

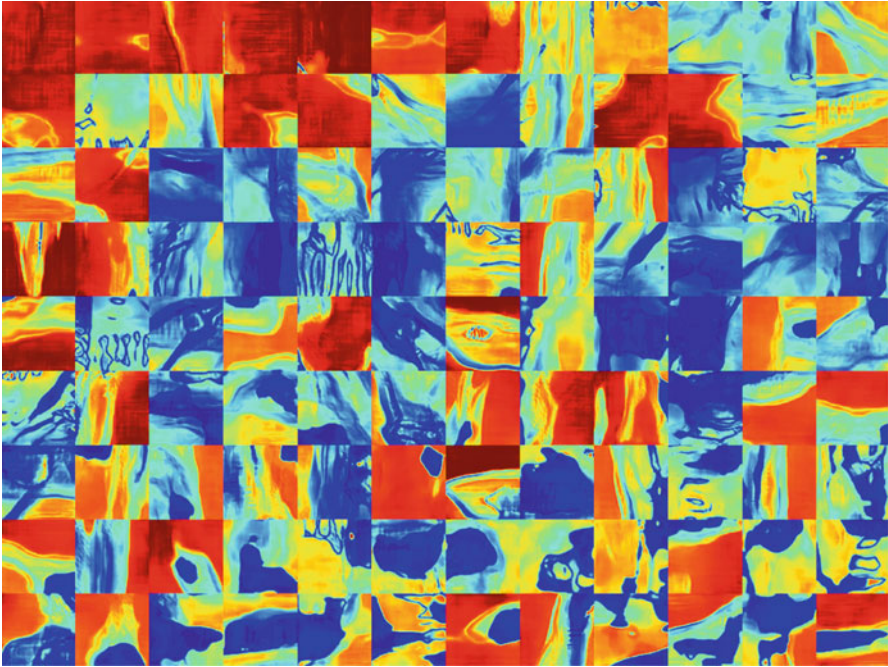
Chapter 7

Deep Learning in X-ray Testing



Abstract Deep learning has been inspired by ideas from neuroscience. The key idea of deep learning is to replace *handcrafted* features (explained in details in Chap. 5) with features that are *learned* efficiently using a hierarchical feature extraction approach. Usually, the learned features are so discriminative that no sophisticated classifiers are required. In last years, deep learning has been successfully used in image and video recognition, and it has been established as the state of the art in many areas such as computer vision, machine translation, and natural language processing. In comparison with other computer vision applications, we have seen that the introduction of techniques based on deep learning in computer vision for X-ray testing has been rather slow. However, there are many methods based on deep learning that have been designed and tested in some X-ray testing applications. In this chapter, we review many relevant concepts of deep learning that can be used in computer vision for X-ray testing. We covered the theory and practice of deep learning techniques in real X-ray testing problems. The chapter explained neural networks, Convolutional Neural Network (CNN) that can be used in classification problems, pre-trained models, transfer learning that are used in sophisticated models, Generative Adversarial Networks (GANs) to generate synthetic images, and modern detection methods that are used to classify and localize objects in an image. In addition, for every method, we give not only the basic concepts but also practical details in real X-ray testing examples that have been implemented in Python.

Cover image: *Synthetic X-ray images generated by a GAN model that has been trained using X-ray images of backpacks with no threat objects (from series B0082 colored with 'jet' colormap).*



7.1 Introduction

Originally, deep learning is inspired by ideas from neuroscience [19]. In last years, deep learning has been successfully used in image and video recognition (see, for example, [3, 31, 58]), and it has been established as the state of the art in many areas such as computer vision, machine translation, and natural language processing [57].

The key idea of deep learning is to replace *handcrafted* features (explained in details in Chap. 5) with features that are *learned* efficiently using a hierarchical feature extraction approach. Usually, the learned features are so discriminative that no sophisticated classifiers are required. In recent years, we have witnessed tremendous improvements in many fields of computer vision by using complex deep neural network architectures trained with thousands or millions of images (e.g., face recognition [10], object recognition and detection [35, 71], diagnosis of prostate cancer [44], classification of skin cancer [14], among others). Methods based on deep learning have become fundamental in these fields, however, an enormous number of images used for training purposes are required in order to achieve satisfactory results.

