Convolutional Neural Network

Computer Vision Problems

- Some of them are :
  - Image classification
  - Object detection
  - Neural Style Transfer : Transferring style from one picture to another

Disadvantage of Multi layered neural Network

- Input for computer vision problems are very big.
- An image of size = 64x64x3 = 12288 features
- An image of size = 1000x1000x3 = 3000000 features
- For 3000000 features we need 3000000 neurons in the input layer of the neural network (as we flatten the image and feed as input).
- The hidden layers will need more input parameters as the size of the input is large. This results in :
  - Huge amount of memory
  - Huge amount of data as input is required
  - May even lead to overfitting

Convolution Operation

- Convolution Operation is considered as one of the building blocks of Convolutional neural network.
- A convolutional layer is the most important component of a CNN.
- It comprises a set of filters (also called convolutional kernels) which are convolved with a given input to generate an output feature map.
- A **filter** is essentially a feature detector.
- Each filter in a convolutional layer is a grid of discrete numbers.

2	0
-1	3

Figure : An example of a 2D image filter.

- The weights of each filter (the numbers in the grid) are learned during the training of CNN.
- This learning procedure involves a random initialization of the filter weights at the start of the training.
- Afterward, given input-output pairs, the filter weights are tuned in a number of different iterations during the learning procedure.
- The convolution layer performs convolution between the filters and the input to the layer.

Example 1

- Let's say that we want to detect vertical and horizontal lines in the image.
- This result is our **feature map**, and it indicates where we've found the feature we're looking for in the original image.
- This operation is called a convolution.
- We take a filter and we multiply it over the entire area of an input image.

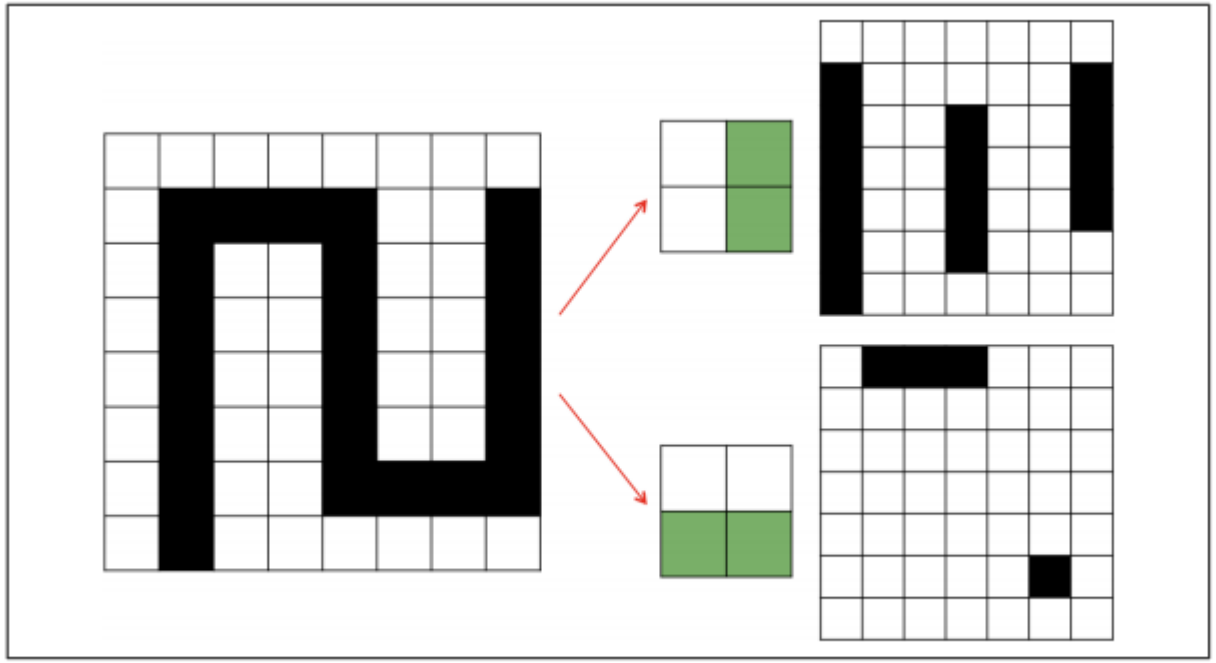
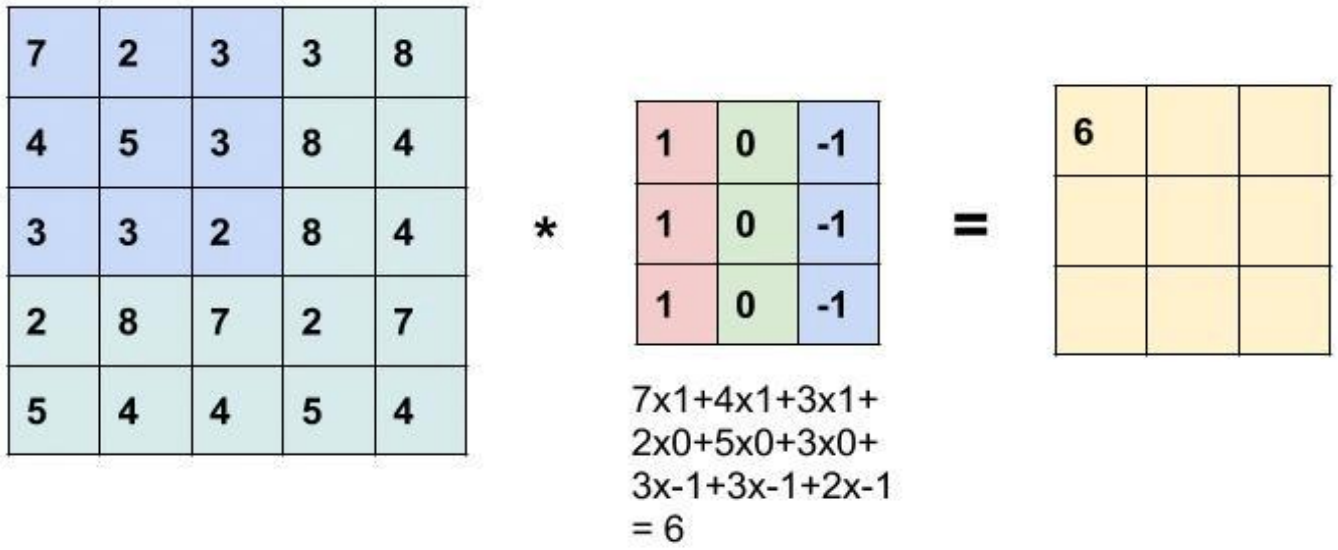
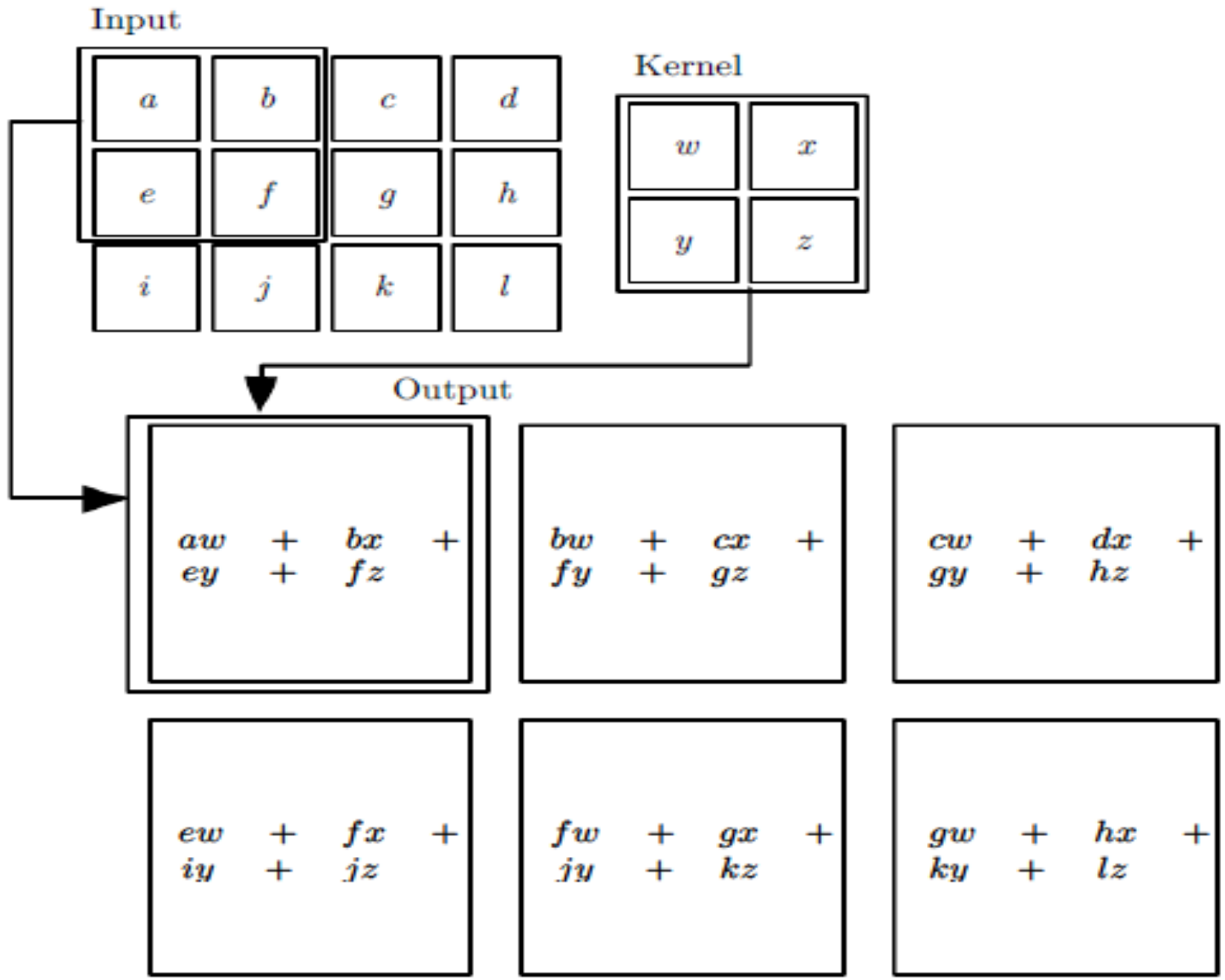


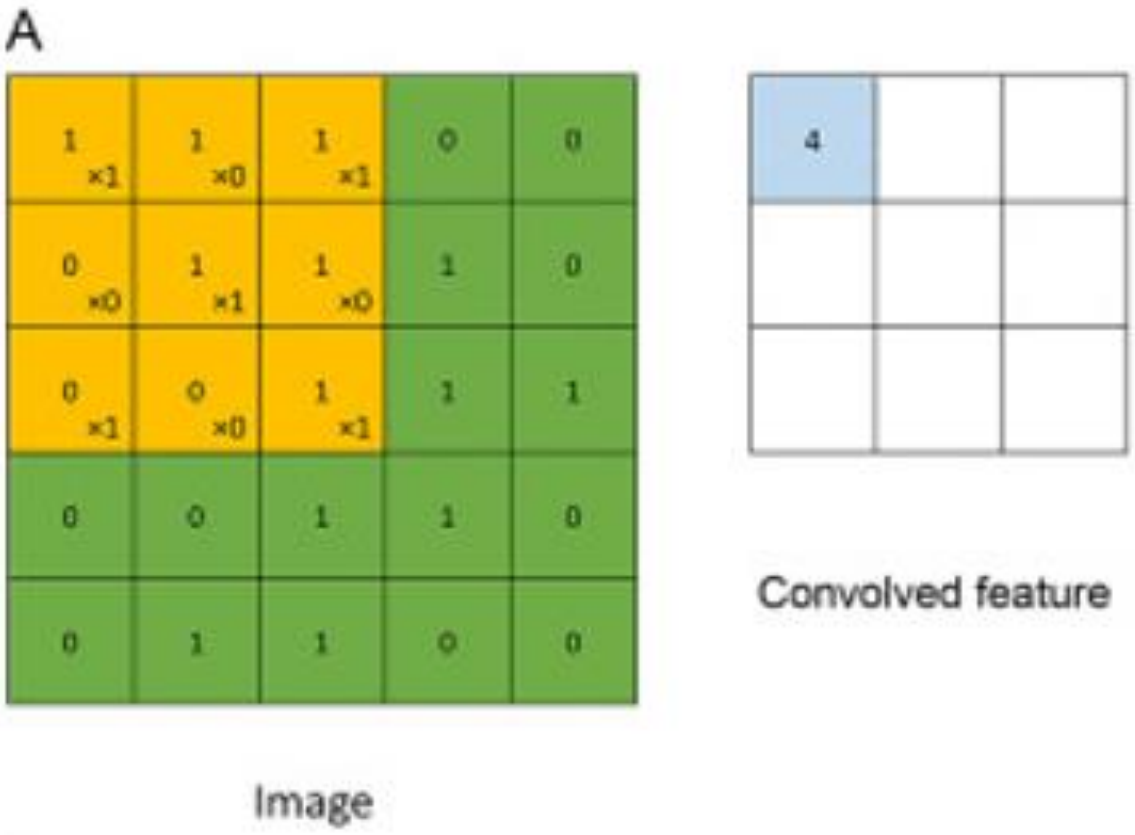
Figure 5-6. Applying filters that detect vertical and horizontal lines on our toy example

Example 2

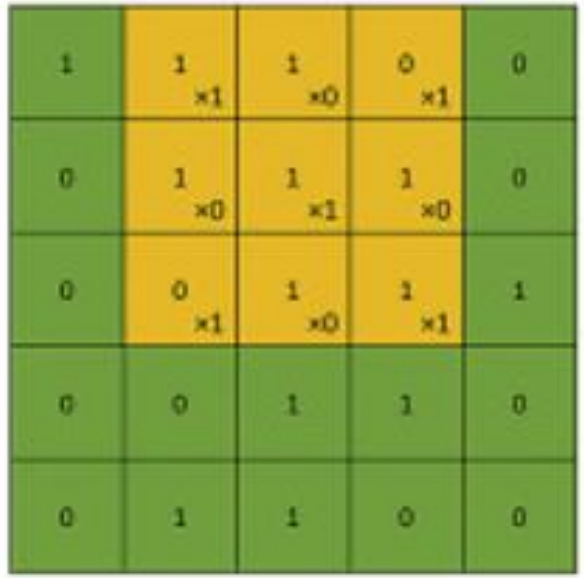

7 2 3 3 8
4 5 3 8 4
3 3 2 8 4
2 8 7 2 7
5 4 4 5 4
*
1 0 -1
1 0 -1
1 0 -1
=
6
7x1+4x1+3x1+
2x0+5x0+3x0+
3x-1+3x-1+2x-1
= 6


