Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

06 - Convolutional Neural Networks

François Pitié

Assistant Professor in Media Signal Processing
Department of Electronic & Electrical Engineering, Trinity College Dublin

[4C16/5C16] Deep Learning and its Applications — 2025/2026

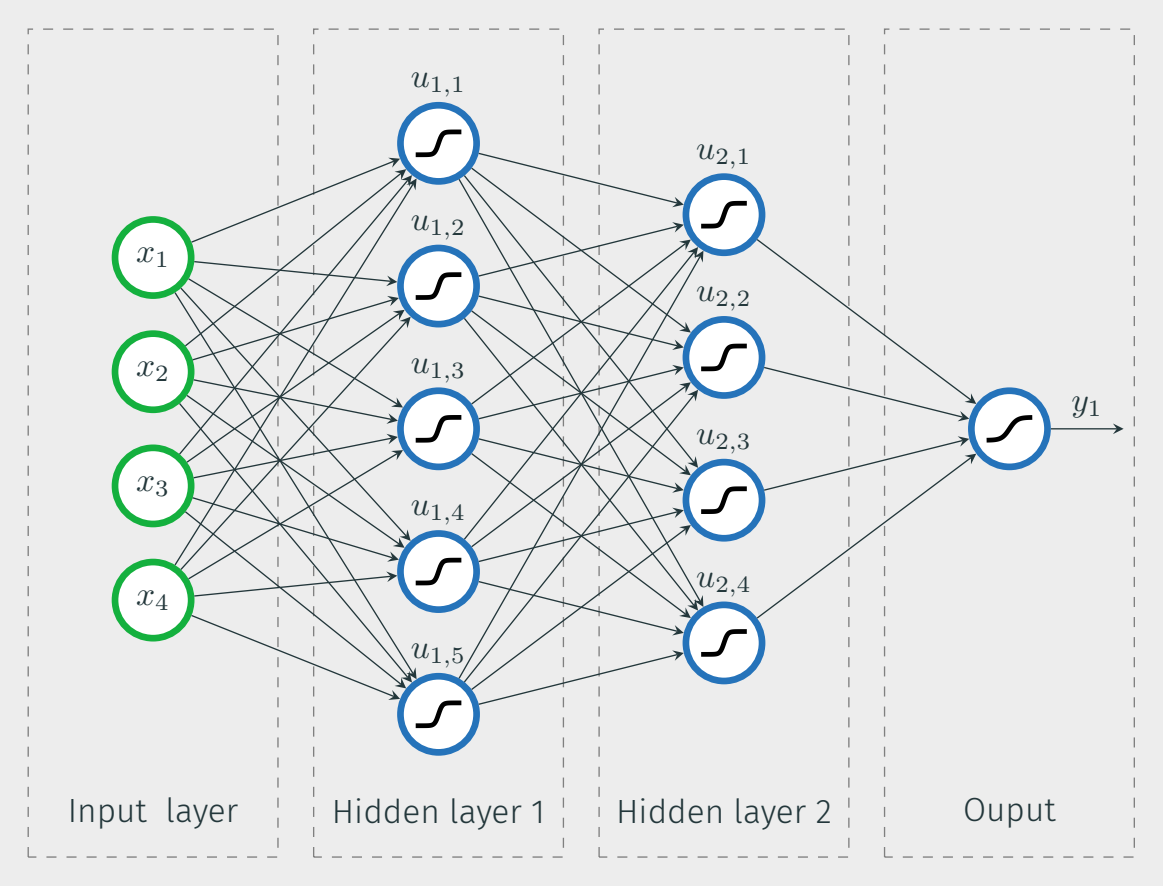
Convolutional Neural Networks, or convnets, are a type of neural net especially used for processing image data.

They are inspired by the organisation of the visual cortex and mathematically based on a well understood signal processing tool: image filtering by convolution.

Convnets gained popularity with LeNet-5, a pioneering 7-level convolutional network by LeCun et al. (1998) that was successfully applied on the MNIST dataset.

Convolution Filters

Recall that in dense layers, every unit in the layer is connected to every unit in the adjacent layers:



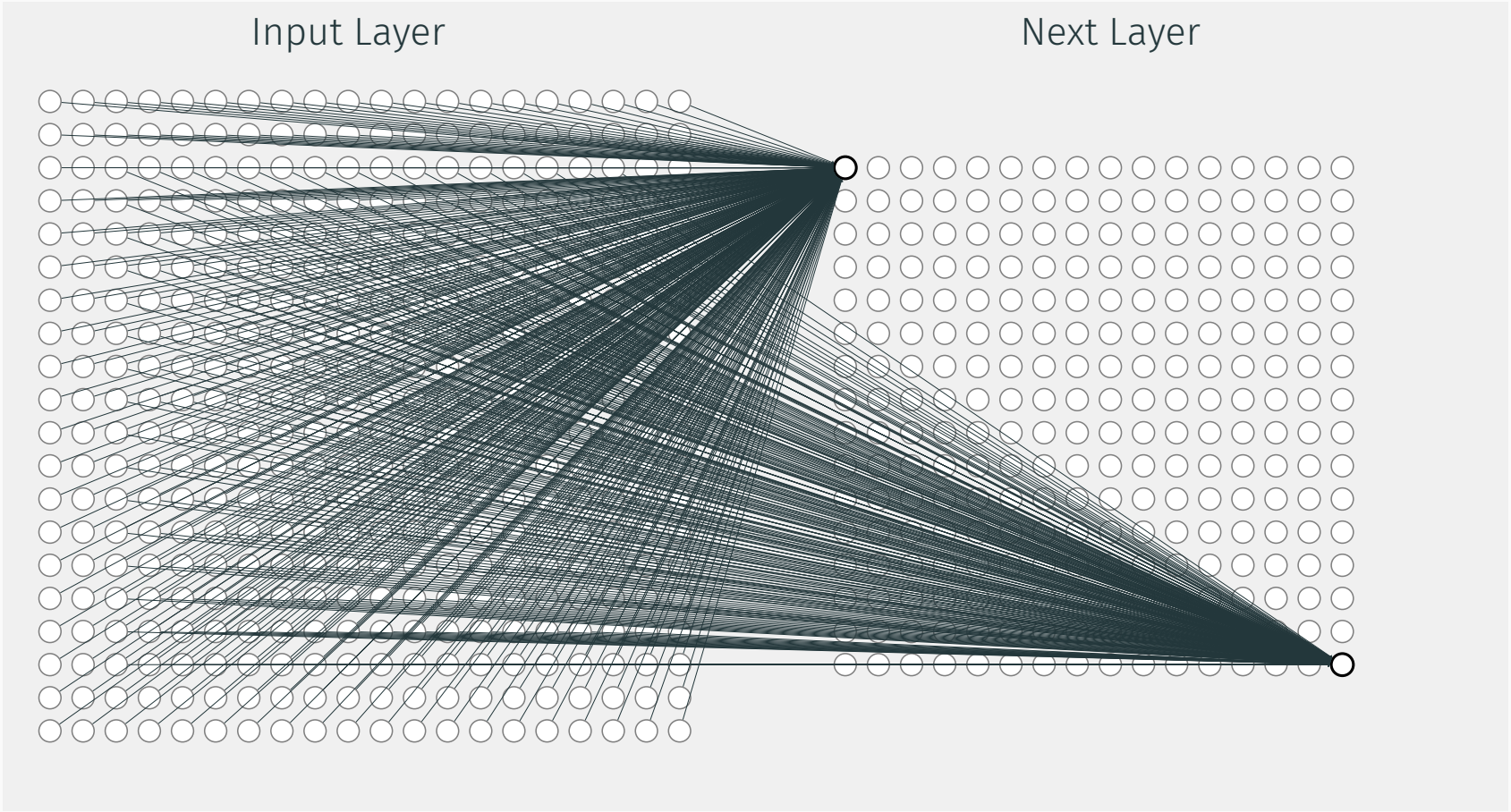
For an input image of dimension `width` by `height` pixels and 3 colour channels, the input layer will contain $3 \times$ `height` $\times$ `width` input units.

If the next layer is of the same size, then we have up to $(3 \times$ `height` $\times$ `width`$)^2$ weights to train, which can become very large very quickly.

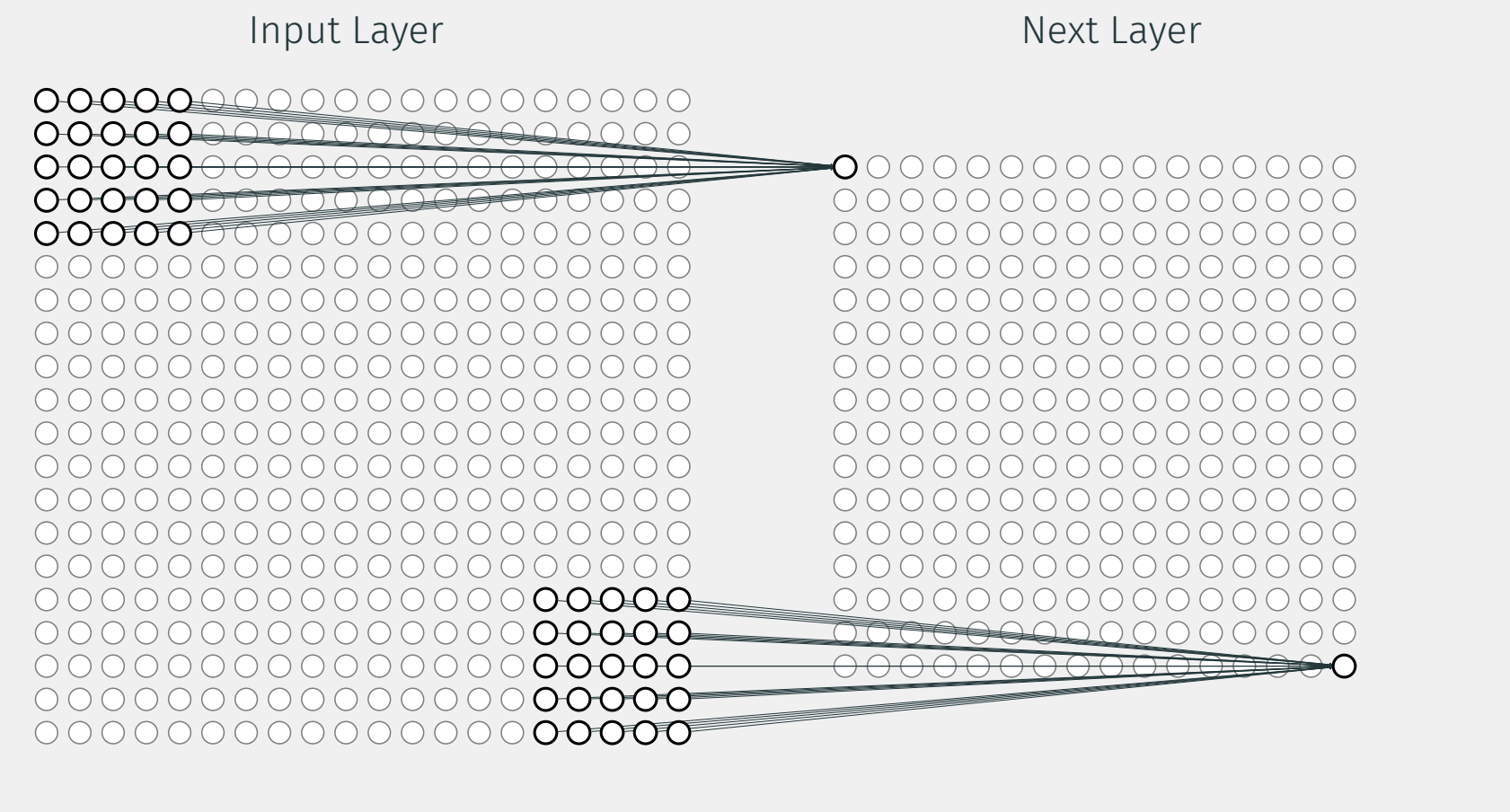
In a fully connected approach, we don't take advantage of the spatial structure of an image.

For instance, we know that pixel values are usually more related to their neighbours than to far away locations. We need to take advantage of this.

This is what is done in Convolutional Neural Networks.



The input layer is made of a grid of the image pixels. In a fully connected layer architecture, pixels in the next layer are connected to all of the pixels in the input layer.



In convolutional layers, the units in the next layer are only connected to their neighbours in the input layer. In this case the neighbourhood is defined as a 5×5 window.

Moreover the weights are shared across all the pixels.

So, in convnets, the weights are associated to the relative positions of the neighbours and shared across all pixel locations. Let us see how they are defined.

Denote the units of a layer as $u_{i,j,k,n}$, where n refers to the layer, i, j to the coordinates of the pixel and k to the channel of consideration.

The logit for that neuron is defined as the result of a convolution filter:

$$\text{logit}_{i,j,k,n} = w_{0,k,n} + \sum_{a=-h_1}^{h_1} \sum_{b=-h_2}^{h_2} \sum_{c=1}^{h_3} w_{a,b,c,k,n} u_{a+i,b+j,c,n-1}$$

where h_1 and h_2 correspond to half of the dimensions of the neighbourhood window and $h3$ is the number of channels of the input image for that layer.

