Lesson 9

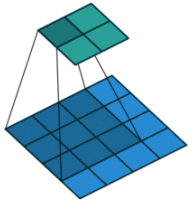
- backpropagation for CNNs
- transposed convolutions
- dilated convolutions
- normalization layers

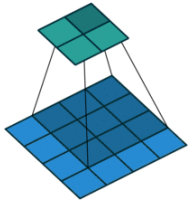
Backpropagation for CNNs

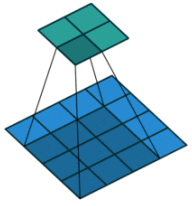
Suggested reading:

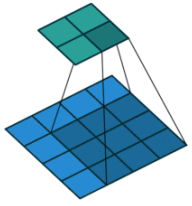
Convolution arithmetic tutorial

The convolution matrix







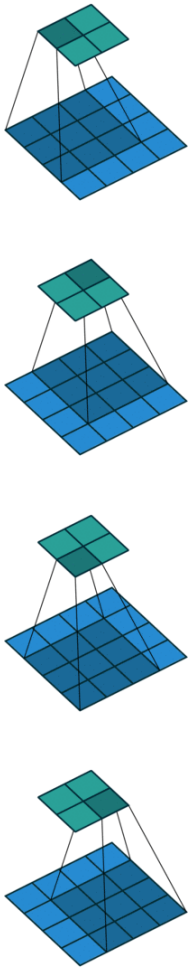


Input and output are unrolled into vectors from left to right, top to bottom. So, the input has dimension 4x4=16, and the output has dimension 2x2=4 (number of times we can apply the convoluion).

The operation performed by the convolution can be seen as a single dense layer, with 16 inputs and 4 outputs.

Let us compute the weights, in terms of the kernel weights.

The convolution matrix



$$\begin{pmatrix}
w_{0,0} & 0 & 0 & 0 \\
w_{0,1} & w_{0,0} & 0 & 0 \\
w_{0,2} & w_{0,1} & 0 & 0 \\
0 & w_{0,2} & 0 & 0 \\
w_{1,0} & 0 & w_{0,0} & 0 \\
w_{1,1} & w_{1,0} & w_{0,1} & w_{0,0} \\
w_{1,2} & w_{1,1} & w_{0,2} & w_{0,1} \\
0 & w_{1,2} & 0 & w_{0,2} \\
w_{2,0} & 0 & w_{1,0} & 0 \\
w_{2,1} & w_{2,0} & w_{1,1} & w_{1,0} \\
w_{2,2} & w_{2,1} & w_{1,2} & w_{1,1} \\
0 & w_{2,2} & 0 & w_{1,2} \\
0 & 0 & w_{2,0} & 0 \\
0 & 0 & w_{2,1} & w_{2,0} \\
0 & 0 & w_{2,2} & w_{2,1} \\
0 & 0 & 0 & w_{2,2}
\end{pmatrix}^T$$

Each column corresponds to a different application of the kernel

$w_{i,j}$ is a kernel weight, with i and j being the row and column of the kernel respectively

The matrix hass been transposed for convenience

Backpropagation

$$\begin{pmatrix}
w_{0,0} & 0 & 0 & 0 \\
w_{0,1} & w_{0,0} & 0 & 0 \\
w_{0,2} & w_{0,1} & 0 & 0 \\
0 & w_{0,2} & 0 & 0 \\
w_{1,0} & 0 & w_{0,0} & 0 \\
w_{1,1} & w_{1,0} & w_{0,1} & w_{0,0} \\
w_{1,2} & w_{1,1} & w_{0,2} & w_{0,1} \\
0 & w_{1,2} & 0 & w_{0,2} \\
w_{2,0} & 0 & w_{1,0} & 0 \\
w_{2,1} & w_{2,0} & w_{1,1} & w_{1,0} \\
w_{2,2} & w_{2,1} & w_{1,2} & w_{1,1} \\
0 & w_{2,2} & 0 & w_{1,2} \\
0 & 0 & w_{2,0} & 0 \\
0 & 0 & w_{2,1} & w_{2,0} \\
0 & 0 & w_{2,2} & w_{2,1} \\
0 & 0 & 0 & w_{2,2}
\end{pmatrix}^T$$

We compute weights updates as usual.

However, updates relative to a same kernel weight must be shared, e.g. taking a mean among all updates.

Transposed convolutions

Transposed convolutions

Normal convolutions with non unitarian strides downsample the input dimension.

