Introducing Deep Learning from Multilayer Perceptron

Rodrigo Fernandes de Mello

Invited Professor at Télécom ParisTech (until July 2019)

Associate Professor at Universidade de São Paulo, ICMC

mello@icmc.usp.br

February 20th, 2019

DigiCosme, Paris-Saclay

Some information on the University of São Paulo, Brazil

- Founded in 1934 in the State of São Paulo
  - 46 million people
  - 12 million in the city of São Paulo (capital)
    - 21 million in the capital and neighborhoods
- 11 Campi
  - Annual budget 1.2 billion Euros
- 96.364 students
  - 58.823 Undergraduation
  - 14.106 Masters
  - 15.894 PhD

DigiCosme, Paris-Saclay

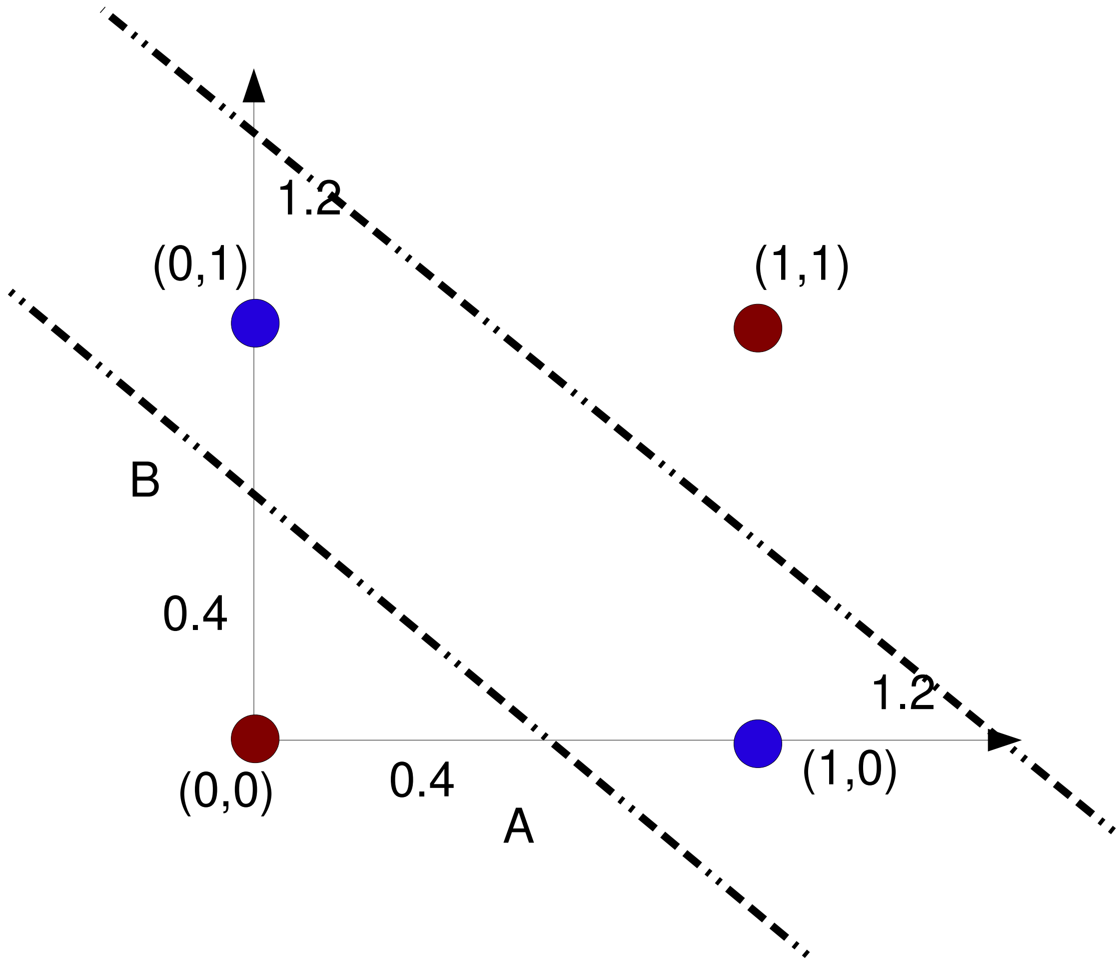
Some information on the University of São Paulo, Brazil

- Students:
  - 52.28% men and 47.72% women
- Staff members
  - 5.844 Professors
  - 14.866 Administrative positions
- World University Ranking of the Times Higher Education

Rank	Name	Overall	Teaching	Research	Citations	Industry Income	International Outlook
251–300	University of São Paulo Brazil	46.4–49.4	55.9	53.5	37.0	39.5	32.7

Multilayer Perceptron is the best way to start

- Classical Toy Problem: XOR

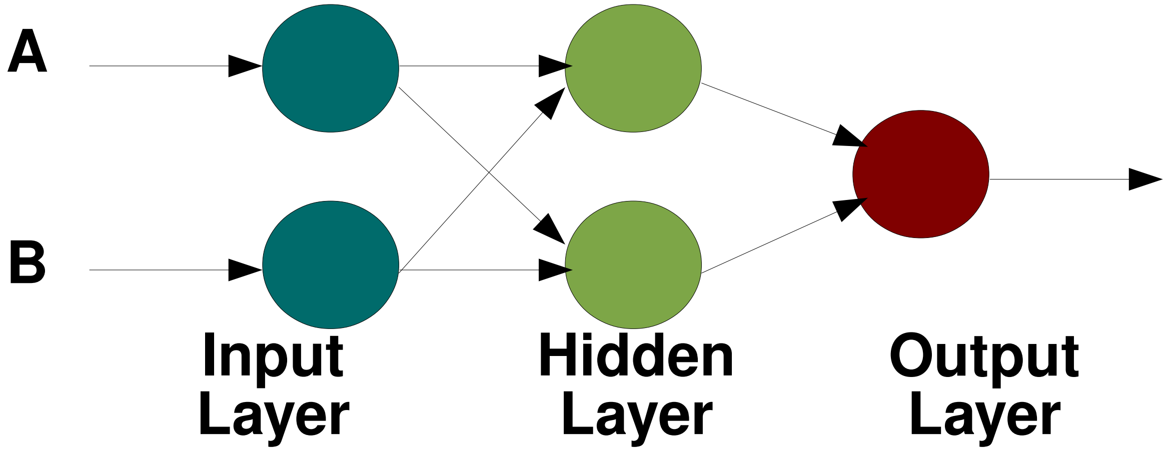


Multilayer Perceptron: Solving XOR

- Feedforward network
  - Inputs produce outputs
  - There is no recurrence such as for BAM and Hopfield neural networks

