Chapter 11

CONVOLUTIONAL NEURAL NETWORK (CNN) FOR IMAGE CLASSIFICATION

CONTENTS

11.1 Background and History

The history of convolutional neural network (CNN) dated back in 1960's when Hubel and Wiesel (1962) published their discoveries of hierarchical organization in the visual systems and visual processing of cats and primates, and twenty years later in 1981, they shared the Nobel Prize for Physiology or Medicine with

11.2 Convolutional Layers

For images and photos taken by digital cameras, they are composed of a large number of pixels, while pixels close enough often represent a very similar type of information, for instance, parts of sky, ocean, leaves, grasses, animals, fur, brick walls; while the exception only arises at the edges and boundaries of few interfaces where two or more different objects "touch" each other. In other words, much crucial information in an image is commonly concentrated in some local regions, which can be extracted by splitting the image into square patches via a moving window approach. The whole philosophy of CNN is to train several multiple (spatial) regression models, smaller in scale, at once so that each of which uses pixel informations, from each square patch, as independent explanatory variables. Most of these regressors are relatively small in size of input variables, *i.e.* the dimension of the moving window frame, and their purpose is to detect different kinds of pattern from the image, for instance, one regressor learns to detect the light blue sky (or ocean), another one may detect the green grass (or leaves), some other may detect the fringes of a tower or a building, and so forth.

11.2.1 Cross-correlation and Convolution

We first introduce some basic arithmetic operations in the training of CNN. A typical CNN model consists of 3 building blocks: (i) convolutional layers; (ii) pooling layers; and (iii) fully-connected layers. Particularly, for the first layers, one has to use some matrix arithmetics: they are *Cross-correlation* and *Convolution*, their vital role is to build another set of matrices, which summarize the extracted information from the naked primitive image $\boldsymbol{\mathcal{I}}$ by calculating some weighted sums in each of the moving windows. Mathematically, we adopt the following notations:

1. The image $\boldsymbol{\mathcal{I}}$, for simplicity, assumed to be a square of size $s_{\boldsymbol{\mathcal{I}}} \times s_{\boldsymbol{\mathcal{I}}}$. Often, even if it is rectangular, the raw image of a size of $s_{\boldsymbol{\mathcal{I}}1} \times s_{\boldsymbol{\mathcal{I}}2}$ would be rescaled into a square image so as to facilitate processing for the sake of ease;
2. by default, i, j denote the indices of the row and column respectively for different matrices to be encountered;
3. $\boldsymbol{F}_r$ be the r^{th}, for $r = 1, \cdots, n_f$, small square matrix with an odd-valued dimension of size $s_f = 2k+1$, for some $k \in \mathbb{Z}^+$. Each of these n_f matrics is called filter, which helps extract any relevant hidden pattern from the image $\boldsymbol{\mathcal{I}}$, and all of them have the same dimension s_f.
4. Each filter $\boldsymbol{F}_r$ will compose with $\boldsymbol{\mathcal{I}}$ to give an output matrix $\boldsymbol{G}_r$ with size of $s_O \times s_O$, for $r = 1, \cdots, n_f$, further superscript will be added to $\boldsymbol{G}_r$, depending on which arithmetic operation is in use;
5. s_{sr} be the step size for the filter $\boldsymbol{F}_r$ moving across the image $\boldsymbol{\mathcal{I}}$ horizontally and vertically, which is also known as the stride for the filter, see Subsection 11.2.2.

We here provide the definitions of two useful matrix arithmetic operations: denote $\mathbf{1}_{s_f} \in \mathbb{R}^{s_f}$ be the unique vector with all entries equal 1. Also denote, for any matrix $\boldsymbol{A} \in \mathbb{R}^{s_{\boldsymbol{\mathcal{I}}} \times s_{\boldsymbol{\mathcal{I}}}}$, the submatrix $\boldsymbol{A}_{ij}(k) = [a_{i+u,j+v}]_{u,v=-k}^{k}$.

Definition 11.2.1. (**Cross-correlation**) The cross-correlation of $\boldsymbol{\mathcal{I}}$ and $\boldsymbol{F}_r$, which is regarded as the output and denoted by $\boldsymbol{G}_r^{(\star)} = \boldsymbol{\mathcal{I}} \star \boldsymbol{F}_r$, where $\star$ denotes the cross-correlation operator, is defined as:

$$\begin{aligned}\boldsymbol{G}_r^{(\star)}[i,j] &= \sum_{u=-k}^{k}\sum_{v=-k}^{k} \boldsymbol{\mathcal{I}}[(k+1)+(i-1)s_{sr}+u,(k+1)+(j-1)s_{sr}+v]\boldsymbol{F}_r[u,v] \\ &= \mathbf{1}_{s_f}^{\top}(\boldsymbol{\mathcal{I}}_{(k+1)+(i-1)s_{sr},(k+1)+(j-1)s_{sr}}(k) \odot \boldsymbol{F}_r)\mathbf{1}_{s_f}\,, \end{aligned} \tag{11.2.1}$$

for $i,j = 1,\cdots,s_O$.

In both cross-correlation and convolution, a filter $\boldsymbol{F}_r$ moves across the image $\boldsymbol{\mathcal{I}}$, just like a moving window, both horizontally and vertically. In comparing cross-correlation with convolution, the same filter $\boldsymbol{F}_r$ is actually rotated with $\pi = 180°$ and that it is moved across the whole image. Mathematically, for a filter $\boldsymbol{F}_r$, $\boldsymbol{F}_{r,\text{rot}\pi}$ is a matrix such that $\boldsymbol{F}_{r,\text{rot}\pi}[i,j] = \boldsymbol{F}_r[k+1-(i-k-1),k+1-(j-k-1)]$, that is just a rotation through a $\pi = 180°$ around the geometric center of $\boldsymbol{F}_r$.

Definition 11.2.2. (**Convolution**) The convolution of $\boldsymbol{\mathcal{I}}$ and $\boldsymbol{F}_r$ is another matrix, denoted as $\boldsymbol{G}_r^{(*)} = \boldsymbol{\mathcal{I}} * \boldsymbol{F}_r$, where $*$ denotes the convolution operator, is defined as:

$$\begin{aligned}\boldsymbol{G}_r^{(*)}[i,j] &= \sum_{u=-k}^{k}\sum_{v=-k}^{k} \boldsymbol{\mathcal{I}}[(k+1)+(i-1)s_{sr}-u,(k+1)+(j-1)s_{sr}-v]\boldsymbol{F}_r[u,v] \\ &= \mathbf{1}_{s_f}^{\top}(\boldsymbol{\mathcal{I}}_{(k+1)+(i-1)s_{sr},(k+1)+(j-1)s_{sr}}(k) \odot \boldsymbol{F}_{r,\text{rot}\pi})\mathbf{1}_{s_f}\,, \end{aligned} \tag{11.2.2}$$

for $i,j = 1,\cdots,s_O$.

Then, we have the identity

$$\boldsymbol{\mathcal{I}} * \boldsymbol{F}_r = \boldsymbol{\mathcal{I}} \star \boldsymbol{F}_{r,\text{rot}\pi}\,; \tag{11.2.3}$$

similarly, we also have $\boldsymbol{\mathcal{I}} \star \boldsymbol{F}_r = \boldsymbol{\mathcal{I}} * \boldsymbol{F}_{r,\text{rot}\pi}$. More explicitly, we write

$$\begin{aligned}&\sum_{u=-k}^{k}\sum_{v=-k}^{k} \boldsymbol{\mathcal{I}}[(k+1)+(i-1)s_{sr}-u,(k+1)+(j-1)s_{sr}-v]\boldsymbol{F}_r[u,v] \\ =\ &\sum_{u=-k}^{k}\sum_{v=-k}^{k} \boldsymbol{\mathcal{I}}[(k+1)+(i-1)s_{sr}+u,(k+1)+(j-1)s_{sr}+v]\boldsymbol{F}_{r,\text{rot}\pi}[u,v]\,, \end{aligned} \tag{11.2.4}$$

for any $i,j = 1,\cdots,s_O$.

Let us now show a simple illustration for these two convolution concepts, with an input $\boldsymbol{\mathcal{I}}$ of dimension 3×3 and a filter $\boldsymbol{F}_r$ of dimension 3×3, for the magnitudes of the numbers specified in this $\boldsymbol{\mathcal{I}}$, we shall motivate further after this example,

$$\boldsymbol{\mathcal{I}} = \begin{pmatrix} 94 & 50 & 221 \\ 98 & 109 & 50 \\ 49 & 213 & 91 \end{pmatrix} \qquad \text{and} \qquad \boldsymbol{F}_r = \begin{pmatrix} 2 & 2 & 1 \\ 1 & 0 & -2 \\ -2 & 2 & 2 \end{pmatrix}.$$

The input dimension is the same as the filter dimension, the output dimension is $s_O \times s_O = 1\times 1$. By

Definition 11.2.1, the cross-correlation of $\boldsymbol{G}_r^{(\star)} = \boldsymbol{\mathcal{I}} \star \boldsymbol{F}_r$ is just $\mathbf{1}_3^\top(\boldsymbol{\mathcal{I}} \odot \boldsymbol{F}_r)\mathbf{1}_3$, that is:

$$\boldsymbol{G}_r^{(\star)} = \boldsymbol{\mathcal{I}} \star \boldsymbol{F}_r = \begin{pmatrix} 94 & 50 & 221 \\ 98 & 109 & 50 \\ 49 & 213 & 91 \end{pmatrix} \star \begin{pmatrix} 2 & 2 & 1 \\ 1 & 0 & -2 \\ -2 & 2 & 2 \end{pmatrix} = \mathbf{1}_3^\top \begin{pmatrix} 188 & 100 & 221 \\ 98 & 0 & -100 \\ -98 & 426 & 182 \end{pmatrix} \mathbf{1}_3 = 1017\,,$$

where $\mathbf{1}_3 = (1,1,1) \in \mathbb{R}^3$. While the convolution can be obtained similarly but by rotating the filter by an angle of $\pi = 180°$, and now

$$\boldsymbol{F}_{r,\text{rot}\pi} = \begin{pmatrix} 2 & 2 & -2 \\ -2 & 0 & 1 \\ 1 & 2 & 2 \end{pmatrix},$$

with which the convolution is equal to

$$\boldsymbol{G}_r^{(*)} = \boldsymbol{\mathcal{I}} * \boldsymbol{F}_r = \begin{pmatrix} 94 & 50 & 221 \\ 98 & 109 & 50 \\ 49 & 213 & 91 \end{pmatrix} \star \begin{pmatrix} 2 & 2 & -2 \\ -2 & 0 & 1 \\ 1 & 2 & 2 \end{pmatrix} = \mathbf{1}_3^\top \begin{pmatrix} 188 & 100 & -442 \\ -196 & 0 & 50 \\ 49 & 426 & 182 \end{pmatrix} \mathbf{1}_3 = 357\,.$$

Now, we come back to the point about how to use these operations for pattern detection, a small multiple regressor has to learn a $s_f \times s_f$ matrix $\boldsymbol{F}$ of spatial coefficients, for a relatively small $s_f \in \mathbb{N}$. To lay down the general structure of these multiple regressors commonly used for CNN, we first sketch out and introduce the common simple image format in grayscale, ranging from 0 to 255, where a pixel with a value of 0 stands for being colored black, a pixel with a value of 255 represents a white pixel, and a pixel with value in between 0 and 255 represents the decreasing intensity from black to grey and then further to white. For simplicity, we rescale the grayscale images into a unit range, where a pixel with a value of 0 represents being colored black; and a value of 1 stands for a white pixel. Just for illustration, those pixel values are further rounded to the nearest 1 decimal place. Suppose now $s_f = 3$, and consider the following filter $\boldsymbol{F}$ that can detect a small cross-shape in the image:

$$\boldsymbol{F} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix}. \tag{11.2.5}$$

Again, for simplicity, consider $\boldsymbol{\mathcal{I}} \in \mathbb{R}^{3\times 3}$, then the higher the value of the cross-correlation of matrices $\boldsymbol{\mathcal{I}}$ and $\boldsymbol{F}$, the more similarity $\boldsymbol{F}$ is with $\boldsymbol{\mathcal{I}}$. To see this, consider the following two matrices:

$$\boldsymbol{\mathcal{I}}_1 = \begin{pmatrix} 0.7 & 0.2 & 0.3 \\ 0.2 & 0.4 & 0.1 \\ 0.5 & 0.3 & 0.8 \end{pmatrix} \quad \text{and} \quad \boldsymbol{\mathcal{I}}_2 = \begin{pmatrix} 0.1 & 0.9 & 0.2 \\ 0.7 & 0.9 & 0.8 \\ 0.2 & 0.8 & 0.1 \end{pmatrix}.$$

Note that $\boldsymbol{F} = \boldsymbol{F}_{\text{rot}\pi}$, the cross-correlation and convolution of $\boldsymbol{\mathcal{I}}_h$ and $\boldsymbol{F}$ would agree with $h = 1, 2$:

$$\boldsymbol{\mathcal{I}}_1 * \boldsymbol{F} = \boldsymbol{\mathcal{I}}_1 \star \boldsymbol{F} = \begin{pmatrix} 0.7 & 0.2 & 0.3 \\ 0.2 & 0.4 & 0.1 \\ 0.5 & 0.3 & 0.8 \end{pmatrix} \star \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix} = \mathbf{1}_3^\top \begin{pmatrix} 0.7\times 0 & 0.2\times 1 & 0.3\times 0 \\ 0.2\times 1 & 0.4\times 1 & 0.1\times 1 \\ 0.5\times 0 & 0.3\times 1 & 0.8\times 0 \end{pmatrix} \mathbf{1}_3 = 1.2\,;$$

and

$$\boldsymbol{\mathcal{I}}_2 * \boldsymbol{F} = \boldsymbol{\mathcal{I}}_2 \star \boldsymbol{F} = \begin{pmatrix} 0.1 & 0.9 & 0.2 \\ 0.7 & 0.9 & 0.8 \\ 0.2 & 0.8 & 0.1 \end{pmatrix} \star \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix} = \mathbf{1}_3^\top \begin{pmatrix} 0.1\times 0 & 0.9\times 1 & 0.2\times 0 \\ 0.7\times 1 & 0.9\times 1 & 0.8\times 1 \\ 0.2\times 0 & 0.8\times 1 & 0.1\times 0 \end{pmatrix} \mathbf{1}_3 = 4.1\,,$$

from which we see that $\boldsymbol{\mathcal{I}}_2 * \boldsymbol{F} > \boldsymbol{\mathcal{I}}_1 * \boldsymbol{F}$ as expected, even we can actually see a "white" cross appearing in

$\mathcal{I}_2$ before the calculations.

In Chapter 10, each hidden layer in an MLP model has a weight matrix $\boldsymbol{W}^{(\ell)}$ and a bias vector $\boldsymbol{b}^{(\ell)}$, while here for CNN, each filter $\boldsymbol{F}_r^{(\ell)}$ can be seen as a neuron and its matrix entries are regarded as the weight vector $\boldsymbol{\omega}_r^{(\ell)}$ of the neuron r, and there is also a bias term $b_r^{(\ell)}$ associated with the filter $\boldsymbol{F}_r^{(\ell)}$. Very often, ReLU is used as the activation function in CNN. As an illustration, Figure 11.2.1 shows the first 6 steps of cross-correlating the cross-shape filter $\boldsymbol{F}_1^{(\ell)}$ in (11.2.5) together with a bias of -1 across the following 5×5 pixel matrix of an image.

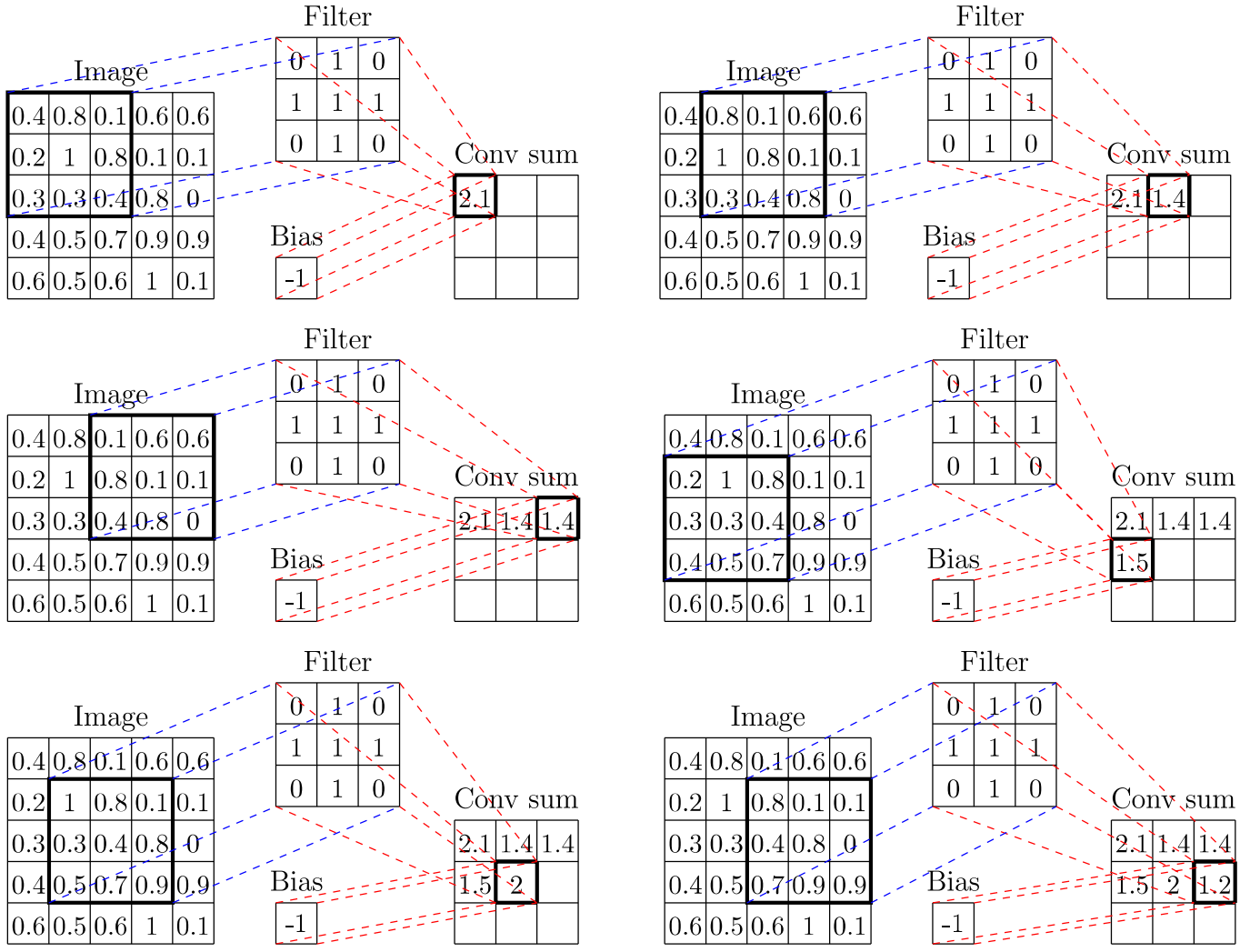
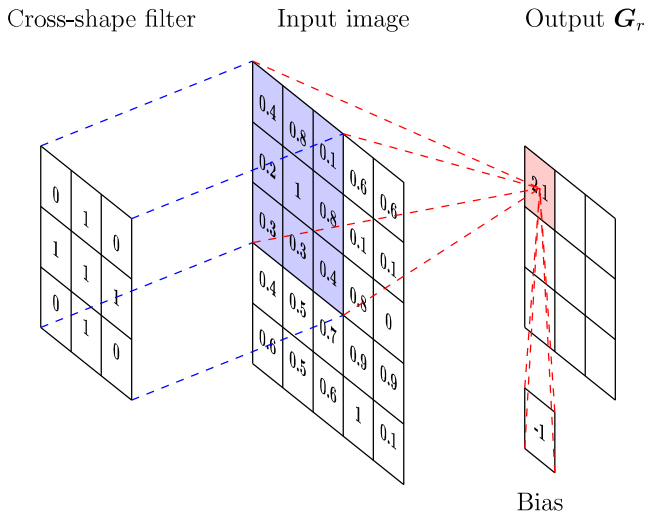


Figure 11.2.1: The cross-shape filter cross-correlating across an image.



In reality, most digital images are composed with pixels, each of which is usually represented by three channels: R (Red), G (Green), and B (Blue), so that each channel is a monochrome picture. The collection of these three R, G, and B monochrome images is called a *volume*, and the size of that collection is called the *volume's depth* $n_v = 3$; of course, n_v can be greater than 3, let say we can incorporate brightness and contrast measures. Figure 11.2.2 shows the first step of cross-correlating of an RGB image $\boldsymbol{\mathcal{I}} = \boldsymbol{\mathcal{I}}_1 \otimes \boldsymbol{\mathcal{I}}_2 \otimes \boldsymbol{\mathcal{I}}_3$ with a volumn depth of $n_v = 3$ by using a filter $\boldsymbol{F}_1 = (\boldsymbol{F}_{1,1} \otimes \boldsymbol{F}_{1,2} \otimes \boldsymbol{F}_{1,3})$ of size $s_f \times s_f \times n_v = 3 \times 3 \times 3$; here $\otimes$ stands for tensor product operation by packing high dimensional objects together to form another even higher dimensional one. More generally, if there are n_f number of filters, the filter tensor is of size $s_f \times s_f \times n_v \times n_f$. For simplicity, each tensor component of the filter tensor used for three different monochrome channels are the same. These different components of filter stack together and become a 3-dimensional tensor with the *filter depth* being the total number of channels n_v. Fortunately, these channels and components of the filter tensor can be cross-correlated individually. For the first channel, we perform the convolution of $\boldsymbol{\mathcal{I}}_1$ with the first component $\boldsymbol{F}_{1,1}$ of the filter tensor; for the second channel, we cross-correlate $\boldsymbol{\mathcal{I}}_2$ with the second component $F_{1,2}$ of F_1; and so forth. After computing all the convolutions, we sum over the volumn depth to obtain our final output image. From Figure 11.2.2, the first grid $\boldsymbol{\mathcal{I}}_{1,22}(3)$, $\boldsymbol{\mathcal{I}}_{2,22}(3)$, and $\boldsymbol{\mathcal{I}}_{3,22}(3)$ are respectively:

$$\boldsymbol{\mathcal{I}}_{1,22}(3) = \begin{pmatrix} 0.1 & 0.1 & 0.2 \\ 0.7 & 0.6 & 0.7 \\ 0.8 & 0.8 & 0.8 \end{pmatrix}, \quad \boldsymbol{\mathcal{I}}_{2,22}(3) = \begin{pmatrix} 0 & 0.3 & 0.9 \\ 1 & 0.3 & 0.7 \\ 0.2 & 0.1 & 0 \end{pmatrix}, \quad \boldsymbol{\mathcal{I}}_{3,22}(3) = \begin{pmatrix} 0.2 & 0.2 & 0.7 \\ 0.3 & 0.5 & 0.4 \\ 0.5 & 0.5 & 0.7 \end{pmatrix}$$

Then, with bias being $b_1 = -3$, the first value of the convolution is:

$$\begin{aligned}
\boldsymbol{G}[1,1] &= (\boldsymbol{\mathcal{I}}_{1,22}(3) \otimes \boldsymbol{\mathcal{I}}_{2,22}(3) \otimes \boldsymbol{\mathcal{I}}_{3,22}(3)) * (\boldsymbol{F}_{1,1} \otimes \boldsymbol{F}_{1,2} \otimes \boldsymbol{F}_{1,3}) + b_1 \\
&= \mathbf{1}_3^\top (\boldsymbol{\mathcal{I}}_{1,22}(3) \odot \boldsymbol{F}_{1,1}) \mathbf{1}_3 + \mathbf{1}_3^\top (\boldsymbol{\mathcal{I}}_{2,22}(3) \odot \boldsymbol{F}_{1,2}) \mathbf{1}_3 + \mathbf{1}_3^\top (\boldsymbol{\mathcal{I}}_{3,22}(3) \odot \boldsymbol{F}_{1,3}) \mathbf{1}_3 + (-3) \\
&= \mathbf{1}_3^\top \begin{pmatrix} 0 & 0.1 & 0 \\ 0.7 & 0.6 & 0.7 \\ 0 & 0.8 & 0 \end{pmatrix} \mathbf{1}_3 + \mathbf{1}_3^\top \begin{pmatrix} 0 & 0.3 & 0 \\ 1 & 0.3 & 0.7 \\ 0 & 0.1 & 0 \end{pmatrix} \mathbf{1}_3 + \mathbf{1}_3^\top \begin{pmatrix} 0 & 0.2 & 0 \\ 0.3 & 0.5 & 0.4 \\ 0 & 0.5 & 0 \end{pmatrix} \mathbf{1}_3 + (-3) \\
&= 2.9 + 2.4 + 1.9 + (-3) = 4.2\,.
\end{aligned}$$

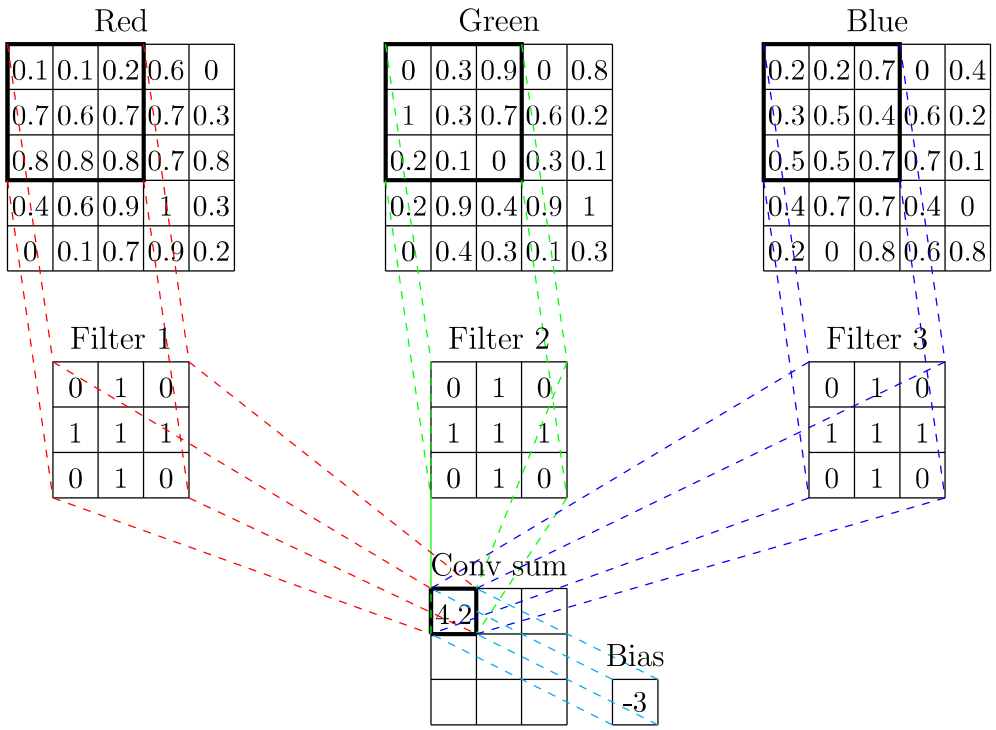


Figure 11.2.2: Convolution with the cross-shape filter of a volume consisting of three monochrome channels.

11.2.2 Stride and Padding

We now discuss further on how to shift the moving windows for conducting different portions of the image. To this point, a *stride* is the step size of the moving window. For instance, we consider:

1. Figure 11.2.1, the stride is 1, and the cross-shape filter slides to the right or down by one cell at a time;
2. Figure 11.2.3, an example of convolution with a stride of 2 is shown.

Clearly, the output matrix $\boldsymbol{G}_r$ is smaller in dimension when stride chosen is bigger involved.

