Deep learning: basics and convolutional neural networks (CNN)

Maria Vakalopoulou, Stergios Christodoulidis, Ninon Burgos, Olivier Colliot, Vincent Lepetit

Chapter 3

Deep Learning: Basics and Convolutional Neural Networks (CNNs)

Maria Vakalopoulou, Stergios Christodoulidis, Ninon Burgos, Olivier Colliot, and Vincent Lepetit

Abstract

Deep learning belongs to the broader family of machine learning methods and currently provides state-of-the-art performance in a variety of fields, including medical applications. Deep learning architectures can be categorized into different groups depending on their components. However, most of them share similar modules and mathematical formulations. In this chapter, the basic concepts of deep learning will be presented to provide a better understanding of these powerful and broadly used algorithms. The analysis is structured around the main components of deep learning architectures, focusing on convolutional neural networks and autoencoders.

Key words Perceptrons, Backpropagation, Convolutional neural networks, Deep learning, Medical imaging

1 Introduction

Recently, deep learning frameworks have become very popular, attracting a lot of attention from the research community. These frameworks provide machine learning schemes without the need for feature engineering, while at the same time they remain quite flexible. Initially developed for supervised tasks, they are nowadays extended to many other settings. Deep learning, in the strict sense, involves the use of multiple layers of artificial neurons. The first artificial neural networks were developed in the late 1950s with the presentation of the perceptron [1] algorithms. However, limitations related to the computational costs of these algorithms during that period, as well as the often-miscited claim of Minsky and Papert [2] that perceptrons are not capable of learning non-linear functions such as the XOR, caused a significant decline of interest for further research on these algorithms and contributed to the so-called artificial intelligence winter. In particular, in their book [2], Minsky and Papert discussed that single-layer perceptrons are

Olivier Colliot (ed.), *Machine Learning for Brain Disorders*, Neuromethods, vol. 197, https://doi.org/10.1007/978-1-0716-3195-9_3,

only capable of learning linearly separable patterns. It was often incorrectly believed that they also presumed this is the case for multilayer perceptron networks. It took more than 10 years for research on neural networks to recover, and in [3], some of these issues were clarified and further discussed. Even if during this period there was not a lot of research interest for perceptrons, very important algorithms such as the backpropagation algorithm [4–7] and recurrent neural networks [8] were introduced.

After this period, and in the early 2000s, publications by Hinton, Osindero, and Teh [9] indicated efficient ways to train multilayer perceptrons layer by layer, treating each layer as an unsupervised restricted Boltzmann machine and then using supervised backpropagation for the fine-tuning [10]. Such advances in the optimization algorithms and in hardware, in particular graphics processing units (GPUs), increased the computational speed of deep learning systems and made their training easier and faster. Moreover, around 2010, the first large-scale datasets, with ImageNet [11] being one of the most popular, were made available, contributing to the success of deep learning algorithms, allowing the experimental demonstration of their superior performance on several tasks in comparison with other commonly used machine learning algorithms. Finally, another very important factor that contributed to the current popularity of deep learning techniques is their support by publicly available and easy-to-use libraries such as Theano [12], Caffe [13], TensorFlow [14], Keras [15], and PyTorch [16]. Indeed, currently, due to all these publicly available libraries that facilitate collaborative and reproducible research and access to resources from large corporations such as Kaggle, Google Colab, and Amazon Web Services, teaching and research about these algorithms have become much easier.

This chapter will focus on the presentation and discussion of the main components of deep learning algorithms, giving the reader a better understanding of these powerful models. The chapter is meant to be readable by someone with no background in deep learning. The basic notions of machine learning will not be included here; however, the reader should refer to Chap. 2 (reader without a background in engineering or computer science can also refer to Chap. 1 for a lay audience-oriented presentation of these concepts). The rest of this chapter is organized as follows. We will first present the deep feedforward networks focusing on perceptrons, multilayer perceptrons, and the main functions that they are composed of (Subheading 2). Then, we will focus on the optimization of deep neural networks, and in particular, we will formally present the topics of gradient descent, backpropagation, as well as the notions of generalization and overfitting (Subheading 3). Subheading 4 will focus on convolutional neural networks discussing in detail the basic convolution operations, while Subheading 5 will give an overview of the autoencoder architectures.

2 Deep Feedforward Networks

In this section, we will present the early deep learning approaches together with the main functions that are commonly used in deep feedforward networks. Deep feedforward networks are a set of parametric, non-linear, and hierarchical representation models that are optimized with stochastic gradient descent. In this definition, the term parametric holds due to the parameters that we need to learn during the training of these models, the non-linearity due to the non-linear functions that they are composed of, and the hierarchical representation due to the fact that the output of one function is used as the input of the next in a hierarchical way.

2.1 Perceptrons

The perceptron [1] was originally developed for supervised binary classification problems, and it was inspired by works from neuroscientists such as Donald Hebb [17]. It was built around a non-linear neuron, namely, the McCulloch-Pitts model of a neuron. More formally, we are looking for a function $f(\boldsymbol{x};\boldsymbol{w}, b)$ such that $f(.;\boldsymbol{w}, b) : \boldsymbol{x}\in\mathbb{R}^p \rightarrow \{+1, -1\}$ where $\boldsymbol{w}$ and b are the parameters of f and the vector $\boldsymbol{x}=[x_1, \ldots, x_p]^\top$ is the input. The training set is $\{(\boldsymbol{x}^{(i)}, y^{(i)})\}$. In particular, the perceptron relies on a linear model for performing the classification:

$$f(\boldsymbol{x};\boldsymbol{w}, b)=\begin{cases}+1 & \text{if } \boldsymbol{w}^\top\boldsymbol{x}+b\geq 0\\ -1 & \text{otherwise}\end{cases}. \tag{1}$$

Such a model can be interpreted geometrically as a hyperplane that can appropriately divide data points that are linearly separable. Moreover, one can observe that, in the previous definition, a perceptron is a combination of a weighted summation between the elements of the input vector $\boldsymbol{x}$ combined with a step function that performs the decision for the classification. Without loss of generality, this step function can be replaced by other activation functions such as the sigmoid, hyperbolic tangent, or softmax functions (*see* Subheading 2.3); the output simply needs to be thresholded to assign the + 1 or − 1 class. Graphically, a perceptron is presented in Fig. 1 on which each of the elements of the input is described as a neuron and all the elements are combined by weighting with the models' parameters and then passed to an activation function for the final decision.

During the training process and similarly to the other machine learning algorithms, we need to find the optimal parameters $\boldsymbol{w}$ and b for the perceptron model. One of the main innovations of Rosenblatt was the proposition of the learning algorithm using an iterative process. First, the weights are initialized randomly, and then using one sample $(\boldsymbol{x}^{(i)}, y^{(i)})$ of the training set, the prediction of the

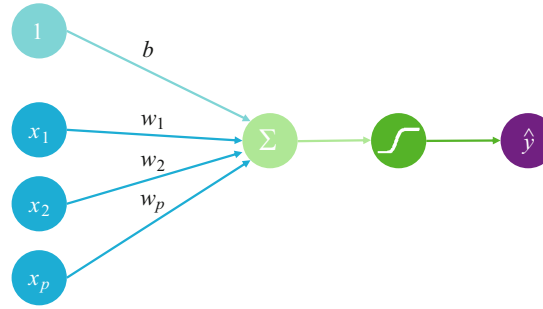

Fig. 1 A simple perceptron model. The input elements are described as neurons and combined for the final prediction $\hat{y}$. The final prediction is composed of a weighted sum and an activation function

perceptron is calculated. If the prediction is correct, no further action is needed, and the next data point is processed. If the prediction is wrong, the weights are updated with the following rule: the weights are increased in case the prediction is smaller than the ground-truth label $y^{(i)}$ and decreased if the prediction is higher than the ground-truth label. This process is repeated until no further errors are made for the data points. A pseudocode of the training or convergence algorithm is presented in Algorithm 1 (note that in this version, it is assumed that the data is linearly separable).

Algorithm 1 Train perceptron

procedure TRAIN($\{(\boldsymbol{x}^{(i)}, y^{(i)})\}$)
  Initialization: initialize randomly the weights $\boldsymbol{w}$ and bias b
  while $\exists i \in \{1, \ldots, n\}, f(\boldsymbol{x}^{(i)}; \boldsymbol{w}, b) \neq y^{(i)}$ **do**
    Pick i randomly
    error $= y^{(i)} - f(\boldsymbol{x}^{(i)}; \boldsymbol{w}, b)$
    if error $\neq 0$ **then**
      $\boldsymbol{w} \leftarrow \boldsymbol{w} + error \cdot \boldsymbol{x}^{(i)}$
      $b \leftarrow b + error$

Originally, the perceptron has been proposed for binary classification tasks. However, this algorithm can be generalized for the case of multiclass classification, $f_c(\boldsymbol{x}; \boldsymbol{w}, b)$, where $c \in \{1, \ldots, C\}$ are the different classes. This can be easily achieved by adding more neurons to the output layer of the perceptron. That way, the number of output neurons would be the same as the number of possible outputs we need to predict for the specific problem. Then, the final decision can be made by choosing the maximum of the different output neurons $f_n = \max_{c \in \{1, \ldots, C\}} f_c(\boldsymbol{x}; \boldsymbol{w}, b)$.

Finally, in the following, we will integrate the bias b in the weights $\boldsymbol{w}$ (and thus add 1 as the first element of the input vector $\boldsymbol{x} = [1, x_1, \ldots, x_p]^\top$). The model can then be rewritten as $f(\boldsymbol{x}; \boldsymbol{w})$ such that $f(.; \boldsymbol{w}) : \boldsymbol{x} \in \mathbb{R}^{p+1} \rightarrow \{+1, -1\}$.

2.2 Multilayer Perceptrons

The limitation of perceptrons to linear problems can be overcome by using multilayer perceptions, often denoted as MLP. An MLP consists of at least three layers of neurons: the input layer, a hidden layer, and an output layer. Except for the input neurons, each neuron uses a non-linear activation function, making it capable of distinguishing data that is not linearly separable. These layers can also be called fully connected layers since they connect all the neurons of the previous and of the current layer. It is absolutely crucial to keep in mind that non-linear functions are necessary for the network to find non-linear separations in the data (otherwise, all the layers could simply be collapsed together into a single gigantic linear function).

2.2.1 A Simple Multilayer Network

Without loss of generality, an MLP with one hidden layer can be defined as:

$$\begin{cases} \boldsymbol{z}(\boldsymbol{x}) = g(\boldsymbol{W}^1 \boldsymbol{x}) \\ \hat{\boldsymbol{y}} = f(\boldsymbol{x}; \boldsymbol{W}^1, \boldsymbol{W}^2) = \boldsymbol{W}^2 \boldsymbol{z}(\boldsymbol{x}) \end{cases}, \tag{2}$$

where $g(x) : \mathbb{R} \rightarrow \mathbb{R}$ denotes the non-linear function (which can be applied element-wise to a vector), $\boldsymbol{W}^1$ the matrix of coefficients of the first layer, and $\boldsymbol{W}^2$ the matrix of coefficients of the second layer.

Equivalently, one can write:

$$\hat{y_c} = \sum_{j=1}^{d_1} \boldsymbol{W}^2_{(c,j)} g(\boldsymbol{W}^{1\top}_{(j)} \boldsymbol{x}), \tag{3}$$

where d_1 is the number of neurons for the hidden layer which defines the width of the network, $\boldsymbol{W}^1_{(j)}$ denotes the first column of the matrix $\boldsymbol{W}^1$, and $\boldsymbol{W}^2_{(c,j)}$ denotes the c, j element of the matrix $\boldsymbol{W}^2$. Graphically, a two-layer perceptron is presented in Fig. 2 on

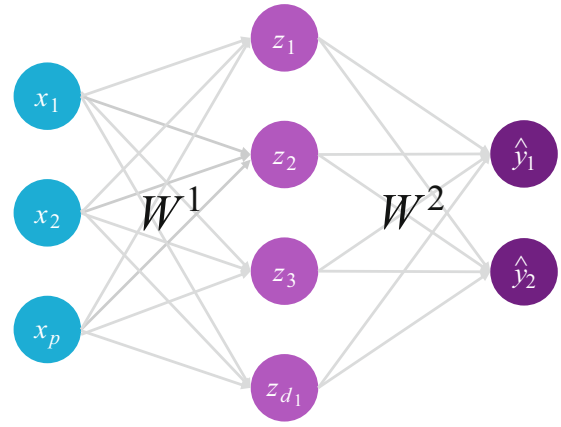

Fig. 2 An example of a simple multilayer perceptron model. The input layer is fed into a hidden layer ($\boldsymbol{z}$), which is then combined for the last output layer providing the final prediction

which the input neurons are fed into a hidden layer whose neurons are combined for the final prediction.

