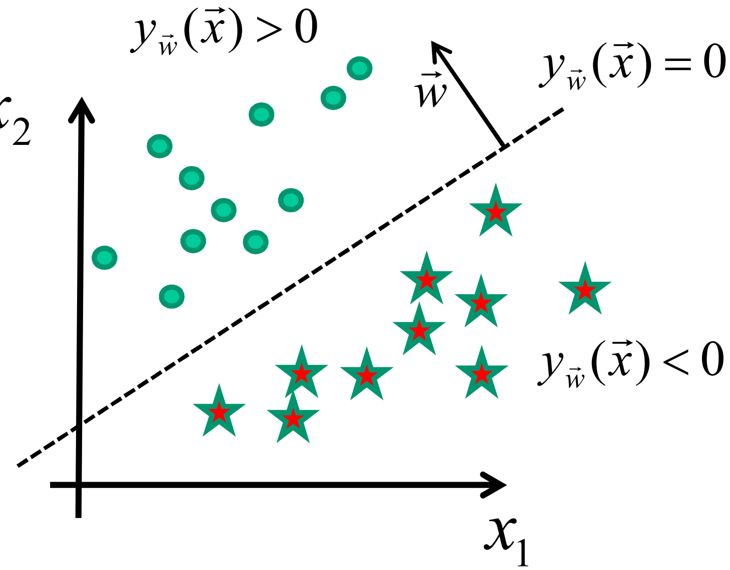
$$y_{\vec{w}}(\vec{x}) = w_0 + w_1 x_1 + w_2 x_1$$
$$= \vec{w}^{\mathrm{T}} \vec{x}$$

Perceptron

(2D and 2 classes)



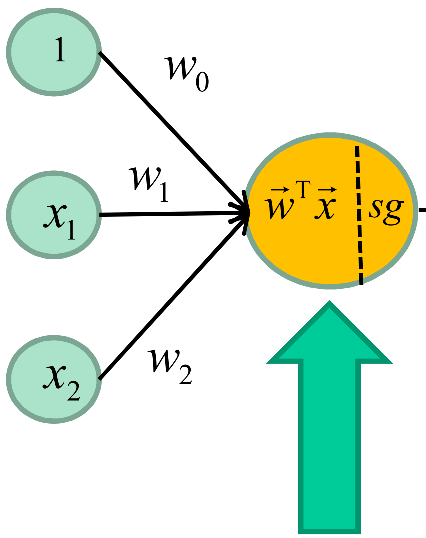
Activation function



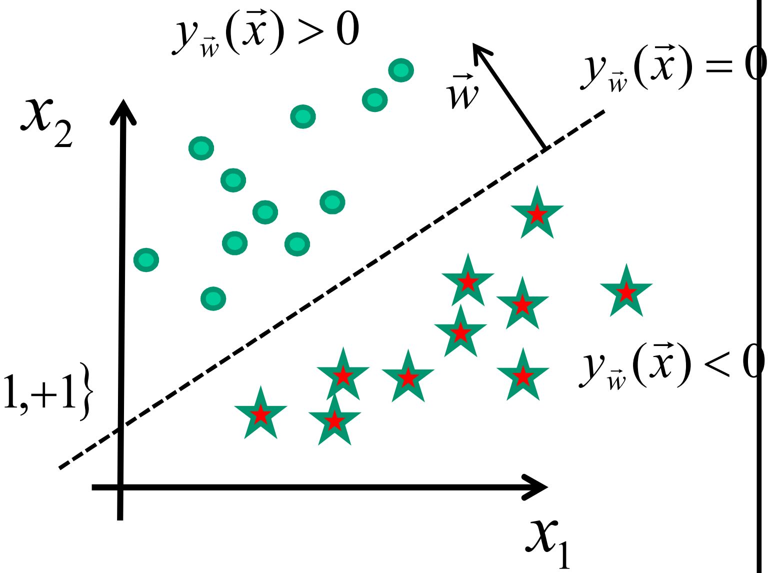
$$y_{\vec{w}}(\vec{x}) = sign\left(\vec{w}^{\mathrm{T}}\vec{x}\right)$$

Perceptron

(2D and 2 classes)



$$y_{\vec{w}}(\vec{x}) = sign(\vec{w}^{\mathrm{T}}\vec{x}) \in \{-1,+1\}$$



Neuron

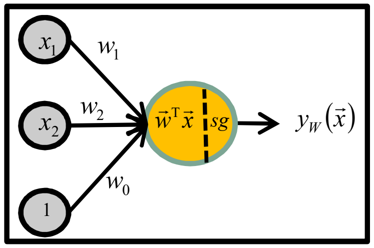
Dot product + activation function

So far…

1. Training dataset: D
2. Classification function (a line in 2D) : $y_{\vec{w}}(\vec{x}) = w_1 x_1 + w_2 x_2 + w_0$
3. Loss function: $L(y_{\vec{w}}(\vec{x}), D)$

So far…

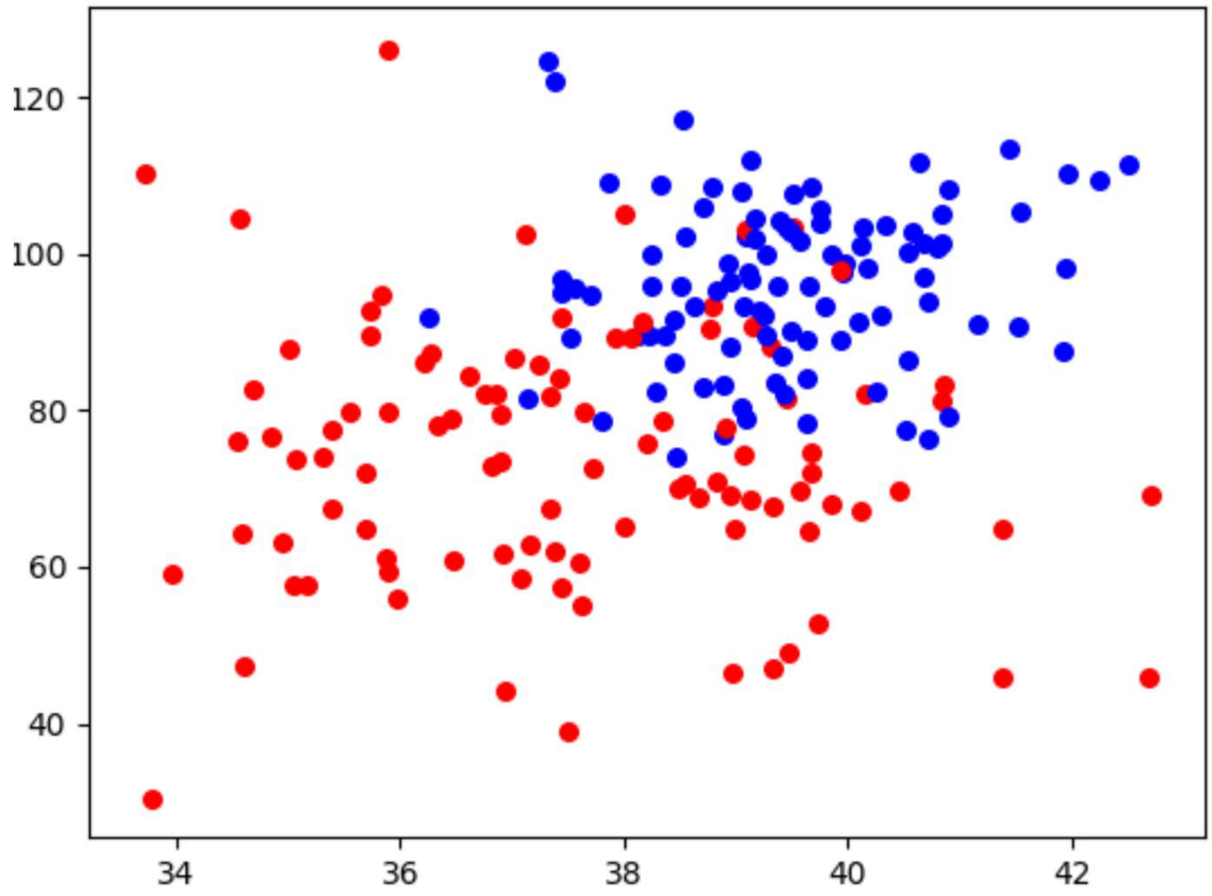
1. Training dataset: D
2. Classification function (a line in 2D) :
3. Loss function: $L\left(y_{\vec{w}}(\vec{x}), D\right)$



4. Training : find (w_0, w_1, w_2) that minimize $L\left(y_{\vec{w}}(\vec{x}), D\right)$

Linear classifiers have **limits**

Non-linearly separable training data



Linear classifier = **<u>large error rate</u>**

Non-linearly separable training data

Three classical solutions

1. Acquire more observations
2. Use a non-linear classifier
3. Transform the data

Non-linearly separable training data

Three classical solutions

1. **Acquire more observations**
2. Use a non-linear classifier
3. Transform the data

Acquire more data

D

	(temp, freq)	diagnostic
Patient 1	(37.5, 72)	healthy
Patient 2	(39.1, 103)	sick
Patient 3	(38.3, 100)	sick
	(…)	…
Patient N	(36.7, 88)	healthy

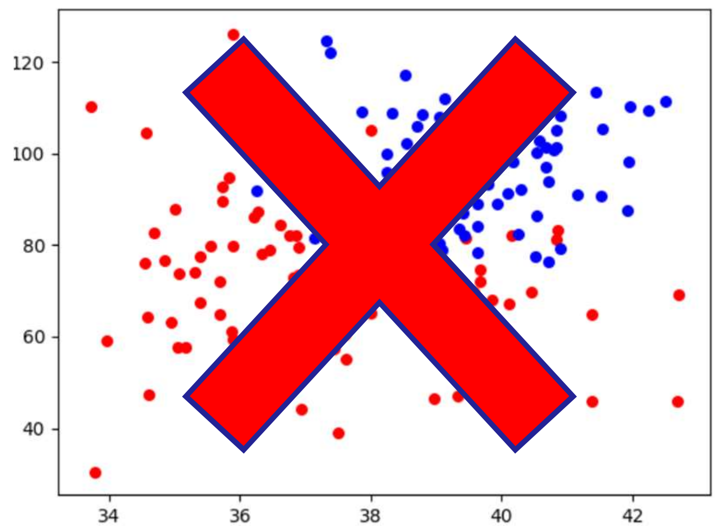
$\vec{x}$ t

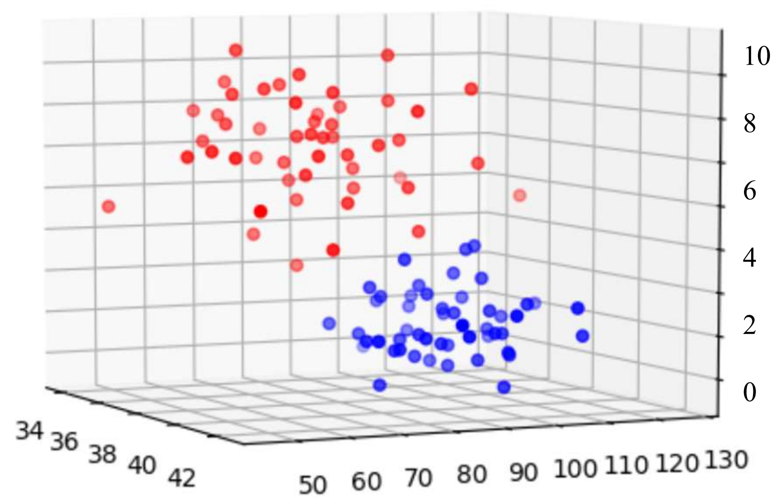
D

	(temp, freq, headache)	Diagnostic
Patient 1	(37.5, 72, 2)	healthy
Patient 2	(39.1, 103, 8)	sick
Patient 3	(38.3, 100, 6)	sick
	(…)	…
Patient N	(36.7, 88, 0)	healthy

$\vec{x}$ t

Non-linearly separable training data





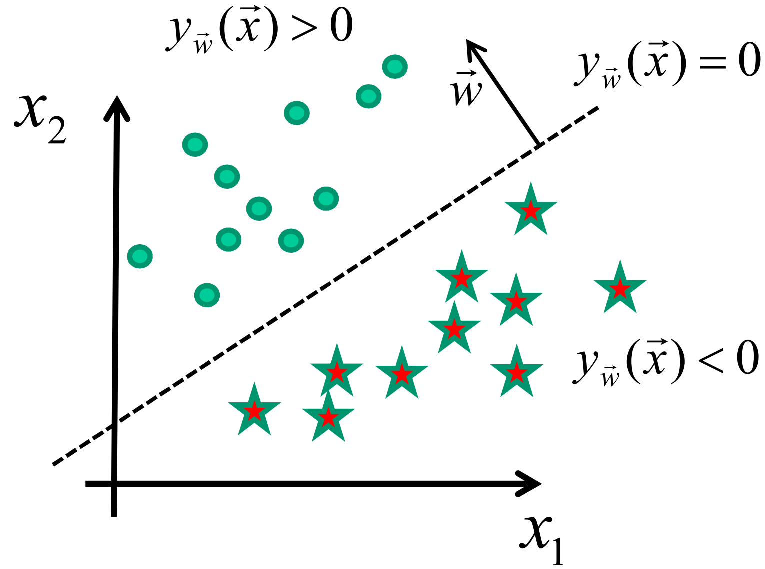
$$y_{\vec{w}}\left(\vec{x}\right) = w_1x_1 + w_2x_2 + w_0$$

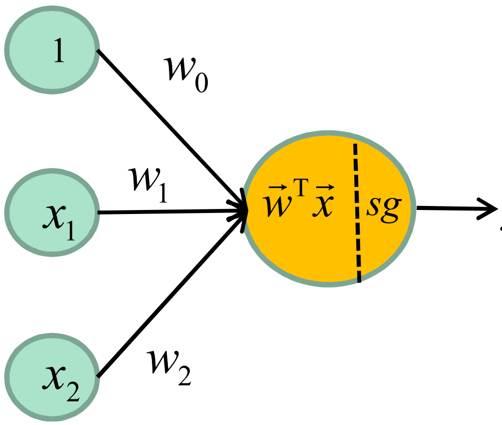
(line)

$$y_{\vec{w}}\left(\vec{x}\right) = w_1x_1 + w_2x_2 + w_3x_3 + w_0$$

(plane)

Perceptron
(2D and 2 classes)



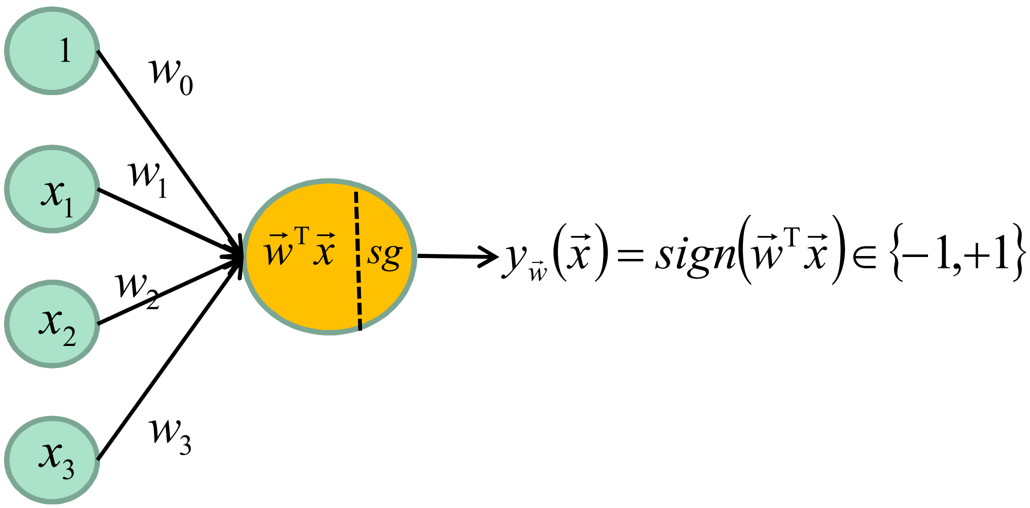


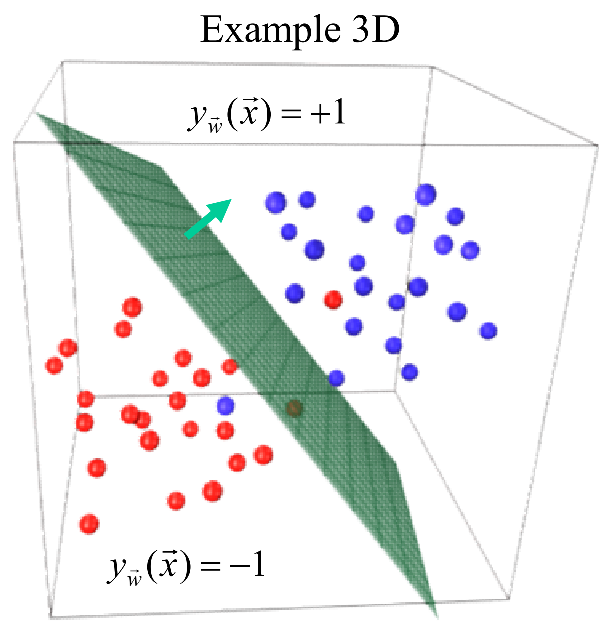
$y_{\vec{w}}(\vec{x}) = sign(\vec{w}^{\mathrm{T}}\vec{x}) \in \{-1,+1\}$

$$y_{\vec{w}}(\vec{x}) = w_1 x_1 + w_2 x_2 + w_0$$

(line)

Perceptron
(3D and 2 classes)



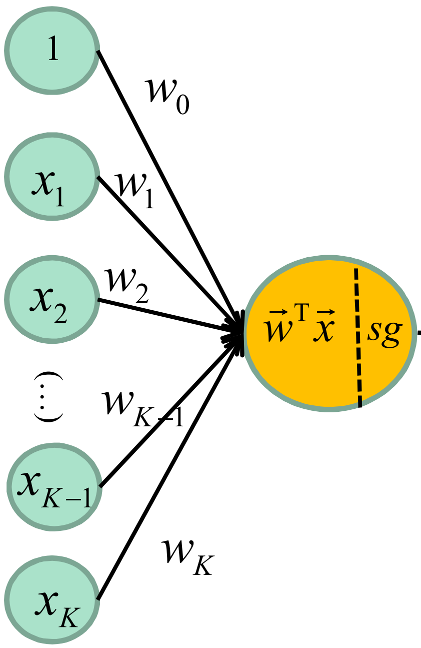


$$y_{\vec{w}}(\vec{x}) = w_1x_1 + w_2x_2 + w_3x_3 + w_0$$

(plane)

Perceptron

(K-D and 2 classes)



$$y_{\vec{w}}(\vec{x}) = sign\left(\vec{w}^{\mathrm{T}}\vec{x}\right) \in \{-1,+1\}$$

$$y_{\vec{w}}(\vec{x}) = w_1 x_1 + w_2 x_2 + w_3 x_3 + \ldots + w_K x_K + w_0$$

(hyperplane)

Learning a machine

<u>The goal</u>: with a set of training data $D = \{(\vec{x}_1, t_1), (\vec{x}_2, t_2), \ldots, (\vec{x}_N, t_N)\}$, estimate $\vec{w}$ so:

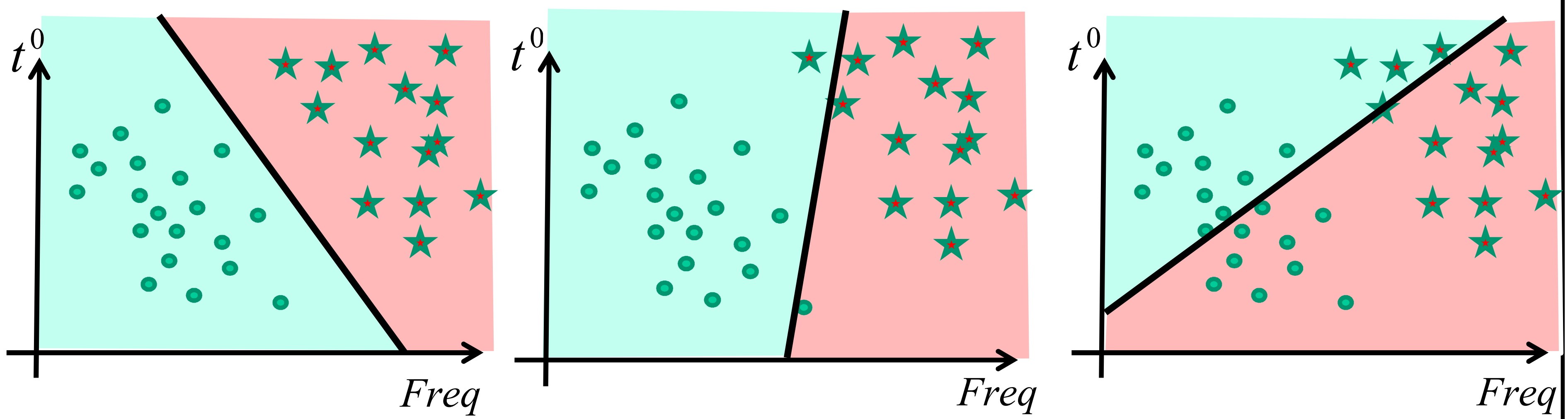
$$y_{\vec{w}}(\vec{x}_n) = t_n \qquad \forall n$$

In other words, minimize the **training loss**

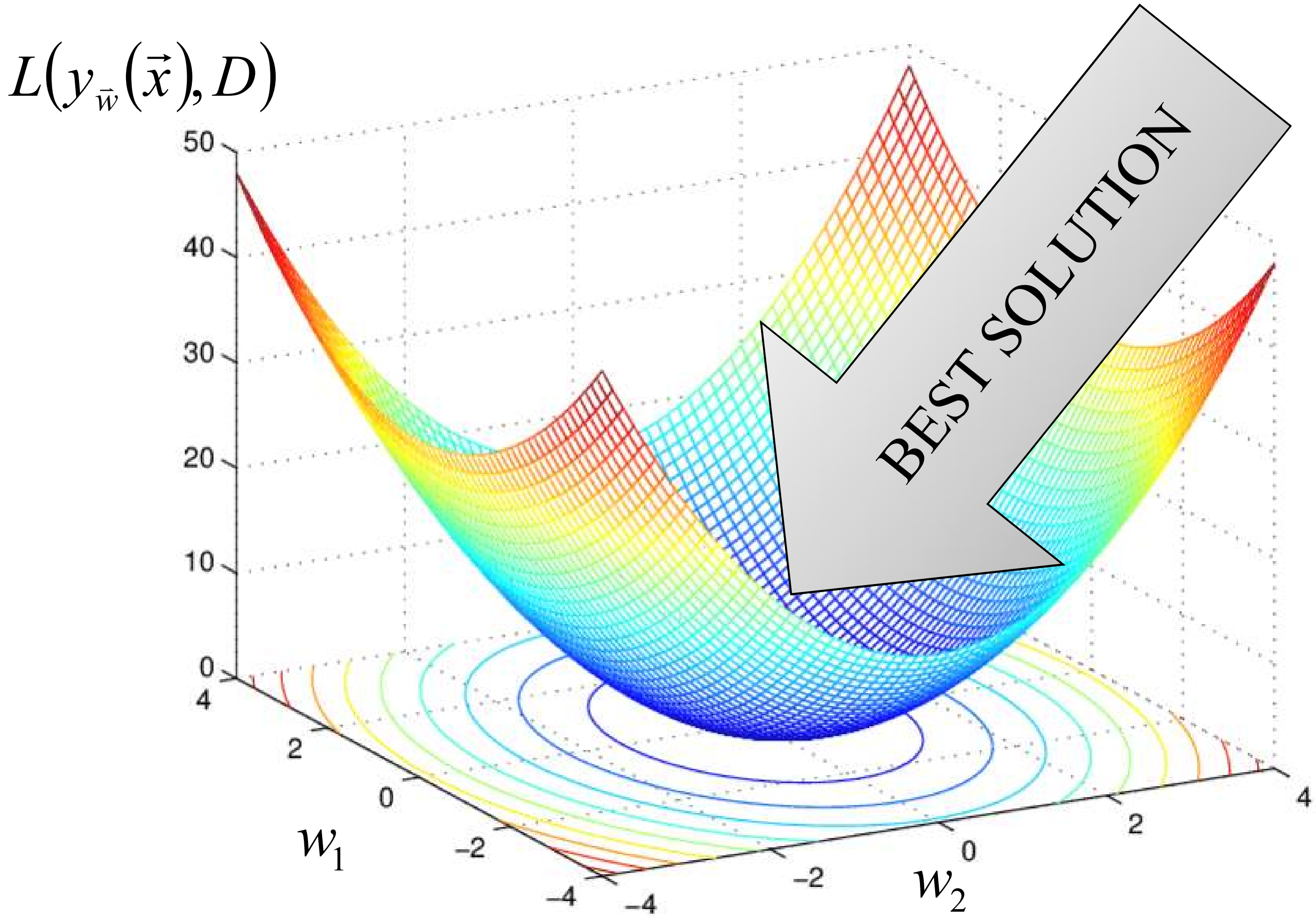
$$L(y_{\vec{w}}(\vec{x}), D) = \sum_{n=1}^{N} l(y_{\vec{w}}(\vec{x}_n), t_n)$$

Optimization problem

Loss function



$L(y_{\vec{w}}(\vec{x}), D) \approx 0$ $\quad$ $L(y_{\vec{w}}(\vec{x}), D) > 0$ $\quad$ $L(y_{\vec{w}}(\vec{x}), D) >> 0$

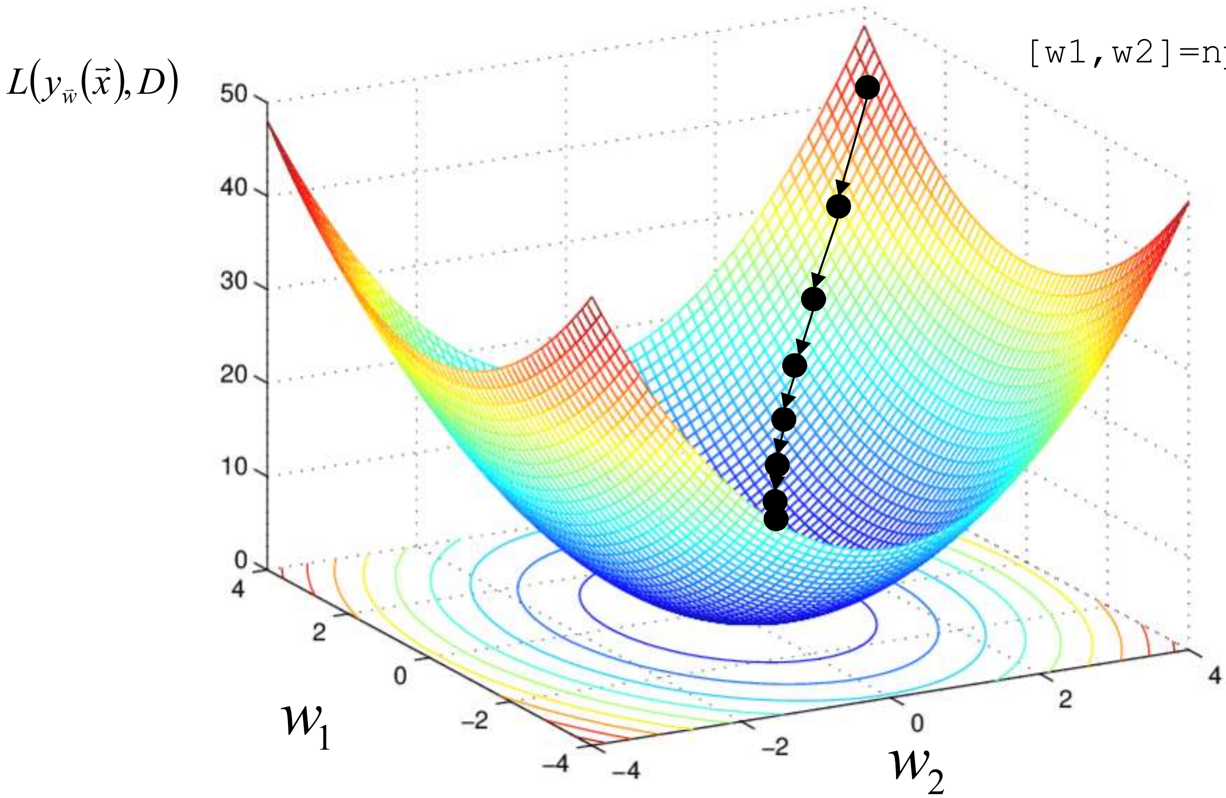

$L(y_{\vec{w}}(\vec{x}), D)$
50
40
30
20
10
0
4
2
0
-2
-4
w_1
-4
-2
0
2
4
w_2
BEST SOLUTION

Perceptron

Question: how to find the best solution? $\nabla L\left(y_{\vec{w}}\left(\vec{x}\right), D\right) = 0$

Random initialization

```
[w1,w2]=np.random.randn(2)
```



Gradient descent

Question: how to find the best solution? $\nabla L\left(y_{\vec{w}}\left(\vec{x}\right), D\right) = 0$

$$\vec{w}^{[k+1]} = \vec{w}^{[k]} - \eta \nabla L\left(y_{\vec{w}^{[k]}}\left(\vec{x}\right), D\right)$$

Gradient of the loss function

Learning rate

Perceptron Criterion (loss)

Observation

A **wrongly classified sample** is when

$$\vec{w}^{\mathrm{T}}\vec{x}_n > 0 \text{ et } t_n = -1$$

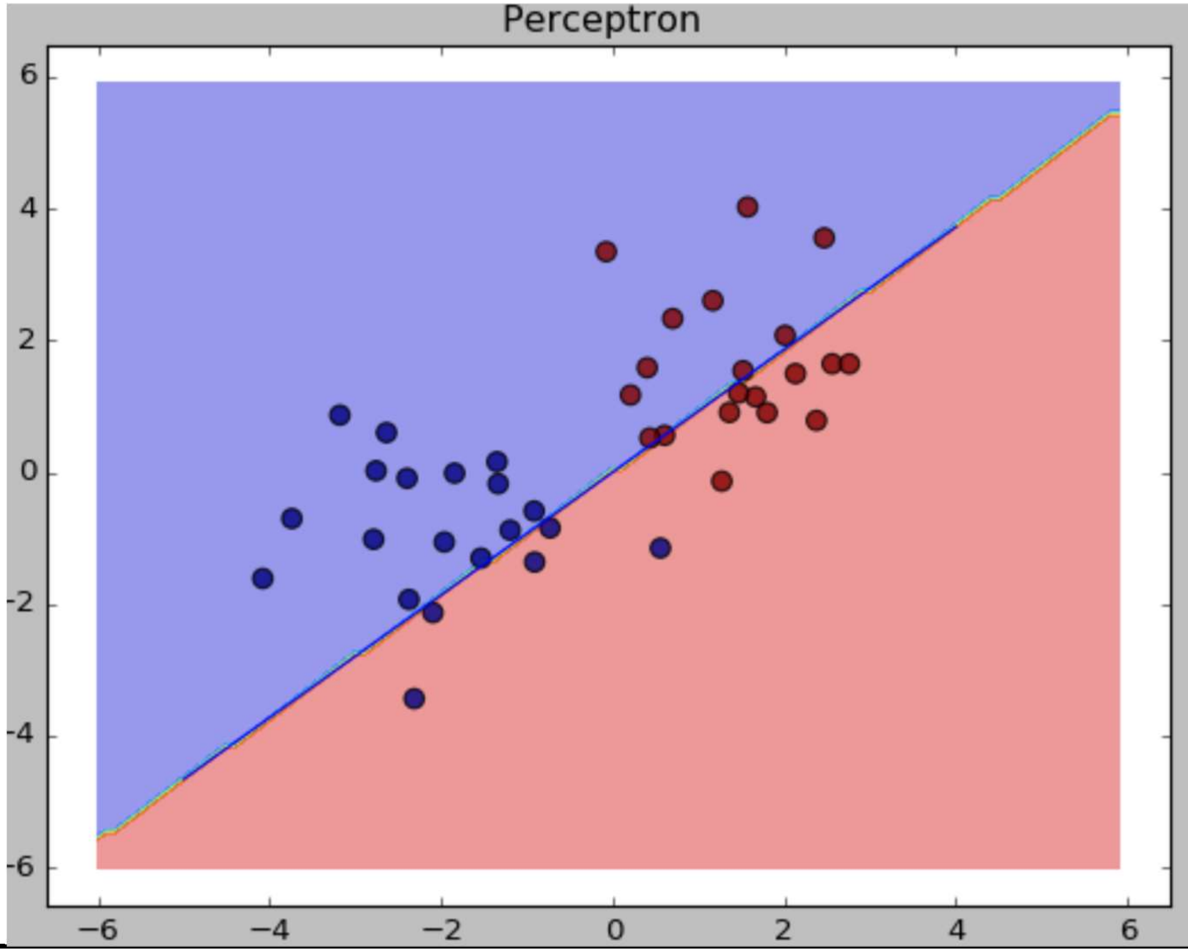
or

$$\vec{w}^{\mathrm{T}}\vec{x}_n < 0 \text{ et } t_n = +1.$$

Consequently $-\vec{w}^{\mathrm{T}}\vec{x}_n t_n$ is **ALWAYS positive for wrongly classified samples**

Perceptron Criterion (loss)

$$L\left(y_{\vec{w}}(\vec{x}), D\right) = \sum_{\vec{x}_n \in V} -\vec{w}^{\mathrm{T}} \vec{x}_n t_n$$ where V is the set of wrongly classified samples



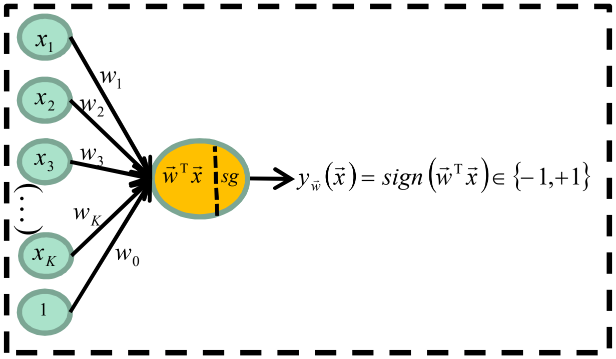
$$L\left(y_{\vec{w}}(\vec{x}), D\right) = 464.15$$

So far…

1. Training dataset: D
2. Linear classification function: $y_{\vec{w}}(\vec{x}) = w_1 x_1 + w_2 x_2 + \ldots + w_M x_M + w_0$
3. Loss function: $L(y_{\vec{w}}(\vec{x}), D) = \sum_{\vec{x}_n \in V} -\vec{w}^{\mathrm{T}} \vec{x}_n t_n$

So far…

1. Training dataset: D
2. Linear classification function:
3. Loss function: $L(y_{\vec{w}}(\vec{x}), D) = \sum_{\vec{x}_n \in V} -\vec{w}^{\mathrm{T}} \vec{x}_n t_n$



4. Training : find $\vec{w}$ that minimizes $L(y_{\vec{w}}(\vec{x}), D)$

$$\vec{w} = \arg\min_{\vec{w}} L(y_{\vec{w}}(\vec{x}), D)$$

$$\nabla L(y_{\vec{w}}(\vec{x}), D) = 0$$

Optimisation

$$\vec{\mathrm{w}}^{[k+1]} = \vec{\mathrm{w}}^{[k]} - \eta^{[k]} \nabla L$$

Gradient of the loss function

learning rate

Stochastic gradient descent (SGD)

Init $\vec{\mathrm{w}}$
k=0
DO k=k+1
  FOR n = 1 to N
    $\vec{w} = \vec{w} - \eta^{[k]} \nabla L(\vec{x}_n)$
UNTIL every data is well classified or k== MAX_ITER

Perceptron gradient descent

$$L\left(y_{\vec{w}}(\vec{x}), D\right) = \sum_{\vec{x}_n \in V} -\vec{w}^{\mathrm{T}} \vec{x}_n t_n$$

$$\nabla L\left(y_{\vec{w}}(\vec{x}), D\right) = \sum_{\vec{x}_n \in V} -t_n \vec{x}_n$$

Stochastic gradient descent (SGD)

Init $\vec{w}$
k=0
DO k=k+1
    FOR n = 1 to N
        IF $\vec{w}^{\mathrm{T}} \vec{x}_n t_n < 0$ THEN /* wrongly classified */
            $\vec{w} = \vec{w} + \eta t_n \vec{x}_n$

UNTIL every data is well classified OR k=k_MAX

NOTE : **learning rate** η :

- **Too low** => slow convergence
- **Too large** => might not converge (even diverge)
- Can **decrease** at each iteration (e.g. $\eta^{[k]} = cst / k$)

Similar loss functions

$$L\left(y_{\vec{w}}(\vec{x}), D\right) = \sum_{\vec{x}_n \in V} -\vec{w}^{\mathrm{T}} \vec{x}_n t_n$$

where V *i*s the set of wrongly classified samples

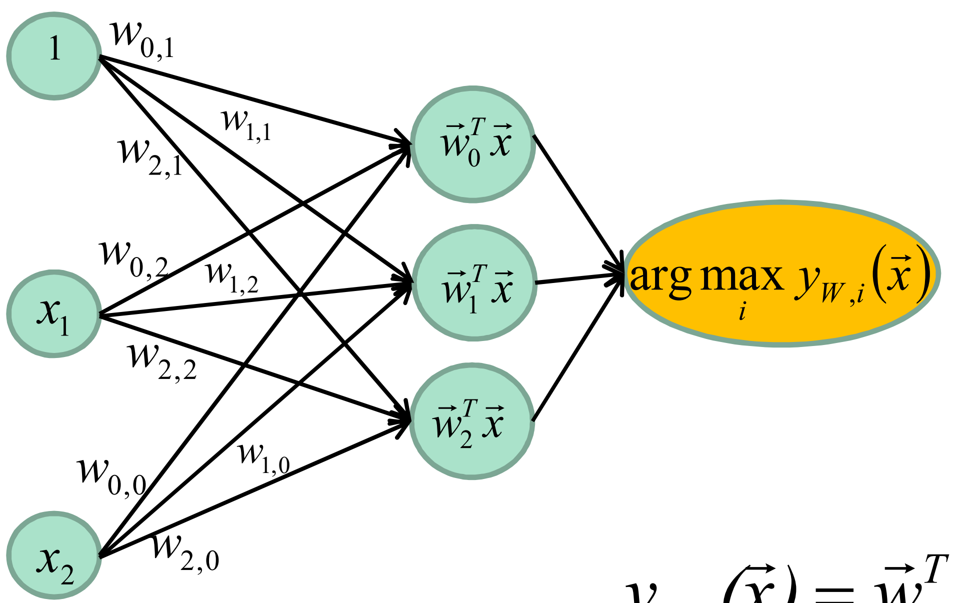
$$L\left(y_{\vec{w}}(\vec{x}), D\right) = \sum_{n=1}^{N} \max\left(0, -t_n \vec{w}^T \vec{x}_n\right)$$

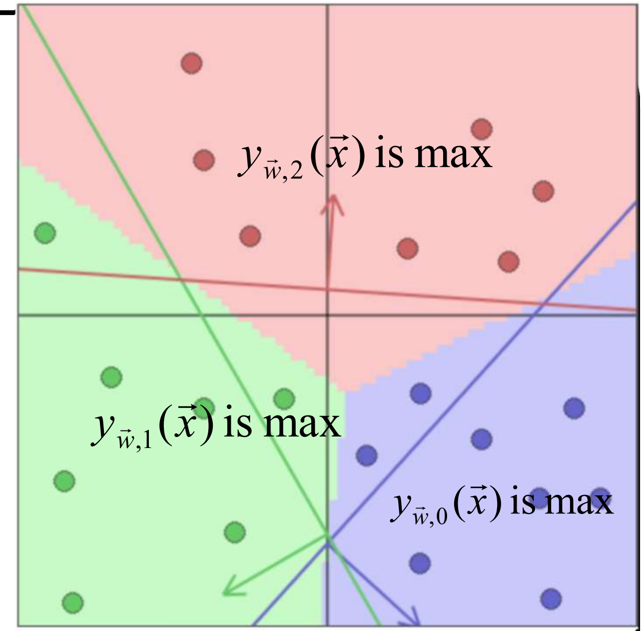
$$L\left(y_{\vec{w}}(\vec{x}), D\right) = \sum_{n=1}^{N} \max\left(0, 1 - t_n \vec{w}^T \vec{x}_n\right)$$

"Hinge Loss" or "SVM" Loss

Multiclass Perceptron

(2D and 3 classes)





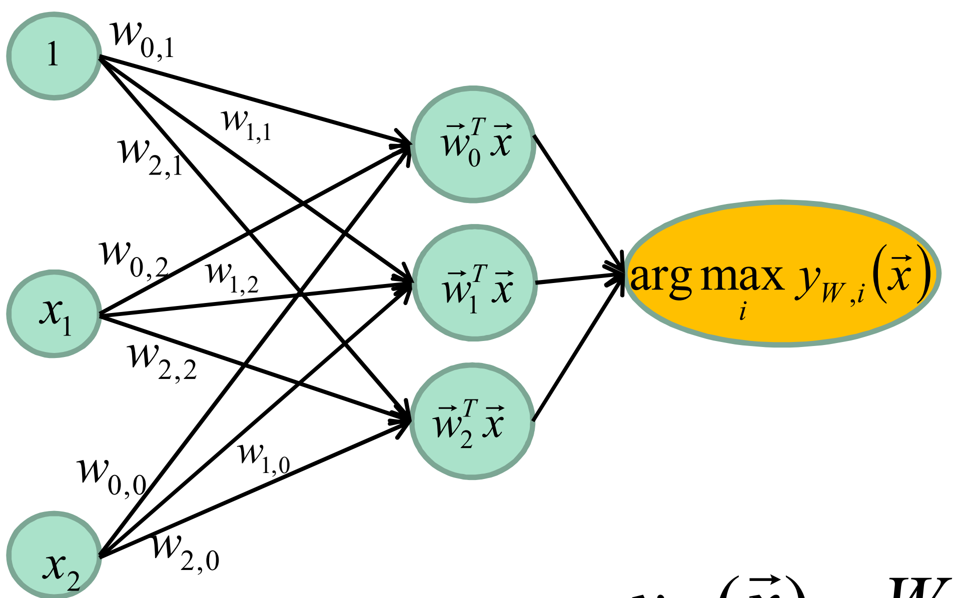
$$y_{\vec{w},0}(\vec{x}) = \vec{w}_0^T \vec{x} = w_{0,0} + w_{0,1}x_1 + w_{0,2}x_2$$

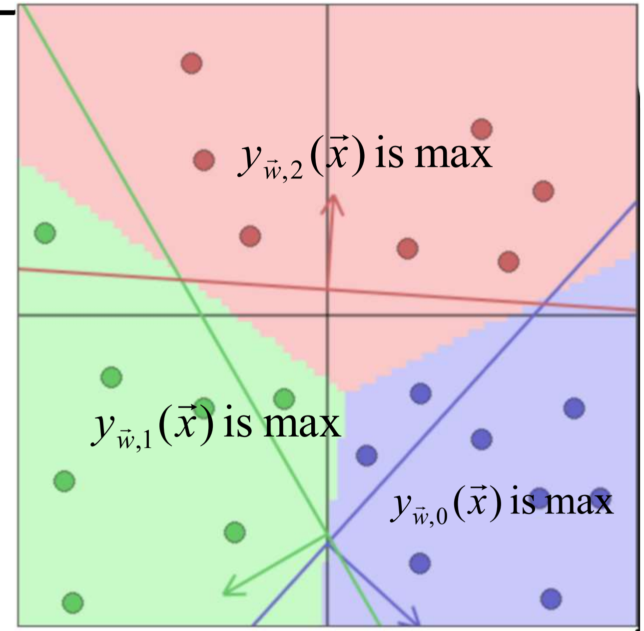
$$y_{\vec{w},1}(\vec{x}) = \vec{w}_1^T \vec{x} = w_{1,0} + w_{1,1}x_1 + w_{1,2}x_2$$

$$y_{\vec{w},2}(\vec{x}) = \vec{w}_2^T \vec{x} = w_{2,0} + w_{2,1}x_1 + w_{2,2}x_2$$

Multiclass Perceptron

(2D and 3 classes)





$$y_W(\vec{x}) = W^{\mathrm{T}}\vec{x}$$

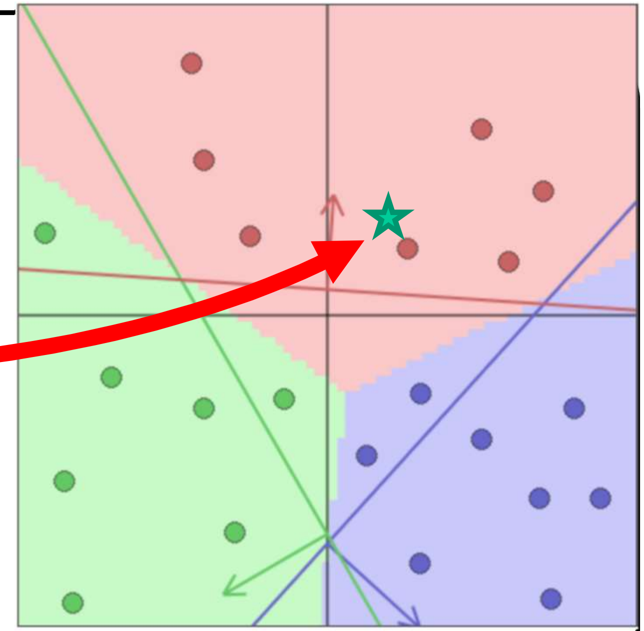
$$y_W(\vec{x}) = \begin{bmatrix} w_{0,0} & w_{0,1} & w_{0,2} \\ w_{1,0} & w_{1,1} & w_{1,2} \\ w_{2,0} & w_{2,1} & w_{2,2} \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}$$

Multiclass Perceptron

(2D and 3 classes)

Example

(1.1, -2.0)



$$y_W(\vec{x}) = \begin{bmatrix} -2 & -3.6 & 0.5 \\ -4 & 2.4 & 4.1 \\ -6 & 4 & -4.9 \end{bmatrix} \begin{bmatrix} 1 \\ 1.1 \\ -2 \end{bmatrix} = \begin{array}{c|l} \begin{bmatrix} -6.9 \\ -9.6 \\ 8.2 \end{bmatrix} & \begin{array}{l} \text{Class 0} \\ \text{Class 1} \\ \text{Class 2} \end{array} \end{array}$$

Multiclass Perceptron

Loss function

$$L\left(y_W(\vec{x}), D\right) = \sum_{\vec{x}_n \in V} \left(\vec{w}_j^T \vec{x}_n - \vec{w}_{t_n}^T \vec{x}_n\right)$$

Sum over all wrongly classified samples

Score of the true class

Score of the wrong class

$$\nabla L\left(y_{\vec{w}}(\vec{x}), D\right) = \sum_{\vec{x}_n \in V} \vec{x}_n$$

Multiclass Perceptron

Stochastic gradient descent (SGD)

init **W**
k=0, i=0
DO k=k+1
    FOR n = 1 to N
        $j = \arg\max \mathrm{W}^T \vec{x}_n$
        IF $j \neq t_i$ THEN /* wrongly classified sample */
            $\vec{w}_j = \vec{w}_j - \eta \vec{x}_n$
            $\vec{w}_{t_n} = \vec{w}_{t_n} + \eta \vec{x}_n$

UNTIL every data is well classified or k > K_MAX.

