1. **What exactly is a feature?**

**In machine learning, features are individual independent variables that act like a input in your system. Actually, while making the predictions, models use such features to make the predictions. And using the feature engineering process, new features can also be obtained from old features in machine learning.**

**To understand in more simple way, lets take an example, where you can consider one column of your data set to be one feature which is also know as “variables or attributes” and the more number of features are known as dimensions. And depending on what you are trying to analyze the features you include in your dataset can vary widely.**

**What is Feature Engineering in Machine Learning?**

**Feature engineering is the process of using the domain knowledge of the data to create features that makes machine learning algorithms work properly. If feature engineering is performed properly, it helps to improve the power of prediction of machine learning algorithms by creating the features using the raw data that facilitate the machine learning process.**

**Why Feature is Important in Machine Learning?**

**Features in machine learning is very important, being building a blocks of datasets, the quality of the features in your dataset has major impact on the quality of the insights you will get while using the dataset for machine learning.**

**However, depending on the different business problems in different industries it is not necessary the features should be same features, so here you need to strongly understand the business goal of your data science project.**

**Where on the other hand, using the “feature selection” and “feature engineering” process you can improve the quality of your datasets’ features, which a very tedious and difficult process. It these techniques are working well, you will get optimal dataset with all of the important features, that bearing on your specific business problem leads to the best possible model development and the most beneficial visual perception.**

**Tops Methods of Feature Selection in ML:**

* **Universal Selection**
* **Feature Importance**
* **Correlation Matrix with Heatmap**

**Feature engineering is the most important part of machine leaning that makes difference between and good and bad model. And there are several steps involved in feature engineering and most preferred steps are given below.**

**Steps To Do Feature Engineering in ML:**

1. **Gathering Data**
2. **Cleaning DATA**
3. **Feature Engineering**
4. **Defining Model**
5. **Training & Testing of model prediction**

**To perform the feature engineering in machine learning you need data experts like data scientists or**[**hire machine learning engineer**](https://www.cogitotech.com/hire-machine-learning-engineer/)**who can understand and perform the feature engineering process with right instructions. Cogito is one the companies providing the hiring and recruitment services with outsourcing of data scientists and machine learning engineers for in-house AI development or for remote locations as per the requirements of various companies.**

**Training & Testing of model prediction**

**To perform the feature engineering in machine learning you need data experts like data scientists or hire machine learning engineer who can understand and perform the feature engineering process with right instructions. [Cogtio](https://www.cogitotech.com/" \t "_blank) is one the companies providing the hiring and recruitment services with outsourcing of data scientists and machine learning engineers for in-house AI development or for remote**[**locations**](https://bit.ly/3cOPGY7)**as per the requirements of various companies. Originally published at**[**Source**](https://cogitoai.home.blog/2019/07/15/what-are-features-in-machine-learning-and-why-it-is-important/)

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

**In a CNN, the convolutional kernel is a shared weight matrix, and is learned in a similar way to other weights. It is initialized in the same way, with small random values, and the weight deltas from back propagation are summed across all the features that receive its output (i.e. usually all "pixels" in the output of the convolutional layer)**

**A typical random kernel will perform a little like an edge detector.**

**After training, the first CNN layer can be displayed and will often have learned some kernels that can be interpreted if you are familiar with image processing**

**In short your answer is this: There is no need to look for correct kernels to use. Instead look for a CNN library where you set params such as number of convolutional layers, and research the way to view the kernels as they learn - most CNN libraries will have a documented way to visualise them.**

**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 |

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.

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):

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?**

**In image processing, a kernel, convolution matrix, or mask is a small matrix. It is used for blurring, sharpening, embossing, edge detection, and more. This is accomplished by doing a convolution between a kernel and an image.**

**In this article, here are some conventions that we are following —**

* **We are specifically referring to 2D convolutions that are usually applied on 2 matrix objects such as images. These concepts also apply for 1D and 3D convolutions, but may not correlate directly.**
* **While applying 2D convolutions like 3X3 convolutions on images, a 3X3 convolution filter, in general will always have a third dimension in size. This filter depends on (and is equal to) the number of channels of the input image. So, we apply a 3X3X1 convolution filter on gray-scale images (the number of channels = 1) whereas, we apply a 3X3X3 convolution filter on a colored image (the number of channels = 3).**
* **We will refer to all the convolutions by their first two dimensions, irrespective of the channels. (We are observing the assumption of zero padding).**

****

**A convolution filter passes over all the pixels of the image in such a manner that, at a given time, we take ‘dot product’ of the convolution filter and the image pixels to get one final value output. We do this hoping that the weights (or values) in the convolution filter, when multiplied with corresponding image pixels, gives us a value that best represents those image pixels. We can think of each convolution filter as extracting some kind of feature from the image.**

**Therefore, convolutions are done usually keeping these two things in mind -**

* **Most of the features in an image are usually local. Therefore, it makes sense to take few local pixels at once and apply convolutions.**
* **Most of the features may be found in more than one place in an image. This means that it makes sense to use a single kernel all over the image, hoping to extract that feature in different parts of the image.**

**Now that we have convolution filter sizes as one of the hyper-parameters to choose from. The choice is can be made between smaller or larger filter size.**

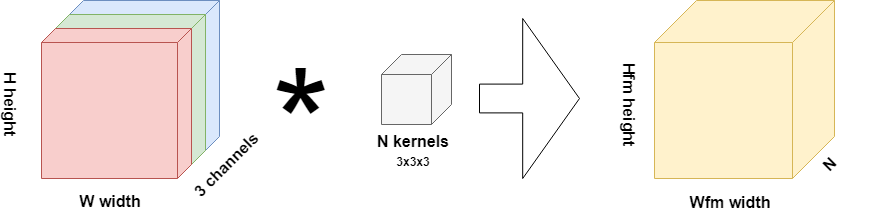
**5. What exactly is padding?**

**What is padding and why do we need it?**

Let's first take a look at what padding is. From this, it gets clear straight away why we might need it for training our neural network. More specifically, our *ConvNet*, because that's where you'll apply padding pretty much all of time time 😄

Now, in order to find out about how padding works, we need to study the internals of a convolutional layer first.

