

Practice – CNN

- Run "7.1. CNN.ipynb"



Load pre-trained image classification models

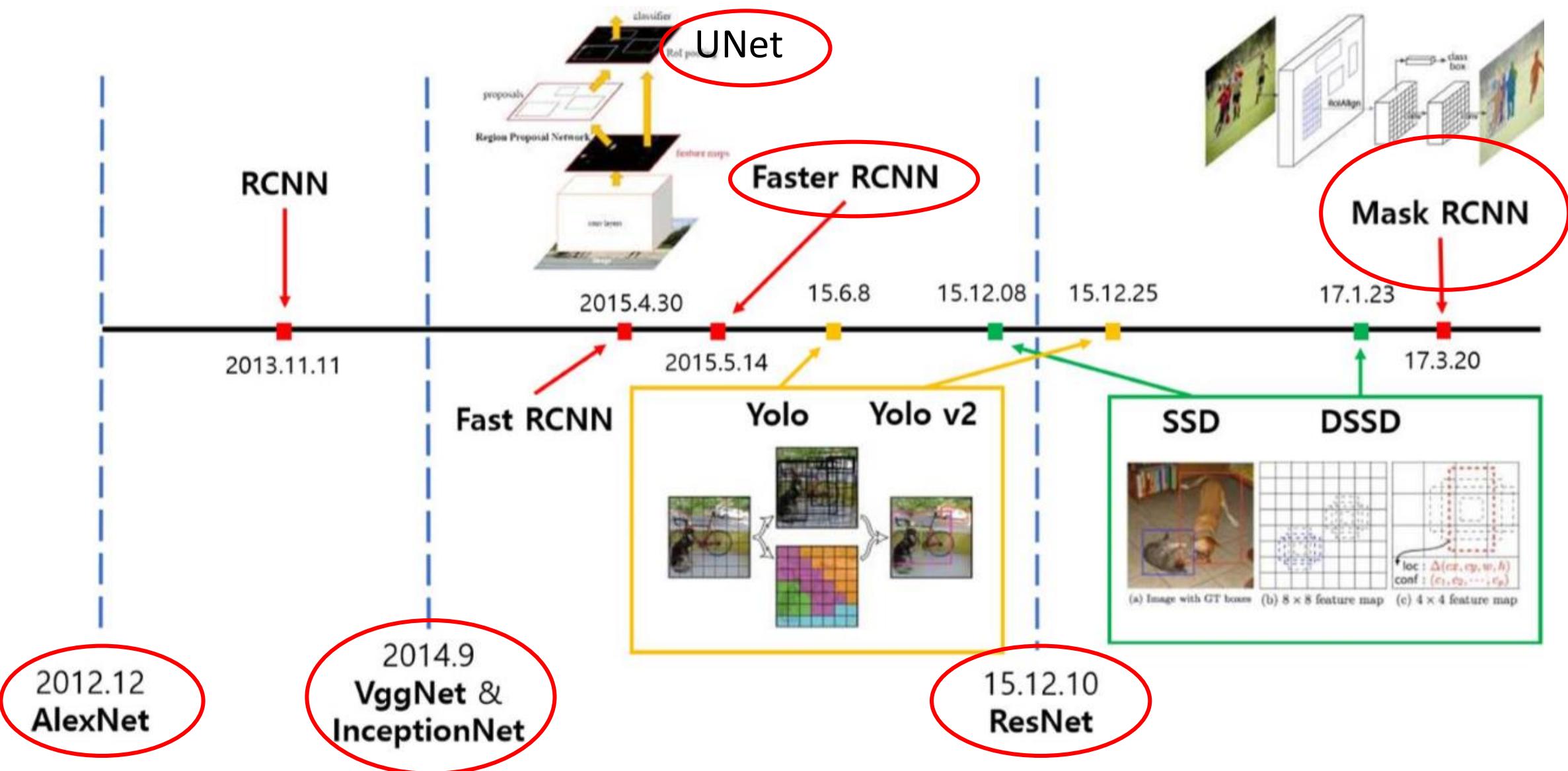
```
In [2]: import torchvision  
model = torchvision.models.alexnet(pretrained=True)  
  
Downloading: "https://download.pytorch.org/models/alexnet-owt-4df8aa71.pth" to /root/.cache/torch/hub/checkpoints/alexnet-owt-4df8aa71.pth  
  
HBox(children=(FloatProgress(value=0.0, max=244418560.0), HTML(value='')))
```

Torchvision - <https://pytorch.org/vision/stable/index.html>

ImageNet - <http://www.image-net.org/>

Image Classification - <https://machinelearningmastery.com/applications-of-deep-learning-for-computer-vision/>

AlexNet and other CNNs



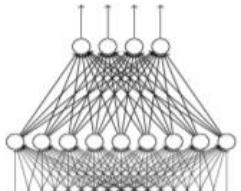
Put model in evaluation mode and in GPU

```
In [3]: model.eval()  
model.to(device)
```

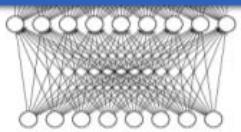
CNN contains two sections: "features" and "classifier"

The whole CNN

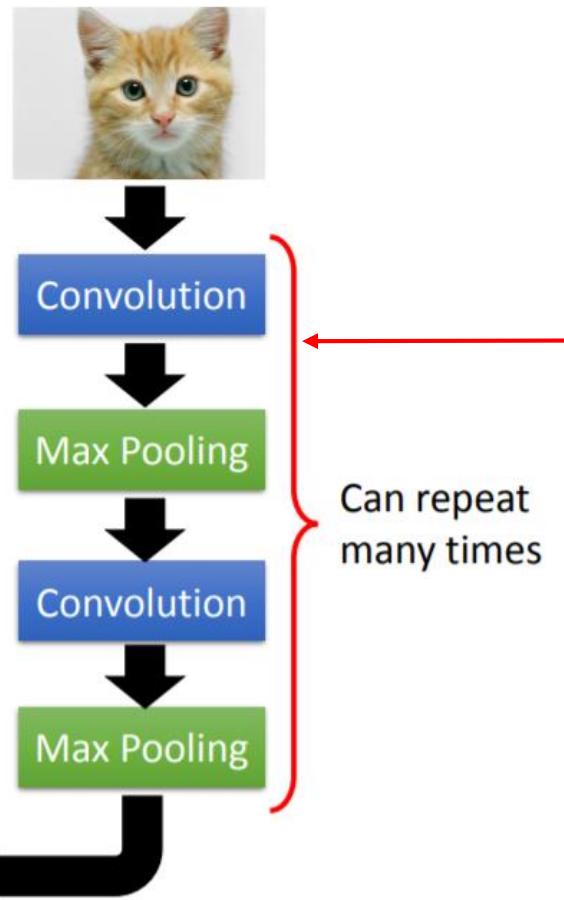
cat dog



Fully Connected
Feedforward network



Reference: 李弘毅 ML Lecture 10
<https://youtu.be/FrKWiRv254g>

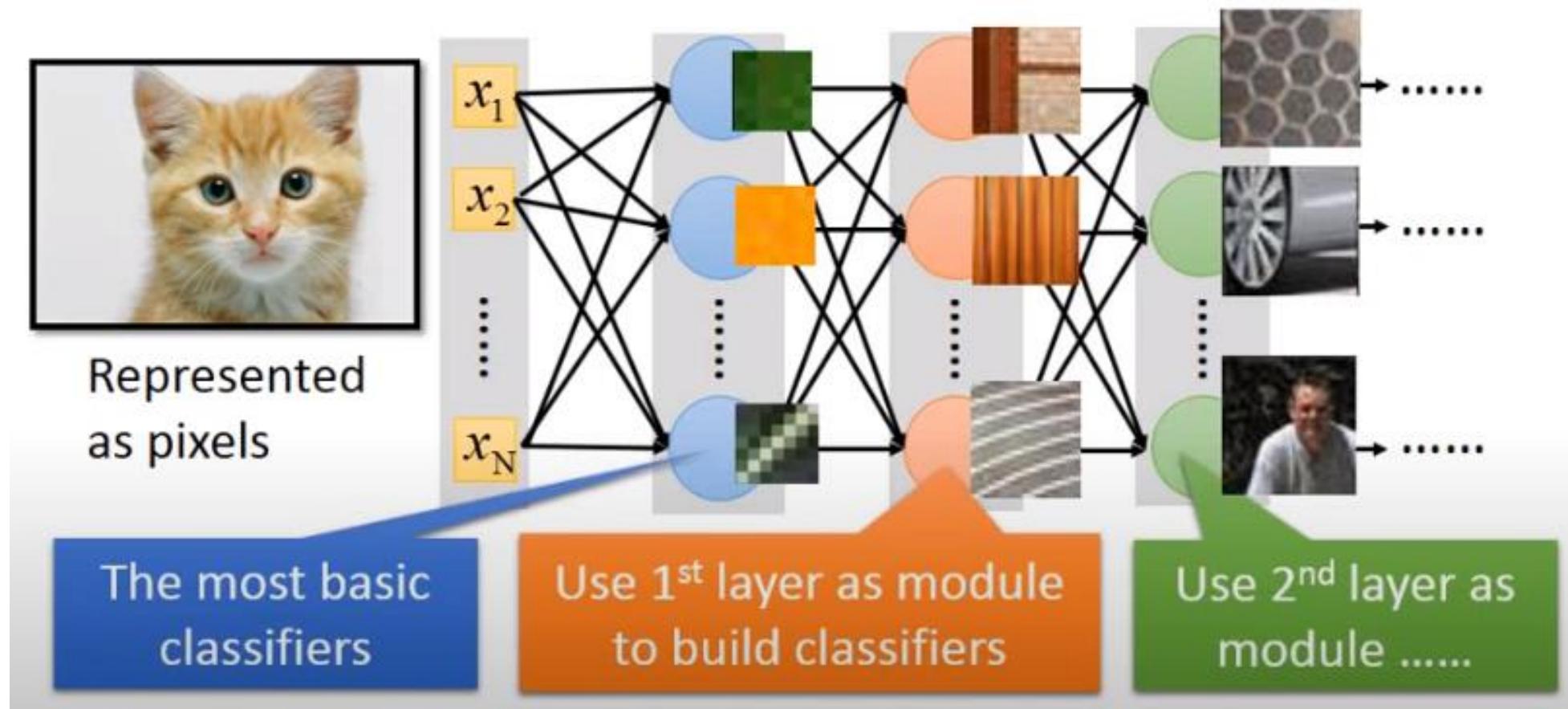


AlexNet

```
(features): Sequential(  
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=0, dilation=1, groups=1, bias=True)  
    (1): ReLU(inplace=True)  
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)  
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=2, dilation=1, groups=1, bias=True)  
    (4): ReLU(inplace=True)  
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)  
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=1, dilation=1, groups=1, bias=True)  
    (7): ReLU(inplace=True)  
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=1, dilation=1, groups=1, bias=True)  
    (9): ReLU(inplace=True)  
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=1, dilation=1, groups=1, bias=True)  
    (11): ReLU(inplace=True)  
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)  
)  
(avgpool): AdaptiveAvgPool2d(output_size=(6, 6))  
(classifier): Sequential(  
    (0): Dropout(p=0.5, inplace=False)  
    (1): Linear(in_features=9216, out_features=4096, bias=True)  
    (2): ReLU(inplace=True)  
    (3): Dropout(p=0.5, inplace=False)  
    (4): Linear(in_features=4096, out_features=4096, bias=True)  
    (5): ReLU(inplace=True)  
    (6): Linear(in_features=4096, out_features=1000, bias=True)  
)
```

What happens if we directly use MLP to classify images?

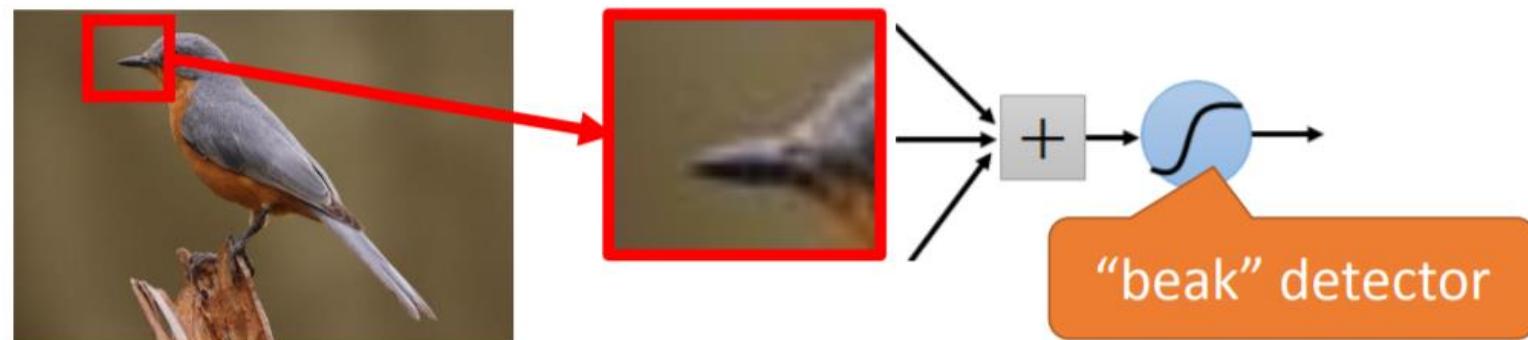
If we feed an image to MLP, then each neuron "sees" the whole image's pixels.



