1. What exactly is a feature?

There is no universal or exact definition of what constitutes a feature, and the exact definition often depends on the problem or the type of application. Nevertheless, a feature is typically defined as an "interesting" part of an [image](https://en.wikipedia.org/wiki/Digital_image), and features are used as a starting point for many computer vision algorithms.

Since features are used as the starting point and main primitives for subsequent algorithms, the overall algorithm will often only be as good as its feature detector. Consequently, the desirable property for a feature detector is [repeatability](https://en.wikipedia.org/wiki/Repeatability): whether or not the same feature will be detected in two or more different images of the same scene.

Feature detection is a low-level [image processing](https://en.wikipedia.org/wiki/Image_processing) operation. That is, it is usually performed as the first operation on an image, and examines every [pixel](https://en.wikipedia.org/wiki/Pixel) to see if there is a feature present at that pixel. If this is part of a larger algorithm, then the algorithm will typically only examine the image in the region of the features. As a built-in pre-requisite to feature detection, the input image is usually smoothed by a [Gaussian](https://en.wikipedia.org/wiki/Gaussian_blur) kernel in a [scale-space representation](https://en.wikipedia.org/wiki/Scale_space) and one or several feature images are computed, often expressed in terms of local [image derivative](https://en.wikipedia.org/wiki/Image_derivative) operations.

2. For a top edge detector, write out the convolutional kernel matrix.

In [image processing](https://en.wikipedia.org/wiki/Image_processing), a kernel, convolution matrix, or mask is a small [matrix](https://en.wikipedia.org/wiki/Matrix_(mathematics)) used for blurring, sharpening, embossing, [edge detection](https://en.wikipedia.org/wiki/Edge_detection), and more. This is accomplished by doing a [convolution](https://en.wikipedia.org/wiki/Kernel_(image_processing)#Convolution) between the kernel and an [image](https://en.wikipedia.org/wiki/Bitmap_image).

The general expression of a convolution is

{\displaystyle g(x,y)=\omega \*f(x,y)=\sum \_{dx=-a}^{a}{\sum \_{dy=-b}^{b}{\omega (dx,dy)f(x-dx,y-dy)}},}

where {\displaystyle g(x,y)} is the filtered image, {\displaystyle f(x,y)} is the original image, {\displaystyle \omega } is the filter kernel. Every element of the filter kernel is considered by {\displaystyle -a\leq dx\leq a} and {\displaystyle -b\leq dy\leq b}.

Depending on the element values, a kernel can cause a wide range of effects. .

|  |  |  |
| --- | --- | --- |
| Operation | Kernel ω | Image result g(x,y) |
| [Identity](https://en.wikipedia.org/wiki/Identity_operation) | {\displaystyle {\begin{bmatrix}\ \ 0&\ \ 0&\ \ 0\\\ \ 0&\ \ 1&\ \ 0\\\ \ 0&\ \ 0&\ \ 0\end{bmatrix}}} |  |
| [Ridge detection](https://en.wikipedia.org/wiki/Ridge_detection) | {\displaystyle {\begin{bmatrix}-1&-1&-1\\-1&\ \ 4&-1\\-1&-1&-1\end{bmatrix}}} |  |
| {\displaystyle {\begin{bmatrix}-1&-1&-1\\-1&\ \ 8&-1\\-1&-1&-1\end{bmatrix}}} |  |
| Sharpen | {\displaystyle {\begin{bmatrix}\ \ 0&-1&\ \ 0\\-1&\ \ 5&-1\\\ \ 0&-1&\ \ 0\end{bmatrix}}} |  |
| [Box blur](https://en.wikipedia.org/wiki/Box_blur) ([normalized](https://en.wikipedia.org/wiki/Normalization_(image_processing))) | {\displaystyle {\frac {1}{9}}{\begin{bmatrix}\ \ 1&\ \ 1&\ \ 1\\\ \ 1&\ \ 1&\ \ 1\\\ \ 1&\ \ 1&\ \ 1\end{bmatrix}}} |  |
| [Gaussian blur](https://en.wikipedia.org/wiki/Gaussian_blur) 3 × 3 (approximation) | {\displaystyle {\frac {1}{16}}{\begin{bmatrix}\ \ 1&\ \ 2&\ \ 1\\\ \ 2&\ \ 4&\ \ 2\\\ \ 1&\ \ 2&\ \ 1\end{bmatrix}}} |  |
| [Gaussian blur](https://en.wikipedia.org/wiki/Gaussian_blur) 5 × 5 (approximation) | {\displaystyle {\frac {1}{256}}{\begin{bmatrix}1&4&6&4&1\\4&16&24&16&4\\6&24&36&24&6\\4&16&24&16&4\\1&4&6&4&1\end{bmatrix}}} |  |
| [Unsharp masking](https://en.wikipedia.org/wiki/Unsharp_masking) 5 × 5 Based on Gaussian blur with amount as 1 and threshold as 0 (with no [image mask](https://en.wikipedia.org/wiki/Image_mask)) | {\displaystyle {\frac {-1}{256}}{\begin{bmatrix}1&4&\ \ 6&4&1\\4&16&\ \ 24&16&4\\6&24&-476&24&6\\4&16&\ \ 24&16&4\\1&4&\ \ 6&4&1\end{bmatrix}}} |  |

