telecom saint-étienne
école d'ingénieurs
nouvelles technologies

A Short Introduction to Deep Learning and Computer Vision

Loïc Denis
2025–2026

These lecture notes were created to support an M.Sc. course taught at Telecom Saint-Etienne, Université Jean Monnet, France. They provide a gentle introduction to the basics of deep learning and some of its applications to computer vision. Given that the course is quite short (9 hours of lectures and 12 hours of tutorials), choices were made to focus on some selected themes. Many topics are not covered in this introduction and left for more advanced courses.

Please, provide feedback to help improve it (typos, errors, additional references or examples)!

History of the document:
– original version: May-September, 2025

Contributors:

- Loïc Denis (original version)
- Olivier Flasseur (corrections)
- Thomas Bultingaire (corrections)
- Corinne Fournier (corrections)

The image on the cover is a very-high resolution image captured by the French satellite Pleiades Neo ©ESA in July 2023 with 30cm spatial resolution (Saint-Aubin harbor, Neuchâtel Lake, Switzerland). This image has been processed by the model Yolo11x-obb to show the oriented bounding boxes of objects detected as "boats". Each object detected is shown with a box with a different color. This corresponds to a typical task in computer vision, solved efficiently using a deep neural network. YOLO models can be downloaded here: `https://www.ultralytics.com`. Even though the boats are only a few pixels wide, they are almost perfectly detected (the original image has been upsampled so that objects are large enough to allow their detection). Note two errors: the pier outside the harbor is mistaken for a boat and a boat has been detected twice (once with a correct orientation, a second time with an incorrect orientation). Computing the result on this 1024×736 image took less than ½ second on a laptop.

Contents

1 Introduction

Deep learning is without doubt a major technological evolution in computer science. The foundations of artificial neural networks date back from the second half of the twentieth century. Yet, the incredible success of neural networks began in the 2010s thanks to the combination of several factors: (i) the availability of very large training datasets; (ii) an increase of the computational capability thanks to the repurposing of graphical processing units to parallelize the core operations of deep neural networks; (iii) progress in the design of the networks and the training strategies (thanks to research in machine learning and computer vision).

This course aims at covering the basics of deep learning and presenting the field of computer vision. It starts with a chapter that introduces the multi-layer perceptron and the methodology to apply deep learning to a practical problem. Chapter 3 then discusses the principles of neural networks training. Next, we present convolutional neural networks, first introduced to solve computer vision problems such as recognizing handwritten digits for postal services, and used nowadays to solve a wide range of problems such as protein unfolding. The last chapter describes three applications of deep learning in computer vision: image classification, object detection, and semantic segmentation.

Several topics are left for further courses, in particular the presentation of the larger domain of *machine learning*, advanced concepts such as recurrent neural networks, transformers, graph neural networks, generative models, natural language processing, biases in AI, privacy issues, distributed learning...

Many important papers are included in the bibliography, at the end of this document. Readers should try to read some of them to see how research results on the topic of deep learning and computer vision are communicated.

The curious reader will find a more detailed presentation as well as many additional topics in the numerous books, online courses, or blogs available on the internet. I would recommend taking a closer look at the following books:

- "Deep learning", by Ian Goodfellow, Yoshua Bengio, and Aaron Courville (Goodfellow et al., 2016), which offers a great introduction to deep learning and covers also many more advanced topics;
- "Dive into Deep Learning", by Aston Zhang, Zachary Lipton, Mu Li, and Alexander Smola Zhang et al., 2023, which gives many interesting insights. It gives both a presentation of the main ideas and some code examples to make things more concrete.
- "Deep learning for vision systems", by Mohamed Elgendy (Elgendy, 2020), which offers a nice introduction to deep learning and computer vision.

2 Multi-Layer Perceptrons

The multi-layer perceptron (MLP) is the simplest generic form of deep neural networks. It defines a parametric mapping $f_{\boldsymbol{\theta}}$ from an input $\boldsymbol{x}$ (a vector of dimension d_{in}) to an output $\boldsymbol{y}$ (a vector of dimension d_{out}):

$$\begin{aligned} \mathbb{R}^{d_{\text{in}}} &\longrightarrow \mathbb{R}^{d_{\text{out}}} \\ \boldsymbol{x} &\mapsto \boldsymbol{y} = f_{\boldsymbol{\theta}}(\boldsymbol{x}) \,. \end{aligned} \tag{2.1}$$

The vector $\boldsymbol{\theta}$ collects the value of all the parameters of the network.

⚙ Concrete example: an MLP for handwritten digit recognition

Figure 2.1 illustrates an MLP that takes as input an image $\boldsymbol{x}$ of a handwritten digit (the N-pixels image is fed to the network as an $\mathbb{R}^N$ vector) and outputs predicted probabilities of the digit being a "0", a "1", . . . , a "9". In this illustration, the MLP is performing well since a very high probability is predicted for the correct value "3". Here, the dimension of the output vector $\boldsymbol{y}$ is 10. An alternative could be to directly output a scalar y equal to 0, or 1, . . . , or 9. It turns out that it is slightly more difficult to find good parameters $\boldsymbol{\theta}$ for the network in this latter case[a]. Representing an integer value k between 1 and K in the form of a vector v in $\mathbb{R}^K$ with $[v]_i = 0$ if $i \neq k$ and $[v]_k = 1$ is called "one-hot encoding". It is often used in classification problems.

[a] note that if $f_{\boldsymbol{\theta}}$ is a good function to predict the probability of each class, computing $\sum_{d=1}^{d_{\text{out}}} (d-1)[f_{\boldsymbol{\theta}}(\boldsymbol{x})]_d$ gives a way to turn those probabilities into a scalar (the expectation of the digit value). Yet, finding the weights $\boldsymbol{\theta}$ from scratch is slightly less easy when minimizing the discrepancy between this sum and the ground truth values.

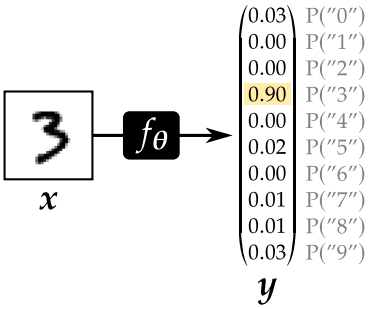

Figure 2.1: A multilayer perceptron applied to handwritten digit recognition.

2.1 Definition of the MLP artificial neural net

The definition of the parametric mappings $f_{\boldsymbol{\theta}}$ takes inspiration from biology (see Fig.2.2):

- artificial neurons accumulate incoming information from several other neurons (biological neurons collect electric signal from their dendrites and soma),
- this incoming information is positively or negatively weighted (synaptic signal is either excitatory or inhibitory),
- a non-linear function is used to produce threshold effects (when the threshold potential is reached, an action potential propagates along the axon: the neuron is firing),
- the outgoing signal is sent to downstream neurons (the electrical signal from biological neurons is sent along the axon to other neurons of the nervous system).

Note that, beyond these similarities, artificial neural networks are not meant to simulate their biological counterparts, but rather engineering considerations drive their design.

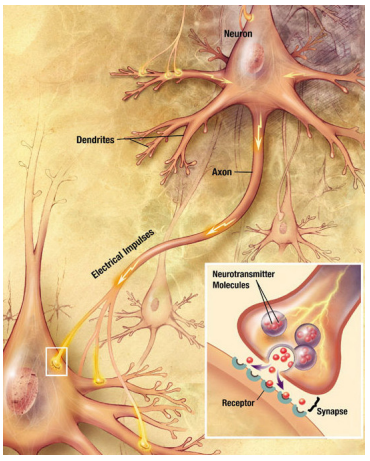

Figure 2.2: Signal propagation from one neuron to the next.
Illustrator: Christy Krames, MA, CMI, for US National Institutes of Health, National Institute on Aging.

MLPs use a simple architecture, called *feedforward*, where information flows from the input to the output of the network (there are no feedback loops*). The functions $f_{\boldsymbol{\theta}}$ are defined through the composition of several simpler functions:

$$f_{\boldsymbol{\theta}} : \boldsymbol{x} \mapsto \boldsymbol{y} = f^{(H)}_{\boldsymbol{\theta}_H}\left(\dots f^{(2)}_{\boldsymbol{\theta}_2}\left(f^{(1)}_{\boldsymbol{\theta}_1}(\boldsymbol{x})\right)\right). \tag{2.2}$$

The stacking of these functions gives rise to a multi-layer structure: each function $f^{(i)}_{\boldsymbol{\theta}_i}$, from $i = 1$ to H, defines a successive layer and H is the depth of the network. The intermediate values $\boldsymbol{h}_1 = f^{(1)}_{\boldsymbol{\theta}_1}(\boldsymbol{x}) \in \mathbb{R}^{d_1}$, $\boldsymbol{h}_2 = f^{(2)}_{\boldsymbol{\theta}_2}\left(f^{(1)}_{\boldsymbol{\theta}_1}(\boldsymbol{x})\right) = f^{(2)}_{\boldsymbol{\theta}_2}(\boldsymbol{h}_1) \in \mathbb{R}^{d_2}, \dots, \boldsymbol{h}_{H-1} = f^{(H-1)}_{\boldsymbol{\theta}_{H-1}}(\boldsymbol{h}_{H-2}) \in \mathbb{R}^{d_{H-1}}$ are called *hidden values* since they correspond to internal representations that are not specified beforehand when a new task is considered (i.e., these representations of the inputs are built during the learning process).

The functions $f^{(1)}_{\boldsymbol{\theta}_1}$ to $f^{(H)}_{\boldsymbol{\theta}_H}$ of an MLP take the following parametric form:

$$f^{(i)}_{\boldsymbol{\theta}_i} : \boldsymbol{h}_{i-1} \mapsto \boldsymbol{h}_i = \underline{g}_i(\mathbf{W}_i\boldsymbol{h}_{i-1} + \boldsymbol{b}_i) = \begin{pmatrix} g_i\left([\mathbf{W}_i]_{1,:}\,\boldsymbol{h}_{i-1} + [\boldsymbol{b}_i]_1\right) \\ g_i\left([\mathbf{W}_i]_{2,:}\,\boldsymbol{h}_{i-1} + [\boldsymbol{b}_i]_2\right) \\ \vdots \\ g_i\left([\mathbf{W}_i]_{d_i,:}\,\boldsymbol{h}_{i-1} + [\boldsymbol{b}_i]_{d_i}\right) \end{pmatrix}, \tag{2.3}$$

with the convention $\boldsymbol{h}_0 = \boldsymbol{x}$ for the first function $f^{(1)}_{\boldsymbol{\theta}_1}$. The matrix $\mathbf{W}_i \in \mathbb{R}^{d_i \times d_{i-1}}$ is referred to as the *weights* of the i-th layer and the column vector $\boldsymbol{b}_i \in \mathbb{R}^{d_i}$ as the *bias* of the layer. All the weights and bias of layer i form the vector of parameters $\boldsymbol{\theta}_i$. The notation $[\mathbf{W}_i]_{k,:}$ represents the k-th row of matrix $\mathbf{W}_i$.

The function $\underline{g}_i : \mathbb{R}^{d_i} \to \mathbb{R}^{d_i}$ defined in equation (2.3) is often separable†: it corresponds to the separate application of the same function $g_i : \mathbb{R} \to \mathbb{R}$ to each component:

$$\underline{g}_i(\boldsymbol{u}) = \begin{pmatrix} g_i([\boldsymbol{u}]_1) \\ g_i([\boldsymbol{u}]_2) \\ \vdots \\ g_i([\boldsymbol{u}]_{d_i}) \end{pmatrix}. \tag{2.4}$$

Functions $\underline{g}_i$ and g_i are called *activation functions* and are non-linear (or skipped altogether, i.e., equal to the identity).

To better understand the interpretation of function $f_{\boldsymbol{\theta}}$ as an artificial neural network, we represent the operations involved to obtain each component of the hidden vector $\boldsymbol{h}_i$ from the previous hidden vector $\boldsymbol{h}_{i-1}$ in Figure 2.3. Each component of the output hidden vector corresponds to the output of a different artificial neuron. Each of these neurons collect

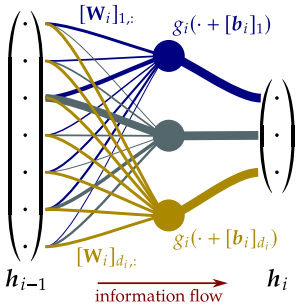

Figure 2.3: The mapping $f^{(i)}_{\boldsymbol{\theta}}$ from one hidden vector to the next can be seen as the action of d_i artificial neurons represented here in different colors.

* this contrasts with how neurons are connected in biological brains, as revealed by connectome reconstructions on insects, using electron microscopy (Winding et al., 2023)

† a notable exception is the *soft-max* activation presented in the sequel; some works suggest using tunable activation functions (parametric activations with parameters tuned during learning) or activation functions that operate on several variables at the same time (e.g., to implement the complex modulus and argument from a pair of reals interpreted as the real and imaginary parts of a complex number), but this is not a common practice

information from all the components of the input hidden vector, by summing the hidden values with weights specific to each connection (i.e., each component of the input and each neuron). These weights can amplify or inhibit the different input values (large weights are represented by thick lines in the Figure and small weights by thin lines). The non-linear function g_i typically produce a threshold effect: the signal may be blocked/damped when the weighted sum of incoming values is below the threshold, or forwarded to the next layer.

The composition of several functions $f_{\boldsymbol{\theta}_i}^{(i)}$ in equation (2.2) to form the parametric mapping $f_{\boldsymbol{\theta}}$ leads to several layers of artificial neurons, see Figure 2.4. Multi-layer perceptrons are *feed-forward networks* since the information flows from one layer to the next, without looping back. There exist other network architectures where some feedback connections are used: *recurrent neural networks* (not covered in this introductory course).

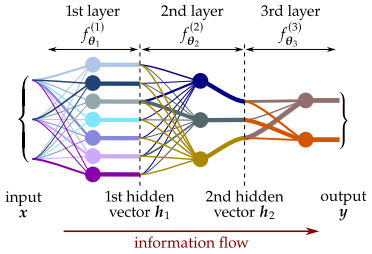

Figure 2.4: The composition of functions $f_{\boldsymbol{\theta}_i}^{(i)}$ corresponds to the stacking of several layers of artificial neurons.

To summarize the complexity of an MLP, the following quantities are generally used:

- the total number of parameters,
- the *depth* of the network, corresponding to the number of layers (H),
- the *width* of the network, corresponding to the dimension of the largest hidden vector $\boldsymbol{h}$.

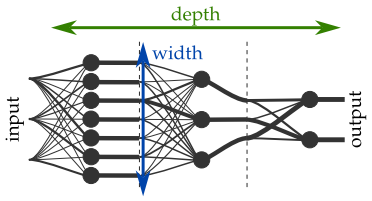

Figure 2.5: The depth and width of a MLP summarize its complexity

⚙ Exercise: complexity of an MLP

What are the depth and width of the MLP represented in Fig. 2.4? What is the total number of parameters of this MLP?

⚙ Exercise: linear MLP

If all activation functions g_i correspond to the identity map, give the expression of the map $f_{\boldsymbol{\theta}}$. Show that, for given input and output dimensions d_{in} and d_{out}, any linear MLP is equivalent to a linear MLP with no hidden layer.

Note that the affine model $f : \boldsymbol{x} \mapsto \mathbf{W}\boldsymbol{x} + \boldsymbol{b}$ can be rewritten in the form of a linear model by adding an element equal to 1 to the input vector $\boldsymbol{x}$, i.e., by defining $f : \begin{pmatrix}\boldsymbol{x}\\1\end{pmatrix} \mapsto \begin{pmatrix}\mathbf{W} & \boldsymbol{b}\end{pmatrix}\begin{pmatrix}\boldsymbol{x}\\1\end{pmatrix}$

2.2 Interpretation as a non-linear feature-extractor

Linear models $f : \boldsymbol{x} \mapsto \mathbf{W}\boldsymbol{x} + \boldsymbol{b}$ map an input vector of explanatory variables $\boldsymbol{x}$ into the response variables $\boldsymbol{y}$. These models are easy to fit to some data, in particular when minimizing the sum of squared differences: see linear least squares in the Estimation course. Yet, for many tasks, such models are too limited. Rather than directly applying a linear model to $\boldsymbol{x}$, a usual strategy consists of computing first a vector of features $\phi(\boldsymbol{x})$, then applying a linear model to the features. The features can be induced by a kernel in kernel methods[‡], built based on expert knowledge[§], or *learned from the data* in the case of deep neural networks.

[‡] ϕ is such that the similarity $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ corresponds to the dot product $\phi(\boldsymbol{x}_i)^\top \phi(\boldsymbol{x}_j)$, see the course on Machine Learning

[§] e.g., to classify different types of bacteria in microscopy, a biologist may suggest to measure the elongation of the bacteria in the image to distinguish *cocci* from *bacilli*

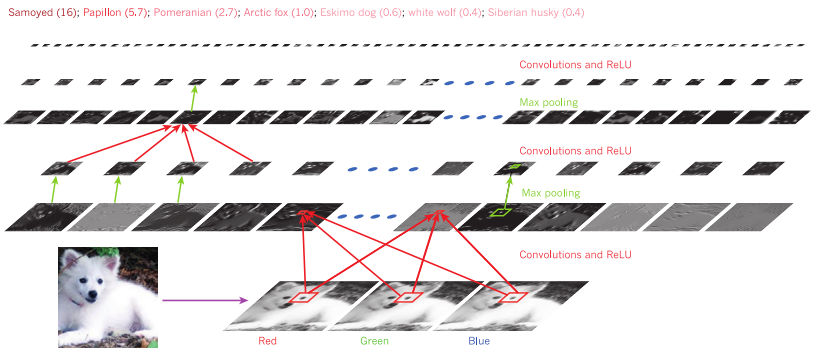

Figure 2.6: Each layer of this MLP trained for image classification extracts feature maps from the preceding layer. The final layer determines a score associated to each class among the 1000 classes the network was trained to recognize. Source: LeCun et al., 2015

If the activation function $\underline{g}_H$ of the last layer of the MLP is the identity, then the MLP mapping $f_{\boldsymbol{\theta}}$ may be seen as a linear model applied to the features vector $\boldsymbol{h}_{H-1}$ (the last hidden vector of the network). The different hidden vectors of the MLP are intermediate representations of the input $\boldsymbol{x}$ that progressively make the estimation of the output $\boldsymbol{y}$ easier, until the features $\phi(\boldsymbol{x}) = f^{(H-1)}_{\boldsymbol{\theta}_{H-1}}\left(\ldots\left(f^{(2)}_{\boldsymbol{\theta}_2}\left(f^{(1)}_{\boldsymbol{\theta}_1}(\boldsymbol{x})\right)\right)\right)$ are obtained (i.e., the last hidden vector $\boldsymbol{h}_{H-1}$), and the output can simply be defined as an affine transform of these features. MLPs learn to extract suitable features from the input vectors $\boldsymbol{x}$ to perform well on the targeted task. The different layers of MLPs provide a way to solve a task in a hierarchical manner: each layer extracts some clues from the preceding hidden vector. First layers generally concentrate on low-level elements (i.e., concepts close to the data such as a jump in a signal, an edge in an image, a periodic pattern). When we advance to deeper layers, more evolved concepts can be captured, see Fig.2.6 and 2.7.

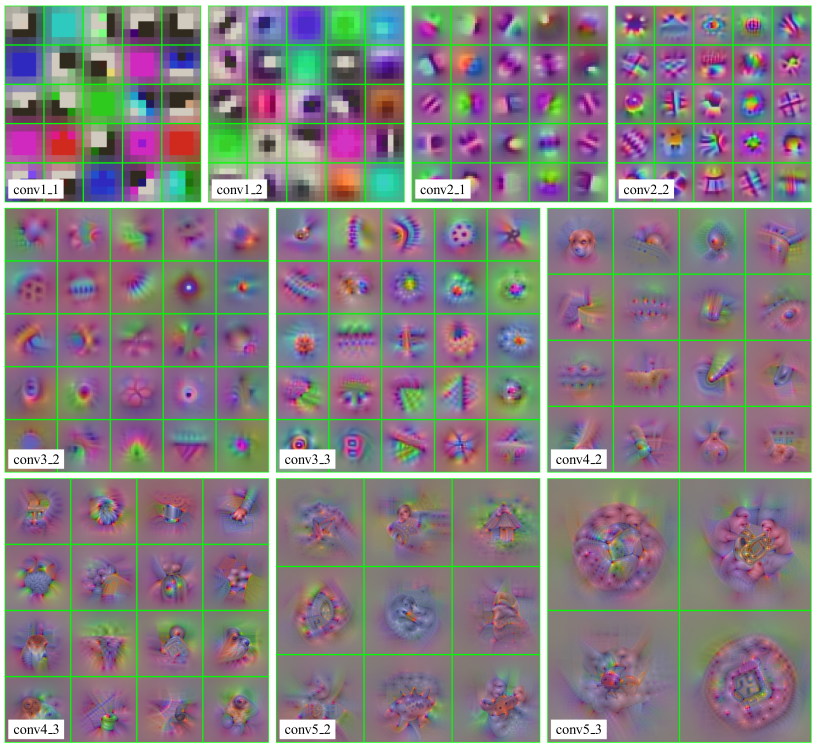

Figure 2.7: The images shown here produce very large values of hidden vectors for layers of increasing depth from left to right and from top to bottom. The MLP used has a special structure (matrices $\mathbf{W}_i$ correspond to spatial convolutions with small kernels) and was trained to classify images, see Simonyan and Zisserman, 2014. Note how first layers respond to simple structures such as edges, while deeper layers are strongly activated by organized/complex structures. Source: Mahendran and Vedaldi, 2016

2.3 Representation power of MLPs

Given the simple mathematical definition of Multi-Layer Perceptrons, one may wonder if they are flexible enough to represent even the complex mappings required to solve difficult tasks. More precisely, if a mapping $f^\star : \mathbb{R}^{d_{\text{in}}} \to \mathbb{R}^{d_{\text{out}}}$ solves perfectly our task (e.g., $f^\star(\boldsymbol{x})$ corresponds to the output that the best human experts would give to any input $\boldsymbol{x}$), can this mapping be represented by an MLP, i.e., can we always find $\boldsymbol{\theta}$ such that $f_{\boldsymbol{\theta}} \approx f^\star$? The short answer is yes, for a wide enough (or deep enough) MLP. More formal answers are given below in the form of *universal approximation theorems*.

Let us start by a result regarding *shallow networks*, i.e., networks with only a single hidden vector (the depth is equal to 2).

Theorem 2.3.1 (Universal approximation of single hidden-layer networks, Leshno et al., 1993)
Let $f^\star$ be a continuous mapping from $\mathbb{R}^{d_{in}}$ to $\mathbb{R}^{d_{out}}$, then, for any compact K in $\mathbb{R}^{d_{in}}$, and any continuous, non-polynomial, separable activation function $\underline{g}_1$,

$$\forall \epsilon > 0,\ \exists d_1 \in \mathbb{N},\ \exists \boldsymbol{b}_1 \in \mathbb{R}^{d_1},\ \exists \mathbf{W}_1 \in \mathbb{R}^{d_1 \times d_{in}},\ \exists \mathbf{W}_2 \in \mathbb{R}^{d_{out} \times d_1},$$
$$\sup_{\boldsymbol{x} \in K} \Big\| \underbrace{\mathbf{W}_2\, \underline{g}_1(\mathbf{W}_1 \boldsymbol{x} + \boldsymbol{b}_1)}_{f_{\boldsymbol{\theta}}(\boldsymbol{x})} - f^\star(\boldsymbol{x}) \Big\| < \epsilon.$$

A direct consequence of the theorem is that an MLP with a single hidden vector can approximate arbitrarily well any continuous mapping provided that the width d_1 of the network be chosen accordingly and that the (separable) activation function of the hidden neurons[¶] be continuous but non-polynomial (which is verified by all activation functions commonly used).

There are many other universal approximation results and this topic is still actively researched[‖]. We give below a theorem that applies to MLPs with fixed width[**]:

Theorem 2.3.2 (Universal approximation of fixed-width MLPs, Park et al., 2021)
Let $f^\star$ be a mapping in $L^p(\mathcal{X}, \mathcal{Y})$, with $\mathcal{X} \subseteq \mathbb{R}^{d_{in}}$ the domain and $\mathcal{Y} \subseteq \mathbb{R}^{d_{out}}$ the codomain, then for any MLP $f_{\boldsymbol{\theta}}$ of width $w \geq \max\{d_{in} + 2, d_{out} + 1\}$ using continuous, non-polynomial, separable activation functions $\underline{g}$ such that

[¶] note that the activation function of the output layer is just the identity and that no bias terms are necessary on that last layer

[‖] the results are generally stated by showing that the set of all parametric functions $f_{\boldsymbol{\theta}}$ for a given MLP size is dense in a given function space; the approximation rate is also of interest, i.e., how quickly the approximation error decreases when the size of the network is increased

[**] reminder:
- a continuously differentiable function is a function that is differentiable and whose derivative is continuous
- the function space L^p contains functions f such that $|f|^p$ is summable

g is continuously differentiable at some z with $g'(z) \neq 0$,

$$\forall \epsilon > 0,\ \exists H \in \mathbb{N},\ \exists \boldsymbol{\theta} = \{\mathbf{W}_1, \boldsymbol{b}_1, \ldots, \mathbf{W}_H, \boldsymbol{b}_H\},\ \left\| f_{\boldsymbol{\theta}} - f^\star \right\|_p < \epsilon$$

with $\mathbf{W}_1 \in \mathbb{R}^{w \times d_{in}}$, $\mathbf{W}_h \in \mathbb{R}^{w \times w}$ for all $1 < h < H$, $\mathbf{W}_H \in \mathbb{R}^{d_{out} \times w}$, $\boldsymbol{b}_h \in \mathbb{R}^w$ for all $h < H$, and $\boldsymbol{b}_H \in \mathbb{R}^{d_{out}}$.

