Examples of convolutional neural networks

Javier Zazo

March 3, 2019

Abstract

In this section we will describe some of the successful convolutional networks that have popularized deep learning and their architectures. We will learn from examples good practices that constituted breakthroughs in massive classification tasks. These include LeNet-5, AlexNet, VGG-16 and Inception nets, among others. Furthermore, we will also discuss the the principles that allow to use deeper and deeper layers, describing ResNets, Network-in-network and Dense networks.

1 Overview of Convolutional Networks (CNNs)

We have seen in class the motivations to use convolutional networks, the operations involved, and the layer types that compose them. Some characteristics of CNNs are the following:

- They require less parameters (weights) to learn than a fully connected network.
- They are invariant to object translation and can tolerate some distortion in the images.
- They are very capable of generalizing and learning features from the input domain.

Convolutional networks are constituted by convolutional layers, pooling layers and fully connected layers. We describe them briefly.

1.1 Convolutional layers

Convolutional layers are formed of filters, feature maps and activation functions. Filters (or kernels) perform essentially the convolution in the layer, and their size determine the number of parameters to train the network. If the input layer is an image, the filter will convolve with the image pixels. In deeper layers, the filter will convolve with latent features, or embeddings.

A feature map is the output of a filter applied to an output of a previous layer, Figure 1. Depending on the stride of the feature map (the number of pixels that the filter moves from one sampling of the output to the next one) and the padding of the input layer (the number of zeros added at input layer to control the size of the convolution), the output size is determined. The formula that governs the output size for a *valid convolution* operation is the following:

$$n_{\text{output}} = \left\lfloor \frac{n_{\text{input}} - f + 2p}{s} + 1 \right\rfloor . \tag{1}$$

Here, f refers to the filter size, s to the stride and p to the padding size. Symbol $\lfloor \cdot \rfloor$ indicates floor integer rounding.

The convolutional layer may incorporate several channel filters, whose number constitutes a design decision for every layer. In modern networks the design principles recommend that layers increase the number of channel filters and decrease the size of the input layers as we move deeper into the network.

Let's give an example. Consider an input image in grayscale of 63×63 pixels. If we use 8 filters of size $f = 3$, $s = 1$ and $p = 2$, we end up with an output of $63 \times 63 \times 8$. This is called *same convolution*, because the padding helps the output be the same size as the input. If we used 8 filters with $f = 3$, $p = 0$ and $s = 2$, we would have an output of $30 \times 30 \times 8$. This is called *valid convolution*, and it "halfs" the input layers size. The number of filters (or channels) $n_C = 8$ is decided across layers and is a design characteristic.

Finally, a convolutional layer normally ends with a nonlinear element-wise activation function, such as ReLu's.

1.2 Pooling layers

The pooling layers generally down-sample the previous layer's feature maps, normally presenting a stride value > 1[1]. They also normally follow a se-

[1] This is not always the case, as we will see with ResNets.

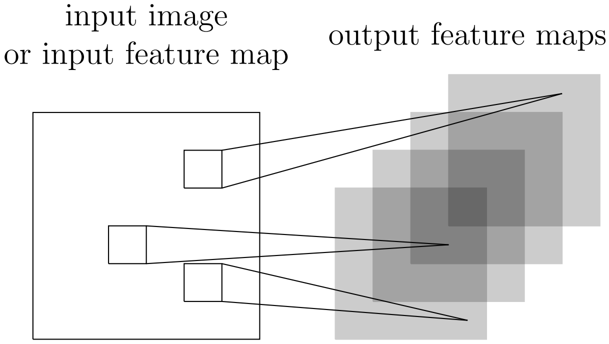


Figure 1: Feature mapping of a convolution.

quence of one or more convolutional layers and are intended to consolidate the features learned and expressed in previous layers. As such, pooling may be considered a technique to compress or generalize feature representations and generally reduce the overfitting of the training data by the model.

Pooling layers are in general very simple, normally taking the maximum or average of the affected inputs. The max-pooling is the default choice for pooling. Note that this layer, does not add optimization parameters to the network.

1.3 Fully connected layers

Fully connected layers (FC) are normal flat feed-forward type of layer. These kind of layers are normally at the end of a convolutional network[2]. To connect these layers with a typical convolutional or pooling layer, their output is normally vectorized, and then connections are created to a subsequent FC layer, or an output layer. These neurons also incorporate nonlinear activation functions typical of feed-forward networks, such as ReLu, sigmoid, tanh or cross-entropy for output validation.

