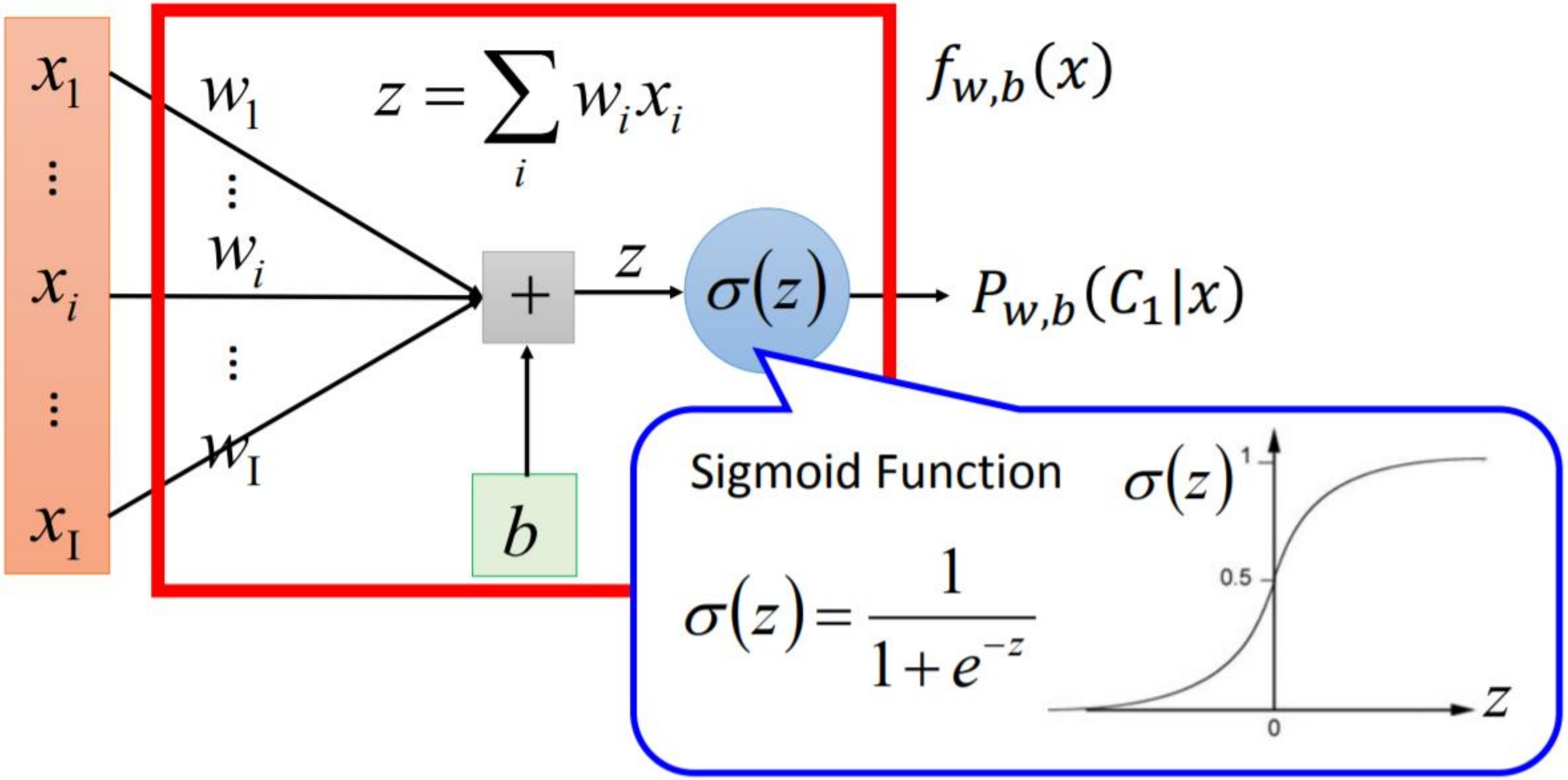
Each neuron is a classifier

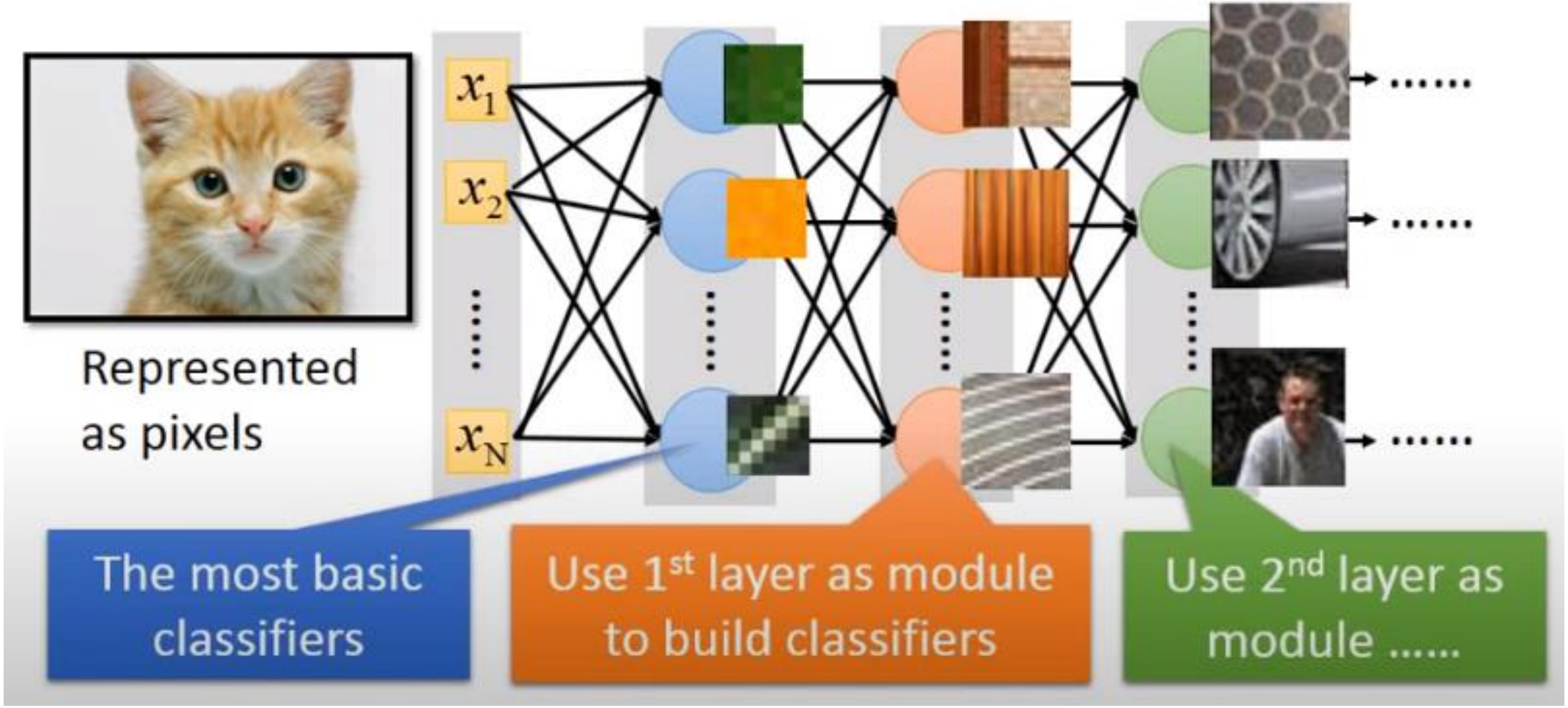
$$P(C_1|\,x) = \sigma(w \cdot x + b) = \sigma\left(\sum_i w_i x_i + b\right)$$



Reference: 李弘毅 ML Lecture 5 https://youtu.be/hSXFuypLukA

Each neuron in NN serves as a classifier

Each neuron classifies a particular pattern of previous layer in an image



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

Why CNN?

When CNN could be used to reduce the complexity of DNN?

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.

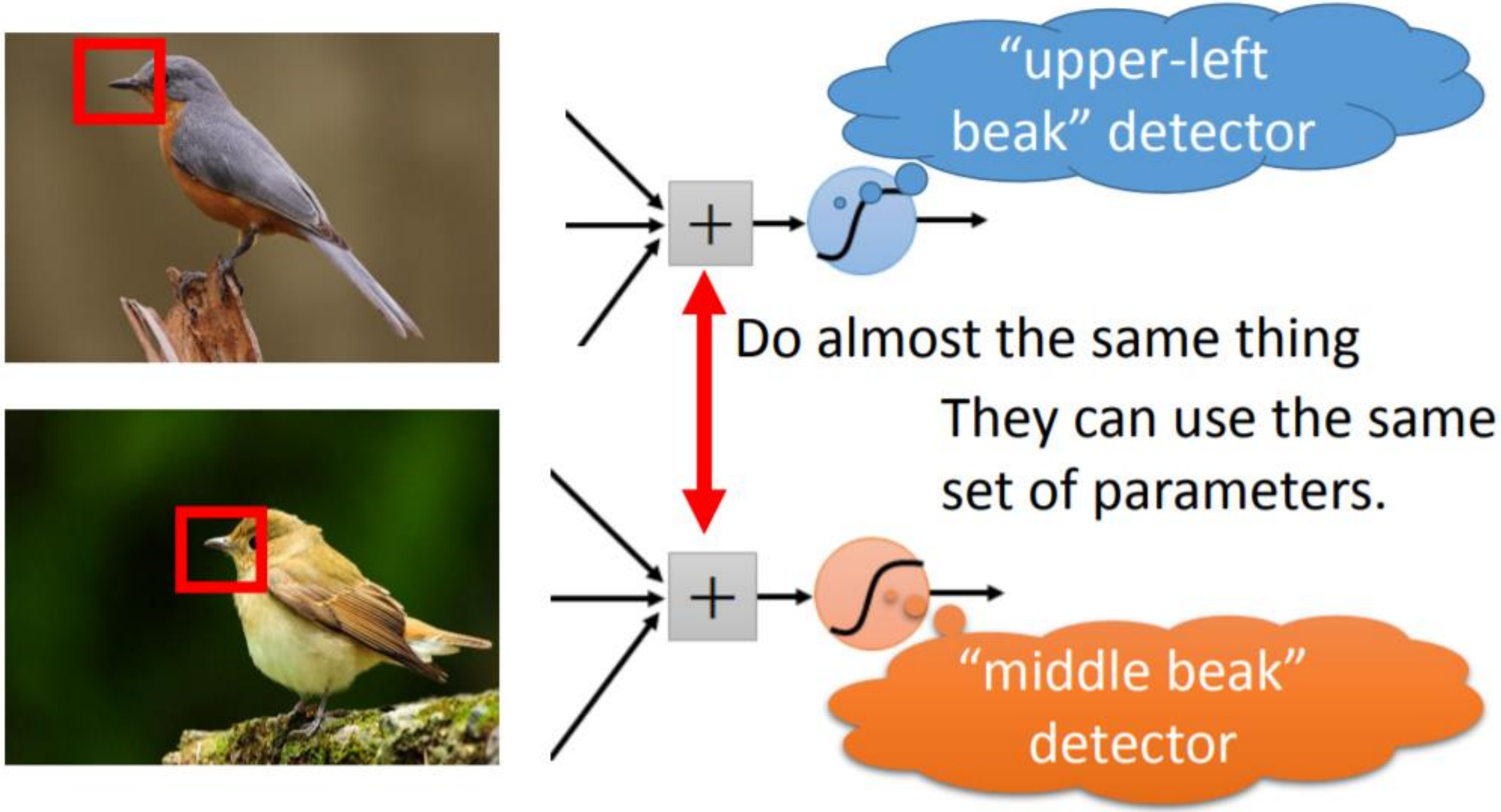
Connecting to small region with less parameters

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

Why CNN?

Property 2

- The same patterns appear in different regions.



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

Why CNN?

