Lucas David, 188972 | Image Manipulation | lucas.david@ic.unicamp.br

1 Introduction

This assignment was submitted to the class of 2021/1 of course Introduction to Image Processing (MO443) at Universidade Estadual de Campinas. Its goal is to implement basic numeric operations over images using Python programming language.

1.1 Dataset and Setup

In this work, I employed TensorFlow [1] as numeric framework for most manipulation performed. Additionally, NumPy [4] is also used whenever it offered an advantage in implementation simplicity. Google Colaboratory [2] was used as development platform, taking into consideration its reasonable resources available and bootstrap simplicity. The notebook produced is available for direct access[1].
To visualize the effect of the implemented filters beyond the few input images provided in class, we utilize 10 random images from the TF-Flowers[2] dataset, which can be downloaded in the tf-records format using the *tensorflow-datasets* library.
This dataset represents a multi-class (mono-label) image classification problem, and comprises 3,670 photographs of flowers associated with one of the following labels: *dandelion, daisy, tulips, sunflowers* and *roses*.
Images are represented by tensors of rank 3, of shape $(H, W, 3)$. They are first loaded into memory as n-dimensional arrays of type *uint8*. As many of the following operations are floating point precision, we opted to cast them imediately to *float32*. This is the default dtype for most operations in TensorFlow graphs.

Image may have different sizes, which prevents them from being directly stacked into a 4-rank tensor `(BATCH, HEIGHT, WIDTH, 3)`. We circumvented this problem using the following procedure:

1. Let $S^\star := (300, 300, 3)$ be a desired output shape and $S_I := (H_I, W_I, 3)$ be the actual shape of any given image I. I is resized so it's smallest component (H_I or W_I) matches the smallest component of $S^\star$ (namely, 300).

2. The largest component in the shape of I is now greater or equal to 300. We extract the central crop of size $(300, 300)$ from I, resulting in an image of shape $(300, 300, 3)$.

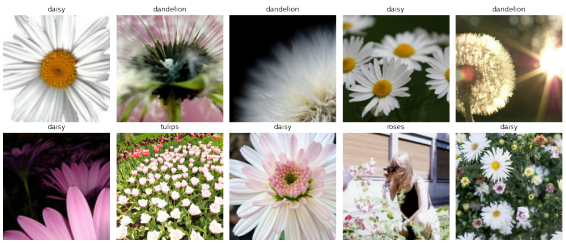

Figure 1: Examples in the TF-Flowers dataset.

2 Colored Images

In this section, we describe the implementation details for the operations applied when the program receives colored images.

2.1 Sepia Filter

$$\begin{pmatrix} R' \\ G' \\ B' \end{pmatrix} = \begin{pmatrix} 0.393R + 0.769G + 0.189B \\ 0.349R + 0.686G + 0.168B \\ 0.272R + 0.534G + 0.131B \end{pmatrix} \quad (1)$$

Proposed activity: implement the Sepia filter for images containing color information, represented by the linear system describe in Eq. 1.
A naive way to implement this transformation is to separate the image tensor I of shape $(B, H, W, 3)$ by its last axis, forming three tensors of shape $(B, H, W, 1)$. We multiply each and every component according to the rule in Eq. 1 and concatenate the result, as described in Lst. 1.

```
@tf.function
def sepia(x):
  x = tf.cast(x, tf.float32)
  r, g, b = tf.split(x, 3, axis=-1)

  y = tf.concat(
    (0.393*r + 0.769*g + 0.189*b,
     0.349*r + 0.686*g + 0.168*b,
     0.272*r + 0.534*g + 0.131*b),
    axis=-1)

  return tf.clip_by_value(y, 0, 255)
```

Listing 1: Naive implementation of the Sepia filter.

[1] Iterative report available at colab/mo-443-assignment-1
[2] TF-Flowers dataset is available at tensorflow.org/datasets/catalog/tf_flowers

A more elegant solution is to remember that every linear system (including $(R', G', B')^\intercal$, described above) can be interpreted as a multiplication between the input matrix and a coefficient matrix:

$$I \cdot S = \begin{bmatrix} \sum_k i_{0,k} s_{k,0} & \dots & \sum_k i_{0,k} s_{k,m} \\ \dots & \dots & \dots \\ \sum_k i_{n,k} s_{k,0} & \dots & \sum_k i_{n,k} s_{k,m} \end{bmatrix} \quad (2)$$