1.4 First example

We consider a very simple convolutional network of four layers. The input to the network 32×32 pixel images with single grey channel. After the input follows a convolutional network with filters of size 5×5 ($f = 5$), we apply

[2] Except when using Network-In-Network ideas, as we'll see later

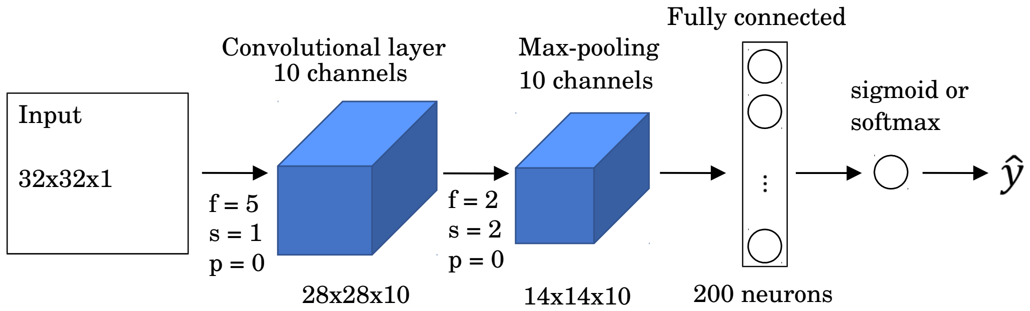


Figure 2: A small neural network with a total of 392,460 parameters.

stride $s = 1$ and add no padding (this corresponds to *valid convolutions*). We consider 10 filters for this layer, which produces $5 \times 5 \times 10 + 10$ with bias weight parameters for this layer. Applying equation Equation (1) the output is $28 \times 28 \times 10$.

The next layer is a max-pooling layer with filter size $f = 2$, stride $s = 2$ and no padding. At this layer we have to use 10 filters because the number of channels is fixed to the input size in pooling layers. This gives an output of size $14 \times 14 \times 10$. There are no parameters to optimize in this layer.

Then, we vectorize the output of the pooling layer and obtain a vector of size 1,960. We add a fully connected layer of 200 neurons, and connect every vector component with every neuron. That incorporates 392,000 weights plus 200 bias terms to be learned.

The network is completed after adding a binary cross-entropy layer, or softmax output neuron for a multiclass classification problem. The network is depicted in Figure 2. There are a total of 392,460 parameters that define the network.

2 Classic Networks

We now present a few neural networks that were successful for certain applications in the deep learning literature. The motivation behind looking at these examples is to help you build your own models learning from successful networks and extrapolate their architectures to your application of interest. A second motivation is to possibly reuse existing architectures and use them in different problems. This is normally regarded as transfer learning, taking

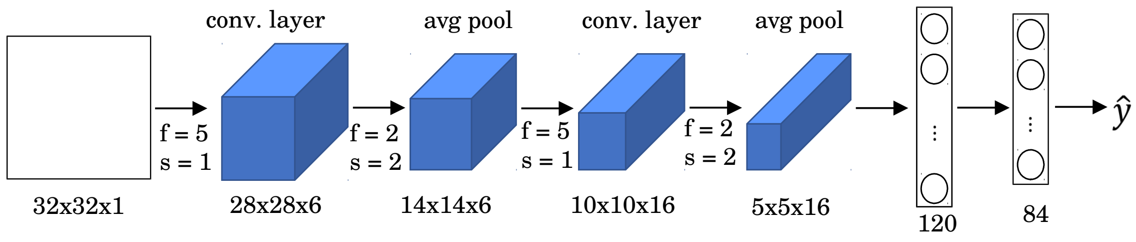


Figure 3: LeNet-5 neural network. Around 60k parameters.

a fully trained network and only retraining the last layer to obtain a different classifier. Finally, by recognizing these networks you will be able to evaluate and assimilate other modern networks from the deep learning literature.

2.1 LeNet-5

This network was presented in [1] and introduced a convolutional network to classify hand written digits. It is depicted in Figure 3. Its formulation is a bit outdated considering current practices, but it follows the idea of using convolutional networks followed by pooling layers and finishing with fully connected layers. Furthermore, it also starts with higher dimensional/spacial features and reduces its size in deeper layers while increasing the number of channels.

This network has in total around 60k parameters. Originally, the final layer did not include a softmax structure but a different classifier, which is now out of use. Additionally, the network did not employ ReLu units as is common nowadays. Nonetheless, it is one of the first modern classifiers that presented high accuracy for digit classification. Current benchmarks of famous databases can be found in [2].

2.2 AlexNet

This network architecture was presented in [3] and was trained to classify 1.2 million high-resolution (227x227x3) images in the ImageNet contest from 2010. The classification problem expanded 1000 different classes, and the network achieved minimum error rates at the time of presentation. The architecture is presented in Figure 4.

