Convolutional layers

Convolutional layers are the major building blocks used in convolutional neural networks.

A convolution is the simple application of a filter to an input that results in an activation. Repeated application of the same filter to an input results in a map of activations called a feature map, indicating the locations and strength of a detected feature in an input, such as an image.

The innovation of convolutional neural networks is the ability to automatically learn a large number of filters in parallel specific to a training dataset under the constraints of a specific predictive modeling problem, such as image classification. The result is highly specific features that can be detected anywhere on input images.

In this tutorial, you will discover how convolutions work in the convolutional neural network.

After completing this tutorial, you will know:

- Convolutional neural networks apply a filter to an input to create a feature map that summarizes the presence of detected features in the input.
- Filters can be handcrafted, such as line detectors, but the innovation of convolutional neural networks is to learn the filters during training in the context of a specific prediction problem.
- How to calculate the feature map for one- and two-dimensional convolutional layers in a convolutional neural network.

Topic Overview

This topic is divided into four parts; they are:

Convolution in Convolutional Neural Networks

The convolutional neural network, or CNN for short, is a specialized type of neural network model designed for working with two-dimensional image data, although they can be used with one-dimensional and three-dimensional data.

Central to the convolutional neural network is the convolutional layer that gives the network its name. This layer performs an operation called a "*convolution*".

In the context of a convolutional neural network, a convolution is a linear operation that involves the multiplication of a set of weights with the input, much like a traditional neural network. Given that the technique was designed for two-dimensional input, the multiplication is performed between an array of input data and a two-dimensional array of weights, called a filter or a kernel.

The filter is smaller than the input data and the type of multiplication applied between a filter-sized patch of the input and the filter is a dot product. A dot product is the element-wise multiplication between the filter-sized patch of the input and filter, which is then summed, always resulting in a single value. Because it results in a single value, the operation is often referred to as the "*scalar product*".

Using a filter smaller than the input is intentional as it allows the same filter (set of weights) to be multiplied by the input array multiple times at different points on the input. Specifically, the filter is applied systematically to each overlapping part or filter-sized patch of the input data, left to right, top to bottom.

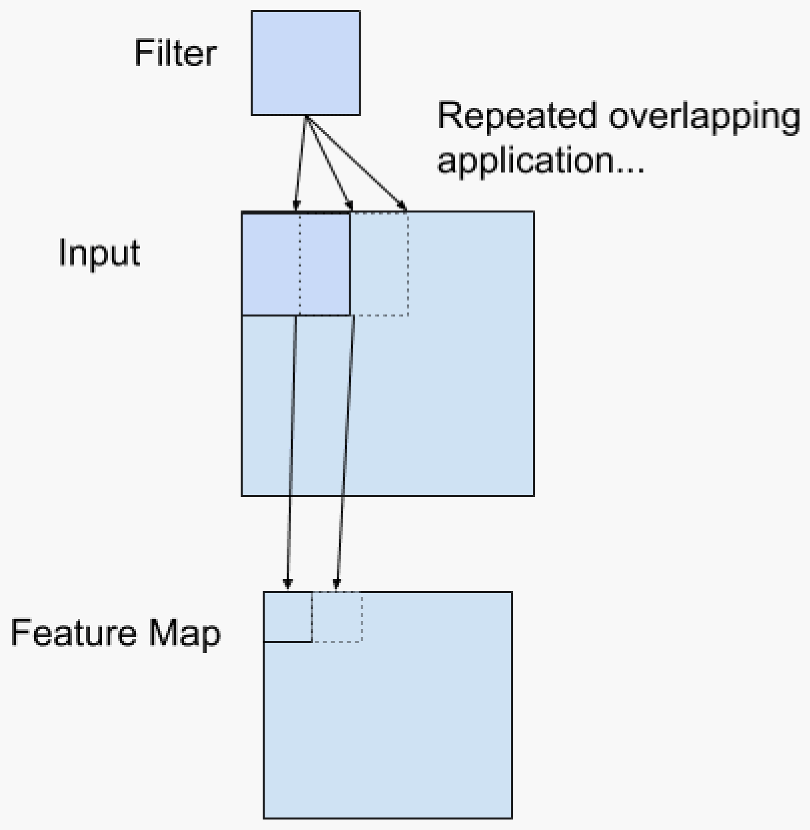
This systematic application of the same filter across an image is a powerful idea. If the filter is designed to detect a specific type of feature in the input, then the application of that filter systematically across the entire input image allows the filter an opportunity to discover that feature anywhere in the image. This capability is commonly referred to as translation invariance, e.g. the general interest in whether the feature is present rather than where it was present.

