

Image classification

6.1. CNN (AlexNet).ipynb

Recap: Regression and Classification

$$L = \frac{1}{N} \sum_{i=1}^N (y^i - \hat{y}^i)^2$$

- Regression
- Classification

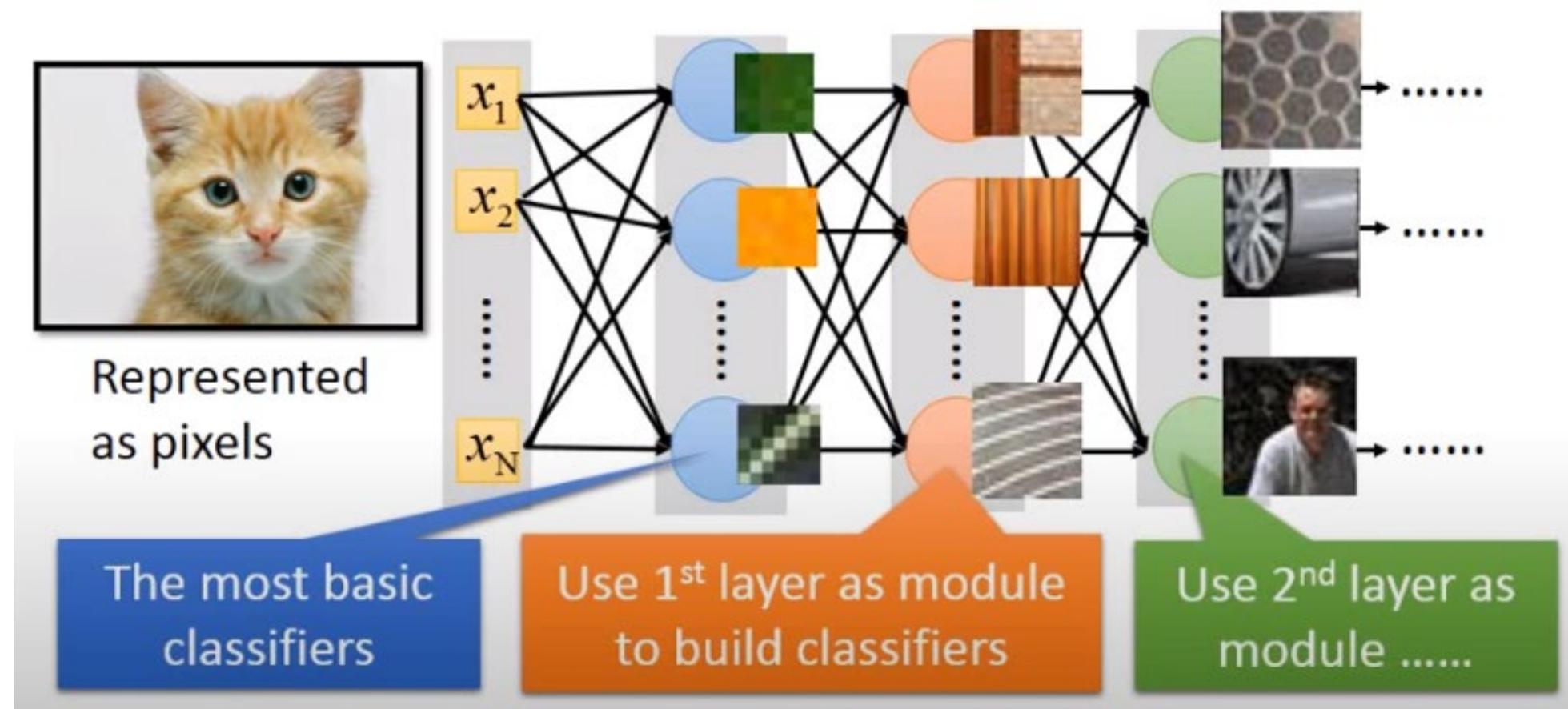
$$y = f(x)$$

$$L = \frac{1}{N} \sum_{i=1}^N CE(p, \hat{p})$$

$$CE(p, \hat{p}) = - \sum_{k=1}^C p_k \ln(\hat{p}_k)$$

Why not using MLP for image classification?

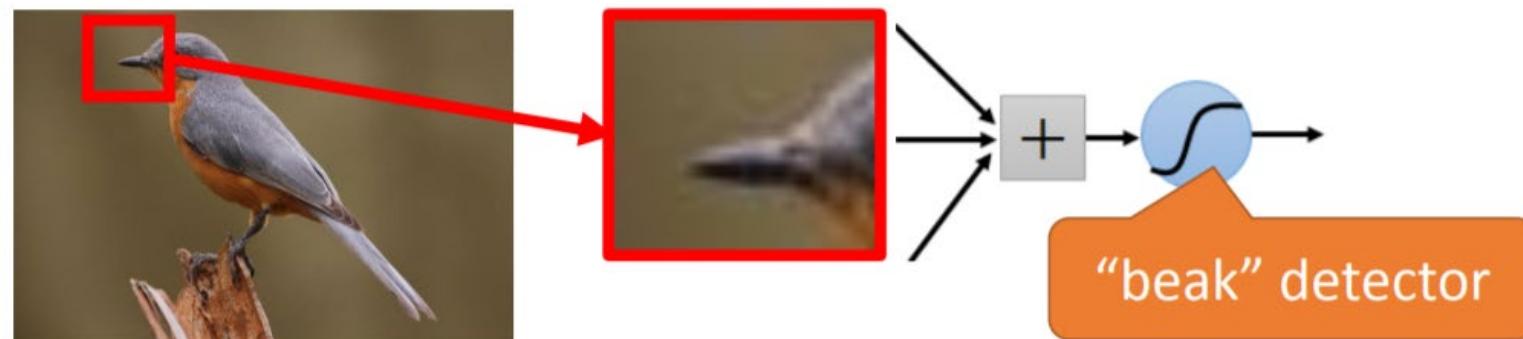
If we feed an image to MLP, then each neuron "sees" the whole image's pixels.



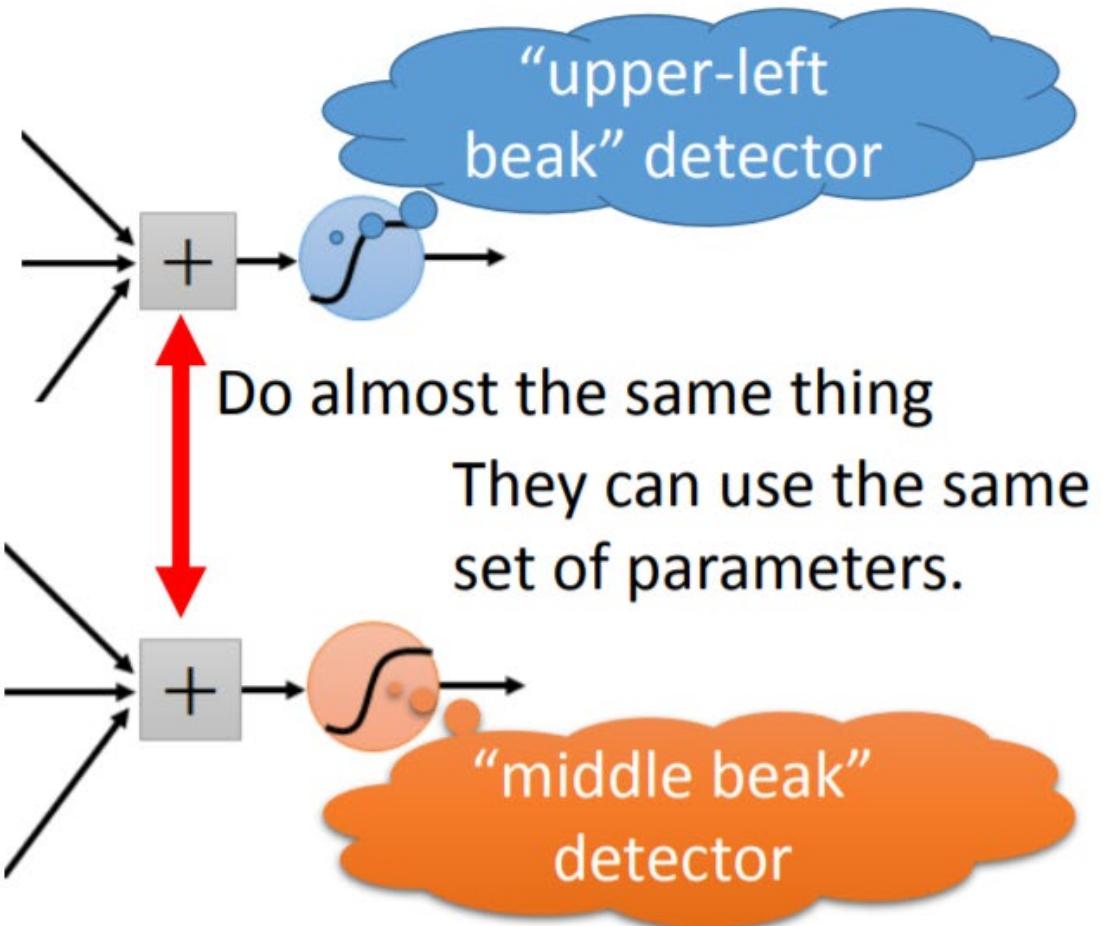
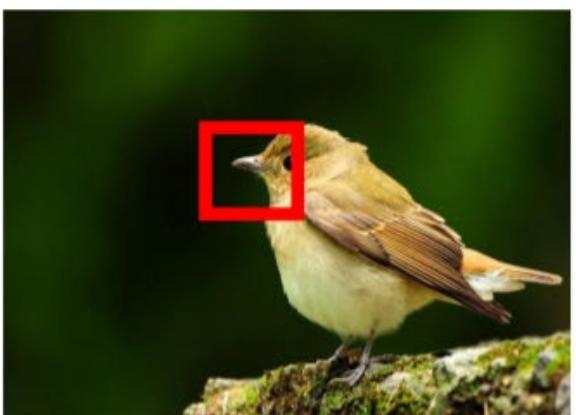
Some patterns are much smaller than the whole image