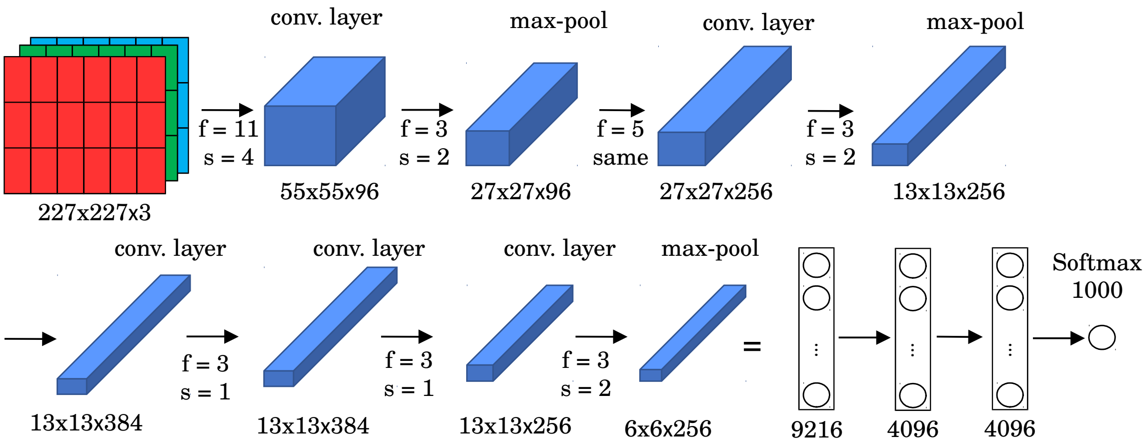


Figure 4: AlexNet neural network. Around 60 million parameters.

This network was much bigger than previous ones, with around 60 million parameters to optimize. It used ReLu units in all layers except on the output layer, employing a softmax unit of size 1000 for all categories. The paper also included a novel procedure to train the network using parallel GPUs, but these details are no longer needed to train modern networks. It used dropout as regularization procedure, and local response normalization (LRN). LRN aims to normalize the values in the channel dimension, so as to limit the number of activating neurons after the ReLu units. The technique is no longer frequently used, but at the time it was thought to help at training.

2.3 VGG-16

VGG-16 stands for “Visual Geometry Group” from Oxford University, who secured first and second positions at the ImageNet Challenge 2014 in the localization and classification tracks, respectively [4]. The network is formed of 16 layers and presents a very procedural scheme, as we will see. The authors also present VGG-19 with 19 layers but performance is comparable with VGG-16. Their network achieves around 25% error rate in the top-1 classification, and around 10% in the top-5 classification categories. The network is depicted in Figure 5.

The 16 layers are formed by sequential convolutional layers, followed by max-pooling layers, and fully connected layers at the end of the network.

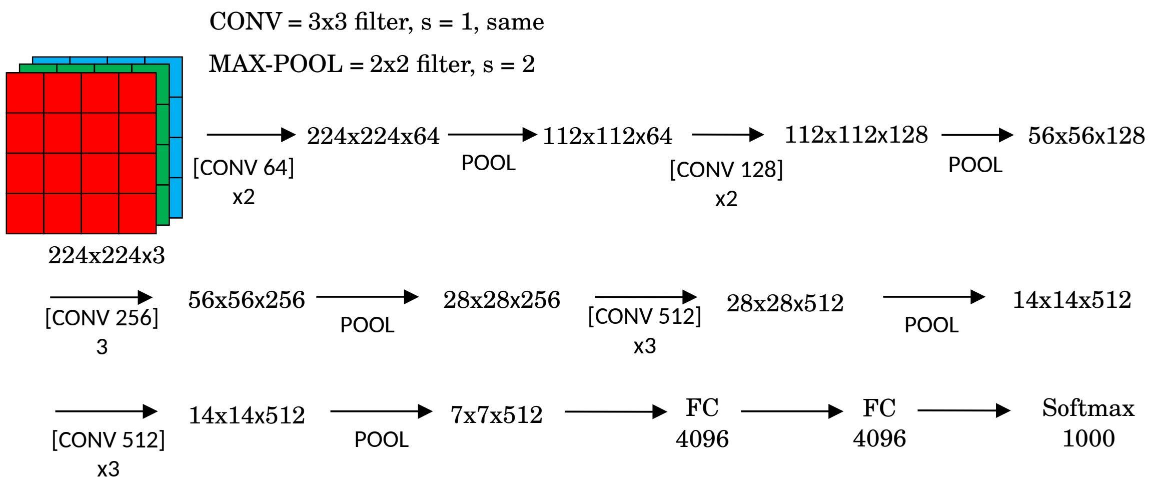


Figure 5: VGG-16. Around 138 million parameters.

Pooling layers do not add to the total count of 16 layers. The network is very easy to characterize because it follows very simple rules. Convolutional layers always use a ‘same’ padding architecture and stride $s = 1$. As a consequence, these layers aim to increase the number of channels in deeper layers but do not reduce the size of the features. On the other hand, max-pooling layers are used after several convolutional layers, with filter sizes $f = 2$ and stride $s = 2$. Therefore, they reduce the size of the features by half systematically and hold the number of channels invariant. Finally, the last network layers consist of two fully connected layers of 4096 neurons, and a softmax function of 1000 elements.

