

Image Processing with Keras in Python

Image Processing with Keras in Python



Black Raven (James Ng)

18 Mar 2021 · 31 min read

🕒 4 hours ▶ 13 Videos ↔ 45 Exercises 👤 21,137 Participants 📖 3,650 XP

This is a memo to share what I have learnt in Image Processing with Keras (in Python), capturing the learning objectives as well as my personal notes. The course is taught by Ariel Rokem from DataCamp, and it includes 4 chapters:

Chapter 1. Image Processing With Neural Networks

Chapter 2. Using Convolutions

Chapter 3. Going Deeper

Chapter 4. Understanding and Improving Deep Convolutional Networks

Deep learning methods use data to train neural network algorithms to do a variety of machine learning tasks, such as classification of different classes of objects. Convolutional neural networks are deep learning algorithms that are particularly powerful for analysis of images. This course will teach you how to construct, train and evaluate convolutional neural networks. You will also learn how to improve their ability to learn from data, and how to interpret the results of the training.

Chapter 1. Image Processing With Neural Networks

Convolutional neural networks use the data that is represented in images to learn. In this chapter, we will probe data in images, and we will learn how to use Keras to train a neural network to classify objects that appear in images.

Introducing convolutional neural networks

CNNs are powerful algorithms for processing images. In fact, these algorithms are currently the best algorithms we have for automated processing of images.

We will use Keras, which is a Python-based library that implements the building blocks you need to build your own CNNs.

```
import matplotlib.pyplot as plt
data = plt.imread('stop_sign.jpg')
plt.imshow(data)
plt.show()
```

data.shape

(2832, 4256, 3)



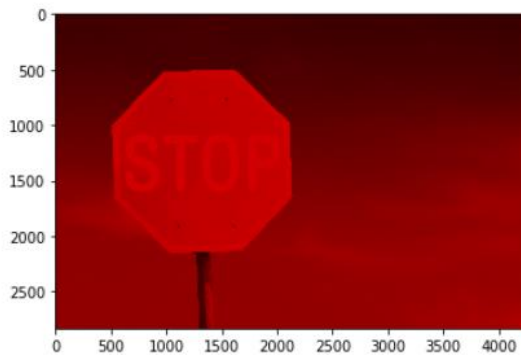
The computer doesn't see the image. All it sees is an array of numbers. Color images are stored in 3-dimensional arrays. The first two dimensions correspond to the height and width of the image (the number of pixels). The last dimension corresponds to the red, green and blue colors present in each pixel.

```
data[1000, 1500]
```

```
array([0.73333333, 0.07843137, 0.14509804])
```



```
data[:, :, 1] = 0  
data[:, :, 2] = 0  
plt.imshow(data)  
plt.show()
```



```
data[250, 3500]
```

```
array([0.25882353, 0.43921569, 0.77254902])
```



We can also change the image by changing the array data. For example, here we set the green and blue values of the pixels to zero. The result is an image that contains only the information in the red channel.

```
data[200:1200, 200:1200, :] = [0, 1, 0]  
plt.imshow(data)  
plt.show()
```



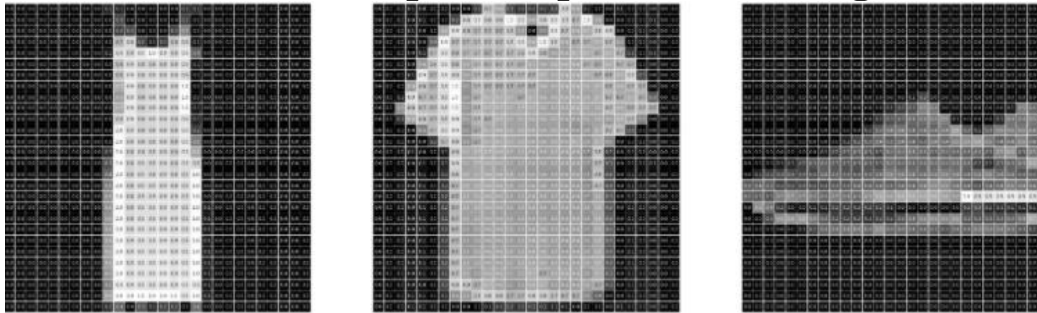
Alternatively, we could set all the pixels within some part of the image to have no red and no blue, but full intensity in the green channel. This results in an image with a green square in it.

In this course, we will mostly look at black and white images. For example, consider these three images of three different items of clothing: a dress, a t-shirt and a shoe.



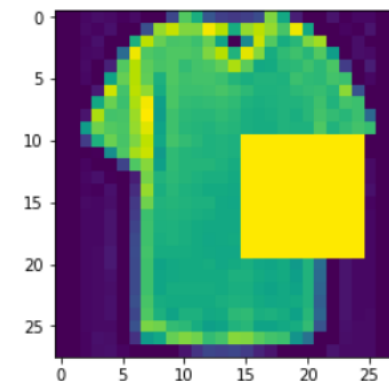
What the computer sees are the numbers that represent the intensity of the image in each pixel.

- High numbers represent parts of the image that are brighter
- Low numbers represent parts of the image that are darker



```
tshirt[10:20, 15:25] = 1  
plt.imshow(tshirt)  
plt.show()
```

In these images as well, we can select a part of the image, using indexing on the rows and columns of the array and slicing, and change part of the array, using assignment.



Images as data: visualizations

To display image data, you will rely on Python's **Matplotlib** library, and specifically use matplotlib's `pyplot` sub-module, that contains many plotting commands. Some of these commands allow you to display the content of images stored in arrays.

```
# Import matplotlib
import matplotlib.pyplot as plt

# Load the image
data = plt.imread('bricks.png')

# Display the image
plt.imshow(data)
plt.show()
```



Visit [Matplotlib's website](https://matplotlib.org/) to learn more about plotting commands you could use.

Images as data: changing images

To modify an image, you can modify the existing numbers in the array. In a color image, you can change the values in one of the color channels without affecting the other colors, by indexing on the last dimension of the array.

The image you imported in the previous exercise is available in `data`.

```
# Set the red channel in this part of the image to 1
data[:10, :10, 0] = 1

# Set the green channel in this part of the image to 0
data[:10, :10, 1] = 0

# Set the blue channel in this part of the image to 0
data[:10, :10, 2] = 0
```

```
# Visualize the result
```

```
plt.imshow(data)
```

```
plt.show()
```

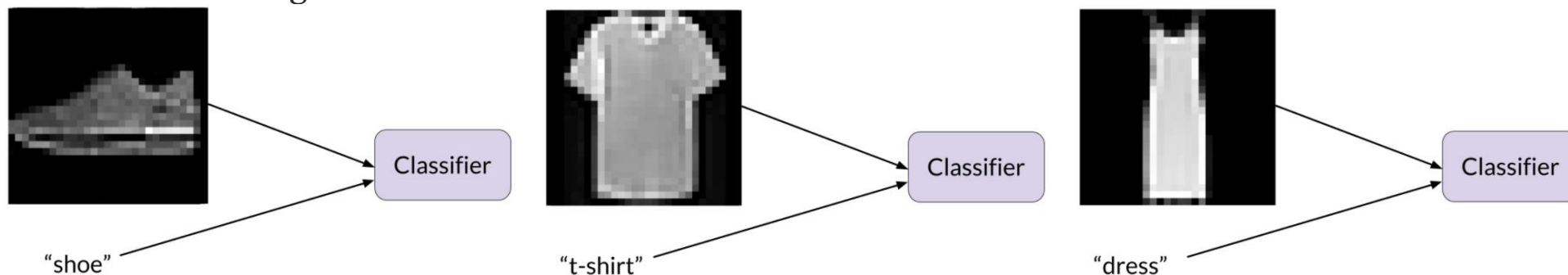


You now know how to manipulate images.

By the way, if you set both the green and red channels to 1, that part of the image would be yellow.

Classifying images

In the training phase, we present the algorithm with samples from these three classes, together with the class labels for each image.



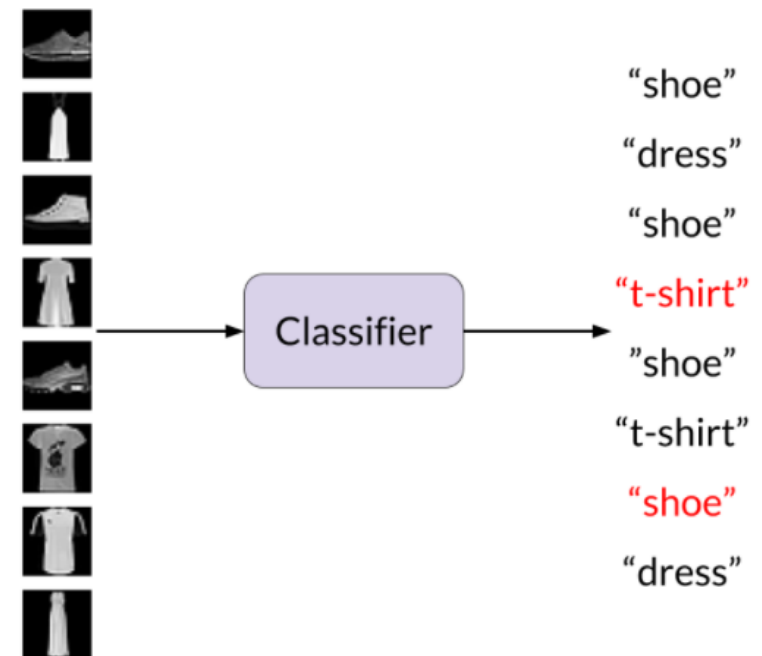
Over the course of training, the algorithm adjusts its parameters to learn the patterns in the data that distinguish between the three different classes of clothing.

