TRINITY INSTITUTE OF TECHNOLOGY & RESEARCH, BHOPAL

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

CS601 MACHINE LEARNING

Topic Covered

CONVOLUTIONAL NEURAL NETWORK

What Are Convolutional Neural Networks?

Convolutional Neural Networks are very similar to ordinary Neural Networks from the previous chapter: they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity.

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.**

Why Not Fully Connected Networks?
We **cannot** make use of fully connected networks when it comes to **Convolutional Neural Networks,** here's why!

Consider the following image:

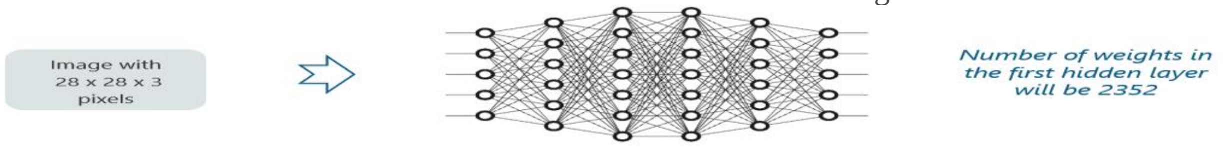


Image with 28 x 28 x 3 pixels
Number of weights in the first hidden layer will be 2352

Here, we have **considered** 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.

But this case **isn't practical**. Now, take a look at this:

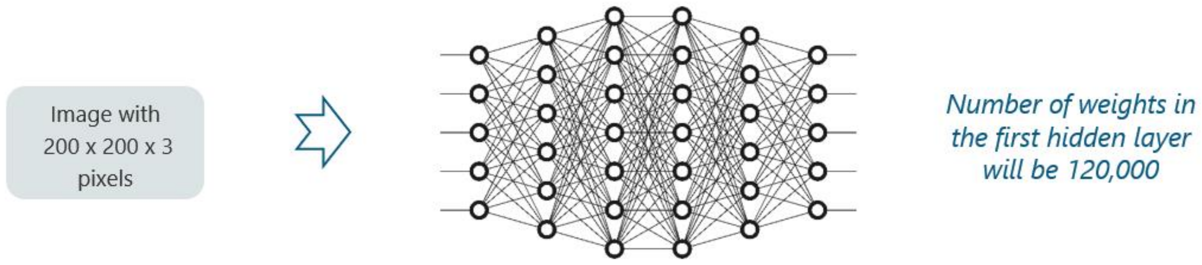
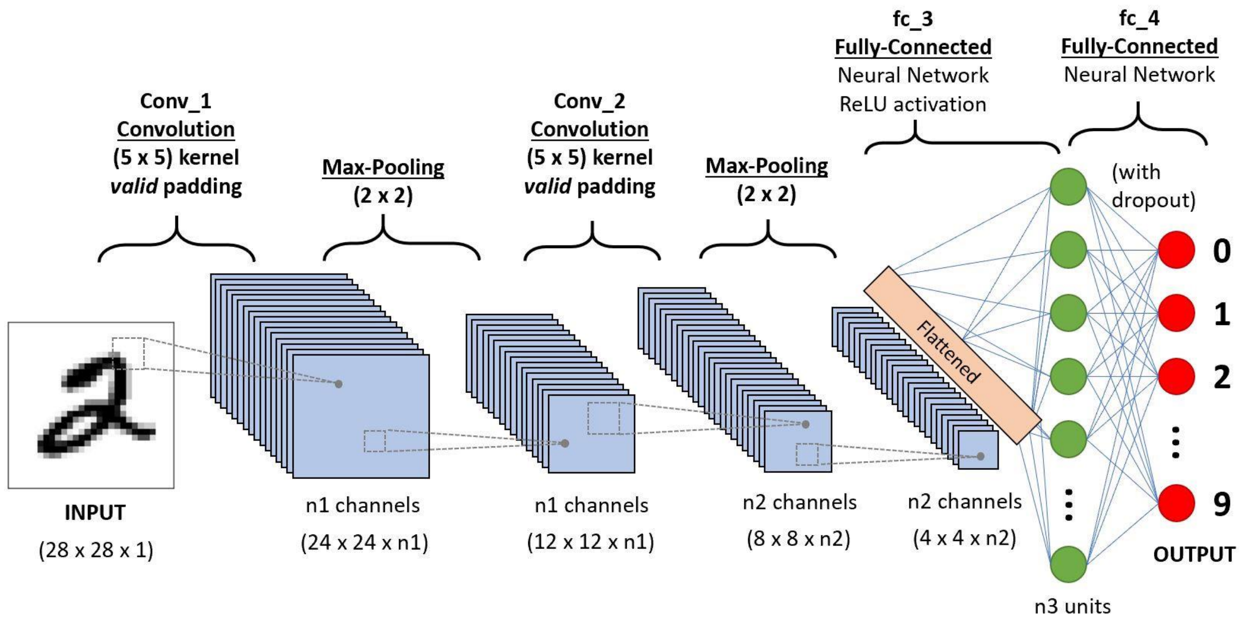