There were a lot of research works indicating the capacity of feedforward neural networks with a single hidden layer of finite size to approximate continuous functions. In the late 1980s, the first proof was published [18] for sigmoid activation functions (*see* Subheading 2.3 for the definition) and was generalized to other functions for feedforward multilayer architectures [19–21]. In particular, these works prove that any continuous function can be approximated under mild conditions as closely as wanted by a three-layer network. As $N \to \infty$, any continuous function f can be approximated by some neural network $\hat{f}$, because each component $g(\boldsymbol{W}_{(j)}^{T}\boldsymbol{x})$ behaves like a basis function and functions in a suitable space admit a basis expansion. However, since N may need to be very large, introducing some limitations for these types of networks, deeper networks, with more than one hidden layer, can provide good alternatives.

2.2.2 Deep Neural Network

The simple MLP networks can be generalized to deeper networks with more than one hidden layer that progressively generate higher-level features from the raw input. Such networks can be written as:

$$\begin{cases} \boldsymbol{z_1}(\boldsymbol{x}) = g(\boldsymbol{W^1}\boldsymbol{x}) \\ \dots \\ \boldsymbol{z_k}(\boldsymbol{x}) = g(\boldsymbol{W^k}\boldsymbol{z_{k-1}}(\boldsymbol{x})) \\ \dots \\ \hat{\boldsymbol{y}} = f(\boldsymbol{x}; \boldsymbol{W}^1, \dots, \boldsymbol{W}^K) = \boldsymbol{z_K}(\boldsymbol{z_{K-1}}(\dots(\boldsymbol{z_1}(\boldsymbol{x})))) \end{cases}, \tag{4}$$

where K denotes the number of layers for the neural network, which defines the depth of the network. In Fig. 3, a graphical representation of the deep multilayer perceptron is presented. Once again, the input layer is fed into the different hidden layers of the network in a hierarchical way such that the output of one layer is the input of the next one. The last layer of the network corresponds to the output layer, which makes the final prediction of the model.

As for networks with one hidden layer, they are also universal approximators. However, the approximation theory for deep networks is less understood compared with neural networks with one hidden layer. Overall, deep neural networks excel at representing the composition of functions.

So far, we have described neural networks as simple chains of layers, applied in a hierarchical way, with the main considerations being the depth of the network (the number of layers K) and the

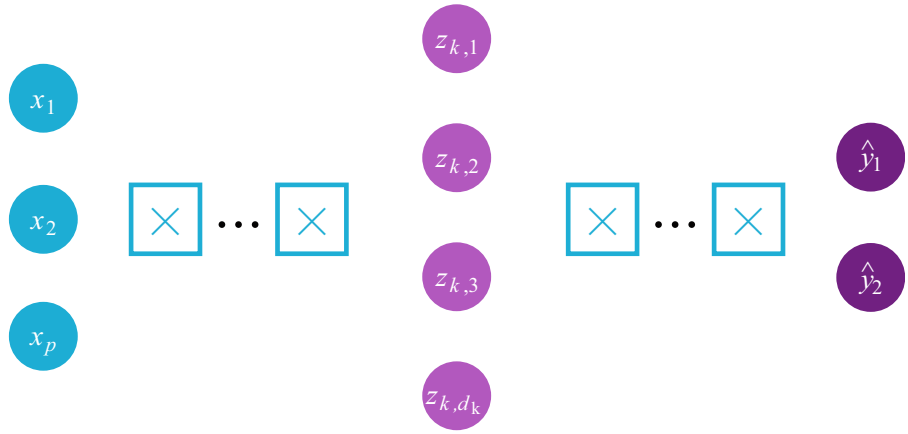

Fig. 3 An example of a deep neural network. The input layer, the *k*th layer of the deep neural network, and the output layer are presented in the figure

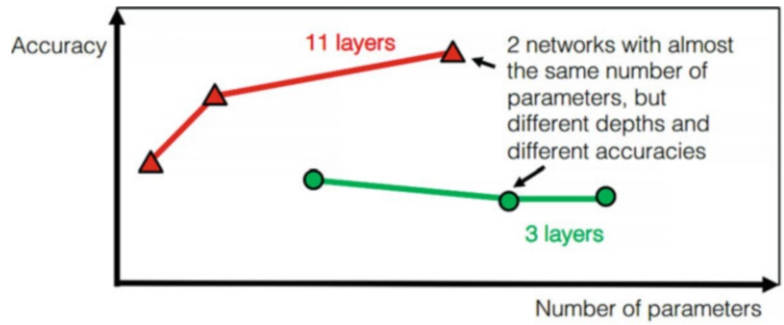

Fig. 4 Comparison of two different networks with almost the same number of parameters, but different depths. Figure inspired by Goodfellow et al. [24]

width of each k layer (the number of neurons d_k). Overall, there are no rules for the choice of the K and d_k parameters that define the architecture of the MLP. However, it has been shown empirically that deeper models perform better. In Fig. 4, an overview of 2 different networks with 3 and 11 hidden layers is presented with respect to the number of parameters and their accuracy. For each architecture, the number of parameters varies by changing the number of neurons d_k. One can observe that, empirically, deeper networks achieve better performance using approximately the same or a lower number of parameters. Additional evidence to support these empirical findings is a very active field of research [22, 23].

Neural networks can come in a variety of models and architectures. The choice of the proper architecture and type of neural network depends on the type of application and the type of data.

Most of the time, the best architecture is defined empirically. In the next section, we will discuss the main functions used in neural networks.

2.3 Main Functions

A neural network is a composition of different functions also called modules. Most of the times, these functions are applied in a sequential way. However, in more complicated designs (e.g., deep residual networks), different ways of combining them can be designed. In the following subsections, we will discuss the most commonly used functions that are the backbones of most perceptrons and multilayer perceptron architectures. One should note, however, that a variety of functions can be proposed and used for different deep learning architectures with the constraint to be differentiable – almost – everywhere. This is mainly due to the way that deep neural networks are trained, and this will be discussed later in the chapter.

2.3.1 Linear Functions

One of the most fundamental functions used in deep neural networks is the simple linear function. Linear functions produce a linear combination of all the nodes of one layer of the network, weighted with the parameters $\boldsymbol{W}$. The output signal of the linear function is $\boldsymbol{Wx}$, which is a polynomial of degree one. While it is easy to solve linear equations, they have less power to learn complex functional mappings from data. Moreover, when the number of samples is much larger than the dimension of the input space, the probability that the data is linearly separable comes close to zero (Box 1). This is why they need to be combined with non-linear functions, also called activation functions (the name activation has been initially inspired by biology as the neuron will be active or not depending on the output of the function).

Box 1: Function Counting Theorem

The so-called Function Counting Theorem (Cover [25]) counts the number of linearly separable dichotomies of n points in general position in $\mathbb{R}^p$. The theorem shows that, out of the total 2^n dichotomies, only $C(n,p) = 2\sum_{j=0}^{p}\binom{n-1}{j}$ are homogeneously, linearly separable.

When $n >> p$, the probability of a dichotomy to be linearly separable converges to zero. This indicates the need for the integration of non-linear functions into our modeling and architecture design. Note that $n >> p$ is a typical regime in machine learning and deep learning applications where the number of samples is very large.

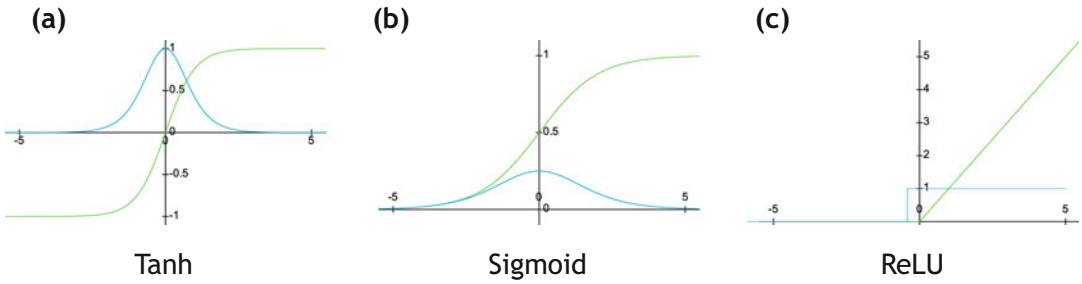

Fig. 5 Overview of different non-linear functions (in green) and their first-order derivative (blue). (**a**) Hyperbolic tangent function (tanh), (**b**) sigmoid, and (**c**) rectified linear unit (ReLU)

2.3.2 Non-linear Functions

One of the most important components of deep neural networks is the non-linear functions, also called activation functions. They convert the linear input signal of a node into non-linear outputs to facilitate the learning of high-order polynomials. There are a lot of different non-linear functions in the literature. In this subsection, we will discuss the most classical non-linearities.

Hyperbolic Tangent Function (tanh)

One of the most standard non-linear functions is the hyperbolic tangent function, aka the tanh function. Tanh is symmetric around the origin with a range of values varying from -1 to 1. The biggest advantage of the tanh function is that it produces a zero-centered output (Fig. 5a), thereby supporting the backpropagation process that we will cover in the next section. The tanh function is used extensively for the training of multilayer neural networks. Formally, the tanh function, together with its gradient, is defined as:

$$\begin{aligned} g &= \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \\ \frac{\partial g}{\partial x} &= 1 - \tanh^{2}(x) \end{aligned} \quad . \tag{5}$$

One of the downsides of tanh is the saturation of gradients that occurs for large or small inputs. This can slow down the training of the networks.

Sigmoid

Similar to tanh, the sigmoid is one of the first non-linear functions that were used to compose deep learning architectures. One of the main advantages is that it has a range of values varying from 0 to 1 (Fig. 5b) and therefore is especially used for models that aim to predict a probability as an output. Formally, the sigmoid function, together with its gradient, is defined as:

$$\begin{aligned} g &= \sigma(x) = \frac{1}{1 + e^{-x}} \\ \frac{\partial g}{\partial x} &= \sigma(x)(1 - \sigma(x)) \end{aligned} \quad . \tag{6}$$

Note that this is in fact the logistic function, which is a special case of the more general class of sigmoid function. As it is indicated in Fig. 5b, the sigmoid gradient vanishes for large or small inputs making the training process difficult. However, in case it is used for the output units which are not latent variables and on which we have access to the ground-truth labels, sigmoid may be a good option.

Rectified Linear Unit (ReLU)

ReLU is considered among the default choice of non-linearity. Some of the main advantages of ReLU include its efficient calculation and better gradient propagation with fewer vanishing gradient problems compared to the previous two activation functions [26]. Formally, the ReLU function, together with its gradient, is defined as:

$$\begin{aligned} g &= \max(0, x) \\ \frac{\partial g}{\partial x} &= \begin{cases} 0, \text{if } x \leq 0 \\ 1, \text{if } x > 0 \end{cases} \end{aligned} . \tag{7}$$

As it is indicated in Fig. 5c, ReLU is differentiable anywhere else than zero. However, this is not a very important problem as the value of the derivative at zero can be arbitrarily chosen to be 0 or 1. In [27], the authors empirically demonstrated that the number of iterations required to reach 25% training error on the CIFAR-10 dataset for a four-layer convolutional network was six times faster with ReLU than with tanh neurons. On the other hand, and as discussed in [28], ReLU-type neural networks which yield a piecewise linear classifier function produce almost always high confidence predictions far away from the training data. However, due to its efficiency and popularity, many variations of ReLU have been proposed in the literature, such as the leaky ReLU [29] or the parametric ReLU [30]. These two variations both address the problem of dying neurons, where some ReLU neurons die for all inputs and remain inactive no matter what input is supplied. In such a case, no gradient flows from these neurons, and the training of the neural network architecture is affected. Leaky ReLU and parametric ReLU change the $g(x) = 0$ part, by adding a slope and extending the range of ReLU.

Swish

The choice of the activation function in neural networks is not always easy and can greatly affect performance. In [31], the authors performed a combination of exhaustive and reinforcement learning-based searches to discover novel activation functions. Their experiments discovered a new activation function that is called Swish and is defined as:

$$\begin{aligned} g &= \boldsymbol{x} \cdot \sigma(\beta \boldsymbol{x}) \\ \frac{\partial g}{\partial \boldsymbol{x}} &= \beta g(\boldsymbol{x}) + \sigma(\beta \boldsymbol{x})(1 - \beta g(\boldsymbol{x})) \end{aligned}, \tag{8}$$

where σ is the sigmoid function and β is either a constant or a trainable parameter. Swish tends to work better than ReLU on deeper models, as it has been shown experimentally in [31] in different domains.

Softmax

Softmax is often used as the last activation function of a neural network. In practice, it normalizes the output of a network to a probability distribution over the predicted output classes. Softmax is defined as:

$$\text{Softmax}(x_i) = \frac{e_i^x}{\sum_j^C e_j^x}. \tag{9}$$

The softmax function takes as input a vector $\boldsymbol{x}$ of C real numbers and normalizes it into a probability distribution consisting of C probabilities proportional to the exponentials of the input numbers. However, a limitation of softmax is that it assumes that every input $\boldsymbol{x}$ belongs to at least one of the C classes (which is not the case in practice, i.e., the network could be applied to an input that does not belong to any of the classes).