At the end of training, we would like to know how well our classifier does. To avoid an estimate that is overly optimistic because of overfitting, we evaluate it by testing it on a portion of the data that has been set aside in advance for this purpose.

In this case, the classifier is able to correctly classify some of the images, but incorrectly classifies an image of a dress as a t-shirt, and an image of a t-shirt as a shoe.

Representing class data: one-hot encoding

```
labels = ["shoe", "dress", "shoe", "t-shirt",  
         "shoe", "t-shirt", "shoe", "dress"]  
  
array([[0., 0., 1.], <= shoe  
       [0., 1., 0.], <= dress  
       [0., 0., 1.], <= shoe  
       [1., 0., 0.], <= t-shirt  
       [0., 0., 1.], <= shoe  
       [1., 0., 0.], <= t-shirt  
       [0., 0., 1.], <= shoe  
       [0., 1., 0.]]) <= dress  
  
categories = np.array(["t-shirt", "dress", "shoe"])  
n_categories = 3  
ohe_labels = np.zeros((len(labels), n_categories))  
for ii in range(len(labels)):  
    jj = np.where(categories == labels[ii])  
    ohe_labels[ii, jj] = 1
```



```
test
```

```
array([[0., 0., 1.],  
       [0., 1., 0.],  
       [0., 0., 1.],  
       [0., 1., 0.],  
       [0., 0., 1.],  
       [0., 0., 1.],  
       [0., 0., 1.],  
       [0., 1., 0.]])
```

```
prediction
```

```
array([[0., 0., 1.],  
       [0., 1., 0.],  
       [0., 0., 1.],  
       [1., 0., 0.], <= incorrect  
       [0., 0., 1.],  
       [1., 0., 0.], <= incorrect  
       [0., 0., 1.],  
       [0., 1., 0.]])
```

```
(test * prediction).sum()
```

```
6.0
```

Using one-hot encoding to represent images

Neural networks expect the labels of classes in a dataset to be organized in a one-hot encoded manner: each row in the array contains zeros in all columns, except the column corresponding to a unique label, which is set to 1.

The fashion dataset contains three categories:

1. Shirts
2. Dresses
3. Shoes

In this exercise, you will create a one-hot encoding of a small sample of these labels.

```
# The number of image categories  
n_categories = 3  
  
# The unique values of categories in the data
```



```
categories = np.array(["shirt", "dress", "shoe"])

# Initialize ohe_labels as all zeros
ohe_labels = np.zeros((len(labels), n_categories))

# Loop over the labels
for ii in range(len(labels)):
    # Find the location of this label in the categories variable
    jj = np.where(categories == labels[ii])
    # Set the corresponding zero to one
    ohe_labels[ii, jj] = 1

labels: ['shoe', 'shirt', 'shoe', 'shirt', 'dress', 'dress', 'dress']
```

You can use this array to test classification performance.

Evaluating a classifier

To evaluate a classifier, we need to test it on images that were not used during training. This is called "cross-validation": a prediction of the class (e.g., t-shirt, dress or shoe) is made from each of the test images, and these predictions are compared with the true labels of these images.

The results of cross-validation are provided as one-hot encoded arrays: `test_labels` and `predictions`.

```
# Calculate the number of correct predictions
number_correct = (test_labels * predictions).sum()
print(number_correct)

# Calculate the proportion of correct predictions
proportion_correct = number_correct/len(predictions)
print(proportion_correct)
```

```
<script.py> output:
6.0
0.75
```

Let's talk about fitting classification models using Keras.

Classification with Keras

```
from keras.models import Sequential
model = Sequential()
```

fully connected network

```
from keras.layers import Dense
train_data.shape
```

```
(50, 28, 28, 1)
```

50 samples, each of 28 by 28 pixels (784), 1=images are black and white

```
model.add(Dense(10, activation='relu',
                input_shape=(784,)))
model.add(Dense(10, activation='relu'))
model.add(Dense(3, activation='softmax'))
```

input, hidden, output

```
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

The model expects samples to be rows in an array, and each column to represent a pixel in the image, so before we fit the model we need to convert the images into a two-dimensional table, using the "reshape" method.

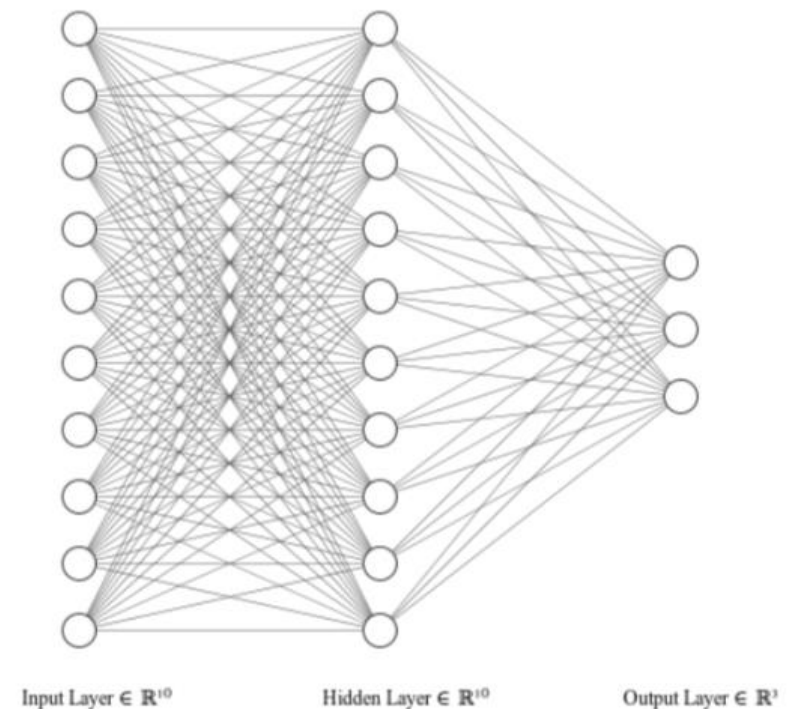
```
train_data = train_data.reshape((50, 784))
```

```
model.fit(train_data, train_labels,
          validation_split=0.2,
          epochs=3)
```

Here, the model will run for 3 epochs, meaning that it will go over all of the training data three times. To avoid overfitting, we set aside 20% as a set of validation images.

```
test_data = test_data.reshape((10, 784))
model.evaluate(test_data, test_labels)
```

```
10/10 [=====] - 0s 335us/step
[1.0191701650619507, 0.4000000059604645]
```



Train on 40 samples, validate on 10 samples

Epoch 1/3

32/40 [=====>.....] - ETA: 0s - loss: 1.0117 - acc: 0.4688

40/40 [=====] - 0s 4ms/step - loss: 1.0438 - acc: 0.4250
- val_loss: 0.9668 - val_acc: 0.4000

Epoch 2/3

32/40 [=====>.....] - ETA: 0s - loss: 0.9556 - acc: 0.5312

40/40 [=====] - 0s 195us/step - loss: 0.9404 - acc: 0.5750
- val_loss: 0.9068 - val_acc: 0.4000

Epoch 3/3

32/40 [=====>.....] - ETA: 0s - loss: 0.9143 - acc: 0.5938

40/40 [=====] - 0s 189us/step - loss: 0.8726 - acc: 0.6750
- val_loss: 0.8452 - val_acc: 0.4000

Accuracy is still rather low, because we used a small amount of data, and we trained for only a few epochs. Final evaluation of the model should be done on a separate test set that was not used during training. This gives us a separate realistic evaluation of the model quality.

Build a neural network

We will use the `Keras` library to create neural networks and to train these neural networks to classify images. These models will all be of the `Sequential` type, meaning that the outputs of one layer are provided as inputs only to the next layer.

In this exercise, you will create a neural network with `Dense` layers, meaning that each unit in each layer is connected to all of the units in the previous layer. For example, each unit in the first layer is connected to all of the pixels in the input images. The `Dense` layer object receives as arguments the number of units in that layer, and the activation function for the units. For the first layer in the network, it also receives an `input_shape` keyword argument.

This course touches on a lot of concepts you may have forgotten, so if you ever need a quick refresher, download the [Keras Cheat Sheet](#) and keep it handy!

