Home (https://www.analyticsvidhya.com/) › Deep Learning (https://www.analyticsvidhya.com/blog/category/deep-learning/) ›

Architecture of Convolutional Neural Networks (CNNs) demystified

DEEP LEARNING (HTTPS://WWW.ANALYTICSVIDHYA.COM/BLOG/CATEGORY/DEEP-LEARNING/)

Introduction

I will start with a confession – there was a time when I didn't really understand deep learning. I would look at the research papers and articles on the topic and feel like it is a very complex topic. I tried understanding Neural networks and their various types, but it still looked difficult.

Then one day, I decided to take one step at a time. I decided to start with basics and build on them. I decided that I will break down the steps applied in these techniques and do the steps (and calculations) manually, until I understand how they work. It was time taking and intense effort – but the results were phenomenal.

Now, I can not only understand the spectrum of deep learning, I can visualize things and come up with better ways because my fundamentals are clear. It is one thing to apply neural networks mindlessly and it is other to understand what is going on and how are things happening at the back.

Today, I am going to share this secret recipe with you. I will show you how I took the Convolutional Neural Networks and worked on them till I understood them. I will walk you through the journey so that you develop a deep understanding of how CNNs work.

In this article I am going to discuss the architecture behind Convolutional Neural Networks, which are designed to address image recognition and classification problems.

I am assuming that you have a basic understanding of how a neural network works. If you're not sure of your understanding I would request you to go through this article (https://www.analyticsvidhya.com/blog/2017/05/neural-network-from-scratch-in-python-and-r/) before you read on.

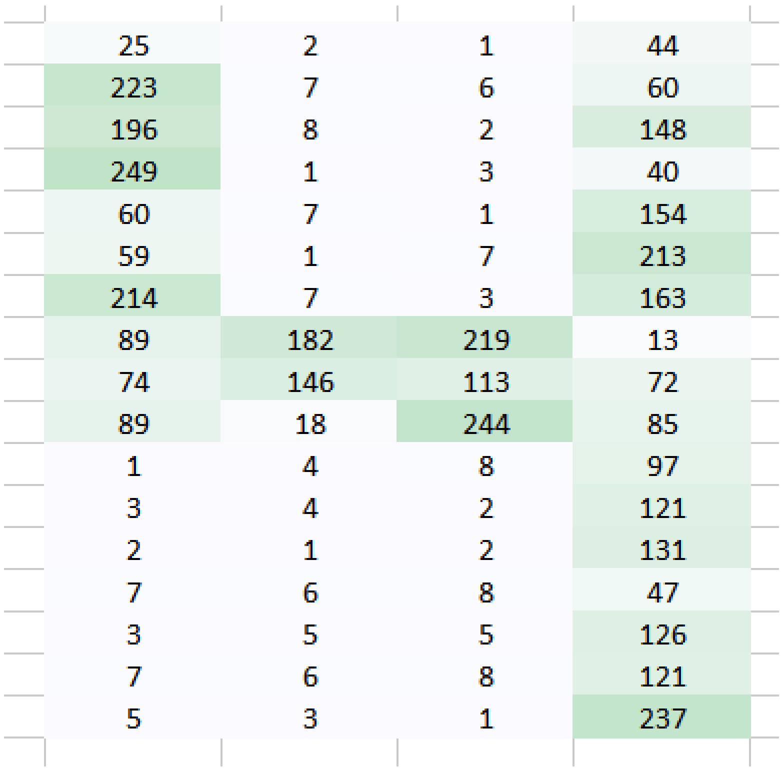
Table of Contents:

1. How does a machine look at an image?

Human brain is a very powerful machine. We see (capture) multiple images every second and process them without realizing how the processing is done. But, that is not the case with machines. The first step in image processing is to understand, how to represent an image so that the machine can read it?

In simple terms, every image is an arrangement of dots (a pixel) arranged in a special order. If you change the order or color of a pixel, the image would change as well. Let us take an example. Let us say, you wanted to store and read an image with a number 4 written on it.

The machine will basically break this image into a matrix of pixels and store the color code for each pixel at the representative location. In the representation below – number 1 is white and 256 is the darkest shade of green color (I have constrained the example to have only one color for simplicity).

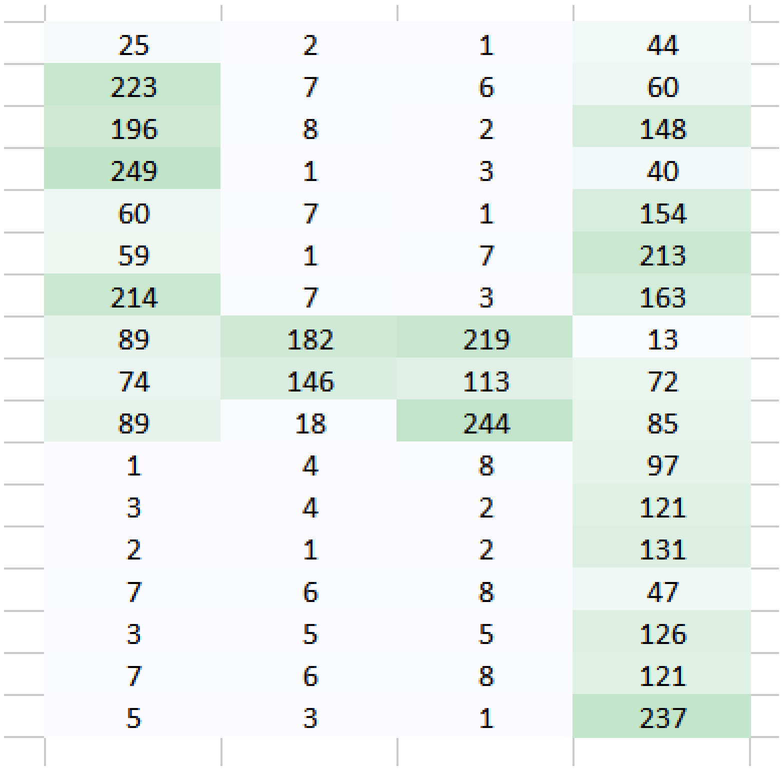


25	2	1	44
223	7	6	60
196	8	2	148
249	1	3	40
60	7	1	154
59	1	7	213
214	7	3	163
89	182	219	13
74	146	113	72
89	18	244	85
1	4	8	97
3	4	2	121
2	1	2	131
7	6	8	47
3	5	5	126
7	6	8	121
5	3	1	237

Once you have stored the images in this format, the next challenge is to have our neural network understand the arrangement and the pattern.

2. How do we help a neural network to identify images ?

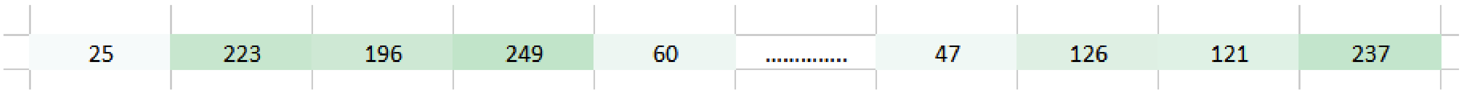
A number is formed by having pixels arranged in a certain fashion.