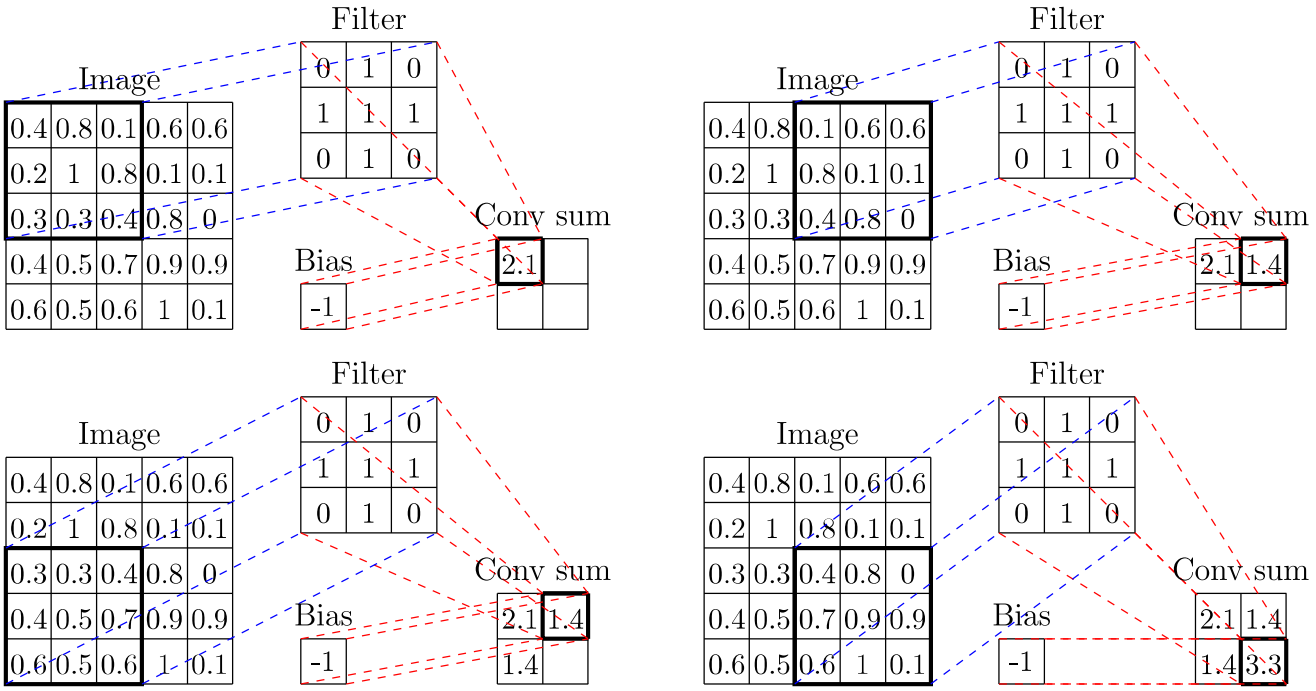


Figure 11.2.3: Convolution with a stride size of 2.

There is another way of augmenting the original image with dummy rows and columns so as to reduce the loss of boundary information, and it is called *padding*, which allows CNN to get a larger output matrix $\boldsymbol{G}_r$. This is the process of adding more dummy cells along the sides of the square image, all before the cross-correlation with a filter is taken. The extra s_p layer of cells added by padding are usually containing zeros only. As discussed above, whether a larger value of stride is taken, the output matrix would become smaller in dimension, here padding is to ensure the dimension of the output matrix to remain similar in size after passing through a few convolutions. Moreover, padding is certainly helpfulfor scanning the boundaries of the image, as the boundary pixels can still convolve with center of a filter. Besides, we also see some examples:

1. In Figure 11.2.1, the padding size is $s_p = 0$, so no additional void cells are added to the image;
2. In Figure 11.2.4, the stride size is $s_{sr} = 2$ and the padding size is $s_p = 1$; cells along the sides of the square image, all before the cross-correlation or convolution with a filter is taken;
3. In Figure 11.2.5, the padding size is now $s_p = 2$.

You can see that the output matrix is bigger when padding is bigger.

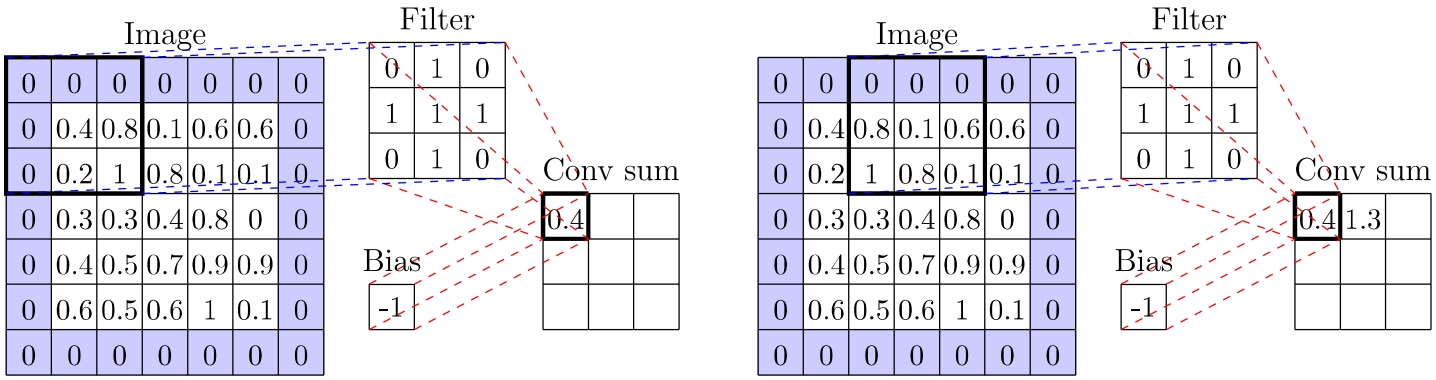
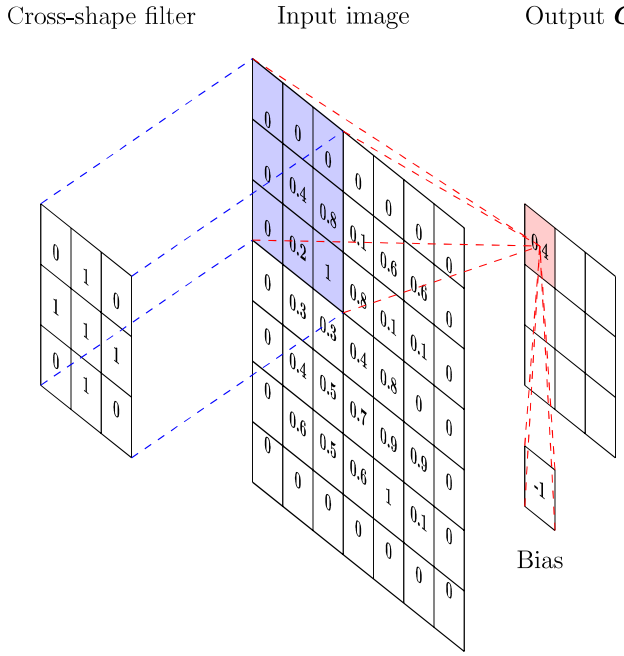


Figure 11.2.4: Convolution with a stride size of 2 and padding with a size of 1.



0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0.4	0.8	0.1	0.6	0.6	0	0
0	0	0.2	1	0.8	0.1	0.1	0	0
0	0	0.3	0.3	0.4	0.8	0	0	0
0	0	0.4	0.5	0.7	0.9	0.9	0	0
0	0	0.6	0.5	0.6	1	0.1	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Figure 11.2.5: Image with a padding of size 2.

As aforementioned, particularly with a stride of 1, applying a filter to the input image matrix still reduces dimension of the output matrix $\boldsymbol{G}_r$, while applying a padding usually helps restore the dimension of the input matrix as if unchanged after a convolution, and this corresponding padding size s_p is given by:

$$s_p = k = \frac{s_f - 1}{2}\,. \tag{11.2.6}$$

This also explain if the filter size s_f is chosen as an odd number, one can conventionally have s_p taken a positive integer. Moreover, if the value is other than 1, ley say the stride size is s_{sr}, this will cause a reduction in the output matrix dimension in comparison with that of input one. The output size s_O is given by:

$$s_{Oj} = \left\lfloor \frac{s_{\boldsymbol{\mathcal{I}}j} + 2s_p - s_f}{s_{sr}} \right\rfloor + 1\,, \qquad \text{for} \quad j = 1, 2\,, \tag{11.2.7}$$

by a simple counting argument, $s_{\boldsymbol{\mathcal{I}}1} \times s_{\boldsymbol{\mathcal{I}}2} \times n_v$ is the input image matrix dimension with n_v channels. In practice, the input image matrix would be made homogeneous along all dimensions by turning it into a square, so as to facilitate an easier implementation of CNN, it means that we aim to set $s_{\boldsymbol{\mathcal{I}}1} = s_{\boldsymbol{\mathcal{I}}2} = s_{\boldsymbol{\mathcal{I}}}$.

As an example, see Figure 11.2.6, we now use (11.2.7), with $s_{\boldsymbol{\mathcal{I}}1} = s_{\boldsymbol{\mathcal{I}}2} = s_{\boldsymbol{\mathcal{I}}}$, to determine the output dimension, and we write the linkage of these dimensions as a simple function mapping:

$$(s_{\boldsymbol{\mathcal{I}}}, s_{\boldsymbol{\mathcal{I}}}, n_v) \circ (s_f, s_f, n_v, n_f) \quad \mapsto \quad (s_O, s_O, n_f)\,. \tag{11.2.8}$$

Now, the input image matrix is of size $224 \times 224 \times 3$, where there are 224×224 pixels and there are 3 different monochrome channels for RGB. Also assume that the stride $s_p = 1$. The filter tensor is of size $3 \times 3 \times 3 \times 64$, and so there are 64 different three filter tensor components for 3 monochrome channels, each of which is of size $s_f = 3$. Finally, the output tensor is $224 \times 224 \times 64$, which is obtained by noting that the padding size and output size are, respectively:

$$s_p = \frac{3-1}{2} = 1 \qquad \text{and} \qquad s_O = \left\lfloor \frac{224 + 2 \cdot 1 - 3}{1} \right\rfloor + 1 = 224\,,$$

hence, (11.2.8) is read with the output dimension being given by:

$$(224, 224, 3) \circ (3, 3, 3, 64) \to (224, 224, 64)\,.$$

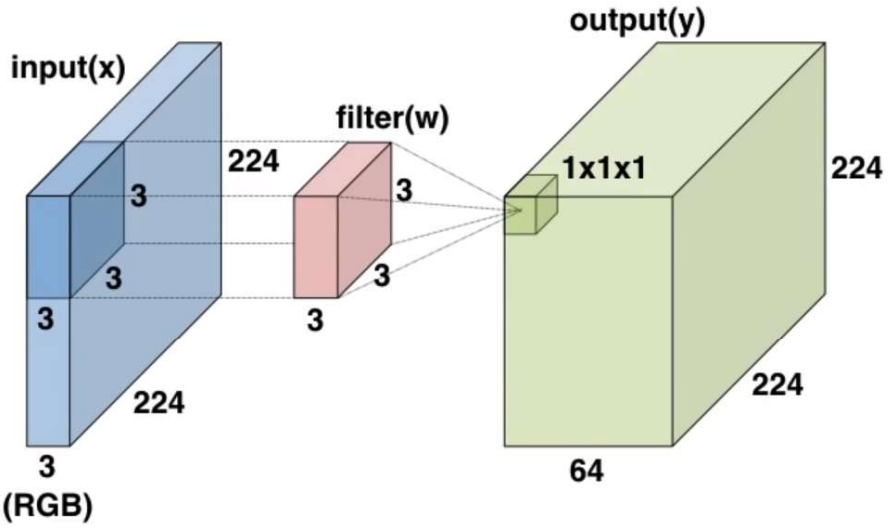


Figure 11.2.6: A convolutional layer with input image of dimension $224 \times 224 \times 3$, filter tensor of dimension $3 \times 3 \times 3 \times 64$, and output tensor is of size $224 \times 224 \times 64$.

11.2.3 Common Examples of Fixed Filter

In the early development of CNN, the concept of convolutional layers had been formulated and these layers were used as an input of another deep neural network for classification analysis, namely an MLP as discussed in Chapter 10. At that time, the fixed filters were dominated such that there was only forward propagation as there were lack of learnable parameters for convolutional layers. Besides, the understanding of the mechanism behind fixed filters can facilitate the interpretation of varying filters. Generally, there are typically two types of fixed filters:

1. **Smoothing**: The sharp fringes become blurry and less contrast that smaller delicate details or the noises are diluted. This by taking a local average of all the neighbouring pixel informations;
2. **Sharpening**: The detail of the edges are enhanced and are contrasted out, which can be done by finding the differences in value between the neighbouring pixels.

For illustration, we only consider 3×3 rotationally symmetric filter $\boldsymbol{F}$, *i.e.* $\boldsymbol{F} = \boldsymbol{F}_{\text{rot}\pi}$ in this subsection, again i and j stands for the row and column indices, respectively, of an image $\boldsymbol{\mathcal{I}}$. Also use the centralized notation as below:

$$\boldsymbol{\mathcal{I}}_{ij}(3) = \begin{pmatrix} \boldsymbol{\mathcal{I}}[i-1,j-1] & \boldsymbol{\mathcal{I}}[i,j-1] & \boldsymbol{\mathcal{I}}[i+1,j-1] \\ \boldsymbol{\mathcal{I}}[i-1,j] & \boldsymbol{\mathcal{I}}[i,j] & \boldsymbol{\mathcal{I}}[i+1,j] \\ \boldsymbol{\mathcal{I}}[i-1,j+1] & \boldsymbol{\mathcal{I}}[i,j+1] & \boldsymbol{\mathcal{I}}[i+1,j+1] \end{pmatrix} \quad \text{and} \quad \boldsymbol{F} = \begin{pmatrix} \boldsymbol{F}[-1,-1] & \boldsymbol{F}[0,-1] & \boldsymbol{F}[1,-1] \\ \boldsymbol{F}[-1,0] & \boldsymbol{F}[0,0] & \boldsymbol{F}[1,0] \\ \boldsymbol{F}[-1,1] & \boldsymbol{F}[0,1] & \boldsymbol{F}[1,1] \end{pmatrix}.$$

With a stride size of $s_{sr} = 1$, the cross-correlation between $\boldsymbol{\mathcal{I}}_{ij}(3)$ and $\boldsymbol{F}$ is given by

$$\boldsymbol{G}^{(\star)}[i,j] = \boldsymbol{\mathcal{I}}_{ij}(3) \star \boldsymbol{F} = \sum_{u=-k}^{k} \sum_{v=-k}^{k} \boldsymbol{\mathcal{I}}_{ij}(3)[i+u, j+v] \cdot \boldsymbol{F}[u,v], \tag{11.2.9}$$

where $i, j = 1, \cdots s_O$, and hence $k = 1$, and $\boldsymbol{\mathcal{I}}[i,j] = 0$ if $i, j \neq 1, \cdots, s_O$, due to the use of padding for the image $\boldsymbol{\mathcal{I}}$; and we just adopt an abuse of notation by identifying the padded $\boldsymbol{\mathcal{I}}$ as $\tilde{\boldsymbol{\mathcal{I}}}$. In the following, we shall discuss the image effect after application of various filters.

We shall use the front cover page of our book, taken at the River Cam of Cambridge, for the illustrations of the effect induced by various filters. The following Programmes 11.2.1 illustrates coding for processing this image by each filter via Python.

```
import numpy as np
import cv2  # OpenCV for image processing
from scipy.signal import convolve

image = cv2.imread("../graphs/originals.jpg", cv2.IMREAD_COLOR) # input image
image = cv2.resize(image, (672, 448)) # Fix the image size as 672x448, here is
    the original size
image = np.pad(image, ((1,1), (1,1), (0, 0)), "constant")

def conv(img, F, name, num=1):  # name is that of the output file
    dim = np.int64(np.sqrt(len(F))) # restricting a return of an integer
    Fil = F.reshape((dim, dim)) # Reshape 1-D vector input into a 2-D square
        matrix filter
    output = convolve(img, np.expand_dims(Fil, axis=-1), mode="valid")

    output = np.clip(output, 0, 255)
    output = np.array(output, np.uint8)
```

```
    cv2.imwrite(f"../graphs/{name}.jpg", output)
```

Programme 11.2.1: Convolving each filters to the input image in Python.

Here importing the `cv2`, also known as OpenCV, library is used for image processing and computer vision. The `convolve()` function in `scipy.signal` performs convolution operator between the input image $\boldsymbol{\mathcal{I}}$ and a Filter $\boldsymbol{F}$. We next import an image using the `cv2.imread()` function, where the path of the image is put as the first argument and the array of color values of the image as the second argument. The argument `cv2.IMREAD_COLOR` indicates that the image is imported in its original color; while another possible argument for this is `cv2.COLOR_BGR2GRAY`, which converts the BGR (Blue-Green-Red) scale images into the gray-scale ones. As a remark, the BGR-scale is the same as the RGB-scale except the orders of the colors are swapped. Next, we pad the image with 0 along its boundary using `np.pad()` function from the `numpy` library. In particular, the first argument is the array of the image to be padded; the second argument is the collection of the number of the width number of the padding cushions along any particular photo edge; the third argument `"constant"` indicates that only the constant 0 is used for all padding. To be more precise, the `((1,1), (1,1), (0,0))` indicates that in the first ordered pair `(1,1)` indicates that only one row of 0 is padded on the top of the image and the second 0 tells us that another row of 0 is padded at the bottom of the image; the second ordered pair `(1,1)` indicates that only one column of 0 is padded on the leftmost while the second 1 implies another column of 0 is padded at the rightmost; the third ordered pair `(0, 0)` indicates that no padding is applied on the top of or at the back of the image; see Figure 11.2.7 for more details.

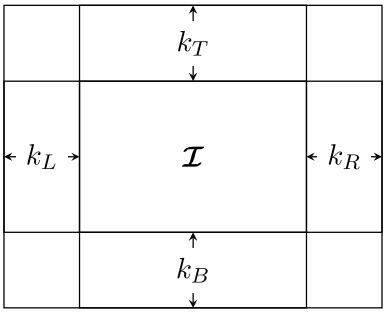


Figure 11.2.7: Using `np.pad()` function with input image $\boldsymbol{\mathcal{I}}$ and second parameter $((k_T, k_B), (k_L, k_R), (0, 0))$.

Next, we use the `convolve()` function from `scipy.signal` to perform convolution, in which the first argument corresponds to the input image, the second argument is the filter, and the third argument `mode="valid"` indicates that the image is directly convolved with the filter without any padding applied to the image; moreover, for the second argument, `np.expand_dims()` function is used to lift the array to become a tensor by adhering one additional dimension, at the right end of the filter, namely channel depth. As we shall discover in the rest of this section, the most of the commonly used fixed filters have negative entries, and so do those of any convolutional products. Note that, the RGB format has the range of possible values only from 0 (absense of color) to 255 (full satuation) for each of the monochrome channels, values below zero or above 255 would not be counted as a legitimate RGB input. One method of dealing with any possible negative values is the usual notion of *clipping* (also see Subsection 10.2.1) through the use of `np.clip()` function, it means that all values smaller than zero are replaced by zero, while those larger

than 255 are replaced by 255. By then, there would be lots of black and white areas in the output photo $\boldsymbol{G}_r$; indeed, in the later part of this section, we shall see that white regions of $\boldsymbol{G}_r$ are corresponding to the edges and boundaries of various objects in the original image $\boldsymbol{\mathcal{I}}$. Finally, we save the output image via the `cv2.imwrite()` function.

Nevertheless, in recent application of CNN, we often consider the convolutional layer as a learning network with indetermined weights $\boldsymbol{K}$ instead of a fixed filter tensor $\boldsymbol{F}_r$, known as *kernel tensor*, being summed with biases $\boldsymbol{b}$ to be tuned. Though this will drag up the computational cost, with the availability of GPUs and TPUs, the training time of the floating parameters $\boldsymbol{K}$ and $\boldsymbol{b}$ can be significantly shorten for a few folds. Therefore, most recent machine learning practitioners tune their overall CNN with learnable kernel tensors and biases instead of fixed filters and biases, *e.g.* the AlexNet; and the overall training procedures will be discussed in Section 11.5. In the rest of this section, we still focus on the interpretation of various fixed filters.

Clearly, a dummy filter $\boldsymbol{F}_{\boldsymbol{\mathcal{I}}}$ just returns the original input image, *i.e.* $\tilde{\boldsymbol{\mathcal{I}}} * \boldsymbol{F}_{\boldsymbol{\mathcal{I}}} = \boldsymbol{\mathcal{I}}$, where

$$\boldsymbol{F}_{\boldsymbol{\mathcal{I}}} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \tag{11.2.10}$$

see the unchanged effect in Table 11.2.1 with the augmented coding of Programme 11.2.2 to the original Programme 11.2.1.

```
18 Filter = np.array([0, 0, 0, 1, 0, 0, 0, 0])
19 conv(image, Filter, "original")
```

Programme 11.2.2: A dummy filter used with function `conv()` defined in Programme 11.2.1.

Description	Filter	Results
Original	$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$	

Table 11.2.1: Original filter.

Let us turn to more detailed discussion on the aforementioned two types of filters, and the following examples will not be treated by the pooling nor the activation function procedures, yet only padding with the padding size of $s_p = 1$ and the stride size of $s_{sr} = 1$.

11.2.3.1 Filters for Smoothing

For the purpose of smoothing, the *weighted average filter* $\boldsymbol{F}_{avg}$ is such that

$$\sum_{u=-k}^{k}\sum_{v=-k}^{k}\boldsymbol{F}_{avg}[u,v]=1\,.$$

Particularly, the averaging effect can straighten out the iid noises over a patch. Suppose that the (i,j)-pixel noise $\varepsilon_{ij}\sim\mathcal{N}(0,\sigma^2)$, and these pixel noises are also independently distributed, for $i,j=1,\cdots,s_O$, and $\boldsymbol{\mathcal{I}}[i,j]=\boldsymbol{X}_{ij}+\varepsilon_{ij}$, here each $\boldsymbol{X}_{ij}$ is considered as a constant. Without the averaging effect, the variance at the pixel (i,j) is σ^2; but for

$$\boldsymbol{G}^{(\star)}_{avg}[i,j]=\sum_{u=-k}^{k}\sum_{v=-k}^{k}\boldsymbol{\mathcal{I}}_{i,j}(3)[i+u,j+v]\boldsymbol{F}_{avg}[u,v]\,,$$

the variance of $\boldsymbol{G}^{(\star)}_{avg}[i,j]$ is

$$\begin{aligned}\mathrm{Var}(\boldsymbol{G}^{(\star)}_{avg}[i,j])&=\mathrm{Var}\left(\sum_{u=-k}^{k}\sum_{v=-k}^{k}(\boldsymbol{X}_{ij}(3)[i+u,j+v]+\varepsilon_{ij})\cdot\boldsymbol{F}_{avg}[u,v]\right)\\&=\sum_{u=-k}^{k}\sum_{v=-k}^{k}(\boldsymbol{F}_{avg}[u,v])^2\cdot\sigma^2<\sigma^2\,,\end{aligned}$$

since

$$\sum_{u=-k}^{k}\sum_{v=-k}^{k}(\boldsymbol{F}_{avg}[u,v])^2<\sum_{u=-k}^{k}\sum_{v=-k}^{k}\boldsymbol{F}_{avg}[u,v]=1\,,$$

hence, the variance of noise at the pixel (i,j) is now reduced after the application of weighted average filter $\boldsymbol{F}_{avg}$. The following mean filter (also called box filter) $\boldsymbol{F}_{\text{Box}}$ and the Gaussian filter $\boldsymbol{F}_{\text{Gauss}}$ are two common types of the weighted average filter:

$$\boldsymbol{F}_{\text{Box}}=\frac{1}{9}\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix}=\frac{1}{3}\begin{pmatrix}1\\1\\1\end{pmatrix}\begin{pmatrix}1&1&1\end{pmatrix}\frac{1}{3}\qquad\text{and}\qquad\boldsymbol{F}_{\text{Gauss}}=\frac{1}{16}\begin{pmatrix}1&2&1\\2&4&2\\1&2&1\end{pmatrix};$$

indeed, the Gaussian filter possesses entries that represent sample outputs from a (degenerated) bivariate Gaussian distribution; recall that (x, y) is said to follow a degenerated bivariate Gaussian distribution if each of them is an independent, one-dimensional Gaussian random variable with the mean 0 and a common variance σ^2, such that the joint p.d.f. $f_{\text{x,y}}(x,y)$ is given by:

$$f_{\text{x,y}}(x,y)=\frac{1}{2\pi\sigma^2}\exp\left(-\frac{x^2+y^2}{2\sigma^2}\right),\qquad-\infty<x<\infty,\,-\infty<y<\infty\,.$$

By very definition, the Gaussian filter $\boldsymbol{F}_{\text{Gauss}}$ is defined by:

$$\boldsymbol{F}_{\text{Gauss}}=\frac{1}{\sum_{u=-1}^{1}\sum_{v=-1}^{1}f(u,v)}\begin{pmatrix}f(-1,-1)&f(0,-1)&f(1,-1)\\f(-1,0)&f(0,0)&f(1,0)\\f(-1,1)&f(0,1)&f(1,1)\end{pmatrix},\tag{11.2.11}$$

where $f(0,0)$ is the central peak of this bivariate Gaussian distribution, the variance σ^2 controls the relative height of this peak. Let us consider two extreme cases: (1) when $\sigma^2\to\infty$, these nine entries become equal in value, *i.e.* $f(i,j)\approx f(0,0)$ for any $i,j=-1,0,1$, and in essence, this Gaussian filter $\boldsymbol{F}_{\text{Gauss}}$ is equivalent to the mean filter $\boldsymbol{F}_{\text{Box}}$; (2) if $\sigma^2\to 0$, we have $f(0,0)=1$ as always, while all other entries $f(i,j)=0$ for $i,j\neq 0$; therefore, the cross-correlation product of the input image $\boldsymbol{\mathcal{I}}$ and the corresponding Gaussian filter $\boldsymbol{F}_{\text{Gauss}}$ returns the original input image $\boldsymbol{\mathcal{I}}$. In general, a larger value of σ^2 leads to a more smoother effect

on the image. Finally, for $\sigma^2 = 1$,

$$\boldsymbol{F}_{\text{Gauss}} = \frac{1}{0.7795}\begin{pmatrix} 0.05855 & 0.09653 & 0.05855 \\ 0.09653 & 0.15916 & 0.09653 \\ 0.05855 & 0.09653 & 0.05855 \end{pmatrix} = \begin{pmatrix} 0.07511 & 0.12384 & 0.07511 \\ 0.12384 & 0.20418 & 0.12384 \\ 0.07511 & 0.12384 & 0.07511 \end{pmatrix},$$

yet for the sake of convenience, we can use simple fractions to approximate these entries, that gives the usual expression:

$$\boldsymbol{F}_{\text{Gauss}} = \frac{1}{16}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} = \frac{1}{4}\cdot\begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}\begin{pmatrix} 1 & 2 & 1 \end{pmatrix}\cdot\frac{1}{4}, \tag{11.2.12}$$

where the rightmost expression demonstrates that the Gaussian filter $\boldsymbol{F}_{\text{Gauss}}$ is a product of one-dimensional Gaussian filter and itself. As an alternative notation, these entries are actually the combinatorial coefficients of the binomial expansion of the polynomial $(1+x)^{s_f-1}$; here with $s_f = 3$, the coefficients are exactly 1, 2, 1. For $s_f = 5$, we obtain the coefficients 1, 4, 6, 4, 1, and so the 5×5

$$\boldsymbol{F}_{\text{Gauss}} = \frac{1}{16}\begin{pmatrix} 1 \\ 4 \\ 6 \\ 4 \\ 1 \end{pmatrix}\begin{pmatrix} 1 & 4 & 6 & 4 & 1 \end{pmatrix}\frac{1}{16}.$$

To make these two interpretation consistent with each other, we first recall the De Moivre's version of Central Limit Theorem, namely, the rescaled combinatorial factor $\binom{n}{r}/2^n$ actually converges to normal density function; besides, being an independent random variable along each dimension, the joint probability is just the product of the marginal densities. Hence, these two facts explain the well-approximation of (11.2.12) or its generalized version.

```
Filter = np.array([1,1,1,1,1,1,1,1,1]) / 9
conv(image, Filter, "box blur")

Filter = np.array([1,2,1,2,4,2,1,2,1]) / 16
conv(image, Filter, "gaussian blur")
```

Programme 11.2.3: Box filter and Gaussian filter used with `conv()` function defined in Programme 11.2.1.