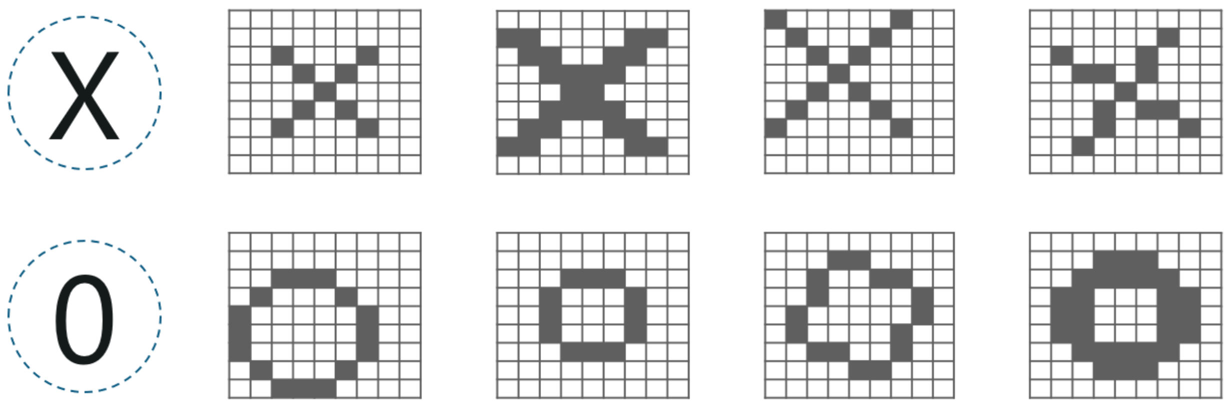
Image with 200 x 200 x 3 pixels
Number of weights in the first hidden layer will be 120,000

Any **generic** input **image** will **atleast** 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.**

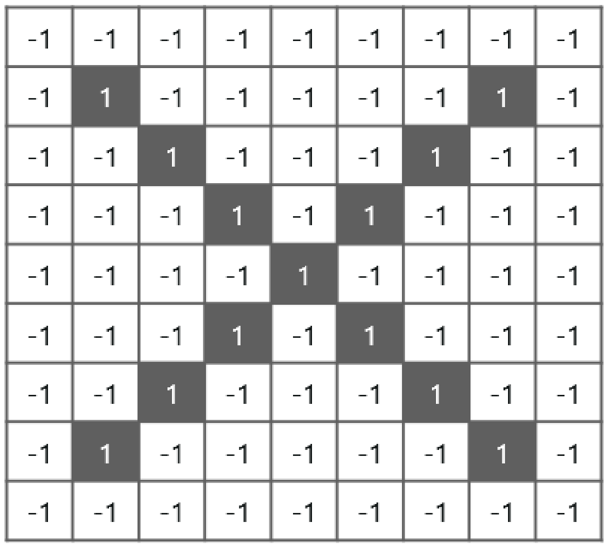
This leads to **over-fitting** and isn't practical. **Hence, we cannot make use of fully connected networks.**



Example of CNN:
Consider the image below:

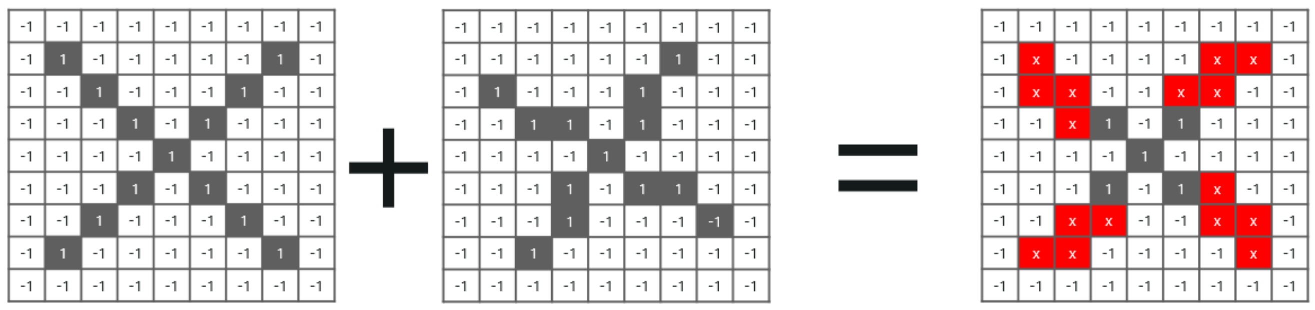


Here, there are multiple renditions of X and O’s. This makes it tricky for the computer to recognize. But the goal is that if the **input signal** looks like **previous** images it has seen before, the **“image” reference** signal will be mixed into, or **convolved** with, the **input** signal. The resulting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