25	2	1	44
223	7	6	60
196	8	2	148
249	1	3	40
60	7	1	154
59	1	7	213
214	7	3	163
89	182	219	13
74	146	113	72
89	18	244	85
1	4	8	97
3	4	2	121
2	1	2	131
7	6	8	47
3	5	5	126
7	6	8	121
5	3	1	237

Let's say we try to use a fully connected network to identify it? What does it do ?

A fully connected network would take this image as an array by flattening it and considering pixel values as features to predict the number in image. Definitely it's tough for the network to understand what's happening underneath.



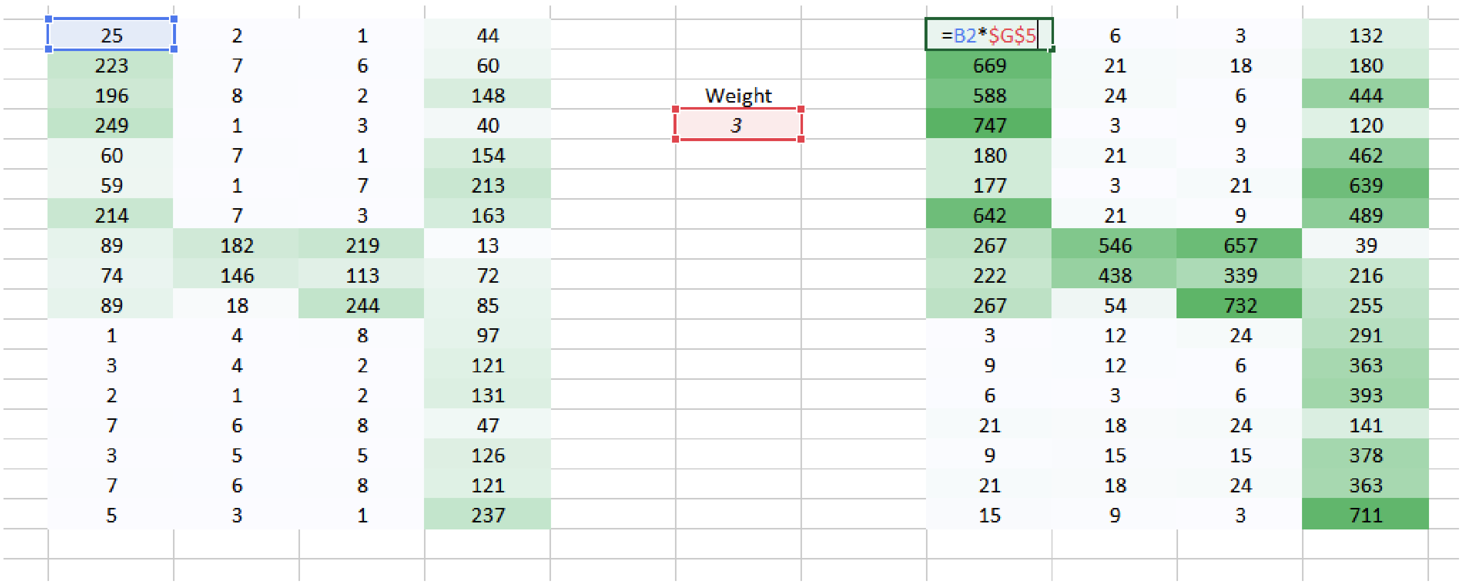
25	223	196	249	60	....	47	126	121	237

It's impossible even for a human to identify that this is a representation of number 4. We have lost the spatial arrangement of pixels completely.

What can we possibly do? Let's try to to extract features from the original image such that the spatial arrangement is preserved.

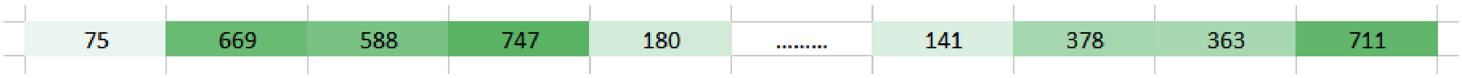
Case 1:

Here we have used a weight to multiply the initial pixel values.



25	2	1	44			=B2*$G$5	6	3	132
223	7	6	60			669	21	18	180
196	8	2	148		Weight	588	24	6	444
249	1	3	40		3	747	3	9	120
60	7	1	154			180	21	3	462
59	1	7	213			177	3	21	639
214	7	3	163			642	21	9	489
89	182	219	13			267	546	657	39
74	146	113	72			222	438	339	216
89	18	244	85			267	54	732	255
1	4	8	97			3	12	24	291
3	4	2	121			9	12	6	363
2	1	2	131			6	3	6	393
7	6	8	47			21	18	24	141
3	5	5	126			9	15	15	378
7	6	8	121			21	18	24	363
5	3	1	237			15	9	3	711

It does get easier for the naked eye to identify that this is a 4. But again to send this image to a fully connected network, we would have to flatten it. We are unable to preserve the spatial arrangement of the image.

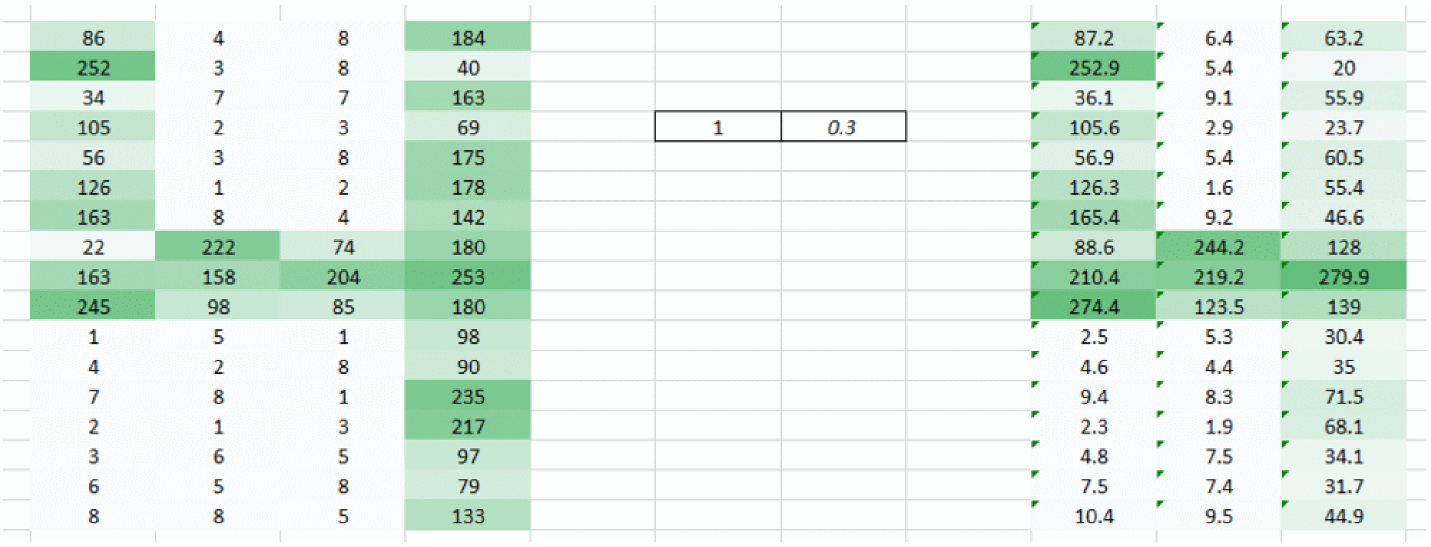


75	669	588	747	180	....	141	378	363	711

Case 2:

Now we can see that flattening the image destroys its arrangement completely. we need to devise a way to send images to a network without flattening them and retaining its spatial arrangement. We need to send 2D/3D arrangement of pixel values.

