Convolutional Neural Networks

6

6.1 A Third Visit to Logistic Regression

In this chapter, we explore convolutional neural networks, which were first invented by Yann LeCun and others in 1998 [1]. The idea which LeCun and his team implemented was older, and built up on the ideas of David H. Hubel and Torsten Weisel presented in their 1968 seminal paper [2] which won them the 1981 Nobel prize in Physiology and Medicine. They explored the animal visual cortex and found connections between activities in a small but well-defined area of the brain and activities in small regions of the visual field. In some cases, it was even possible to pinpoint exact neurons that were in charge of a part of the visual field. This led them to the discovery of the *receptive field*, which is a concept used to describe the link between parts of the visual fields and individual neurons which process the information.

The idea of a receptive field completes the third and final component we need to build convolutional neural networks. But what were the other two part we have? The first was a technical detail: flattening images (2D arrays) to vectors. Even though most modern implementations deal readily with arrays, under the hood they are often flattened to vectors. We adopt this approach in our explanation since it has less handwaiving, and enables the reader to grasp some technical details along the way. You can see an illustration of flattening a 3 by 3 image in the top of Fig. 6.1. The second component is the one that will take the image vector and give it to a single workhorse neuron which will be in charge of processing. Can you figure out what can we use?

If you said 'logistic regression', you were right! We will however be using a different activation function, but the structure will be the same. A convolutional neural network is a neural network that has one or more convolutional layers. This is not a hard definition, but a quick and simple one. There will be architectures using

S. Skansi, *Introduction to Deep Learning*, Undergraduate Topics
in Computer Science, https://doi.org/10.1007/978-3-319-73004-2_6

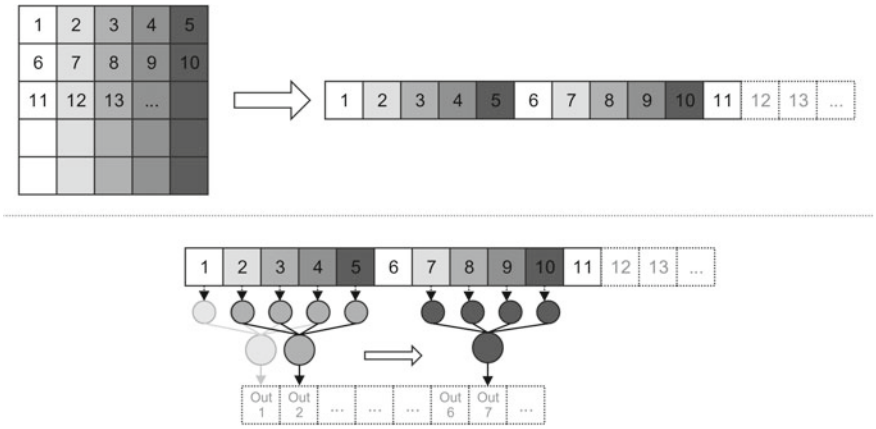

Fig. 6.1 Building a 1D convolutional layer with a logistic regression

convolutional layers which will not be called 'convolutional neural networks'.[1] So now we have to describe what a convolutional layer is.

A convolutional layer takes an image[2] and a small logistic regression with e.g. input size 4 (these sizes are usually 4 or 9, sometimes 16) and passes the logistic regression over the whole image. This means that the first input consists of components 1–9 of the flattened vector, the second input are the components 2–10, the third are components 3–11, and so on. You can see an overview of the process in the bottom of Fig. 6.1. This process creates an output vector which is smaller than the overall input vector, since we start at component 1, but take four components, and produce a single output. The end result is that if we were to move along a 10-dimensional vector with the logistic regression (this logistic regression is called *local receptive field* in convolutional neural networks), we would produce a 7-dimensional output vector (see the bottom of Fig. 6.1). This type of convolutional layer is called a *1D convolutional layer* or a *temporal convolutional layer*. It does not have to use a time series (it can use any data, since you can flatten out any data), but the name is here to distinguish it from a classical 2D convolutional layer.

We can take also a different approach and say we want the output dimension to be same as the input, but then our 4-dimensional local receptive field would have to start at input at 'cells' $-1, 0, 1, 2$ and then continue to $0, 1, 2, 3$, and so on, finishing at $9, 10, 11$ (you can draw it yourself to see why we do not need to go to 12). Putting

[1] Yann LeCun once told in an interview that he prefers the name 'convolutional network' rather than 'convolutional neural network'.

