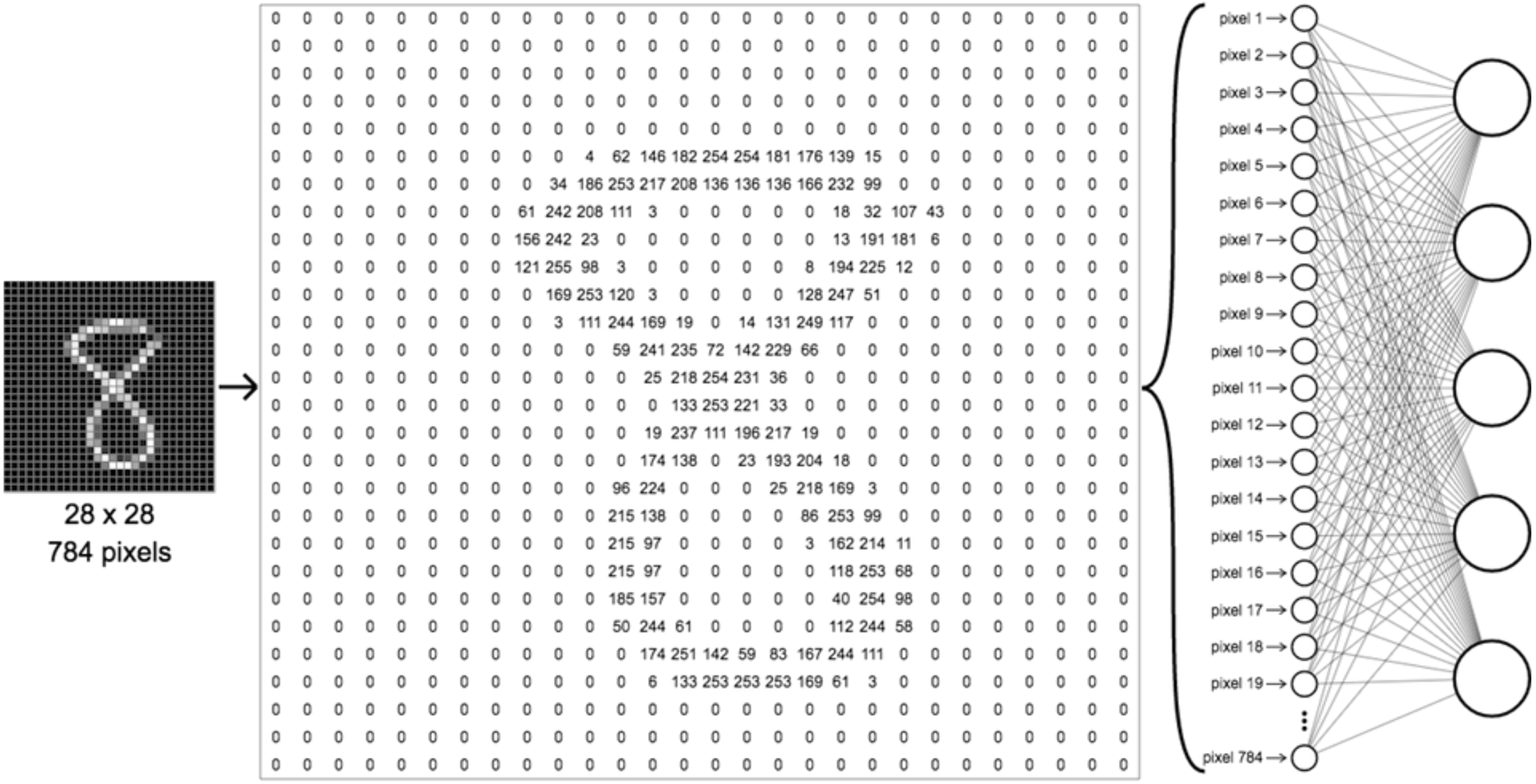
Deep Learning

Convolutional Neural Networks

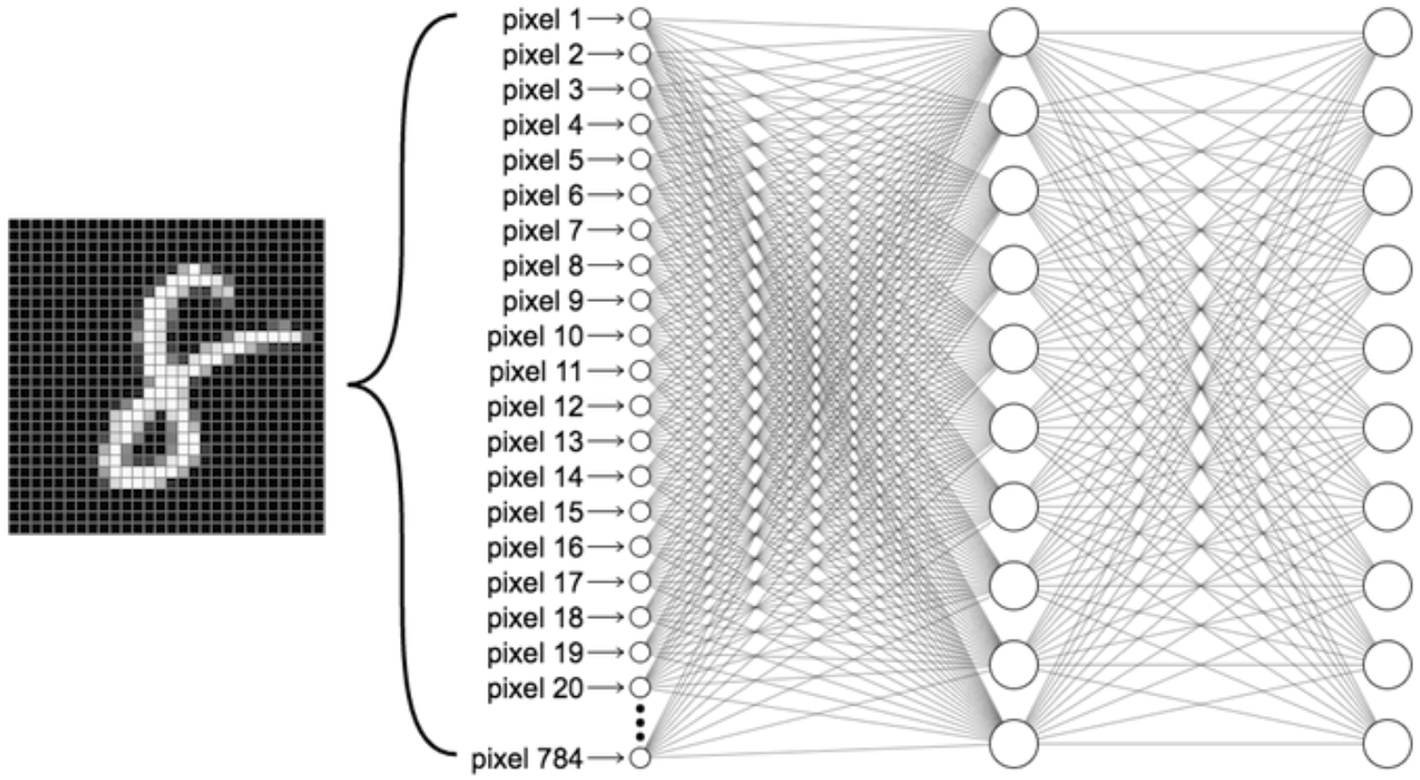
Images and FC neural networks



Flattening operation

https://mc.ai/what-is-deep-learning/

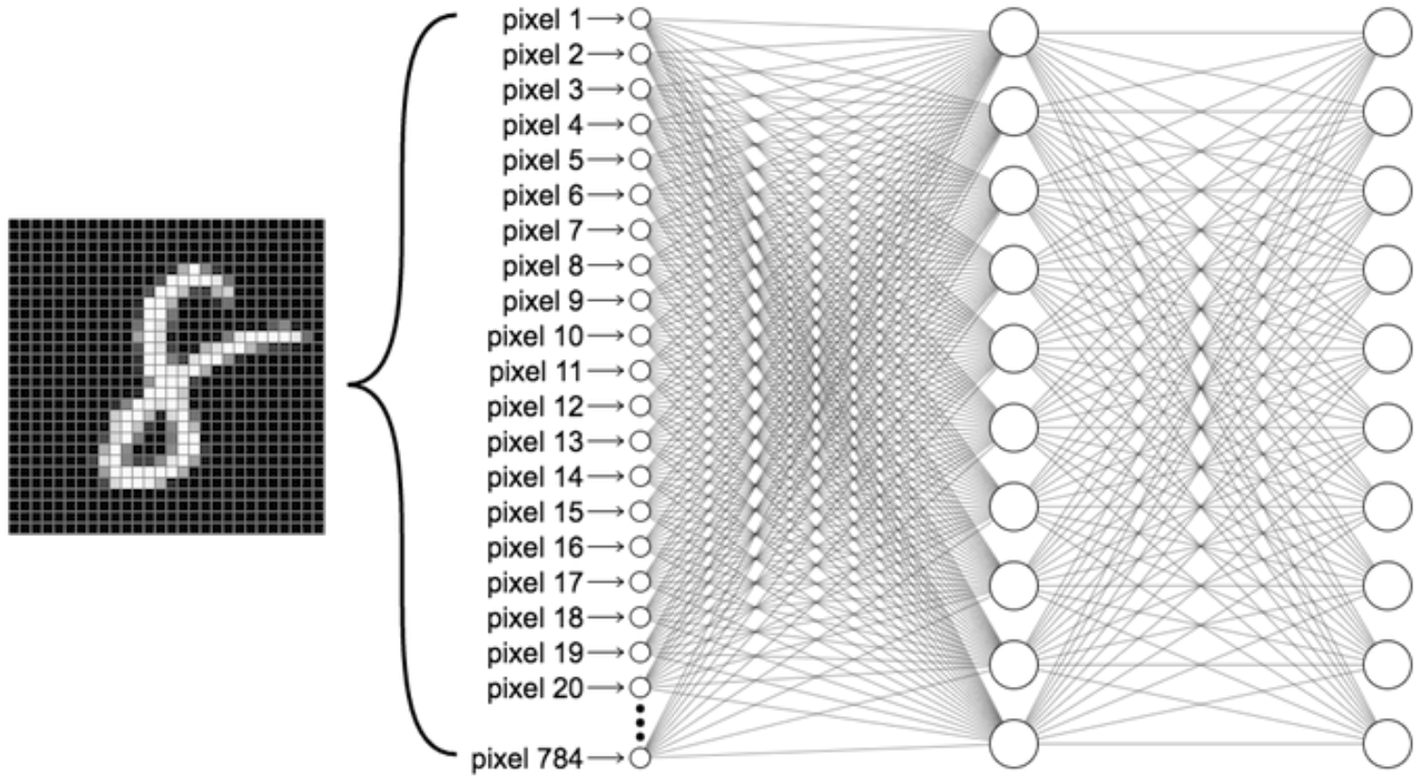
Images and FC neural networks



https://www.google.com/url?sa=i&url=https%3A%2F%2Fml4a.github.io%2Fml4a%2Flooking_inside_neural_nets%2F&psig=AOvVaw0UknUai0JapuIndBrY_BpX&ust=1604588386364000&source=images&cd=vfe&ved=0CAIQjRxqFwoTCOiB-NmT6ewCFQAAAAAdAAAAABAD

Say we have a dataset of one-megapixel photographs we need to classify. This means that each input to the network has one million dimensions.
Even an aggressive reduction to one thousand hidden dimensions would require a fully-connected layer characterized by $10^6 \times 10^3 = 10^9$ parameters.
Moreover, learning a classifier by fitting so many parameters might require collecting an enormous dataset.

