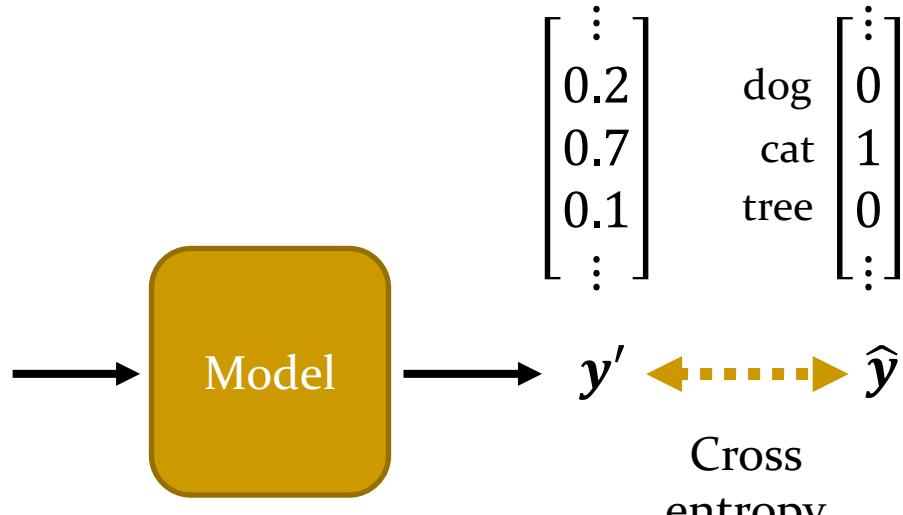


# Lecture 9

# Convolution Neural Network

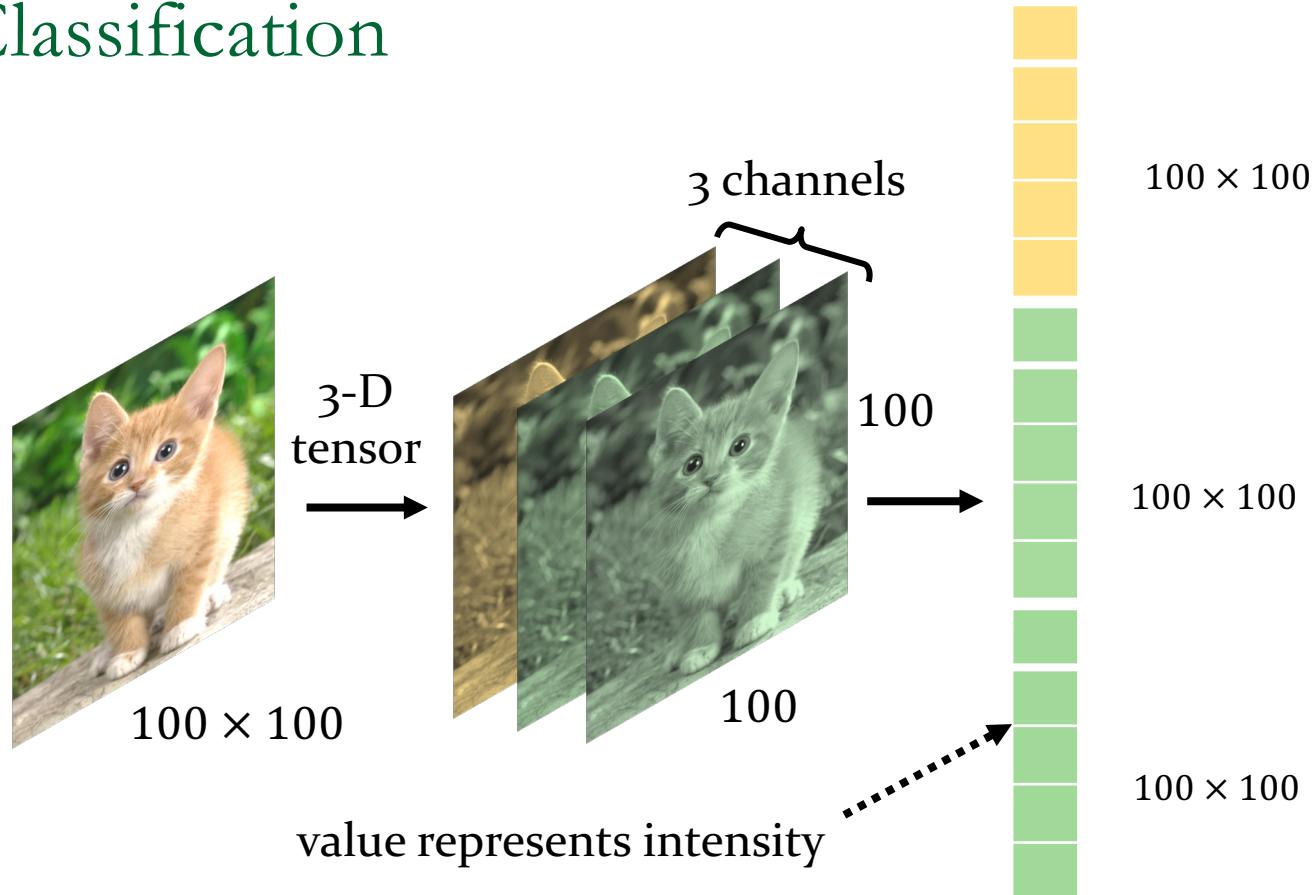


# Image Classification

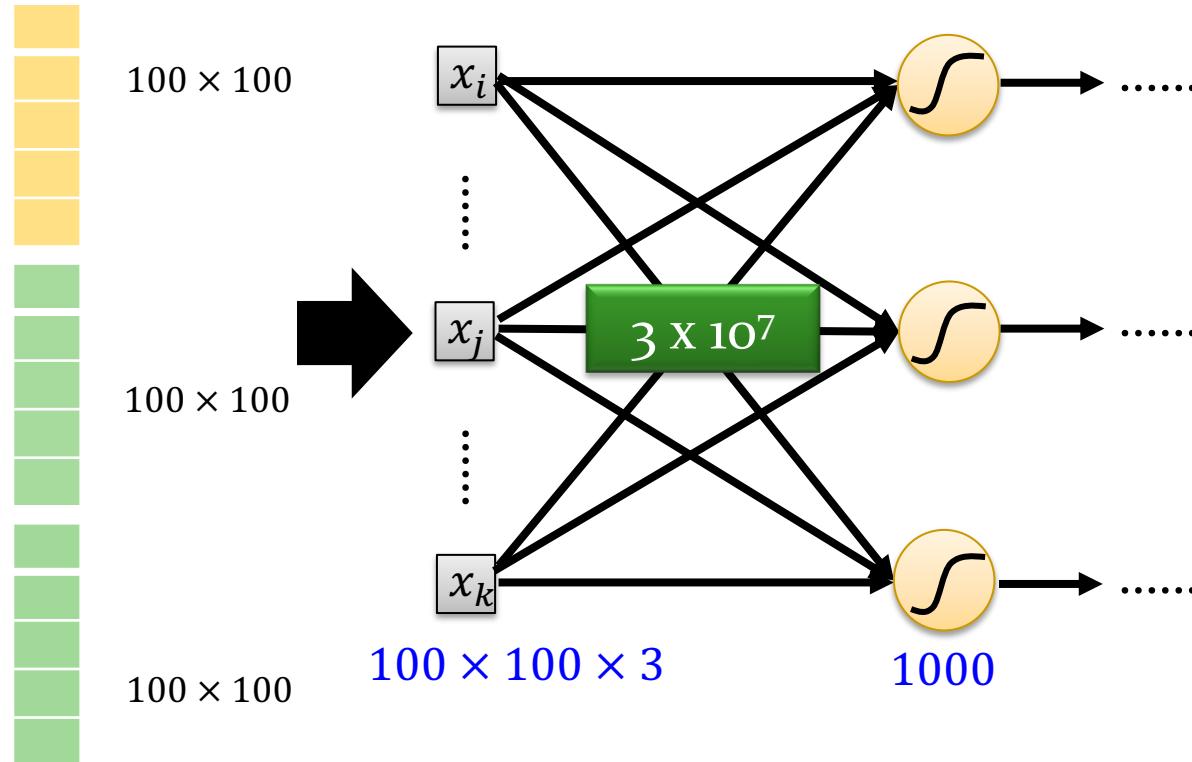


(All the images to be classified have the same size.)

