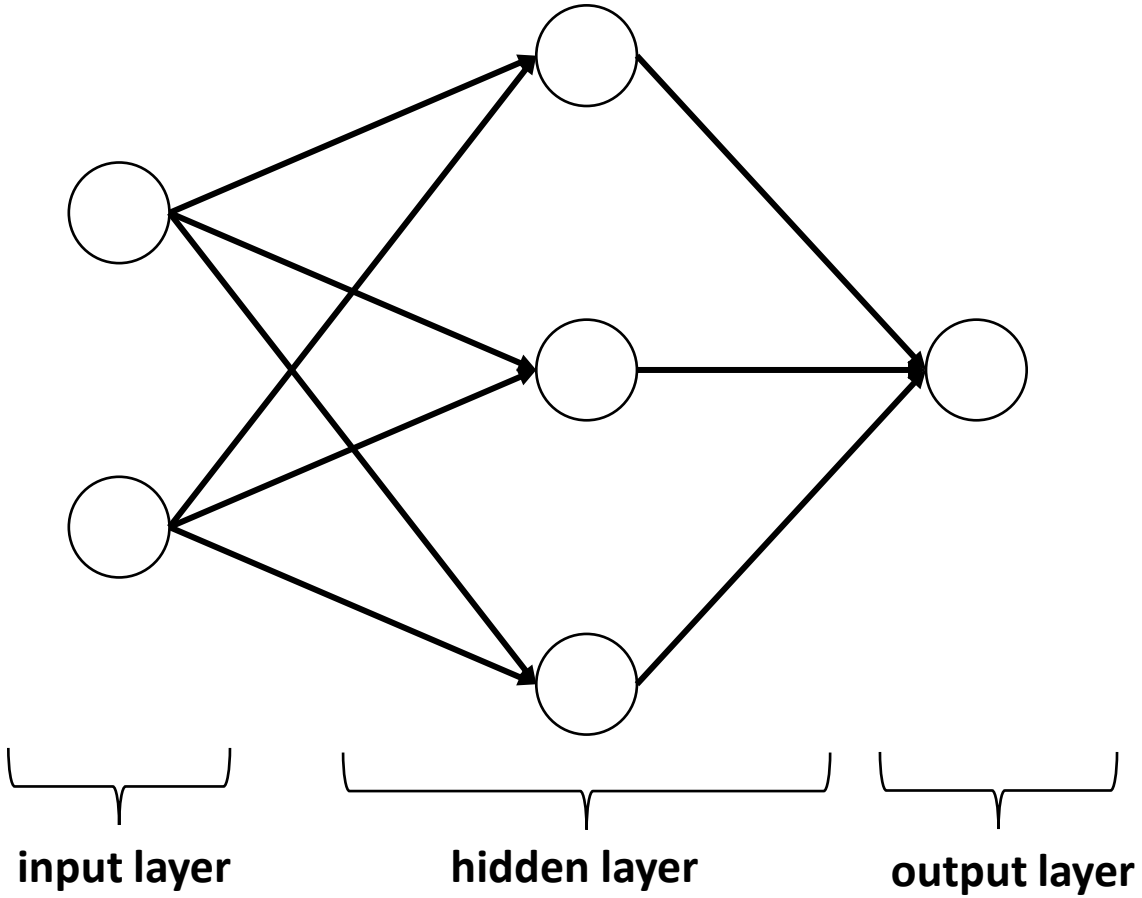
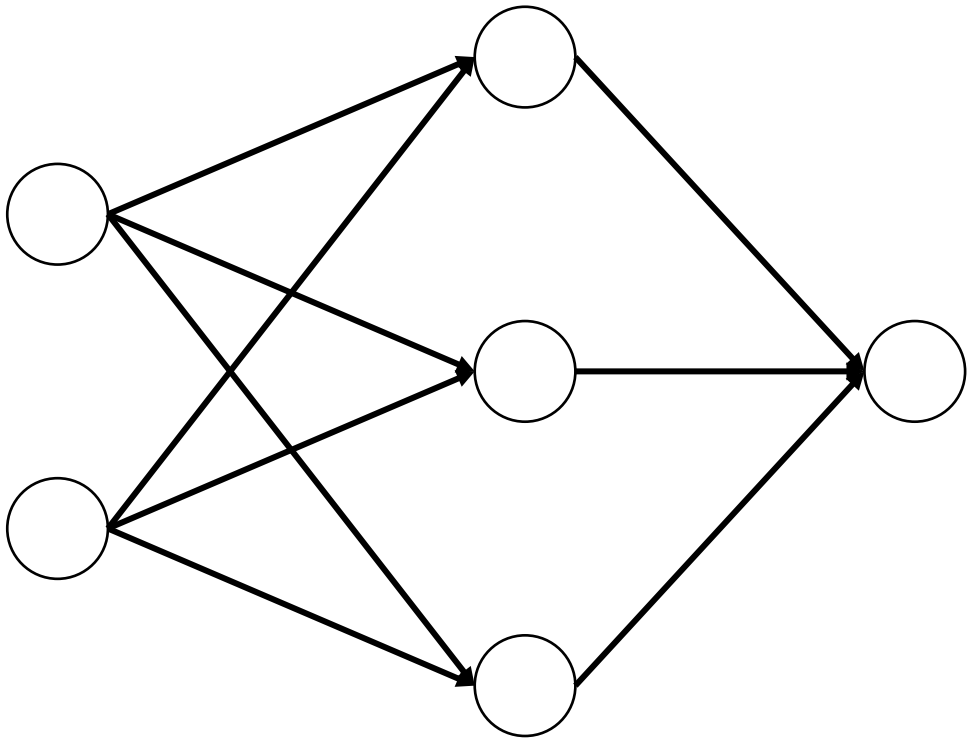
ARTIFICIAL INTELLIGENCE

DEEP NEURAL NETWORKS

FEEDFORWARD NEURAL NETWORKS



FEEDFORWARD NEURAL NETWORKS



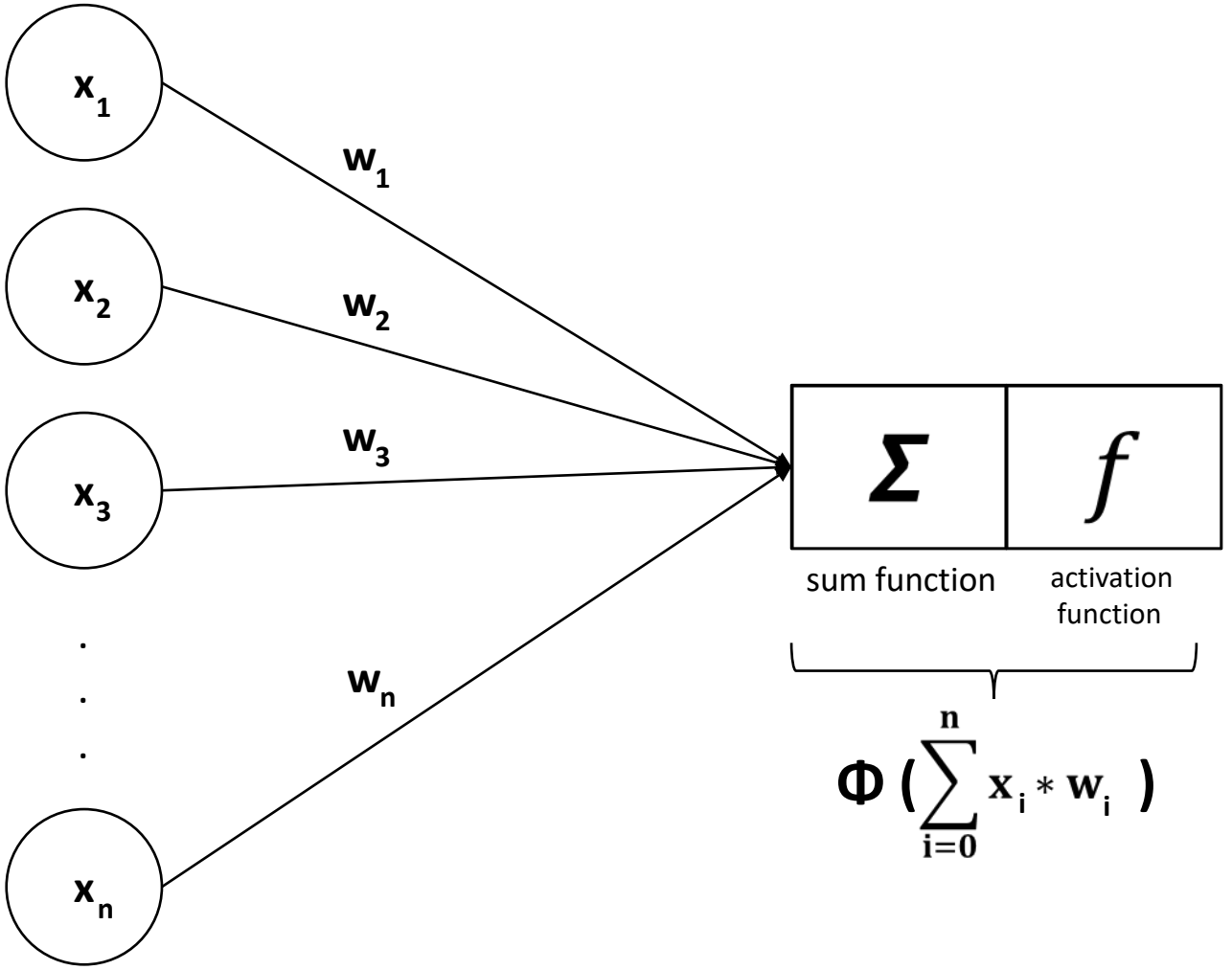
→ every node is connected to every node in the next layer
(there are lots of weights in such a network)

→ a little change in the **w** edge weights results in change in the output !!!

→ **TRAINING A NETWORK**: means adjusting the edge weights in the layers

→ there are several hyperparameters we can tune
For example: learning rate, momentum ...

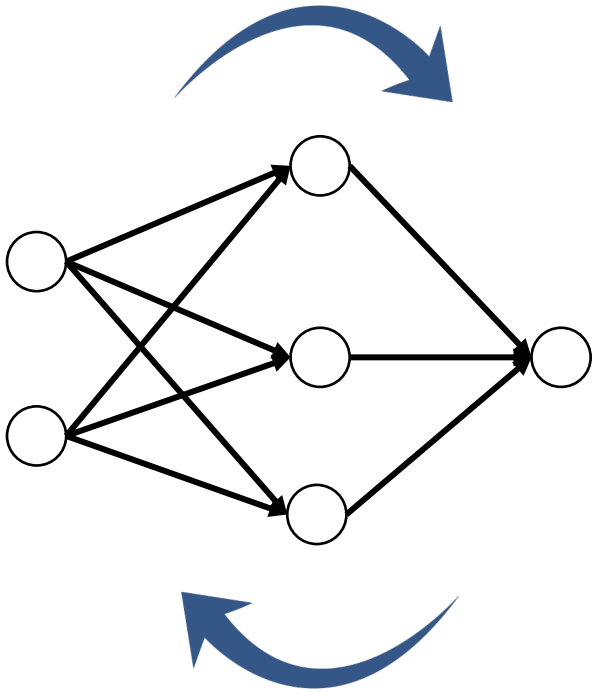
FEEDFORWARD NEURAL NETWORKS



FEEDFORWARD NEURAL NETWORKS

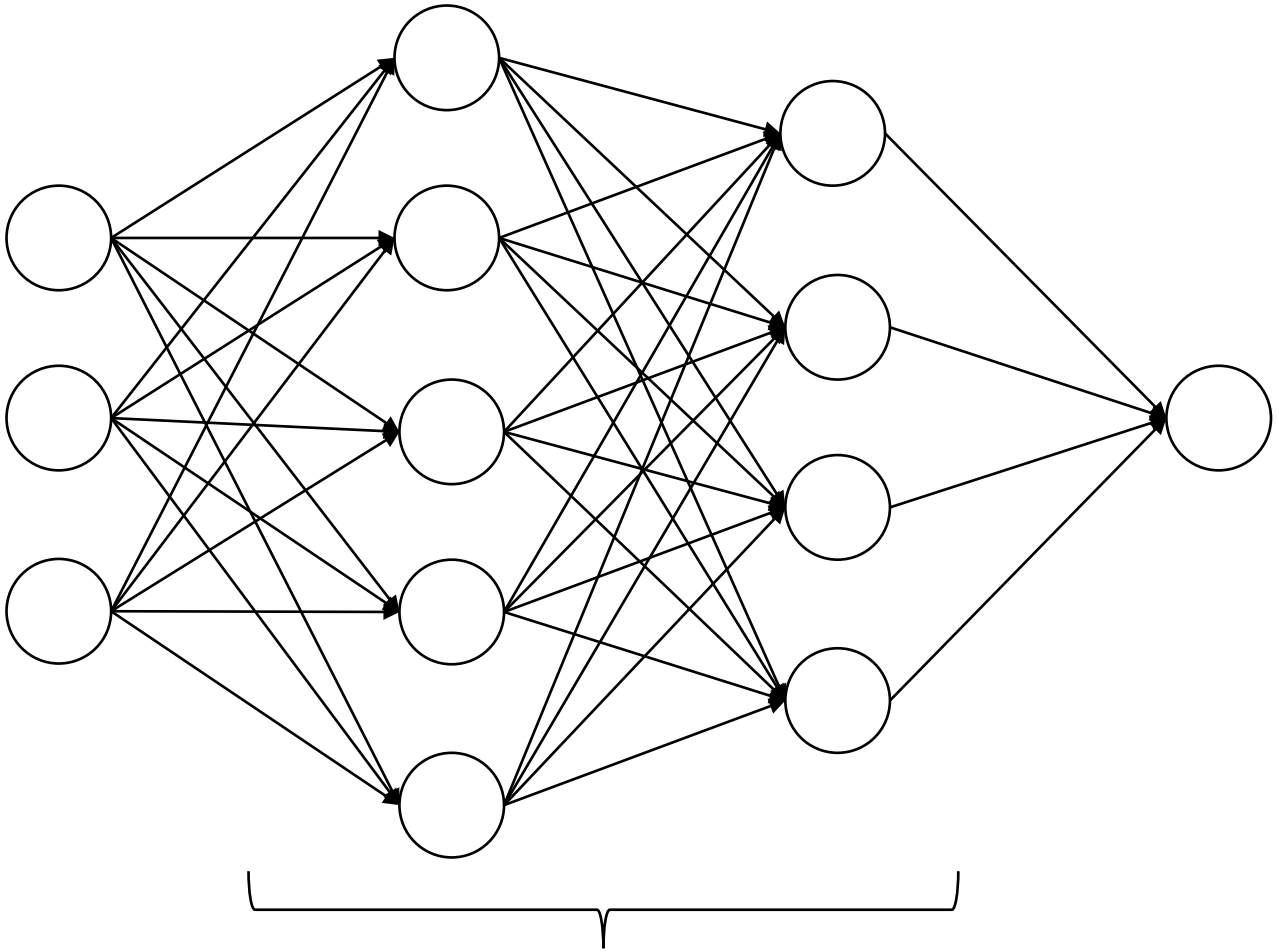
x	y	x **XOR** y
0	0	0
0	1	1
1	0	1
1	1	0

we feed the data to the network and check the output accordingly



we change the edge weights according to the error „backpropagation”

DEEP NEURAL NETWORKS



Deep learning means we have several hidden layers: usually **5-10** hidden layers

~ other problems may arise because of this

ACTIVATION FUNCTIONS

First of all, why do we need activation functions?

THE ROLE OF ACTIVATION FUNCTIONS IS TO MAKE NEURAL NETWORKS NON-LINEAR !!!

→ without the activation function the network is just a linear transformation, which is not strong enough to many kinds of data.

→ we could add more parameters to the model instead of using activation functions
Not a good idea: the training would be slower ...

1.) linear activation function

2.) sigmoid activation function

3.) tanh activation function

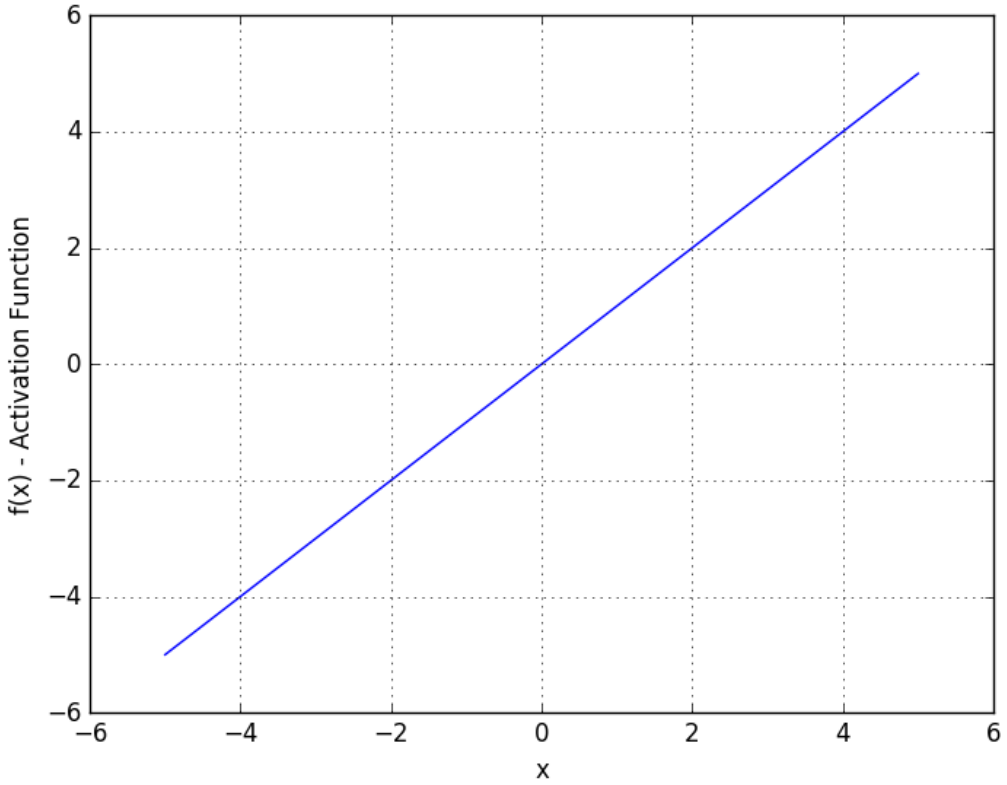
4.) ReLU activation function

5.) leaky ReLU activation function

ACTIVATION FUNCTIONS

First of all, why do we need activation functions?

THE ROLE OF ACTIVATION FUNCTIONS IS TO MAKE NEURAL NETWORKS NON-LINEAR !!!



This is a linear activation function **f(x) = x**

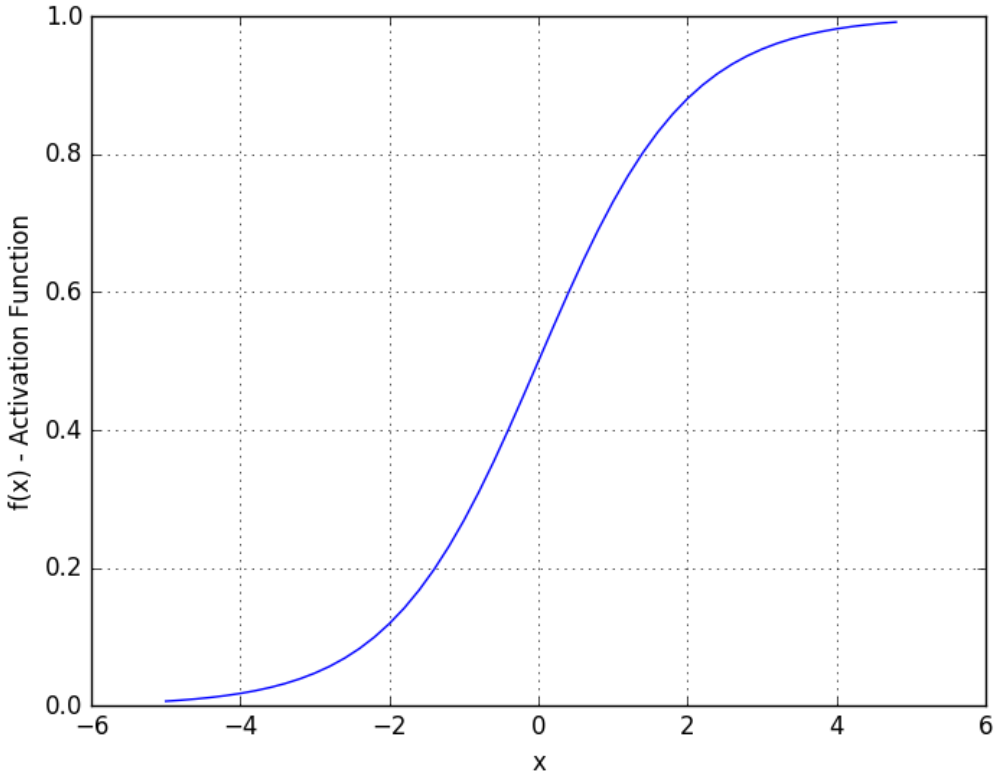
~ it is the identity operator basically: it means the function passes the signal through unchanged

~ usually we do not change the input when dealing with the input layer: so we can say that the input layer has linear activation function !!!

ACTIVATION FUNCTIONS

First of all, why do we need activation functions?

THE ROLE OF ACTIVATION FUNCTIONS IS TO MAKE NEURAL NETWORKS NON-LINEAR !!!



We like sigmoid transformation because it reduce extreme values and outliers in the data without removing them !!!

→ sigmoid function transforms the data in the range **[0,1]**

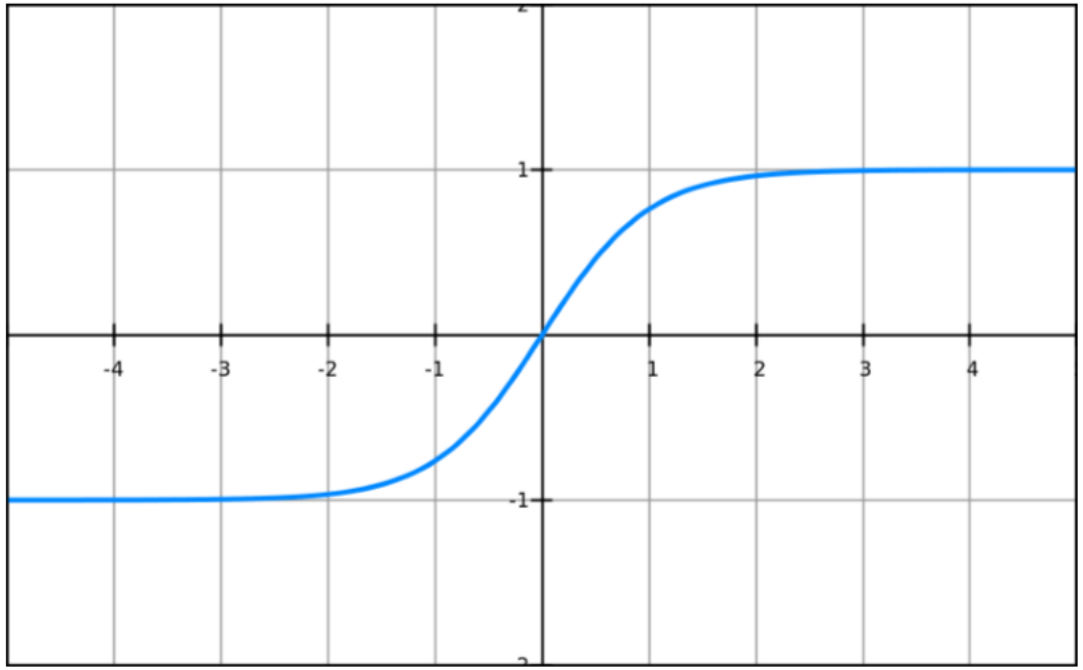
→ we can interpret the results as probabilities

→ sigmoid activation function outputs an independent probability for each class

ACTIVATION FUNCTIONS

First of all, why do we need activation functions?

THE ROLE OF ACTIVATION FUNCTIONS IS TO MAKE NEURAL NETWORKS NON-LINEAR !!!



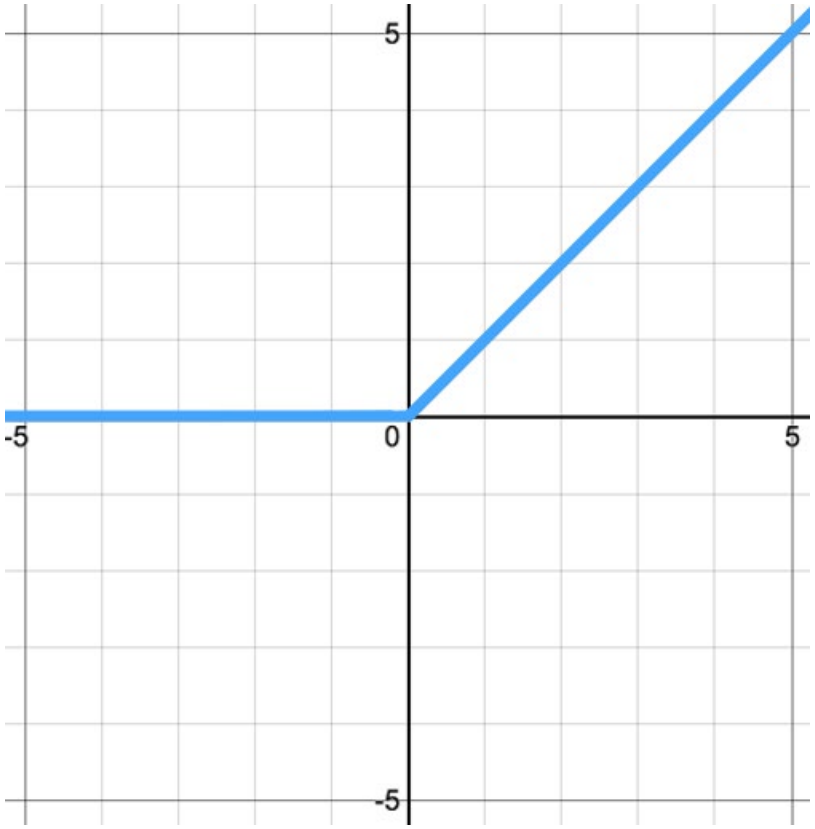
This activation function is very similar to the sigmoid function but the range of tanh is **[-1,1]**

~ it can handle negative values as well !!!

ACTIVATION FUNCTIONS

First of all, why do we need activation functions?

THE ROLE OF ACTIVATION FUNCTIONS IS TO MAKE NEURAL NETWORKS NON-LINEAR !!!



The rectified linear (ReLU) activation function activates a node only if the input is above a certain quantity

f(x) = max(0,x) ReLU activation function

→ this is the most popular function: the gradient is either zero or a constant, so it can solve the the vanishing gradient issue !!!

→ training procedure relies on the derivative of the activation function. Each of the neural network's weights receives an update proportional to the gradient of the error function with respect to the current weight in each iteration of training.

The problem is that in some cases, the gradient will be vanishingly small, effectively preventing the weight from changing its value. In the worst case, this may completely stop the neural network from further training

ACTIVATION FUNCTIONS

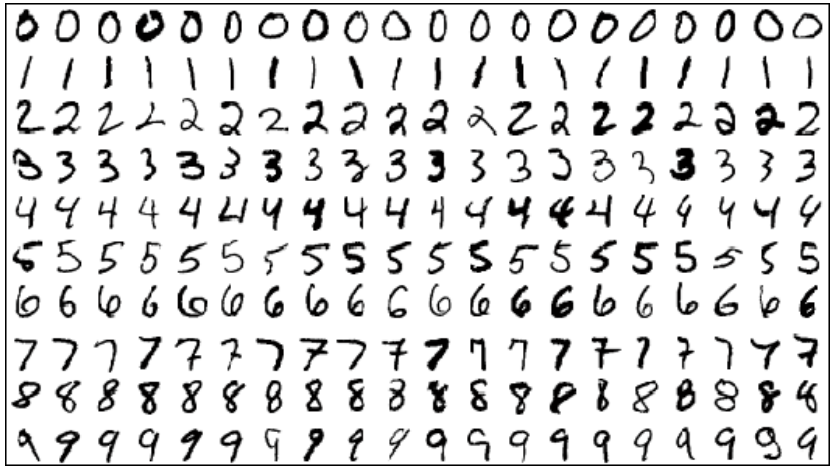
Softmax function:

In mathematics, the softmax function (normalized exponential function) is a generalization of the logistic function

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{k} e^{z_k}} \; for\; j = 1, \ldots, k$$

→ transforms the values in the range **[0,1]** that add up to **1**

→ the softmax function is used in various multiclass classification methods
For example: when we classify digits ...