Input
a
b
c
d
e
f
g
h
i
j
k
l
Kernel
w
x
y
z
Output
aw + bx + ey + fz
bw + cx + fy + gz
cw + dx + gy + hz
ew + fx + iy + jz
fw + gx + jy + kz
gw + hx + ky + lz

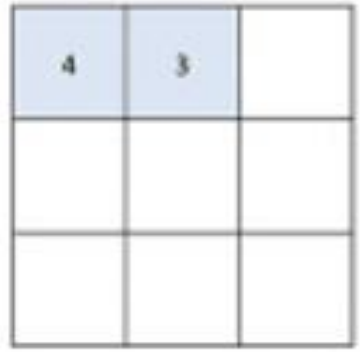
Example 3


A
1 ×1 1 ×0 1 ×1 0 0
0 ×0 1 ×1 1 ×0 1 0
0 ×1 0 ×0 1 ×1 1 1
0 0 1 1 0
0 1 1 0 0
Image
4
Convolved feature

B

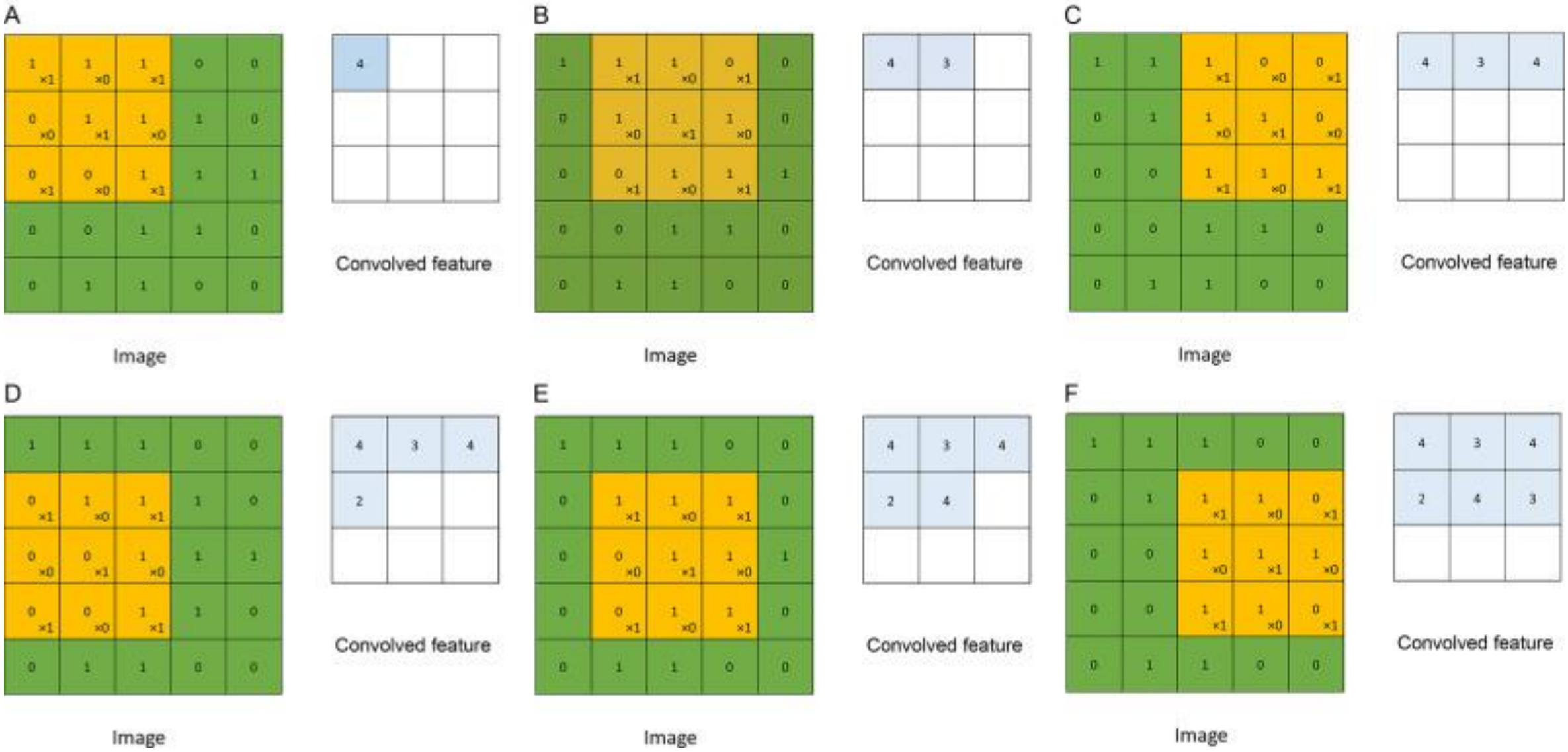


Image



Convolved feature

Note that the filter slides along the width and height of the input feature map and this process continues until the filter can no longer slide further.



Padding

0	0	0	0	0	0	0
0	60	113	56	139	85	0
0	73	121	54	84	128	0
0	131	99	70	129	127	0
0	80	57	115	69	134	0
0	104	126	123	95	130	0
0	0	0	0	0	0	0

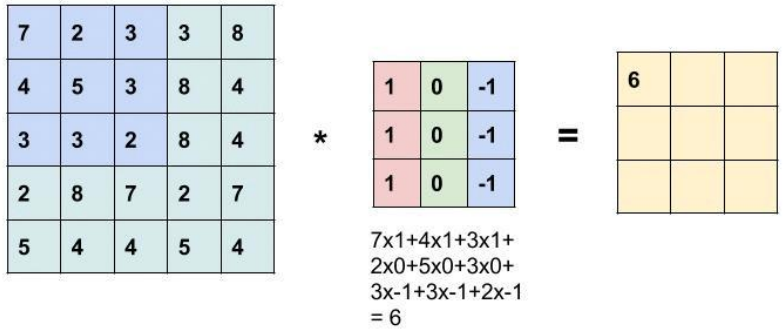
Kernel

0	-1	0
-1	5	-1
0	-1	0

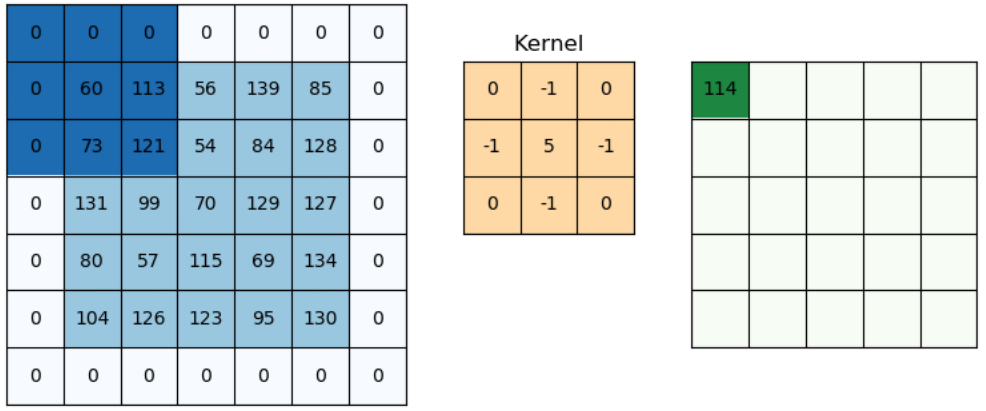
114				

Need for padding

- Without padding if we consider an nxn image with fxf filter, then the output will have : n-f+1 x n-f+1 size



Without padding



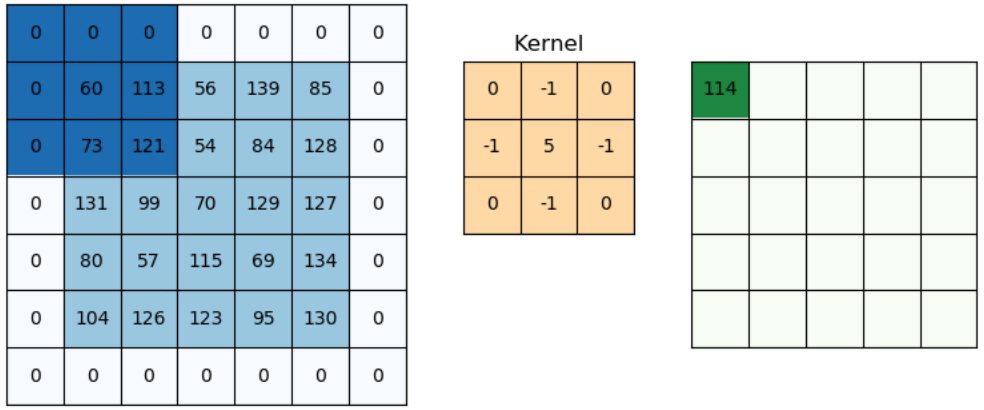
With padding

- Convolution always shrinks our output.
- Pixels at the edge/corners are used only once during feature detection, than the other pixels which are at the center of the image and are used more than once.
- Less usage of pixel throws information from the edge.
- To overcome the above two problems, we make use of padding.
- With padding : our output is of size => n+2p-f+1 * n+2p-f+1
- For same output to be of the same size as that of input :

$n+2p-f+1 = n$

$2p-f+1 = 0$

$p = (f-1)/2$



Effect of Different types of filters

$$* \begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix} =$$

blurs the image

$$* \begin{matrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{matrix} =$$

sharpens the image

Neighbors are similar and are –ve , so only the center pixel is magnified. So the image sharpens.

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

$*$ $=$

detects the edges

- Acts like an edge detector.
- If the neighboring pixels have same value , the output pixel will be zero.
- If neighboring pixels have different value, then the output will have different value at the boundary, acting like an edge detector.

Stride

- In the earlier examples, in order to calculate each value of the output feature map, the filter takes a step of 1 along the horizontal or vertical position (i.e., along the column or the row of the input).
- This step is termed as the stride of the convolution filter, which can be set to a different (than 1) value if required.
- Compared to the stride of 1, the stride of 2 results in a smaller output feature map.
- This reduction in dimensions is referred to as the sub-sampling operation.
- Such a reduction in dimensions provides moderate invariance to scale and pose of the objects, which is a useful property in applications such as object recognition.

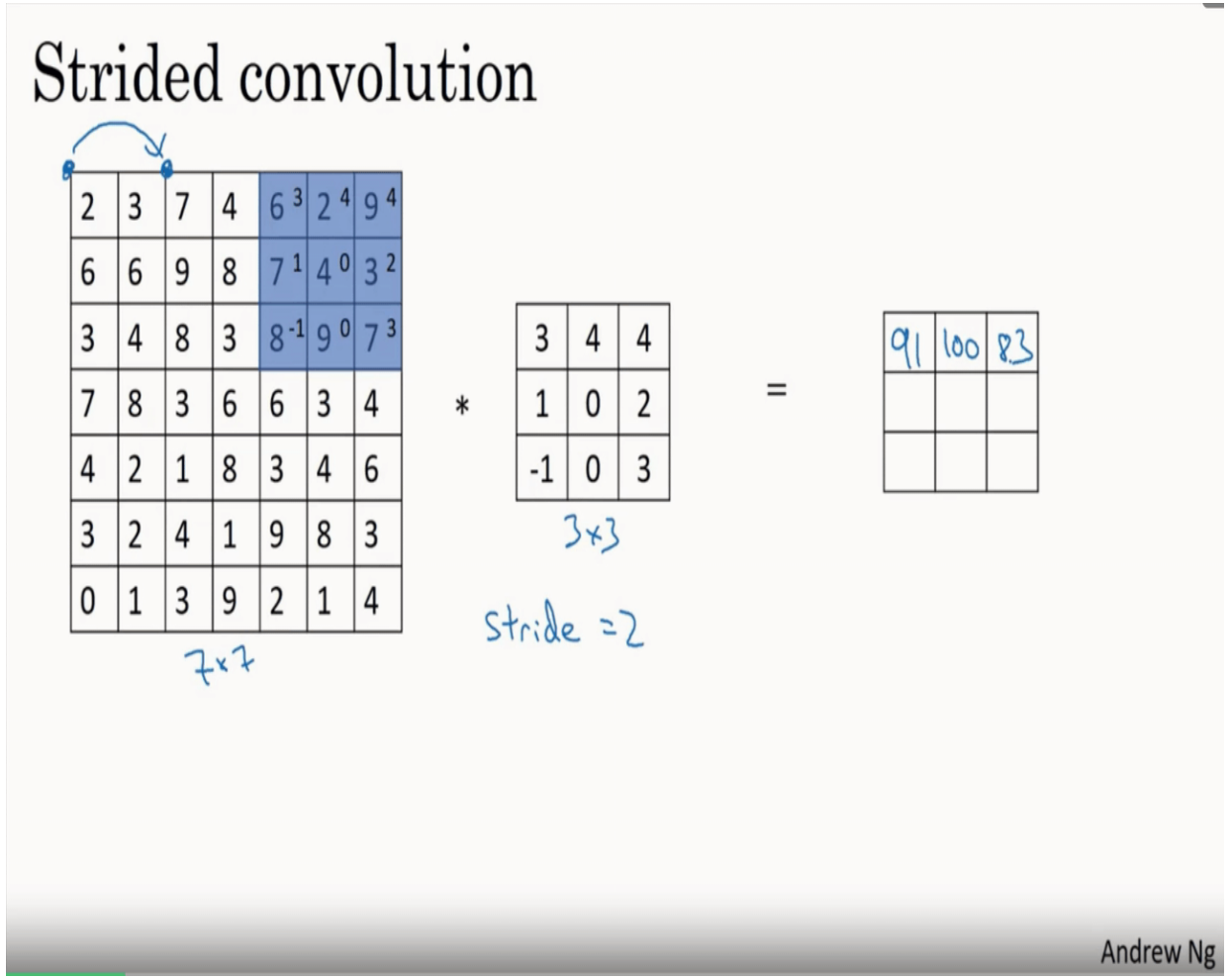
Difference between stride = 1 and stride =2

0	0	0	0	0	0	0
0	60	113	56	139	85	0
0	73	121	54	84	128	0
0	131	99	70	129	127	0
0	80	57	115	69	134	0
0	104	126	123	95	130	0
0	0	0	0	0	0	0