Here you've got one, although it's very generic:

[](https://www.machinecurve.com/wp-content/uploads/2019/09/CNN.png)

What you see on the left is an RGB input image - width [latex]W[/latex], height [latex]H[/latex] and three channels. Hence, this layer is likely the *first layer in your model*; in any other scenario, you'd have feature maps as the input to your layer.

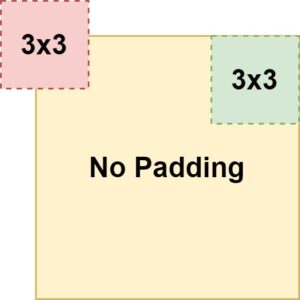
Now, what is a feature map? That's the yellow block in the image. It's a collection of [latex]N[/latex] one-dimensional "maps" that each represent a particular "feature" that the model has spotted within the image. This is why convolutional layers are known as feature extractors.

Now, this is very nice - but how do we get from input (whether image or feature map) to a feature map? This is through *kernels*, or *filters*, actually. These filters - you configure some number [latex]N[/latex] per convolutional layer - "slide" (strictly: convolve) over your input data, and have the same number of "channel" dimensions as your input data, but have much smaller widths and heights. For example, for the scenario above, a filter may be 3 x 3 pixels wide and high, but always has 3 channels as our input has 3 channels too.

Now, when they slide over the input - from left to right horizontally, then moving down vertically after a row has been fully captured - they perform *element-wise multiplications* between what's "currently under investigation" within the input data and the *weights present within the filter*. These weights are equal to the weights of a "classic" neural network, but are structured in a different way. Hence, optimization a ConvNet involves computing [a loss value](https://www.machinecurve.com/index.php/2019/10/04/about-loss-and-loss-functions/) for the model and subsequently using [an optimizer](https://www.machinecurve.com/index.php/2019/10/24/gradient-descent-and-its-variants/) to change the weights.

Through these weights, as you may guess, the model learns to detect the presence of particular features - which, once again, are represented by the feature maps. This closes the circle with respect to how a convolutional layer works :)

**Conv layers might induce spatial hierarchy**

[](https://www.machinecurve.com/wp-content/uploads/2020/02/pad-nopad-conv-1.jpg)

If the width and/or height of your kernels is [latex]> 1[/latex], you'll see that the width and height of the feature map being output gets smaller. This occurs due to the fact that the feature map slides over the input and computes the element-wise multiplications, but is too large in order to inspect the "edges" of the input. This is illustrated in the image to the right, where the "red" position is impossible to take and the "green" one is part of the path of the convolution operation.

As it cannot capture the edges, it won't be able to effectively "end" at the final position of your row, resulting in a smaller output width and/or height.

For example, take the model that we generated in our blog post ["Reducing trainable parameters with a Dense-free ConvNet classifier"](https://www.machinecurve.com/index.php/2020/01/31/reducing-trainable-parameters-with-a-dense-free-convnet-classifier/). In the model summary, you clearly see that the output shape gets smaller in terms of width and height. Primarily, this occurs due to [max pooling](https://www.machinecurve.com/index.php/2020/01/30/what-are-max-pooling-average-pooling-global-max-pooling-and-global-average-pooling/), but you also see that the second Conv2D layer impacts the width and height of the feature map (and indeed, also the *number* of maps, but this is not relevant for now).

Model: "GlobalAveragePoolingBased"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

conv2d (Conv2D) (None, 26, 26, 32) 320

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d (MaxPooling2D) (None, 13, 13, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout (Dropout) (None, 13, 13, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_1 (Conv2D) (None, 11, 11, 64) 18496

We call this *a spatial hierarchy.* Indeed, convolutional layers may cause a "hierarchy"-like flow of data through the model. Here, you have a schematic representation of a substantial hierarchy and a less substantial one - which is often considered to be *less efficient*:

**Padding avoids the loss of spatial dimensions**

Sometimes, however, you need to apply filters of a fixed size, but you *don't want to lose width and/or height dimensions in your feature maps*. For example, this is the case when you're [training an autoencoder](https://www.machinecurve.com/index.php/2019/12/20/building-an-image-denoiser-with-a-keras-autoencoder-neural-network/). You need the output images to be of the same size as the input, yet need an [activation function](https://www.machinecurve.com/index.php/2020/01/24/overview-of-activation-functions-for-neural-networks/) like e.g. [Sigmoid](https://www.machinecurve.com/index.php/2019/09/04/relu-sigmoid-and-tanh-todays-most-used-activation-functions/) in order to generate them.

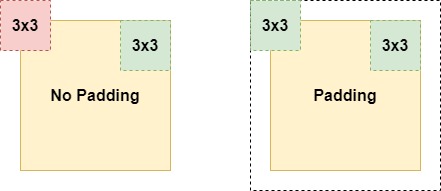
If you would do so with a Conv layer, this would become problematic, as you'd reduce the size of your feature maps - and hence would produce outputs unequal in size to your inputs.

That's not what we want when we create an autoencoder. We want the original output and the original output only ;-)

Padding helps you solve this problem. Applying it effectively adds "space" around your input data or your feature map - or, more precisely, "extra rows and columns" [with some instantiation] (Chollet, 2017).

[](https://www.machinecurve.com/wp-content/uploads/2020/02/pad-nopad.jpg)

The consequences of this fact are rather pleasurable, as we can see in the example below.