Images and FC neural networks



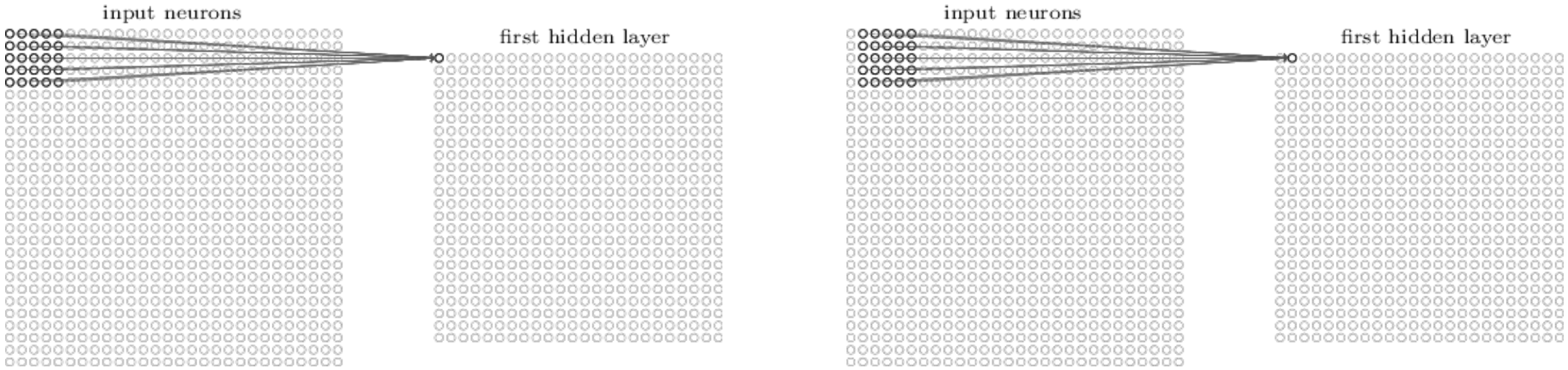
https://www.google.com/url?sa=i&url=https%3A%2F%2Fml4a.github.io%2Fml4a%2Flooking_inside_neural_nets%2F&psig=AOvVaw0UknUai0JapuIndBrY_BpX&ust=1604588386364000&source=images&cd=vfe&ved=0CAIQjRxqFwoTCOiB-NmT6ewCFQAAAAAdAAAAABAD

Convolutional neural networks (CNNs) are one creative way that machine learning has embraced for exploiting some of the known structure in natural images in order to **minimize the number of parameters** and also to build interesting **invariances in the model**.

Minimizing the number of parameters: LRF's

One possible solution for this problem is to consider **local receptive fields** (LRF) (an inspiration that comes from neuroscience).

A local receptive field is a standard MLP with units that have a local view of the image: its input is a submatrix $M \times M$ of the original $N \times N$, where $M << N$.



No flattening operation!

MNIST image size= 28 x 28 pixels = 784 dimensions

A FC layer with 100 units has (784 x 100 + 100) parameters = 78500 parameters

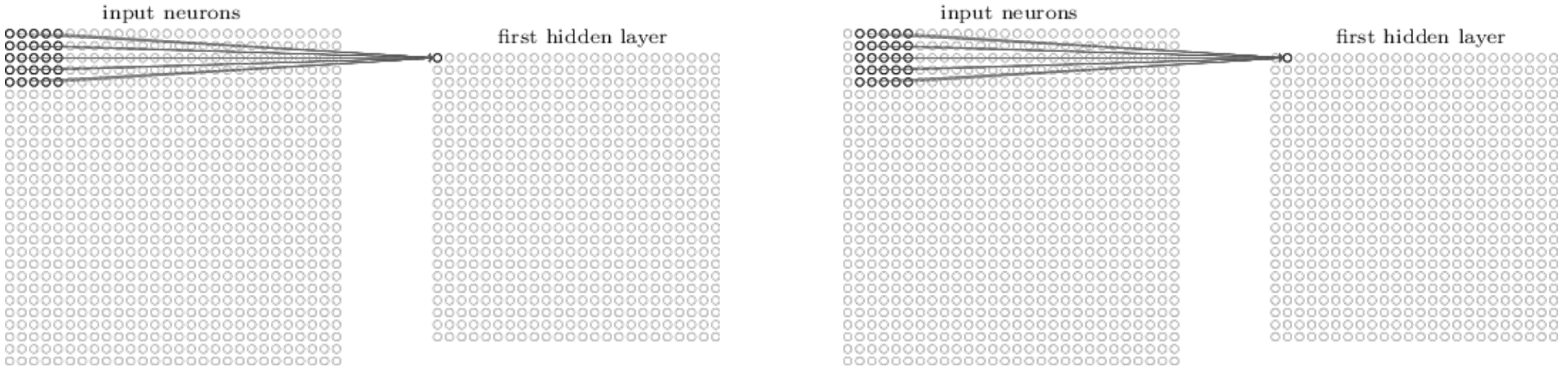
This LRF layer has 25 x 24 x 24 + (24 x 24) parameters = 14976 parameters

http://neuralnetworksanddeeplearning.com/chap6.html

Minimizing the number of parameters: LRF's

One possible solution for this problem is to consider **local receptive fields** (LRF) (an inspiration that comes from neuroscience).

A local receptive field is a standard MLP with a local view of the image: its input is a submatrix $M \times M$ of the original $N \times N$, where $M << N$.



Original image = 28 x 28 pixels

This LRF layer has 14976 parameters

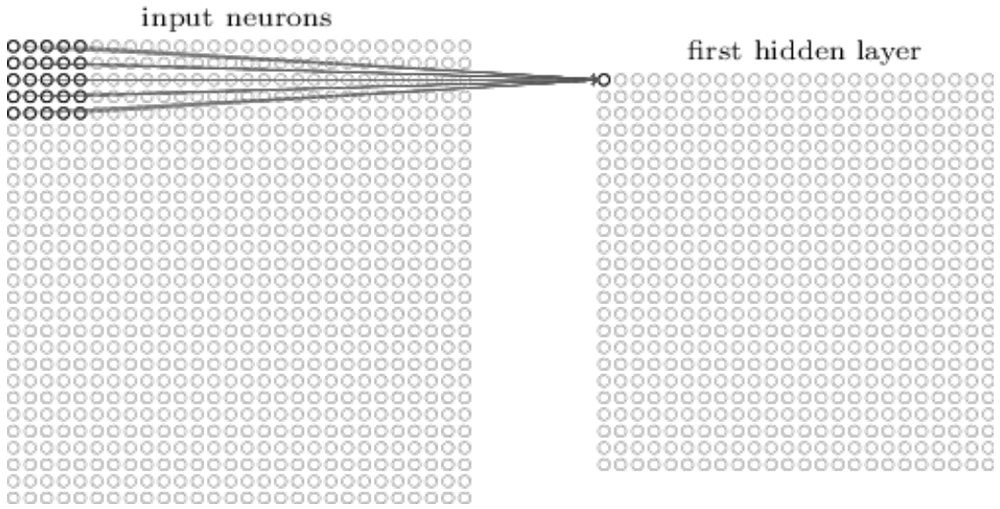
We have shown the local receptive field being moved by one pixel at a time, but this can be redundant in the case of natural images. In fact, sometimes a different **stride** length is used. For instance, we might move the local receptive field 2 pixels to the right (or down), in which case we'd say a stride length of 2 is used. This also **reduces the number units of the first layer** and consequently the number of parameters.

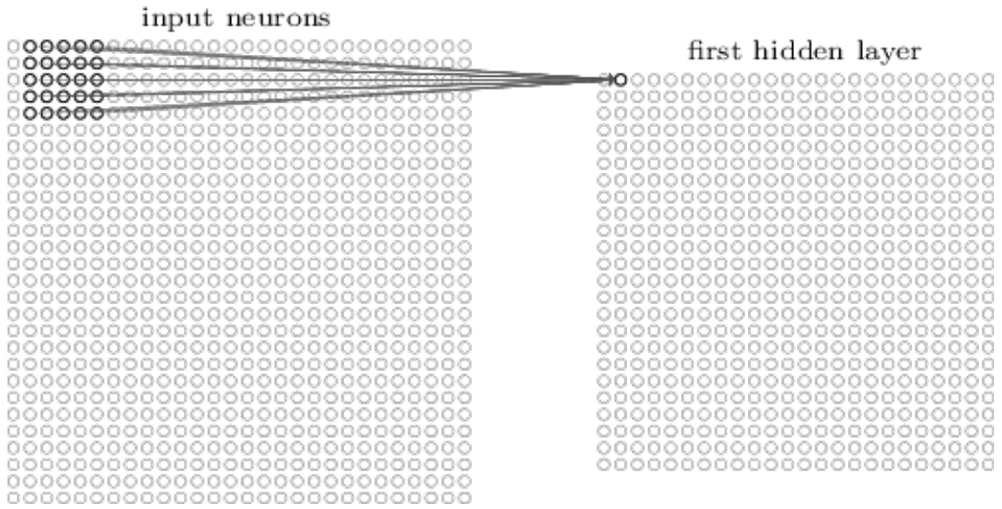
http://neuralnetworksanddeeplearning.com/chap6.html

Minimizing the number of parameters: LRF's

One possible solution for this problem is to consider **local receptive fields** (LRF) (an inspiration that comes from neuroscience).

A local receptive field is a standard MLP with a local view of the image: its input is a submatrix $M \times M$ of the original $N \times N$, where $M << N$.





Original image = 28 x 28 pixels

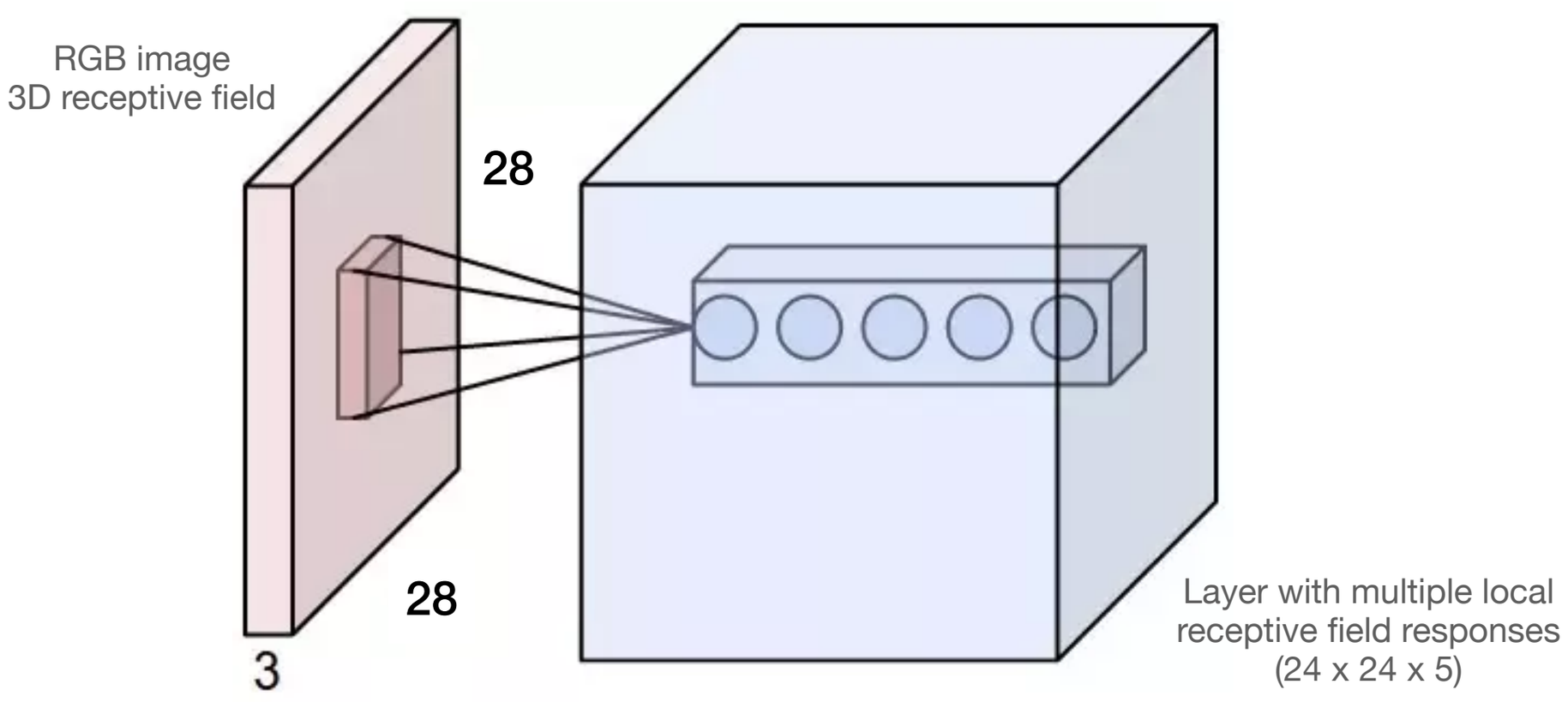
This LRF layer has 25 x 24 x 24 + (24 x 24) parameters = 14976 parameters