We use the softmax activation function in the last layer because we want to classify handwritten digits
~ we choose the class with the highest probability

LOSS FUNCTION

We have a dataset: we want to make sure the predictions made by the network is approximately the same as the labels in the dataset

→ **loss function**: measures how close the given neural network is to the ideal toward which it is training

~ we can calculate a value based on the error we observe in the network's predictions

So we want to find the optimal bias values and weights that will minimize the loss function (we use gradient descent algorithm for the optimization)

OPTIMIZATION PROBLEM

LOSS FUNCTION

Mean Squared Error (MSE)

We use mean squared error when we are dealing with regression (so we have only one output feature), so the output is a real value

$$L(\underline{w}) = \frac{1}{N} \sum_{i=1}^{N} ||y_i - y_i'||^2$$

„cost function"

actual value that we know from our training dataset

prediction made by our artificial neural network

LOSS FUNCTION

Negative Log Likelihood

We usually use negative log likelihood loss function when dealing with classification
~ so in this case there are several output values: digits when we classify **MNIST** dataset

→ the logarithm function is monotonically increasing. So minimizing the negative log likelihood is the same as maximizing the probability

For example: Python optimize() method finds the minimum not the maximum of the given function

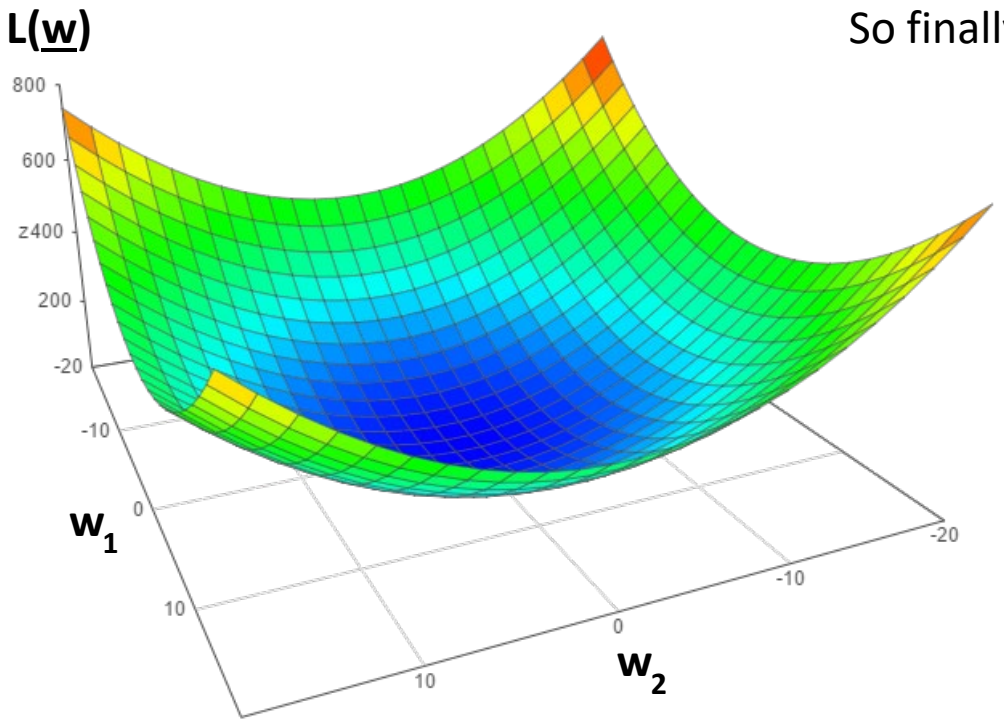
this is mathematically equivalent to what is called the cross-entropy (cross-entropy has something to do with information theory)

$$L(\underline{w}) = -\sum_{i=1}^{N} \sum_{j=1}^{M} y_{i,j} * \log y'_{i,j}$$

M: number of classes (**10** for handwritten digit classification)

N: number of samples in the dataset

GRADIENT DESCENT



this „landscape” (the loss function) has as many dimensions as the number of weights

So finally we have to make some optimization in order to reduce the error as much as possible

we change the **w** weights
(this is how we train the network)

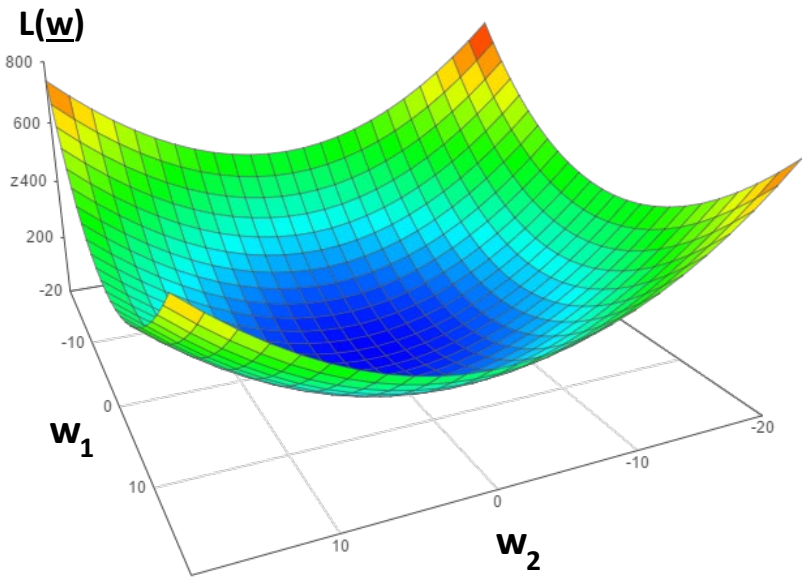
↓

the **L(w)** loss function will change as well

↓

we are after the **w** weights that minimize the **L(w)** loss function

GRADIENT DESCENT



GRADIENT DESCENT ALGORITHM

„higher" regions: the networks makes lots of mistakes, because the **L(w)** loss function's values are big

„lower" regions: network is making good predictions, because the **L(w)** loss function's values are small

SO WE HAVE TO FIND THE LOWER REGIONS !!!

We initialize the **w** weights at random at the begining: so we start from a given point in this „landscape"
with the given **w** values and **L(w)** value
~ the negative gradient is pointing in the direction of the lowest point ... just follow the gradient

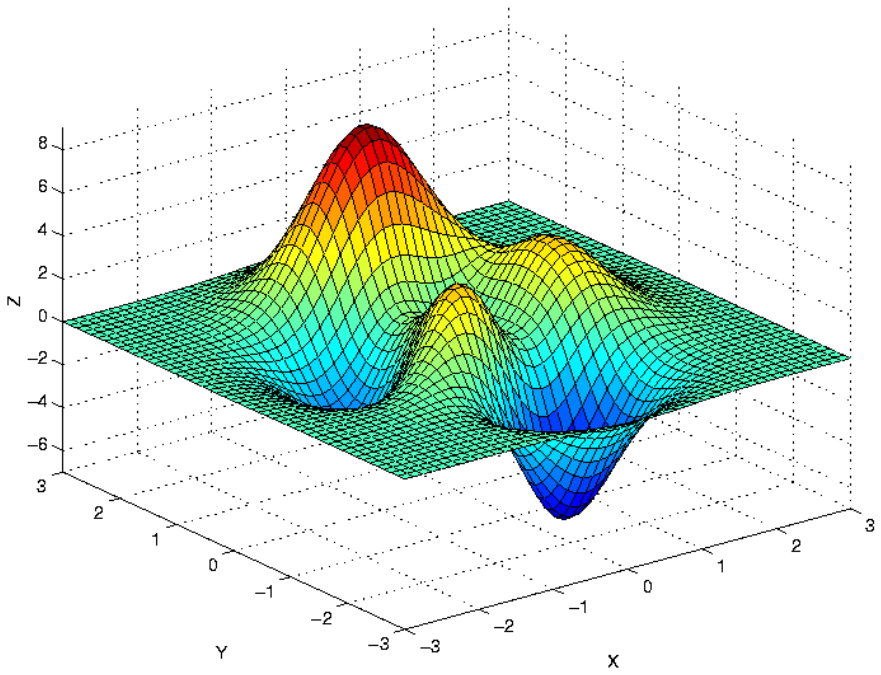
repeate until convergence {

$$w_j = w_j - \alpha \frac{\partial L(w)}{\partial w_j}$$

α: learning rate

}

GRADIENT DESCENT



Gradient descent is working fine for convex loss functions
BUT usually this is not the case !!!

~ if gradient descent reaches a local minimum, it is effectively trapped
and this is one drawback of the algorithm

What to do? (genetic algorithms or simulated annealing)

$$X \longrightarrow \frac{x - \min(x)}{\max(x) - \min(x)}$$

By the way normalizing the original dataset is usually helpful when dealing with machine learning
or artificial neural networks (**min-max normalization** or z-score normalization)
~ normalization makes sure gradient descent will converge faster and more accurately

STOCHASTIC GRADIENT DESCENT

Gradient descent → we calculate the overall loss across all of the training dataset
and then we calculate the gradient

Stochastic gradient descent → we compute the gradient and parameter vector update after every single training sample
(if you use a subset of the original training dataset, it is called minibatch stochastic gradient descent)

GRADIENT DESCENT

→ slower convergence

→ hence more accurate

→ deterministic: converges to the same minimum

STOCHASTIC GRADIENT DESCENT

→ faster convergence (uses less data)

→ hence not that accurate

→ stochastic: not always converges to the same minimum

HYPERPARAMETERS

We can tune our neural networks with different parameters
For example: learning rate, momentum ...

→ we want to avoid overfitting as well as underfitting

→ but we want to make sure the algorithm is capable of learning the structure of the data as quickly as possible

1.) LEARNING RATE:

This is how we define the pace of learning: the coefficient when we are dealing with the gradient

$$w_j = w_j - \alpha \frac{\partial L(w)}{\partial w_j}$$

$\alpha > 1$ if the learning rate is too high: the training will be fast but it may miss the optimum

$\alpha < 0.01$ if the learning rate is too small: the training will be very slow but it will find the optimum (local or global optimum)

HYPERPARAMETERS

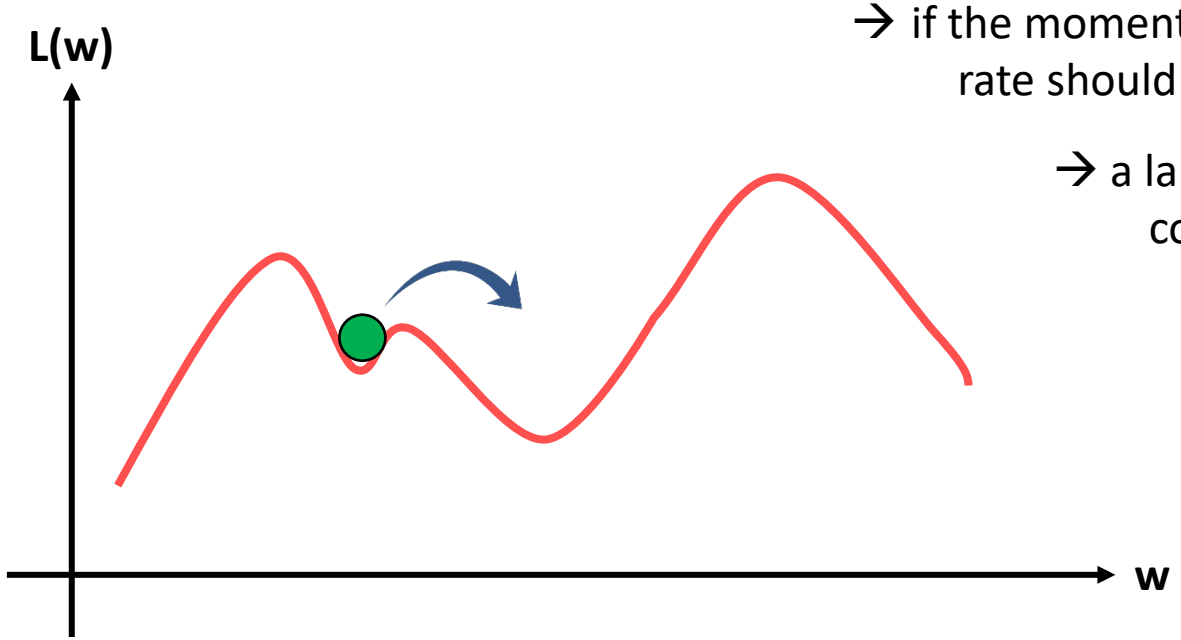
2.) MOMENTUM:

Momentum helps the learning algorithm get out of spots in the search space where it would otherwise become stuck

→ is a value between **[0,1]** that increases the size of the steps taken towards the minimum by trying to jump from a local minima

→ if the momentum term is large then the learning rate should be kept smaller

→ a large value of momentum also means that the convergence will happen fast



HYPERPARAMETERS

REGULARIZATION:

Why to use regularization? This is how we can control overfitting in machine learning as well as in artificial intelligence

DROPOUT is an inexpensive regularization method

→ dropout means we set the activation of a given neuron to be **0** temporarily

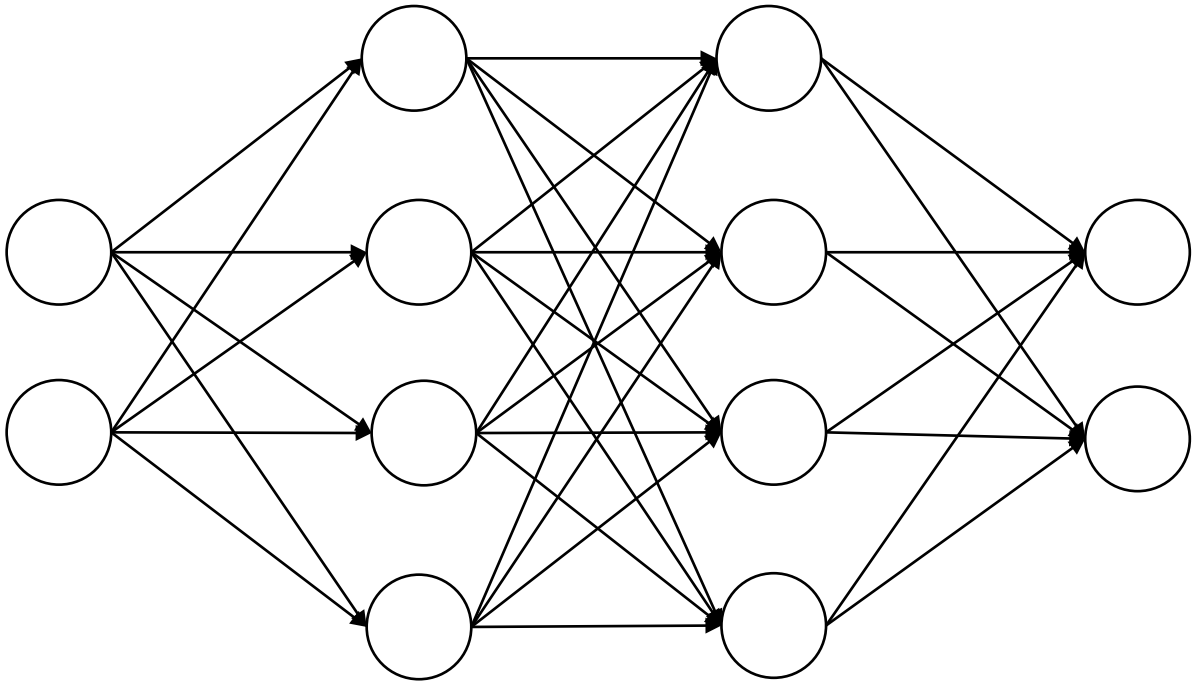
→ works well with stochastic gradient descent method

→ usually we apply dropout in the hidden layer exclusively
We simply omit neurons with **p** (~0.5) probability !!!

We can prevent coadaptation among detectors, which helps drive better generalization in given models

// less effective as the number of training records rises up (> tens of millions)

DEEP NEURAL NETWORKS – XOR PROBLEM



x	y	x **XOR** y
0	0	(1,0)
0	1	(0,1)
1	0	(0,1)
1	1	(1,0)

~ usually we use as many digits for the output when dealing with classification as the number of classes

In this case: **0** and **1** are the classes but we use the **(1,0)** to represent the first class and **(0,1)** to represent the other

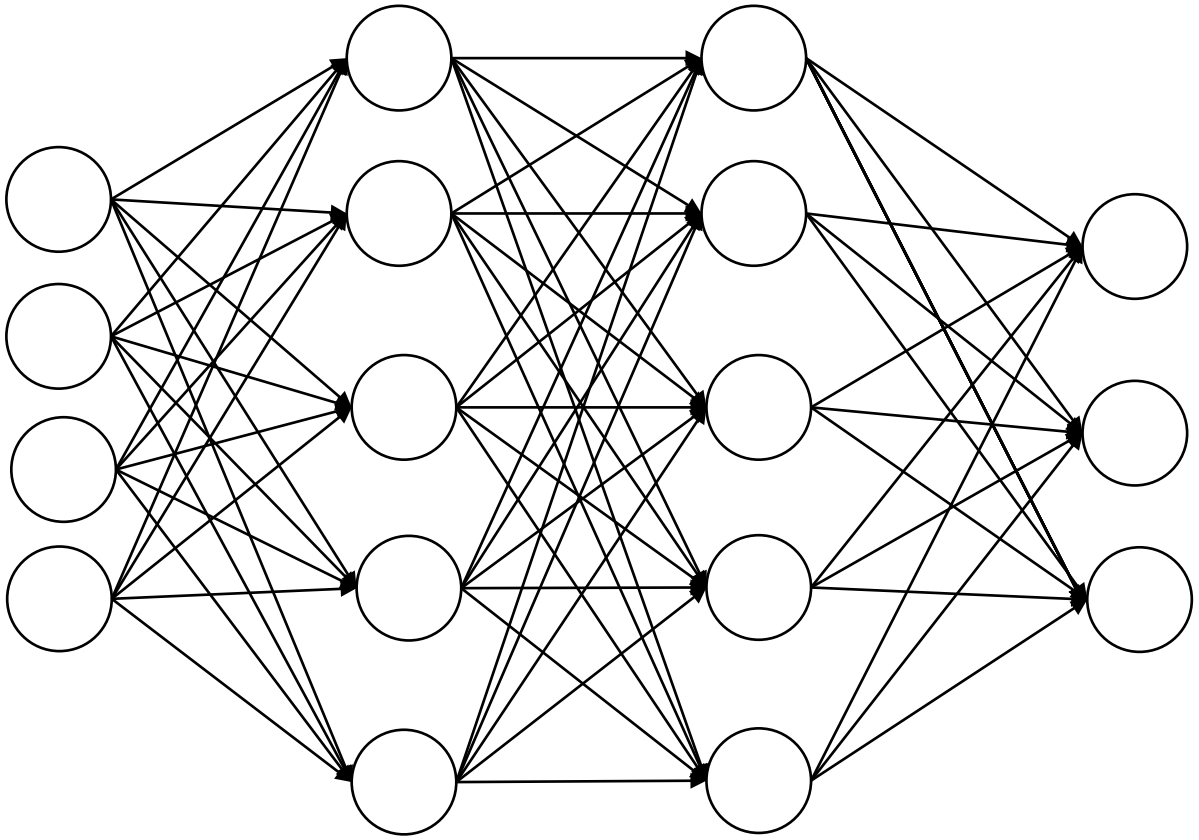
DEEP NEURAL NETWORKS

If we are dealing with handwritten digit classification: we have **10** classes

~ so again we represent the classes in binary

0	→	**(1 0 0 0 0 0 0 0 0 0)**
1	→	**(0 1 0 0 0 0 0 0 0 0)**
2	→	**(0 0 1 0 0 0 0 0 0 0)**
3	→	**(0 0 0 1 0 0 0 0 0 0)**
4	→	**(0 0 0 0 1 0 0 0 0 0)**
5	→	**(0 0 0 0 0 1 0 0 0 0)**
6	→	**(0 0 0 0 0 0 1 0 0 0)**
7	→	**(0 0 0 0 0 0 0 1 0 0)**
8	→	**(0 0 0 0 0 0 0 0 1 0)**
9	→	**(0 0 0 0 0 0 0 0 0 1)**

DEEP NEURAL NETWORKS – IRIS DATASET



x	y	z	u	classes
.	.	.	.	(1,0,0)
.	.	.	.	
.	.	.	.	(0,0,1)
.	.	.	.	
				(0,1,0)

~ usually we use as many digits for the output when dealing with classification as the number of classes

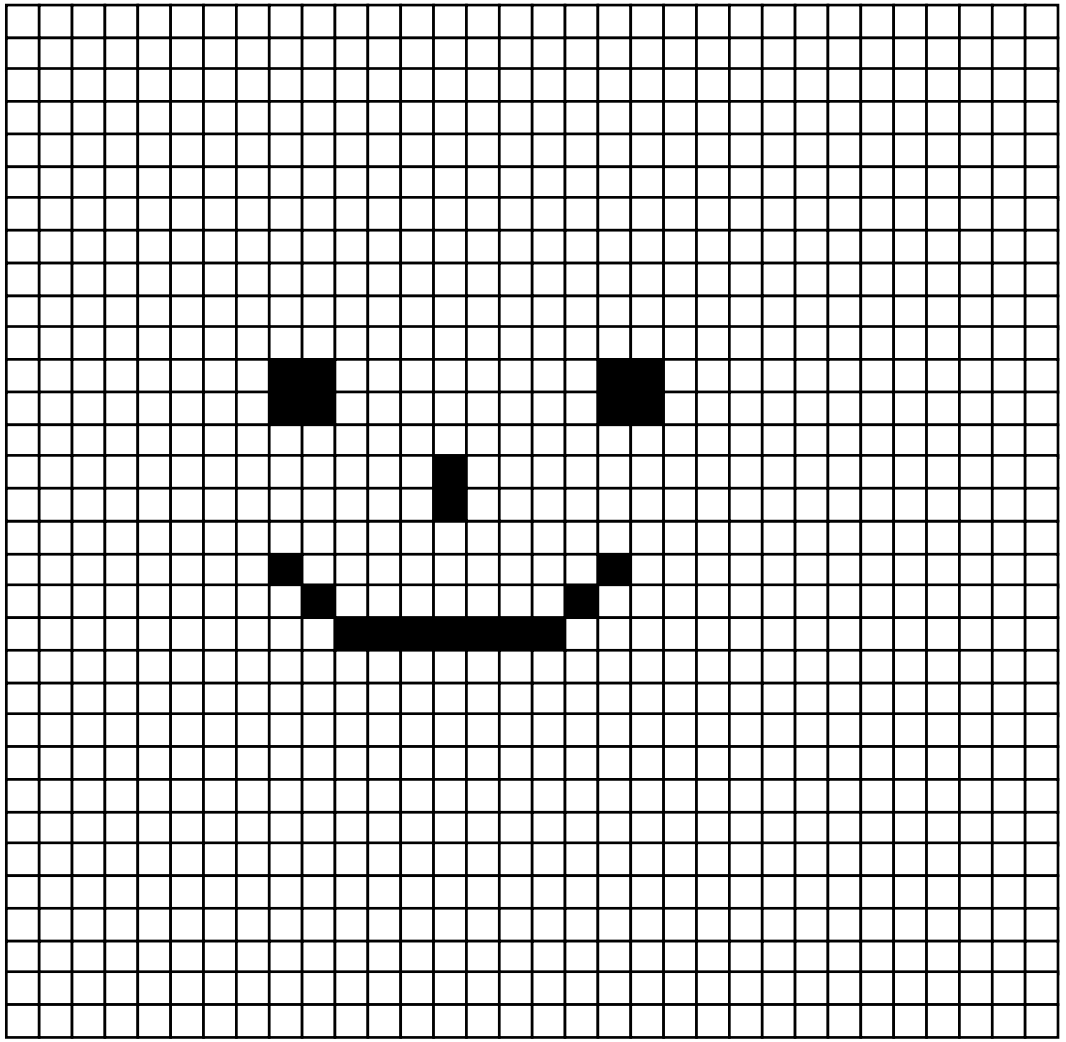
In this case there are three classes: **(1,0,0)**, **(0,1,0)** and **(0,0,1)** because there are 3 flowers in the dataset !!!

CONVOLUTIONAL NEURAL NETWORKS

Dense neural networks are working fine ...

But if there are 1000 neurons in each layer, the number of weights increase dramatically

~ training the network will be extremely slow !!!



If we have a **32x32** image it means we have **1024** pixels all together (we should consider colors as well)

→ there will be **1000s** of connections and weights to train in the network „combinatorial explosion”

→ gradient descent changes the edge weights according to the learning rate and the gradient

SLOW PROCEDURE !!!

CONVOLUTIONAL NEURAL NETWORKS

Convolutional neural networks have an assumption: the inputs are images
So we can encode certain properties into the architecture

→ **NEURONS ARE NOT CONNECTED TO EVERY NEURON IN THE NEXT LAYER**

→ under the hood it uses a standard neural network but at the beginning
it transforms the data in order to achieve the best accuracy possible

→ self driving cars or pedestrian detection have something to do
with convolutional neural networks
~ convolutional neural networks outperform machine learning
techniques such as **SVM**

There are 3 important steps:

1.) convolutional operation

2.) pooling

3.) flattening

CONVOLUTIONAL NEURAL NETWORKS

Convolutional operation

$$(f \bullet g)(t) = \int_{-\infty}^{+\infty} f(u)g(t-u)du$$

In image processing, convolution is the process of adding each element of the image to its local neigbors, weighted by the kernel

image → matrix representation
kernel (feature detector or filter) → another matrix

Convolution: matrix operation ... we have to multiply values

$$f(x,y) \bullet g(x,y) = \sum_{n=0}^{cols} \sum_{m=0}^{rows} g(n,m) * f(x-n, y-m)$$

CONVOLUTIONAL NEURAL NETWORKS

What is the main problem in machine learning and AI?

FEATURE SELECTION !!!

→ what features to use to build our model?

For example: stock prices, credit scoring, animal classification ...

What features to use in order to recognize faces?

- Location of nose...
- Distance between eyes...
- Eyebrows...
- Location of mouth...

What features to use in order to recognize a cheetah?

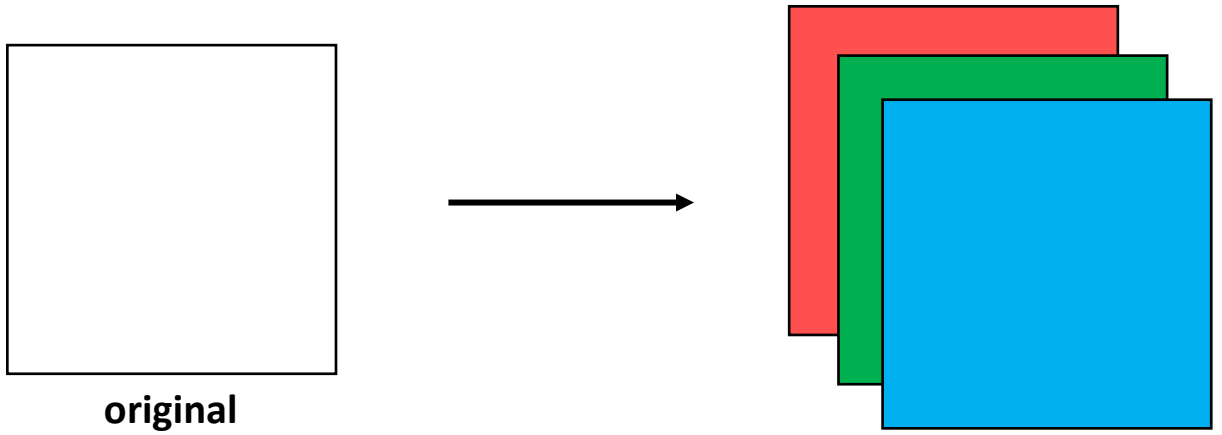
- Shape of ears...
- Black pattern under eyes...
- Nose...

CONVOLUTIONAL NEURAL NETWORKS FIND THE RELEVANT FEATURES !!!

CONVOLUTIONAL NEURAL NETWORKS

If we have colored images we have to deal with all the color channels separately

RGB colors: we decompose the image into 3 layers



original

we deal with all the layers separately

CONVOLUTIONAL NEURAL NETWORKS

Feature detector / kernel / filter:

Feature detectors are represented as matrixes

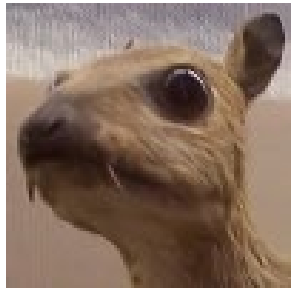
~ helps to detect the relevant features in a given image

This is the **sharpen kernel**: it makes the given image more sharp

0	**-1**	**0**
-1	**5**	**-1**
0	**-1**	**0**

→ increase the pixel intensity of the given pixel

→ reduce the pixel intensity of neigbor pixels



the original image

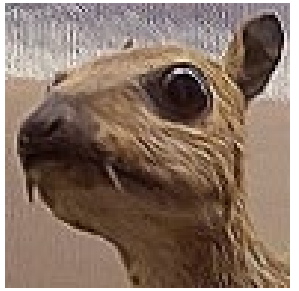


image after applying the sharpen kernel

CONVOLUTIONAL NEURAL NETWORKS

Feature detector / kernel / filter:

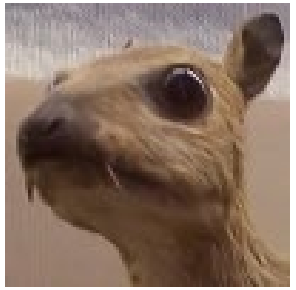
Feature detectors are represented as matrixes
~ helps to detect the relevant features in a given image

0	**1**	**0**
1	**-4**	**1**
0	**1**	**0**

This is the **edge detection kernel**: it can detect edges, sometimes this is how we can end up with relevant features

→ decrease the pixel intensity of the given pixel

→ do not change the pixel intensity of neigbor pixels



the original image

image after applying
the edge detector kernel

CONVOLUTIONAL NEURAL NETWORKS

Feature detector / kernel / filter:

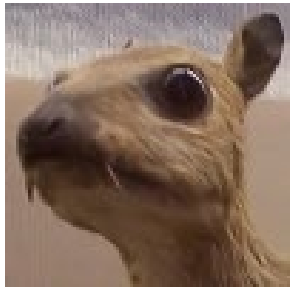
Feature detectors are represented as matrixes
~ helps to detect the relevant features in a given image

1	**1**	**1**
1	**1**	**1**
1	**1**	**1**

This is the **blur kernel**: it is not that useful so we do not use this technique in convolutional neural networks

→ do not change the pixel intensity of the given pixel

→ we use the neighbor pixel intensity values as well



the original image

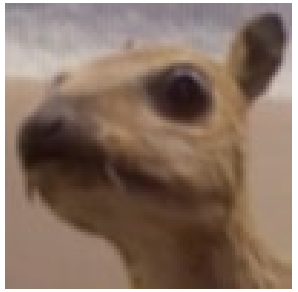


image after applying the blur kernel

CONVOLUTIONAL NEURAL NETWORKS

Feature detector / kernel / filter:

Every single kernel will use a specific feature of the image

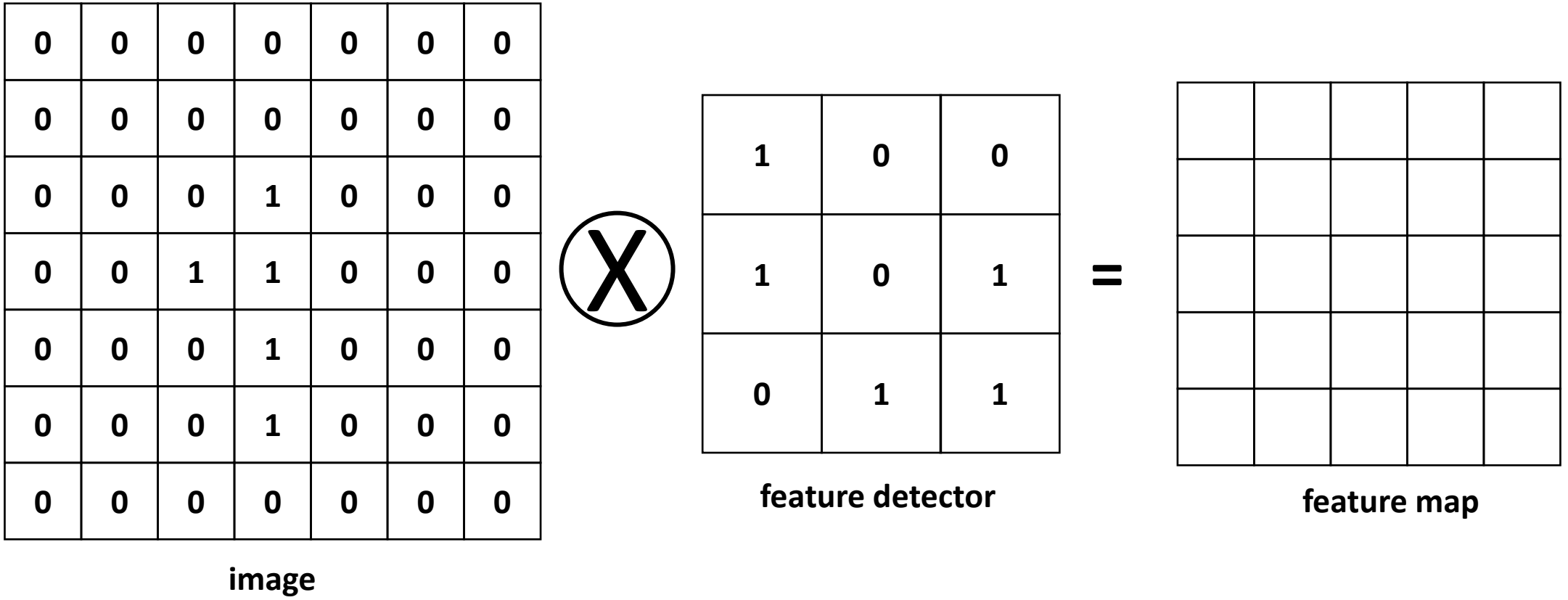
When using edge detector, we assume the edges are important

BUT HOW TO DECIDE WHAT FEATURE DETECTOR TO USE?