[](https://www.machinecurve.com/wp-content/uploads/2020/02/pad-nopad-conv.jpg)

Adding the "extra space" now allows us to capture the position we previously couldn't capture, and allows us to detect features in the "edges" of your input. This is great! 😊

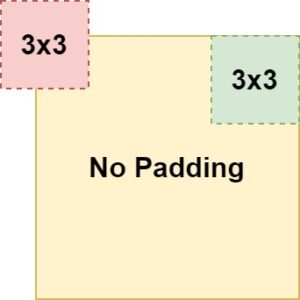
**Types of padding**

Now, unfortunately, padding is not a binary option - i.e., it cannot simply be turned on and off. Rather, you can choose which padding you use. Based on the Keras docs (Keras, n.d.) and PyTorch docs (PyTorch, n.d.), we'll cover these types of padding next:

* Valid padding (or no padding);
* Same padding;
* Causal padding;
* Constant padding;
* Reflection padding;
* Replication padding.

Please note that the discussion next doesn't contain any Python code. We'll cover the padding options in terms of code in a different blog post ;)

**Valid padding / no padding**

[](https://www.machinecurve.com/wp-content/uploads/2020/02/validpad.jpg)

Valid padding simply means "no padding" (Keras, n.d.).

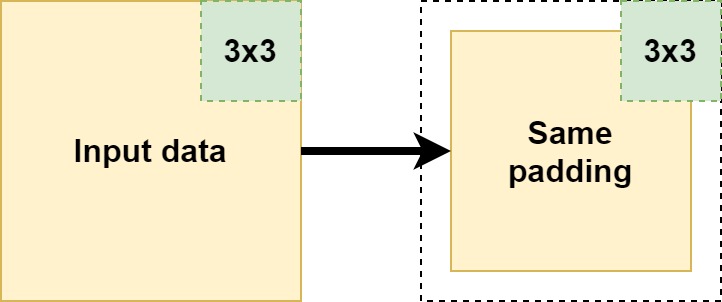
This equals the scenario to the right, where capturing the "edges" only is not possible.

It may seem strange to you that frameworks include an option for valid padding / no padding, as you could simply omit the padding as well. However, this is not strange at all: if you specify some padding attribute, there must be a default value. As it may be confusing to perform some padding operation if you didn't specify any, at least Keras chooses to set padding to 'valid' if none is provided. By consequence, you can also *specify it yourself*. A bit useless, but possible by design :)

**Same padding / zero padding**

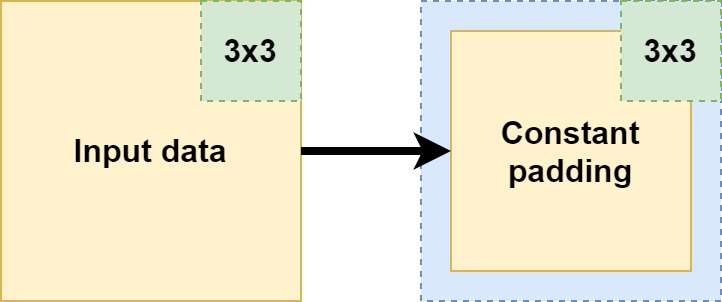
Another option would be "same padding", also known as "zero padding". Here, the padding ensures that the output has the same shape as the input data, as you can see in the image below (Keras, n.d.). It is achieved by adding "zeros" at the edges of your layer output, e.g. the white space on the right of the image.

Side note: in Keras, there is an inconsistency between backends (i.e., TensorFlow, Theano and CNTK) [as described here](https://github.com/keras-team/keras/pull/9473#issuecomment-372166860) (Keras, n.d.). However, with TensorFlow 2.0 being the "recommended choice" these days, this shouldn't be too much of a problem.

[](https://www.machinecurve.com/wp-content/uploads/2020/02/same-pad.jpg)

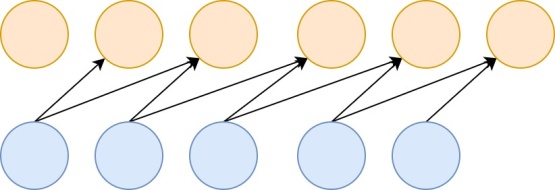
**Constant padding**

A type of padding that really resembles same padding is *constant padding*. Here, the outcome can be the same - the output will have the same shape as the input. However, rather than "zeros" - which is what same padding does - constant padding allows you to pad with a user-specified constant value (PyTorch, n.d.). In PyTorch, it is also possible to specify the padding at the boundary level (e.g. pad on the left and the top but not on the right and at the bottom). This obviously breaks with *same padding* covered earlier; be aware of this.

[](https://www.machinecurve.com/wp-content/uploads/2020/02/constantpad.jpg)

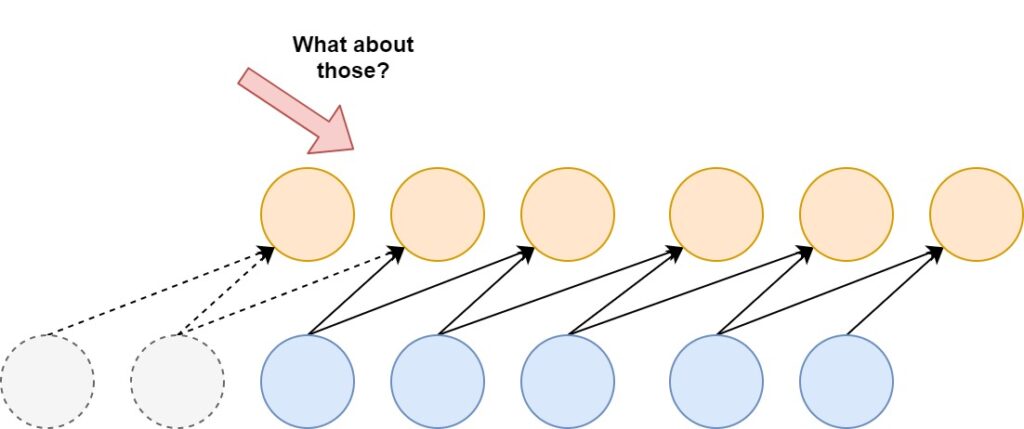
**Causal padding**

Suppose that you have a time series dataset, where *two inputs* together determine an *output*, in a causal fashion. Like this:

[](https://github.com/christianversloot/machine-learning-articles/blob/main/images/Causalpad-2.jpg)

It's possible to create a model that can handle this by means of a Conv1D layer with a kernel of size 2 - the learnt kernel will be able to map the inputs to the outputs successfully.

