Introduction to Convolutional Neural Networks

Anahita Zarei, Ph.D.

Parallels Between Regular and Convolutional Neural Networks

Both type of networks:

- Have hidden neurons and layers
- Involve summation, multiplication, and non-linear operators
- Have learnable weights and biases
- Minimize a cost function

Differences Between Regular and Convolutional Neural Networks

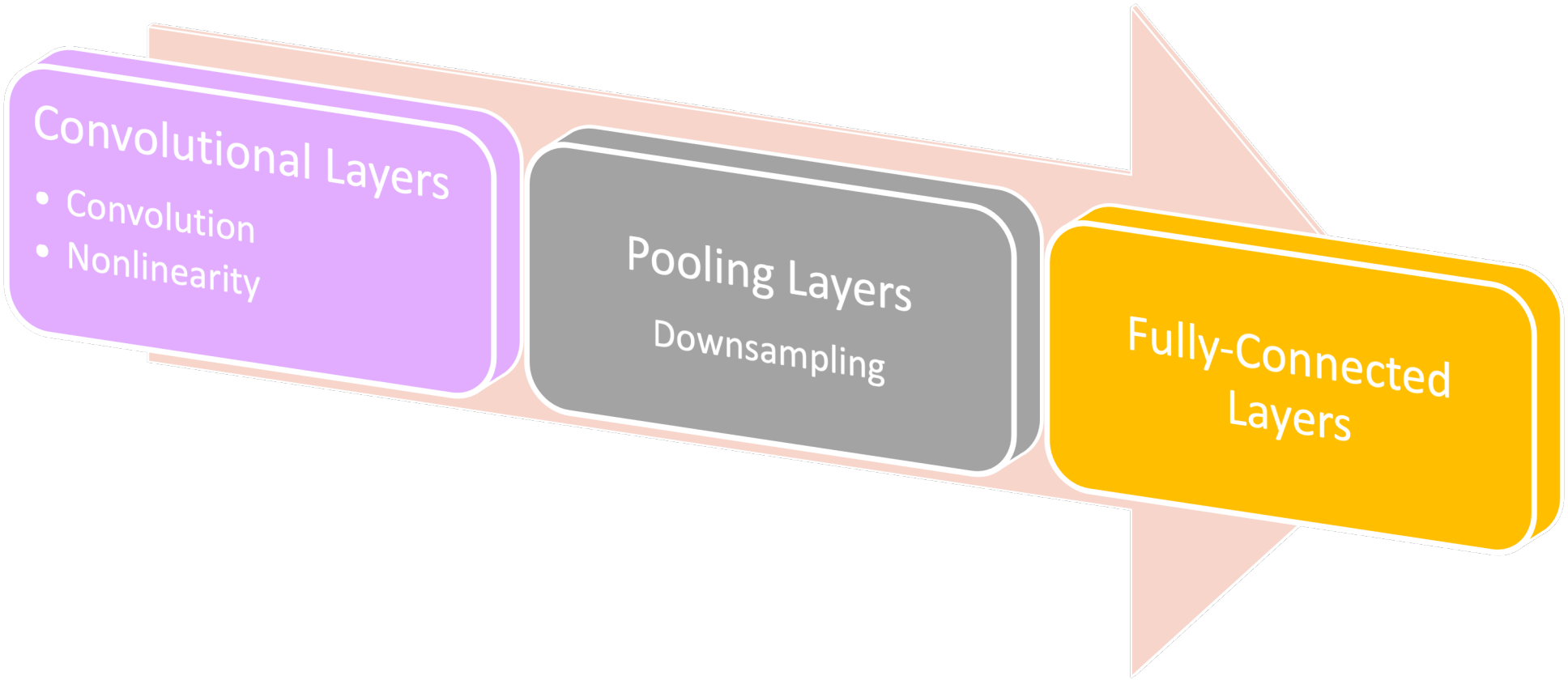
Regular Networks

- Are fully connected
- Don't consider the underlying architecture of input

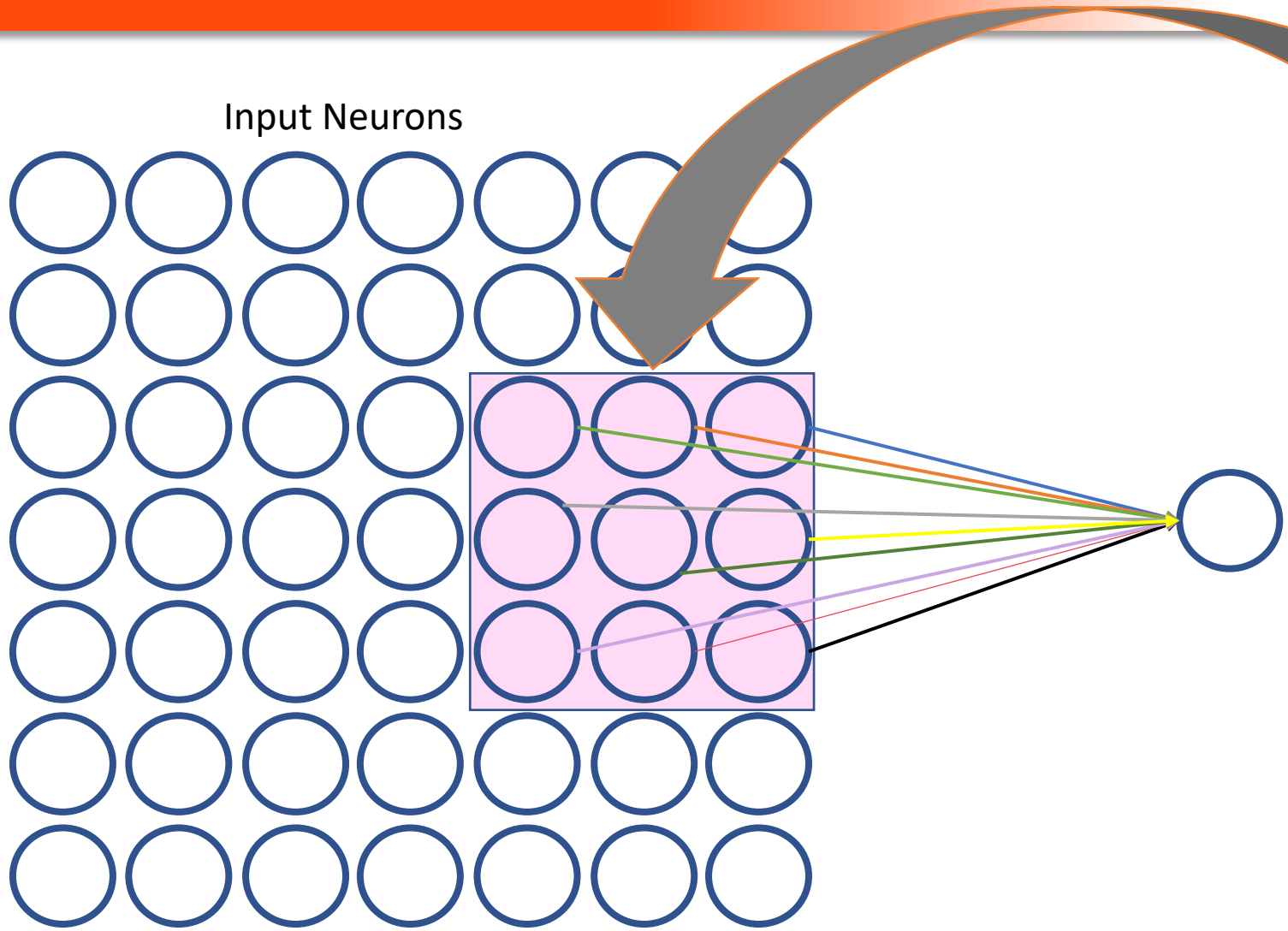
Convolutional Networks

- Are NOT fully connected (except in the final layer(s))
- Take advantage of the spatial structure of input

Layer Types in Convolutional Neural Network



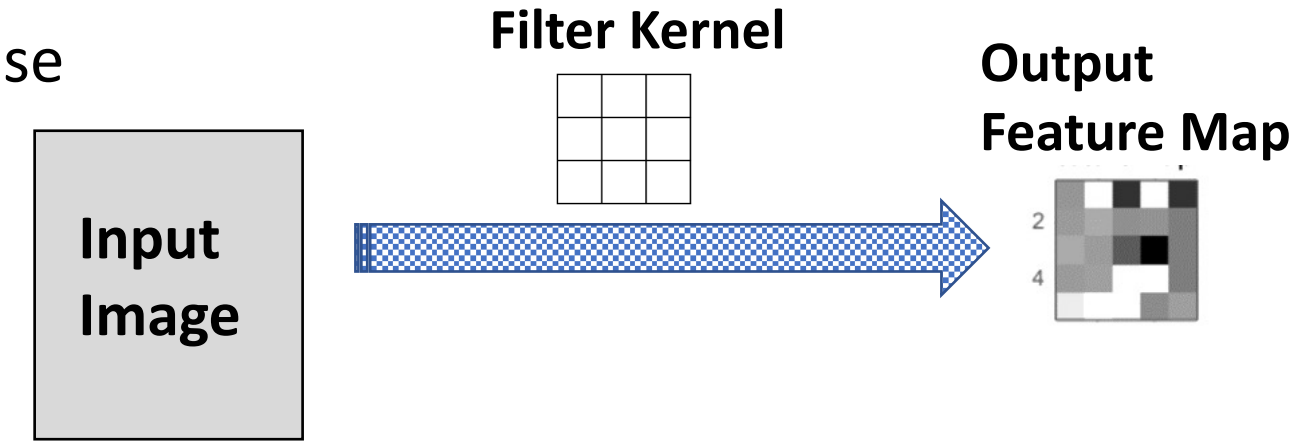
Convolutional Layer

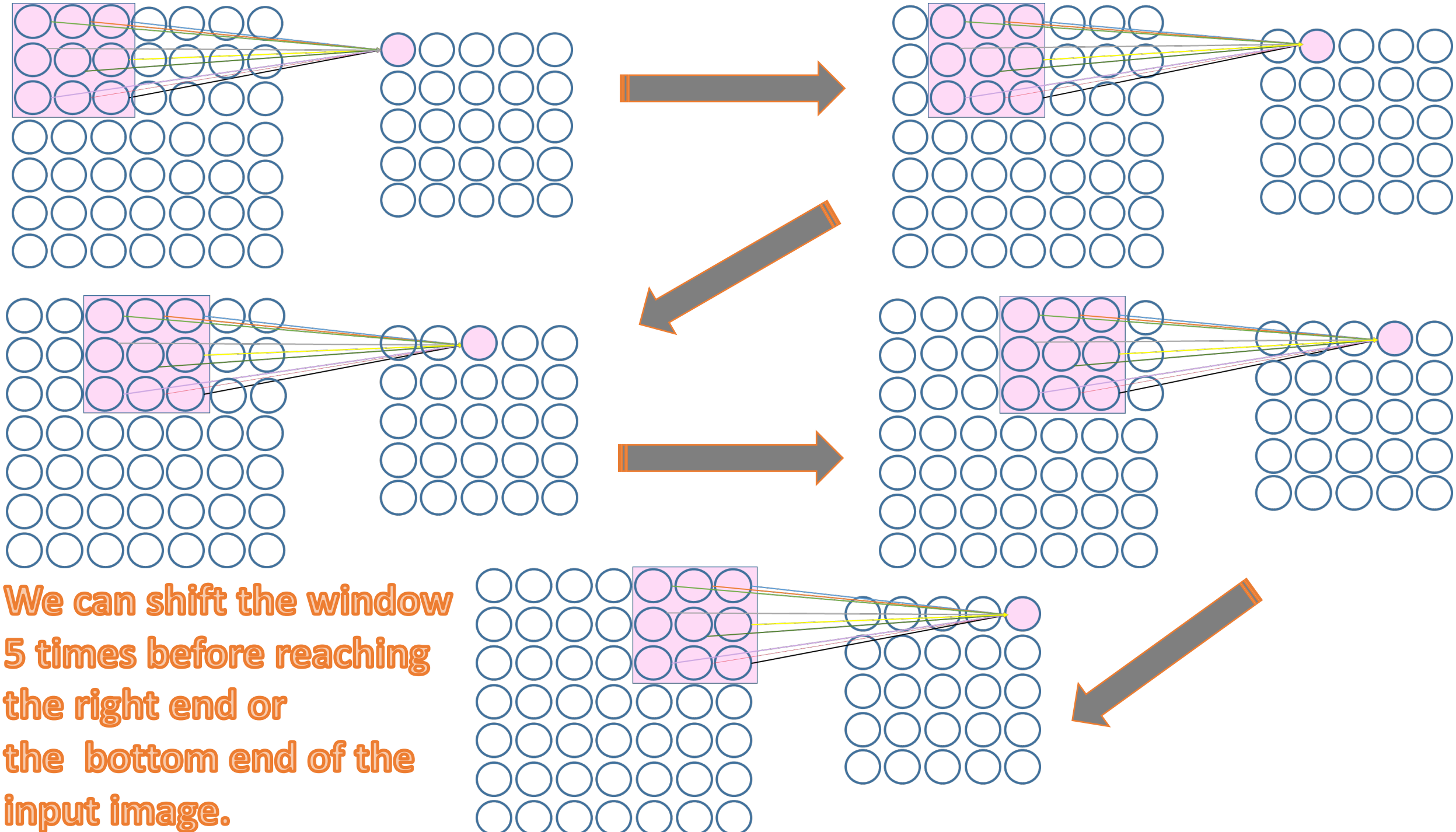


This 3x3 area corresponding to 9 input pixels, is called the receptive field

Convolutional Layer

- The convolution operation extracts patches from its input feature map (receptive field) and applies the same transformation to all of these patches, producing an *output feature map*.
- This output feature map has a width and a height.
- It is the result of filtering (convolving) the input image with the kernel filter.