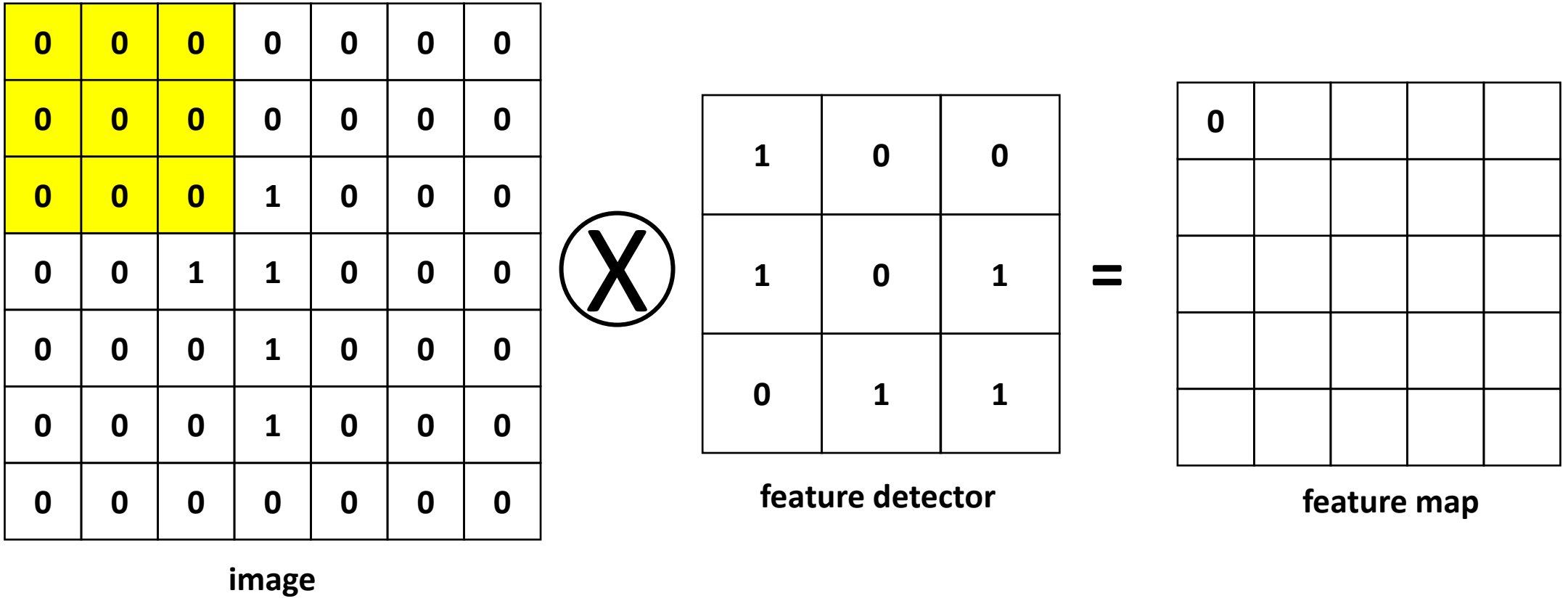
→ no need to decide in advance !!!

→ convolutional neural network uses many kernels and during the training procedure it eventually selects the best possible