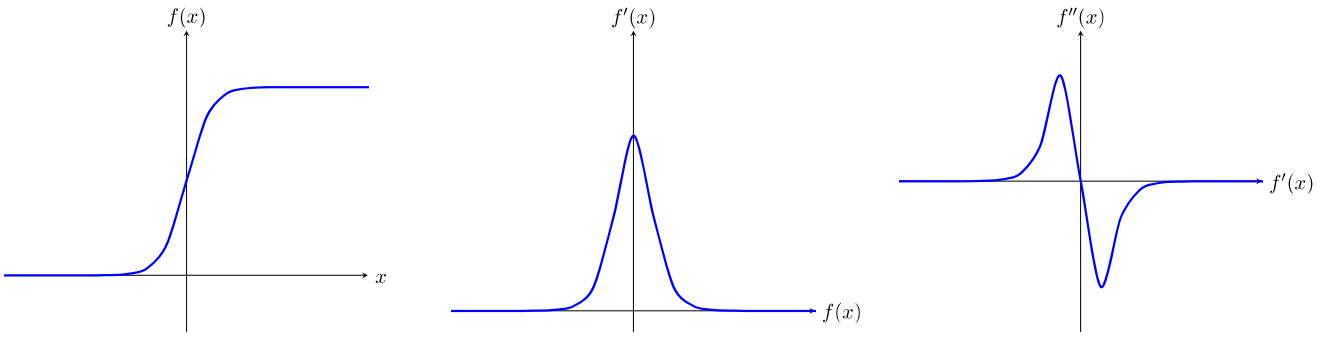
Description	Filter	Results
Box blur	$\frac{1}{9} \cdot \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	
Gaussian blur	$\frac{1}{16} \cdot \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$	

Table 11.2.2: Box filter and Gaussian filter.

Comparing with Table 11.2.1, the output images after applying the mean filter or the Gaussian filter shown in Table 11.2.2 via the Programme 11.2.3 are both blurry; moreover, as expected, box filter returns an even more blurry image than that after the Gaussian filter.

11.2.3.2 Filters for Sharpening

Normally, edges illustrate a significant difference in color values among the neighbouring pixels, and this thematically phenomenon can be "smoothly" modelled by a steep S-shaped sigmoid function $f(x)$ as shown in Figure 11.2.8(a), where the first order derivative $f'(x)$ and the second order derivative $f''(x)$ are also shown in Figure 11.2.8(b) and 11.2.8(c), respectively.



(a) A sigmoid function $f(x)$. (b) First order derivative $f'(x)$. (c) Second order derivative $f''(x)$.

Figure 11.2.8: The first order and second order derivative of a steep S-shaped sigmoid function $f(x)$.

As clear from the last subsubsection, many common filters can be decomposed as a product of two one-dimensional filters, which are respectively responsible for the horizontal and the verical axes. Recall an alternative definition of the first order derivative for a function f:

$$\frac{\partial f}{\partial x}(x) = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}, \qquad x \in \mathbb{R},$$

this motivates the first order finite difference approximation,

$$\frac{\partial f}{\partial x}(x) \approx \frac{1}{2}\Big(f(x+1) - f(x-1)\Big) = \frac{1}{2}\Big((-1)\cdot f(x-1) + 0 \cdot f(x) + 1 \cdot f(x+1)\Big),$$

where we take $h = 1$ pixel width which is considered as very small quantity for the whole image; based on this, we can form an one-dimensional filter $(-1, 0, 1)^\top$ that returns the gradient of color-valued function along one dimension, either horizontally or vertically. Therefore, by multiplying two one-dimensional filters, we further obtain

$$\boldsymbol{F}_{\text{first}} = \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix} \begin{pmatrix} -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{pmatrix}; \tag{11.2.13}$$

indeed, the first order partial derivative $f(x, y)$ with respect to x tells us that:

$$\frac{\partial f}{\partial x}(x, y) = \lim_{h \to 0} \frac{f(x+h, y) - f(x-h, y)}{2h}, \quad (x, y) \in \mathbb{R}^2,$$

which serves horizontally. Besides, the second order partial derivative $f(x, y)$ with respect to x and y is:

$$\frac{\partial^2 f}{\partial x \partial y}(x, y) = \lim_{h \to 0} \lim_{k \to 0} \frac{(f(x+h, y+k) - f(x+h, y-k)) - (f(x-h, y+k) - f(x-h, y-k))}{4hk}, \quad (x, y) \in \mathbb{R}^2,$$

this motivates the second order finite difference approximation with $h = k = 1$,

$$\frac{\partial^2 f}{\partial x \partial y}(x, y) \approx \frac{1}{4}\Big(1 \cdot f(x-1, y-1) + (-1) \cdot f(x+1, y-1) + (-1) \cdot f(x-1, y+1) + 1 \cdot f(x+1, y+1)\Big),$$

which resembles the filter in (11.2.13) that

$$4 \cdot \frac{\partial^2 f}{\partial x \partial y}(x, y) \approx \begin{pmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{pmatrix} * \begin{pmatrix} f(x-1, y-1) & f(x, y-1) & f(x+1, y-1) \\ f(x-1, y) & f(x, y) & f(x+1, y) \\ f(x-1, y+1) & f(x, y+1) & f(x+1, y+1) \end{pmatrix},$$

see Table 11.2.3 through the use of Programme 11.2.4 for an illustration of the effect of $\boldsymbol{F}_{\text{first}}$ on $\boldsymbol{\mathcal{I}}$.

```
Filter = np.array([1,0,-1,0,0,0,-1,0,1])
conv(image, Filter, "first order")
```

Programme 11.2.4: First order filter used with `conv()` function defined in Programme 11.2.1.

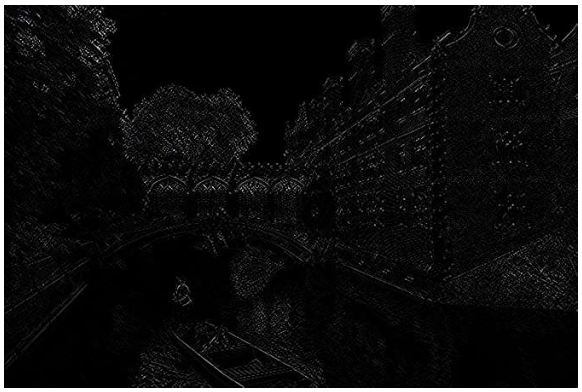
Description	Filter	Results
First order	$\begin{pmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{pmatrix}$	

Table 11.2.3: First order filter $\boldsymbol{F}_{\text{first}}$.

Furthermore, we can also combine this finite differencing one-dimensional filter with the box filter $(1,1,1)^\top$, so as to obtain a horizontal version of Prewitt filter $\boldsymbol{F}_{\text{Prewitt, horizontal}}$ for detecting horizontal edges or a vertical version of Prewitt filter $\boldsymbol{F}_{\text{Prewitt, vertical}}$ for detecting vertical edges:

$$\boldsymbol{F}_{\text{Prewitt, horizontal}} = \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}, \tag{11.2.14}$$

$$\boldsymbol{F}_{\text{Prewitt, vertical}} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \begin{pmatrix} -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix}, \tag{11.2.15}$$

also see Table 11.2.4 with the aid of the Programme 11.2.5 for their effects. Particularly, for a vertical edge of image such as $\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$ after the convolution with these filters, we see that

$$\boldsymbol{F}_{\text{Prewitt, horizontal}} * \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix} \star \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} = 0,$$

$$\boldsymbol{F}_{\text{Prewitt, vertical}} * \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix} \star \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} = 3,$$

that means $\boldsymbol{F}_{\text{Prewitt, horizontal}}$ cannot discover the vertical changes in colour, while $\boldsymbol{F}_{\text{Prewitt, vertical}}$ can do the job; similar examples apply to horizontal edges. In general, the sum of all entries in each of these edge detecting filters should be zero so as to only detect sharp changes in colour, while whenever all nearby pixels having almost the same colour will return a cross-correlation output sum of zero.

```
Filter = np.array([-1, -1, -1, 0, 0, 0, 1, 1, 1])
conv(image, Filter, "Prewitt horizontal")

Filter = np.array([-1, 0, 1, -1, 0, 1, -1, 0, 1])
```

```
34 conv(image, Filter, "Prewitt vertical")
```

Programme 11.2.5: Prewitt filters used with `conv()` function defined in Programme 11.2.1.

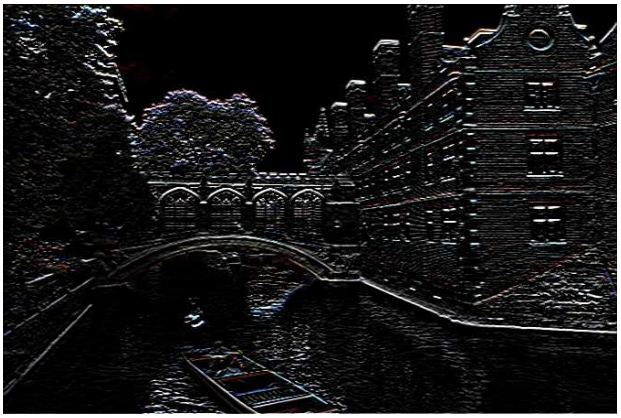
Description	Filter	Results
Prewitt horizontal	$\begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	
Prewitt vertical	$\begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix}$	

Table 11.2.4: Prewitt filters for detecting horizontal and vertical edges, respectively.

Similarly, using a one-dimensional Gaussian filter together with the differentiation kernel $(-1, 0, 1)^\top$, we obtain a horizontal Sobel filter[14] $\boldsymbol{F}_{\text{Sobel, horizontal}}$ for detecting horizontal edges and a vertical Sobel filter $\boldsymbol{F}_{\text{Sobel, vertical}}$ for detecting vertical edges:

$$\boldsymbol{F}_{\text{Sobel, horizontal}} = \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 1 \end{pmatrix} = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}, \tag{11.2.16}$$

$$\boldsymbol{F}_{\text{Sobel, vertical}} = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \begin{pmatrix} -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, \tag{11.2.17}$$

also see Table 11.2.5 through Programme 11.2.6. In comparison with the Prewitt filter, Sobel filter gives more contrast in edges because there is an additional weighting on the center of the one-dimensional filter

[14]Sobel filter was named after Irwin Sobel and Gary Feldman from Stanford Artificial Intelligence Laboratory (SAIL).

$(1, 2, 1)$ vs Prewitt filter $(1, 1, 1)$.

```
Filter = np.array([-1, -2, -1, 0, 0, 0, 1, 2, 1])
conv(image, Filter, "Sobel horizontal")

Filter = np.array([-1, 0, 1, -2, 0, 2, -1, 0, 1])
conv(image, Filter, "Sobel vertical")
```

Programme 11.2.6: Sobel filters used with `conv()` function defined in Programme 11.2.1.

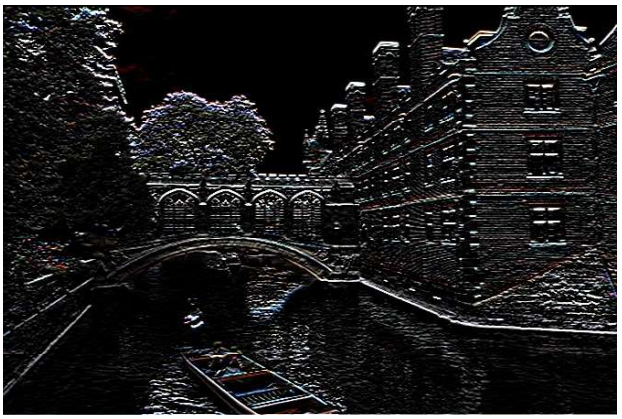
Description	Filter	Results
Sobel horizontal	$\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$	
Sobel vertical	$\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$	

Table 11.2.5: Sobel filters for detecting horizontal and vertical edges, respectively.

So far, we only introduced filters that highlight either horizontal or vertical edges but not both. We next illustrate a family that can capture both, and it is called Laplacian filters. Let us consider first the following Laplacian filter:

$$\boldsymbol{F}_{\text{Laplace},4} = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix}. \tag{11.2.18}$$

To motivate its name after the well-known historical figure Pierre-Simon Laplace (1749 – 1827) , we first consider the second order partial derivative of $f(x, y)$ with respect to x:

$$\frac{\partial^2 f}{\partial x^2}(x, y) = \lim_{h \to 0} \frac{f(x + h, y) - f(x, y) - (f(x, y) - f(x - h, y))}{h^2}, \qquad (x, y) \in \mathbb{R}^2,$$

in which we have the corresponding second order finite approximation with $h = 1$:

$$\frac{\partial^2 f}{\partial x^2}(x,y) \approx f(x+1,y) - 2f(x,y) + f(x-1,y)\,.$$

We next consider the following finite differencing approximation of the Laplacian operator for a regular enough function f, where h and k are both taken 1 unit of pixel of the image $\boldsymbol{\mathcal{I}}$ horizontally and vertically respectively:

$$\begin{aligned}\Delta f(x,y) &:= \frac{\partial^2 f}{\partial x^2}(x,y) + \frac{\partial^2 f}{\partial y^2}(x,y)\,, \qquad (x,y) \in \mathbb{R}^2 \\ &\approx f(x+1,y) - 2f(x,y) + f(x-1,y) + f(x,y+1) - 2f(x,y) + f(x,y-1) \\ &= 1 \cdot f(x+1,y) + 1 \cdot f(x-1,y) + (-4) \cdot f(x,y) + 1 \cdot f(x,y+1) + 1 \cdot f(x,y-1)\,,\end{aligned}$$

and we immediately think of the following Laplacian mask $\boldsymbol{M}_{\text{Laplace,4}}$:

$$\Delta f(x,y) \approx \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} * \begin{pmatrix} f(x-1,y-1) & f(x,y-1) & f(x+1,y-1) \\ f(x-1,y) & f(x,y) & f(x+1,y) \\ f(x-1,y+1) & f(x,y+1) & f(x+1,y+1) \end{pmatrix}. \tag{11.2.19}$$

In particular, $\boldsymbol{F}_{\text{Laplace,4}} = -\boldsymbol{M}_{\text{Laplace,4}}$ is the commonly used Laplacian kernel, where $\boldsymbol{M}_{\text{Laplace,4}}$ takes care of outward edges as the outer four entries are positive and the center number is negative, while $\boldsymbol{F}_{\text{Laplace,4}}$ looks for inward edges. Recall that black color has an RGB value of (0, 0, 0) and white color has an RGB value of (255, 255, 255). Therefore, the (positive) Laplacian Mask $M_{\text{Laplace,4}}$ detects the edges that appear brighter than the surrounding environment, and so appear to be pointing outward in the photo towards as; whereas the (negative) Laplacian Mask $F_{\text{Laplace,4}}$ detects the edges that appear darker than the neighborhood, seems to be a pit in the photo. From Table 11.2.6, we can see that its application is mainly to extract both horizontal and vertical edges but only vaguely if not too sharp. Therefore, if we add the Laplacian filter $\boldsymbol{F}_{\text{Laplace,4}}$ to the original filter in (11.2.10), we can sharpen the original image $\boldsymbol{\mathcal{I}}$. By then, as illustrated in Table 11.2.6 via the Programme 11.2.7, the final filter is:

$$\boldsymbol{F}_{\text{Laplace,5}} = \boldsymbol{F}_{\boldsymbol{\mathcal{I}}} + \boldsymbol{F}_{\text{Laplace,4}} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix},$$

The mechanism behind this whitening effect can be understood from Figure 11.2.9; for instance, along one direction, the colour value difference shown in Figure 11.2.9 is enlarged along an edge, as shown in Figure 11.2.8(a).

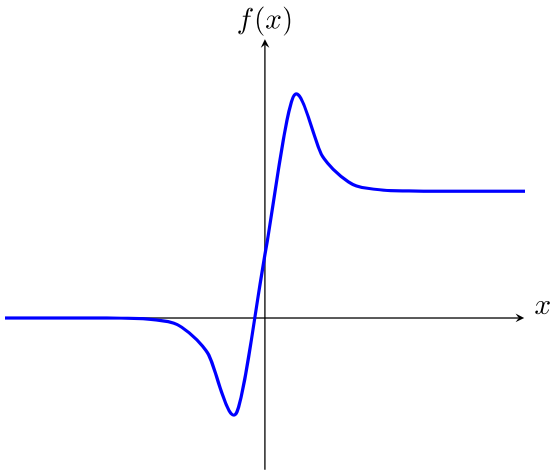


Figure 11.2.9: Sharpened function obtained from $f(x) - f''(x)$.

Moreover, we can extend the cross-shape Laplacian filter with its diagonal version:

$$\boldsymbol{F}_{\text{Laplace,8}} = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix} + \begin{pmatrix} -1 & 0 & -1 \\ 0 & 4 & 0 \\ -1 & 0 & -1 \end{pmatrix} = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}, \tag{11.2.20}$$

where the second filter in the sum is obtained by rotating the axes by $\pi/4$, and so it can detect edges at $\pi/4$ or $3\pi/4$ with the horizontal axis; as a whole $\boldsymbol{F}_{Laplace,8}$ can effectively detect edges along all possible directions in practice. Finally, adding this effective edge detector filter $\boldsymbol{F}_{Laplace,8}$ to the original filter in (11.2.10):

$$\boldsymbol{F}_{\text{Laplace,9}} = \boldsymbol{F}_{\boldsymbol{\mathcal{I}}} + \boldsymbol{F}_{\text{Laplace,8}} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix} = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{pmatrix},$$

we can produce a much whitening effect as shown in the last figure in Table 11.2.6.

```
Filter = np.array([0, -1, 0, -1, 4, -1, 0, -1, 0])
conv(image, Filter, "Laplacian 4")

Filter = np.array([0, -1, 0, -1, 5, -1, 0, -1, 0])
conv(image, Filter, "Laplacian 5")

Filter = np.array([-1, -1, -1, -1, 8, -1, -1, -1, -1])
conv(image, Filter, "Laplacian 8")

Filter = np.array([-1, -1, -1, -1, 9, -1, -1, -1, -1])
conv(image, Filter, "Laplacian 9")
```

Programme 11.2.7: Laplacian filters used with `conv()` function defined in Programme 11.2.1.

Description	Filter	Results
Laplacian 4	$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix}$	
Laplacian 4 Sharpening	$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$	
Laplacian 8	$\begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$	
Laplacian 8 Sharpening	$\begin{pmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{pmatrix}$	

Table 11.2.6: Laplacian filters.

11.3 Intermediate Layers

11.3.1 Pooling Layer

In a convolutional layer, the raw input image photo matrix should be of a very high dimension, and so the computations of convolutions operation should be a huge amount. To relieve a bit of this complexity for the next convolutinoal layer, a pooling layer is usually, but not compulsory, introduced after a convolutional layer to the next, and this pooling uses the output of former convolutional layer as an input; typically, pooling also uses another filter tensors of size 2 or 3 and a stride of size $s_{sr} = 2$ in most common practice. There are no parameters to learn, but only hyperparameters have to be chosen in advance, *e.g.* filter and stride sizes. There are mainly two types of pooling:

1. **Max Pooling**: The maximum value of all entries in the moving frame will be returned as an output;
2. **Average Pooling**: The average value of all the entries enclosed in the moving window will serve as an output of the operation.

In practice, the max pooling is more popular in use than average pooling, and the former often gives better performance. Let $s_{\boldsymbol{\mathcal{I}}}^{(\ell)} \times s_{\boldsymbol{\mathcal{I}}}^{(\ell)} \times n_v^{(\ell)}$ be the input dimension of the pooling layer ℓ, $s_q^{(\ell)}$ as the dimension size of the moving frame, called the *pool size*, and $s_{sr}^{(\ell)}$ as the stride size of the pooling layer, then the output size is

$$s_O^{(\ell)} = \left\lfloor \frac{s_{\boldsymbol{\mathcal{I}}}^{(\ell)} - s_q^{(\ell)}}{s_{sr}^{(\ell)}} \right\rfloor + 1 \,. \tag{11.3.1}$$

In general, if a convolutional layer $\ell - 1$ is followed by this pooling layer ℓ, we have $n_v^{(\ell)} = n_f^{(\ell-1)}$; if another pooling layer $\ell - 1$ is followed by this pooling layer ℓ, we have $n_v^{(\ell)} = n_v^{(\ell-1)}$. As an illustration, in Figure 11.3.1, it has a pool size of 2, stride size of 2, and the max pooling is taken.

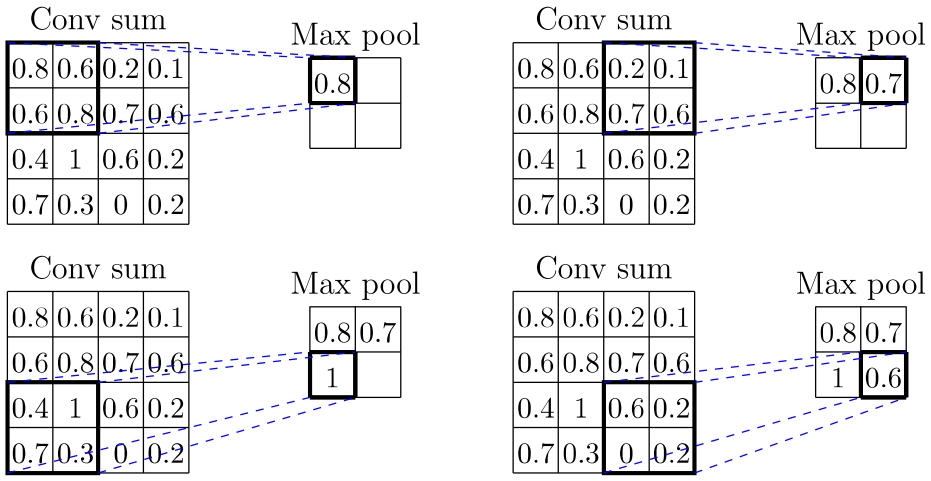


Figure 11.3.1: Max pooling with a pool size of 2 and a stride size of 2.

Therefore, using (11.3.1), the output size is

$$s_O = \left\lfloor \frac{4-2}{2} \right\rfloor + 1 = 2 \,.$$

Some well-known examples of CNN such as AlexNet, there are three pooling layers being inserted in the convolutional layers.

11.3.2 Connecting Convolutional Layers with Fully-connected Layers

After the convolutional layers that help extract various underlying patterns and profiles, we then use another deep neural network to analyze these abstract information for the very original purpose of classification. In this second stage, this DNN or classification layer, which produces the final outputs, is fed with integral values of labels y_n's, which stands for the categories to whose the input image belongs. Often, between the convolutional layers and the classification layer, there is one more layer connecting them, which helps compactify the input from the former to the latter. There are typically two types of this connecting layer as follows. Let say the output of the layer $\ell - 1$ is a 3-dimensional tensor of dimension $s_O^{(\ell-1)} \times s_O^{(\ell-1)} \times n_f^{(\ell-1)}$, this output serves as the input of this connecting layer ℓ with $s_{\boldsymbol{\mathcal{I}}}^{(\ell)} = s_O^{(\ell-1)}$ and $n_v^{(\ell)} = n_f^{(\ell-1)}$.

1. **Flattening layer**: The output tensor is flattened into a 1-dimensional very long vector as illustrated in Figure 11.3.2.

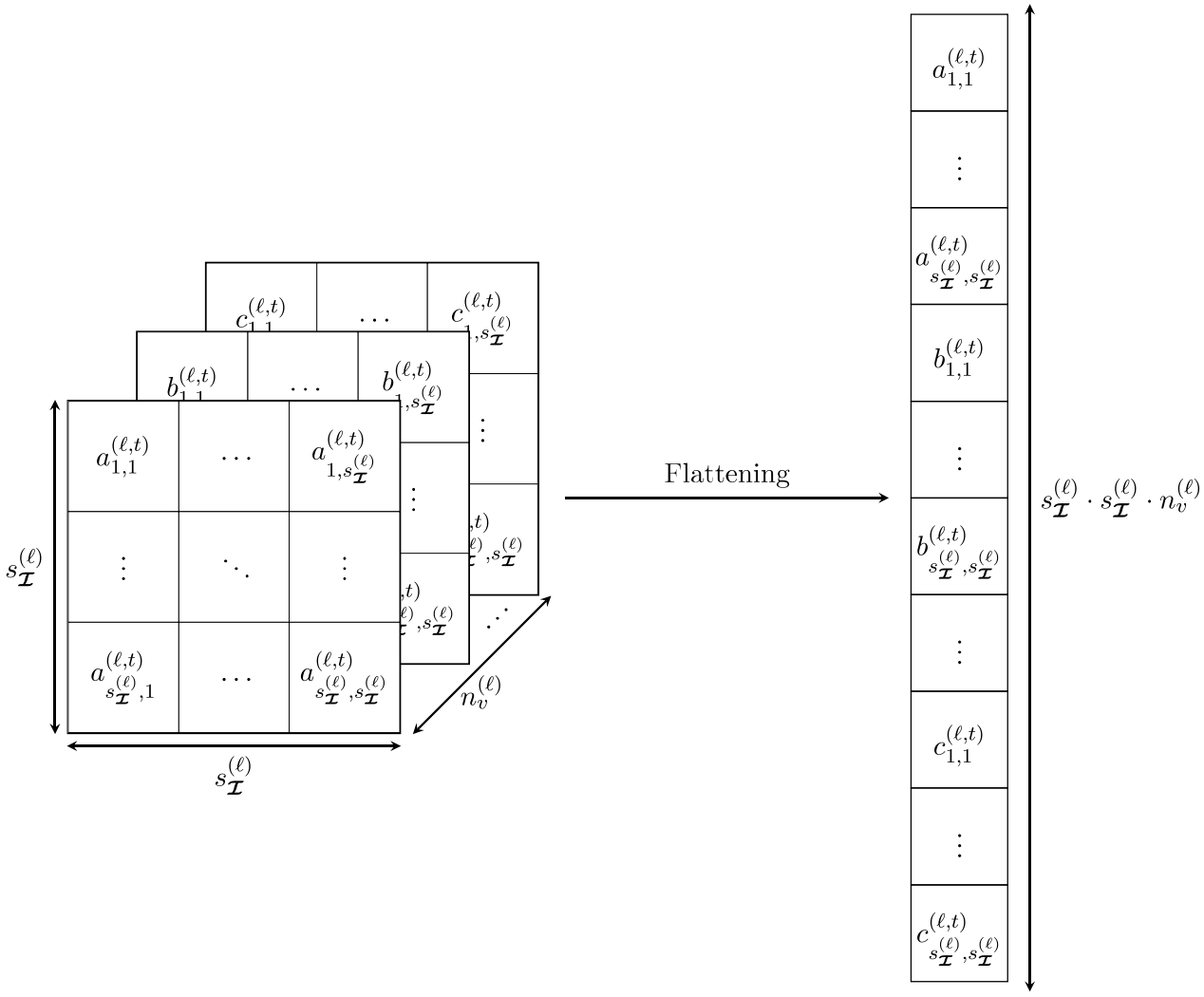


Figure 11.3.2: Flattening Layer ℓ.

This flattening technique is a traditional approach of transforming the output from the CNN as input for the classification layer. Clearly, there are so many coefficient parameters that connect the

flattening layer entries to the classification layer. To this point, some researcher suggested to introduce additional convolutional layers with fixed filter tensors or other dense neural network to summarize the information, and hence relieve the complexity caused by a large number of parameters from the flattening layer to the classification layer. The drawback of this proposal is that will increase the computational cost, which in turn takes a larger training time, let alone the possiblity of more severe overfitting.

2. **Global Average Pooling** (GAP): The average number of all entries is computed for the overall outputs of each filter tensor, which leads to only $n_v^{(\ell)}$ number of inputs for the classification layer, see Figure 11.3.3.

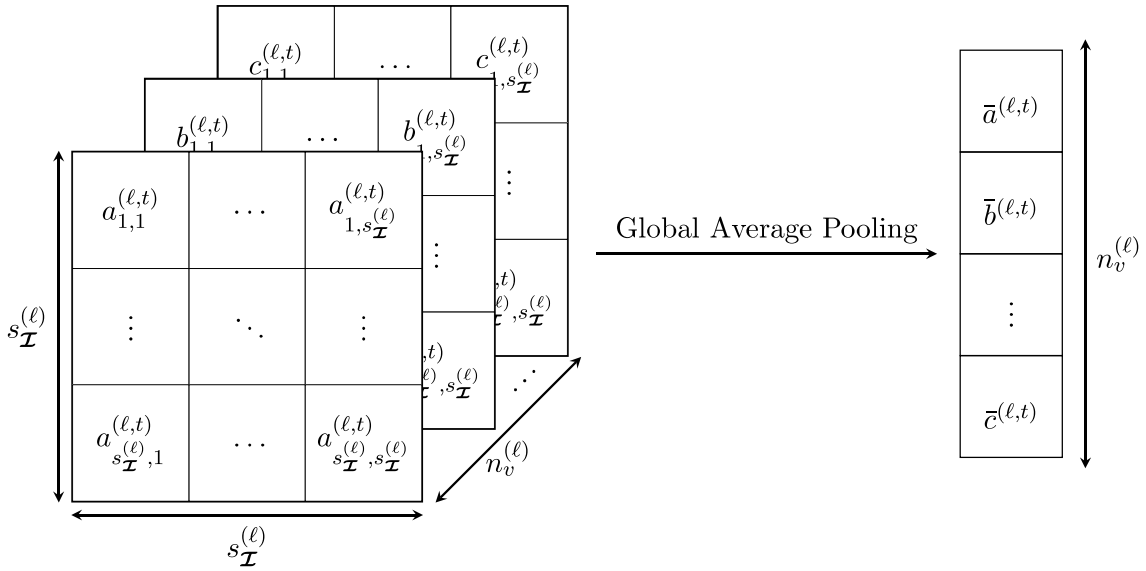


Figure 11.3.3: Global Average Pooling Layer ℓ.

