Practice – CNN

- Run "7.1. CNN (AlexNet).ipynb"

Load pre-trained image classification models

```
[2]   import torchvision
      model = torchvision.models.alexnet(pretrained=True)

      Downloading: "https://download.pytorch.org/models
      100% ████████████████████████  233M/233M
```

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/

Computer vision tasks

6, 7: CNN
8, 9, 10: Alex Net
HW: VGG16
11, 12, HW5: Res Net

15: U Net

13, 14: Faster RCNN

Mask RCNN

Classification

Semantic Segmentation

GRASS, CAT, TREE, SKY

No objects, just pixels

Classification + Localization

CAT

Single Object

Object Detection

DOG, DOG, CAT

Instance Segmentation

DOG, DOG, CAT

Multiple Object

This image is CC0 public domain

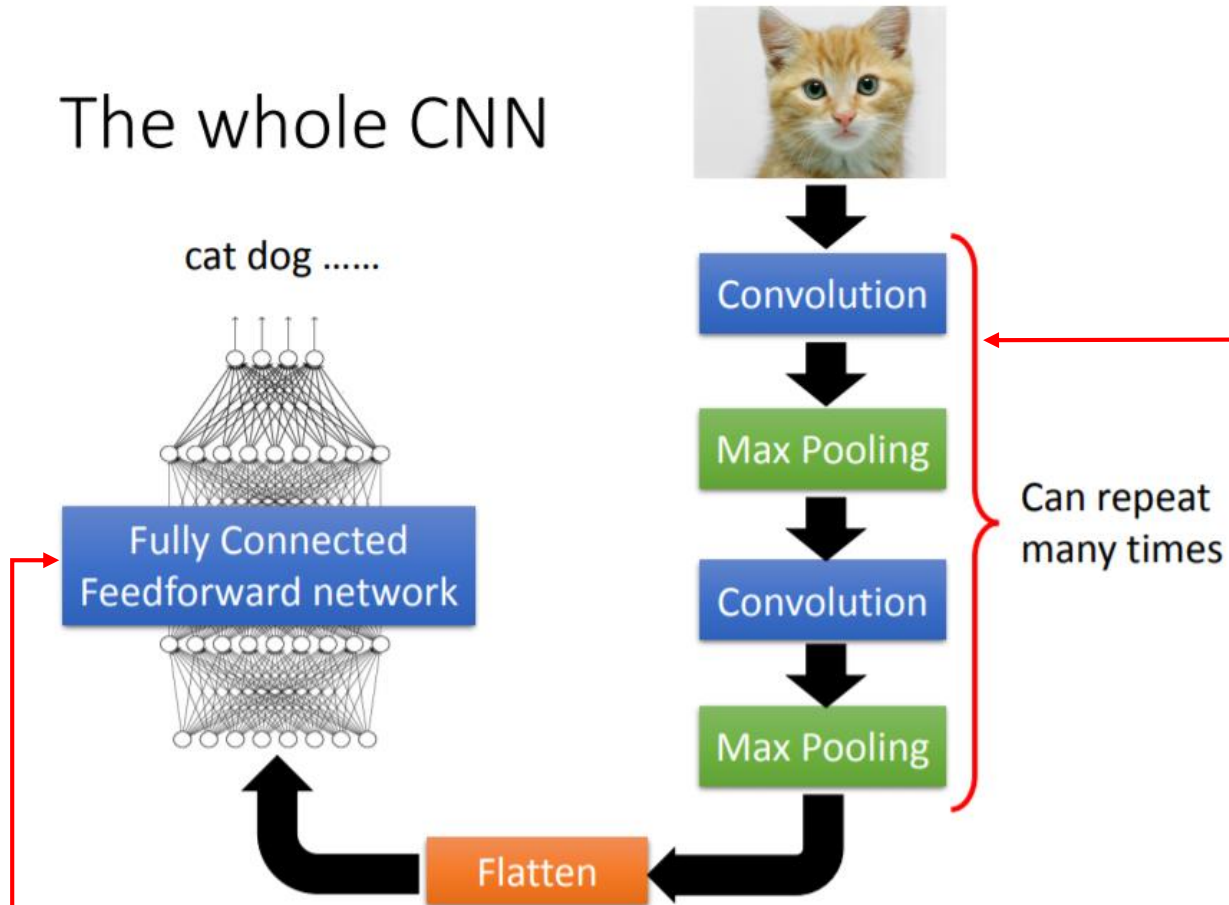圖片來源: https://kharshit.github.io/blog/2019/08/23/quick-intro-to-instance-segmentation

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

Flatten

Convolution

Max Pooling

Convolution

Max Pooling

Can repeat many times

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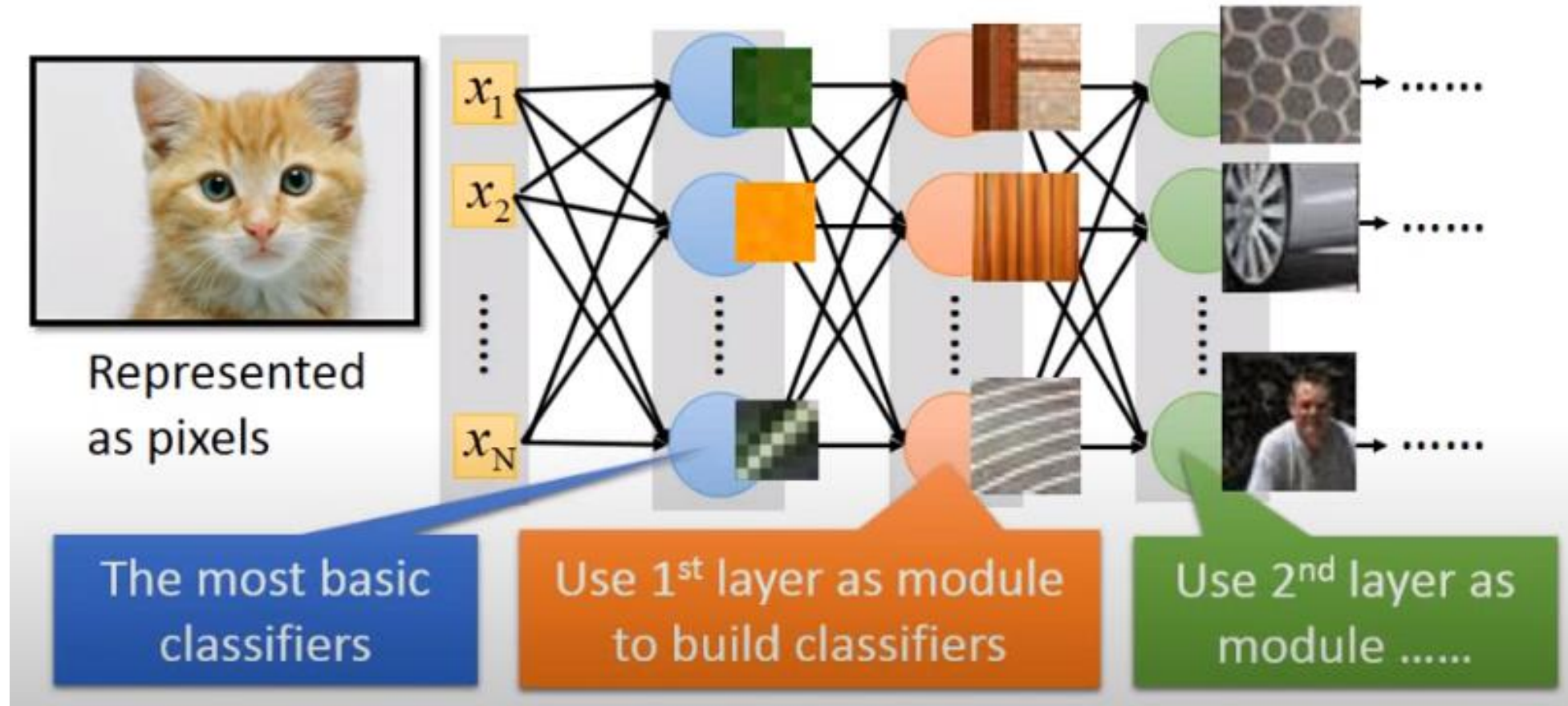https://youtu.be/FrKWiRv254g

```
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)
  )
)
```

Why not using MLP to classify images?