Property 3

- Subsampling the pixels will not change the object

bird

subsampling

bird

We can subsample the pixels to make image smaller

→ Less parameters for the network to process the image

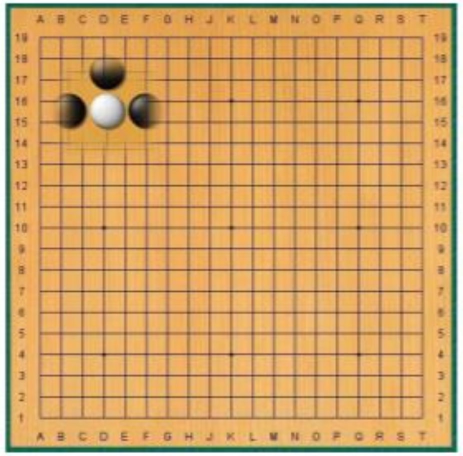
Reference: 李弘毅 ML Lecture 10 https://youtu.be/FrKWiRv254g

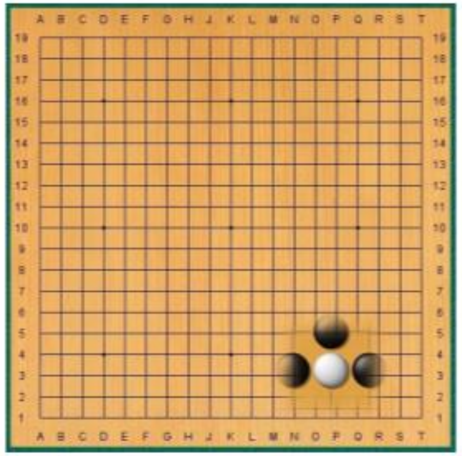
Example – use CNN in Alpha GO

- 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.





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

Example – 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 ……

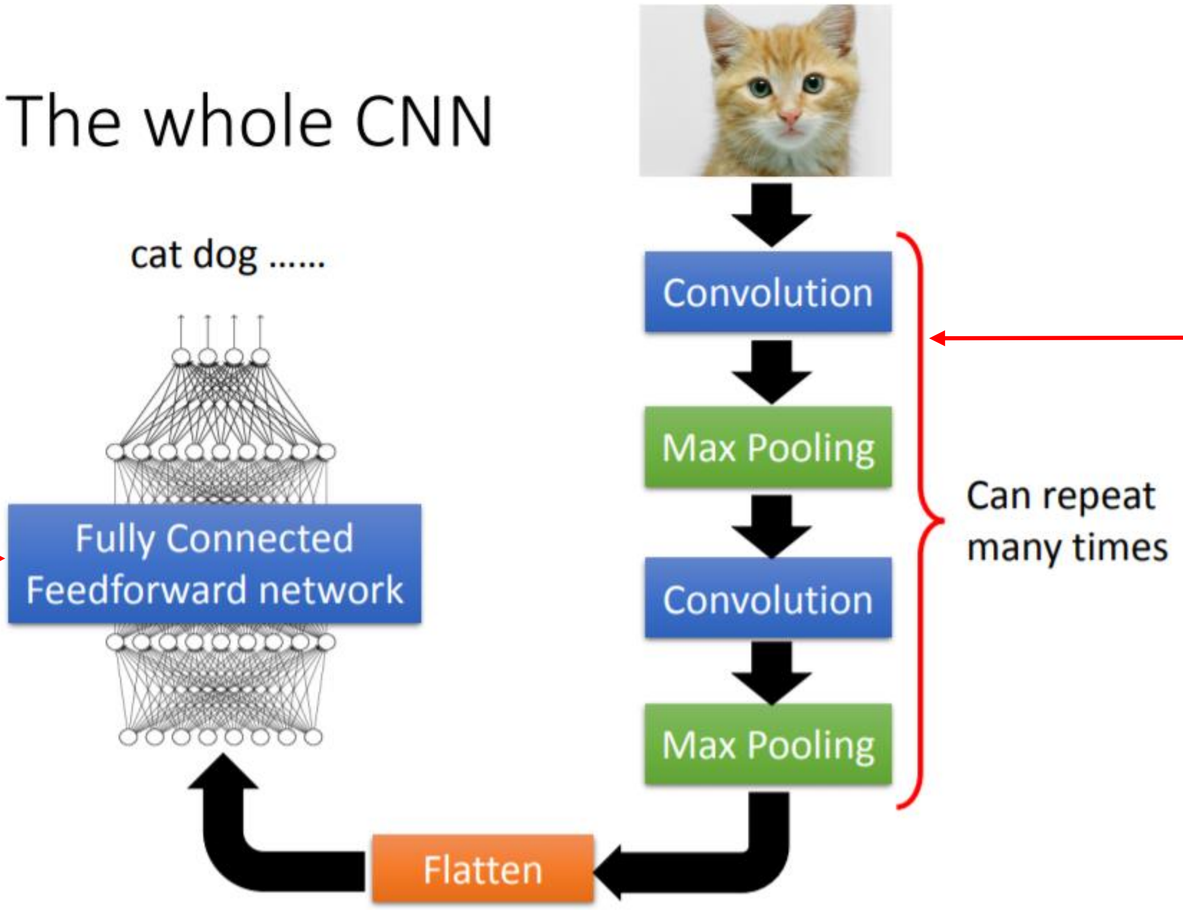
Extended Data Table 3 additionally show the results of training with $k = 128$, 256 and 384 filters.

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

Practice – CNN

- Run “7.1. CNN.ipynb"

The whole CNN



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

The whole CNN

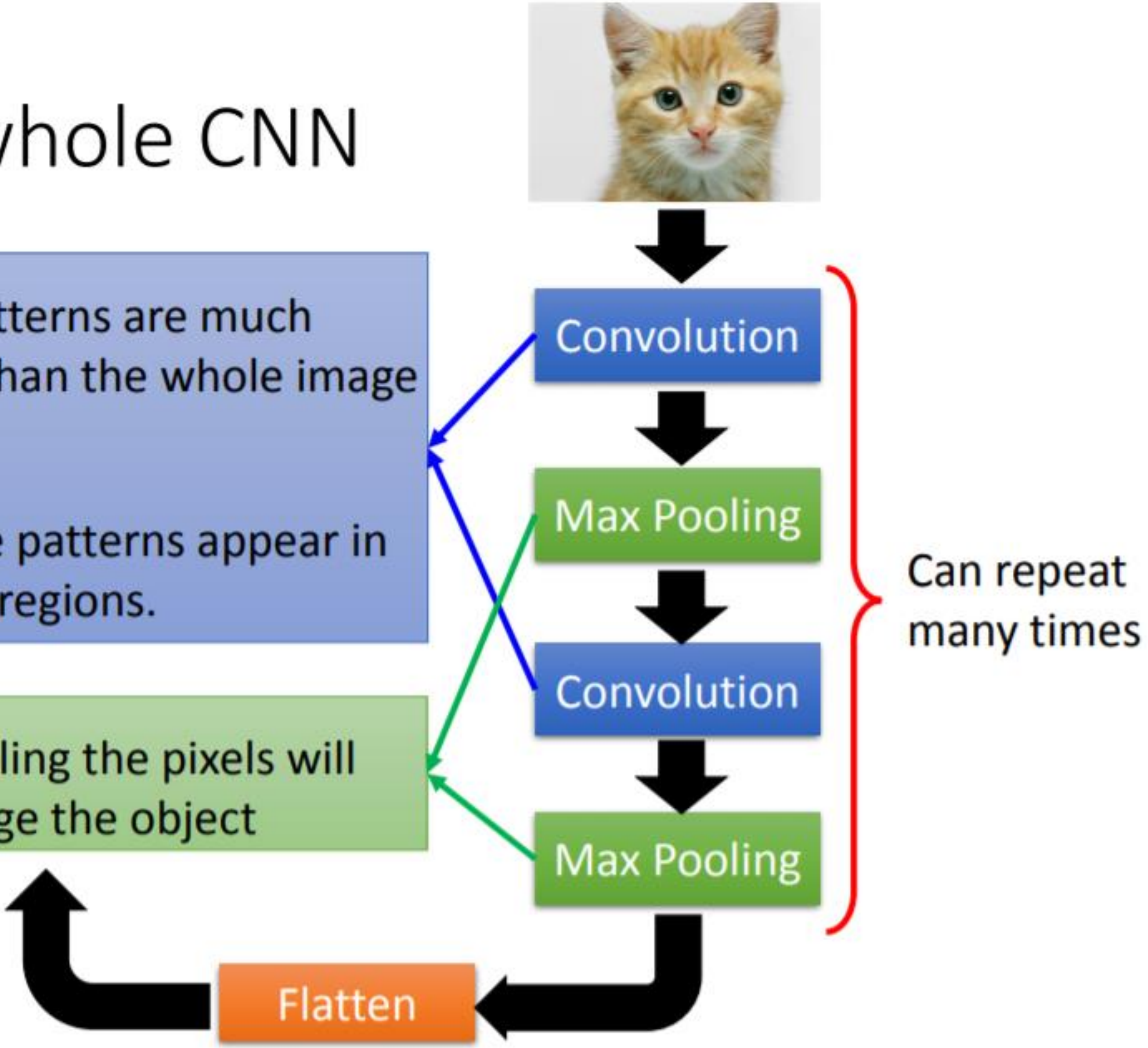
convolution

Property 1
- Some patterns are much smaller than the whole image

Property 2
- The same patterns appear in different regions.

pooling

Property 3
- Subsampling the pixels will not change the object



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

Practice – 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

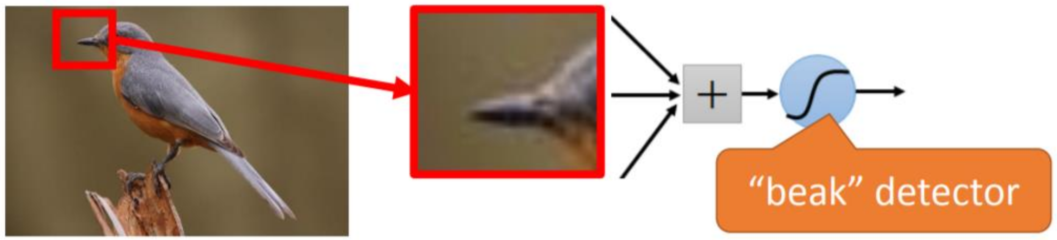
Each filter detects a small pattern (3 x 3).

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



Stride determines how filter shifts

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

stride=1

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 -1

Use Excel to verify!

