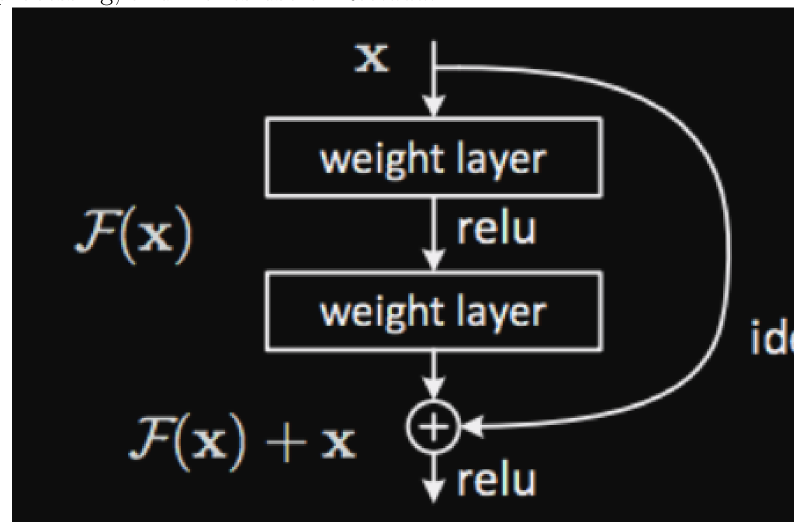
Inception V3

In inception-like CNNs, we have a deep/convolutional part comprised of inception modules, in which we extract features and information from the input. At the very end, we have some kind of pooling operation, and we process the information depending of which kind of purpose has our CNN (i.e., classification, regression…). This final part is usually not very, since it is comprised of only 2/3 dense layers.

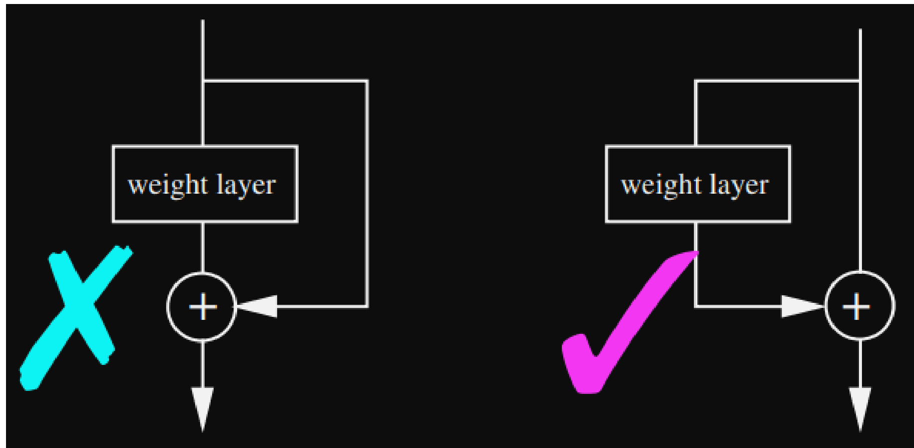
ResNet - Residual Learning

This type of CNN is also used for image processing, and makes use of *Residual Learning*, which is described by this image:



In Residual Learning, we have some kind of connection from the input to the output. In particular, instead of learning a function $F(x)$ you try to learn $F(x) + x$.

The intuition of this approach may be better understood by this image:



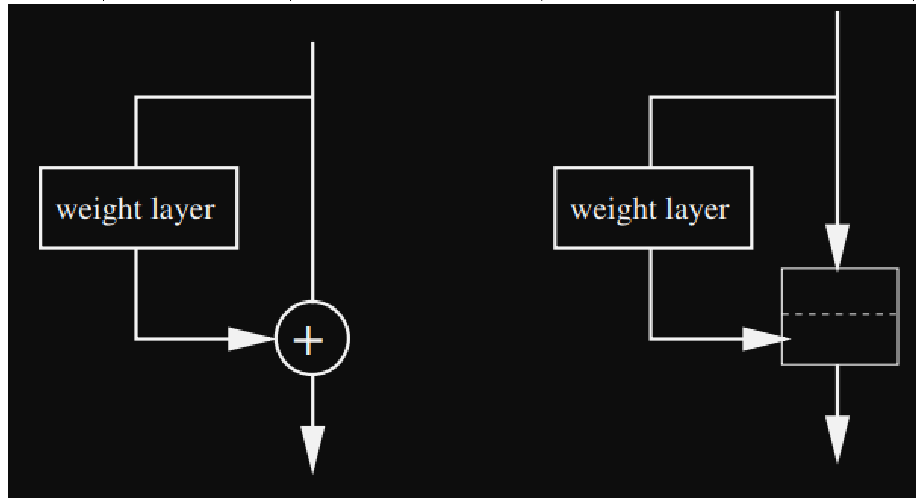
Essentially, we are adding a *residual computation* to our input, so we are computing a kind of *delta* (which is F(x)) to the input, it is a kind of residual that can help in the computation. In general, it can be seen as a way to see if our input has some kind of improvement or not.

You add a residual shortcut connection every 2-3 layers. Inception Resnet is an example of a such an architecture.

The **main advantage** is that along these links, you have a *better backpropagation* of the *loss*. It's true that if use RELU (Rectified Linear Units) instead of, say a sigmoid, you do not have the problem of vanishing gradient, but in any case there's a quick degradation of the loss going deeper and deeper in the net. Instead, using this method, ==there's no *degradation of the gradient*==.

Why Residual Learning works? Not well understood yet. The usual explanation is that during back propagation, *the gradient at higher layers can easily pass to lower layers*, withouth being mediated by the weight layers, which may cause vanishing gradient or exploding gradient problem.

Residual Learning - Sum or concatenation? The "sum" operation can be interpreted in a liberal way. A common variant consists in concatenating (*skip connection*) instead of adding (usually along the channel axis):



By using a concatenation, we are usually just skipping some parts of the network.

The point is to induce the net to learn different filters. One important example is the UNet, which we'll see in the future.

Efficient Net

A decision that we have to make when developing image processing network (CNN) is, for example, the size of the input.

ConvNets essentially grow in *three directions*: - **Layers**: the number of layers - **Channels**: the number of channels for layers - **Resolution**: the spatial width of layers Is there a principled method to scale up ConvNets that can achieve better accuracy and efficiency?

The resolution of a ConvNet is an important aspect that we have to address. In fact, we may decide that the information contained in high-res images is not important, and we may decide to lose this information in order to decrease the computational cost of the model.

The resolution of the input also affects the resolutions of *all the feature maps*.

Regarding the channels, we know that if we decrease the spacial dimension, we

have an increase along the channel axis. Usually, we have to try to work with as many channel that we have at our disposal.

At the same time, we have to work uniformly along the "3 dimensions", so if we increase the resolution, we should probably also increase the channels etc.
More info in this paper here.

Transfer Learning

In **transfer learning**, we try to *transfer knowledge* from a model into another model, ==usually we do this when we have a very good network that has been trained on a lot of data, and we want to transfer knowledge to a more specific network for which we do not have a lot of data==.

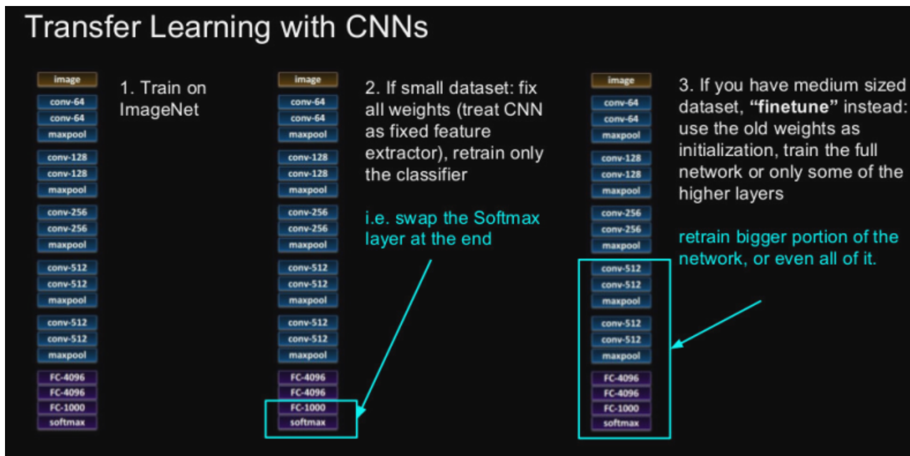
We can understand better this process if we consider the typical structure of NN, in which we have a part in which we are trying to extract features from the data, and another final part (the dense part, made by dense layers) in which we solve the problem that we're interested in.