After activation f, the output of the neuron is simply:

$$u_{i,j,k,n} = f\left(\text{logit}_{i,j,k,n}\right)$$

Example

Consider the case of a grayscale image (1 channel) where the convolution is defined as:

$$\text{logit}_{i,j,n} = u_{i+1,j,n-1} + u_{i-1,j,n-1} + u_{i,j+1,n-1} + u_{i,j-1,n-1} - 4u_{i,j,n-1}$$

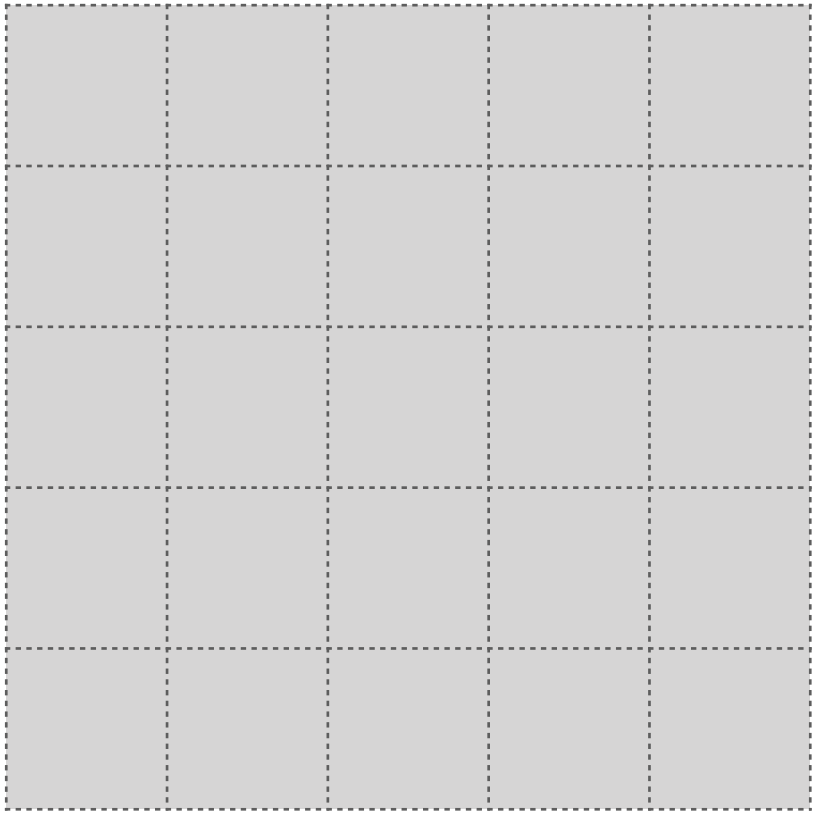
The weights can be arranged as a weight **mask** (also called **kernel**):

w:

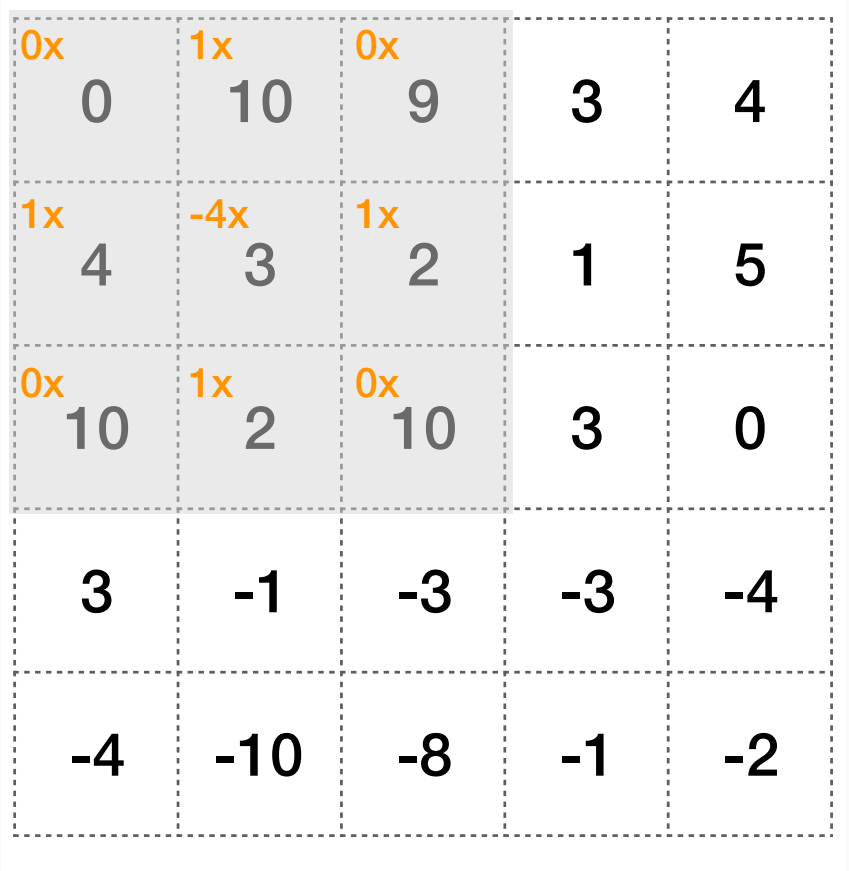
0	1	0
1	-4	1
0	1	0

0	10	9	3	4
4	3	2	1	5
10	2	10	3	0
3	-1	-3	-3	-4
-4	-10	-8	-1	-2

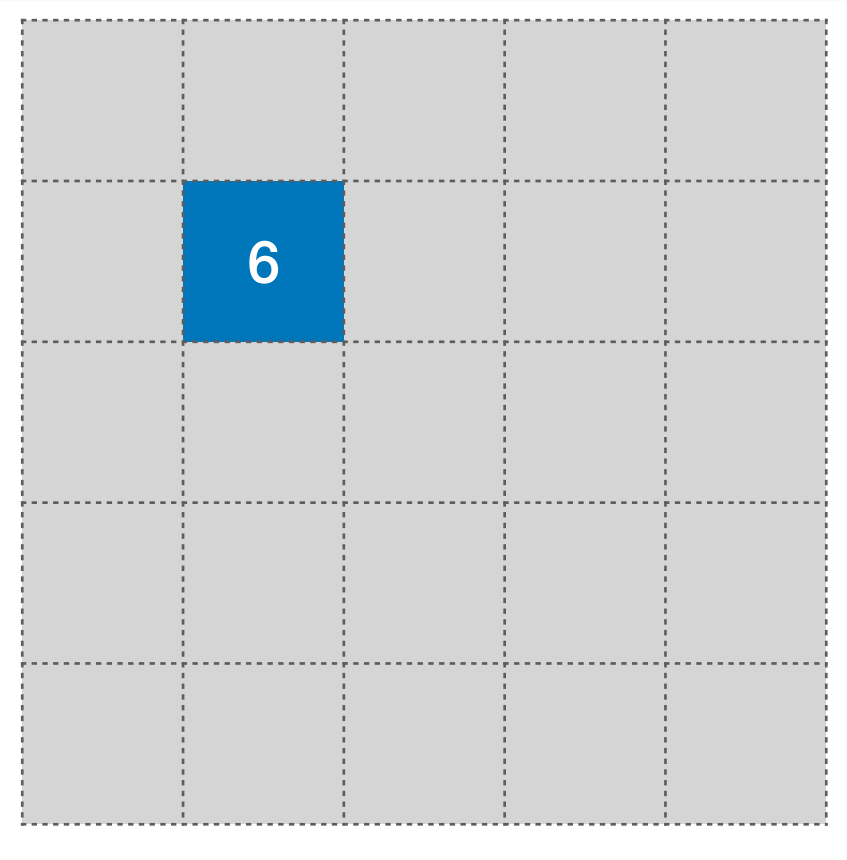
pixel values in previous layer



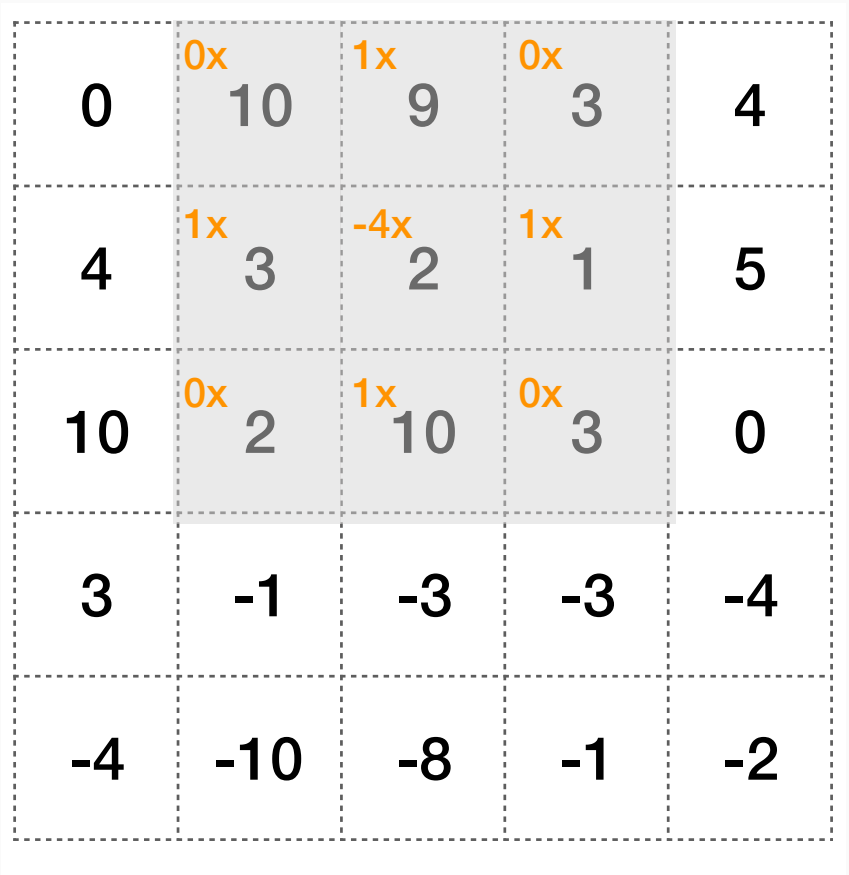
logits after convolution



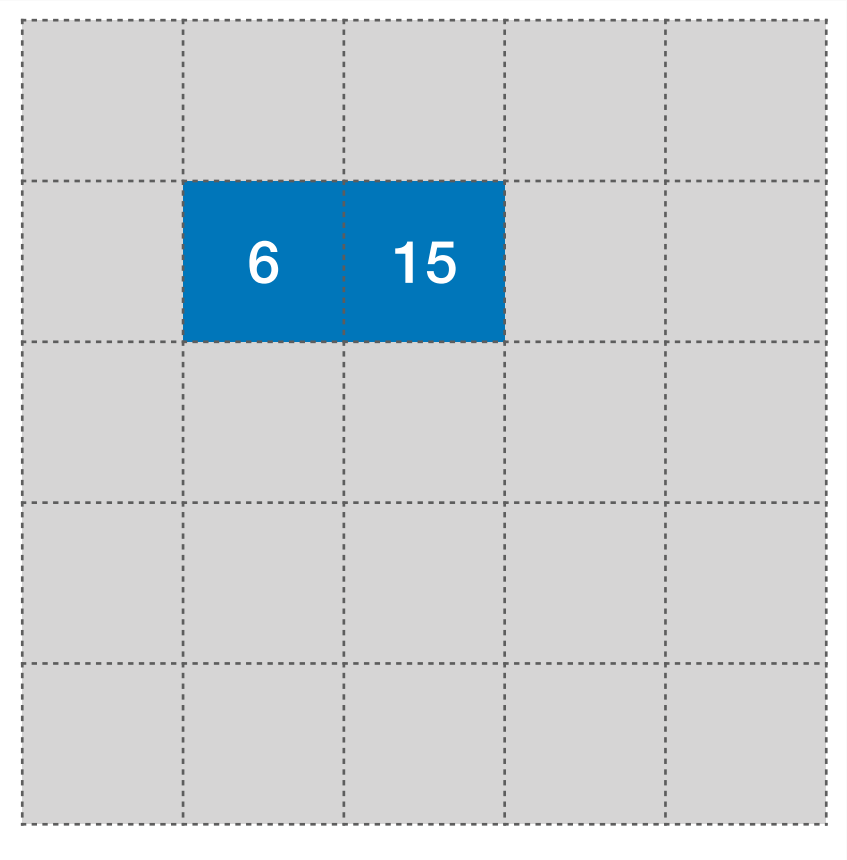
pixel values in previous layer



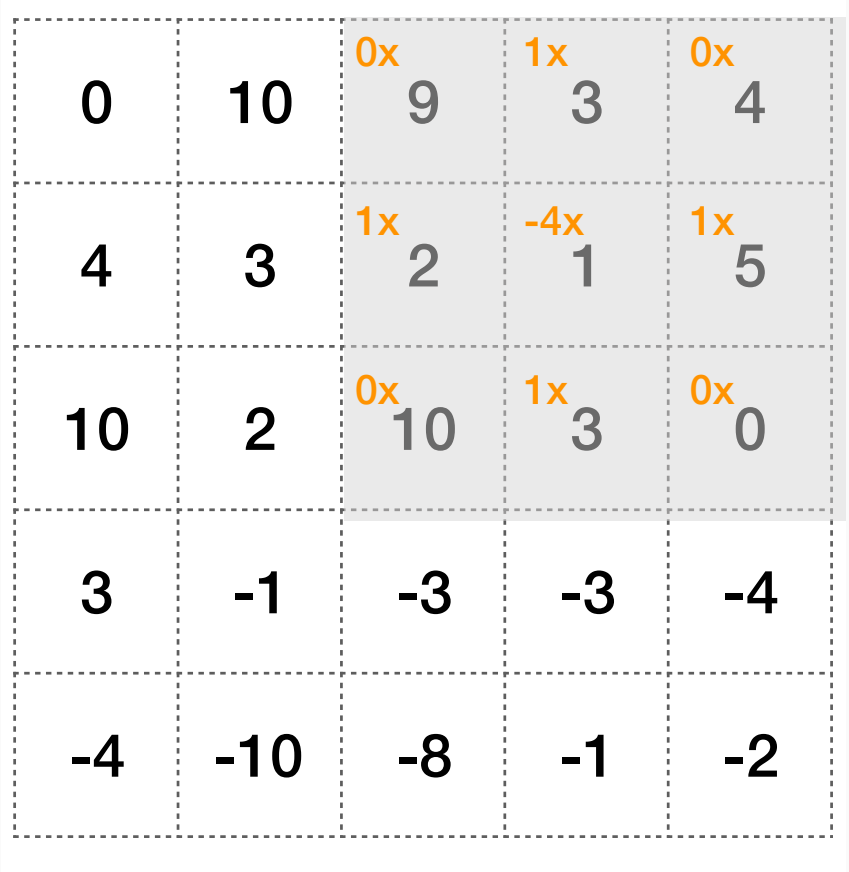
logits after convolution



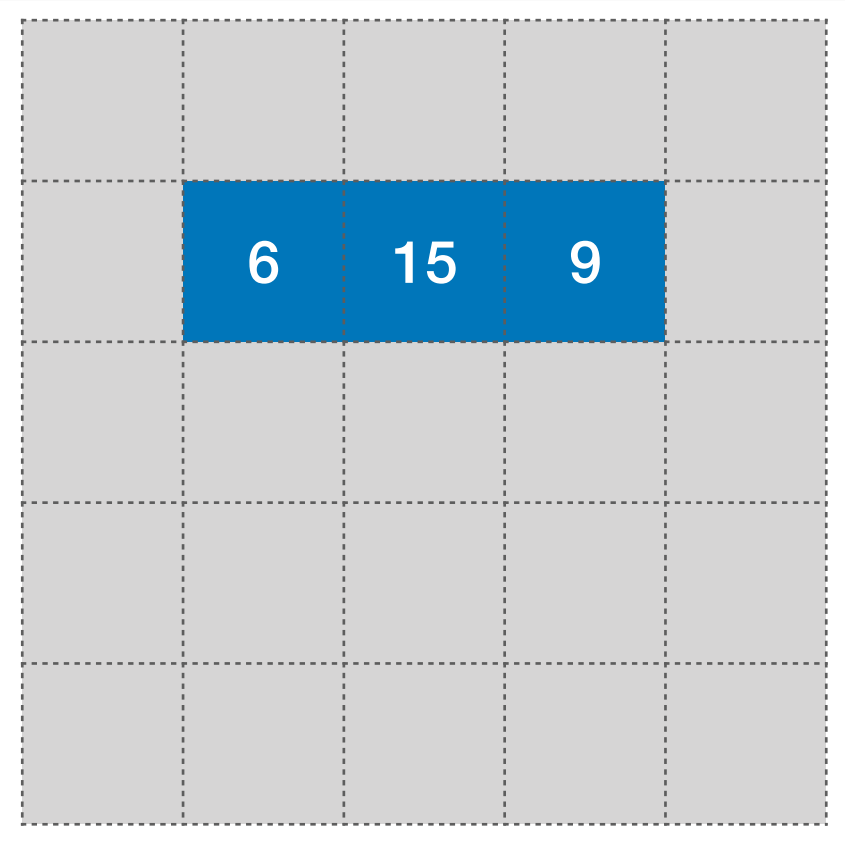
pixel values in previous layer



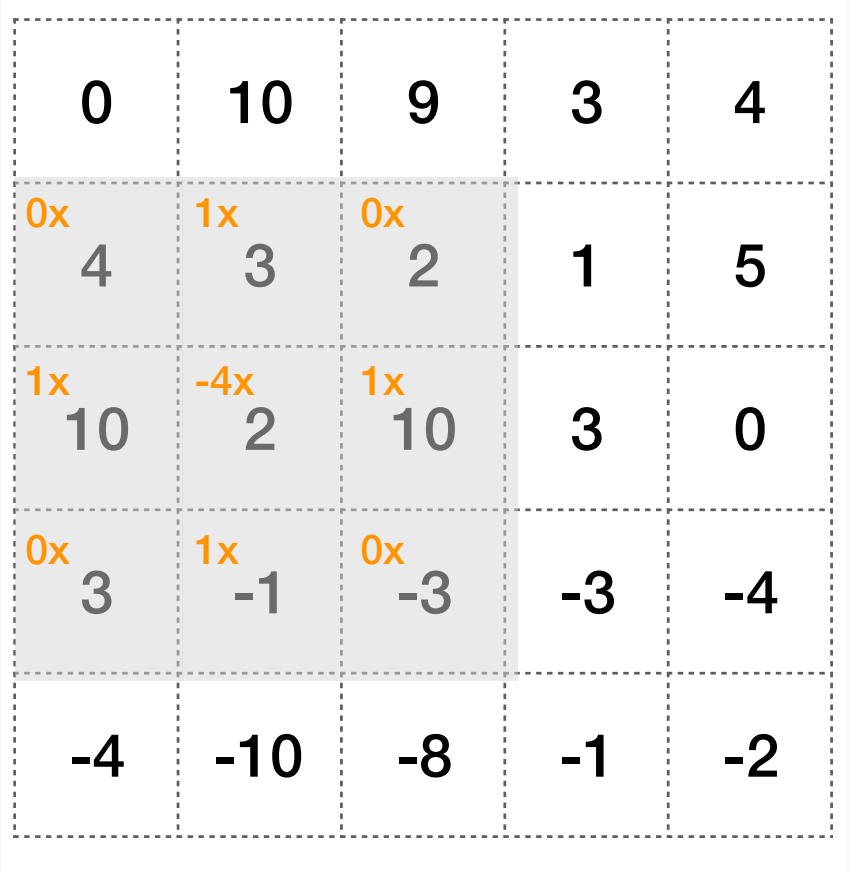
logits after convolution



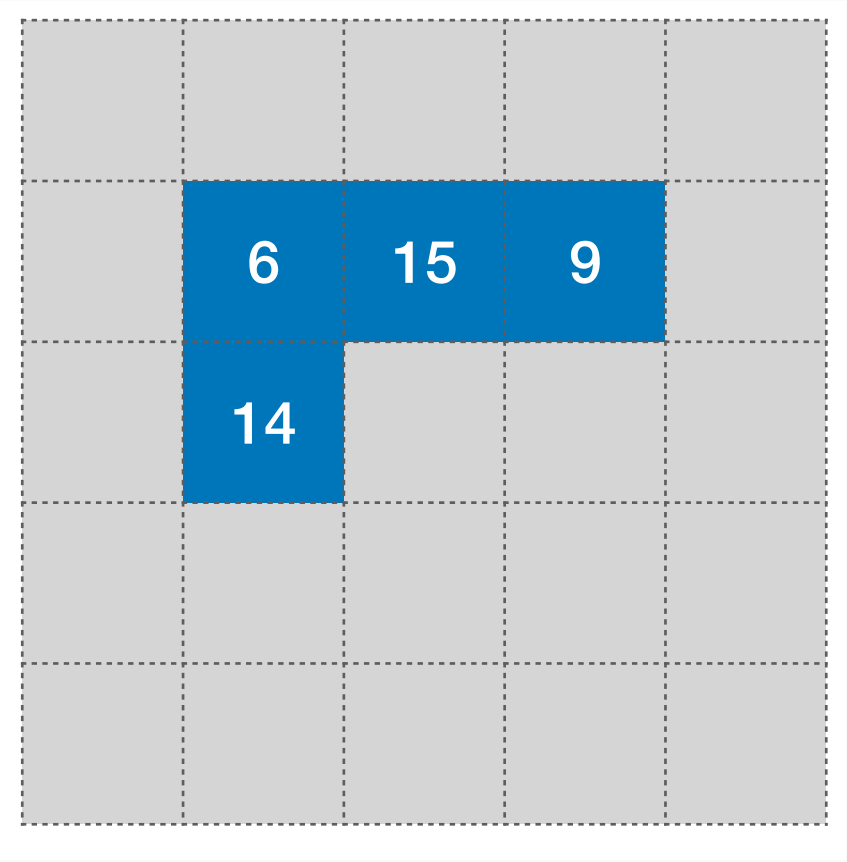
pixel values in previous layer



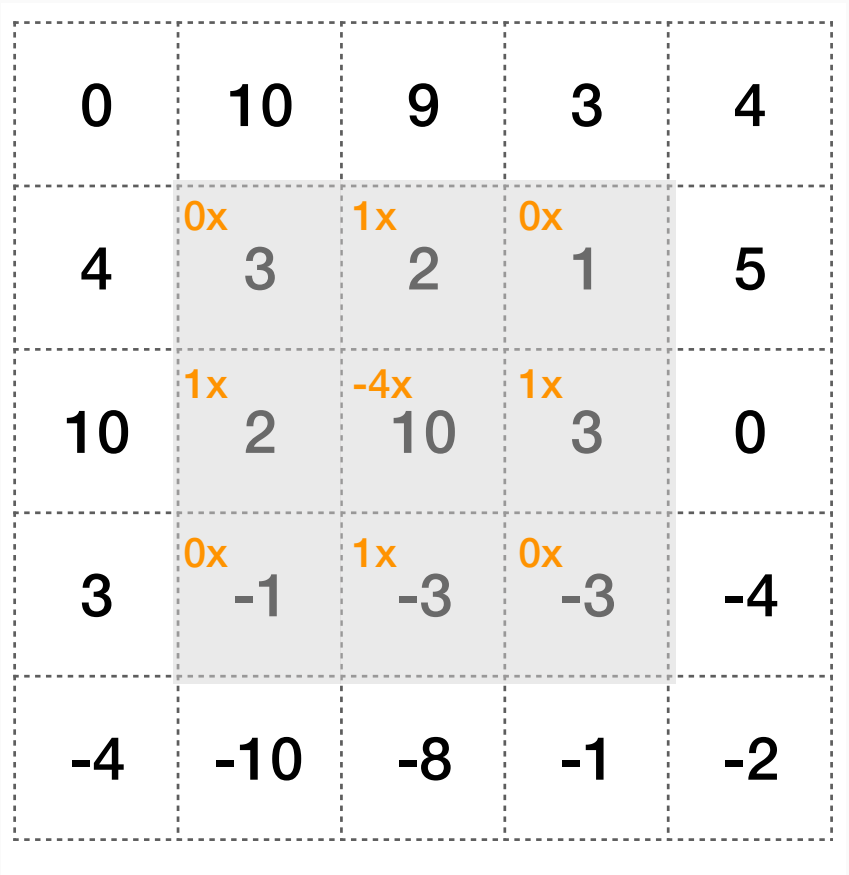
logits after convolution



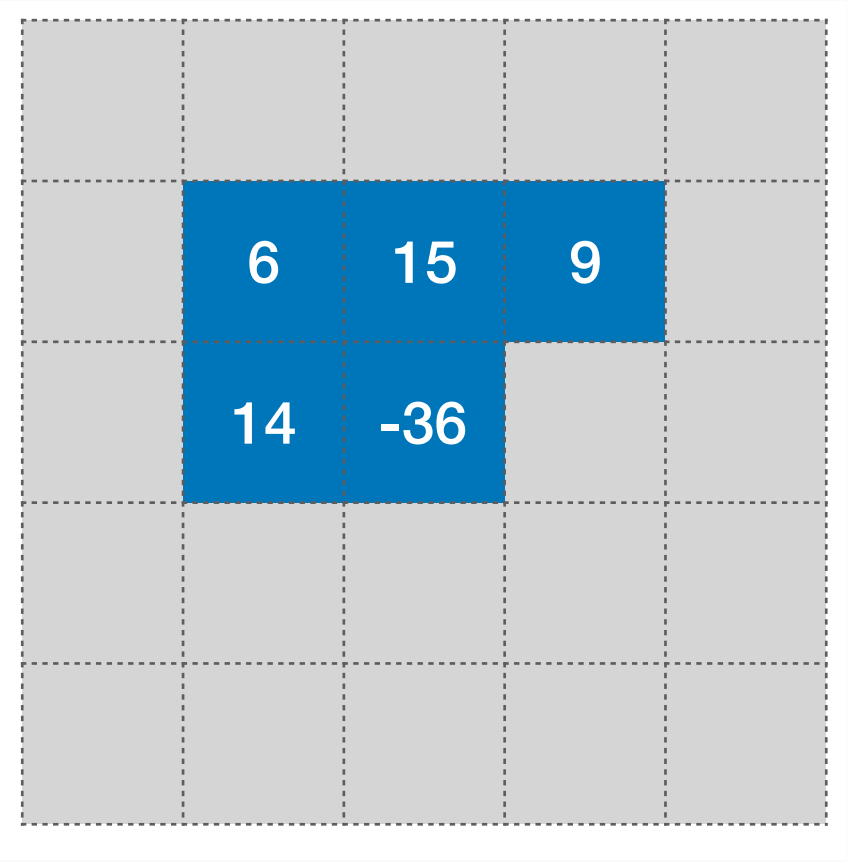
pixel values in previous layer



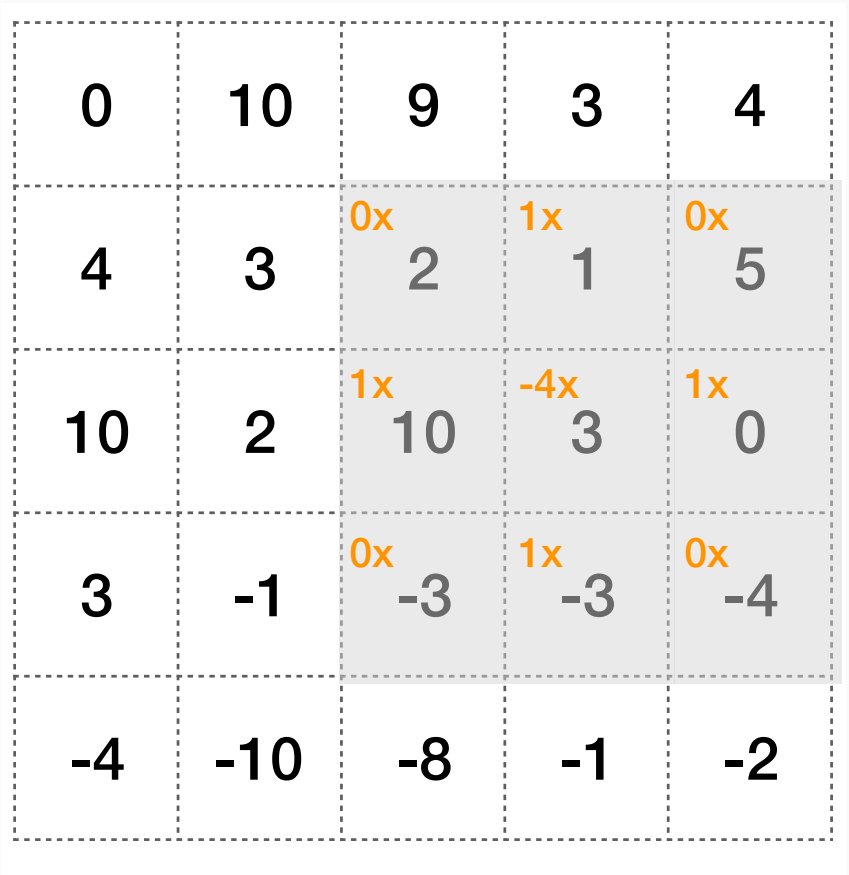
logits after convolution



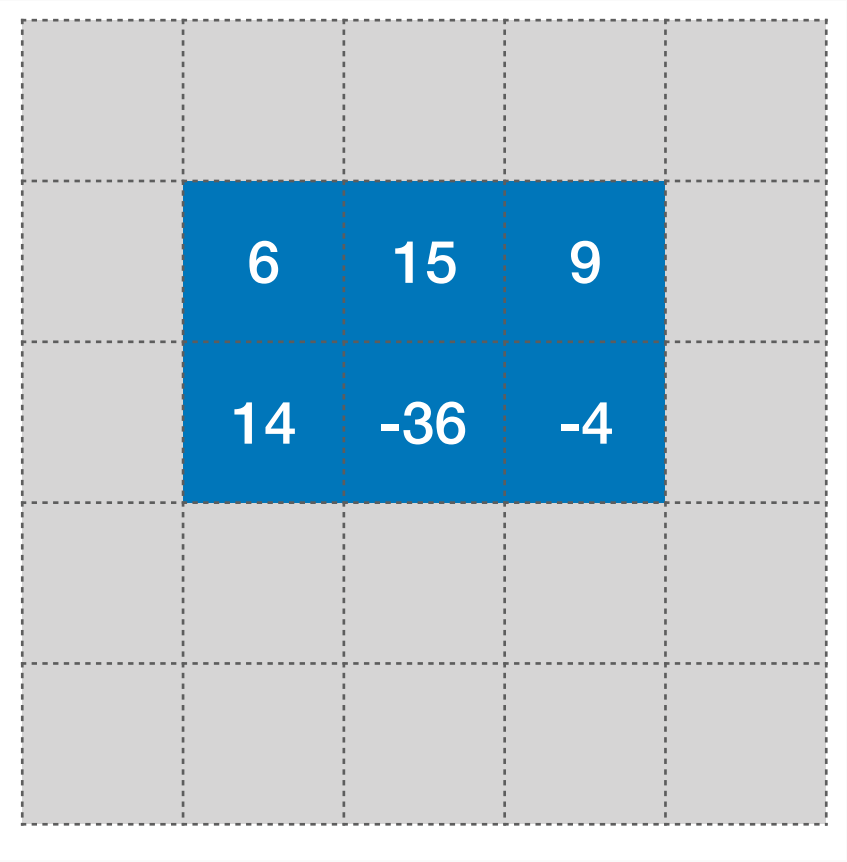
pixel values in previous layer



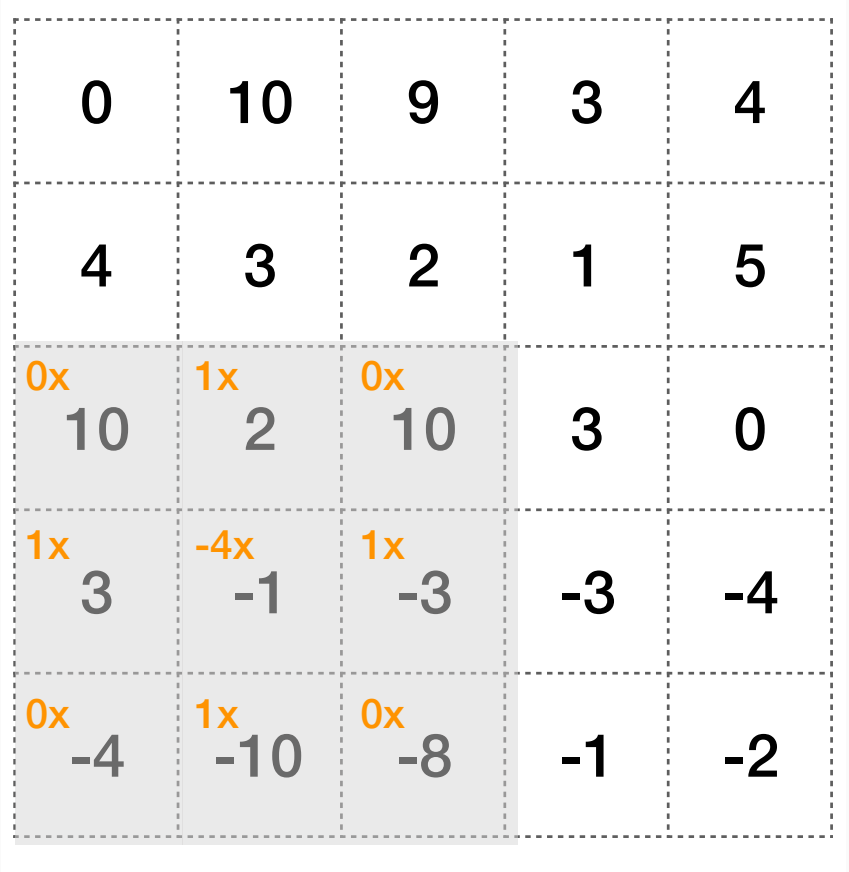
logits after convolution



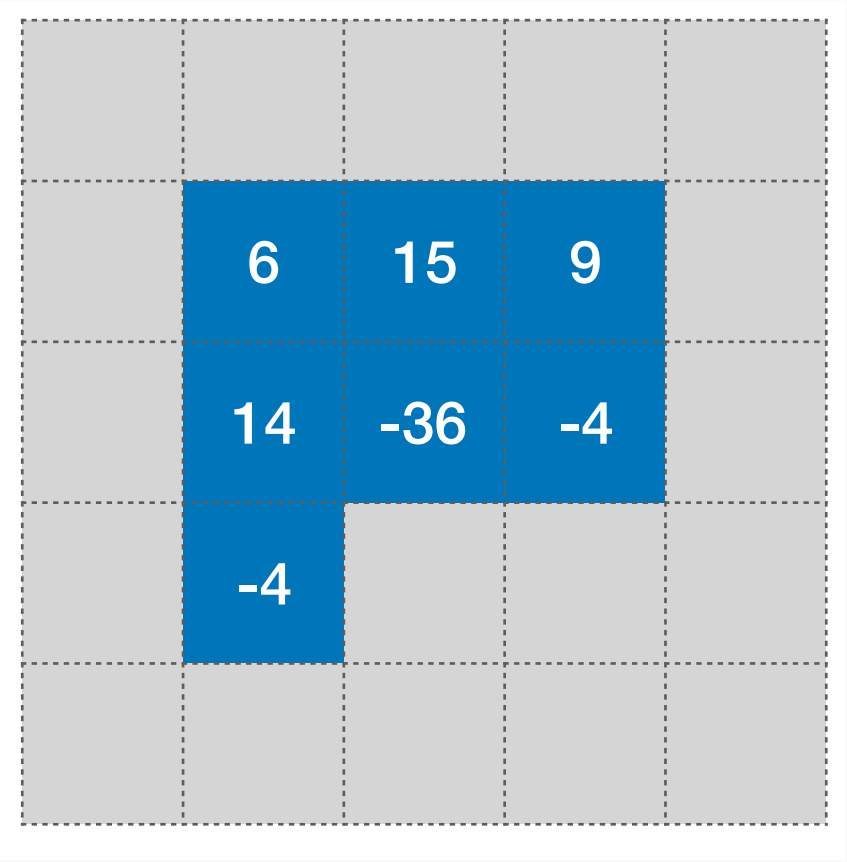
pixel values in previous layer



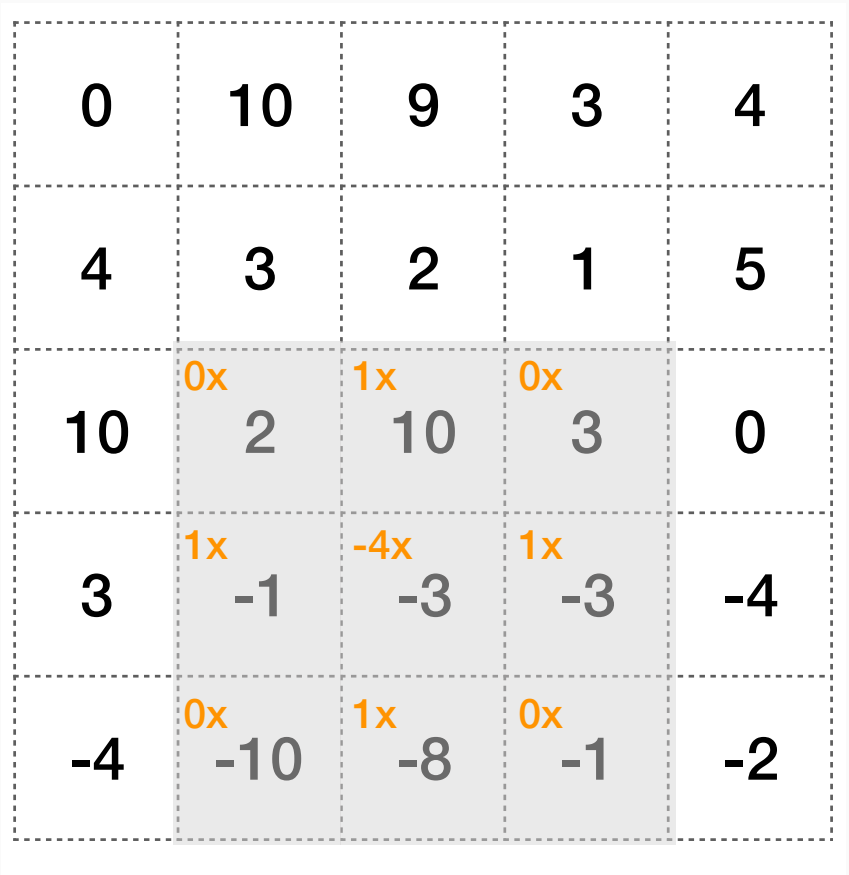
logits after convolution



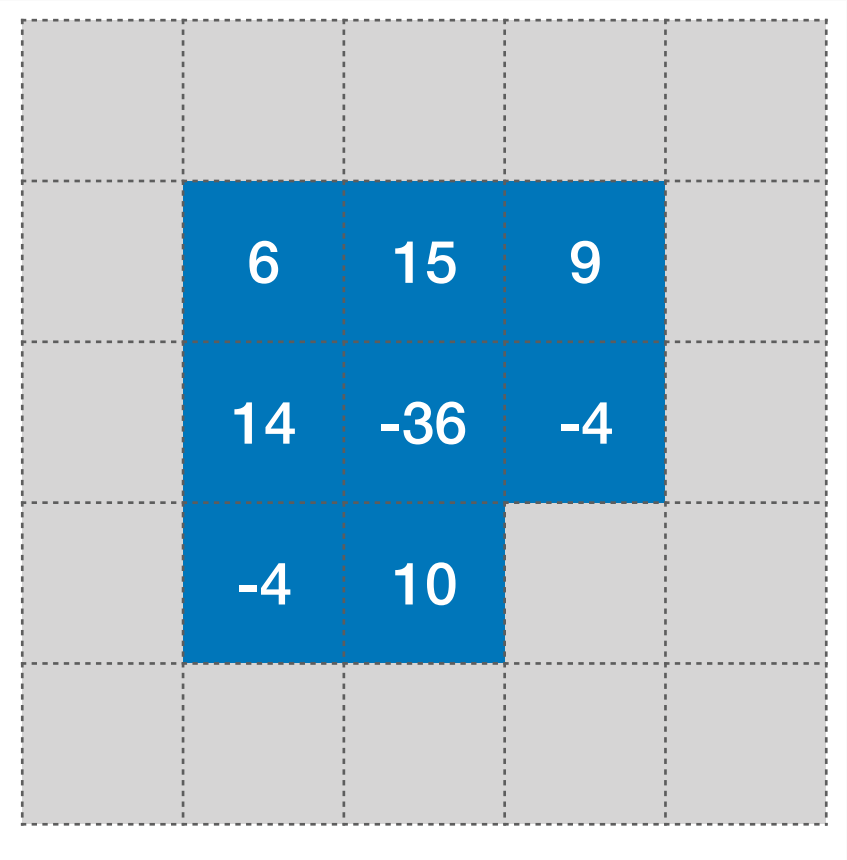
pixel values in previous layer



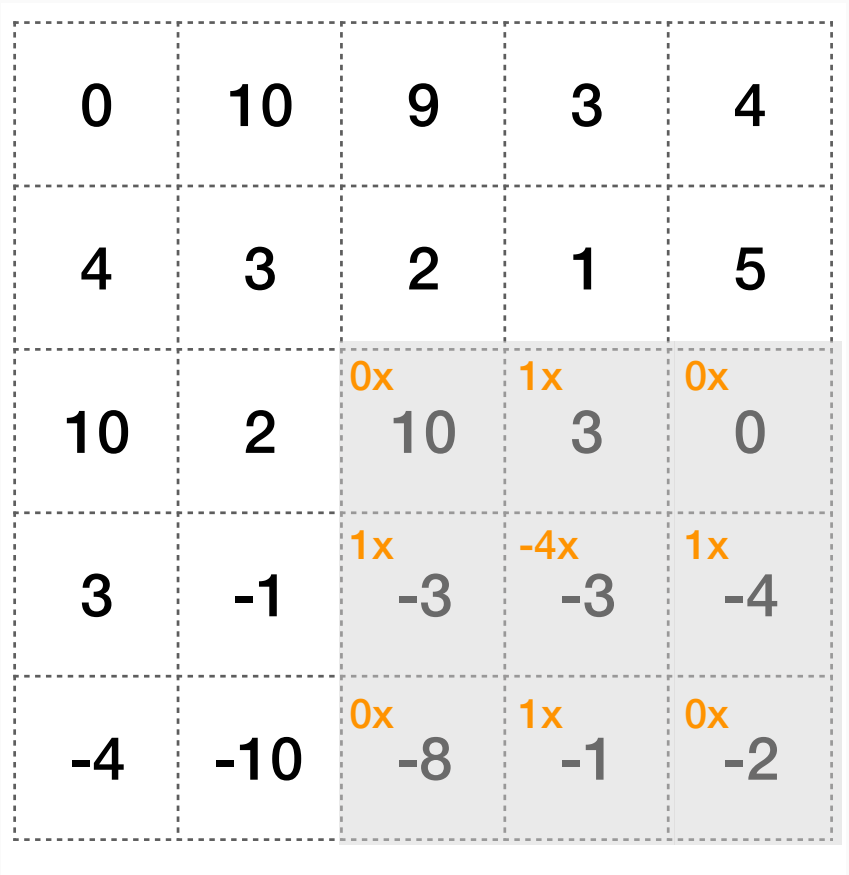
logits after convolution



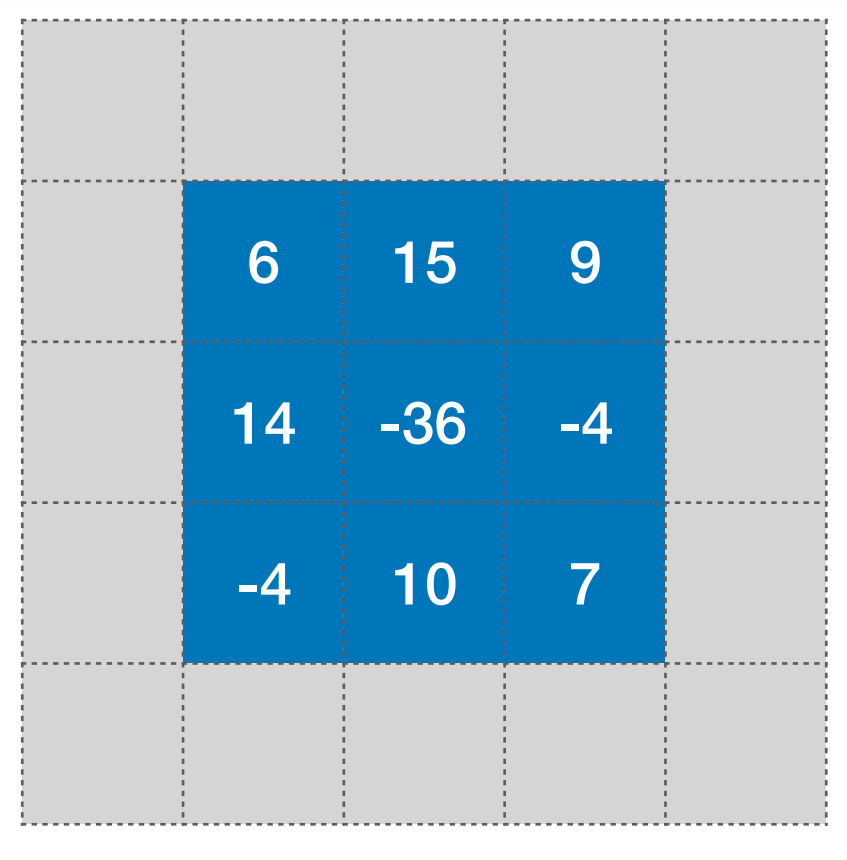
pixel values in previous layer



logits after convolution



pixel values in previous layer



logits after convolution

Padding

At the picture boundaries, not all neighbours are defined (see figure in next slide).

In PyTorch two main padding strategies are possible:

`padding='same'` means that the values outside of image domain are extrapolated to zero.

`padding='valid'` means that we don't compute the pixels that need neighbours outside of the image domain. This means that the picture is slightly cropped.

Padding

0x ?	1x ?	0x ?	?	?	?	?
1x ?	-4x 0	1x 10	9	3	4	?
0x ?	1x 4	0x 3	2	1	5	?
?	10	2	10	3	0	?
?	3	-1	-3	-3	-4	?
?	-4	-10	-8	-1	-2	?
?	?	?	?	?	?	?

input layer. Pixels outside the image domain are marked with '?'.

?	?	?	?	?
?	6	15	9	?
?	14	-36	-4	?
?	-4	10	7	?
?	?	?	?	?

after 3×3 convolution. Boundary pixels require out of domain neighbours.

Each convolutional layer defines a number of convolution filters and the output of a layer is thus a new image, where each channel is the result of a convolution filter followed by activation.

Example

On next slide is a colour picture of size $3 \times 592 \times 443$ (width=443, height=592, number of channels=3). The input is thus a multidimensional array, or **tensor**. PyTorch uses NCHW convention for ordering the dimensions: we start with N is the batch size, C the number of channels, H the height, and W the width.

The convolutional layer used has a kernel of size 5×5, and produces 6 different filters. The padding strategy is set to `'valid'` thus we loose 2 pixels on each side. The output of the convolutional layer is a picture of size $6 \times 588 \times 439$. In PyTorch, this could be defined as:

```
import torch
import torch.nn as nn
x = torch.randn(1, 3, 592, 443) # batchsize=1; channels=3; h=592, w=443
conv = nn.Conv2d(in_channels=3, out_channels=6, kernel_size=5)
```