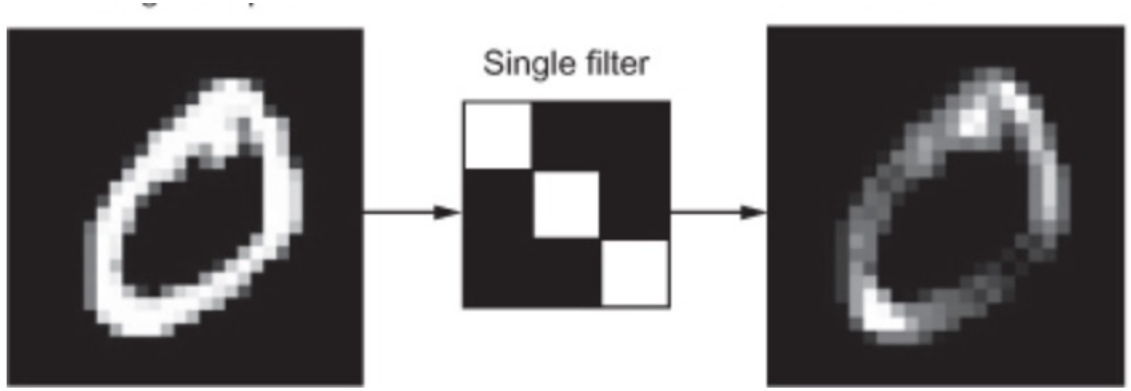


We can shift the window
5 times before reaching
the right end or
the bottom end of the
input image.

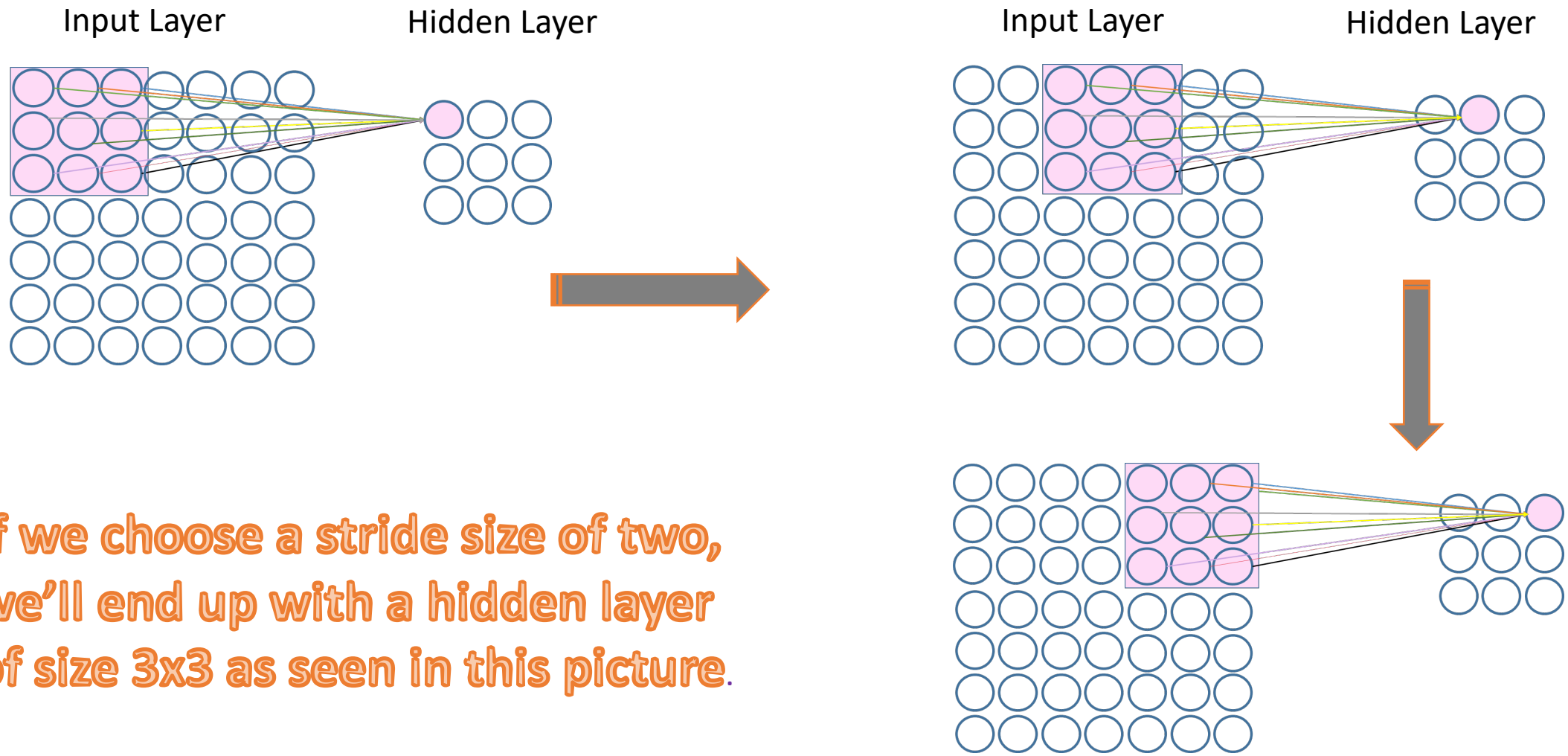
Convolutional Layer

- Convolutions are defined by two key parameters:
- ***Size of the patches extracted from the inputs (Filter size):*** These are typically 3 × 3 or 5 × 5.
- ***Depth of the output feature map:*** The number of filters computed by the convolution.

- The feature map quantifies the presence of the filters pattern at different locations.



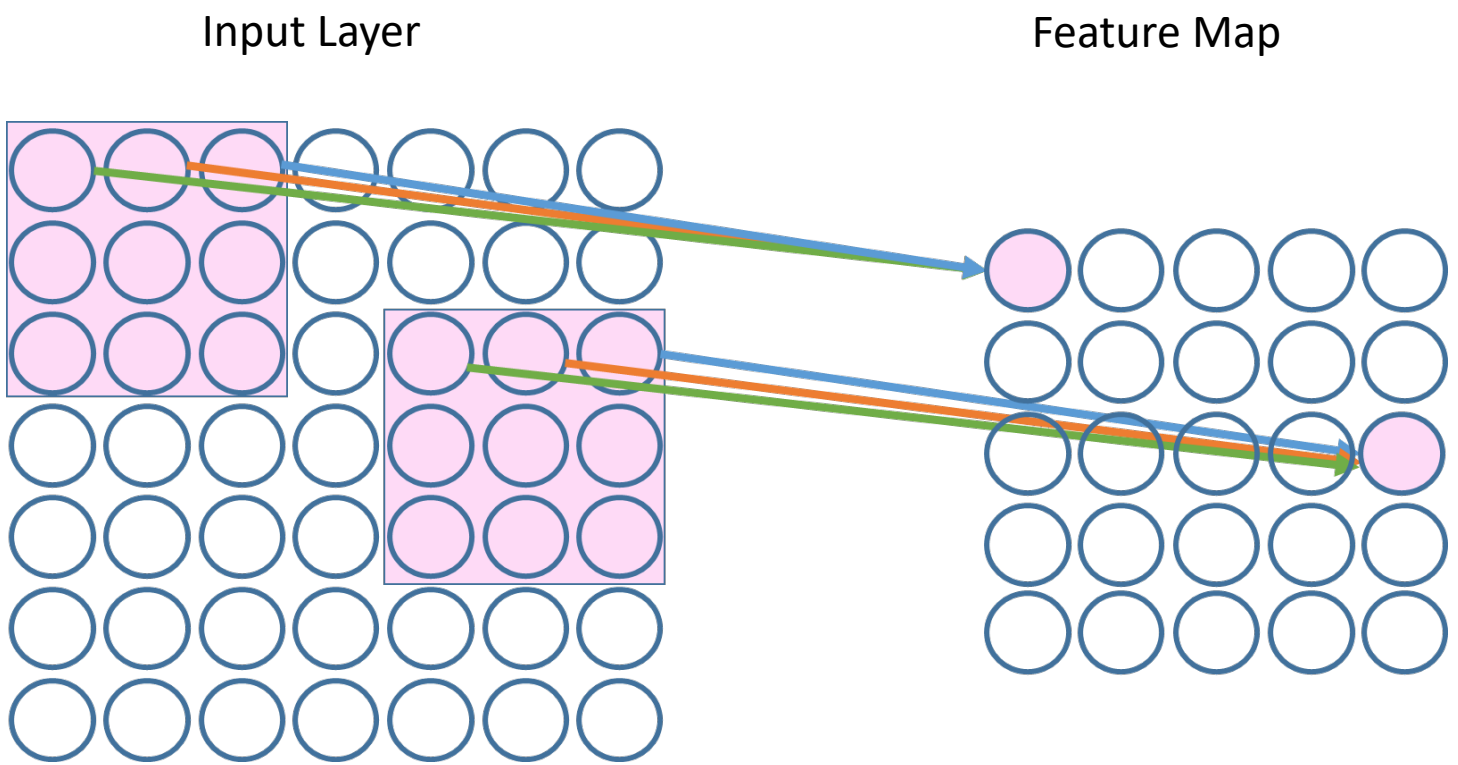
Stride Size



If we choose a stride size of two, we'll end up with a hidden layer of size 3x3 as seen in this picture.

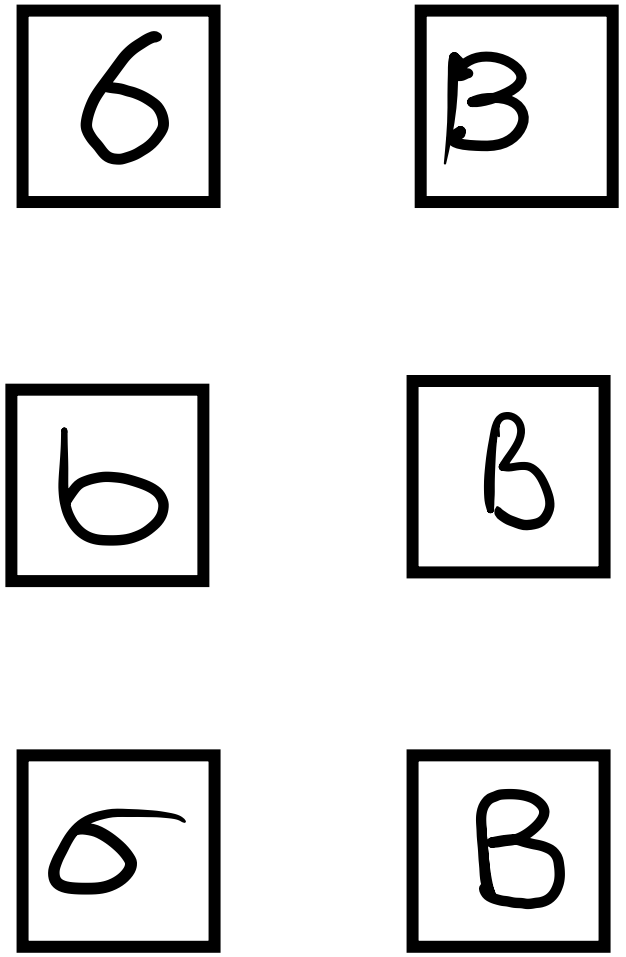
Shared Weights

Every node in this feature map has the same set of incoming weights.
This allows to detect the same feature in different positions.



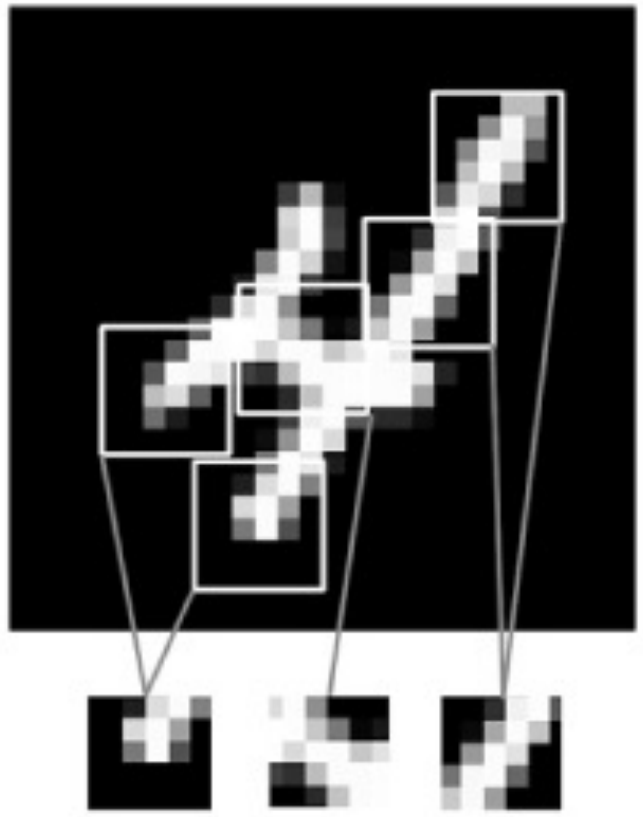
Translation Invariance

Translation Invariance:
The ability to detect a feature regardless of where it appears.