The above are just a few examples of effects achievable by convolving kernels and images.

Origin[[edit](https://en.wikipedia.org/w/index.php?title=Kernel_(image_processing)&action=edit&section=2)]

The origin is the position of the kernel which is above (conceptually) the current output pixel. This could be outside of the actual kernel, though usually it corresponds to one of the kernel elements. For a symmetric kernel, the origin is usually the center element.

Convolution[[edit](https://en.wikipedia.org/w/index.php?title=Kernel_(image_processing)&action=edit&section=3)]

See also: [Symmetric convolution](https://en.wikipedia.org/wiki/Symmetric_convolution)

2D Convolution Animation

Convolution is the process of adding each element of the image to its local neighbors, weighted by the kernel. This is related to a form of [mathematical convolution](https://en.wikipedia.org/wiki/Convolution). [The matrix operation](https://en.wikipedia.org/wiki/Hadamard_product_(matrices)) being performed—convolution—is not traditional matrix multiplication, despite being similarly denoted by \*.

For example, if we have two three-by-three matrices, the first a kernel, and the second an image piece, convolution is the process of flipping both the rows and columns of the kernel and multiplying locally similar entries and summing. The element at coordinates [2, 2] (that is, the central element) of the resulting image would be a weighted combination of all the entries of the image matrix, with weights given by the kernel:

{\displaystyle \left({\begin{bmatrix}a&b&c\\d&e&f\\g&h&i\end{bmatrix}}\*{\begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix}}\right)[2,2]=(i\cdot 1)+(h\cdot 2)+(g\cdot 3)+(f\cdot 4)+(e\cdot 5)+(d\cdot 6)+(c\cdot 7)+(b\cdot 8)+(a\cdot 9).}

The other entries would be similarly weighted, where we position the center of the kernel on each of the boundary points of the image, and compute a weighted sum.

The values of a given pixel in the output image are calculated by multiplying each kernel value by the corresponding input image pixel values. This can be described algorithmically with the following pseudo-code:

for each image row in input image:

for each pixel in image row:

set accumulator to zero

for each kernel row in kernel:

for each element in kernel row:

if element position corresponding\* to pixel position then

multiply element value corresponding\* to pixel value

add result to accumulator

endif

set output image pixel to accumulator

\*corresponding input image pixels are found relative to the kernel's origin.

If the kernel is symmetric then place the center (origin) of the kernel on the current pixel. The kernel will overlap the neighboring pixels around the origin. Each kernel element should be multiplied with the pixel value it overlaps with and all of the obtained values should be summed. This resultant sum will be the new value for the current pixel currently overlapped with the center of the kernel.