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

http://neuralnetworksanddeeplearning.com/chap6.html

Minimizing the number of parameters: LRF's

Additionally, to have more capacity, we can compute **several local receptive fields** per pixel and also to consider **3D local receptive fields** for color images:



number of receptive fields per pixel

This LRF layer has $\underline{25 \times 3} \times \underline{5 \times 24 \times 24} + \underline{(24 \times 24 \times 5)} = 218880$ parameters

parameters per receptive field

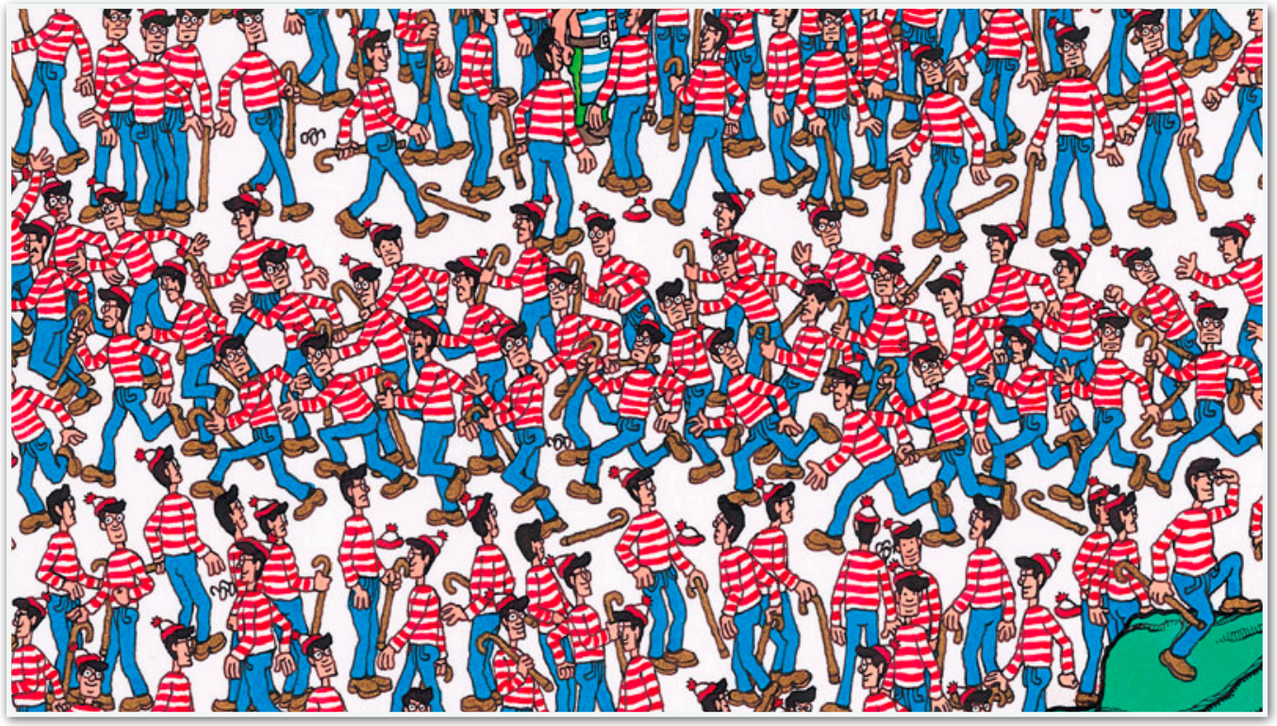
number of units in the first layer

biases

Froom LRF's to convolutions

Using LRF is like decomposing the problem of image analysis in a set of local experts that are specialized in certain parts of the image. The outcome of these specialists can be integrated at a high level of the model by **stacking** layers. This can be useful for some applications.

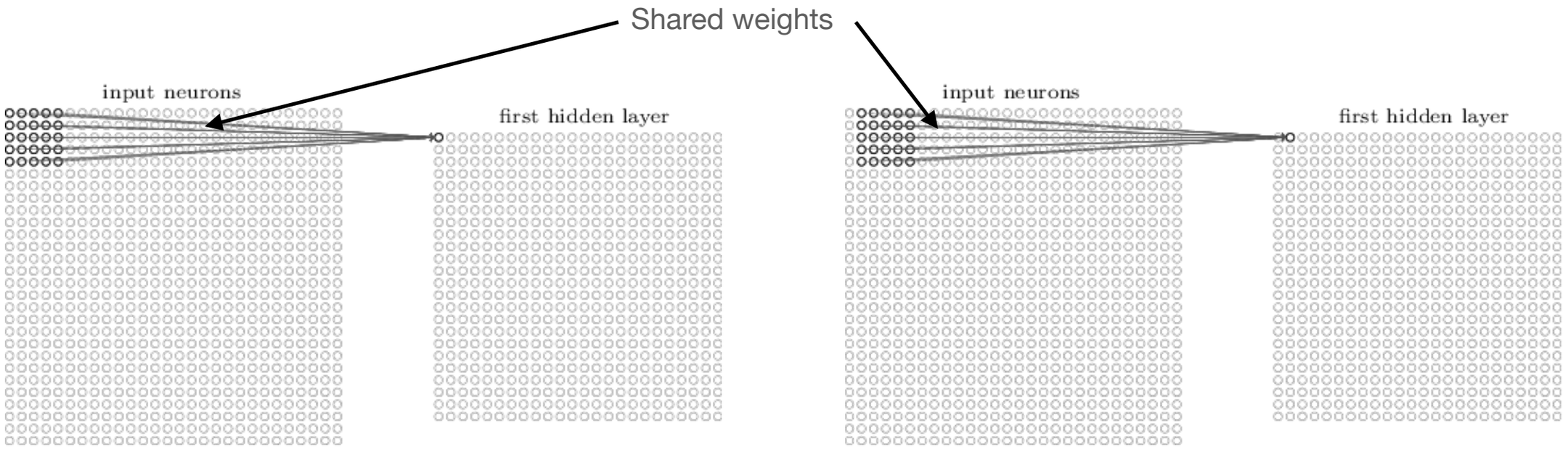
But in natural images, an object may appear at any image coordinates!



In this case LRF are not suited for this task because in order to learn a detector we should provide the model with a lot of examples in order to consider all possible object localizations!

Froom LRF's to convolutional layers

But, what about **sharing** LRF weights (and using several receptive fields per pixel)?



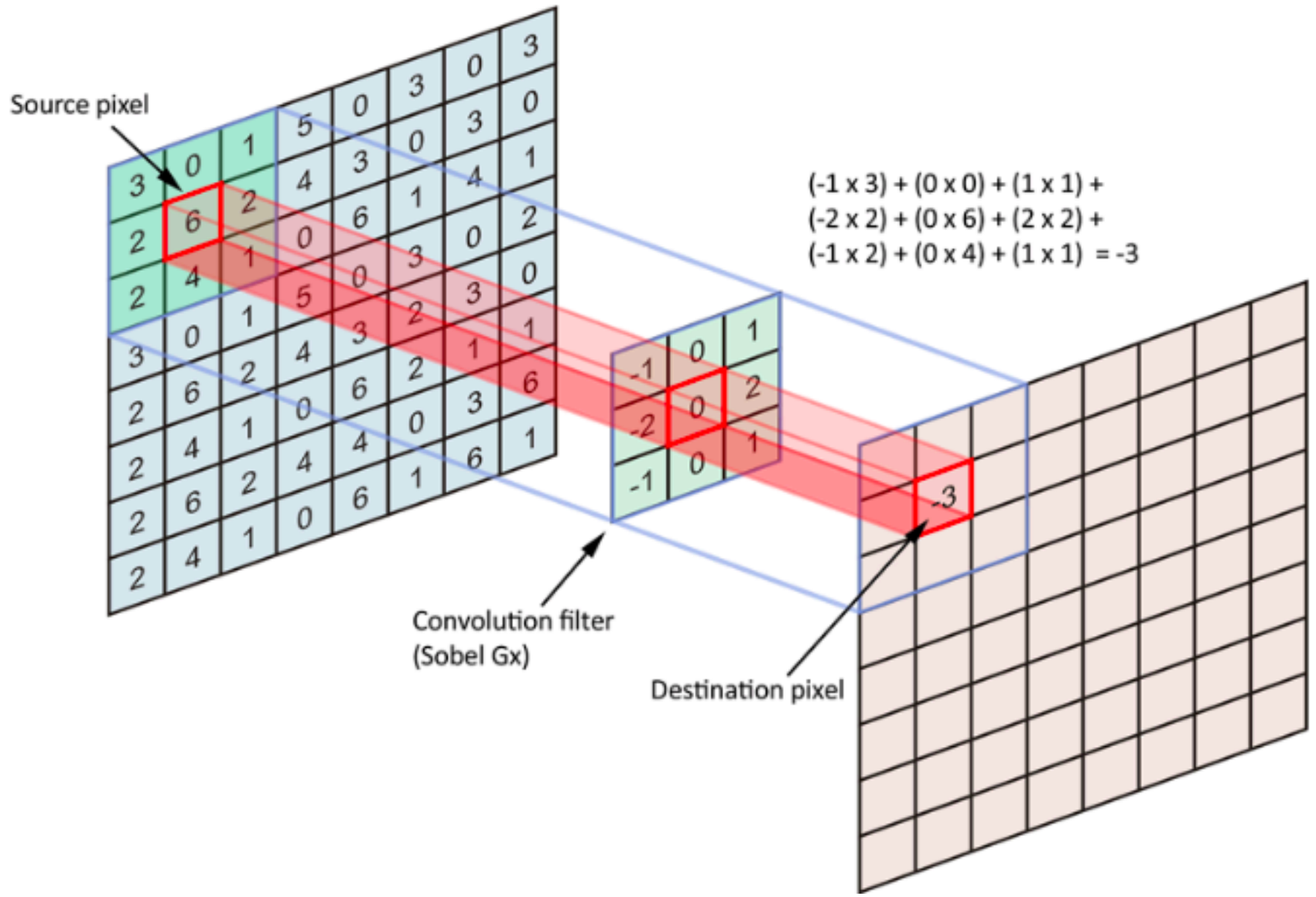
This can be expressed as a **convolution**. Now, for the (j, k) hidden unit, the output is of "shared" a receptive field can be expressed as:

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

where b is the shared value for the bias, $w_{l,m}$ is a 5x5 array of shared weights and $a_{x,y}$ represents the input activation at position x, y .

Convolutional layers

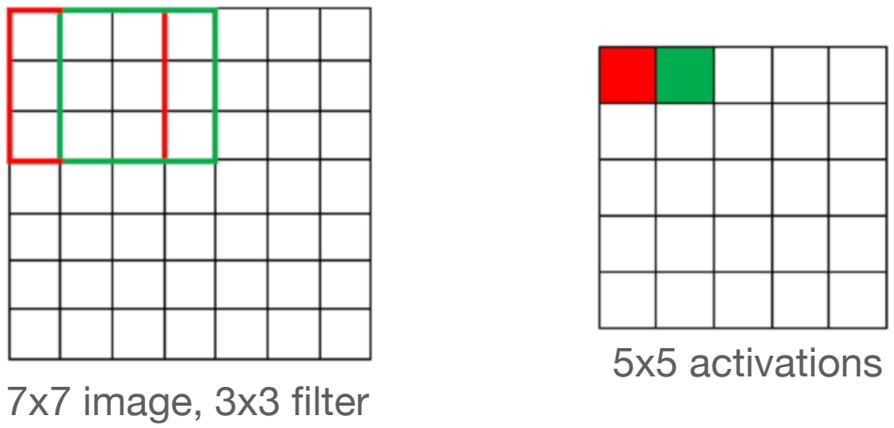
The term **convolution** refers to the mathematical combination of two functions to produce a third function. It merges two sets of information.



https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2

Convolutional layers

For a gray scale (n x n) image and (f x f) filter/kernel/LRF, the dimensions of the image resulting from a convolution operation is (n – f + 1) x (n – f + 1).



Thus, the image shrinks every time a convolution operation is performed. This places an upper **limit to the number of times such an operation could be performed before the image reduces to nothing** thereby precluding us from building deeper networks.

To overcome these problems, we use **padding**. Padding is simply a process of adding “dimensions” to our input images so as to avoid the problems mentioned above.