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

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

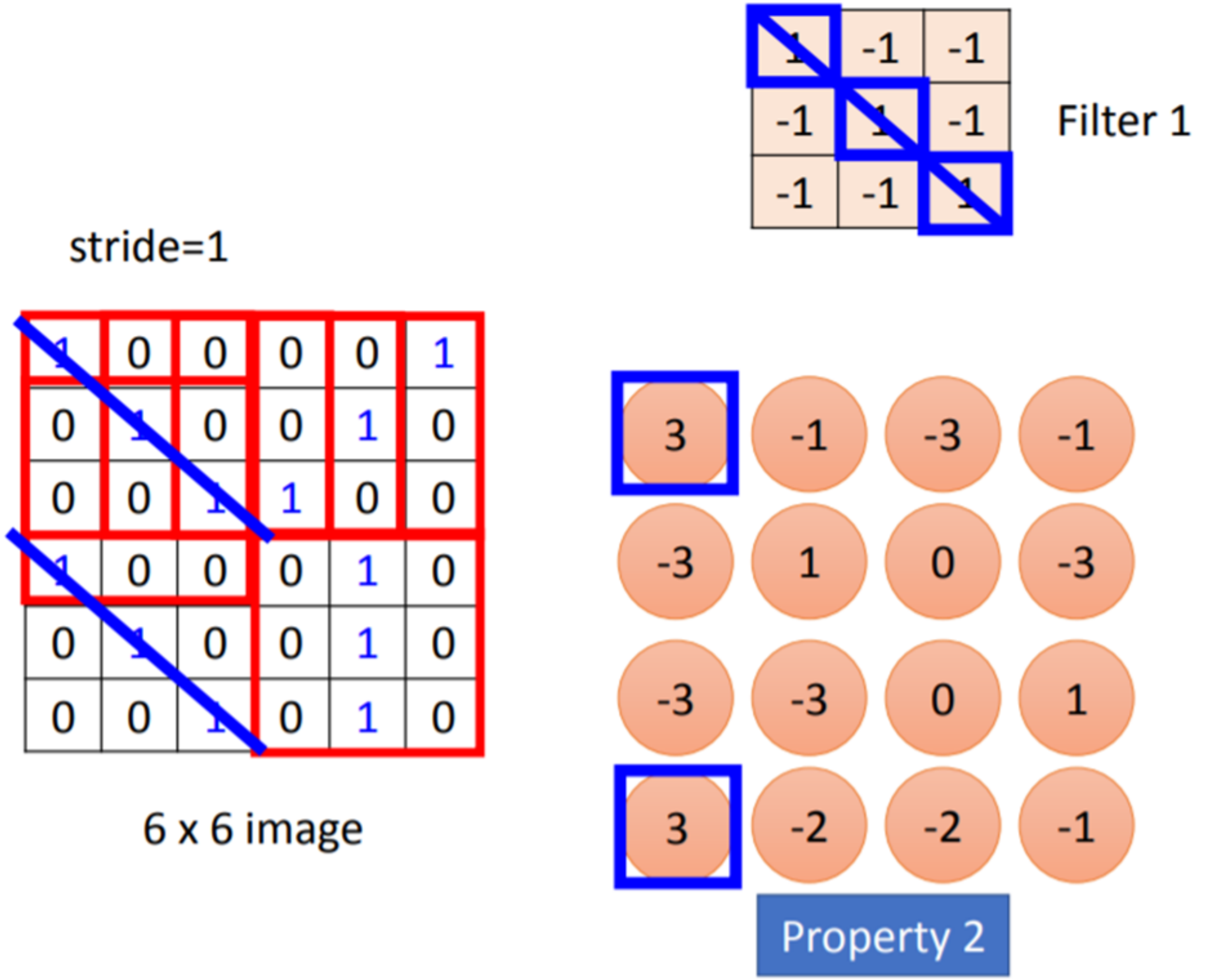
Filter 1

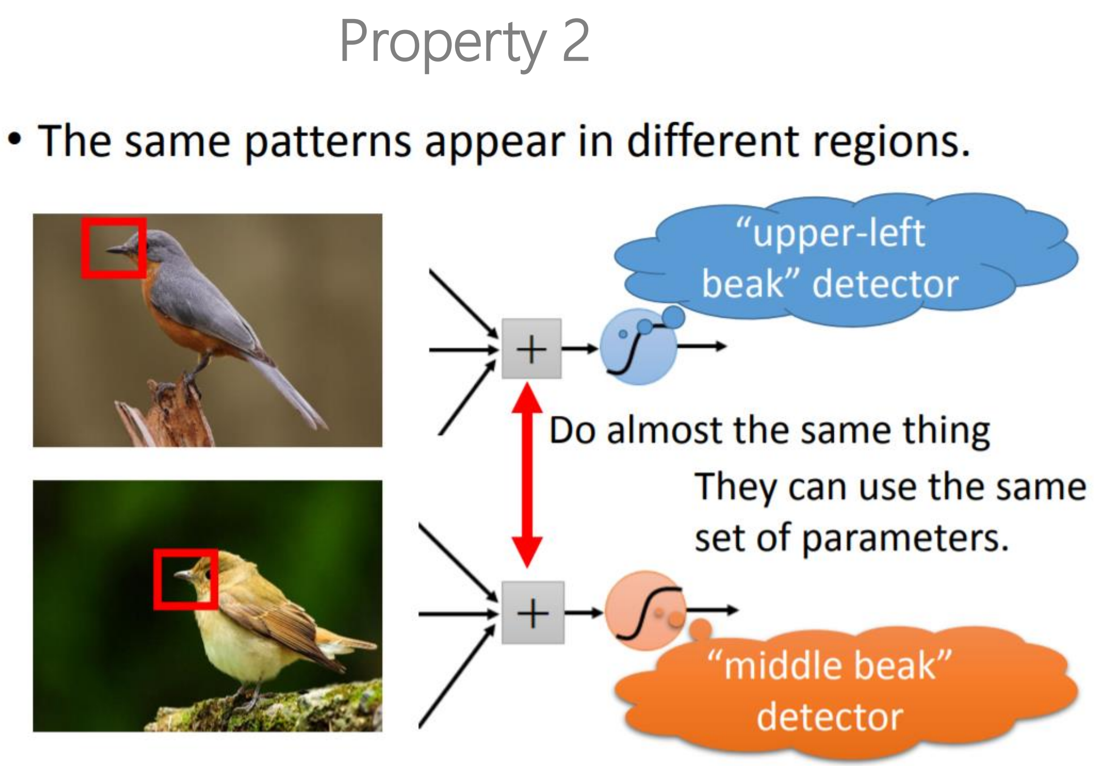
We set stride=1 below

Use Excel to verify!

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

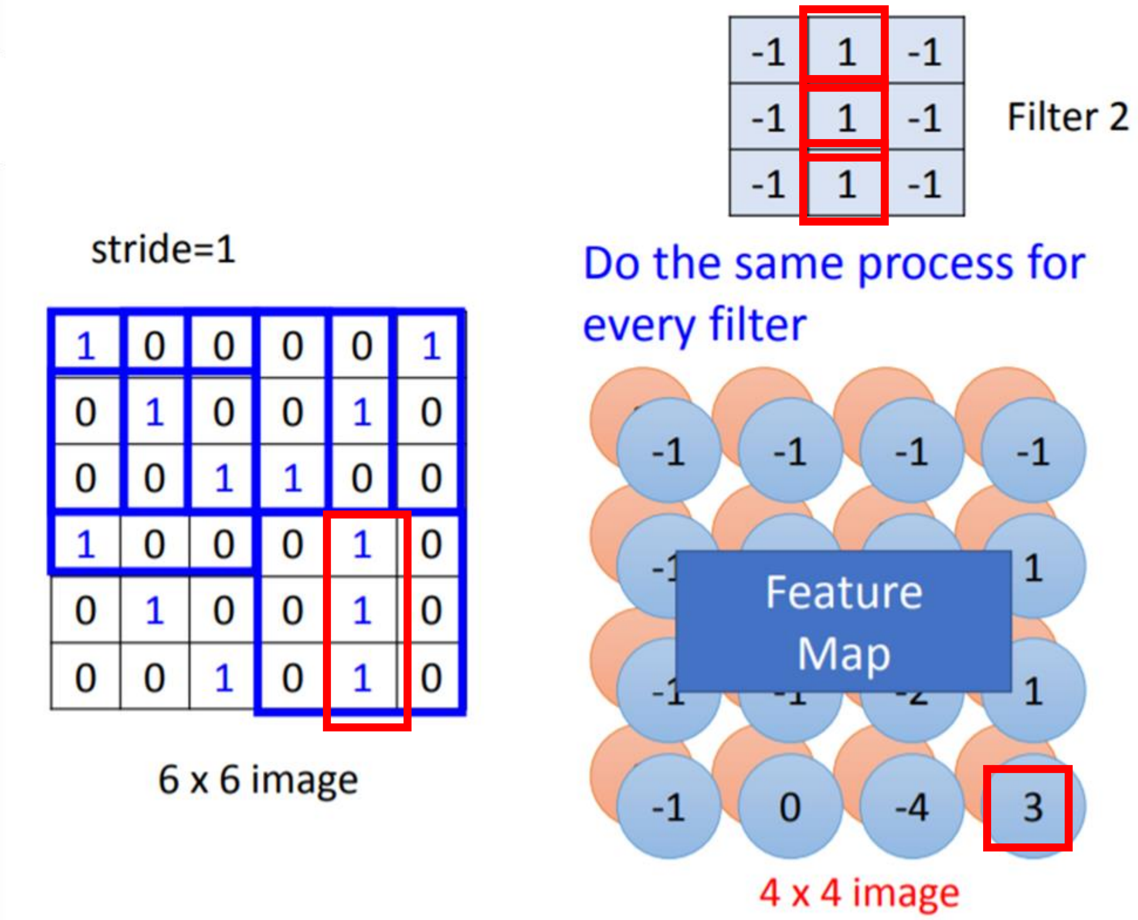
Filter searches a particular pattern in different regions





Feature maps

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

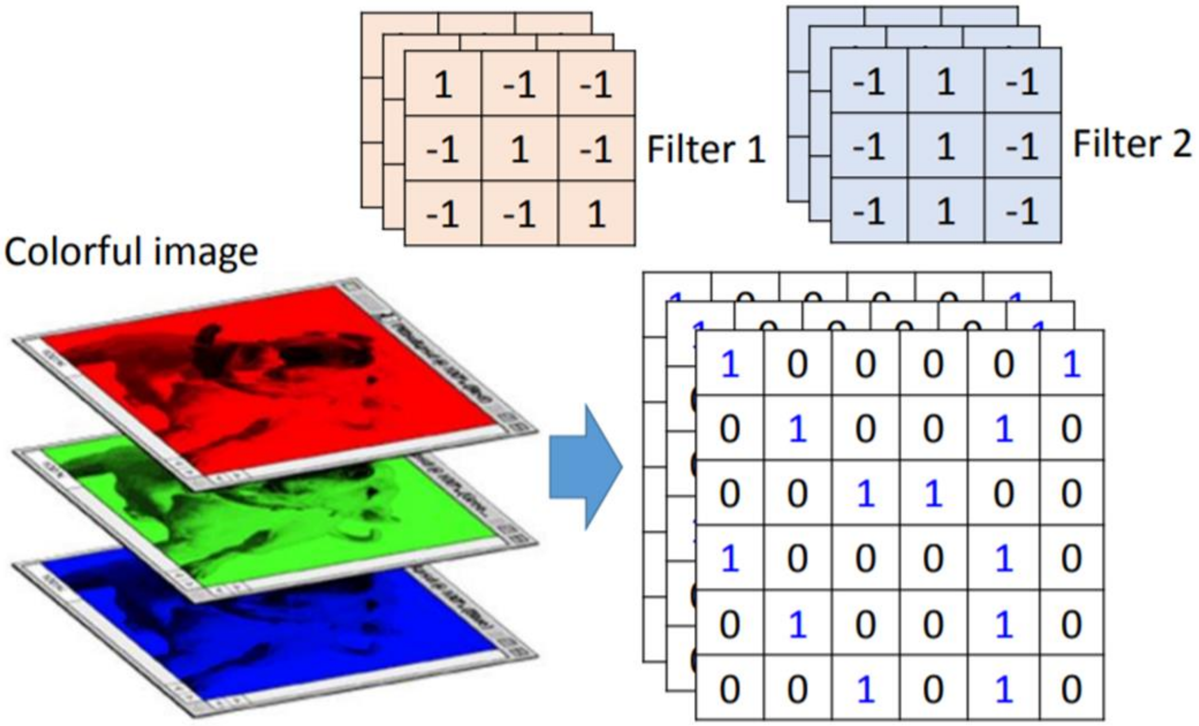



```
In [10]: conv1 = model.features[0]
         print(conv1)
         #InChannel=3(RGB),OutChannel=64, filter size

         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])
```