If the kernel is not symmetric, it has to be flipped both around its horizontal and vertical axis before calculating the convolution as above.[[1]](https://en.wikipedia.org/wiki/Kernel_(image_processing)#cite_note-1)

The general form for matrix convolution is

{\displaystyle {\begin{bmatrix}x\_{11}&x\_{12}&\cdots &x\_{1n}\\x\_{21}&x\_{22}&\cdots &x\_{2n}\\\vdots &\vdots &\ddots &\vdots \\x\_{m1}&x\_{m2}&\cdots &x\_{mn}\\\end{bmatrix}}\*{\begin{bmatrix}y\_{11}&y\_{12}&\cdots &y\_{1n}\\y\_{21}&y\_{22}&\cdots &y\_{2n}\\\vdots &\vdots &\ddots &\vdots \\y\_{m1}&y\_{m2}&\cdots &y\_{mn}\\\end{bmatrix}}=\sum \_{i=0}^{m-1}\sum \_{j=0}^{n-1}x\_{(m-i)(n-j)}y\_{(1+i)(1+j)}}

Edge Handling[[edit](https://en.wikipedia.org/w/index.php?title=Kernel_(image_processing)&action=edit&section=4)]

Extend Edge-Handling

Kernel convolution usually requires values from pixels outside of the image boundaries. There are a variety of methods for handling image edges.

Extend

The nearest border pixels are conceptually extended as far as necessary to provide values for the convolution. Corner pixels are extended in 90° wedges. Other edge pixels are extended in lines.

Wrap

The image is conceptually wrapped (or tiled) and values are taken from the opposite edge or corner.

Mirror

The image is conceptually mirrored at the edges. For example, attempting to read a pixel 3 units outside an edge reads one 3 units inside the edge instead.

Crop / Avoid overlap

Any pixel in the output image which would require values from beyond the edge is skipped. This method can result in the output image being slightly smaller, with the edges having been cropped. Move kernel so that values from outside of image is never required. Machine learning mainly uses this approach. Example: Kernel size 10x10, image size 32x32, result image is 23x23.

Kernel Crop

Any pixel in the kernel that extends past the input image isn't used and the normalizing is adjusted to compensate.

Constant

Use constant value for pixels outside of image. Usually black or sometimes gray is used. Generally this depends on application.

Normalization[[edit](https://en.wikipedia.org/w/index.php?title=Kernel_(image_processing)&action=edit&section=5)]

Normalization is defined as the division of each element in the kernel by the sum of all kernel elements, so that the sum of the elements of a normalized kernel is unity. This will ensure the average pixel in the modified image is as bright as the average pixel in the original image.

Optimisation[[edit](https://en.wikipedia.org/w/index.php?title=Kernel_(image_processing)&action=edit&section=6)]

Fast convolution algorithms include: [[2]](https://en.wikipedia.org/wiki/Kernel_(image_processing)#cite_note-2)

separable convolution

separable convolution[[edit](https://en.wikipedia.org/w/index.php?title=Kernel_(image_processing)&action=edit&section=7)]

2D convolution with M×N kernel requires M×N multiplications for each sample ( pixel). If the kernel is separable, then the computation can be reduced to M + N multiplications. Using separable convolutions can significantly decrease the computation, do 1D convolution twice instead of one 2D convolution.[[3]](https://en.wikipedia.org/wiki/Kernel_(image_processing)#cite_note-3)

Implementation[[edit](https://en.wikipedia.org/w/index.php?title=Kernel_(image_processing)&action=edit&section=8)]

Here a concrete convolution implementation done with the [GLSL](https://en.wikipedia.org/wiki/OpenGL_Shading_Language) shading language :

// author : csblo

// Work made just by consulting :

// https://en.wikipedia.org/wiki/Kernel\_(image\_processing)

// Define kernels

#define identity mat3(0, 0, 0, 0, 1, 0, 0, 0, 0)

#define edge0 mat3(1, 0, -1, 0, 0, 0, -1, 0, 1)

#define edge1 mat3(0, 1, 0, 1, -4, 1, 0, 1, 0)

#define edge2 mat3(-1, -1, -1, -1, 8, -1, -1, -1, -1)