In our case, I has rank 4 (not a matrix), but the same equivalence applies, as the matrix multiplication is a specific case of the tensor dot product. There are multiple ways to perform this operation in TensorFlow:

1. `y = tf.matmul(x, s)`: inner-product over the inner-most indices in the input tensors (last axis of `x` and antepenultimate axis of `s`). This assumes the other axes represent batch-like information, and generalizes the matrix-multiplication operation for all cases (the input is a matrix, batch of matrix or batch of sequence of matrix, ...).
2. `y = x @ s`: override of `tf.matmul`, same resulting operation
3. `y = tf.tensorproduct(x, s, 1)`: the tensor dot product over one rank (the last in `x` and first in `sepia_weights`)
4. `y = tf.einsum('bhwc,ck->bhwk', x, s)`: Einstein's summation over the rank `c` (last in `x` and first in `s`)

```
sepia_weights = tf.constant(
  [[0.393, 0.349, 0.272],
   [0.769, 0.686, 0.534],
   [0.189, 0.168, 0.131]]
)

@tf.function
def sepia(x):
  y = x @ sepia_weights
  return tf.clip_by_value(y, 0, 255)
```

Listing 2: Most efficient implementation of the Sepia filter.

Fig. 2.1 illustrates the results of the application of the function described in Lst. 2 onto the 10 samples from the TF-Flowers dataset.

Figure 2: Sepia filter applied over the image samples.

2.2 Gray-Scale Transform

Proposed activity: given a colored image I in the RGB format, alter the image s.t. it contains only one color band, defined by the following equation:

$$I = 0.2989R + 0.5870G + 0.1140B$$

The solution is very similar to what was done in Subsection 2.1, except that the coefficient tensor is no longer a matrix, which preclude the usage of `@`. I used `tf.tensordot` to compute the inner product between the last axis of `x` and first (and only) axis of `gray_weights` (Lst. 3).
An acceptable alternative form would be `tf.einsum('bhwc,c->bhw', x, gray_weights)`.

```
gray_weights = tf.constant(
  [0.2989, 0.5870, 0.1140]
)

@tf.function
def grayscale(x):
  y = tf.tensordot(x, gray_weights, 1)
  return tf.clip_by_value(y, 0, 255)
```

Listing 3: RGB to gray-scale transformation.

Figure 3: The result of the gray-scale transformation applied over the image samples.

3 Monochromatic Images

In this section, we describe the implementation details for the operations applied when the program receives one or more monochromatic images. The proposed activity is to apply distinct constant kernels — 7 kernels of shape $(3, 3)$ and 2 of shape $(5, 5)$ — to the aforementioned images.

3.1 Convolution and Cross-correlation Operators

The convolution of a 1-D input sinal f and a 1-D kernel g is defined as the integration of the product between the two signals, when evaluated over the temporal component:

$$(f * g)(t) = \int f(\tau) g(t - \tau) d\tau$$

We observe from the equation above that one of the signals is reflected. This is essential so both functions are evaluated over the same time interval, resulting in the effect of "zipping" the two functions together. This effect is illustrated in the first column of Fig. 4.

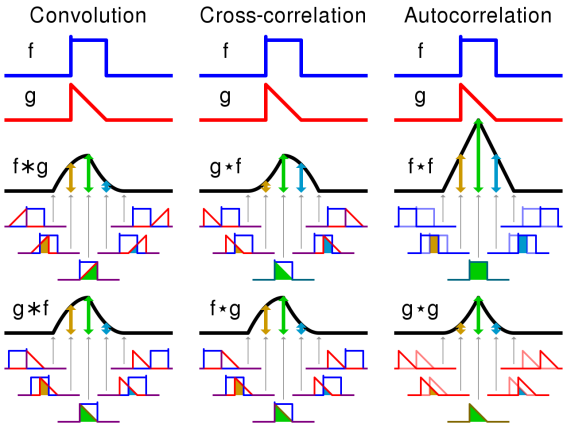

Figure 4: Comparison between convolution, cross-correlation and auto-correlation. Wikipedia. Available at: wikipedia.org/Cross-correlation.

On the other hand, *Cross-correlation* is a similar operation in which g slides over f without the aforementioned reflection:

$$(f * g)(t) = \int f(\tau)g(t+\tau)d\tau$$

The signals are associated in an inverted fashion, which is illustrated in the second column of Fig. 4. Finally, we can imagine that a 2-D signal (such as images) is reflected when both (x, y) axes are reflected. This is equivalent of rotating the image in $180°$.