Let's try taking two pixel values of the image at a time rather than taking just one. This would give the network a very good insight as to how does the adjacent pixel look like. Now that we're taking two pixels at a time, we shall take two weight values too.



86	4	8	184
252	3	8	40
34	7	7	163
105	2	3	69
56	3	8	175
126	1	2	178
163	8	4	142
22	222	74	180
163	158	204	253
245	98	85	180
1	5	1	98
4	2	8	90
7	8	1	235
2	1	3	217
3	6	5	97
6	5	8	79
8	8	5	133

1	0.3

87.2	6.4	63.2
252.9	5.4	20
36.1	9.1	55.9
105.6	2.9	23.7
56.9	5.4	60.5
126.3	1.6	55.4
165.4	9.2	46.6
88.6	244.2	128
210.4	219.2	279.9
274.4	123.5	139
2.5	5.3	30.4
4.6	4.4	35
9.4	8.3	71.5
2.3	1.9	68.1
4.8	7.5	34.1
7.5	7.4	31.7
10.4	9.5	44.9

I hope you noted that the image now became a 3 column arrangement from a 4 column arrangement initially. The image got smaller since we're now moving two pixels at a time (pixels are getting shared in each movement). We made the image smaller and we can still understand that it's a 4 to quite a great extent. Also, an important fact to realise is that we we're taking two consecutive horizontal pixels, therefore only horizontal arrangement is considered here.

This is one way to extract features from an image. We're able to see the left and middle part well, however the right side is not so clear. This is because of the following two problems-

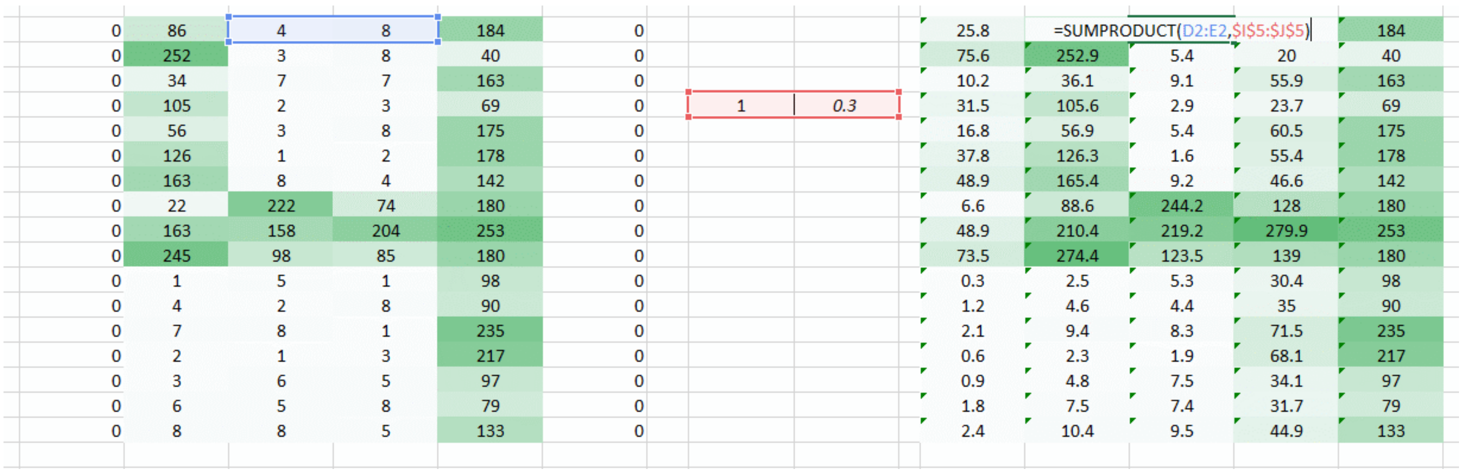
1. The left and right corners of the image is multiplied by the weights just once.
2. The left part is still retained since the weight value is high while the right part is getting slightly lost due to low weight value.

Now we have two problems, we shall have two solutions to solve them as well.

Case 3:

The problem encountered is that the left and right corners of the image is getting passed by the weight just once. What we need to do is we need the network to consider the corners also like other pixels.

We have a simple solution to solve this. Put zeros along the sides of the weight movement.



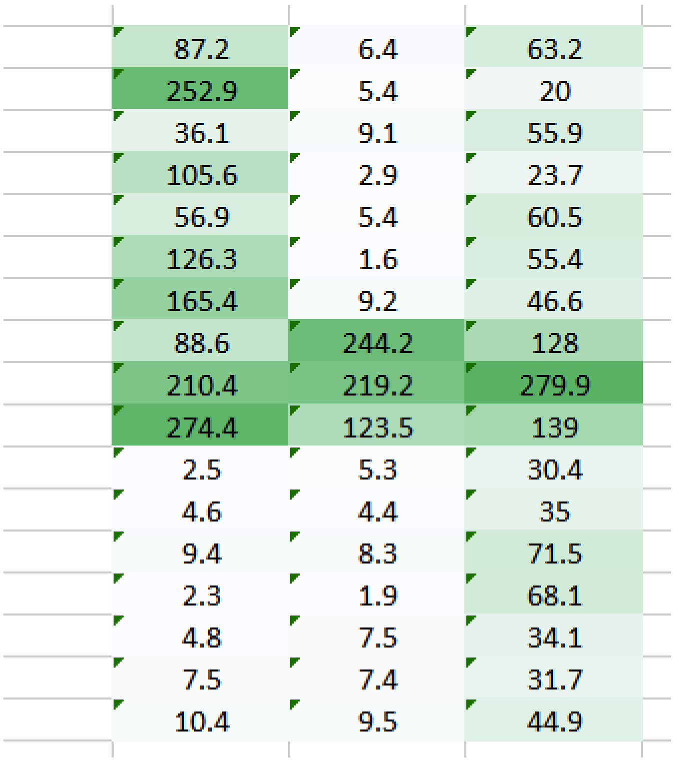
0	86	4	8	184	0			25.8	=SUMPRODUCT(D2:E2,$I$5:$J$5)			184
0	252	3	8	40	0			75.6	252.9	5.4	20	40
0	34	7	7	163	0			10.2	36.1	9.1	55.9	163
0	105	2	3	69	0	1	0.3	31.5	105.6	2.9	23.7	69
0	56	3	8	175	0			16.8	56.9	5.4	60.5	175
0	126	1	2	178	0			37.8	126.3	1.6	55.4	178
0	163	8	4	142	0			48.9	165.4	9.2	46.6	142
0	22	222	74	180	0			6.6	88.6	244.2	128	180
0	163	158	204	253	0			48.9	210.4	219.2	279.9	253
0	245	98	85	180	0			73.5	274.4	123.5	139	180
0	1	5	1	98	0			0.3	2.5	5.3	30.4	98
0	4	2	8	90	0			1.2	4.6	4.4	35	90
0	7	8	1	235	0			2.1	9.4	8.3	71.5	235
0	2	1	3	217	0			0.6	2.3	1.9	68.1	217
0	3	6	5	97	0			0.9	4.8	7.5	34.1	97
0	6	5	8	79	0			1.8	7.5	7.4	31.7	79
0	8	8	5	133	0			2.4	10.4	9.5	44.9	133