This network is a perfect example of a very systematic way to design a network and obtain state of the art performances. The trained weights of these networks are publicly available for use in different set of applications.

3 Residual networks (ResNets)

Residual networks appeared in [5] as a means to train very deep neural architectures. Their design uses ‘residual blocks’, which bypass a connection from one layer and incorporates it to a subsequent layer several steps ahead in the normal path. The methodology allows to improve the problem of vanishing and exploding gradients for very deep networks. This procedure has helped

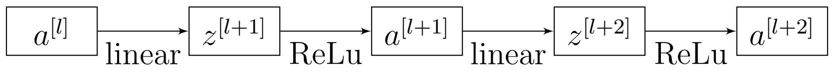


Figure 6: Plain network structure for layers l to $l+2$.

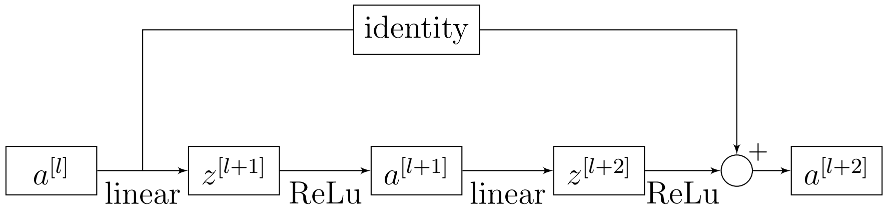


Figure 7: Residual network structure for layers l to $l+2$.

to successfully train architectures over 100 layers.

3.1 Residual block

A plain network with ReLu units has a structure as in Figure 6. The feedforward equations that govern that structure would be:

$$a^{[l]} = g(z^{[l]}) \tag{2a}$$
$$z^{[l+1]} = W^{[l+1]}a^{[l]} + b^{[l+1]} \tag{2b}$$
$$a^{[l+1]} = g(z^{[l+1]}) \tag{2c}$$
$$z^{[l+2]} = W^{[l+2]}a^{[l+1]} + b^{[l+2]} \tag{2d}$$
$$a^{[l+2]} = g(z^{[l+2]}), \tag{2e}$$

where $g(\cdot)$ would correspond to the non-linear activation function.

The residual block adds a 'shortcut' path, or 'skips' a connection as shown in Figure 7. The equations in this case become:

$$a^{[l]} = g(z^{[l]}) \tag{3a}$$
$$z^{[l+1]} = W^{[l+1]}a^{[l]} + b^{[l+1]} \tag{3b}$$
$$a^{[l+1]} = g(z^{[l+1]}) \tag{3c}$$
$$z^{[l+2]} = W^{[l+2]}a^{[l+1]} + b^{[l+2]} \tag{3d}$$
$$a^{[l+2]} = g(z^{[l+2]} + a^{[l]}), \tag{3e}$$

where now (3e) differs from (2e). The idea is that with this extra connection, gradients can travel backwards more easily, while the block learns other

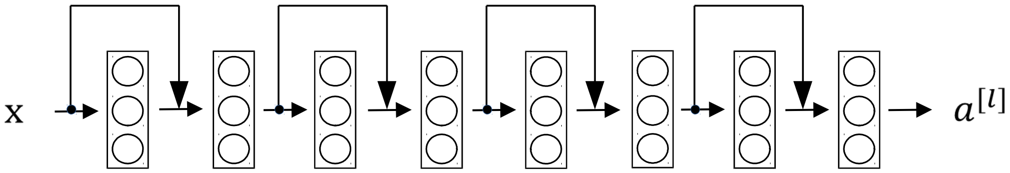


Figure 8: Residual network structure for layers l to $l+2$.

features.

We can also see that from a learning perspective, the residual block can very easily learn the identity function by setting $W^{[l+2]} = 0$ and $b^{[l+2]} = 0$. In such case, $a^{[l+2]} = g(a^{[l]}) = a^{[l]}$ for ReLu units, and the extra layer block did not hinder the learning capabilities of the larger network. In fact, it becomes a flexible block that can expand the capacity of the network, or simply transform into a identity function that would not affect training.

3.2 Residual networks

A residual network stacks residual blocks sequentially. The idea is to follow a scheme as in Figure 8. This will allow the network to become deeper without increasing the training complexity. The reason is that when training plain networks, there is a point in which the training error starts to increase after a certain number of layers because of the inefficiency of the gradients. ResNets help to overcome this problem. The idea of the error curve is depicted in Figure 9. Note also that the ResNet curve may reach a plateau after a certain number of layers, because residual blocks start learning the identity function and stop reducing the training error.

These kind of networks implement some of the blocks with convolutional layers that use 'same' padding option. This allows the block to learn the identity function. On the other hand, the designer may want to reduce the size of features and use 'valid' padding. In such case, the shortcut path can implement a new set of convolutional layers that reduces the size appropriately. An example of a network with 34 layers from [5] is shown in Figure 10. In this example, the authors use both types of 'valid' and 'same' padding. Finally, the error rates achieved with a ResNet network of size 152 layers is on Table 1.