#define sharpen mat3(0, -1, 0, -1, 5, -1, 0, -1, 0)

#define box\_blur mat3(1, 1, 1, 1, 1, 1, 1, 1, 1) \* 0.1111

#define gaussian\_blur mat3(1, 2, 1, 2, 4, 2, 1, 2, 1) \* 0.0625

#define emboss mat3(-2, -1, 0, -1, 1, 1, 0, 1, 2)

// Find coordinate of matrix element from index

vec2 kpos(int index)

{

return vec2[9] (

vec2(-1, -1), vec2(0, -1), vec2(1, -1),

vec2(-1, 0), vec2(0, 0), vec2(1, 0),

vec2(-1, 1), vec2(0, 1), vec2(1, 1)

)[index] / iResolution.xy;

}

// Extract region of dimension 3x3 from sampler centered in uv

// sampler : texture sampler

// uv : current coordinates on sampler

// return : an array of mat3, each index corresponding with a color channel

mat3[3] region3x3(sampler2D sampler, vec2 uv)

{

// Create each pixels for region

vec4[9] region;

for (int i = 0; i < 9; i++)

region[i] = texture(sampler, uv + kpos(i));

// Create 3x3 region with 3 color channels (red, green, blue)

mat3[3] mRegion;

for (int i = 0; i < 3; i++)

mRegion[i] = mat3(

region[0][i], region[1][i], region[2][i],

region[3][i], region[4][i], region[5][i],

region[6][i], region[7][i], region[8][i]

);

return mRegion;

}

// Convolve a texture with kernel

// kernel : kernel used for convolution

// sampler : texture sampler

// uv : current coordinates on sampler

vec3 convolution(mat3 kernel, sampler2D sampler, vec2 uv)

{

vec3 fragment;

// Extract a 3x3 region centered in uv

mat3[3] region = region3x3(sampler, uv);

// for each color channel of region

for (int i = 0; i < 3; i++)

{

// get region channel

mat3 rc = region[i];

// component wise multiplication of kernel by region channel

mat3 c = matrixCompMult(kernel, rc);

// add each component of matrix

float r = c[0][0] + c[1][0] + c[2][0]

+ c[0][1] + c[1][1] + c[2][1]

+ c[0][2] + c[1][2] + c[2][2];

// for fragment at channel i, set result

fragment[i] = r;

}

return fragment;

}

void mainImage( out vec4 fragColor, in vec2 fragCoord )

{

// Normalized pixel coordinates (from 0 to 1)

vec2 uv = fragCoord/iResolution.xy;

// Convolve kernel with texture

vec3 col = convolution(emboss, iChannel0, uv);

// Output to screen

fragColor = vec4(col, 1.0);

}

3. Describe the mathematical operation that a 3x3 kernel performs on a single pixel in an image.

The use of [Kernels](http://en.wikipedia.org/wiki/Kernel_%28image_processing%29) - also known as convolution matrices or masks - is invaluable to image processing. Techniques such as blurring, edge detection, and sharpening all rely on kernels - small matrices of numbers - to be applied across an image in order to process the image as a whole.

So what is a kernel? In image processing a Kernel is simply a 2-dimensional matrix of numbers. While this matrix can range in dimensions, for simplicity this article will stick to 3x3 dimensional kernels. An example of a kernel is shown below:

|  |  |  |
| --- | --- | --- |
| 0.111 | 0.111 | 0.111 |
| 0.111 | 0.111 | 0.111 |
| 0.111 | 0.111 | 0.111 |

A 3x3 symmetrical Kernel, or convolution matrix.