This means that, by using an MLP with a proper depth and a width at least comparable to the dimension of the input and output vector dimensions, any function $f^\star$ can be approximated arbitrarily well ($f^\star$ and the MLP $f_{\boldsymbol{\theta}}$ can only differ on a null set, i.e., a set that has measure zero).

In summary, MLPs can be very expressive provided that enough parameters are used. In practice, for a given number of parameters, deeper MLP architectures usually perform better than wider ones.

2.4 How to apply an MLP to a machine learning problem?

MLPs can be applied to many different problems. Their most straightforward use is in the context of *supervised learning*, which involves the following steps:

- collecting N pairs of input data and desired output vectors (the ground-truths) $\{(\boldsymbol{x}_1, \boldsymbol{y}_1^\star), (\boldsymbol{x}_2, \boldsymbol{y}_2^\star), \ldots, (\boldsymbol{x}_N, \boldsymbol{y}_N^\star)\}$,
- defining an *objective function* (also called a *training function* or *loss function*) $\mathscr{L}(\boldsymbol{y}, \boldsymbol{y}^\star)$ to reward outputs of the MLP $\boldsymbol{y}$ that are close to the reference $\boldsymbol{y}^\star$,
- selecting an appropriate network depth and the dimension of each hidden vector, i.e., specifying the *network architecture* (number of artificial neurons and the way they are connected),
- choosing an activation function $\underline{g}_i$ for each layer,
- *training* the network by finding a set of parameters $\hat{\boldsymbol{\theta}}$ (weights and biases of each layer of the network) such that the network predictions $\{f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)\}$ are close to the desired values $\{\boldsymbol{y}_i^\star\}$, for all the training samples, according to the loss function $\mathscr{L}$:

$$\hat{\boldsymbol{\theta}} \in \arg\min_{\boldsymbol{\theta}} \frac{1}{N} \sum_{i=1}^{N} \mathscr{L}\left(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \boldsymbol{y}_i^\star\right), \tag{2.5}$$

- *inferring* the output $\widehat{\boldsymbol{y}} = f_{\boldsymbol{\theta}}(\boldsymbol{x})$ from some new input $\boldsymbol{x}$, by applying the MLP.

In practice, training rarely works out of the box, but rather multiple iterations are necessary to tune the training parameters and to balance each term of the loss function $\mathscr{L}(\boldsymbol{y}, \boldsymbol{y}^\star)$, when it contains several terms. Only then can a satisfactory performance of the neural network be reached.

Let us break down each of the different steps:

2.4.1 Collecting training samples

We discussed in Section 2.3 how MLPs can approximate any target mapping $f^\star$ provided that a sufficient number of hidden neurons are used. Yet in practice, we have no idea of the ideal function $f^\star$. We can only indicate the output $\boldsymbol{y}_i^\star$ we desire for some input $\boldsymbol{x}_i$. Obviously, the more examples of $(\boldsymbol{x}_i, \boldsymbol{y}_i^\star)$ pairs we provide, the more information we give about the desired function $f^\star$, see Fig.2.8. Several pitfalls must be avoided when building the collection of training examples:

- diversify as much as possible the training samples (in Fig.2.8 too many samples with similar outputs are given by the green crosses),
- sample as uniformly as possible the space (the green crosses do not cover uniformly the domain of $f^\star$),
- limit the impact of *outliers* (indicated by purple arrows in Fig.2.8), by filtering the training pairs in a preprocessing step, or making sure that the training process does not lead to overfitting.

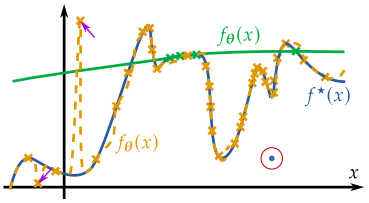

Figure 2.8: We cannot access the ideal function $f^\star(\boldsymbol{x})$, only examples marked by crosses. By fitting the few green crosses, we may obtain the model $f_{\boldsymbol{\theta}}(\boldsymbol{x})$ which is very good for values of $\boldsymbol{x}$ close to the pairs provided for learning, but extremely bad elsewhere (extrapolation regime). With many more examples (orange crosses), we can fit a better model. Although the model $f_{\boldsymbol{\theta}}(\boldsymbol{x})$ perfectly reproduces the outputs of the training samples (it goes through each of the orange crosses), it makes some errors at unseen points, in particular due to two outliers pointed by the purple arrows. Note that the ideal function $f^\star(\boldsymbol{x})$ is discontinuous at a point indicated by a red circle. Even with an infinite number of examples and a big enough network, the network may never model $f^\star$ at this point (a null set).

To accommodate for the difficulty of the task (i.e., the intricate evolution of $f^\star$ with $\boldsymbol{x}$, the large dimensions d_{in} and d_{out}), the size of the network and the number of training samples must be well-adjusted. A usual mantra in deep learning is: the more data, the better. As an example, the large language model Llama3.3 released by Meta in 2024 used text in different languages representing approximately 5×10^{12} words for training. According to Meta, the training then took about 40 million hours of computation on a single H100-80GB graphic cards (the training time was obviously reduced by using data-centers to parallelize computations over tens of thousands of such graphic cards), representing about 10.4 t eq. CO_2. Training deep neural networks on large datasets requires huge computational costs, which implies an environmental cost both in terms of natural resources to build the hardware to store and process the data and in terms of energy to power them. These impacts must be taken into account when designing products that rely on such technology. Several steps can be taken to mitigate these impacts, such as:

- *amortized inference*: refers to the large difference between the huge training cost of deep neural networks and the limited cost when applying an already trained network to some new data (i.e., inferring the output for some unseen data). Once a network is trained, its application is often less costly than alternative methods;
- *frugal design*: training and inference can be both sped up by considering networks with small or simple architectures, i.e. by integrating the computational cost among the constraints at design time;
- *model sharing*: sharing the network weights with others can save a lot of training time by avoiding the replication of similar trainings each time a user needs a network to solve a common task, these pretrained models can then either be applied directly or *finetuned* to a specific task; general purpose models are referred to as *foundational models*;
- *model simplification*: there are many approaches to reduce the computational complexity of a model, for example a larger model can be used to train a simpler model (a process called *model distillation*), weights can be quantized, and reduced precision used in all arithmetic operations;

- *edge computing* consists of using embedded devices to process locally the data, this can reduce the required bandwidth to communicate with other devices and servers as well as address privacy issues (local processing removes the need to exchange sensitive personal information).

Beyond the environmental cost, deep learning also has impact on the labor involved in the production of training data and the evaluation or correction of network predictions. In the framework of *supervised learning*, it is necessary to provide the desired output $\boldsymbol{y}_i^\star$ for each training sample $\boldsymbol{x}_i$. For an application such as automatically recognizing traffic signs, many human annotators spent days labelling images to train deep neural networks or improve the performance of existing ones. This invisible aspect of artificial intelligence has been documented by several investigations, see for example the 2024 article "What's behind the AI boom? Exploited humans" by James Muldoon, Mark Graham, and Callum Cant in the Los Angeles Times[††] or the Fairwork reports (Howson et al., 2022). Deep neural networks are widely used in the WEIRD[‡‡] population and rely heavily on tedious work of a labor often located in non-WEIRD countries.

For several tasks it is possible to avoid a costly manual labelling, for example by using simulated data (i.e., drawing ground-truth outputs $\boldsymbol{y}^\star \sim \mathrm{p}(\boldsymbol{y}^\star)$ and matching input data $\boldsymbol{x} \sim \mathrm{p}(\boldsymbol{x}|\boldsymbol{y}^\star)$), provided that a (good) simulator is available, generally based on domain-specific knowledge. Another approach consists of self- or unsupervised learning. The idea is to define tasks for which the correct completion can be evaluated using raw data alone. For example, if the task is to denoise an image acquired by a given instrument, we can capture pairs of images of the same objects with the instrument, restore the first image with a neural network and check if the restored image is close to the second noisy acquisition (i.e., minimize $\mathbb{E}_{y\sim\mathrm{p}(y),\, x_1\sim\mathrm{p}(x|y),\, x_2\sim\mathrm{p}(x|y)}[\|f_{\boldsymbol{\theta}}(\boldsymbol{x}_1) - \boldsymbol{x}_2\|_2^2]$). If the noise is centered (i.e., zero mean) and independent from one acquisition to the next, then the image that is closest, on average, to the second acquisition is a noiseless image of the objects, see Noise2Noise approaches and its many variants (Lehtinen et al., 2018) (video presentation: https://www.youtube.com/watch?v=dcV0OfxjrPQ).

Another example is masking: the task is to model the distribution of some missing data $\mathrm{p}(\boldsymbol{x}|\mathbf{M}\boldsymbol{x})$, where $\mathbf{M}$ is a diagonal matrix with $[\mathbf{M}]_{ii} = 1$ if the i-th entry of $\boldsymbol{x}$ is available and 0 if it is masked ($\boldsymbol{x}$ may correspond to some text, represented as a real-valued vector by a transform called tokenization, or an image). It suffices to collect large amounts of text or images, mask parts of each text or image sample, and evaluate how well the network prediction matches with the actual text or image under the mask. Masking is at the core of the training strategy of many large language models (LLMs), see for example Fig. 2.9 for text and Fig.2.10 for images, and several variants can be considered (e.g., next sentence prediction: is the proposed sentence the sequel of the first sentence or not? as shown in Fig.2.9, or jigsaw puzzles illustrated in Fig.2.11).

```
Input = [CLS] the man went to
[MASK] store [SEP] he bought
a gallon [MASK] milk [SEP]
Label = IsNext
Input = [CLS] the man [MASK]
to the store [SEP] penguin
[MASK] are flight ##less
birds [SEP]
Label = NotNext
```

Figure 2.9: Word masking and next sentence prediction used to train the large language model BERT (from Devlin et al., 2019).

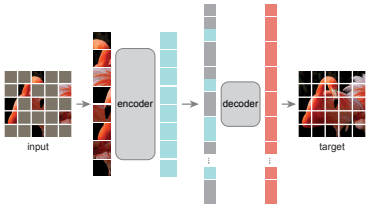

Figure 2.10: Learning to reconstruct an image with missing blocks helps to build representations of the image content (from He et al., 2022).

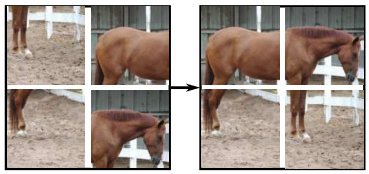

Figure 2.11: Solving automatically-generated jigsaw puzzles also requires understanding the image content, which is helpful for many tasks such as image classification (from Carlucci et al., 2019).

[††] https://www.latimes.com/opinion/story/2024-07-12/artificial-intelligence-workers-labor-feeding-the-machine

[‡‡] WEIRD stands for Western, Educated, Industrialized, Rich, and Democratic

2.4.2 Defining a loss function

The role of the loss function $\mathscr{L}(\boldsymbol{y}, \boldsymbol{y}^\star)$ is to quantify how close the network output $\boldsymbol{y}$ is to the desired output $\boldsymbol{y}^\star$. The loss $\mathscr{L}$ corresponds to a performance criterion with respect to which the network parameters are optimized. There are two main ways to define a loss function:

- derive $\mathscr{L}$ from statistical considerations,
- define an evaluation criterion for the task and use this criterion as a loss function.

We give below examples of some common losses, see also (Goodfellow et al., 2016; Bach, 2024).

For binary classification problems (i.e., when $y^\star \in \{0,1\}$) if the prediction y is also binary we could use the "0–1 loss" $\mathscr{L}(y, y^\star) = \mathbb{1}_{y \neq y^\star}$ (the loss is 0 if the classification is correct and 1 otherwise). However, to estimate the network parameters using an optimization technique based on the computation of the gradient, a binary output and binary loss is a bad choice since it introduces discontinuities and results in the non-differentiability of the loss with respect to the network parameters $\boldsymbol{\theta}$. It is preferable that the prediction $y = f_{\boldsymbol{\theta}}(\boldsymbol{x})$ corresponds to the probability $\mathrm{P}(y^\star = 1|\boldsymbol{x}) \in [0,1]$. This probability distribution can be compared, using a statistical distance, to the ground-truth probability $\mathrm{P}^\star(y^\star = 1|\boldsymbol{x})$. The Kullback-Leibler divergence $\mathscr{D}_{\mathrm{KL}}(\mathrm{p} \,\|\, \mathrm{q})$ between two probability distributions p and q is given by:

$$\mathscr{D}_{\mathrm{KL}}(\mathrm{p} \,\|\, \mathrm{q}) = \int_{\mathbb{R}} \mathrm{p}(y) \log \frac{\mathrm{p}(y)}{\mathrm{q}(y)} \mathrm{d}y \tag{2.6}$$

$$= \underbrace{\int_{\mathbb{R}} \mathrm{p}(y) \log \mathrm{p}(y) \mathrm{d}y}_{-\text{entropy of p}} \underbrace{- \int_{\mathbb{R}} \mathrm{p}(y) \log \mathrm{q}(y) \mathrm{d}y}_{\text{cross-entropy: } -\mathbb{E}_p[\log q]} \tag{2.7}$$

or, for discrete probability functions, by

$$\mathscr{D}_{\mathrm{KL}}(\mathrm{P} \,\|\, \mathrm{Q}) = \sum_x \mathrm{P}(y) \log \frac{\mathrm{P}(y)}{\mathrm{Q}(y)} \tag{2.8}$$

$$= \underbrace{\sum_y \mathrm{P}(y) \log \mathrm{P}(y)}_{-\text{entropy of P}} \underbrace{- \sum_y \mathrm{P}(y) \log \mathrm{Q}(y)}_{\text{cross-entropy}} . \tag{2.9}$$

$\mathscr{D}_{\mathrm{KL}}$ is always non-negative and equal to 0 if and only if $\mathrm{p}(y) = \mathrm{q}(y)$ almost everywhere. Let P denote the ground truth distribution $\mathrm{P}^\star(y^\star = 1|\boldsymbol{x})$ and $Q = f_{\boldsymbol{\theta}}$ our predicted distribution. Minimizing the Kullback-Leibler divergence $\mathscr{D}_{\mathrm{KL}}$ amounts to minimizing the cross-entropy (the other term being constant). The cross-entropy term corresponds to a sum over the two possible outcomes: $y|\boldsymbol{x}$ equal to 0 and $y|\boldsymbol{x}$ equal to 1, which gives:

$$-\mathrm{P}^\star(y^\star = 1|\boldsymbol{x}) \log \left[f_{\boldsymbol{\theta}}(\boldsymbol{x})\right] - \mathrm{P}^\star(y^\star = 0|\boldsymbol{x}) \log \left[1 - f_{\boldsymbol{\theta}}(\boldsymbol{x})\right] . \tag{2.10}$$

In practice, we don't have access to the ground truth probability $\mathrm{P}^\star$, only to training samples $(\boldsymbol{x}_i, y_i^\star)_{i=1..N}$. For Bernoulli distribution, the

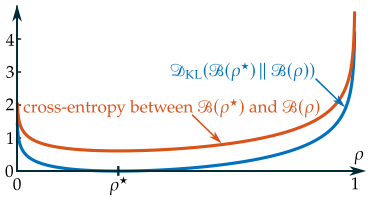

Figure 2.12: Evolution of the Kullback-Leibler divergence $\mathscr{D}_{\mathrm{KL}}$ and cross-entropy between a ground truth Bernoulli distribution $\mathscr{B}(\rho^\star)$ of parameter $\rho^\star$ and another Bernoulli distribution $\mathscr{B}(\rho)$. The two criteria reach their minimum when $\rho = \rho^\star$. The divergence $\mathscr{D}_{\mathrm{KL}}$ is equal to 0 when the two distributions are identical.

probability of success is also equal to the mean. For each training pair, we can replace $\mathrm{P}^\star(y^\star = 1|\boldsymbol{x}) \log\left[f_{\boldsymbol{\theta}}(\boldsymbol{x})\right]$ by $\mathbb{E}_{y^\star|\boldsymbol{x}}\left[\log\left[f_{\boldsymbol{\theta}}(\boldsymbol{x})\right]\right]$ and $\mathrm{P}^\star(y^\star = 0|\boldsymbol{x}) \log\left[1 - f_{\boldsymbol{\theta}}(\boldsymbol{x})\right]$ by $\mathbb{E}_{1-y^\star|\boldsymbol{x}}\left[\log\left[1 - f_{\boldsymbol{\theta}}(\boldsymbol{x})\right]\right]$, and approximate the expectations with empirical averages:

$$\hat{\boldsymbol{\theta}} \in \arg\min_{\boldsymbol{\theta}} \underbrace{\frac{1}{N}\sum_{i=1}^{N} -y_i^\star \log\left[f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)\right] - (1 - y_i^\star)\log\left[1 - f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)\right]}_{\text{binary cross-entropy loss}}. \quad (2.11)$$

binary cross-entropy is the standard loss for binary classification tasks

This is called *binary cross-entropy* minimization and is the standard way to train deep neural networks to solve binary classification tasks[§§].

For multi-class classification problems (i.e., $y^\star \in \{0, 1, C-1\}$) the cross-entropy minimization approach can be extended. As discussed in the introduction of the chapter, learning is made easier if we use one-hot encoding for the class, i.e., if the network outputs a vector $\boldsymbol{y} \in [0,1]^C$ such that, for $1 \leq i \leq C$, y_i represents the probability that the input vector $\boldsymbol{x}$ corresponds to the i-th class (to represent probabilities, $\boldsymbol{y}$ has to verify the constraints $\forall i, 0 \leq y_i \leq 1$ and $\sum_i y_i = 1$). The *cross-entropy* defined in equation (2.9) becomes: $-\log y_{y^\star}$, i.e., the opposite of the log probability predicted for the ground truth class (0 if that class was correctly predicted with total confidence, $+\infty$ if the prediction was that this class was impossible for the given input).

Note that a special activation function is used on the output layer when predicting class probabilities: the soft-max activation described on page 14.

For regression problems (i.e., $y^\star \in \mathbb{R}$) a natural loss is the mean squared error (MSE) (corresponding to the L^2 loss, up to a constant multiplicative factor). It evaluates the square of the difference between the model prediction and the ground truth, leading to the classical least-squares minimization problem:

$$\hat{\boldsymbol{\theta}} \in \arg\min_{\boldsymbol{\theta}} \frac{1}{N}\sum_{i=1}^{N}\left[f_{\boldsymbol{\theta}}(\boldsymbol{x}_i) - y_i^\star\right]^2. \quad (2.12)$$

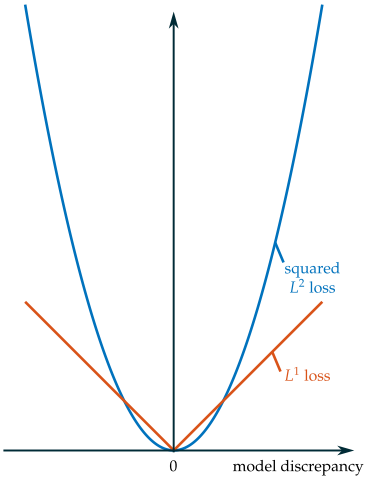

Figure 2.13: The L^2 and L^1 loss functions: the L^1 loss penalizes less strongly large model discrepancies.

To give less weight to samples for which the model has a poor fit (potential outliers), the mean absolute error is sometimes preferred, see Fig. 2.13:

$$\hat{\boldsymbol{\theta}} \in \arg\min_{\boldsymbol{\theta}} \frac{1}{N}\sum_{i=1}^{N}\left|f_{\boldsymbol{\theta}}(\boldsymbol{x}_i) - y_i^\star\right|. \quad (2.13)$$

Multi-task learning consists of training a network to solve several tasks at once. Using a single network rather than separate networks for each task can promote interactions between tasks. For example, if an image has to be analyzed to segment each object and estimate the depth (i.e., the distance from the camera) of each pixel, a synergy between the tasks can

[§§] Note that $\log[f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)]$ and $\log[1 - f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)]$ can be arbitrarily low. To prevent numerical issues, the popular Python library for deep-learning `Pytorch` clamps the results of the log to -100 in the function `BCELoss`.

help the network to learn meaningful features. Multi-task learning is a vast topic, see for example (Vandenhende et al., 2021). Many architectures and training strategies have been proposed in the literature. The tasks are usually combined by forming a linear combination of losses for each task: $\alpha_1 \cdot \mathscr{L}_1 + \alpha_2 \cdot \mathscr{L}_2 + \dots$

Regularization consists of including additional terms to the loss function to improve the generalization of the network, i.e., the performance reached, after training, when the network is applied to new data. These supplementary terms are introduced to promote mappings $f_{\boldsymbol{\theta}}$ with some form of *regularity*. Regularization is a very common approach in statistics when a model with many parameters is fitted to a set of observations. By requiring the model to not only fit well the data but also be smooth (in some sense), we prevent the model to produce arbitrarily bad predictions for unseen data.

The most standard regularization is the (squared) L^2 norm, (also called weight decay, ridge regression, or Tikhonov regularization): $\frac{1}{2}\sum_{i=1}^{H} \|\boldsymbol{W}_i\|_2^2$. It favors weights that are close to zero. The L^1 norm is preferred in some cases because it leads to solutions that are *sparse*, i.e., with many zeros. It provides a way to build an over-parameterized model and discover, during training, which meaningful parameters should be kept (i.e., with a non-zero value).

2.4.3 Activation functions

Activation functions used for the hidden layers and for the output layer serve different purposes, their choice obeys different considerations. We discuss first activation functions g used for hidden layers.

Historically, the sigmoid σ was the function of choice for activation functions, see figure 2.14:

$$\sigma(x) = \frac{1}{1+\exp(-x)}. \tag{2.14}$$

This activation function tends to 1 when $x \longrightarrow \infty$ and to 0 when $x \longrightarrow -\infty$. This saturation behavior corresponds to a smooth on/off. Its interpretation is quite easy, yet training networks with such activation functions is difficult because in the saturation areas ($|x|$ large), the function is almost flat and its derivative is very small. This causes gradient-based training algorithms to converge very slowly or get stuck due to numerical accuracy issues.

Note that the hyperbolic tangent $\tanh(x) = [\exp(x) - \exp(-x)]/[\exp(x) + \exp(-x)]$, shown in figure 2.15, is closely related to the sigmoid: $\tanh(x) = 2\sigma(2x) - 1$; it has the same shape as the sigmoid, up to a scaling and an offset (which leads to a range symmetrical with respect to 0: $[-1, 1]$). Within hidden layers, the sigmoid and the hyperbolic tangent can be used interchangeably.

An activation function that has been widely used to train (very) deep neural networks is the Rectified Linear Unit (ReLU). This activation corresponds to the function: $g(x) = \max(0, x)$ shown in figure 2.16. It is a piecewise linear function, which means that, when combined with affine

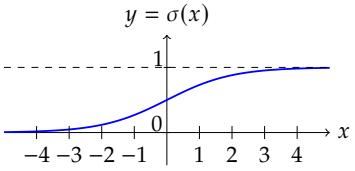

Figure 2.14: The sigmoid activation function.

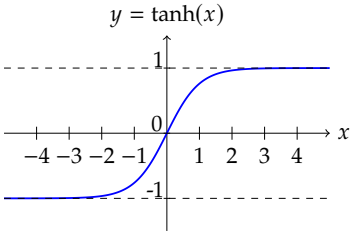

Figure 2.15: The hyperbolic tangent activation function.

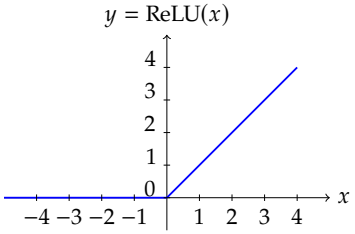

Figure 2.16: The ReLU activation function.

transforms and other piecewise linear functions in an MLP, we get an output that is also a piecewise linear function. Learning the parameters of the network specifies where the breakpoints of the output piecewise linear function are located and the slope of each linear piece. The ReLU is often the default recommendation (Goodfellow et al., 2016), although it is not perfect: the flat area, when $x < 0$, gives no indication of how the parameters should be modified, and it is non-differentiable at $x = 0$. Many variants have been suggested in the literature: the *leaky-ReLU* (figure 2.17) is also piecewise linear but avoids a flat area by using a small slope for $x < 0$: $g(x) = \max(0, x) + \alpha \cdot \min(0, x)$ with $\alpha \ll 1$, for example $\alpha = 0.01$ (default choice in `Pytorch`). The non-differentiability issue can be solved by using one of many smooth approximations of the ReLU function such as the *softplus* function[¶¶]: $g(x) = \frac{1}{\beta} \log[1 + \exp(\beta \cdot x)]$, shown in figure 2.18.

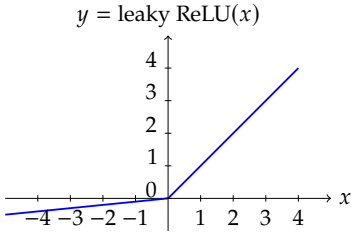

Figure 2.17: The leaky ReLU activation function (shown here with $\alpha = 0.1$).