# Image Classification

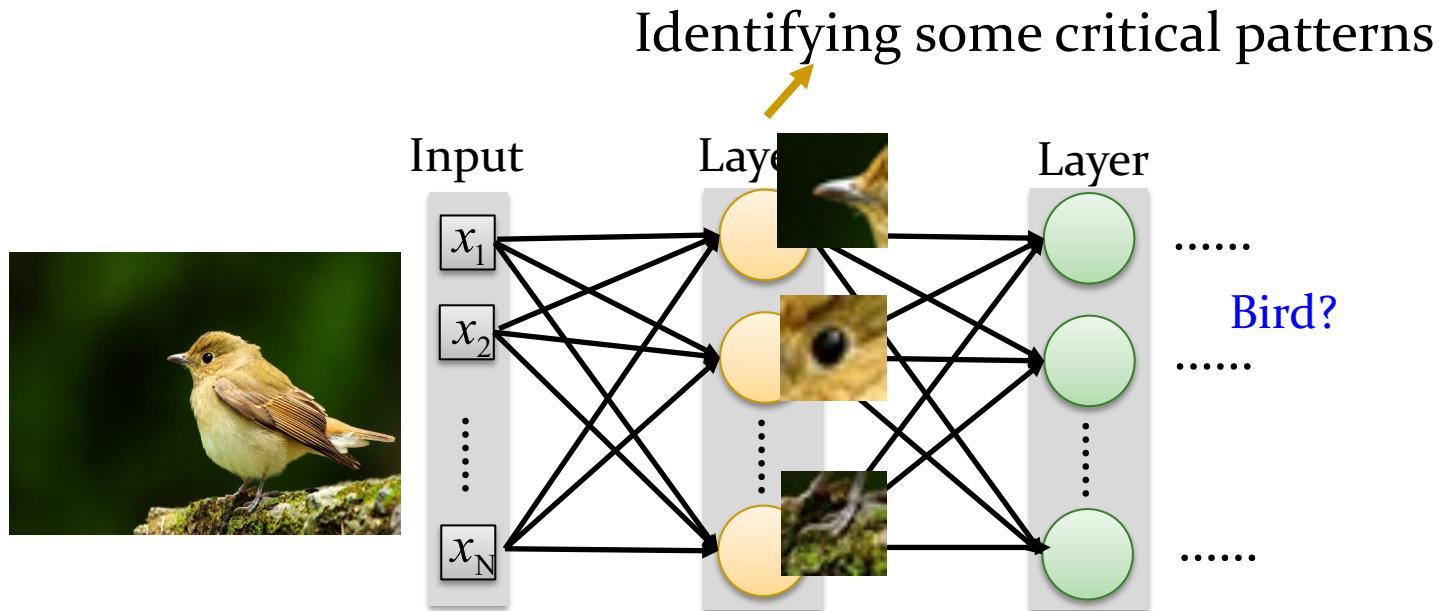


# Fully Connected Network



Do we really need “fully connected” in image processing?

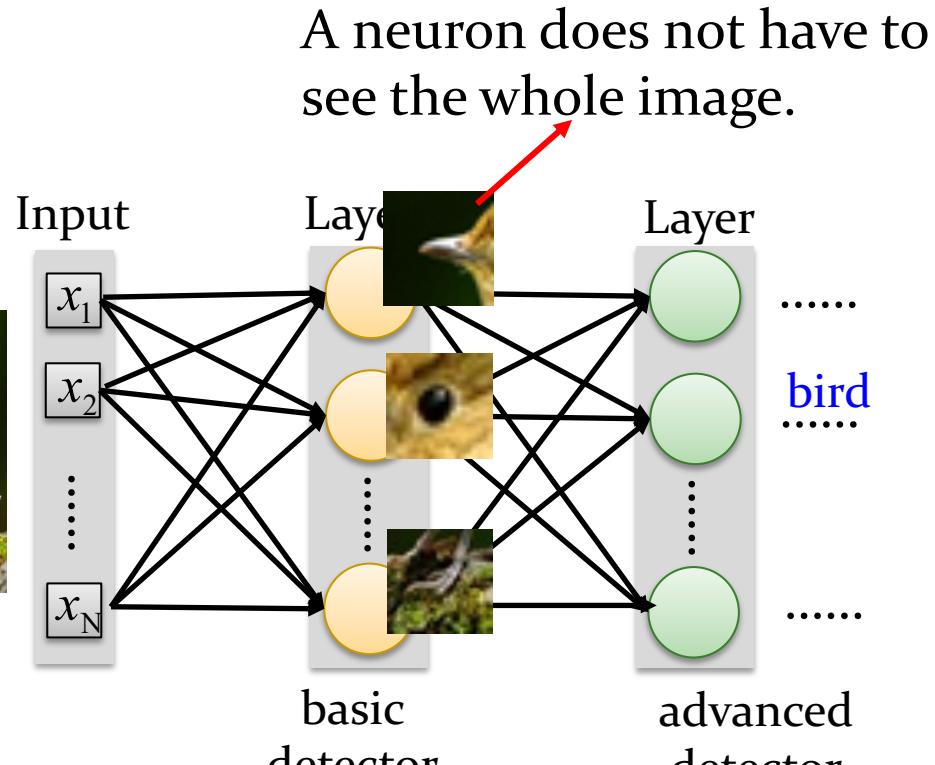
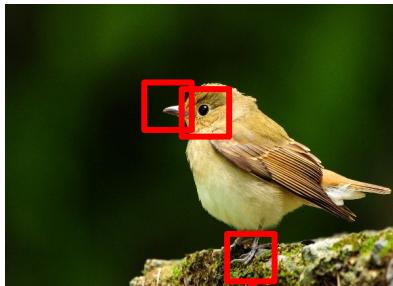
# Observation 1



Perhaps human also identify birds in a similar way ... ☺

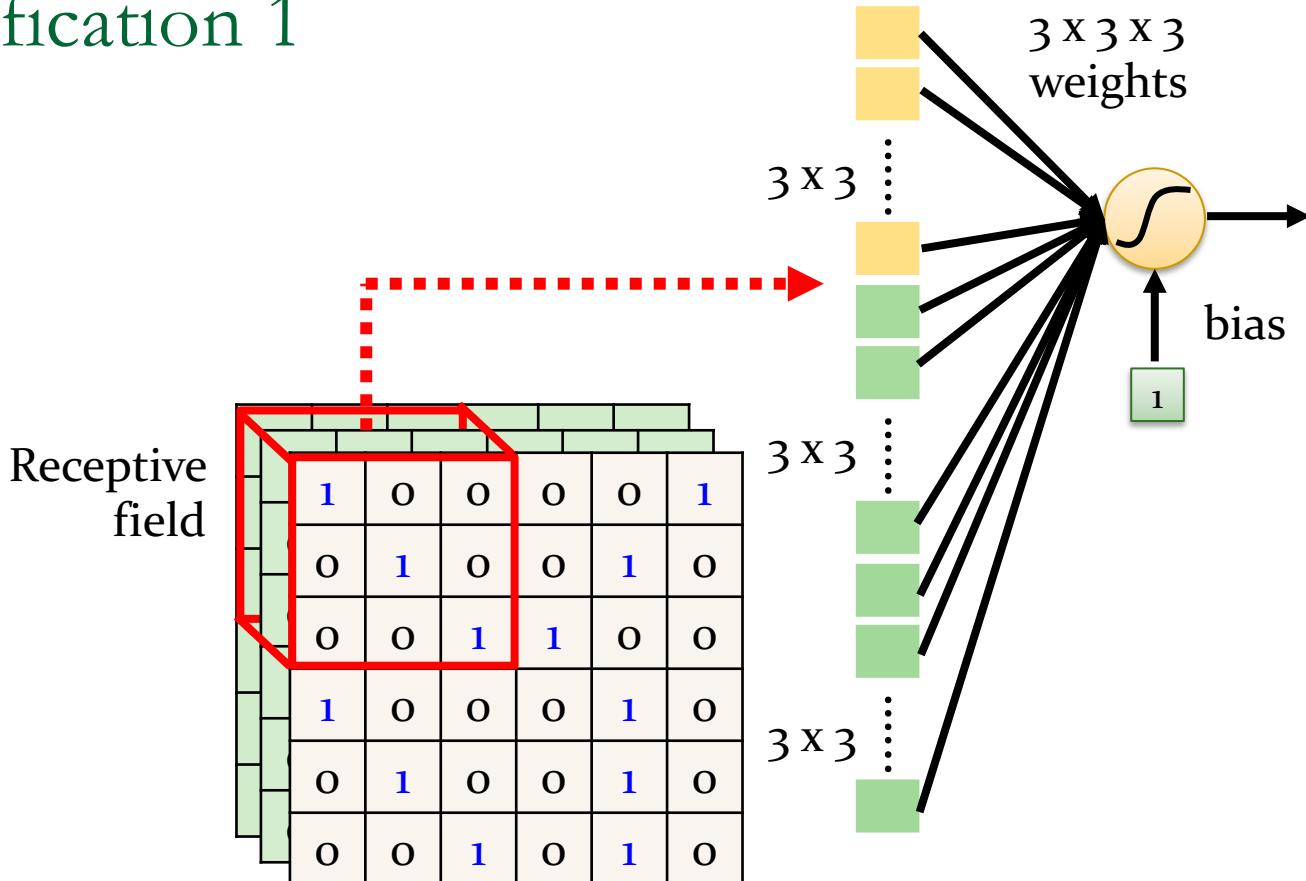
# Observation 1

Need to see the whole image?