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

Represented as pixels

x_1
x_2
x_N

The most basic classifiers

Use 1st layer as module to build classifiers
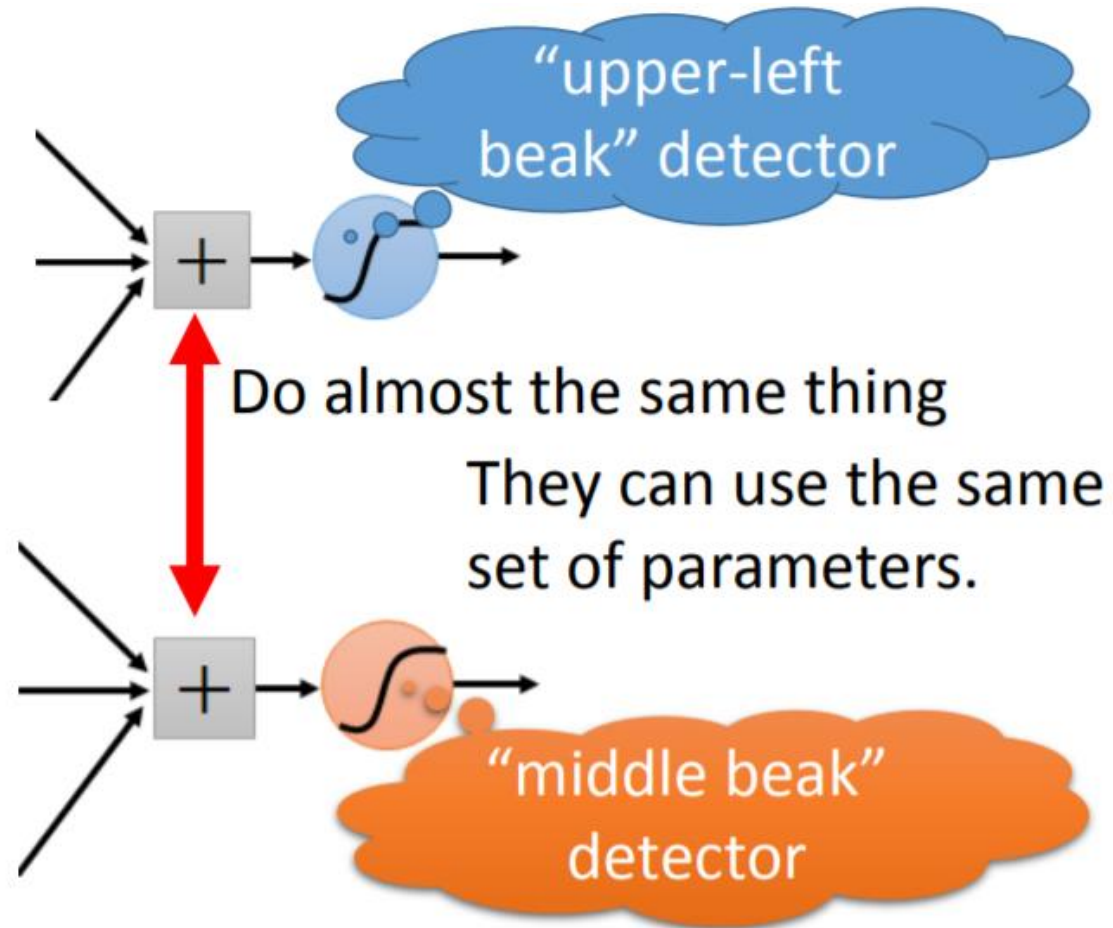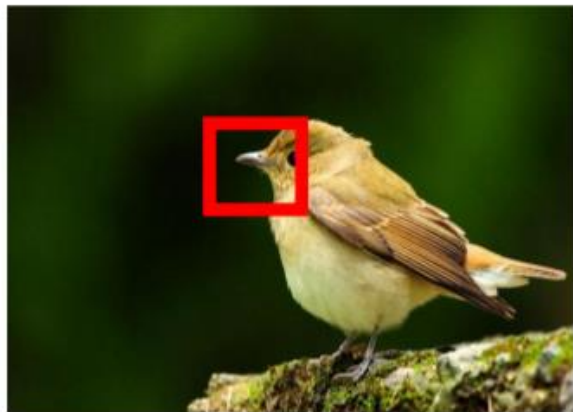
Use 2nd layer as module ......

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
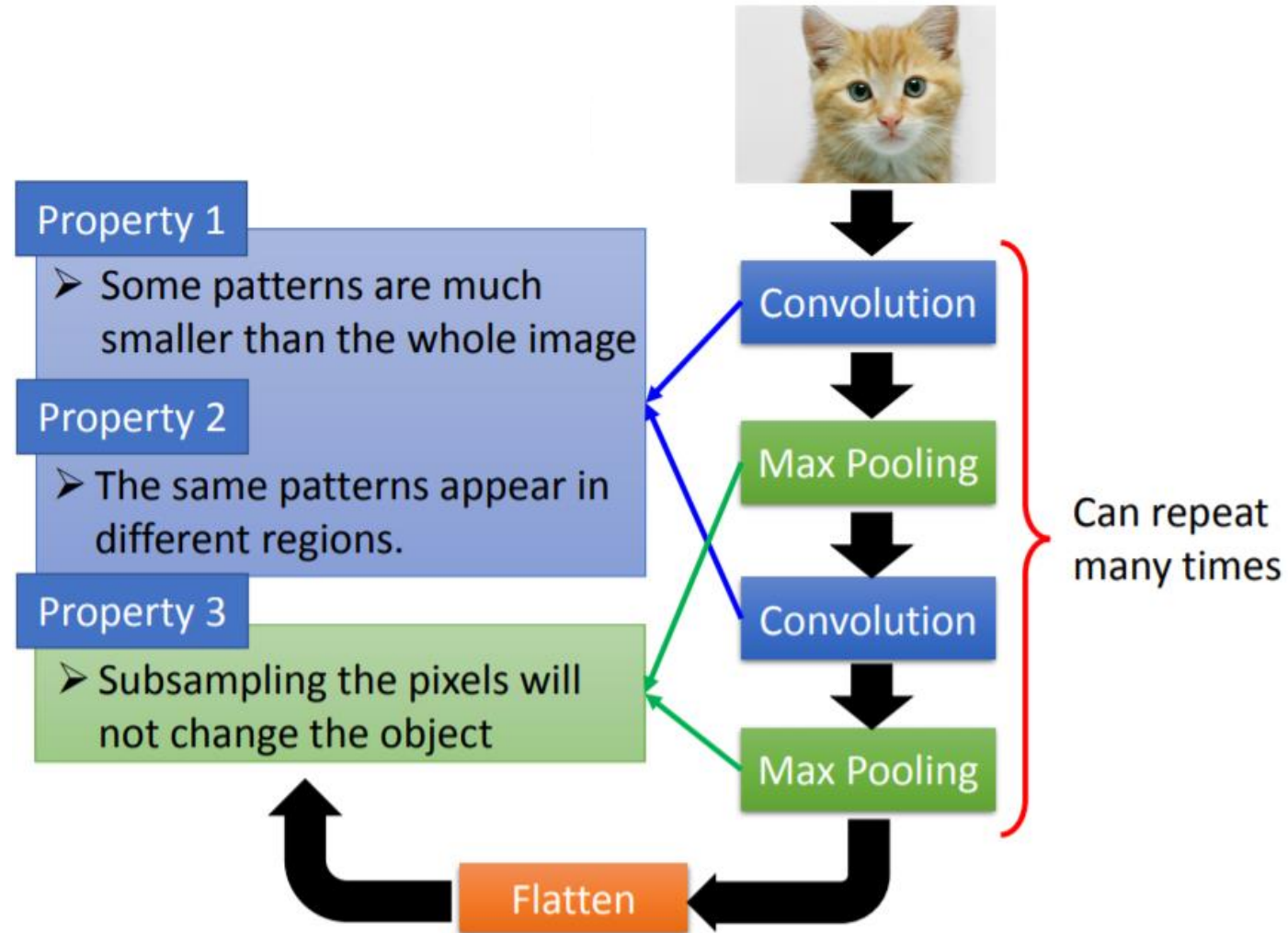
"beak" detector

The same patterns appear in different regions

"upper-left beak" detector

Do almost the same thing
They can use the same set of parameters.

"middle beak" detector

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

Use convolution and pooling operations to extract important features from input image

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

Property 2
- The same patterns appear in different regions.

Property 3
- Subsampling the pixels will not change the object

Convolution → Max Pooling → Convolution → Max Pooling

Can repeat many times

Flatten

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

Use CV to read image file

```python
import cv2
import matplotlib.pyplot as plt
image = cv2.imread(fname)
image = cv2.cvtColor(image,cv2.COLOR_BGR2RGB)
plt.imshow(image)
plt.show()
```

In [6]:

W

H
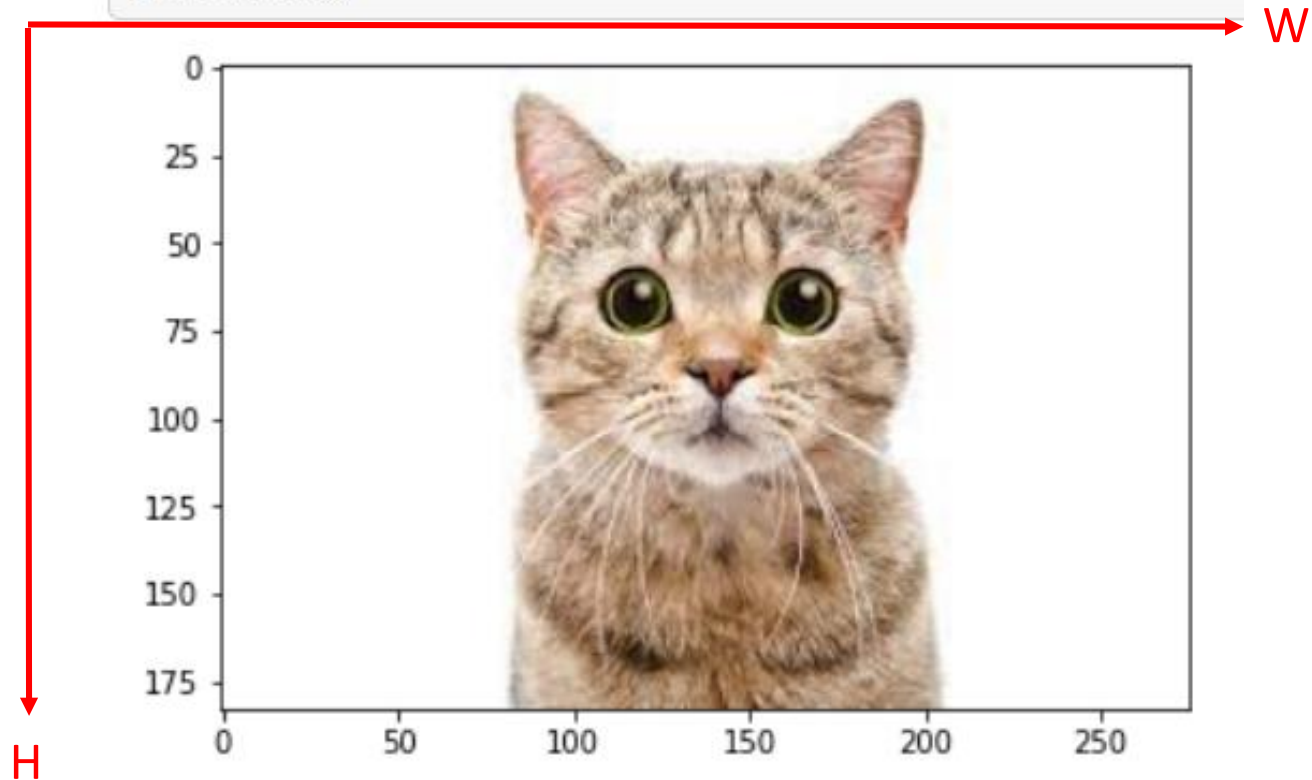
Image pre-processing

- Image width and height – resize, center crop
- Pixel values – Standardized to [0, 1], normalized to N(0, 1)

```python
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] )])
```

```python
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])
```

