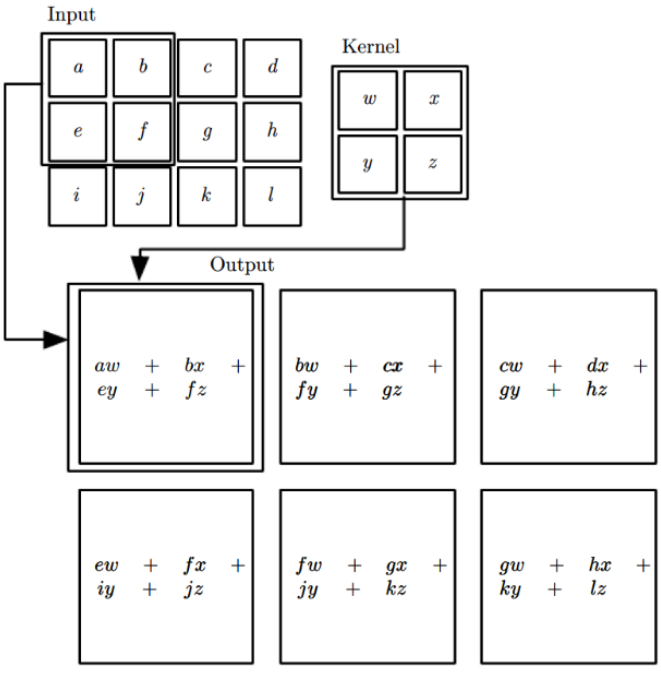
# Convolutional Neural Networks Notes

Matteo Lugli

January 4, 2024

## 1 Basic Idea



# 2 Notation Summary

The following table provides a summary of the variables used in the example described in 3.

| Explanation | Symbol |
|---|---|
| P | padding layers |
| X | input image |
| H | kernel |
| Y | output |
| N | filters applied in a layer |
| w | input width |
| h | input height |
| k | kernel size |

Table 1: Summary of symbols used

To understand the formulation it is really important to clearify a couple of important things:

- The following formula, given the plain kernel, it flips it and applies it to the image, performing the so called **convolution** (if you don't flip it, it's just a correlation);
- For the kernel matrix, we use indexes in such a way that the central element has coordinates [0,0]. This means that we will end up with some negative indices.
- The input matrix needs to be considered with **zero padding**. The first row and column of the resulting matrix will have index -P;
- The output matrix has standard coordinates, so starting from [0,0];

# 3 Example

We are going to use the following formula, Input and Kernel:

$$Y[m,n] = \sum_{i=-P+m} \sum_{j=-P+n} X[i,j] \;\; H[m-i, n-j] \tag{1}$$

| $i,j$ | -1 | 0 | 1 | 2 | 3 |
|---|---|---|---|---|---|
| -1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 2 | 3 | 0 |
| 1 | 0 | 4 | 5 | 6 | 0 |
| 2 | 0 | 7 | 8 | 9 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 |

Table 2: Input X

| $i,j$ | -1 | 0 | 1 |
|---|---|---|---|
| -1 | -1 | -2 | -3 |
| -0 | 0 | 0 | 0 |
| 1 | 1 | 2 | 3 |

Table 3: Kernel H

As you can see padding is alredy applied in the input matrix, applying 1 layer of zeroes means that we are going to start counting indexes at -1.

$$
\begin{aligned}
Y[0,1] \Rightarrow m = 0, n = 1, P = -1 \\
& i = -1 \\
& X[-1,0] \cdot H[1,1] = 0 \cdot 3 + \\
& X[-1,1] \cdot H[1,0] = 0 \cdot 2 + \\
& X[-1,2] \cdot H[1,-1] = 0 \cdot 1 + \\
& i = 0 \\
& X[0,0] \cdot H[0,1] = 1 \cdot 0 + \\
& X[0,1] \cdot H[0,0] = 2 \cdot 0 + \\
& X[0,2] \cdot H[0,-1] = 3 \cdot 0 + \\
& i = 1 \\
& X[1,0] \cdot H[-1,1] = 4 \cdot -3 + \\
& X[1,1] \cdot H[-1,0] = 5 \cdot -2 + \\
& X[1,2] \cdot H[-1,-1] = 6 \cdot -1 + \\
& = -28
\end{aligned} \tag{2}
$$

As you can see in table 4, we flipped the kernel on both axis and overlapped it with the correct portion of the input.

| $i, j$ | -1 | 0 | 1 | 2 | 3 |
|---|---|---|---|---|---|
| -1 | 0 | $0^{(3)}$ | $0^{(2)}$ | $0^{(1)}$ | 0 |
| 0 | 0 | $1^{(0)}$ | $2^{(0)}$ | $3^{(0)}$ | 0 |
| 1 | 0 | $4^{(-3)}$ | $5^{(-2)}$ | $6^{(-1)}$ | 0 |
| 2 | 0 | 7 | 8 | 9 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 |

