Review: dense layers

In ordinary neural networks with dense (also: fully connected) layers, the information contained in a particular sample – and flowing through the network – is represented by 1D NumPy vectors. While such networks are a good choice for a broad range of problems, they also have their downsides:

- The number of learnable parameters (weights + biases) in a single fully-connected layer with N_{in} input variables and N_{units} neurons is $(N_{in} + 1) \cdot N_{units}$, and can easily become huge. If, for example, the input data consists of color images of size 128 × 128, the first hidden layer with 256 neurons would already have over 12,5 million parameters to train – which would both be extremely slow, and lead to serious overfitting.

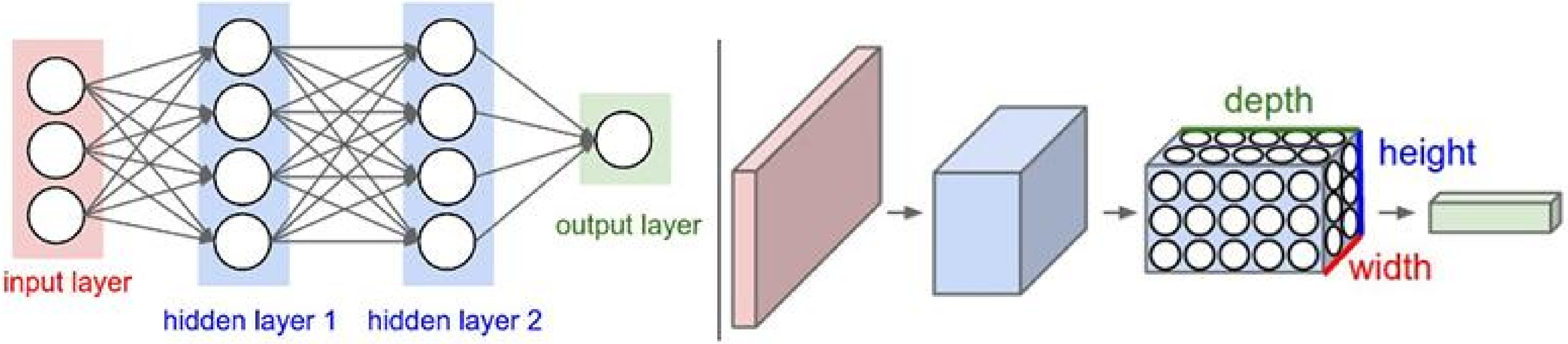
- By construction, a dense layer treats all the input variables on an equal footing. When the input samples are images, this means that e.g. two samples representing the same object slightly displaced or rotated, have to be learned separately (both samples correspond to entirely different sets of pixels) by the network. This makes learning very inefficient.

Both these problems are solved by selecting a different network architecture → CNNs.

Data representations in ConvNets

Convolutional Neural Networks (CNNs, ConvNets) are specially designed for solving problems related to visual recognizion (e.g. image classification), and are used in this field almost without exception.

In ConvNets, the input data is fed to the network in 4D tensors of shape (samples, width, height, color). The information related to any individual sample, therefore, flows through the network in 3D volumes: width, height, and depth. For the input layer, the depth dimension refers to the color channels; in the following layers, the depth corresponds to the number of different units in the layer.