This convolution layer is defined by $3 \times 6 \times 5 \times 5 = 450$ weights. This is only a fraction of what would be required in a dense layer.

input layer = red channel green channel blue channel

output filtered images:

Reducing the tensor size

If convolution filters offer a way of reducing the number of weights in the network, the number of units still remains high.

For instance, applying `Conv2d(3, 16, kernel_size=5)` to an input tensor image of size $3 \times 2000 \times 2000$ only requires $5 \times 5 \times 3 \times 16 = 1200$ weights to train, but still produces $2000 \times 2000 \times 16 = 64$ million units.

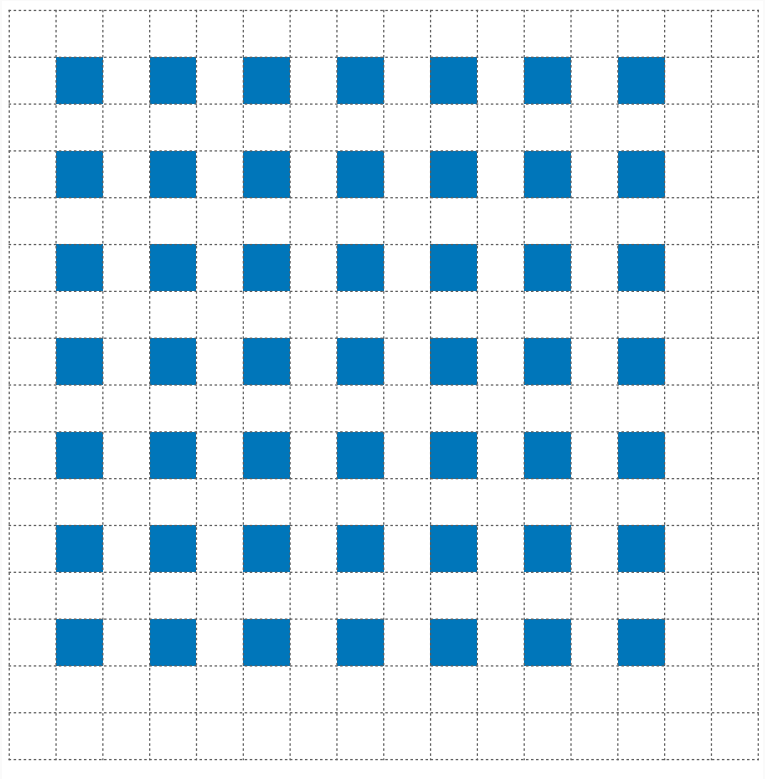
In this section, we'll see how stride and pooling can be used to down-sample the images and thus reduce the number of units.

Stride

In image processing, the **stride** is the distance that separates each processed pixel. A stride of 1 means that all pixels are processed and kept. A stride of 2 means that only every second pixel in both x and y directions are kept.

Stride

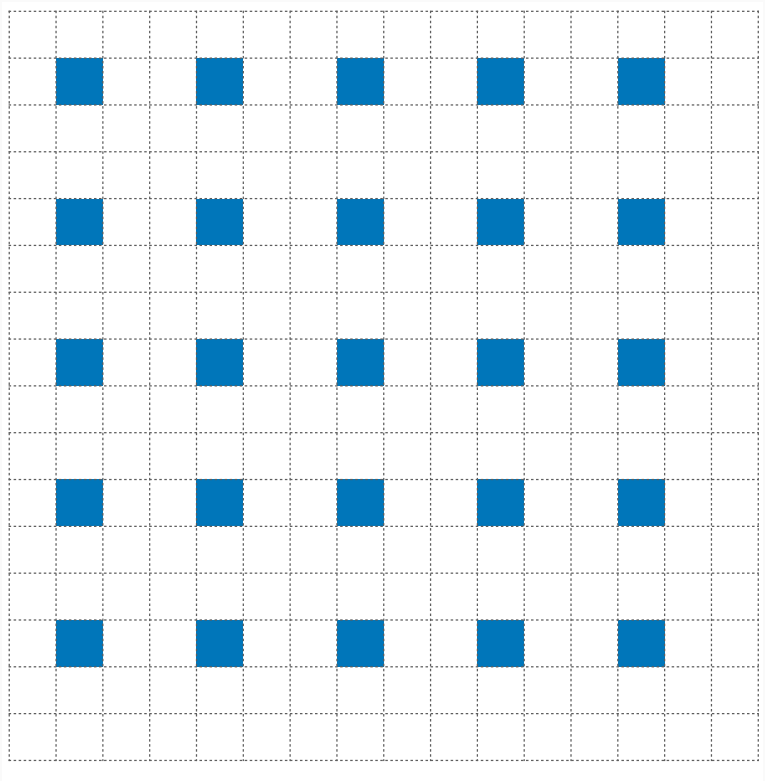
```
conv = nn.Conv2d(1, 1, kernel_size=3, stride=2)
x = torch.randn(1, 1, 16, 16)
x = conv(x)
print(x.shape) # torch.Size([1, 1, 7, 7])
```



Example for a stride of 2. Previous layer is 16×16. Convolution mask is 3×3. Convolution is only computed for one in four pixels (marked in blue). Output is of size 7×7.

Stride

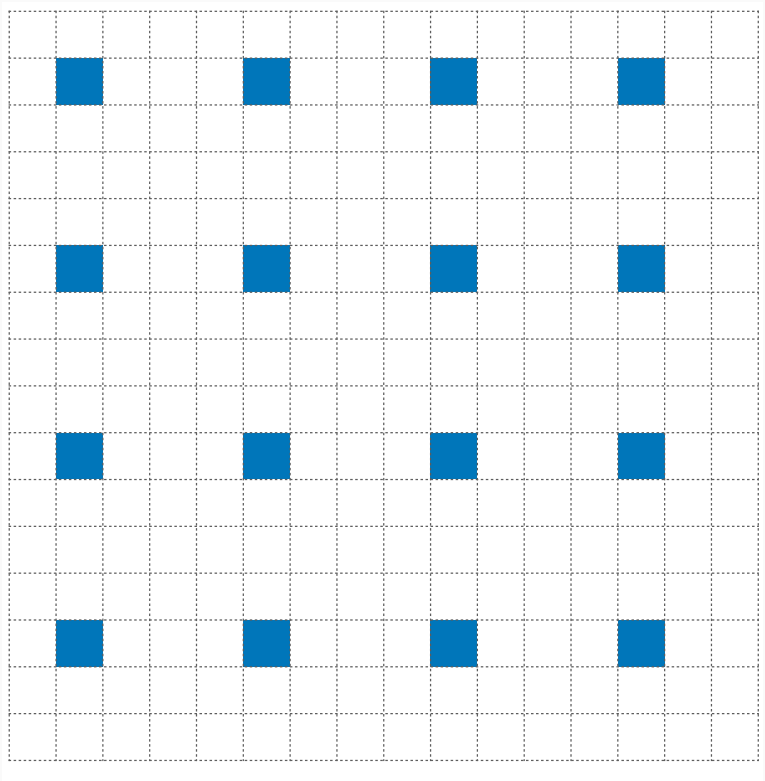
```
conv = nn.Conv2d(1, 1, kernel_size=3, stride=3)
x = torch.randn(1, 1, 16, 16)
x = conv(x)
print(x.shape) # torch.Size([1, 1, 5, 5])
```



Example for a stride of 3. Only 1 in 9 pixels are kept.