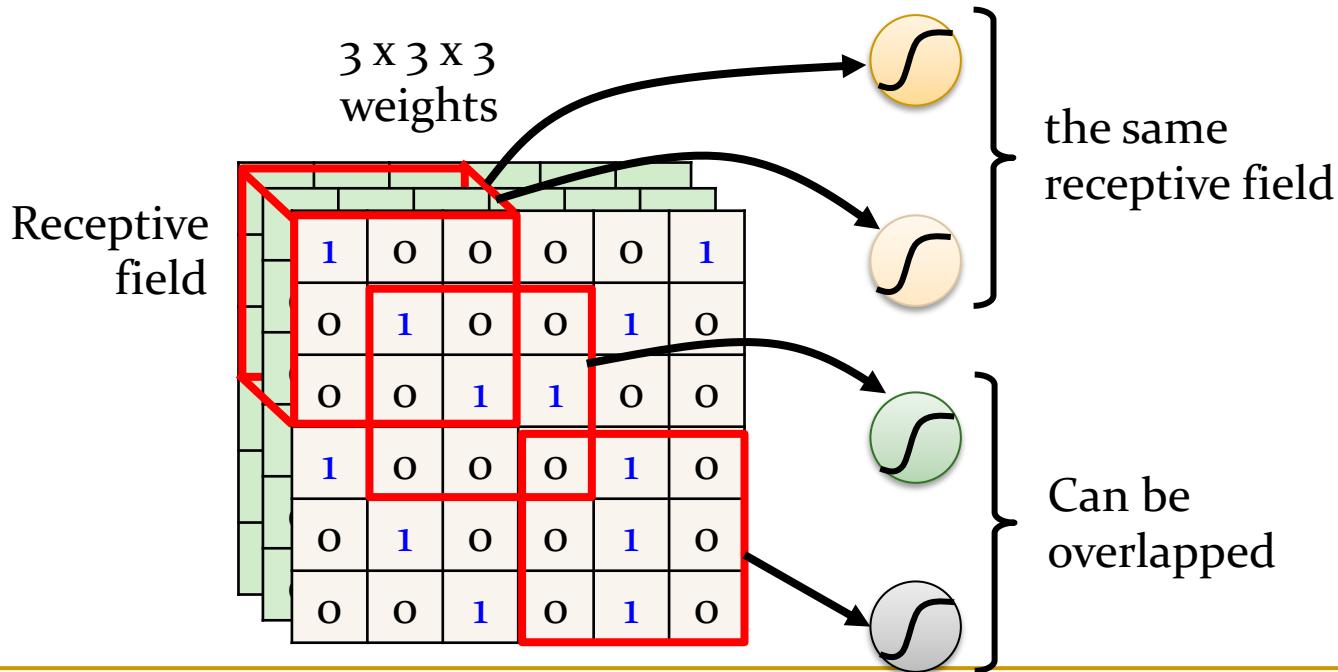
Some patterns are much smaller than the whole image.

# Simplification 1



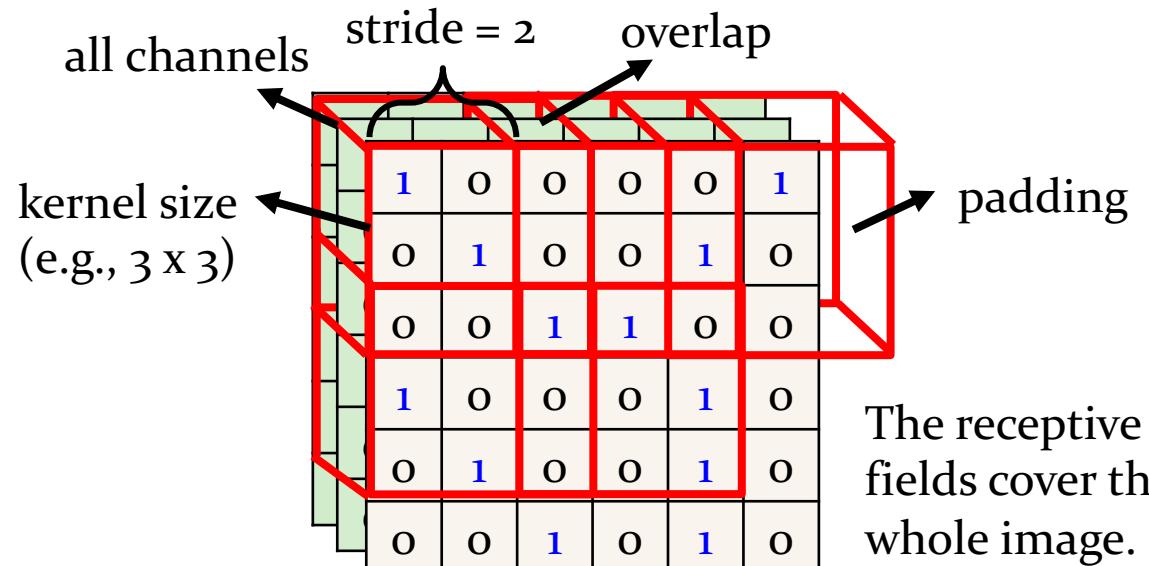
# Simplification 1

- Can different neurons have different sizes of receptive field?
- Cover only some channels?
- Not square receptive field?



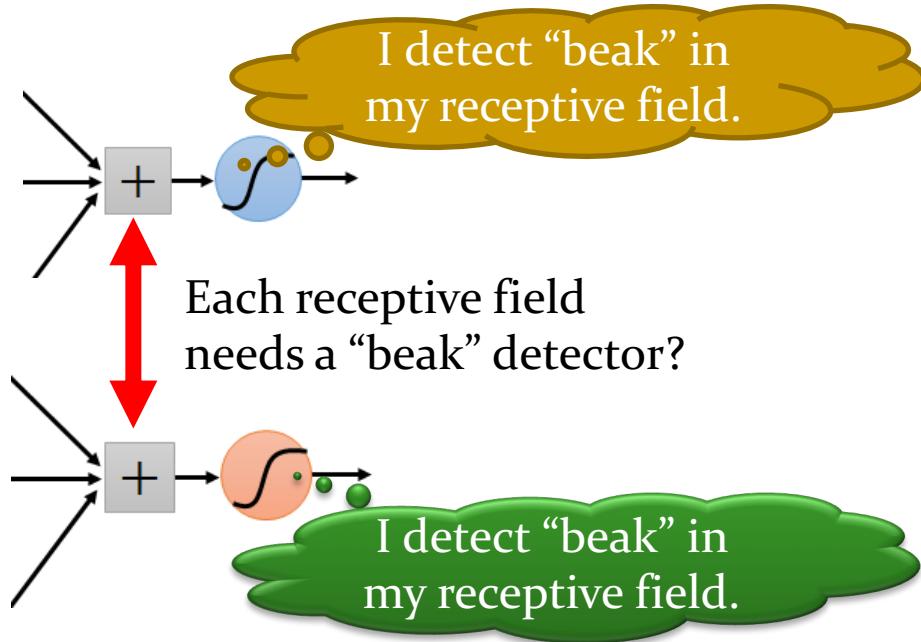
# Simplification 1 – Typical Setting

Each receptive field has a set of neurons (e.g., 64 neurons).

