Convolutional and Recurrent Neural Networks

# Note for Jojo, Aditee and Matt:

Hallo. Please note that the content of this chapter has been changed during discussion with Suresh. At the beginning the idea was to have a chapter on both networks with detailed tensorflow implementation. I realized during the writing process that this was not possible, due to the sheer amount of material needed. With Suresh John Celestin we decided to put this material in a second book that is in the process of approval. That means that this chapter will contain only a very basic explanation of CNN and RNN, and a very basic tensorflow implementation that I hope will give the reader a basic understanding of the building blocks of both kind of networks. So please keep this in mind while reviewing this chapter. Thanks for all your feedback and help! Best, Umberto

In the past chapters we have looked at fully connected networks and all the problems you encounter while training them. The network architecture we have used, one where each neuron in a layer is connected to all neurons in the previous and next layer, is not really good at many fundamental tasks like image recognition, speech recognition, time series prediction and many more. Convolutional neural networks (CNN) and Recurrent neural networks (RNN) are the most used advanced architectures used today. In this chapter we will look at convolution and pooling, the basic building blocks of CNNs. Then we will check how RNNs work on a high level and we will look at a selected number of examples of applications. I will also discuss a complete, although basic, implementation of CNNs and RNNs in tensorflow.

# Kernels and filters

One of the main components of CNNs are filters, square matrices that have dimensions , where usally is a small number, like or . Sometimes filters are also called kernels. Let's define here four different filters and let's check later in the chapter their effect when used in convolution operations. For those examples, we will work with filters. For the moment just take the following definitions as a reference and we will see how to use them later in the chapter.

* The following kernel will allow the detection of horizontal edges
* The following kernel will allow the detection of vertical edges
* The following kernel will allow the detection of edges when luminosity changes drastically
* The following kernel will blur edges in an image

In the next sections we will apply convolution to a test image with the filters and you will see what their effect is.

# Convolution

The first step to understand CNNs is to understand convolution. The easiest way is to see it in action in a few simple cases. First, in the context of neural networks, convolution is done between tensors. The operation gets two tensors as input and produce a tensor as output. The operation is usually indicated with the operator . Let's see how it works. Let's get two tensors, both with dimensions . The convolution operation is done applying the following formula

in this case the result is simply the sum of each element multiplied by the respective element . In a more typical matrix formalism this formula could be written with a double sum as

But the first version as the advantage of making the fundamental idea very clear: each element from one tensor is multiplied by the correspondent element (the element in the same position) of the second tensor and then all the values are summed to get the result.

In the previous section we have talked about kernels, and the reason is that convolution is usually done between a tensor, that we may indicate here with , and a kernel. Typically, kernels are small, or , while the input tensors are normally bigger. In image recognition for example, the input tensors are the images that may have dimensions as high as , where is the resolution and the last dimension () is the number of the color channels, the RGB values. In advanced applications the images may even have higher resolution. How to apply convolution when we have matrices with different dimensions? To understand it, let's consider a matrix that is

And let's see how to do convolution with a Kernel that we will take for this example to be

The idea is to start on the top left corner of the matrix and select a region. In the example that would be

or the elements marked in bold below

Then we perform the convolution, as explained at the beginning between this smaller matrix and getting (we will indicate the result with

Then we need to shift the selected region in matrix of one column to the right and select the elements marked in bold below

that will give us the second sub-matrix

and we perform again the convolution between this smaller matrix and

now we cannot shift our region anymore to the right, since we have reached the end of the matrix , so what we do is that we shift it one row down and start again from the left side. The next selected region would be

again, we perform convolution of with

as you have guessed at this point, the last step is to shift our selected region to the right of one column and perform again convolution. Our selected region will now be

and the convolution will give the result

Now we cannot shift our region anymore, neither right nor down. We have calculated values: , , and . Those elements will form the resulting tensor of the convolution operation giving us the tensor

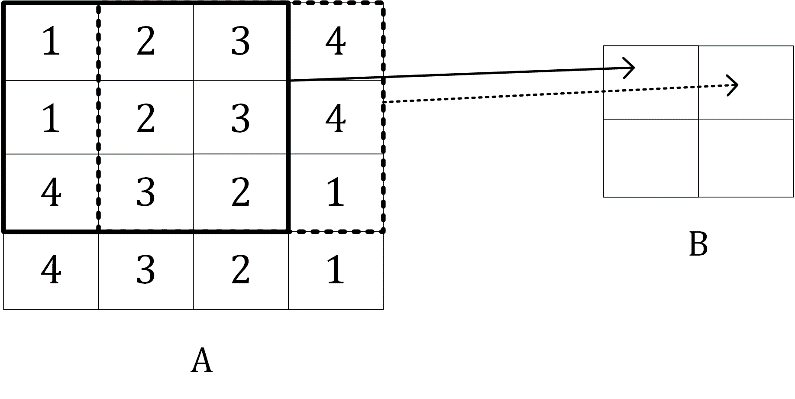
the same process can be applied when the tensor is bigger. You will simply get a bigger resulting tensor, but the algorithm to get the elements is the same. Before moving on, there is still a small detail that we need to discuss, and that is the concept of stride. In the process above we have moved our region always one column to the right and one row down. The number of rows and columns, in this example 1, is called stride and is often indicated with . Stride means simply that we would shift our region two columns to the right and two rows down. Something else that we need to discuss is the size of the selected region in the input matrix . The dimensions of the selected region that we shifted around in the process, must be the same as of the kernel used. If you use a kernel, then you will need to select a region in . In general, given a kernel you will select a region in .

In a more formal definition, convolution with stride in the neural network context, is a process that takes a tensor of dimensions and a kernel of dimensions and gives as output a matrix of dimensions with

Where we have indicated with the integer part of (in the programming world, this is often called the floor of ). A proof of this formula would take too long to discuss but is easy to see why it is true (try to derive it). To make things a bit easier we will suppose that is odd. You will see soon why this is important (although not fundamental). Let's start explaining formally the case with a stride . The algorithm generates a new tensor from an input tensor and a kernel according to the formula

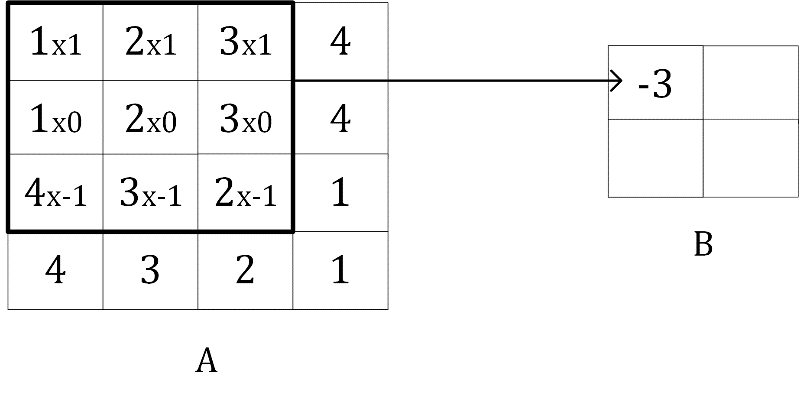
The formula is cryptic and is very difficult to understand. Let's check some more examples to grasp the meaning better. In Figure 8-1 you can see a visual explanation of how convolution works. Suppose you have a filter. Then in the Figure you can see that the top left 9 elements of the matrix , marked by a square drawn with a black continuous line, are the one used to generate the first element of the matrix according to the formula above. The elements marked by the square drawn with a dotted line are the one used to generate the second element and so on. To reiterate what we discuss in the example at the beginning, the basic idea is that each element of the square from matrix is multiplied by the correspondent element of the Kernel and all the numbers are summed. The sum is then the element of the new matrix . After having calculated the value for you shift the region you are considering in the original matrix of one column to the right (the square indicated in Figure 8-1 with a dotted line) and repeat the operation. You continue to shift your region to the right until you reach the border and then you move one element down and start again from the left and you continue in this fashion until the lower right angle of the matrix. The same kernel is used for all the regions in the original matrix.

Figure 8-1: a visual explanation of convolution.