How does this matrix relate to image processing? An image is just a 2-dimensional matrix of numbers, or [pixels](http://en.wikipedia.org/wiki/Pixel). Each pixel is represented by a number - depending upon the image format these numbers can vary: for an 8 bit RGB image each pixel has a red, green, and blue component with a value ranging from 0 to 255. A kernel works by operating on these pixel values using straightforward mathematics to construct a new image. Lets take the above kernel and do some math: for each pixel, center the kernel over the pixel, multiply the kernel values times the corresponding pixel values, and add the result - this final value is the new value of the current pixel.

|  |  |  |
| --- | --- | --- |
| 0.111 | 0.111 | 0.111 |
| 0.111 | 0.111 | 0.111 |
| 0.111 | 0.111 | 0.111 |

X

|  |  |  |
| --- | --- | --- |
| 10 | 20 | 13 |
| 19 | 25 | 16 |
| 22 | 26 | 21 |

=

|  |  |  |
| --- | --- | --- |
| 0.11 \* 10 = 1 | 0.11 \* 20 = 2 | 0.11 \* 13 = 1 |
| 0.11 \* 19 = 2 | 0.11 \* 25 = 3 | 0.11 \* 16 = 2 |
| 0.11 \* 22 = 2 | 0.11 \* 26 = 3 | 0.11 \* 21 = 2 |

= 1 + 2 + 1 + 2 + 3 + 2 + 2 + 3 + 2 = 18

An example kernel operation.

As each pixel is processed, a new image emerges based upon the calculated values. The new image is highly dependent upon the kernel used - each kernel has specific properties depending upon its values. Take the kernel demonstrated above: the mathematics of this matrix results in a value that is the average of all pixels in a 3x3 pixel grid. In short - each pixel is the average of its neighbors - this results in a blurred image.

Unmodified (left) and the same image processed with a local average blur kernel.

What other types of kernels are there?

Edge detection: this kernel detects edges within an image. A 3x3 example:

|  |  |  |
| --- | --- | --- |
| 0 | -1 | 0 |
| -1 | 4 | -1 |
| 0 | -1 | 0 |

Notice that if all pixel values are comparable, the the resultant pixel value will be close to 0. However, edges - locations with extreme differences in pixel values - will result in values far from zero.

Gaussian Blur: This kernel is similar to the blur kernel presented above, but is different in that it is dependent upon the [Gaussian function](http://en.wikipedia.org/wiki/Gaussian_function) - a function which creates a distribution of values around the center point. This results in a kernel in which pixels near the center contribute more towards the new pixel value than those further away.

Sharpening: This kernel sharpens an image - accentuating the edges of the image. Sharpening an image add contrast to edges, and a 3x3 version of this mask is similar to the edge detection kernel with a center value of 5. This adds contrast around an edge by accentuating bright and dark areas.

Unsharp Mask: Used to sharpen an image, this technique is based upon first creating a gaussian blurred copy of the image. This blurred copy is then subtracted from the original - pixels above a given threshold are sharpened by enhancing light and dark pixels.

Of course we are not restricted to 3x3 kernels - this was only done for simplicity. Kernels can be of just about any size. More sophisticated kernels are typically larger, in fact many image processing software packages have options to customize a kernel. For instance, Adobe Photoshop has a custom filter option to allow a user to enter their own kernel values (Filter->Other->Custom):

The custom filter dialog in Adobe Photoshop showing an edge detection kernel.

Of course customizing a kernel in this manner can be a time consuming, trial and error process. However this technique provides a great deal of flexibility in creating new ways to process an image, or fine-tuning older well established workflows.

4. What is the significance of a convolutional kernel added to a 3x3 matrix of zeroes?

Here is a mathematician's domain. Most of filters are using convolution matrix. With the Convolution Matrix filter, if the fancy takes you, you can build a custom filter.

What is a convolution matrix? It's possible to get a rough idea of it without using mathematical tools that only a few ones know. Convolution is the treatment of a matrix by another one which is called “kernel”.

The Convolution Matrix filter uses a first matrix which is the Image to be treated. The image is a bi-dimensional collection of pixels in rectangular coordinates. The used kernel depends on the effect you want.

GIMP uses 5x5 or 3x3 matrices. We will consider only 3x3 matrices, they are the most used and they are enough for all effects you want. If all border values of a kernel are set to zero, then system will consider it as a 3x3 matrix.