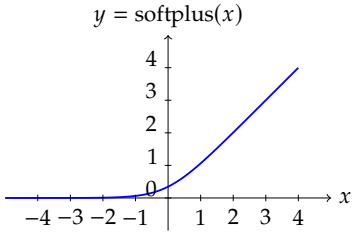

Figure 2.18: The softplus activation function (shown here with $\beta = 2$) provides a smooth approximation of the ReLU.

Another possible choice for activation function is the sine function. Like the hyperbolic tangent, it gives an output in the bounded range $[-1, 1]$ while avoiding flat areas (i.e., saturation behavior). In contrast to the ReLU, it is very smooth (of differentiability class C^∞), which can be very useful for applications requiring the evaluation of (partial) derivatives of the network outputs with respect to its inputs, i.e., when representing a function $f_{\boldsymbol{\theta}}(\boldsymbol{x})$ with a network and checking whether this function fulfills some partial differential equation. While the oscillating behavior of the sine function may seem a major drawback for an activation function, it can be beneficial to model functions that vary quickly when the input is slightly modified, i.e., high-frequency functions (for example, to build a continuous approximation of an image, the network output must vary quickly when the network input, indicating the spatial coordinates of pixels, changes from the location of a dark area of the image to a close-by brighter area), see (Sitzmann et al., 2020).

There are many available options for activation functions, see the recent review (Dubey et al., 2022). Several researchers also proposed to make the shape of the activation function learnable, see (Apicella et al., 2021).

2.4.4 Activation functions of the output layer

Depending on the task and the application domain, the network outputs may represent physical quantities that are restricted to some predefined ranges: non-negative values (e.g, a length) or bounded values (e.g., time in the range $[t_{\text{begin}}, t_{\text{end}}]$). The ReLU and its variants such as softplus can be applied to ensure a non-negative output. The sigmoid can be used to constrain the values to remain within a range $[v_{\min}, v_{\max}]$: $g(x) = (v_{\max} - v_{\min}) \cdot \sigma(x) + v_{\min}$.

If the output corresponds to the probability of a binary event such as a detection problem (positive detection or absence of detection), the sigmoid function is a perfect fit: it transforms the output such that it lies in the range $]0, 1[$.

In many applications such as classification, the network output vector is interpreted as the probability for each class, i.e., $\forall i,\ 0 \leq y_i \leq 1$ and $\sum_i y_i = 1$. These two properties can be enforced with the soft-max

[¶¶] for numerical stability, when $\beta \cdot x$ is large $g(x)$ is set to x ($\beta \cdot x > 20$ by default in `Pytorch`)

activation function. In contrast to the activation functions discussed in the previous paragraph, for hidden layers, the soft-max is not separable (i.e., it is applied jointly on several output values). It is defined by:

$$\forall i,\ [\text{softmax}(\boldsymbol{x})]_i = \frac{\exp([\boldsymbol{x}]_i)}{\sum_j \exp([\boldsymbol{x}]_j)}. \tag{2.15}$$

2.4.5 Evaluation of deep neural network performances

To train a neural network, it is necessary to both select training data (collection of data on which to apply the network) and define a loss function (that the network should minimize). The loss function should reflect how the performance of the network on the task of interest will be evaluated ultimately. Ideally, the same criterion should be used both for training and evaluation (so as to maximize the evaluated performance), but this is not always feasible (the evaluation criterion may be non-differentiable, or too costly) and surrogate functions are used instead during training. The training samples should be representative of the data on which the network will be applied post-training, otherwise, a drop of performance should be expected (e.g., a network trained to recognize objects in images of indoor scenes may fail when given outdoor images of the same objects). Even worse, if given images of very different nature (e.g., underwater images), the network might still confidently recognize objects (such as sofas or bedside lamps in our example, even though these objects make no sense in this new context), without ever detecting that the input is completely different. This problem is called *out-of-distribution* detection, it is an active research topic in machine learning.

To assess how well a network will perform in practice, on future data, it is common practice to split the dataset into several parts, see figure 2.19. Only part of the dataset is actually used to learn the network parameters by minimizing the loss function (the *training set*), the rest of the data are used either to select some hyper-parameters (e.g., weight different terms of the loss function, find a good scheduler for the learning rate, set the number of training iterations, see sections 3.2.2 and 3.2.3) with the so-called *validation set*, only used from time to time to monitor progress, or evaluate the performance after training is complete (the *testing set*).

If the training has been successful, the network should at least perform well on samples from the training set. Beyond these particular data on which the network has been trained, we are usually interested on how well the network behaves on *never seen* data, i.e., the *generalization* capability of the model. High performance on the training set does not necessarily translate into good performance on new samples due to the risk of *overfitting*, see figure 2.20.

Combating overfitting by early stopping: It often happens that (i) part of the training data is noisy (e.g., some labels may be incorrect), and (ii) the neural network possesses so many parameters that it could fit perfectly all the training set. Learning to reproduce erroneous labels does not make sense. It also leads to a degradation of the performance on unseen data (to predict incorrect labels, the model has to "bend" the decision boundary to accommodate for these outliers, impacting negatively the performance

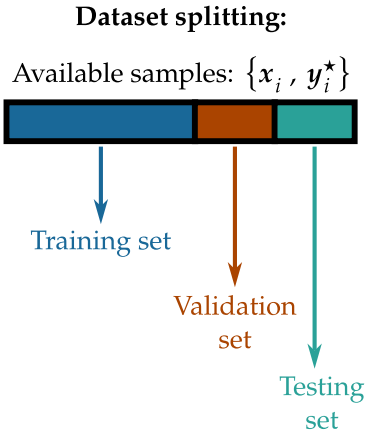

Figure 2.19: The dataset (pairs of inputs $\boldsymbol{x}$ and expected outputs $\boldsymbol{y}^\star$) is usually split into 3 parts: (i) the largest part of samples are used to train the model by minimizing the average loss (the *training set*); (ii) samples are periodically used to monitor how the network performs on unseen data and decide when to stop training (the *validation set*); (iii) at the end of training, evaluation of the network is carried out on data never accessed during the training (*testing set*).

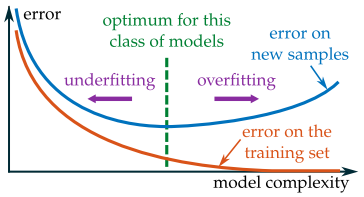

Figure 2.20: The underfitting and overfitting problems: a model with a limited complexity cannot fit well the problem and results in *underfitting*: a strong bias due to the use of an over-simplified model; when the model complexity is too high, it may *overfit* the training set: it becomes too sensitive to small fluctuations in the training set and the performance on new sample worsens: it suffers from large variance. Using more training data shifts the optimal model complexity towards more complex models.

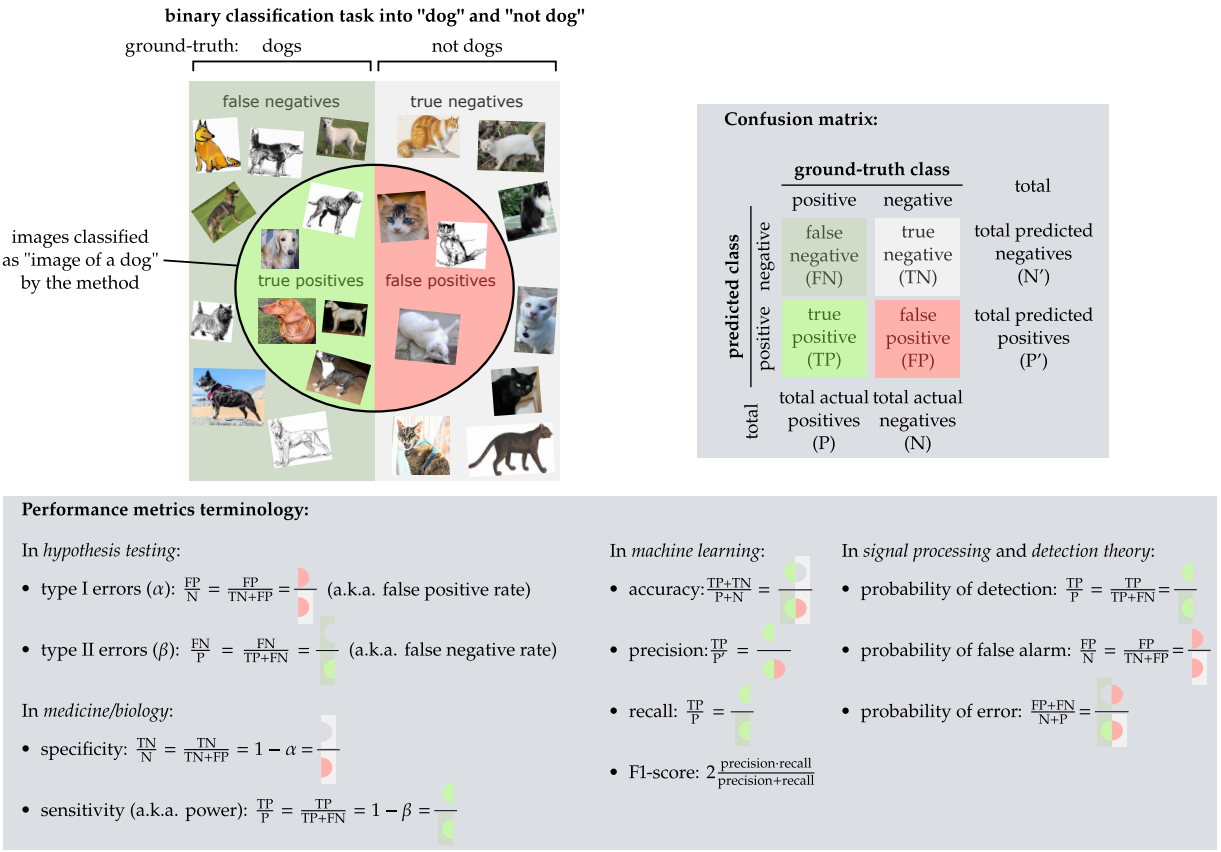

Figure 2.21: Several performance metrics can be derived from the confusion matrix. The terminology depends on the domain. Adapted from the illustrations on Wikipedia by Walber and Epachamo.

on many other input data). A common approach to avoid this *overfitting* phenomenon consists of stopping the training before convergence is reached on the training set (this is called *early stopping*, since continuing the training could have further improved the loss function on the training set). Deciding when to stop can be based on progress on the *validation loss*, i.e., the performance on some samples held out for evaluation. When the progress on the validation loss is not sufficient, the training is stopped (this is sometimes called a patience criterion (Zhang et al., 2023)).

Classical evaluation criteria: the first element of evaluation is how well the network outputs match the ground truth. Evaluation criteria depend on the task. For binary classification problems, i.e., when $y^\star \in \{0, 1\}$, the first step is to compute the confusion matrix: count the number positive samples wrongly assigned to the negative class (false negative), the number of negative samples wrongly assigned to the positive class (false positive), the number of correctly classified negative samples (true negative) and correctly classified positive samples (true positive), see the top half of figure 2.21. Several performance metrics can be computed from the confusion matrix, with a domain-specific terminology. In machine learning, the four most used criteria are:

accuracy, indicating the proportion of correctly classified samples; this criterion alone is not sufficient when classes are imbalanced (e.g., for a rare pathology occurring only once every 10,000 samples, never detecting the pathology leads to a negligible decrease of the accuracy); note that 1 − accuracy corresponds to the (empirical) *probability of error*,

precision, that corresponds to the fraction of detections (i.e., outputs equal to 1) that are correct; note that 1-precision gives the proportion of detections that are erroneous;

recall, showing the fraction of positive samples correctly classified; note that this corresponds to the probability of detection, also called sensitivity or power, in other domains;

F1-score, combines in a single criterion the precision and recall using an harmonic mean, i.e., the reciprocal of the mean of 1/precision and 1/recall, see figure 2.22.

For classification problems with $C > 2$ classes, the confusion matrix becomes a $C \times C$ matrix. The extension of the definition of the *accuracy* is straightforward: the proportion of correctly classified elements among all samples (i.e., trace of the confusion matrix divided by the sum of all entries). Since neural networks do not generally output a single class but rather the probability for each class, given the input data, it may be reasonable to reward a network that assigns a large probability to the correct class, even if a wrong class has a slightly higher probability. Top-k accuracy counts the proportion of samples for which the true class is among the k classes predicted with the largest probabilities.

Metrics used in binary classification can still be computed using a one-vs-rest strategy. Different values are then obtained for each class. These values can be averaged, possibly with weights depending on the importance of each class, either to aggregate false negative, true negative, true positive, and false positive values (micro-averaging), or directly combine the metrics such as precision or recall (macro-averaging).

For regression problems, i.e., when the output of the network is real-valued, the mean square error (MSE) or root mean square error (RMSE) are often reported***:

$$\text{(MSE)} \qquad \frac{1}{\text{Card(testing set)}} \sum_{i \in \text{testing set}} \| f_{\boldsymbol{\theta}}(\boldsymbol{x}_i) - \boldsymbol{y}_i^\star \|_2^2 \,, \tag{2.16}$$

where $\{\boldsymbol{x}_i, \boldsymbol{y}_i^\star\}_{i \in \text{testing set}}$ denotes the pairs of input samples and associated ground truth values from the testing set. The RMSE is simply the square root of the MSE, it has the same units as $\boldsymbol{y}$. Replacing the squared L^2 norm by an L^1 norm leads to the mean absolute error (MAE):

$$\text{(MAE)} \qquad \frac{1}{\text{Card(testing set)}} \sum_{i \in \text{testing set}} \| f_{\boldsymbol{\theta}}(\boldsymbol{x}_i) - \boldsymbol{y}_i^\star \|_1 \,. \tag{2.17}$$

Computational and memory efficiencies: Beyond the raw performance of the network, it is essential when comparing models to also consider the computational and memory cost of each network. These directly affect to cost (both economical and environmental) of deploying such networks. The memory footprint can be evaluated by counting the number of parameters of the network and multiplying it with the size of a single parameter (which depends on data type and precision). The

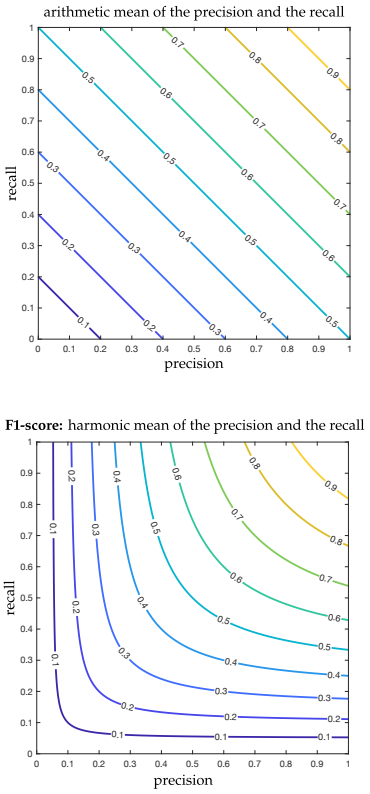

Figure 2.22: When we combine precision and recall into a single criterion with the harmonic mean (i.e., the F1-score), we give more weight to the least favorable criterion, compared to the arithmetic mean (precision+recall)/2 represented on top.

*** estimating both the MSE and the MAE can be useful since these two metrics have different sensitivity to large errors

computational cost can be assessed using specialized libraries such as `ptflops`[†††] or `fvcore`[‡‡‡].

Closely related to the computational and memory costs is the CO_2 impact of training a deep neural network and applying it to new data (inference). There are several approaches to evaluate this impact, from direct measurement of the power consumption of the computer used to train/apply the network to a more global evaluation based on runtime, type of processor / graphical processing unit (PGU), amount of memory used (see https://calculator.green-algorithms.org/), including dedicated programming libraries to get a more fine-grained estimation (see for example the `Python` library https://codecarbon.io/). The CO_2 impact also depends on the carbon intensity of the local electricity production, which varies largely with the geographical location and fluctuates over time, see https://app.electricitymaps.com/map/72h/hourly.

[†††] https://pypi.org/project/ptflops/
[‡‡‡] https://github.com/facebookresearch/fvcore/blob/main/docs/flop_count.md

Finding neural network weights: automatic differentiation and stochastic minimization

3

In the previous chapter, we have left out important technical details on how to find neural network weights that minimize our loss function. This is achieved in practice by minimization techniques that use only a random subset of the training data at each step (hence the term *stochastic minimization*), and that rely on computing the gradient of the loss with respect to the network weights. The opposite of the gradient indicates the *steepest descent*, i.e., the direction of variation that would lead to the largest decrease of the loss if the network weights were to be only slightly modified. In deep learning, computation of the gradient relies on an efficient technique from automatic differentiation known as *back-propagation*. To understand why back-propagation is used in this context, we will present quickly the domain of *automatic differentiation* in Section 3.1. Then, we will introduce the topic of stochastic minimization in section 3.2.

3.1 Automatic differentiation

This section is mostly based on (Baydin et al., 2018), the interested reader is invited to read this survey paper for a more detailed presentation.

3.1.1 Evaluating a derivative on a computer

In computer science and applied mathematics, several techniques are available to compute derivatives:

- *numerical differentiation*, i.e., finite differences,
- *manual differentiation*, i.e., computing the derivative by hand and evaluating on a computer the expression obtained at any requested point,
- *symbolic differentiation*, i.e., obtaining the closed-form expression of the derivative using a *computer algebra system* (a symbolic math software),
- *automatic differentiation*, i.e., computing the numerical value of the derivative at a given point, using derivation rules.

Numerical differentiation approximates the definition of the derivative of a multivariate function $\mathcal{L} : \mathbb{R}^n \to \mathbb{R}$:

$$\frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial x_i} \stackrel{\text{def}}{=} \lim_{h \to 0} \frac{\mathcal{L}(\boldsymbol{x} + h\boldsymbol{e}_i) - \mathcal{L}(\boldsymbol{x})}{h} \approx \frac{\mathcal{L}(\boldsymbol{x} + \epsilon \boldsymbol{e}_i) - \mathcal{L}(\boldsymbol{x})}{\epsilon} \text{ for } \epsilon \text{ small,} \tag{3.1}$$

where $\boldsymbol{e}_i$ is the indicator of the i-th component (i.e., $[\boldsymbol{e}_i]_j = 0$ for all $j \neq i$ and $[\boldsymbol{e}_i]_i = 1$). It is very easy to implement* but costly to compute $\boldsymbol{\nabla}_x \mathcal{L}(\boldsymbol{x})$ when n is large since it requires $n+1$ evaluations of $\mathcal{L}$. Choosing

* the approximation may be slightly improved by using central differences: $[\mathcal{L}(\boldsymbol{x} + \epsilon \boldsymbol{e}_i) - \mathcal{L}(\boldsymbol{x} - \epsilon \boldsymbol{e}_i)]/(2\epsilon)$, at a larger cost of $2n$ evaluations of $\mathcal{L}$ to obtain the gradient vector in full.

a relevant value for ϵ can also be tricky because very small values of ϵ lead to large round-off errors while large values of ϵ make the approximation less acceptable. The number of parameters in deep neural networks (often in the order of millions or billions) make this approach completely inapplicable due to its prohibitive computational cost.

Manual differentiation is applicable to simple models as encountered in many inverse problems in imaging. In these cases, the function $\mathscr{L}$ involves only a few linear transforms of $\boldsymbol{x}$ (e.g. blur) or non-linear operations (e.g. a squared L^2 norm). It is then tractable to obtain by hand the closed-form expression[†] of the gradient $\nabla_{\boldsymbol{x}}\mathscr{L}(\boldsymbol{x})$. It then suffices to implement a function that evaluates this expression at a given point $\boldsymbol{x}$. In deep learning, the numerous operations involved by the stacking of several layers and the need to explore various architectures and/or loss functions by trial and error quickly make this approach too time-consuming and cumbersome.

Symbolic differentiation relies on computer algebra systems to manipulate symbolic expressions and apply derivation rules. This offloads the effort of manual differentiation to a dedicated software or library. Factorization and simplification rules can potentially produce a simpler close-form expression than the direct application of derivation rules (e.g., the expression $\mathscr{L}(x) = x(x-1)^2 + (1+x)(2x-3) - x^3$ leads to $\frac{\mathrm{d}}{\mathrm{d}x}\mathscr{L}(x) = (x-1)^2 + 2x(x-1) + 2x - 3 + 2(1+x) - 3x^2$, which simplifies to 0; once the simplification has been identified, evaluation of the derivative is much faster). Yet, special care must be paid to how common sub-expressions are handled to prevent unnecessary repetitions of some computations (expanding the composition of many functions can lead to many terms).

Automatic differentiation, also known as *algorithmic differentiation*, applies the same derivation rules as in symbolic differentiation, but numerical values are handled to obtain the derivative at some point, rather than symbol expressions.

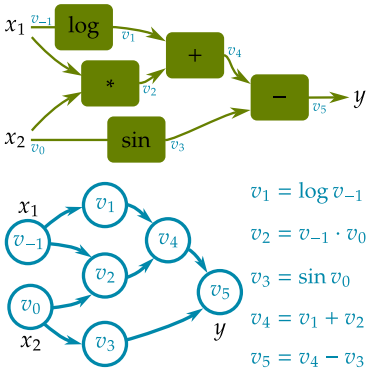

Figure 3.1: Representation of the function $y = \mathscr{L}(x_1, x_2) = \log(x_1) + x_1 x_2 - \sin(x_2)$. Operations are shown in green boxes. Intermediate values are indicated in blue. The computational graph drawn below shows how these values are combined to obtain the final result y. A node in the computational graph corresponds to a value (or a tensor) that is a function of the values at the origin of the incoming edges.

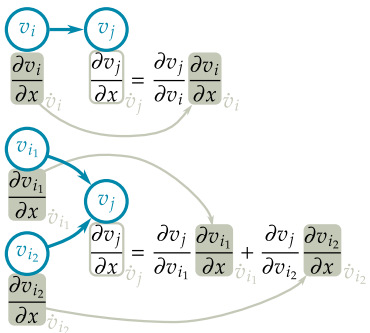

Figure 3.2: The application of the chain rule at a node v_j requires the value of the partial derivatives at the parent nodes.

3.1.2 The forward mode of automatic differentiation

The evaluation of a function $\mathscr{L}$ follows a sequence of steps (i.e., a computer program) that can be represented by an acyclic graph called the *computational graph* (or the *evaluation trace*), connecting intermediate values (nodes of the graph), usually through unary or binary operations, see figure 3.1. To compute the partial derivative of a value v_j with respect to an input variable x, the chain rule applies:

$$\frac{\partial v_j}{\partial x} = \sum_{v_i \in \mathscr{P}(v_j)} \frac{\partial v_j}{\partial v_i}\frac{\partial v_i}{\partial x}, \tag{3.2}$$

where $\mathscr{P}(v_j)$ is the set of parent nodes in the computational graph, i.e., nodes v_i such that there exists an edge directed from v_i to v_j in the computational graph, indicating a direct dependency of v_j on v_i.

Figure 3.2 illustrates the chain rule in the case of a unary function (e.g., $v_j = \sin v_i$) and in the case of a binary function (e.g., $v_j = v_{i_1} \cdot v_{i_2}$). In the forward accumulation mode (aka tangent linear mode), each

[†] two useful resources to derive the expression of the gradient are the Matrix Cookbook (Petersen, Pedersen, et al., 2008) and the website https://www.matrixcalculus.org/

Evaluation of $\mathcal{L}(x_1, x_2) = \log(x_1) + x_1 x_2 - \sin(x_2)$		
v_{-1}	$= x_1$	$= 2$
v_0	$= x_2$	$= 5$
v_1	$= \log v_{-1}$	$= \log 2$
v_2	$= v_{-1} \cdot v_0$	$= 2 \cdot 5$
v_3	$= \sin v_0$	$= \sin 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$
$\mathcal{L}(x_1, x_2)$	$= v_5$	$= 11.652$

Evaluation of $\partial\mathcal{L}/\partial x_1$		
$\dot{v}_{-1}$	$= \dot{x}_1$	$= 1$
$\dot{v}_0$	$= \dot{x}_2$	$= 0$
$\dot{v}_1$	$= \dot{v}_{-1}/v_{-1}$	$= 1/2$
$\dot{v}_2$	$= \dot{v}_{-1} \cdot v_0 + v_{-1} \cdot \dot{v}_0$	$= 1 \cdot 5 + 0 \cdot 2$
$\dot{v}_3$	$= \dot{v}_0 \cdot \cos v_0$	$= 0 \cdot \cos 5$
$\dot{v}_4$	$= \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
$\dot{v}_5$	$= \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
$\frac{\partial\mathcal{L}}{\partial x_1}(x_1, x_2)$	$= \dot{v}_5$	$= 5.5$

Figure 3.3: Evaluation of the function $\mathcal{L}$ defined in Fig.3.1 and its partial derivative with respect to the input parameter x_1, using the forward automatic differentiation mode. The evaluations can be run in parallel.

intermediate value v_i also has an associated value $\dot{v}_i$, called the *tangent*, that represents the partial derivative $\partial v_i/\partial x$. Note how the computation of a tangent value is obtained from the tangent values of the parent nodes, with information flowing in the same direction as the computational graph (shown with arrows "$\rightarrow$" in Fig. 3.2). To each elementary function $v_j(v_{i_1}, \ldots, v_{i_k})$ of the computational graph, we associate a derivative (or *forward tangent*) $\dot{v}_j(v_{i_1}, \ldots, v_{i_k}, \dot{v}_{i_1}, \ldots, \dot{v}_{i_k})$ in order to compute in parallel the derivative, see figure 3.3. In the figure, each line on the left table corresponds to an elementary operation of the computational graph. On the right table, the associated derivative is computed by application of the chain rule at that node. A run of the primal and tangent traces (i.e., the sequence of operations represented in the left and right tables) gives the value of $\mathcal{L}$ and its partial derivative with respect to a given input variable. To obtain the partial derivative with respect to a different input variable, the initialization must be modified (only the variable of interest has a tangent different from 0) and the whole tangent trace has to be reevaluated. Note that the method generalizes to functions $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ with several outputs: running the forward trace on a vector $\boldsymbol{a}$ and the tangent trace on a vector $\dot{\boldsymbol{x}} = \boldsymbol{e}_i$ (the i-th unit vector) gives the i-th column of the Jacobian matrix

tangent: $\dot{v}_i = \partial v_i/\partial x$

$$\mathbf{J}_f \overset{\text{def}}{=} \left.\begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}\right|_{\boldsymbol{x}=\boldsymbol{a}}$$

where y_j is the j-th component of the output $\boldsymbol{y} = f(\boldsymbol{x})$. Note that the Jacobian-vector product $\mathbf{J}_f \boldsymbol{r}$ can be computed in a single pass by initializing the tangent trace with $\dot{\boldsymbol{x}} = \boldsymbol{r}$. *Forward mode* automatic differentiation is efficient for functions f with $n \ll m$ since n passes are required to obtain the derivatives with respect to each of the n input variables. In the context of deep learning where there are many input variables (the network weights) and a single output variable (the value of the loss function), the *forward mode* is far too costly, a different approach is followed: the *reverse accumulation mode*.