Prepare input format

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])
```

1st convolution

```
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=
  `
```

```
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))
```

Filter searches patterns in a small region

6 x 6 image

Those are the network parameters to be learned.

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

Filter 1 Matrix

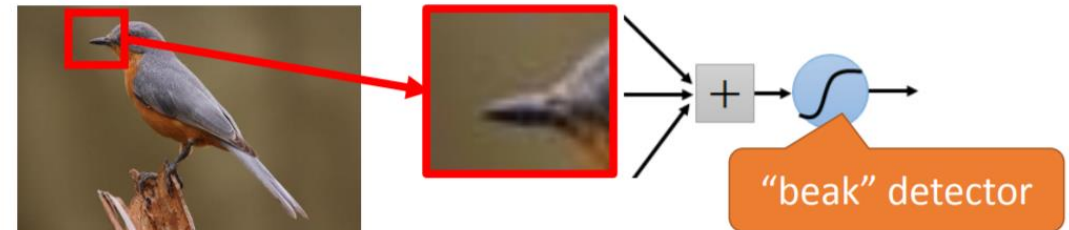| -1 | 1 | -1 |
| -1 | 1 | -1 |
| -1 | 1 | -1 |

Filter 2 Matrix

Property 1 Each filter detects a small pattern (3 x 3).

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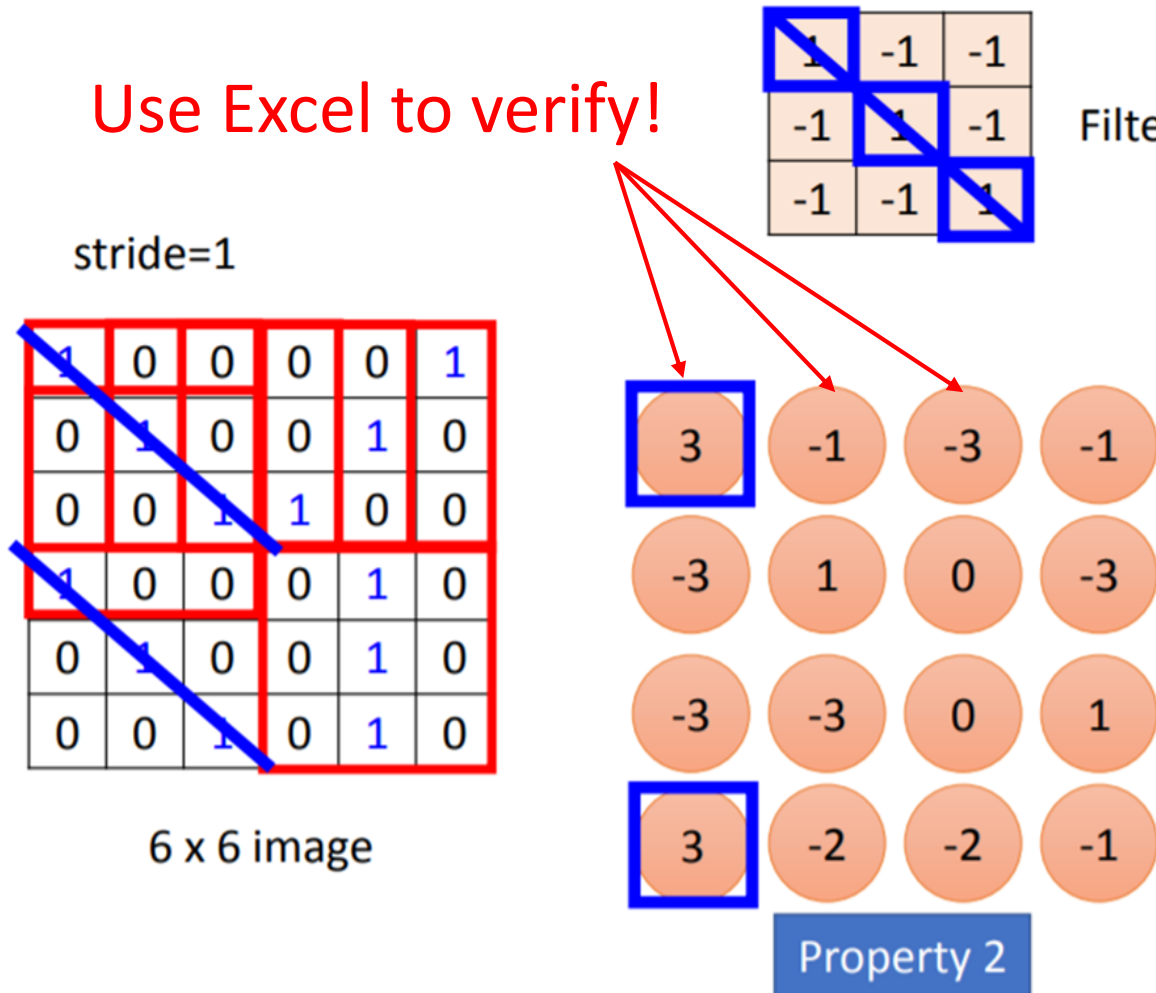
• Some patterns are much smaller than the whole image

A neuron does not have to see the whole image to discover the pattern.
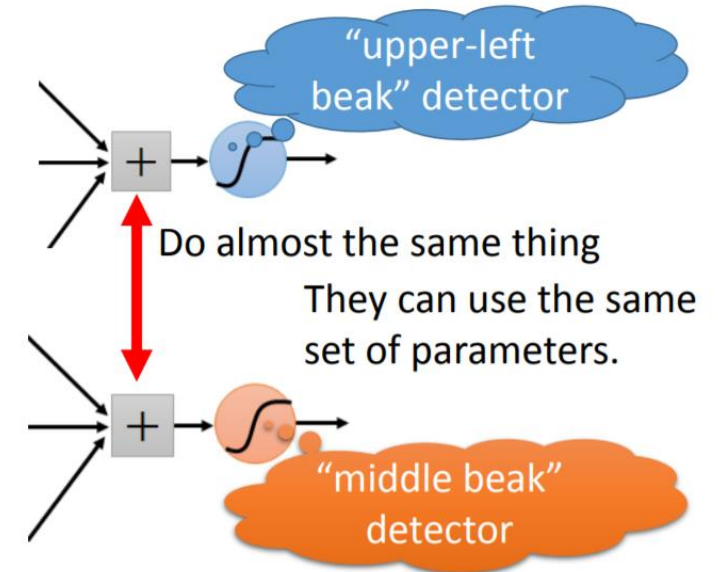
Connecting to small region with less parameters

"beak" detector

Filter searches a particular pattern in different regions

Use Excel to verify!

stride=1

6 x 6 image

Filter 1

Property 2

Property 2

Property 2

• The same patterns appear in different regions.

"upper-left beak" detector

Do almost the same thing
They can use the same set of parameters.

"middle beak" detector
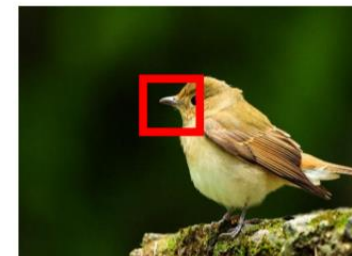
Stride determines how filter shifts

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

1	-1	-1	Filter 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)

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
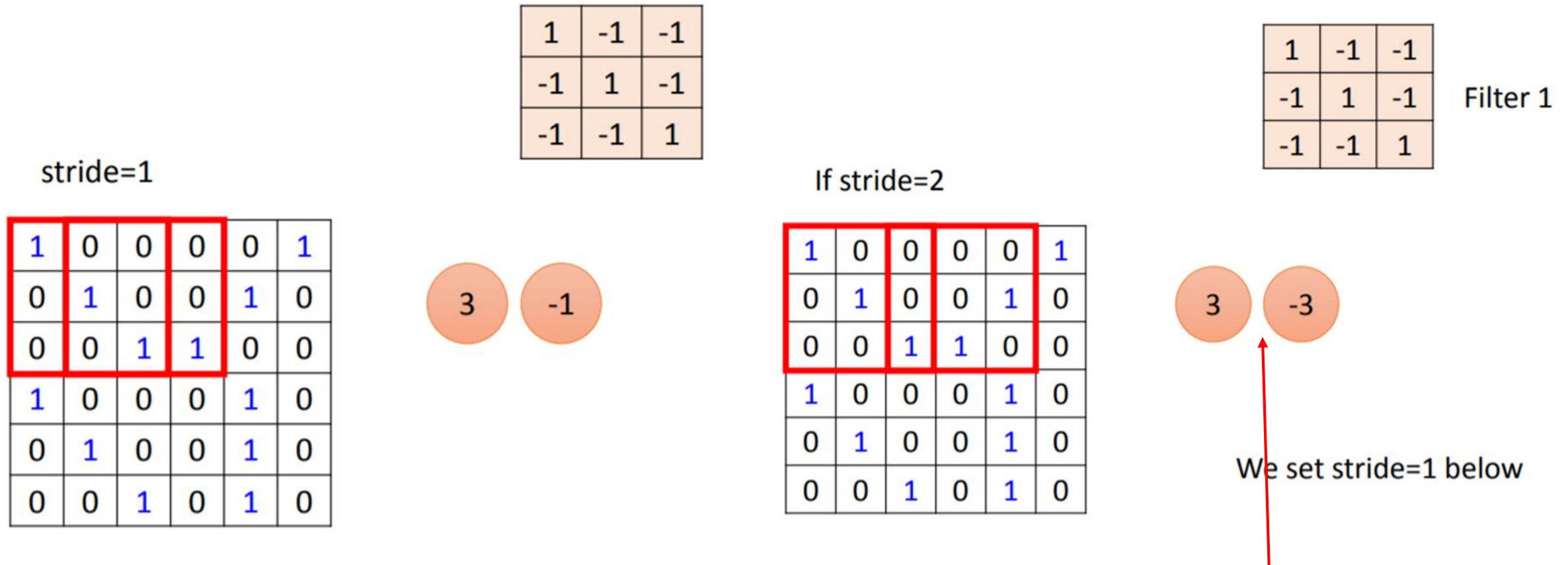