2.3.3 Loss Functions

Besides the activation functions, the loss function (which defines the cost function) is one of the main elements of neural networks. It is the function that represents the error for a given prediction. To that purpose, for a given training sample, it compares the prediction $f(\boldsymbol{x}^{(i)}; \boldsymbol{W})$ to the ground truth $y^{(i)}$ (here we denote for simplicity as $\boldsymbol{W}$ all the parameters of the network, combining all the $\boldsymbol{W^1}, \ldots, \boldsymbol{W^K}$ in the multilayer perceptron shown above). The loss is denoted as $\ell(y, f(\boldsymbol{x}; \boldsymbol{W}))$. The average loss across the n training samples is called the cost function and is defined as:

$$J(\boldsymbol{W}) = \frac{1}{n} \sum_{i=1}^{n} \ell\left(y^{(i)}, f(\boldsymbol{x}^{(i)}; \boldsymbol{W})\right), \tag{10}$$

where $\{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1..n}$ composes the training set. The aim of the training will be to find the parameters $\boldsymbol{W}$ such that $J(\boldsymbol{W})$ is minimized. Note that, in deep learning, one often calls the cost function the loss function, although, strictly speaking, the loss is for a given sample, and the cost is averaged across samples. Besides, the objective function is the overall function to minimize, including the cost and possible regularization terms. However, in the remainder of this chapter, in accordance with common usage in deep learning, we will sometimes use the term loss function instead of cost function.

In neural networks, the loss function can be virtually any function that is differentiable. Below we present the two most common losses, which are, respectively, used for classification or regression problems. However, specific losses exist for other tasks, such as segmentation, which are covered in the corresponding chapters.

Cross-Entropy Loss

One of the most basic loss functions for classification problems corresponds to the cross-entropy between the expected values and the predicted ones. It leads to the following cost function:

$$J(\boldsymbol{W}) = -\sum_{i=1}^{n} \log\left(P\left(\mathrm{y} = y^{(i)} | \mathbf{x} = \boldsymbol{x}^{(i)}; \boldsymbol{W}\right)\right), \tag{11}$$

where $P\left(\mathrm{y}= y^{(i)} | \mathbf{x} = \boldsymbol{x}^{(i)}; \boldsymbol{W}\right)$ is the probability that a given sample is correctly classified.

The cross-entropy can also be seen here as the negative log-likelihood of the training set given the predictions of the network. In other words, minimizing this loss function corresponds to maximizing the likelihood:

$$J(\boldsymbol{W}) = \prod_{i=1}^{n} P\left(\mathrm{y} = y^{(i)} | \mathbf{x} = \boldsymbol{x}^{(i)}; \boldsymbol{W}\right). \tag{12}$$

Mean Squared Error Loss

For regression problems, the mean squared error is one of the most basic cost functions, measuring the average of the squares of the errors, which is the average squared difference between the predicted values and the real ones. The mean squared error is defined as:

$$J(\boldsymbol{W}) = \sum_{i=1}^{n} || \, y^{(i)} - f(\boldsymbol{x}^{(i)}; \boldsymbol{W}) \, ||^{2}. \tag{13}$$

3 Optimization of Deep Neural Networks

Optimization is one of the most important components of neural networks, and it focuses on finding the parameters $\boldsymbol{W}$ that minimize the loss function $J(\boldsymbol{W})$. Overall, optimization is a difficult task. Traditionally, the optimization process is performed by carefully designing the loss function and integrating its constraints to ensure that the optimization process is convex (and thus, one can be sure to find the global minimum). However, neural networks are non-convex models, making their optimization challenging, and, in general, one does not find the global minimum but only a local one. In the next sections, the main components of their optimization will be presented, giving a general overview of the optimization process, its challenges, and common practices.

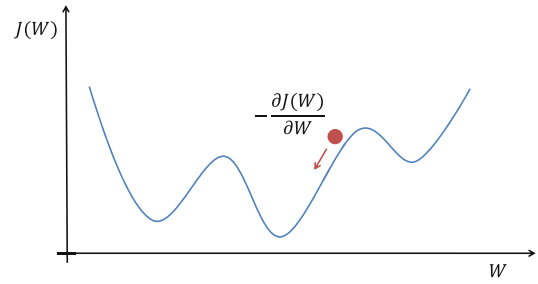

Fig. 6 The gradient descent algorithm. This first-order optimization algorithm is finding a local minimum by taking steps toward the opposite direction of the gradient

3.1 Gradient Descent

Gradient descent is an iterative optimization algorithm that is among the most popular and basic algorithms in machine learning. It is a first-order[1] optimization algorithm, which is finding a local minimum of a differentiable function. The main idea of gradient descent is to take iterative steps toward the opposite direction of the gradient of the function that needs to be optimized (Fig. 6).

That way, the parameters $\boldsymbol{W}$ of the model are updated by:

$$\boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^{t} - \eta \frac{\partial J(\boldsymbol{W}^{t})}{\partial \boldsymbol{W}^{t}}, \tag{14}$$

where t is the iteration and η, called learning rate, is the hyperparameter that indicates the magnitude of the step that the algorithm will take.

Besides its simplicity, gradient descent is one of the most commonly used algorithms. More sophisticated algorithms require computing the Hessian (or an approximation) and/or its inverse (or an approximation). Even if these variations could give better optimization guarantees, they are often more computationally expensive, making gradient descent the default method for optimization.

In the case of convex functions, the optimization problem can be reduced to the problem of finding a local minimum. Any local minimum is then guaranteed to be a global minimum, and gradient descent can identify it. However, when dealing with non-convex functions, such as neural networks, it is possible to have many local minima making the use of gradient descent challenging. Neural networks are, in general, non-identifiable [24]. A model is said to be identifiable if it is theoretically possible, given a sufficiently large training set, to rule out all but one set of the model's parameters. Models with latent variables, such as the hidden layers of neural networks, are often not identifiable because we can obtain equivalent models by exchanging latent variables with each other.

[1] First-order means here that the first-order derivatives of the cost function are used as opposed to second-order algorithms that, for instance, use the Hessian.

However, all these minima are often almost equivalent to each other in cost function value. In that case, these local minima are not a problematic form of non-convexity. It remains an open question whether there exist many local minima with a high cost that prevent adequate training of neural networks. However, it is currently believed that most local minima, at least as found by modern optimization procedures, will correspond to a low cost (even though not to identical costs) [24].

For $\boldsymbol{W}^*$ to be a local minimum, we need mainly two conditions to be fulfilled:

- $\left\| \frac{\partial J}{\partial \boldsymbol{W}} (\boldsymbol{W}^*) \right\| = 0.$
- All the eigenvalues of $\left(\frac{\partial^2 J}{\partial \boldsymbol{W}^2} (\boldsymbol{W}^*) \right)$ to be positive.

For random functions in n dimensions, the probability for the eigenvalues to be all positive is $\frac{1}{n}$. On the other hand, the ratio of the number of saddle points to local minima increases exponentially with n [32]. A saddle point, or critical point, is a point where the derivatives are zero without being a minimum of the function. Such points could result in a high error making the optimization with gradient descent challenging. In [32], this issue is discussed, and an optimization algorithm that leverages second-order curvature information is proposed to deal with this issue for deep and recurrent networks.

3.1.1 Stochastic Gradient Descent

Gradient descent efficiency is not enough when it comes to machine learning problems with large numbers of training samples. Indeed, this is the case for neural networks and deep learning which often rely on hundreds or thousands of training samples. Updating the parameters $\boldsymbol{W}$ after calculating the gradient using all the training samples would lead to a tremendous computational complexity of the underlying optimization algorithm [33]. To deal with this problem, the stochastic gradient descent (SGD) algorithm is a drastic simplification. Instead of computing the $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$ exactly, each iteration estimates this gradient on the basis of a small set of randomly picked examples, as follows:

$$\boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^t - \eta_t G(\boldsymbol{W}^t), \tag{15}$$

where

$$G(\boldsymbol{W}^t) = \frac{1}{K} \sum_{k=1}^{K} \frac{\partial J_{(i_k)} \boldsymbol{W}^t}{\partial \boldsymbol{W}}, \tag{16}$$

where J_{i_k} is the loss function at training sample i_k, $\{(\boldsymbol{x}^{(i_k)}, y^{(i_k)})\}_{k=1\ldots K}$ is the small subset of K training samples ($K << N$). This subset of K samples is called a mini-batch or sometimes a batch.[2] In such a way, the iteration cost of stochastic

[2] Note that, as often in deep learning, the terminology can be confusing. In isolation, the term batch is usually a synonym of mini-batch. On the contrary, batch gradient descent means computing the gradient using all training samples and not only a mini-batch [24].

gradient descent will be $\mathcal{O}(K)$ and for gradient descent $\mathcal{O}(N)$. The ideal choice for the batch size is a debated question. First, an upper limit for the batch size is often simply given the available GPU memory, in particular when the size of the input data is large (e.g., 3D medical images). Besides, choosing K as a power of 2 often leads to more efficient computations. Finally, small batch sizes tend to have a regularizing effect which can be beneficial [24]. In any case, the ideal batch size usually depends on the application, and it is not uncommon to try different batch sizes. Finally, one calls an epoch a complete pass over the whole training set (meaning that each training sample has been used once). The number of epochs is the number of full passes over the whole training set. It should not be confused with the number of iterations which is the number of mini-batches that have been processed.

Note that various improvements over traditional SGD have been introduced, leading to more efficient optimization methods. These state-of-the-art optimization methods are presented in Subheading 3.4.

Box 2: Convergence of SGD Theorem

In [34], the authors prove that stochastic gradient descent converges if the network is sufficiently overparametrized. Let $(\boldsymbol{x}^{(i)}, y^{(i)})_{1\leq i\leq n}$ be a training set satisfying $\min_{i,j:i\neq j}\|\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}\|_2>\delta>0$. Consider fitting the data using a feedforward neural network with ReLU activations. Denote by D (resp. W) the depth (resp. width) of the network. Suppose that the neural network is sufficiently overparametrized, i.e.:

$$W \gg \text{polynomial}\left(n, D, \frac{1}{\delta}\right). \tag{17}$$

Then, with high probability, running SGD with *some random initialization* and properly chosen step sizes η_t yields $J(\boldsymbol{W}^t) < \epsilon$ in $t \propto \log \frac{1}{\epsilon}$.

3.2 Backpropagation

The training of neural networks is performed with backpropagation. Backpropagation computes the gradient of the loss function with respect to the parameters of the network in an efficient and local way. This algorithm was originally introduced in 1970. However, it started becoming very popular after the publication of [6], which indicated that backpropagation works faster than other methods that had been proposed back then for the training of neural networks.

Fig. 7 A multilayer perceptron with one hidden layer

The backpropagation algorithm works by computing the gradient of the loss function (J) with respect to each weight by the chain rule, computing the gradient one layer at a time, and iterating backward from the last layer to avoid redundant calculations of intermediate terms in the chain rule. In Fig. 7, an example of a multilayer perceptron with one hidden layer is presented. In such a network, the backpropagation is calculated as:

$$\begin{aligned} \frac{\partial J(\boldsymbol{W})}{\partial w_2} &= \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial w_2} \\ \frac{\partial J(\boldsymbol{W})}{\partial w_1} &= \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z_1} \times \frac{\partial z_1}{\partial w_1} \end{aligned} . \tag{18}$$

Overall, backpropagation is very simple and local. However, the reason why we can train a highly non-convex machine with many local minima, like neural networks, with a strong local learning algorithm is not really known even today. In practice, backpropagation can be computed in different ways, including manual calculation, numerical differentiation using finite difference approximation, and symbolic differentiation. Nowadays, deep learning frameworks such as [14, 16] use automatic differentiation [35] for the application of backpropagation.

3.3 Generalization and Overfitting

Similar to all the machine learning algorithms (discussed in Chapter 2), neural networks can suffer from poor generalization and overfitting. These problems are caused mainly by the optimization of the parameters of the models performed in the $\{(\boldsymbol{x_i}, y_i)\}_{i=1,\ldots,n}$ training set, while we need the model to perform well on other unseen data that are not available during the training. More formally, in the case of cross-entropy, the loss that we would like to minimize is:

$$J(\boldsymbol{W}) = -\log \prod_{(x,y)\in T_T} P(\mathrm{y} = y|\mathbf{x} = \boldsymbol{x}; \boldsymbol{W}), \tag{19}$$

where T_T is the set of any data, not available during training. In practice, a small validation set T_V is used to evaluate the loss on unseen data. Of course, this validation set should be distinct from the training set. It is extremely important to keep in mind that the performance obtained on the validation set is generally biased upward because the validation set was used to perform early stopping or to choose regularization parameters. Therefore, one should have an independent test set that has been isolated at the

beginning, has not been used in any way during training, and is only used to report the performance (*see* Chap. 20 for details). In case one cannot have an additional independent test set due to a lack of data, one should be aware that the performance may be biased and that this is a limitation of the specific study.