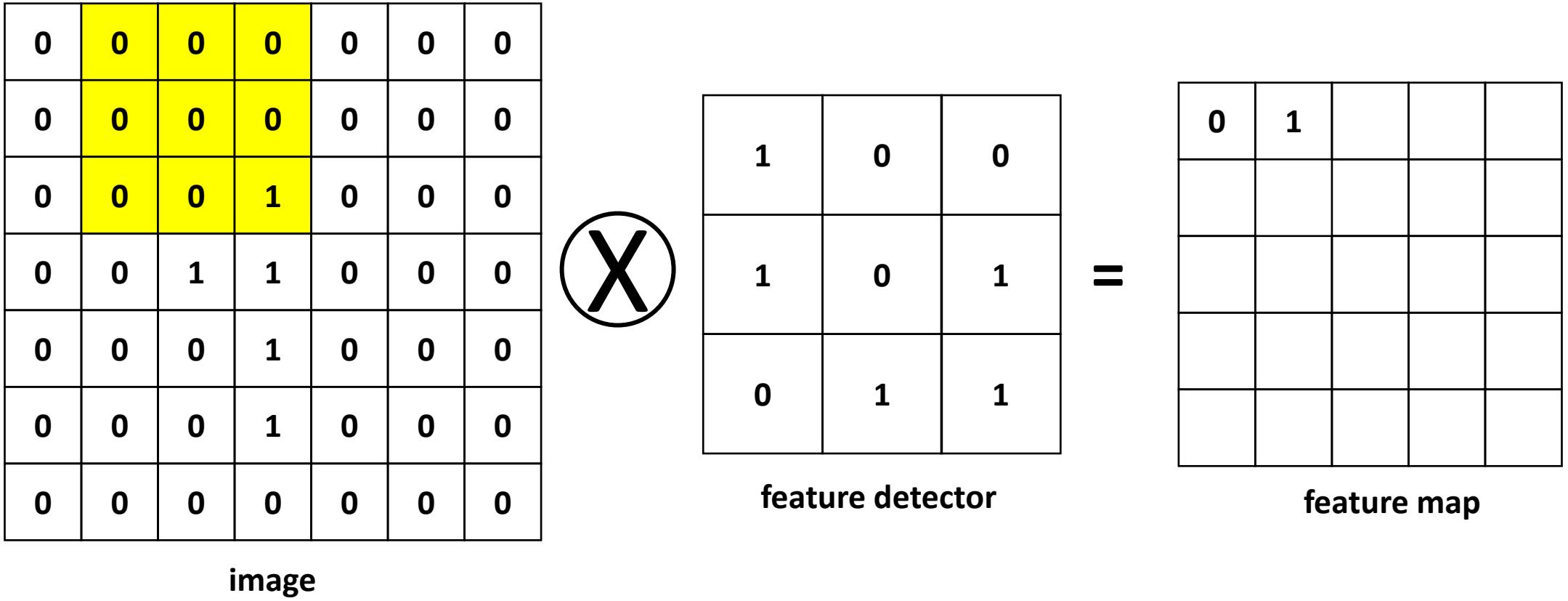
CONVOLUTIONAL NEURAL NETWORKS



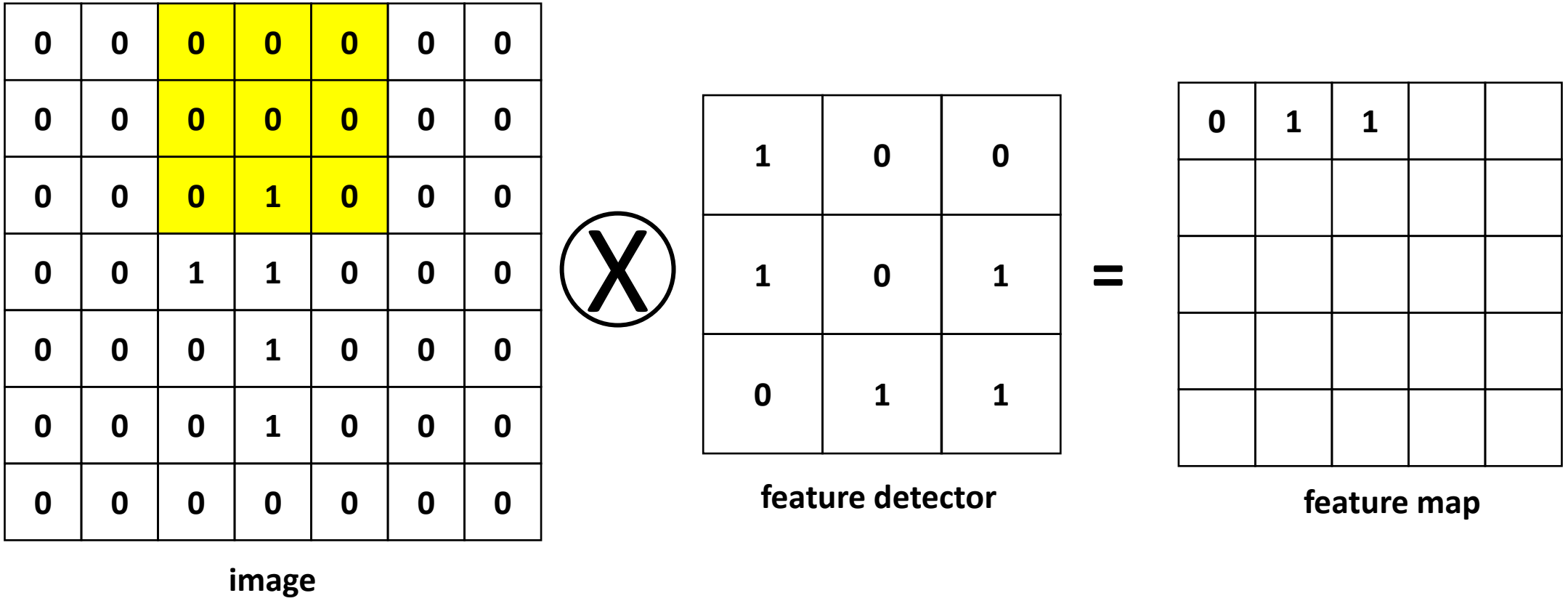
CONVOLUTIONAL NEURAL NETWORKS



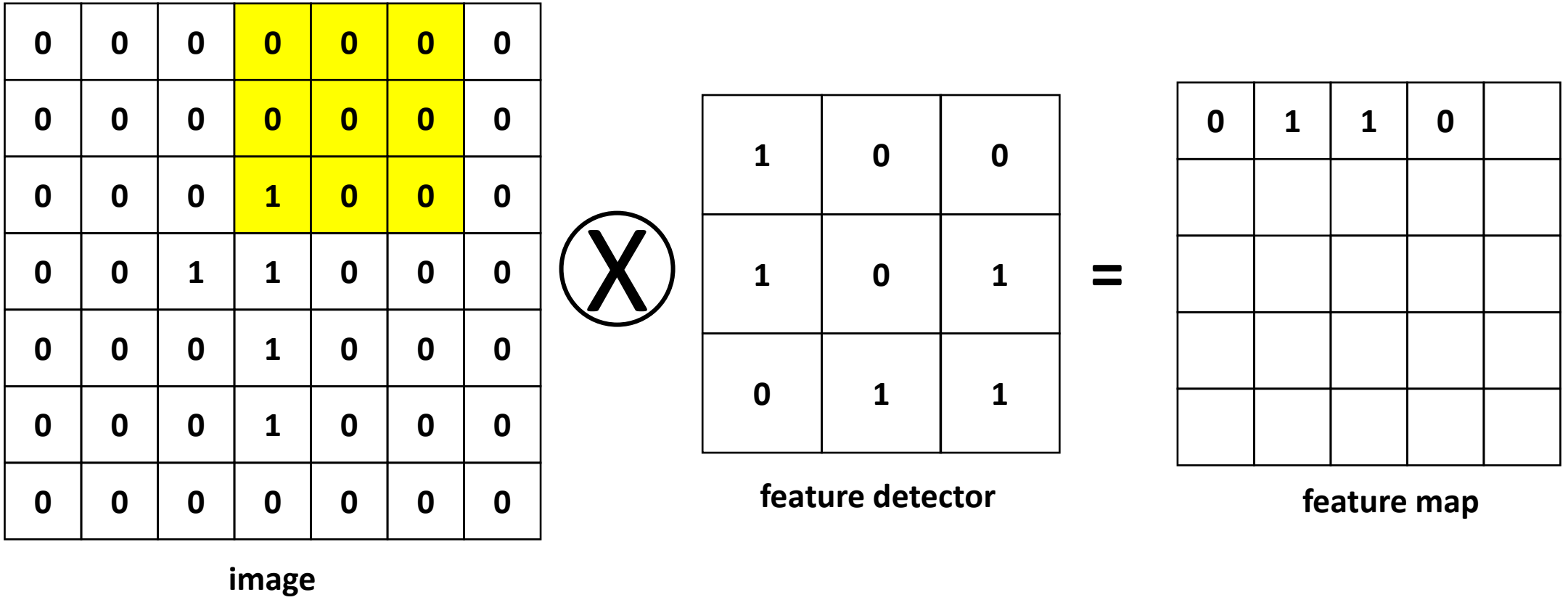
CONVOLUTIONAL NEURAL NETWORKS



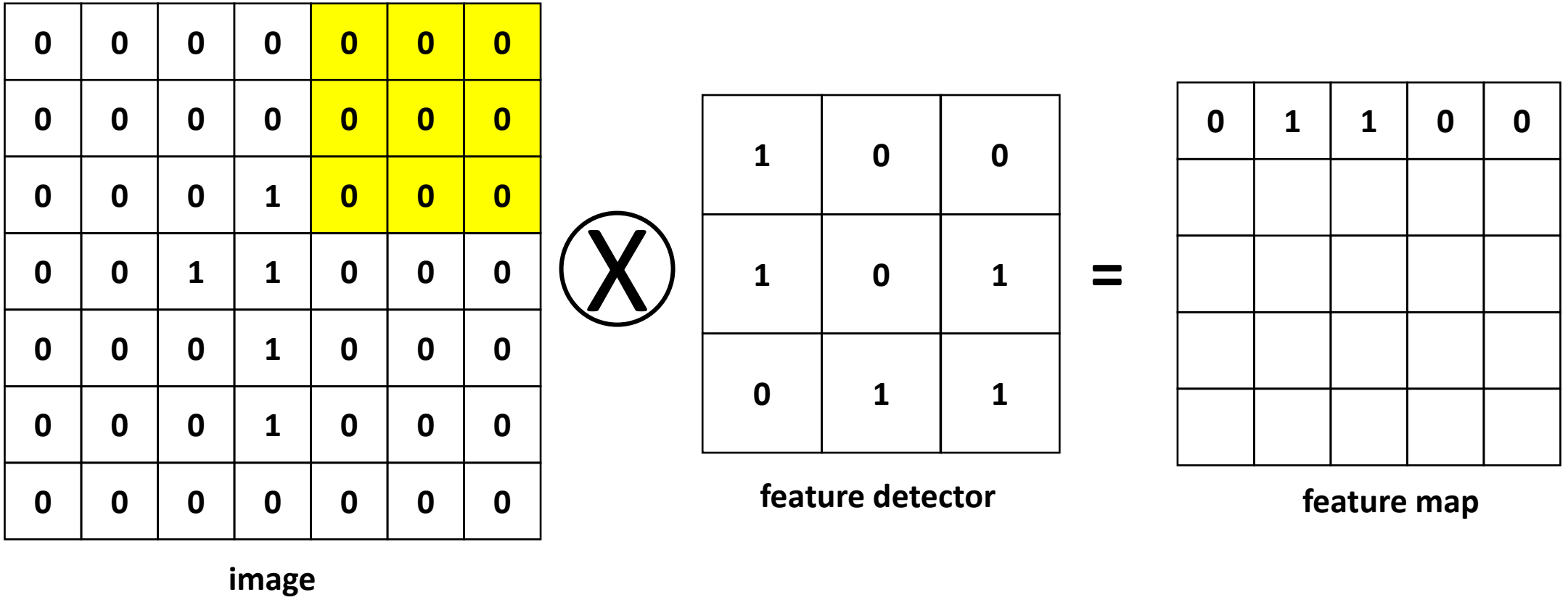
CONVOLUTIONAL NEURAL NETWORKS



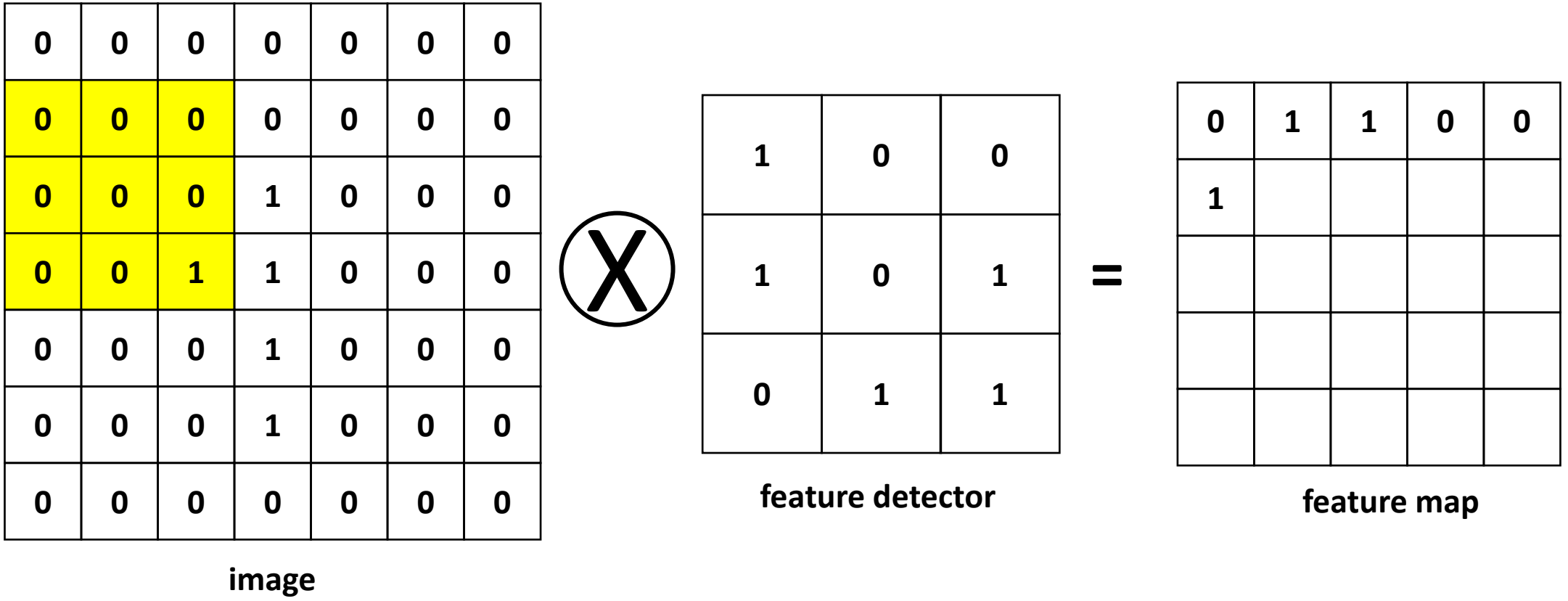
CONVOLUTIONAL NEURAL NETWORKS



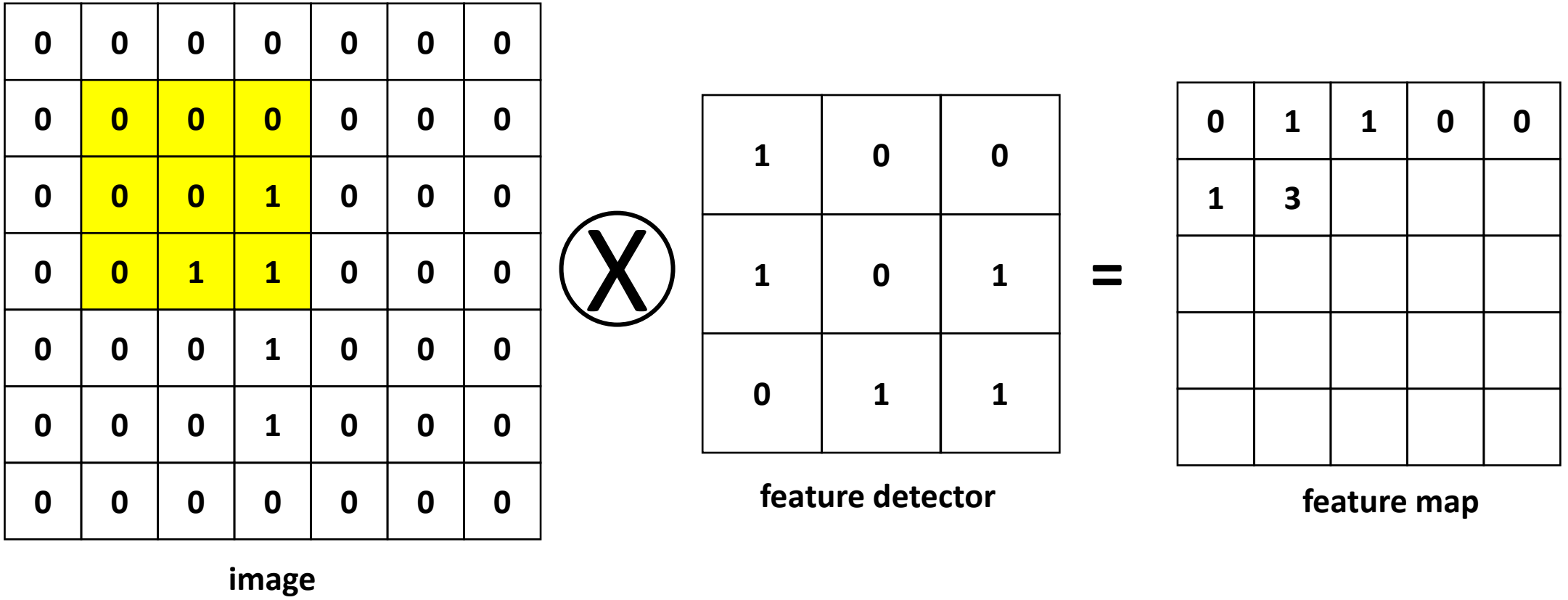
CONVOLUTIONAL NEURAL NETWORKS



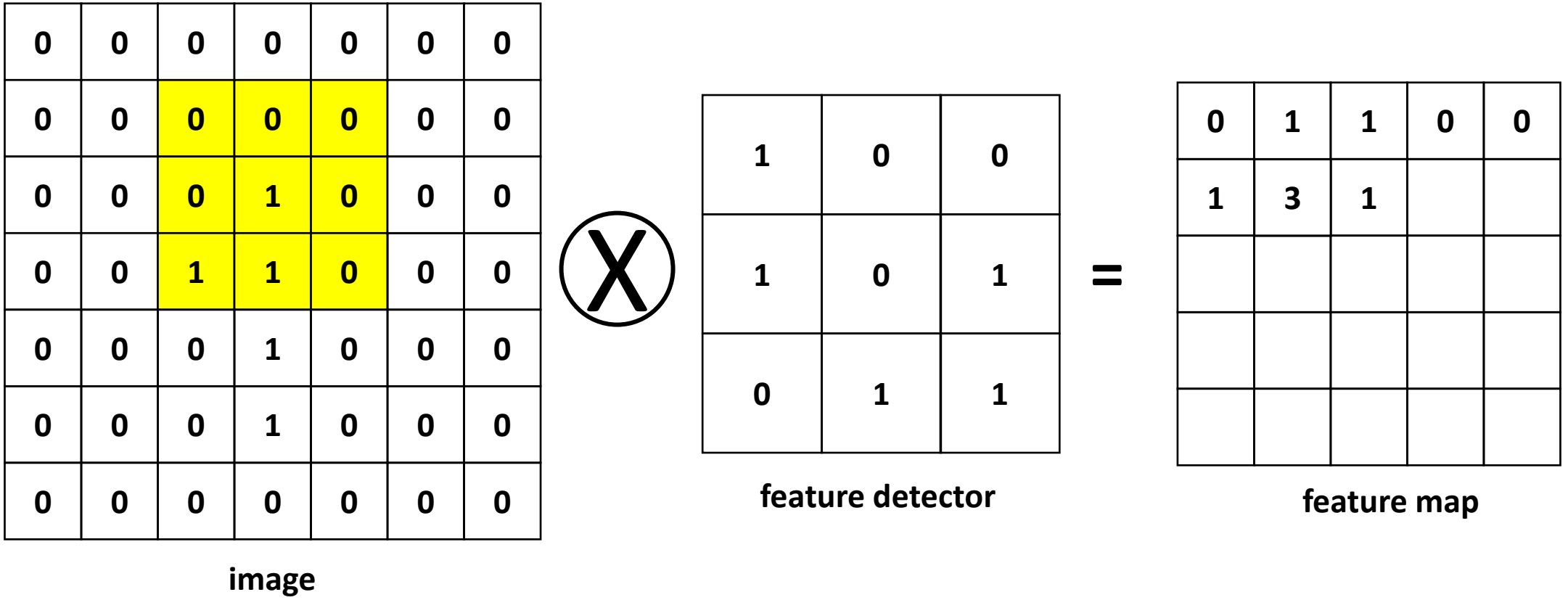
CONVOLUTIONAL NEURAL NETWORKS



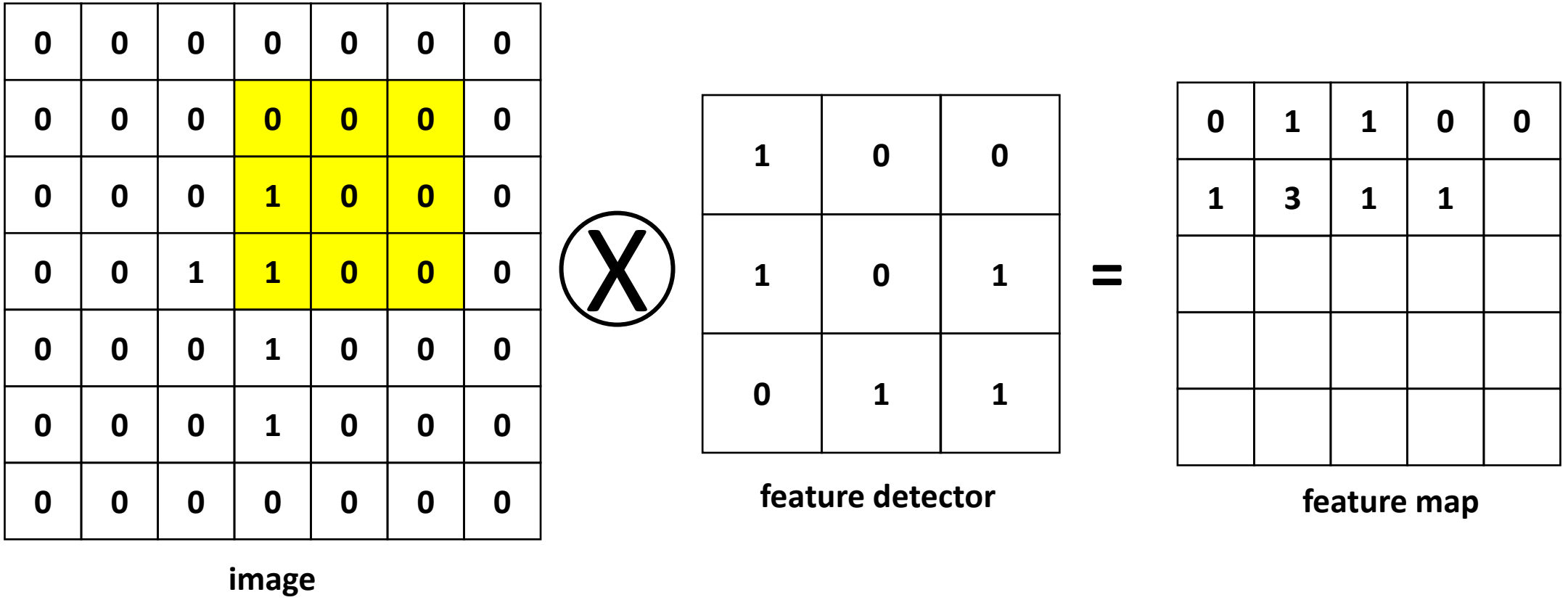
CONVOLUTIONAL NEURAL NETWORKS



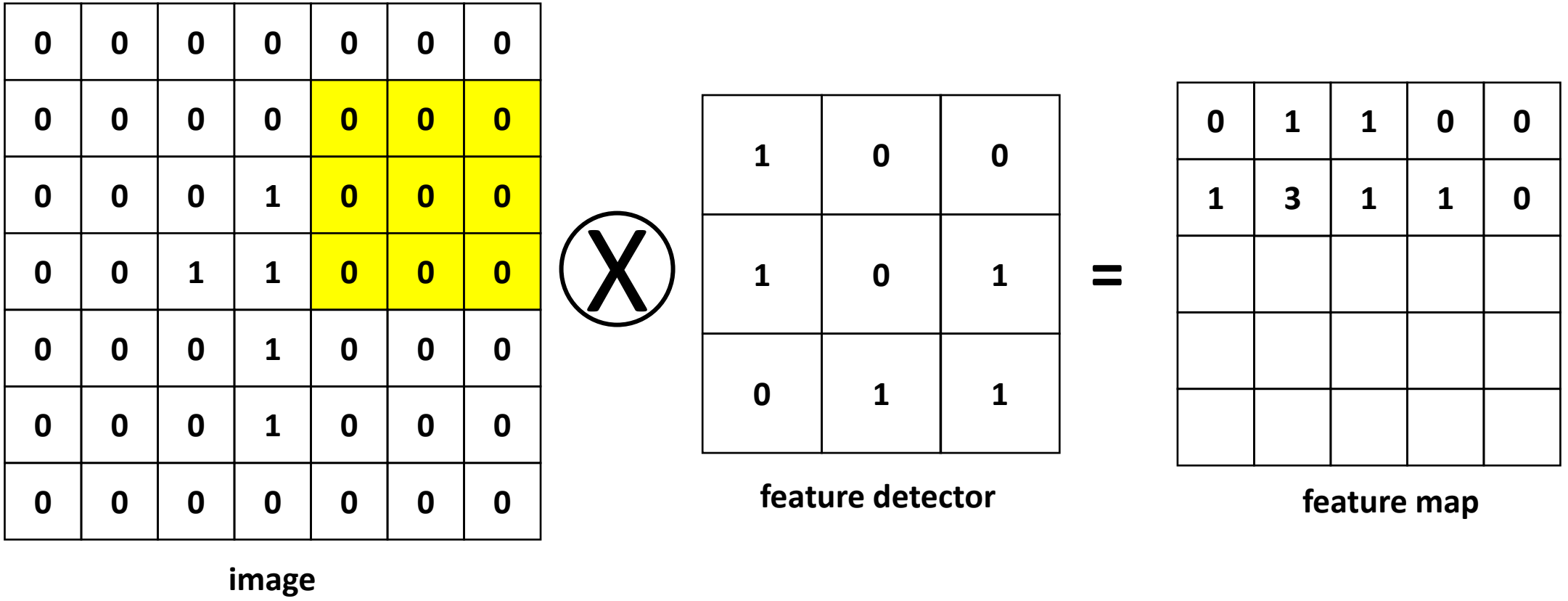
CONVOLUTIONAL NEURAL NETWORKS



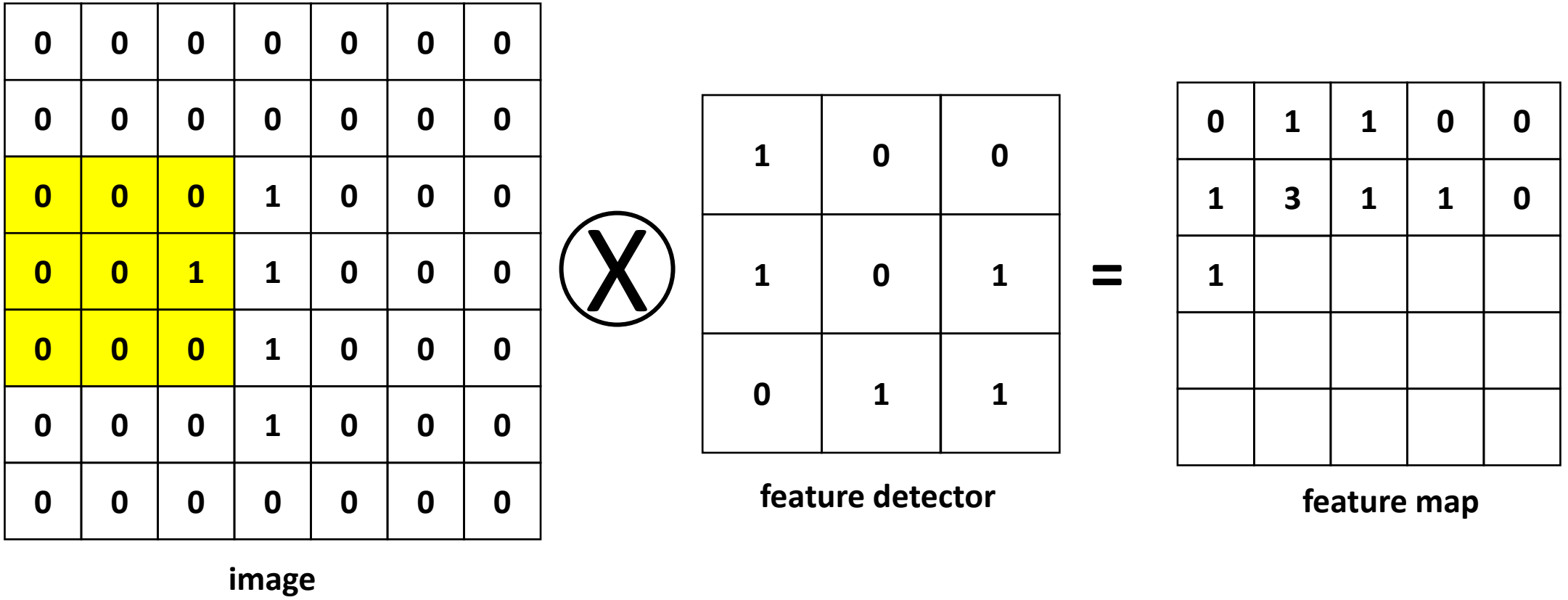
CONVOLUTIONAL NEURAL NETWORKS



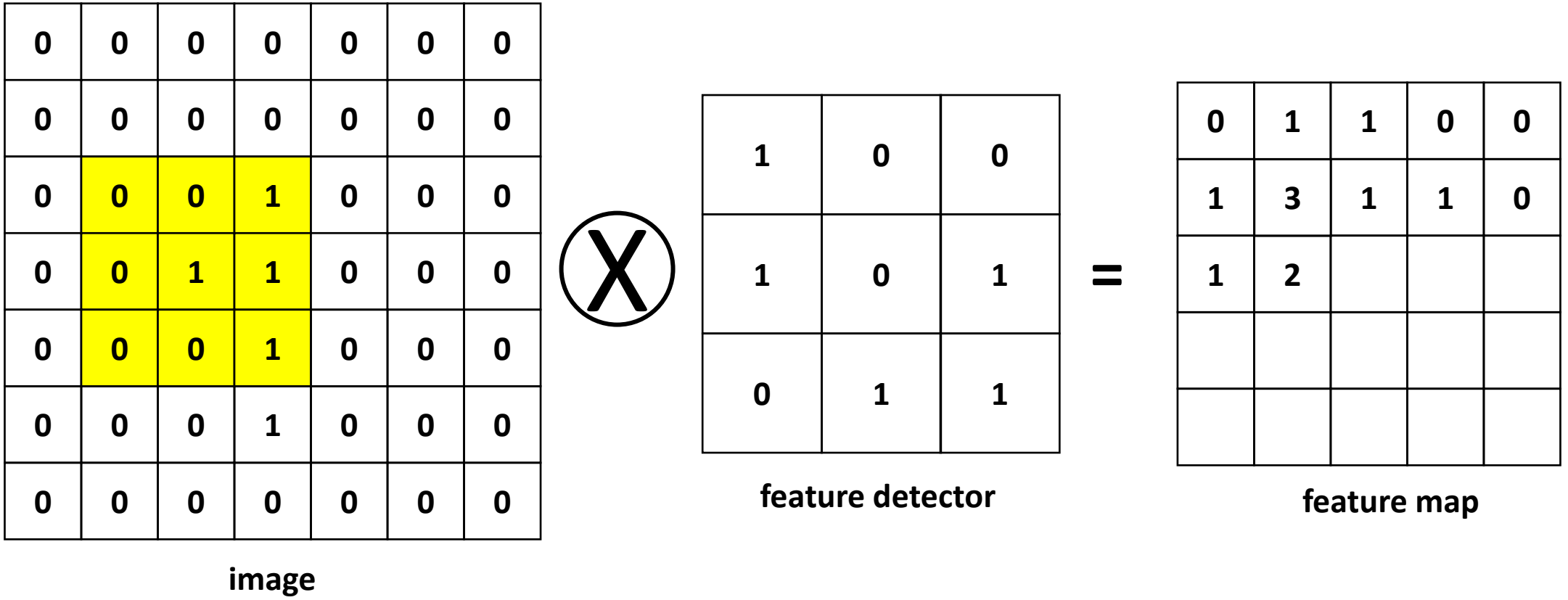
CONVOLUTIONAL NEURAL NETWORKS



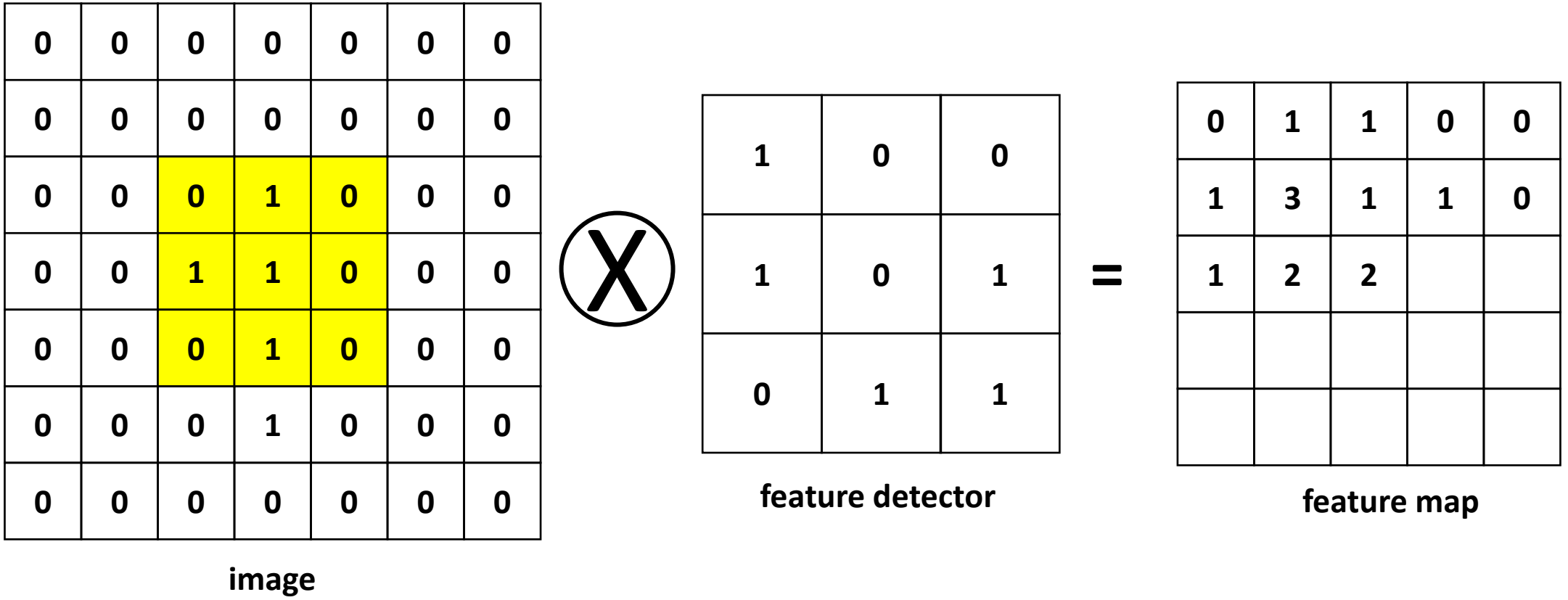
CONVOLUTIONAL NEURAL NETWORKS



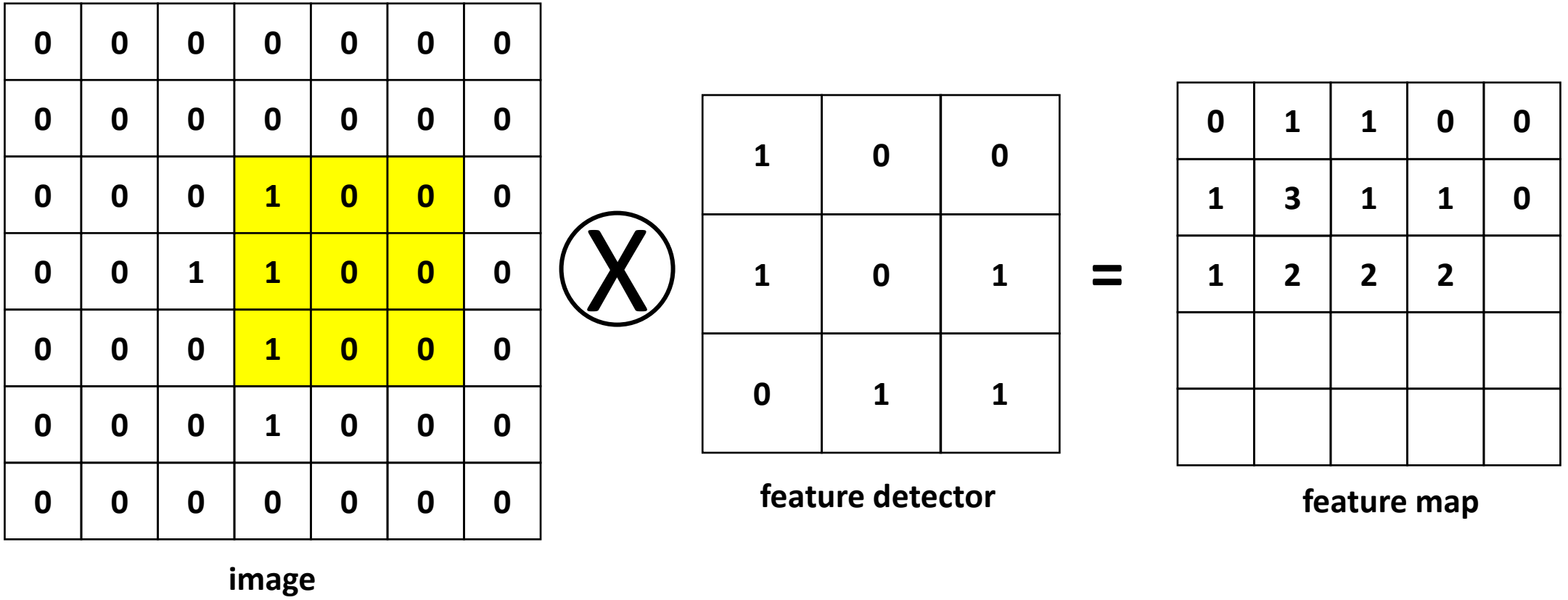
CONVOLUTIONAL NEURAL NETWORKS



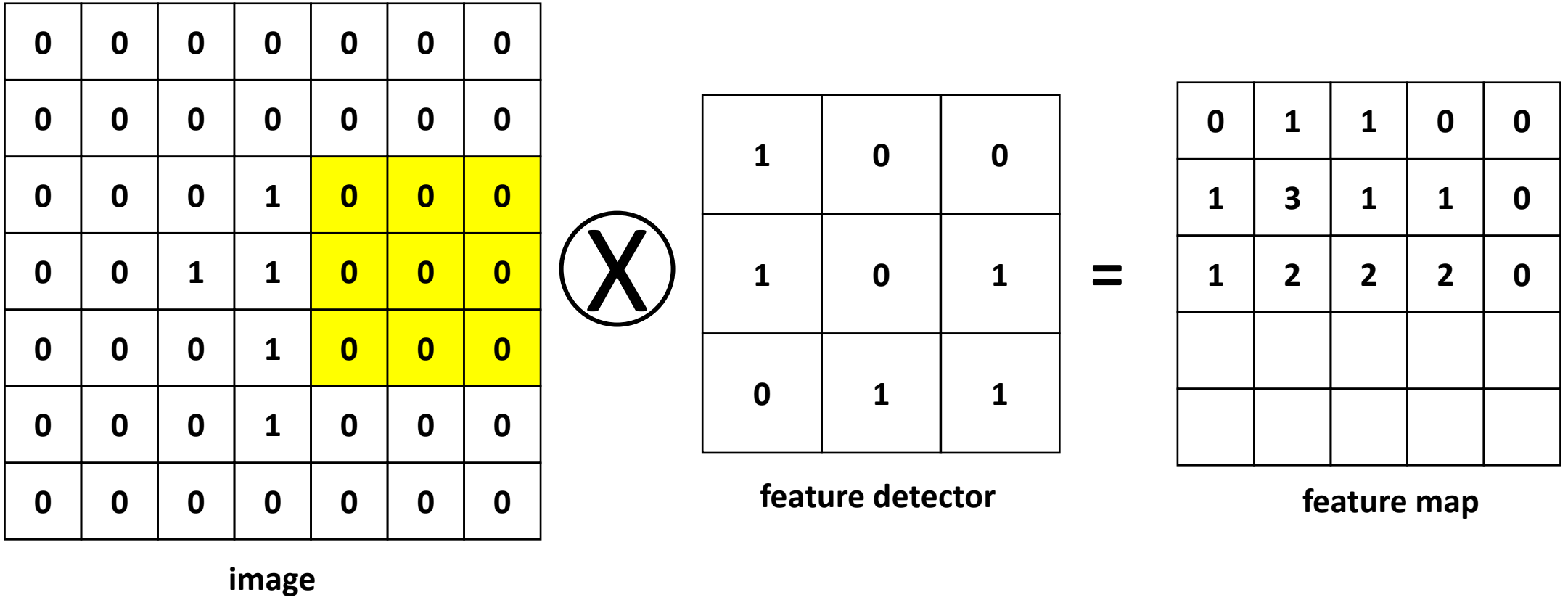
CONVOLUTIONAL NEURAL NETWORKS



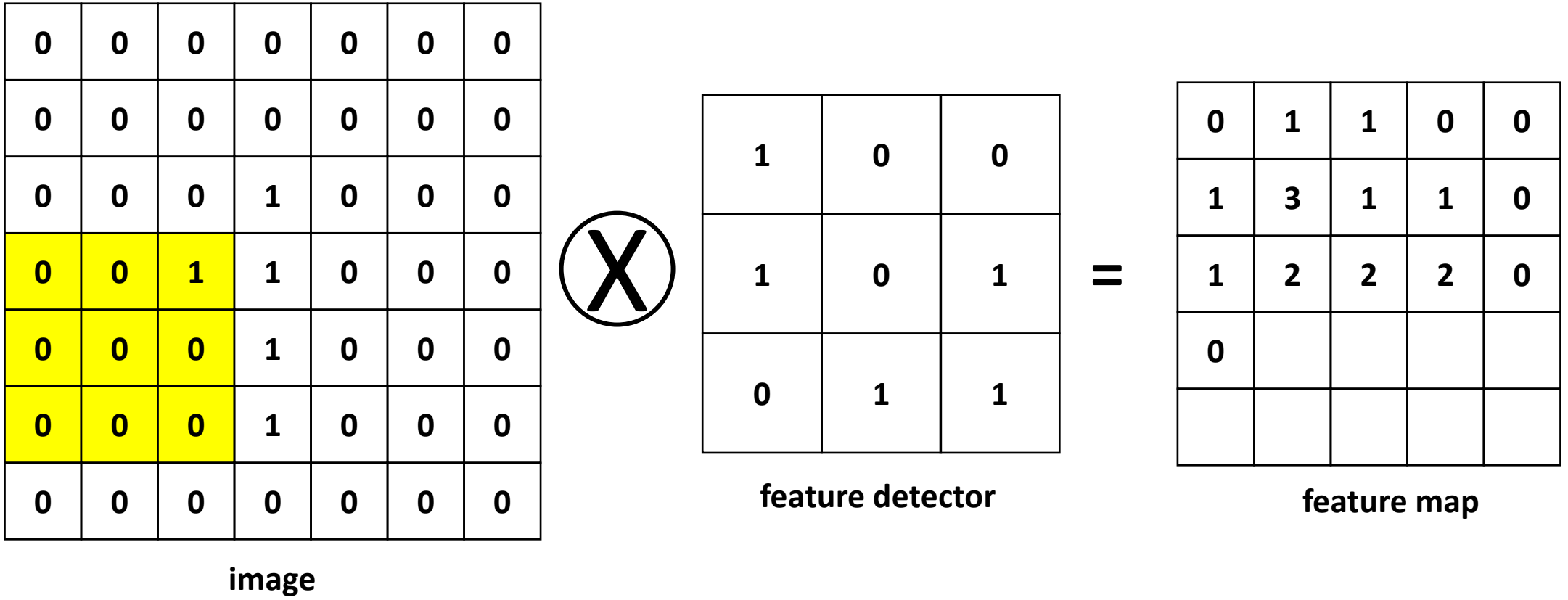
CONVOLUTIONAL NEURAL NETWORKS



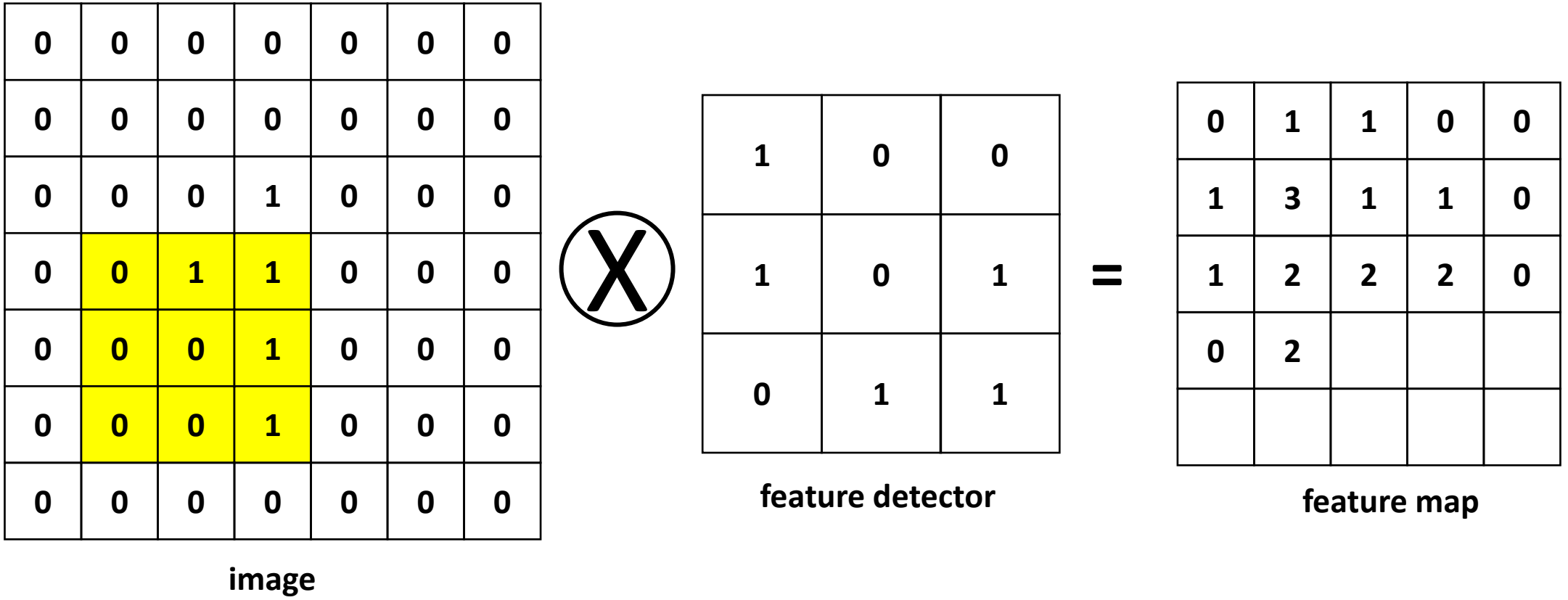
CONVOLUTIONAL NEURAL NETWORKS



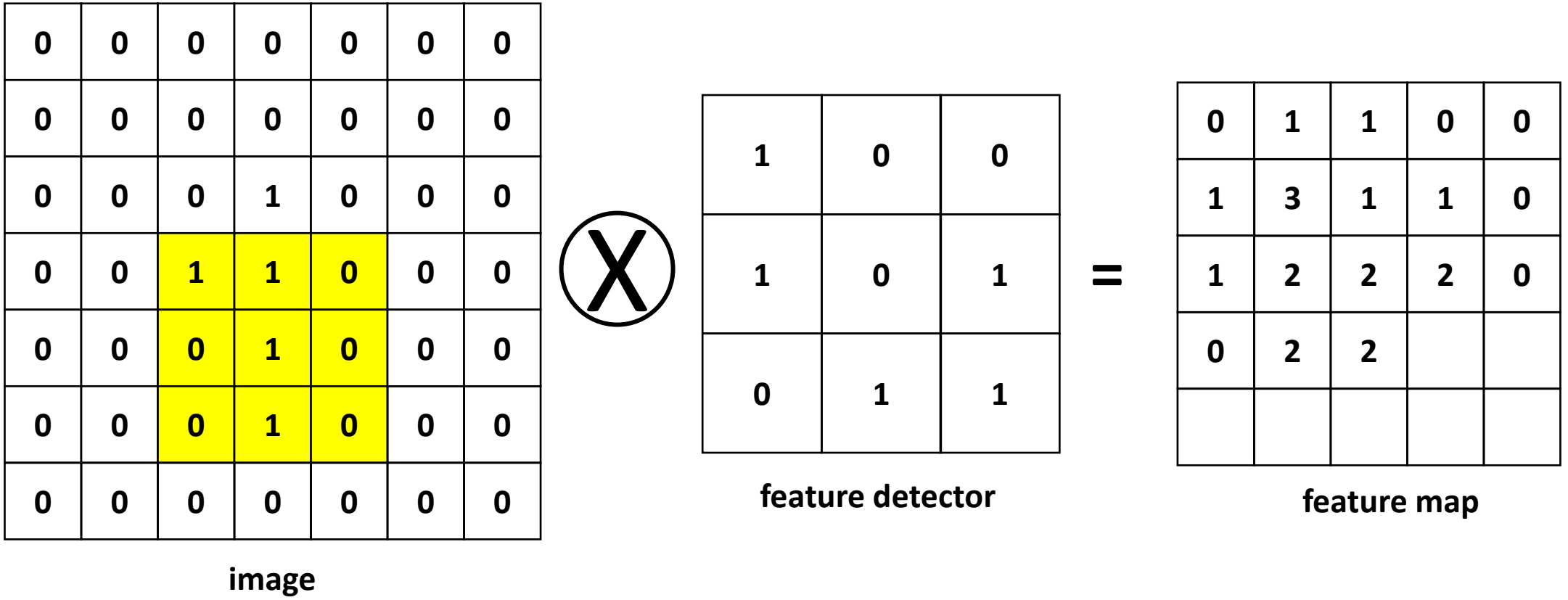
CONVOLUTIONAL NEURAL NETWORKS



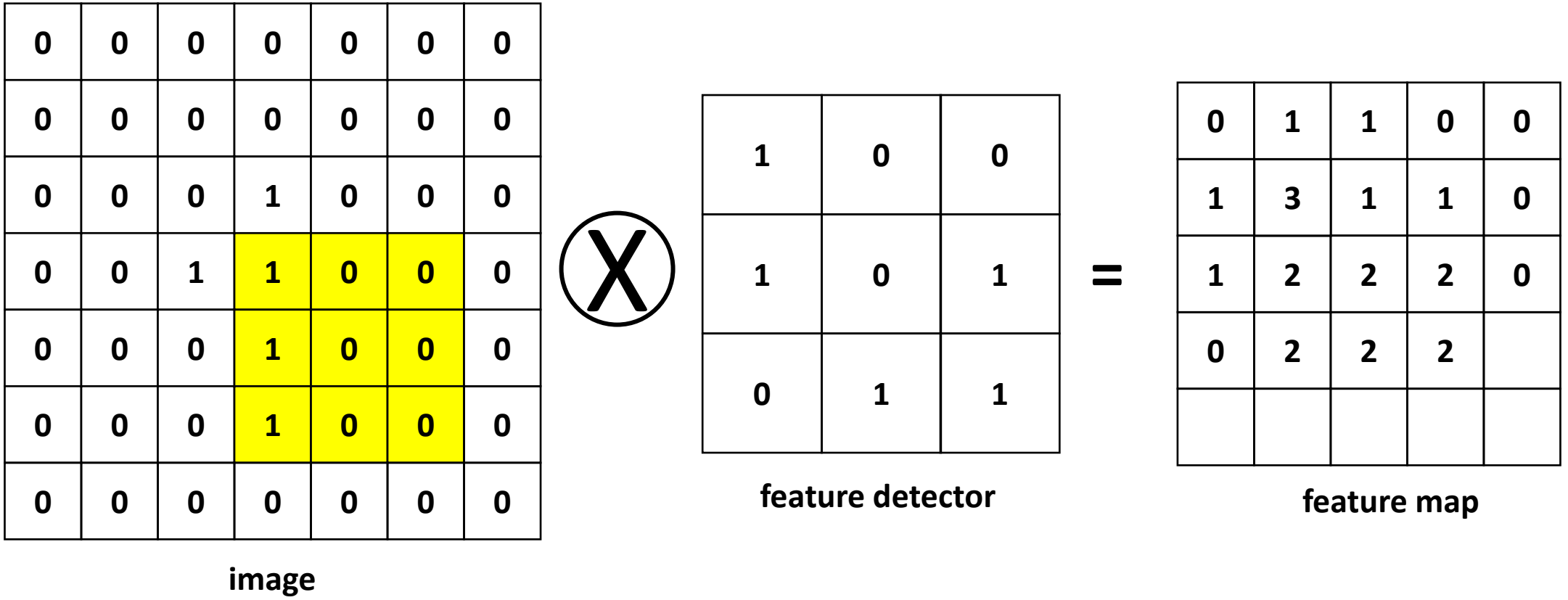
CONVOLUTIONAL NEURAL NETWORKS



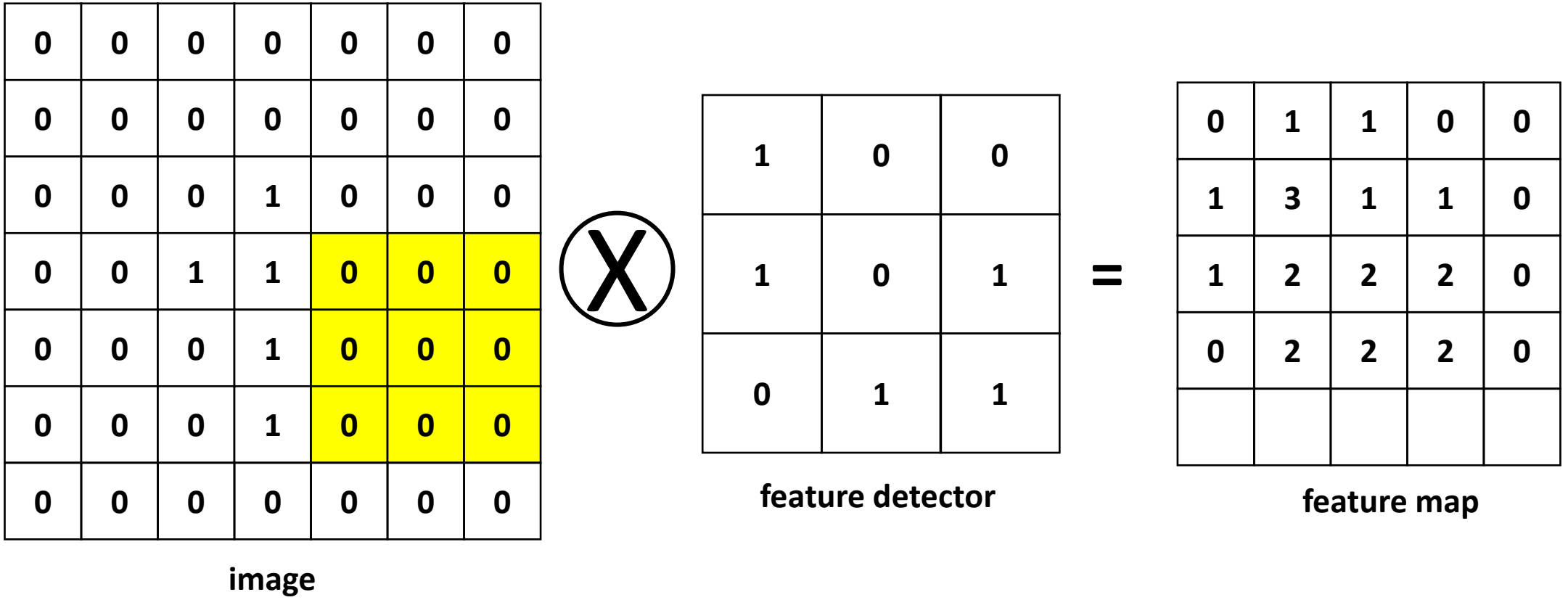
CONVOLUTIONAL NEURAL NETWORKS



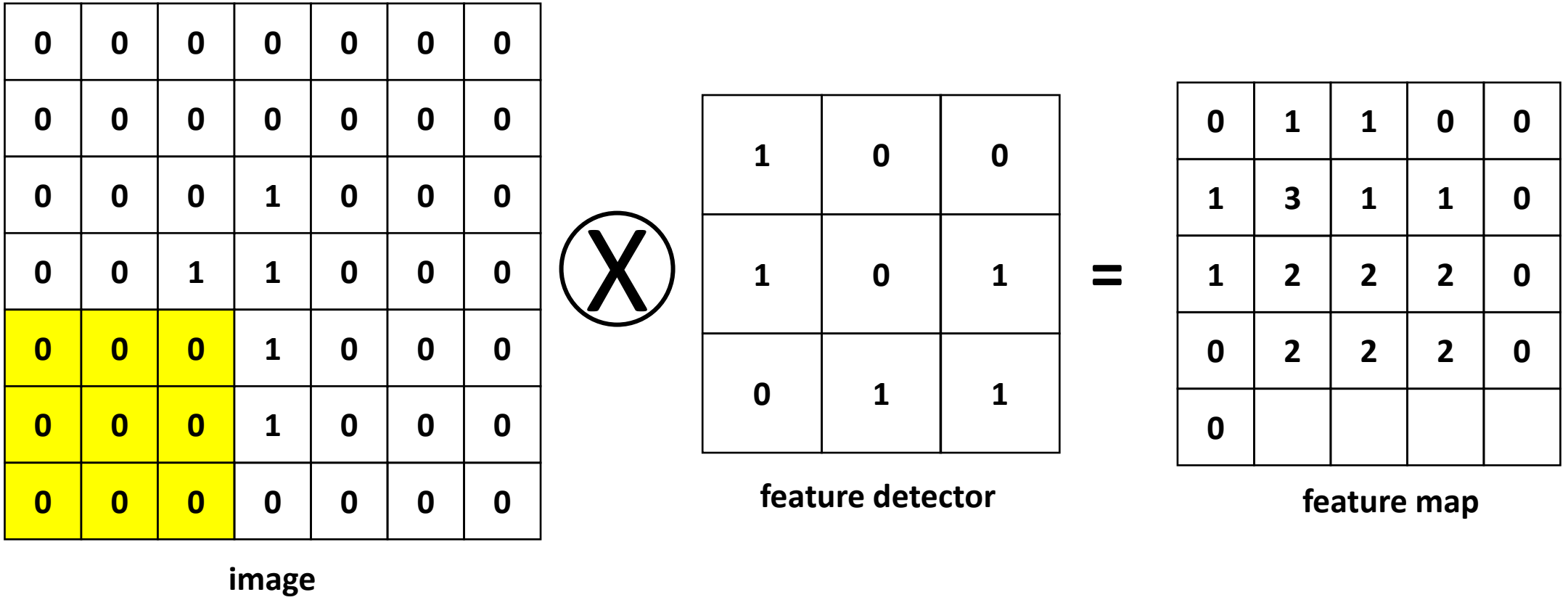
CONVOLUTIONAL NEURAL NETWORKS



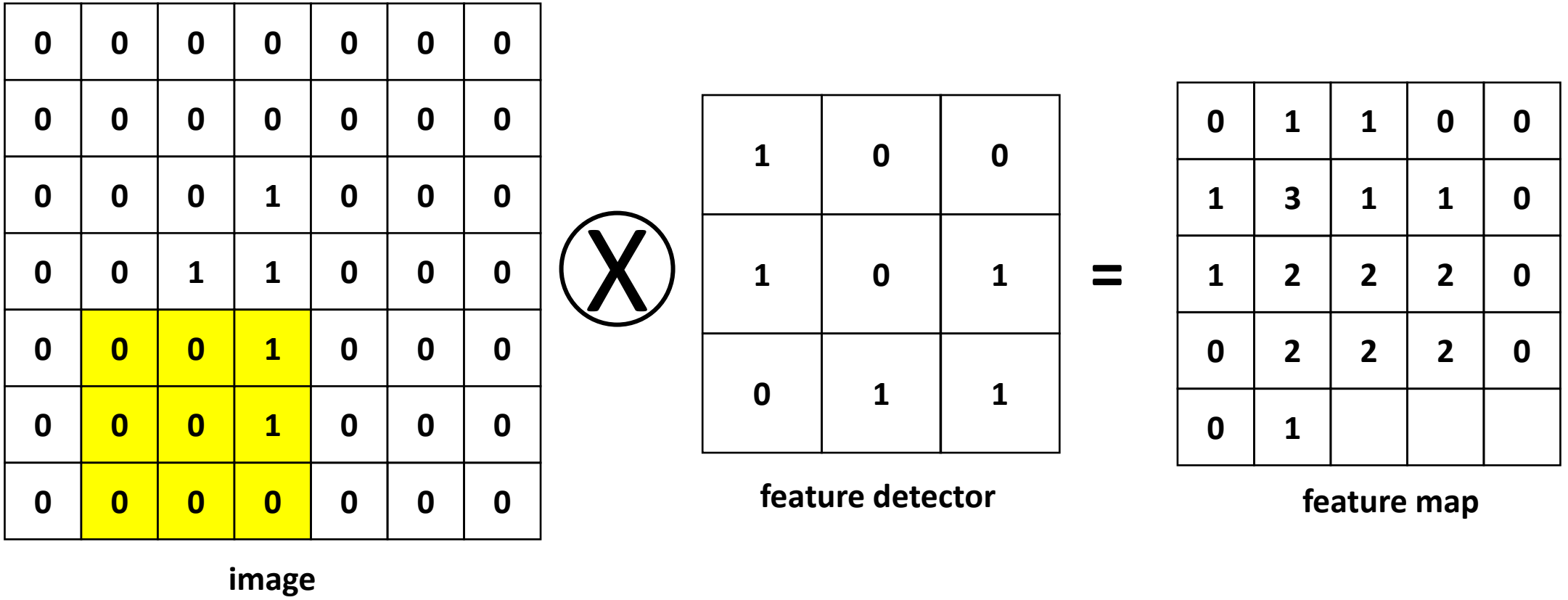
CONVOLUTIONAL NEURAL NETWORKS



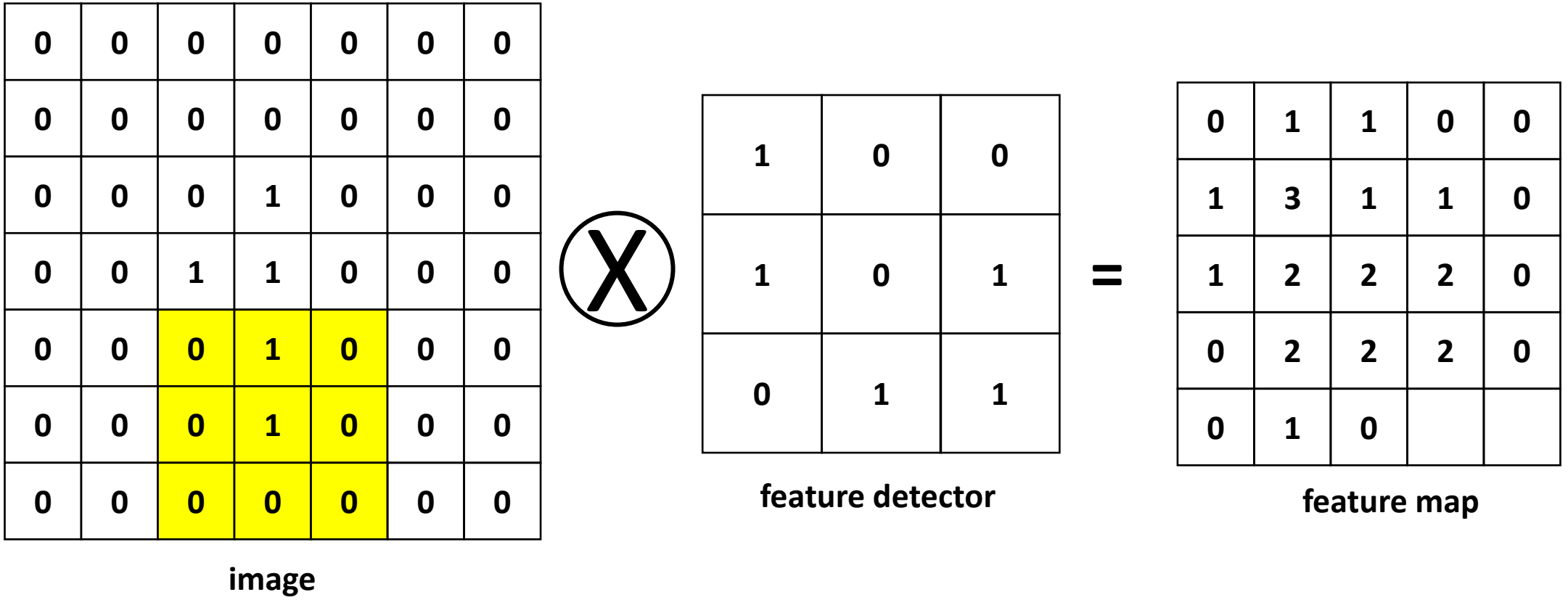
CONVOLUTIONAL NEURAL NETWORKS



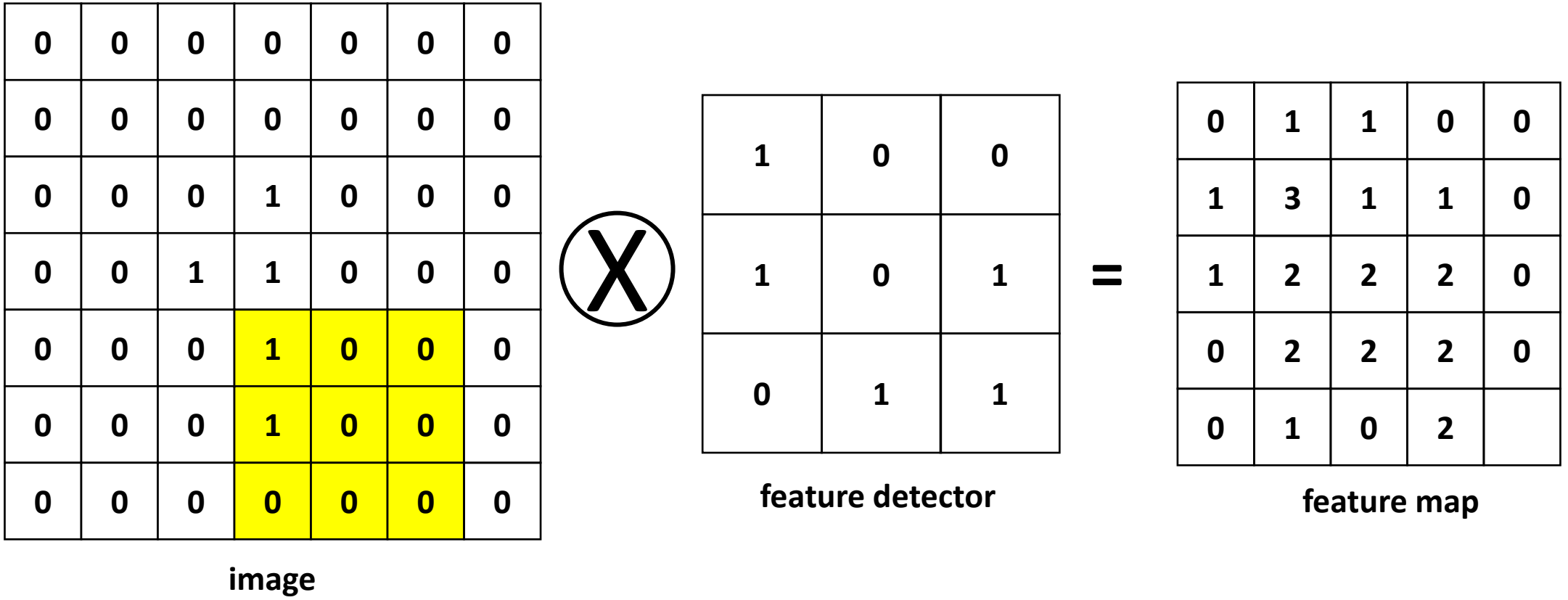
CONVOLUTIONAL NEURAL NETWORKS



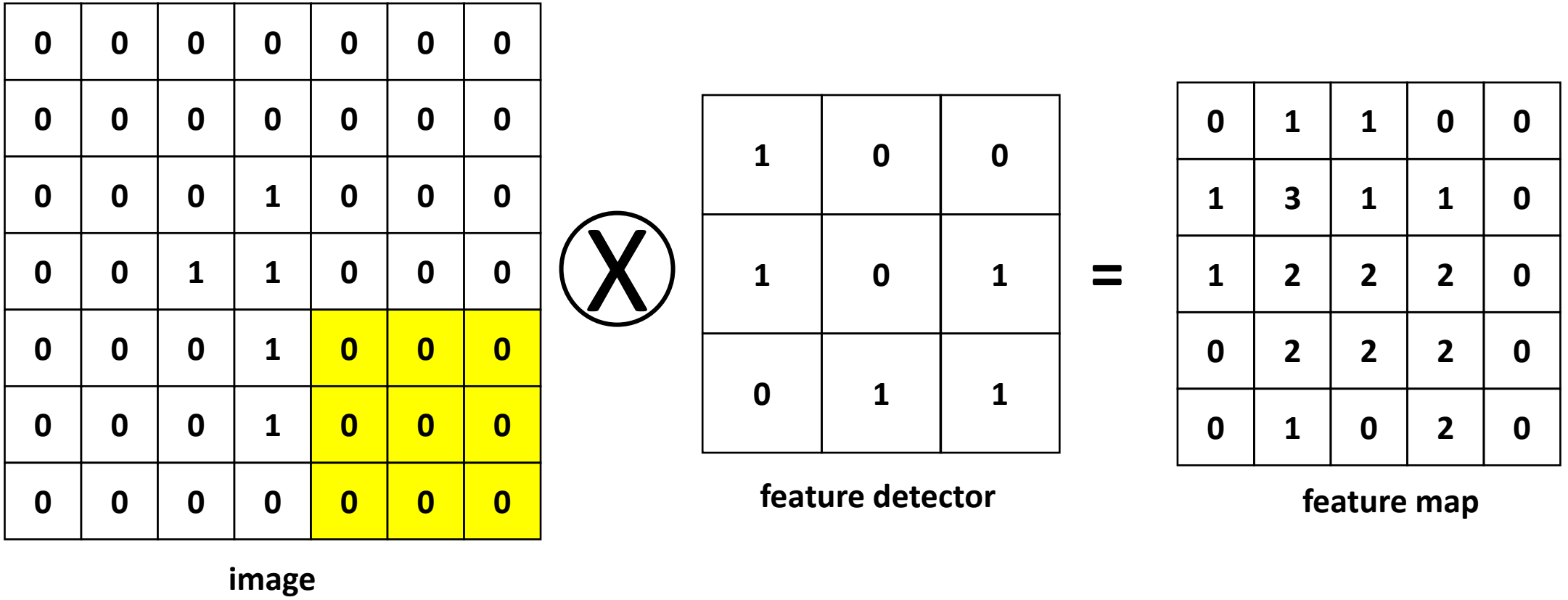
CONVOLUTIONAL NEURAL NETWORKS



CONVOLUTIONAL NEURAL NETWORKS



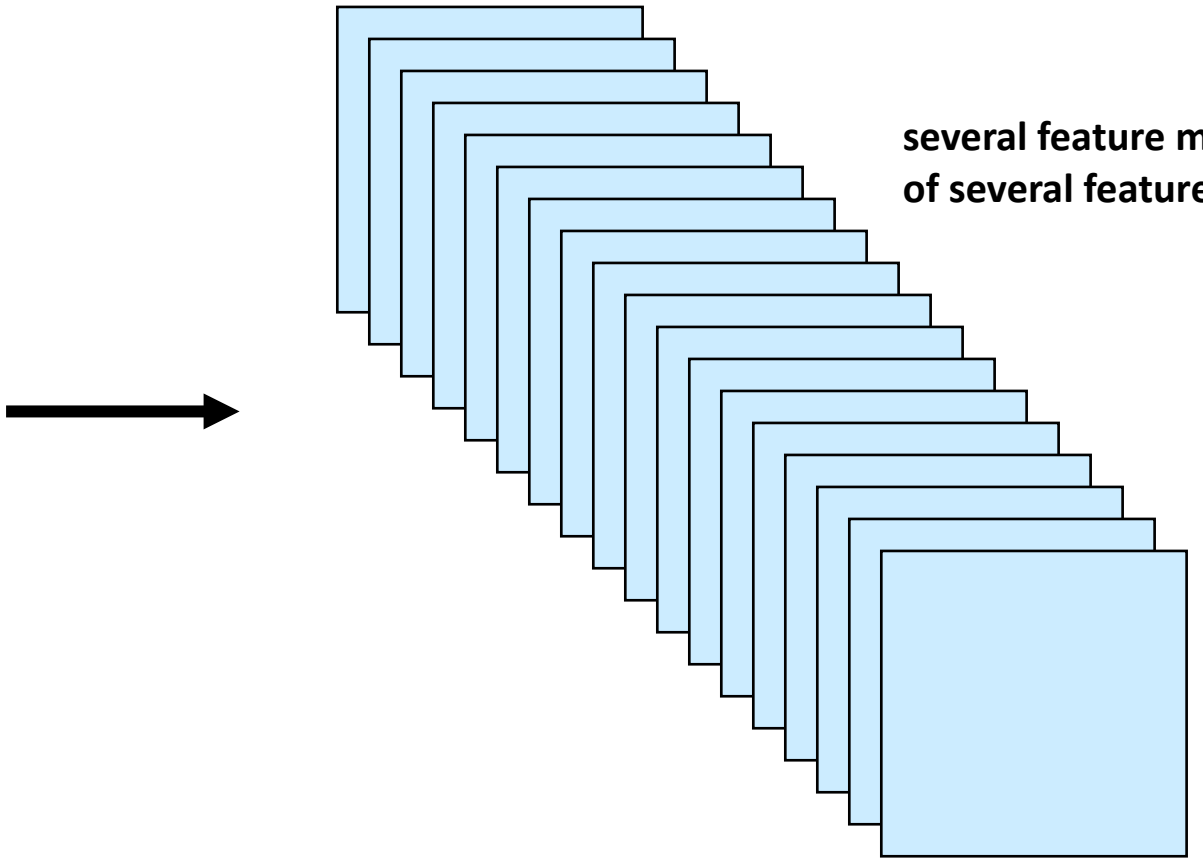
CONVOLUTIONAL NEURAL NETWORKS



CONVOLUTIONAL NEURAL NETWORKS

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	1	0	0	0
0	0	0	1	0	0	0
0	0	0	1	0	0	0
0	0	0	0	0	0	0

image

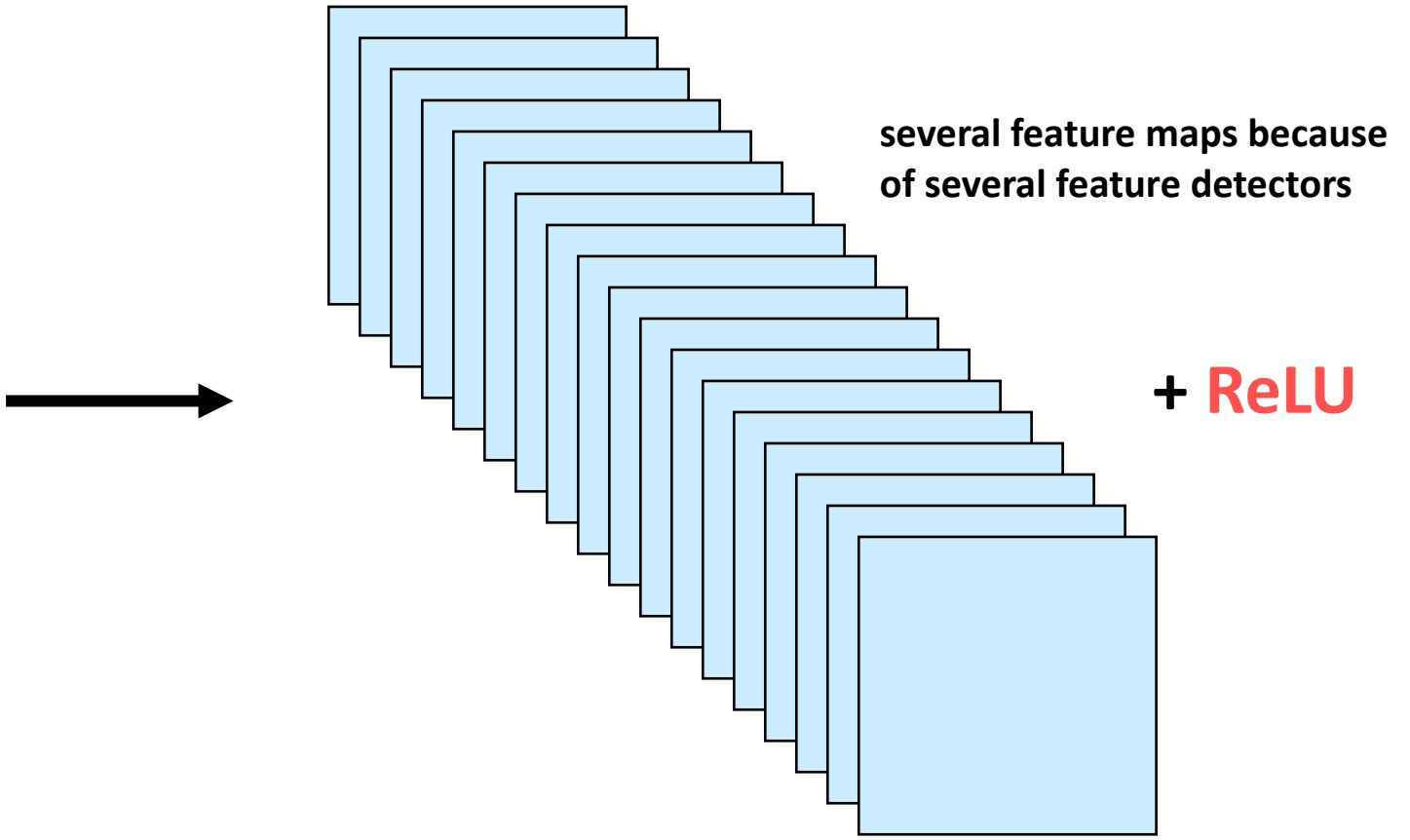


several feature maps because of several feature detectors

CONVOLUTIONAL NEURAL NETWORKS

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	1	0	0	0
0	0	0	1	0	0	0
0	0	0	1	0	0	0
0	0	0	0	0	0	0

image



CONVOLUTIONAL NEURAL NETWORKS