Translation Invariance

- After learning a certain pattern in one location of a picture, a convnet can recognize it anywhere else.
- A densely connected network would have to learn the pattern all over again if it appeared at a new location.
- This property makes convnets efficient when processing images . Hence they need fewer training samples to learn representations that have generalization power

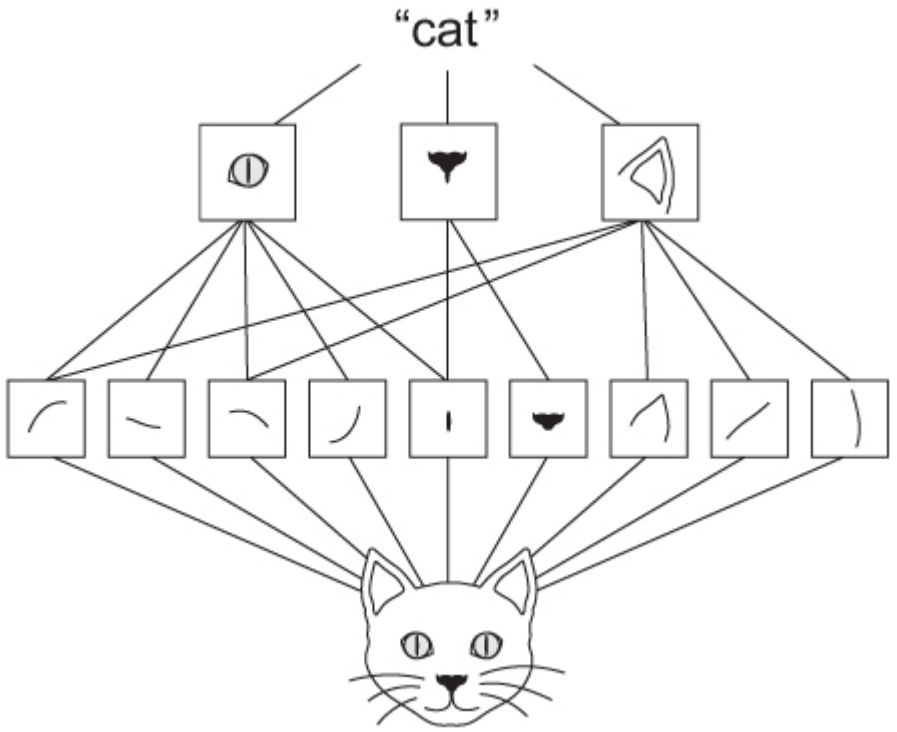


From: Deep Learning with Python by Chollet

Spatial Hierarchies

A first convolution layer will learn small local patterns such as edges, a second convolution layer will learn larger patterns made of the features of the first layers, and so on. This allows convnets to efficiently learn increasingly complex and abstract visual concepts (because the visual world is fundamentally spatially hierarchical).

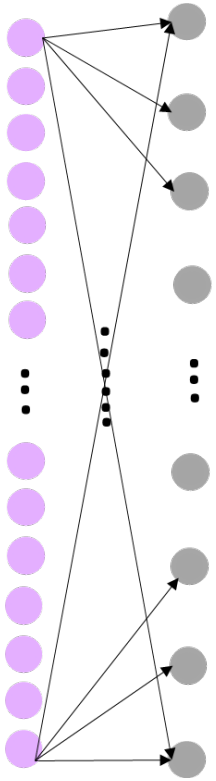
The visual world forms a spatial hierarchy of visual modules: In this figure hyperlocal edges combine into local objects such as eyes or ears, which combine into high-level concepts such as “cat.”



From: Deep Learning with Python by Chollet

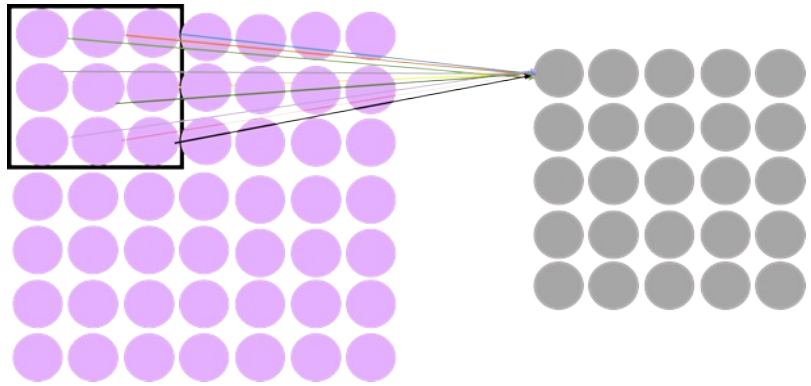
Reduced Number of Parameters

Regular Neural Network



$49{\times}25\ (weights) + 25(bias) = 1250$

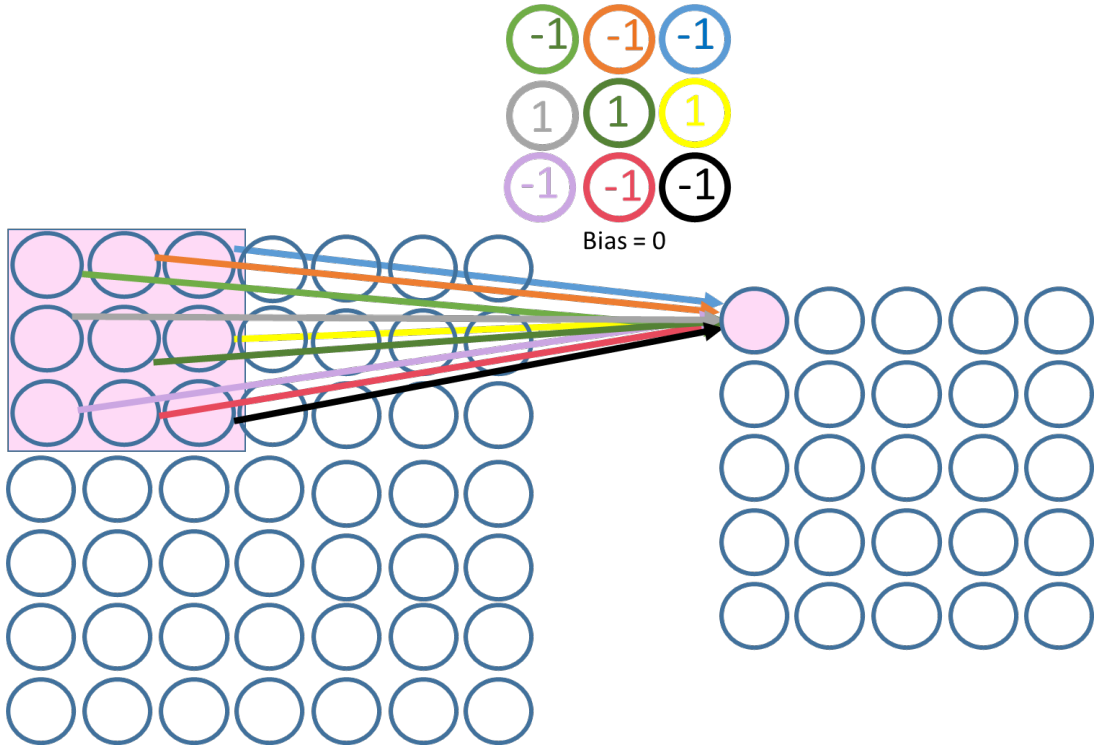
Convolutional Neural Network



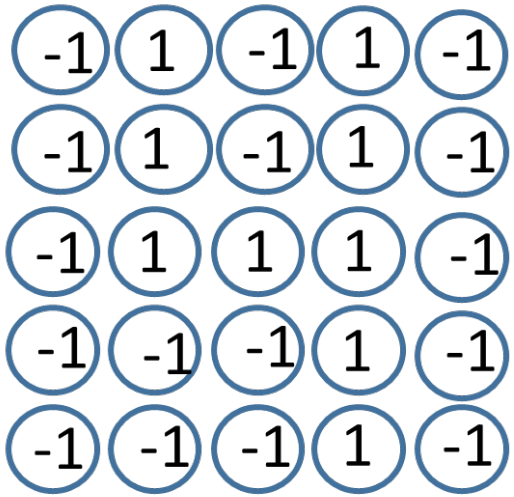
$9\ (shared\ weights) + 1(bias) = 10$

Convolution (Filter)

- A convolution works by sliding filter windows of size 3 × 3 or 5 × 5 over the input feature map, stopping at every possible location.
- Each patch is then transformed (via a dot product with the same learned weight matrix, called the convolution kernel).
- All of these values are then spatially reassembled into a output map of shape (height, width, output_depth = # of filters).
- Every spatial location in the output feature map corresponds to the same location in the input feature map (for example, the lower-right corner of the output contains information about the lower-right corner of the input).