Stride

```
conv = nn.Conv2d(1, 1, kernel_size=3, stride=4)
x = torch.randn(1, 1, 16, 16)
x = conv(x)
print(x.shape) # torch.Size([1, 1, 4, 4])
```



Example for a stride of 4. Only 1 in 16 pixels are kept.

Max Pooling

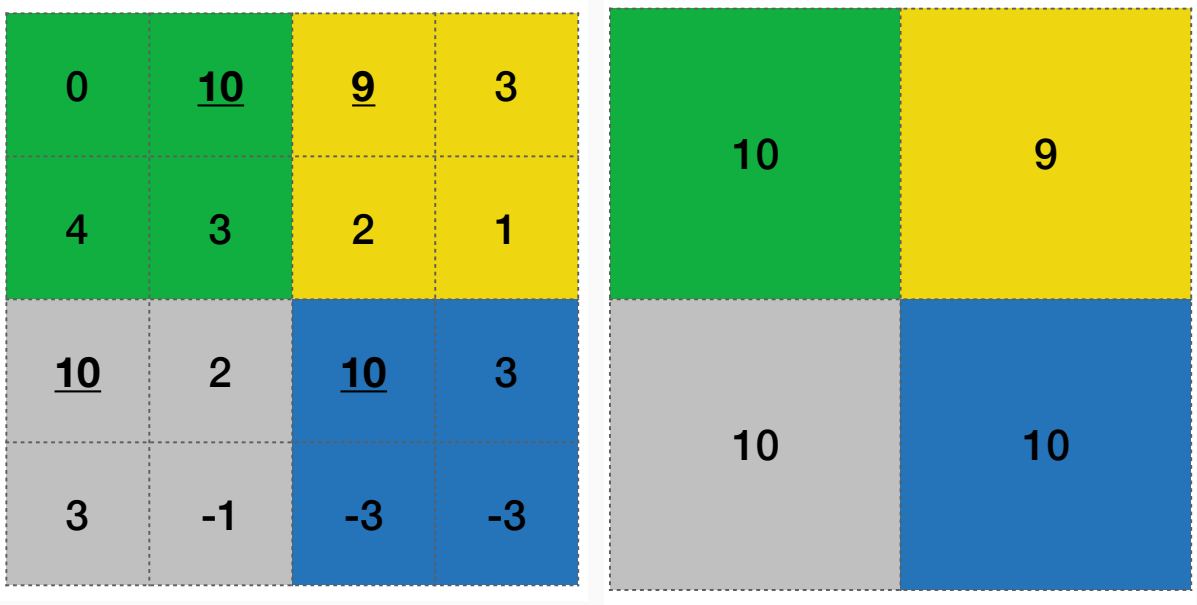
Whereas stride is set on the convolution layer itself, Pooling is a separate node that is appended after the conv layer. The Pooling layer operates a sub-sampling of the picture.

Different sub-sampling strategies are possible: average pooling, max pooling, stochastic pooling.

Max Pooling

```
torch.nn.MaxPool2d(kernel_size=2, stride=2)
```

The maximum of each block is kept.



Example

```
import torch
import torch.nn as nn

x = torch.randn(1, 3, 32, 32)
conv = nn.Conv2d(3, 16, 5, padding='same')
x = nn.functional.relu(conv(x))
print(x.shape) # torch.Size([1, 16, 32, 32])

pool = nn.MaxPool2d(kernel_size=2, stride=2)
x = pool(x)
print(x.shape) # torch.Size([1, 16, 16, 16])
```

This PyTorch code transforms the original image of size $3 \times 32 \times 32$ into a new image of size $16 \times 32 \times 32$. Each of the 16 output image channels are obtained through their own $3 \times 5 \times 5$ convolution filter.

Then maxpooling reduces the image size to $16 \times 16 \times 16$.

Increasing the tensor size

Transposed Convolution

Similarly we can increase the horizontal and vertical dimensions of a tensor using an upsampling operation. This step is sometimes called up-convolution, deconvolution or transposed convolution.

This step is equivalent to first upsampling the tensor by inserting zeros in-between the input samples and then applying a convolution layer (see link).

In PyTorch:

```
import torch
import torch.nn as nn

x = torch.randn(4, 128, 10, 8)
nfilters = 32; kernel_size = (3,3); stride = (2, 2)
deconv = nn.ConvTranspose2d(128, nfilters, kernel_size, stride)
y = deconv(x)

print(y.shape) # torch.Size([4, 32, 21, 17])
```

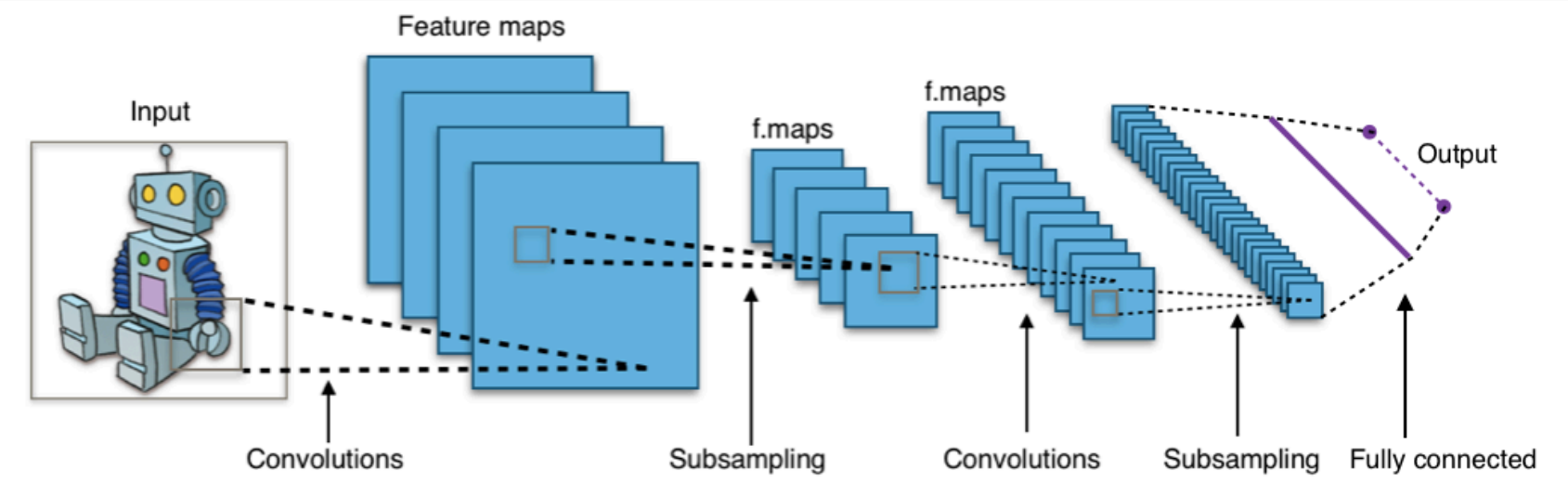
Transposed Convolution

Just note that Deconvolution is an unfortunate term for this step as deconvolution is already used in signal processing and refers to trying to estimate the input signal/tensor from the output signal. (eg. trying to recover the original image from an blurred image).

Architecture Design

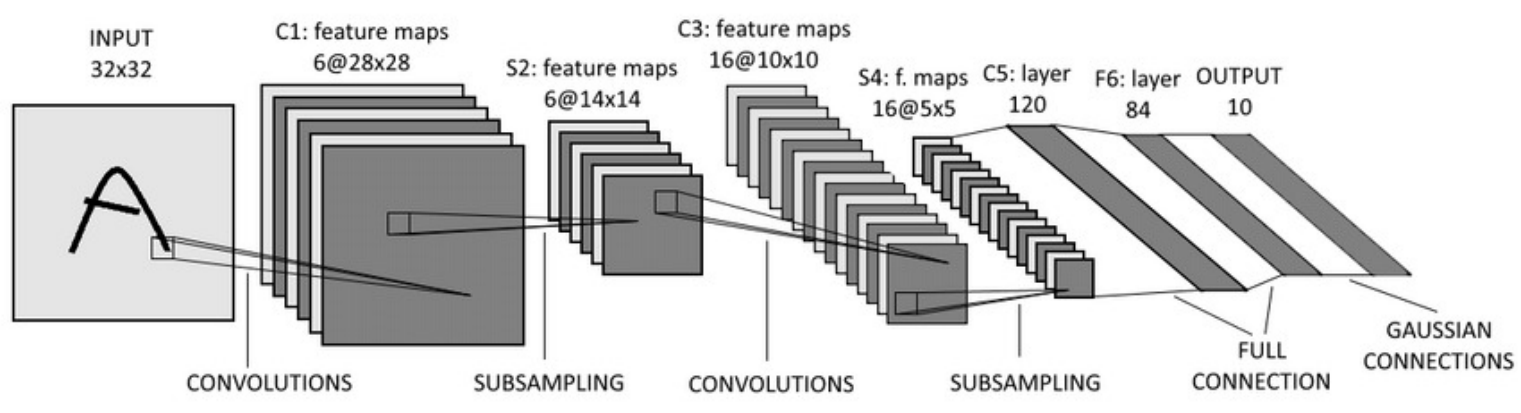
A typical convnet architecture for classification is based on interleaving convolution layers with pooling layers. Conv layers usually have a small kernel size (eg. 5×5 or 3×3). As you go deeper, the picture becomes smaller in resolution but also contains more channels.

At some point the picture is so small (eg. 7×7), that it doesn't make sense to call it a picture. You can then connect it to fully connected layers and terminate by a last softmax layer for classification:



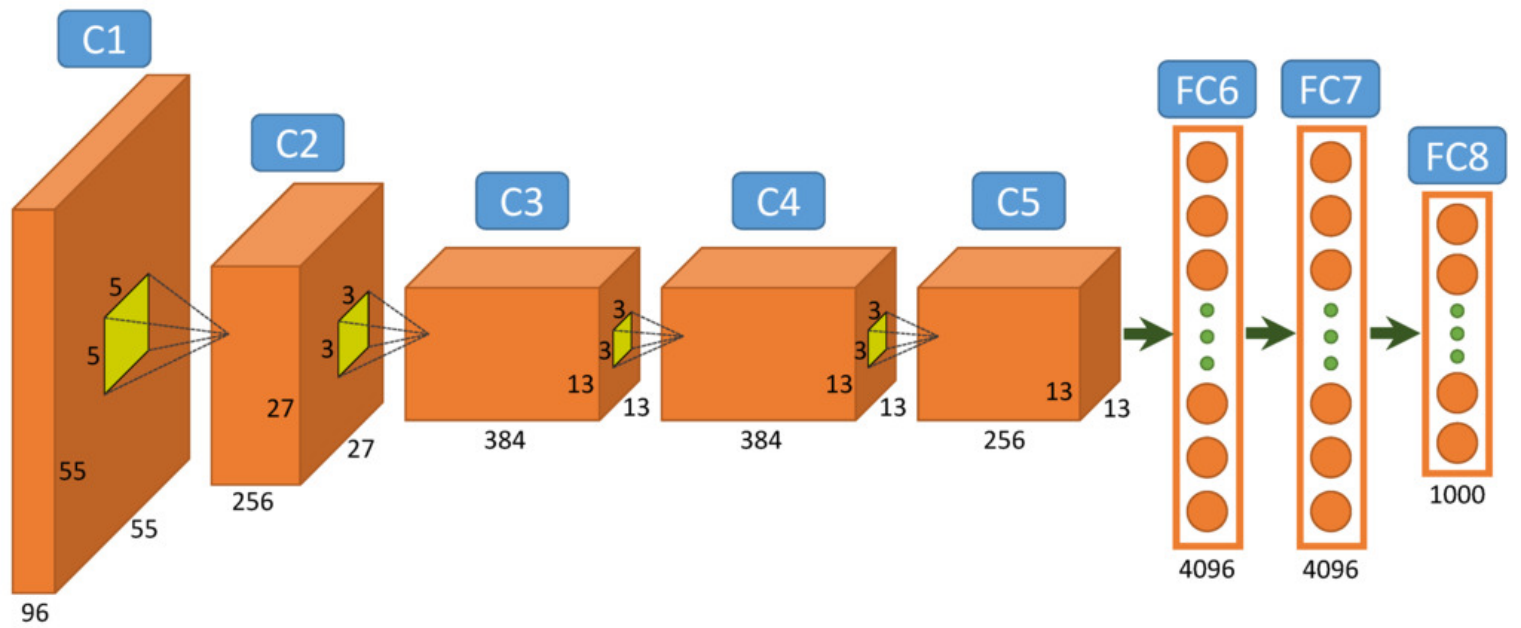
The idea is that we start from a few low level features (eg. image edges) and as we go deeper, we built more and more features that are increasingly more complex.

The following slides show some of the landmark networks.