Perceptron

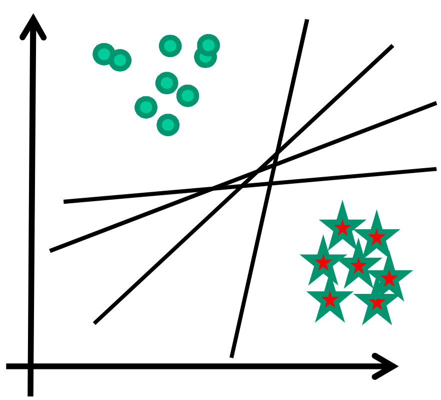
Advantages:

- Very simple
- Does **NOT** assume the data follows a **Gaussian distribution**.
- If data is **linearly separable**, convergence is **guaranteed**.

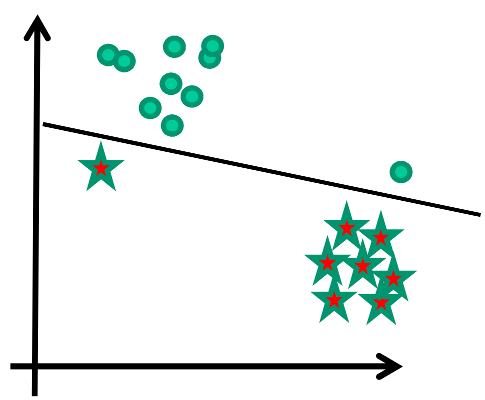
Limitations:

- Zero gradient for many solutions => several "perfect solutions"
- Data must be **linearly separable**

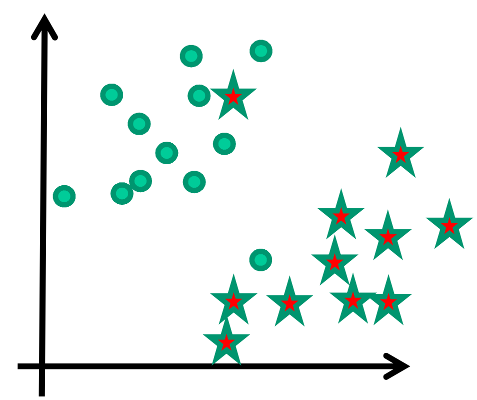
Many "optimal" solutions



Suboptimal solution



Will never converge



Two famous ways of improving the Perceptron

1. New **activation function** + new **Loss**

Logistic regression

1. New network architecture

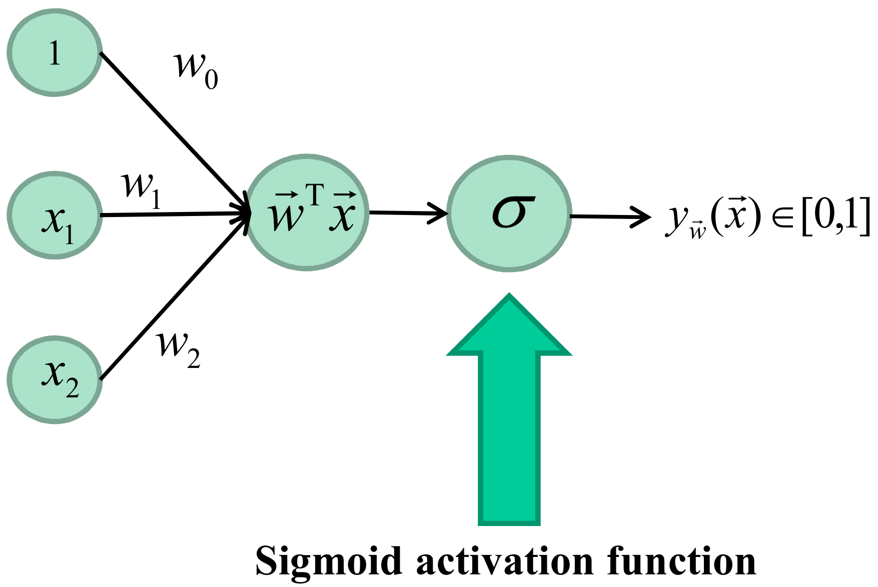
Multilayer Perceptron / CNN

Logistic regression

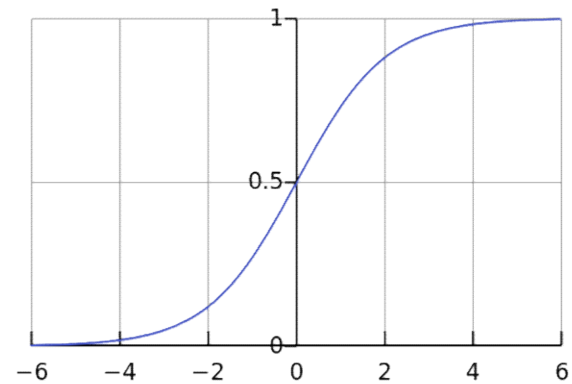
Logistic regression

(2D, 2 classes)

New activation function: **sigmoid**



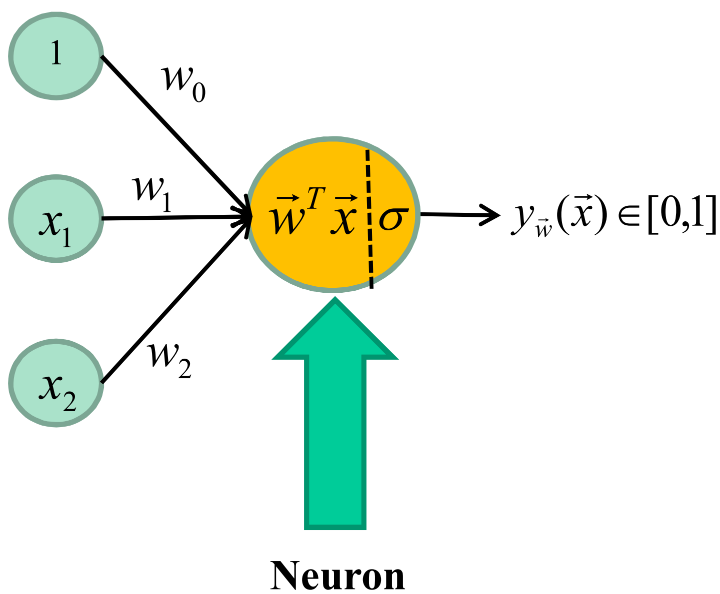
$$\sigma(t) = \frac{1}{1+e^{-t}}$$



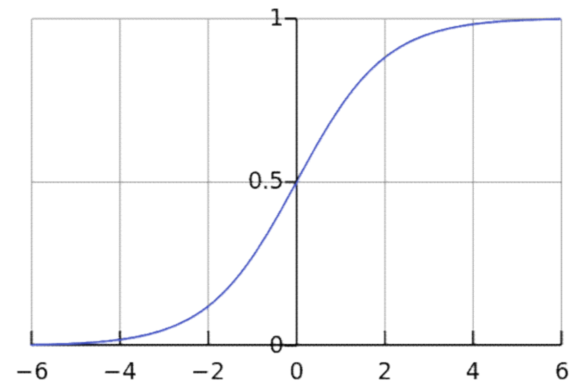
Logistic regression

(2D, 2 classes)

New activation function: **sigmoid**



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

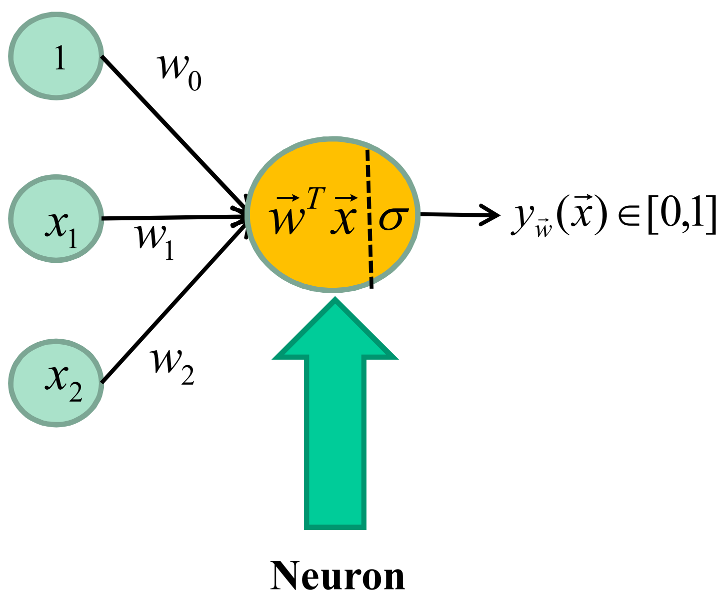


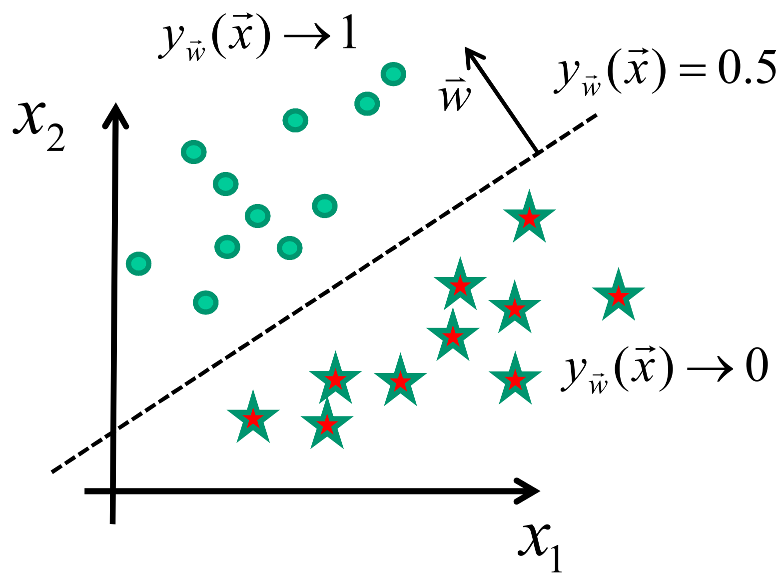
$$\boxed{y_{\vec{w}}(\vec{x}) = \sigma\left(\vec{w}^T\vec{x}\right) = \frac{1}{1+e^{-\vec{w}^T\vec{x}}}}$$

Logistic regression

(2D, 2 classes)

New activation function: **sigmoid**





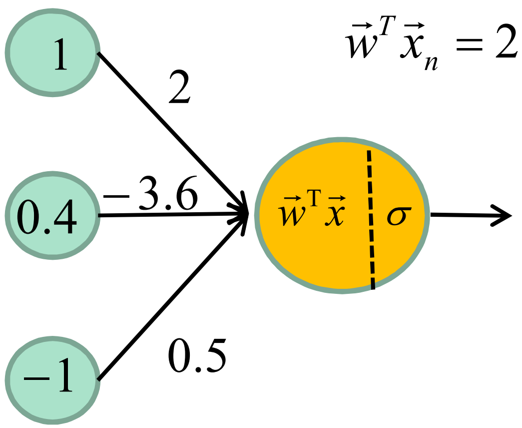
$$y_{\vec{w}}(\vec{x}) = \sigma\left(\vec{w}^T\vec{x}\right)$$

Logistic regression

(2D, 2 classes)

Example

$$\vec{x}_n = (0.4, -1.0), \vec{w} = [2.0, -3.6, 0.5]$$



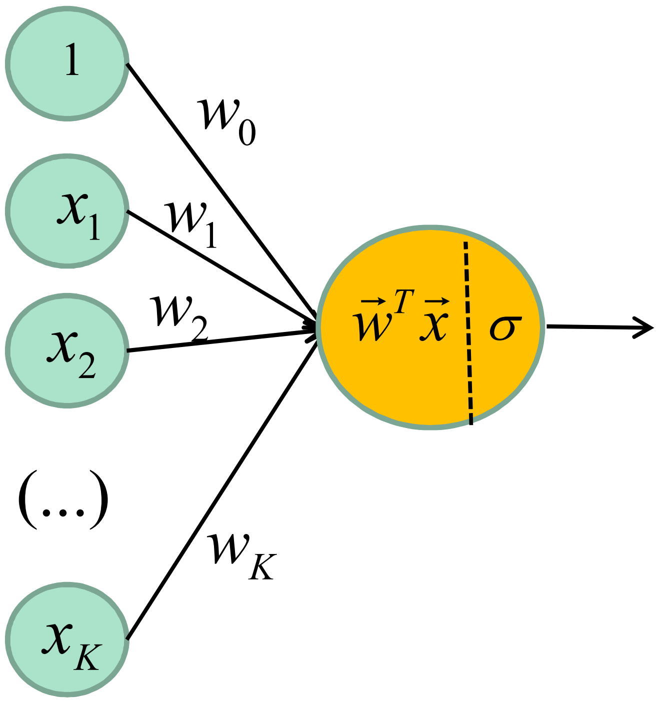
$$\vec{w}^T\vec{x}_n = 2 - 3.6 * 0.4 - 0.5 = -1.94$$

$$y_{\vec{w}}(\vec{x}) = \sigma(-1.94) = \frac{1}{1+e^{1.94}} = 0.125$$

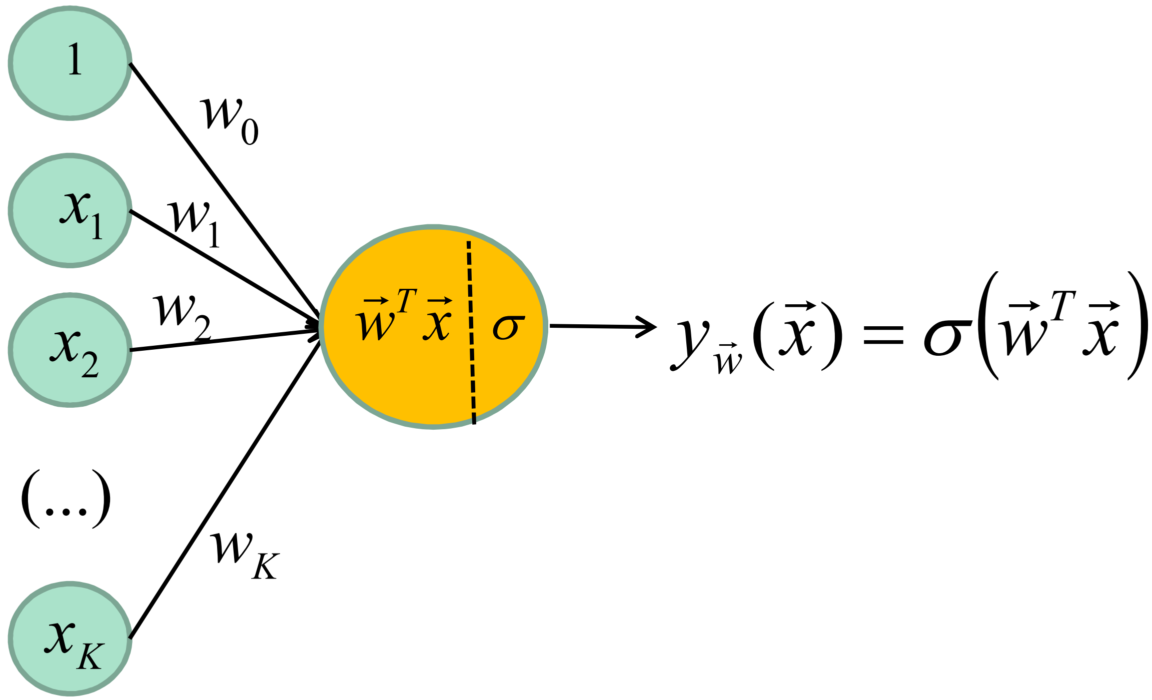
Since 0.125 is lower than 0.5, $\vec{x}_n$ is **behind** the plan.

Logistic regression
(K-D, 2 classes)

Like the Perceptron the logistic regression accomodates for K-D vectors

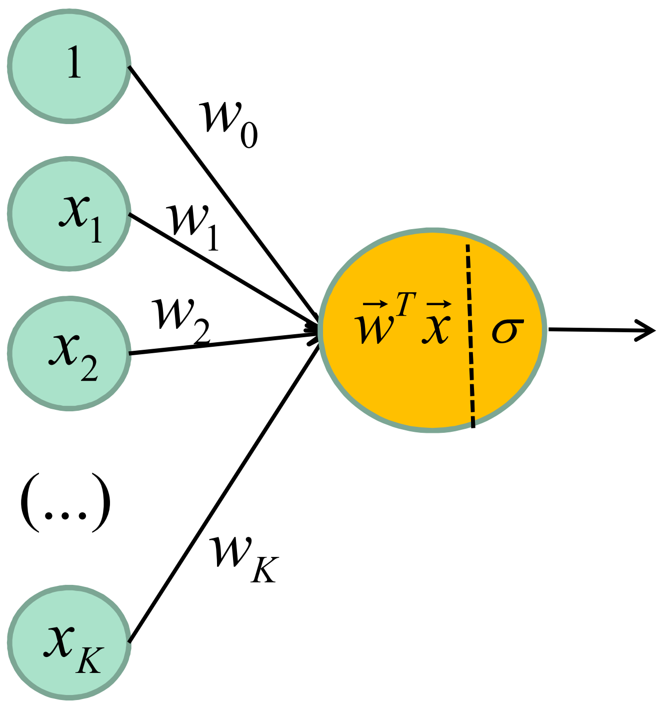


$$y_{\vec{w}}(\vec{x}) = \sigma\left(\vec{w}^T\vec{x}\right)$$



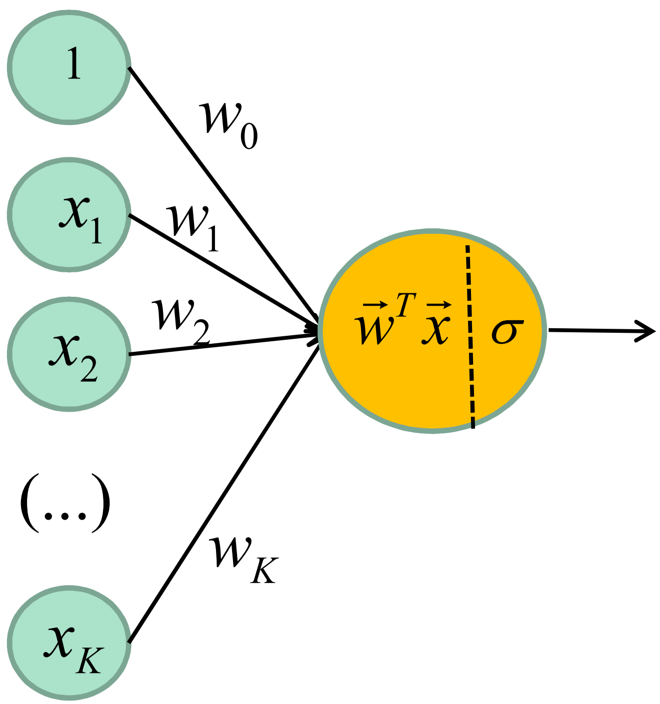
What is the loss function?

With a sigmoid, we can **simulate a conditional probability** of c_1 GIVEN $\vec{x}$



$$y_{\vec{w}}(\vec{x}) = \sigma\left(\vec{w}^T\vec{x}\right) \approx P\left(c_1 \mid \vec{x}\right)$$

With a sigmoid, we can **simulate a conditional probability** of c_1 GIVEN $\vec{x}$



$$y_{\vec{w}}(\vec{x}) = \sigma\left(\vec{w}^T\vec{x}\right) \approx P\left(c_1 \mid \vec{x}\right)$$

$$\Rightarrow P\left(c_0 \mid \vec{w}, \vec{x}\right) = 1 - y_{\vec{w}}(\vec{x})$$

Cost function is **–ln of the likelihood**

$$L\left(y_{\vec{w}}(\vec{x}), D\right) = -\sum_{n=1}^{N} t_n \ln\left(y_{\vec{w}}(\vec{x}_n)\right) + (1 - t_n)\ln\left(1 - y_{\vec{w}}(\vec{x}_n)\right)$$

2 Class Cross entropy

We can also show that

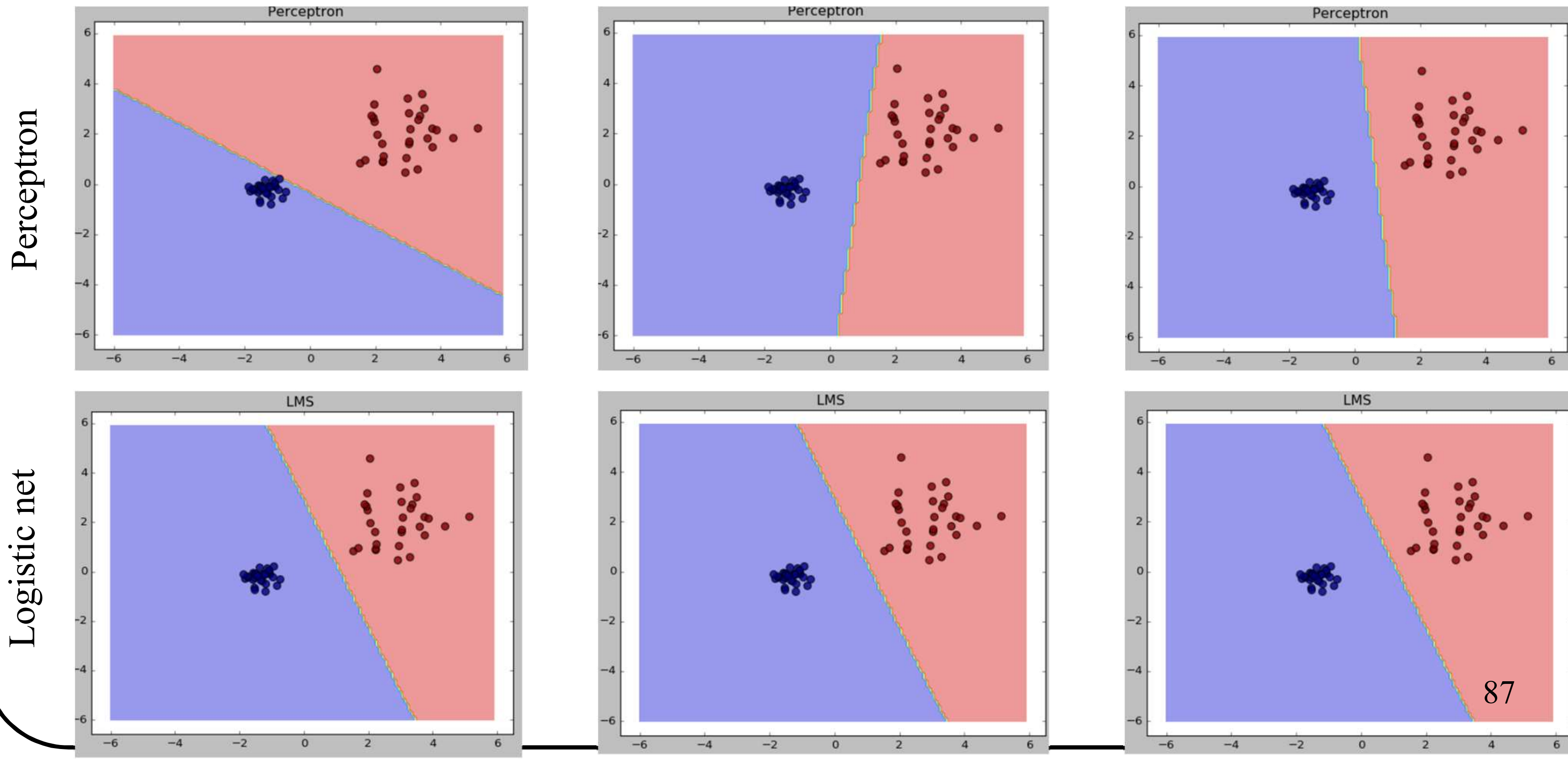
$$\frac{dL\left(y_{\vec{w}}(\vec{x}), D\right)}{d\vec{w}} = \sum_{n=1}^{N} \left(y_{\vec{w}}(\vec{x}_n) - t_n\right)\vec{x}_n$$

As opposed to the Perceptron
the gradient does not depend
on the wrongly classified samples

Logistic Network

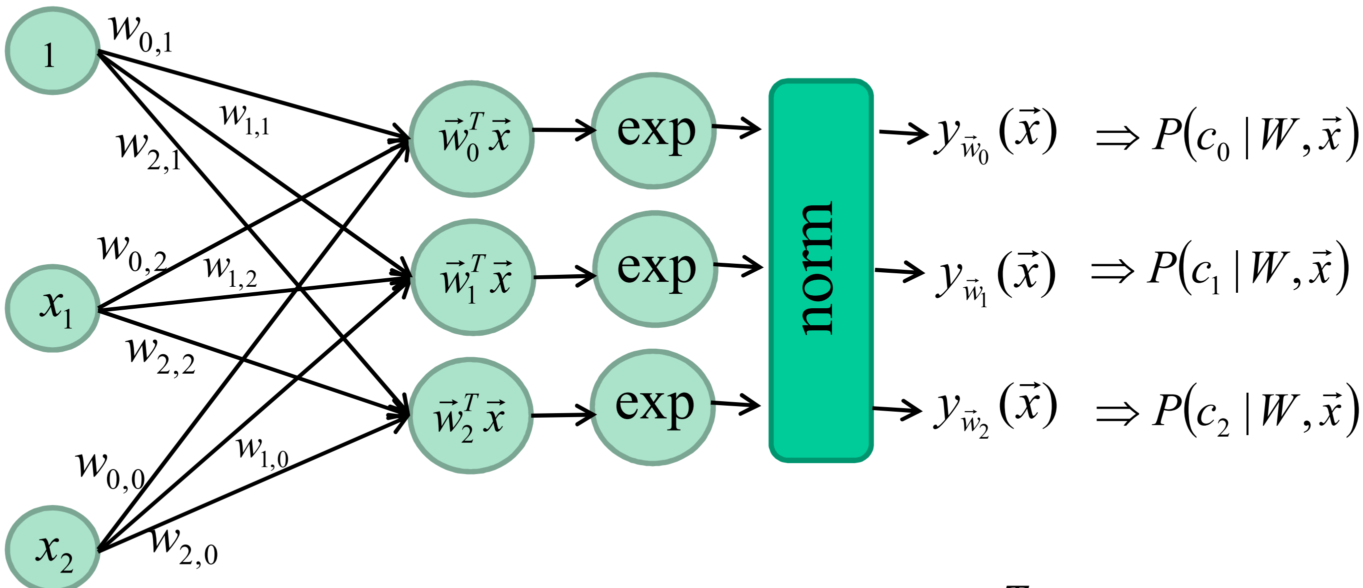
Advantages:

- **More stable than the Perceptron**
- More effective when the data is **non separable**



And for K>2 classes?

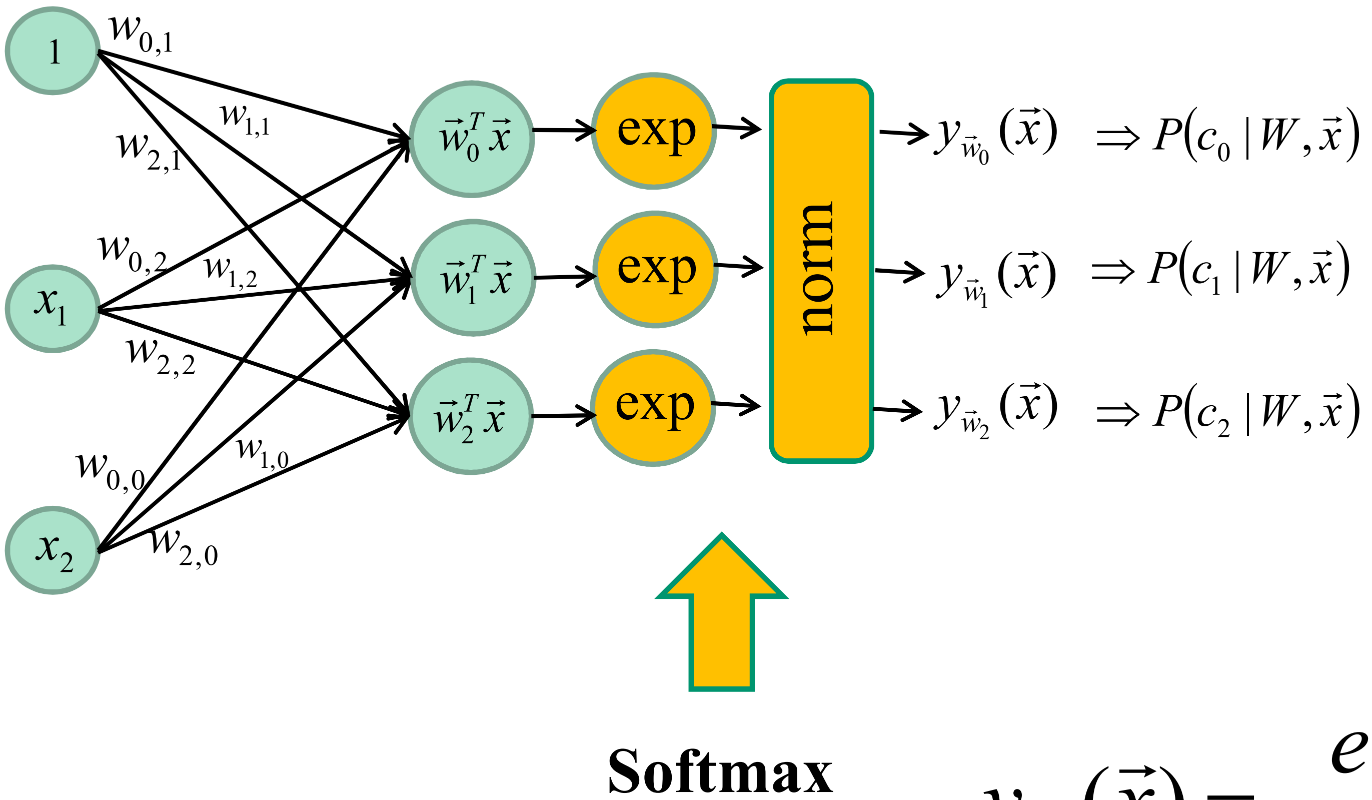
New activation function : **Softmax**



$$y_{\vec{w}_i}(\vec{x}) = \frac{e^{\vec{w}_i^T\vec{x}}}{\sum_c e^{\vec{w}_c^T\vec{x}}}$$

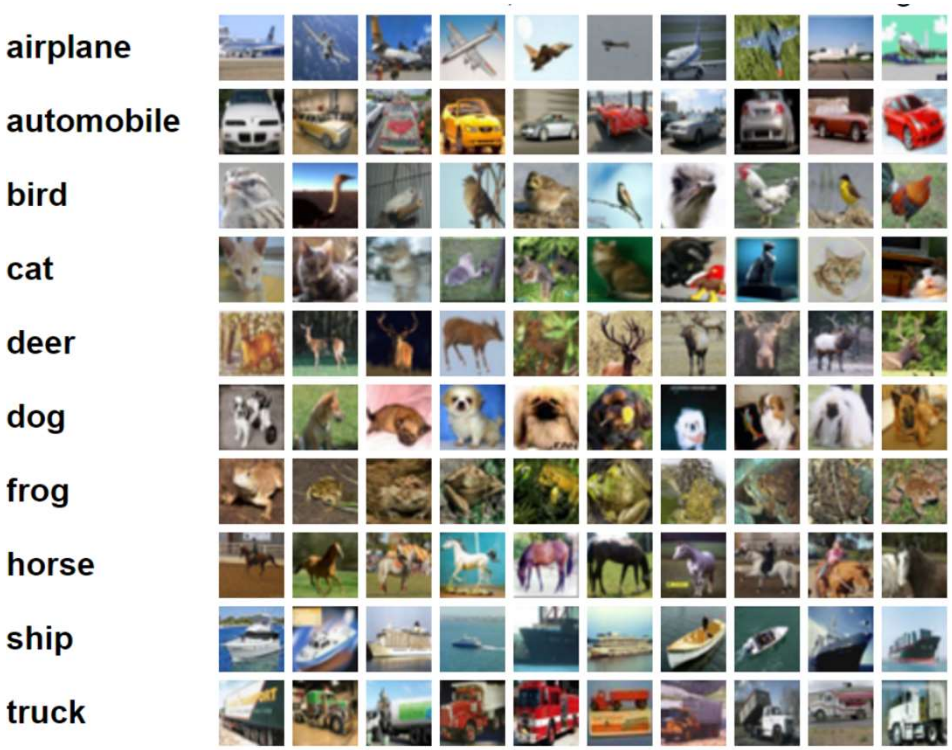
And for K>2 classes?

New activation function : **Softmax**



$$y_{\vec{w}_i}(\vec{x}) = \frac{e^{\vec{w}_i^T \vec{x}}}{\sum_c e^{\vec{w}_c^T \vec{x}}}$$

And for K>2 classes?



Cifar10

$$'\text{airplane}' \Rightarrow t = [1000000000]$$
$$'\text{automobile}' \Rightarrow t = [0100000000]$$
$$'\text{bird}' \Rightarrow t = [0010000000]$$
$$'\text{cat}' \Rightarrow t = [0001000000]$$
$$'\text{deer}' \Rightarrow t = [0000100000]$$
$$'\text{dog}' \Rightarrow t = [0000010000]$$
$$'\text{frog}' \Rightarrow t = [0000001000]$$
$$'\text{horse}' \Rightarrow t = [0000000100]$$
$$'\text{ship}' \Rightarrow t = [0000000010]$$
$$'\text{truck}' \Rightarrow t = [0000000001]$$

Class labels : **one-hot vectors**

K>2 classes

Cross entropy Loss

$$\boxed{L\left(y_W\left(\vec{x}\right), D\right) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{kn} \ln y_{W_k}\left(\vec{x}_n\right)}$$

K-Class *cross entropy* loss

$$\boxed{L\left(y_W\left(\vec{x}\right), D\right) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{kn} \ln y_{W_k}\left(\vec{x}_n\right)}$$

$$\boxed{\nabla L = \sum_{n=1}^{N} \vec{x}_n \left(y_W\left(\vec{x}_n\right) - t_{kn}\right)}$$

Regularization

Different weights may give the same score

$$\vec{x} = (1.0, 1.0, 1.0)$$

$$\vec{w}_1^T = [1,0,0]$$

$$\vec{w}_2^T = [1/3, 1/3, 1/3]$$

$$\vec{w}_1^T \vec{x} = \vec{w}_2^T \vec{x} = 1$$

Which weights are the best?

Solution: Maximum a posteriori

Maximum *a posteriori*

Regularization

Constant

$$\arg\min_{W} = L\left(y_{\vec{w}}\left(\vec{x}\right), D\right) + \lambda R\left(W\right)$$

Loss function

Regularization

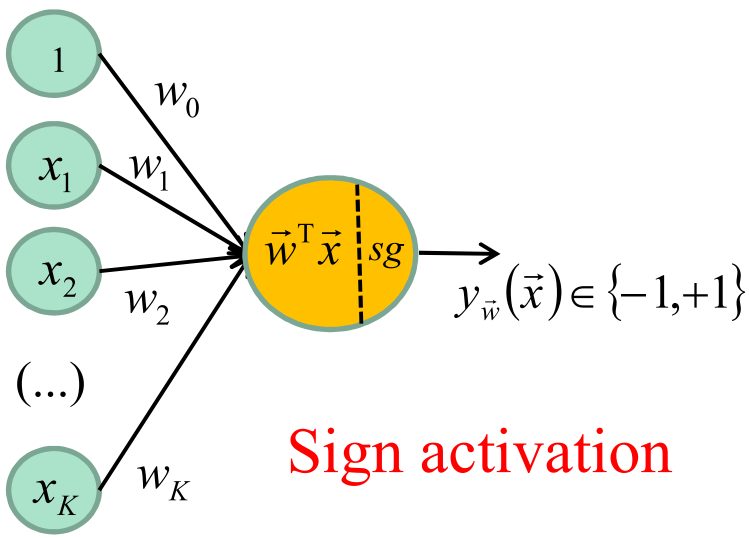
In general L1 or L2 $R\left(\theta\right) = \left\|\mathrm{W}\right\|_1 \text{ ou } \left\|\mathrm{W}\right\|_2$

Wow! Loooots of information!