Convolutional layers

Input: 6x6 image

0	0	0	0	0	0	0	0
0	1	2	3	1	3	5	0
0	2	2	5	4	2	5	0
0	0	6	9	6	2	2	0
0	2	0	1	9	4	0	0
0	5	5	4	6	7	6	0
0	6	1	3	7	1	5	0
0	0	0	0	0	0	0	0

0-padding

*

3x3 Filter

1	0	-1
1	0	-1
1	0	-1

=

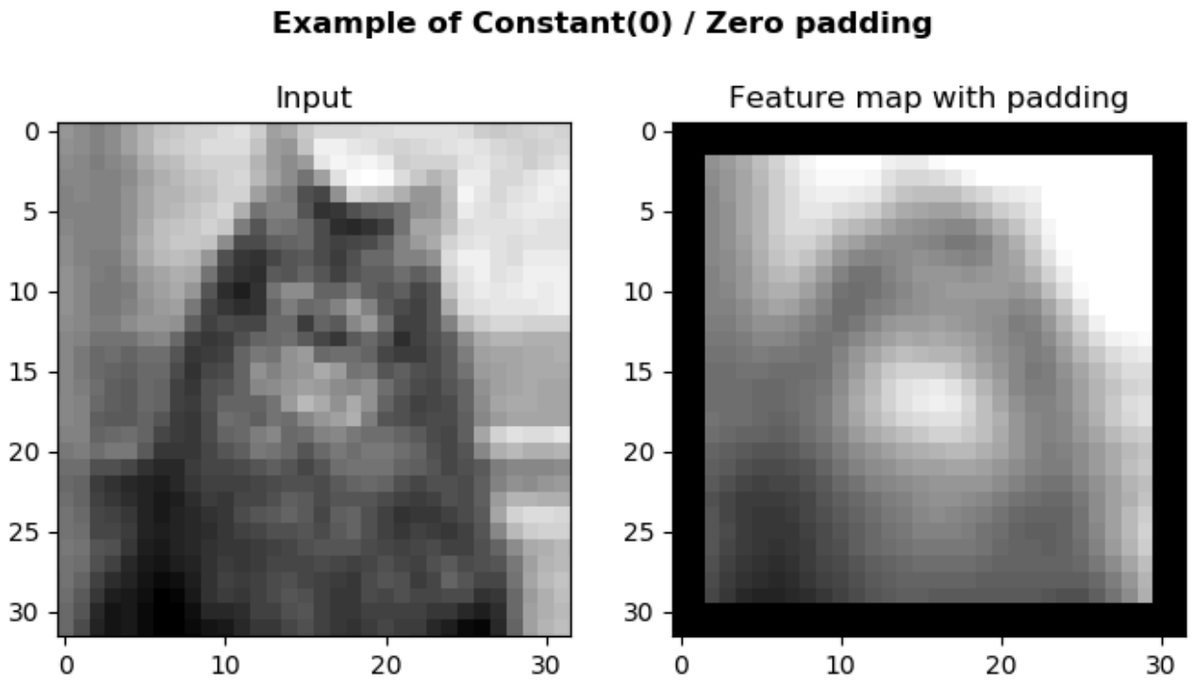
Output: 6x6 activations

-4	-5	-1	3	-5	5
-10	-14	-1	10	-1	7
-8	-11	-11	7	12	8
11	-7	-10	1	13	13
-6	5	-16	-4	10	12
-6	4	-7	-1	2	8

https://www.brilliantcode.net/1584/convolutional-neural-networks-1-convolution-layer-stride-padding-kernel/

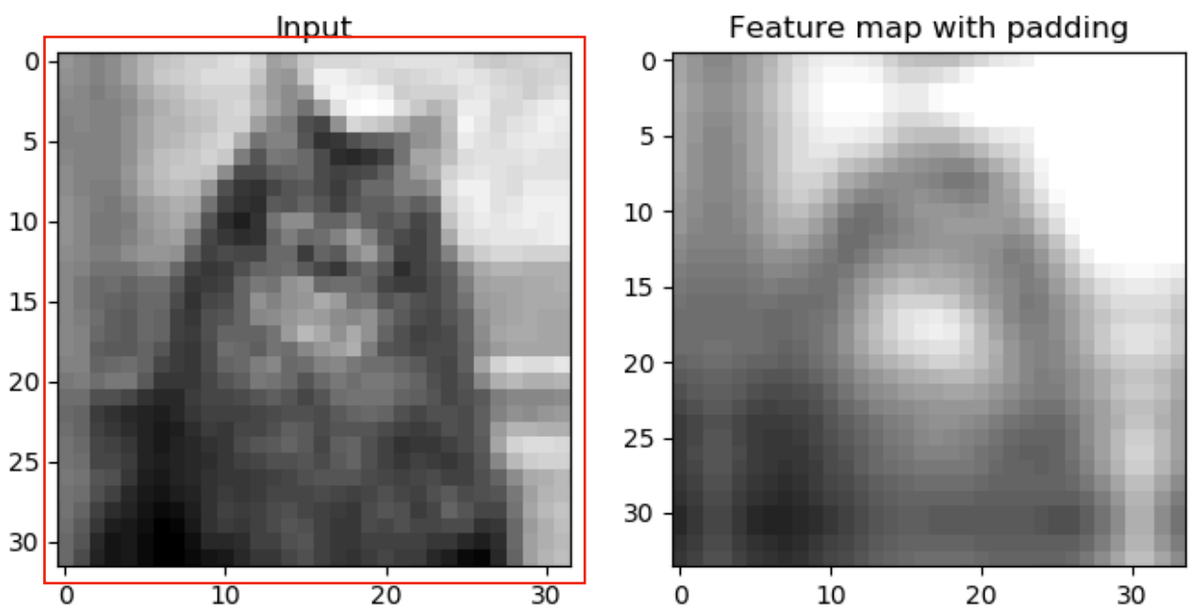
Convolutional layers

32 x 32 image



32 x 32 activation map

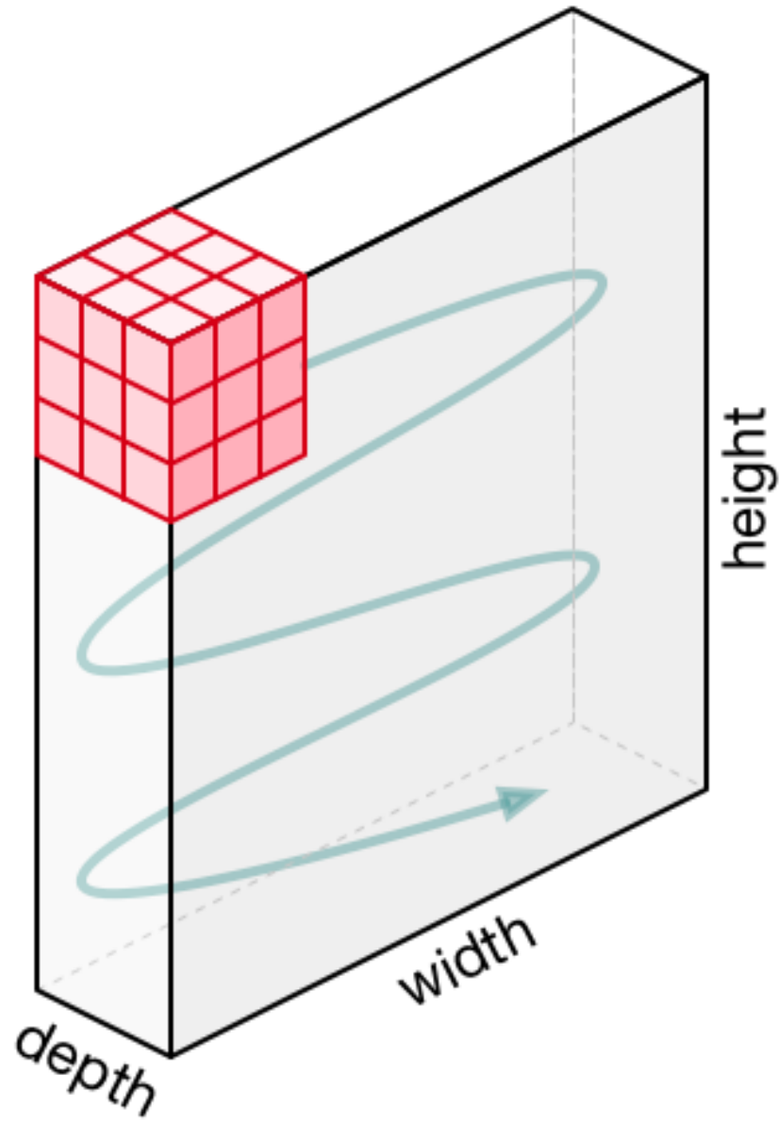
34 x 34 image



32 x 32 activation map

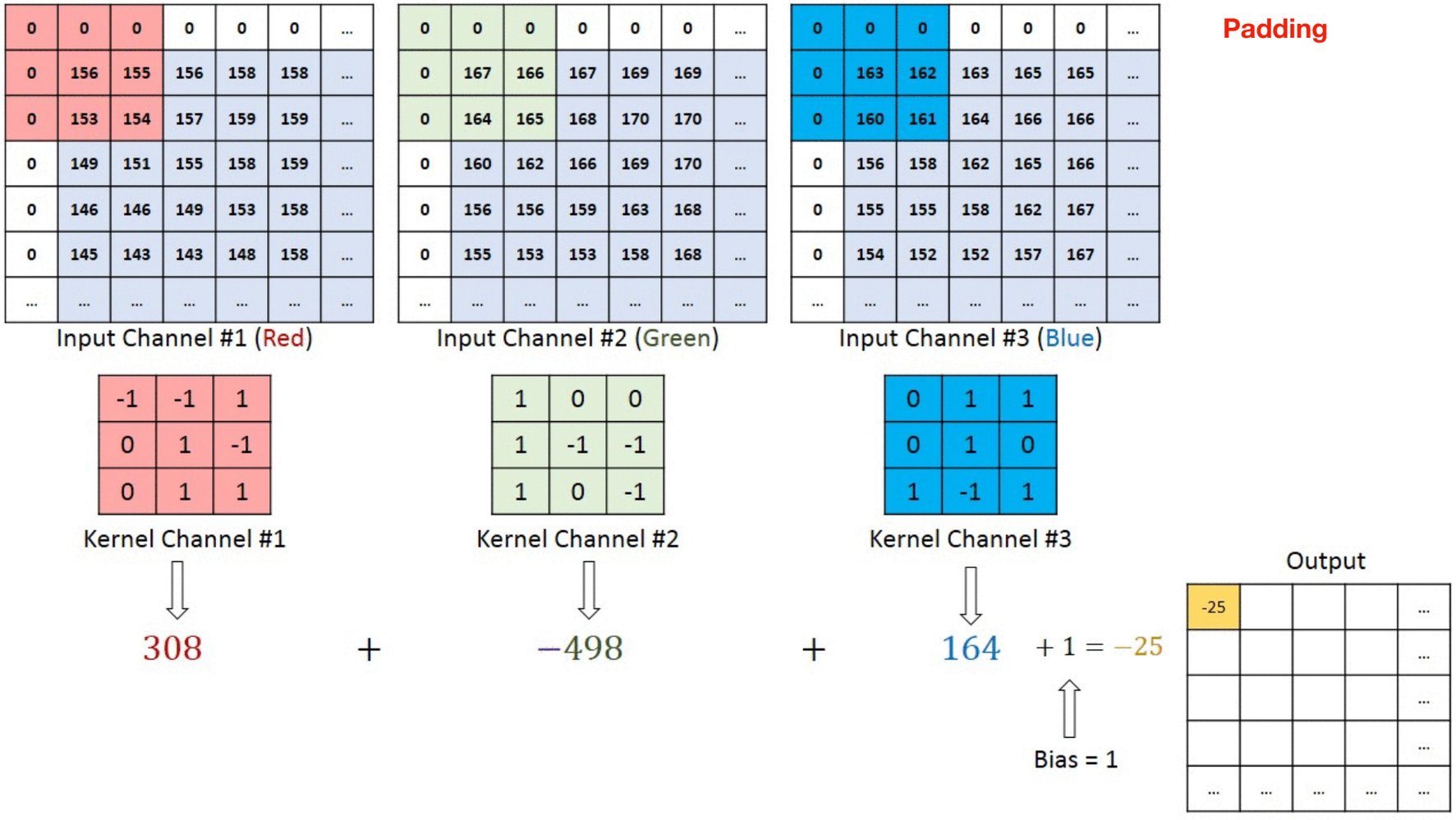
https://www.machinecurve.com/index.php/2020/02/07/what-is-padding-in-a-neural-network/

