Chapter 15

Classifying Images with Deep Convolutional Neural Networks

September 4, 2018

Deep learning

- Outline
  - Understanding convolution operations in one and two dimensions
  - Learning about the building blocks of CNN architectures
  - Implementing the fundamental backpropagation algorithm for neural network training from scratch
  - Implementing deep convolutional neural networks

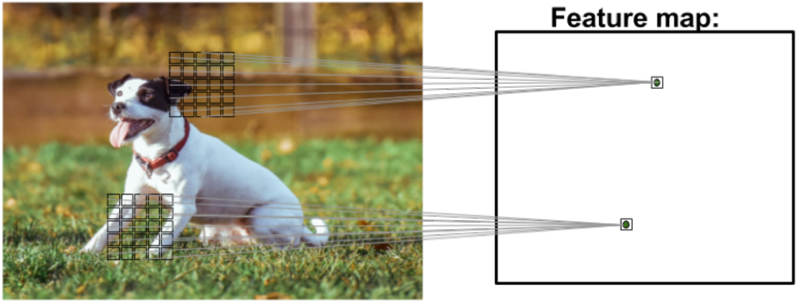
Convolutional neural networks (CNNs)

- Family of models that were inspired by how the visual cortex of human brain works when recognizing objects.
- 1990's, when Yann LeCun and his colleagues proposed a novel neural network architecture for classifying handwritten digits from images.
- CNNs are used as feature extraction engines.

Convolutional neural networks

- Relevant features are keys to the performance of any machine learning algorithm.
- Neural networks are able to automatically learn the features from raw data that are most useful for a particular task.
- Multilayer neural networks, and in particular, deep CNNs, construct a so-called feature hierarchy by combining the low-level features in a layer-wise fashion to form high-level features.
- In images, then low-level features, such as edges and blobs, are extracted from the earlier layers, which are combined together to form high-level features – as object shapes like a building, a car, or a dog.

Feature Maps



CNNs usually performs very well for image-related tasks, and that's largely due to two important ideas:

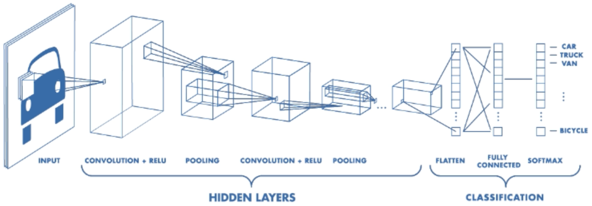
- **Sparse-connectivity**: A single element in the feature map is connected to only a small piece (patch) of pixels (Perceptrons connect the whole input image).
- **Parameter-sharing**: The same weights are used for different pieces (patches) of the input image.

Convolutional neural networks

- As a direct consequence of these two ideas, the number of weights (parameters) in the network decreases dramatically, and improves the ability to capture relevant features.
- Nearby pixels are probably more relevant to each other than pixels that are far away from each other.

Convolutional Neural Networks

- Typically, CNNs are composed of several Convolutional layers and subsampling (Pooling), that are followed by one or more Fully Connected.
- Pooling layers do not have learnable parameters (no weights or bias units).
- The flattening step is needed so that you can make use of fully connected layers after some convolutional layers. Each feature map channel in the output of a CNN layer is a "flattened" 2D array.



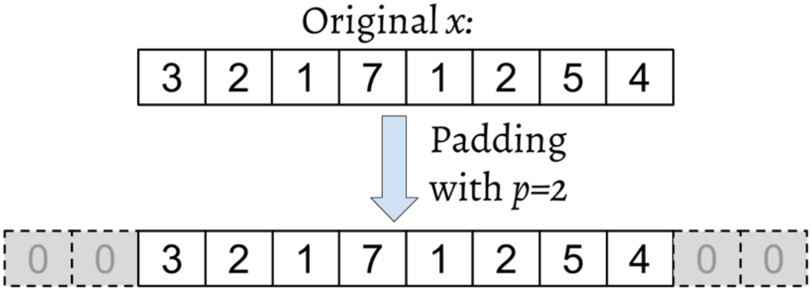
Performing discrete convolutions

- A **discrete convolution** for two one-dimensional vectors x and w is denoted by $y = x \times w$, in which vector x is our input (sometimes called signal) and w is called the filter or kernel.
- The multiplication is performed in the opposite direction.
- if x has 10 features with indices 0,1,2,...,8,9, then indices $-\infty$: - 1 and 10 : $+\infty$ are out of bounds for x.
- To correctly compute the summation shown in the preceding formula, it is assumed that x and w are filled with zeros. Then, x is padded only with a finite number of zeros.

$$y = x \times w, y[i] = \sum_{k=+\infty}^{-\infty} x[i-k]w[k] \tag{1}$$

Padding

This process is called zero-padding or simply padding (p is the number of zero padded).



