Convolutional Neural Networks

Introduction

Convolutional Neural Networks (CNNs) are networks that exploit the operation of convolution in order to process high dimensional data (like images) transforming the dimensionality of the original space.

Convolutions

Convolutions are mathematical operation between two functions, where one function is multiplied by the other one shifted. In the continuous case the operation is expressed as an integral:

$$(x * w)(t) = \int_{a=-\infty}^{\infty} x(a)w(t-a)da$$

The corresponding formula for convolution in the discrete case is based on a summation:

$$(x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

In the case of CNNs, we will use limited convolutions (we have a limited amount of terms). The two functions involved are conventionally called Input function I and Kernel function K:

$$(I * K)(i,j) = \sum_{m \in S_1} \sum_{n \in S_2} I(m,n) K(i-m, j-n)$$

What we have seen so far is the 2D case, but the operation can be easily extended to n-dimensions by taking into account n indexes. For example, the equivalent in 3D is:

$$(I * K)(i,j,k) = \sum_{m \in S_1} \sum_{n \in S_2} \sum_{u \in S_3} I(m,n,u) K(i-m, j-n, k-u)$$

Convolutions hold the following properties:

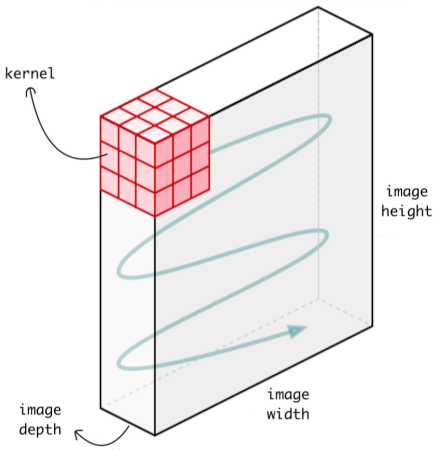
- Commutative: $(I * K)(i,j) = (K * I)(i,j)$
- Cross-correlation: flipping the kernels in the convolution operation produce the same result.

The kernel function

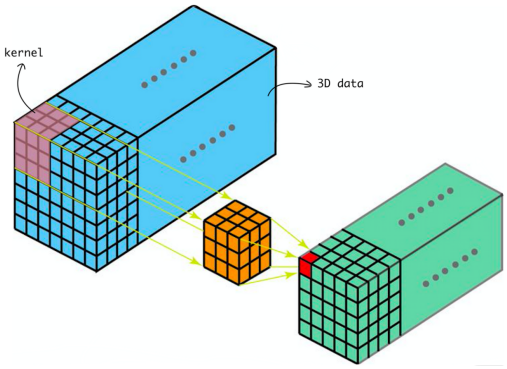
Our goal with convolution will be finding the proper kernels for our problem. We can think about kernels as filters that can be applied to inputs through the operation of convolution.

The dimension of the convolution defines the direction where the kernel will move:

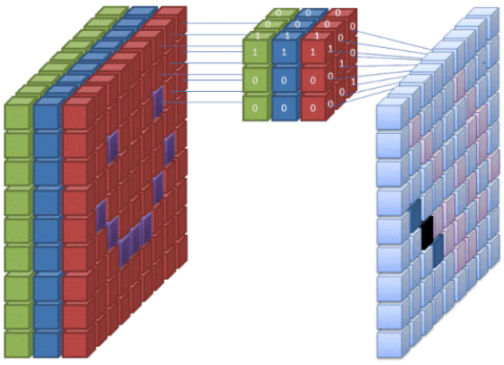
Hence, the dimension of convolution is independent from the size of the input. For example a 1D convolution can be applied to a 2D or 3D input etc.



In practice, when we apply convolution each time we take a slice of the input corresponding to the size of the kernel, we multiply element-wise the two slices and sum each element. Then the kernel is shifted and the operation is repeated at each shift:



We will focus particularly on 2D convolutions. As we already mentioned, this means that our kernel function will be able to move in 2 directions, which implies that the kernel's depth must always be the same size of the input's depth:



Some terminology

A typical architecture for a CNN consists in three layers: an **input layer**, a **convolutional layer** and finally an **output layer**.

