Lecture 04

Convolutional Neural Networks (CNNs)

csci e-89 Deep Learning, Fall 2025

Zoran B. Djordjević & Rahul Joglekar

References

Most of the material in this lecture follows:

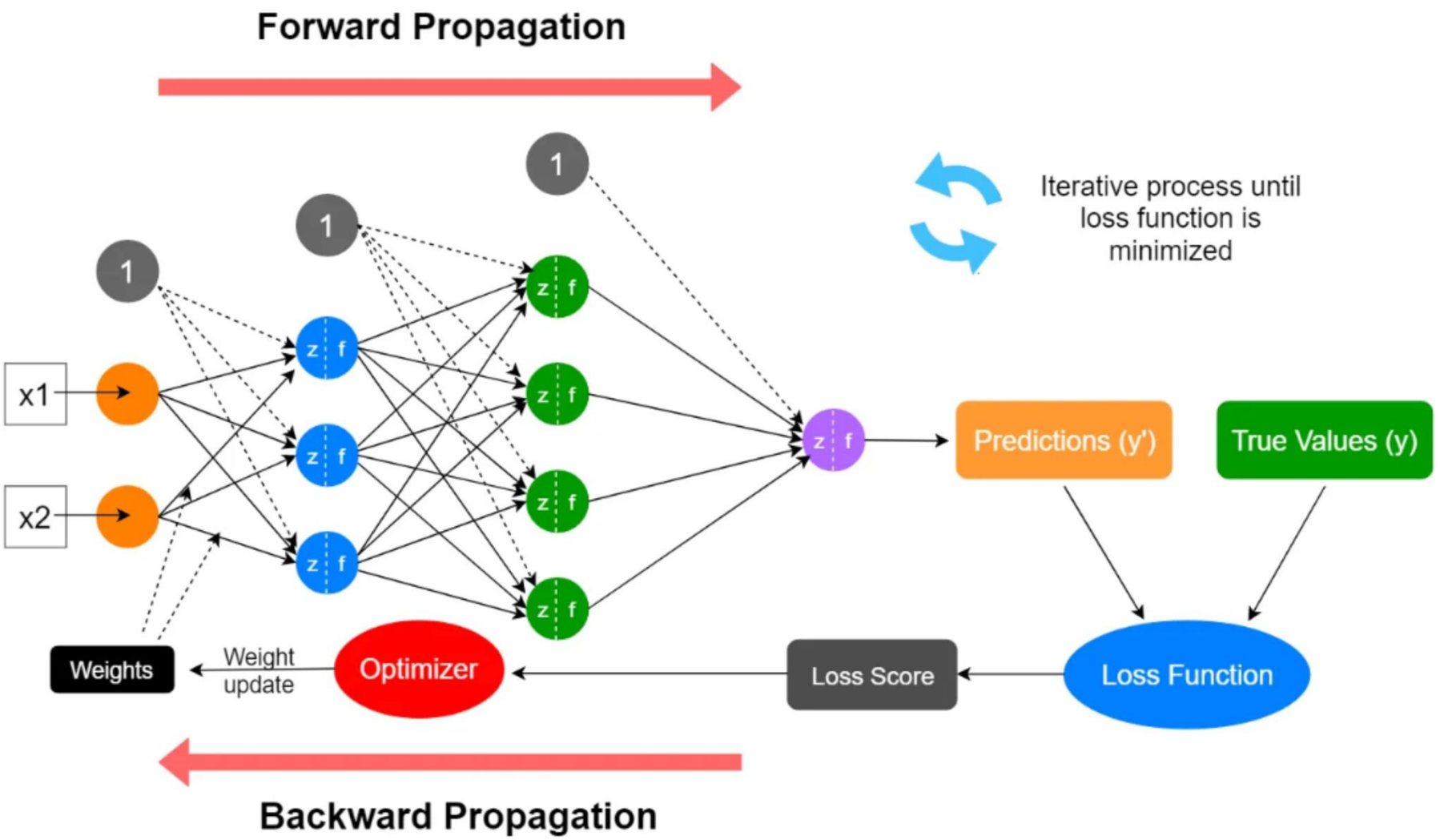
- Chapter 5 of “Deep Learning in Python” 1st Ed. by Francois Chollet, Manning 2017
  Chapter 8 of “Deep Learning in Python” 2nd Ed. by Francois Chollet, Manning 2021
- Chapters 14 of “Hands-on Machine Learning with Sciki-Learn, Keras & TensorFlow” in 2nd & 3rd Edition, by Aurelien Geron, O’Reilly 2019 and 2022

Objectives

Objectives of today's lecture:

- ✓ Introduce CNNs – Image processing
  - ✓ Why CNNs and Motivation for CNNs
- ✓ Components of Convolutional Networks
- ✓ The Convolution Function
- ✓ Filters and Feature Extraction
- ✓ Activation functions
- ✓ Padding
- ✓ Pooling
- ✓ Spatial arrangements
- ✓ Callbacks

Building Blocks of NN



Triva

Quick check about our knowledge of Neural Networks

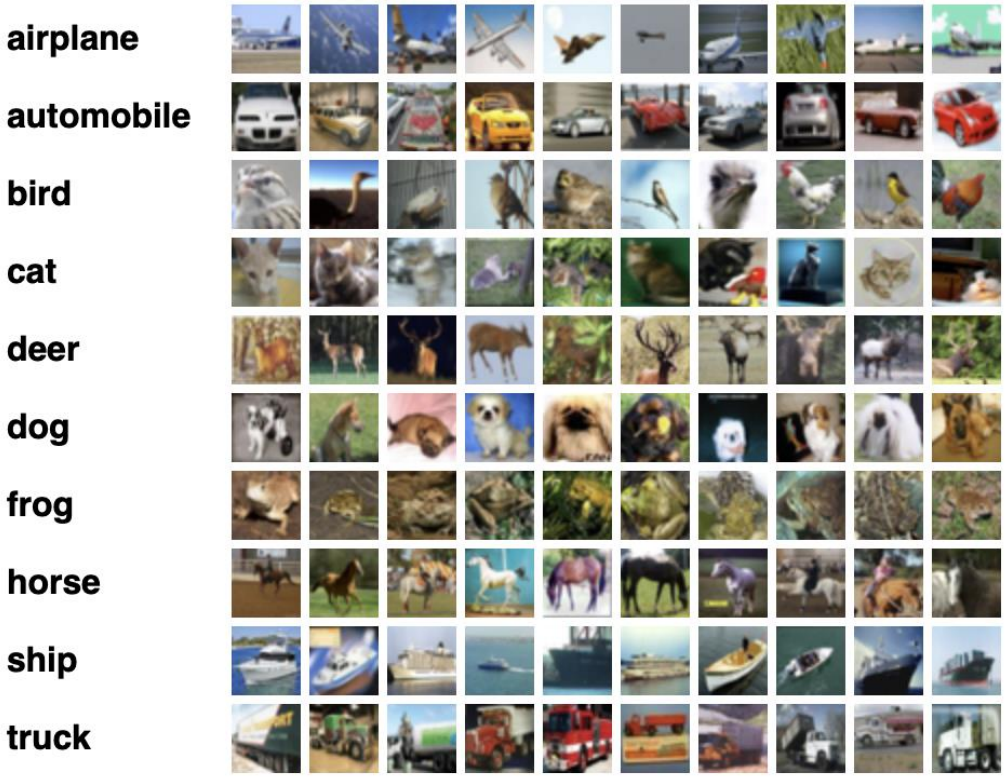
a) What is the difference between Feed Forward Neural Network and a Fully Connected Neural Network?
b) For a Fully Connected Network, the learning happens in the forward pass where weights are updated (True/False)
c) The Job of a “Loss function” is to is to tell the difference between ___ & ___
d) RELU is an example of a loss function (True/False)
e) For a FCN all neurons from one layer are connected to other subsequent layer and neurons within the layer are also connected to each other (True/False)
f) The Optimizer function activates every neuron when certain weights are passed
g) Softmax and Sigmoid are examples of what type of function ?
h) For a regression problem we would use Sigmoid function in final layer ?
i) One epoch means we have run a mini batch through the network (True/False)
j) The regularization techniques are applied for only certain neurons in the network (True/False)

Structure of Neural Networks

- So far, we have shown a few small examples of neural networks and gained a few impressions. For example:
  - the number of nodes in the first hidden layer is “close” to the number of dimensions in the input layer.
  - every hidden neuron has as many inputs (dendrites) as the dimension of the previous (input) layer and
  - the number of output nodes is the number of classes (categories) we are trying to identify.
- Layers made of neurons fully connected to all elements of the input are ***Dense*** layers or ***Fully Connected*** layers. As we will see shortly, there are other types of hidden layers.

Scaling of Fully Connected NN

- *Fully Connected Neural Networks do not scale well for images with dimensions of hundreds by hundreds of pixels.*
- CIFAR-10 and CIFAR-100 are labeled subsets of a dataset with 80 million tiny images. They were collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton.



Scaling of Fully Connected NN

- In CIFAR-10, images are of size 32x32x3 (32 wide, 32 high, 3 color channels). A single fully-connected neuron in the first hidden layer of a regular Neural Network would have 32*32*3 = 3072 weights. This number seems manageable.
- Consider an image of more respectable size, e.g., 200x200x3. Neurons in the first fully connected layer would have 200*200*3 = 120,000 weights.
- The first layer would need many such neurons. The number of parameters that we need to optimize would become very large!
- Full connectivity is wasteful, and the huge number of parameters leads to overfitting.
- We do not need so many parameters to represent images. Images do not contain that much information.

A view of a forest, Information Content

- This photo has perhaps 660x440X3 ~ 300,000 pixels x3 RGB colors ~ 900,000 bits of information. That does not mean that the image has 900,000 relevant features.
- We could extract tree trunks, branches, and leaves and would come up with a modest number of components constituting this image.

~ 30 trunks
~ 100 branches
~ 3000 leaves

- Information content of this image is moderately small.
- There are 400-500 features.
- This is a general rule for most images.
- The numbers of actual features are much fewer than the numbers of pixels.

Feature Extraction

- As the image size n^2 grows, the numbers of weights, i.e., parameters of the fully connected hidden layers could grow at the rate of $(n^2)^2 = n^4$
- One simple solution to this problem is to restrict the number of connections between the hidden units and the input units, allowing each hidden unit to connect to only a small subset of the input units.
- This idea of having locally connected networks also draws inspiration from the way how the biological visual systems appear to be wired up.
- Specifically, neurons in the visual cortex have localized *receptive fields* (i.e., they respond only to stimuli in a certain, restricted, location).
- Some people argue that CNNs took an inspiration from the brain's visual cortex.
- Some people claim that the visual cortex took an inspiration from CNNs.
- The visual cortex has small regions of cells that are sensitive to specific alignment of the visual field.

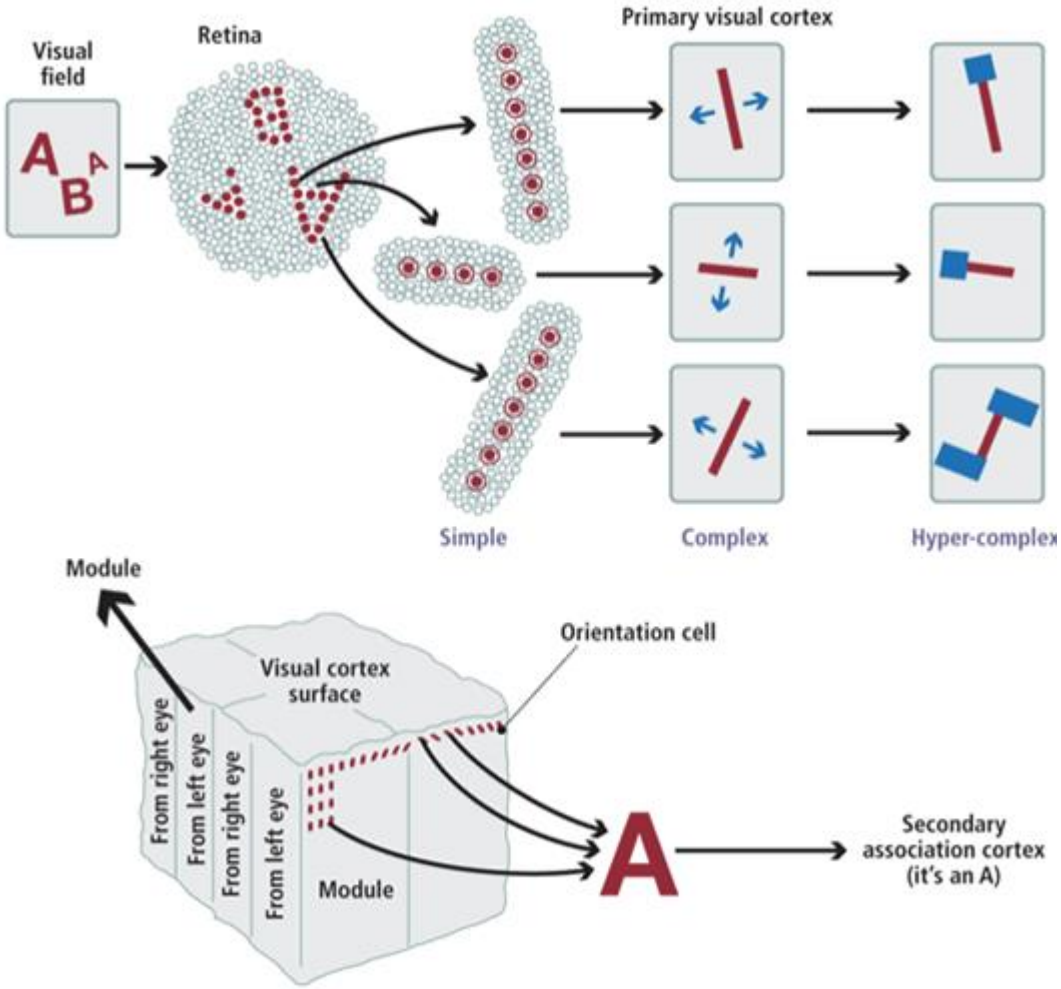
Visual Cortex

Visual Cortex Motivation

- David H. Hubel and Torsten Wiesel performed a series of experiments on cats in 1958 and 1959 (and a few years later on monkeys), giving crucial insights into the structure of the visual cortex.
- The authors received the Nobel Prize in Physiology and Medicine in 1981 for that work.
- They showed that many neurons in the visual cortex have a small local receptive field, meaning they react only to visual stimuli located in a limited region of the visual field (see the image on next slide)
- The receptive fields of different neurons may overlap, and together they tile the whole visual field. A receptive field is a region on retina from which neurons accept inputs.
- Hubel and Wiesel showed that some neurons react only to images of horizontal lines, others react only to vertical lines, and others to images with different orientations.
- Two neurons may have the same receptive field but react to different line orientations.
- Some neurons react to the combinations of the outputs of neurons activated by lower-level patterns, in effect reacting to more complex patterns.
- These observations led to the idea that neurons in visual cortex are organized in layers (hierarchy) and higher-level neurons respond to the outputs of lower-level neurons.
- Each neuron is connected only to a few neurons from the previous layer.
- This neural architecture can detect all sorts of complex patterns in any area of the visual field.

