Lecture Notes

Convolutional Neural Network

Convolutional Neural Networks, or CNNs, are specialized architectures which work particularly well with visual data, i.e. images and videos. They have been largely responsible for revolutionizing 'deep learning' by setting new benchmarks for many image processing tasks that were very recently considered extremely hard. Although the vanilla neural networks (MLPs) can learn extremely complex functions, their architecture does not exploit what we know about how the brain reads and processes images. For this reason, although MLPs are successful in solving many complex problems, they haven't been able to achieve any major breakthroughs in the image processing domain.

The ImageNet Challenge

CNNs had first demonstrated their extraordinary performance in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC). The ILSVRC uses a list of about 1000 image categories or "classes" and has about 1.2 million training images. The original challenge is an image classification task. You can see the impressive results in Figure 1 where they now outperform humans (having 5% error rate). The error rate of the ResNet, a recent variant in the CNN family, is close to 3%.

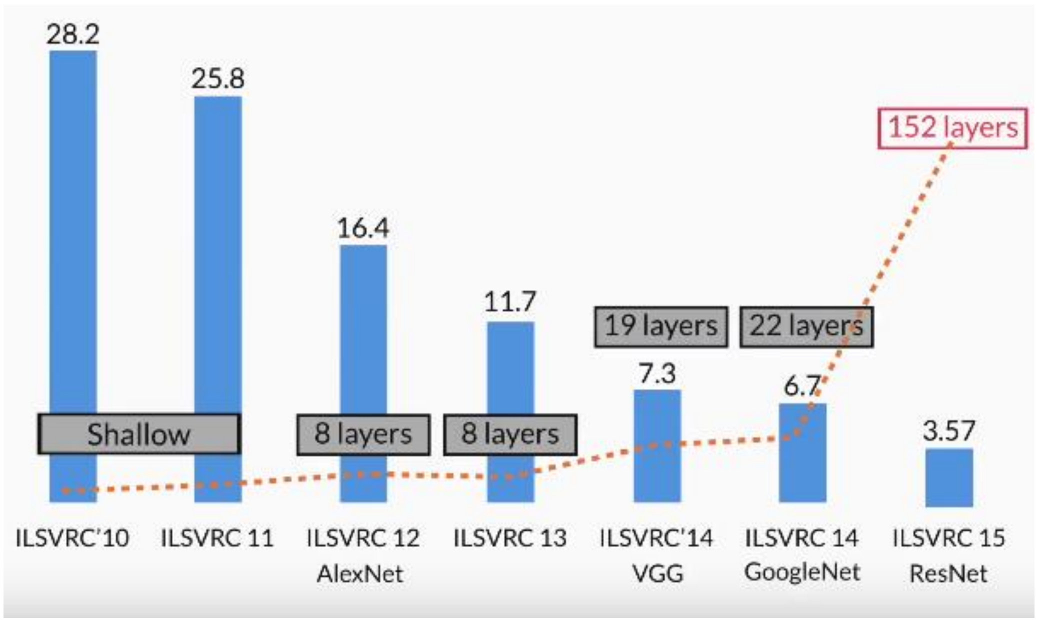


Figure 1: ImageNet Challenge

Reading Digital Images

An input to any neural network should be numeric. Images are naturally represented as arrays (or matrices) of numbers.

Greyscale Images

Images are made up of pixels. For most images, pixel values are integers that range from 0(black) to 255(white). The range 0-255 represents the colour intensity of each pixel.

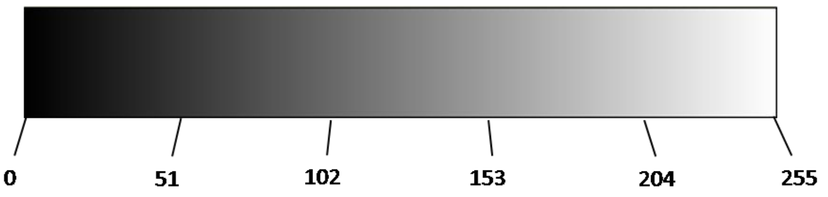
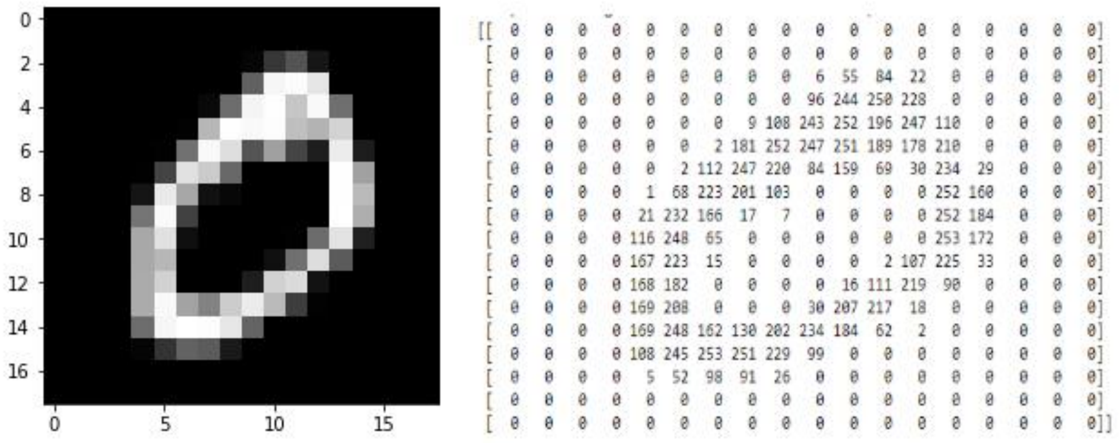


Figure 2: Pixel value in greyscale image

Figure 2 represents 18x18 greyscale image of ‘0’ and corresponding matrix.