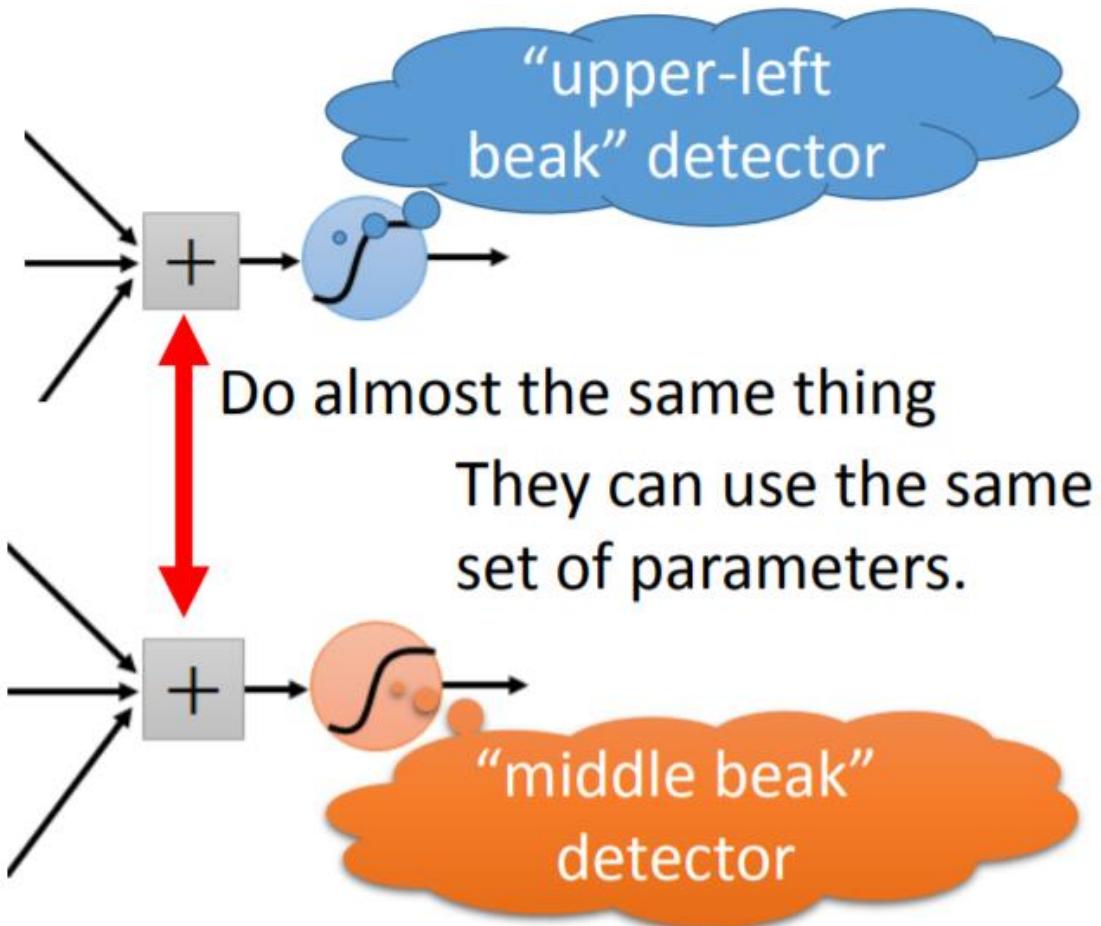
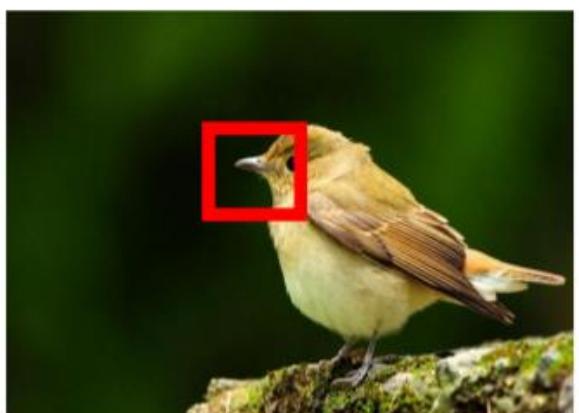
Some patterns are much smaller than the whole image

A neuron does not have to see the whole image
to discover the pattern.

Connecting to small region with less parameters



The same patterns appear in different regions



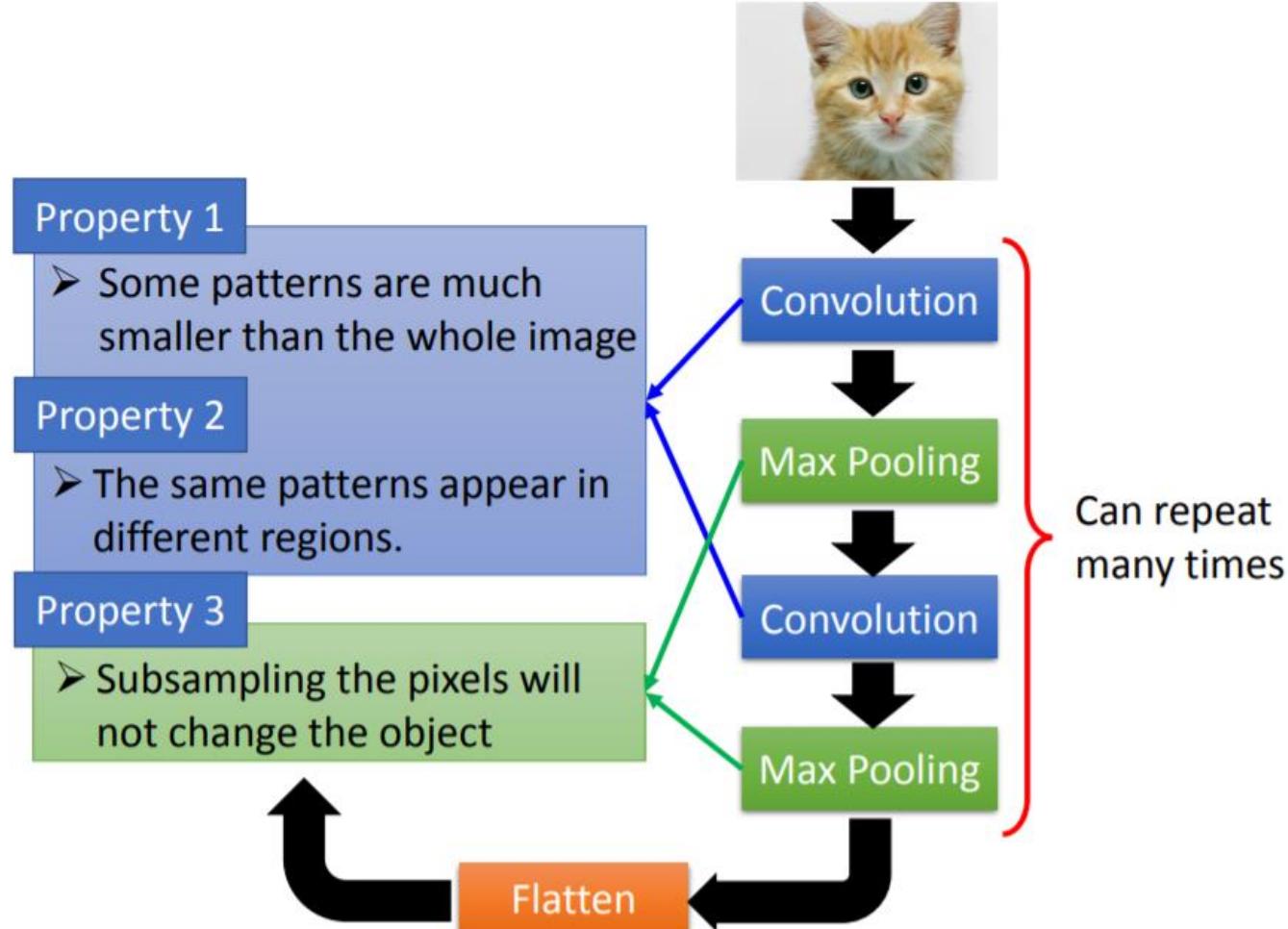
Subsampling the pixels will not change the object



We can subsample the pixels to make image smaller

→ Less parameters for the network to process the image

CNN's "features" section extracts the important features from the input image through convolution and pooling operations



Use CV to read image file

```
In [6]: import cv2  
import matplotlib.pyplot as plt  
image = cv2.imread(fname)  
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)  
plt.imshow(image)  
plt.show()
```

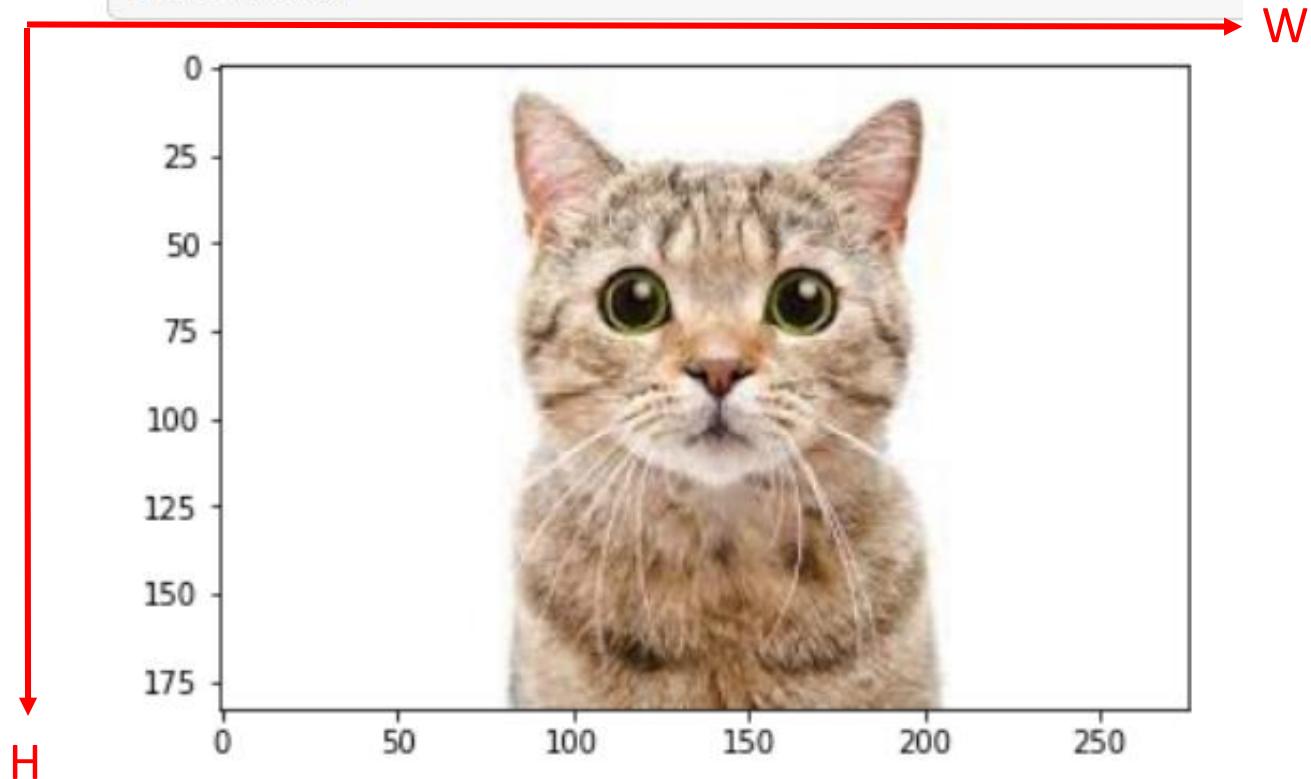


Image pre-processing

```
In [7]: from torchvision import transforms  
transformer = transforms.Compose([  
    transforms.Resize(224),  
    transforms.CenterCrop(224),  
    transforms.ToTensor(),  
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]))
```

```
In [8]: from PIL import Image  
PILImg= Image.fromarray(image.astype('uint8')).convert('RGB')  
PILImg = transformer(PILImg)  
PILImg.shape
```

```
Out[8]: torch.Size([3, 224, 224])
```

Input to CNN model

Input to CNN

```
In [9]: imageTensor = torch.unsqueeze(PILImg, 0)  
imageTensor.shape
```

```
Out[9]: torch.Size([1, 3, 224, 224])
```

Input to MLP

```
In [9]: tensorX = torch.FloatTensor(trainX).to(device)  
tensorY_hat = torch.LongTensor(trainY_hat).to(device)  
print(tensorX.shape, tensorY_hat.shape)
```

```
torch.Size([128, 2]) torch.Size([128])
```

Convolution

$$H_{out} = \frac{H_{in} + 2 \times padding - kernel\ size}{Stride} + 1$$

```
In [10]: conv1 = model.features[0]
print(conv1)
#InChannel=3(RGB),OutChannel=64, filter size=11, stride=4, padding=2

Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
```

```
In [11]: weight1 = conv1.weight.data.cpu().numpy()
print(weight1.shape)
#64 filters, depth=3, size =11 by 11

(64, 3, 11, 11)
```

```
In [12]: conv1_out = conv1(imageTensor.to(device))
conv1_out.shape
#output image (feature map) has 64 channels
```

Out[12]: torch.Size([1, 64, 55, 55]) $\frac{224 + 2 \times 2 - 11}{4} + 1 = 55.25$



Filter searches patterns in a small region

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

6 x 6 image

Those are the network parameters to be learned.