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



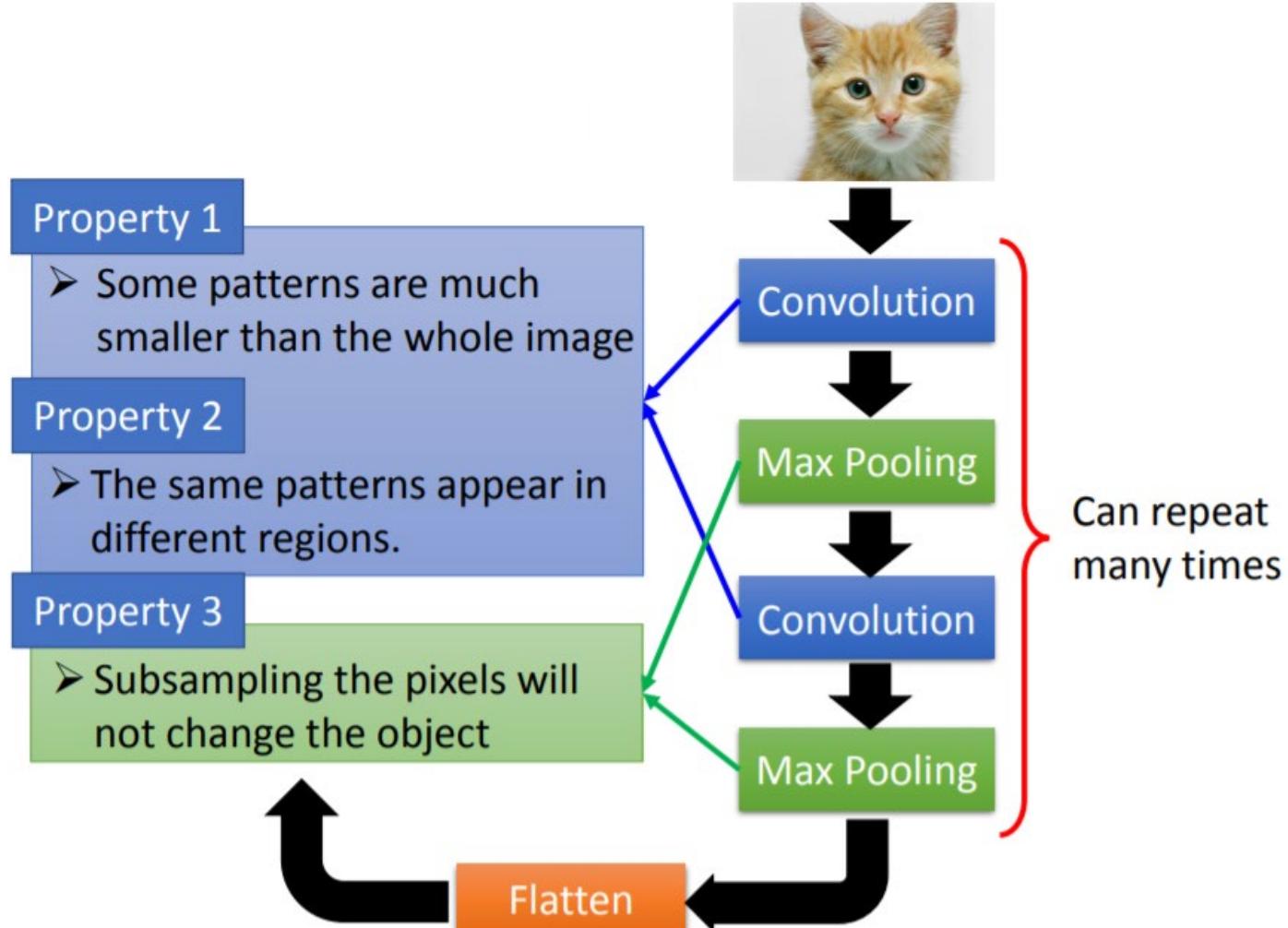
Subsampling the pixels will not change the object



We can subsample the pixels to make image smaller

→ Less parameters for the network to process the image

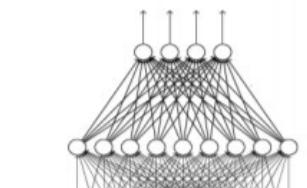
Use convolution and pooling operations to extract important features from input image



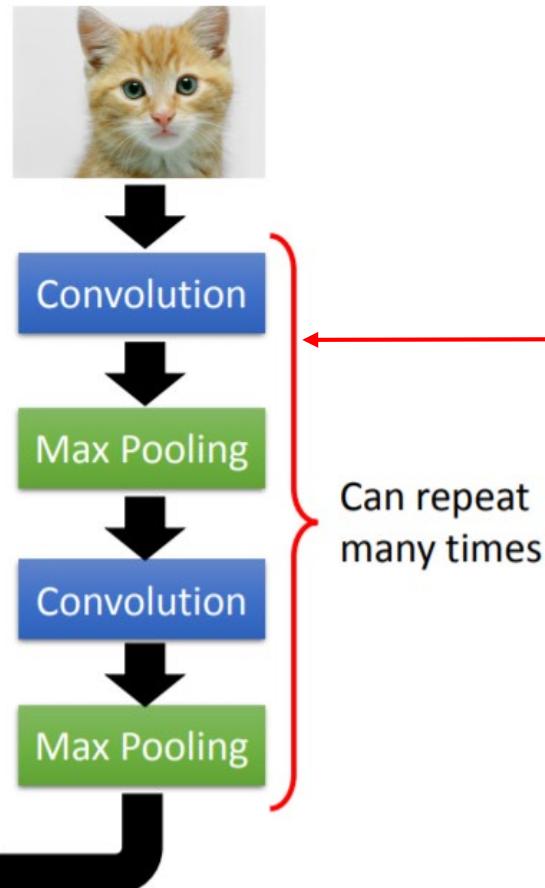
CNN contains two sections: "features" and "classifier"

The whole CNN

cat dog



Fully Connected
Feedforward network



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

AlexNet

```
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, stride=(2, 2))
        (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, stride=(2, 2))
        (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, stride=(2, 2))
    ),
    (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)
    )
)
```