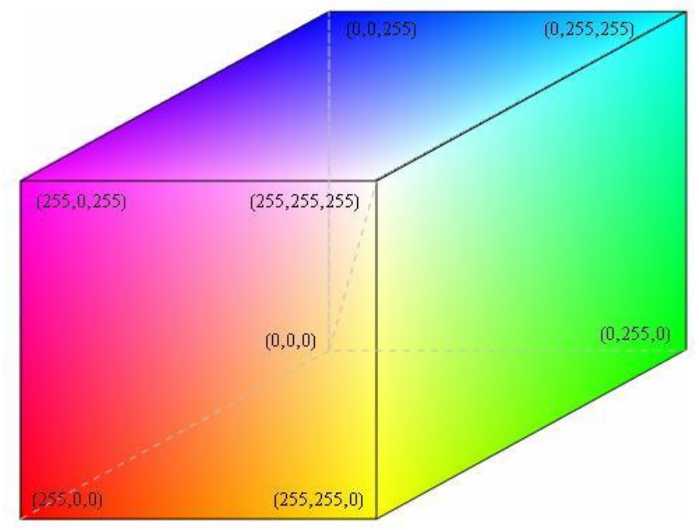


```
[[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   6  55  84  22   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0  96 244 250 228   0   0   0   0   0]
 [  0   0   0   0   0   0   0   9 108 243 252 196 247 110   0   0   0   0]
 [  0   0   0   0   0   0   2 181 252 247 251 189 178 210   0   0   0   0]
 [  0   0   0   0   0   2 112 247 220  84 159  69  30 234  29   0   0   0]
 [  0   0   0   0   1  68 223 201 103   0   0   0   0 252 160   0   0   0]
 [  0   0   0   0  21 232 166  17   7   0   0   0   0 252 184   0   0   0]
 [  0   0   0   0 116 248  65   0   0   0   0   0   0 253 172   0   0   0]
 [  0   0   0   0 167 223  15   0   0   0   0   2 107 225  33   0   0   0]
 [  0   0   0   0 168 182   0   0   0   0  16 111 219  90   0   0   0   0]
 [  0   0   0   0 169 208   0   0   0  30 207 217  18   0   0   0   0   0]
 [  0   0   0   0 169 248 162 130 202 234 184  62   2   0   0   0   0   0]
 [  0   0   0   0 108 245 253 251 229  99   0   0   0   0   0   0   0   0]
 [  0   0   0   0   5  52  98  91  26   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]]
```

Figure 3: 18x18 greyscale image

Colour Images

All colours can be made by mixing red, blue and green at different degrees of “saturation” (0-100% intensity) as you can see in Figure 4. For example, a pure red pixel has 100% intensity of red, and 0% intensity of blue and green. So, it is represented as (255,0,0). White is the combination of 100% intensity of red, green and blue. So, it is represented as (255,255,255).



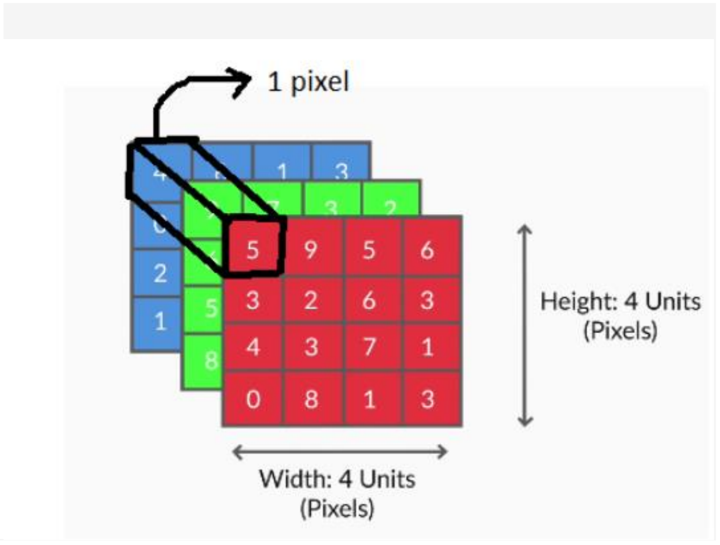


Figure 4: Pixel values in colour image

Figure 5: Color image represented as matrix

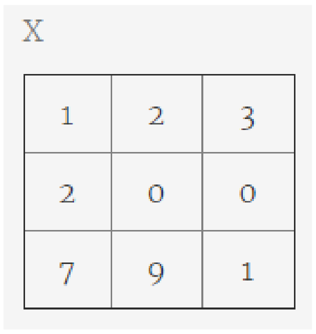
Figure 5 represents the color image of size 4x4x3, where height is 4, width is 4, the number of channels is 3(RGB). The number of pixels is 16 (height x width).

Next, you will the main concepts in CNNs:

- Convolution, and why it 'shrinks' the size of the input image
- Stride and Padding
- Pooling layers
- Feature maps

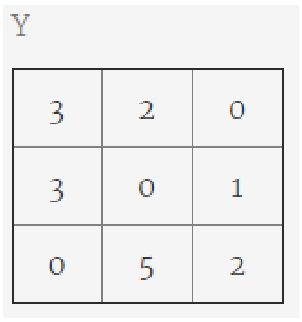
Convolution

Mathematically, the convolution operation is the summation of the element-wise product of two matrices. Let’s take two matrices, X and Y. If you 'convolve the image X using the filter Y', this operation will produce the matrix Z. Let’s say when we have X and Y of the same dimension.


X

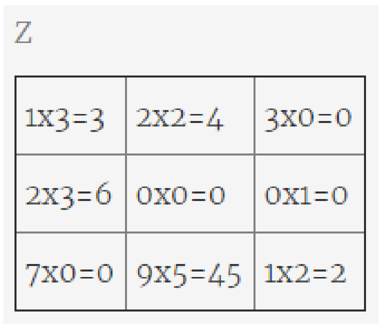
1	2	3
2	0	0
7	9	1

*


Y

3	2	0
3	0	1
0	5	2

=


Z

1X3=3	2X2=4	3X0=0
2X3=6	0X0=0	0X1=0
7X0=0	9X5=45	1X2=2

Let’s see another case when Image size is 5x5 and filter size is 3x3.