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1
Matrix

-1	1	-1
-1	1	-1
-1	1	-1

Filter 2
Matrix

⋮

Property 1

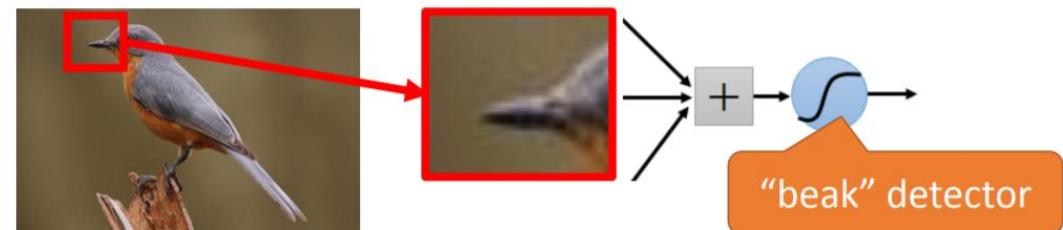
Each filter detects a small pattern (3 x 3).

Property 1

- Some patterns are much smaller than the whole image

A neuron does not have to see the whole image to discover the pattern.

Connecting to small region with less parameters



Filter searches a particular pattern in different regions

Use Excel to verify!

stride=1

1	0	0	0	0	0	1
0	-1	0	0	1	0	0
0	0	1	0	0	0	0
1	0	0	0	0	1	0
0	0	0	0	1	0	0
0	0	1	0	0	1	0

6 x 6 image

$$\begin{matrix} 1 & -1 & -1 \\ -1 & 1 & -1 \\ -1 & -1 & 1 \end{matrix}$$

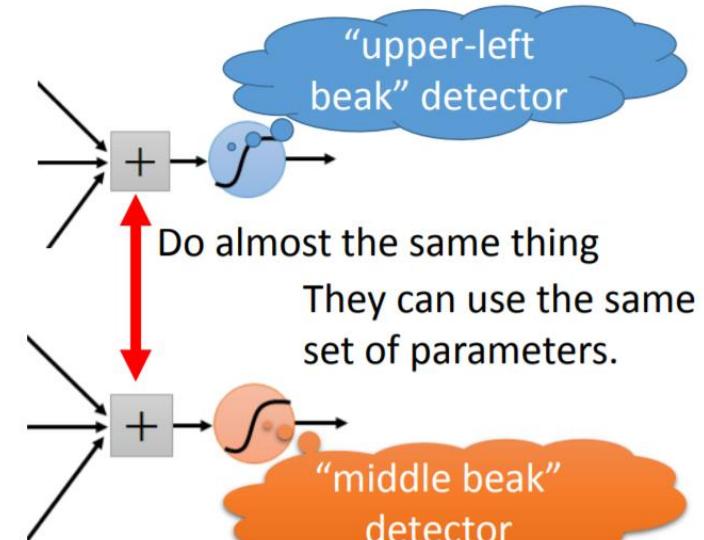
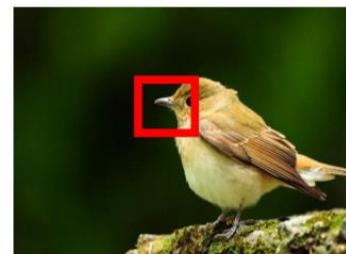
Filter 1

$$\begin{matrix} 3 & -1 & -3 & -1 \\ -3 & 1 & 0 & -3 \\ -3 & -3 & 0 & 1 \\ 3 & -2 & -2 & -1 \end{matrix}$$

Property 2

Property 2

- The same patterns appear in different regions.



Stride determines how filter shifts

stride=1

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

1	-1	-1
-1	1	-1
-1	-1	1

3 -1

If stride=2

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

3 -3

We set stride=1 below

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

Use Excel to verify!

Feature maps

Each filter searches a small region and summarizes how the specified pattern appears in different regions in a feature map

stride=1

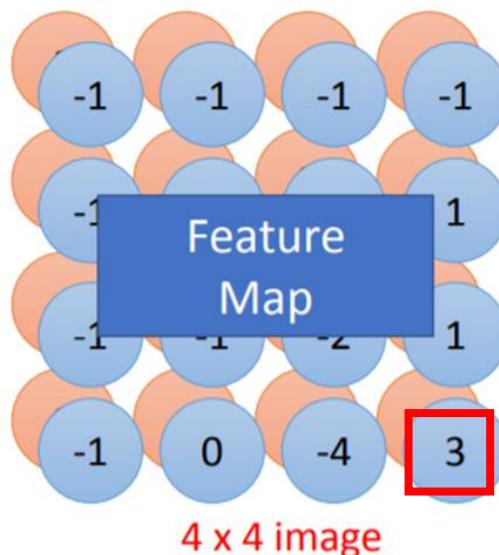
1	0	0	0	0	0	1
0	1	0	0	0	1	0
0	0	1	1	0	0	0
1	0	0	0	1	0	0
0	1	0	0	1	0	0
0	0	1	0	1	0	0

6 x 6 image

-1	1	-1
-1	1	-1
-1	1	-1

Filter 2

Do the same process for every filter



In [10]:

```
conv1 = model.features[0]
print(conv1)
#InChannel=3(RGB),outChannel=64, filter size
```

Conv2d(3, 64, kernel_size=(11, 11), stride=(

In [11]:

```
weight1 = conv1.weight.data.cpu().numpy()
print(weight1.shape)
#64 filters, depth=3, size =11 by 11
```

(64, 3, 11, 11)

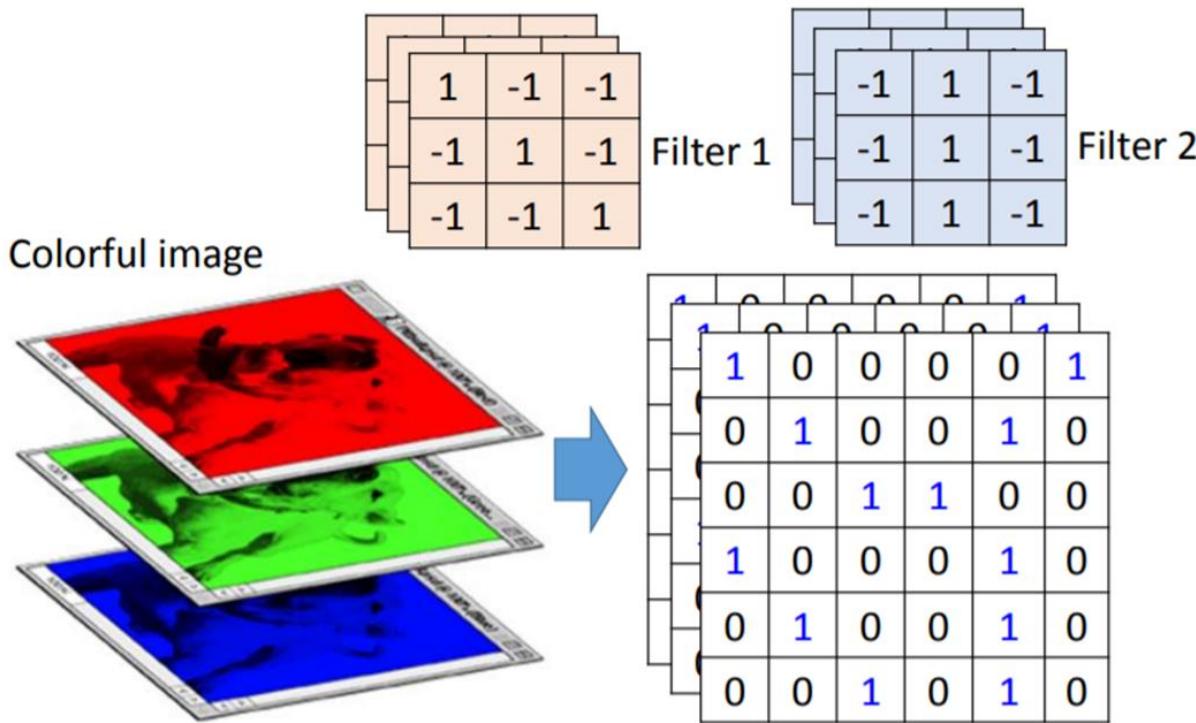
In [12]:

```
conv1_out = conv1(imageTensor.to(device))
conv1_out.shape
#output image (feature map) has 64 channels
```

out[12]: torch.Size([1, 64, 55, 55])

Filter has depth

If input image has 3 channels, then each convolution filter also has 3 channels



```
[10] conv1 = model.features[0]
      print(conv1)
      #InChannel=3 (RGB), OutChannel=64, filter size=11,
```

Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4),

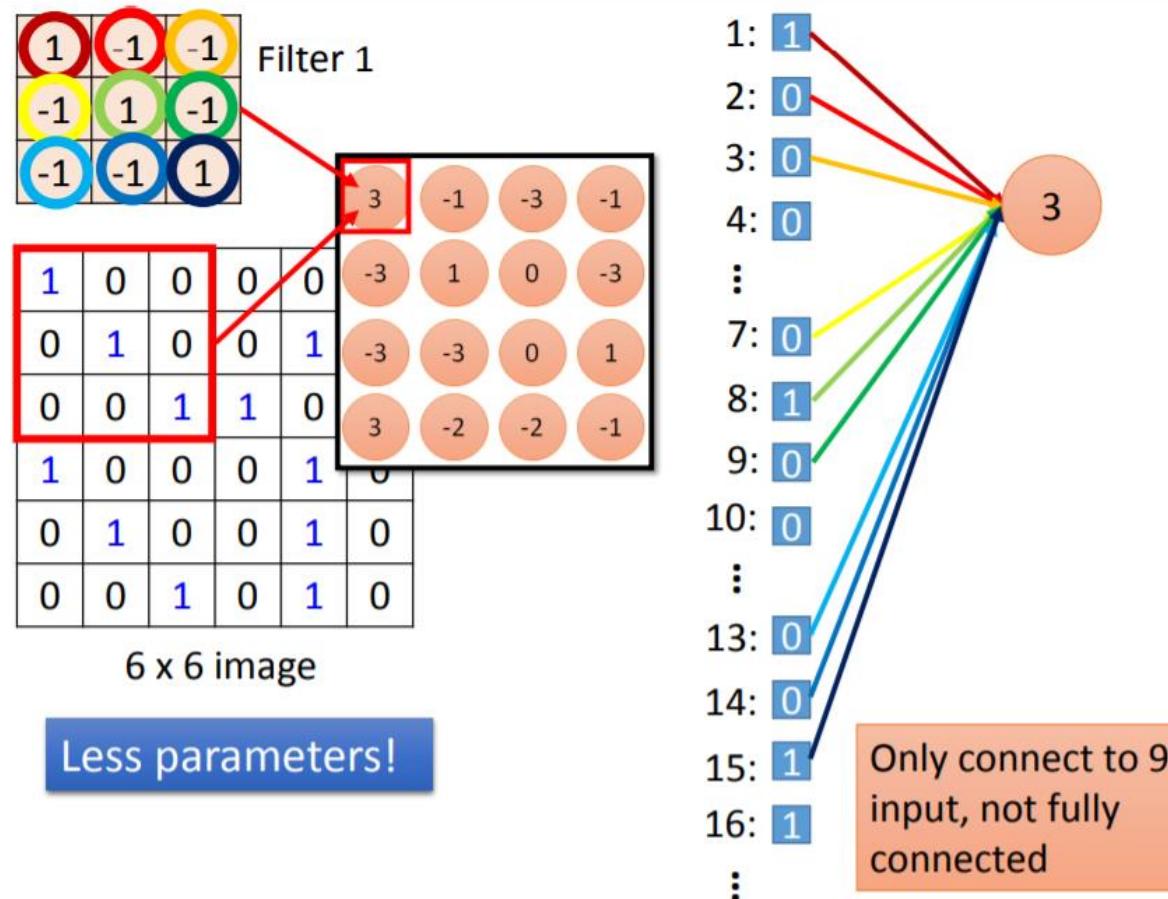
```
[11] weight1 = conv1.weight.data.cpu().numpy()
      print(weight1.shape)
      #64 filters, depth=3, size =11 by 11
```

(64, 3, 11, 11)