We can think about the convolutional layer as a transformation of the input space in some other (tipically reduced) space.

The **input size** is denoted as $w_{in} \times h_{in} \times d_{in}$. The depth of the input usually identifies its number of channels. For example an RGB image will have $d_{in} = 3$.

Similarly, the **kernel size** will be defined bu $w_k \times h_k \times d_k$ and the **output size** $w_{out} \times h_{out} \times d_{out}$

Since we can use multiple kernels, we define d ad the number of kernels of a convolutional layer.

We define the **feature map** (or depth slice) a particular section of the output. In general, we can say that the output of a convolution is a composition of feature maps.

For example the convolution between a $32 \times 32 \times 3$ input and a single $5 \times 5 \times 3$ kernel outputs a $28 \times 28 \times 1$ feature map. If we have $d = 6$ kernels, we will obtain a $28 \times 28 \times 6$ feature map.

The **trainable parameters** of a convolutional layer correspond to the total parameters of its kernels. The i-th convolutional layer will have $w_k^{(i)} \times h_k^{(i)} \times d_k^{(i)} \times d^{(i)}$ trainable parameters.

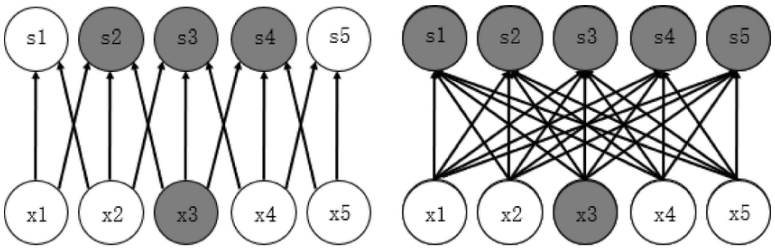
Inside a convolutional layer

Internally, a convolutional layer is composed by three stages. Initially, the **convolution stage** will take the input and convolutes it with a kernel. Then the convololution is passed to a **detector stage**, that applies a non-linear function. Finally the **pooling stage** will transform the output of the detector by applying another filter.

Sparse connectivity

When defining the architecture of a CNN, we have to remember the problem we want to solve, which is usually to work with images or videos. In these cases we care about locality where considering the features of the input. In fact it’s much more likely that a pixel of an image is correlated to it’s neighbour with respect to another pixel far away.

This is why introduce the concept of **sparse connectivity**. Instead of working with fully connected layers, we exploit the concept of **locality**, we connect a pixel just to the adjacent one, thus saving in terms of numbers of parameters needed.



(a) sparse connectivity **(b)** dense connectivity

Kernels work with local operations by computing each time a subset of the input.

Parameter sharing

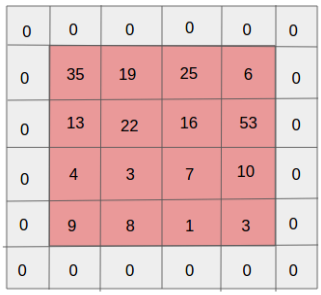
Parameter sharing is a constraint on the weights imposing that some of the weights of the same layer must have the same value. This method allows for a substantial reduction in the number of total trainable parameters.

When we use a kernel, we are basically repeating operations on different slices of the output by using each time the same kernel parameters. This implies a substantial re In fact, when we look at how kernels work, they basically implement both concepts of sparse connectivity and parameter sharing.

Padding

We have seen that in general the operation of convolution between an input and a kernel results in an output with lower dimensions that the input. In particular, if we have an input of sice w_{in} and a kernel of size w_k, the output will have a width of $w_{out} = w_{in} - w_k + 1$. Notice that this happens if the convolution operation starts with the kernel aligned to the border of the input.

Sometimes, we may want our output to have the same size of the input. In this cases we introduce the concept of padding. Basically we compensate the loss of size by extending the origin input using some filling values (usually 0).