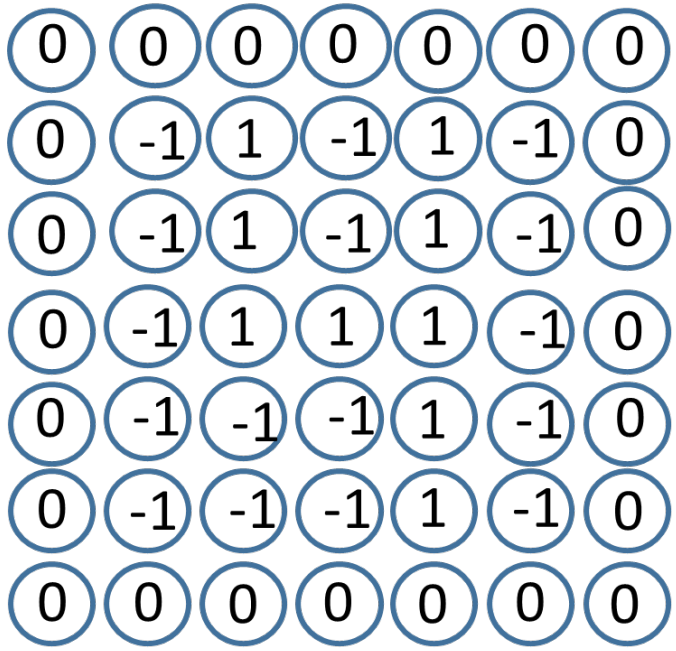


Zero-padding Input Image



Original Image

Add Zeros to Borders



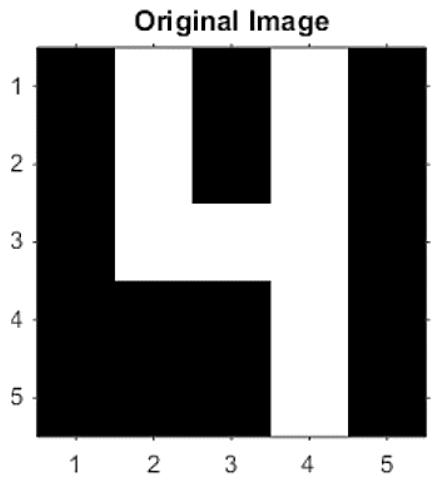
Zero-Padded Image

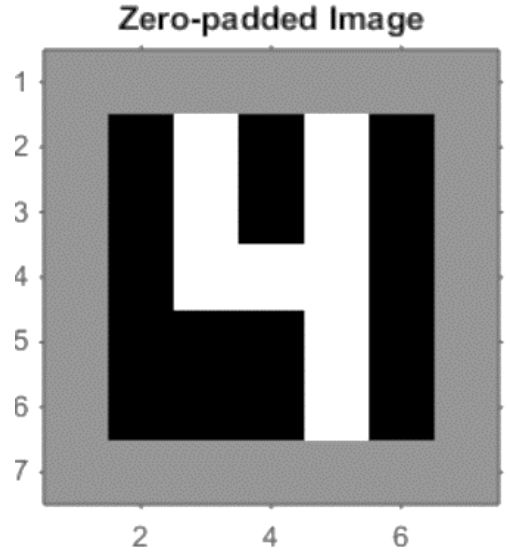
Visualizing the Filter

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

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

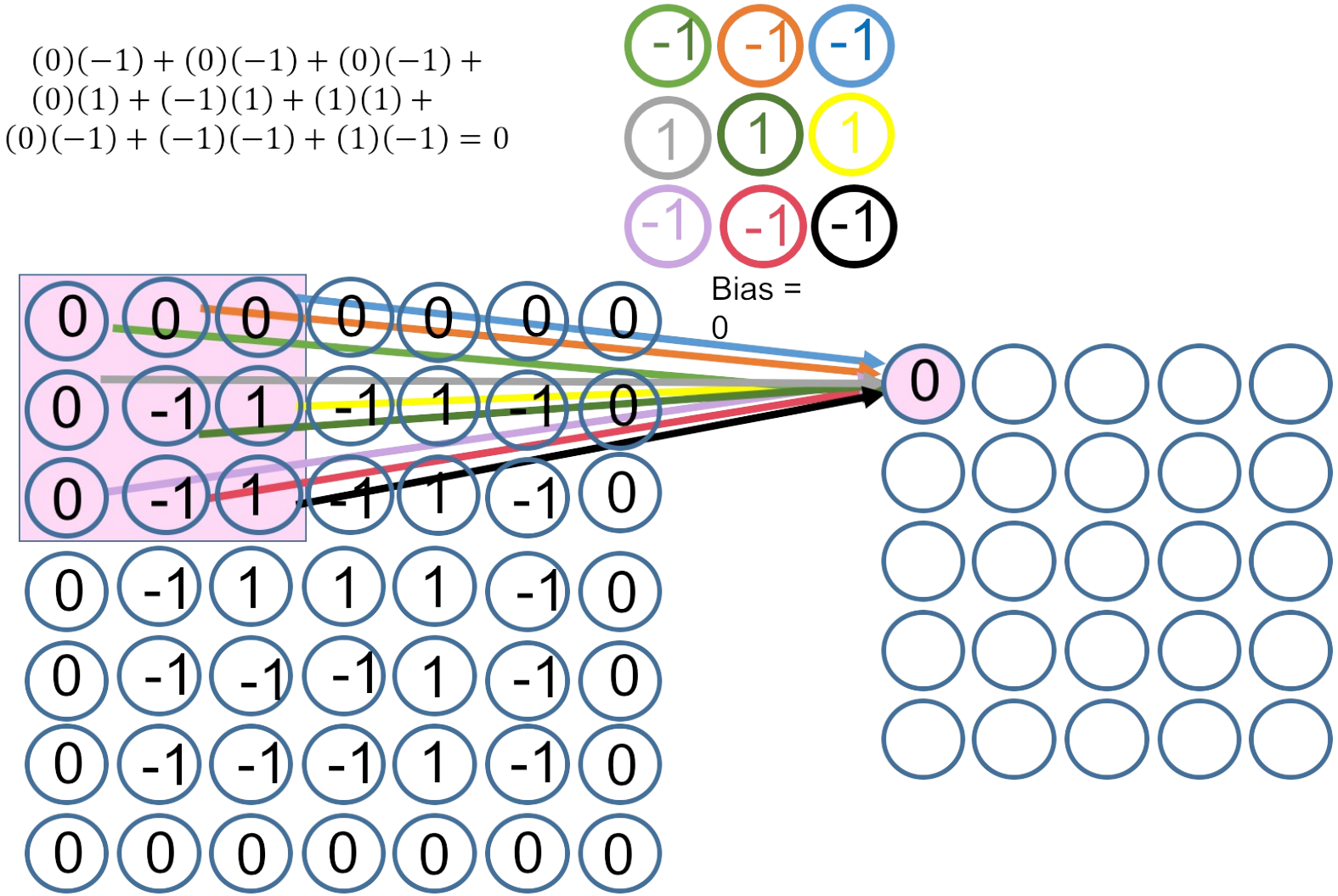
-1	-1	-1
1	1	1
-1	-1	-1





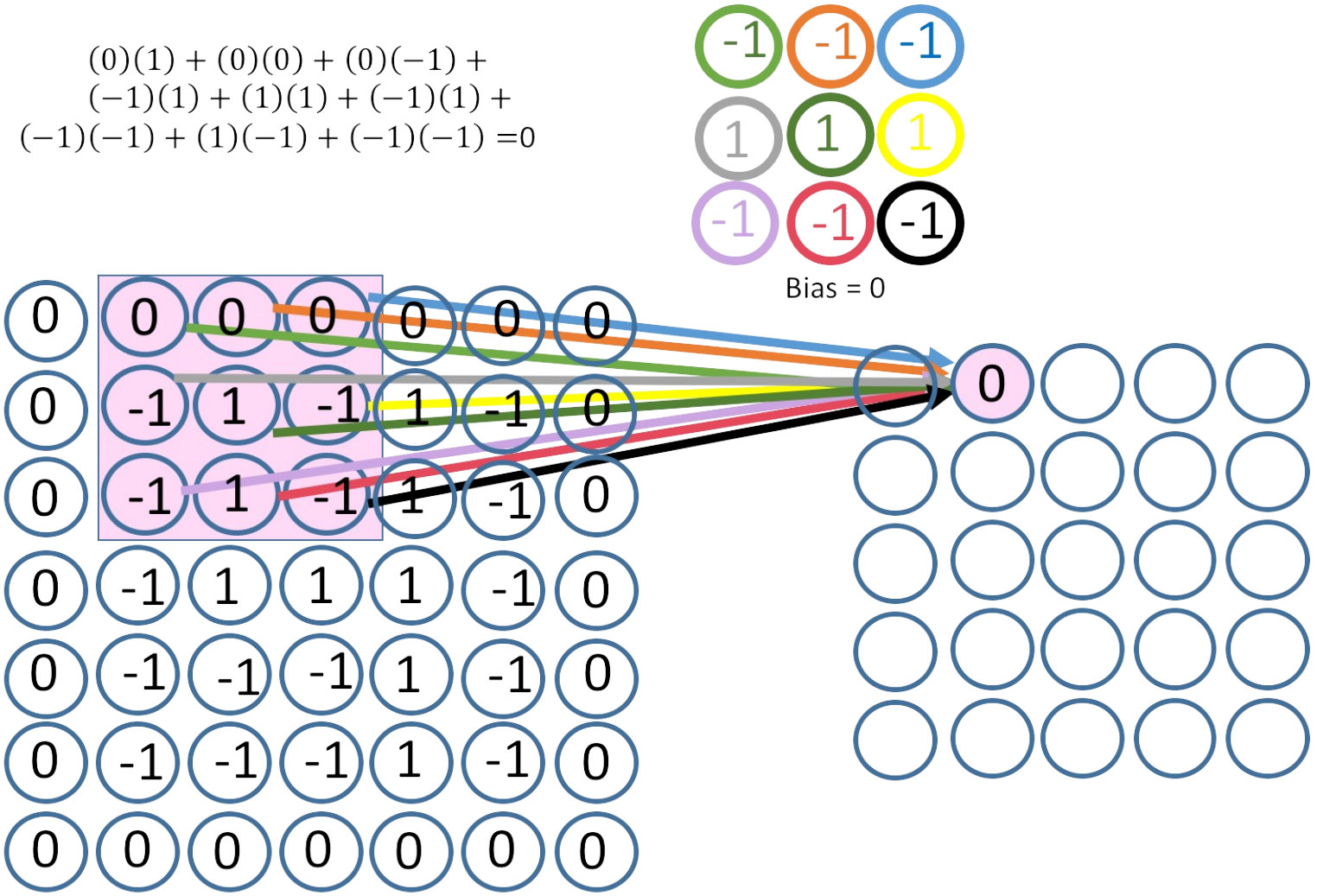
Convolution

$$(0)(-1)+(0)(-1)+(0)(-1)+$$
$$(0)(1)+(-1)(1)+(1)(1)+$$
$$(0)(-1)+(-1)(-1)+(1)(-1)=0$$

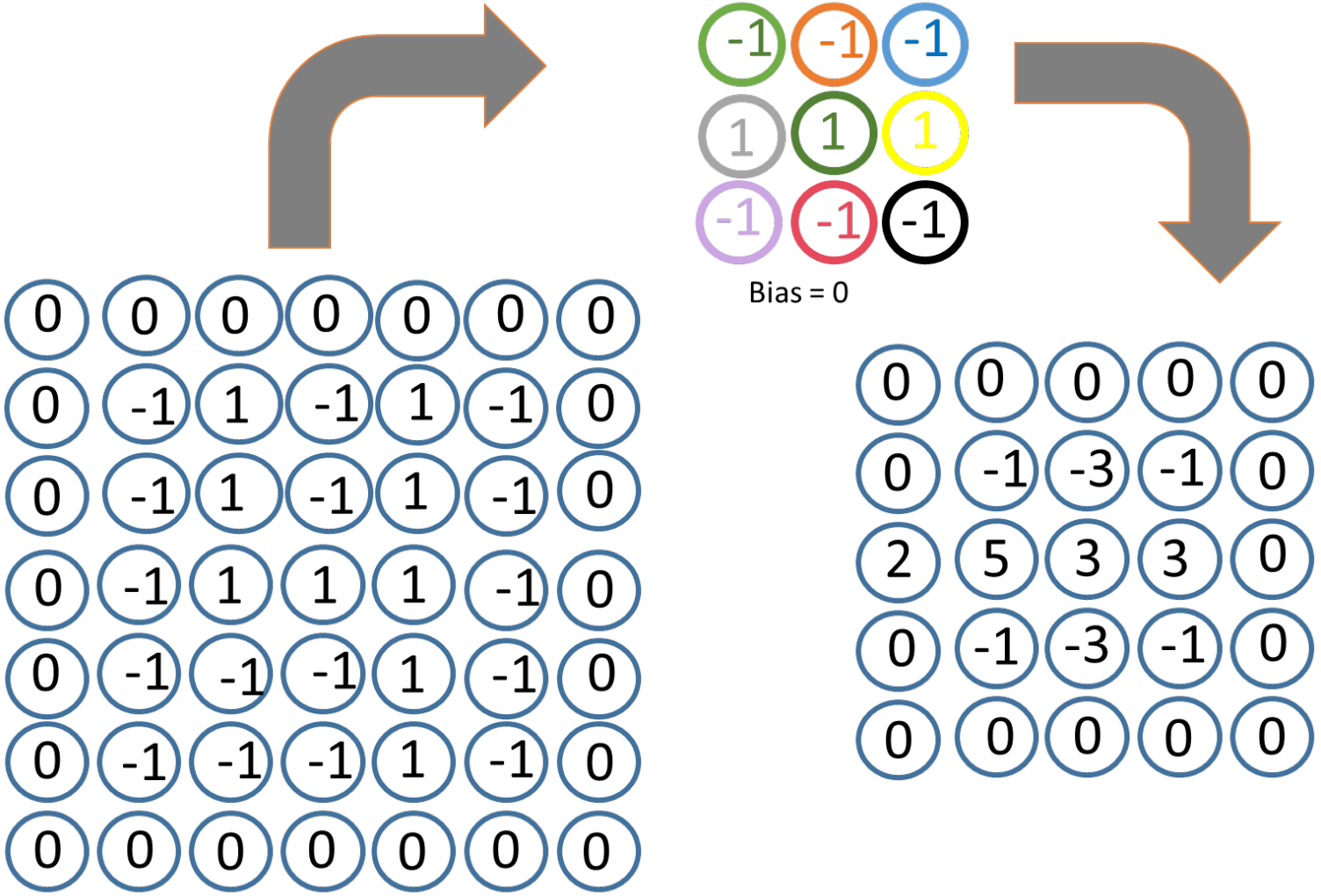


Convolution

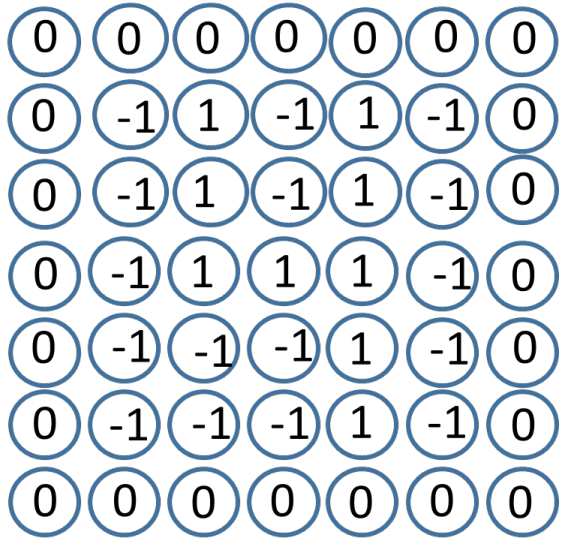
$$(0)(1) + (0)(0) + (0)(-1) + (-1)(1) + (1)(1) + (-1)(1) + (-1)(-1) + (1)(-1) + (-1)(-1) = 0$$