Performing discrete convolutions

- Original input x and filter w have n and m elements, respectively, where $m \leq n$.
- The padded x_p has size $n + 2p$.
- The **pratical formula for computing a discrete convolution** is:

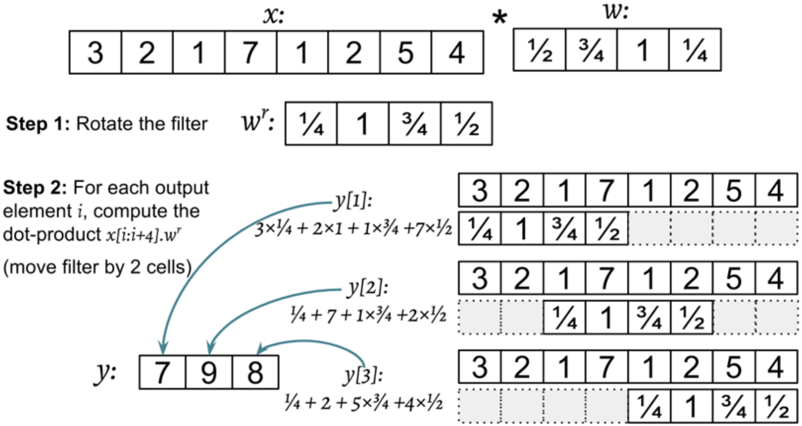
$$y = x \times w, y[i] = \sum_{k=0}^{m-1} x_p[i + m - k] w[k] \tag{2}$$

- Notice x and w are indexed in different directions in this summation. Flip one of those vectors, x or w, after they are padded.
- Then, we can simply compute their dot product:

$$y[i] = x[i : i + m] w^T \tag{3}$$

Performing discrete convolutions/Stride

- Padding size is zero (p = 0).
- Notice rotated filter w^T is shifted by two cells each time. Hyperparameter of a convolution: the stride s. In this example, s = 2.



The effect of zero-padding in a convolution

- Now consider an example where $n = 5, m = 3$. Then, $p = 0$, $x[0]$ is only used in computing one output element (for instance, $y[0]$), while $x[1]$ is used in the computation of two output elements (for instance, $y[0]$ and $y[1]$) – also depends on the stride s value.
- Different treatment of elements of x can artificially put more emphasis on the middle element, $x[2]$, since it has appeared in most computations.
- We can avoid this issue if we choose $p = 2$, in which case, each element of x will be involved in computing three elements of y.

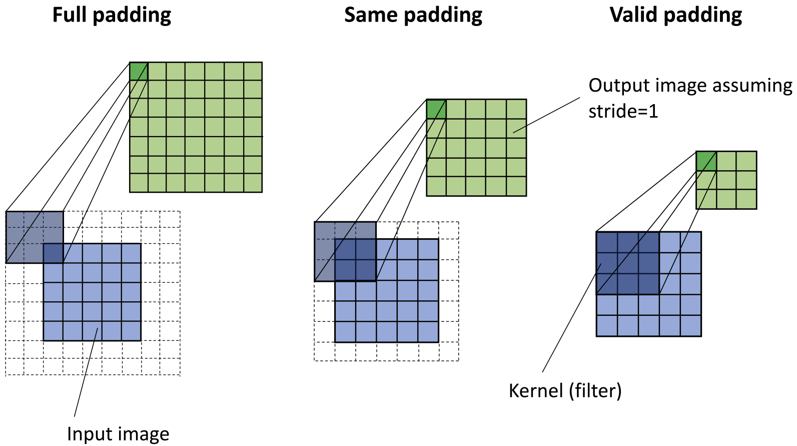
The effect of zero-padding in a convolution

There are three modes of padding that are commonly used in practice: **full, same, and valid**:

- In the **full** mode, $p = m - 1$ (m is the size of kernel columns/lines). Full padding increases the dimensions of the output; thus, it is rarely used in convolutional neural network architectures.
- **Same** padding is usually used if you want to have the size of the output the same as the input vector x. In this case, p is computed according to the filter size, along with the requirement that the input size and output size are the same.
- **Valid** mode refers to the case where $p = 0$ (no padding).