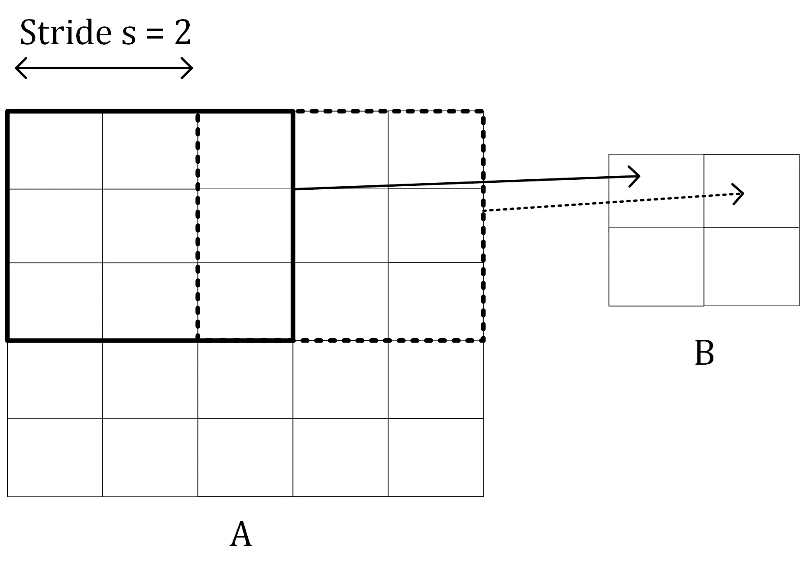
Given the kernel for example you can see in Figure 8-2 which element of gets multiplied by which element in and the result for the element , that is nothing else as the sum of all the multiplications

Figure 8-2: a visualisation of convolution with the kernel



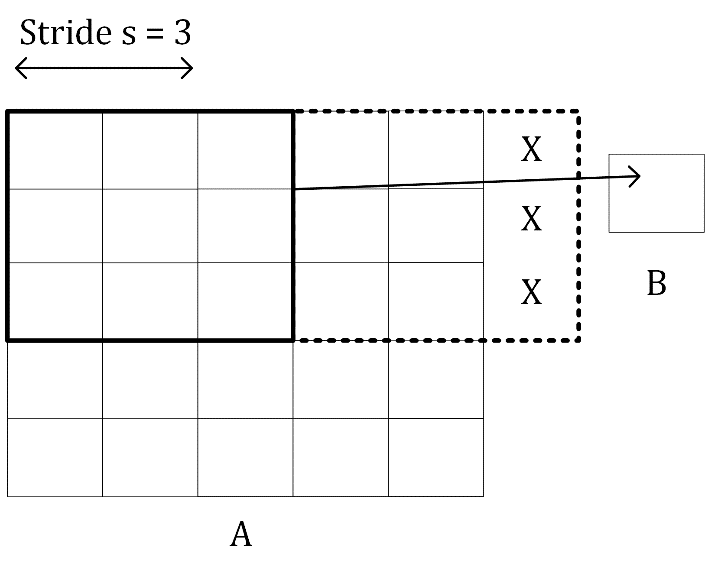
In Figure 8-3 you can see an indicative example of convolution with stride .

Figure 8-3: an visual explanation of convolution with stride



The reason why the dimension of the output matrix takes only the floor (the integer part) of can be seen intuitively in Figure 8-4. If , what can happen, depending on the dimensions of , is that at a certain point you cannot shift your window on matrix (the black square you can see in Figure 8-3 for example) anymore, and you cannot cover all the matrix completely. In Figure 8-4 you can see how you would need an additional column on the right of matrix (marked by many X) to be able to perform the convolution operation. In Figure 8-4 we have chosen , and since we have and , will a scalar as a result

Figure 8-4: a visual explanation why the floor function is needed when evaluating the resulting matrix dimensions



You can easily see from Figure 8-4 how with a region you can only cover the top left region of , since with stride you would end up outside and therefore can consider one region only for the convolution operation, therefore ending up with a scalar for the resulting tensor .

Let's now make a few additional examples to make this formula even clearer. Let's start with a small matrix

And let's consider the kernel

and stride . The convolution will be given by

and the result will be a scalar, since , therefore

If you consider now a matrix with dimensions , or , and you will get as output a matrix with dimensions , since

For example, you can verify that given

And

we have with stride

let's verify one of the elements: with the formula I gave you. We have

Note that the formula I gave you for the convolution works only for stride , but can be easily generalized for other values of .

This calculation is very easy to implement in Python. The following function can evaluate the convolution of two matrices easily enough for (you can do it in Python with already existing functions, but I think is instructive to see how to do it from scratch)

import numpy as np

def conv\_2d(A, kernel):

output = np.zeros([A.shape[0]-(kernel.shape[0]-1), A.shape[1]-(kernel.shape[0]-1)])

for row in range(1,A.shape[0]-1):

for column in range(1, A.shape[1]-1):

output[row-1, column-1] = np.tensordot(A[row-1:row+2, column-1:column+2], kernel)

return output

note that the input matrix does not even need to a square one, but it is assumed that the kernel is and that its dimension is odd. The previous example can be evaluated with the following code

A = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]])

K = np.array([[1,2,3],[4,5,6],[7,8,9]])

print(conv\_2d(A,K))

this gives the result

[[ 348. 393.]   
[ 528. 573.]]

# Examples of convolution

Now let's try to apply the kernels we have defined at the beginning to a test image and let's see the results. As a test image let's create a chessboard of dimensions pixels with the code

chessboard = np.zeros([8\*20, 8\*20])

for row in range(0, 8):

for column in range (0, 8):

if ((column+8\*row) % 2 == 1) and (row % 2 == 0):

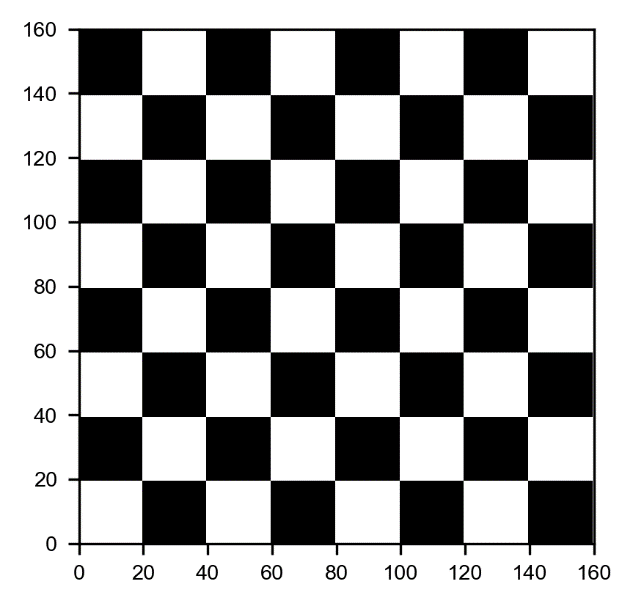
chessboard[row\*20:row\*20+20, column\*20:column\*20+20] = 1

elif ((column+8\*row) % 2 == 0) and (row % 2 == 1):

chessboard[row\*20:row\*20+20, column\*20:column\*20+20] = 1

In Figure 8-5 you can see how the chessboard looks like.

Figure 8-5: the chessboard image generated with code.



Now let's try to apply convolution to this image with the different kernels with stride .

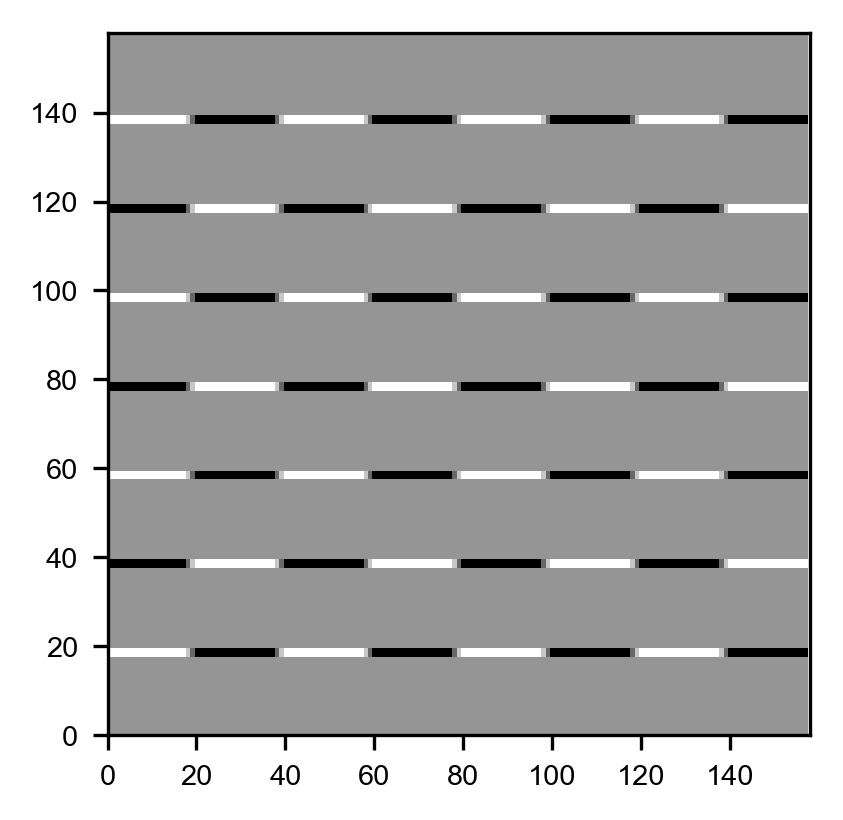
Using the kernel will detect the horizontal edges. This can be applied with the code

edgeh = np.matrix('1 1 1; 0 0 0; -1 -1 -1')

outputh = conv\_2d (chessboard, edgeh)

In Figure 8-6 you can see how the output looks like.

Figure 8-6: the result of performing a convolution between the kernel and the chessboard image.