To avoid overfitting and improve the generalization performance of the model, usually, the validation set is used to monitor the loss during the training of the networks. Tracking the training and validation losses over the number of epochs is essential and provides important insights into the training process and the selected hyperparameters (e.g., choice of learning rate). Recent visualization tools such as TensorBoard[3] or Weights & Biases[4] make this tracking easy. In the following, we will also mention some of the most commonly applied optimization techniques that help with preventing overfitting.

Early Stopping Using the reported training and validation errors, the best model in terms of performance and generalization power is selected. In particular, early stopping, which corresponds to selecting a model corresponding to an earlier time point than the final epoch, is a common way to prevent overfitting [36]. Early stopping is a form of regularization for models that are trained with an iterative method, such as gradient descent and its variants. Early stopping can be implemented with different criteria. However, generally, it requires the monitoring of the performance of the model on a validation set, and the model is selected when its performance degrades or its loss increases. Overall, early stopping should be used almost universally for the training of neural networks [24]. The concept of early stopping is illustrated in Fig. 8.

Weight Regularization Similar to other machine learning methods (Chap. 2), weight regularization is also a very commonly used technique for avoiding overfitting in neural networks. More specifically, during the training of the model, the weights of the network start growing in size in order to specialize the model to the training data. However, large weights tend to cause sharp transitions in the different layers of the network and, that way, large changes in the output for only small changes in the inputs [37]. To handle this problem, during the training process, the weights can be updated in such a way that they are encouraged to be small, by adding a penalty to the loss function, for instance, the ℓ_2 norm of the parameters $\lambda \|\boldsymbol{W}\|^2$, where λ is a trade-off parameter between the loss and the regularization. Since weight regularization is quite popular in

[3] https://www.tensorflow.org/tensorboard.

[4] https://wandb.ai/site.

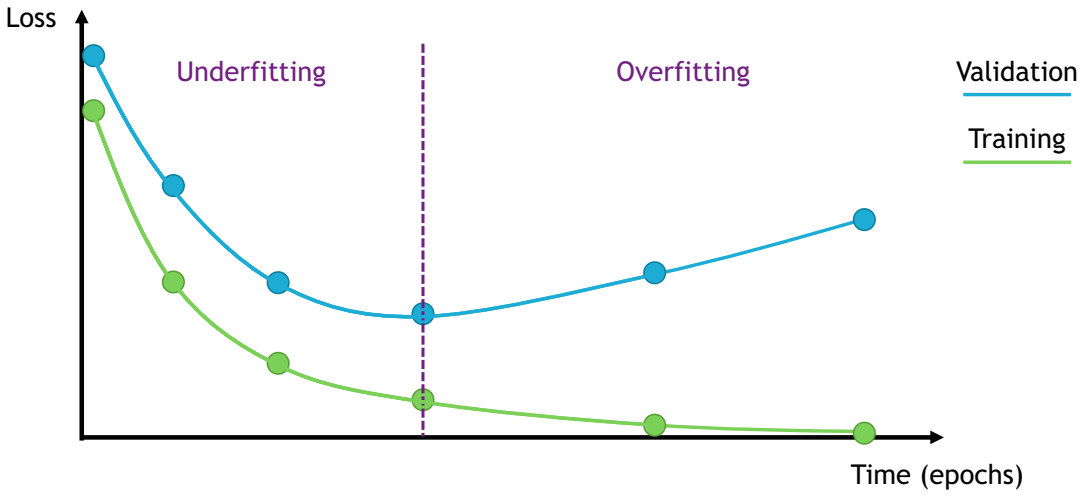

Fig. 8 Illustration of the concept of early stopping. The model that should be selected corresponds to the dashed bar which is the point where the validation loss starts increasing. Before this point, the model is underfitting. After, it is overfitting

neural networks, different optimizers have integrated them into their optimization process in the form of weight decay.

Weight Initialization The way that the weights of neural networks will be initialized is very important, and it can determine whether the algorithm converges at all, with some initial points being so unstable that the algorithm encounters numerical difficulties and fails altogether [24]. Most of the time, the weights are initialized randomly from a Gaussian or uniform distribution. According to [24], the choice of Gaussian or uniform distribution does not seem to matter very much; however, the scale does have a large effect both on the outcome of the optimization procedure and on the ability of the network to generalize. Nevertheless, more tailored approaches have been developed over the last decade that have become the standard initialization points. One of them is the Xavier Initialization [38] which balances between all the layers to have the same activation variance and the same gradient variance. More formally the weights are initialized as:

$$\boldsymbol{W}_{i,j} \sim \text{Uniform}\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right), \tag{20}$$

where m is the number of inputs and n the number of outputs of matrix $\boldsymbol{W}$. Moreover, the biases $\boldsymbol{b}$ are initialized to 0.

Drop-out There are other techniques to prevent overfitting, such as drop-out [39], which involves randomly destroying neurons during the training process, thereby reducing the complexity of

Fig. 9 Examples of data transformations applied in the MNIST dataset. Each of these generated samples is considered additional training data

the model. Drop-out is an ensemble method that does not need to build the models explicitly. In practice, at each optimization iteration, random binary masks on the units are considered. The probability of removing a unit (p) is defined as a hyperparameter during the training of the network. During inference, all the units are activated; however, the obtained parameters $\boldsymbol{W}$ are multiplied with this probability p. Drop-out is quite efficient and commonly used in a variety of neural network architectures.

Data Augmentation Since neural networks are data-driven methods, their performance depends on the training data. To increase the amount of data during the training, data augmentation can be performed. It generates slightly modified copies of the existing training data to enrich the training samples. This technique acts as a regularizer and helps reduce overfitting. Some of the most commonly used transformations applied during data augmentation include random rotations, translations, cropping, color jittering, resizing, Gaussian blurring, and many more. In Fig. 9, examples of different transformations on different digits (first column) of the MNIST dataset [40] are presented. For medical images, the TorchIO library allows to easily perform data augmentation [41].

Batch Normalization To ensure that the training of the networks will be more stable and faster, batch normalization has been proposed [42]. In practice, batch normalization re-centers and re-scales the layer's input, mitigating the problem of internal

covariate shift which changes the distribution of the inputs of each layer affecting the learning rate of the network. Even if the method is quite popular, its necessity and use for the training have recently been questioned [43].

3.4 State-of-the-Art Optimizers

Over the years, different optimizers have been proposed and widely used, aiming to provide improvements over the classical stochastic gradient descent. These algorithms are motivated by challenges that need to be addressed with stochastic gradient descent and are focusing on the choice of the proper learning rate, its dynamic change during training, as well as the fact that it is the same for all the parameter updates [44]. Moreover, a proper choice of optimizer could speed up the convergence to the optimal solution. In this subsection, we will discuss some of the most commonly used optimizers nowadays.

3.4.1 Stochastic Gradient Descent with Momentum

One of the limitations of the stochastic gradient descent is that since the direction of the gradient that we are taking is random, it can heavily oscillate, making the training slower and even getting stuck in a saddle point. To deal with this problem, stochastic gradient descent with momentum [45, 46] keeps a history of the previous gradients, and it updates the weights taking into account the previous updates. More formally:

$$\begin{aligned} g^t &\leftarrow \rho g^{t-1} + (1-\rho) G(W^t) \\ \Delta W^t &\leftarrow -\eta_t g^t \\ W^{t+1} &\leftarrow W^t + \Delta W^t \end{aligned} , \tag{21}$$

where g^t is the direction of the update of the weights in time-step t and $\rho \in [0, 1]$ is a hyperparameter that controls the contribution of the previous gradients and current gradient in the current update. When $\rho = 0$, it is the same as the classical stochastic gradient descent. A large value of ρ will mean that the update is strongly influenced by the previous updates.

The momentum algorithm accumulates an exponentially decaying moving average of the past gradients and continues to move in their direction [24]. Momentum increases the speed of convergence, while it is also helpful to not get stuck in places where the search space is flat (saddle points with zero gradient), since the momentum will pursue the search in the same direction as before the flat region.

3.4.2 AdaGrad

To facilitate and speed up, even more, the training process, optimizers with adaptive learning rates per parameter have been proposed. The adaptive gradient (AdaGrad) optimizer [47] is one of them. It updates each individual parameter proportionally to their component (and momentum) in the gradient. More formally:

$$\begin{aligned} g^t &\leftarrow G(W^t) \\ r^t &\leftarrow r^{t-1} + g^t \odot g^t \\ \Delta W^t &\leftarrow -\frac{\eta}{\delta + \sqrt{r^t}} \odot g^t \\ W^{t+1} &\leftarrow W^t + \Delta W^t \end{aligned}, \tag{22}$$

where g^t is the gradient estimate vector in time-step t, r^t is the term controlling the per parameter update, and δ is some small quantity that is used to avoid the division by zero. Note that r^t constitutes of the gradient's element-wise product with itself and of the previous term r^{t-1} accumulating the gradients of the previous terms.

This algorithm performs very well for sparse data since it decreases the learning rate faster for the parameters that are more frequent and slower for the infrequent parameters. However, since the update accumulates gradients of the previous steps, the updates could decrease very fast, blocking the learning process. This limitation is mitigated by extensions of the AdaGrad algorithm as we discuss in the next sections.

3.4.3 RMSProp

Another algorithm with adaptive learning rates per parameter is the root mean squared propagation (RMSProp) algorithm, proposed by Geoffrey Hinton. Despite its popularity and use, this algorithm has not been published. RMSProp is an extension of the AdaGrad algorithm dealing with the problem of radically diminishing learning rates by being less influenced by the first iterations of the algorithm. More formally:

$$\begin{aligned} g^t &\leftarrow G(W^t) \\ r^t &\leftarrow \rho r^{t-1} + (1-\rho) g^t \odot g^t \\ \Delta W^t &\leftarrow -\frac{\eta}{\delta + \sqrt{r^t}} \odot g^t \\ W^{t+1} &\leftarrow W^t + \Delta W^t \end{aligned}, \tag{23}$$

where ρ is a hyperparameter that controls the contribution of the previous gradients and the current gradient in the current update. Note that RMSProp estimates the squared gradients in the same way as AdaGrad, but instead of letting that estimate continually accumulate over training, we keep a moving average of it, integrating the momentum. Empirically, RMSProp has been shown to be an effective and practical optimization algorithm for deep neural networks [24].

3.4.4 Adam

The effectiveness and advantages of the AdaGrad and RMSProp algorithms are combined in the adaptive moment estimation (Adam) optimizer [48]. The method computes individual adaptive learning rates for different parameters from estimates of the first and second moments of the gradients. More formally:

$$\begin{aligned} g^t &\leftarrow G(W^t) \\ s^t &\leftarrow \rho_1 s^{t-1} + (1-\rho_1) g^t \\ r^t &\leftarrow \rho_2 r^{t-1} + (1-\rho_2) g^t \odot g^t \\ \hat{s}^t &\leftarrow \frac{s^t}{1-(\rho_1)^t} \\ \hat{r}^t &\leftarrow \frac{r^t}{1-(\rho_2)^t} \\ \Delta W^t &\leftarrow -\frac{\lambda}{\delta + \sqrt{\hat{r}^t}} \odot \hat{s}^t \\ W^{t+1} &\leftarrow W^t + \Delta W^t \end{aligned} , \tag{24}$$

where s^t is the gradient with momentum, r^t accumulates the squared gradients with momentum as in RMSProp, and $\hat{s}^t$ and $\hat{r}^t$ are smaller than s^t and r^t, respectively, but they converge toward them. Moreover, δ is some small quantity that is used to avoid the division by zero, while ρ_1 and ρ_2 are hyperparameters of the algorithm. The parameters ρ_1 and ρ_2 control the decay rates of each moving average, respectively, and their value is close to 1. Empirical results demonstrate that Adam works well in practice and compares favorably to other stochastic optimization methods, making it the go-to optimizer for deep learning problems.

3.4.5 Other Optimizers

The development of efficient (in terms of speed and stability) optimizers is still an active research direction. RAdam [49] is a variant of Adam, introducing a term to rectify the variance of the adaptive learning rate. In particular, RAdam leverages a dynamic rectifier to adjust the adaptive momentum of Adam based on the variance and effectively provides an automated warm-up custom-tailored to the current dataset to ensure a solid start to training. Moreover, LookAhead [50] was inspired by recent advances in the understanding of loss surfaces of deep neural networks and provides a breakthrough in robust and stable exploration during the entirety of the training. Intuitively, the algorithm chooses a search direction by looking ahead at the sequence of fast weights generated by another optimizer. These are only some of the optimizers that exist in the literature, and depending on the problem and the application, different optimizers could be selected and applied.

