Alma Mater Studiorum · Università di Bologna

DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA
Laurea Magistrale di Informatica

Computer Vision

Docente:
Giuseppe Lisanti

Studente:
Massimo Rondelli

Anno Accademico 2023/2024

Contents

Chapter 1

Image Formation Process

The image creation is done using an imaging device where its goal is to get the light reflected by 3D objects and create a 2D representation of the scene, which is the image itself. In Computer Vision we try to invert this process. We try to infer knowledge on the objects from one or more digital images. The image formation and acquisition process is done in three main points:

- The geometric relationship between scene points and image points
- The radiometric relationship between the brightness of image points and the light emitted by scene points
- The image digitization process

1.1 Pinhole camera Model

The pinhole camera is the simplest imaging device. The pinhole camera model describes the mathematical relationship between the coordinates of a point in three-dimensional space and its projection onto the image plane of an ideal pinhole camera, where the camera aperture is described as a point and no lenses are used to focus light. It can only be used as a first order approximation of the mapping from a 3D scene to a 2D scene.

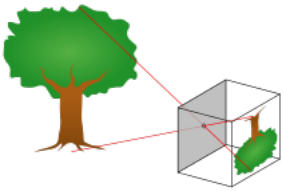


Figure 1.1: A diagram of a pinhole camera.

1.2 Perspective Projection

Let's see now the geometry behind the pinhole model. Focus on Figure 1.2. The figure

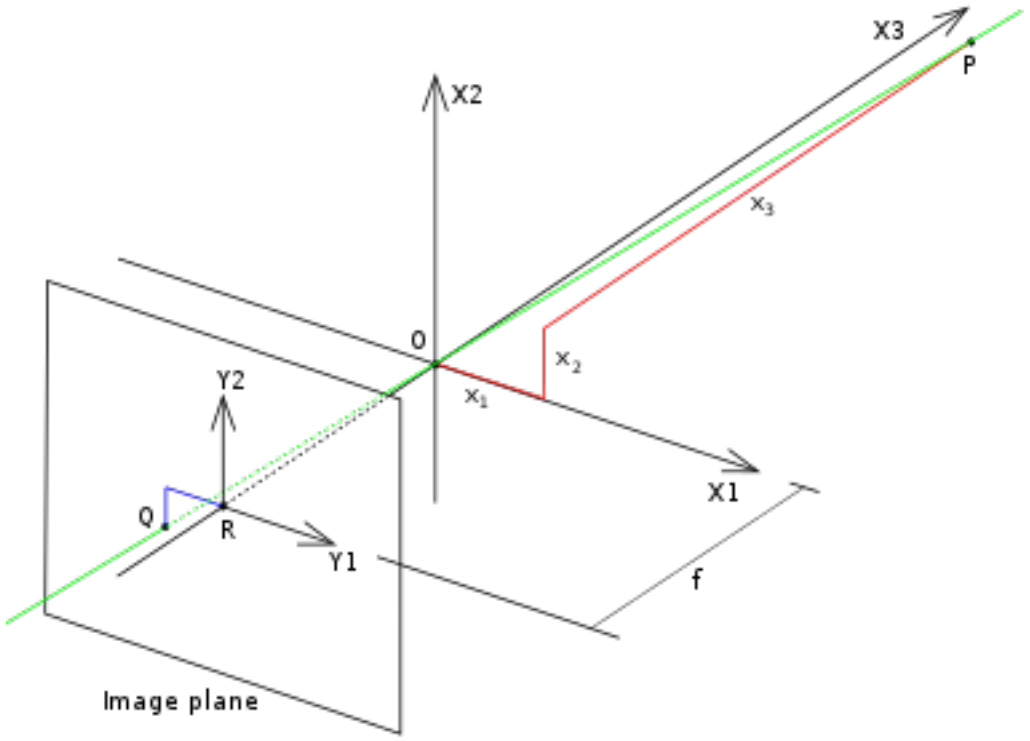


Figure 1.2: The geometry of a pinhole camera. Note: the x1x2x3 coordinate system in the figure is left-handed, that is the direction of the OZ axis is in reverse to the system the reader may be used to.

contains the following basic objects:

- A 3D orthogonal coordinate system with its origin at $\boldsymbol{O}$. This is also where the camera aperture is located. The three axes of the coordinate system are referred to as $X1$, $X2$, $X3$. Axis $X2$ is pointing in the viewing direction of the camera and is referred to as the optical axis. The plane which is spanned by axes $X1$ and $X2$ is the front side of the camera, or *principale plane*.
- An image plane, where the 3D world is projected through the aperture of the camera. The image plane is parallel to axes $X1$ and $X2$ and is located at distance $\boldsymbol{f}$ from the origin $\boldsymbol{O}$ in the negative direction of the $X3$ axis, where f is the focal length f the pinhole camera.
- A point $\boldsymbol{R}$ at the intersection of the optical axis and the image plane. This point is referred to as *image center*.

- A point $\boldsymbol{P}$ somewhere in the world at coordinate (x_1, x_2, x_3) relative to the axes $X1$, $X2$ and $X3$.
- The projection line of point $\boldsymbol{P}$ into the camera. This is the green line which passes through point $\boldsymbol{P}$ and the point $\boldsymbol{O}$.
- The projection of point $\boldsymbol{P}$ onto the image plane, denoted $\boldsymbol{Q}$. This point is given by the intersection of the projection line (green) and the image plane.
- There is also a 2D coordinate system in the image plane, with origin at $\boldsymbol{R}$ and with axes $Y1$ and $Y2$ which are parallel to $X1$ and $X2$, respectively. The coordinates of point Q relative to this coordinate system is (y_1, y_2).

Next, we want to understand how the coordiantes (y_1, y_2) of point $\boldsymbol{Q}$ depend on the coordiantes (x_1, x_2, x_3) of point $\boldsymbol{P}$. This can be done with the help of the following Figure 1.3, which shows the same scene as the previous one but from above, looing down in the negative direction of the $X2$ axis.

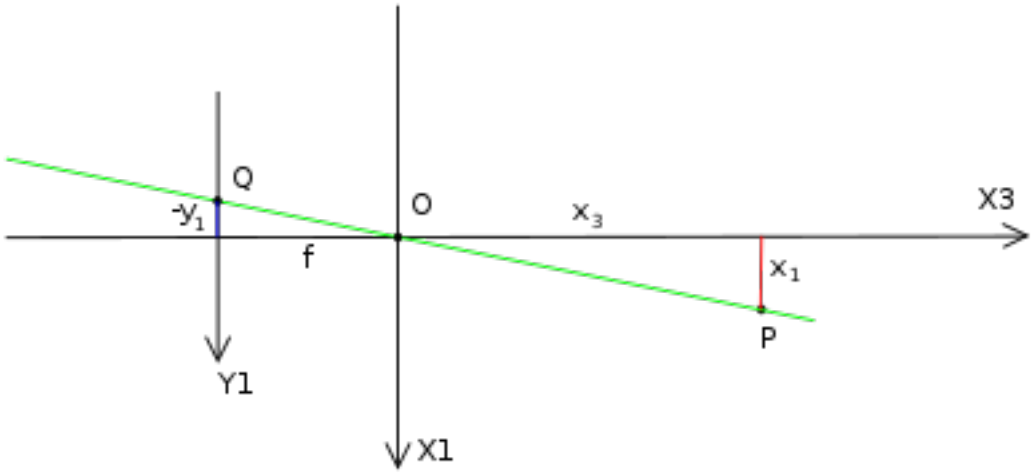


Figure 1.3: The geometry of a pinhole camera as seen from the X2 axis

In this Figure 1.3, we can see two similar triangles, both having parts of the projection line (green line) as their hypotenuses. The catheti of the left traingle are $-y_1$ and f and the catheti of the right tiangle are x_1 and x_3. Since the two traingles are similar it follows that:

$$\frac{-y_1}{f} = \frac{x_1}{x_2} \text{ or } y_1 = -\frac{x_1}{x_3}$$

A similar investigation, looking at the negative direction of the $X1$ axis gives

$$\frac{-y_2}{f} = \frac{x_2}{x_3} \text{ or } y_2 = -\frac{x_2}{x_3}$$

This can be summarized as

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = -\frac{f}{x_3}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

which is the expression that describes the relation between the 3D coordinates (x_1, x_2, x_3) of point $\boldsymbol{P}$ and its cooridnates (y_1, y_2) given by point $\boldsymbol{Q}$ in the image plane.

It is important to remember that the image formation process deals with mapping a 3D space onto a 2D space, which means there is loss of information. A given a scene point is mapped into a unique image point. Recovering the 3D structure of a scene from a single image is an ill-posed problem, where the solution is not unique. For an image point we can only state that its corresponding scene point lays on a line but cannot disambiguate a specific 3D point along such a line *(i.e. we know nothing about the distance to the camera).*
How can we solve this problem? Let's see know stereo images.

1.3 Stereo Images

Computer stereo vision is the extraction of 3D information from digital images. By comparing information about a scene from two vantage points, 3D information can be extracted by examining the relative positions of objects in the two panels. In traditional stereo vision, two cameras, displaced horizontally from one another, are used to obtain two differing views on a scene. By comparing these two images, the relative depth information can be obtained in the form of a disparity map, which encodes the difference in horizontal coordinates of corresponding image points. The correspondence problem refers to the problem of discovering which parts of one image corresponde to which parts of another image, where differences are due to movement of the camera, the elapse of time, and/or movement of object in the photos. So, given correspondences, 3D information can be recovered easly by triangulation. This is possible with at least two images, which means due stereo vision.
Let's see how stereo geometry works. Consider the geometry of a standard, "narrow-baseline" stereo rig. If we send out two rays from a 3D scene point to the two camera centers, a triangle is constructed in the process. Our goal is to "reverse" this 3D to 2D projection. The x-coordinate is the only difference, one is translated by the baseline B. shift in 1-dimension, by one number. 2 points will lie on the same row in both images. we know where to look.

$$Z = f\frac{B}{d}$$

To construct a simple model of stereo vision, we have two cameras whose optic axes are parallel. Each camera points down the Z-axis. The Figure 1.4 shows how it works:

Now, consider just the plane spanned by the x and z-axes, with a constant y value, as shown in Figure 1.5

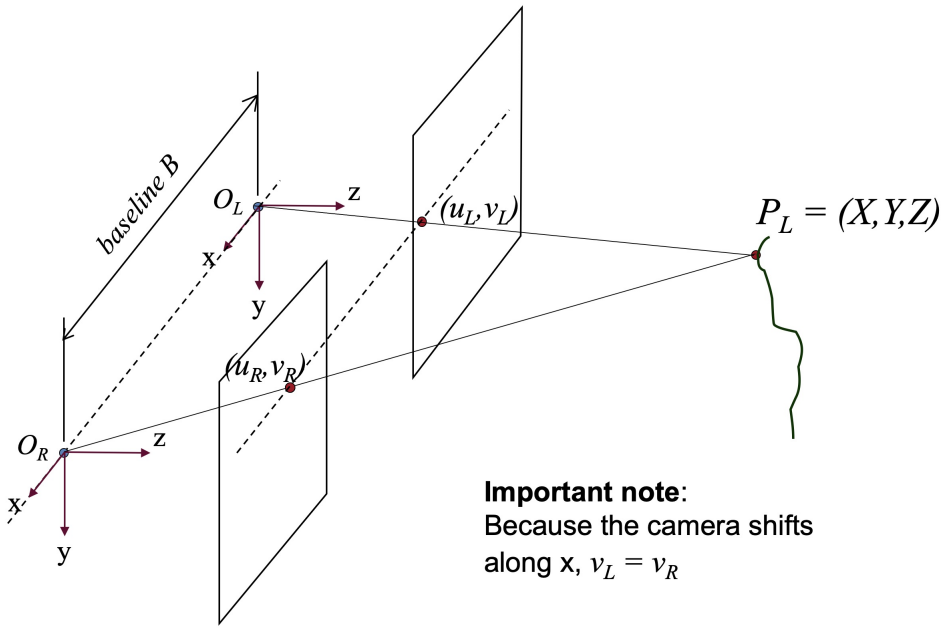


Figure 1.4: Two cameras with optical centers O_L and O_R are separated by a baseline B. The z-axis extends towards the world, away from the camera.

In the Figure 1.5, the world point $P = (x, z)$ is projected into the left image as p_l and into the right image as p_r. By defining right triangles we can find two similar triangles: with vertices at $(0,0) - (0,z) - (x,z)$ and $(0,0) - (0,f) - (f,u_l)$. Since they share the same angle θ, then $\tan(\theta) = \frac{\text{opposite}}{\text{adjacent}}$ for both, meaning:

$$\frac{z}{f} = \frac{x}{u_l}$$

We notice another pair of similar triangles $(b,0) - (b,z) - (x,z)$ and $(b,0) - (b,f) - (b+u_r, f)$, which by the same logic gives us:

$$\frac{z}{f} = \frac{x-b}{u_r}$$

We'll derive a closed form expression for depth in terms of disparity. We already know that

$$\frac{z}{f} = \frac{x}{u_l}$$

Multiply both sides by f, and we get an expression for our depth from the observer:

$$z = f(\frac{x}{x_l})$$

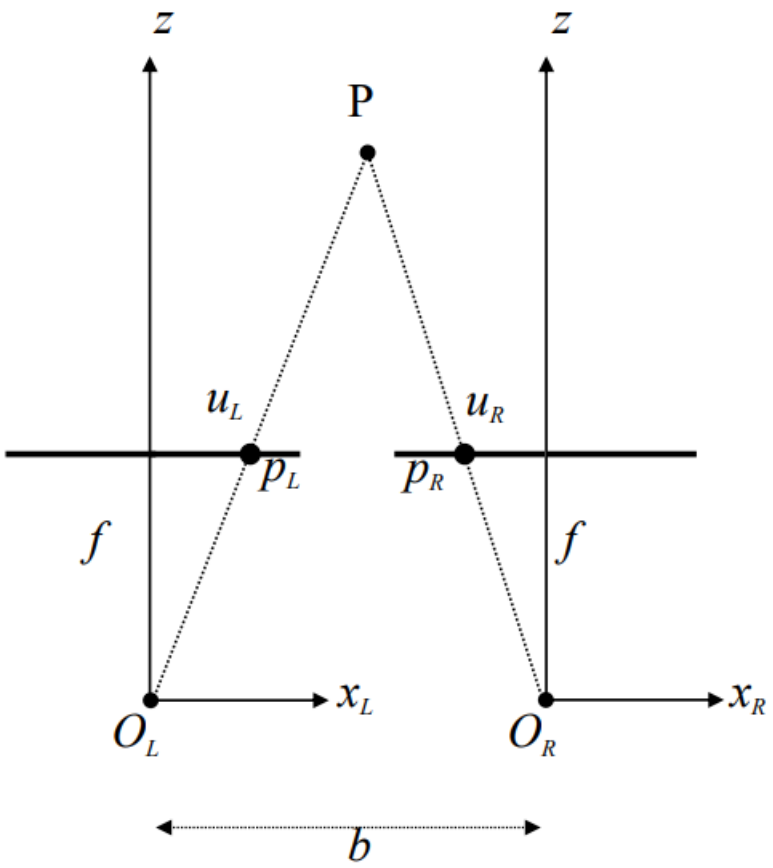


Figure 1.5: Two cameras L and R are separated by a baseline b. Here the Y-axis is perpendicular to the page. f is our (horizontal) focal length.

We now want to find an expression for $\frac{x}{u_l}$ in terms of $u_l - u_r$, the disparity between the two images. Making some calculations we can say that:

$$z = f(\frac{x}{u_l}) = f(\frac{b}{u_l - u_r})$$

1.4 Epipolar Geometry

The Epipolar Geometry is the geometry of stereo vision. When two cameras view a 3D scene from two distinct positions, there are a number of geometric relations between the 3D points and their projections onto the 2D images that lead to constraints between the image points. These relations are derived based on the assumption that the cameras can be approximated by the pinhole camera model. In the Figure 1.6 illustrate two pinhole cameras looking at point X. O_L and O_R represent the center of symmetry of the two

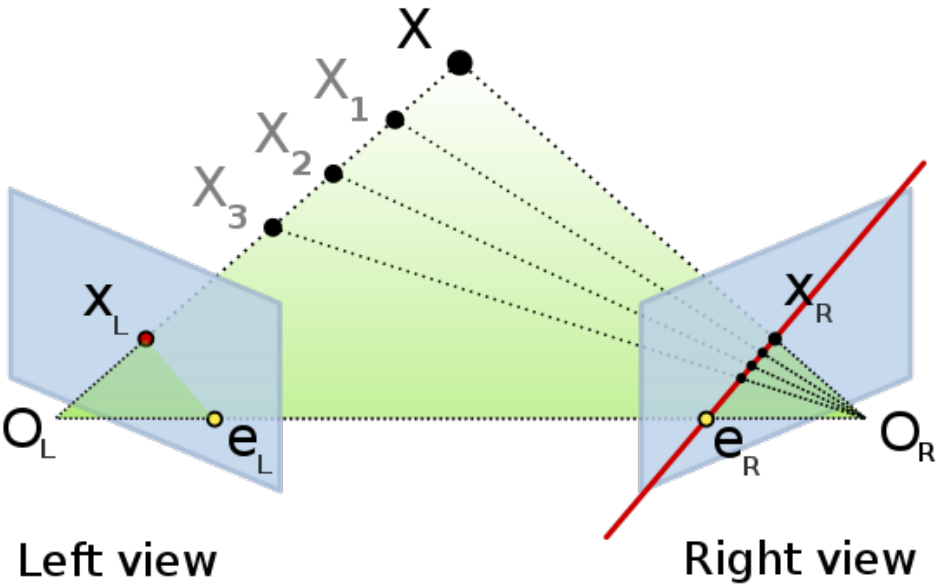


Figure 1.6: Epipolar geometry

cameras lenses. X represents the point of interest in both cameras. Points X_L and X_R are the projections of point X onto the image planes. Each camera captures a 2D image of the 3D world. This conversion from 3D to 2D is referred to Perspective Projection, Section 1.2.

Since the optical centers of the cameras lenses are distinct, each center projects onto a distinct point into the other camera's image plane. These two image points, denoted by e_L and e_R are called *epipolar points.* Both points e_L and e_R in their respective image planes and both optical centers O_L and O_R lie on a single 3D line.

The line $O_L - X$ is seen by the left camera as a point because it is directly in line with the camera's lens optical center. However, the right camera sees this line as a line in its image plane. That line $(e_R - e_R)$ in the right camera is called an *epipolar line.* Symmetrically, the line $O_R - X$ is seen by the right camera as a point and is seen as epipolar line $(e_L - e_L)$ by the left camera. An epipolar line is a function of the position of point X in the 3D space. Since the 3D line $O_L - X$ passes through the optical center of the lens O_L, the corresponding epipolar line in the right image must pass through the epipole e_R (and correspondingly for epipolar lines in the left image).

Assume the projection point X_L is known, and the epipolar line $(e_R - e_R)$ is known and the point X projects into the right image, on a point X_R which must lie on this

particular epipolar line. This means that for each point observed in one image the same point must be observed in the other image on a known epipolar line. This provides an epipolar constraint: the projection of X on the right camera plane X_R must be contained in the $(e_R - e_R)$ epipolar line. All points X, e.g. $X1, X2, X3$ on the $O_L - X_L$ line will verify that constraint. It means that it is possible to test if two points correspond to the same 3D point.

If the points X_L and X_R are known, their projection lines are also known. If the two image points correspond to the same 3D point X the projection lines must intersect precisely at X. This means that X can be calculated from the coordinates of the two images points. This process is called triangulation.

The epipolar geometry is simplified if the two camera image planes coincide. In this case, the epipolar lines also coincide ($e_L - X_L = e_R - X_R$). Furthermore, the epipolar lines are parallel to the line O_L–O_R between the centers of projection. This means that for each point in one image, its corresponding point in the other image can be found by looking only along a horizontal line. If the cameras cannot be positioned in this way, the image coordinates from the camers may be trasformed to emulate having a common image plane. This process is called image rectification, Section 1.5.

1.5 Image Rectification

Since it is almost impossible to build a stereo rig which is perfectly aligned horizontally and searching through oblique epipolar lines is awkward and computationally less efficient, what can we do? What people do in practice is to convert epipolar geometry to standard geometry, also called rectification or warping. Image rectification is a transformation process used to project images onto a common image plane.

The rectification process can be implemented by projecting the original pictures onto the new image plane. With an appropriate choice of coordinate system, the rectified epipolar lines are scanlines of the new images, and they are also parallel to the baseline. In the case of rectified images, the informal notizion of disparity takes a concrete meaning. The disparity is the difference between the horizontal coordinates in the left and right images (horizontal displacement). So, given two points p and p^1 located on the same scanline of the left and right images, with coordinates (x, y) and (x^1, y), the disparity is defined as the difference $d = x^1 - x$. We assume that image coordinates are normalized. If B denotes the distance between the optical centers, also called the basilne in the context, the depth of P in the normalized coordinate system attached to the first camera is $Z = \frac{-B}{d}$. In particular, the coordinate vector of the point P in the frame attached to the first camera is $P = -(\frac{B}{d})p$, where $p = (x, y, 1)^T$ is the vector of normalized image coordinates of p. This provides yet another reconstruction method for rectified stereo pairs.

1.6 Image Digitization

Generally speaking, the image plane of a camera consists of a planar sensor which converts the irradiance at any point into an electric quatity, e.g. a voltage. How do we discretise it?

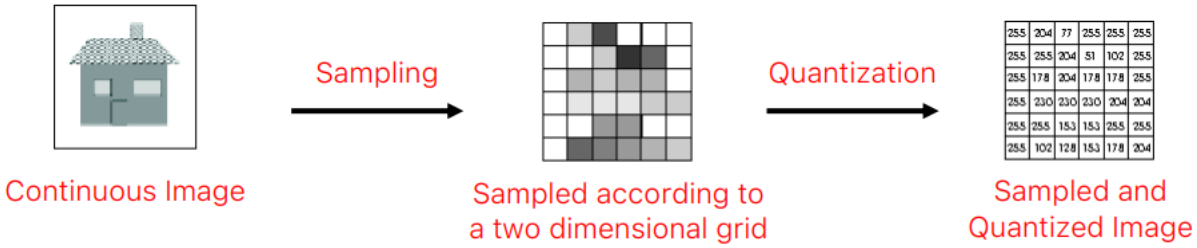


Figure 1.7: Image Digitization

The digital image is produced by the process of digitization. The continuous optical image is sampled, commonly on a rectangular grid, and those sample values are quantizied to produce a rectangular array of integers. That is, the coordinate positinos (n, m) are integers, and the light intensity at a given integer spatial position is represented by a nonnegative integer. Further, random noise is introduced into the resulting data. What we actually process or analyze in the computer, of course, is the digital image. This array of sample values (pixels) taken from the optical image,however, is only a relative of the specimen, and a rather distant one at that. It is the responsibility of the user to ensure that the relevant information about the specimen that is conveyed by the optical image is preserved in the digital image as well.

We have mentioned that digitization (sampling and quantization) is the process that generates a corresponding digital image from an existing optical image. To go the other way, from discrete to continuous, we use the process of interpolation. By interpolating a digital image, we can generate an approximation to the continuous image (analytic function) that corresponds to the original optical image. If all goes well, the continuous function that results from interpolation will be a faithful representation of the optical image.

Chapter 2

Spatial Filtering

Image noise is random variation of brightness or color information in images. There are several types of noise. In general, we can say the noise can be expressed as follows:

$$I_k(p) = \tilde{I} + n_k(p) \text{ with } n_k(p) \text{ i.i.d and } n_k(p) \sim N(0, \sigma)$$

To denoise an image, we can take a mean across time of the noise, expressed like this:

$$O(p) = \frac{1}{N}\sum_{k=1}^{N} I_k(p) = \frac{1}{N}\sum_{k=1}^{N}(\tilde{I}(p) + n_k(p)) = \frac{1}{N}\sum_{k=1}^{N}\tilde{I}(p) + \frac{1}{N}\sum_{k=1}^{N} n_k(p) \cong \tilde{I}(p)$$

This process can be done if we have multiple images across time, but what if we are given a single image? We may compute a mean across neighbouring pixels, i.e. a spatial rather than temporal mean.

2.1 Image Filters

Image filters are image processing operators that compute the new intesity (colour) of a pixel **p**, based on the intesities (colours) of those belonging to a neighbourhood, also called support, of p. We want to introduce some filters that can perform the denoising locally. We have a point p and a supporting windows, $S(p)$. We apply some sort of function on the neighbourhood, which is going to provide for us $o(p)$, which is the output value. It is the new intesity value at position p. This operation not only accomplish denoinsing, but a variety of useful image processing functions. Sharpening is another operation that can be made.

An important sub-class of filters is given by Linear and Translation-Equivariant (LTE) operators. Every time you take an operator and apply an operation on a image pixel,

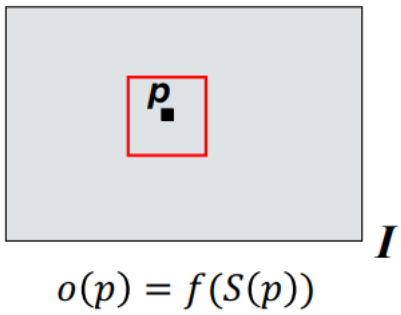


Figure 2.1: Image Filters

the operation you've performed is a convolution if the filter we are using is LTE. Given an input 2D signal $i(x, y)$, a 2D operator, $T\{\cdot\} : o(x, y) = T\{i(x, y)\}$, is said to be linear if and only if:

$$T\{\alpha\ i_1(x, y) + \beta\ i_2(x, y)\} = \alpha\ o_1(x, y) + \beta\ o_2(x, y) \text{ with } o_1 = T\{i_1\} \text{ and } o_2 = T\{i_2\}$$

and α, β are two constants. The operator is said to be *translation-equivariant* if and only if:

$$T\{i(x - x_0, y - y_0)\} = o(x - x_0, y - y_0)$$

Translation-equivariant means that if I shift a little bit the input position of the input signal that I provided to the operator, the output of the operator is going to shift accordingly. It means that the operator can be slided across the image.

We need these two properties to say that the operations between an input signal and an operator that respect LTE is called a *convolution*. We can speak about convolution if and only if the operator is LTE. The main difference between convolution and correlation is that with convolution the kernel is flipped. The mathematically definition of convolution is the following one:

$$o(x, y) = T\{i(x, y)\} = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} i(\alpha, \beta)\ h(x - \alpha, y - \beta)\ d\alpha\ d\beta$$

More in general, we can say that convolution is a mathematical operation on two functions, f and g, that produces a third function $f * g$. The term convolution refers to both the result function and to the process of computing it. It is the integral of the product of the two functions after one is reflected about the y-axis and shifted. Graphically, it expresses how the 'shape' of one function is modified by the other.

