WEEK 09 - DEEP LEARNING II

CONVOLUTIONAL NEURAL NETWORKS

Thakshila Dasun
BSc. Hons in Mechanical Engineering
(Mechatronics Specialization)
CIMA, UK
Academy of Innovative Education

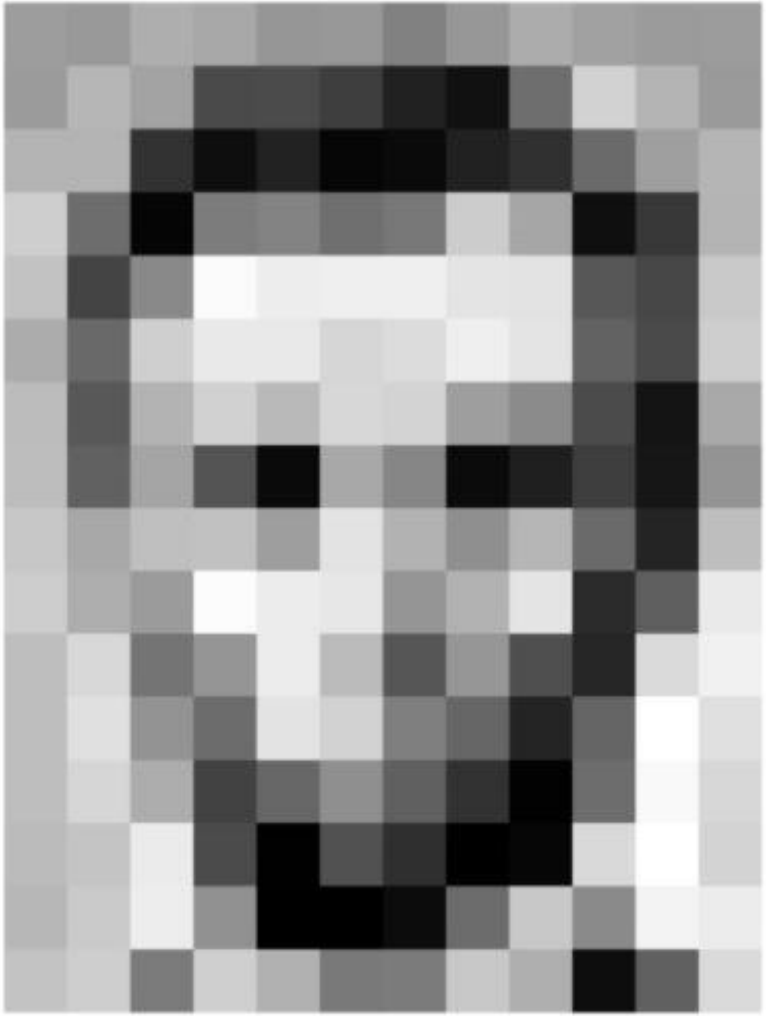
CONVOLUTIONAL NEURAL NETWORKS (CNN)

"Convolutional Neural Networks are designed to address image recognition systems and classification problems. Convolutional Neural Networks have wide applications in image and video recognition, recommendation systems and natural language processing"

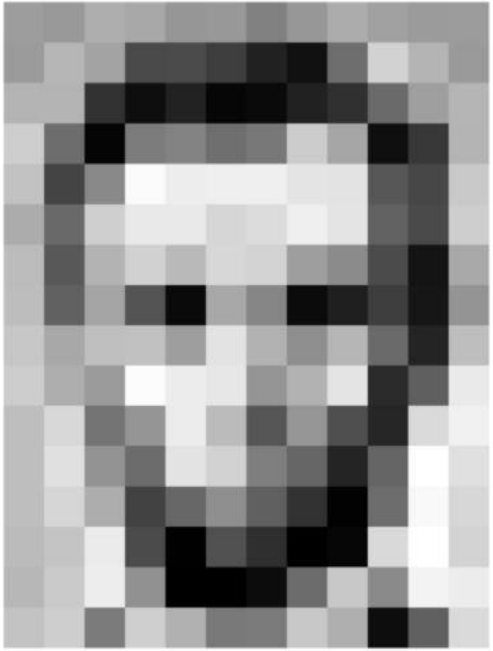
Images in Computers

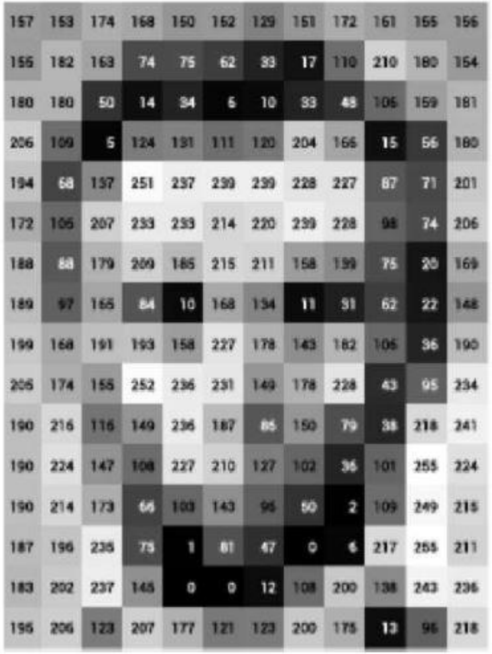
Neurons in our Visual Cortex,

1. Training data of 5000 million years
2. Responds to Spatial Invariant Features



Images are Numbers





What the computer sees

157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

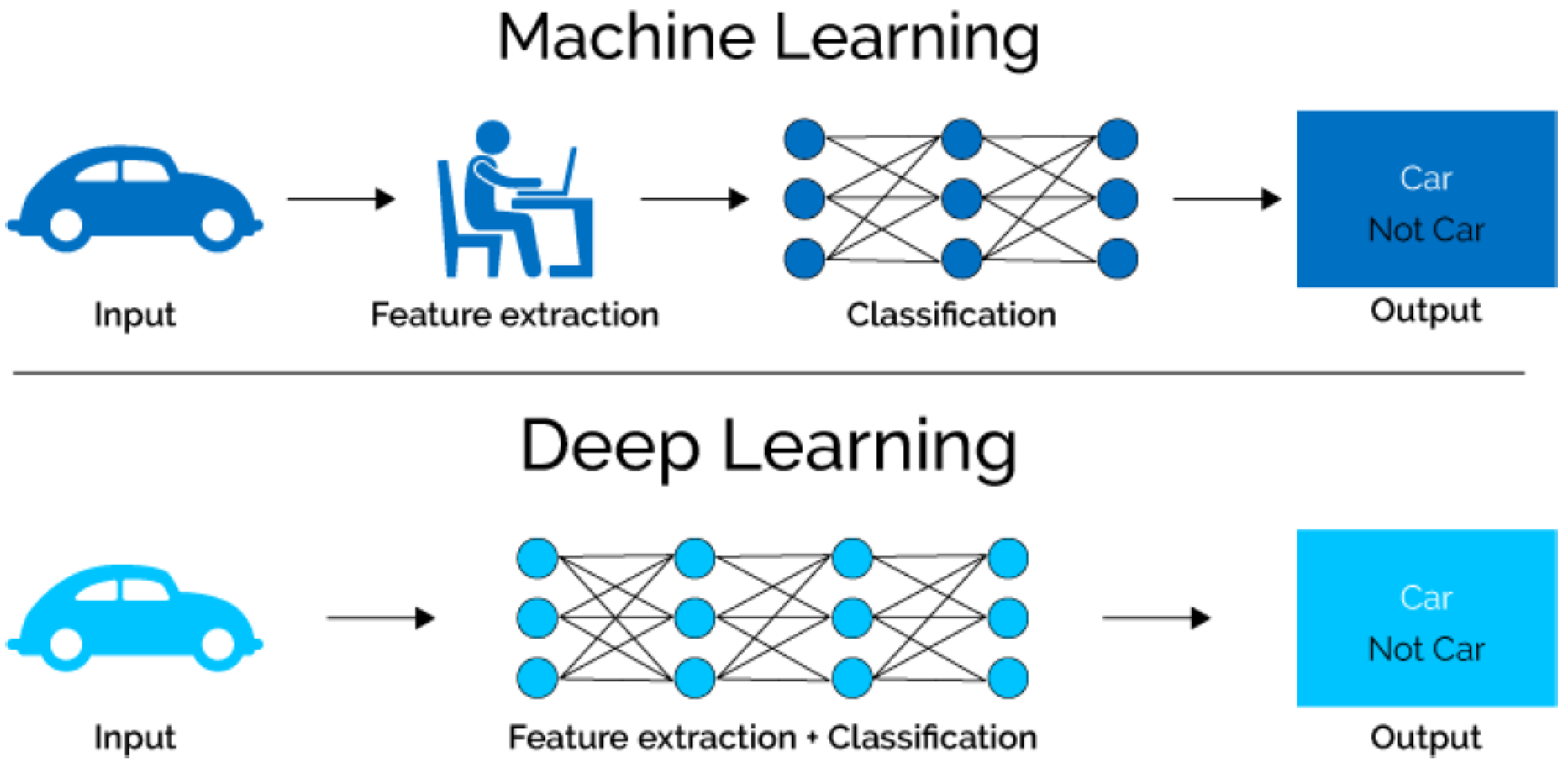
An image is just a matrix of numbers [0,255]!
i.e., 1080x1080x3 for an RGB image

Let's identify key features in each image category

Nose,
Eyes,
Mouth

Wheels,
License Plate,
Headlights

Door,
Windows,
Steps


Machine Learning
Input
Feature extraction
Classification
Car
Not Car
Output
Deep Learning
Input
Feature extraction + Classification
Car
Not Car
Output

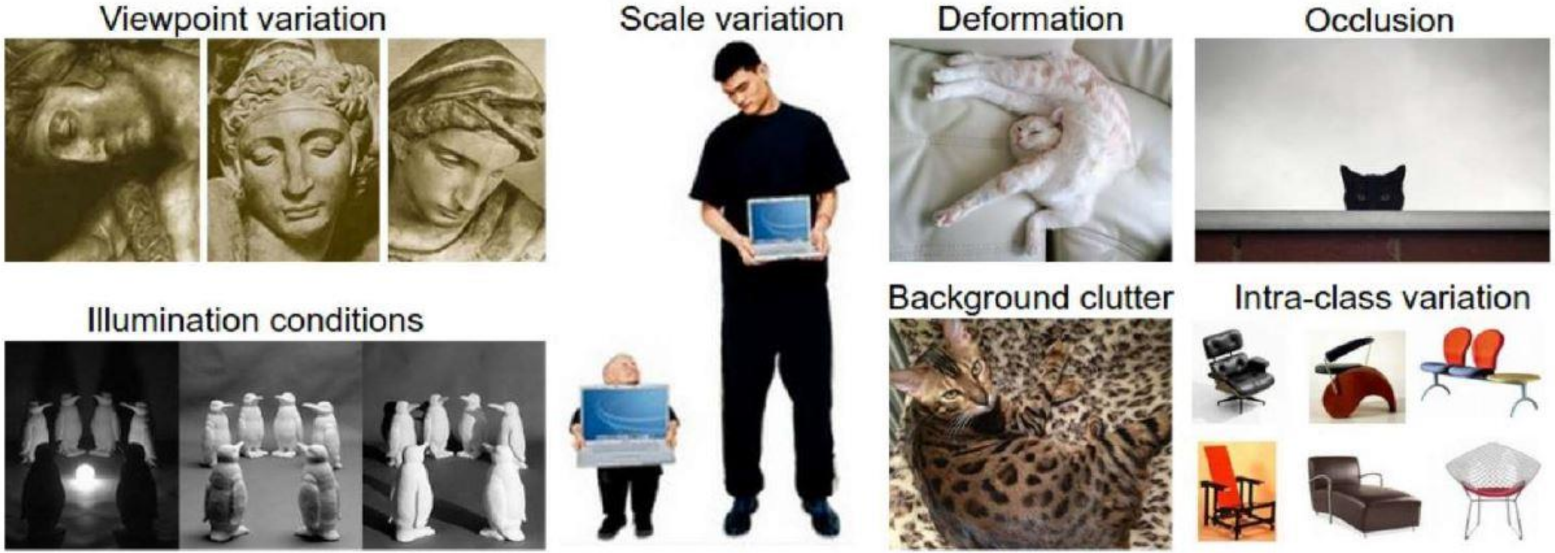
Manual Feature Extraction

Domain knowledge → Define features → Detect features to classify

Problems?

