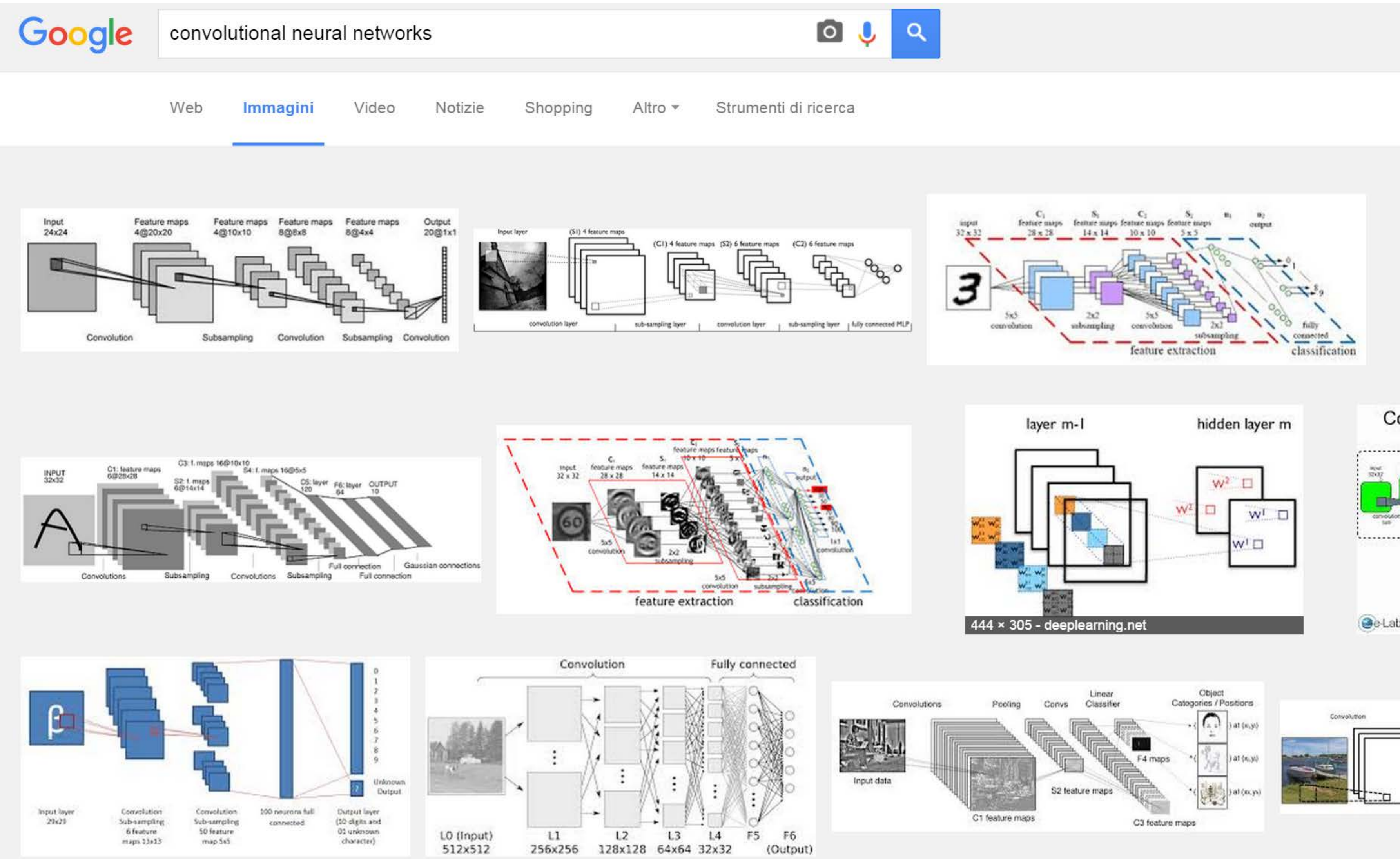
Convolutional Neural Networks

Francesco Ciompi, Radboudumc

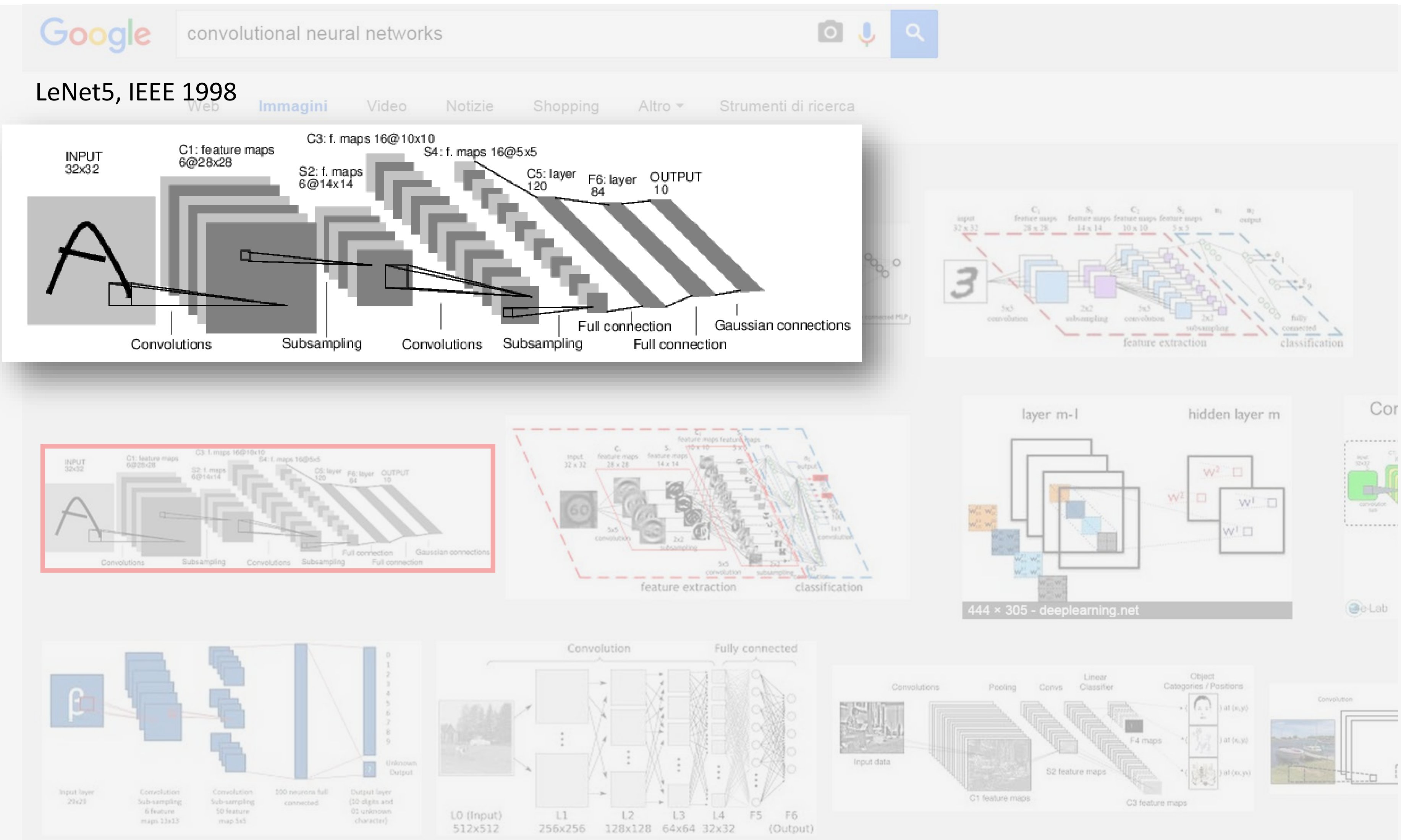
NFBIA Summer School 2015
September 15, 2015, Nijmegen

Convolutional Neural Networks



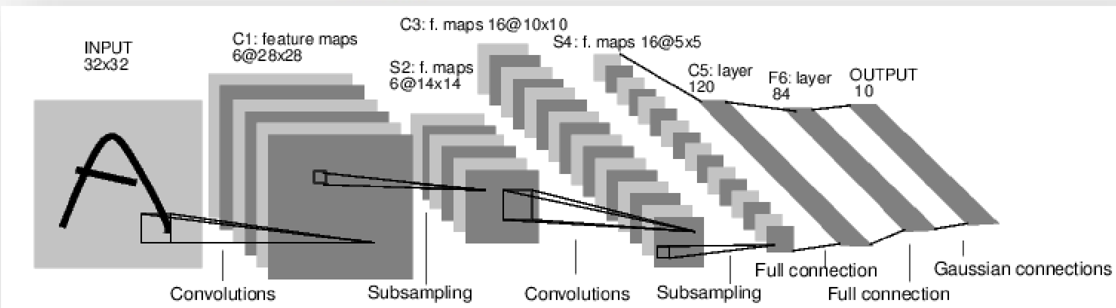
Convolutional Neural Networks

LeNet5, IEEE 1998

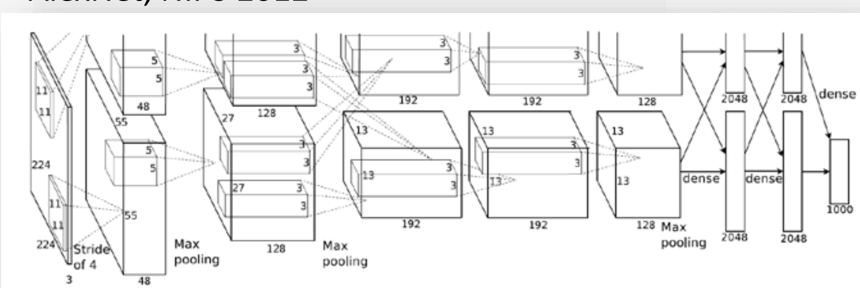


Convolutional Neural Networks

LeNet5, IEEE 1998

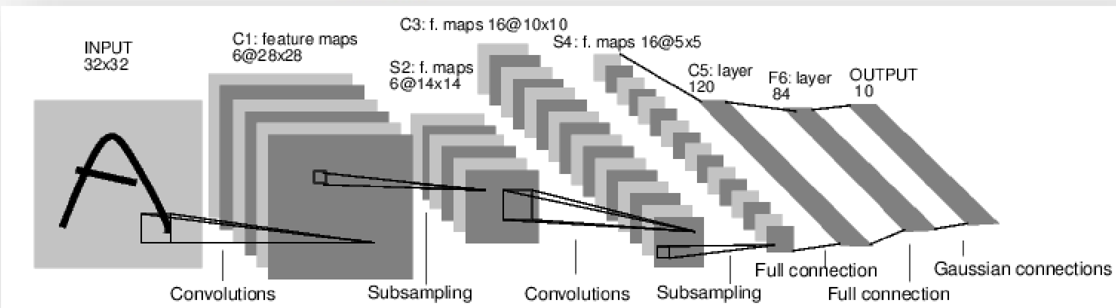


AlexNet, NIPS 2012

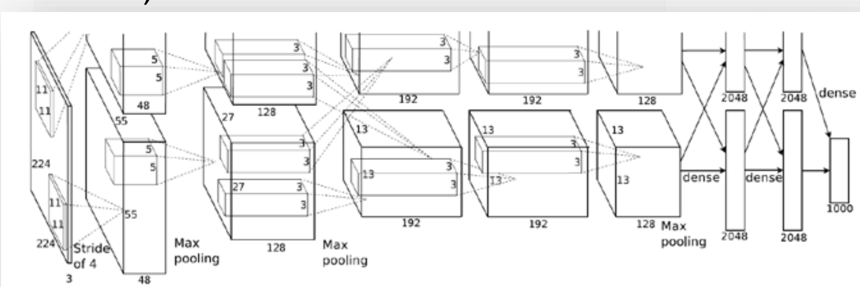


Convolutional Neural Networks

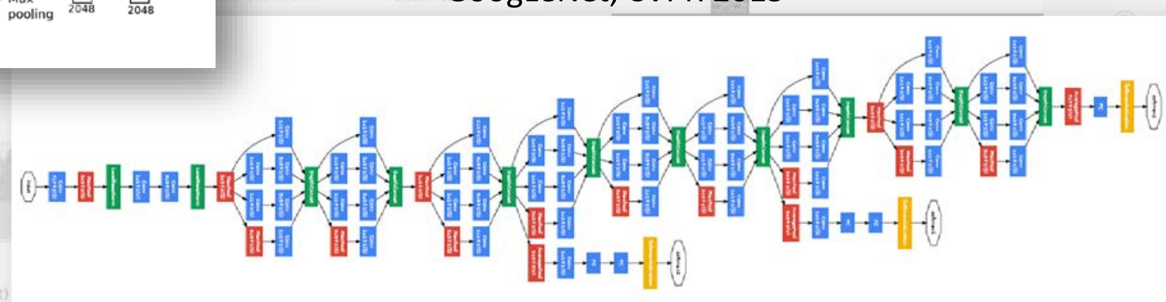
LeNet5, IEEE 1998



AlexNet, NIPS 2012



GoogLeNet, CVPR 2015



Convolutional

Convolutional

Neural

Convolutional
Neural
Networks

Convolutional

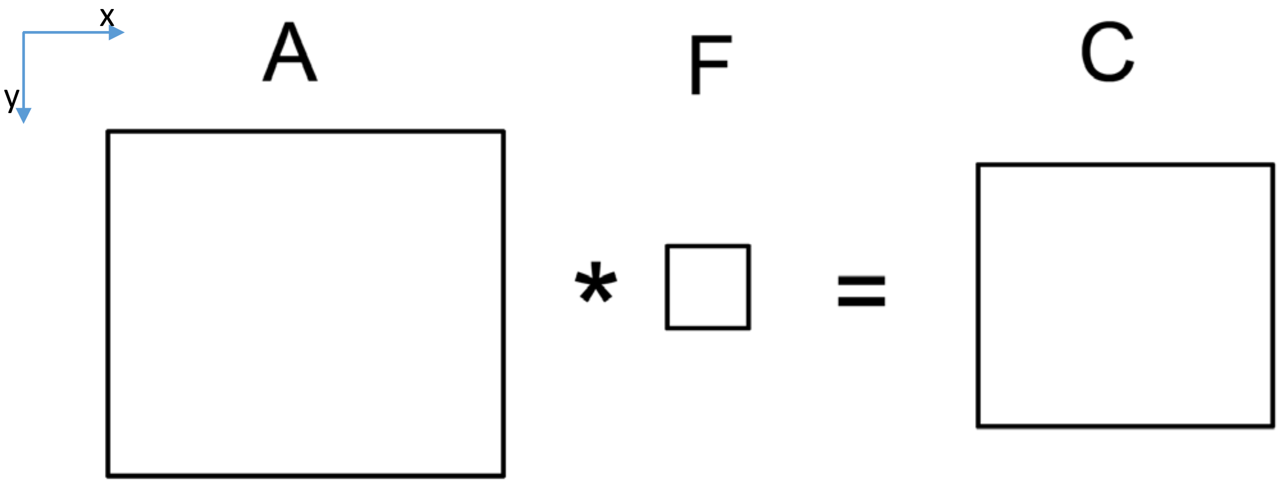
Neural

Networks

CNN ≠ CNN

ConvNets

Convolutional Neural Network



A: MxN input channel

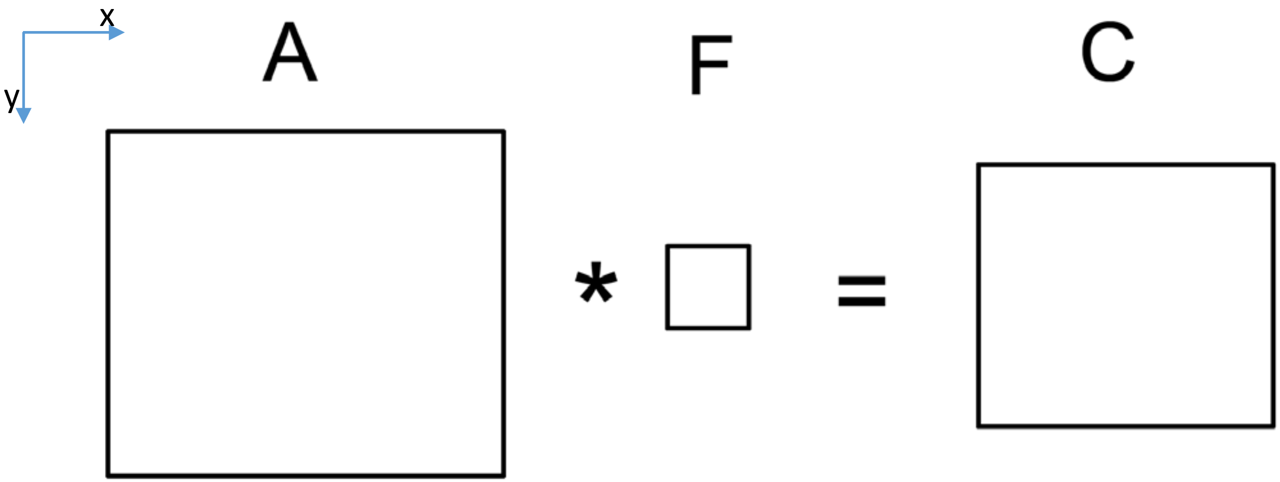
F: mxn filter (receptive field)