Convolutional layers (tensors)



https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Convolutional layers

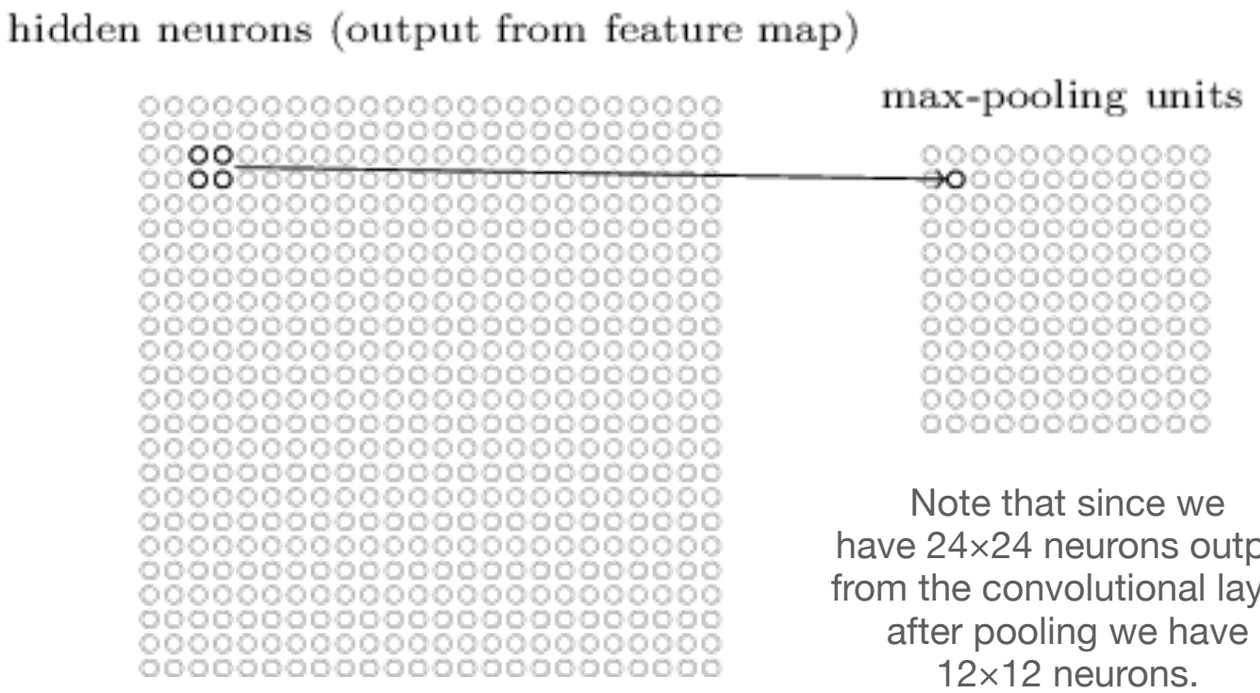


https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Convolutional layers

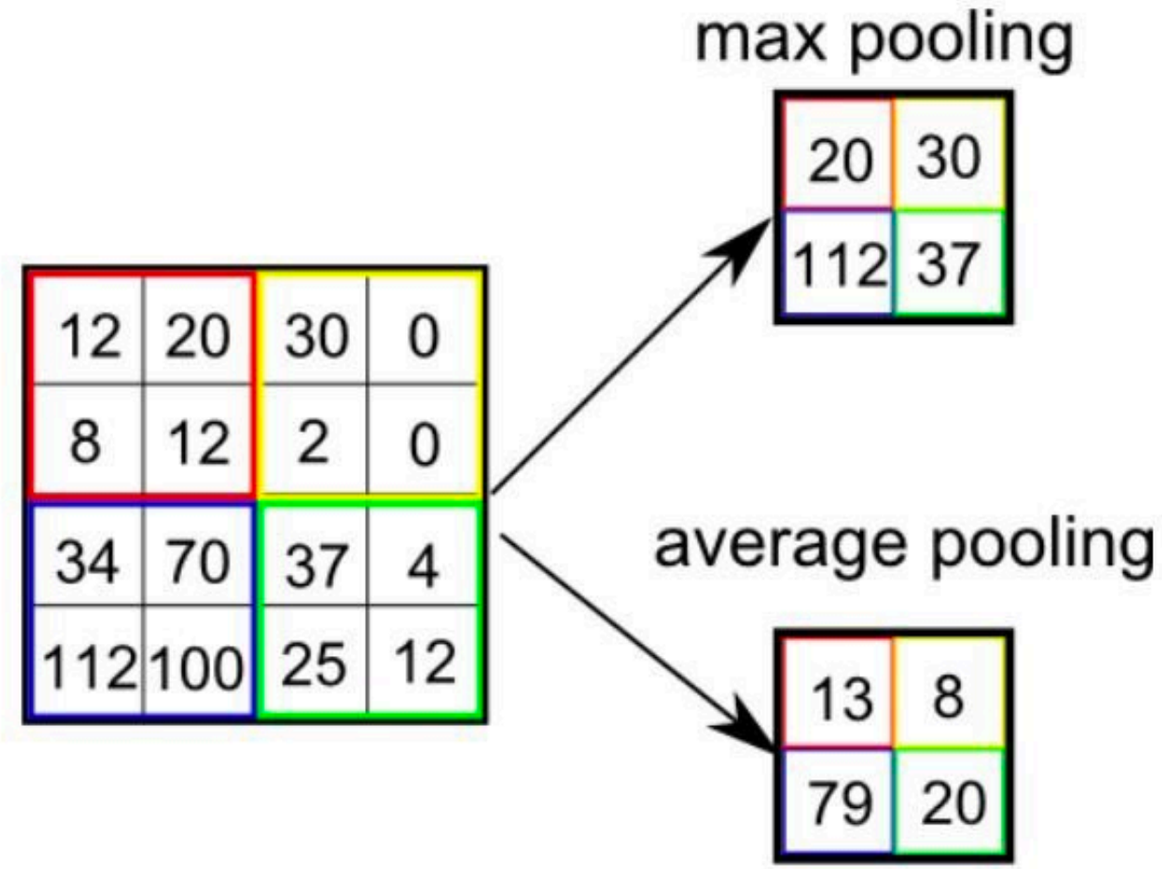
In addition to the convolutional layers just described, convolutional neural networks also contain **pooling layers**. Pooling layers are usually used immediately after convolutional layers. What the pooling layers do is **simplify the information in the output** from the convolutional layer.

As a concrete example, one common procedure for pooling is known as *max-pooling*. In max-pooling, a pooling unit simply outputs the maximum activation in the 2×2 input region, as illustrated in the following diagram:



https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2

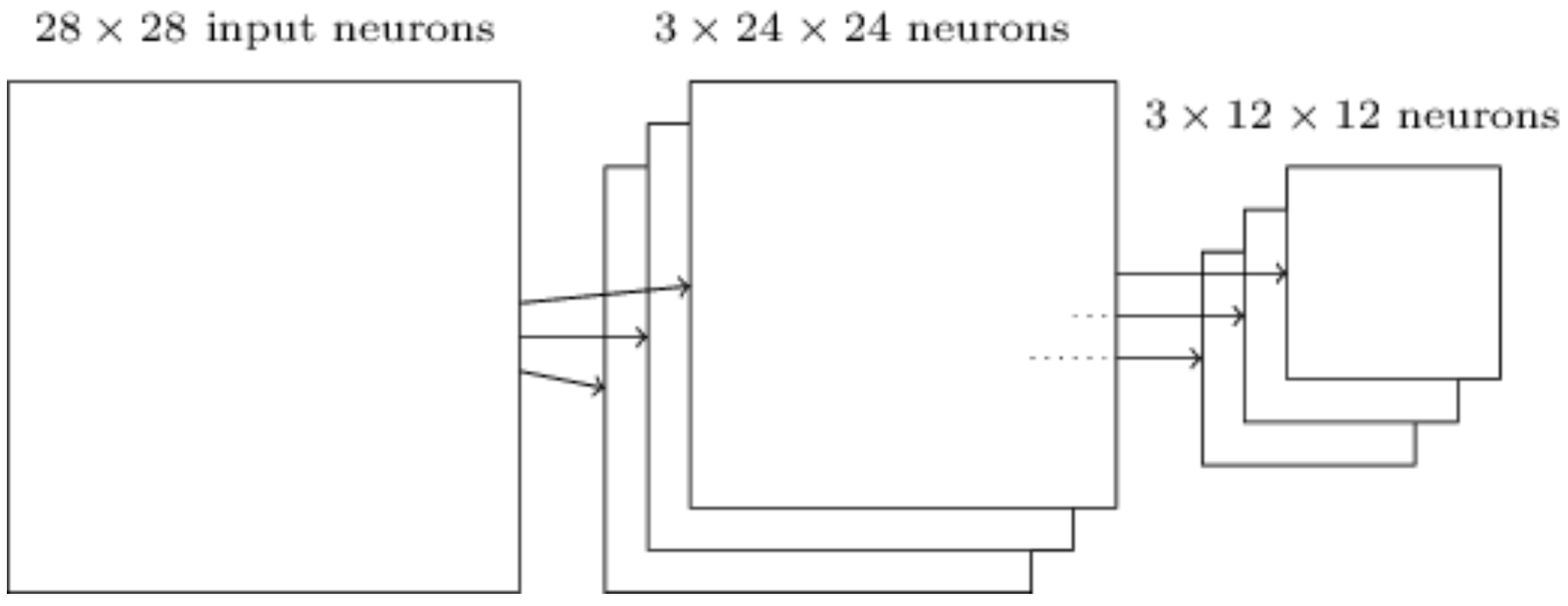
Convolutional layers



https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Convolutional layers

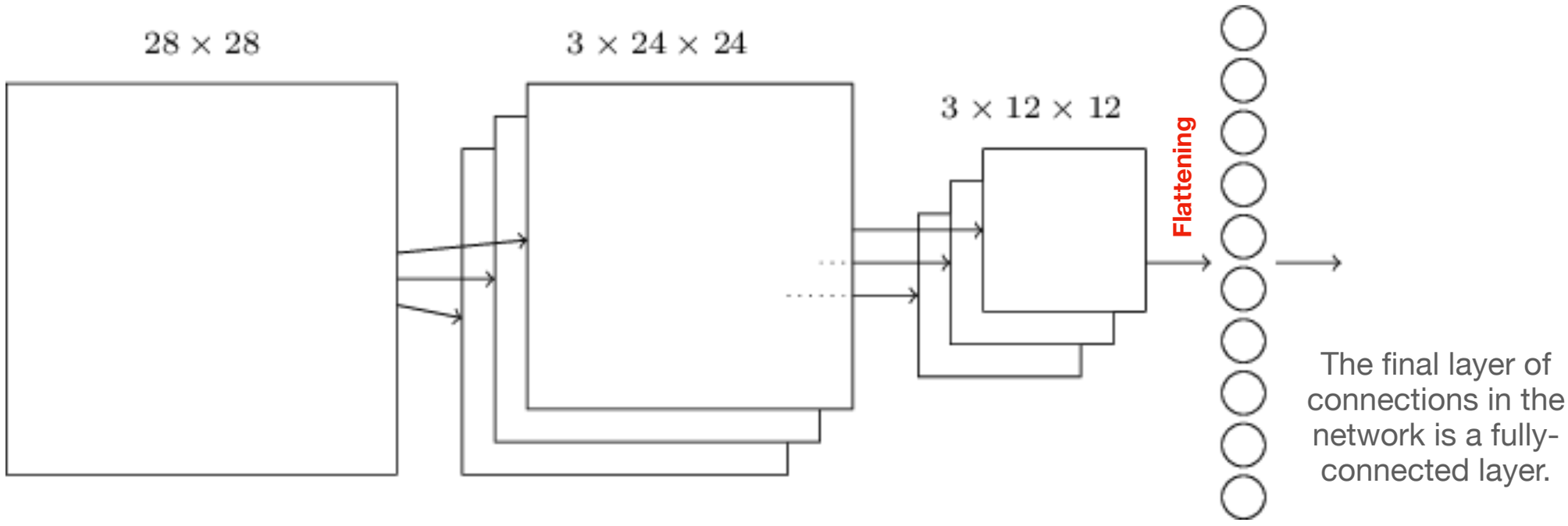
If there were three feature maps with no padding, the combined convolutional and max-pooling layers would look like:



https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2

Convolutional layers

We can now put all these ideas together to form a complete convolutional neural network:



https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2

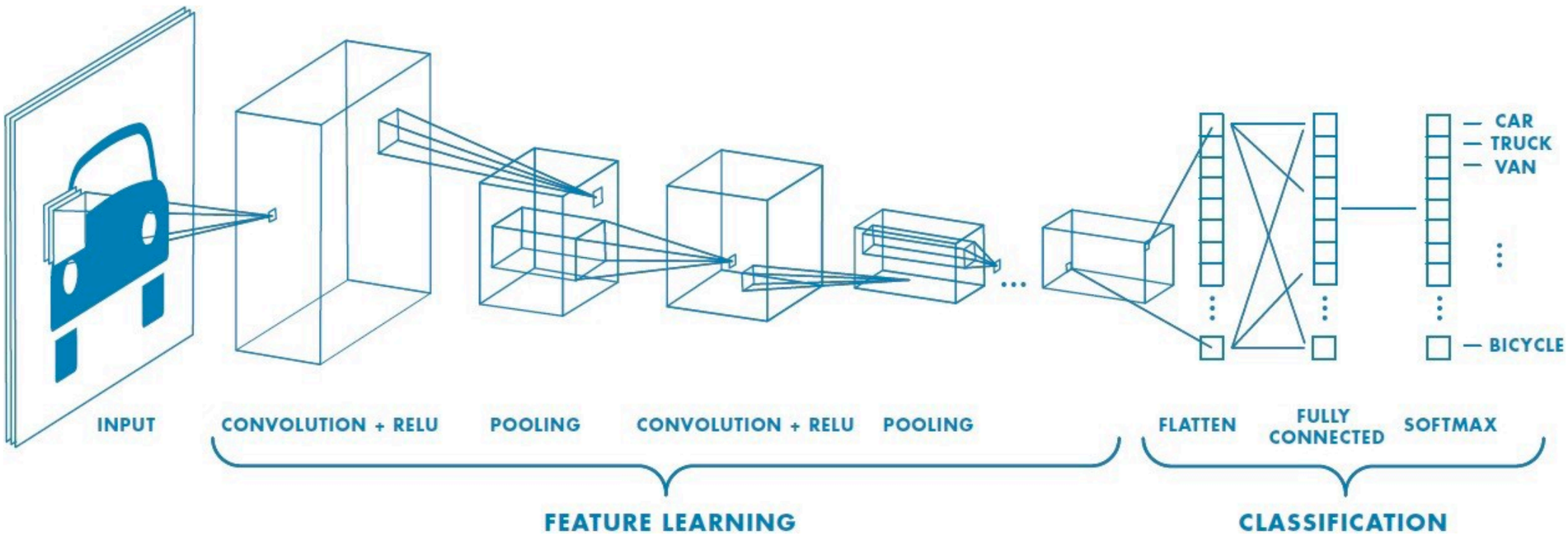
The network begins with 28×28 input neurons, which are used to encode the pixel intensities for the MNIST image. This is then followed by a convolutional layer using a 5×5 local receptive field and 3 feature maps. The result is a layer of 3×24×24 hidden feature neurons. The next step is a max-pooling layer, applied to 2×2 regions, across each of the 3 feature maps. The result is a layer of 3×12×12 hidden feature neurons.

Convolutional layers and stacking

The **stacking** of convolutional layers allows a **hierarchical decomposition** of the input.

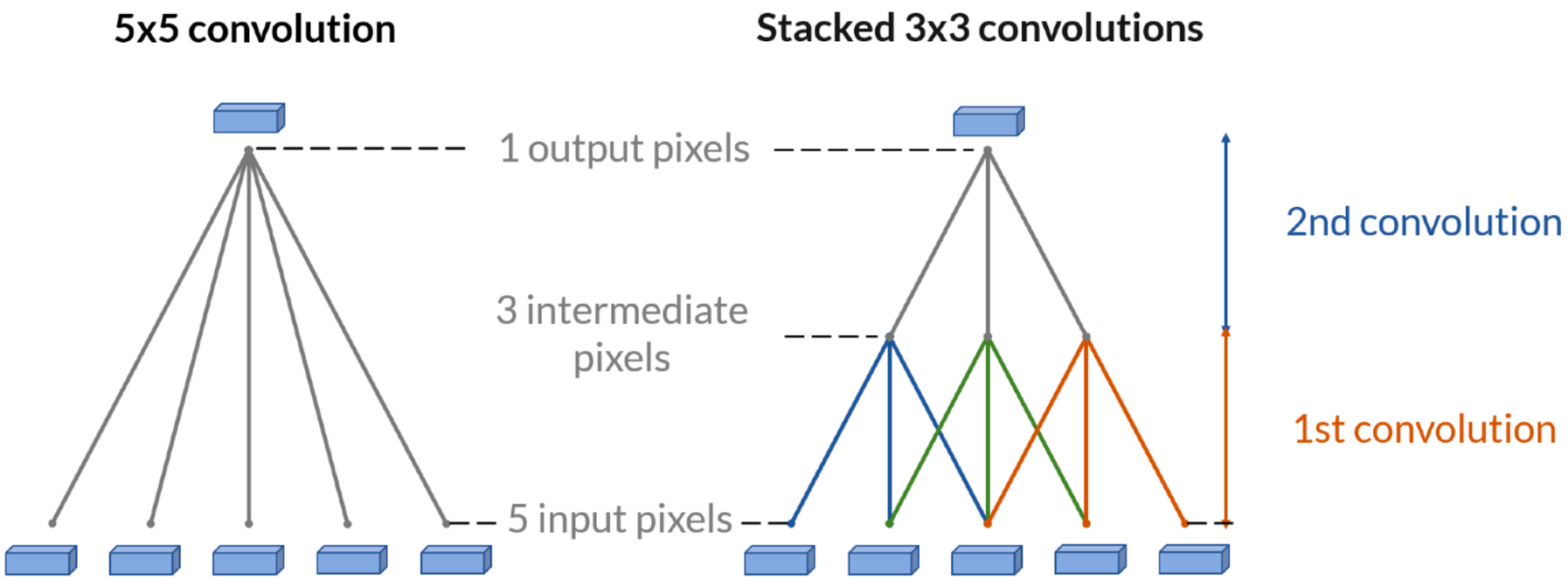
Consider that the filters that operate directly on the raw pixel values will learn to extract low-level features, such as lines and small patterns. The filters that operate on the output of the first line layers may extract features that are combinations of lower-level features, such as features that comprise multiple lines to express shapes.

This process continues until very deep layers are extracting faces, animals, houses, and so on.



https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Convolutional layers and stacking



We can replace large convolution kernels with multiple 3x3 convolutions on top of one another.

This is good for two reasons: deeper is better and less computational cost.

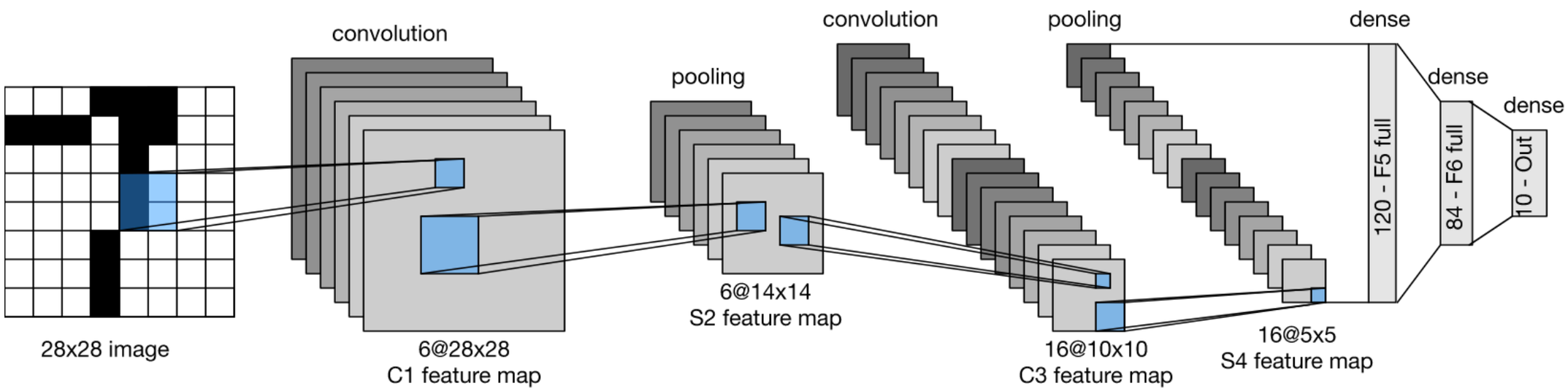
Convolutional layers in Keras

```
1
2 model = keras.Sequential(
3     [
4         keras.Input(shape=input_shape),
5         layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
6         layers.MaxPooling2D(pool_size=(2, 2)),
7         layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
8         layers.MaxPooling2D(pool_size=(2, 2)),
 9         layers.Flatten(),
10         layers.Dropout(0.5),
11         layers.Dense(num_classes, activation="softmax"),
12     ]
13 )
14
15 model.summary()
16
```

```
Test loss: 0.027600124478340 15
Test accuracy: 0.990000009536743 2
```

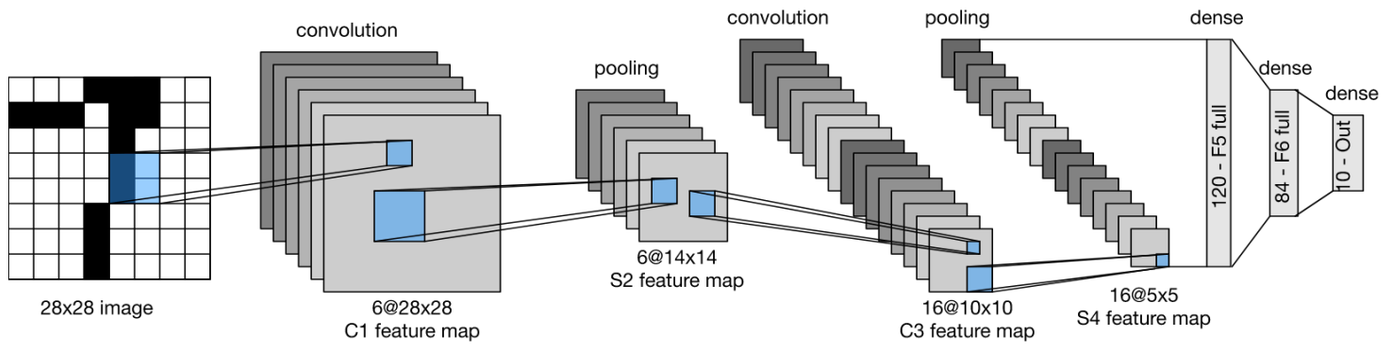
LeNET

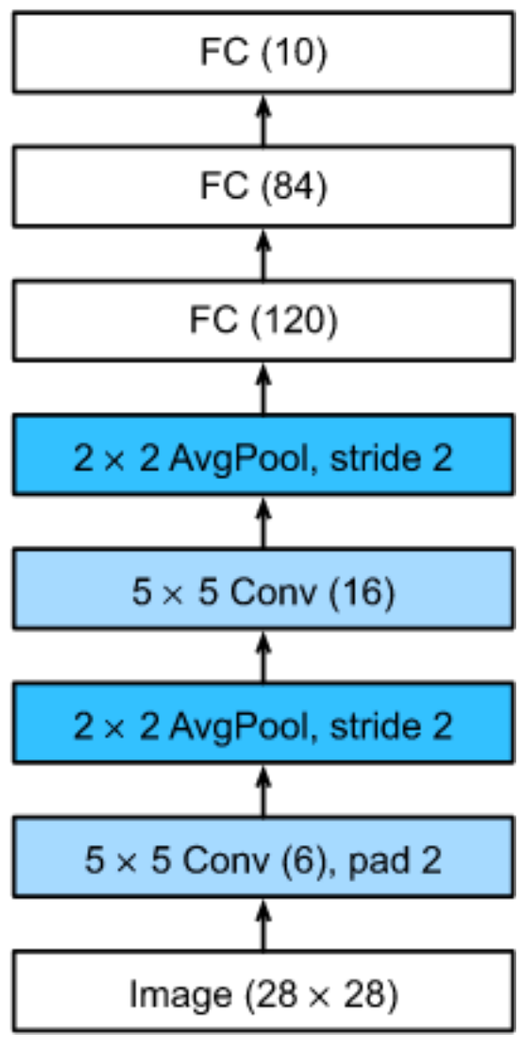
LeNet was the first published CNNs to capture wide attention for its performance on computer vision tasks. The model was introduced by (and named for) Yann LeCun, then a researcher at AT&T Bell Labs, for the purpose of recognizing handwritten digits in images.
This work represented the culmination of a decade of research developing the technology. In 1989, LeCun published the first study to successfully train CNNs via backpropagation.