Use Excel to verify!

Filter searches a particular pattern in different regions

Use Excel to verify!
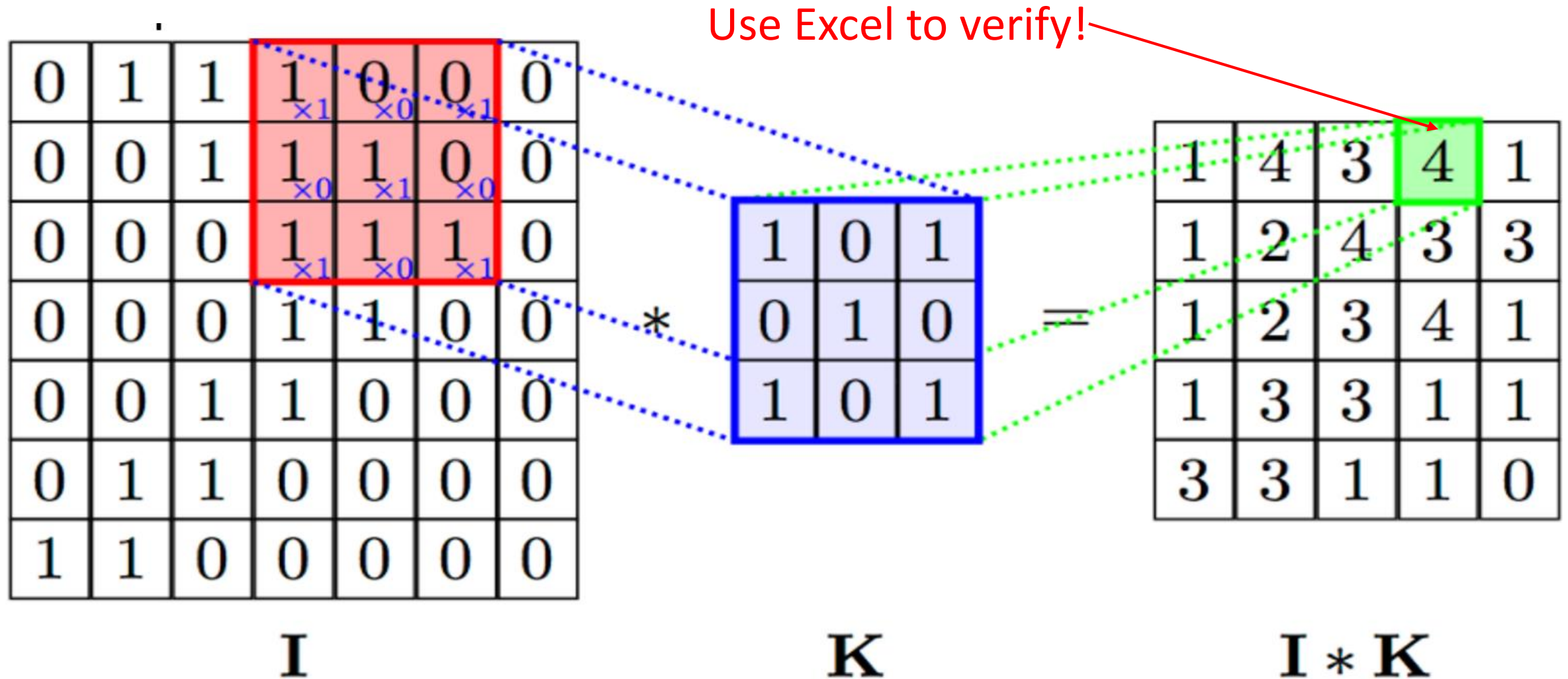
Visualization of a convolution operation. I is the input to the network, k represents the vector and i*k represents the matrix multiplication resulting of sliding the kernel through the input. Reference (Veličković, 2016)

Filter searches a particular pattern in different regions

INPUT

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

FILTER

1	0	1
0	1	0
1	0	1

CONVOLVED FEATURE

4		

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

1	0	1
0	1	0
1	0	1

4		
2		

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

1	0	1
0	1	0
1	0	1

4	3	
2		

Use Excel to verify!
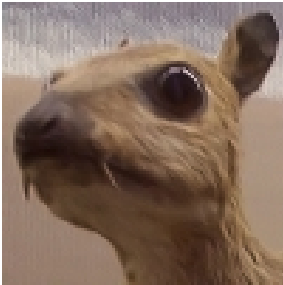
Input image   Convolution Kernel   Feature map

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

圖來源: https://developer.nvidia.com/discover/convolution

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

1	0	1
0	1	0
1	0	1

| 4	3	4
2	4	3
2	3	4

19

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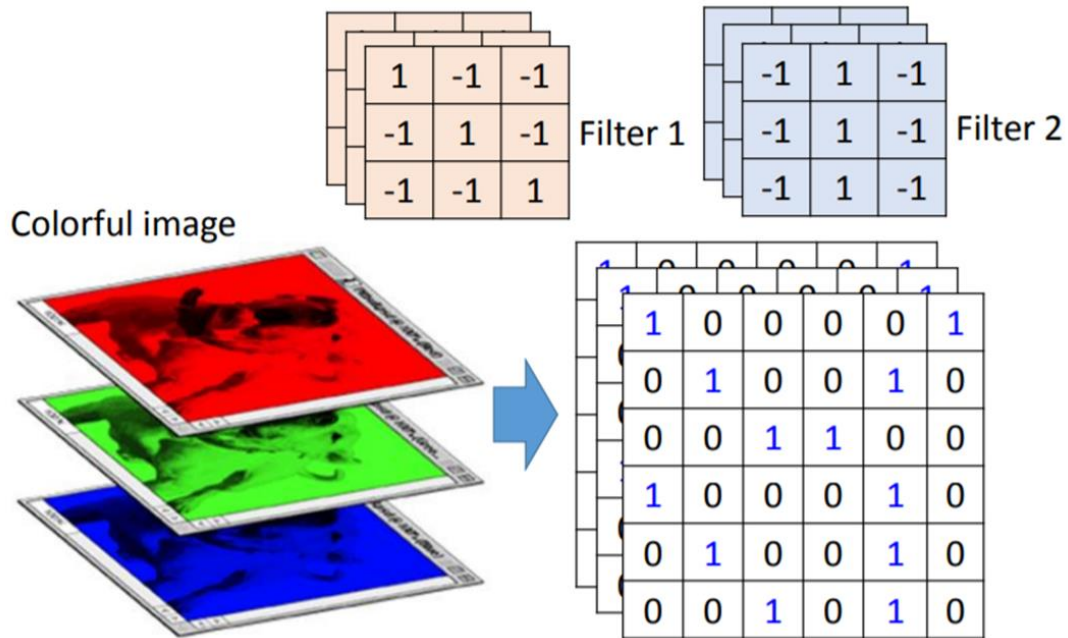

6 x 6 image

Less parameters!

Filter 1

Only connect to 9 input, not fully connected
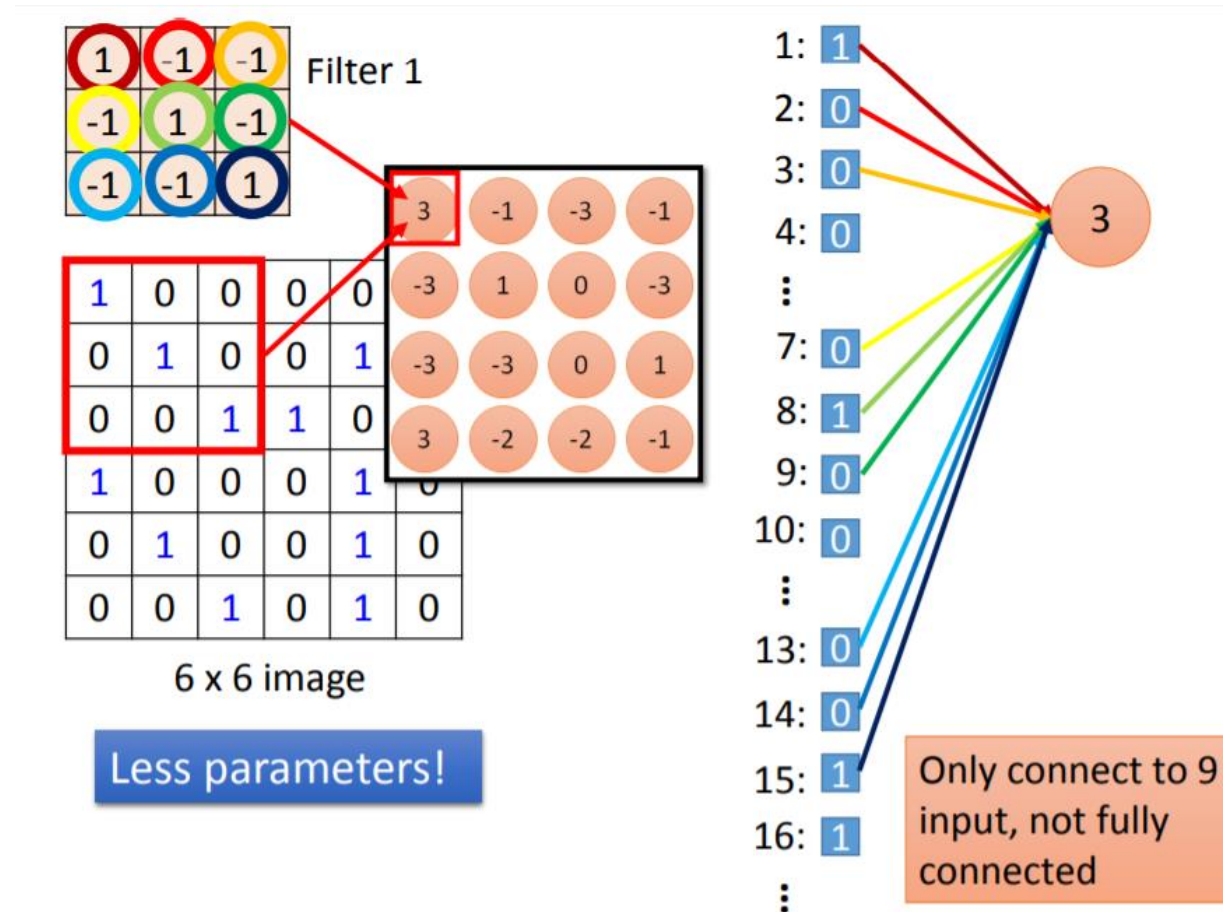
Partially connected NN with shared weights and hence with even less parameters.

Filter 1

6 x 6 image

Less parameters!

Even less parameters!

Shared weights
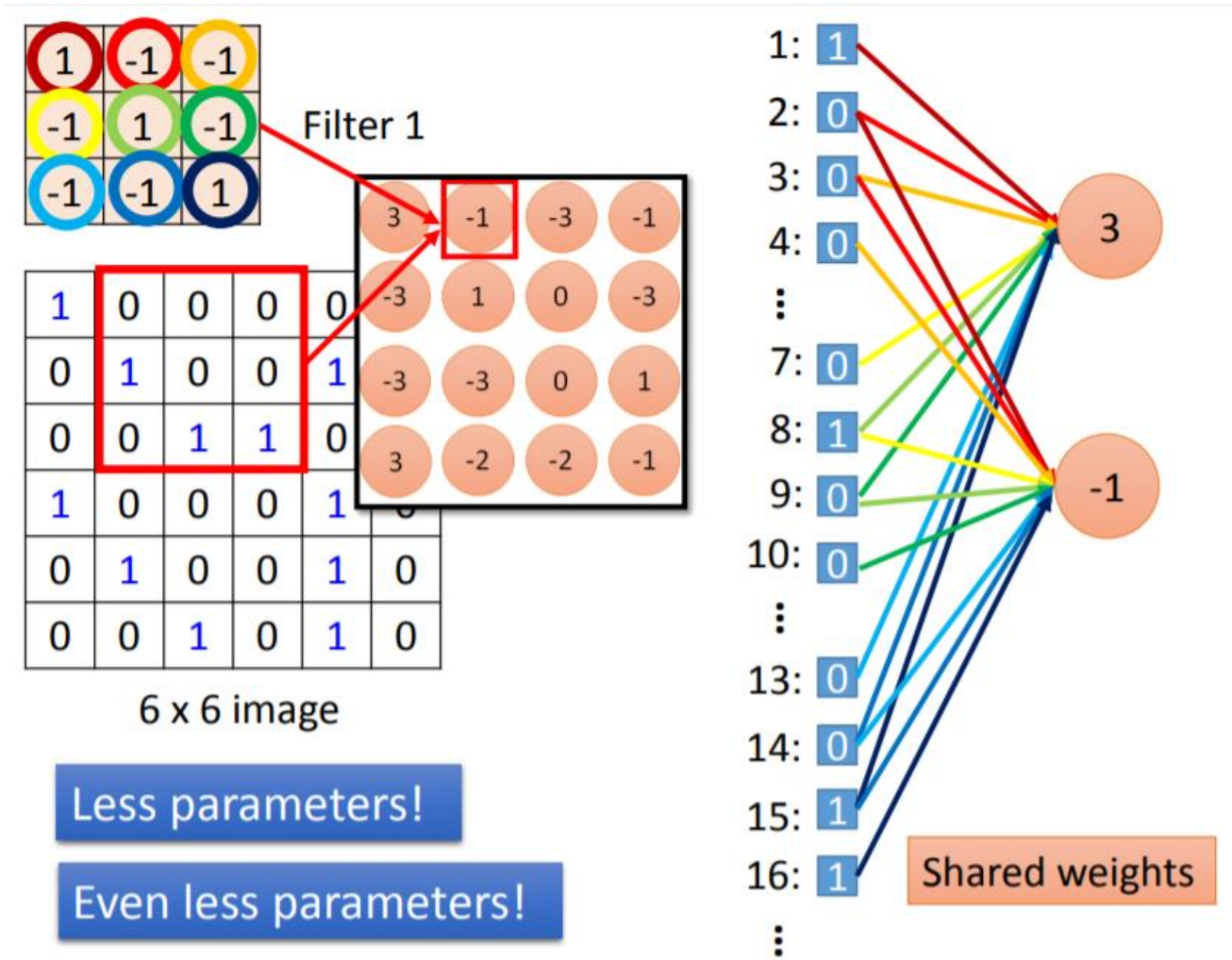
Filter has depth

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)
```