In comparison with other computer vision applications, we have seen that the introduction of techniques based on deep learning in computer vision for X-ray testing has been rather slow. In our opinion, this is due to three reasons. The first has to do with the availability of public databases that can be used for these pur-

poses. While in some areas of computer vision (e.g., face recognition), there are hundreds of databases since the 1990s, in X-ray testing, there is only one public database for X-ray testing for general purposes [40] created in the last 5 years with around 20,000 X-ray images, and another one for baggage inspection [42] created recently with around 1 million X-ray images. The rest of the datasets used in the experiments reported by the industry and academia are private. In many cases, the entities (industry, government, or academia) that fund research in X-ray testing do not allow databases to be made public. Sometimes this happens in baggage inspection research (for security reasons) or in other industrial applications (to prevent competitors from having access to data that could improve their processes). The second reason is related to the number of experts working in this field. While in other areas of computer vision almost anyone can be an expert (such as in object recognition), in nondestructive testing the relative number of people working on these subjects is rather low and usually, their work is expensive. In this kind of computer applications, experts are necessary to label the data (make annotations, define bounding boxes, etc.). It is very simple to find people that detect bicycles in photographs, however, it is not so easy to find human operators that can distinguish the anomalies in a welding process by observing an X-ray image. Finally, the last reason is that, in other applications of computer vision, color photos can be acquired with inexpensive equipment (often a cell phone), whereas in X-ray testing, we need expensive equipment.

In this chapter, we review many relevant concepts of deep learning that can be used in computer vision for X-ray testing. This chapter should be considered as an introduction to the subject rather than an in-depth treatise.¹ We will cover many relevant topics, so the reader will be able to understand and apply these techniques in real X-ray testing problems. The chapter begins with the basics, i.e., neural networks (see Sect. 7.2). Afterwards, we will review the Convolutional Neural Network (CNN) (see Sect. 7.3) that can be used in a classification problem. CNNs can be trained from scratch or using pre-trained models (see Sect. 7.4) or transfer learning (see Sect. 7.5). In addition, we cover the Generative Adversarial Networks (GANs) (see Sect. 7.6) that have been proposed to generate synthetic images. Finally, we give an overview of more complex architectures that can be used as detection methods (see Sect. 7.7), i.e., when we want to classify and localize an object in an image. For every section, we will cover the basic concepts, give practical details (e.g., training and testing) and show some examples in X-ray testing using Python.

7.2 Neural Networks

Artificial neuronal networks are mathematical tools derived from what is known about the mechanisms and physical structure of biological learning, based on the function of a neuron. They are parallel structures for the distributed processing of information [4]. A neural network consists of artificial neurons connected in a net-

¹Recommendation for further reading: [1, 18, 31].

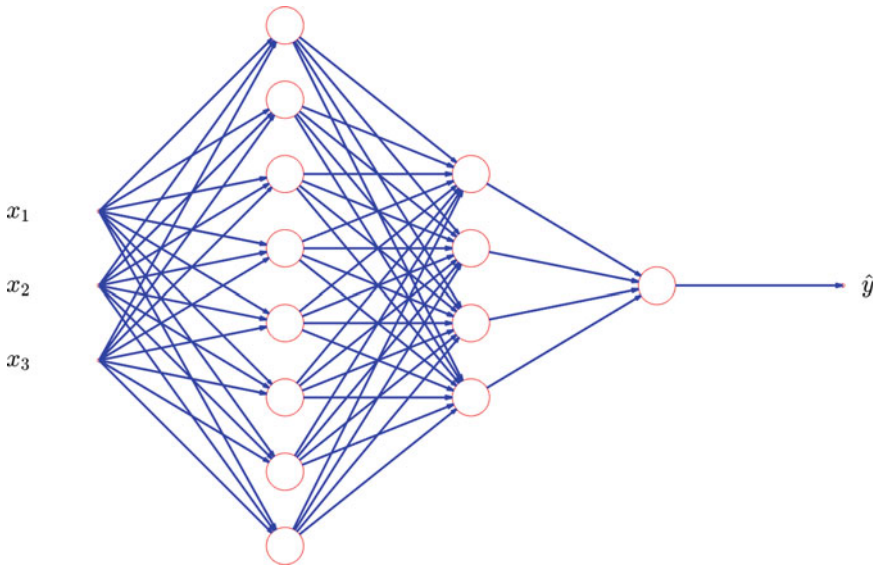


Fig. 7.1 Simple neural network with three inputs $\mathbf{x} = (x_1, x_2, x_3)$, one output \hat{y} , and two hidden layers (one with 8 nodes and the another with 4). In this example, the input can be classified as class ω_1 if $\hat{y} > 0.5$, and otherwise as class ω_0

work that is able to classify a test feature vector \mathbf{x} evaluating a linear weighted sum of non-linear functions as illustrated in Fig. 7.1. The weights, the functions, and the connections are estimated in a training phase by minimizing the classification error [4, 5]. Neural networks have been established as one of the best classification approaches in pattern recognition. The basic structure of the neural networks and the learning strategies developed for training neural networks are the basis of deep learning models.