Kernel

0	-1	0
-1	5	-1
0	-1	0

114				

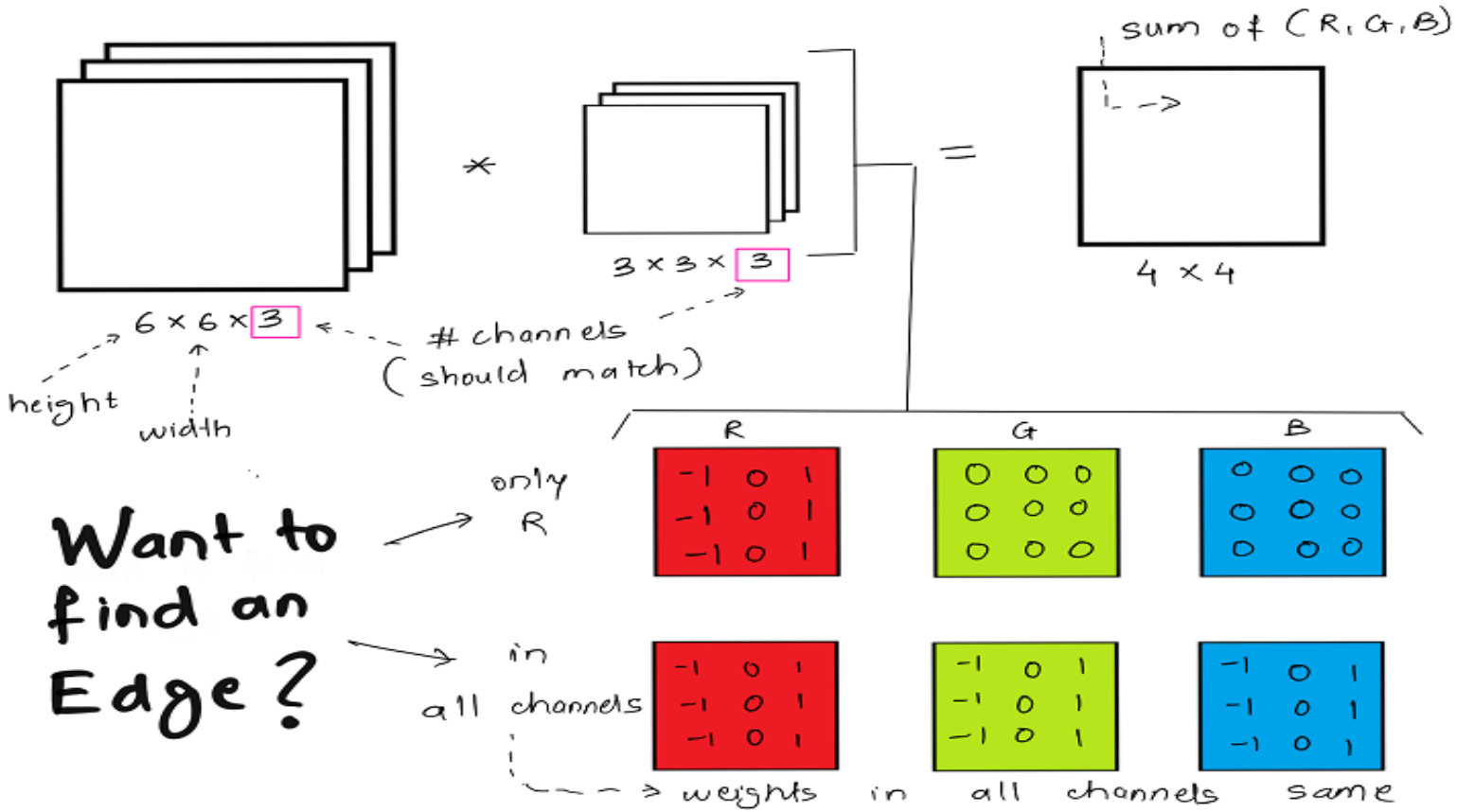


- nxn * fxf $\Rightarrow \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$
  - What we have been doing from last few exercises is called cross correlation, instead of convolution.
  - Convolution means to flip the 3x3 filter first and then we convolve it with nxn matrix.

Convolution over volumes

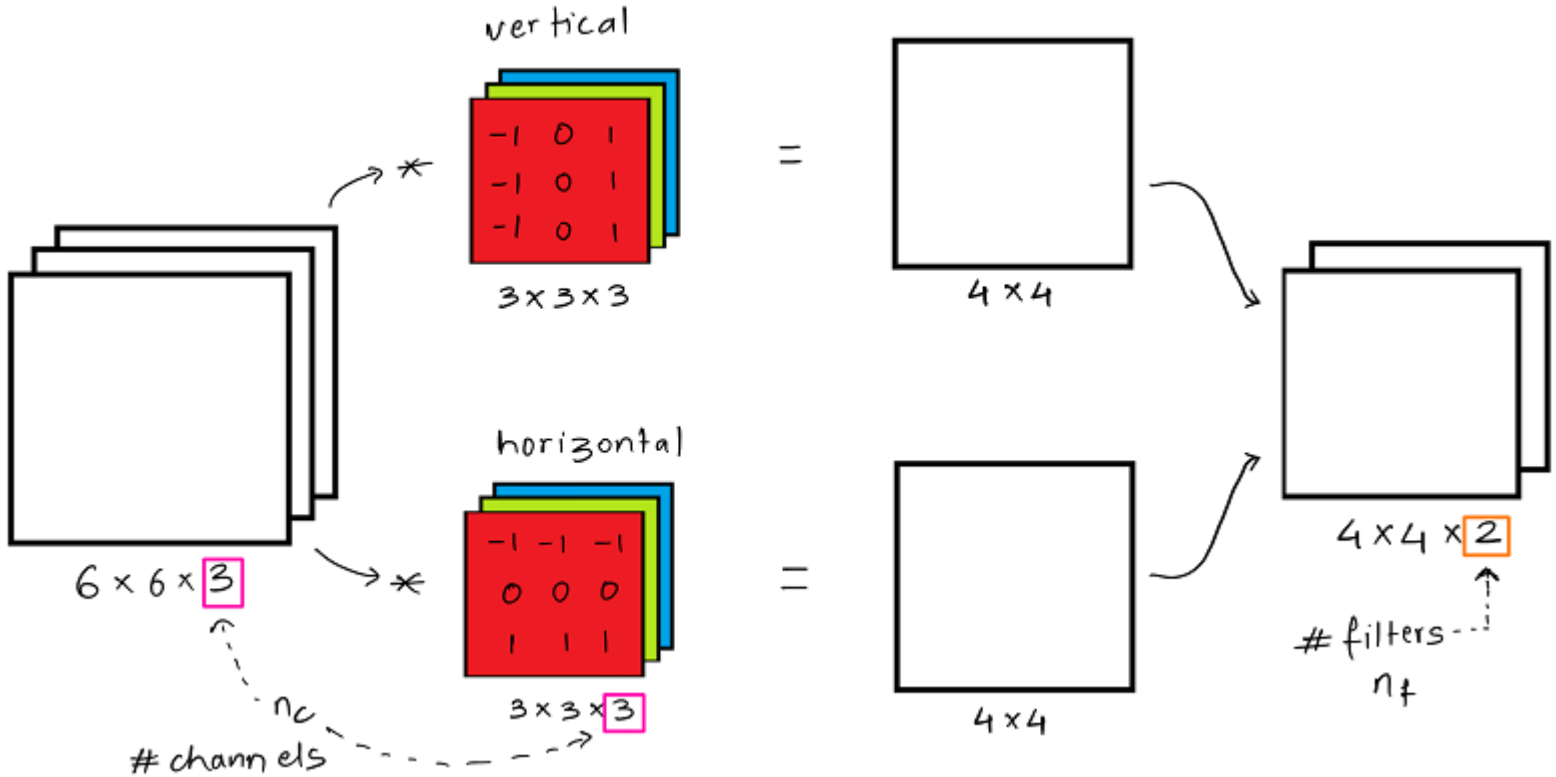
- We try to find features in a color image.
- So if the image dimension is n x n x #channels, so the filter which was earlier f x f would also now be required to be of dimension f x f x #channels.
- Intuitively our input data is no more 2 dimensional but in fact 3 dimension if you consider the channels, and hence the name **volume**.

- Below is a simple example of convolution over volume, of an image having dimension 6 x 6 x 3 with 3 denoting the 3 channels R, G and B. Similarly the filter is of dimension 3 x 3 x 3.



Multiple filters at one time

- There is a high chance that you may need to extract alot of different features from an image, for which you will use multiple filters.
- If individual filters are convolved separately, it will increase the computation time and so its more convenient to use all required filters at a time directly.
- Convolution is carried out individually as is the case with a single filter, and then results of both convolutions are merged together in a stack to form an output volume with a 3rd dimension representative of the numbers of filters.



The output dimension can be calculated for any general case using the following equation :

$$(n \times n \times n_c) * (f \times f \times n_c) = (n - f + 1) \times (n - f + 1) \times n_f$$

Here, nc is the number of channels in the input image and nf are the number of filters used.

Pooling

- Pooling is a key-step in convolutional based systems that reduces the dimensionality of the feature maps.
- It combines a set of values into a smaller number of values, i.e., the reduction in the dimensionality of the feature map.

Popular Pooling Methods

- **Average Pooling**

As shown in Fig. 1, an average pooling layer performs down-sampling by dividing the input into rectangular pooling regions and computing the average values of each region.

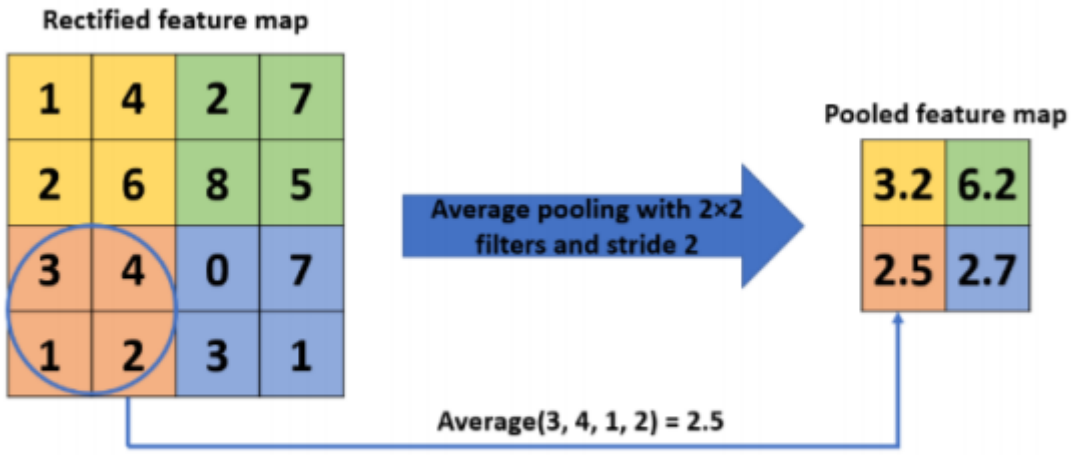


Fig. 1. Example of Average Pooling operation.

- **Max-Pooling**
- A max-pooling operator can be applied to down-sample the convolutional output bands, thus reducing variability.
- The max-pooling operator passes forward the maximum value within a group of R activations.

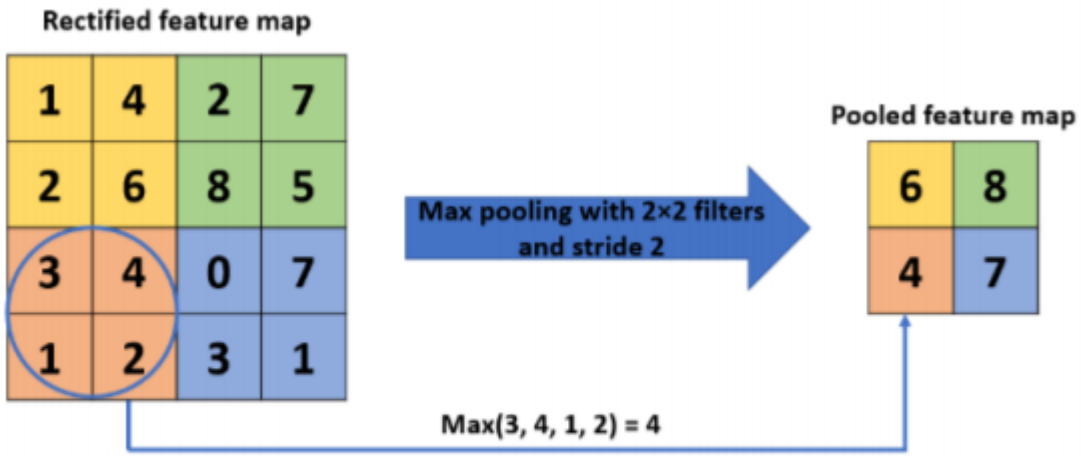


Fig. 2. Example of Max-Pooling operation.

- **Mixed Pooling**
- Max pooling extracts only the maximum activation whereas average pooling down-weighs the activation by combining the non-maximal activations.
- To overcome this problem, a hybrid approach by combining the average pooling and max pooling.

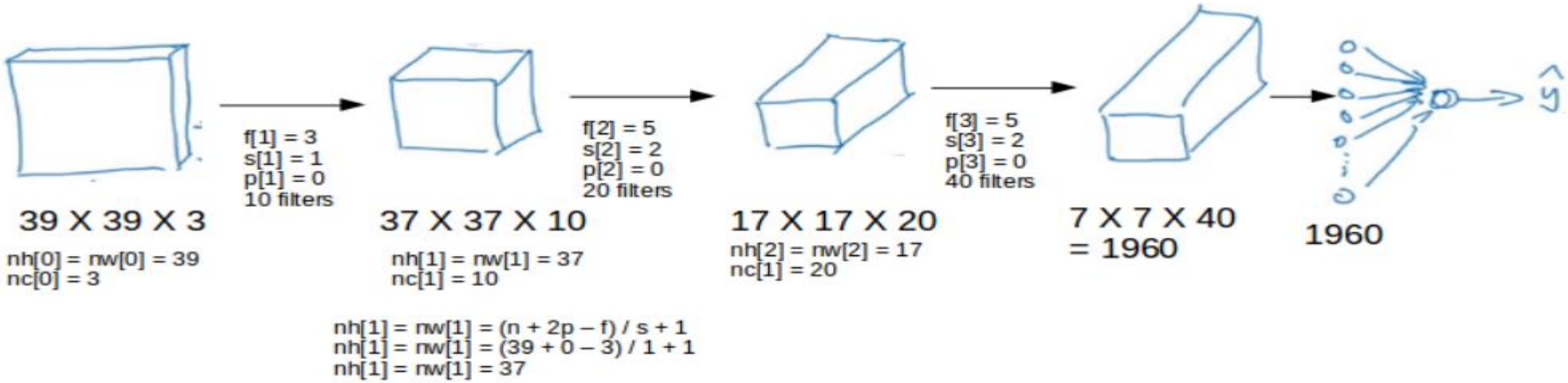
$$s_j = \lambda \max_{i \in R_j} a_i + (1 - \lambda) \frac{1}{|R_j|} \sum_{i \in R_j} a_i$$

- where λ decides the choice of either using max pooling or average pooling.
- The value of λ is selected randomly either 0 or 1.
- When λ = 0, it behaves like average pooling, and when λ = 1, it works like max pooling.