Now you can understand why I said that this kernel detects horizontal edges. Additionally, this kernel detects if you go from light to dark or vice versa. Note this image is only pixels as expected since

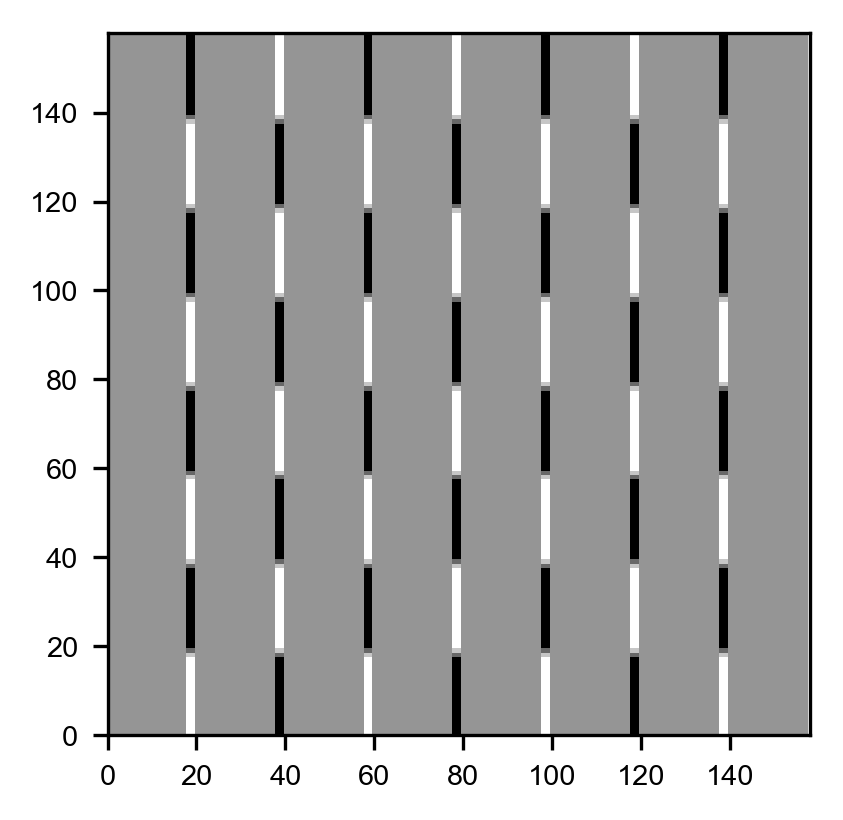
Now let's apply with the code

edgev = np.matrix('1 0 -1; 1 0 -1; 1 0 -1')

outputv = conv\_2d (chessboard, edgev)

This gives the result in Figure 8-3.

Figure 8-7: the result of performing a convolution between the kernel and the chessboard image.

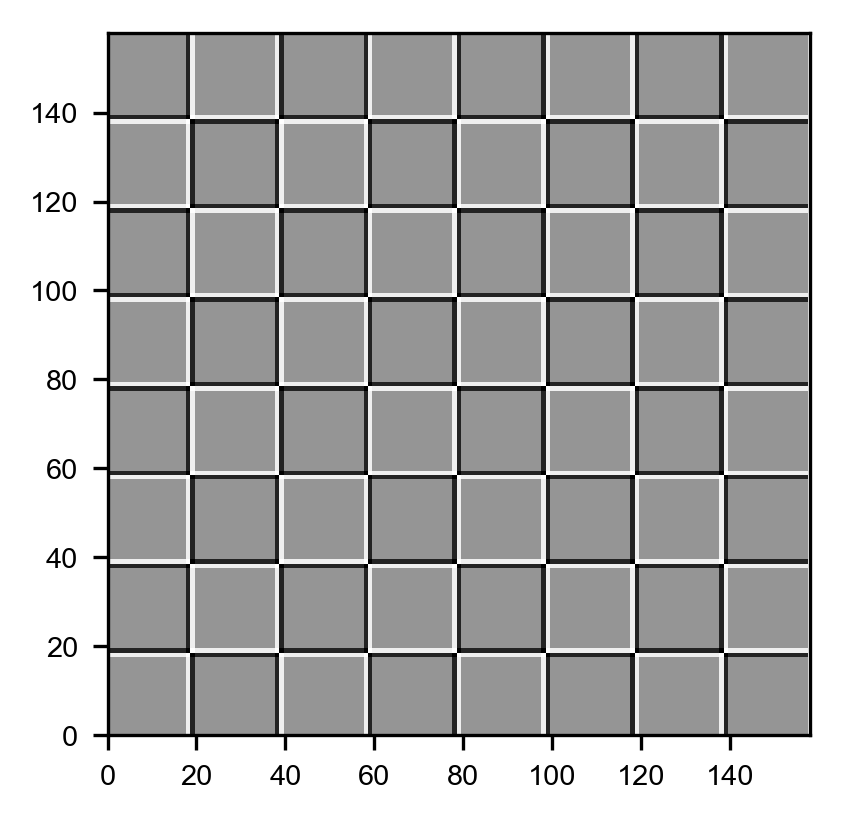


Now we can use the kernel

edgel = np.matrix ('-1 -1 -1; -1 8 -1; -1 -1 -1')  
outputl = conv\_2d (chessboard, edgel)

This gives the result in Figure 8-8.

Figure 8-8: the result of performing a convolution between the kernel and the chessboard image.



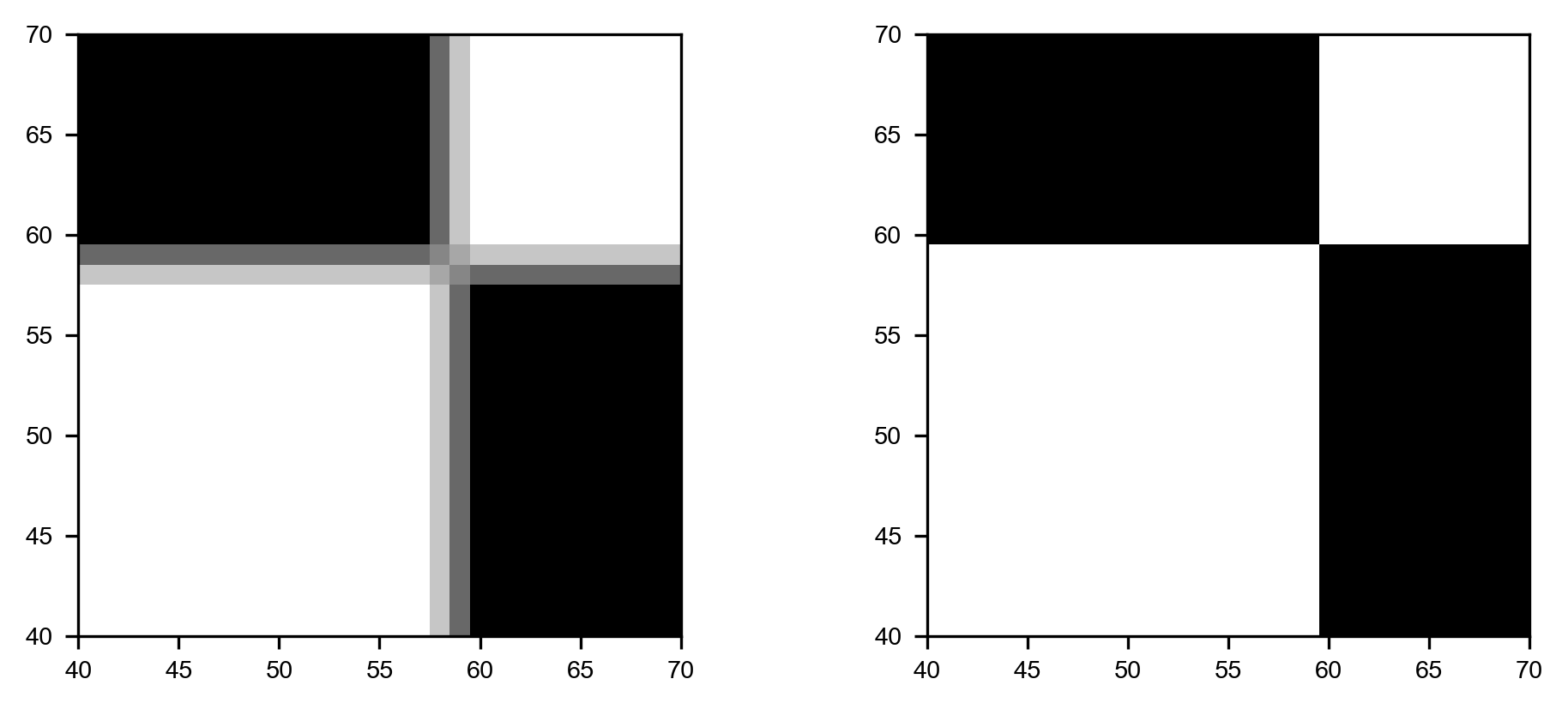
And finally, we can apply the blurring kernel

edge\_blur = -1.0/9.0\*np.matrix('1 1 1; 1 1 1; 1 1 1')

output\_blur = conv\_2d (chessboard, edge\_blur)

In Figure 8-9 you can see two plots: on the left the blurred image and on the right the original one. The images show only a small region of the original chessboard to make the blurring clearer.

Figure 8-9: the effect of the blurring kernel . On the left the blurred image and on the right the original one.



