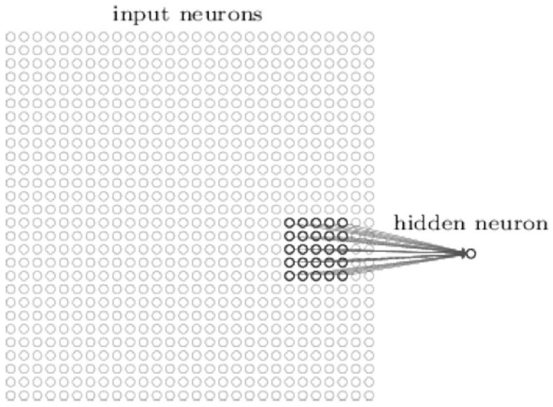
Convolutional Neural Networks (Part I)

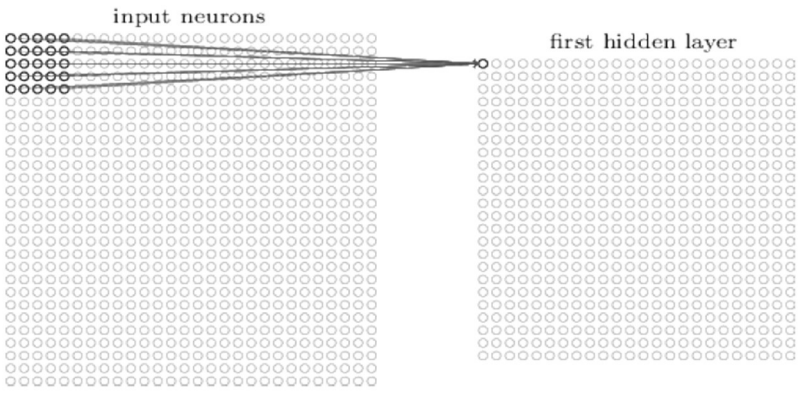
So far, we have seen neural networks made of dense layers with possible residual network connections between these layers. In this tutorial, we'll introduce a new type of neural network that is mainly use in imagery called Convolutional Neural Network (ConvNet, in short). ConvNet are neural networks that contain two type of layers: convolutional layers and pooling layers described in the two following sections.

1. Convolutional layers

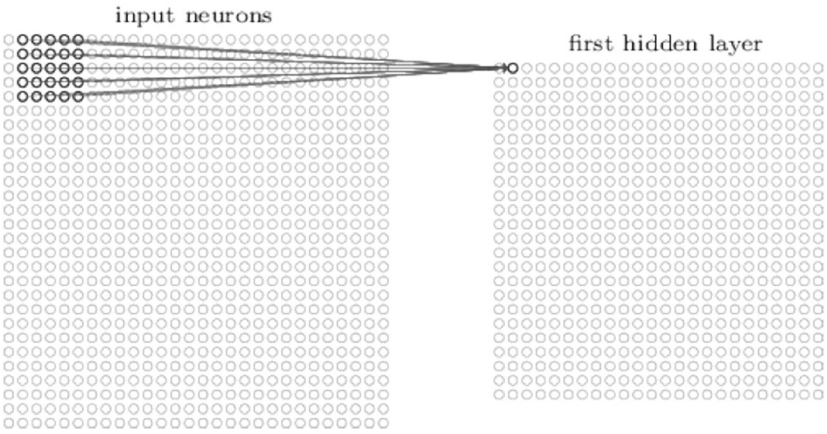
In a convolutional net, it'll help to think of the inputs as a 28x28 square of neurons (instead of a vector of size 784), whose values correspond to the 28x28 pixel intensities. As usual, we'll connect the input pixels to a layer of hidden neurons. But we won't connect every input pixel to every hidden neuron. Instead, we only make connections in small, localized regions of the input image. To be more precise, each neuron in the first hidden layer will be connected to a small region of the input neurons, say, for example, a 5x5 region, corresponding to 25 input pixels. So, for a particular hidden neuron, we might have connections that look like this:



That region in the input image is called the local receptive field for the hidden neuron. It's a little window on the input pixels. Each connection learns a weight. And the hidden neuron learns an overall bias as well. You can think of that particular hidden neuron as learning to analyze its particular local receptive field. We then slide the local receptive field across the entire input image. For each local receptive field, there is a different hidden neuron in the first hidden layer. To illustrate this concretely, let's start with a local receptive field in the top-left corner:



Then we slide the local receptive field over by one pixel to the right (i.e., by one neuron), to connect to a second hidden neuron:



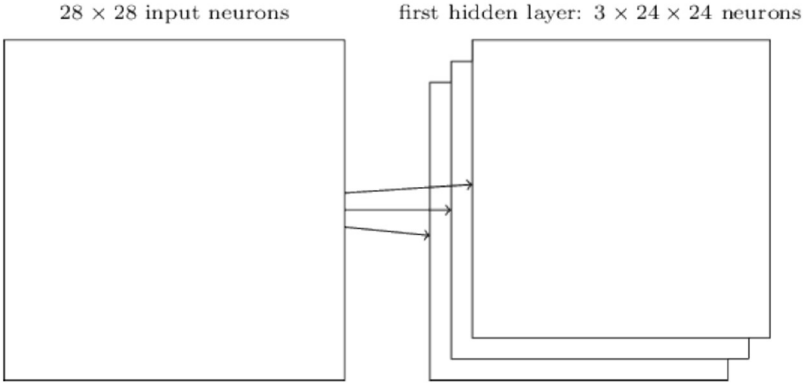
And so on, building up the first hidden layer. Note that if we have a 28x28 input image, and 5x5 local receptive fields, then there will be 24x24 neurons in the hidden layer. This is because we can only move the local receptive field 23 neurons across (or 23 neurons down), before colliding with the right-hand side (or bottom) of the input image.

Here, we have shown the local receptive field being moved by one pixel at a time. In fact, sometimes a different stride length is used. For instance, we might move the local receptive field 2 pixels to the right (or down), in which case we'd say a stride length of 2 is used. We said that each hidden neuron has a bias and 5x5 weights connected to its local receptive field. What we did not yet mention is that we're going to use **the same weights and bias** for each of the 24x24 hidden neurons. In other words, for the (*j*, *k)* hidden neuron, the output is:

$$\sigma\left(b + \sum_{l=0}^{4}\sum_{m=0}^{4} w_{lm} a_{j+l,k+m}\right)$$

where σ is the activation function (for example the sigmoid function) *b* is the bias and w_{lm} are the weights. And, finally, we use a_{ij} to denote the input (i,j) of the layer. This means that all the neurons in the first hidden layer detect exactly the same feature, just at different locations in the input image. We sometimes call the map from the input layer to the hidden layer a *feature map*. We call the weights defining the feature map the *shared weights*. And we call the bias defining the feature map in this way the *shared bias*. The shared weights and bias are often said to define a *kernel* or a ***filter***.

The network structure described so far can detect just a single kind of localized feature. To do image recognition we'll need more than one feature map. And so, a complete convolutional layer consists of several different feature maps:



In the example shown, there are 3 feature maps. Each feature map is defined by a set of 5x5 shared weights, and a single shared bias. The result is that the network can detect 3 different kinds of features, with each feature being detectable across the entire image.
We've shown just 3 feature maps, to keep the diagram above simple. However, in practice convolutional networks may use more (a few dozens) feature maps. The layer described above can be coded with Keras as follows:

```
model.add(layers.Conv2D(3, (5, 5), activation='relu'))
```

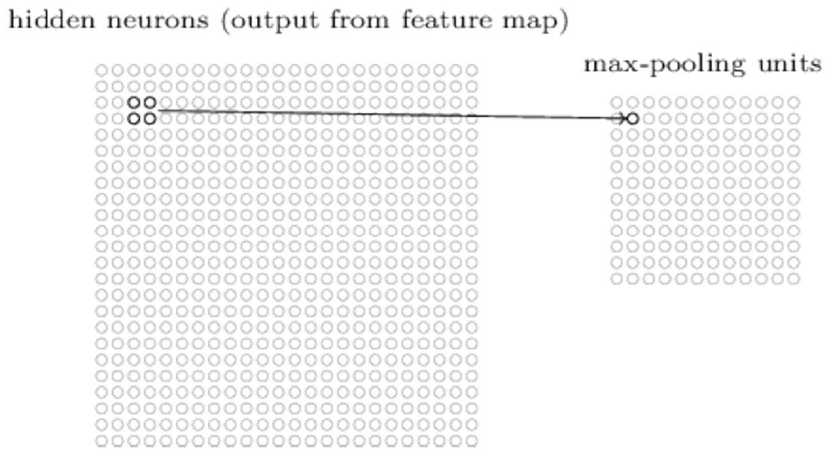
and the same layer coded using Keras functional API would be:

```
y=layers.Conv2D(3, (5, 5), activation='relu')(x)
```