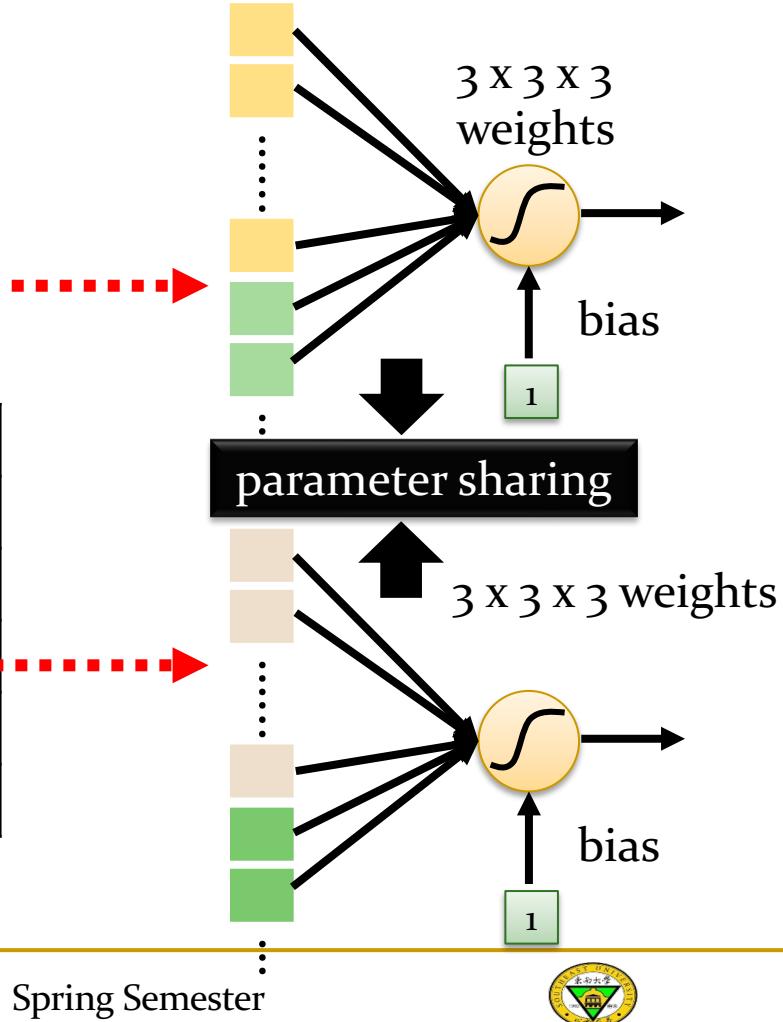
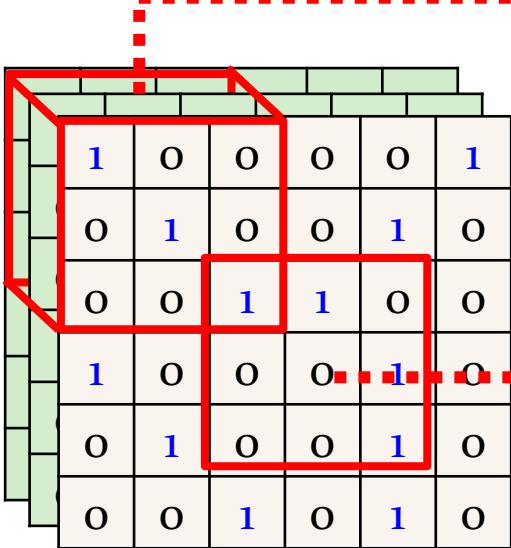


# Observation 2

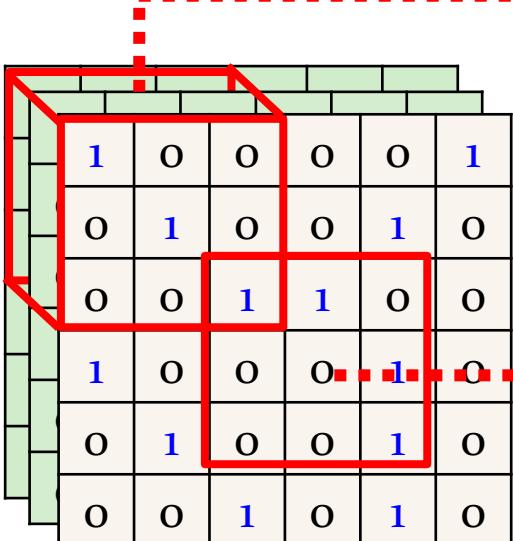
- The same patterns appear in different regions.



# Simplification 2

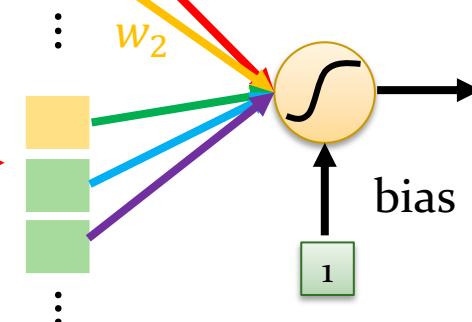


# Simplification 2

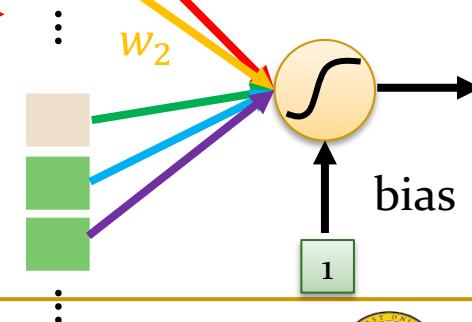


Two neurons with the same receptive field would not share parameters.

$$x_1 \quad \sigma(w_1 x_1 + w_2 x_2 + \dots)$$
$$x_2$$
$$\vdots$$
$$w_1$$
$$w_2$$

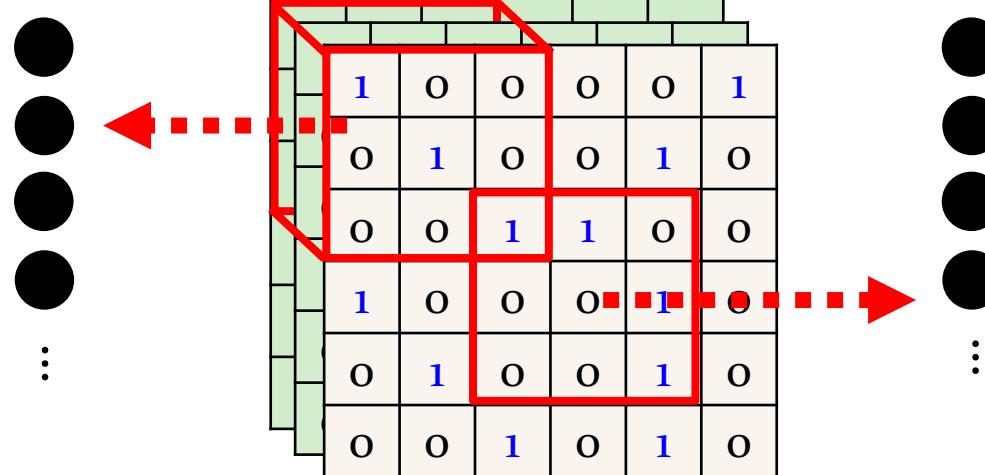


$$x'_1 \quad \sigma(w_1 x'_1 + w_2 x'_2 + \dots)$$
$$x'_2$$
$$\vdots$$
$$w_1$$
$$w_2$$



# Simplification 2 – Typical Setting

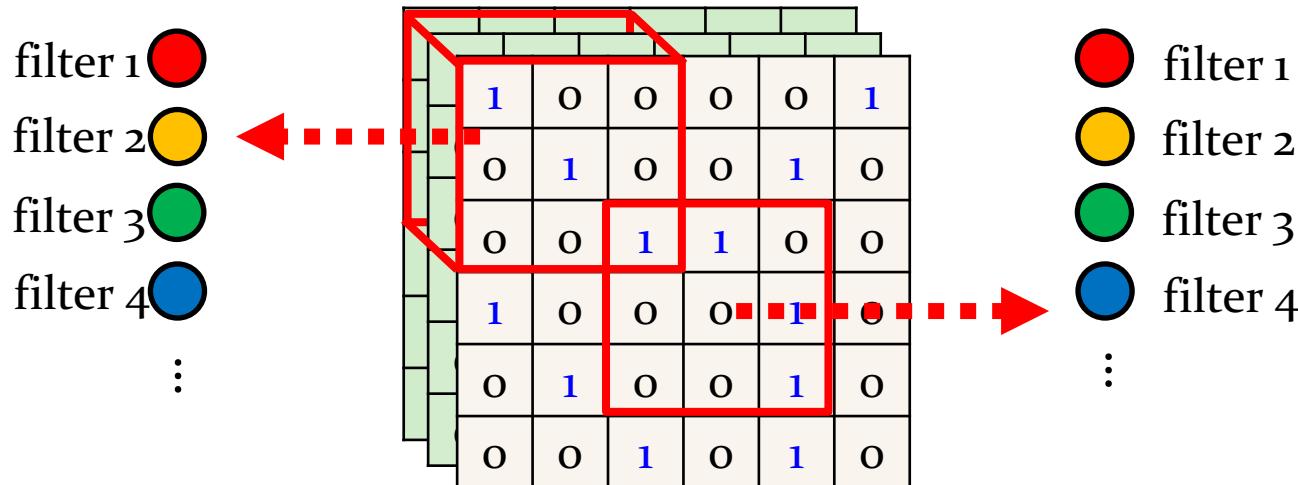
Each receptive field has a set of neurons (e.g., 64 neurons).