Take a look at the learned filter weights of 1st convolution

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

      (64,  3,  11,  11)
```
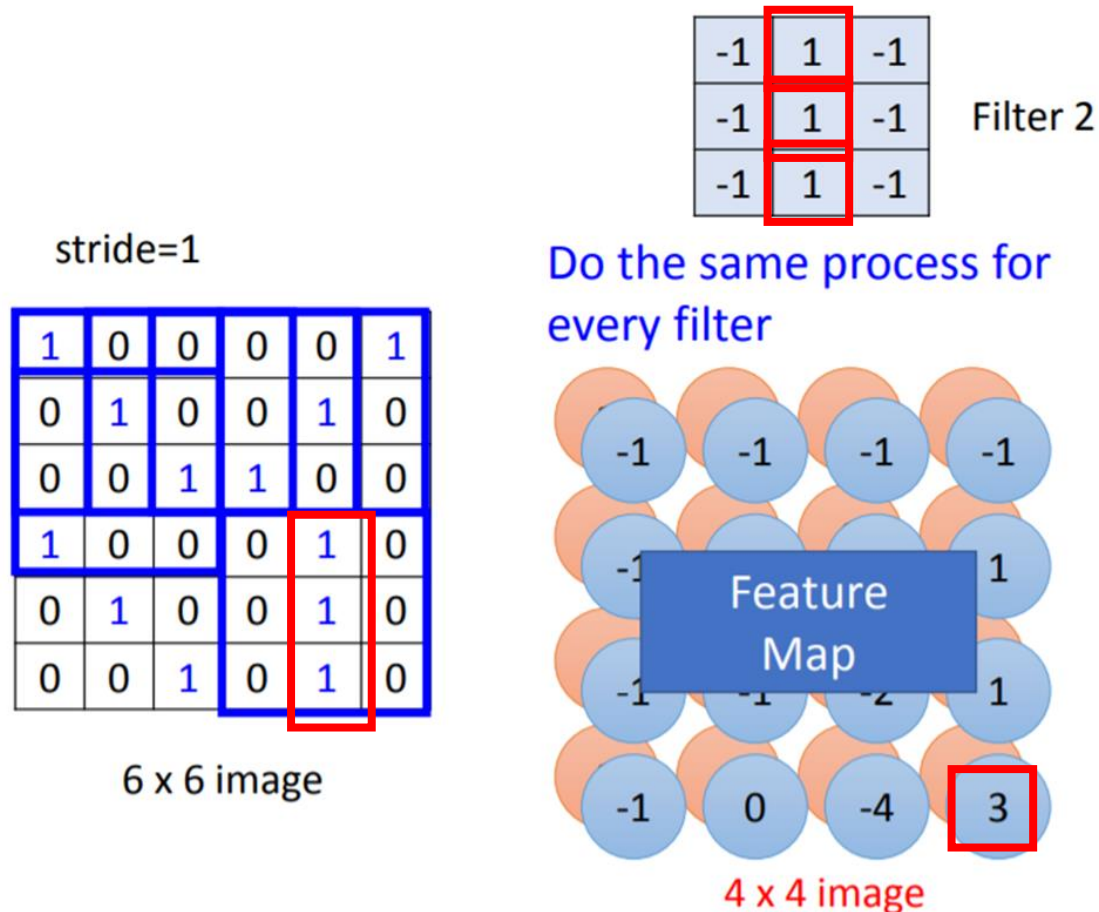
Visualize filters in the 1st convolution layer