Image

1	0	3	7	2
5	7	10	0	7
4	12	0	2	0
0	1	11	1	3
10	7	0	8	1

$*$

Filter

1	0	1
-1	0	0
0	1	0

You can see the result of convolution in Figure 6.

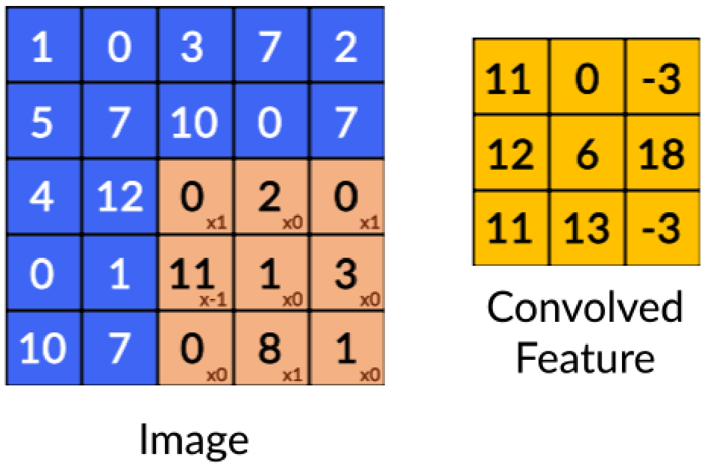


Figure 6

The basic idea of filters is to detect desired features (such as vertical or horizontal edges) through convolution. In the Figure 7, filter extracts edge present in the image.

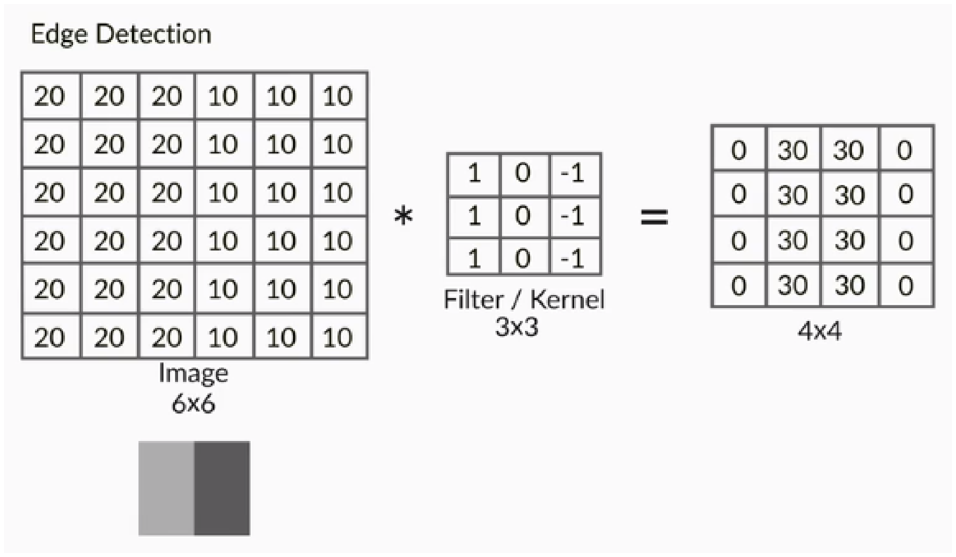


Figure 7: Edge Detection

Convolution in Depth

Let's take image with RGB channel as in Figure 8 and convolve with their corresponding filters. The result is shown in Figure 9.

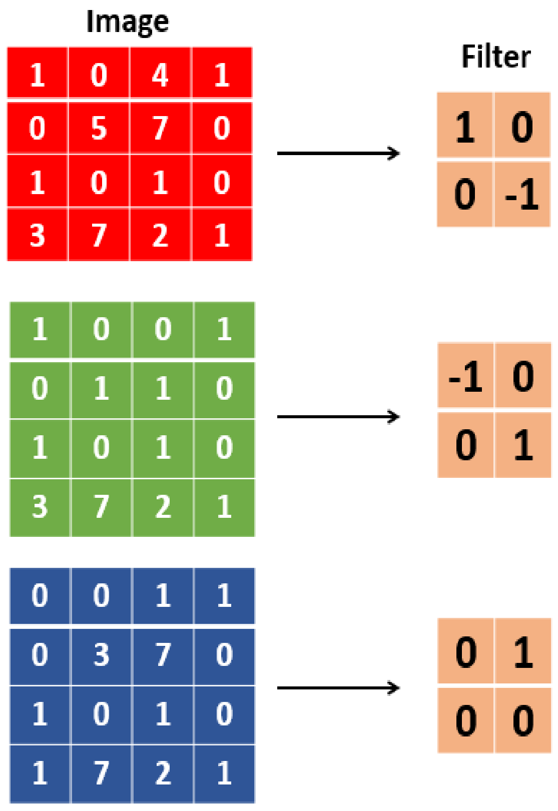


Figure 8: RGB channels of Image with corresponding filters.