[2] An image in this sense is any 2D array with values between 0 and 255. In Fig. 6.1 we have numbered the positions, and you may think of them as 'cell numbers', in the sense that they will contain some value, but the number on the image denotes only their order. In addition, note that if we have e.g. 100 by 100 RGB images, each image would be a 3D array (tensor) with dimensions (100, 100, 3). The last dimension of the array would hold the three channels, red, green and blue.

in $-1, 0$ and 11 components to get the output vector to have the same size as the input vector is called *padding*. The additional components usually get values 0, but it sometimes makes sense to take either the values of the first and last components of the image or the average of all values. The important thing when padding is to think how not to trick the convolutional layer in learning regularities of the padding. Padding (and some other concepts we discussed) will become much more intuitive when we switch from flattened vectors to non-flattened images. But before we continue, one final comment. We moved the local receptive field one component at a time, but we could move it by two or more. We could even experiment with dynamically changing by how much we move, by moving quicker around the ends and slower towards the centre of the vector. The parameter which says by how many components we move the receptive field between taking inputs is called the *stride* of the convolutional layer.

Let us review the situation in 2D, as if we did not flatten the image into a vector. This is the classical setting for convolutional layers, and such layers are called *2D convolutional layers* or *planar convolutional layers*. If we were to use 3D, we would call it *spatial*, and for 4D or more *hyperspatial*. In the literature is common to refer to the 2D convolutional layer as 'spatial', but this makes one's spider sense tingle.

The logistic regression (local perceptive field) inputs now should be also 2D, and this is the reason why we most often use 4, 9 and 16, since they are squares of 2 by 2, 3 by 3 and 4 by 4 respectively. The stride now represents a move of this square on the image, staring from left, going to the right and after it is finished, one row down, move all the way to the left without scanning, and start scanning from left to right (you can see the steps of this process on the top part of Fig. 6.2). One thing that becomes obvious is that now we will get less outputs. If we use a 3 by 3 local

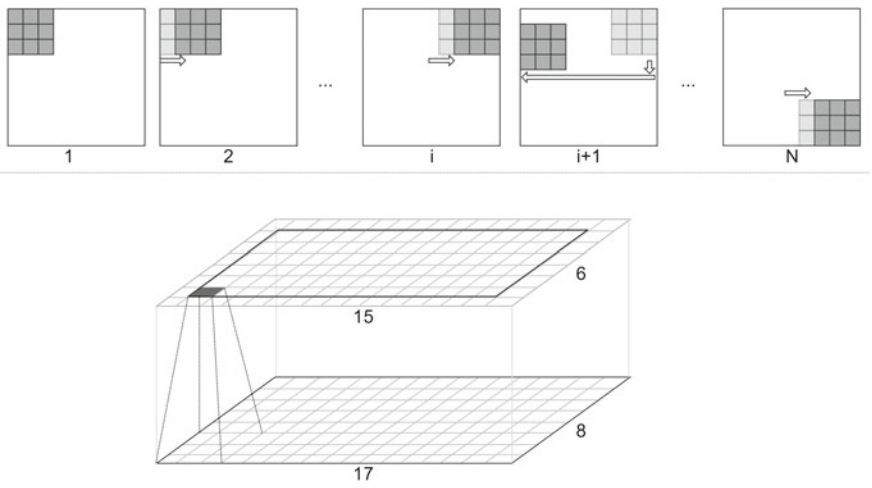

Fig. 6.2 2D Convolutional layer

receptive field to scan a 10 by 10 image, as the output from the local receptive field we will get an 8 by 8 array (see bottom part of Fig. 6.2). This completes a convolutional layer.

A convolutional neural network has multiple layers. Imagine a convolutional neural network consisting of three convolutional layers and one fully connected layer. Suppose it will be processing an image of size 10 and that all three layers have a local receptive field of 3 by 3. Its task is to decide whether a picture has a car in it or not. Let us see how the network works.