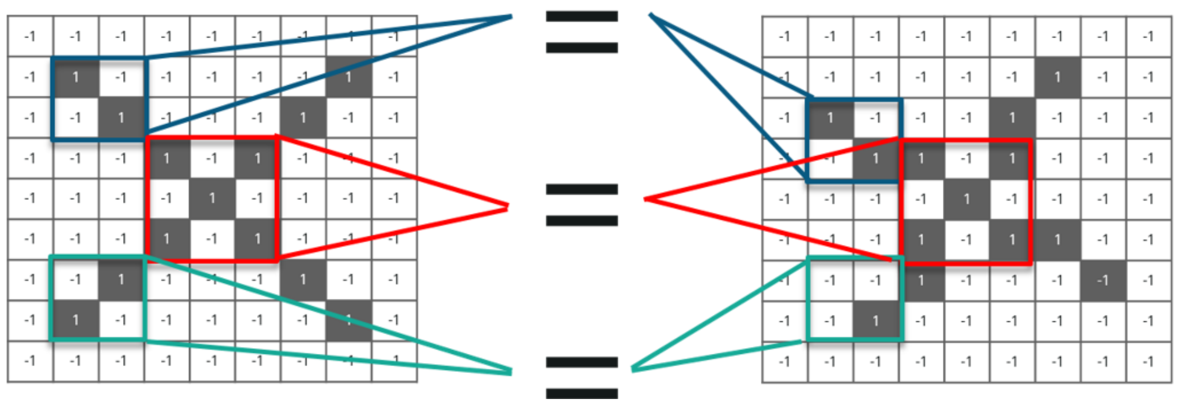
-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	-1	-1	1	-1	-1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1

So, 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.



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?



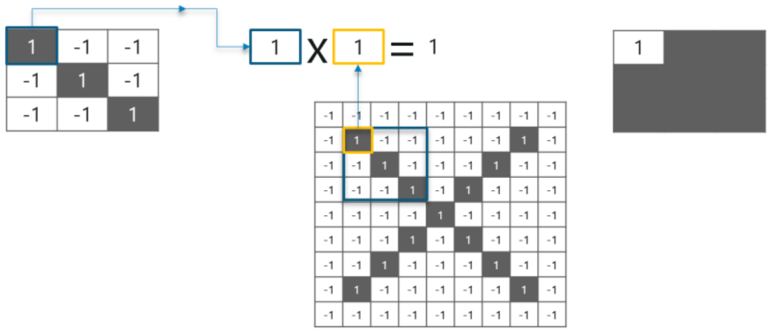
We take **small patches** of the pixels called **filters** and try to **match** them in the corresponding **nearby** locations to see if we get a **match.** By doing this, the Convolutional Neural Network **gets a lot better** at seeing **similarity** than directly trying to match the **entire image.**

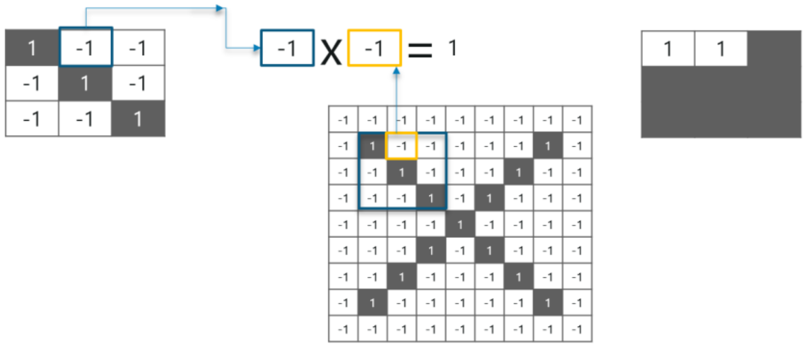
Convolution Of An Image

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).

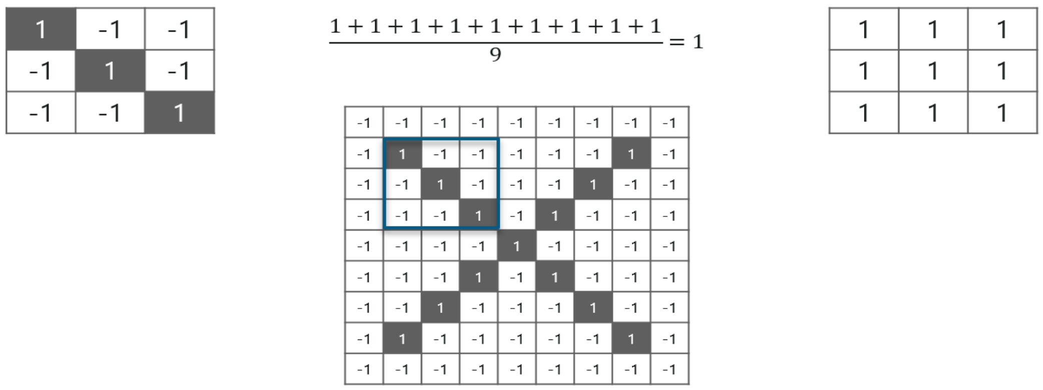
We have **4 steps** for convolution:

- **Line up** the feature and the image
- **Multiply** each **image** pixel by corresponding **feature** pixel
- **Add** the values and find the **sum**
- **Divide** the sum by the **total** number of pixels in the **feature**