Advantage of CNN over classical neural network.

- A CNN differs from classical neural networks for the presence of convolutional layers, which can better model and discern the spatial correlation of neighboring pixels than normal fully connected layers.
- For a classification problem, the final outputs of the CNN are the classes which the network has been trained on.
- The training phase is usually extremely expensive from a computational point of view, and may take a long time to complete.
- Once the network has been trained and the classifier has been initialized accordingly, the run time phase of prediction is quite fast and efficient.

Simple Convolutional Network Example

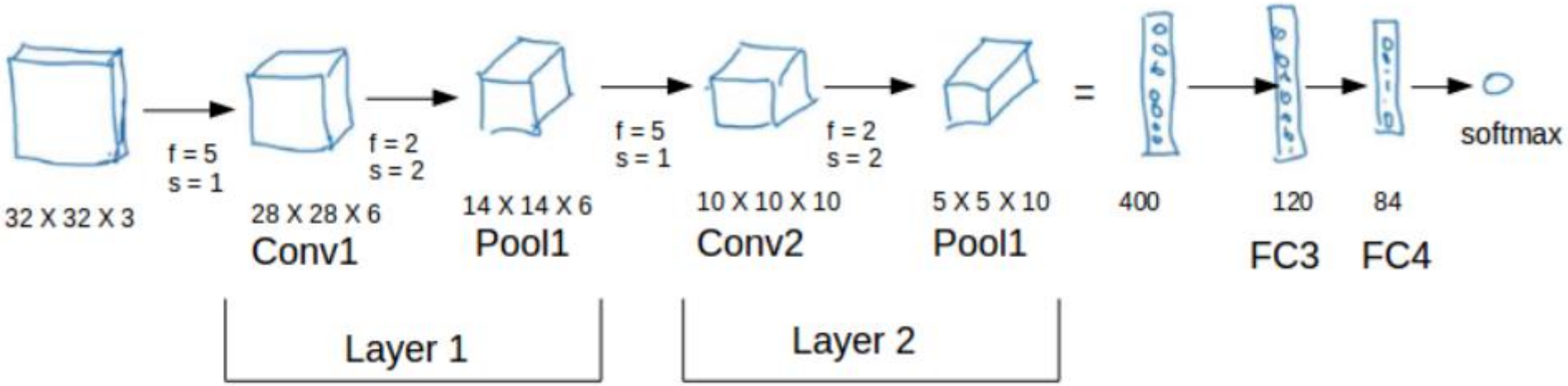


We take an input image (size = 39 X 39 X 3 in our case), convolve it with 10 filters of size 3 X 3, and take the stride as 1 and no padding. This will give us an output of 37 X 37 X 10. We convolve this output further and get an output of 7 X 7 X 40 as shown above. Finally, we take all these numbers (7 X 7 X 40 = 1960), unroll them into a large vector, and pass them to a classifier that will make predictions. This is a microcosm of how a convolutional network works.

There are a number of hyperparameters that we can tweak while building a convolutional network. These include the number of filters, size of filters, stride to be used, padding, etc.

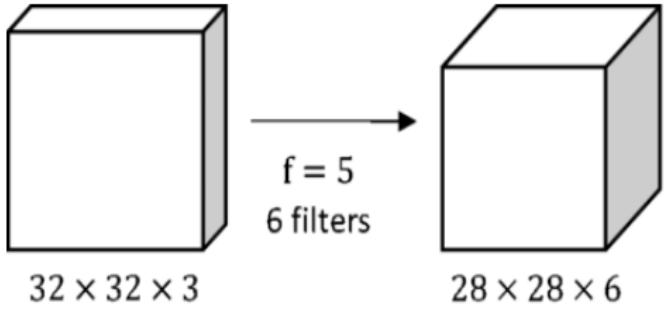
As we go deeper into the network, the size of the image shrinks whereas the number of channels usually increases.

CNN Example


f = 5
s = 1
f = 2
s = 2
f = 5
s = 1
f = 2
s = 2
=
softmax
32 X 32 X 3
28 X 28 X 6
Conv1
14 X 14 X 6
Pool1
10 X 10 X 10
Conv2
5 X 5 X 10
Pool1
400
120
FC3
84
FC4
Layer 1
Layer 2

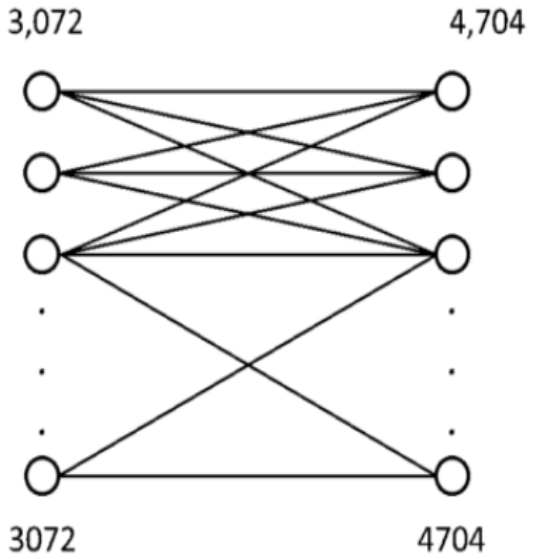
Why convolutions ?

- Let’s say that we have a 32×32×3 dimensional image.
- Let’s say we use 6, 5×5 filters (f=5), so this gives 28×28×6 dimensional output.



$5 \times 5 + 1 = 26$

$6 \times 26 = 156$ *parameters*



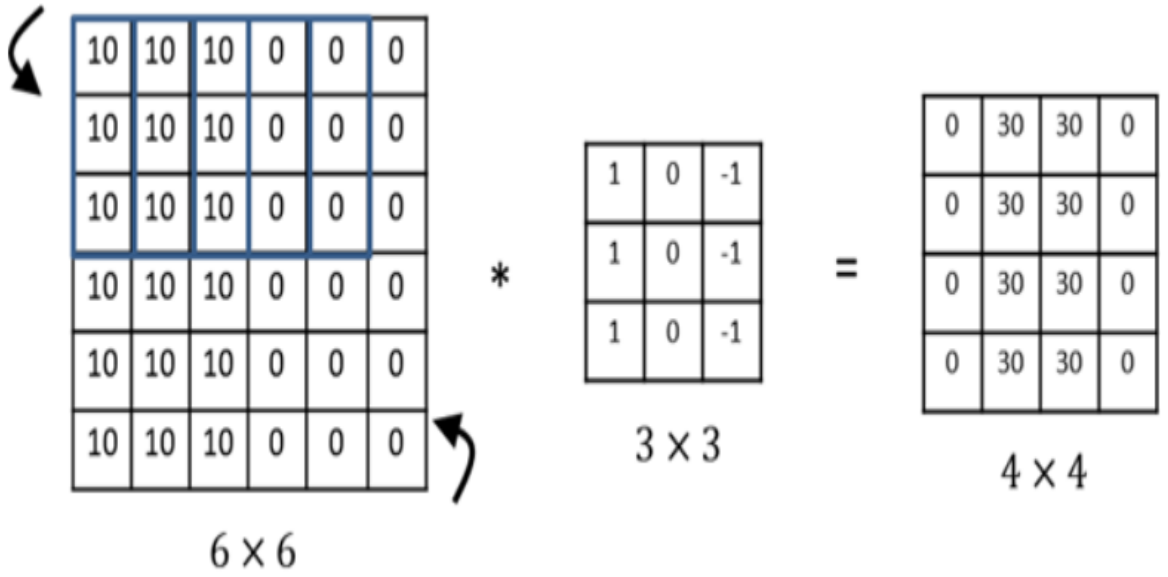
$3072 \times 4704 \approx 14M$

The main advantage of Conv layers over Fully connected layers is number of parameters to be learned

- The reasons that a Convnet has a small number of parameters is really two reasons:
  - **Parameter sharing**
  - **Sparsity of connections**

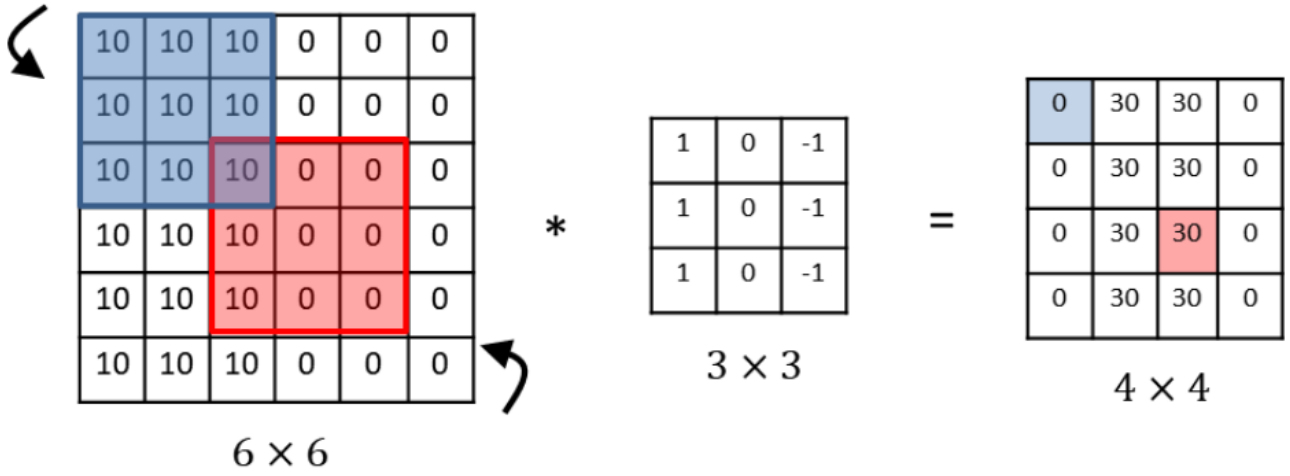
Parameter sharing

- A feature detector, for example a vertical edge detector, that's useful in one part of the image is probably useful in another part of the image.
- What this means is that if we apply a 3×3 filter for detecting vertical edges at one part of an image, we can then apply the same 3×3 filter at another position in the image .



Sparsity of connections

- Convnets are able to have relatively few parameters due to sparse connections.



If we look at the zero (blue-upper left): it was computed by a 3×3 convolution and it depends only on this 3×3 input grid of cells.
So, this output unit on the right is connected only to 9 out of these 36 (6×6) input features.
In particular, the rest of these pixel values, do not have any effect on that output.
As an another example, this red output depends only on 9 input features, and as if only those 9 input features are connected to this output.

One another reason why convolutional neural networks are so good in computer vision is:

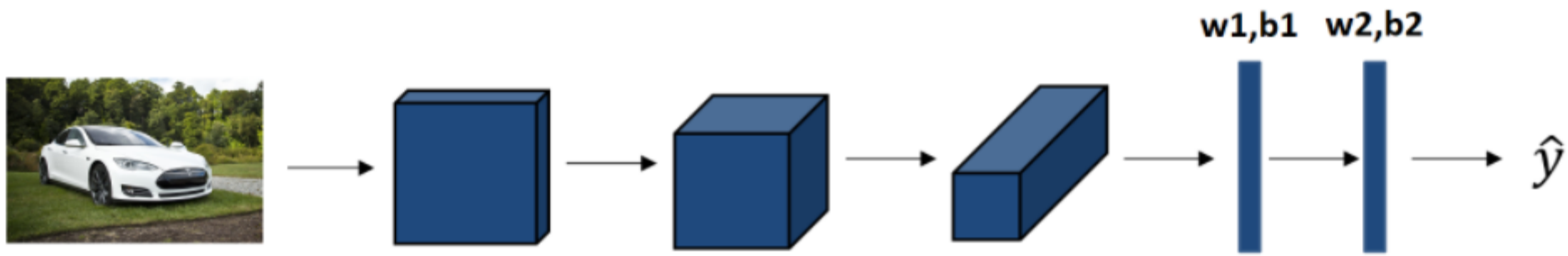
- **Translation invariance** :
- Convolutional neural networks are being very good at capturing translation invariance, and that means that a picture of a car shifted a couple pixels to the right is still a car.
- Also, convolutional structure helps neural network because the fact that an image shifted a few pixels should result in pretty similar features and should probably be assigned the same output label.
- The fact that we're applying the same filter to all the positions of the image both in the early layers and in the later layers, that helps a neural network to automatically learn to be a more robust, or to better capture this desirable property of translation invariance .

CNN are also..

- They are also known as shift invariant or space invariant artificial neural networks (SIANN), based on their shared-weights architecture and translation invariance characteristics.

Convolutional neural network (CNN, or ConvNet)

- A CNN consists of three main layers: convolution layer, pooling layer, and fully connected layer.
- Each of these layers does certain spatial operations.
- In convolution layers, CNN uses different kernels for convolving the input image for creating the feature maps.
- The pooling layer is usually inserted after a convolution layer.
- The application of this layer reduces the size of feature maps and network parameters.
- After the pooling layer, there is a flatten layer followed by some fully connected layers.
- In the flatten layer, 2D feature maps produced in the previous layer are converted into 1D feature maps to be suitable for the following fully connected layers.
- The flattened vector can be used later for the classification of the images.


w1,b1
w2,b2
$\hat{y}$

Variants of the Basic Convolution Function

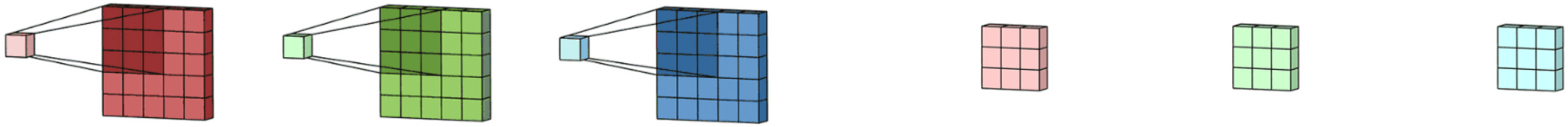
- First, when we refer to convolution in the context of neural networks, we usually actually mean an operation that consists of many applications of convolution in parallel.
- This is because convolution with a single kernel can only extract one kind of feature, albeit at many spatial locations.
- Usually we want each layer of our network to extract many kinds of features, at many locations.

