# Artificial Neuronal Networks in Python

## *From Scratch!*

**Part II**

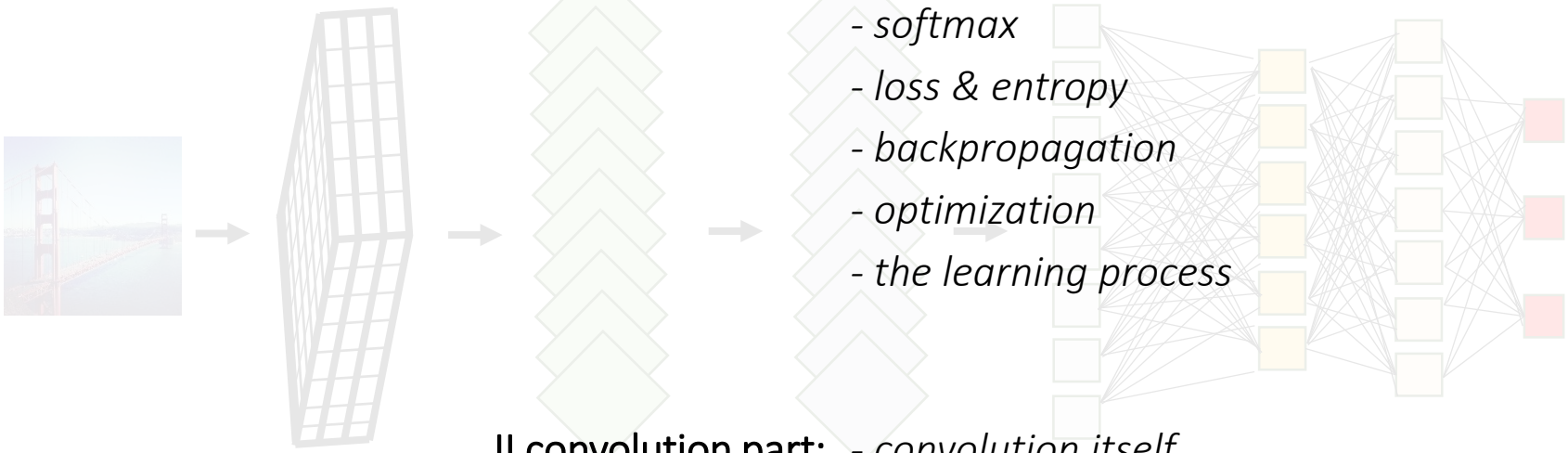

**Dr. Markus Hohle**

*) Yes, I know it's not Python, but it's still funny : )

outline

0 intro

I the core:
- *a single neuron*
- *layers of neurons*
- *activation functions*
- *softmax*
- *loss & entropy*
- *backpropagation*
- *optimization*
- *the learning process*



II convolution part:
- *convolution itself*
- *pooling*
- *sigmoid*
- *flattening*
- *backpropagation - again*
- *testing the CNN & final remarks*

## outline

0 intro

I the core:
- *a single neuron*
- *layers of neurons*
- *activation functions*
- *softmax*
- *loss & entropy*
- *backpropagation*
- *optimization*
- *the learning process*

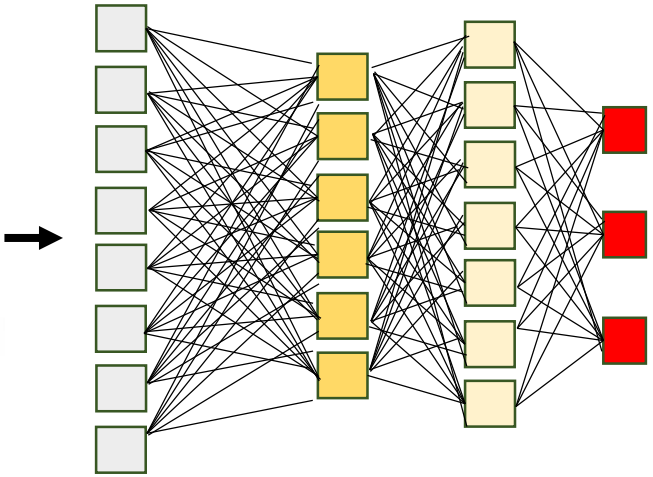
II convolution part:
- *convolution itself*
- *pooling*
- *sigmoid*
- *flattening*
- *backpropagation - again*
- *testing the CNN & final remarks*

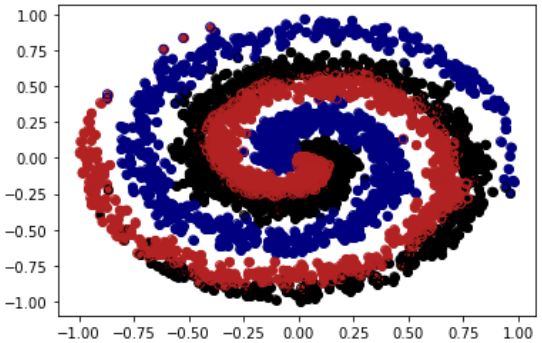
## outline

What we have so far:

- an input layer for a matrix containing coordinates
- an activation layer
- a soft-max layer for calculating probabilities

```
In [6]: print(x)
[[-0.00000000e+00  0.00000000e+00]
 [ 7.94393385e-04  6.09050207e-04]
 [-5.63224823e-05  2.00120958e-03]
 ...
 [-9.27303549e-01 -3.68928357e-01]
 [-9.61714970e-01 -2.70376249e-01]
 [-9.38792891e-01  3.44482085e-01]]
```






input: numerical matrix

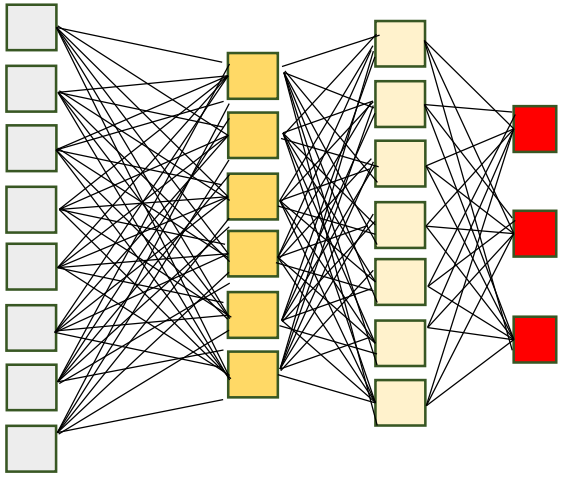
our ANN

output:
- blue dots
- red dots
- black dots

→ classification

Basic building blocks of most ANNs:

- an input layer for a matrix containing coordinates
- an activation layer
- a soft-max layer for calculating probabilities



output:
- cat
- dog
- cat
- cat
- dog

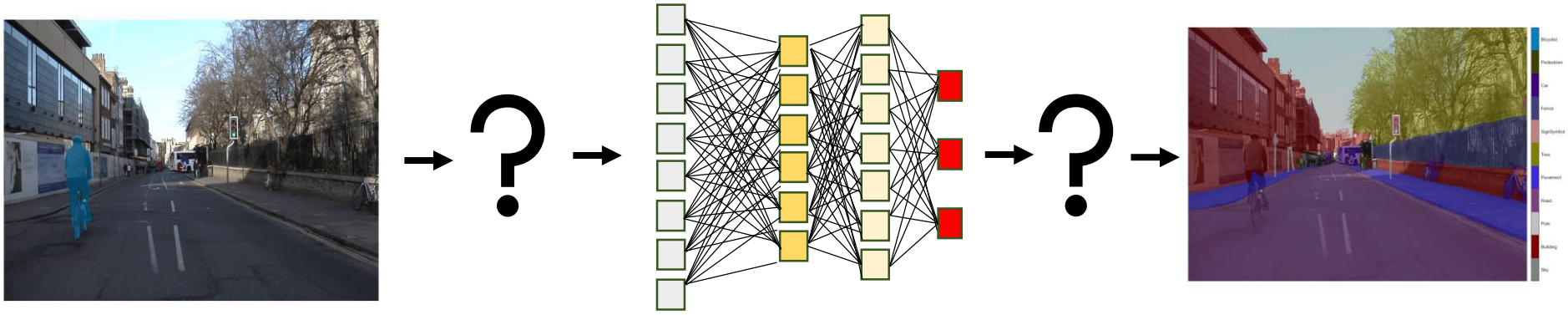
input: image,
aka numerical matrix

our ANN

→ classification

Basic building blocks of most ANNs:

- an input layer for a matrix containing coordinates
- an activation layer
- a soft-max layer for calculating probabilities



input: image,
aka numerical matrix

our ANN

output: categorizing each pixel
aka *Semantic Segmentation*

→ classification

**image curtesy: mathworks**

Basic building blocks of most ANNs:

- an input layer for a matrix containing coordinates
- an activation layer
- a soft-max layer for calculating probabilities

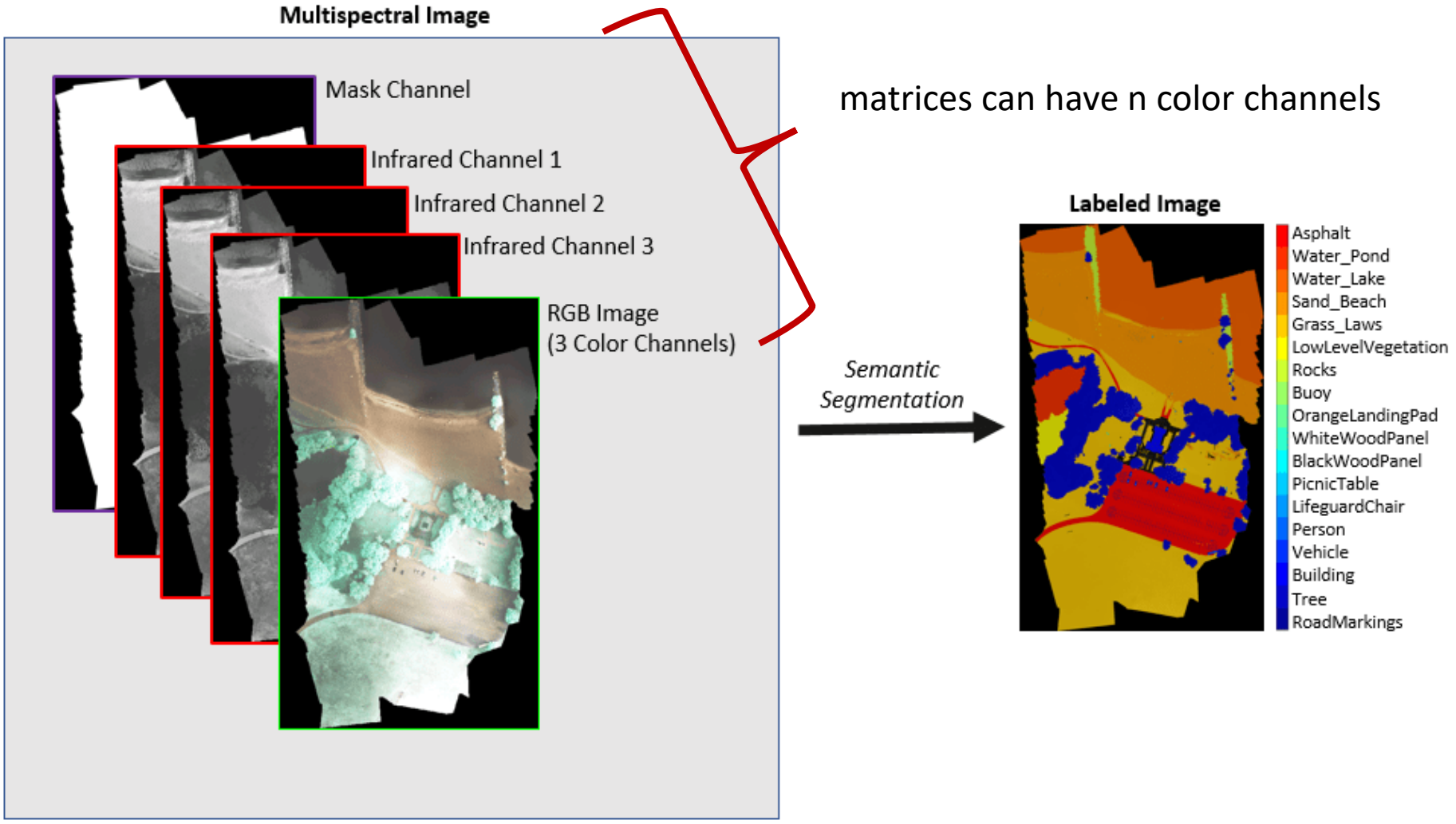
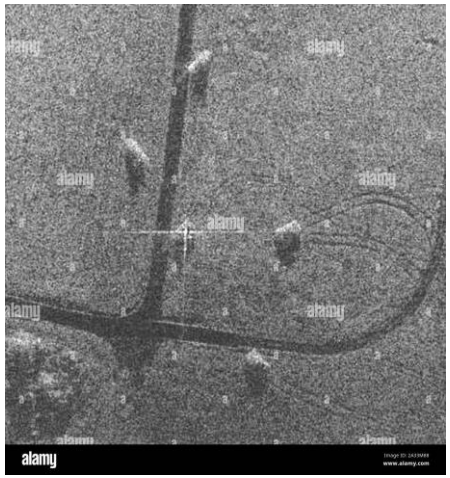


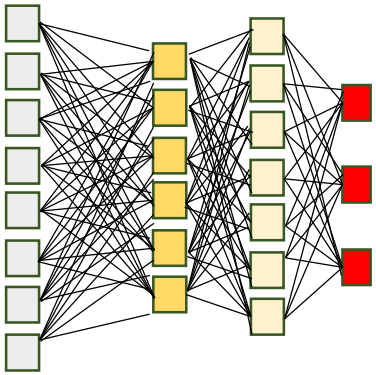

**image curtesy: mathworks**

Basic building blocks of most ANNs:

- an input layer for a matrix containing coordinates
- an activation layer
- a soft-max layer for calculating probabilities




→ ? →



? →

output:

- are these enemy tanks; yes/no
- suitable shell is XYZ
- attack; yes/no

input: image,
aka numerical matrix

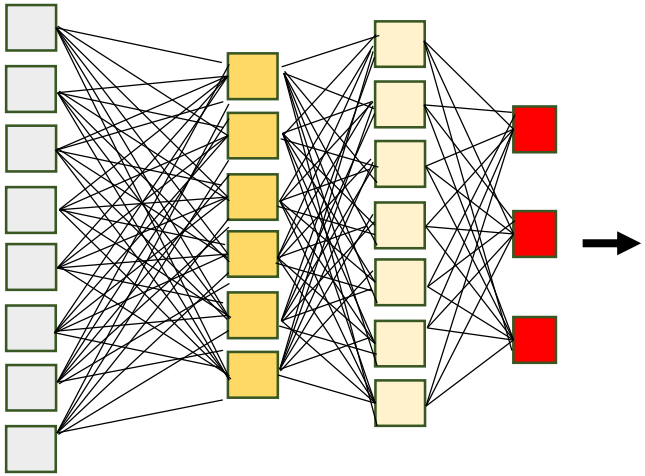
our ANN

→ autonomous systems

Basic building blocks of most ANNs:

- an input layer for a matrix containing coordinates
- an activation layer
- a soft-max layer for calculating probabilities

input: image,
aka numerical matrix



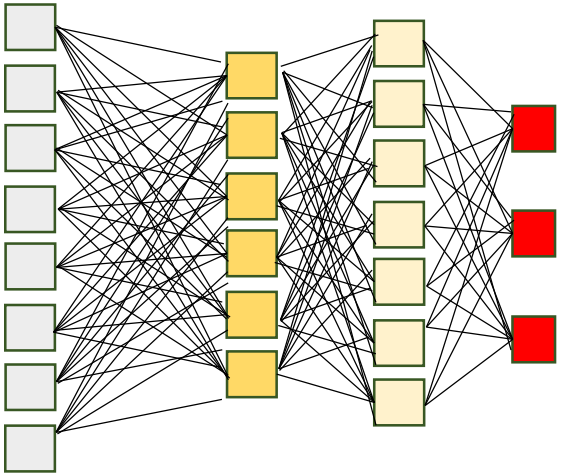
our ANN

output:
turning angle

→ regression

Our goal: creating an ANN that can tell cats from dogs

→ ? →



→ output: - cat
- dog
- cat
- cat
- dog

input: image,
aka numerical matrix

our ANN

→ classification

Our goal: creating an ANN that can tell cats from dogs

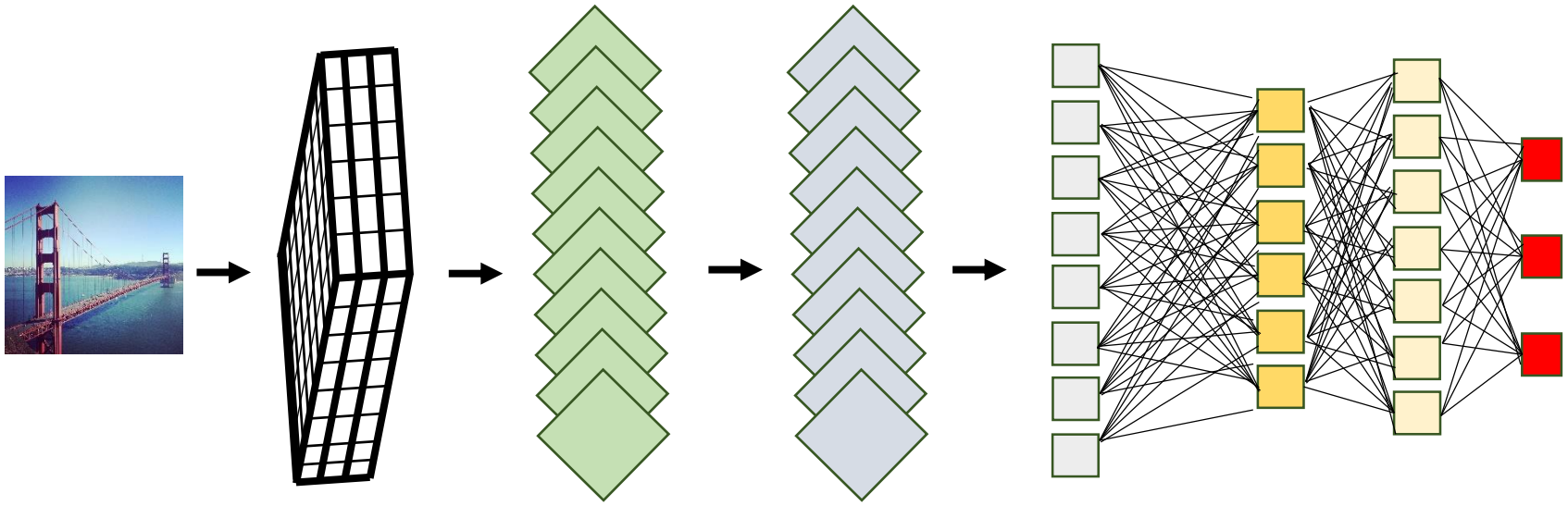


image: 3-d matrix convolution max pool flattening

Our goal: creating an ANN that can tell cats from dogs

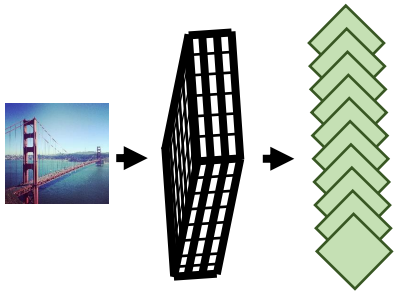


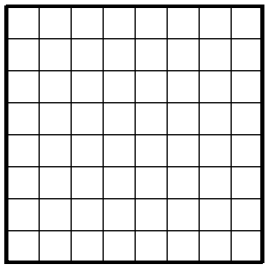
image: 3-d matrix convolution

→ What is convolution and why do we need it?

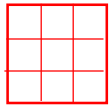
What is convolution?

$$(f * g)(x) := \int_{\mathbb{R}^n} f(\zeta)\boldsymbol{g(x-\zeta)}d\zeta$$

image ***f*** and filter ***g***



**image**



**filter (aka kernel)**

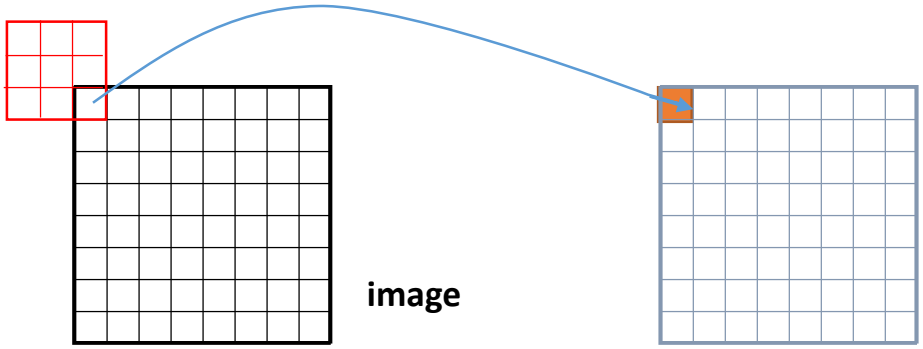
What is convolution?

$$(f * g)(x) := \int_{\mathbb{R}^n} f(\zeta)\, g(x - \zeta)\, d\zeta$$

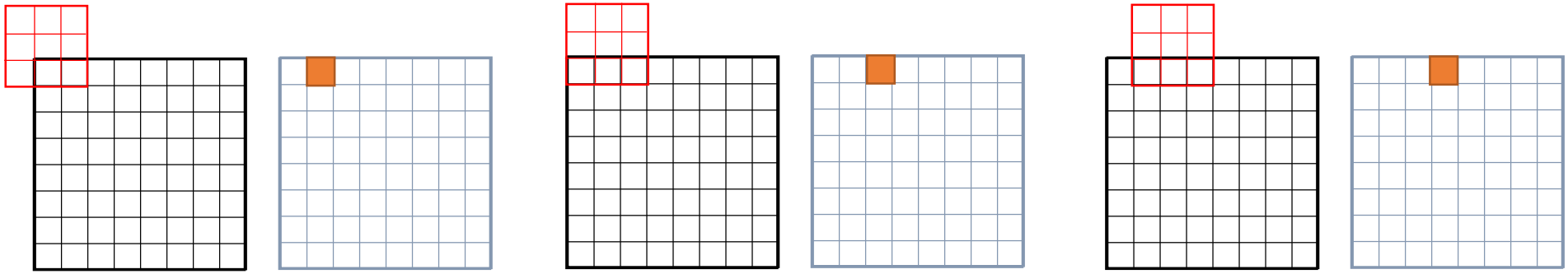
image ***f*** and filter ***g***

**filter (aka kernel)**

**convoluted image**




**1) multiplying matrix values pixelwise**
**2) summing up products**
**3) → value for new matrix**



What is convolution?

$$(f * g)(x) := \int_{\mathbb{R}^n} f(\zeta)\, g(x - \zeta)\, d\zeta$$

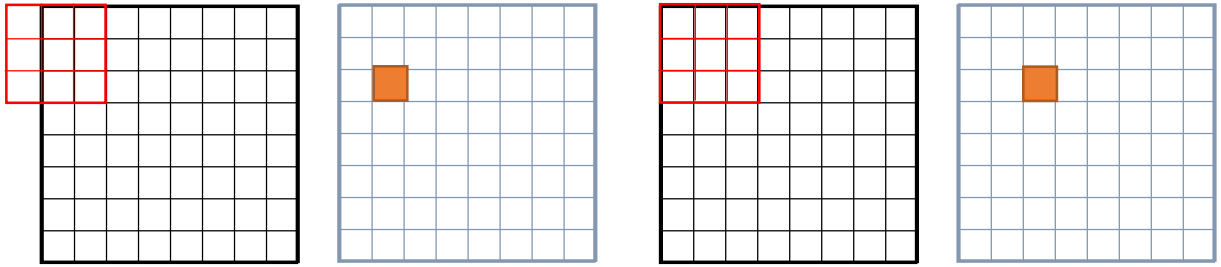
image ***f*** and filter ***g***

**1) multiplying matrix values pixelwise**
**2) summing up products**
**3) → value for new matrix**

**filter (aka kernel)**

**image**

**convoluted image**



**and so on…**

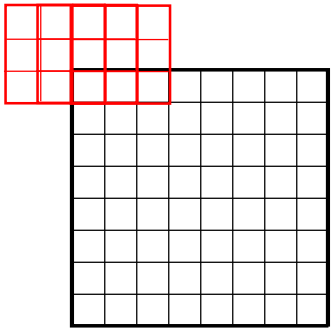
What is convolution?

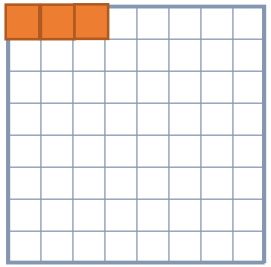
$$(f * g)(x) := \int_{\mathbb{R}^n} f(\zeta)\boldsymbol{g(x-\zeta)}d\zeta$$

image ***f*** and filter ***g***

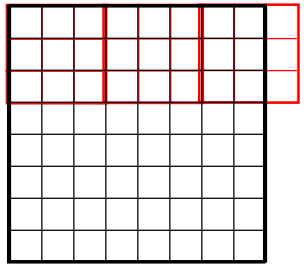
**1) multiplying matrix values pixelwise**
**2) Summing up products**
**3) → value for new matrix**

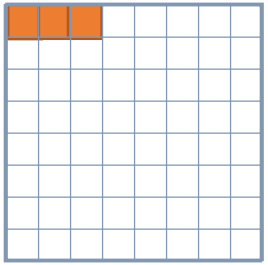
different techniques:





padding = 2; stride length = 1





padding = 0; stride length = 3

What is convolution?

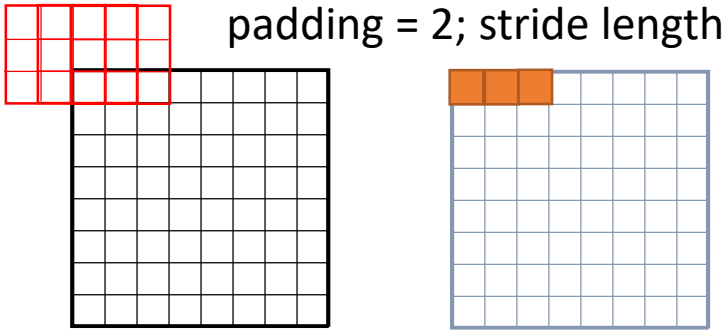
$$(f * g)(x) := \int_{\mathbb{R}^n} f(\zeta)\boldsymbol{g(x-\zeta)}d\zeta$$

image ***f*** and filter ***g***

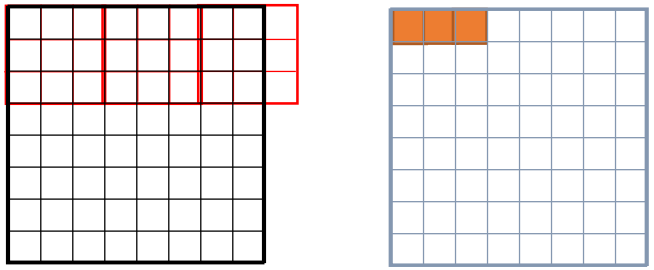
different techniques:

**1) multiplying matrix values pixelwise**
**2) Summing up products**
**3) → value for new matrix**

padding = 2; stride length = 1



padding = 0; stride length = 3



the resulting image has the following size (N is the number of rows/columns):

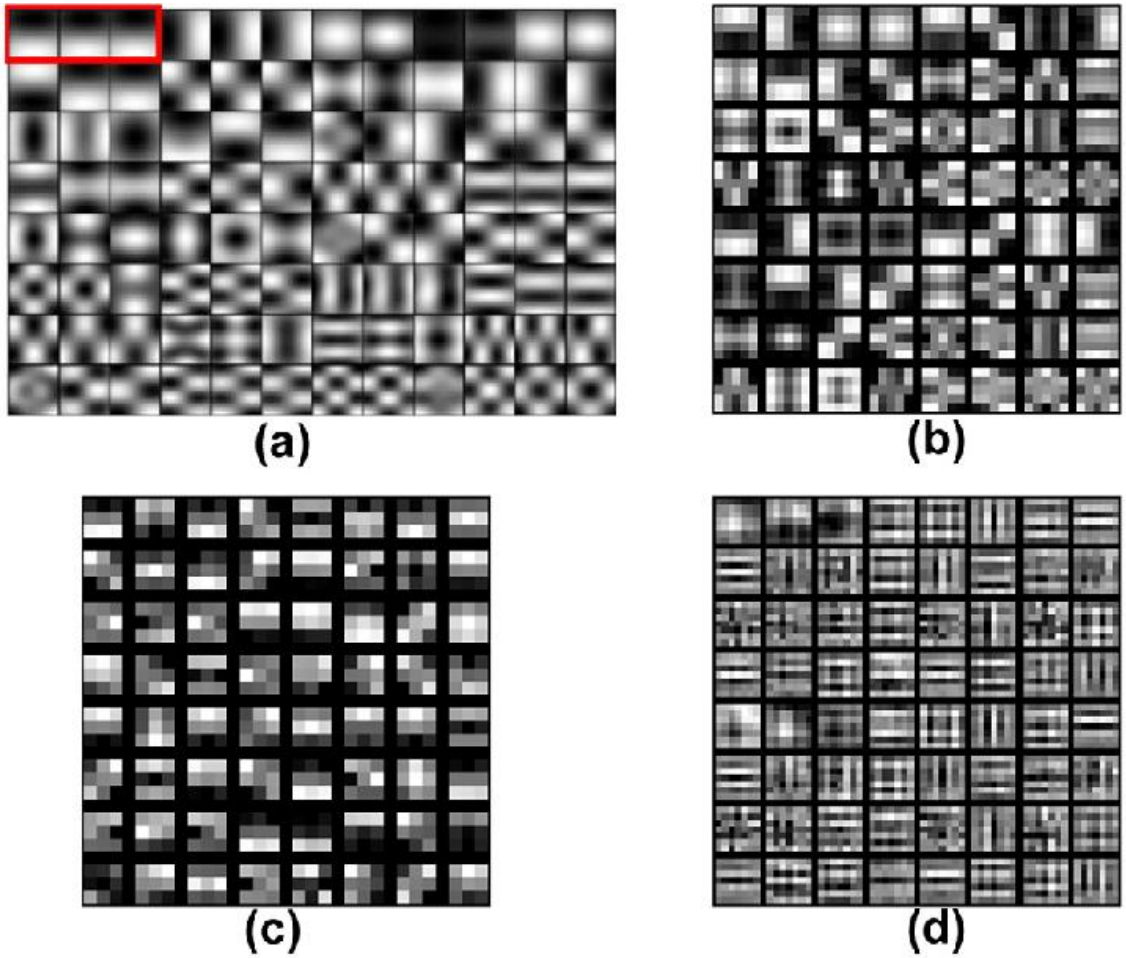
$$N_{out} = \frac{\left(N_{in} - N_{filt} + 2 * padding\right)}{stride\ length} + 1$$

What is convolution?

$$(f * g)(x) := \int_{\mathbb{R}^n} f(\zeta) g(x - \zeta) d\zeta$$

image ***f*** and filter ***g***

filters:




**1) multiplying matrix values pixelwise**
**2) summing up products**
**3) → value for new matrix**

**DOI:10.1016/j.actbio.2017.09.025**

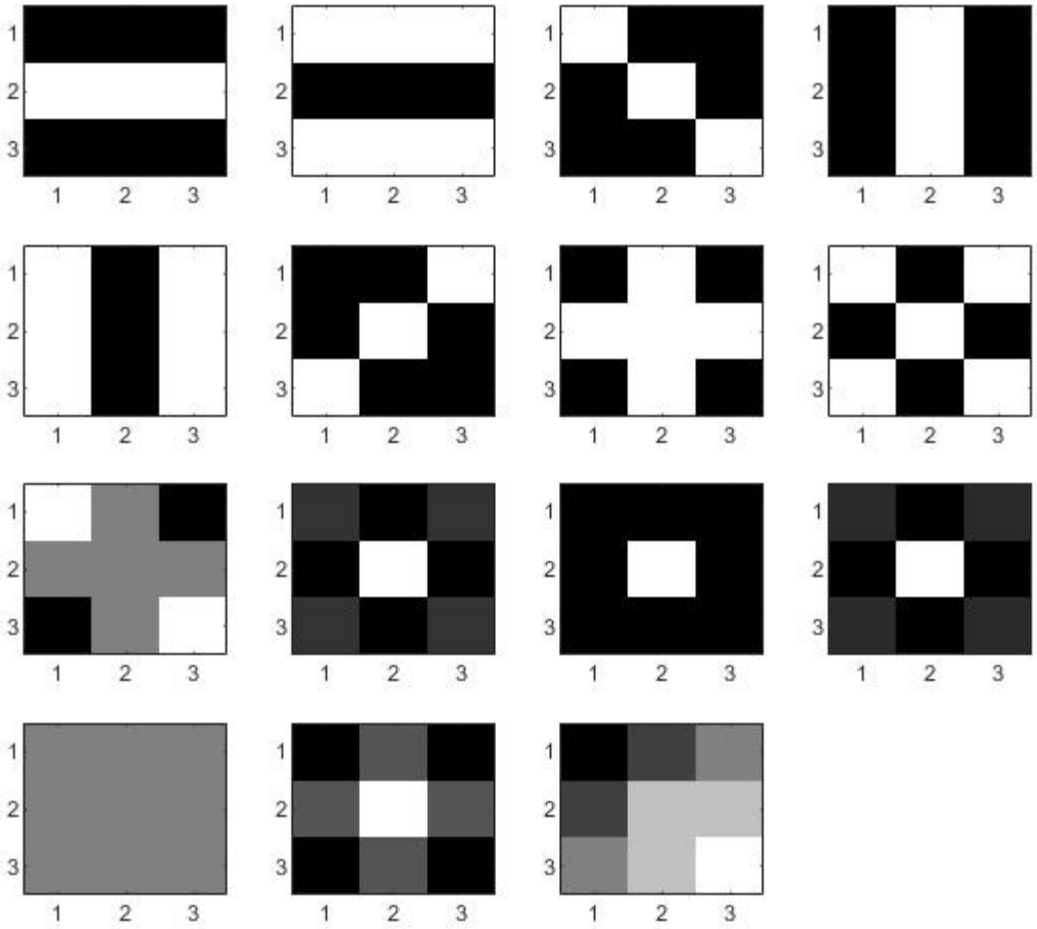
Why do we need it?

$$(f * g)(x) := \int_{\mathbb{R}^n} f(\zeta)\boldsymbol{g(x-\zeta)}d\zeta$$

image ***f*** and filter ***g***

image ***f***

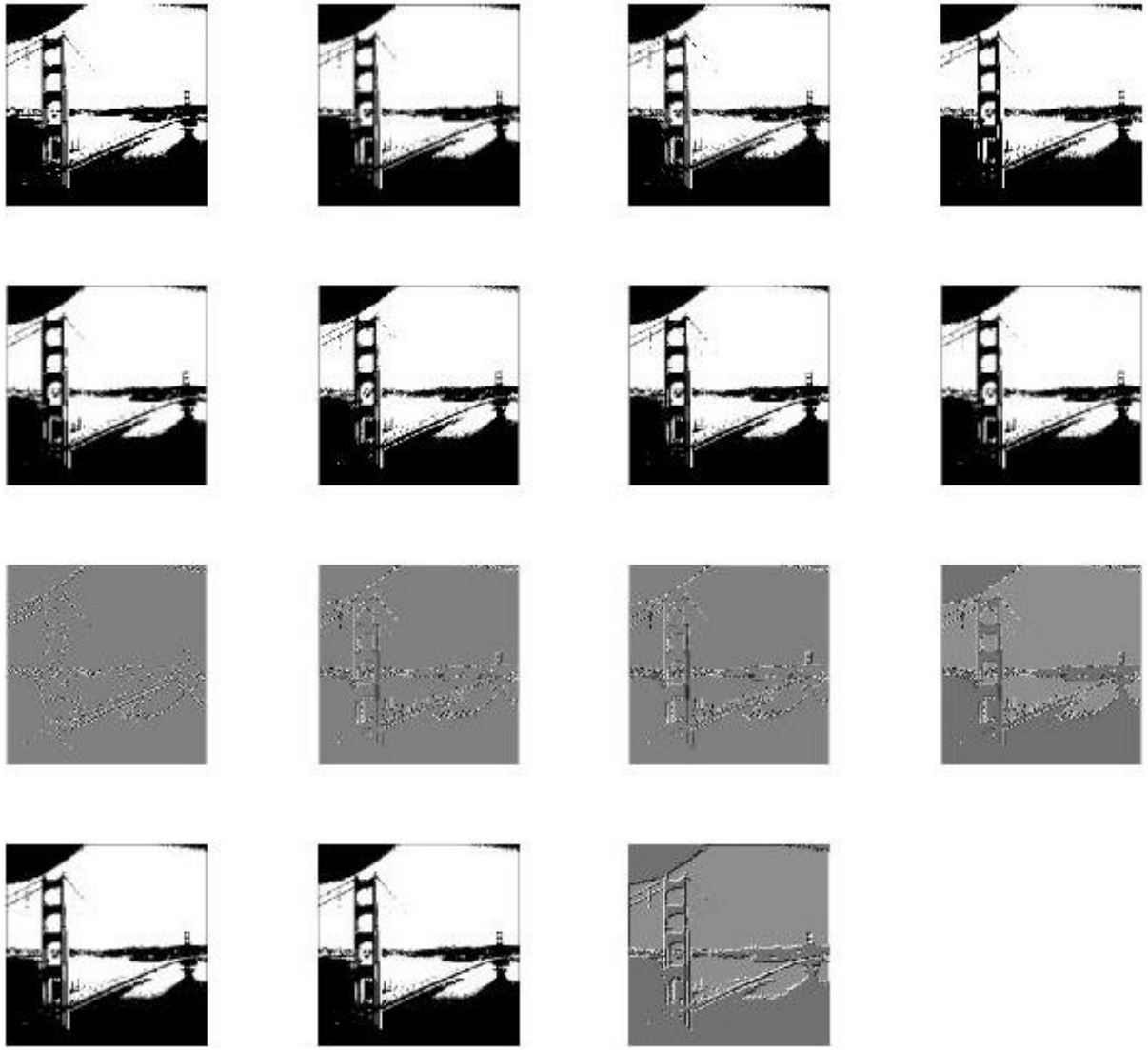
filter ***g***



Why do we need it?

$$(f * g)(x) := \int_{\mathbb{R}^n} f(\zeta) g(x - \zeta) d\zeta$$

image ***f*** and filter ***g***

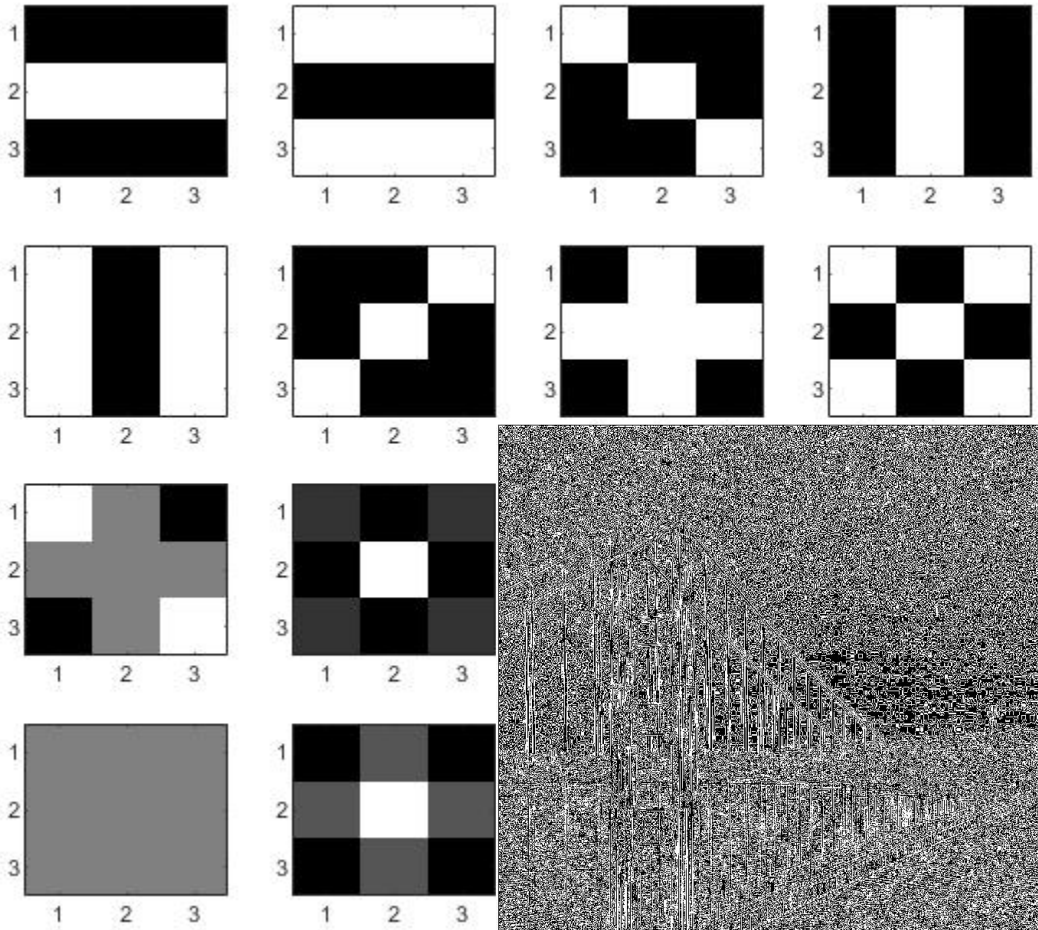


Why do we need it?