```python
[13] # Visualize the first 32 of the filter weights
     import numpy as np
     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()
```

The learned filter's weights
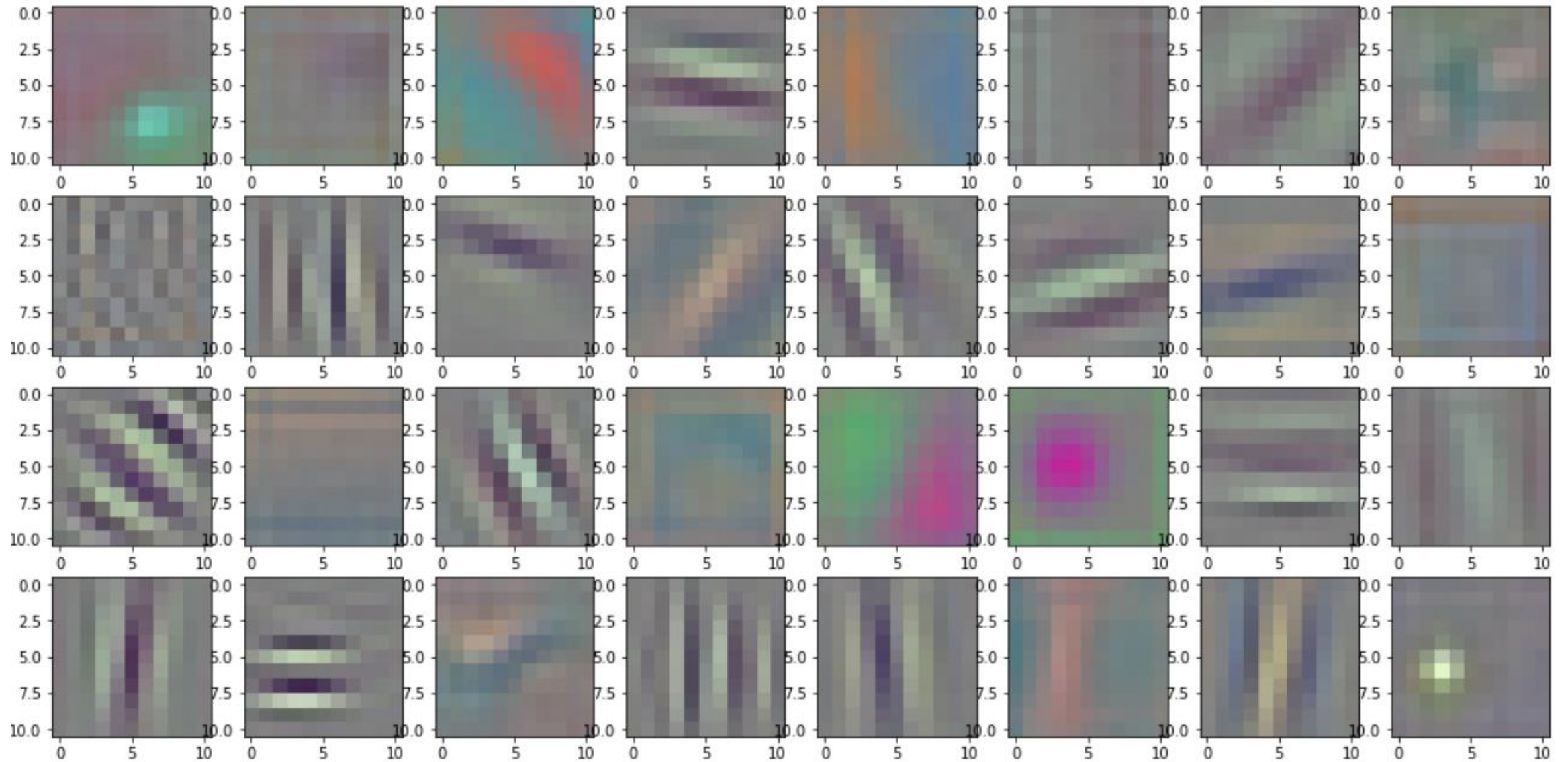
Feature maps

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


```
-1   1  -1
-1   1  -1     Filter 2
-1   1  -1
```

Do the same process for every filter

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
```

6 x 6 image

Feature Map

```
-1  -1  -1  -1
-1          1
-1          1
-1   0  -4   3
```

4 x 4 image

```
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])
```
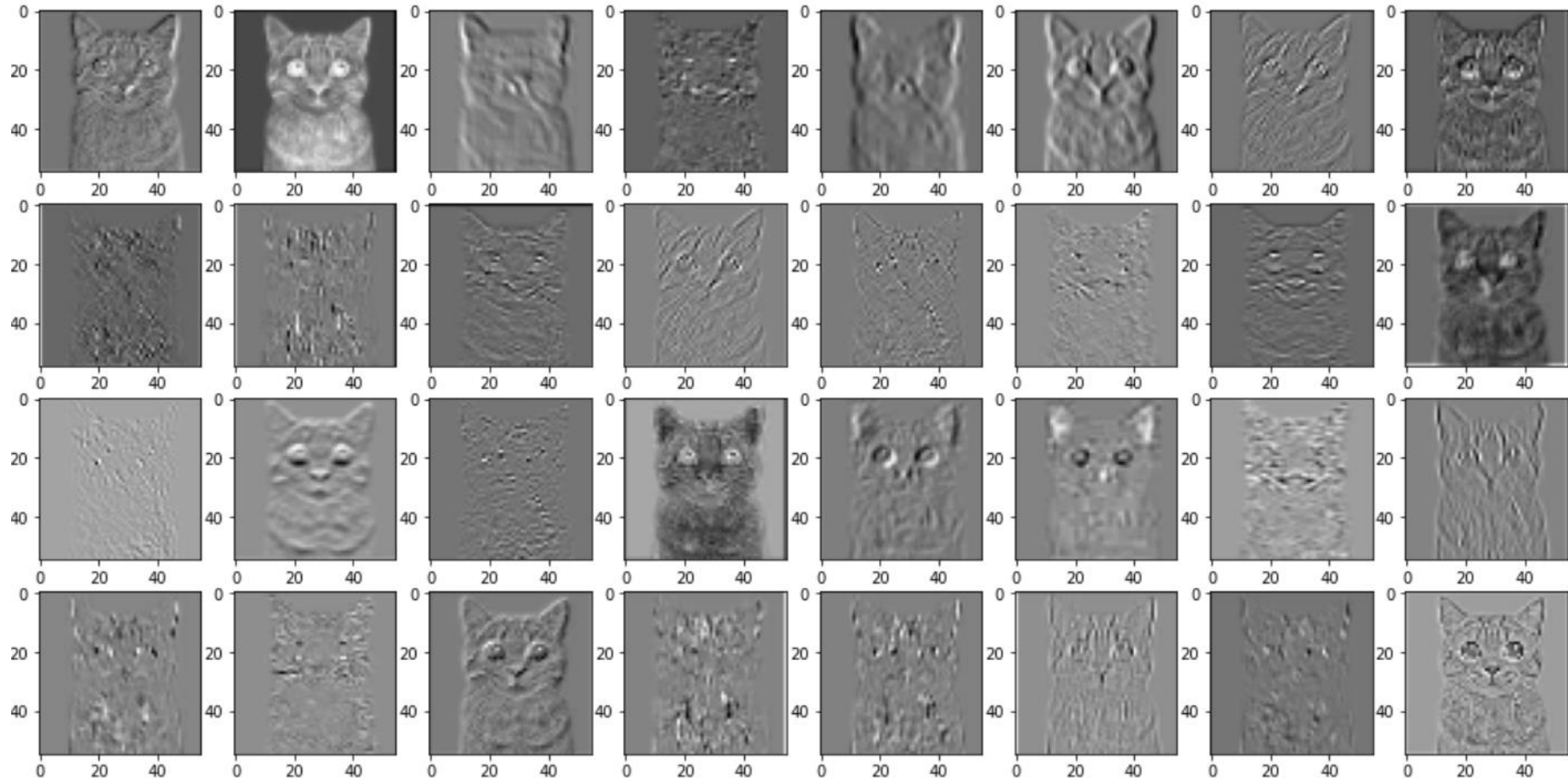
Feature map's width and height

```
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])

$$\frac{224 + 2 \times 2 - 11}{4} + 1 = 55.25$$

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

First 32 channels of the output feature map, shape = 55x55x64

Max pooling

Max pooling

max pooling

20	30
112	37

average pooling

13	8
79	20

Input:

12	20	30	0
8	12	2	0
34	70	37	4
112	100	25	12

(Singhal, 2017)

圖來源: https://medium.com/@bdhuma/which-pooling-method-is-better-maxpooling-vs-minpooling-vs-average-pooling-95fb03f45a9

Apply max pooling to the feature map from 1ˢᵗ convolution

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 channe
         fig.add_subplot(4, 8, i+1)
         plt.imshow(imgArray[i], cmap='gray')
       plt.show()
```
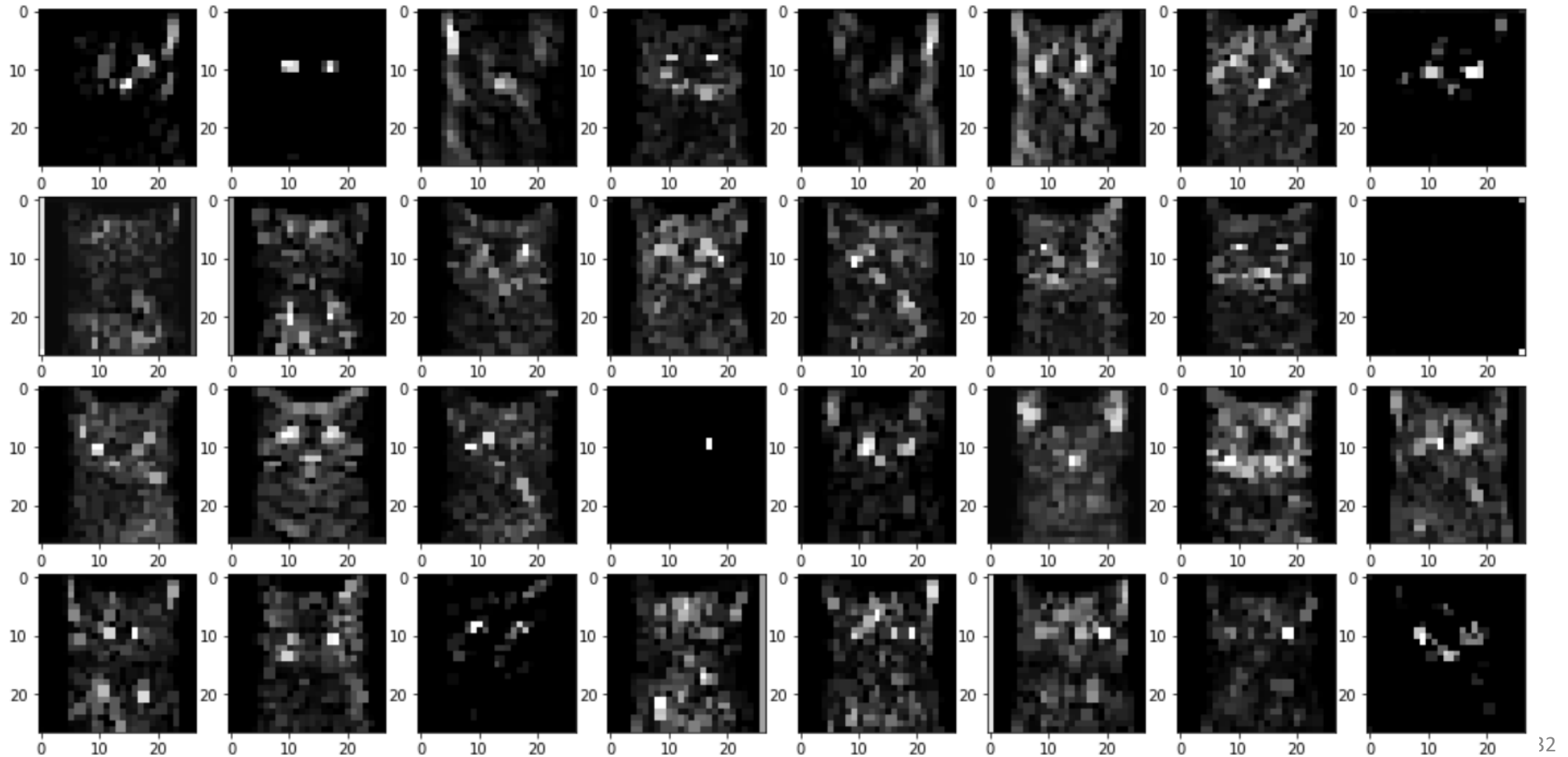
torch.Size([1, 64, 27, 27])