7.2.1 Basics of Neural Networks

The basic processing unit is the neuron, made up of multiple inputs and only one output as shown in Fig. 7.2. This output is determined by an activation function that operates on input values, and a transfer function that operates on the activation value. In other words, if we consider the input vector $\mathbf{x} = [x_1 \dots x_n]^T$, the weight vector $\mathbf{w} = [w_1 \dots w_n]^T$, the activation value z , and the output value of the neuron a , the values of z and a can be described by a linear projection and an a non-linear function:

$$z = \mathbf{w}^T \mathbf{x} + b \quad a = \sigma(z), \quad (7.1)$$

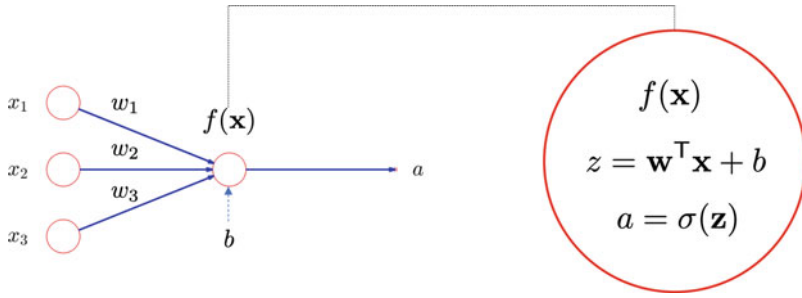


Fig. 7.2 Single neuron with three inputs (x_1, x_2, x_3), three weights, one weight for each input, (w_1, w_2, w_3), one bias value (b), and one output a

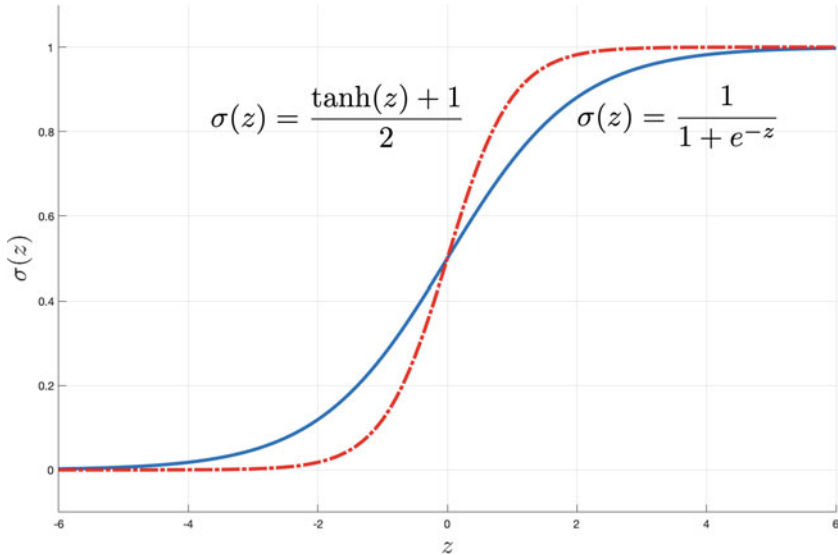


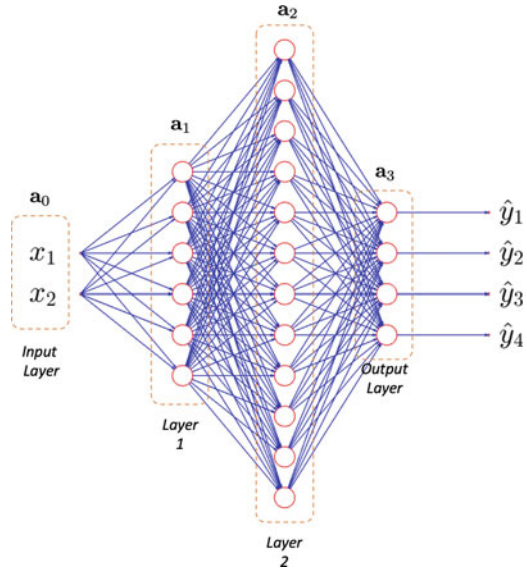
Fig. 7.3 Two typical sigmoids as activation functions

where b is the bias value and $\sigma(z)$ is the so-called transfer function or activation function and is generally a sigmoid such as (see Fig. 7.3)

$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad \text{or} \quad \sigma(z) = \frac{\tanh(z) + 1}{2}. \tag{7.2}$$

A very simple structure, called *logistic regression*, is defined for two classes and no hidden layer, i.e., the output of the model is $y = a$. Thus, class ω_1 is determined when $y > 0.5$, and ω_0 otherwise. This is a linear approach because the separation of both classes corresponds to a hyperplane (or a straight line for a feature space of two dimensions).