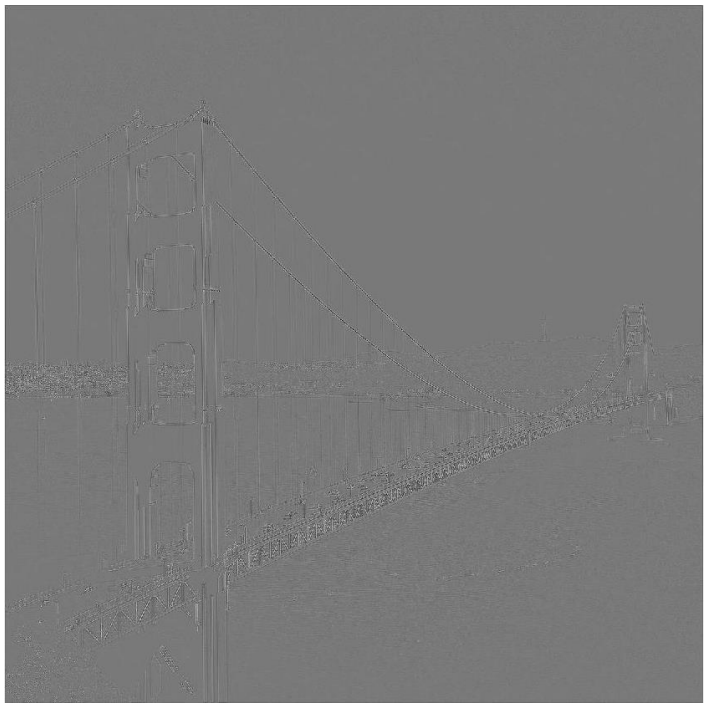
$$(f * g)(x) := \int_{\mathbb{R}^n} f(\zeta) g(x - \zeta) d\zeta$$

image ***f*** and filter ***g***

filter ***g***



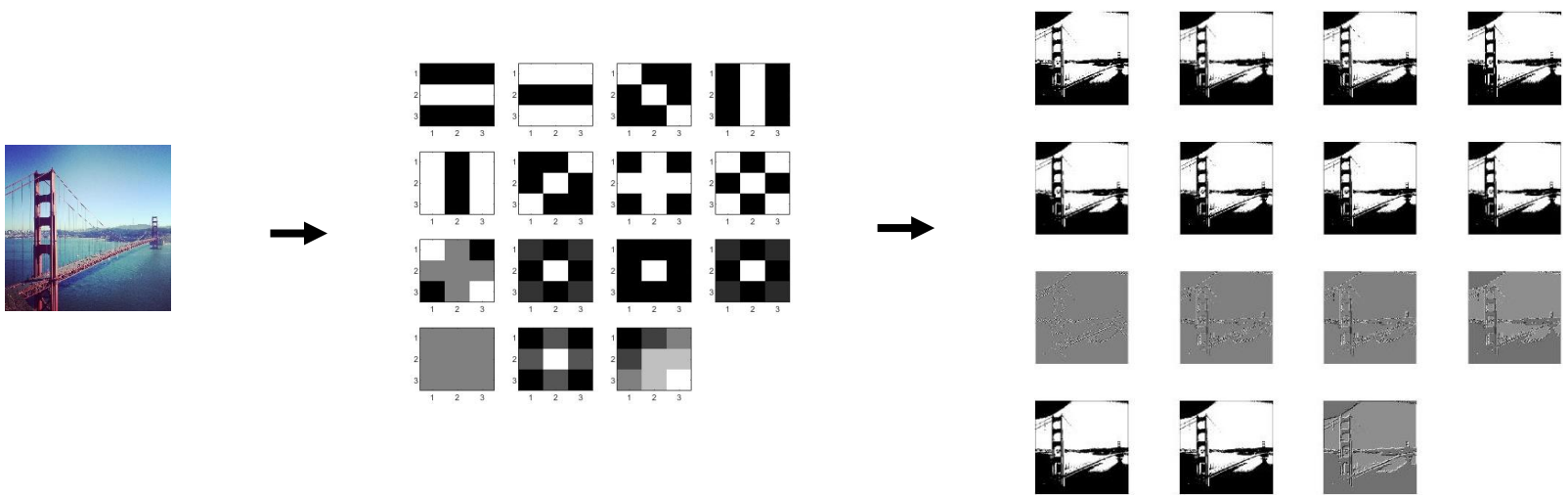




Why do we need it?

$$(f * g)(x) := \int_{\mathbb{R}^n} f(\zeta) g(x - \zeta) d\zeta$$

image ***f*** and filter ***g***



idea:
- filtering relevant features → for classification
- Which features are relevant? → well, that's what the ANN learns
→ learning filter weights

Convolution is the main part → **C**onvolutional **N**euronal **N**etwork

$$(f * g)(x) := \int_{\mathbb{R}^n} f(\zeta) g(x - \zeta) d\zeta$$

This is all somewhat abstract → let us write a code, that

1) loads an image
2) loads different convolution filter (aka kernel)
3) shows the filtered image

I: reading & displaying images in python

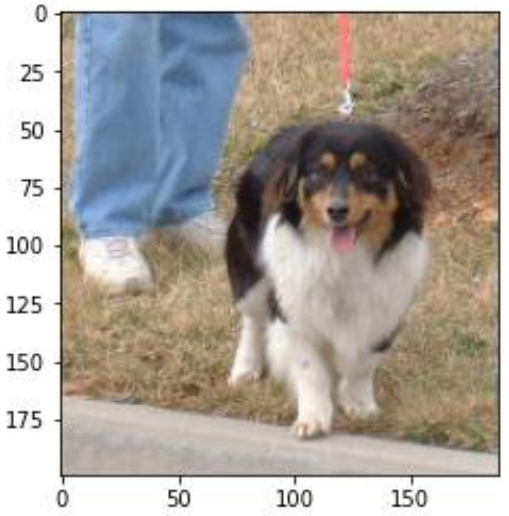
```
import matplotlib.pyplot as plt
```

the commands we are going to need are part of `matplotlib`

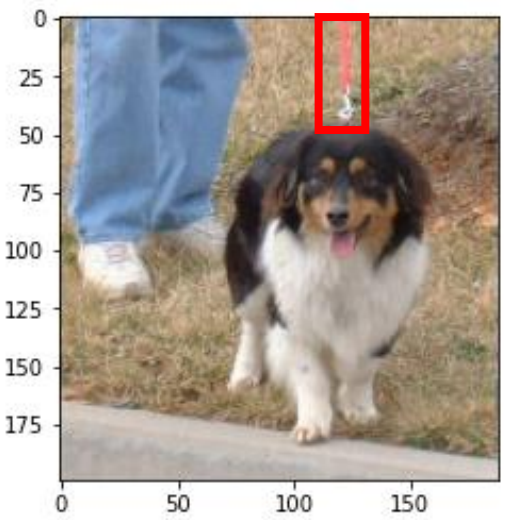
```
I = plt.imread('C:/my_path/Dog/2.jpg')
```

| Nar | Type | Size | Value |
|---|---|---|---|
| I | Array of uint8 | (199, 188, 3) | [[[190 161 129]<br>[189 160 130] |

```
plt.imshow(I)
```



$$(f * g)(x) := \int_{\mathbb{R}^n} f(\zeta) g(x - \zeta) d\zeta$$

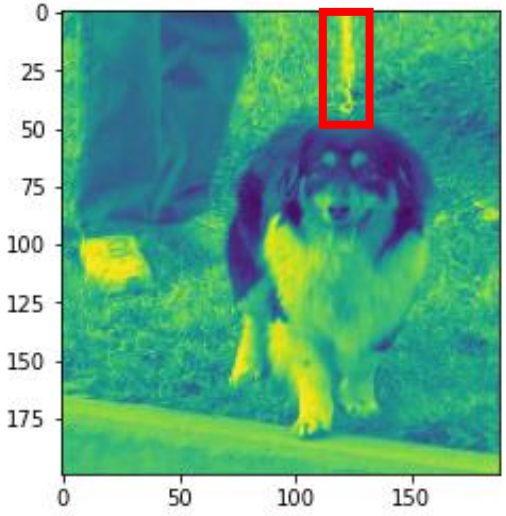



I: reading & displaying images in python

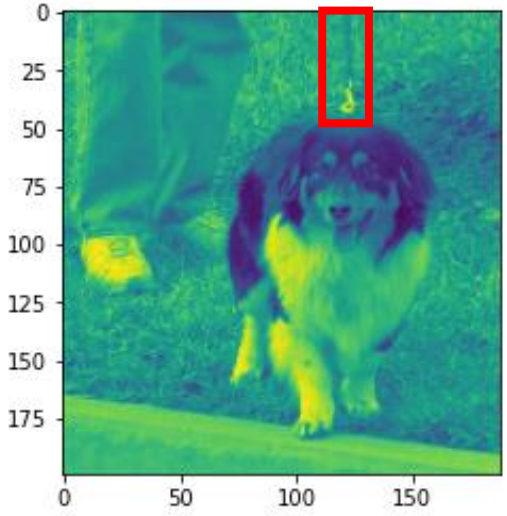
```
import matplotlib.pyplot as plt
I = plt.imread('C:/my_path/Dog/2.jpg')
```

| Nar | Type | Size | Value |
|---|---|---|---|
| I | Array of uint8 | (199, 188, 3) | [[[190 161 129]<br>[189 160 130] |

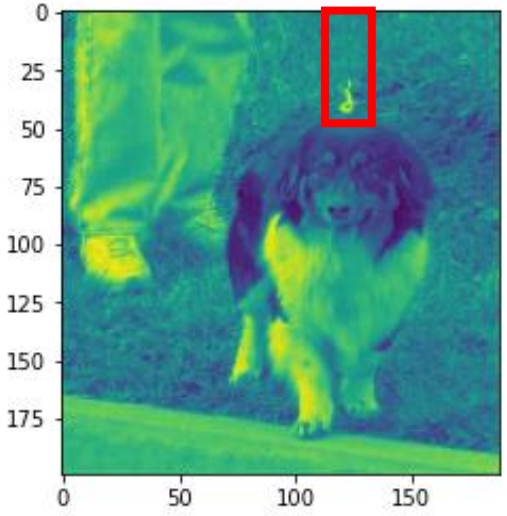
```
plt.imshow(I[:,:,0])
```




```
plt.imshow(I[:,:,1])
```




```
plt.imshow(I[:,:,2])
```



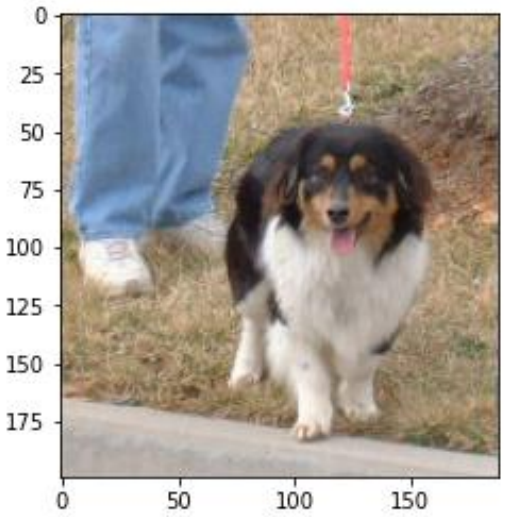
I: reading & displaying images in python

```
def My_Convolution(*Image):   # we wanna read the image as an optional
                              # input argument


    import matplotlib.pyplot as plt
    import numpy as np
    from scipy.signal import convolve as Conv # N-Dim convolution


#displaying the image
    if Image:
        # we need the loop because python interprets
        # the input as list of arbitrary size
         for Image in Image:
             plt.imshow(Image)
    else:
         # read some default image
         Image = plt.imread('C:/my_path/Dog/2.jpg')
         plt.imshow(Image)
```

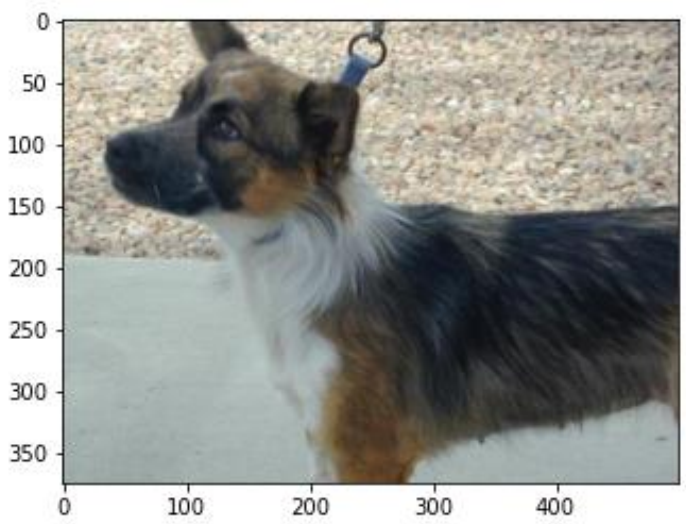
## I: reading & displaying images in python

compile and run the code:

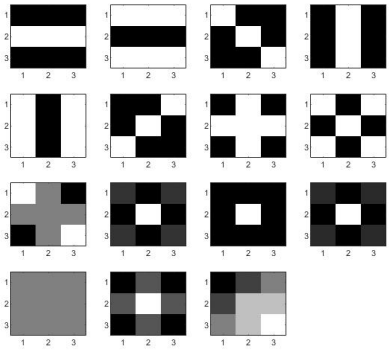
`My_Convolution()`




`My_Convolution(I)`




now it's time for the filters:



## II: Convolution with different filter

```
def My_Convolution(*Image):

...
   else:
        Image = plt.imread('C:/my_path/Dog/2.jpg')
        plt.imshow(Image)


 #creating kernels
    K1  = np.array([[0, 0, 0], [0, 1, 0], [0, 0, 0]])
    #edges
    K2  = np.array([[1, 0, -1], [0, 0, 0], [-1, 0, 1]])
    K3  = np.array([[0, 1, 0], [1, -4, 1], [0, 1, 0]])
    K4  = np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]])
    #sharpen
    K5  = np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]])
    #blur
    K6  = np.array([[1, 1, 1], [1, 1, 1], [1, 1, 1]])
    K6  = K6/9
    K7  = np.array([[1, 2, 1], [2, 4, 2], [1, 2, 1]])
```

## II: Convolution with different filter

```
def My_Convolution(*Image):

...
 #creating kernels
    K1  = np.array([[0, 0, 0], [0, 1, 0], [0, 0, 0]])
    #edges
    K2  = np.array([[1, 0, -1], [0, 0, 0], [-1, 0, 1]])
    K3  = np.array([[0, 1, 0], [1, -4, 1], [0, 1, 0]])
    K4  = np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]])
    #sharpen
    K5  = np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]])
    #blur
    K6  = np.array([[1, 1, 1], [1, 1, 1], [1, 1, 1]])
    K6  = K6/9
    K7  = np.array([[1, 2, 1], [2, 4, 2], [1, 2, 1]])

    #skeewed
    K8  = np.array([[-2, -1, 0], [-1, 1, 1], [0, 1, 2]])
    #misc
    K9  = np.array([[1, 0, 1], [0, 1, 0], [1, 0, 1]])
    K10 = np.array([[1, 1, 1], [0, 0, 0], [1, 1, 1]])
    K11 = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
    K12 = np.array([[0, 0, 1], [0, 1, 0], [1, 0, 0]])
    K13 = np.array([[1, 0, 1], [1, 0, 1], [1, 0, 1]])
```

## II: Convolution with different filter

```
def My_Convolution(*Image):

...
    #skeewed
    K8  = np.array([[-2, -1, 0], [-1, 1, 1], [0, 1, 2]])
    #misc
    K9  = np.array([[1, 0, 1], [0, 1, 0], [1, 0, 1]])
    K10 = np.array([[1, 1, 1], [0, 0, 0], [1, 1, 1]])
    K11 = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
    K12 = np.array([[0, 0, 1], [0, 1, 0], [1, 0, 0]])
    K13 = np.array([[1, 0, 1], [1, 0, 1], [1, 0, 1]])
```

putting all filters into one matrix:

```
    K   = np.dstack((K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13))
```

```
    # some parameter we are going to need soon:
    NK = K.shape[2]
    NC = Image.shape[2]
```

## II: Convolution with different filter

```
def My_Convolution(*Image):

...
    NK = K.shape[2]
    NC = Image.shape[2]

PS = np.math.ceil(NK**0.5)

    for i in range(NC):

        plt.figure(figsize = (15, 12))
        plt.suptitle("after convolution of channel " + str(i+1),\
                     fontsize = 20, y = 0.95)
        plt.subplots_adjust(hspace = 0.5)

        for k in range(NK):

            plt.subplot(PS,PS,k+1)
            Out = Conv(Image[:,:,i],K[:,:,k])
            plt.imshow(Out, cmap = 'gray')

        plt.show()
```

We want to display the filtered images for each filter and color channel

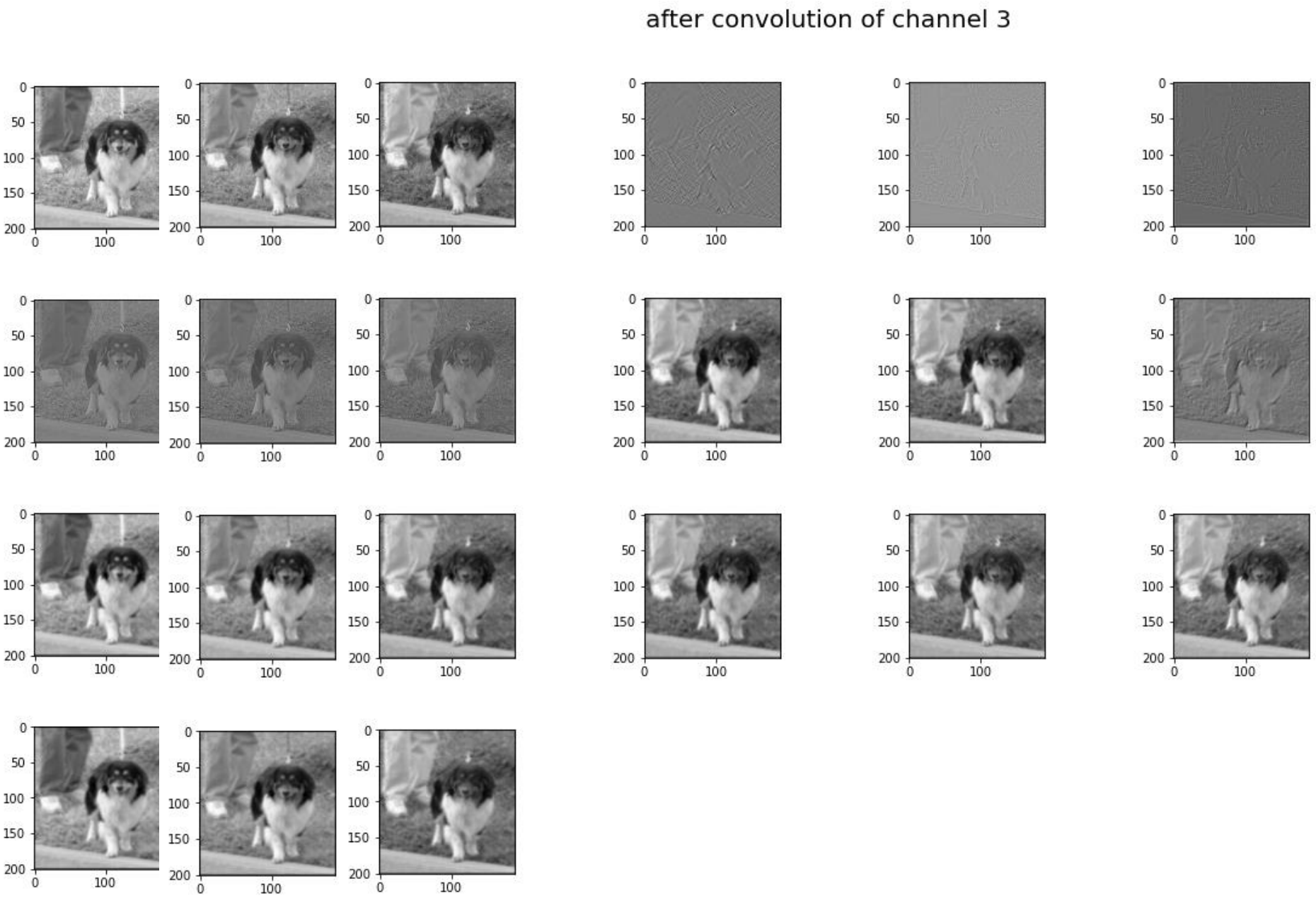
one figure for each channel

creating different subplots

the actual convolution

## II: Convolution with different filter

compile and run the code:



## II: Convolution with different filter

This has worked perfectly, however, we would like to understand **convolution entirely**

→ writing our own convolution routine
→ input arguments: the image itself, stride length, padding
→ output arguments: convoluted image

aim:
→ once this code is working, we can copy/paste it into our ANN with slight modifications
→ especially for backpropagation it will become crucial to understand what we are doing (will be the most laborious part)

```
def ConvSelfMade(*Image, K, padding = 0, stride = 1):

    import numpy as np
    import matplotlib.pyplot as plt
```

## II: Convolution with different filter

```
def ConvSelfMade(*Image, K, padding = 0, stride = 1):

    import numpy as np
    import matplotlib.pyplot as plt

        if Image:
                for Image in Image:
                    plt.imshow(Image)
                     plt.show()
        else:
                Image = plt.imread('C:/my_path/Dog/2.jpg')
                plt.imshow(Image)
                plt.show()
```

again, we'd like to keep the option of loading a default image

## II: Convolution with different filter

```
def ConvSelfMade(*Image, K, padding = 0, stride = 1):

    import numpy as np
    import matplotlib.pyplot as plt

            if Image:
                for Image in Image:
                  plt.imshow(Image)
                   plt.show()
            else:
                  Image = plt.imread('C:/my_path/Dog/2.jpg')
                  plt.imshow(Image)
                  plt.show()

    xImgShape   = Image.shape[0]
    yImgShape   = Image.shape[1]
    numChans    = Image.shape[2]

    xK          = K.shape[0]
    yK          = K.shape[1]

    #determining size of output image
     xOutput = int(((xImgShape - xK + 2 * padding)/stride) + 1)
     yOutput = int(((yImgShape - yK + 2 * padding)/stride) + 1)
```

determining sizes of image & kernel
third image dimension = number of color channels

recall: $N_{out} = \frac{(N_{in} - N_{filt} + 2 * padding)}{stride\ length} + 1$

## II: Convolution with different filter

```
def ConvSelfMade(*Image, K, padding = 0, stride = 1):
...
    xImgShape  = Image.shape[0]
    yImgShape  = Image.shape[1]
    numChans   = Image.shape[2]

    xK         = K.shape[0]
    yK         = K.shape[1]

    #determining size of output image
    xOutput = int(((xImgShape - xK + 2 * padding)/stride) + 1)
    yOutput = int(((yImgShape - yK + 2 * padding)/stride) + 1)
```

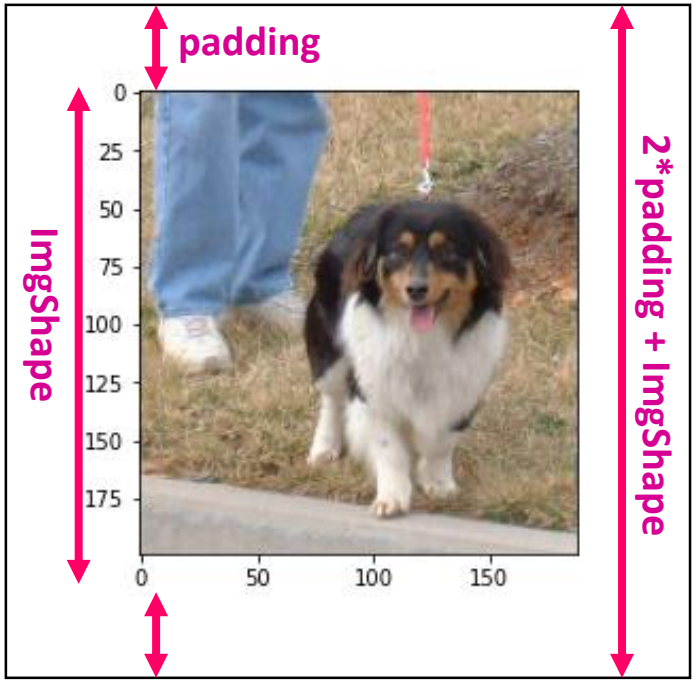
Note: we don't use np.empty here, since it would lead to NaNs

```
    #initializing empty output matrix
    output = np.zeros((xOutput,yOutput,numChans))


    imagePadded = np.zeros(((xImgShape + padding*2),
                            (yImgShape + padding*2), numChans))

    imagePadded[int(padding):int(padding + xImgShape),\
                int(padding):int(padding + yImgShape),:] = Image
```

## II: Convolution with different filter

```
def ConvSelfMade(*Image, K, padding = 0, stride = 1):
...
```

```
 #initializing empty output matrix
 output = np.zeros((xOutput,yOutput,numChans))

imagePadded = np.zeros(((xImgShape + padding*2),
                        (yImgShape + padding*2), numChans))

imagePadded[int(padding):int(padding + xImgShape),\
            int(padding):int(padding + yImgShape),:] = Image
```



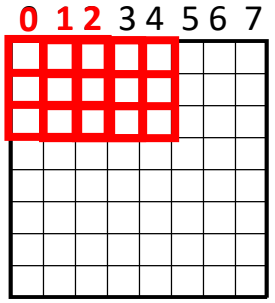
## II: Convolution with different filter

```
def ConvSelfMade(*Image, K, padding = 0, stride = 1):
...

imagePadded = Image

 for c in range(numChans):# loop over channels
        for y in range(yOutput):# loop over y axis of output
            for x in range(xOutput):# loop over x axis of output

                # finding corners of the current "slice"
                y_start = y*stride
                y_end   = y*stride + yK
                x_start = x*stride
                x_end   = x*stride + xK

                #selecting the current part of the image
                current_slice = imagePadded[x_start:x_end,\
                                            y_start:y_end,c]

                #the actual convolution part
                s              = np.multiply(current_slice,K)
                output[x,y,c] = np.sum(s)
```

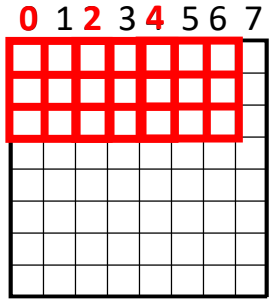
## II: Convolution with different filter

```
def ConvSelfMade(*Image, K, padding = 0, stride = 1):
...
```

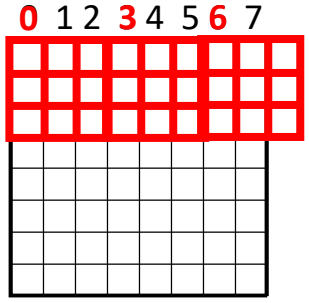
```
for c in range(numChans):# loop over channels
        for y in range(yOutput):# loop over y axis of output
            for x in range(xOutput):# loop over x axis of output

                # finding corners of the current "slice"
                y_start = y*stride
                y_end   = y*stride + yK
                x_start = x*stride
                x_end   = x*stride + xK

                #selecting the current part of the image
                current_slice = imagePadded[x_start:x_end,\
                                            y_start:y_end,c]

                #the actual convolution part
                s             = np.multiply(current_slice,K)
                output[x,y,c] = np.sum(s)
```




stride = 1




stride = 2




stride = 3

## II: Convolution with different filter

```
def ConvSelfMade(*Image, K, padding = 0, stride = 1):
...
 for c in range(numChans):# loop over channels
        for y in range(yOutput):# loop over y axis of output
            for x in range(xOutput):# loop over x axis of output

                # finding corners of the current "slice"
                y_start = y*stride
                y_end   = y*stride + yK
                x_start = x*stride
                x_end   = x*stride + xK

                #selecting the current part of the image
                current_slice = imagePadded[x_start:x_end,\
                                    y_start:y_end,c]

                #the actual convolution part
                s             = np.multiply(current_slice,K)
                output[x,y,c] = np.sum(s)
```

finally, in ANNs,
the results are summed
over the color channels

```
plt.imshow(output.sum(2), cmap = 'gray')

plt.title('after convolution with padding'\
                  '=' + str(padding) + ' and  stride=' + str(stride))
plt.show()

return(output)
```



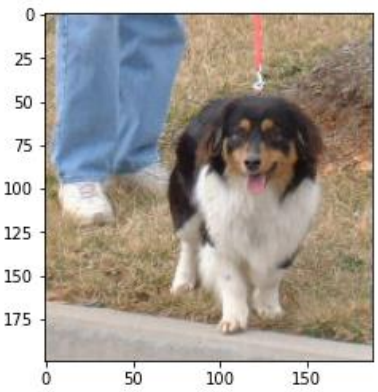
## II: Convolution with different filter

Let us now test the code with different settings for one **particular kernel**

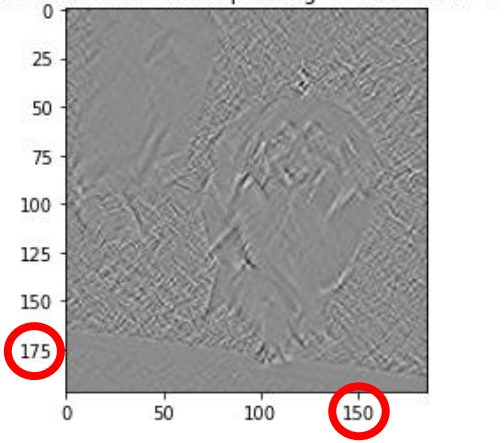
```
kernel = np.array([[1, 0, -1], [0, 0, 0], [-1, 0, 1]])

ConvSelfMade(K = kernel, padding = 0, stride = 1)

ConvSelfMade(K = kernel, padding = 0, stride = 2)

ConvSelfMade(K = kernel, padding = 0, stride = 5)

ConvSelfMade(K = kernel, padding = 0, stride = 10)

ConvSelfMade(K = kernel, padding = 0, stride = 20)

ConvSelfMade(K = kernel, padding = 30, stride = 1)
```

## II: Convolution with different filter

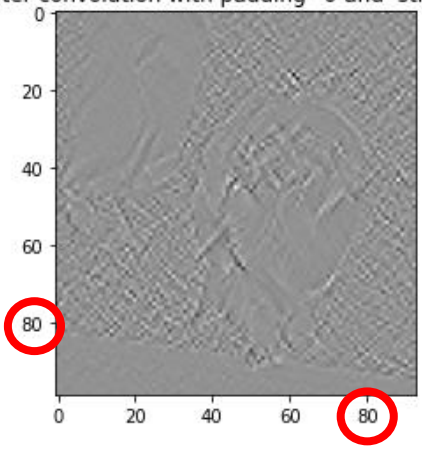
Let us now test the code with different settings for one **particular kernel**



after convolution with padding=0 and stride=1

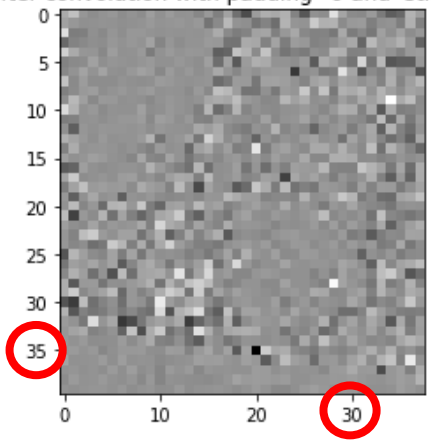


after convolution with padding=0 and stride=2

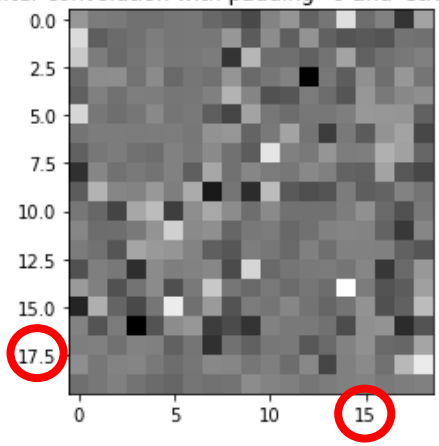


$$N_{out} = \frac{(N_{in} - N_{filt} + 2 * padding)}{\textbf{\textit{stride length}}} + 1$$

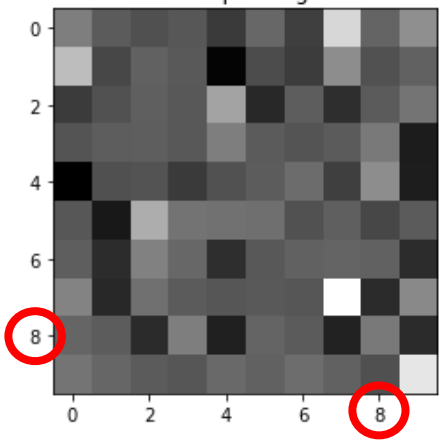
after convolution with padding=0 and stride=5



after convolution with padding=0 and stride=10

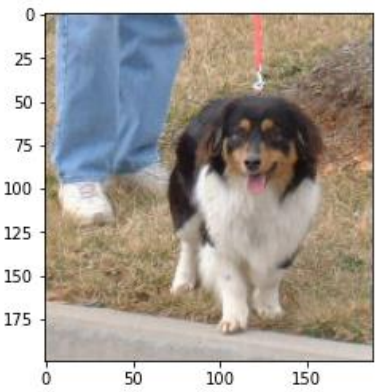


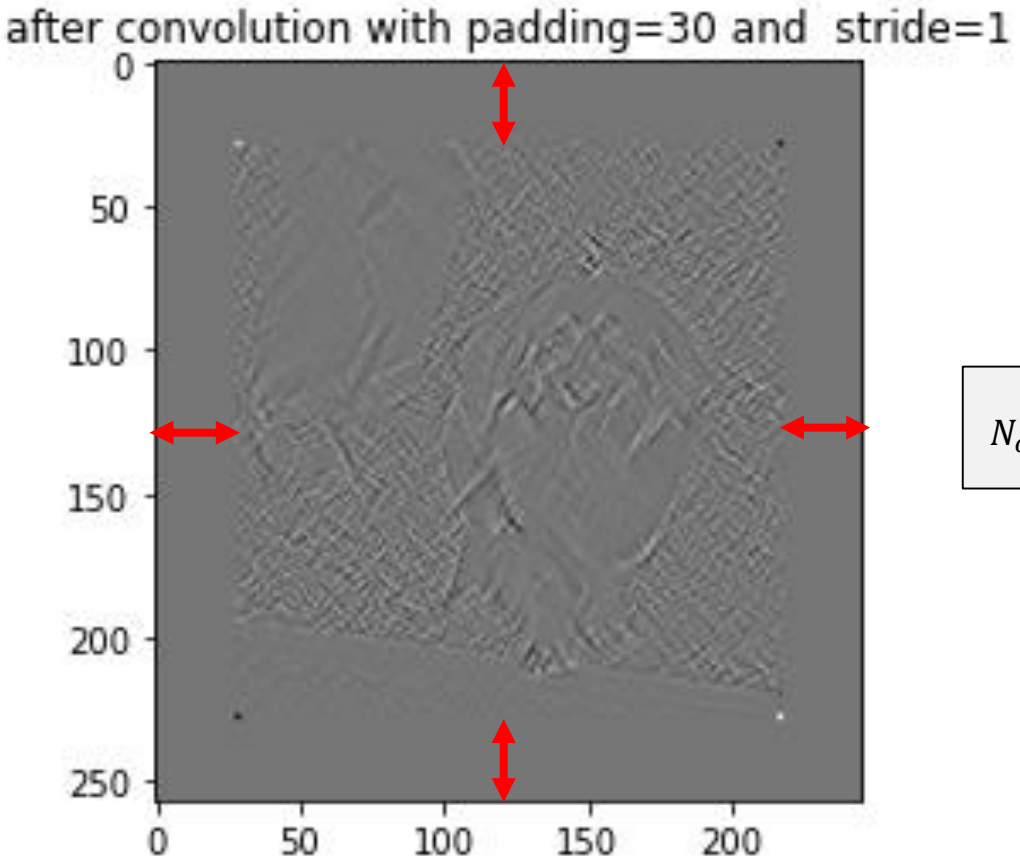
after convolution with padding=0 and stride=20



## II: Convolution with different filter

Let us now test the code with different settings for one **particular kernel**



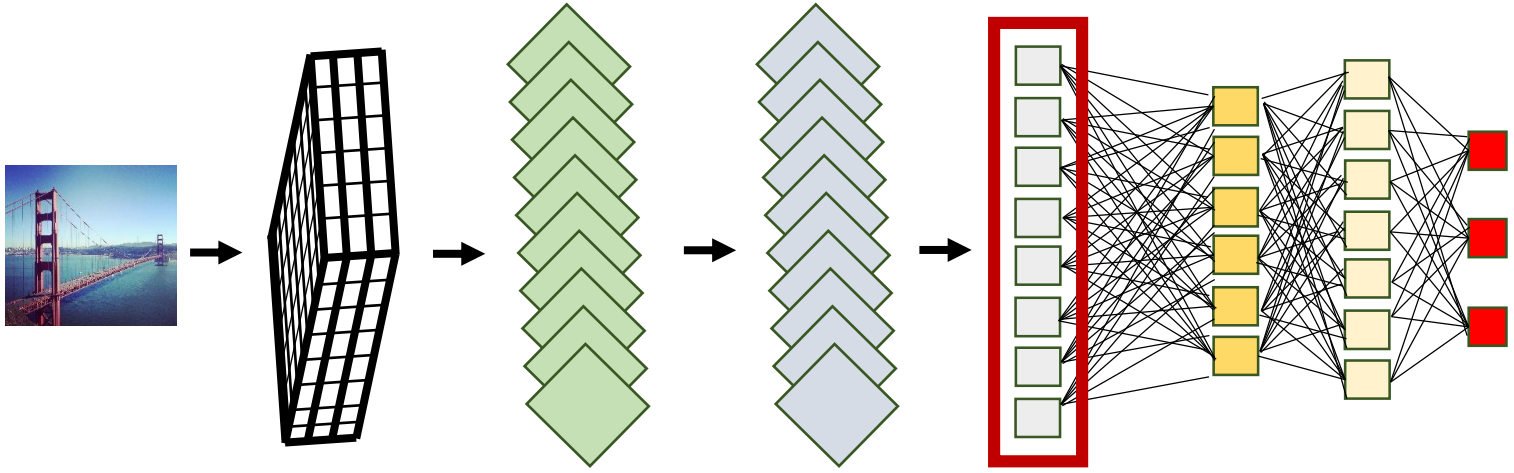

$$N_{out} = \frac{\left(N_{in} - N_{filt} + \mathbf{2 * padding}\right)}{stride\ length} + 1$$

## III: Loading a batch of images

We are almost ready for including the convolution part in our network!

An CNN needs **hundreds**, if not **thousands** of images as training sample
→ loading only **mini batches** of sizes 64, 128, 256… at once

Eventually, we have to **link our convolution part** somehow to the actual network…



…but the images have **different sizes** → **rescaling all images**

**We have to do these things first!**

## III: Loading a batch of images

For this, we need to load a few more libraries:

→ go to our current ANN and add the following lines to the header:

```
import glob as gl       # for ls like in linux

import random           # for picking mini batches randomly

from PIL import Image   # for resizing images
```

as usual, we want to set up a class

```
class Read_Scale_Imds:

    def Read_Scale(minibatch_size, ANN_size):
```

input:

1) `minibatch_size` **(number)**
→ how many images does the CNN run at once

2) `ANN_size` **(array of two numbers)**
→ pixel size to which all images have to be scaled

## III: Loading a batch of images

```
import glob as gl       # for ls like in linux
import random           # for picking mini batches randomly
from PIL import Image   # for resizing images

class Read_Scale_Imds:

    def Read_Scale(minibatch_size, ANN_size):
```

```
path_dogs = 'C:/my_path/Dog/*.jpg'
path_cats = 'C:/my_path/Cat/*.jpg'
```

setting paths to data

```
Cats = gl.glob(path_cats)
Dogs = gl.glob(path_dogs)
```

equivalent to *ls* in linux

```
Ld = len(Dogs)
Lc = len(Cats)
D  = np.zeros((Ld,1))
C  = np.zeros((Lc,1)) + 1
```

creating a list of all images and assigning classes to training data:
dogs = 0
cats = 1

```
All_class = np.array(np.vstack((D,C)))
All_imds  = np.hstack((Dogs,Cats))
```

putting both together

## III: Loading a batch of images

```
import glob as gl      # for ls like in linux
import random          # for picking mini batches randomly
from PIL import Image  # for resizing images

class Read_Scale_Imds:

        def Read_Scale(minibatch_size, ANN_size):
        ...
        All_class = np.array(np.vstack((D,C)))
        All_imds  = np.hstack((Dogs,Cats))

        idx = random.sample(range(Ld+Lc), minibatch_size)

        classes = All_class[idx]
        imds    = All_imds[idx]
```

randomly selecting a subsample as mini batch

## III: Loading a batch of images

```
import glob as gl      # for ls like in linux
import random          # for picking mini batches randomly
from PIL import Image  # for resizing images

class Read_Scale_Imds:

        def Read_Scale(minibatch_size, ANN_size):
        ...
        All_class = np.array(np.vstack((D,C)))
        All_imds  = np.hstack((Dogs,Cats))

        idx = random.sample(range(Ld+Lc), minibatch_size)

        classes = All_class[idx]
        imds    = All_imds[idx]
```

loading & resizing the images and storing them in one big array

```
#size ANN x size ANN x RGB x minibatch matrix
ImdsMatrix = np.zeros((ANN_size[0],ANN_size[1],3,minibatch_size))

for i in range(minibatch_size):

    I                    = Image.open(imds[i])
    Ire                  = I.resize((ANN_size[1],ANN_size[0]))
    Iar                  = np.array(Ire) # better be save
    ImdsMatrix[:,:,:,i] = Iar
```

## III: Loading a batch of images

```
import glob as gl      # for ls like in linux
import random          # for picking mini batches randomly
from PIL import Image  # for resizing images

class Read_Scale_Imds:

        def Read_Scale(minibatch_size, ANN_size):
        • • •
        All_class = np.array(np.vstack((D,C)))
        All_imds  = np.hstack((Dogs,Cats))

        idx = random.sample(range(Ld+Lc), minibatch_size)

        classes = All_class[idx]
        imds    = All_imds[idx]

        #size ANN x size ANN x RGB x minibatch matrix
        ImdsMatrix = np.zeros((ANN_size[0],ANN_size[1],3,minibatch_size))

        for i in range(minibatch_size):

            I                   = Image.open(imds[i])
            Ire                 = I.resize((ANN_size[1],ANN_size[0]))
            Iar                 = np.array(Ire) # better be save
            ImdsMatrix[:,:,:,i] = Iar


        ImdsMatrix.astype(np.uint8)   # again, making sure,
                                      # that we have the correct type


        return(ImdsMatrix,classes)
```

## III: Loading a batch of images

```
import glob as gl      # for ls like in linux
import random          # for picking mini batches randomly
from PIL import Image  # for resizing images

class Read_Scale_Imds:

    def Read_Scale(minibatch_size, ANN_size):

        path_dogs = 'C:/my_path/Dog/*.jpg'
        path_cats = 'C:/my_path/Cat/*.jpg'

        Cats = gl.glob(path_cats)
        Dogs = gl.glob(path_dogs)

        Ld = len(Dogs)
        Lc = len(Cats)
        D  = np.zeros((Ld,1))
        C  = np.zeros((Lc,1)) + 1

        All_class = np.array(np.vstack((D,C)))
        All_imds  = np.hstack((Dogs,Cats))

        idx = random.sample(range(Ld+Lc), minibatc

        classes = All_class[idx]
        imds    = All_imds[idx]

        #size ANN x size ANN x RGB x minibatch matrix
        ImdsMatrix = np.zeros((ANN_size[0],ANN_size[1],3,minibatch_size))

        for i in range(minibatch_size):

            I                  = Image.open(imds[i])
            Ire                = I.resize((ANN_size[1],ANN_size[0]))
            Iar                = np.array(Ire) # better be save
            ImdsMatrix[:,:,:,i] = Iar

        ImdsMatrix.astype(np.uint8)          # again, making sure,
                                             # that we have the correct type

        return(ImdsMatrix,classes)
```

We just need to add a tiny loop if some images are b/w

```
if len(Iar.shape) != 3:

        I3D = np.zeros((ANN_size[0],\
              ANN_size[1],3))
        I3D[:,:,0] = Iar
        I3D[:,:,1] = Iar
        I3D[:,:,2] = Iar
        Iar        = I3D
```

## III: Loading a batch of images

```
import glob as gl      # for ls like in linux
import random          # for picking mini batches randomly
from PIL import Image  # for resizing images

class Read_Scale_Imds:

        def Read_Scale(minibatch_size, ANN_size):
            ...
            All_class = np.array(np.vstack((D,C)))
            All_imds  = np.hstack((Dogs,Cats))

            idx = random.sample(range(Ld+Lc), minibatch_size)

            classes = All_class[idx]
            imds    = All_imds[idx]
            #size ANN x size ANN x RGB x minibatch matrix
            ImdsMatrix = np.zeros((ANN_size[0],ANN_size[1],3,minibatch_size))

            for i in range(minibatch_size):

                I                   = Image.open(imds[i])
                Ire                 = I.resize((ANN_size[1],ANN_size[0]))
                Iar                 = np.array(Ire) # better be save
                ImdsMatrix[:,:,:,i] = Iar

                if len(Iar.shape) != 3:

                        I3D = np.zeros((ANN_size[0], ANN_size[1],3))
                        I3D[:,:,0] = Iar
                        I3D[:,:,1] = Iar
                        I3D[:,:,2] = Iar
                        Iar        = I3D

            ImdsMatrix.astype(np.uint8)          # again, making sure,
                                                 # that we have the correct type
            return(ImdsMatrix,classes)
```

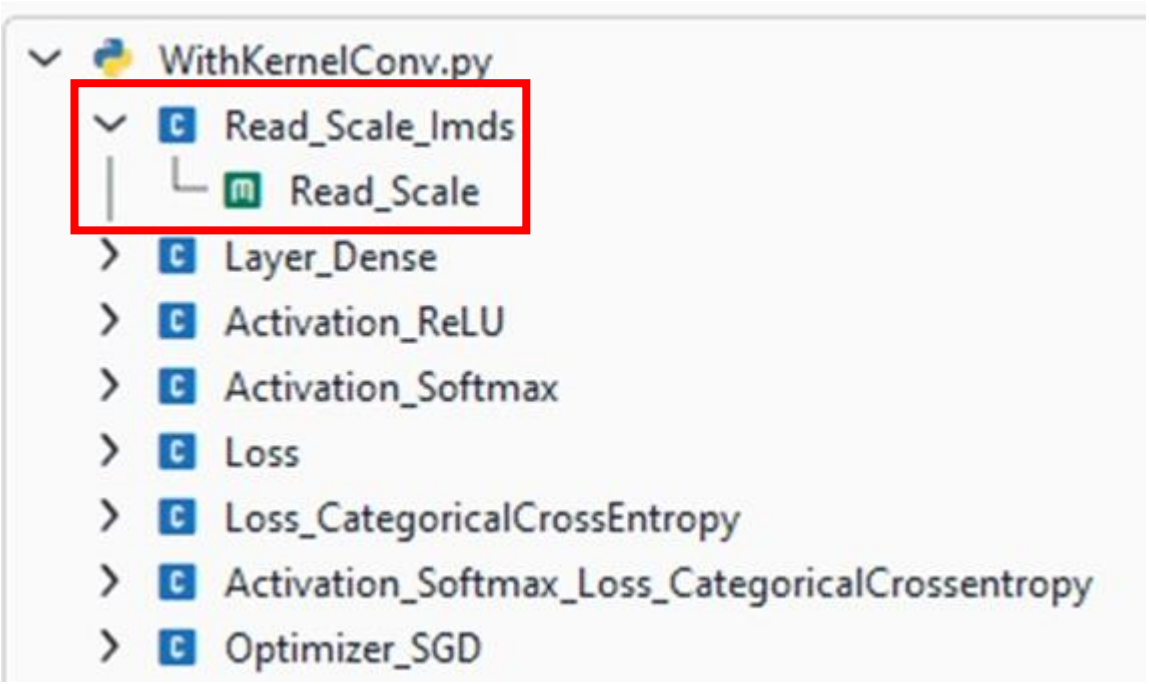
our complete code
→ check and test run the code

## III: Loading a batch of images

→ check and test run the code

in the console:

```
import WithKernelConv as MyANN
```




```
In [78]: MyANN.Read_Scale_Imds.Read_Scale(22,[128, 128])
```

IV: Including convolution part with kernels

→ go to the function `My_Convolution.py`

→ copy the part with the kernels, starting at

`K1 = np.array( ...`

until

`K = np.dstack( ...`

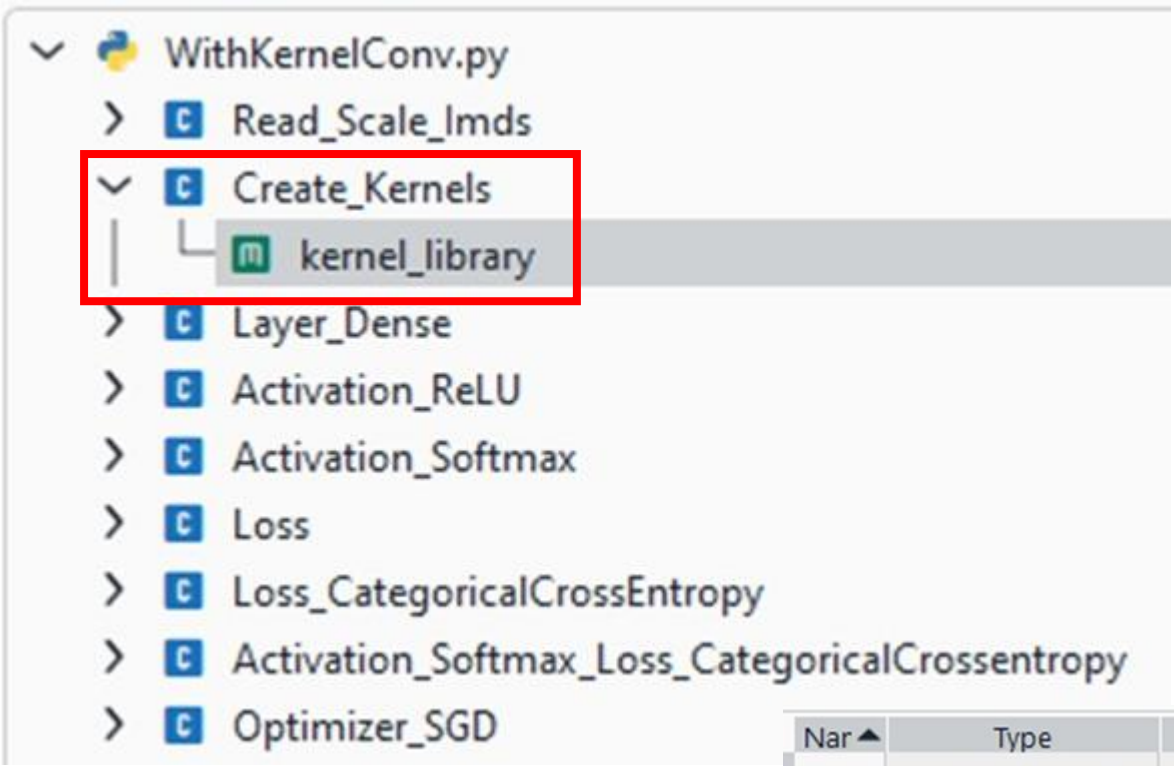
→ paste it into the ANN

```
class Create_Kernels:

    def kernel_library():


        #creating kernels
        K1  = np.array([[0, 0, 0], [0, 1, 0], [0, 0, 0]])

        #edges
        K2  = np.array([[1, 0, -1], [0, 0, 0], [-1, 0, 1]])

        ...

        K = np.dstack((K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13))


        return(K)
```

IV: Including convolution part with kernels

in the console:

```
import WithKernelConv as MyANN
```




| Nar | Type | Size | Value |
|---|---|---|---|
| Kout | Array of float64 | (3, 3, 13) | [[[ 0. 1. 0. ... 1. 0. 1.]<br>[ 0. 0. 1. ... 0. 0. 0.] |

```
Kout = MyANN.Create_Kernels.kernel_library()
```

## IV: Including convolution part with kernels

we now want to integrate the convolution part itself:

the `__init__` part calls the kernels

→ the entries of the kernels are the *weights* **(which will be learned)** of the convolution part

```
class ConvLayer:

    def __init__(self):
        #reading kernels from library
        K = Create_Kernels.kernel_library()

        #kernel sizes
        self.xKernShape = K.shape[0]
        self.yKernShape = K.shape[1]
        self.Kernnumber = K.shape[2]
```

## IV: Including convolution part with kernels

we now want to integrate the convolution part itself:

```
class ConvLayer:

    def __init__(self):
        #reading kernels from library
        K = Create_Kernels.kernel_library()

        #kernel sizes
        self.xKernShape = K.shape[0]
        self.yKernShape = K.shape[1]
        self.Kernnumber = K.shape[2]
```

```
self.weights    = K

self.biases     = np.zeros((1, self.Kernnumber))
```