4 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a specific category of deep neural networks that employ the convolution operation in order to process the input data. Even though the main concept dates back to the 1990s and is greatly inspired by neuroscience [51] (in particular by the organization of the visual cortex), their widespread use is due to a relatively recent success on the ImageNet Large Scale Visual Recognition Challenge of 2012 [27]. In contrast

to the deep fully connected networks that have been already discussed, CNNs excel in processing data with a spatial or grid-like organization (e.g., time series, images, videos, etc.) while at the same time decreasing the number of trainable parameters due to their weight sharing properties. The rest of this section is first introducing the convolution operation and the motivation behind using it as a building block/module of neural networks. Then, a number of different variations are presented together with examples of the most important CNN architectures. Lastly, the importance of the receptive field – a central property of such networks – will be discussed.

4.1 The Convolution Operation

The convolution operation is defined as the integral of the product of the two functions (f, g)[5] after one is reversed and shifted over the other function. Formally, we write:

$$h(t) = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)\ d\tau. \tag{25}$$

Such an operation can also be denoted with an asterisk $(*)$, so it is written as:

$$h(t) = (f * g)(t). \tag{26}$$

In essence, the convolution operation shows how one function affects the other. This intuition arises from the signal processing domain, where it is typically important to know how a signal will be affected by a filter. For example, consider a uni-dimensional continuous signal, like the brain activity of a patient on some electroencephalography electrode, and a Gaussian filter. The result of the convolution operation between these two functions will output the effect of a Gaussian filter on this signal which will, in fact, be a smoothed version of the input.

A different way to think of the convolution operation is that it shows how the two functions are related. In other words, it shows how similar or dissimilar the two functions are at different relative positions. In fact, the convolution operation is very similar to the cross-correlation operation, with the subtle difference being that in the convolution operation, one of the two functions is inverted. In the context of deep learning specifically, the exact differences between the two operations can be of secondary concern; however, the convolution operation has more properties than correlation, such as commutativity. Note also that when the signals are symmetric, both operations will yield the same result.

In order to deal with discrete and finite signals, we can expand the definition of the convolution operation. Specifically, given two

[5] Note that f and g have no relationship to their previous definitions in the chapter. In particular, f is not the deep learning model.

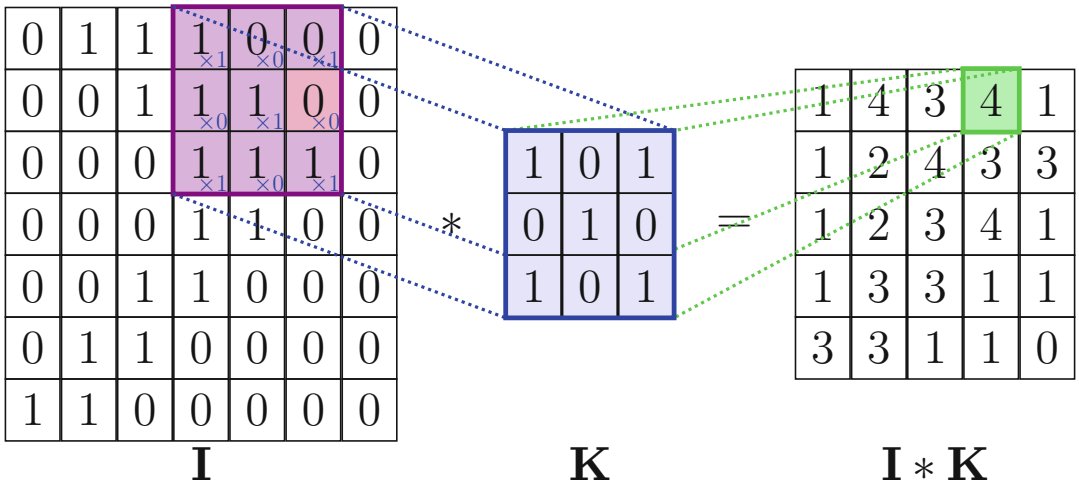

Fig. 10 A visualization of the discrete convolution operation in 2D

discrete signals $f[k]$ and $g[k]$, with $k \in \mathbb{Z}$, the convolution operation is defined by:

$$h[k] = \sum_n f[k-n]g[n]. \tag{27}$$

Lastly, the convolution operation can be extended for multidimensional signals similarly. For example, we can write the convolution operation between two discrete and finite two-dimensional signals (e.g., $I[i,j]$, $K[i,j]$) as:

$$H[i,j] = \sum_m \sum_n I[i-m, j-n]K[m,n]. \tag{28}$$

Very often, the first signal will be the input of interest (e.g., a large size image), while the second signal will be of relatively small size (e.g., a 3×3 or 4×4 matrix) and will implement a specific operation. The second signal is then called a kernel. In Fig. 10, a visualization of the convolution operation is shown in the case of a 2D discrete signal such as an image and a 3×3 kernel. In detail, the convolution kernel is shifted over all locations of the input, and an element-wise multiplication and a summation are utilized to calculate the convolution output at the corresponding location. Examples of applications of convolutions to an image are provided in Fig. 11. Finally, note that, as in multilayer perceptrons, a convolution will generally be followed by a non-linear activation function, for instance, a ReLU (*see* Fig. 12 for an example of activation applied to a feature map).

In the following sections of this chapter, any reference to the convolution operation will mostly refer to the 2D discrete case. The

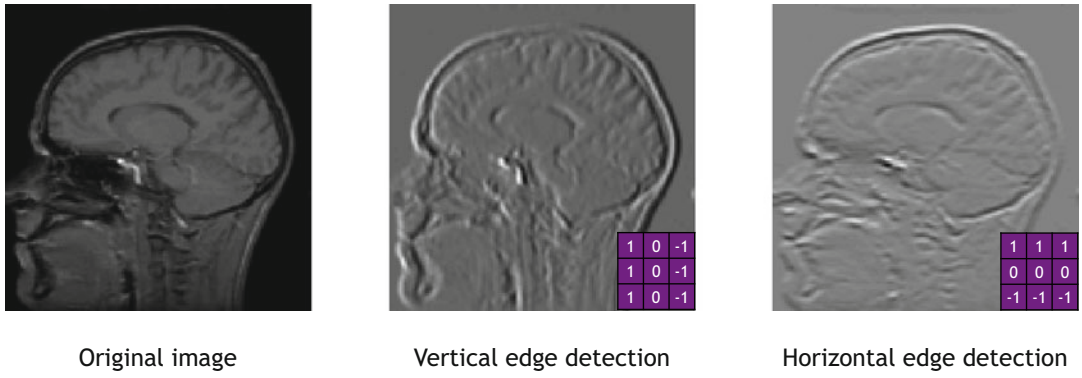

Fig. 11 Two examples of convolutions applied to an image. One of the filters acts as a vertical edge detector and the other one as a horizontal edge detector. Of course, in CNNs, the filters are learned, not predefined, so there is no guarantee that, among the learned filters, there will be a vertical/horizontal case detector, although it will often be the case in practice, especially for the first layers of the architecture

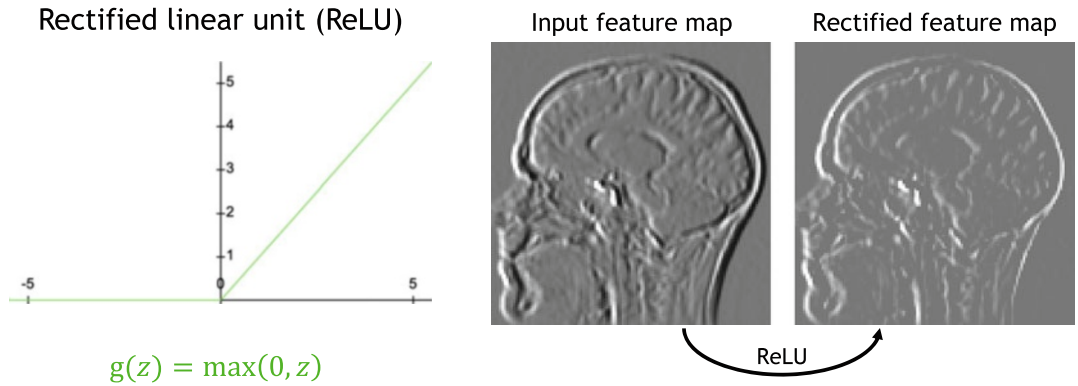

Fig. 12 Example of application of a non-linear activation function (here a ReLU) to an image

extension to the 3D case, which is often encountered in medical imaging, is straightforward.

4.2 Properties of the Convolution Operation

In the case of a discrete domain, the convolution operation can be performed using a simple matrix multiplication without the need of shifting one signal over the other one. This can be essentially achieved by utilizing the Toeplitz matrix transformation. The Toeplitz transformation creates a sparse matrix with repeated elements which, when multiplied with the input signal, produces the convolution result. To illustrate how the convolution operation can be implemented as a matrix multiplication, let's take the example of a 3×3 kernel (K) and a 4×4 input (I):

$$K = \begin{bmatrix} k_{00} & k_{01} & k_{02} \\ k_{10} & k_{11} & k_{12} \\ k_{20} & k_{21} & k_{22} \end{bmatrix} \quad \text{and} \quad I = \begin{bmatrix} i_{00} & i_{01} & i_{02} & i_{03} \\ i_{10} & i_{11} & i_{12} & i_{13} \\ i_{20} & i_{21} & i_{22} & i_{23} \\ i_{30} & i_{31} & i_{32} & i_{33} \end{bmatrix}.$$

Then, the convolution operation can be computed as a matrix multiplication between the Toepliz transformed kernel:

$$K = \begin{bmatrix} k_{00} & k_{01} & k_{02} & 0 & k_{10} & k_{11} & k_{12} & 0 & k_{20} & k_{21} & k_{22} & 0 & 0 & 0 & 0 & 0 \\ 0 & k_{00} & k_{01} & k_{02} & 0 & k_{10} & k_{11} & k_{12} & 0 & k_{20} & k_{21} & k_{22} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & k_{00} & k_{01} & k_{02} & 0 & k_{10} & k_{11} & k_{12} & 0 & k_{20} & k_{21} & k_{22} & 0 \\ 0 & 0 & 0 & 0 & 0 & k_{00} & k_{01} & k_{02} & 0 & k_{10} & k_{11} & k_{12} & 0 & k_{20} & k_{21} & k_{22} \end{bmatrix}$$

and a reshaped input:

$$I = [\, i_{00} \quad i_{01} \quad i_{02} \quad i_{03} \quad i_{10} \quad i_{11} \quad i_{12} \quad i_{13} \quad i_{20} \quad i_{21} \quad i_{22} \quad i_{23} \quad i_{30} \quad i_{31} \quad i_{32} \quad i_{33} \,]^{\top}.$$

The produced output will need to be reshaped as a 2×2 matrix in order to retrieve the convolution output. This matrix multiplication implementation is quite illuminating on a few of the most important properties of the convolution operation. These properties are the main motivation behind using such elements in deep neural networks.

By transforming the convolution operation to a matrix multiplication operation, it is evident that it can fit in the formalization of the linear functions, which has already been presented in Subheading 2.3. As such, deep neural networks can be designed in a way to utilize trainable convolution kernels. In practice, multiple convolution kernels are learned at each convolutional block, while several of these trainable convolutional blocks are stacked on top of each other forming deep CNNs. Typically, the output of a convolution operation is called a *feature map* or just *features*.