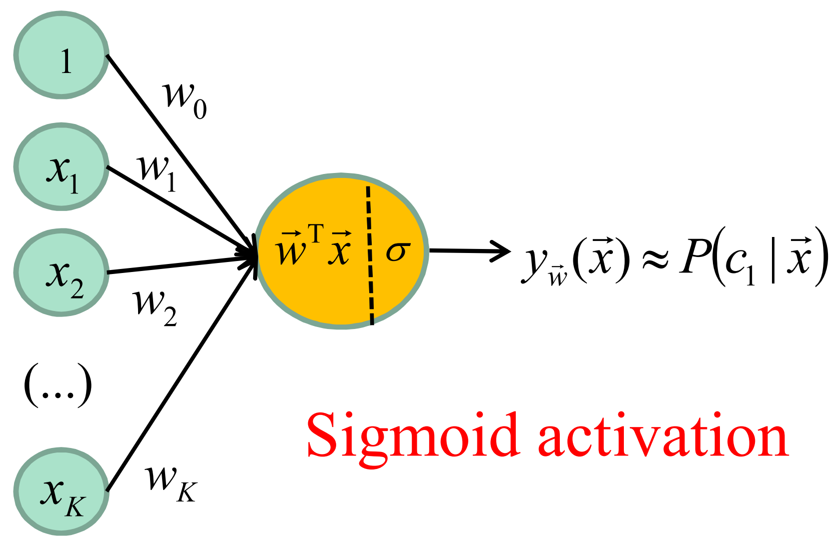
Lets recap…

Neural networks

2 classes

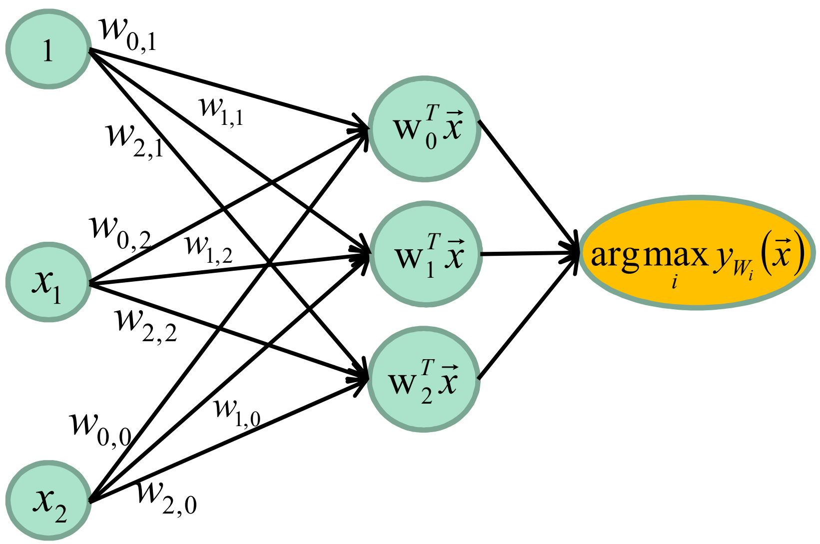


Perceptron

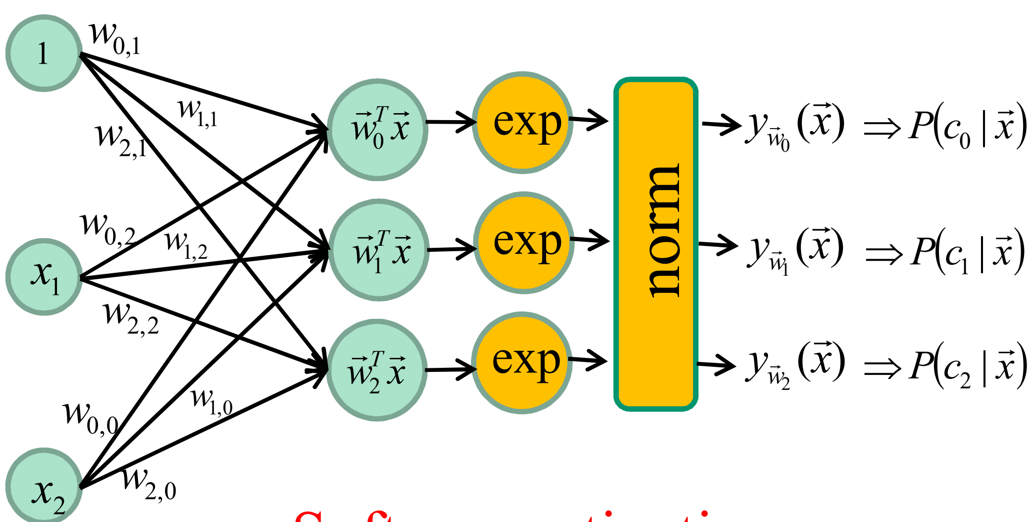


Logistic regression

K-Class Neural networks



Perceptron



Logistic regression

Softmax activation

Loss functions

2 classes

$$L\left(y_{\vec{w}}(\vec{x}), D\right) = \sum_{\vec{x}_n \in V} -t_n \vec{w}^{\mathrm{T}} \vec{x}_n$$ where V is the set of wrongly classified samples

$$L\left(y_{\vec{w}}(\vec{x}), D\right) = \sum_{n=1}^{N} \max\left(0, -t_n \vec{w}^{\mathrm{T}} \vec{x}_n\right)$$

$$L\left(y_{\vec{w}}(\vec{x}), D\right) = \sum_{n=1}^{N} \max\left(0, 1 - t_n \vec{w}^{\mathrm{T}} \vec{x}_n\right)$$ "Hinge Loss" or "SVM" Loss

$$L\left(y_{\vec{w}}(\vec{x}), D\right) = -\sum_{n=1}^{N} t_n \ln\left(y_{\vec{w}}(\vec{x}_n)\right) + \left(1 - t_n\right) \ln\left(1 - y_{\vec{w}}(\vec{x}_n)\right)$$ Cross entropy loss

Loss functions

K classes

$$L\left(y_{\vec{w}}(\vec{x}), D\right) = \sum_{\vec{x}_n \in V} \left(\vec{w}_j^T \vec{x}_n - \vec{w}_{t_n}^T \vec{x}_n\right)$$ where V *i*s the set of wrongly classified samples

$$L\left(y_{\vec{w}}(\vec{x}), D\right) = \sum_{n=1}^{N} \sum_{j} \max\left(0, \vec{w}_j^T \vec{x}_n - \vec{w}_{t_n}^T \vec{x}_n\right)$$

$$L\left(y_{\vec{w}}(\vec{x}), D\right) = \sum_{n=1}^{N} \sum_{j} \max\left(0, 1 + \vec{w}_j^T \vec{x}_n - \vec{w}_{t_n}^T \vec{x}_n\right)$$ "Hinge Loss" or "SVM" Loss

$$L\left(y_{\vec{w}}(\vec{x}), D\right) = -\sum_{n=1}^{N} \sum_{k=1}^{K} t_{kn} \ln y_{W_k}(\vec{x}_n)$$ *Cross entropy* loss with a Softmax

Maximum *a posteriori*

Constante

$$L\left(y_{\vec{w}}(\vec{x}), D\right) = \sum_{n=1}^{N} l\left(y_W(\vec{x}_n), t_n\right) + \lambda R(\mathrm{W})$$

Loss function

Regularization

$$R(W) = \|W\|_1 \ or \ \|W\|_2$$

Optimisation

$$\vec{\mathrm{w}}^{[k+1]} = \vec{\mathrm{w}}^{[k]} - \eta^{[k]} \nabla L$$

Gradient of the loss function

learning rate

Stochastic gradient descent (SGD)

Init $\vec{\mathrm{w}}$
k=0
DO k=k+1
    FOR n = 1 to N
        $\vec{w} = \vec{w} - \eta^{[k]} \nabla L(\vec{x}_n)$

UNTIL every data is well classified or k== MAX_ITER

Now, lets go

DEEPER

Non-linearly separable training data

Three classical solutions

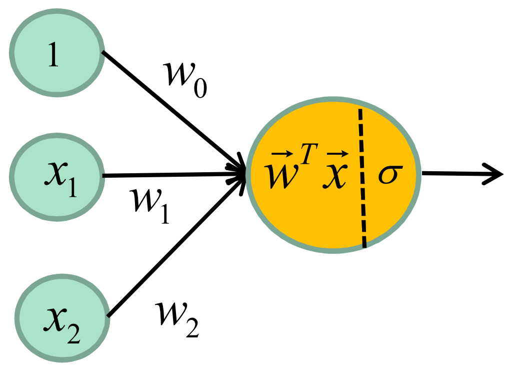
1. Acquire more data
2. Use a non-linear classifier
3. Transform the data

Non-linearly separable training data

Three classical solutions

1. Acquire more data
2. Use a non-linear classifier
3. **Transform the data**

2D, 2Classes, Linear logistic regression

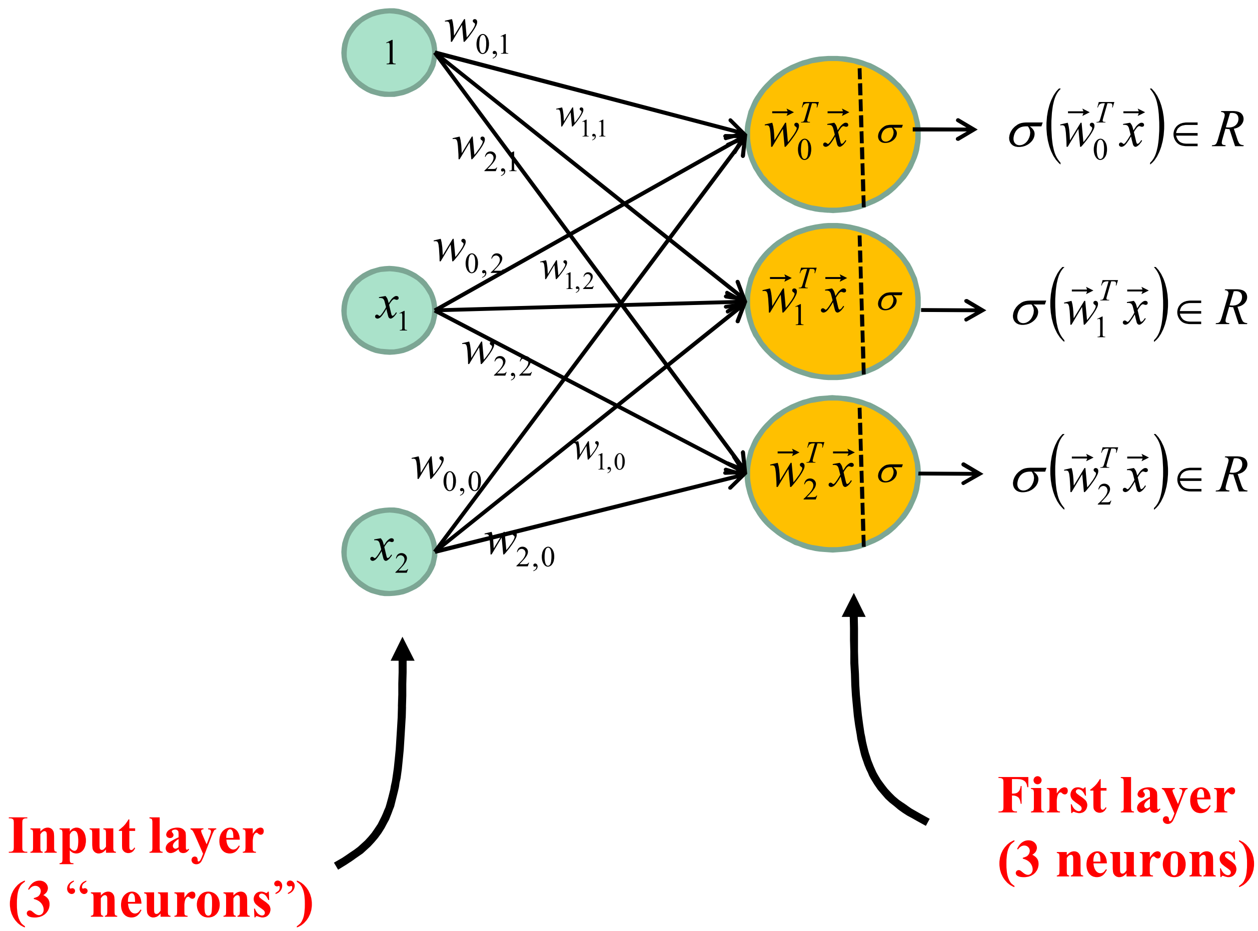


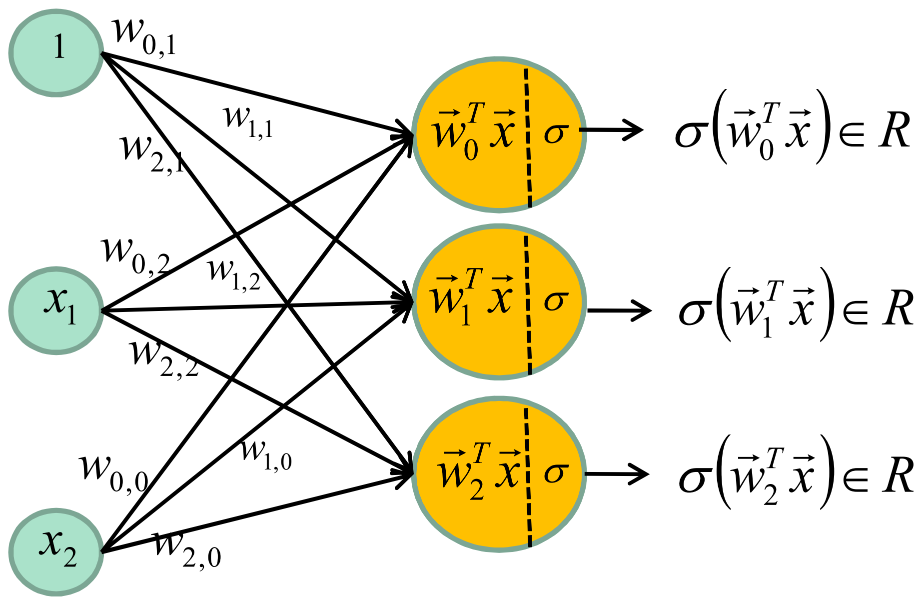
$$y_{\vec{w}}(\vec{x}) = \sigma\left(\vec{w}^T\vec{x}\right) \in R$$

Input layer
(3 "neurons")

Output layer
(1 neuron with sigmoid)

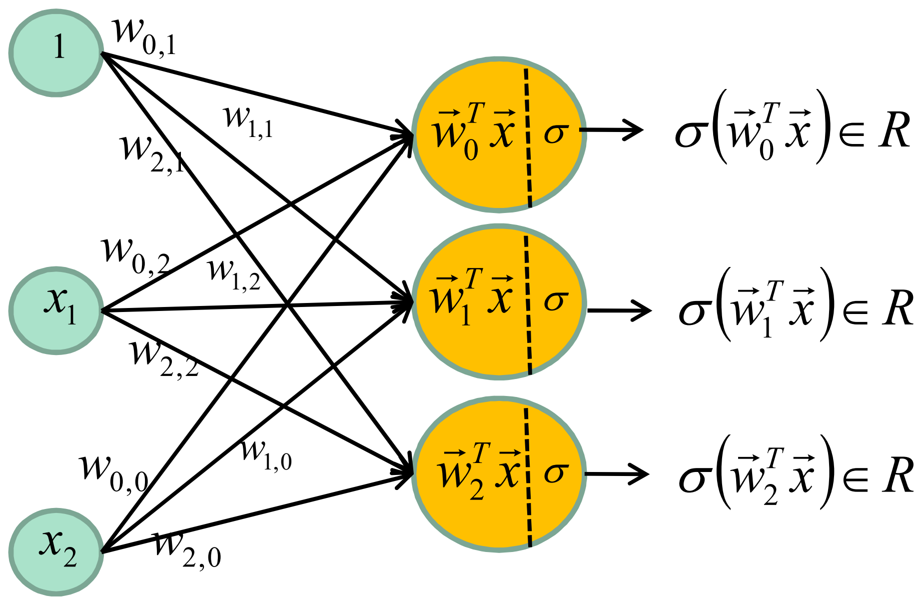
Let's add 3 neurons





NOTE: The output of the first layer is a vector of **3 real** values

$$\sigma\left(\begin{bmatrix} w_{0,0} & w_{0,1} & w_{0,2} \\ w_{1,0} & w_{1,1} & w_{1,2} \\ w_{2,0} & w_{2,1} & w_{2,2} \end{bmatrix}\begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}\right)\in R^3$$

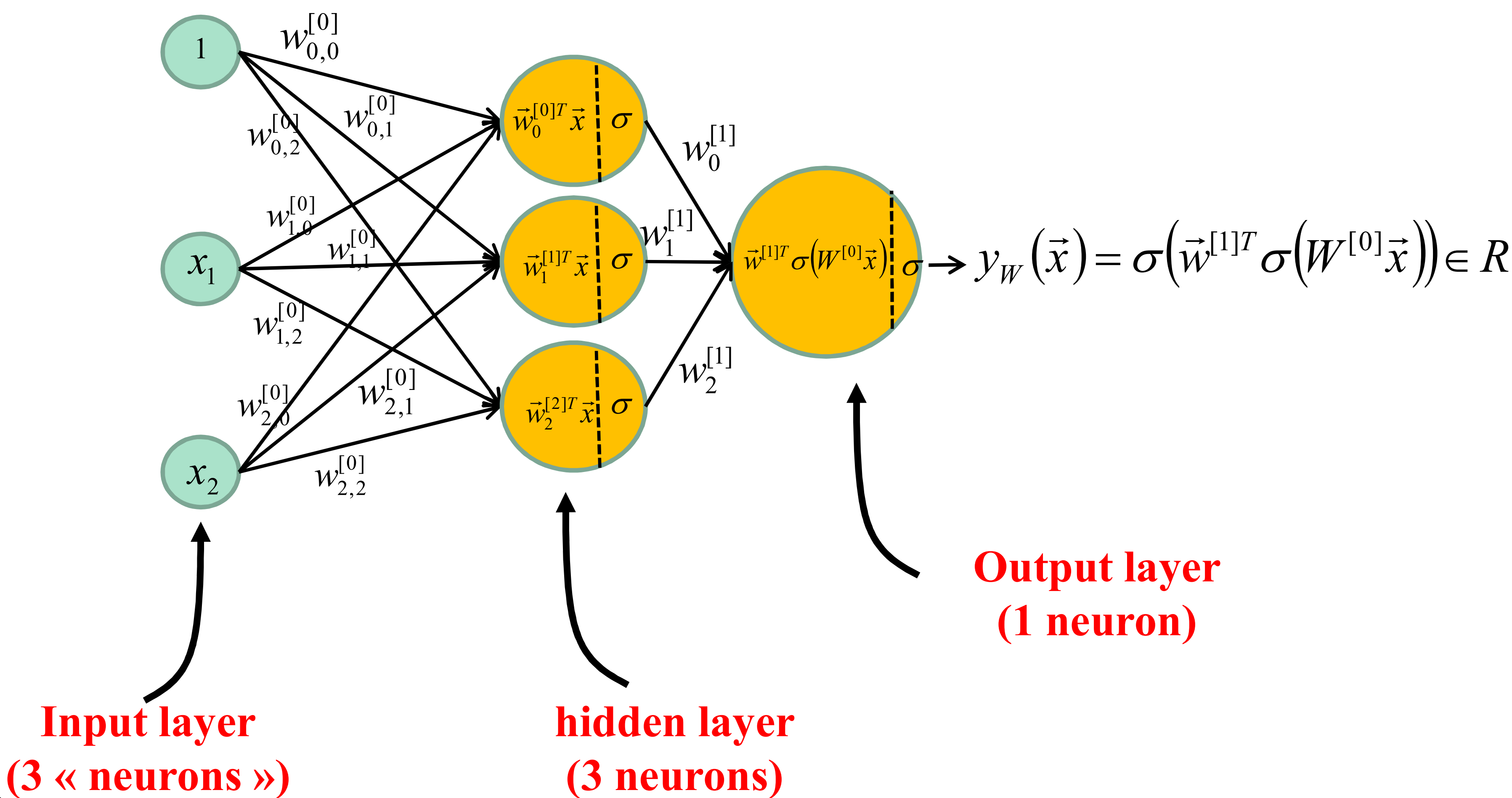


NOTE: The output of the first layer is a vector of **3 real** values

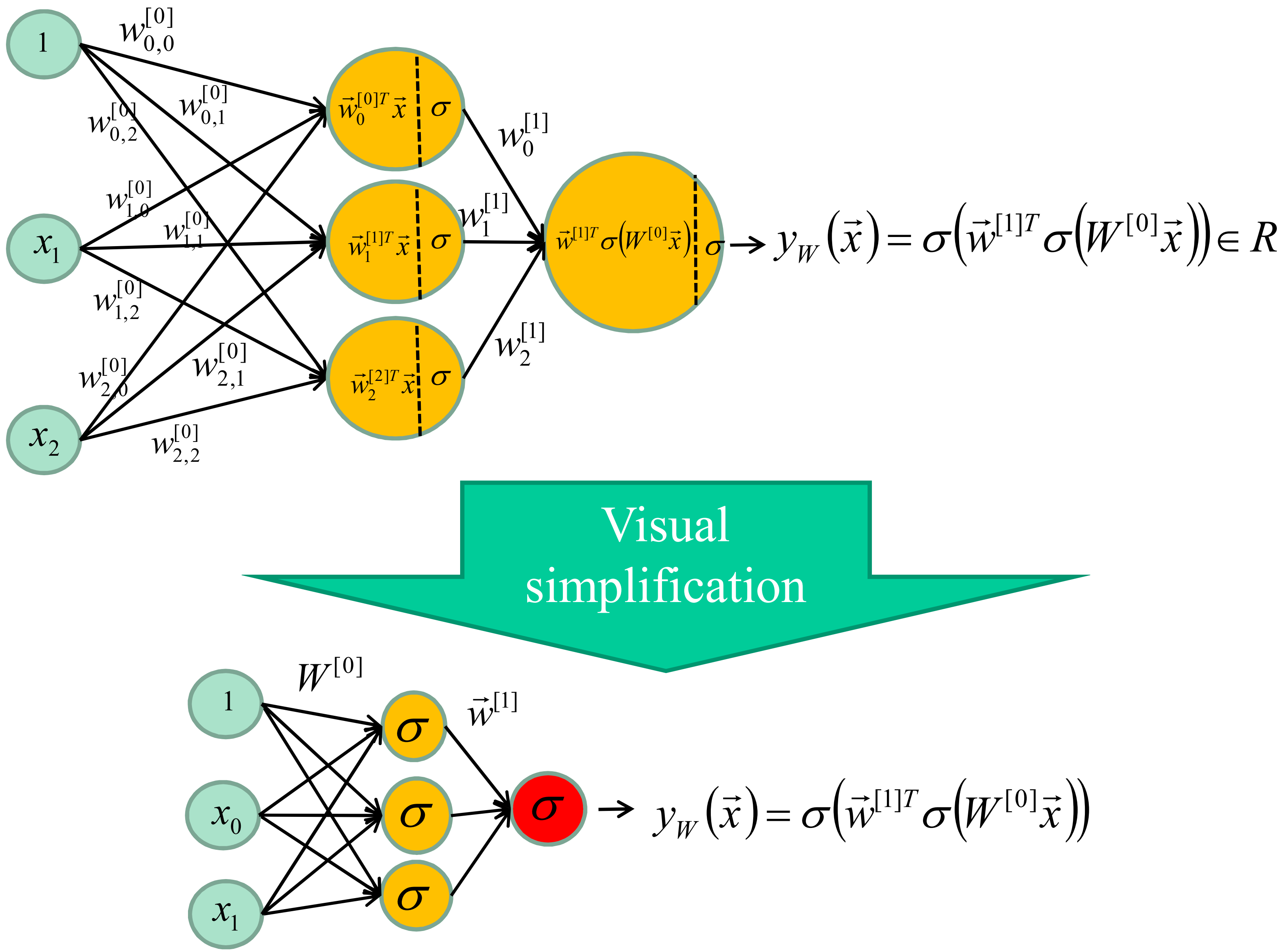
$$\sigma\left(W^{[0]}\vec{x}\right)$$

2-D, 2-Class, 1 hidden layer

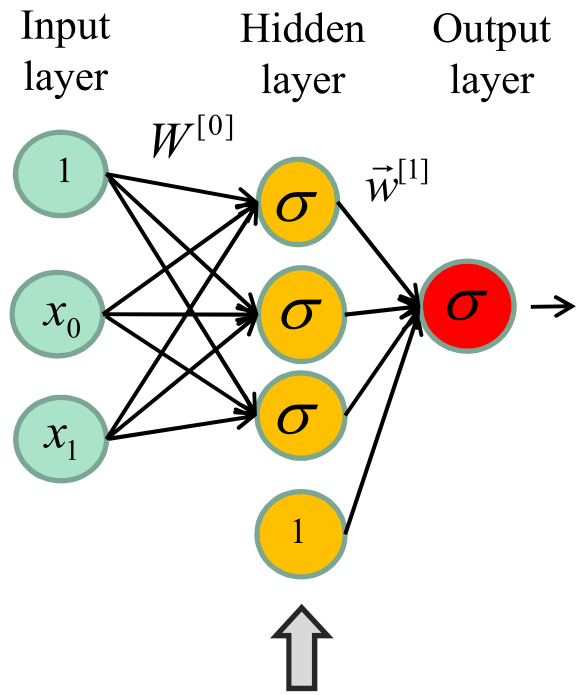
If we want a **2-class Classification** via a **logistic regression** (a **cross entropy loss**) we must add an **output neuron.**



2-D, 2-Class, 1 hidden layer



2-D, 2-Class, 1 hidden layer



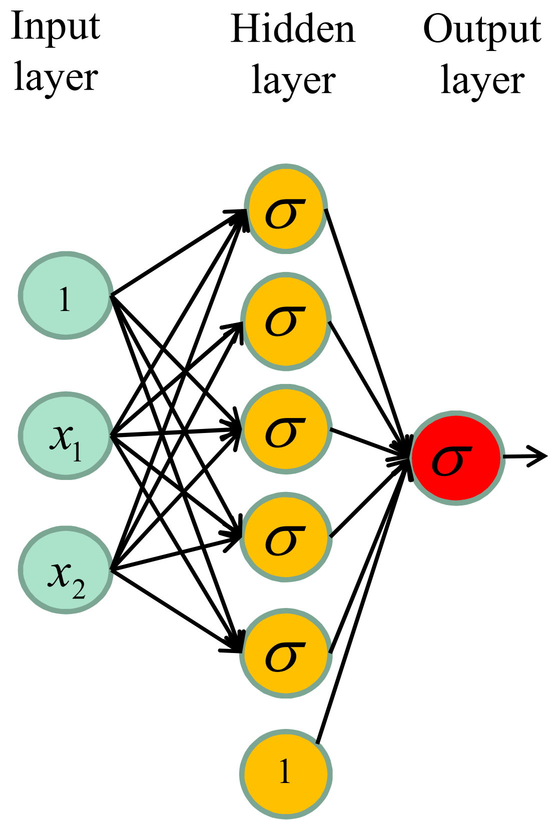
$$y_W(\vec{x}) = \sigma\left(\vec{w}^{[1]T}\sigma\left(W^{[0]}\vec{x}\right)\right)$$

bias neuron.

This network contains a total of **13 parameters**

$\underbrace{3\text{x}3}_{\text{Input layer / Hidden layer}}$ $\underbrace{1\text{x}4}_{\text{Hidden layer / Output layer}}$

2-D, 2-Class, 1 hidden layer



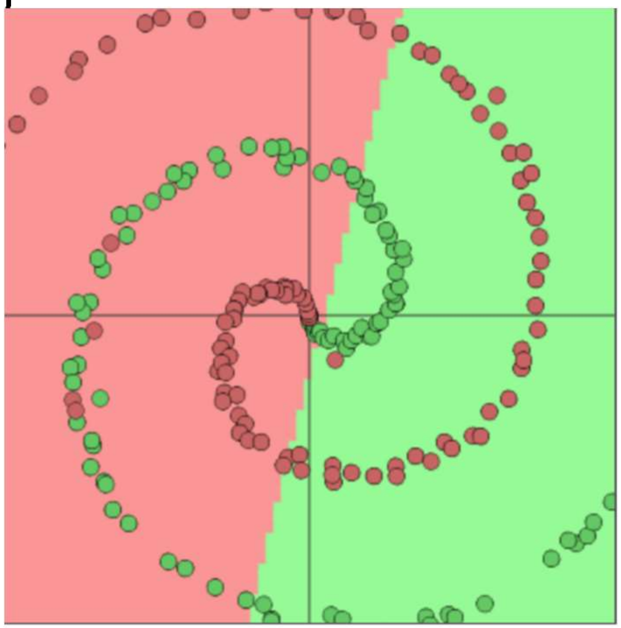
$$y_W\left(\vec{x}\right) = \sigma\left(\vec{w}^{[1]T}\sigma\left(W^{[0]}\vec{x}\right)\right)$$

Increasing the number of neurons = increasing the **capacity of the model**

This network has 5x3+1x6=**21 parameters**

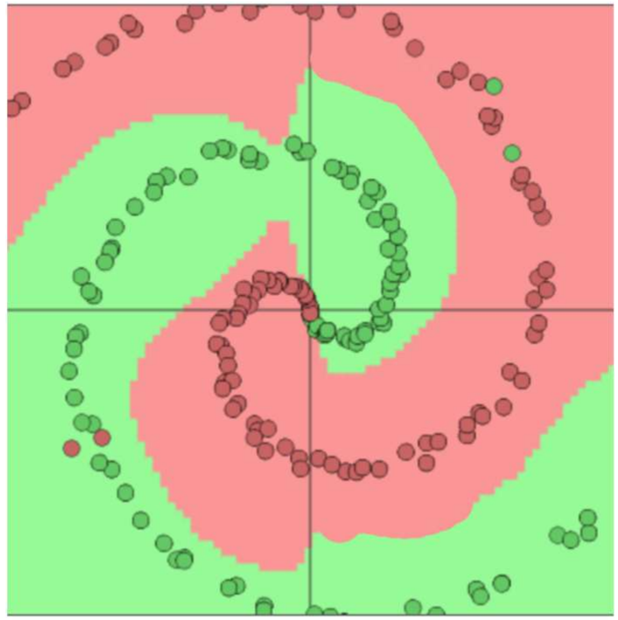
Nb neurons VS Capacity

No hidden neuron



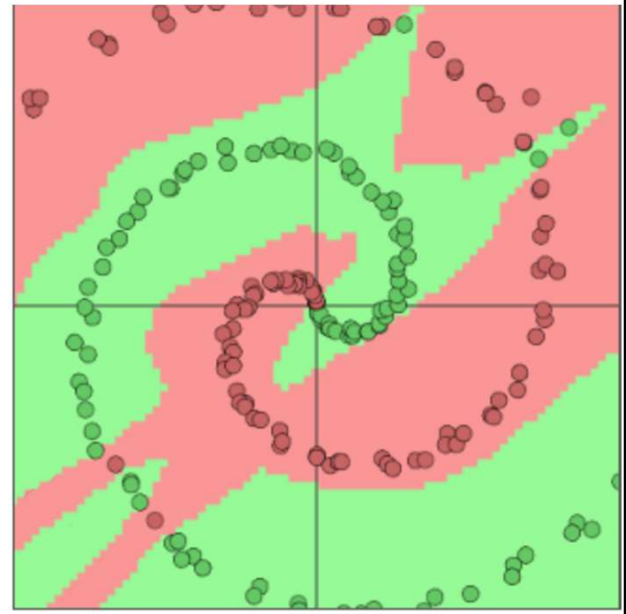
Linear classification
Underfitting
(low capacity)

12 hidden neurons



Non linear classification
Good result
(good capacity)

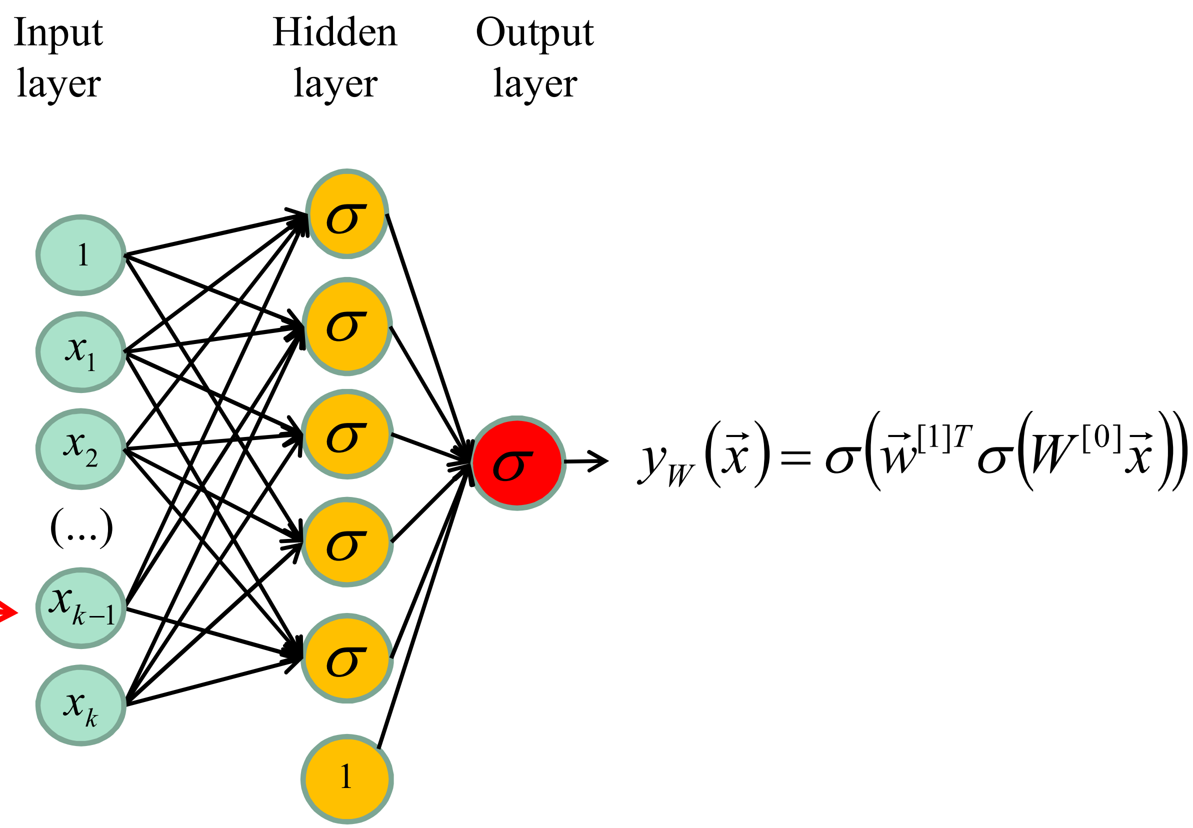
60 hidden neurons



Non linear classification
Over fitting
(too large capacity)

http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

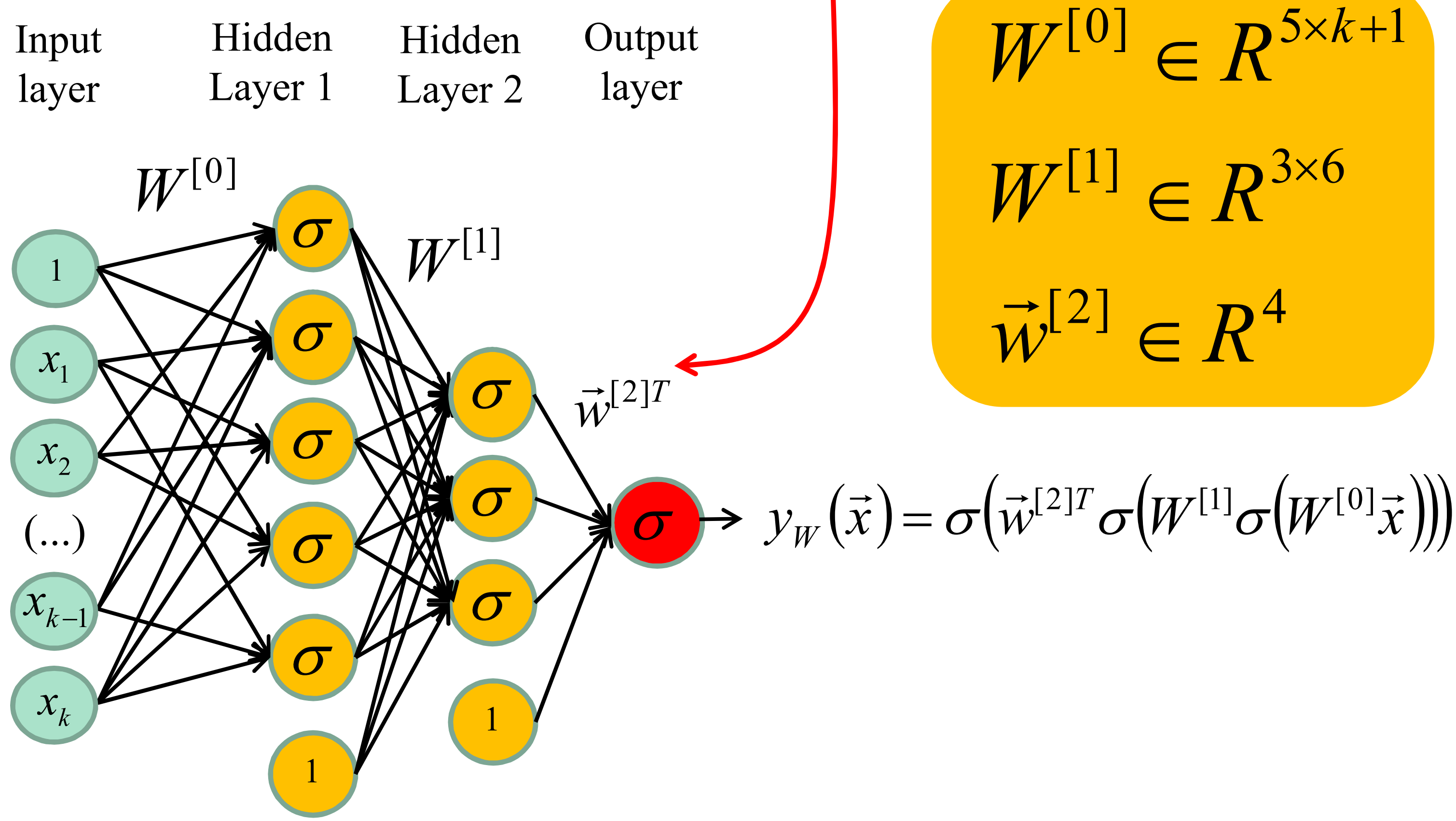
kD, 2Classes, 1 hidden layer



Increasing the dimensionality of the data = **more columns in** $W^{[0]}$

This network has 5x(k+1)+1x6 **parameters**

kD, 2Classes, **2 hidden layers**



$$W^{[0]} \in R^{5\times k+1}$$

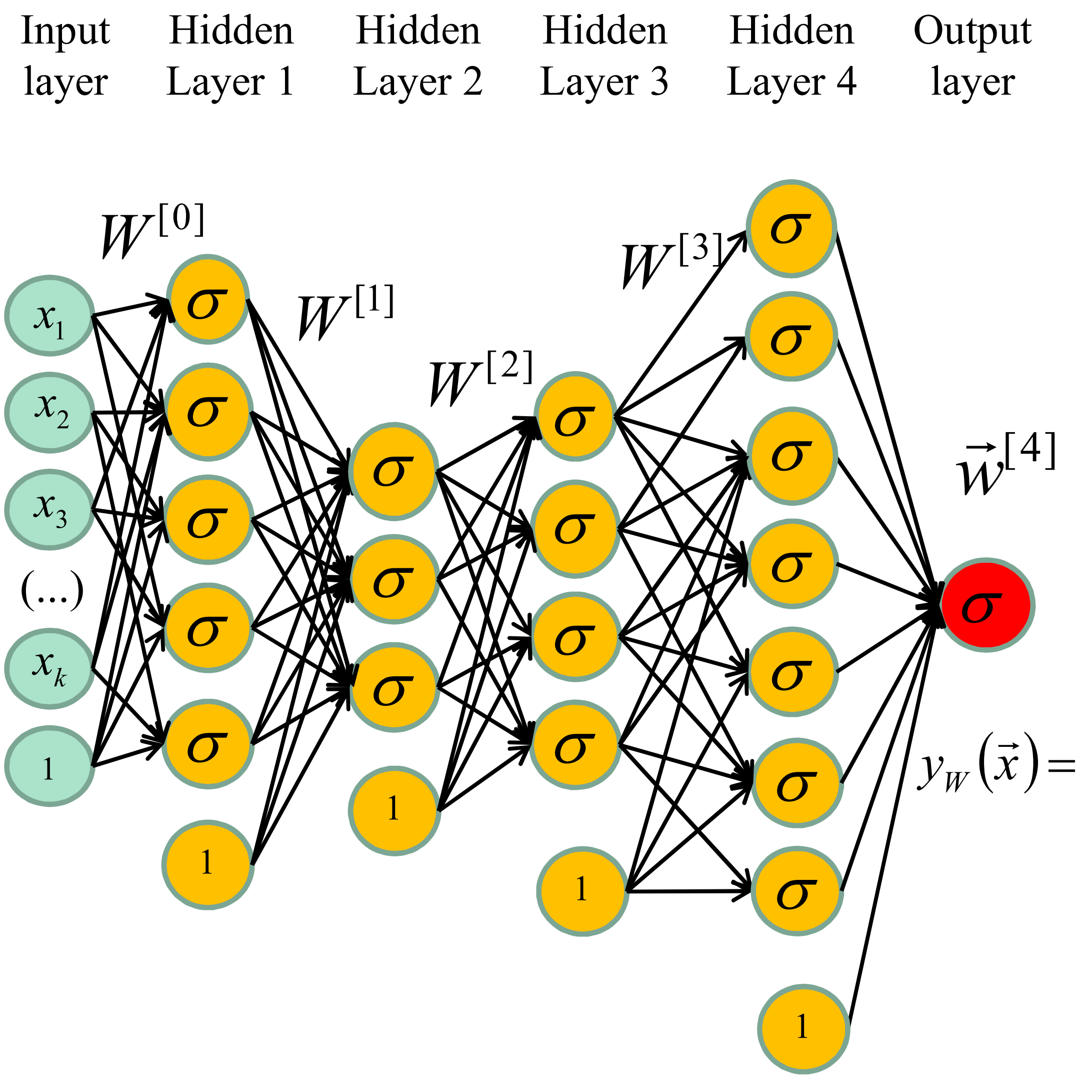
$$W^{[1]} \in R^{3\times 6}$$

$$\vec{w}^{[2]} \in R^{4}$$

Adding an hidden layer = Adding a matrix multiplication

This network has 5x(k+1)+6x3 + 1x4 **parameters**

kD, 2 Classes, 4 hidden layer network



$$W^{[0]} \in R^{5\times k+1}$$

$$W^{[1]} \in R^{3\times 6}$$

$$W^{[2]} \in R^{4\times 4}$$

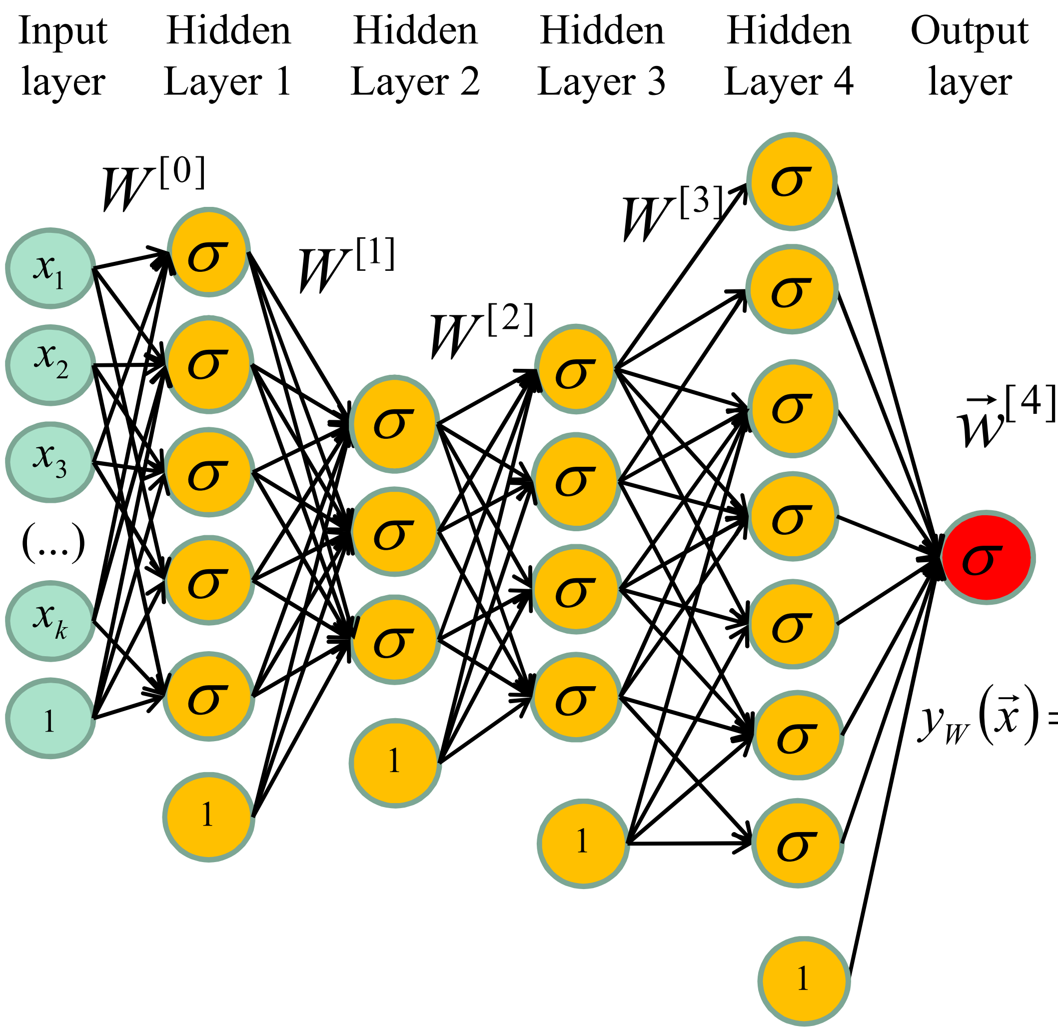
$$W^{[3]} \in R^{7\times 5}$$

$$\vec{w}^{[4]} \in R^{8}$$

$$y_W(\vec{x}) = \sigma\left(\vec{w}^{[4]T}\sigma\left(W^{[3]}\sigma\left(W^{[2]}\sigma\left(W^{[1]}\sigma\left(W^{[0]}\vec{x}\right)\right)\right)\right)\right)$$