Notwithstanding its name, the `tf.nn.conv2d(f, g)` function implements the cross-correlation function, and the convolution operation is supposedly performed by assuming the kernel g is already reflected. In the example below, we observe the output signal is obtained by the Cross-correlation eq.:

```
s = tf.constant(
  [[1., 2., 3.],
   [4., 5., 6.],
   [7., 8., 9.]]
)
k = tf.constant(
  [[1., 1.],
   [0., 0.]]
)
c = tf.nn.conv2d(
  tf.reshape(s, (1, 3, 3, 1)),
  tf.reshape(k, (2, 2, 1, 1)),
  strides=1,
  padding='VALID'
)

signal:
[[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
kernel:
[[1. 1.]
 [0. 0.]]
s*k:
[[ 3.  5.]
 [ 9. 11.]]
```

Listing 4: Illustration of the cross-correlation operation being executed when `tf.nn.conv2d` function is invoked.

This is a design decision which takes performance into account, as the rotation operations can be omitted during the feed-forward process. During training, kernels are correctly learnt through back-propagation by minimizing a given loss function (e.g. cross-entropy, N-pairs, focal loss). Interestingly enough, the differential of the **real-valued** cross-correlation with respect to its kernel (which is used to update the kernels) is the cross-correlation itself, rotated $180°$ (e.g. convolution).

In this particular case, in which the kernels are fixed, they supposedly represent regular filters. I therefore chose to implement the convolution as the correlation between an input signal I and the kernel k rotated $180°$.

Assuming the kernels are stored in the variables `h17` and `h89` of shape $(7, 3, 3)$ and $(2, 5, 5)$ respectively, this assignment can be trivially solved:

```
def rot180(k):
  k = tf.expand_dims(k, -1)
  k = tf.image.rot90(k, 2)
  k = tf.transpose(k, (1, 2, 3, 0))
  return k

y17 = tf.nn.conv2d(x, rot180(h17), 1, 'SAME')
y89 = tf.nn.conv2d(x, rot180(h89), 1, 'SAME')

yr1c2 = tf.sqrt(
  tf.nn.conv2d(
    x, rot180(h17[0]), 1, 'SAME')**2
  + tf.nn.conv2d(
      x, rot180(h17[1]), 1, 'SAME')**2
)

y = tf.concat((y17, y89, yr1c2), axis=-1)
y = tf.clip_by_value(y, 0, 255)
```

Listing 5: Implementation of the convolution operation using TensorFlow functions.

3.2 Implementation

In this subsection, I present my implementation (and derivation thereof) of the 2-dimensional convolution function. For simplicity, it is implemented as the correlation between an input signal I and the reflection of an input kernel k.

3.2.1 Study of a Use Case

I decided to start by considering a simple use case. For analytical convenience, I imagined this case to have the following characteristics:

1. Only one image and one kernel is involved in this operation.
2. No padding is performed in the input signal (i.e. *padding valid*).
3. I and k have different height and width values — namely, (H_I, W_I) and (H_k, W_k) —, and they are prime numbers. This is interesting when flattening a matrix into a vector, as prime numbers have distinct products and will result in dimensions that are easier to understand.
4. The kernel is not symmetric. Hence the convolution and cross-correlation functions will result in different signals.

I considered the following signals in my use-case:

$$I = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \\ 26 & 27 & 28 & 29 & 30 \\ 31 & 32 & 33 & 34 & 35 \end{bmatrix} \quad (3)$$

$$k = \begin{bmatrix} 0 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad (4)$$

I simulated the effect of a 2-D kernel k sliding across the spatial dimensions of I by "extracting" multiple non-mutually disjointed subsections of the image into a sequence of flattened patches, which could then be broadcast-multiplied by the flattened kernel and reduced with the sum operation. To make this "extraction" more efficient, an index-based mask was built and applied over the image. This approach requires the construction of two integer matrices R and C — each of shape $((H_I - H_k + 1)(W_I - W_k + 1), H_k W_I)$ —, but does not replicate the values contained within the input signal I. Rather, index-based masks create merely *views* of the n-dimensional arrays over which they are applied.
The indexing procedure can be described by the following steps.