C: output channel

Convolution: C = A*F

$$C(i,j) = \sum_{x=0}^{m-1} \sum_{y=0}^{n-1} A(x-i, y-j) F(x,y)$$

Convolutional Neural Network



A: MxN input channel

F: mxn filter (receptive field)

C: output channel

Convolution: C = A*F

$$C(i,j) = \sum_{x=0}^{m-1} \sum_{y=0}^{n-1} A(x-i, y-j)F(x,y)$$

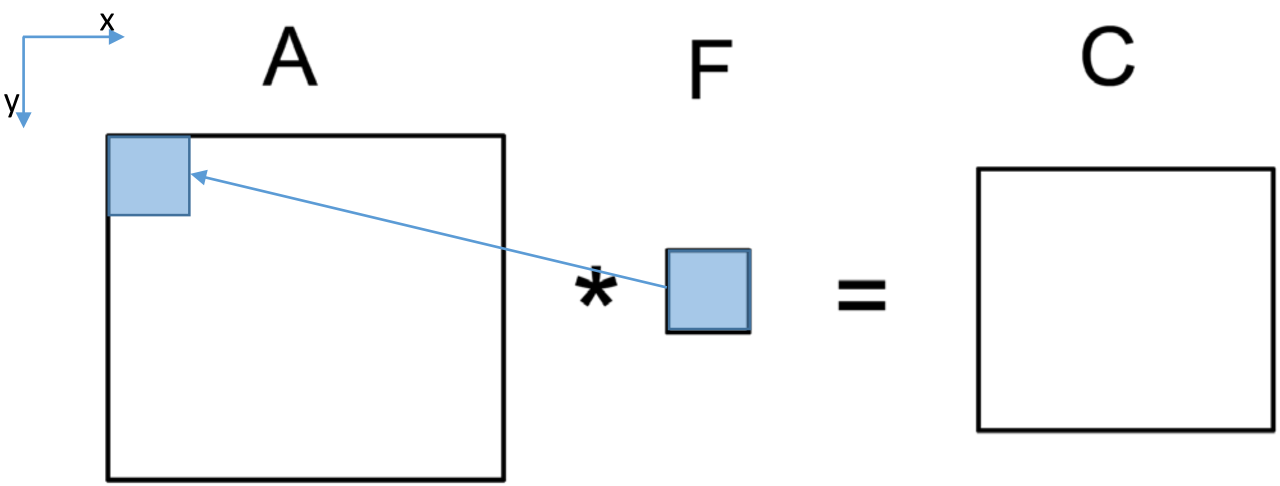
associative: (A*(B*F) = (A*B)*F)

Correlation

$$C(i,j) = \sum_{x=0}^{m-1} \sum_{y=0}^{n-1} A(x+i, y+j)F(x,y)$$

not associative

Convolutional Neural Network



A: MxN input channel

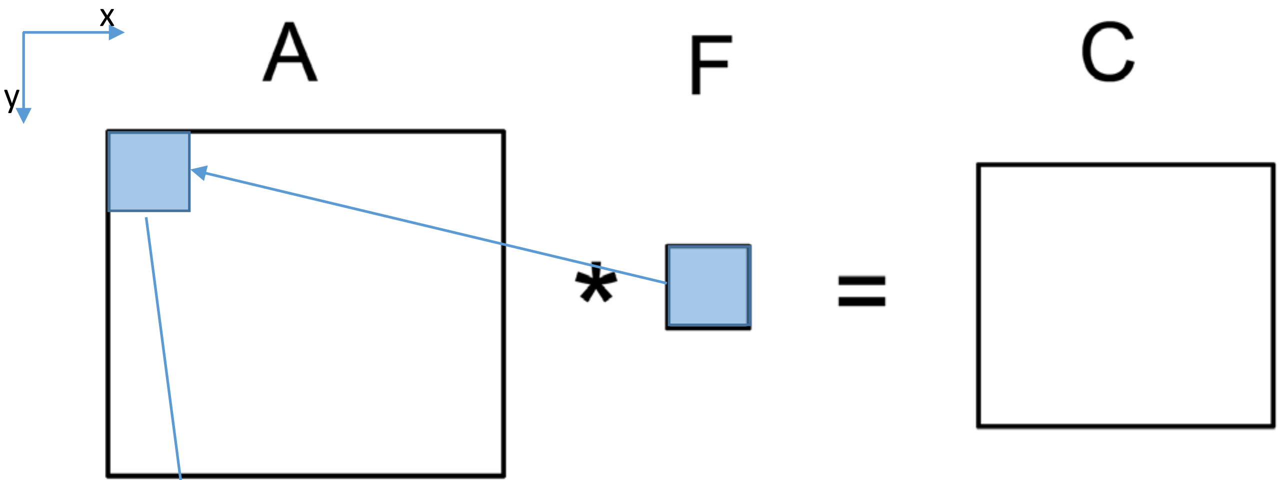
F: mxn filter (receptive field)

C: output channel

Convolution

$$C(i,j) = \sum_{x=0}^{m-1} \sum_{y=0}^{n-1} A(x-i, y-j) F(x,y)$$

Convolutional Neural Network



A: MxN input channel

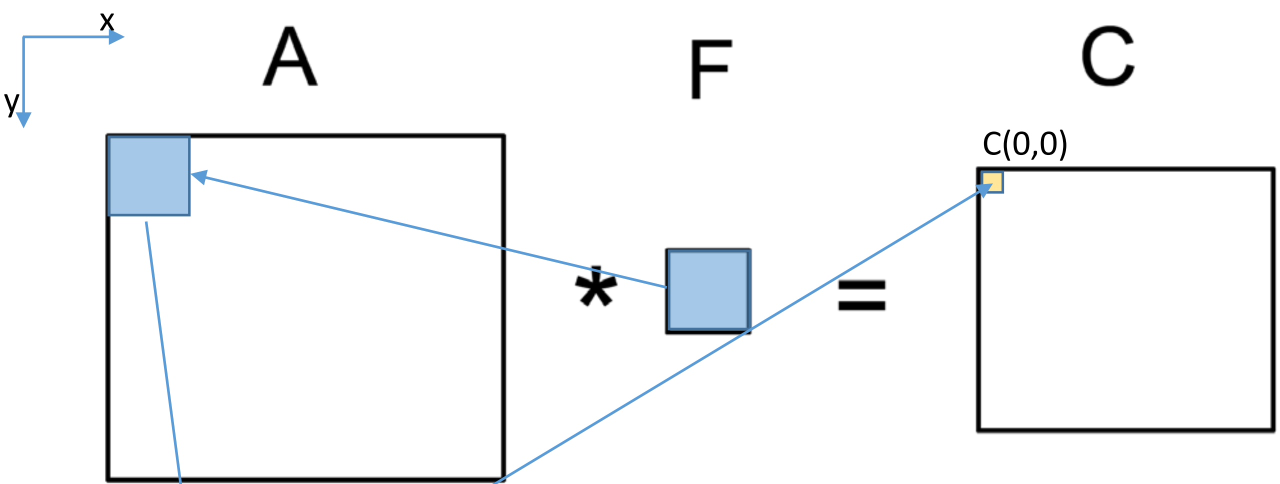
F: mxn filter (receptive field)

C: output channel

Convolution

$$C(i,j) = \sum_{x=0}^{m-1} \sum_{y=0}^{n-1} A(x-i, y-j)F(x,y)$$

Convolutional Neural Network



A: MxN input channel

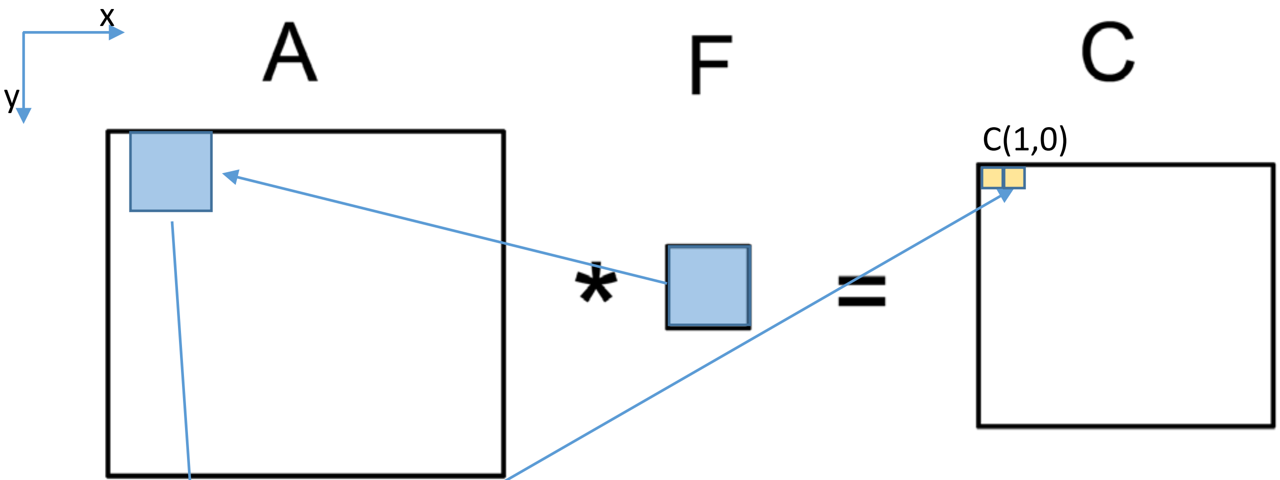
F: mxn filter (receptive field)

C: output channel

Convolution

$$C(i,j) = \sum_{x=0}^{m-1} \sum_{y=0}^{n-1} A(x-i, y-j) F(x,y)$$

Convolutional Neural Network



A: MxN input channel

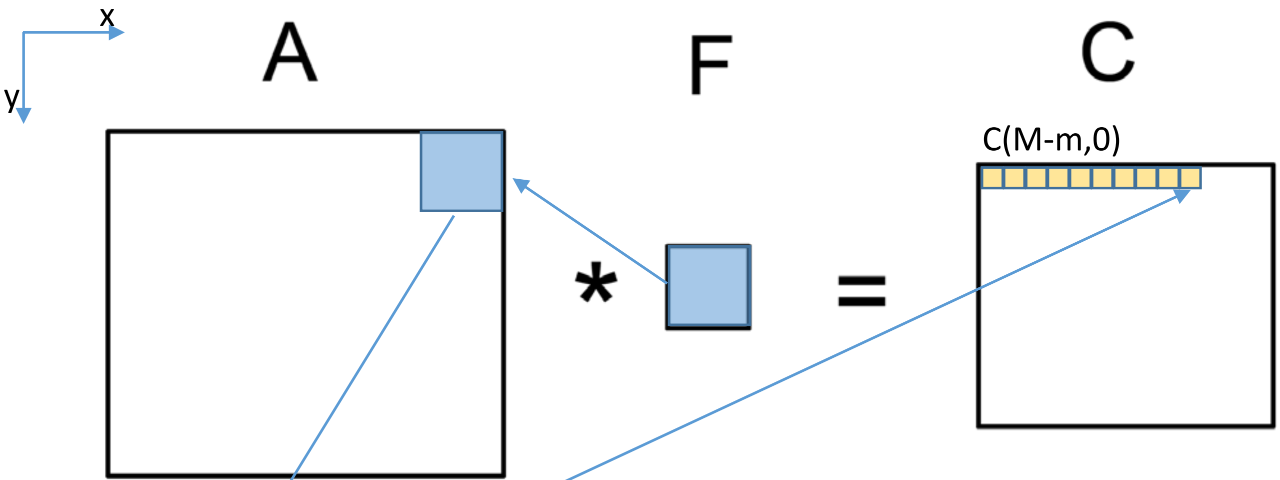
F: mxn filter (receptive field)

C: output channel

Convolution

$$C(i,j) = \sum_{x=0}^{m-1} \sum_{y=0}^{n-1} A(x-i, y-j) F(x,y)$$

Convolutional Neural Network



A: MxN input channel

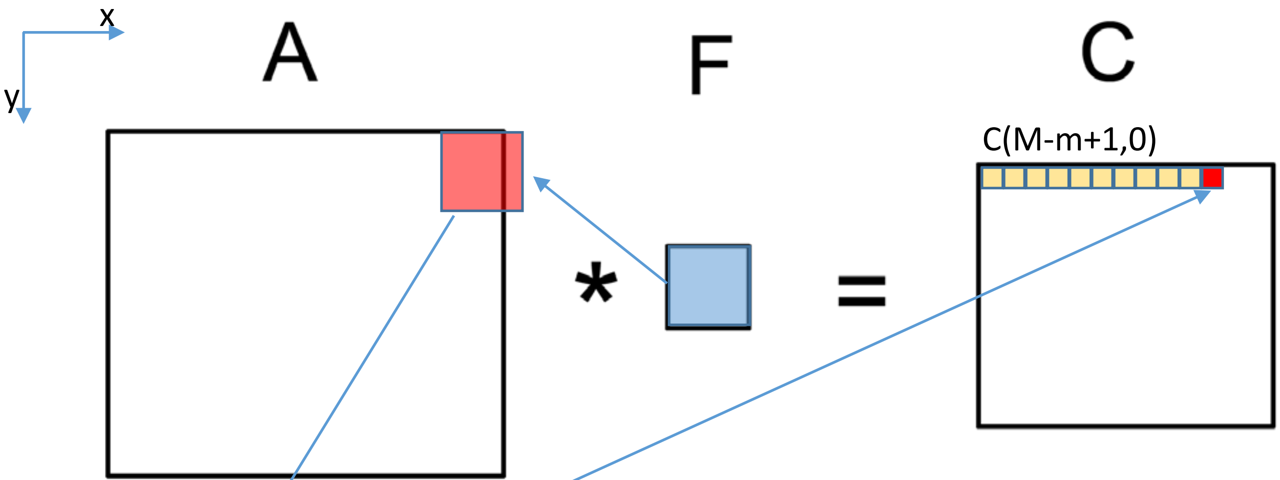
F: mxn filter (receptive field)

C: output channel

Convolution

$$C(i,j) = \sum_{x=0}^{m-1} \sum_{y=0}^{n-1} A(x-i, y-j) F(x,y)$$

Convolutional Neural Network



A: MxN input channel

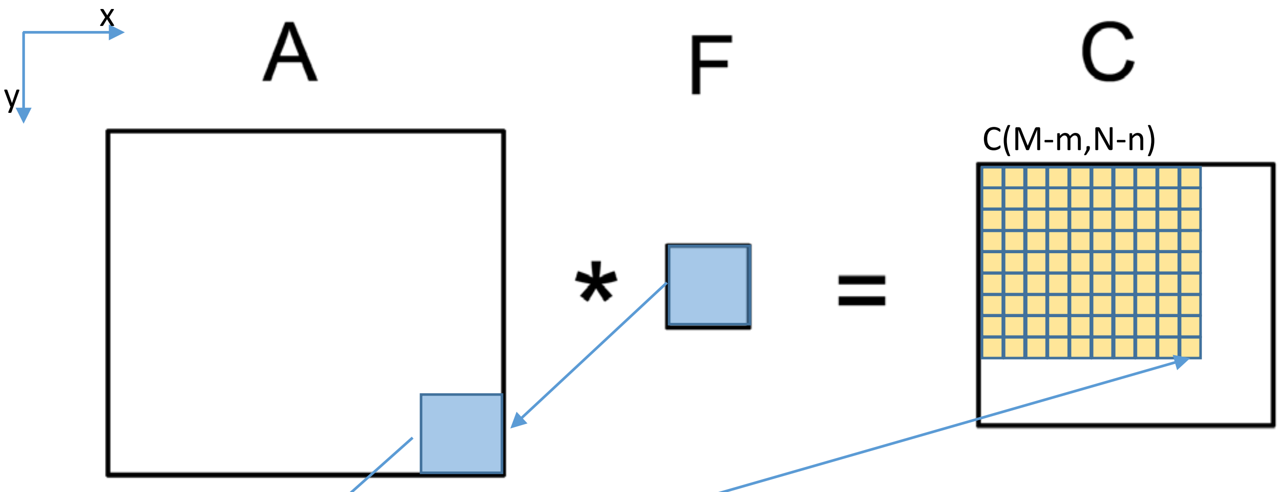
F: mxn filter (receptive field)

C: output channel

Convolution

$$C(i,j) = \sum_{x=0}^{m-1} \sum_{y=0}^{n-1} A(x-i, y-j) F(x,y)$$

Convolutional Neural Network



A: MxN input channel

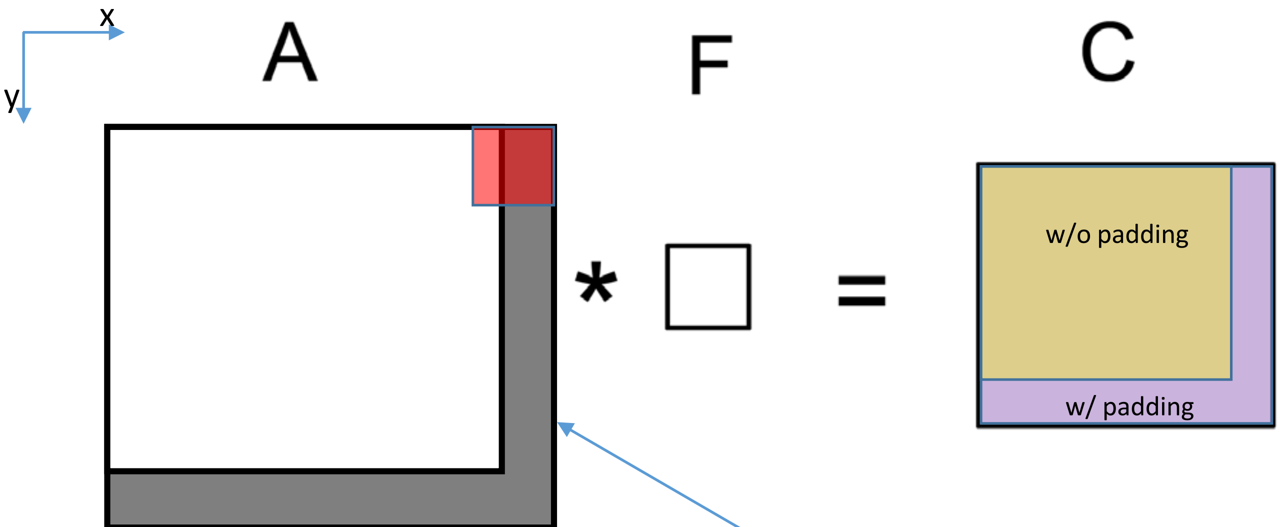
F: mxn filter (receptive field)

C: output channel

Convolution

$$C(i,j) = \sum_{x=0}^{m-1} \sum_{y=0}^{n-1} A(x-i, y-j)F(x,y)$$

Convolutional Neural Network



A: MxN input channel

F: mxn filter (receptive field)

C: output channel

- The size of the output channel of a convolution might be **smaller** than the size of the input channel.
- In order to avoid that, we can apply **zero padding** to the input channel

Convolutional Neural Network

http://deeplearning.net/software/theano/library/tensor/nnet/conv.html#theano.tensor.nnet.conv.conv2d

- 2D convolution in Theano:

`theano.tensor.nnet.conv.`**conv2d**(*input, filters, image_shape=None, filter_shape=None, border_mode='valid', subsample=(1, 1), **kargs*)

This function will build the symbolic graph for convolving a stack of input images with a set of filters. The implementation is modelled after Convolutional Neural Networks (CNN). It is simply a wrapper to the ConvOp but provides a much cleaner interface.