```
# Imports components from Keras
from keras.models import Sequential
from keras.layers import Dense

# Initializes a sequential model
model = Sequential()

# First layer
model.add(Dense(10, activation='relu', input_shape=(784,)))

# Second layer
model.add(Dense(10, activation='relu'))

# Output layer
model.add(Dense(3, activation='softmax'))
```

You've built a neural network!

Compile a neural network

Once you have constructed a model in `Keras`, the model needs to be compiled before you can fit it to data. This means that you need to specify the optimizer that will be used to fit the model and the loss function that will be used in optimization. Optionally, you can also specify a list of metrics that the model will keep track of. For example, if you want to know the classification accuracy, you will provide the list `['accuracy']` to the `metrics` keyword argument.

```
# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

This neural network model is now ready to be fit to data.

Fitting a neural network model to clothing data

In this exercise, you will fit the fully connected neural network that you constructed in the previous exercise to image data. The training data is provided as two variables: `train_data` that contains the pixel data for 50 images of the three clothing classes and `train_labels`, which contains one-hot encoded representations of the labels for each one of these 50 images. Transform the data into the network's expected input and then fit the model on training data and training labels.

The `model` you compiled in the previous exercise, and `train_data` and `train_labels` are available in your workspace.

```
# Reshape the data to two-dimensional array
```

```
train_data = train_data.reshape(50, 784)
```

```
# Fit the model
```

```
model.fit(train_data, train_labels, validation_split=0.2, epochs=3)
```

```
<script.py> output:
```

```
Train on 40 samples, validate on 10 samples
```

```
Epoch 1/3
```

```
32/40 [=====>.....] - ETA: 0s - loss: 1.0043 - acc: 0.5000
```

```
40/40 [=====] - 0s 7ms/step - loss: 1.0062 - acc: 0.5000 - val_loss: 0.9919 - val_acc: 0.4000
```

```
Epoch 2/3
```

```
32/40 [=====>.....] - ETA: 0s - loss: 0.9582 - acc: 0.5625
40/40 [=====] - 0s 213us/step - loss: 0.9444 - acc: 0.6000 - val_loss: 0.9596 - val_acc: 0.4000
Epoch 3/3

32/40 [=====>.....] - ETA: 0s - loss: 0.8760 - acc: 0.5625
40/40 [=====] - 0s 205us/step - loss: 0.8938 - acc: 0.5500 - val_loss: 0.9225 - val_acc: 0.4000
```

This model works well on the training data, but does it work well on test data?

Cross-validation for neural network evaluation

To evaluate the model, we use a separate test data-set. As in the train data, the images in the test data also need to be reshaped before they can be provided to the fully-connected network because the network expects one column per pixel in the input.

The `model` you fit in the previous exercise, and `test_data` and `test_labels` are available in your workspace.

```
# Reshape test data
test_data = test_data.reshape(10, 784)

# Evaluate the model
model.evaluate(test_data, test_labels)

<script.py> output:

10/10 [=====] - 0s 210us/step
```

The model cross-validates rather accurately.

Chapter 2. Using Convolutions

Convolutions are the fundamental building blocks of convolutional neural networks. In this chapter, you will be introduced to convolutions and learn how they operate on image data. You will also see how you incorporate convolutions into Keras neural networks.

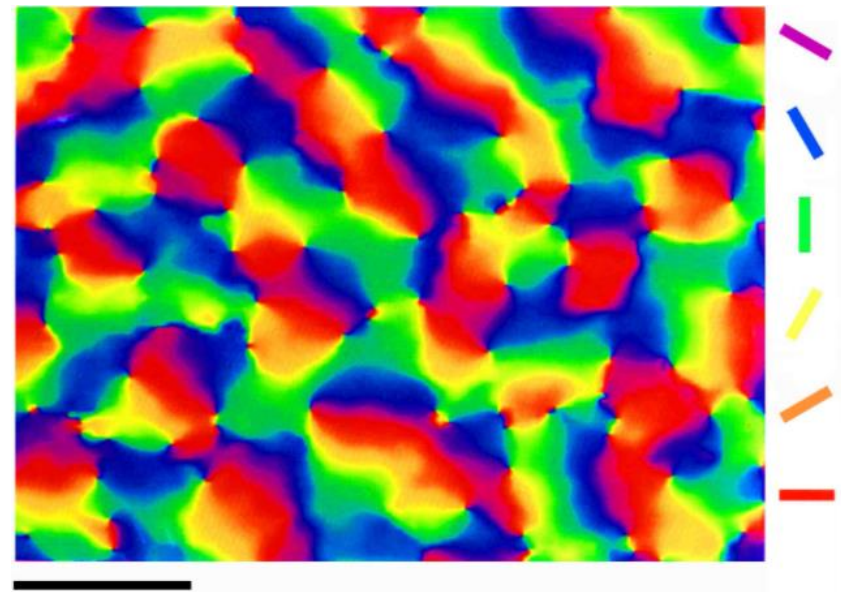
Convolutions

In the neural network that we previously constructed, each unit in the first layer had a weight connecting it separately with every pixel in the image.

Pixels in most images are not independent from their neighbors, ie, spatial correlations. For example, images of objects contain edges, and neighboring pixels along an edge tend to have similar patterns.

Our own visual system uses these correlations, and each nerve cell in the visual areas in our brain responds to oriented edges at a particular location in the visual field.

This image depicts a small part of the visual cortex (the scale bar is 1 millimeter in size). Each part of the image responds to some part of the visual field, and to the orientation depicted by the colors on the right. Looking for the same feature, such as a particular orientation, in every location in an image is like a mathematical operation called a convolution. This is the fundamental operation that convolutional neural networks use to process images.



Let's start with a simple version: a convolution in one dimension. We create here an array that contains 5 zeros followed by 5 ones. This array contains an "edge" in the middle, where the values go from zero to one.

```
array = np.array([0, 0, 0, 0, 0, 1, 1, 1, 1, 1])
kernel = np.array([-1, 1])
```

The kernel defines the feature that we are looking for. In this case, we are looking for a change from small values on the left to large values on the right. We start the result as all zeros. Then, we slide the kernel along the array. In each location we multiply the values in the array with the values in the Kernel and sum them up.

```
conv = np.array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
conv[0] = (kernel * array[0:2]).sum()
conv[1] = (kernel * array[1:3]).sum()
conv[2] = (kernel * array[2:4]).sum()
...
for ii in range(8):
    conv[ii] = (kernel * array[ii:ii+2]).sum()
```

This is the result for that location:

```
conv
```

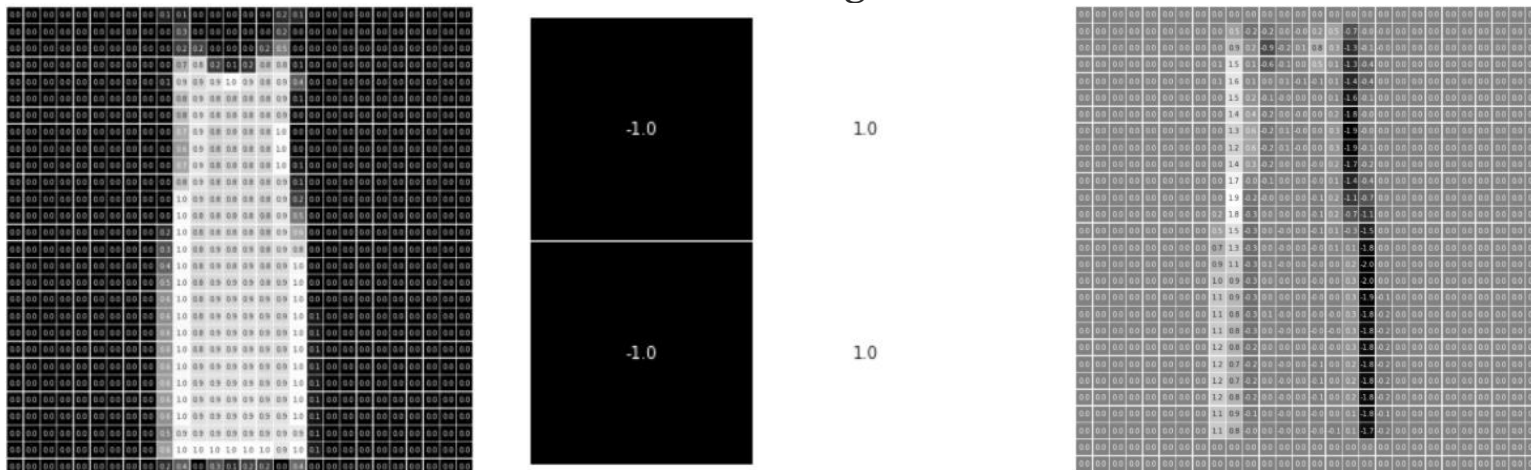
```
array([0, 0, 0, 0, 1, 0, 0, 0, 0, 0])
```

In this example, the array goes between 0 and 1 twice. In this case, the edges that go from zero to one match the kernel, but the edges from 1 to 0 are the opposite of the kernel. In these locations, the convolution becomes negative.

```
array = np.array([0, 0, 1, 1, 0, 0, 1, 1, 0, 0])
kernel = np.array([-1, 1])
conv = np.array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
for ii in range(8):
    conv[ii] = (kernel * array[ii:ii+2]).sum()
conv
```

```
array([ 0,  1,  0, -1,  0,  1,  0, -1,  0])
```


Convolutions of images do the same operation, but in two dimensions. In this case, we convolve the image of a dress with a kernel that matches vertical edges on the left.