Fig. 7.4 Multi-Layer Perceptron (MLP) with one input layer with two inputs, two hidden layers (Layer 1 and Layer 2) with 6 and 12 nodes respectively and one output layer with four outputs. [→ Example 7.1] [→ Example 7.2]



The structure of a neuronal network can have one or more neurons and depending on the type of problem and the training, these networks receive different names. They have the capacity to associate and classify patterns, compress data, perform process control, and approximate non-linear functions [43].

The most often used type of neural network in classification is the Multi-Layer Perceptron (MLP) which consists of sequential layers of neurons. The structure of an MLP is shown in Fig. 7.4 where each neuron has Eq. (7.1) associated to it. It consists of a input layer, hidden layers, and an output layer (in Fig. 7.4, there are two hidden layers, a_1 and a_2). In a classification problem based on neural networks, the input \mathbf{x} corresponds to the feature vector, and the output $\hat{\mathbf{y}}$ that is the classification of \mathbf{x} . Output $\hat{\mathbf{y}}$ is defined as a vector of K elements for a classification into K classes. The value \hat{y}_i can be understood as the probability that sample \mathbf{x} is classified a class ω_i . Formerly,

$$h_m(\mathbf{x}) = \underset{i}{\operatorname{argmax}} \{ \hat{y}_1, \dots, \hat{y}_K \}. \quad (7.3)$$

Usually, an index $k = 0, \dots, m$ is used to denote a layer, where $k = 0$ is the input layer and $k = m$ is the output layer (in Fig. 7.4, $m = 3$). In addition, index $i = 1, \dots, n_k$ is used to denote the node i of layer k . In Fig. 7.4, the two hidden layers have $n_1 = 6$ and $n_2 = 12$ nodes respectively. We define the output of layer k as vector \mathbf{a}_k , and it is a vector with n_k elements. In this definition, \mathbf{a}_0 corresponds to \mathbf{x} , i.e., the input vector of the neural network with n_0 elements (in Fig. 7.4, $n_0 = 2$ for two inputs). Similarly, \mathbf{a}_m corresponds to $\hat{\mathbf{y}}$, i.e., the output vector of the neural network (in Fig. 7.4, $n_m = 4$ for four outputs). Thus, in general, layer k is defined by

$$\mathbf{a}_k = \begin{bmatrix} a_k(1) \\ a_k(2) \\ \vdots \\ a_k(n_k) \end{bmatrix}, \tag{7.4}$$

where the input and output layers are respectively:

$$\mathbf{x} = \mathbf{a}_0 = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n_0} \end{bmatrix} = \begin{bmatrix} a_0(1) \\ a_0(2) \\ \vdots \\ a_0(n_0) \end{bmatrix} \tag{7.5}$$

$$\hat{\mathbf{y}} = \mathbf{a}_m = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_{n_m} \end{bmatrix} = \begin{bmatrix} a_m(1) \\ a_m(2) \\ \vdots \\ a_m(n_m) \end{bmatrix}. \tag{7.6}$$

With these definitions, it is simple to write the equation of each node according to (7.1) for $k = 1, \dots, m$:

$$\mathbf{z}_k = \mathbf{W}_k \mathbf{a}_{k-1} + \mathbf{b}_k, \quad \mathbf{a}_k = \sigma(\mathbf{z}_k). \tag{7.7}$$

In this equation, \mathbf{z}_k , \mathbf{a}_k , and \mathbf{b}_k are n_k -element vectors, \mathbf{a}_{k-1} is a n_{k-1} -element vector, and \mathbf{W}_k is a matrix of $n_k \times n_{k-1}$ elements, where $w_k(i, j)$ is the weight of the connection between node i of layer k and node j of layer $k - 1$.

For our example of Fig. 7.4 with $m = 3$, output \mathbf{y} can easily computed using following steps:

$$\begin{bmatrix} \mathbf{z}_1 = \mathbf{W}_1 \mathbf{a}_0 + \mathbf{b}_1, & \mathbf{a}_1 = \sigma(\mathbf{z}_1) \\ \mathbf{z}_2 = \mathbf{W}_2 \mathbf{a}_1 + \mathbf{b}_2, & \mathbf{a}_2 = \sigma(\mathbf{z}_2) \\ \mathbf{z}_3 = \mathbf{W}_3 \mathbf{a}_2 + \mathbf{b}_3, & \mathbf{a}_3 = \sigma(\mathbf{z}_3). \end{bmatrix} \tag{7.8}$$

with $\mathbf{a}_0 = \mathbf{x}$ and $\mathbf{a}_3 = \hat{\mathbf{y}}$.