Parameters:
- **input** (*symbolic 4D tensor*) – Mini-batch of feature map stacks, of shape (batch size, stack size, nb row, nb col) see the optional parameter image_shape
- **filters** (*symbolic 4D tensor*) – Set of filters used in CNN layer of shape (nb filters, stack size, nb row, nb col) see the optional parameter filter_shape

padding option

- **border_mode** (*{'valid', 'full'}*) – 'valid'only apply filter to complete patches of the image. Generates output of shape: image_shape – filter_shape + 1. 'full' zero-pads image to multiple of filter shape to generate output of shape: image_shape + filter_shape – 1.
- **subsample** (*tuple of len 2*) – Factor by which to subsample the output. Also called strides elsewhere.
- **image_shape** (*None, tuple/list of len 4 of int, None or Constant variable*) – The shape of the input parameter. Optional, used for optimization like loop unrolling You can put None for any element of the list to tell that this element is not constant.
- **filter_shape** (*None, tuple/list of len 4 of int, None or Constant variable*) – Optional, used for optimization like loop unrolling You can put None for any element of the list to tell that this element is not constant.
- **kwargs** –
  Kwargs are passed onto ConvOp. Can be used to set the following: unroll_batch, unroll_kern, unroll_patch, openmp (see ConvOp doc).

  openmp: By default have the same value as

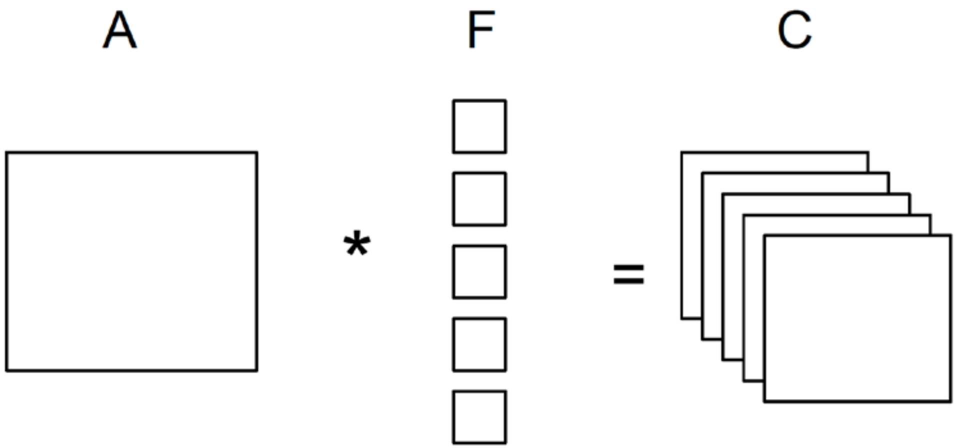
    config.openmp. For small image, filter, batch size, nkern and stack size, it can be faster to disable manually openmp. A fast and incomplete test show that with image size 6x6, filter size 4x4, batch size==1, n kern==1 and stack size==1, it is faster to disable it in valid mode. But if we grow the batch size to 10, it is faster with openmp on a core 2 duo.

Returns: Set of feature maps generated by convolutional layer. Tensor is of shape (batch size, nb filters, output row, output col).

Return type: symbolic 4D tensor

Convolutional Neural Network

- Applying **multiple filters** we obtain **multiple output** channels



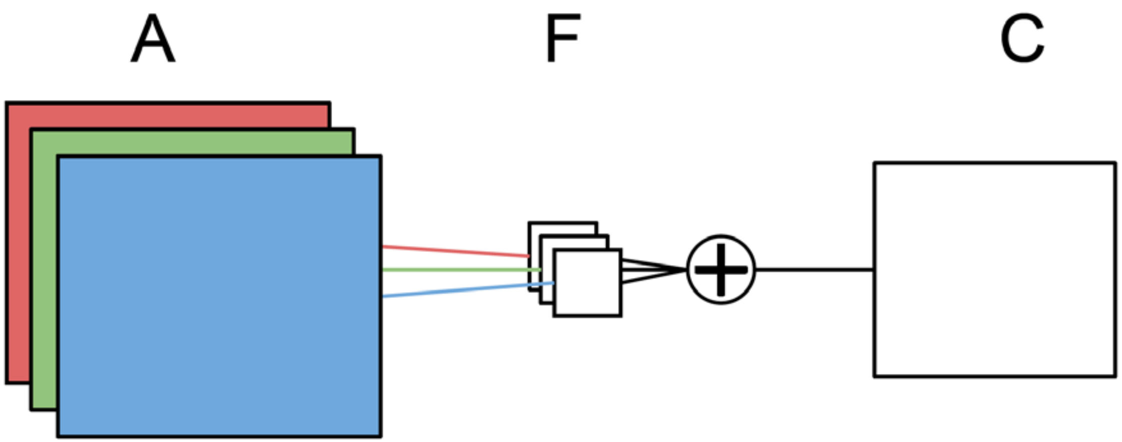
$$C_o(i,j) = \sum_{x=0}^{m-1} \sum_{y=0}^{n-1} A(x-i, y-j) F_o(x,y)$$

o is the index of the output channel

[slide adapted from Arnaud Bergeron]

Convolutional Neural Network

- Convolution with **multiple input channels** (example: RGB images)



One example of **multiple channels** is also the **output** of the previous slide

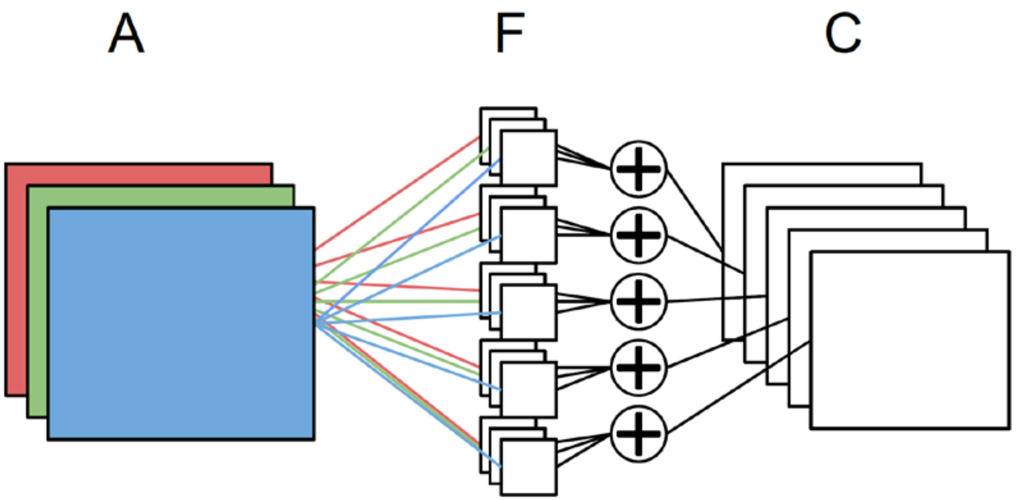
$$C(i,j) = \sum_{k=0}^{l} \sum_{x=0}^{m-1} \sum_{y=0}^{n-1} A_k(x-i, y-j) F_k(x,y)$$

F is now a set of filters
k is the index of the input channel

[slide adapted from Arnaud Bergeron]

Convolutional Neural Network

- Convolution with **multiple inputs** and **outputs**



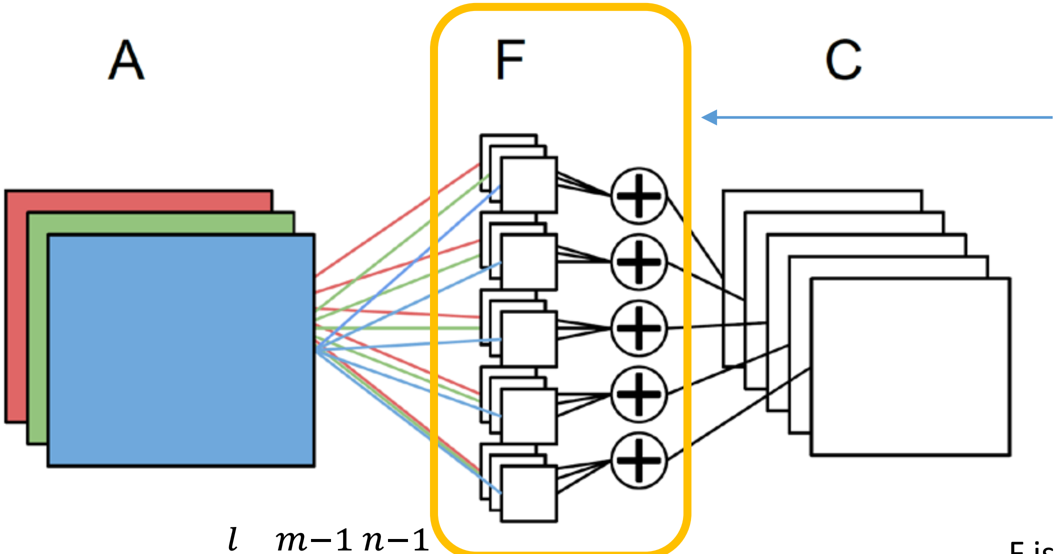
$$C_o(i,j) = \sum_{k=0}^{l} \sum_{x=0}^{m-1} \sum_{y=0}^{n-1} A_k(x-i, y-j) F_{ko}(x,y)$$

F is now a set of filters
k is the index of the input channel
o is the index of the output channel

[slide adapted from Arnaud Bergeron]

Convolutional Neural Network

- Convolution with **multiple inputs** and **outputs**



We can call this a **Convolutional layer**

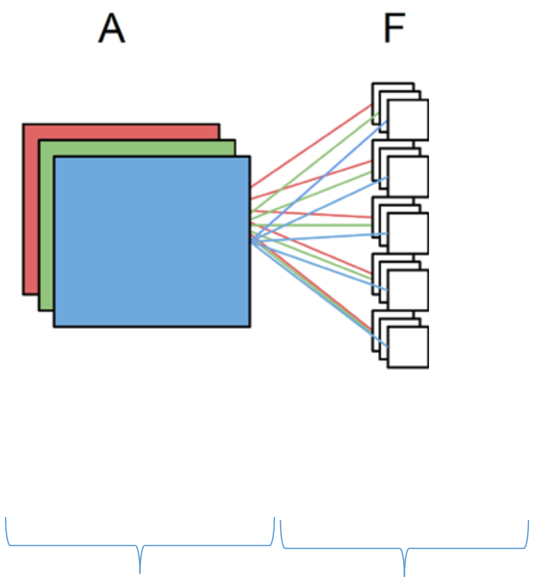
$$C_o(i,j) = \sum_{k=0}^{l} \sum_{x=0}^{m-1} \sum_{y=0}^{n-1} A_k(x-i, y-j) F_{ko}(x,y)$$

F is now a set of filters
k is the index of the input channel
o is the index of the output channel

[slide adapted from Arnaud Bergeron]