The effect of zero-padding in a convolution

The following figure illustrates the three different padding modes for a simple 5 x 5 pixel input with a kernel size of 3 x 3 and a stride of 1



The effect of zero-padding in a convolution

- The most commonly used padding mode in convolutional neural networks is same padding.
- **Advantage**: preserves the height and width of the input images, which makes designing a network architecture more convenient.
- In practice, it is recommended that you preserve the spatial size using same padding for the convolutional layers and decrease the spatial size via pooling layers instead (not by using valid padding).

The effect of zero-padding in a convolution

- Big disadvantage of the **valid padding**: the volume of the tensors would decrease substantially in neural networks with many layers, which can be detrimental to the network performance.
- Full padding: its size results in an output larger than the input size. It is usually used in signal processing applications where it is important to minimize boundary effects.
- **However, in deep learning context, boundary effect is not usually an issue, so we rarely see full padding.**

Determining the size of the convolution output

- The output size of a convolution is determined by the total number of times that we shift the filter w along the input vector.
- Let's assume that the input vector has size n and the filter is of size m. Then, the size of the output resulting from $x \times w$ with padding p and stride s is determined as follows:

$$o = \left\lfloor \frac{n + 2p - m}{s} \right\rfloor + 1 \tag{4}$$

Determining the size of the convolution output

- Compute the output size for an input vector of size 10 with a convolution kernel of size 5, padding 2, and stride 1:
  $n = 10, m = 5, p = 2, s = 1 \rightarrow o = \left\lfloor \frac{10+2*2-5}{1} \right\rfloor + 1 = 10$
- Output size turns out to be the same as the input; therefore, we conclude this as mode=**same**.
- How can the output size change for the same input vector, but have a kernel of size 3, and stride 2?
  $n = 10, m = 3, p = 2, s = 2 \rightarrow o = \left\lfloor \frac{10+2*2-3}{2} \right\rfloor + 1 = 6$

Performing a discrete convolution in 1D

iPython notebook

Performing a discrete convolution in 2D

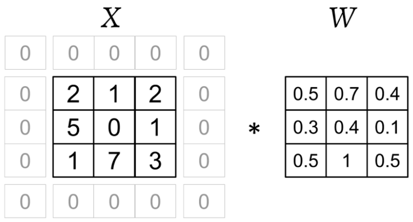
- When we deal with two-dimensional input, such as a matrix $X_{n_1 x n_2}$ and the filter matrix $W_{m_1 m_2}$, where $m_1 \leq n_1$ and $m_2 \leq n_2$, then the matrix $Y = X \times W$ is the result of 2D convolution of X with W.

$$Y = X * W \rightarrow Y[i,j] = \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} X[i-k_1, j-k_2] W[k_1, k_2]$$

- Notice that if you omit one of the dimensions, the remaining formula is exactly the same as the one we used previously to compute the convolution in 1D (Equation 1).
- All techniques, such as zero-padding, rotating the filter matrix, and the use of strides, are also applicable to 2D convolutions.

Performing a discrete convolution in 2D

Consider X_{33} , a kernel matrix W_{33} , padding $p = (1, 1)$, and stride $s = (2, 2)$. After padding, X becomes X_{5x5}^{padded} .



With the preceding filter, the rotated filter will be (flip lines and

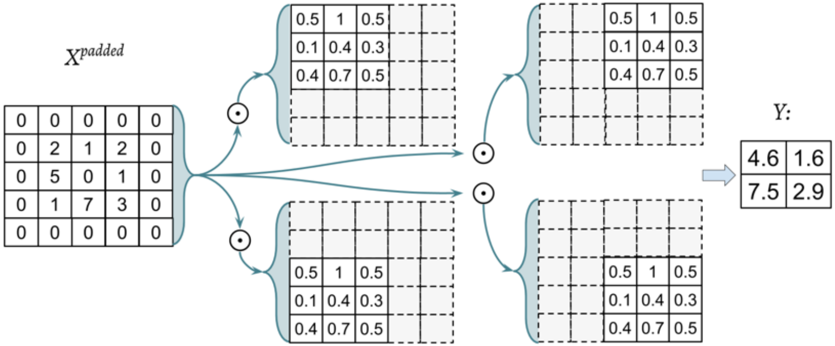
$$W' = \begin{bmatrix} 0.5 & 1 & 0.5 \\ 0.1 & 0.4 & 0.3 \\ 0.4 & 0.7 & 0.5 \end{bmatrix}$$

columns):

Note that this rotation is not the same as the transpose matrix.