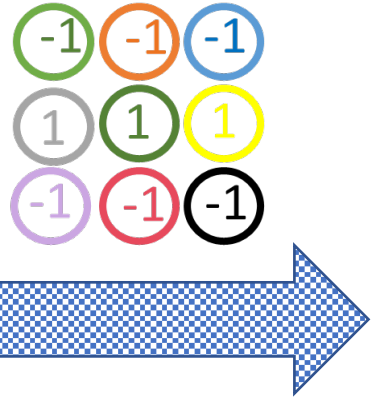


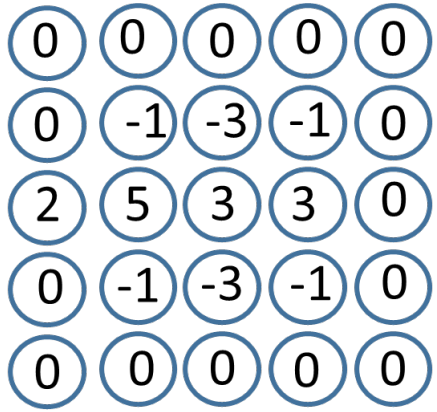
Convolution

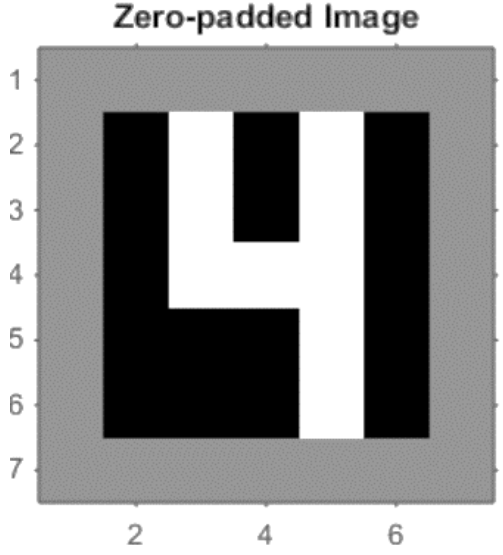


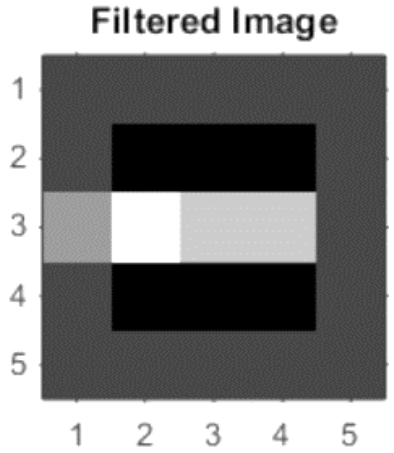
Visualizing the Filter Output



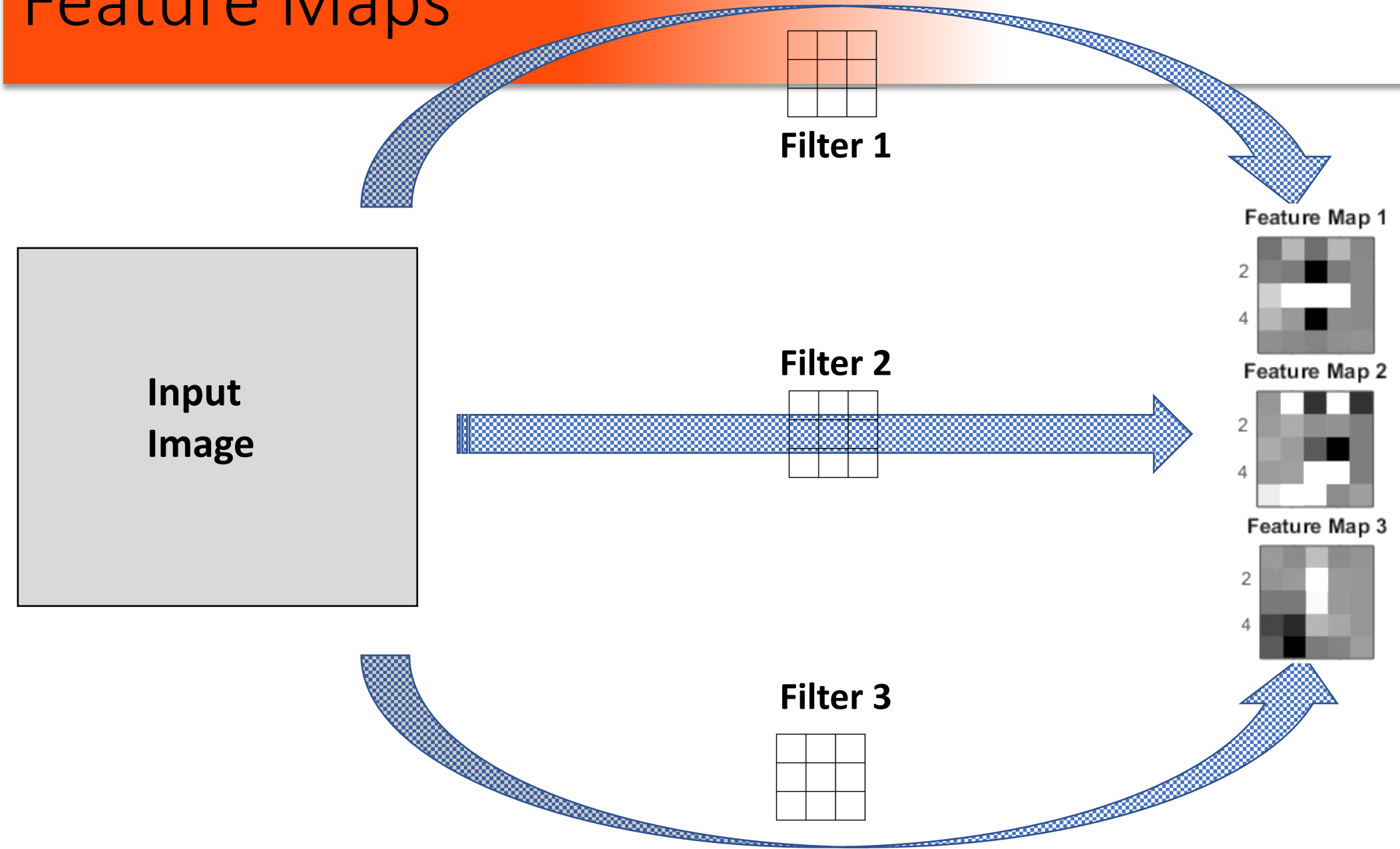






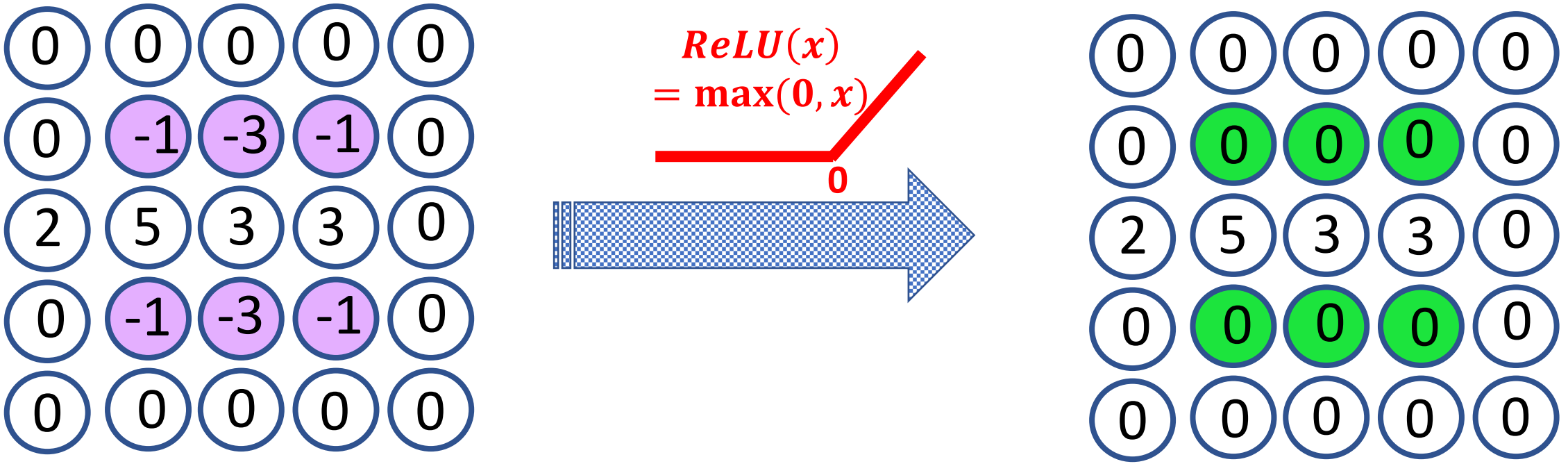


Feature Maps



- **Size of the patches extracted from the inputs**: These are typically 3 × 3 or 5 × 5
- **Depth of the output feature map**: The number of filters computed by the convolution.

ReLU Activation



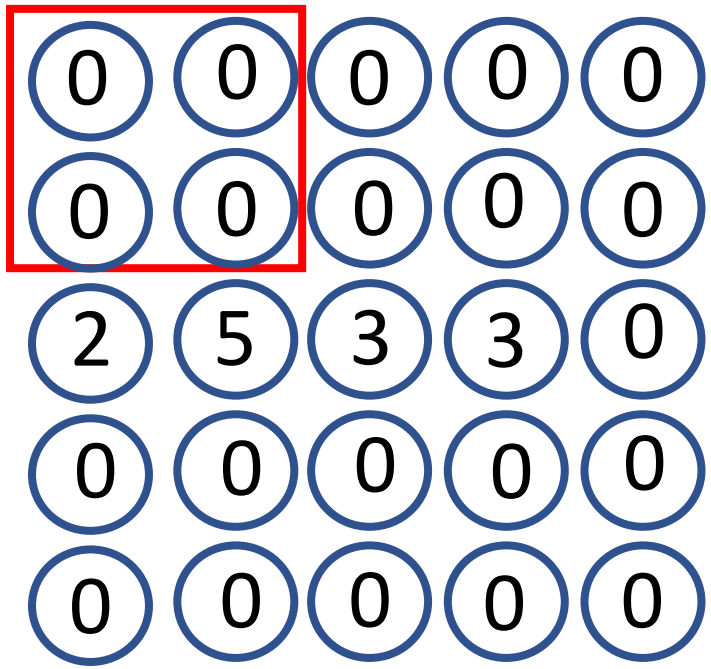
Why ReLU Activation

- ReLU activation function has become prevalent in deep networks.
- It accelerates the convergence of stochastic gradient descent compared to the other activation functions.
  - The gradient computation is very simple (either 0 or 1 depending on the sign of x).
- It also helps with **sparsity of the activation.**
  - Imagine a big neural network with a lot of neurons. Using a sigmoid or tanh will cause almost all neurons to fire in an analog way.
  - Using a sigmoid or tanh will cause almost all neurons to fire in an analog way.
  - ReLu give this benefit where random initialized weights and almost 50% of the network yields 0 activation because of the characteristic of ReLu (output 0 for negative values of x).
  - This means a fewer neurons are firing (sparse activation) and the **network is lighter.**

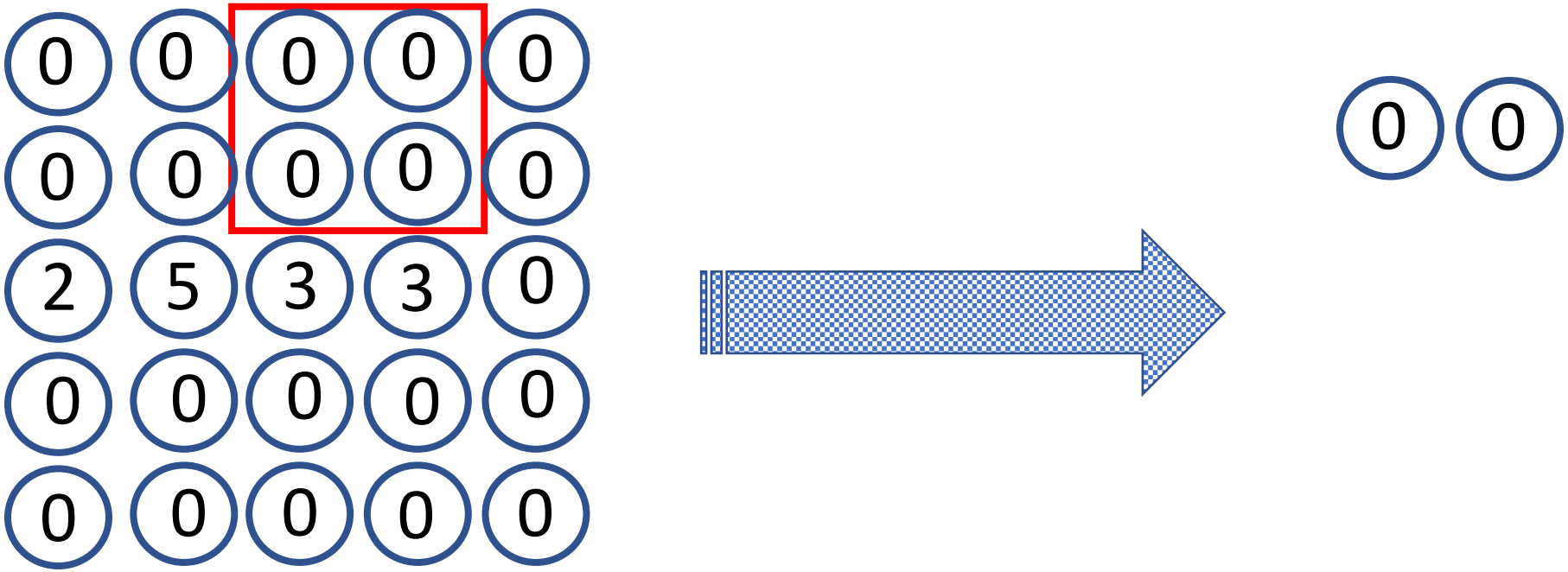
Pooling Layer

- Convolutional layers are usually followed by pooling layers. **The objective of pooling is to pool information from the convolutional layer and summarize it in a smaller feature map and hence end up with less parameters in subsequent layers.**
- In the pooling (also known as subsampling) process, we choose a grid, for example a 2x2 grid, on the feature map in the convolutional layer. We then choose the maximum activation value on that grid and replace the entire grid with that max value which means effectively we've done a 4 to 1 compression.