- Additionally, the input is usually not just a grid of real values. Rather, it is a grid of vector-valued observations.
- For example, a color image has a red, green and blue intensity at each pixel.
- In a multilayer convolutional network, the input to the second layer is the output of the first layer, which usually has the output of many different convolutions at each position.
- When working with images, we usually think of the input and output of the convolution as being 3-D tensors, with one index into the different channels and two indices into the spatial coordinates of each channel.
- Software implementations usually work in batch mode, so they will actually use 4-D tensors, with the fourth axis indexing different examples in the batch, but we will omit the batch axis in our description here for simplicity.

1. Convolution: the single channel version

2. Convolution: the multi-channel version

Original image (RGB) R channel G channel B channel



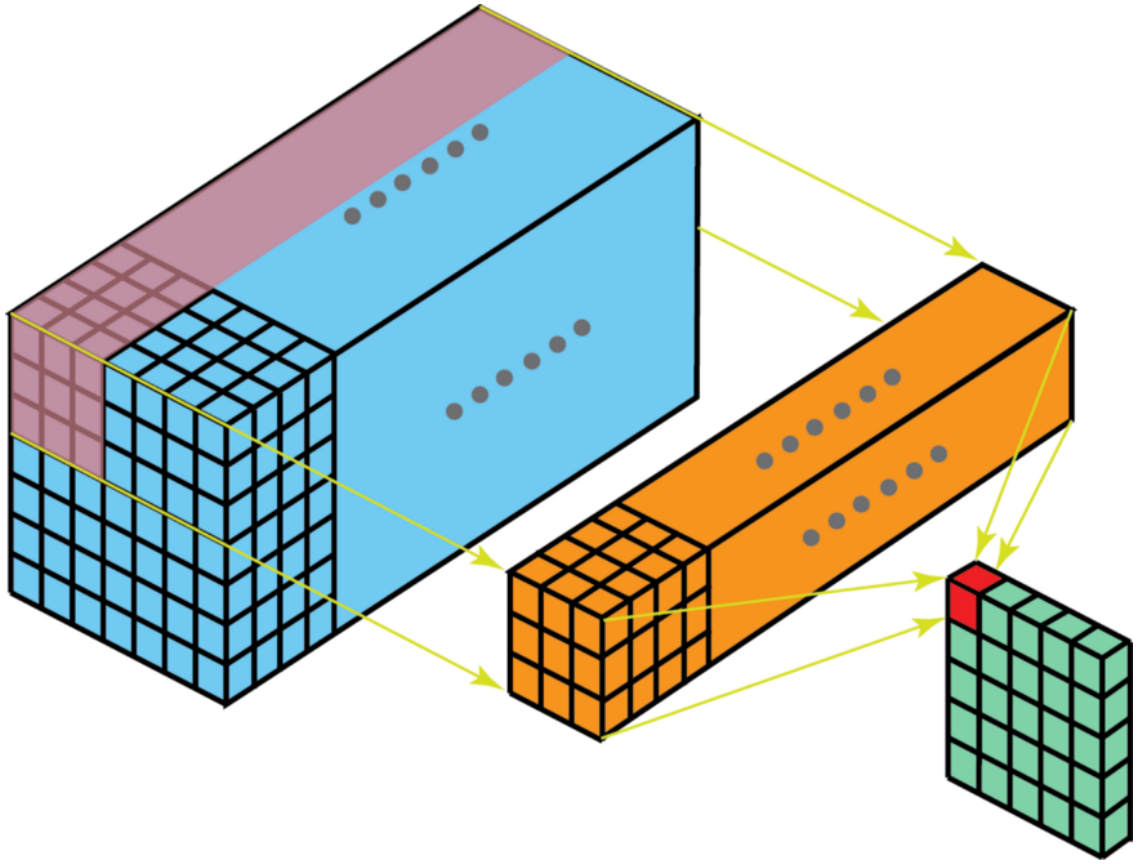
Sliding a 3D filter matrix through the input layer.

Notice that the input layer and the filter have the same depth (channel number = kernel number).

The 3D filter moves only in 2-direction, height & width of the image (That's why such operation is called as 2D convolution although a 3D filter is used to process 3D volumetric data).

At each sliding position, we perform element-wise multiplication and addition, which results in a single number.

In the example shown below, the sliding is performed at 5 positions horizontally and 5 positions vertically. Overall, we get a single output channel.

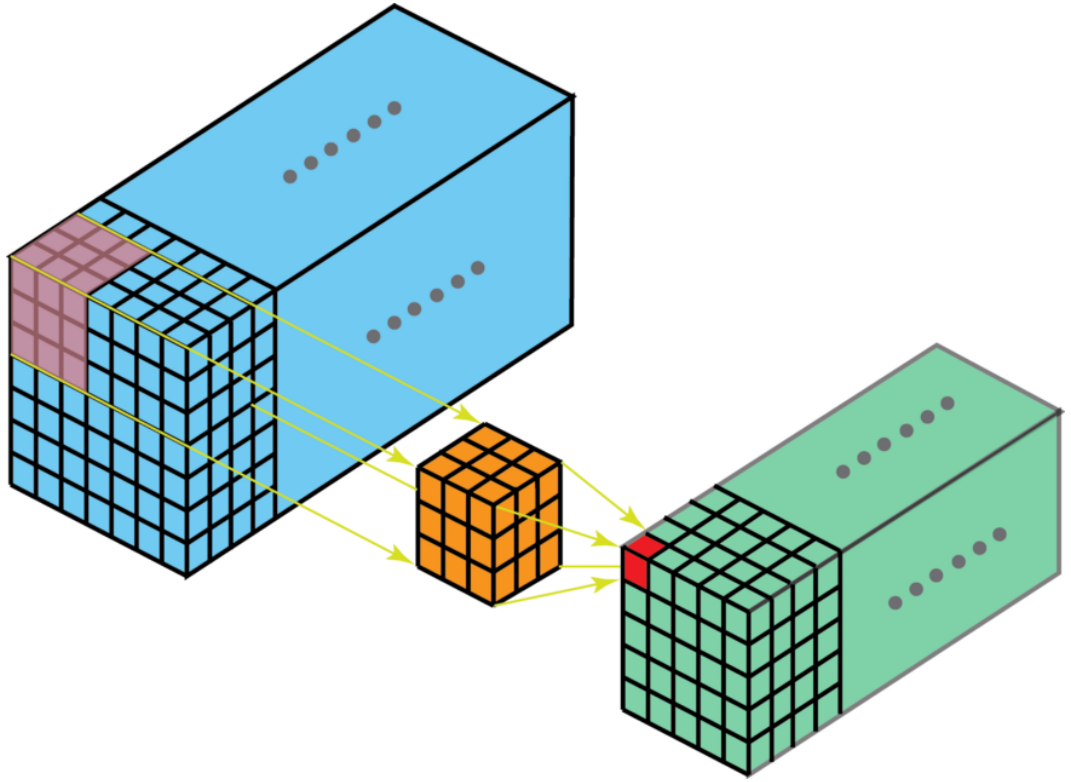


3. 3D Convolution

- *Here in 3D convolution, the filter depth is smaller than the input layer depth (kernel size < channel size).*

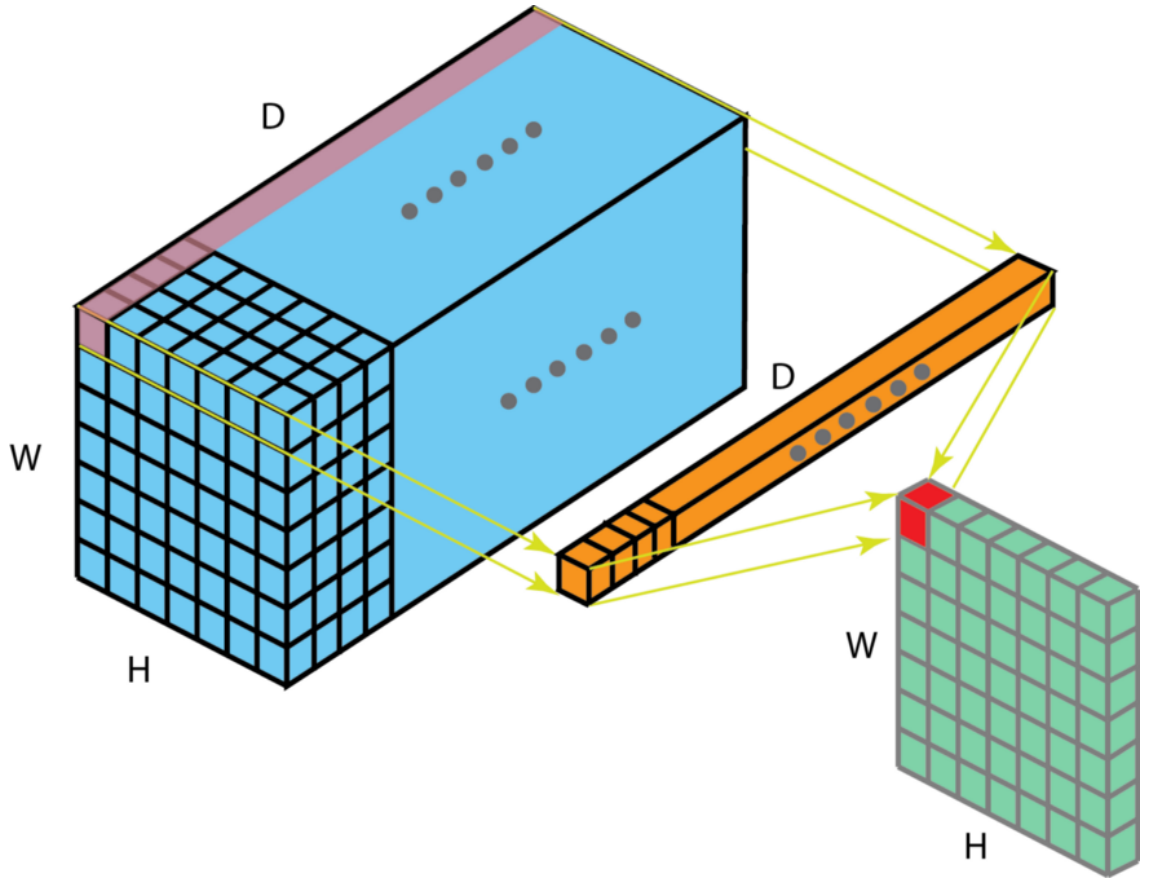
- *As a result, the 3D filter can move in all 3-direction (height, width, channel of the image)*. At each position, the element-wise multiplication and addition provide one number.

- Since the filter slides through a 3D space, the *output numbers are arranged in a 3D space as well. The output is then a 3D data.*



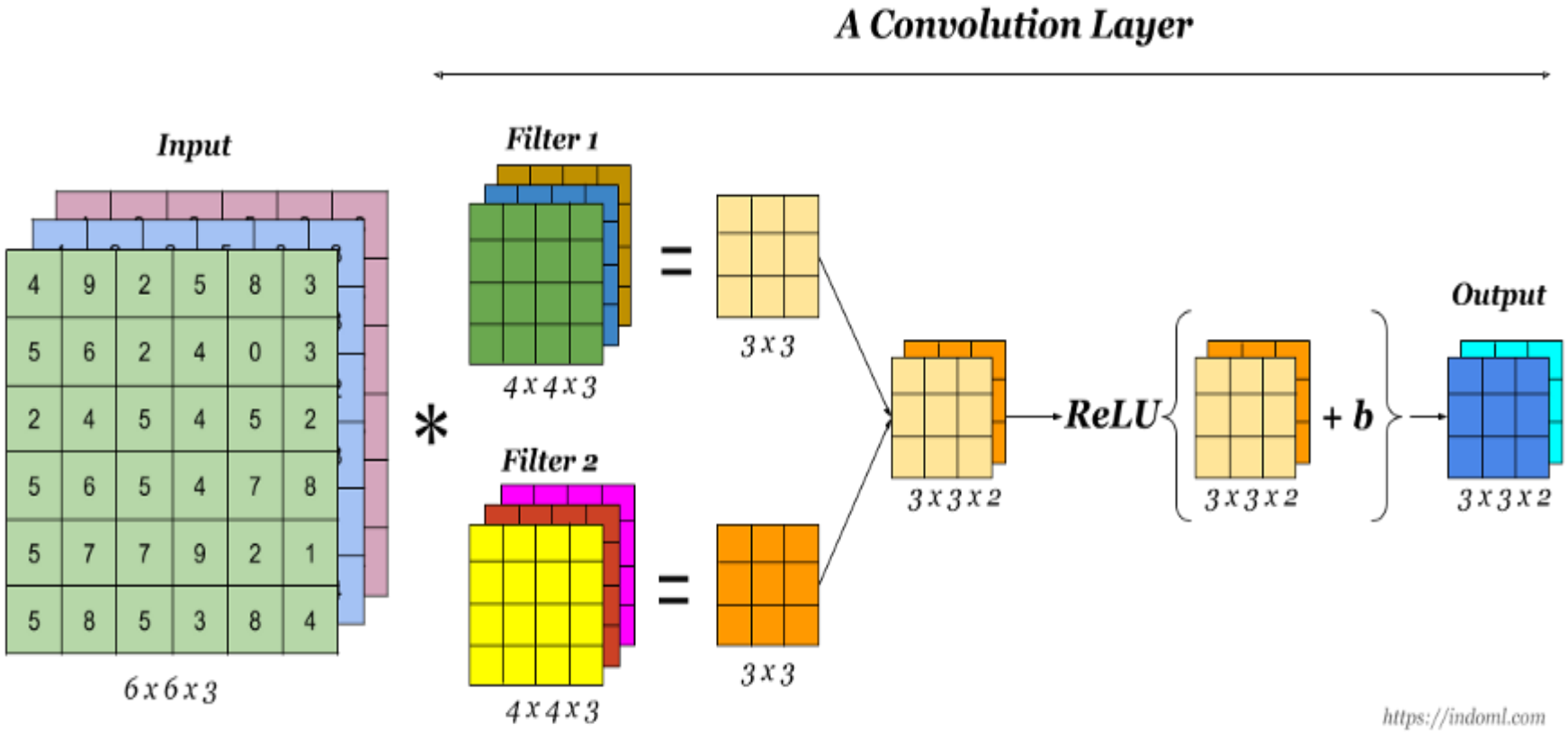
4. 1 x 1 Convolution

- The following picture illustrates how 1 x 1 convolution works for an input layer with dimension H x W x D.
- After 1 x 1 convolution with filter size 1 x 1 x D, the output channel is with dimension H x W x 1.
- If we apply N such 1 x 1 convolutions and then concatenate results together, we could have a output layer with dimension H x W x N.