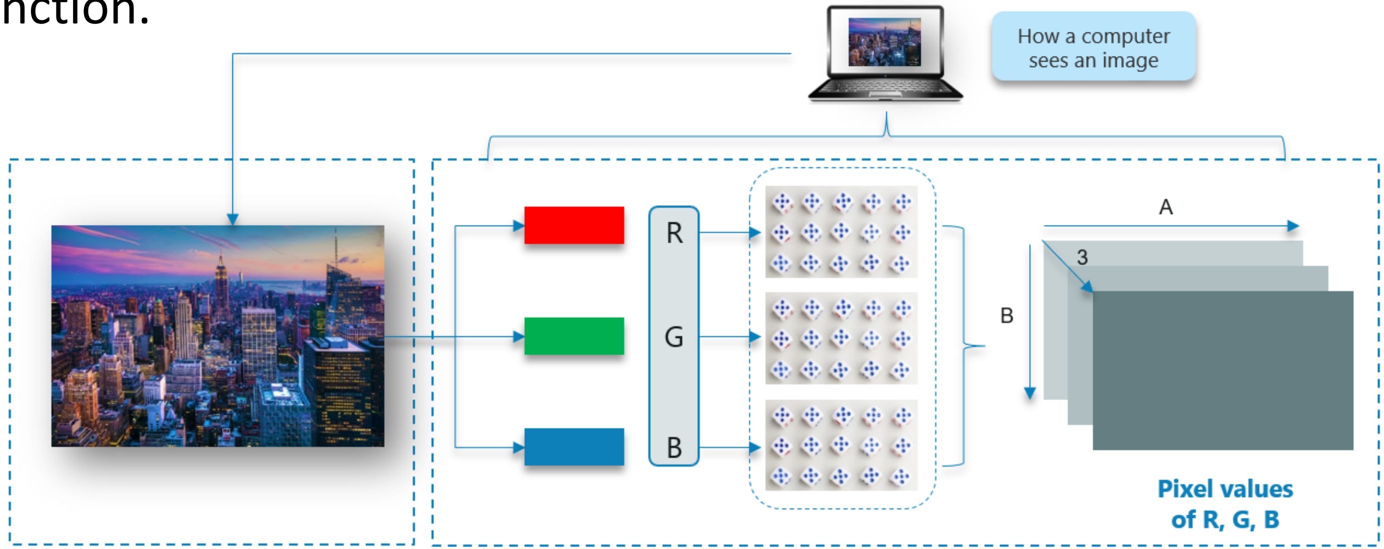
Manual Feature Extraction

Domain knowledge → Define features → Detect features to classify

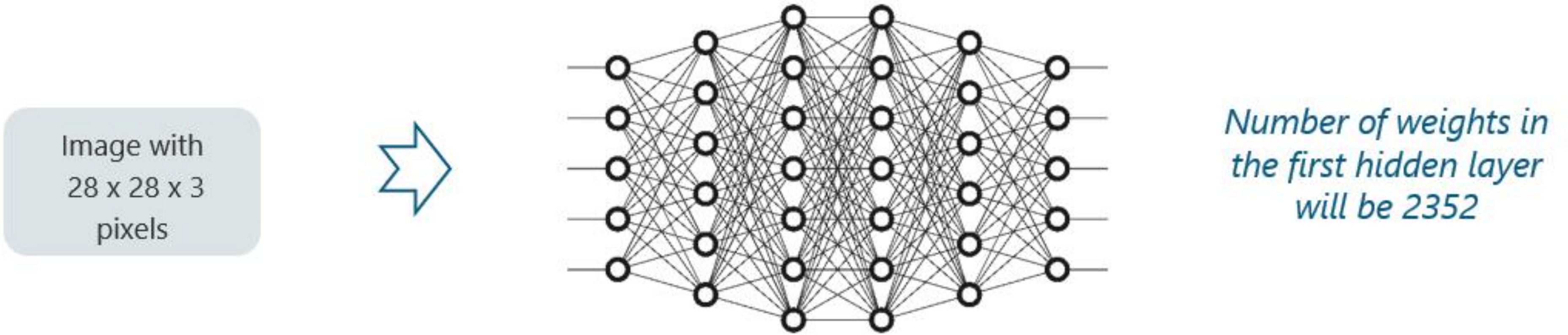


Images in Computers

- The image is broken down into 3 color-channels which is Red, Green and Blue. Each of these color channels are mapped to the image's pixel.
- Idea of neural networks began unsurprisingly as a model of how neurons in the brain function.

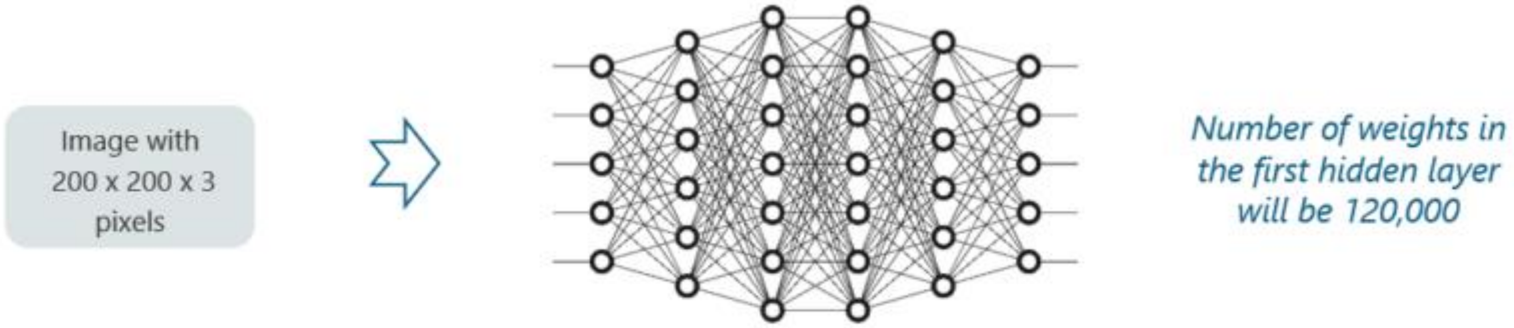


Why Not Fully Connected FFNNs?



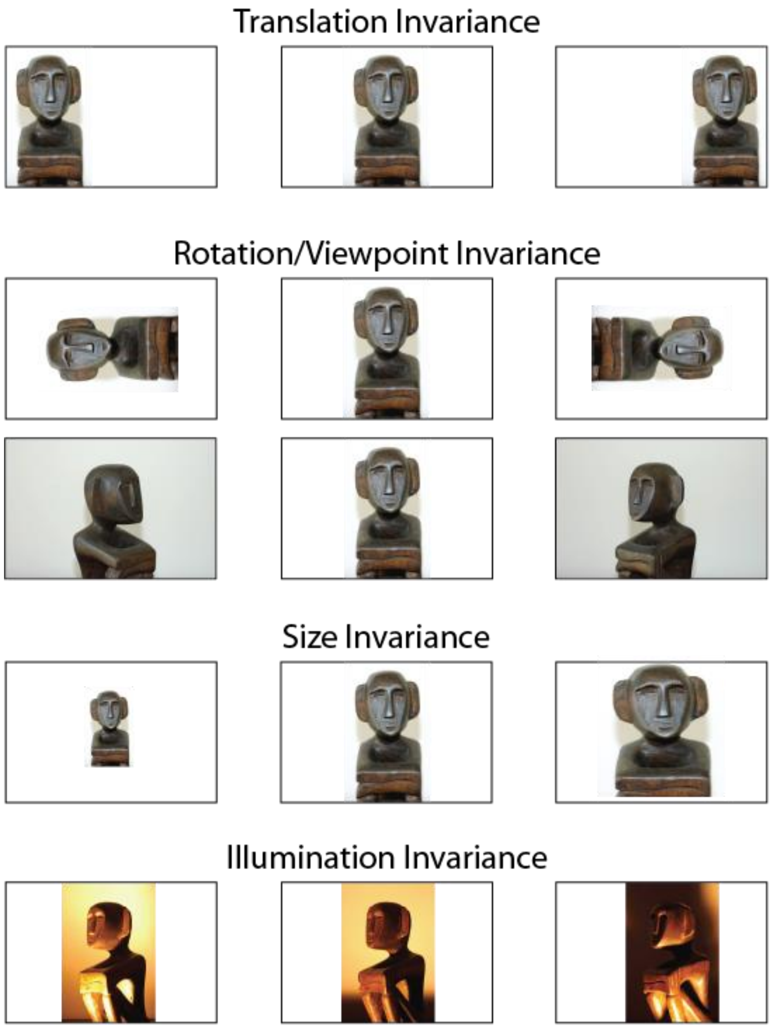
- **Consider an input** of images with the size **28x28x3** pixels. If we **input** this to our Convolutional Neural Network,
- we will have about **2352 weights** in the **first** hidden layer itself.

Why Not Fully Connected FFNNs?



- Any **generic** input **image** will **at least** have **200x200x3 pixels** in size.
- The size of the first hidden layer becomes a **whooping 120,000**.
- If this is just the **first** hidden layer, imagine the **number of neurons** needed to process an **entire** complex **image-set.**

Why Not
Fully Connected FFNNs?



Convolutional Neural Networks (1)

- Convolutional Neural Networks, like FFNNs, are made up of **neurons** with **learnable weights** and **biases**.
- Each **neuron** receives several **inputs**, takes a weighted **sum** over them, **pass** it through an **activation function** and responds with an **output**.
- The whole network has a **loss function** and all the tips and tricks that we developed for neural networks still apply on **Convolutional Neural Networks.**

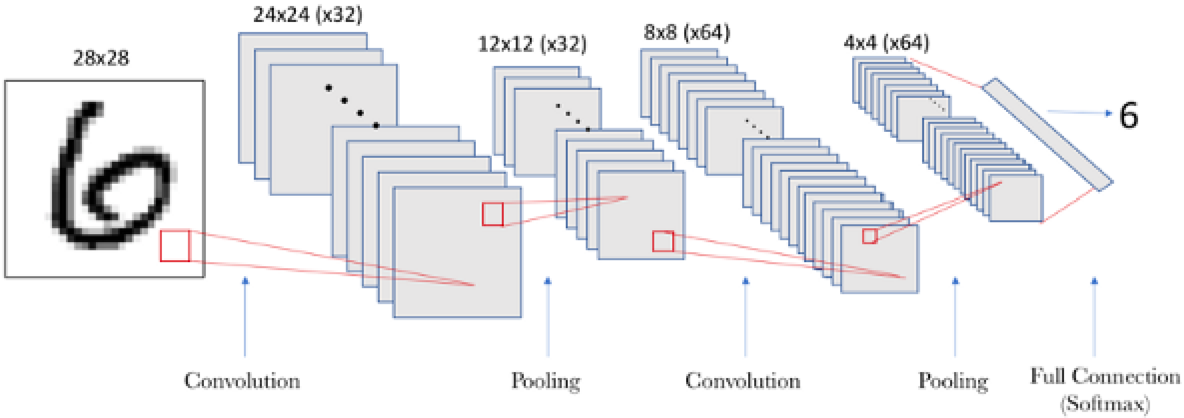
Convolutional Neural Networks (2)

- Let's take the example of **automatic image recognition.** The process of **determining** whether a **picture**contains a **cat** involves an **activation function**. If the picture resembles prior cat images the neurons have **seen before,** the label **"cat"** would be **activated.**
- **Hence,** the **more** labeled images the neurons are **exposed** to, the **better** it learns how to recognize other unlabelled images. We call this the process of **training** neurons.