This network has 5x(k+1)+6x3 + 4x4+7x5+1x8 **parameters**

kD, 2 Classes, 4 hidden layer network



$$W^{[0]} \in R^{5\times k+1}$$

$$W^{[1]} \in R^{3\times 6}$$

$$W^{[2]} \in R^{4\times 4}$$

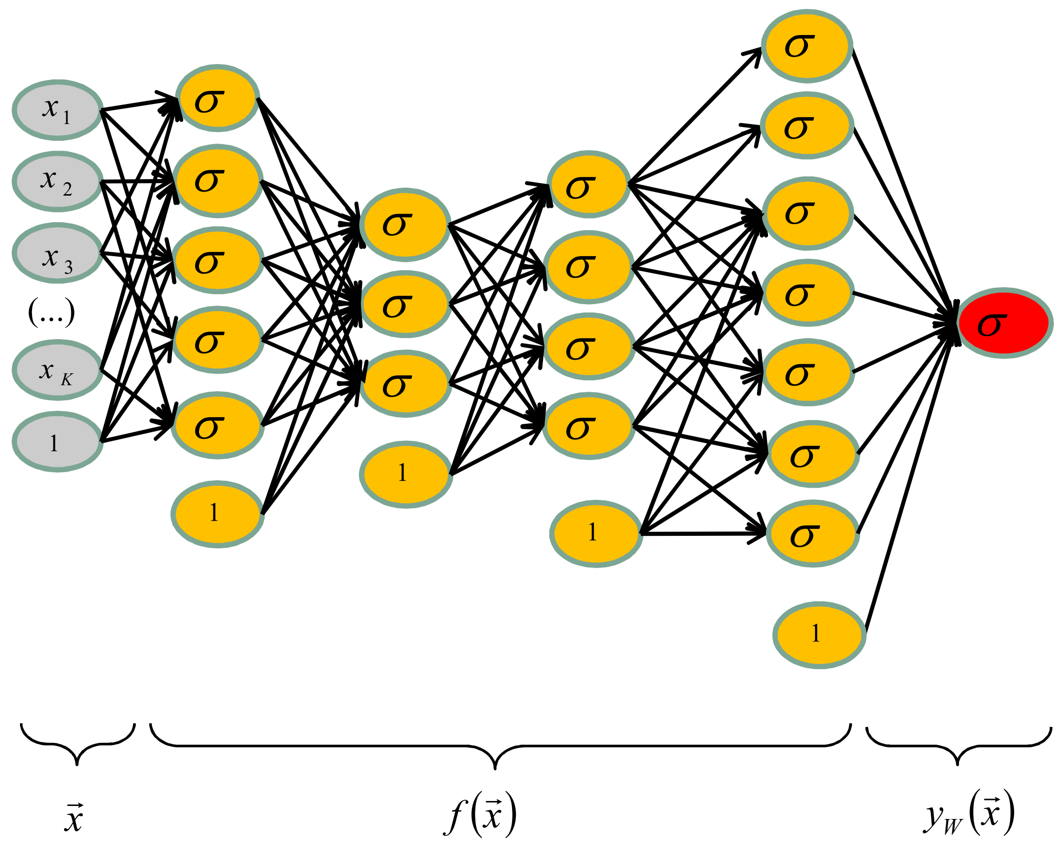
$$W^{[3]} \in R^{7\times 5}$$

$$\vec{w}^{[4]} \in R^{8}$$

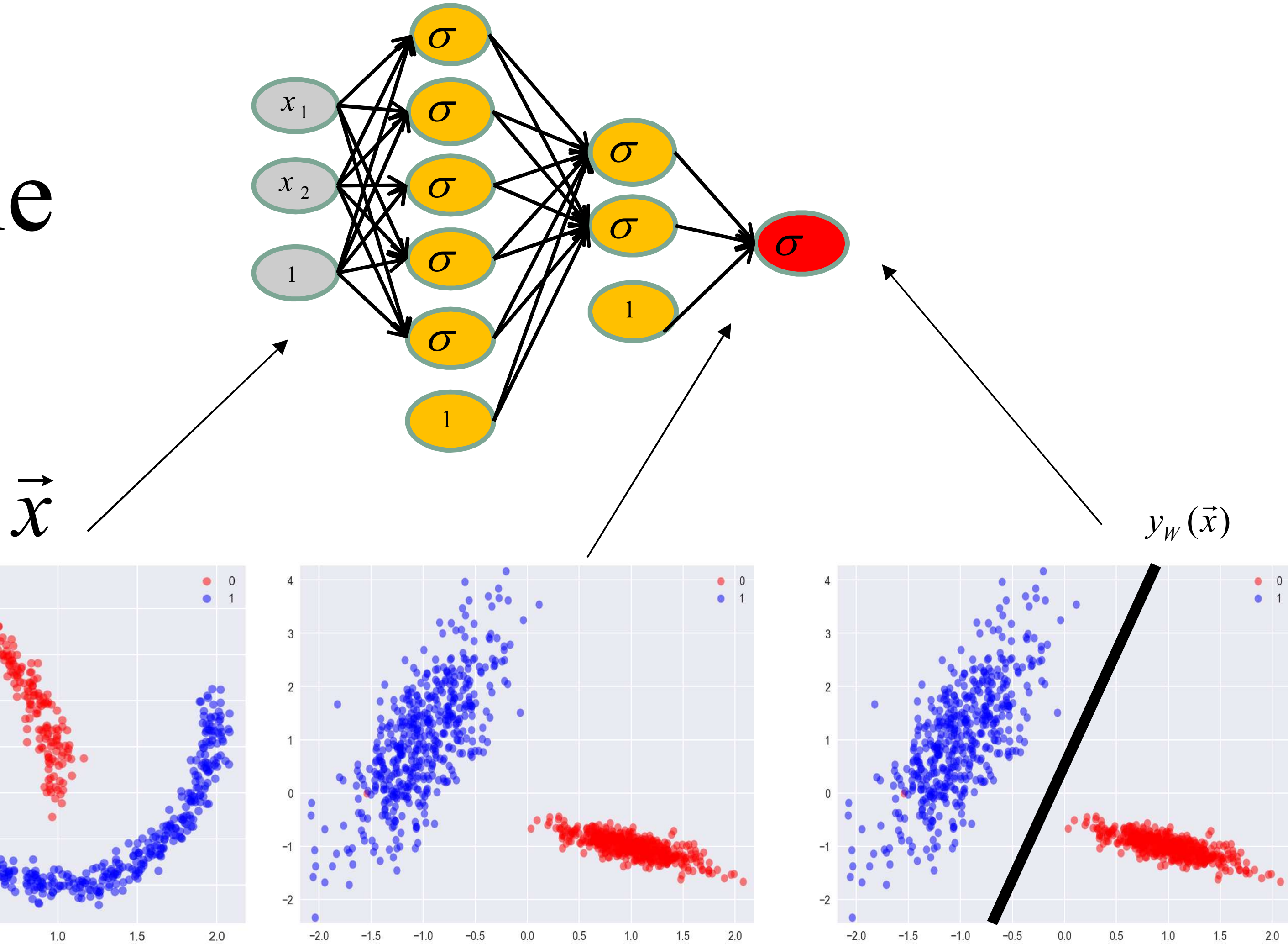
$$y_W(\vec{x}) = \sigma\left(\vec{w}^{[4]T}\sigma\left(W^{[3]}\sigma\left(W^{[2]}\sigma\left(W^{[1]}\sigma\left(W^{[0]}\vec{x}\right)\right)\right)\right)\right)$$

NOTE : More hidden layers = **deeper** network = **more capacity**.

Multilayer Perceptron



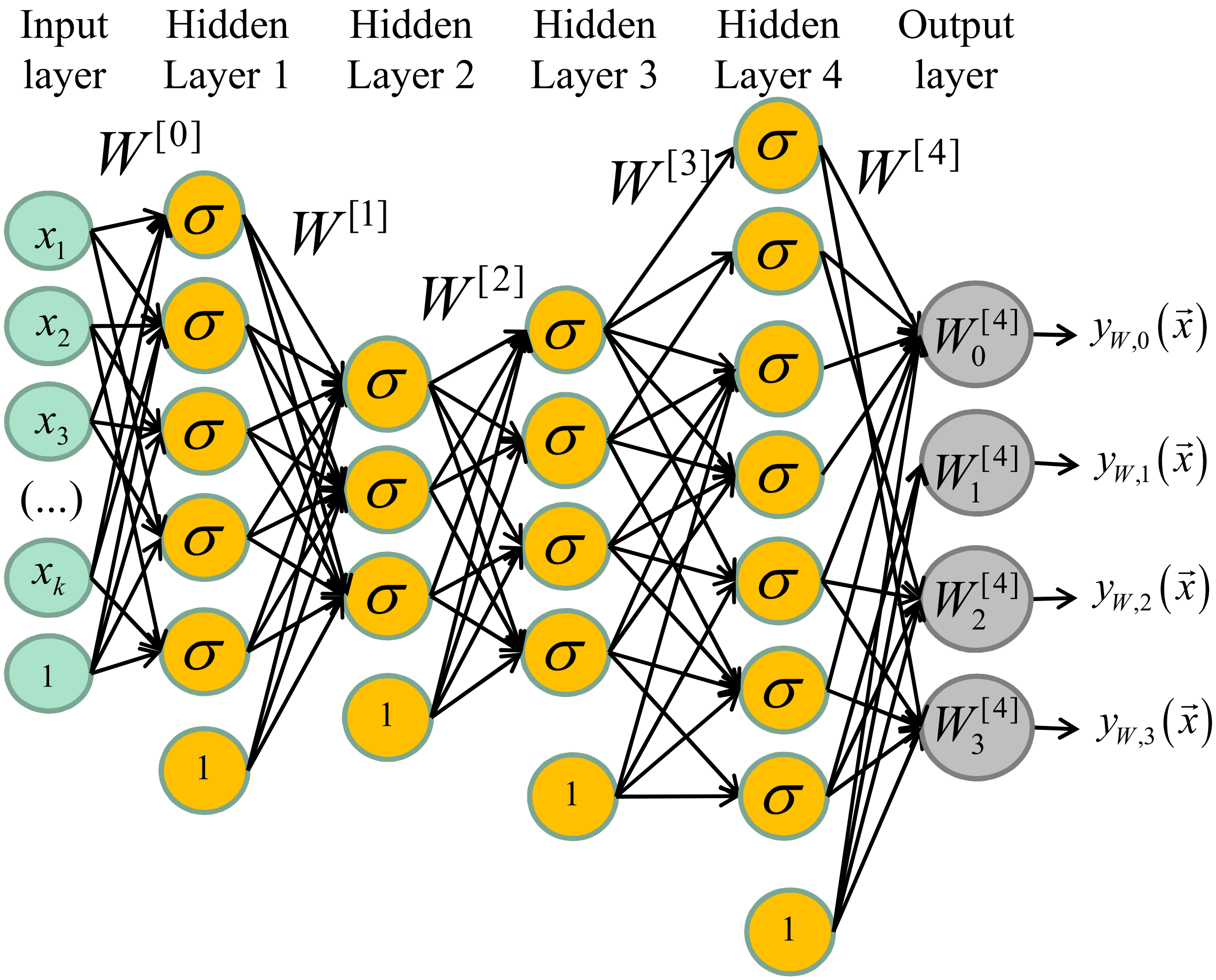
Example



Input data | Output of the last layer | Output of the network

A **K-Class** neural network has **K output** neurons.

kD, **4 Classes**, 4 hidden layer network



$W^{[0]} \in R^{5\times k+1}$

$W^{[1]} \in R^{3\times 6}$

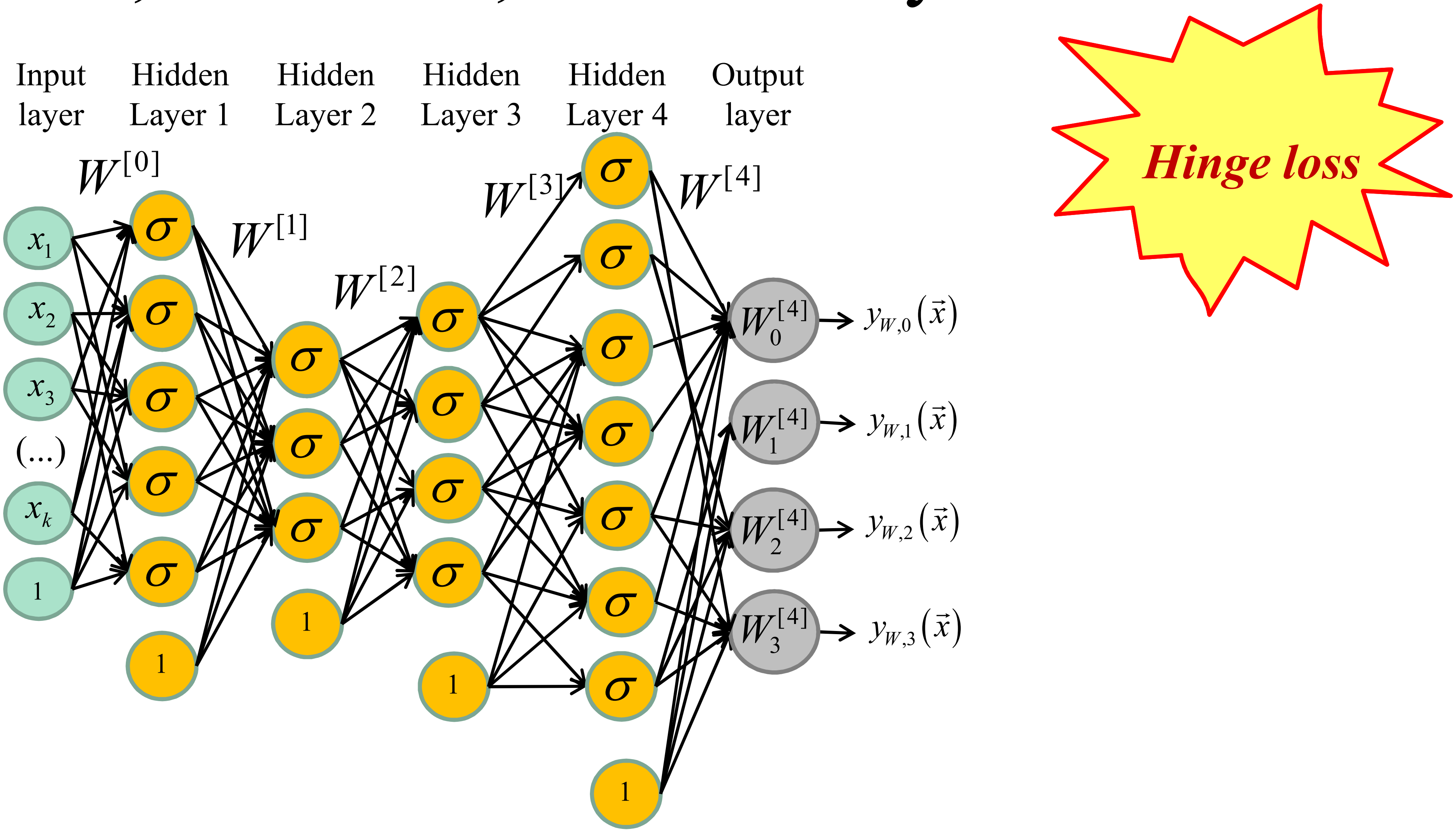
$W^{[2]} \in R^{4\times 4}$

$W^{[3]} \in R^{7\times 5}$

$W^{[4]} \in R^{8\times 4}$

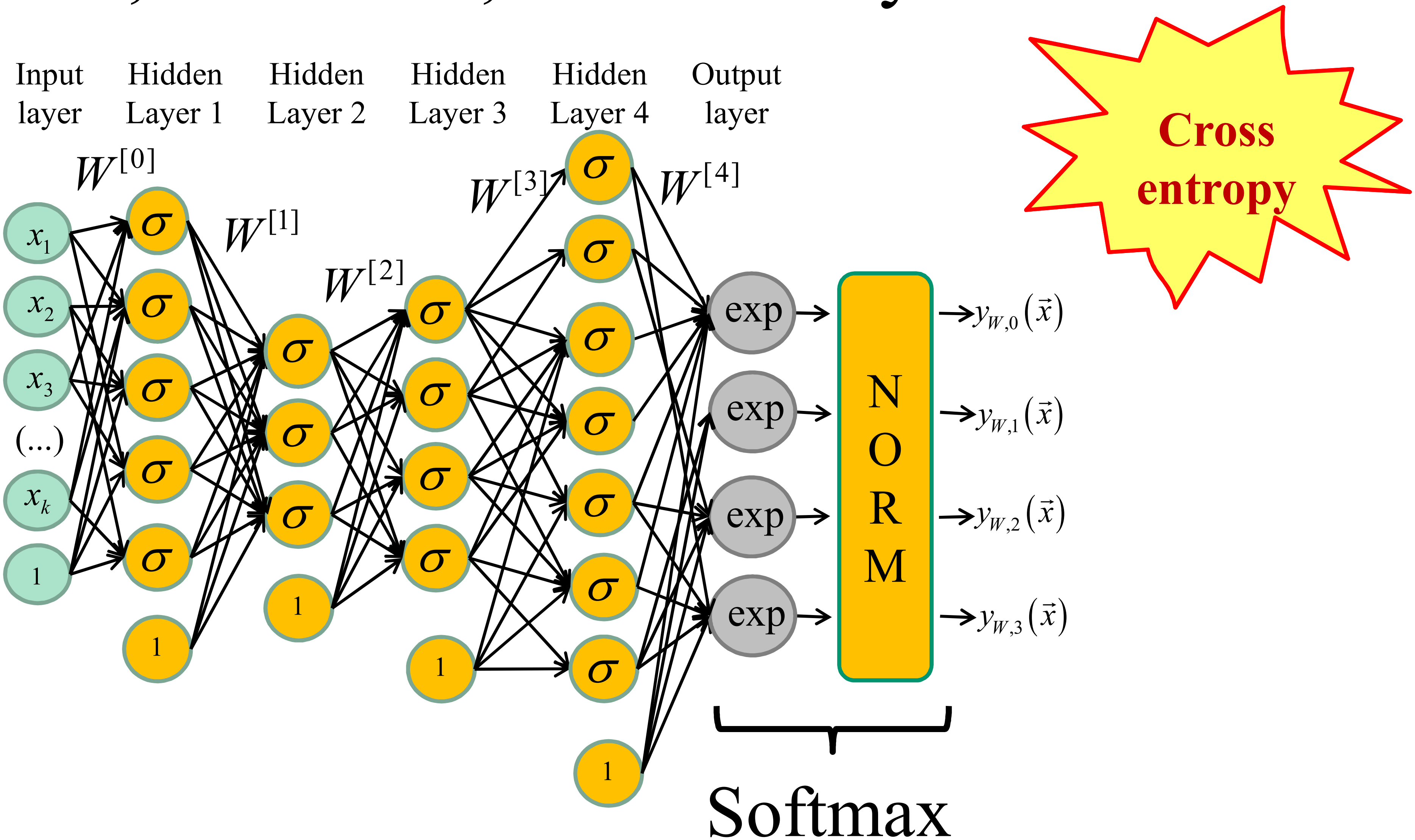
$$y_W(\vec{x}) = W^{[4]}\sigma\left(W^{[3]}\sigma\left(W^{[2]}\sigma\left(W^{[1]}\sigma\left(W^{[0]}\vec{x}\right)\right)\right)\right)$$

kD, **4 Classes**, 4 hidden layer network



$$y_W(\vec{x}) = W^{[4]}\sigma\left(W^{[3]}\sigma\left(W^{[2]}\sigma\left(W^{[1]}\sigma\left(W^{[0]}\vec{x}\right)\right)\right)\right)$$

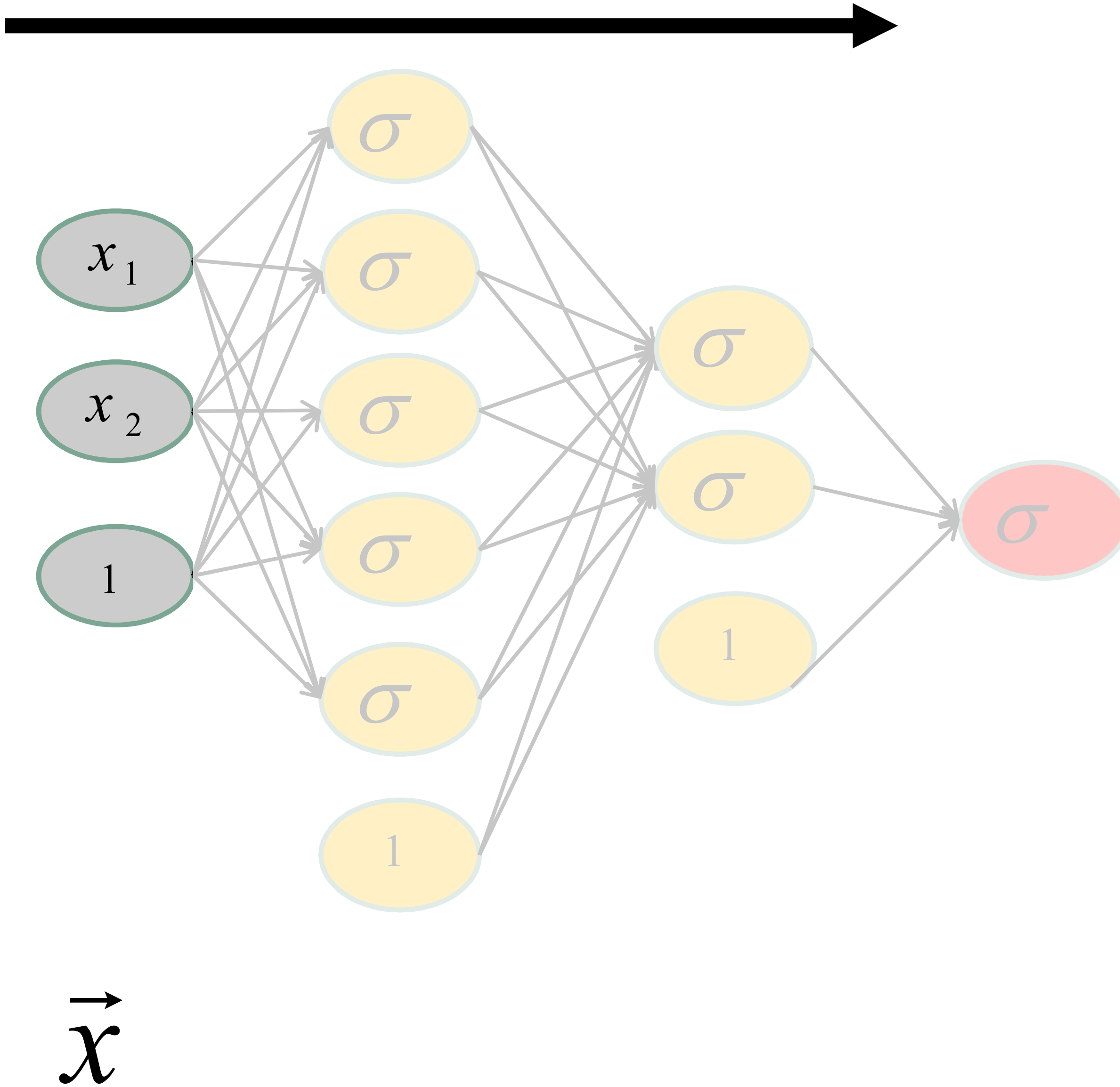
kD, **4 Classes**, 4 hidden layer network



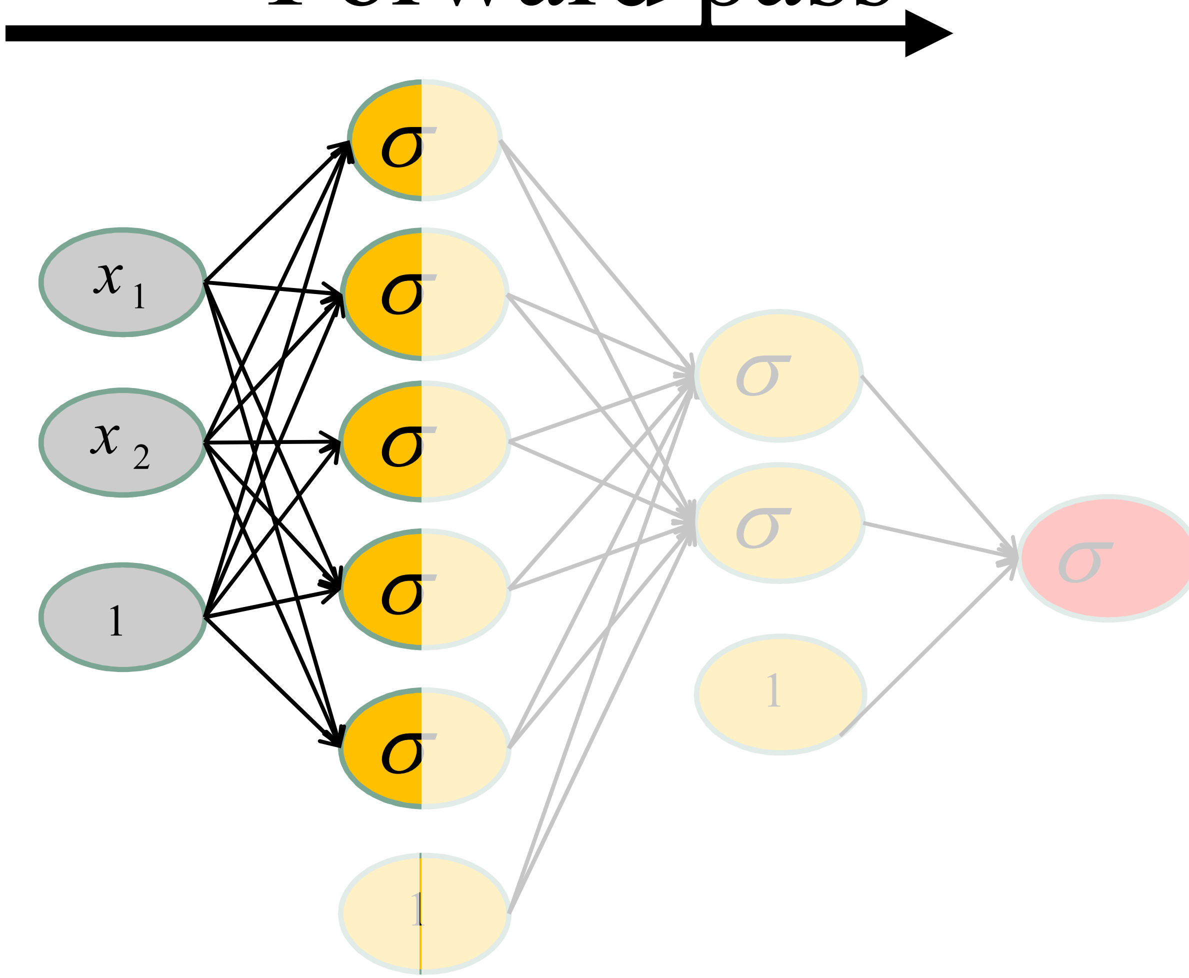
$$y_W(\vec{x}) = \mathrm{softmax}\left(W^{[4]}\sigma\left(W^{[3]}\sigma\left(W^{[2]}\sigma\left(W^{[1]}\sigma\left(W^{[0]}\vec{x}\right)\right)\right)\right)\right)$$

How to make a prediction?

Forward pass

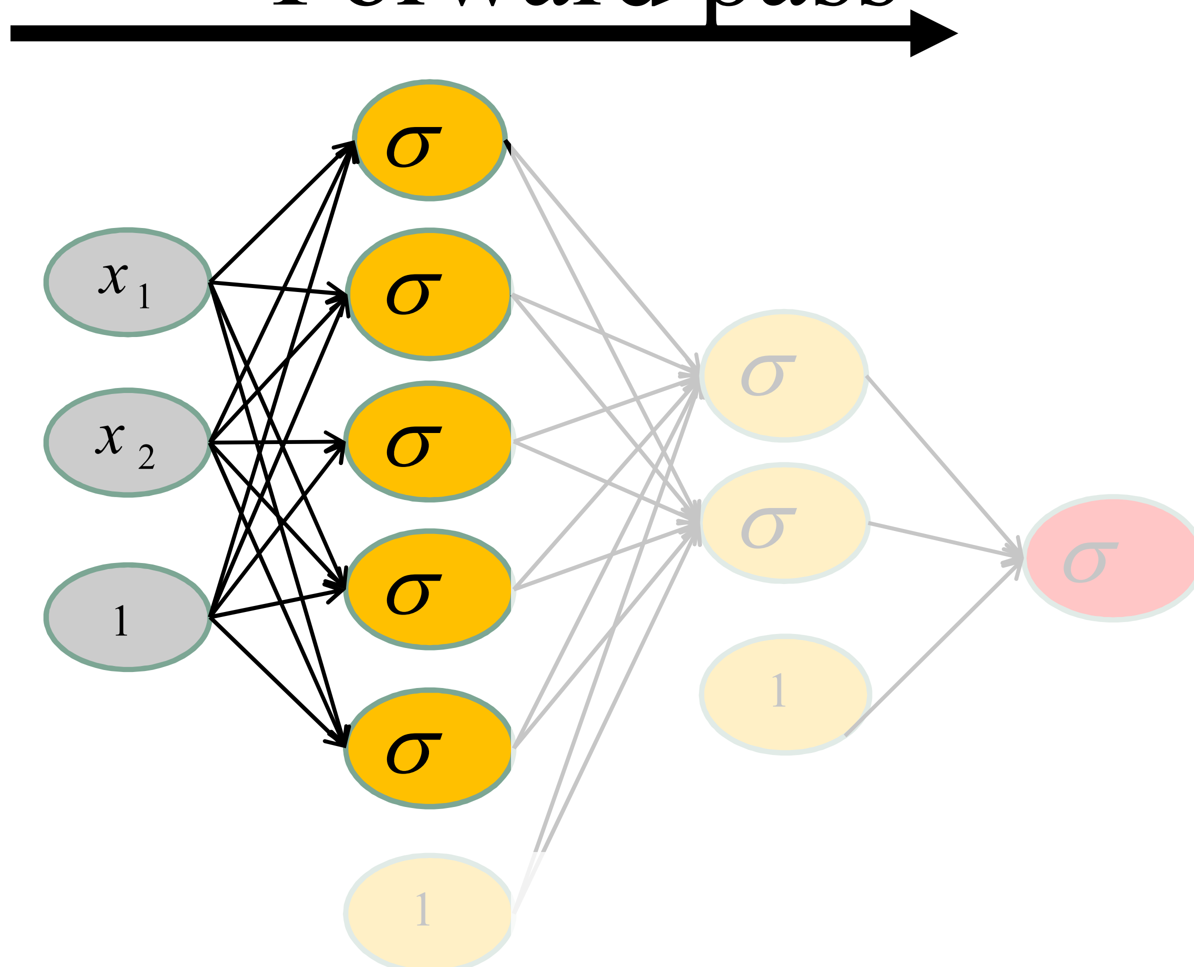


Forward pass



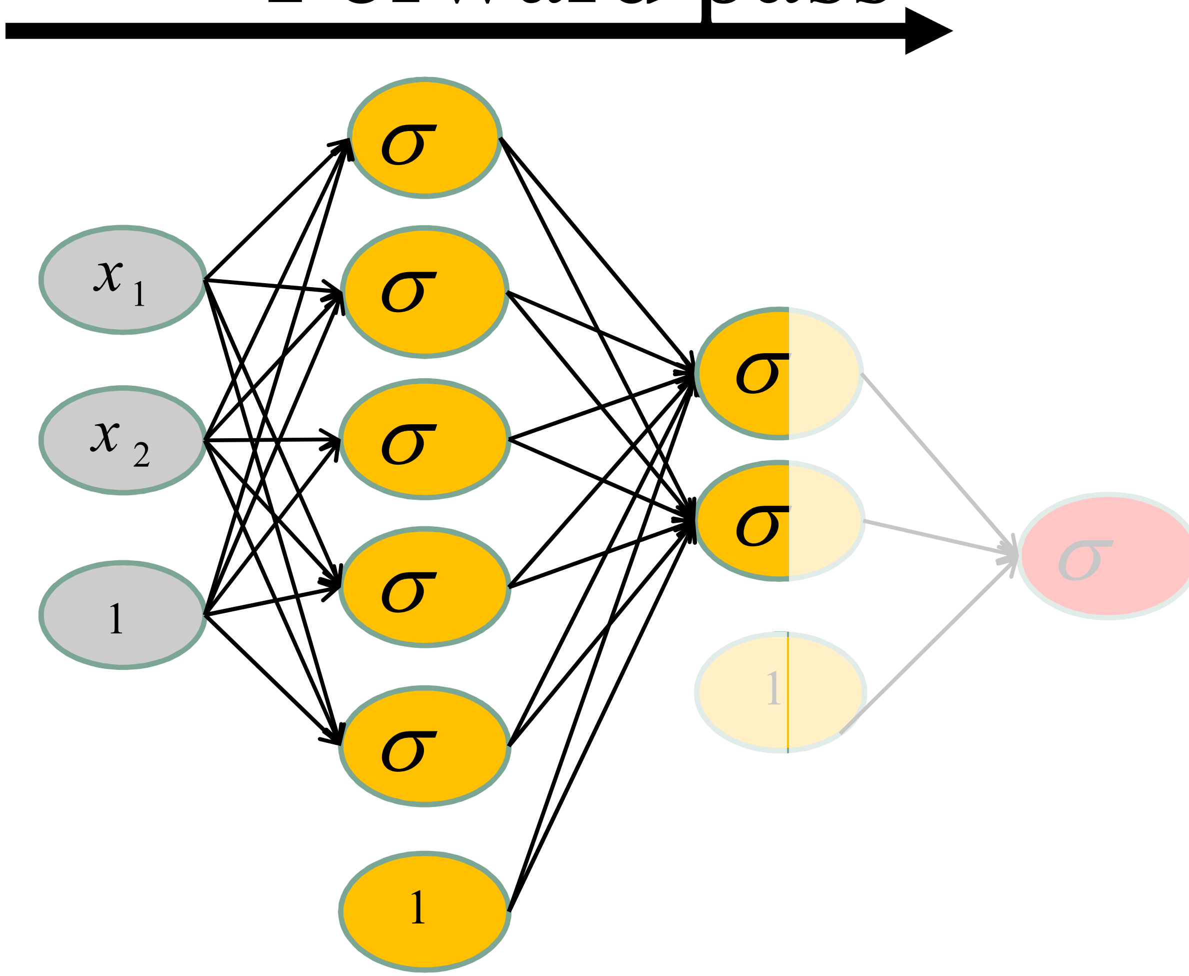
$W^{[0]}\vec{x}$

Forward pass



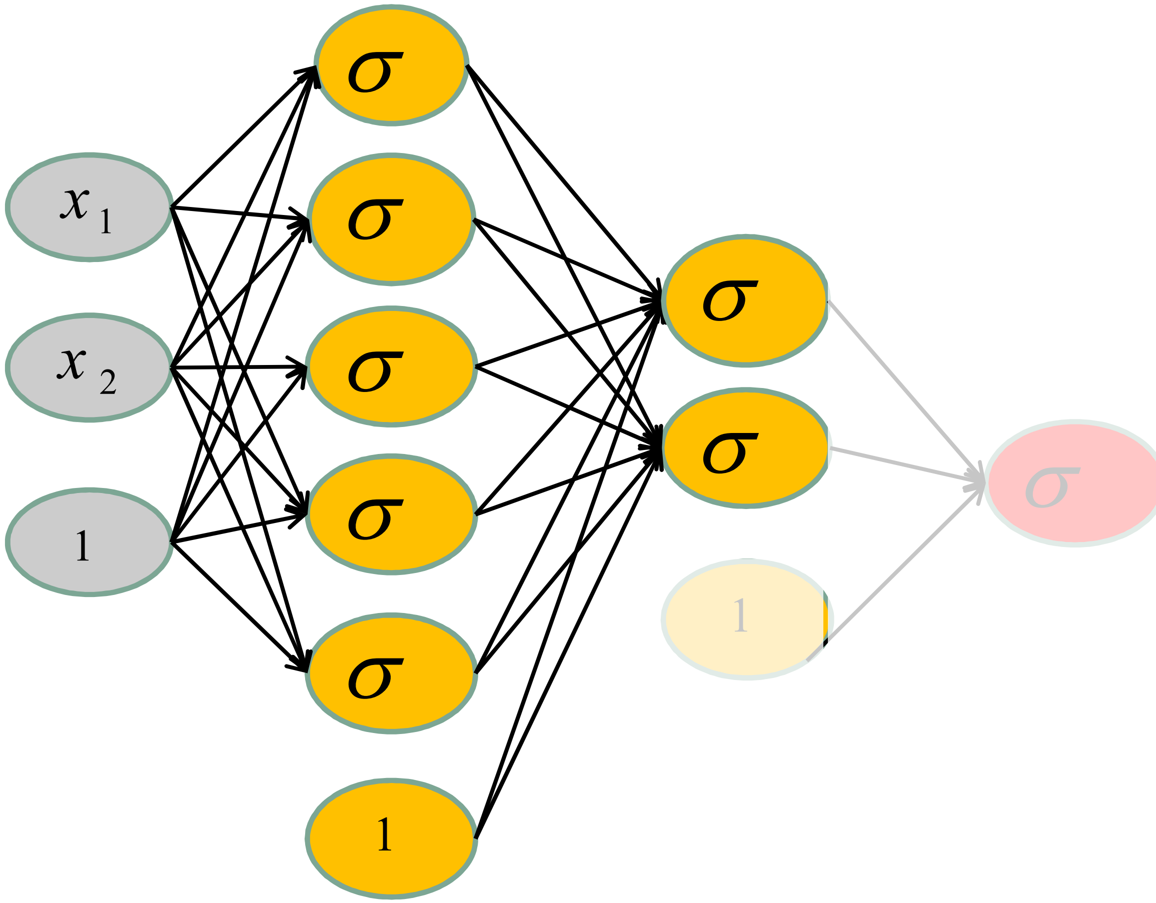
$$\sigma\left(W^{[0]}\vec{x}\right)$$

Forward pass



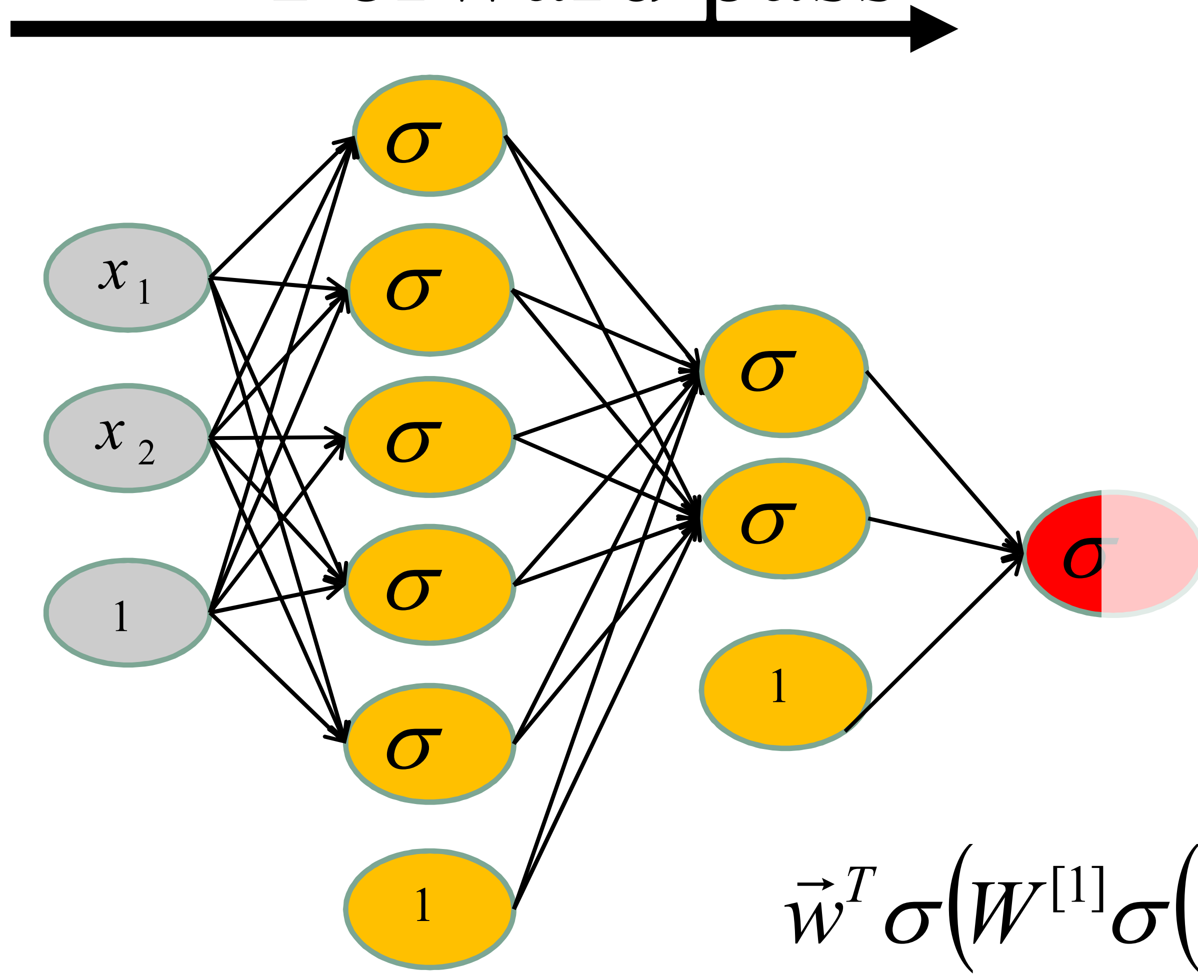
$$W^{[1]}\sigma\left(W^{[0]}\vec{x}\right)$$

