CHAPTER 20

Convolutional neural networks

Jonas Teuwen[a,b], **Nikita Moriakov**[a]
[a]Radboud University Medical Center, Department of Radiology and Nuclear Medicine, Nijmegen, the Netherlands
[b]Netherlands Cancer Institute, Department of Radiation Oncology, Amsterdam, the Netherlands

Contents

20.1. Introduction

Convolutional neural networks (CNNs) – or *convnets*, for short – have in recent years achieved results which were previously considered to be purely within the human realm. In this chapter we introduce CNNs, and for this we first consider regular neural networks, and how these methods are trained. After introducing the convolution, we introduce CNNs. They are very similar to the regular neural networks as they are also made up of neurons with learnable weights. But, in contrast to MLPs, CNNs make the explicit assumption that inputs have specific structure like images. This allows encoding this property into the architecture by sharing the weights for each location in the image and having neurons respond only locally.

Handbook of Medical Image Computing and Computer Assisted Intervention
https://doi.org/10.1016/B978-0-12-816176-0.00025-9

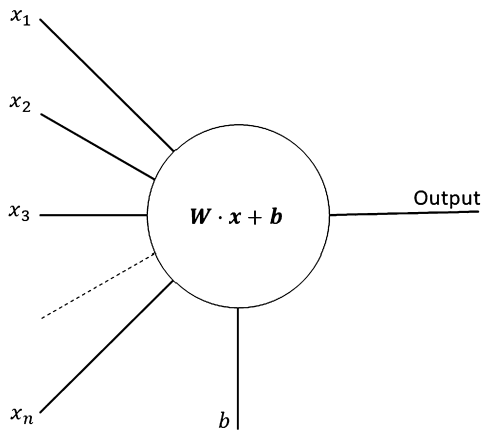


Figure 20.1 Schematic version of neuron.

20.2. Neural networks

To understand convolutional neural networks, we need to take one step back and first look into regular neural networks. Most concepts can readily be explained by using these simpler networks. The initial development of these networks originates in the work of Frank Rosenblatt on perceptrons and starts with the definition of a neuron. Mathematically, a neuron is a nonlinearity applied to an affine function. The input features $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ are passed through an affine function composed with a nonlinearity φ:

$$T(\boldsymbol{x}) = \varphi\Big(\sum_i W_i x_i + b\Big) = \varphi(\boldsymbol{W} \cdot \boldsymbol{x} + \boldsymbol{b}) \tag{20.1}$$

with given *weights* $\boldsymbol{W}$ and *bias* $\boldsymbol{b}$. Schematically this is represented in Fig. 20.1. A typical nonlinearity, or *activation function* is the *sigmoid* defined by

$$\sigma(x) = \frac{1}{1 + \mathrm{e}^{-x}}. \tag{20.2}$$

There are many choices for such nonlinearities, and different choices will be given when we discuss CNNs in Sect. 20.3.2.

Such a neural network can be modeled as a collection of neurons which are connected in an acyclic graph. That is, the output of some of the neurons become inputs to other neurons, and cycles where the output of a neuron maps back to an earlier intermediate input are forbidden. Commonly such neurons are organized in layers of neurons. Such a network consists of an input layer, one or more *hidden layers,* and an output layer. In contrast to the hidden layers, the output layer usually does not have

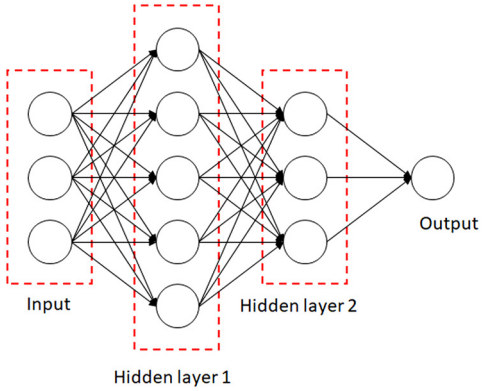


Figure 20.2 A 3-layer neural network with three inputs, two hidden layers of respectively 5 and 3 neurons, and one output layer. Notice that in both cases there are connections between neurons across layers, but not within a layer.

an activation function. Such networks are usually referred to as Multilinear Perceptron (MLP) or less commonly as Artificial Neural Network (ANN). If we want to be more explicit about the number of layers, we could refer to such a network as an N-layer network where N counts the number of layers, excluding the input layer. An example of this is given in Fig. 20.2. To use a neural network for prediction, we need to find the proper values for the parameters $(\boldsymbol{W}, \boldsymbol{b})$ and define a function to map the output of the neural network to a prediction; this could, for instance, be a class (i.e., malignant or benign) or a real value in the case of a regression problem. These parameters are the so-called *trainable parameters,* and the number of these parameters serves as a metric for the size (or capacity) of the neural network. In the example of Fig. 20.2, there are in total 8 neurons, where the hidden layers have $3 \cdot 5$ and $5 \cdot 3$ weights, and 5 and 3 biases, respectively. The output layer has 3 weights and 1 bias. In total this network has 27 learnable parameters. In modern neural network architectures, these numbers can run into the millions.

As mentioned, the output layer most commonly does not have an activation function because the output layer is often used to represent, for instance, class scores through a softmax function, which we will discuss in more detail below or some other real-valued target in the case of regression.

20.2.1 Loss function

Depending on a task, neural networks can be trained in a supervised or unsupervised way. For the type of tasks most frequently encountered in medical imaging, we are typically working with discriminative models, meaning that we have two spaces of objects X ("the inputs") and Y ("the labels") and we would like to learn a neural network f_θ

with parameters θ, the hypothesis, which outputs an object in Y given an object in X, or $f_\theta : X \to Y$. To do this, we have a training set of size m at our disposal $(\boldsymbol{X}_i, \boldsymbol{y}_i)_{i=1}^m$ where $\boldsymbol{X}_i$ is the input, for instance, an image, and $\boldsymbol{y}_i$ is the corresponding label. To put it differently, $\boldsymbol{y}_i$ is the response we expect from the model g_θ with $\boldsymbol{X}_i$ as input.