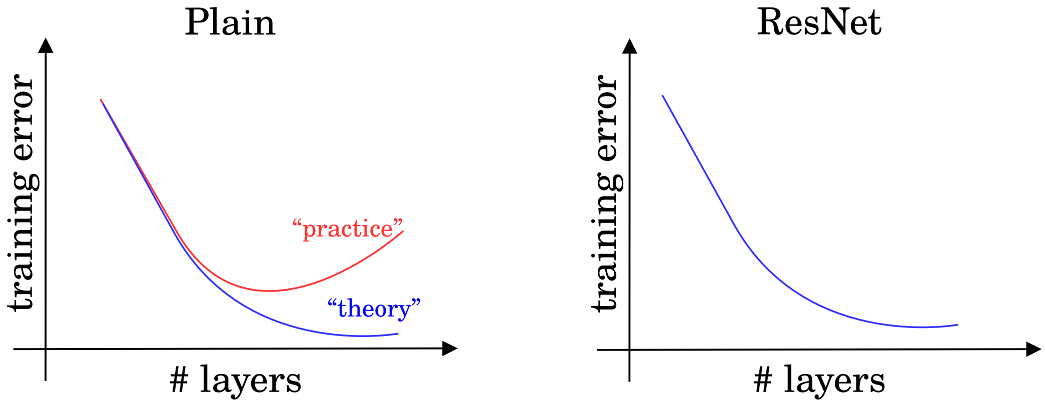


Figure 9: Training error curves vs. the number of layers in plain networks and ResNets.

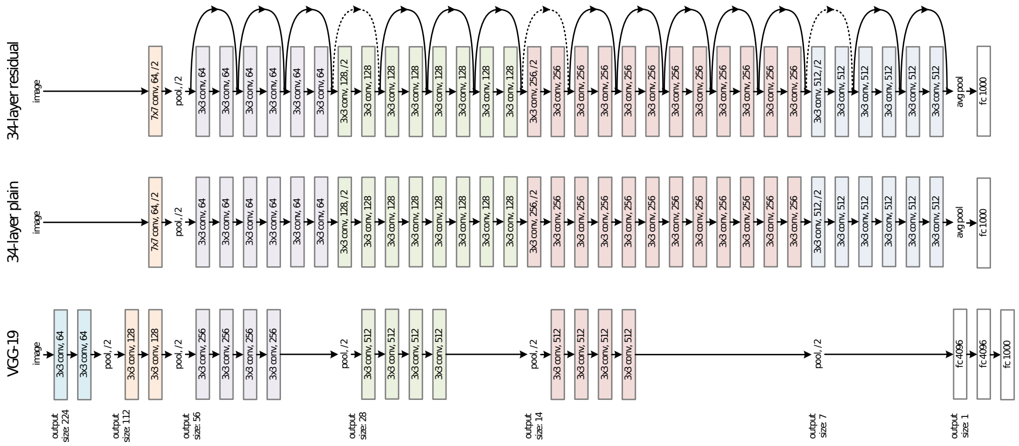
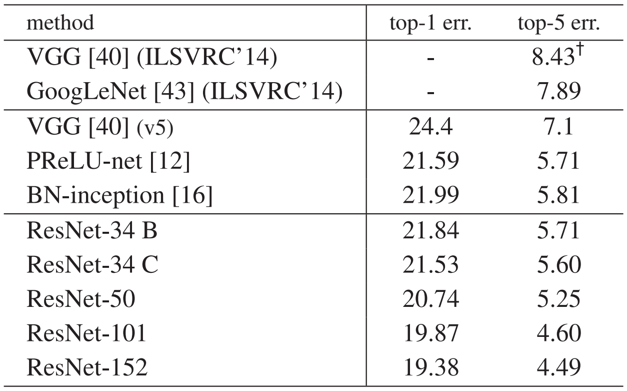


Figure 10: ResNet of 34 layers proposed in [5] and compared to VGG.



method	top-1 err.	top-5 err.
VGG [40] (ILSVRC'14)	-	$8.43^{\dagger}$
GoogLeNet [43] (ILSVRC'14)	-	7.89
VGG [40] (v5)	24.4	7.1
PReLU-net [12]	21.59	5.71
BN-inception [16]	21.99	5.81
ResNet-34 B	21.84	5.71
ResNet-34 C	21.53	5.60
ResNet-50	20.74	5.25
ResNet-101	19.87	4.60
ResNet-152	19.38	4.49

Table 1: Error rates on the ImageNet validation set as presented in [5].

