Convolutional Neural Networks

Danilo Comminiello

Neural Networks 2023/2024

November 2, 2023

Sapienza
Università di Roma

Table of contents

1 Operations in Convolutional Layers

The Cross-Correlation Operator
Paddin and Stride
Convolutional Layers with Multiple Channels
Pooling

Representation of a convolutional layer

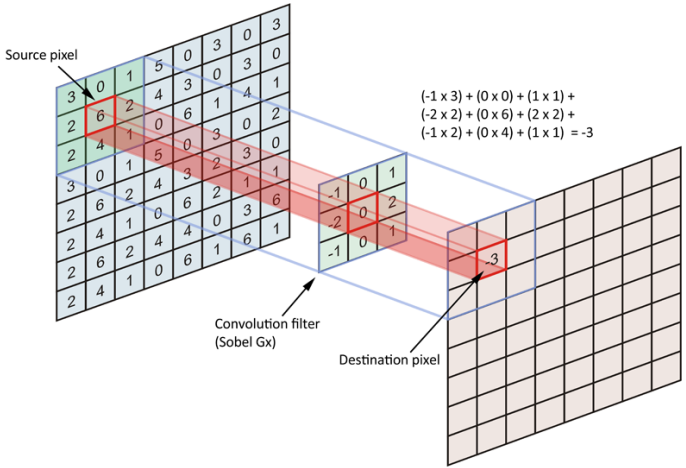

Figure 1: Graphical visualization of a convolutional layer. Image source: Stack Exchange. An animation of the 2D convolution can be found here.

The cross-correlation operator

In a convolutional layer, an *input array* and a *correlation kernel array*, also called *kernel* or *filter* are combined to produce an output array through a **cross-correlation operation**.

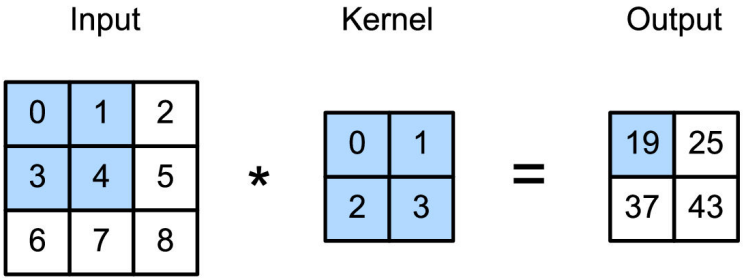

Figure 2: The input is a 3×3 2D array, and the kernel array is 2×2. The shape of the kernel window (also known as the *convolution window*) is given precisely by the height and width of the kernel. It is possible to notice the correspondence between cross-correlation and convolution. The latter can be easily obtained by simply flipping the kernel from the bottom left to the top right [1].

Learning a kernel

The parameters of a convolutional layer are precisely the values that constitute the kernel and the scalar bias.

As we look at large kernels, and consider successive layers of convolutions, it might be impossible to specify precisely what each filter should be doing manually.

Thus, we need to learn the parameters of the convolutional layers.

When training the models based on convolutional layers, we typically initialize the kernels *randomly*, just as we would with a fully-connected layer.

Output shape of a convolutional layer

The output shape of the convolutional layer is determined by the shapes of input and convolution kernel window, as shown in Fig. 2.

In some cases we might want to control the size of the output:

- In general, kernels generally have width and height greater than 1, thus, after applying many successive convolutions, the output results much smaller than the input. **Padding** handles this issue.
- In some cases, we want to reduce the resolution drastically if we find an original input resolution to be unwieldy. **Strides** can help in these instances.

Padding

The padding method is used to avoid an excessive loss of pixels and consists in adding extra pixels of filler around the boundary of our input image, thus increasing the effective size of the image.

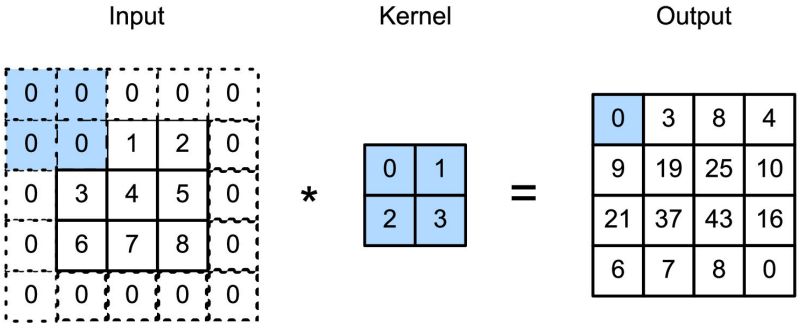

Figure 3: 2D cross-correlation with padding. We pad a 3×5 input, increasing its size to 5×7. The corresponding output then increases to a 4×6 matrix. Typically, we set the values of the extra pixels to 0 [1].