Visual Cortex

- In the Visual Cortex, simple cells become active when they are subjected to simple stimuli such as edges.
- Complex cells combine the information of several simple cells and detect the position and orientation of a structure.
- Hyper-complex cells detect the endpoints and crossing lines from the position and orientation information.
- Information from hyper-complex cells is then used in the brain's secondary cortex for information association.



Structure of NNs and CNNs

- "Ordinary" Neural Networks receive an input (a single vector) and transform it through a series of *hidden layers*.
- Each hidden layer is made up of a set of neurons, where each neuron could be fully connected to all neurons in the previous layer, and where neurons in a single layer operate without any mutual connections.
- The last fully-connected layer is called the "output layer" and, in classification problems, it produces the class scores.
- Convolutional Neural Networks (CNNs) are similar to ordinary Neural Networks: they are made up of neurons that have learnable (trainable) weights and biases.
- Each neuron receives some inputs, performs a dot product with weights and is optionally followed by a non-linearity. The whole network expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other.
- CNNs usually have a *softmax* function on the last (usually fully-connected) layer.
- CNN architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These make the forward function more efficient to implement and vastly reduce the number of parameters in the network.
- CNNs work well with one dimensional time series data, as well.

Software Implementation of Visual Cortex

Software Implementation of Visual Cortex

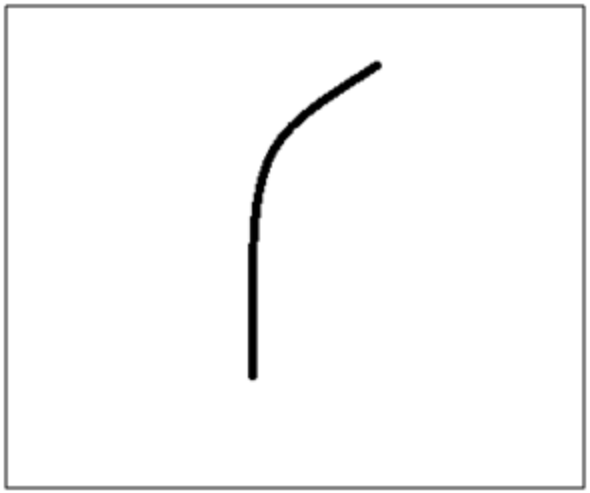
- Findings of David H. Hubel and Torsten Wiesel suggest that at least the first layers of neurons in the visual cortex are made of cells specializing in detecting the simplest of features: horizontal lines, vertical lines, lines at 45 degrees upwards, 45 degrees downwards, etc.
- Each neuron serves as a filter for one specific shape. Filters can be thought of as **feature identifiers**. Features are straight edges, simple colors, and other simple shapes (curves).
- There must be many artificial neurons detecting each features. Dendrites emanating from those neurons cover our retina for presence of individual features in the image projected onto the retina.
- We could imitate such neurons with an arrangements similar to the one depicted on the next slide.
- If the input images are in color, e.g., RGB, they are presented with 3 matrixes, one for each color. For color images, we would have to scan each of three color matrices for every specific features.

Curve Detection Filter

- Curve filter of size 7 x 7 could look like the matrix below and could be used as a curve detector.
- For the moment we will ignore the fact that the filters needs to be 3 units deep for color images. We analyze gray images at the moment.
- As a curve detector, the filter has a pixel structure in which there are higher numerical values along the area that has shape of a curve.
- Filters are matrices made of numbers

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

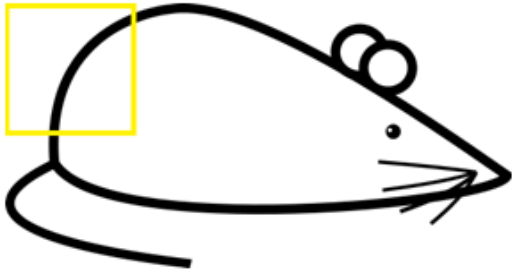


Visualization of a curve detector filter

Mouse Parts

- Filter from the previous slide could for example be used for detection of the back of a mouse on an image that we analyze for the presence of mice. We could put our filter at the top left corner of the below image.

Original image



Visualization of the filter on the image

Visualization of the receptive field

0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

Pixel representation of the receptive field

*

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

Multiplication and Summation = (50*30)+(50*30)+(50*30)+(20*30)+(50*30) = 6600 (A large number!)

- If in the input image, there is a shape that resembles the curve of our filter, then all the element-by-element multiplications summed together will result in a large value.

Mouse Parts

- What happens when we move our filter to the right.

Visualization of the filter on the image

0	0	0	0	0	0	0
0	40	0	0	0	0	0
40	0	40	0	0	0	0
40	20	0	0	0	0	0
0	50	0	0	0	0	0
0	0	50	0	0	0	0
25	25	0	50	0	0	0

Pixel representation of receptive field

*

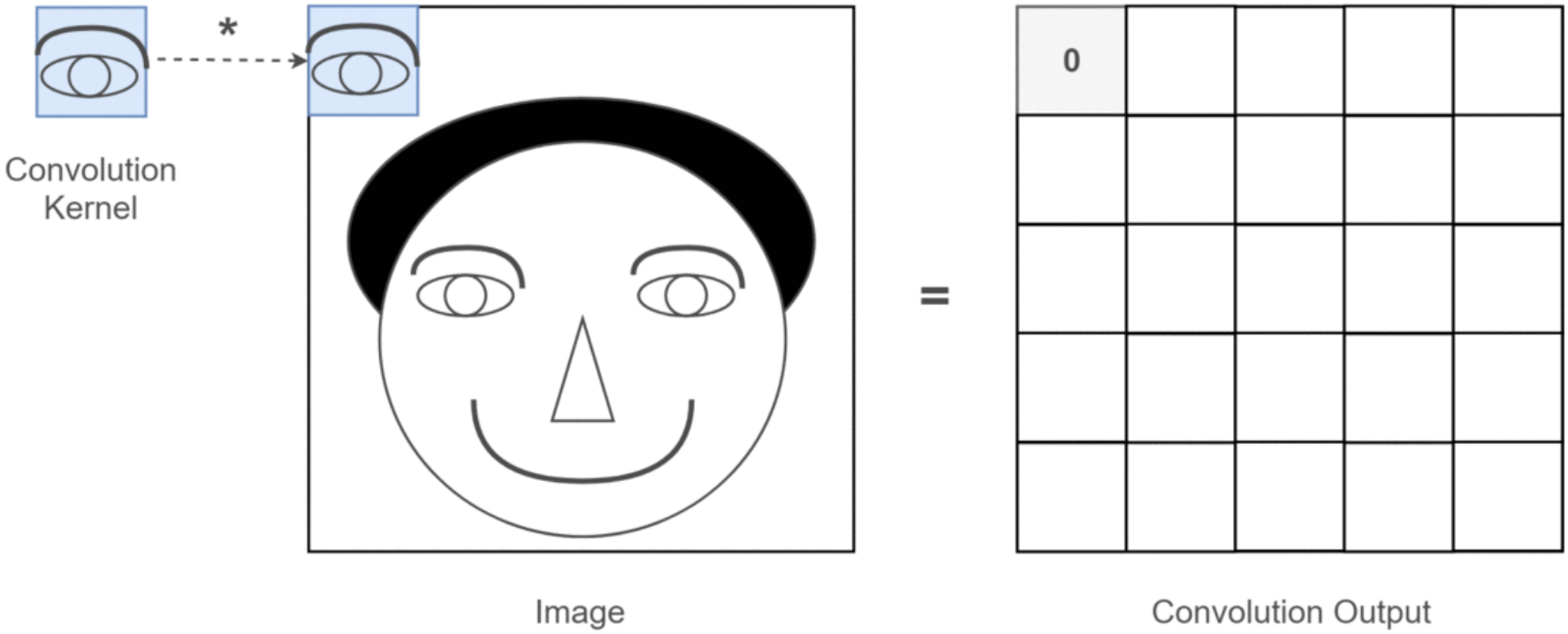
0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

Multiplication and Summation = 0

- The value is much lower! This because there is nothing in the image section that responds to the curve detector filter.
- The output of this conv layer is called an activation map. In the simple case of one filter convolution (with the curve detector filter), the activation map will show the areas in which there are most likely to be “mouse back like” curves in the picture. In above example, the top left value of our 28 x 28 x 1 activation map will be 6600. This high value means that it is likely that there is some sort of curve in the input volume that causes the filter to activate.
- The top right value in the activation map is 0 because there is nothing in the input volume that could cause the filter to activate (there is no curve in that region of the original image).
- This is just for one filter which detect lines that curve outward and to the right.
- We could have other filters for lines that curve to the left or for straight edges and so on.
- With more filters, e.g. , filter for eyes, filter for whiskers, filter for tail, etc., we can create an activation map of greater depth, and gather more information about the input image, and with some luck be able to recognize mice in our images.

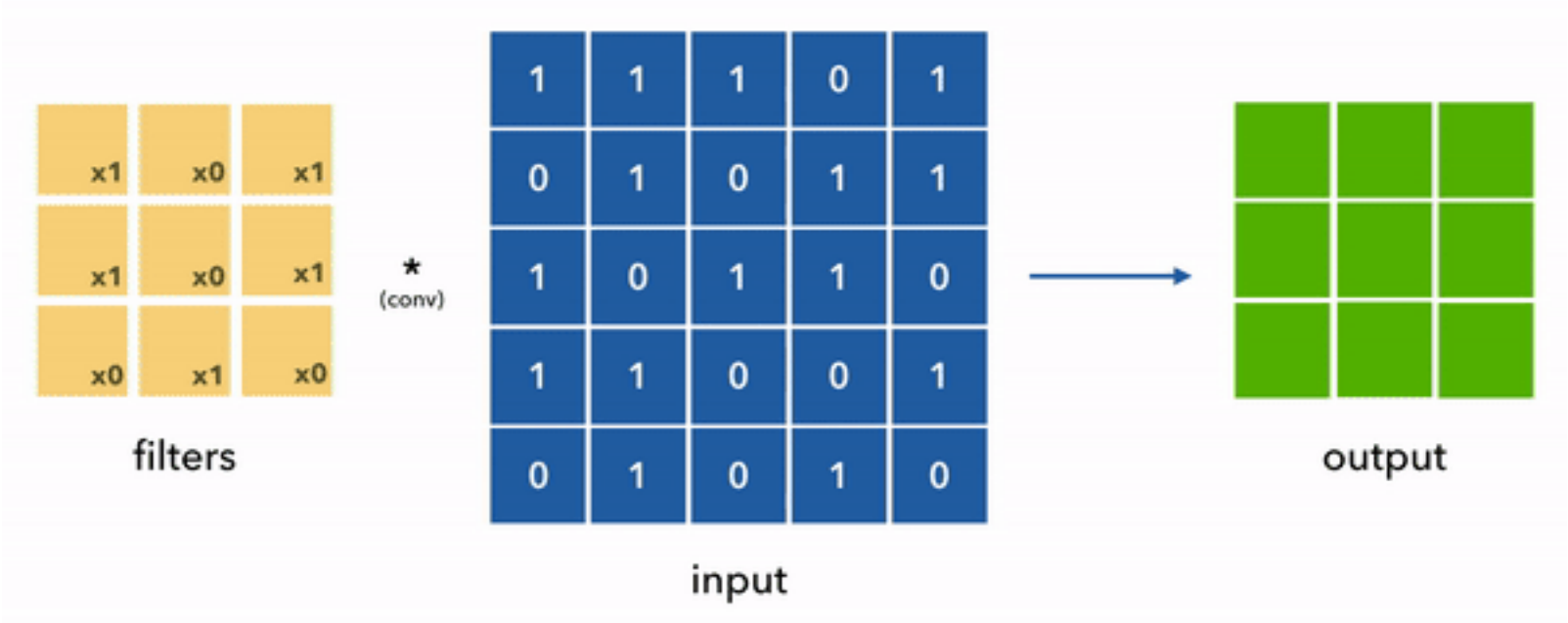
Filter in Action



- A conceptual representation above of an eye filter detecting an eye

Source credits : Stackoverflow

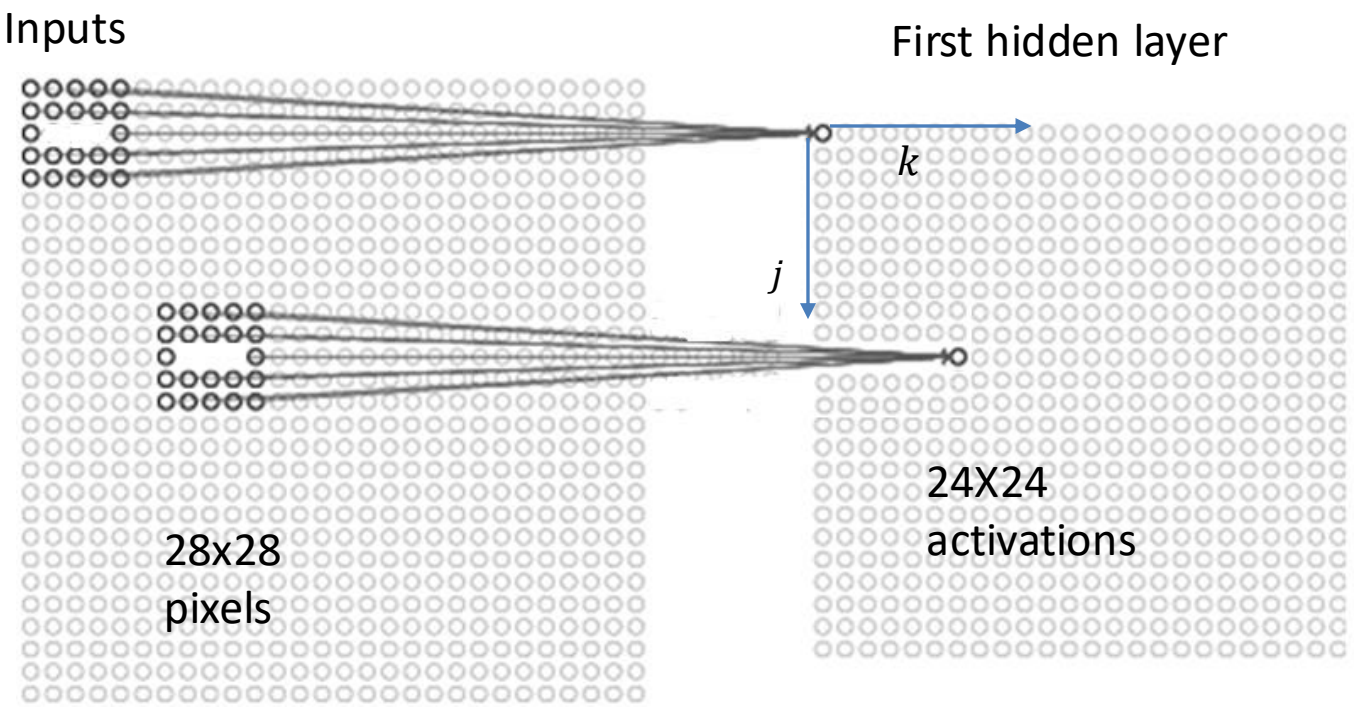
Filter in action



- A conceptual representation above
- A 3x3 filter scanning an input Image of size 5 x 5 (no padding) produces a 3 x 3 output (feature map) through convolution.