Convolutional Neural Network

- A convolutional network is (basically) a stack of convolutional layers

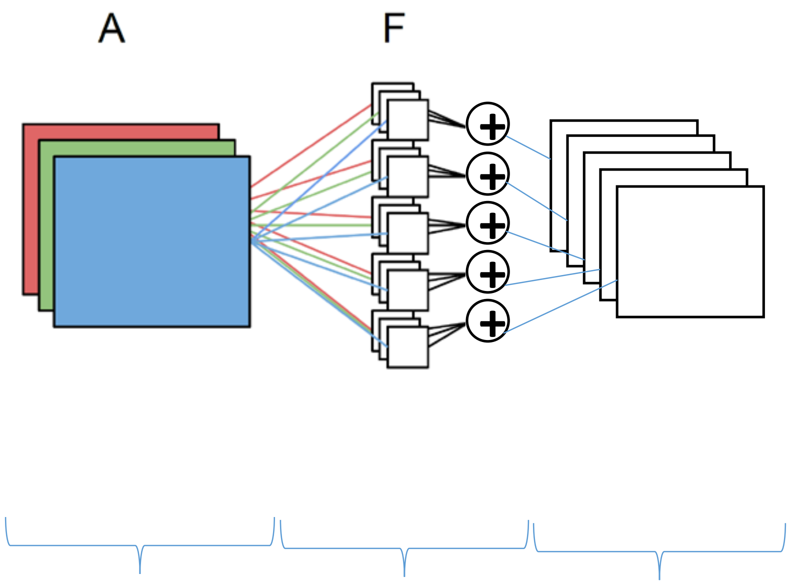


3 channel input

5*(3 x m x n) filters

Convolutional Neural Network

- A convolutional network is (basically) a stack of convolutional layers



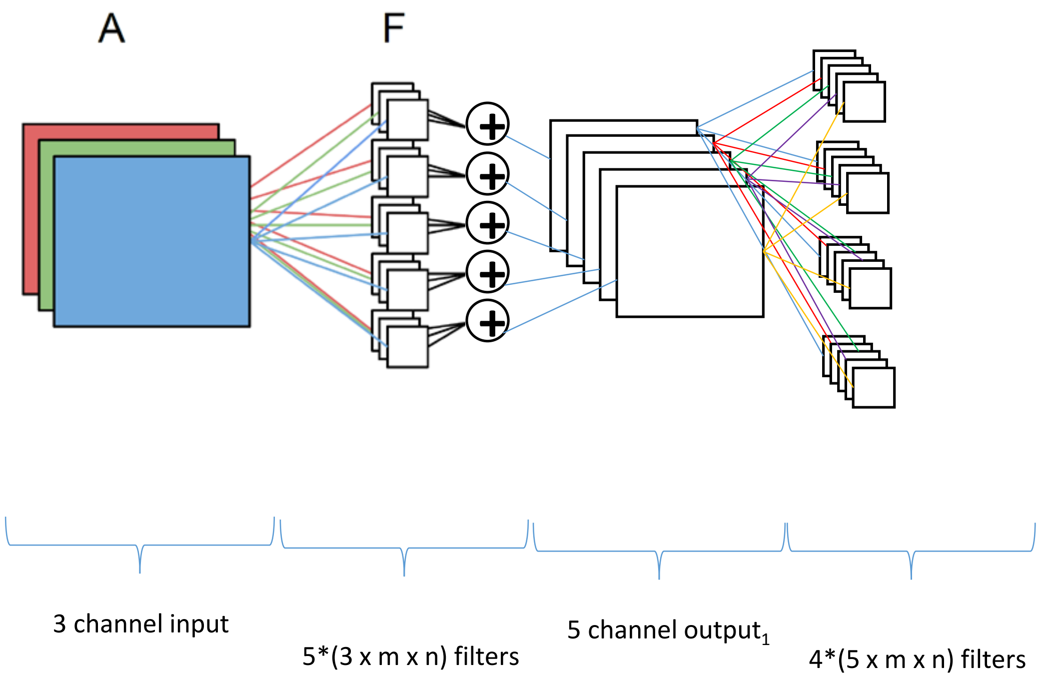
3 channel input

5*(3 x m x n) filters

5 channel output$_1$

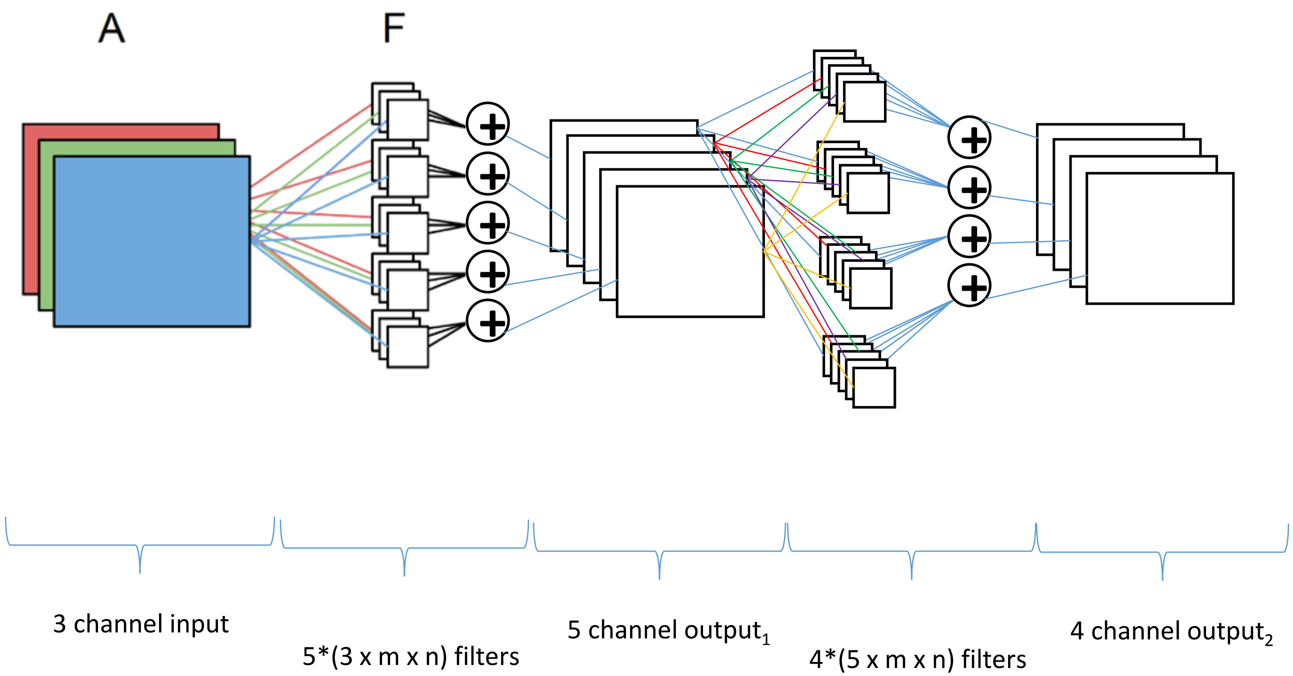
Convolutional Neural Network

- A convolutional network is (basically) a stack of convolutional layers



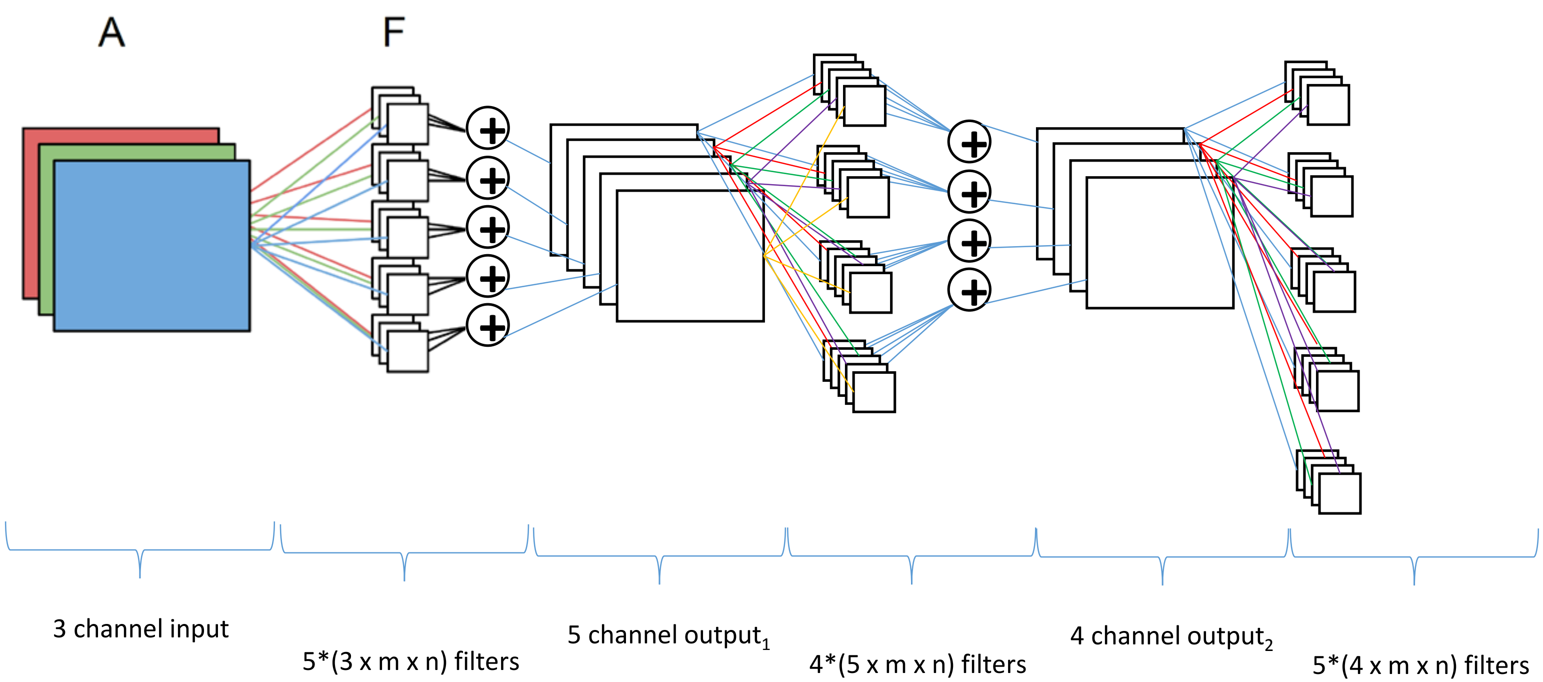
Convolutional Neural Network

- A convolutional network is (basically) a stack of convolutional layers



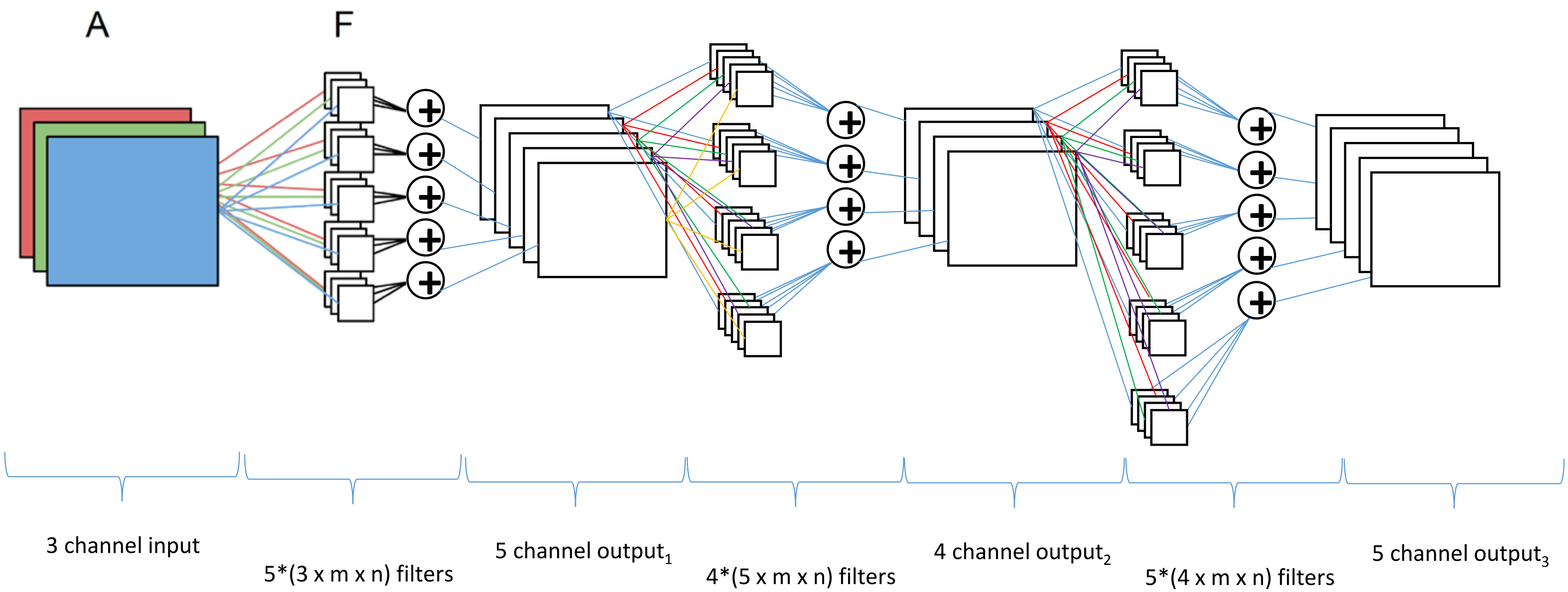
Convolutional Neural Network

- A convolutional network is (basically) a stack of convolutional layers



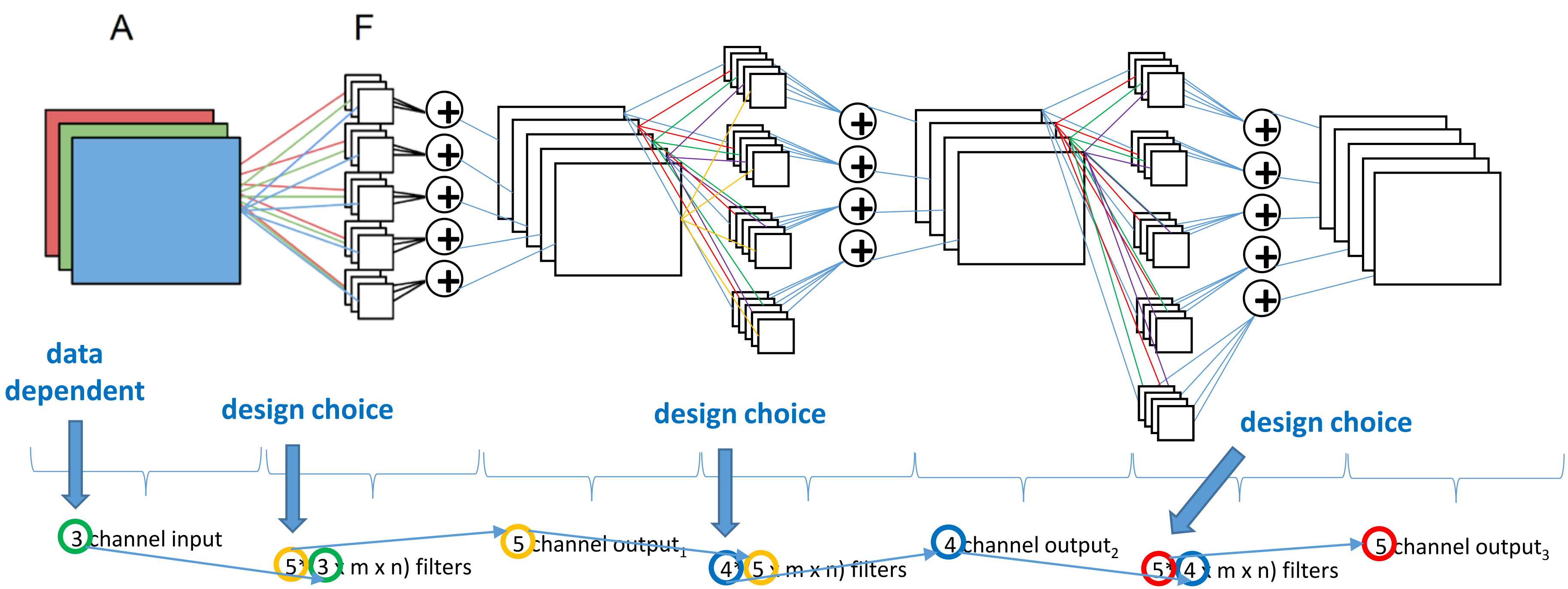
Convolutional Neural Network

- A convolutional network is (basically) a stack of convolutional layers

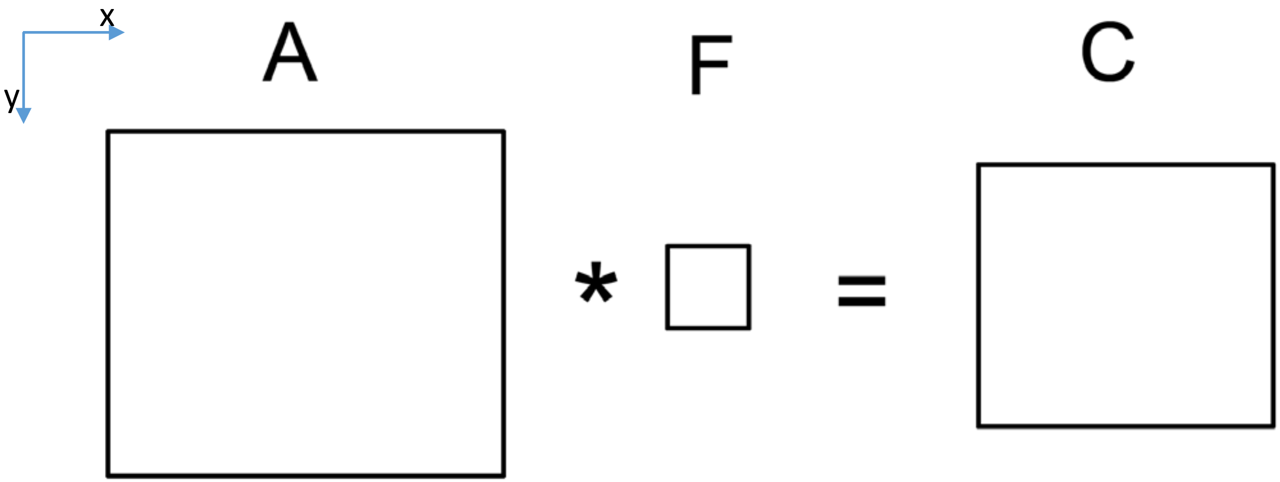


Convolutional Neural Network

- A convolutional network is (basically) a stack of convolutional layers



Convolutional Neural Network



A: MxN input channel

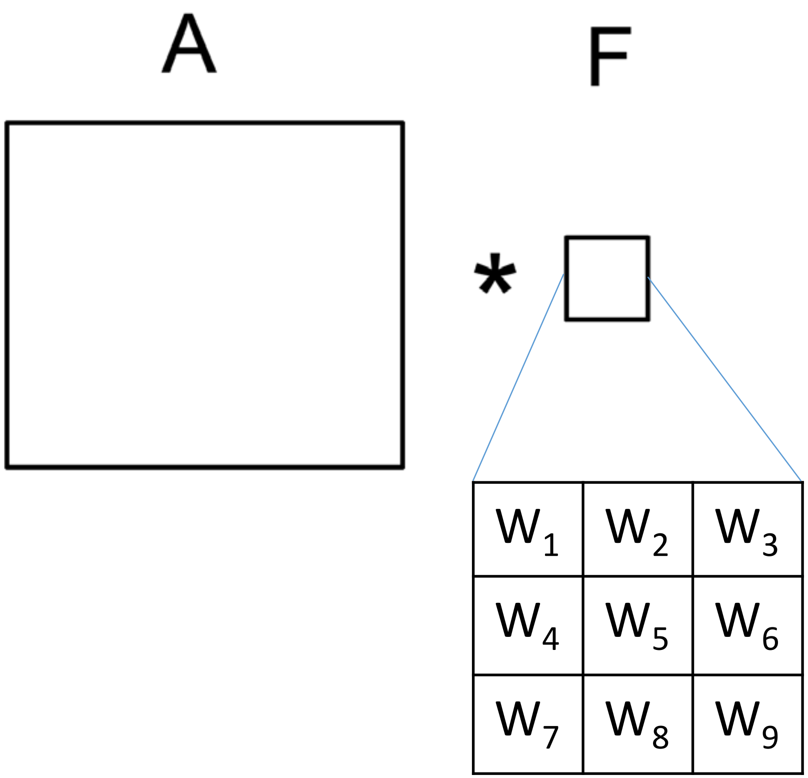
F: mxn filter (receptive field)

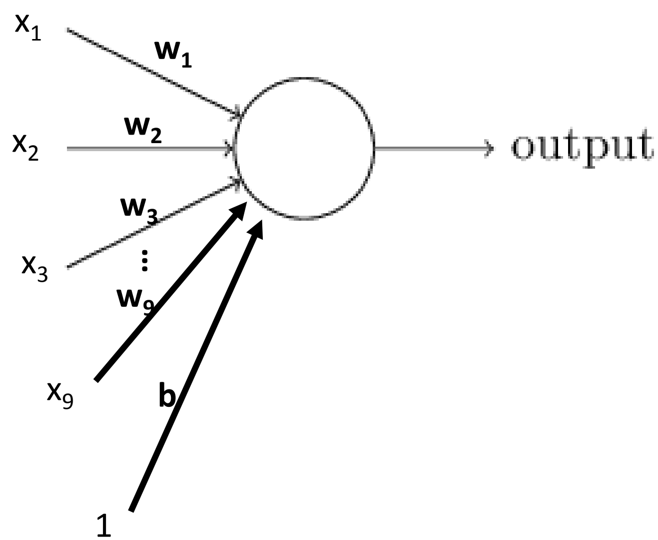
C: output channel

Convolution

$$C(i,j) = \sum_{x=0}^{m-1} \sum_{y=0}^{n-1} A(x-i, y-j) F(x,y)$$

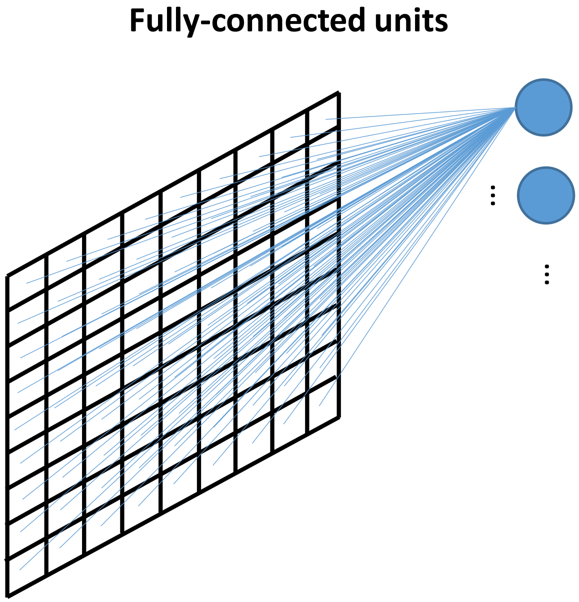
Convolutional Neural Network





Training a convolutional network means learn these parameters!

