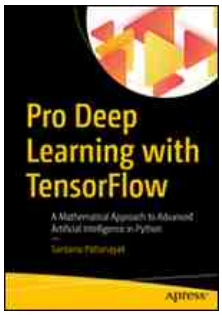


Pro Deep Learning with TensorFlow: A Mathematical Approach to Advanced Artificial Intelligence in Python

by Santanu Pattanayak

Chapter 3: Convolutional Neural Networks

S. Pattanayak, *Pro Deep Learning with TensorFlow*, https://doi.org/10.1007/978-1-4842-3096-1_3

Overview

Artificial neural networks have flourished in recent years in the processing of unstructured data, especially images, text, audio, and speech. Convolutional neural networks (CNNs) work best for such unstructured data. Whenever there is a topology associated with the data, convolutional neural networks do a good job of extracting the important features out of the data. From an architectural perspective, CNNs are inspired by multi-layer Perceptrons. By imposing local connectivity constraints between neurons of adjacent layers, CNN exploits local spatial correlation.

The core element of convolutional neural networks is the processing of data through the convolution operation. Convolution of any signal with another signal produces a third signal that may reveal more information about the signal than the original signal itself. Let's go into detail about convolution before we dive into convolutional neural networks.

Convolution Operation

The convolution of a temporal or spatial signal with another signal produces a modified version of the initial signal. The modified signal may have better feature representation than the original signal suitable for a specific task. For example, by convolving a grayscale image as a 2D signal with another signal, generally called a filter or kernel, an output signal can be obtained that contains the edges of the original image. Edges in an image can correspond to object boundaries, changes in illumination, changes in material property, discontinuities in depth, and so on, which may be useful for several applications. Knowledge about the linear time invariance or shift invariance properties of systems helps one appreciate the convolution of signals better. We will discuss this first before moving on to convolution itself.

Linear Time Invariant (LTI) / Linear Shift Invariant (LSI) Systems

A system works on an input signal in some way to produce an output signal. If an input signal $x(t)$ produces an output, $y(t)$, then $y(t)$ can be expressed as

$$y(t) = f(x(t))$$

For the system to be linear, the following properties for scaling and superposition should hold true:

$$\textbf{\textit{Scaling}}: f(\alpha x(t)) = \alpha f(x(t))$$

$$\textbf{\textit{Superposition}}: f(\alpha x_1(t) + \beta x_2(t)) = \alpha f(x(t)) + \beta f(x_2(t))$$

Similarly, for the system to be time invariant or in general shift invariant,

$$f(x(t-\tau)) = y(t-\tau)$$

Such systems that have properties of linearity and shift invariance are termed linear shift invariant (LSI) systems in general. When such systems work on time signals, they are referred to as linear time invariant (LTI) systems. For the rest of the chapter, we will refer to such systems as LSI systems without any loss of generality. See Figure 3-1.

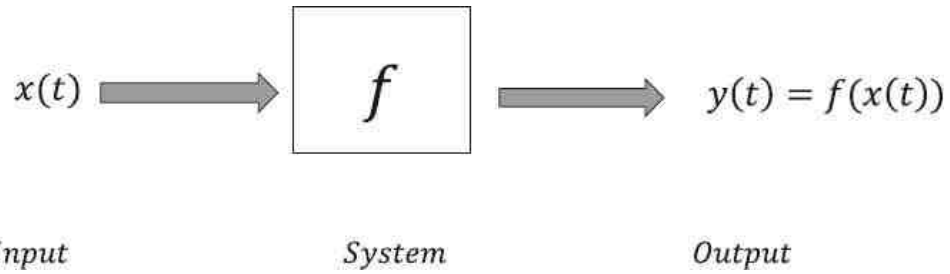


Figure 3-1: Input–Output system

The key feature of an LSI system is that if one knows the output of the system to an impulse response then one can compute the output response to any signal.

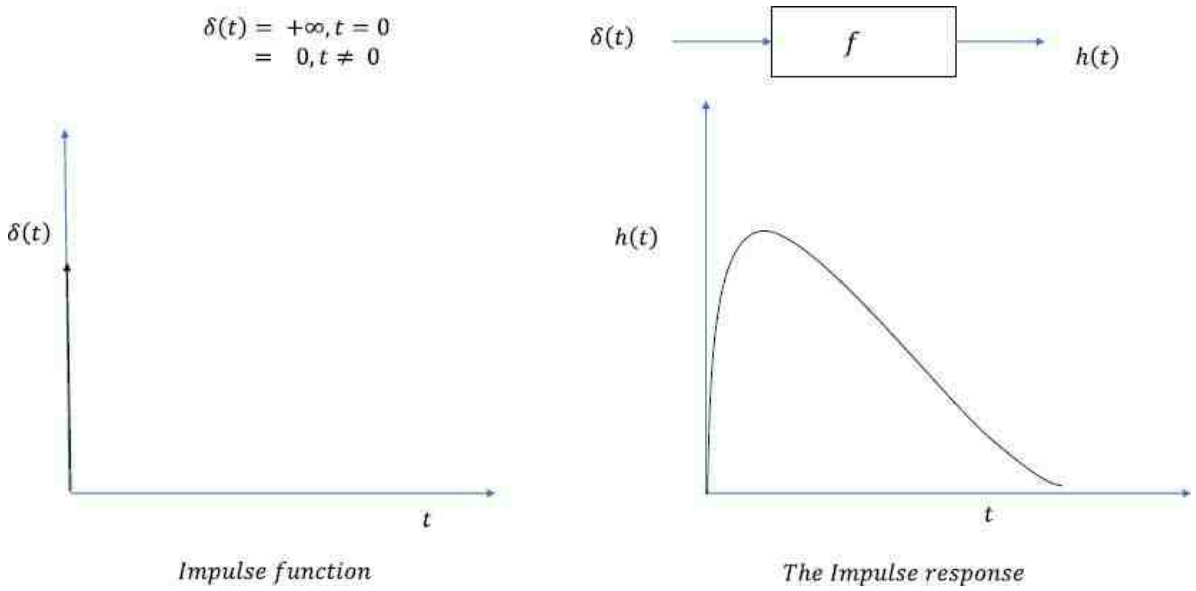


Figure 3-2a: Response of an LSI system to an impulse (Dirac Delta) function

In Figures 3-2a and 3-2b we illustrate the impulse response of the systems to different kinds of impulse functions. Figure 3-2a shows the continuous impulse response of the system to a Dirac Delta impulse, whereas Figure 3-2b shows the discrete impulse response of the system to a step impulse function. The system in Figure 3-2a is a continuous LTI system, and hence a Dirac Delta is required to determine its impulse response. On the other hand, the system in Figure 3-2b is a discrete LTI system and so a unit step impulse is needed to determine its impulse response.

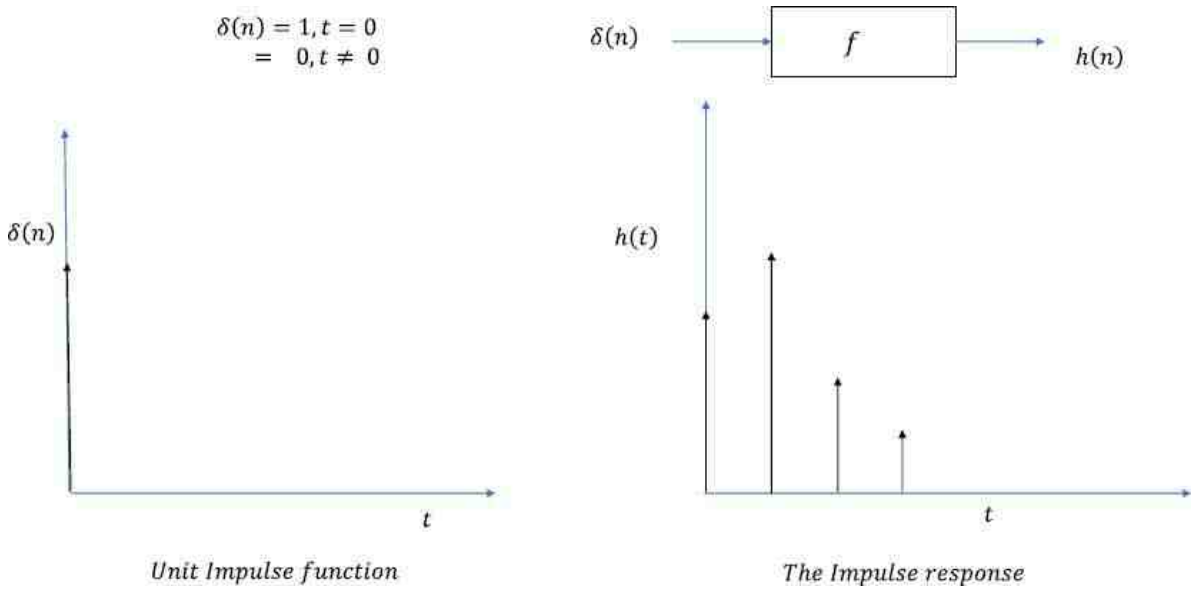


Figure 3-2b: Response of an LTI system to a unit step impulse

Once we know the response $h(t)$ of an LSI system to an impulse function $\delta(t)$ we can compute the response $y(t)$ of the LTI system to any arbitrary input signal $x(t)$ by convolving it with $h(t)$. Mathematically, it can be expressed as $y(t) = x(t)(*)h(t)$, where the (*) operation denotes convolution.

The impulse response of a system can either be known or be determined from the system by noting down its response to an impulse function. For example, the impulse response of a Hubble space telescope can be found out by focusing it on a distant star in the dark night sky and then noting down the recorded image. The recorded image is the impulse response of the telescope.

Convolution for Signals in One Dimension

Intuitively, convolution measures the degree of overlap between one function and the reversed and translated version of another function. In the discrete case,

$$y(t)=x(t)(*)h(t)=\sum_{\tau=-\infty}^{+\infty}x(\tau)h(t-\tau)$$

Similarly, in the continuous domain the convolution of two functions can be expressed as

$$y(t)=x(t)(*)h(t)=\int_{\tau=-\infty}^{+\infty}x(\tau)h(t-\tau)d\tau$$

Let's perform convolution of two discrete signals to better interpret this operation. See Figures 3-3a to 3-3c.

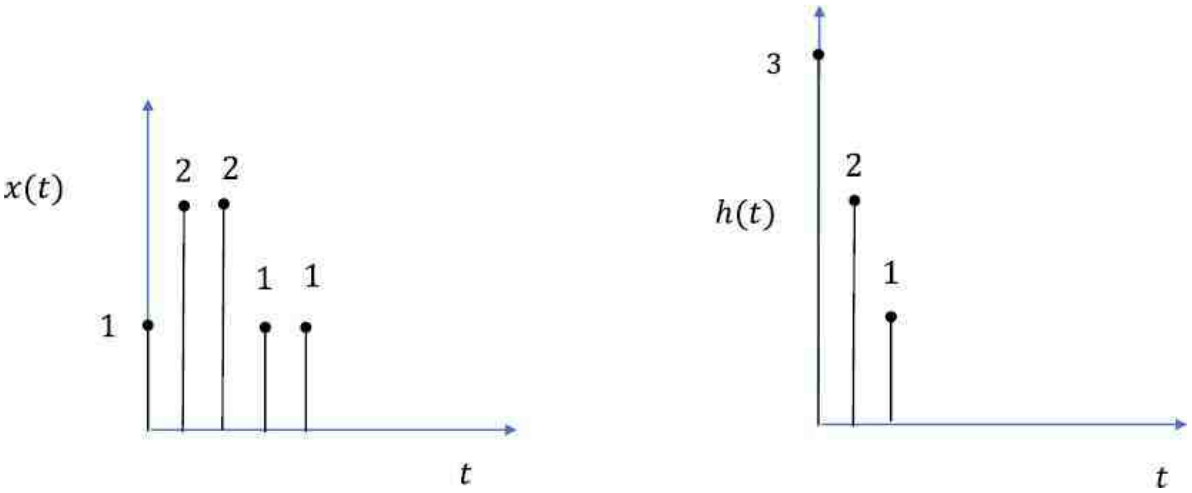


Figure 3-3a: Input signals

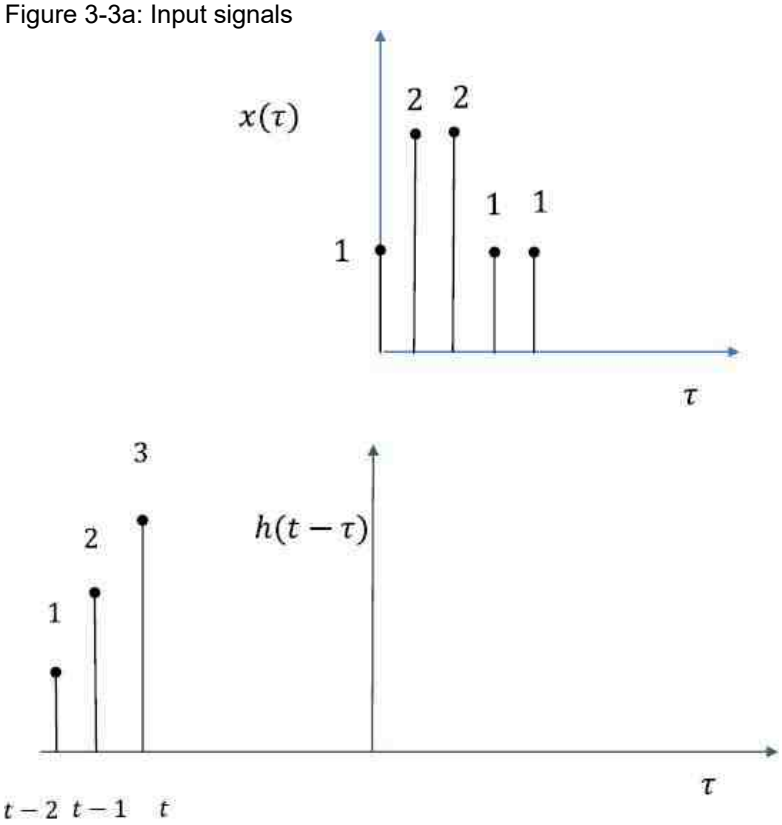


Figure 3-3b: Functions for computing convolution operation

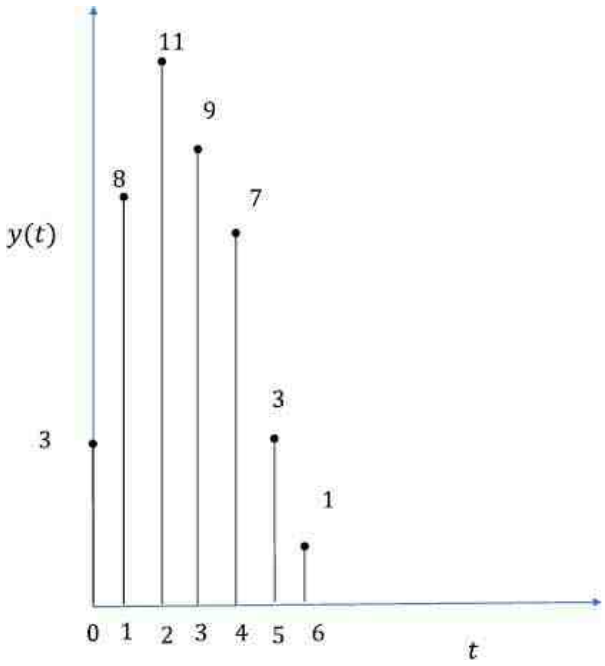


Figure 3-3c: Output function from convolution

In Figure 3-3b, the function $h(t-\tau)$ needs to be computed for different values of t by sliding it across the horizontal axis. At each value of t the convolution sum $\sum_{\tau=-\infty}^{+\infty} x(\tau)h(t-\tau)$ needs to be computed. The sum can be thought of as a weighted average of $x(\tau)$ with the weights being provided by $h(t-\tau)$.

- When $t = -1$ the weights are given by $h(1-\tau)$ but the weights don't overlap with $x(\tau)$ and hence the sum is 0.
- When $t = 0$ the weights are given by $h(-\tau)$ and the only element of $x(\tau)$ in overlap with the weights is $x(\tau = 0)$, the overlapping weight being $h(0)$. Hence, the convolving sum is $x(\tau = 0) * h(0) = 1*3 = 3$. Thus, $y(0)= 3$.
- When $t = 1$ the weights are given by $h(1-\tau)$. The elements x(0) and x(1) are in overlap with the weights h(1) and h(0) respectively. Hence, the convolving sum is $x(0)* h(1)+x(1)*h(0)= 1*2+2*3 = 8$.
- When $t= 2$ the weights are given by $h(2-\tau)$. The elements $x(0)$, $x(1)$, and $x(2)$ are in overlap with the weights $h(2)$, $h(1)$, and $h(0)$ respectively. Hence, the convolving sum is elements $x(0)*h(2)+ x(l)*h(l)+x(2)*h(0) = 1*1 + 2*2 + 2*3 = 11$. The overlap of the two functions for $t = 2$ is illustrated in Figure 3-3d.

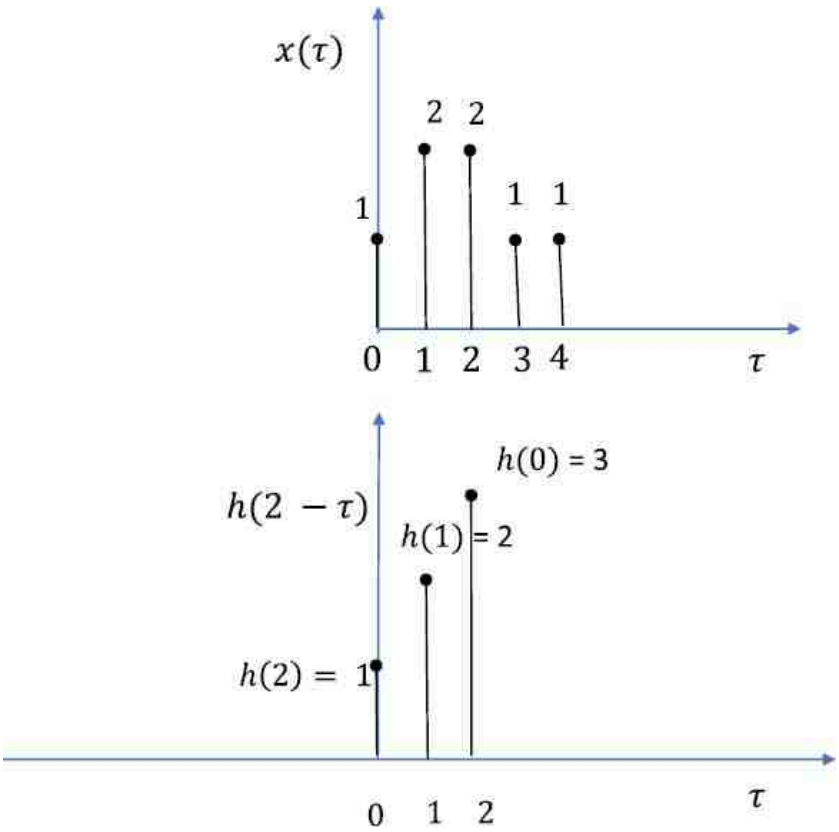


Figure 3-3d: Overlap of the functions in convolution at $t = 2$

Analog and Digital Signals

In general, any quantity of interest that shows variation in time and/or space represents a signal. Hence, a signal is a function of time and/or space. For instance, the stock market prices of a specific stock over a period of a week represent a signal.

Signals can be analogous or digital in nature. However, a computer can't process analogous continuous signals, so the signal is made into a digital signal for processing. For example, speech is an acoustic signal in time where both time and the amplitude of the speech energy are continuous signals. When the speech is transmitted through a microphone, this acoustic continuous signal is converted into an electrical continuous signal. If we want to process the analog electrical signal through a digital computer, we need to convert the analog continuous signal into a discrete signal. This is done through sampling and quantization of the analog signal.

Sampling refers to taking the signal amplitudes only at fixed spatial or time intervals. This has been illustrated in Figure 3-4a.

Not all possible continuous values of the signal amplitude are generally noted, but the signal amplitude is generally quantized to some fixed discrete values, as shown in Figure 3-4b. Through sampling and quantization some information is lost from the analog continuous signal.

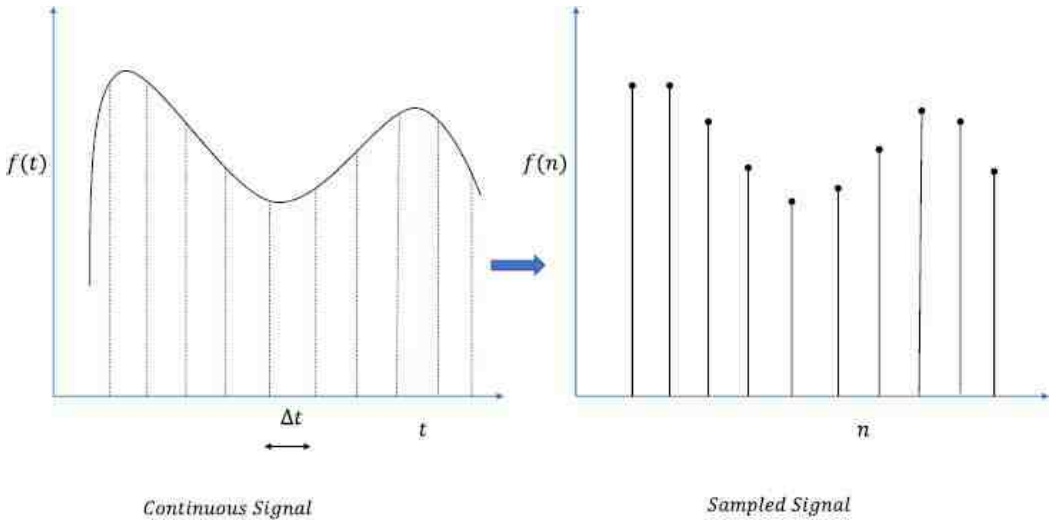


Figure 3-4a: Sampling of a signal

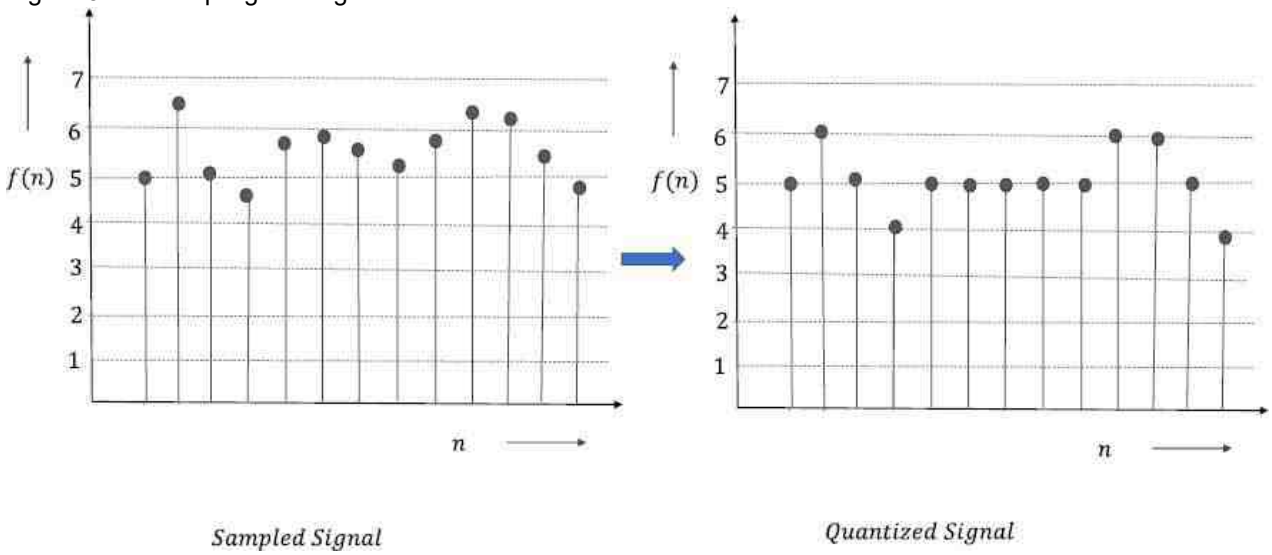


Figure 3-4b: Quantization of signal at discrete amplitude values

The activities of sampling and quantization convert an analog signal to a digital one.

A digital image can be expressed as a digital signal in the two-dimensional spatial domain. The colored RGB image has three channels: Red, Green, and Blue. Each of the channels can be considered a signal in the spatial domain such that at each spatial location the signal is represented by a pixel intensity. Each pixel can be represented by 8 bits, which in binary allows for 256 pixel intensities from 0 to 255. The color at any location is determined by the vector of pixel intensities at that location corresponding to the three channels. So, to represent a specific color, 24 bits of information is used. For a grayscale image, there is only one channel, and the pixel intensities range from 0 to 255. 255 represents the color white, while 0 represents the color black.