Load pre-trained CNN for image classification

```
[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/>

Deep neural networks for image understanding

Alex Net
VGG16
Res Net

U Net

Yolo
Faster RCNN

Mask RCNN

OpenPose
Keypoints RCNN

Classification



Semantic Segmentation



GRASS, CAT,
TREE, SKY

Object Detection



DOG, DOG, CAT

Instance Segmentation



DOG, DOG, CAT

Joint detection



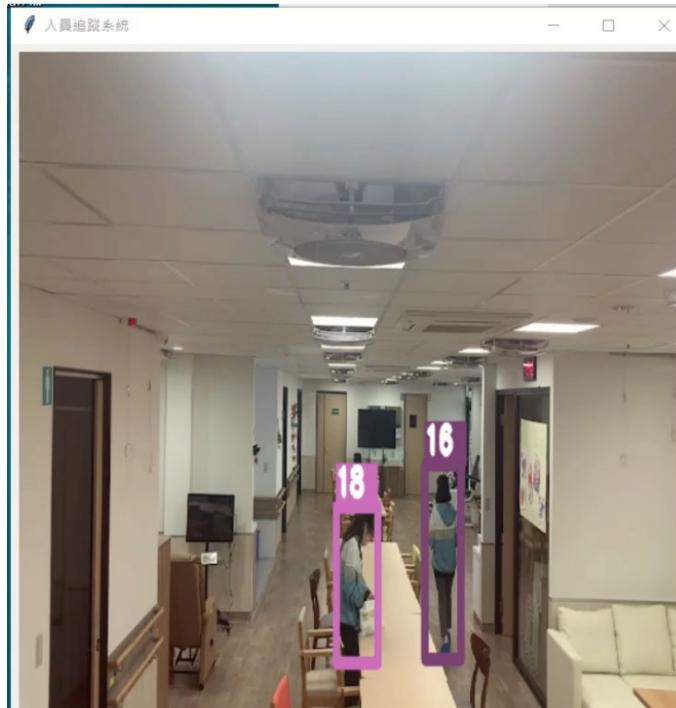
圖片來源: <https://kharshit.github.io/blog/2019/08/23/quick-intro-to-instance-segmentation>

Image sequence understanding

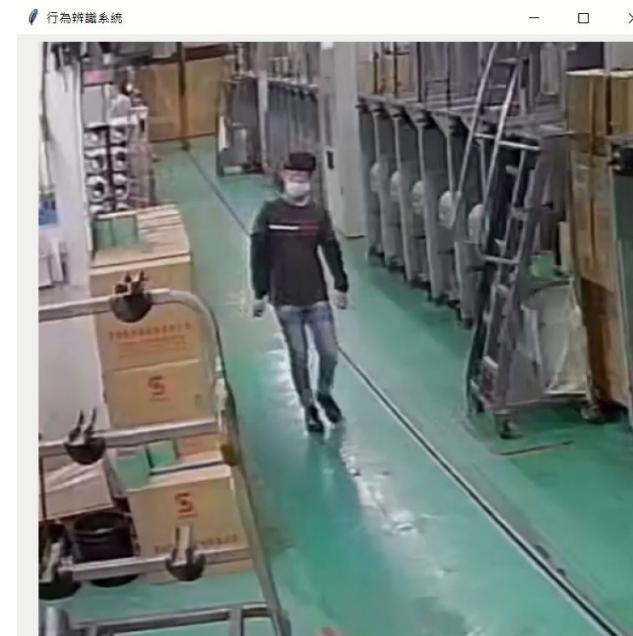
**SORT, ByteTrack
DeepSORT, JDE**

SlowFast

Object tracking



Action classification



21:10:22 Action = spraying 0.22, cleaning floor 0.18, garbage collecting 0.16,
21:10:32 Action =

Pre-trained NN for CV

Alex Net

Faster RCNN

VGG16

Keypoints

Res Net

RCNN



SlowFast



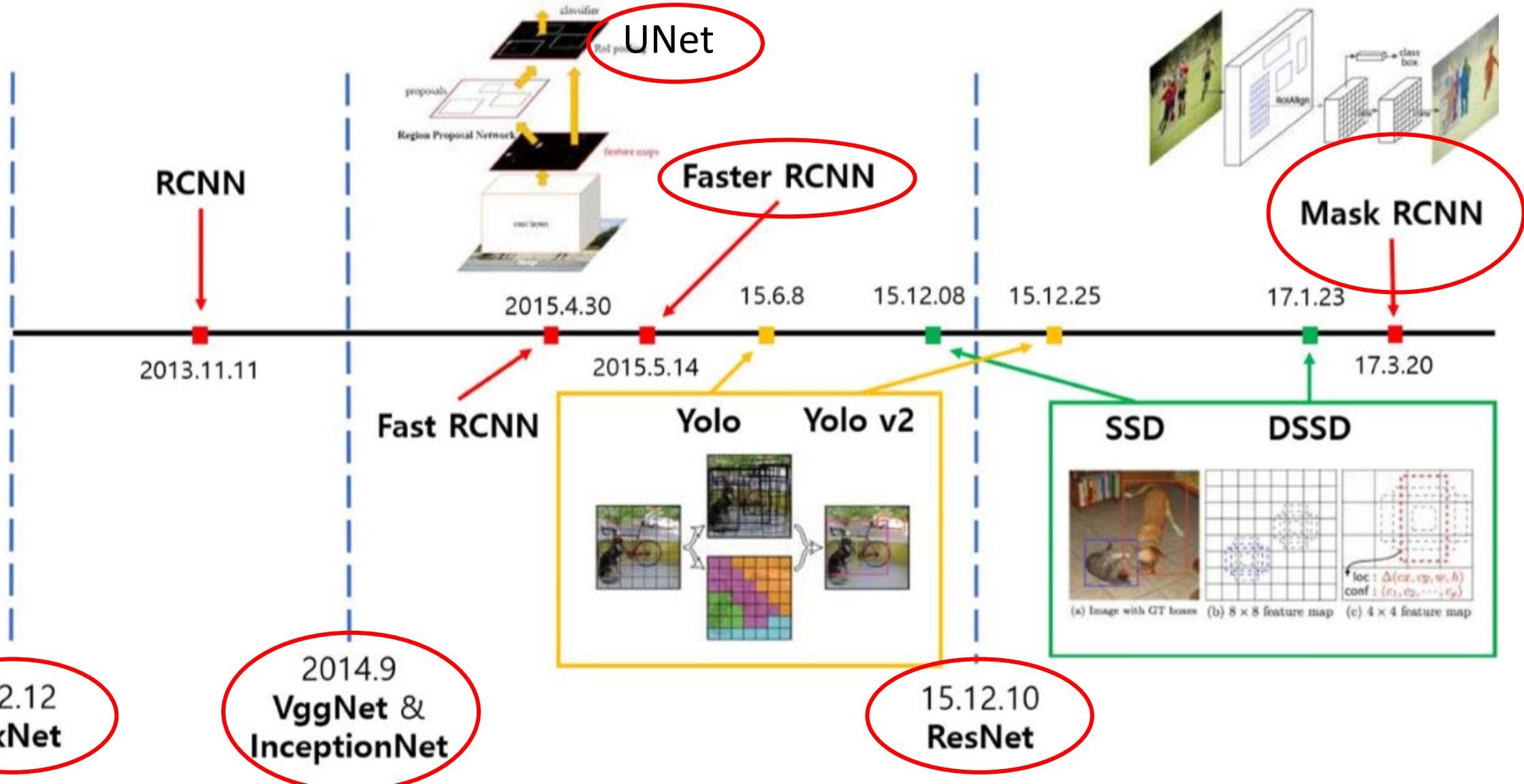
SORT, ByteTrack

DeepSORT, JDE

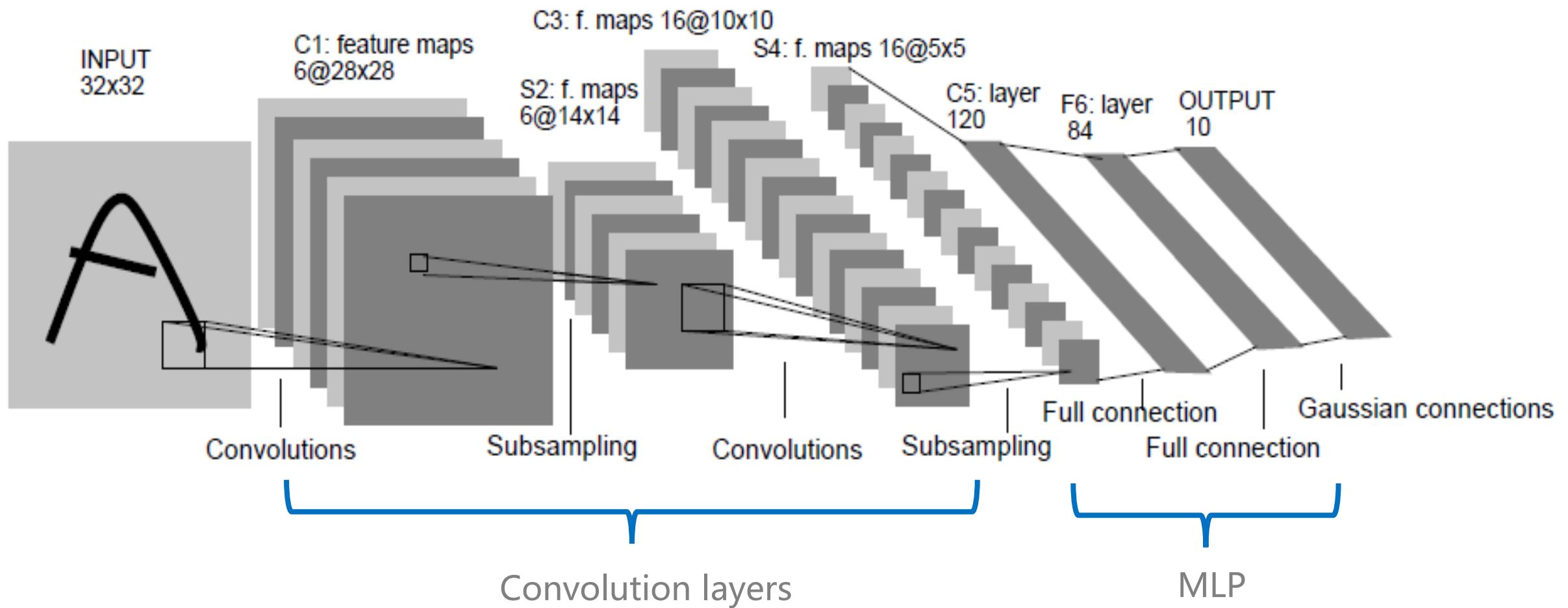


<https://github.com/microsoft/computervision-recipes>

History of CNN families



LeNet



LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278-2324.

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()
```

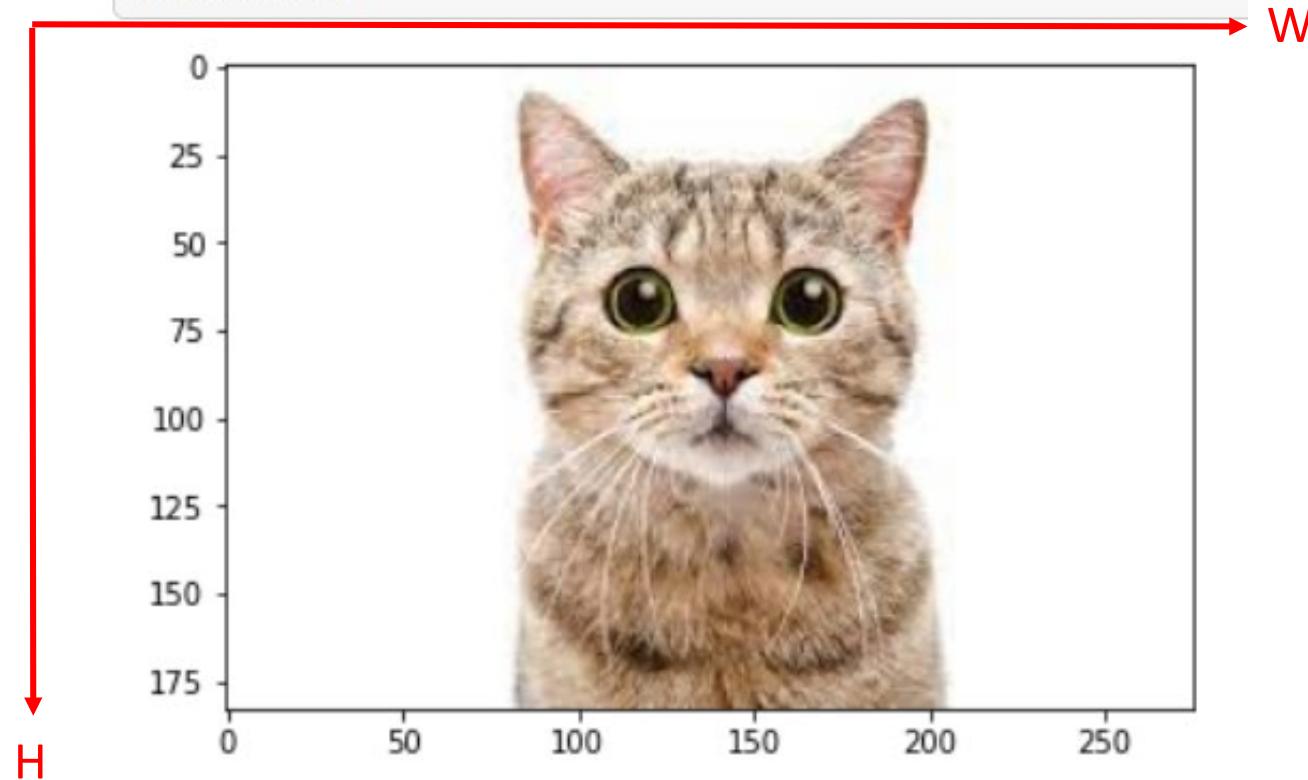


Image pre-processing

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

```
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]))
```

Feature scaling

```
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 shape = [batch size, input data size]

Input to CNN

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

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

Input to MLP

2.1. Regression HW.ipynb

```
# test mini-batch  
for (batchX, batchY) in loader:  
    break  
print(batchX.shape, batchY.shape)
```

```
torch.Size([500, 7]) torch.Size([500, 1])
```

4. Classifier.ipynb

```
torch.size([batch size, input data size])  
torch.size([batch size, 1])
```

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))
```

Practice: Convolution operation

Excel: SUMPRODUCT(X:X, X:X)

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

6 x 6 image

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

Filter size: 3x3

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	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	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	-2	1
-1	-1	-2	1
-1	0	-4	3

Filter + Convolution to extract features

Filter searches patterns in a small region

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

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).