The first layer takes a 10 by 10 image, produces an output (it has randomly initialized weights and bias) of size 8 by 8, which is then given to the second convolutional layer (which has its own local receptive field with randomly initialized weights and biases but we have decided to have it also 3 by 3), which produces an output of size 6 by 6, and this is given to the third layer (which has a third local receptive field). This third convolutional layer produces a 4 by 4 image. We then flatten it to a 16-dimensional vector and feed it to a standard fully-connected layer which has one output neuron and uses a logistic function as its nonlinearity. This is actually another logistic regression in disguise, but it could have had more than one output neuron, and then it would not be a proper logistic regression, so we call it a *fully-connected layer of size 1*. The input layer size is not specified and it is assumed to be equal to the output of the previous layer. Then, since it uses the logistic function, it produces an output between 0 and 1 and compares its output to the image label. The error is calculated and backpropagated, and this is repeated for every image in the dataset which completes the training of the network.

Training a convolutional layer means training the local receptive fields of the layers (and weights and biases of fully-connected layers). It has a single bias, and small number of weights (equal to the number of units in the local receptive field). In this respect, it is just like a small logistic regression, and that is what makes convolutional networks quick to train–they have only a small number of parameters to learn. The main structural difference between a logistic regression and a local receptive field is that in a local receptive field we can use any activation function and in logistic regression we are supposed to use the logistic function (if we want to call it 'logistic regression'). The activation function which is most often used is the *rectified linear unit* or ReLU. A ReLU of x is simply the maximal value of 0 and x, meaning that it will return a 0 if the input is negative or the raw input otherwise. In symbols:

$$\rho(x) = max(x, 0) \tag{6.1}$$

Padding in 2D is simply a 'frame' of n pixels around the image. Note that it does not make much sense to use a padding of say 3 (pixels) if we use only a 3 by 3 local receptive field, since it will only go one pixel over the image border.

6.2 Feature Maps and Pooling

Now that we know how a convolutional neural network works, we can use a trick. Recall that a convolutional layer scans a 10 by 10 image with an e.g. 3 by 3 local receptive field (9 weights, 1 bias) and builds a new 8 by 8 'image' as the output. Imagine also that the image has three channels for colours. How would you process an image with three channels? A natural answer is to run over the same receptive field (which has trainable but randomly initialized weights and bias). This is a good strategy. But what if we invert it, and instead of using one local receptive field over three channels, we want to use five local receptive fields over one channel? Remember that a local receptive field is defined by its size and by its weights and bias. The idea here is to keep the same size but initialize the other receptive fields with different weights and biases.

This means that when they scan a 10 by 10 3-channel image, they will construct 15 8 by 8 output images. These images are called *feature maps*. It is like having an 8 by 8 image with 15 channels. This is very useful since only one feature map which learns a good representation (e.g. eyes and noses on pictures of dogs) will boost considerably the overall accuracy of the network[3] (suppose that the task for the whole network was to classify images of dogs and various non-dog objects (i.e. detecting a dog in an image)).

One of the main ideas here is that a 10 by 10 3-channel image turns into an 8 by 8 15-channel image. The input image was transformed into a smaller but deeper object, and this will happen in every convolutional layer.[4] Getting the image smaller, means packing the information in a more compact (but deeper) representation. In our quest for compactness, we may add a new layer after or before a convolutional layer. This new layer is called a *max-pooling* layer. The max-pooling layer takes a pool size as a hyperparameter, usually 2 by 2. It then processes its input image in the following way: divide the image in 2 by 2 areas (like a grid), and take from each four-pixel pool the pixel with the maximal value. Compose these pixels into a new image, with the same order as the original image. A 2 by 2 max-pooling layer produces an image that is half the size of the original image (it does not increase the channel number). Of course, instead of the maximum, a different pixel selection or creation can be devised, such as the average of the four pixels, the minimum, and so on.

The idea behind max-pooling is that important information in a picture is seldom contained in adjacent pixels (this accounts for the 'pick-one-out-of-four' part), and it is often contained in darker pixels (this accounts for using the max). You may notice right away that this is a very strong assumption which may not be generally valid. It must be said that max-pooling is rarely used on images themselves (although it can be used), but rather on learned feature maps, which are images but they are very

[3] Here you might notice how important is weight initialization. We do have some techniques that are better than random initialization, but to find a good weight initialization strategy is an important open research problem.