Source credits : Wikipedia

Artificial neurons detecting horizontal edges MNIST



- Imagine a 5x5 filter scanning an input volume.
- Volume = image area x depth (depth = 3 for 3 RGB colors)
- The filter produces an activation map of size 24X24. Every color is scanned separately.
- In Mathematics, this process of scanning is referred to as a convolution. There the filter matrix is flipped and then applied to the substrate. No such “flipping” takes place here.
- Filter is sliding left to right and up down in strides of 1,2, 3 and applied to the input image through element-by-element multiplication.

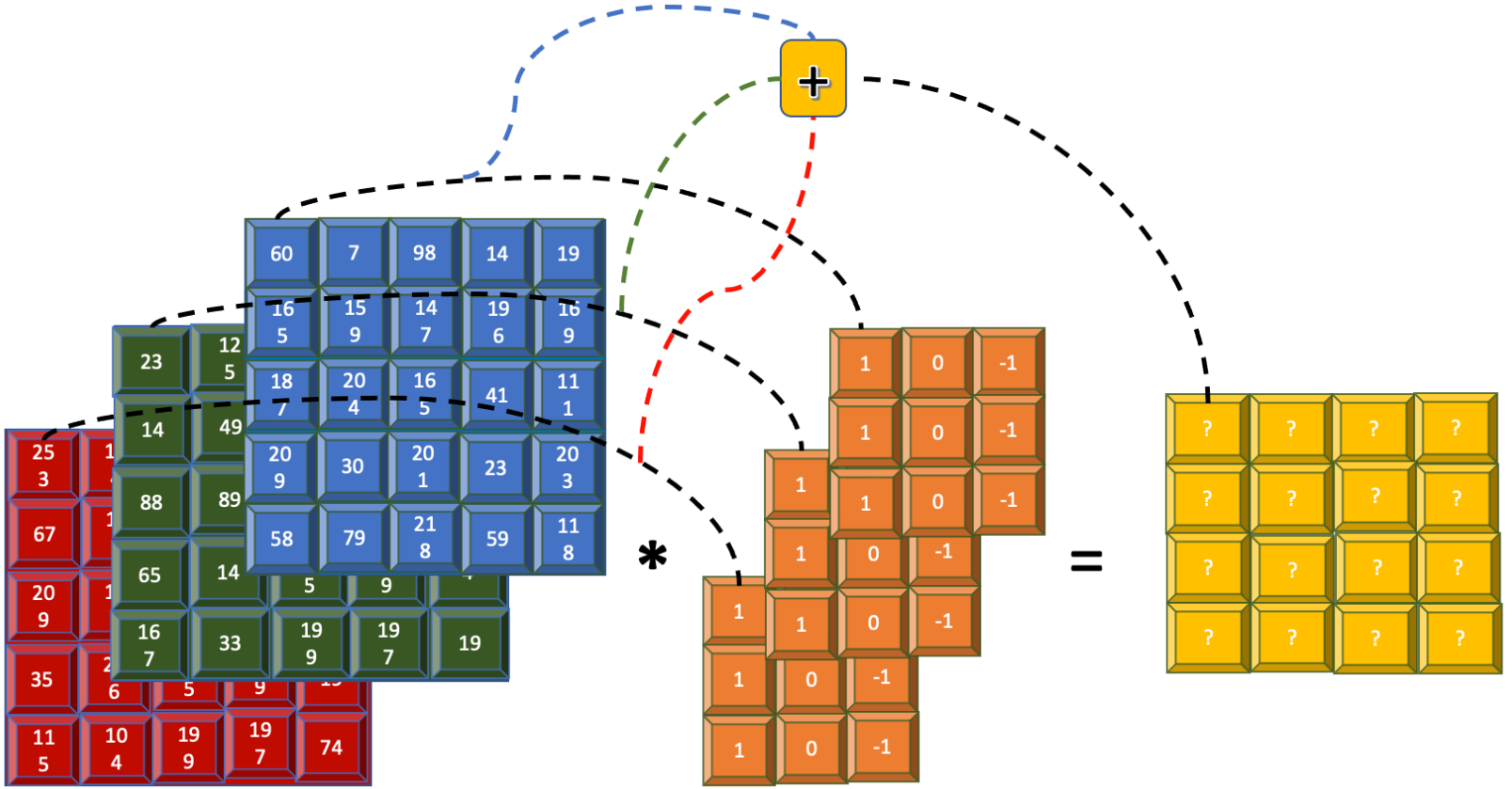
Convolution Layer

- As the filter is sliding, over the input image, it is multiplying the values in the filter with the original pixel values of the image (performing **element wise matrix multiplications**).
- At every position of a 5x5 filter, there would be 25 multiplications. These multiplications are all summed up and the result is a single number.
- As we slide the filter over an image, we repeat this process for every location. For example, we move the filter to the right by 1 unit, then right again by 1, and so on. At the end of the row, we move to the next row and repeat the process.
- After sliding a 5x5 filter over all accessible image locations on a 28X28X1 image we are left with an array of 24 x 24x 1 numbers. We call that array an **activation map** or a **feature map**.
- We produce a 24 x 24 array since there are 24x24 = 576 different locations that a 5 x 5 filter can fit on a 28 x 28 input image.
- If we move the same filter over each of 3 RGB layers, every neuron in the first hidden layer receives inputs from a volume of 28X28X3 numbers.

Convolution Layers and Convolutional Neural Networks

- Hidden layers of artificial neurons that perform feature detection or extraction using filters are now called **Convolutional Layer**s.
- Artificial Neural Networks made of convolutional layers are called **Convolutional Neural Networks or CNNs.**
- We can imagine dendrites emanating from every hidden neuron as light beams of a flash-light. Next, imagine this flash light sliding across the input image pixel by pixel down along the first column. Then it moves to the next column and repeats the slide.
- This flashlight is called a **filter** (or sometimes referred to as a **kernel**) and the region that it is shining over is called the **receptive field**.
- This filter is also an array of numbers (the numbers are called **weights** or **parameters**).
- Convolution layers could have many sublayer each representing one special filter. The number of sublayers is referred to as **depth of a layer.**
- If the input is an RGB encoded color image, every filter (every neuron in every sublayer) takes input from all 3 RGB channels.

Convolution Layers and Convolutional Neural Networks



For a *5x5* image represented over 3 channels, the *3x3* filter is now replicated three times, once for each channel. The input image is a *5x5x3*array, and the filter is a *3x3x3* array. However, the output map is still a 2D *4x4* array. The convolutions on the same pixel through the different channel are added and are collectively represented within each cell.

Convolution

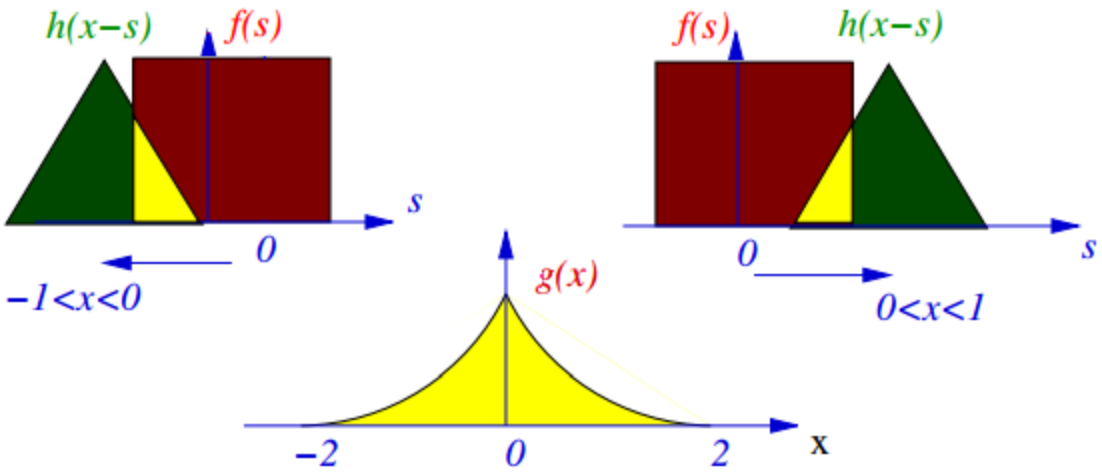
Mathematical Convolution

- The convolution defines a product on the space of integrable functions.
- Convolution is typically presented as a special product between two functions, with a thick asterisk "*" or a dot in a circle "⊙" rather than a simple " ·", dot, or " ", absence of a dot, which are used for regular products of two functions.

$$f * h = h * f$$

- The above usually implies the following integral expression:

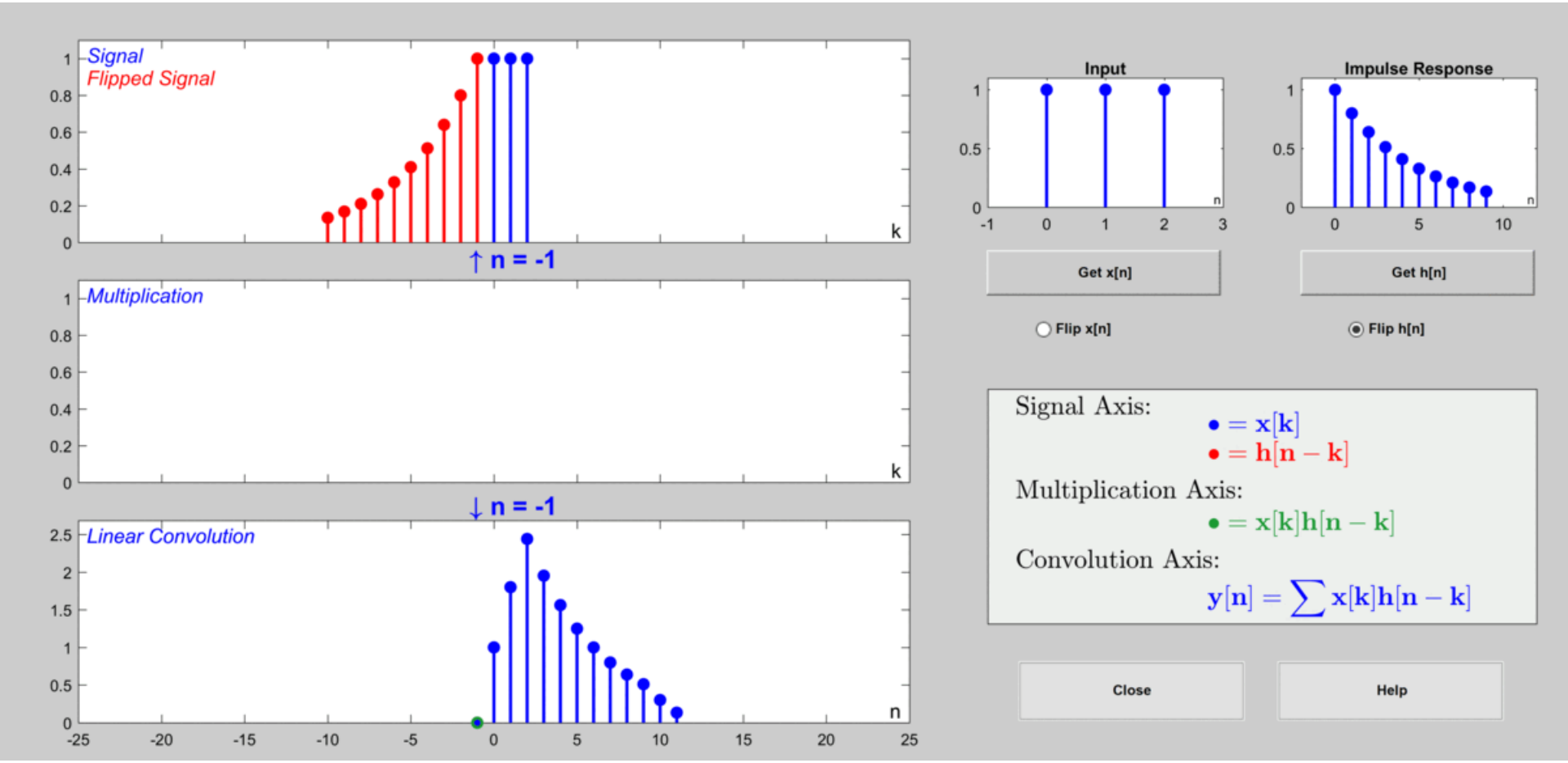
$$g(x) = f * h = \int_{-\infty}^{\infty} f(s)h(x-s)ds$$



You visualize the convolution by imagining two functions. One of them, e.g., $f(s)$, is stationary and $h(x-s)$ is rolling from left to right. The rolling overlap of two functions is the convolution. Image below illustrates the concept.