2.1.1 Properties of Convolution

The convolution operation is often denoted using the symbol "$*$" , e.g.

$$o(x, y) = i(x, y) * h(x, y)$$

The main properies of convolution are the following ones:

- **Associative Property:** $f * (g * h) = (f * g) * h$
- **Commutative Property:** $f * g = g * f$
- **Distributive Property wrt the Sum:** $f * (g + h) = f * g + f * h$
- **Convolution Commutes with Differentiation:** $(f * g)' = f' * g = f * g'$

2.2 Convolution and Correlation

The correlation of h with reference to i to similar to convolution: the product of the two signals is integrated after traslating h without reflection, so the kernel is not flipped. Hence, if h is an even function ($h(x, y) = h(-x, -y)$), the convolution between i and h, ($i * h = h * i$), is the same as the correlation of h with reference to i. Let's see it mathematically:

$$\begin{aligned} i(x, y) * h(x, y) = h(x, y) * i(x, y) &= \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} i(\alpha, \beta)\, h(x - \alpha, y - \beta)\, d\alpha\, d\beta \\ &= \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} i(\alpha, \beta)\, h(\alpha - x, \beta - y)\, d\alpha\, d\beta \\ &= h(x, y) \circ i(x, y) \end{aligned}$$

It si worth observing that correlation is never commutative, even if h is an even function: To recap, we can say that:

- **Convolution is commutative:** $i * h = h * i$
- **Correlation is not commutative:** $i \circ h \neq h \circ i$
- **If h is an even function:** $i * h = h * i = h \circ i$

Since, we never work in a continuos domain, we work in a discrete domain. We work on image's pixels which are arrays of pixels. So we are going to use the discretized convolution which is the same as the continuos one, but instead of having the integral we have summation:

$$o(x, y) = T\{i(x, y)\} = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} i(\alpha, \beta)\, h(x - \alpha, y - \beta)\, d\alpha\, d\beta$$

$$o(x,y) = T\{i(x,y)\} = \sum_{m=-\infty}^{+\infty} \sum_{n=-\infty}^{+\infty} I(m,n)\, H(i-m, j-n)$$

where $I(i,j)$ and $O(i,j)$ are the discrete 2D input and output signals, respectively, and $H(i,j) = T\{\delta(i,j)\}$ is the kernel of the discrete LTE operator, i.e. the response to the 2D discrete unit impulse. As for continuous signals, discrete convolution consists in summing the product of the two signals where one has been reflected about the origin and translated. A practical implementation, in image processing, both the input image and the kernel are stored into matrixes of given finite sizes, with the image being much larger than the kernel. One would cycle through the kernel:

$$o(x,y) = T\{i(x,y)\} = \sum_{m=-\infty}^{+\infty} \sum_{n=-\infty}^{+\infty} I(m,n)\, H(i-m, j-n)$$

$$\begin{pmatrix} & \vdots & \\ & \vdots & \\ & \vdots & \\ \dots & I(i,y) & \dots \\ & \vdots & \\ & \vdots & \\ & \vdots & \end{pmatrix} * \begin{pmatrix} K(-k,-k) & \dots & K(-k,0) & \dots & K(-k,k) \\ \vdots & & \vdots & & \vdots \\ K(0,-k) & \dots & K(0,0) & \dots & K(0,k) \\ \vdots & & \vdots & & \vdots \\ K(k,-k) & \dots & K(k,0) & \dots & K(k,k) \end{pmatrix}$$

Conceptually, to obtain the output image we need to slide the kernel across the whole input image and compute the convolution at each pixel. It is important that we are not going to overwrite the input matrix, but just compute the convolution across each pixel in the input image.

While we compute convolution or correlation between a pixel and its neighborhood, when a pixel is on the board, how can we compute this operation? Is it possible? We are missing part of the information because we should go out of the image during the operation. In image processing, the common behaviour is to crop the image. It means that we are going to consider only a subspace of the whole image. In this case the output will be smaller. With Deep Learning, the padding technique is used. We add the missing pixels in different ways. This pixels are fake, but they're necessary to compute the convolution on border's pixels. The most common one is *zero-padding*, where you simply enlarge the image putting zeros. In this way you do not lose resolution. The output image will have the same dimension of input image.

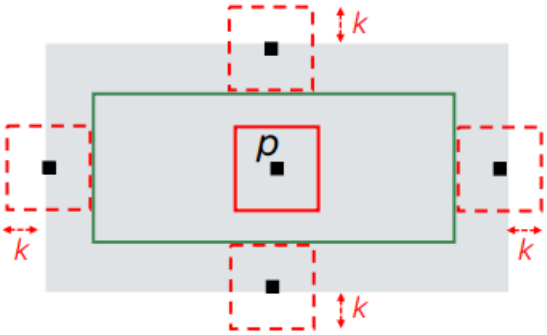


Figure 2.2: Border issue. CROP and PAD

2.3 Mean Filter

In order for an operation to be considered a convolution if and only if Linear and Translation Equivariant (LTE). Lineay means that it respects linearity and Translation means that if we shift the input, also the output shifts. The first LTE operator which is going to compute a convolution is the Mean Filter. When you have an image, the simplest kernel you can use to smooth and convolve the image is the average of the pixels. It consists in replacing each pixel intensity by the averate intensity over a chosen neighbourhood (e.g. 3x3, 5x5, 7x7, ..). The Mean Filter is an LTE operator as it can be expressed as a convolution with a kernel. Below, the kernels for a 3x3 mean filter:

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

2.4 Gaussian Filter

The Gaussian Filter is a filter for which the operator response function is a Gaussian function. If I pass something to the operator, the output will look like a Gaussian function. A 2D Gaussian is expressed as follows:

$$G(x, y) = G(x)G(y) = \frac{1}{2\pi\sigma^2} \exp^{-\frac{x^2+y^2}{2\sigma^2}}$$

Why Gaussian Filter is better than Mean Filter? Instead of having all same weights for the image's pixels, like in the Mean Filter where every pixel has the same importance in the computation on the new value for that specific position, in this case, closer pixels get a higher weigths due the shape of the Gaussian and far pixels get lower weights. Why this is good? Because it is more likely that closer pixels are part of the same object and

far pixels not. This is the main reason on why Gaussian Filter works pretty well for smoothing images.

Another important property about Gaussian Filter is that depending on σ, the Gaussian could be very bigger (higher) or smaller (more spreaded). Depeping on σ, you can smoothing centraly or in a larger region. It is possible to see in Figure 2.3 that the

Figure 2.3: Gaussian Filter with different sigma σ

higher σ, the stronger the smoothing caused by the filter. This can be understood, e.g., by observing that as σ increases, the weights of closer points get smaller while those of farther points get larger. As σ gets larger, small details disappear and the image content deals with larger size structures. Thus, filtering with a chosen σ can be thought of as setting the "scale" of interest to analyse image content.

How do we create a Gaussian kernel? Assuming that we are working with a 1D Gaussian function, $G(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp^{-\frac{x^2}{2\sigma^2}}$, in order to create a Gaussian kernel we select a σ value since we have to decide which shape our function has, bigger or spreaded, and then we sample at fixed position our Gaussian. When I have a pick in the Gaussian function, which is when $\sigma = 1$, maybe a kernel of 7x7 is sufficent to sample correctly my Gaussian. But when σ is larger, in order to describe the whole shape of the Gaussian, a 7x7 kernel is too small because the horizontal line is too large. There is a Rule-of-thumb to choose the size of the filter given σ:

- $\sigma = 1 \Rightarrow 7x7$
- $\sigma = 1.5 \Rightarrow 11x11$
- $\sigma = 2 \Rightarrow 13x13$
- $\sigma = 3 \Rightarrow 19x19$

By the properity of the Gaussian, we know that the 99% of the area of the Gaussian is cointained between $[-3\sigma, +3\sigma]$, a typical rule-of-thumb dictates taking a $(2k+1)x(2k+1)$ kernel with $k = [3\sigma]$.

2.5 Median Filter

In order to deal with Gaussian and Impulse noise, we need to use non-linear filters. Non-linear cannot be computed with convolution. Most simple non-linear filter is Median Filter.

$$\text{median}[A(x) + B(x)] \neq \text{median}[A(x)] + \text{median}[B(x)]$$

Median filtering prevent impulse noise effectively, as outliers (i.e. noisy pixels) tend to fall at either the top or bottom end of the sorted intensities. Median filtering tends to

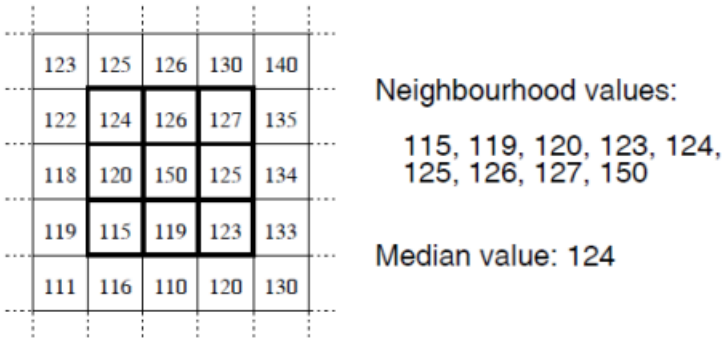


Figure 2.4: Median Filter example

keep sharper edges than linear filters such as the Mean or Gaussian. When dealing with impulse noise, the Median Filter can effectively denoise the image without introducing significant blur. Yet, Gaussian-like noise, such as sensor noise, cannot be dealt with by the Median, as this would require computing new noiseless intensities. Purposely, the Median may be followed by linear filtering. So, first we'd apply a Median Filter and then the Gaussian one.

2.6 Bilateral Filter

The Bilateral Filter is used as an advanced non-linear filter to accomplish denoising of Gaussian-like noise without blurring the image (aka edge preserving smoothing). With linear filter, during the denoising process, we go to remove details of the image. Doing so, edges are not visibale anymore. We want to preserve the sharpness of the image. Let's see the implementation.

Instead of just looking at the spatial position, we look also at the intesity values, so on how much they change between each other. The kernel created depends both on space and intensity values. This is the bilateral filter. Let's explain the Figure 2.5. For

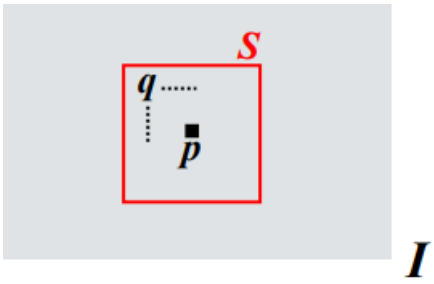


Figure 2.5: How Bilateral Filter is made

every pixel q in the neighborhood of pixel p, we compute a function H for every pair (p, q), where I_q is the intensity value of the pixel q. H is computed by the product of two function based on two distances, spatial distance d_s and intensity distance d_r.

$$O(p) = \sum_{q \in S} H(p, q) \cdot I_q$$

$$H(p, q) = \frac{1}{W(p)} G_{\sigma_s}(d_s(p, q)) G_{\sigma_r}(d_r(p, q))$$

$$d_s(p, q) = ||p - q||_2 = \sqrt{(u_p - u_q)^2 + (v_p - v_q)^2}$$

$$d_r(I_p, I_q) = |I_p - I_q|$$

To conclude, bilateral filtering is an effective way to smooth an image while preserving its discontinuities and also to separate image structures of different scales.

Chapter 3

Edge Detection

Gradiente di una funzione Prima di continuare con l'Edge Detection penso sia necessario fare l'introduzione di qualche concetto teorico matematica, col fine di comprendere a pieno i contenuti di seguito illustrati. Il gradiente di una funzione è un vettore che ha come componenti le derivate parziali della funzione. In generale, il gradient di una funzione f, denotato con ∇f, è definto in ciasun punto della seguente relazione: per un qualunque vettore $\vec{v}$, il prodotto scalare $\vec{v} \cdot \nabla f$ dà il valore della derivata direzionale di f rispetto a $\vec{v}$. Più semplicemente lo possiamo definire come un vettore che rappresenta la direzione e l'intensità massima di crescita di una funzione in un punto specifico. Esso è composto dalle derivate parziali della funzione rispetto alle variabili indipendenti e fornisce informazioni sulla variazione della funzione in diverse direzioni. In sostanza, il gradiente indica la direzione in cui la funzione cresce più rapidamente.

One of the first features, informations that can be extracted from the images, are edges. Edges being the contour of objects. What are edges really about? We can extract just the countour of the object. We extract also every possible variation in the image, so as you can see in the Figure 3.1, the edge detection is done both on the square that on the circle. Qualitatively, edges occur at boundaries between regions of different color, intensity of texture. Unfortunately, segmenting an image into coherent regions is a diffuclt task. Often, it is preferable to detect edges using only purely local information. A resonable approach is to define an edge as a location of rapid intensity or color variation. Think of an image as a height field. On such a surface, edges occur at locations of steep slopes. A mathematical way to define the slope and direction of a surface is through its gradient

$$J(x) = \nabla I(x) = \Big(\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\Big)(x)$$

The local gradient vector J points in the direction of steepest ascent in the intensity function. Its magnitude is an indication of the slope of the variation, while its orientation points in a direction perpendicular to the local contour.

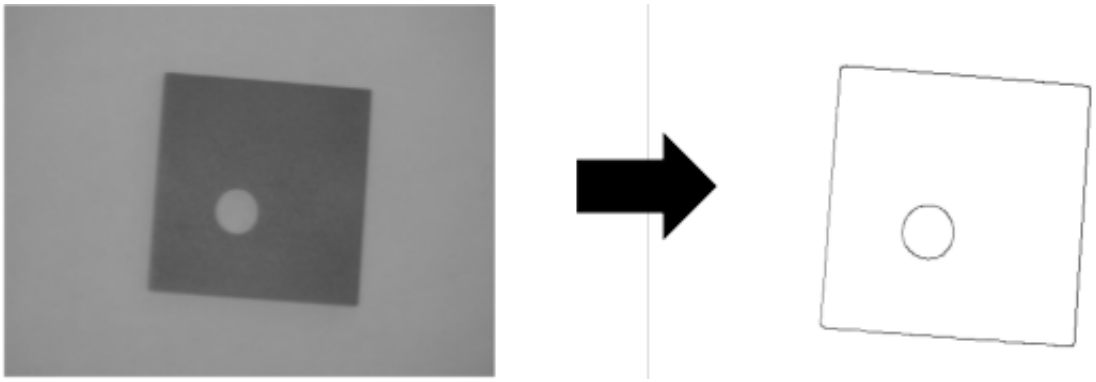


Figure 3.1: Edge Detection of an object.

Unfortunately, taking image derivatives accentuates high frequencies and hence amplifies noise, as the proportion of noise to signal is larger at high frquencies. It is therefore prudent to smooth the image with a low-pass filter prior to computing the gradient. Because we would like the response of our edge detector to be indepedent of orientation, a circularly symmetric smoothing filter is desirable. The Gaussian is the only separable circularly symmetric filter, so it is used in most edge detection algorithms.

Because differentiation is a linear operation, it commutes with other linear filtering operations. The gradient of the smoothed image can therefore be written as

$$J_\sigma(x) = \nabla[G_\sigma(x) * I(x)] = [\nabla G_\sigma](x) * I(x)$$

For many applications, however, we wish to thin such a continuous gradient image to return isolated edges only, i.e., as single pixels at discrete locations along the edge contours. This can be achieved by looking for *maxima* in the edge strength (gradient magnitude) in a direction *perpendicular* to the edge orientation, i.e., along the gradient direction.

Finding this maximum corresponds to taking a directional derivative of the strength field in the direction of the gradient and then looking for zero crossings. The desired directinoal derivative is equivalent to the dot product between a second gradient operator and the results of the first,

$$S_\sigma(x) = \nabla \cdot J_\sigma(x) = [\nabla^2 G_\sigma](x) * I(x)$$

The gradient operator dot product with the gradient is called the *Laplacian*. The convolution kernel

$$\nabla^2 G_\sigma(x) = \Big(\frac{x^2 + y^2}{\sigma^4} - \frac{2}{\sigma^2}\Big) G_\sigma(x)$$

is therefore called the *Laplacian of Gaussian* (LoG) kernel.

LoG is a robust edge detector which includes a smoothing step to filter out noise. This detector can be summarized as:

- Gaussian Smoothing: $\tilde{I}(x, y) = I(x, y) * G(x, y)$
- Second order differentation by the Laplacian.
- Extraction of the zero-crossing of $\nabla^2 \tilde{I}(x, y)$

3.1 Non-Maxima Suppression (NMS)

This technique is used in different topics in Computer Vision. Most classical edge detection algorithms are based on the concept of first derivatives. At an edge location in an image, there is a quick transition from low to high intesity or vice-versa. To detect an edge we simply take the first derivative of the pixel intesiteis and look for the maximum points.

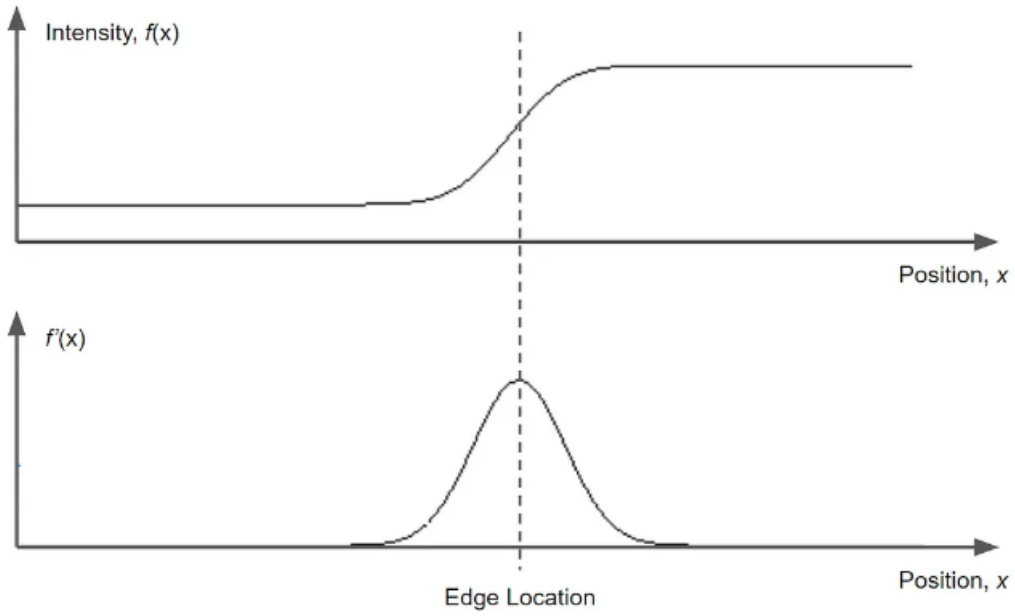


Figure 3.2: Edge Detection 1D Derivative

3.2 Canny Edge Detector

Usually, this derivative is combined with a Gaussian Filter in order to perform image smoothing and edge detection in one step. We simply apply the differentiated Gaussian filter directly to the image. The output already gives the position of edges with a high value. Next step, is apply **Non-Maxima Suppression (NMS)**. NMS is done by tracking along the high values in the output image then checking for maximal gradient in

a 3x3 neighborhood. The counter pixel has to be the highest in the direction perpedicular to the edge, or else it will be set to 0.

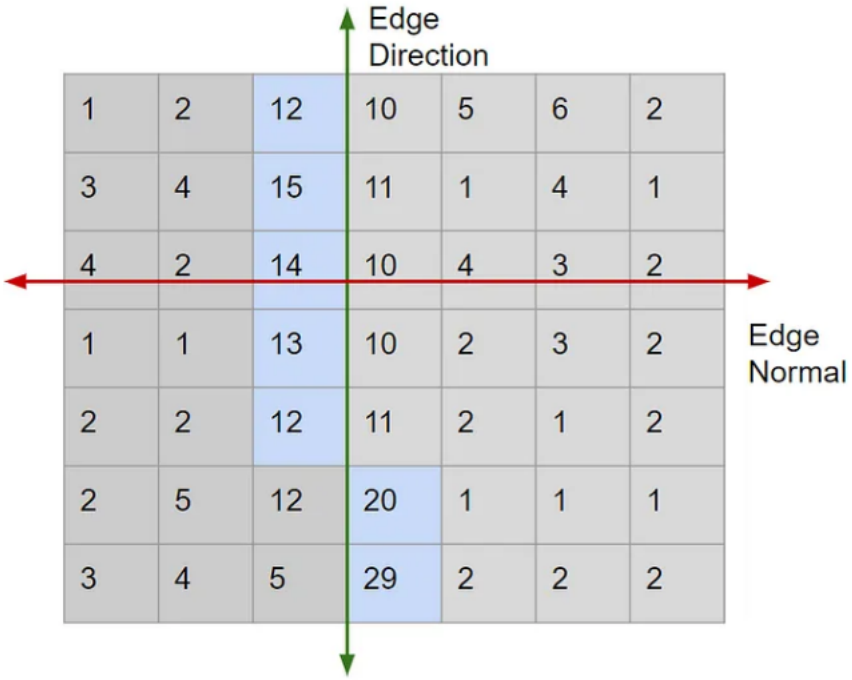


Figure 3.3: Non-Maxima Suppression (NMS)

This is a 7x7 matrix with sample outputs from Gaussian Filter, Figure 3.3. High values indicates that the first derivatives is high at that location. To perform NMS, we track along the *edge direction* (green line), then set values to 0 if they are not the maximum in the *edge normal direction* (red line). Therefore, after NMS the blue cells will remain and the white ones will be set to 0. This gives a fine edge output that is only one pixel wide.

3.2.1 Hysteresis Thresholding

This step is performed in tandem with NMS and the main diea is to prevent noisy edges from causing fragmentation in the final edge detection. Two threshold values are selected with $T_1 > T_2$. The NMS tracking any starts if the gradient values are higher than T_1 and stop only if gradient values are lower than T_2. Which means that a pixel is taken as an edge if either the gradient magnitude is higher than T_h **or** higher than T_l **and** the pixel is a neighbor of on already detected edge.

2D of the second derivatives of the signal to locate edges convolution by a Gaussian can be slow and we can leverage on separability of the Gaussian function to speed-up

the calculation. Look for *zero-crossing* of the second derivatives of the signal to locate edges (instead of the peaks of the first derivative).

Chapter 4

Local Features

So far we've just seen the edges as features in an image. In this chapter we are going to see in details more features that can be seen in an image and how they can be extrapolated.

To start we can look at one of the most important problem in Computer Vision: finding correspondences between images. For correspondences we mean the projection in two different images of the same 3D point in the scene. An example is panorama stitching. They idea is that you have two images where they share parts of the scene, so you've some overlap between the two images. In order to be able to construct the panorama, you need to apply a mathematical transformation which is known as *homography*. An homography is a 3x3 matrix with 8 degree of freedom, where in order to compute the 8 values, we need at least 4 correspondences, so 4 points (x,y). This correspondences can be found among *salient points* that are present in the image.

The first step to create panorama stitching algorithm is to define the algorithm to detect salient points. Detect points in the image independently for the two images which are salient. Once we have detected salient points, we want to match these salient points between the two images. If we take just a pixel value, red dot in Figure 4.1, and we try to match it with the pixel value in the other image, there would be a lot of pixels with the same intensity. In order to describe the content of a salient point, we usually consider a neighbourhood of points around that point and we write a algorithm that uses all these intesities to compute a feature vector, known as *descriptor*. Once we have for all salient points detected a descriptor, we match them between the two images and we choose the point that provides the shorter distance.

Descriptors should be invariant to as many transformations as possible. The description algorithm should capture the salient information around a keypoint, so to keep important tokens and ignore changes due to nuisances (e.g. light changes) and noise. The descriptor should also be as concise as possible, to minimize memory occupancy

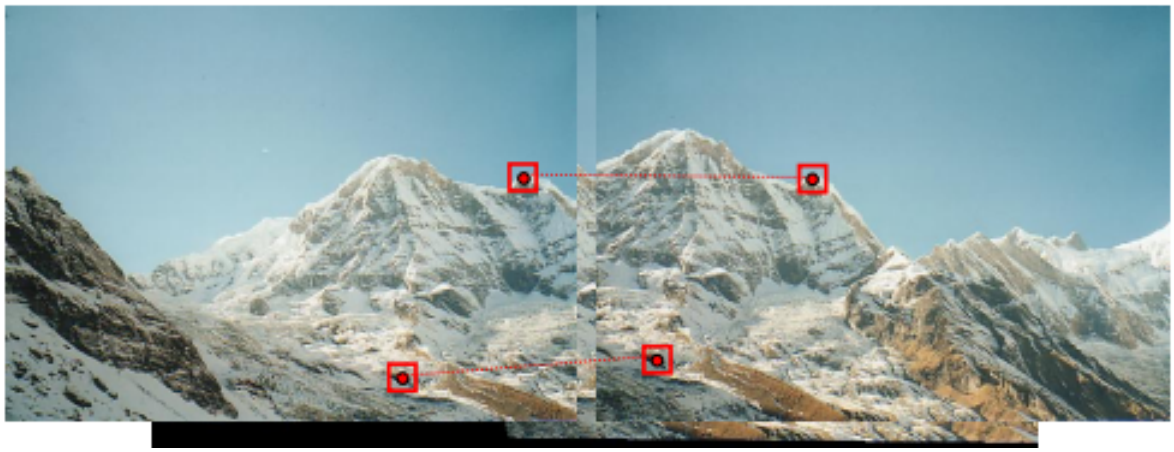


Figure 4.1: Panorama Stitching Image

and allow for efficient matching. Instead, the detector should find the same keypoints in different views of the scene despite the transformations undergone by the images. It also should find keypoints surrounded by informative patterns (good for making them discriminative for the matching). The speed is desirable for both, but in particular for detectors, which need to be run on the whole image, while descripotr are computed at keypoints only. Those are properties for good detectors and descriptors.

4.1 Corners vs Edges

Can edges be found repeatably across different images? Are these points easily identified across different images? No. Edge pixels can be hardly told apart as they look very similar along the direction perpendicular to the gradient. Edges are locally ambiguous, many other points that look just the same. It's possible to see it in Figure 4.2. We want

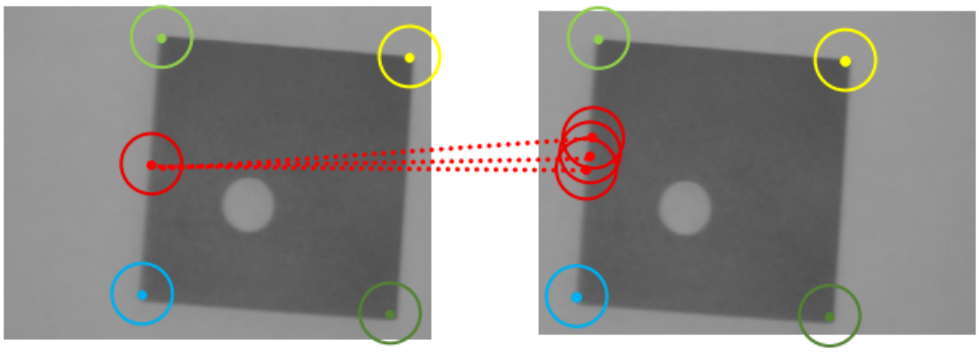


Figure 4.2: Corners vs Edges

pixels that exhibits a large variation along all directions. A first great solution has been present by Moravec. The cornerness at p, which is a point in the image, is given by the minimum squared difference between a patch (7x7 for example) centered at p and those at its 8 neighbours.

$$C(p) = \min_{q \in n_8(p)} ||N(p) - N(q)||^2$$

The algorithm is runned all over the image because the detector goes all over the image.

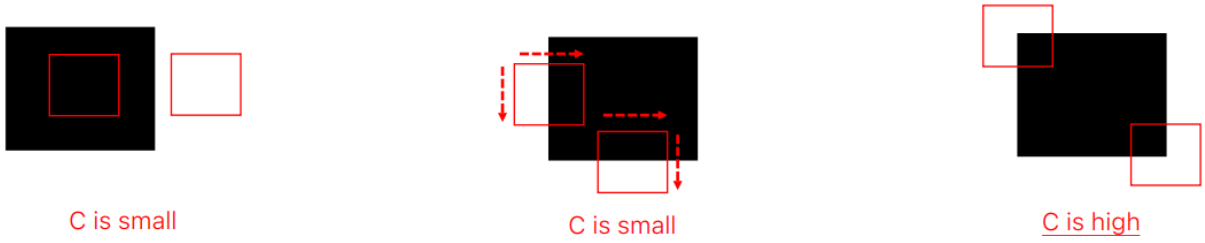


Figure 4.3: **Left Image:** Uniform Region - no change in all directions. **Center Image:** Edge - no change along the edge direction. **Right Image:** Corner - significant change in all directions.

So, for every point in the image the minimum value is computed. Once the patch has been applied on the whole image, some thresholding is fixed to filter out some cornerness value. This one was the very first and simple corner detector. Moravec detector suffers from a problem: it is not invariant to rotation. We need a detector that is going to be invariant to rotation, illumination changes and scale changes.

4.1.1 Harris Corner Detector

Harris did just a small change to Moravec detector. He moved it in a continuos formulation to define a corner detector which is focus to a couple of things: not just rotation but also a bit on illumination changes. Instead of moving to one pixel to another, Harris moves his detector to an infinitesimal part. Let's see more in details.

Assume to shift the image with a generic infinitesimal shift $(\Delta x, \Delta y)$. $w(x, y)$ is a window set to 1 around the pixel under evaluation and 0 in all the image. In the end consider only the pixel around the x, y position. Due to the shift being infinitesimal, we can deploy Taylor's expansion of the intensity function at (x, y). The first order taylor expansion says that if I translate my function for a very small quantity the input, this can be represented exactly as the function at point x with the change weighted infinitesimal value. Mathematical we mean: $f(x + \Delta x) = f(x) + f'(x)\Delta x$.