A video is a sequence of images with a temporal dimension. A black and white video can be expressed as a signal of its spatial and temporal coordinates (*x*, *y*, *t*). A colored video can be expressed as a combination of three signals, with the spatial and temporal coordinates corresponding to the three color channels—Red, Green, and Blue.

So, a grayscale *n*×*m* image can be expressed as function *I*(*x*,*y*), where *I* denotes the intensity of the pixel at the *x*, *y* coordinate. For a digital image, the *x*, *y* are sampled coordinates and take discrete values. Similarly, the pixel intensity is quantized between 0 and 255.

2D and 3D signals

A grayscale image of dimension *N* × *M* can be expressed as a scalar 2D signal of its spatial coordinates. The signal can be represented as

$$x(n_1,n_2),\ \ 0<n_1<M-1,0<n_2<N-1$$

where n_1 and n_2 are the discrete spatial coordinates along the horizontal and vertical axes respectively and $x(n_1, n_2)$ denotes the pixel intensity at the spatial coordinates. The pixel intensities take up values from 0 to 255.

A colored RGB image is a vector 2D signal since there is a vector of pixel intensities at each spatial coordinate. For an RGB image of dimensions $N \times M \times 3$, the signal can be expressed as

$$x(n_1,n_2)=\left[x_R(n_1,n_2),x_G(n_1,n_2),x_B(n_1,n_2)\right],\ 0<n_1<M-1,0<n_2<N-1$$

where x_R, x_G, and x_B denote the pixel intensities along the Red, Green, and Blue color channels. See Figures 3-5a and 3-5b.

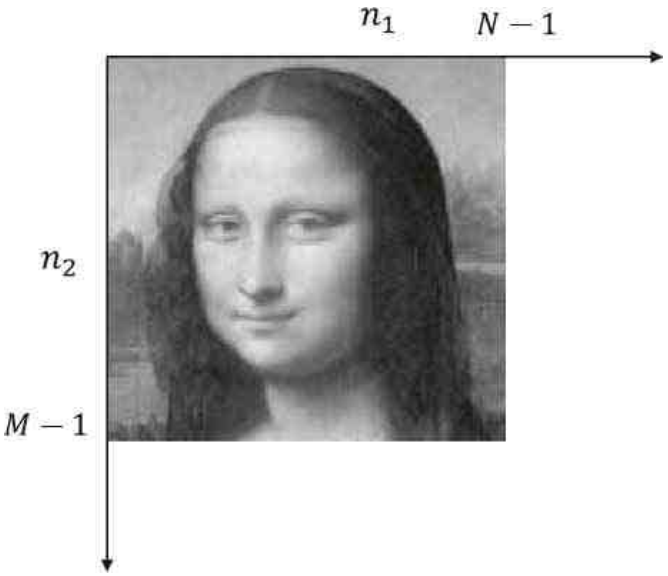


Figure 3-5a: Grayscale image as a 2D discrete signal

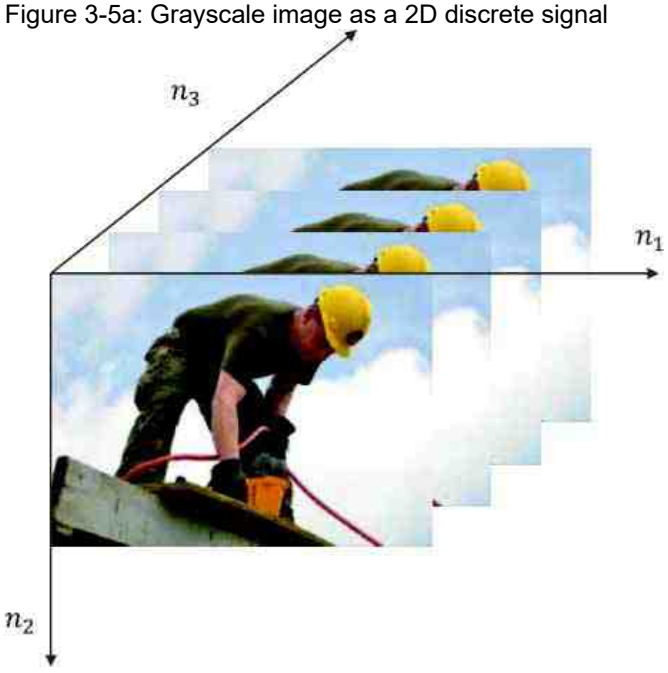


Figure 3-5b: Video as a 3D object

2D Convolution

Now that we have expressed grayscale images as 2D signals, we would like to process those signals through 2D convolution. The images can be convolved with the impulse response of an image-processing system to achieve different objectives, such as the following:

- Remove the visible noise in the image through noise-reduction filters. For white noise, we can use a Gaussian filter. For salt and pepper noise, a median filter can be used.
- For detecting edges, we need filters that extract high-frequency components from an image.

The image-processing filters can be thought of as image-processing systems that are linear and shift invariant. Before we go to image processing, it's worthwhile to know the different impulse functions.

Two-dimensional Unit Step Function

A two-dimensional unit step function $\delta(n_1\ n_2)$, where n_1 and n_2 are the horizontal and vertical coordinates, can be expressed as

$$\delta(n_1,n_2)=1\ when\ n_1=0\ and\ n_2=0$$
$$=0\ elsewhere$$

Similarly, a shifted unit step function can be expressed as

$$\delta(n_1-k_1,n_2-k_2)=1\ when\ n_1=k_1\ and\ n_2=k_2$$
$$=0\ elsewhere$$

This has been illustrated in Figure 3-6.

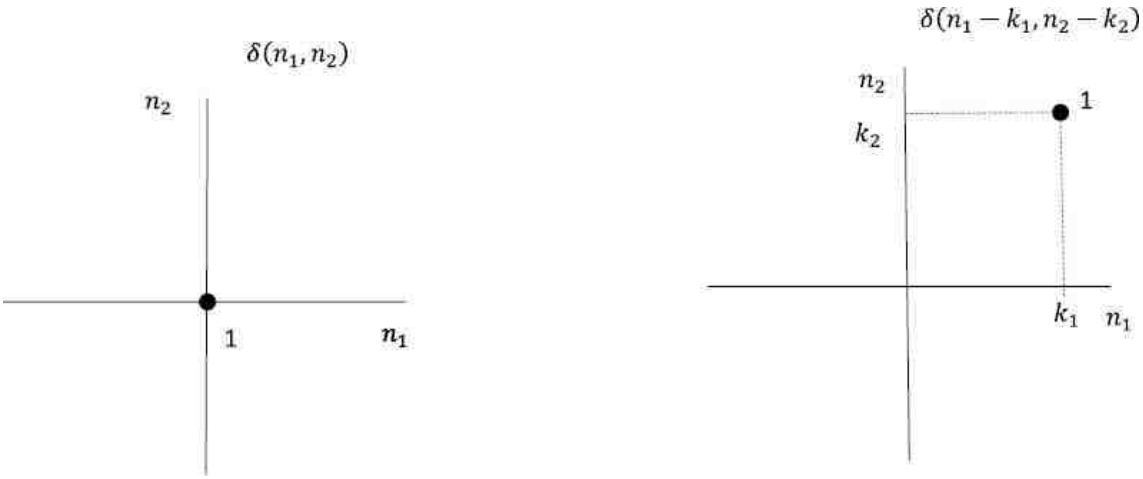
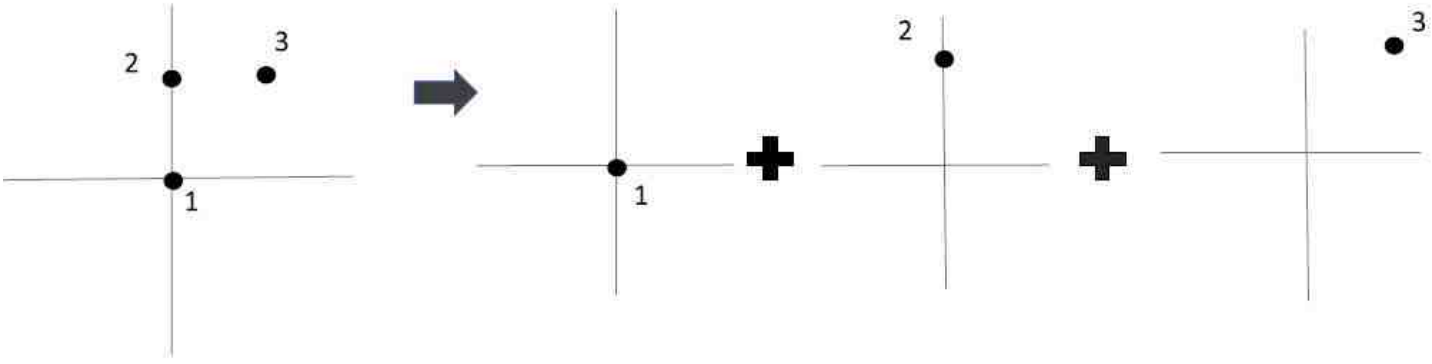


Figure 3-6: Unit step functions

Any discrete two-dimensional signal can be expressed as the weighted sum of unit step functions at different coordinates. Let's consider the signal $x(n_1, n_2)$ as shown in Figure 3-7.

$$x(n_1,n_2)=1\ when\ n_1=0\ and\ n_2=0$$
$$=2\ when\ n_1=0\ and\ n_2=1$$
$$=3\ when\ n_1=1\ and\ n_2=1$$
$$=0\ elsewhere$$
$$x(n_1,n_2)=x(0,0)*\delta(n_1,n_2)+x(0,1)*\delta(n_1,n_2-1)+x(1,1)*\delta(n_1-1,n_2-1)$$
$$=1*\delta(n_1,n_2)+2*\delta(n_1,n_2-1)+3*\delta(n_1-1,n_2-1)$$

$x(n_1,n_2)$ $\quad x(0,0)*\delta(n_1,n_2)$ $\quad x(0,1)*\delta(n_1,n_2-1)$ $\quad x(1,1)*\delta(n_1-1,n_2-1)$



$x(0,0)=1$ $\quad x(0,1)=2$ $\quad x(1,1)=3$

Figure 3-7: Representing a 2D discrete signal as the weighted sum of unit step functions

So, in general, any discrete 2D signal can be written as follows:

$$x(n_1,n_2)=\sum_{k_2=-\infty}^{+\infty}\sum_{k_1=-\infty}^{+\infty}x(k_1,k_2)\delta(n_1-k_1,n_2-k_2)$$

2D Convolution of a Signal with an LSI System Unit Step Response

When any discrete 2D signal as expressed above signal passes through an LSI system with transformation *f*, then, because of the linearity property of LSI systems,

$$f(x(n_1,n_2))=\sum_{k_2=-\infty}^{+\infty}\sum_{k_1=-\infty}^{+\infty}x(k_1,k_2)f(\delta(n_1-k_1,n_2-k_2))$$

Now, the unit step response of an LSI system $f(\delta(n_1, n_2) = h(n_1, n_2)$, and since an LSI system is shift invariant, $f(\delta(n_1-k_1,n_2-k_2)=h(n_1-k_1,n_2-k_2)$.

Hence, $f(x(n_1, n_2))$ can be expressed as follows:

(1) $$f(x(n_1,n_2))=\sum_{k_2=-\infty}^{+\infty}\sum_{k_1=-\infty}^{+\infty}x(k_1,k_2)h(n_1-k_1,n_2-k_2)$$

The preceding expression denotes the expression for 2D convolution of a signal with the unit step response of an LSI system. To illustrate 2D convolution, let's walk through an example in which we convolve $x(n_1, n_2)$ with $h(n_1, n_2)$. The signal and the unit step response signal are defined as follows, and have also been illustrated in Figure 3-8:

$$\begin{aligned}
x(n_1,n_2) &= 4 \quad when\ n_1=0,n_2=0\\
&= 5 \quad when\ n_1=1,n_2=0\\
&= 2 \quad when\ n_1=0,n_2=1\\
&= 3 \quad when\ n_1=1,n_2=1\\
&= 0 \quad elsewhere\\
h(n_1,n_2) &= 1 \quad when\ n_1=0,n_2=0\\
&= 2 \quad when\ n_1=1,n_2=0\\
&= 3 \quad when\ n_1=0,n_2=1\\
&= 4 \quad when\ n_1=1,n_2=1\\
&= 0 \quad elsewhere
\end{aligned}$$

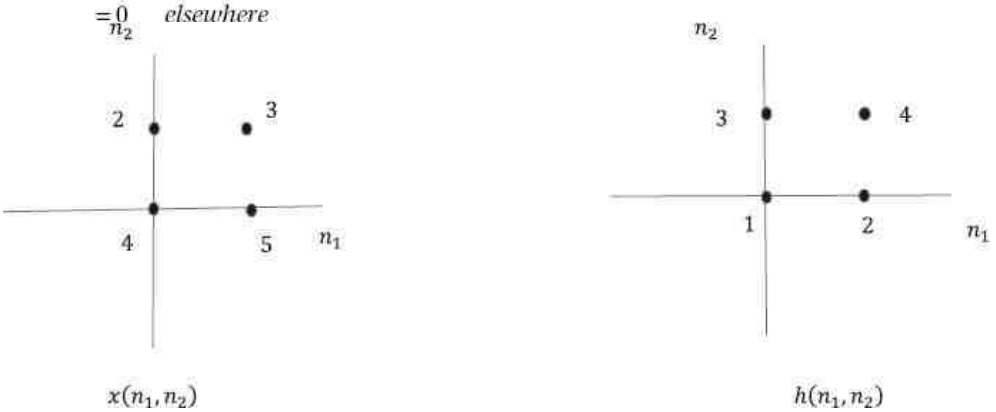


Figure 3-8: 2D signal and unit step response of LSI system

To compute the convolution, we need to plot the signals on a different set of coordinate points. We chose k_1 and k_2 on the horizontal and vertical axes respectively. Also, we reverse the impulse response $h(k_1, k_2)$ to $h(-k_1-k_2)$ as plotted in Figure 3-9(b). We then place the reversed function $h(-k_1-k_2)$ at different offset values for n_1 and n_2. The generalized reversed function can be expressed as $h(n_1-k_1,n_2-k_2)$. For computing the output $y(n_1, n_2)$ of convolution at a specific value of n_1 and n_2, we see the points at which $h(n_1-k_1,n_2-k_2)$ overlaps with $x(k_1, k_2)$ and take the total sum of the coordinate-wise product of the signal and impulse response values as the output.

As we can see in Figure 3-9(c), for the $(n_1 = 0,n_2 = 0)$ offset the only point of overlap is $(k_1 = 0,k_2 = 0)$ and so $y(0,0) = x(0,0)$*

$h(0,0) = 4*1 = 4$.

Similarly, for offset ($n_1 = 1, n_2 = 0$), the points of overlap are the points ($k_1 = 0, k_2 =$ and ($k_1 = 1, k_2 = 0$) as shown in Figure 3-9(d).

$$y(1,0) = x(0,0)*h(1-0,0-0) + x(1,0)*h(1-1,0-0)$$
$$= x(0,0)*h(1,0) + x(1,0)*h(0,0)$$
$$= 4*2 + 5*1 = 13$$

For offset ($n_l = l, n_2 = l$),the points of overlap are the points ($k_1 = 1, k_2 = 0$),as shown in Figure 3-9(e).

$$y(2,0) = x(1,0)*h(2-1,0-0)$$
$$= x(1,0)*h(1,0)$$
$$= 5*2 = 10$$

Following this approach of shifting the unit step response signal by altering n_1 and n_2, the entire function $y(n_1, n_2)$ can be computed.

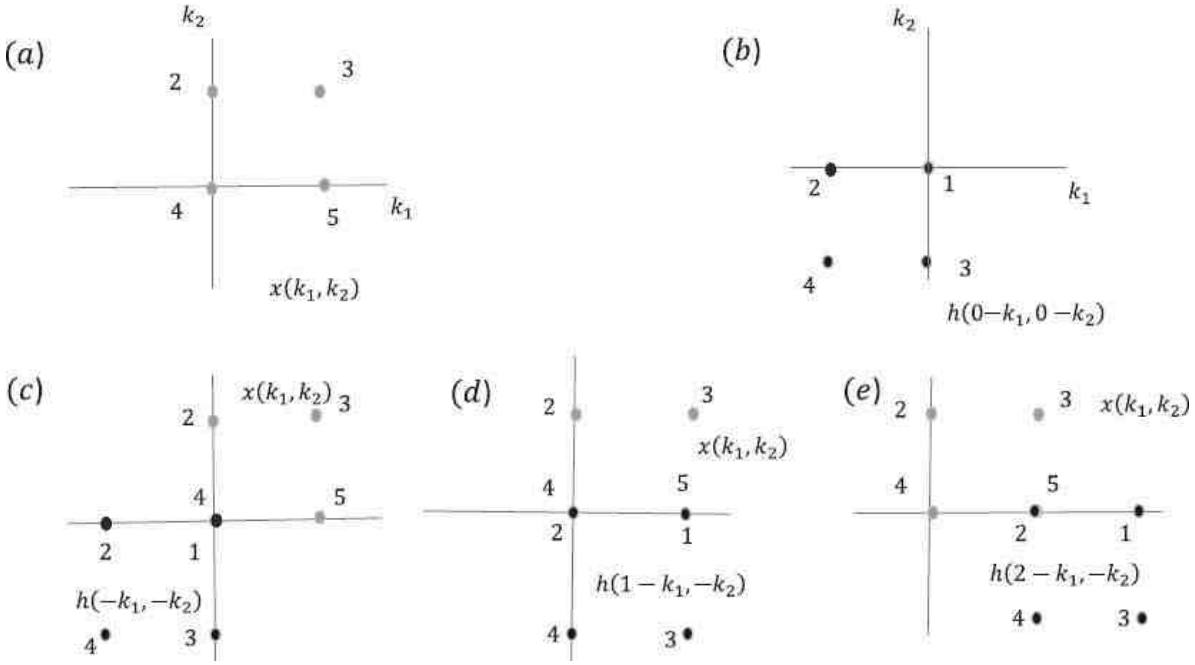


Figure 3-9: Convolution at different coordinate points

2D Convolution of an Image to Different LSI System Responses

Any image can be convolved with an LSI system's unit step response. Those LSI system unit step responses are called filters or kernels. For example, when we try to take an image through a camera and the image gets blurred because of shaking of hands, the blur introduced can be treated as an LSI system with a specific unit step response. This unit step response convolves the actual image and produces the blurred image as output. Any image that we take through the camera gets convolved with the unit step response of the camera. So, the camera can be treated as an LSI system with a specific unit step response.

Any digital image is a 2D discrete signal. The convolution of an $N \times M$ 2D image $x(n_1, n_2)$ with a 2D image-processing filter $h(n_1, n_2)$ is given by

$$y(n_1, n_2) = \sum_{k_2=0}^{N-1} \sum_{k_1=0}^{M-1} x(k_1, k_2) h(n_1 - k_1, n_2 - k_2)$$
$$\text{where } 0 \le n_1 \le N-1, 0 \le n_2 \le M-1.$$

The image-processing filters work on a grayscale image's (2D) signal to produce another image (2D signal). In cases of multi-channel images, generally 2D image-processing filters are used for image processing, which means one must process each image channel as a 2D signal or convert the image into a grayscale image.

Now that we have gone through the concepts of convolution, we know to term any unit step response of an LSI system with

which we convolve an image as a filter or kernel.

An example of 2D convolution is illustrated in Figure 3-10a.

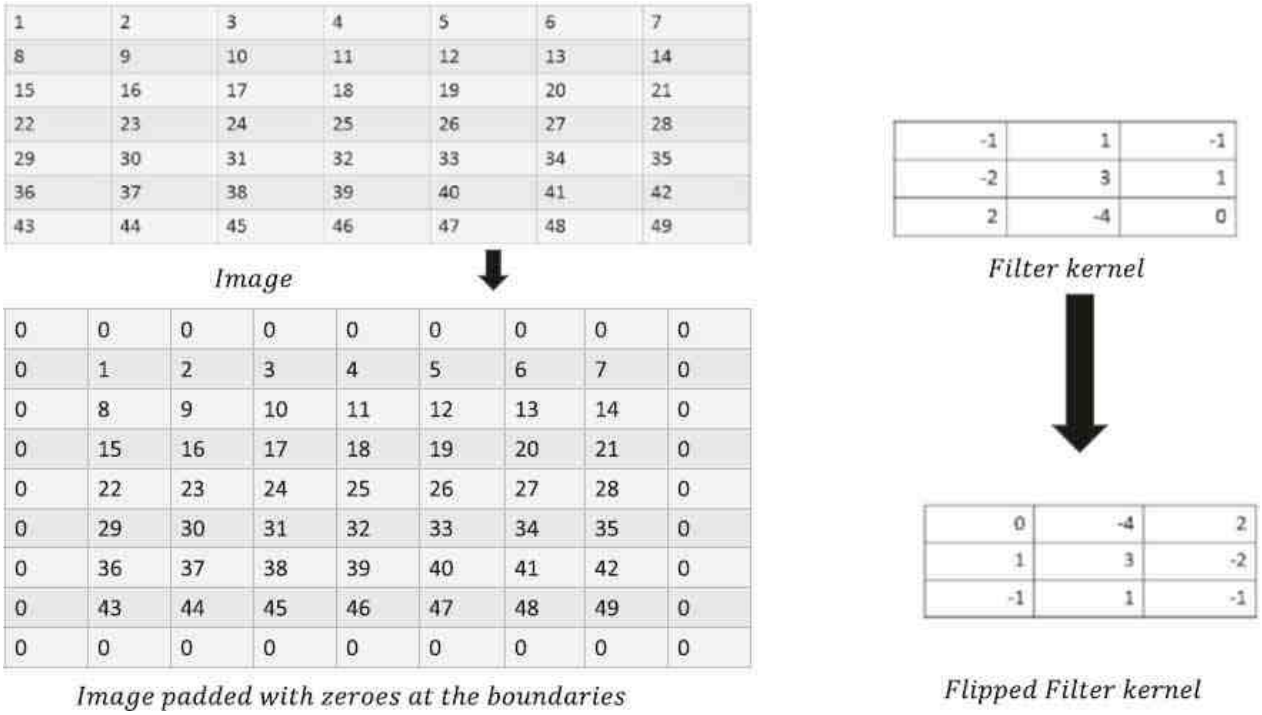


Figure 3-10a: Example of 2D convolution of images

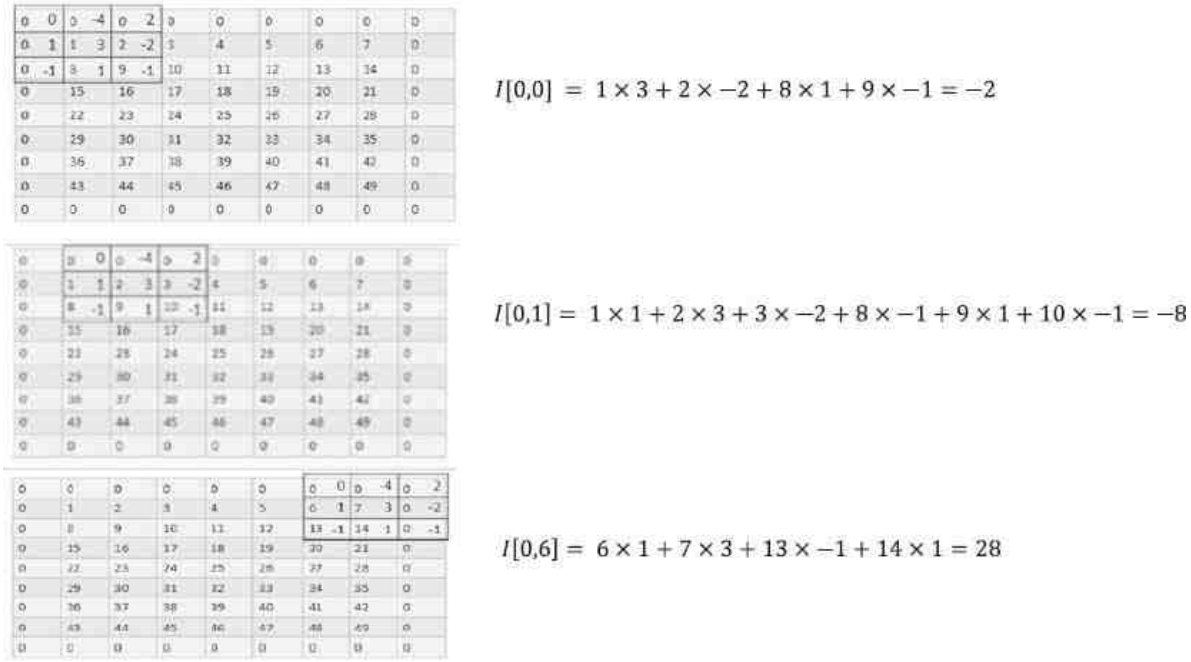


Figure 3-10b

To keep the length of the output image the same as that of the input image, the original image has been zero padded. As we can see, the flipped filter or kernel is slid over various areas of the original image, and the convolution sum is computed at each coordinate point. Please note that the indexes in the intensity $I[i, j]$ as mentioned in Figure 3-10b denote the matrix coordinates. The same example problem is worked out through scipy 2D convolution as well as through basic logic in Listing 3-1. In both cases the results are the same.