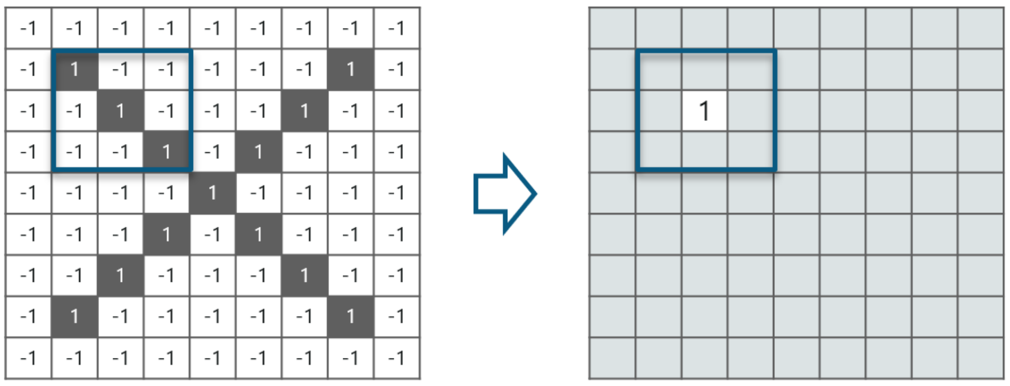


Consider the above image – As you can see, we are **done** with the first **2 steps**. We considered a **feature image** and **one pixel** from it. We **multiplied** this with the **existing image** and the product is stored in another **buffer feature image**.

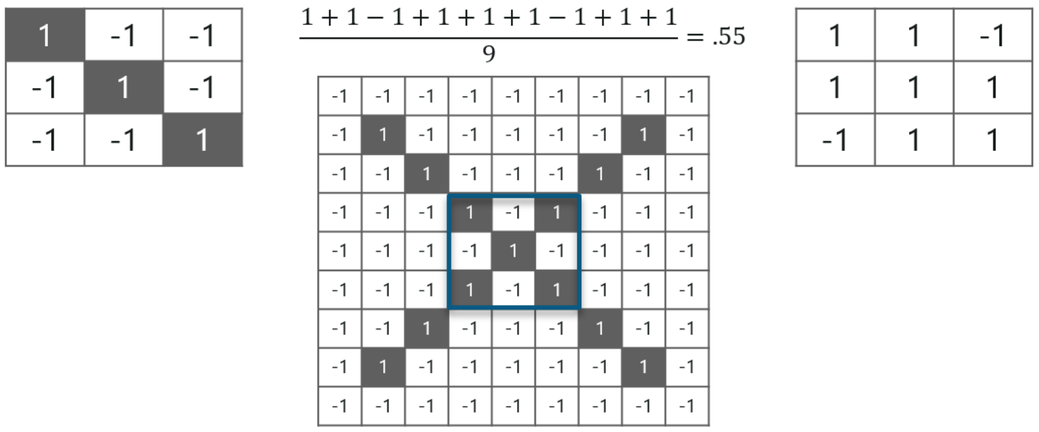


$$\frac{1+1+1+1+1+1+1+1+1}{9} = 1$$

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:

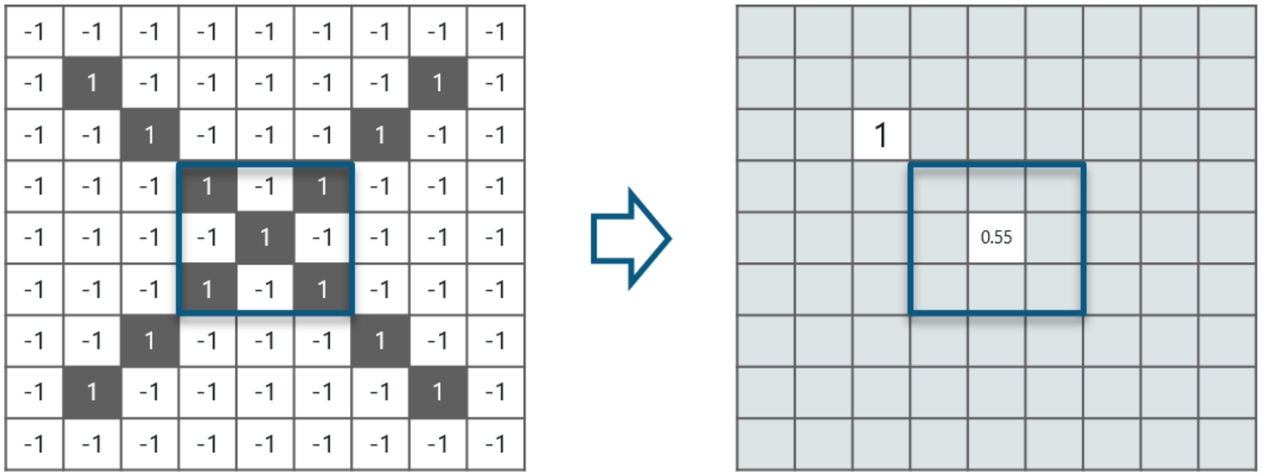


Now, we can **move** this **filter** around and do the **same** at **any pixel** in the image. For **better clarity,** let's consider **another example:**