3.1.3 The reverse accumulation mode of automatic differentation: back-propagation

The reverse mode propagates the partial derivative information backwards, by storing together with the intermediate value v_i an additional

Evaluation of $\mathcal{L}(x_1, x_2) = \log(x_1) + x_1 x_2 - \sin(x_2)$		
v_{-1}	$= x_1$	$= 2$
v_0	$= x_2$	$= 5$
v_1	$= \log v_{-1}$	$= \log 2$
v_2	$= v_{-1} \cdot v_0$	$= 2 \cdot 5$
v_3	$= \sin v_0$	$= \sin 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$
$\mathcal{L}(x_1, x_2)$	$= v_5$	$= 11.652$

Evaluation of $\nabla_{\boldsymbol{x}} \mathcal{L}$		
$\frac{\partial \mathcal{L}}{\partial x_1}$	$= \bar{x}_1 = \bar{v}_{-1}$	$= 5.5$
$\frac{\partial \mathcal{L}}{\partial x_2}$	$= \bar{x}_2 = \bar{v}_0$	$= 1.716$
$\bar{v}_{-1}$	$= \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	$= \bar{v}_{-1} + \bar{v}_1 / v_{-1} = 5 + 1/2 = 5.5$
$\bar{v}_0$	$= \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$	$= \bar{v}_0 + \bar{v}_2 \cdot v_{-1} = -0.284 + 2 = 1.716$
$\bar{v}_{-1}$	$= \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	$= \bar{v}_2 \cdot v_0 = 5$
$\bar{v}_0$	$= \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	$= \bar{v}_3 \cdot \cos 5 = -0.284$
$\bar{v}_2$	$= \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= \bar{v}_4 \cdot 1 = 1$
$\bar{v}_1$	$= \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$= \bar{v}_4 \cdot 1 = 1$
$\bar{v}_3$	$= \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$= \bar{v}_5 \cdot (-1) = -1$
$\bar{v}_4$	$= \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$= \bar{v}_5 \cdot 1 = 1$
$\bar{v}_5$	$= \bar{y} = \frac{\partial y}{\partial y}$	$= 1$

Figure 3.4: Evaluation of the same function $\mathcal{L}$ as in Fig.3.3 and its gradient, using the reverse accumulation mode (i.e., back-propagation). The evaluation of $\mathcal{L}$ is performed first, then the evaluation of $\nabla_{\boldsymbol{x}} \mathcal{L}$ is conducted by following the computational graph in the reverse direction.

variable $\bar{v}_i$, the adjoint $\bar{v}_i = \partial y_j / \partial v_i$, which indicates how the output y_j would be impacted by an infinitesimal change of the value v_i. Note that in deep learning, $y = \mathcal{L}(\boldsymbol{x})$ is a scalar corresponding to the value of the loss function (i.e., $y_j = y_1 = y$). The determination of the adjoint $\bar{v}_k$ at node v_k of the computation graph follows from the application of the chain rule:

$$\frac{\partial y_j}{\partial v_k} = \sum_{v_i \in \mathscr{C}(v_k)} \frac{\partial y_j}{\partial v_i} \frac{\partial v_i}{\partial v_k}, \tag{3.3}$$

adjoint: $\bar{v}_i = \partial y_j / \partial v_i$

where $\mathscr{C}(v_k)$ is the set of child nodes in the computational graph, i.e., nodes v_i such that there exists an edge directed from v_k to v_i in the computational graph, indicating a direct dependency of v_i on v_k. Figure 3.5 illustrates how the adjoint values can be computed by following the computational graph backwards from node y_j and accumulating terms originating from each of the child nodes. Compared to the forward mode of automatic differentiation, the adjoint value information flows backward in the computation graph (hence the name, *back-propagation*). This is shown in figure 3.4 where the gradient of the loss function $\mathcal{L}(x_1, x_2) = \log(x_1) + x_1 x_2 - \sin(x_2)$ is obtained on a single backward pass (right part of the figure). This contrasts with the forward mode which requires n passes to compute the gradient of a function $f : \mathbb{R}^n \to \mathbb{R}$. For a general function $f : \mathbb{R}^n \to \mathbb{R}^m$, m passes are required since the Jacobian matrix is obtained row by row (rather than column by column as with the forward mode).

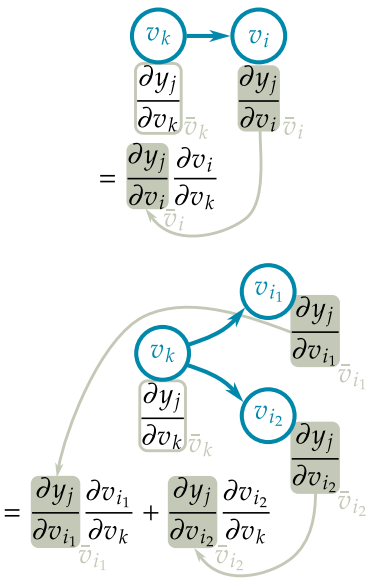

Figure 3.5: The application of the chain rule at a node v_k requires the value of the partial derivatives at the child nodes.

In deep-learning, the network parameters $\boldsymbol{\theta}$ are learned by minimizing a loss function $\mathcal{L}$ using gradient-based (stochastic) minimization. This requires efficiently evaluating the gradient $\nabla_{\boldsymbol{\theta}} \mathcal{L}$, which can be achieved by automatic differentiation in reverse mode, i.e., by back-propagation.

3.2 Stochastic minimization

3.2.1 Improving computational efficiency in the large data regime of machine learning

We presented in section 2.4 the way used to learn the parameters $\boldsymbol{\theta}$ of a neural network from a collection of training samples (pairs $(\boldsymbol{x}_i, \boldsymbol{y}_i^\star)$ of input data and desired output): a loss function $\mathscr{L}$ is first introduced to measure how close an estimate $\boldsymbol{y}_i = f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)$ is to the ground truth $\boldsymbol{y}_i^\star$, then, the parameters $\boldsymbol{\theta}$ that minimize this loss averaged over all the N training samples are sought (eq. 2.5, recalled here):

$$\hat{\boldsymbol{\theta}} \in \arg\min_{\boldsymbol{\theta}} \frac{1}{N} \sum_{i=1}^{N} \mathscr{L}\left(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \boldsymbol{y}_i^\star\right). \tag{3.4}$$

Applying the steepest descent method (i.e., *gradient descent*) to solve this minimization problem gives the sequence of network parameters:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha_t \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} \mathscr{L}\left(f_{\boldsymbol{\theta}_t}(\boldsymbol{x}_i), \boldsymbol{y}_i^\star\right), \tag{3.5}$$

where $\alpha_t \in \mathbb{R}^{+*}$ is the step size (called *learning rate* in machine learning).

Training complex models that reach a high performance requires a huge amount of training data (N is very large, for example: 14 million images in the ImageNet[‡] dataset for image classification and object detection). Performing a single step of gradient descent, eq. (3.5), therefore requires $\mathcal{O}(N)$ evaluations of the gradient $\nabla_{\boldsymbol{\theta}} \mathscr{L}\left(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \boldsymbol{y}_i^\star\right)$ of the loss function, one evaluation for each training pair $(\boldsymbol{x}_i, \boldsymbol{y}_i^\star)$. Since many iterations of gradient descent are necessary to get close to a (local) minimum, maintaining feasible computational times would drastically limit the size N of the training set. Practical considerations have thus led to select only a small subset of the training data for each step in eq. (3.5), and change this subset from one iteration to the next. Beyond computational efficiency, this leads to other advantages such as a reduction of the risk to get stuck in a poor local minimum. Since the samples are selected *randomly* at each step, the minimization algorithm becomes *stochastic* (i.e., it leverages randomization to avoid poor configurations and achieve good performance; the final estimate $\boldsymbol{\theta}_T$ does not depend only on the initial value of the parameters $\boldsymbol{\theta}_0$ and the step lengths α_t, but also on the outcome of the random draws which define each subset.

3.2.2 Stochastic gradient descent

The objective function in equation (3.4) approximates the ideal objective corresponding to the expectation $\mathbb{E}_{(\boldsymbol{x},\boldsymbol{y}^\star)\sim \mathrm{p}(\boldsymbol{x},\boldsymbol{y}^\star)}\left[\mathscr{L}\left(f_{\boldsymbol{\theta}}(\boldsymbol{x}) - \boldsymbol{y}^\star\right)\right]$ defined over the statistical distribution of pairs $(\boldsymbol{x}_i, \boldsymbol{y}_i^\star)$ (which could only be reached with an infinite number of training samples, faithfully representing the actual distribution of data). Let us consider a (small) collection of $M \ll N$ training samples, drawn randomly from the complete training

[‡] https://www.image-net.org/

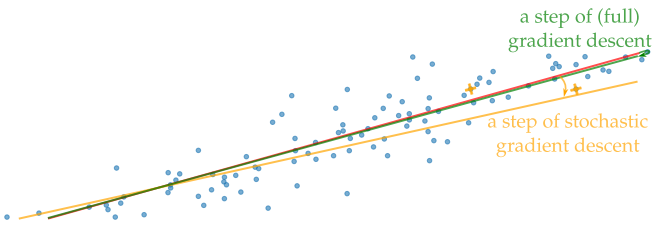

Figure 3.6: A simple model (a straight line) in the process of being fitted to 100 points (the whole training set). Gradient descent uses the gradient of the loss evaluated on the whole dataset while stochastic gradient restricts the evaluation to a small mini-batch (here: the two points marked by yellow crosses). The red model almost reached convergence. Using the same learning rate, the full gradient update only slightly changes the model (red → green line) while stochastic gradient leads to a larger perturbation (red → yellow line) due to the gradient on the mini-batch being 10 times larger (the model is a poor fit for the two points of the mini-batch). Close to convergence, much smaller learning rates are required for the stochastic gradient descent algorithm.

set. Such a collection is called a *mini-batch*. The gradient $\widehat{\boldsymbol{g}}_t$ evaluated at step t on a mini-batch forms an unbiased estimate of the true gradient:

$$\widehat{\boldsymbol{g}}_t = \frac{1}{M} \sum_{i=1}^{M} \nabla_{\boldsymbol{\theta}} \mathscr{L} \left(f_{\boldsymbol{\theta}_t}(\boldsymbol{x}_{(i)}), \boldsymbol{y}^\star_{(i)} \right), \tag{3.6}$$

with

$$\mathbb{E}_{(\boldsymbol{x},\boldsymbol{y}^\star)\sim \mathrm{p}(\boldsymbol{x},\boldsymbol{y}^\star)} \left[\widehat{\boldsymbol{g}}_t \right] = \nabla_{\boldsymbol{\theta}} \, \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y}^\star)\sim \mathrm{p}(\boldsymbol{x},\boldsymbol{y}^\star)} \left[\mathscr{L} \left(f_{\boldsymbol{\theta}_t}(\boldsymbol{x}) - \boldsymbol{y}^\star \right) \right], \tag{3.7}$$

where the M samples of the mini-batches are identified with the indices (1) to (M).

If M is chosen very small, then the estimate of the gradient gets quite noisy but fast to compute. The choice $M = N$ corresponds to using the whole training set to evaluate the gradient, which is generally too costly.

The stochastic gradient descent algorithm consists in replacing the full-gradient in (3.5) by the approximation $\widehat{\boldsymbol{g}}_t$, computed over a different mini-batch at each step:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha_t \widehat{\boldsymbol{g}}_t. \tag{3.8}$$

Setting the value of the learning rate α_t differs in gradient descent and in stochastic gradient descent. When $\mathscr{L} \left(f_{\boldsymbol{\theta}}(\boldsymbol{x}_{(i)}), \boldsymbol{y}^\star_{(i)} \right)$ is strongly convex and smooth, with bounded curvature, a constant learning rate can be used in the gradient descent algorithm (the largest learning rate value allowing convergence depends on the curvature). In more general cases (such as non-convex problems), a line-search algorithm should be applied to find a step-size α_t ensuring sufficient progress at each iteration. It is also recommended using another descent direction than the steepest descent, e.g. using a preconditioner or an approximation of the inverse of the Hessian such as BFGS, see the course on Numerical Optimization. With stochastic gradient descent, we cannot keep a constant learning rate: it must decrease throughout the iterations otherwise the "noise" due to samples ignored when restricting to a mini-batch will lead to an everlasting fluctuation of the iterates $\boldsymbol{\theta}_t$ (the gradient does not vanish when we get closer to a local minimum, see figure 3.6). That said, reducing too quickly the learning rate may stop abruptly the evolution of the iterates. (Bottou, 1998) proved that, under mild hypotheses, it suffices to choose learning rates such that $\sum_{t=1}^{\infty} \alpha_t = \infty$ and $\sum_{t=1}^{\infty} \alpha_t^2 < \infty$, such as $\alpha_t = \alpha_0/(1 + \mu t)$ (where the initial learning rate α_0 and the parameter μ can be set by trial and error). In practice, various choices for the change of the learning rate are possible, often based on empirical observations, and the evolution may even be non-monotonic (starting with a small learning

rate, increasing it to a larger value, then slowly reducing it to reach much lower values), see the options to set different learning rate schedulers in deep learning libraries (for example `lr_scheduler.py` in Pytorch).

Using only a small subset of the training data at once when evaluating the gradient of the loss function drastically improves the computational complexity and also contributes to avoiding some poor local minima thanks to the perturbations with respect to the full gradient.

during an *epoch*, all training samples are seen once

Covering all samples in the training set, after many mini-batches, corresponds to one *epoch* of training. Reporting how many times the whole dataset has been seen, i.e., how many epochs were conducted during training, is often informative.

There are several variants of the stochastic gradient descent algorithm. To prevent the stochastic gradient updates to wander too much when getting close to a (local) minimum, a memory of the previous direction can be introduced:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \boldsymbol{\Delta}_{t+1} \text{ with } \boldsymbol{\Delta}_{t+1} = \beta \boldsymbol{\Delta}_t - \alpha_t \widehat{\boldsymbol{g}}_t \tag{3.9}$$

where $\beta < 1$ controls the weight of the memory of the past descent directions ($\beta = 0$ gives the standard stochastic gradient descent). Descent directions $-\alpha_t \widehat{\boldsymbol{g}}_t$ provided by the mini-batch gradients are noisy. The updates $\boldsymbol{\Delta}_{t+1}$ correspond to a smoothed version of these noisy directions (corresponding to a recursive filter, i.e., an infinite impulse response linear filter, see course on digital filtering: this corresponds to an exponential moving average). Another interpretation of equation (3.9) comes from mechanics. It consists in interpreting the network parameters $\boldsymbol{\theta}_t$ as a point in space, and $\boldsymbol{\Delta}_t$ as the velocity of that point. The variation of $\boldsymbol{\Delta}_t$ from one iterate to the next (i.e., the acceleration) is $(\beta - 1)\boldsymbol{\Delta}_t - \alpha_t \widehat{\boldsymbol{g}}_t$. According to Newton's law of motion, for a unit mass point, this corresponds to the sum of forces applied to it. The steepest descent direction $-\alpha_t \widehat{\boldsymbol{g}}_t$ can be interpreted as a force (deriving from the potential $\mathcal{L}$), pulling the point downhill (to low values of the loss) while the term $(\beta - 1)\boldsymbol{\Delta}_t$ corresponds to a viscous drag (that dampens the oscillations). While the stochastic gradient descent instantaneously reacts to changes of the gradient, the update of equation (3.9) produces some *momentum*: the velocity also depends on the past velocity so that flat areas can be crossed without abruptly stopping and zigzagging phenomena in narrow valleys are reduced. This method is often called *stochastic gradient descent with momentum* and dates back from (Polyak, 1964).

The impact of the fluctuations of stochastic minimization algorithms can be reduced be either keeping the values of the parameters that reached the best performance so far (rather than the last iterate), or by averaging parameters $\boldsymbol{\theta}_t$ along the trajectory (Polyak and Juditsky, 1992). Recent works have found that averaging the parameters using an exponential moving average drastically improves the quality of the estimates[§], in

[§] note the difference with stochastic gradient descent with momentum where exponential moving average filtering was applied to the mini-batch gradients to reduce noise and this strategy that directly averages the parameters to obtain an "average model" with interesting properties in terms of inference

particular at the early stages of the optimization, see (Morales-Brotons et al., 2024).

There are many variants of the stochastic gradient descent not covered here. ADAM (Kingma and Ba, 2014) is a widely used algorithm that is also based on the idea of averaging values of the mini-batch gradients to reduce noise. It performs very well in practice.

3.2.3 Practical issues

Unlike (striclty) convex minimization problems which benefit from the uniqueness of the minimizer, minimization algorithms to train a neural network only give *approximate solutions* (no guarantee to reach the global minimum: many local minima and flat areas present), and these solutions depend heavily on the *initial value* of the network parameters.

Since we have no real clue of good initial values, we merely try to avoid really bad ones. A terrible choice would be to use the same initialization value for all the parameters of the network. In that case, two neurons located in the same layer of the network would receive identical inputs and their output would contribute equally to the loss function. The gradient of the loss function with respect to the weights associated with each one of these two neurons would be identical, so their evolution would be identical throughout the iterations and weights would remain equal forever. For neurons within a given layer to extract different things from the same input, they must be initialized with diverse weights (symmetries in the architecture must be broken by asymmetries in the weight initializations). A simple way to ensure sufficient diversity is to initialize the network parameters *randomly* (e.g., i.i.d. uniform or Gaussian distributed). Another important issue is the *scale* of the parameters that can lead to numerical errors (underflows or overflows) when evaluating the loss function or its gradient, or difficulties for training (e.g., very slow progress due to too small gradient values, i.e., the *vanishing gradient* problem, or instabilities due to large gradient values, i.e., the *exploding gradient*). Several strategies can be applied to solve these issues:

1. *careful initialization* of the network weights,
2. *normalization steps*,
3. avoiding activation functions with a *saturation* behavior, and introducing *skip connections*,
4. *clipping* large gradient values,
5. *non-monotonic learning rate schedulers*.

Initialization of the network weights: The k-th layer of an MLP applies an affine transform to the previous hidden vector $\boldsymbol{h}_{k-1} \in \mathbb{R}^{d_{k-1}}$ in order to produce, after application of the activation function, the k-th hidden vector $\boldsymbol{h}_k = \underline{g}_k(\mathbf{W}_k \boldsymbol{h}_{k-1} + \boldsymbol{b}_k) \in \mathbb{R}^{d_k}$. The parameters of the affine transform should be initialized such that the scale of the hidden vector values is preserved[¶]. To simplify the analysis, we neglect the effect of the activation function (linear layer, or linear regime of a non-linear

[¶] remember how deep neural networks are obtained by stacking numerous layers: if the hidden vector values are amplified by a factor β by each layer, after d layers we get an amplification by a factor β^d leading to exploding values when d is large if $\beta > 1$, or vanishing values if $0 < \beta < 1$.

activation function) and assume that $\boldsymbol{b}_k = \mathbf{0}$ (biases are initialized at 0). The scale of a hidden vector $\boldsymbol{h}_k$ can be characterized by $\frac{1}{d_k}\|\boldsymbol{h}_k\|_2^2$. If we initialize the weight matrix $\mathbf{W}_k$ randomly, with centered independent and identically distributed (i.i.d.) entries, with second-order moment $\mathbb{E}\{[\mathbf{W}_k]_{ij}^2\} = \sigma^2$, we can express the scale of the hidden vector $\boldsymbol{h}_k$ as a function of the scale of the previous hidden vector $\boldsymbol{h}_{k-1}$:

$$\begin{aligned}
\mathbb{E}_{\mathbf{W}_k}\left\{\frac{1}{d_k}\|\boldsymbol{h}_k\|_2^2\right\} &= \mathbb{E}_{\mathbf{W}_k}\left\{\frac{1}{d_k}\left\|\underline{g}_k(\mathbf{W}_k\boldsymbol{h}_{k-1}+\boldsymbol{b}_k)\right\|_2^2\right\} \\
&= \mathbb{E}_{\mathbf{W}_k}\left\{\frac{1}{d_k}\|\mathbf{W}_k\boldsymbol{h}_{k-1}\|_2^2\right\} \text{ under our assumptions} \\
&= \mathbb{E}_{\mathbf{W}_k}\left\{\frac{1}{d_k}\sum_{i=1}^{d_k}[\mathbf{W}_k\boldsymbol{h}_{k-1}]_i^2\right\} \\
&= \mathbb{E}_{\mathbf{W}_k}\left\{\frac{1}{d_k}\sum_{i=1}^{d_k}\left(\sum_{j=1}^{d_{k-1}}[\mathbf{W}_k]_{ij}[\boldsymbol{h}_{k-1}]_j\right)^2\right\} \\
&= \mathbb{E}_{\mathbf{W}_k}\left\{\frac{1}{d_k}\sum_{i=1}^{d_k}\sum_{j=1}^{d_{k-1}}\sum_{\ell=1}^{d_{k-1}}[\mathbf{W}_k]_{ij}[\mathbf{W}_k]_{i\ell}[\boldsymbol{h}_{k-1}]_j[\boldsymbol{h}_{k-1}]_\ell\right\} \\
&= \frac{1}{d_k}\sum_{i=1}^{d_k}\sum_{j=1}^{d_{k-1}}\mathbb{E}_{\mathbf{W}_k}\left\{[\mathbf{W}_k]_{ij}^2\right\}[\boldsymbol{h}_{k-1}]_j^2 \text{ (since } [\mathbf{W}_k]_{ij} \text{ are i.i.d. and centered)} \\
&= \sigma^2 \cdot \|\boldsymbol{h}_{k-1}\|_2^2. \qquad (3.10)
\end{aligned}$$

Hidden vector values are neither amplified nor attenuated if we initialize the weights independently according to a distribution with a variance equal to $1/d_{k-1}$, e.g. if we draw each weight $[\mathbf{W}_k]_{ij}$ according to a Gaussian distribution $\mathcal{N}(\mathbf{0}, 1/d_{k-1})$ and set $\boldsymbol{b}_k = \mathbf{0}$. Beyond the preservation of the scale of the hidden values, the scale of the gradient should also be preserved to avoid the vanishing gradient / exploding gradient issues. This leads to drawing weights according to $\mathcal{N}(\mathbf{0}, 1/d_k)$, and, to achieve a trade-off between hidden value normalization and gradient value normalization, to use the distribution $\mathcal{N}(\mathbf{0}, 2/(d_{k-1}+d_k))$, see (Glorot and Bengio, 2010) referred to as *Xavier's initialization*. The analysis for ReLU activation functions led (He et al., 2015a) to recommend the initialization according to $\mathcal{N}(\mathbf{0}, 2/d_k)$ (i.e., He's initialization uses twice the variance of Xavier's initialization).

Normalization steps: Training stability of very deep neural networks can be largely improved by controlling the normalization of the hidden vector values not only at initialization but also throughout the optimization. The principle is to center and standardize the activations of the network at each layer. This is achieved by the following affine transform:

$$[\widehat{\boldsymbol{h}}_k]_i = \frac{[\boldsymbol{h}_k]_i - \widehat{\mu}_{k,i}}{\widehat{\sigma}_{k,i}}\,, \qquad (3.11)$$

where the empirical mean $\widehat{\mu}_{k,i}$ and standard deviation[‖] $\widehat{\sigma}_{k,i}$ are computed over all activation values for a given channel (for batch normalization),

[‖] to prevent divisions by 0, a small positive offset is added to the empirical variance

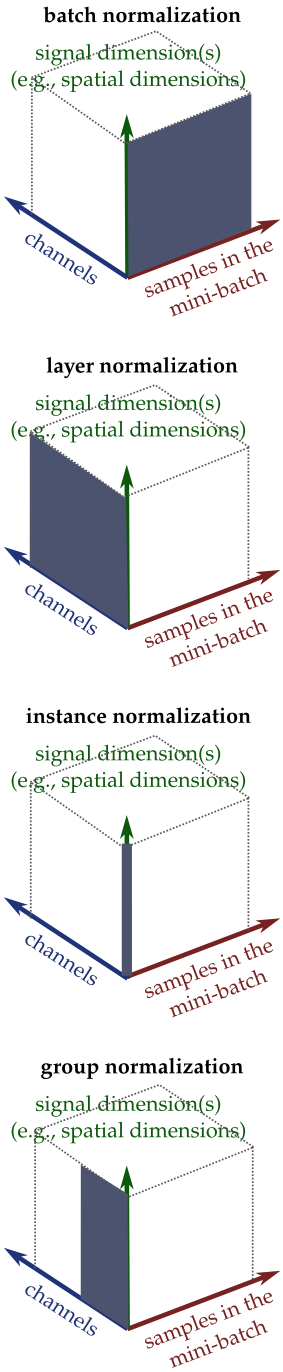

Figure 3.7: Different normalization strategies for the hidden vectors: empirical mean and variance are computed over the subset represented in gray. Adapted from (Wu and He, 2018).

for a given training sample (for layer normalization), for a given training sample and channel (for instance normalization), or for a given training sample and subset of channels (for group normalization), see Fig. 3.7 and (Wu and He, 2018).