Odd kernel size

Usually, convolutional kernels are chosen with odd height and width values, such as 1, 3, 5, or 7.

Odd kernel sizes has the benefit that we can:

- preserve the spatial dimensionality,
- padding with the same number of rows on top and bottom, and the same number of columns on left and right.

Stride I

When computing the cross-correlation, by default we slide the kernel one pixel at a time.

However, sometimes, either for computational efficiency or because we wish to downsample, we move our window more than one pixel at a time, thus skipping the intermediate locations.

We refer to the number of rows and columns traversed per slide as the **stride**, which thus can be larger than 1.

Stride II

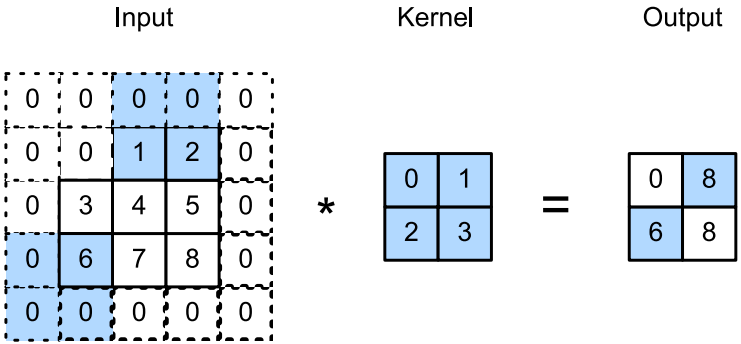

Figure 4: 2D cross-correlation with a stride of 3 vertically and 2 horizontally. When the second element of the first column is output, the convolution window slides down three rows. The convolution window slides two columns to the right when the second element of the first row is output. When the convolution window slides three columns to the right on the input, there is no output because the input element cannot fill the window (unless we add another column of padding). The shaded portions are the output element and the input and core array elements used in its computation: $0 \times 0 + 0 \times 1 + 1 \times 2 + 2 \times 3 = 8$, $0 \times 0 + 6 \times 1 + 0 \times 2 + 0 \times 3 = 6$ [1].

Cross-correlation with multiple input channels

When the input data contains multiple channels, we need to construct a convolution kernel with the same number of input channels as the input data, so that it can perform cross-correlation with the input data.

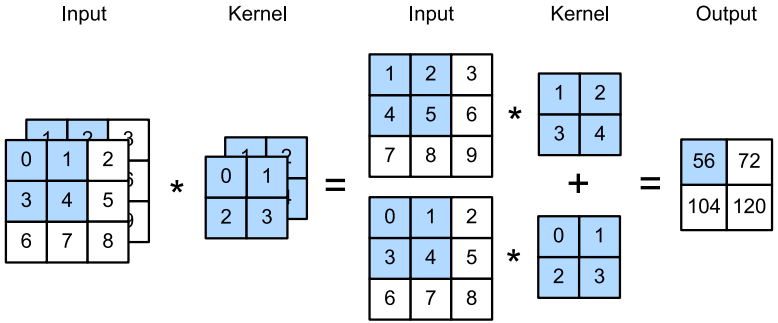

Figure 5: Cross-correlation computation with 2 input channels. The shaded portions are the first output element as well as the input and kernel array elements used in its computation: $(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56$ [1].

Cross-correlation with multiple output channels

Often, we may need to have multiple output channels since representations are not learned independent but are rather optimized to be jointly useful.