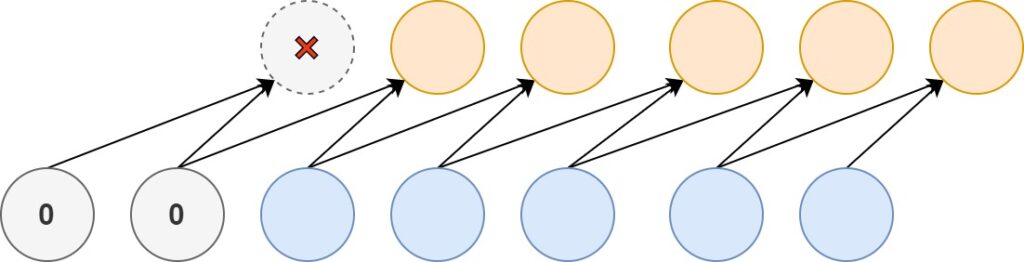
But what about the first two targets?

[](https://www.machinecurve.com/wp-content/uploads/2020/02/Causalpad-3.jpg)

Although they are valid targets, the *inputs* are incomplete - that is, there is insufficient input data available in order to successfully use them in the training process (The Blog, n.d.). For the second target, *one* input - visible in gray - is missing (whereas the second is actually there), while for the first target both aren't there.

For the first target, there is no real hope for success (as we don't have any input at all and hence do not know which values produce the target value), but for the second, we have a partial picture: we've got half the inputs that produce the target.

Causal padding on the Conv1D layer allows you to include the partial information in your training process. By padding your input dataset with zeros at the front, a causal mapping to the first, missed-out targets can be made (Keras, n.d.; The Blog, n.d.). While the first target will be useless for training, the second can now be used based on the partial information that we have:

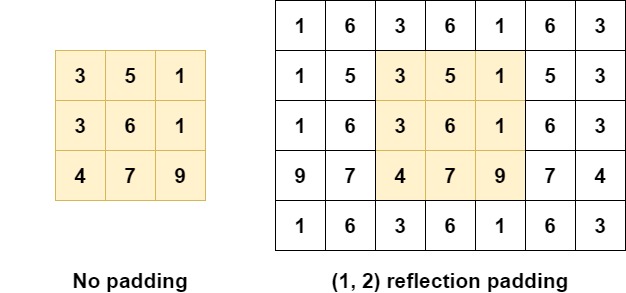
[](https://www.machinecurve.com/wp-content/uploads/2020/02/Causalpad-4.jpg)

**Reflection padding**

Another type of padding is "reflection padding" (TensorFlow, n.d.). As you can see, it pads the values with the "reflection" or "mirror" of the values directly in the opposite direction of the edge of your to be padded shape.

For example, if you look at the image below, for the first row of the yellow box (i.e., your shape):

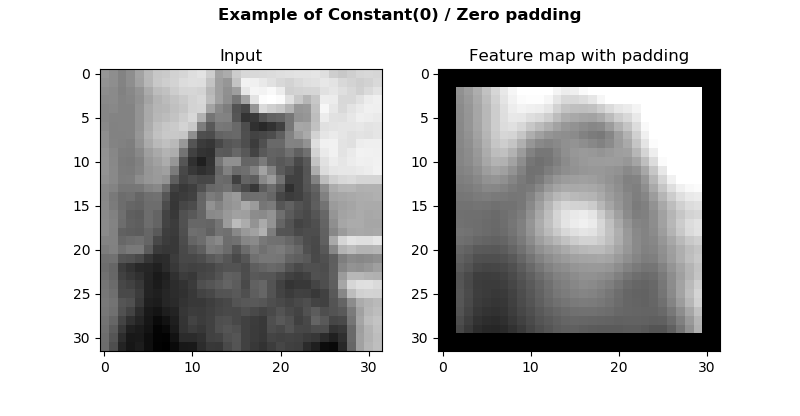
* If you go to the right, you'll see a 1. Now, you need to fill the padding element directly to the right. What do you find when you move in the *opposite* direction of the edge? Indeed, a 5. Hence, your first padding value is a 5. When you move further, it's a 3, so the next padding value following the 5 is a 3. And so on.
* In the opposite direction, you get a mirrored effect. Having a 3 at the edge, you'll once again find the 5 (as it's the center value) but the second value for padding will be a 1.
* And so on!

[](https://www.machinecurve.com/wp-content/uploads/2020/02/reflection_pad.jpg)

Reflective padding seems to improve the empirical performance of your model (Physincubus, n.d.). Possibly, this occurs because of how "zero" based padding (i.e., the "same" padding) and "constant" based padding alter the distribution of your dataset:

<https://twitter.com/karpathy/status/720622989289644033>

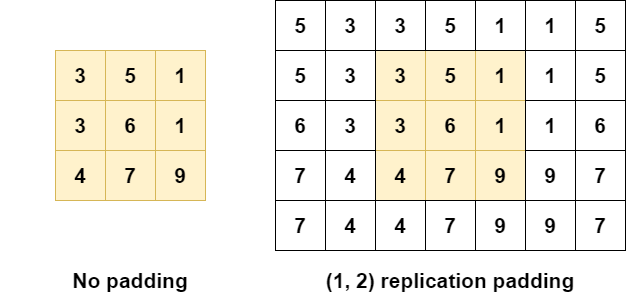
This becomes clear when we actually visualize the padding when it is applied:

* [](https://www.machinecurve.com/wp-content/uploads/2020/02/zero_padding.png)
* [](https://www.machinecurve.com/wp-content/uploads/2020/02/reflection.png)