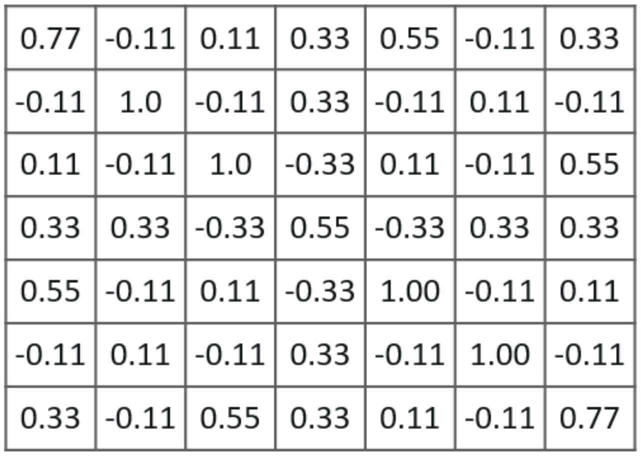


$$\frac{1+1-1+1+1+1-1+1+1}{9} = .55$$

As you can see, here after performing the first 4 steps we have the value at 0.55! We take this value and place it in the image as explained before. This is done in the following 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:



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

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.**

Strides

Stride is the number of pixels shifts over the input matrix. When the stride is 1 then we move the filters to 1 pixel at a time. When the stride is 2 then we move the filters to 2 pixels at a time and so on. The below figure shows convolution would work with a stride of 2.

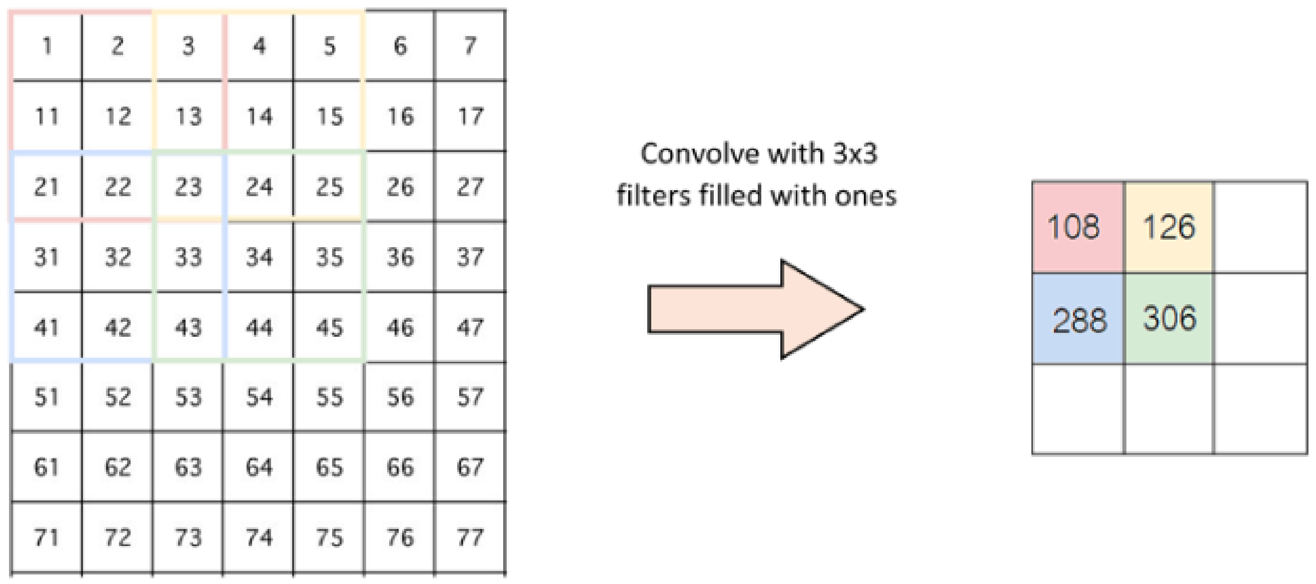


Figure 6 : Stride of 2 pixels

Padding

Sometimes filter does not fit perfectly fit the input image. We have two options:

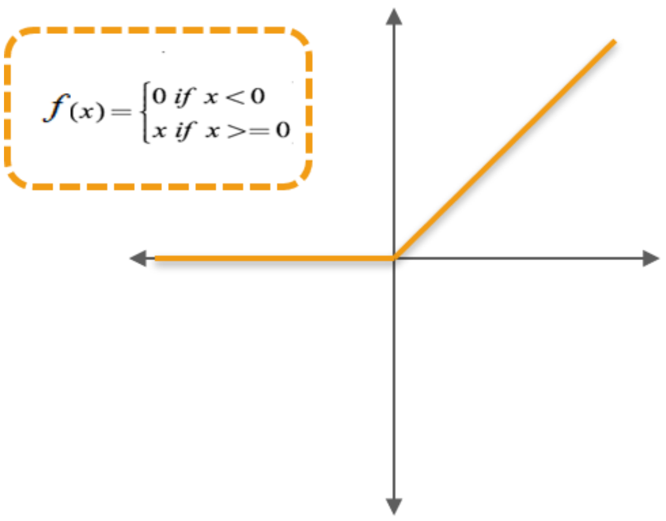
- Pad the picture with zeros (zero-padding) so that it fits
- Drop the part of the image where the filter did not fit. This is called valid padding which keeps only valid part of the image.