In some cases, we may be interested to upsample the input, e.g. for

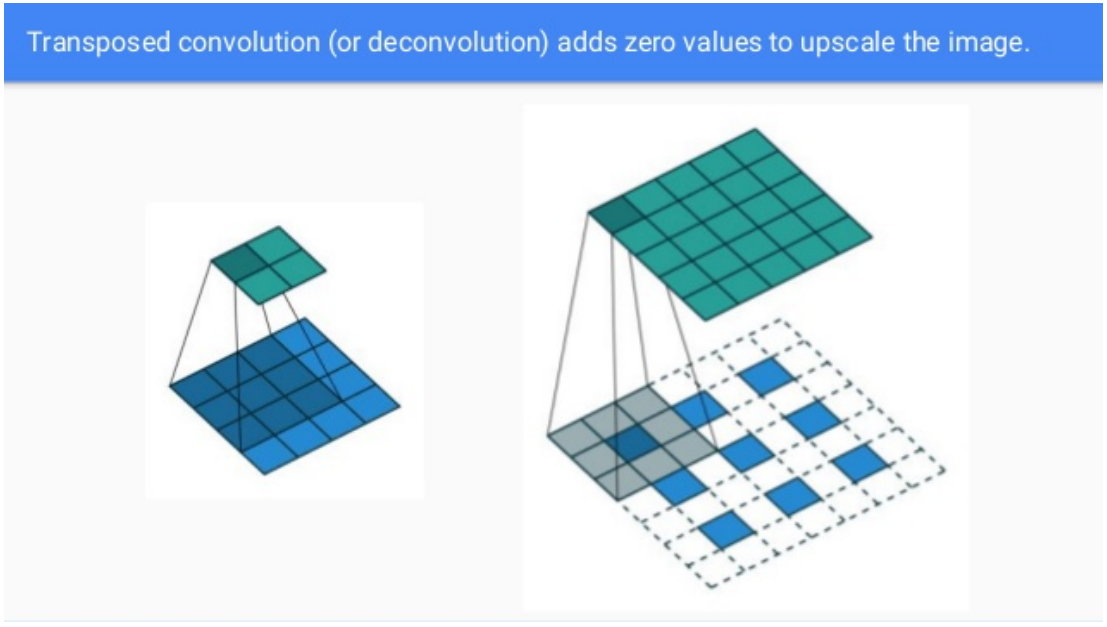
- image to image processing, to obtain an image of the same dimension of the input (or higher) after some compression to an internal encoding
- project feature maps to a higher-dimensional space.

We shall see several applications later on.

Transposed convolutions as fractionally strided conv.

A transposed convolution (sometimes called deconvolution) can be thought as a normal convolution with subunitarian stride.

To mimic subunitarian stride, we must first properly upsample the input (e.g. inserting empty rows and columns) and then apply a single strided convolution:



A better view

$$\begin{pmatrix}
w_{0,0} & 0 & 0 & 0 \\
w_{0,1} & w_{0,0} & 0 & 0 \\
w_{0,2} & w_{0,1} & 0 & 0 \\
0 & w_{0,2} & 0 & 0 \\
w_{1,0} & 0 & w_{0,0} & 0 \\
w_{1,1} & w_{1,0} & w_{0,1} & w_{0,0} \\
w_{1,2} & w_{1,1} & w_{0,2} & w_{0,1} \\
0 & w_{1,2} & 0 & w_{0,2} \\
w_{2,0} & 0 & w_{1,0} & 0 \\
w_{2,1} & w_{2,0} & w_{1,1} & w_{1,0} \\
w_{2,2} & w_{2,1} & w_{1,2} & w_{1,1} \\
0 & w_{2,2} & 0 & w_{1,2} \\
0 & 0 & w_{2,0} & 0 \\
0 & 0 & w_{2,1} & w_{2,0} \\
0 & 0 & w_{2,2} & w_{2,1} \\
0 & 0 & 0 & w_{2,2}
\end{pmatrix}^T$$

Given a kernel, we may define a convolution matrix passing from some input dimenson d to an output dimension o.

By simply transposing the matrix, we may convert an input of dimension o into an output of dimension i.

So, the kernel defines a convolution matrix, but whether it is a direct convolution or a transposed convolution is determined by how the matrix is applied.