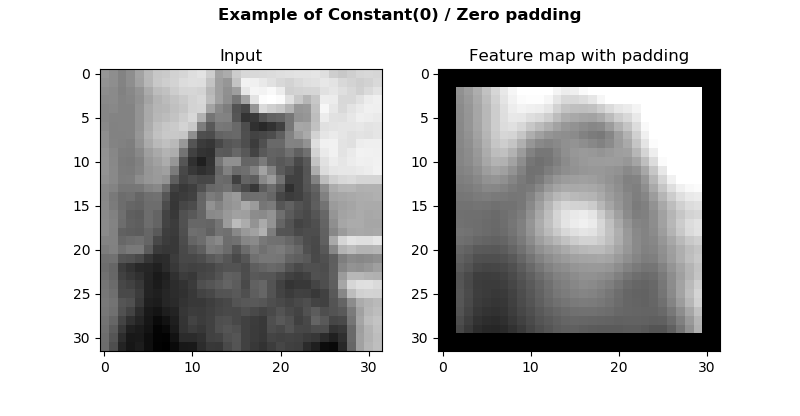
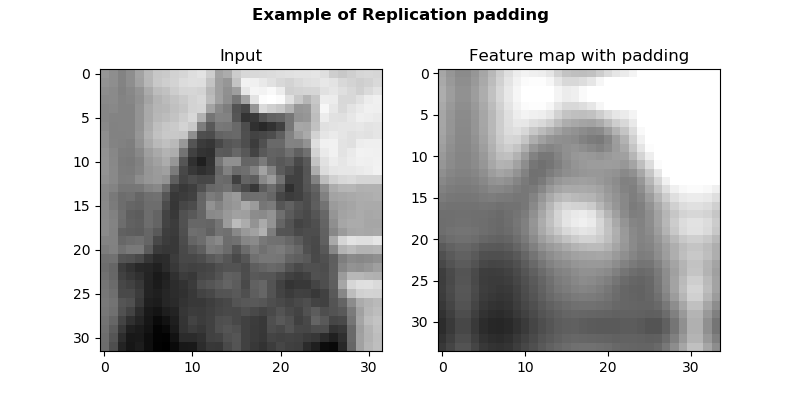
**Replication padding / symmetric padding**

Replication padding looks like reflection padding, but is slightly different (TensorFlow, n.d.). Rather than *reflecting* like a *mirror*, you simply take a copy, and mirror it. Like this:

* You're at the first row again, at the right. You find a 1. What is the next value?
* Simple: you copy the entire row, mirror it, and start adding it as padding values horizontally. So, for row 1 with [latex][3, 5, 1][/latex], this will be [latex][1, 5, 3][/latex] being added. As you can see, since we only pad 2 elements in width, there are 1 and 5, but 3 falls off the padding.

[](https://github.com/christianversloot/machine-learning-articles/blob/main/images/replication_pad.png)

As with reflection padding, replication padding attempts to reduce the impact of "zero" and "constant" padding on the quality of your data by using "plausible data values by re-using what is along the borders of the input" (Liu et al., 2018):

* [](https://www.machinecurve.com/wp-content/uploads/2020/02/zero_padding.png)
* [](https://www.machinecurve.com/wp-content/uploads/2020/02/replication.png)

**Which padding to use when?**

There are no hard criteria that prescribe when to use which type of padding. Rather, it's important to understand that padding is pretty much important all the time - because it allows you to preserve information that is present at the borders of your input data, and present there only.

We've seen multiple types of padding. If you have causal data (i.e. multiple inputs that lead to one target value) and use a one-dimensional convolutional layer to improve model efficiency, you might benefit from "causal" padding to stress the importance of causality in your data by ensuring that your target is never present *before all your input data*.

If you have an image classification problem, or wish to use Conv layers differently, causal padding might not be interesting for you. But "zero" padding, "constant" padding, "reflection" padding and "replication" padding may be. All of them add one or multiple columns and/or rows of padded elements around your shape, but each works differently. While zero and constant padding add zeros and constants, reflection and replication padding attempt to preserve the distribution of your data by re-using what's present along the borders. This, scholars like Liu et al. (2018) expect, could improve model performance. Hence, if you're in this scenario, you may wish to start with reflection or replication padding, moving to constant and eventually zero padding if they don't work.

**Summary**

This blog post discussed the necessity of padding that you may encounter in your machine learning problems - and especially when using Conv layers / when creating a ConvNet. It did so by taking a look at convolutional layers, explaining why borders only cannot be inspected when you don't add padding to your inputs.

Subsequently, we discussed various types of padding - valid padding (a.k.a. no padding), same (or zero) padding, constant padding, reflection padding and replication padding. Through this discussion, you are now likely able to explain the differences between those types of padding.

I hope you've learnt something today! If you did, please feel free to leave a comment in the comments section below 😊 Please do the same if you have any questions, remarks or when you spot a mistake.

Thank you for reading MachineCurve today and happy engineering! 😎

1. **What is the concept of stride?**

**What is Stride?**

**Stride is a parameter that dictates the movement of the kernel, or filter, across the input data, such as an image. When performing a convolution operation, the stride determines how many units the filter shifts at each step. This shift can be horizontal, vertical, or both, depending on the stride's configuration.**

**For example, a stride of 1 moves the filter one pixel at a time, while a stride of 2 moves it two pixels. A larger stride will produce a smaller output dimension, effectively downsampling the image.**

**Importance of Stride**

**The choice of stride affects the model in several ways:**

* **Output Size: A larger stride will result in a smaller output spatial dimension. This is because the filter covers a larger area of the input image with each step, thus reducing the number of positions it can occupy.**
* **Computational Efficiency: Increasing the stride can decrease the computational load. Since the filter moves more pixels per step, it performs fewer operations, which can speed up the training and inference processes.**
* **Field of View: A higher stride means that each step of the filter takes into account a wider area of the input image. This can be beneficial when the model needs to capture more global features rather than focusing on finer details.**
* **Downsampling:**