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

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

```
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
    )
```

PYTÖRCH

First 32 channels of the output feature map from max pooling, shape = 27x27x64

2nd convolution

```
[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])
```

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
    )
```

PYTORCH

Feature map after 2nd convolution


```
torch.Size([1, 192, 27, 27])
```
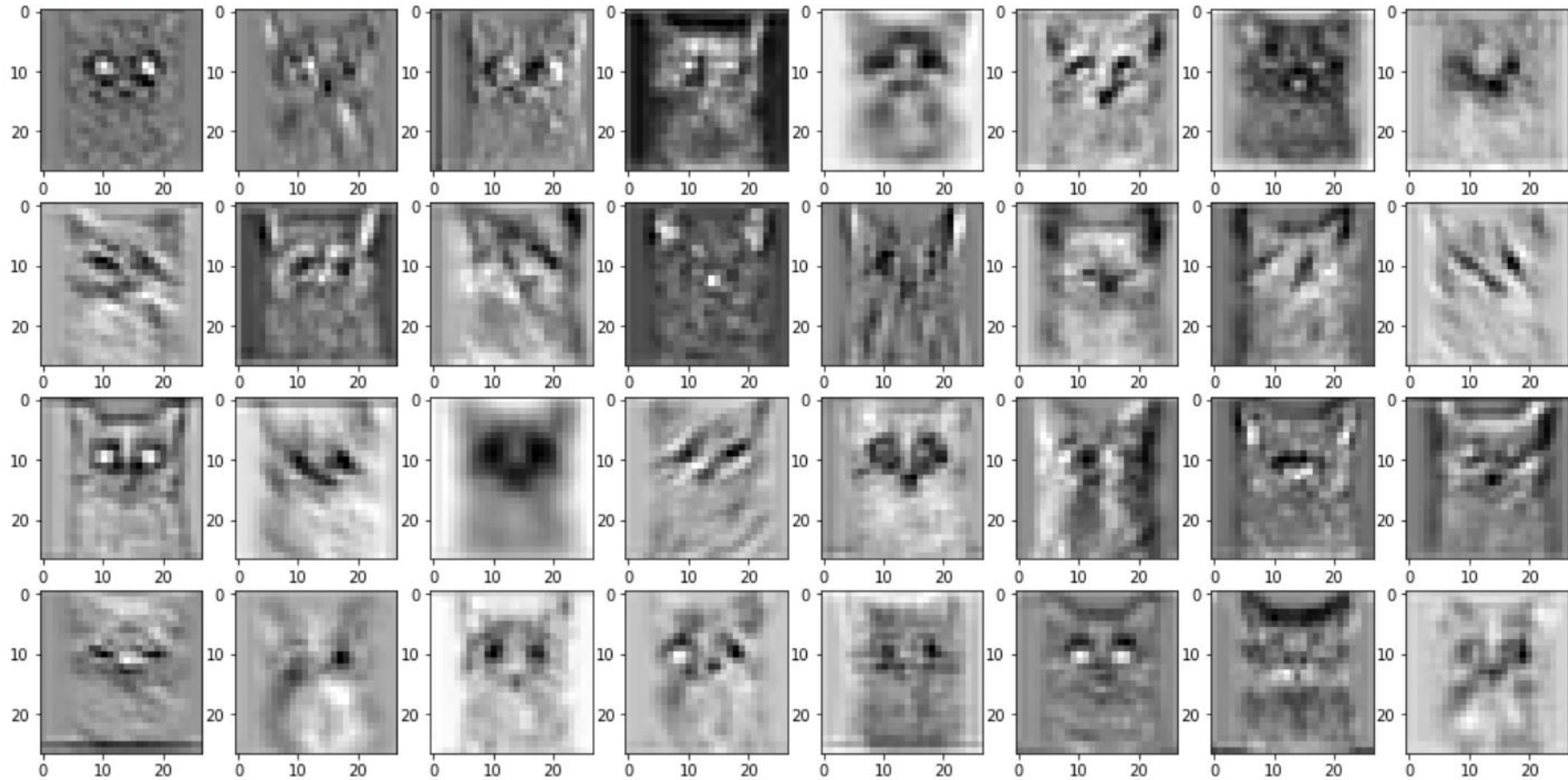
Apply max pooling to the feature map from 2nd convolution

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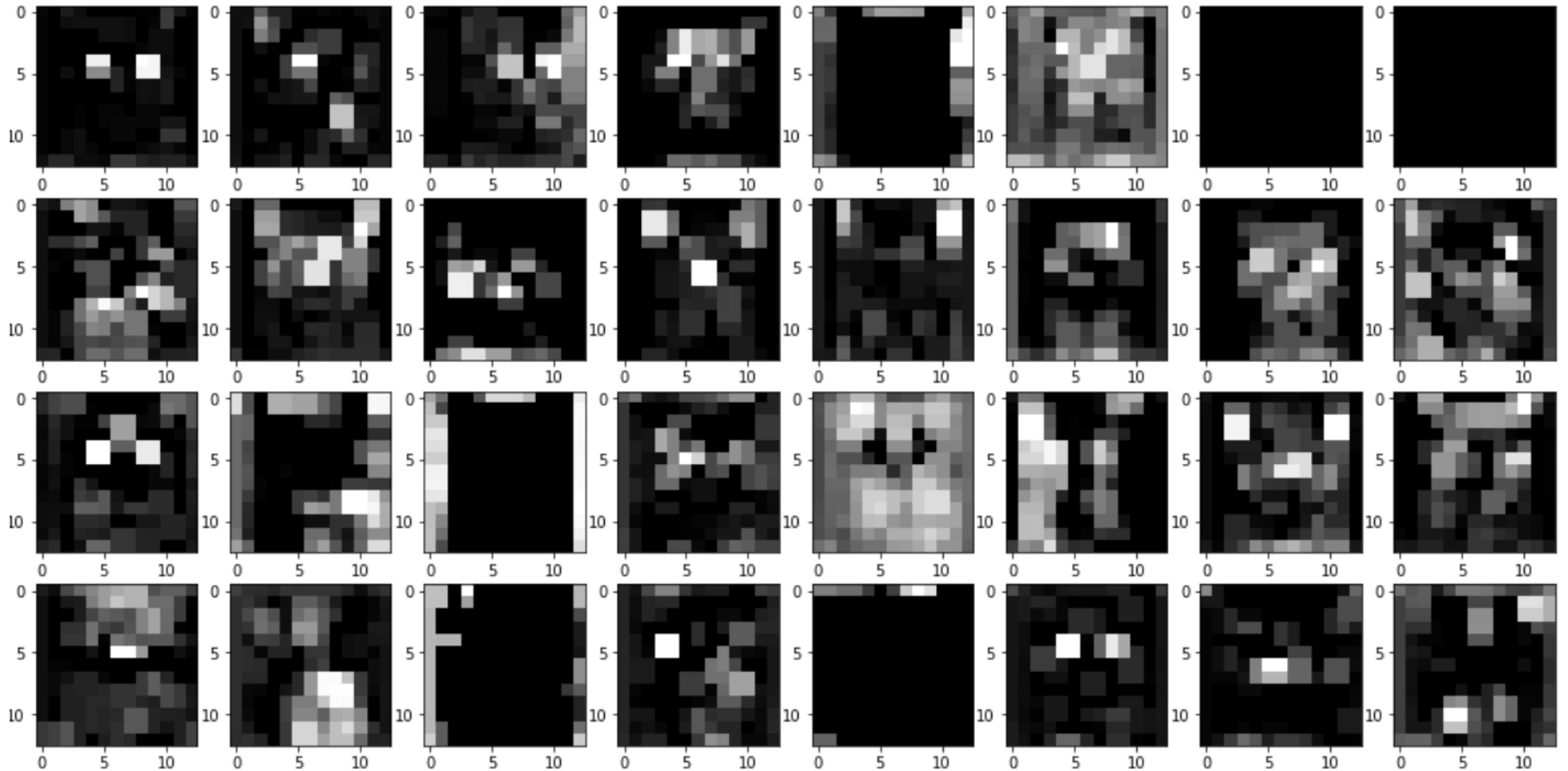
torch.Size([1, 192, 13, 13])

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