Listing 3-1

```
## Illustate 2D convolution of images through an example

import scipy.signal
```

```
import numpy as np
# Take a 7x7 image as example
image = np.array([[1, 2, 3, 4, 5, 6, 7],
         [8, 9, 10, 11, 12, 13, 14],
         [15, 16, 17, 18, 19, 20, 21],
         [22, 23, 24, 25, 26, 27, 28],
         [29, 30, 31, 32, 33, 34, 35],
         [36, 37, 38, 39, 40, 41, 42],
         [43, 44, 45, 46, 47, 48, 49]])

# Defined an image-processing kernel
filter_kernel = np.array([[-1, 1, -1],
                 [-2, 3, 1],
                 [2, -4, 0]])
# Convolve the image with the filter kernel through scipy 2D convolution to produce an
output image of same dimension as that of the input

I = scipy.signal.convolve2d(image, filter_kernel,mode='same', boundary='fill', fillvalue=0)
print(I)

# We replicate the logic of a scipy 2D convolution by going through the following steps
# a) The boundaries need to be extended in both directions for the image and padded with
zeroes.
#  For convolving the 7x7 image by 3x3 kernel, the dimensions need to be extended by
(3-1)/2-i.e., 1-
#on either side for each dimension. So a skeleton image of 9x9 image would be created
# in which the boundaries of 1 pixel are pre-filled with zero.
# b) The kernel needs to be flipped-i.e., rotated-by 180 degrees
# c) The flipped kernel needs to placed at each coordinate location for the image and then
the sum of
#coordinate-wise product with the image intensities needs to be computed. These sums for
each coordinate would give
#the intensities for the output image.

row,col=7,7
## Rotate the filter kernel twice by 90 degrees to get 180 rotation
filter_kernel_flipped = np.rot90(filter_kernel,2)
## Pad the boundaries of the image with zeroes and fill the rest from the original image
image1 = np.zeros((9,9))
for i in xrange(row):
    for j in xrange(col):
        image1[i+1,j+1] = image[i,j]
print(image1)

## Define the output image
image_out = np.zeros((row,col))
## Dynamic shifting of the flipped filter at each image coordinate and then computing the
convolved sum.
for i in xrange(1,1+row):
    for j in xrange(1,1+col):
        arr_chunk = np.zeros((3,3))

        for k,k1 in zip(xrange(i-1,i+2),xrange(3)):
            for l,l1 in zip(xrange(j-1,j+2),xrange(3)):
                arr_chunk[k1,l1] = image1[k,l]

        image_out[i-1,j-1] = np.sum(np.multiply(arr_chunk,filter_kernel_flipped))

print(image_out)
```

```
[[ -2  -8  -7  -6  -5  -4  28]
 [  5  -3  -4  -5  -6  -7  28]
 [ -2 -10 -11 -12 -13 -14  28]
 [ -9 -17 -18 -19 -20 -21  28]
 [-16 -24 -25 -26 -27 -28  28]
 [-23 -31 -32 -33 -34 -35  28]
 [-29  13  13  13  13  13  27]]
```

Scipy convolve2d output

```
[[ -2.  -8.  -7.  -6.  -5.  -4.  28.]
 [  5.  -3.  -4.  -5.  -6.  -7.  28.]
 [ -2. -10. -11. -12. -13. -14.  28.]
 [ -9. -17. -18. -19. -20. -21.  28.]
 [-16. -24. -25. -26. -27. -28.  28.]
 [-23. -31. -32. -33. -34. -35.  28.]
 [-29.  13.  13.  13.  13.  13.  27.]]
```

2D convolution implementation

Figure 3-11

Based on the choice of image-processing filter, the nature of the output images will vary. For example, a Gaussian filter would create an output image that would be a blurred version of the input image, whereas a Sobel filter would detect the edges in an image and produce an output image that contains the edges of the input image.

Common Image-Processing Filters

Let's discuss image-processing filters commonly used on 2D images. Make sure to be clear with notations since the natural way of indexing an image doesn't align well with how one would prefer to define the *x* and *y* axes. Whenever we represent an image-processing filter or an image in the coordinate space, n_1 and n_2 are the discrete coordinates for the *x* and *y* directions. The column index of the image in numpy matrix form coincides nicely with the *x* axis, whereas the row index moves in the opposite direction of the *y* axis. Also, it doesn't matter which pixel location one chooses as the origin for the image signal while doing convolution. Based on whether zero padding is used or not, one can handle the edges accordingly. Since the filter kernel is of a smaller size, we generally flip the filter kernel and then slide it over the image and not the other way around.

Mean Filter

The Mean filter or Average filter is a low-pass filter that computes the local average of the pixel intensities at any specific point. The impulse response of the Mean filter can be any of the form seen here (see Figure 3-12):

$$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

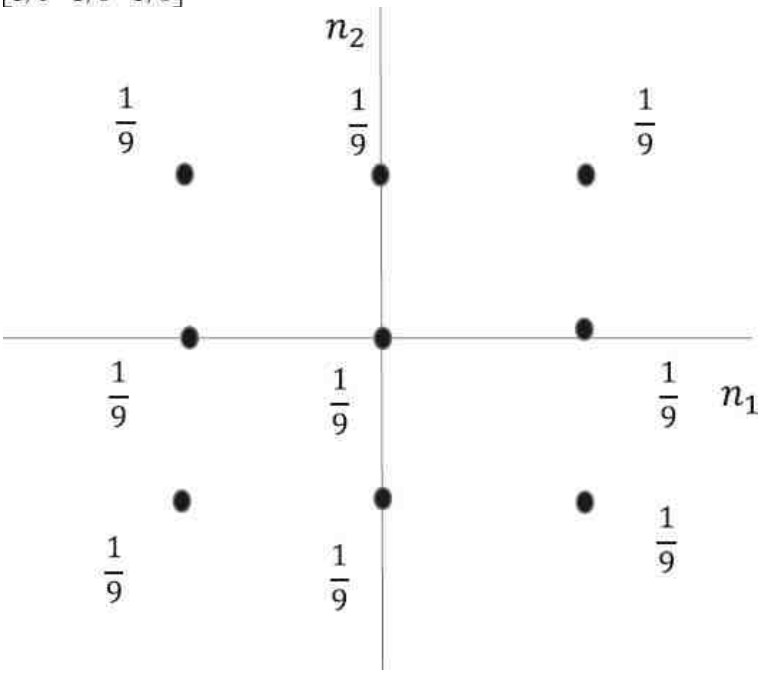


Figure 3-12: Impulse response of a Mean filter

Here, the matrix entry h_{22} corresponds to the entry at the origin. So, at any given point, the convolution will represent the average of the pixel intensities at that point. The code in Listing 3-2 illustrates how one can convolve an image with an image-processing filter such as the Mean filter.

Please note that in many Python implementations we would be using OpenCV to perform basic operations on the image, such as reading the image, converting the image from RGB format to grayscale format, and so on. OpenCV is an open source image-processing package that has a rich set of methodologies for image processing. Readers are advised to explore OpenCV or any other image-processing toolbox in order to get accustomed to the basic image-processing functions.

Listing 3-2: Convolution of an Image with Mean Filter

```
import cv2
img = cv2.imread('monalisa.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
plt.imshow(gray,cmap='gray')
mean = 0
var = 100
sigma = var**0.5
row,col = 650,442
gauss = np.random.normal(mean,sigma,(row,col))
gauss = gauss.reshape(row,col)
```

```
gray_noisy = gray + gauss
plt.imshow(gray_noisy,cmap='gray')
## Mean filter
Hm = np.array([[1,1,1],[1,1,1],[1,1,1]])/float(9)
Gm = convolve2d(gray_noisy,Hm,mode='same')
plt.imshow(Gm,cmap='gray')
```

In Listing 3-2, we read an image of the *Mona Lisa* and then introduce some Gaussian white noise into the image. The Gaussian noise has a mean of 0 and a variance of 100. We then convolve the noisy image with a Mean filter to reduce the white noise. The noisy image and the image after convolution has been plotted are shown in Figure 3-13.

Noisy Image with Gaussian Noise

Image after Convolving with Mean Filter

Figure 3-13: Mean filter processing on Mona Lisa image

The Mean filter is mostly used to reduce the noise in an image. If there is some white Gaussian noise present in the image, then the Mean filter will reduce the noise since it averages over its neighborhood, and hence the white noise of the zero mean will be suppressed. As we can see from Figure 3-13, the Gaussian white noise is reduced once the image has been convolved with the Mean filter. The new image has fewer high-frequency components and thus is relatively less sharp than the image before convolution, but the filter has done a good job of reducing the white noise.

Median Filter

A 2D Median filter replaces each pixel in a neighborhood with the median pixel intensity in that neighborhood based on the filter size. The Median filter is good for removing salt and pepper noise. This type of noise presents itself in the images in the form of black and white pixels and is generally caused by sudden disturbances while capturing the images. Listing 3-3 illustrates how salt and pepper noise can be added to an image and then how the noise can be suppressed using a Median filter.

Listing 3-3

```
## Generate random integers from 0 to 20
## If the value is zero we will replace the image pixel with a low value of 0 that
corresponds to a black pixel
## If the value is 20 we will replace the image pixel with a high value of 255 that
corresponds to a white pixel
## We have taken 20 integers, out of which we will only tag integers 1 and 20 as salt and
pepper noise
## Hence, approximately 10% of the overall pixels are salt and pepper noise. If we want to
reduce it
## to 5% we can take integers from 0 to 40 and then treat 0 as an indicator for a black
pixel and 40 as an indicator for a white pixel.

np.random.seed(0)
gray_sp = gray*1
sp_indices = np.random.randint(0,21,[row,col])
for i in xrange(row):
    for j in xrange(col):
        if sp_indices[i,j] == 0:
            gray_sp[i,j] = 0
        if sp_indices[i,j] == 20:
            gray_sp[i,j] = 255
```

```
plt.imshow(gray_sp,cmap='gray')

## Now we want to remove the salt and pepper noise through a Median filter.
## Using the opencv Median filter for the same

gray_sp_removed = cv2.medianBlur(gray_sp,3)
plt.imshow(gray_sp_removed,cmap='gray')

##Implementation of the 3x3 Median filter without using opencv

gray_sp_removed_exp = gray*1
for i in xrange(row):
    for j in xrange(col):
        local_arr = []
        for k in xrange(np.max([0,i-1]),np.min([i+2,row])):
            for l in xrange(np.max([0,j-1]),np.min([j+2,col])):
                local_arr.append(gray_sp[k,l])
        gray_sp_removed_exp[i,j] = np.median(local_arr)
plt.imshow(gray_sp_removed_exp,cmap='gray')
```

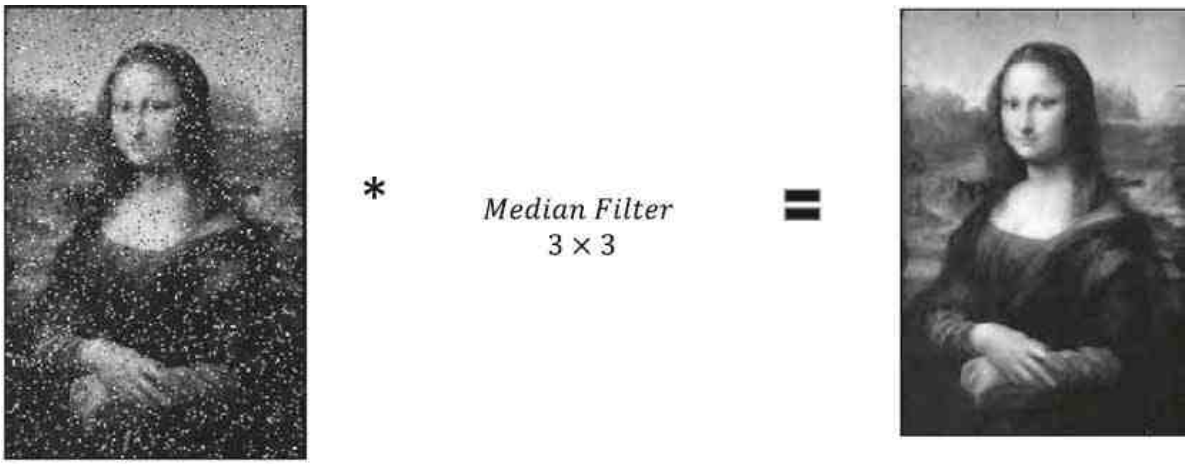


Figure 3-14: Median filter processing

As we can see, the salt and pepper noise has been removed by the Median filter.

Gaussian Filter

The Gaussian filter is a modified version of the Mean filter where the weights of the impulse function are distributed normally around the origin. Weight is highest at the center of the filter and falls normally away from the center. A Gaussian filter can be created with the code in Listing 3-4. As we can see, the intensity falls in a Gaussian fashion away from the origin. The Gaussian filter, when displayed as an image, has the highest intensity at the origin and then diminishes for pixels away from the center. Gaussian filters are used to reduce noise by suppressing the high-frequency components. However, in its pursuit of suppressing the high-frequency components it ends up producing a blurred image, called Gaussian blur.

In Figure 3-15, the original image is convolved with the Gaussian filter to produce an image that has Gaussian blur. We then subtract the blurred image from the original image to get the high-frequency component of the image. A small portion of the high-frequency image is added to the original image to improve the sharpness of the image.

Listing 3-4

```
Hg = np.zeros((20,20))
for i in xrange(20):
    for j in xrange(20):
        Hg[i,j] = np.exp(-((i-10)**2 + (j-10)**2)/10)
plt.imshow(Hg,cmap='gray')
gray_blur = convolve2d(gray,Hg,mode='same')
plt.imshow(gray_blur,cmap='gray')
gray_enhanced = gray + 0.025*gray_high
plt.imshow(gray_enhanced,cmap='gray')
```

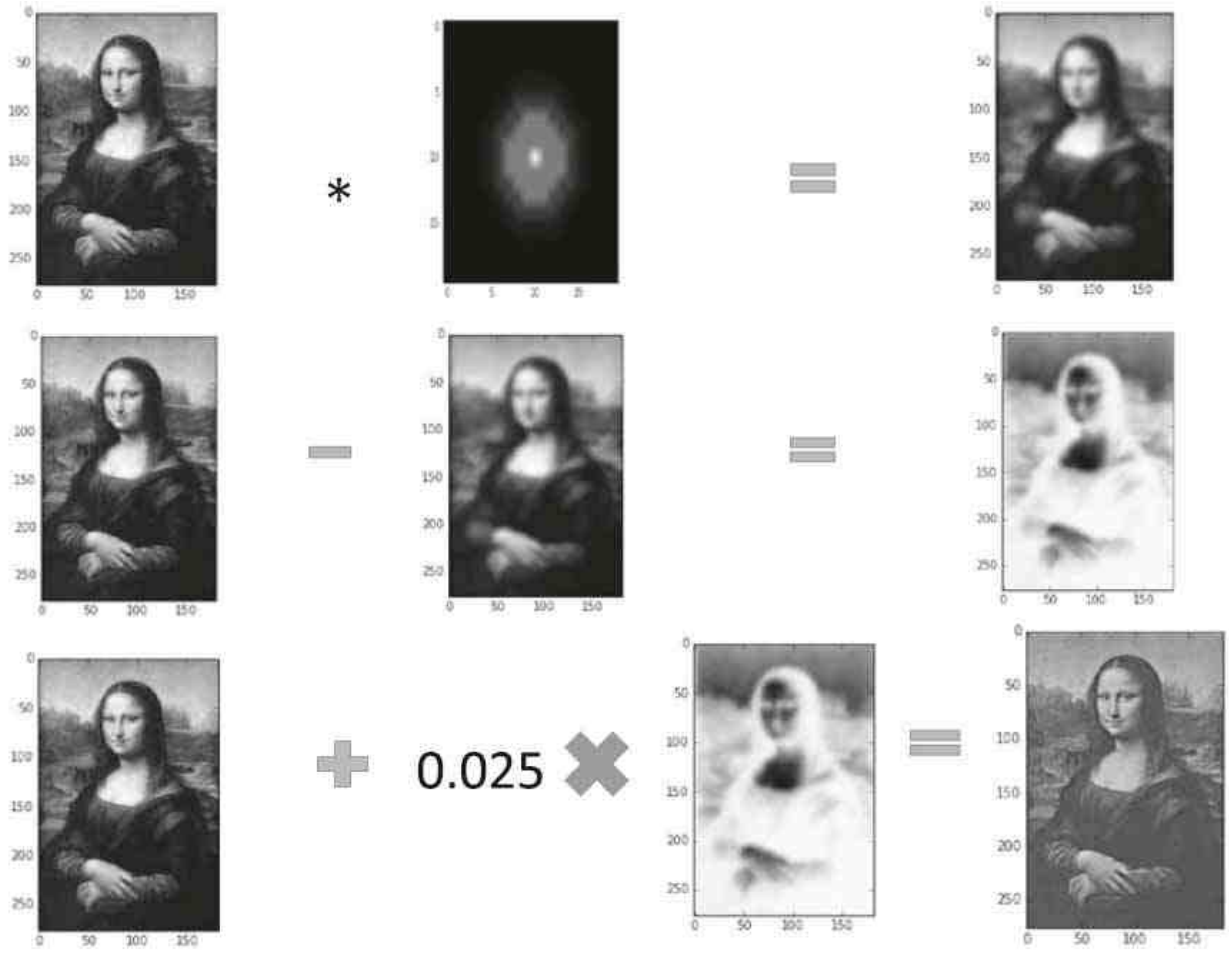


Figure 3-15: Various activities with Gaussian filter kernel

Gradient-based Filters

To review, the gradient of a two-dimensional function $I(x,y)$ is given by the following:

$$\nabla I(x,y)=\left[\frac{\partial I(x,y)}{\partial x}\frac{\partial I(x,y)}{\partial y}\right]^T$$

where the gradient along the horizontal direction is given by - $\frac{\partial I(x,y)}{\partial x}=\lim_{h\to 0}\frac{I(x+h,y)-I(x,y)}{h}$ or $\lim_{h\to 0}\frac{I(x+h,y)-I(x-h,y)}{2h}$ based on convenience and the problem at hand.

For discrete coordinates, we can take $h = 1$ and approximate the gradient along the horizontal as follows:

$$\frac{\partial I(x,y)}{\partial x}=I(x+1,y)-I(x,y)$$

This derivative of a signal can be achieved by convolving the signal with the filter kernel $\begin{bmatrix}0 & 0 & 0\\ 0 & 1 & -1\\ 0 & 0 & 0\end{bmatrix}$.

Similarly,

$$\frac{\partial I(x,y)}{\partial x}\propto I(x+1,y)-I(x-1,y)$$

from the second representation.

This form of derivative can be achieved by convolving the signal with the filter kernel $\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix}$.

For the vertical direction, the gradient component in the discrete case can be expressed as

$$\frac{\partial I(x,y)}{\partial y} = I(x,y+1) - I(x,y) \text{ or by } \frac{\partial I(x,y)}{\partial y} \propto I(x,y+1) - I(x,y-1)$$

The corresponding filter kernels to compute gradients through convolution are $\begin{bmatrix} 0 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ and $\begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$ respectively.

Do note that these filters take the direction of the *x* axis and *y* axis, as shown in Figure 3-16. The direction of *x* agrees with the matrix index n_2 increment, whereas the direction of *y* is opposite to that of the matrix index n_1 increment.

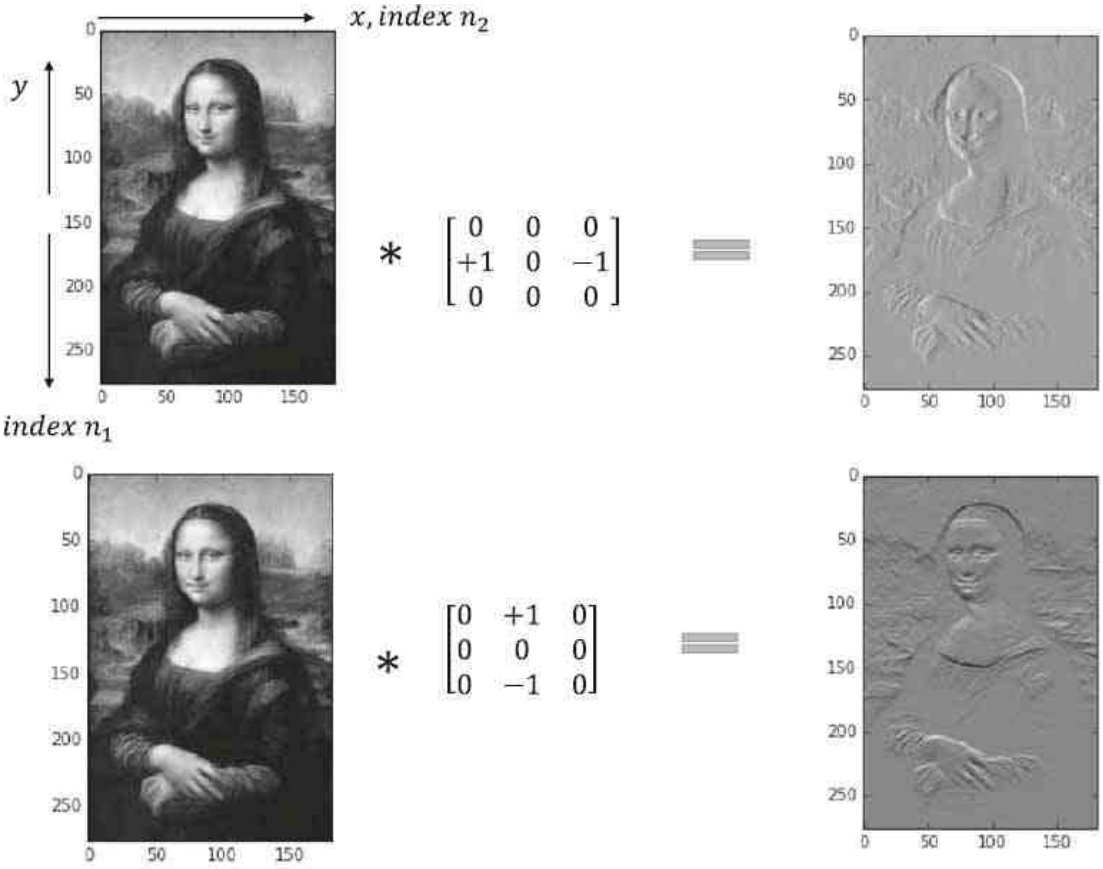


Figure 3-16: Vertical and horizontal gradient filters

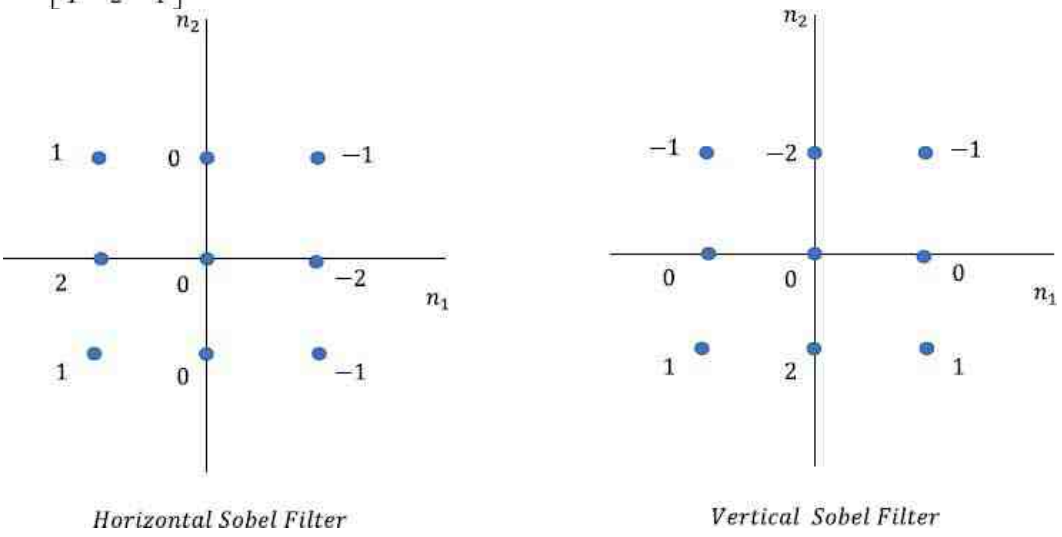
Figure 3-16 illustrates the convolution of the *Mona Lisa* image with the Horizontal and Vertical Gradient filters.