Invariance to local translation can be a very useful property if we care more about whether some feature is present than exactly where it is. For example, when determining whether an image contains a face, we need not know the location of the eyes with pixel-perfect *accuracy, we just need to know that there is an eye on the left side of the face and an eye on the right side of the face.*

— Page 342, Deep Learning, 2016.

The output from multiplying the filter with the input array one time is a single value. As the filter is applied multiple times to the input array, the result is a two-dimensional array of output values that represent a filtering of the input. As such, the two-dimensional output array from this operation is called a "*feature map*".

Once a feature map is created, we can pass each value in the feature map through a nonlinearity, such as a ReLU, much like we do for the outputs of a fully connected layer.



Example of a Filter Applied to a Two-Dimensional Input to Create a Feature Map

If you come from a digital signal processing field or related area of mathematics, you may understand the convolution operation on a matrix as something different. Specifically, the filter (kernel) is flipped prior to being applied to the input. Technically, the convolution as described in the use of convolutional neural networks is actually a “*cross-correlation”*. Nevertheless, in deep learning, it is referred to as a “*convolution*” operation.

Many machine learning libraries implement cross-correlation but call it convolution.

— Page 333, Deep Learning, 2016.

In summary, we have a *input*, such as an image of pixel values, and we have a *filter*, which is a set of weights, and the filter is systematically applied to the input data to create a *feature map*.

Convolution in Computer Vision

The idea of applying the convolutional operation to image data is not new or unique to convolutional neural networks; it is a common technique used in computer vision.

Historically, filters were designed by hand by computer vision experts, which were then applied to an image to result in a feature map or output from applying the filter then makes the analysis of the image easier in some way.

For example, below is a hand crafted 3×3 element filter for detecting vertical lines:

```
0.0, 1.0, 0.0
0.0, 1.0, 0.0
0.0, 1.0, 0.0
```

Applying this filter to an image will result in a feature map that only contains vertical lines. It is a vertical line detector.

You can see this from the weight values in the filter; any pixels values in the center vertical line will be positively activated and any on either side will be negatively activated. Dragging this filter systematically across pixel values in an image can only highlight vertical line pixels.

A horizontal line detector could also be created and also applied to the image, for example:

```
0.0, 0.0, 0.0
1.0, 1.0, 1.0
```

```
0.0, 0.0, 0.0
```

Combining the results from both filters, e.g. combining both feature maps, will result in all of the lines in an image being highlighted.

A suite of tens or even hundreds of other small filters can be designed to detect other features in the image.

The innovation of using the convolution operation in a neural network is that the values of the filter are weights to be learned during the training of the network.

The network will learn what types of features to extract from the input. Specifically, training under stochastic gradient descent, the network is forced to learn to extract features from the image that minimize the loss for the specific task the network is being trained to solve, e.g. extract features that are the most useful for classifying images as dogs or cats.

In this context, you can see that this is a powerful idea.

Power of Learned Filters

Learning a single filter specific to a machine learning task is a powerful technique.

Yet, convolutional neural networks achieve much more in practice.

Multiple Filters

Convolutional neural networks do not learn a single filter; they, in fact, learn multiple features in parallel for a given input.

For example, it is common for a convolutional layer to learn from 32 to 512 filters in parallel for a given input.

This gives the model 32, or even 512, different ways of extracting features from an input, or many different ways of both "*learning to see*" and after training, many different ways of "*seeing*" the input data.

This diversity allows specialization, e.g. not just lines, but the specific lines seen in your specific training data.

Multiple Channels

Color images have multiple channels, typically one for each color channel, such as red, green, and blue.

From a data perspective, that means that a single image provided as input to the model is, in fact, three images.

A filter must always have the same number of channels as the input, often referred to as "*depth*". If an input image has 3 channels (e.g. a depth of 3), then a filter applied to that image must also have 3 channels (e.g. a depth of 3). In this case, a 3×3 filter would in fact be 3x3x3 or [3, 3, 3] for rows, columns, and depth. Regardless of the depth of the input and depth of the filter, the filter is applied to the input using a dot product operation which results in a single value.