The Harris corner detection algorithm can be summarized as follows:

- Compute C at each pixel

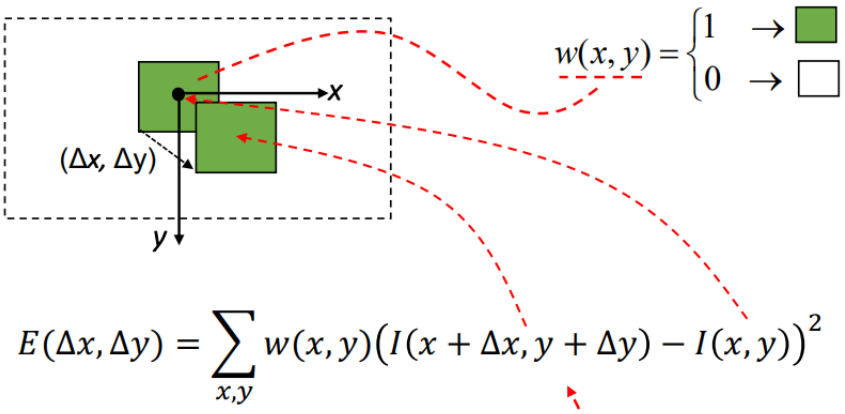


Figure 4.4: Harris detector for edge and corner

- Select all pixels where C is higher than a chosen positive threshold (T)
- Within the previous set, detect as corners only those pixels that are local maxima of C (NMS)

It is worth highlighting that the weighting function $w(x, y)$ used by the Harris corner detector is Gaussian rather than Box-shaped, so to assign more weight to closer pixels and less weight to those farther away, Figure 4.5.

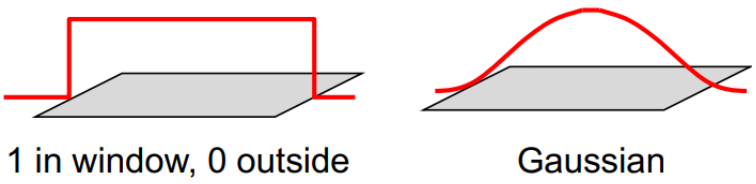


Figure 4.5: Gaussian Box-shaped Harris

4.1.2 Invariance Properties

In many situations, detecting features at the finest stable scale possible may no be appropriate. For example, when matching images with little high-frequency detail (like clouds), fine-scale features may not exist. One solution to the problem is to extract features at a variety of scales, e.g., by performing the same operations at multiple resolutions in a pyramid and then matching features at the same level. However, for most object recognition applications, the scale of the object in the image is unknown. Instead of extracting features at maany different scales and then matching all of them, is it more efficient to extract features that are stable in both location and scale.

As with the Harris operator, pixels where there is strong asymmetry in the local curvature of the indicator function are rejected. Harris is not scale invariant. Given the chosen size of the detection window, all the points along the border are likely to be classified as edges. Should the object appear smaller in the image, use of the same window size would lead to detect a corner. The use of a fixed detection window size makes it impossible to repeatably detect homologous features when they appear at different scales in images. In Figure 4.6 is possible to see how does it work. An image contains

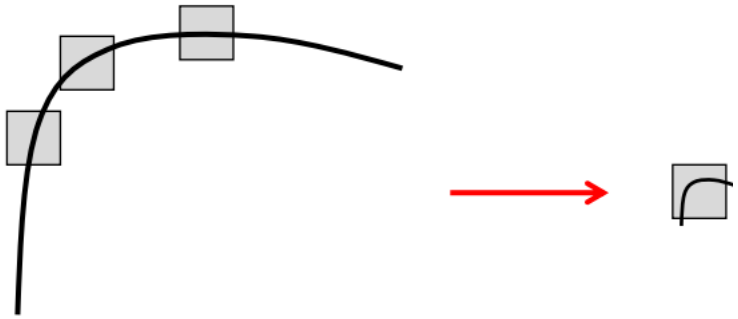


Figure 4.6: Scale invariance in Harris detector with a fixed detection window

features at different scales, indeed points that stand-out as interesting as long as a proper neighbourhood size is chosen to evaluate the chosen interestingness criterion. Detecting all features requires to analyze the image across the range of scales "deemed as relevant". The more features we gather the higher the change to match across pictures. Depending

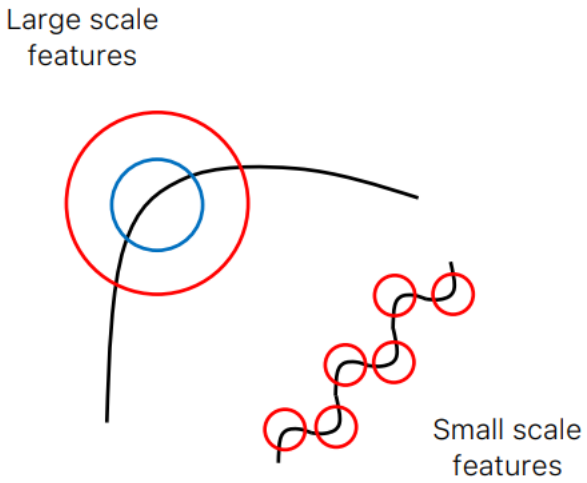


Figure 4.7: Different scale features images

on the acquisition settings (distance and focal length) an object may look differently in the image, in particular it may exhibit more or less details (i.e. features), as shown in

Figure 4.8. Features do exist within a certain range of scales. Finding similar features despite the different scales at which they may appear in different images. The same feature would simply be detected at different steps within a multi-scale image analysis process.

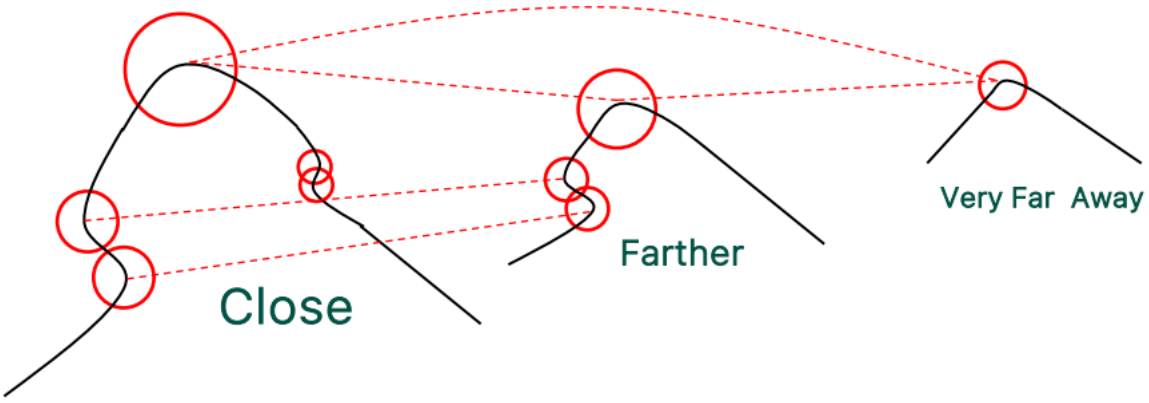


Figure 4.8: Image differently looking based on acquisition settings.

4.2 Feature Descriptors

After detecting keypoint features, we must match them, i.e., we must determine which features come from corresponding locations in differente images. In some situations, for video sequences for example, the local motion around each feature point may be mostly translational. In this case, simple error metrics, can be used to directly compare the intesities in smal patches around each feature point. Because feature points may not be exactly located, a more accurate matching score can be computed by performing incremental motion refinement, but this can be time-consuming and can sometimes even decrease performance.

4.2.1 Scale Invariant Feature Transform (SIFT)

SIFT features are formed by computing the gradient at each pixel in a 16 x 16 window around the detected keypoint, using the appropriate level of the Gaussian pyramid at which the keypoint was detected. The gradient magnitudes are downweighted by a Gaussian fall-off function (shown as a blue circle in Figure 4.9a) to reduce the influence of gradients far from the center, as these are more affected by small misregistrations.

In each 4 x 4 quadrant, a gradient orientation histogram is formed by (conceptually) adding the gradient values weighted by the Gaussian fall-off function to one of eight orientation histogram bins. To reduce the effects of location and dominant orientation

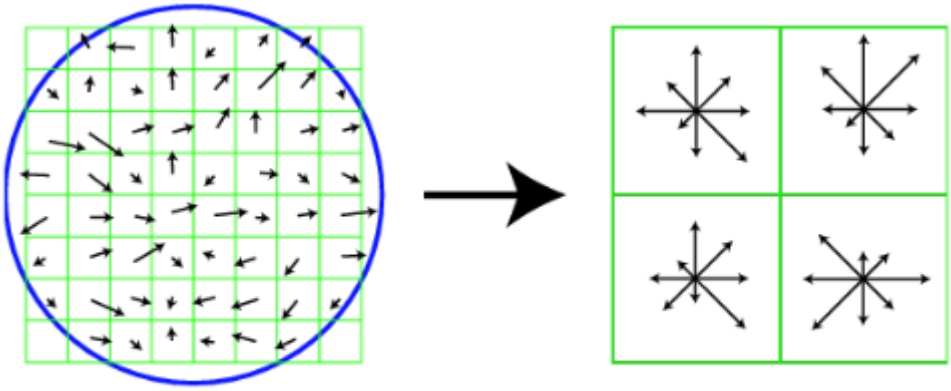


Figure 4.9: A schematic representation of Lowe's (2004) scale invariant feature transform (SIFT): (a) Gradient orientations and magnitudes are computed at each pixel and weighted by a Gaussian fall-off function (blue circle). (b) A weighted gradient orientation histogram is then computed in each subregion, using trilinear interpolation. While this figure shows an 8×8 pixel patch and a 2×2 descriptor array, Lowe's actual implementation uses 16×16 patches and a 4×4 array of eight-bin histograms.

misestimation, each of the original 256 weighted gradient magnitudes is softly added to 2 x 2 x 2 adjacent histogram bins in the (x, y, θ) space using trilinear interpolation. The 4x4 array of eight-bin histogram yields 128 non-negative values form a raw version of the SIFT descriptor vector. To reduce the effects of contrast or gain (additive variations are already removed by the gradient), the 128-D vector is normalized to unit length.

The SIFT descriptor, so, is computed as follows:

- A 16x16 oriented pixel grid around each keypoint is considered
- This is further divided into 4x4 regions (each of size 4x4 pixels)
- A gradient orientation histogram is created for each region
- Each histogram has 8 bins (i.e. bin size 45°)
- Each pixel in the region contributes to its designated bin according to
  - Gradient magnitude
  - Gaussian weighting function centered at the keypoint (with σ equal to hald the grid size)

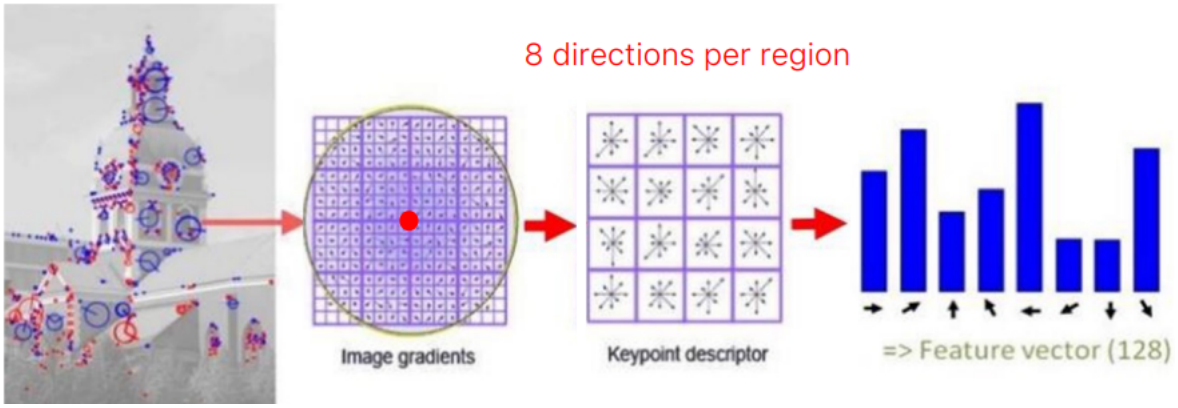


Figure 4.10: The descriptor size is given by the number of regions times the number of histogram bins per region, i.e. 4x4x8 = 128 (histograms are concatenated)

Chapter 5

Instance Object Detection

The *Instance-level Object Detection* problem occurs in countless applications and can be formulated as follows. Given a refernece image of a specfic object (i.e. the model image), determine whether the object is present or not in the image under analysis (i.e. the target image) and, in case of detection, estimate the pose of the object. Depending on the application, the pose may often be given by a translation, a roto-translation or a similarity, which is roto-translation plus scale. The problem has a number of diverse facets: the sought object may appear only once or multiple times in the target image, we may be interested in detecting a number of diverse objects, each of which, in turn, may appear once of multiple times. For sake of simplicity, we will refer mainly to the basic setting: detecting a single object that may appear once in the target image. Typical troubles to deal with are *intensity changes*, *occlusions* and *clutter*. This problem is

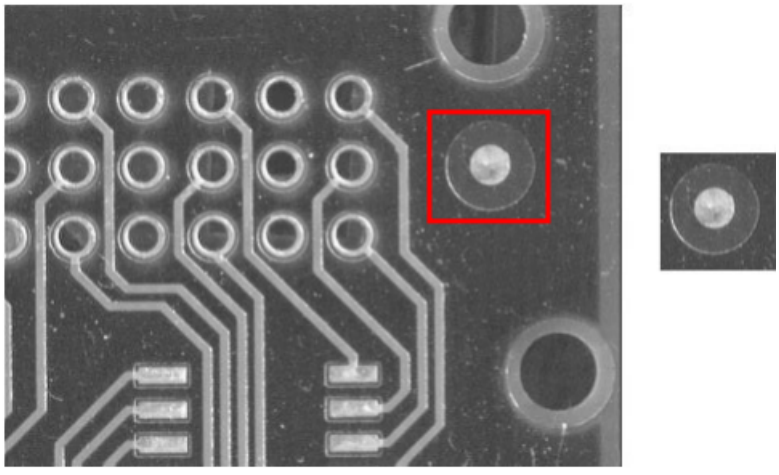


Figure 5.1: The reference image of a specific object on the right and the target image on the left. The idea is to determine if the reference image is present in the target image.

characterized by a limited variability. Computational efficiency is a major requirement

in most practical applications. There's also the *Category-level Object Detection*, which aims at detecting certain kind of object(s), like car, pedestrains, tree etc., regardless of their appearance and pose. Due to the high-variability, this problem is addressed by machine and deep learning techniques.

5.1 Template Matching

The model image is slid across the target image to be compared at each position to an equally sized window by means of a suitable (dis)similarity function, Figure 5.3. An example is shown in Figure 5.2. We move the model image T across the target image I.

Figure 5.2: Template Matching

$\tilde{I}(i,j)$, the window at position (i,j) of the target image having the same size as T. We compute then a pixel-wise intensities differences:

$$\mathrm{SSD}(i,j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} (I(i+m, j+n) - T(m,n))^2$$

SSD is the Sum of Squared Differences. It's also used the Sum of Absolute Differences

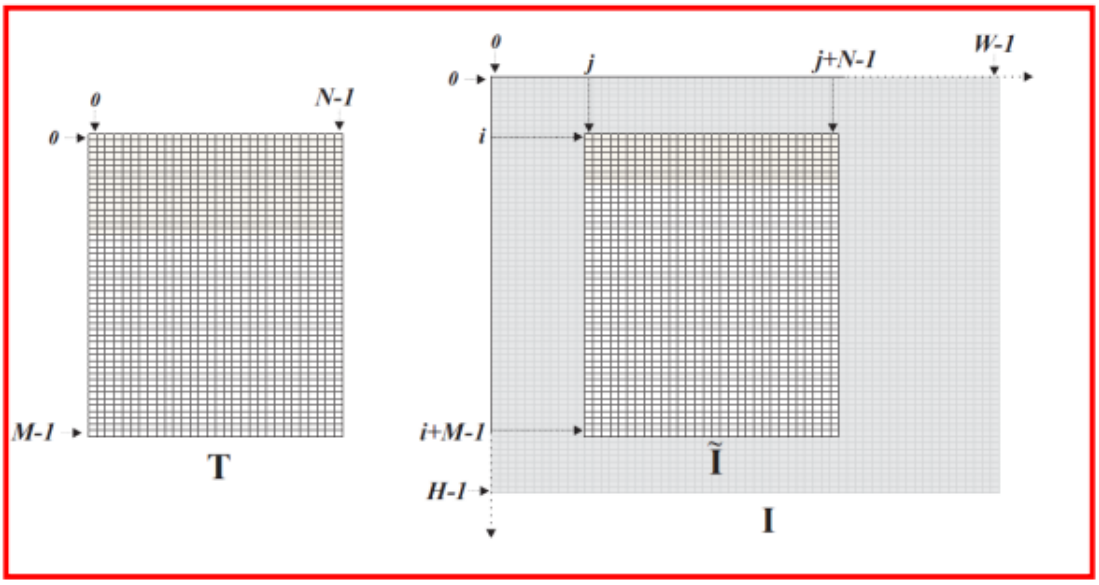


Figure 5.3: (Dis)Similarity Functions

(SAD): $\mathrm{SAD}(i,j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} |(I(i+m, j+n) - T(m,n)|$.

The main problem using SSD and SAD is that they are not invariant to intensity changes. To solve this problem is used the Zero-Mean Normalised Cross-Correlation (ZNCC). ZNCC turns out to be a similarity function very robust to intensity changes, Figure 5.4.

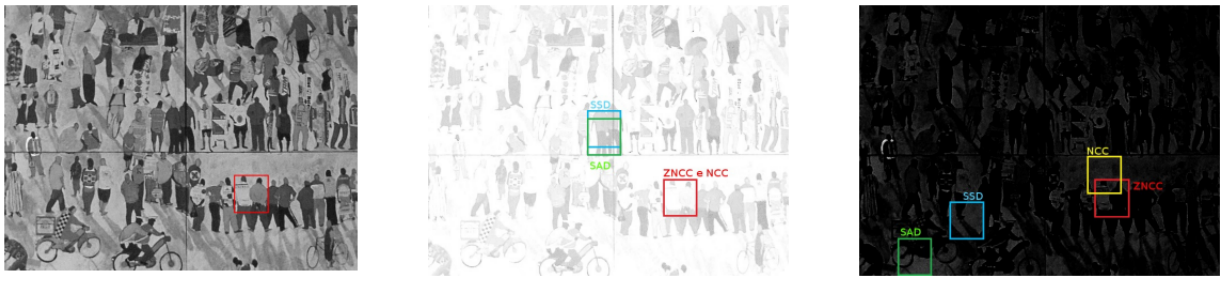


Figure 5.4: Comparison under significant intensity changes

Tempalce matching may be exceedingly slow whenever the model and/or target images have a large size, the computational complexity increases. A popular approach is to deploy an image pyramid.

5.2 The Hough Transform

The Hough Transform (HT) enables to detect objects having a known shape that can be expressed by an equation (e.g. lines, circles, ellipses, etc.) based on projection of the input data into a suitable space referred to as parameter or Hough space. The HT turns a global detection probelm into a local one, since it looks for feature points into the parameter space, instead of looking for the whole space in the image space. So the Hough space is different from the image space. The HT is usually applied after an edge detection process. The actual input data consist of the edge pixels extracted from the original image. The HT is robust to noise and allows for detecting the sought shape even though it is partially occluded into the image.

5.2.1 Basic Principle

HT formulation for lines is $y - mx - c = 0$. What is fixed and what is changing in this equation? In the usual image space interpretation of the line equation the parameters $(\hat{m}, \hat{c})$ are fixed $y - \hat{m}x - \hat{c} = 0$, so that the equation represents the mapping from point $\hat{m}, \hat{c}$ of the paramter space to the image points belonging to the line. However, we may instead fix $(\hat{x}, \hat{y})$, where $\hat{y} - m\hat{x} - c = 0$, so the equation represents the mapping from image point $(\hat{x}, \hat{y})$ to the parameter space providing all the lines through the image point.

Consider two image points P_1, P_2, and map both into the parameter space, we get two lines intersecting at the parameter space point representing the image line through P_1, P_2.

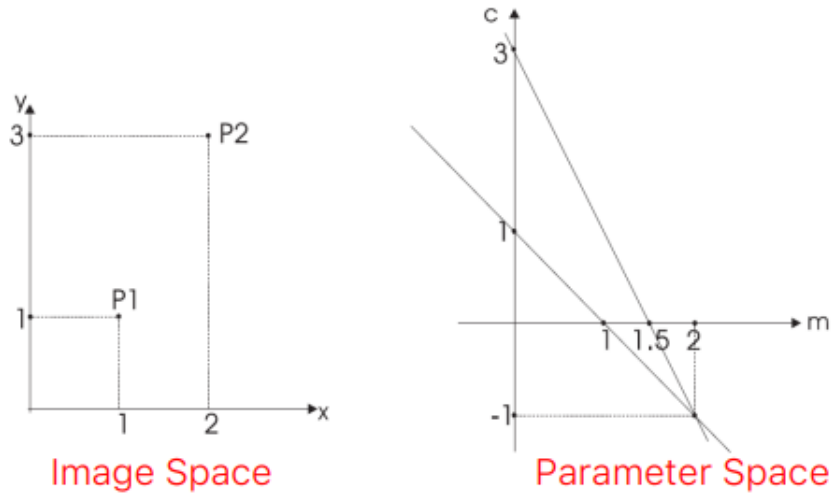


Figure 5.5: Image and Parameter space

m and c represent the parameters of the line passing through P_1 and P_2.

$$\begin{cases} \hat{y}_1 - m\hat{x}_1 - c = 0 \\ \hat{y}_2 - m\hat{x}_2 - c = 0 \end{cases} \Longrightarrow \begin{cases} m = \frac{\hat{y}_2}{\hat{y}_1} \\ c = \frac{\hat{x}_2\hat{y}_1 - \hat{x}_1\hat{y}_2}{\hat{x}_2 - \hat{x}_1} \end{cases} \tag{5.1}$$

More generally, if we map n image points, we get as many intersections as $n(n-1)/2$, which means the number of lines through the n image points. Considering n collinear image points, we can notice that their corresponding transforms, i.e. parameter space lines, will intersect at a single parameter space point representing the image line along which such n points lay, Figure 5.6.

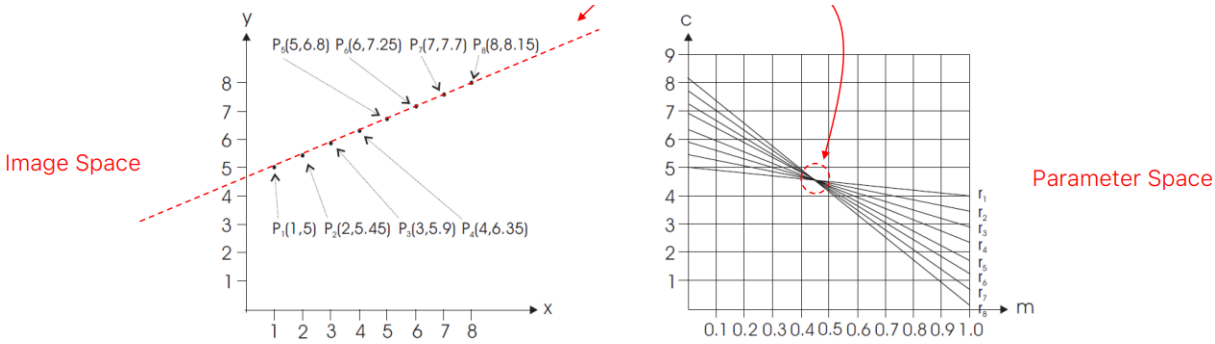


Figure 5.6: Rather than looking at an extended shape into the image, we look for a specific feature (where lines intersect) in the parameter space of lines

Therefore, given a sought analytic shape represented by a set of parameters, the HT consists in mapping image points (i.e. usually edge points) so as to create curves into the parameter space of the shape. Intersections of parameter space curves indicate the presence of image points explained by a certain instance of the shape. The more the intersecting curves the more are such image points and thus the higher is the evidence of the presence of that instance in the image. Detecting objects through the HT consists in finding parameter space points through which many curves do intersect (a local rather than global detection problem). To make it work in practice, the parameter space needs to be quantized and allocated as a memory array, which is often refereed to as Accumulator Array (AA). Curves are "drawn" into the AA by a so called voting process:

1. The transform equation is repeatedly computed to increment the bins satisfying the equation.

2. A nigh number of intersecting curves at a point of the parameter space will provide a high number of votes into a bin of the AA.

3. Finding parameter space points through which many curves do intersect is thus implemented in practice by finding peaks of the AA, i.e. local maxima showing a high number of votes.

It's possibile to see an example: the AA highlights the presence of a line with $m \in [0.3, 0.6], c \in [4, 5]$. To detect the line more accurately, the AA should be quantized more

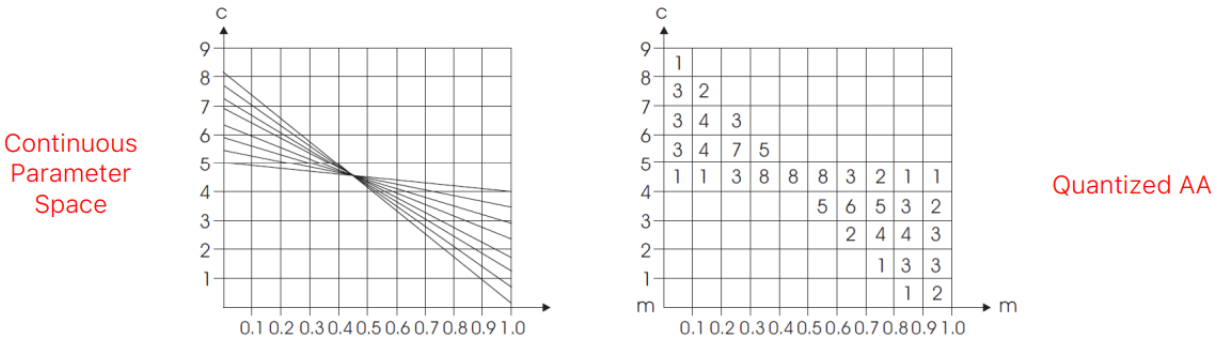


Figure 5.7: AA for line detection(Voting)

finely. The HT is robust to noise because spurious votes due to noise unlikely accumulate into a bin so as to trigger a false detection.

Chapter 6

Calibration

Camera calibration is a necessary step in 3D computer vision to extract metric information from 2D images. It involves estimating the parameters of a camera's lens and image sensor, including intrinsic parameters (focal length, principal point, skew coefficient, and distortion coefficients) and extrinsic parameters (rotation and translation vectors that define the position and orientation of the camera in the world coordinate system).

6.1 Homography

Homography is a fundamental concept in computer vision that describes the relationship between two planar projections of an image. It is a projective transformation that maps points in one plane to corresponding points in another plane. Mathematically, a homography is represented by a 3x3 matrix H that satisfies the equation $x^{'} = Hx$. The homography matrix H can be estimated from a set of corresponding points in the two planes. At least 4 point correspondences are required to compute H using linear methods. More robust estimation can be achieved using techniques like RANSAC to handle outliers. Some key properties of homography:

- It maps lines to lines (preserves collinearity).
- It maps circles to circles (preserves circularity).
- It does not necessarily preserve angles or distances (not an isometry).

Conceptually, homography is a mapping that allows points in one image plane to be transformed to corresponding points in another image plane. The key idea behind homography is to map points in one image (let's call it image B) to points in another image

(image A) using the homography matrix H. Mathematically, this can be represented as:

$$\begin{bmatrix} x^{'} \\ y^{'} \\ 1 \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

where $x^{'}$ and $y^{'}$ represent the points in image B mapped to the points x, y in image A. Homography is particularly effective when the camera motion involves only rotation without translation. In cases where translation is involved, points in the images may align, causing issues in image stitching and creating artifacts in the output. However, when imaging objects that lie in a plane, such as in satellite images where all points are assumed to lie on the ground plane, homography works well as the points cannot align in a problematic way.

To calculate the homography matrix given two images, the process typically involves four main steps:

1. **Feature Detection:** Identifying distinctive features in the images that can be used for comparison.

2. **Feature Description:** Describing the detected features in a way that allows for comparison across images.

3. **Feature Matching:** Matching the features between the images to find corresponding points.