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

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
```

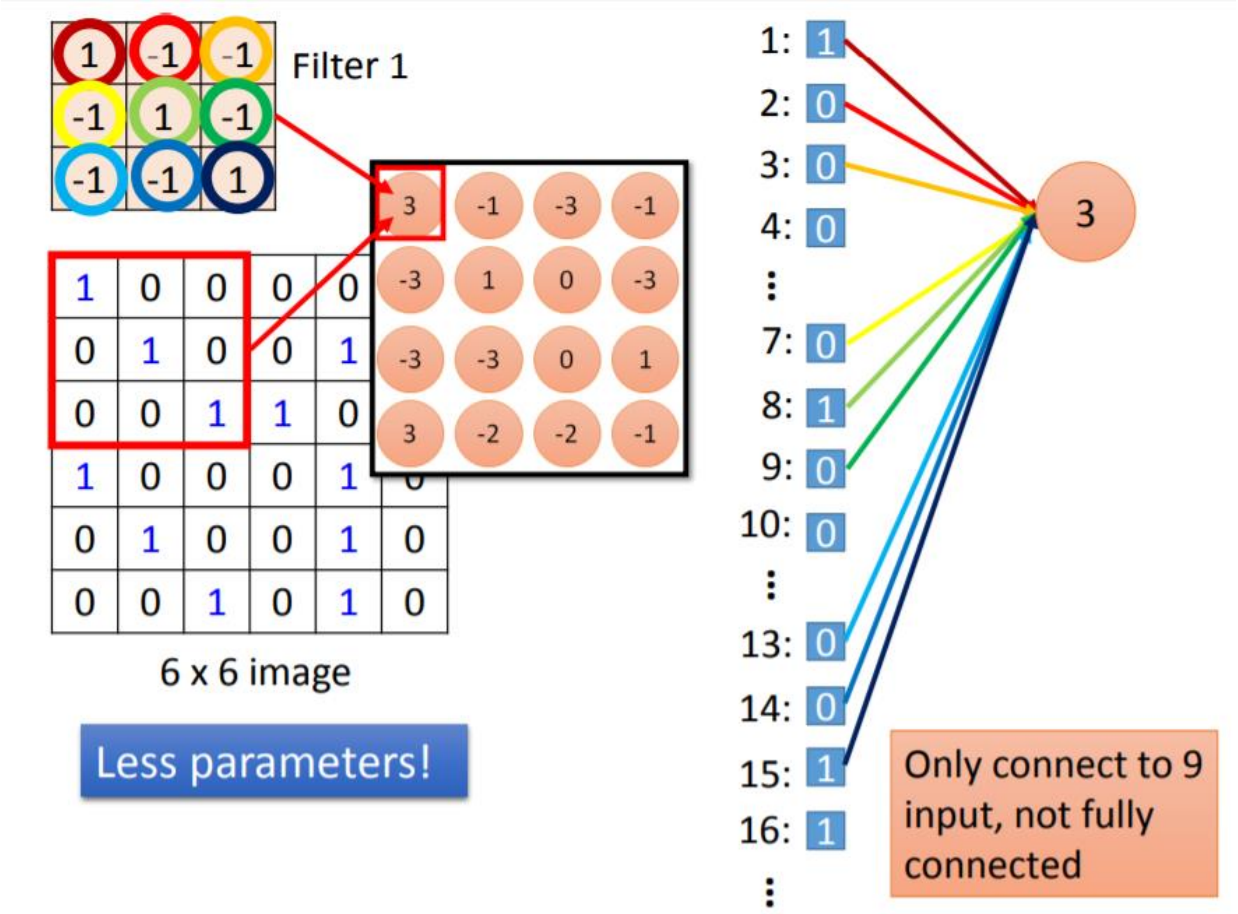
```
     (64, 3, 11, 11)
```

64 filters, each filter has
3 channels

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

Convolution represented as neural network

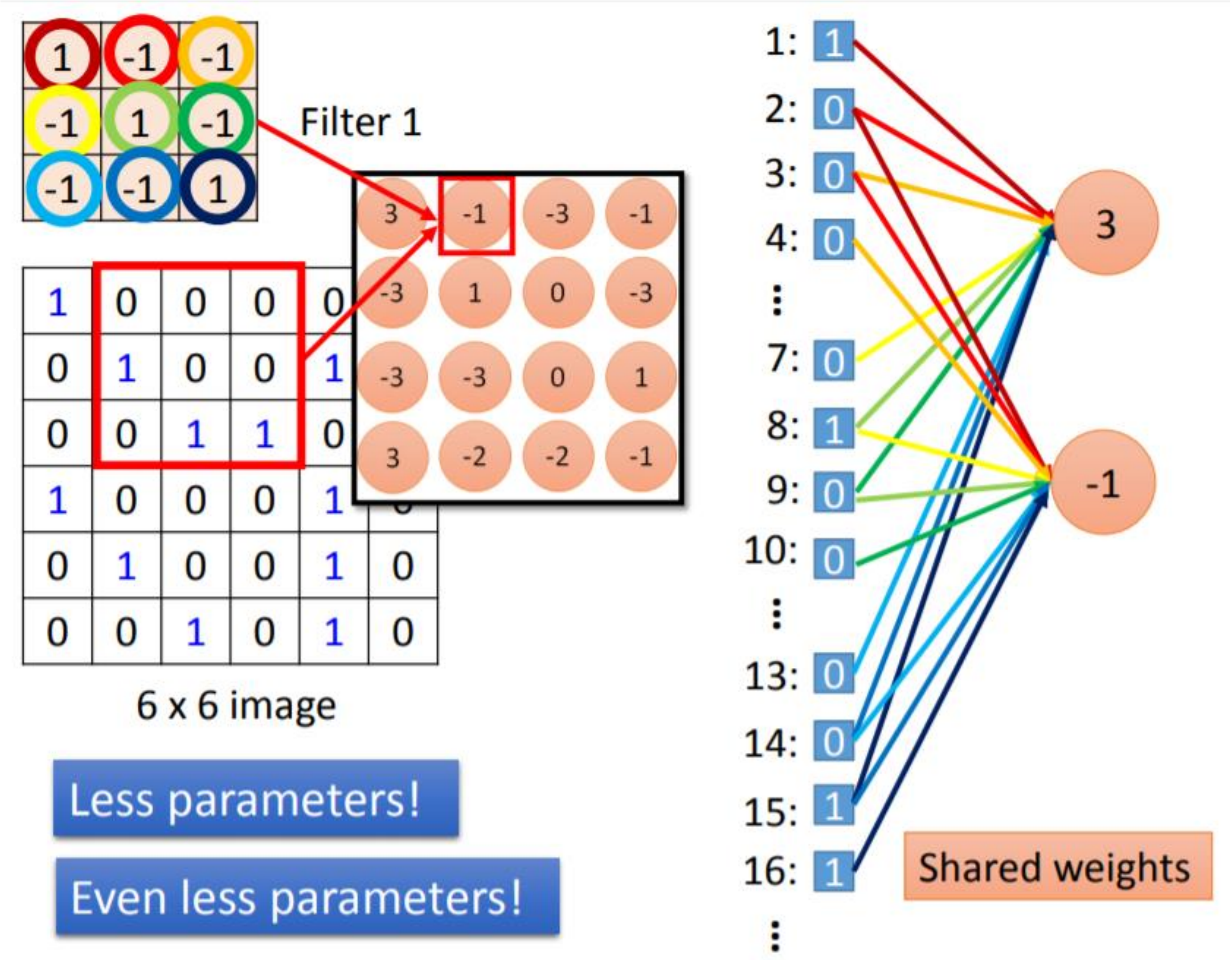
Convolution can be represented as partially connected NN, which has less parameters and is less complicated than the fully connected NN.



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

Convolution can be represented as neural network

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



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

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])
```

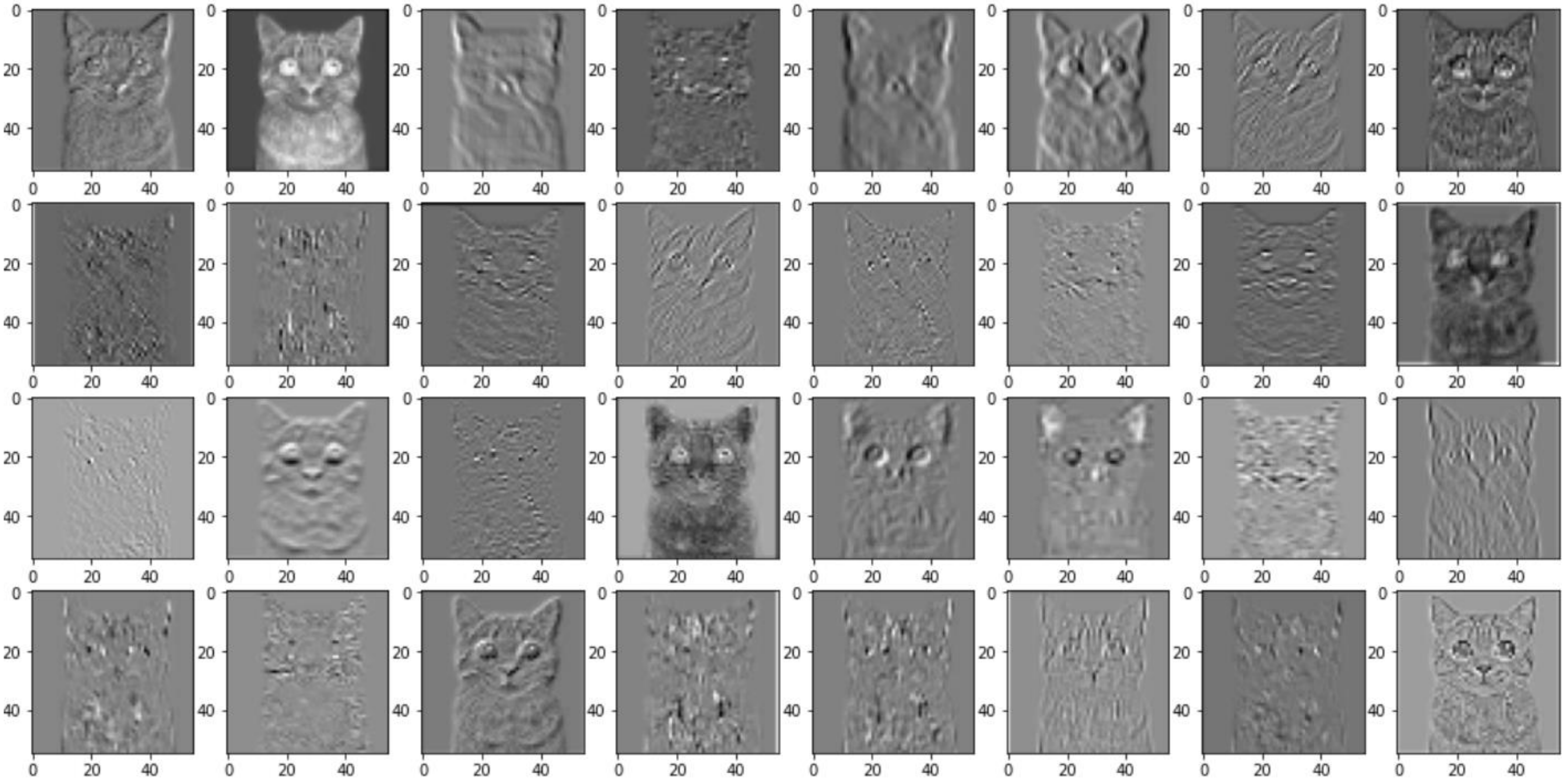
After convolution, the output image (feature map) has 64 channels