5. Convolution Arithmetic

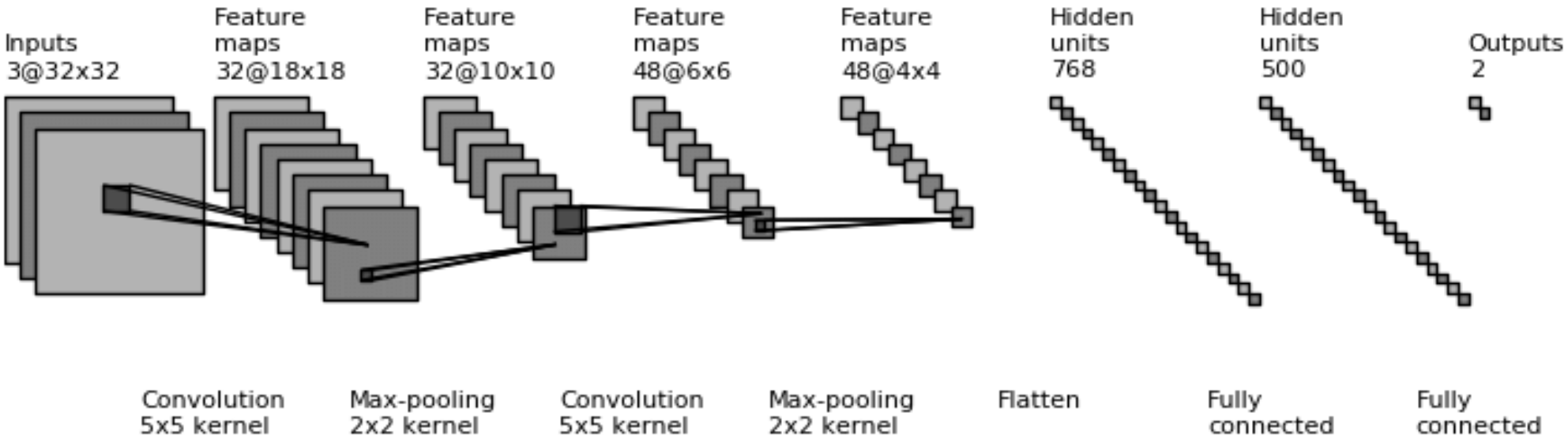
- Here are a few terminologies:
- Kernel size: The kernel size defines the field of view of the convolution.
- Stride: it defines the step size of the kernel when sliding through the image. Stride of 1 means that the kernel slides through the image pixel by pixel. Stride of 2 means that the kernel slides through image by moving 2 pixels per step (i.e., skipping 1 pixel). We can use stride (>= 2) for down sampling an image.
- Padding: the padding defines how the border of an image is handled. A padded convolution ('same' padding in Tensorflow) will keep the spatial output dimensions equal to the input image, by padding 0 around the input boundaries if necessary. On the other hand, unpadded convolution ('valid' padding in Tensorflow) only perform convolution on the pixels of the input image, without adding 0 around the input boundaries. The output size is smaller than the input size.



An activation function is the last component of the convolutional layer to increase the non-linearity in the output. Generally, ReLu function or Tanh function is used as an activation function in a convolution layer. Here is an image of a simple convolution layer, where a 6X6X3 input image is convolved with two kernels of size 4X4X3 to get a convolved feature of size 3X3X2, to which activation function is applied to get the output, which is also referred to as feature map.

Fully Connected Layer

A fully connected layer is at the end of a convolutional neural network.
The features map produced by the earlier layer is flattened to a vector.
Then this vector is fed to a fully connected layer so that it captures complex relationships between high-level features.
The output of this layer is a one-dimensional feature vector.



Building a CNN for CIFAR-10

- CIFAR-10 Photo Classification Dataset
- CIFAR is an acronym that stands for the Canadian Institute For Advanced Research and the CIFAR-10 dataset was developed along with the CIFAR-100 dataset by researchers at the CIFAR institute.
- The dataset is comprised of 60,000 32×32 pixel color photographs of objects from 10 classes, such as frogs, birds, cats, ships, etc.

CIFAR-10 Photo Classification Dataset

- The class labels and their standard associated integer values are listed below.

- 0: airplane
- 1: automobile
- 2: bird
- 3: cat
- 4: deer
- 5: dog
- 6: frog
- 7: horse
- 8: ship
- 9: truck
- These are very small images, much smaller than a typical photograph, and the dataset was intended for computer vision research.

```
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
from tensorflow.keras.losses import sparse_categorical_crossentropy
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
```

```
# Model configuration
batch_size = 50
img_width, img_height, img_num_channels = 32, 32, 3
loss_function = sparse_categorical_crossentropy
no_classes = 10
no_epochs = 100
optimizer = Adam()
validation_split = 0.2
verbosity = 1
```

```
# Load CIFAR-10 data
(input_train, target_train), (input_test, target_test) = cifar10.load_data()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [==============================] - 11s 0us/step
170508288/170498071 [==============================] - 11s 0us/step
```

```
# Determine shape of the data
input_shape = (img_width, img_height, img_num_channels)
```

```
# Parse numbers as floats
input_train = input_train.astype('float32')
input_test = input_test.astype('float32')
```

```
# Normalize data
input_train = input_train / 255
input_test = input_test / 255
```

```
# Create the model
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(128, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dense(no_classes, activation='softmax'))
```

```
# Compile the model
model.compile(loss=loss_function,
              optimizer=optimizer,
              metrics=['accuracy'])
```

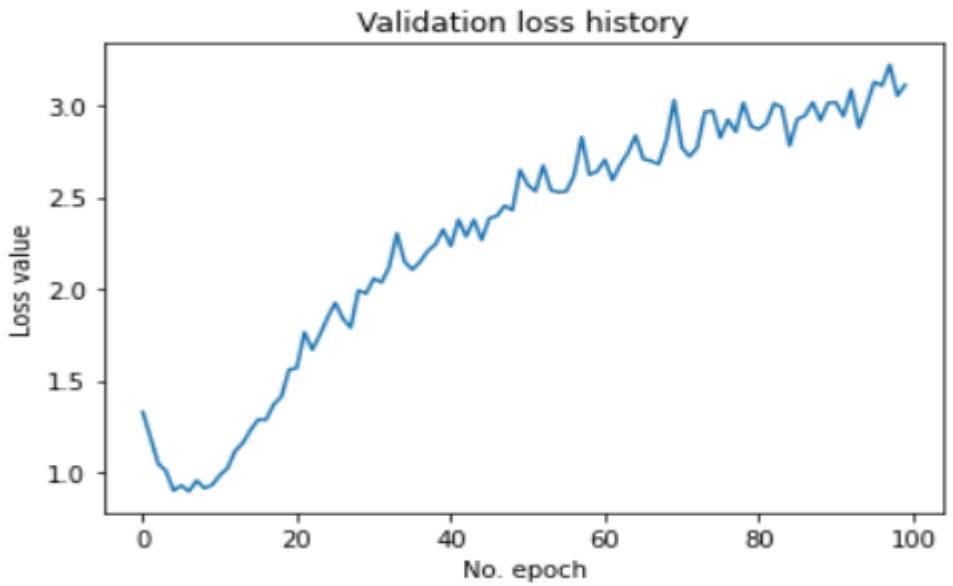
```
# Fit data to model
history = model.fit(input_train, target_train,
            batch_size=batch_size,
            epochs=no_epochs,
            verbose=verbosity,
            validation_split=validation_split)
```

```
Epoch 1/100
800/800 [==============================] - 60s 74ms/step - loss: 1.6191 - accuracy: 0.4026 - val_loss: 1.3279 - val_accuracy: 0.5206
Epoch 2/100
800/800 [==============================] - 59s 74ms/step - loss: 1.2245 - accuracy: 0.5590 - val_loss: 1.1888 - val_accuracy: 0.5783
Epoch 3/100
800/800 [==============================] - 59s 74ms/step - loss: 1.0444 - accuracy: 0.6325 - val_loss: 1.0474 - val_accuracy: 0.6329
Epoch 4/100
800/800 [==============================] - 59s 74ms/step - loss: 0.9215 - accuracy: 0.6765 - val_loss: 1.0070 - val_accuracy: 0.6486
Epoch 5/100
800/800 [==============================] - 59s 74ms/step - loss: 0.8228 - accuracy: 0.7092 - val_loss: 0.9004 - val_accuracy: 0.6929
Epoch 6/100
800/800 [==============================] - 59s 74ms/step - loss: 0.7435 - accuracy: 0.7387 - val_loss: 0.9282 - val_accuracy: 0.6804
Epoch 7/100
800/800 [==============================] - 59s 74ms/step - loss: 0.6718 - accuracy: 0.7660 - val_loss: 0.8961 - val_accuracy: 0.6962
Epoch 8/100
800/800 [==============================] - 59s 73ms/step - loss: 0.6127 - accuracy: 0.7851 - val_loss: 0.9553 - val_accuracy: 0.6894
Epoch 9/100
800/800 [==============================] - 59s 73ms/step - loss: 0.5525 - accuracy: 0.8039 - val_loss: 0.9126 - val_accuracy: 0.7020
Epoch 10/100
800/800 [==============================] - 59s 73ms/step - loss: 0.4979 - accuracy: 0.8246 - val_loss: 0.9309 - val_accuracy: 0.7064
Epoch 11/100
800/800 [==============================] - 59s 73ms/step - loss: 0.4495 - accuracy: 0.8398 - val_loss: 0.9834 - val_accuracy: 0.7046
Epoch 12/100
800/800 [==============================] - 59s 73ms/step - loss: 0.3950 - accuracy: 0.8601 - val_loss: 1.0222 - val_accuracy: 0.7147
Epoch 13/100
```

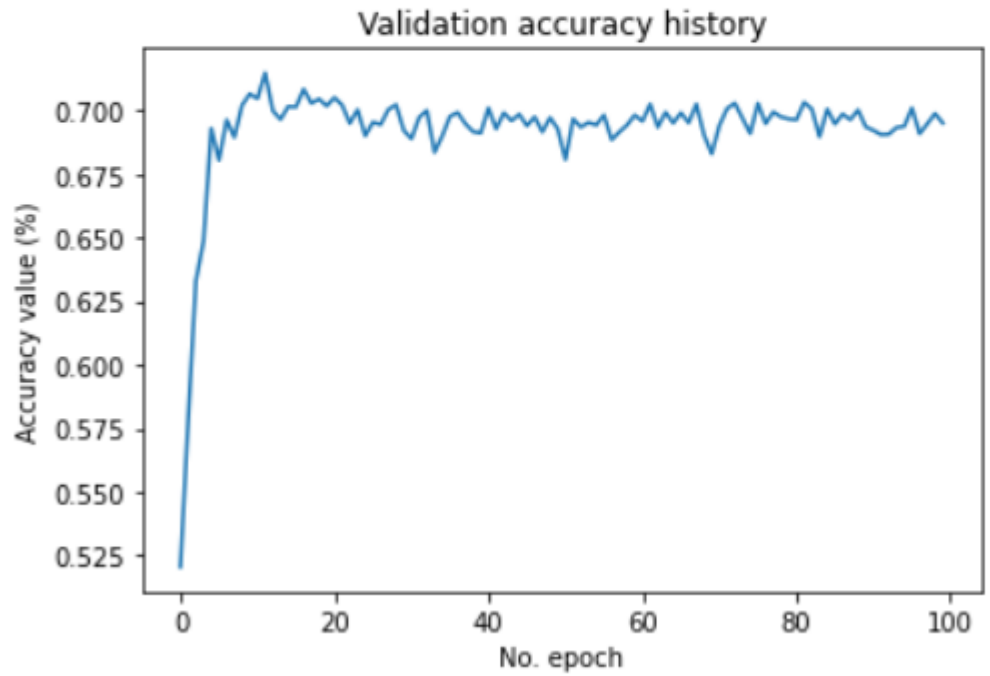
```
# Generate generalization metrics
score = model.evaluate(input_test, target_test, verbose=0)
print(f'Test loss: {score[0]} / Test accuracy: {score[1]}')
```

```
Test loss: 3.1887290477752686 / Test accuracy: 0.6827999949455261
```

```
# Visualize history
# Plot history: Loss
plt.plot(history.history['val_loss'])
plt.title('Validation loss history')
plt.ylabel('Loss value')
plt.xlabel('No. epoch')
plt.show()
```



```
# Plot history: Accuracy
plt.plot(history.history['val_accuracy'])
plt.title('Validation accuracy history')
plt.ylabel('Accuracy value (%)')
plt.xlabel('No. epoch')
plt.show()
```



Visualizing learning in Convolutional Neural Networks

- Deep learning neural networks are generally opaque, meaning that although they can make useful and skillful predictions, it is not clear how or why a given prediction was made.
- Convolutional neural networks, have internal structures that are designed to operate upon two-dimensional image data, and as such preserve the spatial relationships for what was learned by the model.
- Specifically, the two-dimensional filters learned by the model can be inspected and visualized to discover the types of features that the model will detect, and the activation maps output by convolutional layers can be inspected to understand exactly what features were detected for a given input image.

How to Visualize Filters

- The simplest visualization to perform is to plot the learned filters directly.
- The learned filters are simply weights.
- The weight values have a spatial relationship

Why the number of filters increases with the depth of CNN?

- Every layer of filters is there to capture patterns.
- For example, the first layer of filters captures patterns like edges, corners, dots etc.
- Subsequent layers combine those patterns to make bigger patterns (like combining edges to make squares, circles, etc.).
- Now as we move forward in the layers, the patterns get more complex; **hence there are larger combinations of patterns to capture**.
- That's why we increase the filter size in subsequent layers to capture as many combinations as possible.