You can see that by adding the zeroes the information from the corners is retained. The size of the image is higher too. This can be used in cases where we don't want the image size to reduce.

Case 4:

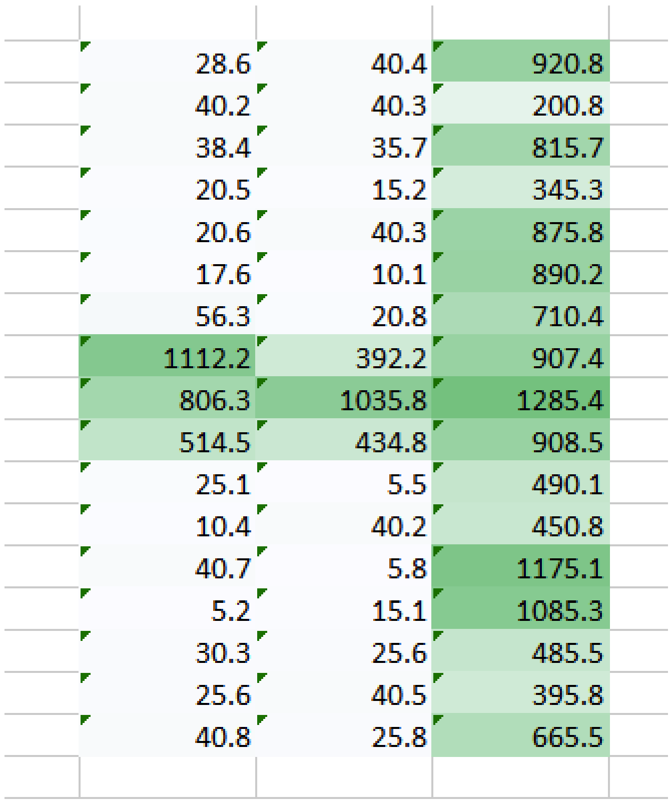
The problem we're trying to address here is that a smaller weight value in the right side corner is reducing the pixel value thereby making it tough for us to recognize. What we can do is, we take multiple weight values in a single turn and put them together.

A weight value of (1,0.3) gave us an output of the form



87.2	6.4	63.2
252.9	5.4	20
36.1	9.1	55.9
105.6	2.9	23.7
56.9	5.4	60.5
126.3	1.6	55.4
165.4	9.2	46.6
88.6	244.2	128
210.4	219.2	279.9
274.4	123.5	139
2.5	5.3	30.4
4.6	4.4	35
9.4	8.3	71.5
2.3	1.9	68.1
4.8	7.5	34.1
7.5	7.4	31.7
10.4	9.5	44.9

while a weight value of the form (0.1,5) would give us an output of the form

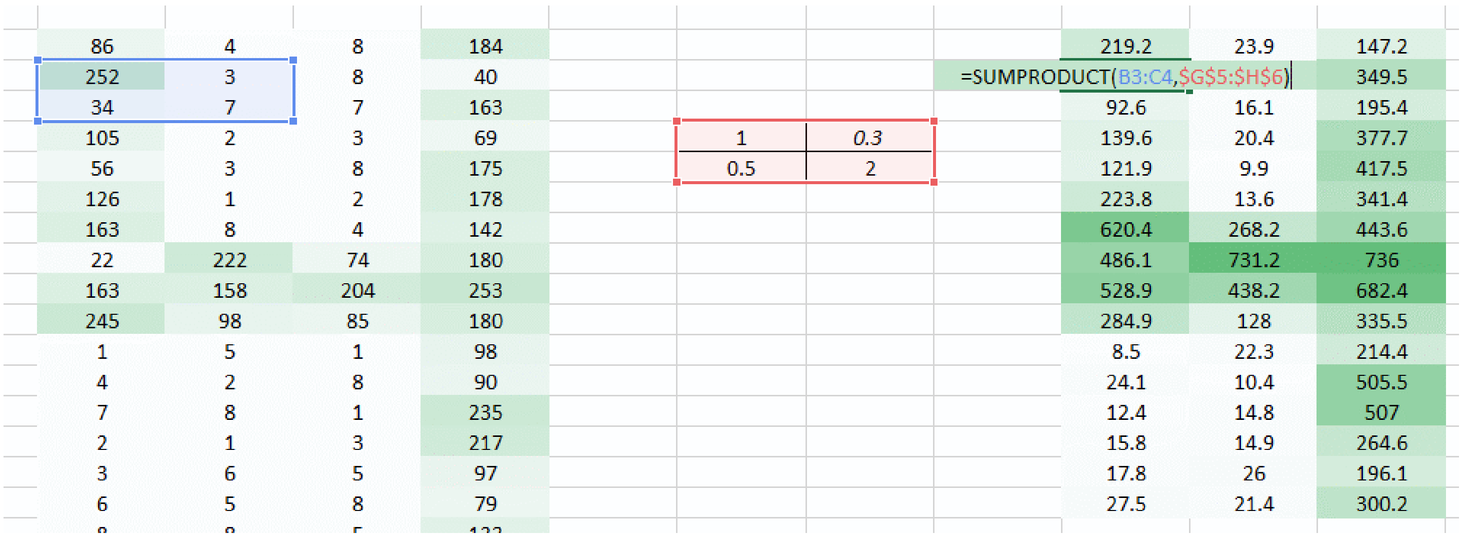


28.6	40.4	920.8
40.2	40.3	200.8
38.4	35.7	815.7
20.5	15.2	345.3
20.6	40.3	875.8
17.6	10.1	890.2
56.3	20.8	710.4
1112.2	392.2	907.4
806.3	1035.8	1285.4
514.5	434.8	908.5
25.1	5.5	490.1
10.4	40.2	450.8
40.7	5.8	1175.1
5.2	15.1	1085.3
30.3	25.6	485.5
25.6	40.5	395.8
40.8	25.8	665.5

A combined version of these two images would give us a very clear picture. Therefore what we did was simply use multiple weights rather than just one to retain more information about the image. The final output would be a combined version of the above two images.