4. **Calculating Homography Using RANSAC:** Using the RANSAC algorithm to find the homography matrix based on the matched feature points.

In summary, homography is a powerful tool in computer vision that enables the transformation of points between two image planes, facilitating applications like image stitching, panorama creation, augmented reality, and more.

Chapter 7

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are very similar to ordinary Neural Networks. They are made up of neurons that have learnable weights and biases. Eache neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. They still have a loss function (e.g. SVM/Softmax) on last (fully-connected) layer and all the tips used in regular neural networks still apply. So what changes? CNNs architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficent to impelement and vastly reduce the amount of parameters in the network.

7.1 Architecture Overview

Recall: Regular Neural Nets. Neural networks receive an input (a single vector), and transform it through a series of hidden layers. Each hidden layer is made up of a set of neurons, where each neuron is fully connected to all neurons in the previous layers, and where neurons in a single layer function completely indepndently and do not share any connections. The last fully-connected layer is called the "output layer" and in classification settings it represents the class scores.

Regualar Neural Nets do not scale well to full images. In CIFAR-10, which is a dataset of images, images are only of size $32x32x3$ (32 wide, 32 high, 3 color channels), so a single fully-connected neuron in a first hidden layer of a regualr Neural Network would have a $32 * 32 * 3 = 3072$ weights. This amount still seems manageable, but clearly this fully-connected structure does not scale to larger images. For example, an image of more respectable size, e.g. 200x200x3, would lead to neurons that have 200*200*3 = 120,000 weights. Moreover, we would almost certainly want to have several such neurons, so the

parameters would add up quickly! Clearly, this full connectivity is wasteful and the huge number of parameters would quickly lead to overfitting.

CNNs take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. In particular, unlike a regular Neural Network, the layers of a CNN have neurons arranged in 3 dimensions: width, height, depth. (Note that the word depth here refers to the third dimension of an activation volume, not to the depth of a full Neural Network, which can refer to the total number of layers in a network.) For example, the input images in CIFAR-10 are an input volume of activations, and the volume has dimensions 32x32x3 (width, height, depth respectively). As we will soon see, the neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner. Moreover, the final output layer would for CIFAR-10 have dimensions 1x1x10, because by the end of the ConvNet architecture we will reduce the full image into a single vector of class scores, arranged along the depth dimension. Here is a visualization:

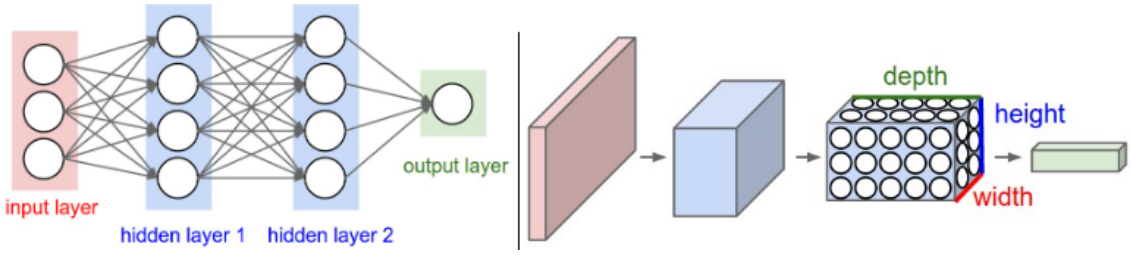


Figure 7.1: Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

A simple CNN is a sequence of layers where every layer transforms one volume of activations to another through a differntiable function. We use three main types of layers to build a CNN architecture: *convolutional Layer*, *Pooling Layer* and a *Fully-Connected Layer* (exactly as seen in regular NN). We will stack these layers to form a full ConvNet architecture.

Example Architecture: Overview. We will go into more details below, but a simple ConvNet for CIFAR-10 classification could have the architecture [INPUT - CONV - RELU - POOL - FC]. In more detail:

- INPUT [32x32x3] will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R,G,B.

- CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as [32x32x12] if we decided to use 12 filters.
- RELU layer will apply an element wise activation function, such as the $max(0, x)$ thresholding at zero. This leaves the size of the volume unchanged ([32x32x12]).
- POOL layer will perform a downsampling operation along the spatial dimensions (width, height), resulting in volume such as [16x16x12].
- FC (i.e. fully-connected) layer will compute the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.

In this way, CNNs transform the original image layer by layer from the original pixel values to the final class scores. Note that some layers contain parameters and other don't. In particular, the CONV/FC layers perform transformations that are a function of not only the activations in the input volume, but also of the parameters (the weights and biases of the neurons). On the other hand, the RELU/POOL layers will implement a fixed function. The parameters in the CONV/FC layers will be trained with gradient descent so that the class scores that the CNN computes are consistent with the labels in the training set for each image.

7.2 Convolutional Layer

The Conv layer is the core building block of a Convolutional Network. The CONV layer's parameters consist of a set of learnable filters. Every filter is small spatially (along width and height), but extends through the full depth of the input volume. For example, a typical filter on a first layer of a ConvNet might have sixe $5x5x3$. During the forward pass we slide (more precisely, convolve) each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. As we slide the filter over the width and height of the input volume, we will produce a 2-dimensional activation map that gives the responses of that filter at very spatial position. Intuitively, the network will learn filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color on the first layer. Now, we will have an entire set of filters in each CONV layer (e.g. 12 filters), and each of them will produce a separate 2-dimensional activation map. We will stack these activation maps along the depth dimension and produce the output volume.

7.2.1 Local Connectivity

When dealing with high-dimensional inputs such sa images, as we saw above it is impractical to connect neurons to all neurons in the previous volume. Instead, we will connect each neuron to only a local region of the input volume. The spatial extent of this connectivity is a hyperparameter called the *receptive field* of the neuron. The extent of the connectivity along the depth axis is always equal to the depth of the input volume. It is important to emphasize again this asymmetry in how we treat the spatial dimensions (width and height) and the depth dimension: The connections are local in 2D space (along width and height), but always full along the entire depth of the input volume.

Example 1. For example, suppose that the input volume has size $[32x32x3]$, (e.g. an RGB CIFAR-10 image). If the receptive field (or the filter size) is $5x5$, then each neuron in the Conv Layer will have weights to a $[5x5x3]$ region in the input volume, for a total of $5*5*3 = 75$ weights (and $+1$ bias parameter). Notice that the extent of the connectivity along the depth axis must be 3, since this is the depth of the input volume.

Example 2. Suppose an input volume had size $[16x16x20]$. Then using an example receptive field size of $3x3$, every neuron in the Conv Layer would now have a total of $3 * 3 * 20 = 180$ connections to the input volume. Notice that, again, the connectivity is local in 2D space (e.g. $3x3$), but full along the input depth (20).

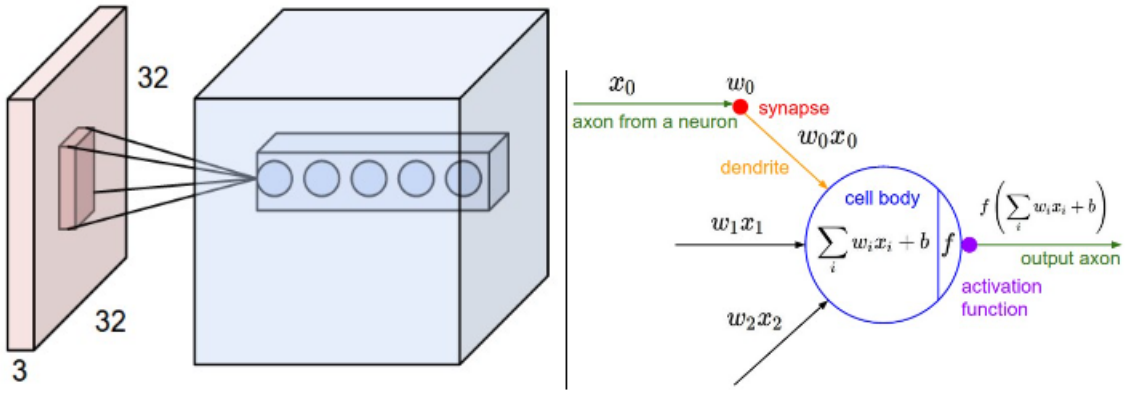


Figure 7.2: **Left:** An example input volume in red (e.g. a 32x32x3 CIFAR-10 image), and an example volume of neurons in the first Convolutional layer. Each neuron in the convolutional layer is connected only to a local region in the input volume spatially, but to the full depth (i.e. all color channels). Note, there are multiple neurons (5 in this example) along the depth, all looking at the same region in the input: the lines that connect this column of 5 neurons do not represent the weights (i.e. these 5 neurons do not share the same weights, but they are associated with 5 different filters), they just indicate that these neurons are connected to or looking at the same receptive field or region of the input volume, i.e. they share the same receptive field but not the same weights. **Right:** The neurons from the Neural Network chapter remain unchanged: They still compute a dot product of their weights with the input followed by a non-linearity, but their connectivity is now restricted to be local spatially.

7.2.2 Spatial arrangement

We have explained the connectivity of each neuron in the CONV layer to the input volume, but we haven't yet discussed how many neurons there are in the output volume or how they are arranged. Three hyperparameters control the size of the output volume: the *depth*, *stride* and *zero-padding*.

1. First, the *depth* of the output volume is a hyperparameter: it corresponds to the number of filters we would like to use, each learning to look for something different in the input. For example, if the first Convolutional Layers takes as input the raw image, then different neurons along the depth dimension may activate in presence of various oriented edges, or blobs of colore. We will refer to a set of neurons that are all looking at the same region of the input as a *depth column*.

2. Second, we must specify the *stride* with which we slide the fliter. When the stride is 1 then we move the filters one pixel at a time. When the stride is 2 then the filters jump 2 pixels at a time as we slide them around. This will produce smaller output volumes spatially.

3. Third, sometimes it will be convenient to pad the input volume with zeros around the border. The size of this zero-padding is a hyperparameter. The nice feature of zero padding is that it will allow us to control the spatial size of the output volumes

We can compute the spatial size of the output volume as a function of the input volume size (W), the receptive filed size of the CONV layer neurons (F), the stride with which they are applied (S), and the amount of zero padding used (P) on the border. The correct formula for calculating how many neurons "fit" is given by $\frac{(W - F + 2P)}{S + 1}$. For example, for a $7x7$ input and a $3x3$ filter with stride 1 and pad 0, we would get a $5x5$ output. With stride 2 we would get a 3x3 output. The graphical example is representent in Figure 7.3.

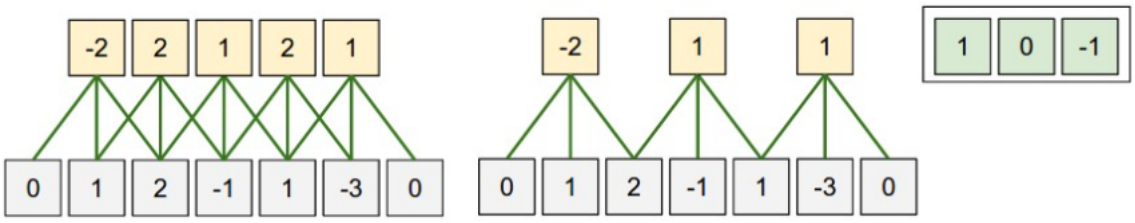


Figure 7.3: In this example there is only one spatial dimension (x-axis), one neuron with a receptive field size of F = 3, the input size is W = 5, and there is zero padding of P = 1. Left: The neuron strided across the input in stride of S = 1, giving output of size (5 - 3 + 2)/1+1 = 5. Right: The neuron uses stride of S = 2, giving output of size (5 - 3 + 2)/2+1 = 3. Notice that stride S = 3 could not be used since it wouldn't fit neatly across the volume. In terms of the equation, this can be determined since (5 - 3 + 2) = 4 is not divisible by 3. The neuron weights are in this example [1,0,-1] (shown on very right), and its bias is zero. These weights are shared across all yellow neurons (see parameter sharing below).

7.2.3 Parameter Sharing

Parameter sharing scheme is used in Convolutional Layers to control the number of parameters. It turns out that we can dramatically reduce the number of parameters by making one reasonable assumption: that if one feature is useful to compute at some spatial position (x, y), then it should also be useful to compute at a different position (x_2, y_2). In other words, denoting a single 2-dimensional slice of depth as a depth slice (e.g. a volume of size $[55x55x96]$ has 96 depth slices, each of size $[55x55]$), we are going to constrain the neurons in each depth slice to use the same weights and bias.

Let's see a real-world example: the AlexNet architecture that won the ImageNet challenge in 2012 accepted images of size $[227x227x3]$. On the first Convolutional Layer,

it used neurons with receptive field size $F = 11$, stride $S = 4$ and no zero padding $P = 0$. Since $(227 - 11)/4 + 1 = 55$, and since the Conv layer had a depth of $K = 96$, the Conv layer output volume had size $[55x55x96]$. Each of the $55 * 55 * 96$ neurons in this volume was connected to a region of size $[11x11x3]$ in the input volume. Moreover, all 96 neurons in each depth column are connected to the same $[11x11x3]$ region of the input, but of course with different weights.

So, talking about parameter sharing, the first Conv Layer in our example would now have only 96 unique set of weights (one for each depth slice), for a total of $96*11*11*3 = 34,848$ unique weights, or $34,944$ parameters (+96 biases). Alternatively, all $55 * 55$ neurons in each depth slice will now be using the same parameters. In practice during backpropagation, every neuron in the volume will compute the gradient for its weights, but these gradients will be added up across each depth slice and only update a single set of weights per slice.

Notice that if all neurons in a single depth slice are using the same weight vector, then the forward pass of the CONV layer can in each depth slice be computed as a convolution of the neuron's weights with the input volume (Hence the name: Convolutional Layer). This is why it is common to refer to the sets of weights as a filter (or a kernel), that is convolved with the input.

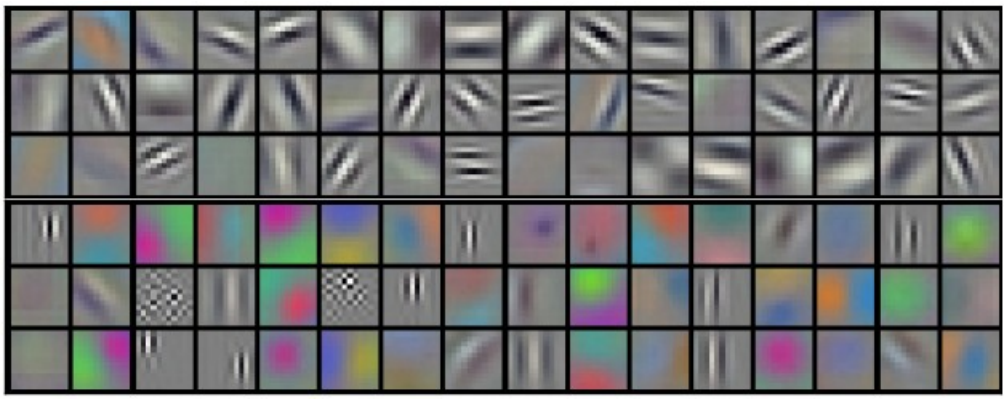


Figure 7.4: Example filters learned by Krizhevsky et al. Each of the 96 filters shown here is of size [11x11x3], and each one is shared by the 55*55 neurons in one depth slice. Notice that the parameter sharing assumption is relatively reasonable: If detecting a horizontal edge is important at some location in the image, it should intuitively be useful at some other location as well due to the translationally-invariant structure of images. There is therefore no need to relearn to detect a horizontal edge at every one of the 55*55 distinct locations in the Conv layer output volume.

To summarize, the Convolutional Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$

- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d_{th} depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d_{th} filter over the input volume with a stride of S, and then offset by d_{th} bias.

7.3 Pooling Layer

It is common to periodically insert a Pooling layer in-between successive Conv layers in a ConvNet architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation. The most common form is a pooling layer with filters of size $2x2$ applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. Every MAX operation would in this case be taking a max over 4 numbers (little $2x2$ region in some depth slice). The depth dimension remains unchanged. More generally, the pooling layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - their spatial extent F,
  - the stride S.

- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F)/S + 1$
  - $H_2 = (H_1 - F)/S + 1$
  - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- For Pooling layers, it is not common to pad the input using zero-padding.

It is worth noting that there are only two commonly seen variations of the max pooling layer found in practice: A pooling layer with $F = 3, S = 2$ (also called overlapping pooling), and more commonly $F = 2, S = 2$. Pooling sizes with larger receptive fields are too destructive.

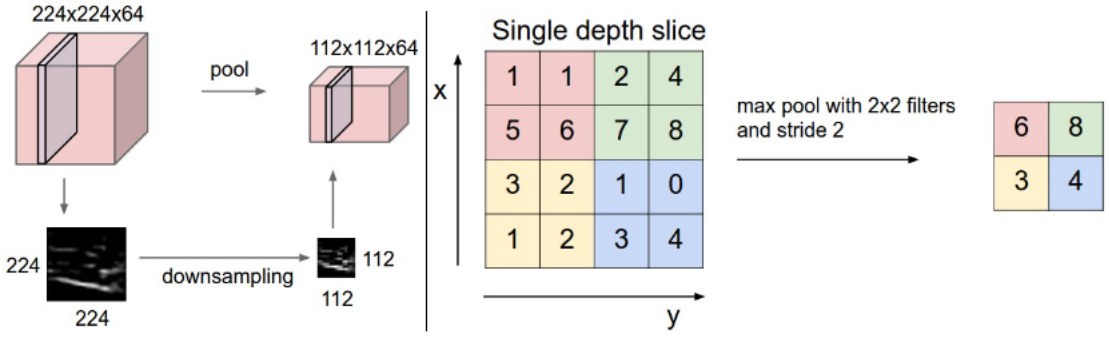


Figure 7.5: Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. Left: In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. Right: The most common downsampling operation is max, giving rise to max pooling, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square).

7.4 Normalization Layer

Normalizing a set of data transforms the set of data to be on a similar scale. For machine learning models, the goal is usally to recenter and rescale our data such that is between 0 and 1 or -1 and 1, depending on the data itself. One common way to accomplish this is to calculate the mean and the standard deviation on the set of data and transform each sample by subtracting the mean and dividing by the standard deviation, which is good if we assume that the data follows a normal distribution as this method helps us standardize the data and achieve a standard normal distribution. Normalization can

help training of our neural networks as the different features are on a similar scale, which helps to stabilize the gradient descent step, allowing us to use larger learning rates or help models converge faster for a given learning rate. To understand better what is layer normalization, we have to make a step back and undestarnd what is batch normalization and the needs.

Normalization, as we said before, works by mapping all values of a feature to be in the range $[0, 1]$ or $[-1, 1]$, using the transformation:

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Why we need that? If we look at the neuraln network architecture, the input layer is not the only input layer. For a network with hidden layers, the output of layer $k-1$ servers as the input to layer k. If the inputs to a particular layer change drastically, we can again run into the problem of unstable gradients. When working with large datasets, you'll split the dataset into multiple batches and run the mini-batch gradient descent. The mini-batch gradient descent algorithm optimizes the parameters of the neural network by batchwise processing of the dataset, one batch at a time. But why does this impede the training process?

For each batch in the input dataset, the mini-batch gradient descent algorithm runs its updates. It updates the weights and biases (parameters) of the neural network so as to fit to the distribution seen at the input to the specific layer for the current batch. Now that the network has learned to fit to the current distribution, if the distribution changes substantially for the next batch, it now has to update the parameters to fit to the new distribution. This slows down the training process. However, if we transpose the idea of normalizing the inputs to the hidden layers in the network, we can potentially overcome the limitations imposed by exploding activations and fluctuating distributions at the layer's input. Batch normalization helps us achieve this, one mini-batch at a time, to accelerate the training process.

Let's see in details what is batch normalization. For any hidden layer h, we pass the inputs through a non-linear activation to get the output. For every neuron (activation) in a particular layer, we can forse the pre-activations to have zero mean and unit standard deviation. This can be achieved by subtracting the mean from each of the input features across the mini-batch and dividing by the standard deviation. Following the output of the layer $k-1$, we can add a layer that performs this normalization operation across the mini-batch so that the pre-activations at layer k are unit Gaussians. The Figure below illustrates this.

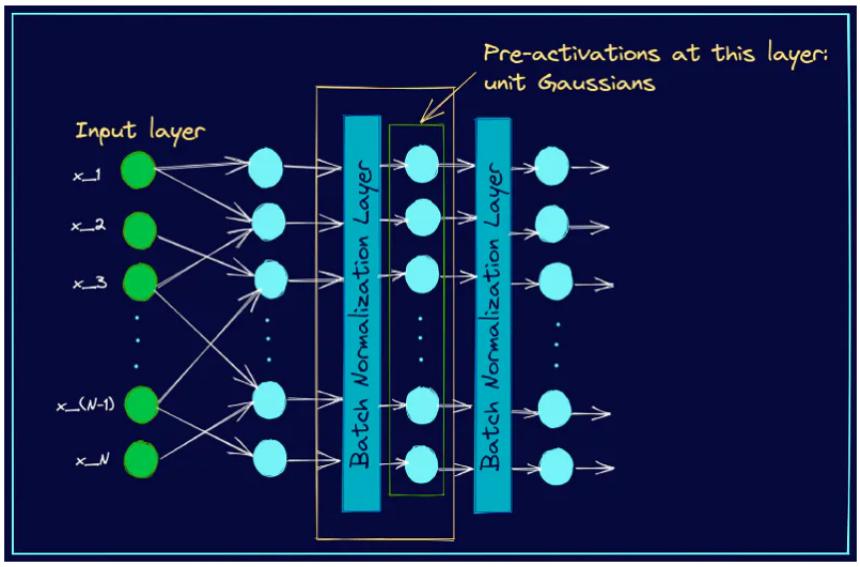


Figure 7.6: Section of a Neural Network with Batch Normalization Layer

As an example, let's consider a mini-batch with 3 input samples, each input vector being four features long. Here's a simple illustration, Figure 7.7, of how the mean and standard deviation are computed in this case. Once we compute the mean and standard deviation, we can subtract the mean and divide by the standard deviation.

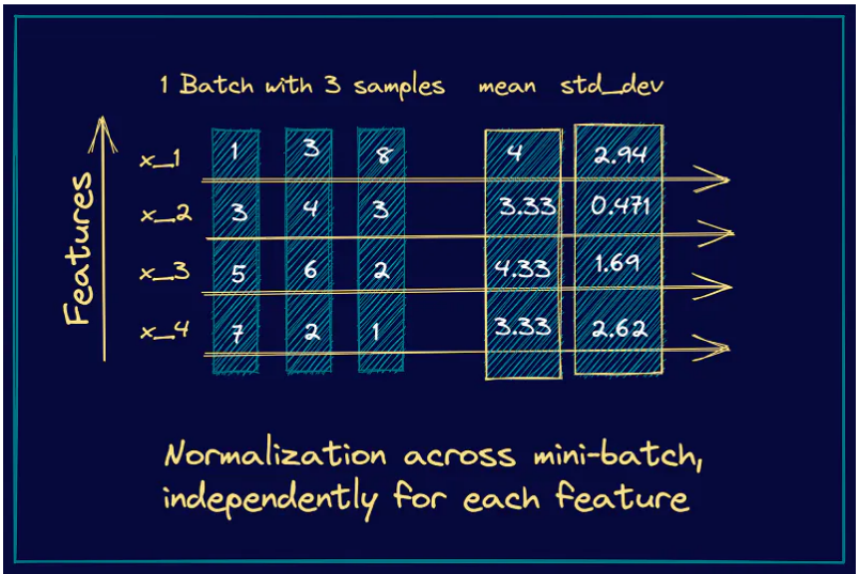


Figure 7.7: How Batch Normalization Works - An Example

Two limitations of batch normalization can arise:

- In batch normalization, we use the batch statistics: the mean and standard deviation corresponding to the current mini-batch. However, when the batch size is small, the sample mean and sample standard deviation are not representative enough of the actual distribution and the network cannot learn anything meaningful.
- As batch normalization depends on batch statistics for normalization, it is less suited for sequence models. This is because, in sequence models, we may have sequences of potentially different lengths and smaller batch sizes corresponding to longer sequences.

Now, we can examine layer normalization, a technique that can be used for sequence models. In layer normalization, all neurons in a particular layer effectively have the same distribution across all features for a given input. For example, if each input has d, it's a d-dimensional vector. If there are B elements in a batch, the normalization is done along the length of the d-dimensional vector and not across the batch of size B. Normalizing across all features but for each of the inputs to a specific layer removes the dependence on batches. This makes layer normalization well suited for sequence models such as transformers and recurrent neural networks (RNNs) that were popular in the pre-transformer era. Here's an example showing the computation of the mean and variance for layer normalization. We consider the example of a mini-batch containing three input samples, each with four features.

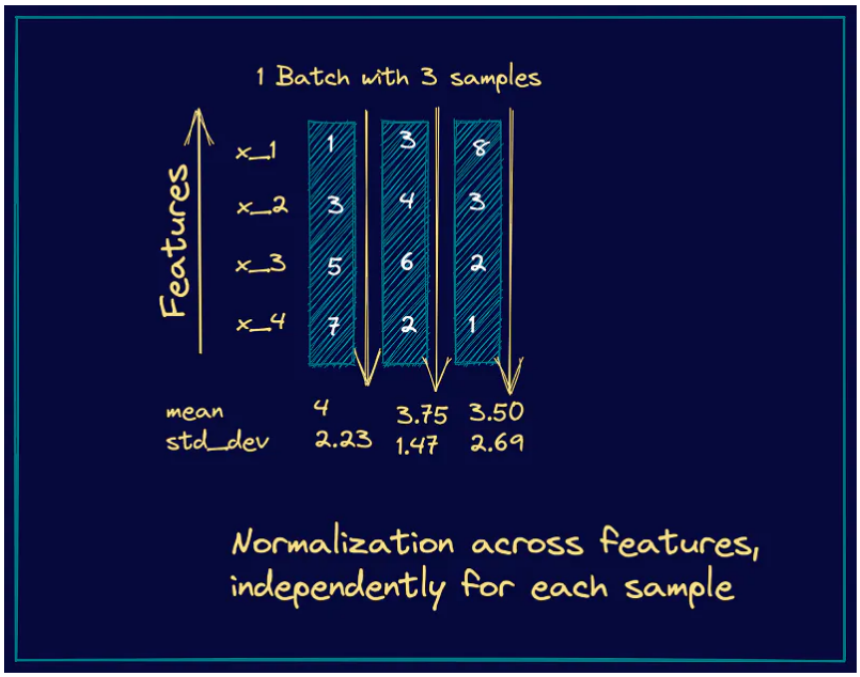


Figure 7.8: How Layer Normalization Works - An Example

Let's summarize the key differences between the two techniques:

- Batch normalization normalizes each feature independently across the mini-batch. Layer normalization normalizes each of the inputs in the batch independently across all features.
- As batch normalization is dependent on batch size, it's not effective for small batch sizes. Layer normalization is independent of the batch size, so it can be applied to batches with smaller sizes as well.
- Batch normalization requires different processing at training and inference times. As layer normalization is done along the length of input to a specific layer, the same set of operations can be used at both training and inference times.

7.5 ConvNet Architectures

We have seen that Convolutional Networks are commonly made up of only three layer types: CONV, POOL (we assume Max pool unless stated otherwise) and FC (short for fully-connected). We will also explicitly write the RELU activation function as a layer, which applies elementwise non-linearity. In this section we discuss how these are commonly stacked together to form entire ConvNets.

7.5.1 Layer Patterns

The most common form of a ConvNet architecture stacks a few CONV-RELU layers, follows them with POOL layers, and repeats this pattern until the image has been merged spatially to a small size. At some point, it is common to transition to fully-connected layers. The last fully-connected layer holds the output, such as the class scores. In other words, the most common ConvNet architecture follows this pattern. Prefer a stack of small filter CONV to one large receptive field CONV layer. Suppose that you stack three 3x3 CONV layers on top of each other (with non-linearities in between, of course). In this arrangement, each neuron on the first CONV layer has a 3x3 view of the input volume. A neuron on the second CONV layer has a 3x3 view of the first CONV layer, and hence by extension a 5x5 view of the input volume. Similarly, a neuron on the third CONV layer has a 3x3 view of the 2nd CONV layer, and hence a 7x7 view of the input volume. Suppose that instead of these three layers of 3x3 CONV, we only wanted to use a single CONV layer with 7x7 receptive fields. These neurons would have a receptive field size of the input volume that is identical in spatial extent (7x7), but with several disadvantages. First, the neurons would be computing a linear function over the input, while the three stacks of CONV layers contain non-linearities that make their features more expressive. Second, if we suppose that all the volumes have C channels, then it can be seen that the

single 7x7 CONV layer would contain $C \times (7 \times 7 \times C) = 49C^2$ parameters, while the three 3x3 CONV layers would only contain $3 \times (C \times (3 \times 3 \times C)) = 27C^2$ parameters. Intuitively, stacking CONV layers with tiny filters as opposed to having one CONV layer with big filters allows us to express more powerful features of the input, and with fewer parameters. As a practical disadvantage, we might need more memory to hold all the intermediate CONV layer results if we plan to do backpropagation.