Units in ConvNet layers

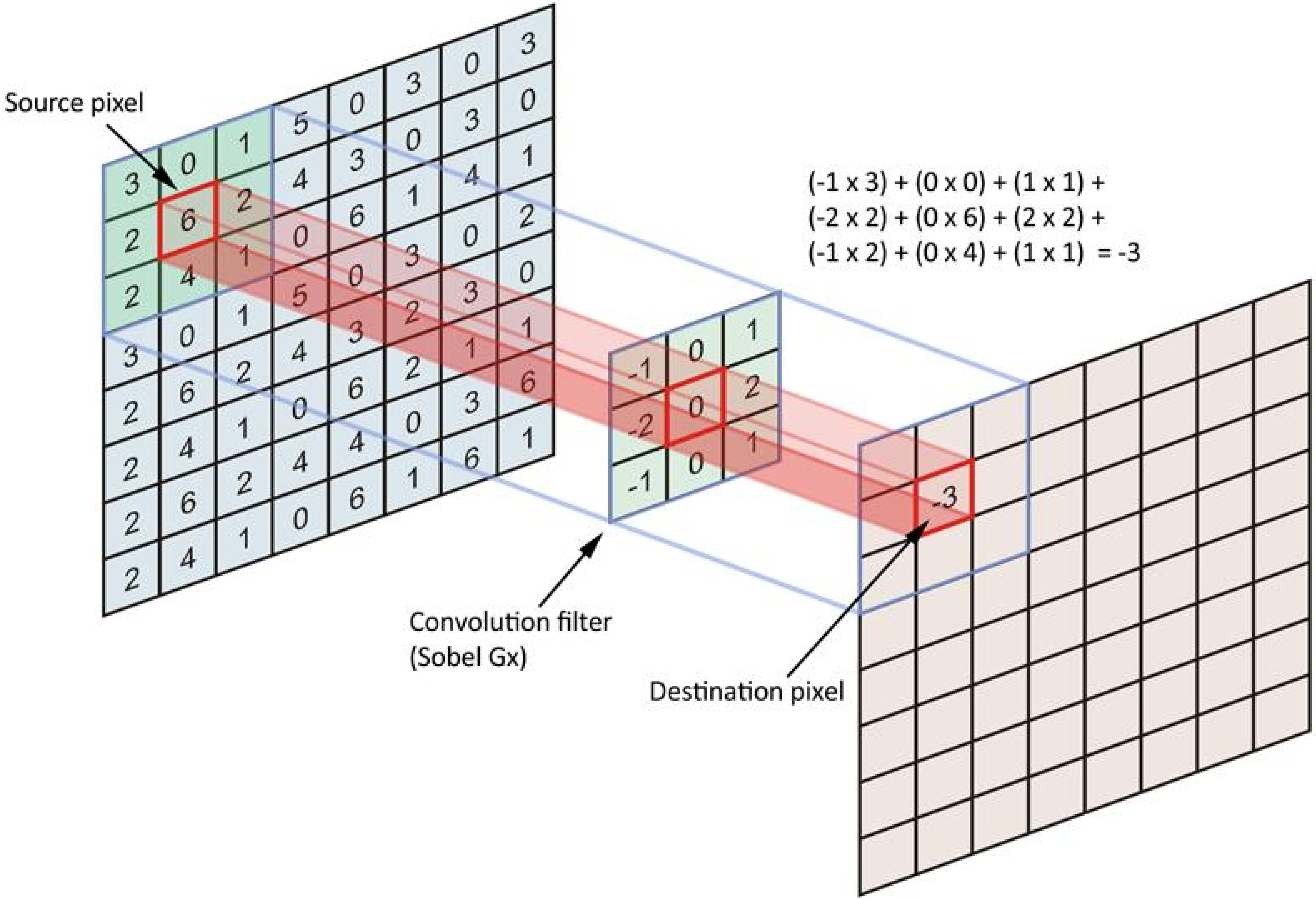
In ConvNets, each neuron only operates on a small number of input variables at a time, corresponding to a small region in the width-height plane – but to full depth. The size of this region, the filter size (or the receptive field size), is very often taken to be 3 × 3 or 5 × 5.

As in all neural networks, the neuron computes a weighted sum of this small set of input variables; with filter size 3 × 3 and 3 color channels, the neurons in the first hidden layer would have $3 \cdot 3 \cdot 3 = 27$ weights (plus one optional bias parameter). The weighted sum is fed into an activation function, which outputs a single number, as usual.

After this, the filter is slided to a new, neighboring position, and the same process repeated with the new variable values corresponding to this 3 × 3 × 3 region, but with the same weights. This parameter sharing keeps the number of trainable weights reasonable.