This means that if a convolutional layer has 32 filters, these 32 filters are not just two-dimensional for the two-dimensional image input, but are also three-dimensional, having specific filter weights for each of the three channels. Yet, each filter results in a single feature map. Which means that the depth of the output of applying the convolutional layer with 32 filters is 32 for the 32 feature maps created.

Multiple Layers

Convolutional layers are not only applied to input data, e.g. raw pixel values, but they can also be applied to the output of other layers.

The stacking of convolutional layers allows a hierarchical decomposition of the input.

Consider that the filters that operate directly on the raw pixel values will learn to extract low-level features, such as lines.

The filters that operate on the output of the first line layers may extract features that are combinations of lower-level features, such as features that comprise multiple lines to express shapes.

This process continues until very deep layers are extracting faces, animals, houses, and so on.

This is exactly what we see in practice. The abstraction of features to high and higher orders as the depth of the network is increased.

Worked Example of Convolutional Layers

The Keras deep learning library provides a suite of convolutional layers.

We can better understand the convolution operation by looking at some worked examples with contrived data and handcrafted filters.

In this section, we'll look at both a one-dimensional convolutional layer and a two-dimensional convolutional layer example to both make the convolution operation concrete and provide a worked example of using the Keras layers.

Example of 1D Convolutional Layer

We can define a one-dimensional input that has eight elements all with the value of 0.0, with a two element bump in the middle with the values 1.0.

```
[0, 0, 0, 1, 1, 0, 0, 0]
```

The input to Keras must be three dimensional for a 1D convolutional layer.

The first dimension refers to each input sample; in this case, we only have one sample. The second dimension refers to the length of each sample; in this case, the length is eight. The third dimension refers to the number of channels in each sample; in this case, we only have a single channel.

Therefore, the shape of the input array will be [1, 8, 1].

```
# define input data
data = asarray([0, 0, 0, 1, 1, 0, 0, 0])
data = data.reshape(1, 8, 1)
```

We will define a model that expects input samples to have the shape [8, 1].

The model will have a single filter with the shape of 3, or three elements wide. Keras refers to the shape of the filter as the *kernel_size*.

```
# create model
model = Sequential()
model.add(Conv1D(1, 3, input_shape=(8, 1)))
```

By default, the filters in a convolutional layer are initialized with random weights. In this contrived example, we will manually specify the weights for the single filter. We will define a filter that is capable of detecting bumps, that is a high input value surrounded by low input values, as we defined in our input example.

The three element filter we will define looks as follows:

```
[0, 1, 0]
```

The convolutional layer also has a bias input value that also requires a weight that we will set to zero.

Therefore, we can force the weights of our one-dimensional convolutional layer to use our handcrafted filter as follows:

```
# define a vertical line detector
weights = [asarray([[[0]],[[1]],[[0]]]), asarray([0.0])]
# store the weights in the model
model.set_weights(weights)
```

The weights must be specified in a three-dimensional structure, in terms of rows, columns, and channels. The filter has a single row, three columns, and one channel.

We can retrieve the weights and confirm that they were set correctly.

```
# confirm they were stored
print(model.get_weights())
```

Finally, we can apply the single filter to our input data.

We can achieve this by calling the *predict()* function on the model. This will return the feature map directly: that is the output of applying the filter systematically across the input sequence.

```
# apply filter to input data
```

```
yhat = model.predict(data)
print(yhat)
```

Tying all of this together, the complete example is listed below.

```
# example of calculation 1d convolutions
from numpy import asarray
from keras.models import Sequential
from keras.layers import Conv1D
# define input data
data = asarray([0, 0, 0, 1, 1, 0, 0, 0])
data = data.reshape(1, 8, 1)
# create model
model = Sequential()
model.add(Conv1D(1, 3, input_shape=(8, 1)))
# define a vertical line detector
weights = [asarray([[[0]],[[1]],[[0]]]), asarray([0.0])]
# store the weights in the model
model.set_weights(weights)
# confirm they were stored
print(model.get_weights())
# apply filter to input data
yhat = model.predict(data)
print(yhat)
```

Running the example first prints the weights of the network; that is the confirmation that our handcrafted filter was set in the model as we expected.

Next, the filter is applied to the input pattern and the feature map is calculated and displayed. We can see from the values of the feature map that the bump was detected correctly.

```
[array([[[0.]],
       [[1.]],
       [[0.]]], dtype=float32), array([0.], dtype=float32)]

[[[0.]
  [0.]
  [1.]
  [1.]
  [0.]
  [0.]]]
```

Let’s take a closer look at what happened here.