## IV: Including convolution part with kernels

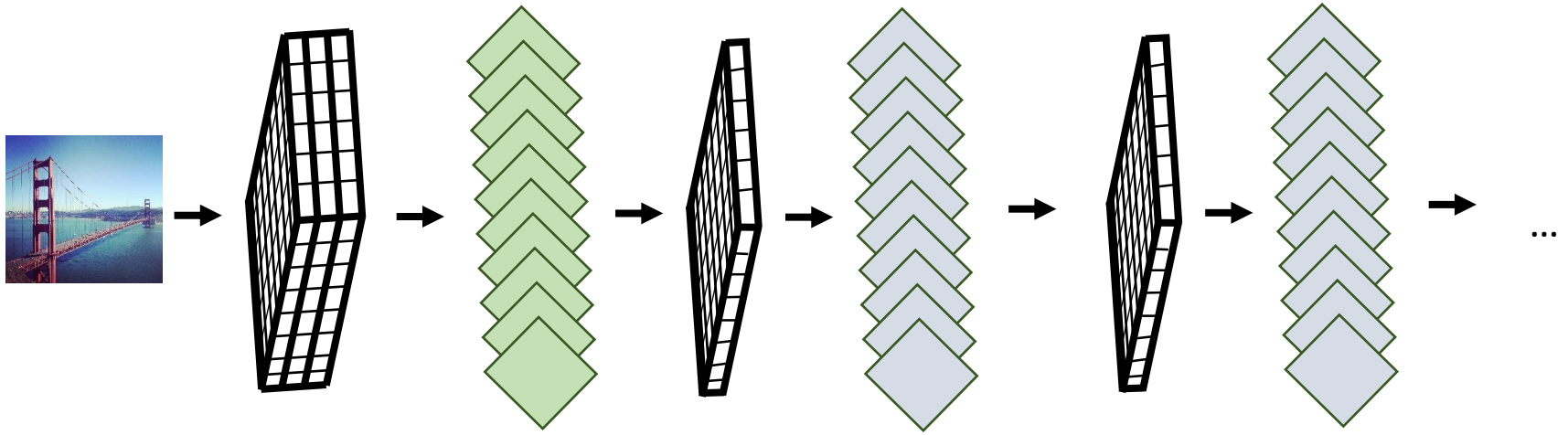
we now want to integrate the convolution part itself:

```
class ConvLayer:

    def __init__(self):
        #reading kernels from library
        K = Create_Kernels.kernel_library()

        #kernel sizes
        self.xKernShape = K.shape[0]
        self.yKernShape = K.shape[1]
        self.Kernnumber = K.shape[2]

        self.weights    = K

        self.biases     = np.zeros((1, self.Kernnumber))
```

```
def forward(self, M, padding = 0, stride = 1):
```

## IV: Including convolution part with kernels

typically, there are many convolution layers in a CNN




3D RGB image

2D gray scale image

2D gray scale image

M: `x, y, numChans, numImds`

M: `x, y, ~~numChans~~, numImds` NK

in our code: `numChans` refers to incoming matrix, `Kernnumber` to output matrix

## IV: Including convolution part with kernels

we now want to integrate the convolution part itself:

```
class ConvLayer:
...
```

```
def forward(self, M, padding = 0, stride = 1):
```

M is the matrix containing the images, according to `Read_Scale_Imds`

```
xImgShape   = M.shape[0]
yImgShape   = M.shape[1]
numChans    = M.shape[2]
numImds     = M.shape[3]
```

numChans refers to incoming matrix, Kernnumber to output matrix

```
xK          = self.xKernShape
yK          = self.yKernShape
NK          = self.Kernnumber

b           = self.biases
```

## IV: Including convolution part with kernels

we now want to integrate the convolution part itself:

```
class ConvLayer:
...
```

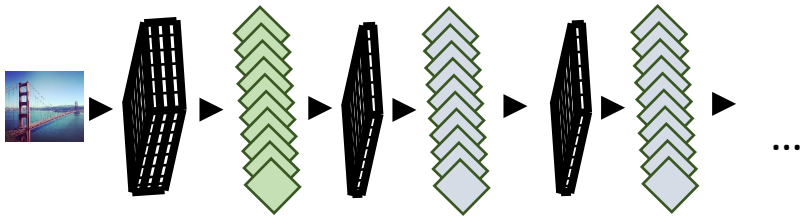
```
def forward(self, M, padding = 0, stride = 1):

    xImgShape  = M.shape[0]
    yImgShape  = M.shape[1]
    numChans   = M.shape[2]
    numImds    = M.shape[3]

    xK         = self.xKernShape
    yK         = self.yKernShape
    NK         = self.Kernnumber

    b          = self.biases

#storage for backprop, see later
self.padding = padding
self.stride  = stride
```

recall: $N_{out} = \frac{(N_{in} - N_{filt} + 2 * padding)}{stride\ length} + 1$

**copy/paste this part from ConvSelfMade.py**

```
xOutput = int(((xImgShape - xK + 2 * padding)/stride) + 1)
yOutput = int(((yImgShape - yK + 2 * padding)/stride) + 1)
```

## IV: Including convolution part with kernels



→ *in our code: numChans refers to incoming matrix, Kernnumber to output matrix*

→ go to the function **ConvSelfMade.py**

→ copy the convolution part, starting at

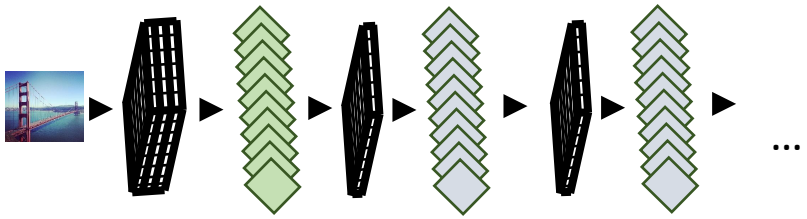
```
output = np.zeros( ...
```

until

```
output[x,y,c] = np.sum(s)
```

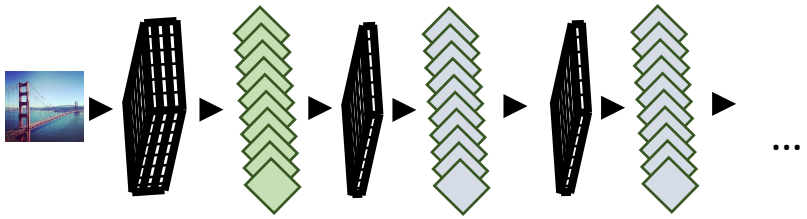
→ paste it into the ANN

## IV: Including convolution part with kernels



```
W = np.nan_to_num(self.weights)

output = np.zeros((xOutput,yOutput,numChans , NK, numImds))


imagePadded = np.zeros(((xImgShape + padding*2).\
                (yImgShape + padding*2),numChans , NK, numImds))


 for k in range(NK):
        imagePadded[int(padding):int(padding + xImgShape),\
        int(padding):int(padding + yImgShape),:,k,:]= M
```

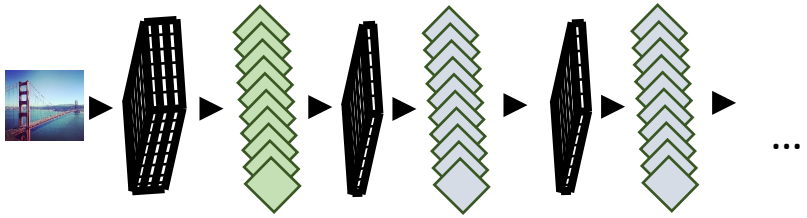
## IV: Including convolution part with kernels



...

```
for i in range(numImds):# loop over number of images
    currentIm_pad = imagePadded[:,:,:,:,i]# select ith padded image
    for y in range(yOutput):# loop over vert axis of output
        for x in range(yOutput):# loop over hor axis of output
            for c in range(numChans):# loop over channels
                for k in range(NK): # loop over filter
                    y_start = y*stride
                    y_end   = y*stride + yK
                    x_start = x*stride
                    x_end   = x*stride + xK
                    #selecting the current part of the image
                    current_slice = currentIm_pad[x_start:x_end,\
                                                  y_start:y_end,c,k]

                    #the actual conv part
                    s             = np.multiply(current_slice, K)
                    output[x,y,c] = np.sum(s)
```

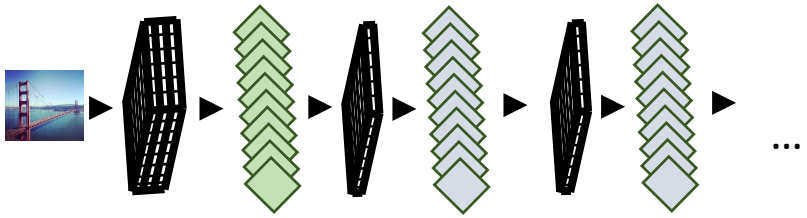
## IV: Including convolution part with kernels



...

```
for i in range(numImds):# loop over number of images
    currentIm_pad = imagePadded[:,:,:,:,i]# select ith padded image
    for y in range(yOutput):# loop over vert axis of output
        for x in range(yOutput):# loop over hor axis of output
            for c in range(numChans):# loop over channels
                for k in range(NK): # loop over filter

                        y_start = y*stride
                        y_end   = y*stride + yK
                        x_start = x*stride
                        x_end   = x*stride + xK

                        #selecting the current part of the image
                        current_slice = currentIm_pad[x_start:x_end,\
                                                      y_start:y_end,c,k]

                s                 = np.multiply(current_slice, W[:,:,k])

                output[x, y, c, k, i] = np.sum(s) + b[0,k].astype(float)

  #results are usually summed over channels...
  output = output.sum(2)
```

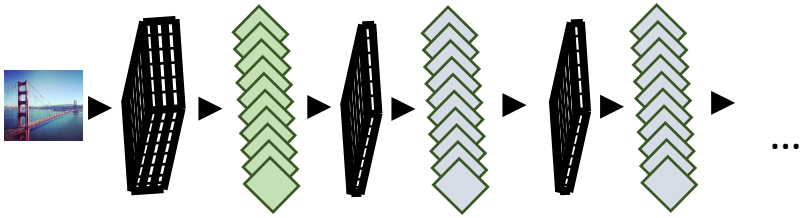
## IV: Including convolution part with kernels



...

```
...

for i in range(numImds):# loop over number of images
    currentIm_pad = imagePadded[:,:,:,:,i]# select ith padded image
    for y in range(yOutput):# loop over vert axis of output
        for x in range(yOutput):# loop over hor axis of output
            for c in range(numChans):# loop over channels
                for k in range(NK): # loop over filter

                        y_start = y*stride
                        y_end   = y*stride + yK
                        x_start = x*stride
                        x_end   = x*stride + xK

                        #selecting the current part of the image
                        current_slice      = currentIm_pad[x_start:x_end,\
                                                           y_start:y_end,c,k]
                        s                  = np.multiply(current_slice, W[:,:,k])

                        output[x, y, c, k, i] = np.sum(s) + b[0,k].astype(float)


 #results are usually summed over channels...
 output = output.sum(2)
```

## IV: Including convolution part with kernels

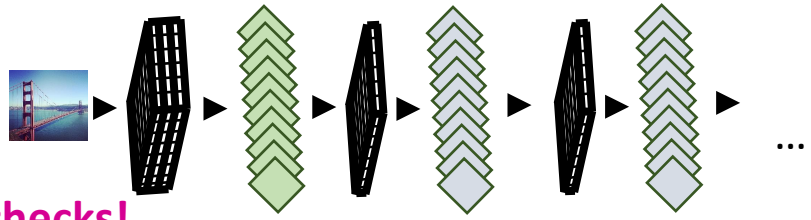


finally, we save the output and M

```
self.output = np.nan_to_num(output)
self.input  = M
self.impad  = imagePadded
```

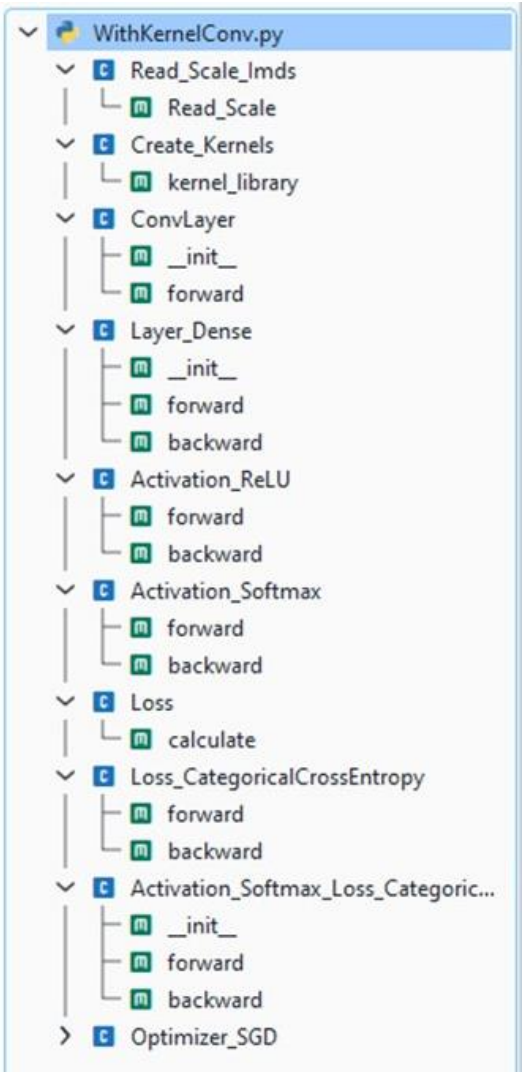
**Let us run the code now and perform some sanity checks!**

## IV: Including convolution part with kernels

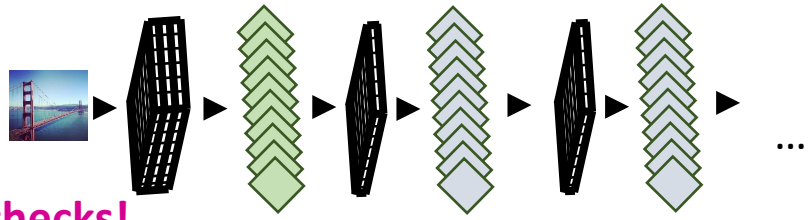


**Let us run the code now and perform some sanity checks!**

check the network structure →



## IV: Including convolution part with kernels



**Let us run the code now and perform some sanity checks!**

```
import WithKernelConv as My_ANN
import matplotlib.pyplot as plt

minibatch_size = 10
ANN_size       = [128,128]
```

calling 10 images randomly and scaling them to part size **(here 128x128)** = number of input neurons

→ returning batch of scaled images and classes C

```
[M, C] = My_ANN.Read_Scale_Imds.Read_Scale(minibatch_size, ANN_size)
```

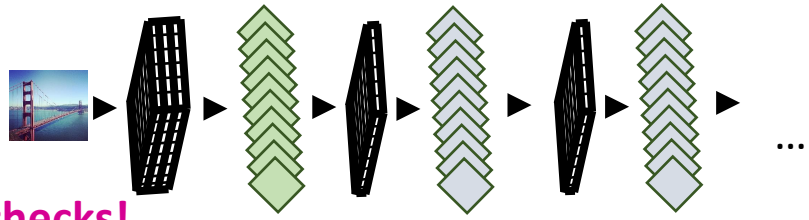
```
Conv1 = My_ANN.ConvLayer()
Conv2 = My_ANN.ConvLayer()
Conv3 = My_ANN.ConvLayer()
```

initializing convolution layers

```
Conv1.forward(M,2,1)
Conv2.forward(Conv1.output)
Conv3.forward(Conv2.output,0,3)
```

run convolution on all images

## IV: Including convolution part with kernels

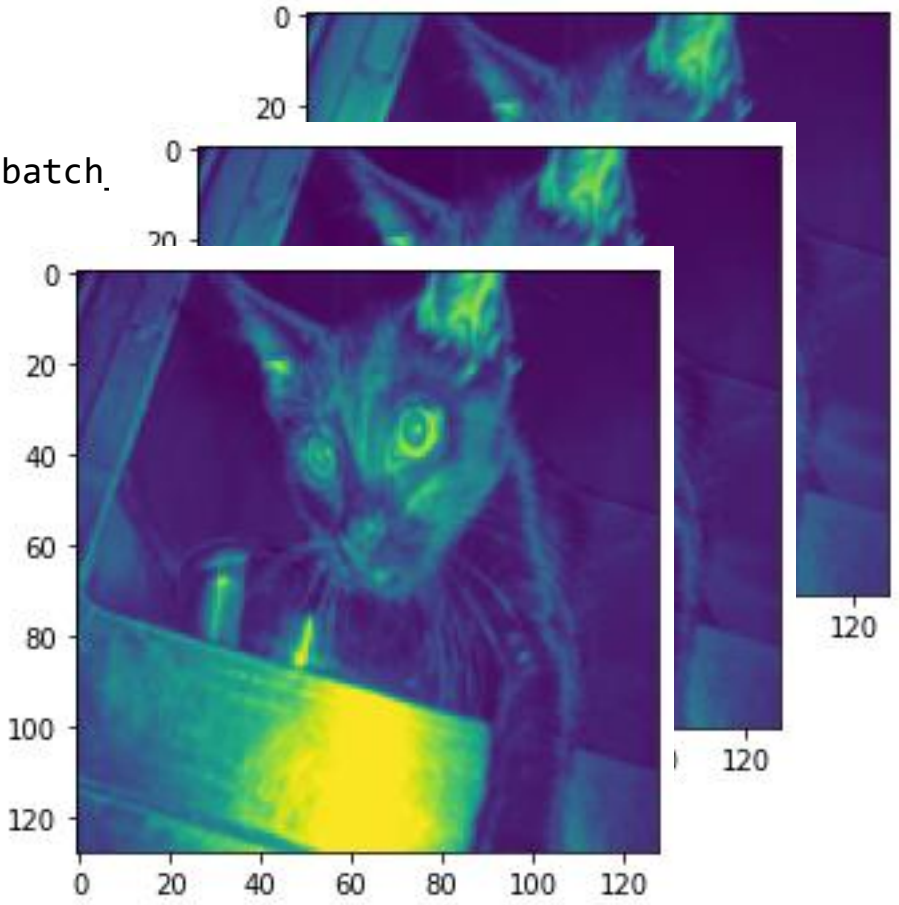


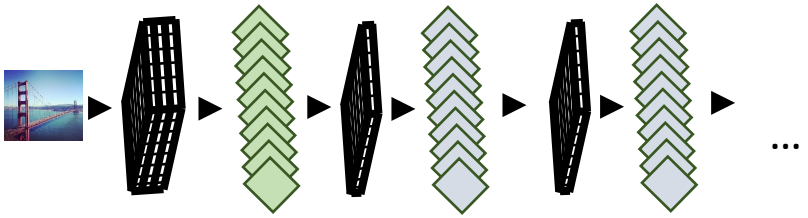
**Let us run the code now and perform some sanity checks!**

```
import WithKernelConv as My_ANN
import matplotlib.pyplot as plt
minibatch_size = 10
ANN_size       = [128,128]

[M, C] = My_ANN.Read_Scale_Imds.Read_Scale(minibatch

Conv1 = My_ANN.ConvLayer()
Conv2 = My_ANN.ConvLayer()
Conv3 = My_ANN.ConvLayer()
Conv1.forward(M,2,1)
Conv2.forward(Conv1.output)
Conv3.forward(Conv2.output,0,3)
```

original image (RGB)

```
plt.imshow(M[:,:,0,1])
plt.show()
plt.imshow(M[:,:,1,1])
plt.show()
plt.imshow(M[:,:,2,1])
plt.show()
```



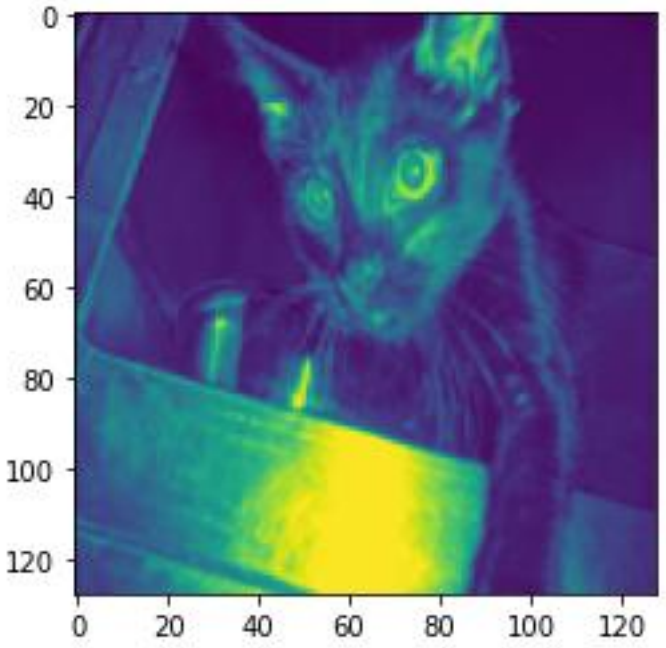


## IV: Including convolution part with kernels

```
import WithKernelConv as My_ANN
import matplotlib.pyplot as plt
minibatch_size = 10
ANN_size       = [128,128]

[M, C] = My_ANN.Read_Scale_Imds.Read_Scale(minibatch_size, ANN_size)

Conv1 = My_ANN.ConvLayer()
Conv2 = My_ANN.ConvLayer()
Conv3 = My_ANN.ConvLayer()

Conv1.forward(M,2,1)
Conv2.forward(Conv1.output)
Conv3.forward(Conv2.output,0,3)

plt.imshow(M[:,:,0,1])
plt.show()
plt.imshow(M[:,:,1,1])
plt.show()
plt.imshow(M[:,:,2,1])
plt.show()
```

**RGB**

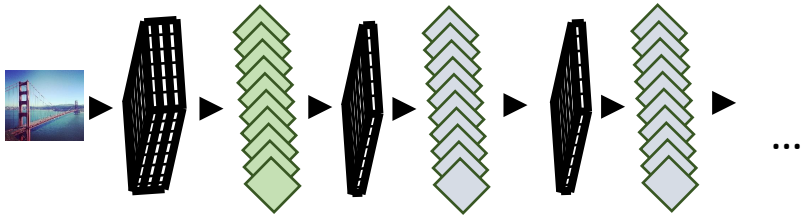
```
In [1]: M.shape
Out[1]: (128, 128, 3, 10)
```

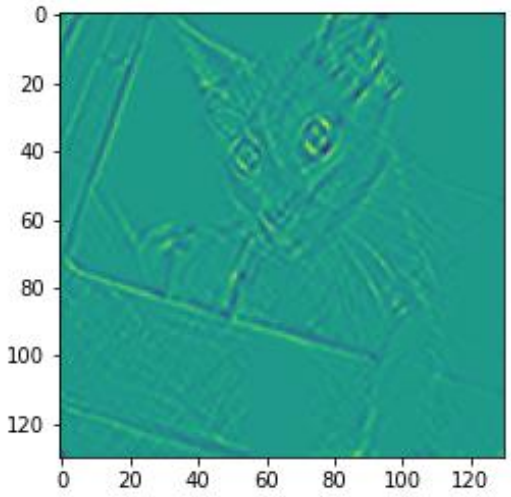


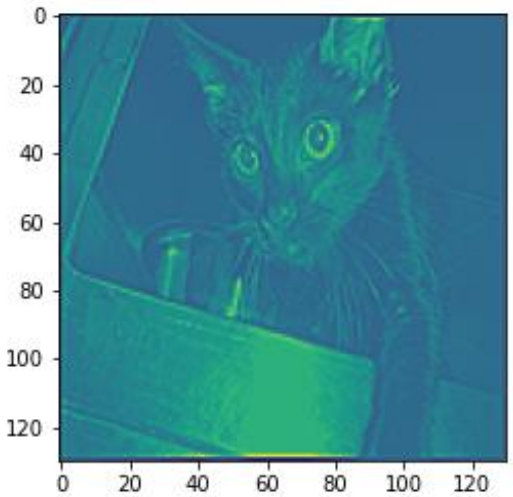

original image (RGB)

## IV: Including convolution part with kernels

```
Conv1.forward(M,2,1)
Conv2.forward(Conv1.output)
Conv3.forward(Conv2.output,0,3)

plt.imshow(M[:,:,0,1])
plt.show()
plt.imshow(M[:,:,1,1])
plt.show()
plt.imshow(M[:,:,2,1])
plt.show()
```






after 1st convolution (filter 2 and 5)

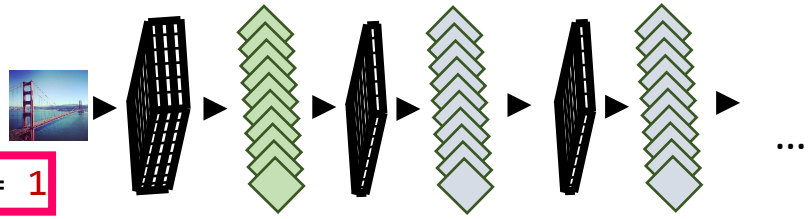
```
In [1]: M.shape
Out[1]: (128, 128, 3, 10)
```

```
plt.imshow(Conv1.output[:,:,1,1])
plt.show()
plt.imshow(Conv1.output[:,:,4,1])
plt.show()
```

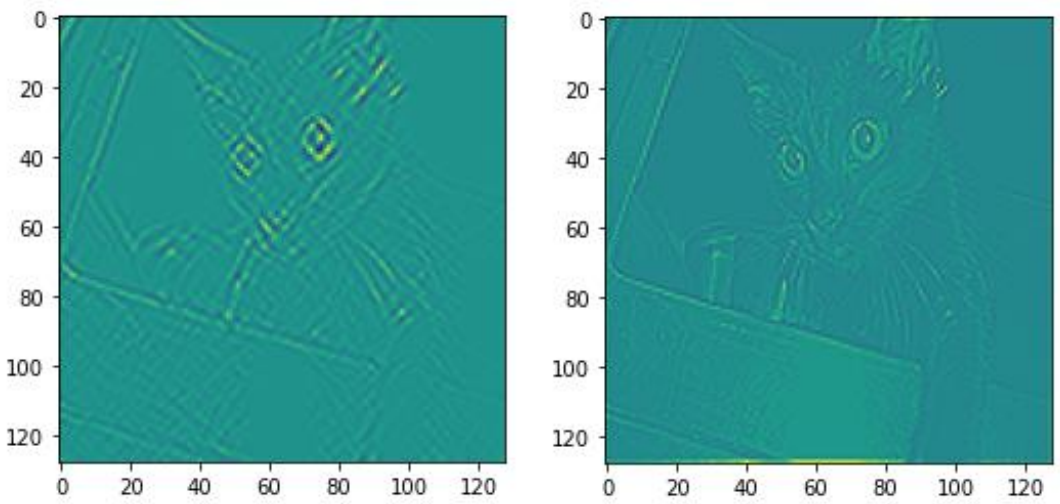
```
In [2]: Conv1.output.shape
Out[2]: (130, 130, 13, 10)
```

recall: $N_{out} = \frac{(N_{in} - N_{filt} + 2 * padding)}{stride\ length} + 1$

## IV: Including convolution part with kernels



```
Conv1.forward(M,2,1)
Conv2.forward(Conv1.output) padding = 0, stride = 1
Conv3.forward(Conv2.output,0,3)

plt.imshow(Conv1.output[:,:,1,1])
plt.show()
plt.imshow(Conv1.output[:,:,4,1])
plt.show()

In [2]: Conv1.output.shape
Out[2]: (130, 130, 13, 10)
```



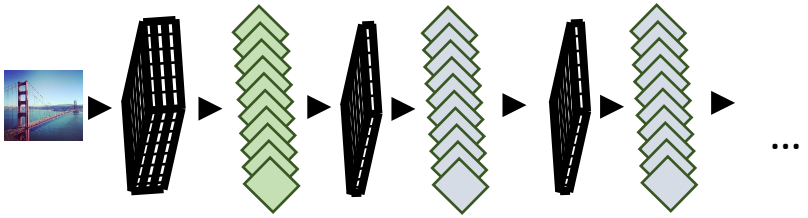
after 2nd convolution (filter 2 and 5)

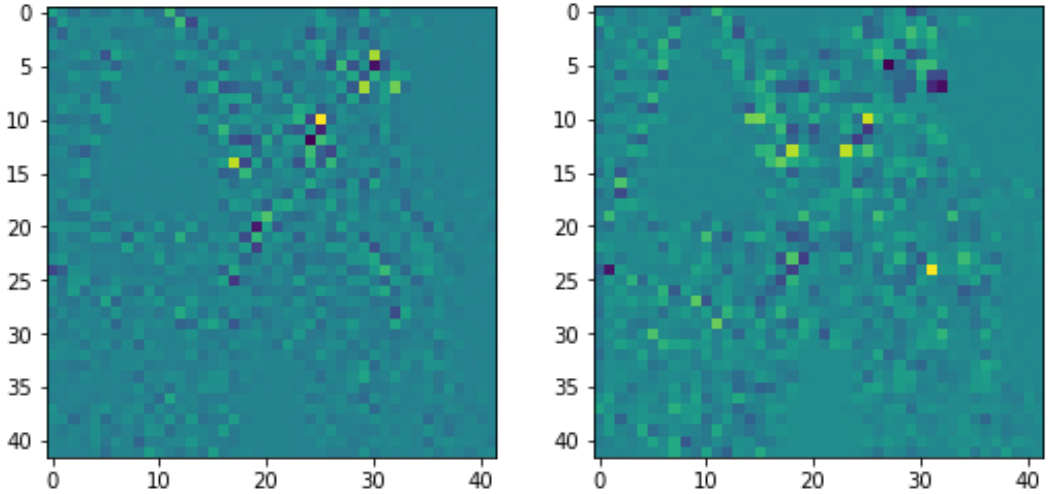
```
plt.imshow(Conv2.output[:,:,1,1])
plt.show()
plt.imshow(Conv2.output[:,:,4,1])
plt.show()

In [3]: Conv2.output.shape
Out[4]: (128, 128, 13, 10)
```

recall: $N_{out} = \frac{(N_{in} - N_{filt} + 2*padding)}{stride\ length} + 1$

## IV: Including convolution part with kernels

```
Conv1.forward(M,2,1)
Conv2.forward(Conv1.output)
Conv3.forward(Conv2.output,0,3)

plt.imshow(Conv2.output[:,:,1,1])
plt.show()
plt.imshow(Conv2.output[:,:,4,1])
plt.show()

In [3]: Conv2.output.shape
Out[4]: (128, 128, 13, 10)
```





after 3rd convolution (filter 2 and 5)

```
plt.imshow(Conv3.output[:,:,1,1])
plt.show()
plt.imshow(Conv3.output[:,:,4,1])
plt.show()

In [3]: Conv3.output.shape
Out[4]: (42, 42, 13, 10)
```

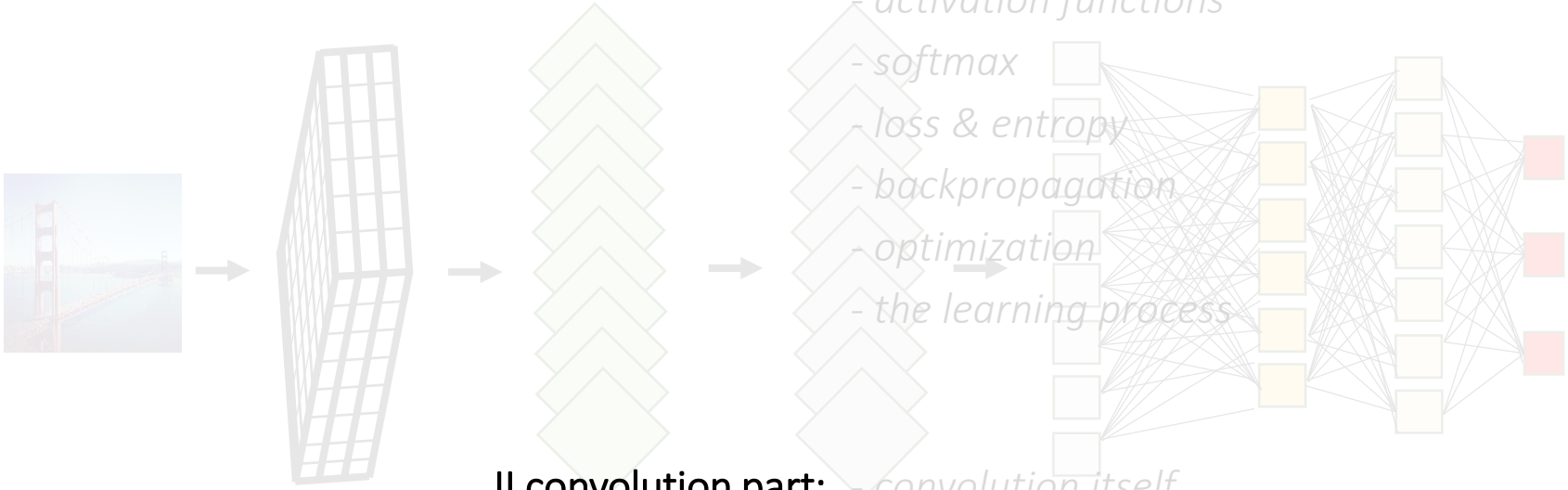
recall: $N_{out} = \frac{(N_{in} - N_{filt} + 2*padding)}{stride\ length} + 1$

## outline

0 intro

I the core:

- *a single neuron*
- *layers of neurons*
- *activation functions*
- *softmax*
- *loss & entropy*
- *backpropagation*
- *optimization*
- *the learning process*



II convolution part:

- *convolution itself*
- *pooling*
- *sigmoid*
- *flattening*
- *backpropagation - again*
- *testing the CNN & final remarks*

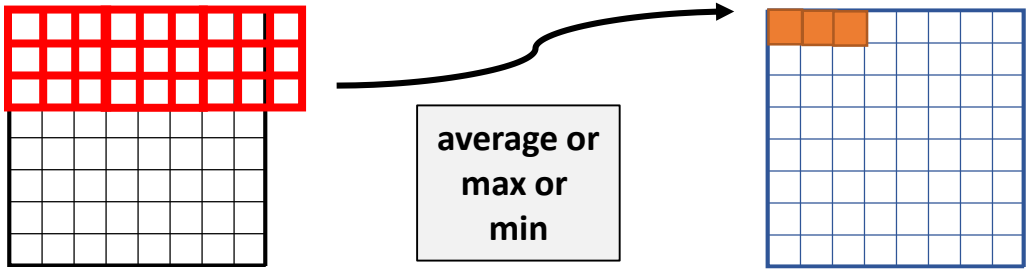
- often, a **pooling layer** is set after each convolution layer
  **(reduces variance and computation complexity)**

- there are three different main pooling methods

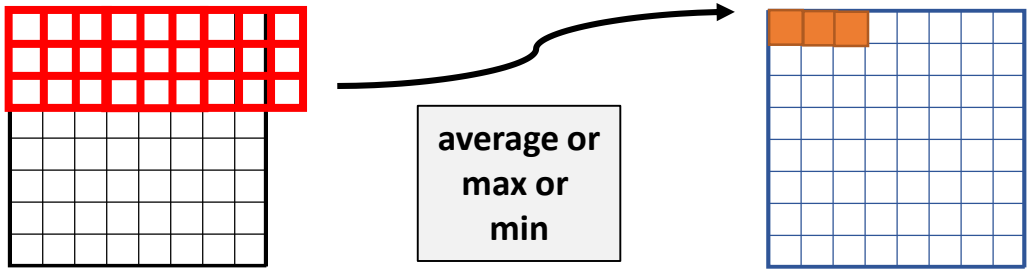
→ **average pool:** **blurs the image**, reduces edges (not what we want here)

→ **max pool:** **reduces dark background** (those pixel values are usually low) and **enhances brighter foreground objects** (exactly what we need here)

→ **min pool:** does the opposite of max pool



- creating a max pool layer: essentially the same as a **convolution layer**
  → just take the max instead of multiplication




```
class Max_Pool:

    def forward(self, M, stride = 1, KernShape = 2):

        xImgShape   = M.shape[0]
        yImgShape   = M.shape[1]
        numChans    = M.shape[2]
        numImds     = M.shape[3]

        self.inputs = M
```

padding doesn't make sense since we want to get the max of a patch

the max pool layer comes after a convolution layer

→ numChans referrers to number of kernel

```
class Max_Pool:

    def forward(self, M, stride = 1, KernShape = 2):

            xImgShape   = M.shape[0]
            yImgShape   = M.shape[1]
            numChans    = M.shape[2]
            numImds     = M.shape[3]
            self.inputs = M


#maxpool usually over nxn matrix
xK = KernShape
yK = KernShape


xOutput = int(((xImgShape - xK +                ) / stride) + 1)
yOutput = int(((yImgShape - yK +                ) / stride) + 1)
```

we can copy the convolution part and paste it into the forward method

```
class Max_Pool:

    def forward(self, M, stride = 1, KernShape = 2):

            ...

            #maxpool usually over nxn matrix
            xK = KernShape
            yK = KernShape

            xOutput = int(((xImgShape - xK) / stride) + 1)
            yOutput = int(((yImgShape - yK) / stride) + 1)
```

```
class Max_Pool:

    def forward(self, M, stride = 1, KernShape = 2):

            ...

imagePadded = M
```

we keep the name so that we don't need to change anything in the subsequent loop

```
#output matrix after max pool
output = np.zeros((xOutput,yOutput,numChans,numImds))
```

creating mask for storing location of maximum → needed for backpropagation

```
imagePadded_copy = imagePadded.copy() *0
```

```
class Max_Pool:

    def forward(self, M, stride = 1, KernShape = 2):

            ...
        imagePadded = M

        output            = np.zeros((xOutput,yOutput,numChans,numImds))
        imagePadded_copy = imagePadded.copy() *0


    for i in range(numImds):# loop over number of images
        currentIm_pad = imagePadded[:,:,:,i]# select ith padded image
        for y in range(yOutput):# loop over vert axis of output
              for x in range(xOutput):# loop over hor axis of output
                    for c in range(numChans):# loop over channels

                          y_start = y*stride
                          y_end   = y*stride + yK
                          x_start = x*stride
                          x_end   = x*stride + xK
```

again, we can copy the convolution part and modify it slightly

```
class Max_Pool:

    def forward(self, M, stride = 1, KernShape = 2):

            ...
            y_start = y*stride
            y_end   = y*stride + yK
            x_start = x*stride
            x_end   = x*stride + xK

sx      = slice(x_start,x_end)
sy      = slice(y_start,y_end)

current_slice = currentIm_pad[sx,sy,c]

slice_max           = float(current_slice.max())
output[x, y, c, i] = slice_max

imagePadded_copy[sx,sy,c,i] += np.equal(\
                currentIm_pad[sx,sy,c],slice_max).astype(float)
```

we are going to need the slicing vectors a few more times and therefore save them as new variable

actual max pool

```
class Max_Pool:

    def forward(self, M, stride = 1, KernShape = 2):

            ...
slice_max           = float(current_slice.max())
output[x, y, c, i] = slice_max

imagePadded_copy[sx,sy,c,i] += np.equal(\
                currentIm_pad[sx,sy,c],slice_max).astype(float)
```

avoids deletion of ones if `stride < KernShape` → has to be normalized to ones and zeros afterwards

compares entries in a matrix to a value → returns matrix with ones (where statement is true) and zeros otherwise

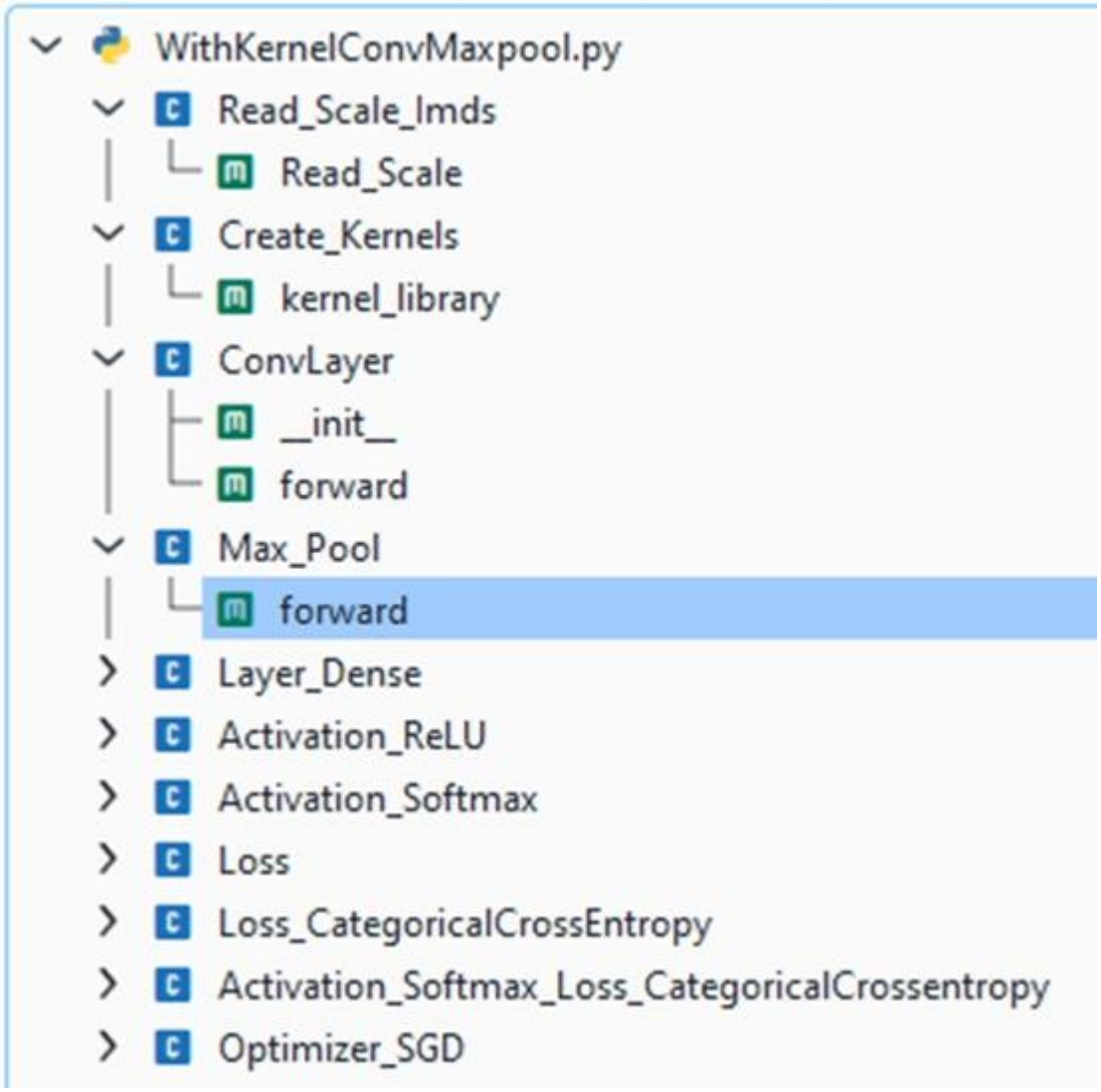
```
class Max_Pool:

    def forward(self, M, stride = 1, KernShape = 2):

            ...
slice_max          = float(current_slice.max())
output[x, y, c, i] = slice_max

imagePadded_copy[sx,sy,c,i] += np.equal(\
                  currentIm_pad[sx,sy,c],slice_max).astype(float)


mask = imagePadded_copy

mask = np.matrix.round(mask/(mask + 1e-7))
```

has to be normalized to ones and zeros afterwards

```
self.xKernShape = xK
self.yKernShape = yK
self.output     = output
self.mask       = mask
self.stride     = stride
```

saving some outputs for backpropagation later and sanity checks

testing the new layer:



testing the new layer:

```
import WithKernelConvMaxpool as My_ANN
import matplotlib.pyplot as plt

minibatch_size = 10
ANN_size       = [128,128]

[M, C] = My_ANN.Read_Scale_Imds.Read_Scale(minibatch_size, ANN_size)


Conv1 = My_ANN.ConvLayer()
MP1   = My_ANN.Max_Pool()


Conv1.forward(M,2,1)
MP1.forward(Conv1.output,2,3)


MP1in  = MP1.inputs
MP1out = MP1.output
Mask   = MP1.mask
```

loading the network, reading the images

initializing layers

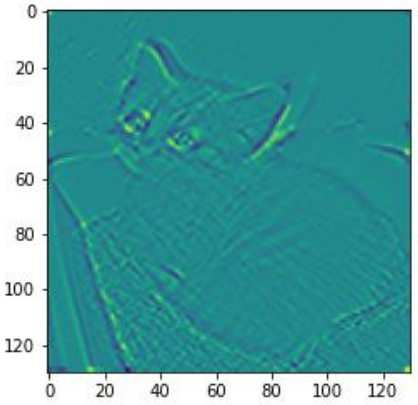
running layers

we want to compare the original image (after convolution) to the max pooled image and the mask

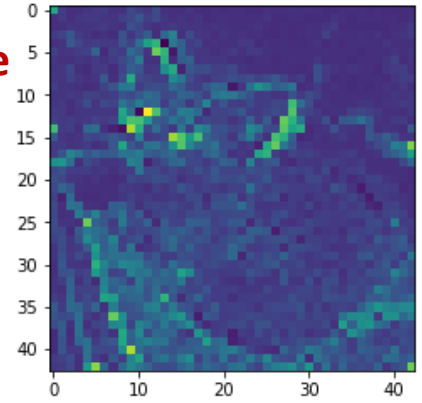
testing the new layer:

```
plt.imshow(MP1in[:,:,1,1])
plt.show()
plt.imshow(MP1out[:,:,1,1])
plt.show()
plt.imshow(Mask[:,:,1,1])
plt.show()
```

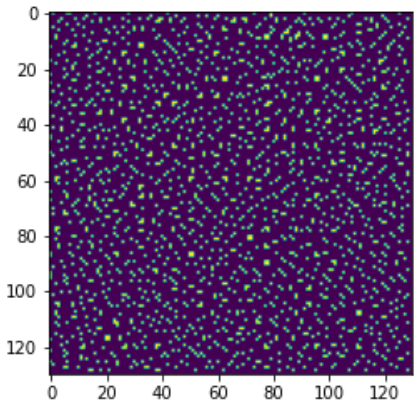
**original image**




**max pool image**




**mask**



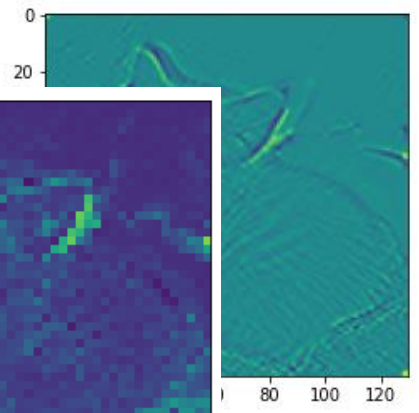

```
I1 = MP1in[:,:,1,1]
I2 = MP1out[:,:,1,1]
I3 = Mask[:,:,1,1]
```

testing the new layer:

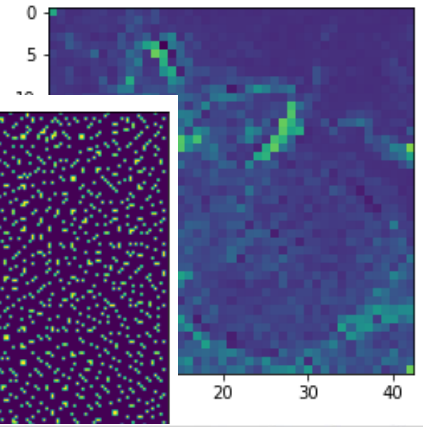
```
plt.imshow(MP1in[:,:,1,1])
plt.show()
plt.imshow(MP1out[:,:,1,1])
plt.show()
plt.imshow(Mask[:,:,1,1])
plt.show()
```

```
I1 = MP1in[:,:,1,1]
I2 = MP1out[:,:,1,1]
I3 = Mask[:,:,1,1]
```