If we calculate the convolution of $h(s)$ with $f(s)$, where $f(s)$ is shifted by T, the resulting $g(x)$ will be shifted by T, as well. This property is called the ***translational invariance***.

Mathematical Convolution



Credit : https://angeloyeo.github.io/2019/06/18/Discrete_Time_Convolution_en.html

Edge Detection in Image Processing

- One use of convolution is edge detection in images. Here, $f(t)$ is an image (a matrix) and the kernel is a $3X3$ matrix. We slide the kernel over the image left to right and top to bottom and at every position perform element-wise multiplication.

$$f(t) =$$ 

$$g(t) = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

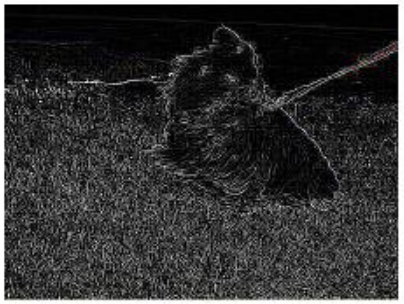
Image Input

Kernel

Convolution Layer

Filtered Output

$$(f \star g)(t)$$



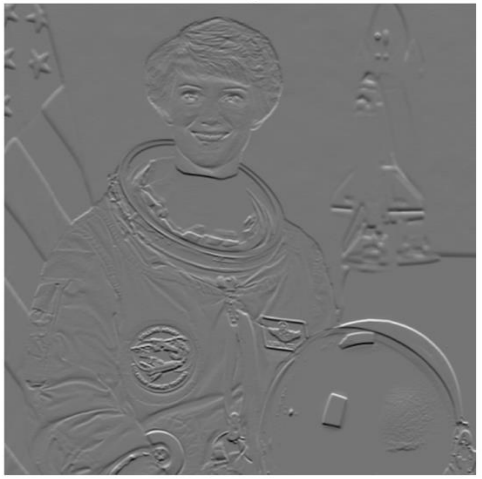
- Operator $g(t)$ is a discrete version of a ***Laplacian***, a special derivative of the second order .
- If you are mathematically inclined, you might notice that we are calculating cross-correlation and not convolution. Please, do not tell anyone.

Edge Detection in Image Processing

Vertical Edges (Sobel-X)

Sharpened Image

Horizontal Edges (Sobel-Y)



Here you can see the effects of **different kernels** applied to the same image:

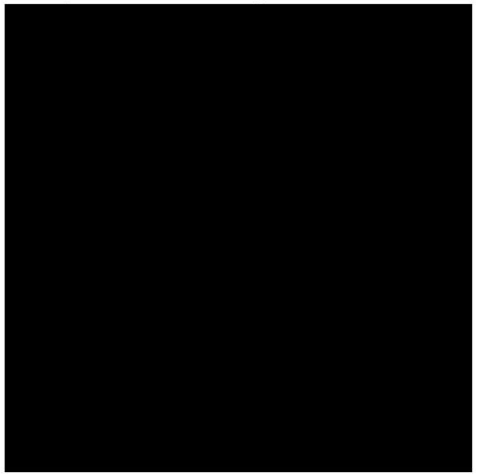
- **Left (Vertical Edges – Sobel-X):** Highlights vertical boundaries (like the astronaut's helmet sides).
- **Middle (Sharpening Filter):** Makes details crisper by enhancing contrast at edges.
- **Right (Horizontal Edges – Sobel-Y):** Highlights horizontal boundaries (like mouth, chin, and shadows).

Why Convolve and not just Multiply ?

Original Image

Convolution (Edge Detection)

Multiplication Only (No Features)



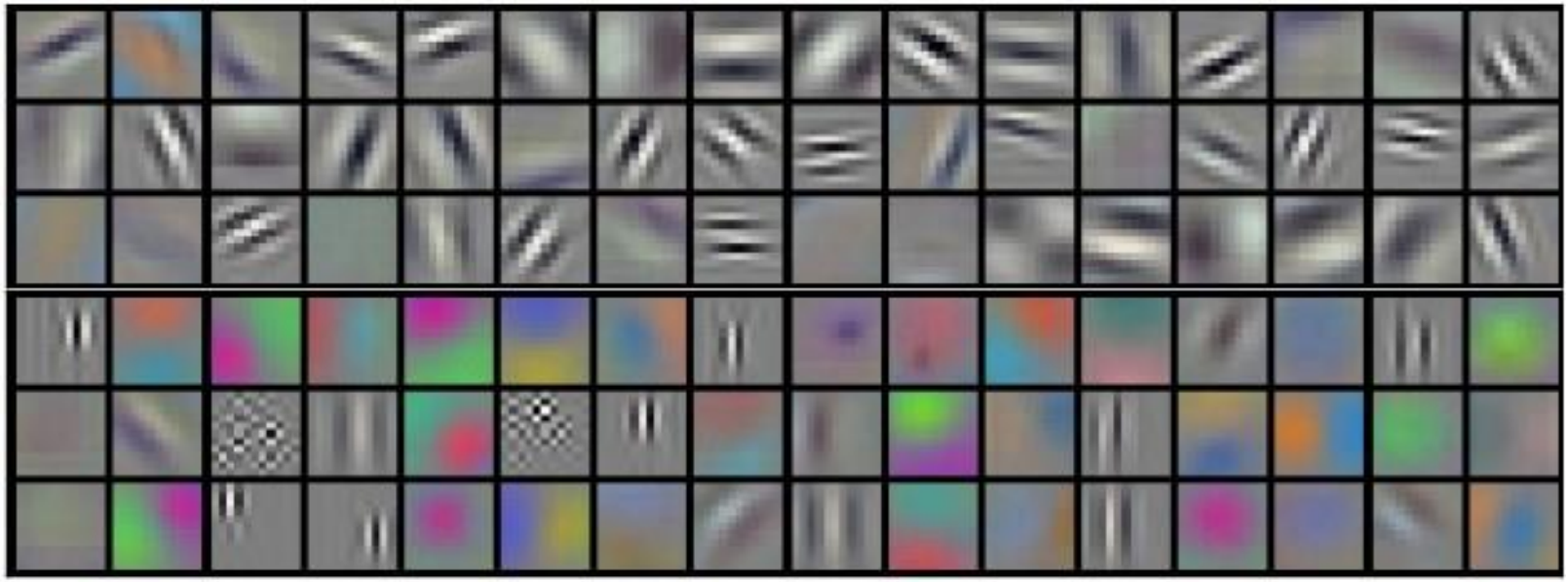
Original Image : astronaut photo.
Convolution (Edge Detection) : edges and structures pop out clearly because convolution combines **neighbors** to detect changes in intensity.
Multiplication Only : image looks basically unchanged (just darker), because multiplying each pixel by a single number ignores spatial context.

Convolution extracts **local patterns** (edges, textures, shapes).
Multiplication only rescales intensity and loses all structural information.

Filters and Feature Extraction

Initial filters and translational Invariance

- In the early work on CNNs by Alex Krizhevsky et al. (2012), (https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks), authors considered 96 filters of size [11x11x3]. Each filter detects certain feature. One detects horizontal edge, the other vertical edge, the third a slanted edge. Some detect various colors.
- A filter which detects a feature at one location in an image, could detect that feature at another location in the same or other images. We refer to that property as the *translational invariance of images*.
- There were 96 sublayers of 55X55 neurons. Each sublayer implemented one filter. All neurons in a sublayer have identical coefficients (weights). This drastically reduces the number of weights we need to find.



Features and Filters

- At the very beginning of development of Convolutional Neural Networks, the researchers indeed imposed filters of many forms, like the ones depicted on the previous slide.
- Each filter had the purpose of determining significance or presence of a particular geometric or chromatic feature. Training of the network determined weights various sublayers applied to their filters.

  https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

  https://www.nvidia.cn/content/tesla/pdf/machine-learning/imagenet-classification-with-deep-convolutional-nn.pdf
- Eventually, it became apparent that it is not necessary to preselect the filters, but that we could let neural network training process determine the shape of those filters.
- As we will see in next lecture, the neural networks discover on their own practically those same filters that Krizhevsky et al. initially imposed on the network.
- Curiously, we know for those particular “feature selecting” filters quite a long time, since early 1940-s and they are known as Gabor’s wavelets.

Images contain object at many scales

Characteristics of Vision, Need for Many Scales

- Images contain object at many scales.
- Consider how our eyes look at the world. In the real world, you can observe a forest like the one of previous pages from many vantage points—in effect, at different scales of resolution.
  - From the window of a jet, for example, the forest appears to be a solid canopy of green.
  - From the window of an automobile on the ground, the canopy resolves into individual trees.
  - If you get out of the car and move closer, you begin to see branches and leaves.
  - If you then pull out a magnifying glass, you will see tiny veins on every leaf.
  - As you zoom in, at smaller and smaller scales, you find details that you didn't see before.
- Try to do that with a photograph, however, and you will be disappointed. Enlarge the photograph to get “closer” to a tree and all you will see is a fuzzier tree; the branch, the leaf, the veins are not visible.
- Although our eyes can see the forest at many scales of resolution, the camera can show only a few at a time.
- Computers do worse than cameras. On a computer screen, a photograph, when we try to zoom in, quickly becomes a collection of pixels that cannot be scaled any further.

A view of a forest, Many Scales, Fractals

- Sometimes, what we see could be observed or analyzed at many scales.
- In Physics and Mathematics such objects/images with many or infinite number of scales are called *fractals*.
- Scale grows exponentially: 2 cm, 4 cm, 8cm, …, 32 cm and so on. We stop when we are out of the picture.
- *CNNs analyze images at many scales as there are convolution layers in the neural network architecture.*

Above figure represents a forest at two different scales.

An Image with Infinite Scale, a Fractal

- The image below has an infinite number of scales.
- As many times you magnify it, you always get the same image.
- Since you can only have CNNs with a finite number of layers, do not apply CNNs to railroads ☺.

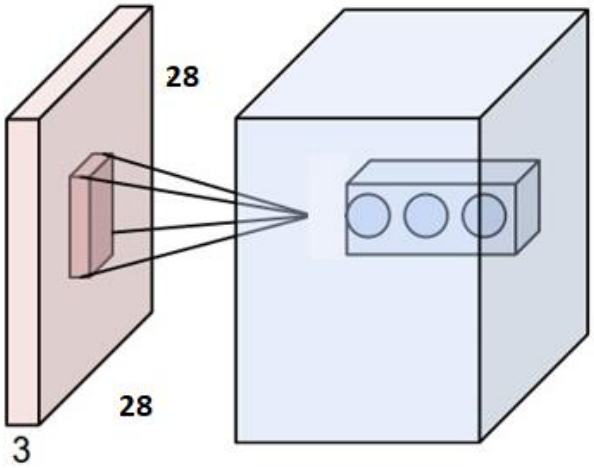
Convolutional Networks

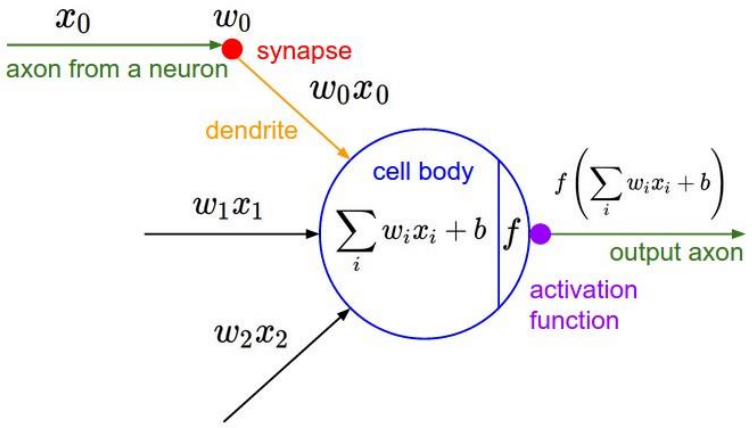
Local Connectivity

- When dealing with high-dimensional inputs such as images, as we saw above, it is impractical to connect neurons to all neurons in the previous volume. Instead, we will connect each neuron to only a local region of the input volume, the **receptive field**
- The spatial extent of local connectivity or size of the receptive filed is a hyper parameter called the ***filter size*** or ***kernel size***. The extent of the connectivity along the depth axis is usually determined by the depth of the input volume (3 for RGB).
- The connections are local in space (along width and height), but almost always full along the entire depth of the input volume.
- For example, suppose that the input volume has size [24x24x3], (an RGB image). If the receptive field (or the filter size) is 5x5, then each neuron in the Conv Layer will have dendrites and corresponding weights to a region of size [5x5x3] in the input volume. The total number of parameters will be 5X5X3 = 75 weights and +1 bias parameter.
- The extent of the connectivity along the depth axis is 3, since this is the depth of the input volume.
- Suppose an input volume had size [16x16x20]. For a receptive field of size 3x3, every neuron in the Convolutional Layer would have a total of 3X3X20 = 180 connections from the input volume. Again, the connectivity is local in space (e.g. 3x3), but full along the input depth (20).