Prints a list of layer details including the layer name and the shape of the filters in the layer.

```
model=models.Sequential()
model.add(layers.BatchNormalization())
model.add(layers.Conv2D(32,(3,3),activation='relu',input_shape=(64,64,3)))
#model.add(layers.Conv2D(32,(3,3),activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D(2,2))
model.add(layers.Conv2D(64,(3,3),activation='relu'))
model.add(layers.BatchNormalization())
#model.add(layers.Conv2D(64,(3,3),activation='relu'))
model.add(layers.MaxPooling2D(2,2))
model.add(layers.Conv2D(128,(3,3),activation='relu'))
model.add(layers.BatchNormalization())
#model.add(layers.Conv2D(128,(3,3),activation='relu'))
model.add(layers.MaxPooling2D(2,2))
model.add(layers.Flatten())
model.add(layers.Dense(512,activation='relu'))
model.add(layers.Dense(256,activation="relu"))
model.add(layers.Dense(128,activation="relu"))
model.add(layers.Dense(2,activation="softmax"))
```

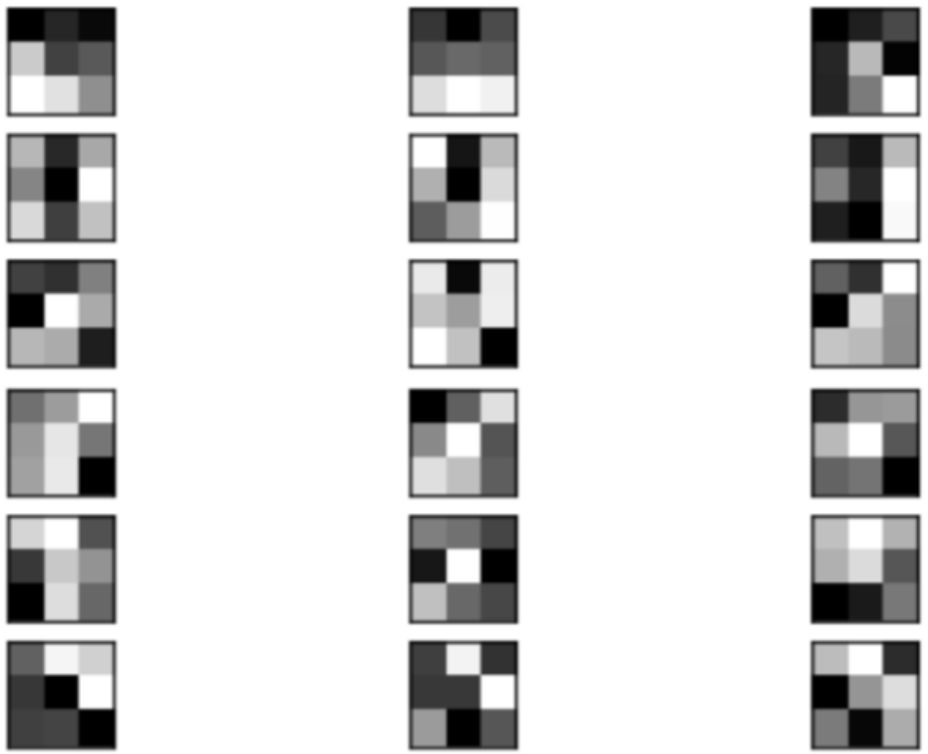
```
for layer in model.layers:
  # check for convolutional layer
  if 'conv' not in layer.name:
    continue
  # get filter weights
  filters, biases = layer.get_weights()
  print(layer.name, filters.shape)
```

```
conv2d (3, 3, 3, 32)
conv2d_1 (3, 3, 32, 64)
conv2d_2 (3, 3, 64, 128)
```

The weight values will likely be small positive and negative values centered around 0.0.
We can normalize their values to the range 0-1 to make them easy to visualize.
Now we can enumerate the first six filters out of the 32 in the block and plot each of the three channels of each filter.

```
# normalize filter values to 0-1 so we can visualize them
f_min, f_max = filters.min(), filters.max()
filters = (filters - f_min) / (f_max - f_min)
```

```
from matplotlib import pyplot
# plot first few filters
n_filters, ix = 6, 1
for i in range(n_filters):
  # get the filter
  f = filters[:, :, :, i]
  # plot each channel separately
  for j in range(3):
    # specify subplot and turn of axis
    ax = pyplot.subplot(n_filters, 3, ix)
    ax.set_xticks([])
    ax.set_yticks([])
    # plot filter channel in grayscale
    pyplot.imshow(f[:, :, j], cmap='gray')
    ix += 1
# show the figure
pyplot.show()
```



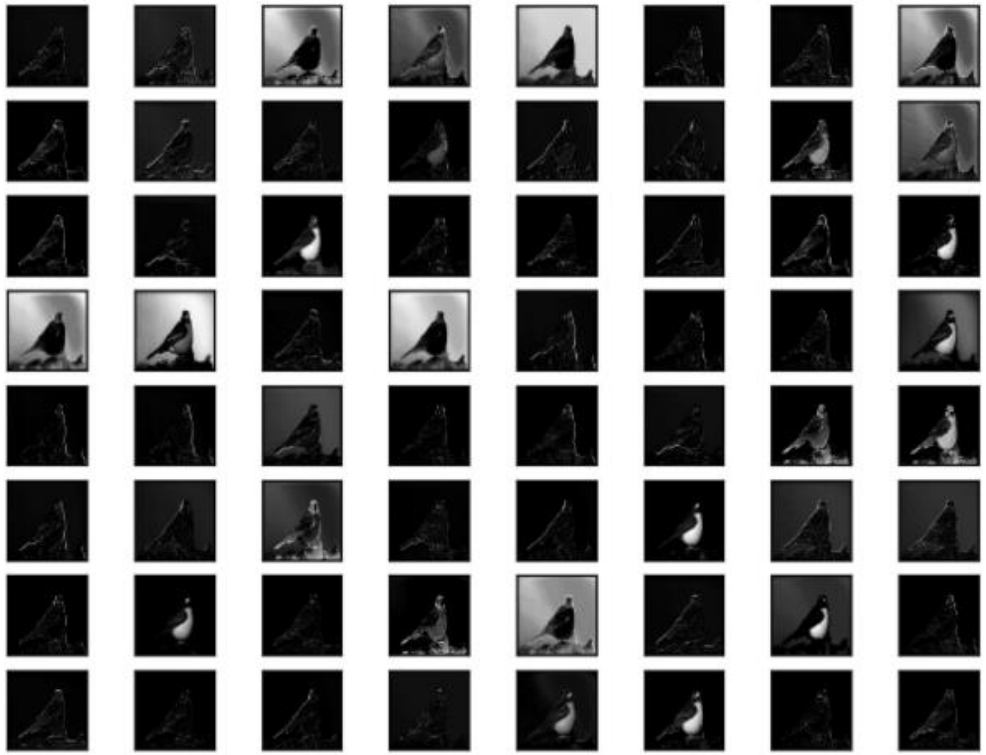
one row for each filter and one column for each channel

How to Visualize Feature Maps

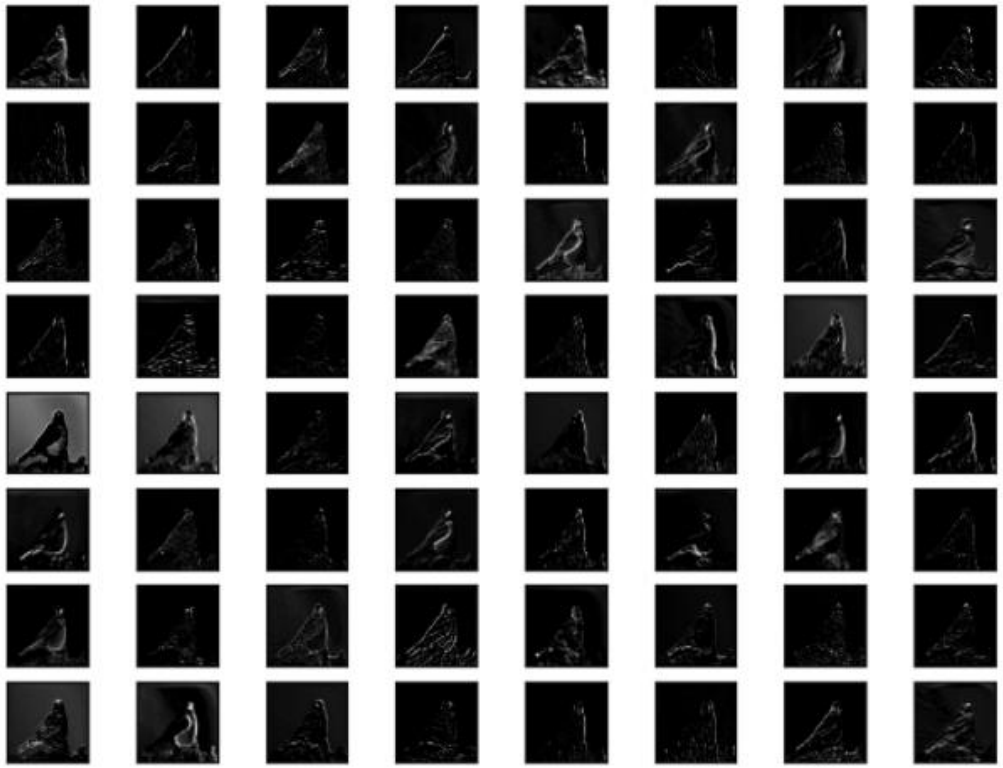
The activation maps, called feature maps, capture the result of applying the filters to input, such as the input image or another feature map.

The idea of visualizing a feature map for a specific input image would be to understand what features of the input are detected or preserved in the feature maps.

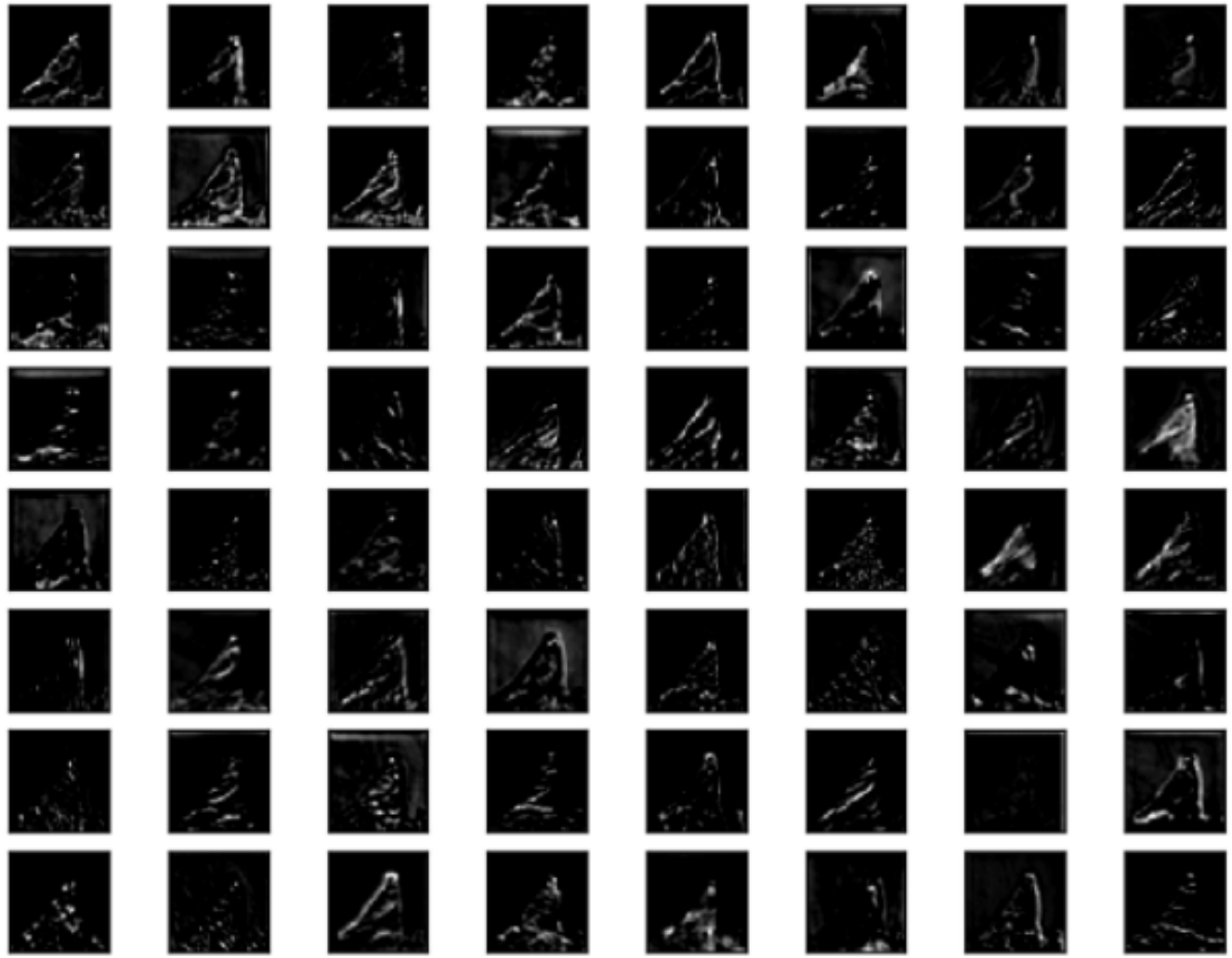
The expectation would be that the feature maps close to the input detect small or fine-grained detail, whereas feature maps close to the output of the model capture more general features.