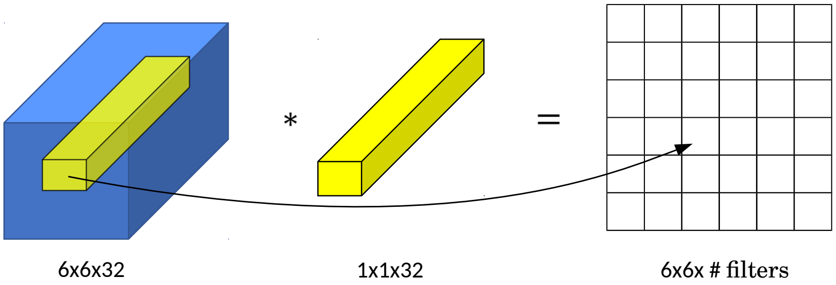


Figure 11: 1×1 convolution. The filter has size $1 \times 1 \times 32$ elements (weights). The number of filters correspond to the number of channels of the output.

4 Network in network

The concept of network-in-network was proposed in [6] and has been quite influential in the deep learning literature. The operation performed by this layer is also called 1×1 convolution, as we will see shortly. The idea is depicted in Figure 11. In a linear convolution, a filter is used and each element of the input is multiplied element-wise with the filter. If the input had two dimensions, the 1×1 convolution would correspond to a scalar multiplication. However, if the input has a greater number of channels (say, 32), the convolutional filter will have $1 \times 1 \times 32$ elements, which corresponds to a more complicated operation than before. Finally, a rectified linear operation is applied to the output.

This operation can be repeated with several filters, which will correspond to the total number of channels of the tensor in the output. Therefore, the

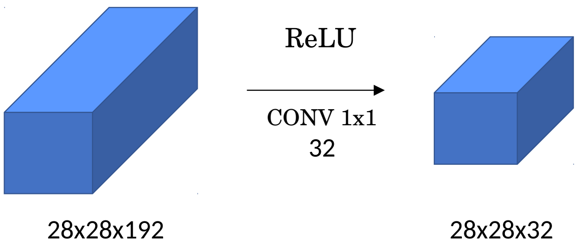


Figure 12: Example of 1×1 convolution within a network to reduce the number of channels.

1×1 convolution operation can be employed to reduce the number of channels of the input, while holding the feature sizes unchanged. An example is given in Figure 12, where the input features are reduced from 192 channels to 32.

This serves two purposes: i) reduce the number of trainable parameters in subsequent layers; ii) reduce overfitting and act as a regularization technique. Think of 1×1 convolution as a type of pooling among channels, where channels are combined (or cancelled if a weight becomes null) as the network is forced to reduce its capacity.

A final note regarding the transformation of convolutional layers into fully connected layers. One way to see such transformation is to vectorize the input into a column vector and then use a FC layer as usual. An alternative, is to use a filter of the same size as the input, i.e., if the input is of size $5\times5\times16$, a filter of size $5\times5\times16$ weights would have a $1\times1\times1$ output. If we stack these filters together, we can get an output of $1\times1\times n_C$, equivalent to using a FC with n_C neurons. A 1×1 convolution can be regarded as a fully connected layer if it goes after a standard fully connected layer. These transformations are seen occasionally in the deep learning literature.

5 Inception networks

The motivation behind inception networks is to use more than a single type of convolutional layer at every layer. They were presented in [7]. The idea is to use 1×1, 3×3, 5×5 convolutional layers, and max-pooling layers all in parallel. Such structure complicates the architecture of the network, but it performs really well in practice, as we can see in Table 1.

The inception module has the following structure, depicted in Figure 13. It incorporates 64 filters of a 1×1 convolution, 128 filters of a 3×3, 32 filters

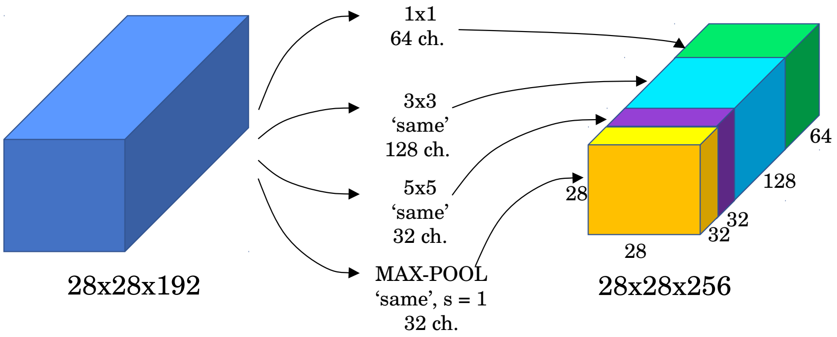


Figure 13: Inception module with 1 × 1, 3 × 3, 5 × 5 convolutional layers, and max-pooling.

of a 5×5 convolution, and 32 filters of max-pooling; all of them using 'same' padding to obtain a $28 \times 28 \times 256$ output image. The authors from [7] refer to this module, however, as a nave implementation. The reason is that having so many convolutions per layer, increases the number of operations significantly and, therefore, propose a computationally less intensive alternative.