Putting this more formally, we assume there is a joint probability distribution $P(x, y)$ over $X \times Y$ and that the training set consists out of m samples $(\boldsymbol{X}_i, \boldsymbol{y}_i)_{i=1}^m$ independently drawn from $\mathbb{P}(x, y)$. In this formality, y is a random variable with conditional probability $\mathbb{P}(y \mid x)$. To formulate the training problem, we assume there is a nonnegative real-valued loss function L which can measure how far the prediction of the hypothesis $\boldsymbol{y}_{\text{pred}}$ from the real value $\boldsymbol{y}$ is. The risk associated with the hypothesis g_θ is defined as the expectation of the loss function

$$R(g_\theta) = \mathbb{E}_{(\boldsymbol{X},\boldsymbol{y})\sim\mathbb{P}(x,y)}\,[L(g_\theta(\boldsymbol{X}), \boldsymbol{y})]. \tag{20.3}$$

The goal of learning is to find parameters θ such that R is minimal, that is, we want to find θ^* such that $\theta^* = \arg\min_{\theta\in\mathbf{R}^m} R(g_\theta)$, where m is the number of parameters of the neural network.

In general, the risk $R(g_\theta)$ cannot be computed because the distribution $\mathbb{P}(x, y)$ is unknown. In this case we replace the expectation with the average over the training set to obtain the *empirical risk* as an approximation to the expectation:

$$R_{\text{emp}}(g_\theta) = \frac{1}{m}\sum_{i=1}^{m} L(g_\theta(\boldsymbol{X}_i), \boldsymbol{y}_i). \tag{20.4}$$

The process of minimizing the empirical risk is called *empirical risk minimization*.

20.2.2 Backpropagation

Nearly all neural networks encountered in practice are almost-everywhere differentiable with respect to parameters θ and are optimized with gradient-based optimization:

$$\nabla_\theta R_{\text{emp}}(g_\theta) = [\frac{\partial}{\partial\theta_1}, \frac{\partial}{\partial\theta_2}, \ldots, \frac{\partial}{\partial\theta_M}]R_{\text{emp}}(g_\theta). \tag{20.5}$$

Backpropagation is an algorithm that allows us to compute the gradients and apply gradient-based optimization schemes such as gradient descent, which is explained in more detail in Sect. 20.5.2.

The structure of a neural network allows for a very efficient computation of the gradient by a process called *backpropagation*, which can be readily understood by applying the chain rule.

For a bit more detail, consider a feedforward neural network, which accepts an input x which is propagated through the network to produce an output $\boldsymbol{y}$. This process

is the *forward propagation* step and results in a scalar loss R_{emp}. Computing the derivative for such a network is straightforward, but numerically evaluating such an expression is not only computationally expensive, it is also sensitive to numerical rounding errors, especially for deeper networks. Using the backpropagation algorithm, we can evaluate this derivative in a computationally efficient way with the additional advantage that the computations can be reused in subsequent steps. Once we have the gradient, we can apply an algorithm such as stochastic gradient descent (SGD) to compute updates to the network weights.

To be able to understand backpropagation properly, we introduce the *computation graph* language. A computation graph is a directed graph where on each node we have an *operation*, and an operation is a function of one or more variables and returns either a number, multiple numbers, or a tensor. The chain rule can now be used to compute the derivatives of functions formed by composing other functions with known derivatives. The backpropagation algorithms allows us to do this in a highly efficient manner.

Before we continue, let us recall the chain rule. For this let f and g be real-valued functions. Suppose additionally that $y = g(x)$ and $z = f(g(x)) = f(y)$ (i.e., two operations on the computation graph). The chain rule is now given by

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}.$$

The generalization to higher dimensions is trivial. Suppose now that $x \in \mathbf{R}^n$ and $y \in \mathbf{R}^m$, $g : \mathbf{R}^n \to \mathbf{R}^m$, and $f : \mathbf{R}^m \to \mathbf{R}$. Then if $y = g(x)$ and $z = f(y)$, we have

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}. \tag{20.6}$$

Using the chain rule (20.6), the back propagation algorithm is readily explained. Before we proceed, we fix some notation. Consider an MLP where the jth neuron in the ℓth layer has weighted output (i.e., before the activation function φ) z_j^ℓ and activation $a_j^\ell := \varphi(z_j^\ell)$. Similarly, the jth neuron in the layer ℓ weights output of the kth neuron in the $(\ell - 1)$th by w_{kj}^ℓ with corresponding bias b_j^ℓ.

As we intend to minimize the empirical risk function R_{emp} through gradient optimization, we are interested in the derivatives

$$\frac{\partial R_{\text{emp}}}{\partial w_{kj}^\ell} \text{ and } \frac{\partial R_{\text{emp}}}{\partial b_j^\ell}.$$

These derivatives can readily be computed using the chain rule (20.6):

$$\frac{\partial R_{\text{emp}}}{\partial w_{kj}^\ell} = \sum_k \frac{R_{\text{emp}}}{\partial z_k^\ell}\frac{\partial z_k^\ell}{\partial w_{kj}^\ell}.$$

As

$$z_k^\ell := \sum_j w_{kj}^\ell a_j^{\ell-1} + b_k^\ell, \tag{20.7}$$

the last derivative is equal to $a_j^{\ell-1}$ when $j = k$ and 0 otherwise, so,

$$\frac{\partial R_{\text{emp}}}{\partial w_{kj}^\ell} = \frac{\partial R_{\text{emp}}}{\partial z_k^\ell} a_j^{\ell-1}.$$

Using the chain rule again, we can see that

$$\frac{\partial R_{\text{emp}}}{\partial b_j^\ell} = \frac{\partial R_{\text{emp}}}{\partial z_j^\ell}.$$

Using these, we can see how the backpropagation rule works by efficiently computing $\varepsilon_j^\ell := \partial R_{\text{emp}}/\partial z_j^\ell$ where we will refer to ε_j^ℓ as the *error* of the jth node in the ℓth layer.