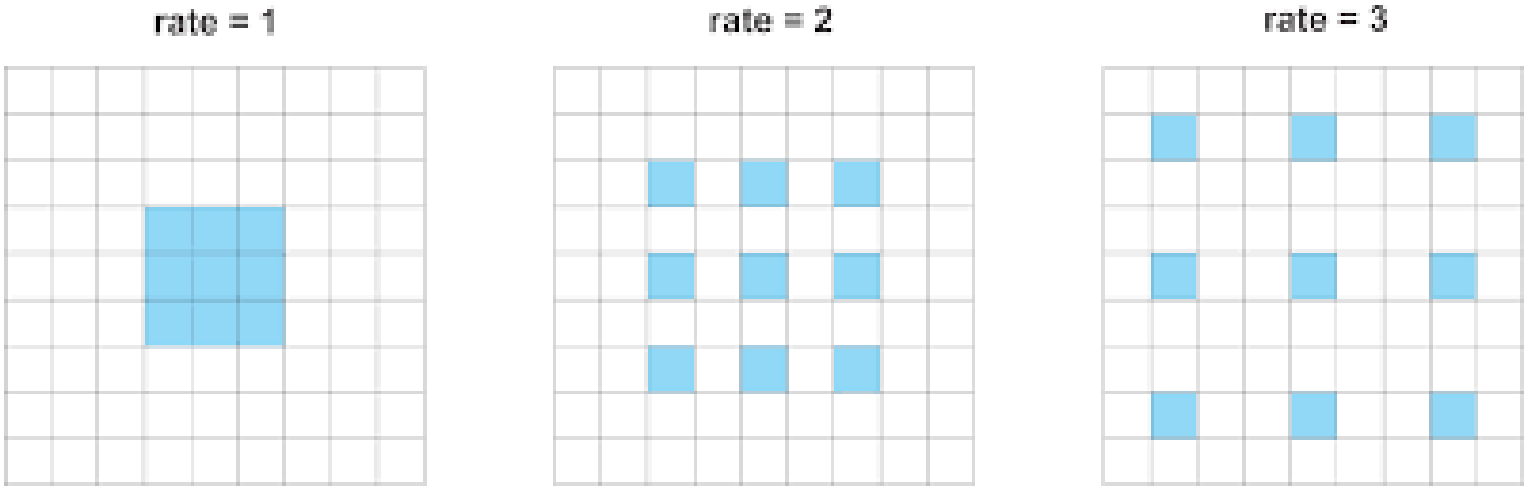
Dilated (aka Atrous) Convolutions

Suggested reading:

Rethinking Atrous Convolution for Semantic Image Segmentation. By L-C.Chen, G.Papandreou, F.Schroff, H.Adam

Dilated convolutions

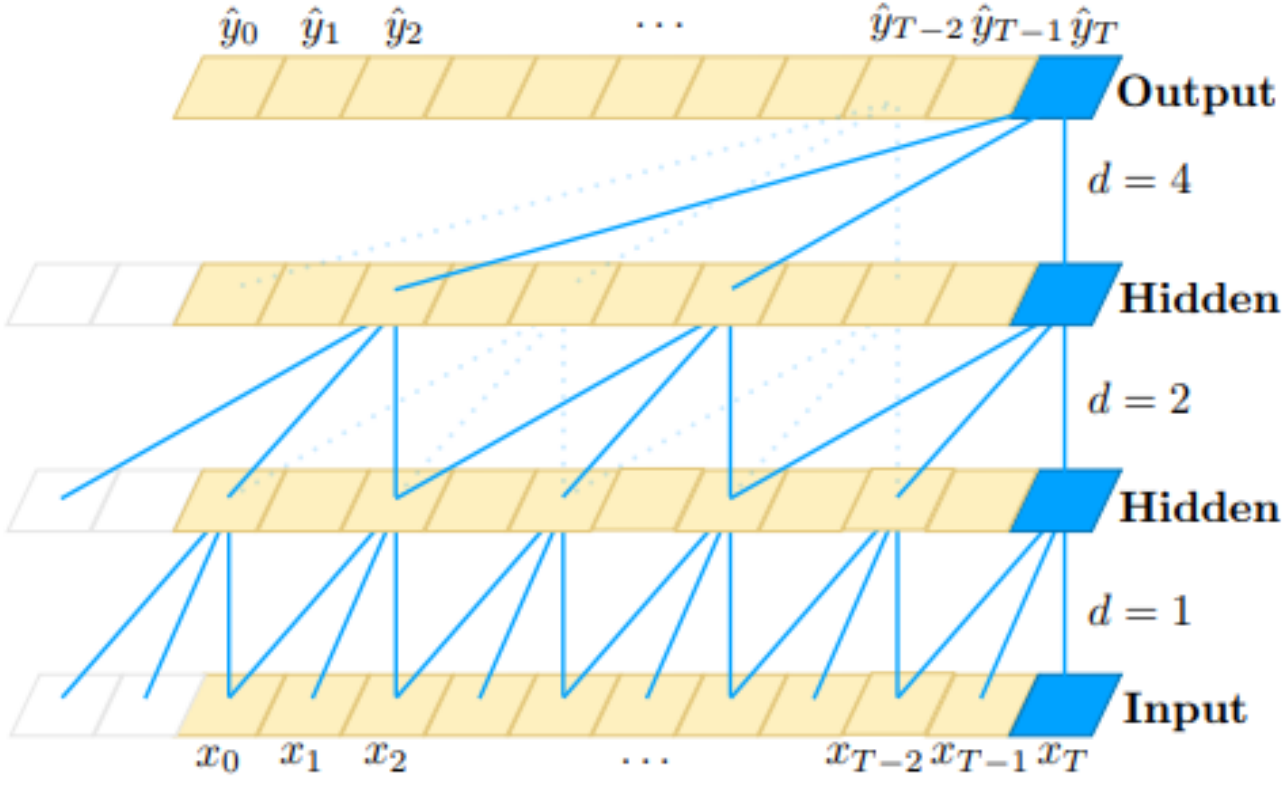
Dilated convolutions are just normal convolutions with holes.