1. The window starts at the first valid position in matrix I, and references the sub-matrix $I[0:2, 0:2]$ with the flatten index vector $[(0,0),(0,1),(0,2),(1,0),(1,1),(1,2)]$ or, for briefness, $\begin{bmatrix}00 & 01 & 02 & 10 & 11 & 12\end{bmatrix}$.
2. k slides horizontally until the last valid index of I, yielding $W_I - W_k + 1$ index vectors in total.
3. The window resets at the column of I, but positioned at the next following row.
4. The three steps above are repeated for each valid vertical index ($H_I - H_k + 1$ times), effectively covering all sections of the matrix I.

The use-case represented by Eq. 3 can therefore be indexed as:

$$M_I = \begin{bmatrix} 00 & 01 & 02 & 10 & 11 & 12 \\ 01 & 02 & 03 & 11 & 12 & 13 \\ 02 & 03 & 04 & 12 & 13 & 14 \\ & & & & & \\ 10 & 11 & 12 & 20 & 21 & 22 \\ 11 & 12 & 13 & 21 & 22 & 23 \\ 12 & 13 & 14 & 22 & 23 & 24 \\ \dots & & & & & \\ 50 & 51 & 52 & 60 & 61 & 62 \\ 51 & 52 & 53 & 61 & 62 & 63 \\ 52 & 53 & 54 & 62 & 63 & 64 \end{bmatrix} \quad (5)$$

By observing the index matrix described in Eq. 5, we can identify multiple patterns.

Firstly, onto the vertical indexing (i.e., the index r in the index pair rc, for $M_I = [rc]_{R \times C}$). In the first column, r is arranged from 0 to $H_I - H_k + 1$, and each number repeats $W_I - W_k + 1$ times. This can be expressed in numpy notation as:

```
B, H, W = s.shape
KH, KW, KC = k.shape

r0 = np.arange(H-KH+1)
r0 = np.repeat(r0, W-KW+1)
r0 = r0.reshape(-1, 1)
```

Across the rows, r repeats itself W_k times, and then it assumes the value $r + 1$ and repeats itself W_k times once again. These two outer-most repetitions are due to the fact that the kernel has 2 rows. In a general case, we would observe H_k repetitions:

```
r1 = np.arange(KH).reshape(1, KH)
r = np.repeat(r0 + r1, KW, axis=1)
```

Notice that the addition between a column vector r_0 and a row vector r_1 will construct a matrix through broadcasting. The matrix R now contains the first index in M_I.

As for the horizontal indexing c, we observe the numbers are sequentially arranged in the first row from 0 to W_k, and that this sequence repeats H_k times:

```
c0 = np.arange(KW)
c0 = np.tile(c0, KH).reshape(1, -1)
```

Furthermore, c increases by 1 each row (as our convolution slides by exactly 1 step), going up to the number of horizontal slide steps of k onto I ($W_I - W_k + 1$). Adding this row vector to c_0 (a column vector) produces the index matrix for all horizontal slides of the kernel. Finally, we vertically tile (outer repeat) this matrix by the number of valid horizontal slide steps ($H_I - H_k + 1$):

```
c1 = np.arange(W-KW+1).reshape(-1, 1)
c = np.tile(c0 + c1, [H-KH+1, 1])
```

Hence, $M_I = [R, C]$.

3.3 Additional Considerations

Batching As M_I was constructed taking the width and height of the signals into consideration, it does not depend on the number of images, nor the number of kernels. Let I be redefined as a signal of shape $(B_1, B_2, \dots, B_n, H_I, W_I)$ (a batch of batches of ... batches of images), and k a signal of shape (H_k, W_k, C) This solution can be easily extended to a multi-image, multi-kernel scenario by broadcasting the index-mask selection of I to all batch-like dimensions and dotting it with the flatten kernels:

```
y = s[..., r, c] @ k.reshape(-1, C)
```

Padding As kernels of sizes greater than 0 slide across the spatial dimensions of an input signal I, they will occupy at most $(H_I - H_k + 1) \times (W_I - W_k + 1) < H_I W_I$ positions, resulting in an output signal smaller than the input signal. More specifically, of shape $(B_1, B_2, \dots, B_n, H_I - H_k + 1, W_I - W_k + 1, C)$. One can imagine many cases in which the maintenance of the signal size is desirable, such as maintaining locality between multiple applications of the convolution; or maintaining visualization consistency.