Case 5:

Till now we have used the weights which were trying to take horizontal pixels together. But in most cases we need to preserve the spatial arrangement in both horizontal and vertical direction. We can take the weight as a 2D matrix which takes pixels together in both horizontal and vertical direction. Also, keep in mind that since we have taken both horizontal and vertical movement of weights, the output is one pixel lower in both horizontal and vertical direction.



86	4	8	184					219.2	23.9	147.2
252	3	8	40					=SUMPRODUCT(B3:C4,$G$5:$H$6)		349.5
34	7	7	163					92.6	16.1	195.4
105	2	3	69		1	0.3		139.6	20.4	377.7
56	3	8	175		0.5	2		121.9	9.9	417.5
126	1	2	178					223.8	13.6	341.4
163	8	4	142					620.4	268.2	443.6
22	222	74	180					486.1	731.2	736
163	158	204	253					528.9	438.2	682.4
245	98	85	180					284.9	128	335.5
1	5	1	98					8.5	22.3	214.4
4	2	8	90					24.1	10.4	505.5
7	8	1	235					12.4	14.8	507
2	1	3	217					15.8	14.9	264.6
3	6	5	97					17.8	26	196.1
6	5	8	79					27.5	21.4	300.2

So what did we do ?

What we did above was that we were trying to extract features from an image by using the spatial arrangement of the images. To understand an image its extremely important for a network to understand how the pixels are arranged. What we did above is what exactly a **convolutional neural network** does. We can take the input image, define a weight matrix and the input is convolved to extract specific features from the image without losing the information about its spatial arrangement.

Another great benefit this approach has is that it reduces the number of parameters from the image. As you saw above the convolved images had lesser pixels as compared to the original image. This dramatically reduces the number of parameters we need to train for the network.

3. Defining a Convolutional Neural Network

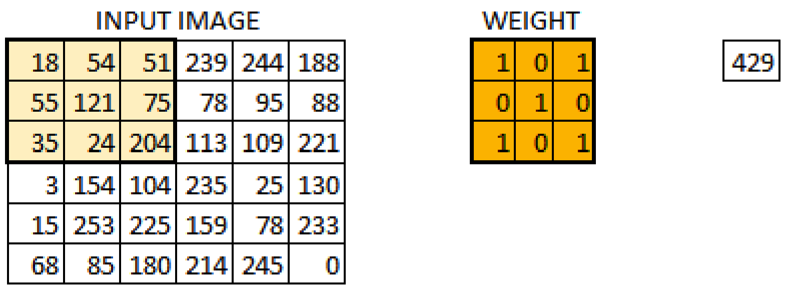
We need three basic components to define a basic convolutional network.

1. The convolutional layer
2. The Pooling layer[optional]
3. The output layer

Let's see each of these in a little more detail

2.1 The Convolution Layer

In this layer, what happens is exactly what we saw in case 5 above. Suppose we have an image of size 6*6. We define a weight matrix which extracts certain features from the images



INPUT IMAGE

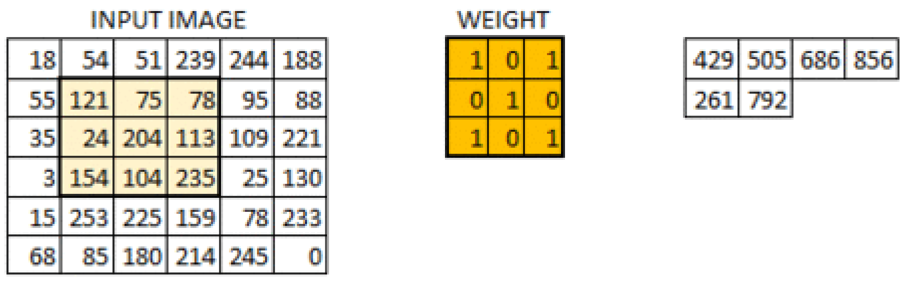
18	54	51	239	244	188
55	121	75	78	95	88
35	24	204	113	109	221
3	154	104	235	25	130
15	253	225	159	78	233
68	85	180	214	245	0

WEIGHT

1	0	1
0	1	0
1	0	1

429

We have initialized the weight as a 3*3 matrix. This weight shall now run across the image such that all the pixels are covered at least once, to give a convolved output. The value 429 above, is obtained by the adding the values obtained by element wise multiplication of the weight matrix and the highlighted 3*3 part of the input image.



INPUT IMAGE

18	54	51	239	244	188
55	121	75	78	95	88
35	24	204	113	109	221
3	154	104	235	25	130
15	253	225	159	78	233
68	85	180	214	245	0

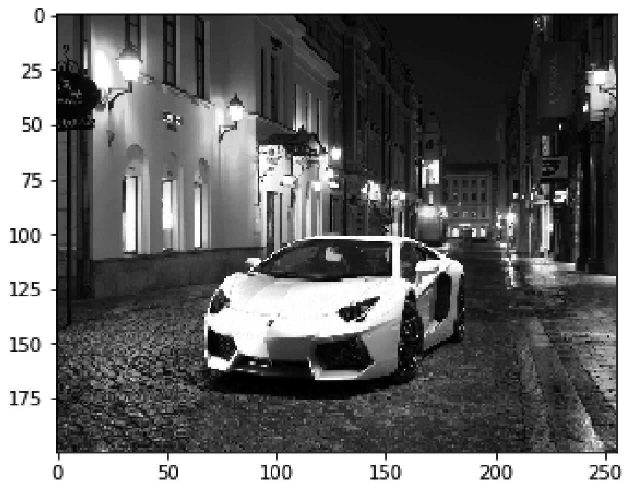
WEIGHT

1	0	1
0	1	0
1	0	1

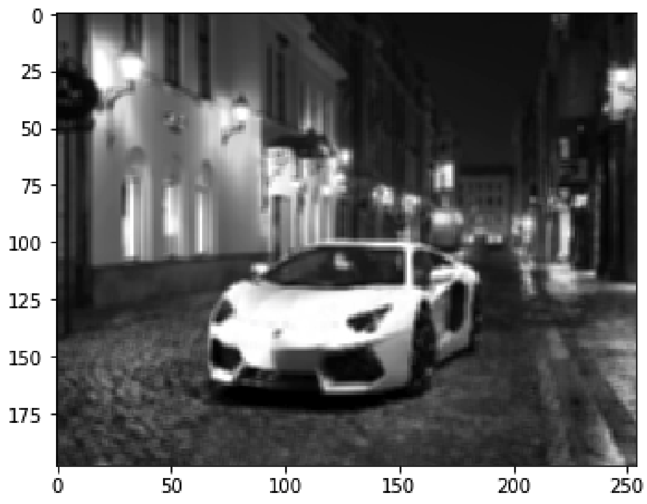
429	505	686	856
261	792		

The 6*6 image is now converted into a 4*4 image. Think of weight matrix like a paint brush painting a wall. The brush first paints the wall horizontally and then comes down and paints the next row horizontally. Pixel values are used again when the weight matrix moves along the image. This basically enables parameter sharing in a convolutional neural network.