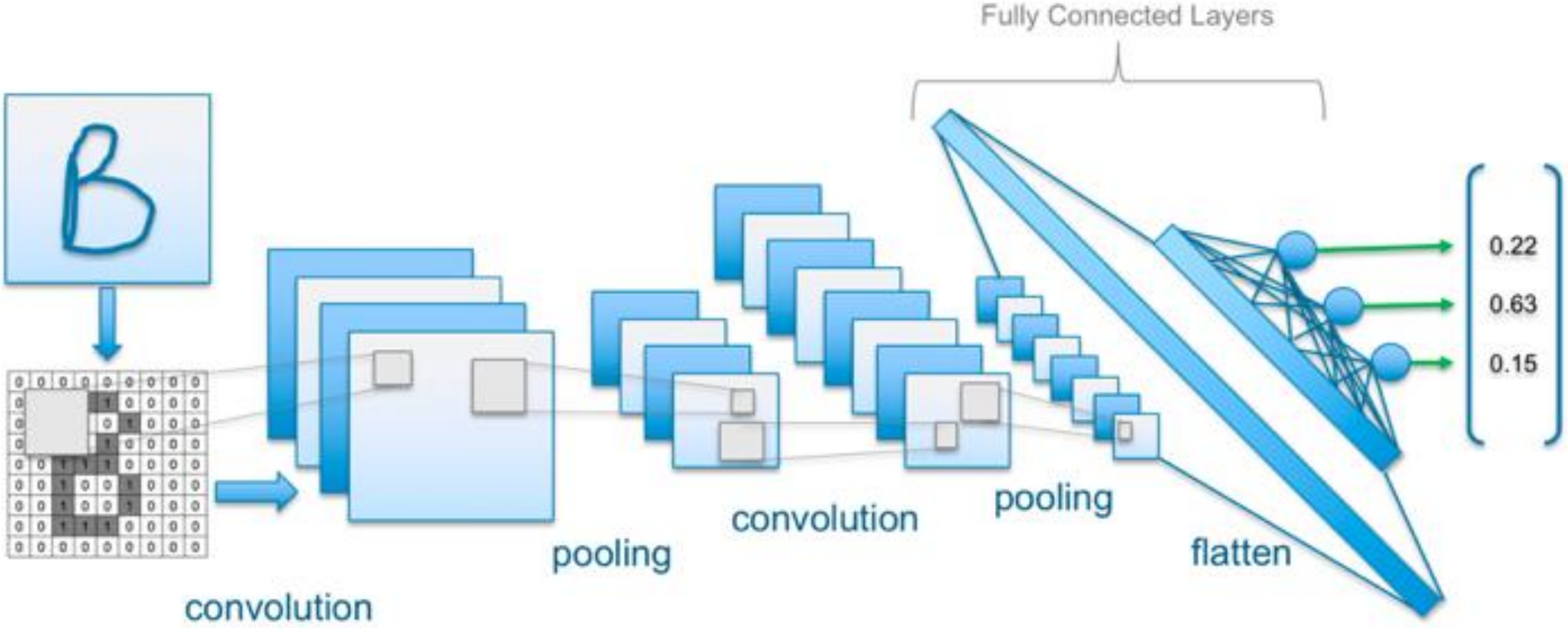
How Do Convolutional Neural Networks Work?

There are **four** layered **concepts** we should understand in Convolutional Neural Networks:

1. Convolution
2. ReLu
3. Pooling
4. Dense



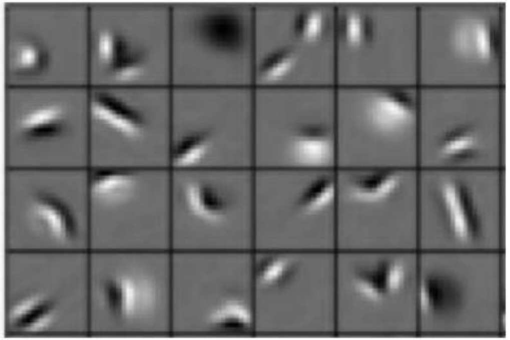
How Do Convolutional Neural Networks Work?



Learning Feature Representations

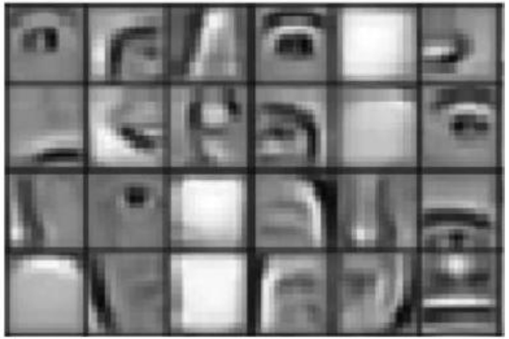
Can we learn a **hierarchy of features** directly from the data instead of hand engineering?

Low level features



Edges, dark spots

Mid level features

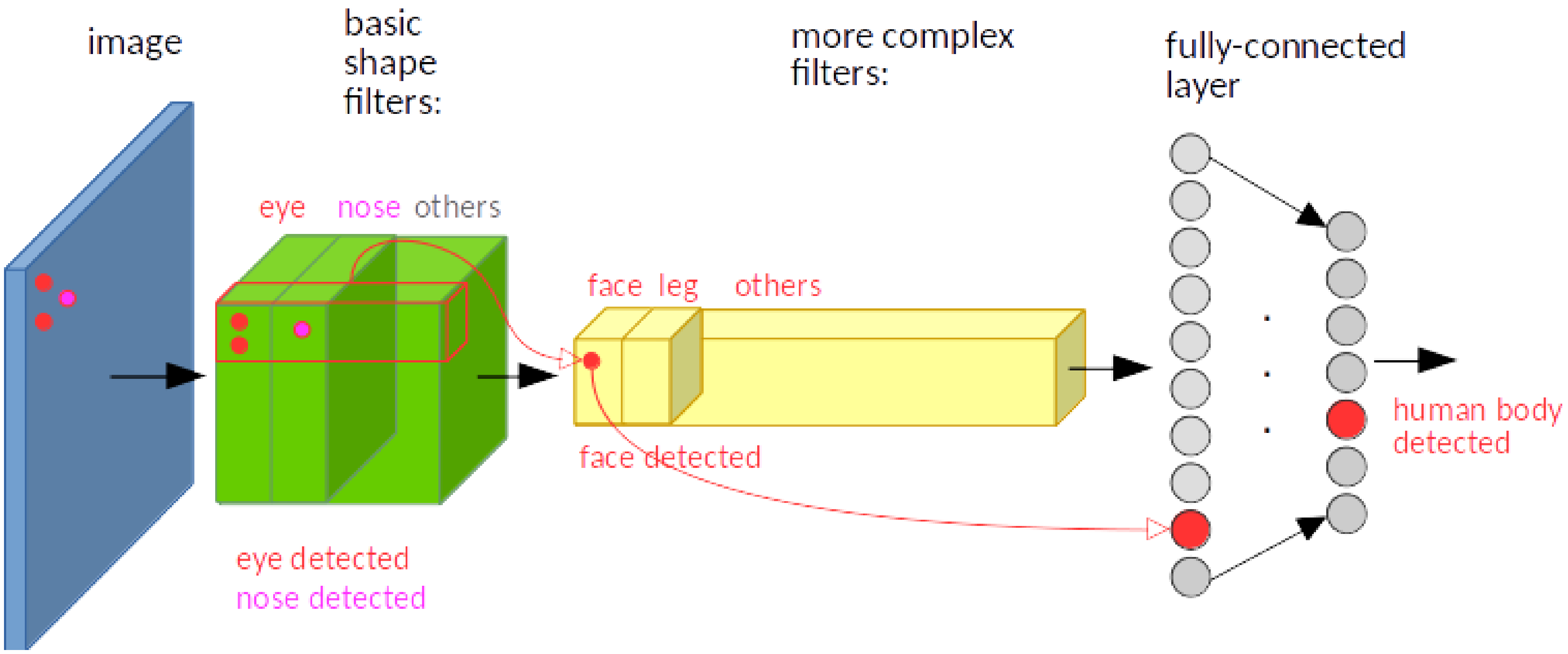


Eyes, ears, nose

High level features

Facial structure

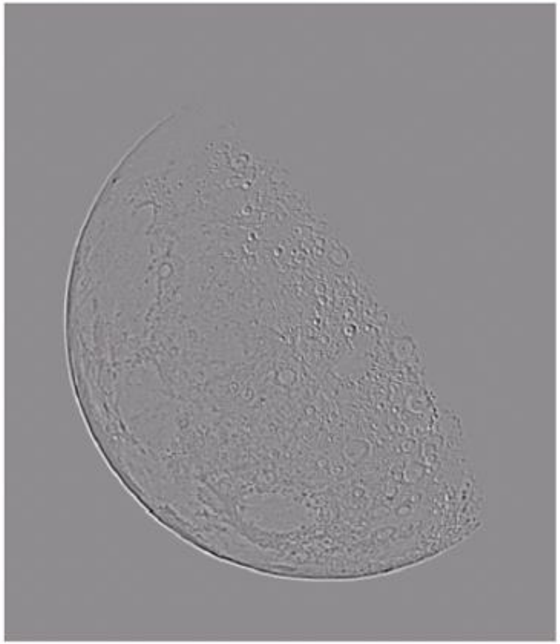
How Do Convolutional Neural Networks Work?