Sobel Edge-Detection Filter

The impulse response of a Sobel Edge Detector along the horizontal and vertical axes can be expressed by the following H_x and H_y matrices respectively. The Sobel Detectors are extensions of the Horizontal and Vertical Gradient filters just illustrated. Instead of only taking the gradient at the point, it also takes the sum of the gradients at the points on either side of it. Also, it gives double weight to the point of interest. See Figure 3-17.

$$Hx = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$$

$$Hy = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$



Horizontal Sobel Filter *Vertical Sobel Filter*

Figure 3-17: Sobel filter impulse response

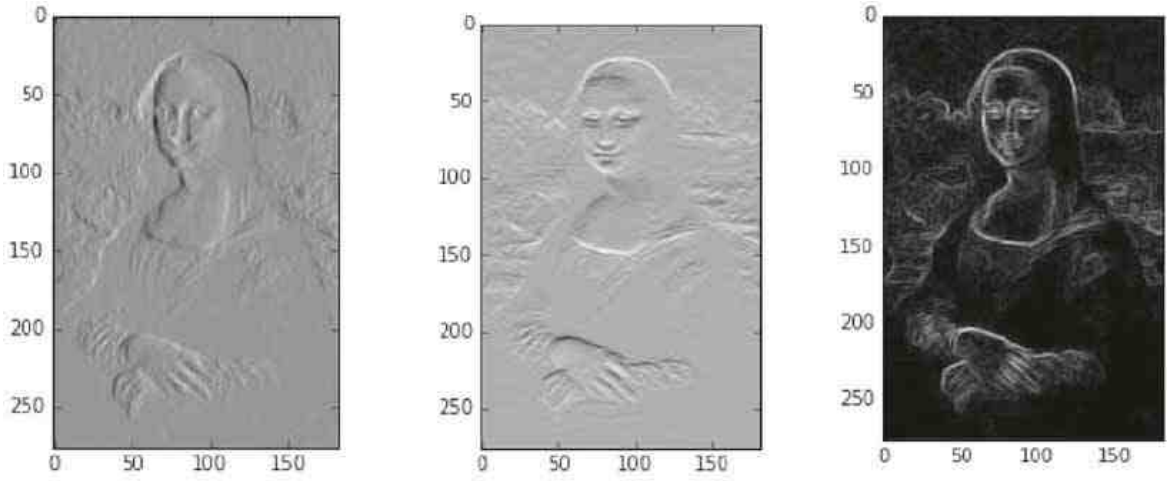
The convolution of the image with the Sobel filters is illustrated in Listing 3-5.

Listing 3-5: Convolution Using a Sobel Filter

```
Hx = np.array([[ 1,0, -1],[2,0,-2],[1,0,-1]],dtype=np.float32)
Gx = convolve2d(gray,Hx,mode='same')
plt.imshow(Gx,cmap='gray')

Hy = np.array([[ -1,-2, -1],[0,0,0],[1,2,1]],dtype=np.float32)
Gy = convolve2d(gray,Hy,mode='same')
plt.imshow(Gy,cmap='gray')

G = (Gx*Gx + Gy*Gy)**0.5
plt.imshow(G,cmap='gray')
```



Output of Convolution with Horizontal Sobel Filter *Output of Convolution with Vertical Sobel Filter* *Output from Combined Sobel Filter*

Figure 3-18: Output of various Sobel filters

Listing 3-5 has the logic required to convolve the image with the Sobel filters. The Horizontal Sobel filter detects edges in the horizontal direction, whereas the Vertical Sobel filter detects edges in the vertical direction. Both are high-pass filters since they

attenuate the low frequencies from the signals and capture only the high-frequency components within the image. Edges are important features for an image and help one detect local changes within an image. Edges are generally present on the boundary between two regions in an image and are often the first step in retrieving information from images. We see the output of Listing 3-5 in Figure 3-18. The pixel values obtained for the images with Horizontal and Vertical Sobel filters for each location can be thought of as a vector $I'(x,y)=\left[I_x(x,y)I_y(x,y)\right]^T$, where $I_x(x,y)$ denotes the pixel intensity of the image obtained through the Horizontal Sobel filter and $I_y(x,y)$ denotes the pixel intensity of the image obtained through the Vertical Sobel filter. The magnitude of the vector $I'(x,y)$ can be used as the pixel intensity of the combined Sobel filter.

$C(x,y)=\sqrt{\left(I_x(x,y)\right)^2+\left(I_y(x,y)\right)^2}$,where $C(x, y)$ denotes the pixel intensity function for the combined Sobel filter.

Identity Transform

The filter for Identity Transform through convolution is as follows:

$$\begin{bmatrix}0 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 0\end{bmatrix}$$

Figure 3-19 illustrates a Unity Transform through convolution.

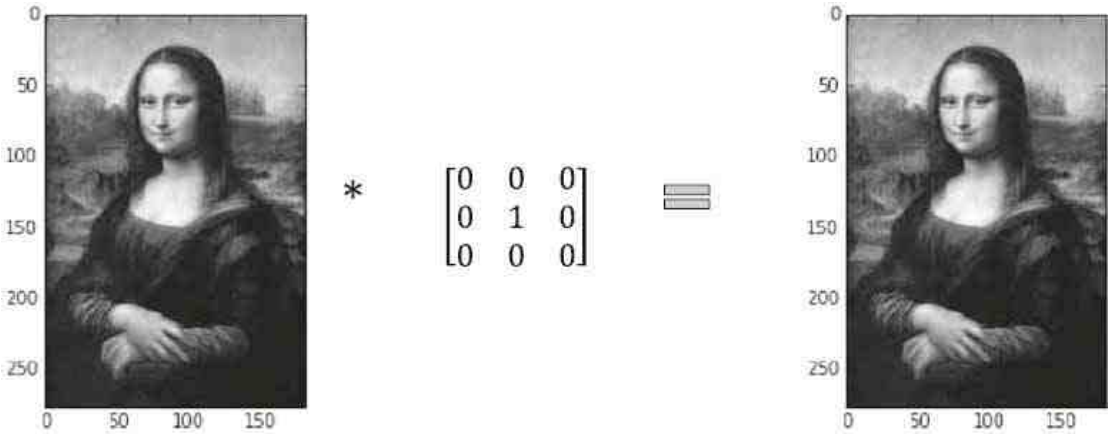


Figure 3-19: Identity transform through convolution

Table 3-1 lists several useful image-processing filters and their uses.

Table 3-1: Image-Processing Filters and Their Uses

Filter	Use
Mean Filter	Reduce Gaussian noise, smooth the image after upsampling
Median Filter	Reduce salt and pepper noise
Sobel Filter	Detect edges in an image
Gaussian Filter	Reduce noise in an image
Canny Filter	Detect edges in an image
Weiner Filter	Reduce additive noise and blurring

Convolution Neural Networks

Convolution neural networks (CNNs) are based on the convolution of images and detect features based on filters that are learned by the CNN through training. For example, we don't apply any known filter, such as the ones for the detection of edges or for removing the Gaussian noise, but through the training of the convolutional neural network the algorithm learns image-processing filters on its own that might be very different from normal image-processing filters. For supervised training, the filters are learned in such a way that the overall cost function is reduced as much as possible. Generally, the first convolution layer learns to detect edges, while the second may learn to detect more complex shapes that can be formed by combining different edges, such as circles and rectangles, and so on. The third layer and beyond learn much more complicated features based on the features generated in the previous layer.

The good thing about convolutional neural networks is the sparse connectivity that results from weight sharing, which greatly

reduces the number of parameters to learn. The same filter can learn to detect the same edge in any given portion of the image through its equivariance property, which is a great property of convolution useful for feature detection.

Components of Convolution Neural Networks

The following are the typical components of a convolutional neural network:

- ***Input layer*** will hold the pixel intensity of the image. For example, an input image with width 64, height 64, and depth 3 for the Red, Green, and Blue color channels (RGB) would have input dimensions of 64×64×3.
- ***Convolution layer*** will take images from the preceding layers and convolve with them the specified number of filters to create images called *output feature maps*. The number of output feature maps is equal to the specified number of filters. Till now, CNNs in TensorFlow have used mostly 2D filters; however, recently 3D convolution filters have been introduced.
- ***Activation function*** for CNNs are generally ReLUs, which we discussed in Chapter 2. The output dimension is the same as the input after passing through the ReLU activation layers. The ReLU layer adds non-linearity in the network and at the same time provides non-saturating gradients for positive net inputs.
- ***Pooling layer*** will downsample the 2D activation maps along the height and width dimensions. The depth or the number of activation maps is not compromised and remains the same.
- ***Fully connected layers*** contain traditional neurons that receive different sets of weights from the preceding layers; there is no weight sharing between them as is typical for convolution operations. Each neuron in this layer will be connected either to all the neurons in the previous layer or to all the coordinate-wise outputs in the output maps through separate weights. For classification, the class output neurons receive inputs from the final fully connected layers.

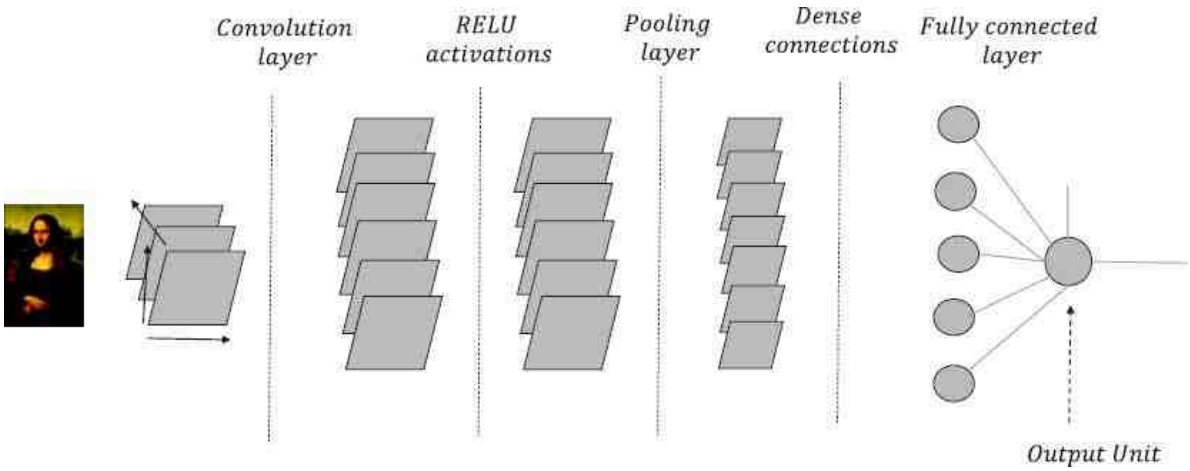


Figure 3-20: Basic flow diagram of a convolutional neural network

Figure 3-20 illustrates a basic convolutional neural network (CNN) that uses one convolutional layer, one ReLU layer, and one pooling layer followed by a fully connected layer and finally the output classification layer. The network tries to discern the *Mona Lisa* images from the *non-Mona Lisa* images. The output unit can be taken to have a sigmoid activation function since it's a binary classification problem for images. Generally, for most of the CNN architectures a few to several convolutional layer-ReLU layer-pooling layer combinations are stacked one after another before the fully connected layers. We will discuss the different architectures at a later point in time. For now, let's look at the different layers in much more detail.

Input Layer

The input to this layer are images. Generally, the images are fed in batches as four-dimensional tensors where the first dimension is specific to the image index, second and third dimensions are specific to the height and width of the image, and the fourth dimension corresponds to the different channels. For a colored image, generally we have the Red (R), Green (G), and Blue (B) channels, while for grayscale images we have only one channel. The number of images in a batch would be determined by the mini-batch size chosen for the mini-batch stochastic gradient descent. The batch size is one for stochastic gradient descent.

The inputs can be fed to the input layer in mini batches through TensorFlow placeholder `tf.placeholder at` runtime.

Convolution Layer

Convolution is the heart of any CNN network. TensorFlow supports both 2D and 3D convolutions. However, 2D convolutions are more common since 3D convolutions are computationally memory intensive. The input images or intermediate images in the form of output feature maps are 2D convolved with 2D filters of the size specified. 2D convolution happens along the spatial dimensions, while there is no convolution along the depth channel of the image volume. For each depth channel, the same number of feature maps are generated, and then they are summed together along the depth dimension before they pass through the ReLU activations. These filters help to detect features in the images. The deeper the convolutional layer is in the network, the more complicated features it learns. For instance, the initial convolutional layer might learn to detect edges in an image, while the second convolutional layer might learn to connect the edges to form geometric shapes such as circles and rectangles. The even deeper convolutional layers might learn to detect more complicated features; for example, in Cat versus Dog classification it might learn to detect eyes, nose, or other body parts of the animals.

In a CNN, only the size of the filters is specified; the weights are initialized to arbitrary values before the start of training. The weights of the filters are learned through the CNN training process and hence they might not represent the traditional image-processing filters such as Sobel, Gaussian, Mean, Median, or other kind of filters. Instead the learned filters would be such that the overall loss function defined is minimized or a good generalization is achieved based on the validation. Although it might not learn the traditional edge-detection filter, it would learn several filters that detect edges in some form since edges are good feature detectors for images.

Some of the terms with which one should be familiar while defining the convolution layer are as follows:

- ***Filter size*** – Filter size defines the height and width of the filter kernel. A filter kernel of size 3×3 would have nine weights. Generally, these filters are initialized and slid over the input image for convolution without flipping these filters. Technically, when convolution is performed without flipping the filter kernel it's called cross-correlation and not convolution. However, it doesn't matter, as we can consider the filters learned as a flipped version of image-processing filters.

- ***Stride*** – The stride determines the number of pixels to move in each spatial direction while performing convolution. In normal convolution of signals, we generally don't skip any pixels and instead compute the convolution sum at each pixel location, and hence we have a stride of 1 along both spatial directions for 2D signals. However, one may choose to skip every alternate pixel location while convolving and thus chose a stride of 2. If a stride of 2 is chosen along both the height and the width of the image, then after convolving the output image would be approximately ¼ of the input image size. Why it is *approximately* — and not ¼ *exactly* ¼ of the original image or feature-map size will be covered in our next topic of discussion.

- ***Padding*** – When we convolve an image of a specific size by a filter, the resulting image is generally smaller than the original image. For example, if we convolve a 5×52D image by a filter of size 3×3, the resulting image is 3×3.

Padding is an approach that appends zeroes to the boundary of an image to control the size of the output of convolution. The convolved output image length *L′* along a specific spatial dimension is given by

$$L' = \frac{L - K + 2P}{S} + 1$$

where

- *L* → Length of the input image in a specific dimension
- *K* → Length of the kernel/filter in a specific dimension
- *P* → Zeroes padded along a dimension in either end
- *S* → Stride of the convolution

In general, for a stride of 1 the image size along each dimension is reduced by (*K* - I)/2 on either end, where *K* is the length of the filter kernel along that dimension. So, to keep the output image the same as that of the input image, a pad length of $\frac{K-1}{2}$ would be required.

Whether a specific stride size is possible can be found out from the output image length along a specific direction. For example, if *L* = 12, *K* = 3, and *P* = 0, stride *S* = 2 is not possible, since it would produce an output length along the spatial dimension as $\frac{(12-3)}{2} = 4.5$, which is not an integer value.

In TensorFlow, padding can be chosen as either `"VALID"` or `"SAME"`. `"SAME"` ensures that the output spatial dimensions of the

image are the same as those of the input spatial dimensions in cases where a stride of 1 is chosen. It uses zero padding to achieve this. It tries to keep the zero-pad length even on both sides of a dimension, but if the total pad length for that dimension is odd then the extra length is added to the right for the horizontal dimension and to the bottom for the vertical dimension.

`"VALID"` doesn't use zero padding and hence the output image dimension would be smaller than the input image dimensions, even for a stride of 1.

TensorFlow Usage

```
def conv2d(x,W,b,strides=1):
    x = tf.nn.conv2d(x,W,strides=[1,strides,strides,1],padding='SAME')
    x = tf.nn.bias_add(x,b)
    return tf.nn.relu(x)
```

For defining a TensorFlow Convolutional layer we use `tf.nn.conv2d` where we need to define the input to the Convolution, weights associated with the Convolution, the stride size and the padding type. Also, we add a bias for each output feature map. Finally, we use the Rectified Linear Units ReLUs as activations to add non-linearity into the system.

Pooling Layer

A pooling operation on an image generally summarizes a locality of an image, the locality being given by the size of the filter kernel, also called the receptive field. The summarization generally happens in the form of max pooling or average pooling. In max pooling, the maximum pixel intensity of a locality is taken as the representative of that locality. In average pooling, the average of the pixel intensities around a locality is taken as the representative of that locality. Pooling reduces the spatial dimensions of an image. The kernel size that determines the locality is generally chosen as 2 × 2 whereas the stride is chosen as 2. This reduces the image size to about ¼ the size of the original image.

TensorFlow Usage

```
''' P O O L I N G  L A Y E R'''
def maxpool2d(x,stride=2):
    return tf.nn.max_pool(x,ksize=[1,stride,stride,1],strides=[1,stride,stride,1],
    padding='SAME')
```

The `tf.nn.max_pool` definition is used to define a max pooling layer while `tf.nn.avg_pool` is used to define an average pooling layer. Apart from the input, we need to input the receptive field or kernel size of max pooling through the `ksize` parameter. Also, we need to provide the strides to be used for the max pooling. To ensure that the values in each spatial location of the output feature map from pooling are from independent neighborhoods in the input, the stride in each spatial dimension should be chosen to be equal to the kernel size in the corresponding spatial dimension.

Backpropagation Through the Convolutional Layer

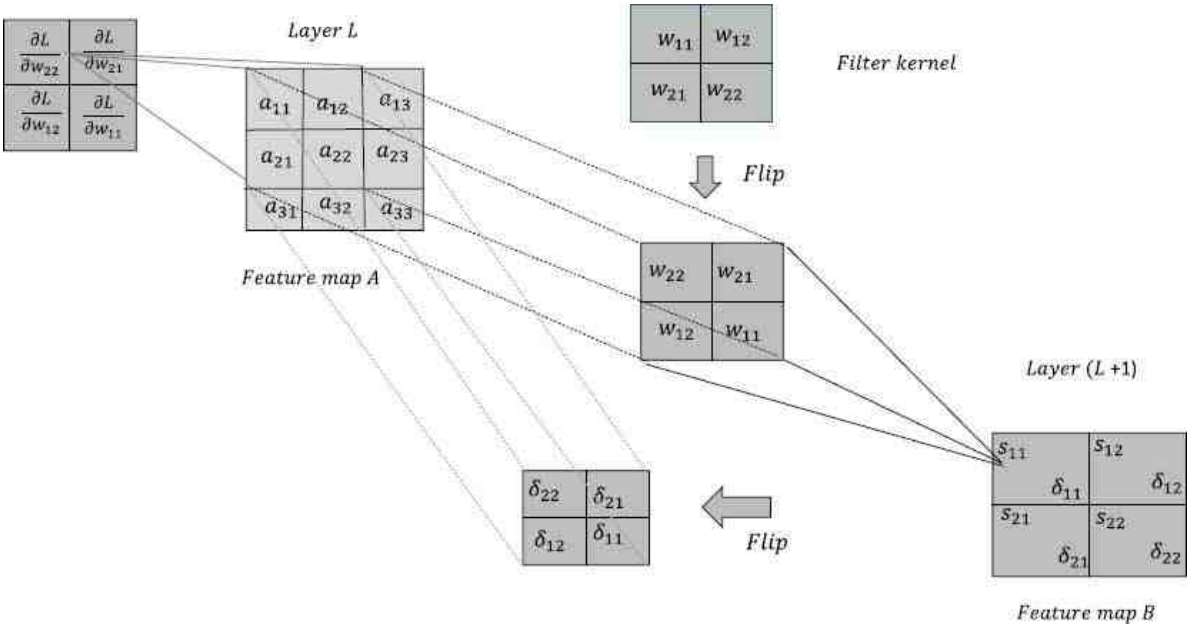


Figure 3-21: Backpropagation through the convolutional layer

Backpropagation through a convolution layer is much like backpropagation for a multi-layer Perceptron network. The only difference is that the weight connections are sparse since the same weights are shared by different input neighborhoods to create an output feature map. Each output feature map is the result of the convolution of an image or a feature map from the previous layer with a filter kernel whose values are the weights that we need to learn through backpropagation. The weights in the filter kernel are shared for a specific input-output feature-map combination.

In Figure 3-21, feature map *A* in layer *L* convolves with a filter kernel to produce an output feature map *B* in layer (*L* +1).

The values of the output feature map are the results of convolution and can be expressed as $s_{ij}\ \forall i,j \in \{1,2\}$:

$$s_{11} = w_{22} * a_{11} + w_{21} * a_{12} + w_{12} * a_{21} + w_{11} * a_{22}$$
$$s_{12} = w_{22} * a_{12} + w_{21} * a_{13} + w_{12} * a_{22} + w_{11} * a_{23}$$
$$s_{21} = w_{22} * a_{21} + w_{21} * a_{22} + w_{12} * a_{31} + w_{11} * a_{32}$$
$$s_{22} = w_{22} * a_{22} + w_{23} * a_{22} + w_{12} * a_{32} + w_{11} * a_{33}$$

In generalized way:

$$s_{ij} = \sum_{n=1}^{2}\sum_{m=1}^{2} w_{(3-m)(3-n)} * a_{(i-1+m)(j-1+n)}$$

Now, let the gradient of the cost function *L* with respect to the net input s_{ij} be denoted by

$$\frac{\partial L}{\partial s_{ij}} = \delta_{ij}$$

Let's compute the gradient of the cost function with respect to the weight w_{22}. The weight is associated with all s_{ij} and hence would have gradient components from all the δ_{ij}:

$$\frac{\partial \mathrm{L}}{\partial w_{22}} = \sum_{j=1}^{2}\sum_{i=1}^{2} \frac{\partial \mathrm{L}}{\partial s_{ij}} \frac{\partial s_{ij}}{\partial w_{22}}$$

$$= \sum_{j=1}^{2}\sum_{i=1}^{2} \delta_{ij} \frac{\partial s_{ij}}{\partial w_{22}}$$

Also, from the preceding equations for different s_{ij}, the following can be derived:

$$\frac{\partial s_{11}}{\partial w_{22}} = a_{11},\ \frac{\partial s_{12}}{\partial w_{22}} = a_{12},\ \frac{\partial s_{13}}{\partial w_{22}} = a_{21},\ \frac{\partial s_{14}}{\partial w_{22}} = a_{22}$$

Hence,

$$\frac{\partial \mathrm{L}}{\partial w_{22}} = \delta_{11} * a_{11} + \delta_{12} * a_{12} + \delta_{21} * a_{21} + \delta_{22} * a_{22}$$

Similarly,

$$\frac{\partial \mathrm{L}}{\partial w_{21}} = \sum_{j=1}^{2}\sum_{i=1}^{2} \frac{\partial \mathrm{L}}{\partial s_{ij}} \frac{\partial s_{ij}}{\partial w_{21}}$$

$$= \sum_{j=1}^{2}\sum_{i=1}^{2} \delta_{ij} \frac{\partial s_{ij}}{\partial w_{21}}$$

Again, $\frac{\partial s_{11}}{\partial w_{21}} = a_{12},\ \frac{\partial s_{12}}{\partial w_{21}} = a_{13},\ \frac{\partial s_{21}}{\partial w_{21}} = a_{22},\ \frac{\partial s_{22}}{\partial w_{21}} = a_{23}$

Hence,

$$\frac{\partial \mathrm{L}}{\partial w_{21}} = \delta_{11} * a_{12} + \delta_{12} * a_{13} + \delta_{21} * a_{22} + \delta_{22} * a_{23}$$

Proceeding by the same approach for the other two weights, we get

$$\frac{\partial \mathrm{L}}{\partial w_{11}} = \sum_{j=1}^{2}\sum_{i=1}^{2} \frac{\partial \mathrm{L}}{\partial s_{ij}} \frac{\partial s_{ij}}{\partial w_{11}}$$

$$= \sum_{j=1}^{2}\sum_{i=1}^{2} \delta_{ij} \frac{\partial s_{ij}}{\partial w_{11}}$$

$$\frac{\partial s_{11}}{\partial w_{11}} = a_{22}, \frac{\partial s_{12}}{\partial w_{11}} = a_{23}, \frac{\partial s_{21}}{\partial w_{11}} = a_{32}, \frac{\partial s_{22}}{\partial w_{21}} = a_{33}$$

$$\frac{\partial \mathrm{L}}{\partial w_{11}} = \delta_{11} * a_{22} + \delta_{12} * a_{23} + \delta_{21} * a_{32} + \delta_{22} * a_{33}$$

$$\frac{\partial \mathrm{L}}{\partial w_{12}} = \sum_{j=1}^{2}\sum_{i=1}^{2} \frac{\partial \mathrm{L}}{\partial s_{ij}} \frac{\partial s_{ij}}{\partial w_{12}}$$

$$\frac{\partial s_{11}}{\partial w_{12}} = a_{21}, \frac{\partial s_{12}}{\partial w_{12}} = a_{22}, \frac{\partial s_{21}}{\partial w_{12}} = a_{31}, \frac{\partial s_{22}}{\partial w_{22}} = a_{32}$$

$$\frac{\partial \mathrm{L}}{\partial w_{12}} = \delta_{11} * a_{21} + \delta_{12} * a_{22} + \delta_{21} * a_{31} + \delta_{22} * a_{32}$$

Based on the preceding gradients of the cost function *L* with respect to the four weights of the filter kernel, we get the following relationship:

$$\frac{\partial \mathrm{L}}{\partial w_{ij}} = \sum_{n=1}^{2}\sum_{m=1}^{2} \delta_{mn} * a_{(i-1+m)(j-1+n)}$$

When arranged in matrix form, we get the following relationship; (x) denotes the cross-correlation:

$$\begin{bmatrix} \frac{\partial \mathrm{L}}{\partial w_{22}} & \frac{\partial \mathrm{L}}{\partial w_{21}} \\ \frac{\partial \mathrm{L}}{\partial w_{12}} & \frac{\partial \mathrm{L}}{\partial w_{11}} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} (\mathrm{x}) \begin{bmatrix} \delta_{11} & \delta_{12} \\ \delta_{21} & \delta_{22} \end{bmatrix}$$

The cross correlation of $\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$ with $\begin{bmatrix} \delta_{11} & \delta_{12} \\ \delta_{21} & \delta_{22} \end{bmatrix}$ can also be thought of as the convolution of $\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$ with flipped $\begin{bmatrix} \delta_{11} & \delta_{12} \\ \delta_{21} & \delta_{22} \end{bmatrix}$; i.e., $\begin{bmatrix} \delta_{22} & \delta_{21} \\ \delta_{12} & \delta_{11} \end{bmatrix}$.