Filter searches a particular pattern in different regions

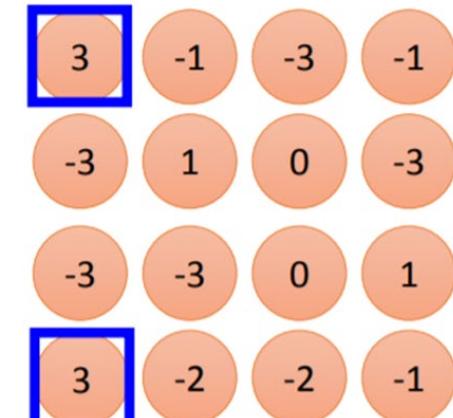
stride=1

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

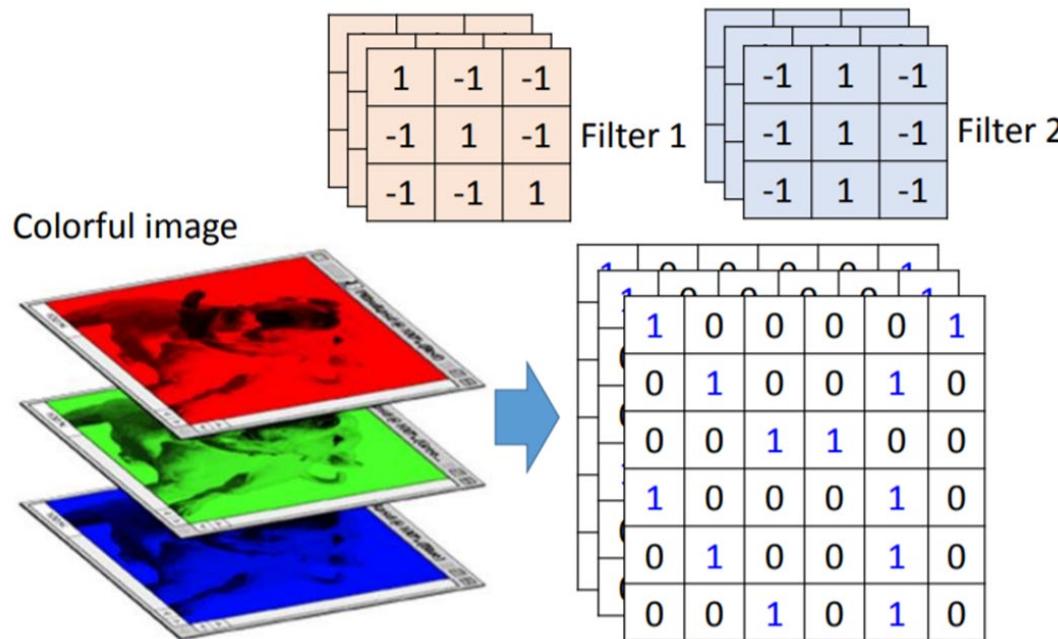
Filter 1



Property 2

Filter has depth

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



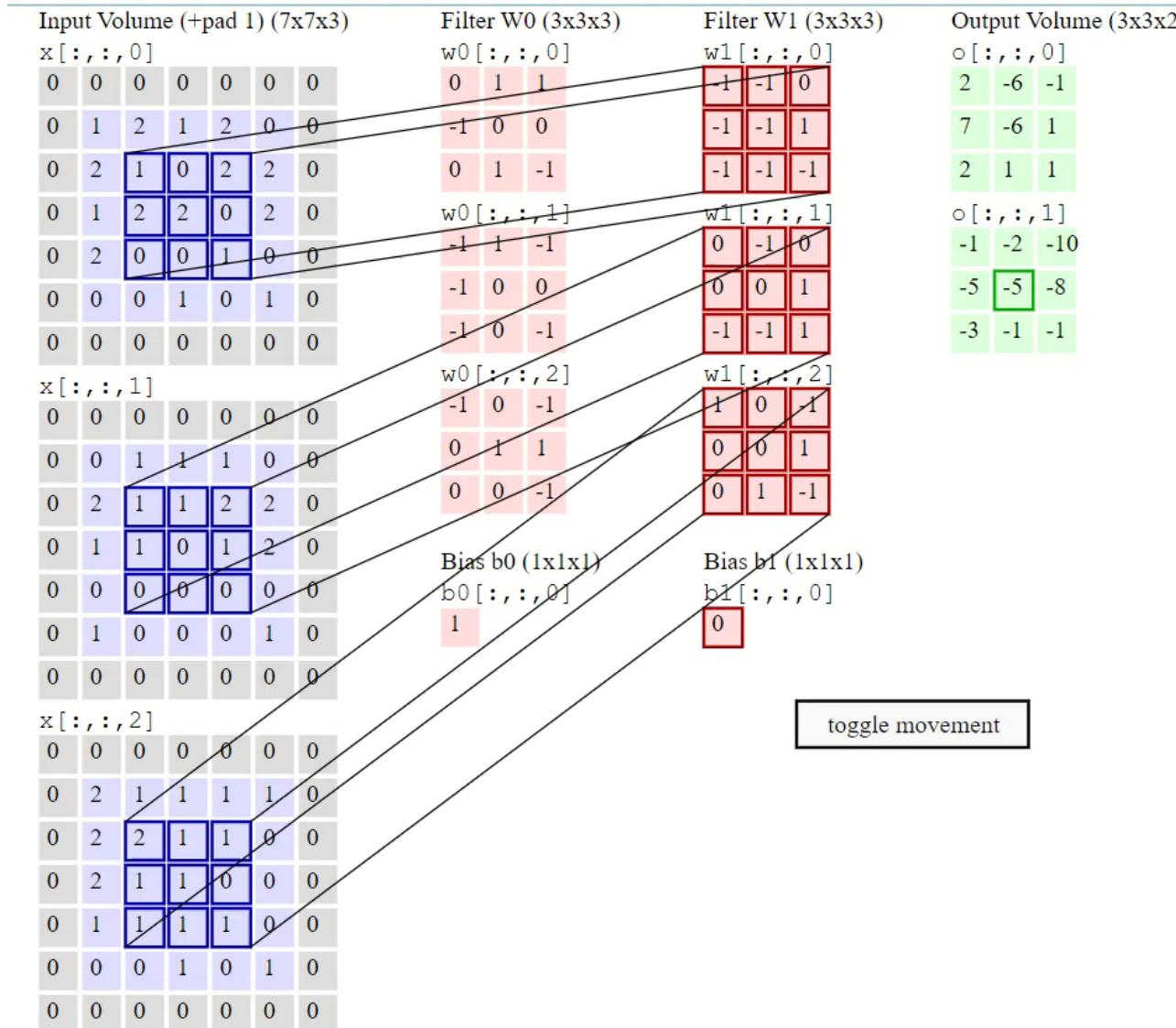
```
[10] conv1 = model.features[0]
     print(conv1)
     #InChannel=3 (RGB), OutChannel=64, filter size=11,
```

```
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
```

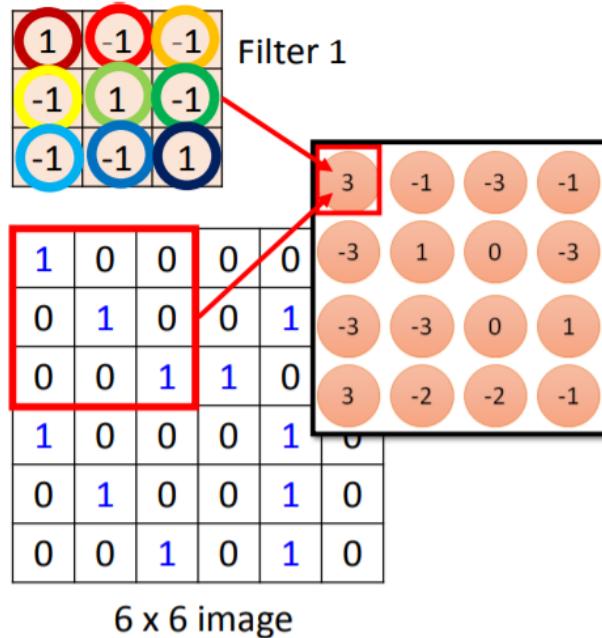
(64, 3, 11, 11)

How filter and convolution work

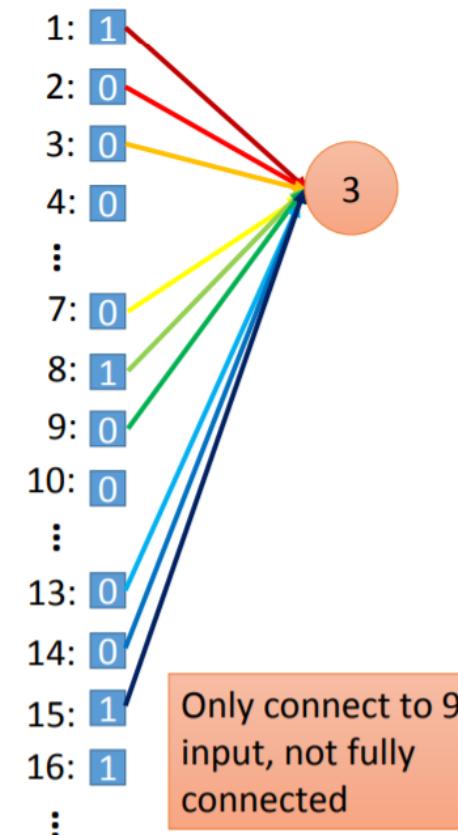


Why convolution helps?