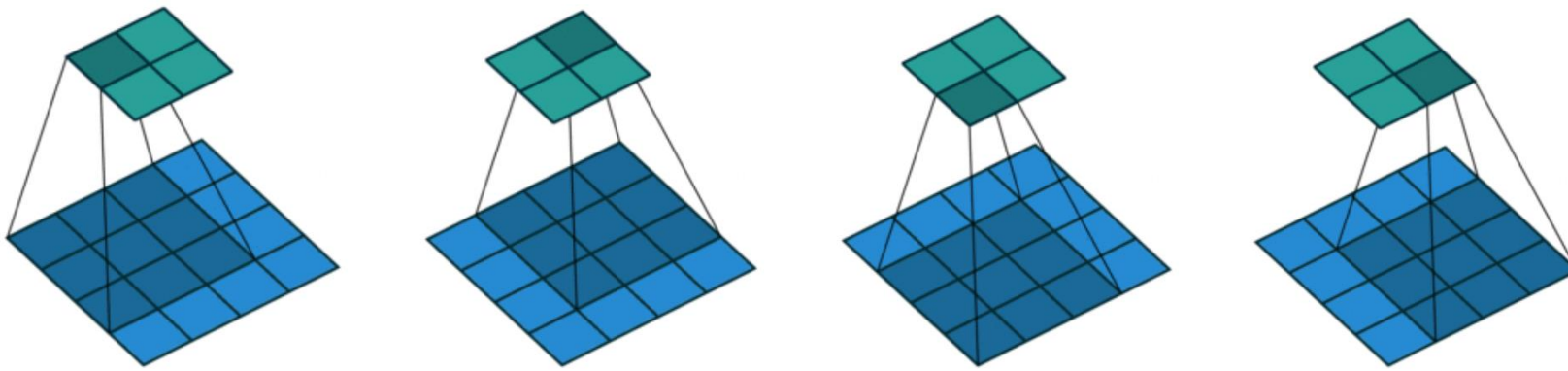
This means that when we convolve the image with this kernel, the left edge is emphasized. The right side of the dress is the opposite of this kernel, and the convolution is negative there.

```
kernel = np.array([[-1, 1],
                   [-1, 1]])

conv = np.zeros((27, 27)
for ii in range(27):
    for jj in range(27):
        window = image[ii:ii+2, jj:jj+2]
        conv[ii, jj] = np.sum(window * kernel)
```

First, we create the kernel. Then, we create the array that will store the results of the convolution. We iterate over all the locations in the image. In each location, we select a window that is the size of the kernel, we multiply that window with the kernel, and then sum it up. This sum is then entered as the value of the convolved image in that location. At the end of the loop, the array will contain the results of the convolution.

Here is a graphic that demonstrates the convolution operation. The kernel is the gray 3-by-3 box that slides over the blue input image at the bottom.



In each location, the window is multiplied with the values in the kernel and added up to create the value of one of the pixels in the resulting green array at the top. In neural networks, we call this resulting array a "feature map", because it contains a map of the locations in the image that match the feature represented by this kernel.

One dimensional convolutions

A convolution of an one-dimensional array with a kernel comprises of taking the kernel, sliding it along the array, multiplying it with the items in the array that overlap with the kernel in that location and summing this product.

```
array = np.array([1, 0, 1, 0, 1, 0, 1, 0, 1, 0])
kernel = np.array([1, -1, 0])
conv = np.array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

```
# Output array
for ii in range(8):
    conv[ii] = (kernel * array[ii:ii+3]).sum()
```

```
# Print conv
print(conv)
```

```
<script.py> output:
[ 1 -1  1 -1  1 -1  0  0]
```

Notice that we've only multiplied the kernel with eight different positions

Image convolutions

The convolution of an image with a kernel summarizes a part of the image as the sum of the multiplication of that part of the image with the kernel. In this exercise, you will write the code that executes a convolution of an image with a kernel using Numpy. Given a black and white image that is stored in the variable `im`, write the operations inside the loop that would execute the convolution with the provided kernel.

```
kernel = np.array([[0, 1, 0], [1, 1, 1], [0, 1, 0]])
result = np.zeros(im.shape)

# Output array
for ii in range(im.shape[0] - 3):
    for jj in range(im.shape[1] - 3):
        result[ii, jj] = (im[ii:ii+3, jj:jj+3] * kernel).sum()

# Print result
print(result)
```

<script.py> output:

```
[[2.68104586 2.95947725 2.84313735 ... 0.      0.      0.      ]
 [3.01830077 3.07058835 3.05098051 ... 0.      0.      0.      ]
 [2.95163405 3.09934652 3.20261449 ... 0.      0.      0.      ]
 ...
 [0.         0.         0.         ... 0.      0.      0.      ]
 [0.         0.         0.         ... 0.      0.      0.      ]
 [0.         0.         0.         ... 0.      0.      0.      ]]
```

In a future exercise, you will see how to use Keras to implement a convolution like this one.

Defining image convolution kernels

In the previous exercise, you wrote code that performs a convolution given an image and a kernel. This code is now stored in a function called `convolution()` that takes two inputs: `image` and `kernel` and produces the convolved image. In this exercise, you will be asked to define the kernel that finds a particular feature in the image.

For example, the following kernel finds a vertical line in images:

```
np.array([[ -1,  1, -1],
          [-1,  1, -1],
          [-1,  1, -1]])
```

Define a kernel that finds horizontal lines in images.

```
kernel = np.array([[ -1, -1, -1],
                   [ 1,  1,  1],
                   [-1, -1, -1]])
```

Define a kernel that finds a light spot surrounded by dark pixels.

```
kernel = np.array([[ -1, -1, -1],
                   [-1,  1, -1],
                   [-1, -1, -1]])
```

Define a kernel that finds a dark spot surrounded by bright pixels.

```
kernel = np.array([[ 1,  1,  1],
                   [ 1, -1,  1],
                   [ 1,  1,  1]])
```

Implementing image convolutions in Keras

Keras has objects to represent convolution layers.

```
from keras.layers import Conv2D
Conv2D(10, kernel_size=3, activation='relu')
```

The two-dimensional convolution connects to the previous layer through a convolution kernel. This means that the output of each unit in this layer is a convolution of a kernel over the image input. Here, we have ten convolution units.

During training of a neural network that has convolutional layers, the kernels in each unit would be adjusted using back-propagation. The principle is the same as learning in the Dense, but with fewer weights. A dense layer has one weight for each *pixel* in the image, but a convolution layer has only one weight for each pixel in the *kernel*. For example, if we set the `kernel_size` argument to 3, that means that the kernel of each unit has 9 pixels. If the layer has 10 units, it would have 90 parameters for these kernels.

```

from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten
model = Sequential()
model.add(Conv2D(10, kernel_size=3, activation='relu',
                 input_shape=(img_rows, img_cols, 1)))
model.add(Flatten())
model.add(Dense(3, activation='softmax'))
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
train_data.shape

```

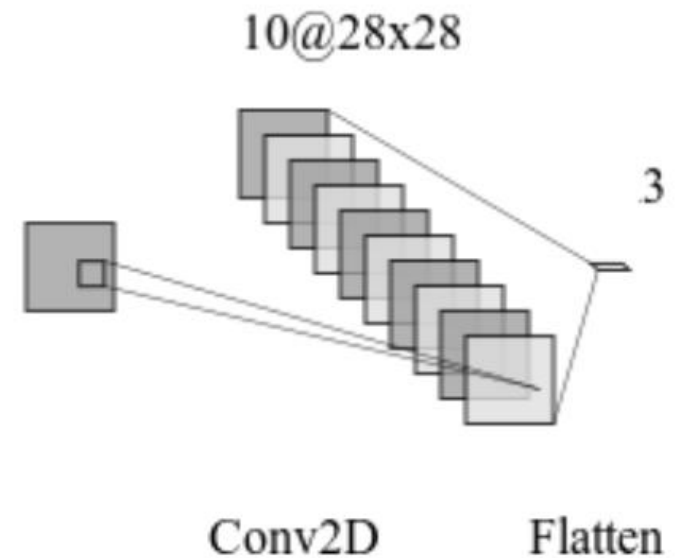
```
(50, 28, 28, 1)
```

```

model.fit(train_data, train_labels, validation_split=0.2,
          epochs=3)

model.evaluate(test_data, test_labels, epochs=3)

```



Convolutional network for image classification

Convolutional networks for classification are constructed from a sequence of convolutional layers (for image processing) and fully connected (`Dense`) layers (for readout). In this exercise, you will construct a small convolutional network for classification of the data from the fashion dataset.