64 filters, each filter has
3 channels

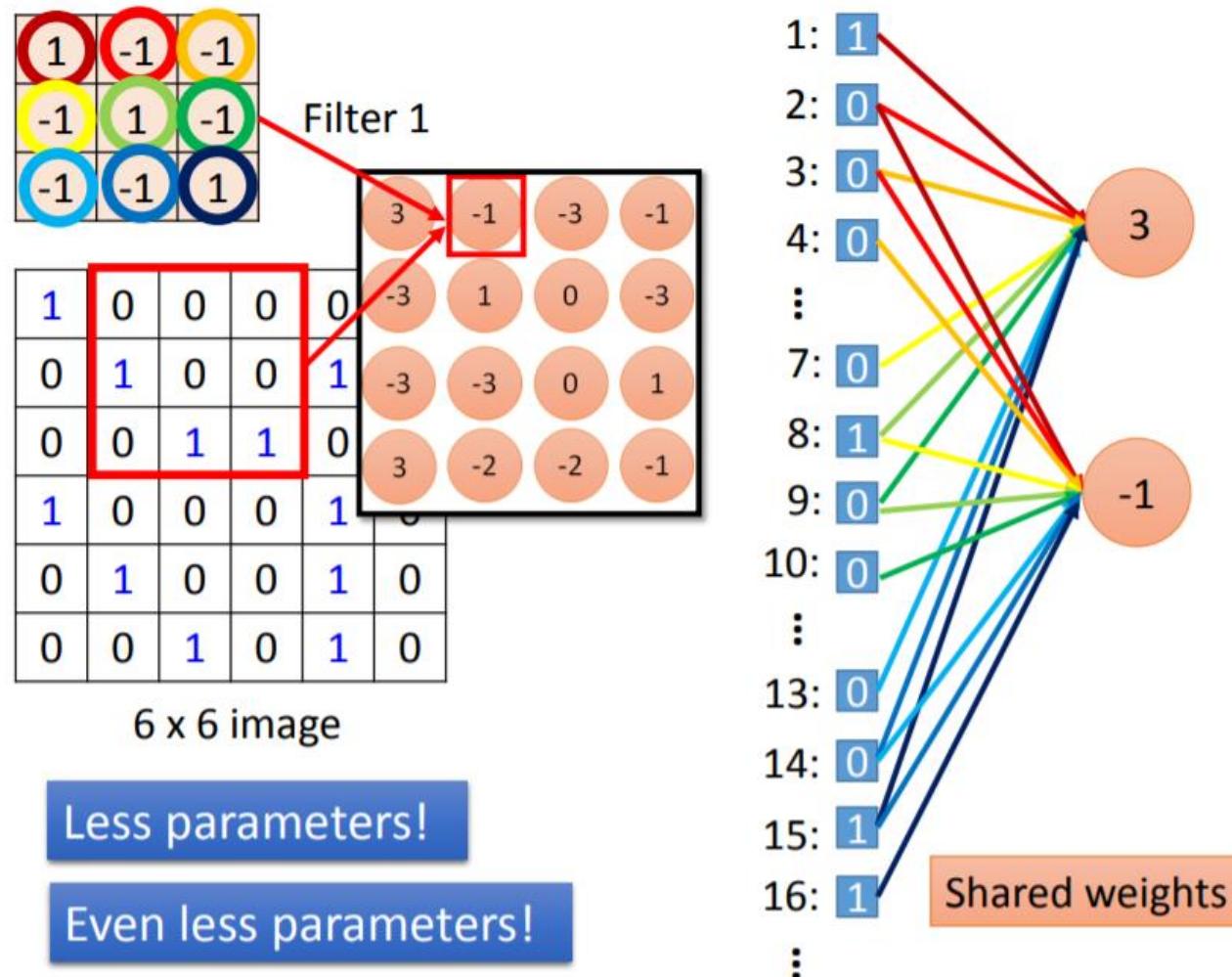
Convolution represented as neural network

Convolution can be represented as partially connected NN, which has less parameters and is less complicated than the fully connected NN.



Convolution can be represented as neural network

Partially connected NN with shared weights and hence with even less parameters.



Feature map with K channels

K = No. of filters

```
[10]: conv1 = model.features[0]
print(conv1)
#InChannel=3(RGB),OutChannel=64, filter size=11, stride=4, padding=2
Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
```

```
[11]: weight1 = conv1.weight.data.cpu().numpy()
print(weight1.shape)
#64 filters, depth=3, size =11 by 11
(64, 3, 11, 11) 64 filters, each has 3 channels, are applied to
                  the input image (with 3 channels RGB)
```

```
[12]: conv1_out = conv1(imageTensor.to(device))
conv1_out.shape
#output image (feature map) has 64 channels
```

```
:[12]: torch.Size([1, 64, 55, 55]) After convolution, the output image (feature
map) has 64 channels
```

```
[13]: # Visualize the first 32 channels of the output feature map
imgArray=conv1_out[0].data.cpu().numpy()
fig=plt.figure(figsize=(18, 9))
for i in range(32):
    fig.add_subplot(4, 8, i+1)
    plt.imshow(imgArray[i], cmap='gray')
plt.show()
```

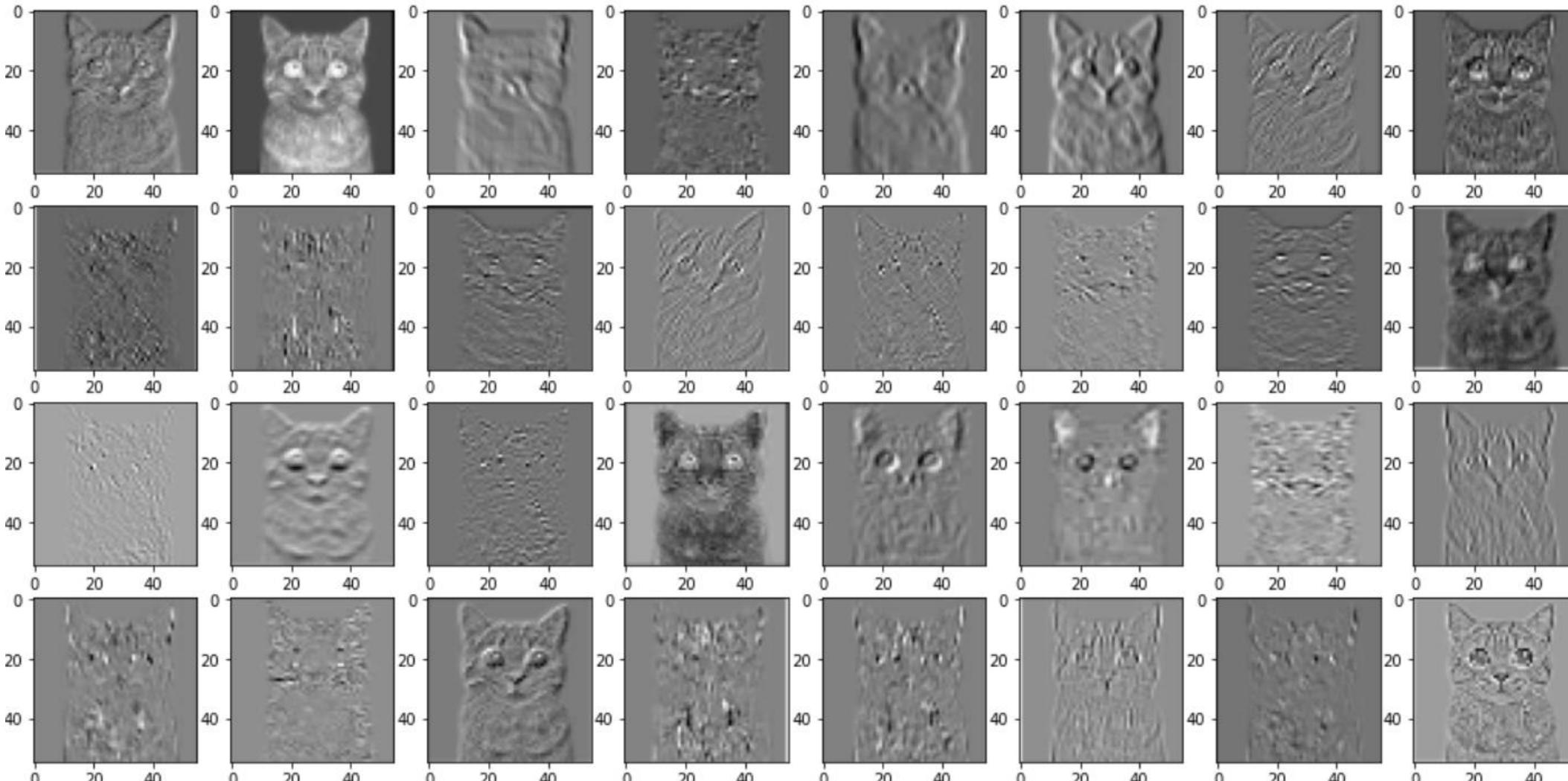


Feature map with K channels

K = No. of filters

Input image with 3 channels

Output image (feature map) with 64 channels



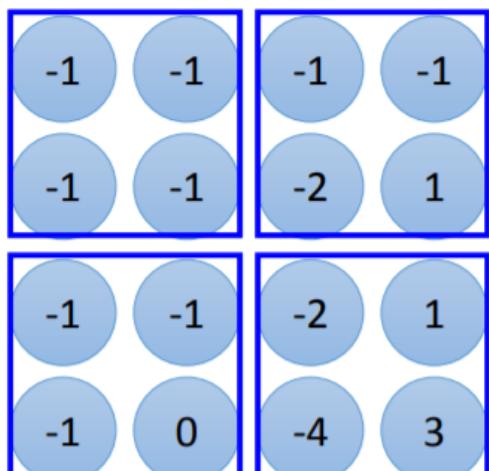
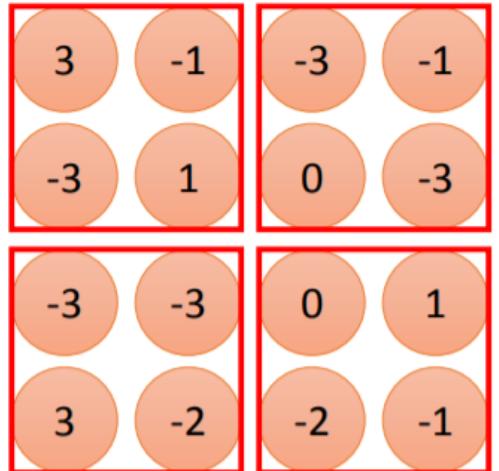
Max pooling

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

-1	1	-1
-1	1	-1
-1	1	-1

Filter 2

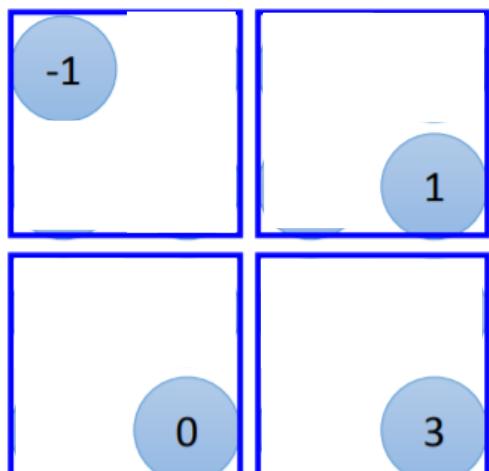
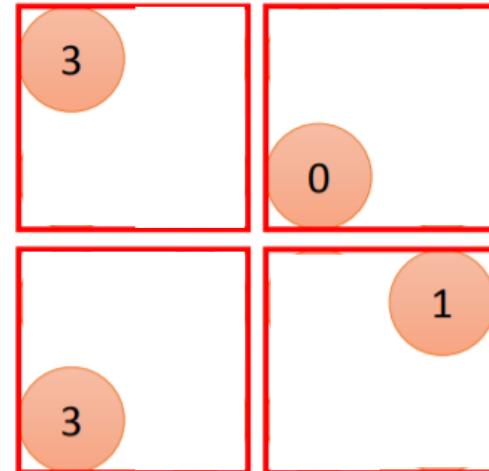


1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

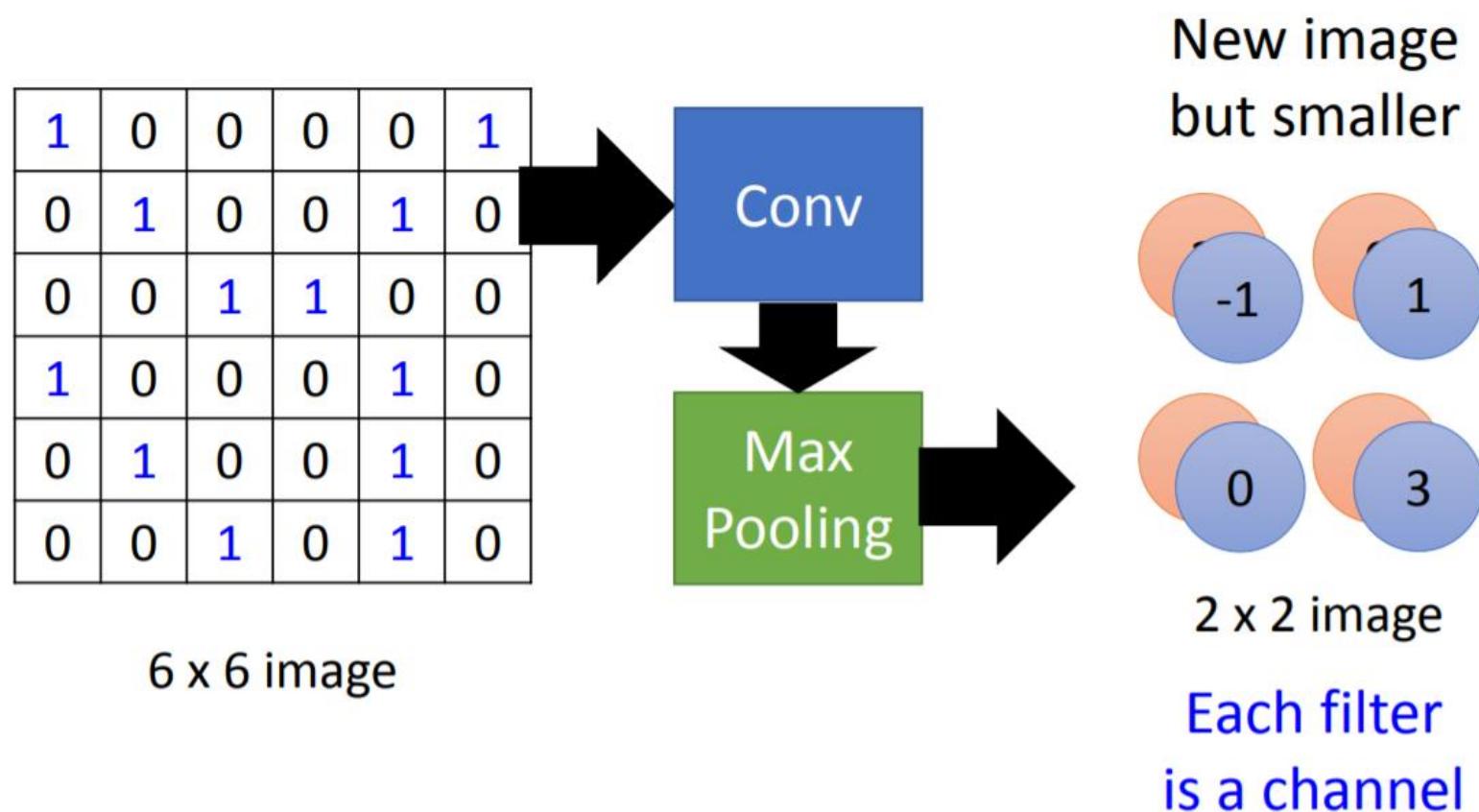
-1	1	-1
-1	1	-1
-1	1	-1

Filter 2



Convolution + Max pooling

After applying K filters of depth 3 + max pooling to the input image, the output image is a smaller feature map with K channels.