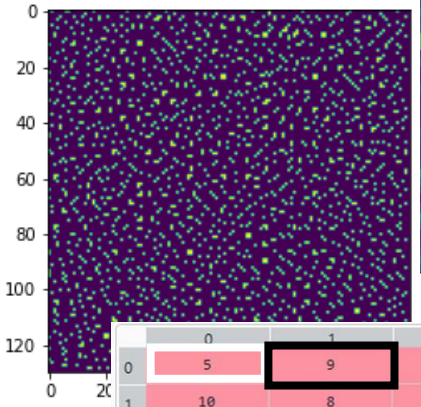
**original image**



**max pool image**



**mask**



I2

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 5 | 9 | 2 | 6 | 3 | 3 |
| 1 | 10 | 8 | 2 | 3 | 54 | 54 |
| 2 | 20 | 8 | 12 | 12 | 133 | 133 |
| 3 | 95 | 12 | 12 | 12 | 123 | 123 |
| 4 | 191 | 1 | -1 | 7 | 37 | 56 |
| 5 | 199 | 312 | 219 | 23 | 23 | 176 |
| 6 | 122 | 312 | 298 | 210 | 23 | 372 |
| 7 | 48 | 48 | 302 | 347 | 313 | 313 |
| 8 | 48 | 48 | 213 | 360 | 360 | 125 |

I1 stride = 2

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 5 | 0 | -2 | 9 | 2 | -6 |
| 1 | 2 | 0 | 1 | 9 | 2 | -6 |
| 2 | -4 | 1 | 5 | -2 | 2 | 0 |
| 3 | 0 | 2 | 5 | -3 | -2 | 0 |
| 4 | 3 | 10 | 8 | -8 | -11 | 0 |
| 5 | 9 | 20 | 6 | -17 | -18 | 0 |
| 6 | 15 | 20 | -2 | -18 | -17 | -2 |
| 7 | 12 | 4 | -17 | -9 | 12 | 3 |
| 8 | 95 | 20 | -82 | -12 | -14 | -9 |

I3

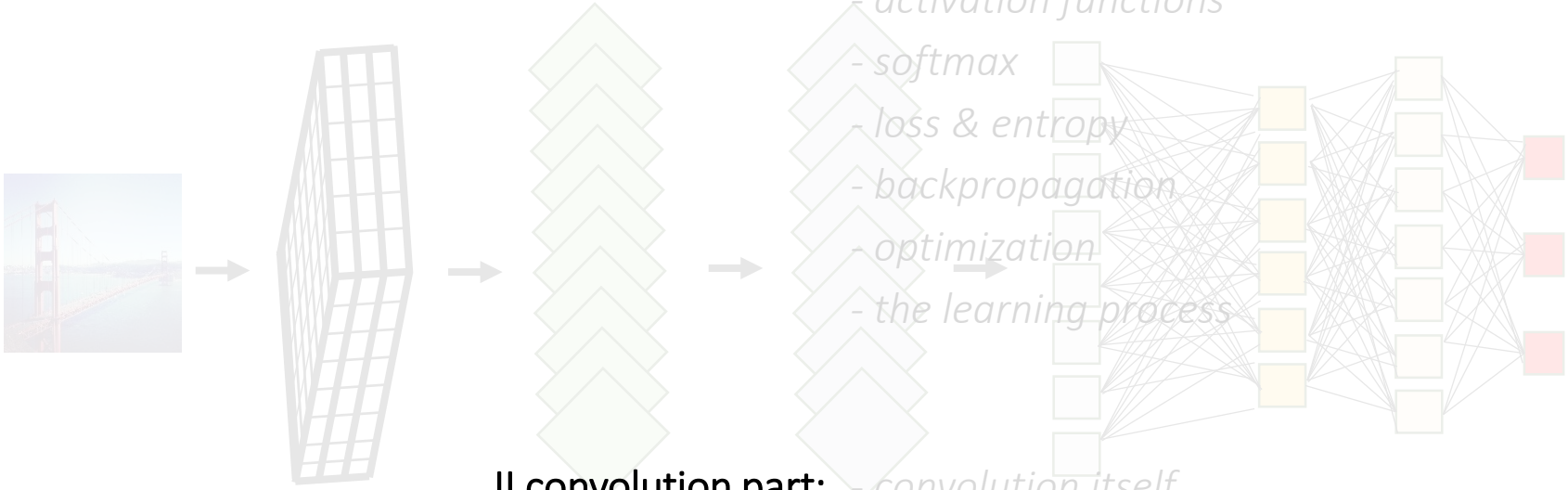
| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 2 | 0 | 0 | 1 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 1 | 1 | 0 | 0 | 0 |
| 5 | 0 | 1 | 0 | 0 | 0 | 0 |
| 6 | 0 | 1 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 1 | 0 |
| 8 | 1 | 0 | 0 | 0 | 0 | 0 |

outline

0 intro

I the core:

- *a single neuron*
- *layers of neurons*
- *activation functions*
- *softmax*
- *loss & entropy*
- *backpropagation*
- *optimization*
- *the learning process*



II convolution part:

- *convolution itself*
- *pooling*
- *sigmoid*
- *flattening*
- *backpropagation - again*
- *testing the CNN & final remarks*

CNNs have many convolution layers: → lots of multiplying/adding

subsequent max pool layers: → again, lots of increasing values

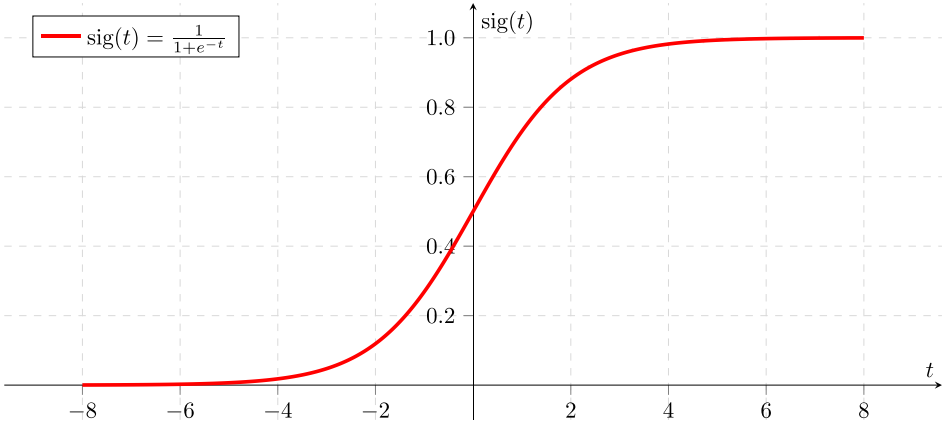
<u>problem:</u> values might increase and get out of control

<u>solution:</u> mapping small values to themselves, but large values to a limit

e.g.: $[-\infty; +\infty]$ → $[-1; +1]$ or $[0; +1]$

called *"compactification"*

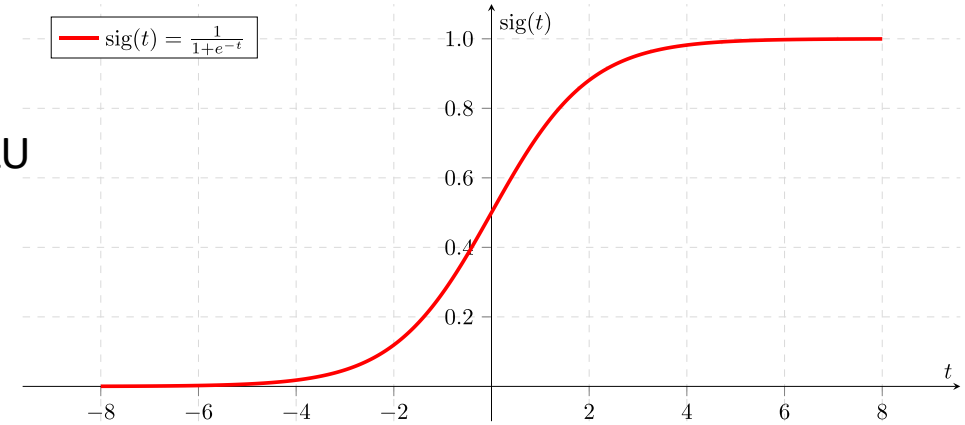
- sigmoid
- atan




moreover, ReLUs tend to overfitting because of their fast convergence

But sigmoids have disadvantages too:

- 2x more computational time than aReLU
- saturate easily, so that derivatives are zero → bad for backpropagation




At the end…depends on the ANN and a bit of **trial and error**

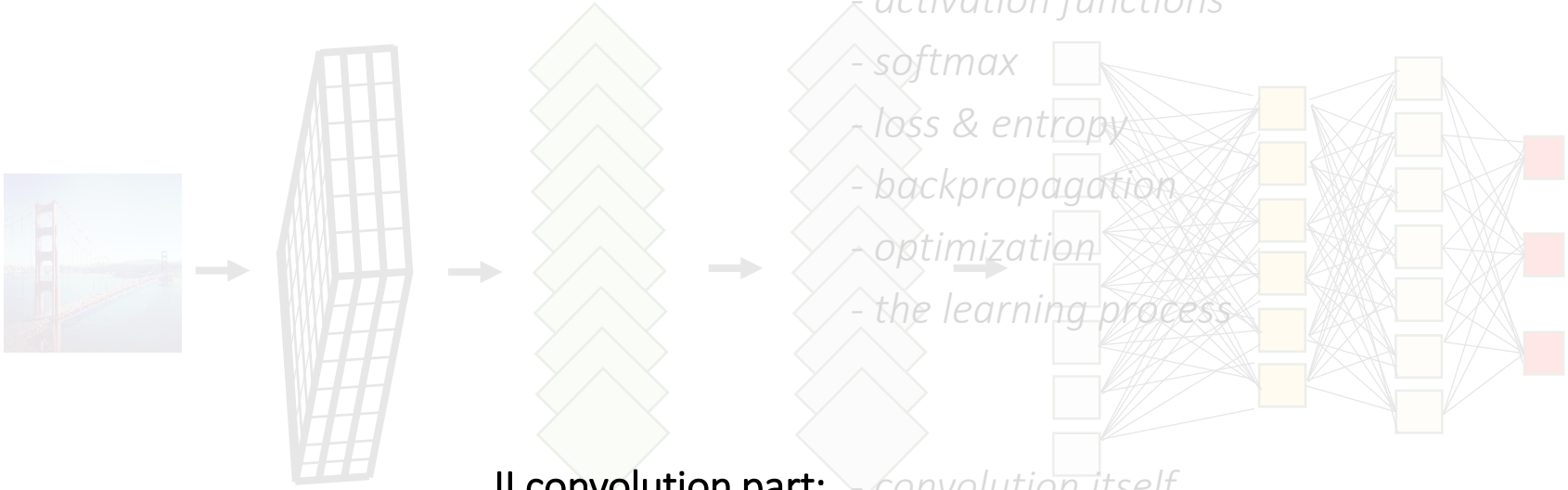
```
class Sigmoid:

    def forward(self, M):

        sigm        = np.clip(1/(1 + np.exp(-M)), 1e-7, 1 - 1e-7)
        self.output = sigm
        self.inputs = sigm #needed for back prop
```

outline

0 intro

I the core:

- *a single neuron*
- *layers of neurons*
- *activation functions*
- *softmax*
- *loss & entropy*
- *backpropagation*
- *optimization*
- *the learning process*

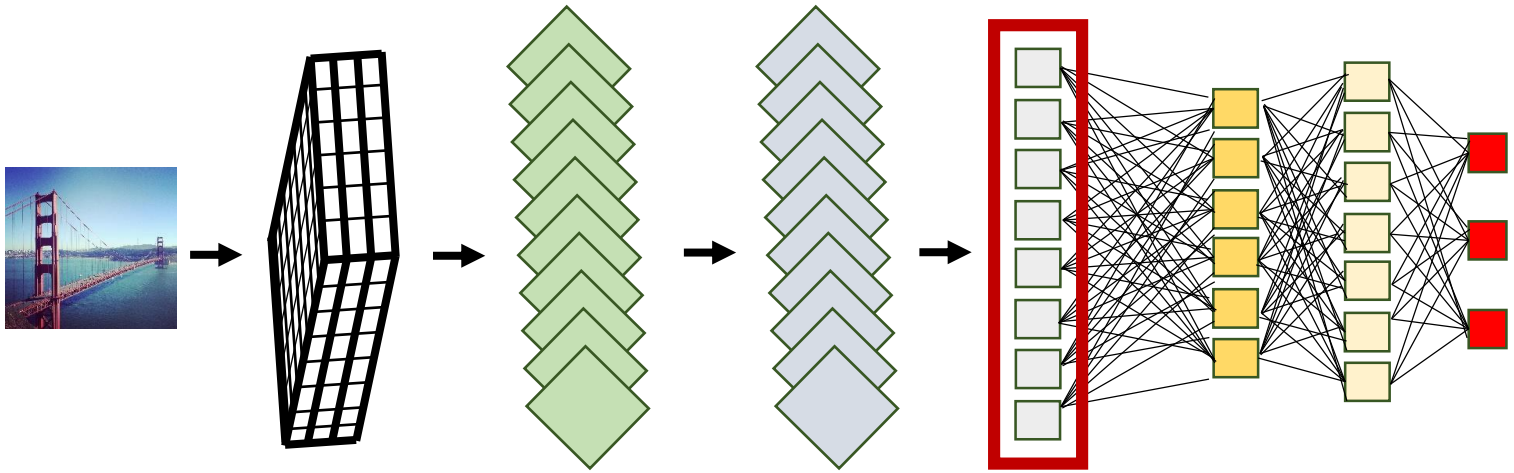


II convolution part:

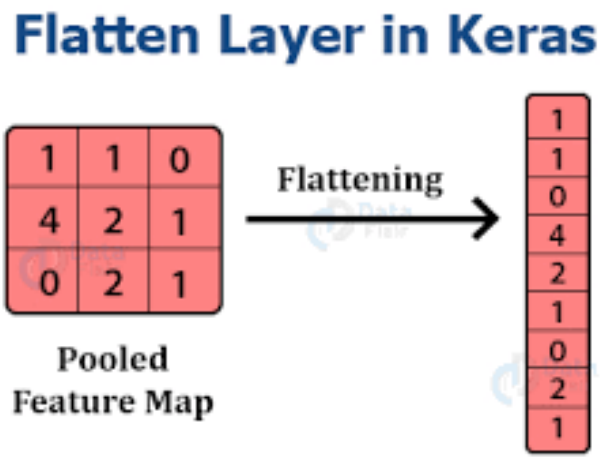
- *convolution itself*
- *pooling*
- *sigmoid*
- *flattening*
- *backpropagation - again*
- *testing the CNN & final remarks*

We are almost done with our CNN

→ linking the CNN part to the dense layer

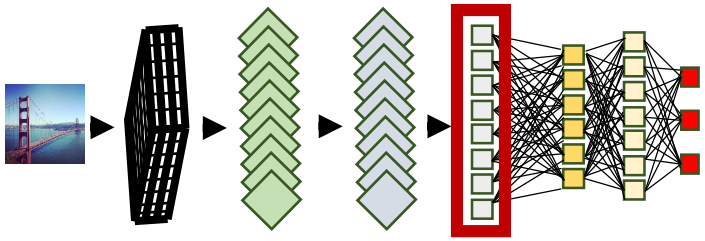


called **flattening**



We are almost done with our CNN

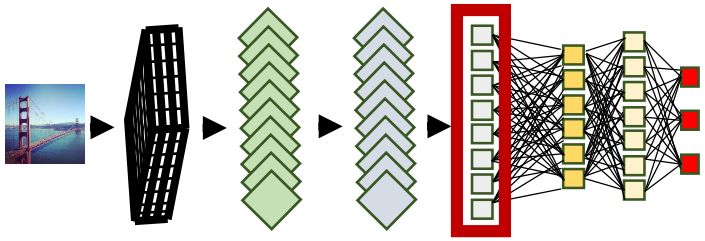
→ linking the CNN part to the dense layer



```
class Flat:

    def forward(self, M):

        self.inputs = M

        xImgShape  = M.shape[0]
        yImgShape  = M.shape[1]
        numChans   = M.shape[2]
        numImds    = M.shape[3]

       #turning each image into a vector of length L
        L = xImgShape*yImgShape*numChans
```

We are almost done with our CNN

→ linking the CNN part to the dense layer

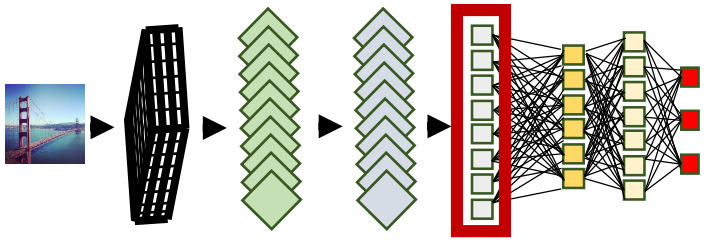


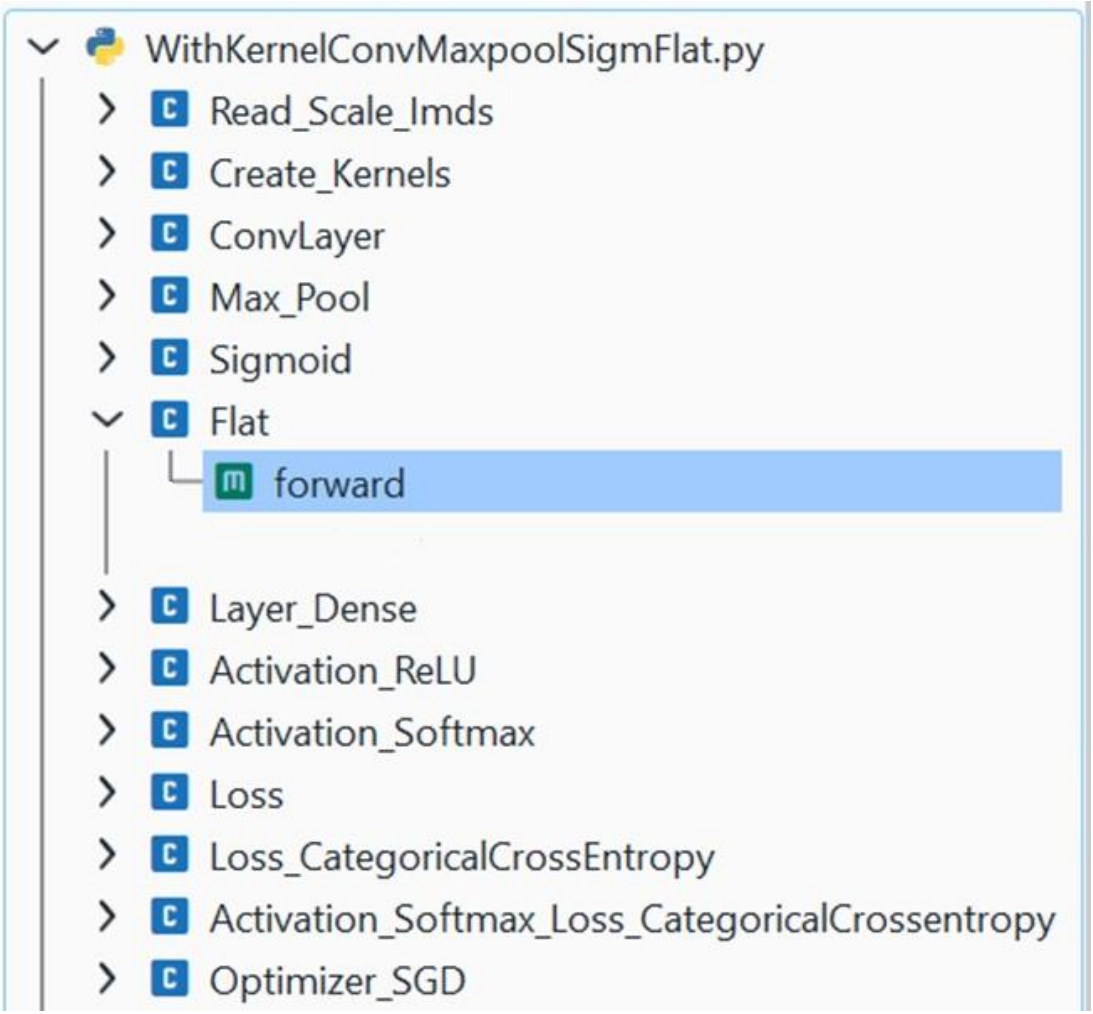
```
class Flat:

    def forward(self, M):

        self.inputs = M

        xImgShape  = M.shape[0]
        yImgShape  = M.shape[1]
        numChans   = M.shape[2]
        numImds    = M.shape[3]

         #turning each image into a vector of length L
        L = xImgShape*yImgShape*numChans

output = np.zeros((numImds,L))

for i in range(numImds):
    output[i,:] = M[:,:,:,i].reshape((1,L))

self.output  = output
```

making sure that every image gets flatted separately
→ using a (slow) **for** loop

We are almost done with our CNN

→ saving and checking the CNN






our network is complete now

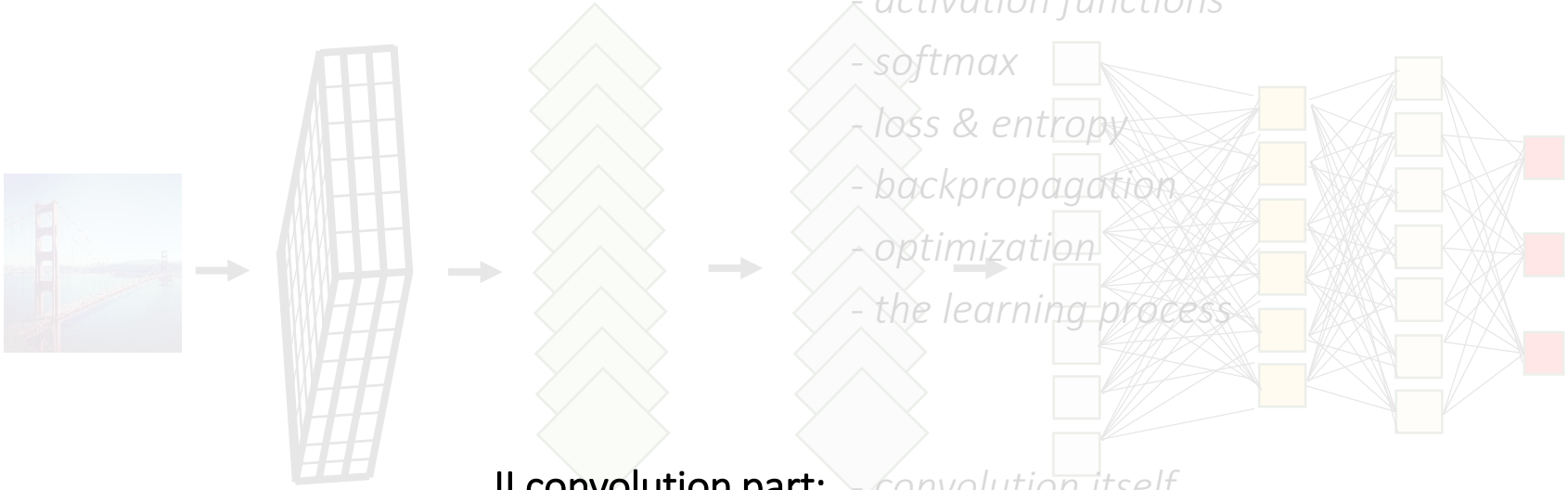
→ we still have to teach it how to learn

outline

0 intro

I the core:

- *a single neuron*
- *layers of neurons*
- *activation functions*
- *softmax*
- *loss & entropy*
- *backpropagation*
- *optimization*
- *the learning process*



II convolution part:

- *convolution itself*
- *pooling*
- *sigmoid*
- *flattening*
- ***backpropagation - again***
- *testing the CNN & final remarks*

We have created all parts needed for a CNN

→ next step: implement the entire backpropagation
→ starting with the simplest layer first: **flattening**

the **flattening** layer only rearranges the data

→ no derivatives are needed
→ turning a vector from the dense layer into the matrix **M** again

```
class Flat:

    def forward(self, M):
        ...

        def backward(self, dvalues):
```

recall: **dvalues** are the derivatives returned from the next lower layer

the **flattening** layer only rearranges the data

→ no derivatives are needed
→ turning a vector from the dense layer into the matrix **M** again

```
class Flat:

    def forward(self, M):
        ...

        def backward(self, dvalues):

            [xImgShape, yImgShape, numChans, numImds] = \
                                        np.shape(self.inputs)

            dinputs = np.zeros((xImgShape, yImgShape, \
                                        numChans, numImds))

            for i in range(numImds):
                dinputs[:,:,:,i] =\
                dvalues[i,:].reshape((xImgShape, yImgShape,\
                                        numChans))
```

recall: **dvalues** are the derivatives returned from the next lower layer

turning each vector **dvalues** back into an image

the **flattening** layer only rearranges the data

→ no derivatives are needed
→ turning a vector from the dense layer into the matrix **M** again

```
class Flat:

    def forward(self, M):
        ...

        def backward(self, dvalues):

                [xImgShape, yImgShape, numChans, numImds] = \
                                                np.shape(self.inputs)

                dinputs = np.zeros((xImgShape, yImgShape, \
                                                numChans, numImds))

                for i in range(numImds):
                        dinputs[:,:,:,i] =\
                        dvalues[i,:].reshape((xImgShape, yImgShape,\
                                                numChans))


        self.dinputs = dinputs
```

checking, if the backward part is recovering the image:

same as before

```
import WithKernelConvMaxpoolSigmFlatBP as My_ANN
import matplotlib.pyplot as plt

minibatch_size = 4
ANN_size       = [128,128]

[M, C] = My_ANN.Read_Scale_Imds.Read_Scale(minibatch_size, ANN_size)

#initializing three convolution layer
Conv1 = My_ANN.ConvLayer()
MP1   = My_ANN.Max_Pool()
F     = My_ANN.Flat()
```

running the layers

```
Conv1.forward(M,2,1)
MP1.forward(Conv1.output,5,5)
F.forward(MP1.output)
```

checking, if the backward part is recovering the image:

```
import WithKernelConvMaxpoolSigmFlatBP as My_ANN
import matplotlib.pyplot as plt

minibatch_size = 4
ANN_size       = [128,128]

[M, C] = My_ANN.Read_Scale_Imds.Read_Scale(minibatch_size, ANN_size)

#initializing layer
Conv1 = My_ANN.ConvLayer()
MP1   = My_ANN.Max_Pool()
F     = My_ANN.Flat()

Conv1.forward(M,2,1)
MP1.forward(Conv1.output,5,5)
F.forward(MP1.output)


F.backward(F.output)

MP1out = MP1.output
Test   = F.dinputs
```

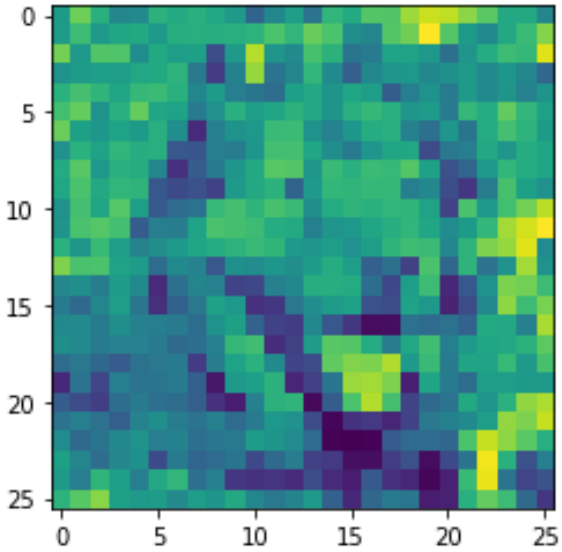
same as before

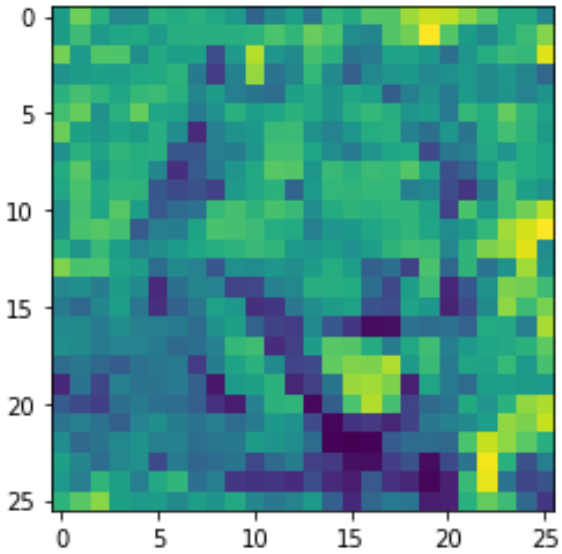
running the layers

checking if **`F.backward`** maintains the structure of **`M`**, if returned as vector from the dense layer

checking, if the backward part is recovering the image:

```
Conv1.forward(M,2,1)
MP1.forward(Conv1.output,5,5)
F.forward(MP1.output)


F.backward(F.output)

MP1out = MP1.output
Test   = F.dinputs


plt.imshow(Test[:,:,1,0])
plt.show()
plt.imshow(MP1out[:,:,1,0])
```



We have created all parts needed for a CNN

→ next step: implement the entire backpropagation
→ continuing with the next layer: **sigmoid**

recall: we need to multiply the inner derivative with **`dvalues`**

inner derivative:

$$\frac{d}{dx}\sigma(x) = \frac{d}{dx}\frac{1}{1+e^{-x}} = \frac{1}{(1+e^{-x})^2}\, e^{-x}$$

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

$$= \frac{1}{1+e^{-x}}\left[\frac{e^{-x}+1-1}{1+e^{-x}}\right]$$

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

$$= \sigma(x)\left(\frac{e^{-x}+1}{1+e^{-x}} - \sigma(x)\right) = \boldsymbol{\sigma(x)\big(1-\sigma(x)\big)}$$

inner derivative:

$$\frac{d}{dx}\sigma(x) = \boldsymbol{\sigma(x)}\big(\mathbf{1} - \boldsymbol{\sigma(x)}\big)$$

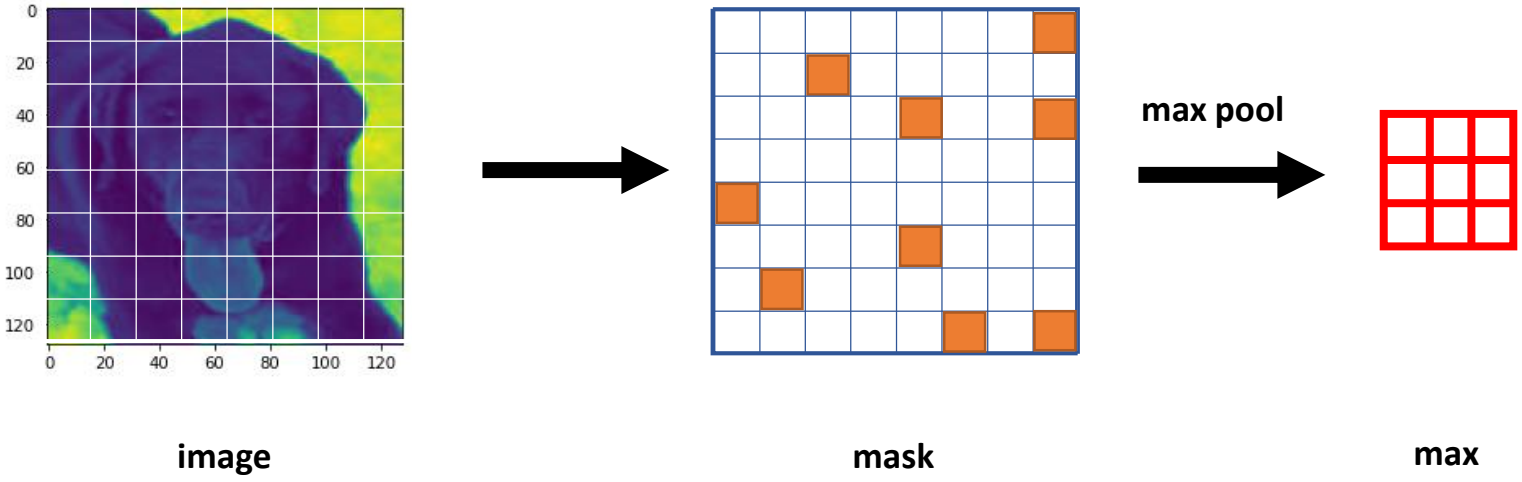
```
class Sigmoid:

    ...

    def backward(self, dvalues):

        sigm          = self.inputs
        deriv         = np.multiply(sigm, (1 - sigm))
        self.dinputs = np.multiply(deriv, dvalues)
```

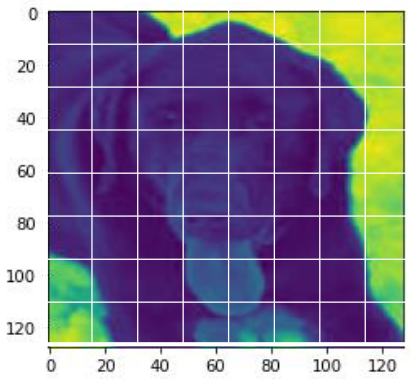
recall: **dvalues** are the derivatives returned from the next lower layer

We have created all parts needed for a CNN

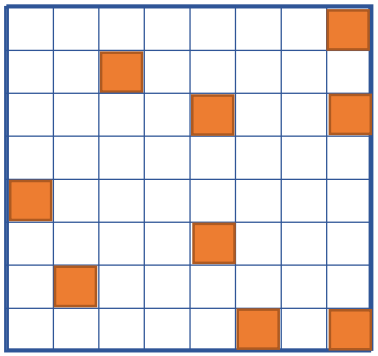
→ next step: implement the entire backpropagation
→ continuing with the next layer: **max pool**

again, no derivatives, only need to multiply the **`mask`** with **`dvalues`**



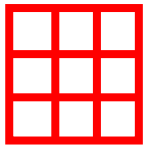



**image**

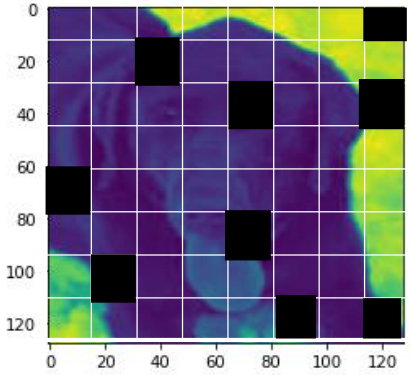


**mask**

**max pool**

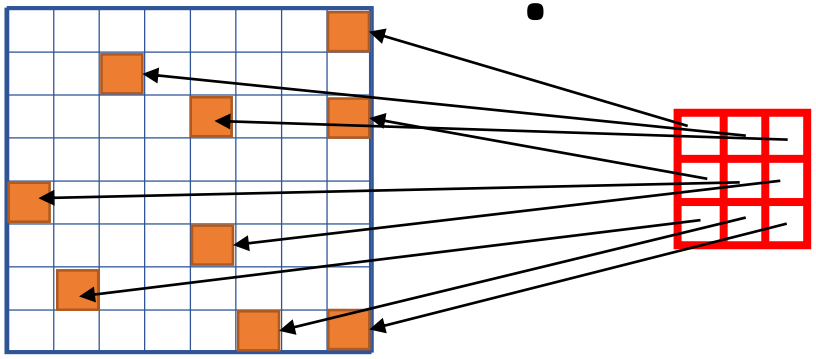


**max**




**`dinputs`**

*



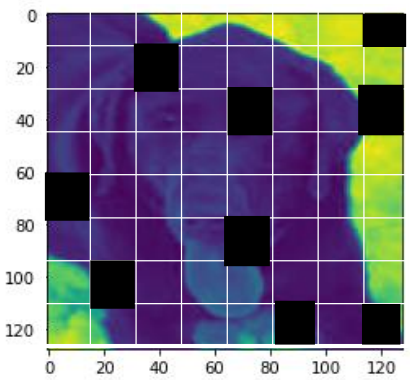

**mask**

**`dvalues`**

Let us first go back to the end of the **forward** part:

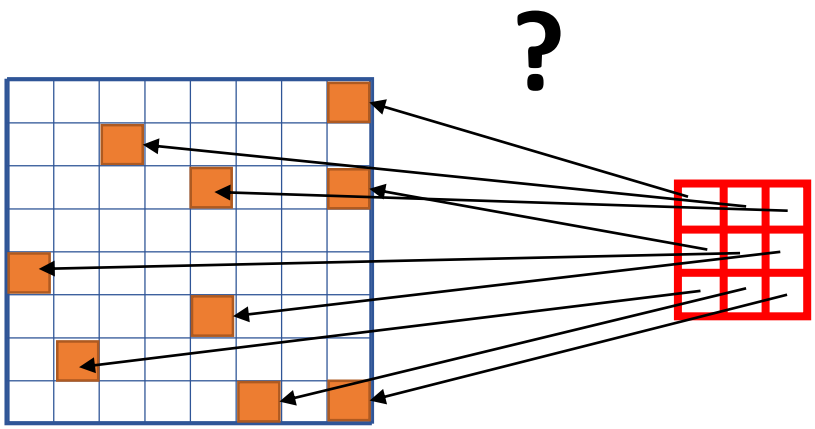
```
mask       = imagePadded_copy

mask_orig = mask

#normalizing to ones and zeros
mask = np.matrix.round(mask/(mask + 1e-7))

#storing info, also for backpropagation
self.xKernShape = xK
self.yKernShape = yK
self.output     = output
self.mask       = mask

self.mask_orig  = mask_orig

self.stride     = stride
```

We are going to need the number of times how often a pixel value was a maximum (see later)




**dinputs**

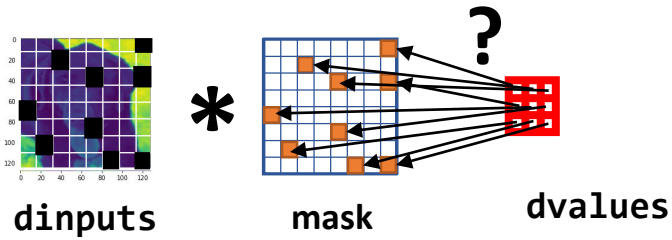
*




**mask** **dvalues**

We have the connection between **mask** coordinates and **dvalues** already from the **forward** part → **backward** part will be similar!

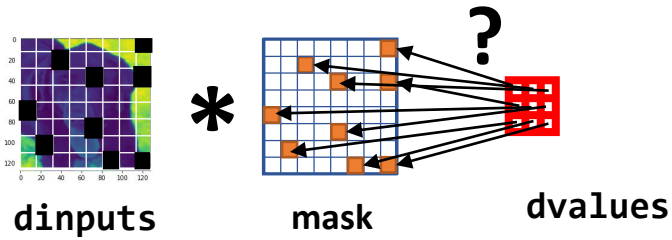
```
def backward(self, dvalues):

    xd = dvalues.shape[0]
    yd = dvalues.shape[1]

    numChans = dvalues.shape[2]
    numImds  = dvalues.shape[3]
```




```
def backward(self, dvalues):

    xd = dvalues.shape[0]
    yd = dvalues.shape[1]

    numChans = dvalues.shape[2]
    numImds  = dvalues.shape[3]

    mask           = self.mask
    mask_copy      = mask
    mask_orig      = self.mask_orig
    mask_orig_copy = mask_orig
    dinputs        = np.zeros(mask.shape)


    stride  = self.stride
    xK      = self.xKernShape
    yK      = self.yKernShape
```

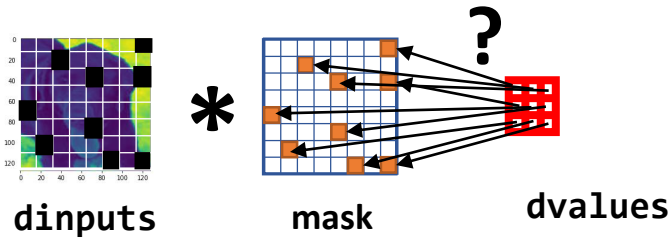
we will see soon, why we need that

output of the current layer which is the input of the next higher layer




```
def backward(self, dvalues):

    xd = dvalues.shape[0]
    yd = dvalues.shape[1]

    numChans = dvalues.shape[2]
    numImds  = dvalues.shape[3]

    mask               = self.mask
    mask_copy          = mask
    mask_orig          = self.mask_orig
    mask_orig_copy     = mask_orig

    dinputs    = np.zeros(mask.shape)

    stride  = self.stride
    xK      = self.xKernShape
    yK      = self.yKernShape

    for i in range(numImds):
        for y in range(yd):
            for x in range(xd):
                for c in range(numChans):
```

We have the connection between **mask** coordinates and **dvalues** already from the **forward** part
→ **backward** part will be similar!

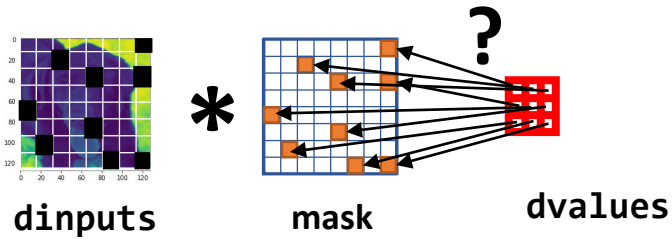



```
def backward(self, dvalues):
    ...
    mask_orig          = self.mask_orig
    mask_orig_copy     = mask_orig
    dinputs            = np.zeros(mask.shape)

    stride  = self.stride
    xK      = self.xKernShape
    yK      = self.yKernShape

    for i in range(numImds):
        for y in range(yd):
            for x in range(xd):
                for c in range(numChans):

                    y_start = y*stride
                    y_end   = y*stride + yK
                    x_start = x*stride
                    x_end   = x*stride + xK

                    sx      = slice(x_start,x_end)
                    sy      = slice(y_start,y_end)
```

just copy/paste from **forward** part

We have the connection between **mask** coordinates and **dvalues** already from the **forward** part
→ **backward** part will be similar!

```
def backward(self, dvalues):

    ...

    y_start = y*stride
    y_end   = y*stride + yK
    x_start = x*stride
    x_end   = x*stride + xK

    sx      = slice(x_start,x_end)
    sy      = slice(y_start,y_end)
```




```
dinputs[sx,sy,c,i]        += mask_copy[sx,sy,c,i]*dvalues[x, y, c, i]
```

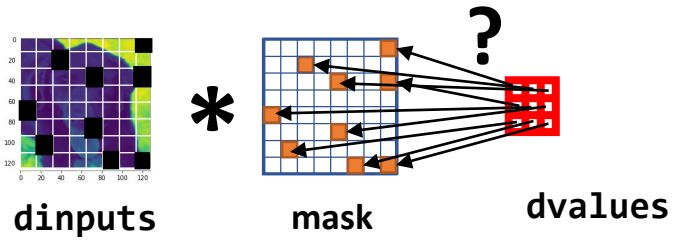
```
mask_orig_copy[sx,sy,c,i] -= mask[sx,sy,c,i]
```

This is a countdown!

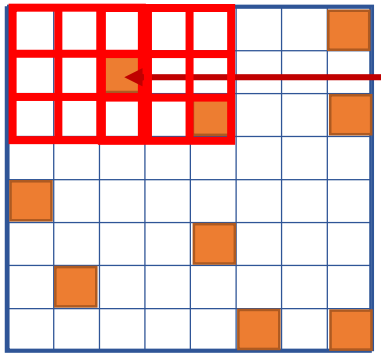
```
mask_copy[sx,sy,c,i] = \
np.round(mask_orig_copy[sx,sy,c,i]/(mask_orig_copy[sx,sy,c,i]+ 1e-7))
```

Will equal *zero*, if pixel value was a max only once, will equal *one* else!

```
def backward(self, dvalues):

    ...
```




```
dinputs[sx,sy,c,i] += mask_copy[sx,sy,c,i]*dvalues[x, y, c, i]
```

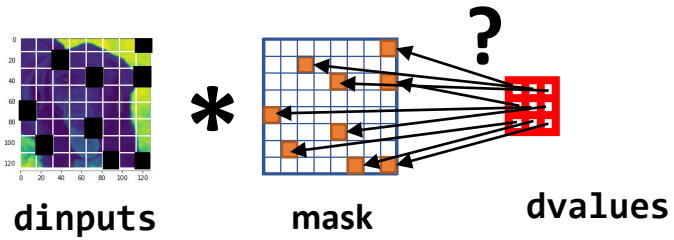


Previous value gets overwritten by new **dvalues** value, if **stride** does not equal kernel size!