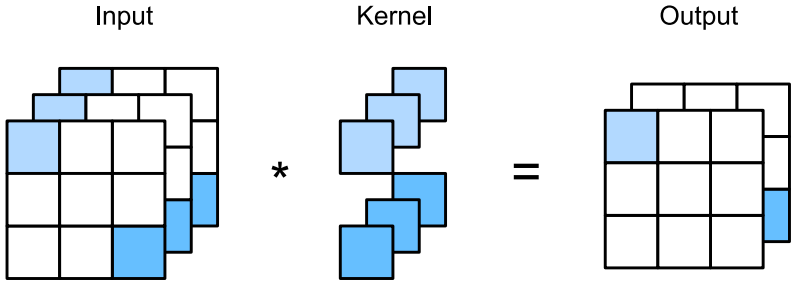

Figure 6: The cross-correlation computation uses the 1×1 convolution kernel with 3 input channels and 2 output channels. The inputs and outputs have the same height and width. The 1×1 convolutional layer is equivalent to the fully-connected layer, when applied on a per pixel basis. It is also typically used to adjust the number of channels between network layers and to control model complexity [1].

Aggregating information

Often, as we process images, we want to gradually reduce the spatial resolution of our hidden representations to increase depth.

Often our ultimate task asks some global question about the image, so typically the nodes of the final layer should be sensitive to the entire input.

By gradually aggregating information, we accomplish this goal of ultimately learning a global representation, while keeping all of the advantages of convolutional layers at the intermediate layers of processing.

Moreover, when detecting lower-level features, such as edges, we often want our representations to be somewhat invariant to translation.

Pooling operators I

Like convolutional layers, **pooling operators** consist of a fixed-shape window that is slid over all regions in the input according to its *stride*, computing a single output for each location traversed by the fixed-shape window (or *pooling window*).

The pooling layer contains no parameters (there is no *filter*).

Instead, pooling operators are *deterministic*, typically calculating either the *maximum* or the *average* value of the elements in the pooling window.

These operations are called maximum pooling (or also max pooling) and average pooling, respectively.

Pooling operators II

At each location that the pooling window hits, it computes the *maximum or average value* of the input subarray in the window.

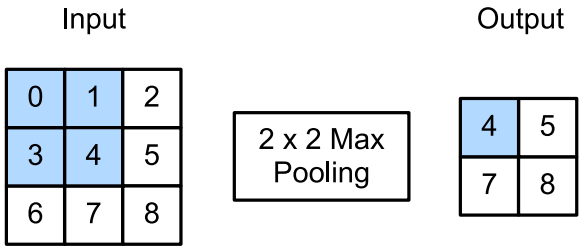

Figure 7: Maximum pooling with a pooling window shape of 2×2. The shaded portions represent the first output element and the input element used for its computation: $\max(0, 1, 3, 4) = 4$ [1].

As with convolutional layers, pooling layers can also change the output shape by padding the input and adjusting the stride.

Representation of pooling approaches

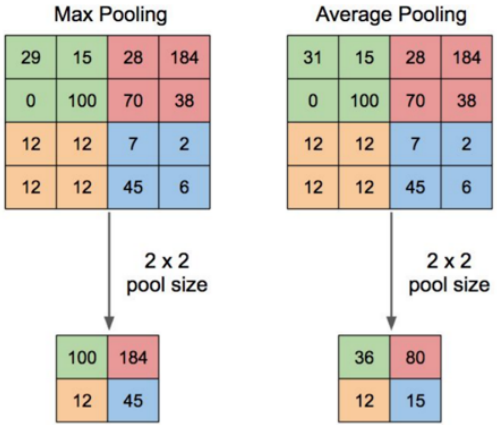

Figure 8: Visualization of max-pooling on a 4×4 image with windows of size 2×2. Note that the maximum operation can be replaced with any differentiable aggregation (e.g., average).

Dilated convolutions I

A simple modification of the convolution operator is the dilated convolution (or originally called *convolution with dilated filters*), which expands the receptive field without loss of resolution.

The expansion of the filter allows to increase its dimensions by *filling the empty positions* with zeros.

Considering the 2D case, let d be a dilation factor, given a filter kernel $h[i, j]$, the receptive field can be widen by rewriting the convolution expression as:

$$y[i, j] = \sum_{k_1=0}^{M_1-1} \sum_{k_2=0}^{M_2-1} h[k_1, k_2]\, x[i - d \cdot k_1, j - d \cdot k_2]$$

that is referred to as dilated convolution.

Dilated convolutions II

Practically, the number of parameters associated with each layer is identical, while the receptive field grows exponentially as a power of two.