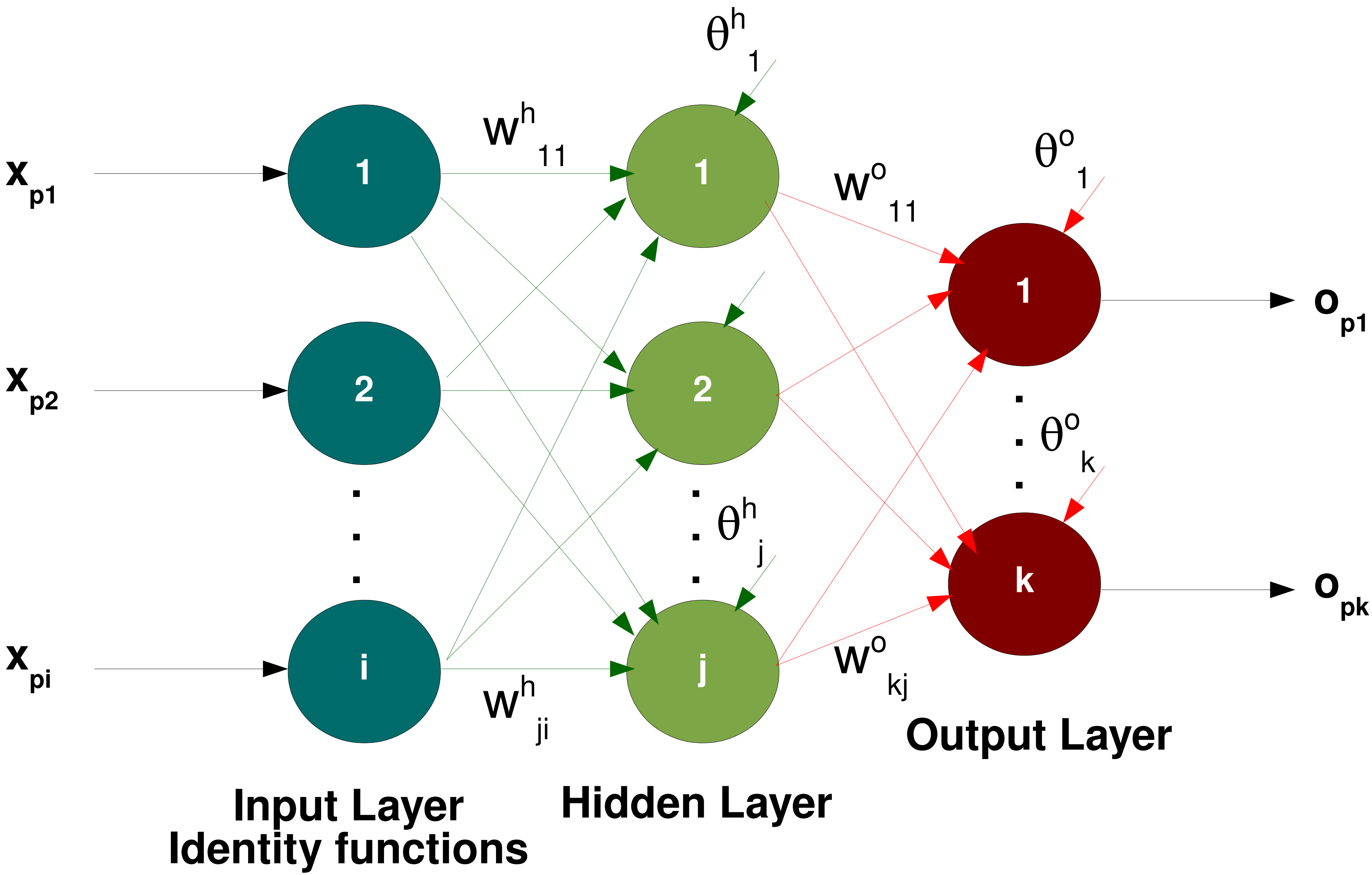
XOR Truth Table

Input		Output
A	B	
0	0	0
0	1	1
1	0	1
1	1	0

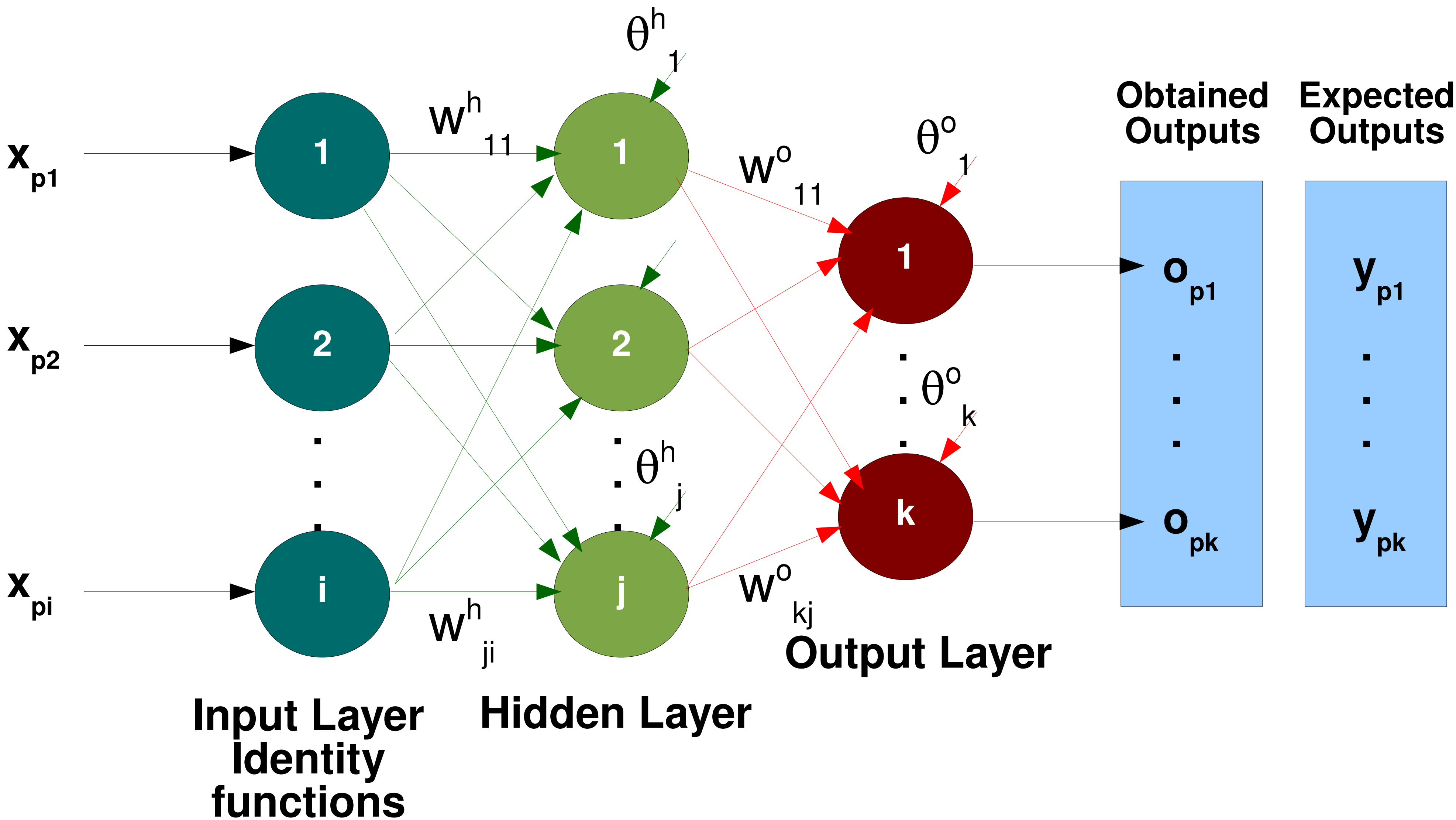


- So we can use the Backpropagation algorithm to train MLP
  - Error is propagated from the last layer in the direction of the first and used to update weights

Multilayer Perceptron: General Purpose Topology

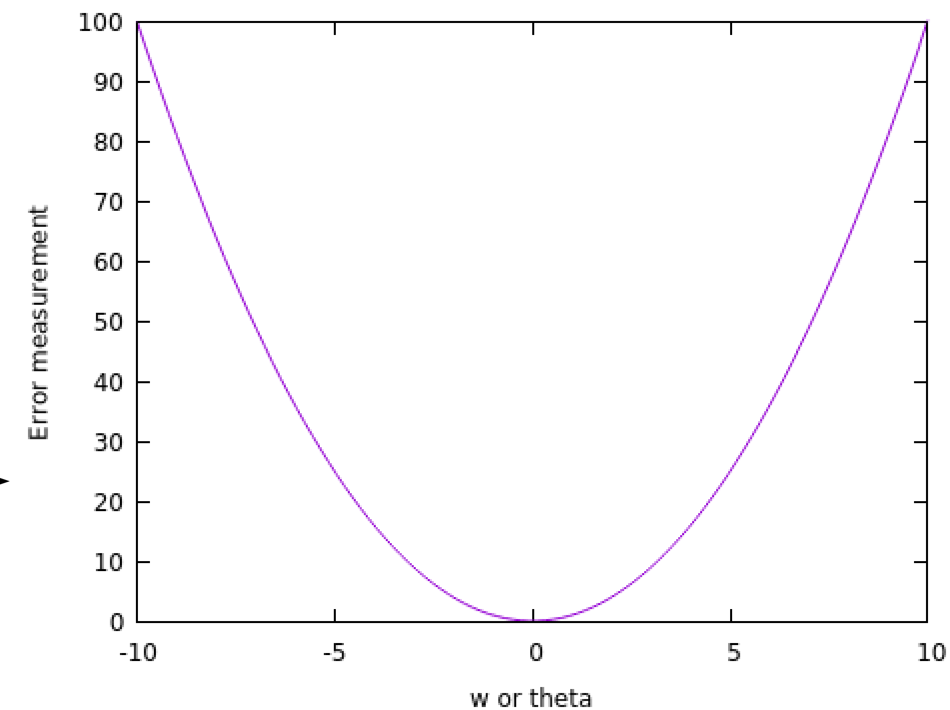


Multilayer Perceptron: General Purpose Topology



Multilayer Perceptron: General Purpose Topology

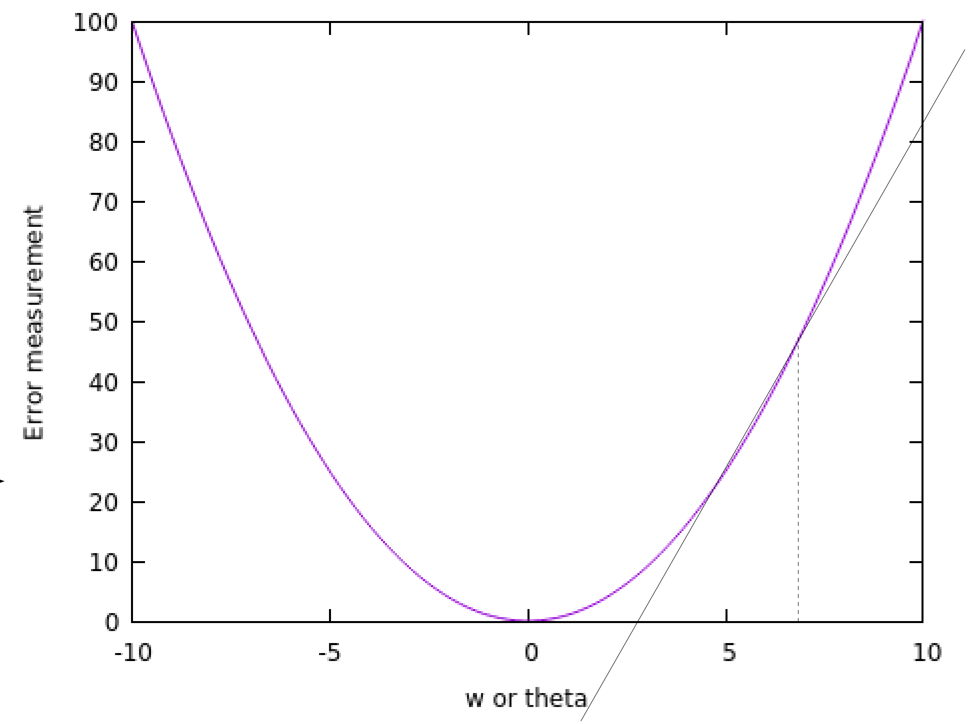
Obtained Outputs	Expected Outputs
o_{p1}	y_{p1}
.	.
.	.
.	.
o_{pk}	y_{pk}



Convex Error Function in terms of w's and θ's

Multilayer Perceptron: General Purpose Topology

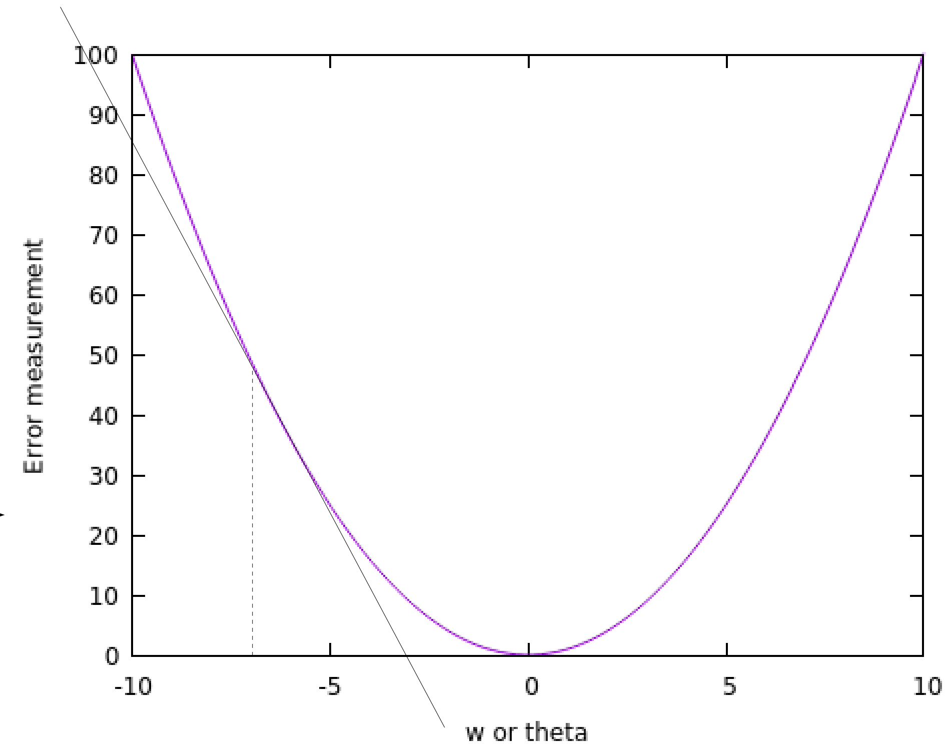
Obtained Outputs	**Expected Outputs**
o_{p1}	y_{p1}
.	.
.	.
.	.
o_{pk}	y_{pk}