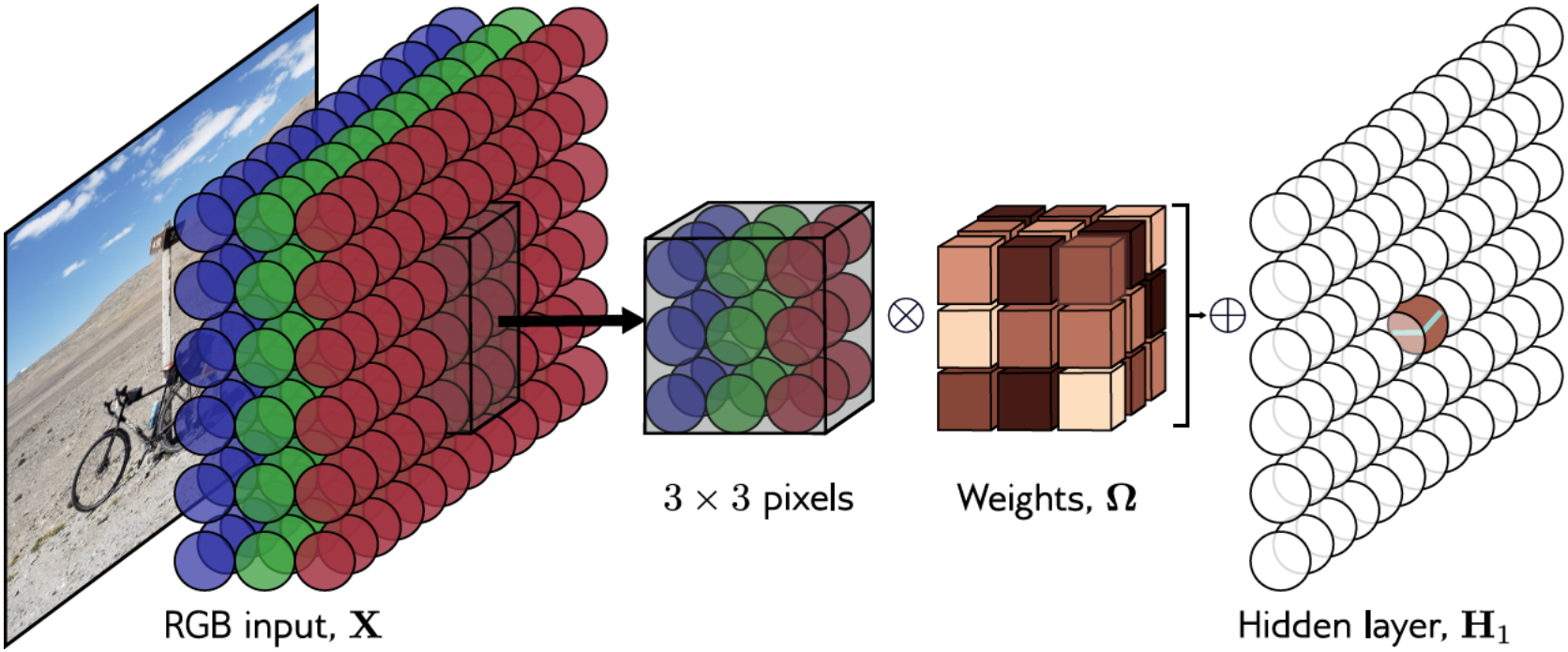
Local Connectivity

- The input volume on the left is 28X28X3, and the number of neurons in the first convolutional layer is 24X24X#filters. Number of filters is somewhat arbitrary: 32, 64, 128
- Each neuron in the convolutional layer is connected only to a local region in the input volume laterally and to all color channels of the input image. Each neuron in the first conv layer sends dendrites to all 3 RGB channels. Weights are different for different channels.





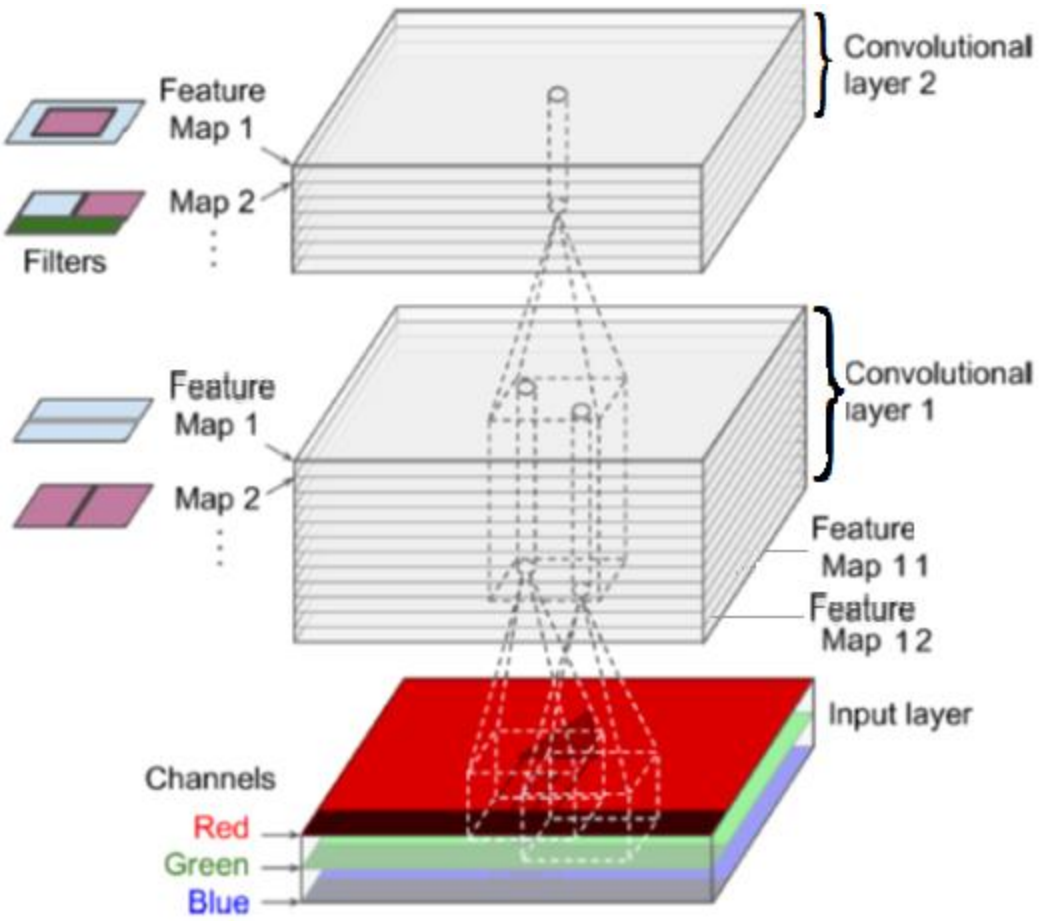
Local Connectivity



The neurons compute a dot product of their weights and the inputs followed by a non-linearity. In difference to fully connected layers, connectivity of each neuron is now spatially localized. Dendrites from any neuron in a dense layer reach every element of the input tensor. Dendrites emanating from a neuron in conv layer reach only a few elements of the input tensor (kernel-size^2 *channels)

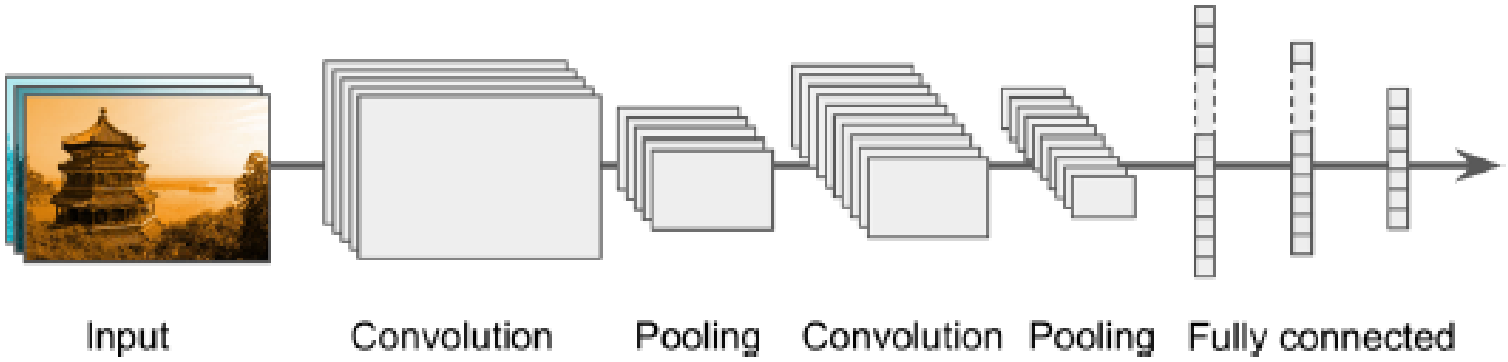
Stacking Feature Maps

- Every filter in Krizhevsky et al. CNN represents a feature. A layer of neurons discovering contribution of that feature to an image produces an output called a *feature map.*
- Within one convolutional layer we stack many filters and they produce as many feature maps. We assume the spatial invariance all neurons within a convolutional layer. All neurons in a layer share the same parameters (weights).
- Typical arrangement of those filters within layers is illustrated in the image to the right.
- Every neuron in a filter (a sublayer) sees (receives inputs) from all feature maps (filters, channels) in the previous layer.



Typical CNN Architecture

- Simple CNN is a sequence of layers, and every layer of a CNN transforms one volume of activations to another through a differentiable function. We use three main types of layers to build CNN architectures: **Convolutional Layer**, **Pooling Layer**, and **Fully-Connected Layer** (the same as used in regular Neural Networks).
- Typical CNN architectures stack a few convolutional layers (each one generally followed by a ReLU layer), then a pooling layer, then another few convolutional layers (+ReLU), then another pooling layer, and so on.
- The image gets smaller and smaller as it progresses through the network, but it also typically gets deeper and deeper (i.e., with more feature maps).
- At the top of the stack, a regular feedforward neural network is added, composed of a few fully connected layers (+ReLUs), and the final layer outputs the prediction (e.g., a softmax layer that outputs estimated class probabilities).



Number of Scales and Number of Conv Layers

- The number of scales over which a CNN Architecture can analyze an image is equal to the number of Convolutional layers used to build a particular network architecture.

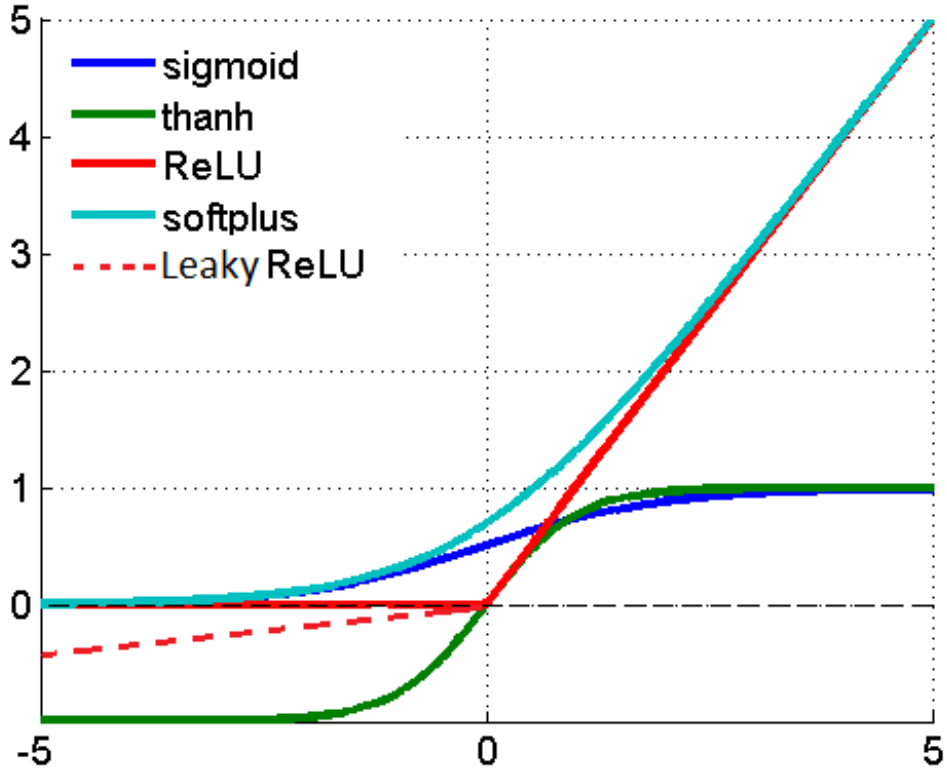
Memory Requirements

- CNNs require large memory. Memory requirements per kernel are not large, however, CNNs typically have large numbers of filters per layer and large number of layers.
- If your training process crashes due to out-of-memory error, try:
  - Reduce the size of (mini-)batch
  - Reduce dimensionality by using stride = 2 or larger.
  - Remove a few layers
  - Use float16 rather than float32.
  - Distribute training over multiple GPUs.

Activation functions

- Initially, we mostly spoke about sigmods, $\sigma(x)$. In CNNs we use other non-linear functions. Rectifier Linear Unite (ReLU) and Leaky ReLU are frequent choices:

$$ReLU(x) = \max(0, x)$$



$$Leaky\ ReLU = \max(0, -\alpha x) + \max(0, x)$$

- Sigmod, $\sigma(x)$, has two flat regions left and right of zero and gradient descent propagation will be very, very slow if our inputs lead us to those regions.
- $ReLU(x)$ has a slope of 1 for $x > 0$ and propagation will not be interrupted for x > 0.
- We resort to the $Leaky\ ReLU$ when we cannot control input variables and are likely to end up left of the zero.

Layers used in CNN

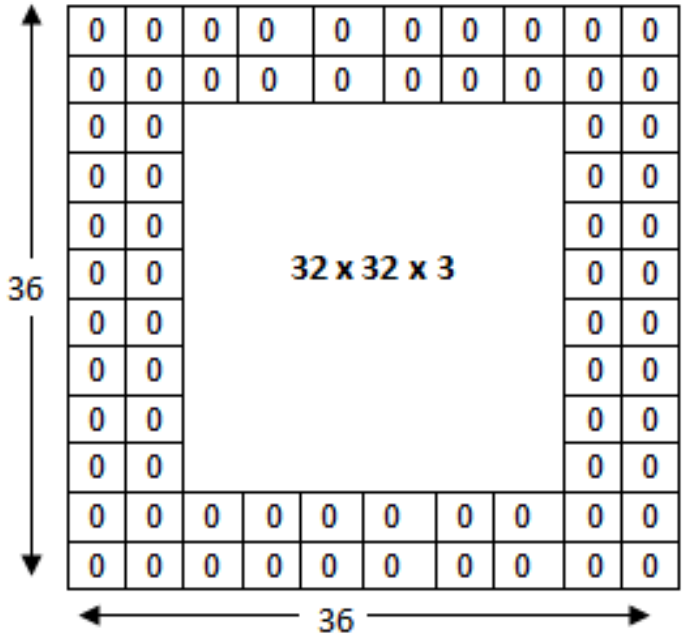
- *A* simple CNN for image classification could have the architecture: INPUT - CONV - RELU - POOL - FC.
- INPUT [32x32x3] holds the raw pixel values of the image, in this case an image of width 32, height 32, and 3 color channels R,G,B.
- CONV layer computes the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small [5x5] region they are connected to in the input volume. This may result in a volume such as [28x28x3] if we decided to use 3 filters, or 28x28x12 if we use 12 filters.
- RELU layer applies an elementwise activation function, such as the $\max(0, x)$ with break at zero. This leaves the size of the volume unchanged ([28x28x3]).
- POOL layer performs a down-sampling operation along the spatial dimensions (width, height), resulting in a volume such as [14x14x3].
- FC (Fully Connected) layer computes the class scores, resulting in a volume of size [1x1x10], where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10. As with ordinary Neural Networks, in Fully Connected layer, each neuron is connected to all the activation outputs in the previous volume.