Performing a discrete convolution in 2D

we can shift the rotated filter matrix along X^{padded} like a sliding window and compute the sum of the element-wise product $\odot$:



The result is Y_{2x2}.

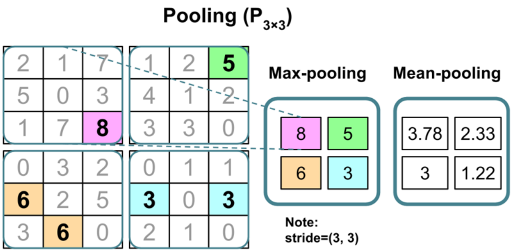
Performing a discrete convolution in 2D

▸ iPython notebook

Subsampling

- Typically applied in two forms of pooling operations in CNNs: **max-pooling** and **mean-pooling** (also known as average-pooling).
- The pooling layer is usually denoted by $P_{n_1 \times n_2}$. The subscript determines the size of the neighborhood (the number of adjacent pixels in each dimension), where the **max** or **mean** operation is performed.
- Such a neighborhood is the pooling size.

Subsampling



Main **advantages** of pooling:

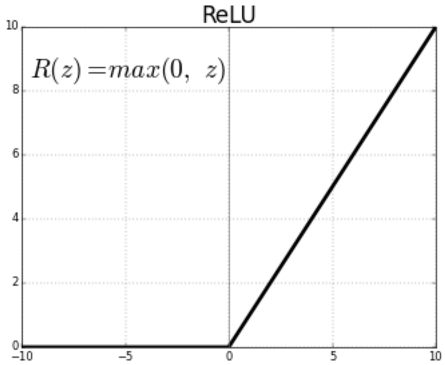
- **Small changes in a local neighborhood do not change the result of max-pooling.** Therefore, it helps to generate features that are more robust to noise in the input data.
- **Pooling decreases the size of features, which results in higher computational efficiency.** Furthermore, reducing the number of features may reduce the degree of overfitting as well.

Putting everything together to build a CNN

- **Matrix-vector multiplications to pre-activations** (or net input) as $a = Wx + b$, where x is a column vector representing pixels, and W is the weight matrix connecting the pixel inputs to each hidden unit.
- **In a convolutional neural network**, this operation is replaced by a convolution operation, as in $A = W * X + b$, where X is a matrix representing the pixels in a height x width arrangement.
- The pre-activations are passed to an activation function to obtain the activation of a hidden unit $H = \phi(A)$, where ϕ is the activation function.
- Subsampling (or pooling) is another building block of a convolutional neural network.

Working with multiple input or color channels

- In reality convolutions are performed in 3D. Each image is namely represented as a 3D matrix with a dimension for width, height, and depth. Depth is a dimension because of the colours channels used in an image (RGB).
- We perfom numerous convolutions on our input, where each operation uses a different filter. This results in different feature maps. In the end, we take all of these feature maps and put them together as the final output of the convolution layer.
- Just like any other Neural Network, we use an activation function to make our output non-linear. This could be the ReLU activation function.


ReLU
$R(z) = max(0, \ z)$
10
8
6
4
2
0
−10
−5
0
5
10

Working with multiple input or color channels

- $X_{N_1 N_2 C_{in}}$, where C_{in} is the number of input channels.
- If the image is colored and uses the RGB color mode, then C_{in} = 3 (for the red, green, and blue color channels in RGB).
- If the image is in grayscale, then we have C_{in} =1 because there is only one channel with the grayscale pixel intensity values.
- How can we incorporate multiple input channels in the convolution operation that we discussed in the previously?

Working with multiple input or color channels

- **Answer**: we perform the convolution operation for each channel separately and then add the results together using the matrix summation.
- The final result h is a feature map.