Hence, the flip of the gradient matrix is the convolution of $\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$ with $\begin{bmatrix} \delta_{22} & \delta_{21} \\ \delta_{12} & \delta_{11} \end{bmatrix}$; i.e.,

$$\begin{bmatrix} \frac{\partial \mathrm{L}}{\partial w_{22}} & \frac{\partial \mathrm{L}}{\partial w_{21}} \\ \frac{\partial \mathrm{L}}{\partial w_{12}} & \frac{\partial \mathrm{L}}{\partial w_{11}} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} (*) \begin{bmatrix} \delta_{22} & \delta_{12} \\ \delta_{21} & \delta_{11} \end{bmatrix}$$

In terms of the layers, one can say the flip of the gradient matrix turns out to be a cross-correlation of the gradient at the (*L*+1)

layer with the outputs of the feature map at layer *L*. Also, equivalently, the flip of the gradient matrix turns out to be a convolution of the flip of the gradient matrix at the (*L*+1) layer with the outputs of the feature map at layer *L*.

Backpropagation Through the Pooling Layers

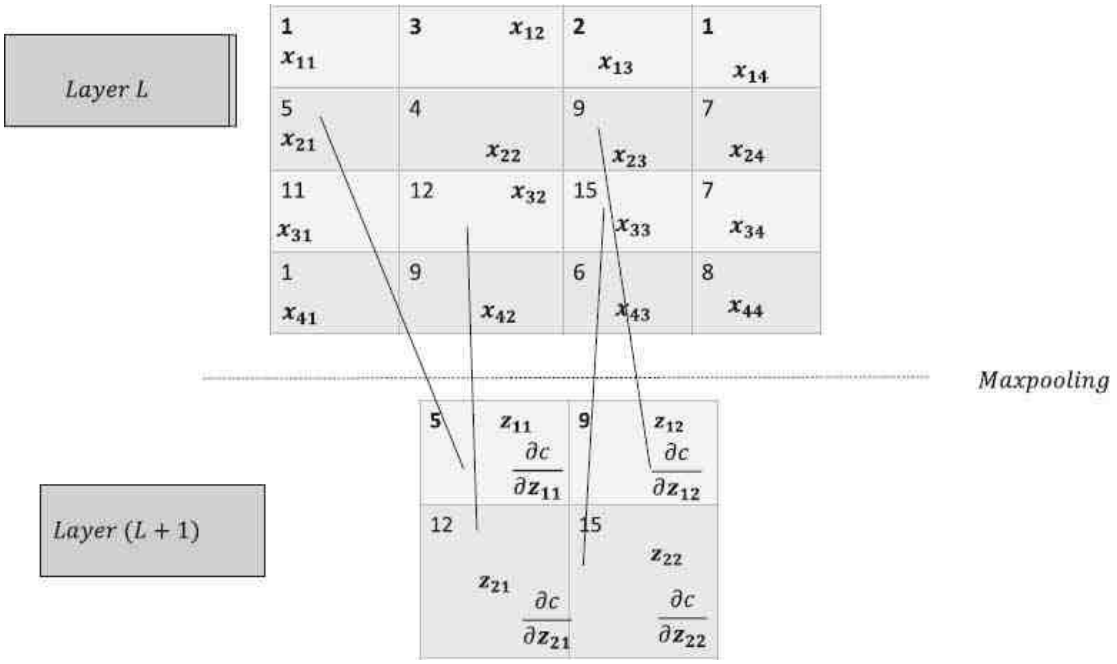


Figure 3-22: Backpropagation through max pooling layer

Figure 3-22 illustrates the max pooling operation. Let a feature map, after going through convolution and ReLU activations at layer *L*, go through the max pooling operation at layer (*L*+1) to produce the output feature map. The kernel or receptive field for max pooling is of size 2×2, and the stride size is 2. The output of the max pooling layer is ¼ the size of the input feature map, and its output values are represented by $z_{ij}, \forall i,j \in \{1,2\}$.

We can see z_{11} gets the value of 5 since the maximum in the 2×2 block is 5. If the error derivative at z_{11} is $\frac{\partial C}{\partial z_{ij}}$, then the whole gradient is passed onto x_{21} with a value of 5, and the rest of the elements in its block—x_{11}, x_{12}, and x_{22}—receive zero gradients from z_{11}.

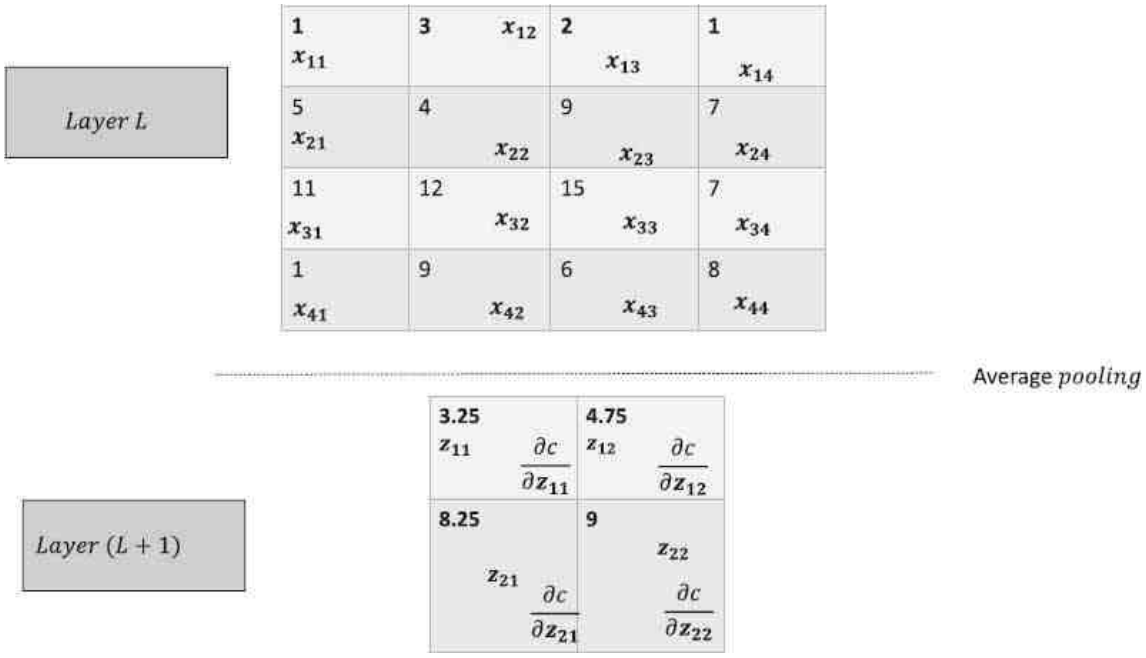


Figure 3-23: Backpropagation through average pooling layer

To use average pooling for the same example, the output is the average of the values in the 2×2 block of the input. Therefore, z_{11} gets the average of the values x_{11}, x_{12}, x_{21}, and x_{22}. Here, the error gradient $\frac{\partial C}{\partial z_{11}}$ at z_{11} would be shared equally by x_{11}, x_{12}, x_{21}, and x_{22}. Hence,

$$\frac{\partial C}{\partial x_{11}}=\frac{\partial C}{\partial x_{12}}=\frac{\partial C}{\partial x_{21}}=\frac{\partial C}{\partial x_{22}}=\frac{1}{4}\frac{\partial C}{\partial z_{11}}$$

Weight Sharing Through Convolution and Its Advantages

Weight sharing through convolution greatly reduces the number of parameters in the convolutional neural network. Imagine we created a feature map of size $k \times k$ from an image of $n \times n$ size with full connections instead of convolutions. There would be k^2n^2 weights for that one feature map alone, which is a lot of weights to learn. Instead, since in convolution the same weights are shared across locations defined by the filter-kernel size, the number of parameters to learn is reduced by a huge factor. In cases of convolution, as in this scenario, we just need to learn the weights for the specific filter kernel. Since the filter size is relatively small with respect to the image, the number of weights is reduced significantly. For any image, we generate several feature maps corresponding to different filter kernels. Each filter kernel learns to detect a different kind of feature. The feature maps created are again convolved with other filter kernels to learn even more complex features in subsequent layers.

Translation Equivariance

The convolution operation provides translational equivariance. That is, if a feature A in an input produces a specific feature B in the output, then even if feature A is translated around in the image, feature B would continue to be generated at different locations of the output.

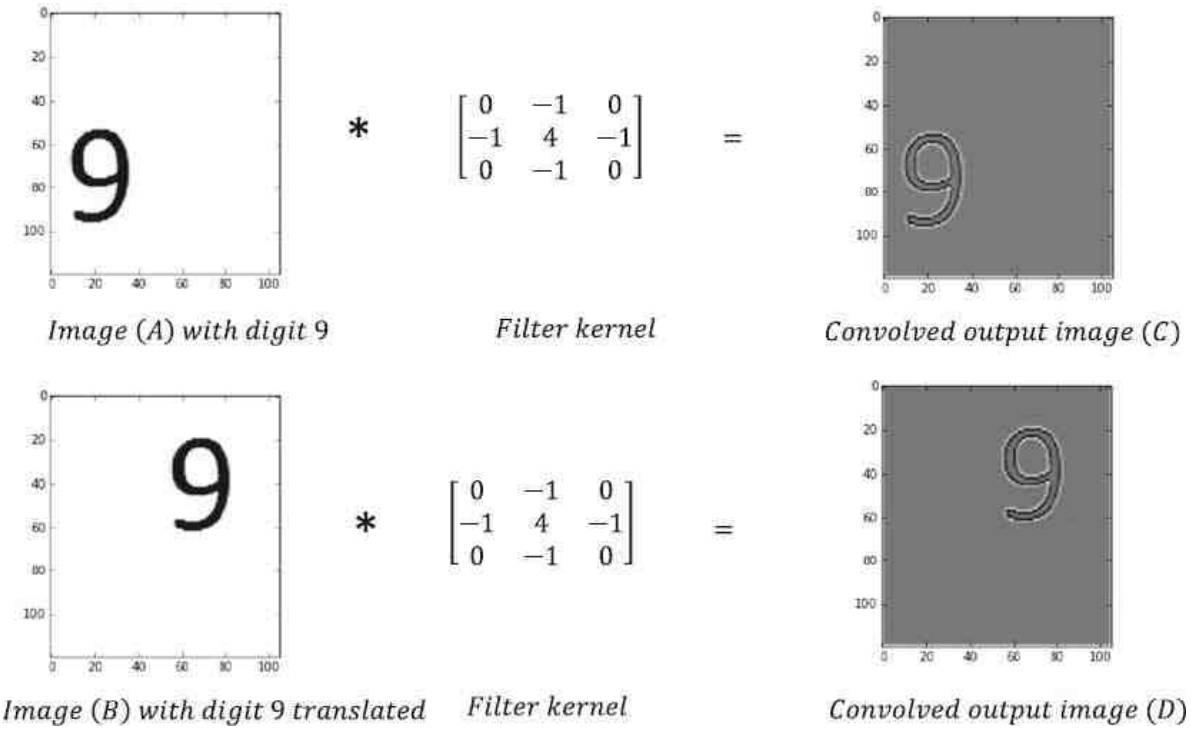


Figure 3-24: Translational equivariance illustration

In Figure 3-24 we can see that the digit 9 has been translated in Image (B) from its position in Image (A). Both the input images (A) and (B) have been convolved with the same filter kernel, and the same feature has been detected for the digit 9 in both output images (C) and (D) at different locations based on its location in the input. Convolution still produced the same feature for the digit irrespective of the translation. This property of convolution is called *translational equivariance*. In fact, if the digit is represented by a set of pixel intensities x, and f is the translation operation on x, while g is the convolution operation with a filter kernel, then the following holds true for convolution:

$$g(f(x))=f(g(x))$$

In our case, $f(x)$ produces the translated 9 in Image (B) and the translated 9 is convolved through g to produce the activated feature for 9, as seen in Image (D). This activated feature for 9 in Image (D) (i.e.,$g(f(x))$) could also have been achieved by

translating the activated 9 in Image (C) (i.e.,g(*x*)) through the same translation *f*.

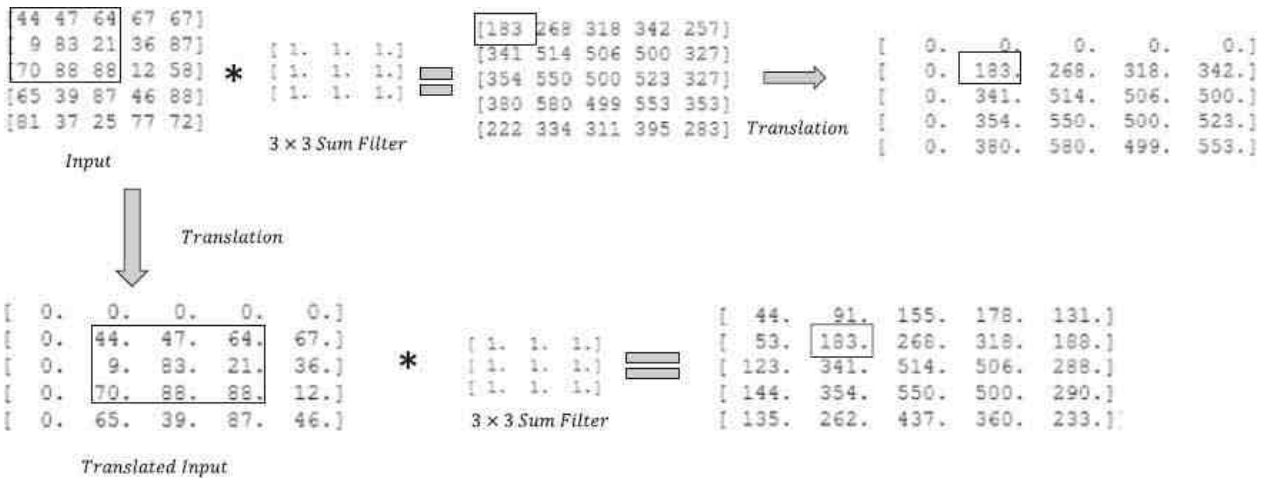


Figure 3-25: Illustration of equivariance with an example

It's a little easier to see equivariance with a small example, as illustrated in Figure 3-25. The part of the input image or 2D signal we are interested in is the left topmost block; i.e., $\begin{bmatrix} 44 & 47 & 64 \\ 9 & 83 & 21 \\ 70 & 88 & 88 \end{bmatrix}$. For easy reference, let's name the block A.

On convolving the input with the sum filter—i.e., $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$—block A would correspond to an output value of 183, which could be treated as the feature detector for A.

On convolving the same sum filter with the translated image, the shifted block A would still produce an output value of 183. Also, we can see that if we were to apply the same translation to the original convoluted image output, the value of 183 would appear at the same location as that of the output of the convoluted image after translation.

Translation Invariance Due to Pooling

Pooling provides some form of translational invariance based on the receptor field kernel size of the pooling. Let's take the case of max pooling, as illustrated in Figure 3-26. The digit in image *A* at a specific position is detected through a convolution filter *H* in the form of values 100 and 80 in the output feature map *P*. Also, the same digit appears in another image *B* in a slightly translated position with respect to image A. On convolving image *B* with filter *H*, the digit 9 is detected in the form of the same values of 100 and 80 in the feature map *P′*, but at a slightly displaced position from the one in. When these feature maps pass through the receptor field kernels of size 2×2 with stride 2 because of max pooling, the 100 and 80 values appear at the same location in both the output M and M′. In this way, max pooling provides some translational invariance to feature detection if the translation distance is not very high with respect to the size of the receptor field or kernel for max pooling.

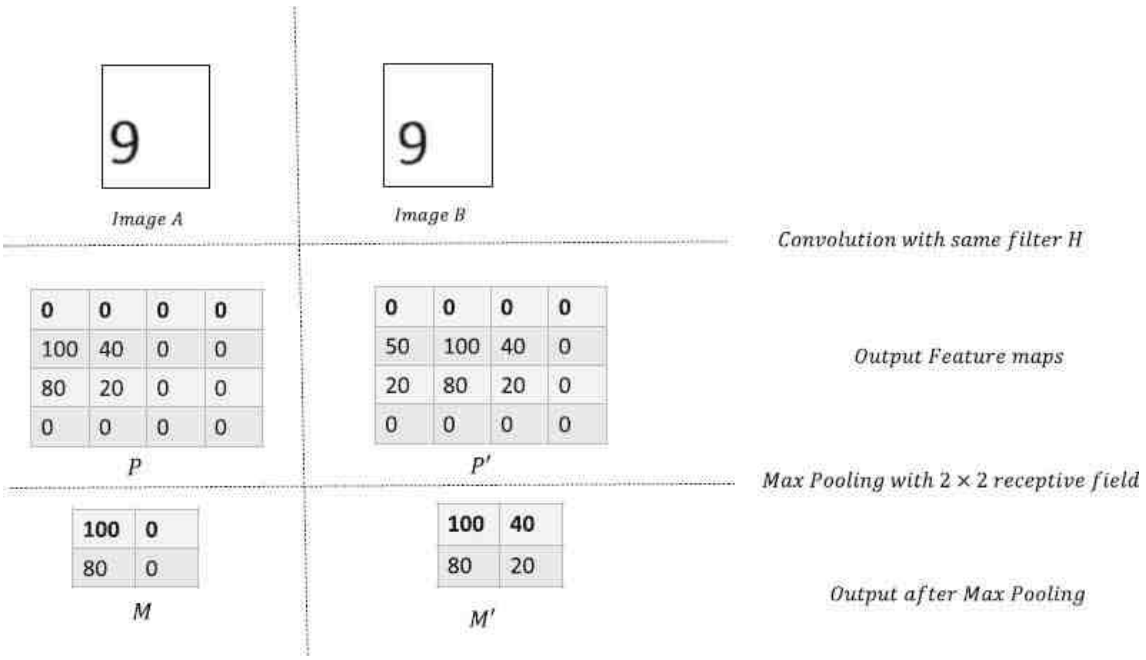


Figure 3-26: Translational invariance through max pooling

Similarly, average pooling takes the average of the values in a locality of a feature map based on the size of the receptor field kernel. So, if a specific feature is detected by high values in its feature map in a locality—let's say at regions of edges—then the averages would continue to be high even if the image were a little translated.

Dropout Layers and Regularization

Dropout is an activity to regularize weights in the fully connected layers of a convolutional neural network to avoid overfitting. However, it is not restricted to convolutional neural networks, but rather applies to all feed-forward neural networks. A specified proportion of neural network units, both hidden and visible, is randomly dropped at training time for each training sample in a mini batch so that the remaining neurons can learn important features all by themselves and not rely on cooperation from other neurons. When the neurons are dropped randomly, all the incoming and outgoing connections to those neurons are also dropped. Too much cooperation between neurons makes the neurons dependent on each other and they fail to learn distinct features. This high cooperation leads to overfitting since it does well on the training dataset, while if the test dataset is somewhat different from the training dataset the predictions on test dataset go haywire.

When the neuron units are dropped randomly, each such setting of remaining available neurons produces a different network. Let's suppose we have a network with N neural units; the number of possible neural network configuration possible is N^2. For each training sample in a mini batch, a different set of neurons is chosen at random based on the dropout probability. So, training a neural network with dropout is equivalent to training a set of different neural networks where each network very seldom gets trained, if at all.

As we can surmise, averaging the predictions from many different models reduces the variances of the ensemble model and reduces overfitting, and so we generally get better, more stable predictions.

For two class problem, trained on two different models M_1 and M_2, if the class probabilities for a datapoint are p_{11} and p_{22} for Model M_1 and p_{21} and p_{22} for Model M_2, then we take the average probability for the ensemble model of M_1 and M_2. The ensemble model would have a probability of $\frac{(p_{11}+p_{21})}{2}$ for Class 1 and $\frac{(p_{12}+p_{22})}{2}$ for Class 2.

Another averaging method would be to take the geometric mean of the predictions from different models. In this case, we would need to normalize over the geometric means to get the sum of new probabilities as 1.

The new probabilities for the ensemble model for the preceding example would be $\frac{\sqrt{p_{11}\times p_{21}}}{\sqrt{p_{11}\times p_{21}}+\sqrt{p_{12}\times p_{22}}}$ and $\frac{\sqrt{p_{12}\times p_{22}}}{\sqrt{p_{11}\times p_{21}}+\sqrt{p_{12}\times p_{22}}}$ respectively.

At test time, it is not possible to compute the predictions from all such possible networks and then average it out. Instead, the single neural network with all the weights and connections is used—but with weight adjustments. If a neural network unit is

retained with probability *p* during training, then the outgoing weights from that unit are scaled down by multiplying the weights by probability *p*. In general, this approximation for prediction on test datasets works well. It can be proved that for a model with a SoftMax output layer, the preceding arrangement is equivalent to taking predictions out of those individual models resulting from dropout and then computing their geometric mean.

In Figure 3-27, a neural network whose three units have been dropped randomly is presented. As we can see, all the input and output connections of the dropped units have also been dropped.

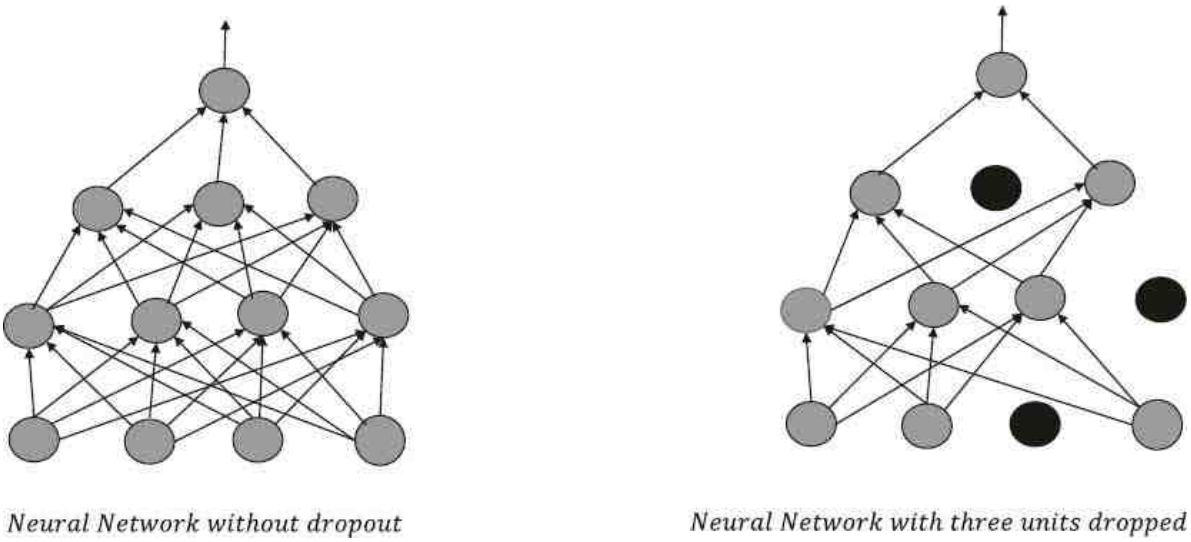


Figure 3-27: Neural network with three units dropped randomly

For a convolutional neural network, the units in the fully connected layers and their corresponding incoming and outgoing connections are usually dropped. Hence, the different filter-kernel weights do not require any adjustment while predicting for the test dataset.