Forward pass



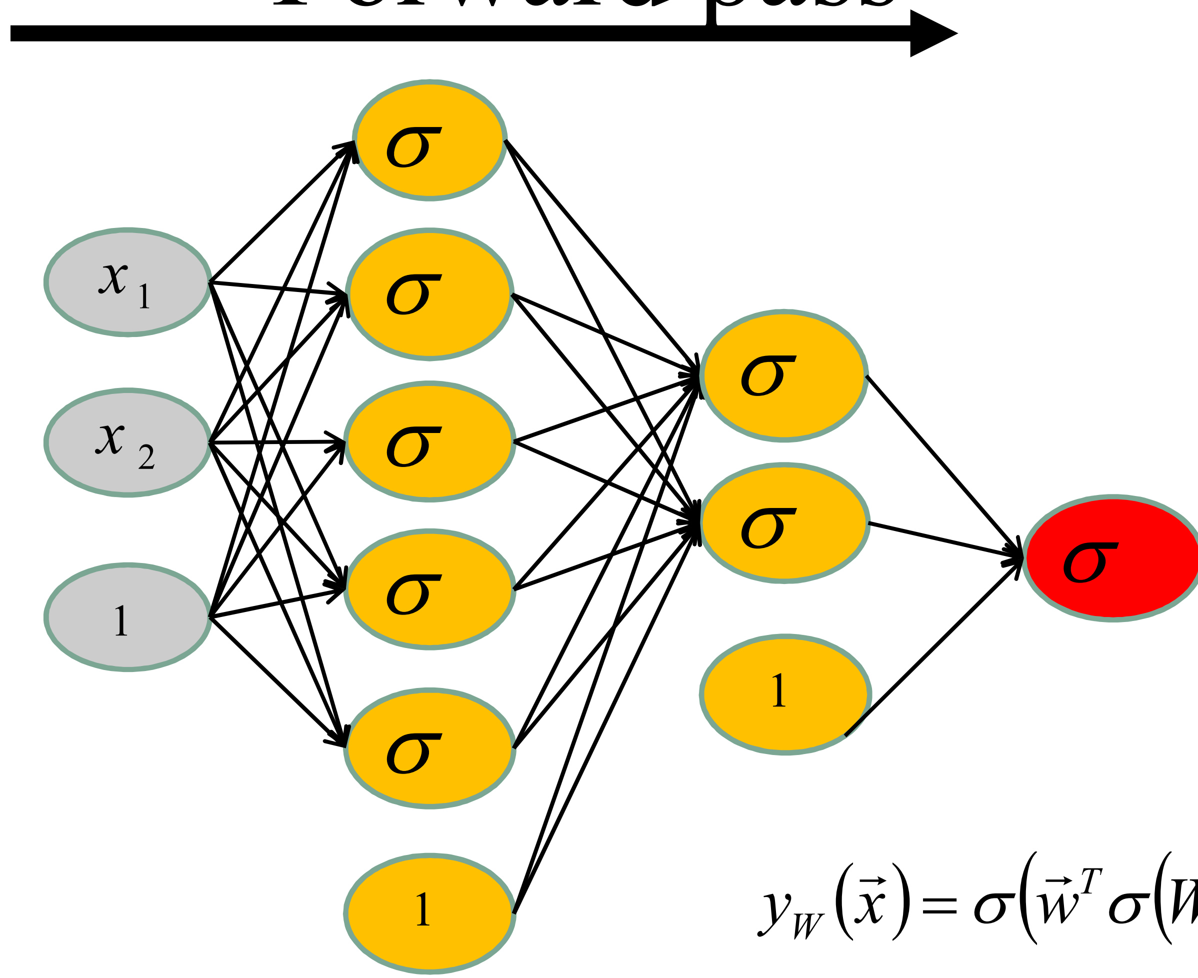
$$\sigma\left(W^{[1]}\sigma\left(W^{[0]}\vec{x}\right)\right)$$

Forward pass



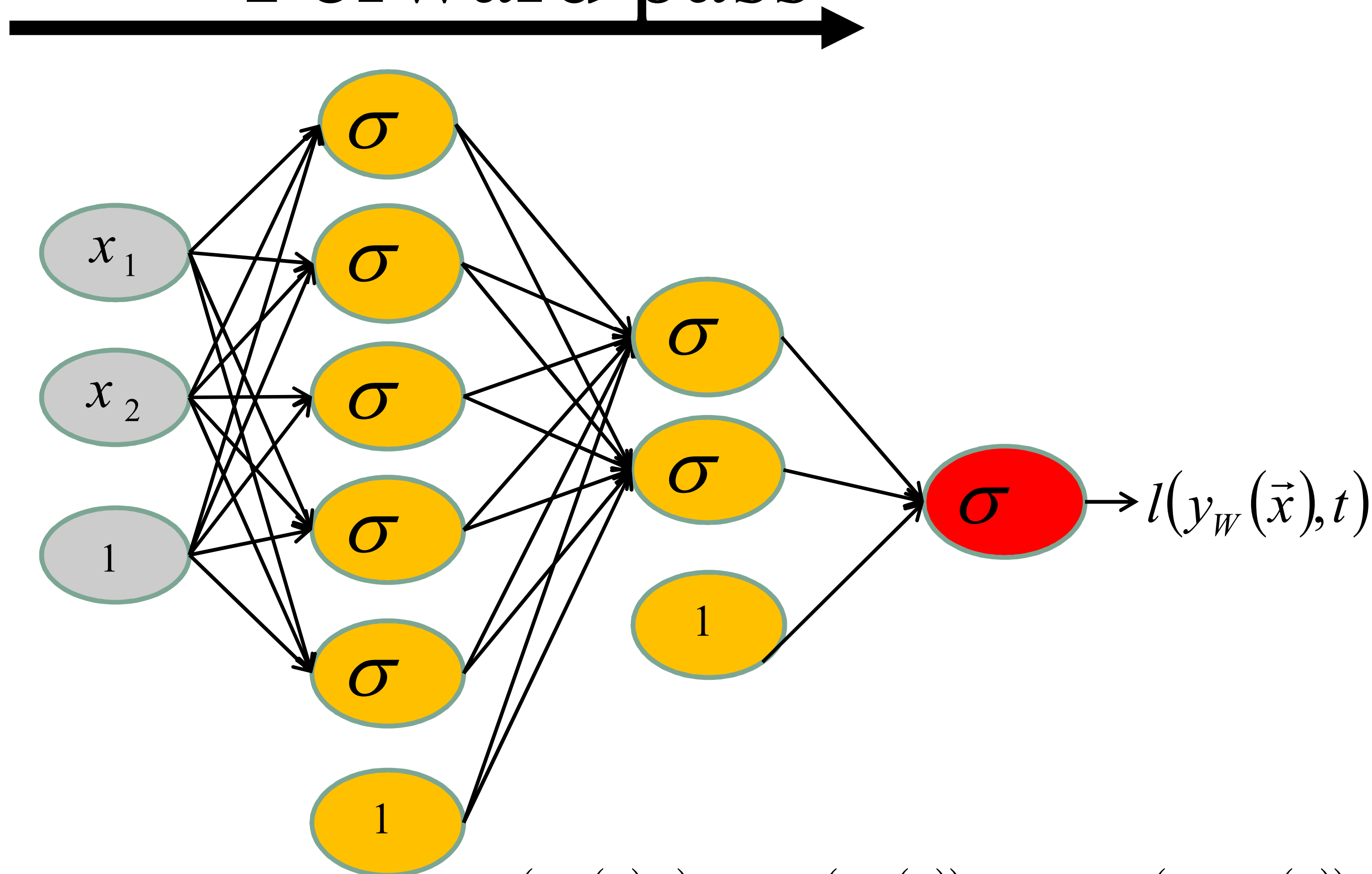
$$\vec{w}^T \sigma\left(W^{[1]}\sigma\left(W^{[0]}\vec{x}\right)\right)$$

Forward pass



$$y_W(\vec{x}) = \sigma\left(\vec{w}^T \sigma\left(W^{[1]}\sigma\left(W^{[0]}\vec{x}\right)\right)\right)$$

Forward pass



$$l(y_W(\vec{x}), t) = -t\ln(y_W(\vec{x})) - (1-t)\ln(1 - y_W(\vec{x}))$$

How to optimize the network?

0- From

$$W = \arg\min_{W} = \sum_{n=1}^{N} l\left(y_W\left(\vec{x}_n\right), t_n\right) + \lambda R\left(W\right)$$

Choose a regularization function

$$R\left(W\right) = \left\|W\right\|_1 \text{ or } \left\|W\right\|_2$$

How to optimize the network?

1- Choose a loss $l\left(y_W\left(\vec{x}_n\right), t_n\right)$ for example

Hinge loss
Cross entropy

Do not forget to adjust the output layer with the loss you have choosen.

cross entropy => Softmax

How to optimize the network?

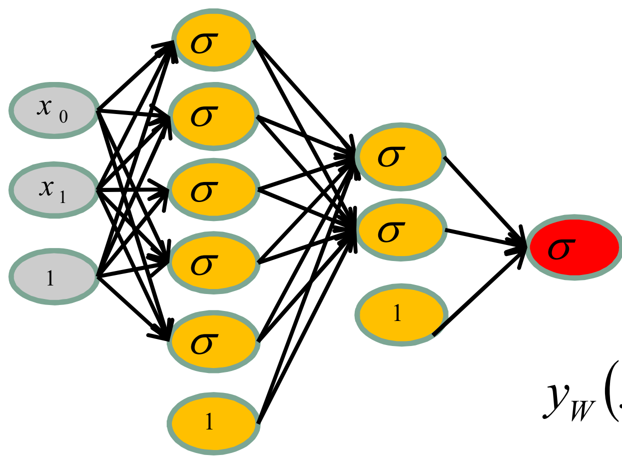
2- Compute the gradient of the loss with respect to each parameter

$$\frac{\partial\left(\sum_{n=1}^{N} l\left(y_W\left(\vec{x}_n\right), t_n\right) + \lambda R(W)\right)}{\partial W_{a,b}^{[c]}}$$

and launch a <u>gradient descent</u> algorithm to update the parameters.

$$W_{a,b}^{[c]} = W_{a,b}^{[c]} - \eta \frac{\partial\left(\sum_{n=1}^{N} l\left(y_W\left(\vec{x}_n\right), t_n\right) + \lambda R(W)\right)}{\partial W_{a,b}^{[c]}}$$

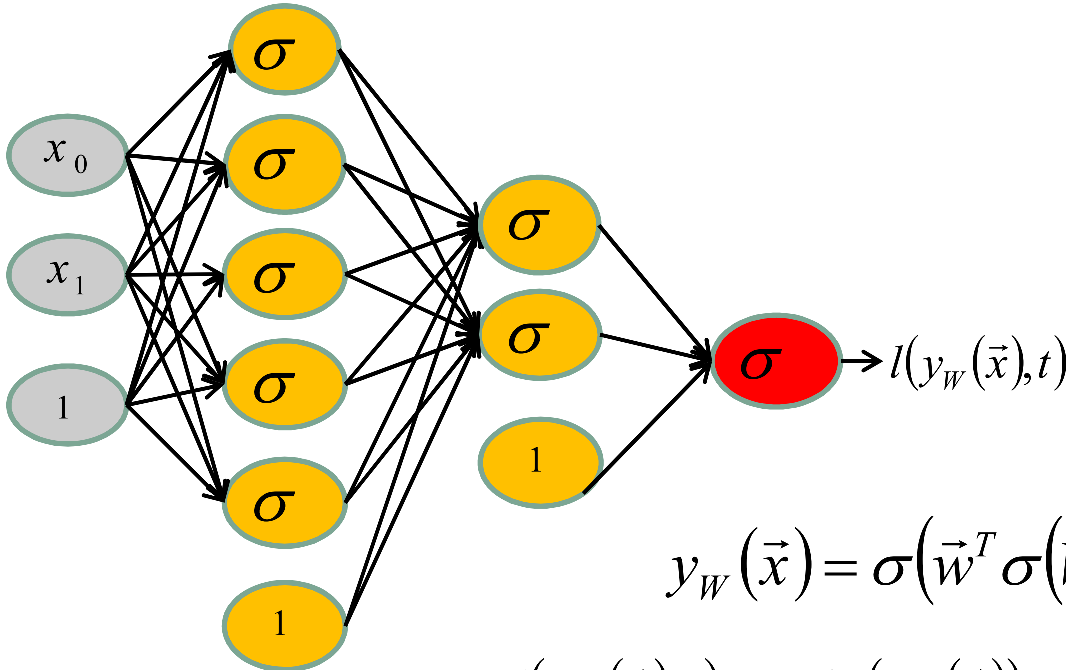
How to optimize the network?



$$y_W(\vec{x}) = \sigma\left(\vec{w}^{[2]}\sigma\left(W^{[1]}\sigma\left(W^{[0]}\vec{x}\right)\right)\right)$$

$$\frac{\partial\left(\sum_{n=1}^{N} l\left(y_W(\vec{x}_n), t_n\right) + \lambda R(W)\right)}{\partial W_{a,b}^{[c]}}$$

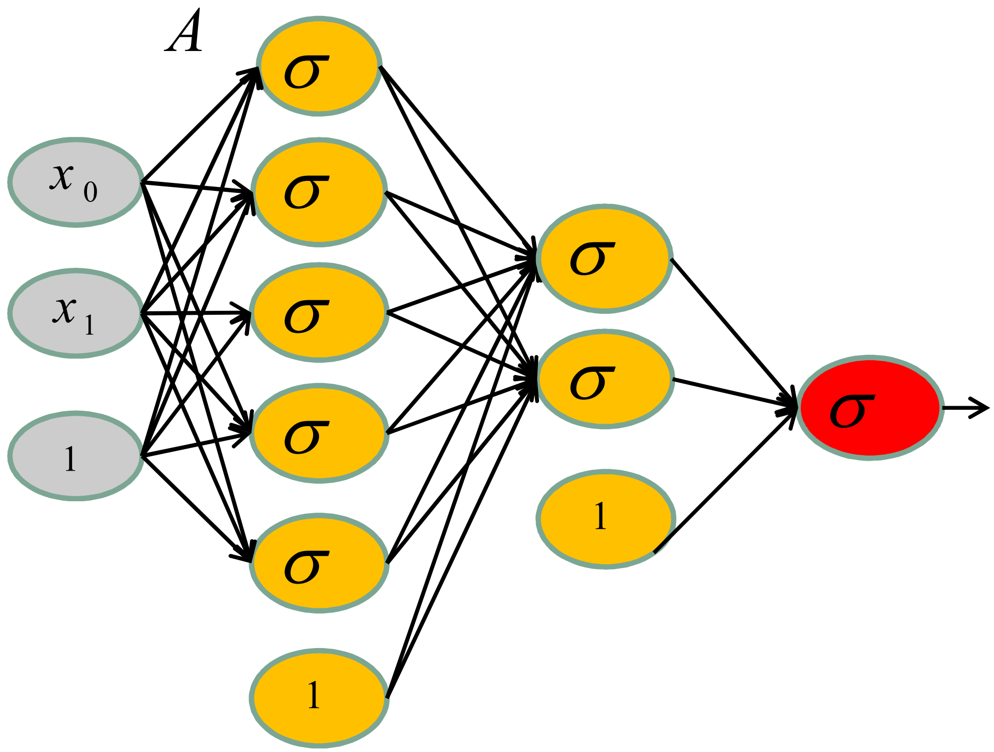
Backpropagation



$$y_W(\vec{x}) = \sigma\left(\vec{w}^T \sigma\left(W^{[1]} \sigma\left(W^{[0]} \vec{x}\right)\right)\right)$$

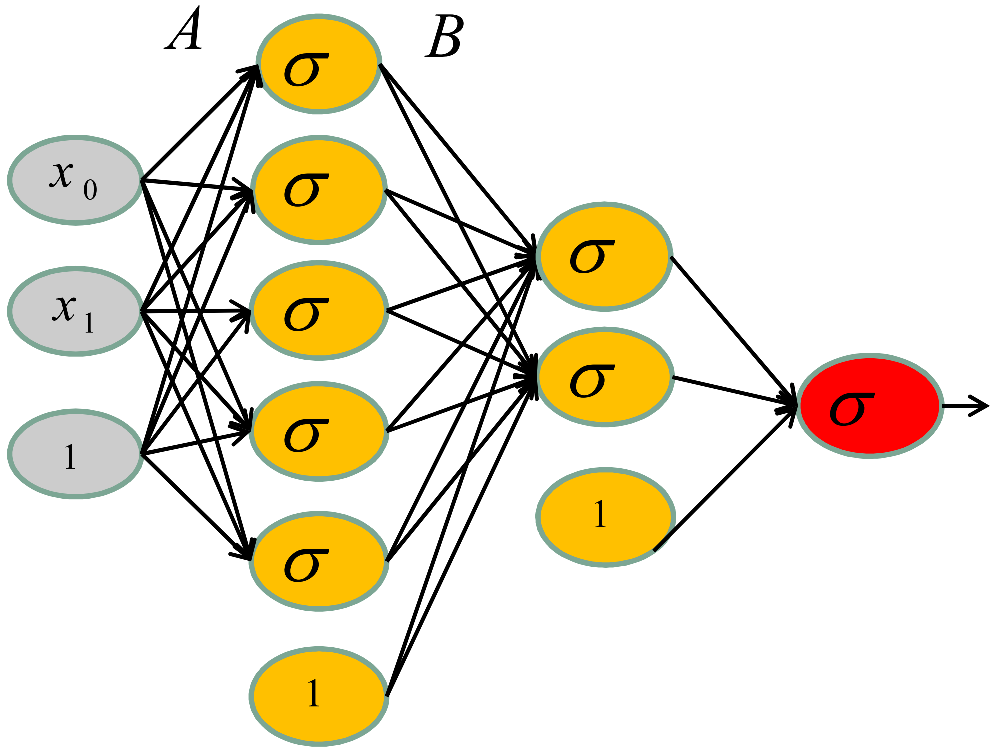
$$l(y_W(\vec{x}),t) = -t \ln(y_W(\vec{x})) - (1-t) \ln(1 - y_W(\vec{x}))$$

$$A = W^{[0]}\vec{x}$$

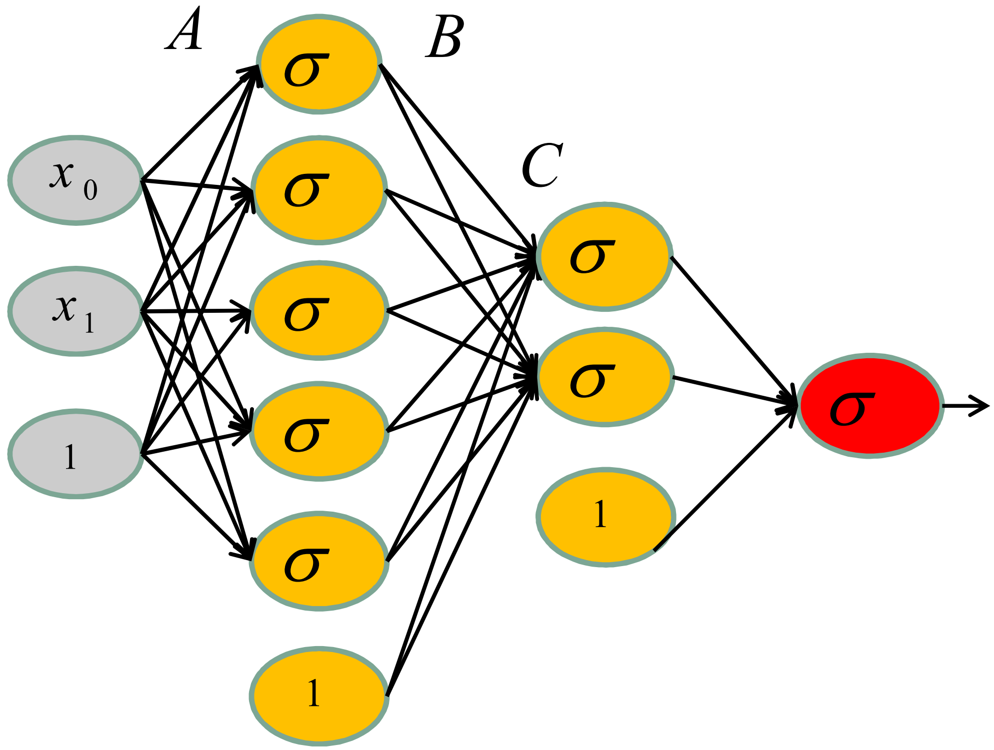


$$A = W^{[0]}\vec{x}$$

$$B = \sigma(A)$$



$$A = W^{[0]}\vec{x}$$
$$B = \sigma(A)$$
$$C = W^{[1]}B$$

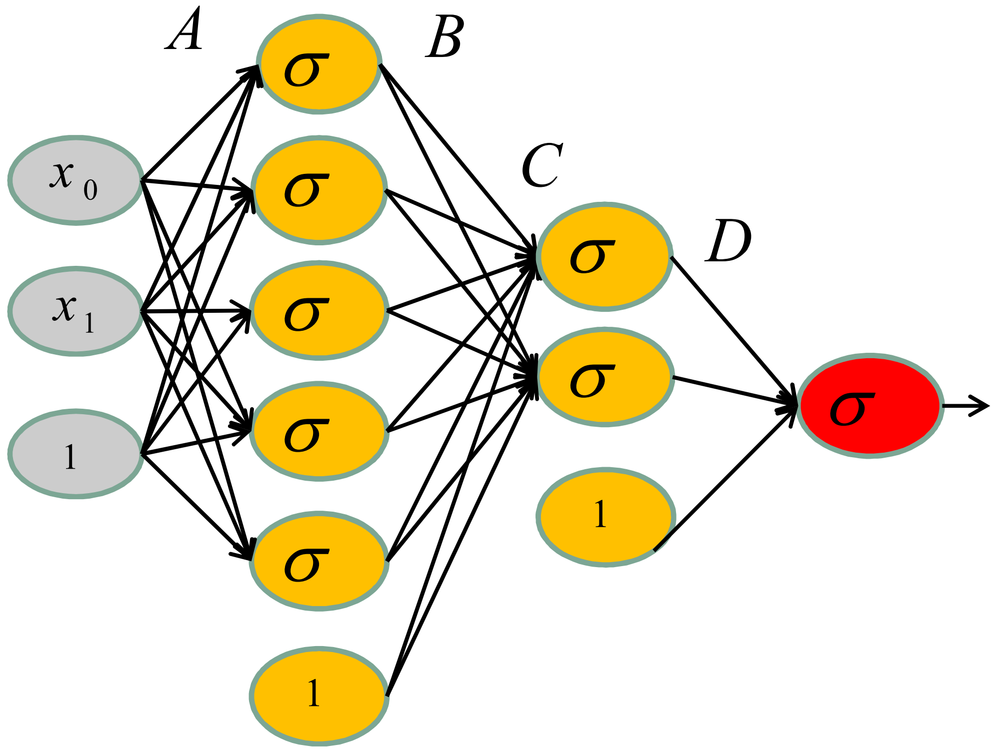


$$A = W^{[0]}\vec{x}$$

$$B = \sigma(A)$$

$$C = W^{[1]}B$$

$$D = \sigma(C)$$



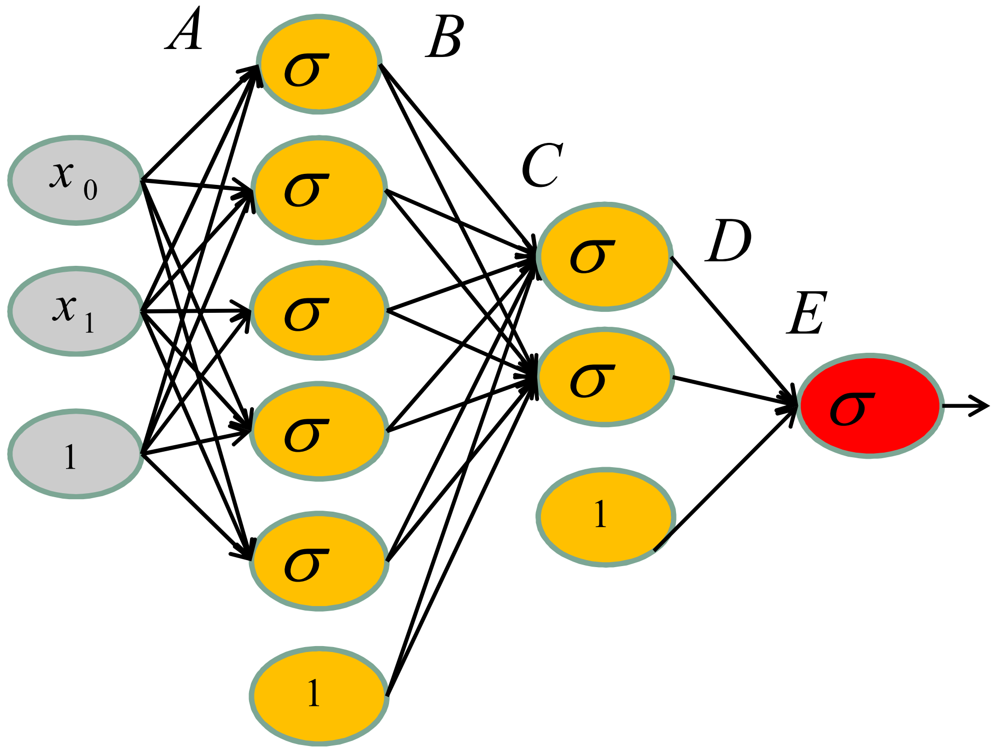
$$A = W^{[0]}\vec{x}$$

$$B = \sigma(A)$$

$$C = W^{[1]}B$$

$$D = \sigma(C)$$

$$E = \vec{w}^T D$$



$$A = W^{[0]}\vec{x}$$

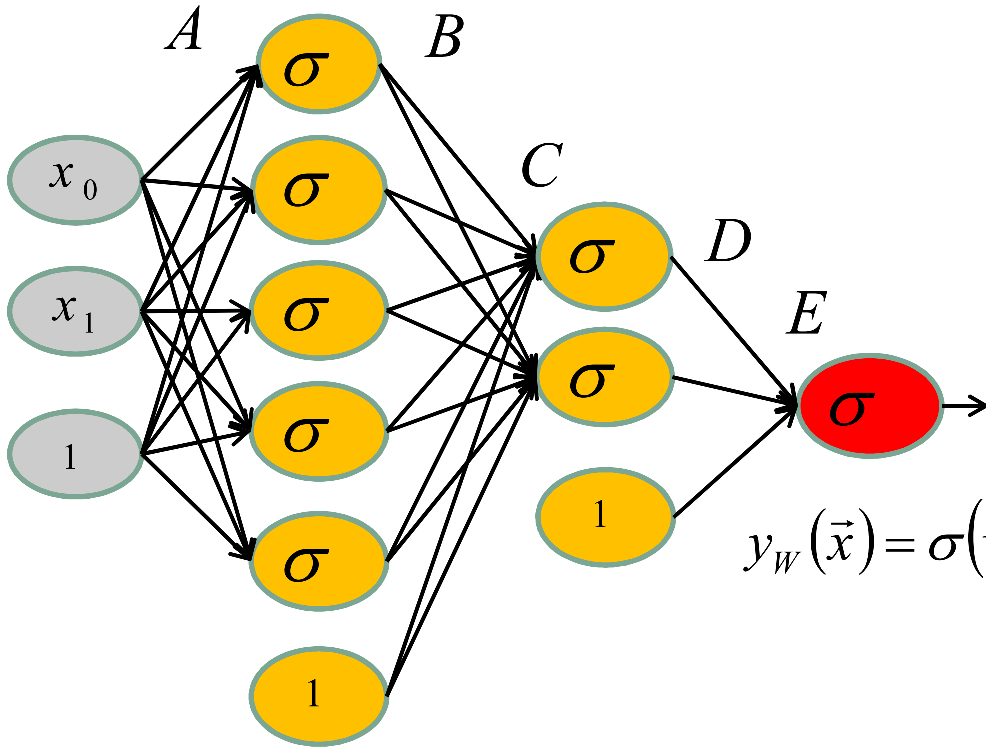
$$B = \sigma(A)$$

$$C = W^{[1]}B$$

$$D = \sigma(C)$$

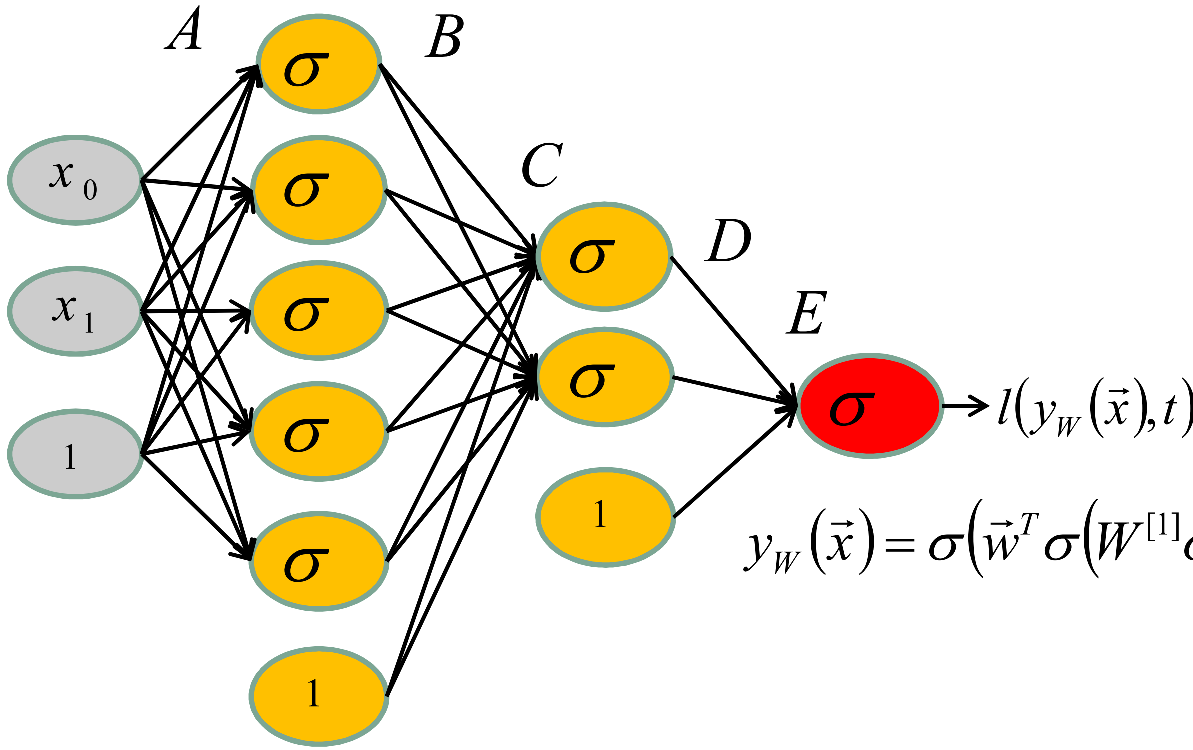
$$E = \vec{w}^T D$$

$$y_W(\vec{x}) = \sigma(E)$$



$$y_W(\vec{x}) = \sigma\left(\vec{w}^T \sigma\left(W^{[1]}\sigma\left(W^{[0]}\vec{x}\right)\right)\right)$$

$$A = W^{[0]}\vec{x}$$
$$B = \sigma(A)$$
$$C = W^{[1]}B$$
$$D = \sigma(C)$$
$$E = \vec{w}^T D$$
$$y_W(\vec{x}) = \sigma(E)$$
$$l(y_W(\vec{x}), t)$$



$$y_W(\vec{x}) = \sigma\left(\vec{w}^T\sigma\left(W^{[1]}\sigma\left(W^{[0]}\vec{x}\right)\right)\right)$$

$$\frac{\partial\left(\sum_{n=1}^{N} l\left(y_W(\vec{x}_n), t_n\right)\right)}{\partial W^{[l]}} \quad ?$$

Chain rule

Chain rule recap

$$
\left.
\begin{aligned}
f(u) &= u^2 \\
u(v) &= 2v \\
v(x) &= 1/x
\end{aligned}
\right\}
\quad \frac{\partial f}{\partial x} = ? \quad
\left\}
\begin{aligned}
\frac{\partial f}{\partial x} &= \frac{\partial f}{\partial u} \times \frac{\partial u}{\partial v} \times \frac{\partial v}{\partial x} \\
&= 2u \times 2 \times \left(-\frac{1}{x^2} \right)
\end{aligned}
\right.
$$

$$A = W^{[0]}\vec{x}$$

$$B = \sigma(A)$$

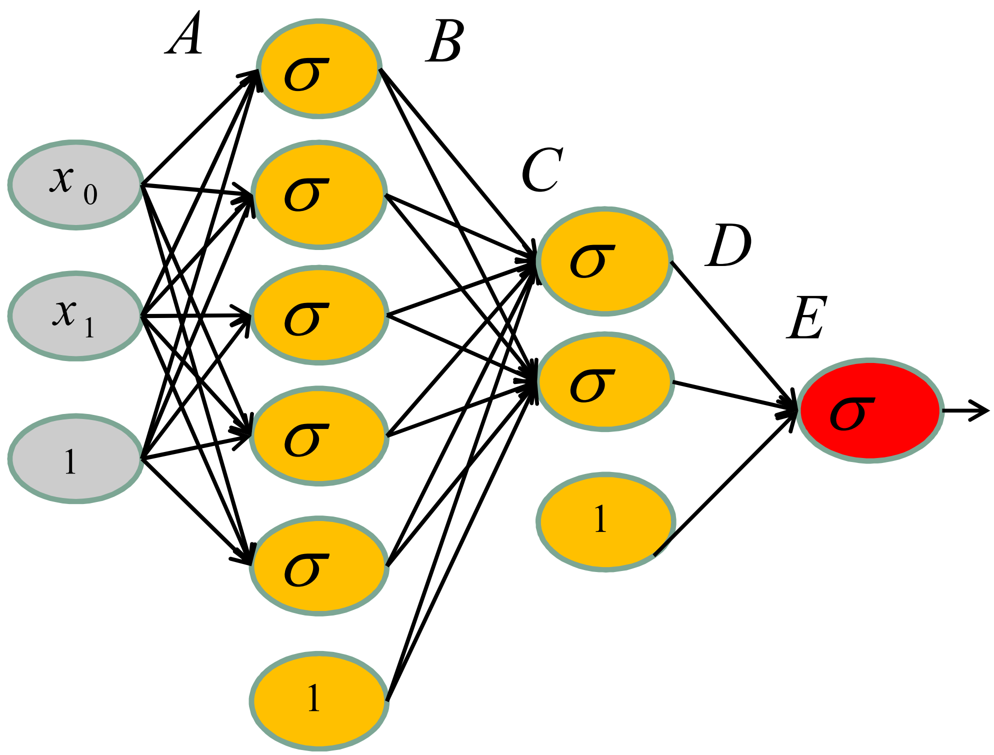
$$C = W^{[1]}B$$

$$D = \sigma(C)$$

$$E = \vec{w}^T D$$

$$y_W(\vec{x}) = \sigma(E)$$

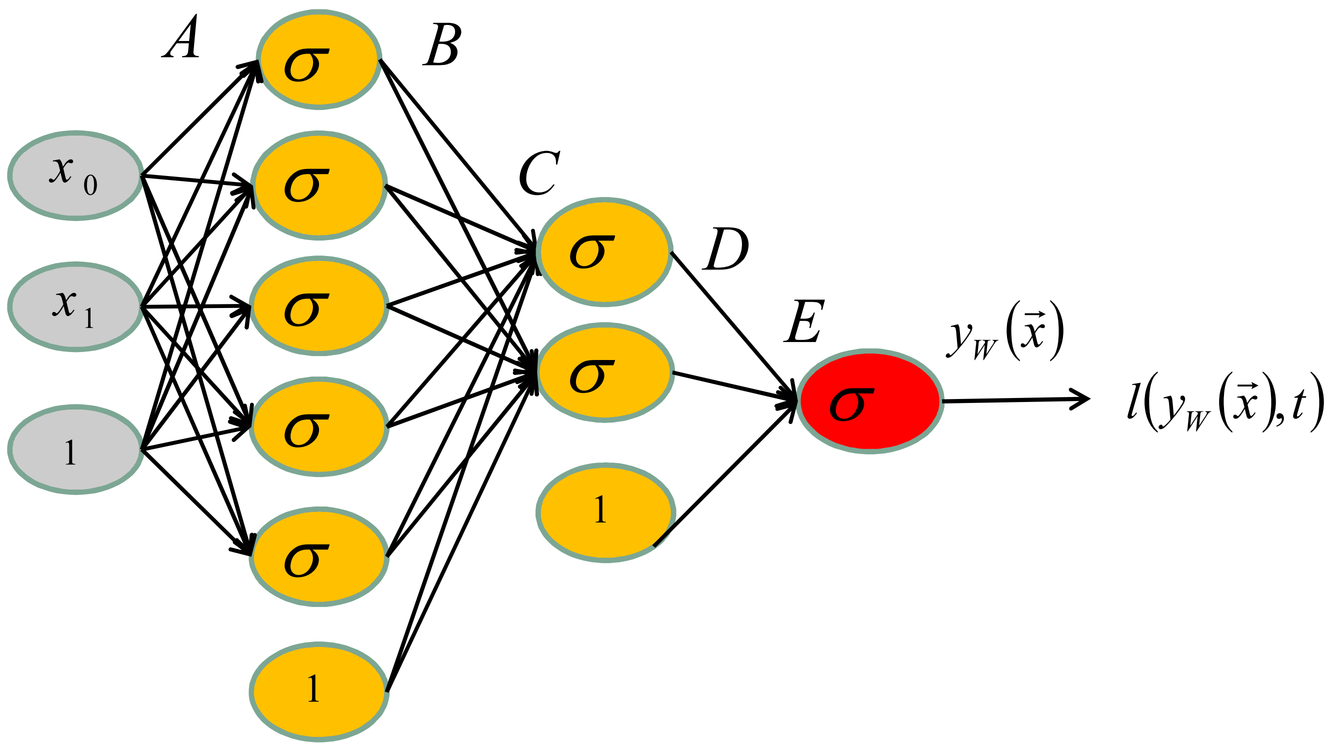
$$l(y_W(\vec{x}), t)$$



$$\frac{\partial(l(y_W(\vec{x}),t))}{\partial W^{[0]}} = \frac{\partial(l(y_W(\vec{x}),t))}{\partial y_W(\vec{x})} \frac{\partial(y_W(\vec{x}))}{\partial E} \frac{\partial(E)}{\partial D} \frac{\partial(D)}{\partial C} \frac{\partial(C)}{\partial B} \frac{\partial(B)}{\partial A} \frac{\partial(A)}{\partial W^{[0]}}$$

Back propagation

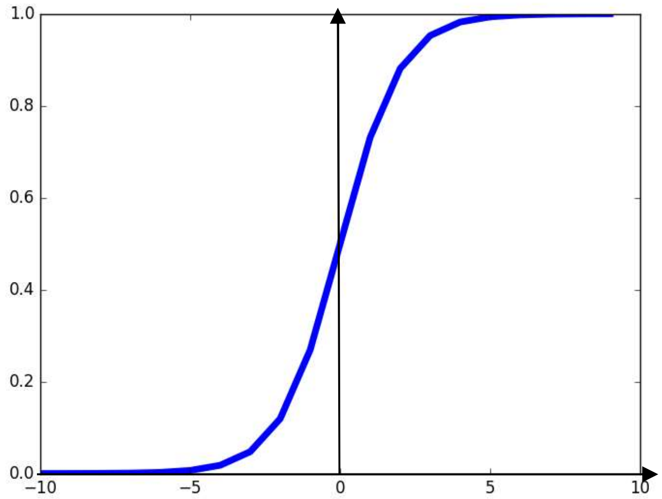
$$\frac{\partial\left(l\left(y_W(\vec{x}),t\right)\right)}{\partial W^{[0]}}=\frac{\partial\left(l\left(y_W(\vec{x}),t\right)\right)}{\partial y_W(\vec{x})}\frac{\partial\left(y_W(\vec{x})\right)}{\partial E}\frac{\partial(E)}{\partial D}\frac{\partial(D)}{\partial C}\frac{\partial(C)}{\partial B}\frac{\partial(B)}{\partial A}\frac{\partial(B)}{\partial W^{[0]}}$$



Activation functions

Activation functions

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

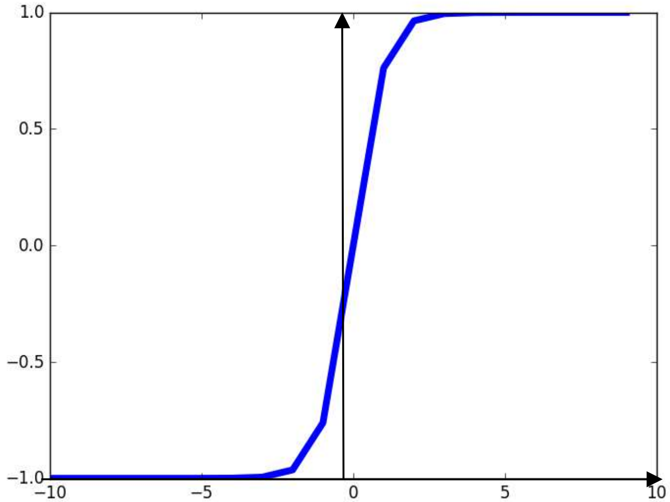


Sigmoïde

3 Problems :

- Gradient saturates when input is large
- Not zero centered
- *exp*() is an expensive operation

Activation functions



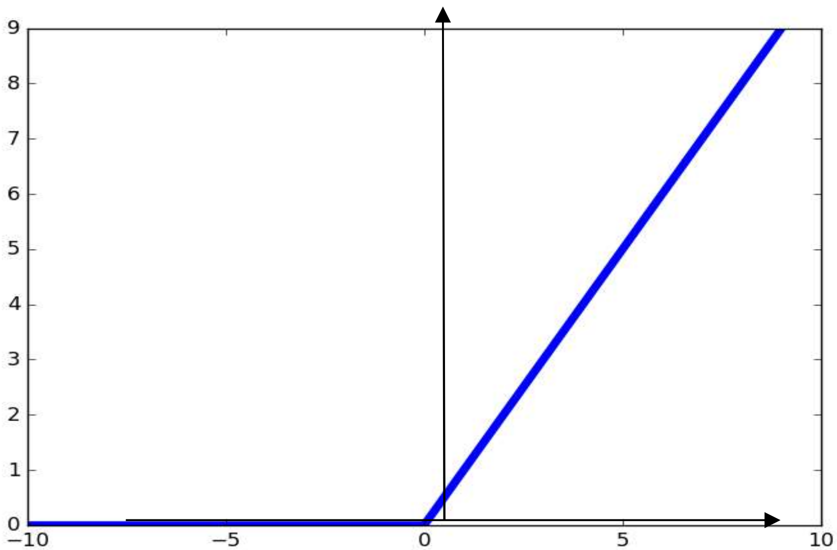
Tanh(x)

- **Output is zero-centered** ☺
- **Small gradient** when input is large ☹

[LeCun et al., 1991]

Activation functions

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



ReLU(x)

(Rectified Linear Unit)

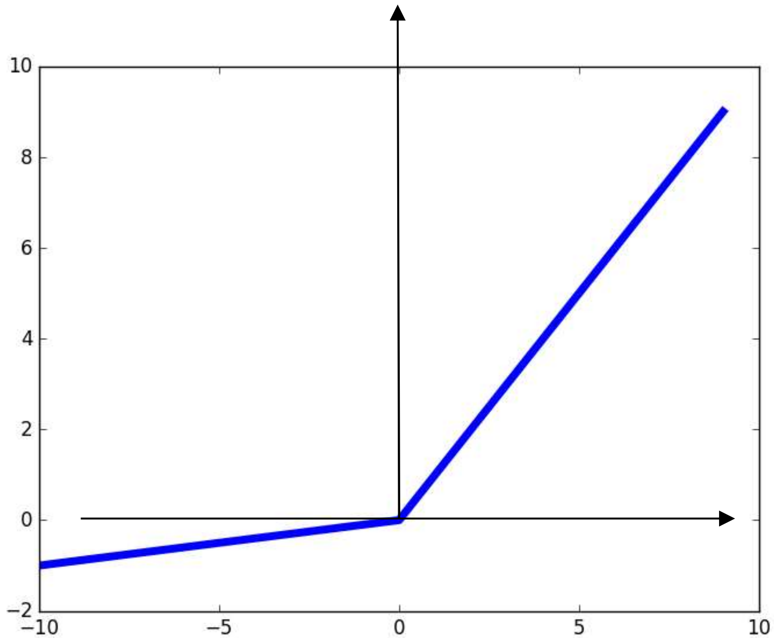
- **Large gradient** for x>0 ☺
- Super **fast** ☺

- Output **non centered at zero** ☹
- **No gradient** when x<0? ☹

[Krizhevsky et al., 2012]

Activation functions

$$\mathrm{LReLU}(x) = \max(0.01x, x)$$



Leaky ReLU(x)

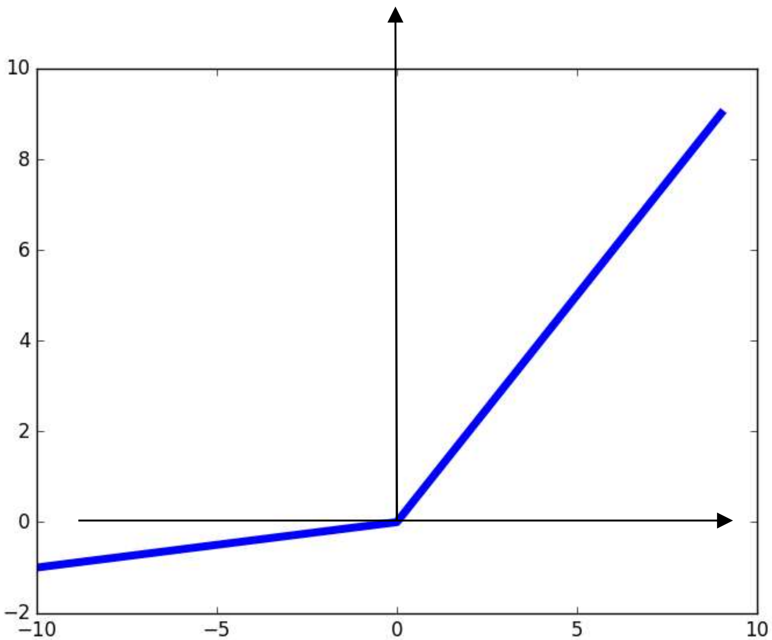
- no **gradient saturation** ☺
- Super **fast** ☺

- 0.01 is an **hyperparameter** 😐

[Mass et al., 2013]
[He et al., 2015]

Activation functions

$$\text{PReLU}(x) = \max(\alpha x, x)$$



Parametric ReLU(x)

- no **gradient saturation** ☺
- Super **fast** ☺

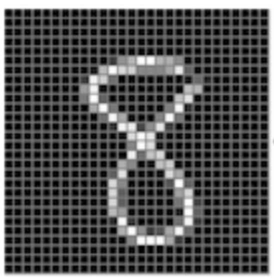
- α **learn** with back prop ☺

[Mass et al., 2013]
[He et al., 2015]

In practice

- By default, people use **ReLU**.
- Try **Leaky ReLU / PReLU / ELU**
- Try **tanh** but might be sub-optimal
- **Do not use sigmoïde** except at the output of a 2 class net.