Spatial Arrangements

- There are three hyper parameters that control the size of the activation volumes: the **depth, stride** and **zero-padding**. Hyper parameters need to be selected and tuned.
- The **depth** of the activation volume corresponds to the number of filters used, each learning to look for something different in the input. For example, if the first convolutional layer takes as input the raw image, then different neurons along the depth dimension may activate in presence of various oriented edged, or blobs of color. We will refer to a set of neurons that all look at the same region of the input as a **depth column.** Some people use the term *fibre*.
- We must specify the **stride** with which we slide the filter. When the stride is 1, we move the filters one pixel at a time. When the stride is 2 (or uncommonly 3 or more pixels) then the filters jump 2 (or more) pixels at a time as we slide them around.
- Stride of 2 produces an activation volume of approximately half the size of the original. Why would we do that? Sometimes, we have too many parameters (weights) and we want to reduce their number to avoid overfitting.
- Sometimes, it is essential not to lose the information along the edges of an image. To preserve that information, we pad the input images with rows and columns of zeros around the border. The size (width) of this **zero-padding** is a hyper parameter.

Padded Images

- The original input volume was 32x32x3. If we pad the borders with 2 rows and 2 columns of zeros (0) we will have a 36x36x3 volume.
- When we apply the convolutional layer with 5x5x3 filters and stride 1, the output, i.e., activation volume, will remain of the same size as the original, i.e., 32x3x3.
- Padding not only preserved the information around the edges, it also prevents the image content from shrinking with subsequent convolutional layers. Sometimes we have a quite a few of convolutional layers (10-100) and loosing 4 = (32 – 28) = 5 – 1 rows and columns every time would quickly diminish the image.



- Sometimes paddings are made not of zeros but of replicated edges. This is presumed to reduce dilution of the image.

Padding and Strides

- In Conv2D layers, padding is configurable via the padding argument, which takes two values: "valid", which means no padding (only valid window locations will be used); and "same", which means "pad in such a way as to have an output with the same width and height as the input." The padding argument defaults to "valid".
- The other factor that can influence output size are *strides*. The description of convolution so far has assumed that the center tiles of the convolution windows are all contiguous. But the distance between two successive windows is a parameter of the convolution, called the *stride*, which defaults to 1. It is possible to have *strided convolutions*: convolutions with a stride higher than 1.
- Using stride 2 means the width and height of the feature map are down-sampled by a factor of 2 (in addition to any changes induced by border effects). Strided convolutions are rarely used in practice, although they can come in handy for some types of models.
- To down-sample feature maps, instead of strides, we tend to use the *max-pooling* operation, which we saw in action in the first CNN example.

Efficient Matrix Multiplications

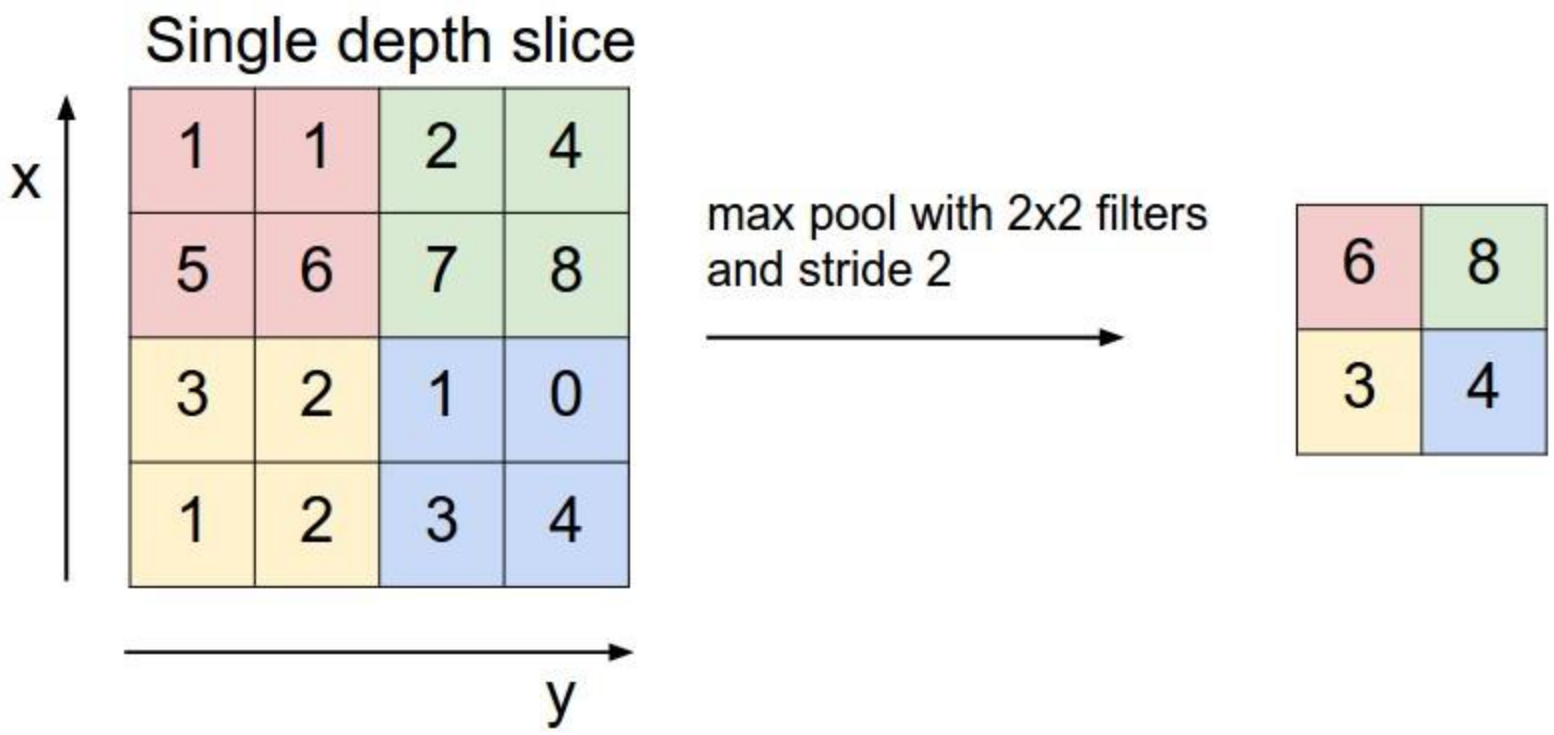
- The convolution operation essentially performs dot products between the filters and local regions of the input. A common implementation pattern of the CONV layer takes advantage of this fact and formulates the forward pass as one big matrix multiply.
- The local regions in the input image are stretched out into columns in an operation commonly called **im2col**. For example, if the input is [227x227x3] and it is to be convolved with 11x11x3 filters at stride 4, then we would take [11x11x3] blocks of pixels in the input and stretch each block into a column vector of size 11*11*3 = 363. Iterating this process in the input at stride of 4 gives (227-11)/4+1 = 55 locations along both width and height, leading to an output matrix X_col of *im2col* of size [363 x 3025], where every column is a stretched out receptive field and there are 55*55 = 3025 of them in total. Note that since the receptive fields overlap, every number in the input volume may be duplicated in multiple distinct columns.
- The weights of the CONV layer are similarly stretched out into rows. For example, if there are 96 filters of size [11x11x3] this would give a matrix W_row of size [96 x 363].
- The result of a convolution is now equivalent to performing one large matrix multiply np.dot(W_row, X_col), which evaluates the dot product between every filter and every receptive field location. In our example, the output of this operation would be [96 x 3025], giving the output of the dot product of each filter at each location.
- The result must finally be reshaped back to its proper output dimension [55x55x96].

Pooling Layer

- It is common to periodically insert a Pooling layer in-between successive Convolutional layers in a CNN architecture.
- The function of Pooling Layer is to progressively reduce the spatial size of the representation and reduce the number of parameters and computations in the network, and, hence, control the overfitting.
- The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation.
- The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2. This pooling layer downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations.
- Average pooling was often used historically but has recently fallen out of favor compared to the max pooling operation, which has been shown to work better in practice.

Illustration of Activity in the Pooling Layer

- Image below illustrates a Pooling layer with filters of size 2x2 applied with a stride of 2.
- MAX operation selects a maximum over 4 numbers (2x2 region in some depth slice). The depth dimension remains unchanged.



A Small CNN for Classification of MNIST Digits

- A small CNN we will examine has a few layers:

```
import tensorflow as tf
from ternsorflow import keras
from tensorflow.keras import layers
from tensorglow.keras import models
model = models.Sequential()
model.add(layers.Conv2D(32,(3, 3),activation='relu',input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

- In the first layer, `32 is` the number of filters, `(3,3)` is the kernel, i.e., size of the receptive field. Strides take the default values `(1,1)`. All other parameters take the default values.
- CNN takes as inputs tensors of shape `(image_height,image_width, image_channels)` (usually, we do not include the batch dimension).
- In this case, we configure the CNN to process inputs of size specified by `input_shape=((28, 28, 1),` which is the format of MNIST images. Parameter `input_shape` is passed only to the first layer. All other layers adjust themselves.
- In general, parameter **`input_shape`** is a
  - `4D` tensor with shape: `(batch, channels, rows, cols)` if `data_format` is `"channels_first"` or
  - `4D` tensor with shape: `(batch, rows, cols, channels)` if `data_format` is `"channels_last"`.
  - Default argument list for Keras with TensorFlow is `channels_last`
- `batch` size, or sample size is usually omitted.

Number of trainable parameters

Summary and Architecture

- We display the details of the architecture using command `model.summary()`

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)            320
_________________________________________________________________
maxpooling2d_1 (MaxPooling2D) (None, 13, 13, 32)           0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 64)            18496
_________________________________________________________________
maxpooling2d_2 (MaxPooling2D) (None, 5, 5, 64)             0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 3, 3, 64)              36928
=================================================================
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
```

- You can see that the output of every Conv2D and MaxPooling2D layer is a 3D tensor of shape `(height, width, channels)`. The width and height dimensions tend to shrink as you go deeper into the network.
- The number of channels (filters) is indicated by the last argument passed to the Conv2D layers (32, 64 and 64 in the first, second and the third Conv2D layer of our network).

Number of unknown parameters, `conv2d_1`

- In the summary, we see the number of unknown parameters:

`conv2d_1 (Conv2D) (None, 26, 26, 32) 320`

Where is the number `26` coming from?

- Input into the first `Conv2D` layer is symmetric (a MNIST image): $(nx_{in}, ny_{in}) = (28,28)$.
- Padding $p = 0$, stride $= (1,1)$, or $s = 1$. Kernel is symmetric: $(k, k) = (3,3)$.

$$x_{out} = \left[\frac{x_{in}+2p-k}{s}\right] + 1 = \left[\frac{28+2\cdot 0-3}{1}\right] + 1 = 26$$

- The first `Conv2D` layer is also symmetric, and has $(nx_{out}, ny_{out}) = (26,26)$ neurons.

Where is number `320` coming from?

- There are $k * k = 9$ dendrites emanating from every neuron in every one of 32 filters (channels) of `conv2d_1`. In CNNs, we assume parameter sharing, what that means is there is only one set of weights for all neurons in one filter sublayer. There is also one bias parameter per every filter sublayer.

$$9 * 32 + 32 = 320$$

`maxpooling2d_1 (MaxPooling2D) (None, 13, 13, 32) 0`

- `MaxPooling2D((2,2))` layer with kernel= (2,2) shrinks the size of the output of the first `Conv2D` layer from (26,26) to (13,13). `MaxPooling2D` does not change the number of output layers, 32. `maxpooling2d_1` layer has no adjustable parameters, therefore `0` at the end of the line.

Shared Weights and Bias

- All neurons in one filter (channel, a sublayer) in a hidden layer share the same parameterization (weight vector and bias). Output of neurons in any one filter form a 'Feature Map'. It is quite common to see "feature map" used as the term for the filter (a sublayer in a hidden layer) itself.
- *Translation Invariance of Images* allows features to be detected regardless of their position in the receptive field. (Here, a feature is an input pattern that will cause a neuron to activate, for example, a horizontal edge)
- All neurons in the first filter (sublayer of a hidden layer) detect exactly the same feature, just at different locations.
- CNNs are well adapted to *translation invariance of images*: move a picture of a cat, left or right in an image and it still is an image of a cat!
- This reduces the number of free parameters, achieving better generalization and computational performance.
- Weights, W, of a hidden layer can be represented as a 4D tensor containing elements for every combination of destination filter (feature map), source filter (feature map), source vertical position, and source horizontal position.
- Biases, b, can be represented as a vector containing one element for every filter (the destination feature map) .

Number of unknown parameters, `conv2d_2`

`conv2d_2 (Conv2D) (None, 11, 11, 64) 18496`

- Where is number `11` coming from?
- Input into the second `Conv2D` layer is symmetric: $(nx_{in}, ny_{in}) = (13,13)$.
- Padding $p = 0$, stride $= (1,1)$, or $s = 1$. Kernel is symmetric: $(k, k) = (3,3)$.

  $x_{out} = \left[\frac{x_{in}+2p-k}{s}\right] + 1 = \left[\frac{13+2\cdot 0-3}{1}\right] + 1 = 11$
- Output of the second `Conv2D` layer is also symmetric: $(nx_{out}, ny_{out}) = (11,11)$.
- Where is number `18496` coming from?
- Every one of 64 filters (sublayers) in `conv2d_2` sees (convolves with) every one of 32 output layers of `maxpooling2d_1`. There are:

  $64 * 32 = 2{,}048$ layer to layer pairs.
- All neurons within a filter (one of 64 sublayers) in `conv2d_2` are the same and have $k * k = 9$ unknown weights (CNNs parameter sharing) on dendrites pointing to every one of 32 input layers. Thus, we have

  $2{,}048 * 9 = 18{,}432$ unknown weights.
- Each one of 64 sublayers in `conv2d_2` has its own bias parameter. Thus, we have

  18,432 + 64 = 18496 unknown parameters.