Idea of Convolution

Original
Image

Laplacian
Filtered Image



Laplacian
Filtered Image
Scaled for Display

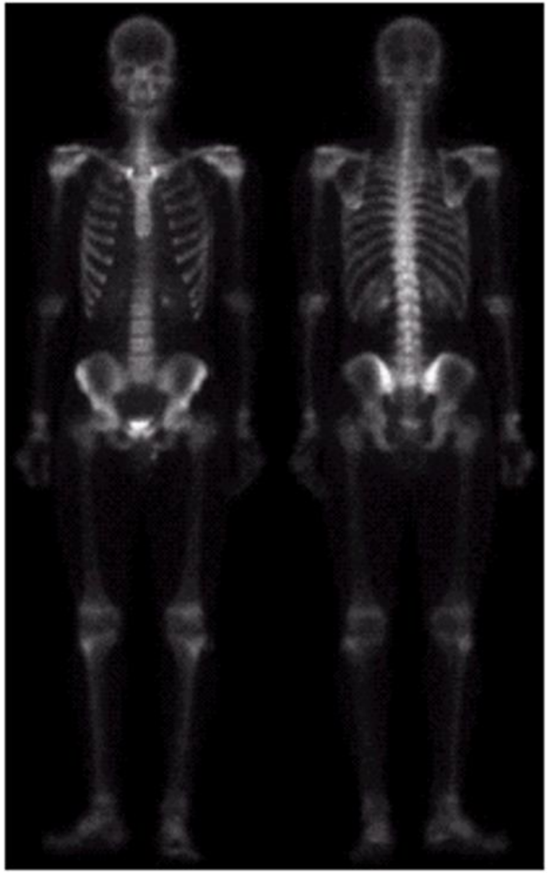
Idea of Convolution

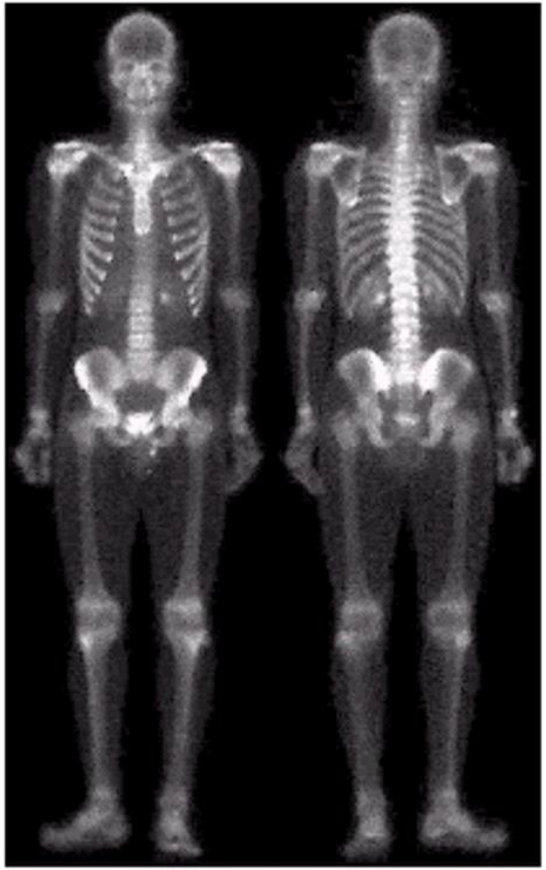
Original
Image

0	1	0
1	-4	1
0	1	0

Laplacian
Filtered Image

Idea of Convolution





Let’s consider an Example: Sobel Gradients

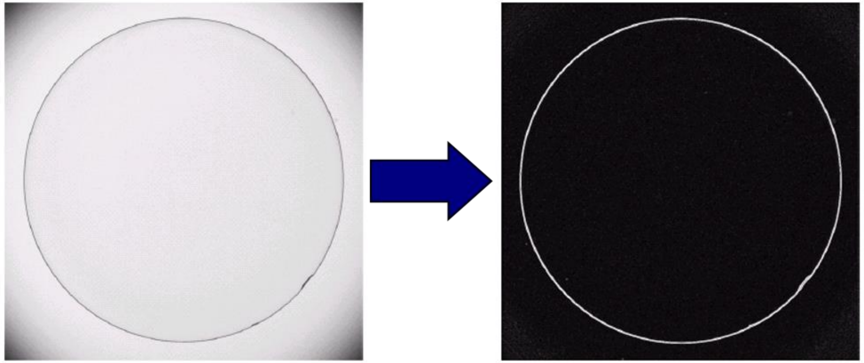
- The Sobel operator performs a 2-D spatial gradient measurement on an image and so emphasizes regions of high spatial frequency that correspond to edges. Typically it is used to find the approximate absolute gradient magnitude at each point in an input grayscale image.
- One kernel is simply the other rotated by 90°.

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy



Idea of Convolution

Original Image

0	1	0
1	-4	1
0	1	0

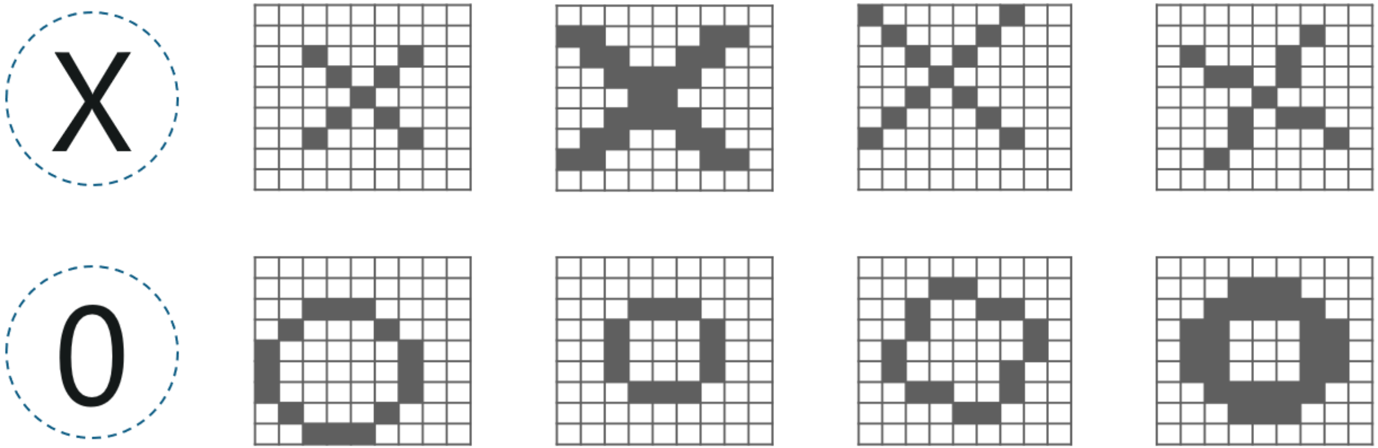
Laplacian Filtered Image

3_0	3_1	2_2	1	0
0_2	0_2	1_0	3	1
3_0	1_1	2_2	2	3
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

How Do Convolutional Neural Networks Work?

- Here, there are multiple renditions of X and O's. This makes it tricky for the computer to recognize.
- But the goal is that if the **input signal** looks like **previous** images it has seen before, the **"image" reference** signal will be mixed into, or **convolved** with, the **input** signal. The resulting **output** signal is then passed on to the **next layer.**



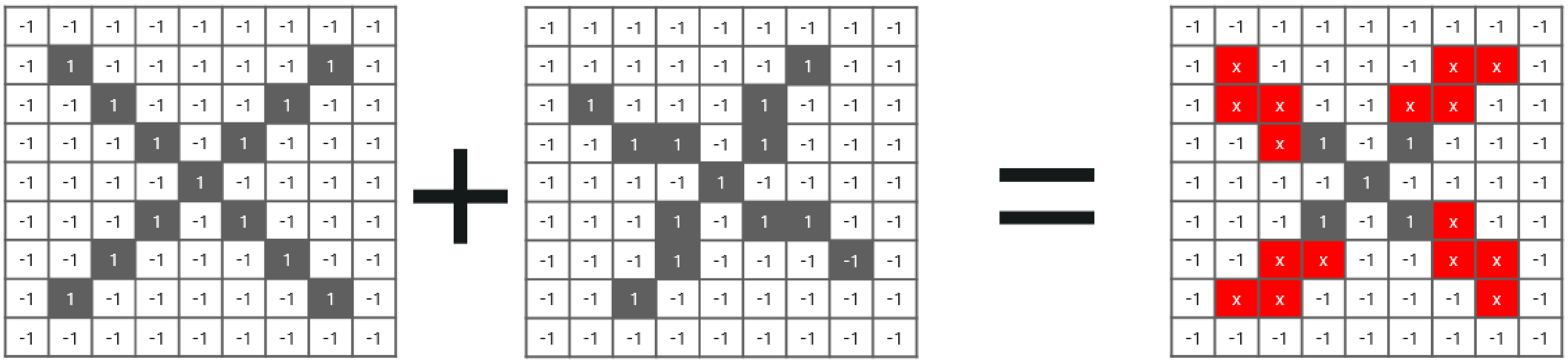
How Do Convolutional Neural Networks Work?

- the **computer understands** every pixel. In this case, the **white** pixels are said to be **-1** while the**black** ones are **1.**
- This is just the way we’ve implemented to **differentiate the pixels** in a basic binary classification.

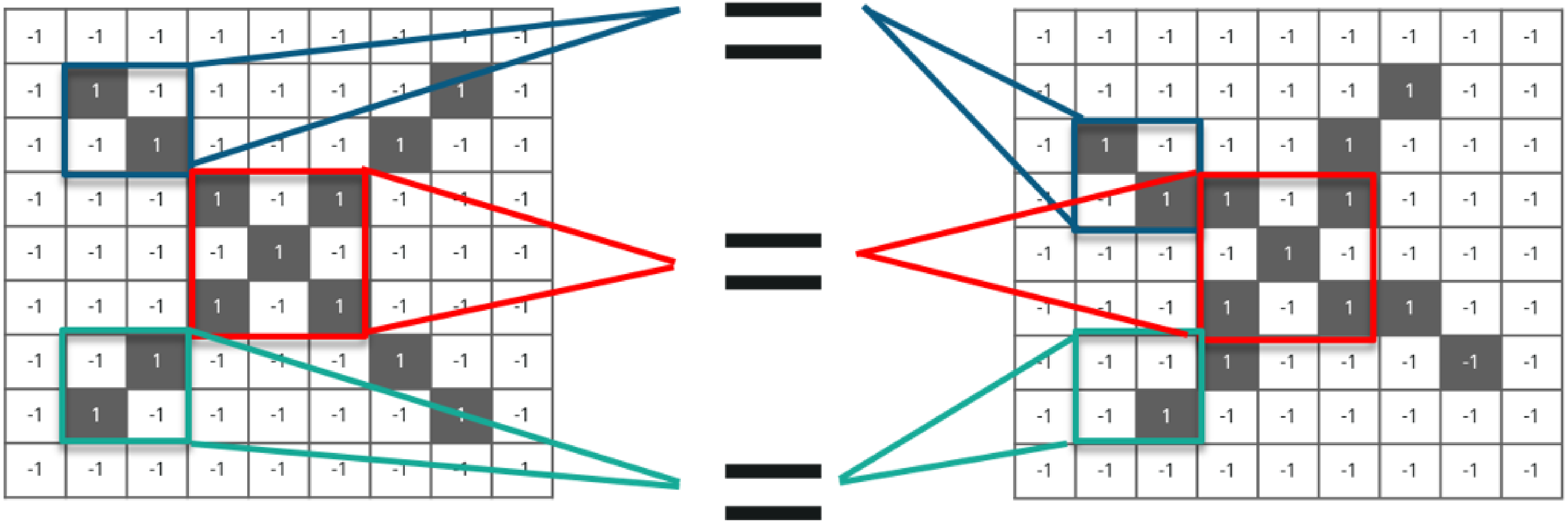
-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	-1	-1	1	-1	-1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1

How Do Convolutional Neural Networks Work?

- Now if we would just **normally search** and **compare** the **values** between a normal image and another **‘x’ rendition,** we would get a **lot** of **missing pixels.**

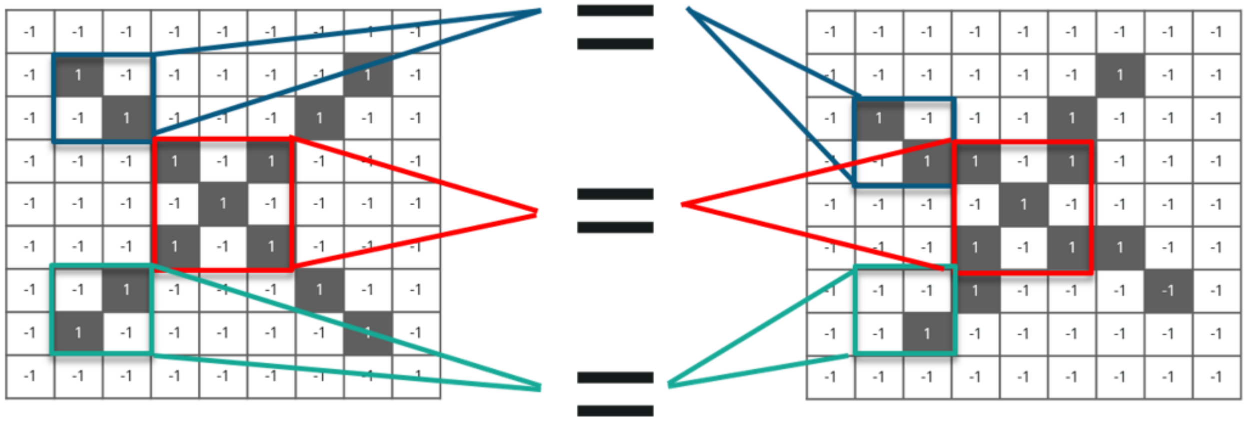


So, how do we fix this?



Attention! This is Tricky.