It enlarges the receptive fields, keeping a low number of parameters.
Might be useful in first layers, when working on high resolution images.

Temporal Convolutional Networks

Used in Temporal Conovlutional Networks (TCNs) to process long input sequences :



Normalization layers

Expected benefit

- have a more stable and possibly faster training
- increase the independence between layers

Batch Normalization

Batch normalization operates on single layers, per channel base.

At each training iteration, the input is normalized according to **batch (moving) statistics**, subtracting the mean μ^B and dividing by the standard deviation σ^B.

Then, an opposite transformation is applied based on **learned** parameters γ (scale) and *beta* (center).

$$BN(x) = \gamma \cdot \frac{x - \mu^B}{\sigma^B} + \beta$$

Batch statistics are **moving** in the sense that we keep running values and slowly update them based on the current minibatch.

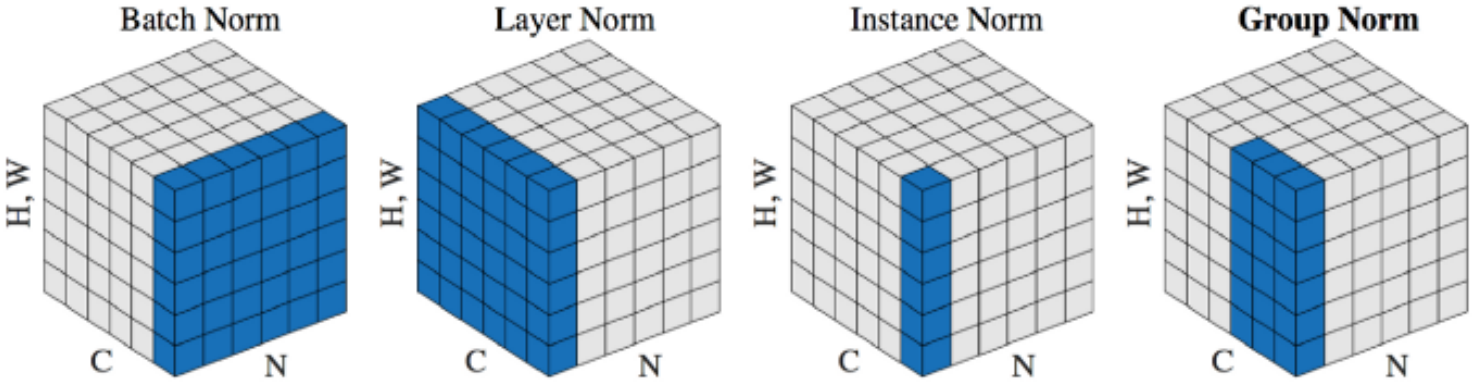
Batch Normalization at prediction time

Batch normalization behaves differently in training mode and prediction mode.

Typically, after training, we use the entire dataset to compute stable estimates of the variable statistics and use them at prediction time.

Once training is concluded, statistics (over a given training set) do not change any more.

Other forms of normalization



Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes.

The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

See Group Normalization for a discussion.