In order to maintain a consistent signal shape, I implemented the flag `padding='SAME'`, what is sometimes referred to as *zero-padding*. I employed here a strategy similar to what is done in the NumPy and TensorFlow libraries: (H_k-1, W_k-1) zeros are added to the input signal's height and width respectively. During the convolution, the spatial dimensions of the input signal become $((H_I+H_k-1)-H_k+1, (W_I+W_k-1)-W_k+1) = (H_I, W_I)$.
For kernels with an odd numbered width and height, this operation becomes trivial: we add $\lfloor H_k/2 \rfloor$ rows to both top and bottom extremities of the signal, and $\lfloor W_k/2 \rfloor$ columns to its left and right extremities.
A particular case must be handled when one of the sizes of the kernel is even: adding $\lfloor H_k/2 \rfloor$ will result in an output signal larger than the input signal by exactly 1 pixel. I handled this case in the same manner NumPy seemly does, by adding more padding to the bottom/right than to the top/left.

3.4 Complete Implementation, Usage Conditions and Limitations

Lst. 6 presents the fundamental parts of my implementation of the 2-D cross-correlation and convolution functions.

```
1  def correlate2d(s, k, padding='VALID'):
2    B, H, W = s.shape
3    KH, KW, KC = k.shape
4
5    if padding == 'SAME':
6      pt, pb = floor((KH-1)/2), ceil((KH-1)/2)
7      pl, pr = floor((KW-1)/2), ceil((KW-1)/2)
8      p = ((0,0), (pt, pb), (pl, pr))
9      s = np.pad(s, p)
10     B, H, W = s.shape
11
12   r0 = np.arange(H-KH+1)
13   r0 = np.repeat(r0, W-KW+1)
14   r0 = r0.reshape(-1, 1)
15   r1 = np.arange(KH).reshape(1, KH)
16   r = np.repeat(r0 + r1, KW, axis=1)
17
18   c0 = np.arange(KW)
19   c0 = np.tile(c0, KH).reshape(1, -1)
20   c1 = np.arange(W-KW+1).reshape(-1, 1)
21   c = np.tile(c0 + c1, [H-KH+1, 1])
22
23   y = s[..., r, c] @ k.reshape(-1, KC)
24   y = y.reshape(B, H-KH+1, W-KW+1, KC)
25
26   return y.clip(0., 255.)
27
28 def convolve2d(s, k, padding='VALID'):
29   k = np.rot90(k, k=2)
30   return correlate2d(s, k, padding)
```

Listing 6: Naive implementation of the Sepia filter.

The input signal (images) must be in the shape of $(B_1, B_2, \ldots, B_n, H_I, W_I)$ and kernels must be in the shape of (H_k, W_k, C). I also remark the following important limitations of this implementation:

1. This function is limited to the two dimensional spatial case, and will not work correctly for 3-D, 4-D and, more generally, n-D spatial signals, where $n > 3$.
2. Stride is always 1, making this function unsuitable for Atrous Convolution [3].

3.5 Validation

Sepia and gray-scale functions implemented here were validated against their naive counterparts. The convolution and cross-correlation results are compared with the ones obtained from `scipy.signal.convolve2d` and `scipy.signal.correlate2d` functions. The test consists of obtaining the exact same result in both implementations for a random input, up to the 6th decimal precision. These tests can be reproduced by executing the unit test component in the project with the command `pytest`.

3.6 Results and Discussions

Fig. 5 illustrates the result of the convolution of multiple input images with each kernel. A brief description of each result is provided below.

H1 highlights regions containing vertical lines, in which the left side represents areas with higher activation intensity, while the ones in the right are dark regions.

H2 is similar to *H1*, but activates strongly over horizontal lines in which the top is brighter than the bottom.

H3 responds strongly to bright regions that are surrounded by dark regions in the input. It seems to extract fine edges regardless of their orientation.

H4 is the uniform blur filter. It runs across the image macro-averaging the pixel activation intensities, reducing their differences.

H5 detects diagonal (bottom-left to top-right) lines — this kernel is symmetric, hence the convolution results in the same signal as the correlation.

H6 similar to H5, but detects diagonal (top-left to bottom-right) lines.

H7 responds to regions with bright top-right corners and dark bottom-right corners, without taking into consideration the information in the in-between section (most of which is multiplied by 0).

H8 is an edge-detector like H3, but seemly results in more detailed edges for these highly-detailed input images.

H9 is the Gaussian blur filter. It weight-averages the differences between the intensities of the pixels giving the central region more importance.

H10 combines *H1* and *H2* into one signal that seems to respond to both vertical and horizontal lines in the same intensity.