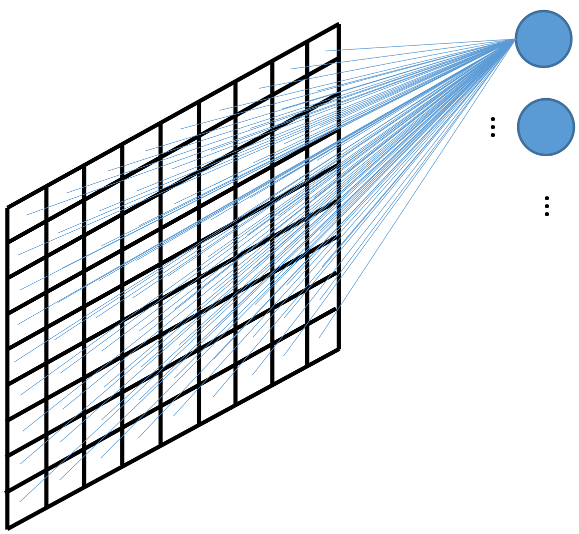
Why not a Multi Layer Perceptron?



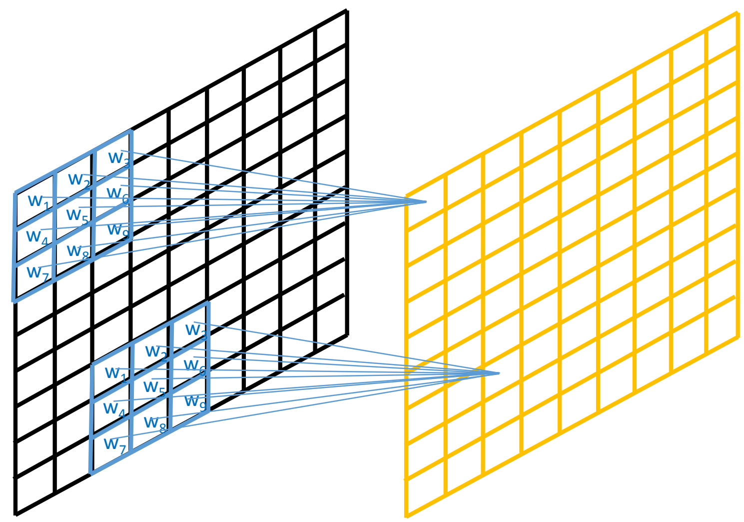
- **Huge first layer**
  - Lots of weights
  - Increase the capacity
  - Lots of training data required
  - Huge memory requirement
- **No robustness to distortions or shift of the input**
- **In MLP, input variables can be presented in any (fixed) order**
  - This does not take into account for topology in data
  - Images have a strong 2D structure

Why not a Multi Layer Perceptron?

Fully-connected units

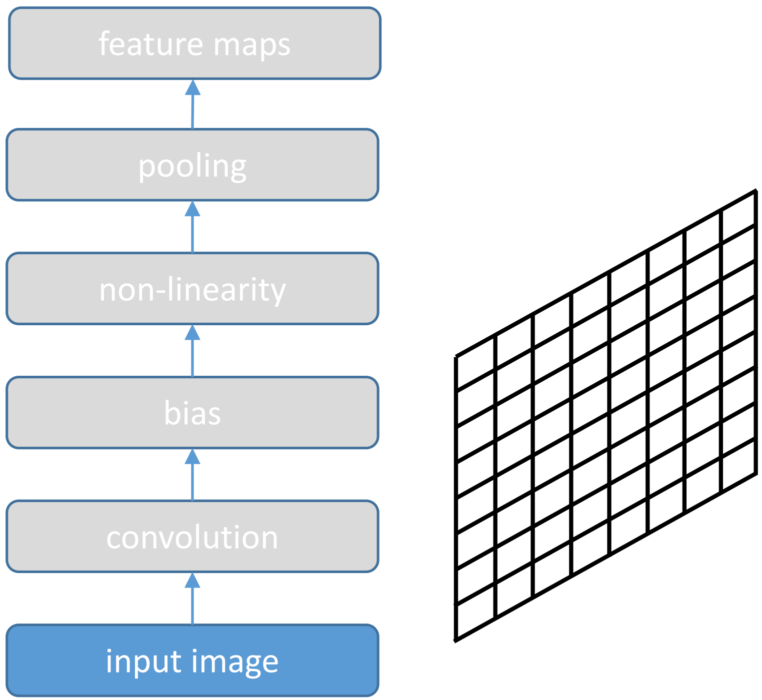


Shared weights

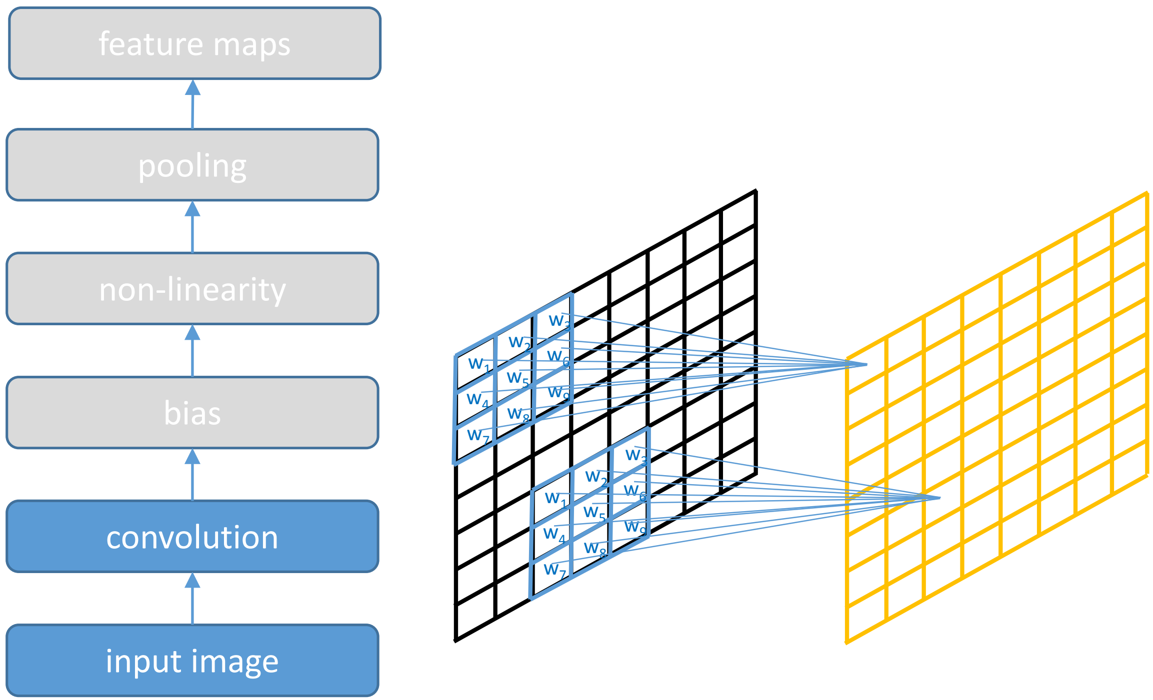


Convolutional Neural Network

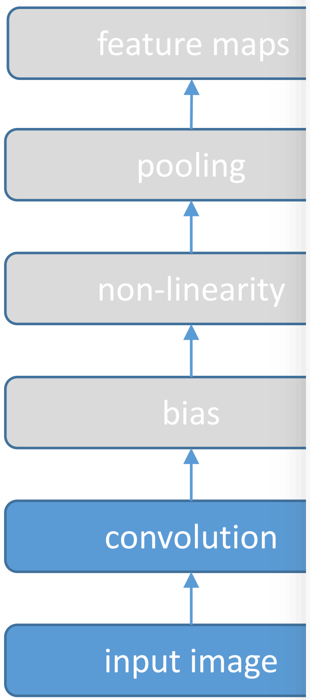
Code snippets from: http://deeplearning.net/tutorial/lenet.html



Convolutional Neural Network

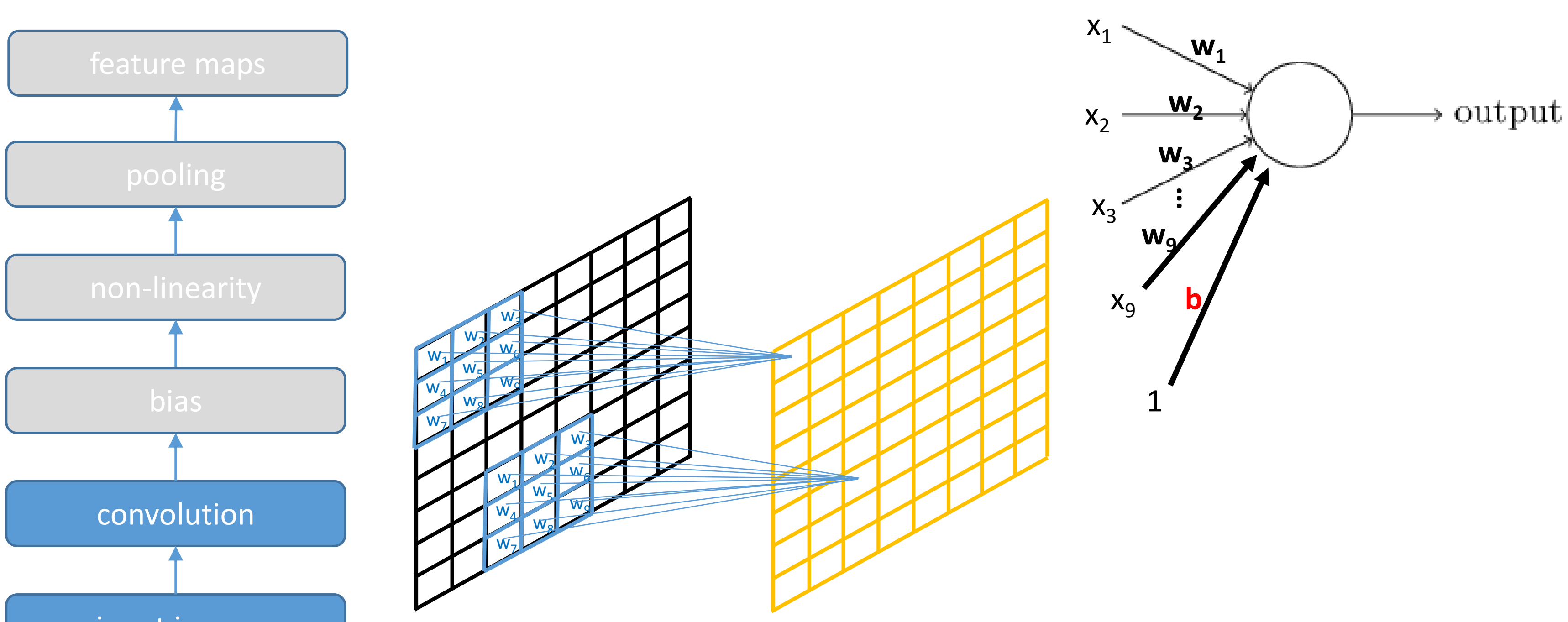


Convolutional Neural Network

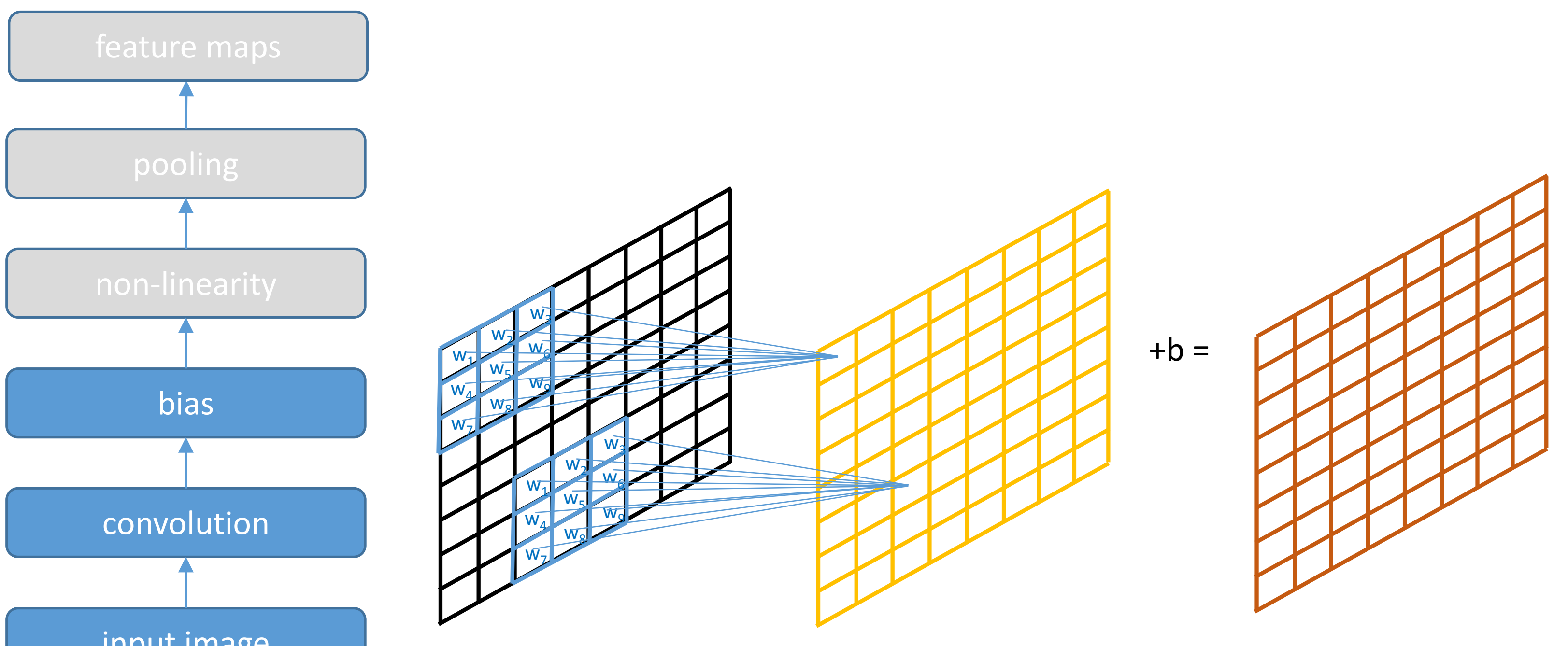



```python
import theano
from theano import tensor as T
from theano.tensor.nnet import conv

import numpy

rng = numpy.random.RandomState(23455)

# instantiate 4D tensor for input
input = T.tensor4(name='input')

# initialize shared variable for weights.
w_shp = (2, 3, 9, 9)
w_bound = numpy.sqrt(3 * 9 * 9)
W = theano.shared( numpy.asarray(
            rng.uniform(
                low=-1.0 / w_bound,
                high=1.0 / w_bound,
                size=w_shp),
            dtype=input.dtype), name ='W')

# initialize shared variable for bias (1D tensor) with random values
# IMPORTANT: biases are usually initialized to zero. However in this
# particular application, we simply apply the convolutional layer to
# an image without learning the parameters. We therefore initialize
# them to random values to "simulate" learning.
b_shp = (2,)
b = theano.shared(numpy.asarray(
            rng.uniform(low=-.5, high=.5, size=b_shp),
            dtype=input.dtype), name ='b')

# build symbolic expression that computes the convolution of input with filters in w
conv_out = conv.conv2d(input, W)
```