The less computational expensive idea implements intermediate 1×1 convolutions that reduce the size of the channel dimension before performing the convolution with the 3×3 or 5×5 filters. Additionally, the number of channels can vary from the input to the output, acting as a bottleneck and reducing overfitting. After the max-pooling layer, another 1×1 convolution layer is required to adjust the number of channels. The efficient implementation of the inception module is presented in Figure 14.

Finally, the inception network is formed by concatenating other inception modules. Figure 15 shows the GoogLeNet network with complete inception modules, as well as several softmax output units to enforce regularization.

6 DenseNets

DenseNets constitute the next iteration after ResNets to achieve deeper networks that can still be successfully trained without suffering from vanishing gradients. The goal is to allow maximum information flow between lower layers, and deeper ones. In order to do so, feature maps from lower layers are stacked to the output of higher ones, allowing feature reuse and avoiding redundant learning.

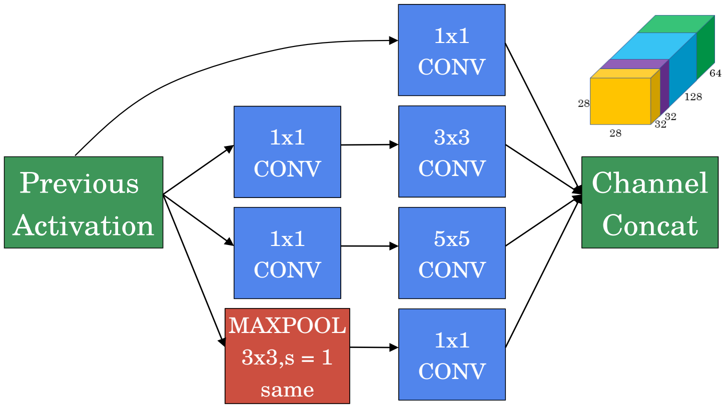


Figure 14: Inception module with 1×1, 3×3, 5×5 convolutional layers, and max-pooling with intermediate 1×1 convolutions.

Because of feature reuse, DenseNets normally incorporate very few filters on increasing layers (think of filters as the number of output channels), so they incorporate fewer learning parameters than ResNets, where each block has a large number of filters. It has been observed that ResNets are inefficient in learning filters, since many of them can be dropped without affecting performance. DenseNets aim to exploit this characteristic by using less number of filters.

Traditional feedforward or convolutional networks feed the ouput of a layer to the next. DenseNets connect every output within a block to the next layers and stack channel outputs together, as depicted in Figure 16. In mathematical form, you can write the following operation:

$$a^{[l]} = g(W^{[l]}[a^{[0]}, a^{[1]}, \dots, a^{[l-1]}] + b^{[l]}) \tag{4}$$

Basically, the filters at layer l stack the chanells of previous layers and use such features at will to implement the task at hand.

Such operation requires that the spacial dimensions of all feature maps within a block remain constant across layers, but the number of filters (channels) is nonetheless increased. Such increment is called the **growth rate**, and is linear growth assument that each layer adds k new filters every time:

$$k^{[l]} = k^{[0]} + k \cdot (l - 1). \tag{5}$$

Finally, layers that reduce the size dimensions between DenseNet blocks

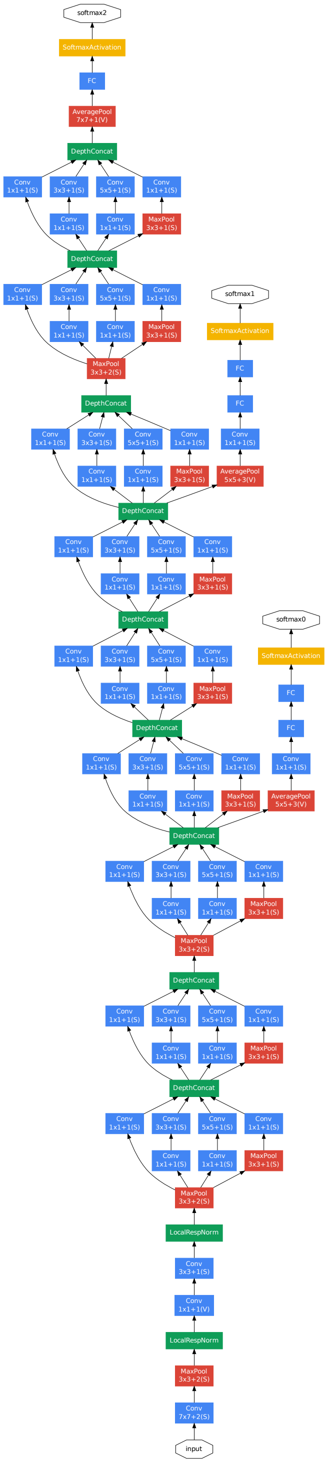
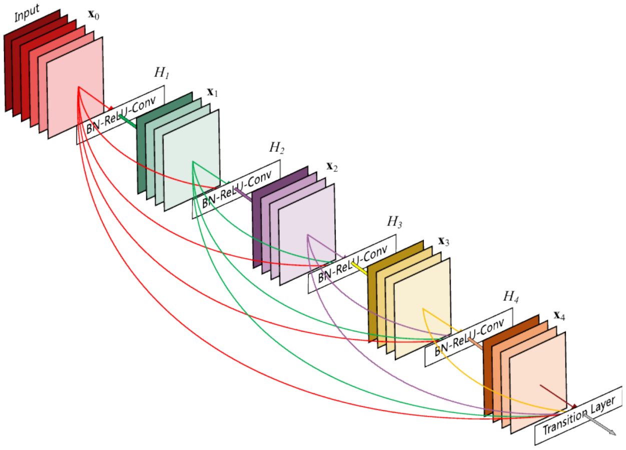