`maxpooling2d_2 (MaxPooling2D) (None, 5, 5, 64) 0`

- `MaxPooling2D((2,2))` layer with kernel=`(2,2)`, shrinks the size of the output of the second `Conv2D` layer from (`11,11`) to `(5,5)`. `MaxPooling2D` does not change the number of output sublayers, 64 .
- `maxpooling2d_1` layer has no adjustable parameters, therefore 0 at the end of the line.

Number of unknown parameters, `conv2d_3`

`conv2d_3 (Conv2D) (None, 3, 3, 64) 36928`

Where is number `3` coming from?

- Input into the third `Conv2D` layer is symmetric: $(nx_{in}, ny_{in}) = (5,5)$.
- Padding $p = 0$, stride $= (1,1)$, or $s = 1$. Kernel is symmetric: $(k, k) = (3,3)$.

$$x_{out} = \left[\frac{x_{in}+2p-k}{s}\right] + 1 = \left[\frac{5+2\cdot 0-3}{1}\right] + 1 = 3$$

- Output of the third `Conv2D` layer is symmetric: $(nx_{out}, ny_{out}) = (3,3)$.

Where is number `36928` coming from?

- Every one of 64 filter sublayers in `conv2d_3` sees, i.e. «convolves» with every one of 64 output layers of `maxpooling2d_2`. There are:

  $64 * 64 = 4{,}096$ layer to layer pairs.

- All neurons within a filter layer (one of 64 sublayers) in `conv2d_3` are the same and have $k * k = 9$ unknown weights (CNNs parameter sharing) on dendrites pointing to every one of 64 input layers. Thus we have

  $4{,}096 * 9 = 36{,}864$ unknown weights.

- Each one of 64 sublayers in `conv2d_3` has its own bias parameter, and we have

  $36{,}864 + 64 = 36{,}928$ unknown parameters.

- The sum of all unknown parameters in our network (so far) is:

$$320 + 18{,}496 + 36{,}928 = 55{,}744$$

Flattening

- We need to feed the output of `conv2d_3` which is a 3D tensor(of shape (3, 3, 64)) into a densely connected classifier layer. For that purpose, we will use a `Flatten`.
- A flatten operation on a tensor reshapes the tensor to have a shape that is equal to the number of elements contained in the tensor. This transforms the tensor of shape (3, 3, 64) with 576 elements into a 1d-array of 576 elements.
- The output of the `Flatten` layer is first fed into a `Dense` layer with 64 neurons. Since we want to perform 10-way classification, the network will end with a final Dense layer with 10 neurons and 10 outputs and the `softmax` activation function

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

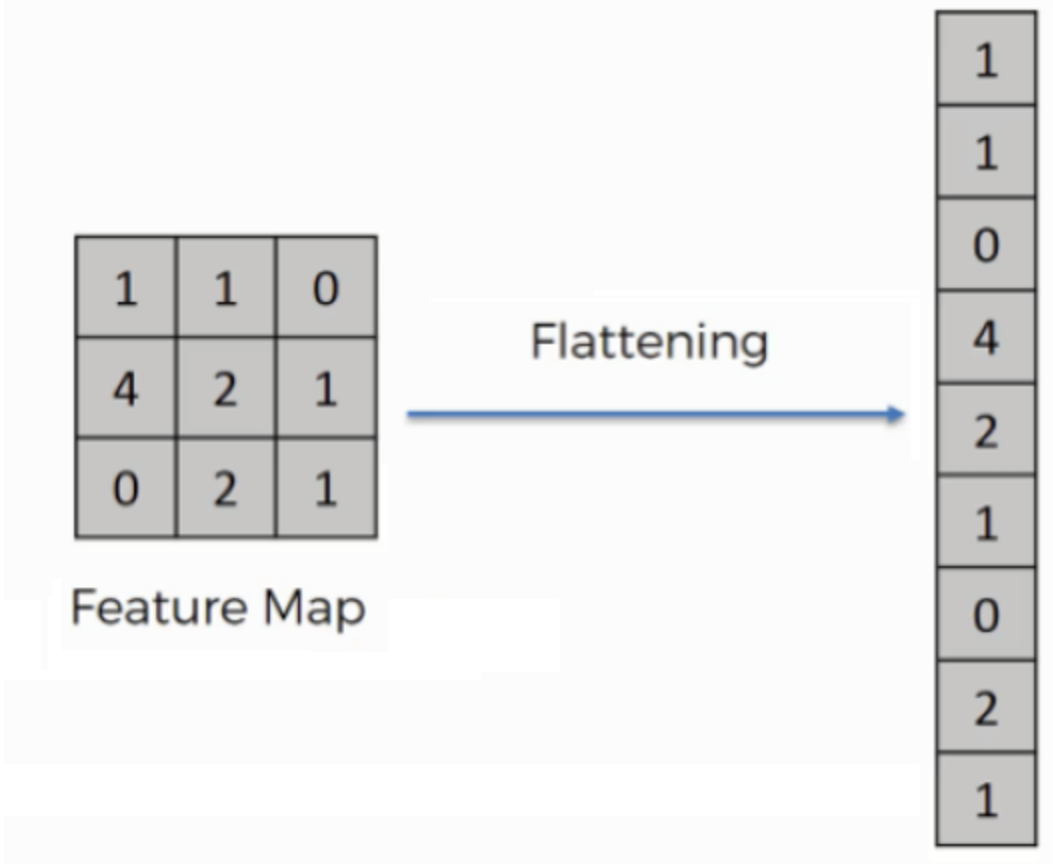
- In **`model.summary()`**, new layers have the following description:

```
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
```

- The `(3, 3, 64)` outputs from `conv2d_3` layer are flattened into vectors of shape `(576,)` before going through two Dense layers.

Flattening

- The image bellow presents the effect of flattening.
- A tensor of any dimension is unwrapped and turned into a long vector.



Softmax Function

- The softmax function takes as input a vector z of K real numbers, and normalizes it into a probability distribution consisting of K probabilities proportional to the exponentials of the input numbers. That is, prior to applying softmax, some vector components could be negative, or greater than one; and might not sum to 1; but after applying softmax, each component will be in the interval (0,1), and the components will add up to 1, so that they can be interpreted as probabilities. Furthermore, the larger input components will correspond to larger probabilities.
- If we have K values: z_i, $i = 1, \dots, K$;
- Softmax function has K values:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}},$$

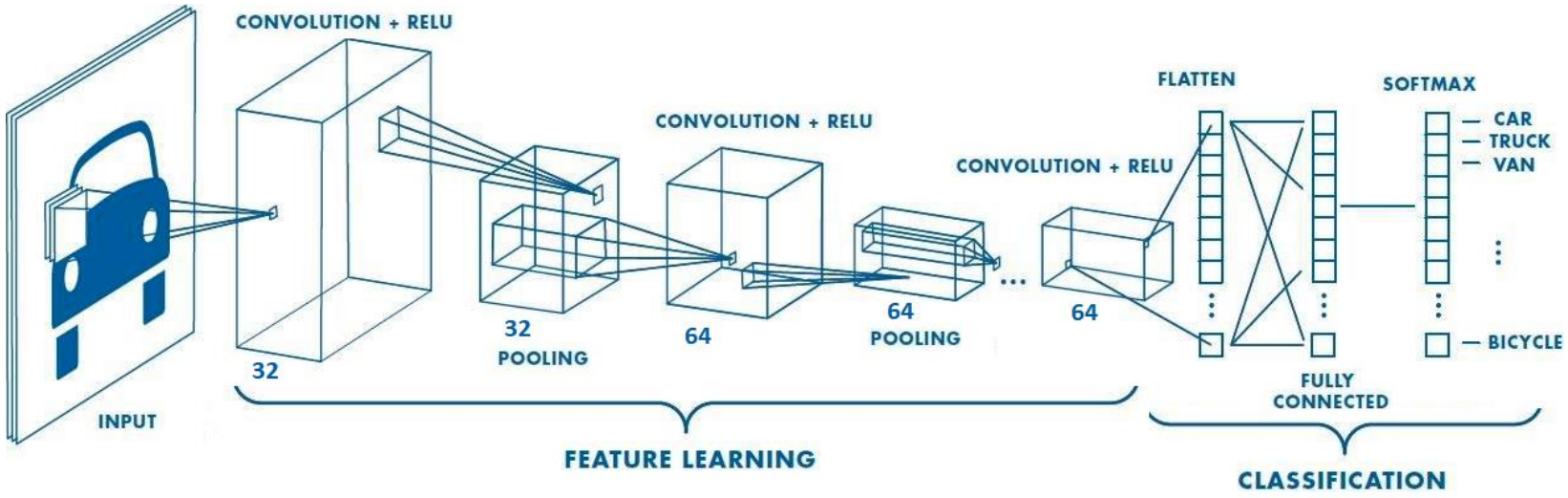
where their sum is equal to 1:

$$\sum_{i=1}^{K} \sigma(z_i) = 1$$

- This allows us to interpret value $\sigma(z_i)$ as the probability that event z_i will happen.
- The softmax function is often used as the last activation function of a neural network to normalize the output of a network to a probability distribution over predicted output classes.

Network Architecture

- Graphics below is a common way of representing the architecture of the CNN.
- This graph is a good presentation of our last network.
- The input could be one channel if digits are gray scale only. This same architecture could be used for RGB images and then the input would have depth 3.
- Convolutional layers have respectively 32, 64 and 64 channels or filters.
- The number of filters tells us the number of outputs of a layer. Output of each filter (sublayer) is a 2D collection of outputs of the activation functions of the individual neurons in that filter.



Training CNN on MNIST Images

- Now, let's train the CNN on the MNIST digits

```
from tensorflow.kears.datasets import mnist
from tensorflow.kears.utils import to_categorical
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
model.compile(optimizer='rmsprop',
loss='categorical_crossentropy',
metrics=['accuracy'])
history = model.fit(train_images, train_labels, epochs=5, batch_size=64)
```

- Let's evaluate the model on the test data:

```
 test_loss, test_acc = model.evaluate(test_images, test_labels)
 test_acc
0.9908000000000001
```

- The basic CNN has a test accuracy of `99.%`. This is a very, very high accuracy.
- We would like to understand where is such high accuracy coming from.

Why Pooling is a Must

Max-Pooling

- In the CNN example, the size of the feature maps is halved after every MaxPooling2D layer. For instance, before the first MaxPooling2D layers, the feature map is 26 × 26, but the max-pooling operation halves it to 13 × 13.
- That's the role of max pooling: to aggressively down-sample feature maps, much like strided convolutions.
- Max pooling consists of extracting windows from the input feature maps and outputting the max value of each channel. It's conceptually similar to convolution, except that instead of transforming local patches via a learned linear transformation (the convolution kernel), they're transformed via a hardcoded max tensor operation. A big difference from convolution is that max pooling is usually done with 2 × 2 windows and stride of 2, to down-sample the linear dimension of feature maps by a factor of 2. The convolution is typically done with 3 × 3 windows and stride of 1.
- Why we down-sample feature maps this way? Why not remove the max-pooling layers and keep fairly large feature maps all the way up?

  One reason to use down-sampling is to reduce the number of feature-map coefficients to process, as well as to induce spatial-filter hierarchies by making successive convolution layers look at increasingly large windows (in terms of the fraction of the original input they cover).

Reasons for Max-Pooling

- Max pooling isn't the only way you can achieve such down-sampling. You can also use strides > 1 in the convolution layer. You can use average pooling instead of max pooling, where each local input patch is transformed by taking the average value of each channel over the patch, rather than the maximum value.
- Max pooling tends to work better than these alternative solutions. In a nutshell, the reason is that features tend to encode the spatial presence of some pattern or concept over the different tiles of the feature map (hence, the term *feature map*), and it's more informative to look at the *maximal presence* of different features than at their *average presence*.
- The most reasonable subsampling strategy is to first produce dense maps of features (via convolutions with stride=1) and then look at the maximal activation of the features over small patches, rather than looking at sparser windows of the inputs (via convolutions with stride ≥ 2) or averaging input patches, which could cause you to miss or dilute feature-presence information.
- You might be under an impression that with `MaxPooling` or any other pooling technique we are just cutting corners and reducing the computational load.
- As it turns out, pooling is essential if we want the neurons in the last layers to feel the influence of all (or most) of the input data.

Why Pooling is a must

- Consider a network with `MaxPooling` layers (blue) and one without those (orange).
- The second blue column contains Sizes of Layers from the summary on slide 50. Conv layers drop in size significantly due to the MaxPooling layers.
- The orange column "Size of Layers" reflects gradual drop in size due to the kernel size only.
- Read numbers from the bottom up. Start with one neuron in the `conv2d_3` layer and determine its receptive field in `maxpool2d_2`. Continue upwards by determining how many neurons in `conv2d_2` are visible by all 3x3 neurons in `maxpool2d_2`. Patch of 3X3 neurons in `maxpool2s_2` is the result of shrinking of 6x6 neurons in `conv2d_2`. So, the receptive field of those 3x3 neurons has size 6x6. Those 6x6 neurons have the receptive field of size 8x8 in maxpool2d_1 layer, and so on.
- With pooling, every neuron in the last conv layer feels the influence of 324 input values or 41.3 % of the input pixels. Without pooling, the last neurons would feel the influence of 6% of the input pixels.