```

# Import the necessary components from Keras
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten

```

```
# Initialize the model object
model = Sequential()

# Add a convolutional layer
model.add(Conv2D(10, kernel_size=3, activation='relu',
                 input_shape=(img_rows, img_cols, 1)))

# Flatten the output of the convolutional layer
model.add(Flatten())

# Add an output layer for the 3 categories
model.add(Dense(3, activation='softmax'))
```

Training a CNN to classify clothing types

Before training a neural network it needs to be compiled with the right cost function, using the right optimizer. During compilation, you can also define metrics that the network calculates and reports in every epoch. Model fitting requires a training data set, together with the training labels to the network. The Conv2D `model` you built in the previous exercise is available in your workspace.

```
# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Fit the model on a training set
model.fit(train_data, train_labels,
          validation_split=0.2,
          epochs=3, batch_size=10)
```

<script.py> output:

```
Train on 40 samples, validate on 10 samples
Epoch 1/3
```

```
10/40 [=====>.....] - ETA: 0s - loss: 1.1084 - acc: 0.3000
```

```
40/40 [=====] - 0s 7ms/step - loss: 0.9589 - acc: 0.4750 - val_loss: 0.6300 - val_acc: 0.9000
```

```
Epoch 2/3
```

```
10/40 [=====>.....] - ETA: 0s - loss: 0.7656 - acc: 0.9000  
40/40 [=====] - 0s 477us/step - loss: 0.6722 - acc: 0.8500 - val_loss: 0.4497 - val_acc: 1.0000  
Epoch 3/3
```

```
10/40 [=====>.....] - ETA: 0s - loss: 0.5367 - acc: 1.0000  
40/40 [=====] - 0s 473us/step - loss: 0.4332 - acc: 0.9750 - val_loss: 0.3750 - val_acc: 1.0000
```

Validation accuracy converges to 100%!

Evaluating a CNN with test data

To evaluate a trained neural network, you should provide a separate testing data set of labeled images. The `model` you fit in the previous exercise is available in your workspace.

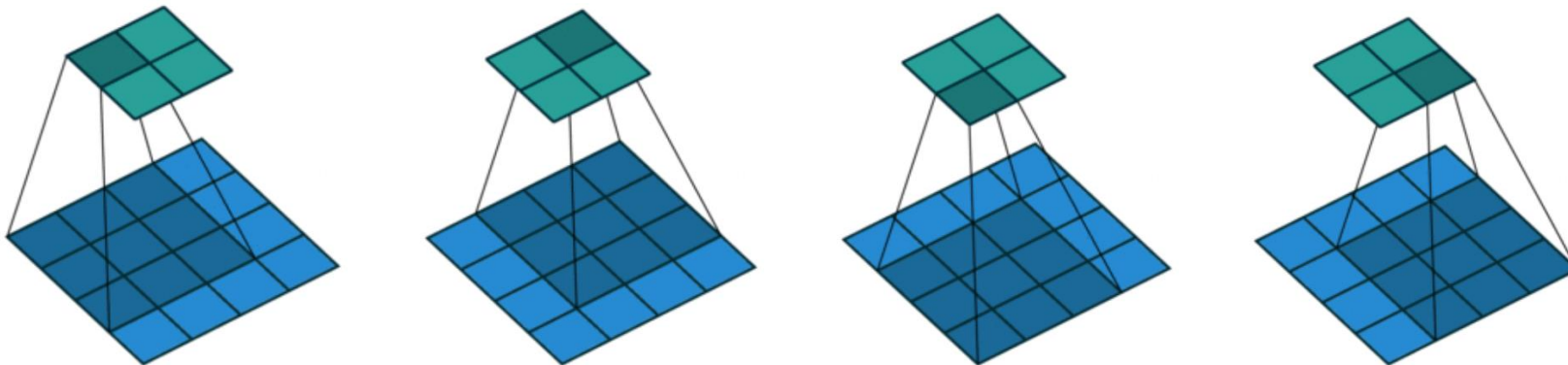
```
# Evaluate the model on separate test data  
model.evaluate(test_data, test_labels, batch_size=10)
```

<script.py> output:

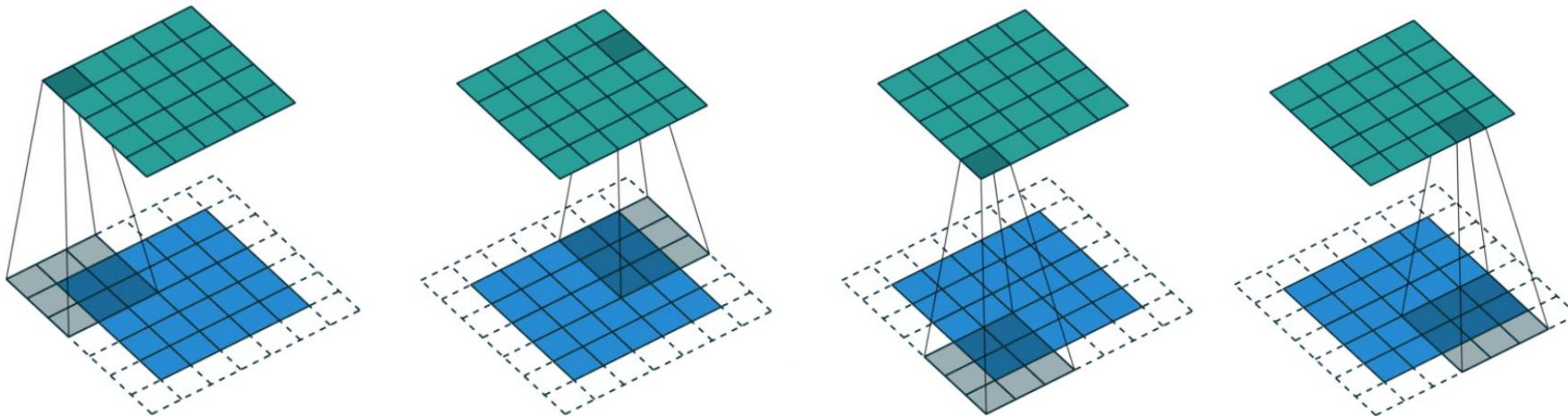
```
10/10 [=====] - 0s 295us/step
```

The first number in the output is the value of the cross-entropy loss, the second is the value of the accuracy. For this model, it's 100%!

Tweaking your convolutions



The blue input image is larger than the green output image. This is because the convolution kernel has the size of three-by-three pixels. In this case, it converts a three-by-three window into one pixel in the output image. One way to deal with this issue is to zero-pad the input image.



When the input image is **zero padded**, the output feature map has the same size as the input. This can be useful if you want to build a network that has many layers. Otherwise, you might lose a pixel off the edge of the image in each subsequent layer.

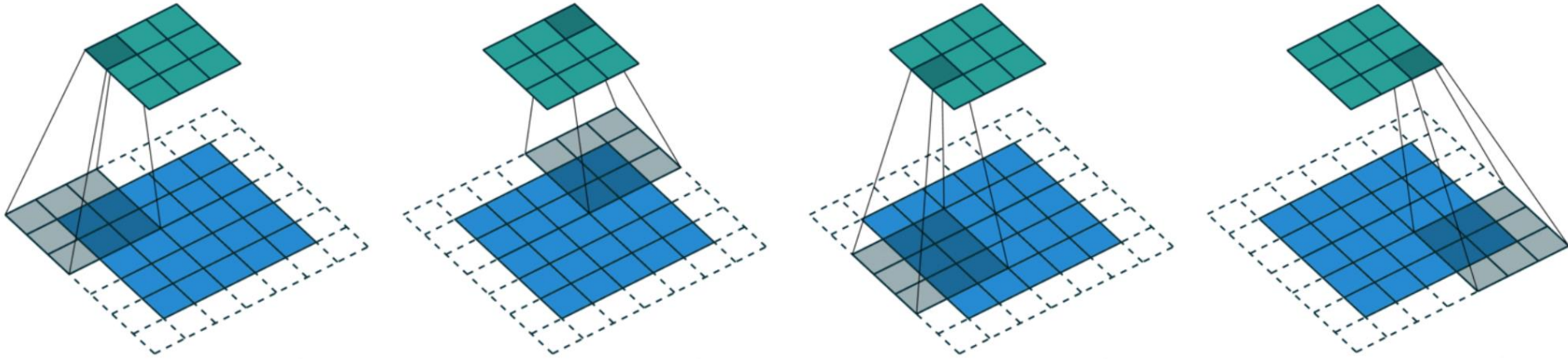
```
model.add(Conv2D(10, kernel_size=3, activation='relu',  
                input_shape=(img_rows, img_cols, 1)),  
          padding='valid')
```

If we provide the value "valid", no zero padding is added (default).