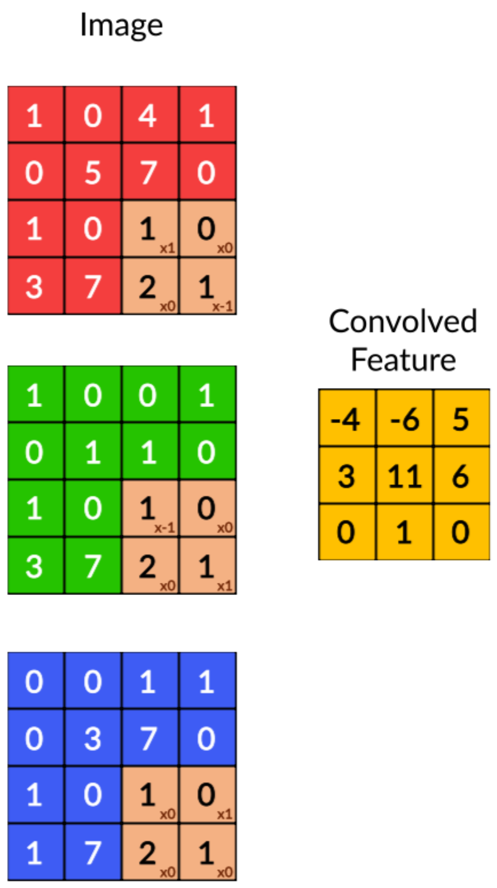


Figure 9: Convolution in Depth

Input, Filter and Output Dimension?

In convolution, the number of filters should match the depth of the image. Say if we have a greyscale image with depth of 1 of size 10x10, the filter also needs to have the depth of 1 of size say 5x5.

In case of a colour image with RGB channel, say an image of size 10x10x3 where 3 is the depth, we need a filter of dimension 5x5x3 where the depth of 3 is present in both the image and the filter. It is also to be noted that the result of convolution is a 2D array/matrix, irrespective of the depth of the input. So, if we use multiple filters, there will be multiple 2D array/matrix and we can stack them up.

Problem with reducing spatial dimension (height and width) in convolution?

If we want to make a very deep convolutional neural network with lots of convolutional layers, the spatial dimension will reduce with each convolutional operation, and we may not be able to make the layers deep enough. But we also want a compact representation of the image for classification. To overcome this problem, we designed a schema in which the spatial dimension of the input should be preserved with each convolutional operation through the choice of proper stride and padding while reducing the representation periodically through pooling. So, the pooling layer alone is responsible for down-sampling the volume spatially.

Stride and Padding

***Stride*:** Stride is the number of pixels we move the filter (both horizontally and vertically) to perform convolution operation.

***Padding*:** Padding is the number of pixels we add all around the image. As you can in Figure 10 that the padding of 1 is used.

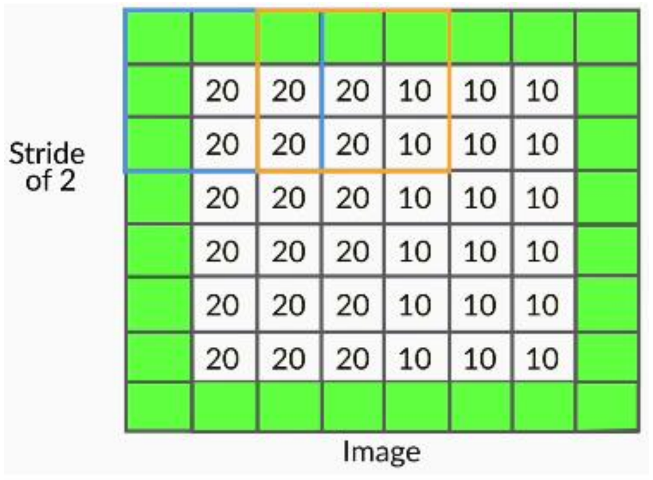


Figure 10: Stride of 2 and Padding of 1

Why we use a stride of 1 in convolution?

Moving filter with smaller stride helps to capture more fine-grained information, Also, moving with larger stride helps to reduce the total number of operations in convolution and reduce the spatial dimension. But, when we are just interested in capturing more fine-grained features, and pooling layer takes care of reducing the spatial dimension, so we use a stride of 1.

Why use padding?

Padding operation helps to maintain the height and width of the output (same as input) that we get after the convolution. Also, it helps to preserve the information at the borders. There are also alternate ways to do convolution by reducing the filter size as we approach the edge, but they are not common.

Zero-padding: Padding image with '0' all around the image.

Weights in CNN

In the case of the neural network, we don't define a specific filter. We just initialize the weights and they are learned during backpropagation. In Figure 8, the filter has three channels, and each channel of the filter convolves to the corresponding channel of the image. Thus, each step in the convolution involves the element-wise multiplication of 12 (4 operations in each RGB) pairs of numbers and adding the resultant products to get a single scalar output. Note that in each step, a single scalar number is generated, and at the end of the convolution, a 2D array is generated.

You can express the convolution operation as a dot product between the weights and the input image. If you treat the (2, 2, 3) filter as a vector w of length 12, and the 12 corresponding elements of the input image as the vector *p* (i.e. both unrolled to a 1D vector), each step of the convolution is simply the dot product of w^T and *p*. The dot product is computed at every patch to get a (3, 3) output array, as shown above.

Apart from the weights, each filter has only 1 also bias.

$$w^T.x+b=\begin{bmatrix} sum(w^T.p_{11}) & sum(w^T.p_{12}) & sum(w^T.p_{13}) \\ sum(w^T.p_{21}) & sum(w^T.p_{22}) & sum(w^T.p_{23}) \\ sum(w^T.p_{31}) & sum(w^T.p_{32}) & sum(w^T.p_{33}) \end{bmatrix} + \begin{bmatrix} b & b & b \\ b & b & b \\ b & b & b \end{bmatrix}$$

$$= \begin{bmatrix} -4 & -6 & 5 \\ 3 & 11 & 6 \\ 0 & 1 & 0 \end{bmatrix} + \begin{bmatrix} b & b & b \\ b & b & b \\ b & b & b \end{bmatrix}$$