[4] If using padding we will keep the same size, but still expand the depth. Padding is useful when there is possibly important information on the edges of the image.

peculiar images. You can try to modify the code in the section below to print out feature maps which come out of a convolutional layer.[5] You can think of max-pooling in terms of decreasing the screen resolution. In general, if you recognize a dog on a 1200 by 1600 image, you will probably recognize him on a grainer 600 by 800 image.

Usually a convolutional neural network is composed of a convolutional layer followed by a max-pooling layer, followed by a convolutional layer, and so on. As the image goes through the network, after a number of layers, we get a small image with a lot of channels. Then we can flatten this to a vector and use a simple logistic regression at the end to extract which parts are relevant for our classification problem. The logistic regression (this time with the logistic function) will pick out which parts of the representation will be used for classification and create a result which will be compared with the target and then the error will be backpropagated. This forms a complete convolutional neural network. A simple but fully functional convolutional network with four layers is shown in Fig. 6.3.

Why are convolutional neural networks easier to train? The answer is in the number of parameters used. A five-layer deep fully connected neural network for MNIST has a lot of weights,[6] through which we need to backpropagate. A five-layer convolutional network (containing only convolutional layers) with all receptive fields of 3 by 3 has 45 weight and 5 biases. Notice that this configuration can be used for arbitrarily large images: we do not have to expand the input layer (which is a convolutional layer in our case), but we will need more convolutional layers then to shrink the image. Even if we add feature maps, the training of each feature map is independent of the other, i.e. we can train it in parallel. This makes the process not only computationally fast, but we can also split it across many processors. By contrast, to backpropagate errors

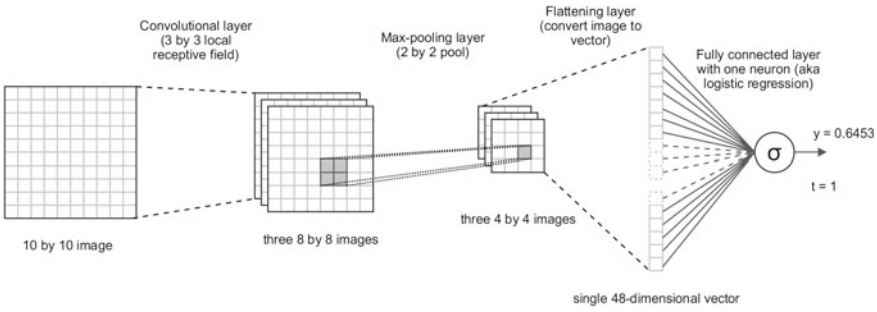

Fig. 6.3 A convolutional neural network with a convolutional layer, a max-pooling layer, a flattening layer and a fully connected layer with one neuron

[5] You have everything you need in this book to get the array (tensor) with the feature maps, and even to squash it to 2D, but you might have to search the Internet to find out how to visualize the tensor as an image. Consider it a good (but advanced) Python exercise.

[6] If it has 100 neurons per layer, with only one output neuron, that makes the total of $784 \cdot 100 + 100 \cdot 100 + 100 \cdot 100 + 100 \cdot 1 = 98500$ parameters, and that is without the biases!.

through a regular feed-forward fully connected network is highly sequential, since we need to have the derivatives of the outer layers to compute the derivatives of the inner layers.

6.3 A Complete Convolutional Network

We now show a complete convolutional neural network in Python. We are using the library Keras, which gives us the ability to build neural networks from components, without having to worry too much about dimensionality. All the code here should be placed in one Python file and then executed in the terminal or command prompt. There are other ways to run Python code, and feel free to experiment with them—nothing will break. The first part of the code which should be placed in the file handles the imports from Keras and Numpy:

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, MaxPooling2D
from keras.utils import np_utils
from keras.datasets import mnist
(train_samples, train_labels), (test_samples, test_labels) = mnist.load_data()
```

You might notice the we are importing MNIST from the Keras repository. The last line of this code loads training samples, training labels, test samples and test labels in four different variables. Most of the code in this Python file will actually be used for formatting (or preprocessing) MNIST data to meed the demands which it must fulfill to be fed into a convolutional neural network. The next part of the code processes the MNIST images:

```
train_samples = train_samples.reshape(train_samples.shape [0], 28, 28, 1)
test_samples = test_samples.reshape(test_samples.shape [0], 28, 28, 1)
train_samples = train_samples.astype('float32')
test_samples = test_samples.astype('float32')
train_samples = train_samples/255
test_samples = test_samples/255
```

First notice that the code is actually duplicated: all operations are performed on both the training set and the testing set, and we will comment only one (we will talk about the training set), the other one functions in the same manner. The first line of this block of code reshapes the array which holds MNIST. The result of this reshaping is a (60000, 28, 28, 1)-dimensional array.[7] The first dimension is simply the number of samples, the second and the third are here to represent the 28

[7] Which is, mathematically speaking, a tensor.

by 28 dimension of the images, and the last one is the channel. It could be RGB, but MNIST is in greyscale, so this might seem redundant, but the whole point of reshaping the array (the initial dimension was (60000, 28, 28)) was actually to add the final dimension with 1 component in it. The reason behind this is that as we progress through convolutional layers, feature maps will be added in this direction, so we need to prepare the tensor to be able to accept it. The third row declares the entries in the array to be of type `float32`. This simply means that they are to be treated as decimal numbers. Python would to his automatically, but Numpy, which speeds up computation drastically, needs type declarations, so we have to put this line in. The fifth line normalizes array entries from a range of 0 to 255 to a range between 0 and 1 (to be interpreted as the percentage of grey in a pixel). That takes care of the samples, now we must preprocess the labels (digits from 0 to 9) with one hot encoding. We do this with the following code:

```
c_train_labels = np_utils.to_categorical(train_labels, 10)
c_test_labels = np_utils.to_categorical(test_labels, 10)
```

With that we are finished preprocessing the data and we may continue to build the actual convolutional neural network. The following code specifies the layers:

```
convnet = Sequential()
convnet.add(Convolution2D(32, 4, 4, activation='relu', input_shape=(28,28,1)))
convnet.add(MaxPooling2D(pool_size=(2,2)))
convnet.add(Convolution2D(32, 3, 3, activation='relu'))
convnet.add(MaxPooling2D(pool_size=(2,2)))
convnet.add(Dropout(0.3))
convnet.add(Flatten())
convnet.add(Dense(10, activation='softmax'))
```

The first line of this block of code creates a new blank model, and the rest of the lines here fill the network specification. The second line adds the first layer, in this case it is a convolutional layer, which has to produce 32 feature maps, has ReLU as the activation function and has a 4 by 4 receptive field. For the first layer, we also have to specify the input dimensions for each training sample that we will be giving it. Notice that Keras takes the first dimension of an array to represent individual training samples and chops up (parses) the dataset along it, so we do not need to worry about giving a (65600, 28, 28, 1) tensor instead of a (60000, 28, 28, 1) after we have specified that it takes `input_shape=(28, 28, 1)`, but the code will crash if we give it a (60000, 29, 29, 1) or even a (60000, 28, 28) dataset. The third row defines a max pooling layer with a pool size of 2 by 2. The next line specifies a third layer, which is a convolutional layer, this time with a receptive field of 3 by 3. Here we do not have to specify the input dimensions, Keras will do that for us. Following that we have another max pooling layer, also with a pool size of 2 by 2.

After this we have a dropout 'layer'. This is not a real layer, but only a modification of the connections between the previous and the following layer. The connections are modified to include a dropout rate of 0.3 for all connections. The next line flattens the

tensor. This is a generalized version of the process which we described for translating fixed-size matrices into a vector,[8] only here it is generalized for arbitrary tensors.

The flattened vector is then fed into the final layer (the final line of code in this block) which is a standard fully-connected feed-forward layer,[9] accepting as many inputs as there are components in the flattened vector, and outputting 10 values (10 output neurons), where each of them will represent one digit and it will output the respective probability. Which of them represents which digit is actually defined only by the order we had when we did one-hot encoding of the labels.

The softmax activation function used in the final layer is a version of the logistic function for more than two classes, but we will describe it in the later chapters, for now just think of it as a logistic function for many classes (we have one class for each label 0–9). Now we have a model specified, and we must compile it. Compiling a model means that Keras can now deduce and fill in all the necessary details we did not specify such as the input size for the second convolutional layer, or the dimensionality of the flattened vector. The next line of code compiles the model:

```
convnet.compile(loss='mean_squared_error', optimizer='sgd', metrics=['accuracy'])
```

Here we can see that we have specified the training method to be `'sgd'` which is stochastic gradient descent, with MSE as the error function. We have also asked the Keras to calculate the accuracy when training. The next line of code trains the compiled model:

```
convnet.fit(train_samples, c_train_labels, batch_size=32, nb_epoch=20, verbose=1)
```

This line of code trains the model using `train_samples` as training samples and `c_train_labels` as training labels. It also uses a batch size of 32 and trains for 20 epochs. The 'verbose' flag is set to 1 which means that it will print out details of training. And now we continue to the final part of the code which prints the accuracy and makes predictions from what it has learned for a new set of data:

```
metrics = convnet.evaluate(test_samples, c_test_labels, verbose=1)
print()
print("%s: %.2f%%" % (convnet.metrics_names[1], metrics[1]*100))
predictions = convnet.predict(test_samples)
```

The last line is important. Here we have put `test_samples`, but if you want to use it for predictions, you should put some fresh samples here, bearing in mind that they have to have *exactly* the same dimensions as `test_samples` asides from the first dimension, which holds individual training samples and along which Keras parses the dataset. The variable `predictions` will have exactly the same dimensionality as `c_test_labels` asides from the first dimension, but the first dimension of `test_samples` and `c_test_labels` will be the same (since they are predicted labels for *that* set of samples). You can add a line to the end saying `print(predictions)` to see the actual predictions, or

[8] Remember how we can convert a 28 by 28 matrix into a 784-dimensional vector.
[9] Keras calls them 'Dense'.

`print(predictions.shape)` to see the dimensionality of the array stored in `predictions`. These 29 lines of code (or 30 if you added one of the last ones) form a fully functional convolutional network.

6.4 Using a Convolutional Network to Classify Text

Even though the standard setting for a convolutional neural network is pattern recognition in images, convolutional neural networks can also be used to classify text. A standard approach is to use characters instead of words as primitives, and then try to map a representation of text on a character level to a higher level idea like positive or negative sentiment. This is very interesting since it allows to do a considerable amount of language processing from raw text, without any fancy feature engineering or a knowledge-heavy logical system—just learning from the letters used. In this section, we explore the now classical paper by Xiang Zhang, Junbo Zhao and Yann LeCun titled *Character-level Convolutional Networks for Text Classification* [3]. The paper itself is considerably more rich than what we present here, but we will be showing the bare bones of the approach that the authors used. We do this to help the reader to understand how to read research papers, and we strongly encourage the reader to download a copy of the paper from `arxiv.org/abs/1509.01626` and compare the text with what we write here. There will be a couple more sections like this, all with the same aim, to help the student understand papers we consider to be especially interesting. Of course, there are many more seminal and interesting papers, but we had to pick only a couple, but we encourage the reader to find more and work through them by herself.

The paper *Character-level Convolutional Networks for Text Classification* uses convolutional neural networks to classify text. One of the tasks the authors explore is the Amazon Review Sentiment Analysis. This is the most widely used sentiment analysis dataset, and it is available from a variety of sources, perhaps the best one being https://www.kaggle.com/bittlingmayer/amazonreviews. You will need a bit of formatting to get it to run, and getting this to work will be a great data wrangling exercise. Every line in these files has a review together with a label at the beginning. Two samples from the raw file are (you can conclude which label is which, there are only these two):

```
__label__1 Waste of money!