**Strides can be used as an alternative to pooling layers for downsampling the input. Pooling layers, such as**[**max pooling**](https://deepai.org/machine-learning-glossary-and-terms/max-pooling)**, are often used to reduce the spatial dimensions and to introduce invariance to small translations. However, increasing the stride in a convolutional layer can achieve a similar effect without the need for an additional pooling layer.**

**Stride in Practice**

**In practice, stride is often set to 1 or 2. A stride of 1 is common when the model needs to maintain a high resolution of features, which is particularly important in the initial layers of the network. A stride of 2 or more may be used in deeper layers or when the input images are large, and the model needs to reduce dimensionality to control the number of parameters and computational cost.**

**It's important to note that while increasing the stride can improve computational efficiency, it may also lead to a loss of information. Strides larger than 1 skip over pixels, which could contain useful information for**[**feature extraction**](https://deepai.org/machine-learning-glossary-and-terms/feature-extraction)**. Therefore, the choice of stride is a trade-off that needs to be carefully considered based on the specific task and dataset.**

**Calculating Output Size with Stride**

**The output size of a convolutional operation can be calculated using the following formula:**

***O = ((W - K + 2P) / S) + 1***

**Where:**

* ***O* is the output size**
* ***W* is the input size (width or height)**
* ***K* is the kernel size**
* ***P***

**is the**[**padding**](https://deepai.org/machine-learning-glossary-and-terms/padding)

* ***S* is the stride**

**This formula helps to determine the dimensions of the output feature map, which is essential for designing and understanding the architecture of a CNN.**

**Conclusion**

**Stride is a fundamental [hyperparameter](https://deepai.org/machine-learning-glossary-and-terms/hyperparameter) in convolutional**[**neural networks**](https://deepai.org/machine-learning-glossary-and-terms/neural-network)**that influences the model's performance and efficiency. It controls how the convolutional filters interact with the input data and affects the size of the output feature maps. Understanding and selecting the appropriate stride is crucial for optimizing CNNs for various tasks in image and video analysis, as well as other domains where CNNs are applicable.**

**When designing a convolutional neural network, one must consider the implications of stride on the network's ability to capture relevant features, computational requirements, and the overall performance of the model. Balancing these factors is key to developing effective and efficient CNNs for machine learning applications.**

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

**A 2D Convolution operation is a widely used operation in computer vision and deep learning. It is a mathematical operation that applies a filter to an image, producing a filtered output (also called a feature map). In this article, we will look at how to apply a 2D Convolution operation in PyTorch.**

**PyTorch provides a convenient and efficient way to apply 2D Convolution operations. It provides functions for performing operations on tensors (PyTorch’s implementation of arrays), and it also provides functions for building deep learning models.**

**Convolutions are a fundamental concept in computer vision and image processing. They are mathematical operations that take an input signal (such as an image) and produce a transformed output signal that highlights certain features of the input. Convolutional neural networks (ConvNets or CNNs) are deep learning models that are built using convolutions as a core component.**

**In the context of PyTorch, the meaning of 1D, 2D, and 3D convolutions is determined by the dimensionality of the input data that the convolution applied.1D Convolutions are applied to 1D input signals such as 1D arrays, sequences, or time series. In this case, the convolution kernel (or filter) slides along the input signal and performs element-wise multiplication and accumulation at each position to produce the output signal.2D Convolutions are applied to 2D input signals such as grayscale or color images. In this case, the convolution kernel slides over the 2D input array, performs element-wise multiplication and accumulation at each position, and produces a 2D output signal.3D Convolutions are applied to 3D input signals such as video or volumetric data. In this case, the convolution kernel slides over the 3D input array, performs element-wise multiplication and accumulation at each position, and produces a 3D output signal.**

**A convolution operation is a mathematical operation that is widely used in image processing and computer vision. It involves applying a convolution kernel, also known as a filter, to an image. The filter acts as a sliding window over the image, computing the dot product of its values with the underlying image pixels at each step.**

**Mathematically, a convolution operation can be represented as:**

**Where f and g are functions representing the image and the filter respectively, and \* denotes the convolution operator.**

### 2D convolution in PyTorch

#### Syntax of  Conv2d() :

***torch.nn.Conv2d(in\_channels, out\_channels, kernel\_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding\_mode=’zeros’, device=None, dtype=None)***

***Cout is determined by the number of filters used in the convolutional layer.***

***in\_channels (int) – Number of channels in the input image.***

***out\_channels (int) – Number of channels produced by the convolution.***

***kernel\_size (int or tuple) – Size of the convolving kernel.***

***bias (bool, optional) – If True, adds a learnable bias to the output. Default: True.***

***stride : controls the stride for the cross-correlation, a single number or a tuple.***

***padding : controls the amount of padding applied to the input. It can be either a string {‘valid’, ‘same’} or a tuple of ints giving the amount of implicit padding applied on both sides.***

***dilation : controls the spacing between the kernel points; also known as the à trous algorithm.  It is harder to describe, but this link has a nice visualization of what dilation does.***

***groups : controls the connections between inputs and outputs. in\_channels and out\_channels must both be divisible by groups.***

#### For 2D convolution in PyTorch, we apply the convolution operation by using the simple formula :

**The input shape refers to the dimensions of a single data sample in a batch. The shape is defined as (N, Cin, Hin, Win), where:**