The backpropagation algorithm is an efficient way to compute the error ε^ℓ iteratively using error $\varepsilon^{\ell+1}$, so we proceed by computing the error from the last layer L, again, using the chain rule:

$$\varepsilon_j^L = \sum_k \frac{\partial R_{\text{emp}}}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L}, \tag{20.8}$$

and, when $k \neq L$, the terms vanish and we obtain:

$$\varepsilon_j^L = \frac{\partial R_{\text{emp}}}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial R_{\text{emp}}}{\partial a_j^L} \varphi'(z_j^L).$$

If we can derive a rule to compute ε_j^ℓ from $\varepsilon_j^{\ell+1}$ efficiently, we are done. This rule can be found, again, through the chain rule:

$$\begin{aligned}\varepsilon_j^\ell = \frac{\partial R_{\text{emp}}}{\partial z_j^\ell} &= \sum_k \frac{\partial R_{\text{emp}}}{\partial z_k^{\ell+1}} \frac{\partial z_k^{\ell+1}}{\partial z_j^\ell} \\ &= \sum_k \varepsilon_k^{\ell+1} \frac{\partial z_k^{\ell+1}}{\partial z_j^\ell}.\end{aligned}$$

Using (20.7) we can compute the last derivative as

$$\frac{\partial z_k^{\ell+1}}{\partial z_j^\ell} = w_{kj}^{\ell+1} \varphi'(z_j^\ell),$$

so, the backpropagation rule becomes

$$\varepsilon_j^\ell = \sum_k \varepsilon_j^{\ell+1} w_{kj}^{\ell+1} \varphi'(z_j^\ell). \tag{20.9}$$

In summary, to compute the derivatives of the empirical risk R_{emp} with respect to the weights and biases, it is sufficient to compute the error ε^ℓ. This can be done iteratively, by first computing the error for the final layer by (20.8) and then proceeding to the input by applying (20.9) for each layer consecutively.

20.3. Convolutional neural networks

Convolutional neural networks (CNNs), or *convnets* for short, are a special case of *feed-forward* neural networks. They are very similar to the neural networks presented above in the sense that they are made up of neurons with learnable weights and biases. The essential difference is that the CNN architecture makes the implicit assumption that the input are image-like, which allows us to encode certain properties in the architecture. In particular, convolutions capture *translation invariance* (i.e., filters are independent of the location).

This in turns makes the forward function more efficient, vastly reduces the number of parameters, and therefore makes the network easier to optimize and less dependent on the size of the data.

In contrast to regular neural networks, the layers of CNNs have neurons arranged in a few dimensions: channels, width, height, and number of filters in the simplest 2D case. A convolution neural network consists, just as an MLP, of a sequence of layers, where every layer transforms the activations or outputs of the previous layer through another differentiable function. There are several such layers employed in CNNs, and these will be explained in subsequent sections, however, the most common building blocks which you will encounter in most CNN architectures are: the convolution layer, pooling layer, and fully connected layers. In essence, these layers are like feature extractors, dimensionality reduction and classification layers, respectively. These layers of a CNN are stacked to form a full convolutional layer.

Before we proceed with an overview of the different layers, we pause a bit at the convolution layer. Essentially, a convolution layer uses a convolutional kernel as a filter for the input. Usually, there are many of such *filters*.

During a forward pass, a filter slides across the input volume and computes the *activation map* of the filter at that point by computing the pointwise product of each value and adding these to obtain the activation at the point. Such a sliding filter is naturally implemented by a *convolution* and, as this is a linear operator, it can be written as a dot-product for efficient implementation.

Intuitively, this means that when training such a CNN, the network will learn filters that capture some kind of visual information such as an edge, orientation, and eventually, in a higher layer of the network, entire patterns. In each such convolution layer, we have an entire set of such filters, each of which will produce a separate activation map. These activation maps are stacked to obtain the output map or *activation volume* of this layer.

20.3.1 Convolutions

Mathematically, the convolution $(x * w)(a)$ of functions x and w is defined in all dimensions as

$$(x * w)(a) = \int x(t)w(a-t)\,\mathrm{d}a, \tag{20.10}$$

where a is in $\mathbf{R}^n$ for any $n \geqslant 1$, and the integral is replaced by its higher-dimensional variant. To understand the idea behind convolutions, it is interesting to pick the Gaussian function $w(a) = \exp(-x^2)$ as an example. If we were taking a photo with a camera and shaking the camera a bit, the blurry picture would be the real picture x convolved with a Gaussian function w.

In the terminology of convolutional neural networks, x is called the *input*, w is called the *filter* or *kernel*, and the output is often referred to as *activation, or feature map*.

Note that we modeled the input and kernel in (20.10) as a continuous function. Due to the discrete nature of image sensors, this will not be the case in practice and it is more realistic to assume that parameter t is *discrete*. If we assume that this is the case, then we can define the discrete convolution

$$(x * w)(a) = \sum_a x(t)w(t-a), \tag{20.11}$$

where a runs over all values in the space, and can be in any dimension. In deep learning, usually x is a multidimensional array of data and the kernel w involves learnable parameters and usually has finite support, that is, there are only finitely many values a for which $w(a)$ is nonzero. This means that we can implement (20.11) as a finite summation. The definition of (20.11) is independent of dimension, but in medical imaging we will mainly be working with 2- or 3-dimensional convolutions:

$$(I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n), \tag{20.12}$$

or

$$(I * K)(i,j,k) = \sum_m \sum_n \sum_\ell I(m,n,\ell)K(i-m, j-n, k-\ell). \tag{20.13}$$

The convolutions (20.10) and (20.11) are commutative, which means that $I * K = K * I$, so that we can also write (20.12) as

$$(I * K)(i, j) = \sum_m \sum_n I(i - m, j - n)K(m, n). \tag{20.14}$$

As K has finite support, this a priori infinite sum becomes finite. Some neural network libraries also implement an operation called the *cross-correlation,* but from a deep learning perspective these operations are equivalent, as one weight set can be directly translated into the other.

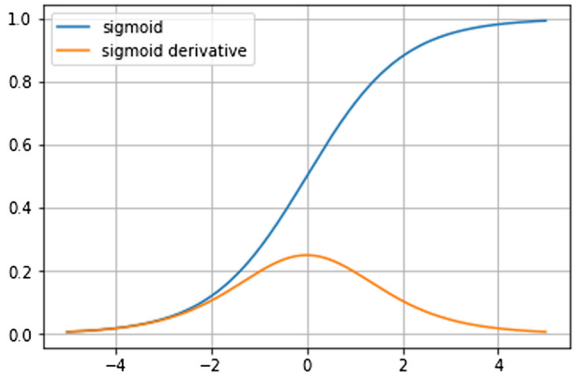
Convolutions as an infinitely strong priors

As a convolution is a linear transformation, it can be written in the form of $\boldsymbol{w} \cdot \boldsymbol{x} + \boldsymbol{b}$ and therefore as a fully connected layer. However, as the kernels are often much smaller than the input, only a small number of inputs will interact with the output (the so-called *receptive field*), and the weight tensor $\boldsymbol{w}$ will be very *sparse*. Additionally, the weight tensor will contain many similar elements, caused by the fact that the kernel is applied to every location in the input. This effect is referred to as *weight sharing*, and, together with the sparsity, this not only means that we need to store fewer parameters, which improves both the memory requirements and statistical efficiency, but additionally puts a prior on the weights: we implicitly assume that a filter, such as an edge filter, can be relevant to every part of the image and that most interactions between pixels are local. For most images, this is definitely a reasonable assumption, but this can break down in the case of other type of image data such a CT sinograms or MRI k-space where local information in the imaging domain can translate to global information in the acquisition space. Sometimes, we refer to this by saying that a convolution is an *infinitely strong prior* in contrast to weaker priors such as ℓ^p-regularization discussed below.