„spatial invariance”: we would like to make sure to detect the same object no matter where is it located on the image or whether it is rotated/transformed !!!

It is a cat: the location on the image does not matter or whether it is rotated or transformed

CONVOLUTIONAL NEURAL NETWORKS

With **max pooling** we select the most relevant features: this is how we deal with spatial invariance. We just care about the most relevant features !!!

1	**0**	**1**	**2**	**0**
1	**2**	**0**	**2**	**1**
1	**0**	**0**	**1**	**0**
1	**1**	**2**	**3**	**0**
0	**0**	**1**	**0**	**0**

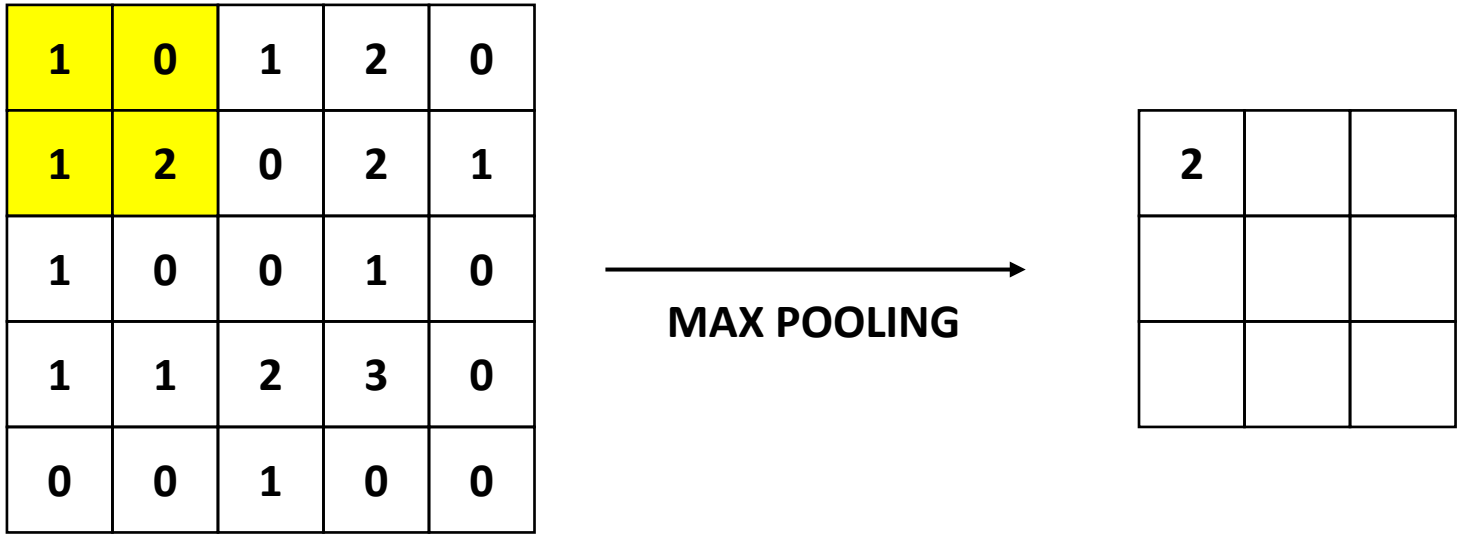
feature map

~ we can reduce the dimension of the image in order to end with a dataset containing the important pixel values without the unnecessary noise

+ we reduce number of parameters: reduce overfitting !!!

CONVOLUTIONAL NEURAL NETWORKS

With **max pooling** we select the most relevant features: this is how we deal with spatial invariance. We just care about the most relevant features !!!



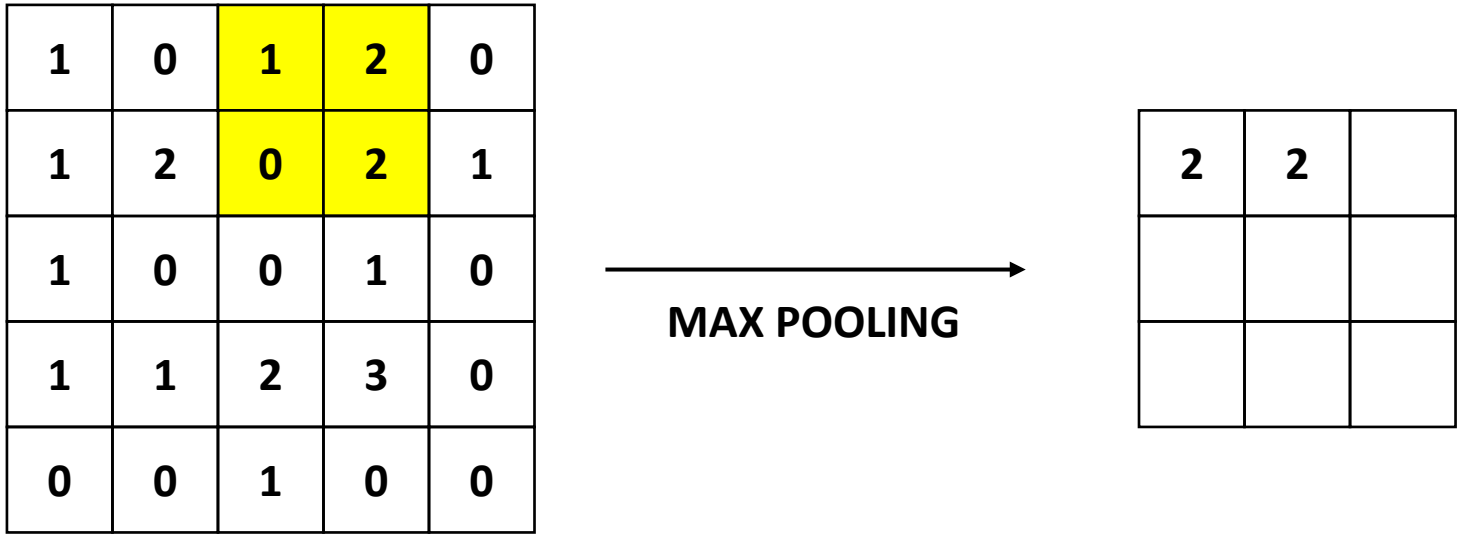
feature map

~ we can reduce the dimension of the image in order to end with a dataset containing the important pixel values without the unnecessary noise

+ we reduce number of parameters: reduce overfitting !!!

CONVOLUTIONAL NEURAL NETWORKS

With **max pooling** we select the most relevant features: this is how we deal with spatial invariance. We just care about the most relevant features !!!



feature map

~ we can reduce the dimension of the image in order to end with a dataset containing the important pixel values without the unnecessary noise

+ we reduce number of parameters: reduce overfitting !!!

CONVOLUTIONAL NEURAL NETWORKS

With **max pooling** we select the most relevant features: this is how we deal with spatial invariance. We just care about the most relevant features !!!

1	0	1	2	0
1	2	0	2	1
1	0	0	1	0
1	1	2	3	0
0	0	1	0	0

feature map

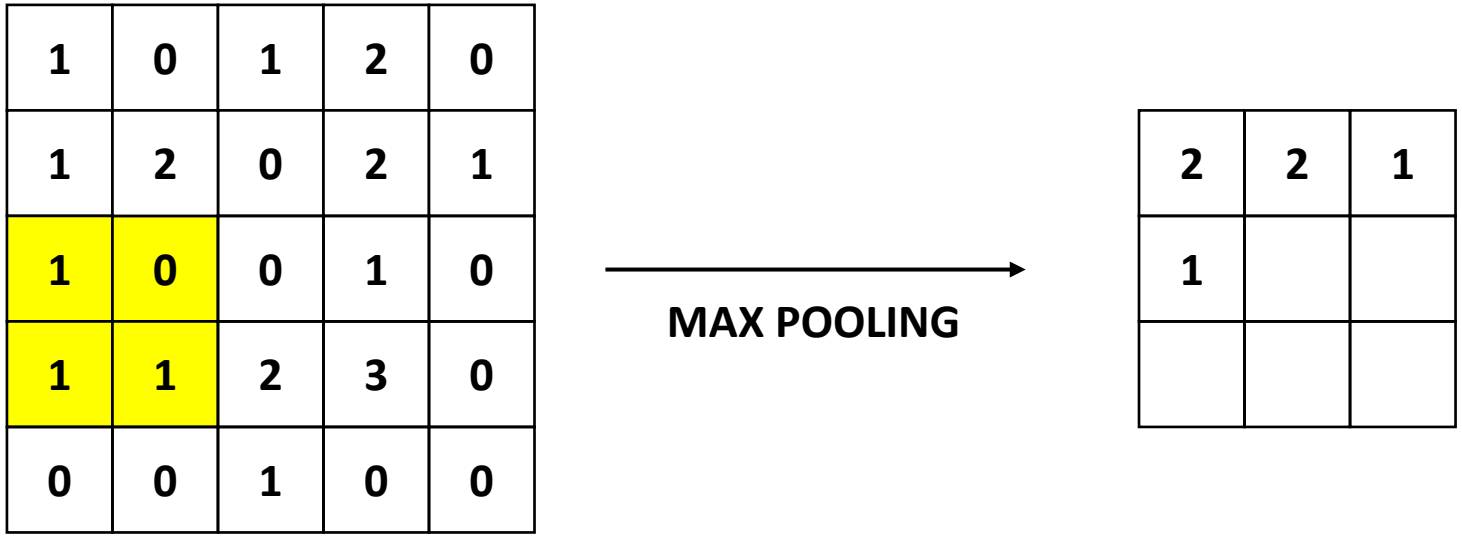
2	2	1

~ we can reduce the dimension of the image in order to end with a dataset containing the important pixel values without the unnecessary noise

+ we reduce number of parameters: reduce overfitting !!!

CONVOLUTIONAL NEURAL NETWORKS

With **max pooling** we select the most relevant features: this is how we deal with spatial invariance. We just care about the most relevant features !!!