How to classify an image?

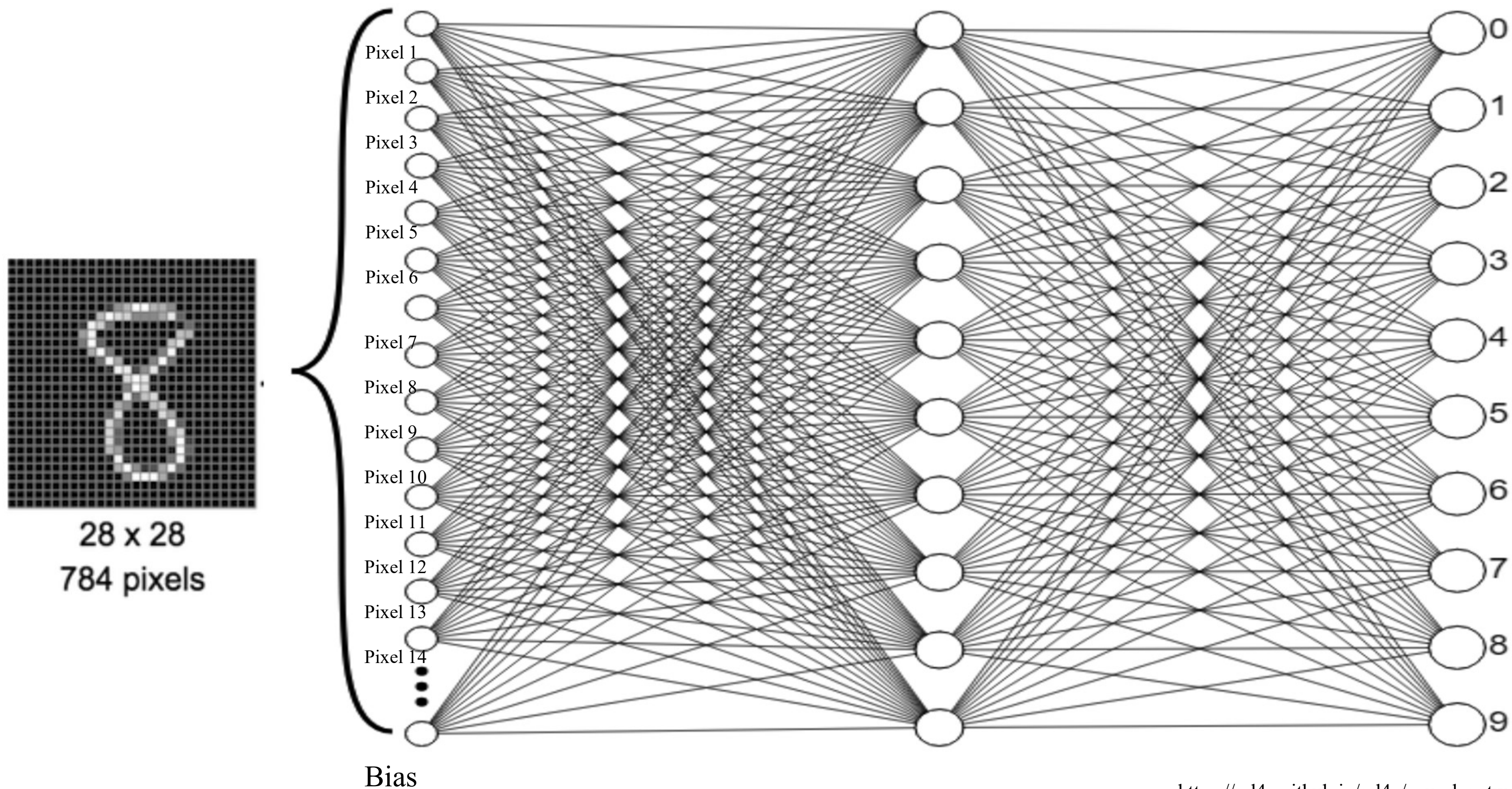


28 x 28
784 pixels

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
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
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
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
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
0	0	0	0	0	0	0	0	0	0	4	62	146	182	254	254	181	176	139	15	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	34	186	253	217	208	136	136	136	166	232	99	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	61	242	208	111	3	0	0	0	0	0	18	32	107	43	0	0	0	0	0	0
0	0	0	0	0	0	0	0	156	242	23	0	0	0	0	0	0	0	13	191	181	6	0	0	0	0	0	0
0	0	0	0	0	0	0	0	121	255	98	3	0	0	0	0	0	8	194	225	12	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	169	253	120	3	0	0	0	0	128	247	51	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	3	111	244	169	19	0	14	131	249	117	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	59	241	235	72	142	229	66	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	25	218	254	231	36	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	133	253	221	33	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	19	237	111	196	217	19	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	174	138	0	23	193	204	18	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	96	224	0	0	0	25	218	169	3	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	215	138	0	0	0	0	86	253	99	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	215	97	0	0	0	0	3	162	214	11	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	215	97	0	0	0	0	0	118	253	68	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	185	157	0	0	0	0	0	40	254	98	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	50	244	61	0	0	0	0	112	244	58	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	174	251	142	59	83	167	244	111	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	6	133	253	253	253	169	61	3	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	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	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	0	0	0	0	0	0	0	0

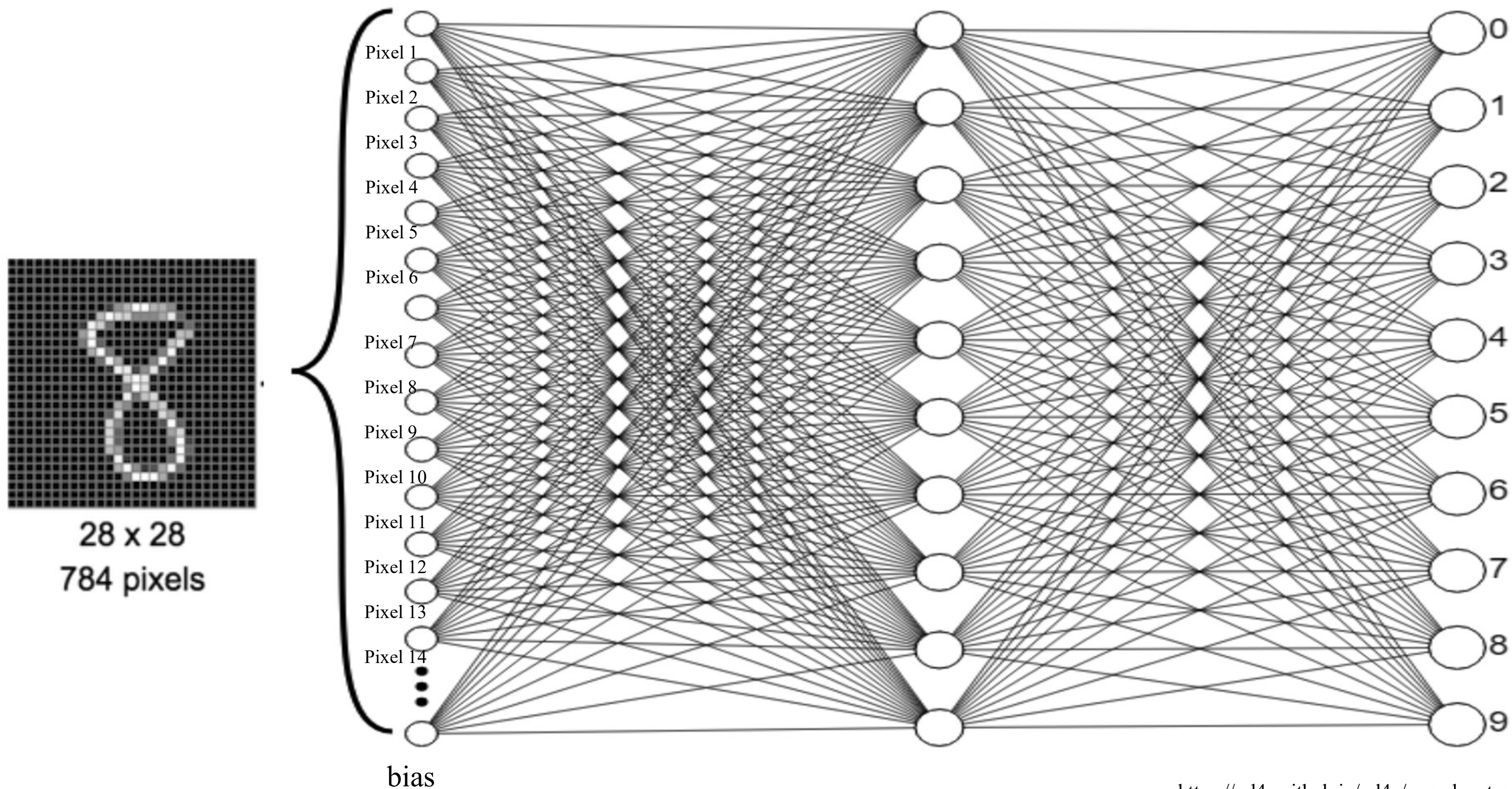
https://ml4a.github.io/ml4a/neural_networks/

How to classify an image?



https://ml4a.github.io/ml4a/neural_networks/

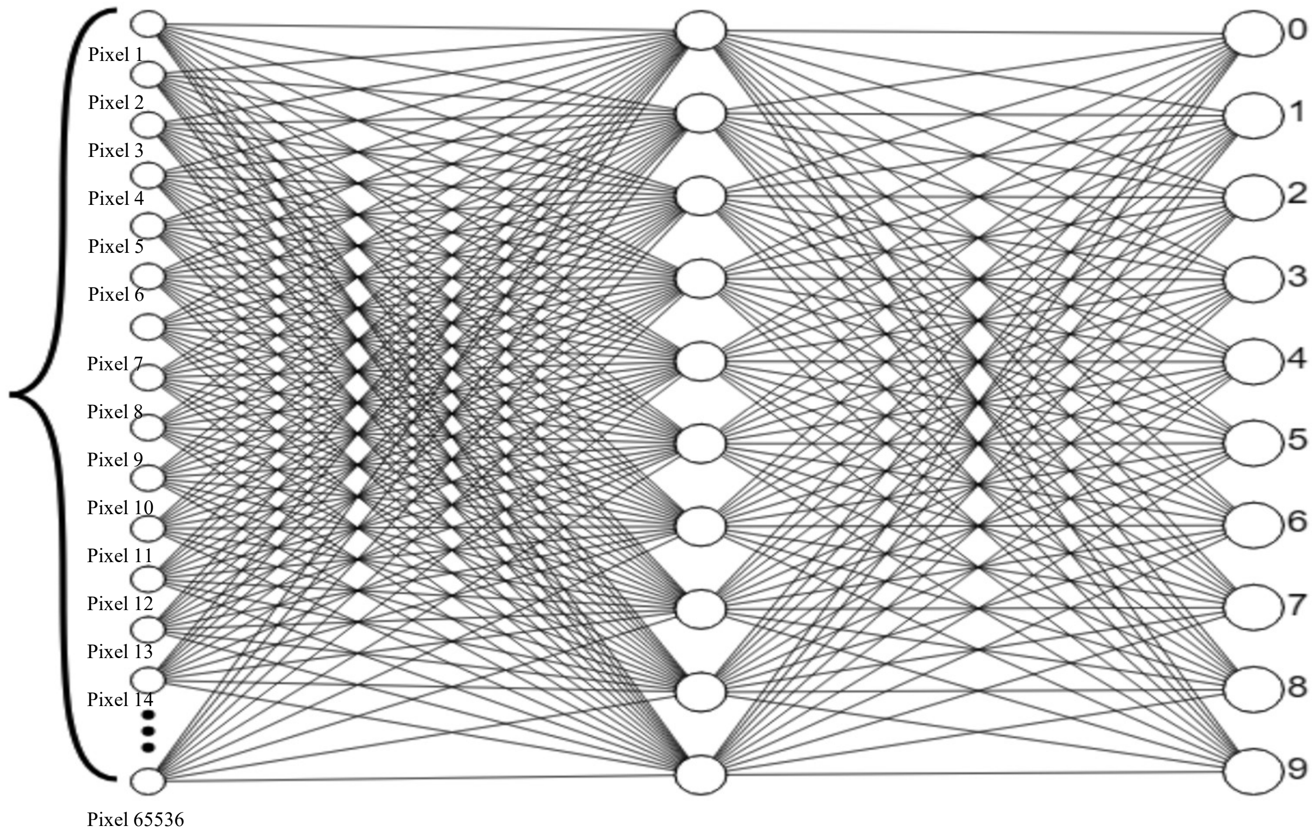
Many parameters (7850 in Layer 1)



https://ml4a.github.io/ml4a/neural_networks/

Too many parameters (655,370 in Layer 1)

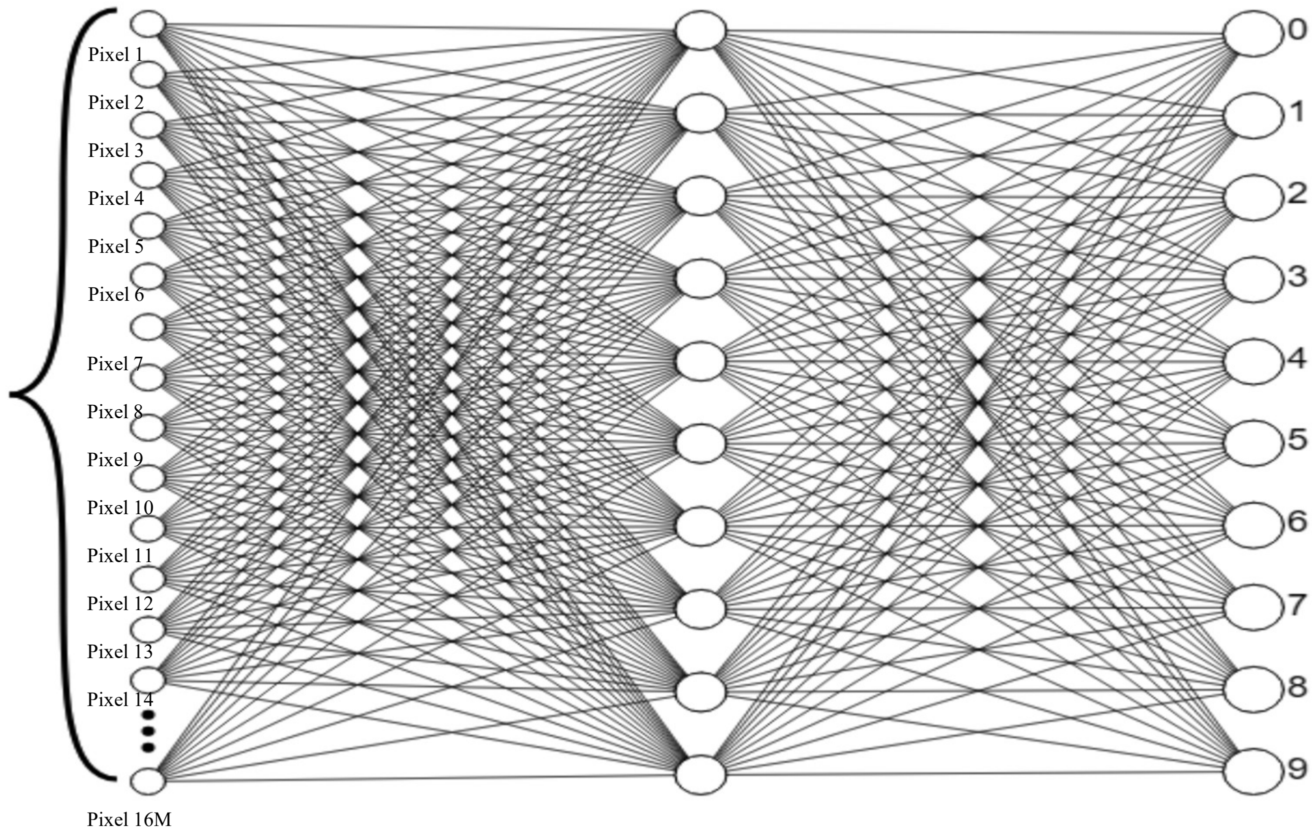
256x256



https://ml4a.github.io/ml4a/neural_networks/

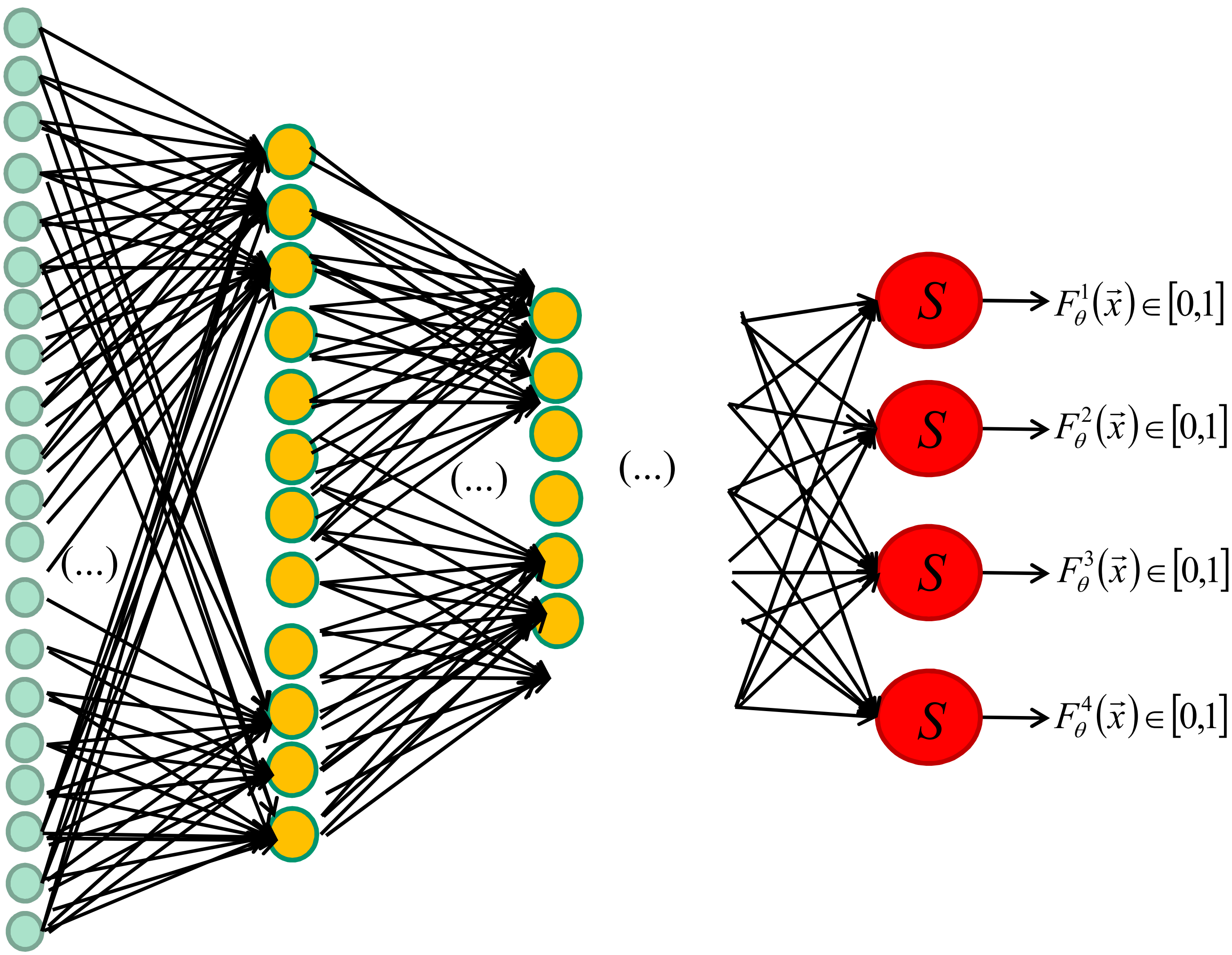
Waaay too many parameters (160M in Layer 1)

256x256x256



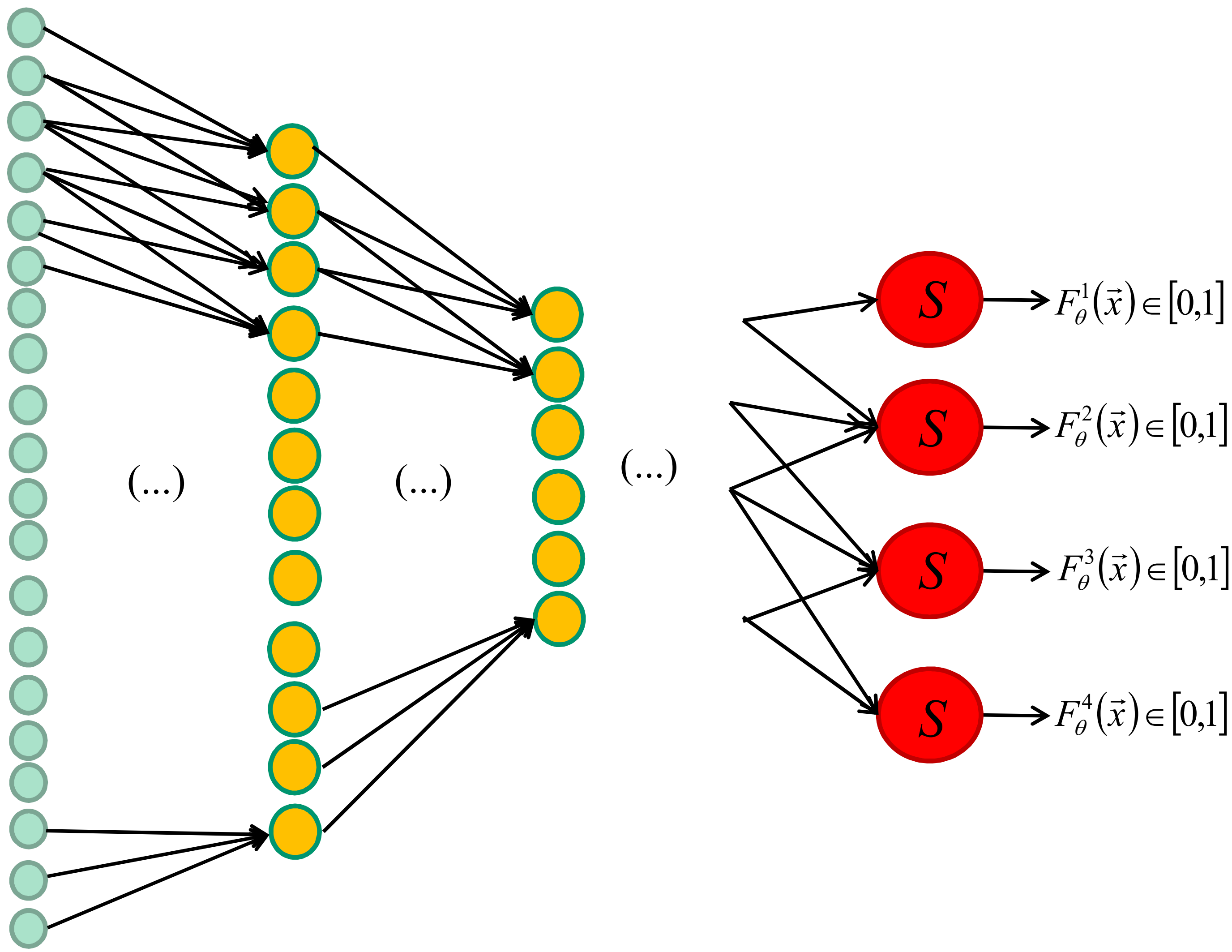
https://ml4a.github.io/ml4a/neural_networks/

Full connections are too many



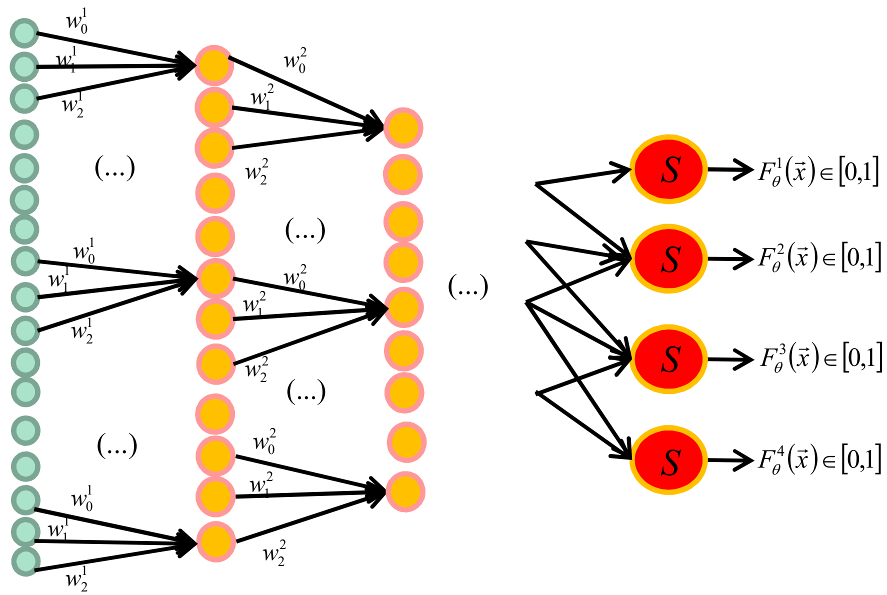
150-D input vector with 150 neurons in Layer 1 => **22,500 parameters!!**

No full connection



150-D input vector with 150 neurons in Layer 1 => **450 parameters!!**

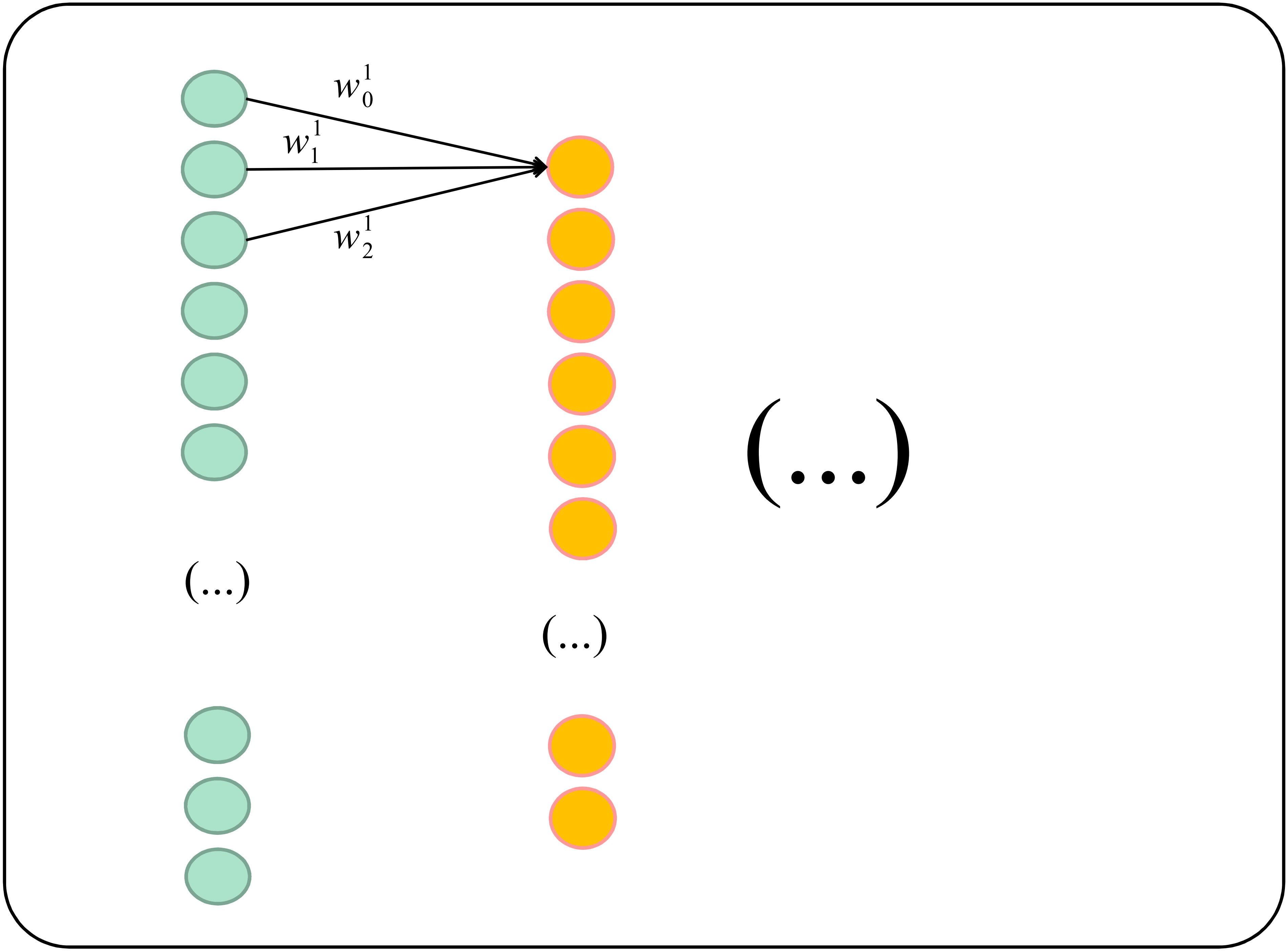
Share weights



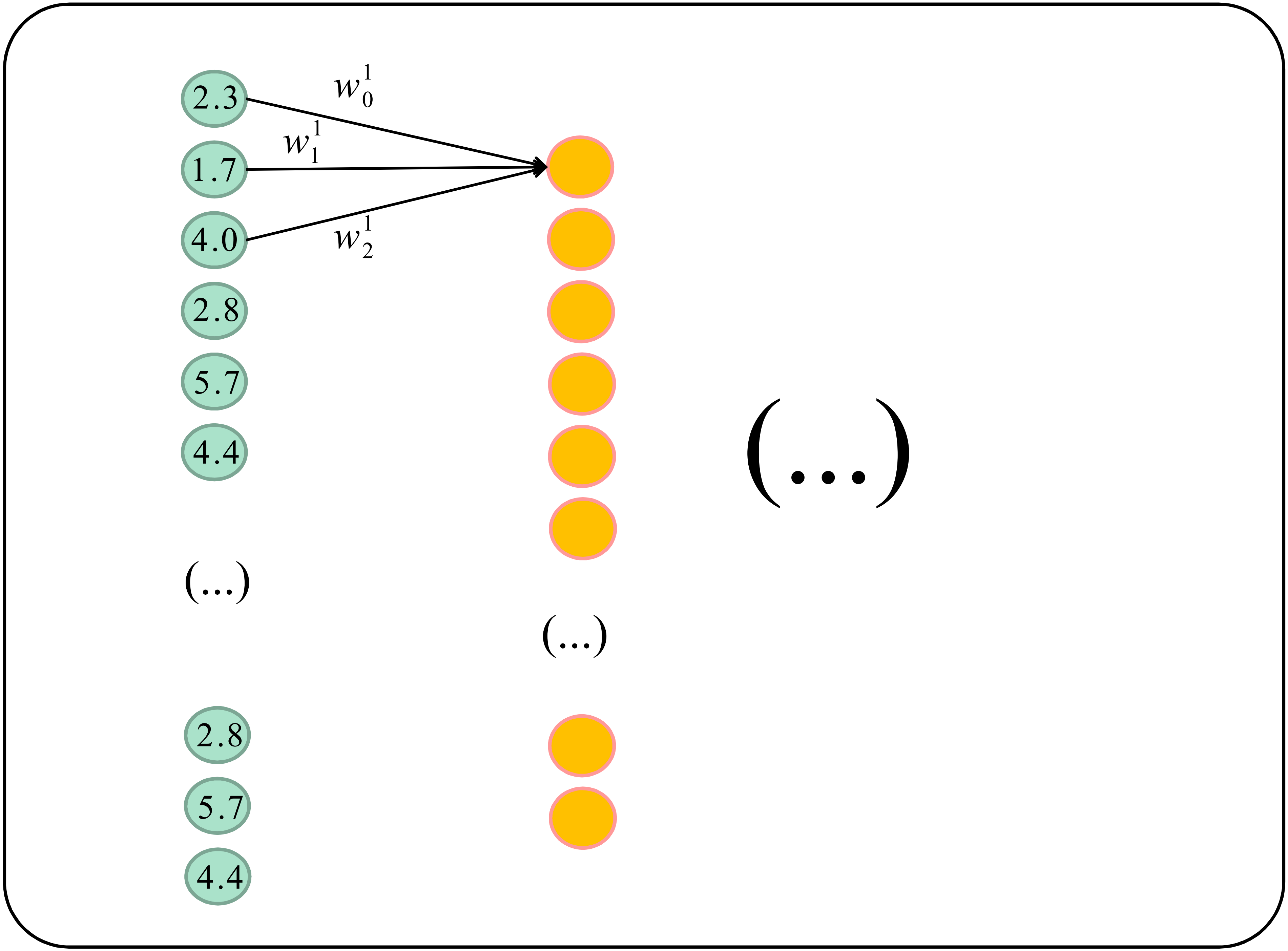
1- Learning convolution filters!

2- Small number of parameters = can make it deep!

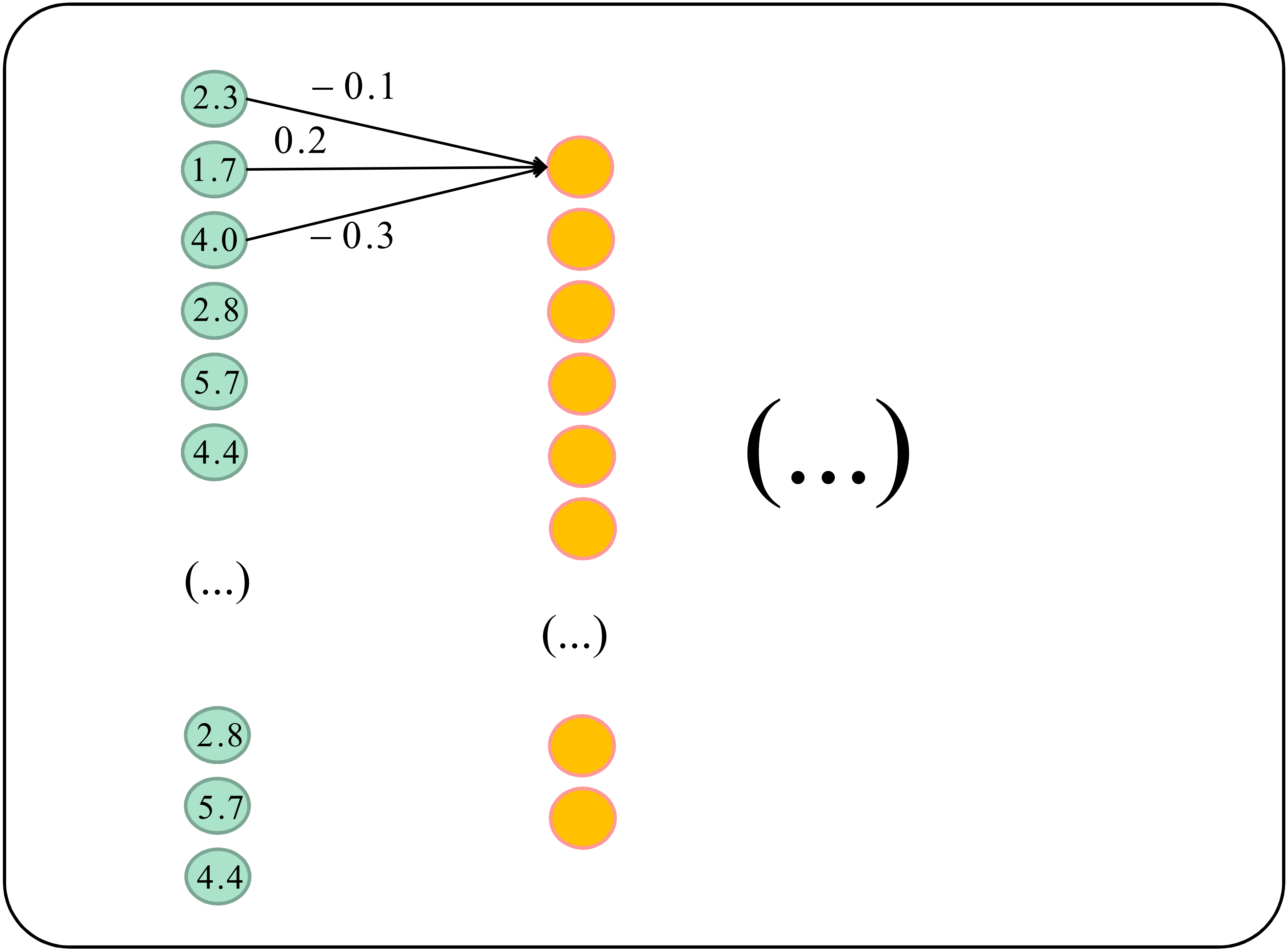
150-D input vector with 150 neurons in Layer 1 => **3 parameters!!**



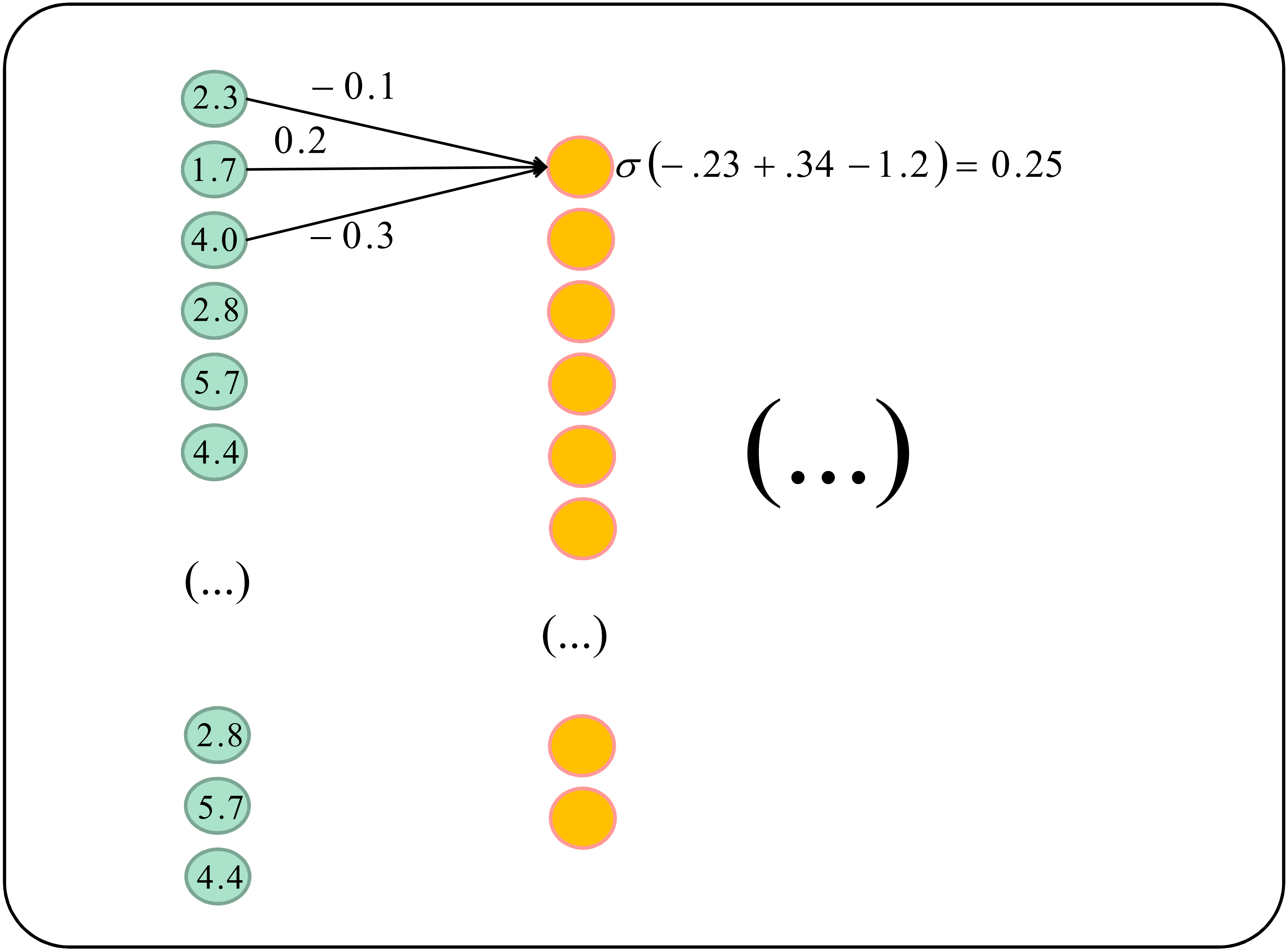
w_0^1
w_1^1
w_2^1
(...)
(...)
(...)