To finish this section let's try to understand better how the edges can be detected. Let's consider the following matrix with a sharp vertical transition, since the left part is full of 10 and the right part full of 0.

ex\_mat = np.matrix('10 10 10 10 0 0 0 0; 10 10 10 10 0 0 0 0; 10 10 10 10 0 0 0 0; 10 10 10 10 0 0 0 0; 10 10 10 10 0 0 0 0; 10 10 10 10 0 0 0 0; 10 10 10 10 0 0 0 0; 10 10 10 10 0 0 0 0')

This looks like this

matrix([[10, 10, 10, 10, 0, 0, 0, 0],   
 [10, 10, 10, 10, 0, 0, 0, 0],   
 [10, 10, 10, 10, 0, 0, 0, 0],   
 [10, 10, 10, 10, 0, 0, 0, 0],   
 [10, 10, 10, 10, 0, 0, 0, 0],   
 [10, 10, 10, 10, 0, 0, 0, 0],   
 [10, 10, 10, 10, 0, 0, 0, 0],   
 [10, 10, 10, 10, 0, 0, 0, 0]])

and let's consider the kernel . We can perform the convolution with the code

ex\_out = conv\_2d (ex\_mat, edgev)

The result is

array([[ 0., 0., 30., 30., 0., 0.],   
 [ 0., 0., 30., 30., 0., 0.],   
 [ 0., 0., 30., 30., 0., 0.],   
 [ 0., 0., 30., 30., 0., 0.],   
 [ 0., 0., 30., 30., 0., 0.],   
 [ 0., 0., 30., 30., 0., 0.]])

in Figure 8-10 you can see the original matrix (on the left) and the output of the convolution on the right. The convolution with the kernel has clearly detected the sharp transition in the original matrix marking with a vertical black line where the transition from black to white happens. For example, consider

Note that in the input matrix

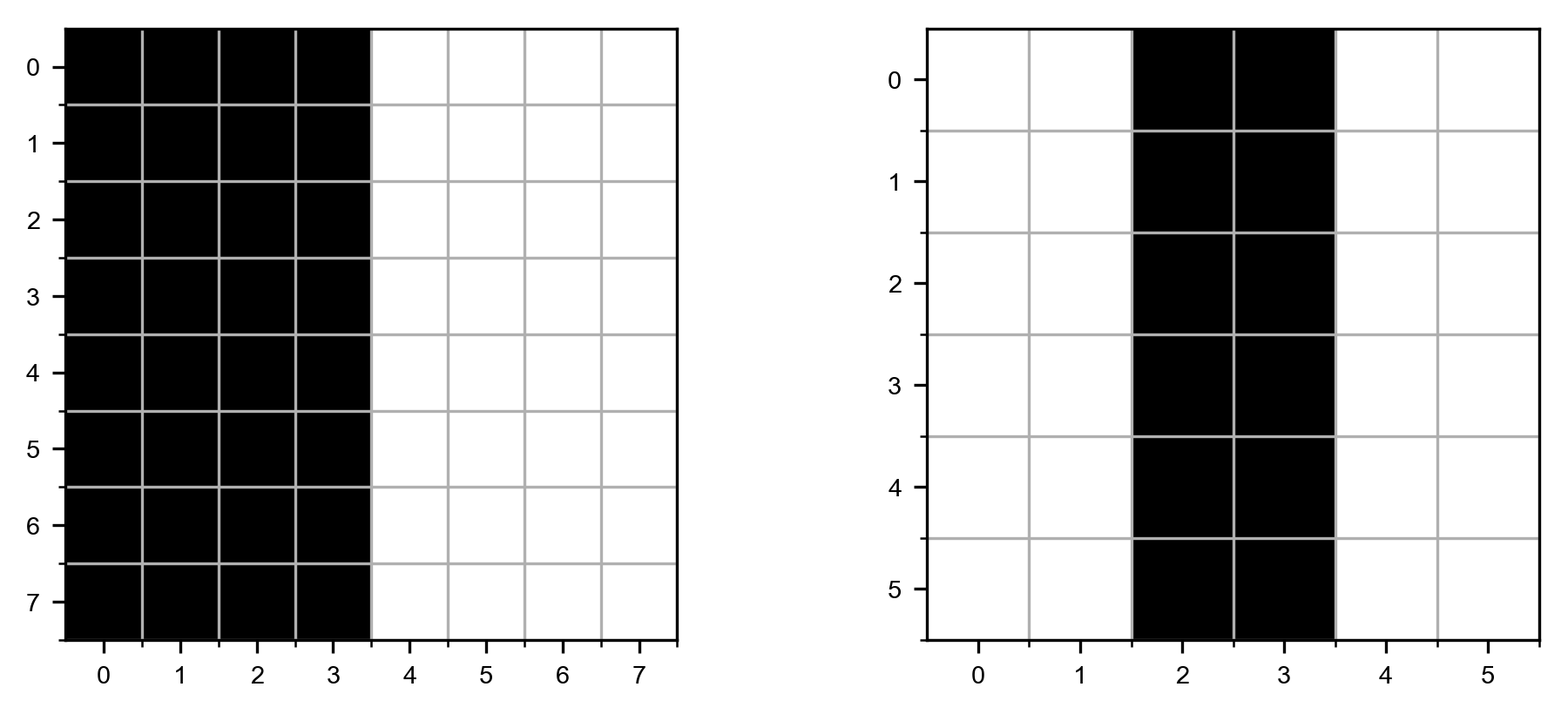
there is no transition, as all the values are the same. On the contrary if you consider you need to consider this region of the input matrix

where there is a clear transition, since the most right column is made of zeros, and the rest of 10. You get now a different result

And this is exactly how, as soon as there is a big change in values along the horizontal direction the convolution will return a high value since the values multiplied by the column with 1 in the Kernel will be bigger. When on the opposite there is a transition from small to high values along the horizontal axis, the elements multiplied by -1 will give a result that is bigger in absolute value and therefore the final result will be negative and big in absolute value. This is the reason why this kernel can also detect if you pass from a light color to a darker color or vice versa. In fact, if you consider the opposite transition (from 0 to 10) in a hypothetical different matrix you would have

since this time, we move from 0 to 10 along the horizontal direction.

Figure 8-10: the resul of the convolution of the matrix ex\_mat with the kernel as described in the text.



Note how, as expected, the output matrix has dimensions since the original matrix has dimensions and the kernel is .

# Pooling

Pooling is the second operation that is fundamental in CNNs. This operation is much easier to understand than convolution. To understand it let's again make a concrete example, and let's consider what is called max pooling. Let's again consider our matrix we have discussed during our convolution discussion

To perform max pooling, we need to define a region of size , analogous to what we did for convolution. Let's consider . What we need to do is to start on the top left corner of our matrix and select a region, in our case from . Here we would select

Or the elements marked in bold face in the matrix here below

from the elements selected, , , and , the max pooling operation select the maximum value giving a result that we will indicate with

then we need to shift our window two columns, typically the same number of columns the selected region has, to the right and select the elements marked in bold

Or in other words the smaller matrix

The max-pooling algorithm will then select the maximum of the values giving a result that we will indicate with

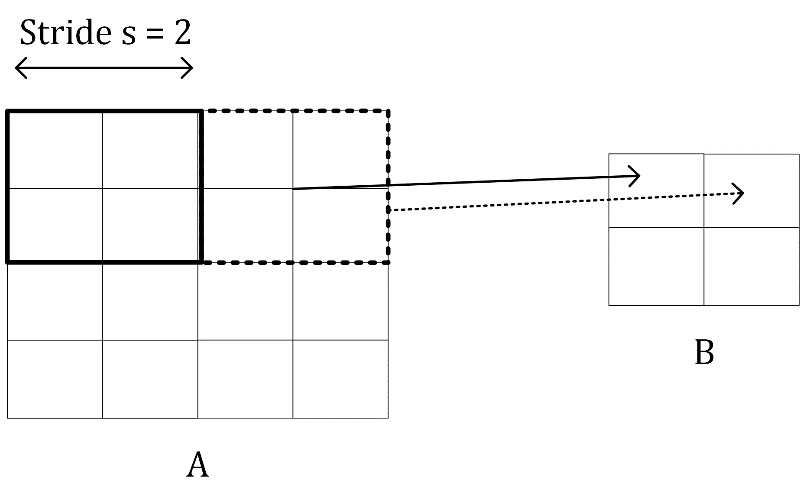
At this point we cannot shift the region to the right anymore, so we shift it two rows down and start the process again from the left side of , selecting the elements marked in bold below and getting the maximum and calling it .

