Convolution

Learn how convolutions work and the role they play in CNNs.

Chapter Goals:

- Learn about convolutions
- Write a convolution layer for the neural network

A. Filters and kernels

As mentioned at the end of the **Image Processing** section, filters play a huge role in image recognition. We use filters to transform inputs and extract features that allow our model to recognize certain images. A very high-level example of this would be a curve detecting filter, which allows our model to distinguish between digits with curves (e.g. 8) and digits without curves (e.g. 1).

The weights of a filter are defined through a *kernel matrix*. The kernel is usually a square matrix and its weights are just floating point numbers.

$$\begin{bmatrix} 0.0 & 1.0 & 0.4 \\ 0.1 & 0.0 & 0.0 \\ 1.2 & 0.0 & 1.1 \end{bmatrix}$$

Example kernel weight matrix, with 3x3 dimension.

Like all neural network weights, the filter's weights are trainable variables. We train our neural network (via the kernel matrix weights) to produce filters that are able to extract the most useful hidden features.

When the input data has multiple channels, a filter will have a separate kernel matrix per channel. The MNIST dataset only has one channel, but for other types of image data (e.g. RGB), we would train the model to obtain optimal weights for each channel's kernel matrix.

B. Convolution

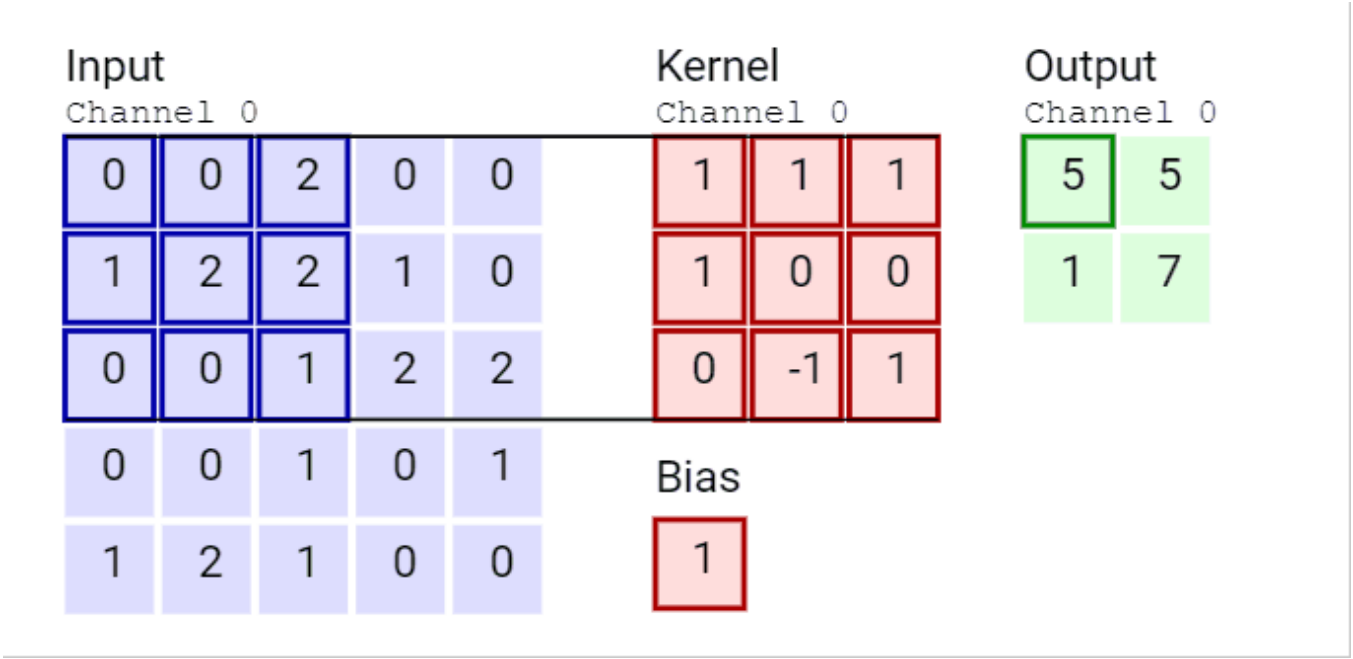
We've now reached the focal point of convolutional neural networks: the convolution. The convolution represents how we apply our filter weights to the input data. The main operation used by a convolution is the matrix dot product, i.e. a summation over the element-wise product of two matrices.

$$\begin{bmatrix} 0.4 & 0.1 \\ 0.1 & 0.2 \end{bmatrix} \cdot \begin{bmatrix} 0.0 & 1.0 \\ 0.4 & 0.2 \end{bmatrix} = 0.18$$

A matrix dot product, where · represents the dot product operation.

In addition to matrix dot products, a convolution also uses a trainable bias term. The bias term is added to the output of each matrix dot product in a convolution.