Let's see how this looks like in a real image.



Original Image



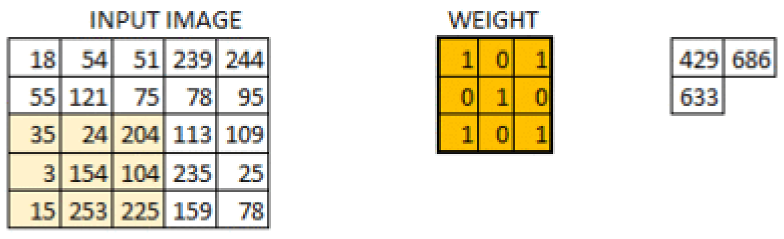
Convoluted image

The weight matrix behaves like a filter in an image extracting particular information from the original image matrix. A weight combination might be extracting edges, while another one might a particular color, while another one might just blur the unwanted noise.

The weights are learnt such that the loss function is minimized similar to an MLP. Therefore weights are learnt to extract features from the original image which help the network in correct prediction. When we have multiple convolutional layers, the initial layer extract more generic features, while as the network gets deeper, the features extracted by the weight matrices are more and more complex and more suited to the problem at hand.

The concept of stride and padding

As we saw above, the filter or the weight matrix, was moving across the entire image moving **one** pixel at a time. We can define it like a hyperparameter, as to how we would want the weight matrix to move across the image. If the weight matrix moves 1 pixel at a time, we call it as a stride of 1. Let's see how a stride of 2 would look like.



INPUT IMAGE

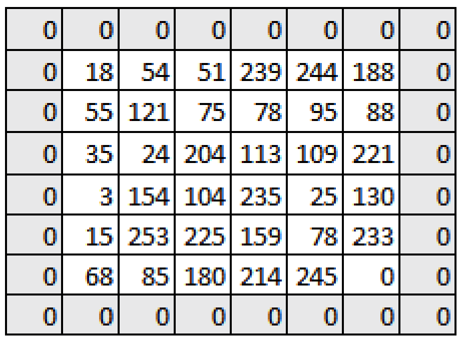
18	54	51	239	244
55	121	75	78	95
35	24	204	113	109
3	154	104	235	25
15	253	225	159	78

WEIGHT

1	0	1
0	1	0
1	0	1

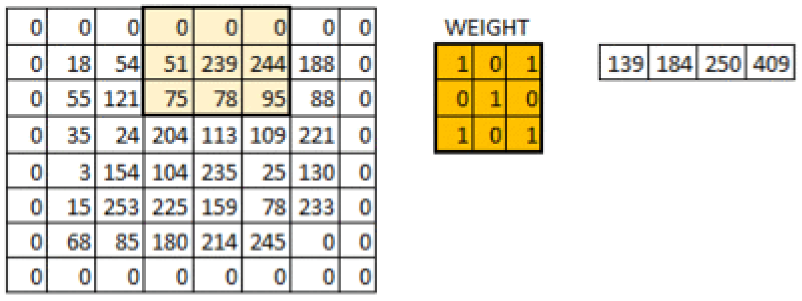
429	686
633	

As you can see the size of image keeps on reducing as we increase the stride value. Padding the input image with zeros across it solves this problem for us. We can also add more than one layer of zeros around the image in case of higher stride values.



0	0	0	0	0	0	0	0
0	18	54	51	239	244	188	0
0	55	121	75	78	95	88	0
0	35	24	204	113	109	221	0
0	3	154	104	235	25	130	0
0	15	253	225	159	78	233	0
0	68	85	180	214	245	0	0
0	0	0	0	0	0	0	0

We can see how the initial shape of the image is retained after we padded the image with a zero. This is known as **same padding** since the output image has the same size as the input.



0	0	0	0	0	0	0	0
0	18	54	51	239	244	188	0
0	55	121	75	78	95	88	0
0	35	24	204	113	109	221	0
0	3	154	104	235	25	130	0
0	15	253	225	159	78	233	0
0	68	85	180	214	245	0	0
0	0	0	0	0	0	0	0

WEIGHT

1	0	1
0	1	0
1	0	1

139	184	250	409

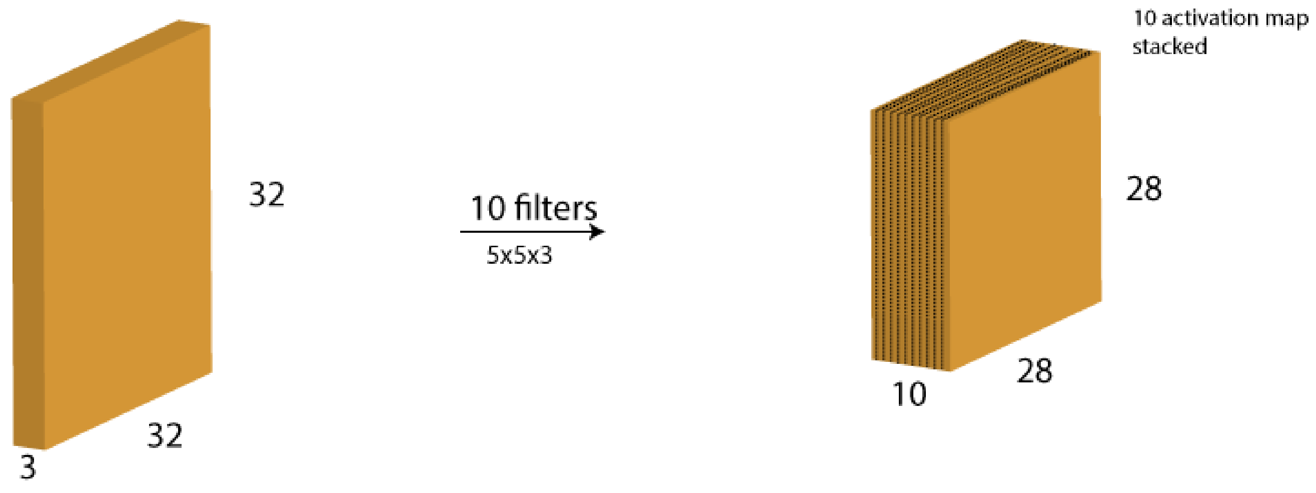
This is known as **same padding** (which means that we considered only the valid pixels of the input image). The middle 4*4 pixels would be the same. Here we have retained more information from the borders and have also preserved the size of the image.

Multiple filters and the activation map

One thing to keep in mind is that the depth dimension of the weight would be same as the depth dimension of the input image. The weight extends to the entire depth of the input image. Therefore, convolution with a single weight matrix would result into a convolved output with a single depth dimension. In most cases instead of a single filter(weight matrix), we have multiple filters of the same dimensions applied together.