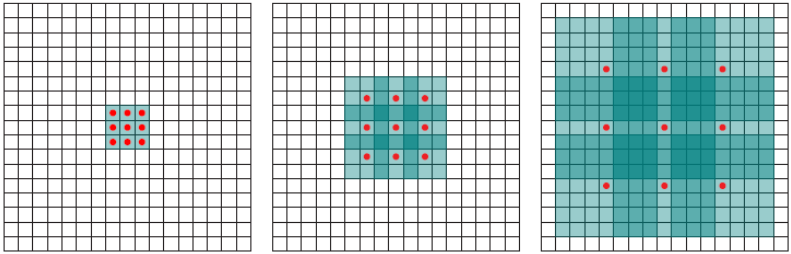

Figure 9: Representation of receptive fields resulting from a dilated convolution. Image source: TowardsDataScience.

2 Convolutional Neural Network Architectures

Complete CNN Architecture

The LeNet5

Complete architecture of a CNN

We are now ready to put all of the tools together to deploy a fully-functional convolutional neural network, according to the following procedure:

- First, interleave multiple convolutional / max-pooling layers;
- Vectorize (*flatten*) the output of the last layer;
- Process the resulting vector with one or more fully-connected layers to obtain the final classification vector.

More recent CNNs add many variations on this basic architecture.

LeNet5: the first CNN

LeNet5 is one of the first published CNNs, by Yann Lecun in 1998 [2], for the purpose of recognizing handwritten digits in images.

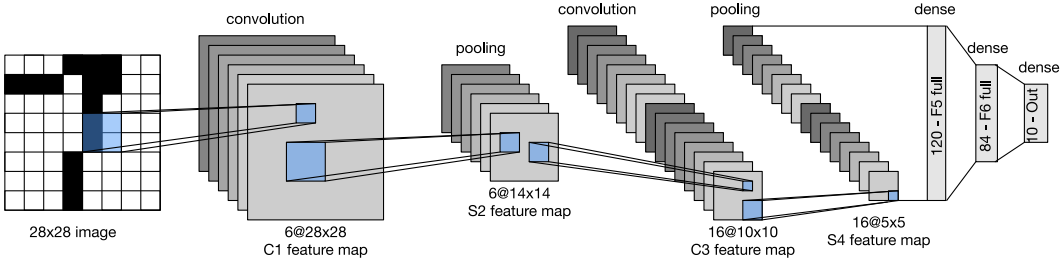

Figure 10: Data flow in LeNet5, consisting of a block of convolutional layers and a block of fully-connected layers. The input is a handwritten digit, the output a probability over 10 possible outcomes. The convolutional layer is used to recognize the spatial patterns in the image, such as lines and the parts of objects, and the subsequent average pooling layer is used to reduce the dimensionality [1].

CNN in action

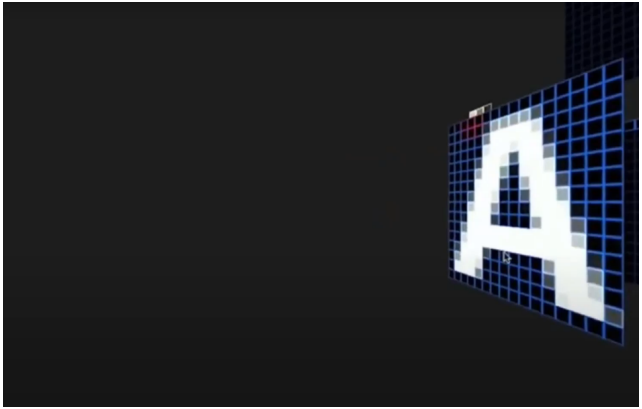

Figure 11: Click on this link to see a convolutional neural network in action.

3 Improving the Classic CNN Pipeline

Improving the Classic CNN Pipeline
Challenges in Training Neural Networks

Classic CNN pipeline

Although LeNet [2] achieved good results on early *small datasets*, the performance and feasability of training convolutional networks on larger, more realistic datasets had yet to be established.

Rather than training *end-to-end* systems (e.g., pixel to classification), classical pipelines looked more like this:

1. Obtain an interesting dataset.
2. Preprocess the dataset with hand-crafted features.
3. Feed the data through a standard set of feature extractors.
4. Dump the resulting representations into your favorite classifier, likely a linear model or kernel method.

Missing ingredients for deep learning

The ultimate breakthrough for deep learning in 2012 can be attributed to two key factors:

- Availability of **data**.
  - Deep models with many layers require large amounts of data in order to enter the regime where they significantly outperform traditional methods.
  - In 2009, ImageNet Challenge dataset pushed machine learning research forward providing 1 million of examples to train models.