Feature Map

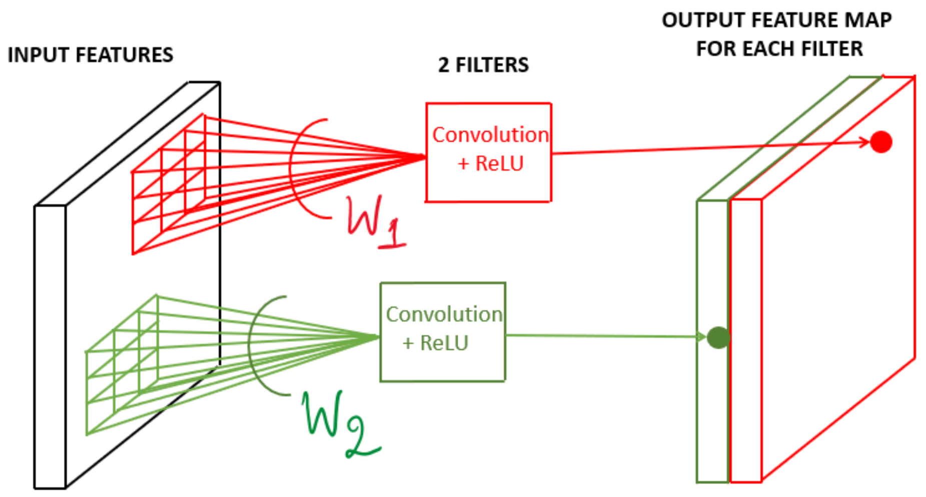


Figure 11: Feature Map

The term 'feature map' refers to the (non-linear) **output of the activation function**, not what goes into the activation function (i.e. the output of the convolution). Generally, **ReLU** is used as an activation function, except in the last layer where we use **SoftMax** activation for classification.

Let's continue the above example, use bias (b) of value '2' and apply activation ReLU to get feature map.

$$\begin{bmatrix} -4 & -6 & 5 \\ 3 & 11 & 6 \\ 0 & 1 & 0 \end{bmatrix} + \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix} = \begin{bmatrix} -2 & -4 & 7 \\ 5 & 13 & 8 \\ 2 & 3 & 2 \end{bmatrix} \xrightarrow[\text{ReLU}]{\text{Activation}} \begin{bmatrix} 0 & 0 & 7 \\ 5 & 13 & 8 \\ 2 & 3 & 2 \end{bmatrix}$$

So, now we get one feature map as [0 0 7; 5 13 8; 2 3 2]. As you can see in the figure 11, when you have multiple filter, you will get multiple feature map.

Pooling

After extracting features (as feature maps), CNNs typically aggregate these features using the pooling layer to make the representation more compact. We already used padding to make the width and height of feature map same as that of input. But we also need compact representation of the feature map. We take aggregate over a patch of feature map to get the output of pooling. Most popular are 'Max Pooling' and 'Average Pooling'. Max pooling is more popular than average pooling as it has shown to work better average pooling.

Max pooling: If any one of the patches says something strongly about the presence of a certain feature, then the pooling layer counts that feature as 'detected'.
Average pooling: If one patch says something very firmly but the other ones disagree, the pooling layer takes the average to find out.

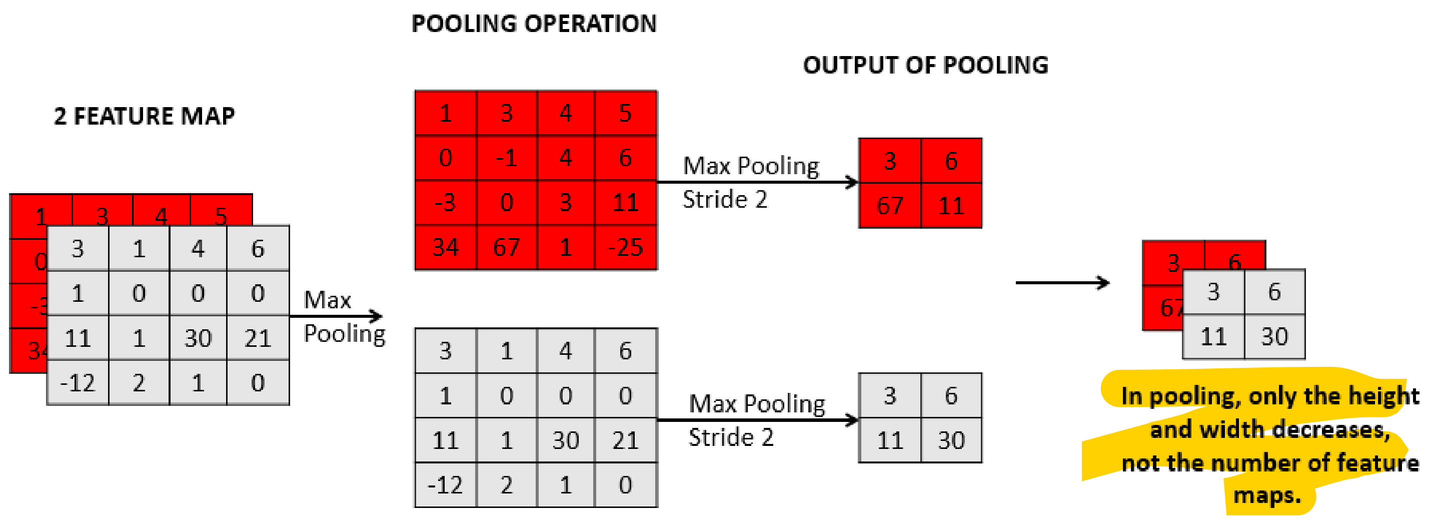


Figure 12: Pooling

In Figure 12, you can see that Pooling only reduces the height and width, keeping the depth of the input same in the output.

Important Formulas

Given the following size :

- Image - n x n
- Filter - k x k
- Padding - P
- Stride - S

After padding, we get an image of size (n + 2P) x (n+2P). After we convolve this padded image with the filter, we get:

$$\text{Size of convolved image} = \left(\frac{n+2P-k}{S}+1\right), \left(\frac{n+2P-k}{S}+1\right)$$

Similarly, to find the size of the **Output after Pooling, you can use the above formula, and make the value of P=0,** since there is no padding in Pooling operation.