The output from the each filter is stacked together forming the depth dimension of the convolved image. Suppose we have an input image of size 32*32*3. And we apply 10 filters of size 5*5*3 with valid padding. The output would have the dimensions as 28*28*10.

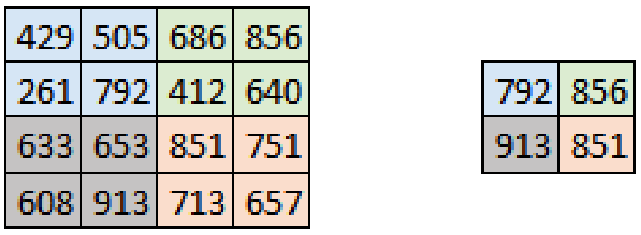
You can visualize it as –



This activation map is the output of the convolution layer.

2.2 The Pooling Layer

Sometimes when the images are too large, we would need to reduce the number of trainable parameters. It is then desired to periodically introduce pooling layers between subsequent convolution layers. Pooling is done for the sole purpose of reducing the spatial size of the image. Pooling is done independently on each depth dimension, therefore the depth of the image remains unchanged. The most common form of pooling layer generally applied is the max pooling.

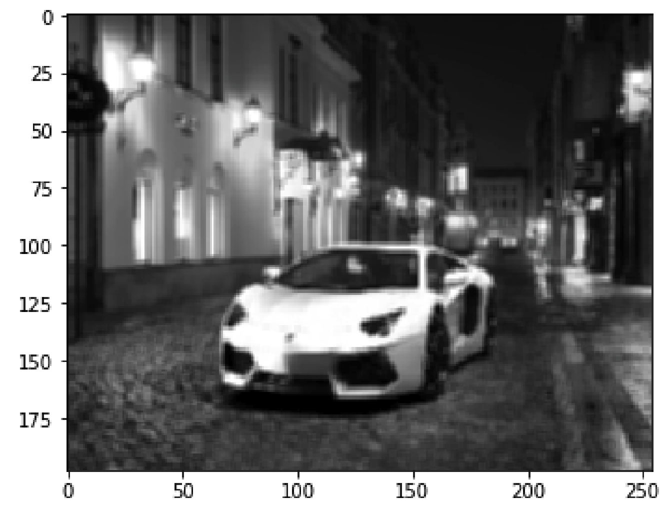


429	505	686	856
261	792	412	640
633	653	851	751
608	913	713	657

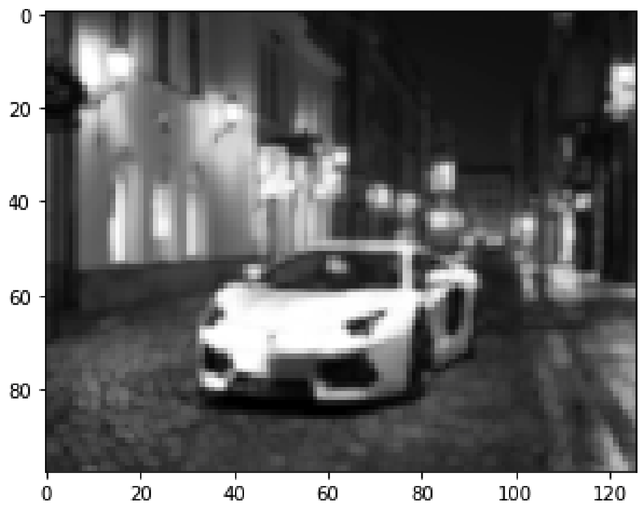
792	856
913	851

Here we have taken stride as 2, while pooling size also as 2. The max operation is applied to each depth dimension of the convolved output. As you can see, the 4*4 convolved output has become 2*2 after the max pooling operation.

Let's see how max pooling looks on a real image.



Convoluted image



MAX Pooling

As you can see I have taken convoluted image and have applied max pooling on it. The max pooled image still retains the information that it's a car on a street. If you look carefully, the dimensions if the image have been halved. This helps to reduce the parameters to a great extent.

Similarly other forms of pooling can also be applied like average pooling or the L2 norm pooling.

Output dimensions

It might be getting a little confusing for you to understand the input and output dimensions at the end of each convolution layer. I decided to take these few lines to make you capable of identifying the output dimensions. Three hyperparameter would control the size of output volume.

1. **The number of filters** – the depth of the output volume will be equal to the number of filter applied. Remember how we had stacked the output from each filter to form an activation map. The depth of the activation map will be equal to the number of filters.
2. **Stride** – When we have a stride of one we move across and down a single pixel. With higher stride values, we move large number of pixels at a time and hence produce smaller output volumes.
3. **Zero padding** – This helps us to preserve the size of the input image. If a single zero padding is added, a single stride filter movement would retain the size of the original image.

We can apply a simple formula to calculate the output dimensions. The spatial size of the output image can be calculated as([W-F+2P]/S)+1. Here, W is the input volume size, F is the size of the filter, P is the number of padding applied and S is the number of strides. Suppose we have an input image of size 32*32*3, we apply 10 filters of size 3*3*3, with single stride and no zero padding.

Here W=32, F=3, P=0 and S=1. The output depth will be equal to the number of filters applied i.e. 10.

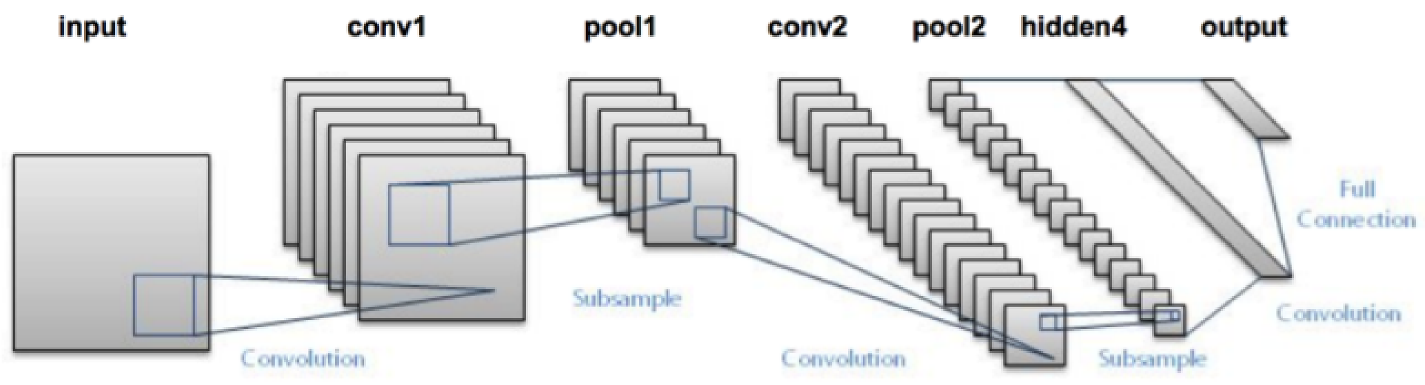
The size of the output volume will be ([32-3+0]/1)+1 = 30. Therefore the output volume will be 30*30*10.