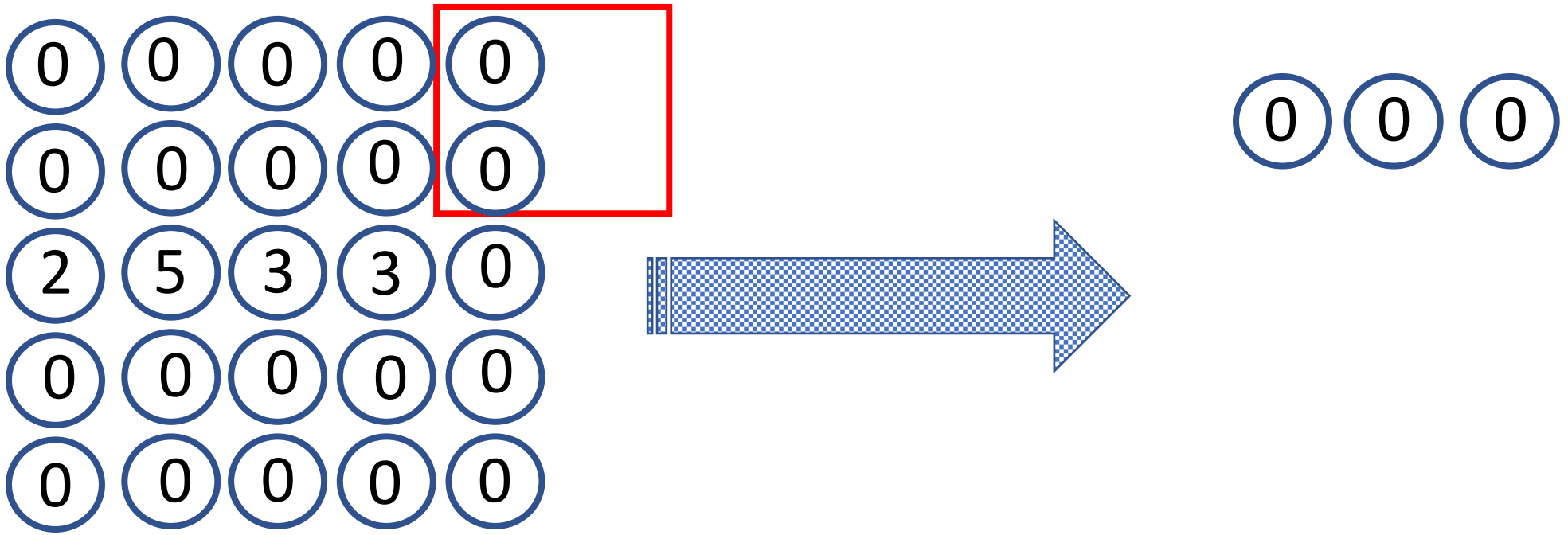
Pooling Layer



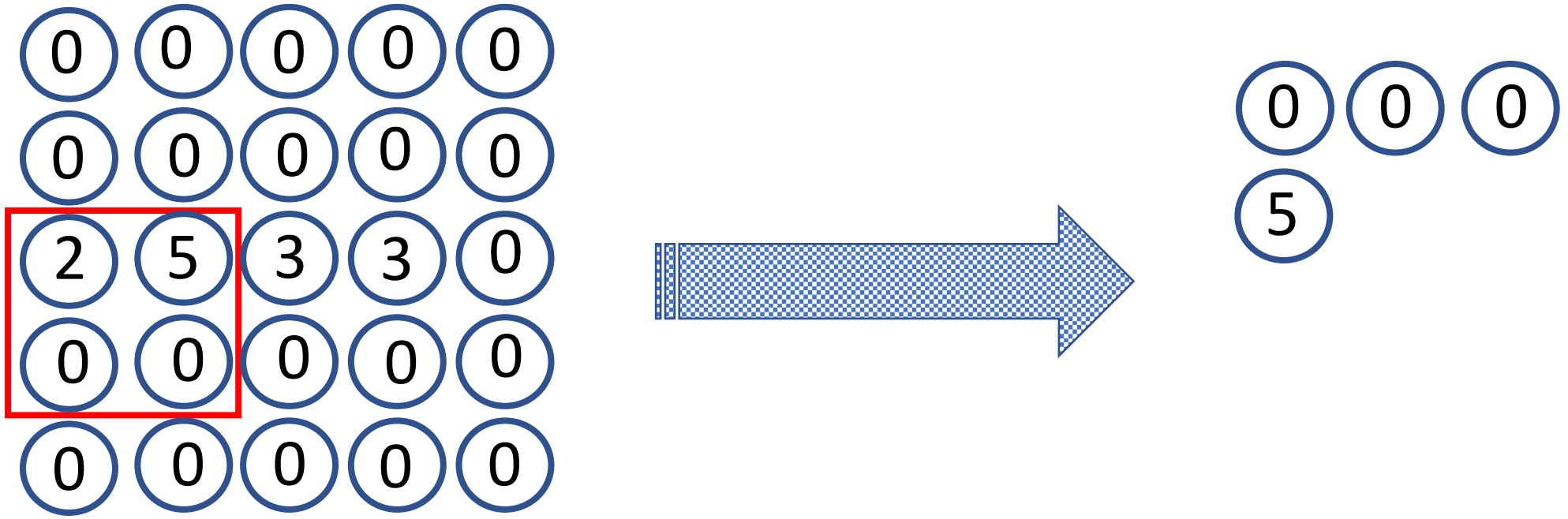
Pooling Layer



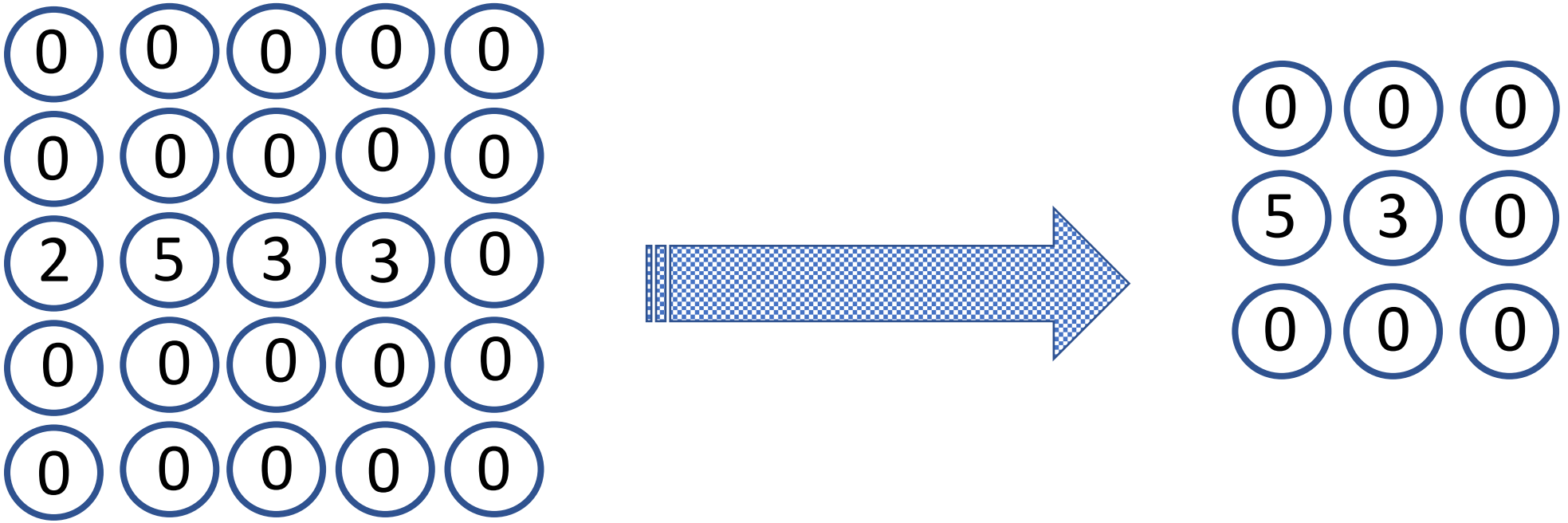
Pooling Layer



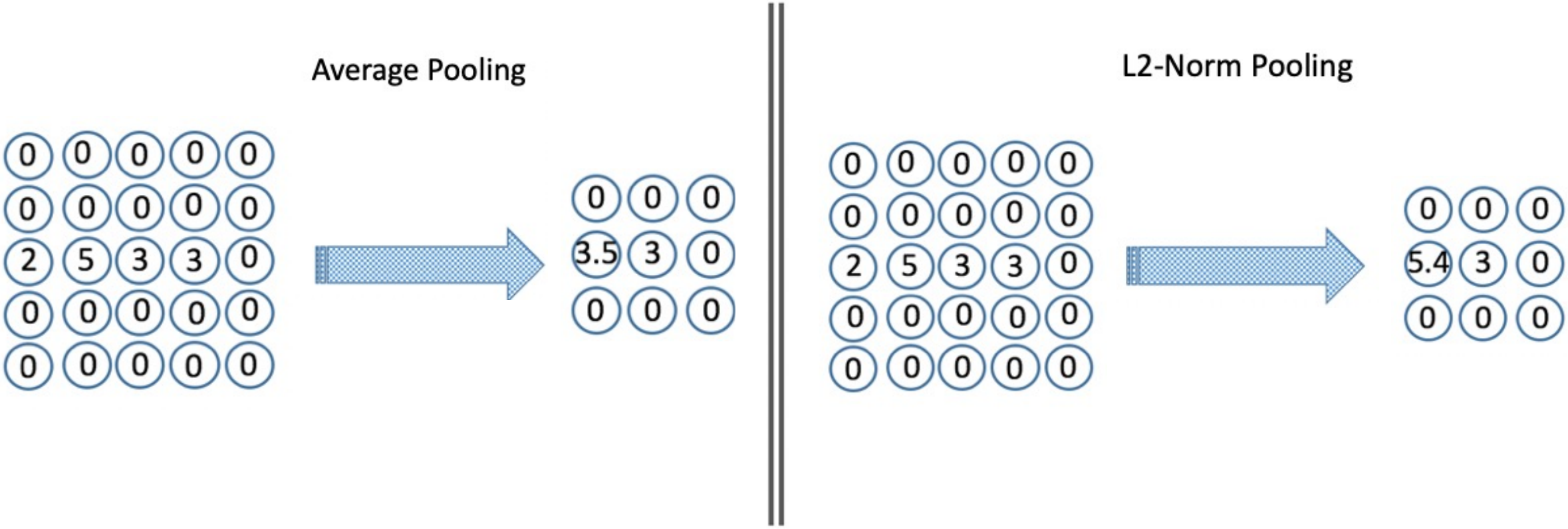
Pooling Layer



Pooling Layer



Other Methods for Pooling

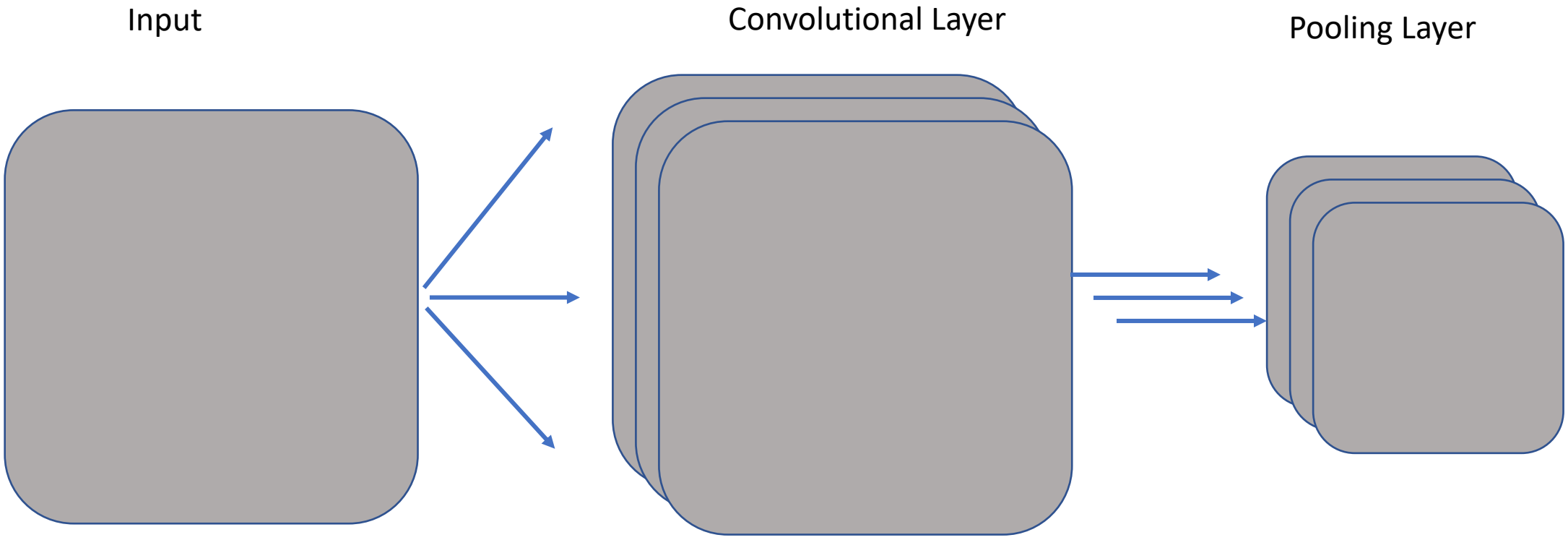


Why Pooling?