In particular, on the resulting network, we just alter the layers that we are interested and delete the others, then add a couple of dense layers, and on those we'll do the *actual learning*.

A better explanation We learned that the first layers of convolutional networks for computer vision *compute feature maps* of the original image of growing complexity. The filters that have been learned (in particular, the most primitive ones) are *likely to be* ***independent*** *from the particular kind of images they have been trained on.* They have been trained on a huge amount of data and are probably very good. It is a good idea to try to *reuse them* for other classification tasks.

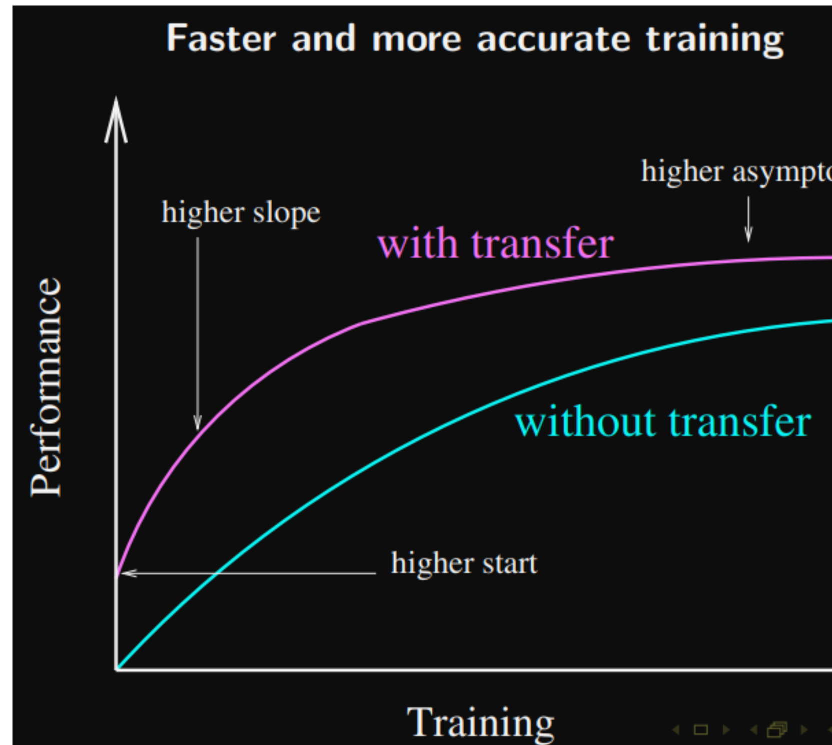
Transferring knowledge from problem A to problem B makes sense if: - the two problems have *"similar" inputs* - we have *much more training data for A* than for B

In all layers, you typically have a *trainable parameter* (which is a boolean), and if it false the learning on this layer freezes.



In this type of learning, we can also do some *fine-tuning* by unfreezing the first part of the network after the dense layer has already been trained, thus training also the first part of the network on the specific data of the transferred network. Finetuning is always dangerous, and there's a risk of overfitting and degrading (possibly) good information.

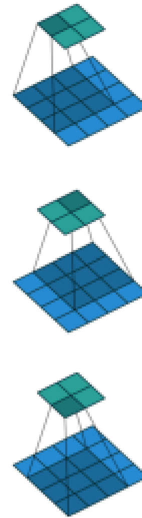
Expectations of transfer learning



Backpropagation for CNNs

To understand how backpropagation works in CNNs, we need to understand how do we have to *change the weights* of the kernel during backpropagation

Since when doing a convolution we are transforming (for example) an input of dimension 4x4 into an output of dimension 2x2.



This can be seen also in the image above. Essentially, what we are trying to do can be also done *through a linear transformation*, involving weights.