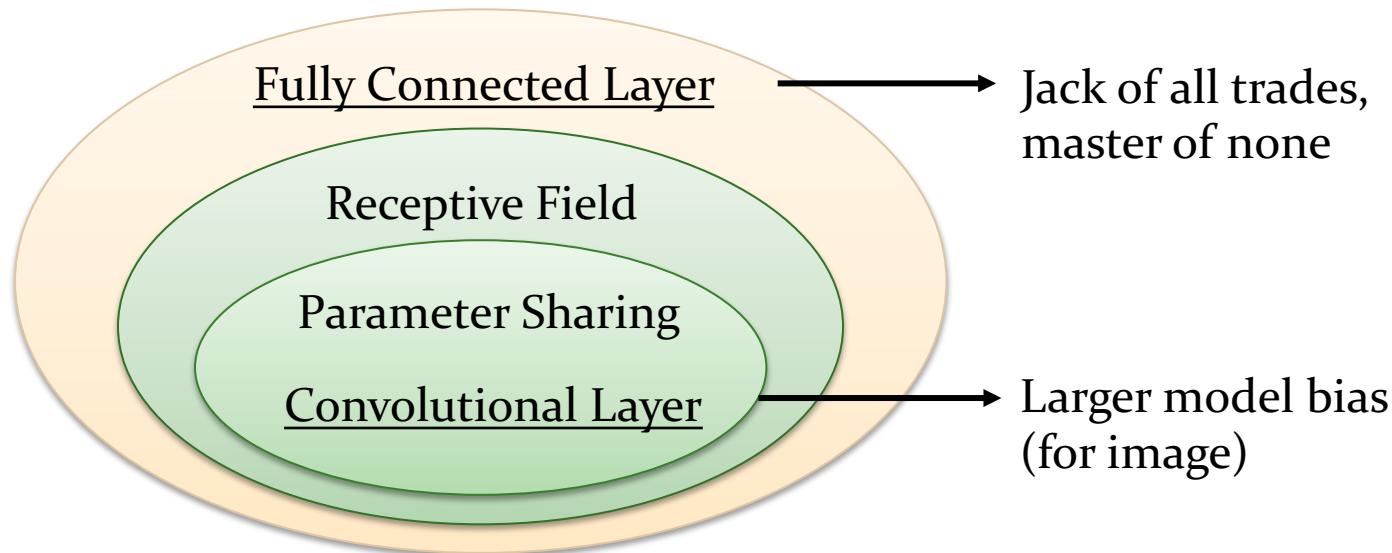
# Simplification 2 – Typical Setting

Each receptive field has a set of neurons (e.g., 64 neurons).

Each receptive field has the neurons with the same set of parameters.

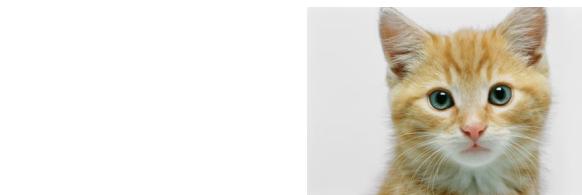


# Benefit of Convolutional Layer



- Some patterns are much smaller than the whole image.
- The same patterns appear in different regions.

# Convolutional Layer



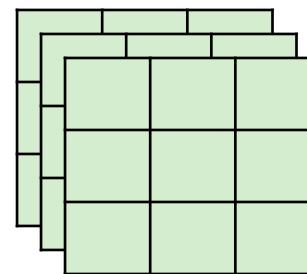
channel = 3 (colorful)  
channel = 1 (black and white)

Another story based on filter ☺

Each filter detects a small pattern ( $3 \times 3 \times$  channel).

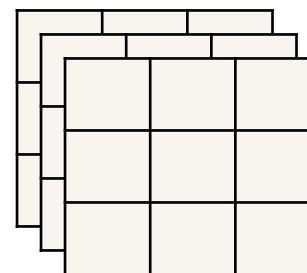
Filter 1

$3 \times 3 \times$  channel  
tensor



Filter 2

$3 \times 3 \times$  channel  
tensor



# Convolutional Layer

Consider channel = 1  
(black and white image)

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

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

Filter 1

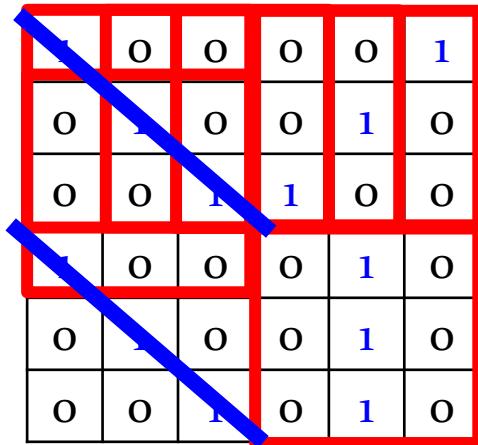
-1	1	-1
-1	1	-1
-1	1	-1

Filter 2

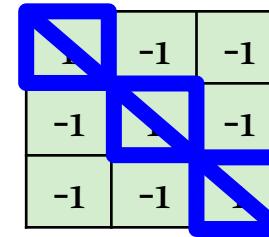
: : (The values in the filters  
are unknown  
parameters.)

# Convolutional Layer

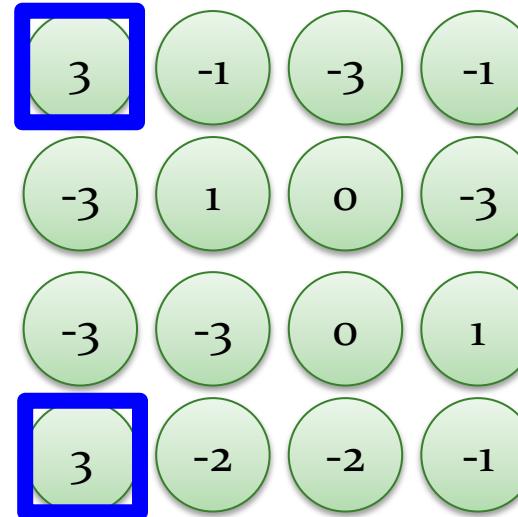
stride=1



6 x 6 image



Filter 1



# Convolutional Layer

stride=1

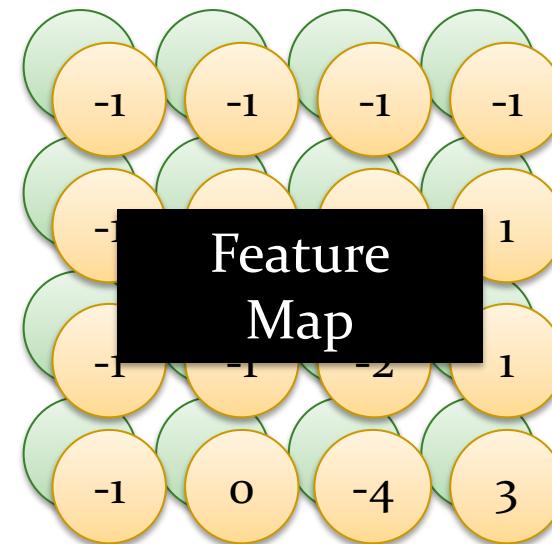
1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