The stride in this context has the same meaning we have already discussed in convolution. It simply the number of rows or columns you move your region when selecting the elements. Finally, we select the last region in the bottom lower part of , selecting the elements , , and . We then get the maximum and we then call it . With the values we obtain in this process, in the example the four values , , and , we will build an output tensor

In the example we have . Basically, this operation takes as input a matrix , a stride , and a kernel size (the dimension of the region we selected in the example before) and return a new matrix with dimensions given by the same formula we discussed for convolution

To reiterate the idea is to start from the top left of your matrix , take a region of dimensions , apply the max function to the selected elements, then shift the region of elements toward the right, select a new region again of dimensions , apply the function to its values and so on. In Figure 8-11 you can see how you would select the elements from a matrix with a stride

Figure 8-11: a visualization of pooling with stride



For example, applying max-pooling to the input

Will get you the result (is very easy to verify it)

since is the maximum of the values marked in bold

is the maximum of the values marked in bold below

And so on. Is worth mentioning another way of doing pooling, although not as widely used as max-pooling: average pooling. Instead of returning the maximum of the selected values it returns the average.

The most used pooling operation is "max pooling". "Average pooling" is not as widely used but can be found in specific network architectures.

## Padding

Something is worth mentioning, and that is the concept of padding. Sometime, when dealing with images, is not optimal to get a result from a convolution operation that has dimensions that are different from the original image. So sometimes you do what is called "padding". Basically, the idea is very simple: it consists in adding rows of pixels on top, bottom and columns of pixels on the right and on the left of the final images filled with some values to make the resulting matrices the same size of the original one. Some strategies are filling the added pixels with zeros, with the values of the closest pixels and so on. For example, in our example our ex\_out matrix with zero padding would like like this

array([[ 0., 0., 0., 0., 0., 0., 0., 0.],  
 [ 0., 0., 0., 30., 30., 0., 0., 0.],   
 [ 0., 0., 0., 30., 30., 0., 0., 0.],   
 [ 0., 0., 0., 30., 30., 0., 0., 0.],   
 [ 0., 0., 0., 30., 30., 0., 0., 0.],   
 [ 0., 0., 0., 30., 30., 0., 0., 0.],   
 [ 0., 0., 0., 30., 30., 0., 0., 0.],  
 [ 0., 0., 0., 0., 0., 0., 0., 0.]])

The use and reasons behind padding goes beyond the scope of this book but is important to know that this exists. Only as a reference, in case you use padding (the width of the rows and columns you use as padding) the final dimensions of the matrix , in case of both convolution and pooling is given by

When dealing with real images, you always have color images, coded in 3 channels: RGB. That means that you need to do convolution and pooling in three dimensions: width, height and color channel. This will add a layer of complexity to the algorithms.

# Building blocks of a CNN

Basically, convolutions and pooling operations are used to build the layers are used in CNNs. In CNNs typically you can find the following layers

* Convolutional layers
* Pooling layers
* Fully connected layers

Fully connected layers are exactly what we have already seen in all previous chapters: a layer where neurons are connected to all neurons of previous and subsequent layers. You know them already. But the first two require some additional explanation.

## Convolutional layers

A convolutional layer takes as input a tensor (can be 3-dimensional, due to the 3 color channels), for example an image of certain dimensions, applies a certain number of kernels, typically 10, 16, or even more, add a bias, apply ReLu activation functions (for example) to introduce non-linearity to the result of the convolution, and produce an output matrix . If you remember the notation we used in the previous chapters, the result of the convolution will have the role of that we discussed in Chapter 3.

Now in the previous sections I have shown you some example of applying convolutions with just one kernel. How can you apply several kernels at the same time? Well the answer is very simple. The final tensor (I use now the word tensor since it will not be a simple matrix anymore) will have now not dimensions but . Let's indicate the number of kernels you want to apply with (the is used since sometimes people talk about channels). You simply apply each filter to the input independently and stack the results. So instead of a single matrix with dimensions you get a final tensor of dimensions . That means that

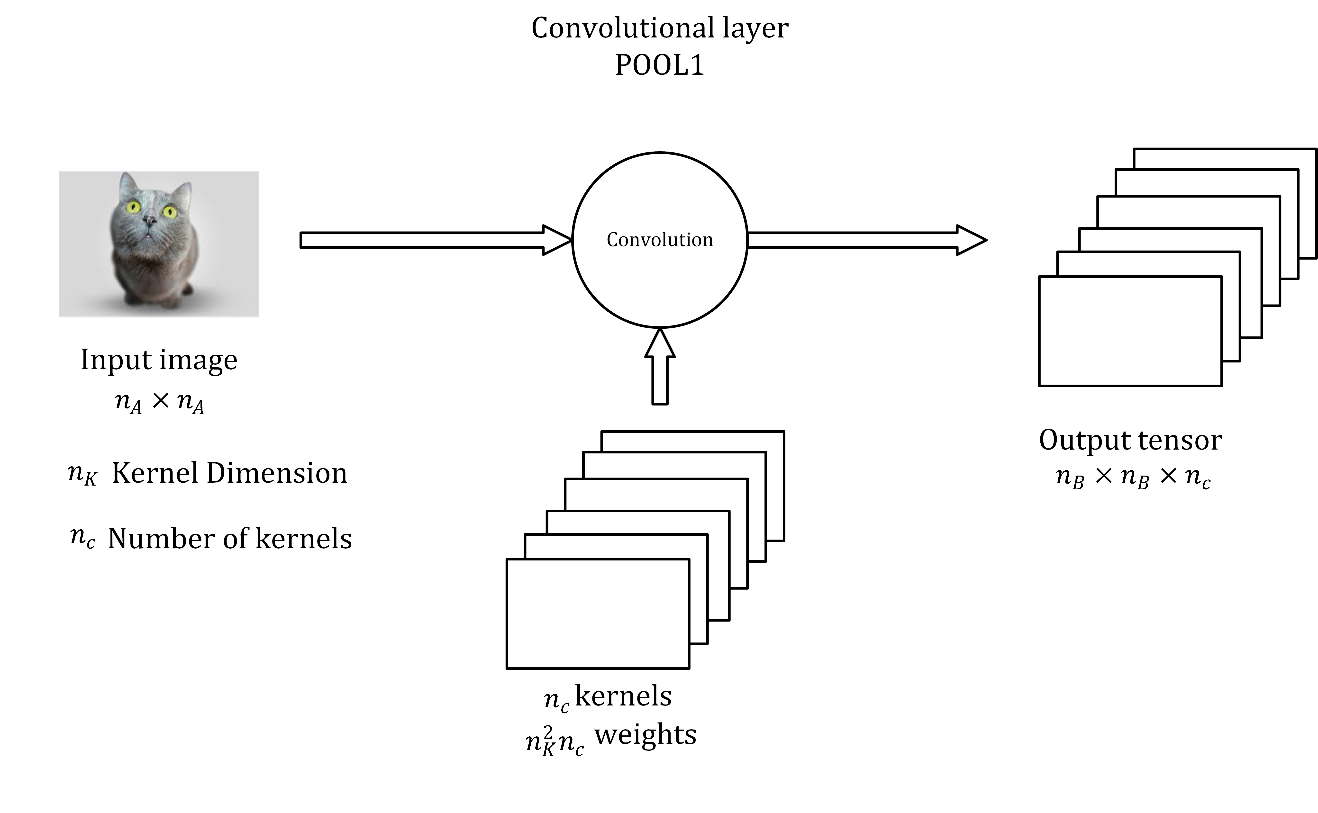
will be the output of convolution of the input image with the first kernel,

will be the output of convolution with the second kernel and so on. The convolution layer is nothing else than something that transform the input into an output tensor. But what are the weights in this layer? The weights, or the parameter that the network learn during the training phase, are the elements of the kernel themselves. We have discussed that we have kernels, each of dimensions. That means that we have parameter in a convolutional layer.

Note that the number of parameters that you have in a convolutional layer, , is independent from the input image size. This fact helps in reducing overfitting, especially when dealing with big size input images.

Sometime this layer is indicated with the word POOL and then a number. In our case we could indicate this layer with POOL1. In Figure 8-12 you can see a representation of a convolutional layer. The input image gets transformed by applying convolution with kernels in a tensor of dimensions .

Figure 8-12: a representation of a convolutional layer[[1]](#footnote-1).



Of course, a convolutional layer must not necessarily be placed immediately after the inputs. A convolutional layer may get as input the output of any other layer of course. Keep in mind that usually your input image will have dimensions , since an image in color has 3 channels: Red, Green and Blue. A complete analysis of the tensors involved in a CNN when considering color images will go beyond the scope of this book. Very often in diagrams the layer is simply indicated as a cube or a square.

## Pooling layers

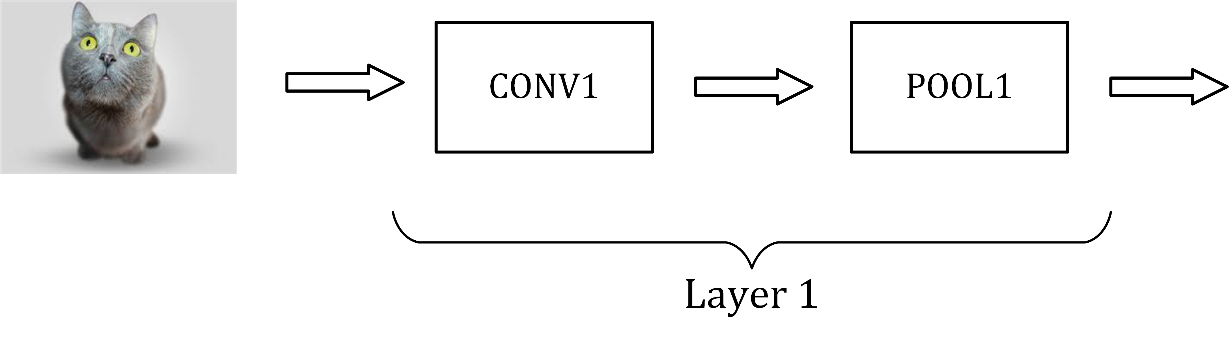
A pooling layer is usually indicated with POOL and a number: for example, POOL1. It takes as input a tensor and gives as output another tensor after applying pooling to the input.