```
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
  )
```

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

Feature map after 2nd convolution and max pooling

3rd convolution

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

```
[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

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

```
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
    )
)
```

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

PYTORCH

Feature map after 3rd convolution

torch.Size([1, 384, 13, 13])
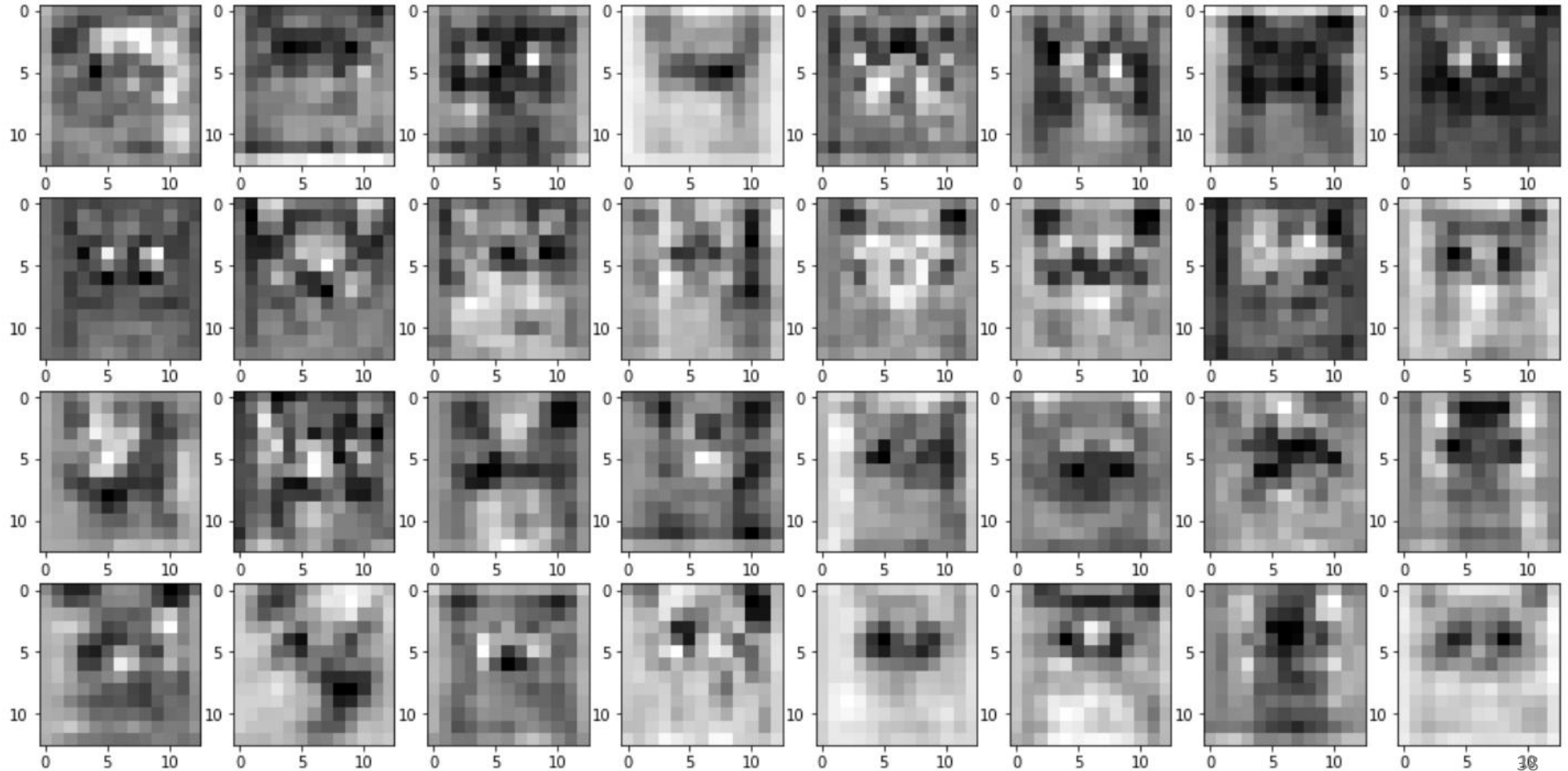
Apply max pooling to feature map from 3rd convolution
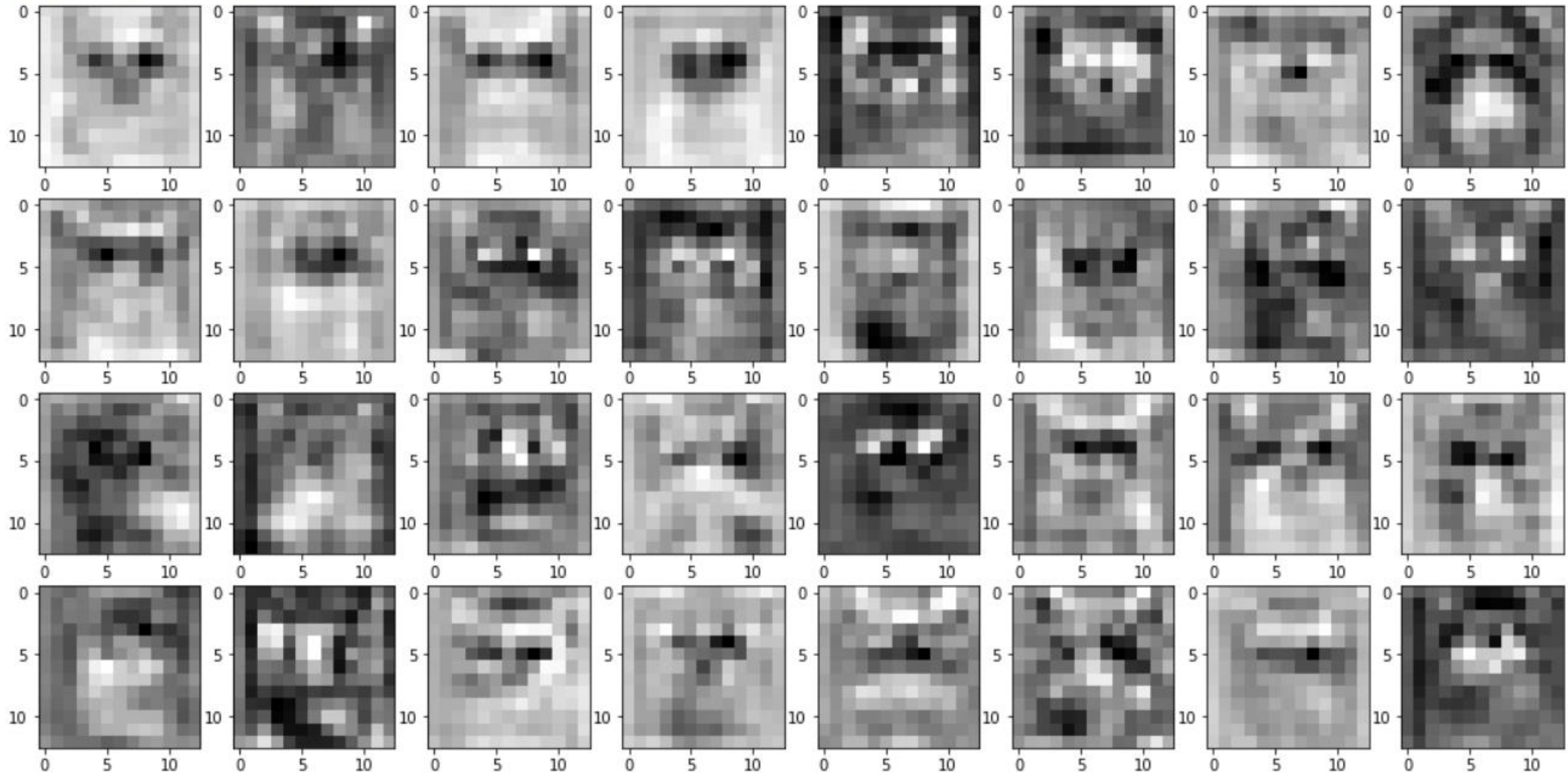
features[7, 8]

```
[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
    )
```

Feature map after 3rd convolution and max pooling

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

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)
  (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)
)
```

Google colab PYTÖRCH

Practice: Draw the structure of AlexNet

```
(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)
)
```