To preserve the ability of the network activation functions to operate on non-standardized inputs, a scale parameter $\boldsymbol{\gamma}$ and shift parameter $\boldsymbol{\beta}$ are introduced:

$$[\text{normalize}(\boldsymbol{h}_k)]_i = [\boldsymbol{\gamma}_k]_i \cdot [\widehat{\boldsymbol{h}}_k]_i + [\boldsymbol{\beta}_k]_i \,, \tag{3.12}$$

where $\boldsymbol{\gamma}_k$ and $\boldsymbol{\beta}_k$ vary along the same dimensions as the empirical mean and standard deviations $\widehat{\mu}_{k,i}$ and $\widehat{\sigma}_{k,i}$ (e.g., for batch normalization, there is a different scale and shift parameter for each channel).

With these normalization strategies, deep neural networks can be trained faster, using larger learning rates (Ioffe and Szegedy, 2015). At test time, the parameters $\widehat{\mu}_{k,i}$ and $\widehat{\sigma}_{k,i}$ are either computed on the new sample(s) or values used at the end of the training are stored for inference time (these can be obtained by exponentially weighted moving average, see for example the role of parameter `momentum` in Pytorch class `BatchNorm2d`).

Combating the effect of the vanishing gradient with non-saturating activations and skip connections: Saturating activation functions like the hyperbolic tangent were once widely used in neural networks. Yet, these functions have a derivative that tends to 0 when the argument tends to $\pm\infty$. Getting out of a region of the parameter space that produces saturations of the activation functions at some layers of the network can require a considerable amount of time (the gradient is almost 0, so the stochastic gradient descent makes tiny steps). Non-saturating activation functions like ReLU have proven much more convenient to train deep neural networks and are widely used in modern architectures.

Another issue is that, with increasing depths, the gradient for the first layers may become very small. An efficient way to improve the training of deep architecture consists of introducing skip connections, i.e., allowing information to flow directly to upper layers without being transformed by intermediate layers. For a feedforward network such as an MLP, the i-th layer is modified in the following way:

$$f_{\boldsymbol{\theta}_i}^{(i)} : \boldsymbol{h}_{i-1} \mapsto \boldsymbol{h}_i = \underline{g}_i(\mathbf{W}_i \boldsymbol{h}_{i-1} + \boldsymbol{b}_i) + \boldsymbol{h}_{i-1}. \tag{3.13}$$

If you compare with equation (2.3), you will notice that the input activations $\boldsymbol{h}_{i-1}$ are forwarded to the output of the layer (the last term $+\boldsymbol{h}_{i-1}$). In this way, the first layers not only influence the loss function through the effect of the deeper layers, but also more directly. As illustrated in figure 3.8, this modifies the loss landscape by reducing non-convexity areas and widening good-quality minima. This also ensures that, by increasing the depth of a network, we define a larger family of parametric functions $f_{\boldsymbol{\theta}}$ that includes all the parametric functions corresponding to shorter depths**.

** setting $\mathbf{W}_i = \mathbf{0}$ and $\boldsymbol{b}_i = \mathbf{0}$ for the additional layers leads to a deeper network that is strictly equivalent to the original network

Gradient clipping: In very steep areas of the loss landscape, we can encounter large values of the gradient that can push the current parameters $\boldsymbol{\theta}_t$ too far away. To prevent such behavior, a *clipping* of the gradient vector $\widehat{\boldsymbol{g}}_t$ can be applied:

$$\widehat{\boldsymbol{g}}_t^{\text{clipped}} = \begin{cases} \widehat{\boldsymbol{g}}_t & \text{if } \|\widehat{\boldsymbol{g}}_t\| \leq v \\ \widehat{\boldsymbol{g}}_t \frac{v}{\|\widehat{\boldsymbol{g}}_t\|} & \text{otherwise.} \end{cases} \tag{3.14}$$

This clipping operation preserves the direction of the gradient but bounds the norm of the gradient to a maximum value v.

Using non-monotonic learning rate schedulers: We discussed in paragraph 3.2.2 that the learning rate of the stochastic gradient descent should decrease to reach convergence. The starting value of the learning rate is constrained by the curvature of the loss function in the area of parameter space corresponding to the initialization. This can constrain the learning rate to moderate values. *Warmup* consists of a progressive increase of the learning rate from a small value (or zero) to a much larger target learning rate that can be applied only to well-behaved areas of the loss landscape. This warmup phase brings the parameters to a region where larger steps can safely be taken, see (Kalra and Barkeshli, 2024).

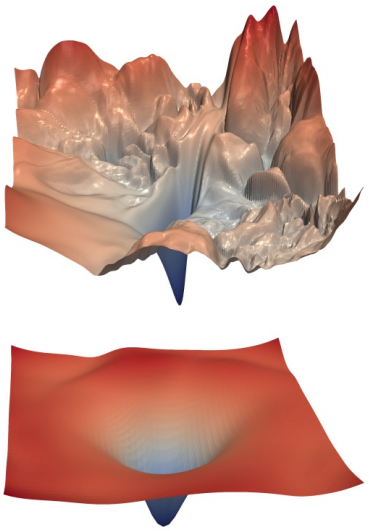

Figure 3.8: The loss landscape is much better behaved when skip connections are used: ResNet-56 without (top) or with (bottom) skip connections. Illustration from (Li et al., 2018).

Convolutional Neural Networks 4

Convolution Neural Networks (CNNs) emerged at the end of the 80s (LeCun et al., 1989) to recognize handwritten zip codes. Images are high-dimensional. To obtain neural networks that generalize well to unseen images, it is necessary to reduce the number of parameters of neural nets by designing specific architectures. To recognize patterns, local operations may be sufficient. Moreover, similar shapes can occur at different spatial locations. It makes sense to share the same network weights when processing various spatial locations. The linear operators $\{\mathbf{W}_i\}_{i=1..H}$ of the Multi Layer Perceptron can then be constrained to obey specific structures, corresponding to discrete convolution operations.

Beyond their huge success in image processing and computer vision, CNNs have been very effective in other domains, such as learning representations to decide next moves in the game of Go (Silver et al., 2016), predicting the 3D shape of a protein from its amino acid sequence (Senior et al., 2020), see Fig. 4.1, or environmental sound classification (Salamon and Bello, 2017).

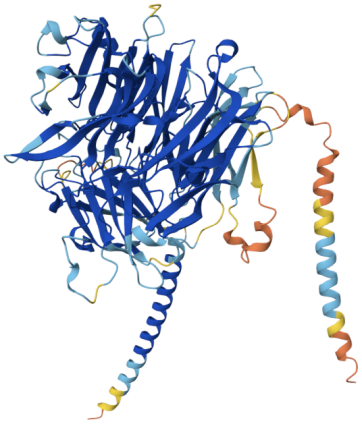

Figure 4.1: 3D structure of a protein that may protect against the malaria parasite, predicted using a convolutional neural network (alpha-fold). The breakthrough made by the designers of this approach earned them the 2024 Nobel Prize in chemistry. Source: https://alphafold.ebi.ac.uk/entry/Q8I3H7

4.1 The building bricks of CNNs

Convolution Neural Networks are Multi Layer Perceptrons whose linear operators $\{\mathbf{W}_i\}_{i=1..H}$ obey special structures in order to (i) better extract information from grid-based multidimensional data, (ii) drastically reduce the number of free parameters.

The network input and hidden vectors are represented as *tensors*, i.e., multidimensional arrays, with dimensions corresponding to:

- the mini-batch dimension* N,
- the channels C (e.g., 3 for a color image, 2 for stereo audio tracks, the number of feature maps for a hidden vector),
- the spatial dimensions of the data (e.g., length L of a 1D signal, size $H \times W$ of a 2D image, size $D \times H \times W$ of a volumetric image).

These tensors can be transformed by various operations described in the following paragraphs.

4.1.1 The "convolution" operation

The discrete convolution of a d-dimensional input signal $\boldsymbol{h}$, of size $D_1 \times \cdots \times D_d$, with a d-dimensional kernel $\boldsymbol{k}$, of size $K_1 \times \cdots \times K_d$, is defined in signal processing by:

$$[\boldsymbol{s}]_{i_1,i_2,..,i_d} = [\boldsymbol{h} * \boldsymbol{k}]_{i_1,i_2,..,i_d} = \sum_{j_1,...,j_d} [\boldsymbol{h}]_{j_1,j_2,..,j_d} [\boldsymbol{k}]_{i_1-j_1,i_2-j_2,..,i_d-j_d}. \quad (4.1)$$

* processing all samples of a mini-batch at once allows for optimized implementations (parallelization) and is required for some operations such as batch normalization

Note how, in the usual mathematical definition of convolution, the kernel $\boldsymbol{k}$ (seen as a function of the indices $j_1, \ldots, j_d$ used in the summation) is flipped along each axis. If we don't flip the kernel, the operation is called a cross-correlation (the symbol $\star$ is used to denote a correlation, while $*$ indicates a convolution):

$$[\boldsymbol{s}]_{i_1,i_2,..,i_d} = [\boldsymbol{h} \star \boldsymbol{k}]_{i_1,i_2,..,i_d} = \sum_{j_1,\ldots,j_d} [\boldsymbol{h}]_{j_1,j_2,..,j_d} [\boldsymbol{k}]_{j_1-i_1,j_2-i_2,..,j_d-i_d}. \quad (4.2)$$

Now, we see that the kernel is just shifted so that its center is positioned at the location $(i_1, i_2, \ldots, i_d)$ where we evaluate the output $\boldsymbol{s}$.

warning: most deep learning libraries and technical documents about deep learning use the term *convolution* in lieu of *cross-correlation*

In deep learning, the "convolution" operation is defined in a slightly different way compared to the usual mathematical and signal processing usage (hence the use of quotes). The first difference is that cross-correlations are used instead of convolutions in deep learning. Since weights are learned, it does not matter whether we learn a kernel or its flipped version as long as the operations performed are consistent in the learning and inference phases. The second difference is that the deep-learning "convolution" operation operates on multi-channel multi-dimensional signals. The channel dimension plays a special role and is not handled the same way as the other dimensions. It is often desired to change[†] the number of channels from one layer to the next of a CNN. Different channels can be thought of as distinct feature maps that represent complementary information extracted from the input data. Each feature map produced by a layer is, therefore, obtained by a different combination of the input channels:

$$[\boldsymbol{s}]_{n,c,\bullet} = [\mathsf{conv}(\boldsymbol{h}, \boldsymbol{k})]_{n,c,\bullet} = \sum_{\tau=1}^{C_{\text{in}}} ([\boldsymbol{h}]_{n,\tau,\bullet}) \star ([\boldsymbol{k}]_{c,\tau,\bullet}), \quad (4.3)$$

where conv denotes the "convolution" operation used in CNNs and $\boldsymbol{h}$, $\boldsymbol{k}$, and $\boldsymbol{s}$ are respectively $N \times C_{\text{in}} \times D_1 \times \cdots \times D_d$, $C_{\text{out}} \times C_{\text{in}} \times K_1 \times \cdots \times K_d$, and $N \times C_{\text{out}} \times D'_1 \times \cdots \times D'_d$ tensors. To simplify the notations, we replaced all spatial indices by $\bullet$, the cross-correlation operation $\star$ is applied along these dimensions.

So, for each sample n of the mini-batch (i.e., each signal/image/volume), multi-dimensional correlations are performed using different weights for each input channel τ, and the results of these correlations are summed to produce a given output channel c. Then, a completely different set of convolution weights is used to compute the next output channel: see figure 4.3.

Note that the operation in (4.3) is linear, i.e., it could be represented by the product of a matrix $\mathbf{W}$ with the input vector $[\boldsymbol{h}]_{n,\cdot,\bullet}$. Yet, not all matrices $\mathbf{W}$ can be factored in the form of equation (4.3): the conv operator is highly structured and depends on much fewer parameters ($C_{\text{out}} \times C_{\text{in}} \times K_1 \times \cdots \times K_d$ instead of $C_{\text{out}} \times D'_1 \times \cdots \times D'_d \times C_{\text{in}} \times D_1 \times \cdots \times D_d$, where $K_\cdot$ are typically equal to 3 or 5 while $D_\cdot$ and $D'_\cdot$ are often in order of tens or hundreds), see the example shown in figure 4.2. Since kernels with short spatial support are usually considered (to reduce the number of parameters that must be learned), the correlation operations are usually

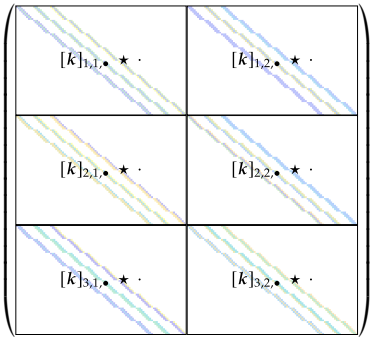

Figure 4.2: The matrix representation $\mathbf{W}$ of an operator conv, represented for $C_{\text{in}} = 2$, $C_{\text{out}} = 3$, $D_1 = 10$, $D_2 = 10$, $K_1 = 3$, and $K_2 = 3$. Zeros in the matrix $\mathbf{W}$ appear in white: they are numerous. Note that within each cross-correlation block the same $K_1 \cdot K_2$ values are repeated at each line due to the convolution structure. In this example, among the 38,400 elements of the 192×200 matrix $\mathbf{W}$, only 3,456 are non-zero, and as little as 54 parameters define the whole operator. (Note: no zero-padding is used here, so the spatial dimensions are reduced respectively by $K_1 - 1$ and $K_2 - 1$, giving the rectangular appearance of each block).

[†] to get full control over the number of channels in output, an operation different from a convolution must be performed in the direction of channels

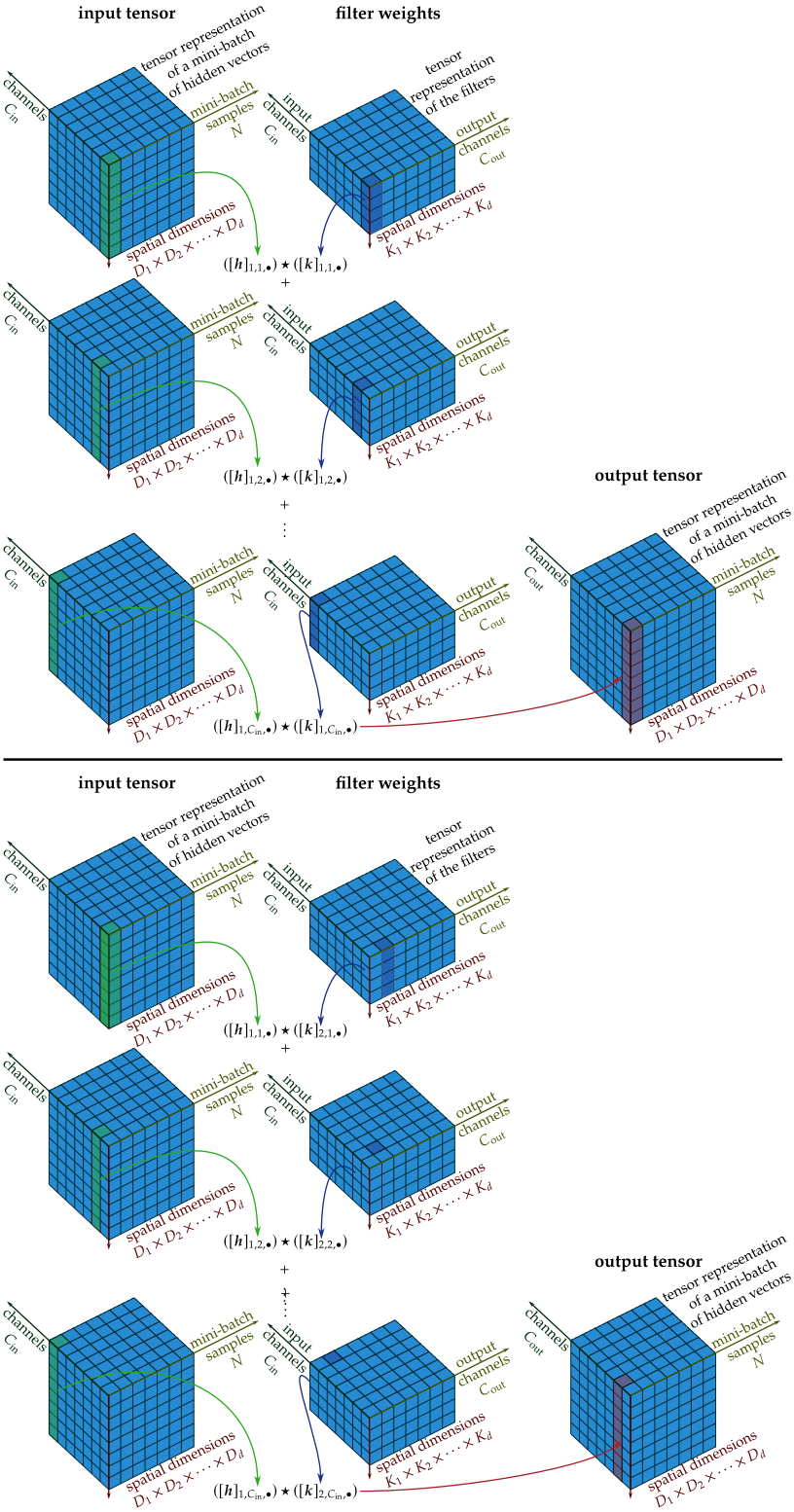

Figure 4.3: The conv operation in deep learning operates on tensors. It sums the cross-correlations between corresponding channels of the input and filter tensors. This is repeated for each output channel and each element of the mini-batch. Here, the computations involved in producing the result of conv for the first two output channels of the first sample of the batch are illustrated.

computed in the direct domain rather than in Fourier domain. On GPUs, these operations can be performed massively in parallel.

⚙ Exercise: interpretation of deep learning's 1×1 convolution

It is common practice in CNNs to have some layers perform a 1×1 convolution. How could you describe this operation mathematically?

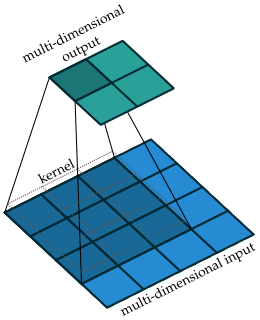

Figure 4.4: Keeping only the valid area of the convolution by a $K_1 \times \cdots \times K_d$ kernel removes a border of size $\lfloor K_1/2 \rfloor \times \cdots \times \lfloor K_d/2 \rfloor$ in the resulting image. Adapted from (Dumoulin and Visin, 2016).

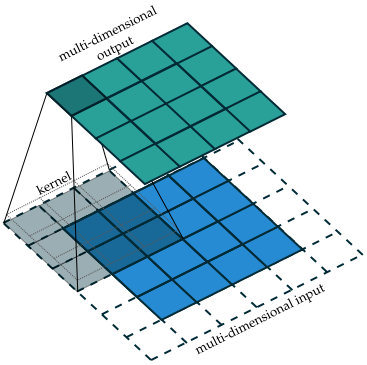

Figure 4.5: Padding the input image with some values produces a convolution result that can preserve the size of the image. Adapted from (Dumoulin and Visin, 2016).

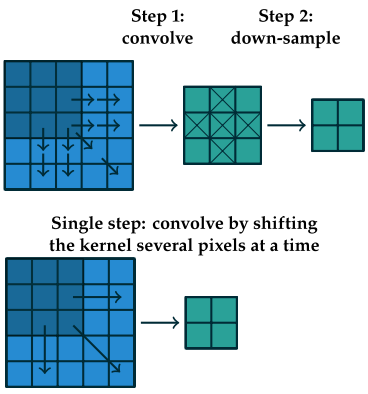

Figure 4.6: Rather than throwing away values during down-sampling, convolutions with strides shift the kernel by more than a pixel to produce an output of the correct size. Adapted from (Dumoulin and Visin, 2016).

So far, we left out technical details about how the borders are handled in the convolutions/correlations. The usual choices of signal/image processing can be considered:

- keeping only the *valid* area, as illustrated in Fig. 4.4,
- padding the input image by some arbitrary value (e.g., 0) to artificially increase its size and get a result of desired dimension (typically: same size as the original image), see Fig. 4.5,
- the input image can also be padded by replicating the value of the closest pixel, by reflection along the image borders, or circularly.

A nice property‡ of CNNs is that convolution operations can be applied to input images of arbitrary spatial dimension. This means that a fully-convolutional model trained for example on 512×512 images can then be applied to images of a different size such as 1280×720. This is not possible for an MLP since the weight matrices $\mathbf{W}_i$ are arrays of fixed dimension. Processing input data with a different size requires either a resizing step before applying the MLP, or retraining a whole network from scratch.

4.1.2 Down-sampling and convolutions with strides

If the task requires to output a low-dimensional result from high-dimensional data (e.g., decide whether an image represents an *indoor* or *outdoor* scene), then the intermediate representations within the network should be of lower spatial dimension when we reach deeper and deeper layers. Reducing the spatial resolution can be done by down-sampling. Yet, it is inefficient to compute the result of a `conv` operation for all pixels and then drop many pixel values during down-sampling. By doing jointly both operations, only the pixels kept after down-sampling are actually considered during the `conv` operation. Specifying how many pixels to skip during the `conv` operation is done with the *stride* parameter, which represents the shift, in pixels, applied to the kernel when moving from one position to the next, see Fig. 4.6.

4.1.3 Up-sampling and transposed convolutions

In an MLP, controlling the dimension of successive layers is very easy: it is directly defined by the dimension of the weight matrix $\mathbf{W}_i \in \mathbb{R}^{d_i \times d_{i-1}}$. A tall matrix leads to an increase of the dimension of the hidden vector while a wide matrix reduces its dimension, i.e., matrix transposition reverses the behavior of the layer in terms of dimensionality. Several network architectures have a symmetrical design, for example,

‡ if down-sampling and up-sampling operations are involved, there may be some restrictions on the size of the input image, e.g., being a multiple of some power of 2

the encoder-decoder architecture progressively reduces the dimension until a *compressed representation* is obtained (the so-called *latent code*, at the *bottleneck* of the network). Then, the second half of the network interprets this code to reproduce an image with the original full-size (a *decoding process*). If conv operations are applied to obtain the latent code, what symmetrical operations should be applied in the decoder to progressively recover the full size of the image? What operations would correspond to the transpose of the linear operator associated with the conv?

To answer this question, let us go back to the definition of the transpose (or adjoint) of a linear operator $W : U \to V$

$$\forall x \in U, \forall y \in V, \langle Wx, y \rangle = \langle x, W^\intercal y \rangle \tag{4.4}$$

If W corresponds to the conv operator, then according to equation (4.3) we have:

$$\forall \boldsymbol{x} \in \mathbb{R}^{N \times C_x \times D_1 \times \cdots \times D_d}, \forall \boldsymbol{y} \in \mathbb{R}^{N \times C_y \times D'_1 \times \cdots \times D'_d}, \forall \boldsymbol{k} \in \mathbb{R}^{C_y \times C_x \times K_1 \times \cdots \times K_d},$$

$$\langle \mathsf{conv}(\boldsymbol{x}, \boldsymbol{k}), \boldsymbol{y} \rangle = \sum_{n=1}^{N} \sum_{c=1}^{C_y} \sum_{i_1,..,i_d} [\boldsymbol{y}]_{n,c,i_1,..,i_d} \sum_{\tau=1}^{C_x} \{([\boldsymbol{x}]_{n,\tau,\bullet}) \star ([\boldsymbol{k}]_{c,\tau,\bullet})\}_{i_1,..,i_d} .$$

By application of the definition of the correlation $\star$ given in equation (4.2), and re-ordering the terms, we get:

$$\begin{aligned} \langle \mathsf{conv}(\boldsymbol{x}, \boldsymbol{k}), \boldsymbol{y} \rangle &= \sum_{n=1}^{N} \sum_{\tau=1}^{C_x} \sum_{j_1,..,j_d} [\boldsymbol{x}]_{n,\tau,j_1,j_2,..,j_d} \\ &\qquad \cdot \sum_{c=1}^{C_y} \sum_{i_1,..,i_d} [\boldsymbol{k}]_{c,\tau,j_1-i_1,j_2-i_2,..,j_d-i_d} [\boldsymbol{y}]_{n,c,i_1,..,i_d} \\ &\stackrel{\text{def}}{=} \langle \boldsymbol{x}, \mathsf{conv}^\intercal(\boldsymbol{y}, \boldsymbol{k}) \rangle \end{aligned}$$

which gives, by identification:

$$[\mathsf{conv}^\intercal(\boldsymbol{y}, \boldsymbol{k})]_{n,\tau,\bullet} = \sum_{c=1}^{C_y} ([\boldsymbol{y}]_{n,c,\bullet}) * ([\boldsymbol{k}]_{c,\tau,\bullet}) . \tag{4.5}$$

Note that the correlation has become a (mathematical) convolution and that the kernels vary along the first index in the summation.