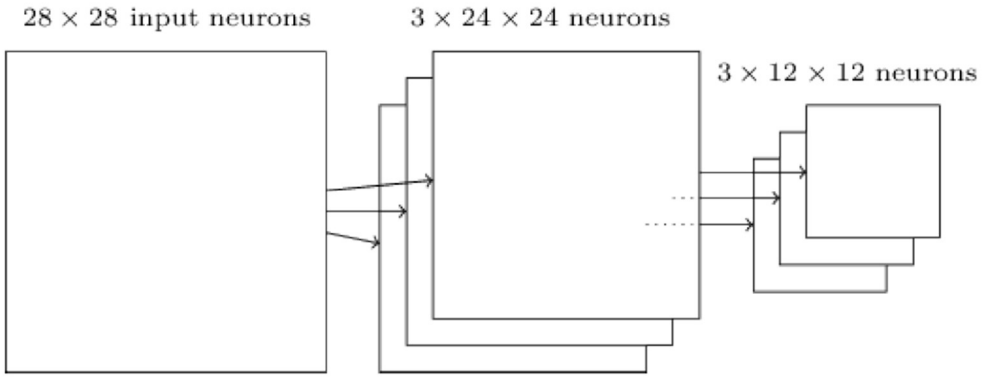
A big advantage of sharing weights and biases is that it greatly reduces the number of parameters involved in a convolutional network. For each feature map we need 25 = 5x5 shared weights, plus a single shared bias. So, each feature map requires 26 parameters. If we have 20 feature maps that's a total of 20x26 = 520 parameters defining the convolutional layer. By comparison, suppose we had a fully connected first layer, with 784 = 28x28 input neurons, and a relatively modest 30 hidden neurons, that's a total of 784 x30 weights, plus an extra 30 biases, for a total of 23,550 parameters. In other words, the fully-connected layer would have more than 40 times as many parameters as the convolutional layer.
Convolutional layers are always used in conjunction with pooling layers that are described in the next section.

2. Pooling layers

In addition to the convolutional layers just described, convolutional neural networks also contain pooling layers. Pooling layers are usually used immediately after convolutional layers. What the pooling layers do is simplify the information in the output from the convolutional layer. In detail, a pooling layer takes each feature map output from the convolutional layer and prepares a condensed feature map. For instance, each unit in the pooling layer may summarize a region of (say) 2x2 neurons in the previous layer. As a concrete example, one common procedure for pooling is known as max-pooling. In max-pooling, a pooling unit simply outputs the maximum activation in the 2x2 input region, as illustrated in the following diagram:



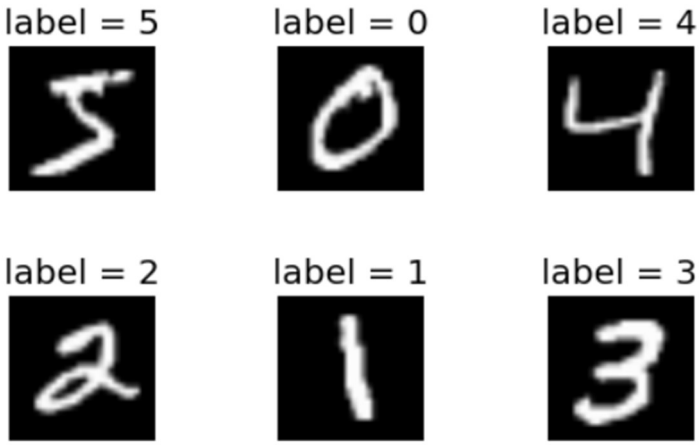
Note that since we have 24x24 neurons output from the convolutional layer, after pooling we have 12x12 neurons. As mentioned above, the convolutional layer usually involves more than a single feature map. We apply max-pooling to each feature map separately. So if there were three feature maps, the combined convolutional and max-pooling layers would look like:



A big benefit is that there are many fewer pooled features, and so this helps reduce the number of parameters needed in later layers Max-pooling isn't the only technique used for pooling. Another common approach is known as *L2 pooling*. Here, instead of taking the maximum activation of a 2x2 region of neurons, we take the square root of the sum of the squares of the activations in the 2x2 region. While the details are different, the intuition is similar to max-pooling: L2 pooling is a way of condensing information from the convolutional layer. In practice, both techniques have been widely used. And sometimes people use other types of pooling operation (average pooling, for example). If you're really trying to optimize performance, you may use validation data to compare several different approaches to pooling, and choose the approach which works best.

3. Building a ConvNet with Keras

In this section, we will build a ConvNet neural network to solve a classification problem with the MNIST dataset. The MNIST dataset is a collection of images with 256 levels of grey (going from 0 to 255) and representing a number going from 0 to 9. Here are some examples of images from the MNIST dataset together with their label:



The output of the neural network is a probability vector of size 10 such that its i^{th} index gives the probability that that the input image represents the figure "i". The dataset MNIST is so famous that it is bundled into Keras library. Loading it into vectors is therefore easy:

```
from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

Let's see the shapes of the different tensors:

```
print(train_images.shape)
print(train_labels.shape)

print(test_images.shape)
print(test_labels.shape)
```

```
(60000, 28, 28)
(60000,)
(10000, 28, 28)
(10000,)
```

We can see that the training set contains 60000 images and the validation set contains 10000 images. We also see that each image has a resolution 28x28 pixels. Lets have a look and see how the first image is encoded:

```
print(train_images[0])
```

```
[[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   3  18  18  18 126 136
  175  26 166 255 247 127   0   0   0   0]
 [  0   0   0   0   0   0   0   0  30  36  94 154 170 253 253 253 253 253
  225 172 253 242 195  64   0   0   0   0]
 [  0   0   0   0   0   0   0  49 238 253 253 253 253 253 253 253 253 251
   93  82  82  56  39   0   0   0   0   0]
 [  0   0   0   0   0   0   0  18 219 253 253 253 253 253 198 182 247 241
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  80 156 107 253 253 205  11   0  43 154
```

It contains many zeros, which is normal since zeros corresponds to the black colour. We also see that the entries of this tensor have a large magnitude. And remember ConvNet (just like dense neural networks) do not work if the input data have not been scaled. This is what we should start to do. Prior to that, convolutional layers in Keras expect an input of the form *(Number of samples, Image height, Image width, Number of channels)*. In our case, the number of channels is one since the images are in grey levels. If we had colour images, there would be three channels for RGB (Red, Green, Blue). This is why we also start by reshaping the input tensor:

```
# Reshaping
X_train = train_images.reshape((60000, 28, 28, 1))
X_test = test_images.reshape((10000, 28, 28, 1))
# Scaling
X_train = X_train/255
X_test = X_test/255
```

The label vector is not encoded in the right way. The label contains a number between 0 and 9 that corresponds to the number represented by the image. But what we want at the output is a probability vector of size 10. This vector will contain zeros everywhere except at the position whose number is represented by the image where it will be one. Keras can do that for us by using the function *to_categorical()*. Such encoding is called one hot encoding and is widely used for classification problems.

```
from keras.utils import to_categorical
y_train = to_categorical(train_labels)
y_test = to_categorical(test_labels)
```

Now is time to build the ConvNet: here we have two convolutional layers, each one followed by a pooling layer.