[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()
```

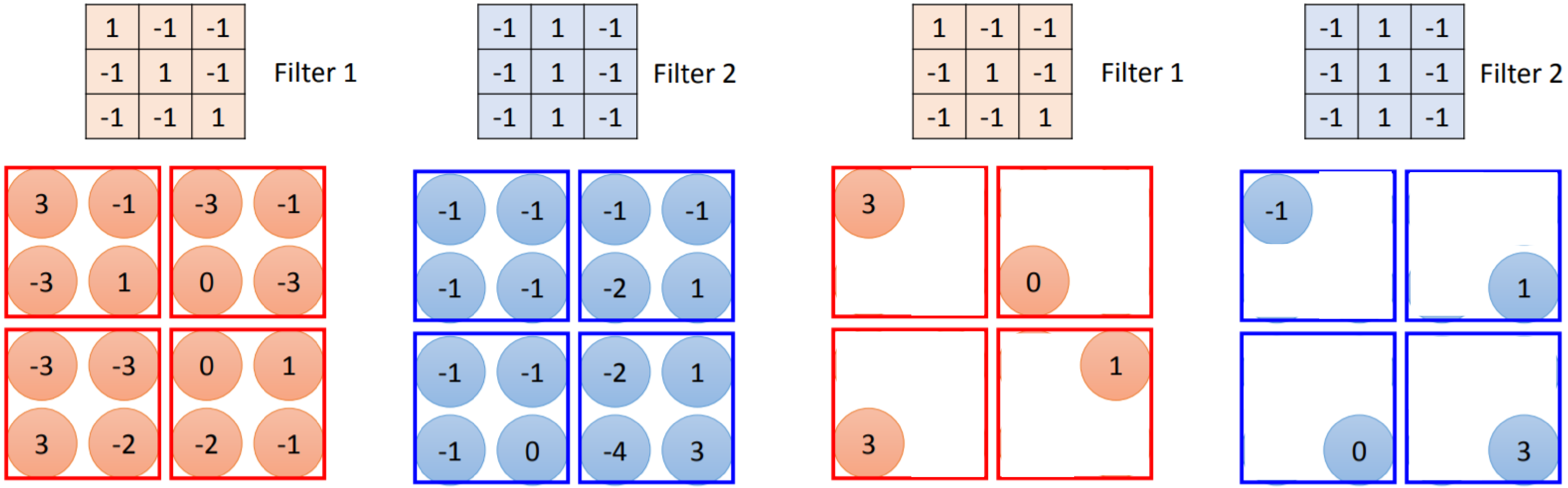
Feature map with *K* channels

K = No. of filters

Input image with 3 channels
Output image (feature map) with 64 channels



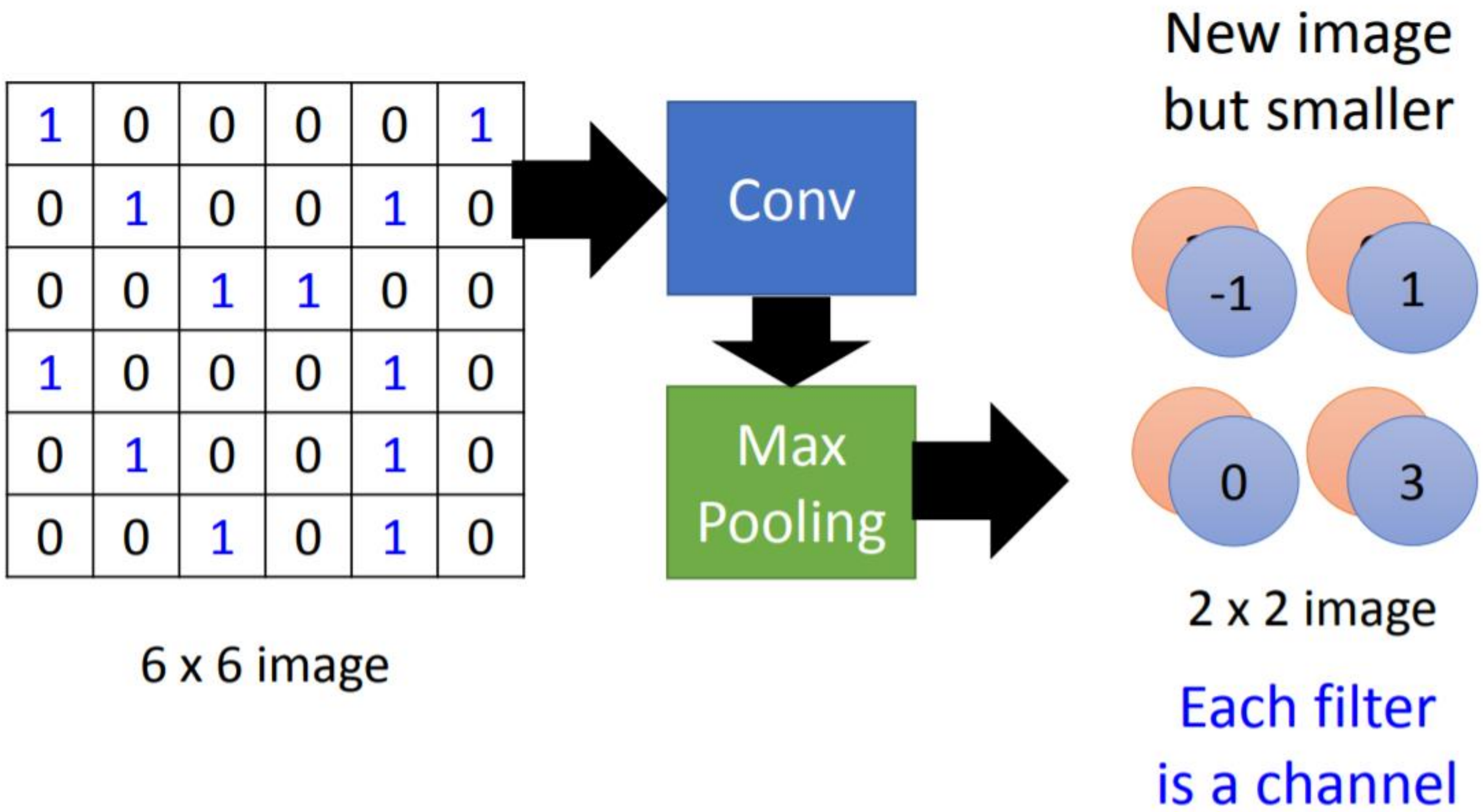
Max pooling



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

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.



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

Output image (feature map) with K channels

features[1, 2]

[14]:
```python
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 channe
  fig.add_subplot(4, 8, i+1)
  plt.imshow(imgArray[i], cmap='gray')
plt.show()
```

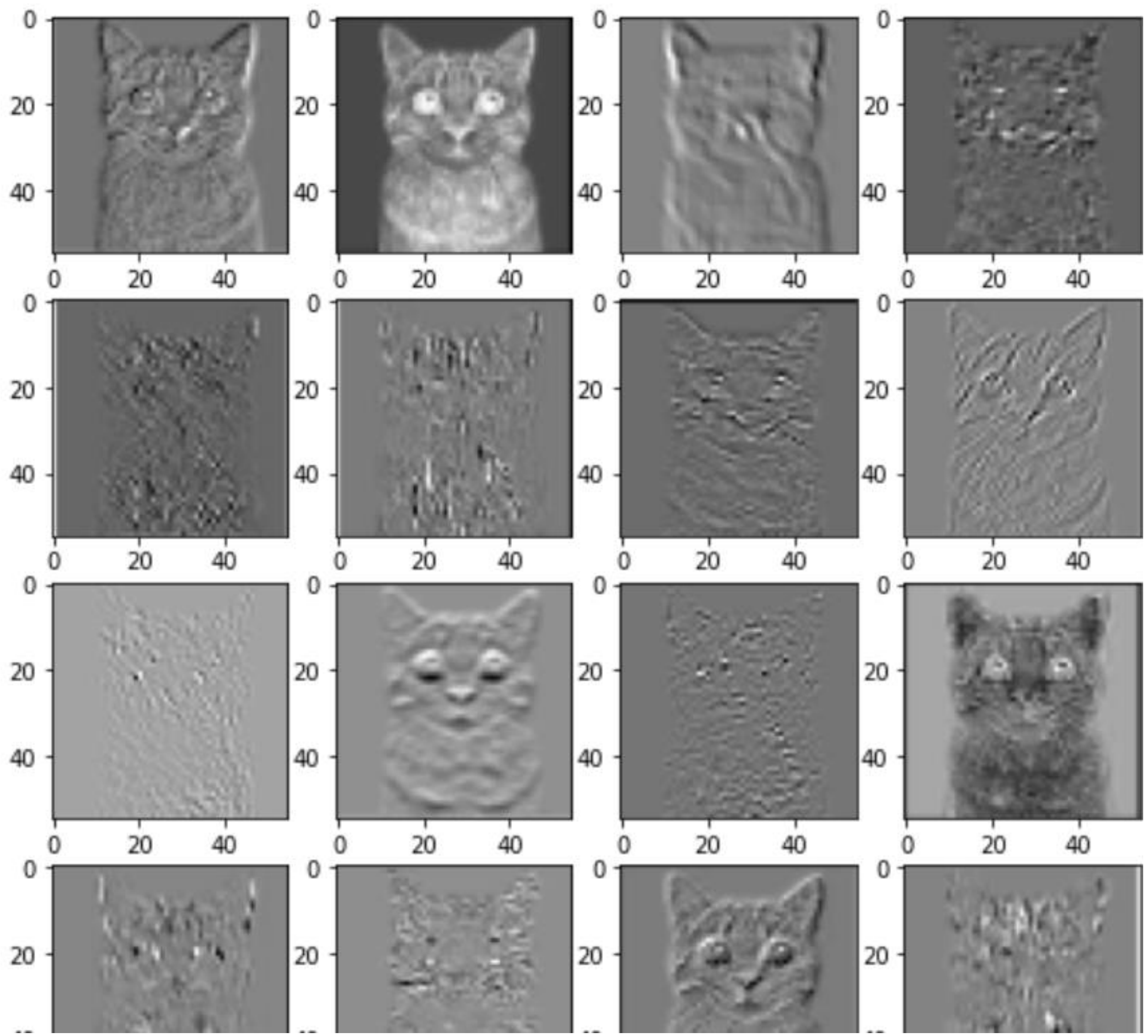
```
torch.Size([1, 64, 27, 27])
```

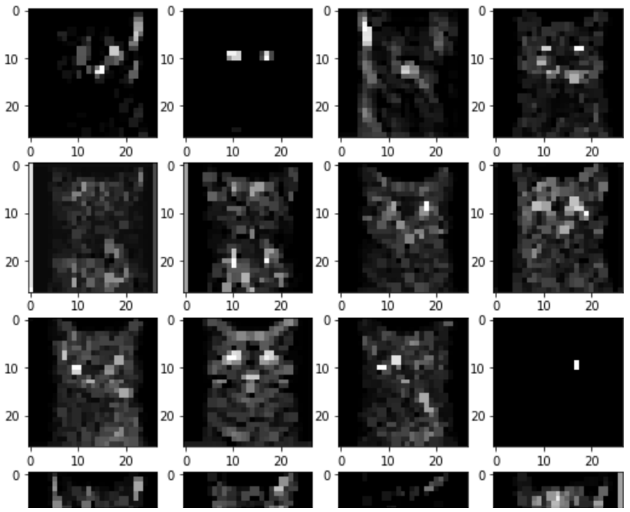
$$\frac{55 + 2 \times 2 - 3}{2} + 1 = 27$$

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

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), 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
  )
```

Output image (feature map) with K channels





Repeat – 2^{nd} convolution

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

```
[15]: conv2 = model.features[3]
      conv2_out = conv2(conv1_out1)
      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])
```

```
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
  )
```