When the filter has traversed the entire plane, the process results in a new 2D representation of the input data, a feature map, as seen by this particular filter. If the layer contains N_f such filters, we end up with a new 3D representation with depth equal to N_f.

Visualizing convolution



3	0	1	5	0	3	0	3
2	6	2	4	3	0	3	0
2	4	1	0	6	1	4	1
3	0	1	5	0	3	0	2
2	6	2	4	3	2	3	0
2	4	1	0	6	2	1	1
2	6	2	4	4	0	3	6
2	4	1	0	6	1	6	1

-1	0	1
-2	0	2
-1	0	1

-3

(-1 x 3) + (0 x 0) + (1 x 1) +
(-2 x 2) + (0 x 6) + (2 x 2) +
(-1 x 2) + (0 x 4) + (1 x 1) = -3

Note: for clarity, this figure does not show explicitly the depth dimension in the pixel data and the filter.

ConvNet layer hyperparameters

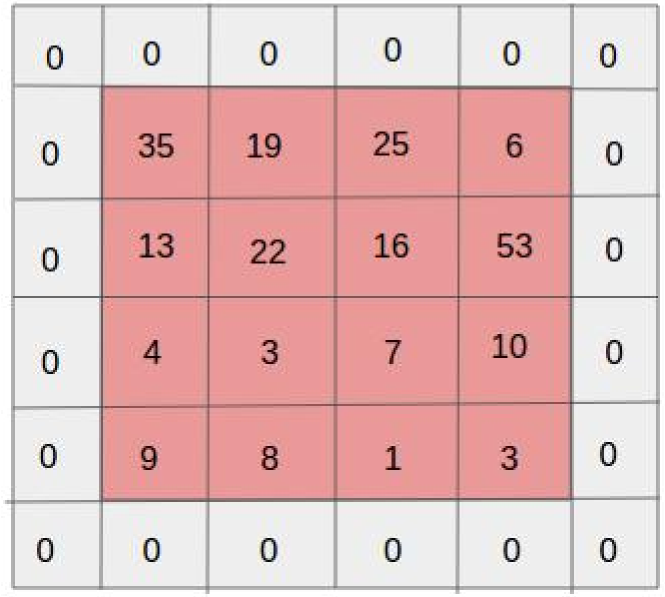
A ConvNet layer has three central hyperparameters: depth, stride and zero padding.

- The depth of the layer is given by the number of different filters (feature maps). Larger depth means a larger number of different learnable features in the image.

- Stride refers to the size of steps used in sliding the filter over the plane during the convolution procedure. Default value 1 means moving one pixel at a time, and is the most common choice. Stride 2 means that the filters move two pixels at a time, reducing the width and height dimensions by a factor of two.

- Due to border effects, the convolution process tends to change the width and height of the data volume: *e.g.* with input images of size 28 × 28, stride 1, and filter size 3 × 3, the number of possible locations for the filter is 26 × 26. In order to have matching values for the width and height of the input and output 3D volumes, the input volume could be padded with zeroes around the border.

Zero padding



0	0	0	0	0	0
0	35	19	25	6	0
0	13	22	16	53	0
0	4	3	7	10	0
0	9	8	1	3	0
0	0	0	0	0	0

Simple example: input size 4 × 4, filter size 3 × 3, stride 1

Without padding: output size 2 × 2

With padding: output size 4 × 4 (same as input size)

In Keras, the default is no padding, padding = ”valid”. Choosing padding = ”same” results in zero padding so that the width and height of input and output volumes match.