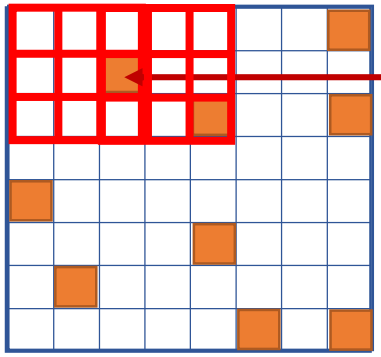
```
mask_copy[sx,sy,c,i] = 0
```

Deletes *ones* that have already been used in previous window.
+= makes sure that previous entries in **dinputs** are not going to be deleted

```
def backward(self, dvalues):
    ...
```



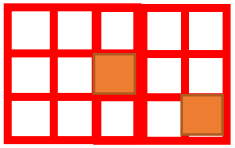

```
dinputs[sx,sy,c,i] += mask_copy[sx,sy,c,i]*dvalues[x, y, c, i]
```



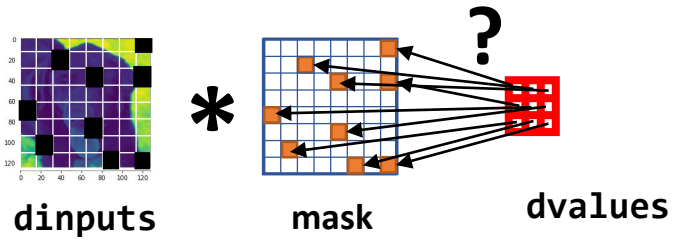
Deletes *ones* that have already been used in previous window.
+= makes sure that previous entries in **`dinputs`** are not going to be deleted

~~`mask_copy[sx,sy,c,i] = 0`~~

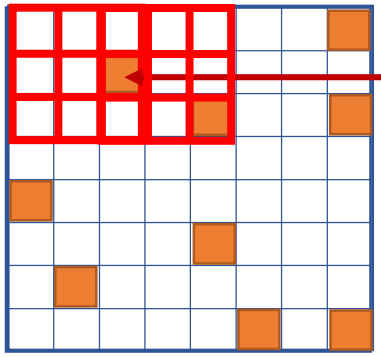
But sometimes one pixel was the max for **more than one pixel** in the output



```
def backward(self, dvalues):

    ...
```

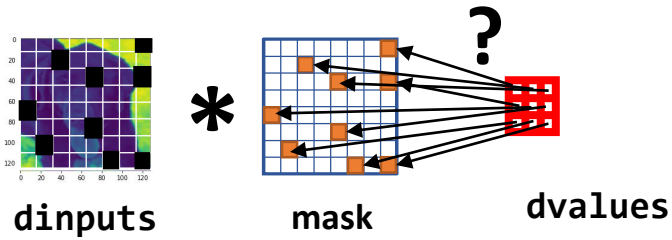



```
dinputs[sx,sy,c,i] += mask_copy[sx,sy,c,i]*dvalues[x, y, c, i]
```

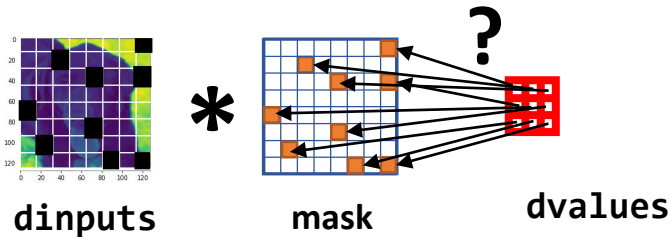


~~`mask_copy[sx,sy,c,i] = 0`~~

```
mask_orig_copy[sx,sy,c,i] -= mask[sx,sy,c,i]

mask_copy[sx,sy,c,i] = \
np.round(mask_orig_copy[sx,sy,c,i]/(mask_orig_copy[sx,sy,c,i]+ 1e-7))
```




```
def backward(self, dvalues):

    ...

    for i in range(numImds):
       for y in range(yd):
           for x in range(xd):
               for c in range(numChans):

                       y_start = y*stride
                       y_end   = y*stride + yK
                       x_start = x*stride
                       x_end   = x*stride + xK

                       sx      = slice(x_start,x_end)
                       sy      = slice(y_start,y_end)

                       dinputs[sx,sy,c,i]   += mask_copy[sx,sy,c,i]\
                                                *dvalues[x, y, c, i]

                       mask_orig_copy[sx,sy,c,i]  -= mask[sx,sy,c,i]

                       mask_copy[sx,sy,c,i] = \
                       np.round(mask_orig_copy[sx,sy,c,i]/(mask_orig_copy[sx,sy,c,i] \
                                                + 1e-7))

    self.dinputs = dinputs
```



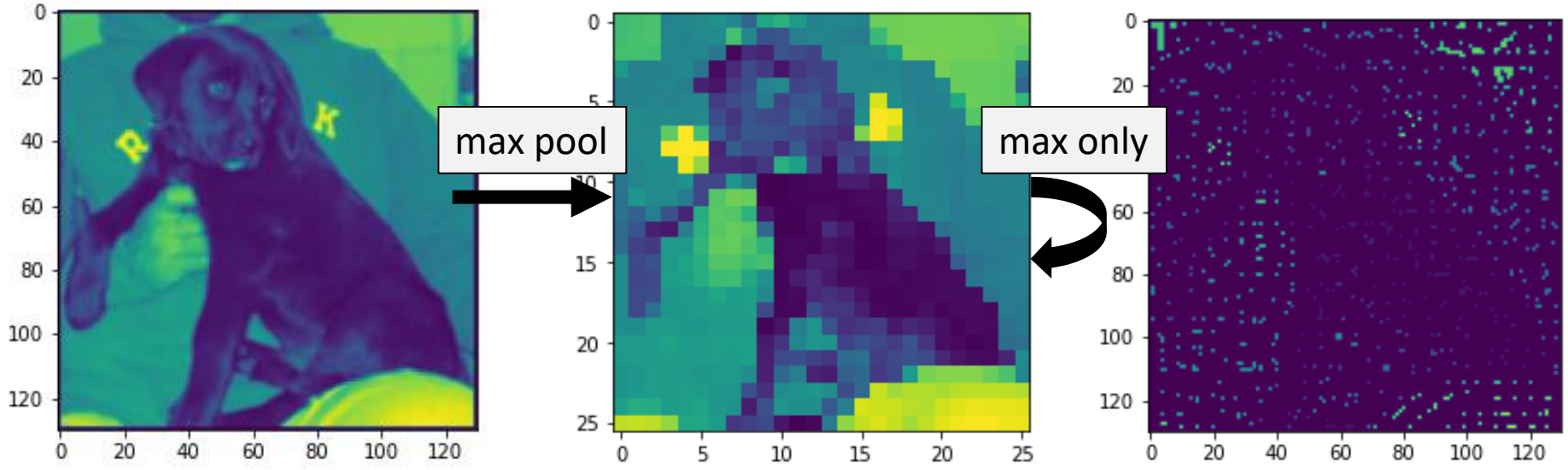

<u>again: let's test the code!</u>

**note:** we want to check if the **`backward`** part of the max pool layer assigns the **`dvalues`** correctly to the image coordinates

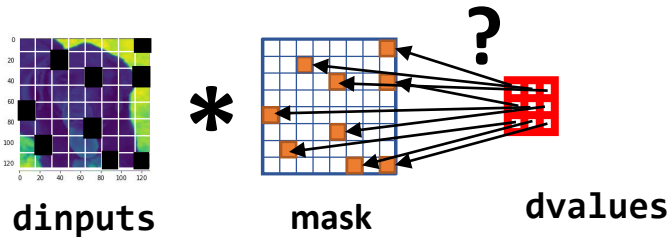
**idea:** We don't have actual **`dvalues`** yet

→ feeding the output of the max pool layer to its **`backward`** part

→ should return maxima of the input image **(easy to check)**



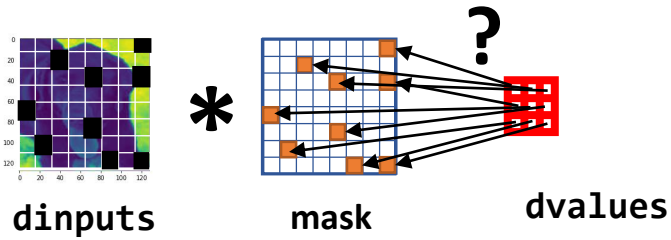
again: let's test the code!




same as before

```
import WithKernelConvMaxpoolBPSigmBPFlatBP as My_ANN
import matplotlib.pyplot as plt

minibatch_size = 4
ANN_size       = [128,128]

[M, C] = My_ANN.Read_Scale_Imds.Read_Scale(minibatch_size, ANN_size)

#initializing layer
Conv1 = My_ANN.ConvLayer()
MP1   = My_ANN.Max_Pool()
```

## again: let's test the code!




```
import WithKernelConvMaxpoolBPSigmBPFlatBP as My_ANN
import matplotlib.pyplot as plt

minibatch_size = 4
ANN_size       = [128,128]

[M, C] = My_ANN.Read_Scale_Imds.Read_Scale(minibatch_size, ANN_size)

#initializing layer
Conv1 = My_ANN.ConvLayer()
MP1   = My_ANN.Max_Pool()


Conv1.forward(M,2,1)
MP1.forward(Conv1.output,3,5)
MP1.backward(MP1.output)

MP1in  = Conv1.output
MP1out = MP1.output
D      = MP1.dinputs
```

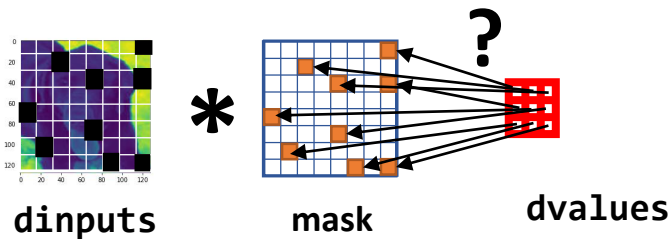
take **stride ≠ KernShape** in order to check for a general case

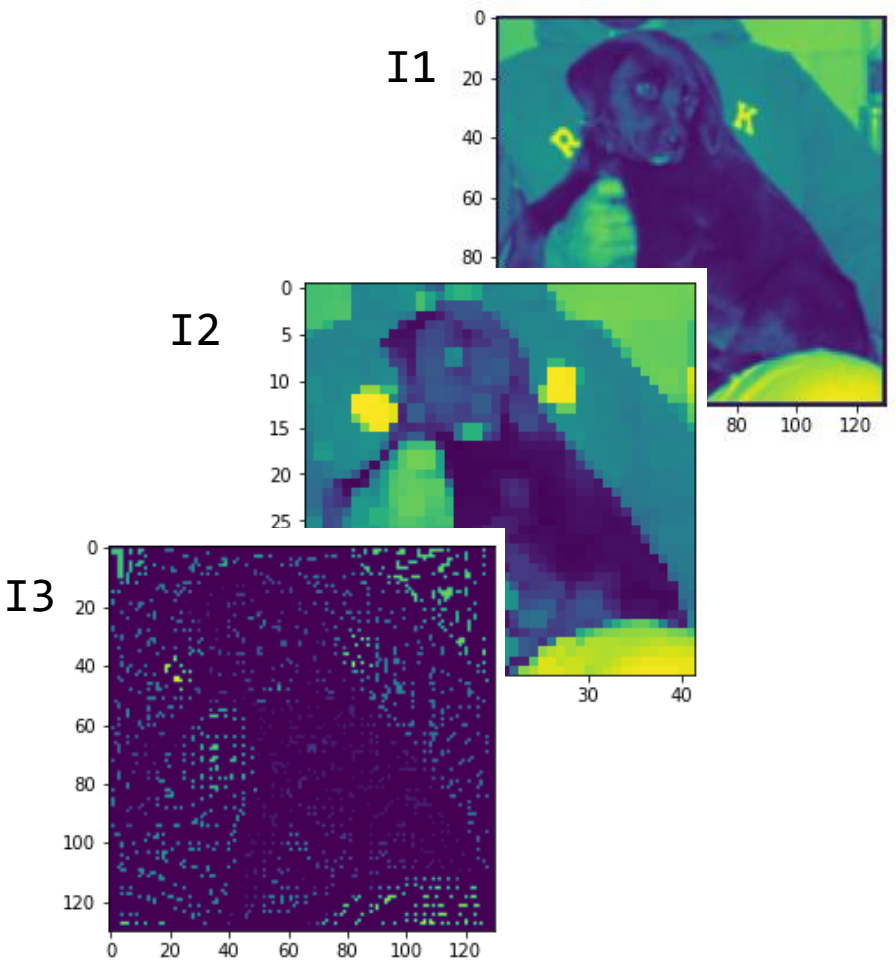
feeding output to **backward** part

## again: let's test the code!

```
Conv1.forward(M,2,1)
MP1.forward(Conv1.output,3,5)
MP1.backward(MP1.output)

MP1in  = Conv1.output
MP1out = MP1.output
D      = MP1.dinputs

I1 = MP1in[:,:,0,0]
I2 = MP1out[:,:,0,0]
I3 = D[:,:,0,0]

plt.imshow(I1)
plt.show()
plt.imshow(I2)
plt.show()
plt.imshow(I3)
```

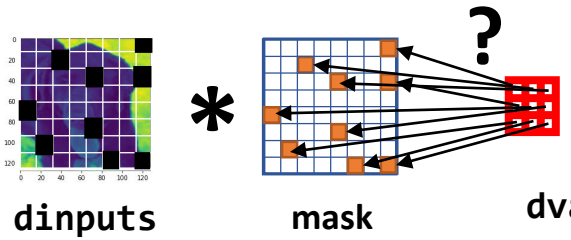
picking a random image



## again: let's test the code!

```
I1 = MP1in[:,:,0,0]
I2 = MP1out[:,:,0,0]
I3 = D[:,:,0,0]

plt.imshow(I1)
plt.show()
plt.imshow(I2)
plt.show()
plt.imshow(I3)
```




I2

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 538 | 544 | 545 | 551 | 543 | 357 | 364 |
| 1 | 538 | 544 | 547 | 550 | 544 | 369 | 361 |
| 2 | 538 | 544 | 548 | 554 | 554 | 369 | 361 |
| 3 | 544 | 547 | 548 | 554 | 554 | 361 | 358 |
| 4 | 544 | 544 | 499 | 364 | 362 | 368 | 375 |
| 5 | 424 | 363 | 355 | 355 | 350 | 363 | 375 |
| 6 | 352 | 354 | 358 | 358 | 356 | 356 | 357 |
| 7 | 352 | 353 | 354 | 359 | 368 | 368 | 350 |
| 8 | 353 | 353 | 362 | 362 | 360 | 359 | 356 |
| 9 | 357 | 356 | 367 | 367 | 366 | 365 | 356 |

I1 stride = 3; xK = yK = 5

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 538 | 538 | 538 | 538 | 541 | 541 |
| 2 | 0 | 538 | 538 | 538 | 538 | 541 | 541 |
| 3 | 0 | 535 | 535 | 538 | 538 | 541 | 541 |
| 4 | 0 | 535 | 535 | 538 | 538 | 541 | 541 |
| 5 | 0 | 535 | 535 | 538 | 538 | 538 | 538 |
| 6 | 0 | 535 | 535 | 538 | 538 | 538 | 538 |
| 7 | 0 | 535 | 535 | 538 | 538 | 541 | 541 |
| 8 | 0 | 535 | 535 | 538 | 538 | 541 | 541 |
| 9 | 0 | 535 | 535 | 538 | 538 | 538 | 544 |

***and so on…***

I3

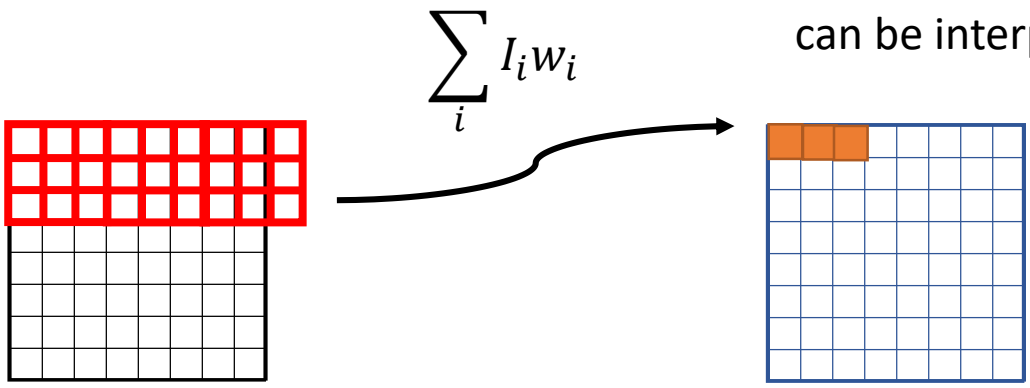
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 538 | 538 | 538 | 538 | 0 | 0 |
| 2 | 0 | 538 | 538 | 538 | 538 | 0 | 0 |
| 3 | 0 | 0 | 0 | 538 | 538 | 0 | 0 |
| 4 | 0 | 0 | 0 | 538 | 538 | 0 | 0 |
| 5 | 0 | 0 | 0 | 538 | 538 | 0 | 0 |
| 6 | 0 | 0 | 0 | 538 | 538 | 0 | 0 |
| 7 | 0 | 0 | 0 | 538 | 538 | 0 | 0 |
| 8 | 0 | 0 | 0 | 538 | 538 | 0 | 0 |
| 9 | 0 | 0 | 0 | 538 | 538 | 0 | 544 |

We have created all parts needed for a CNN

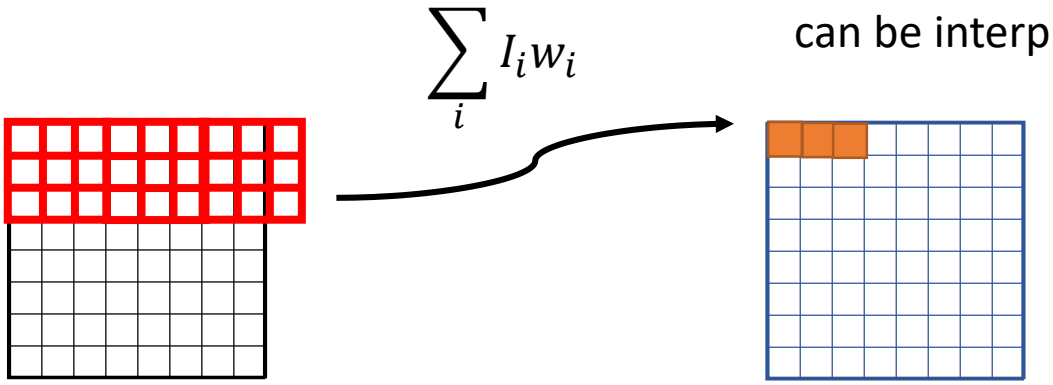
→ next step: implement the entire backpropagation
→ continuing with the next layer: **convolution!**

The **last** part is going to be the most tricky one!

$$(f * g)(x) := \int_{\mathbb{R}^n} f(\zeta)\boldsymbol{g(x-\zeta)}d\zeta$$

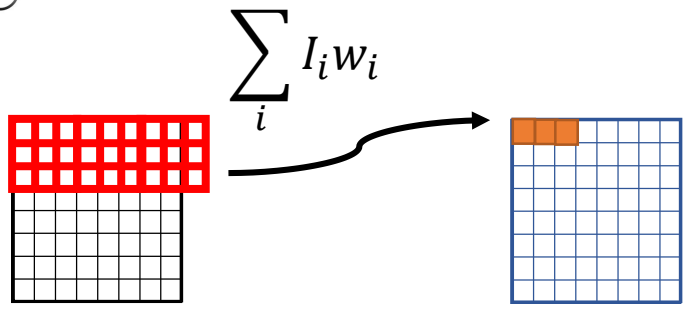



can be interpreted as a neuron!




can be interpreted as a neuron!

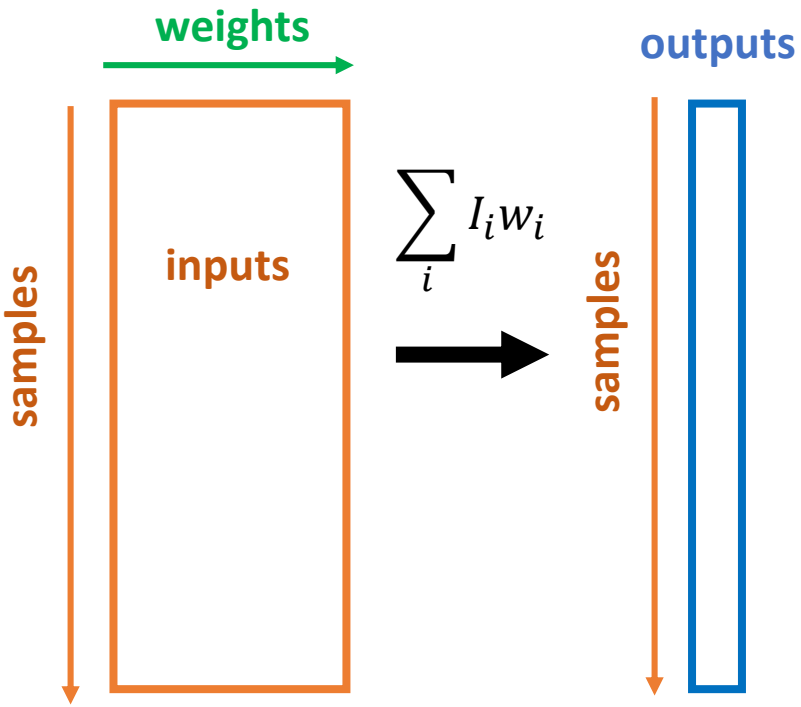
```
dweights      = -dvalues * inputs
dinputs       = -dvalues * weights
dbiases       = -dvalues
```




```
dweights      = -dvalues * inputs
dinputs       = -dvalues * weights
dbiases       = -dvalues
```

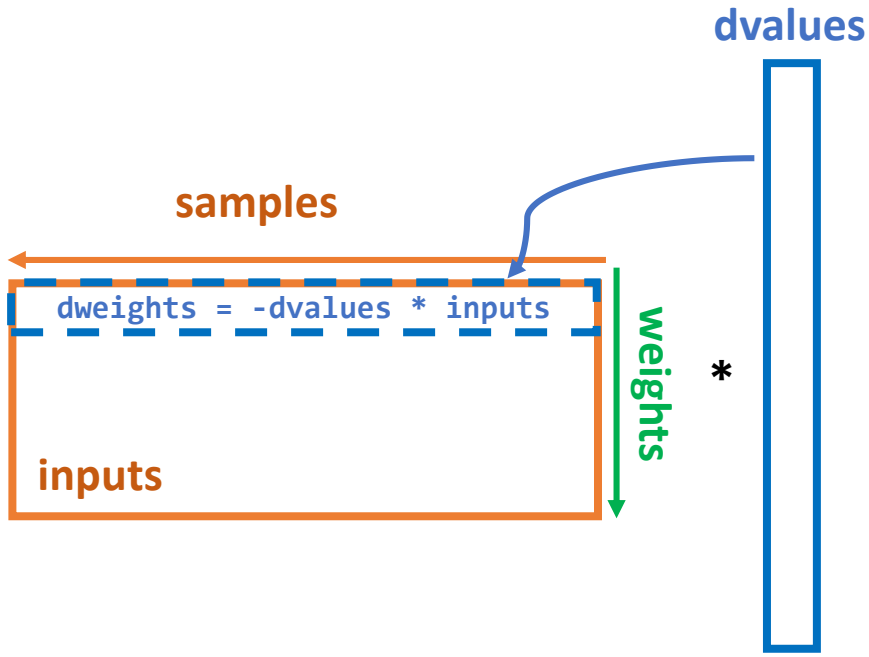
single neuron:

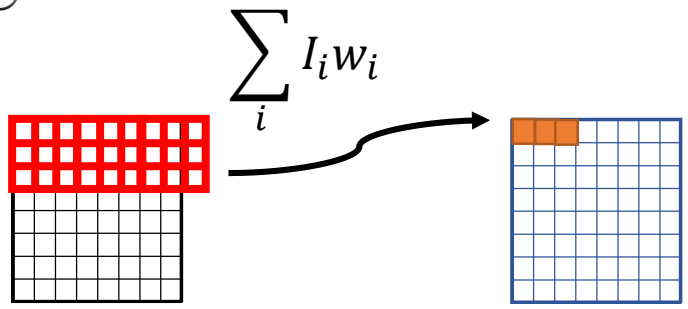
**forward**




**backward**

```
dweights = -dvalues * inputs
```



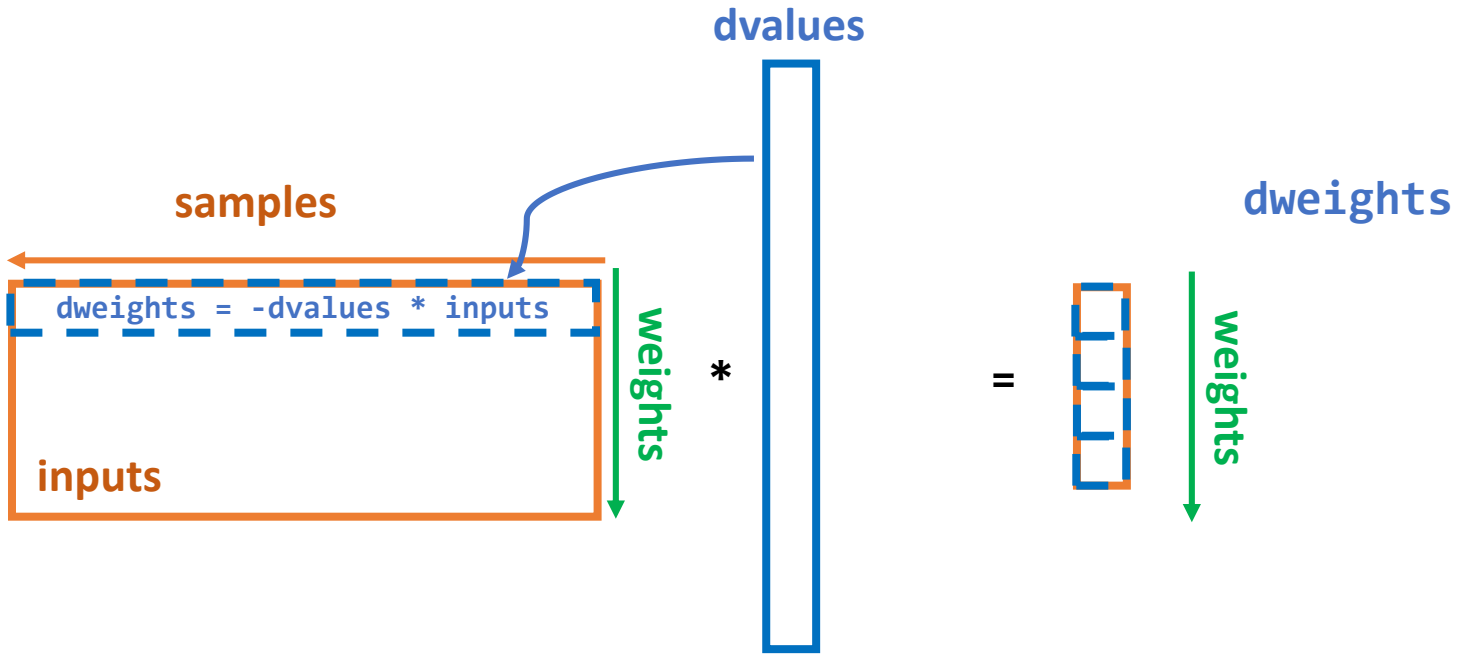


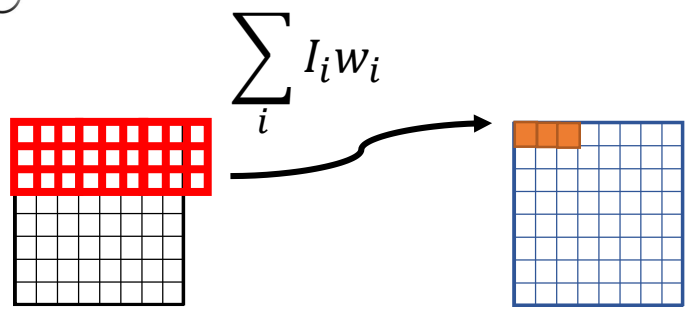

```
dweights      = -dvalues * inputs
dinputs       = -dvalues * weights
dbiases       = -dvalues
```

single neuron:

**backward**

```
dweights = -dvalues * inputs
```

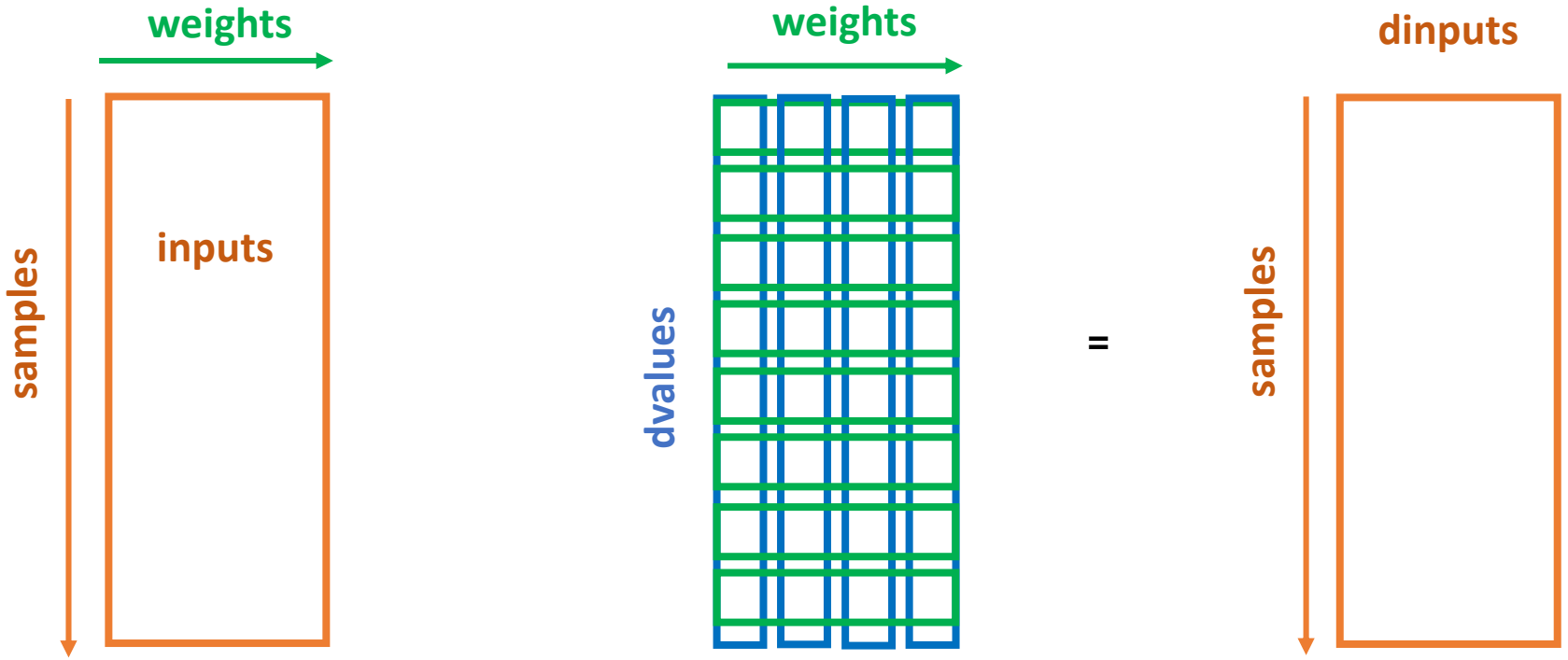


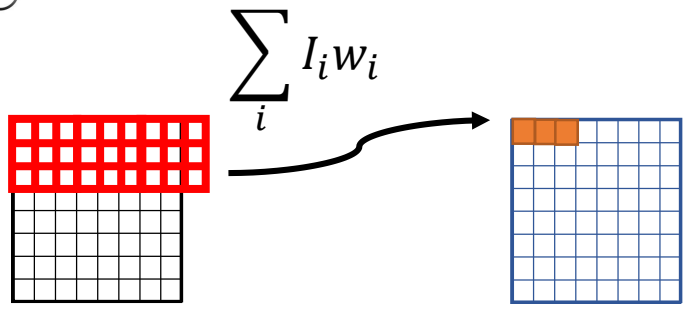



```
dweights      = -dvalues * inputs
dinputs       = -dvalues * weights
dbiases       = -dvalues
```

single neuron:

`backward` **`dinputs = -dvalues * weights`** (we don't need for first conv layer!)

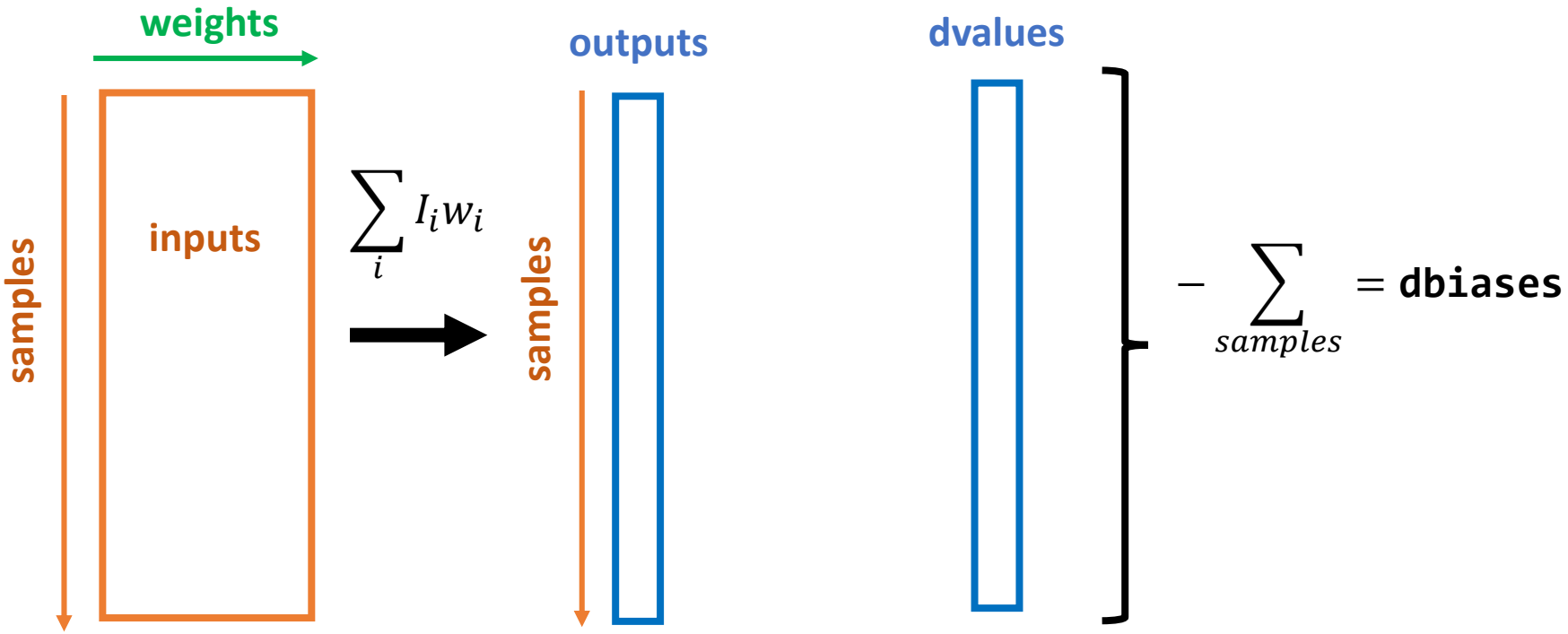





```
dweights      = -dvalues * inputs
dinputs       = -dvalues * weights
dbiases       = -dvalues
```

single neuron:

```
backward    dbiases = -dvalues
```

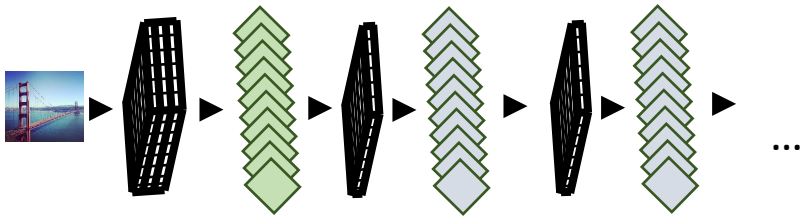


```
class ConvLayer:
    ...

    def backward(self, dvalues):

        biases     = self.biases
        weights    = self.weights

        stride     = self.stride
        padding    = self.padding

        xK         = self.xKernShape
        yK         = self.yKernShape
        NK         = self.Kernnumber

        imagePadded = self.impad

        S = imagePadded.shape
```

As in the previous classes, we need some information from the **forward** part first

```
def backward(self, dvalues):

    ...

    xK          = self.xKernShape
    yK          = self.yKernShape
    NK          = self.Kernnumber

    imagePadded = self.impad

    S = imagePadded.shape

    dinputs     = np.zeros((S[0],S[1],S[2],S[4]))
    dbiases     = np.zeros(biases.shape)
    dweights    = np.zeros(weights.shape)

    xd          = dvalues.shape[0]
    yd          = dvalues.shape[1]
    numChans    = S[2]
    numImds     = dvalues.shape[3]
```

recall: the third index [2] refers to numChan (incoming matrix) and the fourth index [3] refers to Kernnumber (outcoming matrix)

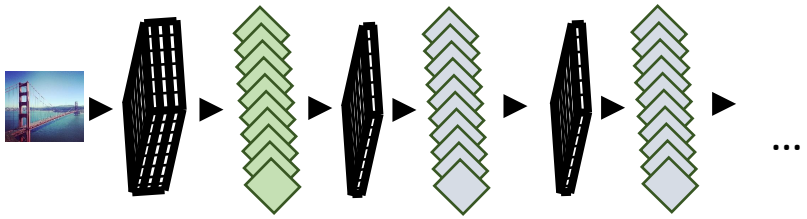


```
def backward(self, dvalues):
    ...
    xd          = dvalues.shape[0]
    yd          = dvalues.shape[1]
    numChans    = S[2]
    numImds     = dvalues.shape[3]

    dinputs     = np.zeros(imagePadded.shape)
    dbiases     = np.zeros(biases.shape)

    #defining matrix for dinputs: note: has do be de-padded at
    #the end; was the same input for all the k
    imagePadded = imagePadded[:,:,:,0,:]
```

same structure
as for **forward** part

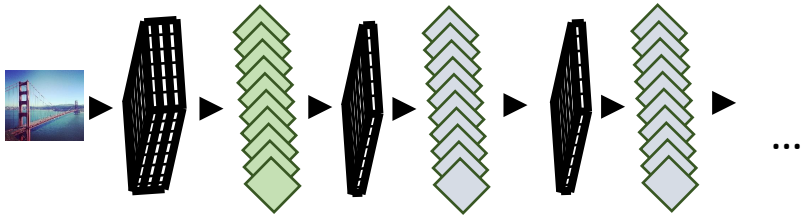
```
    for i in range(numImds):
        currentIm_pad = imagePadded[:,:,:,i]
        for k in range(NK):
            for c in range(numChans):
                for y in range(yd):
                    for x in range(xd):
```

```
def backward(self, dvalues):
    ...
```



```
for i in range(numImds):
    currentIm_pad = imagePadded[:,:,:,:,i]
     for k in range(NK):
        for c in range(numChans):
          for y in range(yd):
              for x in range(xd):

                  y_start = y*stride
                  y_end   = y*stride + yK
                  x_start = x*stride
                  x_end   = x*stride + xK

                  sx      = slice(x_start,x_end)
                  sy      = slice(y_start,y_end)

                  current_slice = currentIm_pad[sx,sy,c]
```

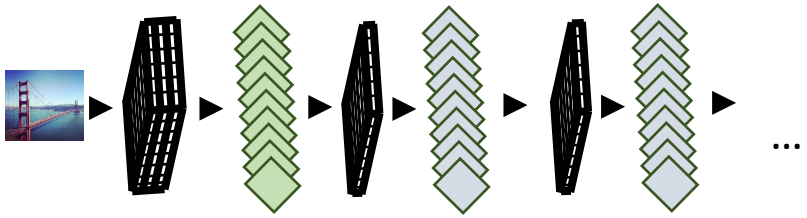
Again, same setup, as in **forward** part



```
def backward(self, dvalues):
    ...

    for i in range(numImds):
        currentIm_pad = imagePadded[:,:,:,:,i]
        for k in range(NK):
            for c in range(numChans):
              for y in range(yd):
                  for x in range(xd):

                      y_start = y*stride
                      y_end   = y*stride + yK
                      x_start = x*stride
                      x_end   = x*stride + xK

                      sx      = slice(x_start,x_end)
                      sy      = slice(y_start,y_end)

                      current_slice = currentIm_pad[sx,sy,c]

                      dweights[:,:,k] += current_slice * \
                                                dvalues[x,y,k,i]

                      dinputs[sx,sy,c,i] += weights[:,:,k]* \
                                                dvalues[x,y,k,i]
```

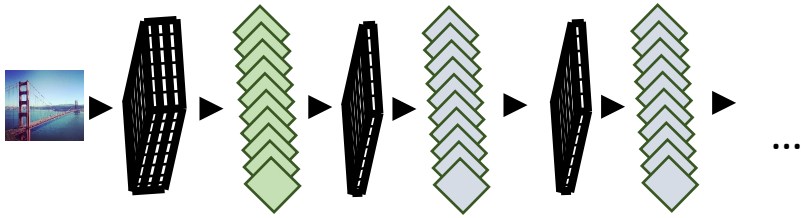
Again, same setup, as in **forward** part

depending on **stride**, some parts of the padded image contributed to different output pixel



```
def backward(self, dvalues):
    ...

    for i in range(numImds):
        currentIm_pad = imagePadded[:,:,:,:,i]
        for k in range(NK):
            for c in range(numChans):
              for y in range(yd):
                  for x in range(xd):


                      y_start = y*stride
                      y_end   = y*stride + yK
                      x_start = x*stride
                      x_end   = x*stride + xK

                      sx      = slice(x_start,x_end)
                      sy      = slice(y_start,y_end)

                      current_slice = currentIm_pad[sx,sy,c]


                      dweights[:,:,k] += current_slice*dvalues[x,y,k,i]
                      dinputs[sx,sy,c,i] += weights[:,:,k]*dvalues[x,y,k,i]

dbiases[0,k] += np.sum(np.sum(dvalues[:,:,k,i],axis=0),axis=0)
```

```
def backward(self, dvalues):
    ...
```

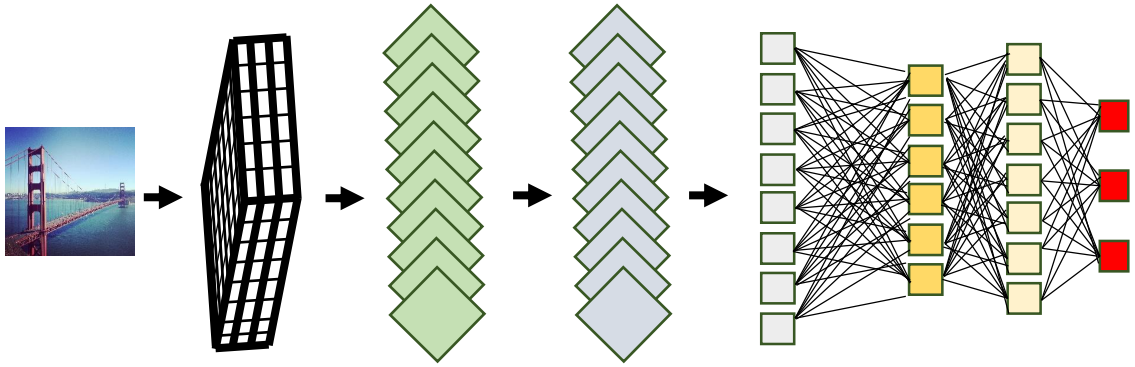


```
y_start = y*stride
y_end   = y*stride + yK
x_start = x*stride
x_end   = x*stride + xK

sx      = slice(x_start,x_end)
sy      = slice(y_start,y_end)

current_slice = currentIm_pad[sx,sy,c]


dweights[:,:,k] += current_slice*dvalues[x,y,k,i]
dinputs[sx,sy,c,i] += weights[:,:,k]*dvalues[x,y,k,i]
```

```
dbiases[0,k] += np.sum(np.sum(dvalues[:,:,k,i],axis=0),axis=0)
```

```
dinputs      = dinputs[padding:S[0]-padding,padding:S[1]-padding,:,:]
self.dinputs = dinputs

self.dbiases  = dbiases
self.dweights = dweights
```

# Congratulations – we are done!

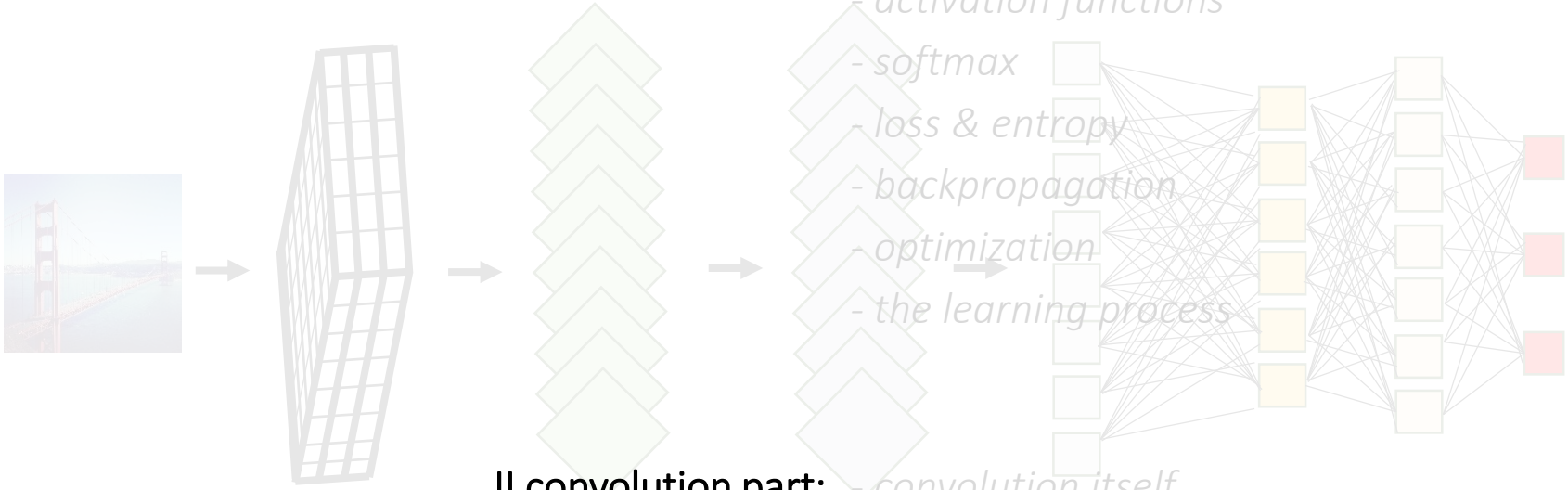


outline

0 intro

I the core:

- *a single neuron*
- *layers of neurons*
- *activation functions*
- *softmax*
- *loss & entropy*
- *backpropagation*
- *optimization*
- *the learning process*

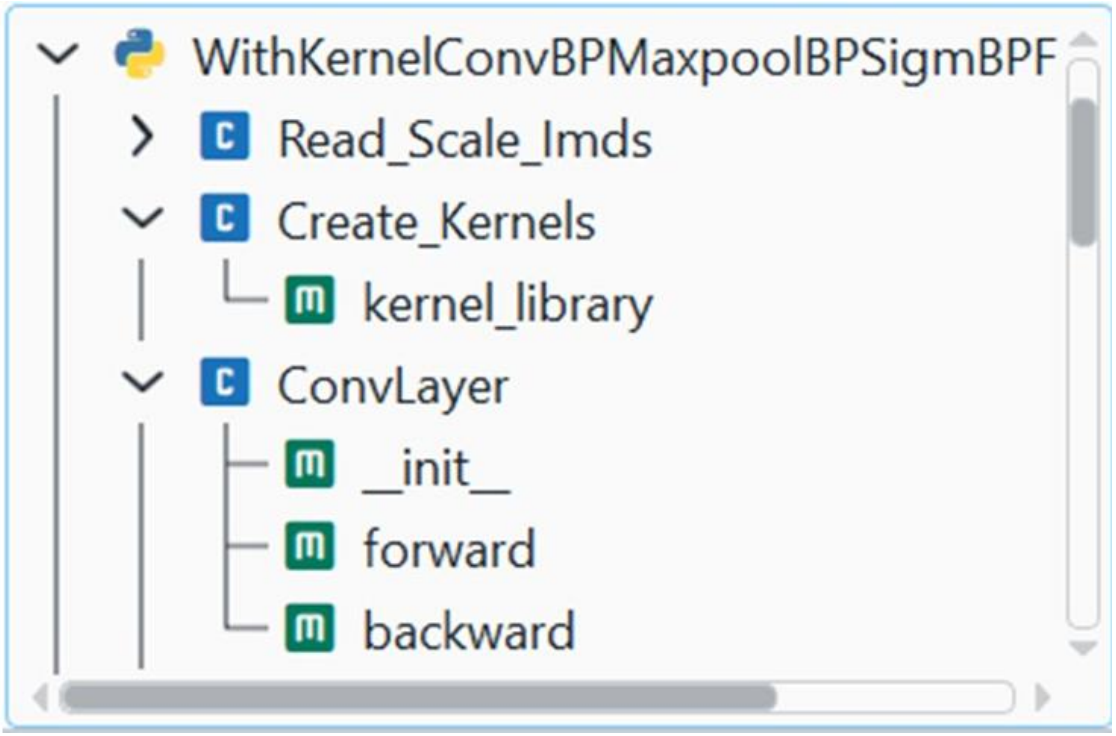


II convolution part:

- *convolution itself*
- *pooling*
- *sigmoid*
- *flattening*
- *backpropagation - again*
- *testing the CNN & final remarks*

We still haven't tested the backpropagation part of the convolution layer yet

→ best way: just run a few convolution layers + backpropagation subsequently
→ choose different values for stride/padding
→ if indices have gotten confused: you will receive an error message