The filter studies successively every pixel of the image. For each of them, which we will call the “initial pixel”, it multiplies the value of this pixel and values of the 8 surrounding pixels by the kernel corresponding value. Then it adds the results, and the initial pixel is set to this final result value.

A simple example

On the left is the image matrix: each pixel is marked with its value. The initial pixel has a red border. The kernel action area has a green border. In the middle is the kernel and, on the right is the convolution result.

Here is what happened: the filter read successively, from left to right and from top to bottom, all the pixels of the kernel action area. It multiplied the value of each of them by the kernel corresponding value and added results. The initial pixel has become 42: (40\*0)+(42\*1)+(46\*0) + (46\*0)+(50\*0)+(55\*0) + (52\*0)+(56\*0)+(58\*0) = 42. (the filter doesn't work on the image but on a copy). As a graphical result, the initial pixel moved a pixel downwards.

8.2.2. Activating the filter

This filter is found in the image window menu under Filters → Generic → Convolution Matrix….

8.2.3. Options

Figure 17.150. “Convolution matrix” options

Matrix

This is the 5x5 kernel matrix: you enter wanted values directly into boxes.

Divisor

The result of previous calculation will be divided by this divisor. You will hardly use 1, which lets result unchanged, and 9 or 25 according to matrix size, which gives the average of pixel values.

Offset

This value is added to the division result. This is useful if result may be negative. This offset may be negative.

Border

When the initial pixel is on a border, a part of kernel is out of image. You have to decide what filter must do:

From left: source image, Extend border, Wrap border, Crop border

Extend

This part of kernel is not taken into account.

Wrap

This part of kernel will study pixels of the opposite border, so pixels disappearing from one side reappear on the other side.

Crop

Pixels on borders are not modified, but they are cropped.

Channels

You can select there one or several channels the filter will work with.

Normalise

If this option is checked, The Divisor takes the result value of convolution. If this result is equal to zero (it's not possible to divide by zero), then a 128 offset is applied. If it is negative (a negative color is not possible), a 255 offset is applied (inverts result).

Alpha-weighting

If this option is not checked, the filter doesn't take in account transparency and this may be cause of some artefacts when blurring.

5. What exactly is padding?

Padding is the space that's inside the element between the element and the border. Padding goes around all four sides of the content and you can target and change the padding for each side (just like a margin).

6. What is the concept of stride?

Stride is a component of [convolutional neural networks](https://deepai.org/machine-learning-glossary-and-terms/convolutional-neural-network), or [neural networks](https://deepai.org/machine-learning-glossary-and-terms/neural-network) tuned for the compression of images and video data. Stride is a parameter of the neural network's filter that modifies the amount of movement over the image or video. For example, if a neural network's stride is set to 1, the filter will move one pixel, or unit,  at a time. The size of the filter affects the encoded output volume, so stride is often set to a whole integer, rather than a fraction or decimal.

How does Stride work?

[Source](https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/)

Imagine a convolutional neural network is taking an image and analyzing the content. If the filter size is 3x3 pixels, the contained nine pixels will be converted down to 1 pixel in the output layer. Naturally, as the stride, or movement, is increased, the resulting output will be smaller. Stride is a parameter that works in conjunction with [padding](https://deepai.org/machine-learning-glossary-and-terms/padding), the feature that adds blank, or empty pixels to the frame of the image to allow for a minimized reduction of size in the output layer. Roughly, it is a way of increasing the size of an image, to counteract the fact that stride reduces the size. Padding and stride are the foundational parameters of any convolutional neural network.

7. What are the shapes of PyTorch's 2D convolution's input and weight parameters?

Two-dimensional convolution is applied over an input given by the user where the specific shape of the input is given in the form of size, length, width, channels, and hence the output must be in a convoluted manner is called PyTorch Conv2d. Conv2d is the function to do any changes in the convolution of two-dimensional data and it mainly pertains to an image in the system where we can apply regularizations too.

What is PyTorch Conv2d?