2.3 The Output layer

After multiple layers of convolution and padding, we would need the output in the form of a class. The convolution and pooling layers would only be able to extract features and reduce the number of parameters from the original images. However, to generate the final output we need to apply a fully connected layer to generate an output equal to the number of classes we need. It becomes tough to reach that number with just the convolution layers. Convolution layers generate 3D activation maps while we just need the output as whether or not an image belongs to a particular class. The output layer has a loss function like softmax, to compute the error in prediction. Once the forward pass is complete the backpropagation begins to update the weight and biases for error and loss reduction.

3. Putting it all together – How does the entire network look like ?

CNN as you can now see is composed of various convolutional and pooling layers. Let's see how the network looks like.



- We pass an input image to the first convolutional layer. The convoluted output is obtained as an activation map. The filters applied in the convolution layer extract relevant features from the input image to pass further.
- Each filter shall give a different feature to aid the correct class prediction. In case we need to retain the size of the image, we use same padding(zero padding), other wise valid padding is used since it helps to reduce the number of features.
- Pooling layers are then added to further reduce the number of parameters
- Several convolution and pooling layers are added before the prediction is made. Convolutional layer help in extracting features. As we go deeper in the network more specific features are extracted as compared to a shallow network where the features extracted are more generic.
- The output layer in a CNN as mentioned previously is a fully connected layer, where the input from the other layers is flattened and sent so as the transform the output into the number of classes as desired by the network.
- The output is then generated through the output layer and is compared to the output layer for error generation. A loss function(generally softmax) is defined in the fully connected output layer to compute the mean square loss. The gradient of error is then calculated.
- The error is then backpropagated to update the filter(weights) and bias values.
- One training cycle is completed in a single forward and backward pass.

4. Using CNN to classify images in KERAS

Let's try taking an example where we input several images of cats and dogs and we try to classify these images into their respective animal category. This is a classic problem of image recognition and classification. What the machine needs to do is it needs to see the image and understand by the various features as to whether its a cat or a dog.

The features can be like extracting the edges, or extracting the whiskers of a cat etc. The convolutional layer would extract these features. Let's take a hand on the data set.

These are the examples of some of the images in the dataset.

we would first need to resize these images to get them all in the same shape. This is something we would generally need to do while handling images, since while capturing images, it would be impossible to capture all images of the same size.

For simplicity of your understanding I have just used a single convolution layer and a single pooling layer, which generally doesn't happen when we're trying to make predictions.

#import various packages

```
import os
import numpy as np
import pandas as pd
import scipy
import sklearn
import keras
from keras.models import Sequential
import cv2
from skimage import io
%matplotlib inline
```

#Defining the File Path

```
cat=os.listdir("/mnt/hdd/datasets/dogs_cats/train/cat")
dog=os.listdir("/mnt/hdd/datasets/dogs_cats/train/dog")
filepath="/mnt/hdd/datasets/dogs_cats/train/cat/"
filepath2="/mnt/hdd/datasets/dogs_cats/train/dog/"
```

#Loading the Images

```
images=[]
label = []
for i in cat:
image = scipy.misc.imread(filepath+i)
images.append(image)
label.append(0) #for cat images

for i in dog:
image = scipy.misc.imread(filepath2+i)
images.append(image)
label.append(1) #for dog images
```

#resizing all the images

```
for i in range(0,23000):
images[i]=cv2.resize(images[i],(300,300))
```

#converting images to arrays

```
images=np.array(images)
label=np.array(label)
```

# Defining the hyperparameters

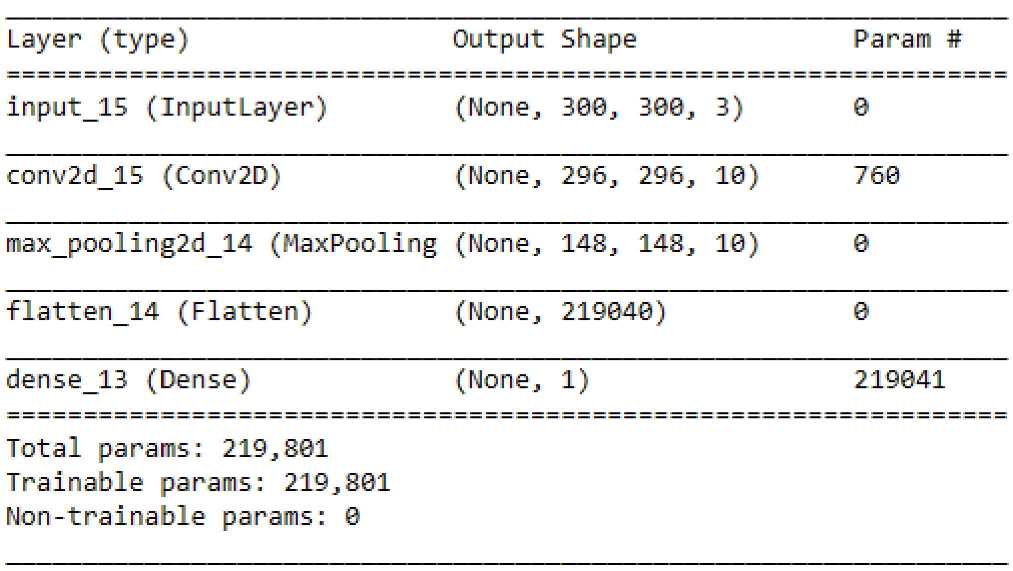
```
filters=10
filtersize=(5,5)

epochs =5
batchsize=128

input_shape=(300,300,3)
```

#Converting the target variable to the required size

```
from keras.utils.np_utils import to_categorical
label = to_categorical(label)
```

#Defining the model

```
model = Sequential()

model.add(keras.layers.InputLayer(input_shape=input_shape))

model.add(keras.layers.convolutional.Conv2D(filters, filtersize, strides=(1, 1),
padding='valid', data_format="channels_last", activation='relu'))
model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))
model.add(keras.layers.Flatten())

model.add(keras.layers.Dense(units=2, input_dim=50,activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=
['accuracy'])
model.fit(images, label, epochs=epochs, batch_size=batchsize,validation_split=0.3)

model.summary()
```



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_15 (InputLayer)        (None, 300, 300, 3)       0
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 296, 296, 10)      760
_________________________________________________________________
max_pooling2d_14 (MaxPooling (None, 148, 148, 10)      0
_________________________________________________________________
flatten_14 (Flatten)         (None, 219040)            0
_________________________________________________________________
dense_13 (Dense)             (None, 1)                 219041
=================================================================
Total params: 219,801
Trainable params: 219,801
Non-trainable params: 0
_________________________________________________________________
```

In this model, I have only used a single convolution and Pooling layer and the trainable parameters are 219,801. Wonder how many would I have had if i had used an MLP in this case. You can reduce the number of parameters by further by adding more convolution and pooling layers. The more convolution layers we add the features extracted would be more specific and intricate.

End Notes