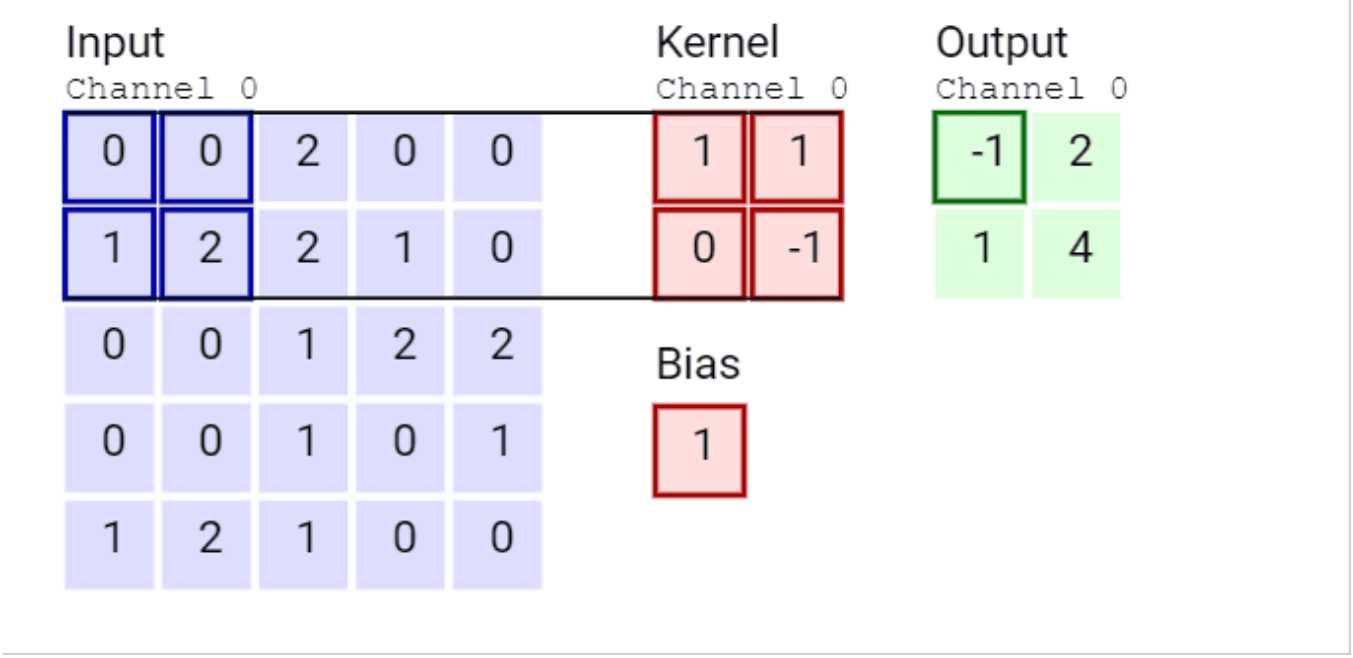
The number of matrix dot products in a convolution depends on the dimensions of the input data and kernel matrix, as well as the *stride size*. The stride size is the vertical/horizontal offset of the kernel matrix as it moves along the input data. Below is an example of a convolution with a stride size of 2:



Convolution with stride size of 2 and kernel dimension of 3x3.

C. Padding

The convolution example from the previous section involved a kernel and stride size that fit nicely with the input data. However, if we had instead used a 2x2 kernel, the last row and column of the input data matrix would not have been used in the convolution:



Convolution with stride size of 2 and kernel dimension of 2x2. Since the input dimension is 5x5, the last row and column of the matrix are not used.

If we want to use all the input data in our convolution, we can *pad* the input data matrix with 0's. This means we add rows/columns made entirely of 0's to the edges of the input data matrix. Since 0 multiplied by any number results in 0, the padding doesn't affect matrix dot products. This is important because we don't want to add any distortions to our convolution.

Convolution with stride size of 2 and kernel dimension of 2x2. The input data is padded with 0's on the right and bottom of the matrix.