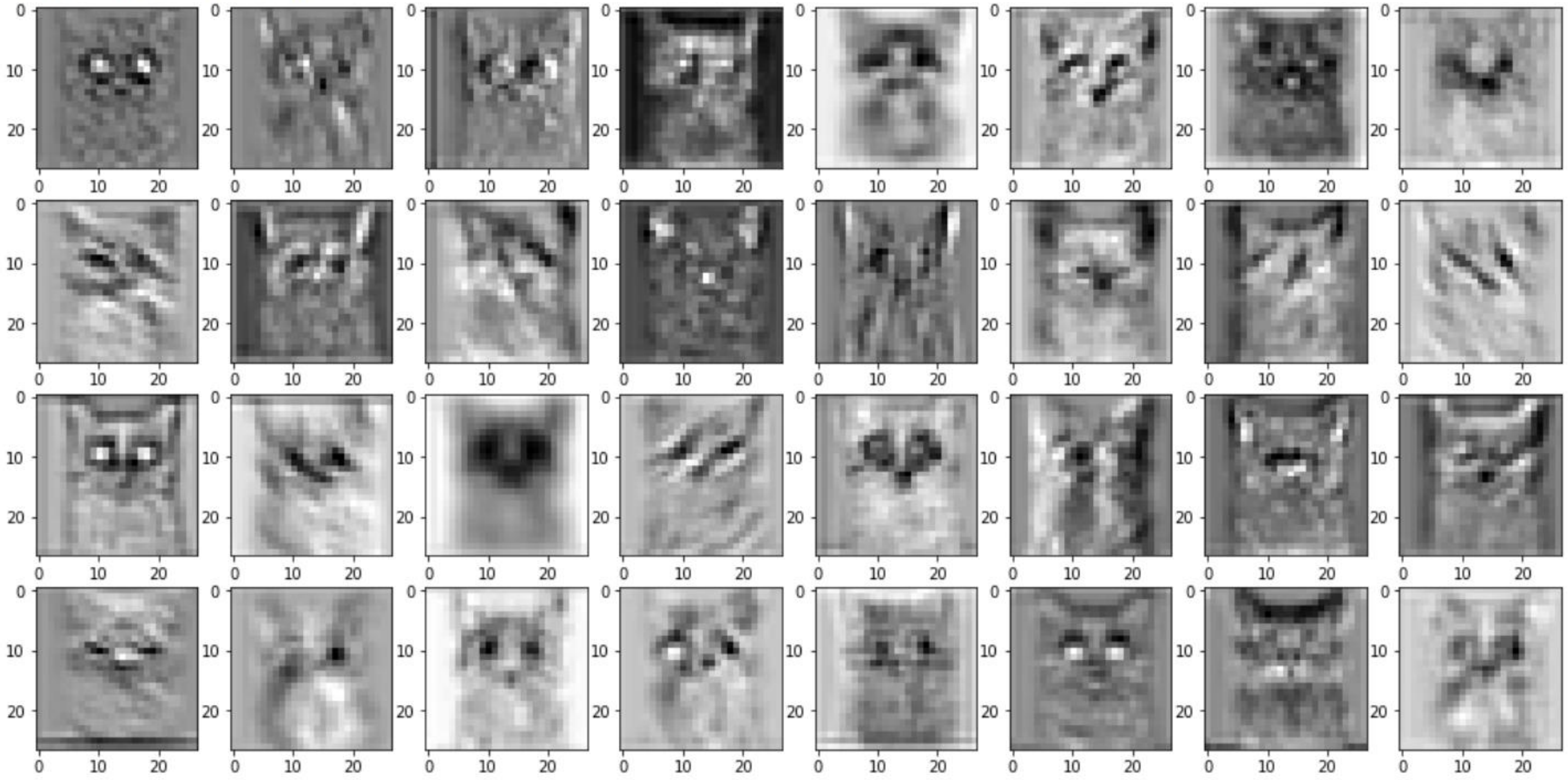
After convolution, the output feature map has 192 channels

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

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

Repeat – 2nd convolution

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



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])
```

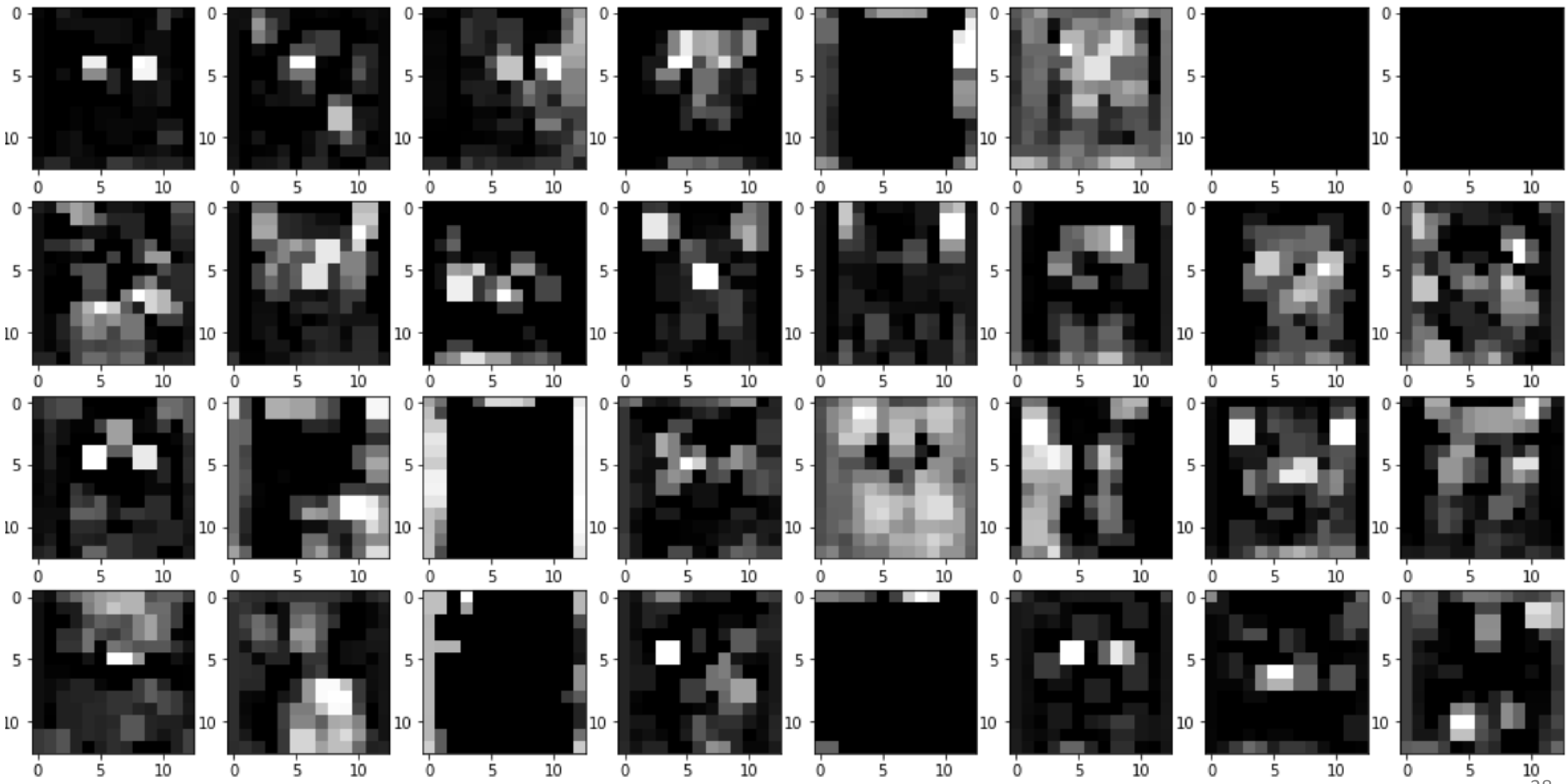
$$\frac{27 + 2 \times 0 - 3}{2} + 1 = 13$$

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

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), 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
  )
```

PYTORCH

Repeat – max pooling



Repeat – 3rd convolution

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

[17]:
```python
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

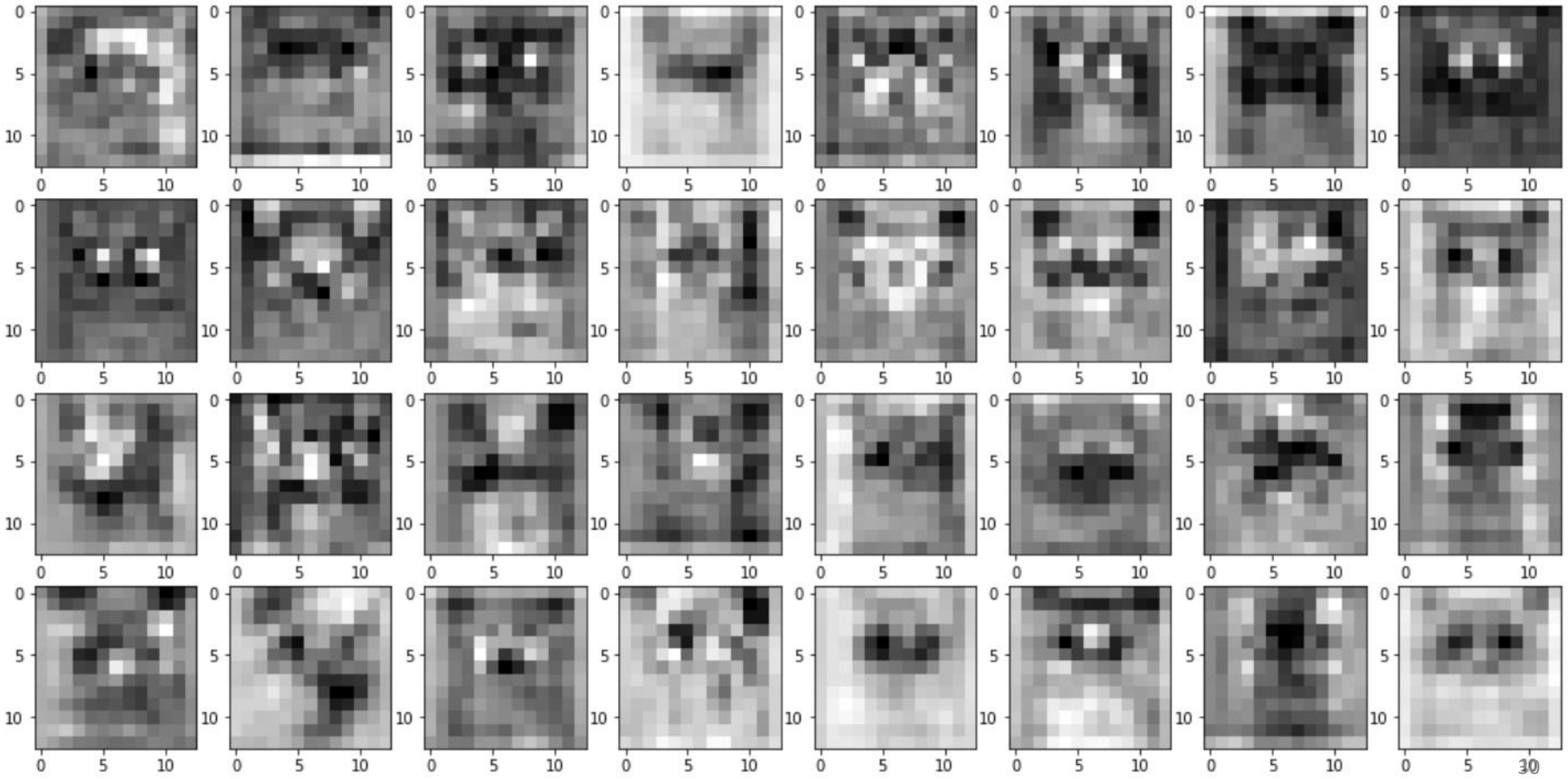
$$\frac{13 + 2 \times 1 - 3}{1} + 1 = 13$$

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