__label__2 Great book for travelling Europe:
```

The authors use a couple of architectures, and we focus on the larger one. The network uses 1D convolutional layers. Note that here we will have an example of a 1D convolutional layer processing a $m \times n$ matrix rather than a vector. This is the same as processing a vector, since the 1D convolutional layer will behave in the same way, except it will take all m rows in a pass instead of a single one as it would if it

were a vector. The 'width' of the local receptive field remains a hyperparameter, as does the stride. The stride here is 1 throughout the paper.

The first layer of the network used in the paper is of size 1024, with a local receptive field (called 'kernel' in the paper) of 7, followed by a pooling layer with a pool of size 3. This all is called 'layer 1' in the paper. The authors consider pooling to be a part of the convolutional layer, which is ok, but Keras treats pooling as a separate layer, so we will re-enumerate the layers here so that the reader can recreate them in Keras. The third and fourth level are the same as the first and second. The fifth, sixth, seventh and eighth layer are the same as the first layer (they are convolutional layers with no pooling), the ninth layer is a max pooling layer with a pool of 3 (i.e. it is like the second layer). The tenth layer is a flattening layer, and the eleventh and twelfth layers are fully-connected layers of size 2048. The final layer's size depends on the number of classes used. For sentiment this is 'positive' and 'negative', so we may use a logistic function with a single output neuron (all other layers use ReLUs). If we were to have more classes, we would use softmax, but we will do this in the later chapters. There are also two dropout layers between the three fully-connected layers and special weight initializations, but we ignore them here.

So now we have explained the task, shown you where to find the dataset with the data and labels, and explored the network architecture. What is left to do is to see how to feed the data to the network, and for this, we need encoding. The encoding is the trickiest part of this paper.[10]

Let us see how the authors encode the text. We have already noted that they use a character based approach, so we have to specify which characters to use, i.e. which we shall leave in the text and which we will remove. The authors substitute all uppercase letters for lower ones, and keep all the 26 letters of the English alphabet as valid characters. In addition, they keep the ten digits and 33 other characters (including brackets, $, #, etc.). They total to 69. They keep also the new line character, often denoted as $\backslash n$. This is the character that the Enter or Return key produces when hit. You do not see it directly, but the computer produces a new line. This means that the vocabulary size is 69, and we shall denote this by M.

The length of the particular review as a string is denoted by L. The review (without the label part) will be one-hot-encoded (aka 1-of-M encoding) using characters, but there is a twist. To make the system behave like human memory, every string is reversed, so `Waste of money!` will become `!yenom fo etsaW`. To see a complete example, imagine we have only allowed a, b, c, d and S as allowed characters,[11] where the S simply represents whitespace, since leaving it as a space would probably cause confusion (and we have used the ␣ for Python code indentation). Suppose the text of the review is 'abbaScadd', and $L_{final} = 7$. First, the reverse it to 'ddacSabba', and then cut it to have a length of 7, to get 'ddacSab'. Then we use one hot encoding to get an M by L_{final} matrix to represent this input sample:

[10] Trivially, every paper will have a 'trickiest part', and it is your job to learn how to decode this part, since it is often the most important part of the paper.

[11] Since the whole alphabet will not fit on a page, but you can easily imagine how it will expand to the normal English alphabet.

a	0	0	1	0	0	1	0
b	0	0	0	0	0	0	1
c	0	0	0	1	0	0	0
d	1	1	0	0	0	0	0
S	0	0	0	0	1	0	0

If on the other hand we had the review 'bad' and $L_{final} = 7$, we would first reverse it to 'dab' and then put it in the left of the M by L_{final} matrix and pad the rest of the columns with zeros:

a	0	1	0	0	0	0	0
b	0	0	1	0	0	0	0
c	0	0	0	0	0	0	0
d	1	0	0	0	0	0	0
S	0	0	0	0	0	0	0

But for a convolutional neural networks, all input matrices must have the same dimension, so we have an L_{final}. All inputs for which $L > L_{final}$ are clipped to L_{final} and all of the inputs for which $L_{final} > L$ are padded by adding enough zeros to the right side to make their length exactly L_{final}. This is why the authors used the reversing, so that we loose only the more remote information at the beginning when clipping, and not the more recent one at the end.

We might ask how to make a Keras-friendly dataset from these? The first task is to view them as a tensor. This just means to collect all of the M by L_{final} matrices and add a third dimension along which they will be 'glued'. This simply means if we have 1000 M by L_{final} matrices, that we will make one M by L_{final} by 1000 tensor. Depending on the implementation you will use, it might make sense to make a 1000 by M by L_{final} tensor. Now initialize this tensor (a 3D Numpy array) with all zeros, and devise a function which will put a 1 where it should be. Try to write Keras code which implements this architecture. As always, if you get stuck, StackOverflow it. If you have never done anything similar before, it might take you even a week[12] to get it to work, even though the end result does not have many lines of code. This is a great exercise in deep learning, so don't skip it.

References

1. Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998)

[12] A couple of hours each day—not a literal week.

2. D.H. Hubel, T.N. Wiesel, Receptive fields and functional architecture of monkey striate cortex. J. Physiol. **195**(1), 215–243 (1968)
3. X. Zhang, J. Zhao, Y. LeCun, Character-level convolutional networks for text classification, in *Advances in Neural Information Processing Systems 28, NIPS* (2015)