7.5.2 Layer Sizing Patterns

We will first state the common rules of thumb for sizing the architectures and then follow the rules with a discussion of the notation:

- The **input layer** (that contains the image) should be divisible by 2 many times. Common numbers include 32 (e.g. CIFAR-10), 64, 96 (e.g. STL-10), or 224 (e.g. common ImageNet ConvNets), 384, and 512.

- The **conv layers** should be using small filters (e.g. 3x3 or at most 5x5), using a stride of $S = 1$, and crucially, padding the input volume with zeros in such way that the conv layer does not alter the spatial dimensions of the input. That is, when $F = 3$, then using $P = 1$ will retain the original size of the input. When $F = 5$, $P = 2$. For a general F, it can be seen that $P = \frac{(F-1)}{2}$ preserves the input size. If you must use bigger filter sizes (such as 7x7 or so), it is only common to see this on the very first conv layer that is looking at the input image.

  It is important to mention the 1x1 convolutions and why they're important in this field. Suppose that I have a conv layer which outputs an (N, F, H, W) shaped tensor where N is the batch size, F is the number of convolutional filters and H, W are the spatial dimensions. Suppose the input is fed into a conv layer with F_1 1x1 filters, zero padding and stride 1. Then, the output of this 1x1 conv layer will have shape (N, F_1, H, W). So, 1x1 conv filters can be used to change the dimensionality in the filter space. If $F_1 > F$ then we are increasing dimensionality, if $F_1 < F$ we are decreasing dimensionality, in the filter dimension.

- The **pool layers** are in charge of downsampling the spatial dimensions of the input. The most common setting is to use max-pooling with 2x2 receptive fields (i.e. $F = 2$), and with a stride of 2 (i.e. $S = 2$).

7.5.3 Case studies

There are several architectures in the field of Convolutional Networks that have a name. The most common are:

- **LeNet.** The first successful applications of Convolutional Networks were developed by Yann LeCun in 1990's. Of these, the best known is the LeNet architecture that was used to read zip codes, digits, etc.

- **AlexNet.** The first work that popularized Convolutional Networks in Computer Vision was the AlexNet, developed by Alex Krizhevsky, Ilya Sutskever and Geoff Hinton. The AlexNet was submitted to the ImageNet ILSVRC challenge in 2012 and significantly outperformed the second runner-up (top 5 error of 16% compared to runner-up with 26% error). The Network had a very similar architecture to LeNet, but was deeper, bigger, and featured Convolutional Layers stacked on top of each other (previously it was common to only have a single CONV layer always immediately followed by a POOL layer).

- **ZF Net.** The ILSVRC 2013 winner was a Convolutional Network from Matthew Zeiler and Rob Fergus. It became known as the ZFNet (short for Zeiler & Fergus Net). It was an improvement on AlexNet by tweaking the architecture hyperparameters, in particular by expanding the size of the middle convolutional layers and making the stride and filter size on the first layer smaller.

- **GoogLeNet.** The ILSVRC 2014 winner was a Convolutional Network from Szegedy et al. from Google. Its main contribution was the development of an Inception Module that dramatically reduced the number of parameters in the network (4M, compared to AlexNet with 60M). Additionally, this paper uses Average Pooling instead of Fully Connected layers at the top of the ConvNet, eliminating a large amount of parameters that do not seem to matter much. There are also several followup versions to the GoogLeNet, most recently Inception-v4.

- **VGGNet.** The runner-up in ILSVRC 2014 was the network from Karen Simonyan and Andrew Zisserman that became known as the VGGNet. Its main contribution was in showing that the depth of the network is a critical component for good performance. Their final best network contains 16 CONV/FC layers and, appealingly, features an extremely homogeneous architecture that only performs 3x3 convolutions and 2x2 pooling from the beginning to the end. Their pretrained model is available for plug and play use in Caffe. A downside of the VGGNet is that it is more expensive to evaluate and uses a lot more memory and parameters (140M). Most of these parameters are in the first fully connected layer, and it was since found that these FC layers can be removed with no performance downgrade, significantly reducing the number of necessary parameters.

- **ResNet.** Residual Network developed by Kaiming He et al. was the winner of ILSVRC 2015. It features special skip connections and a heavy use of batch normalization. The architecture is also missing fully connected layers at the end of

the network. The reader is also referred to Kaiming's presentation (video, slides), and some recent experiments that reproduce these networks in Torch. ResNets are currently by far state of the art Convolutional Neural Network models and are the default choice for using ConvNets in practice (as of May 10, 2016). In particular, also see more recent developments that tweak the original architecture from Kaiming He et al. Identity Mappings in Deep Residual Networks (published March 2016).

Chapter 8

Object Detection

The idea around object detection is pretty simple: detect a set of objects in a RGB image. For each object o_j, the output is a category $c_j \in [1, ..., C]$ (from a fixed list of categories, as in image classification) and a bounding box $BB_j = [x_j, y_j, w_j, h_j]$.

8.1 Viola Jones Object Detection

The Viola–Jones object detection framework is a machine learning object detection framework proposed in 2001. It was motivated primarily by the problem of face detection, although it can be adapted to the detection of other object classes. While it has lower accuracy than more modern methods such as convolutional neural network, its efficiency and compact size (only around 50k parameters, compared to millions of parameters for typical CNN like DeepFace) means it is still used in cases with limited computational power.

8.1.1 Problem Description

Face detection is a binary classification problem combined with a localization problem: given a picture, decide whether it contains faces, and construct bounding boxes for the faces. To make the task more manageable, the Viola–Jones algorithm only detects full view (no occlusion), frontal (no head-turning), upright (no rotation), well-lit, full-sized (occupying most of the frame) faces in fixed-resolution images.

A full presentation of the algorithm is in. Consider an image $I(x, y)$ of fixed resolution (M, N). Our task is to make a binary decision: whether it is a photo of standardized face (frontal, well-lit, etc) or not. Viola-Jones is essentially a boosted feature learning algorithm, trained by running AdaBoost algorithm on Haar feature classifiers to find a sequence of classifiers $f_1, f_2, ..., f_k$.

AdaBoost, short for Adaptive Boosting, is a supervised learning algorithm that is used to classify data by combining multiple weak or base learners (e.g., decision trees) into a strong learner. AdaBoost works by weighting the instances in the training dataset based on the accuracy of previous classifications. A weak learner (WL) is a simple classifier whose error is slightly better than random guessing (i.e. whose accuracy is slightly larger than 50% in a binary problem). AdaBoost is a way to train and build an ensemble of M weak classifiers to obtain a strong learner SL.

$$SL(x) = \sum_{j=1}^{M} \alpha_j WL_j(x) > 0$$

Weak classifiers used to detect faces are simple rectangular filters, comprising 2, 3 or 4 rectangles applied at a fixed position within a 24x24 patch.

$$WL_j(x) = \begin{cases} 1 \text{ if } s_j f_j > s_j p_j \\ -1 \text{ ow} \end{cases}$$

This is not slided. It is applied at a precision position in the image. It does not move. The threshold (rho) is learnt. Even with this simple definition and limited size of the patch, there are over 160K possible filters in a 24x24 patch. AdaBoost is used to select a small subset of the most effective ones. The window with -1 and $+1$ in Figure 8.1 is made by $s_j f_j$ in the weak learner formula above.

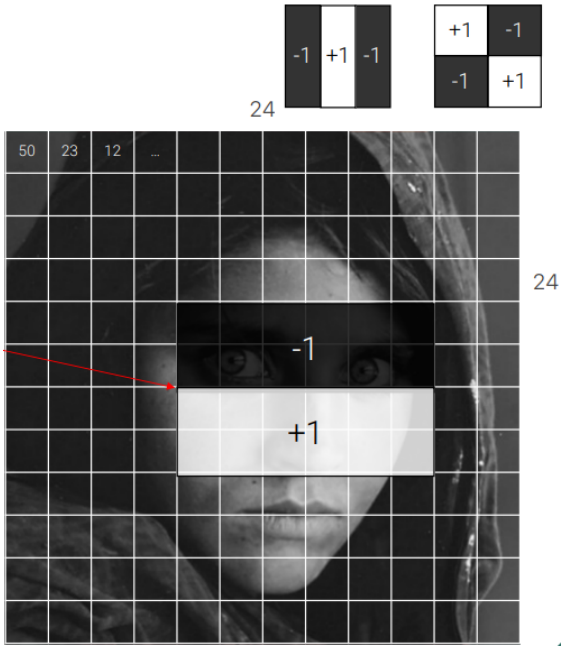


Figure 8.1: Viola-Jones application

8.1.2 Cascading classifier

Maybe the AdaBoost will finally select the best features, but it is still time-consuming process to calculate these features for each region. We have a 24x24 window which we slide over the input image, and we need to find if any of those regions containt the face. The job of the cascade is to quickly discard non-faces, and avoid wasting precious time and computations. Thus, achieving the speed necessary for real-time face detection. We set up a cascaded system in which we divide the process of identifying a face into multiple stages.

In the first stage, we have a classifier which is made up of our best features, in other words, in the first stage, the subregion passes through the best features such as the feature which identifies the nose bridge or the one that identifies the eyes. In the next stages, we have all the remaining features. When an image subregion enters the cascade, it is evaluated by the first stage. If that stage evaluates the subregion as positive, meaning that it thinks it's a face, the output of the stage is maybe. When a subregion gets a maybe, it is sent to the next stage of the cascade and the process continues as such till we reach the last stage. If all classifiers approve the image, it is finally classified as a human face and is presented to the user as a detection.

Now how does it help us to increase our speed? Basically, If the first stage gives a negative evaluation, then the image is immediately discarded as not containing a human face. If it passes the first stage but fails the second stage, it is discarded as well. Basically, the image can get discarded at any stage of the classifier

8.2 Box Overlap

Faces will give rise to several, overlapping detections in test images. To check if two boxes overlap, we measure the **Intersection over Union (IoU)** score.

$$IoU(BB_i, BB_j) = \frac{\text{area of intersection}}{\text{area of union}}$$

0.55 corresponds to a partial overlap, 0.75 corresponds to a good overlap and 0.90 corresponds to perfect overlap. Look Figure 8.2 below.

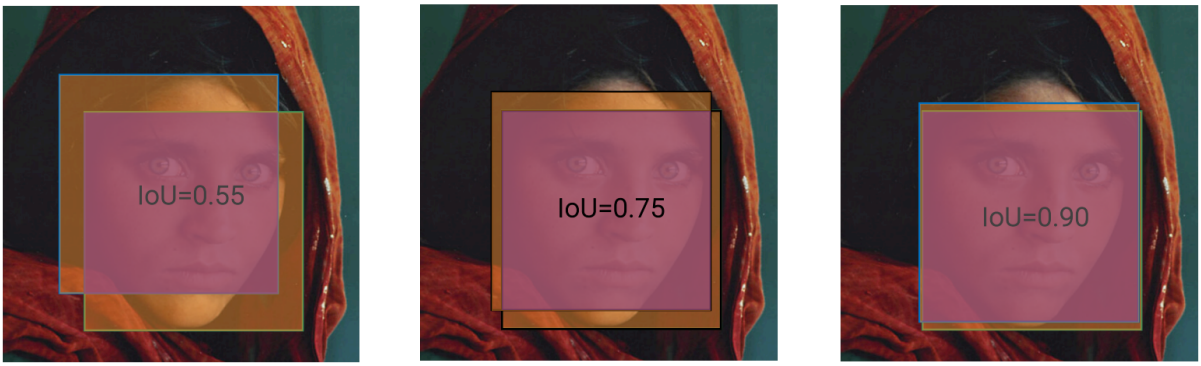


Figure 8.2: Different Intersection over Union (IoU)

In test images, we get several overlapping detections. Usually, a detection has an associated score which measures the confidence of the detector in its prediction. To obtain a single detection out of a set of overlapping boxes, we perform Non Maxima Suppression of boxes: consider the highest scoring BB, then eliminate all boxes with overlap greater than a threshold (i.e. IoU ¿ 0.5) and then repeat until all boxes have been tested.

8.3 Transfer Learning

In practice, very few people train an entire Convolutional Network from scratch (with random initialization), because it is relatively rare to have a dataset of sufficient size. Instead, it is common to pretrain a ConvNet on a very large dataset (e.g. ImageNet, which contains 1.2 million images with 1000 categories), and then use the ConvNet either as an initialization or a fixed feature extractor for the task of interest. The three major Transfer Learning scenarios look as follows:

- **ConvNet as fixed feature extractor.** Take a ConvNet pretrained on ImageNet, remove the last fully-connected layer (this layer's outputs are the 1000 class scores for a different task like ImageNet), then treat the rest of the ConvNet as a fixed feature extractor for the new dataset. In an AlexNet, this would compute a 4096-D vector for every image that contains the activations of the hidden layer immediately before the classifier. We call these features CNN codes. It is important for performance that these codes are ReLUd (i.e. thresholded at zero) if they were also thresholded during the training of the ConvNet on ImageNet (as is usually the case). Once you extract the 4096-D codes for all images, train a linear classifier (e.g. Linear SVM or Softmax classifier) for the new dataset.

- **Fine-tuning the ConvNet.** The second strategy is to not only replace and retrain the classifier on top of the ConvNet on the new dataset, but to also fine-

tune the weights of the pretrained network by continuing the backpropagation. It is possible to fine-tune all the layers of the ConvNet, or it's possible to keep some of the earlier layers fixed (due to overfitting concerns) and only fine-tune some higher-level portion of the network. This is motivated by the observation that the earlier features of a ConvNet contain more generic features (e.g. edge detectors or color blob detectors) that should be useful to many tasks, but later layers of the ConvNet becomes progressively more specific to the details of the classes contained in the original dataset. In case of ImageNet for example, which contains many dog breeds, a significant portion of the representational power of the ConvNet may be devoted to features that are specific to differentiating

- **Pretrained models.** Since modern ConvNets take 2-3 weeks to train across multiple GPUs on ImageNet, it is common to see people release their final ConvNet checkpoints for the benefit of others who can use the networks for fine-tuning. For example, the Caffe library has a Model Zoo where people share their network weights.

8.4 Object Localization

Object Localization refers to the task of precisely identifying and localizing objects of interest within an image. In the context of CNN-based localizers, object localization involves training a convolutional neural network (CNN) to predict the coordinates of bounding boxes that tightly enclose the objects within an image. The localization process typically follows a two-step pipeline, with a backbone CNN extracting image features and a regression head predicting the bounding box coordinates. Let's see these parts in more details:

- **CNN backbone.** Choosing a Standard CNN Architecture (such as ResNet 18, ResNet 50, VGG, etc.) for Finetuning Pre-trained Models on Imagenet Classification Tasks. Enhancing the Backbone Network with Additional CNN Layers for Feature Map Size Reduction.

- **Vectorizer.** The output of the CNN backbone is a 3D tensor. But the resultant output of the Localizer is a 1D vector with four values corresponding to each coordinate for the bounding box. To convert the 3D tensor into a vector, we employ a vectorizer or utilize a Flatten layer as an alternative approach.

- **Regression Head.** We construct a fully connected regression head specifically for this task. After that, the feature vector, obtained from the backbone, is fed to the regression head. The regression head consists of 4 nodes at the end corresponding to the (x1, y1, x2, y2) or any other equivalent bounding box representations.

8.5 Region-based CNNs (R-CNNs)

We do not know how many regression head to train when dealing with multiple objects. We can apply a classification CNN as a sliding window detector. There are two problems to acheive this goal: first, we need to add a background class when fine-tuning the backbone on the detection dataset. Second, there are too many boxes to try. For a $w \times h$ window, there are $(W - w + 1) \times (H - h + 1)$ possible positions and we have to try all (or most of) the scales and aspect ratios. The solution to all that is **region proposals**.

Region proposal algorithms like Selective Search inspect the image and attempt to find regions of an image that likely contain an object. It first oversegments the image into highly uniform regions (i.e. "superpixels") then, based on similarity scores of color, texture and size iteratively aggregates them. The two most similar regions are grouped together, and new similarities are calculated between the resulting region and its neighbors, until the whole image becomes a single region. Each aggregation is a region.

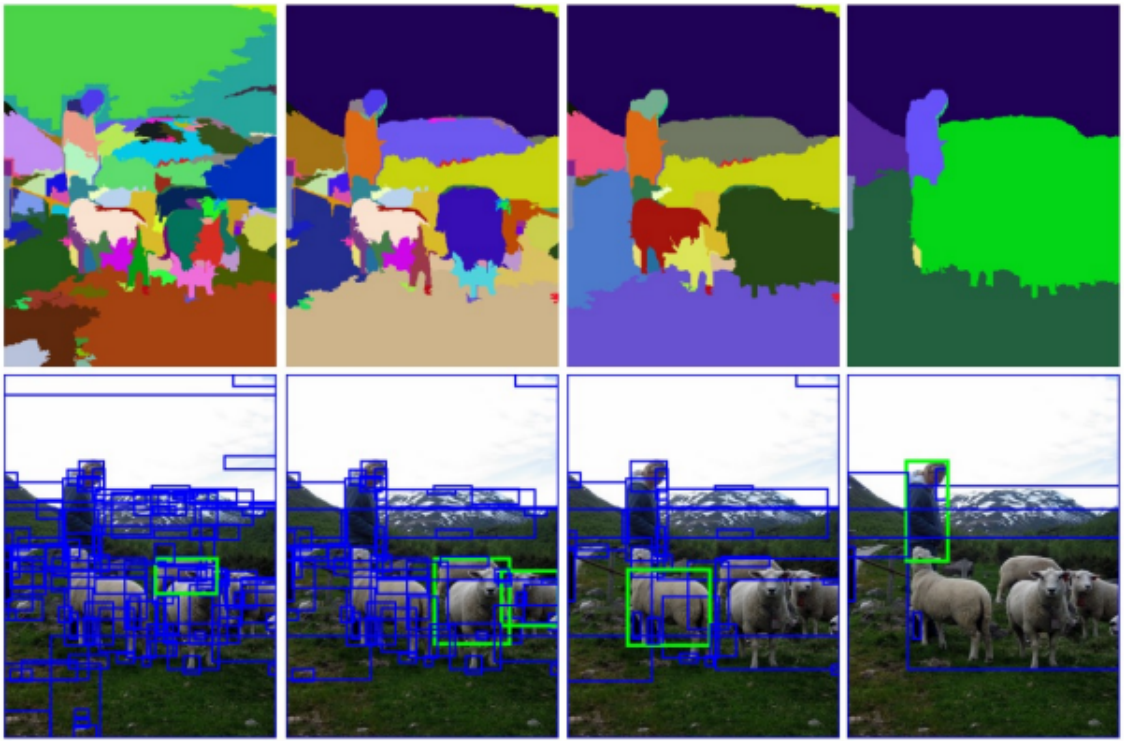


Figure 8.3: Region proposal example.

Region-based CNNs or regions with CNN features (R-CNNs) are also among many pioneering approaches of applying deep learning to object detection.

8.5.1 R-CNNs

The R-CNN first extracts many (e.g., 2000) region proposals from the input image (e.g., anchor boxes can also be considered as region proposals), labeling their classes and bounding boxes (e.g., offsets). Then a CNN is used to perform forward propagation on each region proposal to extract its features. Next, features of each region proposal are used for predicting the class and bounding box of this region proposal.

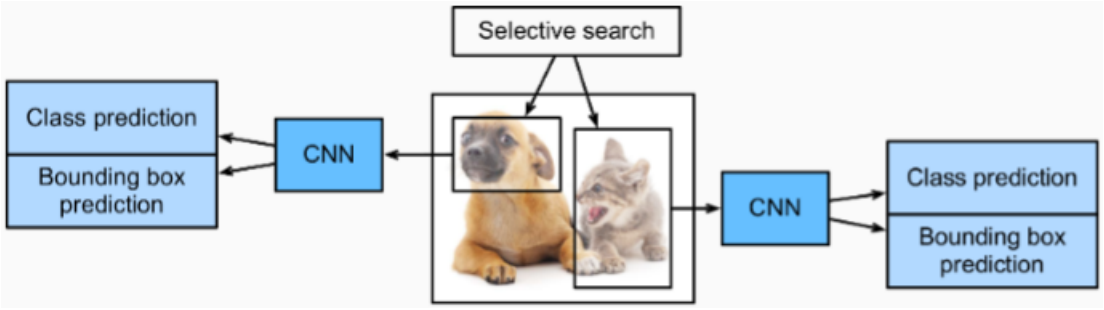


Figure 8.4: The R-CNN Model

Figure 8.4 shows the R-CNN model. More concretely, the R-CNN consists of the following four steps: it performs selective search to extract multiple high-quality region proposals on the input image. These proposed regions are usually selected at multiple scales with different shapes and sizes. Each region proposal will be labeled with a class and a ground-truth bounding box. Then, choose a pretrained CNN and truncate it before the output layer. Resize each region proposal to the input size required by the network, and output the extracted features for the region proposal through forward propagation. At this point, take the extracted features and labeled class of each region proposal as an example. Train multiple support vector machines to classify objects, where each support vector machine individually determines whether the example contains a specific class. Finally, take the extracted features and labeled bounding box of each region proposal as an example. Train a linear regression model to predict the ground-truth bounding box.

Although the R-CNN model uses pretrained CNNs to effectively extract image features, it is slow. Imagine that we select thousands of region proposals from a single input image: this requires thousands of CNN forward propagations to perform object detection. This massive computing load makes it infeasible to widely use R-CNNs in real-world applications.

8.5.2 Fast R-CNN

The main performance bottleneck of an R-CNN lies in the independent CNN forward propagation for each region proposal, without sharing computation. Since these regions

usually have overlaps, independent feature extractions lead to much repeated computation. One of the major improvements of the fast R-CNN from the R-CNN is that the CNN forward propagation is only performed on the entire image.

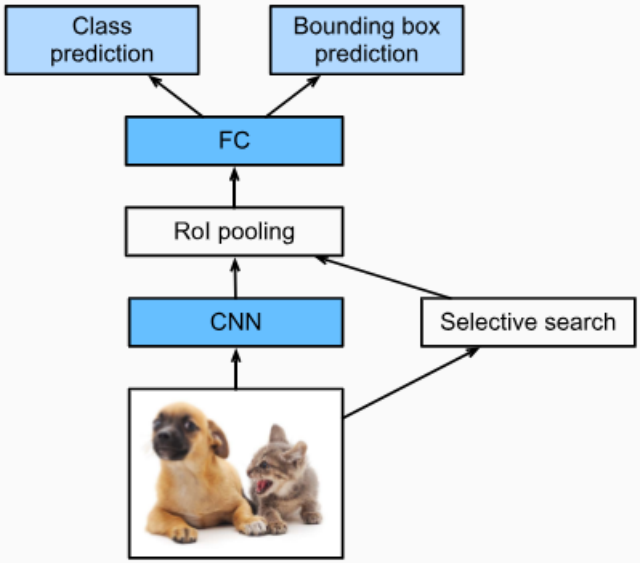


Figure 8.5: The fast R-CNN model

Figure 8.5 describes the fast R-CNN model. Its major computations are as follows:

1. Compared with the R-CNN, in the fast R-CNN the input of the CNN for feature extraction is the entire image, rather than individual region proposals. Moreover, this CNN is trainable. Given an input image, let the shape of the CNN output be $1 \times c \times h_1 \times w_1$.

2. Suppose that selective search generates n region proposals. These region proposals (of different shapes) mark regions of interest (of different shapes) on the CNN output. Then these regions of interest further extract features of the same shape (say height h_2 and width w_2 are specified) in order to be easily concatenated. To achieve this, the fast R-CNN introduces the region of interest *(RoI)* pooling layer: the CNN output and region proposals are input into this layer, outputting concatenated features of shape $n \times c \times h_2 \times w_2$ that are further extracted for all the region proposals.

3. Using a fully connected layer, transform the concatenated features into an output of shape $n \times d$, where d depends on the model design.

4. Predict the class and bounding box for each of the n region proposals. More concretely, in class and bounding box prediction, transform the fully connected

layer output into an output of shape $n \times q$ where q is the number of classes and an output of shape $n \times 4$, respectively. The class prediction uses softmax regression.

The region of interest pooling layer proposed in the fast R-CNN is different from the pooling layer seen previously. In the pooling layer, we indirectly control the output shape by specifying sizes of the pooling window, padding, and stride. In contrast, we can directly specify the output shape in the region of interest pooling layer. For example, let's specify the output height and width for each region as h_2 and w_2, respectively. For any region of interest window of shape $h \times w$, this window is divided into a $h_2 \times w_2$ grid of subwindows, where the shape of each subwindow is approximately $(h/h_2) \times (w/w_2)$.

In practice, the height and width of any subwindow shall be rounded up, and the largest element shall be used as output of the subwindow. Therefore, the region of interest pooling layer can extract features of the same shape even when regions of interest have different shapes. As an illustrative example, in Figure 8.6, the upper-left 3×3 region of interest is selected on a $4\times$ 4 input. FOr this region of interest, we use a 2×2 region of interest pooling layer to obtain a 2×2 output. Note that each of the four diveded subwindows contains element $0, 1, 4$ and 5 (5 is the maximum); 2 and 6 (6 is the maximum); 8 and 9 (9 is the maximum); and 10.

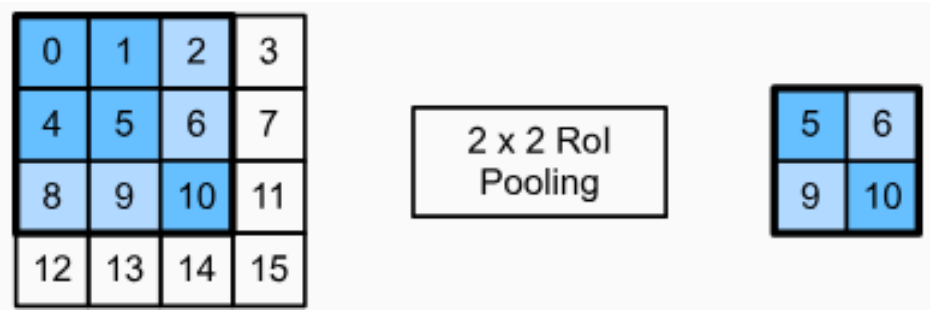


Figure 8.6: A 2×2 region of interest pooling layer

8.5.3 Faster R-CNN

To be more accurate in object detection, the fast R-CNN model usually has to generate a lot of region proposals in selective search. To reduce region proposals without loss of accuracy, the faster R-CNN proposes to replace selective search with a region proposal network.

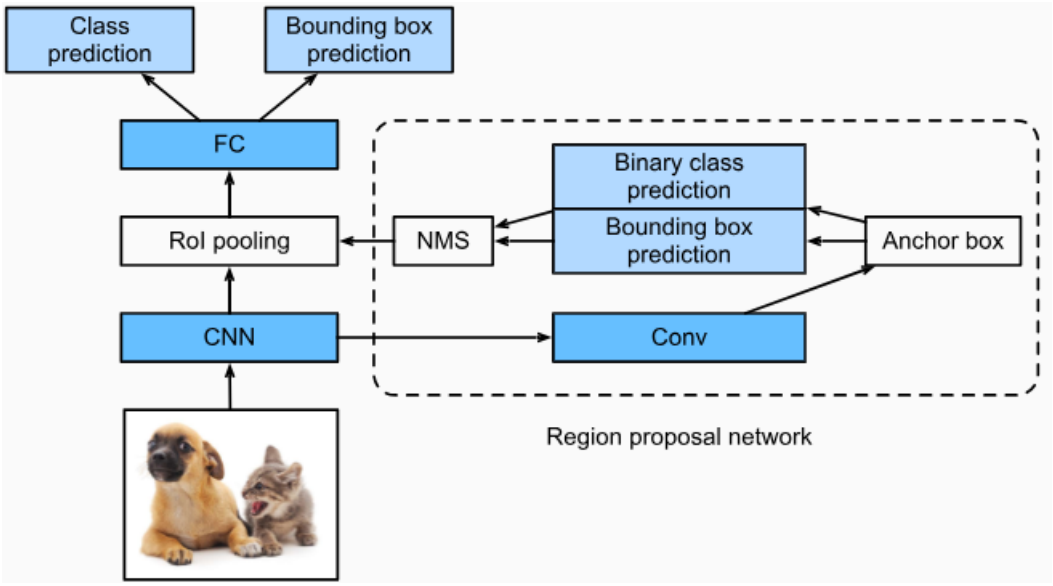


Figure 8.7: The faster R-CNN model

Figure 8.7 shows the faster R-CNN model. Compared with the fast R-CNN, the faster R-CNN only changes the region proposal method from selective search to a region proposal network. The rest of the model remain unchanged. The region proposal network works in the following steps:

1. Use a 3×3 convolutional layer with padding of 1 to transform the CNN output to a new output with c channels. In this way, each unit along the spatial dimensions of the CNN-extracted feature maps gets a new feature vector of length c.