Shared Weights

The concept of shared weights is used in the convolutional layers to control the number of parameters. It means that same filter is used across the image when extracting a feature instead of changing weights at every different position (x,y), as if the weights are shared across different positions. We make assumption that if one filter extracts a feature at some spatial position (x,y), then it should also be useful to compute the same feature at a different position (x1,y1).

Flatten, Fully Connected and Softmax

Why we need to Flatten?

The output of the convolutional layer is 3-dimension matrix say 32x32x256. But for classification, we need a dense layer. Say, we if we want to classify an image into 5 classes, finally we need 5 neurons what will output probability. So, we need to convert the 3-dimensional matrix into fully connected. Generally, we use pooled feature map to convert into the dense layer.

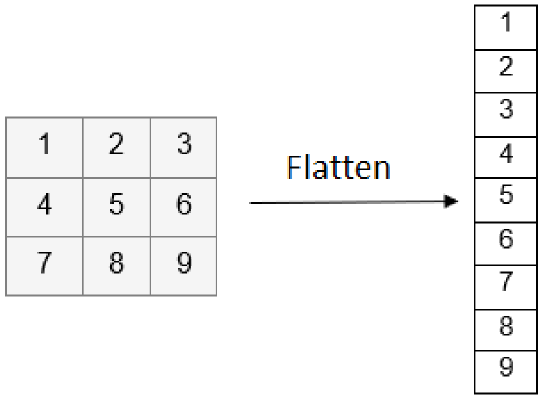


Figure 13: Flattening a 2D matrix. Similarly, we can flatten 3D matrix.

Why we need to Fully connected layer?

After flattening the layer, we need to look at all the features extracted by the convolutional layer before we classify them. So, we need the dense layer in which all neurons in the next layer is connected to all the neurons in the previous layer. On the top of fully connected, we use 'softmax' activation for classification.

Layer Pattern

One of the common layer pattern in CNN is stacks a few CONV->ReLU layers, followed by POOL layers, and repeats this process until the spatial size of the image gets sufficiently small. After that, we connect the FLATTEN layer followed by FC->SOFTMAX layer to perform classification.

INPUT -> ((CONV->ReLU) * X -> POOL) * Y) -> FLATTEN -> (FC-> ReLU)*Z -> FC->SOFTMAX

Ideally, the values of X, Y and Z can be any number. When comparing the above pattern with the VGG16 network, you can find possible values of them. Also, you have trained CIFAR-10 dataset on the above pattern.

Although above architecture is most commonly used and simple, it may give good accuracy, but may not be a most efficient pattern. Let's see the graph from the paper " An Analysis of Deep Neural Network Models for Practical Applications".

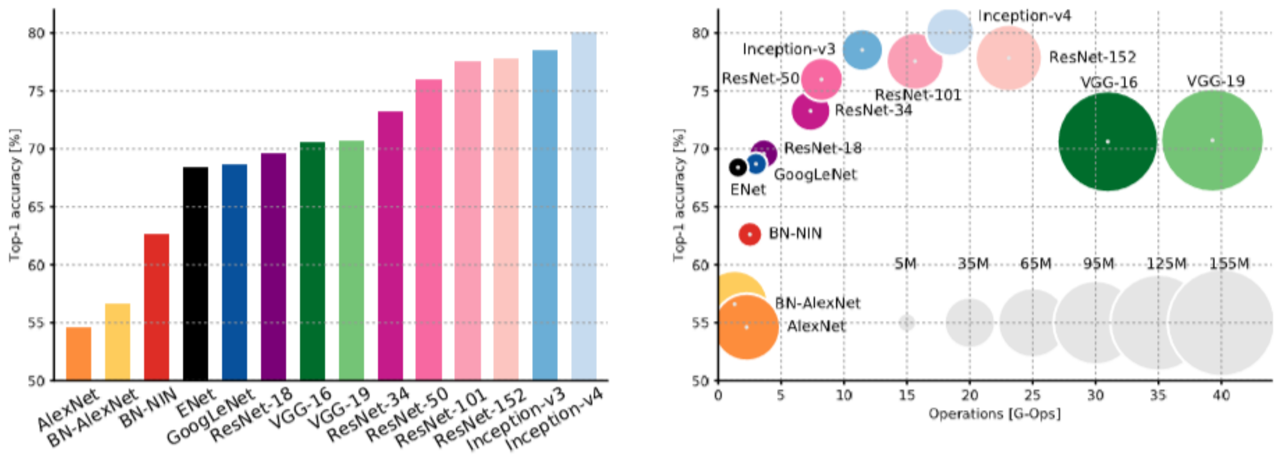


Figure 1: **Top1** ***vs.*** **network.** Single-crop top-1 validation accuracies for top scoring single-model architectures. We introduce with this chart our choice of colour scheme, which will be used throughout this publication to distinguish effectively different architectures and their correspondent authors. Notice that networks of the same group share the same hue, for example ResNet are all variations of pink.

Figure 2: **Top1** ***vs.*** **operations, size $\propto$ parameters.** Top-1 one-crop accuracy versus amount of operations required for a single forward pass. The size of the blobs is proportional to the number of network parameters; a legend is reported in the bottom right corner, spanning from 5×10^6 to 155×10^6 params. Both these figures share the same *y*-axis, and the grey dots highlight the centre of the blobs.