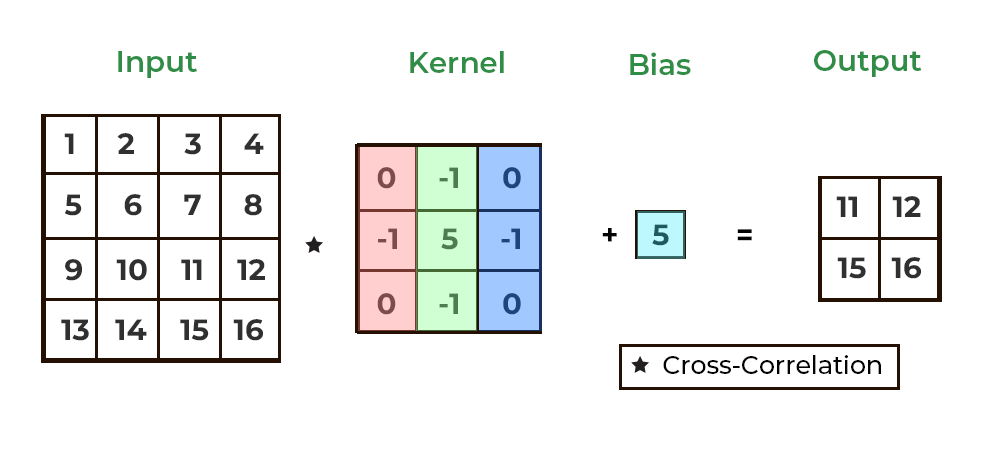
* **N is the batch size or number of samples in the batch**
* **Cin is the number of channels in the input data**
* **Hin is the height of the input data**
* **Win is the width of the input data**

**The output shape refers to the dimensions of the output from a convolutional layer. The shape is defined as (N, Cout, Hout, Wout), where:**

* **N is the batch size or number of samples in the batch**
* **Cout is the number of channels in the output data**
* **Hout is the height of the output data**
* **Wout is the width of the output data  
  These shapes can be determined mathematically based on the kernel size, stride, and padding of the convolutional layer. The formula for Hout is:**

**Similarly, the formula for Wout is:**

### ****Let’s consider this with an example, Here we define a custom image of shape 4X4 and kernel 3X3 and bias 1X1.****



**8. What exactly is a channel?**

**A channel in a CNN (Convolutional Neural Network) refers to a specific feature map resulting from applying filters to the input data, typically representing different learned patterns or features.**

**In a Convolutional Neural Network (CNN), a channel refers to a specific dimension along which feature maps are organized. To understand this concept, let’s break down how CNNs work:**

1. **Convolutional Layers: In CNNs, convolutional layers are responsible for learning and extracting features from the input data. Each layer consists of multiple filters (also known as kernels), which are small matrices applied to different regions of the input data through convolution operations.**
2. **Feature Maps: When a filter is applied to the input data through convolution, it produces an output known as a feature map. Each filter learns to detect a specific pattern or feature in the input data, such as edges, textures, or shapes. Multiple filters are typically used in each convolutional layer to capture different types of features.**
3. **Channel Dimension: The feature maps generated by applying different filters are organized along a dimension known as the channel dimension. Each feature map corresponds to a specific channel, representing the activation or response of the corresponding filter to different regions of the input data.**
4. **Depth of Convolutional Layers: The number of channels in a convolutional layer corresponds to the depth of that layer. For example, if a convolutional layer has 32 filters, each producing a feature map, then the layer would have 32 channels. The depth of the convolutional layers increases as we move deeper into the network, allowing the network to learn increasingly complex and abstract features.**
5. **Interpretation of Channels: Each channel in a convolutional layer captures different aspects or representations of the input data. For example, early layers may capture low-level features like edges and textures, while deeper layers may capture higher-level features like object parts or semantic information.**
6. **Computational Representation: Mathematically, the output of a convolutional layer can be represented as a 3D tensor, with dimensions corresponding to width, height, and channels. For example, an image input with dimensions 32x32x3 (width x height x channels) would produce a feature map with dimensions 32x32xN, where N is the number of channels in that layer.**

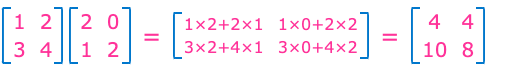
**In summary, a channel in a CNN represents a specific feature map produced by applying filters to the input data through convolution. Channels organize the learned features along a specific dimension, allowing the network to capture diverse patterns and representations of the input data across different channels.**

**9.Explain relationship between matrix multiplication and a convolution?**

**Matrix multiplication and convolution are both linear operations, but they are different in the way they operate on the input data.**

# Matrix multiplication

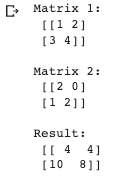
**Matrix multiplication involves the element-wise multiplication of two matrices followed by summing the resulting products to produce a scalar value. The result is a new matrix with dimensions that depend on the dimensions of the input matrices. Specifically, the resulting matrix has a number of rows equal to the number of rows in the first matrix and a number of columns equal to the number of columns in the second matrix.**

****

**Figure 1: Matix Multiplication**

## Matrix multiplication in Python

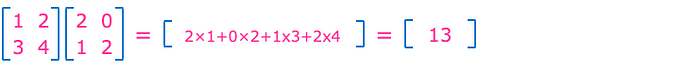
**import numpy as np  
  
# define two 2x2 matrices  
matrix1 = np.array([[1, 2], [3, 4]])  
matrix2 = np.array([[2, 0], [1, 2]])  
  
# perform matrix multiplication using numpy.dot() function  
result = np.dot(matrix1, matrix2)  
  
# print the matrices and the result  
print("Matrix 1:\n", matrix1)  
print("\nMatrix 2:\n", matrix2)  
print("\nResult:\n", result)**

****

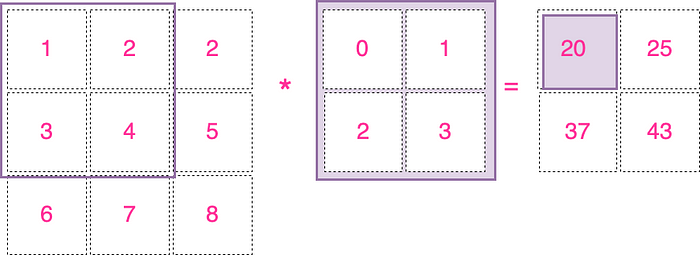
**Figure 2: Result of matrix multiplication**

# Mechanics of Convolution

**Contrarily, convolution involves sliding a small matrix, called a kernel or filter, over the input matrix, and computing the element-wise product between the kernel and the overlapping sub-matrix of the input. The resulting products are summed up to produce a single value for each position of the kernel on the input matrix, which is stored in the output matrix. The dimensions of the output matrix depend on both the kernel’s size and the size of the input matrix.**

****

**Figure 3: Convolution computation**

****

**Figure 4: Convolution example with Kernel**

## Code for convolution in Python

**import numpy as np  
  
def conv2d(image, kernel):  
 """  
 Computes a 2D convolution of an image with a kernel.  
   