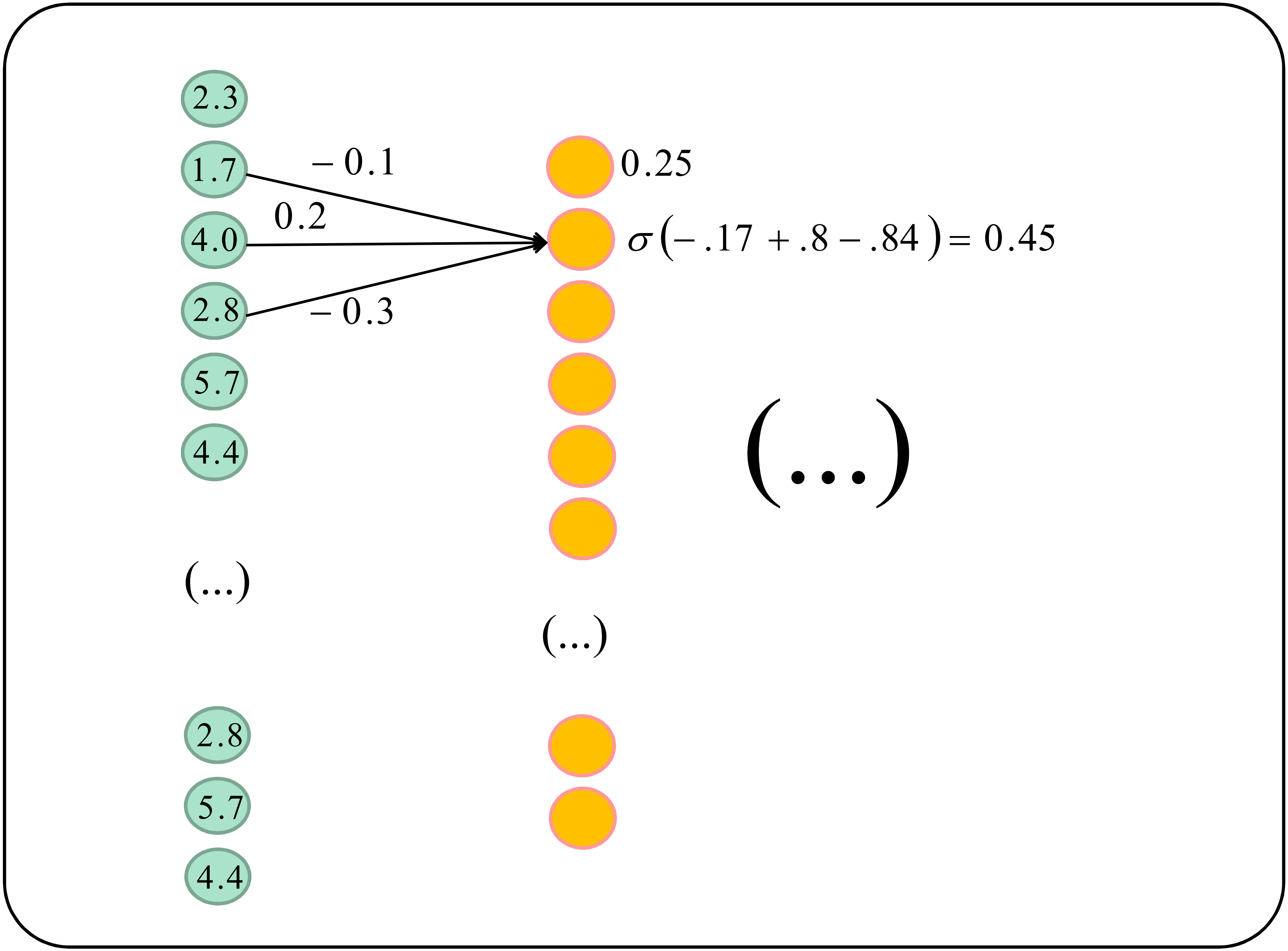
2.3
1.7
4.0
2.8
5.7
4.4
(...)
2.8
5.7
4.4
w_0^1
w_1^1
w_2^1
(...)
(...)

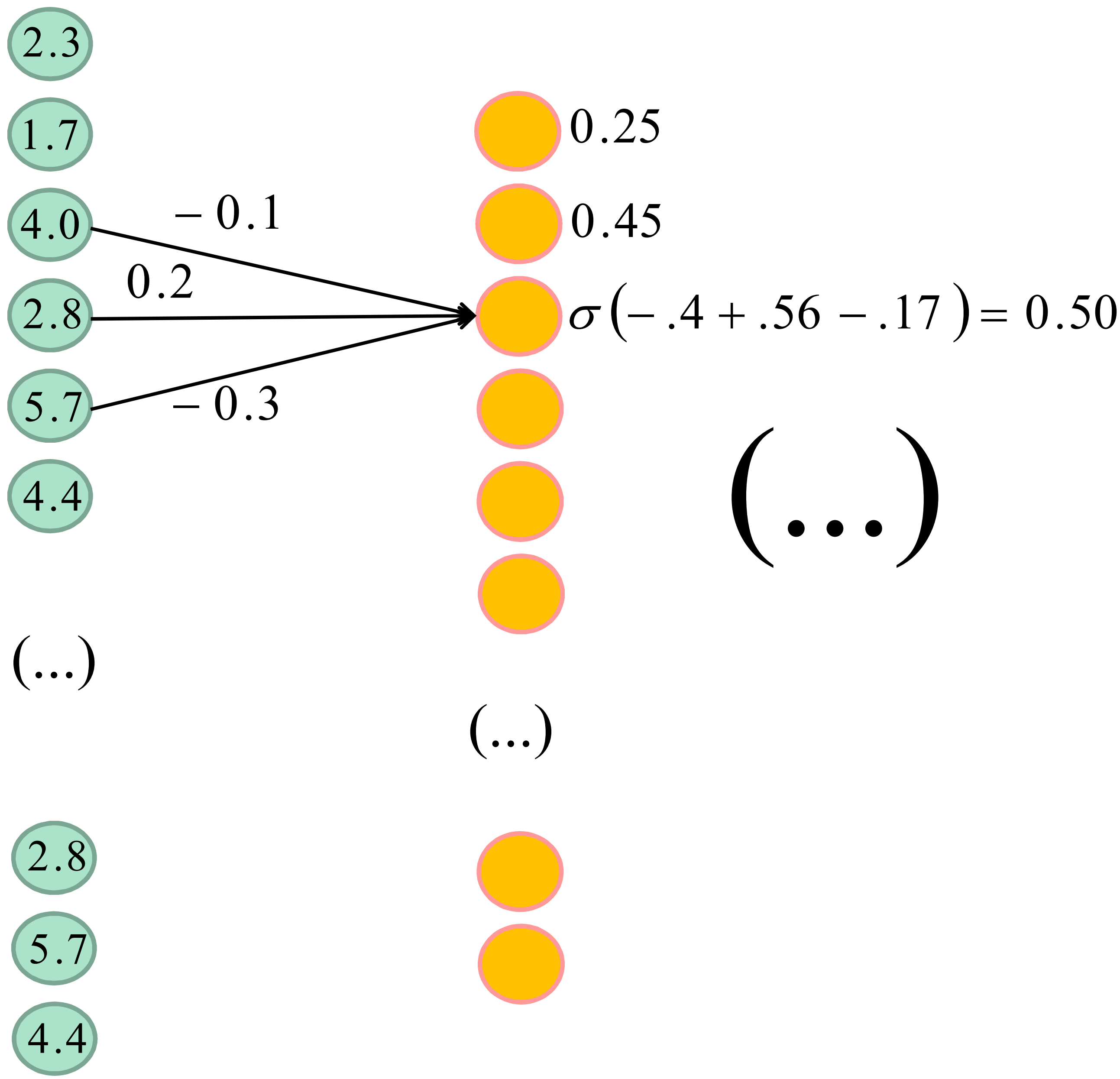


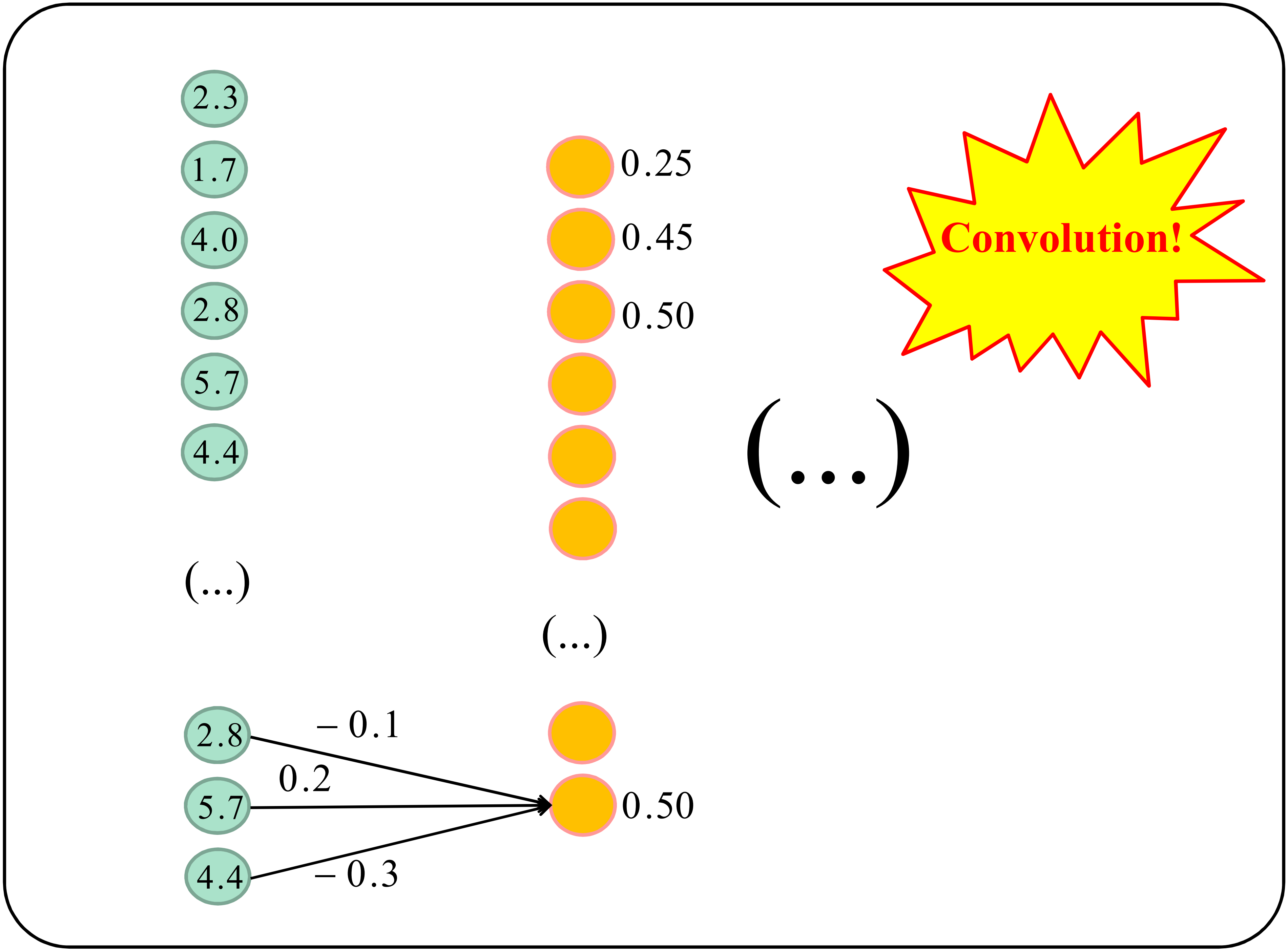
2.3
1.7
4.0
2.8
5.7
4.4
(...)
2.8
5.7
4.4
– 0.1
0.2
– 0.3
(...)
(...)



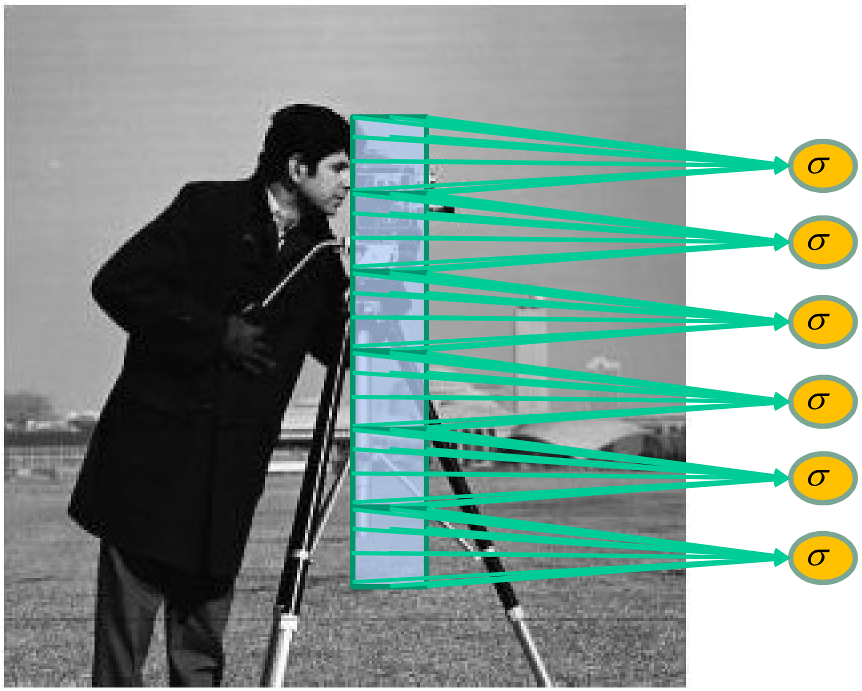
2.3
1.7
4.0
2.8
5.7
4.4
(...)
2.8
5.7
4.4
-0.1
0.2
-0.3
$\sigma(-.23 + .34 - 1.2) = 0.25$
(...)
(...)


2.3
1.7
4.0
2.8
5.7
4.4
(...)
2.8
5.7
4.4
− 0.1
0.2
− 0.3
0.25
$\sigma(-.17 + .8 - .84) = 0.45$
(...)
(...)

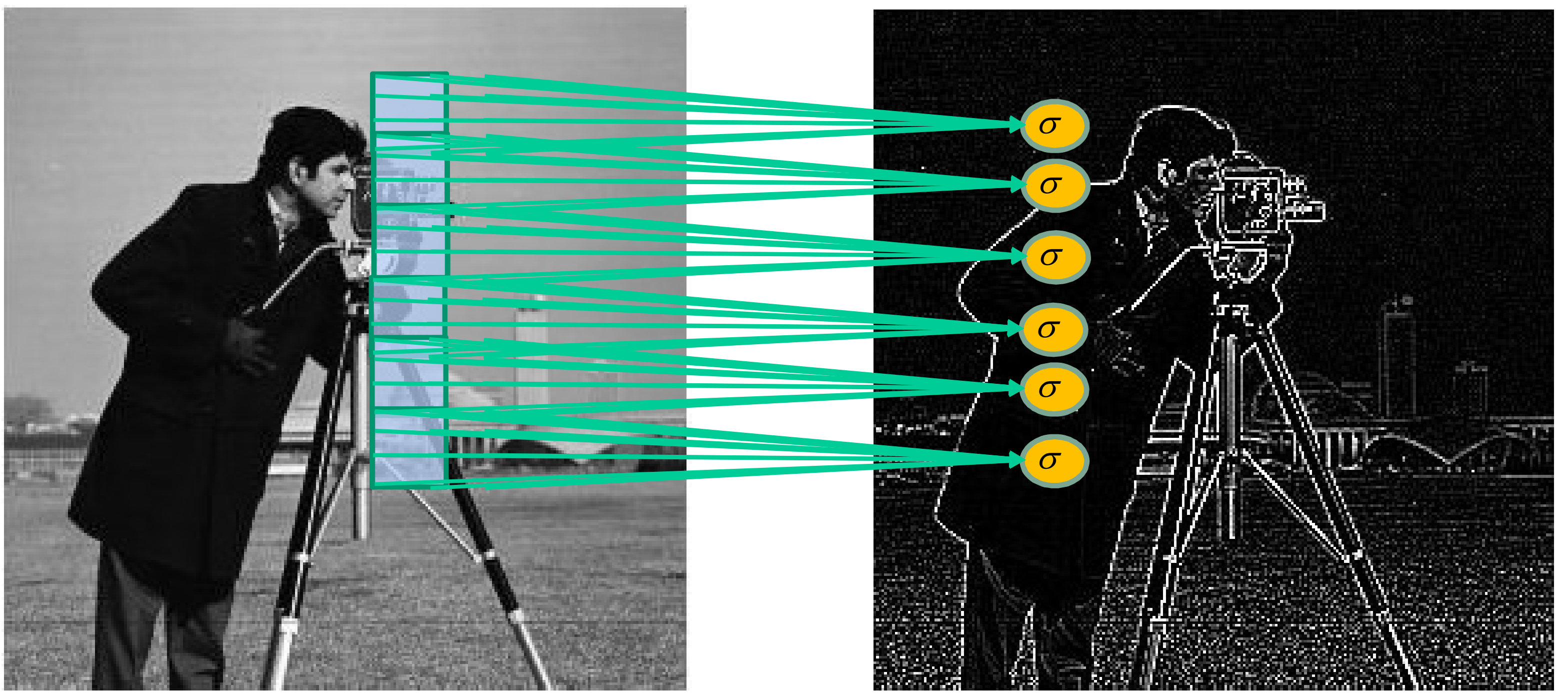

2.3
1.7
4.0
2.8
5.7
4.4
(...)
2.8
5.7
4.4
− 0.1
0.2
− 0.3
0.25
0.45
σ(− .4 + .56 − .17) = 0.50
(...)
(...)



2.3
1.7
4.0
2.8
5.7
4.4
(...)
0.25
0.45
0.50
(...)
Convolution!
(...)
2.8
5.7
4.4
− 0.1
0.2
− 0.3
0.50

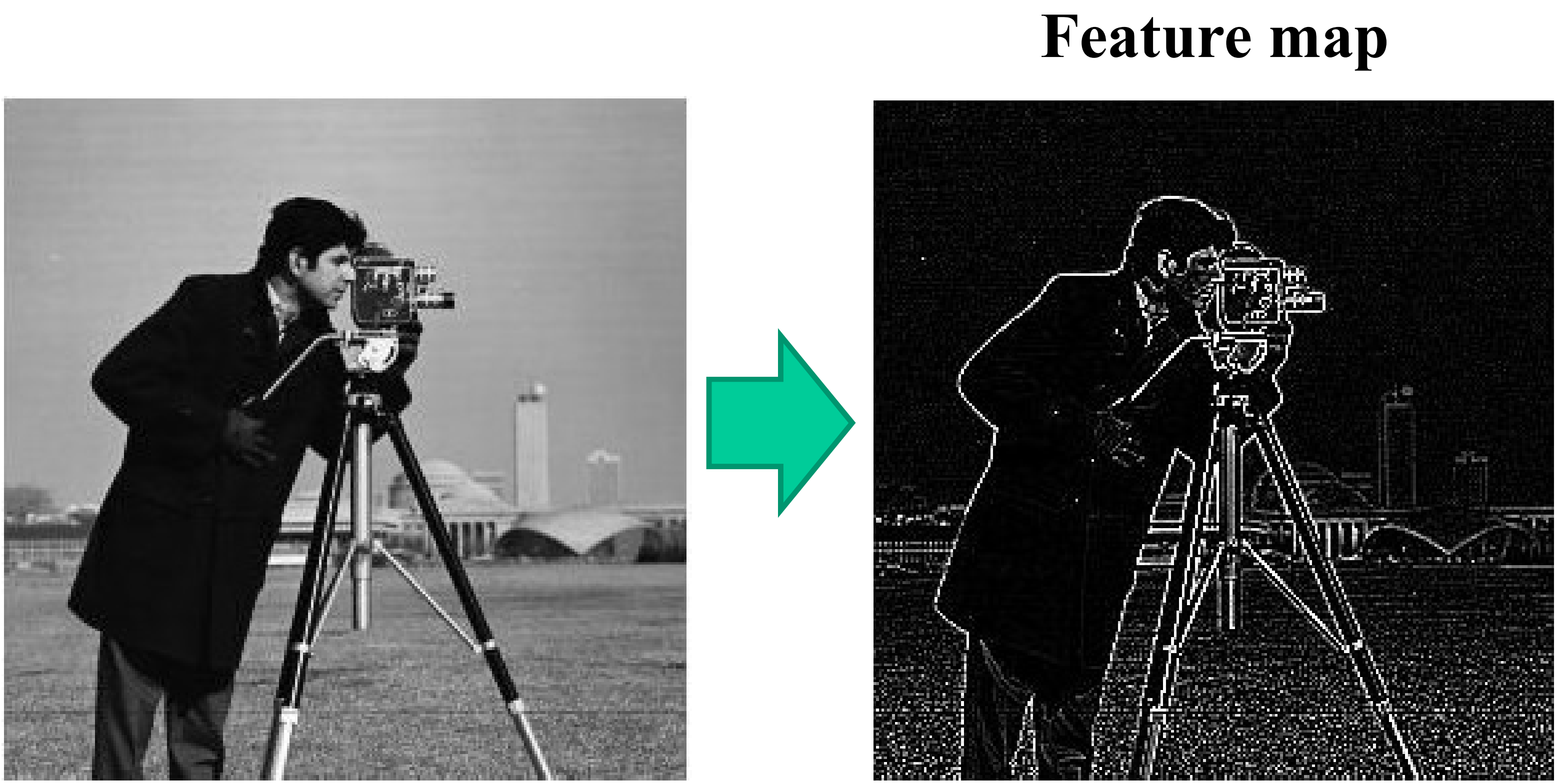


Each neuron of layer 1 is connected to 3x3 pixels, layer 1 has **9 parameters!!**

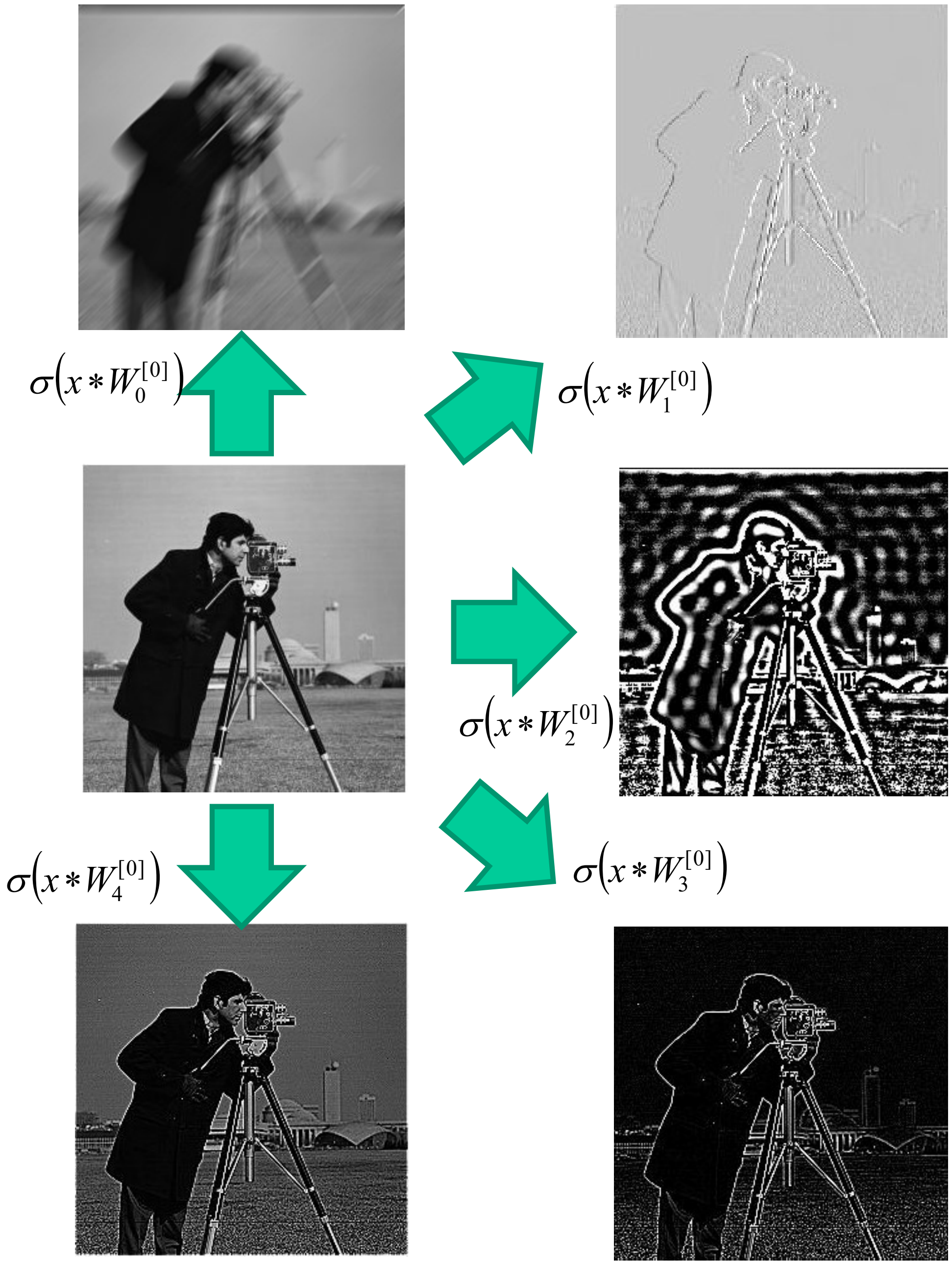


σ
σ
σ
σ
σ
σ

Convolution operation

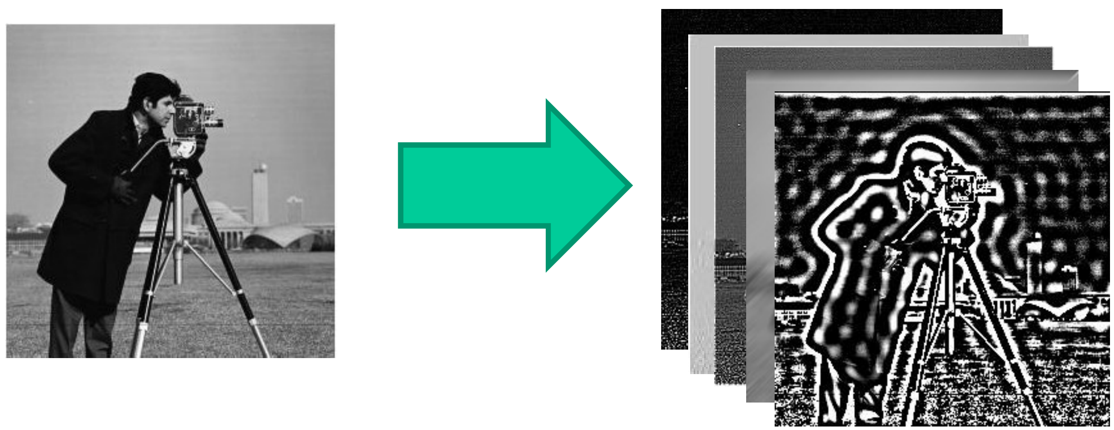


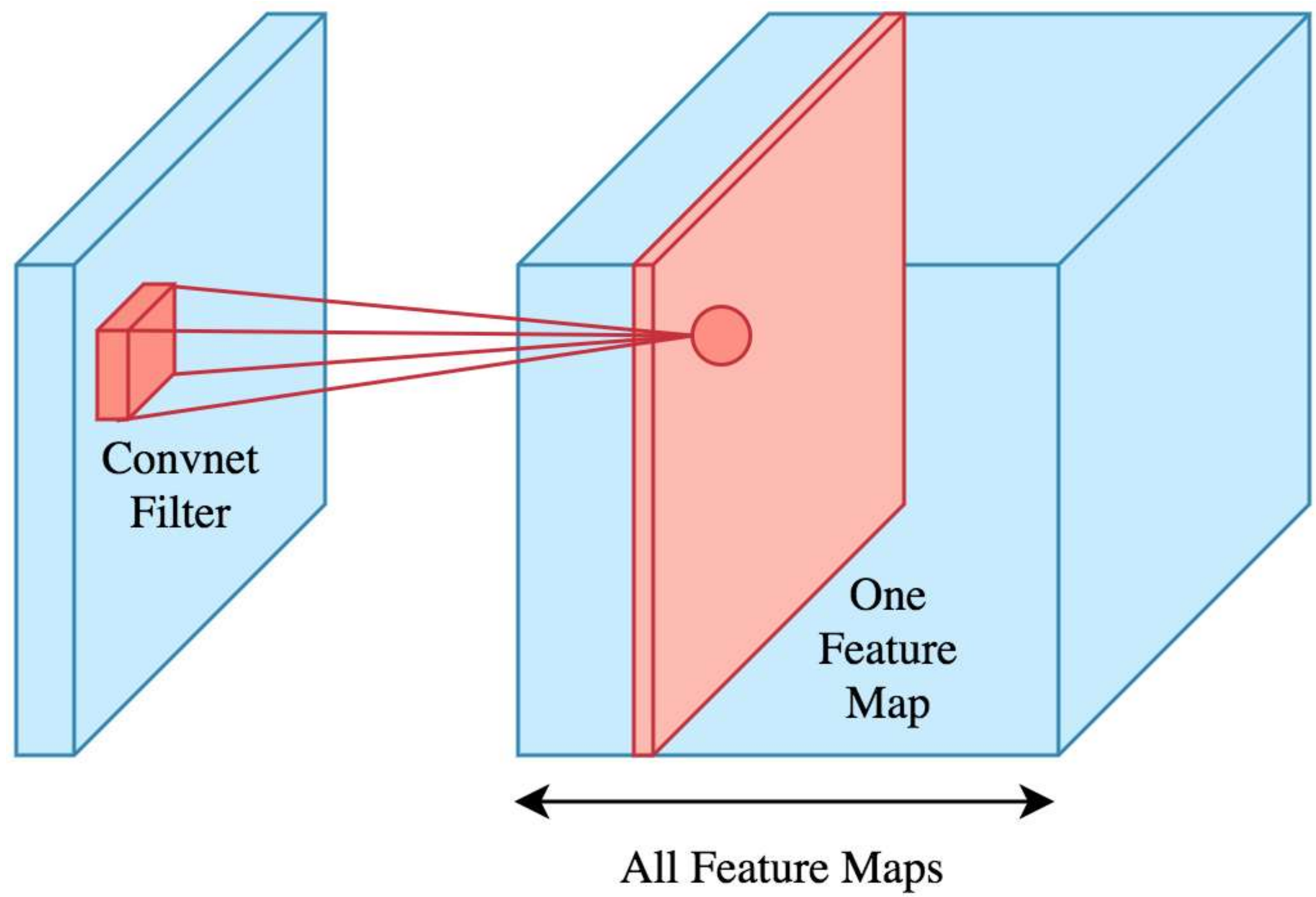
$$F = \sigma\left(x * W^{[0]}\right)$$



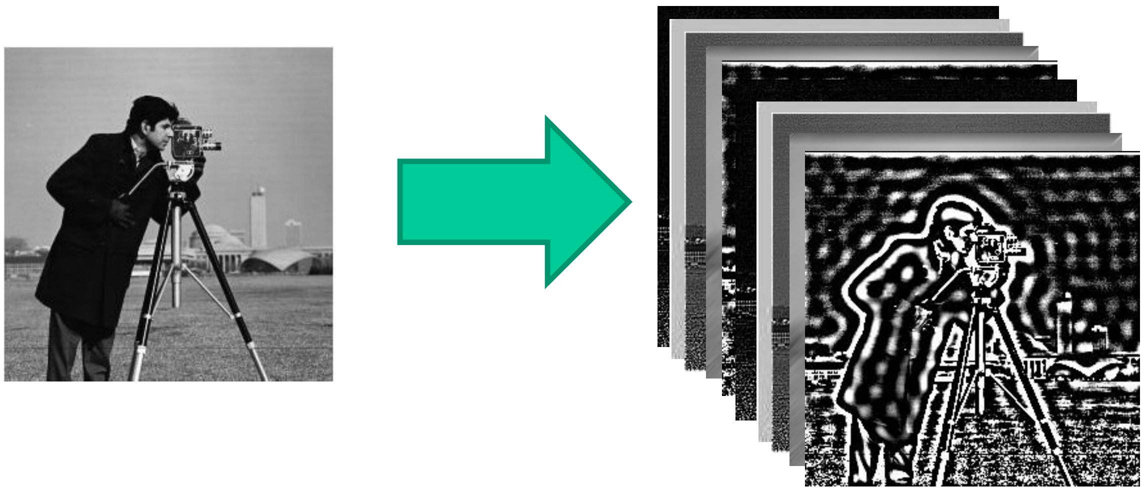
$\sigma\left(x * W_0^{[0]}\right)$
$\sigma\left(x * W_1^{[0]}\right)$
$\sigma\left(x * W_2^{[0]}\right)$
$\sigma\left(x * W_4^{[0]}\right)$
$\sigma\left(x * W_3^{[0]}\right)$

5-feature map convolution layer

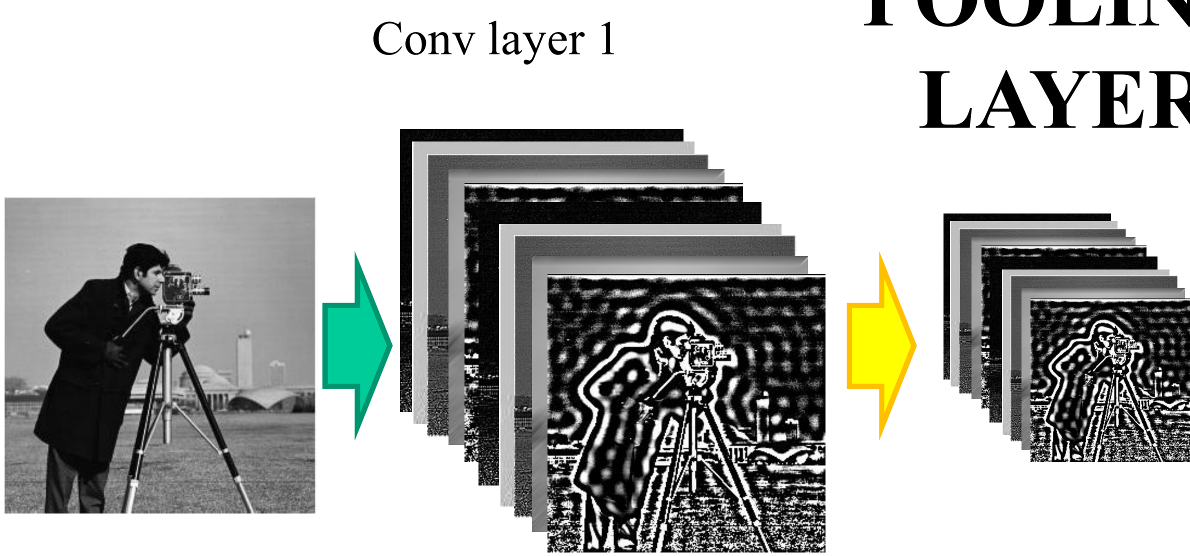



Convnet
Filter
One
Feature
Map
All Feature Maps

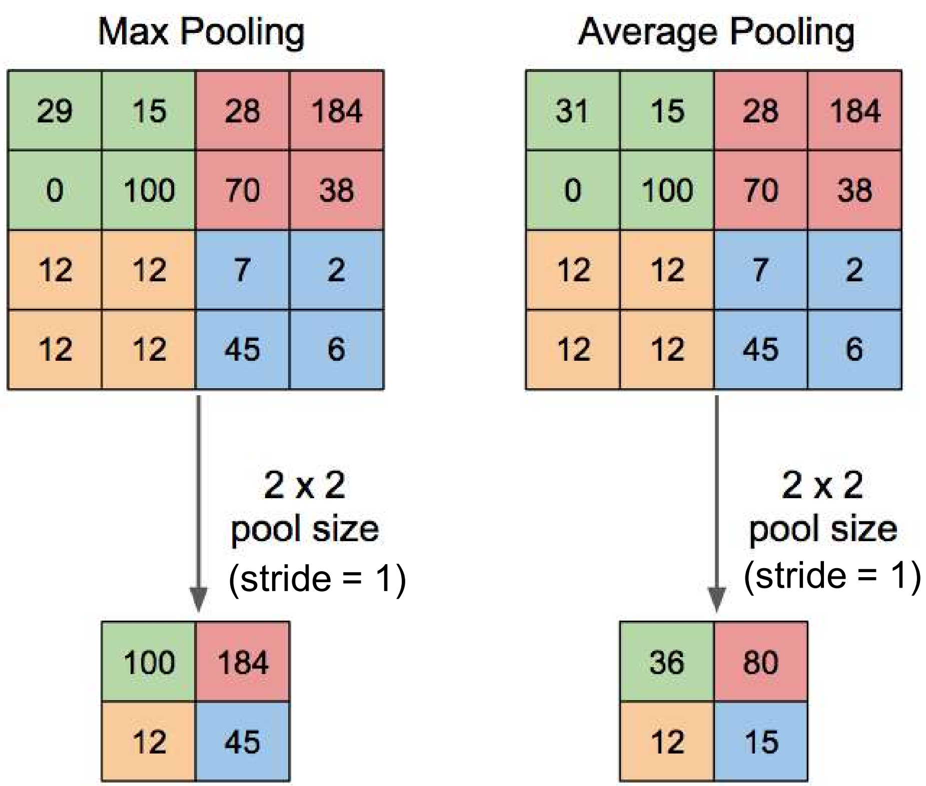
K-feature map convolution layer



POOLING LAYER



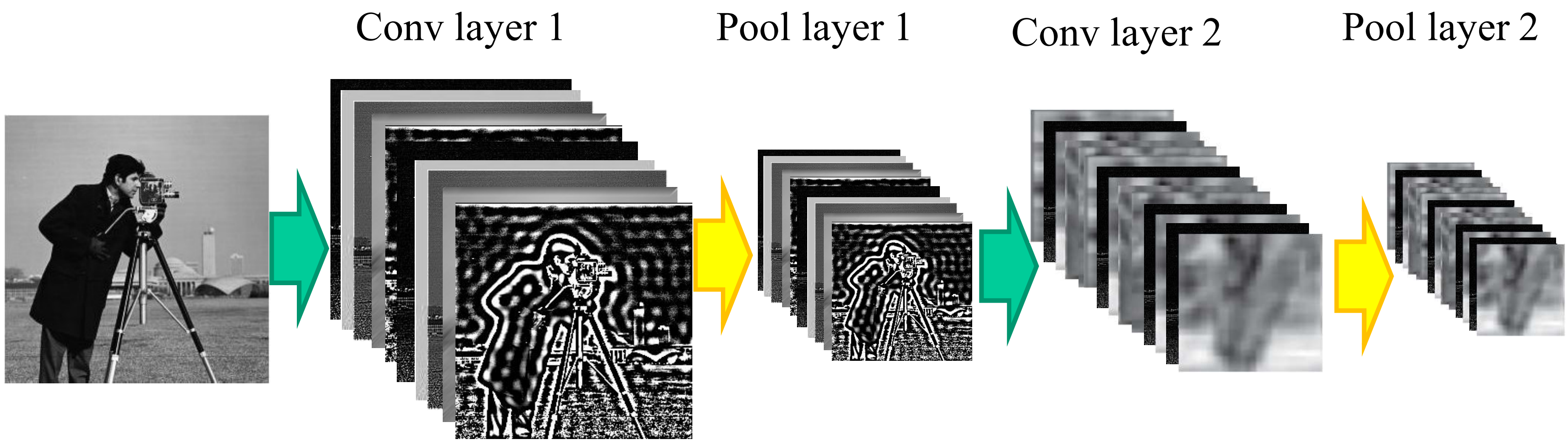
Pooling layer



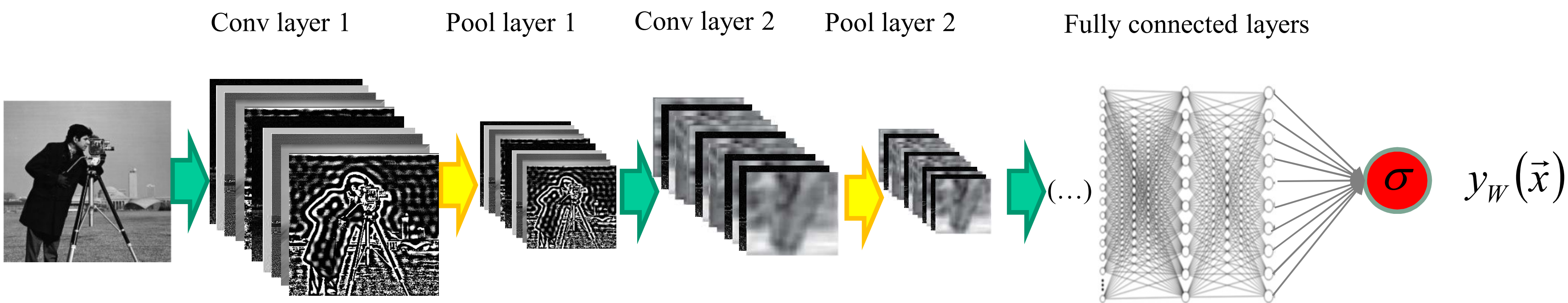
Goals

- Reduce the spatial resolution of feature maps
- Lower memory and computation requirements
- Provide partial invariance to position, scale and rotation