Convex Error Function in terms of w's and θ's

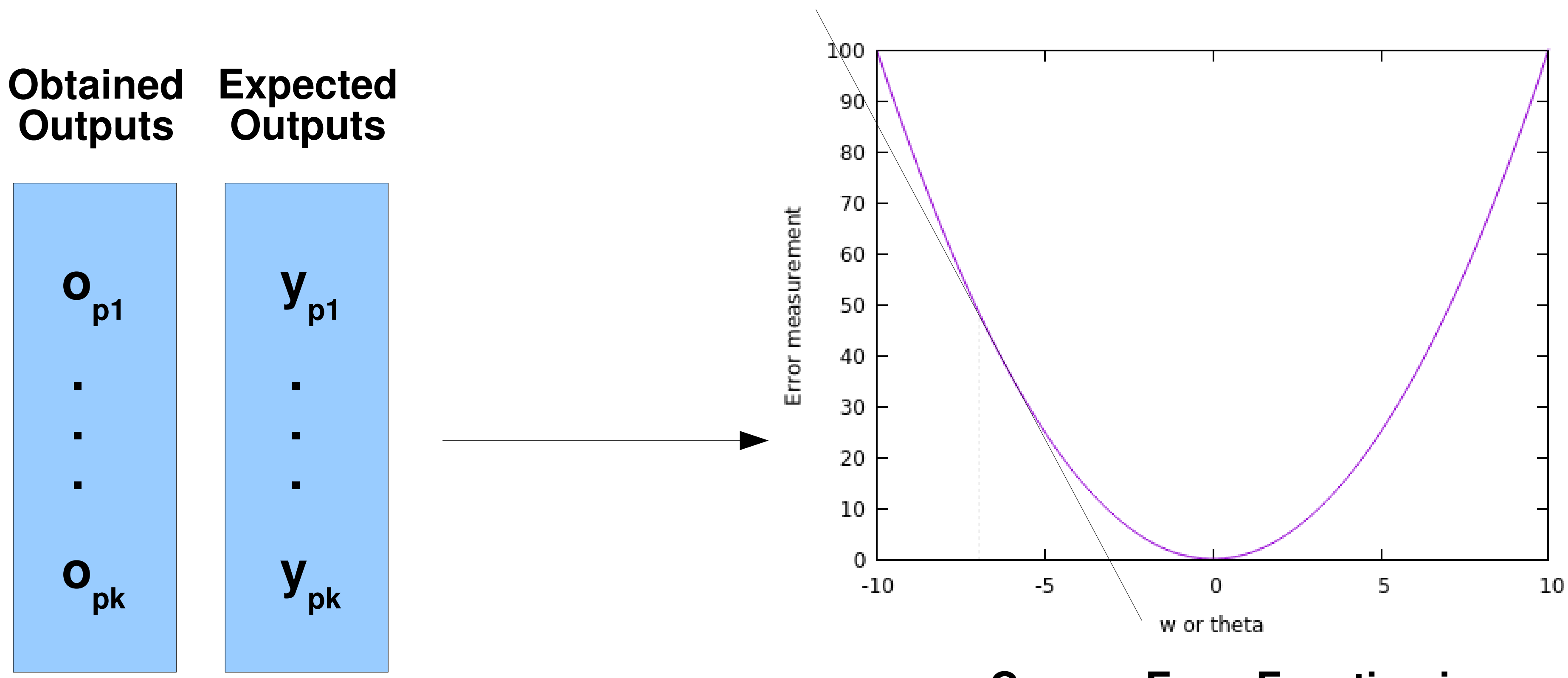
Multilayer Perceptron: General Purpose Topology

Obtained Outputs	Expected Outputs
o_{p1}	y_{p1}
.	.
.	.
.	.
o_{pk}	y_{pk}



Convex Error Function in terms of w's and θ's

Multilayer Perceptron: General Purpose Topology



Convex Error Function in terms of w's and θ's

$$w^h_{ji}(t+1) = w^h_{ji}(t) - \eta \, d\,E / d\,w^h_{ji}$$
$$\theta^h_{j}(t+1) = \theta^h_{j}(t) - \eta \, d\,E / d\,\theta^h_{j}$$
$$w^o_{kj}(t+1) = w^o_{kj}(t) - \eta \, d\,E / d\,w^o_{kj}$$
$$\theta^o_{k}(t+1) = \theta^o_{k}(t) - \eta \, d\,E / d\,\theta^o_{k}$$

Multilayer Perceptron: Generalized Delta Rule

- Training (weight adaptation) occurs using the error measured at the output layer
- Learning follows the Generalized Delta Rule (GDR)
  - It is a generalization of LMS (Least Mean Squares), seen previously
  - LMS is used for linear regression (separates the space using linear functions)
    - This GDR allows non-linear regression
- Suppose:
  - Pairs of vectors (input, expected output):

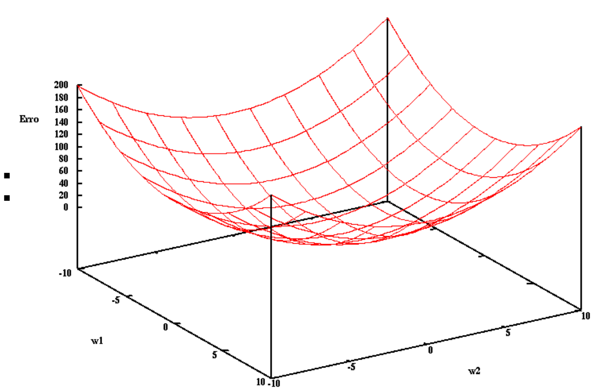
$$(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_p, \mathbf{y}_p)$$

  - Given:

$$\mathbf{y} = \phi(\mathbf{x}) : \mathbf{x} \in \mathbb{R}^N, \mathbf{y} \in \mathbb{R}^M$$

  - The learning objective is to obtain an approximation:

$$\bar{\mathbf{y}} = \bar{\phi}(\mathbf{x})$$



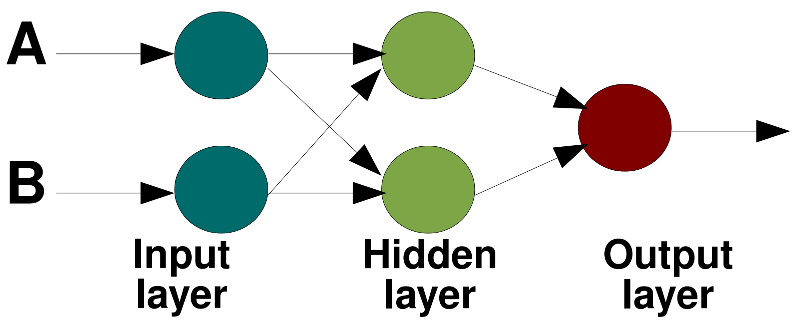
Multilayer Perceptron: Generalized Delta Rule

- The input layer is simple:
  - Neurons only forward values to the hidden layer
- The hidden layer computes:

$$\mathbf{net}_{pj}^{h} = \sum_{i=1}^{N} w_{ji}^{h} x_{pi} + \theta_j^{h}$$

- The output layer computes:

$$\mathbf{net}_{pk}^{o} = \sum_{j=1}^{L} w_{kj}^{o} i_{pj} + \theta_k^{o}$$



- In which:

w_{ji}^{h} é o peso da conexão com o neurônio de entrada i

w_{kj}^{o} é o peso da conexão com o neurônio j da camada escondida

θ_j^{h} e θ_k^{o} são os bias

- **Updating weights at the output layer**
- The output layer may contain several neurons
  - The error for a given neuron at this layer is given by:

$$\delta_{pk} = (y_{pk} - o_{pk})$$

  - Having:

y_{pk} saída esperada do neurônio k para vetor de entrada p
o_{pk} saída produzida pelo neurônio k para vetor de entrada p
p identifica o vetor de entrada usado no treinamento
k indica o neurônio da camada de saída

Multilayer Perceptron: Generalized Delta Rule
Updating weights at the output layer

- The objective is to minimize the squared sum of errors to all output units, considering an input p

$$\mathbf{E}_p = \frac{1}{2}\sum_{k=1}^{M}\delta_{pk}^2$$

  - The factor ½ is added to simplify the derivative
    - As we will have another constant to adapt weights, this term ½ does not change anything in terms of concepts, only the step
  - **M** indicates the number of neurons in the output layer
  - Important:
    - The error associated to each input is squared

- Our objective is to “walk” in the direction to reduce the error, which varies according to weights w

$$\mathbf{E}_p = \frac{1}{2}\sum_{k=1}^{M}\delta_{pk}^2$$

$$\delta_{pk} = (y_{pk} - o_{pk})$$

$$\mathbf{E}_p = \frac{1}{2}\sum_{k=1}^{M}(y_{pk} - o_{pk})^2$$

- Deriving the error in the direction of what can be changed (weights w), for that we have (according to the chain rule):

$$\frac{\partial}{\partial x} f(g(x)) = f'(g(x))g'(x) \text{ ou } \frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial x}$$

- Simplifying:

$$E_{pk} = \frac{1}{2}(y_{pk} - o_{pk})^2 \text{ em que:}$$
$$f(g(x)) = \frac{1}{2}(y_{pk} - o_{pk})^2$$
$$g(x) = y_{pk} - o_{pk}$$

Multilayer Perceptron: Generalized Delta Rule
Updating weights at the output layer

- Thus:

$$f'(g(x)) = 2 \cdot \frac{1}{2}(y_{pk} - o_{pk})$$
$$g'(x) = 0 - o'_{pk}$$
$$\text{em que } y_{pk} \text{ é uma constante (saída esperada)}$$

- in which:

$$o_{pk} = f_k^o(\mathbf{net}_{pk}^o)$$

- Therefore the derivative will be (also following the chain rule):

$$o'_{pk} = \frac{\partial f_k^o}{\partial \mathbf{net}_{pk}^o} \cdot \frac{\partial \mathbf{net}_{pk}^o}{\partial w_{kj}^o}$$

- Unifying:

$$f'(g(x))g'(x) = \left(2 \cdot \frac{1}{2}(y_{pk} - o_{pk})\right) \cdot \left(0 - \frac{\partial f_k^o}{\partial \mathbf{net}_{pk}^o} \frac{\partial \mathbf{net}_{pk}^o}{\partial w_{kj}^o}\right)$$

- We have:

$$\frac{\partial E_{pk}}{\partial w_{kj}^o} = -(y_{pk} - o_{pk}) \frac{\partial f_k^o}{\partial \mathbf{net}_{pk}^o} \frac{\partial \mathbf{net}_{pk}^o}{\partial w_{kj}^o}$$

- Solving the last term we find:

$$\mathbf{net} = \sum_{j=1}^{L} w_{kj}^o i_{pj} + \theta_k^o$$

$$\frac{\partial \mathbf{net}_{pk}^o}{\partial w_{kj}^o} = \frac{\partial}{\partial w_{kj}^o} \left(\sum_{j=1}^{L} w_{kj}^o i_{pj} + \theta_k^o \right) = i_{pj}$$

Multilayer Perceptron: Generalized Delta Rule
Updating weights at the output layer

- Substituting:

$$\frac{\partial E_{pk}}{\partial w^o_{pj}} = -(y_{pk} - o_{pk})\frac{\delta f^o_k}{\partial \mathbf{net}^o_{pk}} i_{pj}$$

- We still have to differentiate the activation function
  - So this function MUST be differentiable
  - This avoids the usage of the step function (Perceptron)
- Examples of activation functions:

$$1)\ \text{se } f^o_k(\mathbf{net}^o_k) = \mathbf{net}^o_k \text{ então } f'^o_k(\mathbf{net}^o_k) = 1$$
$$\text{assim como } f(x) = x \text{ temos } f'(x) = 1$$

Linear Function

$$2)\ \text{se } f^o_k(\mathbf{net}^o_k) = (1 + e^{-\mathbf{net}^o_{jk}})^{-1} \text{ então}$$
$$f'^o_k(\mathbf{net}^o_k) = f^o_k(1 - f^o_k)$$

Sigmoid Function

Multilayer Perceptron: Generalized Delta Rule
Updating weights at the output layer