Output image (feature map) with K channels

```
features[1, 2]  
[14]: conv1_pooling = model.features[1:3]  
conv1_out1 = conv1_pooling(conv1_out)  
print(conv1_out1.shape)  
imgArray=conv1_out1[0].data.cpu().numpy()  
fig=plt.figure(figsize=(18, 9))  
for i in range(32): #visualize the first 32 channels  
    fig.add_subplot(4, 8, i+1)  
    plt.imshow(imgArray[i], cmap='gray')  
plt.show()  
  
torch.Size([1, 64, 27, 27])
```

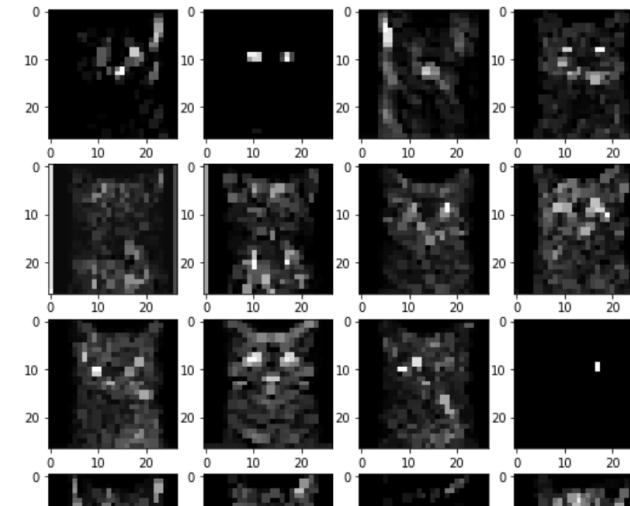
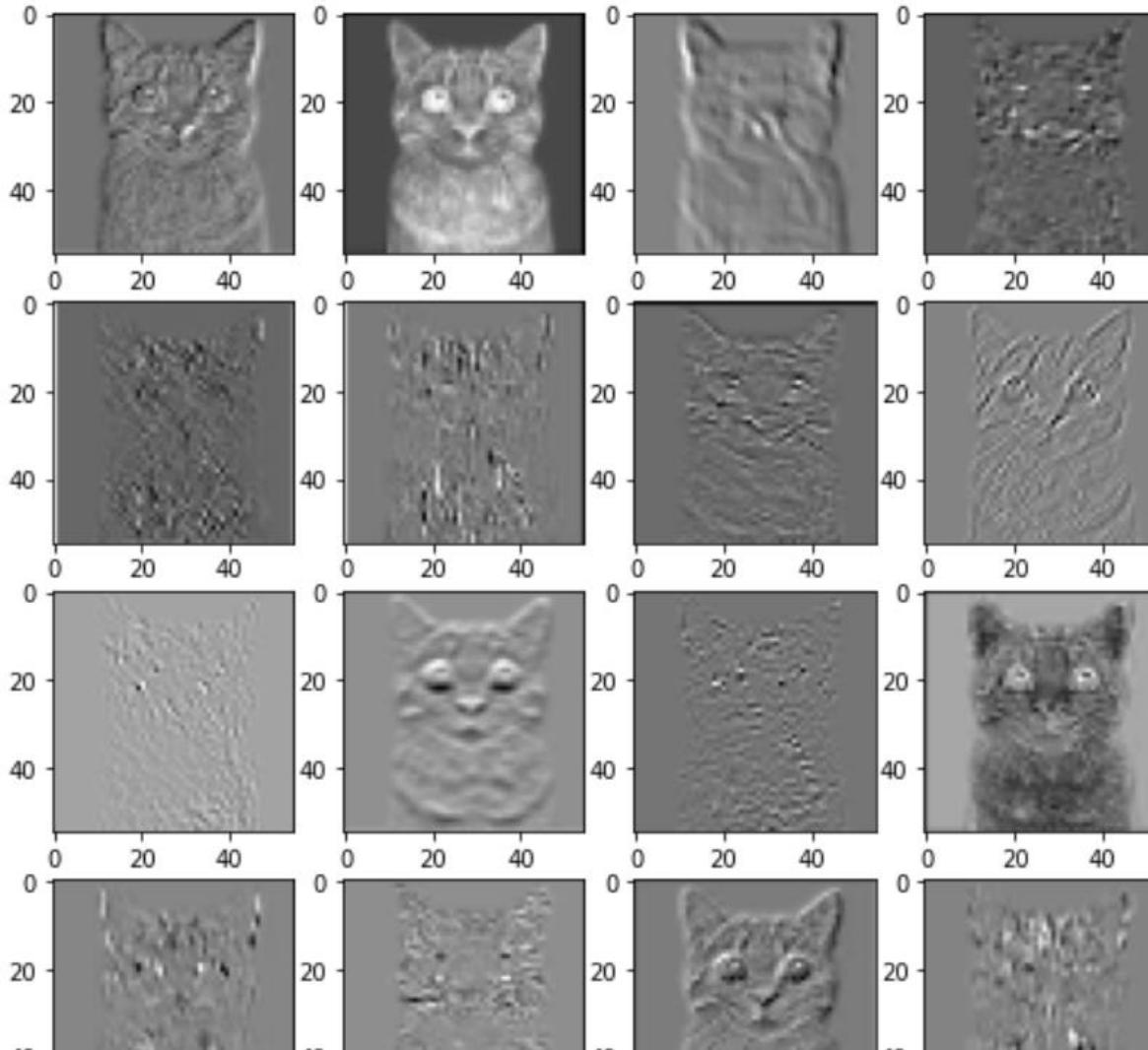
$$\frac{55 + 2 \times 2 - 3}{2} + 1 = 27$$

$$H_{out} = \frac{H_{in} + 2 \times padding - kernel\ size}{Stride} + 1$$

```
AlexNet(  
    (features): Sequential(  
        (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=0, dilation=1, groups=1, bias=True)  
        (1): ReLU(inplace=True)  
        (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)  
        (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=2, dilation=1, groups=1, bias=True)  
        (4): ReLU(inplace=True)  
        (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)  
        (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=1, dilation=1, groups=1, bias=True)  
        (7): ReLU(inplace=True)  
        (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=1, dilation=1, groups=1, bias=True)  
        (9): ReLU(inplace=True)  
        (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=1, dilation=1, groups=1, bias=True)  
        (11): ReLU(inplace=True)  
        (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)  
    )
```



Output image (feature map) with K channels



Repeat – 2nd convolution

```
[15]: conv2 = model.features[3]
conv2_out = conv2(conv1_out)
print(conv2_out.shape)
imgArray=conv2_out[0].data.cpu().numpy()
fig=plt.figure(figsize=(18, 9))
for i in range(32): #visualize the first 32 channels
    fig.add_subplot(4, 8, i+1)
    plt.imshow(imgArray[i], cmap='gray')
plt.show()

torch.Size([1, 192, 27, 27])
```

After convolution, the output feature map has 192 channels

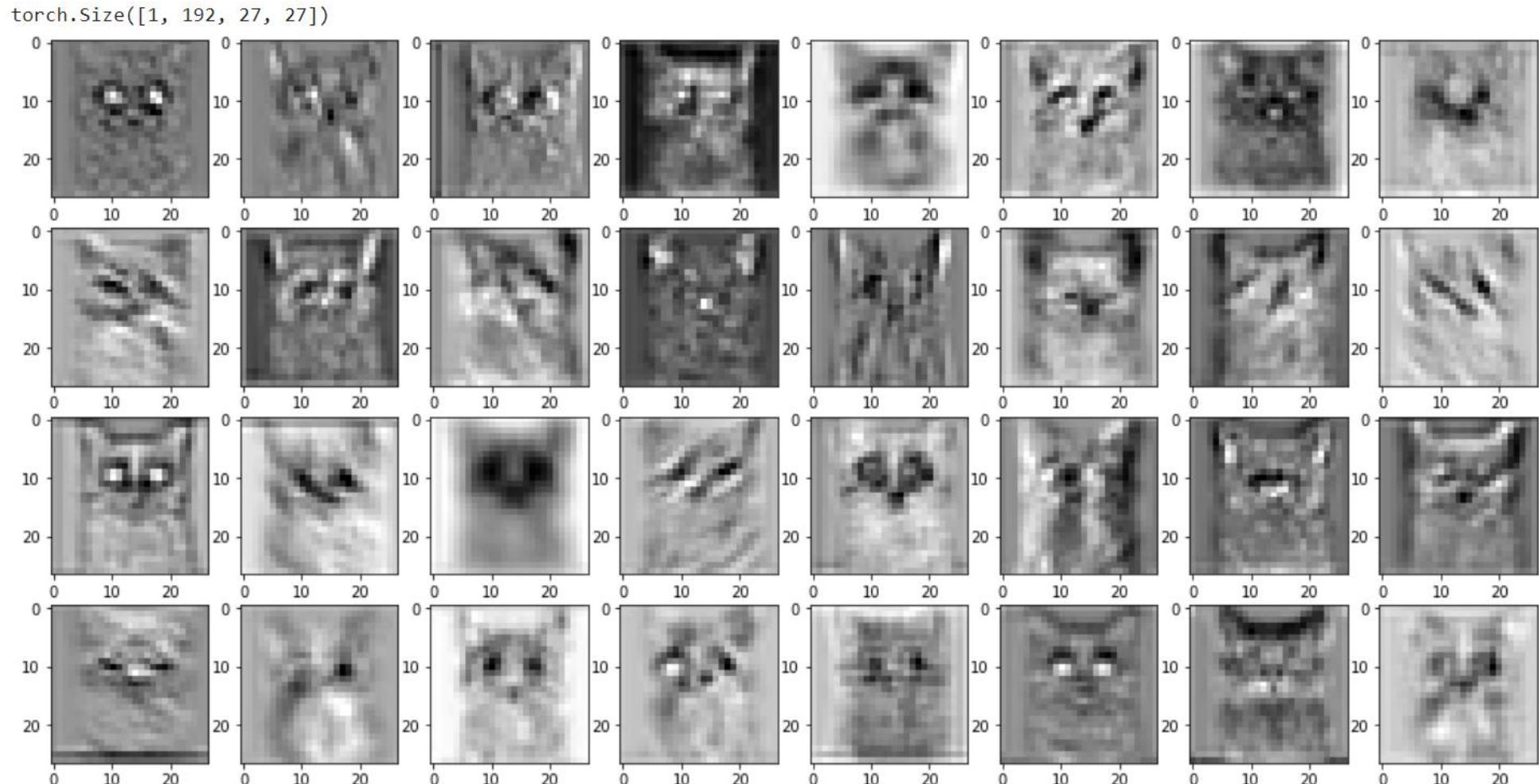
$$\frac{27 + 2 \times 2 - 5}{1} + 1 = 27$$

$$H_{out} = \frac{H_{in} + 2 \times padding - kernel\ size}{Stride} + 1$$

192 filters, each has 64 channels, are applied to the input feature map (with 64 channels)

```
AlexNet(
    features: Sequential(
        (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padd
        (1): ReLU(inplace=True)
        (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=
        (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padd
        (4): ReLU(inplace=True)
        (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=
        (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), pad
        (7): ReLU(inplace=True)
        (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), pad
        (9): ReLU(inplace=True)
        (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), pa
        (11): ReLU(inplace=True)
        (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation
    )
```