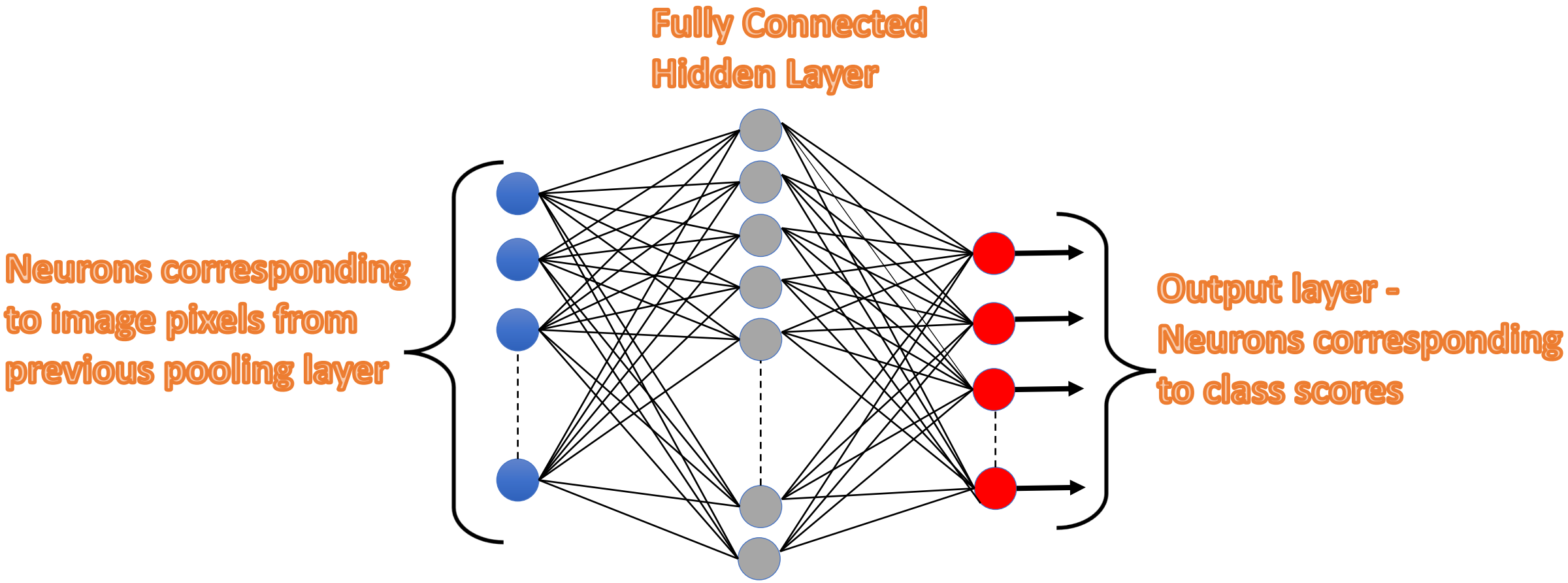
- A limitation of the feature map output is that they record the **precise position** of features in the input.
- This means that small movements in the position of the feature in the input image will result in a different feature map. This can happen with re-cropping, rotation, shifting, and other minor changes to the input image.
- Downsampling, creates a **lower resolution** version of an input signal that still contains the large or important structural elements, without the fine detail that may not be as useful to the task.
- Pooling helps to make the representation become **approximately invariant** to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change.

Why Max Pooling?

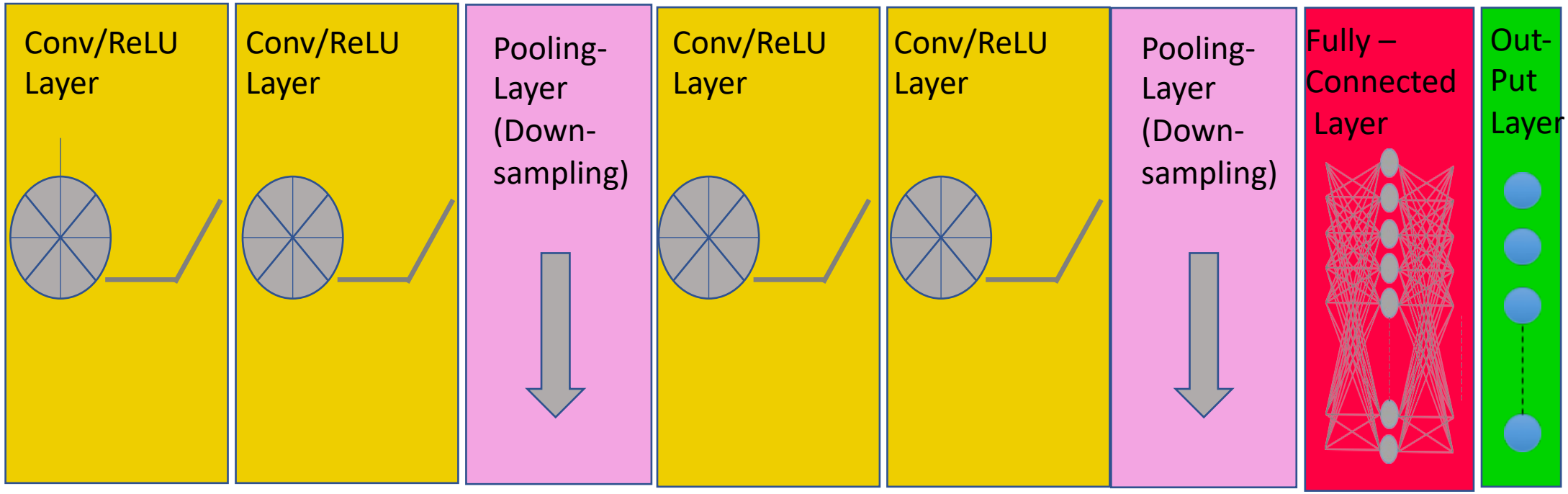
- Note that max pooling isn't the only way you can achieve such downsampling.
- You can also use strides in the prior convolution layer as well as average/L2 norm pooling instead of max pooling. But max pooling tends to work better than these alternative solutions.
- The reason is that it's **more informative to look at the maximal presence** of different features than at their average presence.
- So the most reasonable subsampling strategy is to first produce dense maps of features (via unstrided convolutions) and then look at the maximal activation of the features over small patches, rather than looking at sparser windows of the inputs (via strided convolutions) or averaging input patches, which could cause you to miss or dilute feature-presence information.

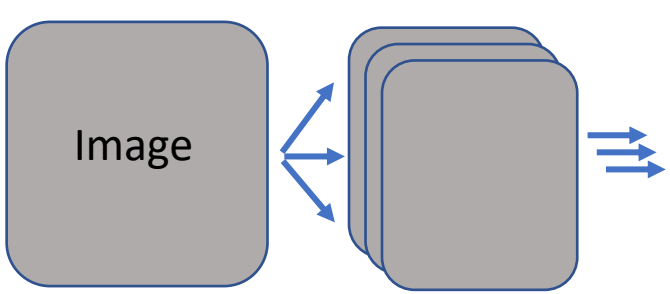

Input
Convolutional Layer
Pooling Layer
5x5
Original Image
(before zero-padding)
5x5x3-volume
Feature-map
3x3x3-volume
Feature-map

Last Layer: Fully Connected Layer



Full ConvNet





Case Study: CIFAR-10 Image Set Classification

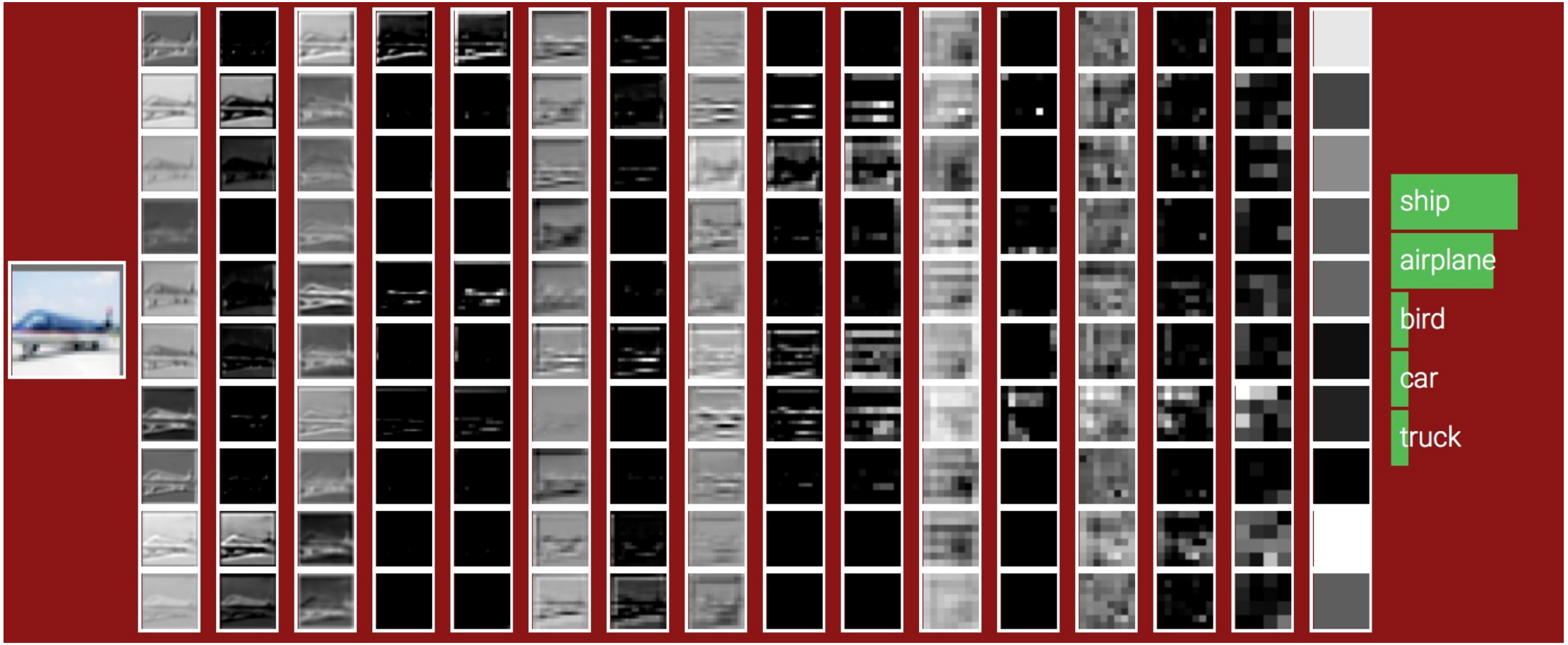
CIFAR, collected by Alex Krizhevsky and his colleagues at University of Toronto, has 60,000 images in the following categories:

- Bird
- Car
- Cat
- Deer
- Dog
- Frog
- Horse
- Plane
- Ship
- Truck

Case Study: CIFAR-10 Image Set Classification

See a live demo of several deep learning networks including this one on Karpathy's Website.

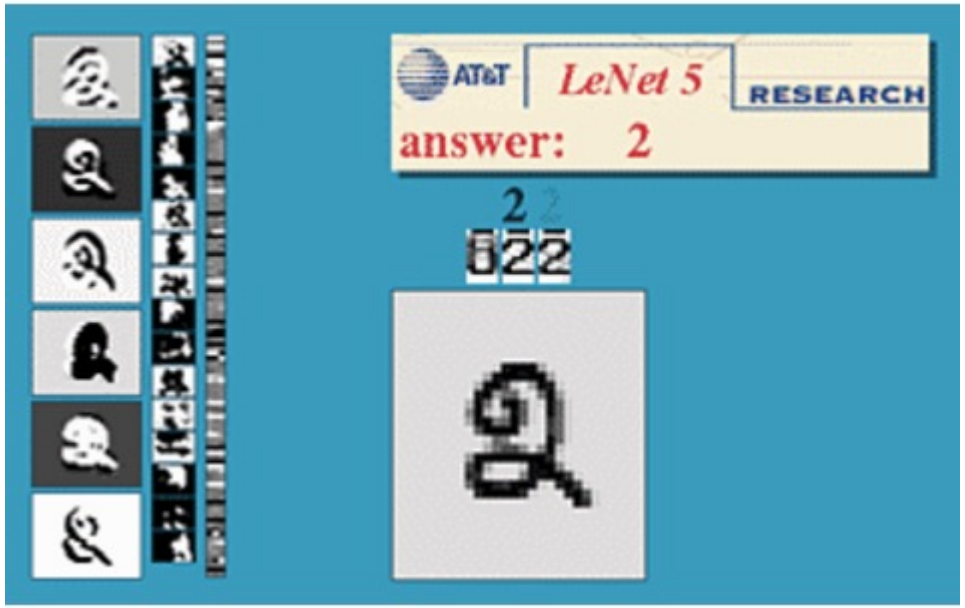
Case Study: CIFAR-10 Image Set Classification

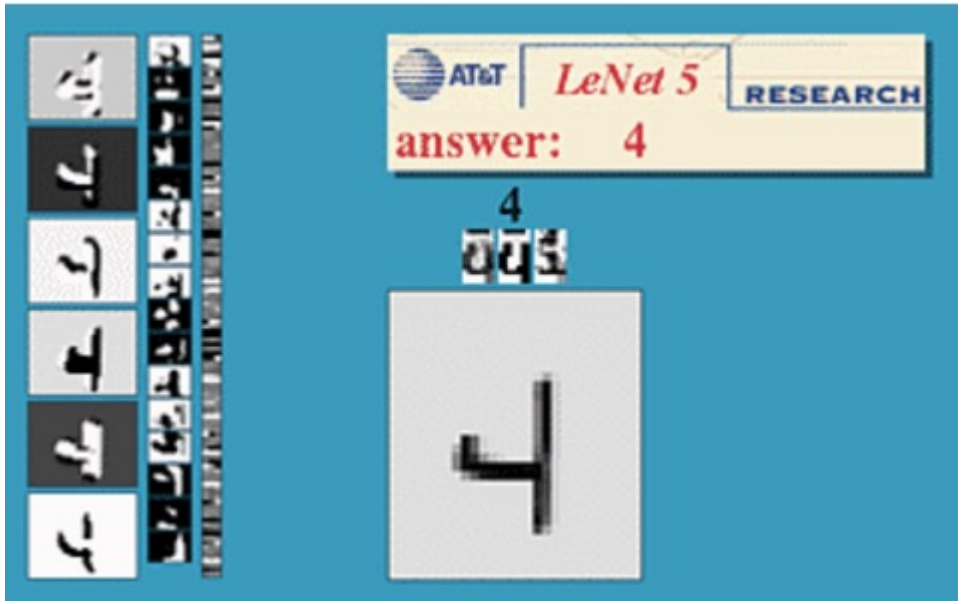


See a live demo of several deep learning networks including this one on Karpathy's Website.