- Considering the two possibilities for the activation function, we have the weight adaptation as follows:
  - For the **linear function**:

$$f_k^o(\mathbf{net}_{jk}^o) = \mathbf{net}_{jk}^o$$

  - We have:

$$w_{kj}^o(t+1) = w_{kj}^o(t) + \eta(y_{pk} - o_{pk})i_{pj}$$

  - For the **sigmoid function**:

$$f_k^o(\mathbf{net}_{jk}^o) = \frac{1}{1 + e^{-\mathbf{net}_{jk}^o}}$$

  - We have:

$$f_k'^o(\mathbf{net}_k^o) = f_k^o(1 - f_k^o) \text{ logo, neste cenário } f_k'^o(\mathbf{net}_k^o) = o_{pk}(1 - o_{pk})$$

$$w_{kj}^o(t+1) = w_{kj}^o(t) + \eta(y_{pk} - o_{pk})o_{pk}(1 - o_{pk})i_{pj}$$

Multilayer Perceptron: Generalized Delta Rule
Updating weights at the output layer

- We can define the adaptation term in a generic way, i.e., for any activation function:

$$\delta^o_{pk} = (y_{pk} - o_{pk}) f^{o\prime}_k(\mathbf{net}^o_{pk})$$

- And generalize (**Generalized Delta Rule**) the weight adaptation for any activation function, as follows:

$$w^o_{kj}(t+1) = w^o_{kj}(t) + \eta \delta^o_{pk} i_{pj}$$

- **Updating hidden layer weights**
  - How can we know the expected output for each neuron at the hidden layer?
    - At the output layer we know what is expected!
  - In some way, error E_p measured at the output layer must influence in the hidden layer weights

Multilayer Perceptron: Generalized Delta Rule
Updating weights at the hidden layer

- The error measured at the output layer is given by:

$$
\begin{aligned}
E_p &= \frac{1}{2}\sum_k (y_{pk} - o_{pk})^2 \\
&= \frac{1}{2}\sum_k (y_{pk} - f_k^o(\mathbf{net}_{pk}^o))^2 \\
&= \frac{1}{2}\sum_k \left(y_{pk} - f_k^o\left(\sum_j w_{kj}^o i_{pj} + \theta_k^o\right)\right)^2
\end{aligned}
$$

- Term i_{pj} refers to the values produced by the previous (hidden) layer
  - So we can explore this fact to build equations to adapt hidden layer weights

Multilayer Perceptron: Generalized Delta Rule
Updating weights at the hidden layer

- In this manner, we define the error variation in terms of hidden layer weights:

$$\frac{\partial E_p}{\partial w_{ji}^h} = \frac{1}{2}\sum_k \frac{\partial}{\partial w_{ji}^h}(y_{pk} - o_{pk})^2$$
$$= -\sum_k (y_{pk} - o_{pk}) \frac{\partial o_{pk}}{\partial \mathbf{net}_{pk}^o} \frac{\partial \mathbf{net}_{pk}^o}{\partial i_{pj}} \frac{\partial i_{pj}}{\partial \mathbf{net}_{pj}^h} \frac{\partial \mathbf{net}_{pj}^h}{\partial w_{ji}^h}$$

- From that we obtain:

$$\frac{\partial E_p}{\partial w_{ji}^h} = -\sum_k (y_{pk} - o_{pk}) f_k^{o\prime}(\mathbf{net}_{pk}^o) w_{kj}^o f_j^{h\prime}(\mathbf{net}_{pj}^h) x_{pi}$$

- In this manner, we define the error variation in terms of hidden layer weights:

$$\frac{\partial E_p}{\partial w_{ji}^h} = \frac{1}{2}\sum_k \frac{\partial}{\partial w_{ji}^h}(y_{pk} - o_{pk})^2$$

$$= -\sum_k (y_{pk} - o_{pk}) \boxed{\frac{\partial o_{pk}}{\partial \mathbf{net}_{pk}^o}} \frac{\partial \mathbf{net}_{pk}^o}{\partial i_{pj}} \boxed{\frac{\partial i_{pj}}{\partial \mathbf{net}_{pj}^h}} \frac{\partial \mathbf{net}_{pj}^h}{\partial w_{ji}^h}$$

Derivative of the activation function at the output layer in the direction of net

- From that we obtain:

Derivative of the activation function at the hidden layer in the direction of net

$$\frac{\partial E_p}{\partial w_{ji}^h} = -\sum_k (y_{pk} - o_{pk}) \boxed{f_k^{o\prime}(\mathbf{net}_{pk}^o)} w_{kj}^o \boxed{f_j^{h\prime}(\mathbf{net}_{pj}^h)} x_{pi}$$

Multilayer Perceptron: Generalized Delta Rule
Updating weights at the hidden layer

- Having:

$$\mathbf{net}_{pk}^{o} = \sum_{j=1}^{L} w_{kj}^{o} i_{pj} + \theta_k^{o}$$

$$\mathbf{net}_{pi}^{h} = \sum_{j=1}^{L} w_{kj}^{h} x_{pi} + \theta_k^{h}$$

$$\begin{aligned}
\frac{\partial E_p}{\partial w_{ji}^{h}} &= \frac{1}{2}\sum_k \frac{\partial}{\partial w_{ji}^{h}}(y_{pk} - o_{pk})^2 \\
&= -\sum_k (y_{pk} - o_{pk}) \frac{\partial o_{pk}}{\partial \mathbf{net}_{pk}^{o}} \boxed{\frac{\partial \mathbf{net}_{pk}^{o}}{\partial i_{pj}}} \frac{\partial i_{pj}}{\partial \mathbf{net}_{pj}^{h}} \boxed{\frac{\partial \mathbf{net}_{pj}^{h}}{\partial w_{ji}^{h}}}
\end{aligned}$$

$$\frac{\partial E_p}{\partial w_{ji}^{h}} = -\sum_k (y_{pk} - o_{pk}) f_k^{o\prime}(\mathbf{net}_{pk}^{o}) \boxed{w_{kj}^{o}} f_j^{h\prime}(\mathbf{net}_{pj}^{h}) \boxed{x_{pi}}$$

- So we can compute the weight adaptation for the hidden layer in form:

$$\triangle_p w_{ji}^h = \eta f_j^{h\prime}(\mathbf{net}_{pj}^h) x_{pi} \sum_k (y_{pk} - o_{pk}) f_k^{o\prime}(\mathbf{net}_{pk}^o) w_{kj}^o$$

$$\triangle_p w_{ji}^h = \eta f_j^{h\prime}(\mathbf{net}_{pj}^h) x_{pi} \sum_k \delta_{pk}^o w_{kj}^o$$

- In this manner, the weight adaptation at the hidden layer depends on the error at the output layer
  - The term **Backpropagation** comes from this notion of dependency on the error at the output layer

Multilayer Perceptron: Generalized Delta Rule
Updating weights at the hidden layer

- So we can compute the term delta for the hidden layer in the same way we did for the output layer:

$$\delta_{pj}^{h} = f_j^{h\prime}(\mathbf{net}_{pj}^{h}) \sum_k \delta_{pk}^{o} w_{kj}^{o}$$

- In this way, the weight adaptation is given by:

$$w_{ji}^{h}(t+1) = w_{ji}^{h}(t) + \eta \delta_{pj}^{h} x_i$$

- Finally, we can implement the MLP learning algorithm

Multilayer Perceptron

- **Backpropagation learning algorithm**
- Essential functions to update weights:
  - Considering the sigmoid activation function (this is the default):

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

- Output layer:

$$\delta^o_{pk} = (y_{pk} - o_{pk}) f^{o\prime}_k(\mathbf{net}^o_{pk})$$

$$w^o_{kj}(t+1) = w^o_{kj}(t) + \eta \delta^o_{pk} i_{pj}$$

This is very important!!!

First, we MUST compute all deltas so then we update weights!!!

- Hidden layer:

$$\delta^h_{pj} = f^{h\prime}_j(\mathbf{net}^h_{pj}) \sum_k \delta^o_{pk} w^o_{kj}$$

$$w^h_{ji}(t+1) = w^h_{ji}(t) + \eta \delta^h_{pj} x_i$$

Multilayer Perceptron

- **Backpropagation learning algorithm**
- Essential functions to update weights:
  - Considering the sigmoid activation function (this is the default):

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

- Output layer:

k identifies the neuron

$$\delta^o_{pk} = (y_{pk} - o_{pk}) f^{o\prime}_k(\mathbf{net}^o_{pk})$$

$$w^o_{kj}(t+1) = w^o_{kj}(t) + \eta \delta^o_{pk} i_{pj}$$

- Hidden layer:

$$\delta^h_{pj} = f^{h\prime}_j(\mathbf{net}^h_{pj}) \sum_k \delta^o_{pk} w^o_{kj}$$

$$w^h_{ji}(t+1) = w^h_{ji}(t) + \eta \delta^h_{pj} x_i$$

Multilayer Perceptron

- **Backpropagation learning algorithm**
- Essential functions to update weights:
    - Considering the sigmoid activation function (this is the default):

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

This refers to the layer (hidden or output)

- Output layer:

$$\delta^o_{pk} = (y_{pk} - o_{pk}) f^{o\prime}_k (\mathbf{net}^o_{pk})$$

$$w^o_{kj}(t+1) = w^o_{kj}(t) + \eta \delta^o_{pk} i_{pj}$$

- Hidden layer:

$$\delta^h_{pj} = f^{h\prime}_j (\mathbf{net}^h_{pj}) \sum_k \delta^o_{pk} w^o_{kj}$$

$$w^h_{ji}(t+1) = w^h_{ji}(t) + \eta \delta^h_{pj} x_i$$

Multilayer Perceptron

- **Backpropagation learning algorithm**
- Essential functions to update weights:
  - Considering the sigmoid activation function (this is the default):

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

- Output layer:

p identifies the input vector

$$\delta^o_{pk} = (y_{pk} - o_{pk}) f^{o\prime}_k(\mathbf{net}^o_{pk})$$

$$w^o_{kj}(t+1) = w^o_{kj}(t) + \eta \delta^o_{pk} i_{pj}$$

- Hidden layer:

$$\delta^h_{pj} = f^{h\prime}_j(\mathbf{net}^h_{pj}) \sum_k \delta^o_{pk} w^o_{kj}$$

$$w^h_{ji}(t+1) = w^h_{ji}(t) + \eta \delta^h_{pj} x_i$$

Multilayer Perceptron

- **Backpropagation learning algorithm**
- Essential functions to update weights:
  - Considering the sigmoid activation function (this is the default):

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

- Output layer:

Input values produced by the hidden layer

$$\delta^o_{pk} = (y_{pk} - o_{pk}) f^{o\prime}_k(\mathbf{net}^o_{pk})$$

$$w^o_{kj}(t+1) = w^o_{kj}(t) + \eta \delta^o_{pk} i_{pj}$$

- Hidden layer:

Input values produced by the input layer

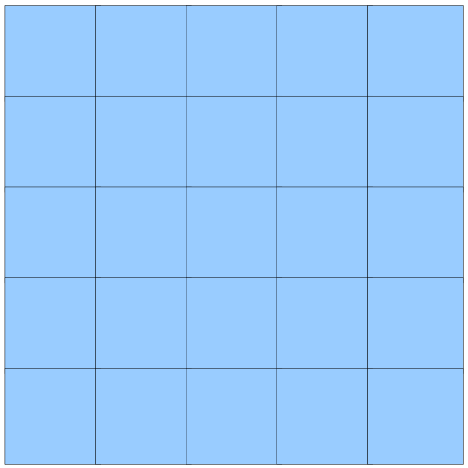
$$\delta^h_{pj} = f^{h\prime}_j(\mathbf{net}^h_{pj}) \sum_k \delta^o_{pk} w^o_{kj}$$

$$w^h_{ji}(t+1) = w^h_{ji}(t) + \eta \delta^h_{pj} x_i$$

Deep Learning in terms of Convolutional Neural Networks (CNNs)

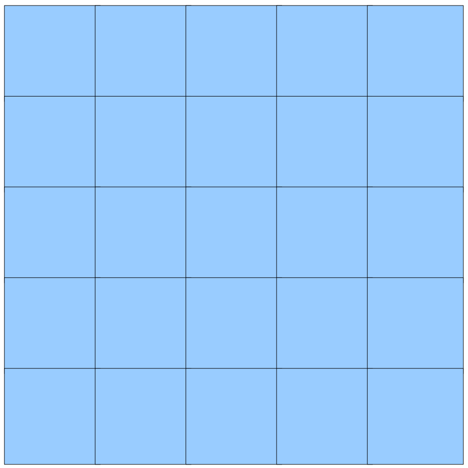
CNNs

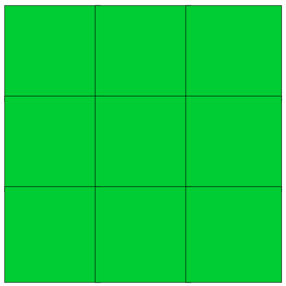
- Images as examples



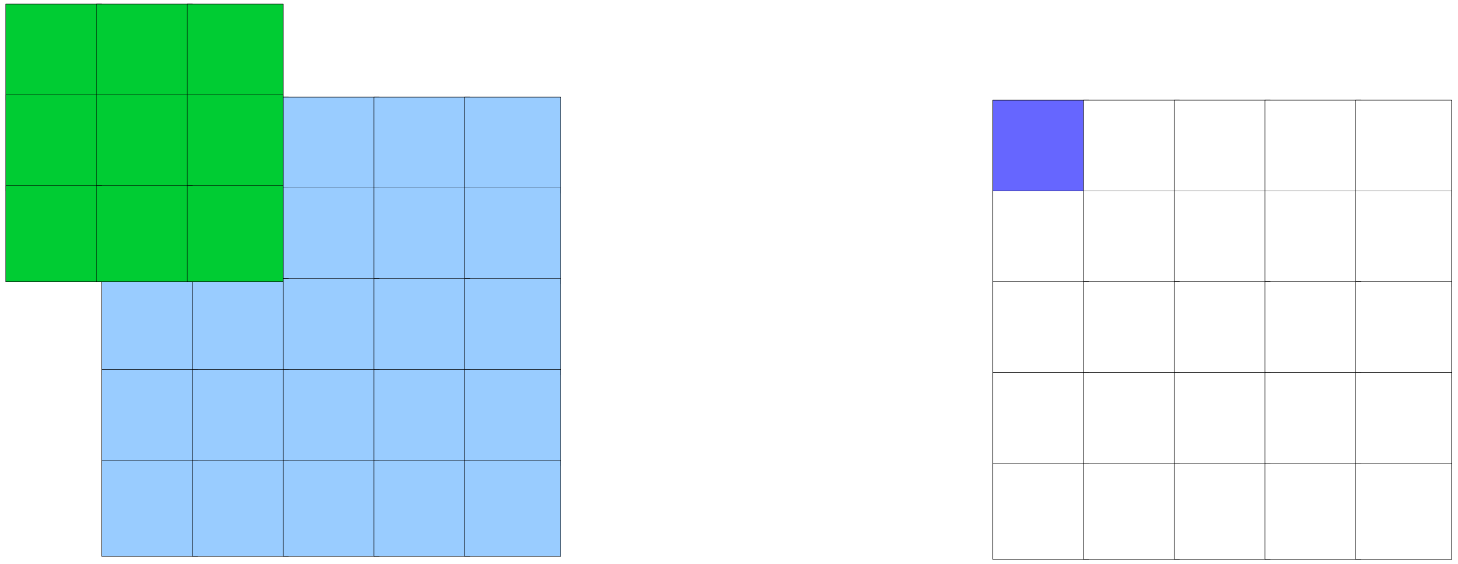
CNNs

- Images as examples and application of convolutional masks

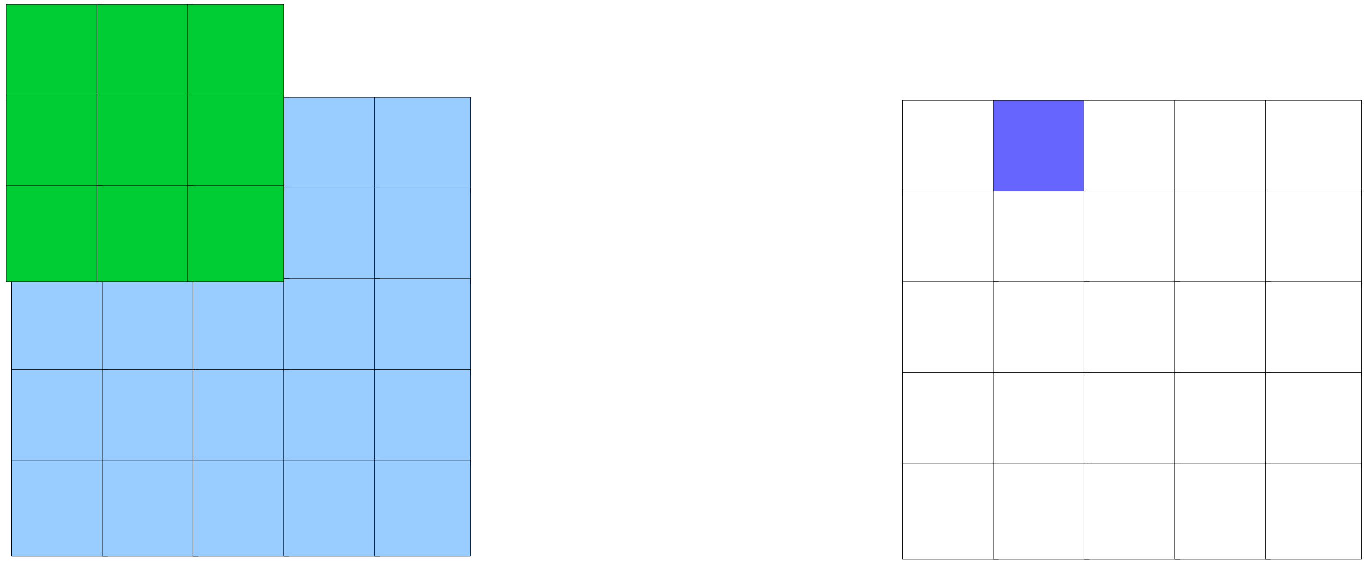




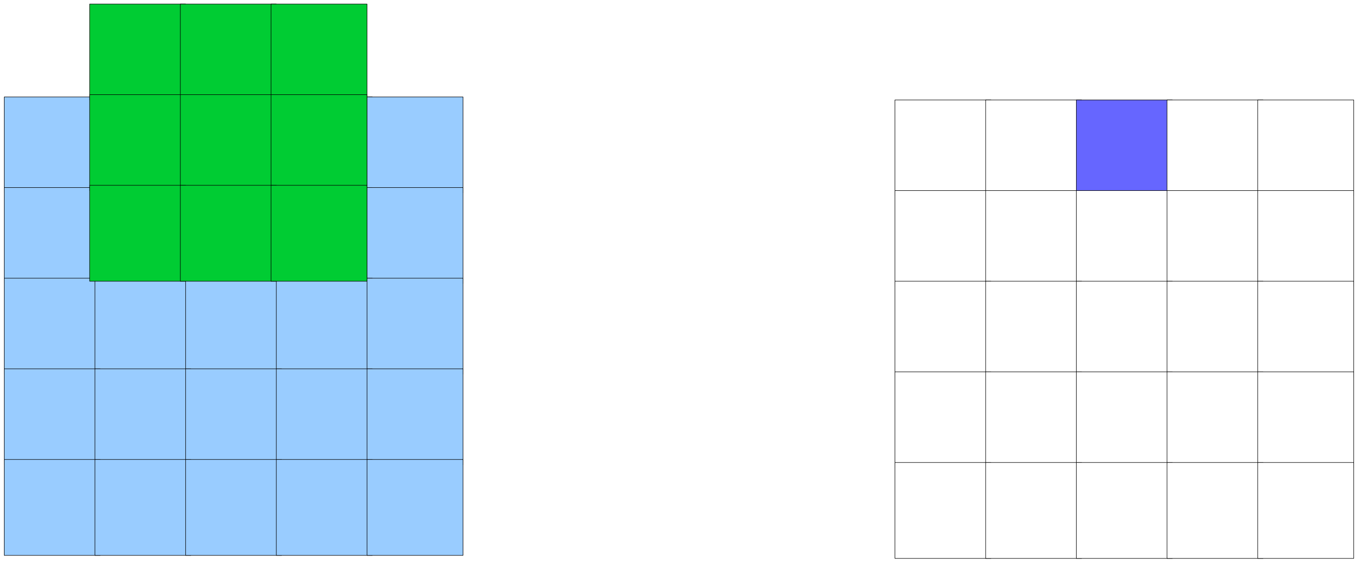
- Images as examples and application of convolutional masks



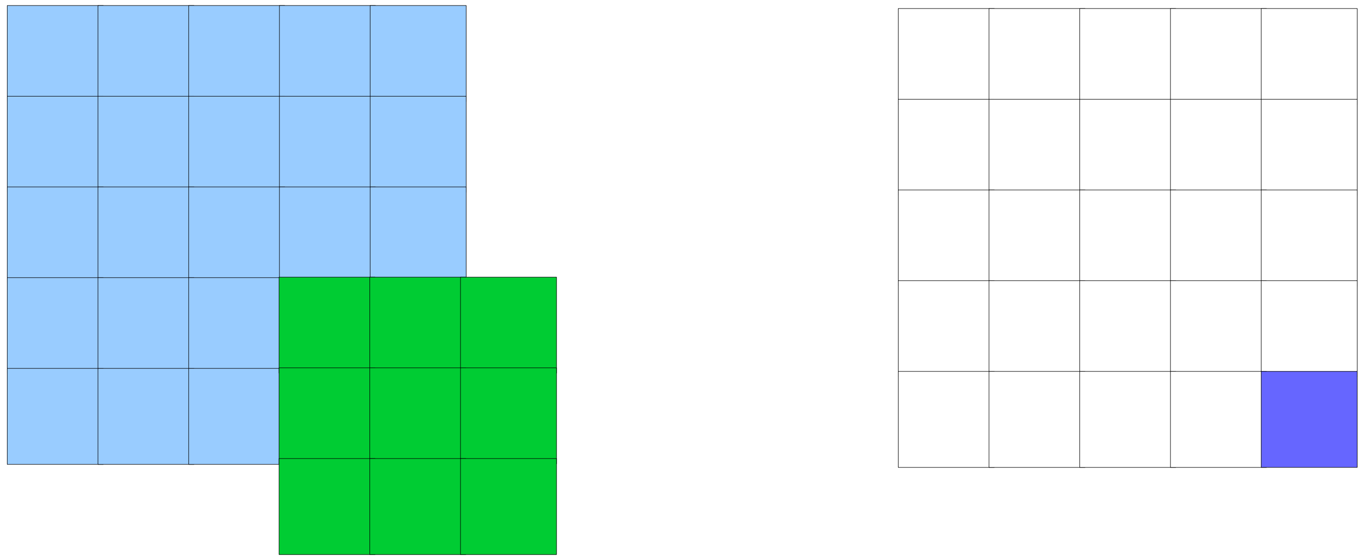
- Images as examples and application of convolutional masks