Convolutional Neural Network

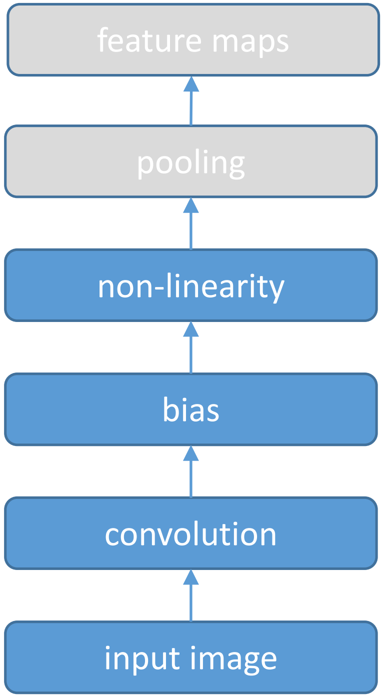


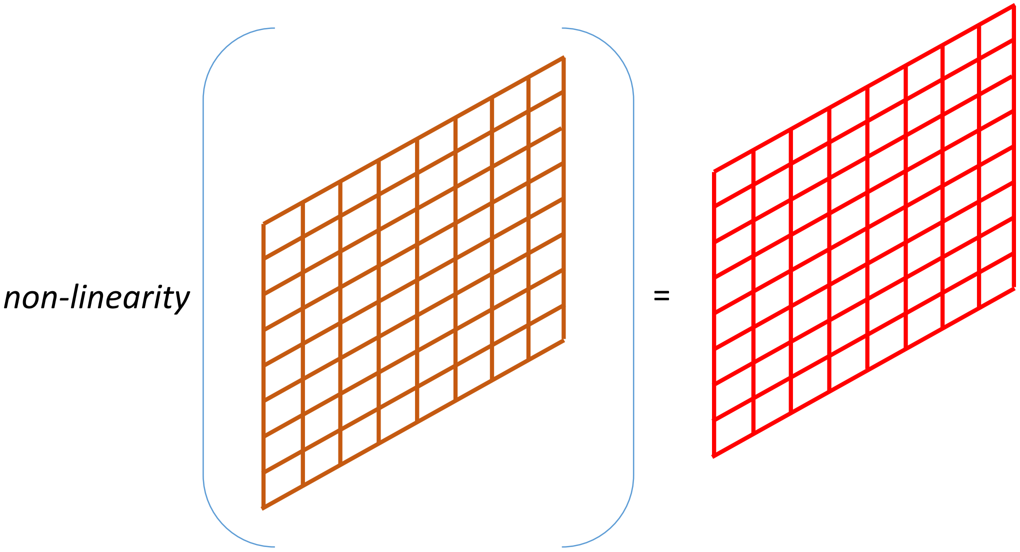
Convolutional Neural Network



We add the trained value b to each pixel.

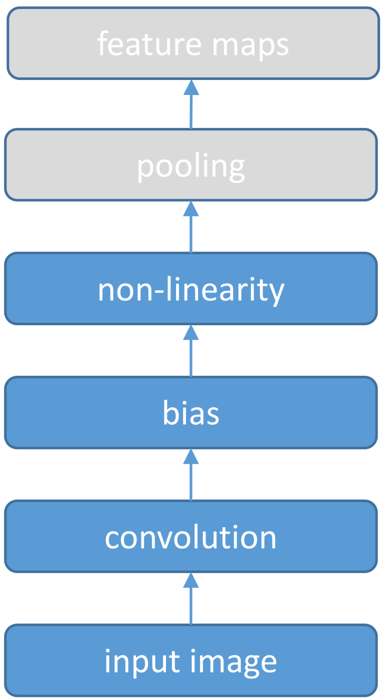
Convolutional Neural Network





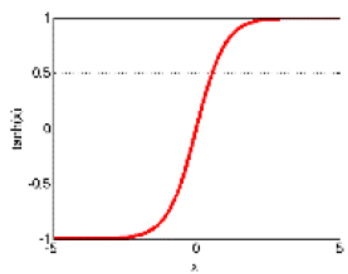
We apply a non-linear function to each pixel. The result is also called **activation**

Convolutional Neural Network

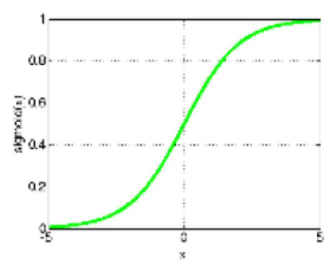


Non-linearity function: continue and differentiable (almost) everywhere

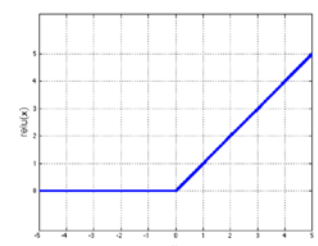
Tanh



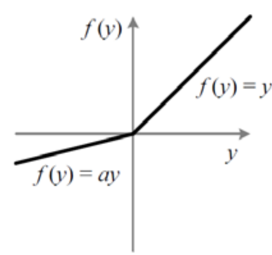
Sigmoid



Rectified linear (ReLU)



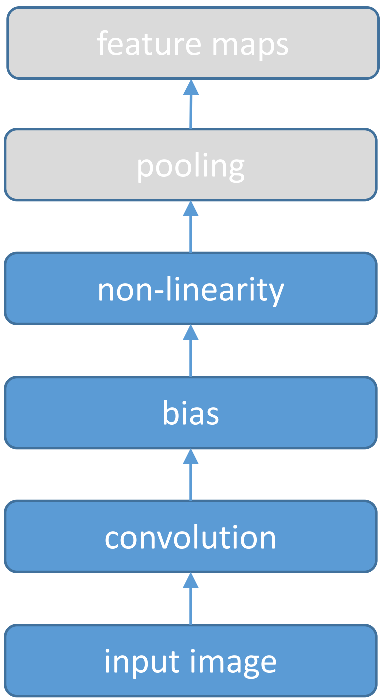
Parametric Rectified linear (PReLU)
[ArXiv:1502.01852]



ReLU

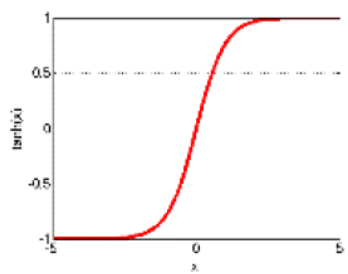
- makes learning faster
- sparse representation
- avoid saturation
- avoid vanishing gradient

Convolutional Neural Network

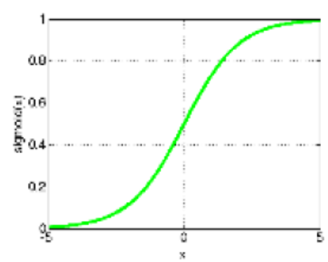


Non-linearity function: continue and differentiable (almost) everywhere

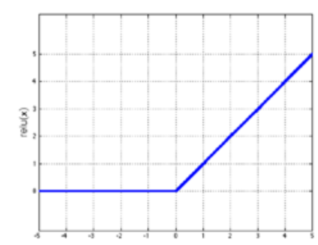
Tanh



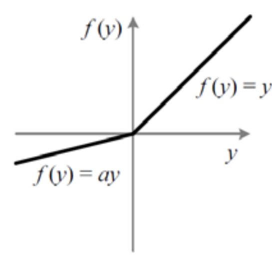
Sigmoid



Rectified linear (ReLU)



Parametric Rectified linear (PReLU)
[ArXiv:1502.01852]

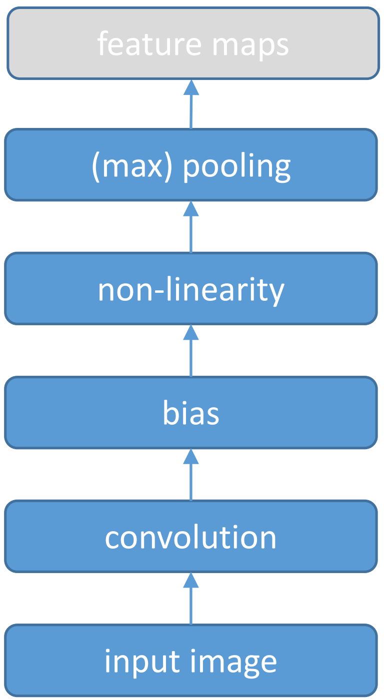


ReLU

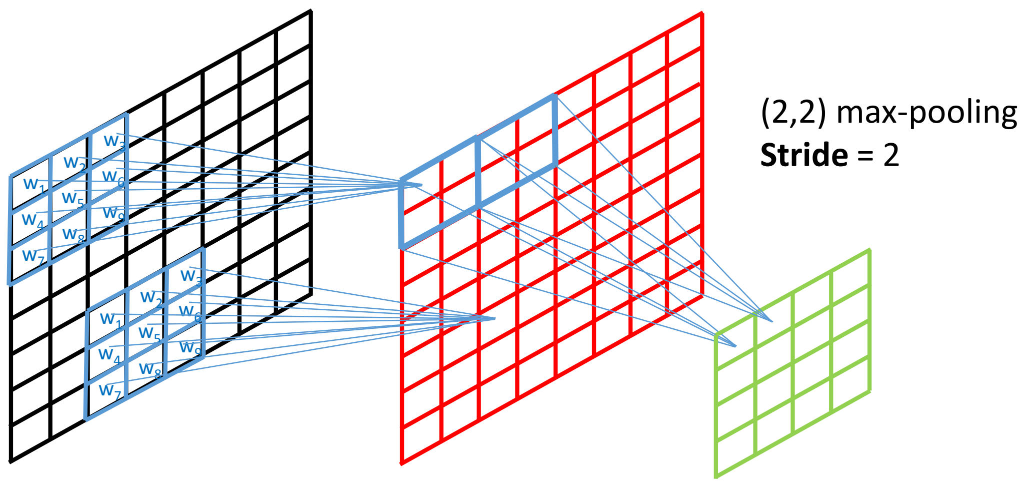
- makes learning faster
- sparse representation
- avoid saturation
- avoid vanishing gradient

```
output = T.nnet.sigmoid(conv_out + b.dimshuffle('x', 0, 'x', 'x'))
```

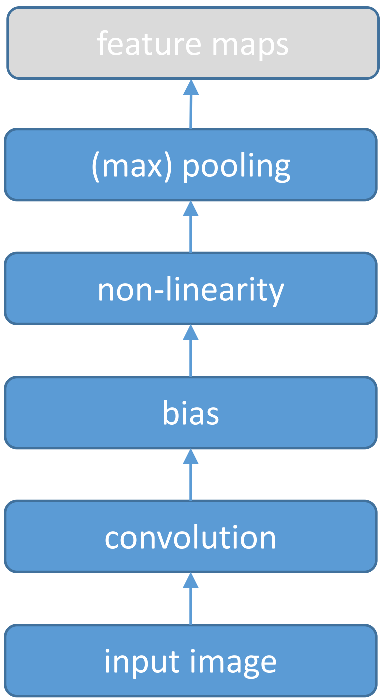
Convolutional Neural Network



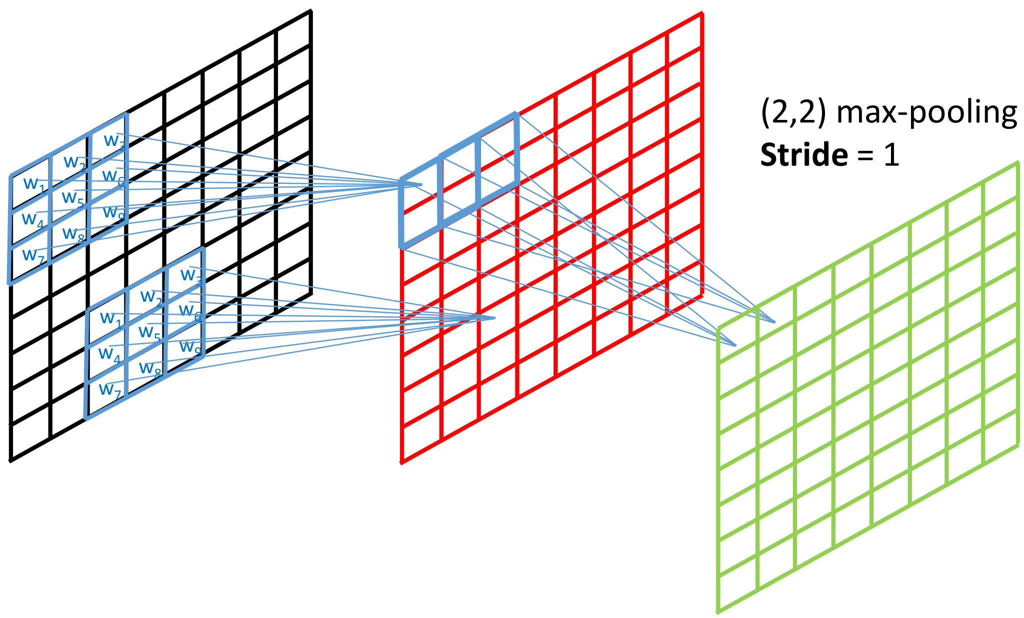
- **Pooling**
  - Encodes a degree of invariance with respect to translations
  - Reduces the size of the layers



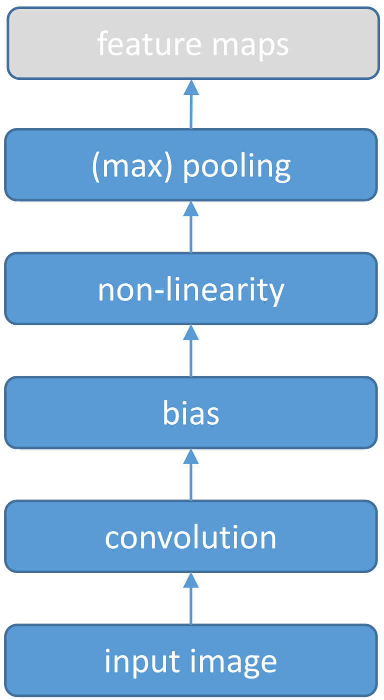
Convolutional Neural Network



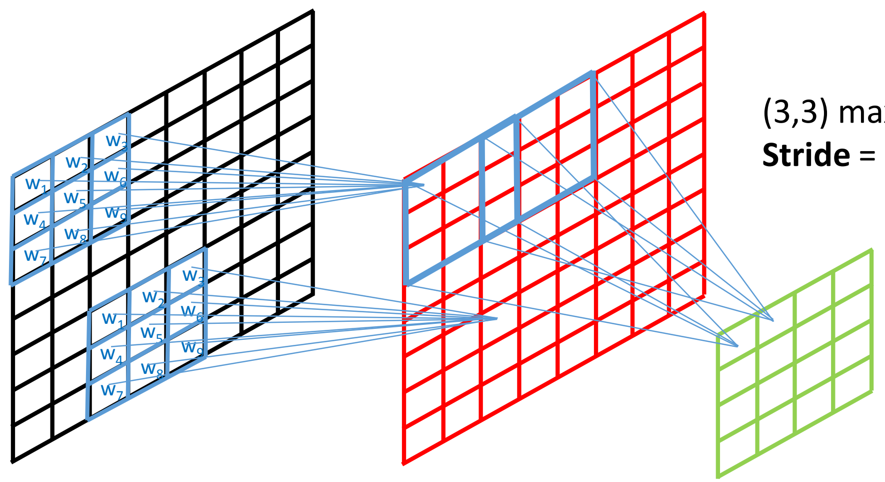
- **Pooling**
  - Encodes a degree of invariance with respect to translations
  - Reduces the size of the layers



Convolutional Neural Network



- **Pooling**
  - Encodes a degree of invariance with respect to translations
  - Reduces the size of the layers



(3,3) max-pooling
Stride = 2

Convolutional Neural Network

downsample – Down-Sampling

theano.tensor.signal.downsample.**max_pool_2d**(*input, ds, ignore_border=None, st=None, padding=(0, 0), mode='max'*)

Takes as input a N-D tensor, where N >= 2. It downscales the input image by the specified factor, by keeping only the maximum value of non-overlapping patches of size (ds[0],ds[1])

Parameters:
- **input** (*N-D theano tensor of input images*) – Input images. Max pooling will be done over the 2 last dimensions.
- **ds** (*tuple of length 2*) – Factor by which to downscale (vertical ds, horizontal ds). (2,2) will halve the image in each dimension.
- **ignore_border** (*bool (default None, will print a warning and set to False)*) – When True, (5,5) input with ds=(2,2) will generate a (2,2) output. (3,3) otherwise.
- **st** (*tuple of lenght 2*) – Stride size, which is the number of shifts over rows/cols to get the next pool region. If st is None, it is considered equal to ds (no overlap on pooling regions).
- **padding** (*tuple of two ints*) – (pad_h, pad_w), pad zeros to extend beyond four borders of the images, pad_h is the size of the top and bottom margins, and pad_w is the size of the left and right margins.
- **mode** (*{'max', 'sum', 'average_inc_pad', 'average_exc_pad'}*) – Operation executed on each window. `max` and `sum` always exclude the padding in the computation. `average` gives you the choice to include or exclude it.