Figure 14

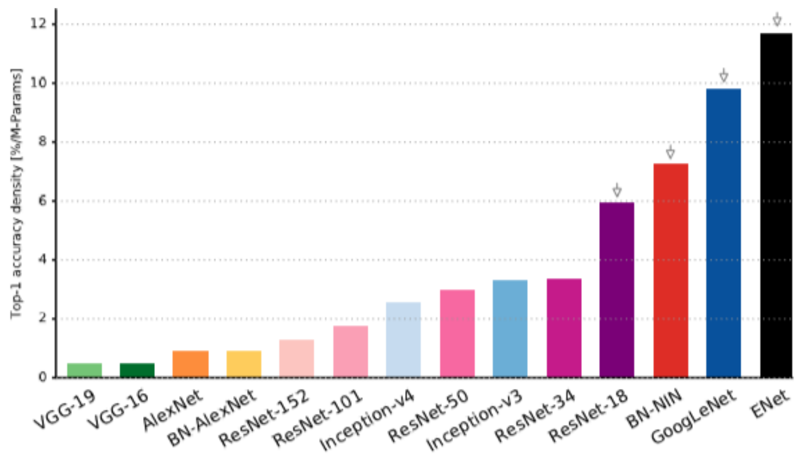


Figure 10: **Accuracy per parameter *vs.* network.** Information density (accuracy per parameters) is an efficiency metric that highlight that capacity of a specific architecture to better utilise its parametric space. Models like VGG and AlexNet are clearly oversized, and do not take fully advantage of their potential learning ability. On the far right, ResNet-18, BN-NIN, GoogLeNet and ENet (marked by grey arrows) do a better job at "squeezing" all their neurons to learn the given task, and are the winners of this section.

Figure 15

In the above Figures 14 and 15, you can clearly see that VGG16 has least information density. If you need good accuracy, better go for models like GoogleNet, ResNet and Inception. Let's see some of the other arrangement of layers.

VGGNet

The key idea in VGGNet was deeper network by using smaller filters. It is important to note that the small size of the filter is preferred than the large filter. Consider the example below. Say we have a 5 x 5 image, and in two different convolution experiments, we use two different filters of size (5, 5) and (3, 3) respectively. We say that the stack of two (3, 3) filters has the same effective receptive field as that of one (5, 5) filter. This is because both these convolutions produce the same output (of size 1 x1 here) whose receptive field is the same 5 x 5 image as you can see in Figure 16.

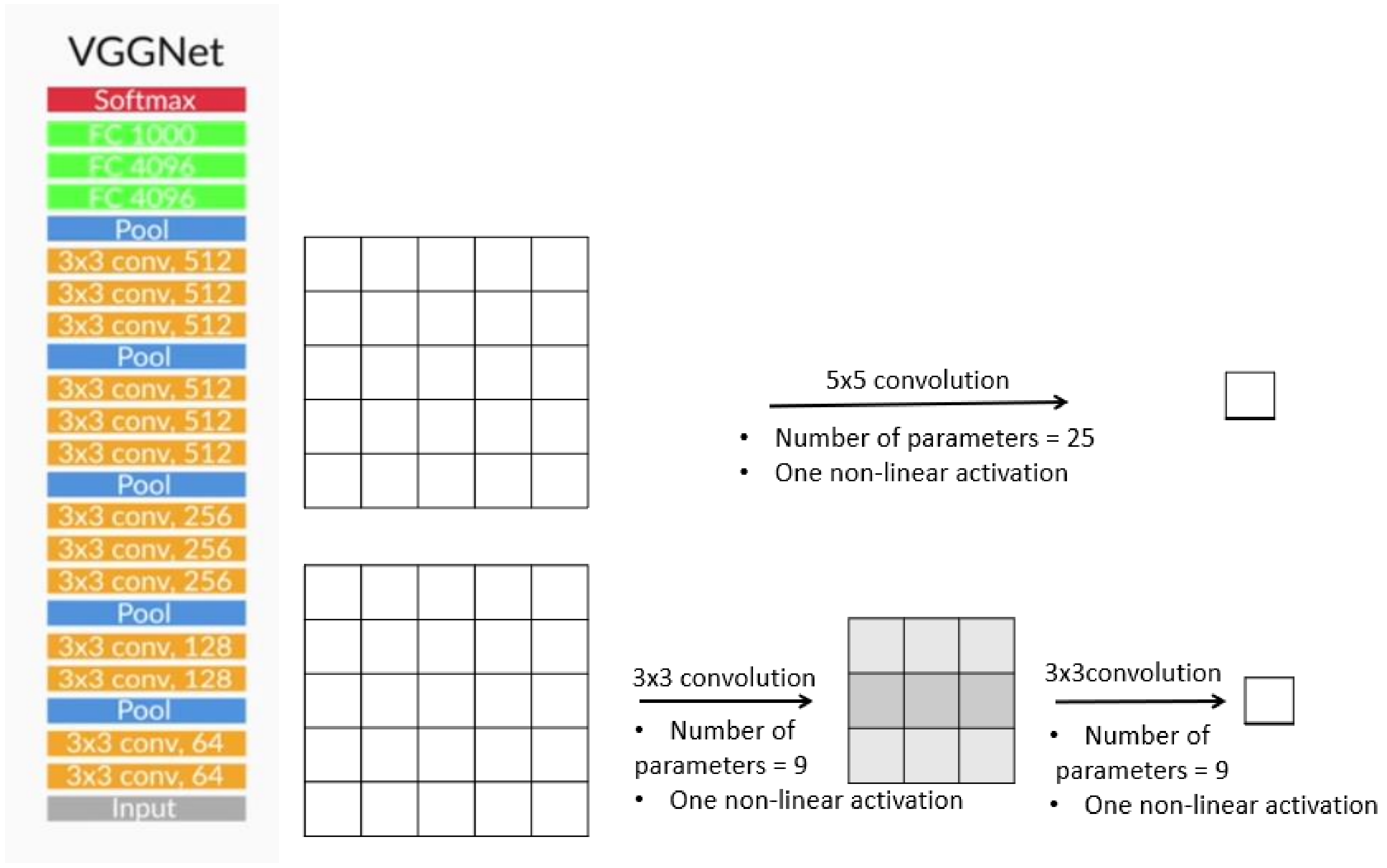


Figure 16: 3x3 and 5x5 kernel

GoogleNet

GoogleNet stacks multiple filter along with 1x1 filter and pooling layer. Inception module as in Figure 17, is the basic building block of the network. It is recommended to go through the paper for further understanding.