Global Average Pooling layer obviously helps to resolve the large parameter issues as stated before, yet are also demand a large number $n_v^{(\ell)}$ of volume depth. More often, the magnitudes of these average values are usually interpreted as their own relative importance of the corresponding filter tensors for a particular purpose of classification. If an average value is small, that specific filter tensor plays a minimal role in predicting the label $\hat{y}_n$; for if an average value is large that filter is significant in determining the true label y_n. Usually, Global Average Pooling layer can be considered as another very big average pooling layer with a pool size being equal to the input size of the flattening layer, *i.e.* $s_q^{(\ell)} = s_{\mathcal{I}}^{(\ell)}$.

Finally, as a remark, the ordering of the entries in either Flattening or Global Average Pooling layers is not a matter, as long as the same order is kept for the later forward and backward propagation in the training of the whole CNN. Besides, there are also some other approaches such as *Global Maximum Pooling* (GMP), in which the maximum entries in the whole output of the filter tensor is sought, and again only $n_v^{(\ell)}$ number of outputs is generated; in practice, however, GMP is less welcomed by the practitioners.

11.4 Training a CNN with Fixed Filters Only

In this section, we first consider the simpler training with fixed filters only, in which the error rate will not be backward-propagated to train any coefficients in these filters in the convolutional layers. Consider the r^{th} filter in the ℓ^{th} convolutional layer with $n_f^{(\ell)}$ of these filters, where $r = 1, \cdots, n_f^{(\ell)}$. As before, we adopt the following notations:

1. i, j be the indices of the row and column respectively;
2. $\boldsymbol{X}^{(\ell)}$ represents the input for a convolutional layer or a pooling layer. In a convolutional layer, the input $\boldsymbol{X}^{(\ell)} = \boldsymbol{X}_1^{(\ell)} \otimes \cdots \otimes \boldsymbol{X}_{n_v^{(\ell)}}^{(\ell)}$ of the dimension of $s_{\boldsymbol{\mathcal{I}}}^{(\ell)} \times s_{\boldsymbol{\mathcal{I}}}^{(\ell)} \times n_v^{(\ell)}$ is often padded with zeros at the image boundary to form a newly augmented image $\tilde{\boldsymbol{X}}^{(\ell)} = \tilde{\boldsymbol{X}}_1^{(\ell)} \otimes \cdots \otimes \tilde{\boldsymbol{X}}_{n_v^{(\ell)}}^{(\ell)}$ of the dimension now $(s_{\boldsymbol{\mathcal{I}}}^{(\ell)} + 2 \cdot s_p^{(\ell)}) \times (s_{\boldsymbol{\mathcal{I}}}^{(\ell)} + 2 \cdot s_p^{(\ell)}) \times n_v^{(\ell)}$; as discussed before, the resulting input and output dimensions should agree even after the cross-correlation or convolution operations, in other words, $s_{\boldsymbol{\mathcal{I}}}^{(\ell)} = s_O^{(\ell)} = s_{\boldsymbol{\mathcal{I}}}^{(\ell+1)}$;
3. $\boldsymbol{F}_r^{(\ell)}$ is the r^{th} fixed filter with the dimension $s_f^{(\ell)} \times s_f^{(\ell)} \times n_v^{(\ell)}$, for $n_v^{(\ell)}$ being the chosen volume depth;
4. $b_r^{(\ell)}$ is a fixed bias with the dimension $1 \times 1 \times 1$ for the filter $\boldsymbol{F}_r^{(\ell)}$ and all volume depths to be combined toegether;
5. $\boldsymbol{Z}_r^{(\ell)}$ with entries $z_{r,i,j}^{(\ell)}$, for $i, j = 1, \cdots, s_O^{(\ell)}$, is the r^{th} weighted sum of the input $\boldsymbol{X}^{(\ell)}$ convoluted (but not cross-correlated) with $\boldsymbol{F}_r^{(\ell)}$ and the bias $b_r^{(\ell)}$ for the r^{th} filter;
6. $\boldsymbol{A}_r^{(\ell)}$ with entries $a_{r,i,j}^{(\ell)} = f_\ell(z_{r,i,j}^{(\ell)})$ is the final output of this convolutional layer ℓ by applying of the activation function f_ℓ to each entry $z_{r,i,j}^{(\ell,t)}$.

We next list out the step-by-step forward-propagation procedures with these fixed filters $\boldsymbol{F}_r^{(\ell)}$'s and biases $b_r^{(\ell)}$'s are as follows. There is no need to backward propagate here, the feed forward is such that:

1. (*Initialization*) For the input layer $\boldsymbol{X}^{(0)} = \boldsymbol{\mathcal{I}}$ with the dimension of $s_{\boldsymbol{\mathcal{I}}}^{(0)} \times s_{\boldsymbol{\mathcal{I}}}^{(0)} \times n_v^{(0)}$, where $n_v^{(0)}$ is the volumn depth of the input image $\boldsymbol{\mathcal{I}}$, it is also the output of this layer, and so it is just the input for the first hidden layer $\boldsymbol{X}^{(1)} = \boldsymbol{X}^{(0)}$ with a dimension of $s_{\boldsymbol{\mathcal{I}}}^{(1)} \times s_{\boldsymbol{\mathcal{I}}}^{(1)} \times n_v^{(1)}$;
2. (*Convolution*) For $\ell \geq 1$, padding the input $\boldsymbol{X}_h^{(\ell)}$, for $h = 1, \cdots, n_v^{(\ell)} = n_f^{(\ell-1)}$, with a padding size of $s_p^{(\ell)} = k = (s_f^{(\ell)} - 1)/2$, to form $\tilde{\boldsymbol{X}}_h^{(\ell)}$, after which we use the filter $\boldsymbol{F}_r^{(\ell)}$, for $r = 1, \cdots, n_f^{(\ell)}$, so as to calculate the weighted sum $z_{r,i,j}^{(\ell)}$ such that:

$$\boldsymbol{Z}_r^{(\ell)}[i,j] := z_{r,i,j}^{(\ell)} = \tilde{\boldsymbol{X}}_{(k+1)+(i-1)s_{sr}^{(\ell)},(k+1)+(j-1)s_{sr}^{(\ell)}}^{(\ell)}(s_f^{(\ell)}) * \boldsymbol{F}_r^{(\ell)} + b_r^{(\ell)}, \qquad i, j = 1, \cdots, s_O^{(\ell)}, \quad (11.4.1)$$

where the notation for the submatrix $\tilde{\boldsymbol{X}}_{i,j}^{(\ell)}(s_f^{(\ell)})$ has been introduced in Subsection 11.2.1, and it has a dimension of $s_f^{(\ell)} \times s_f^{(\ell)} \times n_v^{(\ell)}$, while $n_v^{(\ell)} = n_f^{(\ell-1)}$ and $s_O^{(\ell)}$ is given by (11.2.7) with the appropriate

arguments. In matrix form, we also write

$$\boldsymbol{Z}_r^{(\ell)} = \begin{pmatrix} z_{r,1,1}^{(\ell)} & \cdots & z_{r,1,s_O}^{(\ell)} \\ \vdots & \ddots & \vdots \\ z_{r,s_O,1}^{(\ell)} & \cdots & z_{r,s_0,s_O}^{(\ell)} \end{pmatrix}$$
$$= \begin{pmatrix} \tilde{\boldsymbol{X}}_{(k+1),(k+1)}^{(\ell)}(s_f^{(\ell)}) * \boldsymbol{F}_r^{(\ell)} + b_r^{(\ell)} & \cdots & \tilde{\boldsymbol{X}}_{(k+1),(k+1)+(s_O^{(\ell)}-1)s_{sr}^{(\ell)}}^{(\ell)}(s_f^{(\ell)}) * \boldsymbol{F}_r^{(\ell)} + b_r^{(\ell)} \\ \vdots & \ddots & \vdots \\ \tilde{\boldsymbol{X}}_{(k+1)+(s_O^{(\ell)}-1)s_{sr}^{(\ell)},(k+1)}^{(\ell)}(s_f^{(\ell)}) * \boldsymbol{F}_r^{(\ell)} + b_r^{(\ell)} & \cdots & \tilde{\boldsymbol{X}}_{(k+1)+(s_O^{(\ell)}-1)s_{sr}^{(\ell)},(k+1)+(s_O^{(\ell)}-1)s_{sr}^{(\ell)}}^{(\ell)}(s_f^{(\ell)}) * \boldsymbol{F}_r^{(\ell)} + b_r^{(\ell)} \end{pmatrix},$$

with a stride of size $s_{sr}^{(\ell)}$. Finally, we apply the activation function f_ℓ to all the entries of the weighted sum $\boldsymbol{Z}_r^{(\ell)}$,

$$\boldsymbol{A}_r^{(\ell)} = \begin{pmatrix} a_{r,1,1}^{(\ell)} & \cdots & a_{r,1,s_O^{(\ell)}}^{(\ell)} \\ \vdots & \ddots & \vdots \\ a_{r,s_O^{(\ell)},1}^{(\ell)} & \cdots & a_{r,s_O^{(\ell)},s_O^{(\ell)}}^{(\ell)} \end{pmatrix} = \begin{pmatrix} f_\ell(z_{r,1,1}^{(\ell)}) & \cdots & f_\ell(z_{r,1,s_O^{(\ell)}}^{(\ell)}) \\ \vdots & \ddots & \vdots \\ f_\ell(z_{r,s_O^{(\ell)},1}^{(\ell)}) & \cdots & f_\ell(z_{r,s_O^{(\ell)},s_O^{(\ell)}}^{(\ell)}) \end{pmatrix}.$$

The activation value being the output of this convolutional layer ℓ is now forward-propagated to the next layer $\ell+1$ such that $\boldsymbol{X}_r^{(\ell+1)} = \boldsymbol{A}_r^{(\ell)}$, for $r = 1, \cdots, n_f^{(\ell)} = n_v^{(\ell+1)}$.

3. (*Pooling*) A pooling layer with a stride size of $s_{sr}^{(\ell)}$ and a pool size of $s_q^{(\ell)} \times s_q^{(\ell)}$, can be chosen either as the *Max Pooling* or *Average Pooling*;

   (a) **Max Pooling**: The maximum value in the moving window of the pooling is returned:
   $$\boldsymbol{P}_h^{(\ell)}[i,j] := p_{h,i,j}^{(\ell)} = \max_{u,v \in \{1,2,\cdots,s_q^{(\ell)}\}} \left\{ \boldsymbol{X}_h^{(\ell)}[s_{sr}^{(\ell)} \cdot (i-1) + u, s_{sr}^{(\ell)} \cdot (j-1) + v] \right\};$$

   (b) **Average Pooling**: The average value in the moving pool is retured:
   $$\boldsymbol{P}_h^{(\ell)}[i,j] := p_{h,i,j}^{(\ell)} = \frac{1}{s_q^{(\ell)} \cdot s_q^{(\ell)}} \sum_{u=1}^{s_q^{(\ell)}} \sum_{v=1}^{s_q^{(\ell)}} \boldsymbol{X}_h^{(\ell)}[s_{sr}^{(\ell)} \cdot (i-1) + u, s_{sr}^{(\ell)} \cdot (j-1) + v],$$

   for $h = 1, \cdots n_v^{(\ell)}$, and $i, j = 1, \cdots, s_O^{(\ell)}$, where the pooling output size $s_O^{(\ell)}$ is determined by (11.3.1):
   $$s_O^{(\ell)} = \left\lfloor \frac{s_{\boldsymbol{\mathcal{I}}}^{(\ell)} - s_q^{(\ell)}}{s_{sr}^{(\ell)}} \right\rfloor + 1.$$

4. (*Flattening*) We use the notation $g_F : \mathbb{R}^{s_{\boldsymbol{\mathcal{I}}}^{(\ell)} \times s_{\boldsymbol{\mathcal{I}}}^{(\ell)} \times n_v^{(\ell)}} \to \mathbb{R}^{s_{\boldsymbol{\mathcal{I}}}^{(\ell)} \cdot s_{\boldsymbol{\mathcal{I}}}^{(\ell)} \cdot n_v^{(\ell)}}$ to denote the flattening function, $g_G : \mathbb{R}^{s_{\boldsymbol{\mathcal{I}}}^{(\ell)} \times s_{\boldsymbol{\mathcal{I}}}^{(\ell)} \times n_v^{(\ell)}} \to \mathbb{R}^{n_v^{(\ell)}}$ for the global average pooling function; and by definition, g_F is an invertible function. The flattening layer ℓ is fully connected to an MLP from the next layer onward, and this DNN serves as the classification analysis layer.

The common choices of fixed filters as listed out in in Subection 11.2.3 can be used in the above steps. The error rate and various sensitivities obtained in the MLP model will not backpropagate to the convolutional, pooling and flattening layers.

11.5 Training of CNN with Varying Filters (Kernels)

We next consider the case when the filters are allowed to be floating or varying so as to be calibrated by backpropagation. Same as in the previous section, we first introduce some relevant notions and notations at the t-step of the backpropagation. We then first discuss the feed-forward mechanism.

1. i, j are the indices for the row and column, respectively;
2. We have the same initialization $\boldsymbol{X}^{(0,t)} = \boldsymbol{\mathcal{I}} = \boldsymbol{X}^{(1,t)}$;
3. $\boldsymbol{X}^{(\ell,t)}$ is the input for a convolutional layer or a pooling layer $\ell \geq 1$. For a convolutional layer, the input $\boldsymbol{X}^{(\ell,t)} = \boldsymbol{X}_1^{(\ell,t)} \otimes \cdots \otimes \boldsymbol{X}_{n_v^{(\ell)}}^{(\ell,t)}$ of a dimension $s_{\boldsymbol{\mathcal{I}}}^{(\ell)} \times s_{\boldsymbol{\mathcal{I}}}^{(\ell)} \times n_v^{(\ell)}$ is again padded with zeros at the boundary so as to form $\tilde{\boldsymbol{X}}^{(\ell,t)} = \tilde{\boldsymbol{X}}_1^{(\ell,t)} \otimes \cdots \otimes \tilde{\boldsymbol{X}}_{n_v^{(\ell)}}^{(\ell,t)}$ with a dimension $(s_{\boldsymbol{\mathcal{I}}}^{(\ell)} + 2 \cdot s_p^{(\ell)}) \times (s_{\boldsymbol{\mathcal{I}}}^{(\ell)} + 2 \cdot s_p^{(\ell)}) \times n_v^{(\ell)}$, as before, $s_{\boldsymbol{\mathcal{I}}}^{(\ell)} = s_O^{(\ell)} = s_{\boldsymbol{\mathcal{I}}}^{(\ell+1)}$;
4. $\boldsymbol{K}^{(\ell,t)}$ is the kernel tensor to be trained of the convolutional layer ℓ with a dimension of $s_f^{(\ell)} \times s_f^{(\ell)} \times n_v^{(\ell)} \times n_f^{(\ell)}$, where $n_v^{(\ell)} = n_f^{(\ell-1)}$;
5. $\boldsymbol{A}^{(\ell,t)}$ is the output of the convolutional layer ℓ with a dimension of $s_O^{(\ell)} \times s_O^{(\ell)} \times n_f^{(\ell)}$;
6. $\boldsymbol{P}^{(\ell,t)}$ is the output of the pooling layer ℓ with a dimension of $s_O^{(\ell)} \times s_O^{(\ell)} \times n_f^{(\ell)}$;
7. $\boldsymbol{W}^{(\ell,t)}$ is the weight matrix used to calibrate the weighted sum for the dense layer ℓ of the MLP;
8. $\boldsymbol{b}^{(\ell,t)}$ is the bias used to compute the weighted sum for either the convolutional or the dense layer ℓ of the MLP.
9. $\boldsymbol{\mathcal{F}}^{(\ell,t)}$ is the output of the flattening layer ℓ, which converts the input tensor into a "long" one-dimensional vector in $\mathbb{R}^{s_{\boldsymbol{\mathcal{I}}}^{(\ell)} \cdot s_{\boldsymbol{\mathcal{I}}}^{(\ell)} \cdot n_v^{(\ell)}}$;
10. $\boldsymbol{a}^{(\ell,t)}$ is the activation value as an output of the dense layer ℓ of the MLP;

11.5.1 Forward Propagation

1. **Convolutional Layer**: For $\ell \geq 1$, we again pad the input $\boldsymbol{X}_h^{(\ell,t)}$, for $h = 1, \cdots, n_f^{(\ell-1)}$, with the padding size $s_p^{(\ell)} = k = (s_f^{(\ell)} - 1)/2$ defined in (11.2.6), to form $\tilde{\boldsymbol{X}}_h^{(\ell,t)}$, in this layer ℓ, and the weighted sum $\boldsymbol{Z}_r^{(\ell,t)}[i,j] \in \mathbb{R}$ can be computed by:

$$\begin{aligned}
&\boldsymbol{Z}_r^{(\ell,t)}[i,j] \\
&:= \tilde{\boldsymbol{X}}^{(\ell,t)}_{(k+1)+(i-1)s_{sr}^{(\ell)},(k+1)+(j-1)s_{sr}^{(\ell)}}(s_f^{(\ell)}) * \boldsymbol{K}_r^{(\ell,t-1)} + b_r^{(\ell,t-1)} \\
&= \sum_{h=1}^{n_v^{(\ell)}} \sum_{u=-k}^{k} \sum_{v=-k}^{k} \tilde{\boldsymbol{X}}_h^{(\ell,t)}[(k+1) + s_{sr}^{(\ell)}(i-1) - u, (k+1) + s_{sr}^{(\ell)}(j-1) - v] \cdot \boldsymbol{K}_{r,h}^{(\ell,t-1)}[u,v] + b_r^{(\ell,t-1)}, \quad (11.5.1)
\end{aligned}$$

here h is the index for the channel depth, from 1 to $n_v^{(\ell)} = n_f^{(\ell-1)}$, and $i, j = 1, \cdots, s_O^{(\ell)}$ so that $s_O^{(\ell)}$ is determined by (11.2.7) with the suitable arguments. After the convolution operation, the activation function f_ℓ is then applied to each component of the weighted sums $\boldsymbol{Z}_r^{(\ell,t)}$ such that:

$$\boldsymbol{A}_r^{(\ell)} = \boldsymbol{f}_\ell(\boldsymbol{Z}_r^{(\ell)}), \qquad r = 1, \cdots, n_f^{(\ell)},$$

and eventually, the activation value serves as the input for the next layer $\ell+1$, also note that $\boldsymbol{A}^{(\ell,t)} = \boldsymbol{A}_1^{(\ell)} \otimes \cdots \otimes \boldsymbol{A}_{n_f^{(\ell)}}^{(\ell)} = \boldsymbol{X}^{(\ell+1,t)}$, which is a tensor of volume depth $n_v^{(\ell+1)} = n_f^{(\ell)}$.

2. **Pooling Layer**: Again, the pooling layer ℓ, with a stride size of $s_{sr}^{(\ell)}$ and a pool size of $s_q^{(\ell)} \times s_q^{(\ell)}$, can be either *Max Pooling* or *Average Pooling*, we here have:

   (a) **Max Pooling**: The maximum value of all entries in the moving pool is returned:
   $$\boldsymbol{P}_h^{(\ell,t)}[i,j] := p_{h,i,j}^{(\ell,t)} = \max_{u,v \in \{1,2,\cdots,s_q^{(\ell)}\}} \left\{ \boldsymbol{X}_h^{(\ell,t)}[s_{sr}^{(\ell)} \cdot (i-1) + u, s_{sr}^{(\ell)} \cdot (j-1) + v] \right\};$$

   (b) **Average Pooling**: The average value of all the entries in the moving pool is returned:
   $$\boldsymbol{P}_h^{(\ell,t)}[i,j] := p_{h,i,j}^{(\ell,t)} = \frac{1}{s_q^{(\ell)} \cdot s_q^{(\ell)}} \sum_{u=1}^{s_q^{(\ell)}} \sum_{v=1}^{s_q^{(\ell)}} \boldsymbol{X}_h^{(\ell,t)}[s_{sr}^{(\ell)} \cdot (i-1) + u, s_{sr}^{(\ell)} \cdot (j-1) + v],$$

   for $h = 1, \cdots, n_v^{(\ell)} = n_f^{(\ell-1)}$ and $i, j = 1, \cdots, s_O^{(\ell)}$, where $s_O^{(\ell)}$ is determined by (11.3.1) with the appropriate arguments:
   $$s_O^{(\ell)} = \left\lfloor \frac{s_{\boldsymbol{\mathcal{I}}}^{(\ell)} - s_q^{(\ell)}}{s_{sr}^{(\ell)}} \right\rfloor + 1,$$
   finally, the output of the pooling layer ℓ is the tensor $\boldsymbol{P}^{(\ell,t)} = \boldsymbol{P}_1^{(\ell,t)} \otimes \cdots \otimes \boldsymbol{P}_{n_v^{(\ell)}}^{(\ell,t)}$.

3. **Flattening Layer**: The flattening function is again denoted by $g_F : \mathbb{R}^{s_{\boldsymbol{\mathcal{I}}}^{(\ell)} \times s_{\boldsymbol{\mathcal{I}}}^{(\ell)} \times n_v^{(\ell)}} \to \mathbb{R}^{s_{\boldsymbol{\mathcal{I}}}^{(\ell)} \cdot s_{\boldsymbol{\mathcal{I}}}^{(\ell)} \cdot n_v^{(\ell)}}$, and $g_G : \mathbb{R}^{s_{\boldsymbol{\mathcal{I}}}^{(\ell)} \times s_{\boldsymbol{\mathcal{I}}}^{(\ell)} \times n_v^{(\ell)}} \to \mathbb{R}^{n_v^{(\ell)}}$ stands for the global average pooling function. The flattening layer ℓ is fully connected to another classifying neural network MLP from the next hidden layer $\ell+1$ onward:
   $$\boldsymbol{a}^{(\ell,t)} = \boldsymbol{\mathcal{F}}^{(\ell,t)} = g(\boldsymbol{X}^{(\ell,t)}),$$
   where $g = g_F$ or g_G, and we later need to consider the inverse of g_F, namely $g_F^{-1} : \mathbb{R}^{(s_{\boldsymbol{\mathcal{I}}}^{(\ell)} \cdot s_{\boldsymbol{\mathcal{I}}}^{(\ell)} \cdot n_v^{(\ell)})} \to \mathbb{R}^{s_{\boldsymbol{\mathcal{I}}}^{(\ell)} \times s_{\boldsymbol{\mathcal{I}}}^{(\ell)} \times n_v^{(\ell)}}$, for the purpose of backwardly propagating the error rates,
   $$\boldsymbol{X}^{(\ell-1,t)} = g_F^{-1}(\boldsymbol{\mathcal{F}}^{(\ell,t)}) = g_F^{-1}(\boldsymbol{a}^{(\ell,t)}).$$

4. **Dense Layer for classification**: An MLP can be fully-connected to the flattening layer, for the detailed discussion of the architecture and theory of MLP, we refer back to Chapter 10, we here only recall the symbols for the introduction to back propagation training for CNN, so the output of this layer ℓ:
   $$\boldsymbol{a}^{(\ell,t)} = \boldsymbol{f}_\ell \left(\boldsymbol{W}^{(\ell,t-1)} \boldsymbol{a}^{(\ell-1,t)} + \boldsymbol{b}^{(\ell,t-1)} \right).$$

5. **Output Layer of the DNN**: The output of the whole CNN is the output layer L of the MLP. *Softmax* function (resp. logistic function) often serves as the activation function for d_L-multi-class (resp. binary) classifiation:
   $$\boldsymbol{a}^{(L,t)} = \boldsymbol{f}_L \left(\boldsymbol{W}^{(L,t-1)} \boldsymbol{a}^{(L-1,t)} + \boldsymbol{b}^{(L,t-1)} \right).$$
   Moreover, as CNN is normally used for d_L-multi-class classification problem[15], the well-known *binary-*

[15] In a d_L-multi-label regression problem, the MSE loss function is usually evaluated. Since we have discussed the regression problem in deep neural network, particularly with MSE as the loss function, throughout Chapters 9 and 10, we here mainly focus on the classification problem.

cross-entropy is commonly used for the loss function:

$$\mathcal{E}^{(t)} = -\frac{1}{d_L}\sum_{p=1}^{d_L}\left[\tilde{y}_p^{(t)}\cdot\ln a_p^{(L,t)} + (1-\tilde{y}_p^{(t)})\cdot\ln\left(1-a_p^{(L,t)}\right)\right], \tag{11.5.2}$$

where for a given label y taking values $c_1,\cdots,c_{d_L}$, $\tilde{y}_p^{(t)} = 1$ if $y = c_p$, or zero otherwise, for $p = 1,\cdots,d_L$. Actually, d_L times of binary-cross-entropy of one single datum is just the negative value of log-likelihood of this datum; generally for a minibatch of size M:

$$\mathcal{E}^{(t)} = \frac{1}{M}\sum_{m=1}^{M}\mathcal{E}_m^{(t)}, \qquad \text{where} \quad \mathcal{E}_m^{(t)} = -\frac{1}{d_L}\sum_{p=1}^{d_L}\left[\tilde{y}_{mp}^{(t)}\cdot\ln a_{mp}^{(L,t)} + (1-\tilde{y}_{mp}^{(t)})\cdot\ln\left(1-a_{mp}^{(L,t)}\right)\right], \tag{11.5.3}$$

and so $d_L\cdot M\cdot\mathcal{E}^{(t)}$ is the negative of the log-likelihood of the minibatch of sample.

11.5.2 Backpropagation Procedure

We again introduce the backward procedure working from the last output layer first and then one layer by another backwardly as below:

1. **Output Layer**: With the binary cross-entropy function (11.5.2) instead of square loss function, the error rate can be computed as

$$\delta_p^{(L,t)} := \frac{\partial\mathcal{E}^{(t)}}{\partial z_p^{(L,t)}} = \frac{\partial\mathcal{E}^{(t)}}{\partial a_p^{(L,t)}}\frac{\partial a_p^{(L,t)}}{\partial z_p^{(L,t)}} = \frac{1}{d_L}\frac{a_p^{(L,t)}-\tilde{y}_p^{(t)}}{a_p^{(L,t)}(1-a_p^{(L,t)})}\cdot f_L'\left(z_p^{(L,t)}\right), \qquad \text{for} \quad p=1,\cdots,d_L, \tag{11.5.4}$$

   where the index p is the only one point selected for this round of update for the SGD. The weights and bias at this output layer L are updated by using the following sensitivities for $p = 1,\cdots,d_L$, $j = 1,\cdots,d_{L-1}$:

$$\frac{\partial\mathcal{E}^{(t)}}{\partial b_p^{(L,t-1)}} = \delta_p^{(L,t)},$$
$$\frac{\partial\mathcal{E}^{(t)}}{\partial\omega_{pj}^{(L,t-1)}} = \delta_p^{(L,t)}\cdot a_j^{(L-1,t)}.$$

2. **Dense (Hidden) Layer of the MLP**: We can backpropagate the error rate $\boldsymbol{\delta}^{(\ell+1,t)}$ to the current layer ℓ, let say without the batch normalization, through the following recursive relation; also see Subsubsection 10.1.1.2 for more details:

$$\boldsymbol{\delta}^{(\ell,t)} = \left[(\boldsymbol{W}^{(\ell+1,t-1)})^\top\cdot\boldsymbol{\delta}^{(\ell+1,t)}\right]\odot\boldsymbol{f}_\ell'\left(\boldsymbol{z}^{(\ell,t)}\right),$$

   with which the weights and bias at this layer ℓ are updated with the use of the following sensitivities:

$$\frac{\partial\mathcal{E}^{(t)}}{\partial b_j^{(\ell,t-1)}} = \delta_j^{(\ell,t)},$$
$$\frac{\partial\mathcal{E}^{(t)}}{\partial\omega_{ji}^{(\ell,t-1)}} = \delta_j^{(\ell,t)}\cdot a_i^{(\ell-1,t)},$$

   for $i = 1,\cdots,d_{\ell-1}$ and $j = 1,\cdots,d_\ell$.