Repeat – 2nd convolution



Repeat – max pooling

features[4, 5]

```
[16]: conv2_pooling = model.features[4:6]
conv2_out1 = conv2_pooling(conv2_out)
print(conv2_out1.shape)
imgArray=conv2_out1[0].data.cpu().numpy()
fig=plt.figure(figsize=(18, 9))
for i in range(32): #visualize the first 32 channels
    fig.add_subplot(4, 8, i+1)
    plt.imshow(imgArray[i], cmap='gray')
plt.show()

torch.size([1, 192, 13, 13])
```

$$\frac{27 + 2 \times 0 - 3}{2} + 1 = 13$$

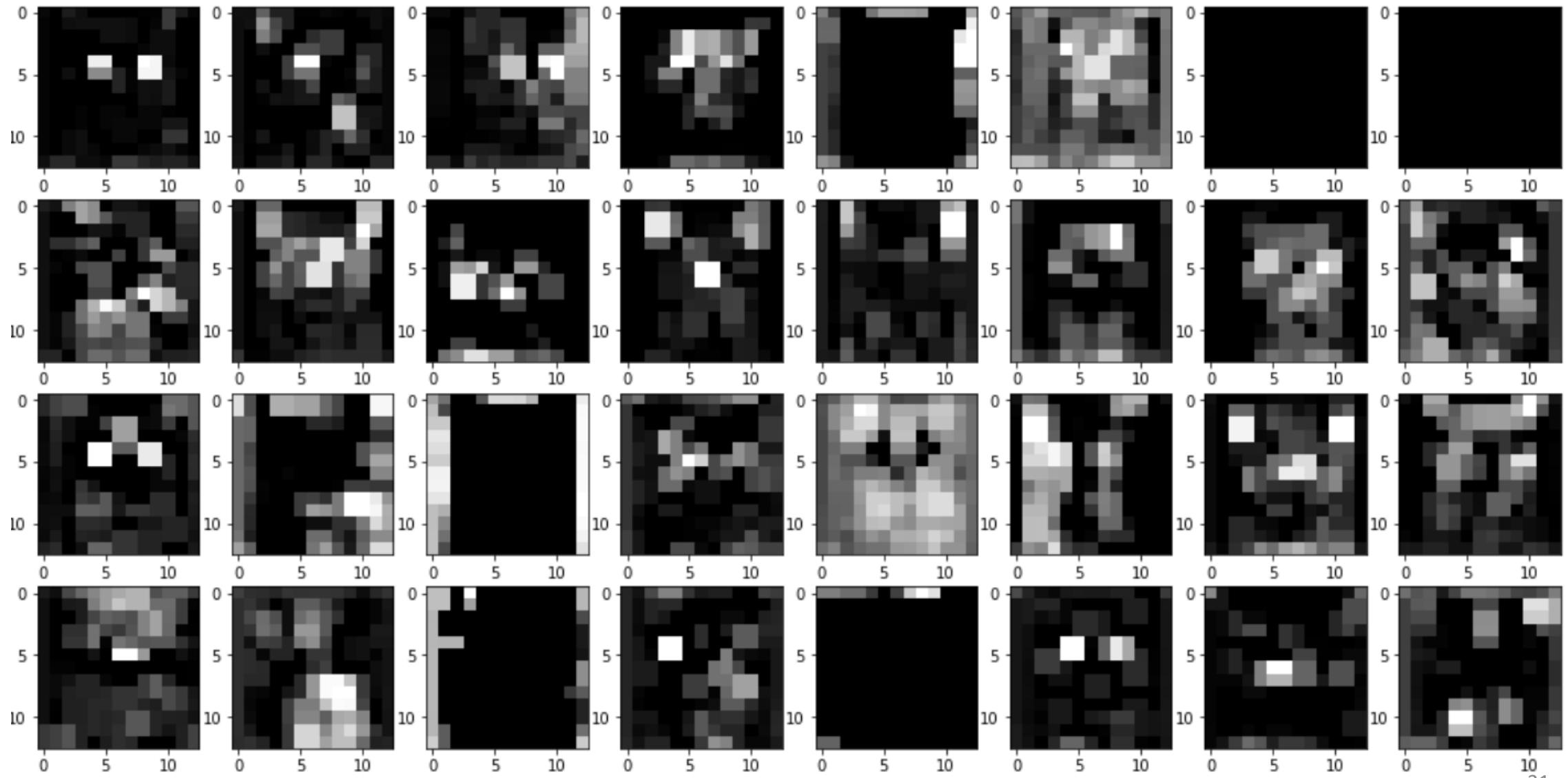
$$H_{out} = \frac{H_{in} + 2 \times padding - kernel\ size}{Stride} + 1$$

AlexNet(

```
(features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=0, dilation=1, bias=True)
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=2, dilation=1, bias=True)
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=1, dilation=1, bias=True)
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=1, dilation=1, bias=True)
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=1, dilation=1, bias=True)
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
)
```



Repeat – max pooling



Repeat – 3rd convolution

```
[17]: conv3 = model.features[6]
conv3_out = conv3(conv2_out1)
print(conv3_out.shape)
imgArray=conv3_out[0].data.cpu().numpy()
fig=plt.figure(figsize=(18, 9))
for i in range(32): #visualize the first 32 channels
    fig.add_subplot(4, 8, i+1)
    plt.imshow(imgArray[i], cmap='gray')
plt.show()

torch.Size([1, 384, 13, 13])
```

After convolution, the output feature map has 394 channels

$$\frac{13 + 2 \times 1 - 3}{1} + 1 = 13$$

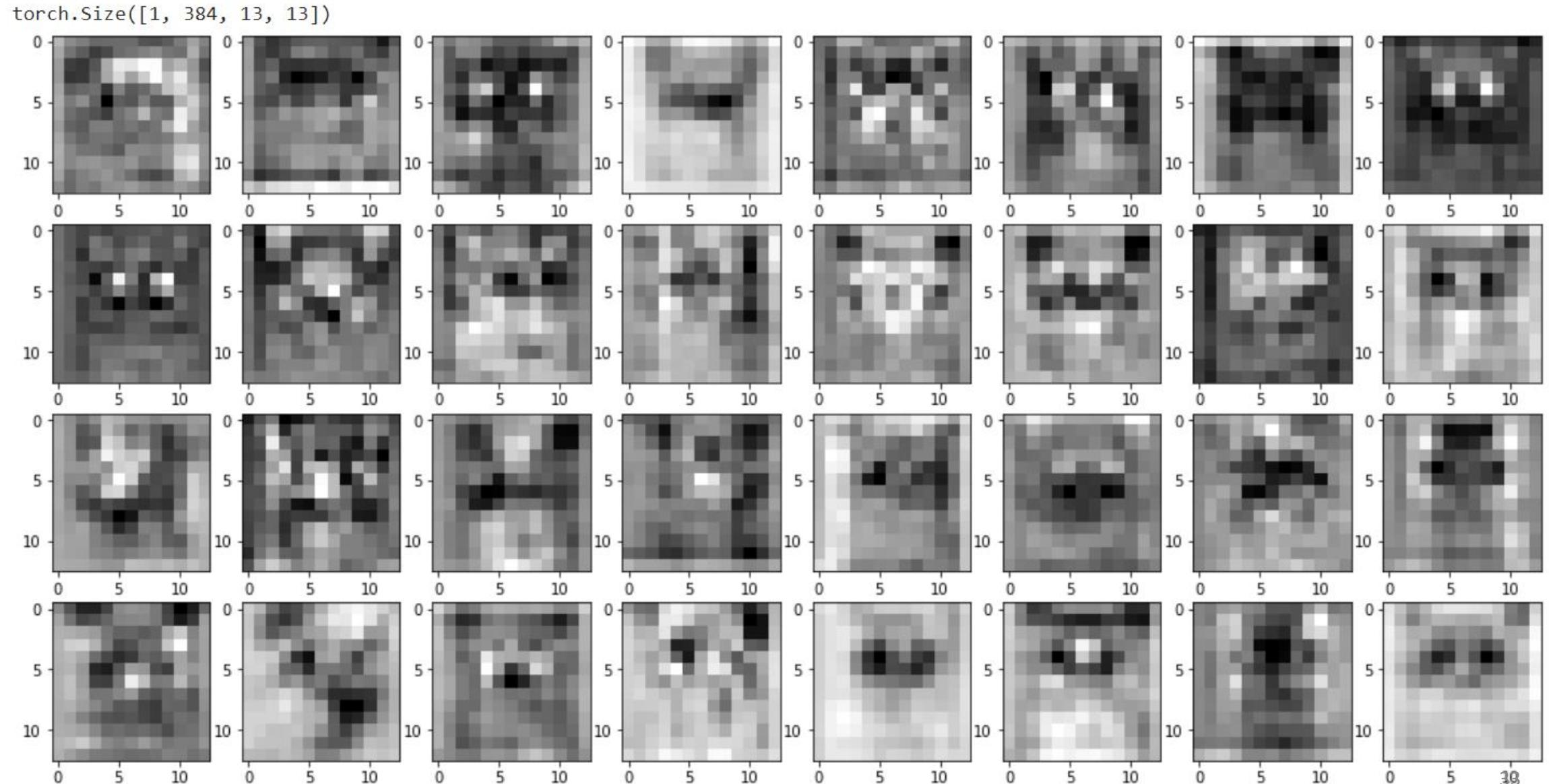
$$H_{out} = \frac{H_{in} + 2 \times padding - kernel\ size}{Stride} + 1$$

394 filters, each has 192 channels, are applied to the input feature map (with 192 channels)

```
AlexNet(
    features: Sequential(
        (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padd
        (1): ReLU(inplace=True)
        (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=
        (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padd
        (4): ReLU(inplace=True)
        (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=
        (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), pad
        (7): ReLU(inplace=True)
        (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), pad
        (9): ReLU(inplace=True)
        (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), pa
        (11): ReLU(inplace=True)
        (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation
```

)

Repeat – 3rd convolution



Repeat – max pooling

features[7, 8]

```
[18]: conv3_pooling = model.features[7:9]
conv3_out1 = conv3_pooling(conv3_out)
print(conv3_out1.shape)
imgArray=conv3_out1[0].data.cpu().numpy()
fig=plt.figure(figsize=(18, 9))
for i in range(32): #visualize the first 32 channels
    fig.add_subplot(4, 8, i+1)
    plt.imshow(imgArray[i], cmap='gray')
plt.show()

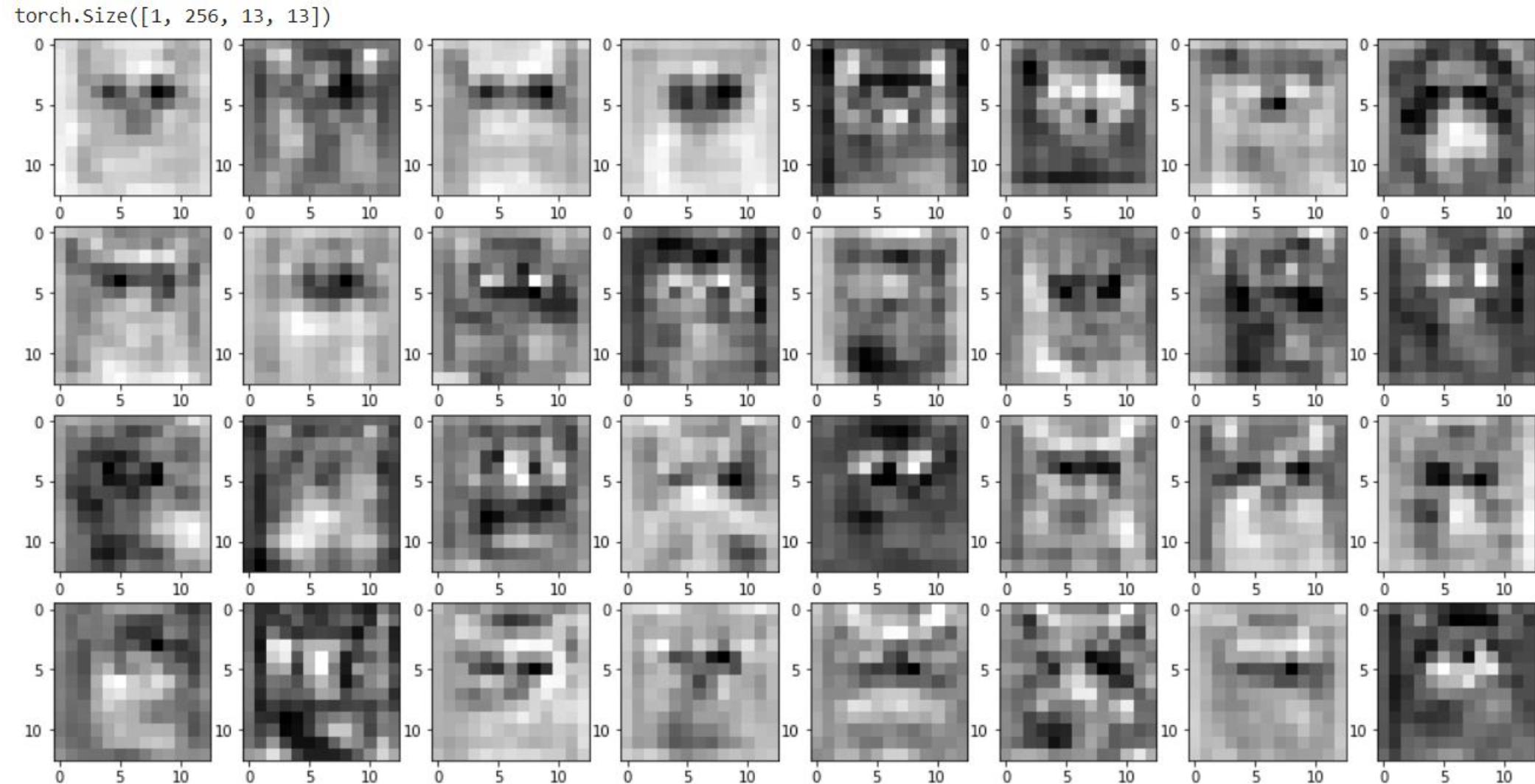
torch.Size([1, 256, 13, 13])
```

AlexNet(

```
(features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=0, dilation=1, groups=1, bias=True)
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=2, dilation=1, groups=1, bias=True)
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=1, dilation=1, groups=1, bias=True)
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=1, dilation=1, groups=1, bias=True)
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=1, dilation=1, groups=1, bias=True)
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
)
```



Repeat – max pooling



Flatten

```
[19]: WholeConvLayers = model.features
out1 = WholeConvLayers(imageTensor.to(device))
print(out1.shape)

AvgPoolLayer = model.avgpool
out2 = AvgPoolLayer(out1)
print(out2.shape)

torch.Size([1, 256, 6, 6])
torch.Size([1, 256, 6, 6])
```

After last convolution and max pooling, the output feature map has 256 channels

$$256 \times 6 \times 6 = 9216$$

```
(features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
)
(avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
(classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True) 9216
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
)
```



Practice – What does CNN learn?