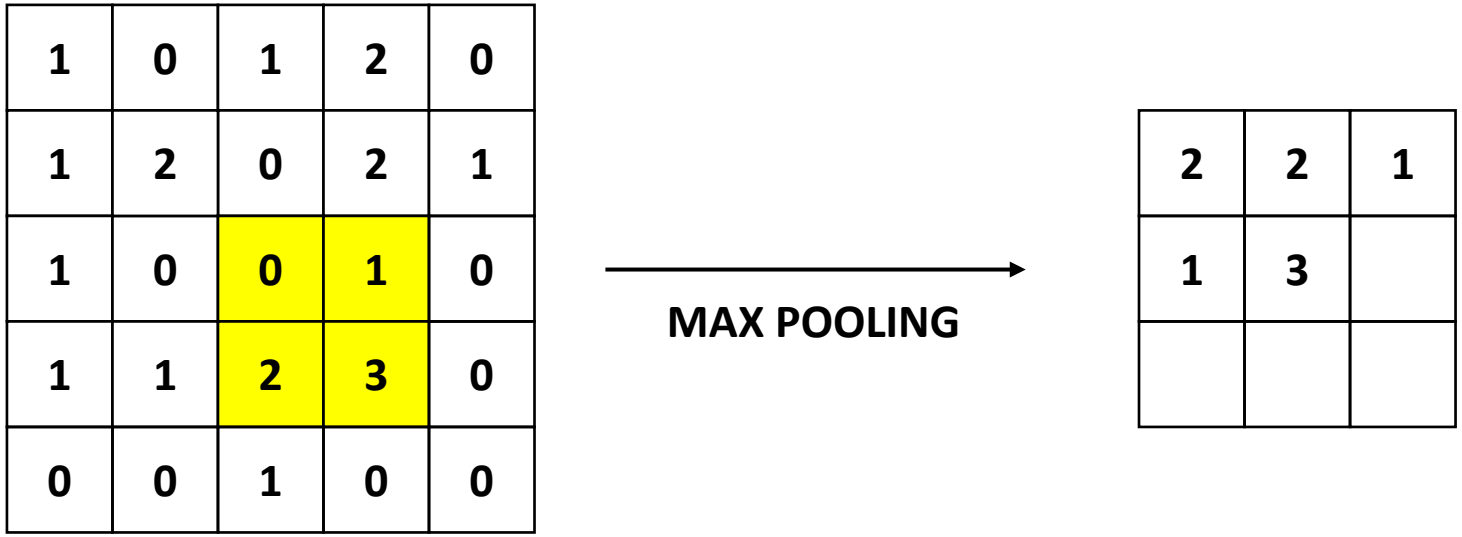
feature map

~ we can reduce the dimension of the image in order to end with a dataset containing the important pixel values without the unnecessary noise

+ we reduce number of parameters: reduce overfitting !!!

CONVOLUTIONAL NEURAL NETWORKS

With **max pooling** we select the most relevant features: this is how we deal with spatial invariance. We just care about the most relevant features !!!



~ we can reduce the dimension of the image in order to end with a dataset containing the important pixel values without the unnecessary noise

+ we reduce number of parameters: reduce overfitting !!!

CONVOLUTIONAL NEURAL NETWORKS

With **max pooling** we select the most relevant features: this is how we deal with spatial invariance. We just care about the most relevant features !!!

1	0	1	2	0
1	2	0	2	1
1	0	0	1	0
1	1	2	3	0
0	0	1	0	0

feature map

2	2	1
1	3	0

~ we can reduce the dimension of the image in order to end with a dataset containing the important pixel values without the unnecessary noise

+ we reduce number of parameters: reduce overfitting !!!

CONVOLUTIONAL NEURAL NETWORKS

With **max pooling** we select the most relevant features: this is how we deal with spatial invariance. We just care about the most relevant features !!!

1	0	1	2	0
1	2	0	2	1
1	0	0	1	0
1	1	2	3	0
0	0	1	0	0

feature map

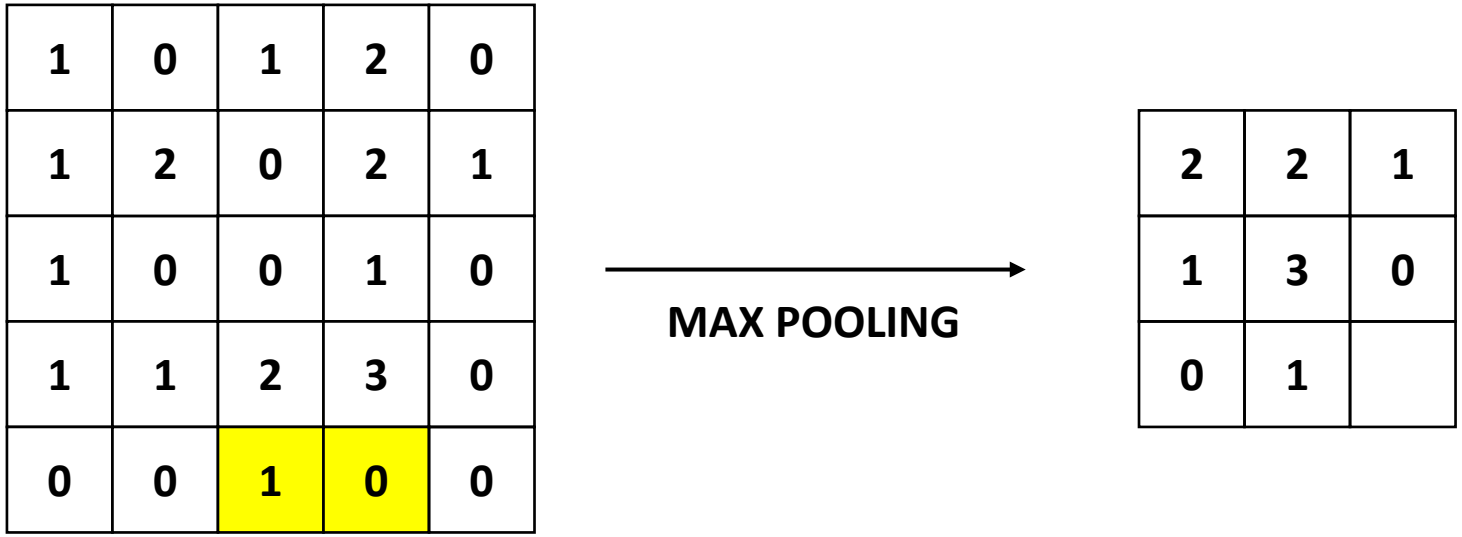
2	2	1
1	3	0
0		

~ we can reduce the dimension of the image in order to end with a dataset containing the important pixel values without the unnecessary noise

+ we reduce number of parameters: reduce overfitting !!!

CONVOLUTIONAL NEURAL NETWORKS

With **max pooling** we select the most relevant features: this is how we deal with spatial invariance. We just care about the most relevant features !!!



feature map

~ we can reduce the dimension of the image in order to end with a dataset containing the important pixel values without the unnecessary noise

+ we reduce number of parameters: reduce overfitting !!!

CONVOLUTIONAL NEURAL NETWORKS

With **max pooling** we select the most relevant features: this is how we deal with spatial invariance. We just care about the most relevant features !!!

1	0	1	2	0
1	2	0	2	1
1	0	0	1	0
1	1	2	3	0
0	0	1	0	0

feature map

2	2	1
1	3	0
0	1	0

~ we can reduce the dimension of the image in order to end with a dataset containing the important pixel values without the unnecessary noise

+ we reduce number of parameters: reduce overfitting !!!

CONVOLUTIONAL NEURAL NETWORKS

With **max pooling** we select the most relevant features: this is how we deal with spatial invariance. We just care about the most relevant features !!!

1	**0**	**1**	**2**	**0**
1	**2**	**0**	**2**	**1**
1	**0**	**0**	**1**	**0**
1	**1**	**2**	**3**	**0**
0	**0**	**1**	**0**	**0**

feature map

MAX POOLING

2	**2**	**1**
1	**3**	**0**
0	**1**	**0**

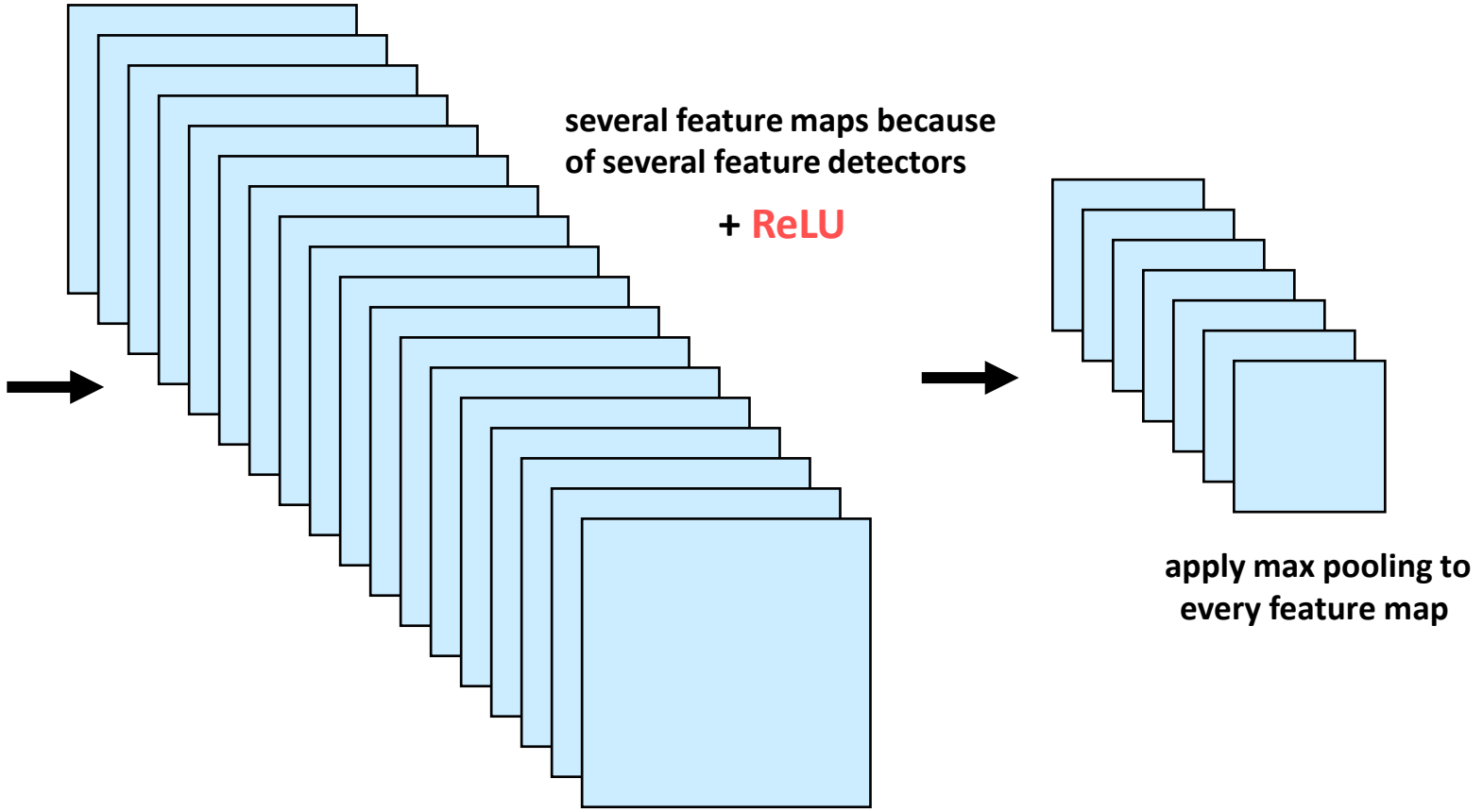
There are other techniques: **average pooling** is popular as well
~ instead of choosing the maximum value we calculate the average of the values present in the subset

CONVOLUTIONAL NEURAL NETWORKS

With **max pooling** we select the most relevant features: this is how we deal with spatial invariance. We just care about the most relevant features !!!

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	1	0	0	0
0	0	0	1	0	0	0
0	0	0	1	0	0	0
0	0	0	0	0	0	0

image



CONVOLUTIONAL NEURAL NETWORKS

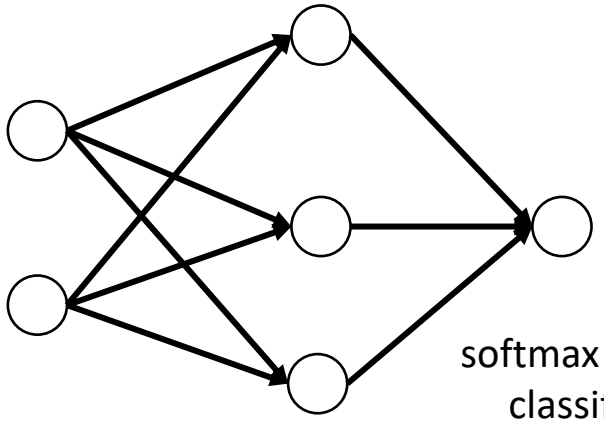
The last operation we have to make is the **flattening** procedure: we transform the matrix into a one-dimensional vector: this is the input of a standard densely connected neural network

2	2	1
1	3	0
0	1	0

pooled feature map

FLATTENING

2
2
1
1
3
0
0
1
0



softmax activation to classify outputs

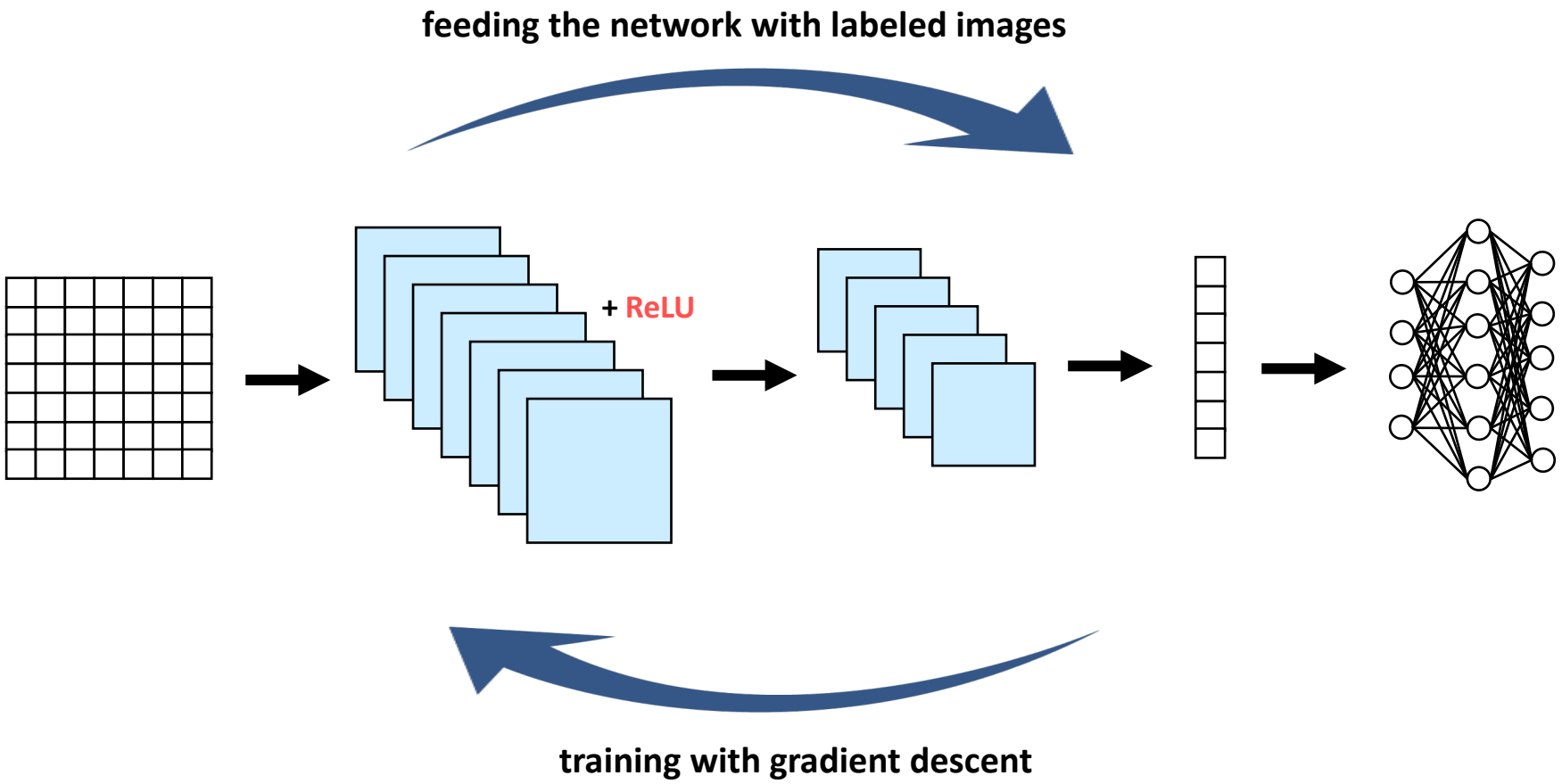
WHY IS IT GOOD? Because we preprocess the data and use ANN with just the relevant feature values

we use multilayer neural network to learn the non-linear combinations of these important features

CONVOLUTIONAL NEURAL NETWORKS- TRAINING

We use gradient descent (backpropagation) as usual as far as training the convolutional neural networks are concerned

~ update the edge weights according to the error + choosing the right filter !!!



CONVOLUTIONAL NEURAL NETWORKS

Xavier weight initialization:

→ it helps signals reach deep into the network

→ weight is **too small**: the signal shrinks as it passes through each layer,
it may become too tiny to be useful

→ Weight is **too large**: signal grows as it passes through each layer,
it may be too massive to be useful

So the solution is to draw weights from a distribution with **0** mean and variance:

$$\sigma^2(w) = \frac{1}{n_{in}}$$

~ **n** is the number of input neurons and the distribution
is the Gaussian-distribution of course

CONVOLUTIONAL NEURAL NETWORKS

Nesterov's momentum/updater:

→ helps gradient descent converges faster

→ increases the scale of updates when those updates are consistently in one direction

→ as if we were going down a smooth but very shallow hill
The direction is clear, but we might want to take larger steps

Neural Networks

SUPERVISED LEARNING

→ artificial neural networks (**ANN**)
These networks are used for regression and classification

→ convolutional neural networks (**CNN**)
Used for computer vision: self-driving cars...

→ recurrent neural networks (**RNN**)
These networks can be used for time series analysis
(stock market or FOREX)

UNSUPERVISED LEARNING

→ Boltzmann-machines and autoencoders (+self-organizing maps)
are used for recommendation systems and feature detection

Recurrent Neural Networks

→ google translator relies heavily on recurrent neural networks

→ we can use recurrent neural networks to make time series analysis

Turing-test: a computer passes the Turing-test if a human is unable to distinguish the computer from a human in a blind test

~ recurrent neural networks are able to pass this test: a well-trained recurrent network is able to „understand" English for example

LEARN LANGUAGE MODELS !!!

Recurrent Neural Networks

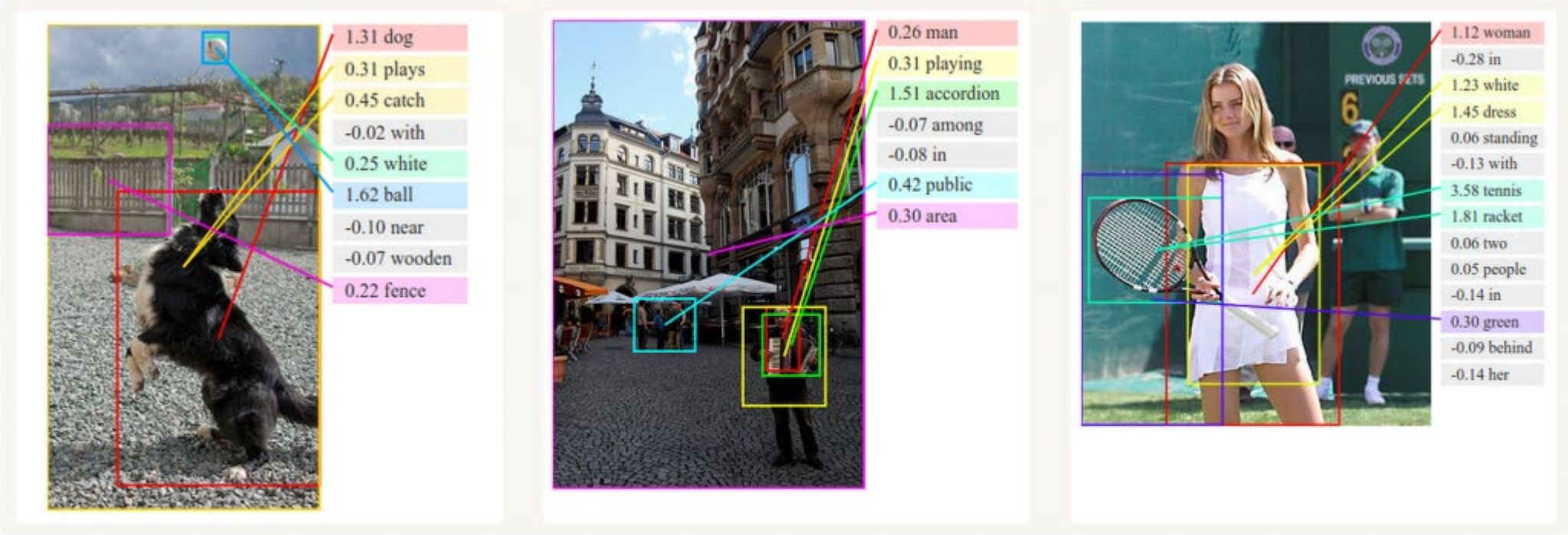
We would like to make sure that the network is able to learn connections in the data
even when they are far away from each other

„**I am from Hungary**. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. **I speak fluent** ...”

Recurrent neural networks are able to deal with relationships
far away from each other
~ it is able to guess the last word: hungarian

Recurrent Neural Networks

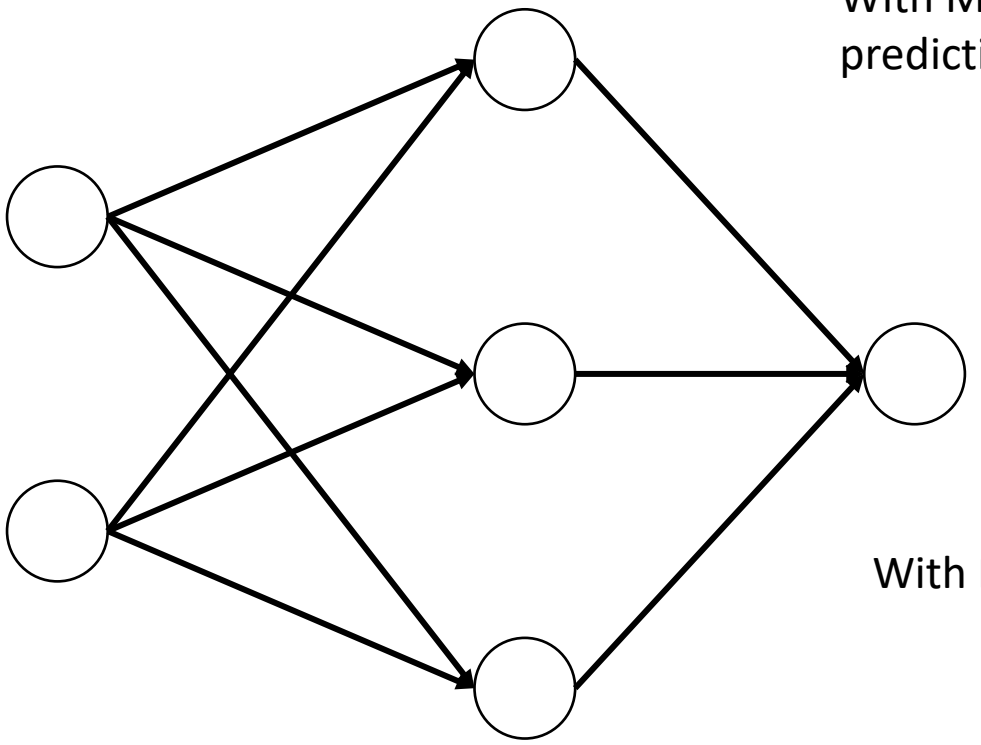
Combining convolutional neural networks with
recurrent neural networks is quite powerful



Deep Visual-Semantic Alignments for Generating Image Descriptions. Source: http://cs.stanford.edu/people/karpathy/deepimagesent/

~ we can generate image descriptions with this hibrid approach

Recurrent Neural Networks



With Multilayer Neural Networks (or deep networks) we make predictions independent of each other

p(t) is not correlated with **p(t-1)** or **p(t-2)** ...

→ training examples are independent of each other

Tigers, elephants, cats...nothing to do with each other

THESE PREDICTIONS ARE INDEPENDENT !!!

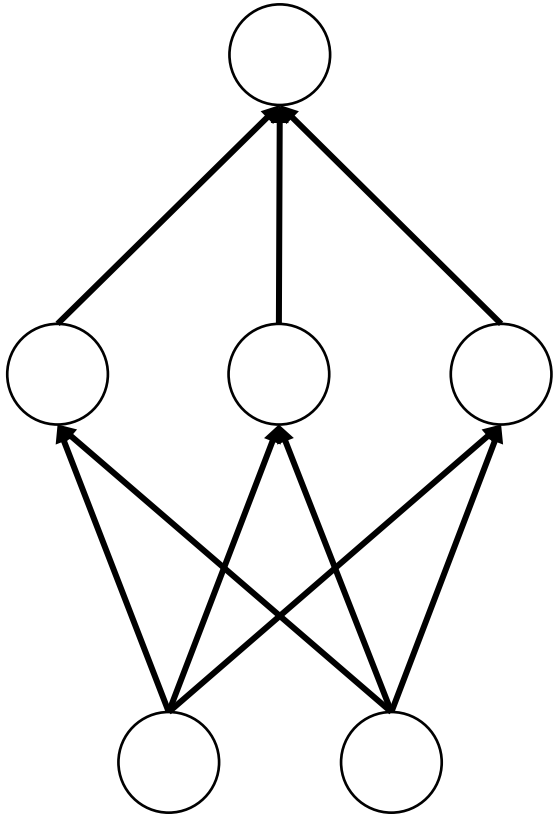
With **Recurrent Neural Networks** we can predict the next word in a given sentence: it is important in natural language processing

~ or we want to predict the stock prices tomorrow

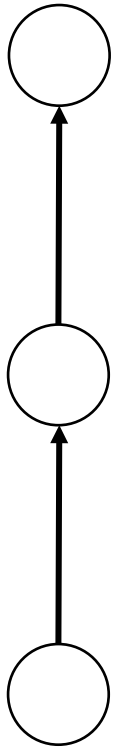
p(t) depends on **p(t-1)**, **p(t-2)** ...

TRAINING EXAMPLES ARE CORRELATED !!!

Recurrent Neural Networks

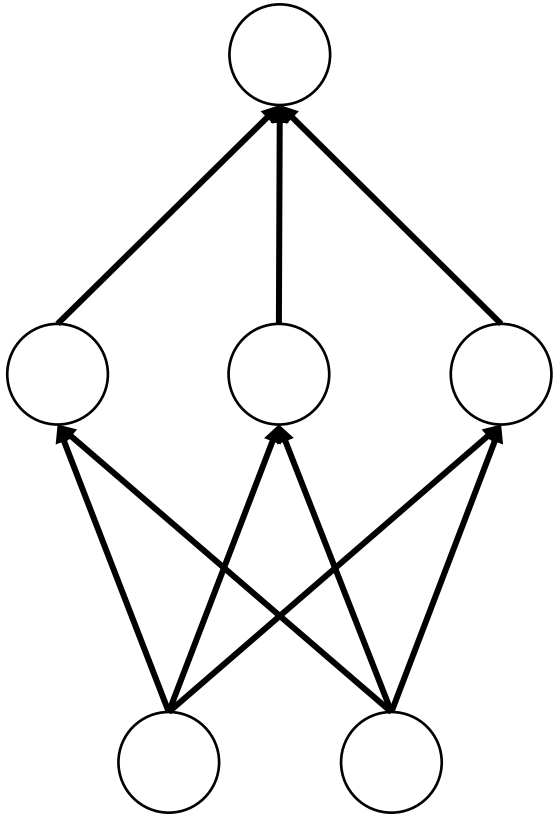


multilayer feedforward
neural network

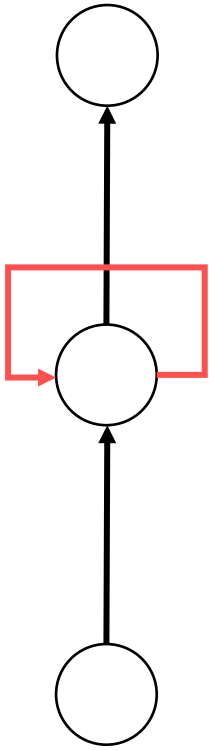


recurrent neural
network representation

Recurrent Neural Networks



multilayer feedforward
neural network



hidden layer gives an output
AND feeds back to itself

recurrent neural
network representation

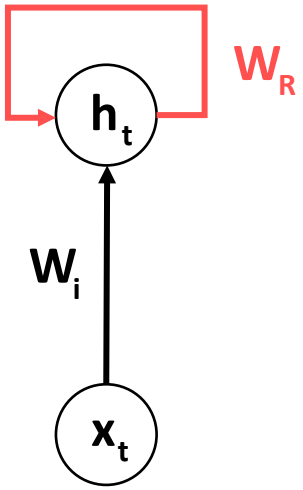
Recurrent Neural Networks

x: input

h: activation after applying the activation function on the output

$$h_t = g_h(W_i x_t + W_R h_{t-1} + b_h)$$

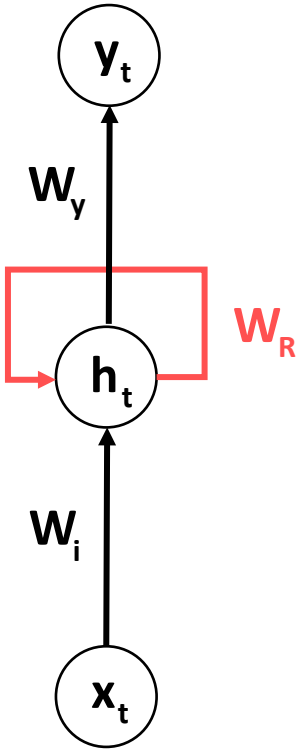
effect of input layer | effect of previous time step | bias



Recurrent Neural Networks

x: input

h: activation after applying the activation function on the output



$$h_t = g_h(W_i x_t + W_R h_{t-1} + b_h)$$

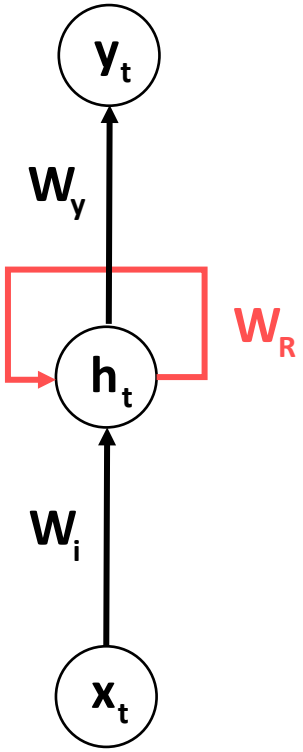
effect of input layer — effect of previous time step — bias

$$y_t = g_y(W_y h_t + b_h)$$ output

Recurrent Neural Networks

How to train a recurrent neural network?