```
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

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



Repeat – max pooling

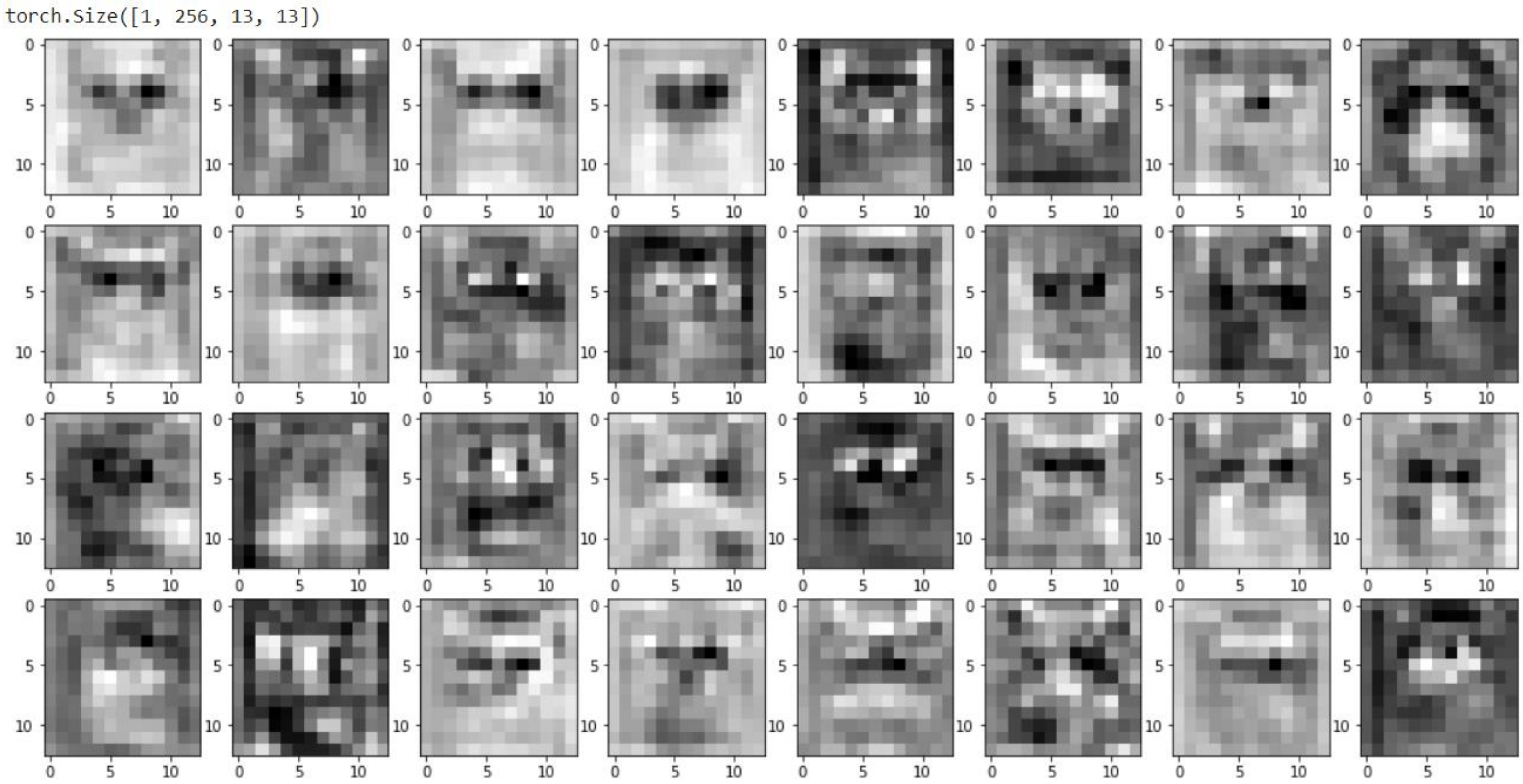
features[7, 8]

[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), 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 – 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])
```