Parameter sharing

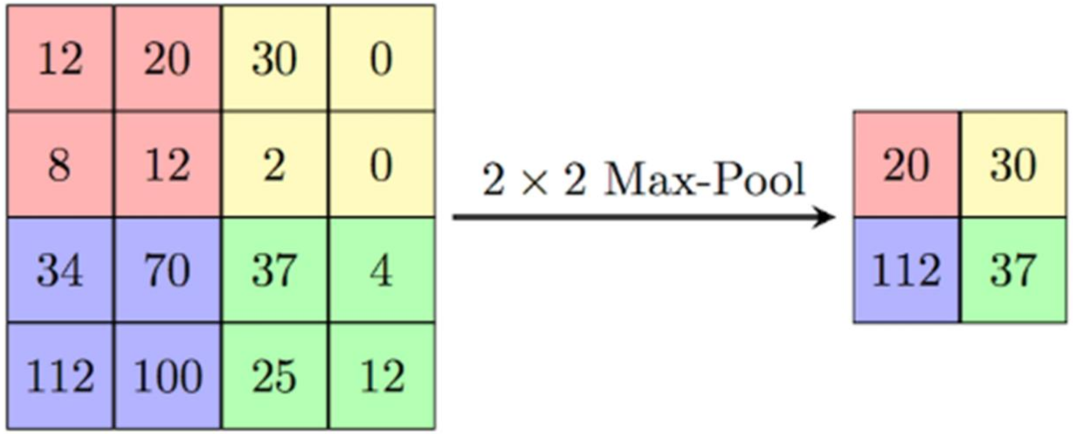
Consider a ConvNet layer with 256 different filters, with filter size 3 × 3. If the input data volume to this layer corresponds to images of size 128 × 128, with 3 color channels, this layer would have altogether $256 \cdot (3 \cdot 3 \cdot 3 + 1) = 7168$ trainable parameters. Note the impressive reduction from the dense-layer value (12,5 million) due to parameter sharing: the same weights are used in different locations. If stride = 1 is used, and no zero padding, the output 3D volume in this example would have size 126 × 126 × 256.

Apart from the savings in computing resources, this procedure makes good sense in working with image data. When the networks learns a certain feature, it learns to recognize this feature at any location within the image.

With a network containing several layers, it is desirable to have successive layers learn features of different hierarchies: from simple small-scale features to complex ones of large scale. This can be achieved by downsampling the data in the width and height dimensions.

Pooling

Convolutional neural networks quite often have pooling layers between different ConvNet layers. Its purpose is to reduce the width and height of the representation, and to reduce the amount of computation that must be performed. A common choice is max pooling with filter size 2 × 2; this means downsampling both width and height by a factor of two by keeping only one of 4 numbers in 2 × 2 regions (the one with the largest numerical value).



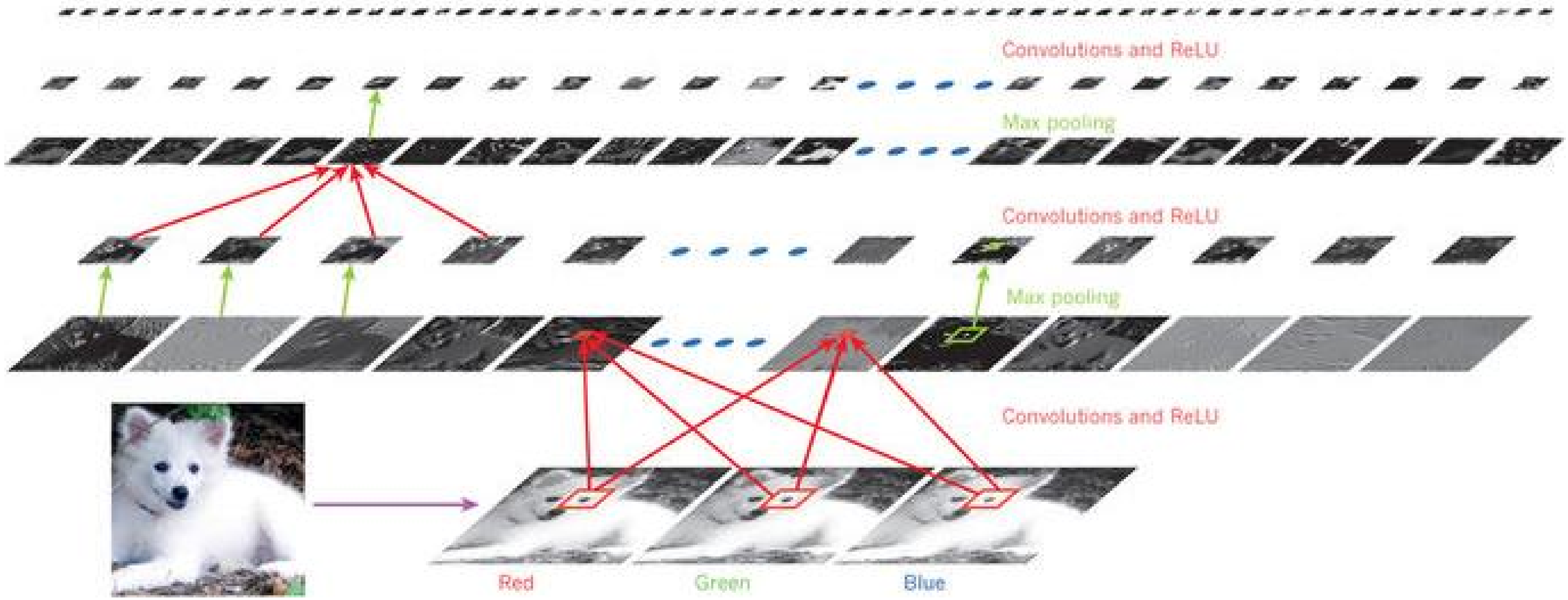
12	20	30	0
8	12	2	0
34	70	37	4
112	100	25	12