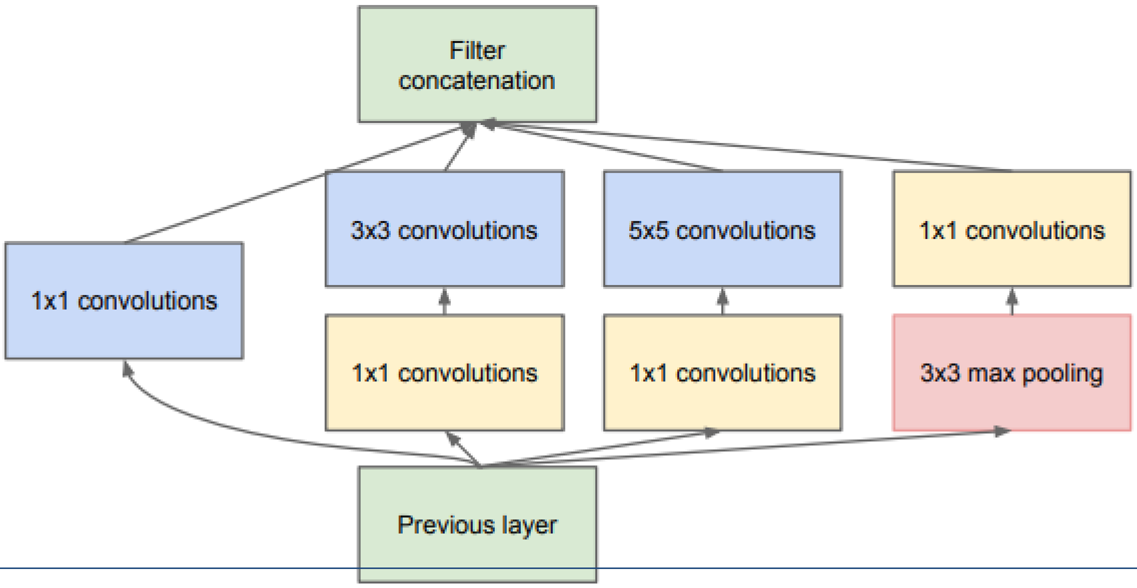


Figure 17: Inception Module

ResNet

According to Kaiming He in paper ‘Deep Residual Learning for Image Recognition’, “Driven by the significance of depth, a question arises: *Is learning better networks as easy as stacking more layers?* An obstacle to answering this question was the notorious problem of vanishing/exploding, which hamper convergence from the beginning. This problem, however, has been largely addressed by normalized initialization and intermediate normalization layers, which enable networks with tens of layers to start converging for stochastic gradient descent (SGD) with backpropagation. When deeper networks are able to start converging, a degradation problem has been exposed: with the network depth increasing, accuracy gets saturated (which might be unsurprising) and then degrades rapidly. Unexpectedly, such degradation is not caused by overfitting, and adding more layers to a suitably deep model leads to higher training error. The degradation (of training accuracy) indicates that not all systems are similarly easy to optimize.”
The **skip connection mechanism** was the key feature of the ResNet which enabled the training of very deep networks. The performance should not decrease, if not increase when adding a new layer.

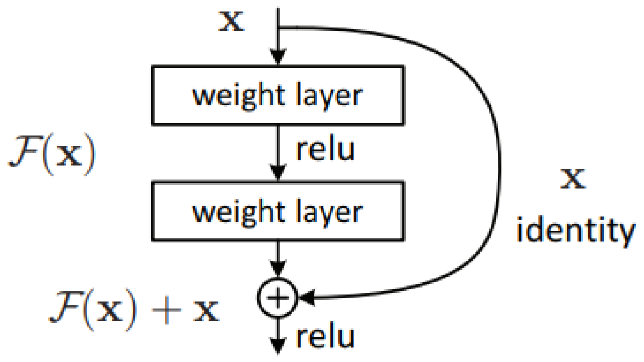

x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 18: Residual Learning: A building block

Transfer Learning

So far, we have discussed CNN based networks which were trained on millions of images of various classes. The ImageNet dataset itself has about 1.2 million images of 1000 classes. However, what these models have 'learnt' is not confined to the ImageNet dataset (or a classification problem). Earlier we had discussed that CNNs are basically **feature-extractors,** i.e. the convolutional layers learn a representation of an image, which can then used for any task such as classification, object detection, etc. Also in practise, very few people train an entire CNN from scratch because it is really difficult to find sufficient size dataset. There are two main ways of using pre-trained nets for transfer learning:

1. **'Freeze'** the initial layers, i.e. use the same weights and biases that the network has learnt from some other task, remove the few last layers of the pre-trained model, add your own new layer(s) at the end and **train only the newly added layer(s).**

2. **Fine-tune the model. Retrain** all or few layers **starting (initialising) from the weights and biases** that the net has already learnt. Retain using backpropagation. Since you don't want to unlearn a large fraction of what the pre-trained layers have learnt. So, for fine-tuning, we will choose a low learning rate.

Let's see the scenario when to use transfer learning.

1. If the task is a lot similar to that the pre-trained model had learnt from, you can use most of the layers except the last few layers which you can retrain. If the new dataset is small than the similar original dataset, you can use most of the layers as even the higher layer features may be useful. But since the new dataset is small it is not advisable to fine-tune the model due to overfitting concerns. If the new dataset is large, then you can fine-tune the model as you have enough data and need not to worry about overfitting. If you don't have much confidence, you can even fine-tune just few last layers instead of fine-tuning the whole model.

2. If you think there is less similarity in the tasks, you can use only a few initial trained weights for your task as they are more generic and not class specific.