Table 4: Convolution to compute element $Y[0,1]$ of output matrix

Let's write also the calculations made to compute $Y[1,2]$

$$
\begin{aligned}
Y[1,2] \Rightarrow m = 1, n = 2, P = -1 \\
& i = 0 \\
& X[0,1] \cdot H[1,1] = 2 \cdot 3 + \\
& X[0,2] \cdot H[1,0] = 3 \cdot 2 + \\
& X[0,3] \cdot H[1,-1] = 0 \cdot 1 + \\
& i = 1 \\
& X[1,1] \cdot H[0,1] = 5 \cdot 0 + \\
& X[1,2] \cdot H[0,0] = 6 \cdot 0 + \\
& X[1,3] \cdot H[0,-1] = 0 \cdot 0 + \\
& i = 2 \\
& X[2,1] \cdot H[-1,1] = 8 \cdot -3 + \\
& X[2,2] \cdot H[-1,0] = 9 \cdot -2 + \\
& X[2,3] \cdot H[-1,-1] = 0 \cdot -1 + \\
& = -30
\end{aligned} \tag{3}
$$

| $i, j$ | -1 | 0 | 1 | 2 | 3 |
|---|---|---|---|---|---|
| -1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | $2^{(3)}$ | $3^{(2)}$ | $0^{(1)}$ |
| 1 | 0 | 4 | $5^{(0)}$ | $6^{(0)}$ | $0^{(0)}$ |
| 2 | 0 | 7 | $8^{(-3)}$ | $9^{(-2)}$ | $0^{(-1)}$ |
| 3 | 0 | 0 | 0 | 0 | 0 |

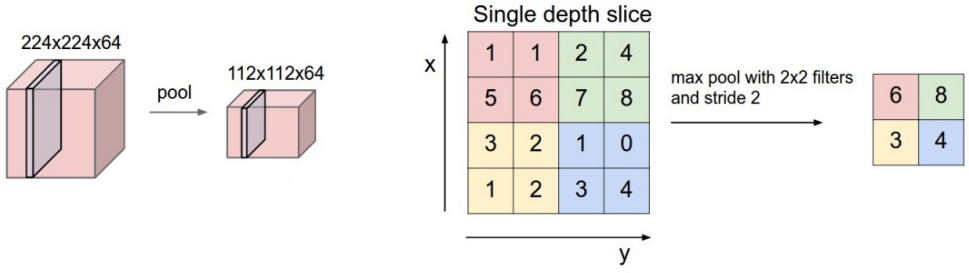
Table 5: Convolution to compute element $Y[1,2]$ of output matrix

# 4 Convolution Layers

In most cases CNNs are used with RGB images. As the name says, such images are made of 3 **channels** (Red, Green and Blue). You can imagine them as $h \times w \times 3$ cubes, or as 3 $h \times w$ matrixes stacked. This means that we have to imagine our kernels as $k \times k \times 3$ dimensional cubes as well! Now it's easy to imagine why for each convolutional layer there are $k \times k \times c \times N + N$ learnable parameters, where $c$ represents the number of channels of the input data (3 in case of RGB images), and $N$ is the number of filters for that layer. It's really important to remember that the number of channels for the next layer becomes $N$: if in the first layer we use 10 kernels to process our plain image, the input that needs to be processed by the next layer will have 10 channels!

# 5 Pooling Layers

Pooling layers are used as noise reduction layers or to perform dimensionality reduction. The most common types of pooling are *max pooling* and *average pooling*, which either compute the maximum value among the overlapped elements or the average value. The figure below should be clear enough.




The only 2 hyperparameters that need to be defined are

- Pool size $k$, in this case equal to 2;
- Pool stride $s$, in this case equal to 2 as well;

Advantages of the use of pooling layers include:

- Robustness to *exact* location of features;
- Preventing overfitting;
- Reduce computational cost by reducing dimensions;
- Increasing receptive field of following layers;

The most common pooling configuration in $K = 2x2$ with stride $S = 2$. With this configuration, 75% of the input volume is discarded.

# 6 Output volume size

A trick that is usually done when implementing CNNs and not mess up with the dimensions, is to use *kernel size* = 3 with a 1 layer padding. An example of such a convolution has alredy been presented in section 4. If you notice, by using such kernel size and padding, you can overlap the center of the kernel with the top left element of the input image, and make the kernel slide untill it overlaps the top right element. Considering that each overlap produces a single element, for each row that we process this way we get $w$ elements. If we do that for each single row (so by applying a complete convolution) the output size of the matrix will be exactly $h \times w$, the same as the size of the input. One of the benefits of doing so is that we can stack how many convolutional layers as we want and we are sure that the height and width of the output will not change untill we apply a pooling layer.