- Availability of **hardware**.
  - Deep learning models are voracious consumers of compute cycles.
  - The computational bottlenecks in CNNs were solved by parallelizing convolutions and matrix multiplications on multiple GPUs.

Challenges in training neural networks

Training deep neural nets and getting them to converge in a reasonable amount of time can be *tricky*.

In particular, some **practical challenges** arise when training machine learning models and neural networks:

1. Choices regarding data preprocessing often make an enormous difference in the final results.
2. Activations in intermediate layers may take values with *widely* varying magnitudes. This alertdrift in the distribution of activations could hamper the convergence of the network.
3. Deeper networks are complex and easily capable of overfitting. This means that regularization becomes more critical.

4 Learning Representations: The AlexNet Model

Algorithms or data?

Learning representations

AlexNet Architecture

Reasons for a success

Algorithms or data?

Before 2012, many attempts were made to improve performance results of convolutional neural networks, by adopting different approaches.

On one hand, machine learning researchers believed the progress would have been driven by thriving and rigorous learning theories and algorithms.

On the other hand, computer vision researches believed the progress would have been driven by bigger o cleaner datasets or by a slightly improved feature-extraction pipeline.

The truth was somewhere in between. Indeed, the progress began when it became clear that features themselves (i.e., *representations*) ought to be learned.

Learning representations

The process of applying a set of feature extractors to input data to obtain a representation was the most important part of a classical pipeline.

Until 2012, this process was calculated *mechanically*.

Then, it was deemed that to be reasonably complex, the features ought to be hierarchically composed with multiple jointly learned layers, each with learnable parameters.

The idea was to build models capable of capturing different levels of representations.

To this end, Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton proposed a new variant of a CNN, **AlexNet** [3], that achieved excellent performance in the 2012 ImageNet challenge.

Learning representations by AlexNet I

In the case of an image, the lowest layers might come to detect edges, colors, and textures.

Interestingly, in the lowest layers of the **AlexNet** [3], the model learned feature extractors that resembled some traditional filters.

Higher layers in the network might build upon these representations to represent larger structures, like eyes, noses, blades of grass, etc.

Ultimately, the final hidden state learns a compact representation of the image that summarizes its contents to separate different data easily.

Learning representations by AlexNet II

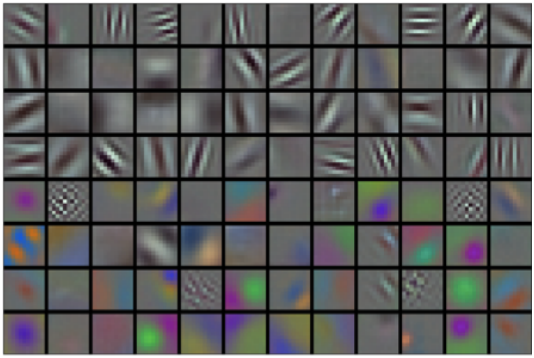

Figure 12: Image filters learned by the first layer of AlexNet [3]. They are able to detect edges, colors, and textures.

AlexNet architecture

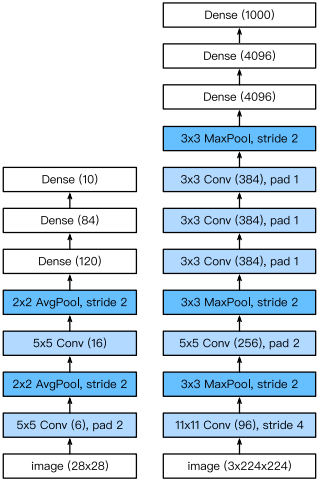

Figure 13: The architectures of AlexNet and LeNet are very similar, but AlexNet is much deeper (eight layers) and used the ReLU activation function instead of the sigmoid. AlexNet proved, for the first time, that the features obtained by learning can transcend manually-design features [1].