Deep Learning in Action

Yann LeCun has a neat live demo at http://yann.lecun.com/exdb/lenet/



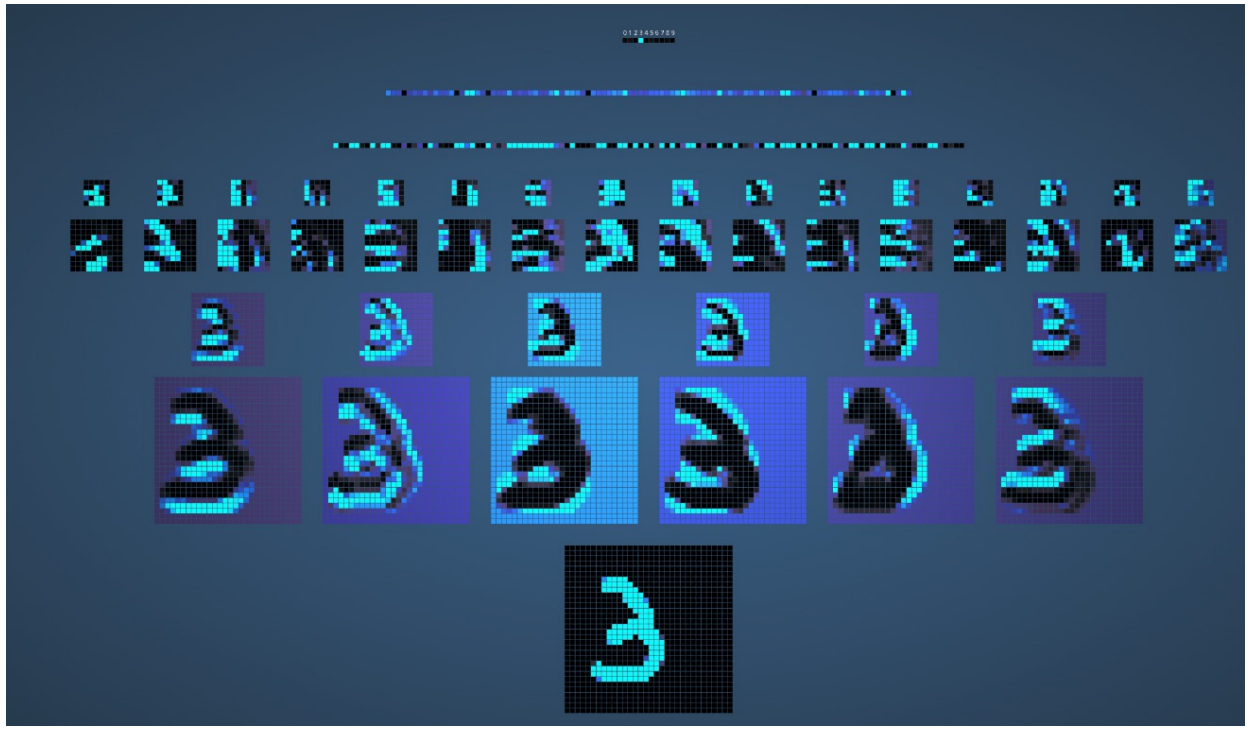


From: http://yann.lecun.com/exdb/lenet/unusual.html

Deep Learning in Action

http://scs.ryerson.ca/~aharley/vis/conv/flat.html

http://scs.ryerson.ca/~aharley/vis/conv/



Final Remarks

ConvNets are special purpose networks that are most suited for problems where the input data has local spatial or temporal correlation.

ConvNets have many hyper-parameters that need to be set

Employ an ensemble of networks rather than just one to improve performance.

ConvNet in Python

- This code will be a simple convnet to classify the MNIST digits.
- You will see that this basic convnet will outperform the performance of the densely connected model we implemented earlier.

```
1  import numpy as np
2  import plotly.express as px
3  import plotly.graph_objects as go
4  import tensorflow as tf
5
6  from keras.datasets import mnist
7  from keras.models import Sequential
8  from keras.layers import Dense,Dropout, Conv2D, Flatten, MaxPooling2D
9  from tensorflow.keras.optimizers import RMSprop, Adam
10 from keras.callbacks import ModelCheckpoint, EarlyStopping
11
12 from keras.utils import np_utils
```

Preprocessing the Data

```
1 (X_train, Y_train), (X_test, Y_test) = mnist.load_data()
2 print(X_train.shape, Y_train.shape, X_test.shape, Y_test.shape)
3 print(X_train.dtype, Y_train.dtype)
```

```
(60000, 28, 28) (60000,) (10000, 28, 28) (10000,)
uint8 uint8
```

```
1 X_train = X_train/255
2 X_test = X_test/255
```

```
1 Y_train = np_utils.to_categorical(Y_train, 10)
2 Y_test = np_utils.to_categorical(Y_test, 10)
```

Instantiating the Convnet and Adding the Fully Connected Layers

- The architecture consists of a stack of Conv2D and MaxPooling2D layers.
- Note that the first layer takes as input tensors of shape (image_height, image_width, image_channels).
- Next we feed the last output tensor into a densely connected classifier
- Note that Dense layers process 1D vectors. Therefore, we have to flatten our 3D outputs to 1D, and then add a few Dense layers on top.

```
:
1  cnnModel = Sequential()
2  cnnModel.add(Conv2D(filters = 32 , kernel_size = (3,3), activation = 'relu', input_shape = (28,28,1)))
3  cnnModel.add(MaxPooling2D(pool_size = (2,2)))
4  cnnModel.add(Conv2D(filters = 64 , kernel_size = (3,3), activation = 'relu'))
5  cnnModel.add(MaxPooling2D(pool_size = (2,2)))
6  cnnModel.add(Conv2D(filters = 64 , kernel_size = (3,3), activation = 'relu'))
7  cnnModel.add(Flatten())
8  cnnModel.add(Dense(64,activation = 'relu'))
9  cnnModel.add(Dense(10, activation = 'softmax'))
10 cnnModel.summary()
```

Model Summary

$$\begin{pmatrix} 3\times3(filter\ parameters)\times32(num\ of\ filters \\ + \\ 32\ (biases\ for\ each\ filter) \end{pmatrix} = 320$$

```
model.summary()
```

Layer (type)	Output Shape	Param #
conv2d_22 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_17 (MaxPooling	(None, 13, 13, 32)	0
conv2d_23 (Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d_18 (MaxPooling	(None, 5, 5, 64)	0
conv2d_24 (Conv2D)	(None, 3, 3, 64)	36928
flatten_1 (Flatten)	(None, 576)	0
dense_1 (Dense)	(None, 64)	36928
dense_2 (Dense)	(None, 10)	650

Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0

$$\begin{pmatrix} 3\times3\times32(filter\ parameters)\times64(num\ of\ filters \\ + \\ 64\ (biases\ for\ each\ filter) \end{pmatrix} = 18496$$

$$3(height)\times3(width)\times64(depth) = 576$$

$$576(num\ of\ input\ neurons)\times 64(num\ of\ hidden\ neurons) + 64(biases\ for\ each\ hidden\ neuron) = 36928$$

$$64(num\ of\ input\ neurons)\times 10(num\ of\ output\ neurons) + 10(biases\ for\ each\ ouput\ neuron) = 650$$

Model Summary

```
10 cnnModel.summary()
```

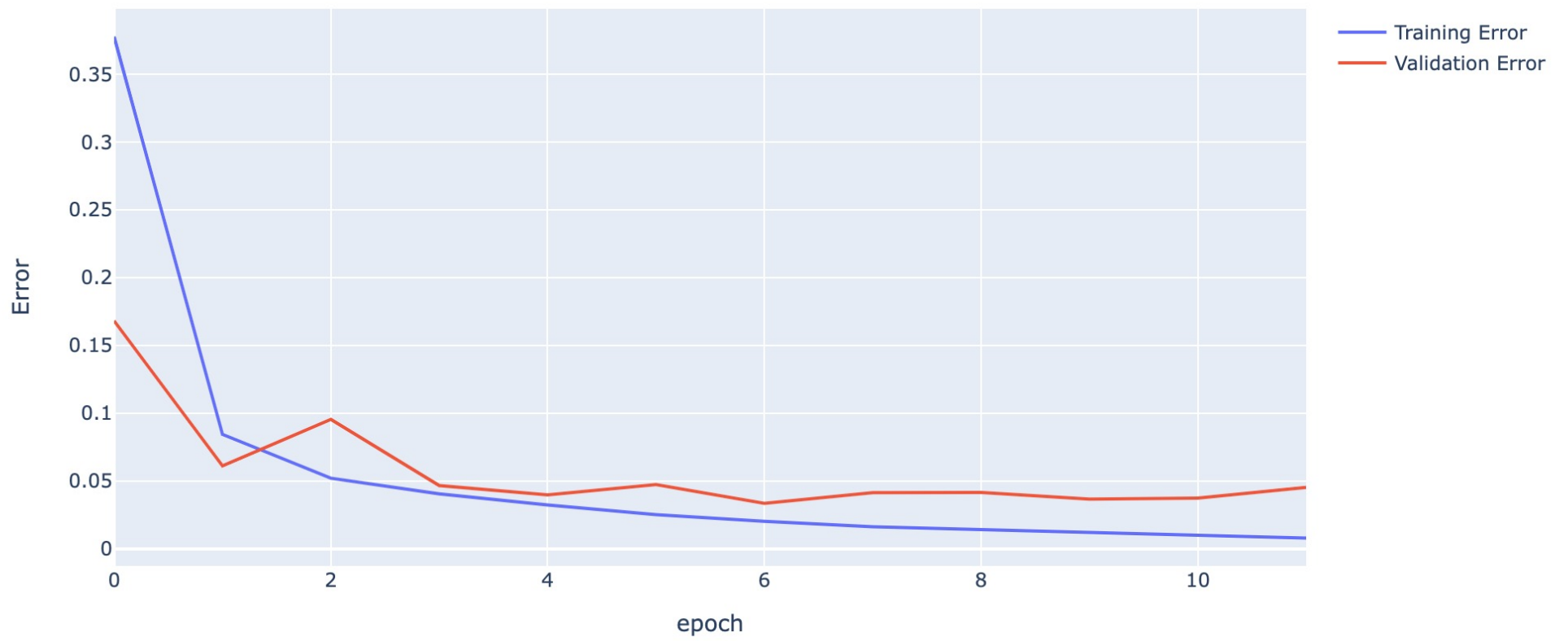
```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 conv2d_4 (Conv2D)            (None, 26, 26, 32)        320

 max_pooling2d_2 (MaxPooling  (None, 13, 13, 32)        0
 2D)

 conv2d_5 (Conv2D)            (None, 11, 11, 64)        18496

 max_pooling2d_3 (MaxPooling  (None, 5, 5, 64)          0
 2D)

 conv2d_6 (Conv2D)            (None, 3, 3, 64)          36928

 flatten_1 (Flatten)          (None, 576)               0

 dense_2 (Dense)              (None, 64)                36928

 dense_3 (Dense)              (None, 10)                650

=================================================================
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
```

- Note that the output of every Conv2D and MaxPooling2D layer is a 3D tensor of shape (height, width, channels(filters)).
- The width and height dimensions tend to shrink as you go deeper in the network.
- The number of channels is controlled by the first argument passed to the Conv2D layers (32 or 64).

Training the Model

- cnnModel.compile(loss = 'categorical_crossentropy', optimizer = 'RMSprop', metrics = ['categorical_accuracy’])
- checkpoint = EarlyStopping(monitor='val_loss',patience=5, verbose=1, mode='auto', restore_best_weights=True)
- cnnModel.fit(X_train, Y_train, epochs =100, batch_size = 256, validation_split = 1/6, verbose = 1, callbacks = [checkpoint])

```
fig = go.Figure()
fig.add_trace(go.Scatter(y=cnnModel.history.history['loss'],
                    mode='lines',
                    name='Training Error'))
fig.add_trace(go.Scatter(y=cnnModel.history.history['val_loss'],
                    mode='lines',
                    name='Validation Error'))
fig.update_layout(yaxis_title = 'Error', xaxis_title = 'epoch')
fig.show()
```



Model Evaluation

- Recall that the densely connected network web implemented earlier had a test accuracy of about 97%.
- This basic convnet has a test accuracy of about 99%.
- This mounts to a 67% relative error decrease $(\frac{0.03-0.01}{0.03} = 0.666).0$

```
print(cnnModel.metrics_names[0],'is', cnnModel.evaluate(X_test, Y_test)[0],'and ',
cnnModel.metrics_names[1],'is', cnnModel.evaluate(X_test, Y_test)[1])
```

References and Recommended Readings

- Networks and Deep Learning by Michael Nielsen
- Deep Learning by Ian Goodfellow and Yoshua Bengio and Aaron Courville
- Learning from Data by Yaser Abu-Mustafa, Malik Magdon-Ismail, and Hsuan_Tien Lin
- http://deeplearning.net
- Deep Learning with Python by Chollet