20	30
112	37

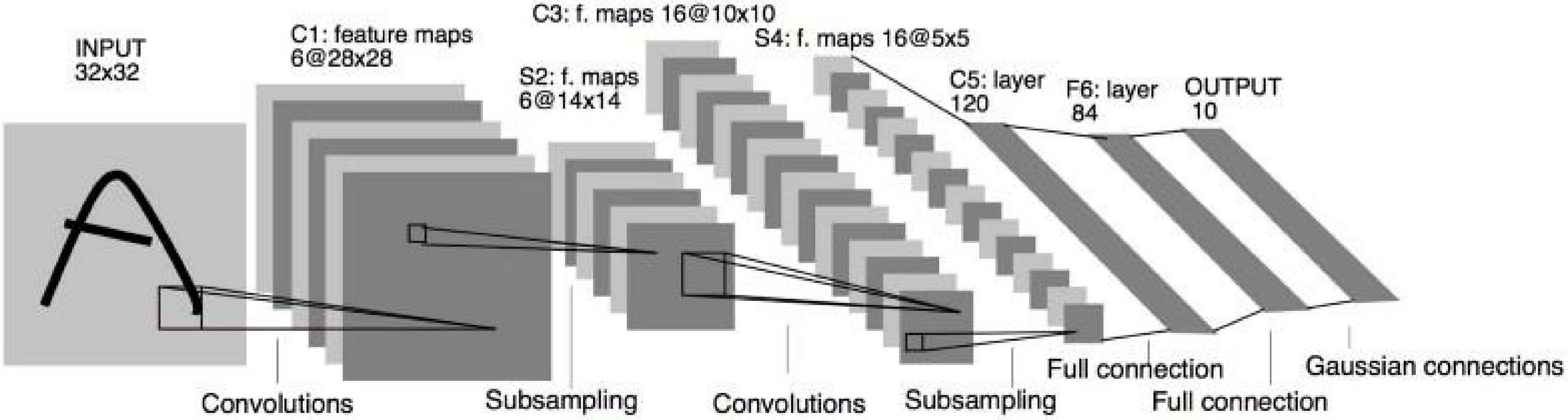
While the pooling layers throw away part of the information, they are useful in practice, because they enable the learning of features of increasing size – and also increasing relevance in view of the problem (e.g. classification) under consideration.

ConvNet architectures

Typical ConvNet architectures contain a stack of successive Conv layers with pooling layers in between. Usually, the network ends with a dense (fully-connected) layer to compute the final output (e.g. the class scores).



Example: LeNet-5 (1994)



Can you find out what was the filter size in C1 and C3?
Or the total number of trainable parameters in C1? C5? C6?