Figure 15: GoogLeNet network with all the bells and whistles [7].

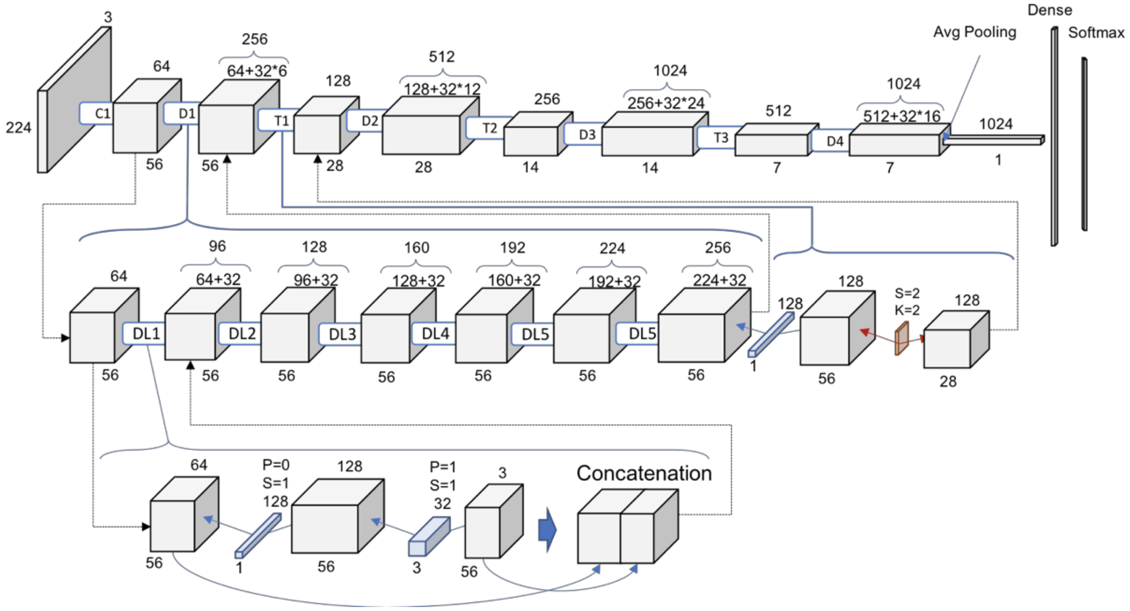
Figure 16: DenseNet general block architecture.



are called **transition layers**, using batch normalization, 1×1 convolutional blocks and pooling layers.

The full DenseNet architecture is given in Figure 17. The first line includes the DenseNet blocks (indicated with D1, D2, etc.) and Transition blocks (indicated with T1, T2, etc.) depicting the whole network. The bottom number within a block indicates the spacial dimension of the block, and the upper number the channels output. Since it constitutes a block of 6 layers and growth rate 32, we get a growing rule following (5). The second line of Figure 17 shows how Dense blocks are built, using Dense layers and ending with a 1×1 convolutional layer and pooling. Finally, the third line within the figure represents how the channels are concatenated together, as we have explained before.

Figure 17: Full DenseNet architecture and building blocks.



References

[1] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998, doi:10.1109/9780470544976.ch9.

[2] Rodrigo Benenson, "Classification datasets results," 2016, `http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html`.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105, doi:10.1145/3065386.

[4] Karen Simonyan and Andrew Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, arXiv:1409.1556.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, "Deep residual learning for image recognition," in *The IEEE Conference*

on Computer Vision and Pattern Recognition (CVPR), June 2016, doi:10.1109/CVPR.2016.90.

[6] Min Lin, Qiang Chen, and Shuicheng Yan, “Network in network,” 2013, arXiv:1312.4400.

[7] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich, “Going deeper with convolutions,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9, doi:10.1109/CVPR.2015.7298594.

Acknowledgments

Figures 3 to 5, 8, 9 and 11 to 13 where adapted from deeplearning.ai. Figure 10 and Table 1 from [5]. Figures 14 and 15 where adapted from [7]. Figure 1 was composed by David Stutz and is licensed under the BSD 3-Clause License.