Convolutional Neural Network for Digit Recognition on the MNIST Dataset

Now that we have gone through the basic building blocks for a convolutional neural network, let's see how good a CNN is at learning to classify the MNIST dataset. The detailed logic for a basic implementation in TensorFlow is documented in Listing 3-6. The CNN takes in images of height 28, width 28, and depth 3 corresponding to the RGB channels. The images go through the series of convolution, ReLU activations, and max pooling operations twice before being fed into the fully connected layer and finally to the output layer. The first convolution layer produces 64 feature maps, the second convolution layer provides 128 feature maps, and the fully connected layer has 1024 units. The max pooling layers have been chosen to reduce the feature map size by ¼. The feature maps can be considered 2D images.

Listing 3-6

```
#############################################################
##Import the required libraries and read the MNIST dataset
#############################################################
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from tensorflow.examples.tutorials.mnist import input_data
import time
mnist = input_data.read_data_sets("MNIST_data/",one_hot=True)

#########################################
## Set the value of the Parameters
#########################################

learning_rate = 0.01
epochs = 20
batch_size = 256
num_batches = mnist.train.num_examples/batch_size
input_height = 28
input_width = 28
n_classes = 10
dropout = 0.75
display_step = 1
```

```
filter_height = 5
filter_width = 5
depth_in = 1
depth_out1 = 64
depth_out2 = 128
########################################
# input output definition
########################################
x = tf.placeholder(tf.float32,[None,28*28])
y = tf.placeholder(tf.float32,[None,n_classes])
keep_prob = tf.placeholder(tf.float32)
########################################
## Store the weights
## Number of weights of filters to be learnt in 'wc1' => filter_height*filter_width*depth_
in*depth_out1
## Number of weights of filters to be learnt in 'wc1' => filter_height*filter_width*depth_
out1*depth_out2
## No of Connections to the fully Connected layer => Each maxpooling operation reduces the
image size to 1/4.
## So two maxpooling reduces the imase size to /16. There are depth_out2 number of images
each of size 1/16 ## of the original image size of input_height*input_width. So there is
total of
## (1/16)*input_height* input_width* depth_out2 pixel outputs which when connected to the
fully connected layer ## with 1024 units would provide (1/16)*input_height* input_width*
depth_out2*1024 connections.
########################################
weights = {
'wc1' : tf.Variable(tf.random_normal([filter_height,filter_width,depth_in,depth_out1])),
'wc2' : tf.Variable(tf.random_normal([filter_height,filter_width,depth_out1,depth_out2])),
'wd1' : tf.Variable(tf.random_normal([(input_height/4)*(input_height/4)* depth_out2,1024])),
'out' : tf.Variable(tf.random_normal([1024,n_classes]))
}
################################################
## In the 1st Convolutional Layer there are 64 feature maps and that corresponds to 64
biases in 'bc1'
## In the 2nd Convolutional Layer there are 128 feature maps and that corresponds to 128
biases in 'bc2'
## In the Fully Connected Layer there are 1024units and that corresponds to 1024 biases in
'bd1'
## In the output layet there are 10 classes for the Softmax and that corresponds to 10
biases in 'out'
################################################
biases = {
'bc1' : tf.Variable(tf.random_normal([64])),
'bc2' : tf.Variable(tf.random_normal([128])),
'bd1' : tf.Variable(tf.random_normal([1024])),
'out' : tf.Variable(tf.random_normal([n_classes]))
}
################################################
## Create the different layers
################################################

'''C O N V O L U T I O N L A Y E R'''
def conv2d(x,W,b,strides=1):
    x = tf.nn.conv2d(x,W,strides=[1,strides,strides,1],padding='SAME')
    x = tf.nn.bias_add(x,b)
    return tf.nn.relu(x)

''' P O O L I N G L A Y E R'''
def maxpool2d(x,stride=2):
     return tf.nn.max_pool(x,ksize=[1,stride,stride,1],strides=[1,stride,stride,1],padding='
     SAME')
################################################
## Create the feed forward model
################################################
def conv_net(x,weights,biases,dropout):
################################################
## Reshape the input in the 4 dimensional image
## 1st dimension - image index
## 2nd dimension - height
## 3rd dimension - width
## 4th dimension - depth
    x = tf.reshape(x,shape=[-1,28,28,1])
################################################
## Convolutional layer 1
    conv1 = conv2d(x,weights['wc1'],biases['bc1'])
    conv1 = maxpool2d(conv1,2)
## Convolutional layer 2
    conv2 = conv2d(conv1,weights['wc2'],biases['bc2'])
```

```
    conv2 = maxpool2d(conv2,2)
## Now comes the fully connected layer
    fc1 = tf.reshape(conv2,[-1,weights['wd1'].get_shape().as_list()[0]])
    fc1 = tf.add(tf.matmul(fc1,weights['wd1']),biases['bd1'])
    fc1 = tf.nn.relu(fc1)
## Apply Dropout
    fc1 = tf.nn.dropout(fc1,dropout)
## Output class prediction
    out = tf.add(tf.matmul(fc1,weights['out']),biases['out'])
    return out
#######################################################
# Defining the tensorflow Ops for different activities
#######################################################
pred = conv_net(x,weights,biases,keep_prob)
# Define loss function and optimizer
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=pred,labels=y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)
# Evaluate model
correct_pred = tf.equal(tf.argmax(pred,1),tf.argmax(y,1))
accuracy = tf.reduce_mean(tf.cast(correct_pred,tf.float32))
## initializing all variables
init = tf.global_variables_initializer()
#################################################
## Launch the execution Graph
#################################################
start_time = time.time()
with tf.Session() as sess:
    sess.run(init)
    for i in range(epochs):
        for j in range(num_batches):

            batch_x,batch_y = mnist.train.next_batch(batch_size)
            sess.run(optimizer, feed_dict={x:batch_x,y:batch_y,keep_prob:dropout})
            loss,acc = sess.run([cost,accuracy],feed_dict={x:batch_x,y:batch_y,keep_prob: 1.})
            if epochs % display_step == 0:
                print("Epoch:", '%04d' % (i+1),
                "cost=", "{:.9f}".format(loss),
                "Training accuracy","{:.5f}".format(acc))
    print('Optimization Completed')

    y1 = sess.run(pred,feed_dict={x:mnist.test.images[:256],keep_prob: 1})
    test_classes = np.argmax(y1,1)
    print('Testing Accuracy:',sess.run(accuracy,feed_dict={x:mnist.test.
    images[:256],y:mnist.test.labels[:256],keep_prob: 1}))
    f, a = plt.subplots(1, 10, figsize=(10, 2))

    for i in range(10):
        a[i].imshow(np.reshape(mnist.test.images[i],(28, 28)))
        print test_classes[i]

end_time = time.time()
print('Total processing time:',end_time - start_time)
```

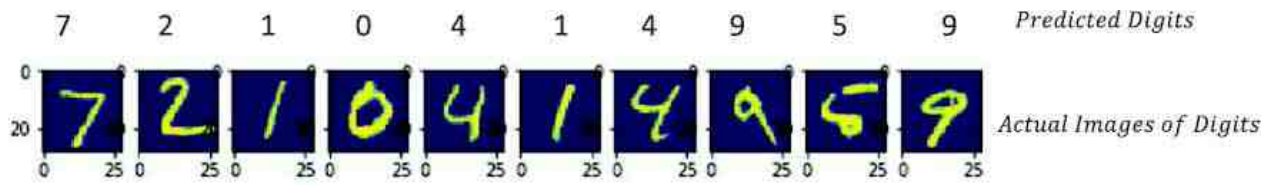


Figure 3-28: Predicted digits versus actual digits from CNN model

With the preceding basic convolutional neural network, which comprises two convolutional–maxpooling-ReLU pairs along with a fully connected layer before the final output SoftMax unit, we can achieve a test-set accuracy of 0.9765625 in just 20 epochs. As we saw previously through the multi-layer Perceptron approach in Chapter 2, with that method we were merely able to get around 91 percent accuracy with 1000 epochs. This is a testimony that for image-recognition problems convolutional neural networks work best.

One more thing I want to emphasize is the importance of tuning the model with the correct set of hyperparameters and prior information. Parameters such as the learning-rate selection can be very tricky since the cost function for neural networks is generally non-convex. A large learning rate can lead to faster convergence to a local minimum but might introduce oscillations, whereas a low learning rate will lead to very slow convergence. Ideally, the learning rate should be low enough that network parameters can converge to a meaningful local minima, and at the same time it should be high enough that the models can reach the minima faster. Generally, for the preceding neural network a learning rate of 0.01 is a little on the higher side, but

since we are only training the data on 20 epochs it works well. A lower learning rate wouldn't have achieved such a high accuracy with just 20 epochs. Similarly, the batch size chosen for the mini-batch version of stochastic gradient descent influences the convergence of the training process. A larger batch size might be good since the gradient estimates are less noisy; however, it may come at the cost of increased computation. One also needs to try out different filter sizes as well as experiment with different numbers of feature maps in each convolution layer. The kind of model architecture we choose works as a prior knowledge to the network.

Convolutional Neural Network for Solving Real-World Problems

We will now briefly discuss how to work on real-world image-analytics problems by going through a problem recently hosted in Kaggle by Intel that involved classifying different types of cervical cancer. In this competition, a model needs to be built that identifies a woman's cervix type based on images. Doing so will allow for the effective treatment of patients. Images specific to three types of cancer were provided for the competition. So, the business problem boils down to being a three-class image-classification problem. A basic solution approach to the problem is provided in Listing 3-7.

Listing 3-7

```
#########################################################
## Load the relevant libraries
#########################################################
from PIL import ImageFilter, ImageStat, Image, ImageDraw
from multiprocessing import Pool, cpu_count
from sklearn.preprocessing import LabelEncoder
import pandas as pd
import numpy as np
import glob
import cv2
import time
from keras.utils import np_utils
import os
import tensorflow as tf
import shuffle

#########################################################
## Read the input images and then resize the image to 64 x 64 x 3 size
#########################################################
def get_im_cv2(path):
    img = cv2.imread(path)
    resized = cv2.resize(img, (64,64), cv2.INTER_LINEAR)
    return resized

#########################################################
## Each of the folders corresponds to a different class
## Load the images into array and then define their output classes based on
## the folder number
#########################################################

def load_train():
    X_train = []
    X_train_id = []
    y_train = []
    start_time = time.time()

    print('Read train images')
    folders = ['Type_1', 'Type_2', 'Type_3']
    for fld in folders:
        index = folders.index(fld)
        print('Load folder {} (Index: {})'.format(fld, index))
        path = os.path.join('.', 'Downloads', 'Intel','train', fld, '*.jpg')
        files = glob.glob(path)

        for fl in files:
            flbase = os.path.basename(fl)
            img = get_im_cv2(fl)
            X_train.append(img)
            X_train_id.append(flbase)
            y_train.append(index)
    for fld in folders:
        index = folders.index(fld)
        print('Load folder {} (Index: {})'.format(fld, index))
        path = os.path.join('.', 'Downloads', 'Intel','Additional', fld, '*.jpg')
        files = glob.glob(path)

        for fl in files:
```

```
            flbase = os.path.basename(fl)
            img = get_im_cv2(fl)
            X_train.append(img)
            X_train_id.append(flbase)
            y_train.append(index)

    print('Read train data time: {} seconds'.format(round(time.time() - start_time, 2)))
    return X_train, y_train, X_train_id

#####################################################################
## Load the test images
#####################################################################

def load_test():
    path = os.path.join('.', 'Downloads', 'Intel','test', '*.jpg')
    files = sorted(glob.glob(path))

    X_test = []
    X_test_id = []
    for fl in files:
        flbase = os.path.basename(fl)
        img = get_im_cv2(fl)
        X_test.append(img)
        X_test_id.append(flbase)
    path = os.path.join('.', 'Downloads', 'Intel','test_stg2', '*.jpg')
    files = sorted(glob.glob(path))
    for fl in files:
        flbase = os.path.basename(fl)
        img = get_im_cv2(fl)
        X_test.append(img)
        X_test_id.append(flbase)

    return X_test, X_test_id

#################################################
## Normalize the image data to have values between 0 and 1
## by diving the pixel intensity values by 255.
## Also convert the class label into vectors of length 3 corresponding to
## the 3 classes
## Class 1 - [1 0 0]
## Class 2 - [0 1 0]
## Class 3 - [0 0 1]
#################################################
def read_and_normalize_train_data():
    train_data, train_target, train_id = load_train()

    print('Convert to numpy...')
    train_data = np.array(train_data, dtype=np.uint8)
    train_target = np.array(train_target, dtype=np.uint8)

    print('Reshape...')
    train_data = train_data.transpose((0, 2,3, 1))
    train_data = train_data.transpose((0, 1,3, 2))

    print('Convert to float...')
    train_data = train_data.astype('float32')
    train_data = train_data / 255
    train_target = np_utils.to_categorical(train_target, 3)

    print('Train shape:', train_data.shape)
    print(train_data.shape[0], 'train samples')
    return train_data, train_target, train_id

#############################################################
## Normalize test-image data
#############################################################

def read_and_normalize_test_data():
    start_time = time.time()
    test_data, test_id = load_test()

    test_data = np.array(test_data, dtype=np.uint8)
    test_data = test_data.transpose((0,2,3,1))
    train_data = test_data.transpose((0, 1,3, 2))

    test_data = test_data.astype('float32')
    test_data = test_data / 255

    print('Test shape:', test_data.shape)
```

```
    print(test_data.shape[0], 'test samples')
    print('Read and process test data time: {} seconds'.format(round(time.time() -
    start_time, 2)))
    return test_data, test_id

############################################################
## Read and normalize the train data
############################################################

train_data, train_target, train_id = read_and_normalize_train_data()

############################################################
## Shuffle the input training data to aid stochastic gradient descent
############################################################
list1_shuf = []
list2_shuf = []
index_shuf = range(len(train_data))
shuffle(index_shuf)
for i in index_shuf:
    list1_shuf.append(train_data[i,:,:,:])
    list2_shuf.append(train_target[i,])
list1_shuf = np.array(list1_shuf,dtype=np.uint8)
list2_shuf = np.array(list2_shuf,dtype=np.uint8)

############################################################
## TensorFlow activities for Network Definition and Training
############################################################
## Create  the different layers

channel_in = 3
channel_out = 64
channel_out1 = 128

'''C O N V O L U T I O N    L A Y E R'''
def conv2d(x,W,b,strides=1):
    x = tf.nn.conv2d(x,W,strides=[1,strides,strides,1],padding='SAME')
    x = tf.nn.bias_add(x,b)
    return tf.nn.relu(x)

''' P O O L I N G  L A Y E R'''
def maxpool2d(x,stride=2):
    return tf.nn.max_pool(x,ksize=[1,stride,stride,1],strides=[1,stride,stride,1],
    padding='SAME')

## Create the feed-forward model

def conv_net(x,weights,biases,dropout):

    ## Convolutional layer 1
    conv1 = conv2d(x,weights['wc1'],biases['bc1'])
    conv1 = maxpool2d(conv1,stride=2)
    ## Convolutional layer 2
    conv2a = conv2d(conv1,weights['wc2'],biases['bc2'])
    conv2a = maxpool2d(conv2a,stride=2)
    conv2 = conv2d(conv2a,weights['wc3'],biases['bc3'])
    conv2 = maxpool2d(conv2,stride=2)

    ## Now comes the fully connected layer

    fc1 = tf.reshape(conv2,[-1,weights['wd1'].get_shape().as_list()[0]])
    fc1 = tf.add(tf.matmul(fc1,weights['wd1']),biases['bd1'])
    fc1 = tf.nn.relu(fc1)

    ## Apply Dropout
    fc1 = tf.nn.dropout(fc1,dropout)
    ## Another fully Connected Layer
    fc2 = tf.add(tf.matmul(fc1,weights['wd2']),biases['bd2'])
    fc2 = tf.nn.relu(fc2)
    ## Apply Dropout
    fc2 = tf.nn.dropout(fc2,dropout)

    ## Output class prediction

    out = tf.add(tf.matmul(fc2,weights['out']),biases['out'])
    return out

#######################################################
## Define several parameters for the network and learning
#######################################################
```

```
start_time = time.time()
learning_rate = 0.01
epochs = 200
batch_size = 128
num_batches = list1_shuf.shape[0]/128
input_height = 64
input_width = 64
n_classes = 3
dropout = 0.5
display_step = 1
filter_height = 3
filter_width = 3
depth_in = 3
depth_out1 = 64
depth_out2 = 128
depth_out3 = 256

########################################################
# input-output definition
########################################################

x = tf.placeholder(tf.float32,[None,input_height,input_width,depth_in])
y = tf.placeholder(tf.float32,[None,n_classes])
keep_prob = tf.placeholder(tf.float32)

########################################################
## Define the weights and biases
########################################################

weights = {
    'wc1' : tf.Variable(tf.random_normal([filter_height,filter_width,depth_in,depth_out1])),
    'wc2' : tf.Variable(tf.random_normal([filter_height,filter_width,depth_out1,depth_out2])),
    'wc3' : tf.Variable(tf.random_normal([filter_height,filter_width,depth_out2,depth_out3])),
    'wd1' : tf.Variable(tf.random_normal([(input_height/8)*(input_height/8)*256,512])),
    'wd2' : tf.Variable(tf.random_normal([512,512])),
    'out' : tf.Variable(tf.random_normal([512,n_classes]))
}
biases = {
    'bc1' : tf.Variable(tf.random_normal([64])),
    'bc2' : tf.Variable(tf.random_normal([128])),
    'bc3' : tf.Variable(tf.random_normal([256])),
    'bd1' : tf.Variable(tf.random_normal([512])),
    'bd2' : tf.Variable(tf.random_normal([512])),
    'out' : tf.Variable(tf.random_normal([n_classes]))
}

########################################################
## Define the TensorFlow ops for training
########################################################

pred = conv_net(x,weights,biases,keep_prob)

# Define loss function and optimizer
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=pred,labels=y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)

# Evaluate model

correct_pred = tf.equal(tf.argmax(pred,1),tf.argmax(y,1))
accuracy = tf.reduce_mean(tf.cast(correct_pred,tf.float32))

## Define the initialization op

init = tf.global_variables_initializer()
########################################################
## Launch the execution graph and invoke the training
########################################################
start_time = time.time()
with tf.Session() as sess:
    sess.run(init)

    for i in range(epochs):

        for j in range(num_batches):

            batch_x,batch_y = list1_shuf[i*(batch_size):(i+1)*(batch_size)],
            list2_shuf[i*(batch_size):(i+1)*(batch_size)]
            sess.run(optimizer, feed_dict={x:batch_x,y:batch_y,keep_prob:dropout})
            loss,acc = sess.run([cost,accuracy],feed_dict={x:batch_x,y:batch_y,keep_prob: 1.})
```

```
        if epochs % display_step == 0:
            print("Epoch:", '%04d' % (i+1),
                  "cost=", "{:.9f}".format(loss),
                  "Training accuracy","{:.5f}".format(acc))

    print('Optimization Completed')
end_time = time.time()
print('Total processing time:',end_time - start_time)

-- output --

('Epoch:', '0045', 'cost=', '0.994687378', 'Training accuracy', '0.53125')
('Epoch:', '0046', 'cost=', '1.003623009', 'Training accuracy', '0.52344')
('Epoch:', '0047', 'cost=', '0.960040927', 'Training accuracy', '0.56250')
('Epoch:', '0048', 'cost=', '0.998520255', 'Training accuracy', '0.54688')
('Epoch:', '0049', 'cost=', '1.016047001', 'Training accuracy', '0.50781')
('Epoch:', '0050', 'cost=', '1.043521643', 'Training accuracy', '0.49219')
('Epoch:', '0051', 'cost=', '0.959320068', 'Training accuracy', '0.58594')
('Epoch:', '0052', 'cost=', '0.935006618', 'Training accuracy', '0.57031')
('Epoch:', '0053', 'cost=', '1.031400681', 'Training accuracy', '0.49219')
('Epoch:', '0054', 'cost=', '1.023633003', 'Training accuracy', '0.50781')
('Epoch:', '0055', 'cost=', '1.007938623', 'Training accuracy', '0.53906')
('Epoch:', '0056', 'cost=', '1.033236384', 'Training accuracy', '0.46094')
('Epoch:', '0057', 'cost=', '0.939492166', 'Training accuracy', '0.60938')
('Epoch:', '0058', 'cost=', '0.986051500', 'Training accuracy', '0.56250')
('Epoch:', '0059', 'cost=', '1.019751549', 'Training accuracy', '0.51562')
('Epoch:', '0060', 'cost=', '0.955037951', 'Training accuracy', '0.57031')
('Epoch:', '0061', 'cost=', '0.963475347', 'Training accuracy', '0.58594')
('Epoch:', '0062', 'cost=', '1.019685864', 'Training accuracy', '0.50000')
('Epoch:', '0063', 'cost=', '0.970604420', 'Training accuracy', '0.53125')
('Epoch:', '0064', 'cost=', '0.962844968', 'Training accuracy', '0.54688')
```

This model achieves a log-loss of around 0.97 in the competition leaderboard, whereas the best model for this competition achieved around 0.78 log-loss. This is because the model is a basic implementation that doesn't take care of other advanced concepts in image processing. We will study one such advanced technique called transfer learning later in the book that works well when the number of images provided is less. A few points about the implementation that might be of interest to the reader are as follows:

- The images have been read as a three-dimensional Numpy array and resized through OpenCV and then appended to a list. The list is then converted to Numpy array, and hence we get a four-dimensional Numpy array or Tensor for both the training and testing datasets. The training and testing image Tensors have been transposed to have the dimensions aligned in order of image number, location along height of image, location along width of image, and image channels.
- The images have been normalized to have values between 0 and 1 by dividing by the maximum value of pixel intensity; i.e., 255. This aids the gradient-based optimization.
- The images have been shuffled randomly so that the mini batches have images of the three classes randomly arranged.
- The rest of the network implementation is like the MNIST classification problem but with three layers of the convolution-ReLU-max pooling combination and two layers of fully connected layers before the final SoftMax output layer.
- The code involving prediction and submission has been left out here.

Batch Normalization

Batch normalization was invented by Sergey Ioffe and Christian Szegedy and is one of the pioneering elements in the deep-learning space. The original paper for batch normalization is titled "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift" and can be located at https://arxiv.org/abs/1502.03I67.

When training a neural network through stochastic gradient descent, the distribution of the inputs to each layer changes due to the update of weights on the preceding layers. This slows down the training process and makes it difficult to train very deep neural networks. The training process for neural networks is complicated by the fact that the input to any layer is dependent on the parameters for all preceding layers, and thus even small parameter changes can have an amplified effect as the network grows. This leads to input-distribution changes in a layer.

Now, let's try to understand what might go wrong when the input distribution to the activation functions in a layer change because of weight changes in the preceding layer.

A sigmoid or tanh activation function has good linear gradients only within a specified range of its input, and the gradient drops to zero once the inputs grow large.