A pooling layer has no parameter to learn, but it introduces additional hyperparameters: and stride . Typically, in pooling layers you don't use any padding, since one of the reason to use pooling is often to reduce the dimensionality of the tensors.

## Stacking layers together

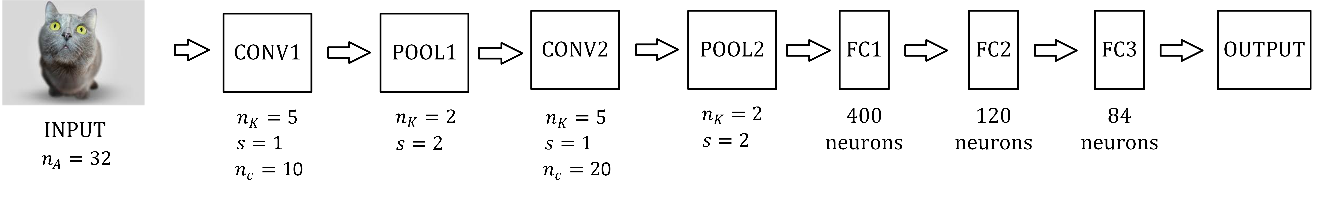
In CNNs you usually stack convolutional and pooling layer together. One after the other. In Figure 8-13 you can see a convolutional and a pooling layer stack. A convolutional is always followed by a pooling layer. Sometime the two together are called a layer. The reason is that a pooling layer has no learnable weights and therefore it is simply seen as a simple operation that is associated with the convolutional layer. So be aware when you read papers or blogs and check what they intend.

Figure 8-13: a representation of how to stak convolutional and pooling layers.



To conclude this part on CNN in Figure 8-14 you can see an example of a CNN. In Figure 8-14 you can see an example like the very famous LeNet-5 network, on which you can read more here: <https://goo.gl/hM1kAL>. You have the inputs, then two times convolution-pooling layer, then 3 fully connected layers and then an output layers, where you may have your softmax function in case, for example, you perform multiclass classification. I put some indicative numbers in the Figure to give you an idea of the size of the different layers.

Figure 8-14: a representation of a CNN similar to the famous LeNet-5 network.



## An example of a CNN

Let's try to build one such a network to give you a feeling of how the process would work and how the code looks like. We will not do any hyperparameter tuning or optimization to keep the section understandable. We will build the following architecture with the following layers in this order:

* Convolution layer 1: 6 Filters , stride
* Max pooling layer 1 with a window stride
* We then apply ReLU to the output of the previous layer
* Convolution layer 2: 16 Filters , stride
* Max pooling layer 2 with a window stride
* We then apply ReLU to the output of the previous layer
* Fully Connected Layer with neurons with activation function ReLU
* Fully Connected Layer with 10 neurons for classification of the Zalando dataset
* Softmax output neuron

We will import the Zalando dataset as we did in Chapter 3:

data\_train = pd.read\_csv('fashion-mnist\_train.csv', header = 0)

data\_test = pd.read\_csv('fashion-mnist\_test.csv', header = 0)

check there to see how to get the files in detail. Then let's prepare the data

labels = data\_train['label'].values.reshape(1, 60000)

labels\_ = np.zeros((60000, 10))

labels\_[np.arange(60000), labels] = 1

labels\_ = labels\_.transpose()

train = data\_train.drop('label', axis=1)

and

labels\_dev = data\_test['label'].values.reshape(1, 10000)

labels\_dev\_ = np.zeros((10000, 10))

labels\_dev\_[np.arange(10000), labels\_dev] = 1

test = data\_dev.drop('label', axis=1)

note that in this case, differently than in Chapter 3, we will use the transpose of all tensors, meaning that in each row we will have an observation. In Chapter 3 each observation was in a column. If you check the dimensions with the code

print(labels\_.shape)

print(labels\_dev\_.shape)

you will get the results

(60000, 10)   
(10000, 10)

In Chapter 3 the dimensions were exchanged. The reason is that to develop the convolutional and pooling layers we will use functions that are provided by tensorflow, since developing them from scratch would require too much time. And for some tensorflow functions is easier if the tensors have along the rows the different observations. As in Chapter 3 we need to normalize the data

train = np.array(train / 255.0)

dev = np.array(dev / 255.0)

labels\_ = np.array(labels\_)

labels\_test\_ = np.array(labels\_test\_)

we can now start to build our network

x = tf.placeholder(tf.float32, shape=[None, 28\*28])

x\_image = tf.reshape(x, [-1, 28, 28, 1])

y\_true = tf.placeholder(tf.float32, shape=[None, 10])

y\_true\_scalar = tf.argmax(y\_true, axis=1)

the one line that require an explanation is the second: x\_image = tf.reshape(x, [-1, 28, 28, 1]). Remember that the convolutional layer will require the 2-dimensional image, and not a flattened list of gray values of the pixel as we have done in Chapter 3, where our input was a vector with 784 ( elements.

One of the biggest advantage of CNNs is that they use the 2-dimensiona information contained in the input image, this is the reason why the input of convolutional layers are 2-dimensional images, and not a flattened vector.

When building CNNs is typical to define functions to build the different layers. In this way hyperparameter tuning later will be easier, as we have seen previously. Another reason is that when we put all the pieces together, with the functions the code will be much more readable. The function names should be self-explanatory. Let's start with a function to build the convolutional layer. Note that tensorflow documentation uses the term filter, so this is what we will use here in the code.

def new\_conv\_layer(input, num\_input\_channels, filter\_size, num\_filters):

shape = [filter\_size, filter\_size, num\_input\_channels, num\_filters]

weights = tf.Variable(tf.truncated\_normal(shape, stddev=0.05))

biases = tf.Variable(tf.constant(0.05, shape=[num\_filters]))

layer = tf.nn.conv2d(input=input, filter=weights, strides=[1, 1, 1, 1], padding='SAME')

layer += biases

return layer, weights

at this point we will initialize the weights from a truncated normal distribution, the biases as a constant and then we will use stride . The stride is a list since it gives the stride in different dimensions. In our examples we have gray images, but we could also have RGB for example, therefore having more dimensions: the 3 color channels.

The pooling layer is easier, since it has no weights

def new\_pool\_layer(input):

layer = tf.nn.max\_pool(value=input, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

return layer

then let's define a function that applies the activation function, in our case a ReLU, to the previous layer

def new\_relu\_layer(input\_layer):

layer = tf.nn.relu(input\_layer)

return layer

and finally, we need one to build the fully connected layer

def new\_fc\_layer(input, num\_inputs, num\_outputs):

weights = tf.Variable(tf.truncated\_normal([num\_inputs, num\_outputs], stddev=0.05))

biases = tf.Variable(tf.constant(0.05, shape=[num\_outputs]))

layer = tf.matmul(input, weights) + biases

return layer

the new tensorflow functions that we have used are tf.nn.conv2d, that build a convolutional layer, and tf.nn.max\_pool that build a pooling layer with max pooling, as you can imagine from the name. We don't have the space here to go into details in what each function does, but you can find a lot of information in the official documentation. Now let's put everything together and actually build the network we described at the beginning

layer\_conv1, weights\_conv1 = new\_conv\_layer(input=x\_image, num\_input\_channels=1, filter\_size=5, num\_filters=6)

layer\_pool1 = new\_pool\_layer(layer\_conv1)

layer\_relu1 = new\_relu\_layer(layer\_pool1)

layer\_conv2, weights\_conv2 = new\_conv\_layer(input=layer\_relu1, num\_input\_channels=6, filter\_size=5, num\_filters=16)

layer\_pool2 = new\_pool\_layer(layer\_conv2)

layer\_relu2 = new\_relu\_layer(layer\_pool2)

now we need to create the fully connected layer, but to use layer\_relu2 as input we first need to flatten it, since now is still 2-dimensional

num\_features = layer\_relu2.get\_shape()[1:4].num\_elements()

layer\_flat = tf.reshape(layer\_relu2, [-1, num\_features])

and then we can create the final layers

layer\_fc1 = new\_fc\_layer(layer\_flat, num\_inputs=num\_features, num\_outputs=128)

layer\_relu3 = new\_relu\_layer(layer\_fc1)

layer\_fc2 = new\_fc\_layer(input=layer\_relu3, num\_inputs=128, num\_outputs=10)