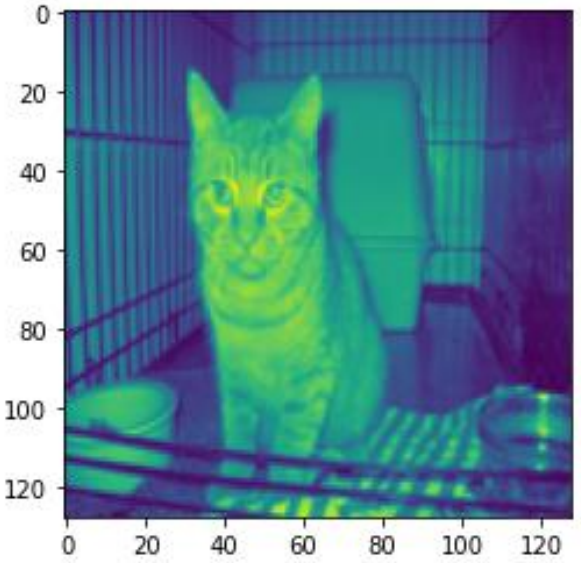


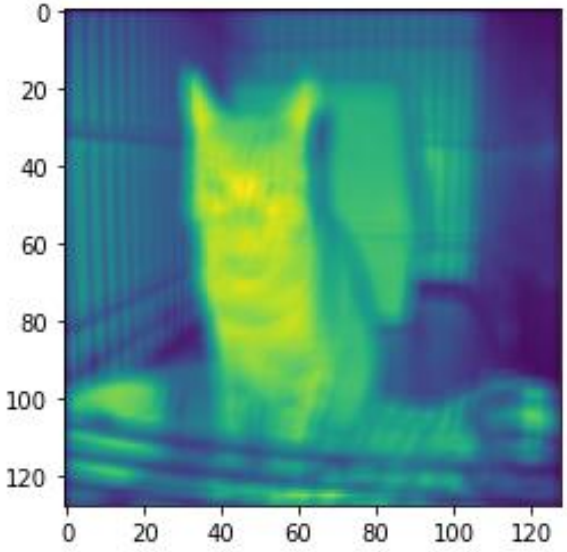
→ best way: just run a few convolution layers + backpropagation subsequently
→ choose different values for stride/padding
→ if indices have gotten confused: you will receive an error message

```
Conv1.forward(M,2,1)
RL1.forward(Conv1.output)
MP1.forward(RL1.output,4,4)

Conv2.forward(MP1.output,1,1)
RL2.forward(Conv2.output)
MP2.forward(RL2.output,1,4)

Conv3.forward(MP2.output,0,3)
RL3.forward(Conv3.output)
MP3.forward(RL3.output,2,2)
```

→ best way: just run a few convolution layers + backpropagation subsequently
→ choose different values for stride/padding
→ if indices have gotten confused: you will receive an error message

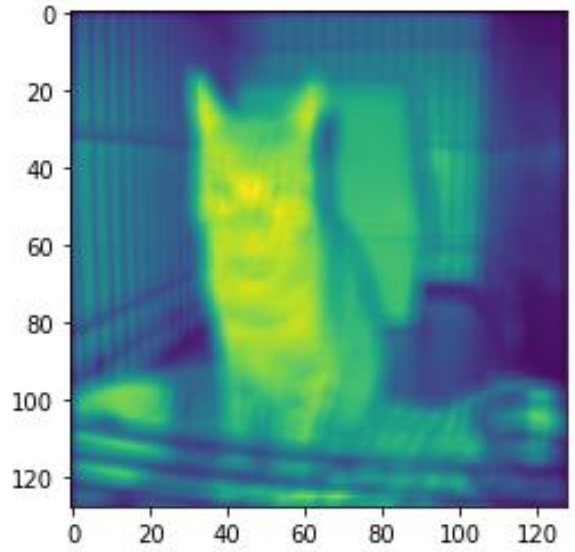
```
Conv1.forward(M,2,1)
RL1.forward(Conv1.output)
MP1.forward(RL1.output,4,4)

Conv2.forward(MP1.output,1,1)
RL2.forward(Conv2.output)
MP2.forward(RL2.output,1,4)

3.forward(MP2.output,0,3)
Forward(Conv3.output)
Forward(RL3.output,2,2)
```

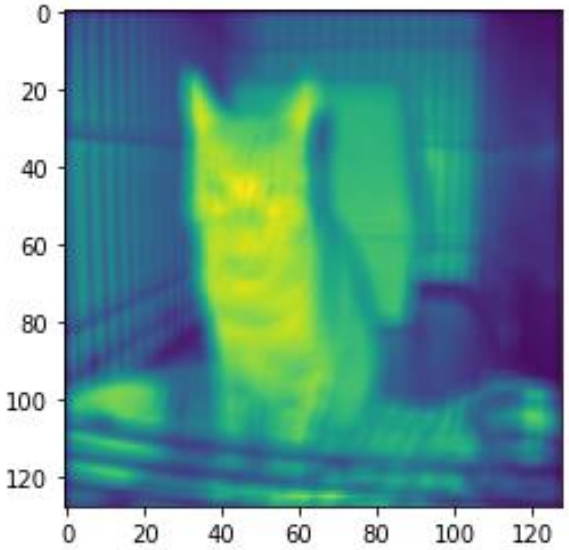


→ best way: just run a few convolution layers + backpropagation subsequently
→ choose different values for stride/padding
→ if indices have gotten confused: you will receive an error message

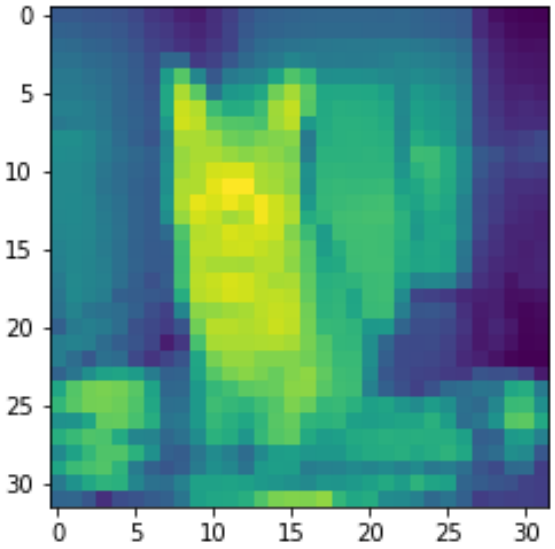
```
Conv1.forward(M,2,1)
RL1.forward(Conv1.output)
MP1.forward(RL1.output,4,4)

Conv2.forward(MP1.output,1,1)
RL2.forward(Conv2.output)
MP2.forward(RL2.output,1,4)

Conv3.forward(MP2.output,0,3)
RL3.forward(Conv3.output)
MP3.forward(RL3.output,2,2)
```

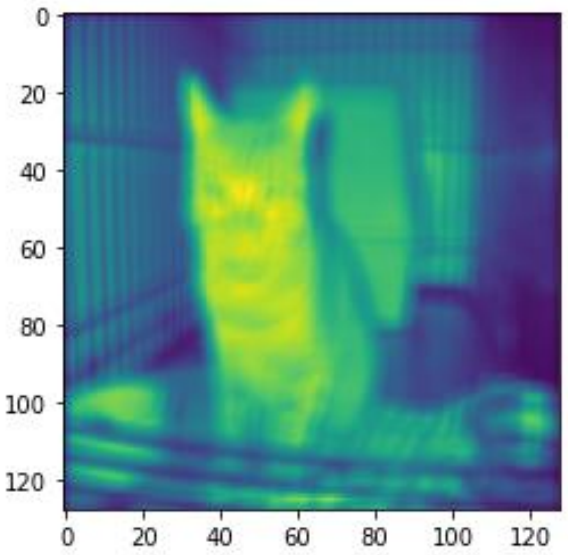


→ best way: just run a few convolution layers + backpropagation subsequently
→ choose different values for stride/padding
→ if indices have gotten confused: you will receive an error message

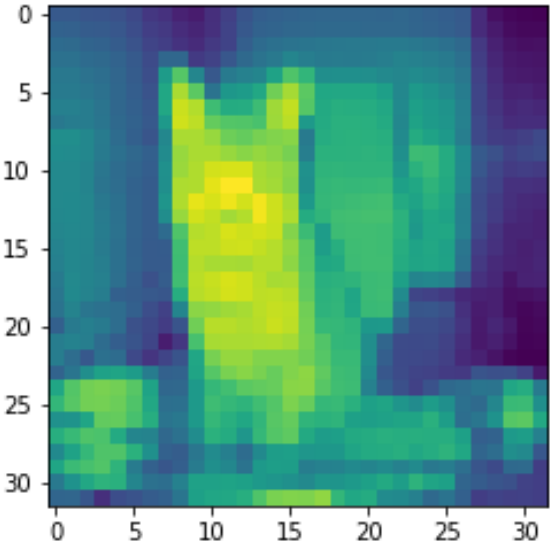
```
Conv1.forward(M,2,1)
RL1.forward(Conv1.output)
MP1.forward(RL1.output,4,4)

Conv2.forward(MP1.output,1,1)
 forward(Conv2.output)
forward(RL2.output,1,4)

3.forward(MP2.output,0,3)
forward(Conv3.output)
forward(RL3.output,2,2)
```



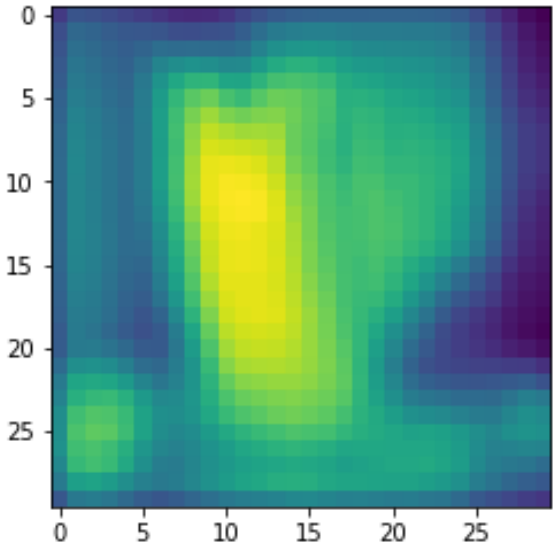



→ best way: just run a few convolution layers + backpropagation subsequently
→ choose different values for stride/padding
→ if indices have gotten confused: you will receive an error message

```
Conv1.forward(M,2,1)
RL1.forward(Conv1.output)
MP1.forward(RL1.output,4,4)
```



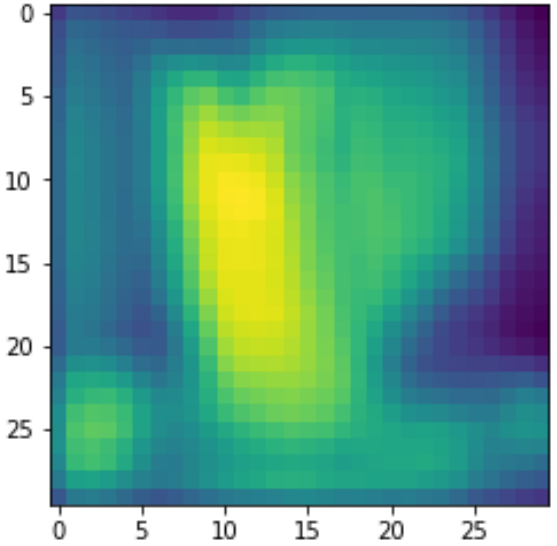

```
Conv2.forward(MP1.output,1,1)
RL2.forward(Conv2.output)
MP2.forward(RL2.output,1,4)
```

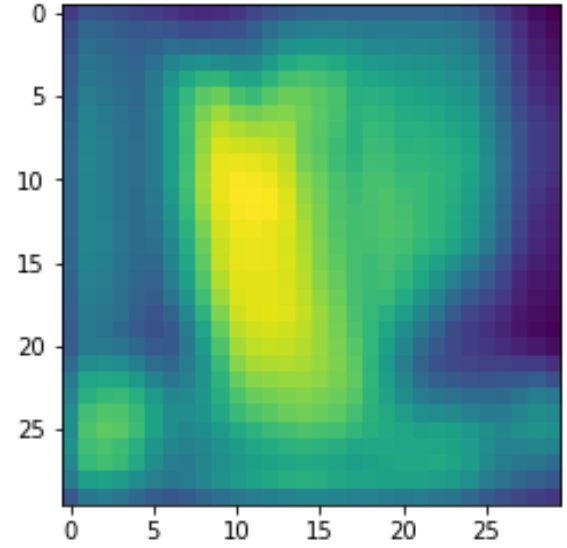



```
.forward(MP2.output,0,3)
orward(Conv3.output)
orward(RL3.output,2,2)
```

→ best way: just run a few convolution layers + backpropagation subsequently
→ choose different values for stride/padding
→ if indices have gotten confused: you will receive an error message

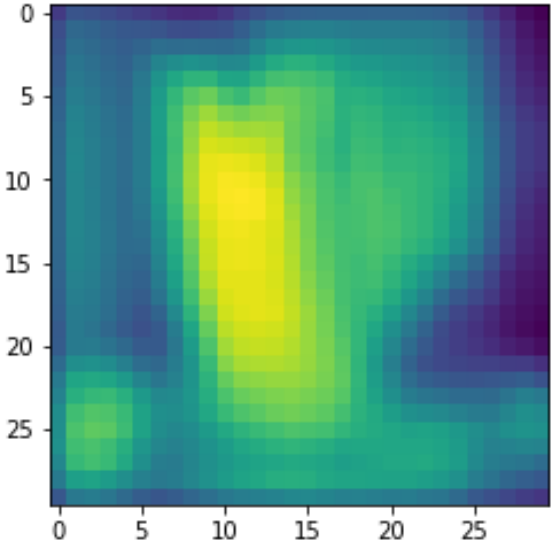
```
Conv1.forward(M,2,1)
RL1.forward(Conv1.output)
MP1.forward(RL1.output,4,4)

Conv2.forward(MP1.output,1,1)
RL2.forward(Conv2.output)
MP2.forward(RL2.output,1,4)

Conv3.forward(MP2.output,0,3)
forward(Conv3.output)
forward(RL3.output,2,2)
```

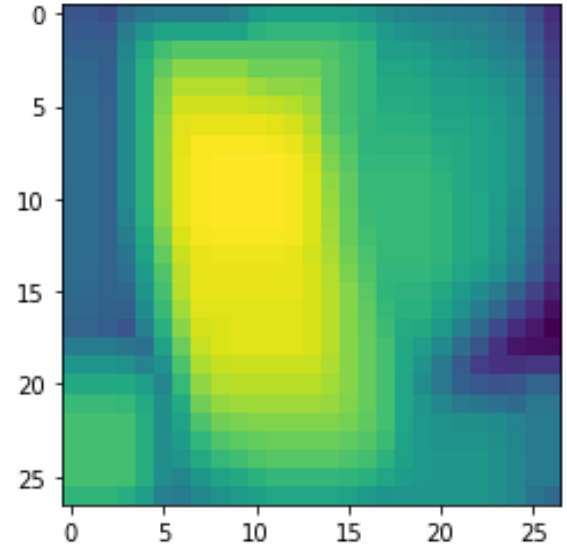


→ best way: just run a few convolution layers + backpropagation subsequently
→ choose different values for stride/padding
→ if indices have gotten confused: you will receive an error message

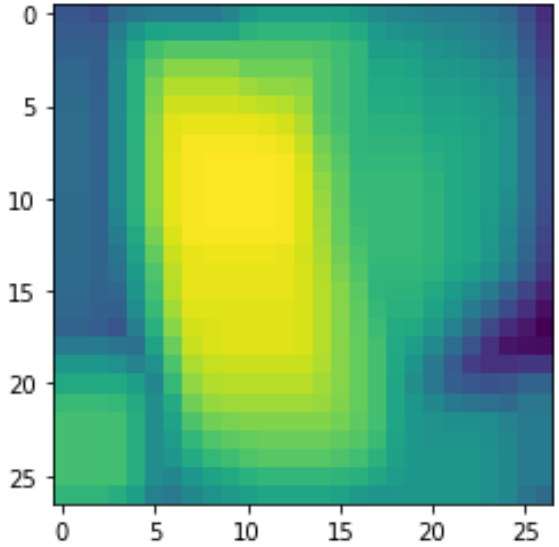
```
Conv1.forward(M,2,1)
RL1.forward(Conv1.output)
MP1.forward(RL1.output,4,4)

Conv2.forward(MP1.output,1,1)
RL2.forward(Conv2.output)
MP2.forward(RL2.output,1,4)

Conv3.forward(MP2.output,0,3)
forward(Conv3.output)
forward(RL3.output,2,2)
```

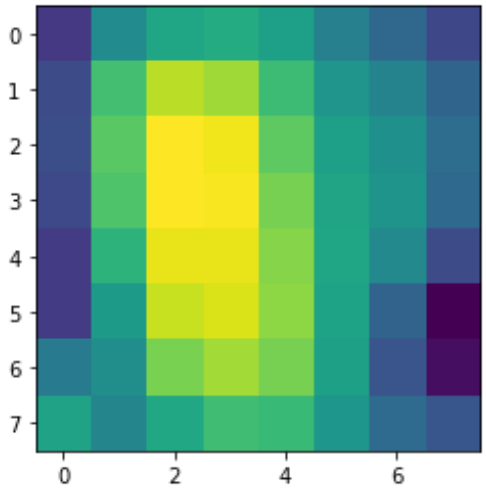


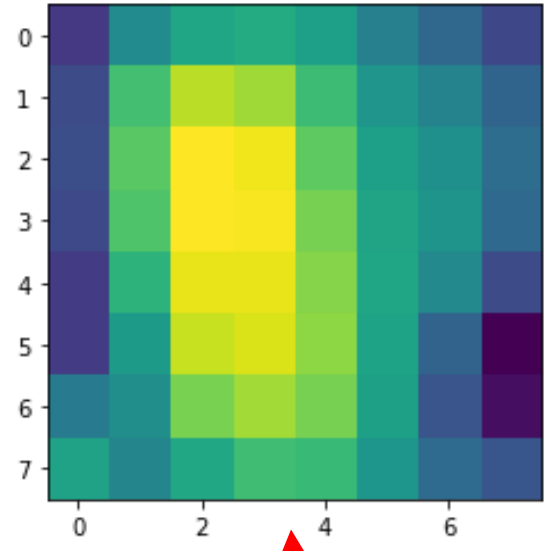
→ best way: just run a few convolution layers + backpropagation subsequently
→ choose different values for stride/padding
→ if indices have gotten confused: you will receive an error message

```
Conv1.forward(M,2,1)
RL1.forward(Conv1.output)
MP1.forward(RL1.output,4,4)

Conv2.forward(MP1.output,1,1)
RL2.forward(Conv2.output)
MP2.forward(RL2.output,1,4)

Conv3.forward(MP2.output,0,3)
RL3.forward(Conv3.output)
3.forward(RL3.output,2,2)
```

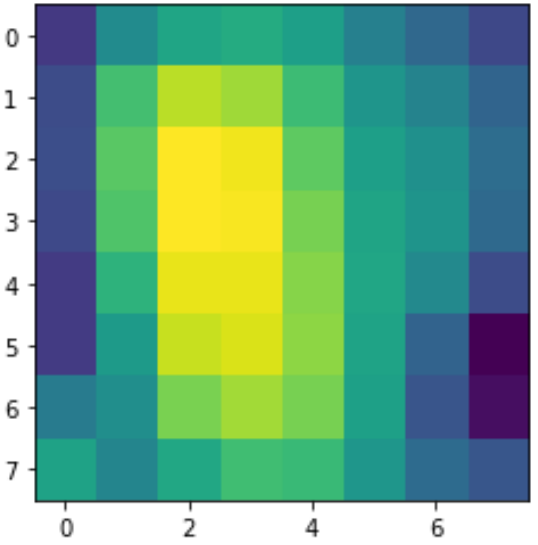


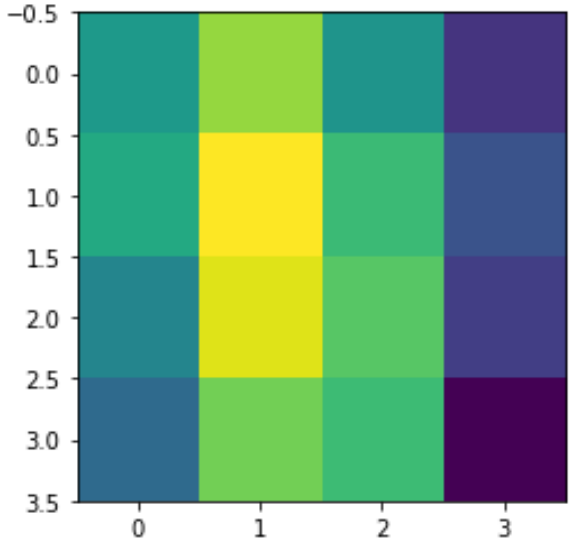
→ best way: just run a few convolution layers + backpropagation subsequently
→ choose different values for stride/padding
→ if indices have gotten confused: you will receive an error message



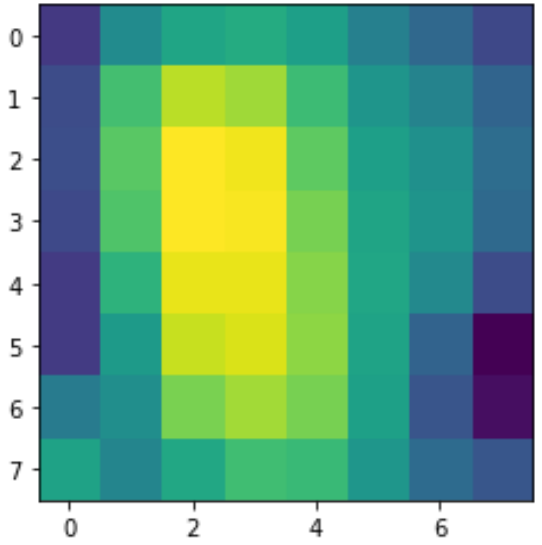

```
v1.forward(M,2,1)
.forward(Conv1.output)
.forward(RL1.output,4,4)

v2.forward(MP1.output,1,1)
.forward(Conv2.output)
.forward(RL2.output,1,4)

v3.forward(MP2.output,0,3)
RL3.forward(Conv3.output)
MP3.forward(RL3.output,2,2)
```



→ best way: just run a few convolution layers + backpropagation subsequently
→ choose different values for stride/padding
→ if indices have gotten confused: you will receive an error message



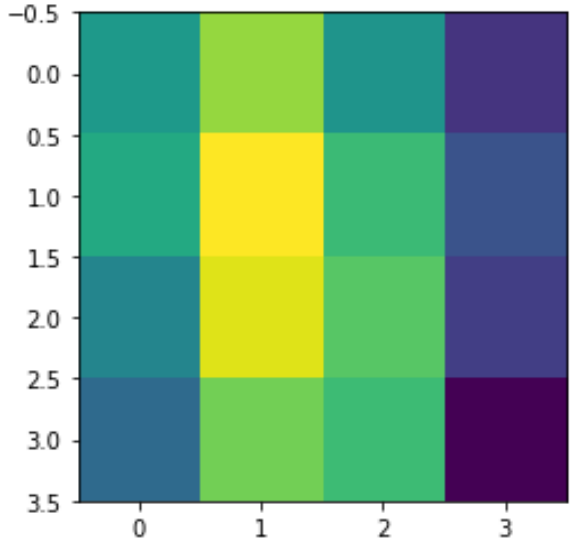

```
.forward(M,2,1)
orward(Conv1.output)
orward(RL1.output,4,4)

.forward(MP1.output,1,1)
orward(Conv2.output)
orward(RL2.output,1,4)

.forward(MP2.output,0,3)
orward(Conv3.output)
MP3.forward(RL3.output,2,2)
```



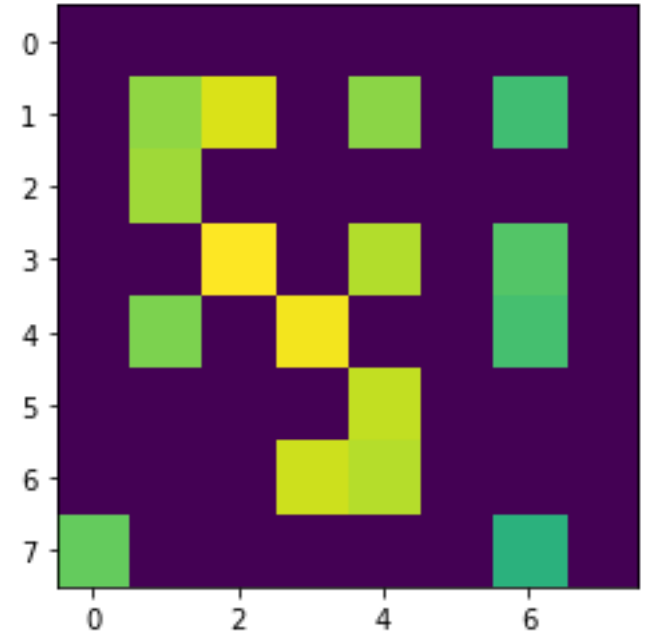
→ best way: just run a few convolution layers + backpropagation subsequently
→ choose different values for stride/padding **now back propagation!**
→ if indices have gotten confused: you will receive an error message

```
MP3.backward(F.dinputs)
RL3.backward(MP3.dinputs)
Conv3.backward(MP3.dinputs)

MP2.backward(Conv3.dinputs)
RL2.backward(MP2.dinputs)
Conv2.backward(RL2.dinputs)

MP1.backward(Conv2.dinputs)
RL1.backward(MP1.dinputs)
Conv1.backward(RL1.dinputs)
```

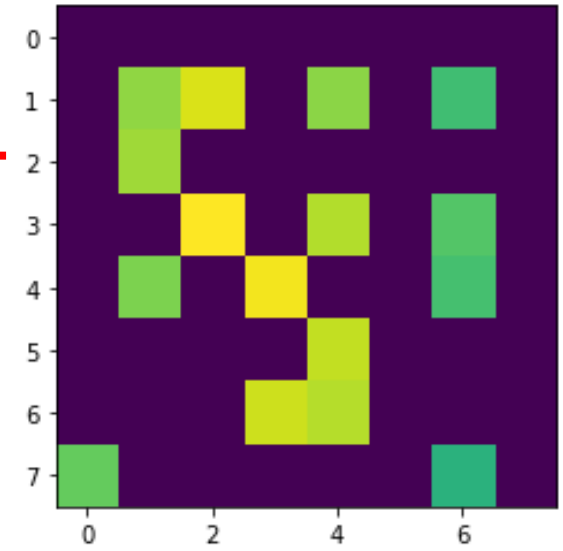
→ best way: just run a few convolution layers + backpropagation subsequently
→ choose different values for stride/padding **now back propagation!**
→ if indices have gotten confused: you will receive an error message



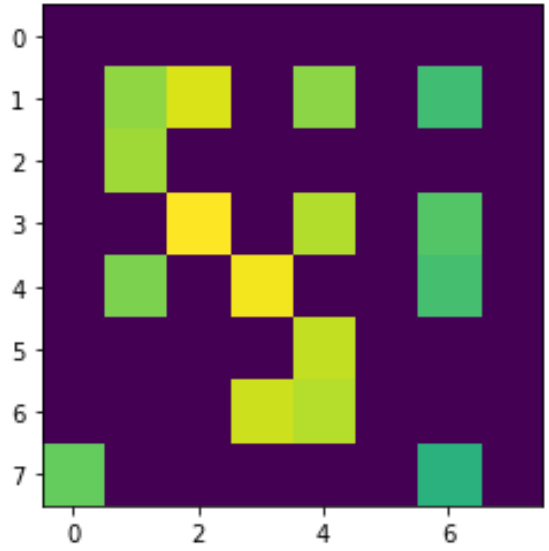

```
MP3.backward(F.dinputs)
RL3.backward(MP3.dinputs)
Conv3.backward(MP3.dinputs)

ckward(Conv3.dinputs)
ckward(MP2.dinputs)
backward(RL2.dinputs)

ckward(Conv2.dinputs)
ckward(MP1.dinputs)
backward(RL1.dinputs)
```



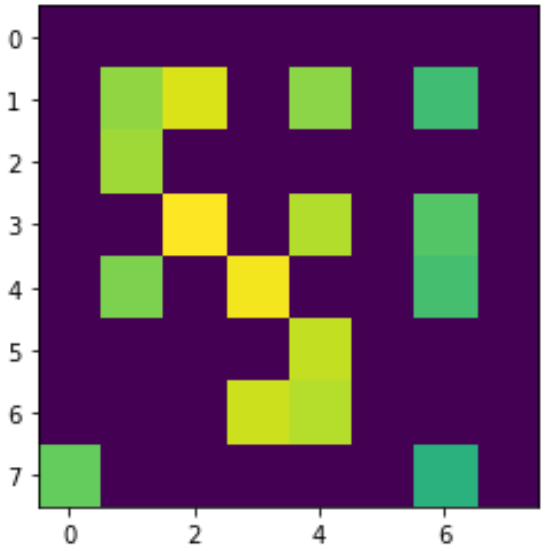
→ best way: just run a few convolution layers + backpropagation subsequently
→ choose different values for stride/padding **now back propagation!**
→ if indices have gotten confused: you will receive an error message



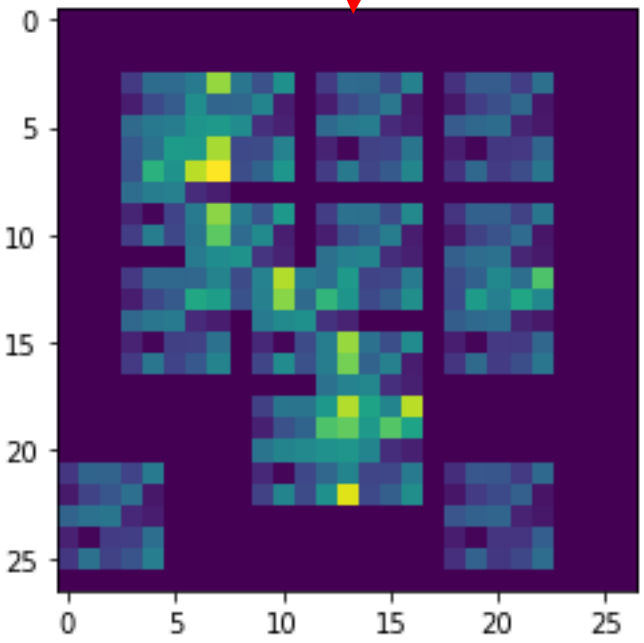

```
MP3.backward(F.dinputs)
RL3.backward(MP3.dinputs)
Conv3.backward(MP3.dinputs)

MP2.backward(Conv3.dinputs)
.backward(MP2.dinputs)
v2.backward(RL2.dinputs)

.backward(Conv2.dinputs)
.backward(MP1.dinputs)
v1.backward(RL1.dinputs)
```



→ best way: just run a few convolution layers + backpropagation subsequently
→ choose different values for stride/padding **now back propagation!**
→ if indices have gotten confused: you will receive an error message



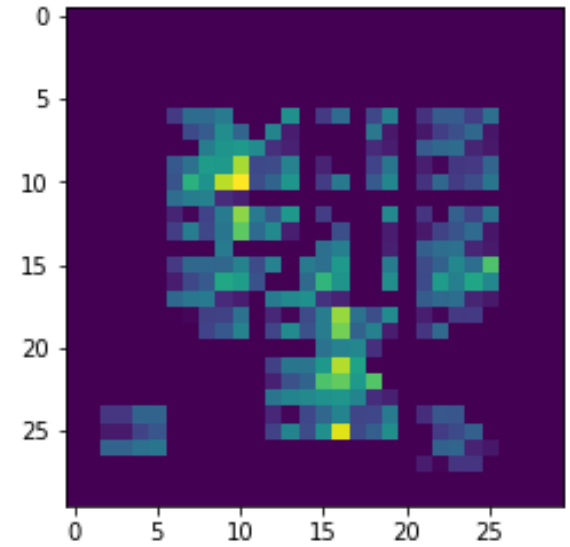

```
MP3.backward(F.dinputs)
RL3.backward(MP3.dinputs)
Conv3.backward(MP3.dinputs)
```

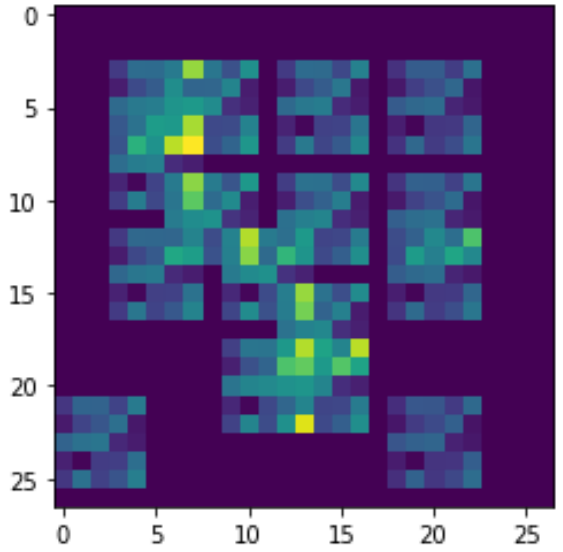



```
kward(Conv3.dinputs)
kward(MP2.dinputs)
ackward(RL2.dinputs)

kward(Conv2.dinputs)
kward(MP1.dinputs)
ackward(RL1.dinputs)
```

→ best way: just run a few convolution layers + backpropagation subsequently
→ choose different values for stride/padding **now back propagation!**
→ if indices have gotten confused: you will receive an error message

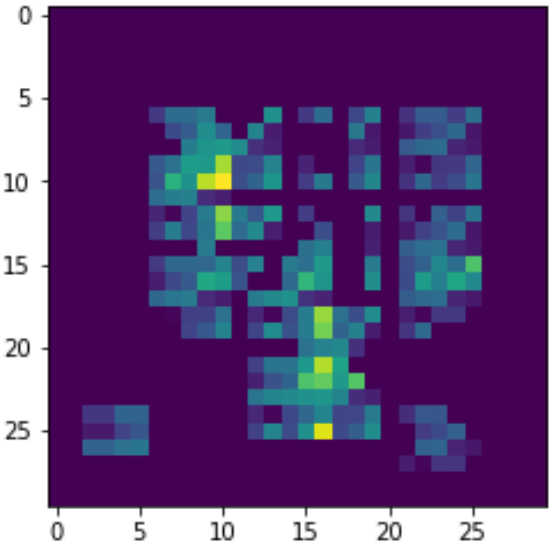
```
MP3.backward(F.dinputs)
RL3.backward(MP3.dinputs)
Conv3.backward(MP3.dinputs)

MP2.backward(Conv3.dinputs)
RL2.backward(MP2.dinputs)
Conv2.backward(RL2.dinputs)

..backward(Conv2.dinputs)
..backward(MP1.dinputs)
nv1.backward(RL1.dinputs)
```

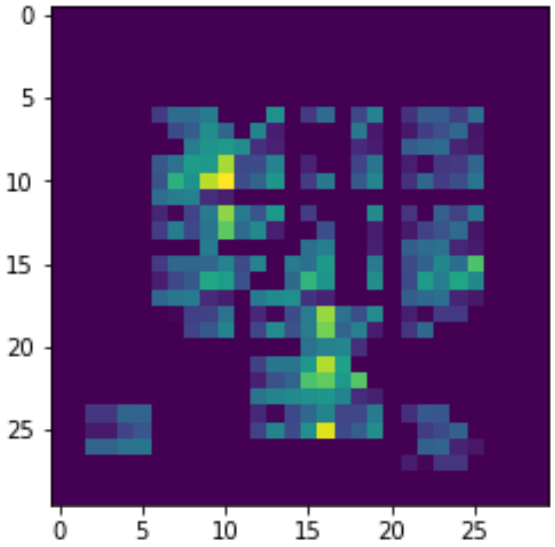


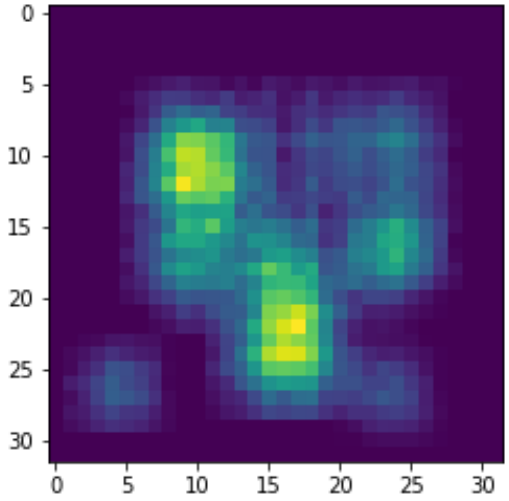
→ best way: just run a few convolution layers + backpropagation subsequently
→ choose different values for stride/padding **now back propagation!**
→ if indices have gotten confused: you will receive an error message

```
MP3.backward(F.dinputs)
RL3.backward(MP3.dinputs)
Conv3.backward(MP3.dinputs)

MP2.backward(Conv3.dinputs)
RL2.backward(MP2.dinputs)
nv2.backward(RL2.dinputs)

l.backward(Conv2.dinputs)
l.backward(MP1.dinputs)
nv1.backward(RL1.dinputs)
```

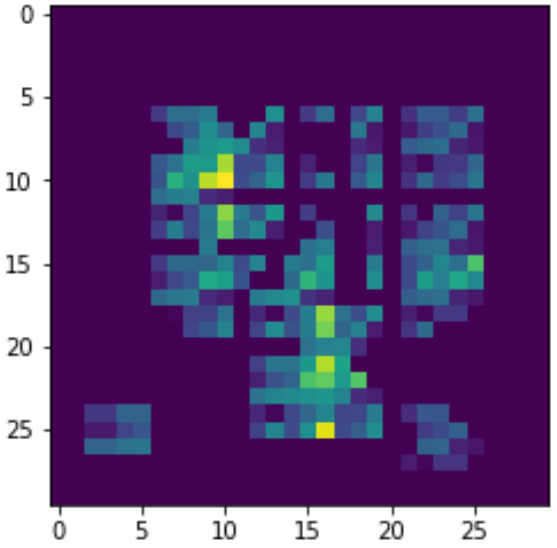


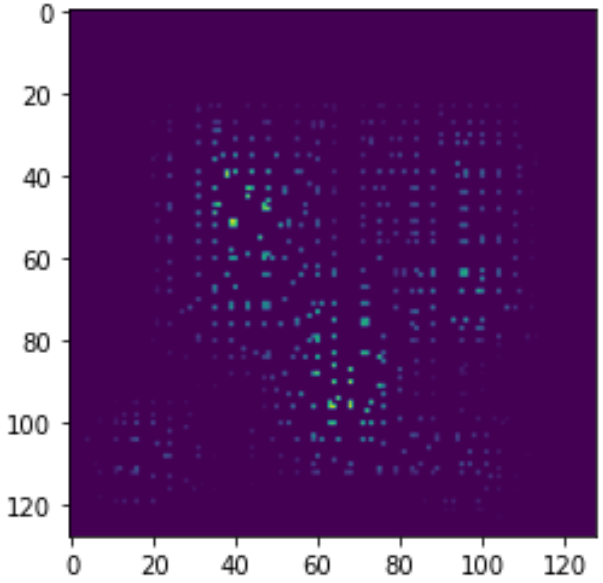
→ best way: just run a few convolution layers + backpropagation subsequently
→ choose different values for stride/padding **now back propagation!**
→ if indices have gotten confused: you will receive an error message



```
MP3.backward(F.dinputs)
RL3.backward(MP3.dinputs)
Conv3.backward(MP3.dinputs)

MP2.backward(Conv3.dinputs)
RL2.backward(MP2.dinputs)
Conv2.backward(RL2.dinputs)

MP1.backward(Conv2.dinputs)
RL1.backward(MP1.dinputs)
Conv1.backward(RL1.dinputs)
```

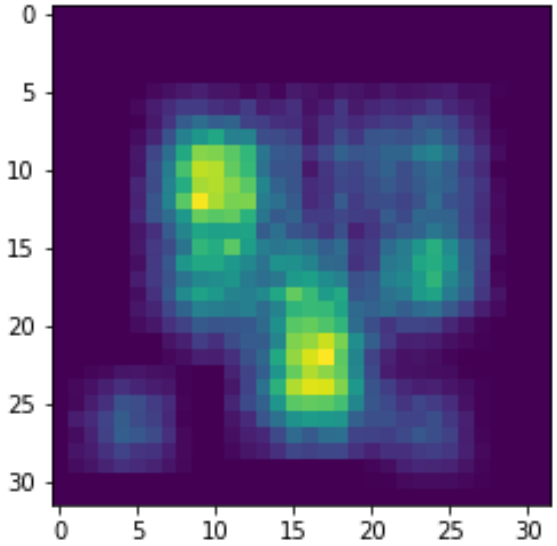


→ best way: just run a few convolution layers + backpropagation subsequently
→ choose different values for stride/padding **now back propagation!**
→ if indices have gotten confused: you will receive an error message

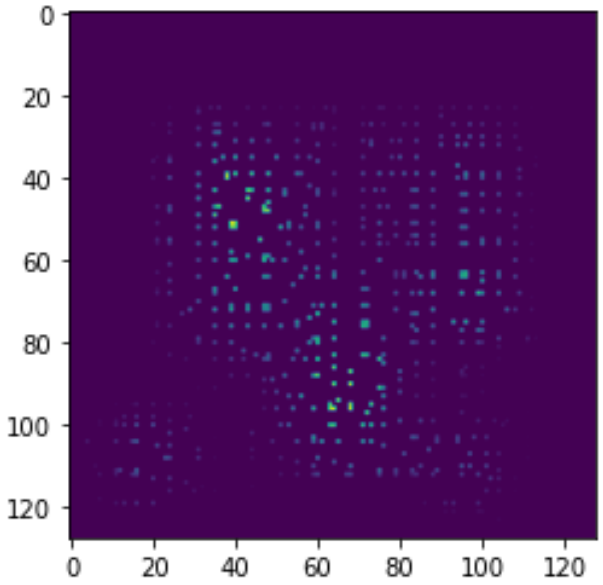



```
backward(F.dinputs)
backward(MP3.dinputs)
.backward(MP3.dinputs)

backward(Conv3.dinputs)
backward(MP2.dinputs)
.backward(RL2.dinputs)
```

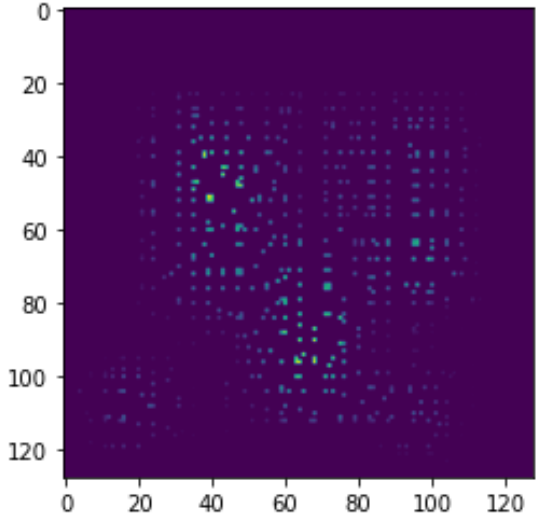
```
MP1.backward(Conv2.dinputs)
RL1.backward(MP1.dinputs)
Conv1.backward(RL1.dinputs)
```



→ best way: just run a few convolution layers + backpropagation subsequently
→ choose different values for stride/padding **now back propagation!**
→ if indices have gotten confused: you will receive an error message



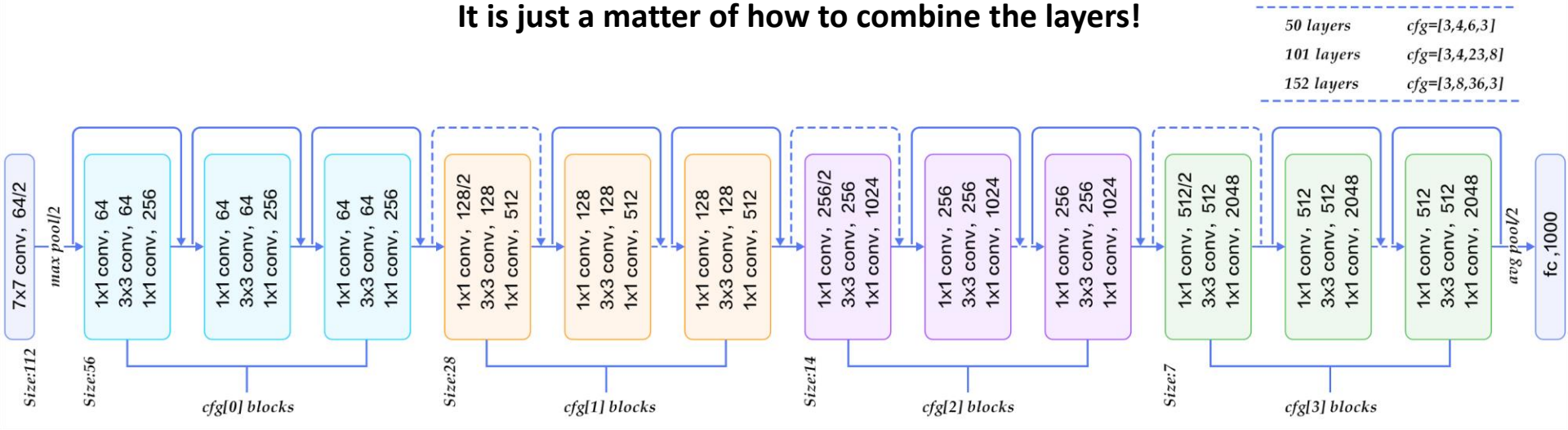

```
backward(F.dinputs)
backward(MP3.dinputs)
.backward(MP3.dinputs)

backward(Conv3.dinputs)
backward(MP2.dinputs)
.backward(RL2.dinputs)

MP1.backward(Conv2.dinputs)
RL1.backward(MP1.dinputs)
Conv1.backward(RL1.dinputs)
```

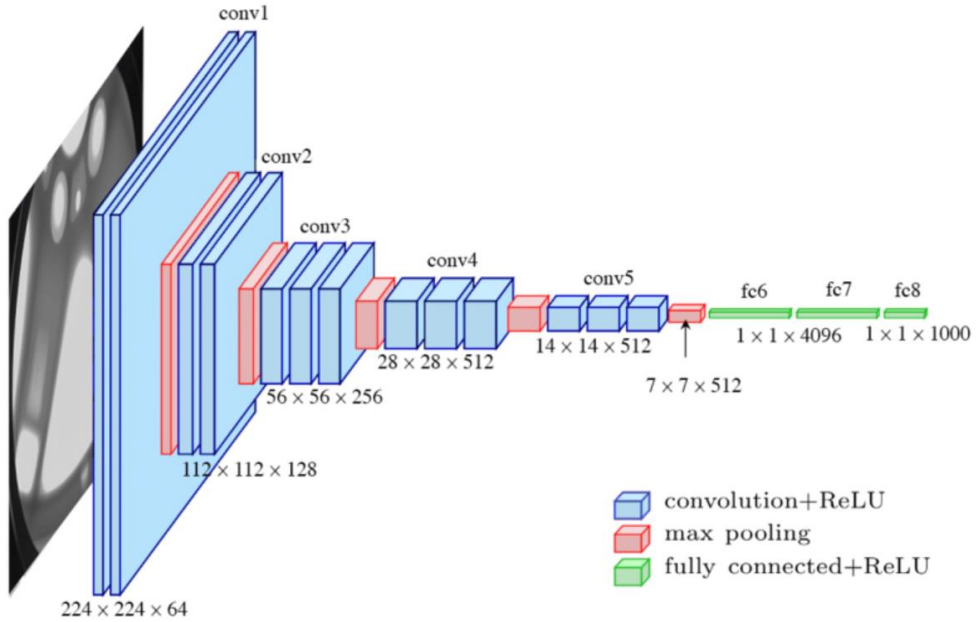



**It all seems to work!**

# We now have the ingredients to build any common CNN!

**It is just a matter of how to combine the layers!**



LeNet: - Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998

- one of the 1st CNN that was able to categorize images

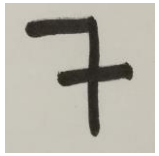
- MNIST data set

- only **seven layers** in total

- modern CNNs (google, ResNet etc have **100 or more** layers)