```
model.add(Conv2D(10, kernel_size=3, activation='relu',  
                input_shape=(img_rows, img_cols, 1)),  
          padding='same')
```

If we provide the value "same", zero padding will be applied to the input to this layer, so that the output of the convolution has the same size as the input into the convolution.

Another factor that affects the size of the output of a convolution is the size of the step that we take with the kernel between input pixels. This is called the **size of the stride**. For example, in this animation the kernel is strided by two pixels in each step. This means again that the output size is smaller than the input size.



```
model.add(Conv2D(10, kernel_size=3, activation='relu',  
                input_shape=(img_rows, img_cols, 1)),  
          strides=1)
```

```
model.add(Conv2D(10, kernel_size=3, activation='relu',  
                input_shape=(img_rows, img_cols, 1)),  
          strides=2)
```

Calculating the size of the output

$$O = ((I - K + 2P)/S) + 1$$

where

- I = size of the input
- K = size of the kernel
- P = size of the zero padding
- S = strides

For example, if the input image is five by five and there is both zero padding and strides set to 2, the output will have a size of three by three.

Generally, we can calculate the size of the output using a simple formula: $(I - K + 2P) / (S + 1)$ Where I is the size of the input, K is the size of the kernel, P is the size of the zero padding, and S is the stride.

$$28 = ((28 - 3 + 2)/1) + 1$$

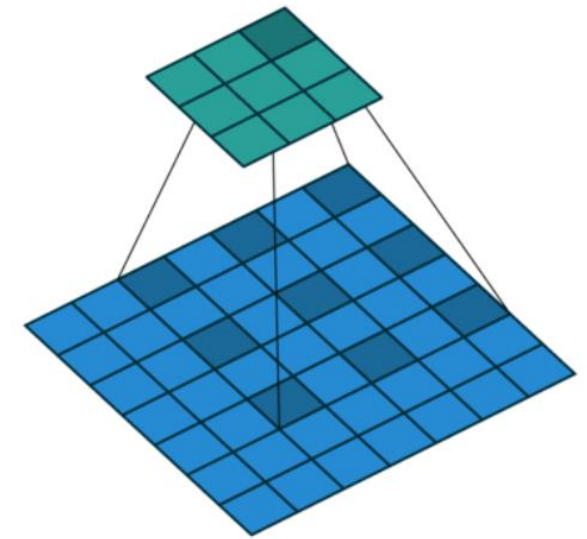
$$10 = ((28 - 3 + 2)/3) + 1$$

For example, if the input is 28 pixels, the kernel is 3 by 3, the padding is of 1 and the stride is 1, this number will be: $(28 - 3 + 2) / 1 + 1 = 28$. If instead the stride is 3 the output size would be 10 by 10.

You can also tweak the spacing between the pixels affected by the kernel. This is called a dilated convolution. In this case, the convolution kernel has only 9 parameters, but it has the same field of view as a kernel that would have the size 5 by 5. This is useful in cases where you need to aggregate information across multiple scales.

This too is controlled through a keyword argument, "dilation_rate", that sets the distance between subsequent pixels.

```
model.add(Conv2D(10, kernel_size=3, activation='relu',  
                input_shape=(img_rows, img_cols, 1)),  
          dilation_rate=2)
```



Add padding to a CNN

Padding allows a convolutional layer to retain the resolution of the input into this layer. This is done by adding zeros around the edges of the input image, so that the convolution kernel can overlap with the pixels on the edge of the image.

```
# Initialize the model  
model = Sequential()  
  
# Add the convolutional layer  
model.add(Conv2D(10, kernel_size=3, activation='relu',  
                input_shape=(img_rows, img_cols, 1),  
                padding='same'))  
  
# Feed into output layer  
model.add(Flatten())
```

```
model.add(Dense(3, activation='softmax'))
```

With padding set to 'same', the output layer will have the same size as the input layer!

Add strides to a convolutional network

The size of the strides of the convolution kernel determines whether the kernel will skip over some of the pixels as it slides along the image. This affects the size of the output because when strides are larger than one, the kernel will be centered on only some of the pixels.

```
# Initialize the model
model = Sequential()

# Add a convolutional layer
model.add(Conv2D(10, kernel_size=3, activation='relu',
                 input_shape=(img_rows, img_cols, 1),
                 strides=2))

# Feed into output layer
model.add(Flatten())
model.add(Dense(3, activation='softmax'))
```

With strides set to 2, the network skips every other pixel.

Calculate the size of convolutional layer output

Zero padding and strides affect the size of the output of a convolution.

What is the size of the output for an input of size 256 by 256, with a kernel of size 4 by 4, padding of 1 and strides of 2?

Answer: 128

☐ 127

☐ 255

☒ 128

☐ 256

Chapter 3. Going Deeper

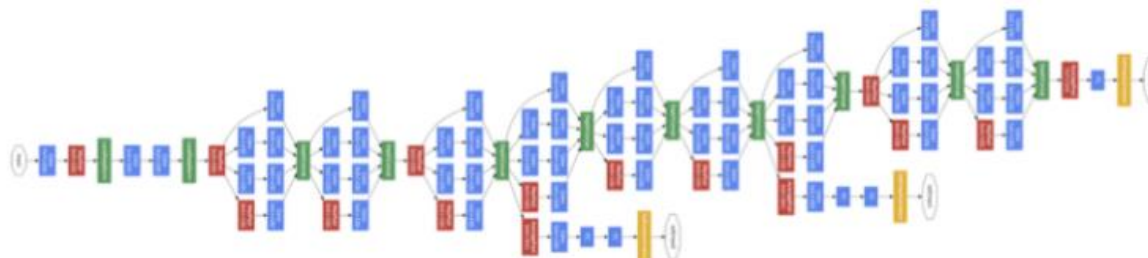
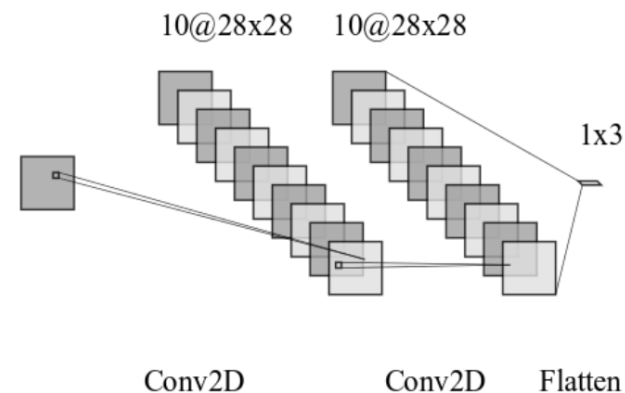
Convolutional neural networks gain a lot of power when they are constructed with multiple layers (deep networks). In this chapter, you will learn how to stack multiple convolutional layers into a deep network. You will also learn how to keep track of the number of parameters, as the network grows, and how to control this number.

Going deeper

One of the major strengths of convolutional neural networks comes from building networks with multiple layers of convolutional filters.

```
model = Sequential()
model.add(Conv2D(10, kernel_size=2, activation='relu',
                 input_shape=(img_rows, img_cols, 1),
                 padding='equal'))