Reasons for the success of AlexNet: data

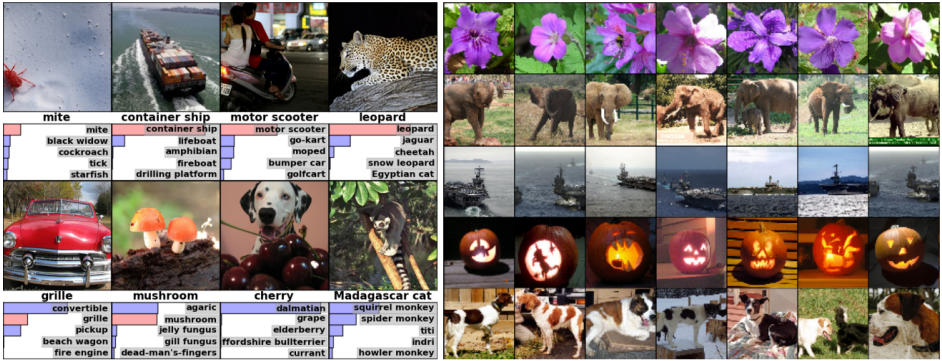

Figure 14: Deep models with many layers require large amounts of data in order to enter the regime where they significantly outperform traditional methods based on convex optimizations (e.g., linear and kernel methods). In 2009, the ImageNet dataset was released, challenging researchers to learn models from 1 million examples, 1000 each from 1000 distinct categories of objects. AlexNet was the first CNN model to obtain record accuracy on the ImageNet LSVRC competition, fueling interest in deep learning. It exploited a few innovations including a new activation function (rectified linear unit), dropout regularization, and GPU parallelization during training [3].

Reasons for the success of AlexNet: hardware I

Deep learning models are voracious consumers of compute cycles.

Training can take hundreds of epochs, and each iteration requires passing data through many layers of computationally-expensive linear algebra operations.

This is one of the main reasons why in the 1990s and early 2000s, simple algorithms based on the more-efficiently optimized convex objectives were preferred.

Graphical processing units (GPUs) proved to be a game changer in making deep learning feasible.

These chips were optimized for high throughput 4×4 matrix-vector products (for many computer graphics tasks), which was fortunately sufficient to calculate convolutional layers.

Reasons for the success of AlexNet: hardware II

Back to 2012, a major breakthrough came when Alex Krizhevsky and Ilya Sutskever implemented a deep CNN that could run on GPU hardware.

They realized that the computational bottlenecks in CNNs, convolutions and matrix multiplications, are all operations that could be parallelized in hardware.

Using two NVIDIA GTX 580s with 3GB of memory, they implemented fast convolutions.

The code cuda-convnet was good enough that for several years it was the industry standard and powered the first couple years of the deep learning boom.

Impact on real applications

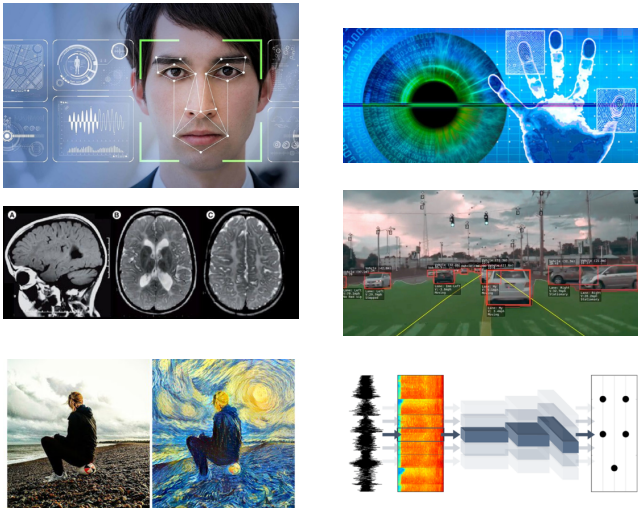

Figure 15: Examples of application in which convolutional neural networks can be adopted.

Next lecture

- We will discuss how to use advanced features on convolutional neural networks.
- Some of the most popular modern *deep convolutional neural networks* will be presented.

References

[1] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive Into Deep Learning*. 0.7.1 ed., 2020.

[2] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, pp. 1097–1105, 2012.

[4] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, (Boston, MA), pp. 1–9, June 2015.

[5] Y. Bengio, "Deep architectures for AI," *Foundations and Trends in Machine Learning*, vol. 2, pp. 1–127, Nov. 2009.

Convolutional Neural Networks

Neural Networks 2023/2024

Danilo Comminiello

https://sites.google.com/uniroma1.it/neuralnetworks2023

{danilo.comminiello, simone.scardapane}@uniroma1.it