→ since we have limited computational resources and
our CNN is not optimized regarding speed → let us built LeNet and see how that works

LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**



**Conv1: 6 filter, size 5x5, padding = 0, stride = 1**

***tanh***

**average pool 1: size 2x2, stride = 2**

**Conv2: 16 filter, size 5x5, padding = 0, stride = 1**

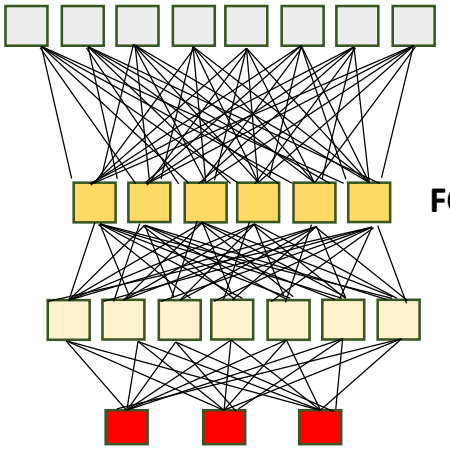
***tanh***

**average pool 2: size 2x2, stride = 2**

**Conv3: 120 filter, size 5x5, padding = 2, stride = 3**

***tanh***

**flat**

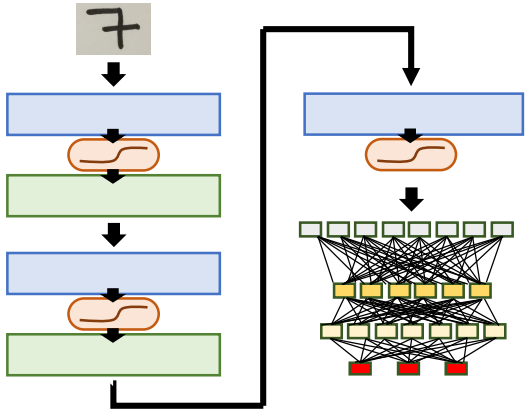


**FC dense, N = 84**

- *tanh* as activation function
- three different filter numbers
- from Conv1 → Conv2 not all the channels are combined
- average pool

LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

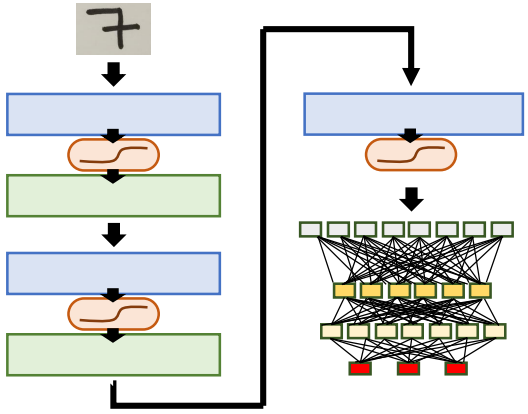
- *tanh* as activation function
- three different filter numbers
- from Conv1 → Conv2 not all the channels are combined
- average pool




```
class Tanh:

    def forward(self, M):

        tanh        = np.tanh(M)
        self.output = np.nan_to_num(tanh)
        self.inputs = np.nan_to_num(tanh) #needed for back prop

    def backward(self, dvalues):

        tanh         = self.inputs
        deriv        = 1 - tanh**2
        deriv        = np.nan_to_num(deriv)
        self.dinputs = np.multiply(deriv, dvalues)
```

LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

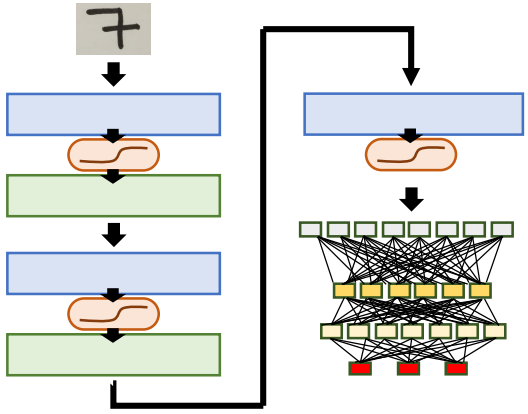
- *tanh* as activation function
- three different filter numbers
- from Conv1 → Conv2 not all the channels are combined
- average pool



```
class ConvLayer:

    def __init__(self , xKernShape = 3, yKernShape = 3,\
                                                Kernnumber = 10):

        self.xKernShape = xKernShape
        self.yKernShape = yKernShape
        self.Kernnumber = Kernnumber

        self.weights    = 10*np.random.randn(xKernShape,\
                                        yKernShape,Kernnumber)

        self.biases     = np.zeros((1, self.Kernnumber))
```

LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

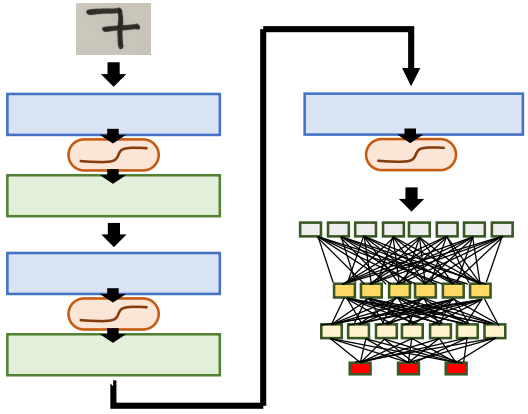
- *tanh* as activation function
- three different filter numbers
- from Conv1 → Conv2 not all the channels are combined
- average pool



```
class ConvLayer:

        def __init__(self, xKernShape = 3, yKernShape = 3, Kernnumber = 10):

                self.xKernShape = xKernShape
                self.yKernShape = yKernShape
                self.Kernnumber = Kernnumber

                self.weights    = 10*np.random.randn(xKernShape,yKernShape,Kernnumber)

                self.biases     = np.zeros((1, self.Kernnumber))
```

```
Conv1   = My_ANN.ConvLayer(5,5,6)
Conv2   = My_ANN.ConvLayer(5,5,16)
Conv3   = My_ANN.ConvLayer(5,5,120)
```

LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**



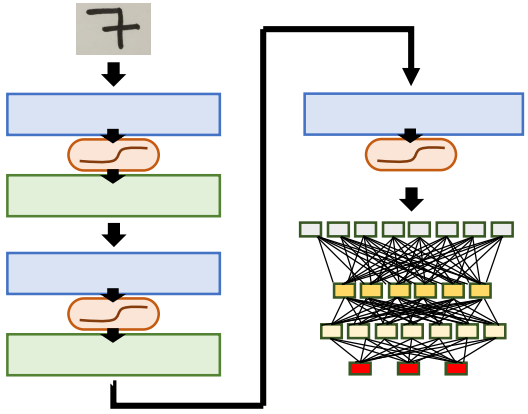
- *tanh* as activation function
- three different filter numbers
- from Conv1 → Conv2 not all the channels are combined
- average pool

```
class ConvLayer:
        ...
        def forward(self, ...):
                ...

                s = np.multiply(current_slice, W[:,:,k])

               output[x, y, c, k, i] = np.sum(s) +\
                                                b[0,k].astype(float)

                ...
```

usually, each channel **c** gets convolved with each filter **k**

LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

- *tanh* as activation function
- three different filter numbers
- from Conv1 → Conv2 not all the channels are combined
- average pool



here:

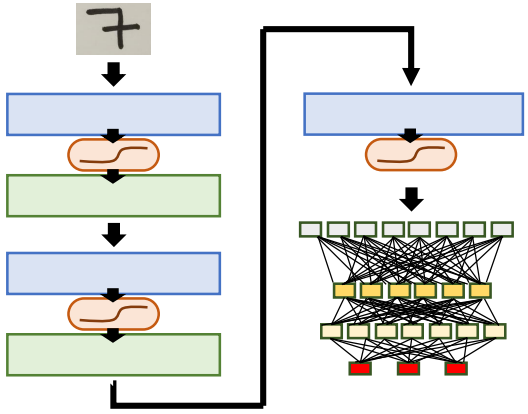
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | X | | | | X | X | X | | | X | X | X | X | | X | X |
| 1 | X | X | | | | X | X | X | | | X | X | X | X | | X |
| 2 | X | X | X | | | | X | X | X | | | X | | X | X | X |
| 3 | | X | X | X | | | X | X | X | X | | | X | | X | X |
| 4 | | | X | X | X | | | X | X | X | X | | X | X | | X |
| 5 | | | | X | X | X | | | X | X | X | X | | X | X | X |

creating a filter matrix:

```
if numChans == 6 & NK == 16:
        filt = np.array([[1,0,0,0,1,1,1,0,0,1,1,1,1,0,1,1],\
                         [1,1,0,0,0,1,1,1,0,0,1,1,1,1,0,1],\
                         [1,1,1,0,0,0,1,1,1,0,0,1,0,1,1,1],\
                         [0,1,1,1,0,0,1,1,1,1,0,0,1,0,1,1],\
                         [0,0,1,1,1,0,0,1,1,1,1,0,1,1,0,1],\
                         [0,0,0,1,1,1,0,0,1,1,1,1,0,1,1,1]])
else:
        filt = np.ones([numChans,NK])

self.filt = filt
```

LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

- *tanh* as activation function
- three different filter numbers
- from Conv1 → Conv2 not all the channels are combined
- average pool



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | X | | | | X | X | X | | | X | X | X | X | | X | X |
| 1 | X | X | | | | X | X | X | | | X | X | X | X | | X |
| 2 | X | X | X | | | | X | X | X | | | X | | X | X | X |
| 3 | | X | X | X | | | X | X | X | X | | | X | | X | X |
| 4 | | | X | X | X | | | X | X | X | X | | X | X | | X |
| 5 | | | | X | X | X | | | X | X | X | X | | X | X | X |

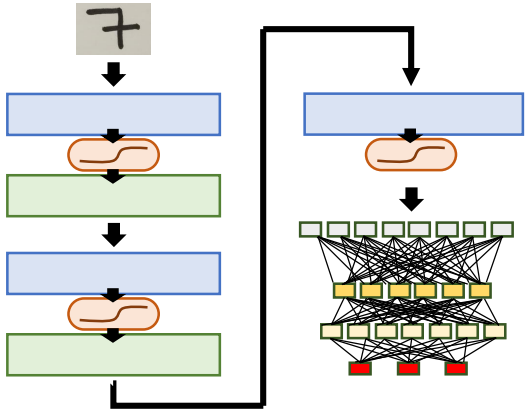
```
filt = np.array([[1,0,0,0,1,1,1,0,0,1,1,1,1,0,1,1],\
                 [1,1,0,0,0,1,1,1,0,0,1,1,1,1,0,1],\
                 [1,1,1,0,0,0,1,1,1,0,0,1,0,1,1,1],\
                 [0,1,1,1,0,0,1,1,1,1,0,0,1,0,1,1],\
                 [0,0,1,1,1,0,0,1,1,1,1,0,1,1,0,1],\
                 [0,0,0,1,1,1,0,0,1,1,1,1,0,1,1,1]])
```

creating a filter matrix: → performing operations in forward and backward part only,

```
if filt[c,k] == 1:
```

LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

- *tanh* as activation function
- three different filter numbers
- from Conv1 → Conv2 not all the channels are combined
- average pool

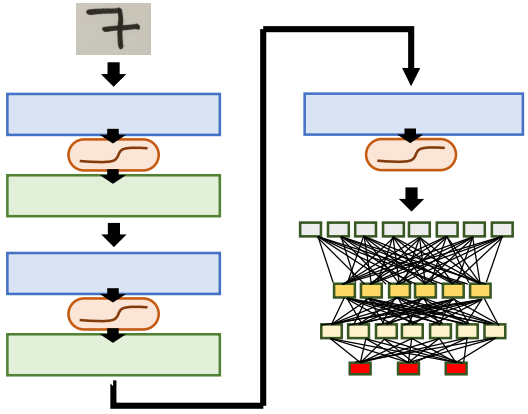


like max pool, only a few differences:

```
def forward(self, ...):

...

#actual average pool
slice_mean          = float(current_slice.mean())
output[x, y, c, i] = slice_mean

...
```

LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

- *tanh* as activation function
- three different filter numbers
- from Conv1 → Conv2 not all the channels are combined
- average pool

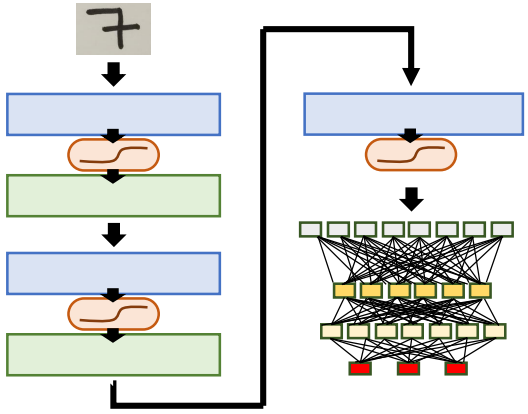


like max pool, only a few differences:

```
def backward(self, ...):

Ones     = Ones/xK/yK
...


dinputs[sx,sy,c,i]  += Ones[sx,sy,c,i]*dvalues[x,y,c,i]

...
```

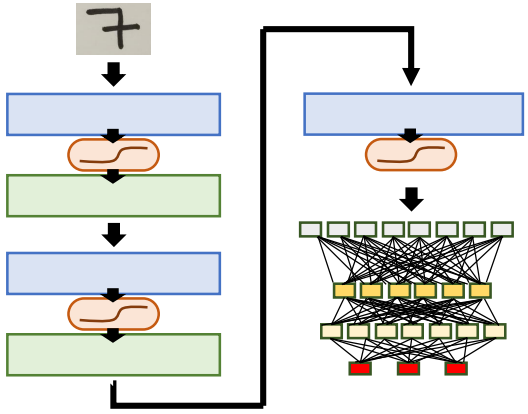
normalization from the derivative of the average

LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**



- ANN_MMH.py
  - ConvLayer
  - Average_Pool
  - Max_Pool
  - Sigmoid
  - Tanh
  - Activation_ReLU
  - Flat
  - Layer_Dense
  - Activation_Softmax
  - Loss
  - Loss_CategoricalCrossEntropy
  - Activation_Softmax_Loss_CategoricalCrossentropy
  - Optimizer_SGD

LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

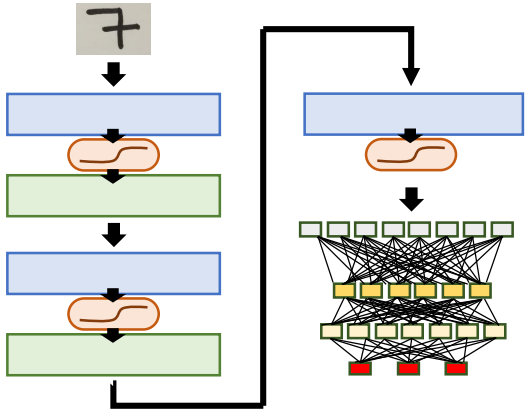


The MNIST data set:

```
#pip install keras
#pip install tensorflow

from keras.datasets import mnist

(train_x, train_y), (test_X, test_y) = mnist.load_data()
```

already a sparse vector

LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

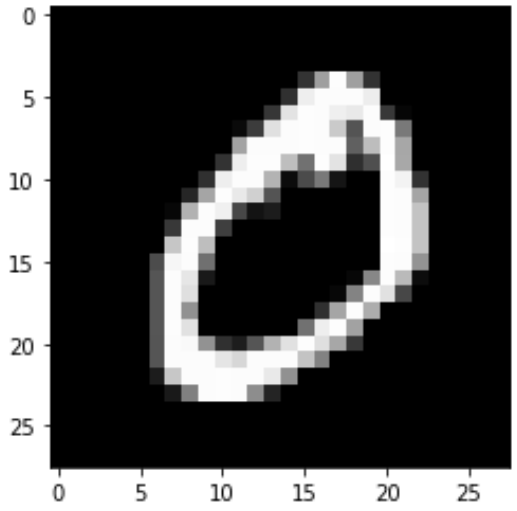


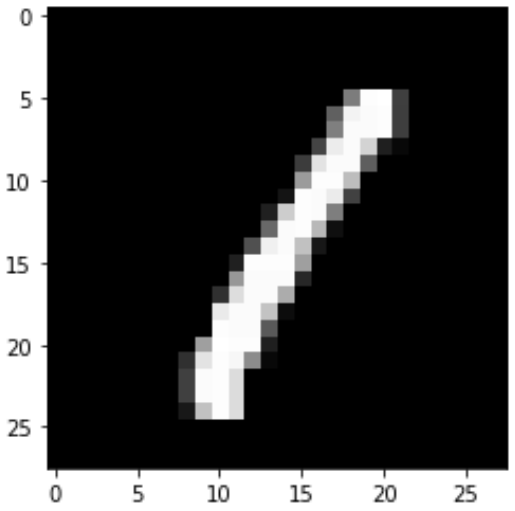
The MNIST data set:

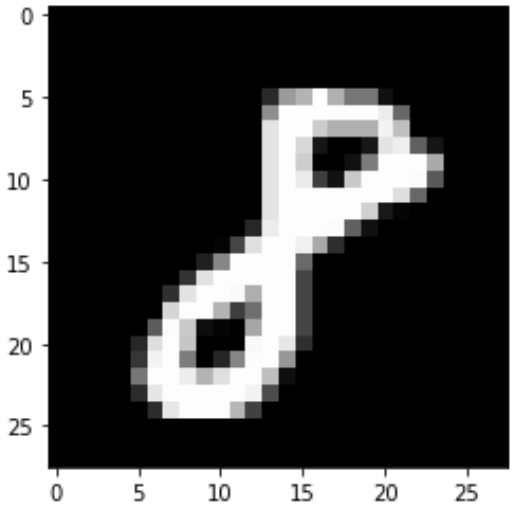
```
#pip install keras
#pip install tensorflow
```

```
from keras.datasets import mnist

(train_x, train_y), (test_x, test_y) = mnist.load_data()
```

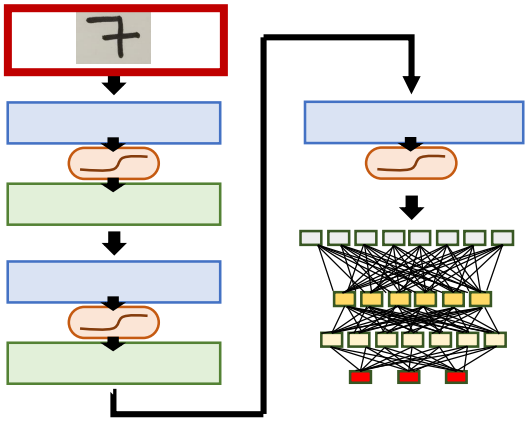
```
plt.imshow(train_x[1,:,:], cmap = 'gray')
```

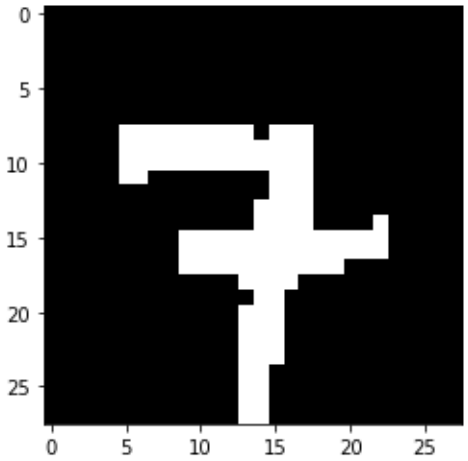




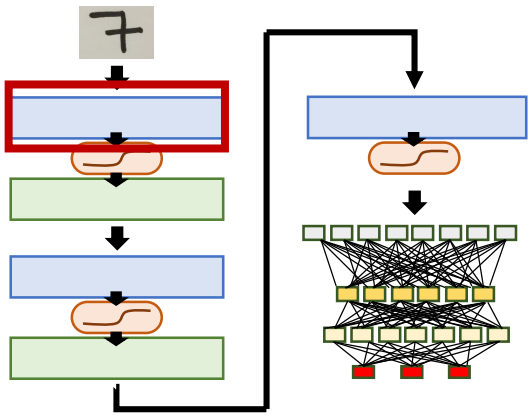


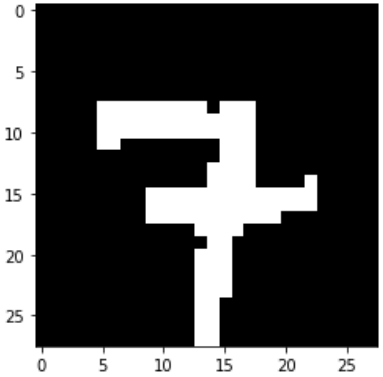
LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

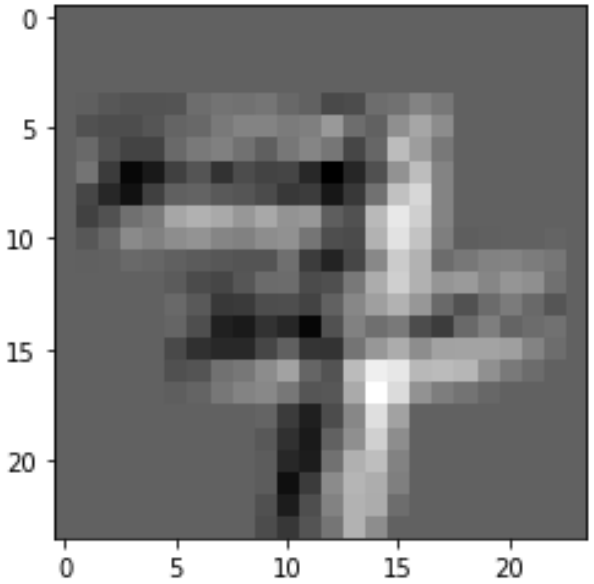




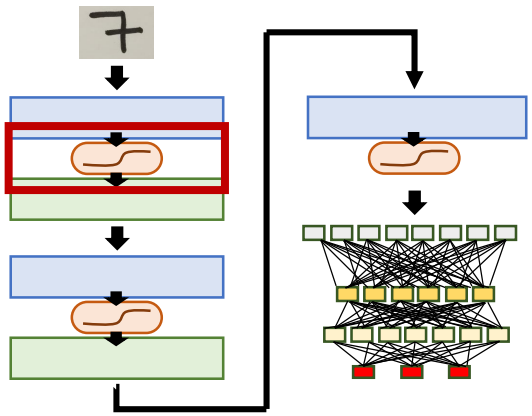
LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

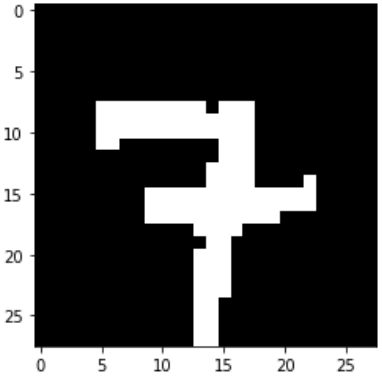


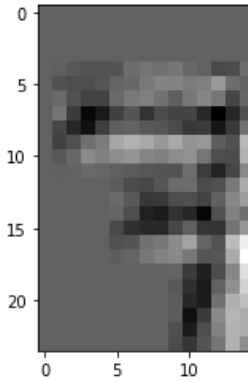


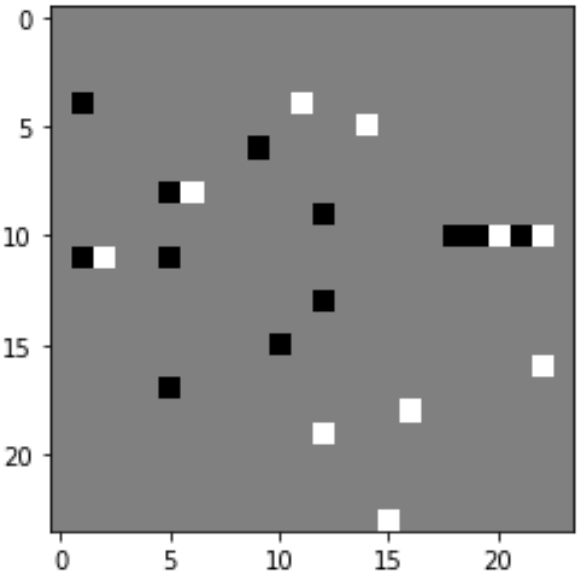



LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

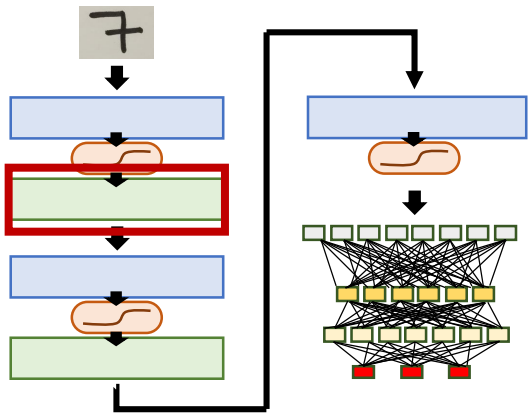


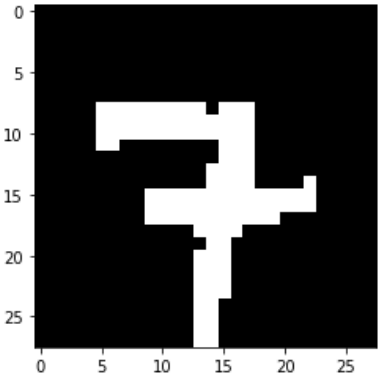


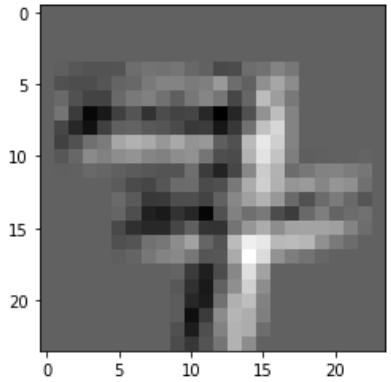



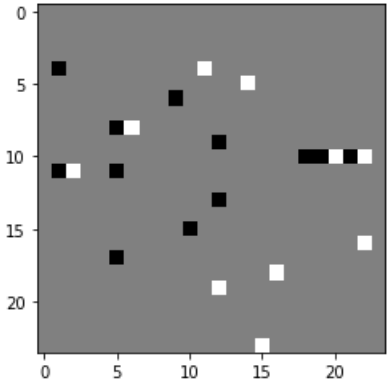



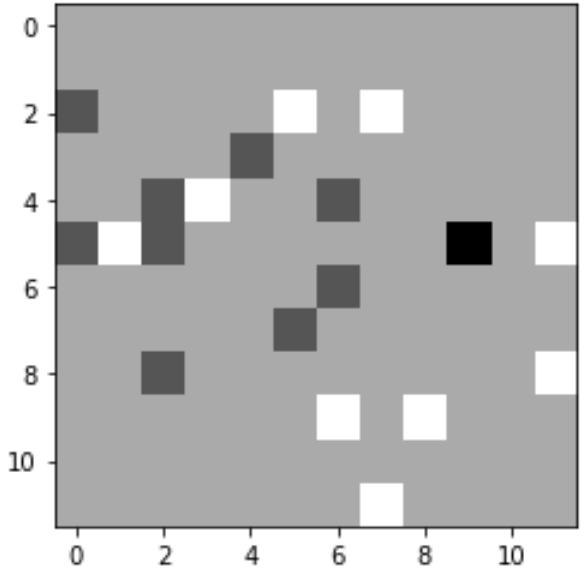
LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**



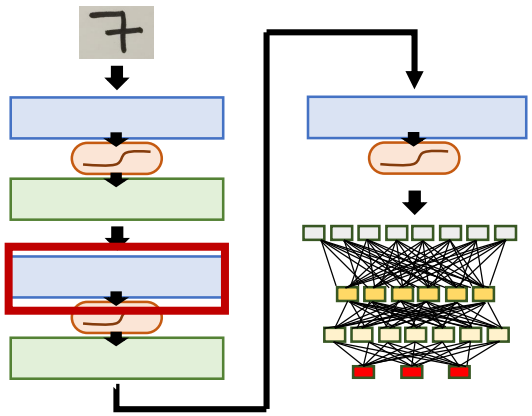


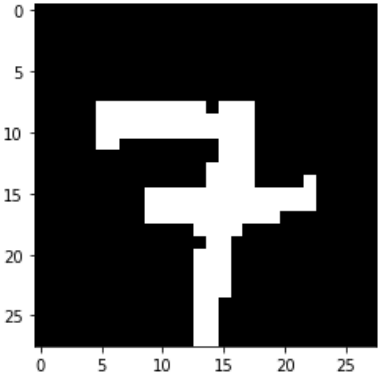



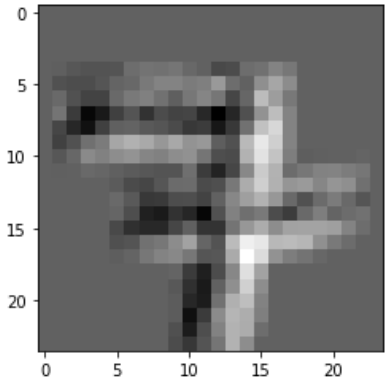



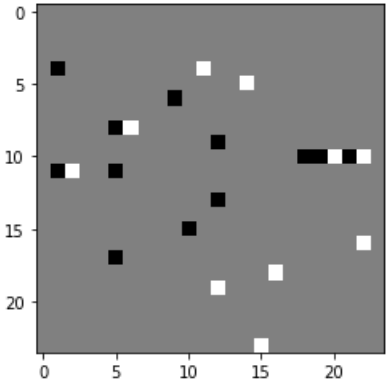



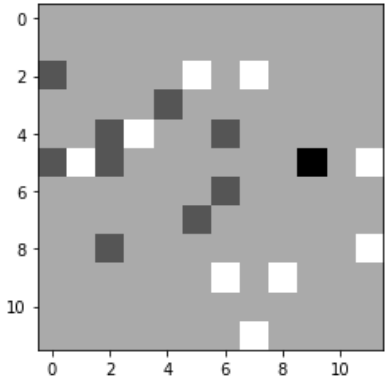
LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

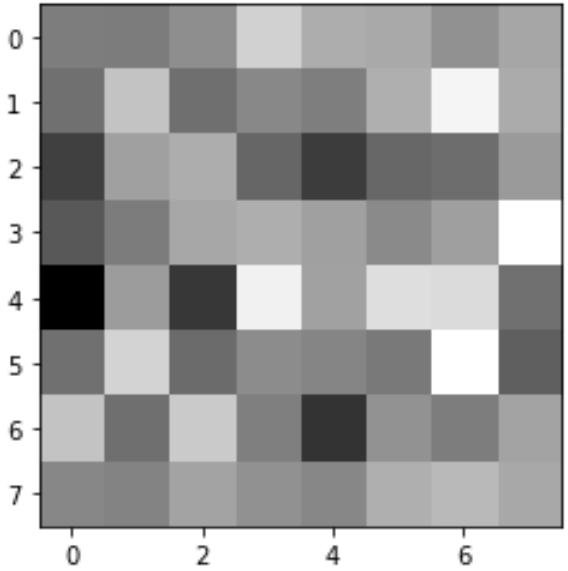




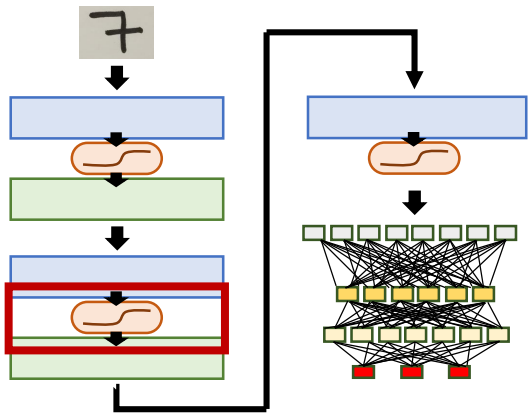


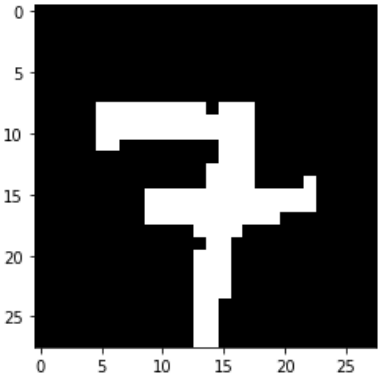


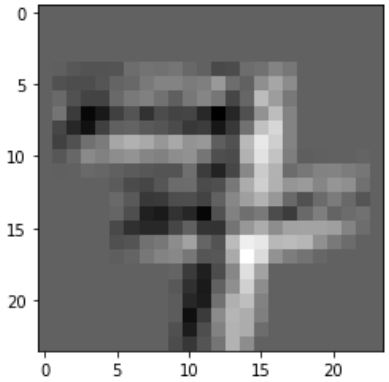


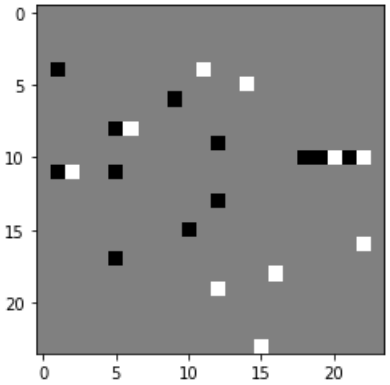


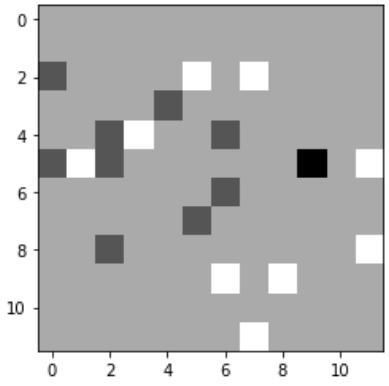
LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

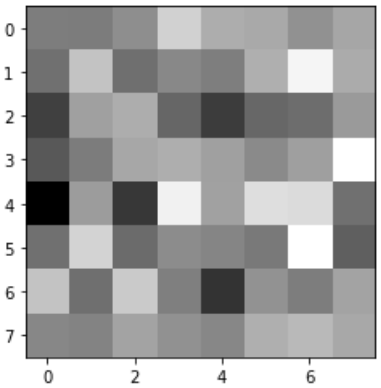


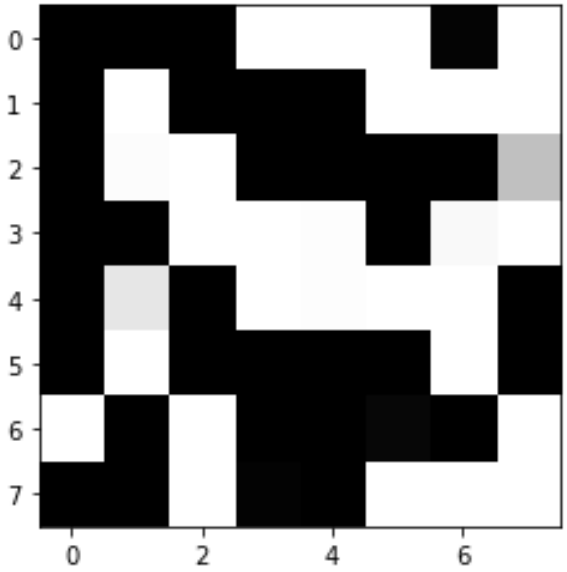




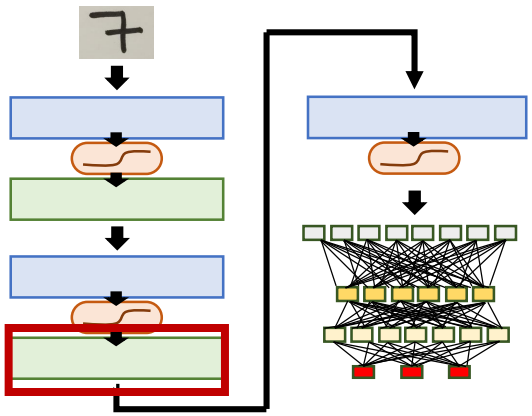


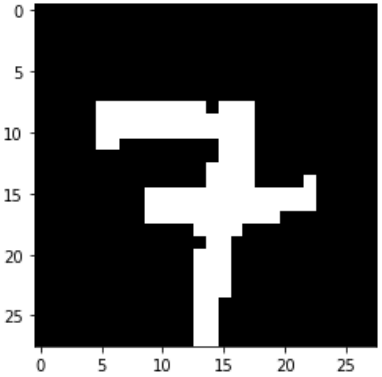


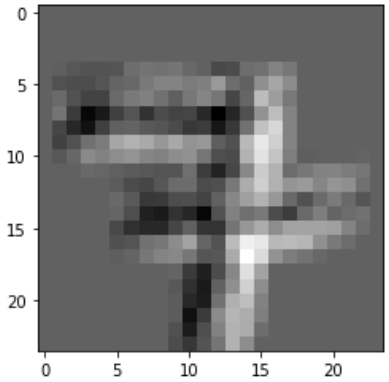


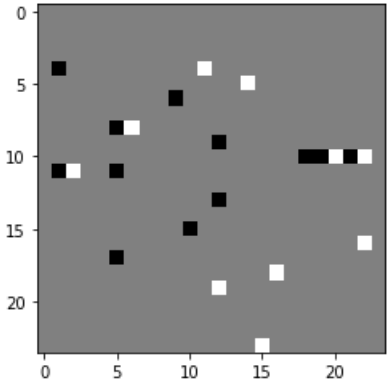


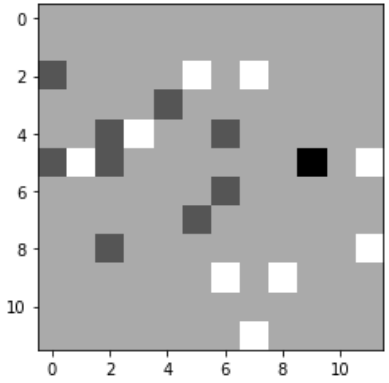
LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

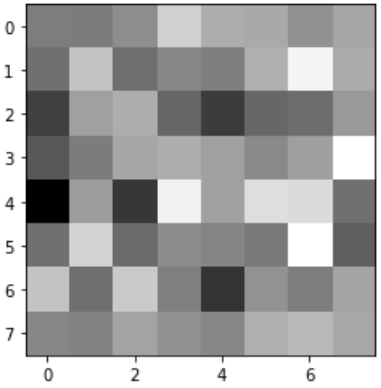


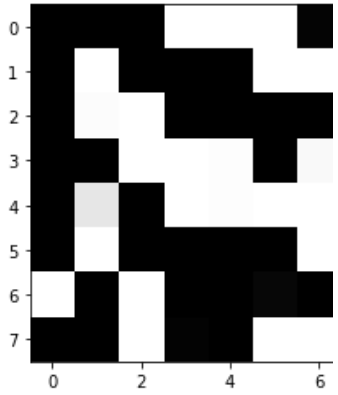


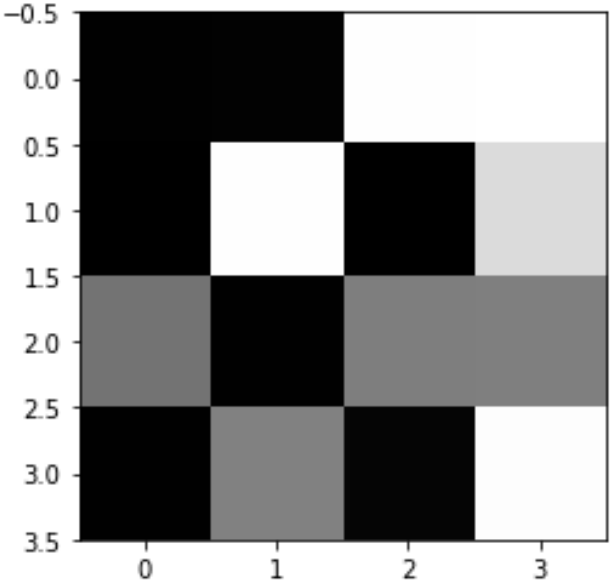




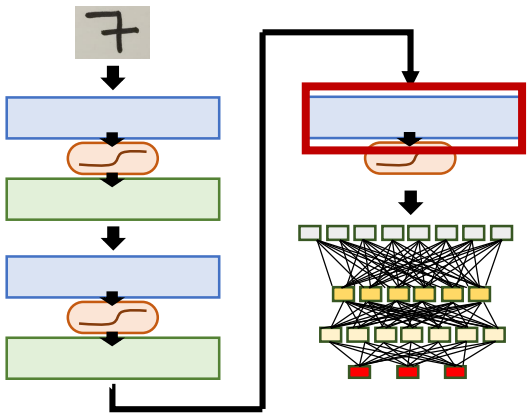


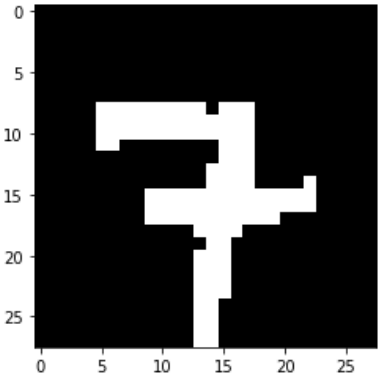


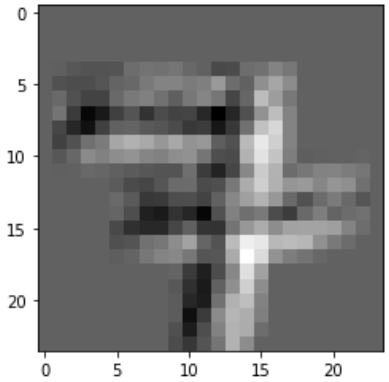


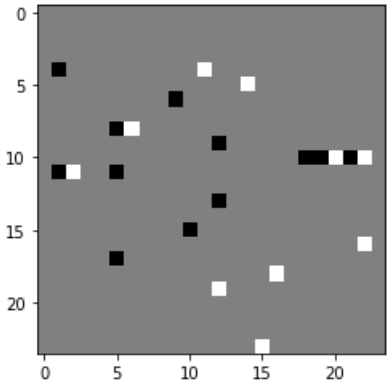


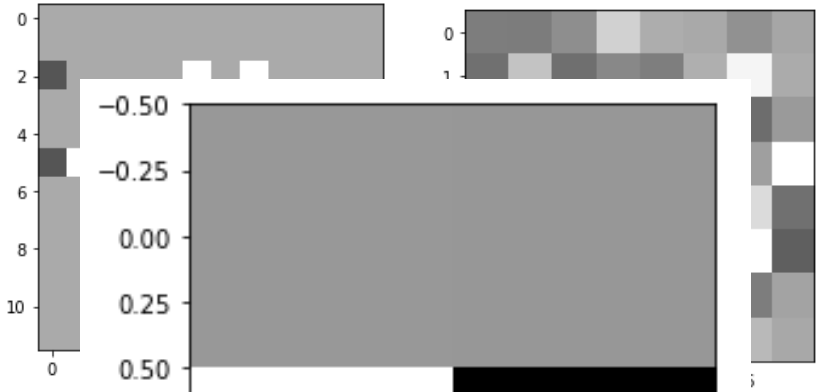
LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**





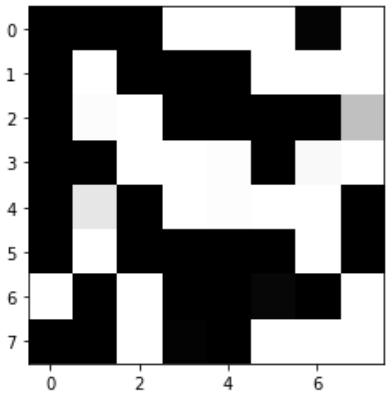


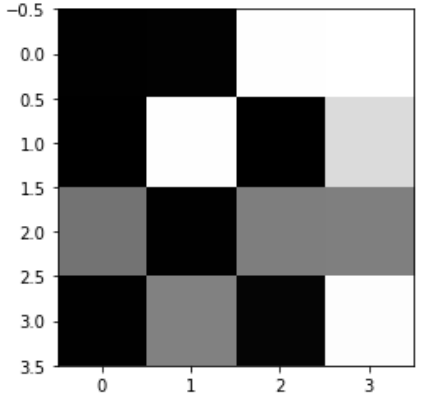




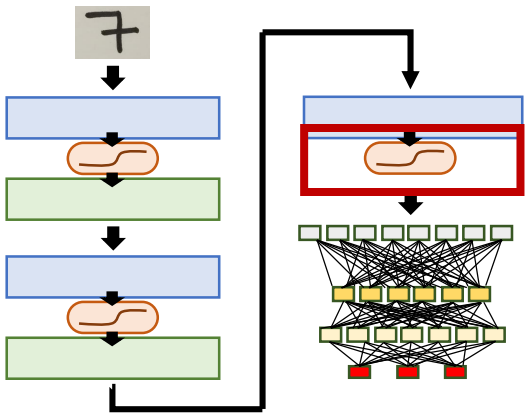


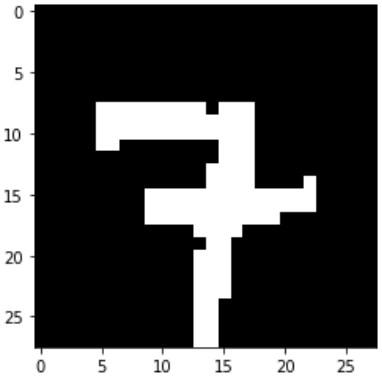


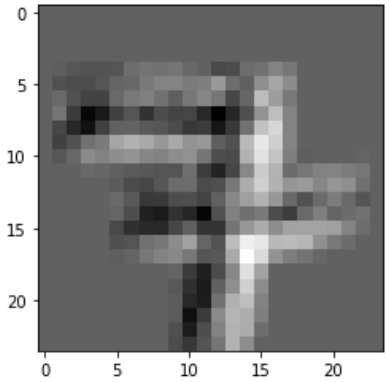


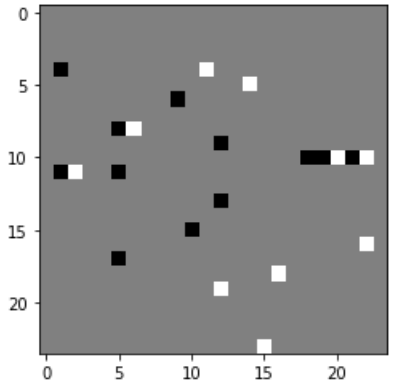


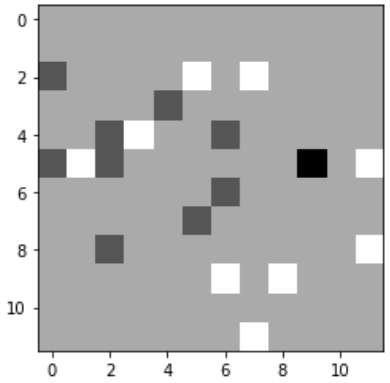
LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