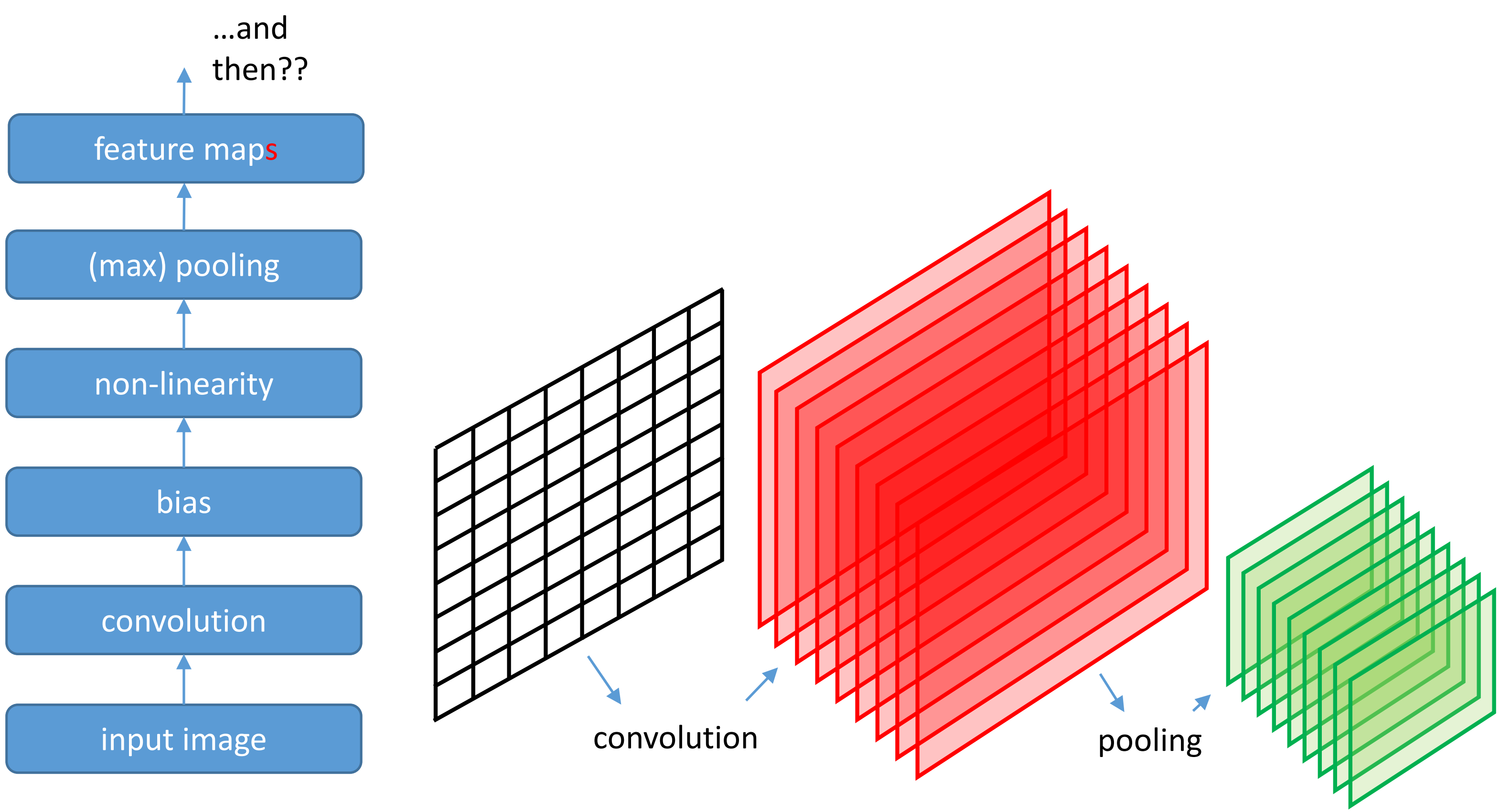
theano.tensor.signal.downsample.**max_pool_2d_same_size**(*input, patch_size*)

Takes as input a 4-D tensor. It sets all non maximum values of non-overlapping patches of size (patch_size[0],patch_size[1]) to zero, keeping only the maximum values. The output has the same dimensions as the input.

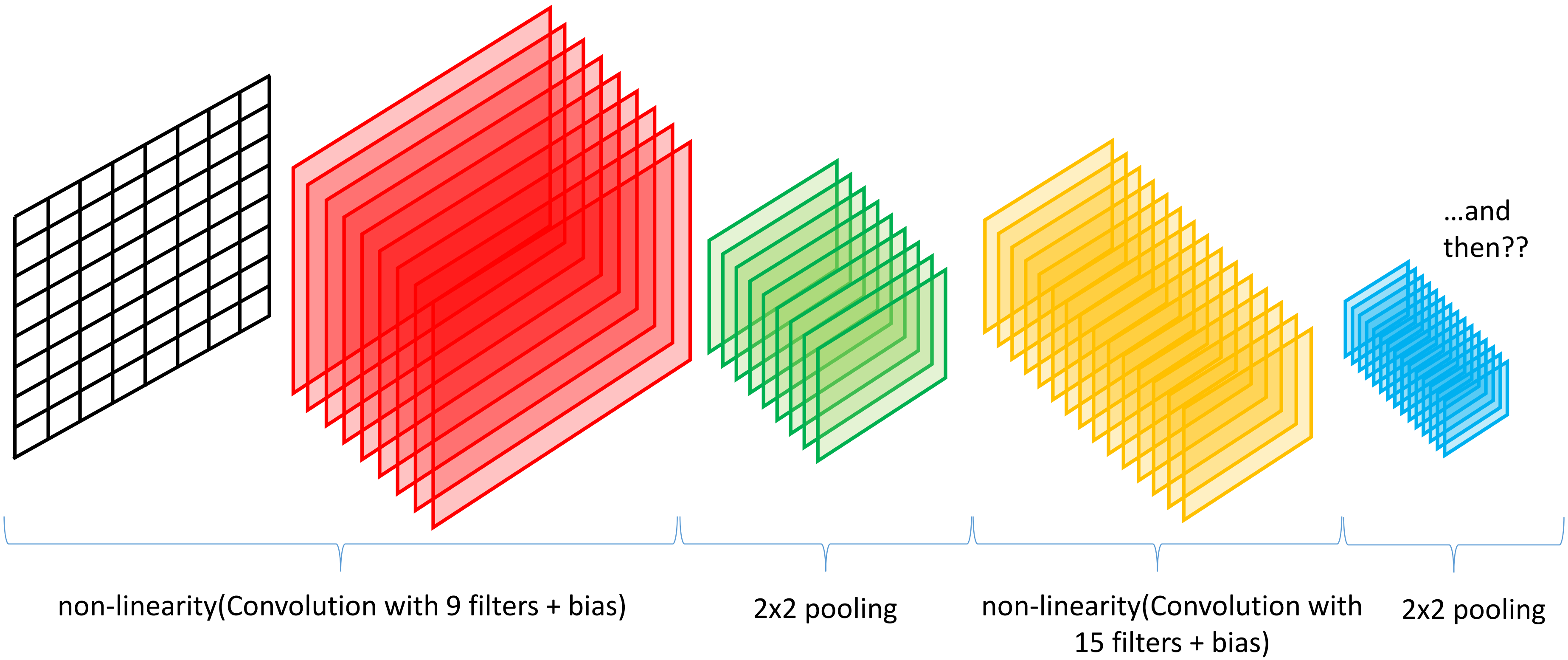
Parameters:
- **input** (*4-D theano tensor of input images*) – Input images. Max pooling will be done over the 2 last dimensions.
- **patch_size** (*tuple of length 2*) – Size of the patch (patch height, patch width). (2,2) will retain only one non-zero value per patch of 4 values.

```
from theano.tensor.signal import downsample

input = T.dtensor4('input')
maxpool_shape = (2, 2)
pool_out = downsample.max_pool_2d(input, maxpool_shape, ignore_border=True)
f = theano.function([input],pool_out)
```