Matrix for applying CNNs linearly

This operation, when done in a linear way, involves a *matrix* of this kind:

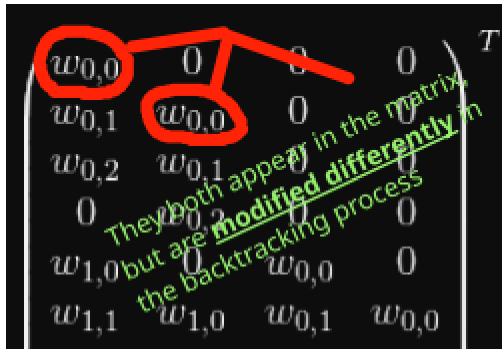
$$\begin{pmatrix}
w_{0,0} & 0 & 0 & 0 \\
w_{0,1} & w_{0,0} & 0 & 0 \\
w_{0,2} & w_{0,1} & 0 & 0 \\
0 & w_{0,2} & 0 & 0 \\
w_{1,0} & 0 & w_{0,0} & 0 \\
w_{1,1} & w_{1,0} & w_{0,1} & w_{0,0} \\
w_{1,2} & w_{1,1} & w_{0,2} & w_{0,1} \\
0 & w_{1,2} & 0 & w_{0,2} \\
w_{2,0} & 0 & w_{1,0} & 0 \\
w_{2,1} &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

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 has been transposed for convenience.

- Since this matrix is very sparse, it is very easy to compute efficiently.
- The weights are also repeated (obviously) since the kernel is repeatly applied onto the same input.

Why are we trying to find this correlation between linear layers and convolutional layers? Well, it's because in linear layers *we know how to apply backpropagation.*

However there's a catch: when we backpropagate, each parameter of the linear matrix is transformed in a different way, so the ==partial derivativative of the loss function for each one of the parameters of the linear layer will be different==.



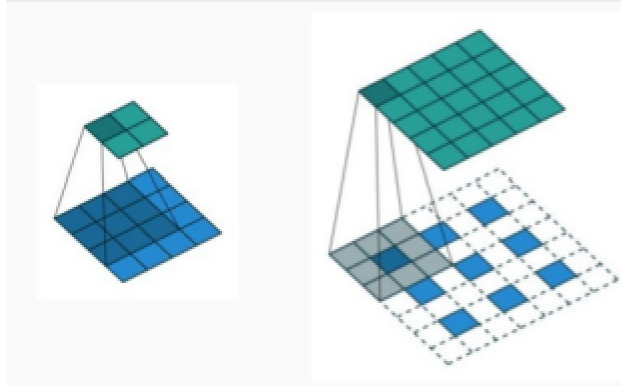
To solve this problem, we update the kernel with the *average* of the updates made to each parameter.

Transposed convolutions

Normal convolutions with non-unitarian (>1) strides downsample the input dimension (usually, for example, to increase their receptive field). 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.

When upsampling, we can also use bilinear transformations. But for now, we'll just focus on transposed convolutions.

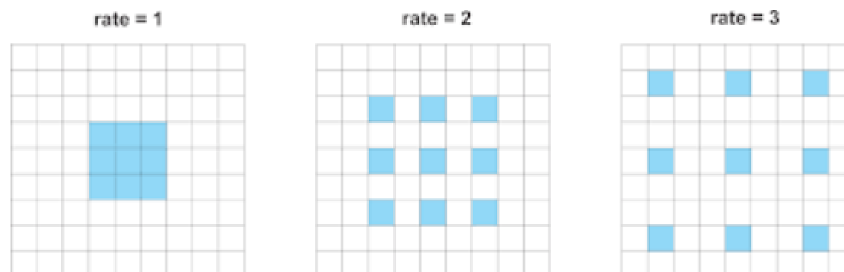
A transposed convolution (sometimes called deconvolution) can be thought as a normal convolution with *subunitarian* stride. To mimic subunitarian stride, we must first properly *upsample the input* (e.g. inserting empty rows and columns) and *then apply a single strided convolution* like in this image:



In this way, we can have an output that is bigger in size than the input.