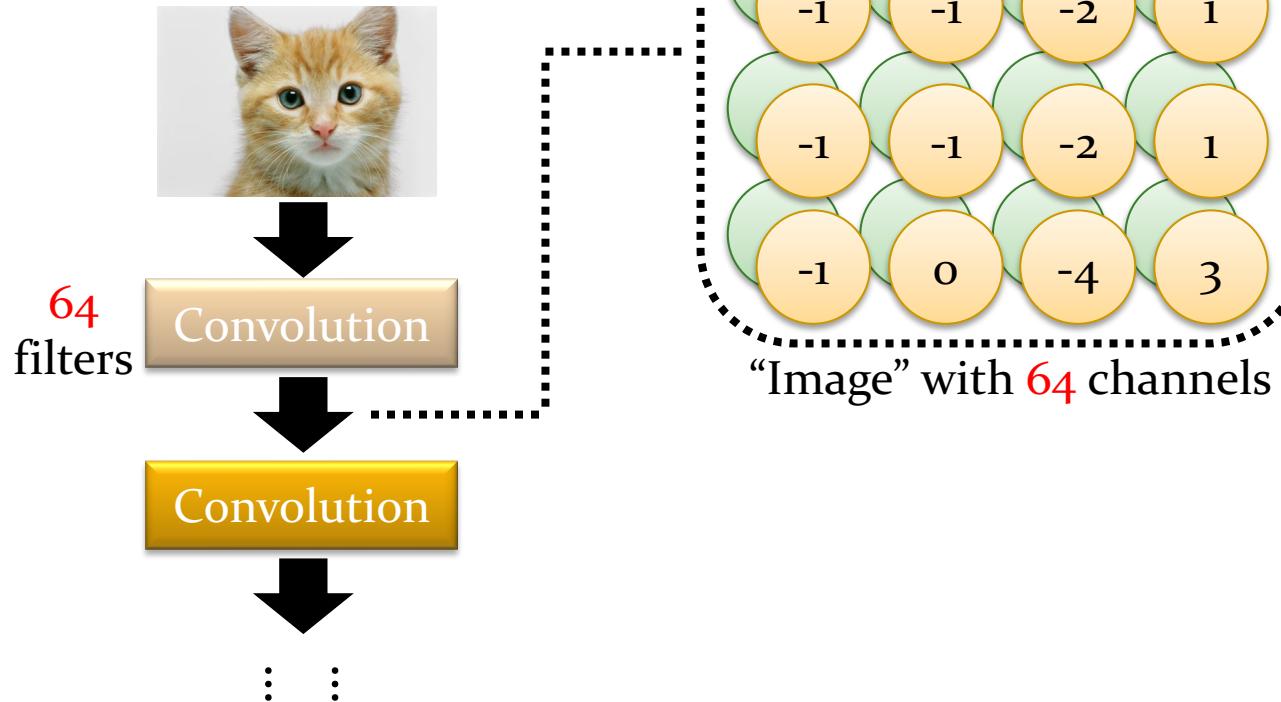
-1	1	-1
-1	1	-1
-1	1	-1

Filter 2

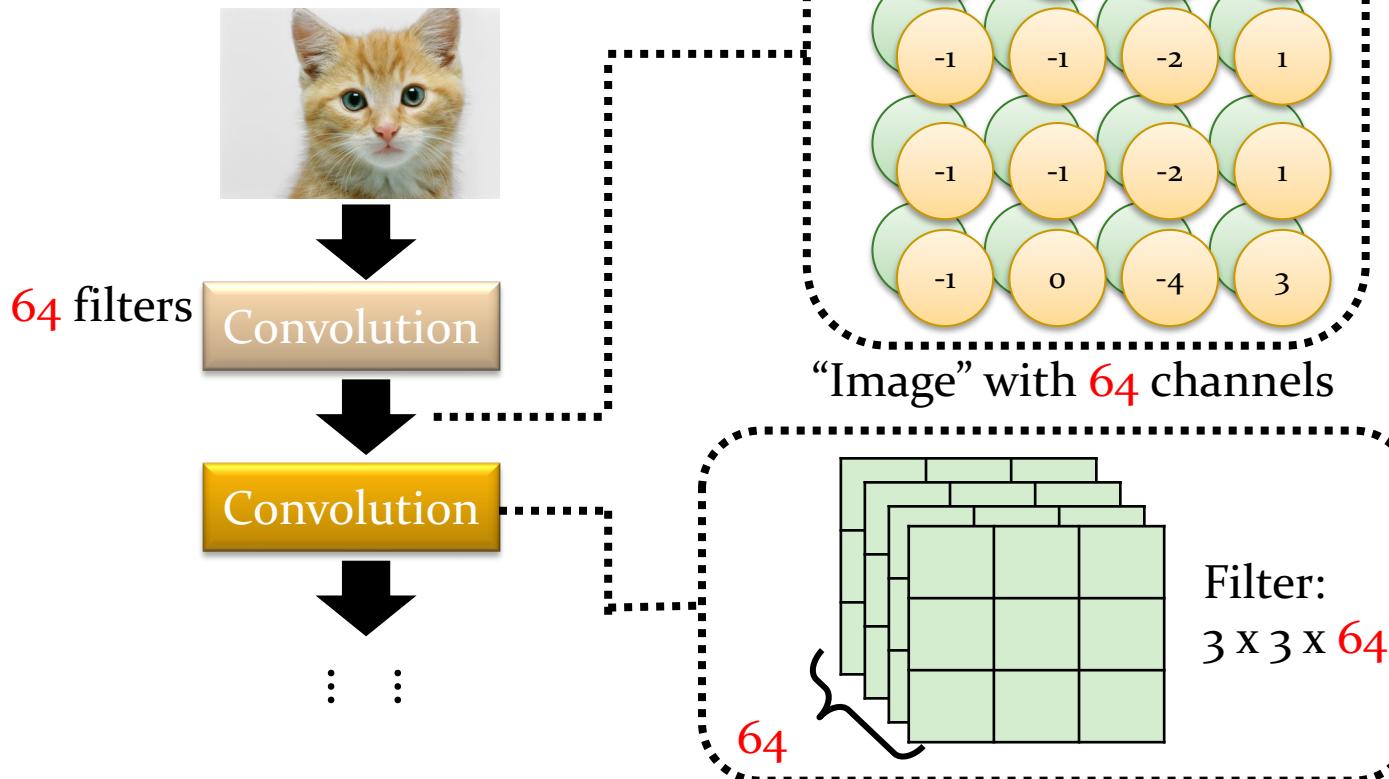
Do the same process  
for every filter



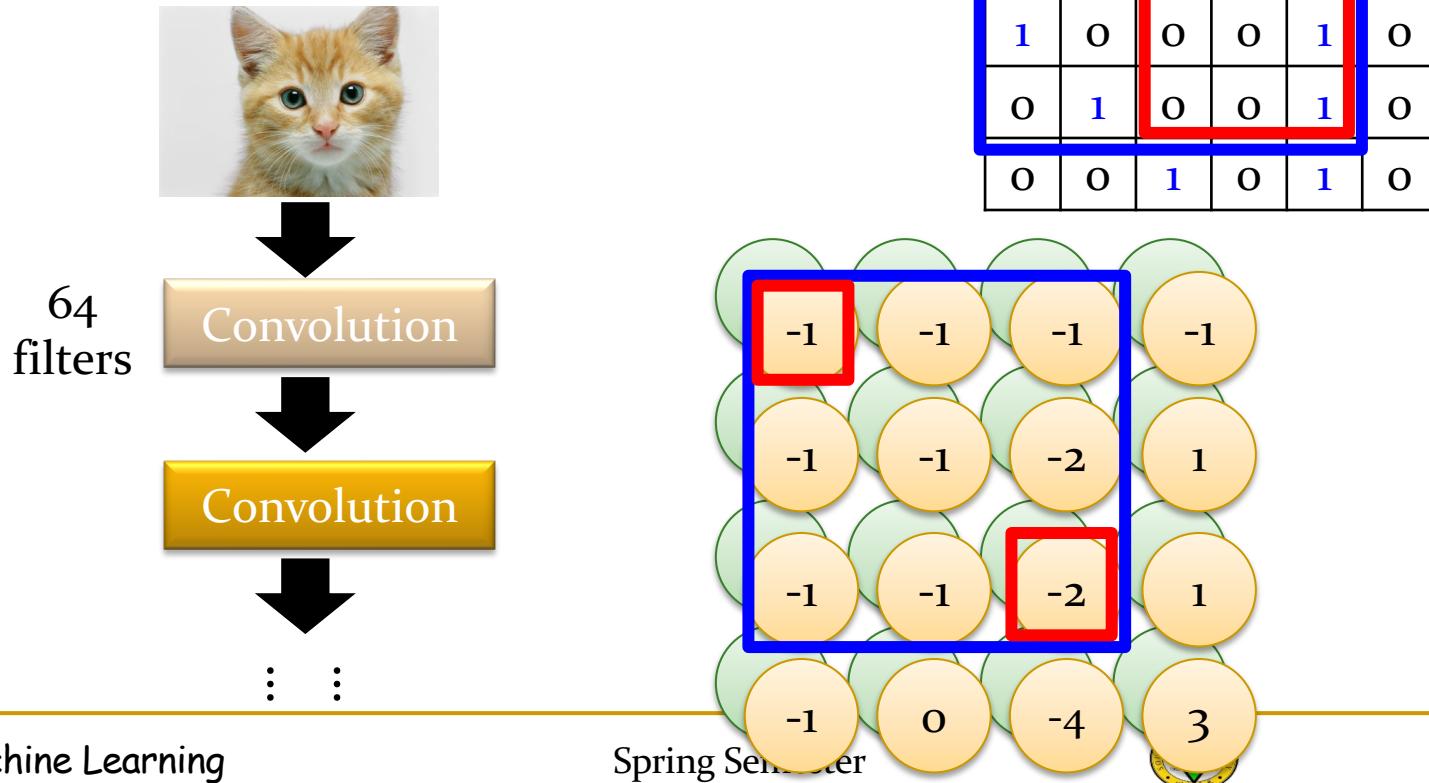
# Convolutional Layer



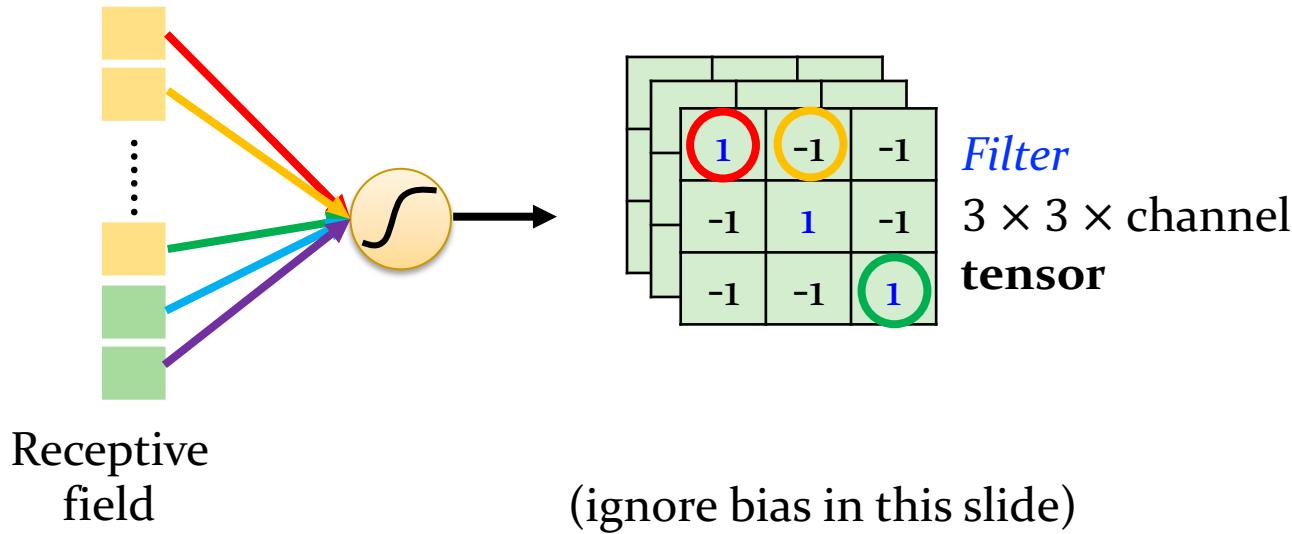
# Multiple Convolutional Layer



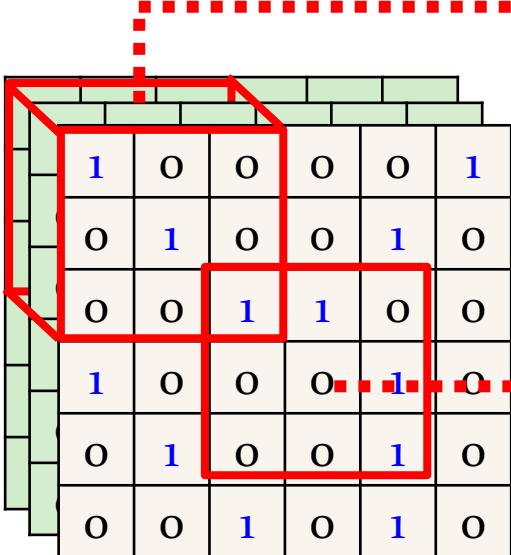
# Multiple Convolutional Layer



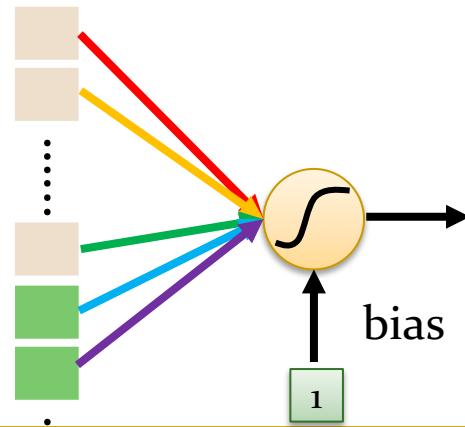
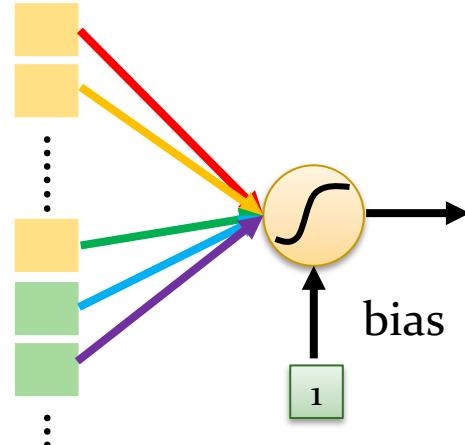
# Comparison of Two Stories



The neurons with different receptive fields share the parameters.