You might ask why we only padded 0's to the right and bottom of the input

data matrix, rather than around all the edges. To avoid superfluous dot

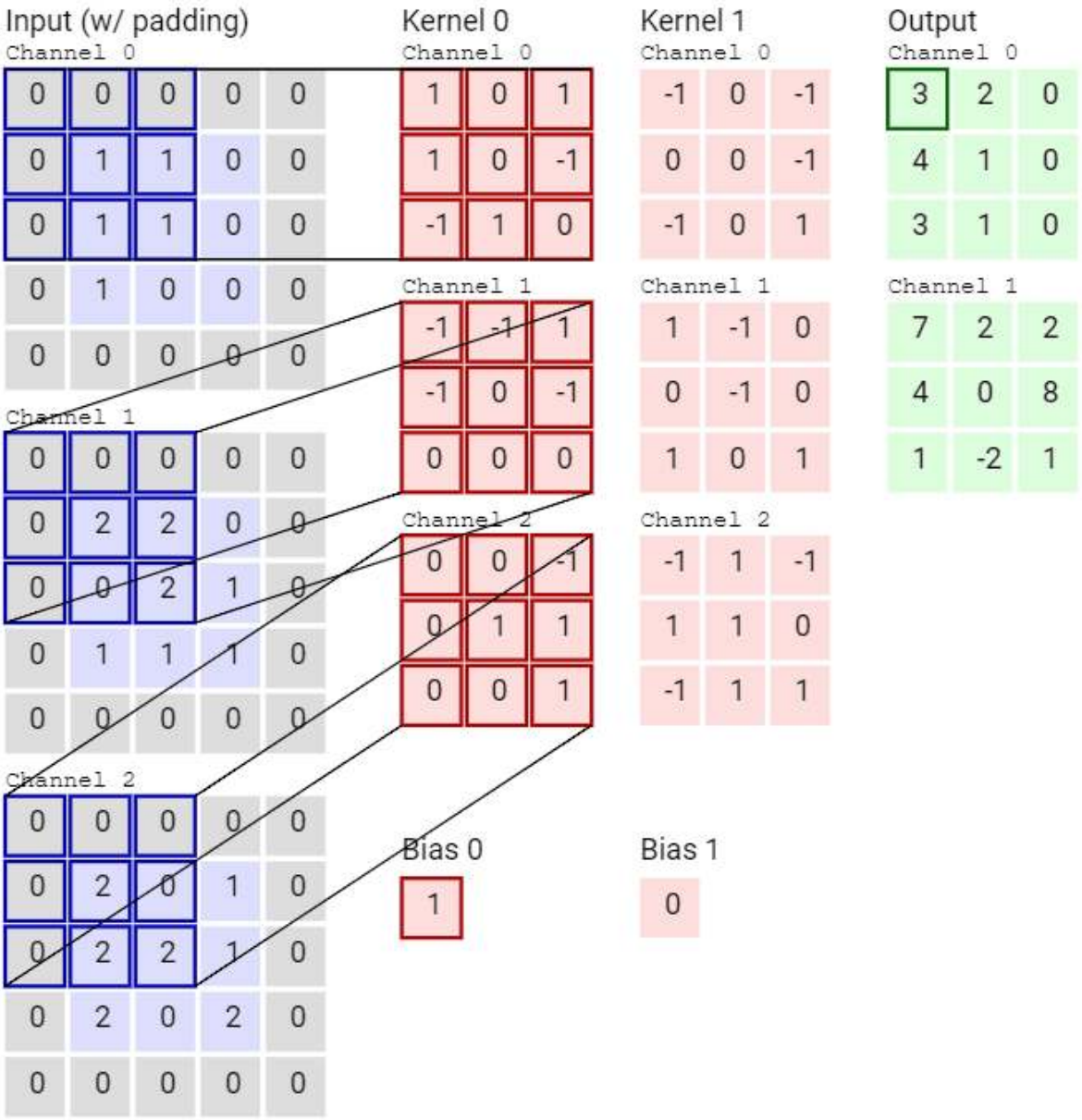
products, we only pad the absolute minimum amount necessary to use all our input data in the convolution.

The padding is distributed as evenly as possible along the top, bottom, left, and right of the matrix. When we have an odd number of padded rows/columns (as in the above example), the additional row/column goes on the bottom and right of the matrix, respectively.

D. Convolution layer

A convolution layer in a CNN applies multiple filters to the input tensor. While each filter has a separate kernel matrix for each of the input channels, the overall result of a filter's convolution is the sum of the convolutions across all the input channels.

Adding more filters to a convolution layer allows the layer to better extract hidden features. However, this comes at the cost of additional training time and computational complexity, since filters add extra weights to the model. The number of channels for the output data is equal to the number of filters the convolution layer uses.



Example convolution layer with 2 filters. The input data has 3 channels and is padded with 0's.

Time to Code!

We'll apply a convolution layer to our `reshaped_inputs` from the previous chapter. In TensorFlow, a convolution layer is implemented with the `tf.layers.conv2d` function. The function takes in the following required arguments:

- `inputs`: The input data tensor.
- `filters`: The number of filters to use.
- `kernel_size`: The height and width dimensions of the kernel matrix.

Some important keyword arguments are:

- `strides`: The stride size for the kernel matrix. Can be a single integer

(same stride size for vertical/horizontal) or a tuple of 2 integers (manually specified vertical/horizontal stride sizes). The first element of the tuple is the vertical stride, the second is the horizontal stride. It defaults to `(1, 1)`.

- `padding`: Either `'valid'` (no padding) or `'same'` (padding).
- `activation`: The activation function to use. Defaults to `None`.
- `name`: The name for the convolution layer (useful for debugging and visualization).

Set `conv1` equal to `tf.layers.conv2d` applied with `reshaped_inputs` as the inputs. The function will use `32` filters, a kernel size of `[5, 5]`, `'same'` padding, and `tf.nn.relu` activation. We'll also set the `name` argument to `'conv1'`.

```
import tensorflow as tf

class MNISTModel(object):
    # Model Initialization
    def __init__(self, input_dim, output_size):
        self.input_dim = input_dim
        self.output_size = output_size

    # CNN Layers
    def model_layers(self, inputs, is_training):
        reshaped_inputs = tf.reshape(
            inputs, [-1, self.input_dim, self.input_dim, 1])
        # CODE HERE
```