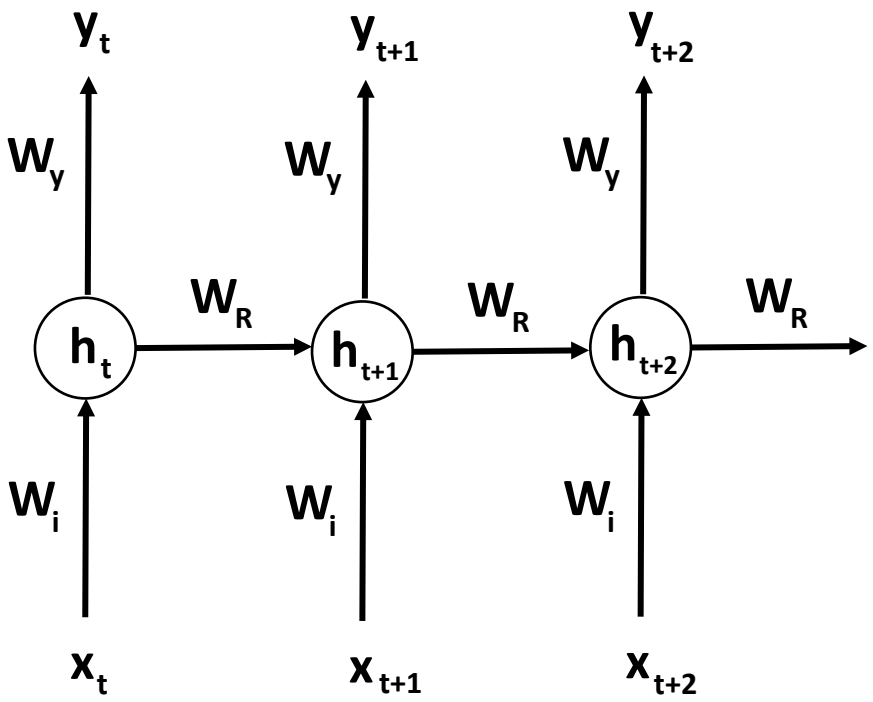
~ we can unroll it in time in order to end up with a standard feedforward neural network: we know how to deal with it



Recurrent Neural Networks

How to train a recurrent neural network?

~ we can unroll it in time in order to end up with a standard feedforward neural network: we know how to deal with it



$$h_t = g_h(W_i x_t + W_R h_{t-1} + b_h)$$

$$y_t = g_y(W_y h_t + b_h)$$

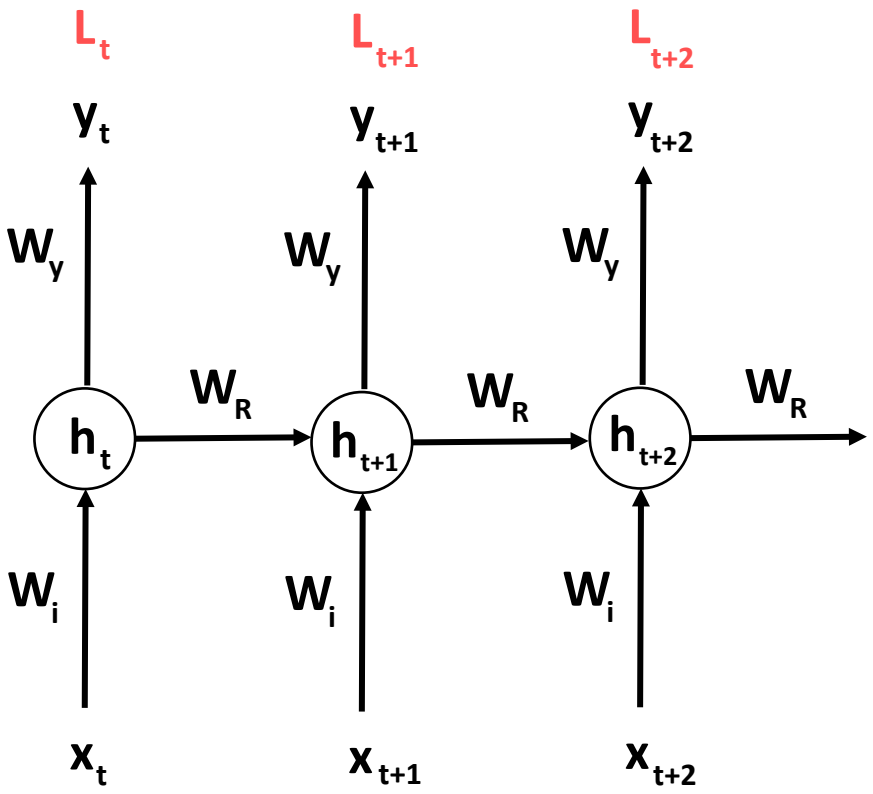
As you can see, several parameters are shared accross every single layer !!! W_R

~ for a feed-forward network these weights are different

Recurrent Neural Networks

Vanishing/exploding gradients problem

When dealing with backpropagation we have to calculate the gradients



$$\frac{\partial L}{\partial W_R} = \sum_t \frac{\partial L_t}{\partial W_R}$$

$$\frac{\partial L_t}{\partial W_R} = \sum_t \left(\frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial w} \right)$$

~ we just have to apply the chain rule several times

$$\frac{\partial L_t}{\partial W_R} = \prod_{t>i>k} W_R^T \, \mathrm{diag}(\sigma'(x_{i-1}))$$

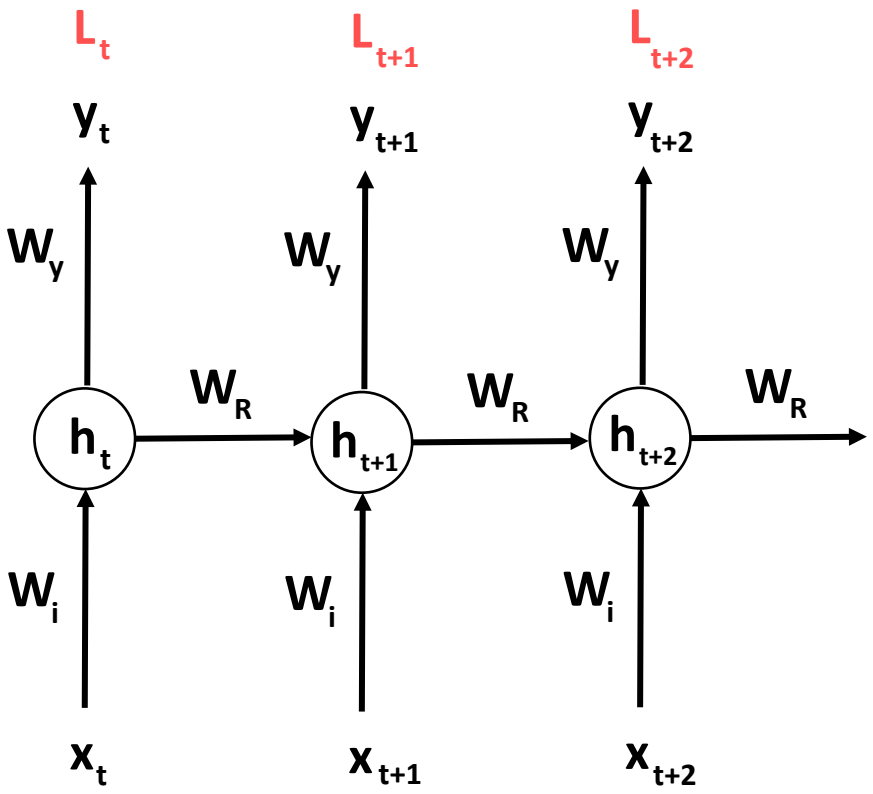
We multiply the weights several times: if you multiply **x<1** several times the result will get smaller and smaller

VANISHING GRADIENT PROBLEM

Recurrent Neural Networks

Vanishing/exploding gradients problem

When dealing with backpropagation we have to calculate the gradients



$$\frac{\partial L}{\partial W_R} = \sum_t \frac{\partial L_t}{\partial W_R}$$

$$\frac{\partial L_t}{\partial W_R} = \sum_t \left(\frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial w} \right)$$

~ we just have to apply the chain rule several times

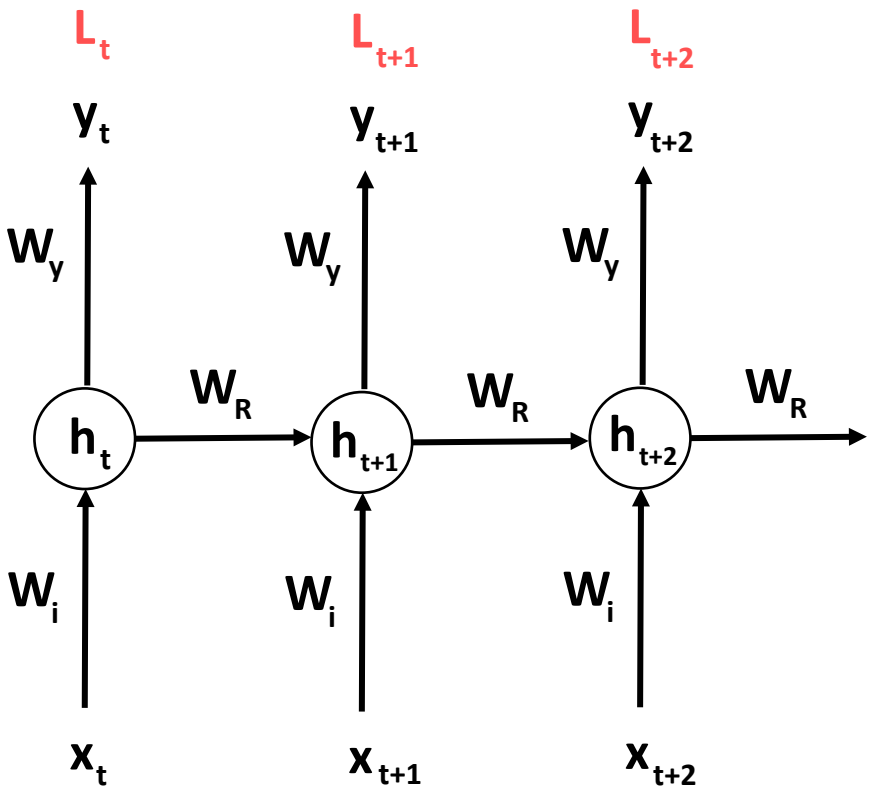
$$\frac{\partial L_t}{\partial W_R} = \prod_{t>i>k} W_R^T \, \mathrm{diag}(\sigma'(x_{i-1}))$$

Backpropagation Through Time (**BPTT**): the same as backpropagation but these gradients/error signals will also flow backward from future time-steps to current time-steps

Recurrent Neural Networks

Vanishing/exploding gradients problem

When dealing with backpropagation we have to calculate the gradients



$$\frac{\partial L}{\partial W_R} = \sum_t \frac{\partial L_t}{\partial W_R}$$

$$\frac{\partial L_t}{\partial W_R} = \sum_t \left(\frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial w} \right)$$

~ we just have to apply the chain rule several times

$$\frac{\partial L_t}{\partial W_R} = \prod_{t>i>k} W_R^T \, \mathrm{diag}(\sigma'(x_{i-1}))$$

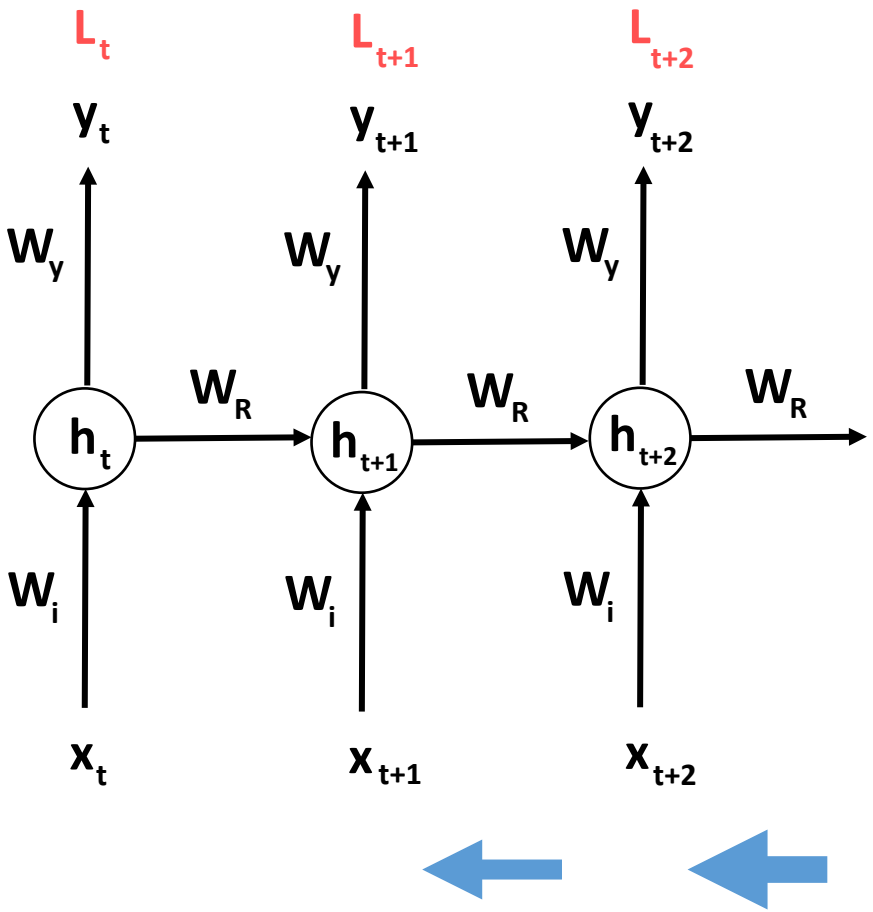
We multiply the weights several times: if you multiply **x<1** several times the result will get smaller and smaller

VANISHING GRADIENT PROBLEM

Recurrent Neural Networks

Vanishing/exploding gradients problem

When dealing with backpropagation we have to calculate the gradients



$$\frac{\partial L}{\partial W_R} = \sum_t \frac{\partial L_t}{\partial W_R}$$

$$\frac{\partial L_t}{\partial W_R} = \sum_t \left(\frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial w} \right)$$

~ we just have to apply the chain rule several times

$$\frac{\partial L_t}{\partial W_R} = \prod_{t>i>k} W_R^T \, \mathrm{diag}(\sigma'(x_{i-1}))$$

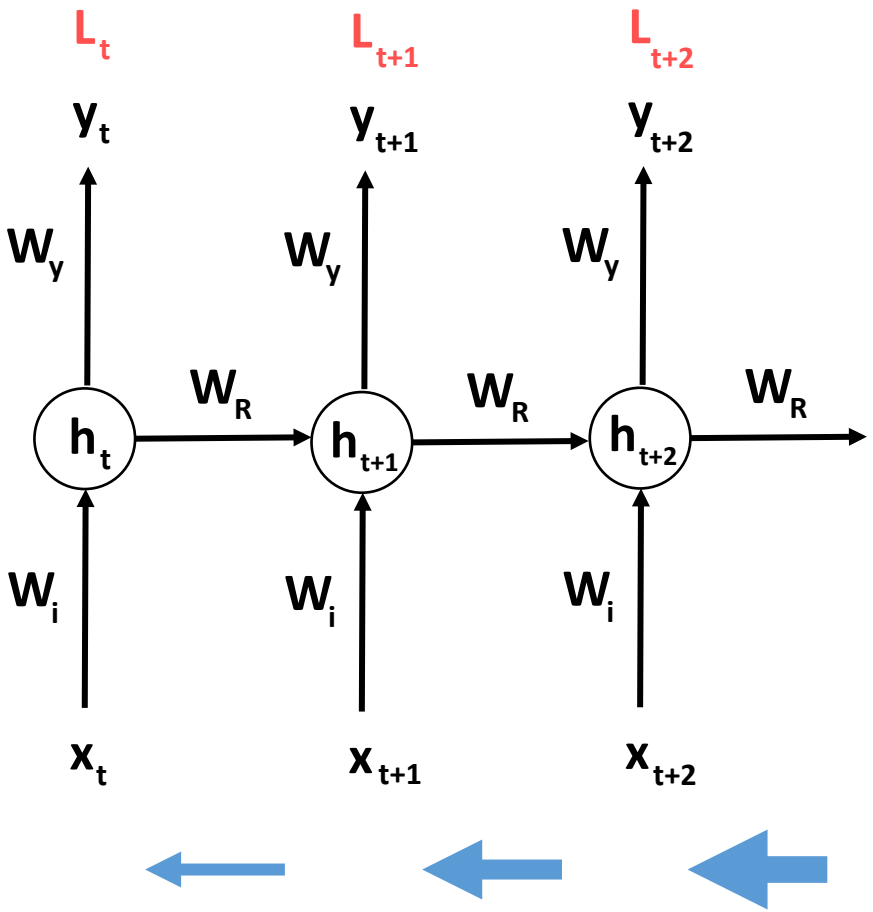
We multiply the weights several times: if you multiply **x<1** several times the result will get smaller and smaller

VANISHING GRADIENT PROBLEM

Recurrent Neural Networks

Vanishing/exploding gradients problem

When dealing with backpropagation we have to calculate the gradients



$$\frac{\partial L}{\partial W_R} = \sum_t \frac{\partial L_t}{\partial W_R}$$

$$\frac{\partial L_t}{\partial W_R} = \sum_t \left(\frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial w} \right)$$

~ we just have to apply the chain rule several times

$$\frac{\partial L_t}{\partial W_R} = \prod_{t>i>k} W_R^T \, \mathrm{diag}(\sigma'(x_{i-1}))$$

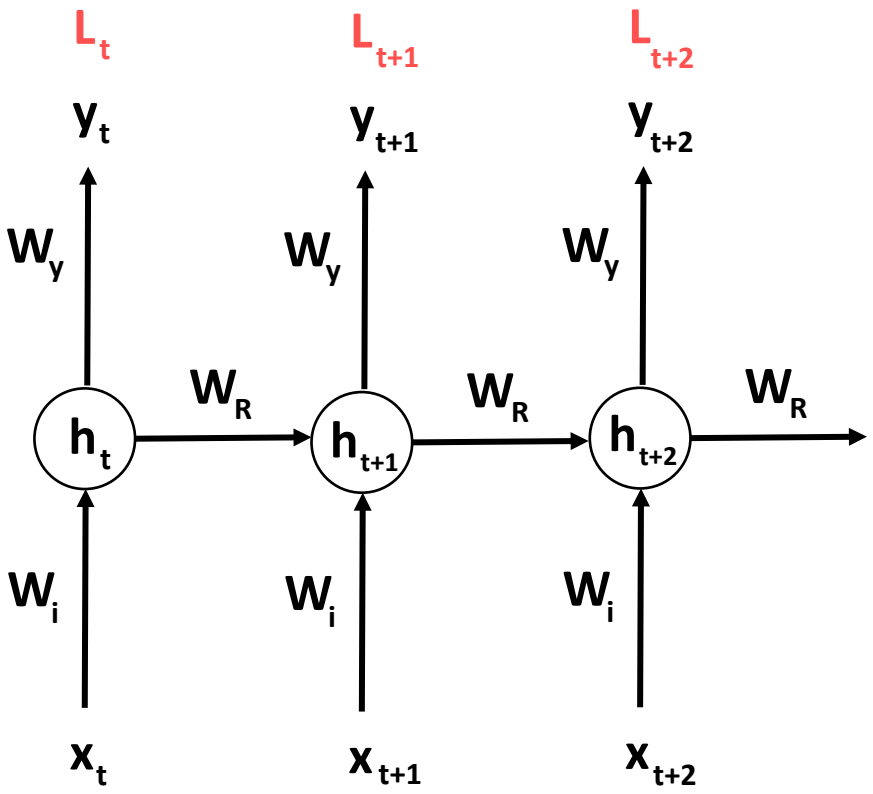
We multiply the weights several times: if you multiply **x<1** several times the result will get smaller and smaller

VANISHING GRADIENT PROBLEM

Recurrent Neural Networks

Vanishing/exploding gradients problem

When dealing with backpropagation we have to calculate the gradients



$$\frac{\partial L}{\partial W_R} = \sum_t \frac{\partial L_t}{\partial W_R}$$

$$\frac{\partial L_t}{\partial W_R} = \sum_t \left(\frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial w} \right)$$

~ we just have to apply the chain rule several times

$$\frac{\partial L_t}{\partial W_R} = \prod_{t>i>k} W_R^T \, \mathrm{diag}(\sigma'(x_{i-1}))$$

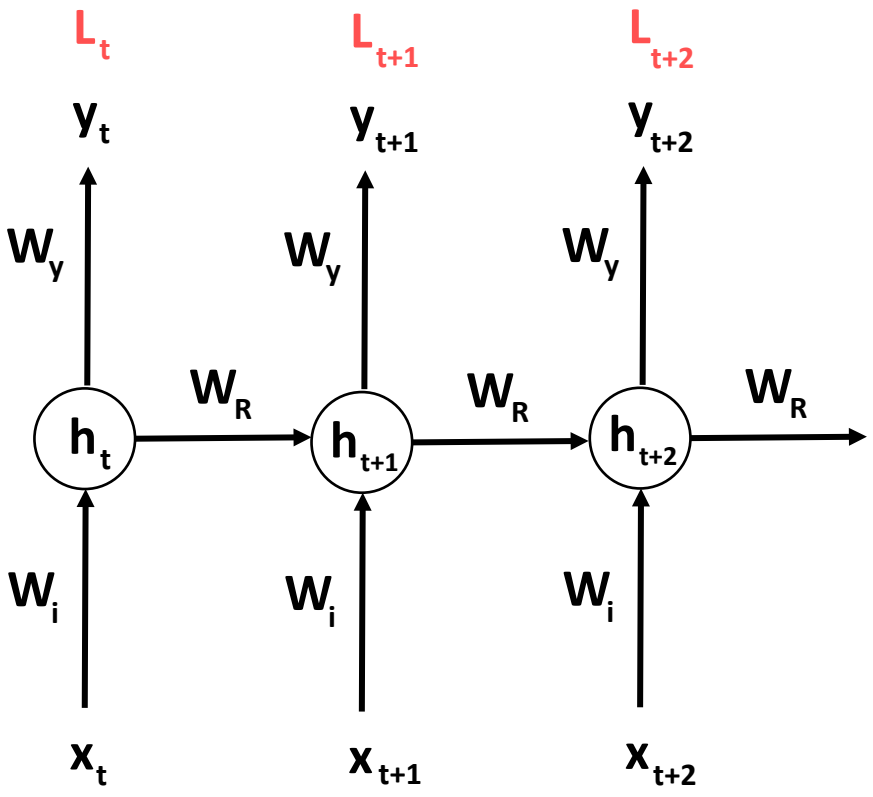
We multiply the weights several times: if you multiply **x>1** several times the result will get bigger and bigger

EXPLODING GRADIENT PROBLEM

Recurrent Neural Networks

Vanishing/exploding gradients problem

When dealing with backpropagation we have to calculate the gradients



$$\frac{\partial L}{\partial W_R} = \sum_t \frac{\partial L_t}{\partial W_R}$$

$$\frac{\partial L_t}{\partial W_R} = \sum_t \left(\frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial w} \right)$$

~ we just have to apply the chain rule several times

$$\frac{\partial L_t}{\partial W_R} = \prod_{t>i>k} W_R^T \, \mathrm{diag}(\sigma'(x_{i-1}))$$

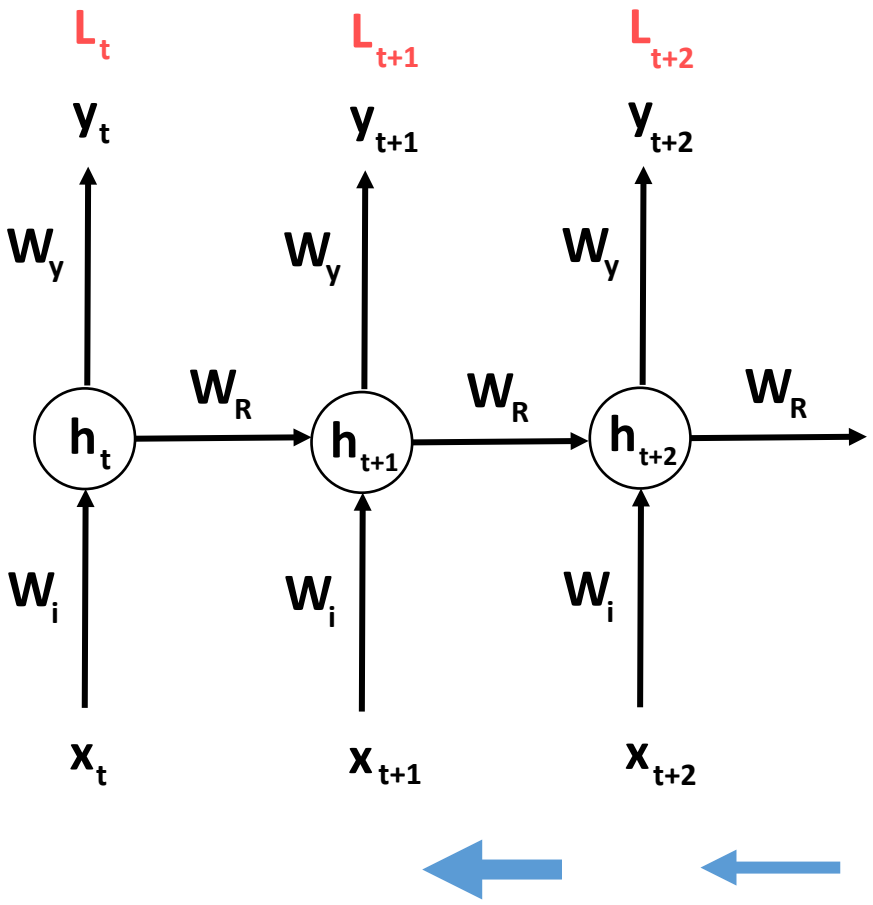
We multiply the weights several times: if you multiply **x>1** several times the result will get bigger and bigger

EXPLODING GRADIENT PROBLEM

Recurrent Neural Networks

Vanishing/exploding gradients problem

When dealing with backpropagation we have to calculate the gradients



$$\frac{\partial L}{\partial W_R} = \sum_t \frac{\partial L_t}{\partial W_R}$$

$$\frac{\partial L_t}{\partial W_R} = \sum_t \left(\frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial w} \right)$$

~ we just have to apply the chain rule several times

$$\frac{\partial L_t}{\partial W_R} = \prod_{t>i>k} W_R^T \ \mathbf{diag}(\sigma'(x_{i-1}))$$

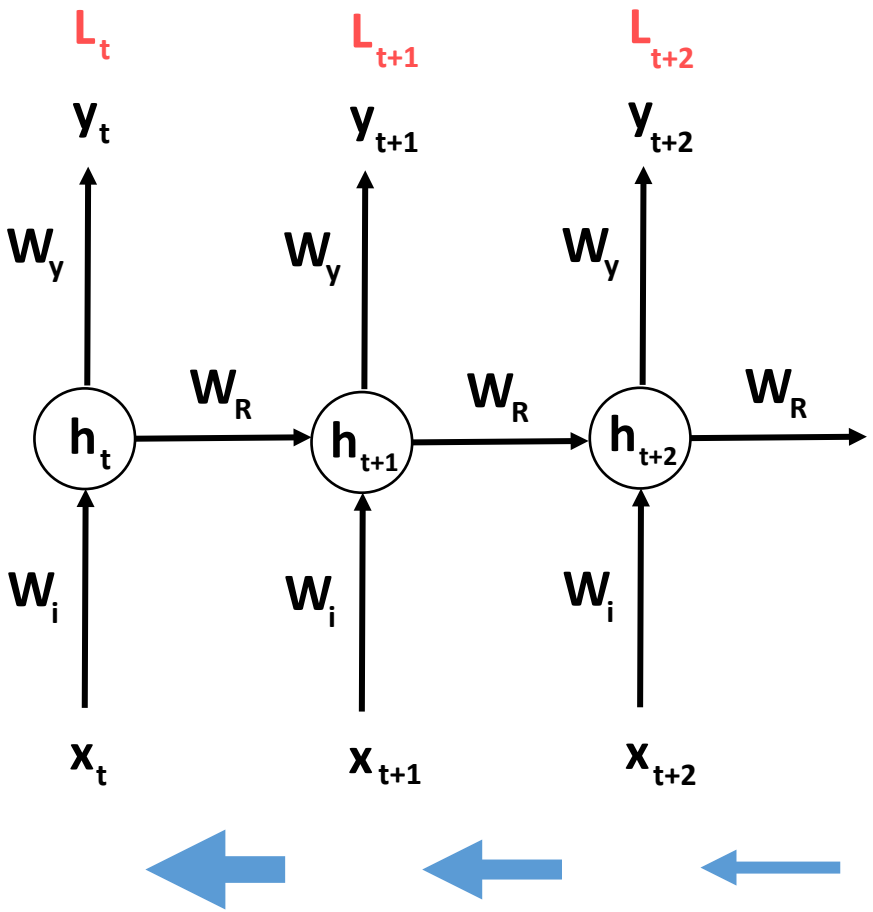
We multiply the weights several times: if you multiply **x>1** several times the result will get bigger and bigger

EXPLODING GRADIENT PROBLEM

Recurrent Neural Networks

Vanishing/exploding gradients problem

When dealing with backpropagation we have to calculate the gradients



$$\frac{\partial L}{\partial W_R} = \sum_t \frac{\partial L_t}{\partial W_R}$$

$$\frac{\partial L_t}{\partial W_R} = \sum_t \left(\frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial w} \right)$$

~ we just have to apply the chain rule several times

$$\frac{\partial L_t}{\partial W_R} = \prod_{t>i>k} W_R^T \, \mathrm{diag}(\sigma'(x_{i-1}))$$

We multiply the weights several times: if you multiply **x>1** several times the result will get bigger and bigger

EXPLODING GRADIENT PROBLEM

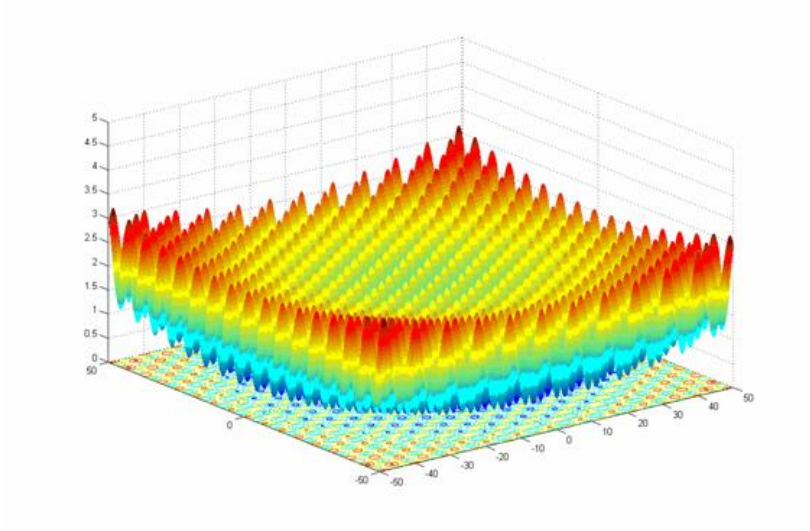
Recurrent Neural Networks

Vanishing/exploding gradients problem

It is a problem when dealing with Recurrent Neural Networks usually
~ because these networks are usually deep !!!

→ why vanishing gradient is a problem?
Because gradients become too small: difficult to model
long-range dependencies

→ for recurrent neural networks, local optima are a much more significant problem
than with feed-forward neural networks
~ error function surface is quite complex



These complex surfaces have several local optima and we want to find
the global one: we can use **meta-heuristic** approaches as well

Recurrent Neural Networks

How to deal with vanishing/exploding gradients problem?