Recall that the input is an eight element vector with the values: [0, 0, 0, 1, 1, 0, 0, 0].

First, the three-element filter [0, 1, 0] was applied to the first three inputs of the input [0, 0, 0] by calculating the dot product (“.” operator), which resulted in a single output value in the feature map of zero.

Recall that a dot product is the sum of the element-wise multiplications, or here it is (0 x 0) + (1 x 0) + (0 x 0) = 0. In NumPy, this can be implemented manually as:

```
from numpy import asarray
print(asarray([0, 1, 0]).dot(asarray([0, 0, 0])))
```

In our manual example, this is as follows:

```
[0, 1, 0] . [0, 0, 0] = 0
```

The filter was then moved along one element of the input sequence and the process was repeated; specifically, the same filter was applied to the input sequence at indexes 1, 2, and 3, which also resulted in a zero output in the feature map.

```
[0, 1, 0] . [0, 0, 1] = 0
```

We are being systematic, so again, the filter is moved along one more element of the input and applied to the input at indexes 2, 3, and 4. This time the output is a value of one in the feature map. We detected the feature and activated appropriately.

```
[0, 1, 0] . [0, 1, 1] = 1
```

The process is repeated until we calculate the entire feature map.

```
[0, 0, 1, 1, 0, 0]
```

Note that the feature map has six elements, whereas our input has eight elements. This is an artefact of how the filter was applied to the input sequence. There are other ways to apply the filter to the input sequence that changes the shape of the resulting feature map, such as padding, but we will not discuss these methods in this post.

You can imagine that with different inputs, we may detect the feature with more or less intensity, and with different weights in the filter, that we would detect different features in the input sequence.

Example of 2D Convolutional Layer

We can expand the bump detection example in the previous section to a vertical line detector in a two-dimensional image.

Again, we can constrain the input, in this case to a square 8×8 pixel input image with a single channel (e.g. grayscale) with a single vertical line in the middle.

```
[0, 0, 0, 1, 1, 0, 0, 0]
[0, 0, 0, 1, 1, 0, 0, 0]
[0, 0, 0, 1, 1, 0, 0, 0]
[0, 0, 0, 1, 1, 0, 0, 0]
[0, 0, 0, 1, 1, 0, 0, 0]
[0, 0, 0, 1, 1, 0, 0, 0]
[0, 0, 0, 1, 1, 0, 0, 0]
[0, 0, 0, 1, 1, 0, 0, 0]
```

The input to a Conv2D layer must be four-dimensional.

The first dimension defines the samples; in this case, there is only a single sample. The second dimension defines the number of rows; in this case, eight. The third dimension defines the number of columns, again eight in this case, and finally the number of channels, which is one in this case.

Therefore, the input must have the four-dimensional shape [samples, rows, columns, channels] or [1, 8, 8, 1] in this case.

```
# define input data
data = [[0, 0, 0, 1, 1, 0, 0, 0],
[0, 0, 0, 1, 1, 0, 0, 0],
[0, 0, 0, 1, 1, 0, 0, 0],
[0, 0, 0, 1, 1, 0, 0, 0],
[0, 0, 0, 1, 1, 0, 0, 0],
[0, 0, 0, 1, 1, 0, 0, 0],
[0, 0, 0, 1, 1, 0, 0, 0],
[0, 0, 0, 1, 1, 0, 0, 0]]
data = asarray(data)
data = data.reshape(1, 8, 8, 1)
```

We will define the Conv2D with a single filter as we did in the previous section with the Conv1D example.

The filter will be two-dimensional and square with the shape 3×3. The layer will expect input samples to have the shape [columns, rows, channels] or [8,8,1].

```
# create model
model = Sequential()
model.add(Conv2D(1, (3,3), input_shape=(8, 8, 1)))
```

We will define a vertical line detector filter to detect the single vertical line in our input data.