This procedure is called *forward-propagation*, and it is used to compute output $\hat{\mathbf{y}}$ from input \mathbf{x} and parameters Θ , where the parameters are defined as $\Theta = \{\boldsymbol{\theta}_k\}_{k=1}^m$ with $\boldsymbol{\theta}_k = (\mathbf{W}_k, \mathbf{b}_k)$, i.e., the parameters of each layer. The reader can observe that

the computation of $\hat{\mathbf{y}}$ is very fast, because in these equations, there are only multiplications and additions of vectors and matrices.

7.2.2 Training of Neural Networks

In order to train a neural network, parameters Θ are to be estimated. For this end, we have a training dataset of N samples $\{\mathbf{x}_i\}_{i=1}^N$ and its corresponding ideal classification $\{\mathbf{y}_i\}_{i=1}^N$. We distinguish between the ground truth \mathbf{y}_i (ideal classification for sample \mathbf{x}_i) and the output of the neural network $\hat{\mathbf{y}}_i$ (real classification of \mathbf{x}_i). The idea of the training is to find parameters Θ so that the difference between \mathbf{y}_i and $\hat{\mathbf{y}}_i$ is minimal for $i = 1, \dots, N$. In neural networks, a loss function, $f_{\text{loss}}(\mathbf{y}_i, \hat{\mathbf{y}}_i)$ is used to compute the difference between ideal (\mathbf{y}_i) and real ($\hat{\mathbf{y}}_i$), so the training process can be stated as an optimization problem in which an objective function is to be minimized:

$$J(\Theta) = \frac{1}{N} \sum_{i=1}^N f_{\text{loss}}(\hat{\mathbf{y}}_i, \mathbf{y}_i) \rightarrow \min. \quad (7.9)$$

Intuitively, the loss function (7.9) can be based on the norm:

$$f_{\text{loss}}(\hat{\mathbf{y}}_i, \mathbf{y}_i) = \frac{1}{2} \|\hat{\mathbf{y}}_i - \mathbf{y}_i\|^2, \quad (7.10)$$

however, more sophisticated loss functions, like *cross-entropy* that minimizes the distance between both probability distributions are typically used [1, 4]:

$$f_{\text{loss}}(\hat{\mathbf{y}}_i, \mathbf{y}_i) = -\mathbf{y}_i \log(\hat{\mathbf{y}}_i) - (1 - \mathbf{y}_i) \log(1 - \hat{\mathbf{y}}_i) \quad (7.11)$$

for a two-class problem.² In order to find the parameters, i.e., to minimize objective function J , a method based on gradient descent can be used. We choose start parameters with random values $(\mathbf{W}_k, \mathbf{b}_k)$ for each layer, and they will be updated iteratively by small increments using the opposite direction of the gradient of the objective function J . The iterative method is summarized as follows:

1. Parameters $\theta_k = (\mathbf{W}_k, \mathbf{b}_k)$, for $k = 1, \dots, m$, are initialized with random values.

$$\mathbf{W}_k := \text{random matrix}(n_k \times n_{k-1}) \quad , \quad \mathbf{b}_k := \text{random vector}(n_k \times 1). \quad (7.12)$$

2. Layer outputs are computed for each training sample i using (7.7):

$$\mathbf{z}_{k,i} = \mathbf{W}_k \mathbf{a}_{k-1,i} + \mathbf{b}_k \quad , \quad \mathbf{a}_{k,i} = \sigma(\mathbf{z}_{k,i}). \quad (7.13)$$

²For a multi-class problem, we sum the loss for each class.

3. Derivatives of the parameters are computed:

$$\Delta \mathbf{W}_k = \frac{\partial J}{\partial \mathbf{W}_k} \quad , \quad \Delta \mathbf{b}_k = \frac{\partial J}{\partial \mathbf{b}_k}. \tag{7.14}$$

4. Parameters are updated using a learning rate α :

$$\mathbf{W}_k := \mathbf{W}_k - \alpha \Delta \mathbf{W}_k \quad , \quad \mathbf{b}_k := \mathbf{b}_k - \alpha \Delta \mathbf{b}_k. \tag{7.15}$$

5. The procedure is repeated from step 2 until convergence. For example, when

$$J(\mathbf{W}_1, \dots, \mathbf{W}_m, \mathbf{b}_1, \dots, \mathbf{b}_m) < \varepsilon. \tag{7.16}$$