- Images as examples and application of convolutional masks



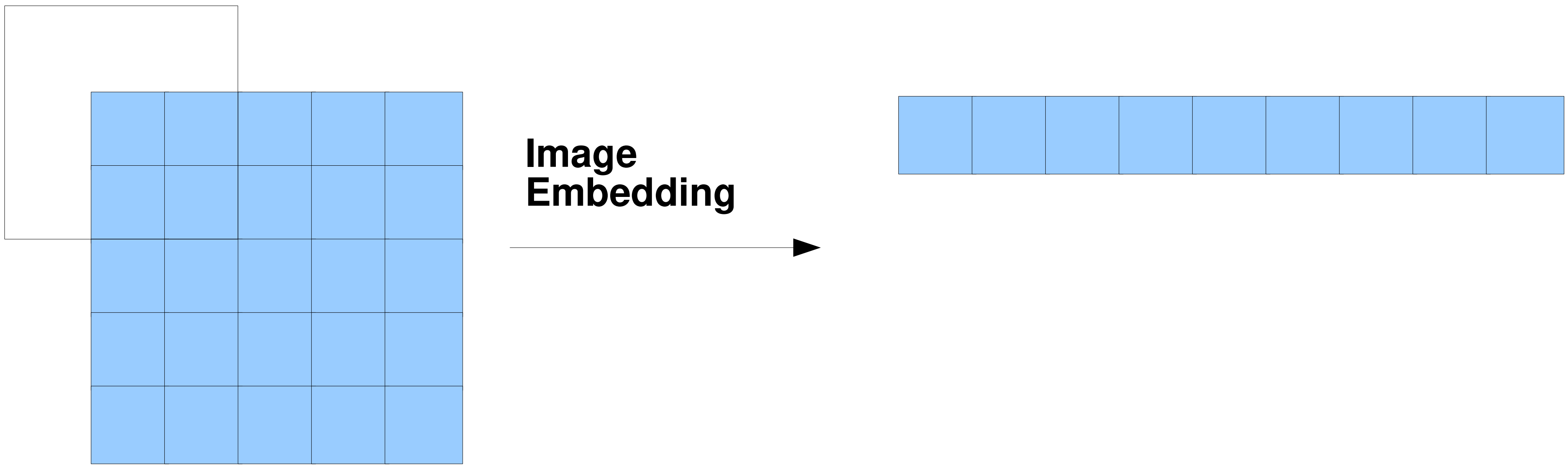
- Images as examples and application of convolutional masks



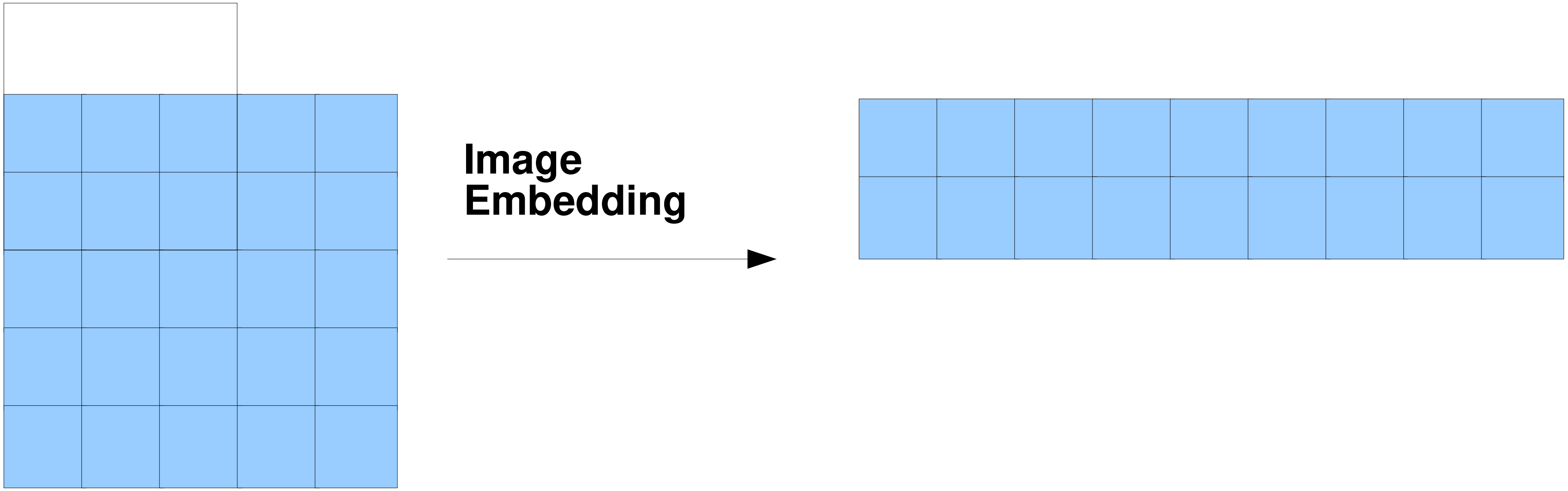
However we can see the same operation from a different Perspective

CNNs

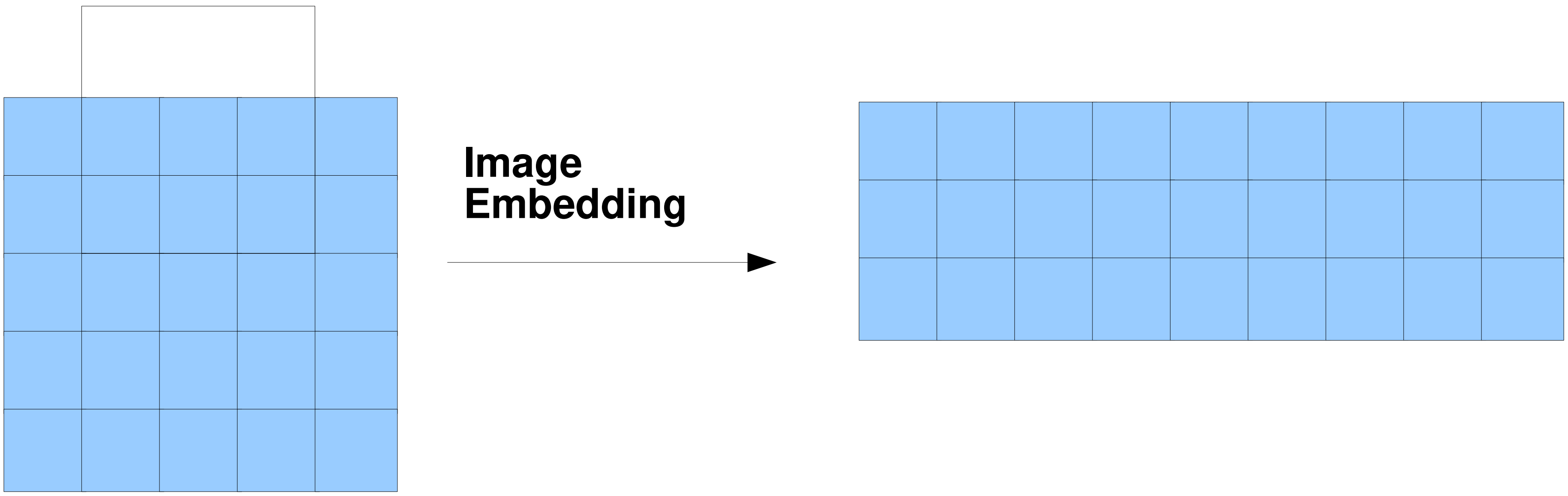
- Images as examples and application of convolutional masks



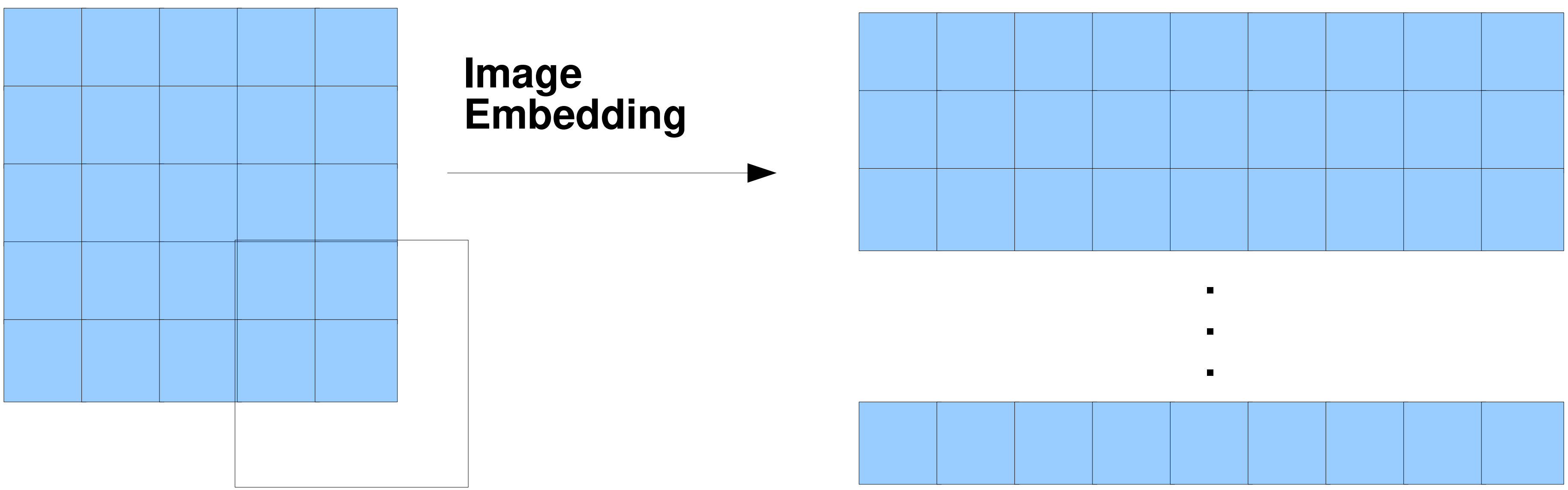
- Images as examples and application of convolutional masks