The filter looks as follows:

```
0, 1, 0
0, 1, 0
0, 1, 0
```

We can implement this as follows:

```
# define a vertical line detector
detector = [[[[0]],[[1]],[[0]]],
            [[[0]],[[1]],[[0]]],
            [[[0]],[[1]],[[0]]]]
weights = [asarray(detector), asarray([0.0])]
# store the weights in the model
model.set_weights(weights)
# confirm they were stored
print(model.get_weights())
```

Finally, we will apply the filter to the input image, which will result in a feature map that we would expect to show the detection of the vertical line in the input image.

```
# apply filter to input data
yhat = model.predict(data)
```

The shape of the feature map output will be four-dimensional with the shape [batch, rows, columns, filters]. We will be performing a single batch and we have a single filter (one filter and one input channel), therefore the output shape is [1, ?, ?, 1]. We can pretty-print the content of the single feature map as follows:

```
for r in range(yhat.shape[1]):
# print each column in the row
print([yhat[0,r,c,0] for c in range(yhat.shape[2])])
```

Tying all of this together, the complete example is listed below.

```
# example of calculation 2d convolutions
from numpy import asarray
from keras.models import Sequential
from keras.layers import Conv2D
# define input data
data = [[0, 0, 0, 1, 1, 0, 0, 0],
[0, 0, 0, 1, 1, 0, 0, 0],
[0, 0, 0, 1, 1, 0, 0, 0],
[0, 0, 0, 1, 1, 0, 0, 0],
[0, 0, 0, 1, 1, 0, 0, 0],
[0, 0, 0, 1, 1, 0, 0, 0],
[0, 0, 0, 1, 1, 0, 0, 0],
[0, 0, 0, 1, 1, 0, 0, 0]]
data = asarray(data)
data = data.reshape(1, 8, 8, 1)
# create model
model = Sequential()
model.add(Conv2D(1, (3,3), input_shape=(8, 8, 1)))
# define a vertical line detector
detector = [[[[0]],[[1]],[[0]]],
            [[[0]],[[1]],[[0]]],
            [[[0]],[[1]],[[0]]]]
weights = [asarray(detector), asarray([0.0])]
# store the weights in the model
model.set_weights(weights)
# confirm they were stored
print(model.get_weights())
# apply filter to input data
yhat = model.predict(data)
for r in range(yhat.shape[1]):
# print each column in the row
print([yhat[0,r,c,0] for c in range(yhat.shape[2])])
```

Running the example first confirms that the handcrafted filter was correctly defined in the layer weights

Next, the calculated feature map is printed. We can see from the scale of the numbers that indeed the filter has detected the single vertical line with strong activation in the middle of the feature map.

```
[array([[[[0.]],
        [[1.]],
        [[0.]]],
       [[[0.]],
        [[1.]],
        [[0.]]],
       [[[0.]],
        [[1.]],
        [[0.]]]], dtype=float32), array([0.], dtype=float32)]

[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
```

Let’s take a closer look at what was calculated.

First, the filter was applied to the top left corner of the image, or an image patch of 3×3 elements. Technically, the image patch is three dimensional with a single channel, and the filter has the same dimensions. We cannot implement this in NumPy using the dot() function, instead, we must use the tensordot() function so we can appropriately sum across all dimensions, for example:

```
from numpy import asarray
from numpy import tensordot
m1 = asarray([[0, 1, 0],
  [0, 1, 0],
  [0, 1, 0]])
m2 = asarray([[0, 0, 0],
  [0, 0, 0],
  [0, 0, 0]])
print(tensordot(m1, m2))
```

This calculation results in a single output value of 0.0, e.g., the feature was not detected. This gives us the first element in the top-left corner of the feature map.

Manually, this would be as follows:

```
0, 1, 0      0, 0, 0
0, 1, 0  .  0, 0, 0 = 0
0, 1, 0      0, 0, 0
```

The filter is moved along one column to the left and the process is repeated. Again, the feature is not detected.

```
0, 1, 0      0, 0, 1
0, 1, 0  .  0, 0, 1 = 0
0, 1, 0      0, 0, 1
```

One more move to the left to the next column and the feature is detected for the first time, resulting in a strong activation.

```
0, 1, 0      0, 1, 1
0, 1, 0  .  0, 1, 1 = 3
0, 1, 0      0, 1, 1
```

This process is repeated until the edge of the filter rests against the edge or final column of the input image. This gives the last element in the first full row of the feature map.

```
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
```

The filter then moves down one row and back to the first column and the process is related from left to right to give the second row of the feature map. And on until the bottom of the filter rests on the bottom or last row of the input image.

Again, as with the previous section, we can see that the feature map is a 6×6 matrix, smaller than the 8×8 input image because of the limitations of how the filter can be applied to the input