We observe that step 2 corresponds to the forward-propagation of the neural network, that means we have an input $\mathbf{x}_i = \mathbf{a}_{0,i}$, and we evaluate forwards (from left to right) the layers of the network until we have the output $\hat{y}_i = \mathbf{a}_{m,i}$, and the output \mathbf{a}_k depends on input \mathbf{a}_{k-1} and parameters \mathbf{W}_k and \mathbf{b}_k . On the other hand, step 3 computes the increments $\Delta \mathbf{W}_k$ and $\Delta \mathbf{b}_k$ that are required in step 4 to update the parameters. Step 3 is performed using a *backward-propagation* approach, that means, we compute the derivatives at the output and we propagate them backwards (from right to left) through the layers using the chain rule for derivatives. The idea is that in the backward-propagation, we will have in each layer the increments $\Delta \mathbf{W}_k$ and $\Delta \mathbf{b}_k$ that depends on $\partial J / \mathbf{a}_k$ and parameters \mathbf{W}_k and \mathbf{b}_k as illustrated in Fig. 7.5. Formerly,

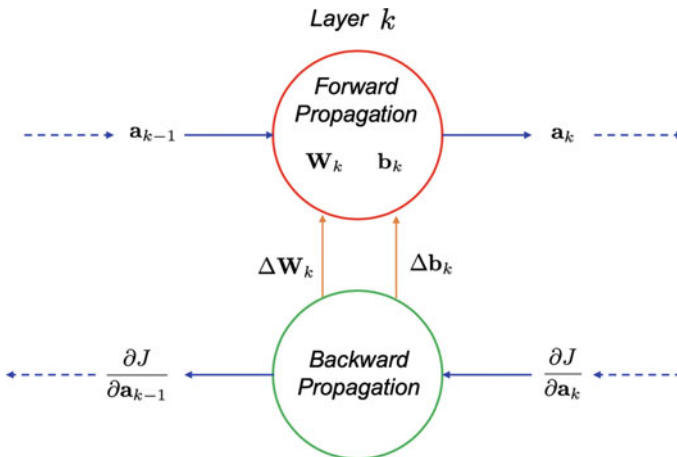


Fig. 7.5 Backward-propagation strategy

$$\Delta \mathbf{W}_k = \frac{\partial J}{\partial \mathbf{W}_k} = \underbrace{\frac{\partial J}{\partial \mathbf{a}_k}}_{\gamma_k} \underbrace{\frac{\partial \mathbf{a}_k}{\partial \mathbf{z}_k}}_{\mathbf{a}_{k-1}} \frac{\partial \mathbf{z}_k}{\partial \mathbf{W}_k} = \gamma_k \mathbf{a}_{k-1} \quad (7.17)$$

$$\Delta \mathbf{b}_k = \frac{\partial J}{\partial \mathbf{b}_k} = \underbrace{\frac{\partial J}{\partial \mathbf{a}_k}}_{\gamma_k} \underbrace{\frac{\partial \mathbf{a}_k}{\partial \mathbf{z}_k}}_1 \frac{\partial \mathbf{z}_k}{\partial \mathbf{b}_k} = \gamma_k \quad (7.18)$$

The last derivatives ($\partial \mathbf{z}_k / \partial \mathbf{W}_k$ and $\partial \mathbf{z}_k / \partial \mathbf{b}_k$) are computed from (7.7), and the term γ_k can be written as:

$$\gamma_k = \underbrace{\frac{\partial J}{\partial \mathbf{a}_k}}_{\text{input}} \underbrace{\frac{\partial \mathbf{a}_k}{\partial \mathbf{z}_k}}_{\sigma'_k} = \frac{\partial J}{\partial \mathbf{a}_k} \mathbf{a}_k (1 - \mathbf{a}_k), \quad (7.19)$$

where the last term is the derivative of the activation function $a = \sigma(z) = 1/(1 + e^{-z})$ and $\sigma'(z) = a(1 - a)$. In this approach, the derivative $\partial J / \partial \mathbf{a}_m$, that is the input of the most right node of the backward-propagation schema, is computed directly from (7.9) and (7.10) with $\hat{\mathbf{y}} = \mathbf{a}_m$:

$$\frac{\partial J}{\partial \mathbf{a}_m} = \frac{\partial}{\partial \mathbf{a}_m} \left\{ \frac{1}{N} \sum_{i=1}^N \frac{1}{2} \|\hat{\mathbf{y}}_i - \mathbf{y}_i\|^2 \right\} = \frac{1}{N} \sum_{i=1}^N (\mathbf{a}_{m,i} - \mathbf{y}_i). \quad (7.20)$$

Thus, the last four equations can be used to estimate the increments $\Delta \mathbf{W}_k$ and $\Delta \mathbf{b}_k$ of step 3 and the updates of the parameters \mathbf{W}_k and \mathbf{b}_k in step 4, for $k = m, m - 1, \dots, 1$.