https://d2l.ai/chapter_convolutional-neural-networks/lenet.html

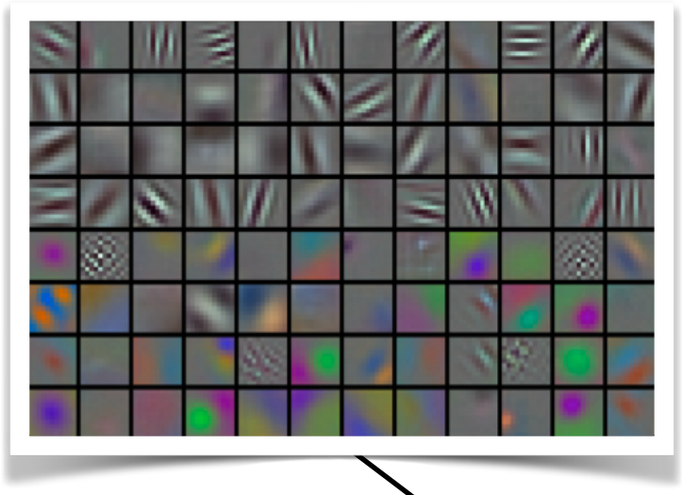
LeNET



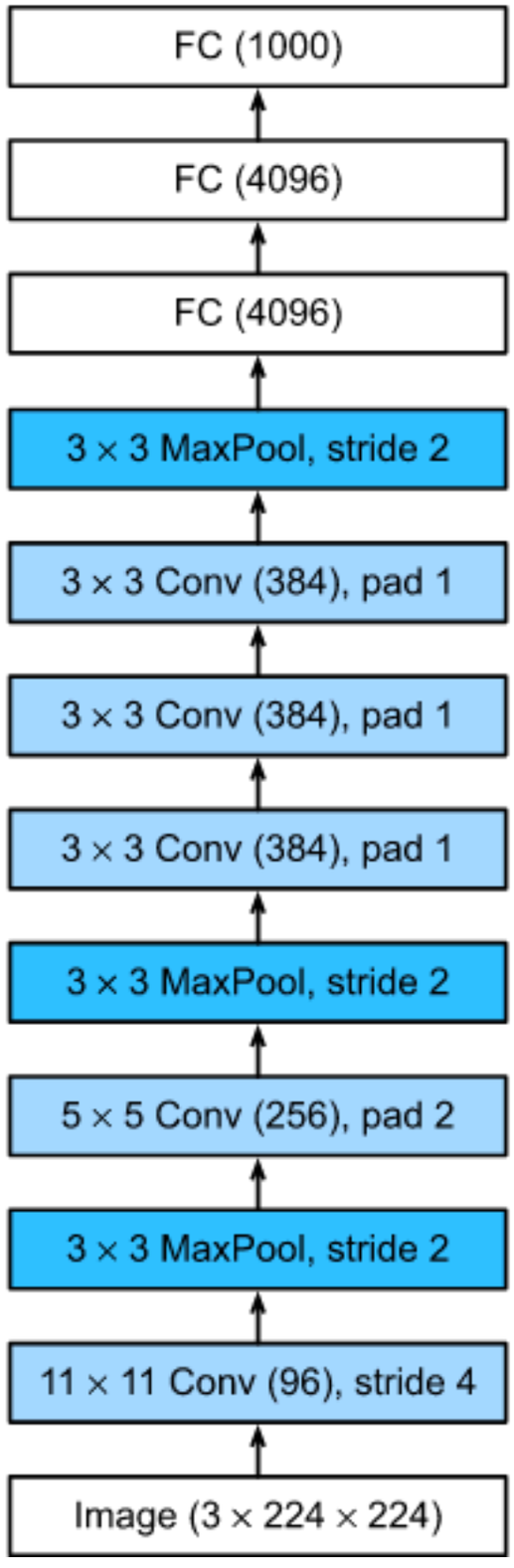


https://d2l.ai/chapter_convolutional-neural-networks/lenet.html

AlexNet

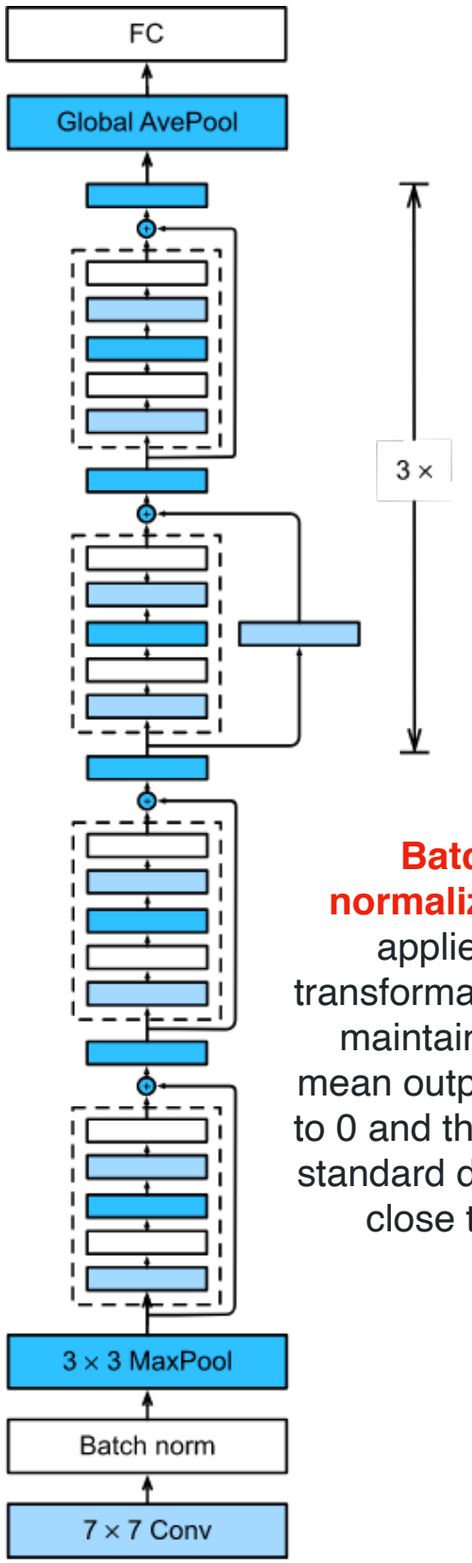


Filters that operate in raw pixels can be visualized because they are like images (11 x 11 x 3).

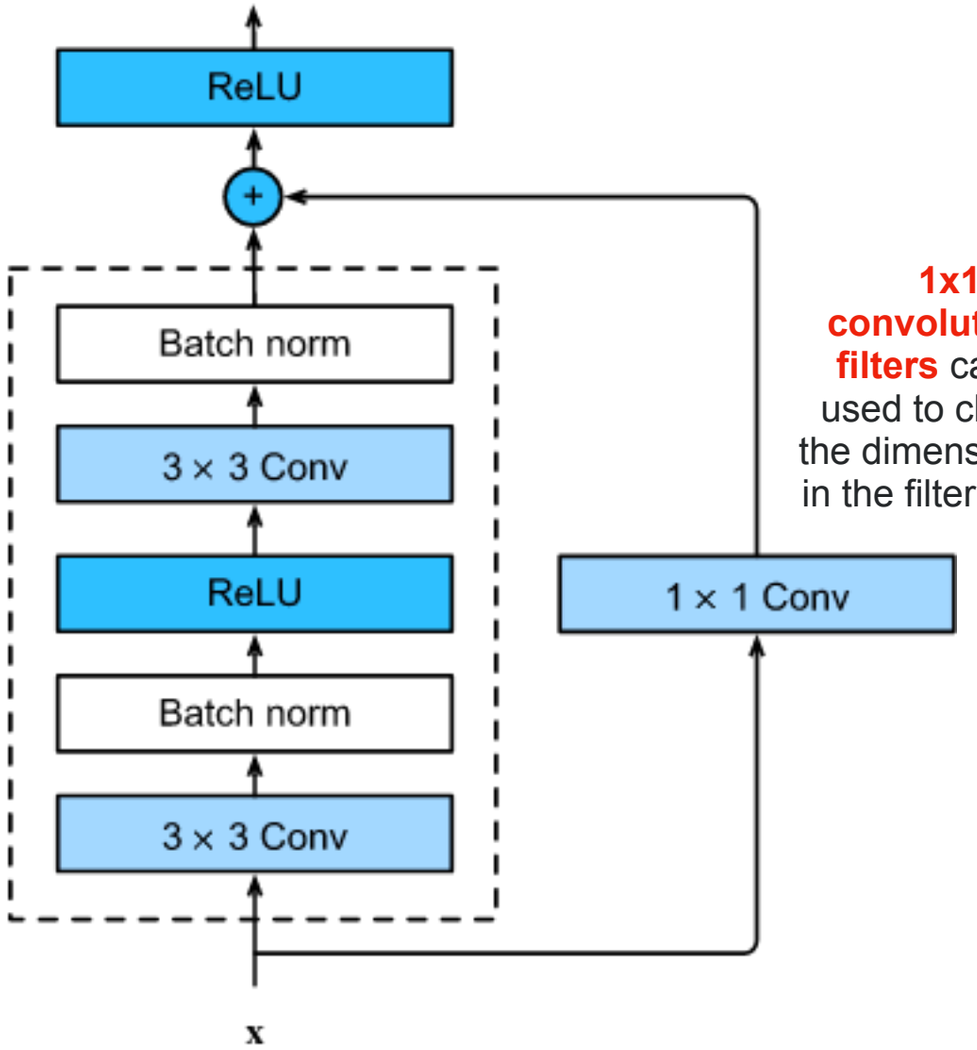


https://d2l.ai/chapter_convolutional-neural-networks/lenet.html

ResNet



Batch normalization applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1.

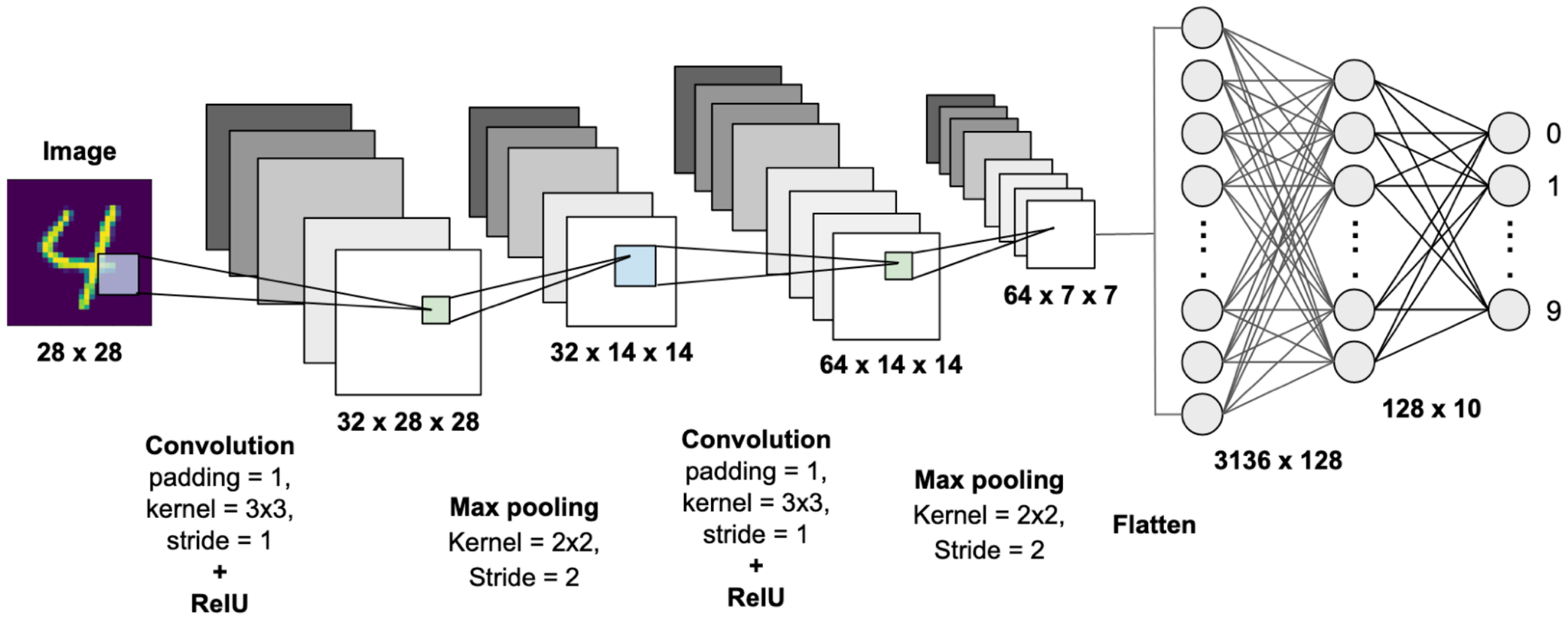


1x1 convolutional filters can be used to change the dimensionality in the filter space.

https://d2l.ai/chapter_convolutional-neural-networks/lenet.html

Fully Convolutional Networks

The use of a fully connected layer at the end of a model represents a severe limitation!



https://becominghuman.ai/building-a-convolutional-neural-network-cnn-model-for-image-classification-116f77a7a236

We can only process images of this size! What about larger images?