Much research has gone into adapting convolutions and imposing new strong priors, for instance, the *group convolution* [1] additionally enforces a certain symmetry group to hold for the image. Further discussion of this topic is beyond the scope of this chapter.

Equivariance

Next to sparsity and weight sharing, convolutions put another prior on the kernel weights in the form of translation equivariance, which to a translation (for instance, shifting) of the convolution means that if we apply a translation to the image, and then apply convolutions, we obtain the same result as first applying the convolution and then translating the feature map. More specifically, an operator T is said to be equivariant with respect to f if for each x we have $T(f(x)) = f(T(x))$. Translation equivariance is a sensible assumption for images, as the features to detect an object in the image should only depend on the object itself and not on its precise location.



20.3.2 Nonlinearities

Nonlinearities are essential for neural network design: without nonlinearities a neural network would compute a linear function of its input, which is too restrictive. The choice of a nonlinearity can have a large impact on the training speed of a neural network.

sigmoid This nonlinearity is defined as

$$\sigma(x) = \frac{1}{1 + \mathrm{e}^{-x}}, \quad x \in \mathbb{R}. \tag{20.15}$$

It is easy to show that $\sigma(x) \in (0, 1)$ for all $x \in \mathbb{R}$. Furthermore, σ is monotone increasing, $\lim_{x \to \infty} \sigma(x) = 1$ and $\lim_{x \to -\infty} \sigma(x) = 0$.

This makes the sigmoid nonlinearity suitable when the goal is to produce outputs contained in the $[0, 1]$ range, such as probabilities or normalized images. One can also show that $\lim_{x \to \infty} \sigma'(x) = \lim_{x \to -\infty} \sigma'(x) = 0$. This fact implies that the sigmoid nonlinearity may lead to *vanishing gradients*: when the input x to the sigmoid is far from zero, the neuron will saturate and the gradient of $\sigma(x)$ with respect to x will be close to zero, which will make successive optimization hard. This is the reason why sigmoid nonlinearities are rarely used in the intermediate layers of CNNs.

tanh The *tanh* nonlinearity is defined as

$$\tanh(x) = \frac{\mathrm{e}^{x} - \mathrm{e}^{-x}}{\mathrm{e}^{x} + \mathrm{e}^{-x}}, \quad x \in \mathbb{R}. \tag{20.16}$$

It is easy to show that $\tanh(x) \in (-1, 1)$ for all $x \in \mathbb{R}$. Furthermore, tanh is monotone increasing, $\lim_{x \to \infty} \tanh(x) = 1$ and $\lim_{x \to -\infty} \tanh(x) = -1$. Similar to the sigmoid nonlinearity, tanh can lead to vanishing gradients and is rarely used in intermediate layers of CNNs.

ReLU This nonlinearity is defined as

$$\mathrm{ReLU}(x) = \max(0, x), \quad x \in \mathbb{R}. \tag{20.17}$$

It is easy to see that $\mathrm{ReLU}'(x) = 1$ for $x > 0$ and that $\mathrm{ReLU}'(x) = 0$ for $x < 0$. ReLU nonlinearity generally leads to faster convergence compared to sigmoid or tanh nonlinearities, and it typically works well in CNNs with properly chosen weight initialization strategy and learning rate. Several modifications of ReLU activation function such as Exponential Linear Units (ELUs) [2] have been proposed.

softmax Softmax nonlinearity is more specialized compared to the general nonlinearities listed above. It is defined as

$$\mathrm{softmax}(x)_i := \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)}, \quad x \in \mathbb{R}^n,$$

and maps a vector $x \in \mathbb{R}^n$ to a probability vector of length n. The intuition behind softmax is as follows: map $x \mapsto \exp(x)$ gives an order-preserving bijection between the set of real numbers $\mathbb{R}$ and the set of strictly positive real numbers $\mathbb{R}_{>0}$, so that for any indexes i, j we have $x_i < x_j$ if and only if $\exp(x_i) < \exp(x_j)$. Subsequent division by $\sum_{j=1}^{n} \exp(x_j)$ normalizes the result, giving probability vector as the output. This nonlinearity is used, e.g., in classification tasks, after the final fully connected layer with n outputs in a n-class classification problem. It should be noted, however, that softmax outputs do not truly model prediction uncertainty in the scenario of noisy labels (such as noisy organ segmentations in medical imaging).

20.3.3 Pooling layers

The goal of a pooling layer is to produce a summary statistic of its input and to reduce the spatial dimensions of the feature map (hopefully without losing essential information). For this the *max pooling* layer reports the maximal values in each rectangular neighborhood of each point (i, j) (or (i, j, k) for 3D data) of each input feature while the *average pooling* layer reports the average values. Most common form of maxpooling uses stride 2 together with kernel size 2, which corresponds to partitioning the feature map spatially into a regular grid of square or cubic blocks with side 2 and taking max or average over such blocks for each input feature.

While pooling operations are common building blocks of CNNs when the aim is to reduce the feature map spatial dimension, it should be noted that one can achieve similar goal by using, e.g., 3×3 convolutions with stride 2 if working with 2D data. In this

case one can also simultaneously double the number of filters to reduce information loss while at the same time aggregating higher level features. This downsampling strategy is used, e.g., in ResNet [3] architecture.

20.3.4 Fully connected layers

Fully connected layer with n input dimensions and m output dimensions is defined as follows. The layer output is determined by the following parameters: the weight matrix $W \in M_{m,n}(\mathbb{R})$ having m rows and n columns, and the bias vector $b \in \mathbb{R}^m$. Given input vector $x \in \mathbb{R}^n$, the output of a fully-connected layer FC with activation function f is defined as

$$FC(x) := f(Wx + b) \in \mathbb{R}^m. \tag{20.18}$$

In the formula above, Wx is the matrix product and the function f is applied componentwise.