Another important aspect of the convolution operation is that it requires much fewer parameters than the fully connected MLP-based deep neural networks. As it can also be seen from the K matrix, the exact same parameters are shared across all locations. Eventually, rather than learning a different set of parameters for the different locations of the input, only one set is learned. This is referred to as *parameter sharing* or *weight sharing* and can greatly decrease the amount of memory that is required to store the network parameters. An illustration of the process of *weight sharing* across locations, together with the fact that multiple filters (resulting in multiple feature maps) are computed for a given layer, is illustrated in Fig. 13. The multiple feature maps for a given layer are stored using another dimension (*see* Fig. 14), thus resulting in a 3D

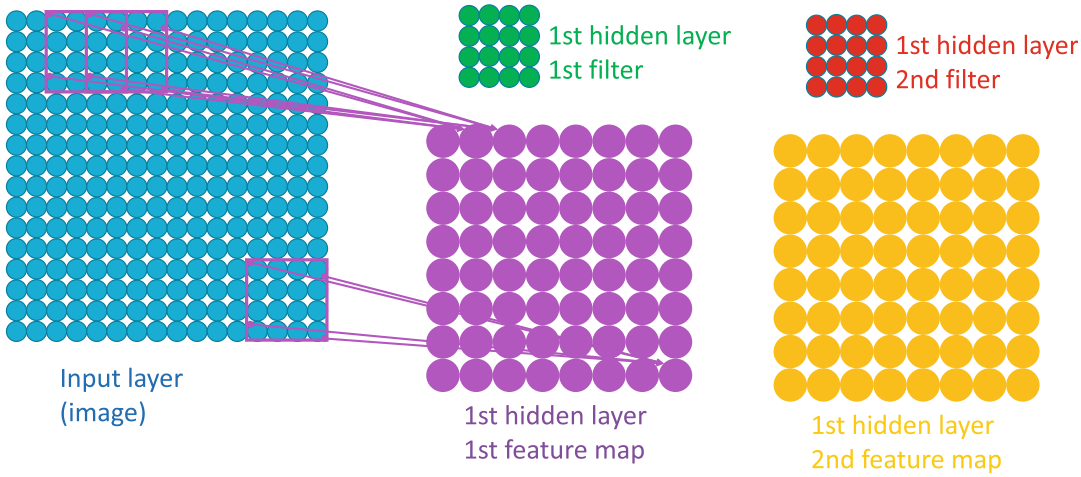

Fig. 13 For a given layer, several (usually many) filters are learned, each of them being able to detect a specific characteristic in the image, resulting in several feature/filter maps. On the other hand, for a given filter, the weights are shared across all the locations of the image

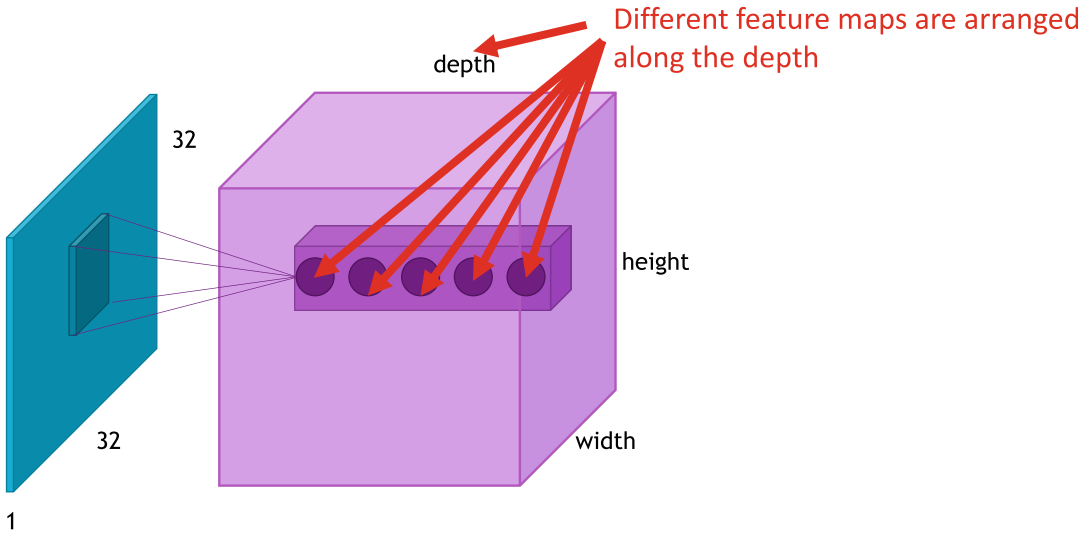

Fig. 14 The different feature maps for a given layer are arranged along another dimension. The feature maps will thus be a 3D array when the input is a 2D image (and a 4D array when the input is a 3D image)

array when the input is a 2D image (and a 4D array when the input is a 3D image).

Convolutional neural networks have proven quite powerful in processing data with spatial structure (e.g., images, videos, etc.). This is effectively based on the fact that there is a local connectivity of the kernel elements while at the same time the same kernel is applied at different locations of the input. Such processing grants a quite useful property called *translation equivariance* enabling the

network to output similar responses at different locations of the input. An example of the usefulness of such a property can be identified on an image detection task. Specifically, when training a network to detect tumors in an MR image of the brain, the model should respond similarly regardless of the location where the anomaly can be manifested.

Lastly, another important property of the convolution operation is that it decouples the size of the input with the trainable parameters. For example, in the case of MLPs, the size of the weight matrix is a function of the dimension of the input. Specifically, a densely connected layer that maps 256 features to 10 outputs would have a size of $\boldsymbol{W} \in \mathbb{R}^{10 \times 256}$. On the contrary, in convolutional layers, the number of trainable parameters only depends on the kernel size and the number of kernels that a layer has. This eventually allows the processing of arbitrarily sized inputs, for example, in the case of fully convolutional networks.

4.3 Functions and Variants

An observant reader might have noticed that the convolution operation can change the dimensionality of the produced output. In the example visualized in Fig. 10, the image of size 7×7, when convolved with a kernel of size 3×3, produces a feature map of size of 5×5. Even though dimension changes can be avoided with appropriate padding (*see* Fig. 15 for an illustration of this process) prior to the convolution operation, in some cases, it is actually desired to reduce the dimensions of the input. Such a decrease can be achieved in a number of ways depending on the task at hand. In this subsection, some of the most typical functions that are utilized in CNNs will be discussed.

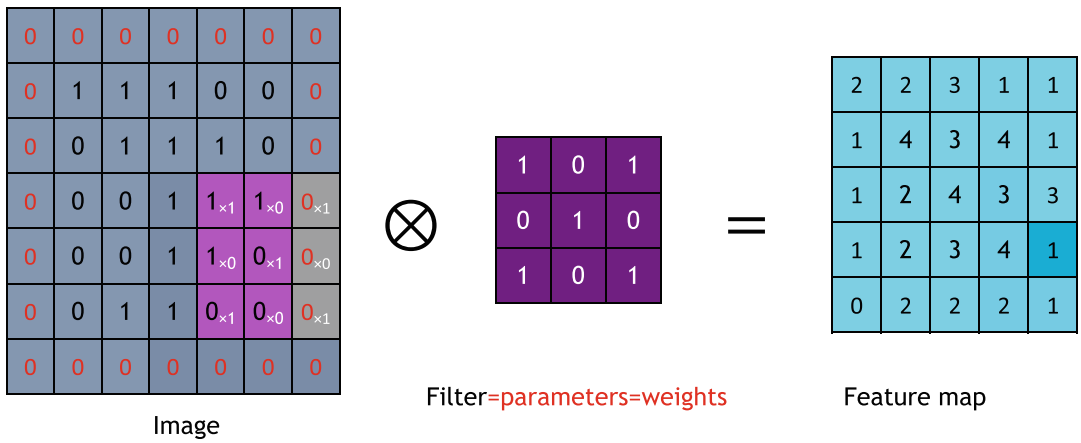

Fig. 15 The padding operation, which involves adding zeros around the image, allows to obtain feature maps that are of the same size as the original image

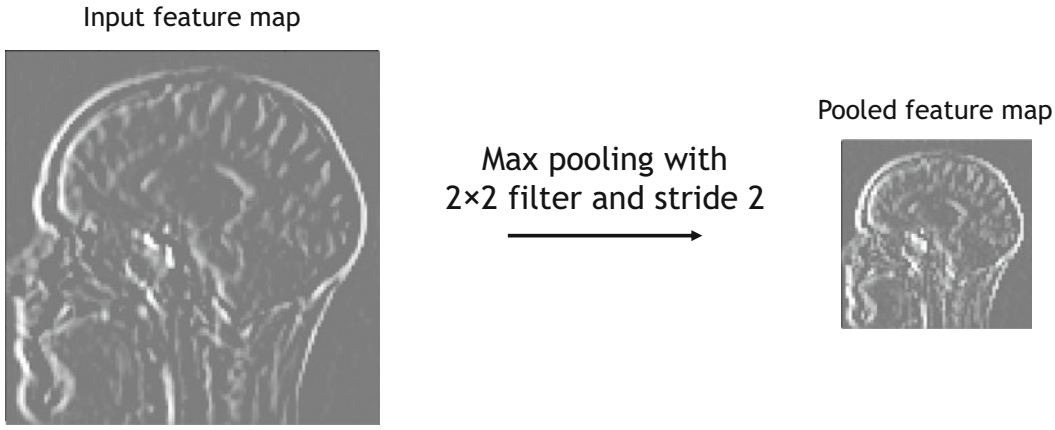

Fig. 16 Effect of a pooling operation. Here, a maximum pooling of size 2×2 with a stride of 2

Downsampling Operations (i.e., Pooling Layers) In many CNN architectures, there is an extensive use of downsampling operations that aim to compress the size of the feature maps and decrease the computational burden. Otherwise referred to as pooling layers, these processing operations are aggregating the values of their input depending on their design. Some of the most common downsampling layers are the *maximum pooling*, *average pooling*, or *global average pooling*. In the first two, either the maximum or the average value is used as a feature for the output across non-overlapping regions of a predefined pooling size. In the case of the global average pooling, the spatial dimensions are all represented with the average value. An example of pooling is provided in Fig. 16.

Strided Convolution The strided convolution refers to the specific case in which, instead of applying the convolution operation for every location using a step size (or stride, s) of 1, different step sizes can be considered (Fig. 17). Such an operation will produce a convolution output with much fewer elements. Convolutional blocks with $s > 1$ can be found on CNN architectures as a way to decrease the feature sizes in intermediate layers.

Atrous or Dilated Convolution Dilated, also called atrous, convolution is the convolution with kernels that have been dilated by inserting zero holes (*à trous* in French) between the non-zero values of a kernel. In this case, an additional parameter (d) of the convolution operation is added, and it is changing the distance between the kernel elements. In essence, it is increasing the reach of the kernel but keeping the number of trainable parameters the same. For example, a dilated convolution with a kernel size of 3×3 and a dilation rate of $d = 2$ would be sparsely arranged on a 5×5 grid.

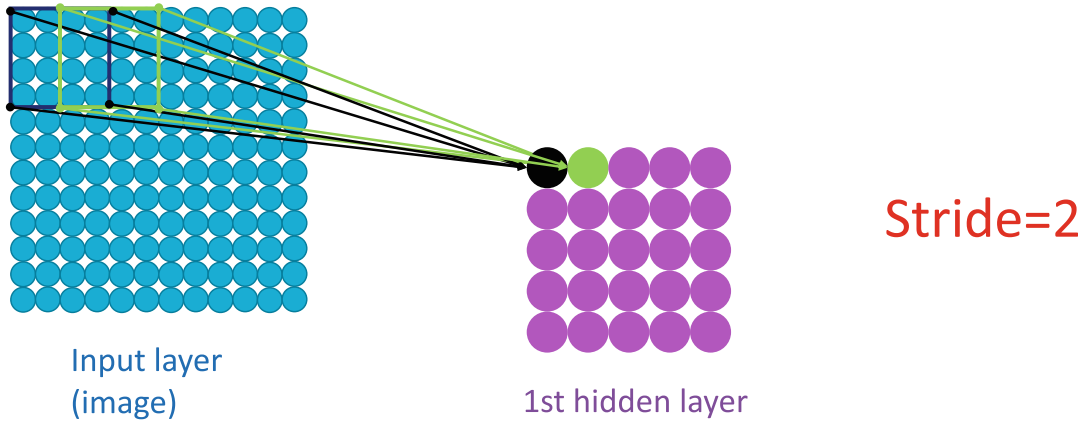

Fig. 17 Stride operation, here with a stride of 2

Transposed Convolution In certain circumstances, one needs not only to downsample the spatial dimensions of the input but also, usually at a later stage of the network, apply an upsample operation. The most emblematic case is the task of image segmentation (*see* Chap. 13), in which a pixel-level classification is expected, and therefore, the output of the neural network should have the same size as the input. In such cases, several upsampling operations are typically applied. The upsampling can be achieved by a transposed convolution operation that will eventually increase the size of the output. In details, the transposed convolution is performed by dilating the input instead of the kernel before applying a convolution operation. In this way, an input of size 5×5 will reach a size of 10×10 after being dilated with $d = 2$. With proper padding and using a kernel of size 3×3, the output will eventually double in size.

4.4 Receptive Field Calculation

In the context of deep neural networks and specifically CNNs, the term receptive field is used to define the proportion of the input that produces a specific feature. For example, a CNN that takes an image as input and applies only a single convolution operation with a kernel size of 3×3 would have a receptive field of 3×3. This means that for each pixel of the first feature map, a 3×3 region of the input would be considered. Now, if another layer were to be added, with again 3×3 size, then the receptive field of the new feature map with respect to the CNN's input would be 5×5. In other words, the proportion of the input that is used to calculate each element of the feature map of the second convolution layer increases.

Calculating the receptive field at different parts of a CNN is crucial when trying to understand the inner workings of a specific architecture. For instance, a CNN that is designed to take as an input an image of size 256×256 and that requires information

from all parts of it should have a receptive field close to the size of the input. The receptive field can be influenced by all the different convolution parameters and down-/upsampling operations described in the previous section. A comprehensive presentation of mathematical derivations for calculating receptive fields for CNNs is given in [52].

4.5 Classical Convolutional Neural Network Architectures

In the last decades, a variety of convolutional neural network architectures have been proposed. In this chapter, we cover only a few classical architectures for classification and regression. Note that classification and regression can usually be performed with the same architecture, just changing the loss function (e.g., cross-entropy for classification, mean squared error for regression). Architectures for other tasks can be found in other chapters.