| Type of layer | Output size |
|---|---|
| Convolutional | $w_2 = \frac{w_1 - k + 2p}{s} + 1$ |
| | $h_2 = \frac{h_1 - k + 2p}{s} + 1$ |
| | $c_2 = N$ |
| Pooling | $w_2 = \frac{w_1 - k}{s} + 1$ |
| | $h_2 = \frac{h_1 - k}{s} + 1$ |
| | $c_2 = c_1$ |

Table 6: Output size cheatsheet

# 7 VGG

VGG is a "very deep" CNN used for image recognition tasks. This particular architecture (shown in figure) is really effective even if it uses really small kernels (3x3). The main idea that the authors (Oxford vision lab) of the paper wanted to highlight is that by increasing the depth of the network (so the number of weight layers) you can achieve better performance. The authors actually released the pre-trained weights, and many more papers fine-tuned the network to perform different tasks. To become

more familiar with dimensions, let's look at the first block of 2 conv layers that are applied on the input, whose dimensions are 224x224x3. Figure 1 is taken from the paper where VGG was first proposed.

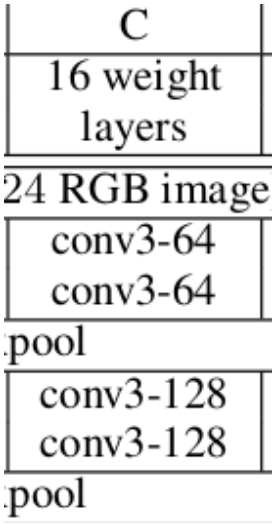



Figure 1: VGG16 first layers

- The input is taken and the first first layer of the first block is applied. The notation for this layer is conv3-64, meaning that we use kernels of 3x3xC and we want the output channels to be 64, so we use 64 kernels. As you can see, the input channels are not specified. In this case C is 3, because we work on the input that has 3 channels(RGB).
  The output dimensions are 224x224x64. As you can see the width and height of the original image do not change, because we use 3x3 kernels with a padding of 1;
- The second layer of the first block is applied on the previous output. The notation for this layer is the same as before, but this time the input channels are 64 and NOT 3 anymore, because the previous conv layer output channels where 64. The output dimensions are the same as the input dimensions, 224x224x64;
- We then apply a pooling layer, halving the width and height dimensions. The output dimensions of this pooling layer are 112x112x64.

- Now we step in the second block, where the output channels are 128. The first layer of the second block uses kernels of 3x3xC, where C in this case is 64. We use 128 kernels because we want the output channels to be 128.
- ...

Always be careful with output and input channels, because the notation might be slightly confusing.
Let's look at the pytorch code of VGG-16 implementation, where the dimensions are specified (in channels, $\text{out}_{\text{channels}}$). The kernel size is wrapped in the `VGGlayer` class because it is always 3 (actually in the real implementation 1x1 kernels are used as well, but you can ignore this detail for now).

```
self.conv_features = nn.Sequential(
    VGGlayer(in_channels, 64),
    VGGlayer(64, 64, max_pool=True),

    VGGlayer(64, 128),
    VGGlayer(128, 128, max_pool=True),

    VGGlayer(128, 256),
    VGGlayer(256, 256),
    VGGlayer(256, 256, max_pool=True),

    VGGlayer(256, 512),
    VGGlayer(512, 512),
    VGGlayer(512, 512, max_pool=True),

    VGGlayer(512, 512),
    VGGlayer(512, 512),
    VGGlayer(512, 512, max_pool=True),
)
```

Figure 2: VGG16 Conv layers implementation

After the conv layers, a MLP is used to perform classification. You can see the structure of this network in Figure 2 together with the forward function, that requries

a flatten step to "unroll" the matrices in a 1D tensor that can be forwarded through the MLP.

```
        self.avgpool = nn.AdaptiveAvgPool2d((7, 7))

        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(),
            nn.Dropout(),
            nn.Linear(4096, num_classes),
        )

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

Figure 3: VGG16 DNN layer and forward function

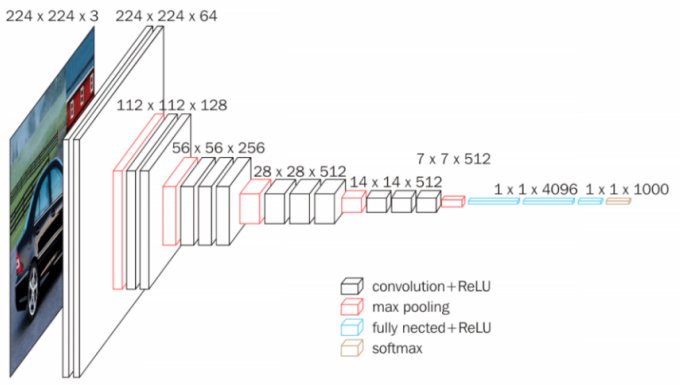



Figure 4: VGG16 Structure