We observe in the backward-propagation approach (Fig. 7.5), that the next layer, at the left, i.e., layer $(k - 1)$, needs $\partial J / \mathbf{a}_{k-1}$, that can be expressed as follows:

$$\frac{\partial J}{\partial \mathbf{a}_{k-1}} = \underbrace{\frac{\partial J}{\partial \mathbf{a}_k}}_{\gamma_k} \underbrace{\frac{\partial \mathbf{a}_k}{\partial \mathbf{z}_k}}_{\mathbf{W}_k} \frac{\partial \mathbf{z}_k}{\partial \mathbf{a}_{k-1}} = \gamma_k \mathbf{W}_k. \quad (7.21)$$

Typically, the iteration is stopped when the increments are small enough, that means that no significant update takes place.

Backpropagation is the learning algorithm normally used to train this type of network. Its goal is to minimize the error function constructed from the difference between the desired (\mathbf{y}) and modeled ($\hat{\mathbf{y}}$) output. In this section, we explained a simple backpropagation approach in four steps, where the increments \mathbf{W}_k and \mathbf{b}_k can be computed in an easy way.

7.2.3 Examples of Neural Networks

In this section, we give two examples:

1. The first one can be used to understand the basic operation and training of a neural network. The idea of this example is to design a neural network *from scratch*. Here, the reader can find all the details of the implementation of forward- and backward-propagation of a network with no sophisticated neural network library, only linear algebra is required (in our implementation for this end, we use the well-known numpy library³ [47]). [→ Example 7.1 🧑🏫]
2. The second example is to show, how we can design a neural network using a well-known library dedicated to machine learning, called ‘sklearn’.⁴ The idea of this example is to give details of the practice in typical applications that can be implemented with a neural network. [→ Example 7.2 🧑🏫]



Python Example 7.1: In this example, we present a classification problem using simulated data with using Gaussian distributions: four classes ($\omega_1, \dots, \omega_4$) for two features (x_1, x_2) (see Fig. 7.6). For this classification problem, we design a neural network with two hidden layers as shown in our example of Fig. 7.4. That means, the input layer has two entries, the hidden layers has 6 and 12 nodes respectively and the output layer as four elements ($\hat{y}_1, \dots, \hat{y}_4$). This example is provided for those readers that want to learn how a neural network is designed from scratch showing the general five steps explained in Eqs. (7.12)–(7.16): (1) random initialization of the parameters, (2) forward-propagation, (3) backward-propagation, (4) update of the parameters, and (5) repeat from step 2 until convergence. The details of all training steps can be found in `classifiers.py` of `pyxvis` Library, where the implementation is performed using only ‘numpy’ library. In this example, we used 80% of the available data for training and 20% for validation purposes.

Listing 7.1 : Neural network from scratch.

```
import numpy as np
from pyxvis.learning.classifiers import nn_definition
from pyxvis.learning.classifiers import nn_forward_propagation, nn_backward_propagation
from pyxvis.learning.classifiers import nn_parameters_update, nn_loss_function
from pyxvis.io.plots import plot_features_y, show_confusion_matrix, plot_loss
from pyxvis.io.data import load_features

# Load training and testing data
(Xtrain, Ytrain, Xtest, Ytest) = load_features('../data/G4/G4', categorical=1)
plot_features_y(Xtrain, Ytrain, 'Training Subset')

# Definitions
N = Xtrain.shape[1] # training samples
n_0 = Xtrain.shape[0] # number of inputs (X)
n_m = Ytrain.shape[0] # number of outputs (Y)
tmax = 1000 # max number of iterations
alpha = 10 # learning rate
loss_eps = 0.01 # stop if loss < loss_eps
```

³See <https://numpy.org>.

⁴See <https://scikit-learn.org> [48].

```

nh      = [6,12]           # nodes of hidden layers
n       = [n_0]+nh+[n_m]  # nodes of each layer
m       = len(n)-1
ltrain  = np.zeros([tmax,1]) # training loss

# Training
t       = -1
train  = 1
W,b    = nn_definition(n,N)           # (step 1)
while train:
    t      = t+1
    a      = nn_forward_propagation(Xtrain,W,b) # (step 2)
    dW,db  = nn_backward_propagation(Ytrain,a,W,b) # (step 3)
    W,b    = nn_parameters_update(W,b,dW,db,alpha) # (step 4)
    ltrain[t] = nn_loss_function(a,Ytrain) # (step 5)
    train  = ltrain[t]>=loss_eps and t<tmax-1

# Loss function on training and validation subsets
plot_loss(ltrain)

# Evaluation on training and testing subsets
a = nn_forward_propagation(Xtrain,W,b) # output layer is a[m]
show_confusion_matrix(a[m],Ytrain,'Training',categorical=1)
a = nn_forward_propagation(Xtest,W,b) # output layer is a[m]
show_confusion_matrix(a[m],Ytest,'Testing',categorical=1)