2. Centered on each pixel of the feature maps, generate multiple anchor boxes of different scales and aspect ratios and label them.

3. Using the length-c feature vector at the center of each anchor box, predict the binary class (background or objects) and bounding box for this anchor box.

4. Consider those predicted bounding boxes whose predicted classes are objects. Remove overlapped results using non-maximum suppression. The remaining predicted bounding boxes for objects are the region proposals required by the region of interest pooling layer.

It is worth noting that, as part of the faster R-CNN model, the region proposal network is jointly trained with the rest of the model. In other words, the objective function of the faster R-CNN includes not only the class and bounding box prediction in object detection, but also the binary class and bounding box prediction of anchor boxes in the region proposal network. As a result of the end-to-end training, the region proposal

network learns how to generate high-quality region proposals, so as to stay accurate in object detection with a reduced number of region proposals that are learned from data.

8.6 Feature Pyramid Network (FPN)

Detecting objects in different scales is challenging in particular for small objects. We can use a pyramid of the same image at different scale to detect objects (the left diagram below). However, processing multiple scale images is time consuming and the memory demand is too high to be trained end-to-end simultaneously. Hence, we may only use it in inference to push accuracy as high as possible, in particular for competitions, when speed is not a concern. Alternatively, we create a pyramid of feature and use them for object detection (the right diagram). However, feature maps closer to the image layer composed of low-level structures that are not effective for accurate object detection.

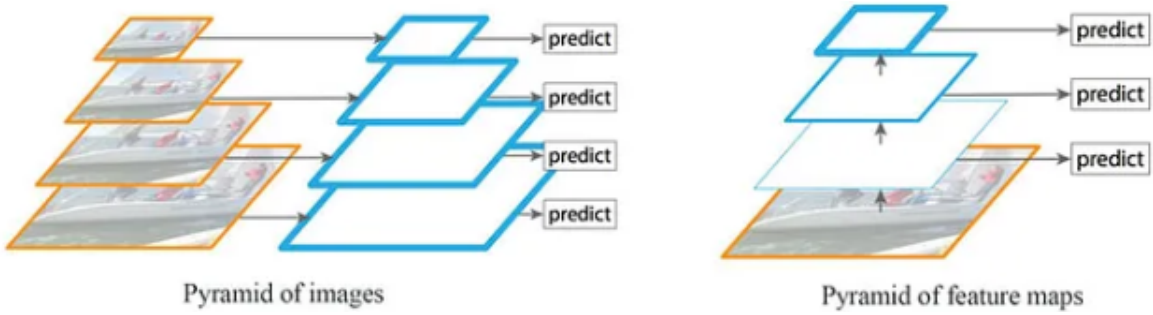


Figure 8.8: Left:Pyramid of images - Right: Pyramid of feature maps

Feature Pyramid Network (FPN) is a feature extractor designed for such pyramid concept with accuracy and speed in mind. It replaces the feature extractor of detectors like Faster R-CNN seen before and generates multiple feature map layers (multi-scale feature maps) with better quality information than the regular feature pyramid for object detection.

8.6.1 Data Flow

FPN composes of a bottom-up and a top-down pathway. The bottom-up pathway is the usual convolutionla network for feature extraction.

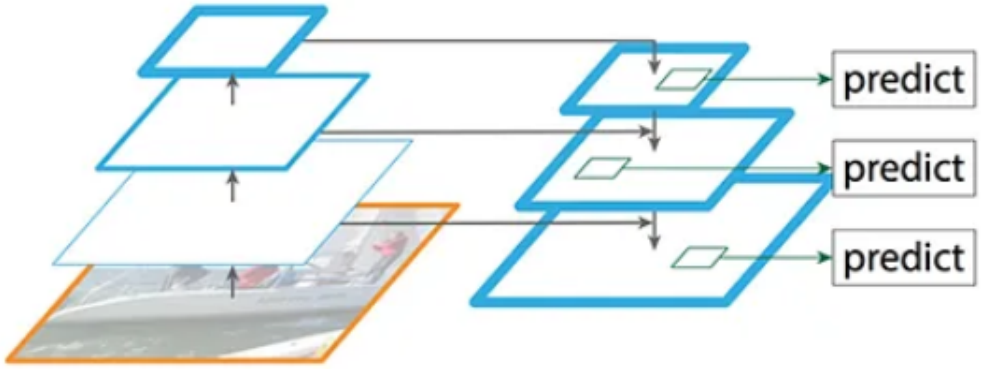


Figure 8.9: FPN Data Flow

As we go up, the spatial resolution decreases. With more high-level structures detected, the semantic value for each layer increases.

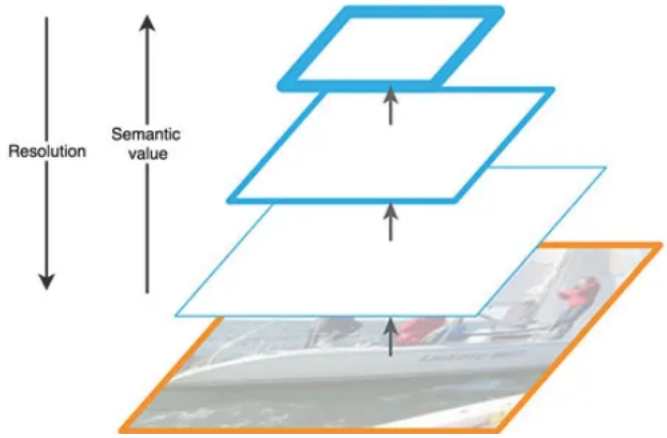


Figure 8.10: Feature extraction in FPN

SSD makes detection from multiple feature maps. However, the bottom layers are not selected for object detection. They are in high resolution but the semantic value is not high enough to justify its use as the speed slow-down is significant. So SSD only uses upper layers for detection and therefore performs much worse for small objects. FPN provides a top-down pathway to construct higher resolution layers from a semantic rich layer. While the reconstructed layers are semantic strong but the locations of objects are not precise after all the downsampling and upsampling. We add lateral connections between reconstructed layers and the corresponding feature maps to help the detector to predict the location betters. It also acts as skip connections to make training easier (similar to what ResNet does).

8.6.2 Bottom-up pathway

The bottom-up pathway uses ResNet to construct the bottom-up pathway. It composes of many convolution modules each has many convolution layers. As we move up, the spatial dimension is reduced by 1/2 (i.e. double the stride). The output of each convolution module is labeled as Ci and later used in the top-down pathway.

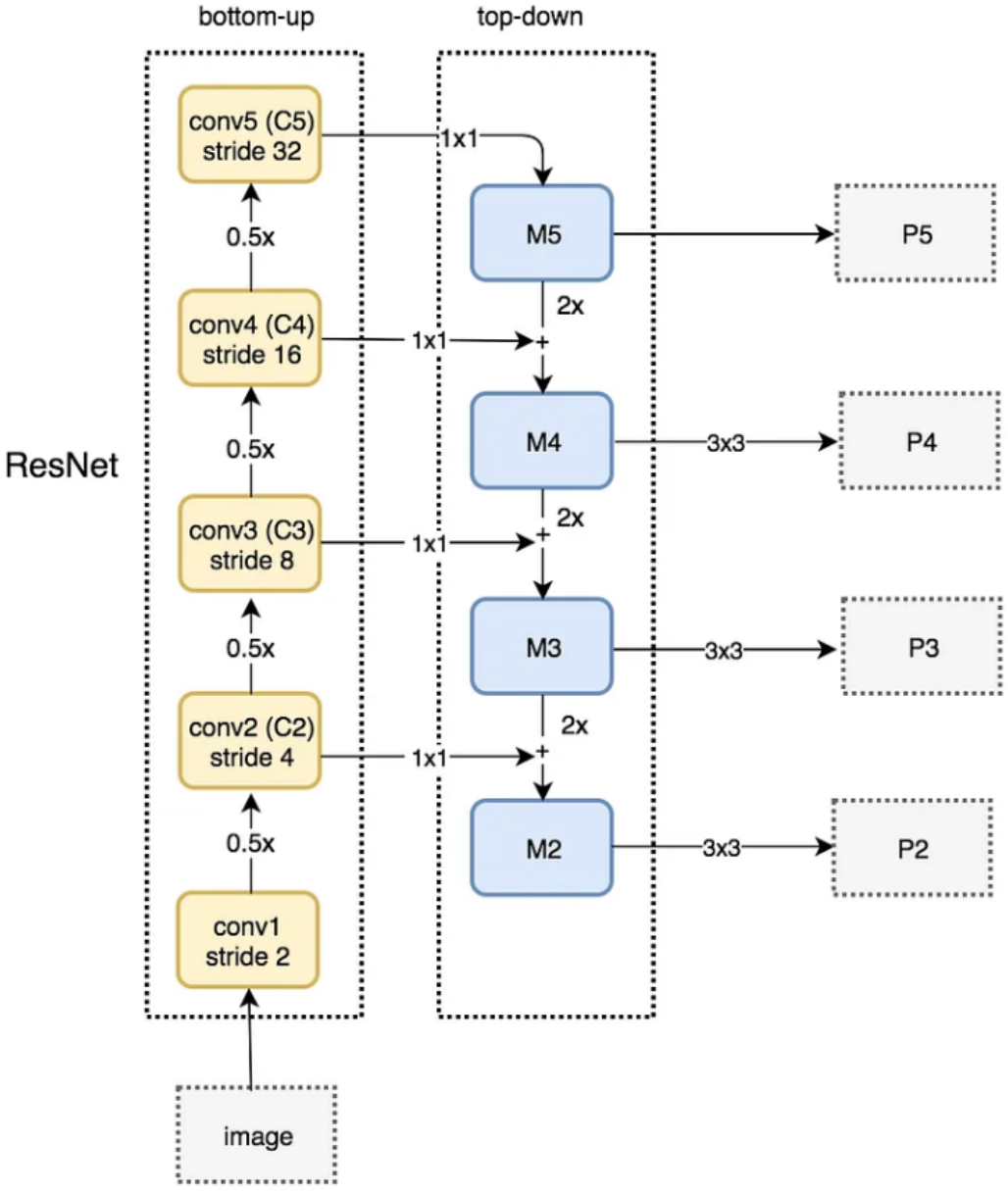
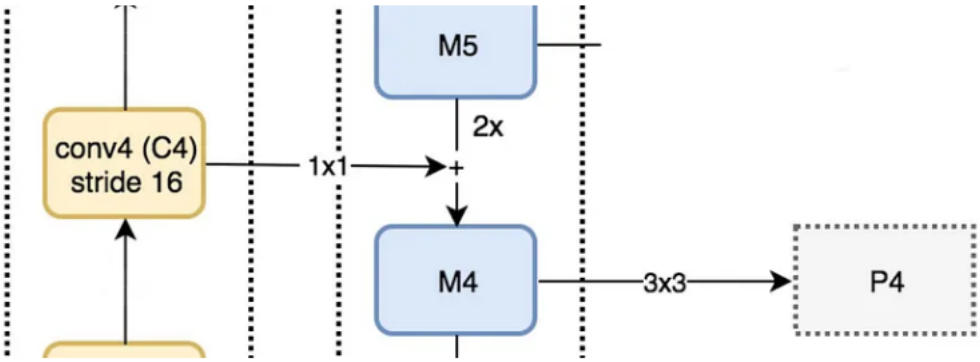


Figure 8.11: Bottom-Up Pathway architecture

8.6.3 Top-down Pathway

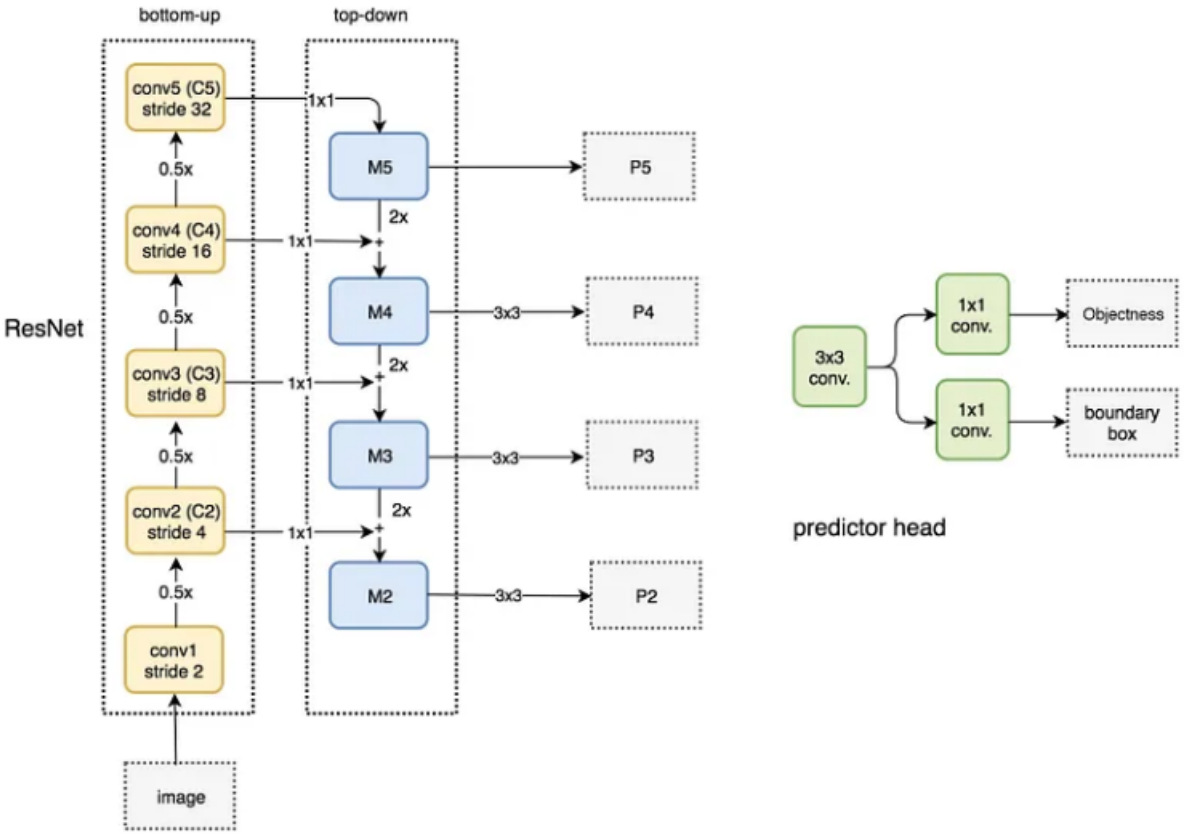
We apply a 1×1 convolution filter to reduce C5 channel depth to 256-d to create M5. This becomes the first feature map layer used for object prediction. As we go down the top-down path, we upsample the previous layer by 2 using nearest neighbors upsampling. We again apply a 1×1 convolution to the corresponding feature maps in the bottom-up pathway. Then we add them element-wise. We apply a 3×3 convolution to all merged layers. This filter reduces the aliasing effect when merged with the upsampled layer.



We repeat the same process for P3 and P2. However, we stop at P2 because the spatial dimension of C1 is too large. Otherwise, it will slow down the process too much. Because we share the same classifier and box regressor of every output feature maps, all pyramid feature maps (P5, P4, P3 and P2) have 256-d output channels.

8.6.4 FPN with RPN (Region Proposal Network)

. FPN is not an object detector by itself. It is a feature extractor that works with object detectors. FPN extracts feature maps and later feeds into a detector, for object detection. RPN applies a sliding window over the feature maps to make predictions on the objectness (has an object or not) and the object boundary box at each location. In the FPN framework, for each scale level (say P4), a 3×3 convolution filter is applied over the feature maps followed by separate 1×1 convolution for objectness predictions and boundary box regression. These 3×3 and 1×1 convolutional layers are called the RPN head. The same head is applied to all different scale levels of feature maps. Check the Figure below to visualize it.



8.7 Single Shot MultiBox Detector (SSD)

This model is simple, fast, and widely used. Although this is just one of vast amounts of object detection models, some of the design principles and implementation details in this section are also applicable to other models.

8.7.1 Model

The Figure 8.12 provides an overview of the design of single-shot multibox detection. This model mainly consists of a base network followed by several multiscale feature map blocks. The base network is for extracting features from the input image, so it can use a deep CNN. For example, the original single-shot multibox detection paper adopts a VGG network truncated before the classification layer, while ResNet has also been commonly used. Through our design we can make the base network output larger feature maps so as to generate more anchor boxes for detecting smaller objects. Subsequently, each multiscale feature map block reduces (e.g., by half) the height and width of the feature maps from the previous block, and enables each unit of the feature maps to increase its

receptive field on the input image. Since multiscale feature maps closer to the top of Figure 8.12 are smaller but have larger receptive fields, they are suitable for detecting fewer but larger objects.

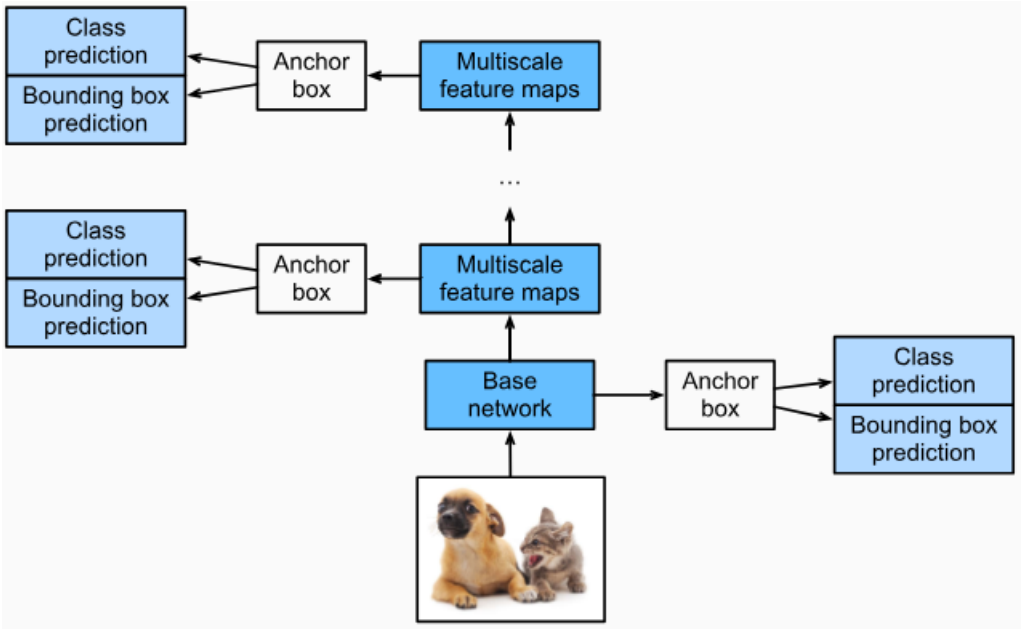


Figure 8.12: As a multiscale object detection model, single-shot multibox detection mainly consists of a base network followed by several multiscale feature map blocks.

In a nutshell, via its base network and several multiscale feature map blocks, single-shot multibox detection generates a varying number of anchor boxes with different sizes, and detects varying-size objects by predicting classes and offsets of these anchor boxes (thus the bounding boxes); thus, this is a multiscale object detection model.

In the following, we will describe the implementation details of different blocks present in the Figure above.

8.7.2 Class Prediction Layer

Let the number of object classes be q. Then anchor boxes have $q + 1$ classes, where class 0 is background. At some scale, suppose that the height and width of feature maps are h and w, respectively. When a anchor boxes are generated with each spatial position of these feature maps as their center, a total of hwa anchor boxes need to be classified. This often makes classifciation with fully connected layers infeasible due to likely heavy parametrization costs. Recall how we used channels of convolutional layers to predict classes, single-shot multibox detection uses the same technique to reduce model complexity.

Specifically, the class prediction layer uses a convolutional layer without altering width or height of feature maps. In this way, there can be a one-to-one correspondence between outputs and inputs at the same spatial dimensions (width and height) of feature maps. More concretely, channels of the output feature maps at any spatial position (x, y) represent class predictions for all the anchor boxes centered on (x, y) of the input feature maps. To produce valid predictions, there must be $a(q + 1)$ output channels, where for the same spatial position the output channel with index $i(q + 1) + j$ represents the prediction of the class $j(0 \leq j \leq q)$ for the anchor box $i(0 \leq i < a)$.

8.7.3 Bounding Box Prediction Layer

The design of the bounding box prediction layer is similar to that of the class prediction layer. The only difference lies in the number of outputs for each anchor box: here we need to predict four offsets rather than $q + 1$ classes.

Chapter 9

Segmentation

The problem definition of a segmentation task is pretty simple, even the idea behind it's simple. As input we have a RGB image of size $W \times H$ and as output we have a category c_{uv} for each pixel $p = (u, v), c_{uv} \in [1, ..., C]$, (a fixed list of categories, as in image clasifciation). So, for each pixel in the image, we try to classify them in a category. That's the idea.

What is Semantic Segmentation? The process of linking each pixel in an image to a class label is refereed to as semantic segmentation. In the case, precise pixel level understanding of the environment is critical. See the Figure below.

Figure 9.1: Sematinc segmentation in outdoor environment

Now a days, semantic segmentation is more popular in various field. especially Biomedical(Breast, Brain cancer etc..), Self driving car, Traffic system, Agriculture, Aerial(Drone) sector etc. And also lot of data available in free resource: COCO, Pascal Voc, Cityscapes, and others. Let's go deep in the architecture now.

9.1 U-Net architecture

One of the approaches used in image segmentation is the Encoder-Decoder Method. This method is represent by U-Net. U-Net architecture designed in "U-shaped", So its represent U-Net. In this architecture, a fully connected convolutional layer is modified so as to work on a few training images yet yielding more accurate precision. This is important since this architecture was initially proposed for medical image processing where lots of training data is scarce. The model applies elastic deformations on the available data in order to achieve augmentation.

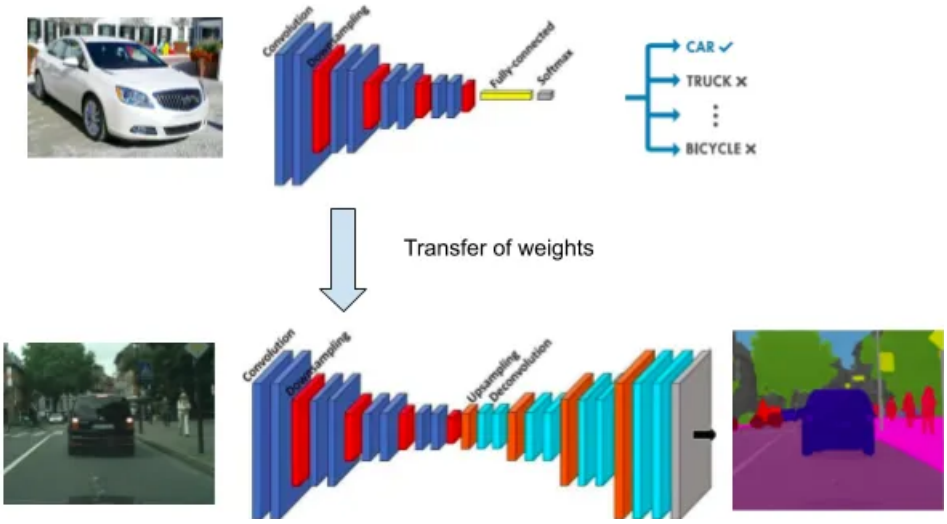


Figure 9.2: Real example of U-Net

This is a network whose training strategy relies on heavy use of data augmentation in order to use the annotated images efficiently. The architecture is made up of: Encoder and Decoder

1. **Encored:** Encoder is represent in Contracting path. And main purpose is captures context. That means Extract features from an image Classification.

2. **Decoder:** Decoder is represent symmetric expanding path whose role is to enable precise localization (Encoder knowledge can be transferred, during the process of image segmentation). Ensure that the generated mask bears resemblance to the

pixel resolution of the input image. Decoder used in different approach: Region based , fully convolutional network based and weakly supervised semantic segmentation.

3. **IoU:** IOU is a common metric used to evaluate the performance of segmentation models. This metric is computed from the ground truth mask and the predicted mask. (Same metric seen also in object detection).

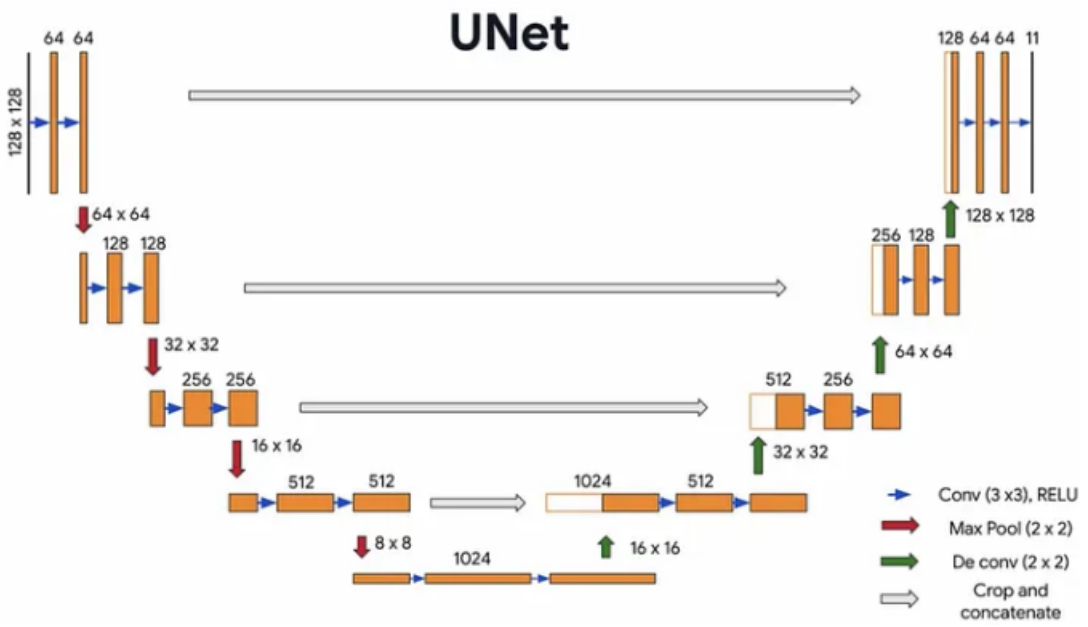


Figure 9.3: U-Net Architecture

9.2 Dilated convolutions

Dilated Convolution is like exploring the input data points in a wide manner or increasing the receptive field of the convolution operation. In simple terms, dilated convolution is just a convolution applied to input with defined gaps. With these definitions, given our input is an 2D image, dilation rate k=1 is normal convolution and k=2 means skipping one pixel per input and k=4 means skipping 3 pixels. The best to see the figures below with the same k values. The figure below shows dilated convolution on 2D data. Red dots are the inputs to a filter which is 3×3 in this example, and green area is the receptive field captured by each of these inputs. Receptive field is the implicit area captured on the initial input by each input (unit) to the next layer.