ReLU Layer

ReLU is an activation function. But, what is an activation function?

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.

Consider the below example:

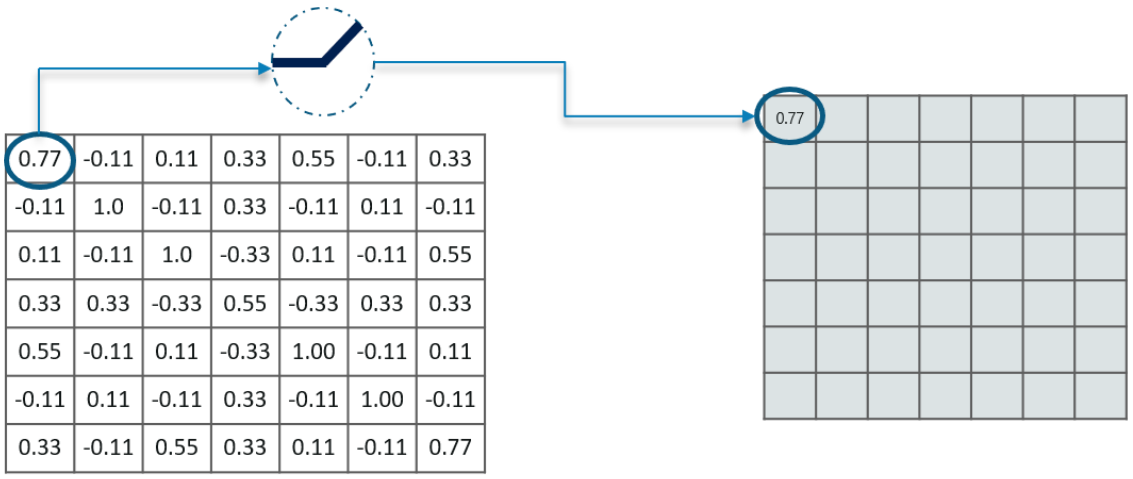


$$f(x) = \begin{cases} 0 \ if \ x < 0 \\ x \ if \ x >= 0 \end{cases}$$

We have considered a simple function with the values as mentioned above. So the function only performs an operation if that value is obtained by the dependent variable. For this example, the following values are obtained:

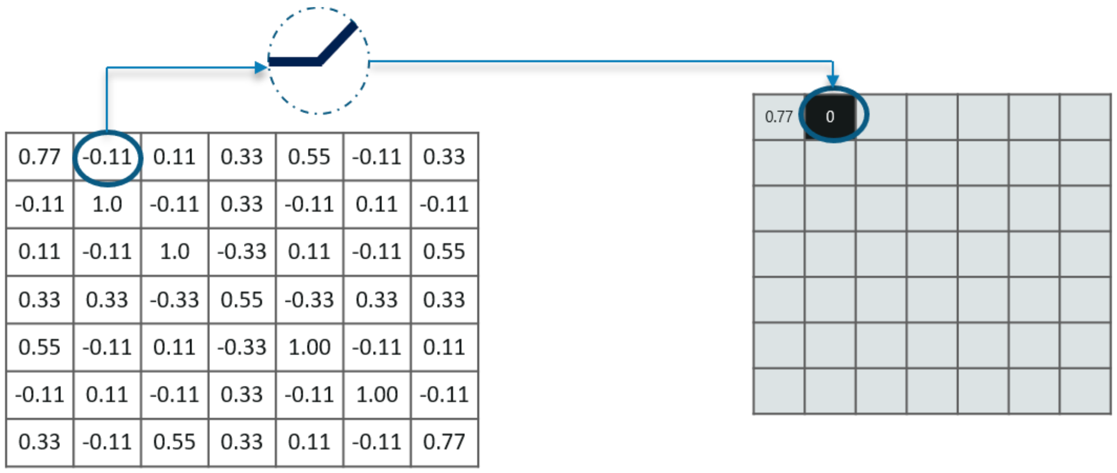
x	*f(x)=x*	F(x)
-3	*f(-3) = 0*	0
-5	*f(-5) = 0*	0
3	*f(3) = 3*	3
5	*f(5) = 5*	5