- We take **small patches** of the pixels called **filters** and try to **match** them in the corresponding **nearby** locations to see if we get a **match.**
- By doing this, the Convolutional Neural Network **gets a lot better** at seeing **similarity** than directly trying to match the **entire image.**

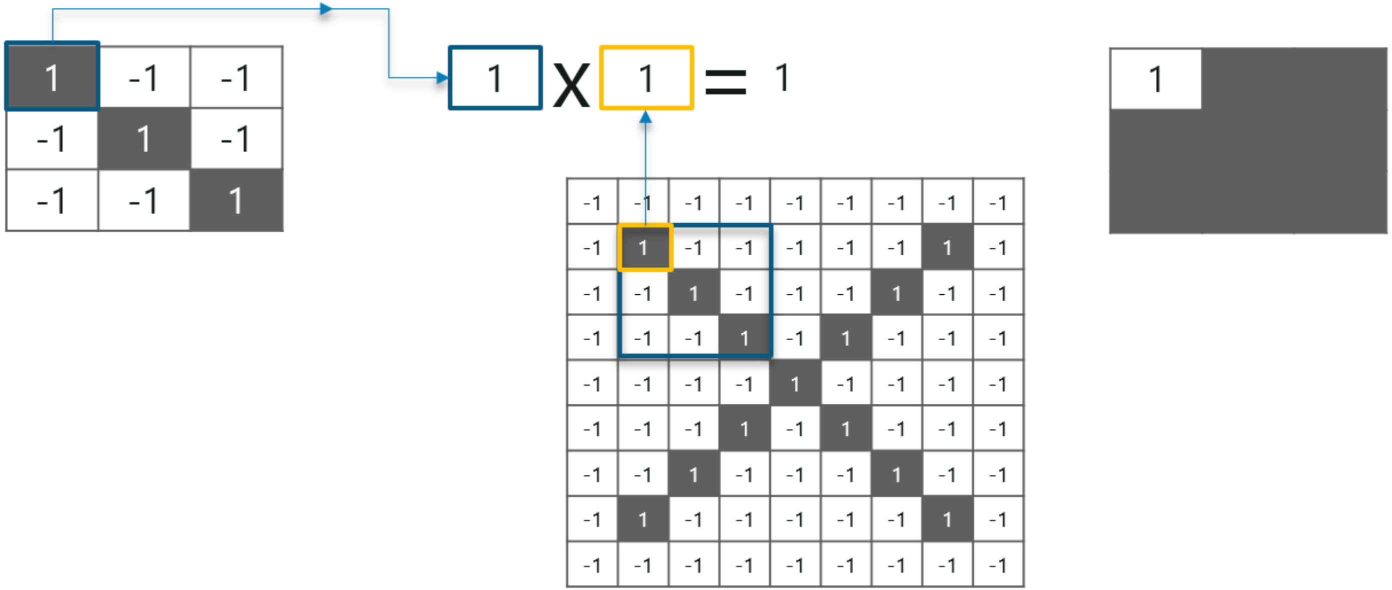


Convolution Of An Image

- Convolution has the nice property of being **translational invariant**.
- Intuitively, this means that **each** convolution filter represents a **feature** of interest (e.g **pixels in letters)** and the Convolutional Neural Network **algorithm** learns which **features** comprise the **resulting reference** (i.e. alphabet).

- We have **4 steps** for convolution:
  1. **Line up** the feature and the image
  2. **Multiply** each **image** pixel by corresponding **feature** pixel
  3. **Add** the values and find the **sum**
  4. **Divide** the sum by the **total** number of pixels in the **feature**

Convolution Of An Image



- Consider the above image – As you can see, we are **done** with the first **2 steps**. We considered a **feature image** and **one pixel** from it. We **multiplied** this with the **existing image** and the product is stored in another **buffer feature image**.

Convolution Of An Image

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

$$\frac{1+1+1+1+1+1+1+1+1}{9} = 1$$

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

1	1	1
1	1	1
1	1	1

- With this **image,** we completed the **last 2 steps.** We added the **values** which led to the **sum.**
- We then, **divide** this **number** by the **total** number of pixels in the **feature image.**
- When that is done, the **final value** obtained is placed at the **center** of the **filtered image** as shown below:

Convolution Of An Image

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

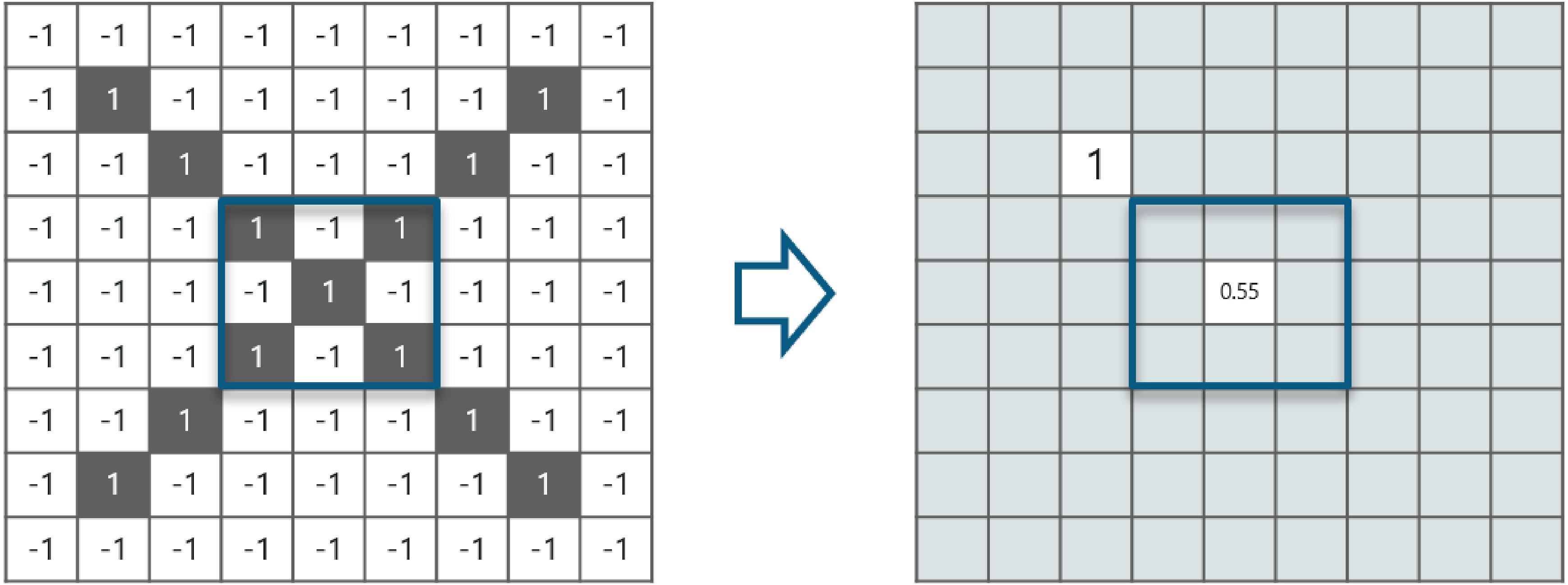
$$\frac{1+1-1+1+1+1-1+1+1}{9} = .55$$

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

1	1	-1
1	1	1
-1	1	1

- Now, we can **move** this **filter** around and do the **same** at **any pixel** in the image. For **better clarity,** let’s consider **another example:**

Convolution Of An Image



- Similarly, we move the feature to every other position in the image and see how the feature matches that area. So after doing this, we will get the output as:

Convolution Of An Image

0.77	-0.11	0.11	0.33	0.55	-0.11	0.33
-0.11	1.0	-0.11	0.33	-0.11	0.11	-0.11
0.11	-0.11	1.0	-0.33	0.11	-0.11	0.55
0.33	0.33	-0.33	0.55	-0.33	0.33	0.33
0.55	-0.11	0.11	-0.33	1.00	-0.11	0.11
-0.11	0.11	-0.11	0.33	-0.11	1.00	-0.11
0.33	-0.11	0.55	0.33	0.11	-0.11	0.77

Convolution Of An Image

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

$$\frac{1+1+1+1+1+1+1+1+1}{9} = 1$$

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

1	1	1
1	1	1
1	1	1

- With this **image,** we completed the **last 2 steps.** We added the **values** which led to the **sum.**
- We then, **divide** this **number** by the **total** number of pixels in the **feature image.**
- When that is done, the **final value** obtained is placed at the **center** of the **filtered image** as shown below:

Convolution Of An Image

- Here we considered just one filter. Similarly, we will perform the same convolution with every other filter to get the convolution of that filter.
- The **output** signal **strength** is not dependent on where the **features** are located, but simply whether the **features** are **present.** Hence, an alphabet could be sitting in **different positions** and the **Convolutional Neural Network** algorithm would still be able to **recognize it.**

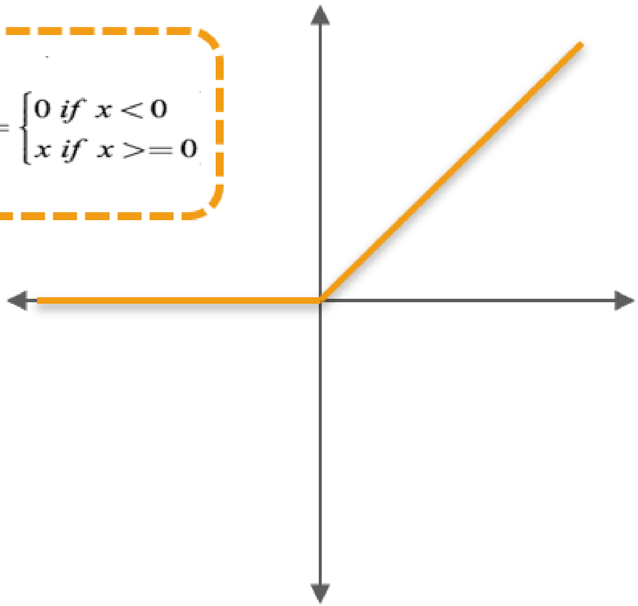
Convolution Of An Image

- Here we considered just one filter. Similarly, we will perform the same convolution with every other filter to get the convolution of that filter.
- The **output** signal **strength** is not dependent on where the **features** are located, but simply whether the **features** are **present.** Hence, an alphabet could be sitting in **different positions** and the **Convolutional Neural Network** algorithm would still be able to **recognize it.**

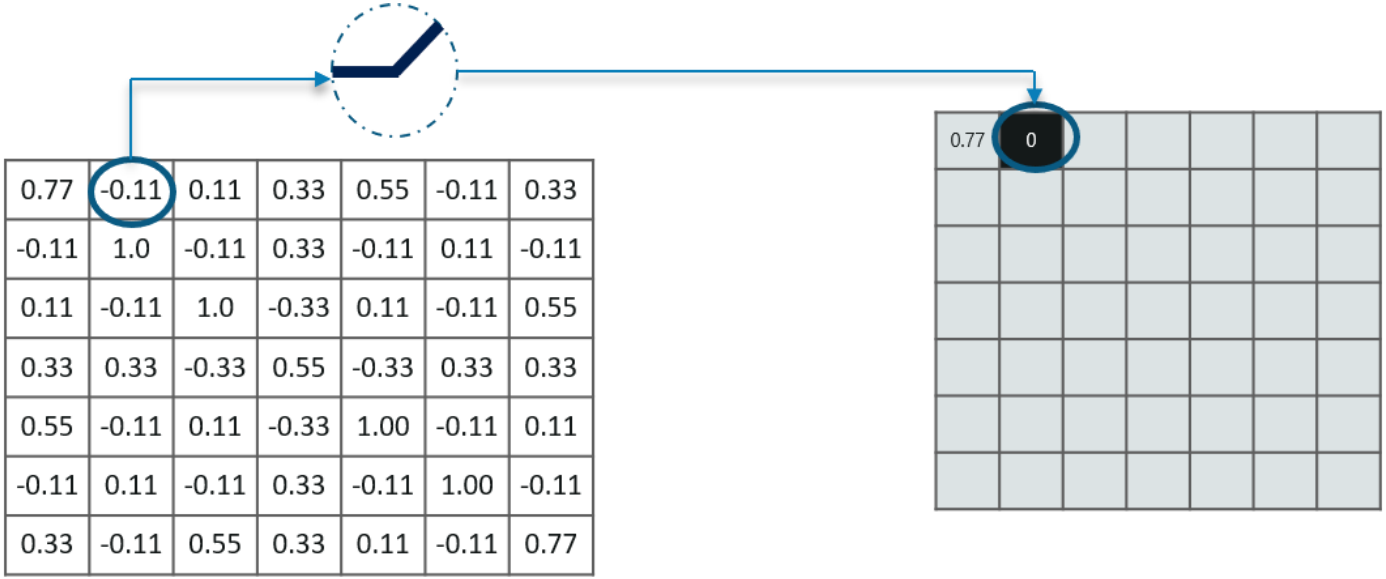
ReLU Layer

- **Rectified Linear Unit** (ReLU) transform function only activates a node if the input is above a certain quantity, while the input is below zero, the output is zero, but when the input rises above a certain threshold, it has a linear relationship with the dependent variable.

$$f(x) = \begin{cases} 0 \ \text{if} \ x < 0 \\ x \ \text{if} \ x >= 0 \end{cases}$$



Why do we require ReLU here?