```
model = models.Sequential()


model.add(layers.Conv2D(32, (3, 3), strides=(1,1),activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), strides=(1,1), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))

model.summary()
```

The first convolution layer has 32 features map and the second one has 64 features map. Both layers have a filter of same size 3x3 and a stride equal to one in each direction. The MaxPooling layers summarize a region 2x2. The next layers are the usual dense layers that you are familiar with. A dense layer expects a vector as input whereas a convolutional or a maxpooling layer returns a matrix. Therefore, it is necessary to transform the matrix into vector and this is what the *Flatten()* layer does. The last layer must contain 10 neurons to ensure an output vector of size 10 and a softmax activation function so that this vector is a probability vector. The same network could also be coded with Keras functional API as follows:

```
from keras import Input
from keras.models import Model

x0 = Input(shape=(28,28,1))

x1 = layers.Conv2D(32, (3, 3), strides=(1,1),activation='relu', input_shape=(28, 28, 1))(x0)
x2 = layers.MaxPooling2D((2, 2))(x1)

x3 = layers.Conv2D(64, (3, 3), strides=(1,1), activation='relu')(x2)
x4 = layers.MaxPooling2D((2, 2))(x3)

x5 = layers.Flatten()(x4)
x6 = layers.Dense(64, activation='relu')(x5)
x7 = layers.Dense(10, activation='softmax')(x6)

model =  Model(inputs=x0, outputs=x7)
```

Now it’s time to compile and fit our model and see how it performs:

```
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(X_train, y_train, epochs=4, batch_size=64, validation_data=(X_test,y_test))
```

```
Epoch 1/4
938/938 [==============================] - 14s 15ms/step - loss: 0.1624 - accuracy: 0.9507 - val_loss: 0.0548 - val_accuracy:
0.9816
Epoch 2/4
938/938 [==============================] - 14s 14ms/step - loss: 0.0491 - accuracy: 0.9851 - val_loss: 0.0369 - val_accuracy:
0.9871
Epoch 3/4
938/938 [==============================] - 14s 15ms/step - loss: 0.0349 - accuracy: 0.9890 - val_loss: 0.0302 - val_accuracy:
0.9904
Epoch 4/4
938/938 [==============================] - 15s 16ms/step - loss: 0.0260 - accuracy: 0.9923 - val_loss: 0.0264 - val_accuracy:
0.9909
```

First thing you will notice is that it takes time to train. And we are using low resolution grey levels images. Imagine how it would be with a colour image of resolution 128x128…In practice, for more realistic problems, one has to do computation on GPU card. Typically, computations that would take hours on a CPU would only take a few seconds or a minute on a GPU card.

Next thing you should notice, is that we get around 99% of correct classification on the validation: this is an excellent result, obtained after only four iterations. In order to reduce the computation time a possibility is to use pretrained ConvNet: that will be the topic of the next tutorial.

4. Your work

The whole code is below in one piece:

```
from keras import Input
from keras.models import Model
from keras import layers
from keras.datasets import mnist
from keras.utils import to_categorical

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
# Reshaping
X_train = train_images.reshape((60000, 28, 28, 1))
X_test = test_images.reshape((10000, 28, 28, 1))

# Scaling
X_train = X_train/255
X_test = X_test/255

# One hot encoding
y_train = to_categorical(train_labels)
y_test = to_categorical(test_labels)


x0 = Input(shape=(28,28,1))

x1 = layers.Conv2D(32, (3, 3), strides=(1,1),activation='relu', input_shape=(28, 28, 1))(x0)
x2 = layers.MaxPooling2D((2, 2))(x1)

x3 = layers.Conv2D(64, (3, 3), strides=(1,1), activation='relu')(x2)
x4 = layers.MaxPooling2D((2, 2))(x3)

x5 = layers.Flatten()(x4)
x6 = layers.Dense(64, activation='relu')(x5)
x7 = layers.Dense(10, activation='softmax')(x6)

model =  Model(inputs=x0, outputs=x7)

model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(X_train, y_train, epochs=4, batch_size=64, validation_data=(X_test,y_test))
```

In this tutorial, you must understand how a ConvNet is built and you must test this code. You must use *model.summary()* to see the dimensions of the layers. You must check out if the dimensions are as expected. Then you should experiment with different parameters. You should be able to go beyond 99% of correct classification for this problem.

Exercice:

Build a neural network that has the structure given in *model.png* and that uses convolutional and maxpooling layers. Then, make a search on the internet about the Dropout layer and how it can be used to regularize a neural network. Add Dropout layers to your neural networks and see if it helps going beyond 99% of accuracy.