A Basic CNN Architecture Let us start with the most simple CNN, which is actually very close to the original one proposed by LeCun et al. [53], sometimes called "LeNet." Such architecture is typically composed of two parts: the first one is based on convolution operations and learns the features for the image and the second part flattens the features and inputs them to a set of fully connected layers (in other words, a multilayer perceptron) for performing the classification/regression (*see* illustration in Fig. 18). Note that, of course, the whole network is trained end to end: the two parts are not trained independently. In the first part, one combines a series of blocks composed of a convolution operation (possibly strided and/or dilated), a non-linear activation function (for instance, a ReLU), and a pooling operation. It is often a good idea to include a drawing of the different layers of the chosen architecture.

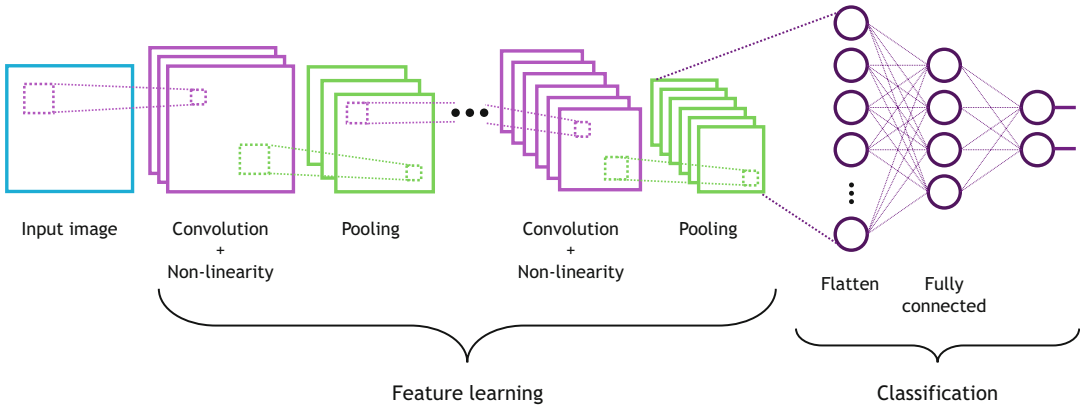

Fig. 18 A basic CNN architecture. Classically, it is composed of two main parts. The first one, using convolution operations, performs feature learning. The features are then flattened and fed into a set of fully connected layers (i.e., a multilayer perceptron), which performs the classification or the regression task

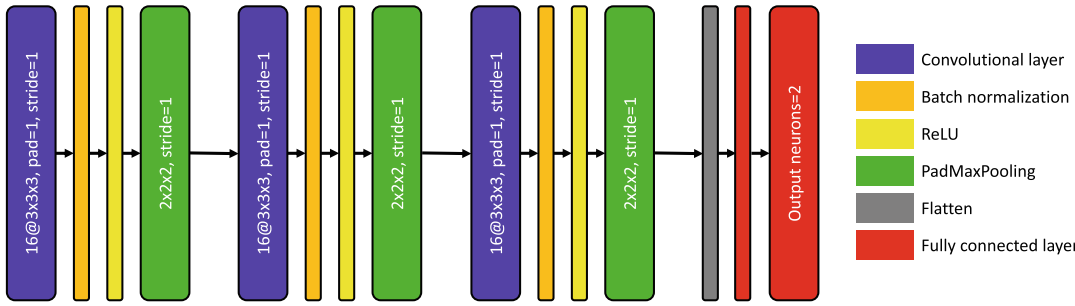

Fig. 19 A drawing describing a CNN architecture. Classically, it is composed of two main parts. Here $16@3 \times 3 \times 3$ means that 16 features with a $3 \times 3 \times 3$ convolution kernel will be computed. For the pooling operation, the kernel size is also mentioned (2×2). Finally, the stride is systematically indicated

Unfortunately, there is no harmonized format for such a description. An example is provided in Fig. 19.

One of the first CNN architectures that follow this paradigm is the AlexNet architecture [54]. AlexNet was one of the first papers that empirically indicated that the ReLU activation function makes the convergence of CNNs faster compared to other non-linearities such as the tanh. Moreover, it was the first architecture that achieved a top 5 error rate of 18.2% on the ImageNet dataset, outperforming all the other methods on this benchmark by a huge margin (about 10%). Prior to AlexNet, best-performing methods were using (very sophisticated) pre-extracted features and classical machine learning. After this advance, deep learning in general and CNNs, in particular, became very active research directions to address different computer vision problems. This resulted in the introduction of a variety of architectures such as VGG16 [55] that reported a 7.3% error rate on ImageNet, introducing some changes such as the use of smaller kernel filters. Following these advances, and even if there were a lot of different architectures proposed during that period, one could mention the Inception architecture [56], which was one of the deepest architectures of that period and which further reduced the error rate on ImageNet to 6.7%. One of the main characteristics of this architecture was the inception modules, which applied multiple kernel filters of different sizes at each level of the architecture. To solve the problem of vanishing gradients, the authors introduced auxiliary classifiers connected to intermediate layers, expecting to encourage discrimination in the lower stages in the classifier, increasing the gradient signal that gets propagated back, and providing additional regularization. During inference, these classifiers were completely discarded.

In the following section, some other recent and commonly used CNN architectures, especially for medical applications, will be presented.

ResNet One of the most commonly used CNN architectures, even today, is the ResNet [57]. ResNet reduced the error rate on ImageNet to 3.6%, while it was the first deep architecture that proposed novel concepts on how to gracefully go deeper than a few dozen of layers. In particular, the authors introduced a deep residual learning framework. The main idea of this residual learning is that instead of learning the desired underlying mapping of each network level, they learn the residual mapping. More formally, instead of learning the $H(x)$ mapping after the convolutional and non-linear layers, they fit another mapping of $F(x) = H(x) - x$ on which the original mapping is recast into $F(x) + x$. Feedforward neural networks can realize this mapping with "shortcut connections" by simply performing identity mapping, and their outputs are added to the outputs of the stacked layers. Such identity connections add neither additional complexity nor parameters to the network, making such architectures extremely powerful.

Different ResNet architectures have been proposed even in the original paper. Even though the depth of the network is increased with the additional convolutions, especially for the 152-layer ResNet (11.3 billion floating point operations), it still has lower complexity (i.e., fewer parameters) than VGG16/VGG19 networks. Currently, different layered-size ResNet architectures pre-trained on ImageNet are used as backbones for various problems and applications, including medical imaging. Pre-trained ResNet models, even if they are 2D architectures, are commonly used on histopathology [58, 59], chest X-ray [60], or even brain imaging [61, 62], while the way that such pre-trained networks work for medical applications gathered the attention of different studies such as [63]. However, it should be noted that networks pre-trained on ImageNet are not always efficient for medical imaging tasks, and there are cases where they perform poorly, much lower than simpler CNNs trained from scratch [64]. Nevertheless, a pre-trained ResNet is very often a good idea to use for a first try in a given application. Finally, there was an effort from the medical community to train 3D variations of ResNet architectures on a large amount of 3D medical data and release the pre-trained models. Such an effort is presented in [65] in which the authors trained and released different 3D ResNet architectures trained on different publicly available 3D datasets, including different anatomies such as the brain, prostate, liver, heart, and pancreas.

EfficientNet A more recent CNN architecture that is worth mentioning in this section is the recently presented EfficientNet [66]. EfficientNets are a family of neural networks that are balancing all dimensions of the network (width/depth/resolution) automatically. In particular, the authors propose a simple yet effective compound scaling method for obtaining these hyperpameters. In particular, the main compound coefficient ϕ uniformly scales

network width, depth, and resolution in a principled way: depth $= \alpha^{\phi}$, width $= \beta^{\phi}$, resolution $= \gamma^{\phi}$ s.t. $\alpha \cdot \beta^2 \cdot \gamma^2 \approx 2, \alpha \geq 1, \beta \geq 1, \gamma \geq 1$. In this formulation, the parameters α, β, γ are constants, and a small grid search can determine them. This grid search resulted in *eight* different architectures presented in the original paper. EfficientNet is used more and more for medical imaging tasks, as can be seen in multiple recent studies [67–69].

5 Autoencoders

An autoencoder is a type of neural network that can learn a compressed representation (called the latent space representation) of the training data. As opposed to the multilayer perceptrons and CNNs seen until now that are used for supervised learning, autoencoders have widely been used for unsupervised learning, with a wide range of applications. The architecture of autoencoders is composed of a contracting path (called the encoder), which will transform the input into a lower-dimensional representation, and an expanding path (called the decoder), which will aim at reconstructing the input as well as possible from the lower-dimensional representation (*see* Fig. 20).

The loss is usually the ℓ_2 loss and the cost function is then:

$$J(\boldsymbol{\theta}, \boldsymbol{\phi}) = \sum_{i=1}^{n} || \boldsymbol{x}^{(i)} - D_{\boldsymbol{\theta}}(E_{\boldsymbol{\phi}}(\boldsymbol{x}^{(i)}))||_2^2, \tag{29}$$

where $E_{\boldsymbol{\phi}}$ is the encoder (and $\boldsymbol{\phi}$ its parameters) and $D_{\boldsymbol{\theta}}$ is the decoder (and $\boldsymbol{\theta}$ its parameters). Note that, in Fig. 20, $D_{\boldsymbol{\theta}}(E_{\boldsymbol{\phi}}(\boldsymbol{x}))$ is denoted as $\hat{\boldsymbol{x}}$. More generally, one can write:

$$J(\boldsymbol{\theta}, \boldsymbol{\phi}) = \mathbb{E}_{\boldsymbol{x} \sim \mu_{\text{xref}}} \left[d\left(\boldsymbol{x}, D_{\boldsymbol{\theta}}(E_{\boldsymbol{\phi}}(\boldsymbol{x}))\right) \right], \tag{30}$$

where μ_{ref} is the reference distribution that one is trying to approximate and d is the reconstruction function. When μ_{ref} is the

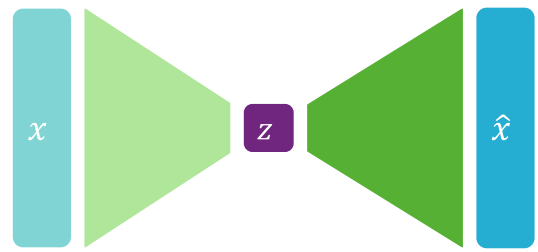

Fig. 20 The general principle of a denoising autoencoder. It aims at learning of a low-dimensional representation (latent space) $\boldsymbol{z}$ of the training data. The learning is done by aiming to provide a faithful reconstruction $\hat{x}$ of the input data $\hat{x}$

empirical distribution of the training set and d is the ℓ_2 norm, Eq. 30 is equivalent to Eq. 29.

Many variations of autoencoders exist, to prevent autoencoders from learning the identity function and to improve their ability to capture important information and learn richer representations. Among them, *sparse autoencoders* offer an alternative method for introducing an information bottleneck without requiring a reduction in the number of nodes at the hidden features. This is done by constructing the loss function such that it penalizes activations within a layer. This is achieved by enforcing sparsity in the network and encouraging it to learn an encoding and decoding which relies only on activating a small number of neurons. This sparsity is enforced in two main ways, an ℓ_1 regularization on the parameters of the network and a Kullback-Leibler divergence, which is a measure of the difference between two probability distributions. More information about sparse autoencoders could be found in [70]. Moreover, a quite common type of autoencoders is the *denoising autoencoders* [71], on which the model is tasked with reproducing the input as closely as possible while passing through some sort of information bottleneck (Fig. 20). This way, the model is not able to simply develop a mapping that memorizes the training data but rather learns a vector field for mapping the input data toward a lower-dimensional manifold. One should note here that the vector field is typically well-behaved in the regions where the model has observed data during training. In out-of-distribution data, the reconstruction error is both large and does not always point in the direction of the true distribution. This observation makes these networks quite popular for anomaly detection in medical data [72]. Additionally, *contractive autoencoders* [73] are other variants of this type of models, adding the contractive regularization loss to the standard autoencoder loss. Intuitively, it forces very similar inputs to have a similar encoding, and in particular, it requires the derivative of the hidden layer activations to be small with respect to small changes in the input. The denoising autoencoders can be understood as a variation of the contractive autoencoder. In the limit of small Gaussian noise, the denoising autoencoders make the reconstruction error resistant to finite-sized input perturbations, while the contractive autoencoders make the extracted features resistant to small input perturbations.

Depending on the input type, different autoencoder architectures could be designed. In particular, when the inputs are images, the encoder and the decoder are classically composed of convolutional blocks. The decoder uses, for instance, transposed convolutions to perform the expansion. Finally, the addition of skip connections has led to the U-Net [74] architectures that are commonly used for segmentation purposes. Segmentation architectures will be more extensively described in Chap. 13. Finally, variational autoencoders, which rely on a different mathematical formulation,

are not covered in the present chapter and are presented, together with other generative models, in Chap. 5.