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

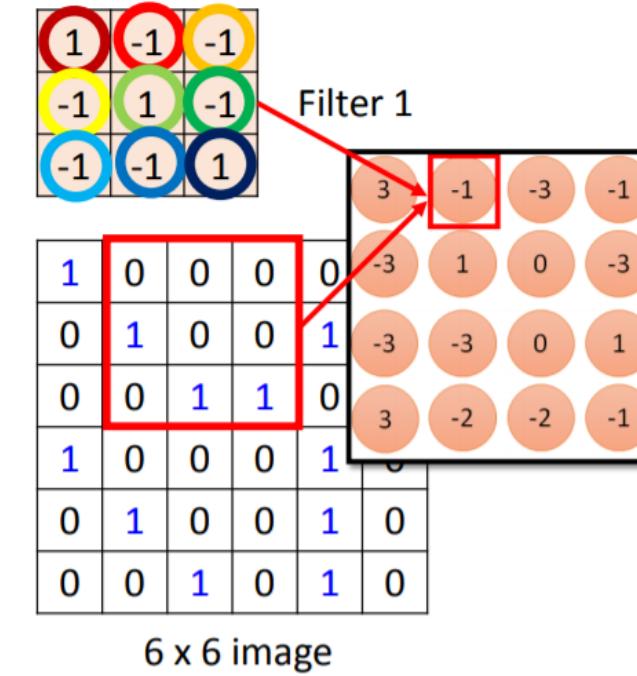


Less parameters!



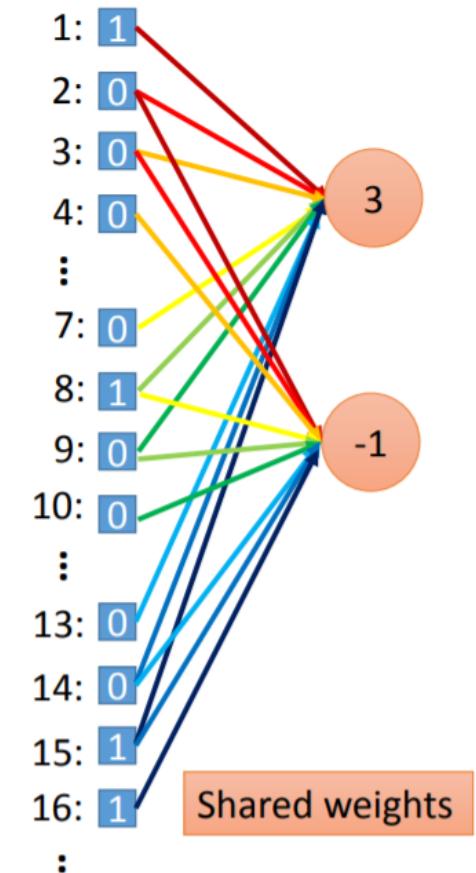
Reference: 李弘毅 ML Lecture 10 <https://youtu.be/FrKWiRv254g>

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



Less parameters!

Even less parameters!

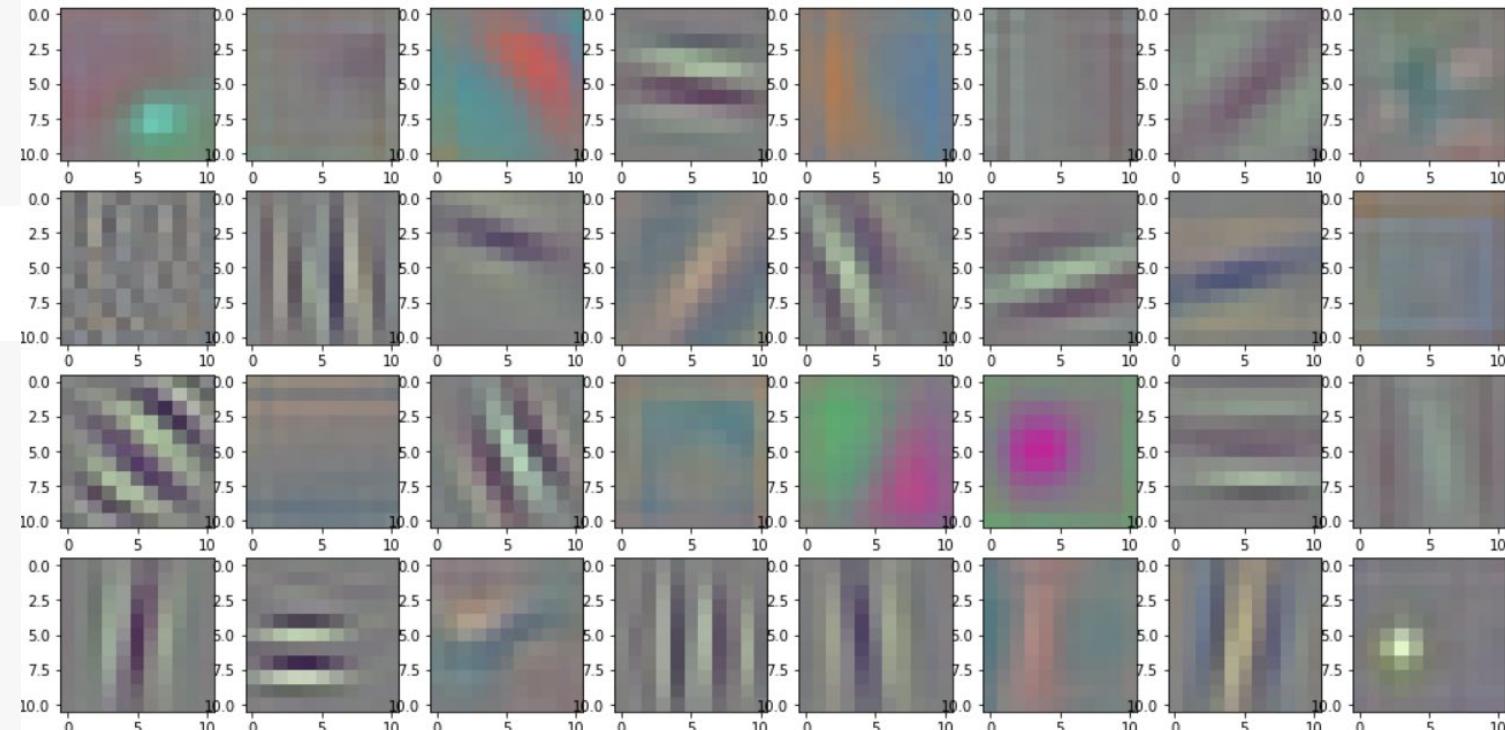


Filter weights of 1st convolution layer in AlexNet

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

(64, 3, 11, 11)

```
[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()
```



Feature maps

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

stride=1

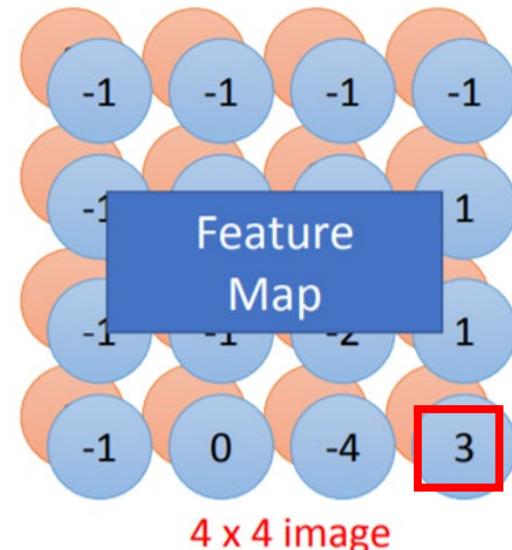
1	0	0	0	0	0	1
0	1	0	0	0	1	0
0	0	1	1	0	0	0
1	0	0	0	1	0	0
0	1	0	0	0	1	0
0	0	1	0	0	1	0

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=(11, 11), stride=(1, 1)
```

Conv2d(3, 64, kernel_size=(11, 11), stride=(1, 1))

```
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
```

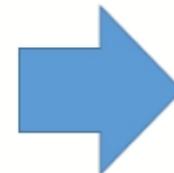
Out[12]: torch.Size([1, 64, 55, 55])

Practice: Calculate feature map h and w

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



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



-1	-1	-1	-1
-1	-1	-2	1
-1	-1	-2	1
-1	0	-4	3

conv2d(in-channel, No of filters, kernel_size=3, stride = (1,1), padding = (0, 0))

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

$$H = \frac{6 + 2 * 0 - 3}{1} + 1 = 4 \quad w = \frac{6 + 2 * 0 - 3}{1} + 1 = 4$$

conv2d(in-channel, No of filters, kernel_size=3, stride = (2,2), padding = (1, 1))

$$H = \frac{6 + 2 * 1 - 3}{2} + 1 = 3.5$$

Width and height of feature maps