- Run "7.2. What does CNN learn.ipynb"



What does CNN learn?

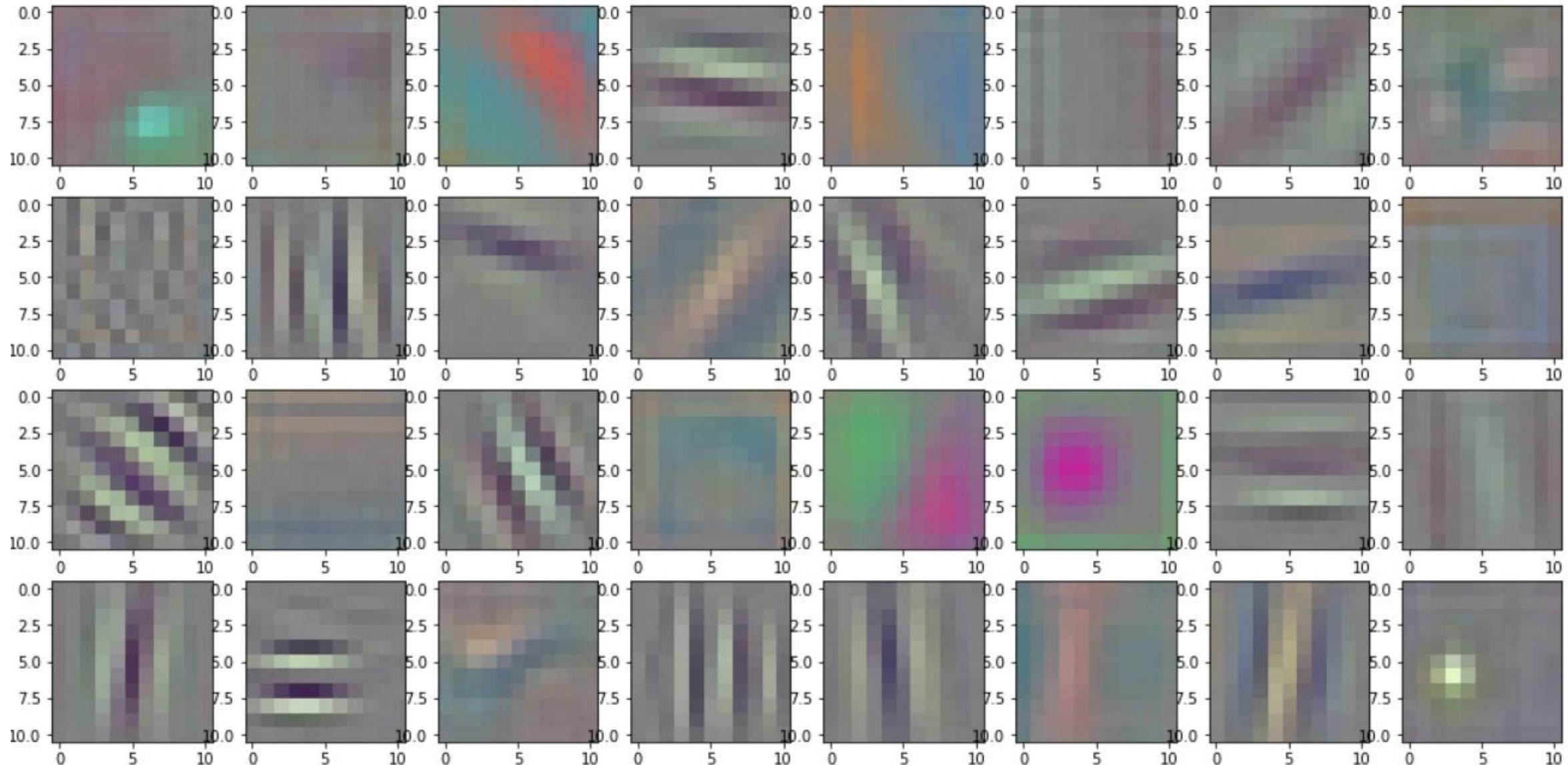
```
[4]: import numpy as np
      import matplotlib.pyplot as plt

conv1 = model.features[0]
weight1 = conv1.weight.data.cpu().numpy()
print(weight1.shape)
#(64, 3, 11, 11)

# Visualize the first 32 of the filter weights
fig=plt.figure(figsize=(18, 9))
for i in range(32):
    fig.add_subplot(4, 8, i+1)
    w = weight1[i]
    ImgArray = np.zeros((w.shape[1], w.shape[2], 3))
    ImgArray[:, :, 0] = w[0, :, :]
    ImgArray[:, :, 1] = w[1, :, :]
    ImgArray[:, :, 2] = w[2, :, :]
    ImgArray = ImgArray*0.5+0.5 # convert[-1, 1] to [0, 1]
    plt.imshow(ImgArray)
plt.show()
```

(64, 3, 11, 11)

What does CNN learn?



What does CNN learn?

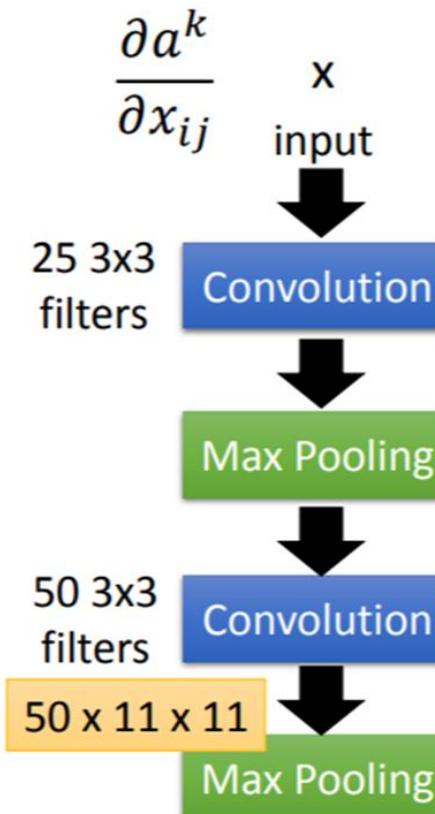
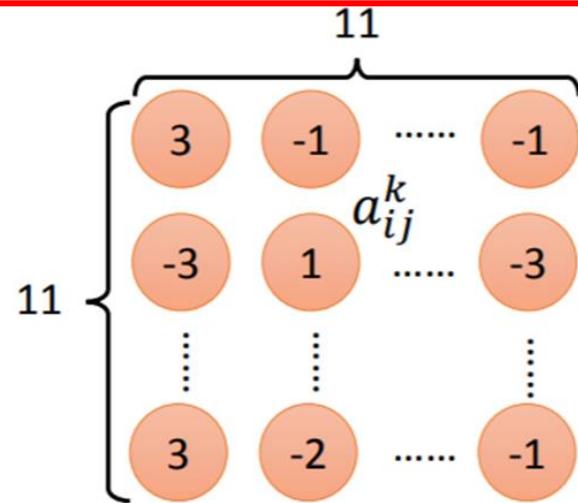
Only the weight of the 1st convolution filters can be directly visualized. How to interpret the filter weights of other convolution layers?

How to use
gradient ascent to
implement this in
PyTorch?

The output of the k-th filter is a
 11×11 matrix.

Degree of the activation
of the k-th filter: $a^k = \sum_{i=1}^{11} \sum_{j=1}^{11} a_{ij}^k$

$$x^* = \arg \max_x a^k \text{ (gradient ascent)}$$



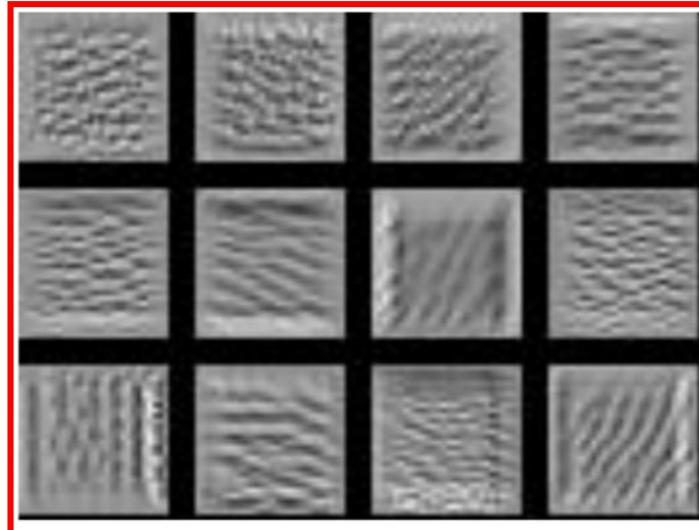
What does CNN learn?

With MNIST data set, in the convolution layer, the filters detects a particular texture pattern.

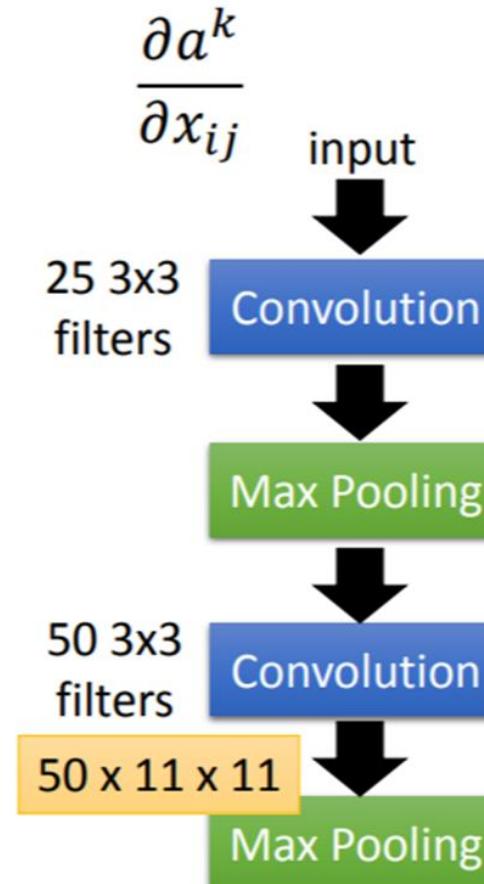
The output of the k-th filter is a 11×11 matrix.

Degree of the activation of the k-th filter: $a^k = \sum_{i=1}^{11} \sum_{j=1}^{11} a_{ij}^k$

$$x^* = \arg \max_x a^k \text{ (gradient ascent)}$$



How to implement this in PyTorch?



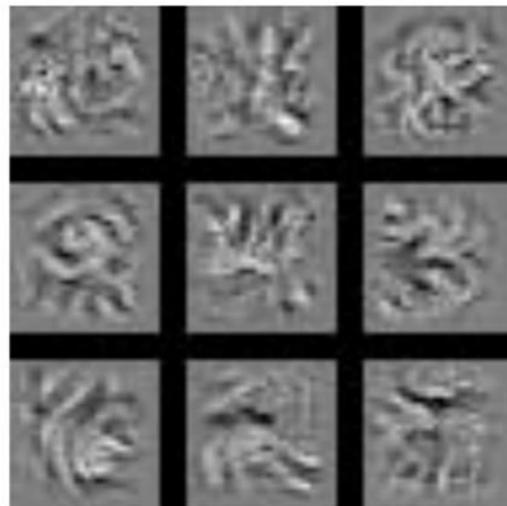
For each filter

What does CNN learn?

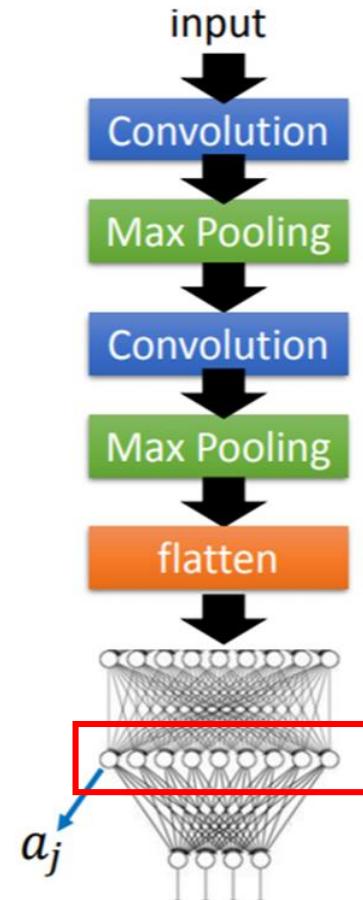
In the hidden layer of the fully-connected NN, each neuron detects an overall pattern in the picture rather than a particular texture pattern.