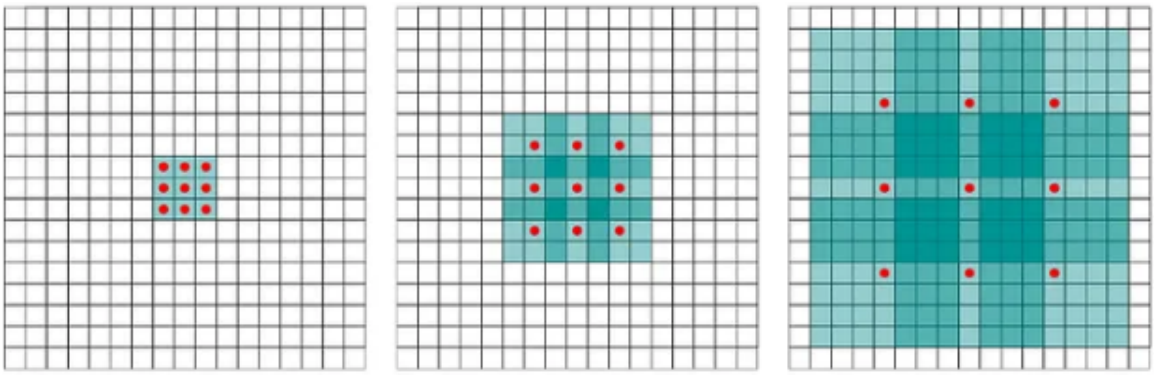


Figure 9.4: Systematic dilation supports exponentaial expansion of the receptive filed without loss of resolution or coverage. Let's call the subimages (a), (b) and (c) from left to right. (a) F_1 is produced from F_0 by a 1-dilated convolution; each element in F_1 has a receptive field of 3×3. (b) F_2 is produced from F_1 by a 2-dilated convolution; each element in F_2 has a receptive field of 7×7. (c) F_3 is produced from F_2 by a 4-dilated convolution; each element in F_3 has a receptive field of 15×15. The number of parameters associated with each layer is identical. The receptive field grows exponentially while the number of parameters grows linearly.

This is more of normal convolution, but help to capture more and more global context from input pixels without increasing the size of the parameters. This can also help to increase the spacial size of the output. But the main thing here is this increases the receptive field size exponentially with the number of layers. This is very common in the signal processing field. Dilated convolution is a way of increasing receptive view (global view) of the network exponentially and linear parameter accretion. With this purpose, it finds usage in applications cares more about integrating knowledge of the wider context with less cost. One general use is image segmentation where each pixel is labelled by its corresponding class. In this case, the network output needs to be in the same size of the input image. Straight forward way to do is to apply convolution then add deconvolution layers to up sample. However, it introduces many more parameters to learn. Instead, dilated convolution is applied to keep the output resolutions high and it avoids the need of up sampling.

9.3 DeepLab Models

DeepLab v1, v2, v3 models are widely used. The following section is made for trying to summarize their principles and sort out their relationships.

9.3.1 DeepLab V1

Compared with semantic segmentation, relevant models based on traditional CNN proposed earlier are more suitable for image classification, target recognition and other tasks, mainly because classical CNN has two good features:

1. **Abstraction of features:** in the classification task, only the high-level abstract features of the object are needed, and the local information such as texture and series at the bottom is not paid much attention to. However, CNN's ability to abstract features can meet this requirement.

2. **Spatial invariance of convolution:** spatial invariance means that it is insensitive to the transformation of the image. In image classification, the spatial transformation of image does not cause the change of category. However, the classical CNN structure has translation invariance.

These two features of CNN enable it to achieve better results in problems such as image classification. However, semantic segmentation is a pixel-level classification problem that is more accurate than image classification. However, the two features of classical CNN will affect the effect of semantic segmentation.

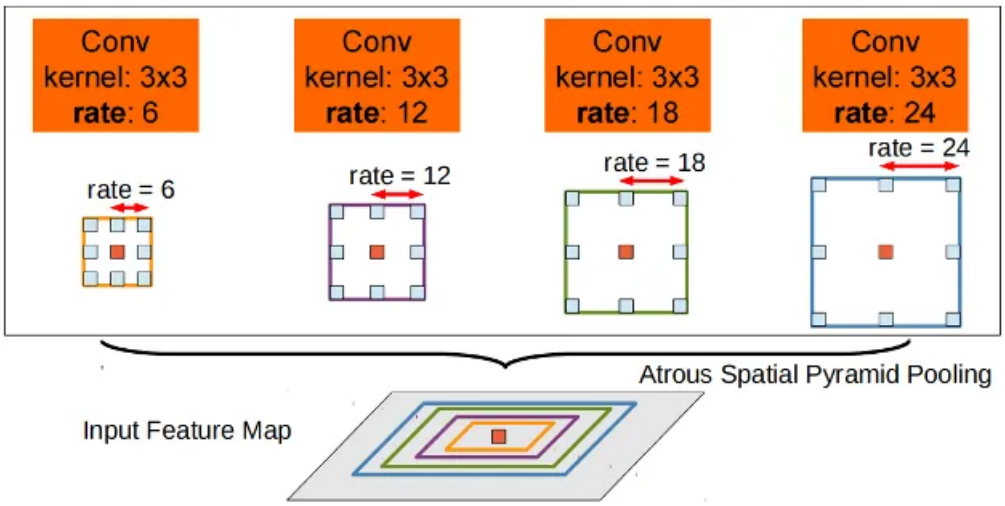
First of all, for the abstraction of features in CNN, it is equivalent to using operations such as maximum pooling to downsample the image. In this process, a lot of pixel information will disappear, thus causing the problem of inaccurate classification in pixel level classification. In DeepLab v1, the dilated convolution method is introduced to solve this problem. By adding gaps between each pixel on the convolution kernel, a larger receptive field can be obtained with the same parameter number and computation amount as before, so that information loss caused by downsampling will not appear. (as seen before). Secondly, for the spatial invariance characteristics of CNN, in semantic segmentation, small transformations of the image may all lead to great changes in the image segmentation results. In the classical CNN structure, as the number of network layers deepens, the position information is lost by the last few layers, which cannot be precisely located and causes the problem of inaccurate semantic segmentation. Therefore, in DeepLab v1, the proposed solution is to add a fully connected conditional random field (CRF) at the end of the convolutional layers to solve the problem of missing position information and inaccurate localization in deep convolutional networks.

The backbone of DeepLab v1 uses the VGG-16 structure and converts the fully connected layers in VGG-16 to the convolutional layers to form a fully convolutional network, just like the FCN network. Compared with the classical VGG-16 structure, DeepLab v1 modifies the parameters of the maximum pooling layer to change the original 32 times downsampling to 8. The last three convolutional layers in the structure are converted into dilated convolution with a dilation factor of 3.

9.3.2 DeepLab V2

Based on DeepLab v1, the main improvement points of DeepLab v2 include three points. Firstly, DeepLab v2 replaces the backbone network from VGG-16 to ResNet, which improves part of the segmentation effect. Secondly, the learning rate setting is adjusted. Finally, DeepLab v2 proposes the Atrous Spatial Pyramid Pooling(ASPP structure) to address the problem of multi-scale targets. Compared with the previous method that only scales the image to multiple scales to achieve multi-scale inferencing through networks separately and finally fusing multiple results, the ASPP module greatly reduces the computational effort and can better segment objects of different sizes.

For the general superposition of the dilated convolution, although its perceptual field is enlarged by adding gaps in the convolution kernel, the enlarged convolution kernel is more suitable for the segmentation of larger objects, while the size of the same object may vary greatly in the actual image. Therefore, a parallel structure of dilated convolution is used in the ASPP module, and the structure diagram is shown in following image. Four dilated convolution branches are connected in parallel to the output feature map of the backbone network, and the output feature map is processed by using dilated convolution with different dilation factors respectively. For each branch, it has different perceptual fields and the number of convolution kernels is equal to the number of semantic classes, and finally the computational results of each branch are summed and fused to achieve the target multi-scale fusion.



Following the Figure below, the structure of DeepLabV2 with ResNet101 as the backbone netowrk. Compared with the classical ResNet101 structure, DeepLab V2 eliminates the downsampling in layer3 and layer4, changes the stride of the convolutional layer from 2 to 1, and converts the convolutional layer to dilated convolution.

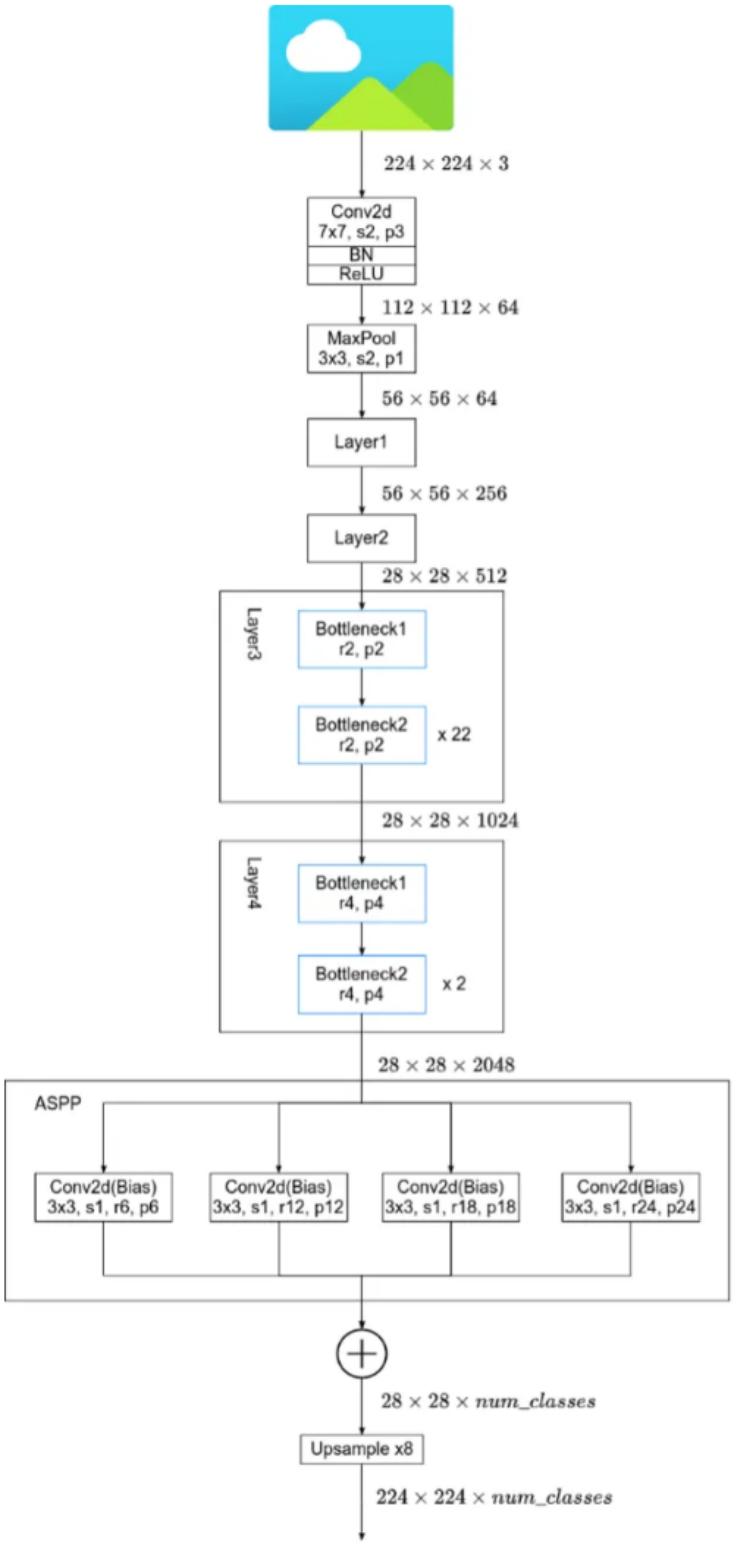
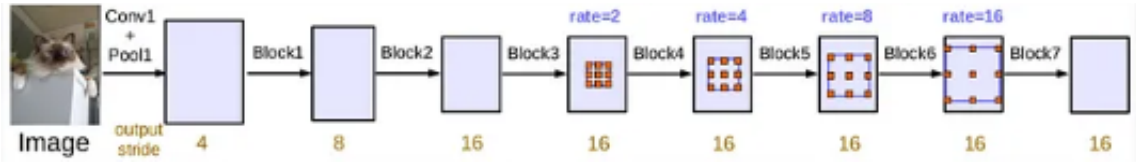


Figure 9.5: DeepLab v2 with ResNet101 architecture

9.3.3 DeepLab V3

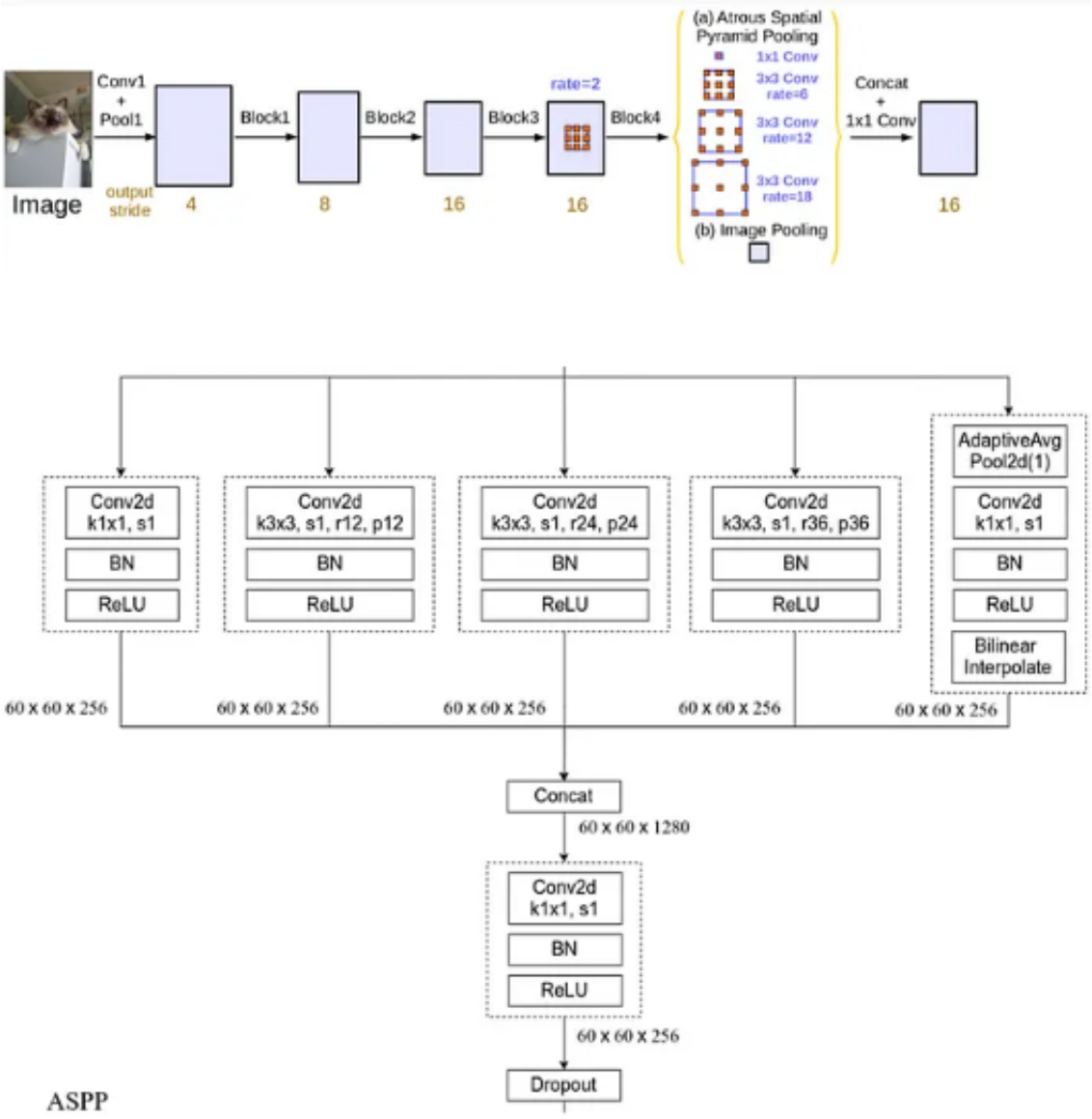
Since the dilated convolution has achieved good semantic segmentation results in the previous DeepLab models, DeepLab v3 continues to use this method and proposes some improvements for the segmentation of multi-scale targets. Firstly, DeepLab v3 is designed respectively with a series structure and a parallel structure model using dilated convolution, and adopts various methods of setting dilation coefficients. Secondly, it follows the ASPP structure in the v2 model, on which the convolution layer structure is improved and the batch normalization layer (BN) is added. Finally, the time-consuming post-processing layer of CRFs is removed in DeepLab v3.

For the proposed tandem model, the structure is shown in following figure. Block1 to Block4 are the layer structures of the original ResNet network, where the stride of the convolutional layer is modified in Block4 so that it is no longer downsampled, and the convolutional layer is replaced with a dilated convolutional layer. Block5 to Block7 are the new added layer structures in DeepLab v3, and their structures are exactly the same as Block4. In this structure, the concept of Multi-grid is introduced. In the previously proposed DeepLab structure, there has been a lack of specification and systematic study on the setting of dilation coefficients. In DeepLab v3, experiments and investigations were conducted, and the method of setting the dilation coefficient when using different numbers of blocks was determined after extensive experiments. The actual dilation coefficient used in the model is the product of the rate in following figure and its corresponding Multi-grid. Since a block contains three dilated convolutions, the Multi-grid corresponding to each block is a three-dimensional vector. For example, if block4 has rate=2 and Multi-grid=(1,2,4), the three convolution layers in block4 use dilation coefficients of 2,4,8, respectively, and the best results are found by the relevant experiments in DeepLab v3 when using block5 to block7 with Multi-grid=(1,2,1). And the best result is when Multi-grid=(1,2,4) without adding several modules from block5 to block7.



For the parallel model, the structure is shown in following figure, using a similar ASPP structure as in DeepLab v2, but the model of v3 is further optimized for ASPP, as shown in next figure. The ASPP in the model contains five parallel branches, which are a 1x1 convolutional layer, three 3x3 dilated convolutional layers, and a global average pooling layer. Among them, the global average pooling branch serves to add full-text contextual information. The outputs of the five branches are then stitched along the channel direction by means of splicing, and finally the 1x1 convolutional layer is used for

further information fusion.



9.4 Image Segmentation and Instance Segmentation

There are two important tasks in the field of computer vision that are similar to semantic segmnetation, namely image segmentation and instance segmentation. Let's see the difference:

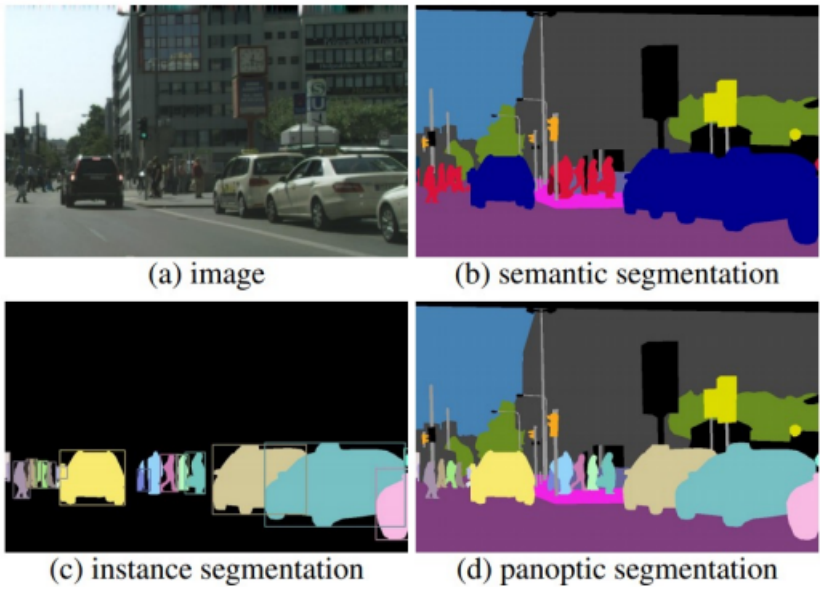
- **Image segmentation:** divides an image into several constitutent regions. The

methods for this type of problem usually make use of the correlation between pixels in the image. It does not need label information about image pixels during training, and it cannot guarantee that the segmented regions will have the semantics that we hope to obtain during prediction.

- **Instance segmentation:** is also called simultaneous detection and segmentation. It studies how to recognize that pixel-level regions of each object instance in an image. Different from semantic segmentation, instance segmentation needs to distinguish not only semantics, but also different object instances. For example, if there are two dogs in the image, instance segmnetation needs to distinguish which of the two dogs a pixel belongs to.

9.5 Stuff and Things

- **Things:** countable object categories, that can be separated into individual instances, such as cars, persons, chairs. The subject of object detection and instance segmentation.
- **Stuff:** uncountable amorphous regions of similar texture or material such as grass, sky, road. Semantic segmentation recognize stuff categories but treat also "things" as "stuff".
- **Panoptic segmentation** unifies them. It requires to label each pixel with a category, and for things categories, also with an instance id


(a) image (b) semantic segmentation
(c) instance segmentation (d) panoptic segmentation

Chapter 10

Metric Learning

Can we use Image Classification to solve the task of Image Verification? We might be able to learn a quite robust Deep Convolutional Neural network that performs excellently on classifying all the employee images in an organization and also takes into account factors like poses, expression, and illumination (just as example). But, this is usually achieved when we have a good amount of data. By good amount I mean 1000's of examples for each class/employee. In Image Classification, if the number of data points per class is small, it might lead to overfitting and yield very poor results. Also, Image Classification generally works well when the number of classes is small. However, It is generally not the case with Person/Image verification tasks. In fact, it is quite opposite. Here, we usually have a very large number of classes and the number of examples per class is quite small. And this is where one-shot learning comes into the picture. One-shot Learning is a classification problem that aims to learn about object categories from one/few training examples/images. In simple words, given just one example/image of a person, you need to recognize him/her. To build a face recognition system, we need to solve this one-shot learning problem.

But deep neural nets usually require vast amounts of data to train on to excel at a particular task, which is not always available. Deep learning models won't work well with just one training example, one-shot learning problem. How to address this issue? We learn a similarity function, which helps us to solve the one-shot learning problem.

10.1 Similarity function

First of all we have to understand what is a metric. A metric is a non-negative function between two points x and y, say $g(x, y)$, that describes the so-called notion of **distance** between these two points. There are several properties that a metric must satisfy:

- **Non-negativity:** $d(x, y) \geq 0$ and $d(x, y) = 0$, iff $x = y$

- **Triangular inequality:** $d(x, y) \leq d(x, z) + d(z, y)$
- **Symmetry:** $g(x, y) = g(y, x)$

What does a basic machine learning algorithm do? — Given the data and the corresponding output labels, the goal is to come up with a set of rules or some complex function that maps those inputs to the corresponding output labels. One of the simplest machine learning algorithms where distance information is captured is the KNN (k- Nearest Neighbours) algorithm, where the idea is to find a neighborhood of k nearest data points for a new data point and assign this data point to the class to which the majority of the k data points belong.

Similarly, the goal of metric learning is to learn a similarity function from data. Metric Learning aims to learn data embeddings/feature vectors in a way that reduces the distance between feature vectors corresponding to faces belonging to the same person and increases the distance between the feature vectors corresponding to different faces.

10.2 Siamese Network

One of the very fundamental ideas where explicit metric learning is performed is the siamese network model. Siamese network is a Symmetric neural network architecture consisting of two identical subnetworks that share the same sets of parameters (hence computing the same function) and learns by calculating a metric between highest level feature encodings of each subnetwork each with a distinct input.

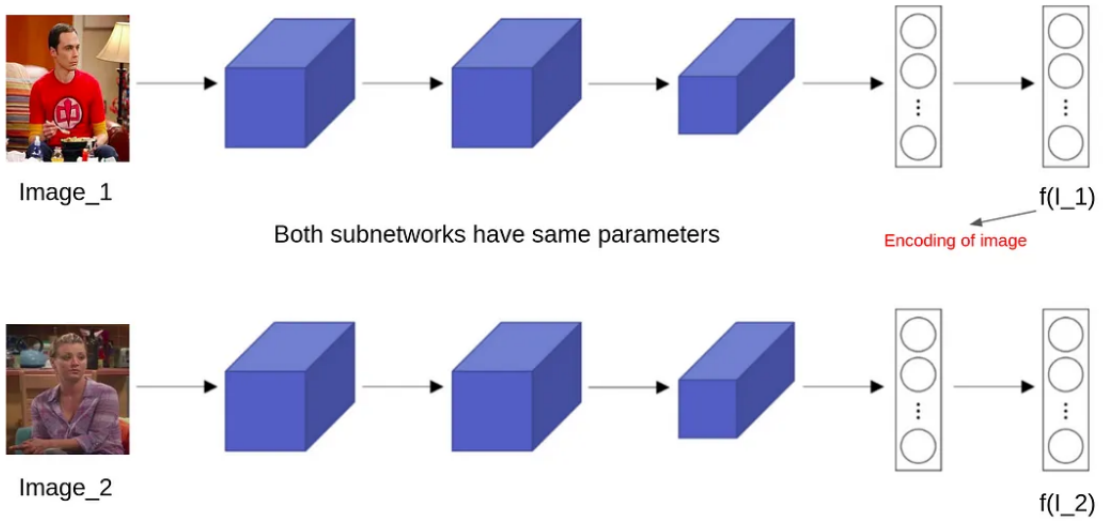


Figure 10.1: Example of Siamese Network with two images.

Mathematically, $d(x, y) = ||f(x) - f(y)||^2$, where x and y represent the training example/images, $d(.)$ represents the metric distance between two images and f represents the encoding of the input images. The goal is to learn parameters so that: if x and y are the same person, $d(x, y)$ is small and viceversa is x and y are different person.

Siamese Networks are mainly used in combination with a Contrastive Loss function. The loss function is mainly used to learn embeddings (feature vectors) in a way that the metric distance between two examples from the same class is small and that between different classes is large in a metric space.

10.3 Contrastive loss function

Contrastive loss is one of the earliest training objectives used for deep metric learning in a contrastive fashion. Given a list of input samles $\{x_i\}$, each has a corresponding label $y_i \in \{1, ..., L\}$ among L classes. We would like to learn a function $f_\theta(.) : X \to \mathbb{R}^d$ that encodes x_i into an embedding vector such that examples from the same class have similar embeddings and samples from different classes have very different ones. Thus, contrastive loss takes a pair of inputs (x_i, x_j) and minimizes the embedding distance when they are from the same class but maximizes the distance otherwise.

$$\mathcal{L}_{cont}(x_i, x_j, \theta) = \mathbb{1}[y_i = y_j]||f_\theta(x_i) - f_\theta(x_j)||_2^2 + \mathbb{1}[y_i \neq y_j] \max(0, \epsilon - ||f_\theta(x_i) - f_\theta(x_j)||_2)^2$$

$\mathbb{1}$ is an indicator function, which is 1 if $y_1 = y_j$ and 0 viceversa. ϵ is a hyperparameter, defining the lower bound distance between samples of different classes.

10.4 Triplet loss

Triplet loss was originally proposed in the FaceNet paper and was used lo learn face recognition of the same person at different poses and angles.

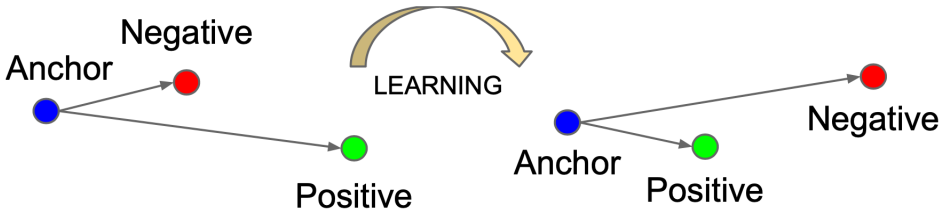


Figure 10.2: Illustration of triplet loss given one positive and one negative per anchor

Given one anchor input x, we select one positive sample x^+ and one negative x^-, meaning that the x^+ and x belong to the same class and x^- is sampled from another

different class. Triplet loss learns to minimize the distance between the anchor x and the positive ex^+ and maximize the distance between anchor x and the negative x^- at the same time with the following equation:

$$\mathcal{L}_{triplet}(x, x^+, x^-) = \sum_{x \in \mathcal{X}} \max(0, ||f(x) - f(x^+)||_2^2 - ||f(x) - f(x^-)||_2^2 + \epsilon)$$

where the margin parameter ϵ is configured as the minimum offset between distances of similar vs dissimilar pairs. It is crucial to select challenging x^- to truly improve the model. It is the most important part in training embedding with the triplet loss.