```
AlexNet(
    (features): Sequential(
        (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
        (1): ReLU(inplace=True))

    imageTensor = torch.unsqueeze(PILImg, 0)
    imageTensor.shape

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

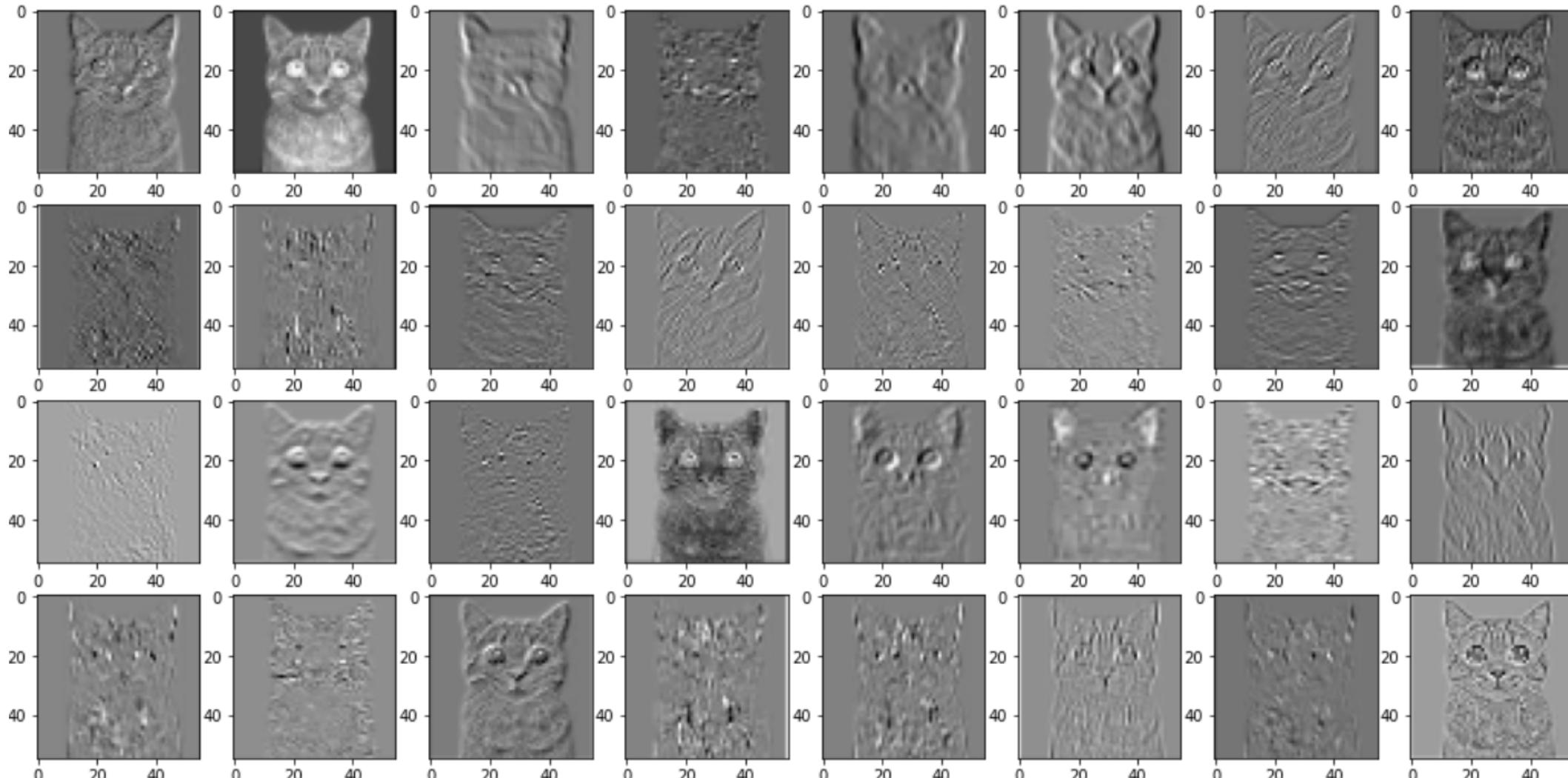
$$H_{out} = \frac{H_{in} + 2 \times padding - kernel\ size}{Stride} + 1$$

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

```
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])`

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



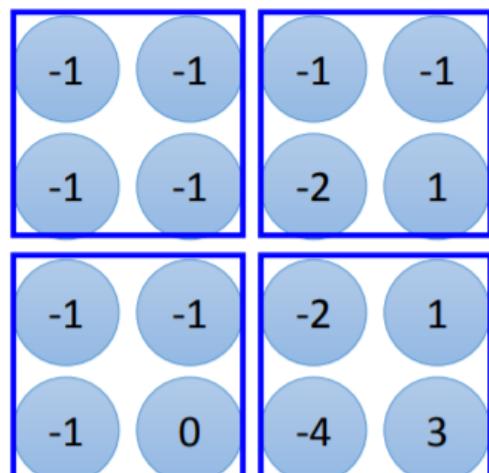
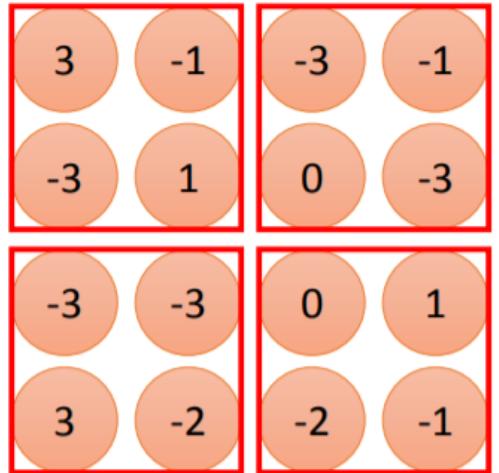
Max pooling

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

Filter 1

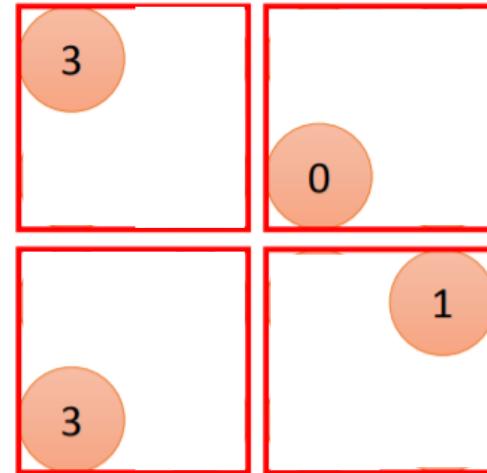
-1	1	-1
-1	1	-1
-1	1	-1

Filter 2



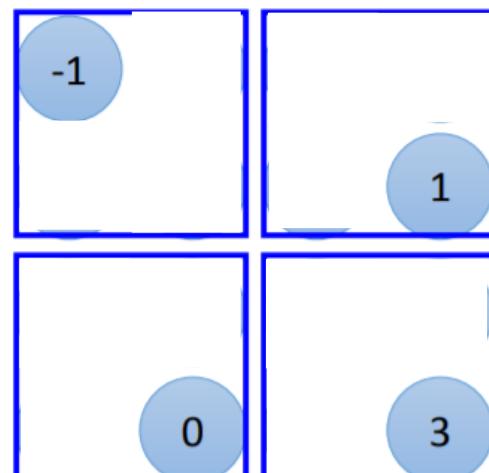
1	-1	-1
-1	1	-1
-1	-1	1

Filter 1



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

Filter 2



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 channels
    fig.add_subplot(4, 8, i+1)
    plt.imshow(imgArray[i], cmap='gray')
plt.show()

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

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

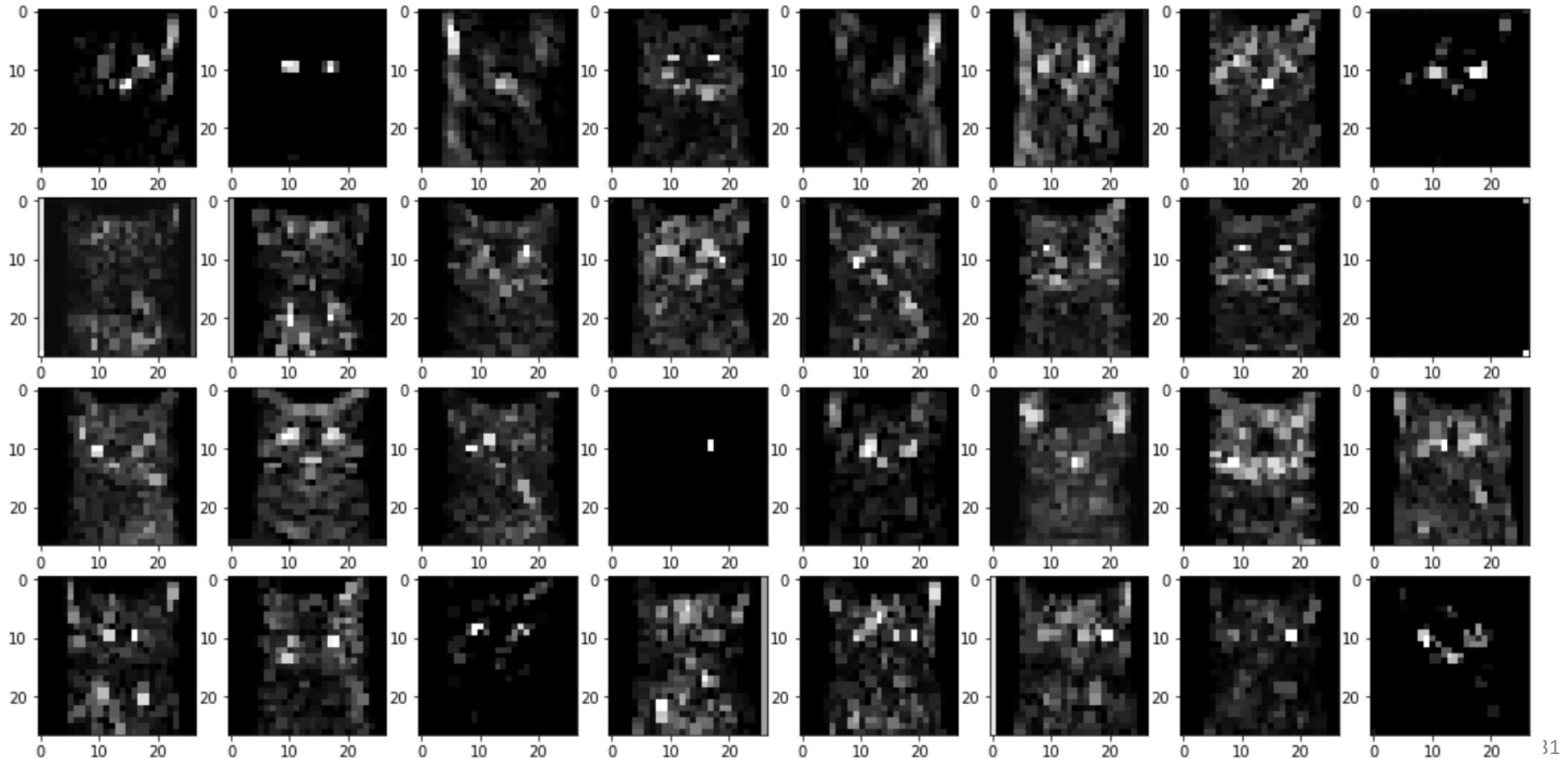
$$H_{out} = \frac{H_{in} + 2 \times padding - kernel\ size}{Stride} + 1$$

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)
    (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)
    (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)
)
```