Fully connected layers are used as final layers in classification problems, where a few (most often one or two) fully-connected layers are attached on top of a CNN. For this, the CNN output is flattened and viewed as a single vector. Another example would be various autoencoder architectures, where FC layers are often attached to the latent code in both encoder and decoder paths of the network. When working with convolutional neural network it is helpful to realize that for a feature map with n channels one can apply a convolution filter with kernel size 1 and m output channels, which would be equivalent to applying a same fully-connected layer with m outputs to each point in the feature map.

20.4. CNN architectures for classification

Convolutional neural networks were originally introduced more than 20 years ago with the development of the LeNet CNN architecture [4,5]. Originally, the applications of CNNs were limited to relatively simple problems like handwritten digit recognition, but in the recent years CNN-based approaches have become dominant in image classification, object localization, and image segmentation tasks. This popularity can be attributed to two major factors: availability of computational resources (mostly GPUs) and data, on the one hand, and improvements in CNN architectures, on the other. Today CNN architectures have been developed that are quite successful in the tasks of image classification, object localization and image/instance segmentation. Below we will discuss a few noteworthy CNN architectures for image classification problems.

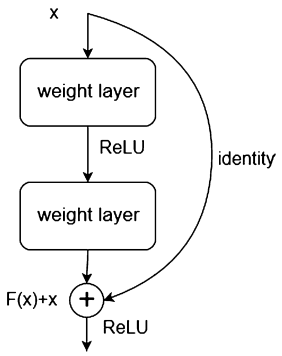
A neural network needs to have enough expressive power, depending on the task, to perform well. A naive approach towards increasing the capacity is increasing the number of filters in convolutional layers and the depth of the network. This approach was

taken in the *AlexNet* [6] architecture, which was the first architecture that popularized CNNs in computer vision by winning the ImageNet ILSVRC challenge [7] in 2012. It featured 5 convolutional layers and only feed-forward connections with the total of 60M trainable parameters. Shortly after it was outperformed by

- *ZF Net* [8] in 2013;
- *GoogLeNet* [9], the winning solution for the 2014 version of the challenge;
- *VGG16/VGG19* [10], which scored the second-best in this challenge but showed better single-net performance and was conceptually simpler.

VGG19 features 19 trainable layers connected in a feed-forward fashion, of which 16 layers are convolutional and relies on 3×3 convolutions with stride 1 and ReLU activations. Convolutional layers are gathered in 2 blocks of 2 layers for the first convolutional blocks and in 3 blocks of 4 layers for the last convolutional blocks. Maxpooling is performed in between the convolutional blocks, and the number of features in convolutional blocks doubles after each maxpooling operation. An important difference between AlexNet and VGG (as well as more modern architectures) is how large effective receptive field size is created: AlexNet used 11×11 filters in its initial layer while VGG uses stacks of 3×3 filters, and it can be easily shown that this is more parameter-efficient way of increasing receptive field size. Compared to VGG19, GoogLeNet has much less trainable parameters (4M for 22 layers vs. 140M for 19 layers of VGG19) which is due to the introduction of the so-called *inception module*, which is a deviation from the standard feedforward pattern, and helps to improve parameter efficiency.

However, all the architectures above still largely rely on feedforward information flow similar to the original LeNet-5 [4,5], while the benefits of these architectures mostly stem from their depth. Making the feedforward architectures even deeper leads to a number of challenges in addition to increased number of parameters. The first obstacle is the problem of vanishing/exploding gradients [11,23,12]. This can be largely addressed by normalized weight initialization and intermediate normalization layers, such as Batch Normalization [13]. Yet another obstacle is the performance degradation problem [3]: as the depth of a feedforward neural network increases, both testing and training accuracies get saturated and degrade afterwards. Such performance degradation is not explained by overfitting, and indicates that such networks are generally harder to optimize. Alternative CNN architectures have been suggested to deal with these shortcomings. A common feature of such architectures is the use of skip connections, which carry over information directly from earlier layers into the later layers without passing through intermediate convolutional layers. This, supposedly, helps in general to prevent information from "washing out". This general idea has been implemented in a number of ways.



ResNet [3] architecture, which was the basis for the winning solution in ILSVRC 2015 challenge [7], deals with the aforementioned issues by introducing *residual blocks*. Suppose that we are considering a residual block with two convolutional layers. In such a block the original input x goes through the first convolutional layer (typically a 3×3 convolution), after that Batch Normalization is applied and then the ReLU nonlinearity follows. The result is fed into the next convolutional layer, after which Batch Normalization is performed. This gives the output $F(x)$, to which x is added pointwise and then ReLU nonlinearity is applied. The output of the block hence equals $\mathrm{ReLU}(F(x)+x)$. In general, ResNets are build out of a varying number of such residual blocks with 2–3 convolutional layers in each block. In particular, it is interesting to note that, according to [3], using only a single convolutional layer in a residual block did not give any advantages compared to the plain feedforward architecture without residual connections. For feature map downsampling, convolutions with stride 2 are used, while the number of feature maps is doubled at the same time. This architecture choice allows training networks with more than a 100 layers and no performance degradation, as is shown in [3]. ResNet-based architectures are often used up to this day, and several extensions were proposed as well [14]. Furthermore, ResNet is often used as a CNN feature extractor for object detection and instance segmentation, e.g., in Mask R-CNN [15].

A more recent development is the DenseNet architecture [16], which is build from a collection of *dense blocks*, with "transition layers" (1×1 convolutions and 2×2 average pooling) to reduce the size of the feature maps in between. The main insight is that each such dense block consist of a few "convolutional layers", which, depending on a DenseNet variant, are either a stack of a 3×3 convolutional layer, Batch Normalization layer, and ReLU nonlinearity, or, alternatively, contain an initial 1×1 convolutional layer with Batch Normalization and ReLU nonlinearity to reduce the number of input features. Each such "convolutional layer" provides its output features to *all* successive convolutional layers in the block, leading to a "dense" connectivity pattern. The output of a dense block is the stack of all resulting feature maps and the input features, and as a result the successive dense blocks have access to these features and don't need to relearn them. This architecture achieves comparable or better performance on ILSVRC as compared to a ResNet architecture while also using significantly less parameters. For more details, we refer to [16].