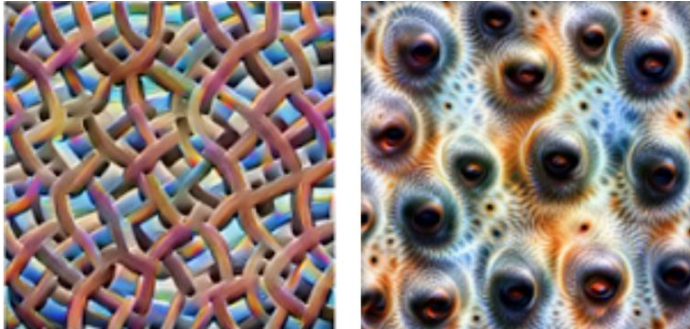
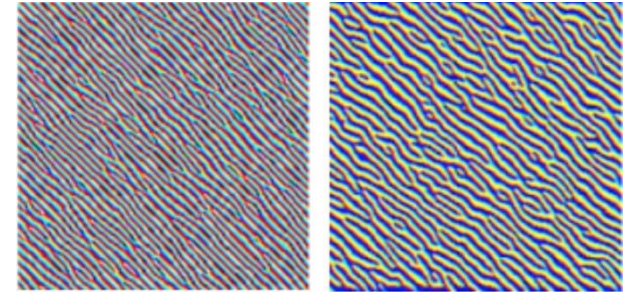
# Second convolutional layer
model.add(Conv2D(10, kernel_size=2, activation='relu'))
model.add(Flatten())
model.add(Dense(3, activation='softmax'))
```



This is the architecture of a network developed by Google researchers in 2014. It has 22 layers of convolutions, and some other kinds of layers, like pooling layers. One way to understand why we would want a network this deep is by looking at the kinds of things that the kernels and feature maps in the different layers tend to respond to.

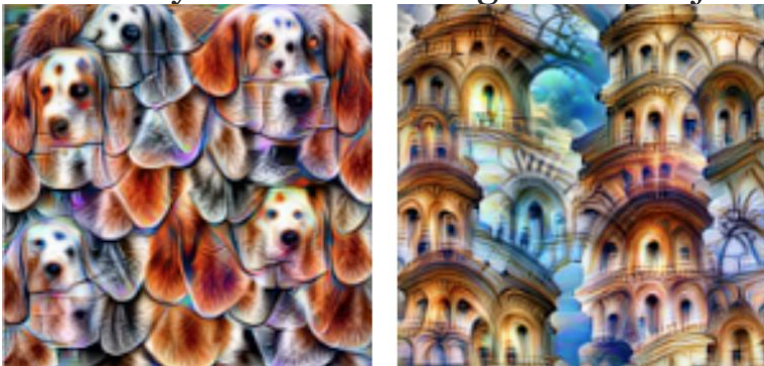
These are the kinds of things that layers in the early part of the network tend to respond to. Oriented lines, or simple textures.

Intermediate layers of the network tend to respond to more complex features, that include simple objects, such as eyes.



By the time the information travels up to higher layers of the network, the feature maps tend to extract specific types of objects. This allows the fully connected layers at the top of the network to extract useful information for object classification based on the responses of these layers.

In other words, having multiple layers of convolutions in the network allows the network to gradually build up representations of objects in the images from simple features to more complex features and up to sensitivity to distinct categories of objects.



How deep should your network be? Consider the following: despite the advantages mentioned above, depth does come at a computational cost. In addition, to train a very deep network you might need a larger amount of training data.

Creating a deep learning network

A deep convolutional neural network is a network that has more than one layer. Each layer in a deep network receives its input from the preceding layer, with the very first layer receiving its input from the images used as training or test data.

Here, you will create a network that has two convolutional layers.

```
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten

model = Sequential()

# Add a convolutional layer (15 units)
model.add(Conv2D(15, kernel_size=2, activation='relu',
                 input_shape=(img_rows, img_cols, 1)))

# Add another convolutional layer (5 units)
model.add(Conv2D(5, kernel_size=2, activation='relu'))

# Flatten and feed to output layer
model.add(Flatten())
model.add(Dense(3, activation='softmax'))
```

You built a network with multiple convolution layers.

Train a deep CNN to classify clothing images

Training a deep learning model is very similar to training a single layer network. Once the model is constructed (as you have done in the previous exercise), the model needs to be compiled with the right set of parameters. Then, the model is fit by providing it with training data, as well as training labels. After training is done, the model can be evaluated on test data.

The `model` you built in the previous exercise is available in your workspace.

```
# Compile model
model.compile(optimizer='adam',
```



```

        loss='categorical_crossentropy',
        metrics=['accuracy'])

# Fit the model to training data
model.fit(train_data, train_labels,
          validation_split=0.2,
          epochs=3, batch_size=10)

# Evaluate the model on test data
model.evaluate(test_data, test_labels, batch_size=10)

```

<script.py> output:

Train on 40 samples, validate on 10 samples

Epoch 1/3

10/40 [=====>.....] - ETA: 2s - loss: 1.0957 - acc: 0.4000

40/40 [=====] - 1s 22ms/step - loss: 1.0861 - acc: 0.5750 - val_loss: 1.0630 - val_acc: 0.7000

Epoch 2/3

10/40 [=====>.....] - ETA: 0s - loss: 1.0665 - acc: 0.7000

40/40 [=====] - 0s 913us/step - loss: 1.0379 - acc: 0.8750 - val_loss: 0.9906 - val_acc: 1.0000

Epoch 3/3

10/40 [=====>.....] - ETA: 0s - loss: 1.0027 - acc: 1.0000

40/40 [=====] - 0s 955us/step - loss: 0.9559 - acc: 0.9750 - val_loss: 0.8835 - val_acc: 1.0000

10/10 [=====] - 0s 467us/step

Accuracy calculated on the test data is not subject to overfitting.

What is special about a deep network?

Networks with more convolution layers are called "deep" networks, and they may have more power to fit complex data, because of their ability to create hierarchical representations of the data that they fit.

What is a major difference between a deep CNN and a CNN with only one convolutional layer?

- ☐ A deep network is inspired by the human visual system.
- ☒ A deep network requires more data and more computation to fit.
- ☐ A deep network has more dense layers.
- ☐ A deep network has larger convolutions.

That's right!

How many parameters?

```
model = Sequential()
```

```
model.add(Dense(
    10, activation='relu',
    input_shape=(784,)))
```

$$\begin{aligned} \text{parameters} &= 784 * 10 + 10 \\ &= 7850 \end{aligned}$$

```
model.add(Dense(
    10, activation='relu'))
```

$$\begin{aligned} \text{parameters} &= 10 * 10 + 10 \\ &= 110 \end{aligned}$$

```
model.add(Dense(
    3, activation='softmax'))
```

$$\begin{aligned} \text{parameters} &= 10 * 3 + 3 \\ &= 33 \end{aligned}$$

$$7850 + 110 + 33 = 7993$$

```
# Call the summary method
model.summary()
```

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 10)	7850
dense_2 (Dense)	(None, 10)	110
dense_3 (Dense)	(None, 3)	33
Total params: 7,993		
Trainable params: 7,993		
Non-trainable params: 0		

```

model = Sequential()

model.add(
    Conv2D(10, kernel_size=3,
           activation='relu',
           input_shape=(28, 28, 1),
           padding='same'))

model.add(
    Conv2D(10, kernel_size=3,
           activation='relu',
           padding='same'))

model.add(Flatten())

```

```

model.add(Dense(
    3, activation='softmax'))

```

$$parameters = 9 * 10 + 10$$

$$= 100$$

$$parameters = 10 * 9 * 10 + 10$$

$$= 910$$

$$parameters = 0$$

$$parameters = 7840 * 3 + 3$$

$$= 23523$$

$$100 + 910 + 0 + 23523 = 24533$$

```

model = Sequential()

model.add(Dense(5, activation='relu',
                input_shape=(784,), padding='same'))

model.add(Dense(15, activation='relu', padding='same'))

model.add(Dense(3, activation='softmax'))

```

```
model.summary()
```

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 28, 28, 10)	100
conv2d_2 (Conv2D)	(None, 28, 28, 10)	910
flatten_3 (Flatten)	(None, 7840)	0
dense_4 (Dense)	(None, 3)	23523
Total params: 24,533		
Trainable params: 24,533		
Non-trainable params: 0		

```
model.summary()
```

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 5)	3925
dense_2 (Dense)	(None, 15)	90
dense_3 (Dense)	(None, 3)	48
Total params: 4,063		
Trainable params: 4,063		
Non-trainable params: 0		

```

model = Sequential()

model.add(Conv2D(5, kernel_size=3, activation='relu',
                input_shape=(28, 28, 1),
                padding="same"))

model.add(Conv2D(15, kernel_size=3, activation='relu',
                padding="same"))

model.add(Flatten())

model.add(Dense(3, activation='softmax'))

```

```
model.summary()
```

Layer (type)	Output Shape	Param #
conv2d_12 (Conv2D)	(None, 28, 28, 5)	50
conv2d_13 (Conv2D)	(None, 28, 28, 15)	690
flatten_6 (Flatten)	(None, 11760)	0
dense_9 (Dense)	(None, 3)	35283
Total params: 36,023		
Trainable params: 36,023		
Non-trainable params: 0		

This is the convolutional network with the equivalent number of units in each layer. In contrast to the densely connected neural network, when we run the summary method on this network, we see that the first few layers use very few parameters, while the last layer uses a lot of parameters. One way to think about this is that the convolutions have more expressive power, so they require less parameters, but reading out these more expressive representations then requires many more parameters on the output side.

How many parameters in a CNN?

We need to know how many parameters a CNN has, so we can adjust the model architecture, to reduce this number or shift parameters from one part of the network to another. How many parameters would a network have if its inputs are images with 28-by-28 pixels, there is one convolutional layer with 10 units kernels of 3-by-3 pixels, using zero padding (input has the same size as the output), and one densely connected layer with 2 units?

Possible Answers

- ☐ 100
- ☐ 1668
- ☒ 15,782
- ☐ 15,682

How many parameters in a deep CNN?

In this exercise, you will use Keras to calculate the total number of parameters along with the number of parameters in each layer of the network.

We have already provided code that builds a deep CNN for you.