0.77	-0.11	0.11	0.33	0.55	-0.11	0.33
-0.11	1.0	-0.11	0.33	-0.11	0.11	-0.11
0.11	-0.11	1.0	-0.33	0.11	-0.11	0.55
0.33	0.33	-0.33	0.55	-0.33	0.33	0.33
0.55	-0.11	0.11	-0.33	1.00	-0.11	0.11
-0.11	0.11	-0.11	0.33	-0.11	1.00	-0.11
0.33	-0.11	0.55	0.33	0.11	-0.11	0.77

- The main aim is to remove all the negative values from the convolution. All the positive values remain the same but all the negative values get changed to zero as shown below:

Why do we require ReLU here?

0.77	-0.11	0.11	0.33	0.55	-0.11	0.33
-0.11	1.0	-0.11	0.33	-0.11	0.11	-0.11
0.11	-0.11	1.0	-0.33	0.11	-0.11	0.55
0.33	0.33	-0.33	0.55	-0.33	0.33	0.33
0.55	-0.11	0.11	-0.33	1.00	-0.11	0.11
-0.11	0.11	-0.11	0.33	-0.11	1.00	-0.11
0.33	-0.11	0.55	0.33	0.11	-0.11	0.77

0.77	0	0.11	0.33	0.55	0	0.33
0	1.00	0	0.33	0	0.11	0
0.11	0	1.00	0	0.11	0	0.55
0.33	0.33	0	0.55	0	0.33	0.33
0.55	0	0.11	0	1.00	0	0.11
0	0.11	0	0.33	0	1.00	0
0.33	0	0.55	0.33	0.11	0	1.77

- So after we process this particular feature we get the following output:

Why do we require ReLU here?

0.77	-0.11	0.11	0.33	0.55	-0.11	0.33
-0.11	1.0	-0.11	0.33	-0.11	0.11	-0.11
0.11	-0.11	1.0	-0.33	0.11	-0.11	0.55
0.33	0.33	-0.33	0.55	-0.33	0.33	0.33
0.55	-0.11	0.11	-0.33	1.00	-0.11	0.11
-0.11	0.11	-0.11	0.33	-0.11	1.00	-0.11
0.33	-0.11	0.55	0.33	0.11	-0.11	0.77

0.77	0	0.11	0.33	0.55	0	0.33
0	1.00	0	0.33	0	0.11	0
0.11	0	1.00	0	0.11	0	0.55
0.33	0.33	0	0.55	0	0.33	0.33
0.55	0	0.11	0	1.00	0	0.11
0	0.11	0	0.33	0	1.00	0
0.33	0	0.55	0.33	0.11	0	1.77

0.33	-0.55	0.11	-0.11	0.11	-0.55	0.33
-0.55	0.55	-0.55	0.33	-0.55	0.55	-0.55
0.11	-0.55	0.55	-0.11	0.55	-0.55	0.11
-0.11	0.33	-0.77	1.00	-0.77	0.33	-0.11
0.11	-0.55	0.55	-0.77	0.55	-0.55	0.11
-0.55	0.55	-0.55	0.33	-0.55	0.55	-0.55
0.33	-0.55	0.11	-0.11	0.11	-0.55	0.33

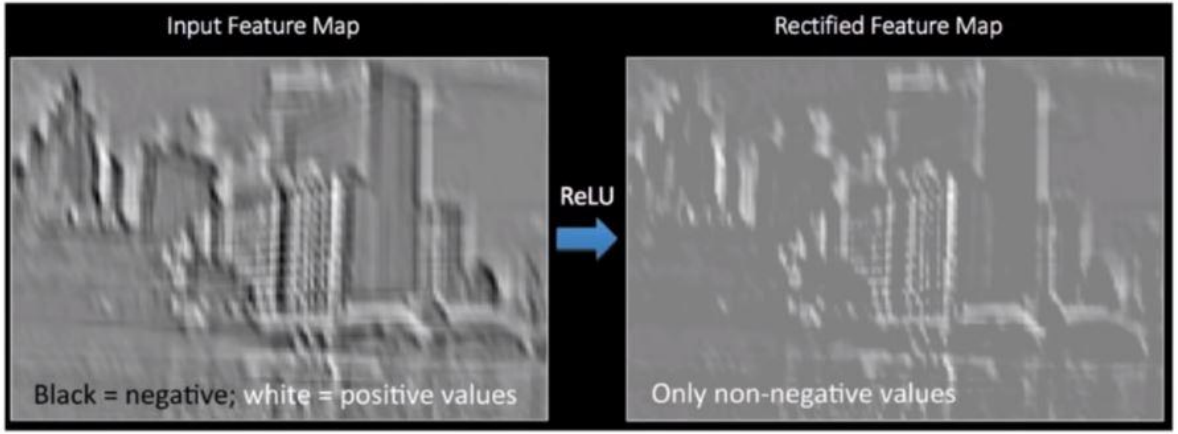
0.33	0	0.11	0	0.11	0	0.33
0	0.55	0	0.33	0	0.55	0
0.11	0	0.55	0	0.55	0	0.11
0	0.33	0	1.00	0	0.33	0
0.11	0	0.55	0	0.55	0	0.11
0	0.55	0	0.33	0	0.55	0
0.33	0	0.11	0	0.11	0	0.33

0.33	-0.11	0.55	0.33	0.11	-0.11	0.77
-0.11	0.11	-0.11	0.33	-0.11	1.00	-0.11
0.55	-0.11	0.11	-0.33	1.00	-0.11	0.11
0.33	0.33	-0.33	0.55	-0.33	0.33	0.33
0.11	-0.11	1.00	-0.33	0.11	-0.11	0.55
-0.11	1.00	-0.11	0.33	-0.11	0.11	-0.11
0.77	-0.11	0.11	0.33	0.55	-0.11	0.33

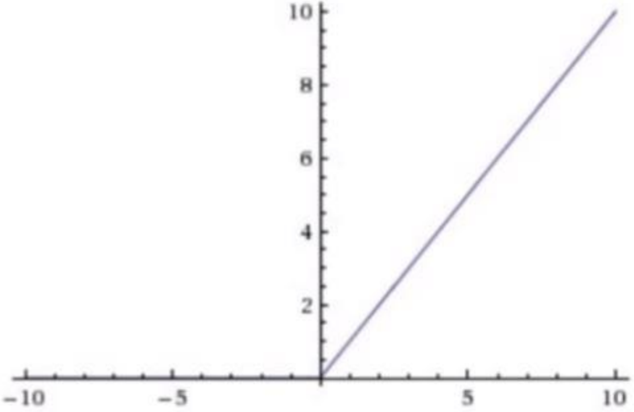
0.33	0	0.55	0.33	0.11	0	0.77
0	0.11	0	0.33	0	1.00	0
0.55	0	0.11	0	1.00	0	0.11
0.33	0.33	0	0.55	0	0.33	0.33
0.11	0	1.00	0	0.11	0	0.55
0	1.00	0	0.33	0	0.11	0
0.77	0	0.11	0.33	0.55	0	0.33

Introducing Non-Linearity

- Apply after every convolution operation (i.e., after convolutional layers)
- ReLU: pixel-by-pixel operation that replaces all negative values by zero. **Non-linear operation!**



Rectified Linear Unit (ReLU)

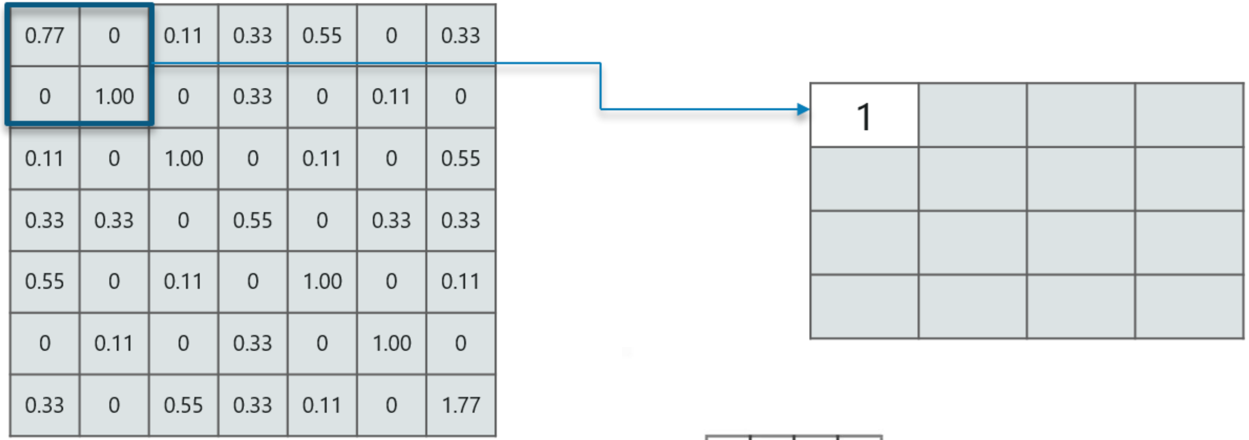


$$g(z) = \max(0, z)$$

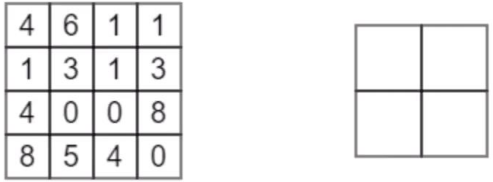
Pooling Layer

- In this layer we **shrink** the **image** stack into a **smaller size.** Pooling is done **after passing** through the **activation** layer. We do this by implementing the following 4 steps:

1. Pick a **window size** (usually 2 or 3)
2. Pick a **stride** (usually 2)
3. **Walk** your window **across** your **filtered** images
4. From each **window,** take the **maximum** value



Methods,
1. Max
2. Min
3. Average etc



Input (4x4)

Output (2x2)

Pooling Layer

0.77	0	0.11	0.33	0.55	0	0.33
0	1.00	0	0.33	0	0.11	0
0.11	0	1.00	0	0.11	0	0.55
0.33	0.33	0	0.55	0	0.33	0.33
0.55	0	0.11	0	1.00	0	0.11
0	0.11	0	0.33	0	1.00	0
0.33	0	0.55	0.33	0.11	0	1.77

1.00	0.33	0.55	0.33
0.33	1.00	0.33	0.55
0.55	0.33	1.00	0.11
0.33	0.55	0.11	0.77

- So in this case, we took **window size** to be **2** and we got **4 values** to choose from. From those 4 values, the **maximum value** there is 1 so we pick 1. Also, note that we **started out** with a **7×7** matrix but now the same matrix after **pooling** came down to **4×4.**
- But we need to **move** the **window across** the **entire** image. The procedure is exactly as same as above and we need to repeat that for the entire image.