20.5. Practical methodology

20.5.1 Data standardization and augmentation

Prior to feeding the data to the neural network for training, some preprocessing is usually done. Many beginners fail to obtain reasonable results not because of the architectures or methods or lack of regularization, but instead because they simply did not normalize and visually inspect their data. Two most important forms of pre-processing are *data standardization* and *dataset augmentation*. There are a few data standardization techniques common in imaging.

- **Mean subtraction.** During mean subtraction, the mean of every channel is computed over the training dataset, and these means are subtracted channelwise from both the training and the testing data.
- **Scaling.** Scaling amounts to computing channelwise standard deviations across the training dataset, and dividing the input data channelwise by these values so as to obtain a distribution with standard deviation equal to 1 in each channel. In place of division by standard deviation one can divide, e.g., by 95-percentile of the absolute value of a channel.
- **Specialized methods.** In addition to these generic methods, there are also some specialized standardization methods for medical imaging tasks, e.g., in chest X-ray one has to work with images coming from different vendors, furthermore, X-ray tubes might be deteriorating. In [17] local energy-based normalization was investigated for chest X-ray images, and it was shown that this normalization technique improves model performance on supervised computer-aided detection tasks. For another example, when working with hematoxylin and eosin (H&E) stained histological slides, one can observe variations in color and intensity in samples coming from different laboratories and performed on different days of the week. These variations can potentially reduce the effectiveness of quantitative image analysis. A normalization algorithm specifically designed to tackle this problem was suggested in [18], where it was also shown that it improves the performance for a few computer-aided detection tasks on these slide images. Finally, in certain scenarios (e.g., working directly with raw sinogram data for CT or Digital Breast Tomosynthesis [19]) it is reasonable to take log-transform of the input data as an extra preprocessing step.

Neural networks are known to benefit from large amounts of training data, and it is a common practice to artificially enlarge an existing dataset by adding data to it in a process called "*augmentation*". We distinguish between train-time augmentation and test-time augmentation, and concentrate on the first for now (which is also more common). In case of train-time augmentation, the goal is to provide a larger training dataset to the algorithm. In a supervised learning scenario, we are given a dataset $\mathcal{D}$ consisting of pairs (x_j, y_j) of a training sample $x_j \in \mathbb{R}^d$ and the corresponding label y_j. Given the dataset $\mathcal{D}$,

one should design transformations $T_1, T_2, \ldots, T_n : \mathbb{R}^d \to \mathbb{R}^d$ which are label-preserving in a sense that for every sample $(x_j, y_j) \in \mathcal{D}$ and every transformation T_i the resulting vector $T_i x_j$ still looks like a sample from $\mathcal{D}$ with label y_j. Multiple transformations can be additionally stacked, resulting in greater number of new samples. The resulting new samples with labels assigned to them in this way are added to the training dataset and optimization as usual is performed. In case of the test-time augmentation the goal is to improve test-time performance of the model as follows. For a predictive model f, given a test sample $x \in \mathbb{R}^d$, one computes the model predictions $f(x), f(T_1 x), \ldots, f(T_n x)$ for different augmenting transformations and aggregates these predictions in a certain way (e.g., by averaging softmax-output from classification layer [6]). In general, choice of the augmenting transformation depends on the dataset, but there are a few common strategies for data augmentation in imaging tasks:

- **Flipping.** Image x is mirrored in one or two dimensions, yielding one or two additional samples. Flipping in horizontal dimension is commonly done, e.g., on the ImageNet dataset [6], while on medical imaging datasets flipping in both dimensions is sometimes used.
- **Random cropping and scaling.** Image x of dimensions $W \times H$ is cropped to a random region $[x_1, x_2] \times [y_1, y_2] \subseteq [0, W] \times [0, H]$, and the result is interpolated to obtain original pixel dimensions if necessary. The size of the cropped region should still be large enough to preserve enough global context for correct label assignment.
- **Random rotation.** An image x is rotated by some random angle φ (often limited to the set $\varphi \in [\pi/2, \pi, 3\pi/2]$). This transformation is useful, e.g., in pathology, where rotation invariance of samples is observed; however, it is not widely used on datasets like ImageNet.
- **Gamma transform.** A grayscale image x is mapped to image x^γ for $\gamma > 0$, where $\gamma = 1$ corresponds to identity mapping. This transformation in effect adjusts the contrast of an image.
- **Color augmentations.** Individual color channels of the image are altered in order to capture certain invariance of classification with respect to variation in factors such as intensity of illumination or its color. This can be done, e.g., by adding small random offsets to individual channel values; an alternative scheme based on PCA can be found in [6].

20.5.2 Optimizers and learning rate

As discussed above, the optimization goal when training neural networks is minimization of the empirical risk R_{emp}. This is done by, firstly, computing the gradient $\nabla_\theta R_{\text{emp}}$ of the risk on a minibatch of training data with respect to the neural network parameters θ using backpropagation, and, secondly, updating the neural network weights θ accordingly. This update in its most basic form of *stochastic gradient descent* is given by the

formula

$$\theta := \theta - \eta \cdot \nabla_\theta R_{\text{emp}},$$

where $\eta > 0$ is the hyperparameter called the *learning rate*. A common extension of this algorithm is the addition of *momentum*, which in theory should accelerate the convergence of the algorithm on flat parts of the loss surface. In this case, the algorithm remembers the previous update direction and combines it with the newly computed gradient to determine the new update direction:

$$\begin{aligned} \delta\theta &:= \alpha \cdot \delta\theta - \eta \cdot \nabla_\theta R_{\text{emp}}, \\ \theta &:= \theta + \delta\theta. \end{aligned}$$

More recent variations to stochastic gradient descent are adaptive methods such as RMSProp and Adam [20], which extends RMSProp by adding momentum for the gradient updates. All these methods (SGD, RMSProp, Adam) are implemented in deep learning frameworks such as Tensorflow and PyTorch. Adam, in particular, is a popular choice once a good starting learning rate is picked. However, one should note that there is some recent research (see, e.g., [21]) suggesting that adaptive methods such as Adam and RMSProp may lead to poorer generalization and that properly tuned SGD with momentum is a safer option.