Chapter 11

Transformer

Before dig into transformer and attention mechanism, we must understand the Recurrrent neural networks and how embeddings work.

11.1 Recurrent Neural Network (RNN)

A sequence model is usually designed to transform an input seuqnece into an output sequence that lives in a different domain. Recurrent neural network, short in RNN, is suitable for this purpose and has shown improvement in problem like handwriting recognition, speech recognition and machine translation.

A recurrent neural network model is born with the capability to process long sequential data and to tackle tasks with context spreading in time. The model processes one element in the sequence at one time step. After computation, the newly updated unit state is passe down to the next time step to facilitate the computation of the next element. Imagine the case when an RNN model reads all the Wikipedia articles, character by character, and then it can predict the following words given the context.

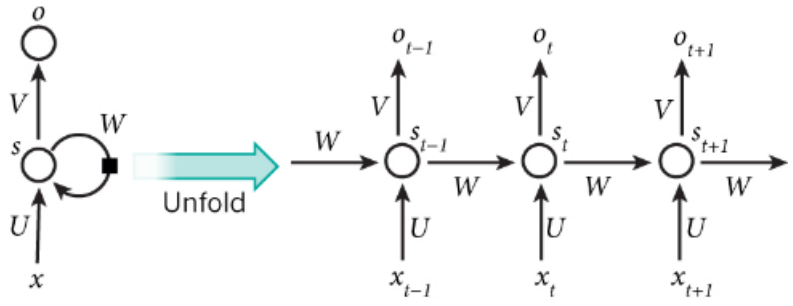


Figure 11.1: A recurrent neural network with one hidden unit (left) and its unrolling vesion in time (right). The unrolling version illustrates what happens in time: $s_t - 1, s_t$ and $s_t + 1$ are the same unit with different states at different time step $t - 1, t, t + 1$

However, simple perceptron neurons that linearly combine the current input element and the last unit state may easily lose the long-term dependencies. For example, we start a sentence with *"Alice is working at..."* and later after the whole paragraph, we want to start the next sentence with *"She"* or *"He"* correctly. If the model forgets the character's name *"Alice"*, we can never know. To resolve the issue, researchers created a special neuron with a much more complicated internal structure for memorizing long-term context, named **"Long-short term memory (LSTM)"** cell. It is smart enough to learn for how long it should memorize the old information, when to forget, when to make use of the new data, and how to combine the old memory with new input.

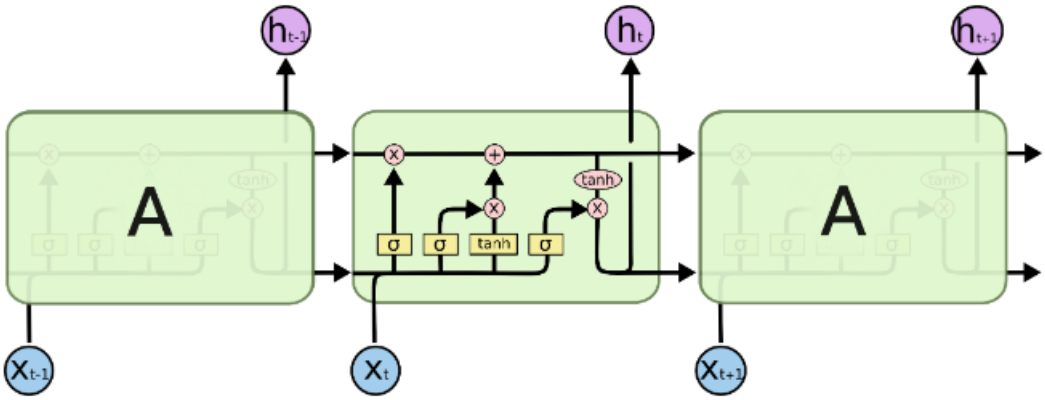


Figure 11.2: The repeating module in an LSTM contains four interacting layers

11.2 RNN: Sequence-to-Sequence Model

The sequence-to-sequence model is an extended version of RNN. Same as RNN, a sequence-to-sequence model operates on sequential data. It consists of two RNNs, encoder and decoder. The encoder learns the contextual information from the input words and then hands over the knowledge to the decoder side through a *context vector* (or *thought vector, as shown in the Figure 11.3*. Finally, the decoder consumes the context vector and generates proper responses.

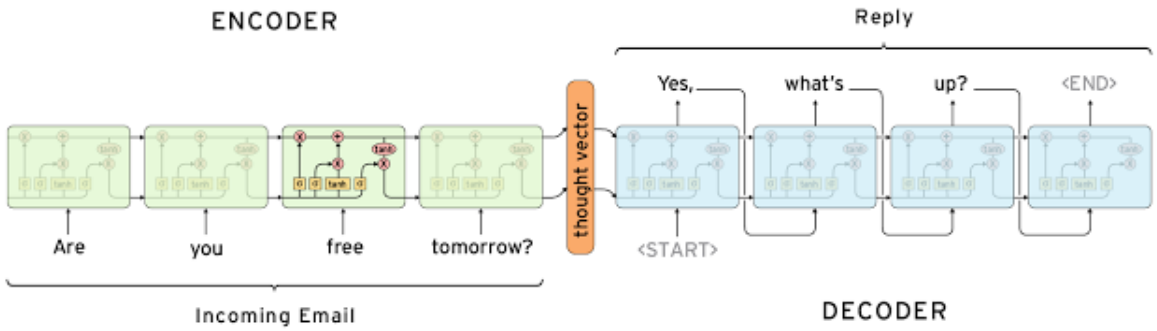


Figure 11.3: A sequence-to-sequence model for generating email auto replies.

The input and output lengths can vary vrom each other. An encoder emits the context C, usually as a function of its final hidden state. A decoder is conditioned on that fixed-length vector to generate the output sequence. If the context C is a vector, the the encoder RNN is a simply a **sequence-to-vector** RNN. If the contect C is a vector, then the decoder RNN is a **vector-to-sequence** RNN.

A critical and apparent disadvantage of this fixed-length context vector design is incapability of remembedering long sentences. Often it has forgotten the first part once it completes processing the whole input. The attention mechanism was bord to resolve this problem.

11.3 Attention mechanism

The attention mechanism was born to help memorize long source sentences in neural machine translation (NMT). Rather than building a single context vector out of the encoder's last hidden state, the secret sauce invented by attention is to create shortcuts between the context vector and the entire source input. The weights of these shortcut connections are customizable for each output element.

While the context vector has access to the entire input sequence, we do not need to worry about forgetting- The alignment between the source and target is learned and controlled by the context vector. Essentially the context vector consumes three pieces of information:

- Encoder hidden states.
- Decoder hidden states.
- alignment between source and target.

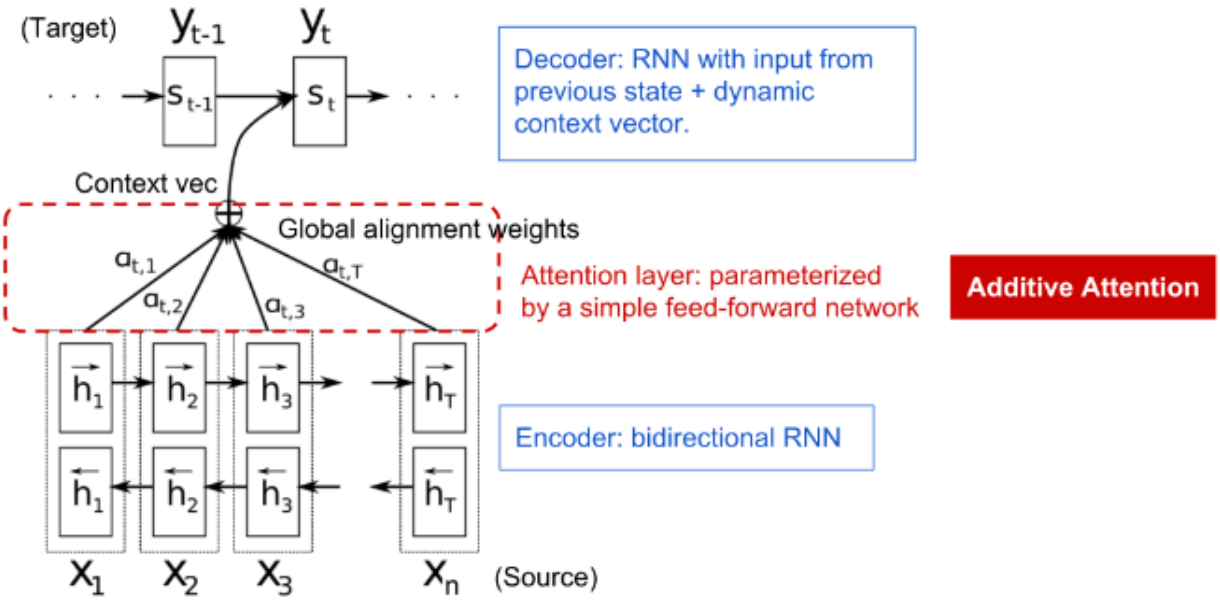


Figure 11.4: The encoder-decoder model with additive attention mechanism in

11.3.1 Definition

Let's define the attention mechanism introduced in NMT in a scientific way. Say, we a have a source sequence $\mathbf{x}$ of length n and try to output a target sequence $\mathbf{y}$ of length m (variables in bold indicate that they are vectors):

$$\mathbf{x} = [x_1, x_2, ..., x_n]$$

$$\mathbf{y} = [y_1, y_2, ..., y_m]$$

The encoder is a bidirectional RNN with a forward hidden state $\overrightarrow{h_i}$ and a backward one $\overleftarrow{h_i}$. A simple concatenation of two represents the encoder state. The motivation is to include both the preceding and followig words in the annotation of one word.

$$\mathbf{h}_i = [\overrightarrow{\mathbf{h}_i^T}; \overleftarrow{\mathbf{h}_i^T}]^T, i = 1, ..., n$$

The decoder network has hidden state $\mathbf{s}_t = f(\mathbf{s}_{t-1}, y_{y-1}, \mathbf{c}_t)$ for the output word at position $t = 1, ..., m$, where the context vector c_t is a sum of hidden states of the input sequence, weighted by alignment scores.

$$\mathbf{c}_t = \sum_{i=1}^{n} \alpha_{t,i} \mathbf{h}_i \text{ ; it is the context vector for output } y_t$$

$$\alpha_{t,i} = \text{align}(y_t, x_i) \text{ ; How well two words } y_t \text{ and } x_i \text{ are aligned.}$$

The alignment model assign a score $\alpha_{t,i}$ to the pair of input at position i and output at position t, (y_t, x_i), based on how well they match. The set of $\{\alpha_{t,i}\}$ are weights defining how much of each source hidden state should be considered for each output. The alignment score α is parametrized by a feed-forward netowrk with a signle hidden layer and this netwrok is jointly trained with other parts of the model. The score function is therefore in the following form, given that *tanh* is used as the non-linear activation function:

$$\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \mathbf{v}_a^T tanh(\mathbf{W}_a[\mathbf{s}_t; \mathbf{h}_i])$$

where both $\mathbf{v}_a$ and $\mathbf{w}_a$ are weight matrices to be learned in the alignement model. The matrix of alignment scores is a nice byproduct to expilictly show the correlation between source and target words.

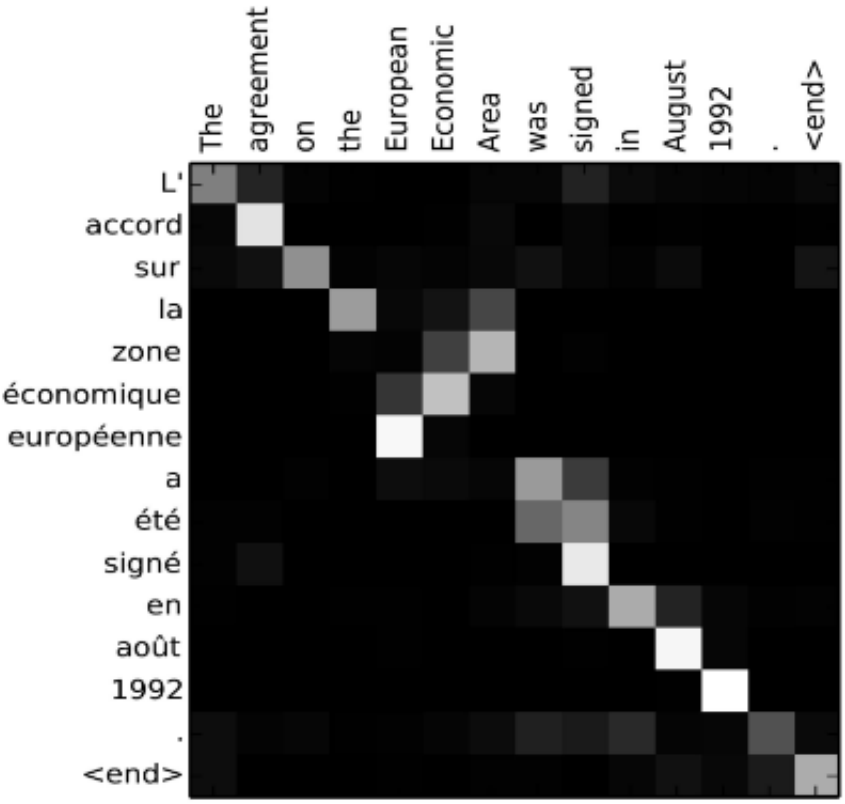


Figure 11.5: Alignment matrix of "L'accord sur l'Espace économique européen a été signé en août 1992" (French) and its English translation "The agreement on the European Economic Area was signed in August 1992".

11.3.2 Self-Attention

Self-attention is an attention mechanism relating different positions of a single sequence in order to compute a representation of the same sequence. It has been shown to be very useful in machine reading or image description generation. The LSTM network paper used self-attention to do machine reading. In the example below, the self-attention mechanism enables us to learn the correlation between the current words and the previous part of the sentence.

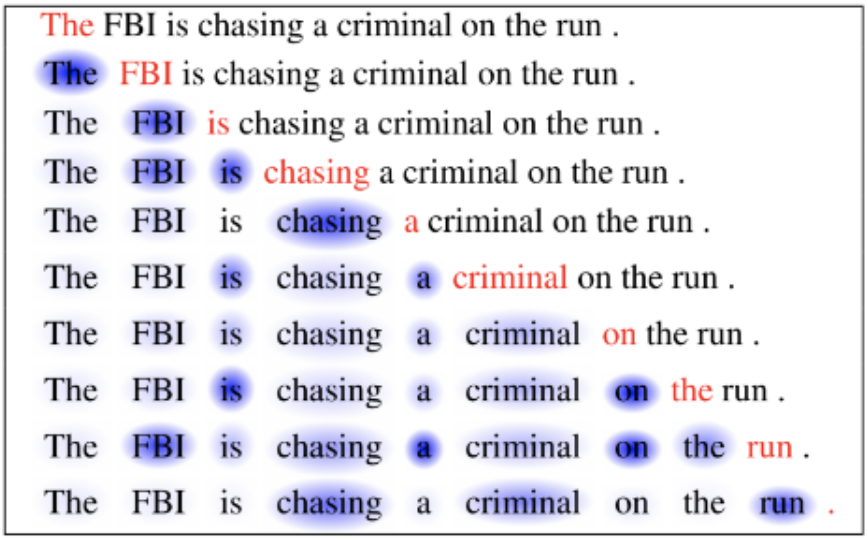


Figure 11.6: The current word is in red and the size of the blue shade indicates the activation level.

More in general we can say that self-attention is a type of attention mechanism where the model makes prediction for one part of a data sample using other parts of the observation about the same sample. Also note that self-attention is permutation-invariant; in other words, it is an operation on sets.

There are various forms of attention or self-attention, Transformer relies on the *scaled dot-product attention*. Given a query matrix Q, a key matrix K and a value matrix V, the output is a wighted sum of the value vectors, where the weight assigned to each value slot is determined by the dot product of the query wwith the corresponding key:

$$\text{Attention}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \text{softmax}(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}})\mathbf{V}$$

11.4 Multi-Head Self-Attention

The multi-head self-attention module is a key component in Transformer. Rather than only computing the attention once, the multi-head mechanism splits the inptus into smaller cunks and then computes the scaled dot-product attention over each subspace in parallel. The independent attetion outputs are simply concatenated and linearly transformed into expected dimensions.

$$\text{MultiHeadAttention}(\mathbf{X}_q, \mathbf{X}_k, \mathbf{X}_v) = [\text{head}_1; ...; \text{head}_h]W^o$$

$$\text{where head}_i = \text{Attention}(\mathbf{X}_q\mathbf{W}_i^q, \mathbf{X}_k\mathbf{W}_i^k, \mathbf{X}_v\mathbf{W}_i^v)$$

where $[.;.]$ is a concatenation operation. $\mathbf{W}_i^q, \mathbf{W}_i^k$ are weight matrices to map input embeddings of size $L \times d$ into query, key and value matrices. And W^o is the output linear transformation. All the weights should be learned during training.

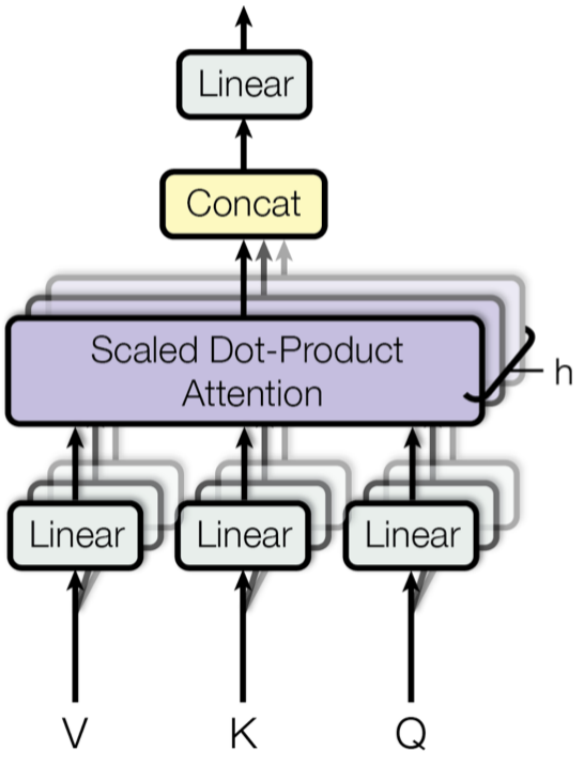


Figure 11.7: Illustration of the multi-head scaled dot-product attention mechanism

11.5 Encoder-Decoder Architecture

The encoder generates an attention-based representation with capability to locate a specific piece of information from a large context. It consists of a stack of 6 identity modules, each containing two submodules, a multi-head self-attention layer and a point-wise fully connected feed-forward network. By point-wise, it means that it applies the same linear transformation (with same weights) to each element in the sequence. This can also be viewed as a convolutional layer with filter size 1. Each submodule has a residual connection and layer normalization. All the submodules output data of the same dimension. The function of Transformer decoder is to retrieve information from the encoded representation. The architecture is quite similar to the encoder, except that the decoder contains two multi-head attention submodules instead of one in each identical repeating module. The first multi-head attention submodule is masked to prevent positions from attending to the future.

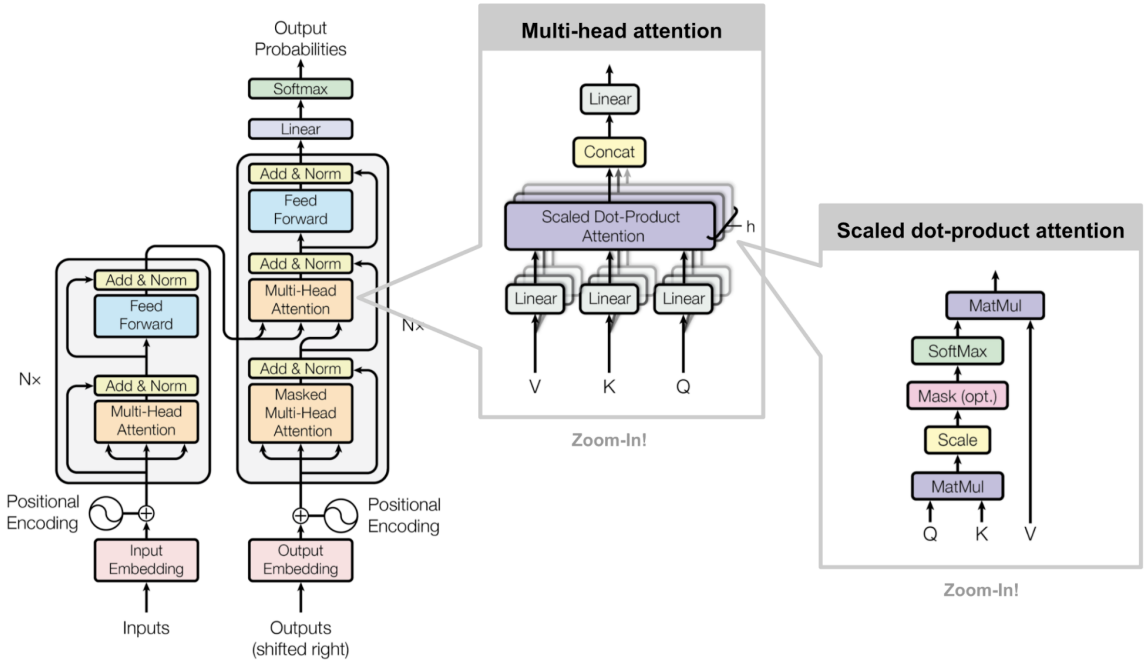


Figure 11.8: The architecture of the vanilla Transformer model

11.5.1 Positional Encoding

Because self-attention operation is permutation invariant, it is important to use proper positional encoding to provide order information to the model. The positional encoding P has the same dimension as the input embedding, so it can be added on the input directly. The vanilla Transformer considered two types of encodings: Sinusoidal positional

encoding is defined as follows, given the token position $i = 1, ..., L$ and the dimension $\delta = 1, ..., d$

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

11.6 Vision Transformer (ViT)

We had seen how the emergence of the Transformer architecture has revolutionized the use of attention, without relying on recurrence and convolutions as earlier attention models had previuously done. In their work, Waswani et al., had applied their model to the specific problem of NLP. Inspired by its success in NLP, in 2021 researchers tried to apply the standard Transformer architecture to images. Their target application at the time was image classification.

Recall to previous sections, the standard Transformer model received a one-dimensional sequence of word embeddings as input, since it was originally meant for NLP. In constrast, when applied to the task of image classification in computer vision, the input data to the Transformer model is provided in the for of two-dimensional images. For the purpose of structuring the input image data in a manner that resembles how the input is structured in the NLP domain (in the sense of having a sequence of individual words), the input image of heigh H, width W and C number of channles is cut up into smaller two-dimensional patches. This result into $N = \frac{HW}{p^2}$ number of patches, where each patch has a resolution of (P, P) pixels. Before feeding the data into the Transfoerm, the following operations are applied:

- Each image patch is flattened into a vector x_p^n of length $P^2 \times C$, where $n = 1, ..., N$.
- A sequence of embedded images patches is generated by mapping the flattened patches to D dimensions, with a trainable linear projection E.
- A learnable class embedding, x_{class}, is prepended to the sequence of embedded image patches. The value of x_{class} represents the classification output, y.
- The patch embeddings are finally augmented with one-dimensional positional embeddings, E_{pos}, hence introducing positional information into the input, which is also learned during training.

The sequence of embedding vectors that results from the previous operations is the following:

$$z_0 = [x_{\text{class}} ; x_p^1 E; ...; x_p^N E] + E_{\text{pos}}$$

In order to perform classification, they feed z_0 at the input of the Transformer encoder (which is the same proposed in *"Attention is all you need"* paper), which consists of a stack of L identical layers. Then, they proceed to take the value of x_{class} at the L^{th} layer of the encoder output, and feed it into a classification head. The classification head is implemented by a MLP with one hidden lyaer at pre-training time and by a single linear layer at fine-tuninig time. The multilayer perceptron (MLP) that forms the classification head implements Gaussian Error Linear Unit (GELU) non-linearity.

In summary, therefore, the ViT employs the encoder part of the original Transformer architecture. The input to the encoder is a sequence of embedded image patches (including a learnable class embedding prepended to the sequence), which is also augmented with positional information. A classification head attached to the output of the encoder receives the value of the learnable class embedding, to generate a classification output based on its state. All of this is illustrated by the figure below:

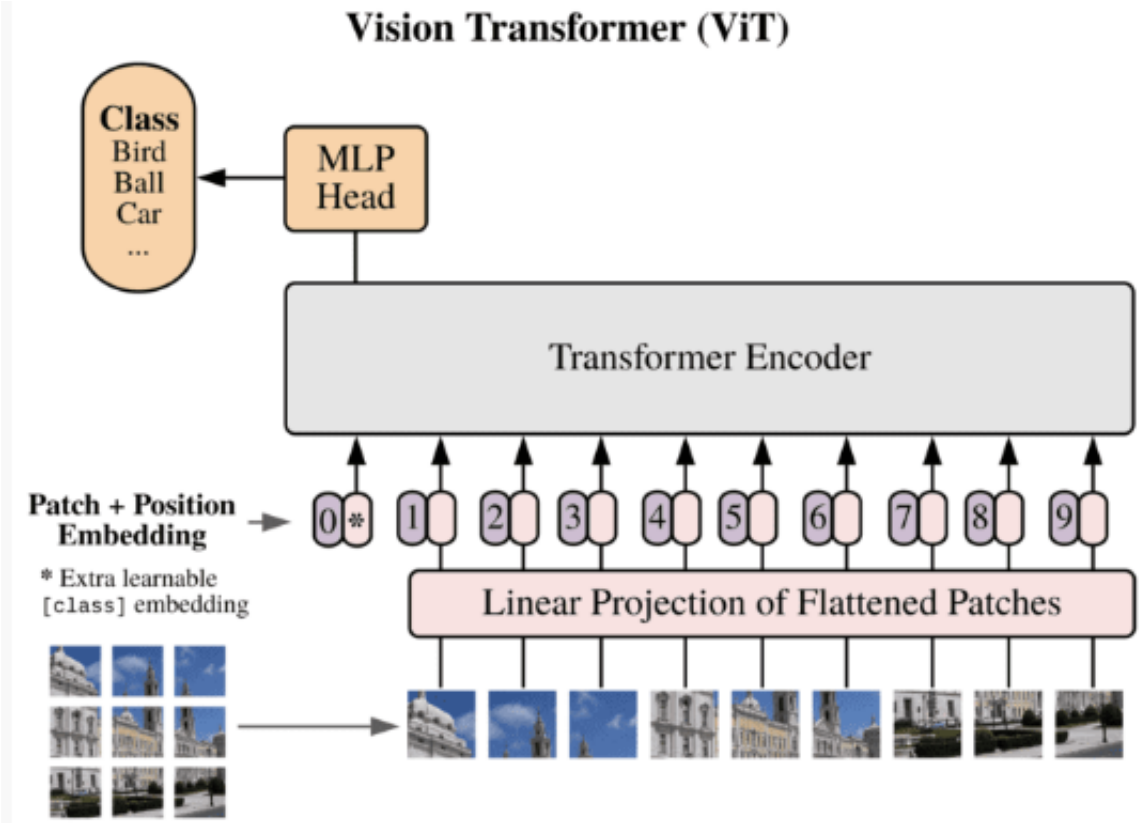


Figure 11.9: The Architecture of the Vision Transformer (ViT)

One further note that Dosovitskiy et al. make, is that the original image can, alternatively, be fed into a CNN before being passed on to the Transformer encoder. The sequence of image patches would the be obtained from the feature maps of the CNN,

while the ensuing process of embedding the feature map patches, prepending a class token, and augmenting with positional information remains the same. Transformers lack some of the inductive biases inherent to CNNs, such as translation equivariance and locality, and therefore do not generalize well when trained on insufficient amounts of data. The convolution operation in combination with max pooling/striding makes CNNs approximately invariant to translation. When a module is translation invariant it means that if we apply translation transformation on the input image, i.e. change the pposition of the objects in the input, the output of the module won't change.