3. **Flattening Layer**: As the flattening layer ℓ possesses no actual calculations except collapsing a matrix input into a column vector, the error rate is:

$$\boldsymbol{\delta}^{(\ell,t)} = (\boldsymbol{W}^{(\ell+1,t-1)})^\top\cdot\boldsymbol{\delta}^{(\ell+1,t)}.$$

   Besides, for the former layer $\ell-1$, we have two possibilities if a convolutional layer $\ell-1$ is connected

to this flattening layer ℓ, we have the sensitivity of $\mathcal{E}^{(t)}$ with respect to the activation value $\boldsymbol{\delta}_{\boldsymbol{A}}^{(\ell-1,t)} = g_F^{-1}(\boldsymbol{\delta}^{(\ell,t)})$, due to the coordinatewise definition of g_F; otherwise, if a pooling layer $\ell-1$ is connected to this flattening layer ℓ, we also have $\boldsymbol{\delta}_{\boldsymbol{P}}^{(\ell-1,t)} = g_F^{-1}(\boldsymbol{\delta}^{(\ell,t)})$.

4. **Pooling Layer**: With the error rate $\boldsymbol{\delta}_{\boldsymbol{P}}^{(\ell,t)}$ with respect to the output of this pooling layer ℓ, we backpropagate this error rate to its input $\boldsymbol{X}^{(\ell,t)}$:

   (a) **Max Pooling**: Since the maximum value is returned in this pooling process, only the argument where the maximum value is returned receives its error rate propagation, and now for $h = 1, \cdots, n_v^{(\ell)}$, the sensitivity of $\mathcal{E}^{(t)}$ with respect to the input $\boldsymbol{X}_h$ is:
   $$\boldsymbol{\delta}_{\boldsymbol{X}_h}^{(\ell,t)}[s_{sr}^{(\ell)} \cdot (i-1) + \hat{u}, s_{sr}^{(\ell)} \cdot (j-1) + \hat{v}] = \boldsymbol{\delta}_{\boldsymbol{P}_h}^{(\ell,t)}[i,j]\,,$$
   if $\boldsymbol{P}_h^{(\ell,t)}[i,j] := p_{h,i,j}^{(\ell,t)} = \boldsymbol{X}_h^{(\ell,t)}[s_{sr}^{(\ell)} \cdot (i-1) + \hat{u}, s_{sr}^{(\ell)} \cdot (j-1) + \hat{v}]$, or taking a value zero otherwise, where $i, j = 1, \cdots, s_O^{(\ell)}$, such that the dimension of pooling output $s_O^{(\ell)}$ was determined by (11.3.1) and for some $\hat{u}, \hat{v} = 1, 2, \cdots, s_q^{(\ell)}$, here we also recall the pooling dimension of size $s_q^{(\ell)}$, and $(\hat{u}, \hat{v})$ is the coordinate for the argument where the maximum value is achieved. Some remarks we can add here; indeed, there could be more than one pooling windows in which the common $\boldsymbol{X}_h^{(\ell,t)}[s_{sr}^{(\ell)} \cdot (i-1) + \hat{u}, s_{sr}^{(\ell)} \cdot (j-1) + \hat{v}]$ is the place where the corresponding maxima are achieved at these different pooling windows, yet the sensitivities due to the error backpropagation calculated should still agree. Secondly, not every iteration, all the components of $\boldsymbol{X}_h^{(\ell)}$ will be updated non-trivially, in fact, there could be quite many that would not be updated indefinitely, but still, they will all be updated eventually at some later iterations t.

   (b) **Average Pooling**: If the average value is returned, the sensitivities of $\mathcal{E}^{(t)}$ with respect $\boldsymbol{X}_h^{(\ell,t)}$'s components are:
   $$\boldsymbol{\delta}_{\boldsymbol{X}_h}^{(\ell,t)}[s_{sr}^{(\ell)} \cdot (i-1) + u, s_{sr}^{(\ell)} \cdot (j-1) + v] = \frac{1}{(s_q^{(\ell)})^2} \boldsymbol{\delta}_{\boldsymbol{P}_h}^{(\ell,t)}[i,j]\,,$$
   where $i, j = 1, \cdots, s_O^{(\ell)}$, where $s_O^{(\ell)}$ is calculated by (11.3.1), and for every $u, v = 1, 2, \cdots, s_q^{(\ell)}$ now, in contrast to the part 4a above, all components of $\boldsymbol{X}_h^{(\ell)}$ are now updated in each iteration !

   Furthermore, if the previous layer $\ell-1$ is also a pooling layer, we have $\boldsymbol{\delta}_{\boldsymbol{P}}^{(\ell-1,t)} = \boldsymbol{\delta}_{\boldsymbol{P}_1}^{(\ell-1,t)} \otimes \cdots \otimes \boldsymbol{\delta}_{\boldsymbol{P}_{n_v^{(\ell-1)}}}^{(\ell-1,t)} = \boldsymbol{\delta}_{\boldsymbol{X}}^{(\ell,t)} = \boldsymbol{\delta}_{\boldsymbol{X}_1}^{(\ell,t)} \otimes \cdots \otimes \boldsymbol{\delta}_{\boldsymbol{X}_{n_v}^{(\ell)}}^{(\ell,t)}$; otherwise, if the previous layer $\ell-1$ is a convolutional layer, we then have $\boldsymbol{\delta}_{\boldsymbol{A}}^{(\ell-1,t)} = \boldsymbol{\delta}_{\boldsymbol{A}_1}^{(\ell-1,t)} \otimes \cdots \otimes \boldsymbol{\delta}_{\boldsymbol{A}_{n_f^{(\ell-1)}}}^{(\ell-1,t)} = \boldsymbol{\delta}_{\boldsymbol{X}}^{(\ell,t)}$.

5. **Convolutional Layer**: With the error rate $\boldsymbol{\delta}_{\boldsymbol{A}}^{(\ell,t)}$ with respect to the activation function value, the error rate $\boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}$, the sensitivity of $\mathcal{E}^{(t)}$ with respect to the weighted sum $\boldsymbol{Z}_r$, is:
   $$\boldsymbol{\delta}_{\boldsymbol{Z}_r}^{(\ell,t)}[i,j] = \boldsymbol{\delta}_{\boldsymbol{A}_r}^{(\ell,t)}[i,j] \cdot f_\ell'(\boldsymbol{Z}_r^{(\ell,t)}[i,j])\,,$$
   By the law of total derivative, the bias is updated via the following sensitivities:
   $$\frac{\partial \mathcal{E}^{(t)}}{\partial b_r^{(\ell,t-1)}} = \sum_{i=1}^{s_O^{(\ell)}} \sum_{j=1}^{s_O^{(\ell)}} \boldsymbol{\delta}_{\boldsymbol{Z}_r}^{(\ell,t)}[i,j] \cdot \frac{\partial \boldsymbol{Z}_r^{(\ell,t)}[i,j]}{\partial b_r^{(\ell,t-1)}} = \sum_{i=1}^{s_O^{(\ell)}} \sum_{j=1}^{s_O^{(\ell)}} \boldsymbol{\delta}_{\boldsymbol{Z}_r}^{(\ell,t)}[i,j]\,.$$

Recall that the weighted sum $\boldsymbol{Z}_r^{(\ell,t)}[i,j] \in \mathbb{R}$ in (11.5.1) can be expressed as

$$\begin{aligned}\boldsymbol{Z}_r^{(\ell,t)}[i,j] &= \tilde{\boldsymbol{X}}^{(\ell,t)}_{(k+1)+(i-1)s_{sr}^{(\ell)},(k+1)+(j-1)s_{sr}^{(\ell)}}(s_f^{(\ell)}) * \boldsymbol{K}_r^{(\ell,t-1)} + b_r^{(\ell,t-1)} \\ &= \tilde{\boldsymbol{X}}^{(\ell,t)}_{(k+1)+(i-1)s_{sr}^{(\ell)},(k+1)+(j-1)s_{sr}^{(\ell)}}(s_f^{(\ell)}) \star [\boldsymbol{K}_r^{(\ell,t-1)}]_{\mathrm{rot}\pi} + b_r^{(\ell,t-1)}\,. \end{aligned} \tag{11.5.5}$$

We first consider the partial derivative of the weighted sum $\boldsymbol{Z}_r^{(\ell,t)}[i,j]$ with respect to the rotated kernel $\boldsymbol{K}_r^{(\ell,t-1)}$:

$$\nabla_{[\boldsymbol{K}_r^{(\ell,t-1)}]_{\mathrm{rot}\pi}} \boldsymbol{Z}_r^{(\ell,t)}[i,j] = \tilde{\boldsymbol{X}}^{(\ell,t)}_{(k+1)+(i-1)s_{sr}^{(\ell)},(k+1)+(j-1)s_{sr}^{(\ell)}}(s_f^{(\ell)})\,, \tag{11.5.6}$$

therefore, the gradient of $\mathcal{E}^{(t)}$ with respect to the r^{th} rotated kernel, for $r = 1, \cdots, n_f^{(\ell)}$, is:

$$\begin{aligned}\nabla_{[\boldsymbol{K}_r^{(\ell,t-1)}]_{\mathrm{rot}\pi}} \mathcal{E}^{(t)} &= \sum_{i=1}^{s_O^{(\ell)}} \sum_{j=1}^{s_O^{(\ell)}} \boldsymbol{\delta}_{\boldsymbol{Z}_r}^{(\ell,t)}[i,j] \cdot \nabla_{[\boldsymbol{K}_r^{(\ell,t-1)}]_{\mathrm{rot}\pi}} \boldsymbol{Z}_r^{(\ell,t)}[i,j] \\ &= \sum_{i=1}^{s_O^{(\ell)}} \sum_{j=1}^{s_O^{(\ell)}} \boldsymbol{\delta}_{\boldsymbol{Z}_r}^{(\ell,t)}[i,j] \cdot \tilde{\boldsymbol{X}}^{(\ell,t)}_{(k+1)+(i-1)s_{sr}^{(\ell)},(k+1)+(j-1)s_{sr}^{(\ell)}}(s_f^{(\ell)})\,. \end{aligned} \tag{11.5.7}$$

In order to express it in a more neat form, we first define a notion of extending a matrix $\boldsymbol{A}^{(\ell,t)}$ of size $s_O^{(\ell)} \times s_O^{(\ell)}$ to $\bar{\boldsymbol{A}}^{(\ell,t)}$ such that the entries $\bar{\boldsymbol{A}}^{(\ell,t)}[i',j'] = \boldsymbol{A}^{(\ell,t)}[i,j]$ if

$$i' = 1 + (i-1)s_{sr}^{(\ell)} \qquad \text{and} \qquad j' = 1 + (j-1)s_{sr}^{(\ell)}\,, \quad \text{for} \quad i,j = 1, \cdots, s_O^{(\ell)}\,; \tag{11.5.8}$$

otherwise $\bar{\boldsymbol{A}}^{(\ell,t)}[i',j'] = 0$. The purpose of introducing this extension is to simplify the notation as if the stride size $s_{sr}^{(\ell)}$ were 1; for if $s_{sr}^{(\ell)} = 1$, by then it is always $s_O^{(\ell)} = s_{\mathcal{I}}^{(\ell)}$, leading to $\bar{\boldsymbol{A}}^{(\ell,t)} = \boldsymbol{A}^{(\ell,t)}$, by a simple matching argument. As i,j take values from $1, \cdots, s_O^{(\ell)}$ onward, we have $1 + (s_O^{(\ell)} - 1)s_{sr}^{(\ell)} = 1 + (s_{\mathcal{I}}^{(\ell)} - 1) \cdot 1 = s_{\mathcal{I}}^{(\ell)}$, and thus i', j' take values $1, \cdots, s_{\mathcal{I}}^{(\ell)}$. With this in mind, we clearly have $\overline{\boldsymbol{\delta}_{\boldsymbol{Z}_r}^{(\ell,t)}} = \boldsymbol{\delta}_{\bar{\boldsymbol{Z}}_r}^{(\ell,t)}$, and (11.5.7) can be further simplified as:

$$\nabla_{[\boldsymbol{K}_r^{(\ell,t-1)}]_{\mathrm{rot}\pi}} \mathcal{E}^{(t)} = \sum_{i'=1}^{s_{\mathcal{I}}^{(\ell)}} \sum_{j'=1}^{s_{\mathcal{I}}^{(\ell)}} \overline{\boldsymbol{\delta}_{\boldsymbol{Z}_r}^{(\ell,t)}}[i',j'] \cdot \tilde{\boldsymbol{X}}^{(\ell,t)}_{k+i',k+j'}(s_f^{(\ell)}) = \tilde{\boldsymbol{X}}^{(\ell,t)} \star \boldsymbol{\delta}_{\bar{\boldsymbol{Z}}_r}^{(\ell,t)}\,. \tag{11.5.9}$$

Recall the notation for matrix derivative introduced in Subsection 3.4.1, we also make use of the following useful lemma:

Lemma 11.5.1. Consider a real-valued function $g : \mathbb{R}^{Q\times Q} \to \mathbb{R}$ of a square matrix of independent variables $\boldsymbol{X} \in \mathbb{R}^{Q\times Q}$, we have

$$\nabla_{\boldsymbol{X}_{\mathrm{rot}\pi}} g(\boldsymbol{X}) = [\nabla_{\boldsymbol{X}} g(\boldsymbol{X})]_{\mathrm{rot}\pi}\,.$$

Its proof immediately follows from the very definition, and we leave it as an exercise for readers.

Therefore, as $\mathcal{E}^{(t)}$ is a scalar, using Lemma 11.5.1,

$$\nabla_{\boldsymbol{K}_r^{(\ell,t-1)}} \mathcal{E}^{(t)} = [\tilde{\boldsymbol{X}}^{(\ell,t)} \star \boldsymbol{\delta}_{\bar{\boldsymbol{Z}}_r}^{(\ell,t)}]_{\mathrm{rot}\pi}\,, \qquad \text{for} \quad r = 1, \cdots, n_f^{(\ell)}\,, \tag{11.5.10}$$

or equivalently

$$\nabla_{\boldsymbol{K}_r^{(\ell,t-1)}} \mathcal{E}^{(t)} = \sum_{i=1}^{s_O^{(\ell)}} \sum_{j=1}^{s_O^{(\ell)}} \boldsymbol{\delta}_{\boldsymbol{Z}_r}^{(\ell,t)}[i,j] \cdot \left[\tilde{\boldsymbol{X}}^{(\ell,t)}_{(k+1)+(i-1)s_{sr}^{(\ell)},(k+1)+(j-1)s_{sr}^{(\ell)}}(s_f^{(\ell)})\right]_{\mathrm{rot}\pi}\,. \tag{11.5.11}$$

Next, we have to backward-propagate the error rate $\boldsymbol{\delta}_{\boldsymbol{Z}_r}^{(\ell,t)}$ (or $\overline{\boldsymbol{\delta}_{\boldsymbol{Z}_r}^{(\ell,t)}}$) with respect to the weighted sum $\boldsymbol{Z}_r^{(\ell,t)}$ (or $\bar{\boldsymbol{Z}}_r^{(\ell,t)}$), to that of input $\boldsymbol{\delta}_{\tilde{\boldsymbol{X}}}^{(\ell,t)}$. The gradient of the extended weighted sum $\bar{\boldsymbol{Z}}_r^{(\ell,t)}[i',j']$ with

respect to $\tilde{\boldsymbol{X}}^{(\ell,t)}_{h,k+i',k+j'}(s_f^{(\ell)})$ is computed as:

$$\begin{aligned}\nabla_{\tilde{\boldsymbol{X}}^{(\ell,t)}_{h,k+i',k+j'}(s_f^{(\ell)})}\bar{\boldsymbol{Z}}_r^{(\ell,t)}[i',j'] &= \nabla_{\tilde{\boldsymbol{X}}^{(\ell,t)}_{h,k+i',k+j'}(s_f^{(\ell)})}\boldsymbol{Z}_r^{(\ell,t)}[i,j]\\ &= \nabla_{\tilde{\boldsymbol{X}}^{(\ell,t)}_{h,k+i',k+j'}(s_f^{(\ell)})}\left(\tilde{\boldsymbol{X}}^{(\ell,t)}_{h,k+i',k+j'}(s_f^{(\ell)}) \star [\boldsymbol{K}^{(\ell,t-1)}_{r,h}]_{\text{rot}\pi} + b_r^{(\ell,t-1)}\right)\\ &= [\boldsymbol{K}^{(\ell,t-1)}_{r,h}]_{\text{rot}\pi}\,,\end{aligned}$$

for i',j' satisfying (11.5.8); otherwise $\nabla_{\tilde{\boldsymbol{X}}^{(\ell,t)}_{h,k+i',k+j'}(s_f^{(\ell)})}\bar{\boldsymbol{Z}}_r^{(\ell,t)}[i',j'] = 0$ as $\bar{\boldsymbol{Z}}_r^{(\ell,t)}[i',j'] \equiv 0$. Note that $\bar{\boldsymbol{Z}}_r^{(\ell,t)}$ and $\boldsymbol{\delta}^{(\ell,t)}_{\bar{\boldsymbol{Z}}_r}$ with a padding size of $s_p^{(\ell)}$, resulting into two matrices $\tilde{\bar{\boldsymbol{Z}}}_r^{(\ell,t)}$ and $\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\bar{\boldsymbol{Z}}}_r}$ both of size $(2s_p^{(\ell)} + s_{\mathcal{I}}^{(\ell)}) \times (2s_p^{(\ell)} + s_{\mathcal{I}}^{(\ell)})$. Therefore, the gradient $\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\boldsymbol{X}}^{(\ell,t)}_{h,k+i',k+j'}(s_f^{(\ell)})}$ of the loss $\mathcal{E}^{(t)}$ with respect to $\tilde{\boldsymbol{X}}^{(\ell,t)}_{h,k+i',k+j'}(s_f^{(\ell)})$, for $i',j' = 1,\cdots,s_{\mathcal{I}}^{(\ell)}$ defined in (11.5.8), is computed as:

$$\begin{aligned}\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\boldsymbol{X}}^{(\ell,t)}_{h,k+i',k+j'}(s_f^{(\ell)})} &= \sum_{r=1}^{n_f^{(\ell)}} \boldsymbol{\delta}^{(\ell,t)}_{\tilde{\bar{\boldsymbol{Z}}}_r}[k+i',k+j'] \cdot \nabla_{\tilde{\boldsymbol{X}}^{(\ell,t)}_{h,k+i',k+j'}(s_f^{(\ell)})}\tilde{\bar{\boldsymbol{Z}}}_r^{(\ell,t)}[k+i',k+j']\\ &= \sum_{r=1}^{n_f^{(\ell)}} \boldsymbol{\delta}^{(\ell,t)}_{\tilde{\bar{\boldsymbol{Z}}}_r}[k+i',k+j'] \cdot \nabla_{\tilde{\boldsymbol{X}}^{(\ell,t)}_{h,k+i',k+j'}(s_f^{(\ell)})}\bar{\boldsymbol{Z}}_r^{(\ell,t)}[i',j']\\ &= \sum_{r=1}^{n_f^{(\ell)}} \boldsymbol{\delta}^{(\ell,t)}_{\tilde{\bar{\boldsymbol{Z}}}_r}[k+i',k+j'] \cdot [\boldsymbol{K}^{(\ell,t-1)}_{r,h}]_{\text{rot}\pi}\,. \qquad (11.5.12)\end{aligned}$$

However, (11.5.12) corresponds the impact of one specific patch of shape $s_f^{(\ell)} \times s_f^{(\ell)}$ centered at the location $(h,k+i',k+j')$ in $\boldsymbol{\delta}_{\tilde{\boldsymbol{X}}_h}$. Firstly, denote (h,a,b) be a location in $\boldsymbol{\delta}_{\tilde{\boldsymbol{X}}_h}$, which is contained in the patch $\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\boldsymbol{X}}^{(\ell,t)}_{h,k+i',k+j'}(s_f^{(\ell)})}$, there can be several other patches centered at some other different $(h,k+i',k+j')$ so that they all overlap at this common point (h,a,b), each of these patches contributes a portion of the error rate at (h,a,b) and so we have to sum all the effects of them up to obtain the overall error rate at (h,a,b). Secondly, since during the foward propagation, we performed padding to the input matrix $\boldsymbol{X}^{(\ell,t)}$, we have to remove the boundary entries in $\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\boldsymbol{X}}_h}$, for $h = 1,\cdots,s_v^{(\ell)}$, in order to obtain purely $\boldsymbol{\delta}^{(\ell,t)}_{\boldsymbol{X}_h}$. Taking these two discussion into account, inspired by Subsection 11.5.3, we therefore have, for $h = 1,\cdots,n_v^{(\ell)} = n_f^{(\ell-1)}$, $a,b = 1,\cdots,s_{\mathcal{I}}^{(\ell)}$,

$$\begin{aligned}\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\boldsymbol{X}}_h}[k+a,k+b] = \boldsymbol{\delta}^{(\ell,t)}_{\boldsymbol{X}_h}[a,b] &= \sum_{u=-k}^{k}\sum_{v=-k}^{k}\left(\sum_{r=1}^{n_f^{(\ell)}} \boldsymbol{\delta}^{(\ell,t)}_{\tilde{\bar{\boldsymbol{Z}}}_r}[k+a+u,k+b+v] \cdot \boldsymbol{K}^{(\ell,t-1)}_{r,h}[u,v]\right)\\ &= \sum_{r=1}^{n_f^{(\ell)}}\left(\sum_{u=-k}^{k}\sum_{v=-k}^{k} \boldsymbol{\delta}^{(\ell,t)}_{\tilde{\bar{\boldsymbol{Z}}}_r}[k+a+u,k+b+v] \cdot \boldsymbol{K}^{(\ell,t-1)}_{r,h}[u,v]\right)\\ &= \sum_{r=1}^{n_f^{(\ell)}} \boldsymbol{\delta}^{(\ell,t)}_{\tilde{\bar{\boldsymbol{Z}}}_r,k+a,k+b}(s_f^{(\ell)}) \star \boldsymbol{K}^{(\ell,t-1)}_{r,h}\,. \qquad (11.5.13)\end{aligned}$$

In the literature, we often write (11.5.13) in the following neat form, which is understood in componentwise sense in (11.5.13):

$$\boldsymbol{\delta}^{(\ell,t)}_{\boldsymbol{X}_h} = \sum_{r=1}^{n_f^{(\ell)}} \boldsymbol{\delta}^{(\ell,t)}_{\tilde{\bar{\boldsymbol{Z}}}_r} \star \boldsymbol{K}^{(\ell,t-1)}_{r,h}\,, \qquad h = 1,\cdots,n_v^{(\ell)} = n_f^{(\ell-1)}\,. \qquad (11.5.14)$$

If the previous layer $\ell - 1$ is a pooling layer, we simply have $\boldsymbol{\delta}_{\boldsymbol{P}}^{(\ell-1,t)} = \boldsymbol{\delta}_{\boldsymbol{X}}^{(\ell,t)}$; otherwise, if the previous layer $\ell - 1$ is another convolutional layer, we have $\boldsymbol{\delta}_{\boldsymbol{A}}^{(\ell-1,t)} = \boldsymbol{\delta}_{\boldsymbol{X}}^{(\ell,t)}$.

11.5.3 Illustrative Examples

The above derivations could be quite abstract, and we shall illustrate the main idea behind through the following fairly useful example. For simplicity, the stride size is taken as $s_{sr}^{(\ell)} = 1$. However, for the sake of clearer illustration, we omit the superscripts, representing the layer ℓ and iteration step t, of all symbols in all the figures below.

(I) The forward propagation procedures of a 3×3 image with a 3×3 filter can be depicted as:

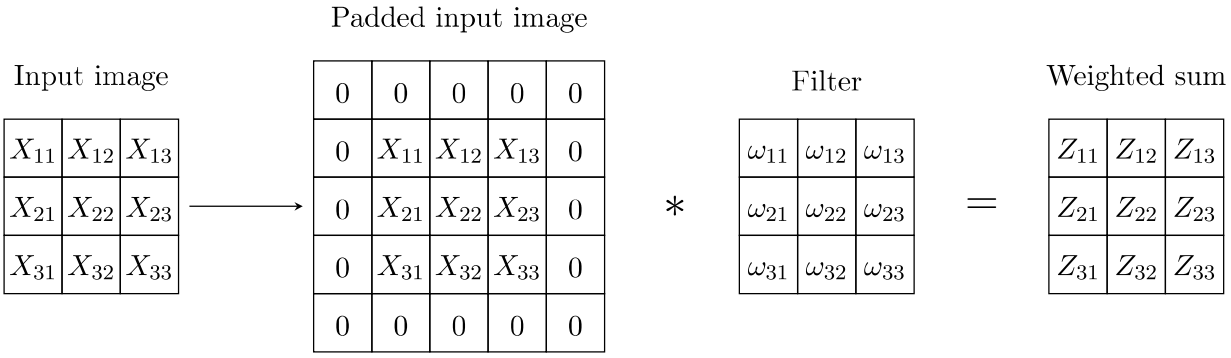


Figure 11.5.1: Forward Propagation of a 3×3 input image.

In convolution, we first rotate the filter with π

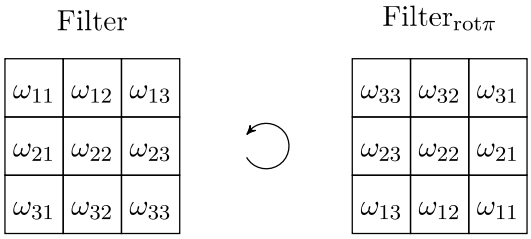


Figure 11.5.2: Rotating the filter π.

and

$\tilde{\boldsymbol{X}}^{(\ell,t)}_{1+1,1+1}(s^{(\ell)}_f)$

0	0	0
0	X_{11}	X_{12}
0	X_{21}	X_{22}

$\tilde{\boldsymbol{X}}^{(\ell,t)}_{1+1,1+2}(s^{(\ell)}_f)$

0	0	0
X_{11}	X_{12}	X_{13}
X_{21}	X_{22}	X_{23}

$\tilde{\boldsymbol{X}}^{(\ell,t)}_{1+1,1+3}(s^{(\ell)}_f)$

0	0	0
X_{12}	X_{13}	0
X_{22}	X_{23}	0

$\tilde{\boldsymbol{X}}^{(\ell,t)}_{1+2,1+1}(s^{(\ell)}_f)$

0	X_{11}	X_{12}
0	X_{21}	X_{22}
0	X_{31}	X_{32}

$\tilde{\boldsymbol{X}}^{(\ell,t)}_{1+2,1+2}(s^{(\ell)}_f)$

X_{11}	X_{12}	X_{13}
X_{21}	X_{22}	X_{23}
X_{31}	X_{32}	X_{33}

$\tilde{\boldsymbol{X}}^{(\ell,t)}_{1+2,1+3}(s^{(\ell)}_f)$

X_{12}	X_{13}	0
X_{22}	X_{23}	0
X_{32}	X_{33}	0

$\tilde{\boldsymbol{X}}^{(\ell,t)}_{1+3,1+1}(s^{(\ell)}_f)$

0	X_{21}	X_{22}
0	X_{31}	X_{32}
0	0	0

$\tilde{\boldsymbol{X}}^{(\ell,t)}_{1+3,1+2}(s^{(\ell)}_f)$

X_{21}	X_{22}	X_{23}
X_{31}	X_{32}	X_{33}
0	0	0

$\tilde{\boldsymbol{X}}^{(\ell,t)}_{1+3,1+3}(s^{(\ell)}_f)$

X_{22}	X_{23}	0
X_{32}	X_{33}	0
0	0	0

Figure 11.5.3: All the patches of $\tilde{\boldsymbol{X}}^{(\ell,t)}_{h,1+i',1+j'}(s^{(\ell)}_f)$, where $k=1$, $s^{(\ell)}_f=3$, and $i',j'=1,\cdots,s^{(\ell)}_{\mathcal{I}}=3$.