The number of this fillings depend on the kernel's size and follows this formula:

$$p = \lfloor \frac{w_k}{2} \rfloor$$

Padding is called:

- Valid if $p = 0$
- Same if $p = \frac{w_k}{2}$

The Pooling stage

We already mentioned that convolutions impliy a change in the size between input and output. Tipically, while the witdth and height become smaller, the depth of the output increases depending on the number of kernels used.

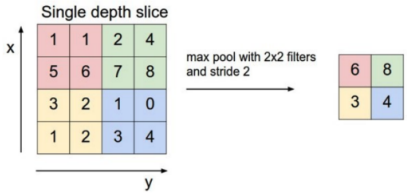
The pooling stage is responsible for further decreasing the size of the final output by applying a filtering operation. This time, the filter consists in a well defined function that has no trainable parameters.

Two common approaches for the pooling stage are:

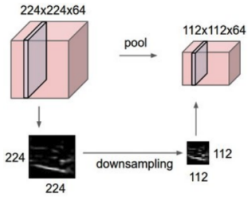
- Max-pooling: we extract the maximum value from the considered region.
- Average-pooling: returns the average value from the considered region.

An important parameter for padding is **stride**. Stride defines by how much the pooling filter moves at each step. In case of more than one dimension, we can define different stride parameters for each dimension. Pooling is considered as a subsampling operation.

Example of **max pooling**



Introduces subsampling:



Computing the size of the output

Considering an input of size $w_{in} \times h_{in} \times d_{in}$ and a kernel $w_k \times h_k \times d_k$ with stride s and padding p, the dimensions of the resulting feature map will be:

$$w_{out} = \frac{w_{in} - w_k + 2p}{s} + 1$$

$$h_{out} = \frac{h_{in} - h_k + 2p}{s} + 1$$

The total number of trainable parameter of a convolutional layer is:

$$|\theta| = w_k \cdot h_k \cdot d_{in} \cdot d_{out} + d_{out}$$

Transfer learning

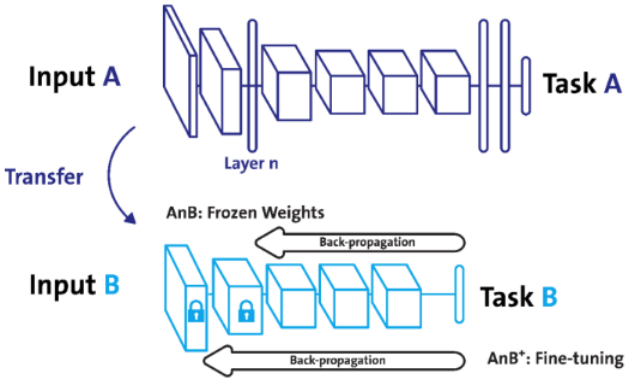
The main idea with transfer learning is to use pre-trained model on new examples belonging to a similar domain. A major advantage of this method is to reduce the computation needed for training from scratch a new model.

Assuming that we have a source domain D_s from where we obtained a function $f_s : X_s \rightarrow Y_s$, with $D(x_s)$ being the distribution that has been used to extract the images from the dataset and $P(y_s|x_s)$ the probability distribution of the output with respect to the input.

Let's now consider a target domain D_t with $f_t : X_t \rightarrow Y_t$, $D(x_t)$ and $P(y_t|x_t)$. Can we exploit the work that has been done in training a model for the source domain and apply it to solve the problem on target domain?

There are many possible techniques.

If both source and target domain are similar (they come from a similar distribution), we can think about keeping the feature extraction part the same and re-training only the last layers. This approach is called **fine-tuning**. Notice that this method still requires a lot of computation, since the final layers of a CNN are dense.



Another intelligent approach is to exploit our CNNs just as feature extractors. In fact, we have seen that CNNs are really good at extracting the fundamental features of the input via the operation of convolution. We could use this property in order to create a pipeline where the CNN extracts the features, that are then fed to another ML model in order to compute a prediction.