Choice of a proper learning rate is still driven largely by trial and error up to this date, including learning rate for adaptive optimizers such as Adam. This choice depends heavily on the neural network architecture, with architectures such as ResNet and DenseNet including Batch Normalization known to work well with relatively large learning rates in the order of 10^{-1}, and the batch size, with larger batches allowing for higher learning rate and faster convergence. In general, it makes sense to pick the batch size as large as possible given the network architecture and image size, and then to choose the largest possible learning rate which allows for stable learning. If the error keeps oscillating (instead of steadily decreasing), it is advised to reduce the initial learning rate. Furthermore, it is common to use *learning rate schedule*, i.e., to change the learning rate during training depending on the current number of epochs and/or validation error. For instance, one can reduce the learning rate by a factor of 10 two times when the epoch count exceeds 50% and 75% of the total epoch budget; or one can choose to decrease the learning rate once the mean error on validation dataset stops decreasing in the process of training.

20.5.3 Weight initialization and pretrained networks

It is easy to see that if two neurons (or convolutional filters) in the same place of a computational graph have exactly the same bias and weights, then they will always

get exactly the same gradients, and hence will never be able to learn distinct features, and this would result in losing some expressive power of the network. The connection between random initialization and expressive power of the network was explicitly examined in [22].

To "break the symmetry" when cold-starting the neural network training it is a common practice to initialize the weights randomly with zero mean and variance depending on the "input size" of the neuron [23,24], while biases are still initialized by zeros. The initialization strategy [24], which is more recent and was particularly derived for ReLU activations, suggests to initialize the weights by a zero-mean Gaussian distribution whose standard deviation equals $\sqrt{\frac{2}{n}}$, where n is determined as follows:

- When initializing a fully-connected layer, n equals the number of input features of a layer;
- When initializing, e.g., a two-dimensional convolutional layer of dimension $k \times k$ with m input feature maps, n equals the product $k^2 \cdot m$.

Most practical initialization strategies such as He initialization are already implemented in deep learning frameworks such as PyTorch and Tensorflow.

A second option to training from cold start is to use a pretrained convolutional network, stack fully connected layers with randomly initialized weights atop for a particular classification task, and then fine-tune the resulting network on a particular dataset. This strategy is motivated by the heuristic that the ImageNet dataset is fairly generic, hence convolutional features learned on ImageNet should be useful for other imaging datasets as well. Pretrained networks such as VGG, ResNet, and DenseNet variants are easy to find online. When fine-tuning a pretrained CNN for a particular classification task, it often makes sense to choose a lower learning rate for updates of the convolutional feature extraction layers and a higher learning rate for the final classification layers.

20.5.4 Regularization

Regularization, generally speaking, is a wide range of ML techniques aimed at reducing overfitting of the models while maintaining theoretical expressive power.

- **L^1/L^2 regularization.** These regularization methods are one of the most well-known regularization methods originating in classical machine learning theory in connection with maximum a posteriori (MAP) estimates for Laplace and Gaussian priors, respectively [25]. So suppose now that we have a neural network with parameters θ and loss function $L(\theta)$. In case of L^2 regularization, the term $\frac{\lambda_2}{2} \cdot \|\theta\|_2^2$ is added to the loss function; in case of L^1 regularization, the term $\lambda_1 \cdot \|\theta\|_1$ is added instead; λ_1, λ_2 are hyperparameters. Intuitively speaking, L^2 regularization encourages the network to use all of its inputs a little, rather than some of the inputs a lot, while L^1 regularization encourages the network to learn sparse weight vectors (which can be used, e.g., for feature selection tasks). Also L^1/L^2 regularization is often already implemented in deep learning frameworks and is easy to use (e.g., in

PyTorch L^2 regularization is added by passing a nonzero parameter λ_2 to an optimizer), however, one should note that there are regularization methods specifically designed for neural networks which can be more effective.

- **Max norm constraints.** Another form of regularization is enforcing an absolute upper bound $\|\theta\|_2 \leqslant c$ on the norm of the weight. In practice, this corresponds to performing the parameter update as usual, and then scaling the resulting θ back to the ball $\{x : \|x\|_2 \leqslant c\}$ of radius c. As a consequence, this form of regularization prevents weights from exploding.
- **Dropout.** Introduced in [26], dropout is a very powerful and simple regularization method for neural networks. While training, dropout is implemented by keeping a neuron active with some probability $p \in (0, 1)$ (which is a hyperparameter that can be different for different layers) while also dividing the output activation by p, and setting it to zero otherwise. During inference, all neurons are kept active and no scaling is applied. Very often probabilities p are chosen in a way that early convolutional layers are kept intact with probabilities close or equal to 1, while the probability of keeping neuron active goes down for deeper layers. Activation scaling during training time in this procedure is introduced in order to keep mean activations the same as during inference time, while saving computation cost at inference time. Dropout is implemented as a layer in frameworks such as PyTorch and Tensorflow and it is straightforward to add it to a model. Dropout is included in many classical NN architectures for classification and segmentation, see, e.g., [16] and [27]. An interesting recent development is the work [28], where it was shown that dropout training in deep neural networks can be viewed as a form of approximate Bayesian inference.

20.6. Future challenges

Despite the enormous success of CNNs in computer vision, in general, and in medical imaging, in particular, in recent years, there remain important challenges as well. Firstly, there is a well-known problem of the lack of interpretability of predictions. For example, in an image classification problem a neural network can produce accurate predictions, but the internal CNN features remain a black box and do not reveal much information. In medical imaging, however, we would like to know what image features are responsible for the prediction. Some work is done in this direction, e.g., there are a few approaches to the visualization of saliency maps [29]. Furthermore, we would be interested in image features that have clear clinical interpretation, but extracting those in an unsupervised manner is challenging.

Secondly, there is often a problem of domain shift, which emerges when a neural network is trained on a dataset from one domain and then it is applied to a related, but different domain. Some examples would be when

- We make a model for object detection in urban scenes and train it on scenes generated in a computer game, then try to apply it on real-life scenes [30];
- We have multiple vendors for, e.g., mammography scanners, which apply some amount of vendor-specific processing so that resulting images look different [31].

In general, developing models that are robust to variations in acquisition equipment remains challenging.

Thirdly, the neural networks remain data-hungry, and there is ongoing work on improving the parameter efficiency [1].