```
(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)
  (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)
)
```

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

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

Practice – What does CNN learn?

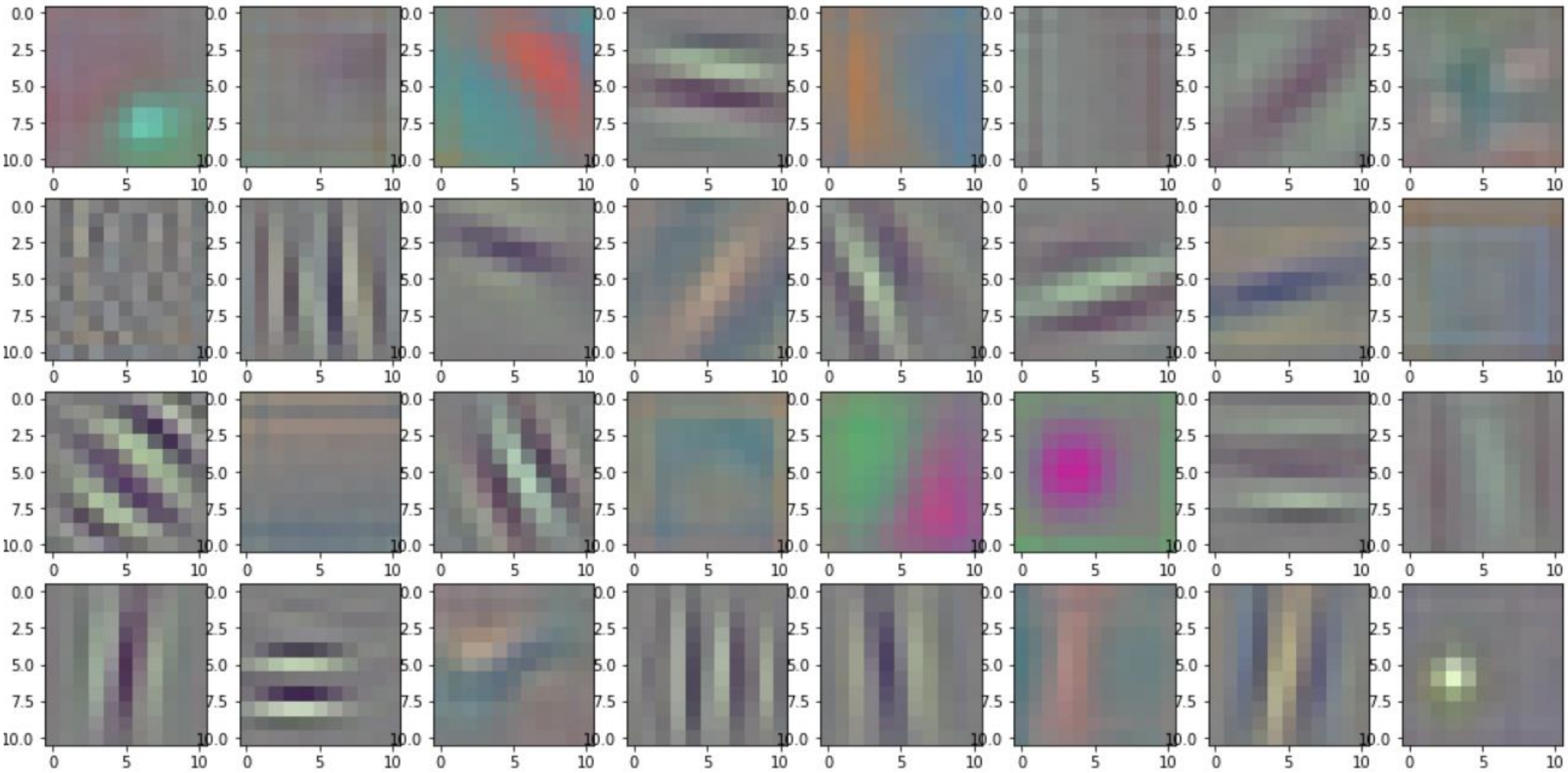
- Run “7.2. What does CNN learn.ipynb"

What does CNN learn?

[4]:
```python
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)
```

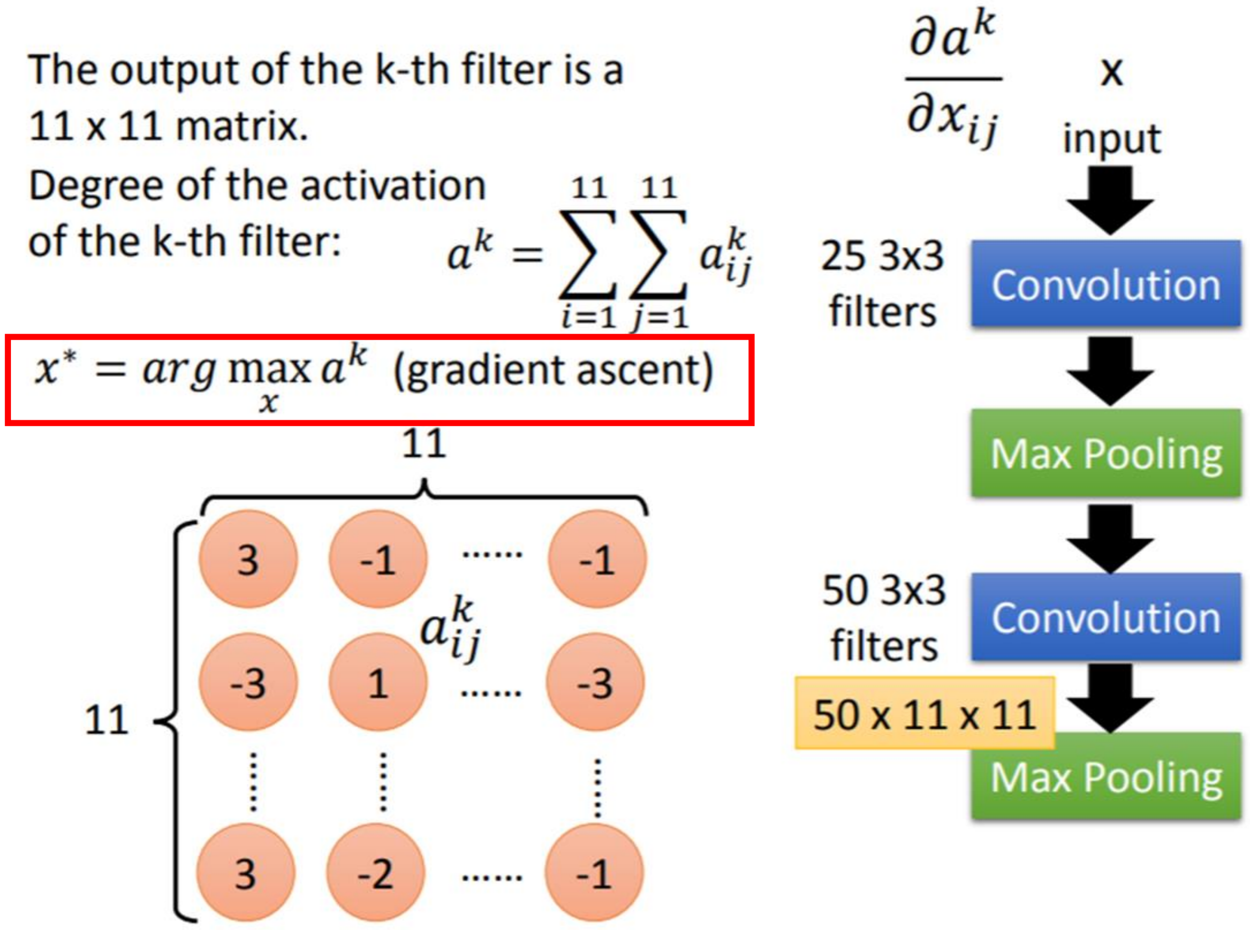
What does CNN learn?



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?



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

What does CNN learn?

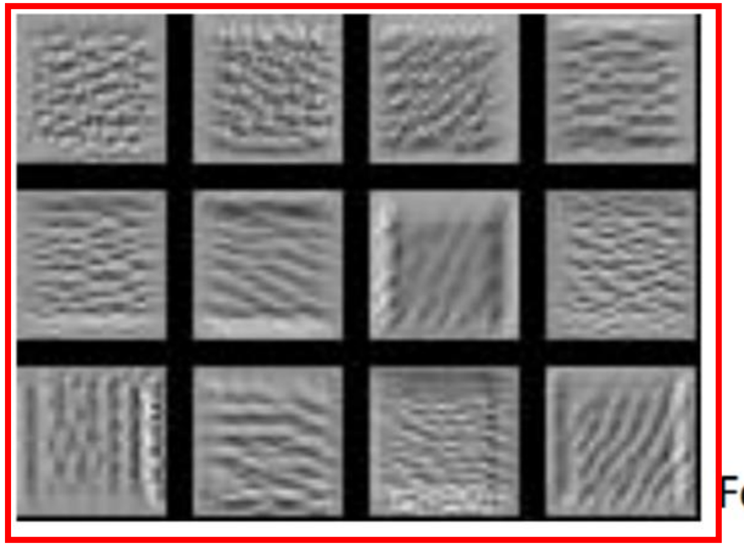
With MNIST data set, in the convolution layer, the filters detects a particular texture pattern.

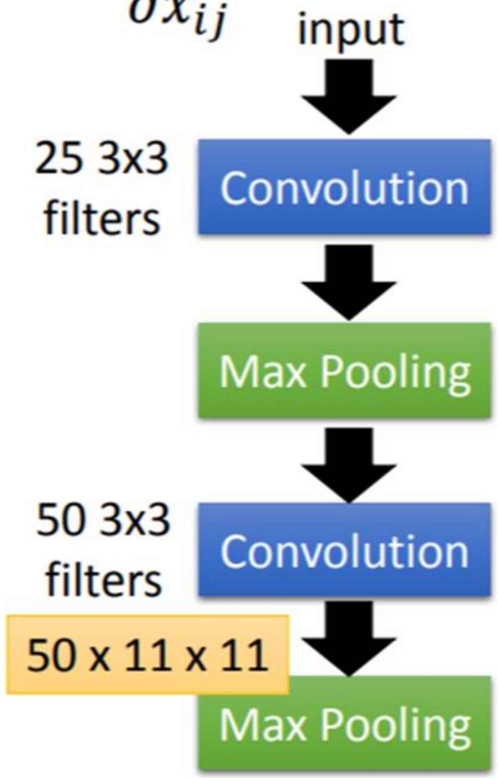
The output of the k-th filter is a 11 x 11 matrix.

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

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

$$\frac{\partial a^k}{\partial x_{ij}}$$





For each filter

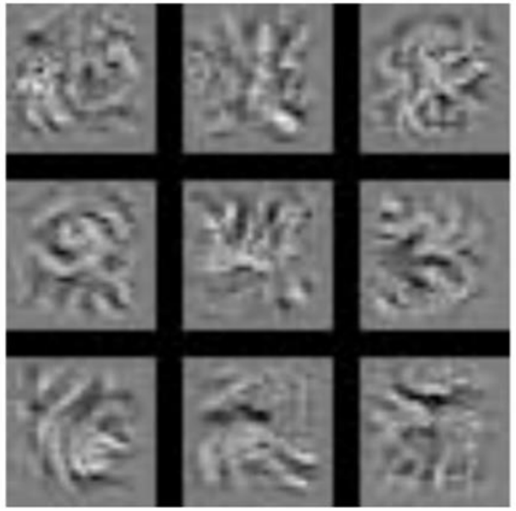
How to implement this in PyTorch?

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

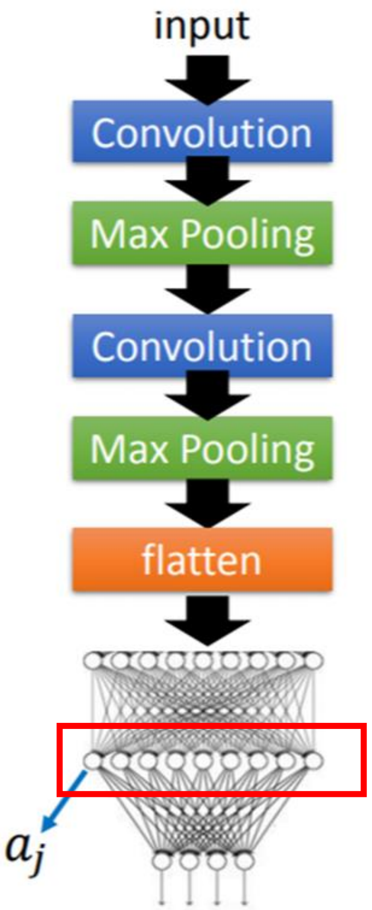
What does CNN learn?

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



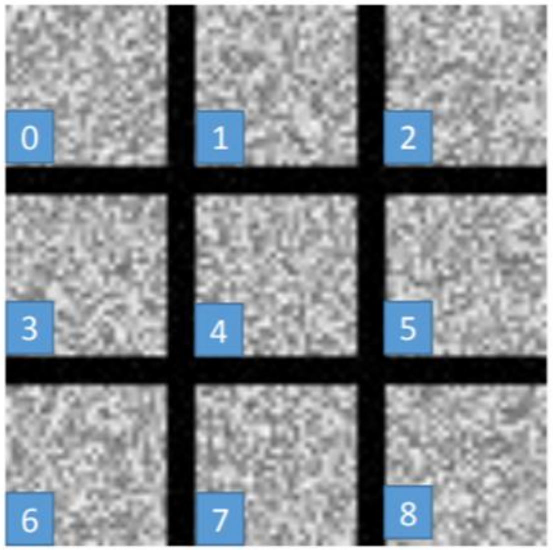
Reference: 李弘毅 ML Lecture 10 https://youtu.be/FrKWiRv254g

What does CNN learn?

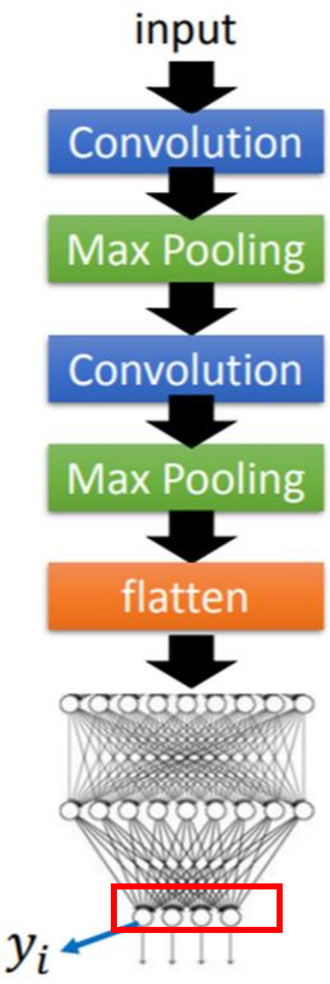
If we watch the output layer node, it is easy to see that CNN is easily fooled.

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

Can we see digits?



Deep Neural Networks are Easily Fooled
https://www.youtube.com/watch?v=M2IebCN9Ht4



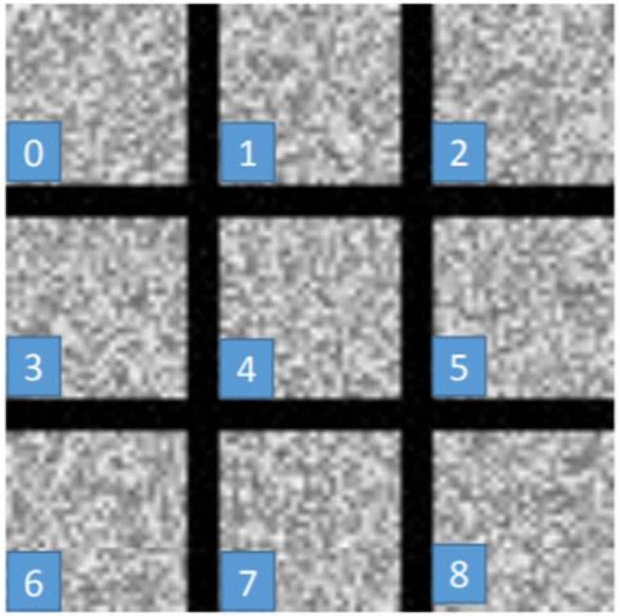
Reference: 李弘毅 ML Lecture 10 https://youtu.be/FrKWiRv254g

What does CNN learn?

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

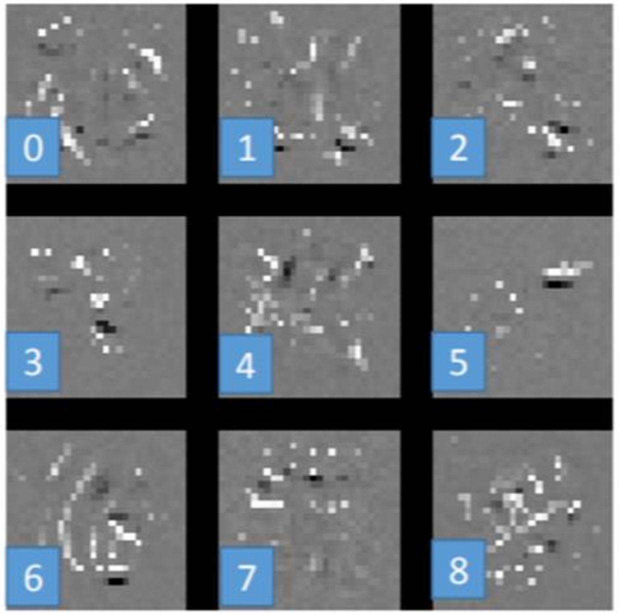
Here white pixels indicate ink, and black pixels indicate “NO INK”.

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



Over all pixel values

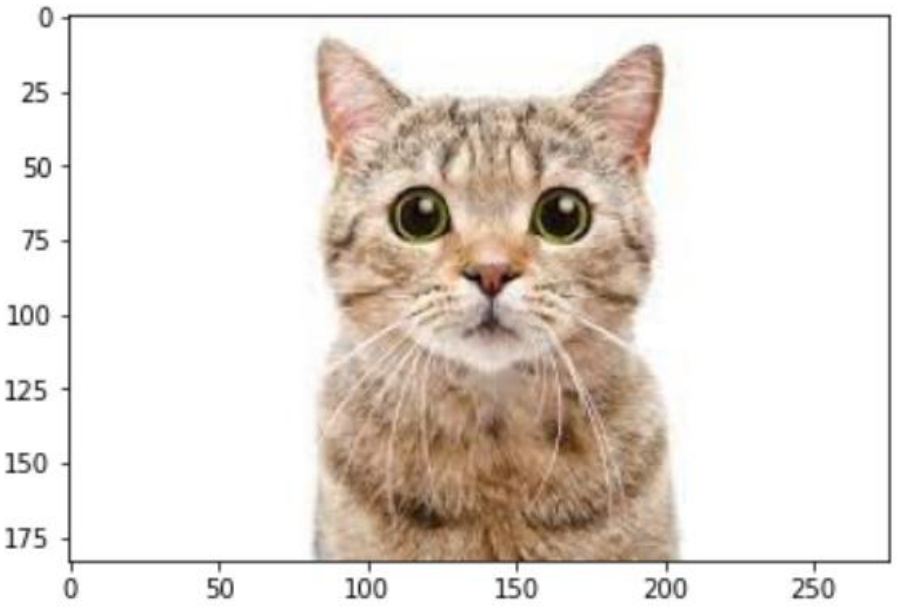
$$x^* = arg\max_x \left(y^i - \boxed{\sum_{i,j} |x_{ij}|} \right)$$

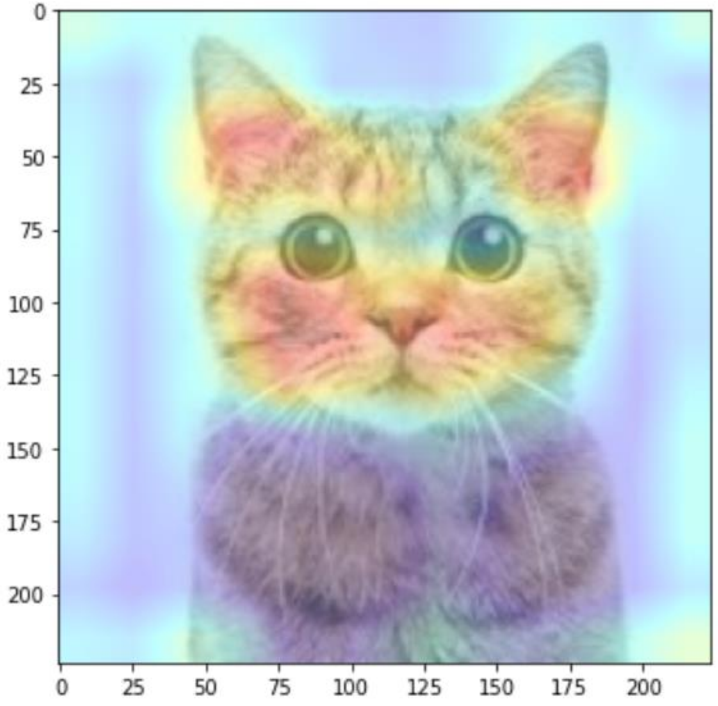


L1 regularization to force xij=0, i.e., force most pixels to be black, NO INK (as only small part of the image has ink)

Practice – What does CNN learn?

- Run “7.3 GradCAM.ipynb"





HW4

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