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



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

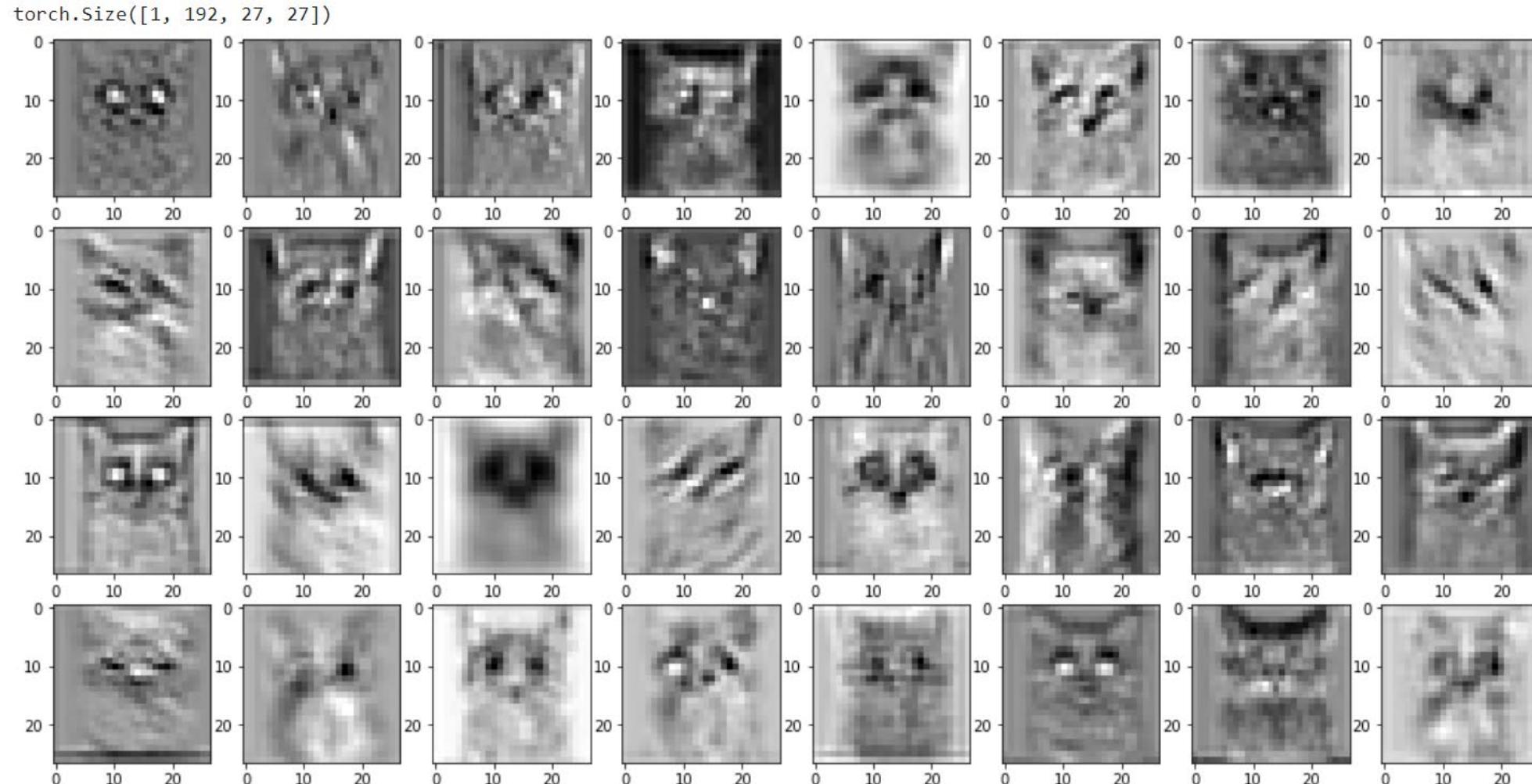
$$\frac{27 + 2 \times 2 - 5}{1} + 1 = 27$$

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

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
    )
```

Feature map after 2nd convolution



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])
```

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

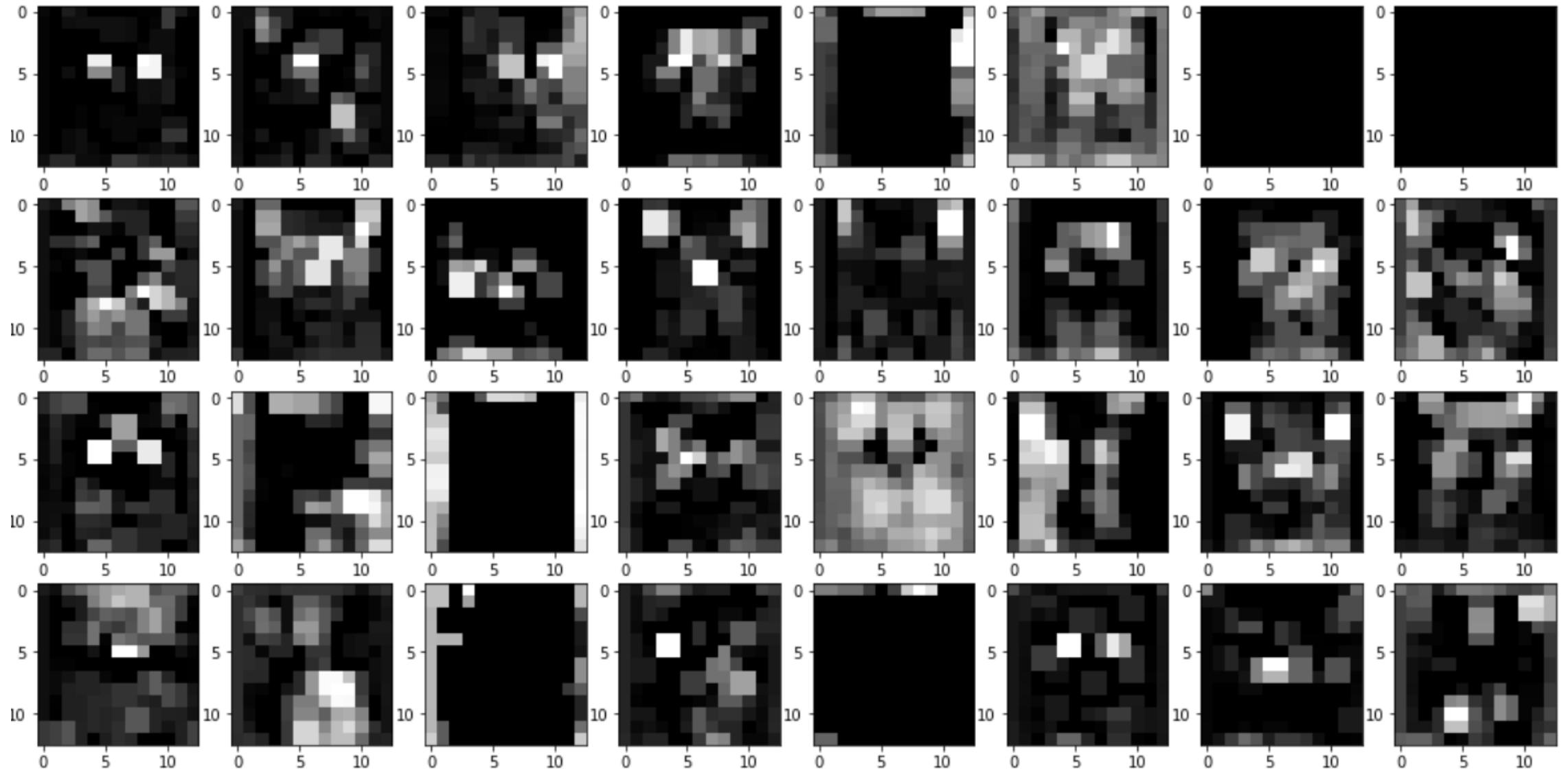
$$H_{out} = \frac{H_{in} + 2 \times padding - kernel\ size}{Stride} + 1$$