The same reasoning could be repeated with the more general formulæ that account for down-sampling or zero-padding. Let us discuss the impact of down-sampling based on the product of linear operators: $\mathtt{strided\_conv}(\bullet, \boldsymbol{k}) = \mathbf{D} \cdot \mathsf{conv}(\bullet, \boldsymbol{k})$, where $\mathbf{D}$ is a down-sampling operator. In 1D, a down-sampling matrix transforming a length 6 vector into a length 3 vector takes the form:

$$\mathbf{D} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}. \tag{4.6}$$

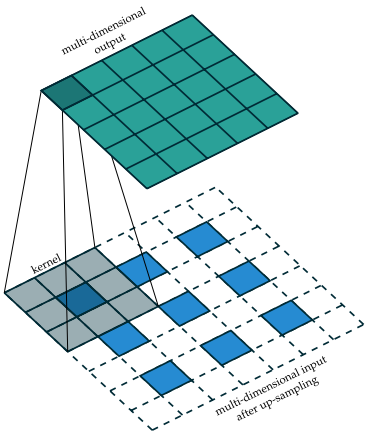

Figure 4.7: The transpose of a convolution with a non-unitary stride is an operation that increases the size of the image: it corresponds to filtering an up-sampled version of the input image. Adapted from (Dumoulin and Visin, 2016).

The transpose operator corresponds to

$$\mathbf{D}^\intercal = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \tag{4.7}$$

which corresponds to up-sampling, i.e., inserting a zero between each sample. The basic transposition rule $(\mathbf{AB})^\intercal = \mathbf{B}^\intercal \mathbf{A}^\intercal$ then gives:

$$\texttt{strided\_conv}^\intercal(\bullet, \boldsymbol{k}) = \texttt{conv}^\intercal(\bullet, \boldsymbol{k}) \cdot \mathbf{D}^\intercal, \tag{4.8}$$

which means that the effect is the same as up-sampling the signal first, then summing the convolutions of each channel with the corresponding d-dimensional filter in $\boldsymbol{k}$. Obviously, rather than performing multiplications with many zeros, efficient implementations of the transposed convolution with non-unitary strides do the computations in a single step.

It is important to note that, in most cases, the transposed convolution does not invert the original convolution (only unitary operators are such that their adjoint is also their inverse, e.g., rotations, reflections, orthonormal transforms like the Fourier transform...; when convolving with a low-pass kernel, the transpose also produces a low-pass effect, so we get a result twice as blurry as the original rather than inverting the blur by applying the adjoint). Yet, the transposed convolution is sometimes called a deconvolution, a term that we prefer to avoid in this context. The transposed convolution is also an essential operation to perform the *back-propagation*: it is required to compute the gradient by automatic differentiation (this is transparent to the user, though).

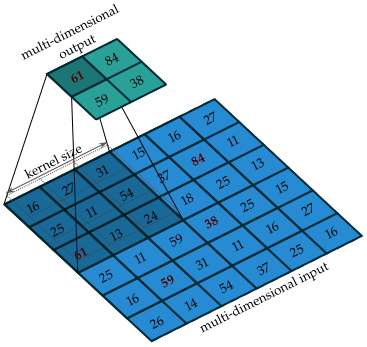

Figure 4.8: Max-pooling extracts the maximum value inside a small window (the kernel), then shifts the window by f pixels (the stride). It reduces the spatial resolution while preserving local maxima. Adapted from (Dumoulin and Visin, 2016).

4.1.4 Non-linear filtering: max-pooling

Mathematical morphology operations are essential transforms in image processing. Non-linear operations such as the dilation and erosion correspond to the computation of order statistics in a local neighborhood (min or max operations). Combined with a down-sampling operation, they can be used to preserve extremum values at a coarser scale. The `max_pooling` operation is widely used; it corresponds to a dilation with an $f \times f$ square, followed by down-sampling by a factor f:

$$[\texttt{max\_pooling}(\boldsymbol{h})]_{n,c,i_1,i_2,\ldots,i_d} = \max_{0 \le \delta_1 < f, \cdots, 0 \le \delta_d < f} [\boldsymbol{h}]_{n,c,f \cdot i_1 + \delta_1, f \cdot i_2 + \delta_2, \ldots, f \cdot i_d + \delta_d}. \tag{4.9}$$

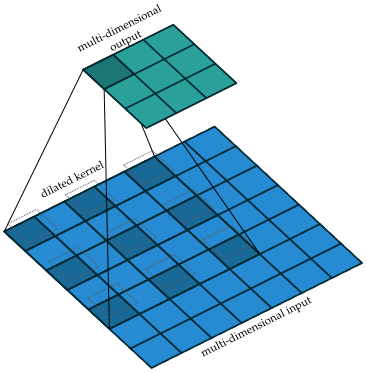

Figure 4.9: Dilated convolutions spread a small kernel over a wider area to capture a larger spatial range without increasing the number of parameters or the number of operations involved. Adapted from (Dumoulin and Visin, 2016).

4.1.5 Dilated convolutions

Convolutions are local operations. To capture patterns at a large scale, it is necessary to increase the size of the kernel, with a cost in terms of operations and parameters to estimate that grows as the side of the kernel to the power d. This quickly becomes prohibitive. Down-sampling the input provides a way to bring together structures that were previously far apart. This reduces, though, the spatial resolution of the feature maps. An alternative consists of dilated convolutions: a small kernel is transformed

into a larger kernel by introducing zeros between the elements. That way, the number of free parameters is kept unchanged while the kernel covers a larger spatial area, see Fig. 4.9.

4.2 The receptive field

When designing a network architecture, one must balance the number of parameters of the model (hence, the difficulty in terms of learning effort and, often closely related, the computational cost of conducting inferences) and the amount of information fed to the network when processing some new data. For fully-connected MLPs, it is clear that the network can possibly combine all input values to produce any of its outputs. This comes at a cost that is prohibitive in terms of parameter count and number of mathematical operations required when applied to multidimensional signals (time series, images, videos, ...). CNNs remove this bottleneck by using convolutions to recycle the weights of a kernel for all locations within the input (translating the kernel provides the next output), and by using local kernels (non-zero weights only within a small spatial support). The *locality* of CNN operations impacts how much the network actually *sees* when computing a given output value. The area in the input tensor that can impact a given output of the network is called the *receptive field*. Computing the size of the receptive field is useful to make sure that it covers the scale of typical patterns that are to be detected or analyzed by the network. A CNN with a receptive field too narrow will never access useful information located outside its receptive field; its performance will, therefore, be limited.

Computing the size of the receptive field requires accounting for the different parameters of the convolutional layers (kernel size, stride, dilation factor, direct or transpose convolution) or non-linear operations such as max-pooling (size of the region over which the maximum is computed). The presence of multiple paths in the network with skip-connections also complicates the analysis. Receptive fields can be computed using automatic differentiation: with randomly initialized weights and zero bias, we can evaluate the gradient of one output value of the network with respect to the input image. Pixels of the input image with non-zero gradient values contribute to the selected output, thus they are part of the receptive field.

In simple cases, it is possible to derive the close-form expression of the receptive field. We give here a brief presentation of the nice derivations of (Araujo et al., 2019). The figure 4.10 illustrates the notations, in a 1D case, for the kernel size K_i, stride factor S_i, and padding value P_i used to perform a conv operation and obtain the pre-activations of layer i (i.e., the activations before applying the non-linear activation function). The extension to multidimensional convolutions is straightforward.

The receptive field can be computed recursively: let R_i denote the spatial extension of the area of hidden vector $\boldsymbol{h}_i$ that impact a given element of the output layer $\boldsymbol{y} \overset{\text{def}}{=} \boldsymbol{h}_H$. On the output layer, $R_H = 1$ (there are no connections between elements of the output layer). Now, let us see how the stride parameter affects the spatial extension R_i between two successive layers by first considering that the kernel size is equal to 1.

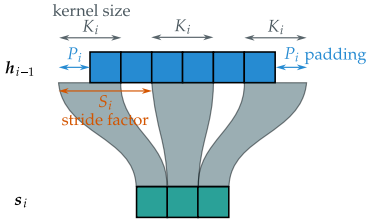

Figure 4.10: From hidden layer $\boldsymbol{h}_{i-1}$ to pre-activations $\boldsymbol{s}_i$, the application of a conv operator connects a number of neurons that depend on several parameters: the kernel size K_i (2, in this example), the stride S_i (3), the padding P_i (1). Adapted from (Araujo et al., 2019).

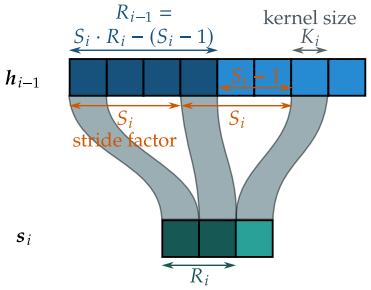

Figure 4.11: R_i consecutive elements of the hidden vector $\boldsymbol{h}_i$ depend on an area of size R_{i-1} in the previous vector. Here, the kernel size is $K_i = 1$, so only the stride factor affects R_{i-1}. Adapted from (Araujo et al., 2019).

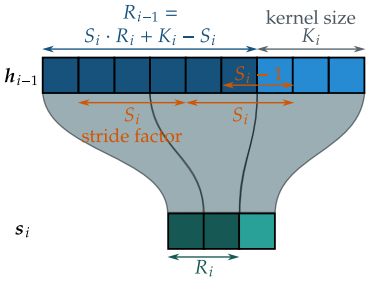

Figure 4.12: A larger kernel size adds up $(K_i - 1)/2$ elements to the right and left of the area. Adapted from (Araujo et al., 2019).

Figure 4.11 shows that an area of spatial extension $R_{i-1} = S_i \cdot R_i - (S_i - 1)$ affects the R_i consecutive values in the i-the layer.

A larger kernel size impacts more elements on the left and right of the area, as illustrated by figure 4.12: the impacted area becomes $R_{i-1} = S_i \cdot R_i + K_i - S_i$. Solving this recurrence equation gives (see proof in (Araujo et al., 2019)):

$$\text{receptive field} \overset{\text{def}}{=} R_0 = \sum_{i=1}^{H} \left((K_i - 1) \prod_{j=1}^{i-1} S_j \right) + 1. \tag{4.10}$$

This formula shows that increasing the depth H of the network and including stride factors larger than 1 are two ways to get a large receptive field. Interestingly, for multidimensional convolutions of dimension d, we have:

$$\text{receptive field in dimension } d = \prod_{k=1}^{d} \left[\sum_{i=1}^{H} \left((K_{i,k} - 1) \prod_{j=1}^{i-1} S_{j,k} \right) + 1 \right], \tag{4.11}$$

so with $\sum_{i=1}^{H} \prod_{k=1}^{d} K_{i,k}$ weights, we can get a receptive field much larger than if a single layer had been used, e.g., for $H = 10$, $K = 3$, $S = 1$, and $D = 2$, we obtain a receptive field formed of 441 pixels using 90 weights, with $S = 2$, the receptive field rises to $4,190,209$ for the same count of weights.

Exercise: dilated convolutions

What is the impact of dilated convolutions on the receptive field?

Note that the computation of the size of the receptive field is useful to check that a given CNN architecture can actually capture enough context to conduct proper inference. Not all pixels of the receptive field have the same weight, though. Usually, the center of the receptive field has a much larger influence than the outer pixels (Luo et al., 2016), but specific architectures have also been used to exclude the central area and create a *blind spot*, a feature that is useful for unsupervised learning, see for example (Laine et al., 2019).

4.3 Connections with linear algebra and signal processing

CNNs drastically reduce the number of free parameters compared to MLPs by restricting the linear operators at each layer to correspond to discrete convolutions, i.e., matrices $\mathbf{W}$ are block-Toeplitz with Toeplitz blocks. To reduce further the learnable parameter count, CNNs also constrain the impulse responses to small supports, e.g., 2D convolutions with 3×3 or 5×5 kernels. As discussed in the previous paragraph, reducing the size of the convolution kernels impacts negatively the size of the receptive field of the network. This can be counterbalanced by increasing the depth of the network and/or using dilated convolutions or larger stride values. Yet, this also impacts the expressivity of the network.

This section discusses connections with linear algebra to better grasp how the composition of several simple linear operators can represent more general linear transforms.

4.3.1 Separable convolutions

Multi-dimensional kernels $\boldsymbol{k}$ that can be decomposed into a product of lower-dimensional kernels $\boldsymbol{m}_1$ to $\boldsymbol{m}_\ell$ are said to be *separable*:

$$[\boldsymbol{k}]_{i_1,i_2,\dots,i_d} = [\boldsymbol{m}_1]_{i_1,i_2,\dots,i_{d_1}} \cdot [\boldsymbol{m}_2]_{i_{d_1+1},i_{d_1+2},\dots,i_{d_1+d_2}} \cdots [\boldsymbol{m}_\ell]_{i_{d-d_\ell+1},i_{d-d_\ell+2},\dots,i_d}. \tag{4.12}$$

Figure 4.13 illustrates the 2D case, where

$$[\boldsymbol{k}]_{i_1,i_2} = [\boldsymbol{m}_1]_{i_1} \cdot [\boldsymbol{m}_2]_{i_2}. \tag{4.13}$$

Separable kernels have fewer free parameters than full kernels. A 2D separable kernel of size $K_1 \times K_2$ only depends on $K_1 + K_2$ parameters. From a computational point of view, this reduction in the number of free parameters also reduces the number of arithmetic operations involved in the computation of a convolution or a correlation:

$$\begin{aligned}
[\boldsymbol{h} \star \boldsymbol{k}]_{i_1,i_2,\dots,i_d} &= \sum_{j_1,\dots,j_d} [\boldsymbol{h}]_{j_1,j_2,\dots,j_d} [\boldsymbol{k}]_{j_1-i_1,j_2-i_2,\dots,j_d-i_d} \\
&= \sum_{j_1,\dots,j_d} [\boldsymbol{h}]_{j_1,j_2,\dots,j_d} [\boldsymbol{m}_1]_{j_1-i_1,j_2-i_2,\dots,j_{d_1}-i_{d_1}} \\
&\qquad \cdots [\boldsymbol{m}_\ell]_{j_{d-d_\ell+1}-i_{d-d_\ell+1},j_{d-d_\ell+2}-i_{d-d_\ell+2},\dots,j_d-i_d} \\
&= [((\boldsymbol{h} \star \boldsymbol{m}_1) \cdots \star \boldsymbol{m}_\ell)]_{i_1,i_2,\dots,i_d}.
\end{aligned} \tag{4.14}$$

This shows that the convolution (or correlation) with a separable multi-dimensional kernel is equivalent to the composition of convolutions (or correlations) with each of the smaller-dimension factors $\boldsymbol{m}$. In 2D, the number of multiplications and additions drops from $\mathcal{O}(D_1 \cdot D_2 \cdot K_1 \cdot K_2)$ to $\mathcal{O}(D_1 \cdot D_2 \cdot (K_1 + K_2))$.

Exercise: separable filters

Show that the boxcar filter (i.e., the square filter) and the Gaussian filter are both separable.

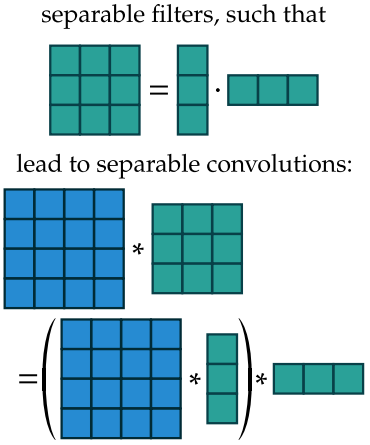

Figure 4.13: When a 2D convolution kernel can be factored into a matrix product of a column vector and a row vector, the convolution can be computed efficiently with two 1D convolutions.

4.3.2 Matrix and tensor decompositions

If the convolution kernel is not separable, can we still use separable convolutions? Is there a middle ground between separable and full convolutions?

Matrix approximation theory provides an answer for 2D filters. Any matrix $\mathbf{K} \in \mathbb{R}^{K_1 \times K_2}$ can be factored under the form $\mathbf{K} = \mathbf{U} \cdot \mathbf{\Sigma} \cdot \mathbf{V}^\top$, where $\mathbf{U} \in \mathbb{R}^{K_1 \times K_1}$ is a real orthonormal matrix, $\mathbf{\Sigma} \in \mathbb{R}_+^{K_1 \times K_2}$ is a rectangular diagonal matrix with non-negative values on the diagonal, in descending order, and $\mathbf{V} \in \mathbb{R}^{K_2 \times K_2}$ is another real orthonormal matrix. This factorization is called the Singular Value Decomposition (SVD). The following

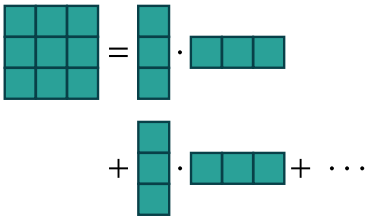

Figure 4.14: The singular value decomposition expresses a matrix as a sum of rank-one terms. Keeping only the first few terms generally provide a good approximation of the original matrix.

theorem provides an optimal way to approximate a matrix by a sum of P rank-one terms:

Theorem 4.3.1 (Eckart–Young theorem, see for example (Martin and Porter, 2012))
The best rank-P approximation of matrix $\mathbf{K}$, according to Frobenius norm, is given by the first P left and right singular vectors, weighted by the corresponding largest singular values:

$$\underbrace{\sum_{p=1}^{P} \boldsymbol{u}_p \cdot \sigma_p \boldsymbol{v}_p^\top}_{\text{truncated SVD}} \in \arg\min_{\mathbf{M}} \|\mathbf{K} - \mathbf{M}\|_F \text{ subject to } \operatorname{rank}(\mathbf{M}) \leq P,$$

with $\boldsymbol{u}_p$ the p-th column of factor $\mathbf{U}$, σ_p the p-th diagonal entry of factor $\mathbf{\Sigma}$ and $\boldsymbol{v}_p$ the p-th column of factor $\mathbf{V}$ of the SVD decomposition of matrix $\mathbf{K}$: $\mathbf{U} \cdot \mathbf{\Sigma} \cdot \mathbf{V}^\top$.

Many convolution kernels can be well approximated by a low-rank matrix. The composition of convolutions with a collection of 1D horizontal filters, followed by convolutions with 1D vertical filters, summed with appropriate weights, can describe a wide variety of filters. Given the moderate cost of 2D kernels, these approximations are mostly reserved to higher-dimension data such as volumetric images in medical imaging, time series, or multi-spectral observations. In these cases, 3D kernels may be avoided and replaced by 2D convolutions followed by 1D convolutions along the third dimension. The process can be formalized as a tensor decomposition of the convolution kernel, or by "flattening" the 3D kernel to allow a representation as a matrix (the first two dimensions K_1 and K_2 are represented as a single dimension with $K_1 \cdot K_2$ elements).

4.3.3 Sparse linear operators

Convolutions with small kernels define a family of sparse linear operators, i.e., linear operators with very few non-zeros in their matrix representation. This family of operators leads to efficient implementations since operations are local, which generally leads to faster memory access. Other sparse structures can be considered, yet, discovering the optimal pattern of non-zeros, or even finding the optimal coefficients for a fixed sparsity structure is NP-hard; special structures like extensions of the butterfly sparsity of the Fast Fourier Transform (FFT) are particularly promising, see the survey paper (Gribonval et al., 2025).

Deep Learning in Computer Vision

5

Computer Vision (CV) is the field of Computer Science devoted to the automatic analysis and understanding of visual information: it seeks algorithms that can extract meaningful information from images and videos. An emblematic task in Computer Vision is *image classification*: recognizing the category of an image (e.g., an image of a cat, of a boat). Several *low-level* cues are useful to draw the conclusion of the correct class: the shape or orientation of the edges in the image, the colors, the textures. Yet, connecting these descriptors of the local image content to higher-level concepts such as "wheel", "face", "tree", and assembling these concepts such as "wheel + frame + handlebar = bike" is very difficult, it is referred to as *bridging the semantic gap*.

To foster research in the field of image classification, an international challenge called the ImageNet Large Scale Visual Recognition Challenge (ILSVRC)* was set up in 2010. It provided 1.2 million images (see example images in Figure 5.1), each labelled by a class among 1000 different categories†. The best performing algorithm in 2010 guessed the correct class among its top-5 predictions 72% of the time (28% top-5 classification error, which is more than 5 times worse than a human expert). In 2012, the classification error dropped to 16% using a convolutional neural network, which was a significant breakthrough. Deep neural networks have dominated the competition the following 5 years, reaching an error below 3%, which is better the human error‡ of ≈ 5%. By then, the problem was considered solved, and the challenge was discontinued.

Since 2012, deep neural networks have become the go-to algorithm for many other computer vision tasks. In this chapter, we will focus on 3 tasks: image classification, object detection, and semantic segmentation.

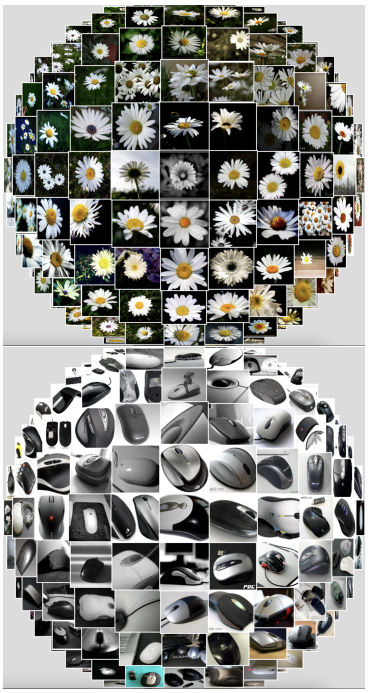

Figure 5.1: Example images from the ImageNet collection, visualized with https://navigu.net, for classes "daisy" and "computer mouse".

5.1 Image classification

The first artificial neural network trained for image classification was developed at AT&T Bell Laboratories by a group led by Yann LeCun, in 1989. The classification task they tackled was handwritten digit recognition for automatic zip code identification at the U.S. Postal Service, see example images in figure 5.3. The network, referred to as LeNet in the literature, contained two convolutional layers followed by two fully-connected layers.

After an initial increase of the dimension, the number of hidden neurons in LeNet is reduced layer after layer to reach 10 output units in the

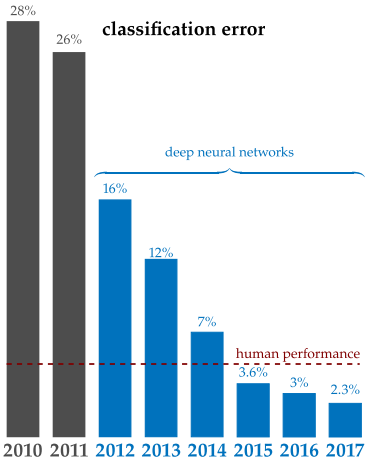

Figure 5.2: Evolution of the top-5 classification error reached by the best team at the annual ImageNet Large Scale Visual Recognition Challenge, source: (Fei-Fei and Deng, 2017).

* https://www.image-net.org/

† Assembling such a massive annotated dataset was a huge endeavour. 160 million images were collected from the web by a group led by Fei-Fei Li at Princeton, based on queries using the class name. Keeping relevant images that matched the labels took almost two years and involved 49,000 workers from 167 different countries, using the Amazon Mechanical Turk to hire workers for on-demand tasks.