Therefore, in the forward propagation, with $k=1$, we have

$$\begin{aligned}
\boldsymbol{Z}^{(\ell,t)}[1,1] &:= Z_{11} = \tilde{\boldsymbol{X}}^{(\ell,t)}_{(1+1),(1+1)}(s^{(\ell)}_f) * \boldsymbol{K}^{(\ell,t-1)} + b^{(\ell,t-1)} = \tilde{\boldsymbol{X}}^{(\ell,t)}_{(1+1),(1+1)}(s^{(\ell)}_f) \star \boldsymbol{K}^{(\ell,t-1)}_{\text{rot}\pi} + b^{(\ell,t-1)} \\
&= 0\cdot\omega^{(\ell,t-1)}_{33} + 0\cdot\omega^{(\ell,t-1)}_{32} + 0\cdot\omega^{(\ell,t-1)}_{31} + 0\cdot\omega^{(\ell,t-1)}_{23} + X^{(\ell,t)}_{11}\cdot\omega^{(\ell,t-1)}_{22} \\
&\quad + X^{(\ell,t)}_{12}\cdot\omega^{(\ell,t-1)}_{21} + 0\cdot\omega^{(\ell,t-1)}_{13} + X^{(\ell,t)}_{21}\cdot\omega^{(\ell,t-1)}_{12} + X^{(\ell,t)}_{22}\cdot\omega^{(\ell,t-1)}_{11} + b^{(\ell,t-1)}\,; \\
\boldsymbol{Z}^{(\ell,t)}[1,2] &:= Z_{12} = \tilde{\boldsymbol{X}}^{(\ell,t)}_{(1+1),(1+1)+1}(s^{(\ell)}_f) * \boldsymbol{K}^{(\ell,t-1)} + b^{(\ell,t-1)} = \tilde{\boldsymbol{X}}^{(\ell,t)}_{(1+1),(1+1)+1}(s^{(\ell)}_f) \star \boldsymbol{K}^{(\ell,t-1)}_{\text{rot}\pi} + b^{(\ell,t-1)} \\
&= 0\cdot\omega^{(\ell,t-1)}_{33} + 0\cdot\omega^{(\ell,t-1)}_{32} + 0\cdot\omega^{(\ell,t-1)}_{31} + X^{(\ell,t)}_{11}\cdot\omega^{(\ell,t-1)}_{23} + X^{(\ell,t)}_{12}\cdot\omega^{(\ell,t-1)}_{22} \\
&\quad + X^{(\ell,t)}_{13}\cdot\omega^{(\ell,t-1)}_{21} + X^{(\ell,t)}_{21}\cdot\omega^{(\ell,t-1)}_{13} + X^{(\ell,t)}_{22}\cdot\omega^{(\ell,t-1)}_{12} + X^{(\ell,t)}_{23}\cdot\omega^{(\ell,t-1)}_{11} + b^{(\ell,t-1)}\,; \\
\boldsymbol{Z}^{(\ell,t)}[1,3] &:= Z_{21} = \tilde{\boldsymbol{X}}^{(\ell,t)}_{(1+1),(1+1)+2}(s^{(\ell)}_f) * \boldsymbol{K}^{(\ell,t-1)} + b^{(\ell,t-1)} = \tilde{\boldsymbol{X}}^{(\ell,t)}_{(1+1),(1+1)+2}(s^{(\ell)}_f) \star \boldsymbol{K}^{(\ell,t-1)}_{\text{rot}\pi} + b^{(\ell,t-1)} \\
&= 0\cdot\omega^{(\ell,t-1)}_{33} + 0\cdot\omega^{(\ell,t-1)}_{32} + 0\cdot\omega^{(\ell,t-1)}_{31} + X^{(\ell,t)}_{12}\cdot\omega^{(\ell,t-1)}_{23} + X^{(\ell,t)}_{13}\cdot\omega^{(\ell,t-1)}_{22} \\
&\quad + 0\cdot\omega^{(\ell,t-1)}_{21} + X^{(\ell,t)}_{22}\cdot\omega^{(\ell,t-1)}_{13} + X^{(\ell,t)}_{23}\cdot\omega^{(\ell,t-1)}_{12} + 0\cdot\omega^{(\ell,t-1)}_{11} + b^{(\ell,t-1)}\,.
\end{aligned}$$

The calculations for $\boldsymbol{Z}^{(\ell,t)}[2,1], \boldsymbol{Z}^{(\ell,t)}[2,2], \boldsymbol{Z}^{(\ell,t)}[2,3], \boldsymbol{Z}^{(\ell,t)}[3,1], \boldsymbol{Z}^{(\ell,t)}[3,2], \boldsymbol{Z}^{(\ell,t)}[3,3]$ are left to readers.

(II) The backward propagation particularly for the convolutional layer is depicted as:

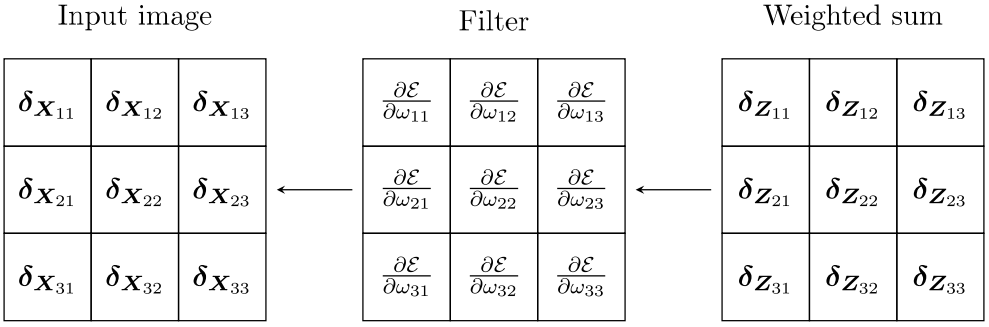


Figure 11.5.4: Backward pass of the error rate $\boldsymbol{\delta}^{(\ell,t)}_{\boldsymbol{Z}_1}$ through the filter.

Therefore, in the backward propagation, using (11.5.11) with $s_O^{(\ell)} = 3$ and $s_{sr}^{(\ell)} = 1$ such that $i = i'$ and $j = j'$, we have

$$\nabla_{\boldsymbol{K}^{(\ell,t-1)}}\mathcal{E}^{(t)} = \sum_{i=1}^{3}\sum_{j=1}^{3} \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[i,j] \cdot \Big[\tilde{\boldsymbol{X}}_{1+i,1+j}^{(\ell,t)}(s_f^{(\ell)})\Big]_{\text{rot}\pi} .$$

$\Big[\tilde{\boldsymbol{X}}_{1+1,1+1}^{(\ell,t)}(s_f^{(\ell)})\Big]_{\text{rot}\pi}$

X_{22}	X_{21}	0
X_{12}	X_{11}	0
0	0	0

$\Big[\tilde{\boldsymbol{X}}_{1+1,1+2}^{(\ell,t)}(s_f^{(\ell)})\Big]_{\text{rot}\pi}$

X_{23}	X_{22}	X_{21}
X_{13}	X_{12}	X_{11}
0	0	0

$\Big[\tilde{\boldsymbol{X}}_{1+1,1+3}^{(\ell,t)}(s_f^{(\ell)})\Big]_{\text{rot}\pi}$

0	X_{23}	X_{22}
0	X_{13}	X_{12}
0	0	0

$\Big[\tilde{\boldsymbol{X}}_{1+2,1+1}^{(\ell,t)}(s_f^{(\ell)})\Big]_{\text{rot}\pi}$

X_{32}	X_{31}	0
X_{22}	X_{21}	0
X_{12}	X_{11}	0

$\Big[\tilde{\boldsymbol{X}}_{1+2,1+2}^{(\ell,t)}(s_f^{(\ell)})\Big]_{\text{rot}\pi}$

X_{33}	X_{32}	X_{31}
X_{23}	X_{22}	X_{21}
X_{13}	X_{12}	X_{11}

$\Big[\tilde{\boldsymbol{X}}_{1+2,1+3}^{(\ell,t)}(s_f^{(\ell)})\Big]_{\text{rot}\pi}$

0	X_{33}	X_{32}
0	X_{23}	X_{22}
0	X_{13}	X_{12}

$\Big[\tilde{\boldsymbol{X}}_{1+3,1+1}^{(\ell,t)}(s_f^{(\ell)})\Big]_{\text{rot}\pi}$

0	0	0
X_{32}	X_{31}	0
X_{22}	X_{21}	0

$\Big[\tilde{\boldsymbol{X}}_{1+3,1+2}^{(\ell,t)}(s_f^{(\ell)})\Big]_{\text{rot}\pi}$

0	0	0
X_{33}	X_{32}	X_{31}
X_{23}	X_{22}	X_{21}

$\Big[\tilde{\boldsymbol{X}}_{1+3,1+3}^{(\ell,t)}(s_f^{(\ell)})\Big]_{\text{rot}\pi}$

0	0	0
0	X_{33}	X_{32}
0	X_{23}	X_{22}

Figure 11.5.5: All the patches of $\Big[\tilde{\boldsymbol{X}}_{h,1+i',1+j'}^{(\ell,t)}(s_f^{(\ell)})\Big]_{\text{rot}\pi}$, where $k = 1$, $s_f^{(\ell)} = 3$, and $i', j' = 1, \cdots, s_{\mathcal{I}}^{(\ell)} = 3$.

In particular,

$$\begin{aligned}
\frac{\partial\mathcal{E}^{(t)}}{\partial\boldsymbol{K}^{(\ell,t-1)}[1,1]} &:= \frac{\partial\mathcal{E}^{(t)}}{\partial\omega_{11}^{(\ell,t-1)}} = \sum_{i=1}^{3}\sum_{j=1}^{3} \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[i,j] \cdot \Big[\tilde{\boldsymbol{X}}_{1+i,1+j}^{(\ell,t)}(s_f^{(\ell)})\Big]_{\text{rot}\pi}[1,1] \\
&= \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[1,1] \cdot X_{22}^{(\ell,t)} + \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[1,2] \cdot X_{23}^{(\ell,t)} + \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[1,3] \cdot 0 \\
&\quad + \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[2,1] \cdot X_{32}^{(\ell,t)} + \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[2,2] \cdot X_{33}^{(\ell,t)} + \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[2,3] \cdot 0 \\
&\quad + \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[3,1] \cdot 0 + \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[3,2] \cdot 0 + \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[3,3] \cdot 0\,; \\
\frac{\partial\mathcal{E}^{(t)}}{\partial\boldsymbol{K}^{(\ell,t-1)}[1,2]} &:= \frac{\partial\mathcal{E}^{(t)}}{\partial\omega_{12}^{(\ell,t-1)}} = \sum_{i=1}^{3}\sum_{j=1}^{3} \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[i,j] \cdot \Big[\tilde{\boldsymbol{X}}_{1+i,1+j}^{(\ell,t)}(s_f^{(\ell)})\Big]_{\text{rot}\pi}[1,2] \\
&= \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[1,1] \cdot X_{21}^{(\ell,t)} + \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[1,2] \cdot X_{22}^{(\ell,t)} + \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[1,3] \cdot X_{23}^{(\ell,t)} \\
&\quad + \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[2,1] \cdot X_{31}^{(\ell,t)} + \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[2,2] \cdot X_{32}^{(\ell,t)} + \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[2,3] \cdot X_{33}^{(\ell,t)} \\
&\quad + \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[3,1] \cdot 0 + \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[3,2] \cdot 0 + \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[3,3] \cdot 0\,; \\
\frac{\partial\mathcal{E}^{(t)}}{\partial\boldsymbol{K}^{(\ell,t-1)}[1,3]} &:= \frac{\partial\mathcal{E}^{(t)}}{\partial\omega_{13}^{(\ell,t-1)}} = \sum_{i=1}^{3}\sum_{j=1}^{3} \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[i,j] \cdot \Big[\tilde{\boldsymbol{X}}_{1+i,1+j}^{(\ell,t)}(s_f^{(\ell)})\Big]_{\text{rot}\pi}[1,3] \\
&= \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[1,1] \cdot 0 + \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[1,2] \cdot X_{21}^{(\ell,t)} + \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[1,3] \cdot X_{22}^{(\ell,t)} \\
&\quad + \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[2,1] \cdot 0 + \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[2,2] \cdot X_{31}^{(\ell,t)} + \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[2,3] \cdot X_{32}^{(\ell,t)} \\
&\quad + \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[3,1] \cdot 0 + \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[3,2] \cdot 0 + \boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}[3,3] \cdot 0\,.
\end{aligned}$$

The remaining calculations for error rate with respect to the filter are left for readers. Therefore, the error rate with respect to the Filter $\boldsymbol{K}_r^{(\ell,t-1)}$, for $r = 1, \cdots, n_f^{(\ell)}$, can be equivalently computed as in (11.5.10)

namely to perform: (1) perform the cross-correlation between $\tilde{\boldsymbol{X}}^{(\ell,t)}$ and $\boldsymbol{\delta}_{\bar{\boldsymbol{Z}}_r}^{(\ell,t)}$; and then (2) to rotate the result with π, *i.e.*

$$\nabla_{\boldsymbol{K}_r^{(\ell,t-1)}}\mathcal{E}^{(t)} = [\tilde{\boldsymbol{X}}^{(\ell,t)} \star \boldsymbol{\delta}_{\bar{\boldsymbol{Z}}_r}^{(\ell,t)}]_{\text{rot}\pi}\,, \qquad \text{for} \quad r = 1, \cdots, n_f^{(\ell)}\,. \tag{11.5.15}$$

$\tilde{\boldsymbol{X}}^{(\ell,t)}$

0	0	0	0	0
0	X_{11}	X_{12}	X_{13}	0
0	X_{21}	X_{22}	X_{23}	0
0	X_{31}	X_{32}	X_{33}	0
0	0	0	0	0

$\star$

$\boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}$

$\boldsymbol{\delta}_{Z_{11}}$	$\boldsymbol{\delta}_{Z_{12}}$	$\boldsymbol{\delta}_{Z_{13}}$
$\boldsymbol{\delta}_{Z_{21}}$	$\boldsymbol{\delta}_{Z_{22}}$	$\boldsymbol{\delta}_{Z_{23}}$
$\boldsymbol{\delta}_{Z_{31}}$	$\boldsymbol{\delta}_{Z_{32}}$	$\boldsymbol{\delta}_{Z_{33}}$

Figure 11.5.6: Convolution of the padded image $\tilde{\boldsymbol{X}}^{(\ell,t)}$ and the error rate $\boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}$.

For bias,

$$\frac{\partial\mathcal{E}^{(t)}}{\partial b_r^{(\ell,t-1)}} = \sum_{i=1}^{3}\sum_{j=1}^{3}\boldsymbol{\delta}_{\boldsymbol{Z}_r}^{(\ell,t)}[i,j]\,, \qquad \text{for} \quad r = 1, \cdots, n_f^{(\ell)}\,.$$

Finally, we backpropagate $\boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}$ to the error rate with respect to the input image $\boldsymbol{\delta}_{\boldsymbol{X}}^{(\ell,t)}$ with one filter $n_f^{(\ell)} = 1$, as shown in (11.5.12):

$$\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\boldsymbol{X}}_{1,k+i',k+j'}(s_f^{(\ell)})} = 1 \cdot \boldsymbol{\delta}_{\tilde{\bar{\boldsymbol{Z}}}_1}^{(\ell,t)}[k+i', k+j'] \cdot [\boldsymbol{K}_{1,1}^{(\ell,t-1)}]_{\text{rot}\pi}\,, \tag{11.5.16}$$

for $i, j = 1, \cdots, s_O^{(\ell)}$ with $i' = 1+(i-1)s_{sr}^{(\ell)} = 1+(i-1)\cdot 1 = i$ and $j' = 1+(j-1)s_{sr}^{(\ell)} = 1+(j-1)\cdot 1 = j$ defined in (11.5.8). After computing $\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\boldsymbol{X}}_{1,i',j'}}(s_f^{(\ell)})$, we sum over these error rates to compute $\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\boldsymbol{X}}_1}[k+a', k+b']$ for $k+a', k+b' = 1, \cdots, s_{\mathcal{I}}^{(\ell)} + 2s_p^{(\ell)} = 3+2 = 5$. Since $s_{sr}^{(\ell)} = 1$, the error rates of the loss $\mathcal{E}^{(t)}$ with respect to the padded extended weighted sum $\tilde{\bar{\boldsymbol{Z}}}_r^{(\ell,t)}$ are $\boldsymbol{\delta}_{\tilde{\bar{\boldsymbol{Z}}}_r}^{(\ell,t)}[1,1] = 0$, $\boldsymbol{\delta}_{\tilde{\bar{\boldsymbol{Z}}}_r}^{(\ell,t)}[2,2] = \boldsymbol{\delta}_{\boldsymbol{Z}_r}^{(\ell,t)}[1,1]$, $\boldsymbol{\delta}_{\tilde{\bar{\boldsymbol{Z}}}_r}^{(\ell,t)}[2,3] = \boldsymbol{\delta}_{\boldsymbol{Z}_r}^{(\ell,t)}[1,2]$, and so on, leading to the grid below:

$\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\tilde{\boldsymbol{Z}}}_1}$

$\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_{11}}$	$\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_{12}}$	$\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_{13}}$	$\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_{14}}$	$\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_{15}}$
$\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_{21}}$	$\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_{22}}$	$\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_{23}}$	$\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_{24}}$	$\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_{25}}$
$\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_{31}}$	$\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_{32}}$	$\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_{33}}$	$\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_{34}}$	$\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_{35}}$
$\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_{41}}$	$\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_{42}}$	$\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_{43}}$	$\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_{44}}$	$\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_{45}}$
$\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_{51}}$	$\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_{52}}$	$\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_{53}}$	$\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_{54}}$	$\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_{55}}$

$=$

$\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\tilde{\boldsymbol{Z}}}_1}$

0	0	0	0	0
0	$\boldsymbol{\delta}_{\boldsymbol{Z}_{11}}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{12}}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{13}}$	0
0	$\boldsymbol{\delta}_{\boldsymbol{Z}_{21}}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{22}}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{23}}$	0
0	$\boldsymbol{\delta}_{\boldsymbol{Z}_{31}}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{32}}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{33}}$	0
0	0	0	0	0

Figure 11.5.7: Construction of $\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\tilde{\boldsymbol{Z}}}_1}$ from $\boldsymbol{\delta}^{(\ell,t)}_{\boldsymbol{Z}_1}$.

Therefore, using (11.5.16), the patches of $\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\boldsymbol{X}}_{1,i',j'}(s_f^{(\ell)})}$ are depicted as follows:

$\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\boldsymbol{X}}_{1,2,2}(s_f^{(\ell)})}$

$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,1}}\omega_{33}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,1}}\omega_{32}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,1}}\omega_{31}$
$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,1}}\omega_{23}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,1}}\omega_{22}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,1}}\omega_{21}$
$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,1}}\omega_{13}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,1}}\omega_{12}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,1}}\omega_{11}$

$\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\boldsymbol{X}}_{1,2,3}(s_f^{(\ell)})}$

$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,2}}\omega_{33}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,2}}\omega_{32}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,2}}\omega_{31}$
$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,2}}\omega_{23}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,2}}\omega_{22}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,2}}\omega_{21}$
$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,2}}\omega_{13}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,2}}\omega_{12}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,2}}\omega_{11}$

$\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\boldsymbol{X}}_{1,2,4}(s_f^{(\ell)})}$

$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,3}}\omega_{33}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,3}}\omega_{32}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,3}}\omega_{31}$
$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,3}}\omega_{23}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,3}}\omega_{22}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,3}}\omega_{21}$
$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,3}}\omega_{13}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,3}}\omega_{12}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{1,3}}\omega_{11}$

$\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\boldsymbol{X}}_{1,3,2}(s_f^{(\ell)})}$

$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,1}}\omega_{33}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,1}}\omega_{32}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,1}}\omega_{31}$
$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,1}}\omega_{23}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,1}}\omega_{22}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,1}}\omega_{21}$
$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,1}}\omega_{13}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,1}}\omega_{12}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,1}}\omega_{11}$

$\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\boldsymbol{X}}_{1,3,3}(s_f^{(\ell)})}$

$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,2}}\omega_{33}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,2}}\omega_{32}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,2}}\omega_{31}$
$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,2}}\omega_{23}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,2}}\omega_{22}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,2}}\omega_{21}$
$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,2}}\omega_{13}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,2}}\omega_{12}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,2}}\omega_{11}$

$\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\boldsymbol{X}}_{1,3,4}(s_f^{(\ell)})}$

$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,3}}\omega_{33}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,3}}\omega_{32}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,3}}\omega_{31}$
$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,3}}\omega_{23}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,3}}\omega_{22}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,3}}\omega_{21}$
$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,3}}\omega_{13}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,3}}\omega_{12}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{2,3}}\omega_{11}$

$\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\boldsymbol{X}}_{1,4,2}(s_f^{(\ell)})}$

$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,1}}\omega_{33}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,1}}\omega_{32}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,1}}\omega_{31}$
$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,1}}\omega_{23}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,1}}\omega_{22}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,1}}\omega_{21}$
$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,1}}\omega_{13}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,1}}\omega_{12}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,1}}\omega_{11}$

$\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\boldsymbol{X}}_{1,4,3}(s_f^{(\ell)})}$

$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,2}}\omega_{33}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,2}}\omega_{32}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,2}}\omega_{31}$
$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,2}}\omega_{23}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,2}}\omega_{22}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,2}}\omega_{21}$
$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,2}}\omega_{13}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,2}}\omega_{12}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,2}}\omega_{11}$

$\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\boldsymbol{X}}_{1,4,4}(s_f^{(\ell)})}$

$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,3}}\omega_{33}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,3}}\omega_{32}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,3}}\omega_{31}$
$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,3}}\omega_{23}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,3}}\omega_{22}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,3}}\omega_{21}$
$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,3}}\omega_{13}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,3}}\omega_{12}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{3,3}}\omega_{11}$

Figure 11.5.8: All the patches of $\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\boldsymbol{X}}_{h,i',j'}(s_f^{(\ell)})}$.

By adding up with respect to the indices position, we compute the error rates of the padded input image $\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\boldsymbol{X}}_1}[k+a', k+b']$, for $k+a', k+b' = 1, \cdots, 5$, using (11.5.13) and also noting that the boundary values are vanished, by then we shall obtain the following table:

$\delta_{\boldsymbol{Z}_{1,1}}\omega_{33}$	$\delta_{\boldsymbol{Z}_{1,1}}\omega_{32} + \delta_{\boldsymbol{Z}_{1,2}}\omega_{33}$	$\delta_{\boldsymbol{Z}_{1,1}}\omega_{31} + \delta_{\boldsymbol{Z}_{1,2}}\omega_{32} + \delta_{\boldsymbol{Z}_{1,3}}\omega_{33}$	$\delta_{\boldsymbol{Z}_{1,2}}\omega_{31} + \delta_{\boldsymbol{Z}_{1,3}}\omega_{32}$	$\delta_{\boldsymbol{Z}_{1,3}}\omega_{31}$
$\delta_{\boldsymbol{Z}_{1,1}}\omega_{23} + \delta_{\boldsymbol{Z}_{2,1}}\omega_{33}$	$\delta_{\boldsymbol{Z}_{1,1}}\omega_{22} + \delta_{\boldsymbol{Z}_{1,2}}\omega_{23} + \delta_{\boldsymbol{Z}_{2,1}}\omega_{32} + \delta_{\boldsymbol{Z}_{2,2}}\omega_{33}$	$\delta_{\boldsymbol{Z}_{1,1}}\omega_{21} + \delta_{\boldsymbol{Z}_{1,2}}\omega_{22} + \delta_{\boldsymbol{Z}_{1,3}}\omega_{23} + \delta_{\boldsymbol{Z}_{2,1}}\omega_{31} + \delta_{\boldsymbol{Z}_{2,2}}\omega_{32} + \delta_{\boldsymbol{Z}_{2,3}}\omega_{33}$	$\delta_{\boldsymbol{Z}_{1,2}}\omega_{21} + \delta_{\boldsymbol{Z}_{1,3}}\omega_{22} + \delta_{\boldsymbol{Z}_{2,2}}\omega_{31} + \delta_{\boldsymbol{Z}_{2,3}}\omega_{32}$	$\delta_{\boldsymbol{Z}_{1,3}}\omega_{21} + \delta_{\boldsymbol{Z}_{2,3}}\omega_{31}$
$\delta_{\boldsymbol{Z}_{1,1}}\omega_{13} + \delta_{\boldsymbol{Z}_{2,1}}\omega_{23} + \delta_{\boldsymbol{Z}_{3,1}}\omega_{33}$	$\delta_{\boldsymbol{Z}_{1,1}}\omega_{12} + \delta_{\boldsymbol{Z}_{1,2}}\omega_{13} + \delta_{\boldsymbol{Z}_{2,1}}\omega_{22} + \delta_{\boldsymbol{Z}_{2,2}}\omega_{23} + \delta_{\boldsymbol{Z}_{3,1}}\omega_{32} + \delta_{\boldsymbol{Z}_{3,2}}\omega_{33}$	$\delta_{\boldsymbol{Z}_{1,1}}\omega_{11} + \delta_{\boldsymbol{Z}_{1,2}}\omega_{12} + \delta_{\boldsymbol{Z}_{1,3}}\omega_{13} + \delta_{\boldsymbol{Z}_{2,1}}\omega_{21} + \delta_{\boldsymbol{Z}_{2,2}}\omega_{22} + \delta_{\boldsymbol{Z}_{2,3}}\omega_{23} + \delta_{\boldsymbol{Z}_{3,1}}\omega_{31} + \delta_{\boldsymbol{Z}_{3,2}}\omega_{32} + \delta_{\boldsymbol{Z}_{3,3}}\omega_{33}$	$\delta_{\boldsymbol{Z}_{1,2}}\omega_{11} + \delta_{\boldsymbol{Z}_{1,3}}\omega_{12} + \delta_{\boldsymbol{Z}_{2,2}}\omega_{21} + \delta_{\boldsymbol{Z}_{2,3}}\omega_{22} + \delta_{\boldsymbol{Z}_{3,2}}\omega_{31} + \delta_{\boldsymbol{Z}_{3,3}}\omega_{32}$	$\delta_{\boldsymbol{Z}_{1,3}}\omega_{11} + \delta_{\boldsymbol{Z}_{2,3}}\omega_{21} + \delta_{\boldsymbol{Z}_{3,3}}\omega_{31}$
$\delta_{\boldsymbol{Z}_{2,1}}\omega_{13} + \delta_{\boldsymbol{Z}_{3,1}}\omega_{23}$	$\delta_{\boldsymbol{Z}_{2,1}}\omega_{12} + \delta_{\boldsymbol{Z}_{2,2}}\omega_{13} + \delta_{\boldsymbol{Z}_{3,1}}\omega_{22} + \delta_{\boldsymbol{Z}_{3,2}}\omega_{23}$	$\delta_{\boldsymbol{Z}_{2,1}}\omega_{11} + \delta_{\boldsymbol{Z}_{2,2}}\omega_{12} + \delta_{\boldsymbol{Z}_{2,3}}\omega_{13} + \delta_{\boldsymbol{Z}_{3,1}}\omega_{21} + \delta_{\boldsymbol{Z}_{3,2}}\omega_{22} + \delta_{\boldsymbol{Z}_{3,3}}\omega_{23}$	$\delta_{\boldsymbol{Z}_{2,2}}\omega_{11} + \delta_{\boldsymbol{Z}_{2,3}}\omega_{12} + \delta_{\boldsymbol{Z}_{3,2}}\omega_{21} + \delta_{\boldsymbol{Z}_{3,3}}\omega_{22}$	$\delta_{\boldsymbol{Z}_{2,3}}\omega_{11} + \delta_{\boldsymbol{Z}_{3,3}}\omega_{21}$
$\delta_{\boldsymbol{Z}_{3,1}}\omega_{13}$	$\delta_{\boldsymbol{Z}_{3,1}}\omega_{12} + \delta_{\boldsymbol{Z}_{3,2}}\omega_{13}$	$\delta_{\boldsymbol{Z}_{3,1}}\omega_{11} + \delta_{\boldsymbol{Z}_{3,2}}\omega_{12} + \delta_{\boldsymbol{Z}_{3,3}}\omega_{13}$	$\delta_{\boldsymbol{Z}_{3,2}}\omega_{11} + \delta_{\boldsymbol{Z}_{3,3}}\omega_{12}$	$\delta_{\boldsymbol{Z}_{3,3}}\omega_{11}$

Figure 11.5.9: The overall error rate $\boldsymbol{\delta}_{\tilde{\boldsymbol{X}}_h}^{(\ell,t)}$.