Pooling Layer

0.77	0	0.11	0.33	0.55	0	0.33
0	1.00	0	0.33	0	0.11	0
0.11	0	1.00	0	0.11	0	0.55
0.33	0.33	0	0.55	0	0.33	0.33
0.55	0	0.11	0	1.00	0	0.11
0	0.11	0	0.33	0	1.00	0
0.33	0	0.55	0.33	0.11	0	1.77

1.00	0.33	0.55	0.33
0.33	1.00	0.33	0.55
0.55	0.33	1.00	0.11
0.33	0.55	0.11	0.77

0.33	0	0.11	0	0.11	0	0.33
0	0.55	0	0.33	0	0.55	0
0.11	0	0.55	0	0.55	0	0.11
0	0.33	0	1.00	0	0.33	0
0.11	0	0.55	0	0.55	0	0.11
0	0.55	0	0.33	0	0.55	0
0.33	0	0.11	0	0.11	0	0.33

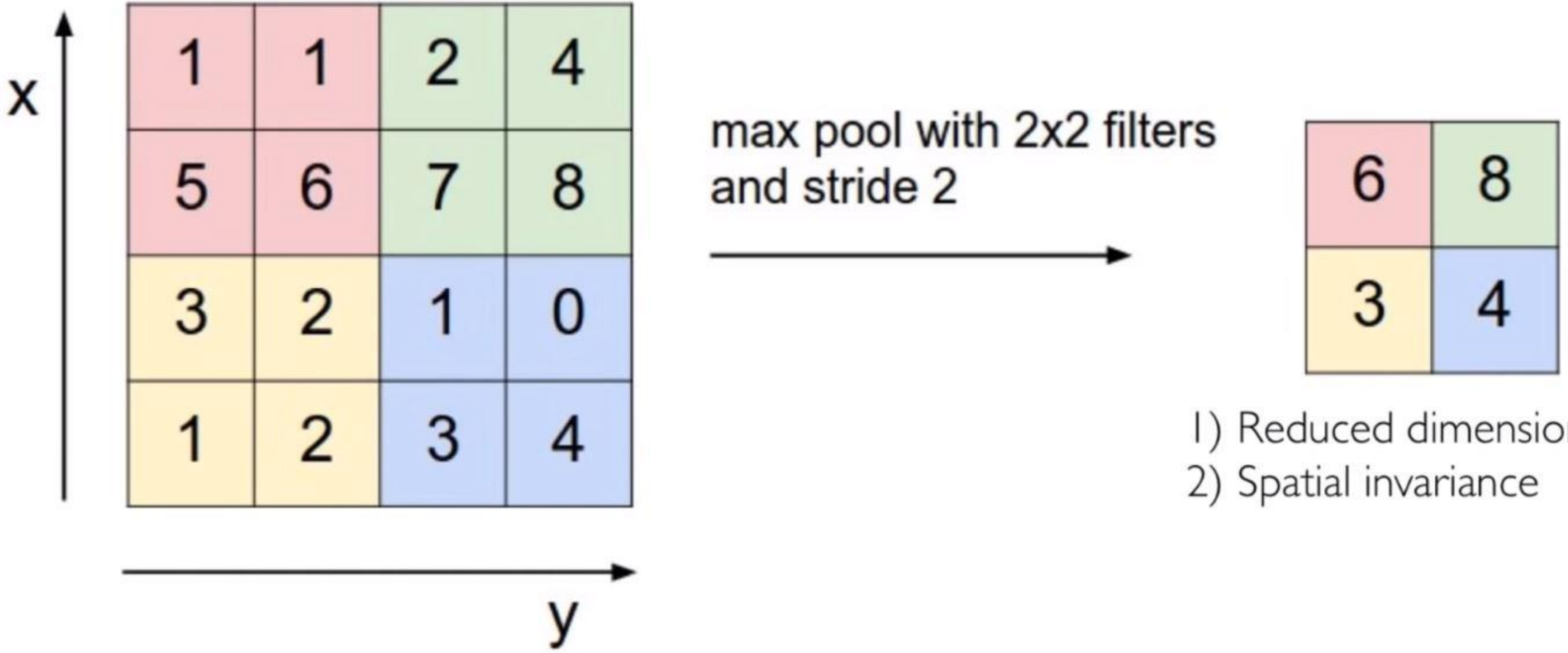
0.55	0.33	0.55	0.33
0.33	1.00	0.55	0.11
0.55	0.55	0.55	0.11
0.33	0.11	0.11	0.33

0.33	0	0.55	0.33	0.11	0	0.77
0	0.11	0	0.33	0	1.00	0
0.55	0	0.11	0	1.00	0	0.11
0.33	0.33	0	0.55	0	0.33	0.33
0.11	0	1.00	0	0.11	0	0.55
0	1.00	0	0.33	0	0.11	0
0.77	0	0.11	0.33	0.55	0	0.33

0.33	0.55	1.00	0.77
0.55	0.55	1.00	0.33
1.00	1.00	0.11	0.55
0.77	0.33	0.55	0.33

- Do note that this is for **one filter.** We need to do it for 2 other filters as well. This is done and we arrive at the above result:

Pooling

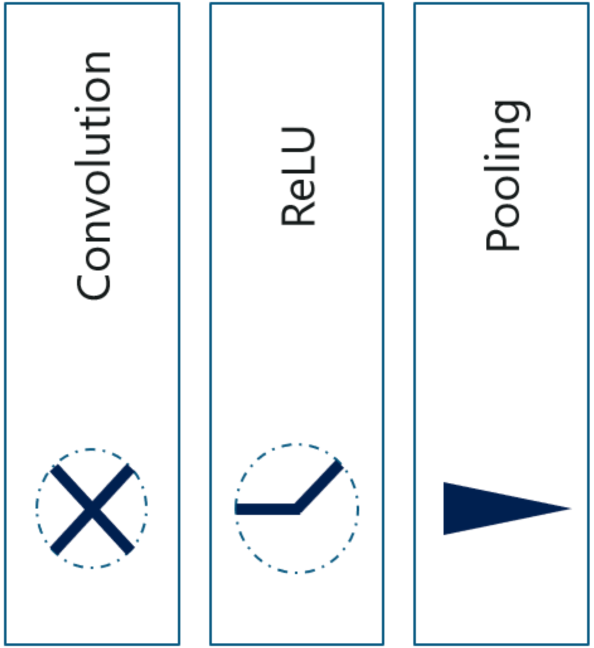


1) Reduced dimensionality
2) Spatial invariance

How else can we downsample and preserve spatial invariance?

Stacking Up The Layers

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



1.00	0.33	0.55	0.33
0.33	1.00	0.33	0.55
0.55	0.33	1.00	0.11
0.33	0.55	0.11	0.77

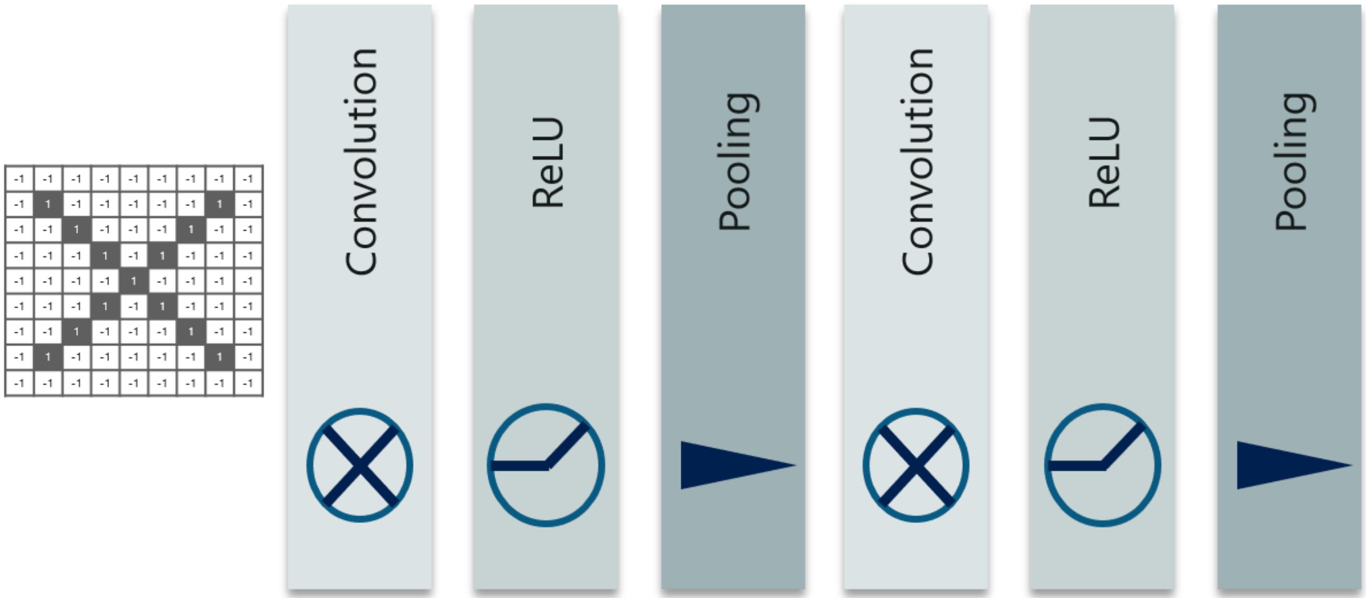
0.55	0.33	0.55	0.33
0.33	1.00	0.55	0.11
0.55	0.55	0.55	0.11
0.33	0.11	0.11	0.33

0.33	0.55	1.00	0.77
0.55	0.55	1.00	0.33
1.00	1.00	0.11	0.55
0.77	0.33	0.55	0.33

- So to get the **time-frame** in one picture we're here with a **4×4** matrix from a **7×7** matrix after passing the input through 3 layers

– **Convolution, ReLU** and **Pooling** as shown above:

Stacking Up The Layers



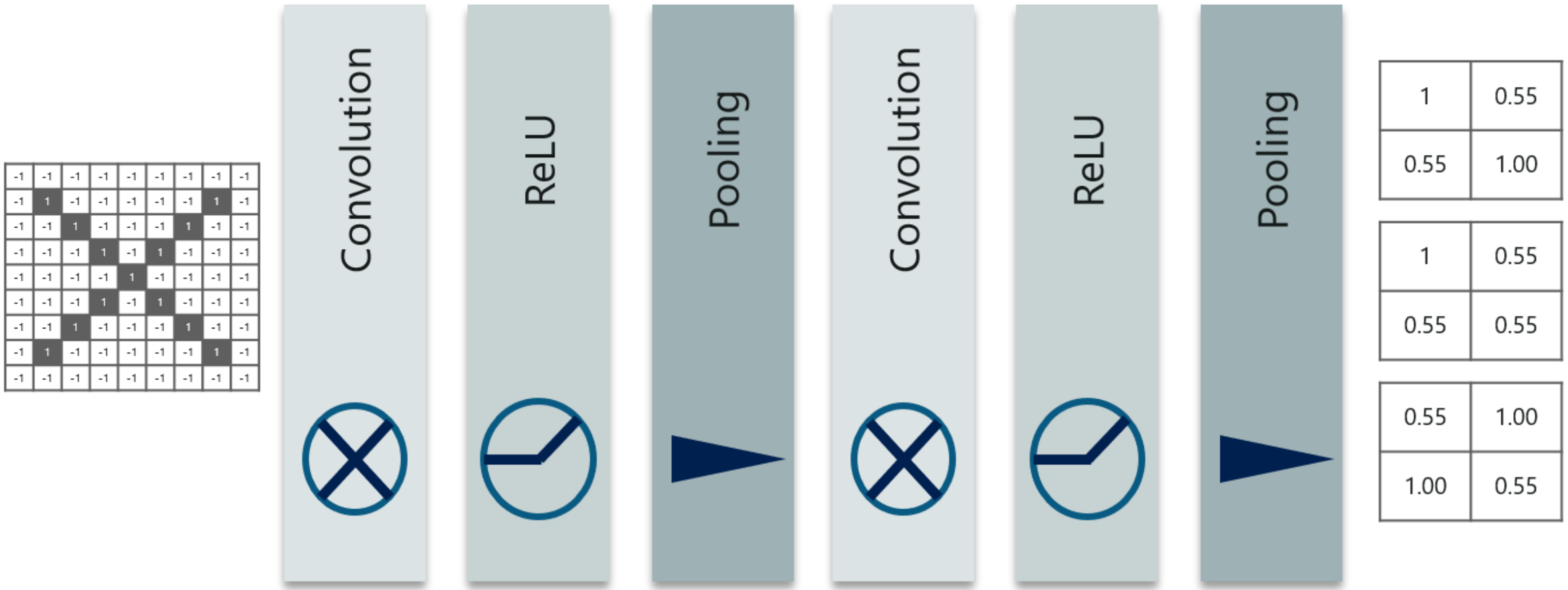
1	0.55
0.55	1.00

1	0.55
0.55	0.55

0.55	1.00
1.00	0.55

- But can we **further reduce** the image from **4×4** to **something lesser?**
- **Yes, we can!** We need to perform the 3 operations in an iteration after the first pass. So after the second pass we arrive at a 2×2 matrix as shown below:

Stacking Up The Layers



- But can we **further reduce** the image from **4×4** to **something lesser?**
- **Yes, we can!** We need to perform the 3 operations in an iteration after the first pass. So after the second pass we arrive at a 2×2 matrix as shown below:

Dense Layer- Last Layer

- This **mimics high level reasoning** where all possible **pathways** from the **input** to **output** are considered.
- Also, fully connected layer is the final layer where the classification actually happens. Here we take our filtered and shrinked images and put them into one single list as shown below:

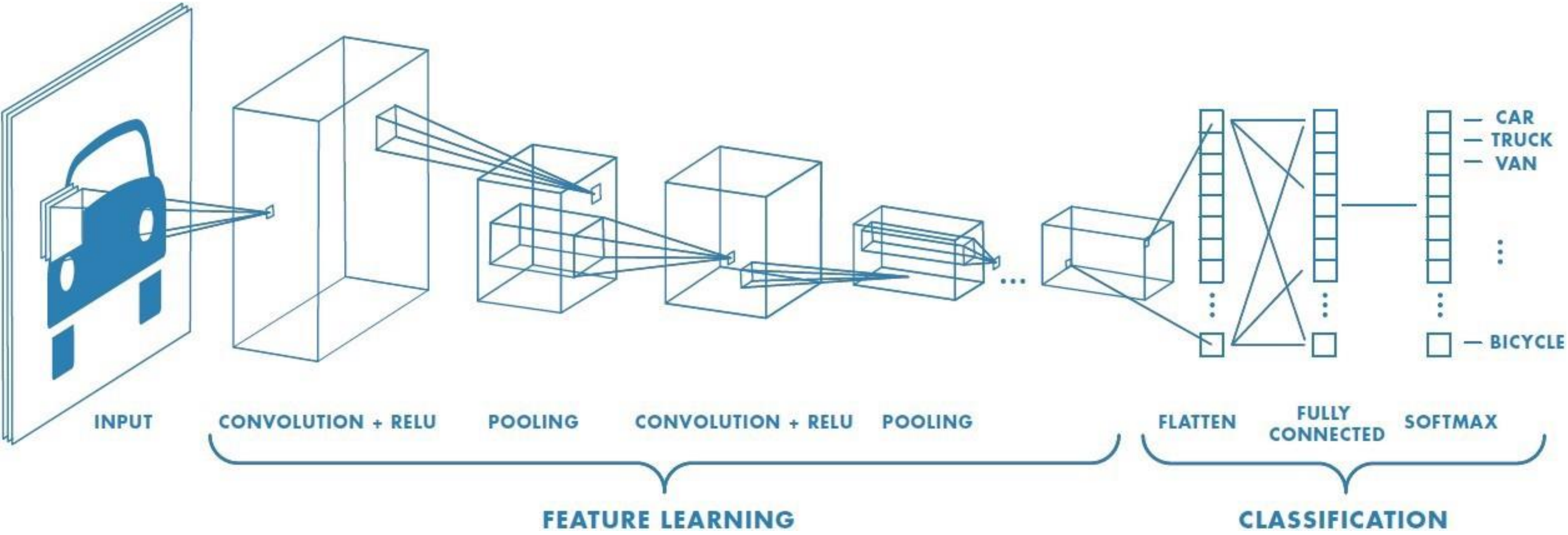
1	0.55
0.55	1.00

1	0.55
0.55	0.55

0.55	1.00
1.00	0.55

1.00
0.55
0.55
1.00
1.00
0.55
0.55
0.55
0.55
1.00
1.00
0.55

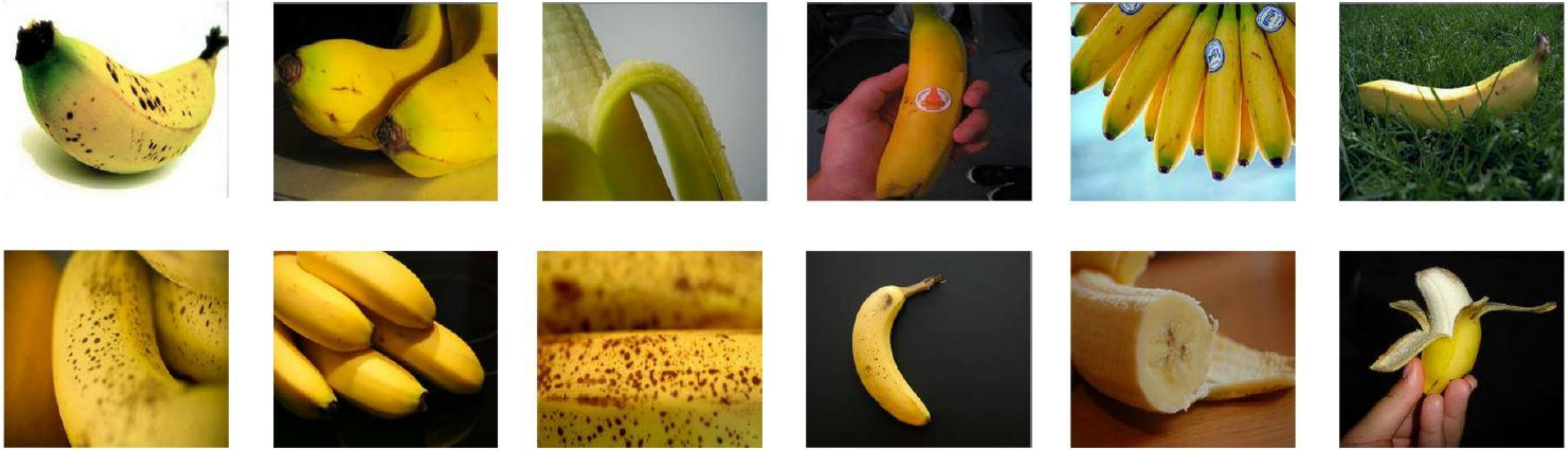
Dense Layer- Last Layer



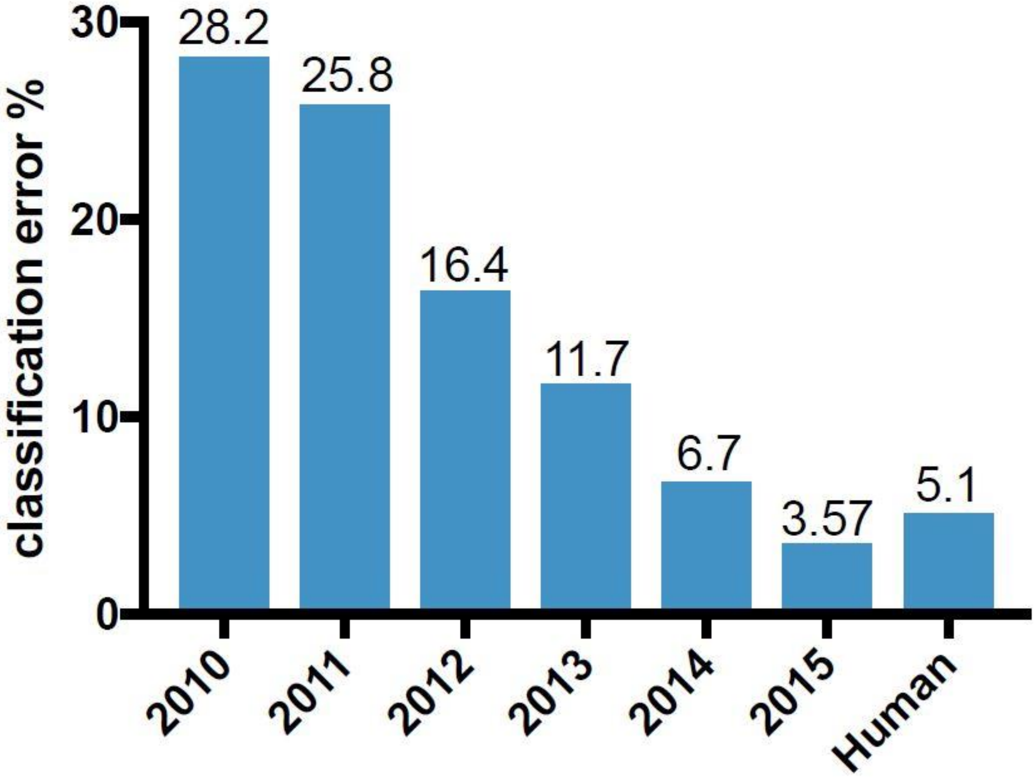
ImageNet Dataset

Dataset of over 14 million images across 21,841 categories

"Elongated crescent-shaped yellow fruit with soft sweet flesh"



ImageNet Challenge: Classification Task



2012: AlexNet. First CNN to win.
- 8 layers, 61 million parameters

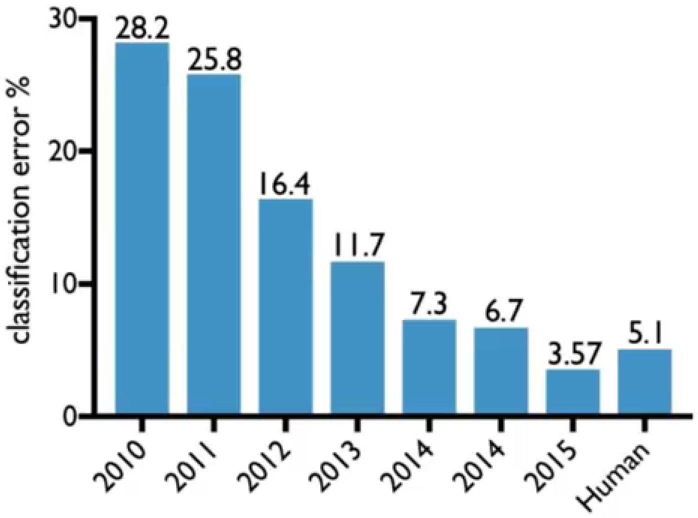
2013: ZFNet
- 8 layers, more filters

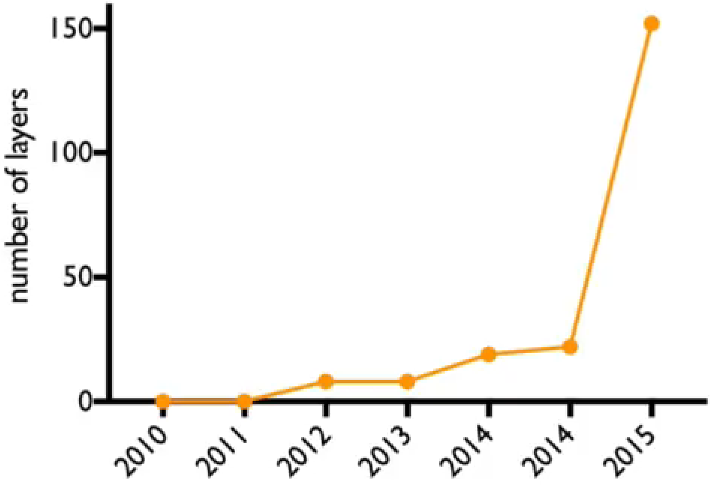
2014: VGG
- 19 layers

2014: GoogLeNet
- "Inception" modules
- 22 layers, 5million parameters

2015: ResNet
- 152 layers

ImageNet Challenge: Classification Task





Deep Learning

IntroToDeepLearning.com

References

- MIT 6.S191 Introduction to Deep Learning (introtodeeplearning.com)

(Some slides are taken from here)