Now let's evaluate the predictions to be able to evaluate the accuracy later

y\_pred = tf.nn.softmax(layer\_fc2)

y\_pred\_scalar = tf.argmax(y\_pred, axis=1)

the array y\_pred\_scalar will contain the class number as a scalar. Now we need to define the cost function, and again we will use an already existing tensorflow function to make our life easier, and to maintain the size of this chapter reasonable

cost = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(logits=layer\_fc2, labels=y\_true))

as usual we need an optimizer

optimizer = tf.train.AdamOptimizer(learning\_rate=1e-4).minimize(cost)

and we can finally define the operations to evaluate the accuracy

correct\_prediction = tf.equal(y\_pred\_scalar, y\_true\_scalar)

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))

Now is time to train our network. We will use mini-batch gradient descent with a batch size of 100 and we will train our network for just 10 epochs. We can define the variables

num\_epochs = 10

batch\_size = 100

the training can be achieved with

with tf.Session() as sess:

sess.run(tf.global\_variables\_initializer())

for epoch in range(num\_epochs):

train\_accuracy = 0

for i in range(0, train.shape[0], batch\_size):

x\_batch = train[i:i + batch\_size,:]

y\_true\_batch = labels\_[i:i + batch\_size,:]

sess.run(optimizer, feed\_dict={x: x\_batch, y\_true: y\_true\_batch})

train\_accuracy += sess.run(accuracy, feed\_dict={x: x\_batch, y\_true: y\_true\_batch})

train\_accuracy /= int(len(labels\_)/batch\_size)

dev\_accuracy = sess.run(accuracy, feed\_dict={x:dev, y\_true:labels\_dev\_})

If you run this code (it took roughly 10 minutes on my laptop), it will start, after just one epoch, with a training accuracy of 63.7%, and after 10 epochs it will reach a training accuracy of 86% (also on the dev set). Remember that with the first dumb network we developed in Chapter 3 with 5 neurons in one layer we reached 66% with mini-batch gradient descent. We have trained our network here only for 10 epochs. You can get much higher accuracy if you train longer. Additionally, note we have not done any hyperparameter tuning so this would get you much better results if you spent time tuning the parameters.

As you may have noticed, every time you introduce a convolutional layer you will introduce new hyperparameters for each layer:

* Kernel size
* Stride
* Padding

Those will need to be tuned to get optimal results. Typically, researchers tend to use existing architectures for specific tasks that have been already optimized by other practitioners and are well documented in papers.

# Introduction to RNNs

RNN are very different from CNNs, and typically used when dealing with sequential information, in other words for data for which the order matters. The typical example given is a series of words in a sentence. You can easily understand how the order of words in a sentence can make a big difference. For example, saying "the man eat the rabbit" has a different meaning than "the rabbit eat the man", the only difference being the order of the words, and who gets eaten by whom. You can use RNNs to predict for example the next word in a sentence. Take for the example the sentence: "Paris is the capital of", it is easy to complete the sentence with "France", that means that there is information about the final word of the sentence encoded in the previous words, and that information is what RNNs wants to exploit to predict the next terms in the sequence. The name recurrent comes from how they work: the network applies the same operation on each element of the sequence, accumulating information about the previous terms. To summarize:

* RNNs make use of sequential data and uses the information encoded in the order of the terms in the sequence
* RNNs apply the same kind of operation to all terms in the sequence and build a memory of the previous terms in the sequence to predict the next term

Before trying to understand a bit better how they work, let's consider a few important use cases where they can be applied to give you a feeling of the range of applications possible.

* Generating Text: predicting probability of words given previous set of words. For example, you can easily generate text that looks like Shakespeare with RNNs, as A. Karpathy has done in his blog here: https://goo.gl/FodLp5.
* Translation: given a set of words in a language you want words in a different language
* Speech recognition: given a series of audio signals (words), we want to predict the sequence of letters forming the words spoken
* Generating image labels: with CNNs, RNNs can be used to generate labels for images. Check the paper " Deep Visual-Semantic Alignments for Generating Image Descriptions" by A. Karpathy on the subject (<https://goo.gl/8Ja3n2>). Be aware that this is a rather advanced paper that requires quite some mathematical background.
* Chatbots: sequence of words given as input, RNNs try to generate answers to the input

As you can imagine, to solve those problems you will need sophisticated architectures that are not easy to describe in a few sentence and that require a deeper (pun intended) understanding of how RNNs are working, things that would go beyond the scope of this chapter and book.

## Notation

Let's consider the sequence: "Paris is the capital of France". This sentence will be fed to a RNN one word at a time: first "Paris", then "is", then "the" and so on. We will have

* "Paris" will be the first word of the sequence: w1 = 'Paris'
* "is" will be the second word of the sequence: w2 = 'is'
* "the" will be the third word of the sequence: w3 = 'the'
* "capital" will be the fourth word of the sequence: w4 = 'capital'
* "of" will be the fifth word of the sequence: w5 = 'of'
* "France" will be the sixth word of the sequence: w6 = 'France'

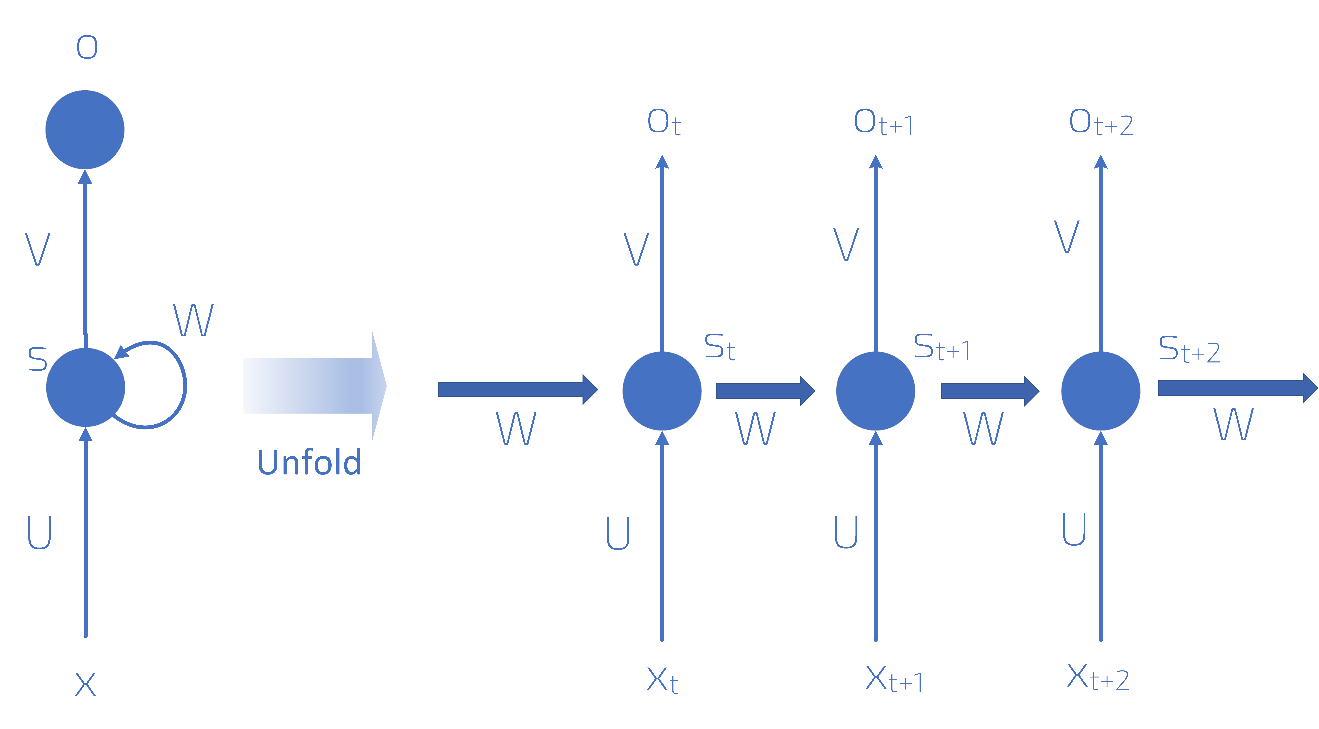
The words will be fed into the RNN in the following order: w1, w2, w3, w4, w5 and then w6. The different words will be processed by the network one after the other, or as someone like to say at different time points. Usually is said that if word w1 is processed at time , then w2 is processed at time , w3 at time and so on. The time is not related to a real time but is meant to suggest the fact that each element in the sequence is processed sequentially and not in parallel. The time is also not related to computing time or anything related to it. And the increment of in does not have any meaning, it simply means that we are talking about the next element in our sequence. You may find the following notations when reading papers, blogs or books:

* : the input at time . For example w1 could be the input at time , w2 at time and so on.
* : is the notation with which the internal memory, that we have not defined yet, at time is indicated. This quantity will contains the accumulated information on the previous terms in the sequence we discussed above. An intuitive understanding of it will have to suffice, since a more mathematical definition would require a too detailed explanation.
* is the output of the network at time , or in other words after all the elements of the sequence until , including the element has been fed to the network.

## Basic idea of RNNs

Typically a RNN is indicated in the literature as the leftmost part of Figure 8-15. The notation used is indicative and has the goal of simply indicating the different elements of the network: the inputs, the internal memory, one set of weights and another set of weights. In reality, this schematic representation is simply a way of depicting the real structure of the network that you can see on the right part of Figure 8-15, sometime called the unfolded version of the network.