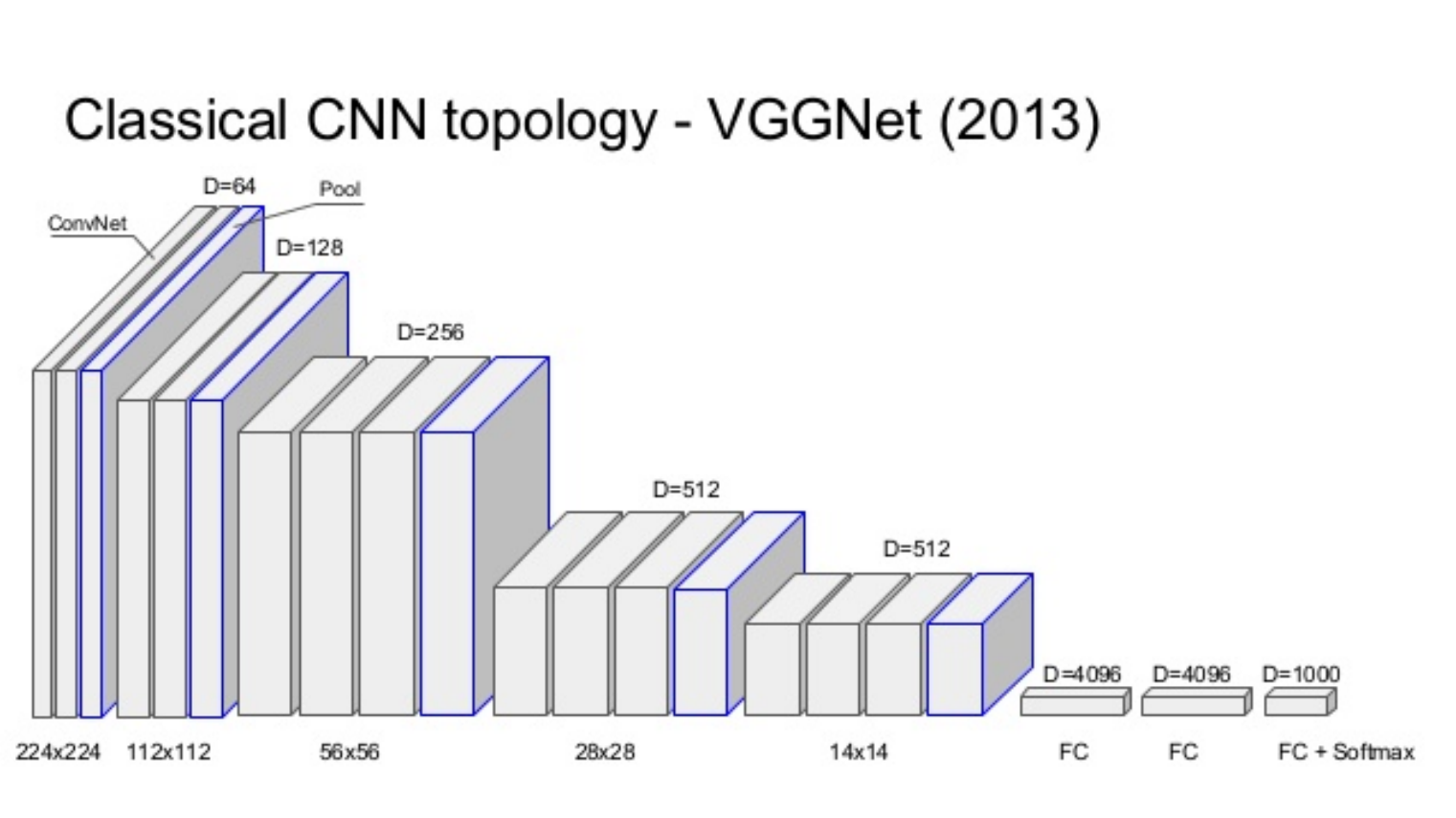
LeNet-5 (LeCun, 1998). The network pioneered the use of convolutional layers in neural nets.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998).
Gradient-based learning applied to document recognition.



AlexNet (Alex Krizhevsky et al., 2012). This is the winning entry of the ILSVRC-2012 competition for object recognition. This is the network that started the deep learning revolution.

> Alex Krizhevsky, Ilya Sutskever, Geoffrey E Hinton (2012)
> Imagenet classification with deep convolutional neural networks.



VGG (Simonyan and Zisserman, 2013). This is a popular 16-layer network used by the VGG team in the ILSVRC-2014 competition for object recognition.

> K. Simonyan, A. Zisserman
> Very Deep Convolutional Networks for Large-Scale Image Recognition

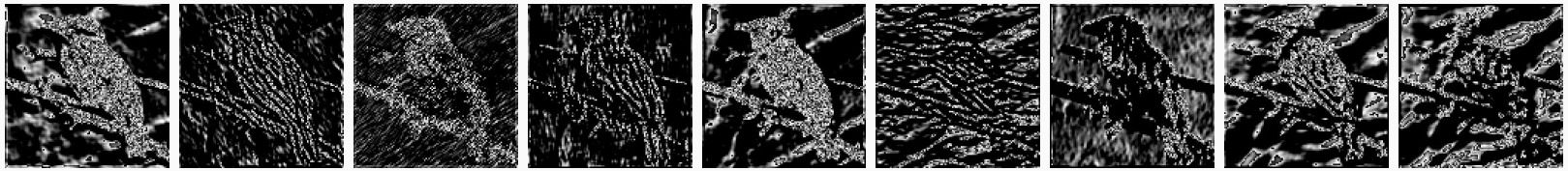
A PyTorch implementation for VGG16:

```
import torch
import torch.nn as nn

class VGG16(nn.Module):
  def __init__(self, num_classes=1000):
    super(VGG16, self).__init__()
    self.features = nn.Sequential(
      # Block 1
      nn.Conv2d(3, 64, kernel_size=3, padding=1),
      nn.ReLU(inplace=True),
      nn.Conv2d(64, 64, kernel_size=3, padding=1),
      nn.ReLU(inplace=True),
      nn.MaxPool2d(kernel_size=2, stride=2),
      # Block 2
      nn.Conv2d(64, 128, kernel_size=3, padding=1),
      nn.ReLU(inplace=True),
      nn.Conv2d(128, 128, kernel_size=3, padding=1),
      nn.ReLU(inplace=True),
      nn.MaxPool2d(kernel_size=2, stride=2),
      # Block 3
      nn.Conv2d(128, 256, kernel_size=3, padding=1),
      nn.ReLU(inplace=True),
      nn.Conv2d(256, 256, kernel_size=3, padding=1),
      nn.ReLU(inplace=True),
      nn.Conv2d(256, 256, kernel_size=3, padding=1),
      nn.ReLU(inplace=True),
      nn.MaxPool2d(kernel_size=2, stride=2),
      # Block 4
      nn.Conv2d(256, 512, kernel_size=3, padding=1),
      nn.ReLU(inplace=True),
      nn.Conv2d(512, 512, kernel_size=3, padding=1),
      nn.ReLU(inplace=True),
      nn.Conv2d(512, 512, kernel_size=3, padding=1),
      nn.ReLU(inplace=True),
      nn.MaxPool2d(kernel_size=2, stride=2),
      # Block 5
      nn.Conv2d(512, 512, kernel_size=3, padding=1),
      nn.ReLU(inplace=True),
      nn.Conv2d(512, 512, kernel_size=3, padding=1),
      nn.ReLU(inplace=True),
      nn.Conv2d(512, 512, kernel_size=3, padding=1),
      nn.ReLU(inplace=True),
      nn.MaxPool2d(kernel_size=2, stride=2),
    )
    self.avgpool = nn.AdaptiveAvgPool2d((7, 7))
    self.classifier = nn.Sequential(
      nn.Linear(512 * 7 * 7, 4096),
      nn.ReLU(inplace=True),
      nn.Linear(4096, 4096),
      nn.ReLU(inplace=True),
      nn.Linear(4096, num_classes),
    )


  def forward(self, x):
    x = self.features(x)
    x = self.avgpool(x)
    x = torch.flatten(x, 1)
    x = self.classifier(x)
    return x
```

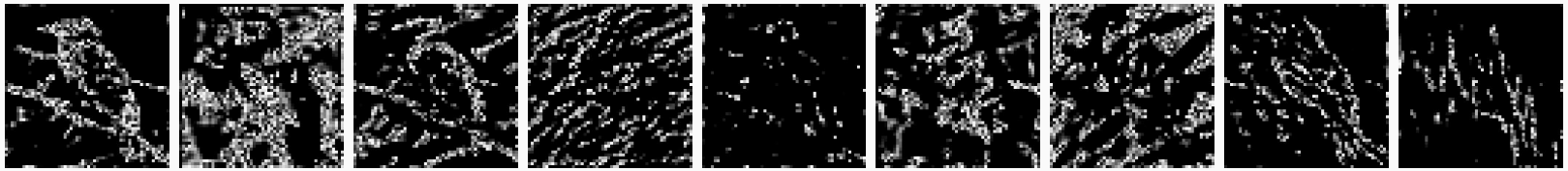
original input image (tensor size $3 \times 224 \times 224$, image has been resized to match that dimension):

a few output images from the 64 filters of `block1_conv2` (size $64 \times 224 \times 224$):

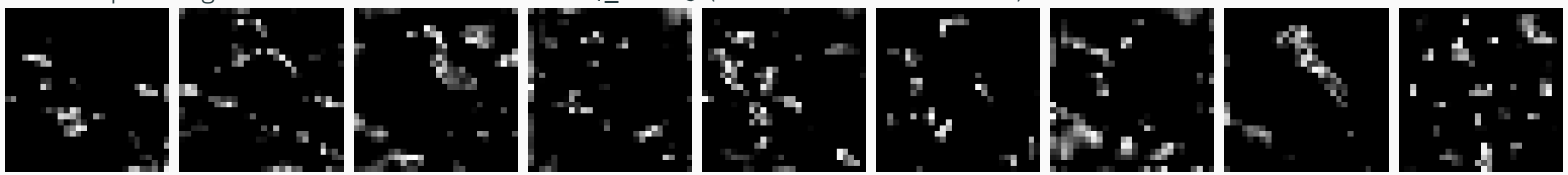
a few output images from the 128 filters of `block2_conv2` (output size $128 \times 112 \times 112$):



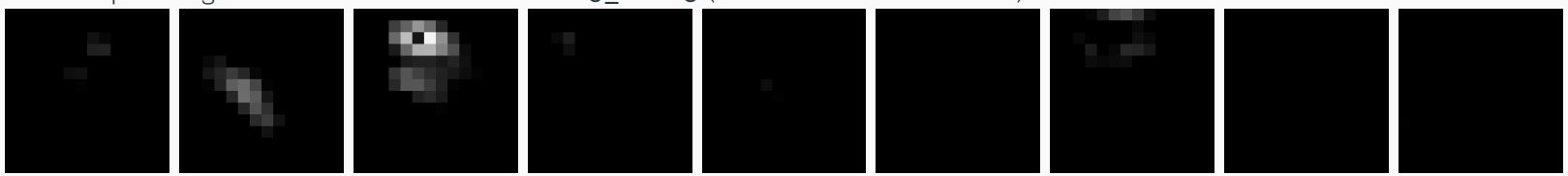
a few output images from the 256 filters of `block3_conv3` (output size 56×56):



a few output images from the 512 filters of `block4_conv3` (tensor size $512 \times 28 \times 28$):

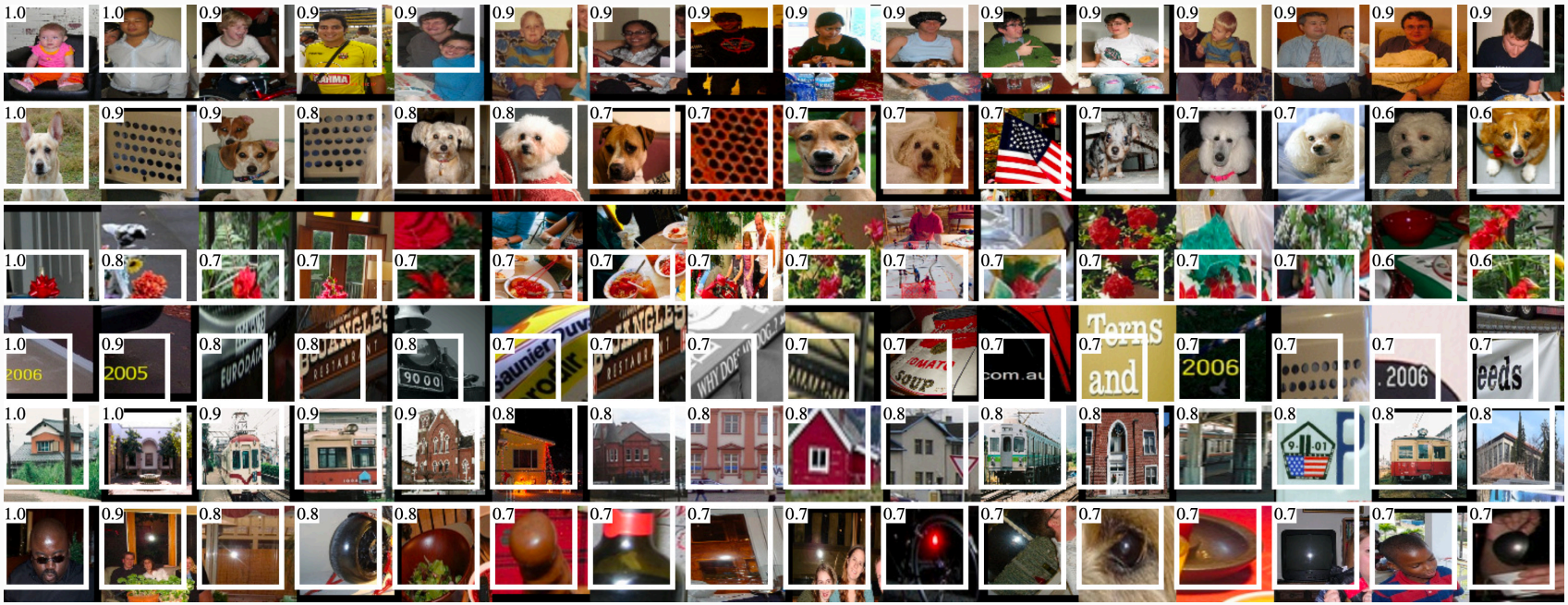


a few output images from the 512 filters of `block5_conv3` (tensor size $512 \times 14 \times 14$):



Visualisation

Understanding each of the inner operations of a trained network is still an open problem. Thankfully Convolutional Neural Nets focus on images and a few visualisation techniques have been proposed.