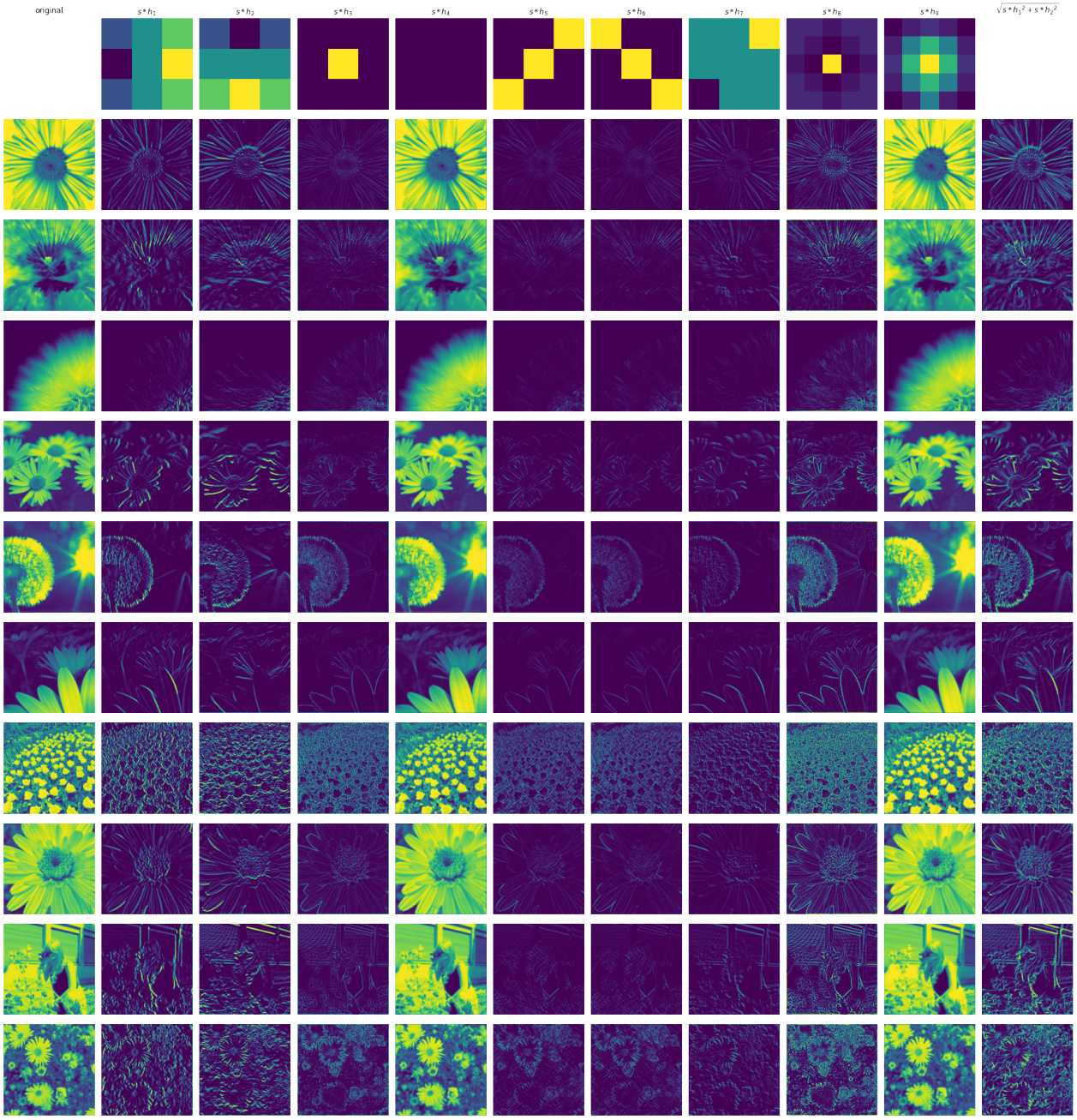

Figure 5: Input images (first column), kernels (first row) and output signal of the convolution between each image and kernel.

References

[1] Martıén Abadi et al. “Tensorflow: A system for large-scale machine learning”. In: *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 2016, pp. 265–283.

[2] Tiago Carneiro et al. “Performance analysis of google colaboratory as a tool for accelerating deep learning applications”. In: *IEEE Access* 6 (2018), pp. 61677–61685.

[3] Liang-Chieh Chen et al. “Rethinking atrous convolution for semantic image segmentation”. In: *arXiv preprint arXiv:1706.05587* (2017).

[4] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. “The NumPy array: a structure for efficient numerical computation”. In: *Computing in science & engineering* 13.2 (2011), pp. 22–30.