- Images as examples and application of convolutional masks

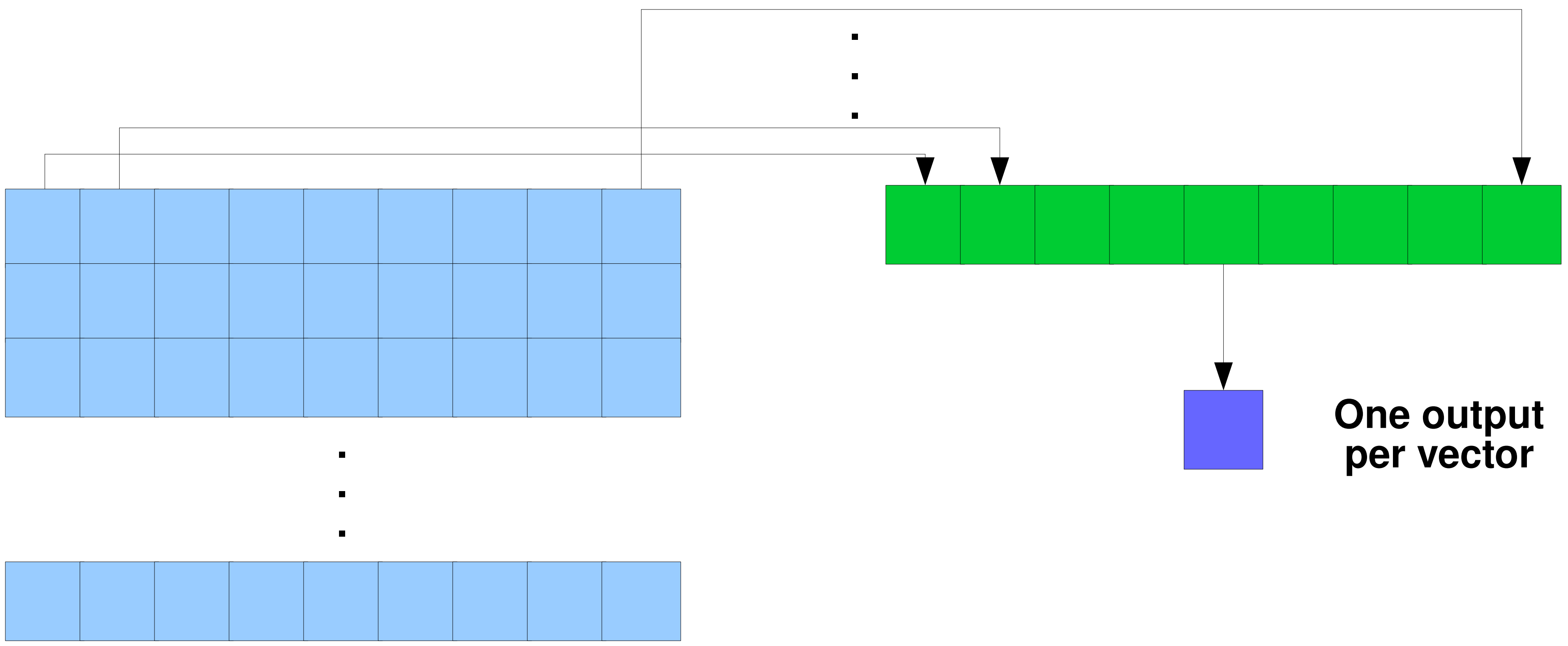


- Images as examples and application of convolutional masks



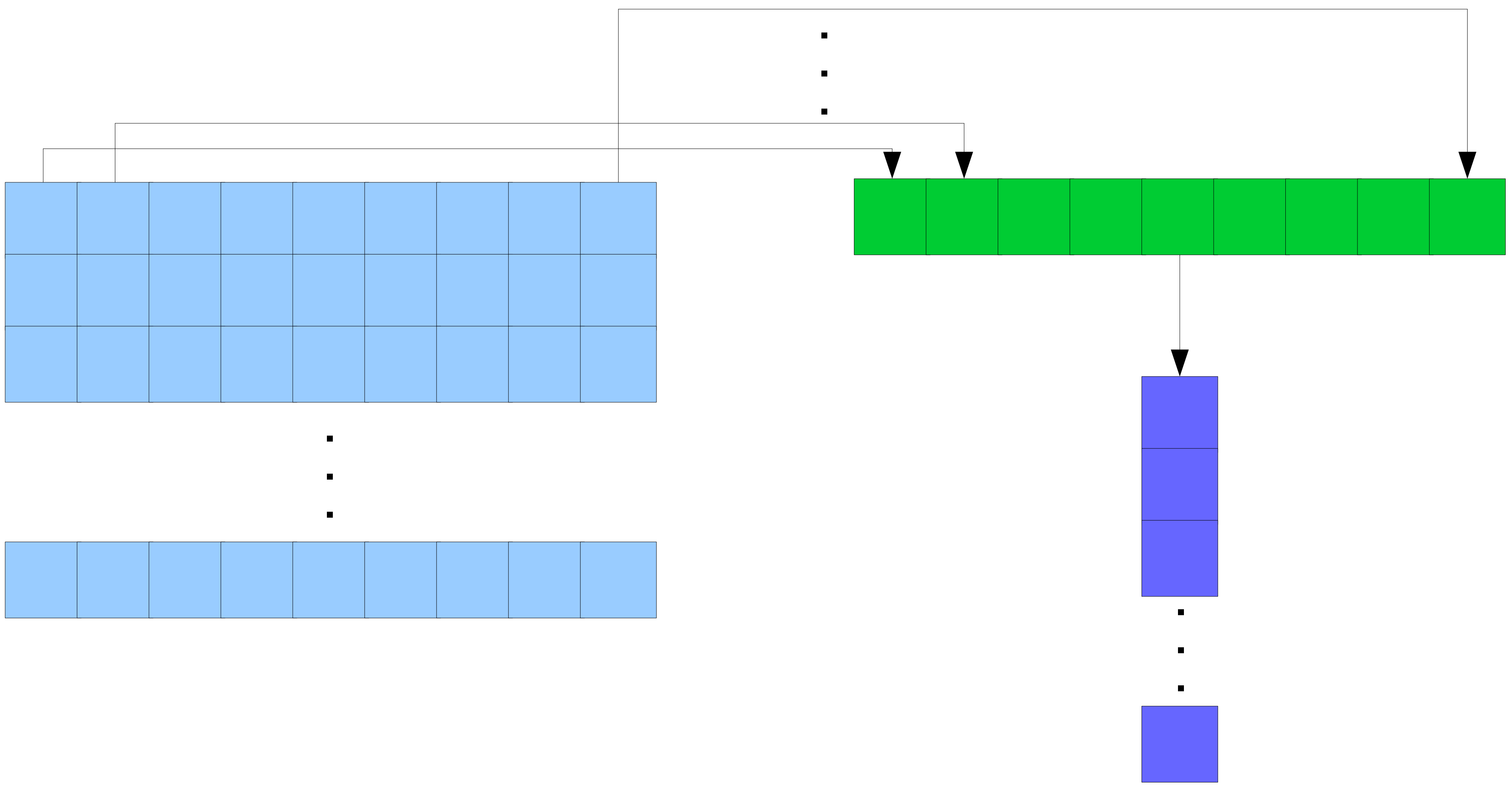
And after embedding...

- Images as examples and application of convolutional masks



CNNs

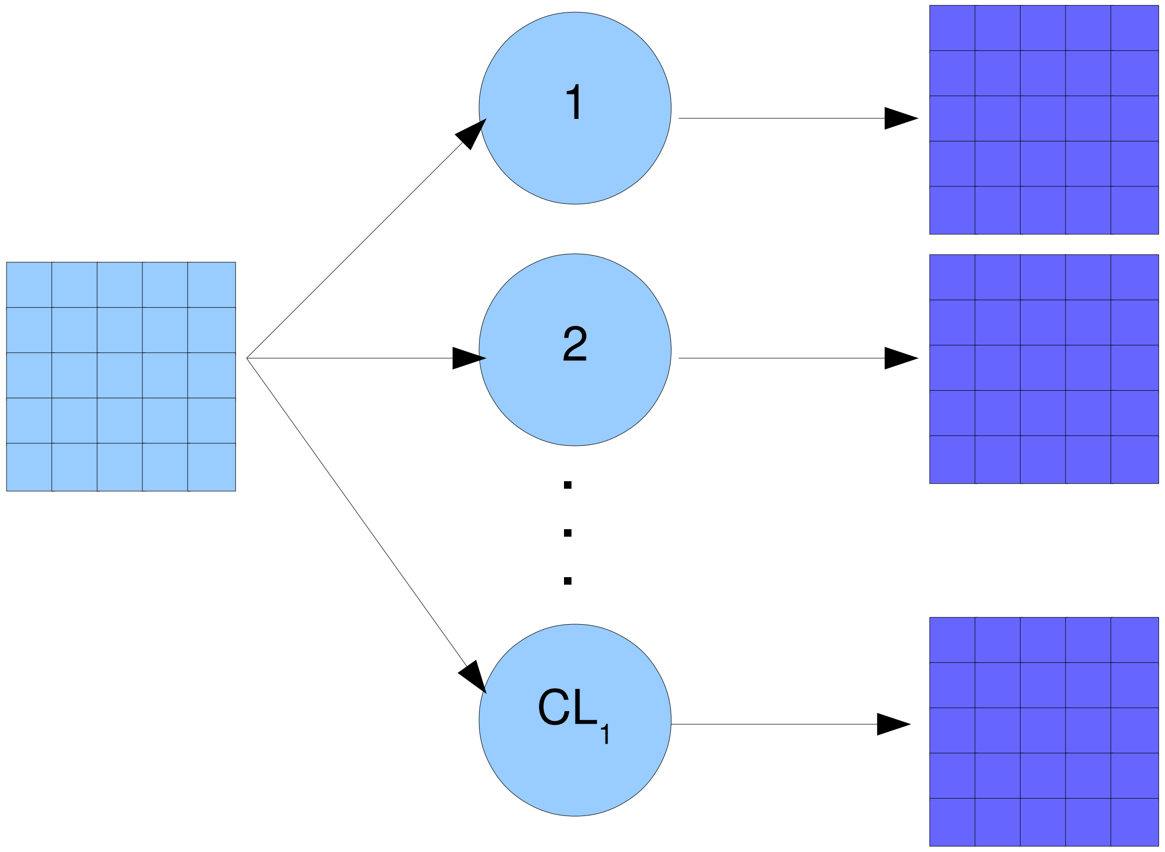
- Images as examples and application of convolutional masks