Here are selected images that maximise the output of 6 filters of AlexNet. The activation values and the receptive field of the particular neuron are shown in white. [Ross Girshick et al.]

Another technique (see link below) is to find an input image that maximises the activation for that specific filter. Recall that the output of ReLU and sigmoid is always positive and that a positive activation means that the filter has detected something. Thus finding the image that maximises the response from that filter will give us a good indication about the nature of that filter.

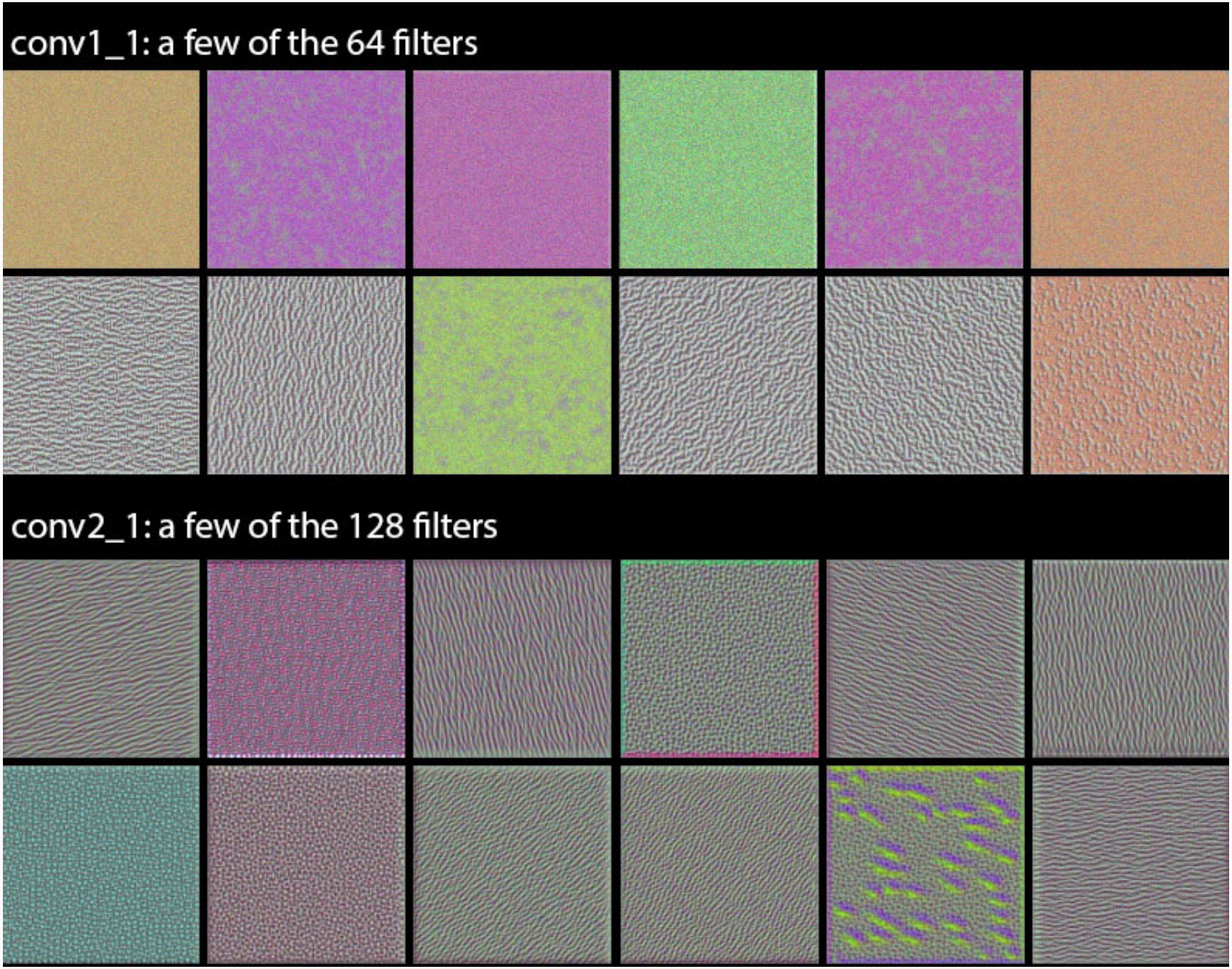
See https://blog.keras.io/category/demo.html

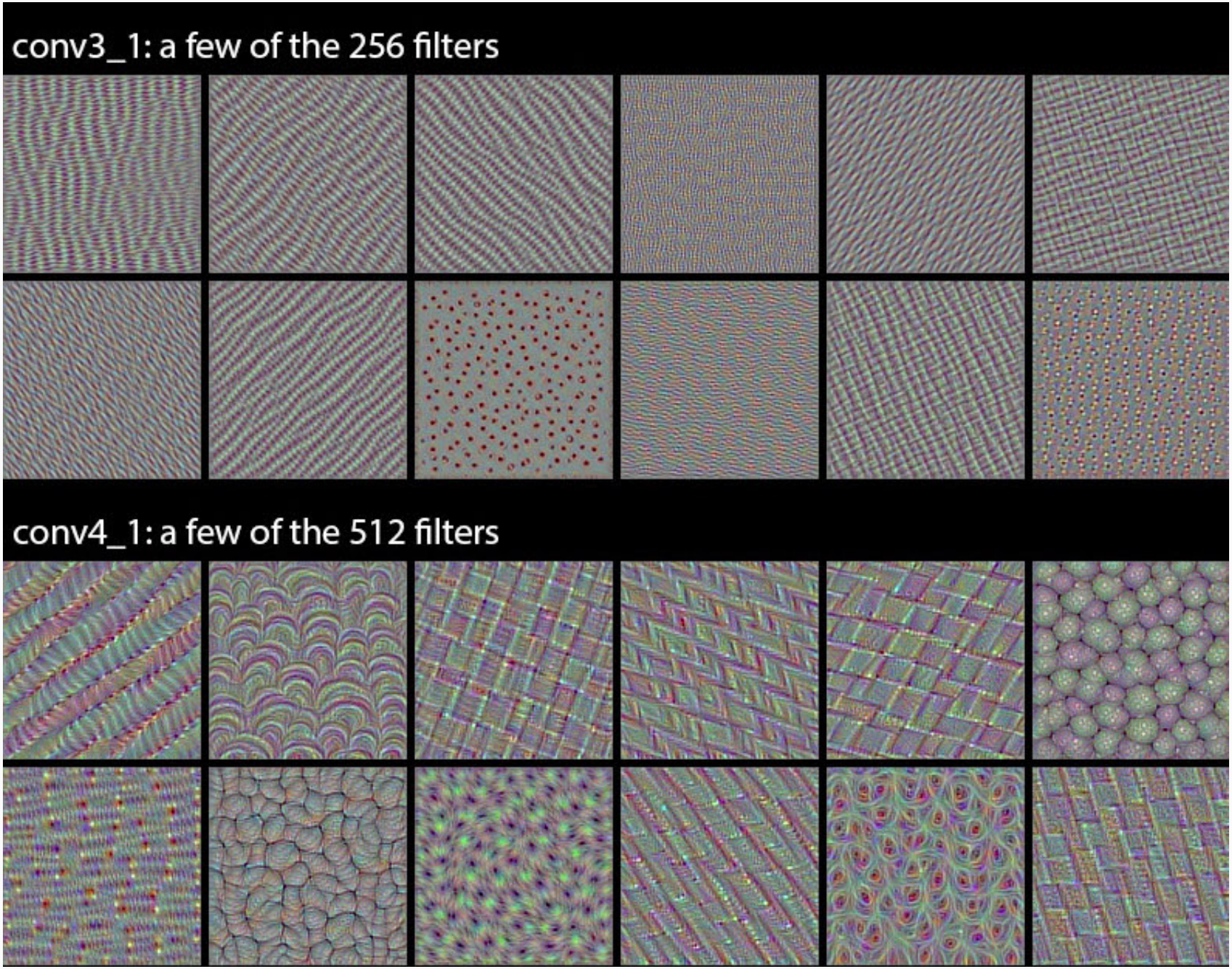
The optimisation proceeds as follows:

1. Define the loss function as the mean value of the activation for that filter.
2. Use backpropagation to compute the gradient of the loss function w.r.t. the input image.
3. Update the input image using a gradient *ascent* approach, so as to *maximise* the loss function. Goto 2.

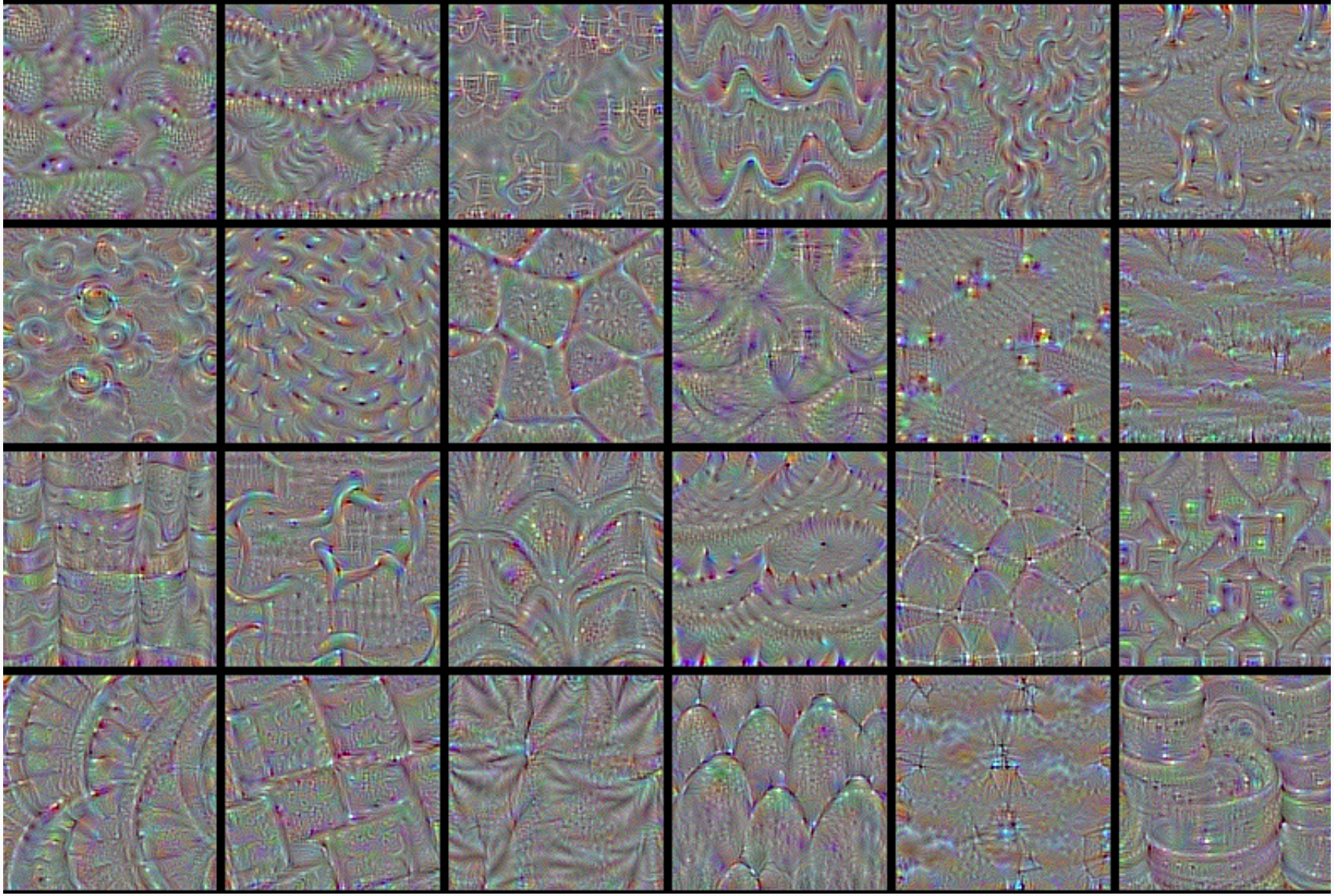
A few examples of optimised input images for VGG16 are presented in the next slides.

> See https://blog.keras.io/category/demo.html


conv1_1: a few of the 64 filters
conv2_1: a few of the 128 filters


conv3_1: a few of the 256 filters
conv4_1: a few of the 512 filters

conv5_1: a few of the 512 filters



Takeaways

Convolutional Neural Nets offer a very effective simplification over Dense Nets when dealing with images. By interleaving pooling and convolutional layers, we can reduce both the number of weights and the number of units.

The successes in Convnet applications (eg. image classification) were key to start the deep learning/AI revolution.

The mathematics behind convolutional filters were nothing new and have long been understood. What convnets have brought, is a framework to systematically train optimal filters and combine them to produce powerful high level visual features.

Useful Resources

[1] Deep Learning (MIT press) from Ian Goodfellow et al. - chapters 9

https://www.deeplearningbook.org

[2] Brandon Rohrer YT channel https://youtu.be/ILsA4nyG7I0

[5] Stanford CS class CS231n http://cs231n.github.io

[7] Michael Nielsen's webpage http://neuralnetworksanddeeplearning.com/