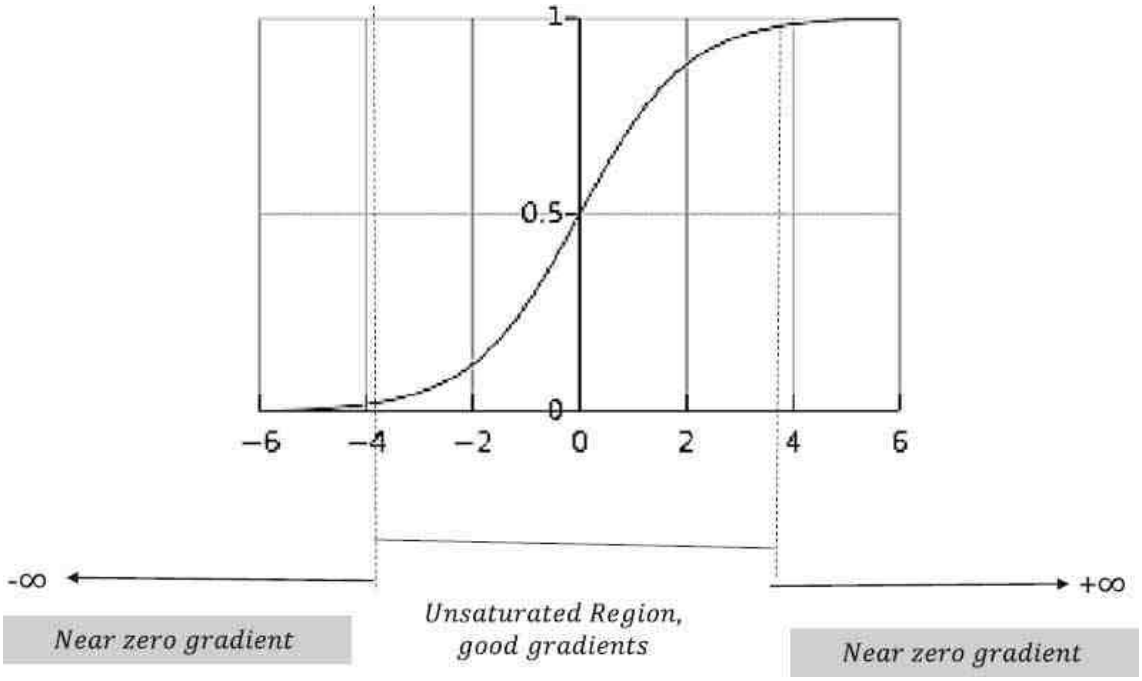


Figure 3-29: Sigmoid function with its small unsaturated region

The parameter change in the preceding layers might change the input probability distribution to a sigmoid units layer in such a way that most of the inputs to the sigmoids belong to the saturation zone and hence produce near-zero gradients, as shown in Figure 3-29. Because of these zero or near-zero gradients, the learning becomes terribly slow or stops entirely. One way to avoid this problem is to have rectified linear units (ReLUs). The other way to avoid this problem is to keep the distribution of inputs to the sigmoid units stable within the unsaturated zone so that stochastic gradient descent doesn't get stuck in a saturated zone.

This phenomenon of change in the distribution of the input to the internal network units has been referred to by the inventors of the batch normalization process as *internal covariate shift*.

Batch normalization reduces the internal covariate shift by normalizing the inputs to a layer to have a zero mean and unit standard deviation. While training, the mean and standard deviation are estimated from the mini-batch samples at each layer, whereas at test-prediction time generally the population variance and mean are used.

If a layer receives a vector $x=\left[x_1 x_2 \ldots x_n\right]^T \in \mathbb{R}^{n\times 1}$ of input activations from the preceding layers, then at each mini batch consisting of m data points the input activations are normalized as follows:

$$\hat{x}_i=\frac{x_i-E\left[x_i\right]}{\sqrt{Var\left[x_i\right]+\epsilon}}$$

where

$$u_B=\frac{1}{m}\sum_{k=1}^{m}x_i^{(k)}$$

$$\sigma_B^2=\frac{1}{m}\sum_{k=1}^{m}\left(x_i^{(k)}-E\left[x_i\right]\right)^2$$

Statistically, u_B and σ_B^2 are nothing but the sample mean and biased sample standard variance.

Once the normalization is done, $\hat{x}_i$ is not fed to the activation function directly but rather is scaled and shifted by introducing parameters γ and β before feeding it to the activation functions. If we restrict the input activations to the normalized values, they may change what the layer can represent. So, the idea is to apply a linear transformation on the normalized value through the following transformation so that if the network, through training, feels that the original values before any transformation are good

for the network it can recover the original values. The actual transformed input activation y_i fed to the activation function is given by

$$y_i = \gamma \hat{x}_i + \beta$$

The parameters u_B, σ_B^2, γ, and β are to be learned by backpropagation much like the other parameters. As stated earlier, the model might learn $\gamma = Var[x_i]$ and $\beta = E[x_i]$ if it feels the original values from the network are more desirable.

A very natural question that may come up is why we are taking the mini-batch mean u_B and variance σ_B^2 as parameters to be learned through batch propagation and not estimating them as running averages over mini batches for normalization purposes. This doesn't work, since the u_B and σ_B^2 are dependent on other parameters of the model through x_i, and when we directly estimate these as running averages this dependency is not accounted for in the optimization process. To keep those dependencies intact, u_B and σ_B^2 should participate in the optimization process as parameters since the gradients of the u_B and σ_B^2 with respect to the other parameters on which x_i depends are critical to the learning process. The overall effect of this optimization modifies the model in such a way that the input $\hat{x}_i$ keeps zero mean and unit standard deviation.

During inference or testing time the population statistics $E[x_i]$ and $Var[x_i]$ are used for normalization by keeping a running average of the mini-batch statistics.

$$E[x_i] = E[u_B]$$

$$Var[x_i] = \left(\frac{m}{m-1}\right) E[\sigma_B^2]$$

This correction factor is required to get an unbiased estimate of the Population variance. Mentioned here are a couple of the advantages of batch normalization:

- Models can be trained faster because of the removal or reduction of internal covariate shift. A smaller number of training iterations would be required to achieve good model parameters.
- Batch normalization has some regularizing power and at times eliminates the need for dropout.
- Batch normalization works well with convolutional neural networks wherein there is one set of γ and β for each output feature map.

Different Architectures in Convolutional Neural Networks

In this section, we will go through a few widely used convolutional neural network architectures used today. These network architectures are not only used for classification, but also, with minor modification, are used in segmentation, localization, and detection. Also, there are pre-trained versions of each of these networks that enable the community to do transfer learning or fine-tune the models. Except LeNet, almost all the CNN models have won the ImageNet competition for classification of a thousand classes.

LeNet

The first successful convolutional neural network was developed by Yann LeCunn in 1990 for classifying handwritten digits successfully for OCR-based activities such as reading ZIP codes, checks, and so on. LeNet5 is the latest offering from Yann LeCunn and his colleagues. It takes in 32×32-size images as input and passes them through a convolutional layer to produce six feature maps of size 28×28. The six feature maps are then sub-sampled to produce six output images of size 14×14. Sub-sampling can be thought of as a pooling operation. The second convolutional layer has 16 feature maps of size 10×10 while the second sub-sampling layer reduces the feature map sizes to 5×5. This is followed by two fully connected layers of 120 and 84 units respectively, followed by the output layer of ten classes corresponding to ten digits. Figure 3-30 represents the LeNet5 architectural diagrams.

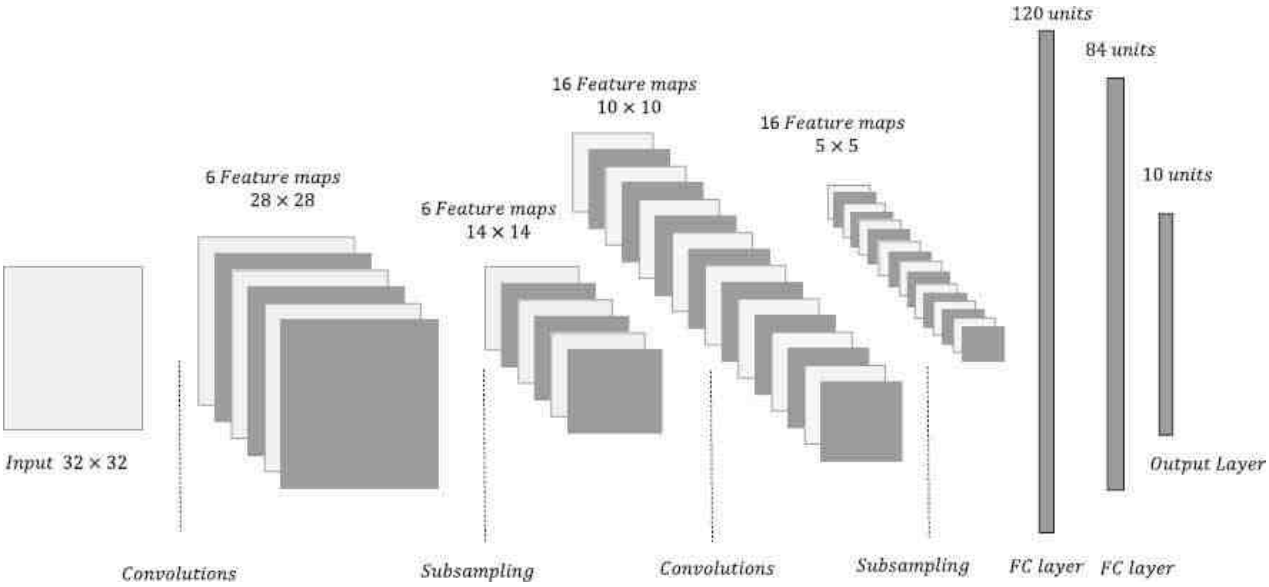


Figure 3-30: LeNet5 architectural diagram

Key features of the LeNet5 network are as follows:

- The pooling though sub-sampling takes 2×2 neighborhood patches and sums up the four-pixel intensity values. The sum is scaled by a trainable weight and a bias and then fed through a sigmoid activation function. This is a little different from what is done for max pooling and average pooling.
- The filter kernel used for convolution is of size 5×5. The output units are radial basis function (RBF) units instead of the SoftMax functions that we generally use. The 84 units of the fully connected layers had 84 connections to each of the classes and hence 84 corresponding weights. The 84 weights/class represent each class's characteristics. If the inputs to those 84 units are very close to the weights corresponding to a class, then the inputs are more likely to belong to that class. In a SoftMax we look at the dot product of the inputs to each of the class's weight vectors, while in RBF units we look at the Euclidean distance between the input and the output class representative's weight vectors. The greater the Euclidean distance, the smaller the chance is of the input belonging to that class. The same can be converted to probability by exponentiating the negative of the distance and then normalizing over the different classes. The Euclidean distances over all the classes for an input record would act as the loss function for that input. Let $x=\left[x_1 x_2 \ldots x_{84}\right]^T \in \mathbb{R}^{84\times 1}$ be the output vector of the fully connected layer. For each class, there would be 84 weight connections. If the representative class's weight vector for the *ith* class is $w_i \in \mathbb{R}^{84\times 1}$ then the output of the *ith* class unit can be given by the following:

$$\left\|x-w_i\right\|_2^2=\sum_{j=1}^{84}\left(x_j-w_{ij}\right)^2$$

- The representative weights for each class are fixed beforehand and are not learned weights.

AlexNet

The AlexNet CNN architecture was developed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton in 2012 to win the 2012 ImageNet ILSVRC (ImageNet Large-Scale Visual Recognition Challenge). The original paper pertaining to AlexNet is titled "ImageNet Classification with Deep Convolutional Neural Networks" and can be located at https://papers.nips.cc/paper/4824-imagenet-classification- with-deep-convolutional-neural-networks.pdf.

It was the first time that a CNN architecture beat other methods by a huge margin. Their network achieved an error rate of 15.4 percent on its top five predictions as compared to the 26.2 percent error rate for the second-best entry. The architectural diagram of AlexNet is represented in Figure 3-31.

AlexNet consists of five convolutional layers, max pooling layers, and dropout layers, and three fully connected layers in addition to the input and output layer of a thousand class units. The inputs to the network are images of size 224×224×3. The first convolutional layer produces 96 feature maps corresponding to 96 filter kernels of size 11×11×3 with strides of four pixel units. The second convolutional layer produces 256 feature maps corresponding to filter kernels of size 5×5×48. The first two convolutional layers are followed by max pooling layers, whereas the next three convolutional layers are placed one after another without any intermediate max pooling layers. The fifth convolutional layer is followed by a max pooling layer, two fully connected layers of 4096 units, and finally a SoftMax output layer of one thousand classes. The third convolutional layer has

384 filter kernels of size 3×3×256, whereas the fourth and fifth convolutional layers have 384 and 256 filter kernels each of size 3×3×192. A dropout of 0.5 was used in the last two fully connected layers. You will notice that the depth of the filter kernels for convolutions is half the number of feature maps in the preceding layer for all but the third convolutional layer. This is because AlexNet was at that time computationally expensive and hence the training had to be split between two separate GPUs. However, if you observe carefully, for the third convolutional activity there is cross-connectivity for convolution and so the filter kernel is of dimension 3×3×256 and not 3×3×128. The same kind of cross-connectivity applies to the fully connected layers, and hence they behave as ordinary fully connected layers with 4096 units.

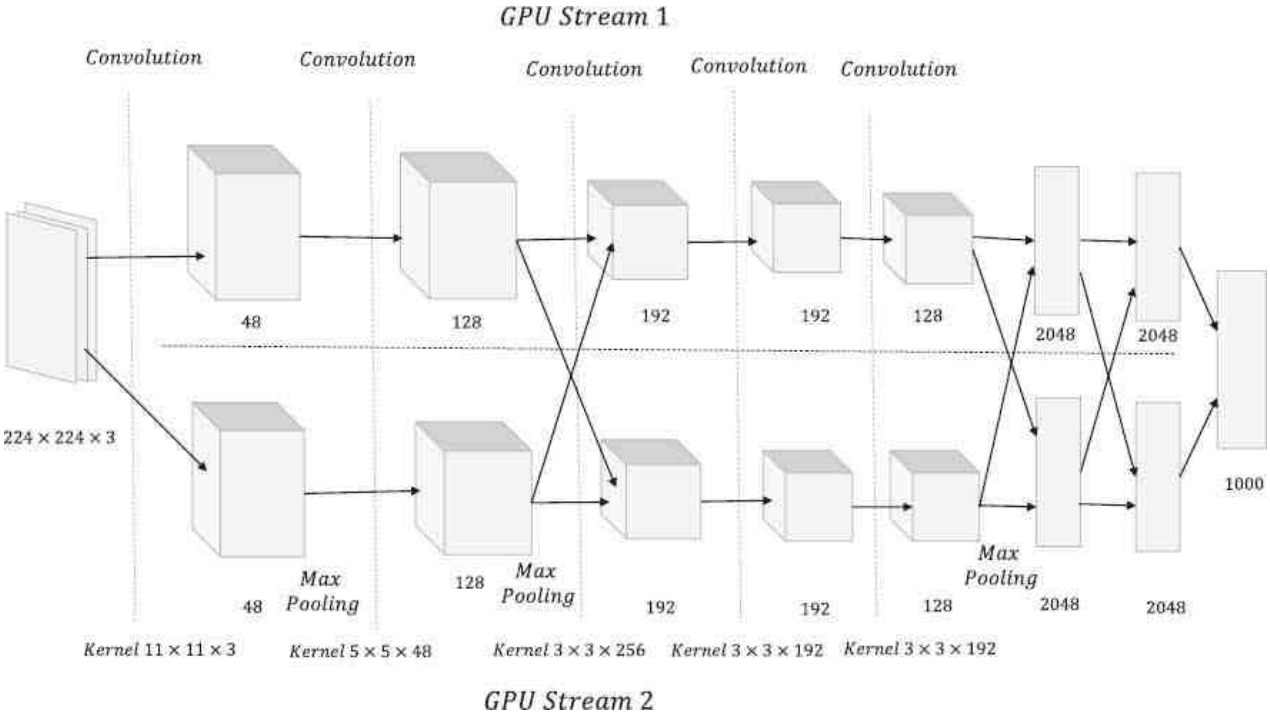


Figure 3-31: AlexNet architecture

The key features of AlexNet are as follows:

- ReLU activation functions were used for non-linearity. They had a huge impact since RELUs are significantly easier to compute and have constant non-saturating gradients as opposed to sigmoid and tanh activation functions, whose gradients tend to zero for very high and low values of input.
- Dropout was used to reduce overfitting in the model.
- Overlapping pooling was used as opposed to non-overlapping pooling.
- The model was trained on two GPU GTX 580 for around five days for fast computation.
- The dataset size was increased through data-augmentation techniques, such as image translations, horizontal reflections, and patch extractions.

VGG16

The VGG group in 2014 were runners up in the ILSVRC-2014 competition with a 16-layer architecture named VGG16. It uses a deep yet simple architecture that has gained a lot of popularity since. The paper pertaining to the VGG network is titled "Very Deep Convolutional Networks for Large-Scale Image Recognition" and is authored by Karen Simonyan and Andrew Zisserman. The paper can be located at https://arxiv.org/abs/1409.1556.

Instead of using a large kernel-filter size for convolution, VGG16 architecture used 3×3 filters and followed it up with ReLU activations and max pooling with a 2×2 receptive field. The inventors' reasoning was that using two 3×3 convolution layers is equivalent to having one 5×5 convolution while retaining the advantages of a smaller kernel-filter size; i.e., realizing a reduction in the number of parameters and realizing more non-linearity because of two convolution-ReLU pairs as opposed to one. A special property of this network is that as the spatial dimensions of the input volume reduces because of convolution and max pooling, the number of feature maps increases due to the increase in the number of filters as we go deep into the network.

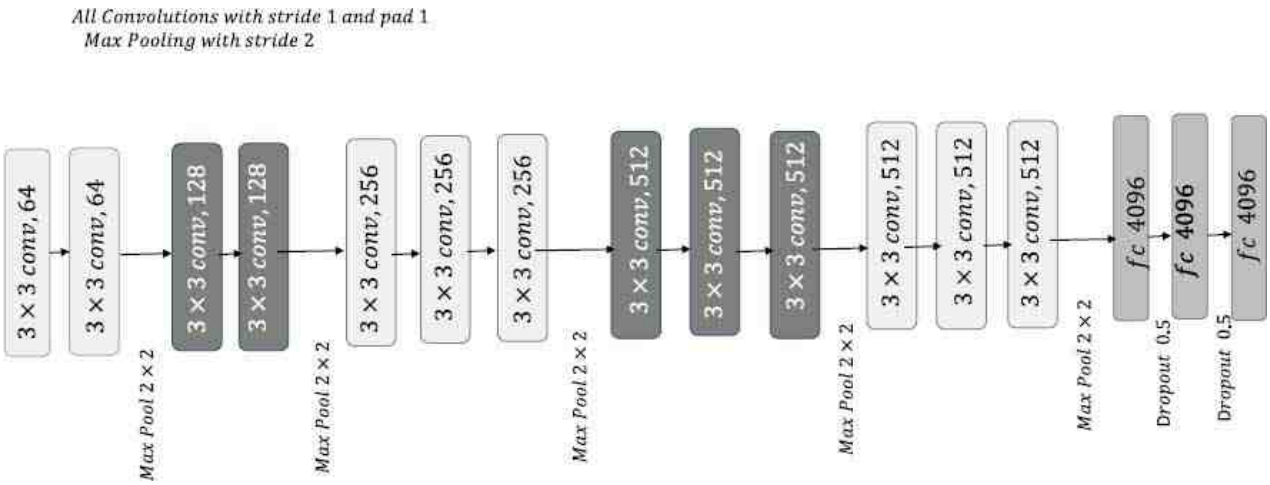


Figure 3-32: VGG16 architecture

Figure 3-32 represents the architecture of VGG16. The input to the network are images of size 224×224×3. The first two convolutional layers produce 64 feature maps, each followed by max pooling. The filters for convolution are of spatial size 3×3 with a stride of 1 and pad of 1. Max pooling is of size 2×2 with stride of 2 for the whole network. The third and fourth convolutional layers produce 128 feature maps, each followed by a max pooling layer. The rest of the network follows in a similar fashion, as shown in the Figure 3-32. At the end of the network there are three fully connected layers of 4096 units, each followed by the output SoftMax layer of a thousand classes. Dropout is set at 0.5 for the fully connected layers. All the units in the network have ReLU activations.

ResNet

ResNet is a 152-layer-deep convolutional neural network from Microsoft that won the ILSVRC 2015 competition with an error rate of only 3.6 percent, which is perceived to be better than the human error rate of 5-10 percent. The paper on ResNet, authored by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, is titled "Deep Residual Learning for Image Recognition" and can be located at https://arxiv.org/abs/1512.03385. Apart from being deep, ResNet implements a unique idea of residual block. After each series of convolution-ReLUs-convolution operations, the input to the operation is fed back to the output of the operation. In traditional methods while doing Convolution and other transformations, we try to fit an underlying mapping to the original data to solve the classification task. However, with ResNet's residual block concept, we try to learn a residual mapping and not a direct mapping from the input to output. Formally, in each small block of activities we add the input to the block to the output. This is illustrated in Figure 3-33. This concept is based on the hypothesis that it is easier to fit a residual mapping than to fit the original mapping from input to output.

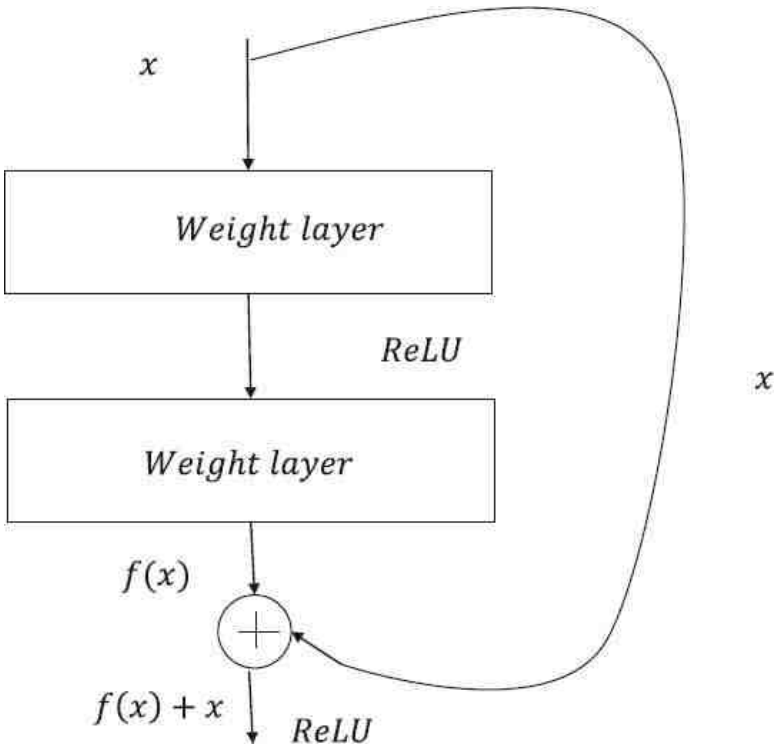


Figure 3-33: Residual block

Transfer Learning

Transfer learning in a broad sense refers to storing knowledge gained while solving a problem and using that knowledge for a different problem in a similar domain. Transfer learning has been hugely successful in the field of deep learning for a variety of reasons.

Deep-learning models in general have a huge number of parameters because of the nature of the hidden layers and the connectivity scheme within the different units. To train such a huge model, lots of data is required or the model will suffer from overfitting problems. In many problems, the huge amount of data required to train the model is not available but the nature of the problem requires a deep-learning solution in order to have a reasonable impact. For instance, in image processing for object recognition, deep-learning models are known to provide state-of-the-art solutions. In such cases, transfer learning can be used to generate generic features from a pre-trained deep-learning model and then use those features to build a simple model to solve the problem. So, the only parameters for this problem are the ones used to build the simple model. The pre-trained models are generally trained on a huge corpus of data and thus have reliable parameters.

When we process images through several layers of convolutions, the initial layers learn to detect very generic features such as curls and edges. As the network grows deeper, the convolutional layers in the deeper layers learn to detect more complex features relevant to the specific kind of dataset. For example, in a classification the deeper layers would learn to detect features such as eyes, nose, face, and so forth.

Let's assume we have a VGG16 architecture model trained on one thousand categories of the ImageNet dataset. Now, if we get a smaller dataset that has fewer categories of images similar to those of the VGG16 pre-trained model dataset, then we can use the same VGG16 model up to the fully connected layer and then replace the output layer with the new classes. Also, we keep the weights of the network fixed till the fully connected layer and only train the model to learn the weights from the fully connected layer to the output layer. This is because the dataset's nature is the same as the smaller dataset's, and thus the features learned in the pre-trained model through the different parameters are good enough for the new classification problem, and we only need to learn the weights from the fully connected layer to the output layer. This is a huge reduction in the number of parameters to learn, and it will reduce the overfitting. Had we trained the small dataset using VGG16 architecture, it might have suffered from severe overfitting because of the large number of parameters to learn on a small dataset.

What do you do when the dataset's nature is very different from that of the dataset used for the pre-trained model?

Well, in that case, we can use the same pre-trained model but fix only the parameters for the first couple of sets of convolution-ReLUs-max pooling layers and then add a couple of convolution-ReLU-max pooling layers that would learn to detect features intrinsic to the new dataset. Finally, we would have to have a fully connected layer followed by the output layer. Since we are using the weights of the initial sets of convolution-ReLUs-max pooling layers from the pre-trained VGG16 network, the

parameters with respect to those layers need not be learned. As mentioned earlier, the early layers of convolution learn very generic features, such as edges and curves, that are applicable to all kinds of images. The rest of the network would need to be trained to learn specific features inherent to the specific problem dataset.