Given a sample $\mathbf{X}_{n_1 \times n_2 \times c_{in}}$,
a kernel matrix $\mathbf{W}_{m_1 \times m_2 \times c_{in}}$,
and bias value b

$$\Rightarrow \begin{cases} \boldsymbol{Y}^{Conv} = \sum_{c=1}^{C_{in}} \boldsymbol{W}[:,:,c] * \boldsymbol{X}[:,:,c] \\ \text{pre-activation:} \qquad \mathbf{A} = \mathbf{Y}^{\text{Conv}} + \text{b} \\ \text{Feature map:} \qquad \boldsymbol{H} = \phi(\boldsymbol{A}) \end{cases}$$

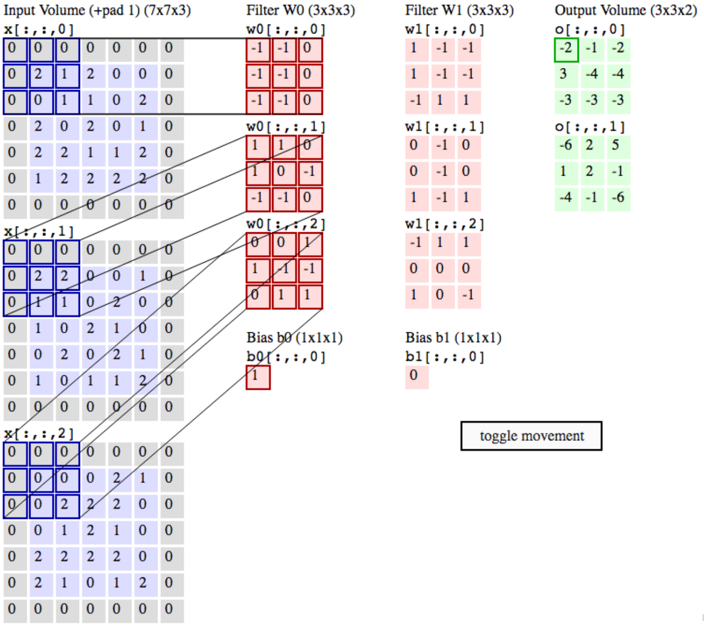
Working with multiple input or color channels

- If we use multiple feature maps, the kernel tensor becomes four-dimensional: width x height x C_{in} x C_{out}, where C_{out} is the number of output feature maps.

Given a sample $\boldsymbol{X}_{n_1 \times n_2 \times C_{in}}$
kernel matrix $\boldsymbol{W}_{m_1 \times m_2 \times C_{in} \times C_{out}}$
and bias vector $\boldsymbol{b}_{C_{out}}$

$$\Rightarrow \begin{cases} \boldsymbol{Y}^{Conv}[:,:,k] = \sum_{c=1}^{C_{in}} \boldsymbol{W}[:,:,c,k] * \boldsymbol{X}[:,:,c] \\ \boldsymbol{A}[:,:,k] = \boldsymbol{Y}^{Conv}[:,:,k] + \boldsymbol{b}[k] \\ \boldsymbol{H}[:,:,k] = \phi(\boldsymbol{A}[:,:,k]) \end{cases}$$

Working with multiple input or color channels



Input Volume (+pad 1) (7x7x3)

x[:,:,0]

0	0	0	0	0	0	0
0	2	1	2	0	0	0
0	0	1	1	0	2	0
0	2	0	2	0	1	0
0	2	2	1	1	2	0
0	1	2	2	2	2	0
0	0	0	0	0	0	0

x[:,:,1]

0	0	0	0	0	0	0
0	2	2	0	0	1	0
0	1	1	0	2	0	0
0	1	0	2	1	0	0
0	0	2	0	2	1	0
0	1	0	1	1	2	0
0	0	0	0	0	0	0

x[:,:,2]

0	0	0	0	0	0	0
0	0	0	0	2	1	0
0	0	2	2	2	0	0
0	0	1	2	1	0	0
0	2	2	2	2	0	0
0	2	1	0	1	2	0
0	0	0	0	0	0	0

Filter W0 (3x3x3)

w0[:,:,0]

-1	-1	0
-1	-1	0
-1	-1	0

w0[:,:,1]

1	1	0
1	0	-1
-1	-1	0

w0[:,:,2]

0	0	1
1	-1	-1
0	1	1

Bias b0 (1x1x1)

b0[:,:,0]

1

Filter W1 (3x3x3)

w1[:,:,0]

1	-1	-1
1	-1	-1
-1	1	1

w1[:,:,1]

0	-1	0
0	-1	0
1	-1	1

w1[:,:,2]

-1	1	1
0	0	0
1	0	-1

Bias b1 (1x1x1)

b1[:,:,0]

0

Output Volume (3x3x2)

o[:,:,0]

-2	-1	-2
3	-4	-4
-3	-3	-3

o[:,:,1]