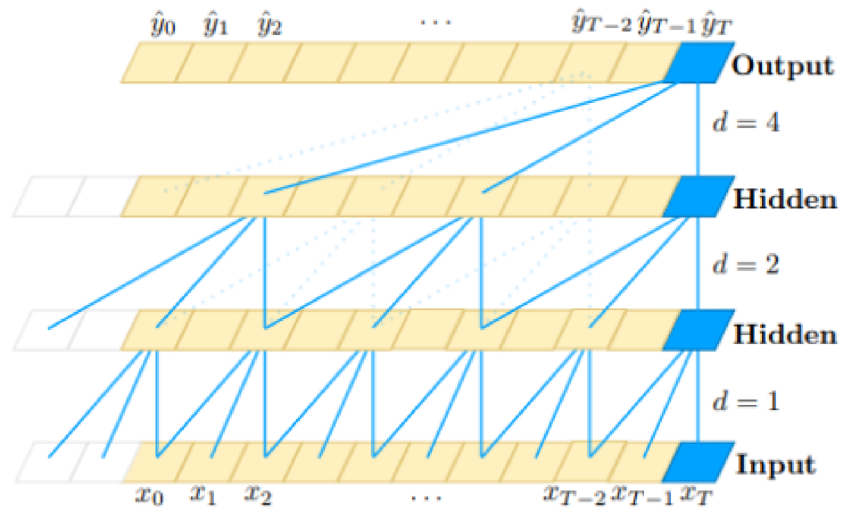
Dilated Convolutions

Sometimes, when applying convolutions, the *kernel may be too small*, and if we increase the size of the kernel, it may become too computationally complex for our needs. 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.

They are also used in **Temporal Conovlutional Networks (TCNs)** to process long input sequences:



> [!WARNING] Remember: - in transpose convolutions, we are dilating the input. - in dilated convolution, we are dilating the kernel.

Normalization layers

Normalization layers allow to renormalize the values after each layer. The potential benefits for this kind of operation are: - have a more *stable* (more controlled activations) and possibly faster training - It allows to solve the NaN output problems which are created by activations that have exploded. - increase the independence between layers

Batch Normalization

Batch normalization operates on a batch of data, so not on the whole dataset, and operates on *single layers*, per *channel base*.

Essentially, we are computing a normalization for each batch, and everytime we find a new batch, we compute a weighted average between the statistics of this new batch and the old ones.

At each training iteration, the input is normalized according to *batch (moving) statistics*, subtracting the *mean* ^{B} and dividing by the *standard deviation* σ^B . - Then, an opposite transformation (denormalization) is applied based on *learned* parameters γ (scale, equivalent to a standard deviation) and β (center, equivalent to a mean). These parameters allow essentially to cancel the normalization operation according to what the network thinks that it's better to do. - This

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

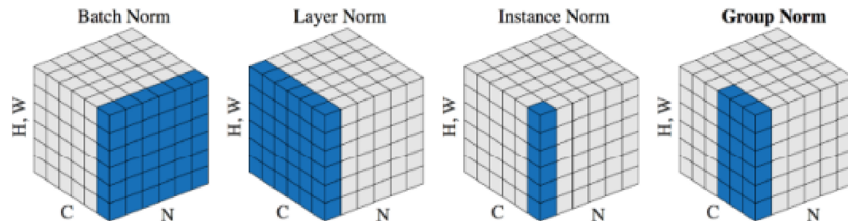
operation can also be disabled.
^{B} and σ^B are the batch statistics.

> [!WARNING] Remember: - Stacking 2 linear layers together in a network is an operation that never makes sense, since a composition of 2 linear layers is just a simple linear layer. So… - if we put a linear layer after a normalization layer that has the learned parameters, it is kind of useless since the operation is basically just learned by the layer. - The only case in which it makes sense is when the layer right next to the normalization layer is an activation layer.

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.

- Batch Norm: we are normalizing along the batch dimension, as we've seen. This is done for each channel of the network.
- Layer Norm: we are normalizing along the channel dimension.
- Instance Norm: we are normalizing along the spatial dimension only.
- Group Norm: we are normalinzing some defined channels, but not all of them (maybe because some of them are more important than others?).