Figure 8-15: a schematic representation of a RNN



The right part of Figure 8-15 should be read left to right. The first neuron in the Figure do its evaluation at an indicative time and produce an output and creates an internal memory state . The second neuron, that do its evaluation at a time , after the first neuron, gets as input both the next element in the sequence and the previous memory state . The second neuron then generates an output and a new internal memory state The third neuron then (the one at the extreme right of Figure 8-15) gets as input the new element of the sequence and the previous internal memory state and the process proceed in this way for a finite number of neurons. You can see in Figure 8-15 that there are two sets of weights: and . One set (here indicated with ) is used for the internal memory states and one for the sequence element. Typically, each neuron will generate the new internal memory state with a formula that will look like this

Where we have indicated with one of the activation function we have already seen as ReLU or tanh. Additionally, the previous formula will be of course multi-dimensional. can be understood as the memory of the network at time . The number of neurons (or time steps) that can be used is a new hyperparameter that needs to be tuned, depending on the problem. Research has shown that when this number is too big, the network has big problems during training.

Something very important to note is that at each time step the weights are not changing. We are performing the same operation at each step, simply changing the inputs every time we perform an evaluation. Additionally, In Figure 8-15 I have for every step an output in the diagram (, and ) but typically this is not necessary. In our example where we want to predict the final word in a sentence we may just need the final output.

## Why the name recurrent

I would like to discuss very briefly why the networks are called recurrent. We have said that the internal memory state at a time is given by

the internal memory state at time is evaluated using the same memory state at time , the one at time with the value at time and so on. This is at the origin of the name recurrent.

## Learning to count

To give you an idea of the power of such networks I would like to give you a very basic example of something RNN are very good at, and standard fully connected networks, as the one we saw in the previous chapters, are really bad at. Let's try to teach a network to count. The problem we want to solve is the following: given a certain vector, that we will take being made of 15 elements, containing just 0 and 1 we want to build a neural network that is able to count the amount of 1s we have in the vector. This is a difficult problem for a standard network, but why? To understand intuitively why let's consider the problem we have analyzed of distinguishing the one and two digits in the MNIST dataset. If you remember that discussion I did when discussing metric analysis, you will remember that the learning happens because the ones and the twos have black pixels in fundamentally different positions. A digit one will always differ in (at least in the MNIST dataset) the same way from the digit two, so the network will identify those differences and as soon as they are detected a clear identification can be made. In our case this is no more possible. Consider for example a simpler case of a vector with just 5 elements. Consider the case when a one appears exactly one time. We have 5 possible cases: [1,0,0,0,0], [0,1,0,0,0], [0,0,1,0,0], [0,0,0,1,0] and [0,0,0,0,1]. There is no discernable pattern to be detected here. There is no easy weight configuration that could cover those cases at the same time. In case of an image this problem is similar to the problem of detecting the position of a black square in a white image. We can build a network in tensorflow and check how good such networks are. Due to the introductory nature of the chapter I will not spend time in dealing with hyperparameter discussion, metric analysis and so on. I will simply give you a basic network that can count.

Let's start with creating our vectors. We will create vectors that we will split in training and dev sets.

import numpy as np

import tensorflow as tf

from random import shuffle

now let's create our list of vectors. The code is a slightly more complicated, and we will look at it in a bit more details.

nn = 15

ll = 2\*\*15

train\_input = ['{0:015b}'.format(i) for i in range(ll)]

shuffle(train\_input)

train\_input = [map(int,i) for i in train\_input]

temp = []

for i in train\_input:

temp\_list = []

for j in i:

temp\_list.append([j])

temp.append(np.array(temp\_list))

train\_input = temp

We want to have all possible combinations of 1 and 0 in vectors of 15 elements. So, an easy way to do that is take all numbers up to in binary format. To understand why, let's suppose we want to do this with only 4 elements: we want all possible combinations of four 0 and 1. Consider all number up to in binary, that you can get with this code

['{0:04b}'.format(i) for i in range(2\*\*4)]

The code simply format all numbers that you get with the range(2\*\*4) function from 0 to 2\*\*4 in binary format with {0:04b}, limiting the number of digits to 4. The result is the following

['0000',   
 '0001',   
 '0010',   
 '0011',   
 '0100',   
 '0101',   
 '0110',   
 '0111',   
 '1000',   
 '1001',   
 '1010',   
 '1011',   
 '1100',   
 '1101',   
 '1110',  
 '1111']

As you can easily verify you have all possible combinations in the list. You have all possible combinations of the one appearing one times ([0001], [0010], [0100] and [1000]), of the ones appearing two times, and so on. For our example we will simply do it with digits, that means we will do it with numbers up to . The rest of the code above is there to simply transform a string like '0100' in a list [0,1,0,0] and then concatenate all the lists with all the possible combinations. If you check the dimension of the output array you will notice that you get . Each observation is an array of dimensions . Then we prepare the target variable, a one-hot encoded version of the counts. That means that if we have an input with four 1s in the vector our target vector will look like [0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0]. As expected the train\_output array will have the dimensions (32768, 16). Now let's split our set in a train and a dev set, as we have done now several times. We will do it here in a dumb way

train\_obs = ll-2000

dev\_input = train\_input\_[train\_obs:]

dev\_output = train\_output[train\_obs:]

train\_input = train\_input[:train\_obs]

train\_output = train\_output[:train\_obs]

remember that this will work since we have shuffled the vectors at the beginning, so we should have a random distribution of cases. We will use 2000 cases for the dev set and the rest (roughly 30000) for the training. The train\_input will have dimensions (30768, 15, 1) and the dev\_input will have dimensions (2000, 16).

Now you can build a network with this code, and you should be able to understand almost all of it by now

tf.reset\_default\_graph()

data = tf.placeholder(tf.float32, [None, nn,1])

target = tf.placeholder(tf.float32, [None, (nn+1)])

num\_hidden\_el = 24

RNN\_cell = tf.nn.rnn\_cell.LSTMCell(num\_hidden\_el, state\_is\_tuple=True)

val, state = tf.nn.dynamic\_rnn(RNN\_cell, data, dtype=tf.float32)

val = tf.transpose(val, [1, 0, 2])

last = tf.gather(val, int(val.get\_shape()[0]) - 1)

W = tf.Variable(tf.truncated\_normal([num\_hidden, int(target.get\_shape()[1])]))

b = tf.Variable(tf.constant(0.1, shape=[target.get\_shape()[1]]))

prediction = tf.nn.softmax(tf.matmul(last, W) + b)

cross\_entropy = -tf.reduce\_sum(target \* tf.log(tf.clip\_by\_value(prediction,1e-10,1.0)))

optimizer = tf.train.AdamOptimizer()

minimize = optimizer.minimize(cross\_entropy)

errors = tf.not\_equal(tf.argmax(target, 1), tf.argmax(prediction, 1))

error = tf.reduce\_mean(tf.cast(errors, tf.float32))

then let's train the network

init\_op = tf.global\_variables\_initializer()

sess = tf.Session()

sess.run(init\_op)

mb\_size = 1000

no\_of\_batches = int(len(train\_input)/mb\_size)

epoch = 50

for i in range(epoch):

ptr = 0

for j in range(no\_of\_batches):

train, output = train\_input[ptr:ptr+mb\_size], train\_output[ptr:ptr+mb\_size]

ptr+=mb\_size

sess.run(minimize,{data: train, target: output})

incorrect = sess.run(error,{data: test\_input, target: test\_output})

print('Epoch {:2d} error {:3.1f}%'.format(i + 1, 100 \* incorrect))

The new part that you probably will not recognize is the following piece of code

num\_hidden\_el = 24

RNN\_cell = tf.nn.rnn\_cell.LSTMCell(num\_hidden\_el,state\_is\_tuple=True)

val, state = tf.nn.dynamic\_rnn(RNN\_cell, data, dtype=tf.float32)

val = tf.transpose(val, [1, 0, 2])

last = tf.gather(val, int(val.get\_shape()[0]) - 1)

For performance reason and to let you realize how efficient RNNs are I am using here a LSTM kind of neuron. This has a special way of calculating the internal state. A discussion would go well beyond the scope of the book. For the moment you should focus on the results and not on the code itself. If you let the code run, you will get the following result

Epoch 0 error 80.1%   
Epoch 10 error 27.5%  
Epoch 20 error 8.2%  
Epoch 30 error 3.8%  
Epoch 40 error 3.1%  
Epoch 50 error 2.0%

After just 50 epochs the network is right in 98% of the cases. Just let it run for more epochs and you can reach incredible precision. After 100 epochs you can reach an error of 0.5%. An instructive exercise is trying to train a fully connected network (as the ones we have discussed so far) to count. You will see how this is not possible.

You should now have a basic understanding of how CNN and RNN networks works, and on what principles they work on. The research on those networks is immense, since they are really flexible, but the discussion in the previous sections should have given you enough information to understand how those architectures work.

1. Cat image source: https://www.shutterstock.com/ [↑](#footnote-ref-1)