Guidelines for Using Transfer Learning

The following are a few guidelines as to when and how to use a pre-trained model for transfer learning:

- The size of the problem dataset is large, and the dataset is similar to the one used for the pre-trained model—this is the ideal scenario. We can retain the whole model architecture as it is except maybe the output layer when it has a different number of classes than the pre-trained one. We can then train the model using the weights of the pre-trained model as initial weights for the model.

- The size of the problem dataset is large, but the dataset is dissimilar to the one used for the pre-trained model—in this case, since the dataset is large, we can train the model from scratch. The pre-trained model will not give any gains here since the dataset's nature is very dissimilar, and since we have a large dataset we can afford to train the whole network from scratch without overfitting related to large networks trained on small datasets.

- The size of the problem dataset is small, and the dataset is similar to the one used for the pre-trained model—this is the case that we discussed earlier. Since the dataset content is similar, we can reuse the existing weights of most of the model and only change the output layer based on the classes in the problem dataset. Then, we train the model only for the weights in the last layer. For example, if we get images like ImageNet for only dogs and cats we can pick up a VGG16 model pre-trained on ImageNet and just modify the output layer to have two classes instead of a thousand. For the new network model, we just need to train the weights specific to the final output layer, keeping all the other weights the same as those of the pre-trained VGG16 model.

- The size of the problem dataset is small and the dataset is dissimilar to the one used in the pre-trained model—this is not such a good situation to be in. As discussed earlier, we can freeze the weights of a few initial layers of a pre-trained network and then train the rest of the model on the problem dataset. The output layer, as usual, needs to be changed as per the number of classes in the problem dataset. Since we don't have a large dataset, we are trying to reduce the number of parameters as much as possible by reusing weights of the initial layers of the pre-trained model. Since the first few layers of CNN learn generic features inherent to any kind of image, this is possible.

Transfer Learning with Google's InceptionV3

InceptionV3 is one of the state-of-the-art convolutional neural networks from Google. It's an advanced version of GoogLeNet that won the ImageNetILSVRC-2014 competition with its out-of-the-box convolutional neural network architecture. The details of the network are documented in the paper titled "Rethinking the Inception Architecture for Computer Vision" by Christian Szegedy and his collaborators. The paper can be located at https://arxiv.org/abs/1512.00567. The core element of GoogLeNet and its modified versions is the introduction of an inception module to do the convolution and pooling. In traditional convolutional neural networks, after a convolution layer we either perform another convolution or max pooling, whereas in the inception module a series of convolutions and max pooling is done in parallel at each layer, and later the feature maps are merged. Also, in each layer convolution is not done with one kernel-filter size but rather with multiple kernel-filter sizes. An inception module is presented in Figure 3-34 below. As we can see, there is a series of convolutions in parallel along with max pooling, and finally all the output feature maps merge in the gilter concatenation block. 1×1 convolutions do a dimensionality reduction and perform an operation like average pooling. For example, let's say we have an input volume of 224×224×160, with 160 being the number of feature maps. A convolution with a 1×1×20 filter kernel will create an output volume of 224×224×20.

This kind of network works well since the different kernel sizes extract feature information at different granular levels based on the size of the filter's receptive field. Receptive fields of 3×3 will extract much more granular information than will a 5×5 receptive field.

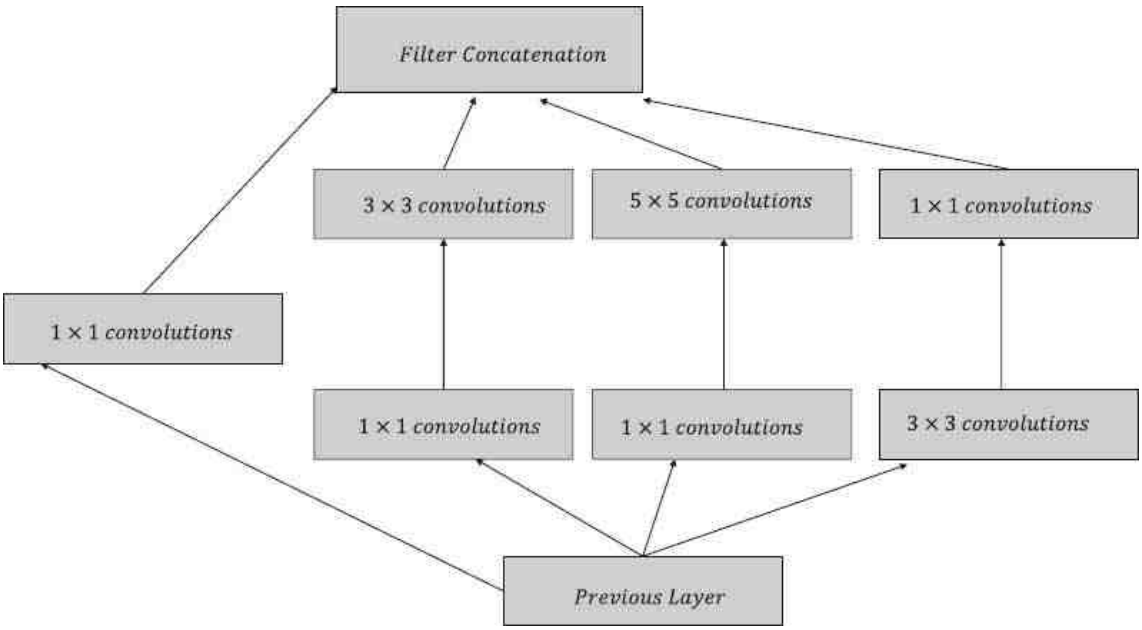


Figure 3-34: Inception module

Google's TensorFlow provides a pre-trained model that is trained on the ImageNet data. It can be used for transfer learning. We use the pre-trained model from Google and retrain it on a set of cat versus dog images extracted from https://www.kaggle.eom/c/dogs-vs-cats/data. The `train.zip` dataset contains 25,000 images, with 12,500 images each for cats and dogs.

The pre-trained model can be found in the TensorFlow GitHub Examples folder. Listing 3-8 shows the steps one needs to follow in order to execute and use the model for transfer-learning purposes. Clone the TensorFlow GitHub repository, since the model sits within it in the Examples folder. Once done, we are all set to enter the cloned TensorFlow folder and execute the commands in Listing 3-8.

Listing 3-8

```
#Step1 - Download the following dataset and un-tar it since it would be used for building
the retrainer for transfer learning


          cd ~curl -O http : //download.tensorflow.org /example _images / flower _photos.tgz

                        tar xzf flower _photos.tgz

# Step2 - Enter the cloned tensorflow folder and build the image retrainer by executing the
following command

                    bazel build tensorflow/examples /image _retraining : retrain

# Step3 - Once the model is built with the preceding command we are all set to retrain the
model based on our input.
# In this case we will test with the Cat vs Dog dataset extracted from Kaggle. The dataset
has two classes. For using the dataset on this model the images pertaining to the classes
have to be kept in different folders. The Cat and Dog sub-folders were created within an
animals folder. Next we retrain the model using the pre-trained InceptionV3 model. All the
layers and weights of the pre-trained model would be transferred to the re-trained model.
Only the output layer would be modified to have two classes instead of the 1000 on which
the pre-trained model is built. In the re-training only the weights from the last fully
connected layer to the new output layer of two classes would be learned in the retraining.
The following command does the retraining:


    bazel -bin/tensorflow/examples/image _retraining/retrain--image_dir ~/Downloads/animals

-- Output Log from Model retraining in the Final Few Steps of Learning --
2017-07-05 09:28:26.133994: Step 3750: Cross entropy = 0.006824
2017-07-05 09:28:26.173795: Step 3750: Validation accuracy = 100.0% (N=100)
2017-07-05 09:28:26.616457: Step 3760: Train accuracy = 99.0%
2017-07-05 09:28:26.616500: Step 3760: Cross entropy = 0.017717
2017-07-05 09:28:26.656621: Step 3760: Validation accuracy = 100.0% (N=100)
2017-07-05 09:28:27.055419: Step 3770: Train accuracy = 100.0%
```

```
2017-07-05 09:28:27.055461: Step 3770: Cross entropy = 0.004180
2017-07-05 09:28:27.094449: Step 3770: Validation accuracy = 99.0% (N=100)
2017-07-05 09:28:27.495100: Step 3780: Train accuracy = 100.0%
2017-07-05 09:28:27.495154: Step 3780: Cross entropy = 0.014055
2017-07-05 09:28:27.540385: Step 3780: Validation accuracy = 99.0% (N=100)
2017-07-05 09:28:27.953271: Step 3790: Train accuracy = 99.0%
2017-07-05 09:28:27.953315: Step 3790: Cross entropy = 0.029298
2017-07-05 09:28:27.992974: Step 3790: Validation accuracy = 100.0% (N=100)
2017-07-05 09:28:28.393039: Step 3800: Train accuracy = 98.0%
2017-07-05 09:28:28.393083: Step 3800: Cross entropy = 0.039568
2017-07-05 09:28:28.432261: Step 3800: Validation accuracy = 99.0% (N=100)
2017-07-05 09:28:28.830621: Step 3810: Train accuracy = 98.0%
2017-07-05 09:28:28.830664: Step 3810: Cross entropy = 0.032378
2017-07-05 09:28:28.870126: Step 3810: Validation accuracy = 100.0% (N=100)
2017-07-05 09:28:29.265780: Step 3820: Train accuracy = 100.0%
2017-07-05 09:28:29.265823: Step 3820: Cross entropy = 0.004463
2017-07-05 09:28:29.304641: Step 3820: Validation accuracy = 98.0% (N=100)
2017-07-05 09:28:29.700730: Step 3830: Train accuracy = 100.0%
2017-07-05 09:28:29.700774: Step 3830: Cross entropy = 0.010076
2017-07-05 09:28:29.741322: Step 3830: Validation accuracy = 100.0% (N=100)
2017-07-05 09:28:30.139802: Step 3840: Train accuracy = 99.0%
2017-07-05 09:28:30.139847: Step 3840: Cross entropy = 0.034331
2017-07-05 09:28:30.179052: Step 3840: Validation accuracy = 100.0% (N=100)
2017-07-05 09:28:30.575682: Step 3850: Train accuracy = 97.0%
2017-07-05 09:28:30.575727: Step 3850: Cross entropy = 0.032292
2017-07-05 09:28:30.615107: Step 3850: Validation accuracy = 100.0% (N=100)
2017-07-05 09:28:31.036590: Step 3860: Train accuracy = 100.0%
2017-07-05 09:28:31.036635: Step 3860: Cross entropy = 0.005654
2017-07-05 09:28:31.076715: Step 3860: Validation accuracy = 99.0% (N=100)
2017-07-05 09:28:31.489839: Step 3870: Train accuracy = 99.0%
2017-07-05 09:28:31.489885: Step 3870: Cross entropy = 0.047375
2017-07-05 09:28:31.531109: Step 3870: Validation accuracy = 99.0% (N=100)
2017-07-05 09:28:31.931939: Step 3880: Train accuracy = 99.0%
2017-07-05 09:28:31.931983: Step 3880: Cross entropy = 0.021294
2017-07-05 09:28:31.972032: Step 3880: Validation accuracy = 98.0% (N=100)
2017-07-05 09:28:32.375811: Step 3890: Train accuracy = 100.0%
2017-07-05 09:28:32.375855: Step 3890: Cross entropy = 0.007524
2017-07-05 09:28:32.415831: Step 3890: Validation accuracy = 99.0% (N=100)
2017-07-05 09:28:32.815560: Step 3900: Train accuracy = 100.0%
2017-07-05 09:28:32.815604: Step 3900: Cross entropy = 0.005150
2017-07-05 09:28:32.855788: Step 3900: Validation accuracy = 99.0% (N=100)
2017-07-05 09:28:33.276503: Step 3910: Train accuracy = 99.0%
2017-07-05 09:28:33.276547: Step 3910: Cross entropy = 0.033086
2017-07-05 09:28:33.316980: Step 3910: Validation accuracy = 98.0% (N=100)
2017-07-05 09:28:33.711042: Step 3920: Train accuracy = 100.0%
2017-07-05 09:28:33.711085: Step 3920: Cross entropy = 0.004519
2017-07-05 09:28:33.750476: Step 3920: Validation accuracy = 99.0% (N=100)
2017-07-05 09:28:34.147856: Step 3930: Train accuracy = 100.0%
2017-07-05 09:28:34.147901: Step 3930: Cross entropy = 0.005670
2017-07-05 09:28:34.191036: Step 3930: Validation accuracy = 99.0% (N=100)
2017-07-05 09:28:34.592015: Step 3940: Train accuracy = 99.0%
2017-07-05 09:28:34.592059: Step 3940: Cross entropy = 0.019866
2017-07-05 09:28:34.632025: Step 3940: Validation accuracy = 98.0% (N=100)
2017-07-05 09:28:35.054357: Step 3950: Train accuracy = 100.0%
2017-07-05 09:28:35.054409: Step 3950: Cross entropy = 0.004421
2017-07-05 09:28:35.100622: Step 3950: Validation accuracy = 96.0% (N=100)
2017-07-05 09:28:35.504866: Step 3960: Train accuracy = 100.0%
2017-07-05 09:28:35.504910: Step 3960: Cross entropy = 0.009696
2017-07-05 09:28:35.544595: Step 3960: Validation accuracy = 99.0% (N=100)
2017-07-05 09:28:35.940758: Step 3970: Train accuracy = 99.0%
2017-07-05 09:28:35.940802: Step 3970: Cross entropy = 0.013898
2017-07-05 09:28:35.982500: Step 3970: Validation accuracy = 100.0% (N=100)
2017-07-05 09:28:36.381933: Step 3980: Train accuracy = 99.0%
2017-07-05 09:28:36.381975: Step 3980: Cross entropy = 0.022074
2017-07-05 09:28:36.422327: Step 3980: Validation accuracy = 100.0% (N=100)
2017-07-05 09:28:36.826422: Step 3990: Train accuracy = 100.0%
2017-07-05 09:28:36.826464: Step 3990: Cross entropy = 0.009017
2017-07-05 09:28:36.866917: Step 3990: Validation accuracy = 99.0% (N=100)
2017-07-05 09:28:37.222010: Step 3999: Train accuracy = 99.0%
2017-07-05 09:28:37.222055: Step 3999: Cross entropy = 0.031987
2017-07-05 09:28:37.261577: Step 3999: Validation accuracy = 99.0% (N=100)
Final test accuracy = 99.2% (N=2593)
Converted 2 variables to const ops.
```

We can see from the Listing 3-8 output that we achieve a testing accuracy of 99.2 percent on the cats versus dogs classification problem, reusing the pre-trained InceptionV3 model by just training the weights to the new output layer. This is the power of transfer learning when used within the right context.

Transfer Learning with Pre-trained VGG16

In this section, we will perform transfer learning by using a VGG16 network pre-trained on a thousand classes of ImageNet to classify the cats versus dogs dataset from Kaggle. The link to the dataset is https://www.kaggle.com/c/dogs-vs-cats/data. First, we would import the VGG16 model from *TensorFlow Slim* and then load the pre-trained weights in the VGG16 network. The weights are from a VGG16 trained on the thousand classes of the ImageNet dataset. Since for our problem we have only two classes, we will take the output from the last fully connected layer and combine it with a new set of weights, leading to the output layer will one neuron to do a binary classification of the cats and dogs dataset from Kaggle. The idea is to use the pre-trained weights to generate features, and finally we learn just one set of weights at the end, leading to the output. In this way, we learn a relatively small set of weights and can afford to train the model on a smaller amount of data. Please find the detailed implementation in Listing 3-9.

Listing 3-9: Transfer Learning with Pre-trained VGG16

```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
from scipy.misc import imresize
from sklearn.model_selection import train_test_split
import cv2
from nets import vgg
from preprocessing import vgg_preprocessing
from mlxtend.preprocessing import shuffle_arrays_unison
sys.path.append("/home/santanu/models/slim")

%matplotlib inline

batch_size = 32
width = 224
height = 224
cat_train = '/home/santanu/CatvsDog/train/cat/'
dog_train = '/home/santanu/CatvsDog/train/dog/'
checkpoints_dir = '/home/santanu/checkpoints'
slim = tf.contrib.slim

all_images = os.listdir(cat_train) + os.listdir(dog_train)
train_images, validation_images = train_test_split(all_images, train_size=0.8, test_
size=0.2)

MEAN_VALUE = np.array([103.939, 116.779, 123.68])
################################################
# Logic to read the images and also do mean correction
################################################

def image_preprocess(img_path,width,height):
    img = cv2.imread(img_path)
    img = imresize(img,(width,height))
    img = img - MEAN_VALUE
    return(img)

################################################
# Create generator for image batches so that only the batch is in memory
################################################

def data_gen_small(images, batch_size, width,height):
        while True:
            ix = np.random.choice(np.arange(len(images)), batch_size)
            imgs = []
            labels = []
            for i in ix:
                data_dir = ' '
                # images
                if images[i].split('.')[0] == 'cat':
                    labels.append(1)
                    data_dir = cat_train
                else:
                    if images[i].split('.')[0] == 'dog':
                        labels.append(0)
                        data_dir = dog_train
                #print 'data_dir',data_dir
                img_path = data_dir + images[i]
                array_img = image_preprocess(img_path,width,height)
```

```
                imgs.append(array_img)

            imgs = np.array(imgs)
            labels = np.array(labels)
            labels = np.reshape(labels,(batch_size,1))
            yield imgs,labels
#####################################################
## Defining the generators for training and validation batches
#####################################################
train_gen = data_gen_small(train_images,batch_size,width,height)
val_gen = data_gen_small(validation_images,batch_size, width,height)

with tf.Graph().as_default():

    x = tf.placeholder(tf.float32,[None,width,height,3])
    y = tf.placeholder(tf.float32,[None,1])

#############################################
## Load the VGG16 model from slim extract the fully connected layer
## before the final output layer
      ############################################
    with slim.arg_scope(vgg.vgg_arg_scope()):
        logits, end_points = vgg.vgg_16(x,
                                num_classes=1000,
                                is_training=False)
              fc_7 = end_points['vgg_16/fc7']
      ############################################
## Define the only set of weights that we will learn W1 and b1
############################################
    Wn =tf.Variable(tf.random_normal([4096,1],mean=0.0,stddev=0.02),name='Wn')
    b = tf.Variable(tf.random_normal([1],mean=0.0,stddev=0.02),name='b')

   ############################################
   ##  Reshape the fully connected layer fc_7 and define
   ## the logits and probability
   ############################################
    fc_7 = tf.reshape(fc_7, [-1,W1.get_shape().as_list()[0]])
    logitx = tf.nn.bias_add(tf.matmul(fc_7,W1),b1)
    probx = tf.nn.sigmoid(logitx)

    ###################################################
    # Define Cost and Optimizer
    # Only we wish to learn the weights Wn and b and hence included them in var_list
   ###################################################

    cost = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=logitx,labels=y))
    optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost,var_
    list=[W1,b1])

    ###################################################
    # Loading the pre-trained weights for VGG16
    ###################################################
    init_fn = slim.assign_from_checkpoint_fn(
        os.path.join(checkpoints_dir, 'vgg_16.ckpt'),
        slim.get_model_variables('vgg_16'))
    ###################################################
    # Running the optimization for only 50 batches of size 32
    ###################################################
    with tf.Session() as sess:
        init_op = tf.global_variables_initializer()
        sess.run(init_op)
        # Load weights
        init_fn(sess)
        for i in xrange(1):
            for j in xrange(50):
                batch_x,batch_y = next(train_gen)
                #val_x,val_y = next(val_gen)
                sess.run(optimizer,feed_dict={x:batch_x,y:batch_y})
                cost_train = sess.run(cost,feed_dict={x:batch_x,y:batch_y})
                cost_val = sess.run(cost,feed_dict={x:val_x,y:val_y})
                prob_out = sess.run(probx,feed_dict={x:val_x,y:val_y})
                print "Training Cost",cost_train,"Validation Cost",cost_val
        out_val = (prob_out > 0.5)*1
        print 'accuracy', np.sum(out_val == val_y)*100/float(len(val_y))
        plt.imshow(val_x[1] + MEAN_VALUE)
        print "Actual Class:",class_dict[val_y[1][0]]
        print "Predicted Class:",class_dict[out_val[1][0]]
        plt.imshow(val_x[3] + MEAN_VALUE)
        print "Actual Class:",class_dict[val_y[2][0]]
```

```
        print "Predicted Class:",class_dict[out_val[2][0]]

--output--

Training Cost 0.12381 Validation Cost 0.398074
Training Cost 0.160159 Validation Cost 0.118745
Training Cost 0.196818 Validation Cost 0.237163
Training Cost 0.0502732 Validation Cost 0.183091
Training Cost 0.00245218 Validation Cost 0.129029
Training Cost 0.0913893 Validation Cost 0.104865
Training Cost 0.155342 Validation Cost 0.050149
Training Cost 0.00783684 Validation Cost 0.0179586
Training Cost 0.0533897 Validation Cost 0.00746072
Training Cost 0.0112999 Validation Cost 0.00399635
Training Cost 0.0126569 Validation Cost 0.00537223
Training Cost 0.315704 Validation Cost 0.00140141
Training Cost 0.222557 Validation Cost 0.00225646
Training Cost 0.00431023 Validation Cost 0.00342855
Training Cost 0.0266347 Validation Cost 0.00358525
Training Cost 0.0939392 Validation Cost 0.00183608
Training Cost 0.00192089 Validation Cost 0.00105589
Training Cost 0.101151 Validation Cost 0.00049641
Training Cost 0.139303 Validation Cost 0.000168802
Training Cost 0.777244 Validation Cost 0.000357215
Training Cost 2.20503e-06 Validation Cost 0.00628659
Training Cost 0.00145492 Validation Cost 0.0483692
Training Cost 0.0259771 Validation Cost 0.102233
Training Cost 0.278693 Validation Cost 0.11214
Training Cost 0.0387182 Validation Cost 0.0736753
Training Cost 9.19127e-05 Validation Cost 0.0431452
Training Cost 1.19147 Validation Cost 0.0102272
Training Cost 0.302676 Validation Cost 0.0036657
Training Cost 2.22961e-07 Validation Cost 0.00135369
Training Cost 8.65403e-05 Validation Cost 0.000532816
Training Cost 0.00838018 Validation Cost 0.00029422
Training Cost 0.0604016 Validation Cost 0.000262787
Training Cost 0.648359 Validation Cost 0.000327267
Training Cost 0.00821085 Validation Cost 0.000334495
Training Cost 0.178719 Validation Cost 0.000776928
Training Cost 0.362365 Validation Cost 0.000317593
Training Cost 0.000330557 Validation Cost 0.000139824
Training Cost 0.0879459 Validation Cost 5.76907e-05
Training Cost 0.0881795 Validation Cost 1.21865e-05
Training Cost 1.11339 Validation Cost 1.9081e-05
Training Cost 0.000440863 Validation Cost 3.60468e-05
Training Cost 0.00730334 Validation Cost 6.98846e-05
Training Cost 3.65983e-05 Validation Cost 0.000141883
Training Cost 0.296884 Validation Cost 0.000196292
Training Cost 2.10772e-06 Validation Cost 0.000269568
Training Cost 0.179874 Validation Cost 0.000185331
Training Cost 0.380936 Validation Cost 9.48413e-05
Training Cost 0.0146583 Validation Cost 3.80007e-05
Training Cost 0.387566 Validation Cost 5.26306e-05
Training Cost 7.43922e-06 Validation Cost 7.17469e-05
accuracy 100.0
```

Figure 3-35: Validation set images and their actual versus predicted classes

We see that the validation accuracy is 100 percent after training the model on only 50 batches of a moderate size of 32 per

batch. The accuracy and the cost are a little noisy since the batch sizes are small, but in general the validation cost is going down while the validation accuracy is on the rise. In Figure 3-35 a couple of validation-set images have been plotted along with their actual and predicted classes to illustrate the correctness of the predictions. Hence, proper utilization of transfer learning helps us to reuse feature detectors learned for one problem in solving a new problem. Transfer learning greatly reduces the number of parameters that need to be learned and hence reduces the computational burden on the network. Also, the training data-size constraints are reduced since fewer parameters require less data in order to be trained.

Summary

In this chapter, we learned about the convolution operation and how it is used to construct a convolutional neural network. Also, we learned about the various key components of CNN and the backpropagation method of training the convolutional layers and the pooling layers. We discussed two critical concepts of CNN responsible for its success in image processing—the equivariance property provided by convolution and the translation invariance provided by the pooling operation. Further, we discussed several established CNN architectures and how to perform transfer learning using the pre-trained versions of these CNNs. In the next chapter, we will discuss recurrent neural networks and their variants in the realm of natural language processing.