```
# CNN model
model = Sequential()
model.add(Conv2D(10, kernel_size=2, activation='relu',
                 input_shape=(28, 28, 1)))
model.add(Conv2D(10, kernel_size=2, activation='relu'))
model.add(Flatten())
model.add(Dense(3, activation='softmax'))

# Summarize the model
model.summary()
```

<script.py> output:

Model: "sequential_1"

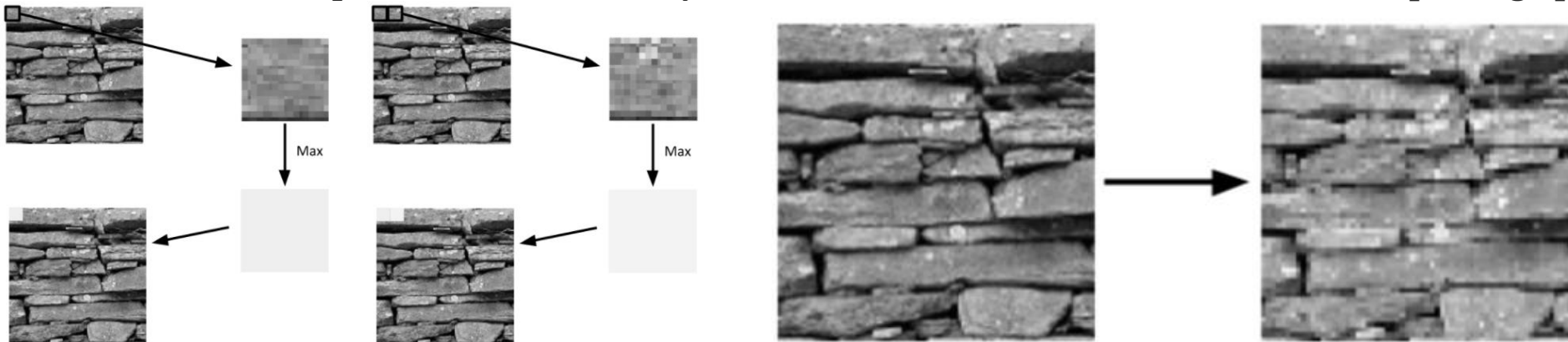
Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 27, 27, 10)	50
conv2d_2 (Conv2D)	(None, 26, 26, 10)	410
flatten_1 (Flatten)	(None, 6760)	0
dense_1 (Dense)	(None, 3)	20283

=====
Total params: 20,743
Trainable params: 20,743
Non-trainable params: 0
=====

This model has 20,743 parameters!

Pooling operations

One of the challenges in fitting neural networks is the large number of parameters. One way to mitigate this is to summarize the output of convolutional layers in concise manner. To do this, we can use pooling operations.



We might summarize a group of pixels based on its maximal value. This is called "max pooling". We might start with the top left corner, extracting the pixels in this part of the image and, calculating the maximal value of the pixels there. In the output, we replace these pixels with one large pixel that stores this maximal value. If we repeat this operation in multiple windows of size 2 by 2, we end up with an image that has a quarter of the number of the original pixels, and retains only the brightest feature in each part of the image.

We can integrate max pooling operations into a Keras convolutional neural network, using the MaxPool2D object.

```
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten, MaxPool2D
model = Sequential()
model.add(Conv2D(5, kernel_size=3, activation='relu',
                 input_shape=(img_rows, img_cols, 1)))
model.add(MaxPool2D(2))
model.add(Conv2D(15, kernel_size=3, activation='relu',
                 input_shape=(img_rows, img_cols, 1)))
model.add(MaxPool2D(2))
model.add(Flatten())
model.add(Dense(3, activation='softmax'))
```

```
model.summary()
```

Layer (type)	Output Shape	Param #
=====		
conv2d_1 (Conv2D)	(None, 26, 26, 5)	50

max_pooling2d_1 (MaxPooling2D)	(None, 13, 13, 5)	0

conv2d_2 (Conv2D)	(None, 11, 11, 15)	690

max_pooling2d_2 (MaxPooling2D)	(None, 5, 5, 15)	0

flatten_1 (Flatten)	(None, 375)	0

dense_1 (Dense)	(None, 3)	1128
=====		

Running the summary method for this model, we can see that using the pooling operation dramatically reduces the number of parameters in the model. Instead of more than 30,000 parameters that this model had with no pooling operations, we now have less than 2,000 parameters.

There are of course trade-offs to reducing the number of parameters. Let's see how this plays out with some examples.

Write your own pooling operation

As we have seen before, CNNs can have a lot of parameters. Pooling layers are often added between the convolutional layers of a neural network to summarize their outputs in a condensed manner, and reduce the number of parameters in the next layer in the network. This can help us if we want to train the network more rapidly, or if we don't have enough data to learn a very large number of parameters.

A pooling layer can be described as a particular kind of convolution. For every window in the input it finds the maximal pixel value and passes only this pixel through. In this exercise, you will write your own max pooling operation, based on the code that you previously used to write a two-dimensional convolution operation.

```
# Result placeholder
result = np.zeros((im.shape[0]//2, im.shape[1]//2))

# Pooling operation
for ii in range(result.shape[0]):
    for jj in range(result.shape[1]):
        result[ii, jj] = np.max(im[ii*2:ii*2+2, jj*2:jj*2+2])
```

The resulting image is smaller, but retains the salient features in every location

Keras pooling layers

Keras implements a pooling operation as a layer that can be added to CNNs between other layers. In this exercise, you will construct a convolutional neural network similar to the one you have constructed before:

Convolution => Convolution => Flatten => Dense

However, you will also add a pooling layer. The architecture will add a single max-pooling layer between the convolutional layer and the dense layer with a pooling of 2x2:

Convolution => Max pooling => Convolution => Flatten => Dense

A Sequential `model` along with `Dense`, `Conv2D`, `Flatten`, and `MaxPool2D` objects are available in your workspace.

```
# Add a convolutional layer
model.add(Conv2D(15, kernel_size=2, activation='relu',
                 input_shape=(img_rows, img_cols, 1)))

# Add a pooling operation
model.add(MaxPool2D(2))

# Add another convolutional layer
model.add(Conv2D(5, kernel_size=2, activation='relu'))

# Flatten and feed to output layer
model.add(Flatten())
model.add(Dense(3, activation='softmax'))
model.summary()
```

```
<script.py> output:
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 27, 27, 15)	75
max_pooling2d_1 (MaxPooling2D)	(None, 13, 13, 15)	0
conv2d_2 (Conv2D)	(None, 12, 12, 5)	305
flatten_1 (Flatten)	(None, 720)	0
dense_1 (Dense)	(None, 3)	2163

=====
Total params: 2,543
Trainable params: 2,543
Non-trainable params: 0
=====

This model is even deeper, but has fewer parameters.

Train a deep CNN with pooling to classify images

Training a CNN with pooling layers is very similar to training of the deep networks that you have seen before. Once the network is constructed (as you did in the previous exercise), the model needs to be appropriately compiled, and then training data needs to be provided, together with the other arguments that control the fitting procedure. The following `model` from the previous exercise is available in your workspace:

Convolution => Max pooling => Convolution => Flatten => Dense

```
# Compile model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Fit the model to training data
model.fit(train_data, train_labels,
          validation_split=0.2,
          epochs=3, batch_size=10)

# Evaluate the model on test data
model.evaluate(test_data, test_labels, batch_size=10)
```

```
<script.py> output:
  Train on 40 samples, validate on 10 samples
  Epoch 1/3

  10/40 [=====>.....] - ETA: 0s - loss: 1.0914 - acc: 0.7000
  40/40 [=====] - 0s 6ms/step - loss: 1.0744 - acc: 0.7000 - val_loss: 1.0685 - val_acc: 0.8000
  Epoch 2/3

  10/40 [=====>.....] - ETA: 0s - loss: 1.0674 - acc: 0.7000
  40/40 [=====] - 0s 470us/step - loss: 1.0524 - acc: 0.8250 - val_loss: 1.0376 - val_acc: 0.9000
  Epoch 3/3

  10/40 [=====>.....] - ETA: 0s - loss: 1.0324 - acc: 1.0000
  40/40 [=====] - 0s 451us/step - loss: 1.0184 - acc: 0.9250 - val_loss: 0.9932 - val_acc: 0.9000

  10/10 [=====] - 0s 171us/step
```

This model is also very accurate!

Chapter 4. Understanding and Improving Deep Convolutional Networks

There are many ways to improve training by neural networks. In this chapter, we will focus on our ability to track how well a network is doing, and explore approaches towards improving convolutional neural networks.

Course completed!

Recap topics covered:

- Basics

Next Steps

- Basics

Happy learning!