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
```

)

Feature map after 2nd convolution and max pooling



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

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

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

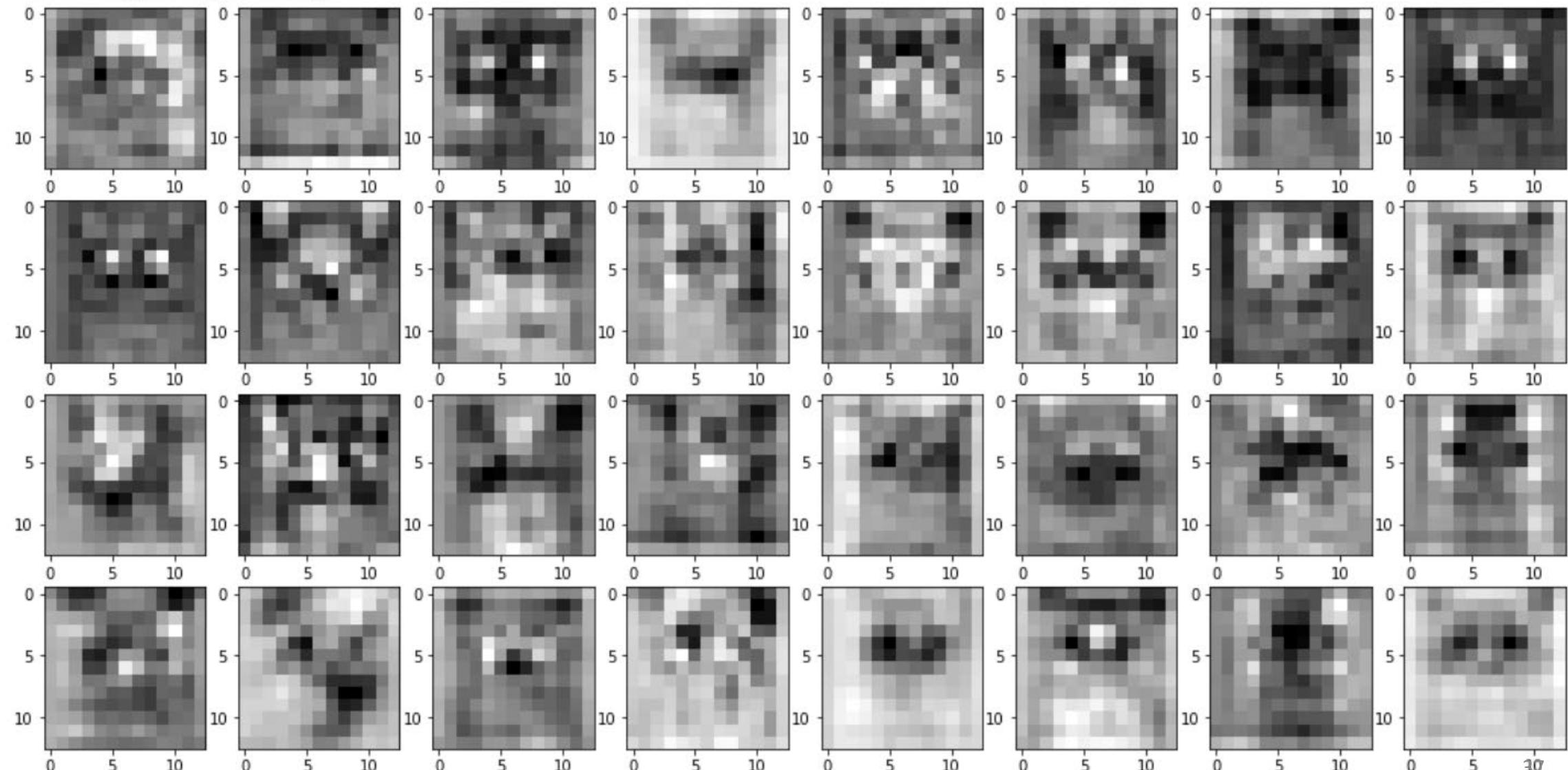
394 filters, each has 192 channels, are applied to the input feature map (with 192 channels)

```
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
```

)

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()

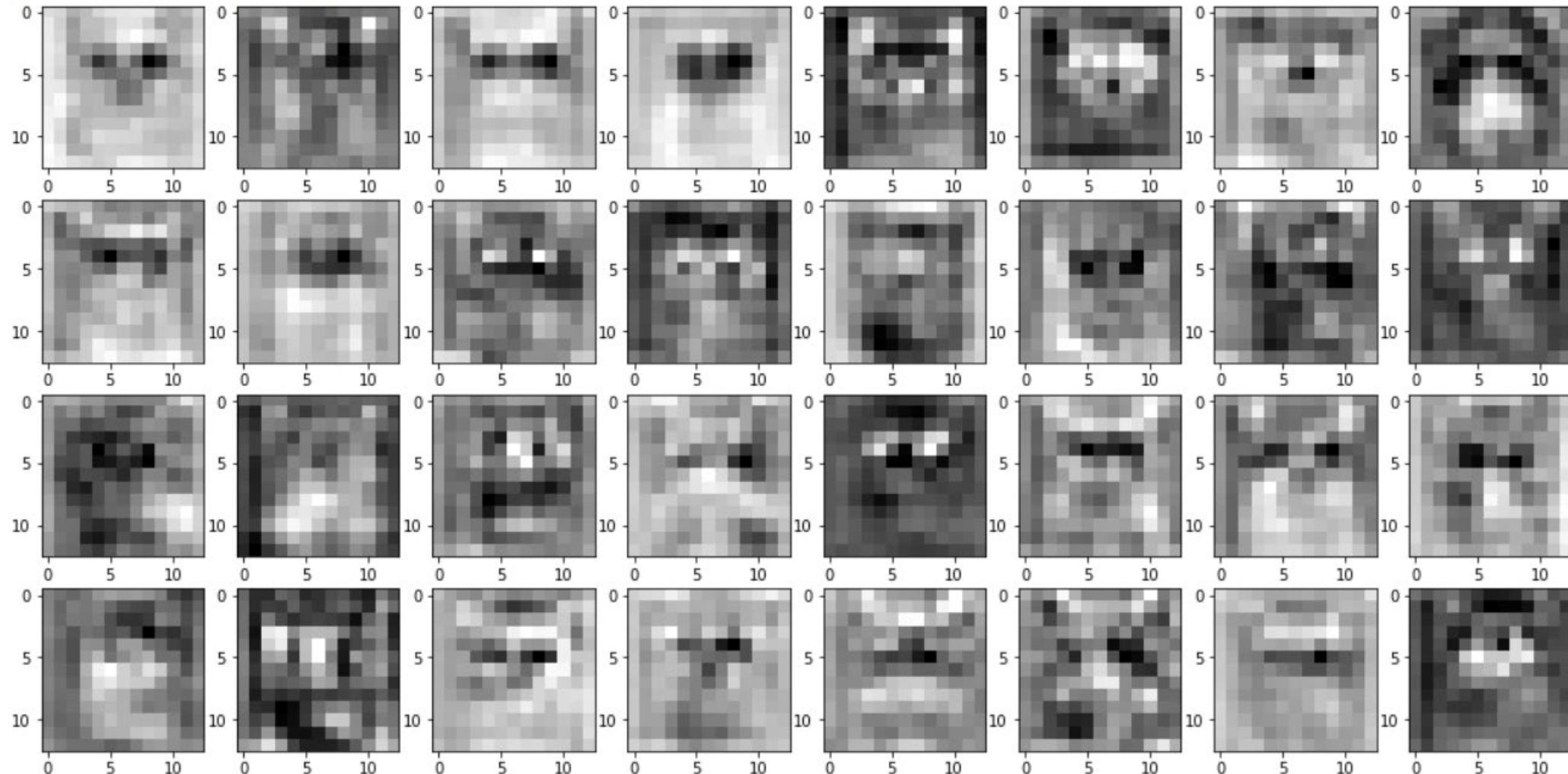
torch.size([1, 256, 13, 13])
```

AlexNet(

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

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])
```

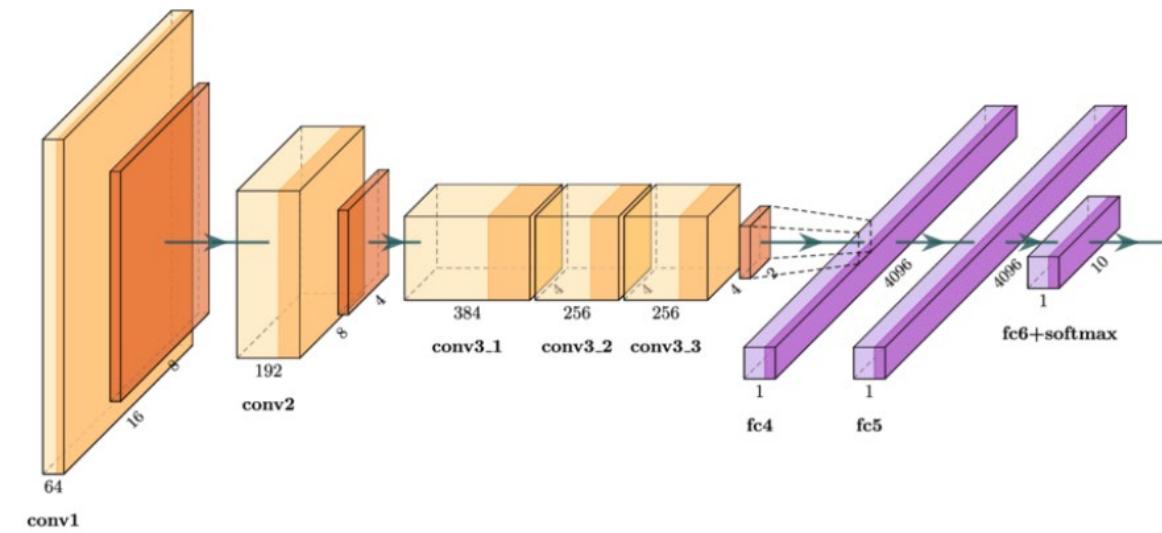
After last convolution and max pooling, the output feature map has 256 channels

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

```
(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) 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 – Draw 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)
)
```

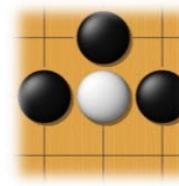


https://www.researchgate.net/figure/AlexNet-architecture-used-as-the-baseline-model-for-the-analysis-of-results-on-the_fig5_339756908

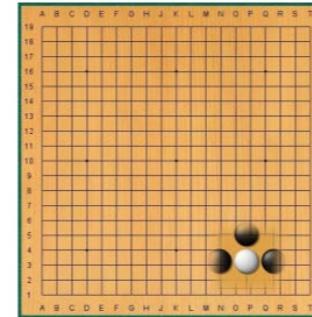
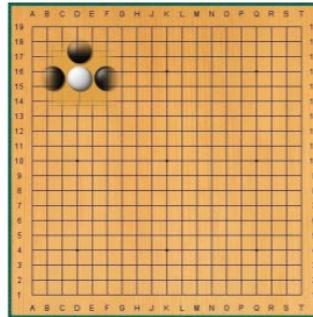
Why Alpha GO also use CNN?

- Some patterns are much smaller than the whole image

Alpha Go uses 5×5 for first layer



- The same patterns appear in different regions.



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.