Layer	Sizes of Layers	Sizes of Receptive Fields	Coverage	Sizes of Layers	Sizes of Receptive Fields	Coverage
input	28X28	18x18 = 324	41.3% of 784	28x28	7x7 = 49	6.25% of 784
conv2d_1	26X26	16x16		26x26	5x**5**	
maxpool2d_1	13X13	8x8				
conv2d_2	11X11	6x6		24x24	3x3	
maxpool2d_2	5X5	3X3				
conv2d_3	3x3	1		22x22	1	

Summary and Architecture (slide 50)

- We display the details of the architecture using command `model.summary()`

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)            320
_________________________________________________________________
maxpooling2d_1 (MaxPooling2D) (None, 13, 13, 32)           0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 64)            18496
_________________________________________________________________
maxpooling2d_2 (MaxPooling2D) (None, 5, 5, 64)             0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 3, 3, 64)              36928
=================================================================
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
```

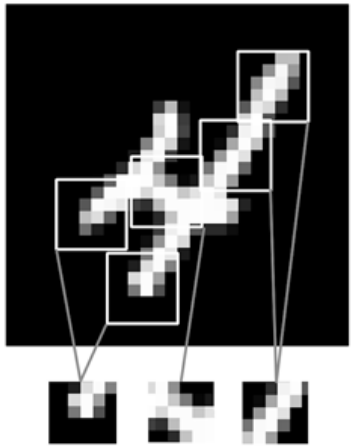
- You can see that the output of every Conv2D and MaxPooling2D layer is a 3D tensor of shape `(height, width, channels)`. The width and height dimensions tend to shrink as you go deeper into the network.
- The number of channels (filters) is indicated by the last argument passed to the Conv2D layers (32, 64 and 64 in the first, second and the third Conv2D layer of our network).

Review of
Key Concepts and Ideas

Densely Connected vs. Convolutional Layers

- Dense layers learn global patterns in their entire input feature space (for a MNIST digit, patterns involving all pixels), whereas convolution layers learn local patterns.

- Typical filters in our network are 3x3, so they learn about small portions of the image.
- If a filter learns how to recognize an edge at 45 degrees in one part of an image, that filter will recognize that edge in other parts of that image. That makes CCNs more efficient.

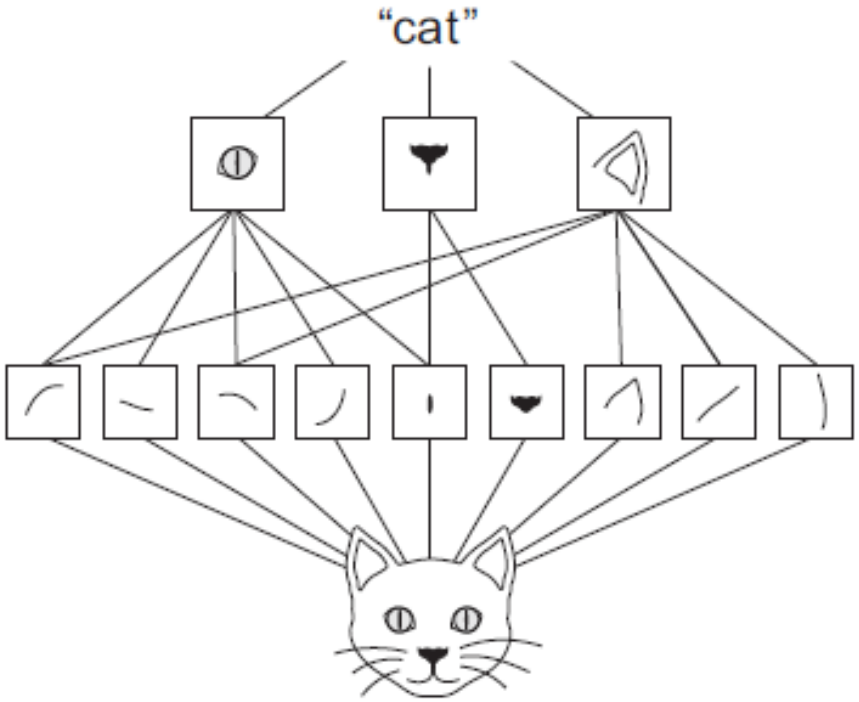


- *Filters learn translationally invariant features.* After learning a certain pattern in the lower-right corner of a picture, a CNN can recognize it anywhere: for example, in the upper-left corner.
- A densely connected network would have to learn the pattern anew if it appears at a new location. This makes CNNs efficient when processing images. *The visual world is fundamentally translation invariant*.
- CNNs need fewer training samples to learn representations that have generalization power.

Hierarchies of patterns

- *CNNs can learn spatial hierarchies of patterns*. 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 CNNs to efficiently learn increasingly complex and abstract visual concepts (because *the visual world is fundamentally spatially hierarchical*).

- An image is a spatial hierarchy of visual modules: hyperlocal edges combine into local objects such as eyes or ears, which combine into high-level concepts such as "cat.“
- A CNN can detect as many levels of a visual hierarchy as there are Convolutional/MaxPooling layers in the network.

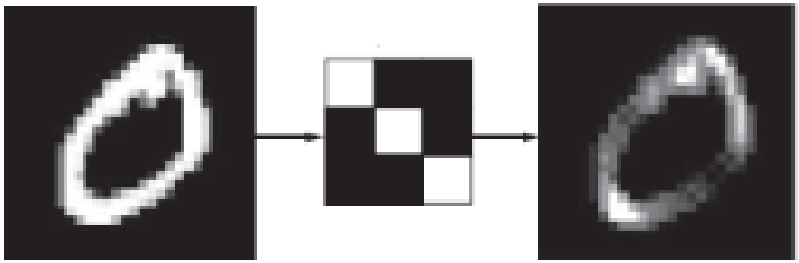


Feature Maps

- Convolutional layers produce and operate on 3D tensors, made of 2D *feature maps*, with two spatial axes (*height* and *width*) organized along a *depth* axis (also called the *channels* axis).
- For an RGB image, the dimension of the depth axis is initially 3, because the image has three color channels: red, green, and blue.
- For a black-and-white picture, like the MNIST digits, the depth is 1 (images are gray).
- The convolution operation extracts patches from its input feature map and applies the same transformation to all these patches, producing an *output feature map*.
- This output feature maps are still organized as a 3D tensor. The tensor has a width and a height. Its depth can be arbitrary, because the output depth is a parameter of the layer. The different channels along that depth axis no longer stand for specific colors as in RGB input; rather, they stand for *filters*.
- Filters encode specific aspects of the input data. At a high level (higher layers), a single filter could encode the concept "presence of a face in the input," for instance.
- At the lower layer, the initial layers, filters encode simple edges or simple color patches.

Feature (Response) Map

- In the MNIST example, the first convolution layer takes an image (a feature map) of size (28, 28, 1) and outputs a feature map of size (26, 26, 32): it computes 32 filters over its input. Each of these 32 output channels contains a 26 × 26 grid of values. Those values contain a *response map* of one filter over the input. The response map is generated as we move the filter pattern over different locations in the input.
- Every 2D tensor along the depth axis of a Conv layer is a feature map giving us the 2D spatial *map* of the response of one filter over the input.
- ***Response Map***: a 2D map quantifying the presence of a pattern at different locations in an input:
- Original Input Filter Response Map



Key Parameters of Convolution Layer

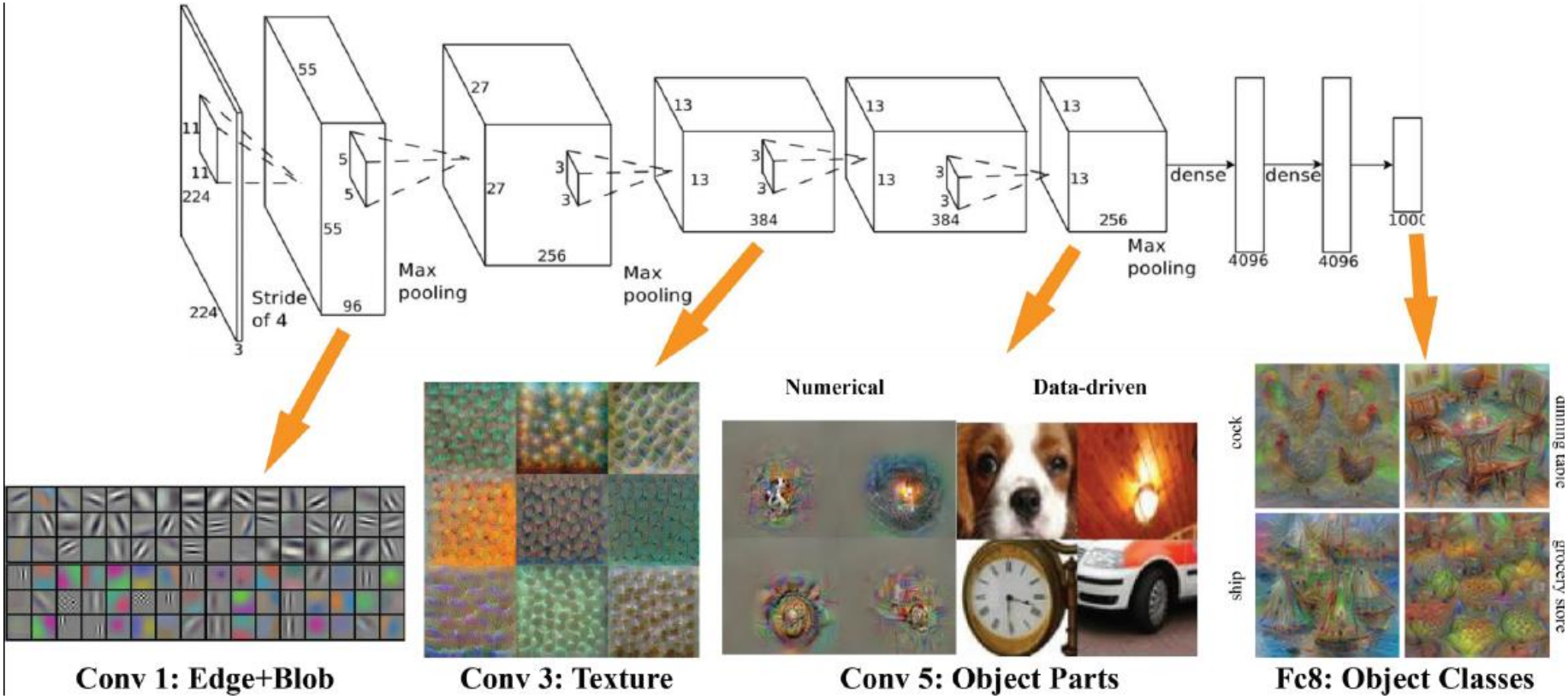
- Convolutions are defined by two key parameters:
  - *Size of the patches extracted from the inputs*—These are typically 3 × 3 or 5 × 5.
  - 3 × 3 is the most common choice.
  - *Depth of the output feature map*—The number of filters computed by the convolution. Our example started with a depth of 32 and ended with a depth of 64.
- In Keras Conv2D layers, these parameters are the first arguments passed to the layer:

```
Conv2D(output_depth, (window_height, window_width)).
```

- A convolution works by *sliding* these windows of size 3 × 3 or 5 × 5 over the 3D input feature map, stopping at every possible location, and extracting the 3D patch of surrounding features (`shape (window_height, window_width, input_depth)).`
- Each such 3D patch is then transformed (via a tensor product with the same learned weight matrix, called the *convolution kernel*) into a 1D vector of shape `(output_depth,)`. All of these vectors are then spatially reassembled into a 3D output map of shape (`height, width, output_depth`).
- 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).
- For instance, with 3 × 3 windows, the vector output[i, j, :] comes from the 3D patch input[i-1:i+1, j-1:j+1, :]. The full process is detailed on the next slide.

AlexNet

- One very popular network and the first very successful convolutional network is AlexNet.
- Its architecture and gradual extraction of the hierarchy of features is presented below.



- Alex Krizhevsky et al. 2012. ImageNet classification with deep convolutional networks

Callbacks

CallBacks

Keras provides us with a very nice way of introspecting a model that is training and also in some cases allows us ot change the hyperparameters at runtime. This function is provided by the Callbacks API.

A callback is an object that can perform actions at various stages of training (e.g. at the start or end of an epoch, before or after a single batch, etc).

Available callbacks

- Base Callback class
- ModelCheckpoint
- BackupAndRestore
- TensorBoard
- EarlyStopping
- LearningRateScheduler
- ReduceLROnPlateau
- RemoteMonitor
- LambdaCallback
- TerminateOnNaN
- CSVLogger
- ProgbarLogger
- SwapEMAWeights

Callbacks

The code is simple to write, you define a callback object and reference it in the fit method

```
callback_es = keras.callbacks.EarlyStopping(monitor='loss', patience=3)
        # This callback will stop the training when there is no improvement in
        # the loss for three consecutive epochs.

model = keras.models.Sequential([keras.layers.Dense(10)])
model.compile(keras.optimizers.SGD(), loss='mse')
history = model.fit(np.arange(100).reshape(5, 20), np.zeros(5),
            epochs=10, batch_size=1, callbacks=[callback_es],
            verbose=0)
```

`keras.layers.Conv2D()` API Documentation

keras.layers.Conv2D()

- On earlier slides we introduced 2D convolution layer (e.g., spatial convolution over images). This layer creates a convolution kernel that scans layer's input to produce a tensor of outputs. The API signature of `Conv2D` reads:

```
keras.layers.Conv2D(
        filters, kernel_size, strides=(1, 1), padding='valid',
        data_format=None, dilation_rate=(1, 1), activation=None,
        use_bias=True, kernel_initializer='glorot_uniform',
        bias_initializer='zeros', kernel_regularizer=None,
        bias_regularizer=None, activity_regularizer=None,
        kernel_constraint=None, bias_constraint=None, **kwargs)
```

- Values after the `=` sign, are the defaults. If `use_bias` is `True`, a bias vector is created and added to the outputs. If `activation` is not `None`, an activation function is applied to the outputs as well.
- When using this layer as the first layer in a model, provide the argument `input_shape (tuple of integers)`, which does not include the batch axis. For example: `input_shape=(128, 128, 3)` for `128x128 RGB` pictures in `data_format="channels_last"`.
- Parameter `output_shape` could also be included. It is a
  - 4D tensor with shape: `(batch, filters, new_rows, new_cols)` if `data_format` is `"channels_first"` or
  - 4D tensor with shape: `(batch, new_rows, new_cols, filters)` if `data_format` is `"channels_last"`. `rows and cols` values might have changed due to `padding`.

`keras.layers.Conv2D()`, Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
- **kernel_size**: An integer or tuple/list of 2 integers, specifying the height and width of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides**: An integer or tuple/list of 2 integers, specifying the steps with which filter shifts in each step along the height and width. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate value != 1`.
- **padding**: one of `"valid" or "same"` (case-insensitive). Note that `"same"` is slightly inconsistent across backends with strides `!= 1`
- **data_format**: A string, one of `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape (`batch, height, width, channels`) while `"channels_first"` corresponds to inputs with shape (`batch, channels, height, width`). It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.

`keras.layers.Conv2D()`, Arguments

- **dilation_rate**: an integer or tuple/list of 2 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any `dilation_rate` value `!= 1` is incompatible with specifying `any_stride` value `!= 1`.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (i.e. "linear" activation, $a(x) = x$, is used).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the kernel weights matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the kernel weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation").
- **kernel_constraint**: Constraint function applied to the kernel matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).