However, at the begining of the convolution, we pad the input image with a value of 0 with a padding size of $s_p^{(\ell)}$, therefore we shall remove that boundary values for $\boldsymbol{\delta}_{\boldsymbol{X}_h}^{(\ell,t)}$. In other words, we first (1) transform $\boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}$ into $\boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_r}^{(\ell,t)}$; and then (2) perform a cross-correlation with Filter $\boldsymbol{K}^{(\ell,t-1)}$ such that, for $h = 1, \cdots, n_v^{(\ell)} = n_f^{(\ell-1)}$, $a', b' = 1, \cdots, s_{\mathcal{I}}^{(\ell)}$, we have

$$\boldsymbol{\delta}_{\boldsymbol{X}_h}^{(\ell,t)}[a', b'] = \sum_{r=1}^{n_f^{(\ell)}} \boldsymbol{\delta}_{\tilde{\tilde{\boldsymbol{Z}}}_r, k+a', k+b'}^{(\ell,t)} (s_f^{(\ell)}) \star \boldsymbol{K}_{r,h}^{(\ell,t-1)}, \tag{11.5.17}$$

which is, by neglecting the bias term, in parallel with the computation of the weighted sum $\boldsymbol{Z}_r^{(\ell,t)}[i, j]$ in (11.5.1) with the unpadded image $\boldsymbol{X}^{(\ell,t)}$.

$\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\tilde{\boldsymbol{Z}}}_1}$

0	0	0	0	0
0	$\boldsymbol{\delta}_{\boldsymbol{Z}_{11}}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{12}}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{13}}$	0
0	$\boldsymbol{\delta}_{\boldsymbol{Z}_{21}}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{22}}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{23}}$	0
0	$\boldsymbol{\delta}_{\boldsymbol{Z}_{31}}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{32}}$	$\boldsymbol{\delta}_{\boldsymbol{Z}_{33}}$	0
0	0	0	0	0

$\star$

$\boldsymbol{K}^{(\ell,t-1)}_{1,1}$

ω_{11}	ω_{12}	ω_{13}
ω_{21}	ω_{22}	ω_{23}
ω_{31}	ω_{32}	ω_{33}

Figure 11.5.10: Cross-correlation of the extended error rate $\boldsymbol{\delta}^{(\ell,t)}_{\tilde{\tilde{\boldsymbol{Z}}}_1}$ and Filter $\boldsymbol{K}^{(\ell,t-1)}_{1,1}$.

11.6 Coding basics in Convolutional Neural Network

In this section, we first introduce the Programme for the flattening layers:

```
import numpy as np

class Flattening():
    def Forward(self, X):
        self.shape = np.shape(X)
        return X.reshape(-1, np.shape(X)[-1])        # g_F(X^(ℓ,t))

    def Backward(self, delta_F):
        return delta_F.reshape(self.shape)           # g_F^{-1}(X^(ℓ,t))

    def Forward_Test(self, X):
        return self.Forward(X)
```

Programme 11.6.1: Flattening layer saved in `CNN.py` in Python.

Here the `Flattening()` class object has three subsidiary functions, the `Forward()` function transforms the 4-dimensional tensor of size $s_{\mathcal{I}}^{(\ell)} \times s_{\mathcal{I}}^{(\ell)} \times n_v^{(\ell)} \times b_s^{(t)}$, with $b_s^{(t)}$ being the batch size of the t iteration, to a 2 dimensional matrix $(s_{\mathcal{I}}^{(\ell)} \cdot s_{\mathcal{I}}^{(\ell)} \cdot n_v^{(\ell)}) \times b_s^{(t)}$ as the input feature matrix for the MLP; the `Backward()` function is the reverse process of the `Forward()` function; the `Forward_Test()` function calls the `Forward()` function when the testing dataset is fed.

```
import numpy as np

class GlobalAveragePooling():
    def Forward(self, X):
        self.shape = np.shape(X)
        return np.average(X, axis=(0, 1))            # g_G(X^(ℓ,t))

    def Backward(self, delta_F):
        delta_X = np.repeat(delta_F, self.shape[0]*self.shape[1])
        return delta_X.reshape(self.shape)           # g_G^{-1}(X^(ℓ,t))

    def Forward_Test(self, X):
        return self.Forward(X)
```

Programme 11.6.2: Global average pooling layer saved in `CNN.py` in Python.

Here the `GlobalAveragePooling()` performs the global average pooling and it is similar to the above `Flattening()` class object in Programme 11.6.1, except that it transforms the 4-dimensional tensor of size $s_{\mathcal{I}}^{(\ell)} \times s_{\mathcal{I}}^{(\ell)} \times n_v^{(\ell)} \times b_s^{(t)}$ to a 2 dimensional matrix $n_v^{(\ell)} \times b_s^{(t)}$ through averaging over the dimensions $s_{\mathcal{I}}^{(\ell)}$, $s_{\mathcal{I}}^{(\ell)}$, and $n_v^{(\ell)}$, as the input feature matrix for the MLP.

Next, we have Programme 11.6.3 for the convolutional layer.

```
import numpy as np
import pickle
from scipy.signal import correlate, convolve
```

```
copy_class = lambda class_obj: pickle.loads(pickle.dumps(class_obj))

class Convolutional:
    # X: s_i x s_i x n_v x M
    # K: s_f x s_f x n_v x n_f
    # Z: s_o x s_o x n_f x M
    def __init__(self, n_f, s_f):
        self.n_f = n_f       # number of filters
        self.s_f = s_f       # size of filter
        self.b, self.K = None, None
        self.shape_PD = ((s_f - 1) // 2, (s_f - 1) // 2)
        assert (s_f % 2) != 0
        assert self.n_f > 0 and isinstance(self.n_f, int)
        assert self.s_f > 0 and isinstance(self.s_f, int)

    def Param_init(self, n_input):                    # He Initialization
        self.K = np.random.randn(*n_input) * np.sqrt(2 / np.sum(n_input))
        self.optimizer_K = self.optimizer

        self.b = np.zeros(self.n_f).reshape(-1, 1)
        self.optimizer_b = copy_class(self.optimizer)

    def Padding(self, X):                             # X̃^(ℓ,t)
        return np.pad(X, (self.shape_PD, self.shape_PD, (0, 0), (0, 0)),
                      "constant")

    def Forward(self, X):
        self.s_x = np.shape(X)
        self.n_v = self.s_x[2]
        self.bs = self.s_x[-1]
        self.tilde_X = self.Padding(X)
        if self.b is None:
            shape_K = (self.s_f, self.s_f, self.n_v, self.n_f)
            self.Param_init(shape_K)

        Z = []
        for r in range(self.n_f):
            expand_K = np.expand_dims(self.K[:,:,:,r], axis=-1)
            # X̃^(ℓ,t) * K^(ℓ,t-1)
            convolve_X_K = convolve(self.tilde_X, expand_K, mode="valid")
            Z.append(np.squeeze(convolve_X_K + self.b[r], axis=2))
        return np.moveaxis(Z, 0, 2)

    def Backward(self, delta_Z):
        delta_X = np.zeros(self.s_x)
        tilde_delta_Z = self.Padding(delta_Z)
```

```
        for h in range(self.n_v):
            expand_K = np.expand_dims(self.K[:,:,h,:], axis=-1)
            # sum_{r=1}^{n_f^(l)} delta_{tilde_Z_r}^{(l,t)} * K_r^{(l,t-1)}
            correlate_dZ_K = correlate(tilde_delta_Z, expand_K, mode="valid")
            delta_X[:,:,h,:] = np.squeeze(correlate_dZ_K, axis=2)

        dg_K = np.zeros(np.shape(self.K))
        for r in range(self.n_f):
            expand_dZ = np.expand_dims(delta_Z[:,:,r,:], axis=2)
            # tilde_X^{(l,t)} * delta_{Z_r}^{(l,t)}
            correlate_X_dZ = correlate(self.tilde_X, expand_dZ, mode="valid")
            dg_K[:,:,:,r] = np.squeeze(correlate_X_dZ, axis=-1)
            dg_K = np.rot90(dg_K, 2, axes=(1,0)) # [tilde_X^{(l,t)} * delta_{Z_r}^{(l,t)}]_{rot pi}

        dg_b = np.sum(delta_Z, axis=(0,1,3)).reshape(-1,1)
        self.K += self.optimizer_K.Delta(dg_K)
        self.b += self.optimizer_b.Delta(dg_b)
        return delta_X

    def Forward_Test(self, X):
        return self.Forward(X)
```

Programme 11.6.3: Convolutional layer saved in `CNN.py` in Python.

Here the `Convolutional()` class object accepts only two input parameters, namely the number of filter used `n_f` and the size of the filter `s_f`; see Section 11.5 for more details. For simplicity, the stride size $s_{sr}^{(\ell)}$ is fixed at 1 and padding is automatically applied to the input image through the `Padding()` function. The padding process uses the `np.pad()` function from the `numpy` library; see Programme 11.2.1 for more details. Similar to Programme 10.1.2 and Programme 10.4.1, `copy_class()` function is used to copy an individual `self.optimizer()` class object.

The `Forward()` function performs the forward propagation of CNN. It computes the weighted sum $\boldsymbol{Z}^{(\ell,t)}$ through convolution for each of the filter. The `np.moveaxis()` function is applied to the convolutional output `Z` of size $b_s^{(t)} \times s_O^{(\ell)} \times s_O^{(\ell)} \times n_f^{(\ell)}$, with $b_s^{(t)}$ being the batch size of the current iteration t, such that the weighted sum is of size $s_O^{(\ell)} \times s_O^{(\ell)} \times n_f^{(\ell)} \times b_s^{(t)}$.

The `Backward()` function performs the backward propagation of CNN. It first computes the sensitivity with respect to the input image $\boldsymbol{\delta}_{\boldsymbol{X}_h}$ for each of the volumn depth through the cross-correlation `correlate()` function from `scipy.signal`, where the input parameters are the same as the `covolution()` function. It next computes the sensitivity with respect to the kernel through correlation between the padded image $\tilde{\boldsymbol{X}}^{(\ell,t)}$ with the sensitivity with respect to the weighted sum $\boldsymbol{\delta}_{\boldsymbol{Z}}^{(\ell,t)}$. Lastly, it is saved in `CNN.py`.

```
import numpy as np

class Pooling:
```

```
    # X: s_i x s_i x n_f x M
    def __init__(self, p_func, s_p):
        self.p_func = p_func.lower()
        self.s_p = s_p
        self.Shape_X = (self.s_p, self.s_p, 1, 1)
        assert self.p_func in ["maxpooling", "averagepooling"]
        assert self.s_p > 0 and isinstance(self.s_p, int)

    def Forward(self, X):
        self.X = X
        Shape_P = (np.shape(X)[0]//self.s_p, self.s_p,
                   np.shape(X)[1]//self.s_p, self.s_p, np.shape(X)[2], -1)
        if self.p_func == "maxpooling":
            self.A_P = X.reshape(Shape_P).max(axis=(3, 1))
        elif self.p_func == "averagepooling":
            self.A_P = X.reshape(Shape_P).mean(axis=(3, 1))
        return self.A_P

    def Backward(self, delta_P):
        delta_X = np.kron(delta_P, np.ones(self.Shape_X))
        if self.p_func == "maxpooling":
            delta_X *= (self.X == np.kron(self.A_P, np.ones(self.Shape_X)))
        elif self.p_func == "averagepooling":
            delta_X /= (self.s_p ** 2)
        return delta_X

    def Forward_Test(self, X):
        return self.Forward(X)
```

Programme 11.6.4: Pooling layer saved in `CNN.py` in Python.

Here the `Pooling()` class object accepts two input parameters, the method `p_func` used for padding (max pooling or average pooling), and the size of the padding `s_p`. Note that the stride size is set to be the padding size.

In `Forward()` propagation, it first transforms the input image into a six dimensional tensor. It basically divides the input image into several pooling grids of size $s_p^{(\ell)} \times s_p^{(\ell)}$, and then finding the maximum value if max pooling is adopted or the average value if average pooling is adopted.

In `Backward()` propagation, the `np.kron()` function from the `numpy` library computes the Kronecker product of two input arrays. In max pooling, it first finds a boolean matrix indicating the position where the maxium value is obtained with a value of 1 and 0 otherwise; in average pooling, it computes the senisitivity with respect to the input image via dividing by the square of the size of pooling.

In `Forward_Test()` function, it performs the forward propagation for the testing dataset, which is the same procedure as in `Forward()` function. Lastly, it is saved in the `CNN.py`.

```
import numpy as np
from tqdm import trange
from Dense import Dense
from Intermediate_Layers import Activation
import pickle
```

```
copy_class = lambda class_obj: pickle.loads(pickle.dumps(class_obj))

class Sequential:
    def __init__(self):
        self.network = []

    def add(self, layers):
        self.network.append(layers)

    def Create_mini_batch(self, X, y=None, bs=32):
        if len(np.shape(X)) in [1,3]:
            X = np.expand_dims(X, axis=0)
        X_batch, y_batch = [], []
        for i in range(0, len(X), bs):
            X_batch.append(X[i:i+bs])
            if y is not None:
                y_batch.append(y[i:i+bs])
        return X_batch, y_batch

    def Binary_Cross_Entropy(self, y_train, a):
        hat_y = a==np.max(a, axis=0)
        acc = np.sum(y_train * hat_y) / self.bs
        one_minus_a = np.clip(1 - a, 1e-25, 1)
        a = np.clip(a, 1e-25, 1)
        l = -np.sum(y_train * np.log(a) +
                    (1 - y_train) * np.log(one_minus_a)) / self.bs
        error = (a - y_train) / (a*one_minus_a) / self.bs
        return error, l, acc

    def MSE(self, y_train, a):
        acc = 0
        l = np.sum((y_train - a)**2) / self.bs / 2
        error = -(y_train - a) / self.bs
        return error, l, acc

    def Labelling(self, y):
        self.uni_labels = np.unique(y)
        label = np.zeros((len(y), len(self.uni_labels)))
        for i in range(len(y)):
            pos = np.where(self.uni_labels == y[i])
            label[i, pos] = 1
        return label

    def compile(self, a_func, optimizer, loss="MSE"):
        self.a_func = a_func.lower()
        self.optimizer = optimizer
        self.loss = loss.lower()
        assert self.loss in ["MSE", "binary cross entropy"]

    def train(self, X, y, bs=32, EPOCHS=1e4, shuffle=True):
        indices = np.arange(len(y))
        if shuffle:
            indices = np.random.permutation(len(y))
        if self.loss == "binary cross entropy":
            self.add(Dense(len(np.unique(y))))
            y = self.Labelling(y)
        elif len(np.shape(y)) == 1:
```

```
            self.add(Dense(1))
        else:
            self.add(Dense(np.shape(y)[1]))
        self.add(Activation(self.a_func))
        for layer in self.network:
            layer.optimizer = copy_class(self.optimizer)

        X_train, y_train = self.Create_mini_batch(X[indices], y[indices], bs)
        self.bs = bs

        for epoch in range(int(EPOCHS)):
            t = trange(len(y_train), desc="Loss: 0.0000, Accuracy: 0.0000")
            Loss, Accuracy = 0, 0
            for i in t:
                a = np.moveaxis(X_train[i], 0, -1) # move m to last index
                for layer in self.network:
                    a = layer.Forward(a)
                if self.loss == "binary cross entropy":
                    err, l, acc = self.Binary_Cross_Entropy(y_train[i].T, a)
                else:
                    err, l, acc = self.MSE(y_train[i].T, a)
                Accuracy += acc / len(y_train)
                Loss += l / len(y_train)

                for layer in reversed(self.network):
                    err = layer.Backward(err)
                description = ("Loss: " + f"{Loss:0.4f}, "
                               + "Accuracy: " + f"{Accuracy:0.4f}")
                t.set_description(description)

    def predict(self, X):
        X_test, y_test = self.Create_mini_batch(X, None, min(self.bs, len(X)))
        y_pred = []
        for i in trange(len(X_test)):
            a = np.moveaxis(X_test[i], 0, -1)
            for layer in self.network:
                a = layer.Forward_Test(a)
            if self.loss == "binary cross entropy":
                hat_y = self.uni_labels[np.argmax(a, axis=0)]
            else:
                hat_y = a
            y_pred.append(hat_y)
        return np.concatenate(y_pred, axis=0)
```

Programme 11.6.5: Sequential model in Python saved as `Sequential.py`.

The `Sequential()` class object can be used in regression problem with MSE as the loss function and classification problem with binary cross-entropy as the loss function. In classification problem, it first transforms the class label y into C, where C is the size of the class, number of dummy variables via the `Labelling()` function. It then creates the mini-batch of the training or testing dataset through the `Create_mini_batch()` function. Note that if the mini-batch has size of 1, the input `X` will be either a one-dimensional array for MLP or a three-dimensional tensor image for CNN, we therefore add one dimension to indicate the batch size of 1. The `Binary_Cross_Entropy()` and `MSE()` function compute the loss and error rate based on the binary cross-entropy and MSE loss functions respectively.

11.6.1 CNN with MNIST

Based on the codes in the previous subsections, we try them with a dataset from MNIST [5], which have been descripted in Subsection **??**. In Python, the MNIST dataset is embedded inside the `tensorflow` library. It particular focus on training and inference of deep neural networks, including ANN, CNN, RNN. However, in this MNIST example, we only use the dataset but not the CNN model. The training set consists of 60,000 data point; while the test set consists of 10,000 data point. Each data point is a gray scale ($n_v = 1$) images with pixel size of 28×28 of handwritten digits from numbers of 0-9.

In general, when training a CNN model under the `tensorflow` library, one should set the number of epochs at least 5. However, due to the computational cost, we only use 2 epochs, but it still yields good prediction power on test set thanks to the use of batch normalization in the Dense layer.

In this model, the input is of shape $1 \times 28 \times 28$. (See Figure 11.6.1)

1. **Conv1**: The first CNN layer has 32 filters, where each filter has shape 3×3 with stride size 1. The output becomes a tensor of shape $32 \times 28 \times 28$, where 32 comes from the number of filters, and 28 comes from the use of padding such that the input dimension and the output dimension remains unchanged after CNN. *Sigmoid* is then used as the activation function.
2. **Pool1**: The first pooling layer has pool size of 2 and stride size of 2, the output becomes another tensor of shape $32 \times 14 \times 14$, where $28/2 = 14$ comes from the pooling mechanism.
3. **Conv2**: The second CNN layer has 32 filters, where each filter has shape 3×3 with stride size 1. The output becomes another tensor of shape $32 \times 14 \times 14$, padding is also used to ensure the input dimension and the output dimension remains unchanged after CNN. *Sigmoid* is then used as the activation function.
4. **Pool2**: The second pooling layer has pool size of 2 and stride size of 2, the output becomes another tensor of shape $32 \times 7 \times 7$, where $14/2 = 7$ comes from the pooling mechanism.
5. **GAP**: The output tensor of the second pooling layer is transformed into a one-dimensional vector of lenght 32 by *Global Average Pooling*, where 32 comes from the number of channels in the second pooling layer.
6. **MLP**: GAP is fully connected to a Dense layer (Multi-Layer Perceptrons: MLP) with 32 neurons, with *sigmoid* as the activation function. Moreover, batch normalization is used in this layer.
7. **Output**: Since the label y is any number between 0-9, the output layer has 10 neurons, with *softmax* as the activation function.

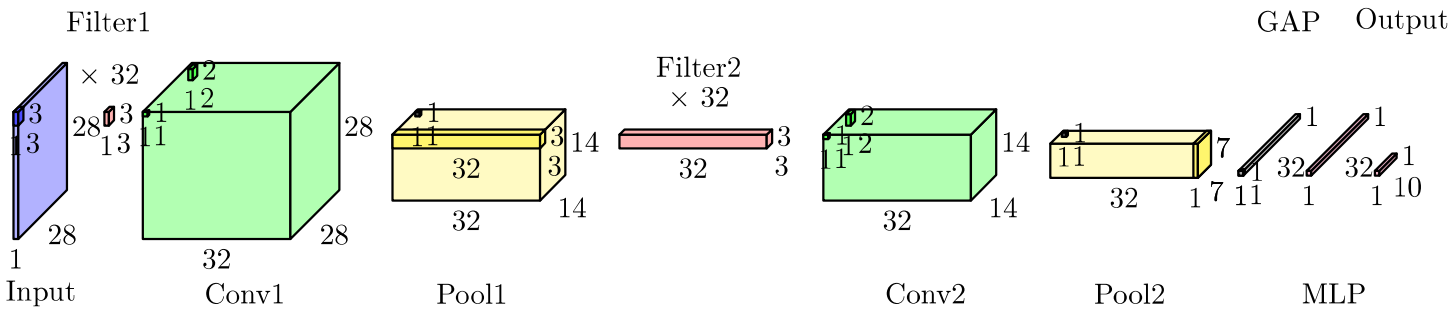


Figure 11.6.1: A simplified flowchart for a CNN model with MNIST.

Here is the example code:

```
import numpy as np
import keras
from Enhanced_Gradient_Descent_v2 import SGD, Adam, RMSprop, Adagrad, Momentum
from sklearn.metrics import confusion_matrix
from CNN import Flattening, Convolutional, Pooling
from Dense import Dense
from Intermediate_Layers import Activation, Dropout, Batch_Normalization
from Sequential import Sequential

(X_train, y_train), (X_test, y_test) = keras.datasets.mnist.load_data()
N_train, N_test = len(X_train), len(X_test)
X_train = X_train.reshape(N_train, -1)/255
X_test = X_test.reshape(N_test, -1)/255

model = Sequential()
model.add(Dropout(0.2, method="gaussian"))
model.add(Dense(64))
model.add(Activation("ReLU"))
model.add(Batch_Normalization())
model.add(Dropout(0.5, method="gaussian"))

model.compile("softmax", Adam(1e-2), loss="Binary Cross Entropy")
model.train(X_train, y_train, EPOCHS=10)

y_pred = model.predict(X_test)
print(np.mean(y_test == y_pred))
print(confusion_matrix(y_test, y_pred))
```

Programme 11.6.6: MLP and CNN models in Python

Here the `Flattening()`, `Convolutional()`, and `Pooling()` come from Programme 11.6.1, Programme 11.6.3, and Programme 11.6.4 respectively; the `Dense()` comes from Programme 10.1.2; the `Activation()`, `Dropout()`, and `Batch_Normalization()` come from Programme 10.1.3, Programme 10.5.1, and Programme 10.4.1 respectively; the `Sequential()` comes from Programme 11.6.5.

Moreover, the `Enhanced_Gradient_Descent_v2` is based on the Programme 8.3.1. The difference is that the later only accounts for $\boldsymbol{\theta}$ is a one-dimensional vector, but the former accounts for multi-dimensional parameters.

```
import numpy as np
```

```
2
3  class SGD():
4      def __init__(self, eta=1e-3):
5          self.eta = eta
6          assert self.eta > 0 and isinstance(self.eta, (int, float))
7
8      def Delta(self, dg):
9          return -self.eta * dg
10
11 class Adagrad():
12     def __init__(self, eta=1e-3, epsilon=1e-8):
13         self.eta = eta
14         self.epsilon = epsilon
15         self.past_g = 0
16         assert self.eta > 0 and isinstance(self.eta, (int, float))
17         assert 0 < self.epsilon < 1 and isinstance(self.epsilon, float)
18
19     def Delta(self, dg):
20         self.past_g += dg**2
21         return -self.eta * dg / (self.past_g**(1/2) + self.epsilon)
22
23     class RMSprop():
24         def __init__(self, eta=1e-3, gamma=0.9, epsilon=1e-8):
25         self.eta = eta
26         self.gamma = gamma
27         self.epsilon = epsilon
28         self.v = 0
29         assert self.eta > 0 and isinstance(self.eta, (int, float))
30         assert 0 < self.gamma < 1 and isinstance(self.gamma, float)
31         assert 0 < self.epsilon < 1 and isinstance(self.epsilon, float)
32
33     def Delta(self, dg):
34         self.v = self.gamma * self.v + (1 - self.gamma) * dg * dg
35         return -self.eta * dg / (self.v**(1/2) + self.epsilon)
36
37 class Momentum():
38     def __init__(self, eta=1e-3, gamma=0.9):
39         self.eta = eta
40         self.gamma = gamma
41         self.m = 0
42         assert self.eta > 0 and isinstance(self.eta, (int, float))
43         assert 0 < self.gamma < 1 and isinstance(self.gamma, float)
44
45     def Delta(self, dg):
46         self.m = self.gamma * self.m - (1 - self.gamma) * dg
47         return self.eta * self.m
48
49 class Adam():
50     def __init__(self, eta=1e-3, beta1=0.9, beta2=0.999, epsilon=1e-8):
51         self.m, self.v = 0, 0
52         self.beta1 = beta1
53         self.beta2 = beta2
54         self.epsilon = epsilon
55         self.eta = eta
56         self.t = 0
57         assert self.eta > 0 and isinstance(self.eta, (int, float))
58         assert 0 < self.beta1 < 1 and isinstance(self.beta1, float)
```

```
        assert 0 < self.beta2 < 1 and isinstance(self.beta2, float)
        assert 0 < self.epsilon < 1 and isinstance(self.epsilon, float)

    def Delta(self, dg):
        self.t += 1
        self.m = self.beta1 * self.m - (1 - self.beta1) * dg
        self.v = self.beta2 * self.v + (1 - self.beta2) * dg**2
        hat_m = self.m/(1 - self.beta1**self.t)
        hat_v = self.v/(1 - self.beta2**self.t)
        return self.eta * hat_m / (hat_v**(1/2) + self.epsilon)
```

Programme 11.6.7: Modified Enhanced gradient descent in Programme 8.3.1 Python.

The accuracy of the model in the training set is around 95.5% and the accuracy of the model in the test set is around 94.5%. Therefore, in general, the model is successful in determing the human handwritten digits. However, even under a decent cpu, this model has been trained for 4 hours.

11.6.2 Cat Dog Example

In this subsection, as a common example, we shall use CNN designed under the `tensorflow` of Python to discriminate images between cats and dogs[16]. The dataset stored at microsoft company, PetImages folder contains a total of 25,000 images, half of them are the images of cats, while another half are that of dogs.

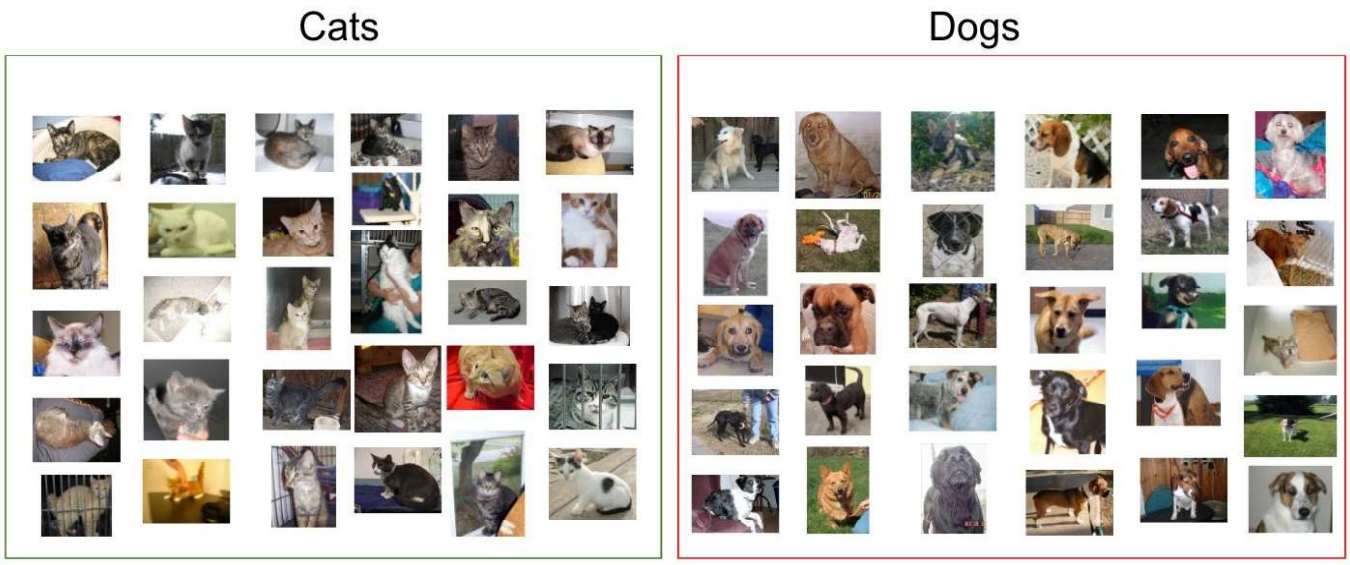


Figure 11.6.2: Cats and Dogs Dataset

From Figure 11.6.2, we can have some preliminary observations about the dataset and refine the dataset a bit before further processing:

1. Images are all colored, one may use RGB with a volume depth $n_v = 3$ fore the channels of the input. Nevertheless, for the sake of computational convenience, we only use gray scale ($n_v = 1$) images as input.
2. Images have different rectangular shapes, so rescaling of the image into ones of the same square shape is applied. In this example, the dimension is transformed into 100×100 pixels by using linear interpolations of colours.
3. The labels are originally in the name of the folder, namely, "cats" and "dogs", the text labels are now converted into numerical labels, "cat=0" and "dog=1".