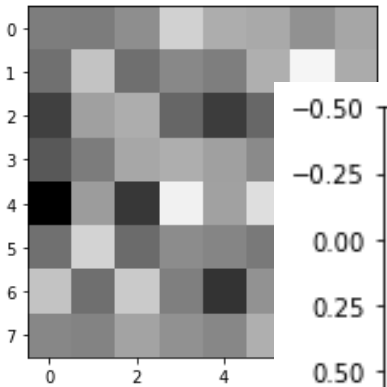


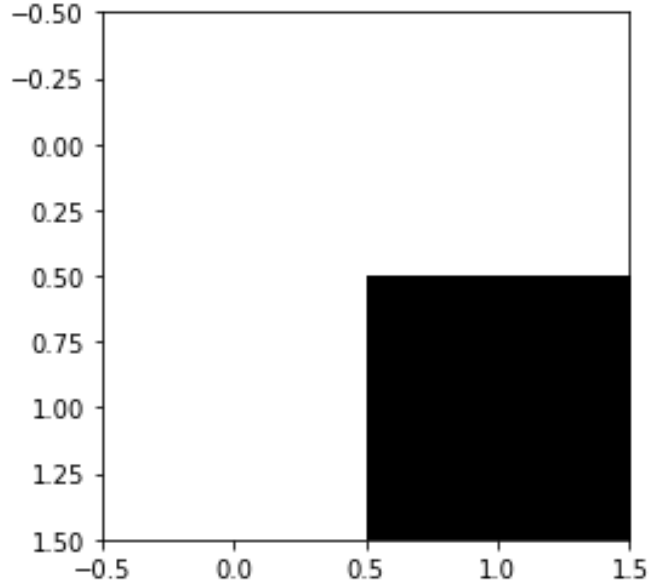


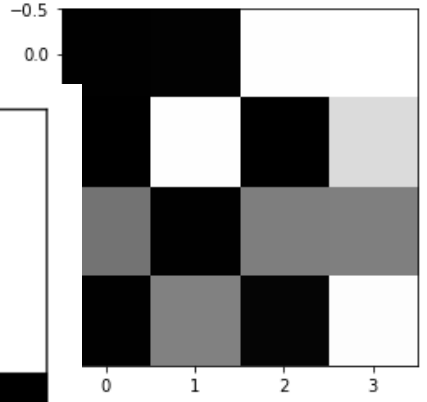


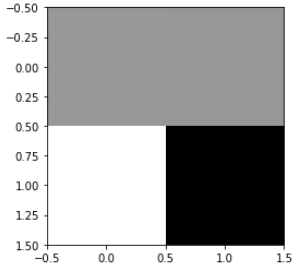






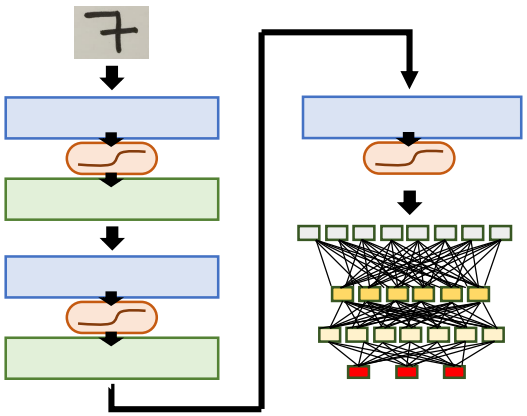


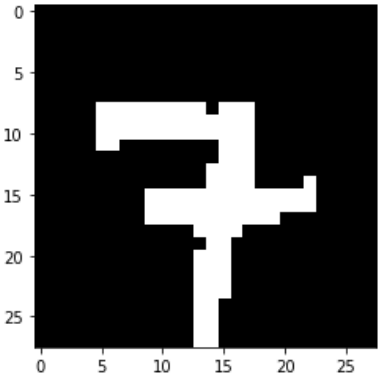


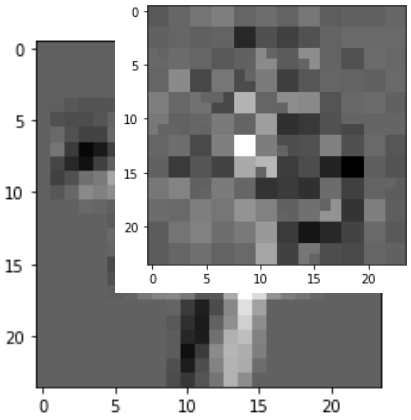


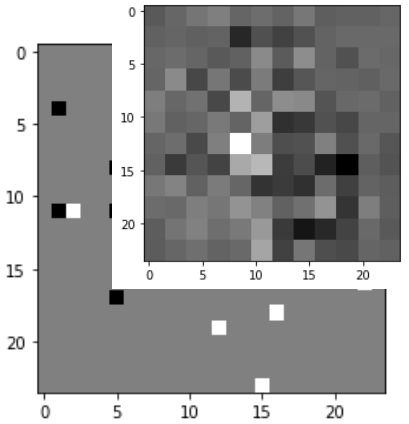
LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

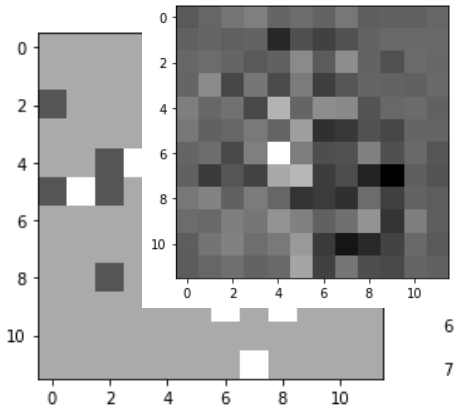
**now back propagation!**

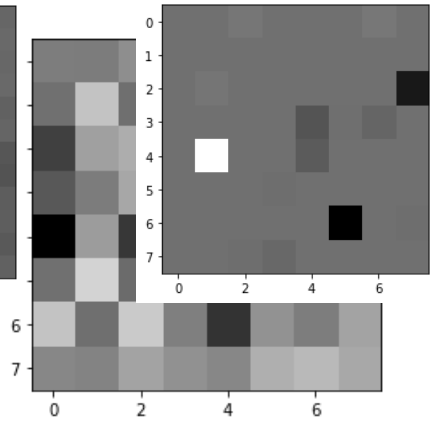


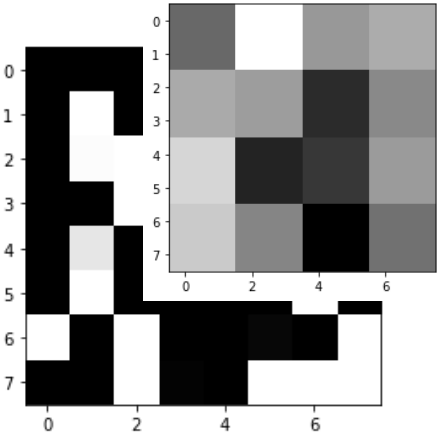


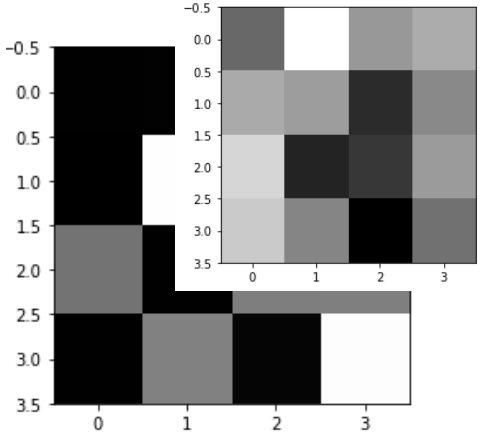


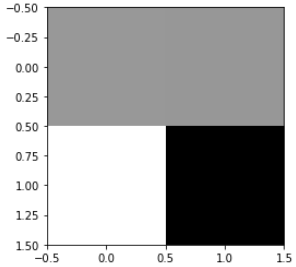


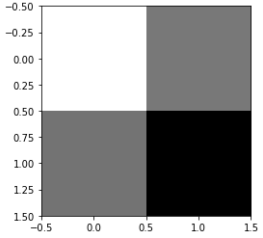


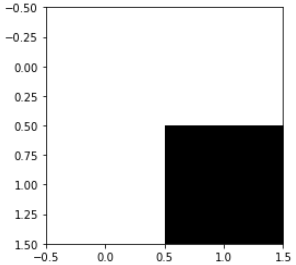


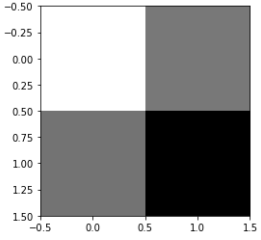








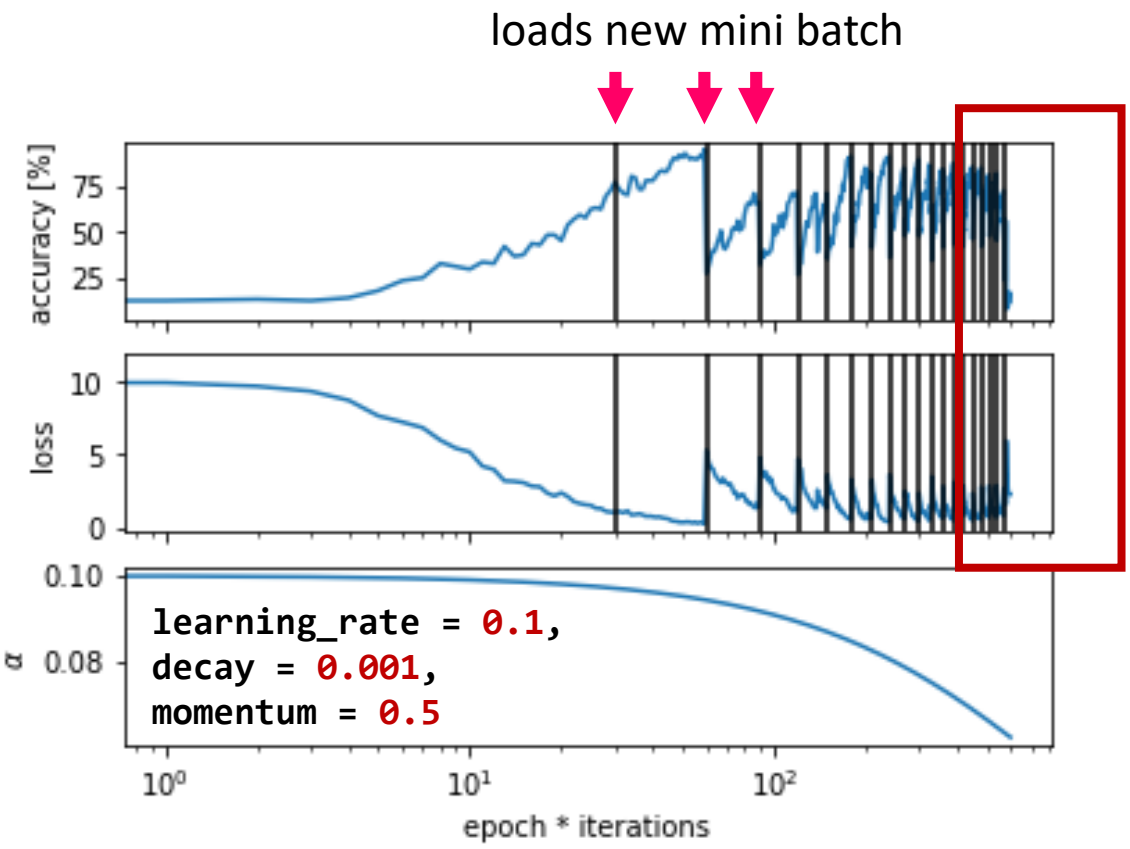




LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

The function `runMyANN_numbers` runs LeNet (ANN_MMH) with the MNIST data set

```
runMyANN_numbers(minibatch_size = 128, iterations = 20, epochs = 70)
```

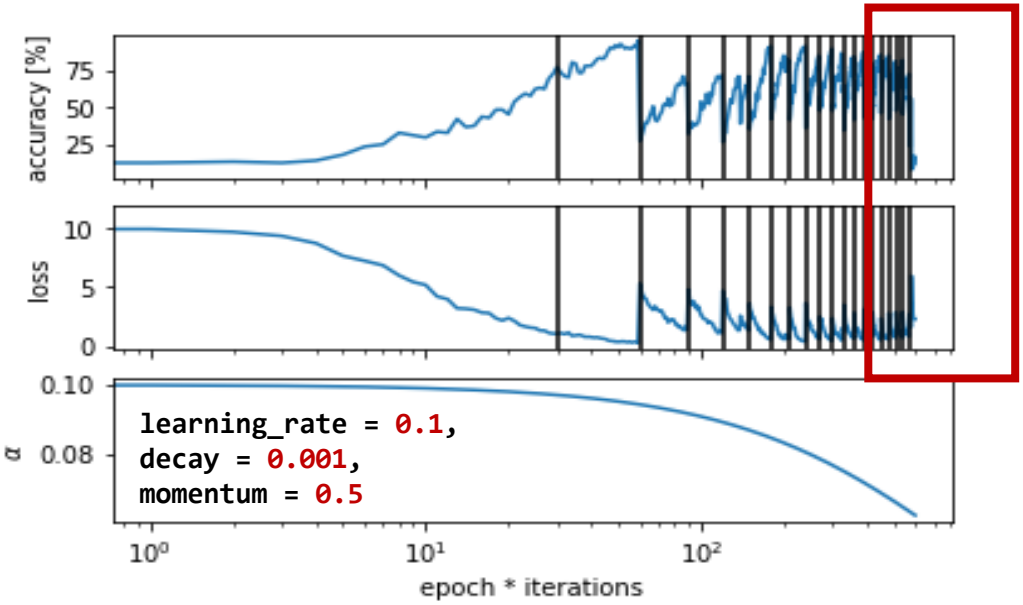



after several iterations:

→ accuracy drops and loss gets up again
→ gets stuck

**Why???**

LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**




Weights are exploding!

→ penalizing large weights
→ L1 & L2 regularization

```
W = np.load('weightsC1.npy')
W[:,:,0]
```

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | -15923.3 | -16647.4 | -16277.1 | -16993.1 | -15715.7 | -15390.1 |
| 1 | 9795.1 | 8468.91 | 9110.83 | 8205.6 | 9852.06 | 11061 |
| 2 | 37956.4 | 36572.6 | 37324.2 | 37008.4 | 37485.4 | 38515.2 |
| 3 | 39686.6 | 39131 | 39087.4 | 39614.6 | 39421.7 | 39876.7 |
| 4 | 25465.3 | 26270.1 | 25232.4 | 25836.7 | 25590.5 | 25270.9 |

LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

**Weights are exploding!**

→ **penalizing large weights**
→ **L1 & L2 regularization**

adding a **linear (L1)** and/or a **quadratic (L2)** term here

$$\rho_{i+1} = \rho_i + \mu \cdot \mu_i - \alpha \cdot grad(E)_{\rho_i} + \mathbf{2} \cdot \boldsymbol{l_2}\, \boldsymbol{\rho_i}$$

$$\mu_{i+1} = \mu \cdot \mu_i - \alpha \cdot grad(E)_{\rho_i}$$

- derivative (gradient!) of L2 term $\boldsymbol{l_2 \rho_i^2}$

- new hyper parameter $\boldsymbol{l_2}$

Hence, for every learnable (**`__init__`** part of convolution and dense layer), we add:

```
L2 = 1e-2
self.weights_L2 = L2
self.biases_L2  = L2
```

LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

**Weights are exploding!**

→ **penalizing large weights**
→ **L1 & L2 regularization**

$$\rho_{i+1} = \rho_i + \mu \cdot \mu_i - \alpha \cdot grad(E)_{\rho_i} + \mathbf{2 \cdot l_2 \rho_i}$$

$$\mu_{i+1} = \mu \cdot \mu_i - \alpha \cdot grad(E)_{\rho_i}$$

- derivative (gradient!) of L2 term $\boldsymbol{l_2 \rho_i^2}$

- new hyper parameter $\boldsymbol{l_2}$

Hence, for every learnable (**__init__** part of convolution and dense layer), we add:

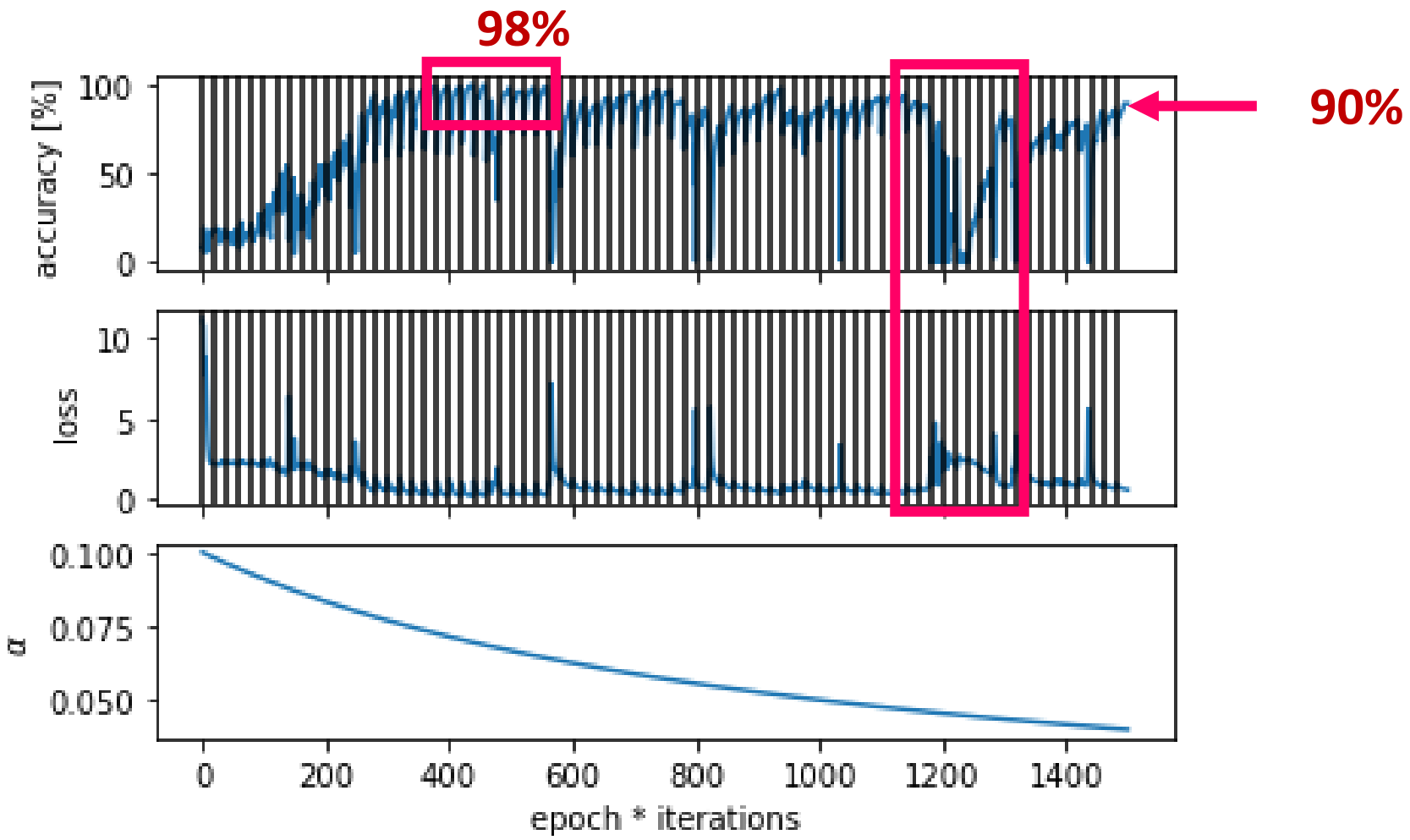
```
L2 = 1e-2
self.weights_L2 = L2
self.biases_L2  = L2
```

thus, for every **backpropagation** part of the convolution and dense layer, we add:

```
self.dbiases  = dbiases  + 2* self.biases_L2  *self.biases
self.dweights = dweights + 2* self.weights_L2 *self.weights
```

LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

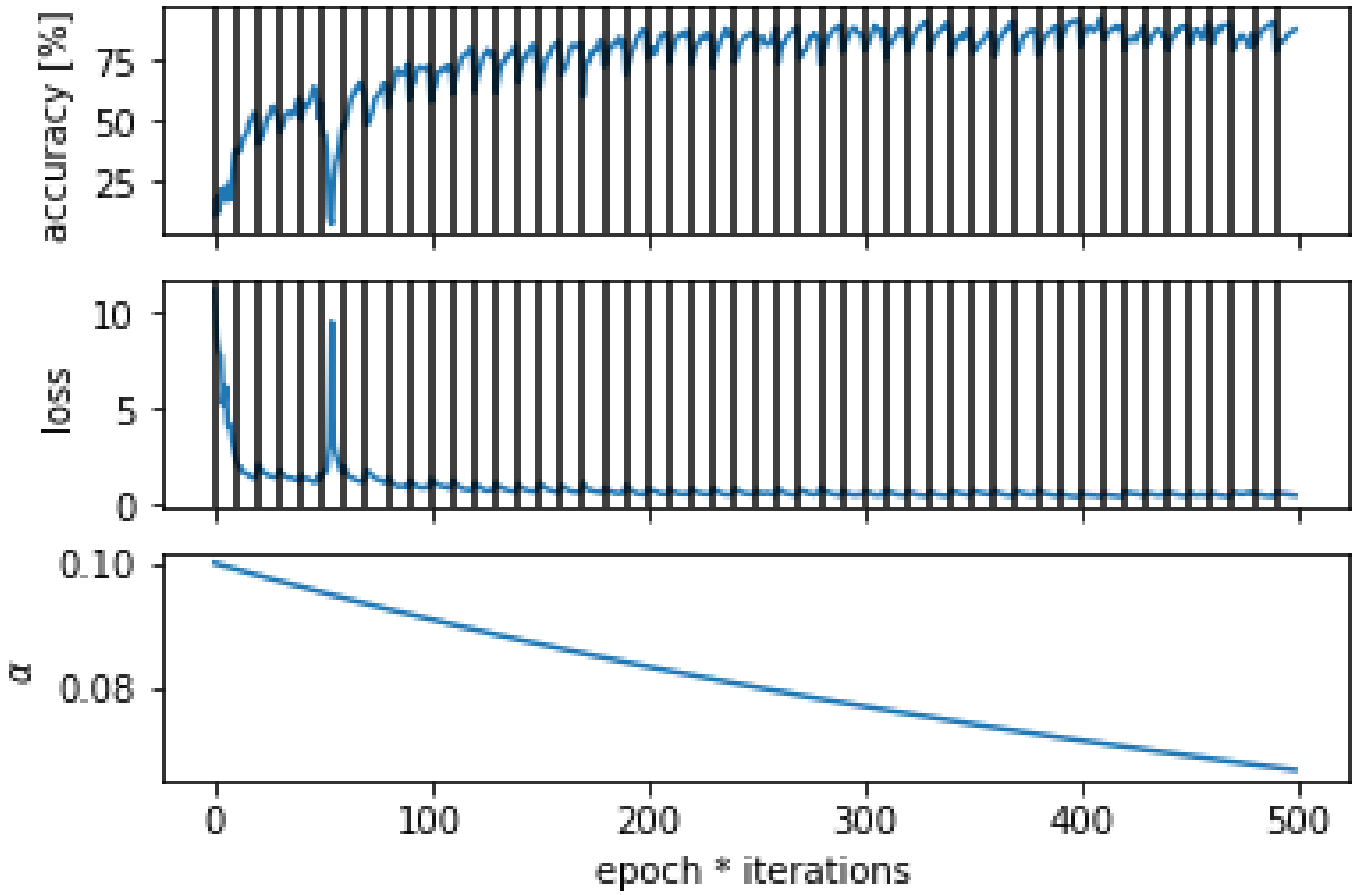
run the function `runMyANN_numbers` again, but this time with `ANN_MMH_L2`



LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

run the function `runMyANN_numbers` again, but this time with `ANN_MMH_L2`

```
minibatch_size = 512, iterations = 10, epochs = 50
```



LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

Now, we can apply the ANN (run `applyMyANN_numbers`) to a new data set:

```
from keras.datasets import mnist
import numpy as np
import random

(train_x, train_y), (test_x, test_y) = mnist.load_data()

test_x   = test_x.transpose(1,2,0)
S        = test_x.shape
test_X3D = np.zeros((S[0],S[1],3,S[2]))

test_X3D[:,:,0,:] = test_x
test_X3D[:,:,1,:] = test_x
test_X3D[:,:,2,:] = test_x

idx = random.sample(range(len(test_y)), 10)
M   = test_X3D[:,:,:,idx]
C   = test_y[idx]

[pred, probs] = applyMyANN_numbers(M)
```

rearranging images, such that last index corresponds to image

our ANN needs a 3D image as input

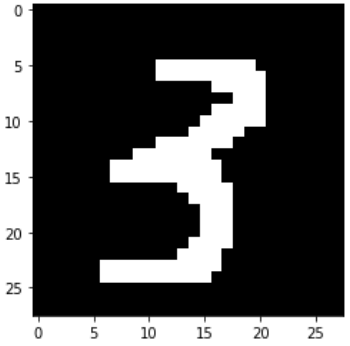
picking ten images randomly

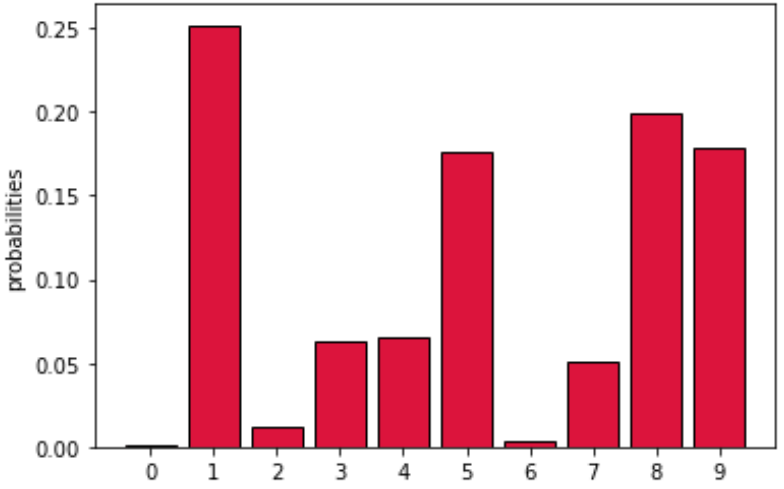
running the ANN

LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

Now, we can apply the ANN (run `applyMyANN_numbers`) to a new data set:

| | predicted | actual |
|---|---|---|
| 0 | 1 | 3 |
| 1 | 4 | 5 |
| 2 | 8 | 8 |
| 3 | 7 | 7 |
| 4 | 5 | 5 |
| 5 | 8 | 3 |
| 6 | 0 | 0 |
| 7 | 6 | 6 |
| 8 | 2 | 2 |

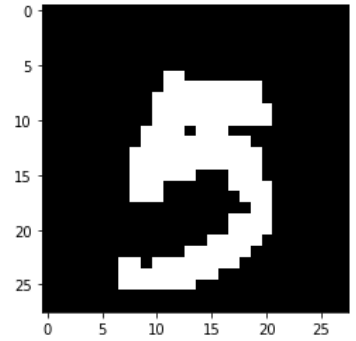


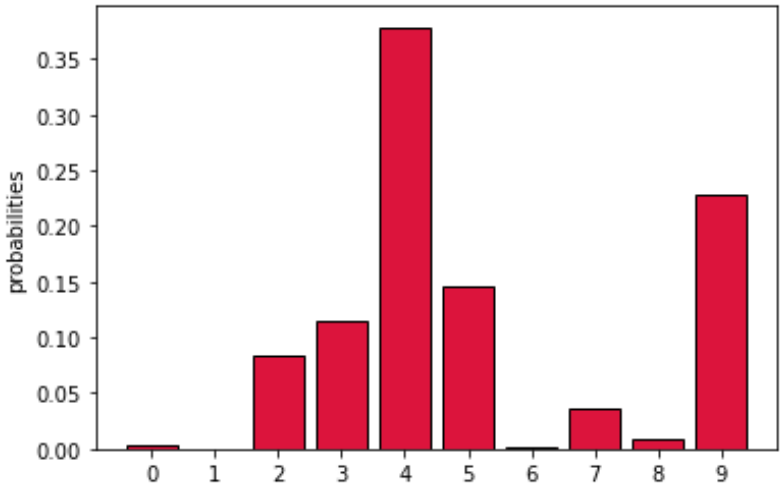


LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

Now, we can apply the ANN (run `applyMyANN_numbers`) to a new data set:

| | predicted | actual |
|---|---|---|
| 0 | 1 | 3 |
| 1 | 4 | 5 |
| 2 | 8 | 8 |
| 3 | 7 | 7 |
| 4 | 5 | 5 |
| 5 | 8 | 3 |
| 6 | 0 | 0 |
| 7 | 6 | 6 |
| 8 | 2 | 2 |

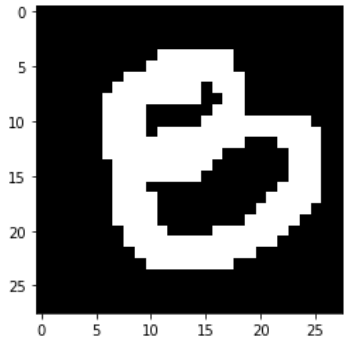


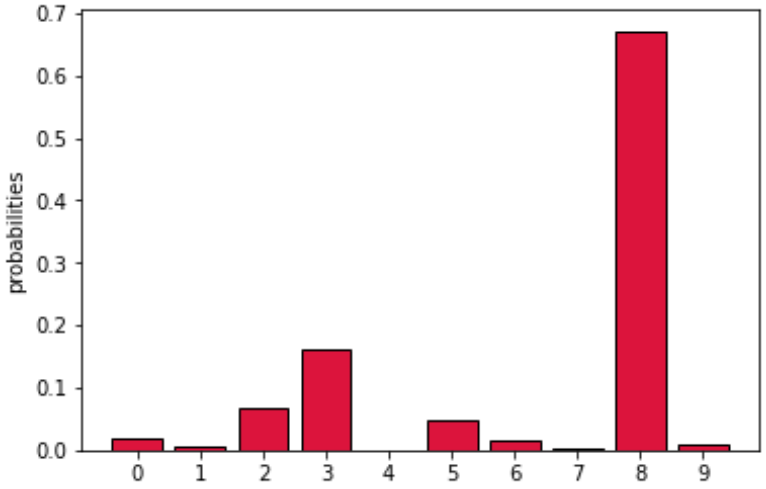


LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

Now, we can apply the ANN (run `applyMyANN_numbers`) to a new data set:

| | predicted | actual |
|---|---|---|
| 0 | 1 | 3 |
| 1 | 4 | 5 |
| 2 | 8 | 8 |
| 3 | 7 | 7 |
| 4 | 5 | 5 |
| 5 | 8 | 3 |
| 6 | 0 | 0 |
| 7 | 6 | 6 |
| 8 | 2 | 2 |

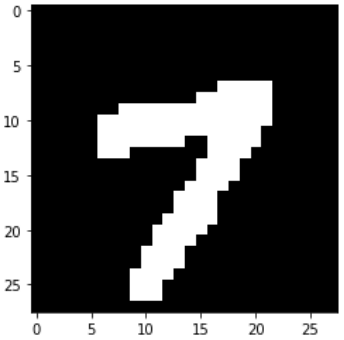


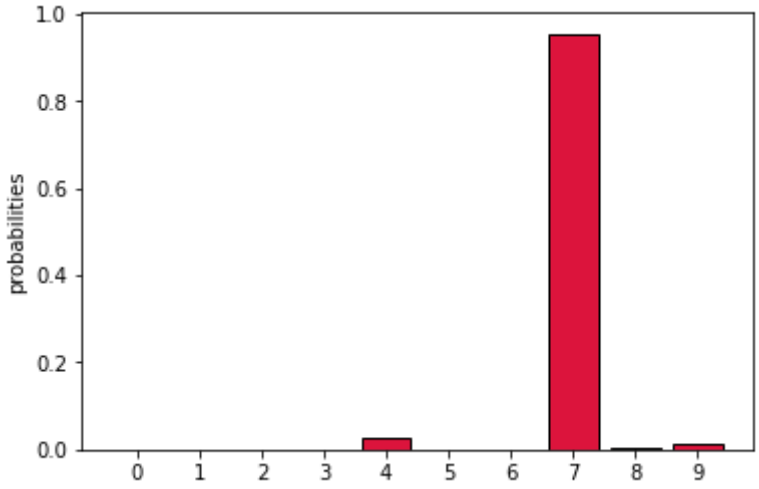


LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

Now, we can apply the ANN (run `applyMyANN_numbers`) to a new data set:

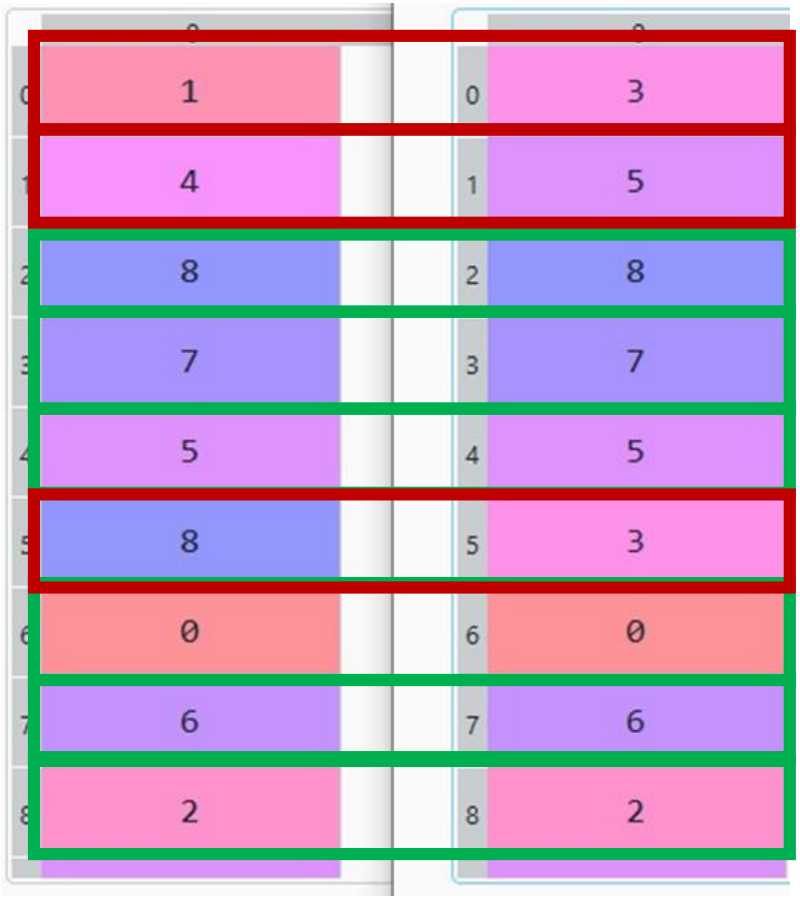
| | predicted | actual |
|---|---|---|
| 0 | 1 | 3 |
| 1 | 4 | 5 |
| 2 | 8 | 8 |
| 3 | 7 | 7 |
| 4 | 5 | 5 |
| 5 | 8 | 3 |
| 6 | 0 | 0 |
| 7 | 6 | 6 |
| 8 | 2 | 2 |





LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

Now, we can apply the ANN (run `applyMyANN_numbers`) to a new data set:



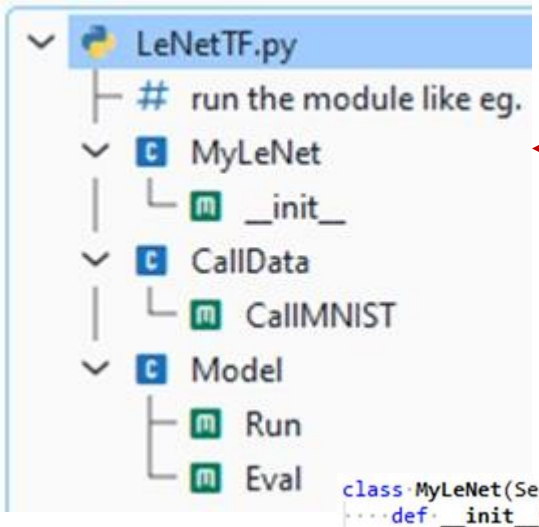

*and so on…*

**homework: train the ANN with the pet pics and evaluate the results!**

LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

let us compare our code to the **actual tensor flow version** of LeNet:

→ explore the file `LeNetTF.py`



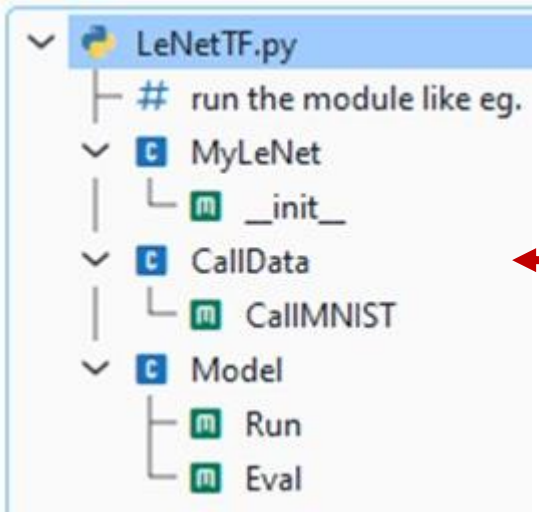

creates LeNet as class

```
class MyLeNet(Sequential):
    def __init__(self, input_shape, nb_classes):
        super().__init__()

        self.add(Conv2D(6, kernel_size=(5, 5), strides=(1, 1), activation='tanh', input_shape=input_shape, padding="same"))
        self.add(AveragePooling2D(pool_size=(2, 2), strides=(2, 2), padding='valid'))
        self.add(Conv2D(16, kernel_size=(5, 5), strides=(1, 1), activation='tanh', padding='valid'))
        self.add(AveragePooling2D(pool_size=(2, 2), strides=(2, 2), padding='valid'))
        self.add(Conv2D(120, kernel_size=(5, 5), strides=(3, 3), activation='tanh', padding='valid'))
        self.add(Flatten())
        self.add(Dense(84, activation='tanh'))
        self.add(Dense(nb_classes, activation='softmax'))

        sgd = tf.keras.optimizers.SGD(learning_rate = 0.01, momentum = 0.9,\
                                      nesterov = True)
        self.compile(optimizer = sgd,\
                     loss = categorical_crossentropy,\
                     metrics = ['accuracy'])
```

LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

let us compare our code to the **actual tensor flow version** of LeNet:

→ explore the file `LeNetTF.py`




just calls and preprocesses the data set

note these two lines:

```
Train_y = to_categorical(Train_y, num_classes)
Test_y  = to_categorical(Test_y, num_classes)
```

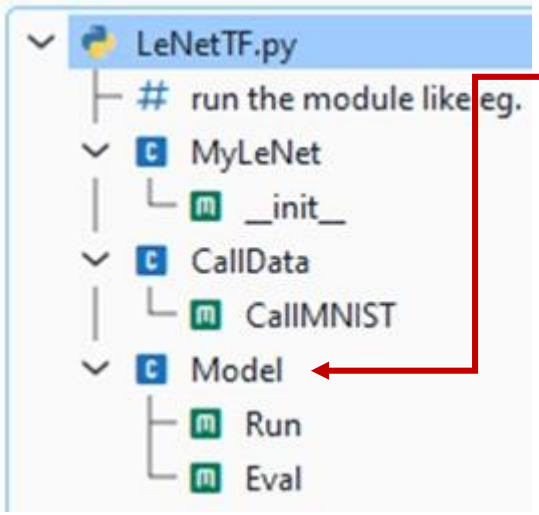
without that Tensor Flow will create a confusing error message when using categorical cross entropy for optimization

*shapes (none, 1) and (none, 10) are incompatible categorical_crossentropy*

LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

let us compare our code to the **actual tensor flow version** of LeNet:

→ explore the file `LeNetTF.py`




we only need to run this part:

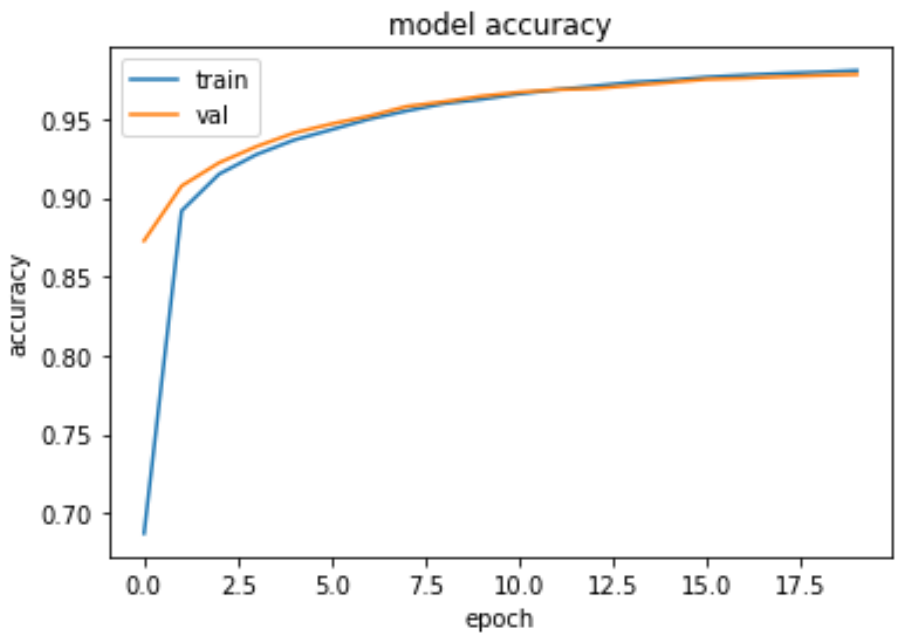
`Run:` trains the model
`Eval:` evaluates the model

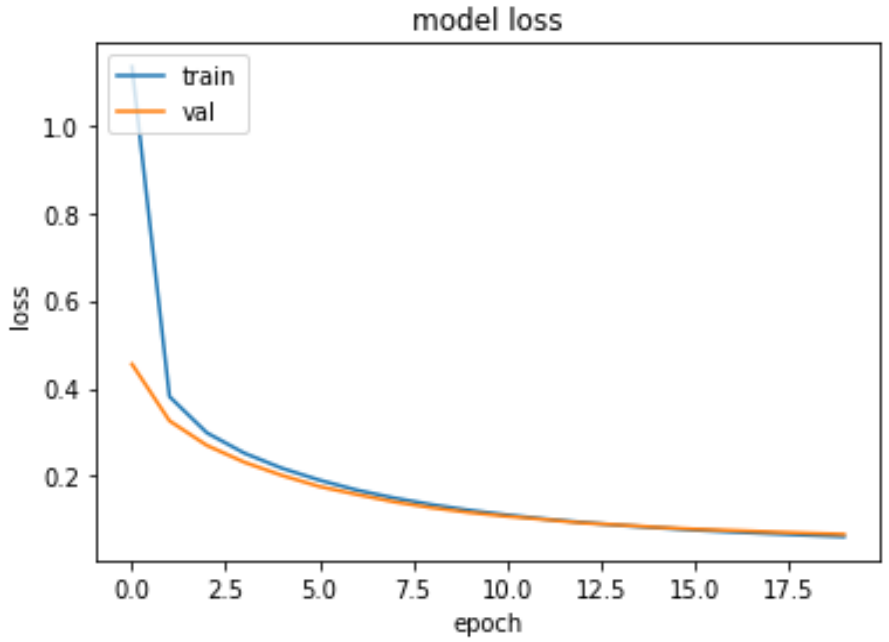
run the model with e.g.

```
import LeNetTF as L
M = L.Model()
M.Run(20)
M.Eval()
```

LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

```
import LeNetTF as L
M = L.Model()
M.Run(20)
M.Eval()
```

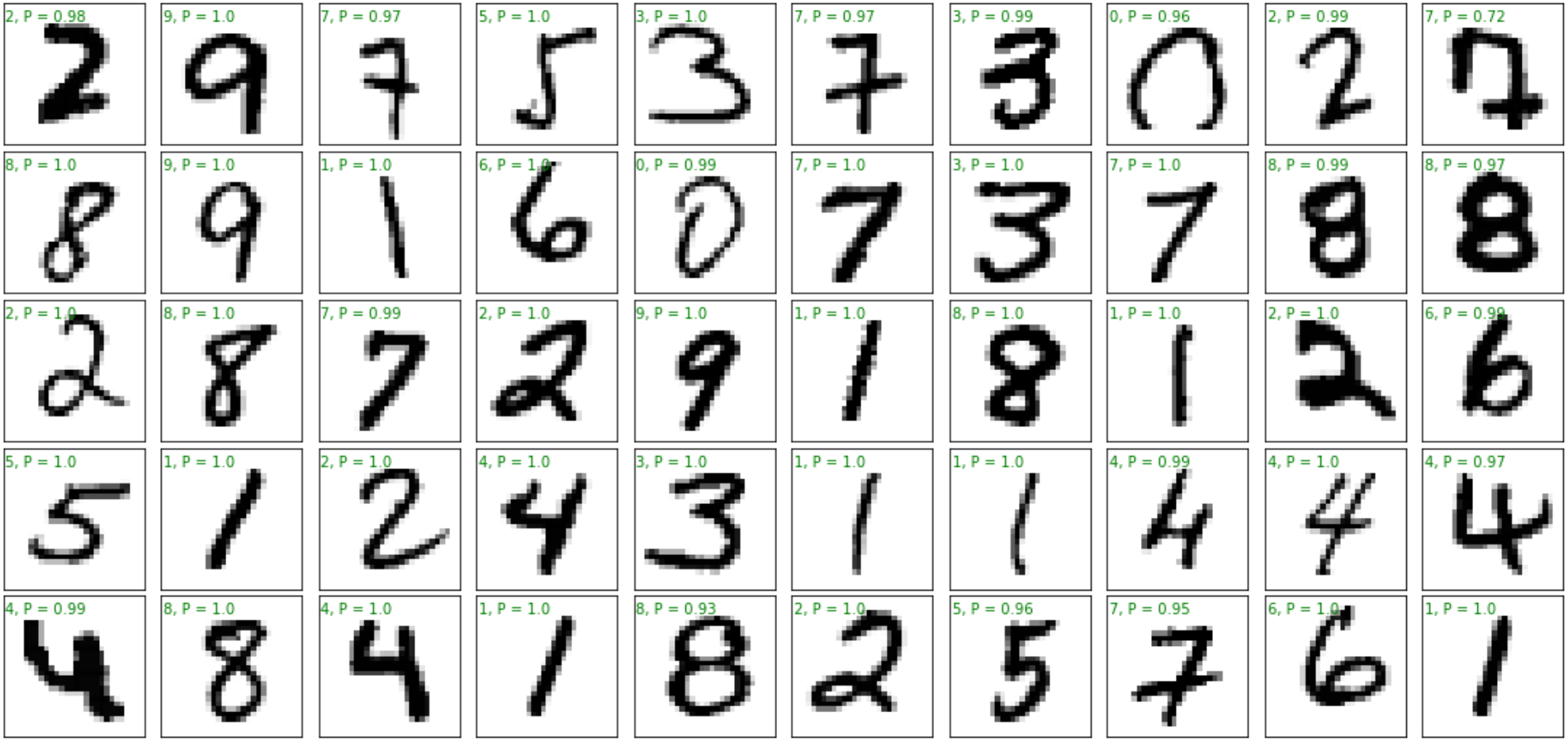



It is 100 times faster than our code!!

LeNet: **- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998**

```
import LeNetTF as L
M = L.Model()
M.Run(20)
M.Eval()
```



modern CNNs have a inception architecture → different filter sizes per layer
→ pooled in next layer(s)

**segmentation** CNNs → labels *within* each image
→ we could use our backpropagation part as decoder (with some slight modifications)

**Golden Rules of training:**

- **diversity:** all kinds of weird angles, blurred, frequency of classes
- **augmentation:** turning/stretching/blurring/random cut out
- **batch effects:** too specific vs too general → pre processing
- **labeling:** as accurate as possible, experience!
- **amount:** the more data the better

# Artificial Neuronal Networks in Python

***From Scratch!***

**Part II**

*Thank you very much for your attention!*