-6	2	5
1	2	-1
-4	-1	-6

toggle movement

When using a CNN, the four important hyperparameters we have to decide on are:

- The kernel size (size of a weight matrix);
- The filter count (that is, how many filters do we want to use);
- The stride (how big are the steps of the filter);
- The padding;

Regularizing a neural network with dropout

- Decide about the size of a weight matrix and the number of layers need to be tuned to achieve a reasonably good performance is challenging.
- The capacity of a network refers to the level of complexity of the function that it can learn. **Small networks, networks with a relatively small number of parameters, have a low capacity and are therefore likely to be under fit.**
- **Very large networks may more easily result in overfitting;**
- When we deal with real-world machine learning problems, we do not know how large the network should be a priori.

Regularizing a neural network with dropout

- To prevent overfitting, we can apply one or multiple regularization schemes to achieve good generalization performance on new data, such as the held-out test set.
- A popular choice for regularization is L2 regularization.
- Another popular regularization technique called **dropout** has emerged that works amazingly well for regularizing (deep) neural networks.

Regularizing a neural network with dropout

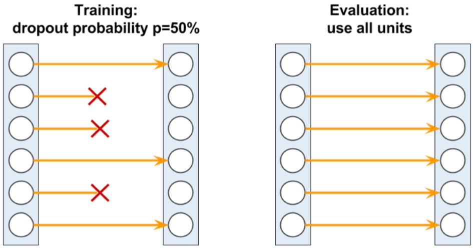
- **Dropout is usually applied to the hidden units of higher layers.**
- During the training phase of a neural network, a fraction of the hidden units is randomly dropped at every iteration with probability P_{drop} (or the keep probability P_{keep} =1 - P_{drop}).
- This dropout probability is determined by the user and the common choice is $p = 0.5$.

Regularizing a neural network with dropout

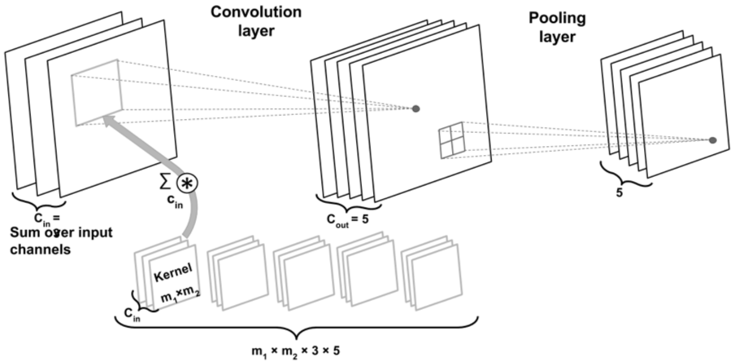
- When dropping a certain fraction of input neurons, the weights associated with the remaining neurons are rescaled to account for the missing (dropped) neurons.
- **The network cannot rely on an activation of any set of hidden units since they may be turned off at any time during training and the network is forced to learn more general and robust patterns from the data.**

Regularizing a neural network with dropout

- It can effectively prevent overfitting.
- An example of applying dropout with probability $p = 0.5$ during the training phase, thereby half of the neurons become inactive randomly.
- **However, during prediction, all neurons will contribute to computing the pre-activations of the next layer.**



Working with multiple input or color channels



How many trainable parameters exist in the example?

Illustrating the advantages of Convolution Layer

- Last figure presents a four-dimensional tensor. So, there are $m_1 x m_2 x 3 x 5$ parameters associated with the kernel.
- There is a bias vector for each output feature map. Thus the size of the bias vector is 5.
- Pooling layers do not have trainable parameters, so $m_1 x m_2 x 3 x 5 + 5$.
- If input tensor is of size $n_1 x n_2 x 3$, supposing we are using **mode='same'** (the size of the output the same as x), then the output feature maps would be $n_1 x n_2 x 5$.

Illustrating the advantages of Convolution Layer

- This number is much smaller than if we have a fully connected layer: $(n_1 x n_2 x 3) x (n_1 x n_2 x 5) = (n_1 x n_2)^2 x 3 x 5$, which is the number of parameters for the weight matrix to reach the same number of output units.
- Given that $m_1 < n_1$ and $m_2 < n_2$, we can see that the difference in the number of trainable parameters is huge.

Developing a model that does better than a baseline in Keras

iPython notebook

Implementing a deep convolutional neural network using TensorFlow

▸ iPython notebook

Using a pretrained Convolutional Model

- Feature Extraction
- Fine tuning