Each filter convolves over the input image.



# Convolutional Layer

## Neuron Version Story

Each neuron only considers a receptive field.

The neurons with different receptive fields share the parameters.

## Filter Version Story

There are a set of filters detecting small patterns.

Each filter convolves over the input image.

They are the same story.



# Observation 3

- Subsampling the pixels will not change the object

bird



subsampling

bird



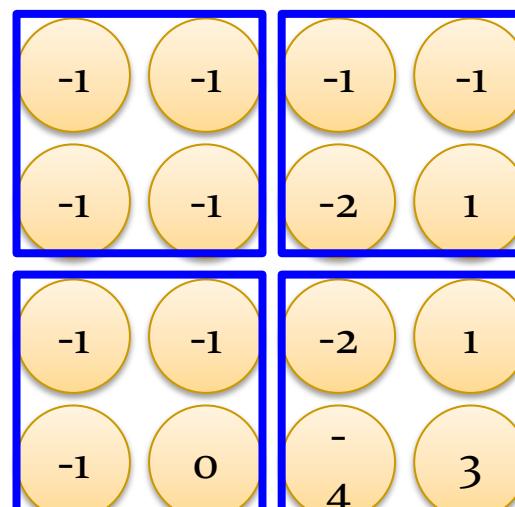
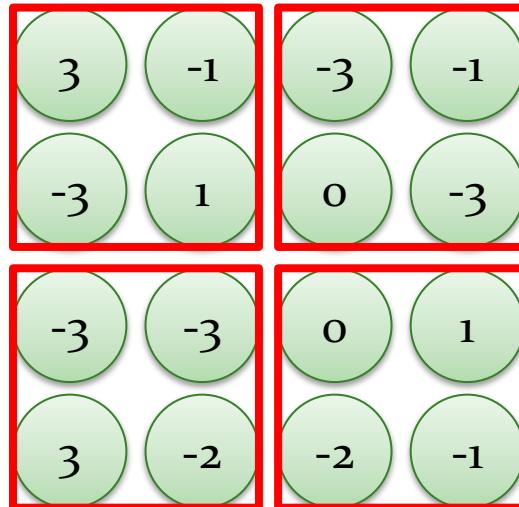
# Pooling – Max Pooling