References

[1] T.S. Cohen, M. Welling, Group equivariant convolutional networks, preprint, arXiv:1602.07576, 2016.
[2] D.A. Clevert, T. Unterthiner, S. Hochreiter, Fast and accurate deep network learning by exponential linear units (ELUs), preprint, arXiv:1511.07289, 2015.
[3] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: CVPR, IEEE Computer Society, 2016, pp. 770–778.
[4] Y. LeCun, B. Boser, J.S. Denker, D. Henderson, R.E. Howard, W. Hubbard, et al., Backpropagation applied to handwritten zip code recognition, Neural Computation 1 (4) (1989) 541–551, https://doi.org/10.1162/neco.1989.1.4.541.
[5] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, Proceedings of the IEEE 86 (11) (1998) 2278–2324.
[6] A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet classification with deep convolutional neural networks, in: F. Pereira, C.J.C. Burges, L. Bottou, K.Q. Weinberger (Eds.), Advances in Neural Information Processing Systems, vol. 25, Curran Associates, Inc., 2012, pp. 1097–1105, http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.
[7] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, et al., ImageNet large scale visual recognition challenge, International Journal of Computer Vision 115 (3) (2015) 211–252, https://doi.org/10.1007/s11263-015-0816-y.
[8] M.D. Zeiler, R. Fergus, Visualizing and understanding convolutional networks, CoRR, abs/1311.2901, 2013, URL http://dblp.uni-trier.de/db/journals/corr/corr1311.html#ZeilerF13.
[9] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, et al., Going deeper with convolutions, in: Computer Vision and Pattern, Recognition, CVPR, 2015, preprint, arXiv:1409.4842.
[10] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, CoRR, abs/1409.1556, 2014.
[11] Y. Bengio, P. Simard, P. Frasconi, Learning long-term dependencies with gradient descent is difficult, IEEE Transactions on Neural Networks 5 (2) (1994) 157–166, https://doi.org/10.1109/72.279181.
[12] J. Hochreiter, Untersuchungen zu dynamischen neuronalen Netzen, Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München, 1991.
[13] S. Ioffe, C. Szegedy, Batch normalization: accelerating deep network training by reducing internal covariate shift, in: Proceedings of the 32Nd International Conference on International Conference on Machine Learning, ICML'15, vol. 37, JMLR.org, 2015, pp. 448–456, URL http://dl.acm.org/citation.cfm?id=3045118.3045167.
[14] S. Xie, R.B. Girshick, P. Dollár, Z. Tu, K. He, Aggregated residual transformations for deep neural networks, in: CVPR, IEEE Computer Society, 2017, pp. 5987–5995.
[15] K. He, G. Gkioxari, P. Dollár, R.B. Girshick, Mask R-CNN, in: IEEE International Conference on Computer Vision, ICCV 2017,Venice, Italy, October 22–29, 2017, 2017, pp. 22–29.

[16] G. Huang, Z. Liu, L. van der Maaten, K.Q. Weinberger, Densely connected convolutional networks, in: 2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21–26, 2017, 2017, pp. 21–26.
[17] R.H.H.M. Philipsen, P. Maduskar, L. Hogeweg, J. Melendez, C.I. Sánchez, B. van Ginneken, Localized energy-based normalization of medical images: application to chest radiography, IEEE Transactions on Medical Imaging 34 (9) (2015) 1965–1975, https://doi.org/10.1109/TMI.2015.2418031.
[18] B.E. Bejnordi, G. Litjens, N. Timofeeva, I. Otte-Höller, A. Homeyer, N. Karssemeijer, et al., Stain specific standardization of whole-slide histopathological images, IEEE Transactions on Medical Imaging 35 (2) (2016) 404–415, https://doi.org/10.1109/TMI.2015.2476509.
[19] N. Moriakov, K. Michielsen, J. Adler, R. Mann, I. Sechopoulos, J. Teuwen, Deep learning framework for digital breast tomosynthesis reconstruction, preprint, arXiv:1808.04640, 2018.
[20] D.P. Kingma, J. Ba, Adam: a method for stochastic optimization, preprint, arXiv:1412.6980, 2014.
[21] A.C. Wilson, R. Roelofs, M. Stern, N. Srebro, B. Recht, The marginal value of adaptive gradient methods in machine learning, preprint, arXiv:1705.08292.
[22] A. Daniely, R. Frostig, Y. Singer, Toward deeper understanding of neural networks: the power of initialization and a dual view on expressivity, in: D.D. Lee, M. Sugiyama, U.V. Luxburg, I. Guyon, R. Garnett (Eds.), Advances in Neural Information Processing Systems, vol. 29, Curran Associates, Inc., 2016, pp. 2253–2261, http://papers.nips.cc/paper/6427-toward-deeper-understanding-of-neural-networks-the-power-of-initialization-and-a-dual-view-on-expressivity.pdf.
[23] X. Glorot, Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, in: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13–15, 2010, 2010, pp. 249–256, URL http://www.jmlr.org/proceedings/papers/v9/glorot10a.html.
[24] K. He, X. Zhang, S. Ren, J. Sun, Delving deep into rectifiers: surpassing human-level performance on ImageNet classification, in: Proceedings of the 2015 IEEE International Conference on Computer Vision, ICCV'15, IEEE Computer Society, Washington, DC, USA, ISBN 978-1-4673-8391-2, 2015, pp. 1026–1034.
[25] C.M. Bishop, Pattern Recognition and Machine Learning, Information Science and Statistics, Springer-Verlag, Berlin, Heidelberg, ISBN 0387310738, 2006.
[26] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: a simple way to prevent neural networks from overfitting, Journal of Machine Learning Research 15 (2014) 1929–1958, URL http://jmlr.org/papers/v15/srivastava14a.html.
[27] S. Jégou, M. Drozdzal, D. Vázquez, A. Romero, Y. Bengio, The one hundred layers Tiramisu: fully convolutional DenseNets for semantic segmentation, CoRR, abs/1611.09326, 2016.
[28] Y. Gal, Z. Ghahramani, Dropout as a Bayesian approximation: representing model uncertainty in deep learning, preprint, arXiv:1506.02142, 2015.
[29] K. Simonyan, A. Vedaldi, A. Zisserman, Deep inside convolutional networks: visualising image classification models and saliency maps, in: 2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14–16, 2014, Workshop Track Proceedings, 2014, pp. 14–16, arXiv:1312.6034.
[30] Y. Chen, W. Li, C. Sakaridis, D. Dai, L.V. Gool, Domain adaptive faster R-CNN for object detection in the wild, CoRR, abs/1803.03243, 2018.
[31] J. van Vugt, E. Marchiori, R. Mann, A. Gubern-Mérida, N. Moriakov, J. Teuwen, Vendor-independent soft tissue lesion detection using weakly supervised and unsupervised adversarial domain adaptation, CoRR 2018, abs/1808.04909.