Images: https://pythonmachinelearning.pro/introduction-to-convolutional-neural-networks-for-vision-tasks/

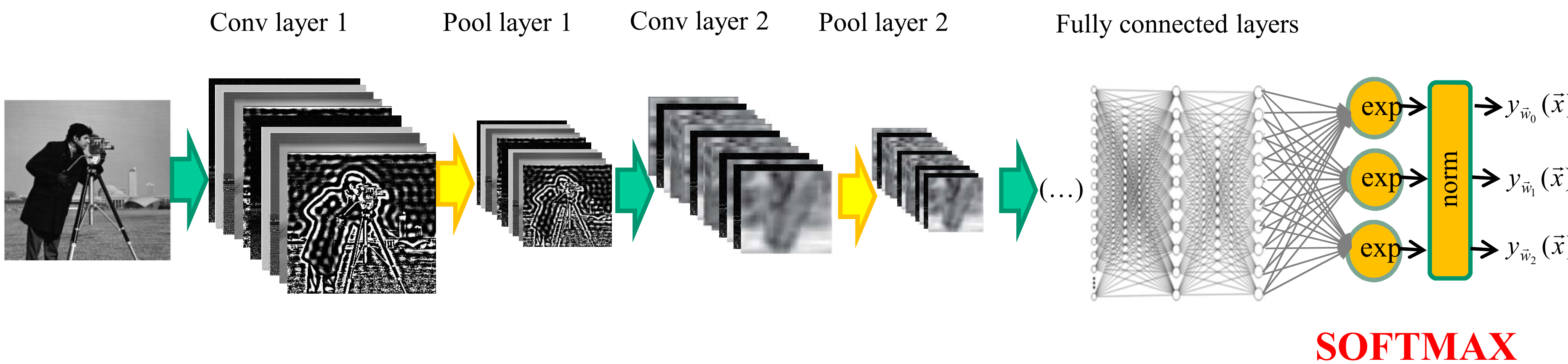

Conv layer 1
Pool layer 1
Conv layer 2
Pool layer 2

2 Class CNN



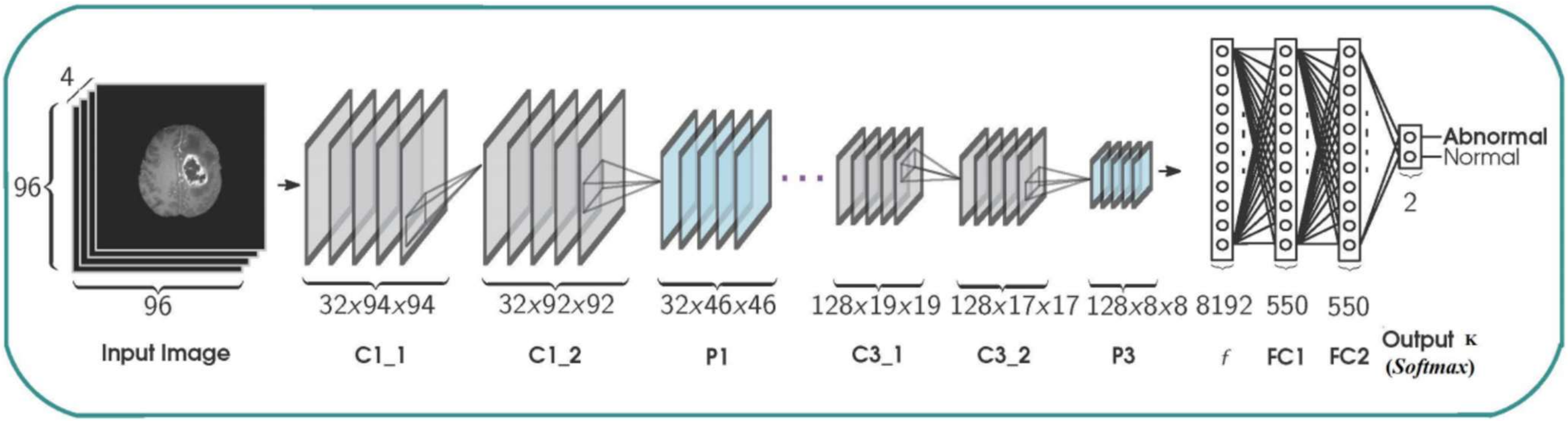
$$l\left(y_W(\vec{x}), t\right) = -t \ln\left(y_W(\vec{x})\right) - (1-t)\ln\left(1 - y_W(\vec{x})\right)$$

K Class CNN



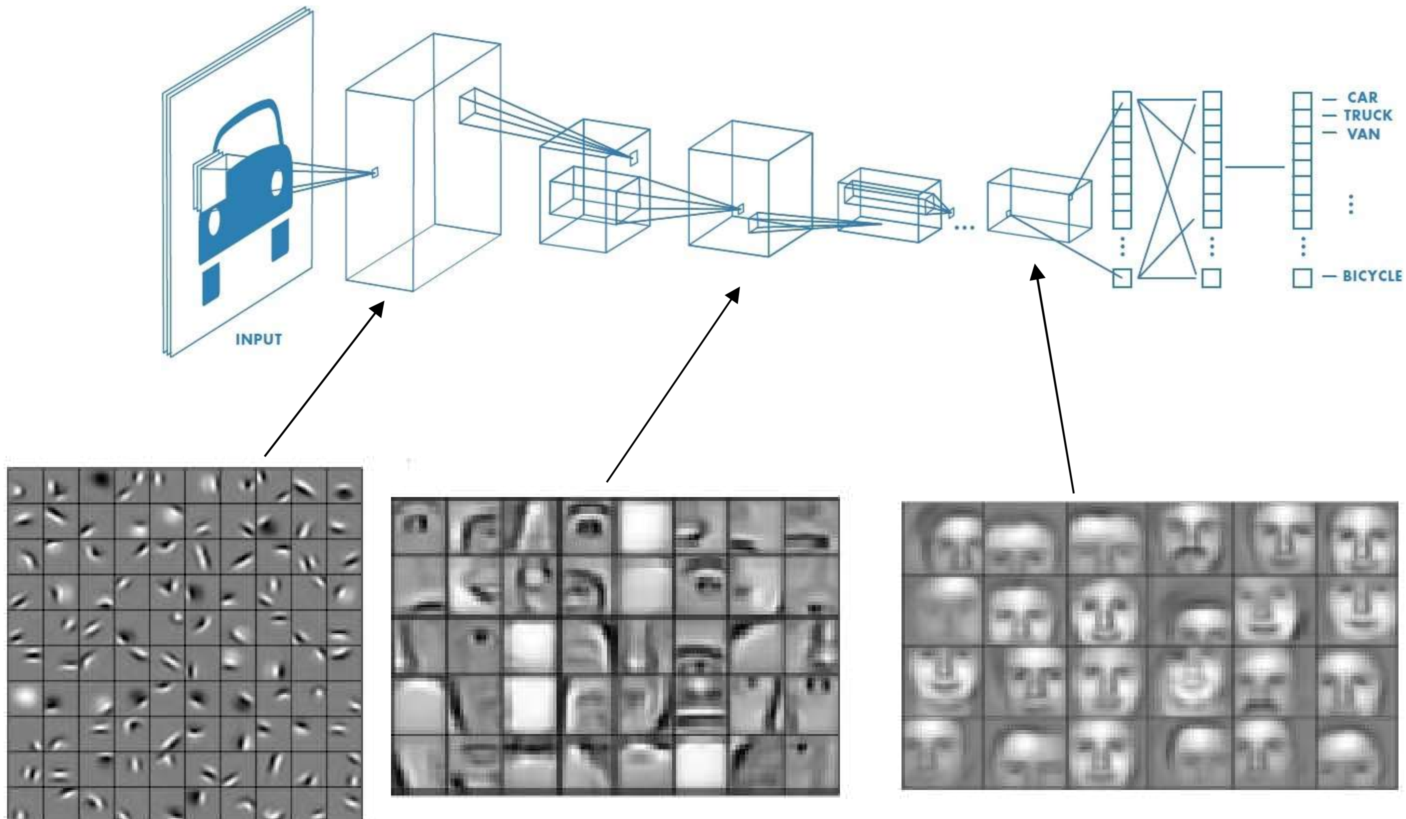
$$l\left(y_W(\vec{x}), t\right) = -t \ln\left(y_W(\vec{x})\right) - (1-t)\ln\left(1 - y_W(\vec{x})\right)$$

Nice example from the litterature



S. Banerjee, S. Mitra, A. Sharma, and B. U Shankar, A CADe System for Gliomas in Brain MRI using Convolutional Neural Networks, arXiv:1806.07589v, 2018

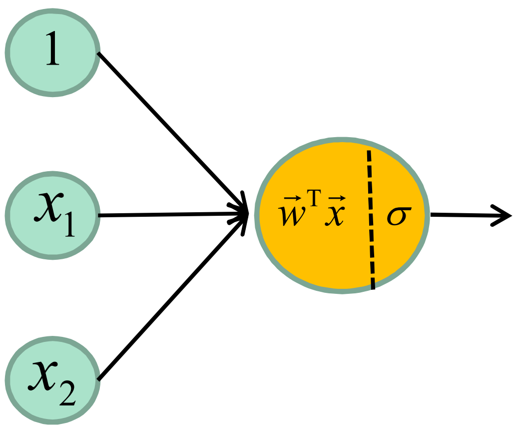
Learn image-based caracteristics



http://web.eecs.umich.edu/~honglak/icml09-ConvolutionalDeepBeliefNetworks.pdf

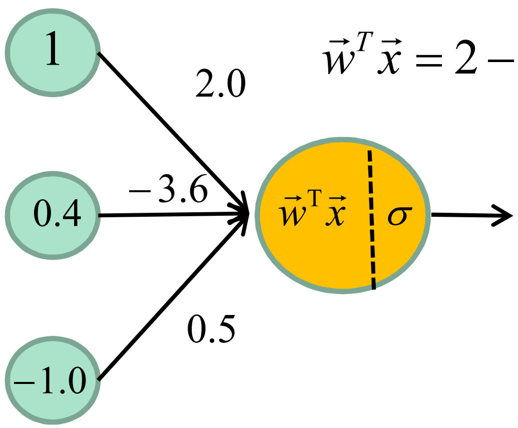
Batch processing

$$\vec{x} = (0.4, -1.0)$$
$$\vec{w} = [2.0, -3.6, 0.5]$$



$$\vec{x} = (0.4, -1.0)$$

$$\vec{w} = [2.0, -3.6, 0.5]$$

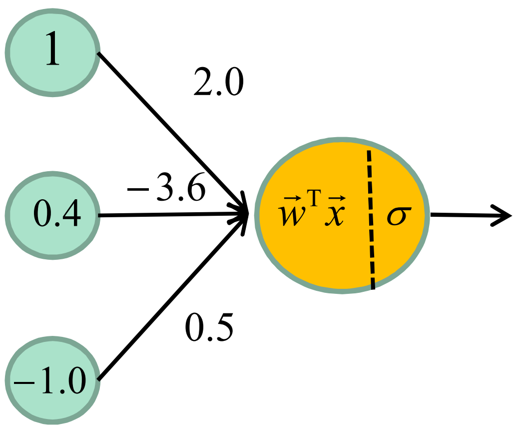


$$\vec{w}^T\vec{x} = 2 - 3.6 * 0.4 - 0.5 = -1.94$$

$$y_{\vec{w}}(\vec{x}) = \sigma(-1.94) = \frac{1}{1+e^{1.94}} = 0.125$$

$$\vec{x} = (0.4, -1.0)$$

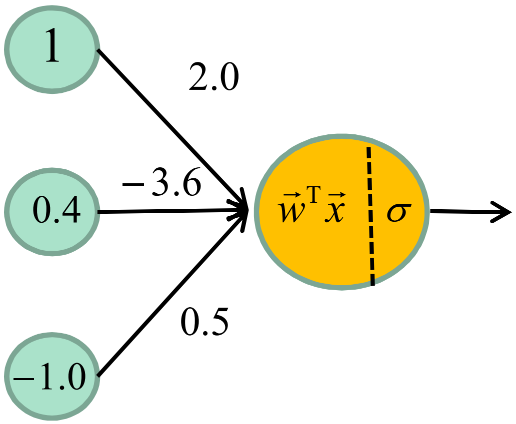
$$\vec{w} = [2.0, -3.6, 0.5]$$



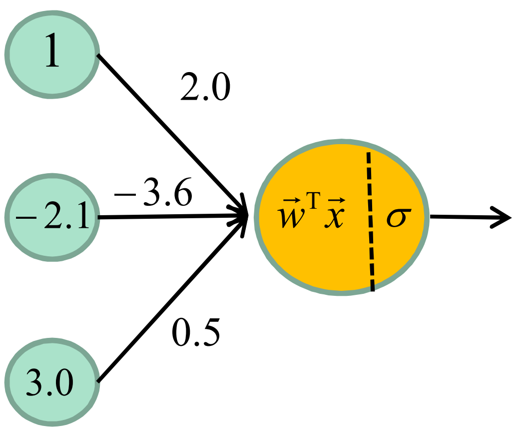
$$y_{\vec{w}}(\vec{x}_a) = \sigma\left((2.0, -3.6, 0.5) \begin{pmatrix} 1 \\ 0.4 \\ -1 \end{pmatrix} \right) = 0.125$$

$$\vec{x}_a = (0.4, -1.0), \vec{x}_b = (-2.1, 3.0)$$

$$\vec{w} = [2.0, -3.6, 0.5]$$

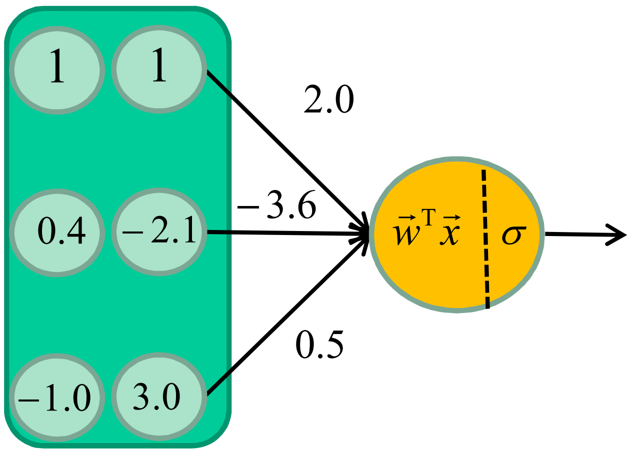


$$y_{\vec{w}}(\vec{x}_a) = \sigma\left((2.0, -3.6, 0.5)\begin{pmatrix}1\\0.4\\-1\end{pmatrix}\right) = 0.125$$



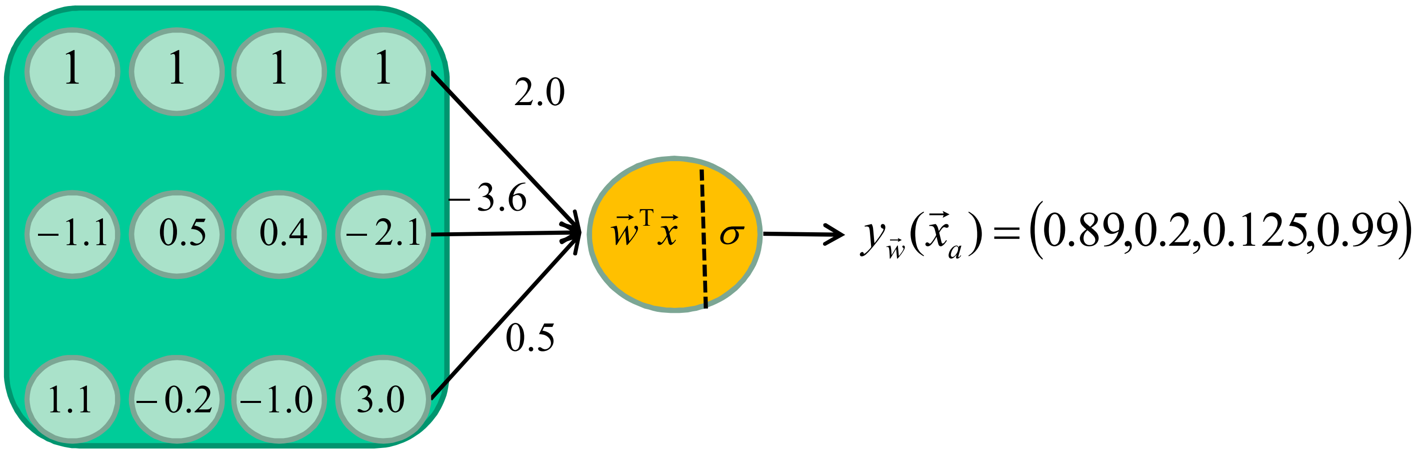
$$y_{\vec{w}}(\vec{x}_b) = \sigma\left((2.0, -3.6, 0.5)\begin{pmatrix}1\\-2.1\\3.0\end{pmatrix}\right) = 0.99$$

Mini-batch processing

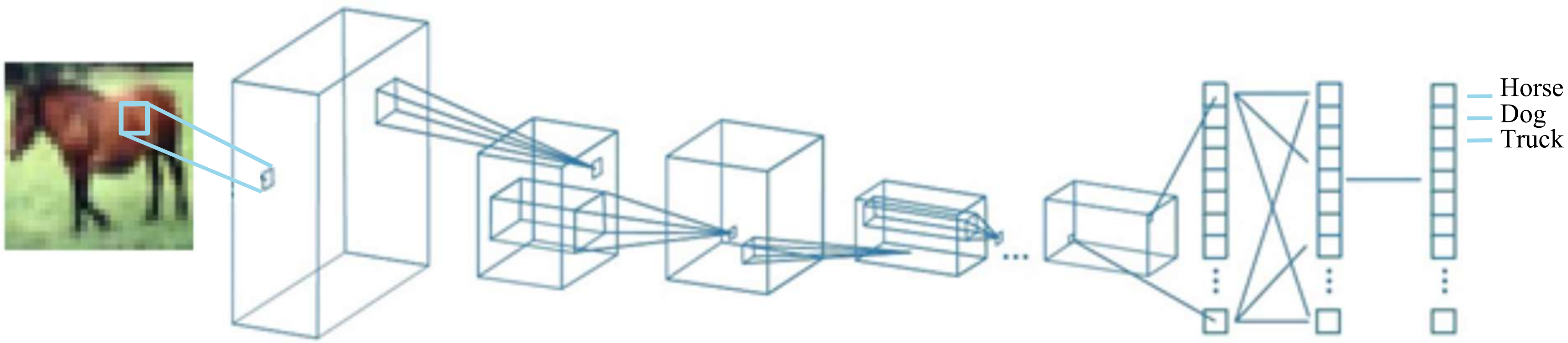


$$y_{\vec{w}}(\vec{x}_a) = \sigma\left((2.0, -3.6, 0.5) \begin{pmatrix} 1 & 1 \\ 0.4 & -2.1 \\ -1 & 3.0 \end{pmatrix} \right) = (0.125, 0.99)$$

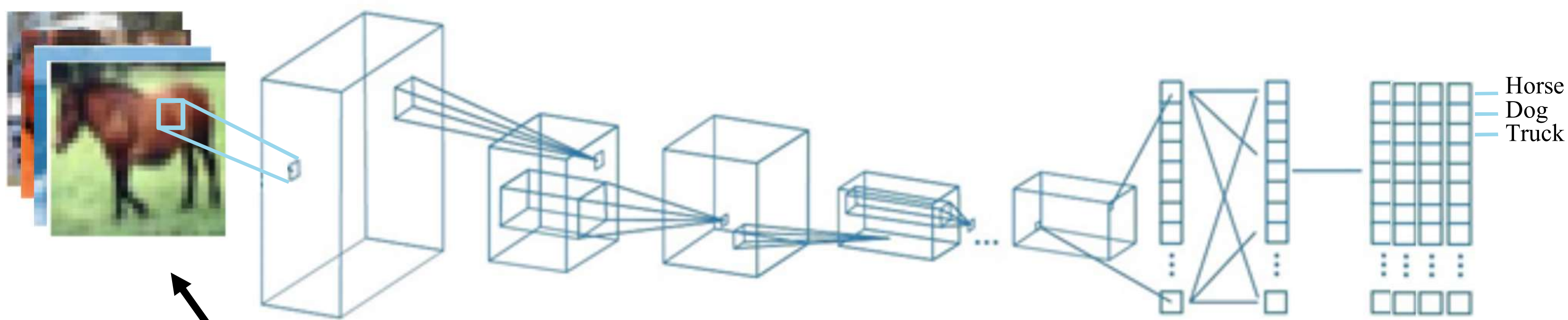
Mini-batch processing



Mini-batch processing



Mini-batch processing

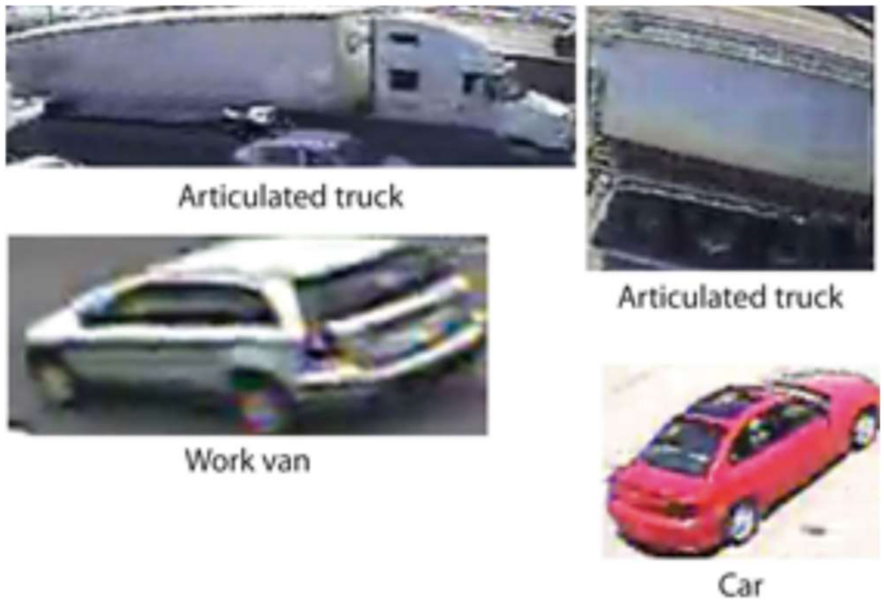


Mini-batch of
4 images

4 predictions

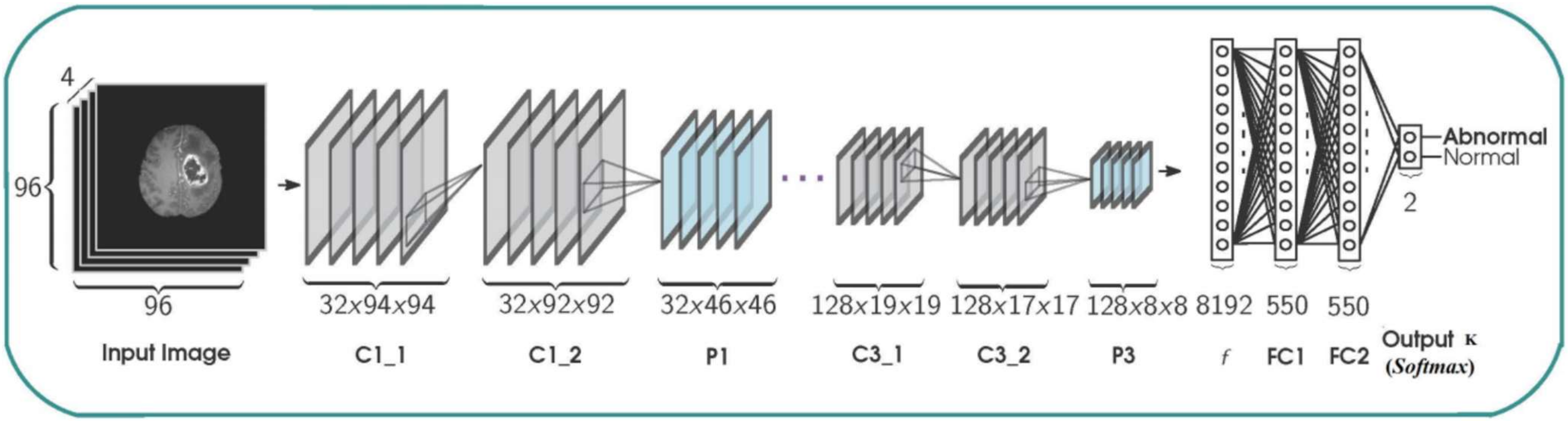
Classical applications of ConvNets

Classification.



Classical applications of ConvNets

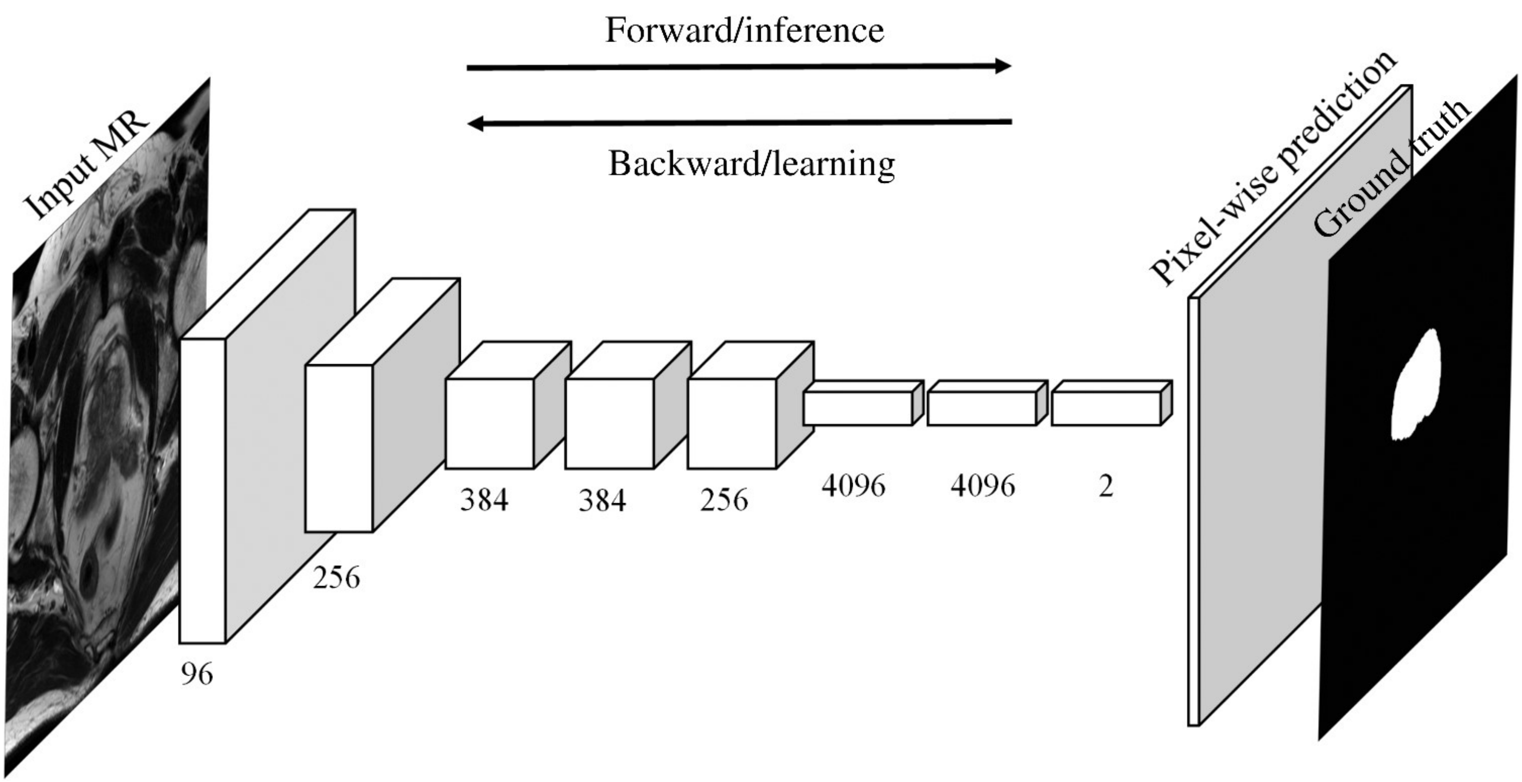
Classification.



S. Banerjee, S. Mitra, A. Sharma, and B. U Shankar, A CADe System for Gliomas in Brain MRI using Convolutional Neural Networks, arXiv:1806.07589v, 2018

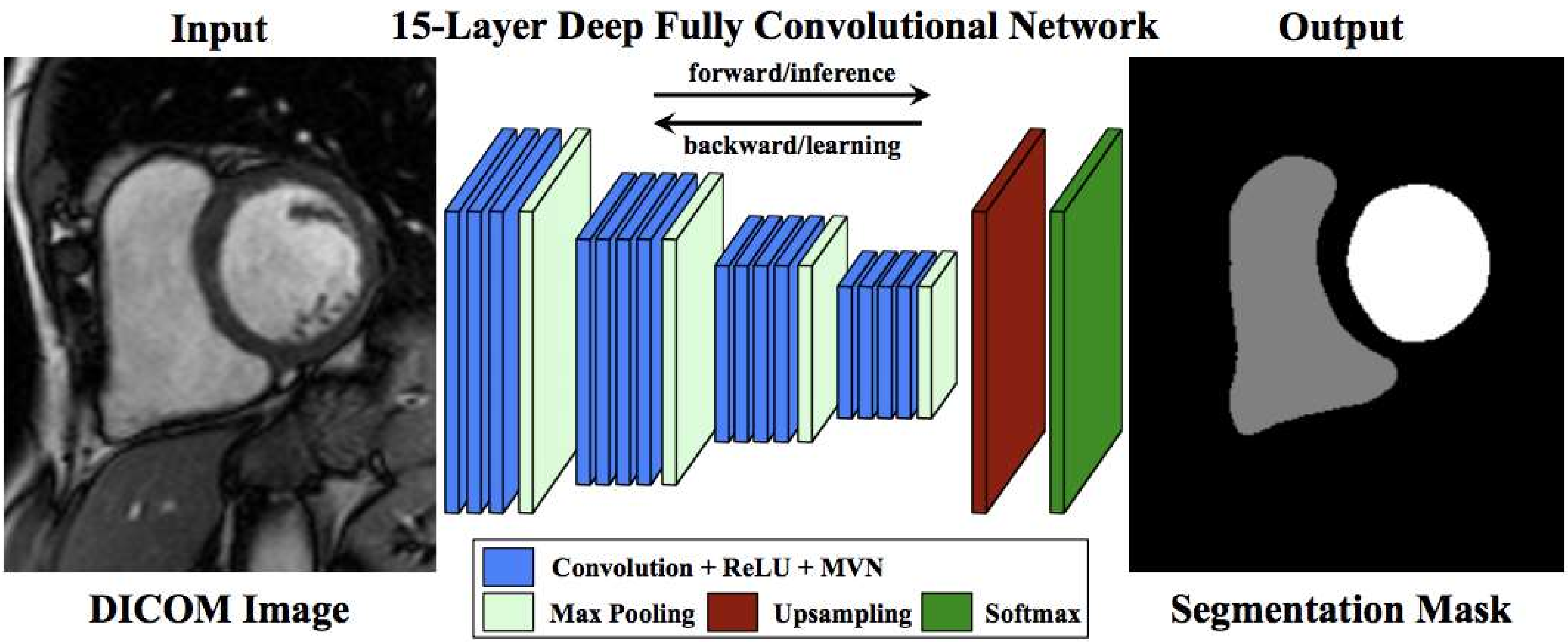
Classical applications of ConvNets

Image segmentation



Classical applications of ConvNets

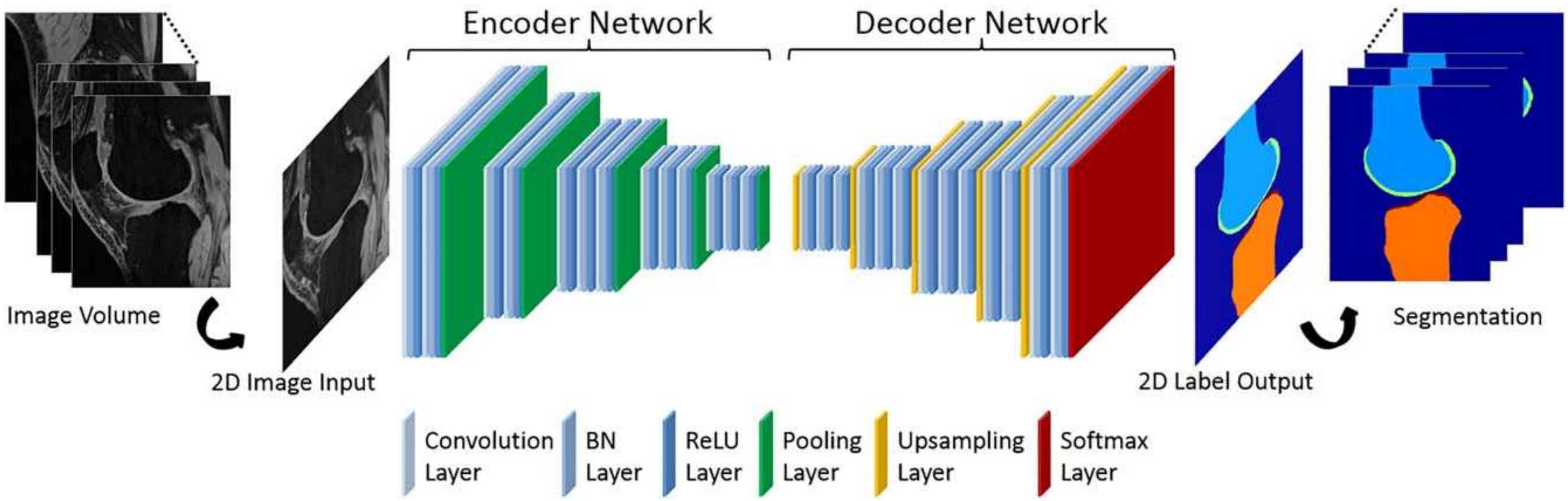
Image segmentation



Tran, P. V., 2016. A fully convolutional neural network for cardiac segmentation in short-axis MRI. arXiv:1604.00494.

Classical applications of ConvNets

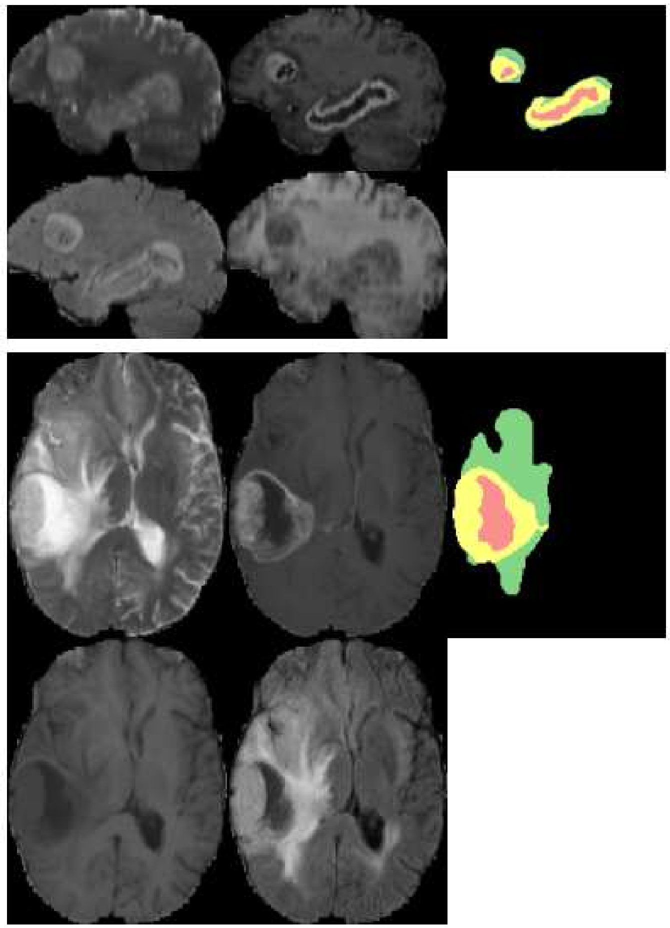
Image segmentation



Fang Liu, Zhaoye Zhou, +3 authors, Deep convolutional neural network and 3D deformable approach for tissue segmentation in musculoskeletal magnetic resonance imaging. in Magnetic resonance in medicine 2018 DOI:10.1002/mrm.26841

Classical applications of ConvNets

Image segmentation



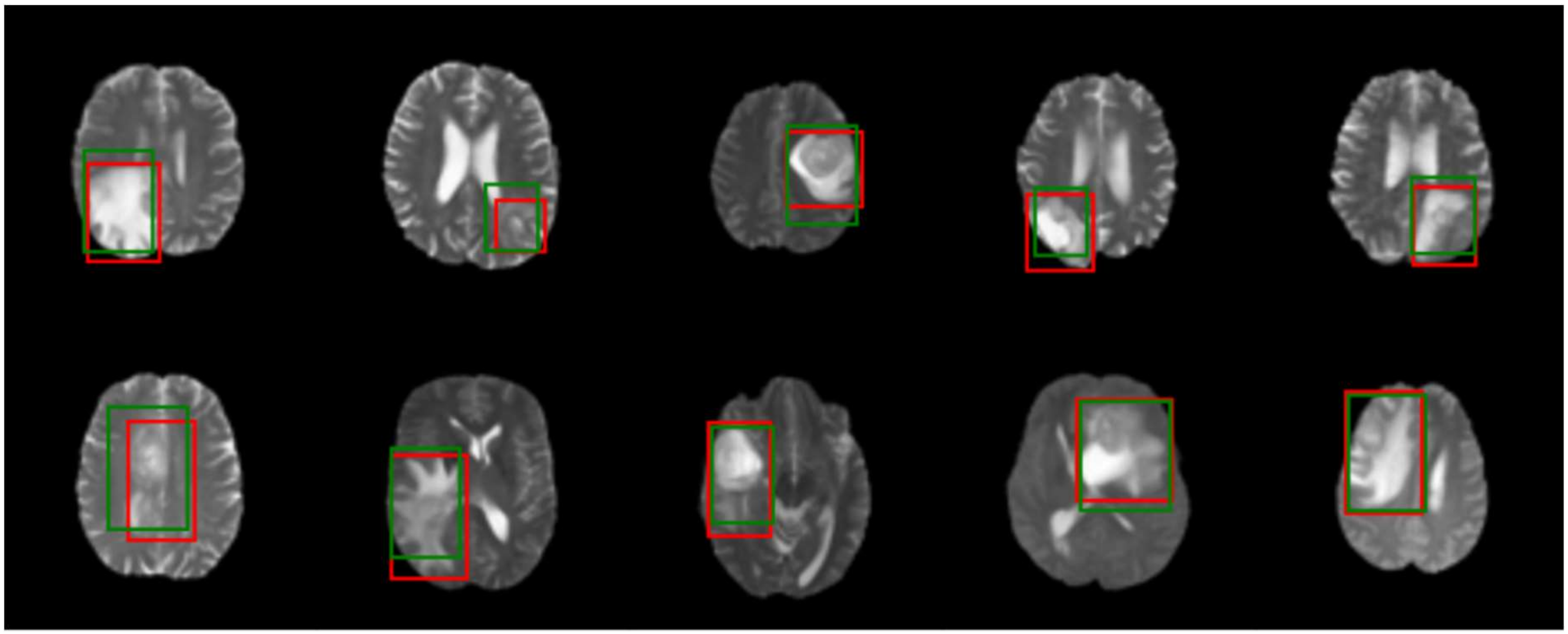
Havaei M., Davy A., Warde-Farley D., Biard A., Courville A., Bengio Y., Pal C., Jodoin P-M, Larochelle H. (2017)
Brain Tumor Segmentation with Deep Neural Networks, Medical Image Analysis, Vol 35, 18-31

Classical applications of ConvNets

Localization

Classical applications of ConvNets

Localization



S. Banerjee, S. Mitra, A. Sharma, and B. U Shankar, A CADe System for Gliomas in Brain MRI using Convolutional Neural Networks, arXiv:1806.07589v, 2018

Conclusion

- Linear classification (1 neuron network)
- Logistic regression
- Multilayer perceptron
- Conv Nets
- Many buzz words
  - Softmax
  - Loss
  - Batch
  - Gradient descent
  - Etc.

Merci