‡ as evaluated by Andrej Karpathy, an AI researcher working then at Standford, see his blog https://karpathy.github.io/2014/09/02/what-i-learned-from-competing-against-a-convnet-on-imagenet/

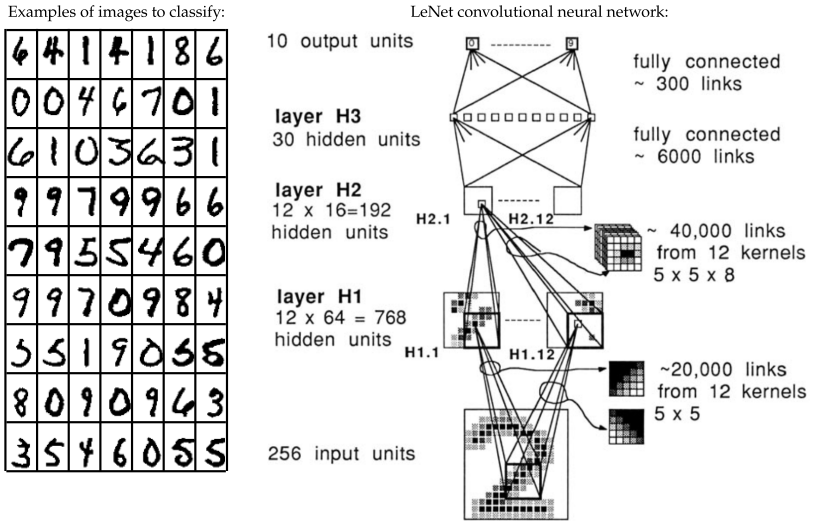

Figure 5.3: The first artificial neural network trained for image classification: Yann LeCun's LeNet, in 1989. The first two layers apply convolutions with 5×5 kernels, two fully-connected layers produce the final classification from the feature maps produced by the convolutional layers. Adapted from (LeCun et al., 1989).

final layer. The convolutional layers extract spatial features from the previous layers, at the scale of the kernel dimension (5×5 pixels). Based on these feature maps, the last two layers perform a classification and output the class (i.e., the digit value) using one-hot encoding. This general architecture composed of a *feature extractor* followed by a *classifier*, learned jointly from the data, proved very successful when deep learning was revived in 2012, thanks to the combination of (i) the availability of large training datasets of images, and (ii) massive computational capability improvements with the use of graphical processing units (GPUs). Here are some milestones in image classification using deep learning:

- AxelNet, introduced in (Krizhevsky et al., 2012) by the team of Geoffrey Hinton, was the winning entry of the 2012 edition of the ILSVRC competition. It followed the same general architecture as LeNet, but deeper (7 hidden layers rather than 3), and with many more channels at each convolutional layer, see figure 5.4. It was trained on the massively larger dataset of ImageNet (million of 224×224 images rather than a few thousand 16×16 images), using two GPUs (the 1989 LeNet was trained during 3 days on a personal computer equipped with a digital signal processor (DSP), AxelNet was trained during 5 to 6 days). A major innovation comes from replacing the hyperbolic tangent by a non-saturating activation function: the Rectified Linear Units (ReLU) introduced two years before (Nair and Hinton, 2010). To reduce the *overfitting* problem, two strategies were used: (i) *data augmentation*, which consists of artificially augmenting the training set by cropping a 224×224 pixels area within a 256×256 pixel image, with a random shift, possibly applying a vertical reflection, and color manipulations; (ii) *dropout*, which randomly sets to zero some hidden vector entries to force the network to produce more robust features and reduce the co-dependence between neurons (dropout was applied to 50% of the neurons of the first two fully-connected layers).
- the VGG network, introduced in (Simonyan and Zisserman, 2015) and named after the Visual Geometry Group of the University of Oxford, showed that the classification performance could be improved by using smaller convolution kernels and deeper architectures (16 to 19 layers), see figure 5.5. Replacing AlexNet's 7×7

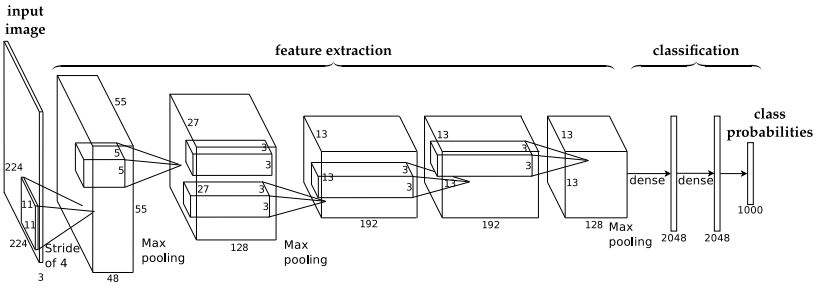

Figure 5.4: AlexNet, 2012's winning entry at the image classification competition ILSVRC, uses a deeper architecture than LeNet and several innovations. Adapted from (Krizhevsky et al., 2012).

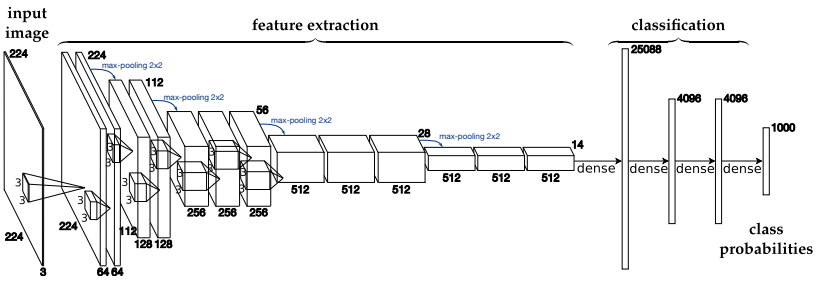

Figure 5.5: VGG-16 is a deeper architecture with small convolution kernels.

convolution kernels by a stack of 3 convolution layers with 3×3 kernels reduces the number of parameters per filter from 49 to 27 (−45%) while allowing learning of more complex functions thanks to the use of 3 activation functions (see section 4.3).

- GoogLeNet (Szegedy et al., 2015), proposed by a team at Google and American Universities, suggested in parallel to the VGG group to train deeper neural networks. Their architecture contains traditional convolutional layers, followed by novel blocks, called *Inception modules*, that run in parallel convolutions with kernels of various sizes (from 1×1 to 5×5), see figure 5.6. Compared to VGG, their more elaborate architecture is more efficient in terms of computational complexity. Note that the final classification performance was slightly improved using computationally expansive strategies: (i) 7 models were trained from scratch and applied to each image, their classification results were then combined; (ii) images were rescaled and cropped in various ways to produce 144 versions, each processed by the classifier. Averaging the softmax probabilities of the different models and various crops/rescales reduced the top-5 error from 10% to 6.7%.

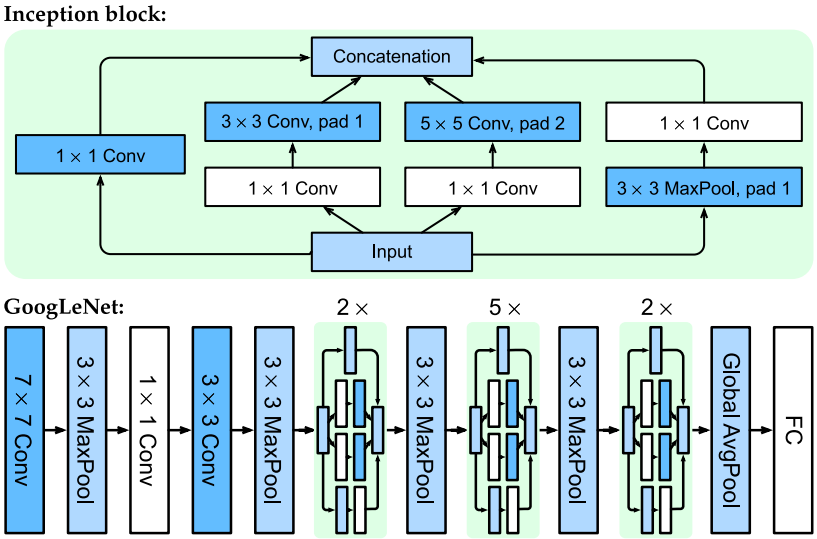

Figure 5.6: GoogLeNet is based on the Inception block that performs dimensionality reduction and small convolutions in parallel. Adapted from the book "Dive into Deep Learning" (Zhang et al., 2023).

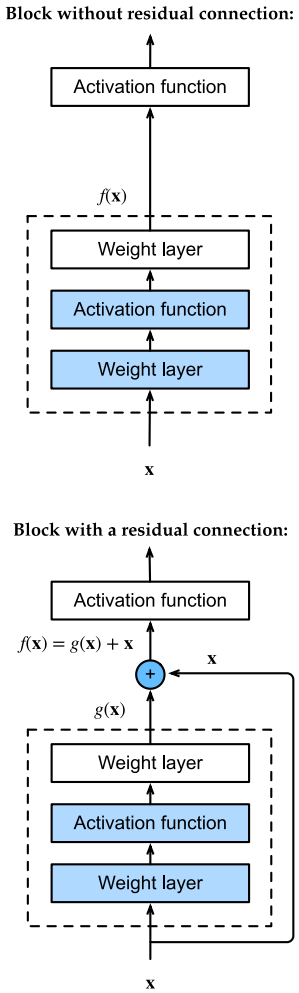

Figure 5.7: Standard block vs residual block, source: (Zhang et al., 2023).

- ResNet (He et al., 2016), addressed a major limitation of previous neural networks: when increasing the depth of the network, the performance decreased significantly when reaching ≈ 20 layers. This is because deeper networks are harder to train (i.e., the issue does not come from having too many parameters and suffering from an over-fitting problem, but rather to fail to fit well the model even on the training set). To ease training, ResNet uses residual blocks, illustrated on figure 5.7, that introduce shortcuts in the network so that the first layers have a more direct influence on the loss function (i.e., not only through the transformations induced by all the subsequent layers). Rather than composing functions when stacking layers, the residual formulation models the function $f(\boldsymbol{x})$ performed by a block as the sum of the actions of the layers in the block $g(\boldsymbol{x})$ and $\boldsymbol{x}$. Note that, when increasing the depth of a network, setting $g(\boldsymbol{x}) = 0$ in the additional residual blocks reverts to the original less-deep network, i.e., the representation power strictly increases. See also our discussion on skip connections on page 28.
  Figure 5.8 shows the architecture of one of the ResNet models used for image classification. The 3×3 convolutions of the VGG model are replaced with residual blocks. When the number of channels changes between the input of the residual block and the output of the layers implementing function $g(\boldsymbol{x})$, a 1×1 convolution is applied to the residual branch to adapt the dimensions so that $g(\boldsymbol{x})$ and $\boldsymbol{x}$ can be added together (see the residual block shown in red in figure 5.8). ResNet won the ILSVRC 2015 classification competition with a 512 layers network, an unprecedented depth for an artificial neural network.
- ResNeXt (Xie et al., 2017) combines the idea of residual blocks at the core of ResNet with the parallel convolutions of the Inception block. To limit the number of parameters and the computational complexity, the channels are split into groups and each parallel branch operates on a subset of channels, using so-called *group-convolutions*. Figure 5.9 illustrates the architecture of a block.

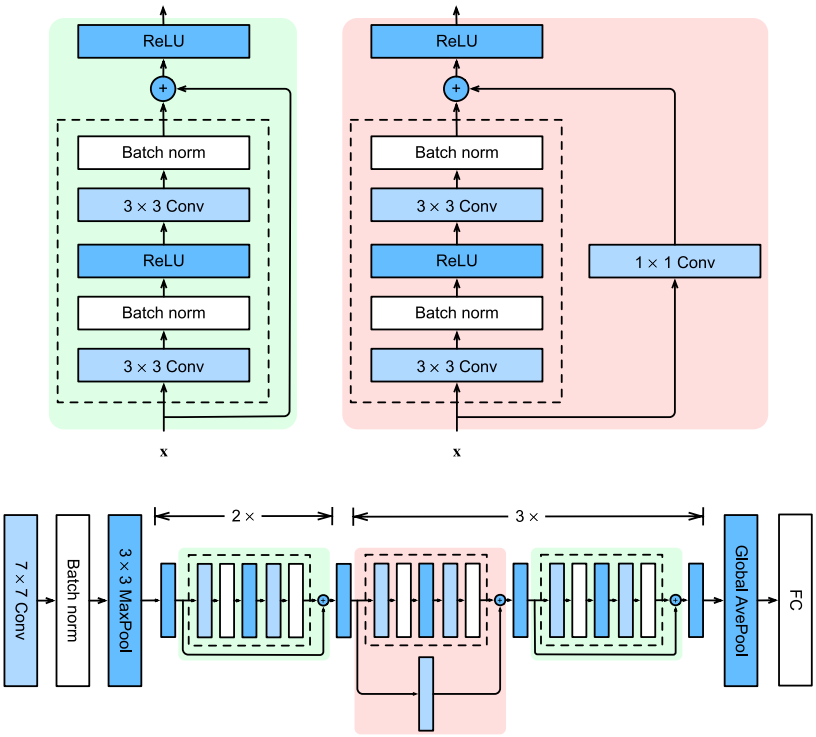

Figure 5.8: The ResNet is a very deep architecture with residual blocks. Here, the ResNet-18 architecture is shown. Deeper architectures are obtained by stacking more residual blocks. Adapted from (Zhang et al., 2023).

- More recent architectures generally include attention mechanisms, in particular in the form of transformers (Chen et al., 2021; Hatamizadeh and Kautz, 2025), which will be covered in more advanced courses.

Our enumeration of various architectures for image classification illustrates that many design choices are possible. Systematically investigating architectures to select the best performing ones, i.e., performing a meta-optimization (maximizing the performance of models, each trained to maximize classification performance) is a problem called Network Architecture Search (NAS). Such exploration can be extremely costly, several strategies can be considered to explore architectures within a fixed computational budget, see (Poyser and Breckon, 2024).

Even the training of a single architecture on large datasets of millions or billions of images is costly. Hopefully, the feature extraction part of classification networks trained on such datasets is very versatile. Fine-tuning a pre-trained network, an approach called *transfer learning*, generally leads to good performance on images from a specialized field (e.g., classifying defects for automated visual inspection systems in quality control).

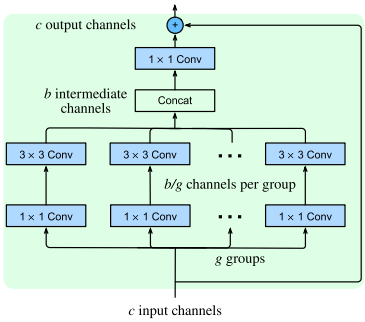

Figure 5.9: The ResNeXt bloc. Adapted from (Zhang et al., 2023).

5.2 Object detection

(The presentation of this paragraph is mostly based on the review paper "Object Detection in 20 Years: A Survey" (Zou et al., 2023))

Object detection is a task that consists of detecting each instance of visual objects of a given class, e.g., birds in Figure 5.10. It answers the questions *What objects?* and *Where?*. The aim of computer vision techniques for object

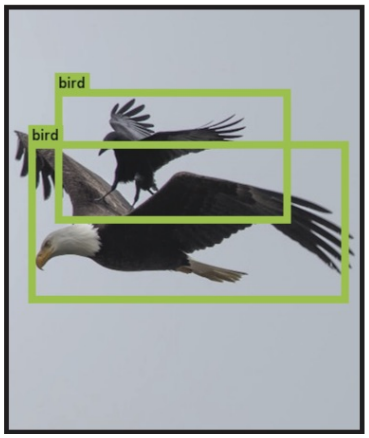

Figure 5.10: Object detection requires *localizing* objects (here, by giving a rectangular bounding box), and *classifying* each object, source: (Redmon et al., 2016).

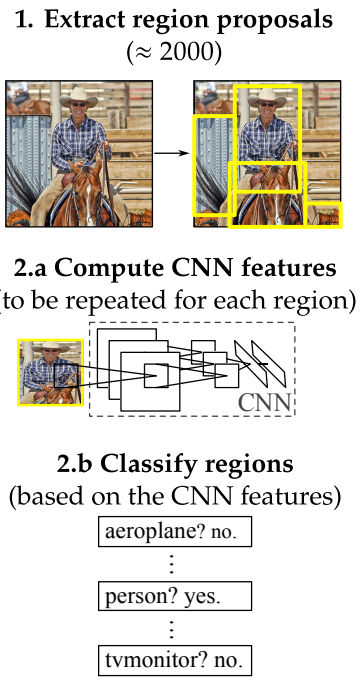

Figure 5.11: Two-stage object detection with RCNN: candidate regions are first selected; features are then computed for each region and used to get the final classification; adapted from: (Girshick et al., 2014).

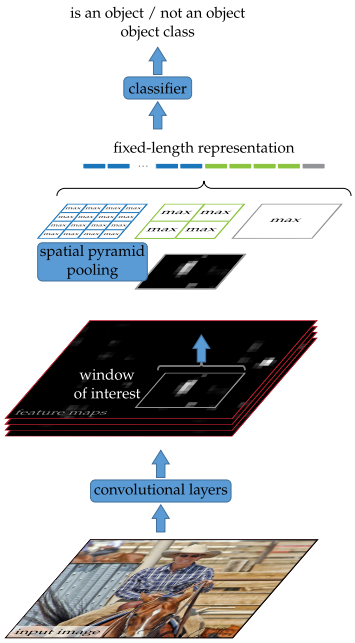

Figure 5.12: Two-stage object detection with SPPNet: spatial features are first computed by a CNN, features inside each window of interest are then converted into a fixed-length representation by spatial pyramid pooling. This representation is then classified. Adapted from (He et al., 2015b)

detection is to reach a *high accuracy* (the object class is correctly recognized, the object localization matches precisely the actual location of the object) and *speed*, to allow for real-time applications. Object detection is widely used in autonomous driving, robot vision and video surveillance. It also serves as a basis for tasks such as instance segmentation (see section 5.3), image captioning, or object tracking.

Object detection is a challenging task because objects can be seen under different viewpoints and illuminations, they may present strong intra-class variations (for example, "chairs" exist in all kinds of shape, material, and color), objects can appear rotated or at different scales (e.g., a car seen in closeup or from a distance). The density of objects (tens or hundreds of objects in an image) is also an added difficulty, as is the occlusion between objects.

Before the advent of deep learning in computer vision, object detection techniques were based on sophisticated handcrafted features. These works introduced several ideas still used by modern deep learning approaches, such as the computation of features on sliding windows, detection cascades that discard less interesting areas and concentrate processing effort on selected regions, multi-scale processing to achieve scale-invariant detection, mixture models to combine classification clues from different regions to make a decision. Handcrafted features varied from a variant of Haar-like features (difference between the average pixel values computed in two neighboring rectangular areas) in the Viola-Jones object detector (Viola and Jones, 2001), to histograms of oriented gradients (HOG) in (Dalal and Triggs, 2005; Felzenszwalb et al., 2009).

Deep-learning based object detectors can be classified in two groups: "two-stage detectors" and "one-stage detectors":

two-stage detectors first locate areas of interest (coarse detection step), then process each region (refinement step):

- RCNN (Region with CNN features) introduced in (Girshick et al., 2014) first extracts candidate regions (yellow rectangles in Figure 5.11). Each region is then rescaled to a fixed-size and fed into a pretrained CNN to extract features relevant for image classification (e.g., AlexNet, see section 5.1). These features are used to detect the presence of an object and its class. The role of the region proposal step is to reduce the number of candidate regions to a few thousands (rather than regions at all locations and scales). Still, the CNN features need to be recomputed many times, which severely impacts the processing time if this approach.
- Spatial pyramid pooling networks (SPPNet) (He et al., 2015b) reduce the computational load by computing the CNN features only once, and extracting a fixed-length descriptor for each window of interest via a process called spatial pyramid pooling. In contrast with the max-pooling operation described in section 4.1.4, spatial pyramid pooling divides the input window into grids of various resolution and computes the maximum within each cell of the grid. The number of elements over which the maximum is computed therefore depends on the size of the feature maps in input, while the number of output elements remains fixed, see figure 5.12. The

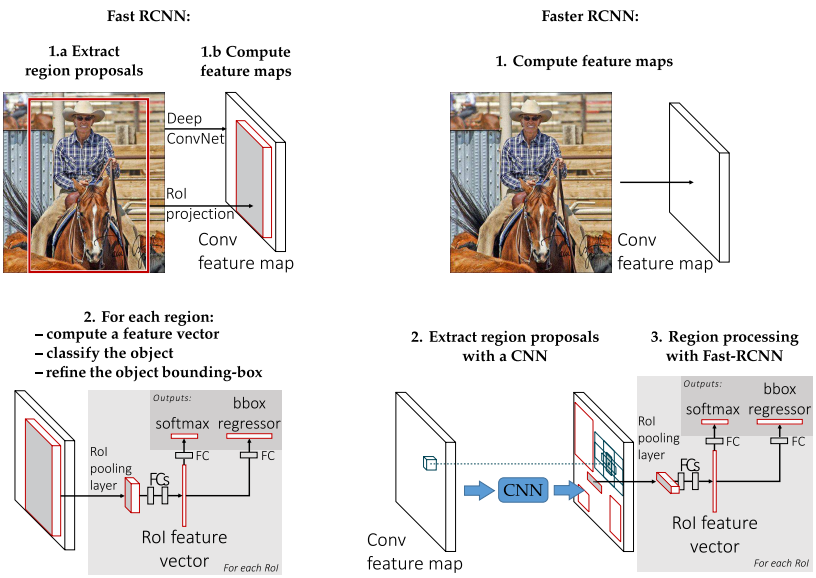

Figure 5.13: Two-stage object detection with FastRCNN and FasterRCNN: Feature maps are computed only once. Proposal windows are either produced by an other algorithm (Fast RCNN) or predicted by a small CNN applied directly to the feature maps. Fixed-length descriptors are then computed for each window by a pooling mechanism (using a single scale) and fed to two networks, one predicting the presence of an object and its class, the other estimating the object's bounding box. Adapted from (Girshick, 2015).

presence/absence of an object in the window and the class of the object are determined by applying a classifier (an SVM) to the representation produced by the spatial pyramid pooling step. Compared to RCNN, SPPNet offers a 20× speedup without sacrificing the detection accuracy.

- Fast RCNN, introduced in (Girshick, 2015), refines the SPPNet approach. It computes the feature maps only once, like SPPNet. Features inside each proposal window are converted into a fixed-length representation using pooling with a fixed-size grid (i.e., single-scale pyramid pooling) and two MLPs are used to (i) classify the region, and (ii) predict a refined bounding-box for the object. While the SPPNet was trained in several steps, Fast RCNN can be trained end-to-end using a loss that combines the classification performance and the regression precision (bounding-box prediction task).
- Faster RCNN (Ren et al., 2015), illustrated on figure 5.13, uses a small CNN to predict quickly windows containing objects. This region proposal network is applied to the feature maps of the image. Once regions have been generated, Fast RCNN is used to process each region.
- Feature Pyramid Networks (Lin et al., 2017a) improve the accuracy by processing not only feature maps at a given level, but a pyramid formed by features at different depths of a CNN pretrained for image classification. By up-sampling the higher-semantic levels at the deeper layers of the network and combining them with lower-semantic layers with high spatial resolution, the feature pyramids provide more discriminant information to localize and classify objects.

one-stage detectors attempt to perform all the detections at once, to speed-up the inference, at the price of a degradation of the accuracy for dense and small objects:

- YOLO (You Only Look Once), introduced in (Redmon et al., 2016) is the first single-step deep object detector. First, it computes feature maps from an input image and predicts, based on the features, a fixed number of bounding boxes,

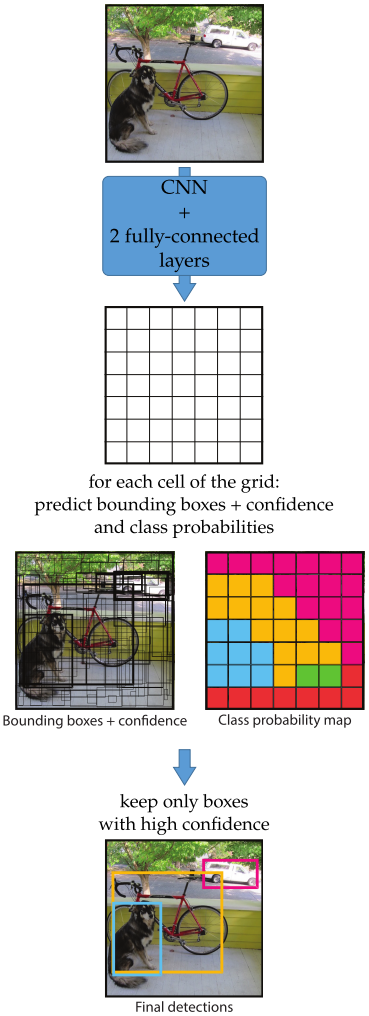

Figure 5.14: YOLO uses a CNN with pooling operations to produce a 7×7 feature map. These features are combined in 2 fully-connected layer to predict the location of bounding boxes related to each cell of the 7×7 grid, estimate the confidence on the presence of an object in the box, and detect the object class.

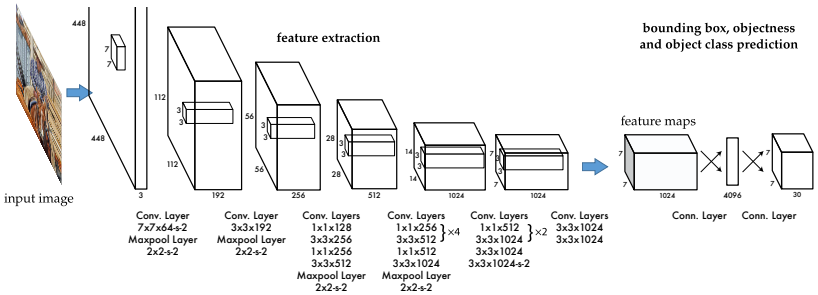

Figure 5.15: Architecture of the YOLO network. Adapted from (Redmon et al., 2016).

spatially organized in a 7×7 grid. Each bounding box is defined by its horizontal and vertical directions, the offset with respect to the cell center, the network confidence on the presence of an object, and the predicted probability that the object corresponds to each of the classes. The name of the method stems from the fact that, unlike two-stage detectors, bounding boxes are not independently processed to refine an initial guess. The architecture of the network is shown on figure 5.15: it can be decomposed into two parts: a first convolutional part that extracts feature maps, and a second part that predicts the 7×7 grid of bounding boxes. Thanks to its single-stage processing, YOLO is very fast: it can be applied in real-time. There have been several other versions since the original paper (Redmon et al., 2016), to improve the accuracy and/or the processing time, see for example `https://docs.ultralytics.com/fr/models/yolo11/`.