In Programme 11.6.8, we lay down the code for standardizing the images:

```
# Operating System to link with the folder in which the images are saved
import os
import cv2          # OpenCV = Open source for Computer Vision library
import numpy as np
import random       # random shuffling for the dataset
# Pickle is the process in which different subsidaries of Python object
# is converted to a byte stream, i.e. a very long vector.
import pickle
# tqdm = taqadum, meaning the processing passed for each iteration [===>...]
```

[16]This dataset can be found at Microsoft company website: `https://www.microsoft.com/en-us/download/details.aspx?id=54765`.

```
from tqdm import tqdm

random.seed(4012)

Train_Name = ["Cat", "Dog"]
Train_Pickle = ["Feature.pickle", "Label.pickle"]
Test_Name = ["Test"]
Test_Pickle = ["gray_test.pickle", "rgb_test.pickle"]
IMG_SIZE = 100 # Define the rescaling factor that rescales all images into 100x
    100 pixels

def Create_Dataset(CATEGORIES, Pickle_Name, Train=True):
    # CATEGORIES = Cat or Dog in text; Pickle_Name = file name of the input
    # for the next CNN saved in "Pickle format"
    # declaration of the array format of these fields
    data, FEATURES, LABELS = [], [], []
    for category in CATEGORIES:
        # path to dataset: abspath = ABSolute PATH of the folder
        # getcwd = GET to the current Working Directory
        # join() = join the text strings by slash "/"
        path = os.path.join(os.path.abspath(os.getcwd()),"PetImages",category)
        # get a numeric label  (0 or 1). In this case, 0 for cat and 1 for dog
        # by default, declaring a list object, the index function will be
        # automatically amended
        y_num = CATEGORIES.index(category)

    # DO NOT save any other files in this folder!!!
    # for all image filenames in the current working directory
    for img_Name in tqdm(os.listdir(path)):
        # create the storage address for the image file with the format
        # "current directory/filename" which is the identity of each image
        img_Path = os.path.join(path, img_Name)
        # convert RGB to Grayscale
        # create a matrix array to store the pixel information of each picture
        img_array = cv2.imread(img_Path)
        # Some error exists in cv2
        if img_array is not None:
            # BGR is the RGB in reverse format
            gray_array = cv2.cvtColor(img_array, cv2.COLOR_BGR2GRAY)
            # rescale the images to predefined size
            rescale_array = cv2.resize(gray_array, (IMG_SIZE, IMG_SIZE))
            if Train:
                data.append([rescale_array, y_num])
            else:
                # For testing, we prepare to return both feature image
                # for the predicted label and the original image
                data.append([rescale_array, img_array])

    if Train:
        random.shuffle(data)

    for X, y in data:
        FEATURES.append(X)
        LABELS.append(y)

    # convert the feature image input as a feature input matrix with a
    # dimension of N x s_I x s_I x n_v, where N is the sample size
```

```
66      FEATURES = np.array(FEATURES).reshape(-1, IMG_SIZE, IMG_SIZE, 1)
67      with open(Pickle_Name[0], "wb") as f:   # wb = Write Binary
68          pickle.dump(FEATURES, f)    # Put feature image in the pickle format
69      with open(Pickle_Name[1], "wb") as f:
70          pickle.dump(LABELS, f)  # Put label or test image in pickle format
71
72  Create_Dataset(Train_Name, Train_Pickle, True)
73  Create_Dataset(Test_Name, Test_Pickle, False)
```

Programme 11.6.8: Create training and testing dataset of Cats and Dogs in Python for the later CNN classification.

In this model, we use 3 convolutional layers with the corresponding max-pooling layer for each, and 2 dense hidden layers for the MLP, where the last `Dense(1)` in the line 43 is the output layer. In this output layer, the closer to the value 0, the prediction will likely be taken as a cat; the closer to the value 1, the more likely we predict the image as a dog. In Programme 11.6.9, we use the tensorflow to carry out this CNN classification:

```
1  from tensorflow as tf
2  from tensorflow.keras.models import Sequential
3  from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten
4  from tensorflow.keras.layers import Dense, Activation, BatchNormalization
5  import numpy as np
6  import pickle
7
8  # loading the data
9  X = pickle.load(open("Feature.pickle","rb"))    # rb: Read Binary
10 y = pickle.load(open("Label.pickle","rb"))
11
12 # X is a feature input matrix of dimension N x s_I x s_I x n_v
13 X = np.array(X)
14 y = np.array(y)
15
16 tf.random.set_seed(4013)
17
18 # Build CNN
19 model = Sequential()
20
21 # Build 3 (convolutional + max-pooling) layers
22 # 32 filters each of size 3x3,
23 # padding is set so that the input and the output are of the "SAME" dimension
24 # recall the shape() function of an numpy array is by default built-in
25 # by the numpy array object declaration
26 model.add(Conv2D(32, (3, 3), padding='SAME', input_shape=X.shape[1:]))
27 model.add(Activation('relu')) # relu:REctified Linear Unit
28 # In practice, pooling size is often of 2
29 model.add(MaxPooling2D(pool_size=(2, 2)))
30
31 model.add(Conv2D(64, (3, 3), padding='SAME'))
32 model.add(Activation('relu'))
33 model.add(MaxPooling2D(pool_size=(2, 2)))
34
35 model.add(Conv2D(128, (3, 3), padding='SAME'))
36 model.add(Activation('relu'))
37 model.add(MaxPooling2D(pool_size=(2, 2)))
38
39 model.add(Flatten())
```

```
model.add(Dense(64, activation="relu"))
model.add(BatchNormalization())
model.add(Dense(32, activation="relu"))
model.add(BatchNormalization())
model.add(Dense(1, activation="sigmoid"))

model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

print(model.summary())
# 0.2 indicates training:testing = 8:2 for each epoch
model.fit(X, y, batch_size=32, epochs=10, validation_split=0.2)

model.save('Dog_Cat.model')
```

Programme 11.6.9: CNN with tensorflow in Python.

When using the `model.summary()`, we have the summary of the model as below:

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_19 (Conv2D)           (None, 100, 100, 32)      320
_________________________________________________________________
activation_19 (Activation)   (None, 100, 100, 32)      0
_________________________________________________________________
max_pooling2d_19 (MaxPooling (None, 50, 50, 32)        0
_________________________________________________________________
conv2d_20 (Conv2D)           (None, 50, 50, 64)        18496
_________________________________________________________________
activation_20 (Activation)   (None, 50, 50, 64)        0
_________________________________________________________________
max_pooling2d_20 (MaxPooling (None, 25, 25, 64)        0
_________________________________________________________________
conv2d_21 (Conv2D)           (None, 25, 25, 128)       73856
_________________________________________________________________
activation_21 (Activation)   (None, 25, 25, 128)       0
_________________________________________________________________
max_pooling2d_21 (MaxPooling (None, 12, 12, 128)       0
_________________________________________________________________
flatten_7 (Flatten)          (None, 18432)             0
_________________________________________________________________
dense_20 (Dense)             (None, 64)                1179712
_________________________________________________________________
batch_normalization_10 (Batc (None, 64)                256
_________________________________________________________________
dense_21 (Dense)             (None, 32)                2080
_________________________________________________________________
```

```
batch_normalization_11 (Batc (None, 32)                128
_________________________________________________________________
dense_22 (Dense)             (None, 1)                 33
=================================================================
Total params: 1,274,881
Trainable params: 1,274,689
Non-trainable params: 192
```

Particularly, the output shape is still $N \times s_{\mathcal{I}} \times s_{\mathcal{I}} \times n_f^{(\ell)}$ at the convolutional or max-pooling layer ℓ. The number of parameters for each convolutional layer ℓ equals to $n_f^{(\ell)} \cdot (s_f^{(\ell)} \cdot s_f^{(\ell)} \cdot n_v^{(\ell)} + 1)$, so that in each filter component, there are $s_f^{(\ell)} \cdot s_f^{(\ell)} \cdot n_v^{(\ell)}$ weights and 1 bias, and $n_v^{(\ell)}$ is essentially equal to the number of filters in the last convolutional layer. Recall in Subsection 11.3.2, the flattening layer has the number of entries $s_{\mathcal{I}}^{(\ell)} \cdot s_{\mathcal{I}}^{(\ell)} \cdot n_v^{(\ell)}$. The input layer for the MLP creates $64 \cdot 18432$ weights plus 64 biases. For the batch normalization, the number of parameters equal to the $d_\ell \cdot 4$, these four are γ, β, μ, and σ^2 of the batch. Finally, the number of non-trainable parameters are exactly those means and sd from the batch normalization equals to $2 \times 64 + 2 \times 32$.

After building the model, we test it with four images, the code is shown below:

```
import tensorflow as tf
import matplotlib.pyplot as plt
import pickle
import numpy as np
import cv2

IMG_SIZE = 100 # Rescale every images into 100x100
CATEGORIES = ["Cat", "Dog"]

Features = pickle.load(open("gray_test.pickle","rb"))
img_test =  pickle.load(open("rgb_test.pickle","rb"))

model = tf.keras.models.load_model('Dog_Cat.model')

fig = plt.figure(figsize=(25, 5))
for i in range(len(img_test)):
    CD_fig = fig.add_subplot(1, 4, i+1)      # rows, columns, index
    # Turn X into a feature input matrix of N x s_I x s_I x n_v,
    # here N=4, n_v is still 1
    X = np.array(Features[i], dtype=np.float32).reshape(-1, 100, 100, 1)
    # squeezing means converting a tensor value into a numerical value
    prediction = np.squeeze(model.predict(X))
    print(prediction)
    # show the image correspondingly to the test sample datum
    # auto is to fit automatically to the figure layout
    CD_fig.imshow(cv2.cvtColor(img_test[i], cv2.COLOR_BGR2RGB), aspect='auto')
    title_name = CATEGORIES[int(np.round(prediction))] + f"\n{prediction}"
    plt.title(title_name, fontsize=20)
    # hide the horizontal and vertical coordinate axes for this figure
    CD_fig.axes.get_xaxis().set_visible(False)
    CD_fig.axes.get_yaxis().set_visible(False)

plt.show()
```

```
34 fig.savefig("CatDog_Predict.png")
```

Programme 11.6.10: Make prediction for four images in Python.

and the results are

Figure 11.6.3: Predicted labels of CNN are on the top; prediction values of CNN are in the middle; these test images are augmented at the bottom.

When the prediction number close to 0 means that the model recognizes the test image as a cat; while the prediction number close to 1 gives that the model consider the test image as a dog.

11.6.3 Towards Explainable AI: Interpreting CNN Mechanism

In Subsection 11.6.1, we illustrated the application of CNN for MNIST. In this last subsection, we shall explain our theory the Extended MNIST (EMNIST)[17] released in Cohen et al. (2017), the handwritten characters and digits dataset, collected from 3600 writers. It contains all 26 handwritten English alphabets with both uppercase and lowercase, and it also contains all 10 handwritten digits from 0 to 9. Each alphabet or each digit in EMNIST contains 8 data files, namely `hsf_0` to `hsf_7` where `hsf` stands for Handwriting Sample Form, each file contains hundreds of handwritten images. However, for simplicity, to illustrate how to interpret the inference results from CNN; we only consider the capital letter "I" and the capital letter "Q" from this EMNIST dataset.

Firstly, as usual, we transform the images into gray-scale and rescale each image to 32×32 pixels.

```
import os
import cv2
import numpy as np
import random
import pickle
from tqdm import tqdm

random.seed(4015)
Train_Pickle = ["Feature", "Label"]
FOLDERS_Train = ["hsf_0", "hsf_1", "hsf_3", "hsf_4", "hsf_6"]
FOLDERS_Test = ["hsf_7"]
Test_Pickle = ["Feature_Test", "Label_Test"]
IMG_SIZE = 32 # Rescale every images into 32x32

def Create_Dataset(LETTERS, Pickle_Name, FOLDERS, Train=True):
    data, FEATURES, LABELS = [], [], []
    for letter in LETTERS:
        for folder in FOLDERS:
            # path to dataset
            path = os.path.join(os.path.abspath(os.getcwd()), letter, folder)
            y_num = LETTERS.index(letter)

            # DO NOT save any other files in this folder!!!
            for img_Name in tqdm(os.listdir(path)):
                img_Path = os.path.join(path, img_Name)
                img_array = cv2.imread(img_Path, cv2.IMREAD_COLOR)
                # Some error exists in cv2
                if img_array is not None:
                    gray_array = cv2.cvtColor(img_array, cv2.COLOR_BGR2GRAY)
                    # rescale the images to predefined size
                    rescale_array = cv2.resize(gray_array, (IMG_SIZE,IMG_SIZE))
                    data.append([rescale_array, y_num])

    if Train:
        random.shuffle(data)

    for X, y in data:
        FEATURES.append(X)
```

[17] The EMNIST dataset can be found in `https://www.nist.gov/srd/nist-special-database-19`. This dataset contains four directories and only the directory `by_class`, that discards the writer and field information, is adopted in this subsection.

```
        LABELS.append(y)

    FEATURES = np.array(FEATURES).reshape(-1, IMG_SIZE, IMG_SIZE, 1)

    with open(f"{Pickle_Name[0]}_{LETTERS[0]}_{LETTERS[1]}.pickle", "wb") as f:
        pickle.dump(FEATURES, f)
    with open(f"{Pickle_Name[1]}_{LETTERS[0]}_{LETTERS[1]}.pickle", "wb") as f:
        pickle.dump(LABELS, f)

Create_Dataset(["I", "Q"], Train_Pickle, FOLDERS_Train, True)
Create_Dataset(["I", "Q"], Test_Pickle, FOLDERS_Test, False)
```

Programme 11.6.11: Creating training and testing dataset of "I" vs "Q" in Python for the later CNN classification.

Here, for both capital letter "I" and "Q", the data files `hsf_0` to `hsf_6` are used as the training set saved as `Feature_I_Q.pickle` for the images and `Label_I_Q.pickle` for the labels, while the file `hsf_7` is used as the test set saved as `Feature_Test_I_Q.pickle` for the images and `Label_Test_I_Q.pickle` for the labels.

Next, we consider a simple 1-layer convolutional network, in which there is only one filter $\boldsymbol{K}$ of size $3 \times 3 \times 1$. Moreover, for the sake of convenience and without bringing in any substantial deviation in performance, we neglect both the pooling layer and the activation function in this convolutional layer. This convolutional layer is directly followed by a flattening layer, and then its output layer contains only one output neuron that incorporated the sigmoid activation function. To conclude, this network can be constructed in Python using `tensorflow` as follows:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Conv2D, Flatten
import tensorflow as tf
import numpy as np
import pickle

tf.random.set_seed(4015)

# I vs Q
X_train = pickle.load(open("Feature_I_Q.pickle","rb")) / 255
y_train = pickle.load(open("Label_I_Q.pickle","rb"))
X_train, y_train = np.array(X_train), np.array(y_train)

model = Sequential()
model.add(Conv2D(1, (3, 3), strides=(1, 1), padding='valid',
          input_shape=X_train.shape[1:]))
model.add(Flatten())
model.add(Dense(1, activation="sigmoid"))
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
model.fit(X_train, y_train, batch_size=32, epochs=25, validation_split=0.2)

K, b_K = model.layers[0].get_weights()
```

Programme 11.6.12: A simple CNN with "I" and "Q" dataset in Python.

We next print the first four "I"'s and first four "Q"'s images obtained from the training dataset.

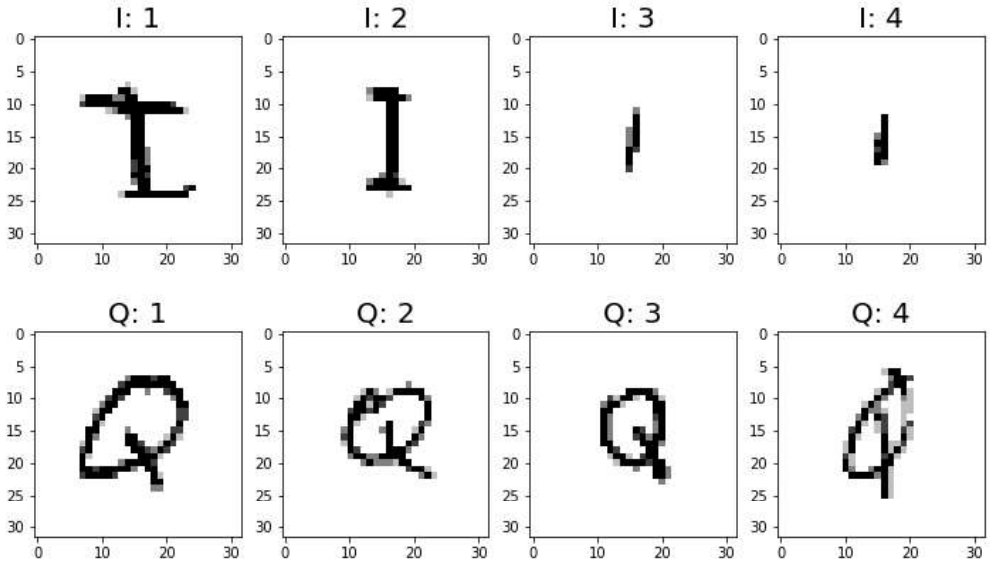


Figure 11.6.4: First four "I"'s and first four "Q"'s images in the training dataset.

Note that in `tensorflow`, it computes cross-correlation instead of convolution in CNN for varying filters. We therefore compute the cross-correlation, instead of convolution, between these "I"'s and "Q"'s images and the filter `K_Q`, we then add a bias `b_Q`:

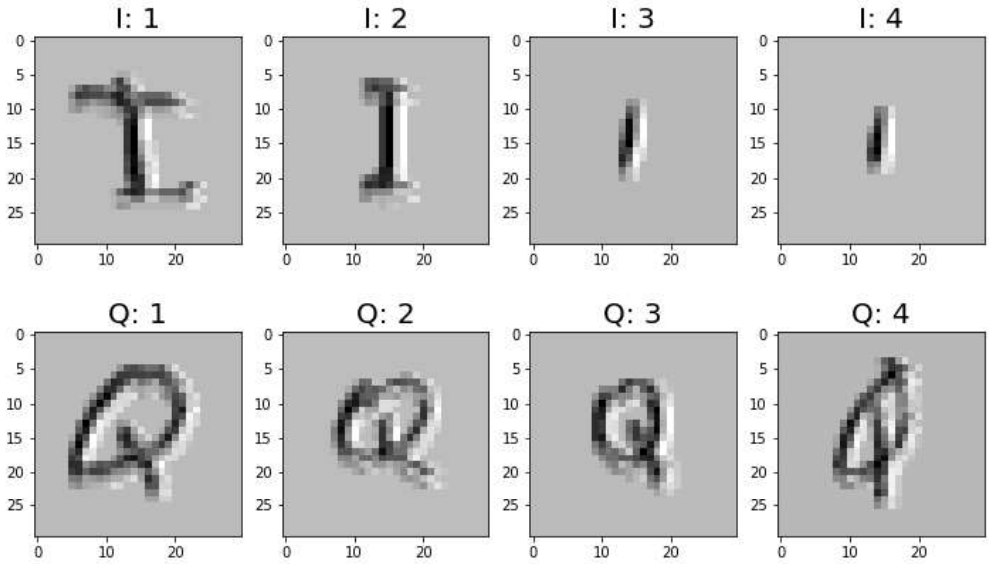


Figure 11.6.5: Cross-correlations between these "I"'s and "Q"'s images in the training dataset with the filter `K` and bias `b_K` learnt in Programme 11.6.12.

We test this filter `K` and bias `b_K` learnt in Programme 11.6.12 with 8 fixed filters with basic straight line edge detection:

$$\boldsymbol{X}_1 = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}, \quad \boldsymbol{X}_2 = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \quad \boldsymbol{X}_3 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}, \quad \boldsymbol{X}_4 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

$$\boldsymbol{X}_5 = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \quad \boldsymbol{X}_6 = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}, \quad \boldsymbol{X}_7 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}, \quad \boldsymbol{X}_8 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$$

The trained filter `K` and bias `b_K` are, respectively:

$$\boldsymbol{K} = \begin{pmatrix} -0.55 & 0.04 & 0.63 \\ 0.05 & -0.03 & 0.37 \\ -0.49 & 0.8 & 1.03 \end{pmatrix} \quad \text{and} \quad b_K = -0.4\,. \tag{11.6.1}$$

The weighted sums $Z_n = \boldsymbol{X}_n \star \boldsymbol{K} + b_K$ (cross-correlation instead of convolution), for $n = 1, \cdots, 8$, are

$$Z_1 = 0\,, \quad Z_2 = 0.42\,, \quad Z_3 = 1\,, \quad Z_4 = 0.05\,, \quad Z_5 = 0\,, \quad Z_6 = 0\,, \quad Z_7 = 0.94\,, \quad Z_8 = 0.02\,.$$

We observe that Z_4 and Z_8 are nearly zero, indicating that the trained filter $\boldsymbol{K}$ does not detect diagonal line segments. On the other hand, Z_2, Z_3 and Z_7 are significantly different from zero, denoting that $\boldsymbol{K}$ aims to detect some specific vertical and horizontal line segments. In particular, the vertical and horizontal lines are the basic building blocks of the capital letter "I", based on which, we suspect that this trained CNN in Programme 11.6.12 classifies whether the image is the capital letter "I" against "Q" which is more round in shape.

We next propose a fixed filter, combined from two simple fixed filters for detecting vertical and horizontal edges respectively:

$$\boldsymbol{F}_1 = \begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix} \quad \text{and} \quad \boldsymbol{F}_2 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}.$$

Consider the following linear combination:

$$\boldsymbol{F} = \boldsymbol{F}_1 + \boldsymbol{F}_2 = \begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{pmatrix}, \tag{11.6.2}$$

with a bias $b_F = -0.67$. The weighted sums $Z_n = \boldsymbol{X}_n \star \boldsymbol{F} + b_F$ (cross-correlation instead of convolution), for $n = 1, \cdots, 8$, are

$$Z_1 = 0\,, \quad Z_2 = 0.33\,, \quad Z_3 = 1\,, \quad Z_4 = 0.33\,, \quad Z_5 = 0\,, \quad Z_6 = 0\,, \quad Z_7 = 1\,, \quad Z_8 = 0.33\,.$$

Here the bias term b_F is chosen at -0.67 to give the highest accuracy by obtaining the lowest loss in the training set given the fixed filter $\boldsymbol{F}$. In particular, b_F reduces the original values of $\boldsymbol{X}_n \star \boldsymbol{F} + b_F$ of Z_2, Z_4, and Z_8 to $1 + b_F = 1 - 0.67 = 0.33$ such that the sensitivities of correlation operations on the corresponding vertical and horizontal edges are reduced. We shall now compute the cross-correlation between these first four "I"'s and "Q"'s images and the fixed filter $\boldsymbol{F}$, and then add a bias $b_F = -0.67$:

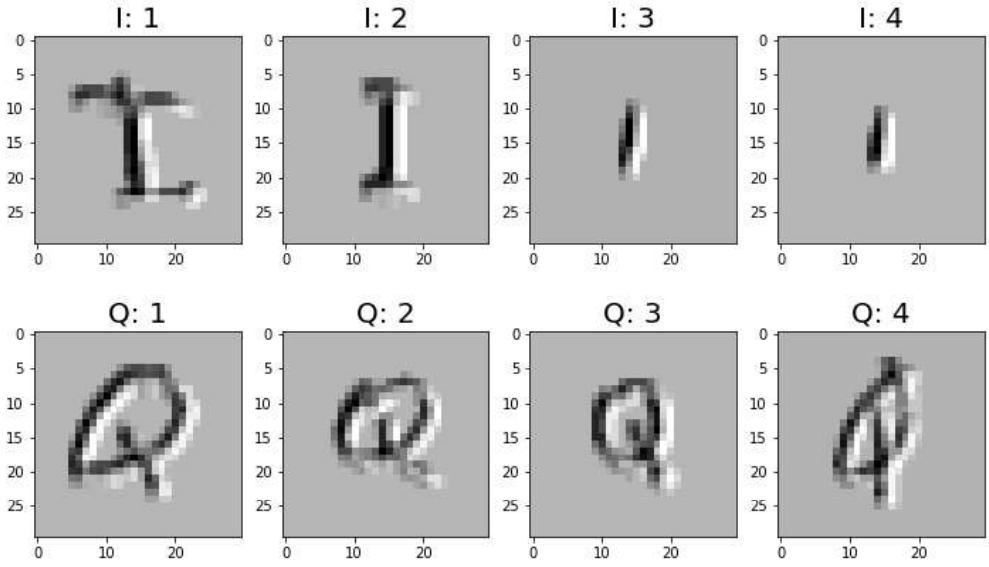


Figure 11.6.6: Cross-correlations of the "I"'s and "Q"'s images in the training dataset with the self-defined fixed filter $\boldsymbol{F}$ and bias b_F.

Finally, we use the test set for computing the loss and the accuracy by using the parameters $(\boldsymbol{F}, b_F)$ in (11.6.2) of `fix_model` and $(\boldsymbol{K}, b_K)$ in (11.6.1) of `model`, respectively:

```
X_test = pickle.load(open("Feature_Test_I_Q.pickle","rb")) / 255
y_test = pickle.load(open("Label_Test_I_Q.pickle","rb"))
X_test, y_test = np.array(X_test), np.array(y_test)

F1 = np.array([[-1,0,1],
               [-1,0,1],
               [-1,0,1]])

F2 = np.array([[0,0,0],
               [0,0,0],
               [1,1,1]])

F = F1 + F2
b_F = -0.67
print(F)
print(b_F)

fix_model = tf.keras.models.clone_model(model)
fix_model.layers[0].set_weights([F.reshape(np.shape(K)),
                                 np.array(b_F).reshape(np.shape(b_K))])
fix_model.layers[2].set_weights(model.layers[2].get_weights())
fix_model.compile(loss='binary_crossentropy', optimizer='adam',
                  metrics=['accuracy'])
fix_model.evaluate(X_train, y_train)

model.evaluate(X_train, y_train)
```

```
[[-1  0  1]
 [-1  0  1]
 [ 0  1  2]]
-0.67
38/38 [==============================] - 0s 1ms/step - loss: 0.0376 - accuracy:
    0.9916
38/38 [==============================] - 0s 1ms/step - loss: 0.0323 - accuracy:
    0.9916
```

Programme 11.6.13: Computing loss and accuracy in the test set using parameters $(\boldsymbol{K}, b_K)$ and $(\boldsymbol{F}, b_F)$ in Python.

From the output above, we observe that both models have essentially the same accuracy but the guessed model using the proposed filter $\boldsymbol{F}$ and bias b_F have a relatively, but not substantially, larger binary loss. Therefore, we infer that the CNN in `tensorflow` under the setting above learns "I"'s and "Q"'s likely through the discretion on whether an image contains both vertical and horizontal lines to conclude that is an "I" or not.

However, most of the CNNs are soley black box, where only a small subset of the CNN models can be interpretable reasonably. In the context of explainable AI, in general we often pick a more robust model though it may not perform the best, but we believe that this robust model is more or less interpretable. This is a tradeoff between the interpretation and the performance of the model.

BIBLIOGRAPHY

[1] Cohen, G., Afshar, S., Tapson, J., and Van Schaik, A. (2017). EMNIST: Extending MNIST to handwritten letters. In 2017 international joint conference on neural networks (IJCNN) (pp. 2921-2926). IEEE.

[2] Fukushima, K. (1975). Cognitron: A self-organizing multilayered neural network. Biological cybernetics, 20(3), 121-136.

[3] Fukushima, K. (1988). Neocognitron: A hierarchical neural network capable of visual pattern recognition. Neural Networks, 1, 119-130.

[4] Hubel, D. H., and Wiesel, T. N. (1962). Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. The Journal of physiology, 160(1), 106-154.

[5] LeCun, Y. (1998). The MNIST database of handwritten digits. http://yann. lecun. com/exdb/mnist/.

[6] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.