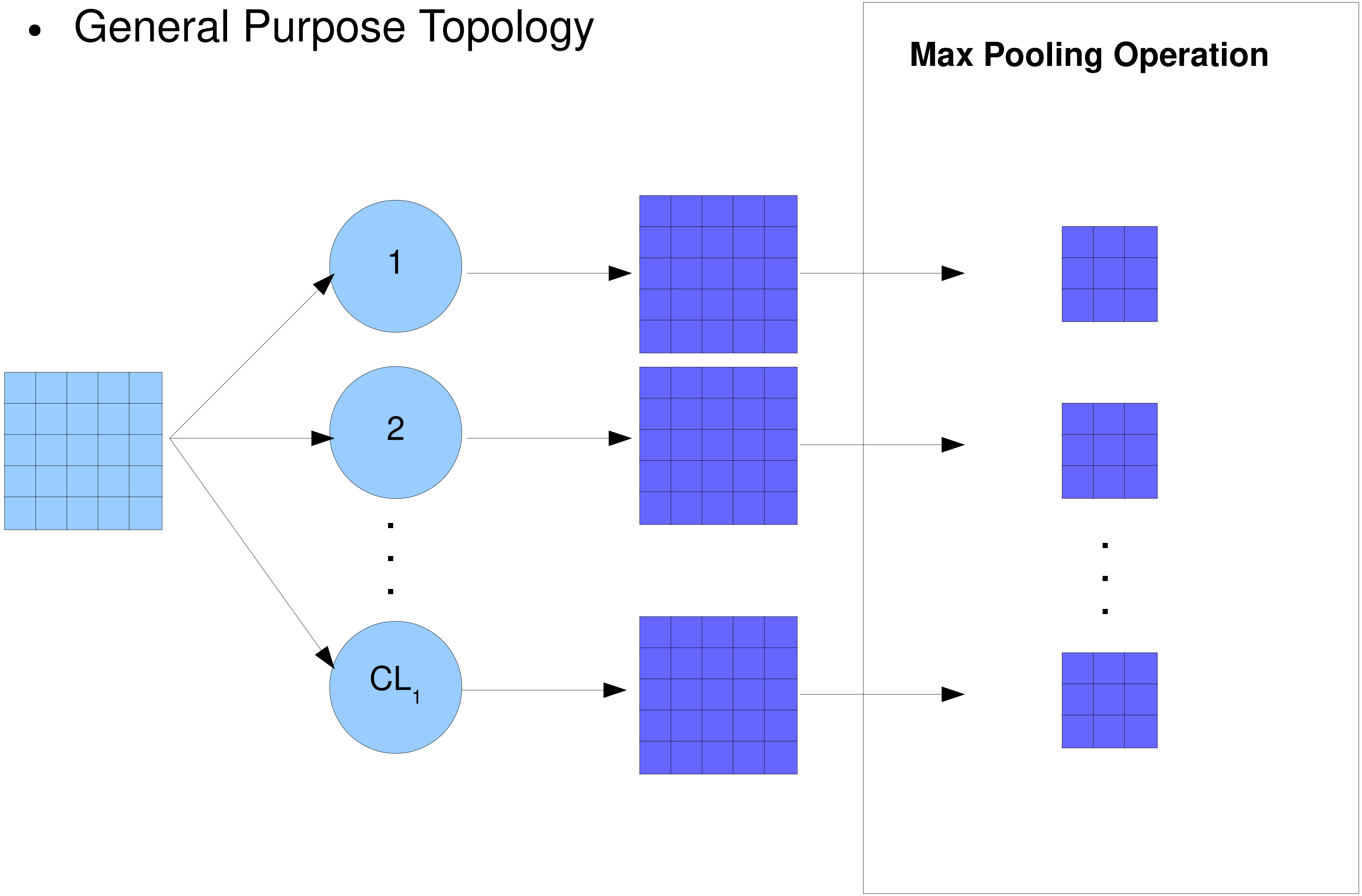
Thus, every convolutional neuron creates a linear hyperplane on the input space

Remember the input space is the result of some embedding...

- General Purpose Topology

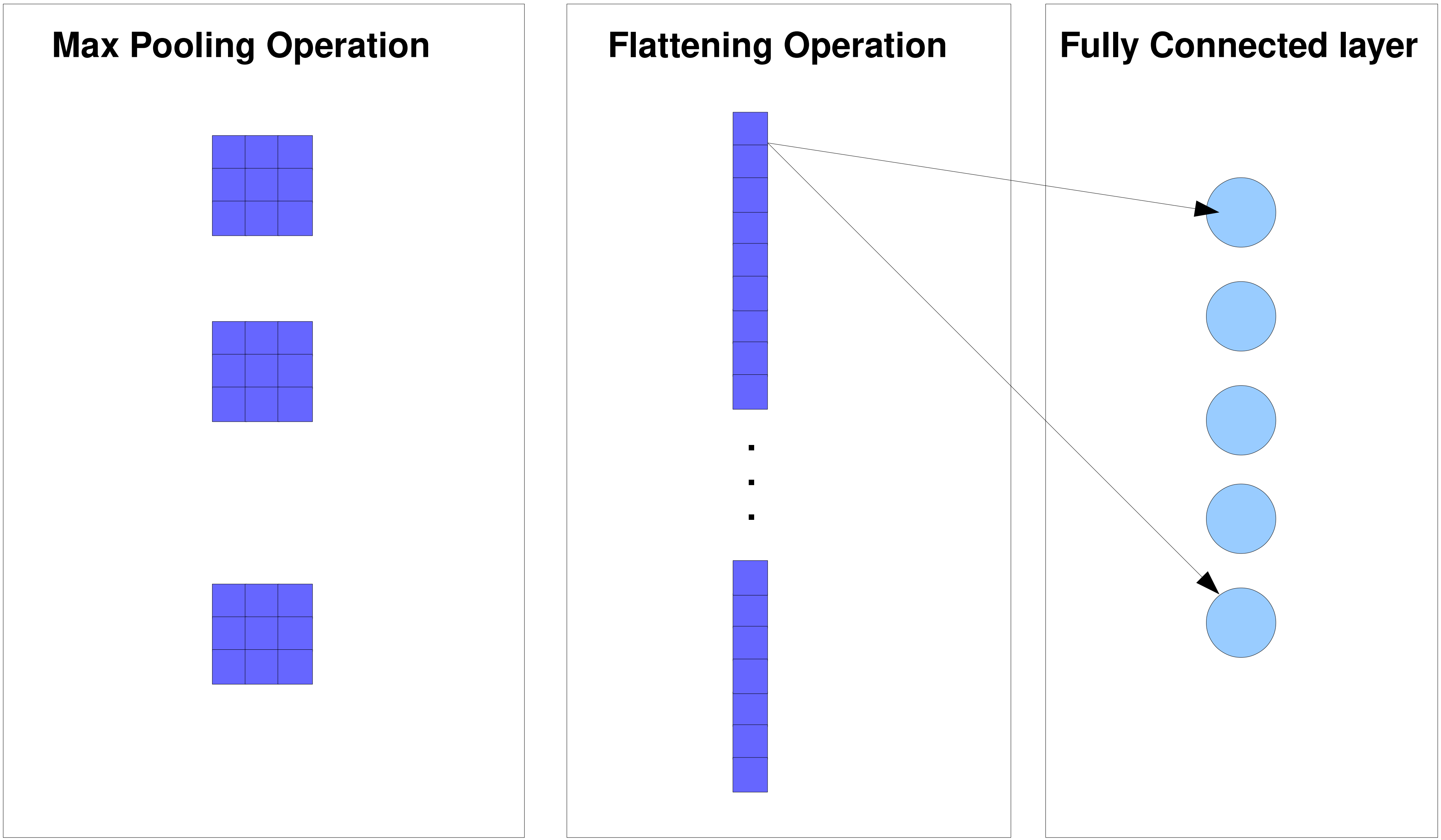


- General Purpose Topology

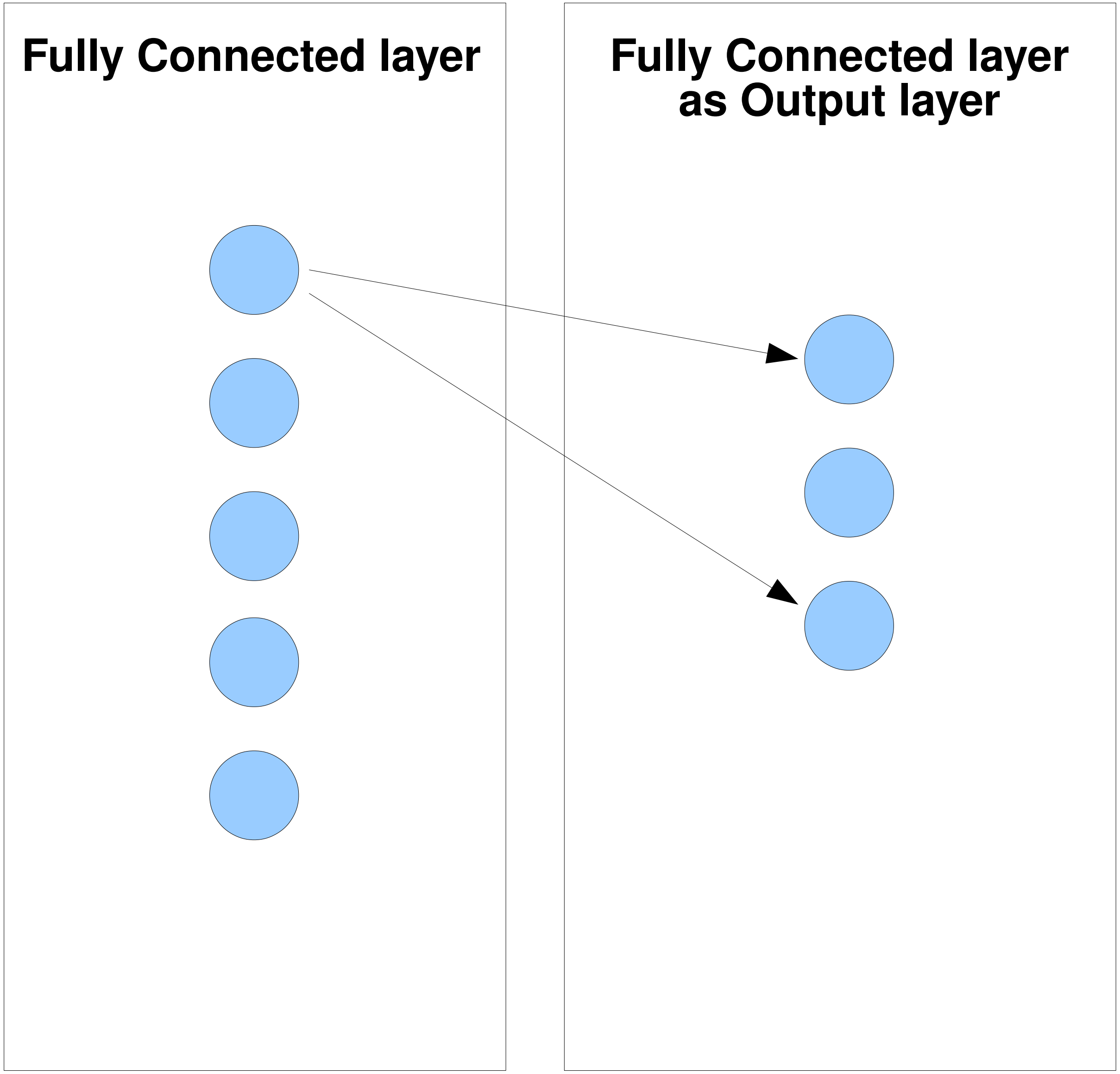


We may have more convolutional and max pooling operations

- General Purpose Topology



- General Purpose Topology



How can we train it?

- Convex Error Function
- Stochastic Gradient Descent Method
- Drop out

Some Analysis

- We can interpret CNNs as:
  - Signal Processing
    - A Bank of Convolutional Filters
  - Dynamical Systems
    - As some embedding followed by local linear shattering
      - Globally as a nonlinear operation
  - Statistical Learning
    - As the implicit design of a kernel

Introducing Deep Learning from Multilayer Perceptron

Rodrigo Fernandes de Mello

Invited Professor at Télécom ParisTech (until July 2019)

Associate Professor at Universidade de São Paulo, ICMC

mello@icmc.usp.br

February 20th, 2019