- The Single-Shot Multibox Detector (SSD), introduced in (Liu et al., 2016), uses feature maps at different resolutions and predict a much larger number of bounding box than YOLO. It achieves a better accuracy than the first version of YOLO.
- RetinaNet (Lin et al., 2017b) introduced a new loss to compensate an issue with previous object detectors based on dense detections: the strong class imbalance between the few foreground objects and numerous boxes that correspond to the background. To focus more heavily on mis-classified examples, the binary cross-entropy defined in equation (2.11) is modified by adding a weighting factor $(1-p)^\gamma$, where γ is a positive parameter ($\gamma = 0$ reverts to the usual binary cross-entropy, $\gamma = 2$ was found to work well in practice) and p is either the prediction $f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)$ of the network, for a positive example ($y_i^\star = 1$), or $1 - f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)$ (when $y_i^\star = 0$). The effect of this weighting is illustrated on the top of figure 5.16. Combined with the binary cross-entropy term, it gives a flat loss function for well-classified examples ($p \gtrapprox 0.5$), with a large contrast for wrongly-classified examples ($p \lessapprox 0.1$).
- CornerNet (Law and Deng, 2018) introduces a more flexible way to define bounding boxes: it uses a CNN to produce two heatmaps with local maxima located at the top-left or bottom-right corners of the objects bounding boxes. Corners are then paired by comparing the embeddings computed by the network: most similar embeddings correspond to corners belonging to the same bounding box.
- CenterNet (Zhou et al., 2019) replaces corners by a central

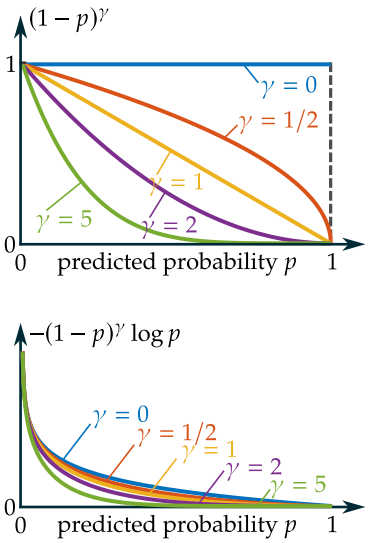

Figure 5.16: The focal loss introduced in (Lin et al., 2017b) re-weights the binary cross-entropy to give more focus to non-detections.

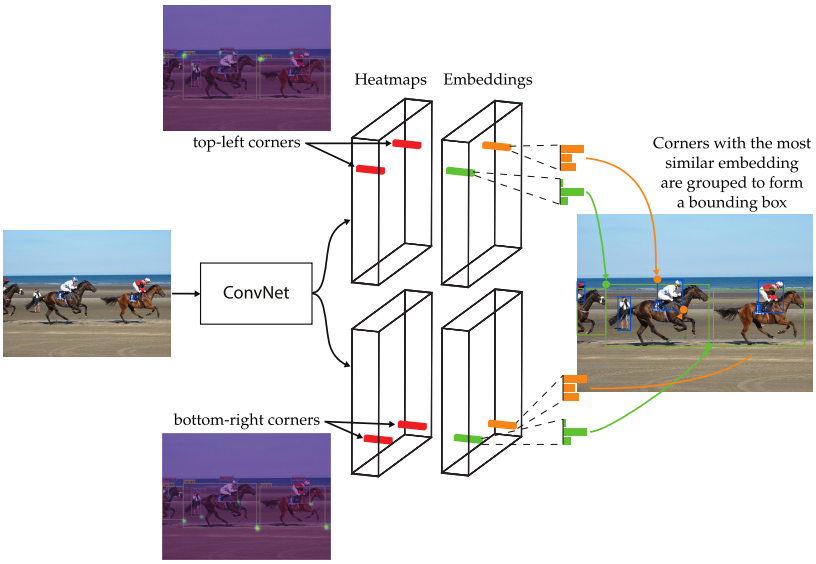

Figure 5.17: CornerNet predicts heatmaps corresponding to the top-left and bottom-right corners + embeddings used to group corners to form bounding-boxes. Adapted from (Law and Deng, 2018).

point in the object, leading to a simple approach to predict several quantities of interest for each object (bounding box, object class, and possibly, motion, depth, pose, 3D size and orientation, see figure 5.18).

Object detection is still an active research domain, with novel architectures developed to improve the detection accuracy or the computation time for real-time applications. Methods are often trained and compared on the Common Objects in Context (COCO) data set (Lin et al., 2014) which contains over 200,000 labeled images with over 80 category labels. Images include complex, everyday scenes with common objects in their natural context. You may find a ranking of several methods on HugginFace's website: `https://huggingface.co/spaces/hf-vision/object_detection_leaderboard`.

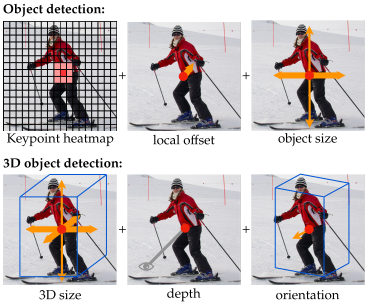

Figure 5.18: CenterNet produces a keypoint heatmap and, for 2D object detection, a local offset and the object size. For 3D objects, the 3D size, depth with respect to the camera, and object orientation are estimated. Adapted from (Zhou et al., 2019).

5.3 Semantic segmentation

Image classification associates a class to a whole image. Object detection defines bounding boxes and object classes to several objects in an image. Classification at pixel level is called *semantic segmentation*. It labels each pixel of the input image with a class.

There are two related tasks in computer vision:

- *image segmentation*, which aims at separating the image into its constituent regions (without prior knowledge of existing classes). Depending on the application, the boundary between regions of interest may vary. Some segmentation techniques are automatic and use solely edge, color, and texture information; others are interactive, the user provides partial labels, typically by selecting a few pixels inside the region and some pixels outside or by drawing a bounding box that encloses the object to segment, see figure 5.20.
- *instance segmentation* not only has to identify the class of each pixel, but also to separate different objects of the same class. In the example shown in figure 5.21, elements of the class "sheep" are

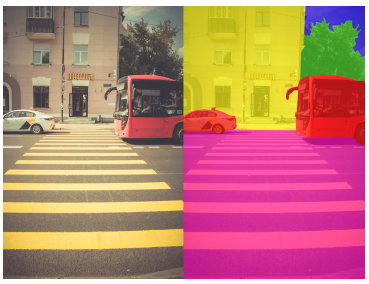

Figure 5.19: Semantic segmentation labels each pixel of the image into a different semantic class, represented on the right image with various colors: vehicle (red), road (violet), building (yellow), sky (blue), vegetation (green). Image by B. Palac, Wikimedia Commons.

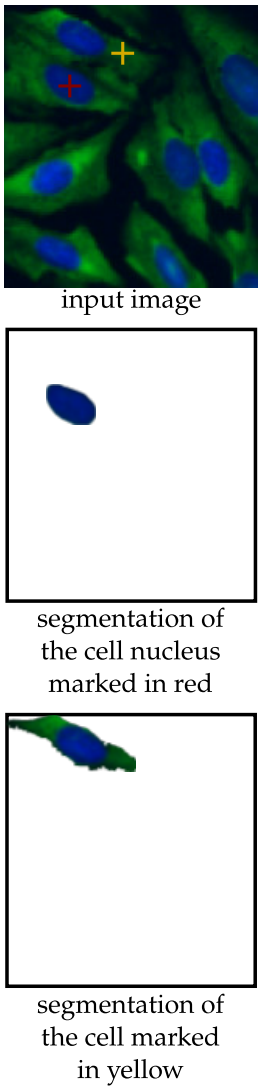

Figure 5.20: Segmentation of a fluorescent microscopy image of cells.

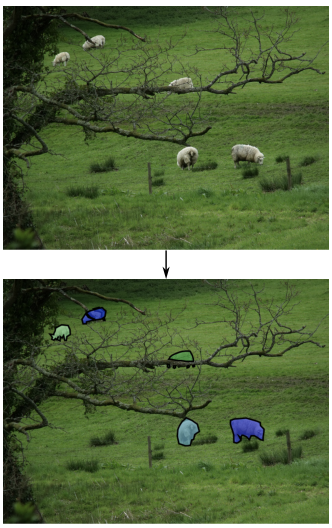

Figure 5.21: Instance segmentation: image pixels for each sheep are given a different label. Source: COCO dataset.

shown. Each instance, i.e., each sheep, gets assigned a different label.

The U-Net, introduced in (Ronneberger et al., 2015), is a fully-convolutional architecture for semantic segmentation. While image classification networks progressively reduce the spatial resolution of feature maps until reaching the final fully-connected classification layers, for semantic segmentation the spatial resolution of the output should match that of the input image. Yet, progressively reducing the spatial resolution when going deeper within the network is helpful to extract higher-level concepts. In order to combine these concepts with the precise localization of the structures in the image, the U-Net uses two branches (forming a U shape), see figure 5.23:

- one branch that reduces the spatial dimension and produces a semantic representation in a *latent space*[§];
- a second branch that increases the spatial resolution of low-resolution latent representations and combines them with higher-resolution representations.

The general architecture corresponds to an auto-encoder, see figure 5.22: the input image is encoded into a latent code by the encoder (the descending branch of the U shape), then it is decoded to produce an output with the same spatial resolution as the original image (the ascending branch of the U shape). Residual connections, represented in orange in figures 5.22 and 5.23, prevent the loss of details.

The U-Net offers several attractive properties. Since it is fully-convolutional, it can be applied to larger images at inference (provided that the image size be a multiple of some power of 2 so that the down-sampling/up-sampling operations yield a result with the same dimension as the original image). The U-Net runs quite quickly compared to methods requiring fixed-dimension inputs and a tiling strategy. It also requires a modest number of training images compared to the models used for image classification (a few tens of partially labeled images are used in (Ronneberger et al., 2015)). There exist many variations of the U-Net, including extensions to 3D images and several improvements. The review paper (Azad et al., 2024) discusses over 100 segmentation algorithms derived from the U-Net for medical image segmentation that were published between 2015 and 2022. Analyzing these techniques goes beyond the scope of this introduction.

[§] hidden representations, i.e., the value of hidden vectors at some given layer of the network, form a different representation of the input data; since the hidden representations must be discovered through learning, they are called *latent representation*, or representations in a *latent space*.

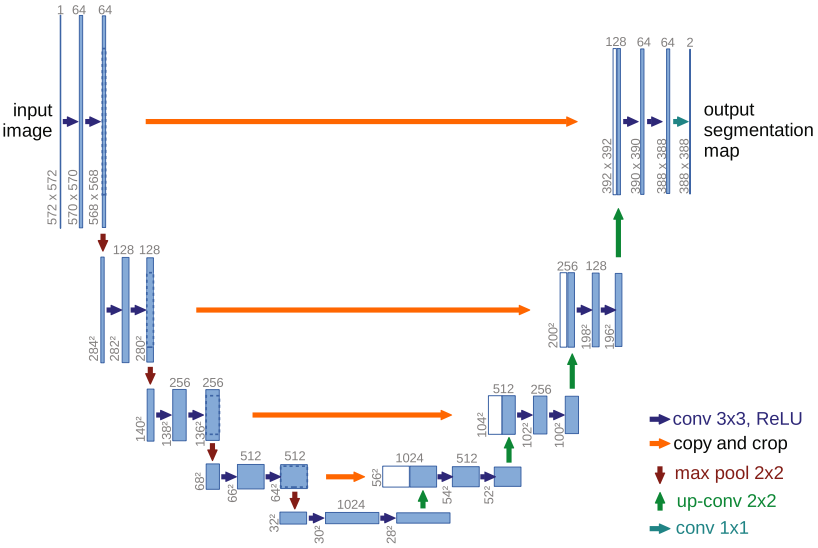

Figure 5.23: Detailed architecture of the U-Net. Adapted from (Ronneberger et al., 2015).

5.4 Concluding words

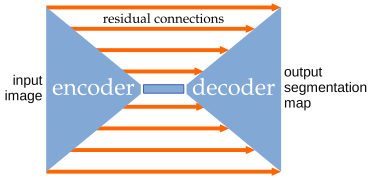

Figure 5.22: The U-Net uses an auto-encoder architecture with residual connections. See the detailed architecture in figure 5.23

There are many other tasks in computer vision that can be solved using deep neural networks. These tasks vary depending on the nature of the input (2D image, volumic image, video, color or multi-spectral image, 3D point cloud, RGB + depth images, text) and the output (an image, some text, a list of object properties, a 3D model, a pixel-based segmentation). Figure 5.24 illustrates some computer vision tasks that were not covered in this short introduction and that will be the subject of more advanced courses.

A recent trend is the training of models on massive datasets. These models, applicable to a wide variety of contexts, are called *foundation models*. To emphasize their versatility, these models are sometimes given generic names, such as "Segment Anything Model" (SAM) (Kirillov et al., 2023; Ravi et al., 2024) or "Depth Anything" (Yang et al., 2024a; Yang et al., 2024b).

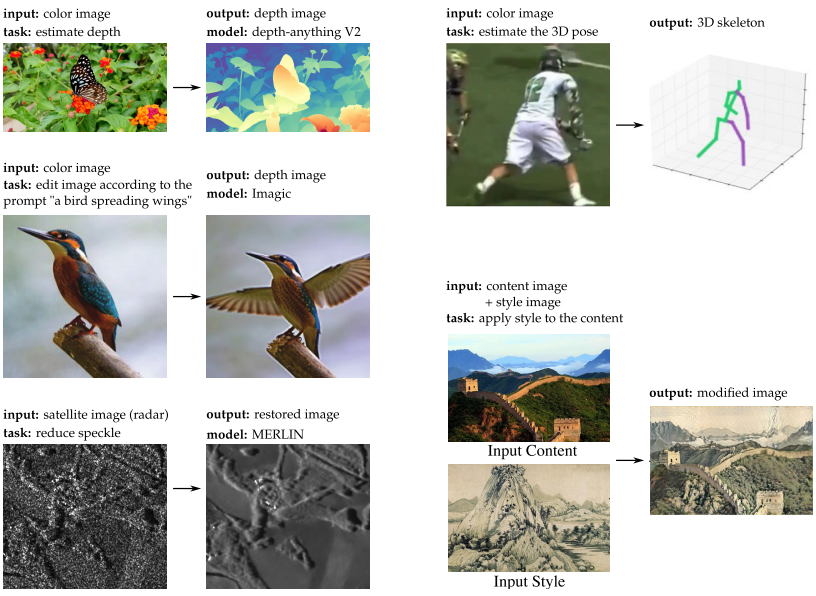

Figure 5.24: Some illustrations of computer vision tasks not covered in this introduction.

Bibliography

Apicella, A., F. Donnarumma, F. Isgrò, and R. Prevete (2021). A survey on modern trainable activation functions. *Neural Networks* 138, pp. 14–32.

Araujo, A., W. D. Norris, and J. Sim (2019). Computing Receptive Fields of Convolutional Neural Networks. *Distill*.

Azad, R., E. K. Aghdam, A. Rauland, Y. Jia, A. H. Avval, A. Bozorgpour, S. Karimijafarbigloo, J. P. Cohen, E. Adeli, and D. Merhof (2024). Medical image segmentation review: The success of u-net. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Bach, F. (2024). *Learning theory from first principles*. MIT press.

Baydin, A. G., B. A. Pearlmutter, A. A. Radul, and J. M. Siskind (2018). Automatic differentiation in machine learning: a survey. *Journal of machine learning research* 18.153, pp. 1–43.

Bottou, L. (1998). Online learning and stochastic approximations. *Online learning in neural networks* 17.9, p. 142.

Carlucci, F. M., A. D'Innocente, S. Bucci, B. Caputo, and T. Tommasi (2019). Domain generalization by solving jigsaw puzzles. *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 2229–2238.

Chen, C.-F. R., Q. Fan, and R. Panda (2021). Crossvit: Cross-attention multi-scale vision transformer for image classification. *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 357–366.

Dalal, N. and B. Triggs (2005). Histograms of oriented gradients for human detection. *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*. Vol. 1. Ieee, pp. 886–893.

Devlin, J., M.-W. Chang, K. Lee, and K. Toutanova (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pp. 4171–4186.

Dubey, S. R., S. K. Singh, and B. B. Chaudhuri (2022). Activation functions in deep learning: A comprehensive survey and benchmark. *Neurocomputing* 503, pp. 92–108.

Dumoulin, V. and F. Visin (2016). A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*.

Elgendy, M. (2020). *Deep learning for vision systems*. Simon and Schuster.

Fei-Fei, L. and J. Deng (2017). Where have we been? Where are we going? *Beyond ImageNet Large Scale Visual Recognition Challenge, CVPR Workshop*. URL: https://image-net.org/static_files/files/imagenet_ilsvrc2017_v1.0.pdf.

Felzenszwalb, P. F., R. B. Girshick, D. McAllester, and D. Ramanan (2009). Object detection with discriminatively trained part-based models. *IEEE transactions on pattern analysis and machine intelligence* 32.9, pp. 1627–1645.

Girshick, R. (2015). Fast r-cnn. *Proceedings of the IEEE international conference on computer vision*, pp. 1440–1448.

Girshick, R., J. Donahue, T. Darrell, and J. Malik (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 580–587.

Glorot, X. and Y. Bengio (2010). Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, pp. 249–256.

Goodfellow, I., Y. Bengio, A. Courville, and Y. Bengio (2016). *Deep learning*. Vol. 1. 2. MIT press Cambridge.

Gribonval, R., E. Riccietti, Q.-T. Le, and L. Zheng (2025). Rapture of the deep: highs and lows of sparsity in a world of depths.

Hatamizadeh, A. and J. Kautz (2025). Mambavision: A hybrid mamba-transformer vision backbone. *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 25261–25270.

He, K., X. Chen, S. Xie, Y. Li, P. Dollár, and R. Girshick (2022). Masked Autoencoders Are Scalable Vision Learners. *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 15979–15988.

He, K., X. Zhang, S. Ren, and J. Sun (2015a). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034.

He, K., X. Zhang, S. Ren, and J. Sun (2015b). Spatial pyramid pooling in deep convolutional networks for visual recognition. *IEEE transactions on pattern analysis and machine intelligence* 37.9, pp. 1904–1916.

He, K., X. Zhang, S. Ren, and J. Sun (2016). Deep residual learning for image recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778.

Howson, K., J. Valente, H. Johnston, P. A. Reneses, F. U. Spilda, D. Arubayi, P. Feuerstein, M. Cole, S. Katta, T. Lopez, et al. (2022). Fairwork Cloudwork Ratings 2025: Work in the Planetary Labour Market.

Ioffe, S. and C. Szegedy (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *International conference on machine learning*. pmlr, pp. 448–456.

Kalra, D. S. and M. Barkeshli (2024). Why warmup the learning rate? underlying mechanisms and improvements. *Advances in Neural Information Processing Systems* 37, pp. 111760–111801.

Kingma, D. P. and J. Ba (2014). Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations (ICLR)*.

Kirillov, A., E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, A. C. Berg, W.-Y. Lo, et al. (2023). Segment anything. *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 4015–4026.

Krizhevsky, A., I. Sutskever, and G. E. Hinton (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25.

Laine, S., T. Karras, J. Lehtinen, and T. Aila (2019). High-quality self-supervised deep image denoising. *Advances in neural information processing systems* 32.

Law, H. and J. Deng (2018). Cornernet: Detecting objects as paired keypoints. *Proceedings of the European conference on computer vision (ECCV)*, pp. 734–750.

LeCun, Y., Y. Bengio, and G. Hinton (2015). Deep learning. *nature* 521.7553, pp. 436–444.

LeCun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation* 1.4, pp. 541–551.

Lehtinen, J., J. Munkberg, J. Hasselgren, S. Laine, T. Karras, M. Aittala, and T. Aila (2018). Noise2Noise: Learning Image Restoration without Clean Data. *International Conference on Machine Learning*. PMLR, pp. 2965–2974.

Leshno, M., V. Y. Lin, A. Pinkus, and S. Schocken (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks* 6.6, pp. 861–867.

Li, H., Z. Xu, G. Taylor, C. Studer, and T. Goldstein (2018). Visualizing the loss landscape of neural nets. *Advances in neural information processing systems* 31.

Lin, T.-Y., P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie (2017a). Feature pyramid networks for object detection. *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2117–2125.

Lin, T.-Y., P. Goyal, R. Girshick, K. He, and P. Dollár (2017b). Focal loss for dense object detection. *Proceedings of the IEEE international conference on computer vision*, pp. 2980–2988.

Lin, T.-Y., M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Doll'ar, and C. L. Zitnick (2014). Microsoft coco: Common objects in context. *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*. Springer, pp. 740–755.

Liu, W., D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg (2016). SSD: Single shot multibox detector. *European conference on computer vision*. Springer, pp. 21–37.

Luo, W., Y. Li, R. Urtasun, and R. Zemel (2016). Understanding the effective receptive field in deep convolutional neural networks. *Advances in neural information processing systems* 29.

Mahendran, A. and A. Vedaldi (2016). Visualizing deep convolutional neural networks using natural pre-images. *International Journal of Computer Vision* 120, pp. 233–255.

Martin, C. D. and M. A. Porter (2012). The extraordinary SVD. *The American Mathematical Monthly* 119.10, pp. 838–851.

Morales-Brotons, D., T. Vogels, and H. Hendrikx (2024). Exponential Moving Average of Weights in Deep Learning: Dynamics and Benefits. *Trans. Mach. Learn. Res.*

Nair, V. and G. E. Hinton (2010). Rectified linear units improve restricted boltzmann machines. *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814.

Park, S., C. Yun, J. Lee, and J. Shin (2021). Minimum width for universal approximation. *9th International Conference on Learning Representations, ICLR 2021*.

Petersen, K. B., M. S. Pedersen, et al. (2008). The matrix cookbook. *Technical University of Denmark* 7.15, p. 510.

Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *Ussr computational mathematics and mathematical physics* 4.5, pp. 1–17.

Polyak, B. T. and A. B. Juditsky (1992). Acceleration of stochastic approximation by averaging. *SIAM journal on control and optimization* 30.4, pp. 838–855.

Poyser, M. and T. P. Breckon (2024). Neural architecture search: A contemporary literature review for computer vision applications. *Pattern Recognition* 147, p. 110052.

Ravi, N., V. Gabeur, Y.-T. Hu, R. Hu, C. Ryali, T. Ma, H. Khedr, R. Rädle, C. Rolland, L. Gustafson, et al. (2024). Sam 2: Segment anything in images and videos. *arXiv preprint arXiv:2408.00714*.

Redmon, J., S. Divvala, R. Girshick, and A. Farhadi (2016). You only look once: Unified, real-time object detection. *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788.

Ren, S., K. He, R. Girshick, and J. Sun (2015). Faster R-CNN: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems* 28.

Ronneberger, O., P. Fischer, and T. Brox (2015). U-net: Convolutional networks for biomedical image segmentation. *International Conference on Medical image computing and computer-assisted intervention*. Springer, pp. 234–241.

Salamon, J. and J. P. Bello (2017). Deep convolutional neural networks and data augmentation for environmental sound classification. *IEEE Signal processing letters* 24.3, pp. 279–283.

Senior, A. W., R. Evans, J. Jumper, J. Kirkpatrick, L. Sifre, T. Green, C. Qin, A. Žídek, A. W. Nelson, A. Bridgland, et al. (2020). Improved protein structure prediction using potentials from deep learning. *Nature* 577.7792, pp. 706–710.

Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature* 529.7587, pp. 484–489.

Simonyan, K and A Zisserman (2015). Very deep convolutional networks for large-scale image recognition. *3rd International Conference on Learning Representations (ICLR 2015)*. Computational and Biological Learning Society.

Simonyan, K. and A. Zisserman (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Sitzmann, V., J. Martel, A. Bergman, D. Lindell, and G. Wetzstein (2020). Implicit neural representations with periodic activation functions. *Advances in neural information processing systems* 33, pp. 7462–7473.

Szegedy, C., W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich (2015). Going deeper with convolutions. *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9.

Vandenhende, S., S. Georgoulis, W. Van Gansbeke, M. Proesmans, D. Dai, and L. Van Gool (2021). Multi-task learning for dense prediction tasks: A survey. *IEEE transactions on pattern analysis and machine intelligence* 44.7, pp. 3614–3633.

Viola, P. and M. Jones (2001). Rapid object detection using a boosted cascade of simple features. *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001.* Vol. 1. Ieee, pp. I–I.

Winding, M., B. D. Pedigo, C. L. Barnes, H. G. Patsolic, Y. Park, T. Kazimiers, A. Fushiki, I. V. Andrade, A. Khandelwal, J. Valdes-Aleman, et al. (2023). The connectome of an insect brain. *Science* 379.6636, eadd9330.

Wu, Y. and K. He (2018). Group normalization. *Proceedings of the European conference on computer vision (ECCV)*, pp. 3–19.

Xie, S., R. Girshick, P. Dollár, Z. Tu, and K. He (2017). Aggregated residual transformations for deep neural networks. *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1492–1500.

Yang, L., B. Kang, Z. Huang, X. Xu, J. Feng, and H. Zhao (2024a). Depth anything: Unleashing the power of large-scale unlabeled data. *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 10371–10381.

Yang, L., B. Kang, Z. Huang, Z. Zhao, X. Xu, J. Feng, and H. Zhao (2024b). Depth anything v2. *Advances in Neural Information Processing Systems* 37, pp. 21875–21911.

Zhang, A., Z. C. Lipton, M. Li, and A. J. Smola (2023). *Dive into deep learning*. Cambridge University Press.

Zhou, X., D. Wang, and P. Krähenbühl (2019). Objects as points. *arXiv preprint arXiv:1904.07850.*

Zou, Z., K. Chen, Z. Shi, Y. Guo, and J. Ye (2023). Object detection in 20 years: A survey. *Proceedings of the IEEE* 111.3, pp. 257–276.

Index