 Args:  
 - image: a 2D NumPy array of shape (input\_height, input\_width)  
 - kernel: a 2D NumPy array of shape (kernel\_height, kernel\_width)  
   
 Returns:  
 - convolved\_image: a 2D NumPy array of shape (input\_height - kernel\_height + 1, input\_width - kernel\_width + 1)  
 """  
 # Get image dimensions and kernel size  
 input\_height, input\_width = image.shape  
 kernel\_height, kernel\_width = kernel.shape  
  
 # Initialize the output image  
 output\_height = input\_height - kernel\_height + 1  
 output\_width = input\_width - kernel\_width + 1  
 convolved\_image = np.zeros((output\_height, output\_width))  
   
 # Loop over every pixel of the output image  
 for i in range(0,output\_height ):  
 for j in range(0, output\_width):  
 # Compute the convolution at the current pixel  
 for ii in range(0, kernel\_height):  
 for jj in range(0, kernel\_width):  
 convolved\_image[i,j] += image[i+ii, j+jj] \* kernel[ii,jj]  
   
 return convolved\_image  
  
# Example usage  
image = np.array([[1, 2, 2],  
 [3, 4, 5],  
 [6, 7, 8]])  
  
kernel = np.array([[0, 1],  
 [2, 3]])  
  
convolved\_image = conv2d(image, kernel)  
  
print(convolved\_image)**

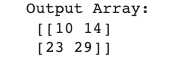
**https://miro.medium.com/v2/resize:fit:141/1*QnmarJcC9RwjciQONJPIBA.png**

**Figure 5:Result of convolution computation**

## Code for convolution using Scipy

**To simplify we can use the “convolve2d” function from scipy.**

**import numpy as np  
from scipy.signal import convolve2d  
  
# define a 2D array (or matrix) as input  
input\_array = np.array([[1, 2, 2],  
 [3, 4, 5],  
 [6, 7, 8]])  
  
# define a kernel (or filter) as a 2D array  
kernel = np.array([[0, 1],  
 [2, 3]])  
  
# perform 2D convolution using scipy.signal.convolve2d  
output\_array = convolve2d(input\_array, kernel, mode='valid')  
  
# print the input, kernel, and output arrays  
print("Input Array:\n", input\_array)  
print("\nKernel:\n", kernel)  
print("\nOutput Array:\n", output\_array)**

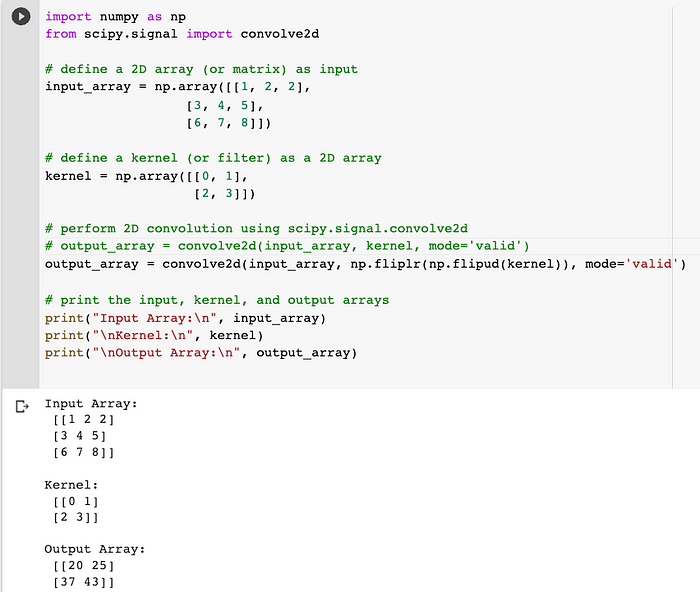
****

**Figure 6:Result of convolution computation using Scipy**

**The problem with the above code is, if you use this function as is, you will get a totally different result than we did with the “Code for convolution in Python” approach above. This is because “convolve2d” does a true convolution and not the deep learning version of convolution (i.e. with plus instead of a minus).**

**In order to make the scipy convolution work the same way, we have to flip the filter both horizontally and vertically and set the mode argument as ‘valid’.**

**output\_array = convolve2d(input\_array, np.fliplr(np.flipud(kernel)), mode='valid')**

****

**Figure 7: Flip the input image and kernel**

**As a side note, convolution is a commutative operative i.e. “input\_array” convolved with “kernel” is the same as the “kernel” convolve with “input\_array”, therefore it does not matter which input we flip.**

**Actually, what we perform in deep learning is cross-correlation, not true convolution. The operation itself is a combination of element-wise multiplication and addition, but it’s conventionally referred to as convolution in the field of deep learning since the filters’ weights are learned through training.**

## Summary

**Convolution and Matrix Multiplication processes are distinct from one another, despite the fact that matrix multiplication can be thought of as a specific example of convolution (where the kernel is the second matrix transposed).**

**Convolution is frequently utilized in machine learning and linear algebra activities like linear regression and neural network calculations, while matrix multiplication is more frequently employed in signal processing tasks like filtering and feature extraction from images.**

**Therefore, convolution and matrix multiplication are independent operations with unique mathematical features and intended applications, despite certain similarities between them.**