6 Conclusion

Deep learning is a very fast evolving field, with numerous still unanswered theoretical questions. However, deep learning-based models have become the state-of-the-art methods for a variety of fields and tasks. In this chapter, we presented the basic principles of deep learning, covering both perceptrons and convolutional neural networks. All architectures were feedforward and recurrent networks are covered in Chap. 4. Generative adversarial networks are covered in Chap. 5, along with other generative models. Chapter 6 presents a recent class of deep learning methods, which does not use convolutions, and that are called transformers. Finally, throughout the other chapters of the book, different deep learning architectures are presented for various types of applications.

Acknowledgements

This work was supported in part by the French government under management of Agence Nationale de la Recherche as part of the "Investissements d'avenir" program, reference ANR-19-P3IA-0001 (PRAIRIE 3IA Institute), reference ANR-10-IAIHU-06 (Institut Hospitalo-Universitaire ICM), and ANR-21-CE45-0007 (Hagnodice).

References

1. Rosenblatt F (1957) The perceptron, a perceiving and recognizing automaton Project Para. Cornell Aeronautical Laboratory, Buffalo
2. Minsky M, Papert S (1969) Perceptron: an introduction to computational geometry. MIT Press, Cambridge, MA
3. Minsky ML, Papert SA (1988) Perceptrons: expanded edition. MIT Press, Cambridge, MA
4. Linnainmaa S (1976) Taylor expansion of the accumulated rounding error. BIT Numer Math 16(2):146–160
5. Werbos PJ (1982) Applications of advances in nonlinear sensitivity analysis. In: System modeling and optimization. Springer, Berlin, pp 762–770
6. Rumelhart DE, Hinton GE, Williams RJ (1986) Learning representations by back-propagating errors. Nature 323(6088): 533–536
7. Le Cun Y (1985) Une procédure d'apprentissage pour réseau à seuil assymétrique. Cognitiva 85:599–604
8. Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput 9(8): 1735–1780
9. Hinton GE, Osindero S, Teh YW (2006) A fast learning algorithm for deep belief nets. Neural Comput 18(7):1527–1554
10. Hinton GE (2007) Learning multiple layers of representation. Trends Cogn Sci 11(10): 428–434
11. Deng J, Dong W, Socher R, Li LJ, Li K, Fei-Fei L (2009) ImageNet: a large-scale hierarchical image database. In: 2009 IEEE conference on computer vision and pattern recognition. IEEE, pp 248–255
12. Bergstra J, Bastien F, Breuleux O, Lamblin P, Pascanu R, Delalleau O, Desjardins G, Warde-Farley D, Goodfellow I, Bergeron A et al

(2011) Theano: deep learning on GPUs with Python. In: NIPS 2011, Big learning workshop, Granada, Spain, vol 3. Citeseer, pp 1–48
13. Jia Y, Shelhamer E, Donahue J, Karayev S, Long J, Girshick R, Guadarrama S, Darrell T (2014) Caffe: convolutional architecture for fast feature embedding. In: Proceedings of the 22nd ACM international conference on Multimedia, pp 675–678
14. Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, Devin M et al (2016) TensorFlow: large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:160304467
15. Chollet F et al (2015) Keras. https://github.com/fchollet/keras
16. Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L et al (2019) PyTorch: an imperative style, high-performance deep learning library. In: Advances in neural information processing systems, vol 32
17. Hebb DO (1949) The organization of behavior: a psychological theory. Wiley, New York
18. Cybenko G (1989) Approximations by superpositions of a sigmoidal function. Math Control Signals Syst 2:183–192
19. Hornik K, Stinchcombe M, White H (1989) Multilayer feedforward networks are universal approximators. Neural Netw 2(5):359–366
20. Mhaskar HN (1996) Neural networks for optimal approximation of smooth and analytic functions. Neural Comput 8(1):164–177
21. Pinkus A (1999) Approximation theory of the MLP model in neural networks. Acta Numer 8: 143–195
22. Poggio T, Mhaskar H, Rosasco L, Miranda B, Liao Q (2017) Why and when can deep-but not shallow-networks avoid the curse of dimensionality: a review. Int J Autom Comput 14(5):503–519
23. Rolnick D, Tegmark M (2017) The power of deeper networks for expressing natural functions. arXiv preprint arXiv:170505502
24. Goodfellow I, Bengio Y, Courville A (2016) Deep learning. MIT Press, Cambridge, MA
25. Cover TM (1965) Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. IEEE Trans Electron Comput 3:326–334
26. Glorot X, Bordes A, Bengio Y (2011) Deep sparse rectifier neural networks. In: Proceedings of the fourteenth international conference on artificial intelligence and statistics, JMLR workshop and conference proceedings, pp 315–323
27. Krizhevsky A, Sutskever I, Hinton GE (2012) ImageNet classification with deep convolutional neural networks. In: Advances in neural information processing systems, vol 25
28. Hein M, Andriushchenko M, Bitterwolf J (2019) Why ReLU networks yield high-confidence predictions far away from the training data and how to mitigate the problem. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, pp 41–50
29. Maas AL, Hannun AY, Ng AY et al (2013) Rectifier nonlinearities improve neural network acoustic models. In: Proc. ICML, Atlanta, Georgia, vol 30. p 3
30. He K, Zhang X, Ren S, Sun J (2015) Delving deep into rectifiers: surpassing human-level performance on ImageNet classification. In: Proceedings of the IEEE international conference on computer vision, pp 1026–1034
31. Ramachandran P, Zoph B, Le QV (2017) Searching for activation functions. arXiv preprint arXiv:171005941
32. Dauphin YN, Pascanu R, Gulcehre C, Cho K, Ganguli S, Bengio Y (2014) Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In: Advances in neural information processing systems, vol 27
33. Bottou L (2010) Large-scale machine learning with stochastic gradient descent. In: Proceedings of COMPSTAT'2010. Springer, Berlin, pp 177–186
34. Allen-Zhu Z, Li Y, Song Z (2019) A convergence theory for deep learning via over-parameterization. In: International conference on machine learning, PMLR, pp 242–252
35. Baydin AG, Pearlmutter BA, Radul AA, Siskind JM (2018) Automatic differentiation in machine learning: a survey. J Mach Learn Res 18:1–43
36. Prechelt L (1998) Early stopping-but when? In: Neural networks: tricks of the trade. Springer, Berlin, pp 55–69
37. Reed R, MarksII RJ (1999) Neural smithing: supervised learning in feedforward artificial neural networks. MIT Press, Cambridge, MA
38. Glorot X, Bengio Y (2010) Understanding the difficulty of training deep feedforward neural networks. In: Proceedings of the thirteenth international conference on artificial intelligence and statistics, JMLR workshop and conference proceedings, pp 249–256
39. Srivastava N, Hinton G, Krizhevsky A, Sutskever I, Salakhutdinov R (2014) Dropout: a simple way to prevent neural networks from

overfitting. J Mach Learn Res 15(1): 1929–1958
40. Deng L (2012) The MNIST database of handwritten digit images for machine learning research. IEEE Signal Process Mag 29(6): 141–142
41. Pérez-García F, Sparks R, Ourselin S (2021) TorchIO: a Python library for efficient loading, preprocessing, augmentation and patch-based sampling of medical images in deep learning. Comput Methods Programs Biomed 208: 106236
42. Ioffe S, Szegedy C (2015) Batch normalization: accelerating deep network training by reducing internal covariate shift. In: International conference on machine learning, PMLR, pp 448–456
43. Brock A, De S, Smith SL, Simonyan K (2021) High-performance large-scale image recognition without normalization. In: International conference on machine learning, PMLR, pp 1059–1071
44. Ruder S (2016) An overview of gradient descent optimization algorithms. arXiv preprint arXiv:160904747
45. Polyak BT (1964) Some methods of speeding up the convergence of iteration methods. USSR Comput Math Math Phys 4(5):1–17
46. Qian N (1999) On the momentum term in gradient descent learning algorithms. Neural Netw 12(1):145–151
47. Duchi J, Hazan E, Singer Y (2011) Adaptive subgradient methods for online learning and stochastic optimization. J Mach Learn Res 12(7)
48. Kingma DP, Ba J (2014) Adam: a method for stochastic optimization. arXiv preprint arXiv:14126980
49. Liu L, Jiang H, He P, Chen W, Liu X, Gao J, Han J (2019) On the variance of the adaptive learning rate and beyond. arXiv preprint arXiv:190803265
50. Zhang M, Lucas J, Ba J, Hinton GE (2019) LookAhead optimizer: k steps forward, 1 step back. Adv Neural Inf Process Syst 32
51. Fukushima K, Miyake S (1982) Neocognitron: a self-organizing neural network model for a mechanism of visual pattern recognition. In: Competition and cooperation in neural nets. Springer, Berlin, pp 267–285
52. Araujo A, Norris W, Sim J (2019) Computing receptive fields of convolutional neural networks. Distill https://doi.org/10.23915/distill.00021
53. LeCun Y, Boser B, Denker JS, Henderson D, Howard RE, Hubbard W, Jackel LD (1989) Backpropagation applied to handwritten zip code recognition. Neural Comput 1(4): 541–551
54. Krizhevsky A, Sutskever I, Hinton GE (2012) ImageNet classification with deep convolutional neural networks. In: Pereira F, Burges C, Bottou L, Weinberger K (eds) Advances in neural information processing systems, vol 25. Curran Associates. https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf
55. Simonyan K, Zisserman A (2014) Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:14091556
56. Szegedy C, Liu W, Jia Y, Sermanet P, Reed S, Anguelov D, Erhan D, Vanhoucke V, Rabinovich A (2015) Going deeper with convolutions. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 1–9
57. He K, Zhang X, Ren S, Sun J (2016) Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 770–778
58. Lu MY, Williamson DF, Chen TY, Chen RJ, Barbieri M, Mahmood F (2021) Data-efficient and weakly supervised computational pathology on whole-slide images. Nat Biomed Eng 5(6):555–570
59. Benkirane H, Vakalopoulou M, Christodoulidis S, Garberis IJ, Michiels S, Cournède PH (2022) Hyper-AdaC: adaptive clustering-based hypergraph representation of whole slide images for survival analysis. In: Machine learning for health, PMLR, pp 405–418
60. Horry MJ, Chakraborty S, Paul M, Ulhaq A, Pradhan B, Saha M, Shukla N (2020) X-ray image based COVID-19 detection using pre-trained deep learning models. Engineering Archive, Menomonie
61. Li JP, Khan S, Alshara MA, Alotaibi RM, Mawuli C et al (2022) DACBT: deep learning approach for classification of brain tumors using MRI data in IoT healthcare environment. Sci Rep 12(1):1–14
62. Nandhini I, Manjula D, Sugumaran V (2022) Multi-class brain disease classification using modified pre-trained convolutional neural networks model with substantial data augmentation. J Med Imaging Health Inform 12(2): 168–183
63. Raghu M, Zhang C, Kleinberg J, Bengio S (2019) Transfusion: understanding transfer learning for medical imaging. In: Advances in neural information processing systems, vol 32

64. Wen J, Thibeau-Sutre E, Diaz-Melo M, Samper-González J, Routier A, Bottani S, Dormont D, Durrleman S, Burgos N, Colliot O (2020) Convolutional neural networks for classification of Alzheimer's disease: overview and reproducible evaluation. Med Image Anal 63:101694
65. Chen S, Ma K, Zheng Y (2019) Med3D: transfer learning for 3D medical image analysis. arXiv preprint arXiv:190400625
66. Tan M, Le Q (2019) EfficientNet: rethinking model scaling for convolutional neural networks. In: International conference on machine learning, PMLR, pp 6105–6114
67. Wang J, Liu Q, Xie H, Yang Z, Zhou H (2021) Boosted EfficientNet: detection of lymph node metastases in breast cancer using convolutional neural networks. Cancers 13(4):661
68. Oloko-Oba M, Viriri S (2021) Ensemble of EfficientNets for the diagnosis of tuberculosis. Comput Intell Neurosci 2021:9790894
69. Ali K, Shaikh ZA, Khan AA, Laghari AA (2021) Multiclass skin cancer classification using EfficientNets—a first step towards preventing skin cancer. Neurosci Inform 2(4):100034
70. Ng A et al (2011) Sparse autoencoder. CS294A Lecture Notes 72(2011):1–19
71. Vincent P, Larochelle H, Bengio Y, Manzagol PA (2008) Extracting and composing robust features with denoising autoencoders. In: Proceedings of the 25th international conference on machine learning, pp 1096–1103
72. Baur C, Denner S, Wiestler B, Navab N, Albarqouni S (2021) Autoencoders for unsupervised anomaly segmentation in brain MR images: a comparative study. Med Image Anal 69: 101952
73. Salah R, Vincent P, Muller X, et al (2011) Contractive auto-encoders: explicit invariance during feature extraction. In: Proceedings of the 28th international conference on machine learning, pp 833–840
74. Ronneberger O, Fischer P, Brox T (2015) U-net: convolutional networks for biomedical image segmentation. In: International Conference on Medical image computing and computer-assisted intervention. Springer, Berlin, pp 234–241