A convolution operation is performed on the 2D matrix provided in the system where any operations on a matrix such as matrix inversion or MAC operation is carried out with Conv2d function in the PyTorch module. This belongs to torch.nn package where all the neural networks functions are available thus managing the tensors and convolutions of matrices. An image is modified and made into two where the product of these two must help in reporting the value in the output.

How to Use Conv2d?

We are adding Conv2d to the layers of the neural network and in PyTorch, it is an instance of the nn module. These layers become the first layers in the network where parameters are very much necessary. A number of channels of the input data to be declared in the parameters along with the number of feature maps in the above layer. Instead of a number of channels, we can use the number of feature maps as input which has to be generated in the output. It is simple as we have to import all the required libraries along with the number of strides that must be given in the code. The syntax must be like this.  
a = Conv2d(in\_channels, out\_channels, kernel\_size=(n, n), stride, padding, bias)

PyTorch conv2d – Parameters

The following parameters are used in PyTorch Conv2d. in\_channels are used to describe how many channels are present in the input image whereas out\_channels are used to describe the number of channels present after convolution happened in the system. The breadth and height of the filter is provided by the kernel. The field stride explains the stride of the convolution that happens in the system. The amount of implicit paddings is controlled by the field padding where a number of points are explained in each dimension.

If there is a bias happens in the result and if the user knows it beforehand, it is better to give in the code beforehand using the field bias. The output we receive from the Conv2d is always a tensor. Padding\_mode is another field that explains how padding happens in the code and the default value happens to be zero. The dilation field explains the space between kernel elements and the default value is 1. Groups can be used if there are any values blocked from input to output and hence it does not appear in the output.

PyTorch Conv2d Example

The first step is to import the torch libraries into the system.

import torch  
import torch. nn as nn

Conv2d instance must be created where the value and stride of the parameter have to be passed in the system. These values are then applied to the input generated data.

a = nn.Conv2d(4, 16, 6, stride=2)  
input\_data = torch.randn(15, 20, 48, 48)  
output\_value = a(input)

When we use square kernels, the code must be like this.

a = nn.Conv2d(2, 22, 2, stride=2)

when the kernels are not squared, the code should be similar to the following.

a = nn.Conv2d(2, 22, (2, 3), stride=(2, 1), padding=(4, 2))  
a = nn.Conv2d(2, 22, (2, 3), stride=(2, 1), padding=(4, 2) , dilation=(3, 1))

We can build a neural network using Conv2d layer. First, we have to load the libraries in the system.

import torch  
from torch. autograd import Variable  
import torchvision. datasets as dsets  
import torchvision. transforms as transforms  
import torch.nn.init

The next step is to set the parameters.

Batch\_lot = 16  
Prob\_k = 1

We are considering MNIST dataset for training and testing here.

mnist\_train = dsets.MNIST(root='MNIST\_data/',  
train=True,  
transform=transforms.ToTensor(),  
download=True)  
mnist\_test = dsets.MNIST(root='MNIST\_data/',  
train=False,  
transform=transforms.ToTensor(),  
download=True)  
data\_loader = torch.utils.data.DataLoader(dataset=mnist\_train,  
batch\_lot=batch\_lot,  
shuffle=True)

We are identifying the dataset.