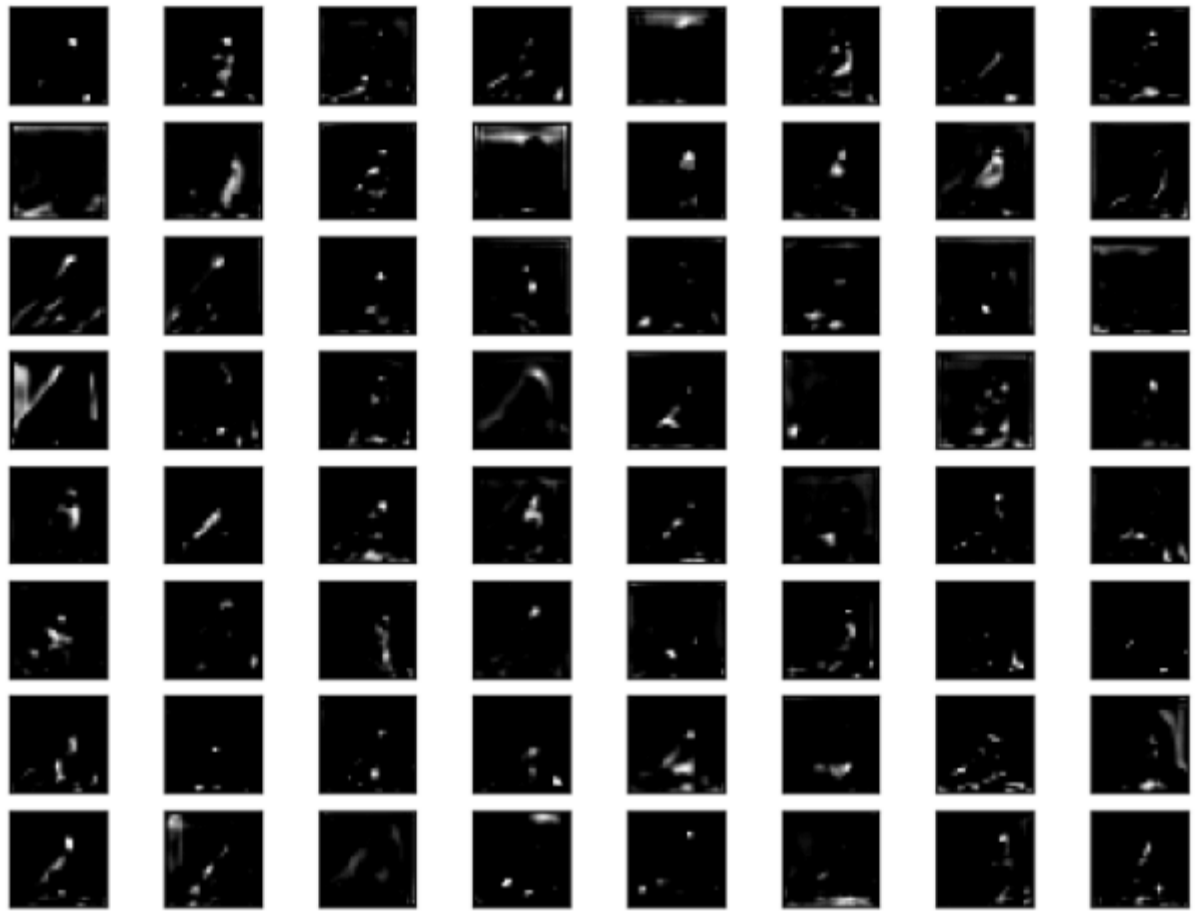
Visualization of the Feature Maps Extracted From Block 1 in the VGG16 Model



Visualization of the Feature Maps Extracted From Block 2 in the VGG16 Model



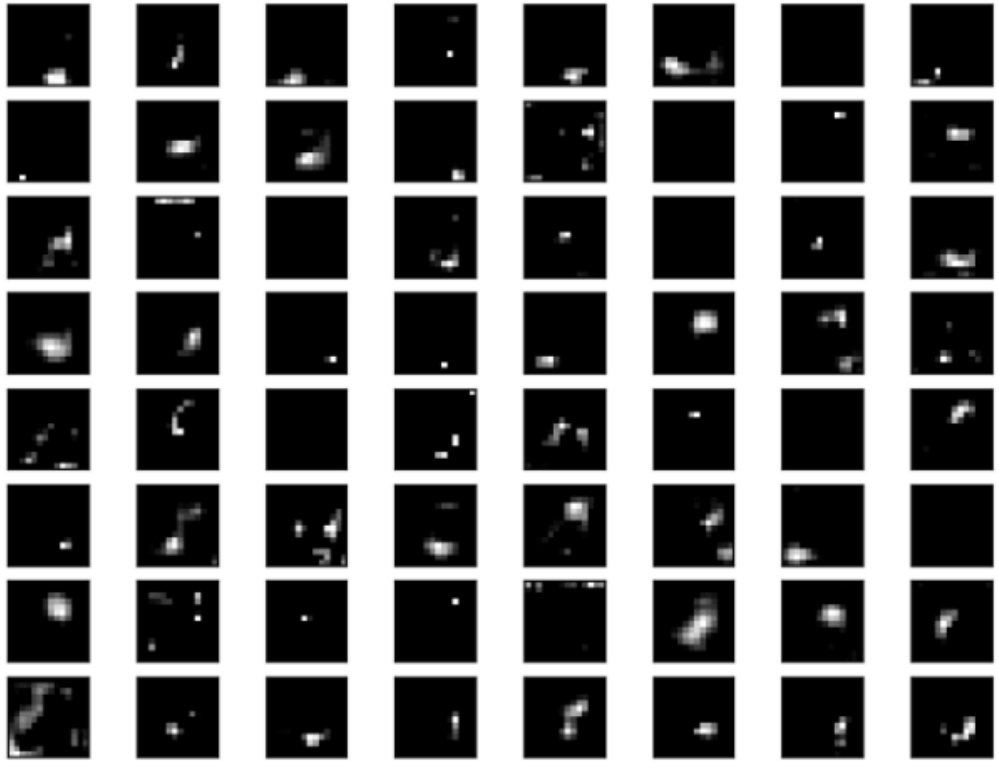
Visualization of the Feature Maps Extracted From Block 3 in the VGG16 Model



Visualization of the Feature Maps Extracted From Block 4 in the VGG16 Model

We can see that the feature maps closer to the input of the model capture a lot of fine detail in the image and that as we progress deeper into the model, the feature maps show less and less detail.

Visualization of the Feature Maps Extracted From Block 1 in the VGG16 Model



Visualization of the Feature Maps Extracted From Block 5 in the VGG16 Model

The model abstracts the features from the image into more general concepts that can be used to make a classification.

State-of-the-art deep learning image classifiers in Keras

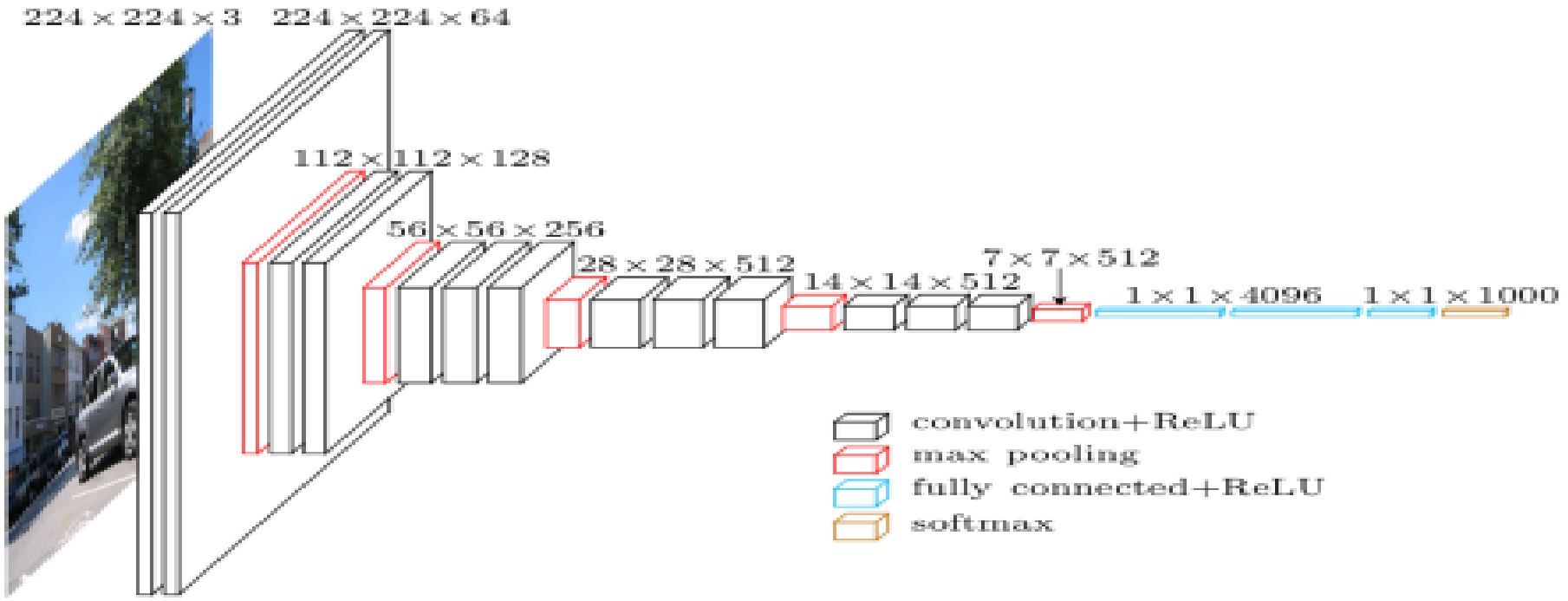
- Keras ships out-of-the-box with five Convolutional Neural Networks that have been pre-trained on the ImageNet dataset:

- VGG16
- VGG19
- ResNet50
- Inception V3
- Xception

What is ImageNet?

- ImageNet is formally a project aimed at (manually) labeling and categorizing images into almost 22,000 separate object categories for the purpose of computer vision research.
- However, when we hear the term "ImageNet" in the context of deep learning and Convolutional Neural Networks, we are likely referring to the ImageNet Large Scale Visual Recognition Challenge, or ILSVRC for short.
- The goal of this image classification challenge is to train a model that can correctly classify an input image into 1,000 separate object categories.
- Models are trained on ~1.2 million training images with another 50,000 images for validation and 100,000 images for testing.
- These 1,000 image categories represent object classes that we encounter in our day-to-day lives, such as species of dogs, cats, various household objects, vehicle types, and much more

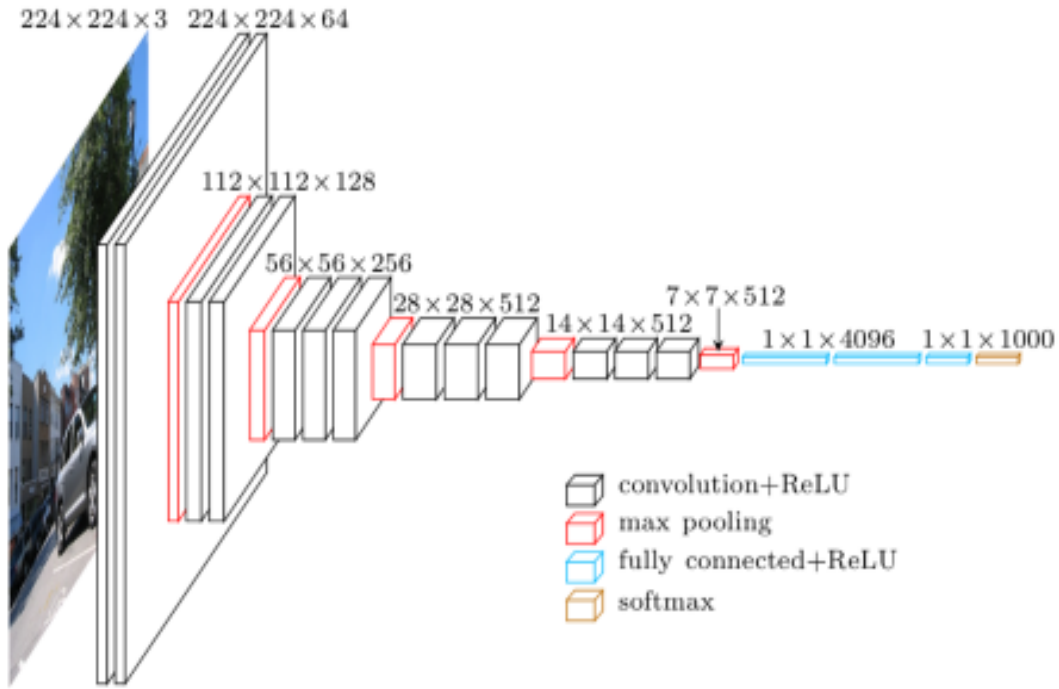
VGG16



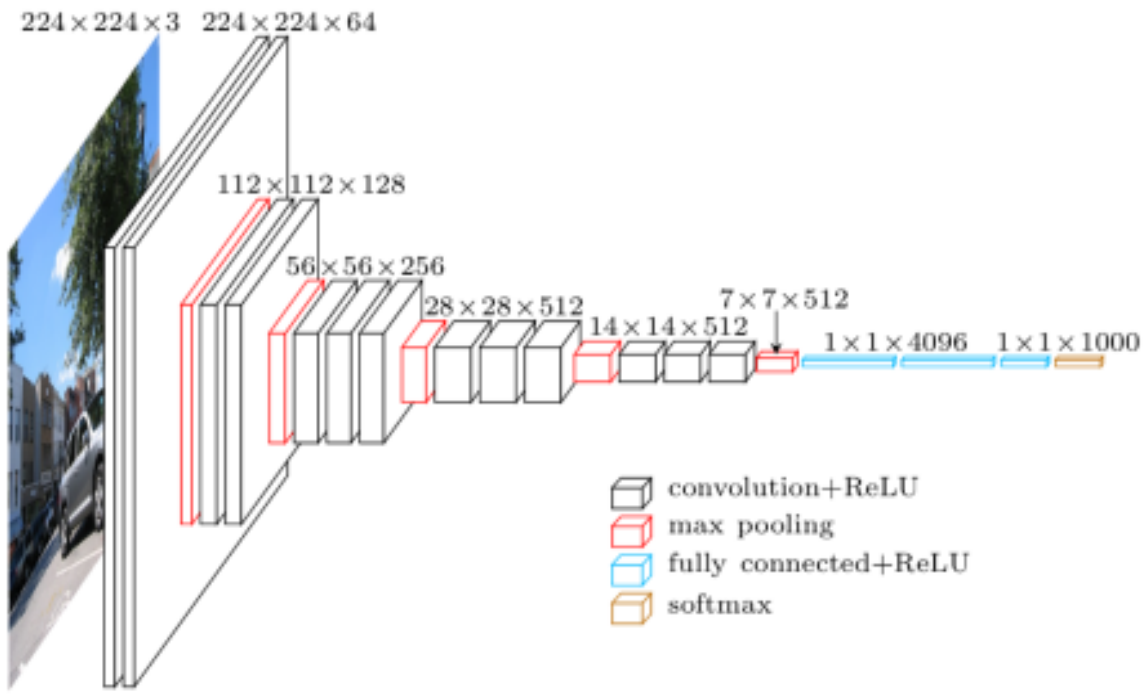
convolution layers of 3x3 filter with a stride 1 and always used same padding and maxpool layer of 2x2 filter of stride 2.

The 16 in VGG16 refers to it has 16 layers that have weights. This network is a pretty large network and it has about 138 million (approx) parameters.

```
model = Sequential()

model.add(Conv2D(input_shape=(224,224,3),filters=64,kernel_size=
(3,3),padding="same", activation="relu"))

model.add(Conv2D(filters=64,kernel_size=(3,3),padding="same",
activation="relu"))

model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))

model.add(Conv2D(filters=128, kernel_size=(3,3), padding="same",
activation="relu"))

model.add(Conv2D(filters=128, kernel_size=(3,3), padding="same",
activation="relu"))

model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))

model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same",
activation="relu"))

model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same",
activation="relu"))

model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same",
activation="relu"))
```



```
model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))

model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same",
activation="relu"))

model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same",
activation="relu"))

model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same",
activation="relu"))

model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))

model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same",
activation="relu"))

model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same",
activation="relu"))

model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same",
activation="relu"))

model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))
```



```
model.add(Flatten())

model.add(Dense(units=4096,activation="relu"))

model.add(Dense(units=4096,activation="relu"))

model.add(Dense(units=2, activation="softmax"))
```