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

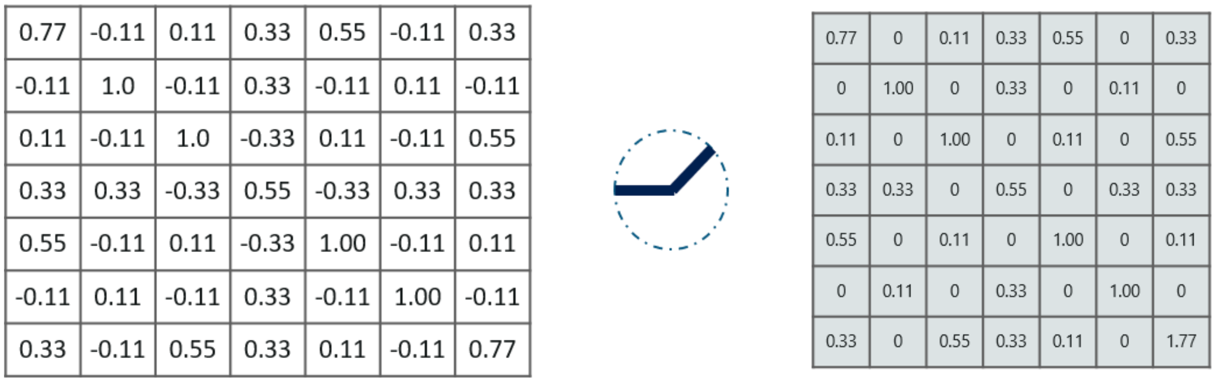
The main aim is to remove all the negative values from the convolution. All the positive values remain the same but all the negative values get changed to zero as shown below:



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					

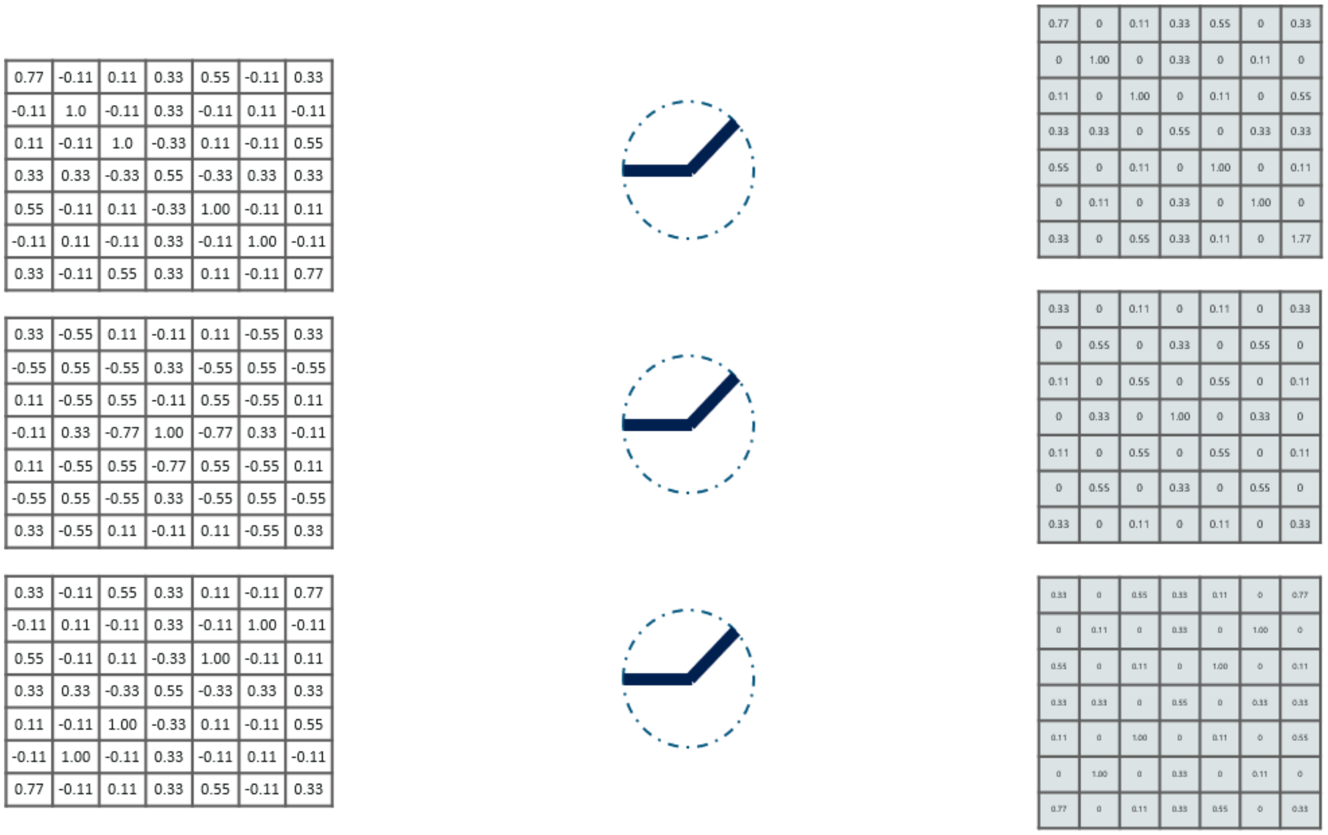
So after we process this particular feature we get the following output:



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

Now, similarly we do the same process to all the other feature images as well:



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

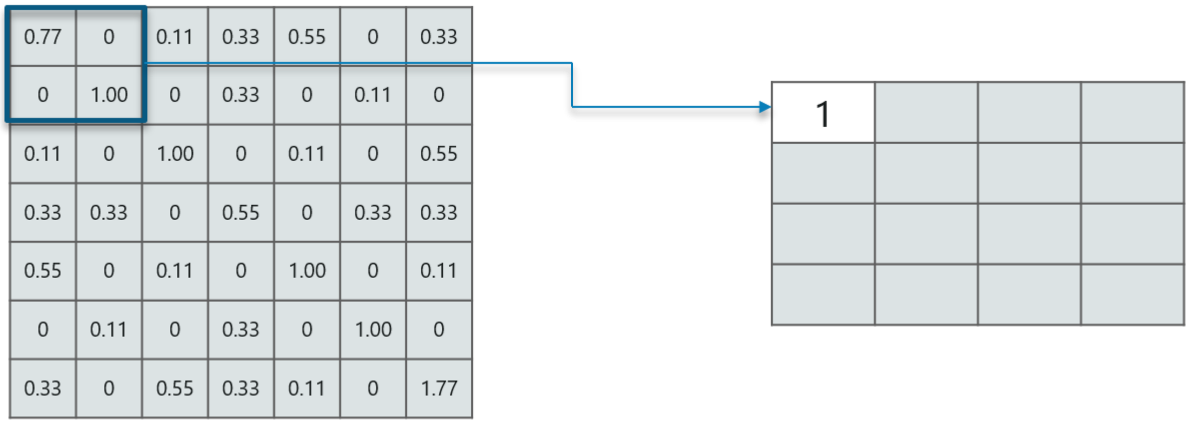
Inputs from the convolution layer can be **"smoothened"** to **reduce** the **sensitivity** of the **filters** to **noise** and **variations.** This smoothing process is called **subsampling** and can be **achieved** by taking **averages** or taking the **maximum** over a **sample** of the signal.

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:

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

Let us understand this with an example. Consider performing pooling with a window size of 2 and stride being 2 as well.

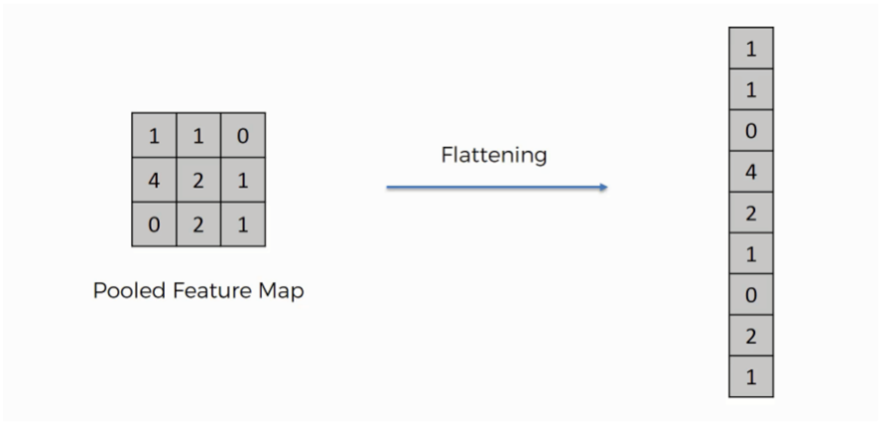


Artificial Intelligence Training

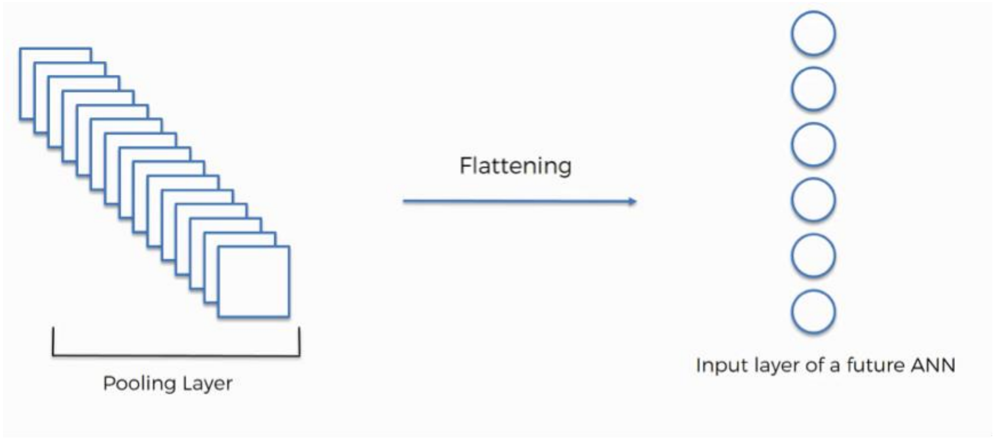
A CNN sequence to classify handwritten digits

Flattening

After finishing the previous two steps, we're supposed to have a pooled feature map by now. As the name of this step implies, we are literally going to flatten our pooled feature map into a column like in the image below.



The reason we do this is that we're going to need to insert this data into an artificial neural network later on.



References/Suggested readings

MACHINE LEARNING: GEEKS FOR GEEKS ,TUTORIAL POINT.