print('The training dataset:\t',mnist\_train)  
print('\nThe testing dataset:\t',mnist\_test)  
Now we are about to create CNN model based on Conv2d layer.  
Class neural(torch.nn.Module):  
def \_\_init\_\_(self):  
super(neural, self).\_\_init\_\_()  
self.layer1 = torch.nn.Sequential(  
torch.nn.Conv2d(1, 32, kernel\_size=3, stride=1, padding=1),  
torch.nn.ReLU(),  
torch.nn.MaxPool2d(kernel\_size=2, stride=2),  
torch.nn.Dropout(p=1 - keep\_prob))  
self.layer2 = torch.nn.Sequential(  
torch.nn.Conv2d(32, 64, kernel\_size=3, stride=1, padding=1),  
torch.nn.ReLU(),  
torch.nn.MaxPool2d(kernel\_size=2, stride=2),  
torch.nn.Dropout(p=1 - keep\_prob))  
self.layer3 = torch.nn.Sequential(  
torch.nn.Conv2d(64, 128, kernel\_size=3, stride=1, padding=1),  
torch.nn.ReLU(),  
torch.nn.MaxPool2d(kernel\_size=2, stride=2, padding=1),  
torch.nn.Dropout(p=1 - keep\_prob))  
torch.nn.init.xavier\_uniform(self.fc1.weight)  
self.layer4 = torch.nn.Sequential(  
self.fc1,  
torch.nn.ReLU(),  
torch.nn.Dropout(p=1 - keep\_prob))  
self.fc2 = torch.nn.Linear(625, 10, bias=True)  
torch.nn.init.xavier\_uniform\_(self.fc2.weight)  
def forward(self, x):  
out = self.layer1(x)  
out = self.layer2(out)  
out = self.layer3(out)  
out = out.view(out.size(0), -1)  
out = self.fc1(out)  
out = self.fc2(out)  
return out  
model = neural()  
model  
we must set the learning rate, optimizer and loss function.  
lrng\_rate = 0.01  
criterion = torch.nn.CrossEntropyLoss()  
optimzr = torch.optim.Adam(params=model.parameters(), lr=lrng\_rate)  
We must train the model for any epochs.  
print('Training the Deep Learning network ...')  
train\_cost = [] train\_accu = [] training\_epochs = 10  
total\_batch = len(mnist\_train) // batch\_size  
print('Size of the training dataset is {}'.format(mnist\_train.data.size()))  
print('Size of the testing dataset'.format(mnist\_test.data.size()))  
print('Batch size is : {}'.format(batch\_size))  
print('Total number of batches is : {0:2.0f}'.format(total\_batch))  
print('\nTotal number of epochs is : {0:2.0f}'.format(training\_epochs))  
for epoch in range(training\_epochs):  
avg\_cost = 0  
for i, (batch\_X, batch\_Y) in enumerate(data\_loader):  
X = Variable(batch\_X)  
Y = Variable(batch\_Y)  
optimizer.zero\_grad()  
hypothesis = model(X)  
cost = criterion(hypothesis, Y)  
cost.backward()  
optimizer.step()  
prediction = hypothesis.data.max(dim=1)[1] train\_accu.append(((prediction.data == Y.data).float().mean()).item())  
train\_cost.append(cost.item())  
if i % 200 == 0:  
print("Epoch= {},\t batch = {},\t cost = {:2.4f},\t accuracy = {}".format(epoch+1, i, train\_cost[-1], train\_accu[-1]))  
avg\_cost += cost.data / total\_batch  
print("[Epoch: {:>4}], averaged cost = {:>.9}".format(epoch + 1, avg\_cost.item()))  
print('Learning Finished!')  
we can visualize using matplotlib library.  
from matplotlib import pylab as plt  
import numpy as np  
plt.figure(figsize=(20,10))  
plt.subplot(121), plt.plot(np.arange(len(train\_cost)), train\_cost), plt.ylim([0,10])  
plt.subplot(122), plt.plot(np.arange(len(train\_accu)), 100 \* torch.as\_tensor(train\_accu).numpy()), plt.ylim([0,100])

8. What exactly is a channel?

There are three channels in an RGB image- red, green and blue. The color space where red, green and blue channels represent images is called RGB color space. In OpenCV, BGR sequence is used instead of RGB. This means the first channel is blue, the second channel is green, and the third channel is red.

9.Explain relationship between matrix multiplication and a convolution?

domain, you simply multiply the signal(X)-which is matrix with Signal(Y), which is also a matrix. So, now you will be able to understand that, Yes convolution is same as matrix multiplication(where matrix X and Y matrix of signal) but ONLY IN FREQUENCY DOMAIN.

convolutions can be mapped as matrix multiplication operations by flattening and rearranging the weights and input features. As illustrated in Figure 2, 64 × 3 kernels with a size of 3 × 3 are mapped to a rearranged matrix with dimensions of 64 × (3 × 3 × 3).