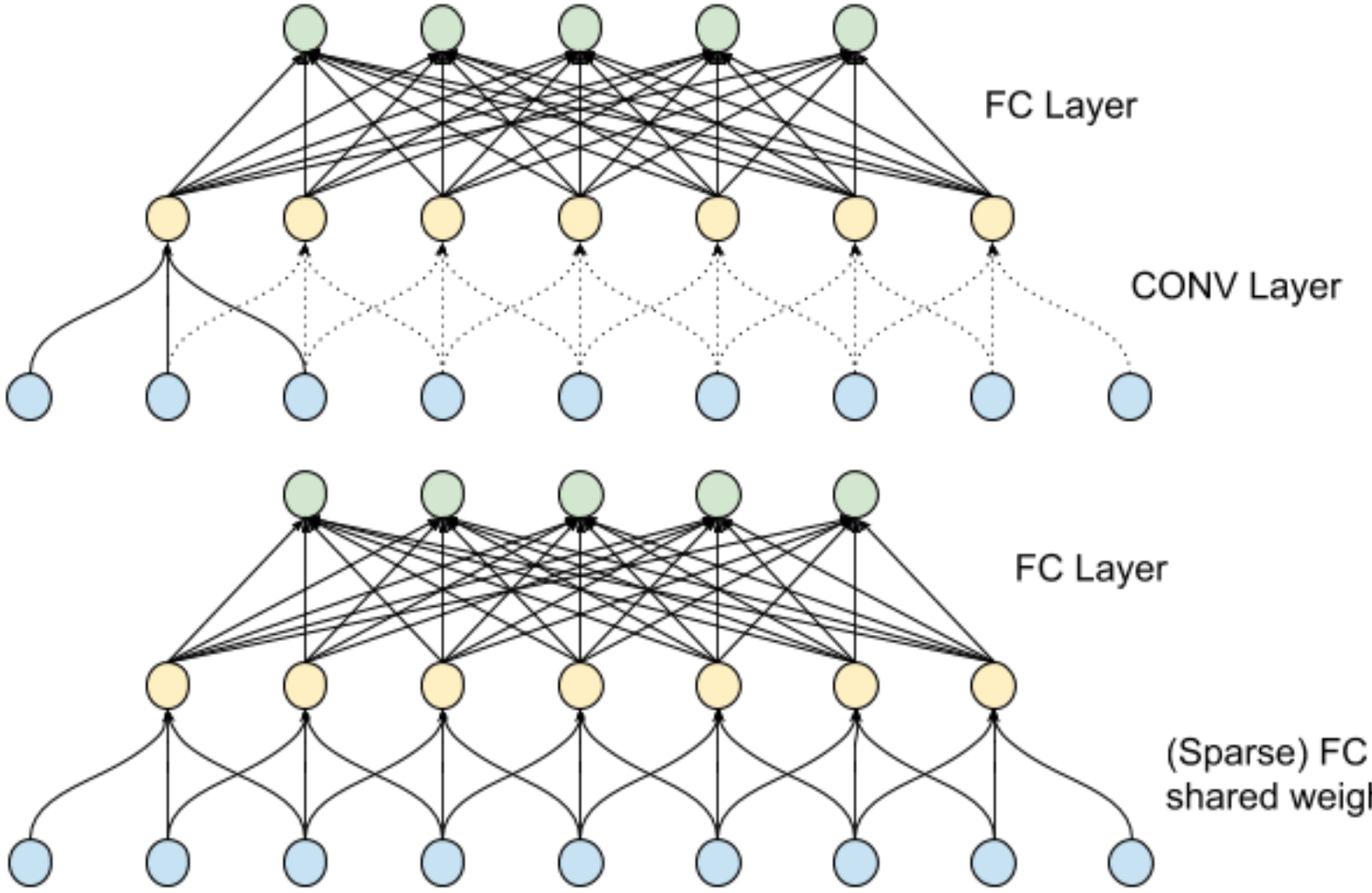
Fully Convolutional Networks

The only difference between Fully Connected (FC) and Convolutional (CONV) layers is that the neurons in the CONV layer are connected only to a local region in the input, and that many of the neurons in a CONV volume share parameters.

However, the neurons in both layers still compute dot products, so their functional form is identical.

Then, it is easy to see that for any CONV layer there is an FC layer that implements the same forward function. The weight matrix would be a large matrix that is mostly zero except for at certain blocks (due to local connectivity) where the weights in many of the blocks are equal (due to parameter sharing).

Fully Convolutional Networks



Computational complexity is higher because we store a very large matrix!

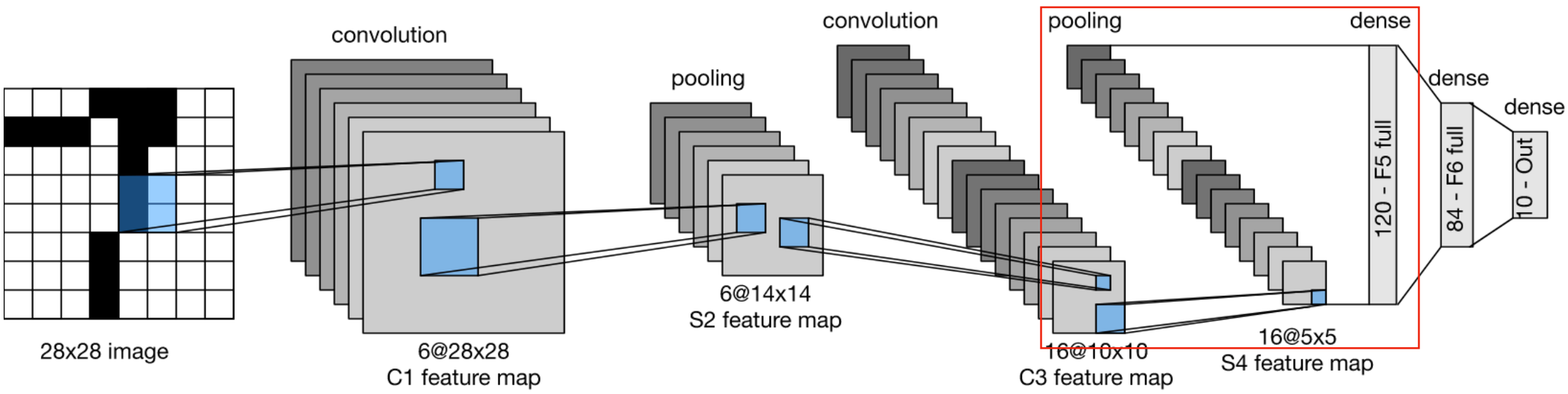
Fully Convolutional Networks

Conversely, any FC layer can be converted to a CONV layer.

F: receptive field size; K:depth

For example, an FC layer with K=4096 that is looking at some input volume of size 7×7×512 can be equivalently expressed as a CONV layer with F=7, K=4096. In other words, we are setting the filter size to be exactly the size of the input volume, and hence the output will simply be 1×1×4096 since only a single depth column "fits" across the input volume, giving identical result as the initial FC layer.

K: 120

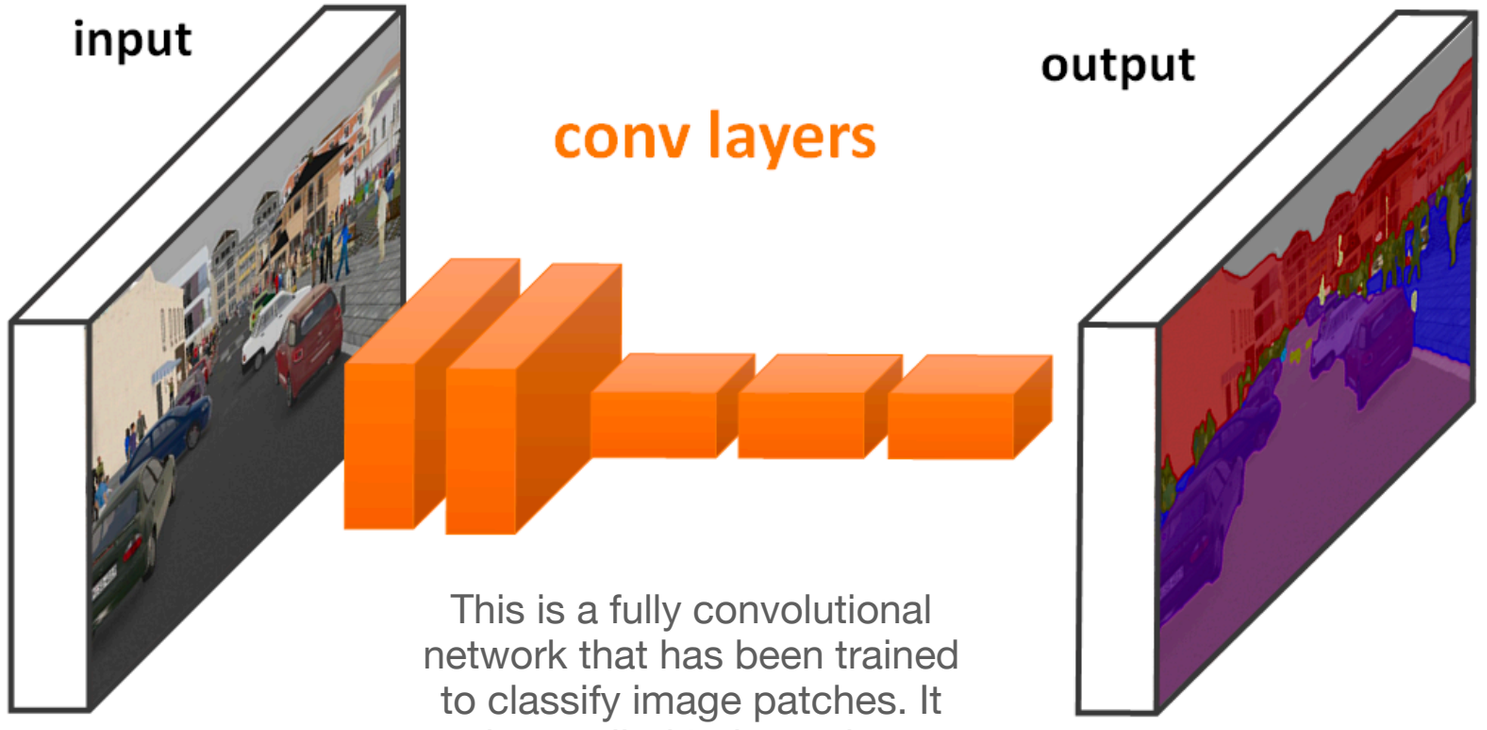


F: 5; K:120

Fully Convolutional Networks

It turns out that this conversion allows us to “slide” the original ConvNet very efficiently across many spatial positions in a larger image, in a single forward pass.

Now the model is independent of the input image size!



Fully Convolutional Networks

```
model = keras.Sequential(
    [
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(128, kernel_size=(3, 3), activation="relu"),
        layers.Conv2D(num_classes, kernel_size=(1, 1), activation="softmax"),
        layers.GlobalAveragePooling2D(),
    ]
)

model.summary()
```

```
Test loss: 0.050505142658948
Test accuracy: 0.9909999966621399
```