Find an image maximizing the output of neuron:

$$x^* = \arg \max_x a^j$$

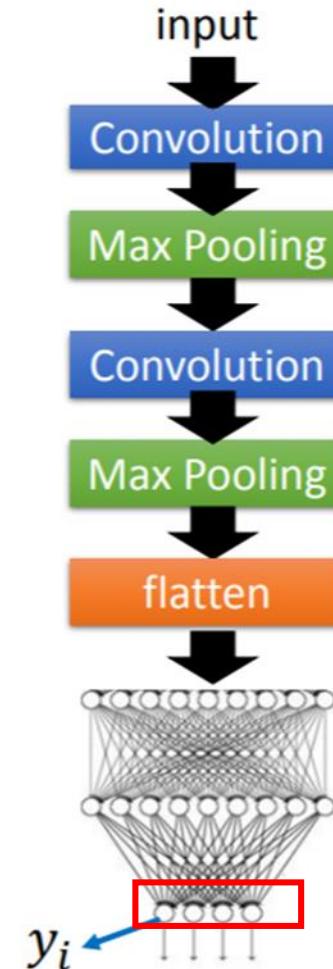
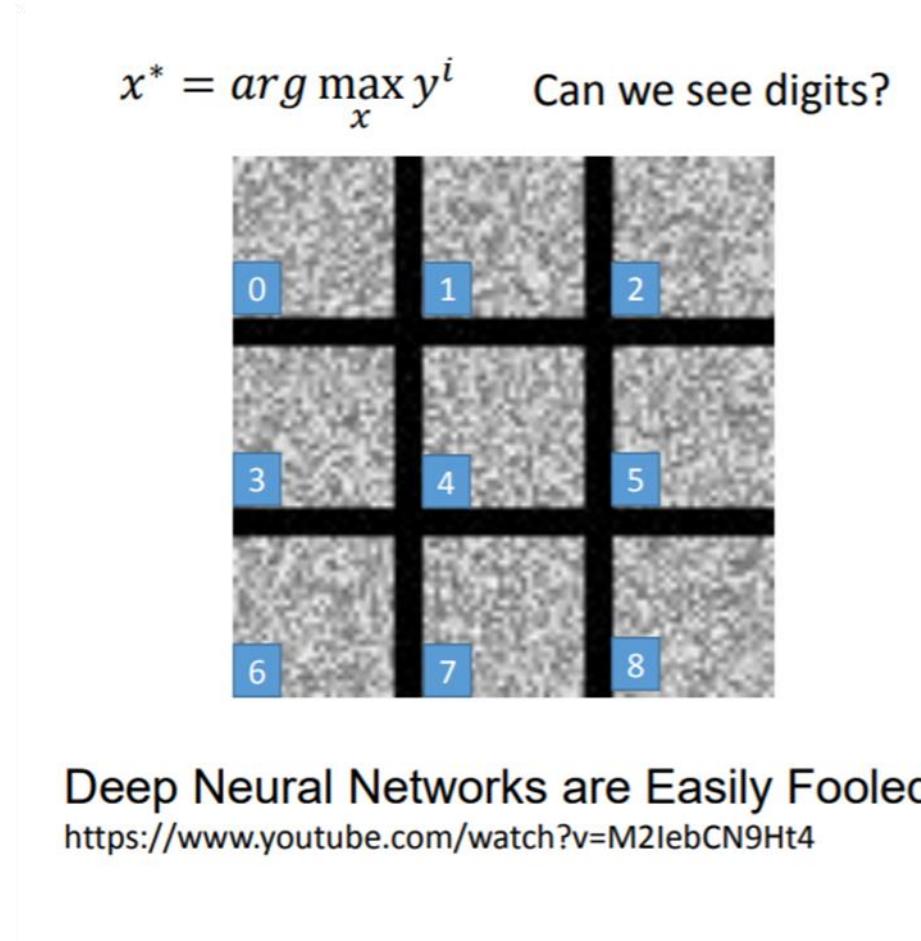


Each figure corresponds to a neuron



What does CNN learn?

If we watch the output layer node, it is easy to see that CNN is easily fooled.

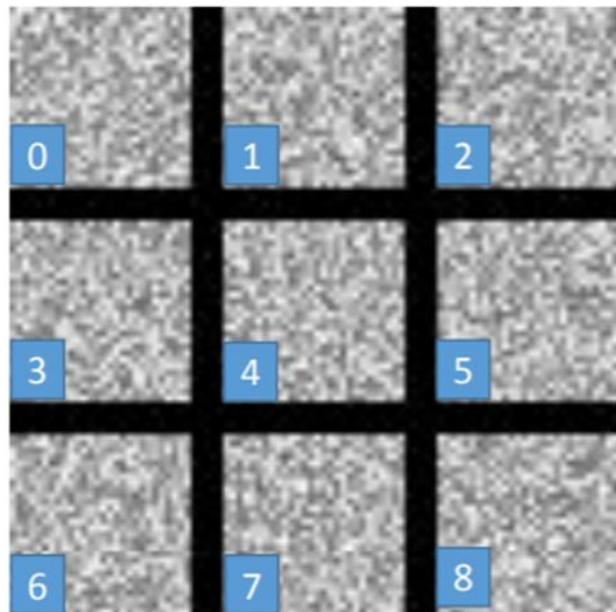


What does CNN learn?

Adding regularization to the objective function to force most pixels be "NO INK" .

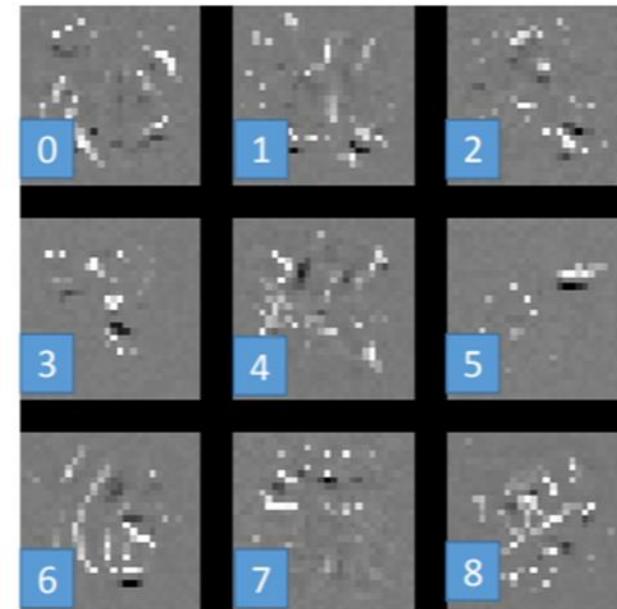
$$x^* = \arg \max_x y^i$$

Here white pixels indicate ink, and black pixels indicate "NO INK".



Over all
pixel values

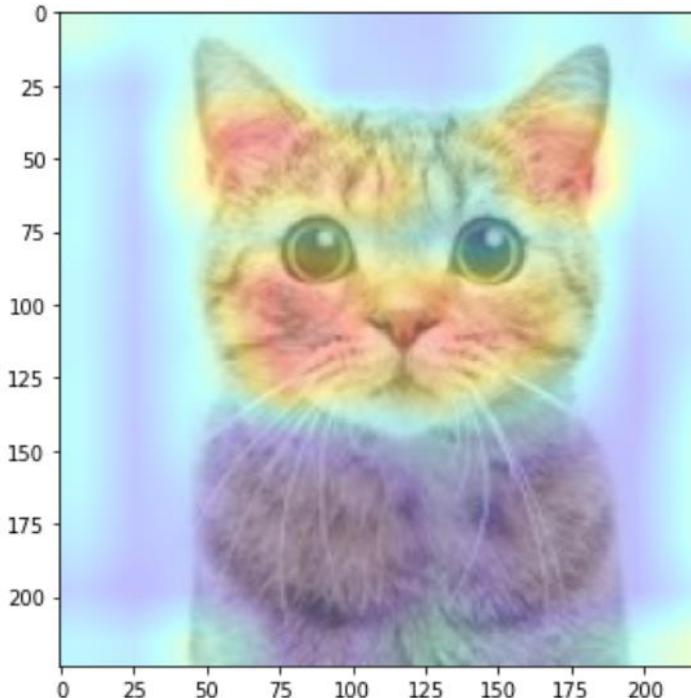
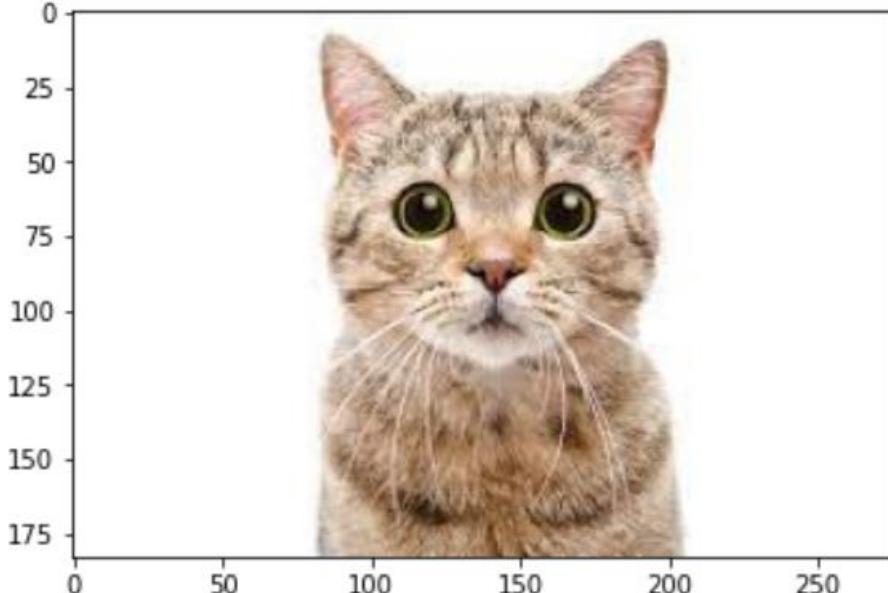
$$x^* = \arg \max_x \left(y^i - \sum_{i,j} |x_{ij}| \right)$$



L1 regularization to force $x_{ij}=0$, i.e., force most pixels to be black, NO INK (as only small part of the image has ink)

Practice – What does CNN learn?

- Run "7.3 GradCAM.ipynb"



HW4

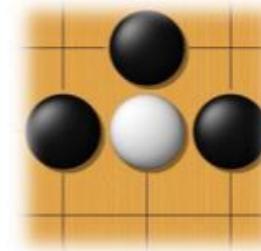
	Class index predicted by the model	Class index you assigned
AlexNet		
VGG		
ResNet18		



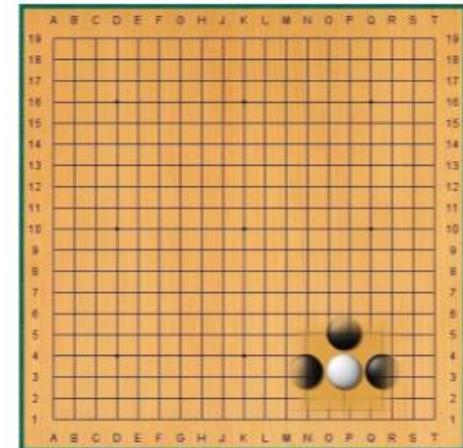
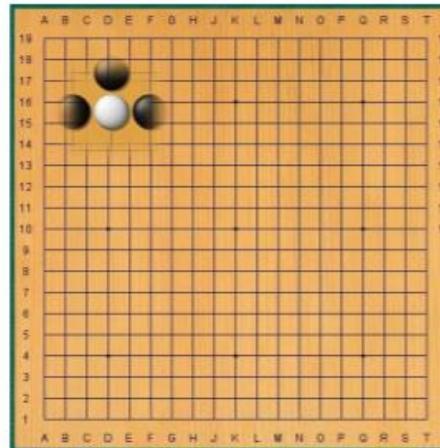
Use CNN in Alpha GO

- Some patterns are much smaller than the whole image

Alpha Go uses 5 x 5 for first layer



- The same patterns appear in different regions.



Use CNN in Alpha GO

Neural network architecture. The input to the policy network is a $19 \times 19 \times 48$ image stack consisting of 48 feature planes. The first hidden layer zero pads the input into a 23×23 image, then convolves k filters of kernel size 5×5 with stride 1 with the input image and applies a rectifier nonlinearity. Each of the subsequent hidden layers 2 to 12 zero pads the respective previous hidden layer into a 21×21 image, then convolves k filters of kernel size 3×3 with stride 1, again followed by a rectifier nonlinearity. The final layer convolves 1 filter of kernel size 1×1 with stride 1 with a different bias for each position, and applies a softmax function. The Alpha Go does not use Max Pooling Extended Data Table 3 additionally show the results of training with $k = 128, 256$ and 384 filters.