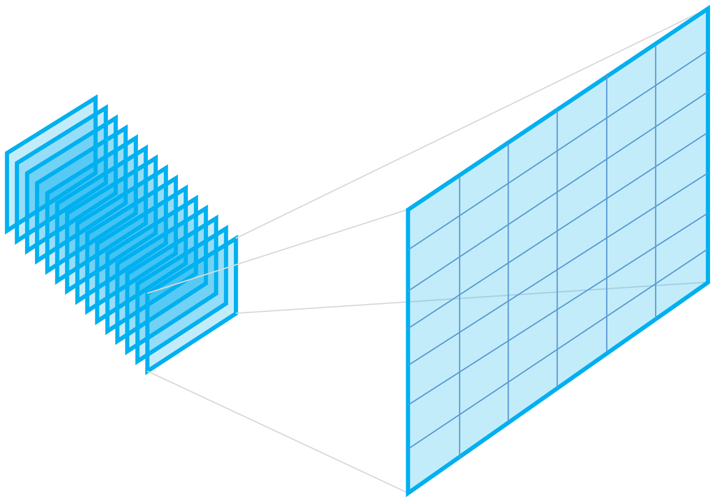
Convolutional Neural Network



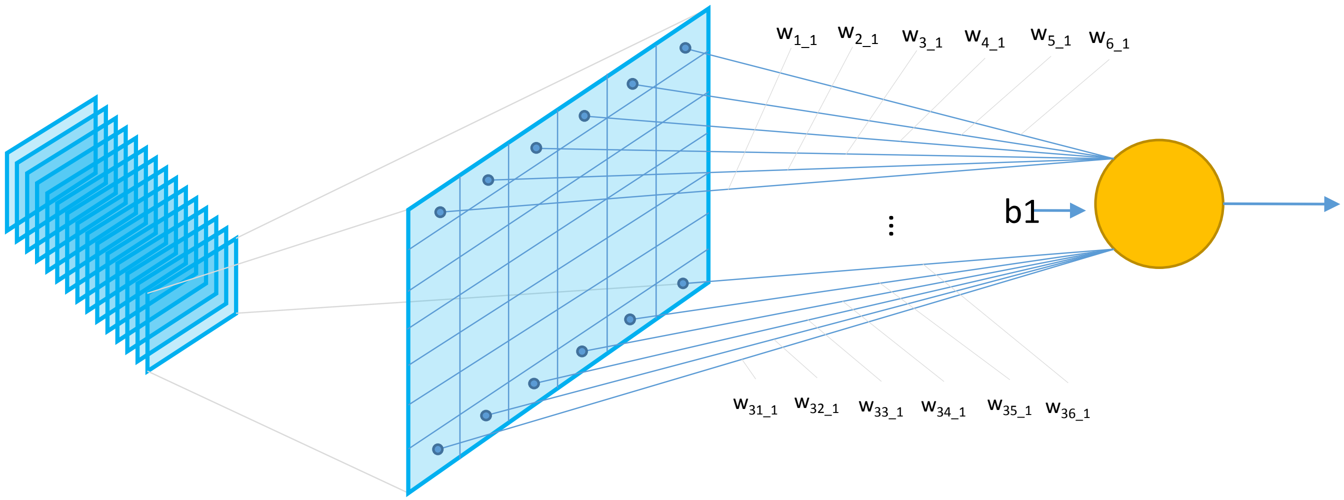
Convolutional Neural Network



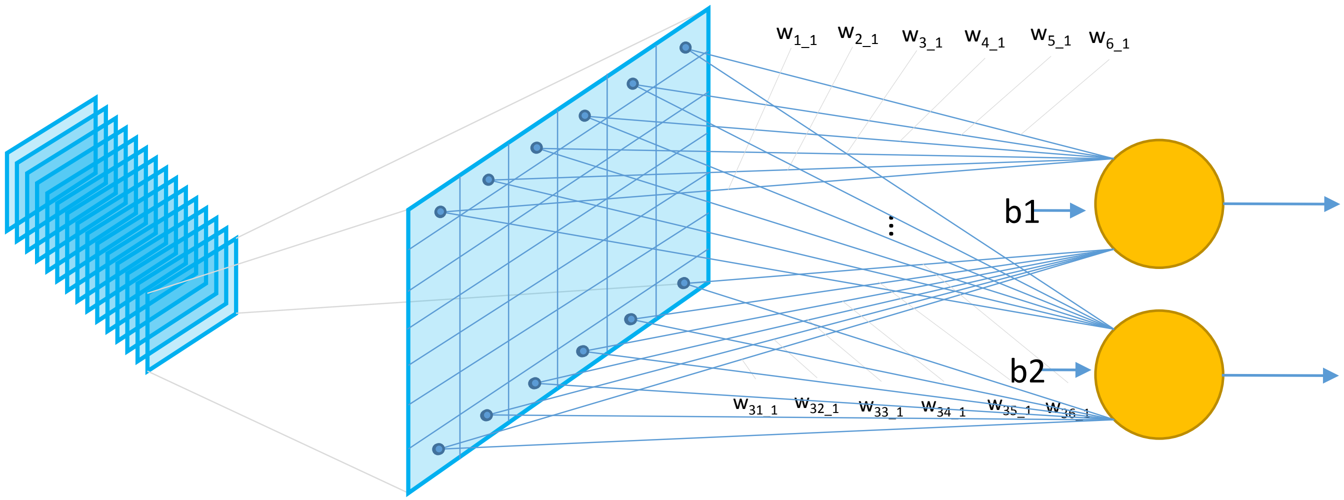
Fully-connected layer



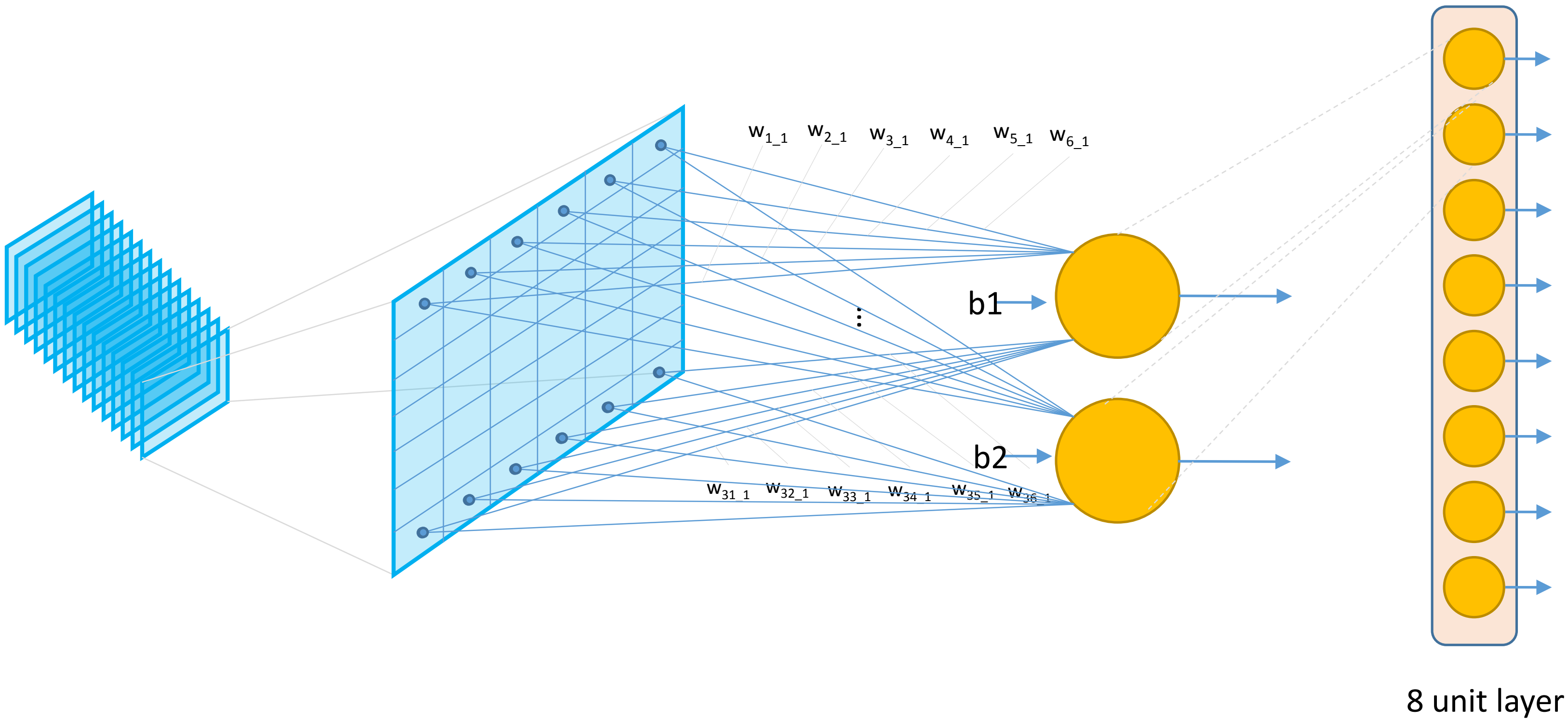
Fully-connected layer



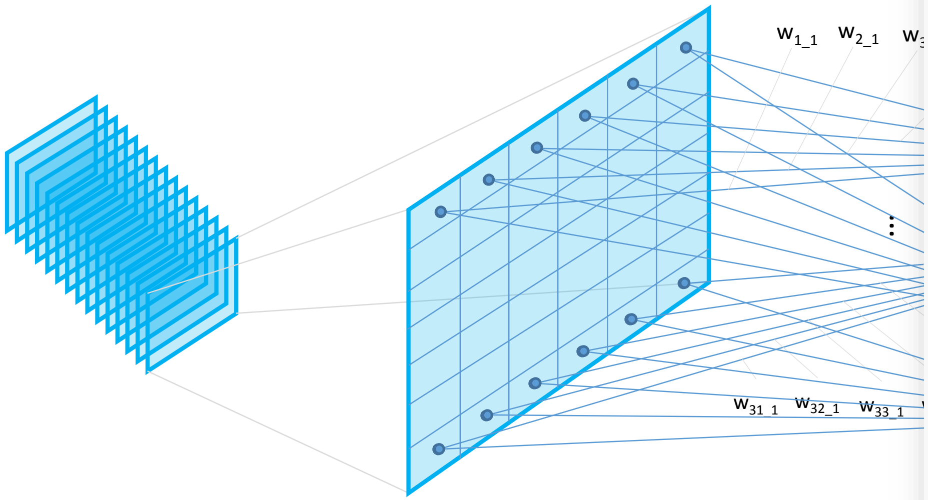
Fully-connected layer



Fully-connected layer

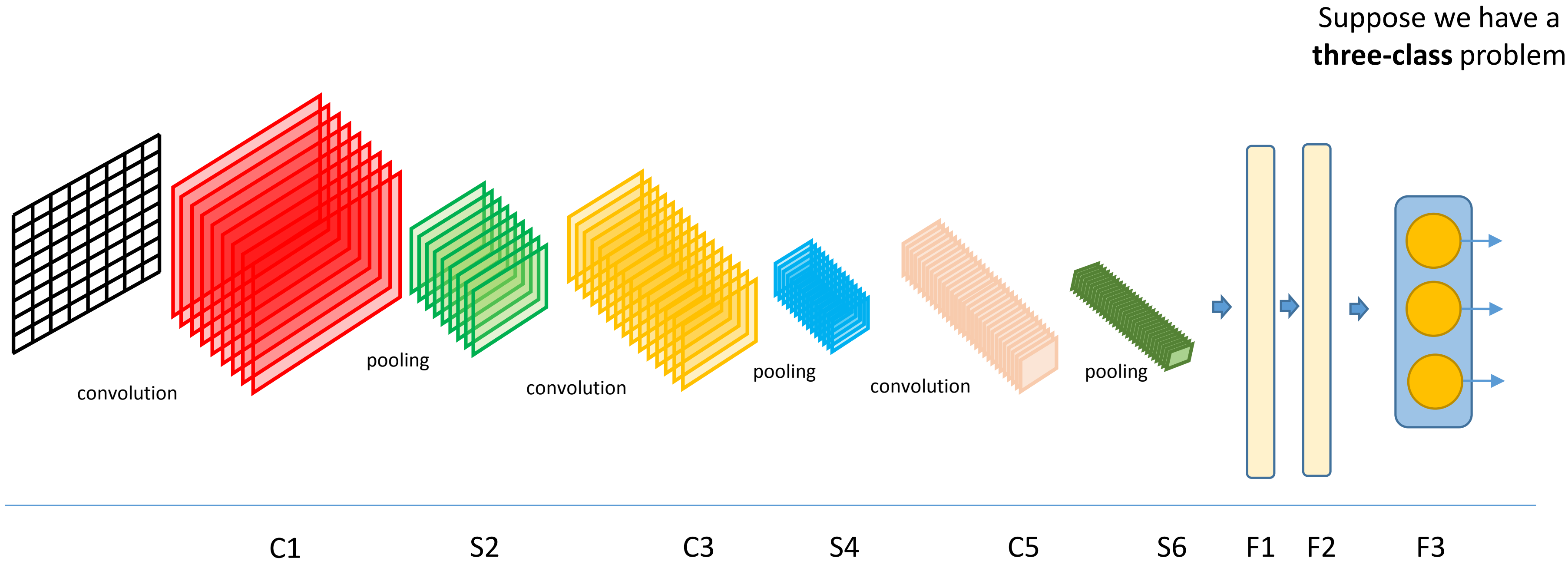


Fully-connected lay




```python
class HiddenLayer(object):
    def __init__(self, rng, input, n_in, n_out, W=None, b=None,
                 activation=T.tanh):
        """
        Typical hidden layer of a MLP: units are fully-connected and have
        sigmoidal activation function. Weight matrix W is of shape (n_in,n_out)
        and the bias vector b is of shape (n_out,).

        NOTE : The nonlinearity used here is tanh

        Hidden unit activation is given by: tanh(dot(input,W) + b)

        :type rng: numpy.random.RandomState
        :param rng: a random number generator used to initialize weights

        :type input: theano.tensor.dmatrix
        :param input: a symbolic tensor of shape (n_examples, n_in)

        :type n_in: int
        :param n_in: dimensionality of input

        :type n_out: int
        :param n_out: number of hidden units

        :type activation: theano.Op or function
        :param activation: Non linearity to be applied in the hidden
                           layer
        """
        self.input = input
        # end-snippet-1

        # `W` is initialized with `W_values` which is uniformely sampled
        # from sqrt(-6./(n_in+n_hidden)) and sqrt(6./(n_in+n_hidden))
        # for tanh activation function
        # the output of uniform if converted using asarray to dtype
        # theano.config.floatX so that the code is runable on GPU
        # Note : optimal initialization of weights is dependent on the
        #        activation function used (among other things).
        #        For example, results presented in [Xavier10] suggest that you
        #        should use 4 times larger initial weights for sigmoid
        #        compared to tanh
        #        We have no info for other function, so we use the same as
        #        tanh.
        if W is None:
            W_values = numpy.asarray(
                rng.uniform(
                    low=-numpy.sqrt(6. / (n_in + n_out)),
                    high=numpy.sqrt(6. / (n_in + n_out)),
                    size=(n_in, n_out)
                ),
                dtype=theano.config.floatX
            )
            if activation == theano.tensor.nnet.sigmoid:
                W_values *= 4

            W = theano.shared(value=W_values, name='W', borrow=True)

        if b is None:
            b_values = numpy.zeros((n_out,), dtype=theano.config.floatX)
            b = theano.shared(value=b_values, name='b', borrow=True)

        self.W = W
        self.b = b

        lin_output = T.dot(input, self.W) + self.b
        self.output = (
            lin_output if activation is None
            else activation(lin_output)
        )
        # parameters of the model
        self.params = [self.W, self.b]
```

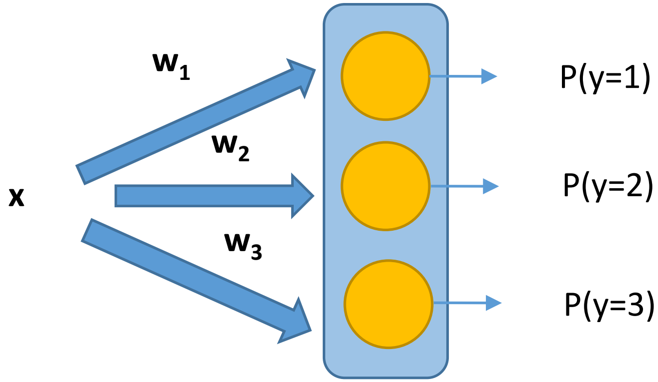
Convolutional Neural Network



Soft-max layer (aka

- Multinomial Logistic Regression
- Polytomous Logistic Regression
- Multiclass Logistic Regression
- Multinomial logit
- …

)



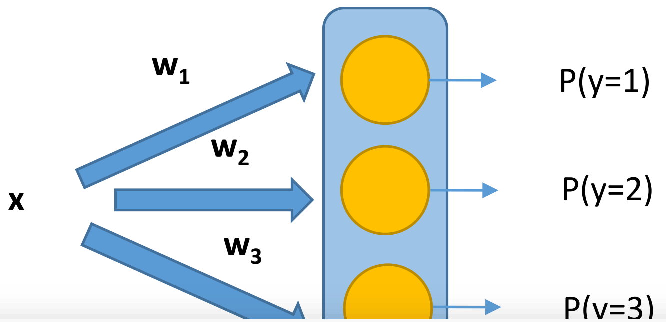
P(y=1)+P(y=2)+P(y=3) = 1

$$P(y=1) = \frac{\exp(\boldsymbol{w_1} \cdot \boldsymbol{x})}{\sum_{k=1}^{3} \exp(\boldsymbol{w_k} \cdot \boldsymbol{x})}$$

$$P(y=2) = \frac{\exp(\boldsymbol{w_2} \cdot \boldsymbol{x})}{\sum_{k=1}^{3} \exp(\boldsymbol{w_k} \cdot \boldsymbol{x})}$$

$$P(y=3) = \frac{\exp(\boldsymbol{w_3} \cdot \boldsymbol{x})}{\sum_{k=1}^{3} \exp(\boldsymbol{w_k} \cdot \boldsymbol{x})}$$

Soft-max layer (ak




```python
# classify the values of the fully-connected sigmoidal layer
layer3 = LogisticRegression(input=layer2.output, n_in=500, n_out=10)
```

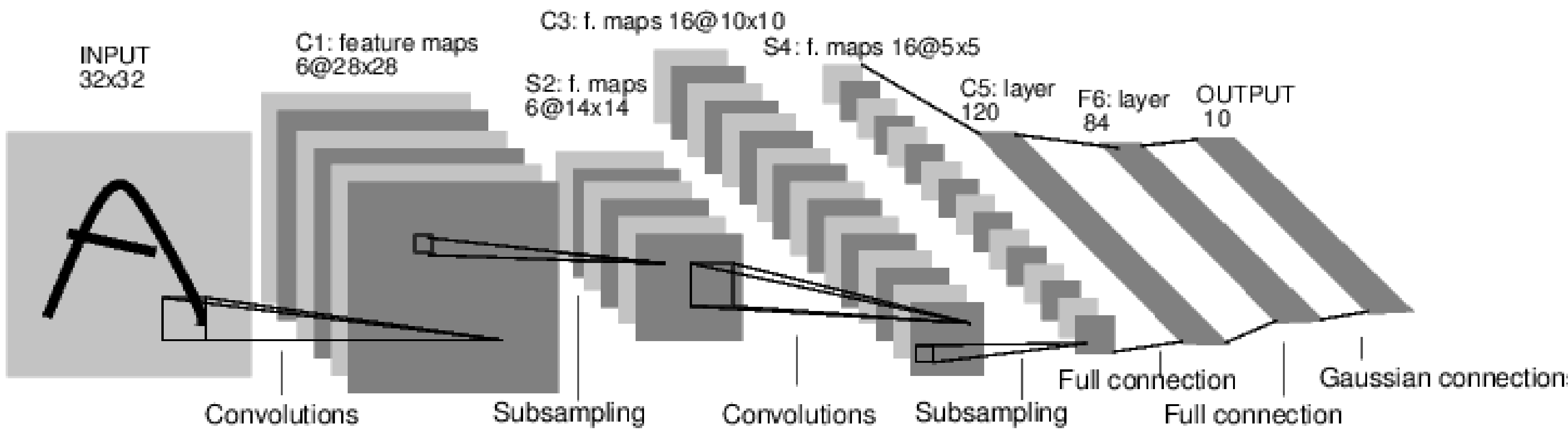
P(y=1)+P(y=2)+P(y=3) = 1

```python
class LogisticRegression(object):
    """Multi-class Logistic Regression Class

    The logistic regression is fully described by a weight matrix :math:`W`
    and bias vector :math:`b`. Classification is done by projecting data
    points onto a set of hyperplanes, the distance to which is used to
    determine a class membership probability.
    """

    def __init__(self, input, n_in, n_out):
        """ Initialize the parameters of the logistic regression

        :type input: theano.tensor.TensorType
        :param input: symbolic variable that describes the input of the
                      architecture (one minibatch)

        :type n_in: int
        :param n_in: number of input units, the dimension of the space in
                     which the datapoints lie

        :type n_out: int
        :param n_out: number of output units, the dimension of the space in
                      which the labels lie

        """
        # start-snippet-1
        # initialize with 0 the weights W as a matrix of shape (n_in, n_out)
        self.W = theano.shared(
            value=numpy.zeros(
                (n_in, n_out),
                dtype=theano.config.floatX
            ),
            name='W',
            borrow=True
        )
        # initialize the biases b as a vector of n_out 0s
        self.b = theano.shared(
            value=numpy.zeros(
                (n_out,),
                dtype=theano.config.floatX
            ),
            name='b',
            borrow=True
        )

        # symbolic expression for computing the matrix of class-membership
        # probabilities
        # Where:
        # W is a matrix where column-k represent the separation hyperplane for
        # class-k
        # x is a matrix where row-j  represents input training sample-j
        # b is a vector where element-k represent the free parameter of
        # hyperplane-k
        self.p_y_given_x = T.nnet.softmax(T.dot(input, self.W) + self.b)

        # symbolic description of how to compute prediction as class whose
        # probability is maximal
        self.y_pred = T.argmax(self.p_y_given_x, axis=1)
        # end-snippet-1
```

LeNet5

[LeCun et al., 1998]

Now we can understand this network!



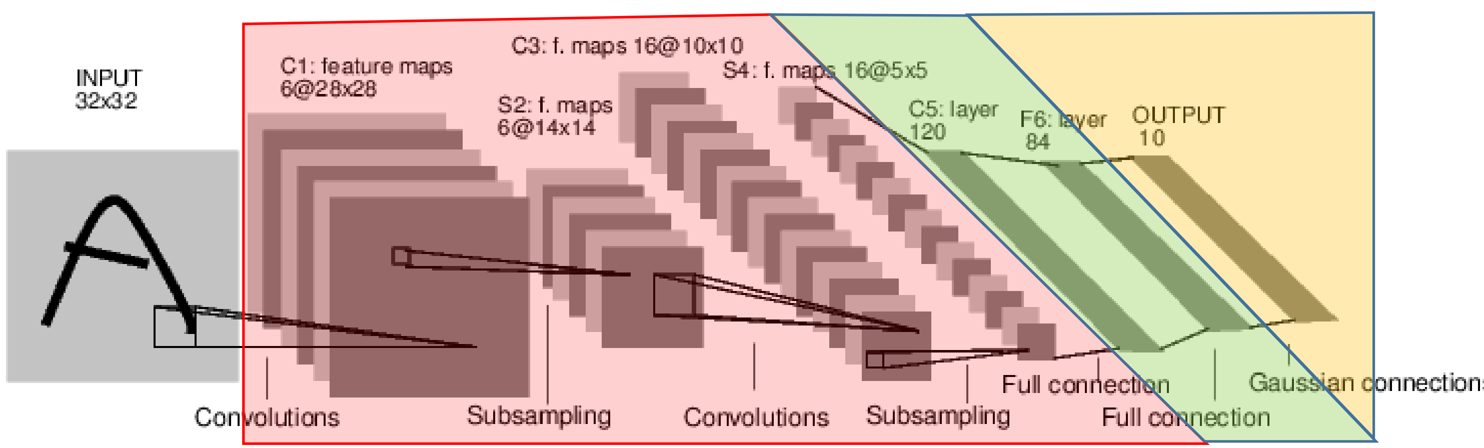
CNNs: a classification pipeline all-in-one

DATA

FEATURE EXTRACTION

Optional:
FEATURE SELECTION
COMBINER

CLASSIFICATION



Convolutional Neural Network

- **Advantages**
  - Unique framework
  - Trained end-to-end
  - No manually defined internal parameters
  - Training on GPUs
  - Available libraries

- **Disadvantages**
  - External hyper-parameters to be (manually) defined
  - Training time approx. days/weeks

Convolutional Neural Network

- **Typical worries of deep learners**
  - How deep?
  - How many fully-connected?
  - Data normalization?
  - Feature map normalization?
  - Initialization strategy?
  - Pooling strategy?
  - …

Convolutional Neural Network

- **Typical worries of deep learners**
  - Why deep? How deep?
  - Why fully-connected? How many fully-connected?
  - Data normalization?
  - Feature map normalization?
  - Initialization strategy?
  - Pooling strategy?
  - …

- **Typical (annoying) answer**
  - It depends on the problem, try several options yourself

Convolutional Neural Network

- **Recipe to build convolutional networks**:
  - Install a good library (Theano, Caffe, Torch)
  - Define number of convolutional layers
  - Define filter size for each layer
  - Define pooling strategy
  - Define fully-connected layer(s)
  - Soft-max layer size depends on the number of classes
  - Collect (a lot of) data
  - Train the network (learn the parameters: weights+bias)

Convolutional Neural Network

- **Recipe to build convolutional networks**:
  - Install a good library (Theano, Caffe, Torch)
  - Define number of convolutional layers
  - Define filter size for each layer
  - Define pooling strategy
  - Define fully-connected layer(s)
  - Soft-max layer size depends on the number of classes
  - Collect (a lot of) data
  - Train the network (learn the parameters: weights+bias)

…HOW TO LEARN THE PARAMETERS ???