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

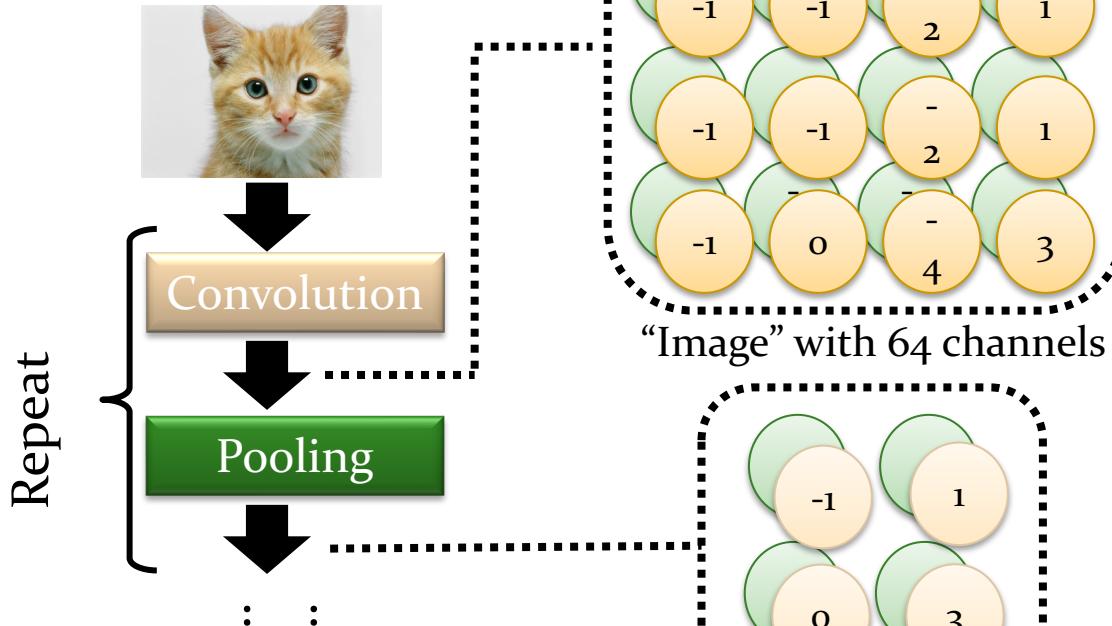
Filter 1

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

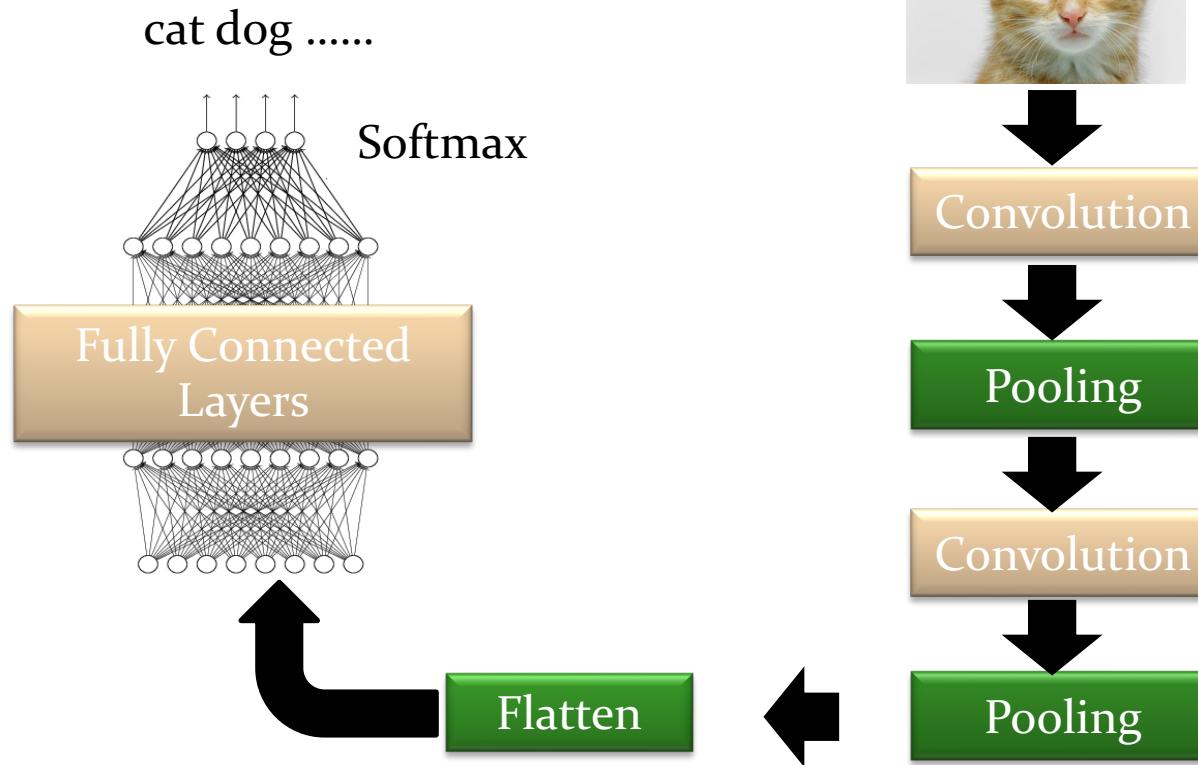
Filter 2



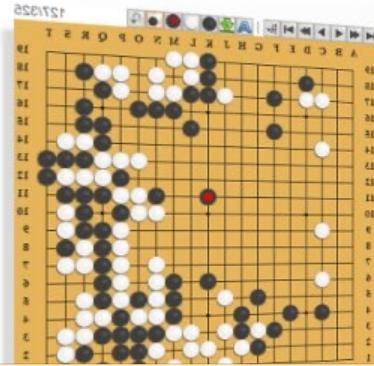
# Convolutional Layers + Pooling



# The whole CNN



# Application: Playing Go



$19 \times 19$  matrix  
(image)

48 channels in  
Alpha Go

Black: 1  
white: -1  
none: 0



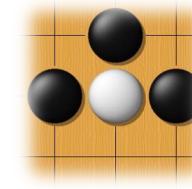
Next move  
 $(19 \times 19$  positions)  
 $19 \times 19$  classes

Fully-connected  
network can be used  
But CNN performs much  
better.

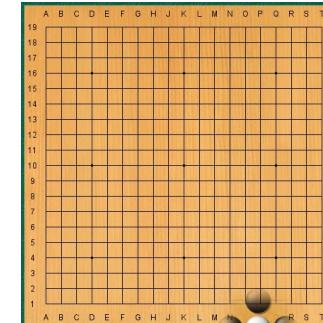
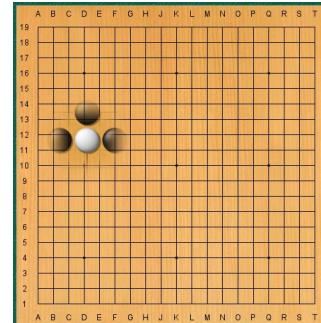
# Why CNN for Go playing?

- Some patterns are much smaller than the whole image

Alpha Go uses  $5 \times 5$  for first layer



- The same patterns appear in different regions.



# Why CNN for Go playing?

- Subsampling the pixels will not change the object



Pooling

How to explain this???

**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  $23 \times 23$  image, then convolves  $k$  filters of kernel size  $5 \times 5$  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  $21 \times 21$  image, then convolves  $k$  filters of kernel size  $3 \times 3$  with stride 1, again followed by a rectifier nonlinearity. The final layer convolves 1 filter of kernel size  $1 \times 1$  with stride 1, with a different bias for each position, and applies a softmax function. The match version of AlphaGo used  $k = 192$  filters; Fig. 2b and Extended Data Table 1 show the architecture. Alpha Go does not use Pooling .....

```
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net = Net()
```

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

for epoch in range(2): # 多批次循环

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # 获取输入
        inputs, labels = data

        # 梯度置0
        optimizer.zero_grad()

        # 正向传播, 反向传播, 优化
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```



# To learn more ...

- CNN is not invariant to scaling and rotation (we need data augmentation ☺).



# Summary

- RNNs allow a lot of flexibility in architecture design
- Vanilla RNNs are simple but don't work very well
- Backward flow of gradients in RNN can explode or vanish.
- Exploding is controlled with gradient clipping. Vanishing is controlled with additive interactions (LSTM)
- Common to use LSTM or GRU: their additive interactions improve gradient flow
- Better/simpler architectures are a hot topic of current research, as well as new paradigms for reasoning over sequences
- Better understanding (both theoretical and empirical) is needed.