EXPLODING GRADIENTS PROBLEM

→ truncated **BPTT** algorithm: we use simple backpropagation but
We only do backpropagation through **k** time-steps

→ adjust the learning rate with **RMSProp** (adaptive algorithm)
We normalize the gradients: using moving average over the root mean squared gradients

VANISHING GRADIENTS PROBLEM

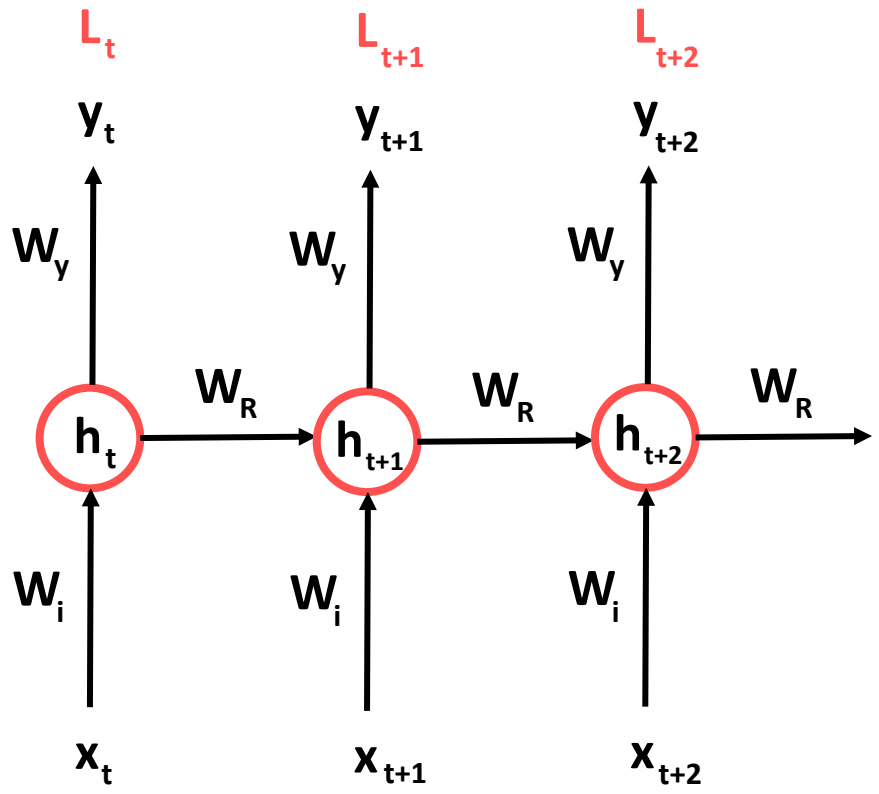
→ initialize the weights properly (Xavier-initialization)

harder to detect

→ proper activation functions such as **ReLU** function

→ using other architectures: **LSTM** or **GRU**s

Long-Short Term Memory (LSTM)



Instead of the **h** units: we want to add some memory to the neural network + we want to manipulate these memory cells

→ flush the memory - **FORGET GATE**

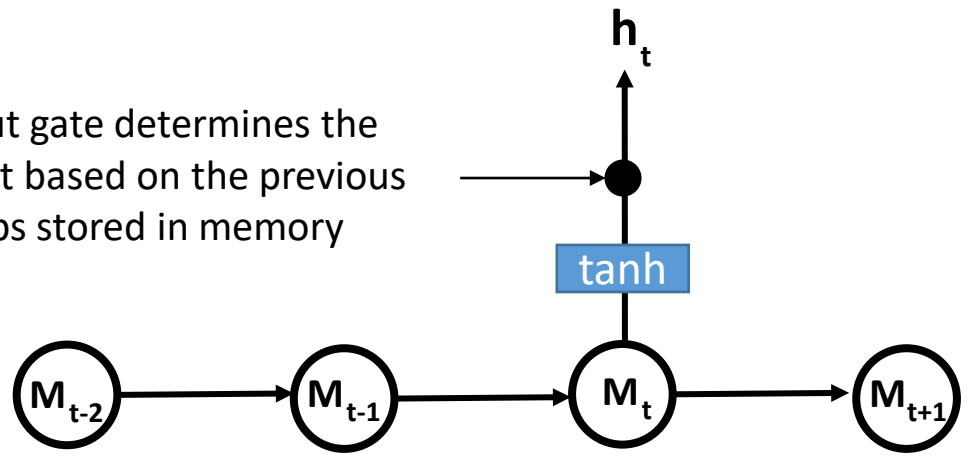
→ add to memory - **INPUT GATE**

→ read from memory - **OUTPUT GATE**

Long-Short Term Memory (LSTM)

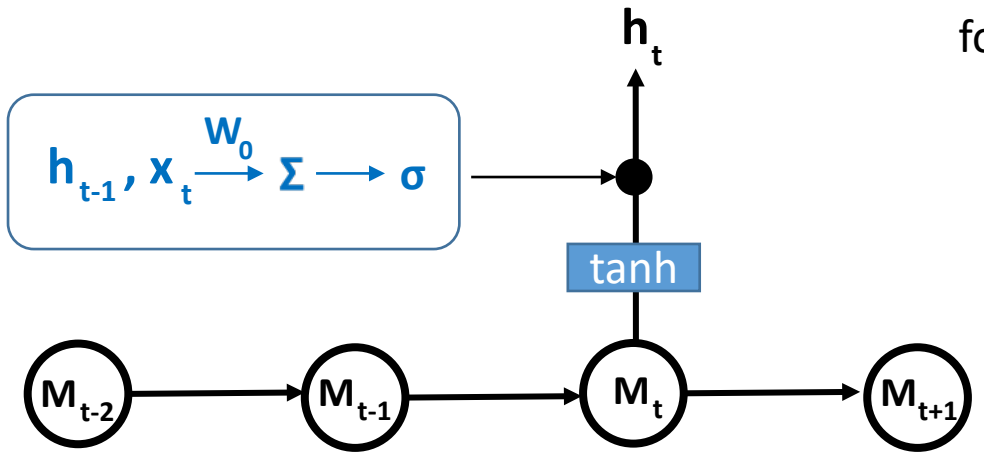
OUTPUT GATE

output gate determines the **h** output based on the previous steps stored in memory



Long-Short Term Memory (LSTM)

OUTPUT GATE



It is important to use the sigmoid activation function for the output gate: we transform the values within the range **[0,1]**

~ we can manipulate the values in the memory

$$h_t = \sigma\ (\ W_0\ [\ x_t, h_{t-1}] + b_0) \bullet \tanh(M_t)$$

element-wise multiplication

0 → we do not care about the information present in the memory

1 → we take all the information present in the memory

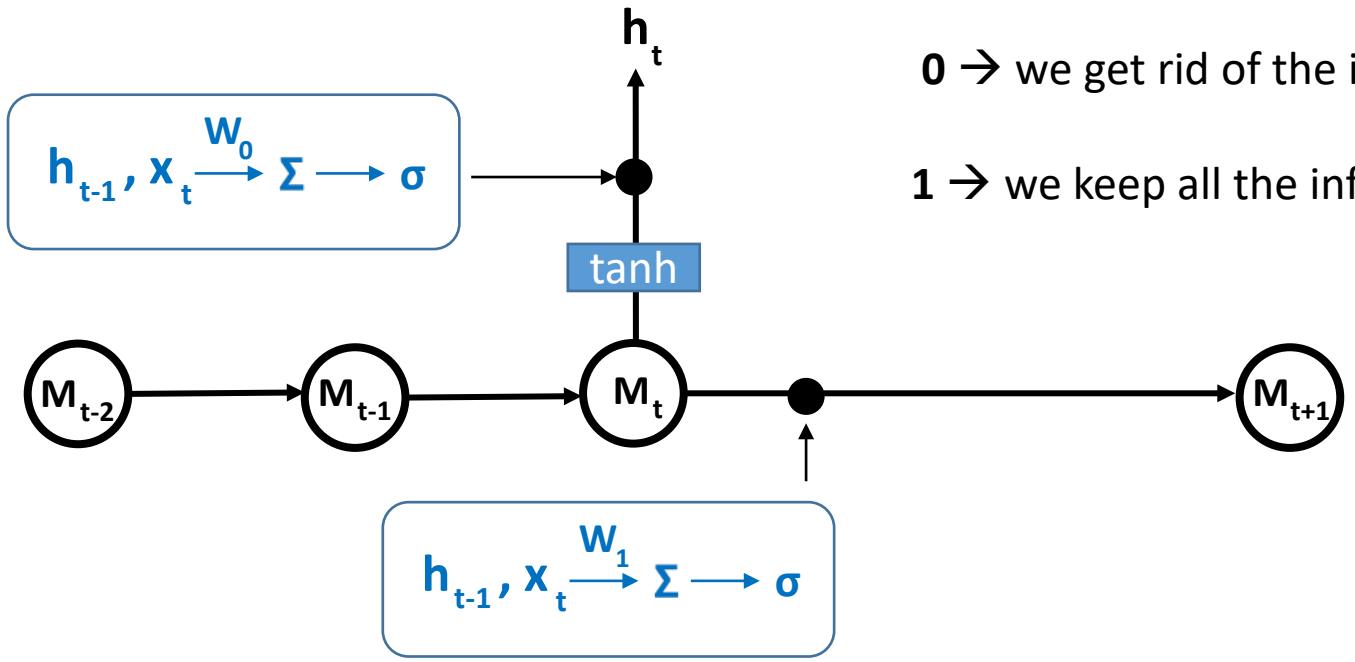
Long-Short Term Memory (LSTM)

FORGET GATE

With forget gate we manipulate not the output but the content of the memory !!!

0 → we get rid of the information present in memory

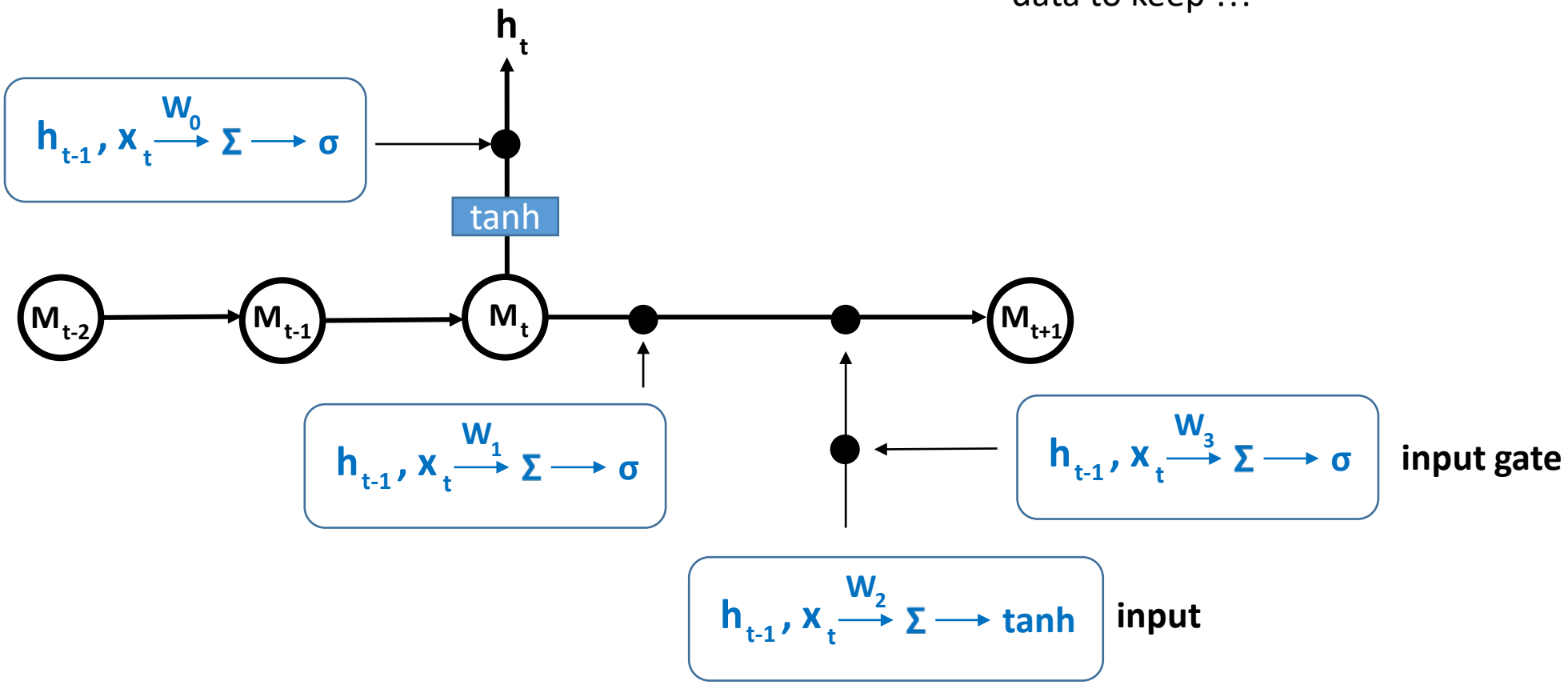
1 → we keep all the information present in memory



Long-Short Term Memory (LSTM)

INPUT GATE

With input gate we can write new data to the memory and with another gate we can control what data to keep !!!



Gated Recurrent Units (GRU)

These units are a simplified **LSTM** blocks

~ all the gates are included in a single update gate

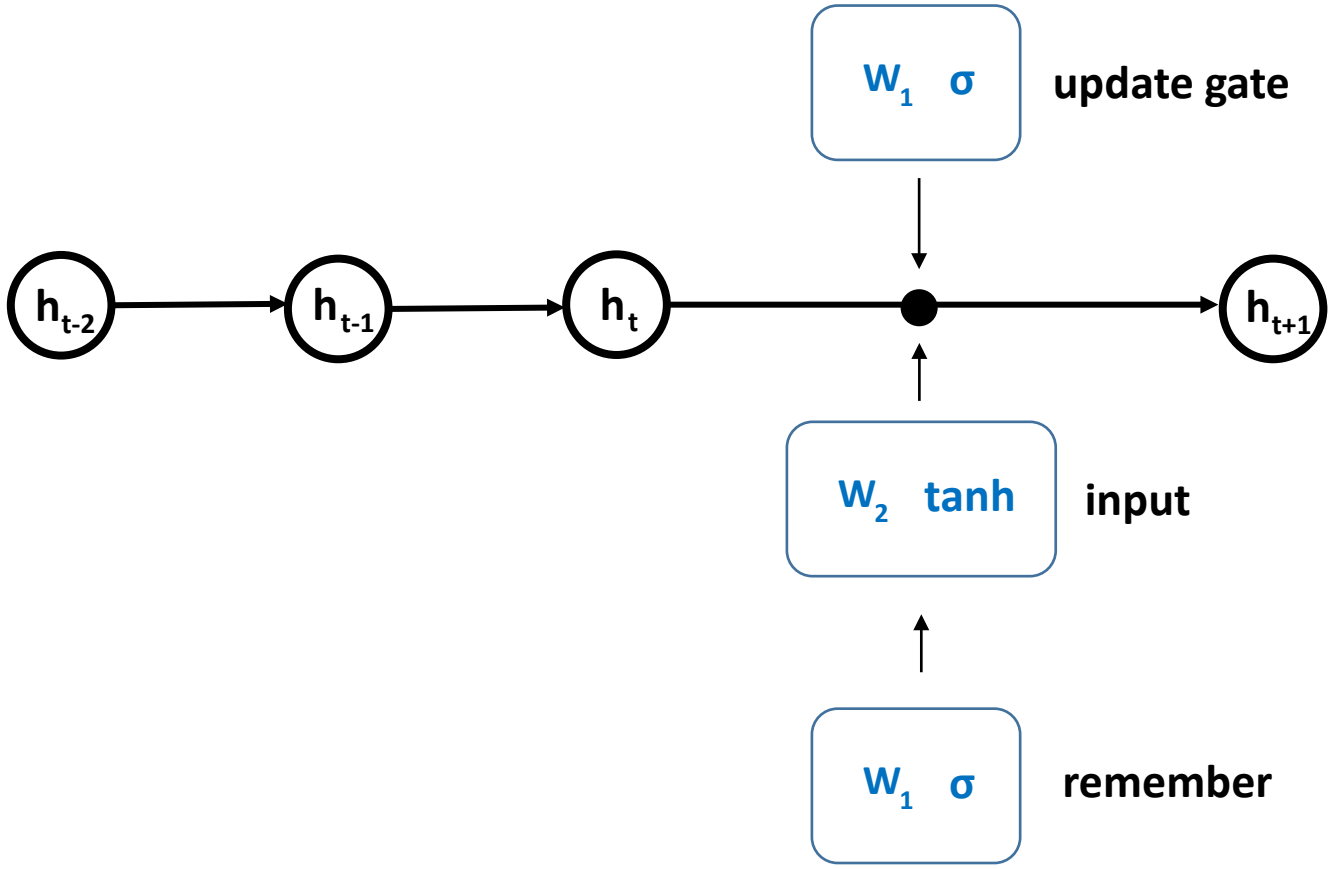
Why to use GRUs? In order to cope with vanishing gradient problem !!!

→ the **GRU** unit controls the flow of information like the **LSTM** unit, but without having to use a memory unit

~ it just exposes the full hidden content without any control

→ quite a simple model: more efficient becasuse the architecture is simple

Gated Recurrent Units (GRU)



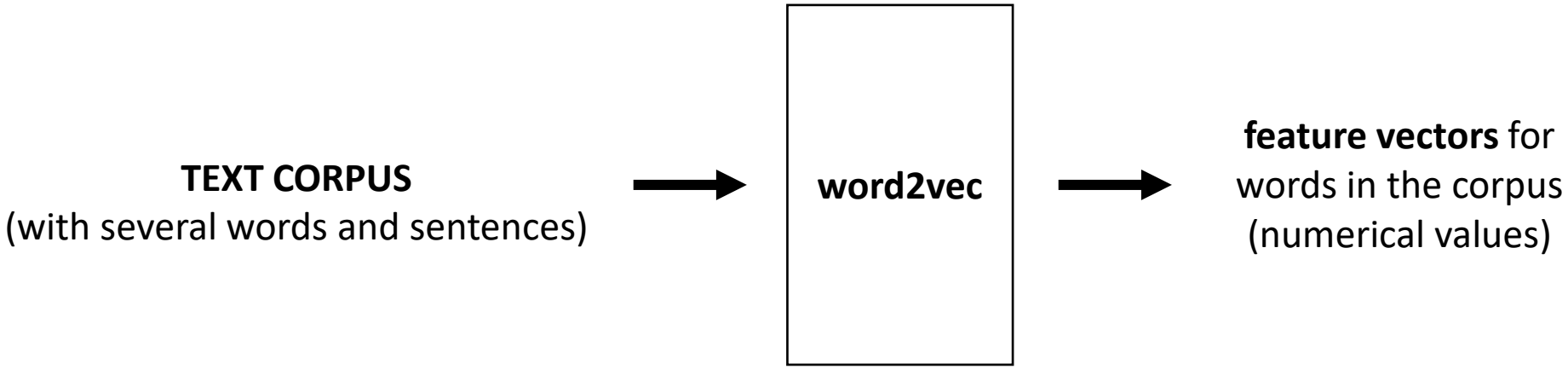
controls the input itself

controls how much previous steps impact the new step

Natural Language Processing (NLP)

The most apparent problem when dealing with natural language processing,
is how to transform characters/strings/texts into numerical data

→ of course algorithms can deal with numerical values exclusively
(machine learning algorithms or artificial intelligence)



Natural Language Processing (NLP)

WHY IS word2vec GOOD?

→ it groups the vectors of similar words together in the vectorspace
~ they are clustered in the same region (k-means clustering)
For example: man, boy, girl, woman

These clusters form the basis of sentiment analysis !!!

→ it is able to detect similarities mathematically

→ with large amount of training data, it can make highly accurate guesses about a word's meaning based on past appearances

Natural Language Processing (NLP)

WHY IS word2vec GOOD?

TEXT CORPUS
(with several words and sentences)

→

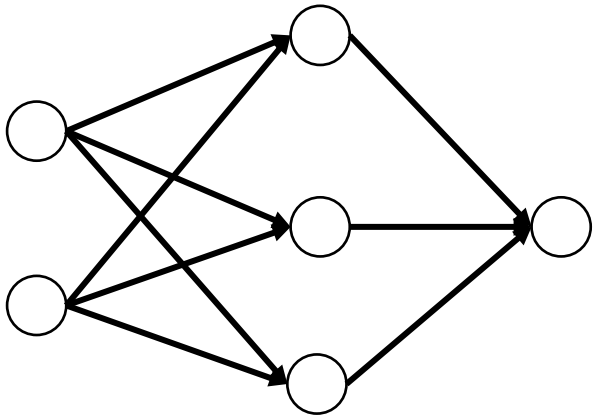
word2vec

two-layer neural net

→

feature vectors for words in the corpus (numerical values)

→



USE DEEP NEURAL NETWORKS TO DETECT RELATIONSHIPS BETWEEN WORDS

Natural Language Processing (NLP)

We can define the cosine similarity of given word vectors: no similarity is expressed as a **90** degree angle, while total similarity of **1** is a **0** degree angle, complete overlap

What are the similar words associated with „Sweden”?

Word	Cosine distance
norway	0.760124
denmark	0.715460
finland	0.620022
switzerland	0.588132
belgium	0.585835
netherlands	0.574631
iceland	0.562368
estonia	0.547621
slovenia	0.531408

Natural Language Processing (NLP)

There are two algorithms:

1.) continuous bag of words (**CBOW**)

This algorithm uses context to predict the target word

„The beautiful city of **Washington** is the capital of the United States"

2.) Skip-Gram method:

This algorithm uses a single word to predict the target context

~ it produces more accurate results

„**The beautiful city of** Washington **is the capital of the United States**"

Skim-Gram Model (NLP)

We are going to train a simple neural network with just a single
hidden layer BUT we will not use this network for further operations

THE GOAL IS TO LEARN THE WEIGHTS IN THE HIDDEN LAYER

~ weights are the word vectors

→ so given a specific word in the middle of a sentence: the network is going to
tells us the probability for every word in our vocabulary of being
the „nearby word”

→ we have to define a **window size**: parameter to the algorithm
How many nearby words to consider

Skim-Gram Model (NLP)

„Quantum physics is a fundamental theory in physics”

(**window size=1** so we consider one word before and one word after the actual one)

Skim-Gram Model (NLP)

„Quantum physics is a fundamental theory in physics”

(**window size=1** so we consider one word before and one word after the actual one)

Quantum physics is a fundamental theory in physics

Skim-Gram Model (NLP)

„Quantum physics is a fundamental theory in physics”

(**window size=1** so we consider one word before and one word after the actual one)

TRAINING SAMPLES

Quantum	physics	is	a	fundamental	theory	in	physics	[quantum,physics]

Skim-Gram Model (NLP)

„Quantum physics is a fundamental theory in physics”

(**window size=1** so we consider one word before and one word after the actual one)

	TRAINING SAMPLES
Quantum physics is a fundamental theory in physics	[quantum,physics]
Quantum physics is a fundamental theory in physics	[physics,quantum], [physics,is]

Skim-Gram Model (NLP)

„Quantum physics is a fundamental theory in physics”

(**window size=1** so we consider one word before and one word after the actual one)

	TRAINING SAMPLES
Quantum **physics** is a fundamental theory in physics	[quantum,physics]
Quantum **physics** is a fundamental theory in physics	[physics,quantum], [physics,is]
Quantum physics **is** a fundamental theory in physics	[is,physics], [is,a]

Skim-Gram Model (NLP)

„Quantum physics is a fundamental theory in physics”

(**window size=1** so we consider one word before and one word after the actual one)

								TRAINING SAMPLES
Quantum	physics	is	a	fundamental	theory	in	physics	[quantum,physics]
Quantum	physics	is	a	fundamental	theory	in	physics	[physics,quantum], [physics,is]
Quantum	physics	is	a	fundamental	theory	in	physics	[is,physics], [is,a]
Quantum	physics	is	a	fundamental	theory	in	physics	[a,is], [a,fundamental]

Skim-Gram Model (NLP)

„Quantum physics is a fundamental theory in physics"

(**window size=1** so we consider one word before and one word after the actual one)

								TRAINING SAMPLES
Quantum	physics	is	a	fundamental	theory	in	physics	[quantum,physics]
Quantum	physics	is	a	fundamental	theory	in	physics	[physics,quantum], [physics,is]
Quantum	physics	is	a	fundamental	theory	in	physics	[is,physics], [is,a]
Quantum	physics	is	a	fundamental	theory	in	physics	[a,is], [a,fundamental]

The neural networks can learn the statistics form the number of times each paring shows up

For example: **New** and **York** will be right next to each other in several training samples

Skim-Gram Model (NLP)

Of course, first we have to transform the text (words and sentences) into numerical format: because we need the input for the neural network

„The sun is shining"

0 – is

1 – shining

2 – sun

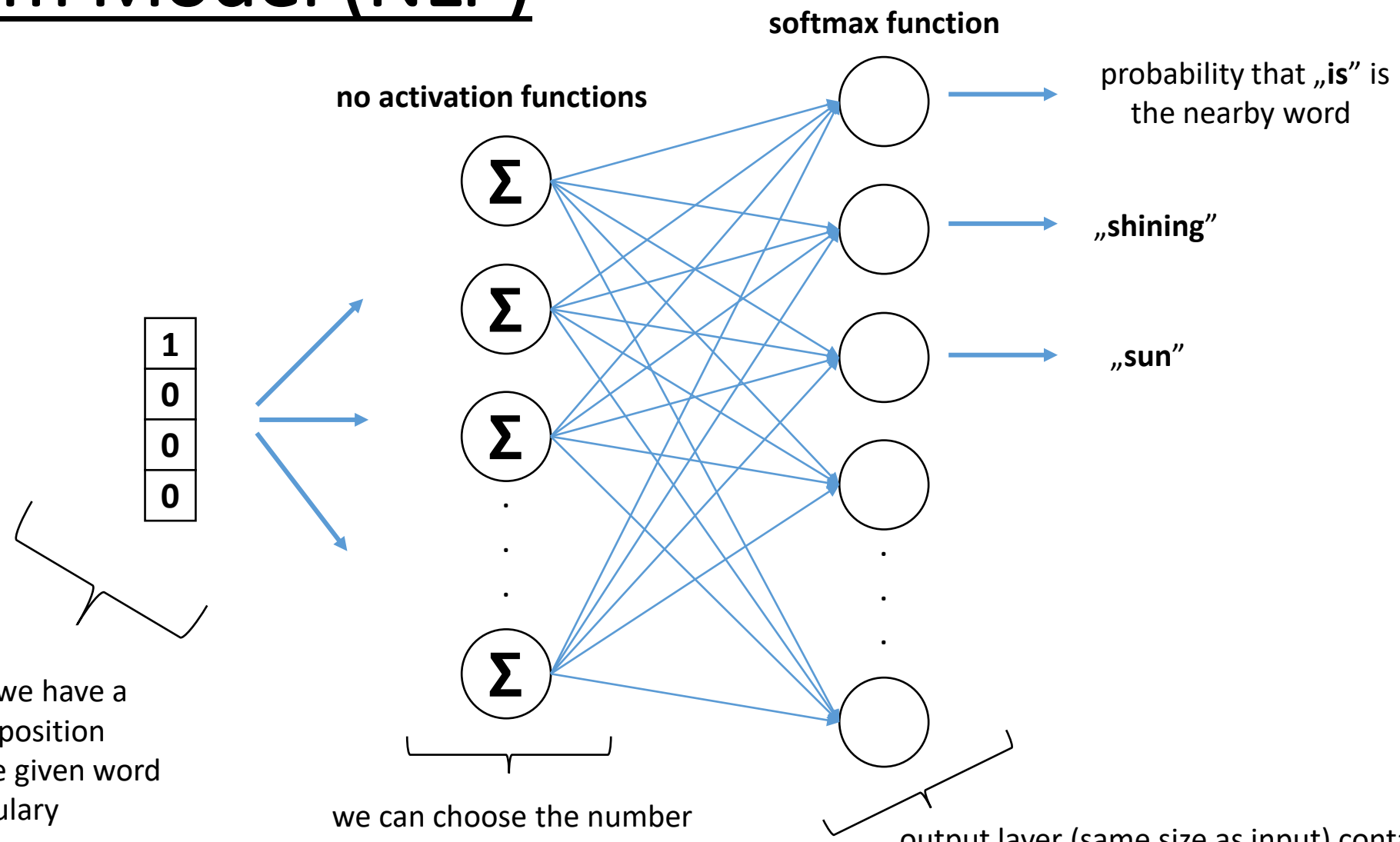
3 – the

So we can build a vocabulary of words from our training text corpus
~ in this case we have just a single sentence

WE CAN REPRESENT AN INPUT WITH A VECTOR !!!

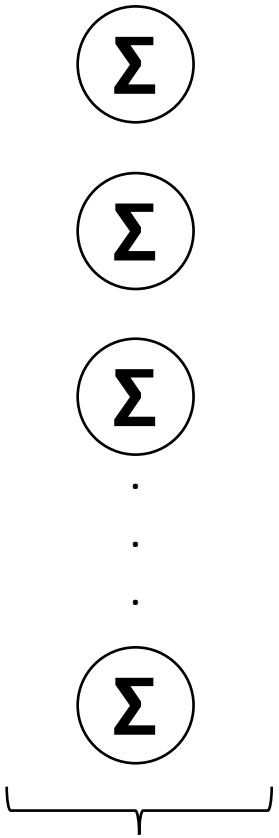
is	shining	sun	the
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Skim-Gram Model (NLP)



Skim-Gram Model (NLP)

no activation functions



we can choose the number
of hidden neurons
(**300 for GOOGLE NEWS dataset**)

So the aim is to transform word in a text into numerical values

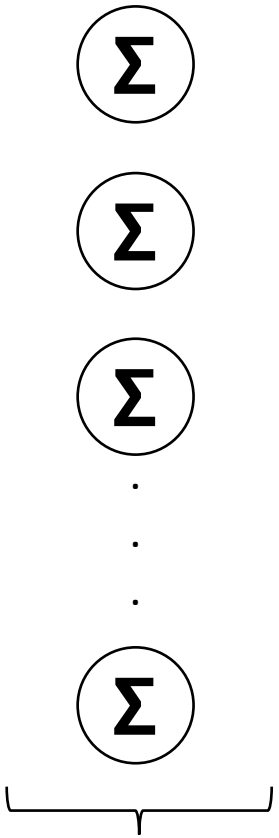
→ if we have **300** neurons in the hidden layer: it means the word vectors size will be **300**

→ so we will associate **300** floating point numbers to every single word in the text

hidden layer wight matrix = word vectors

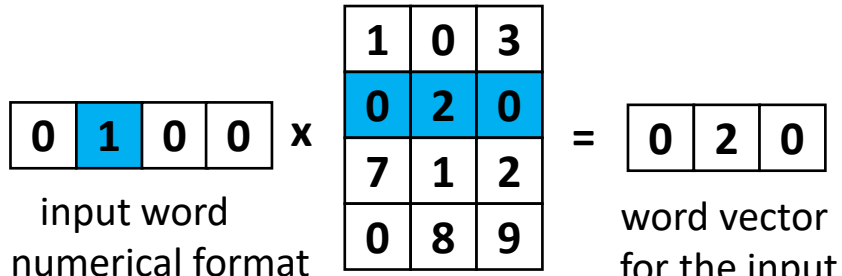
Skim-Gram Model (NLP)

no activation functions



we can choose the number
of hidden neurons
(**300 for GOOGLE NEWS dataset**)

Basically this neural network does a lookup (like hashtables in Java)
in the hidden layer



→ words with similar meanings will
have similar word vectors
For example: boy, guy, man

Data Augmentation

We need a huge dataset for deep learning
What if we don't have a huge datset?

→ we can use data augmentation (it reduces overfitting)

→ let's apply random trandformations on the images
(rotations, flipping, scaling ...)

THIS IS HOW WE CAN MAKE SEVERAL IMAGES !!!

→ because of data augmentation the learning algorithm never uses
tha same image twice
~ there will not be overfitting

→ solves a huge problem: usually there is no big dataset
to train the algorithm on (data augmentation solves it)