```

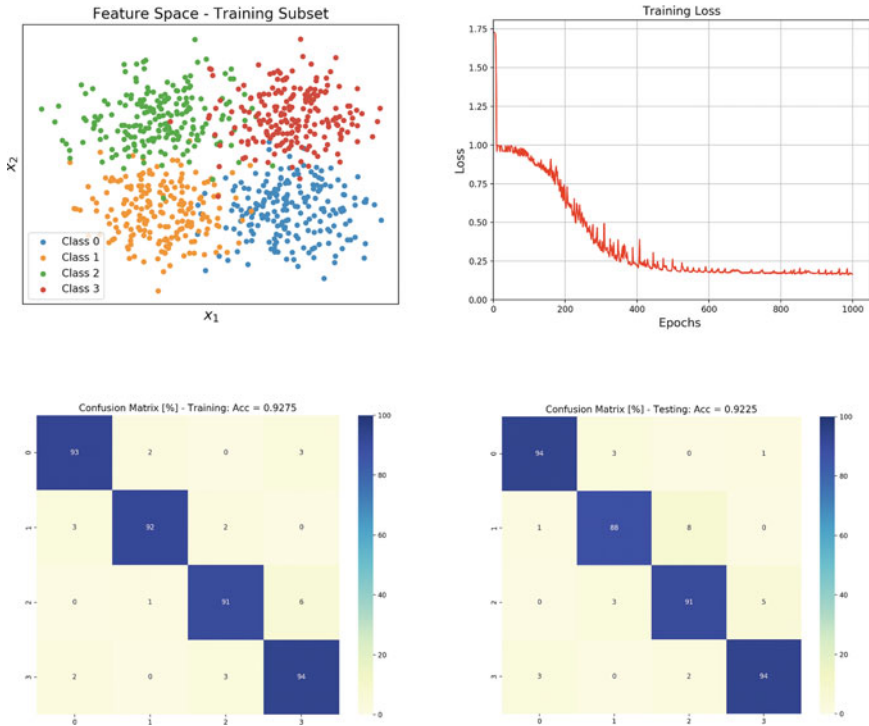


Fig. 7.6 Feature space, loss function, and confusion matrices on training and testing subsets of a four-class problem using a neural network. [→ Example 7.1 📄]

The output of this code is in Fig. 7.6. We can see how the loss function is minimized and how are the samples of each classified (see confusion matrix). In this example, the accuracy in the testing dataset was 92.25%. It is difficult to obtain better results due to the overlapping of the classes. The reader can evaluate the performance of a new network with only one layer with 12 nodes (the line for `nh` definition should be `nh = [12]`). □

The initially developed backward-propagation algorithm used the steepest descent first-order method as the learning rule. Nonetheless, other more powerful optimization approaches are in common use today. The reader is referred to [1] for more training approaches based on gradient descent strategies, like second-order methods, and stochastic methods like Adam [28], etc.



Python Example 7.2: This example is very similar to the previous one [→ Example 7.1] with two hidden layers as shown in Fig. 7.4. However, the implementation is given using `sklearn` library. The reader can study the syntax of Python class `MLPClassifier` for MLP neural networks. The training stage is performed by function `fit` and the prediction that evaluates the trained network on input data is performed by function `predict`. The optimization approach is performed by a *solver* that is in charge of estimate the parameters of the objective function. Similar to previous example, we use two hidden layers (with 6 and 12 nodes) respectively. In `MLPClassifier`, there are three possible solvers: (1) `'lbfgs'` for quasi-Newton methods, (2) `'sgd'` for stochastic gradient descent methods, and (3) `'adam'` stochastic gradient descent method based on Adam approach [28].

Listing 7.2 : Neural network using `sklearn` library.

```
from sklearn.neural_network import MLPClassifier
from pyxvis.io.plots import plot_features2, show_confusion_matrix, plot_loss
from pyxvis.io.data import load_features

# Load training and testing data
(Xtrain, Ytrain, Xtest, Ytest) = load_features('./data/G4/G4')
plot_features2(Xtrain, Ytrain, 'Training+Testing Subsets')

# Definitions
alpha = 1e-5 # learning rate
nh = (6, 12) # nodes of hidden layers
tmax = 2000 # max number of iterations
solver = 'adam' # optimization approach ('lbfgs', 'sgd', 'adam')

# Training
net = MLPClassifier(solver=solver, alpha=alpha, hidden_layer_sizes=nh,
                   random_state=1, max_iter=tmax)
print(Xtrain.shape)
print(Ytrain.shape)
net.fit(Xtrain, Ytrain)

# Evaluation
Ym = net.predict(Xtrain)
show_confusion_matrix(Ym, Ytrain, 'Training')

Ys = net.predict(Xtest)
show_confusion_matrix(Ys, Ytest, 'Testing')
```

The output of this code is the accuracy and confusion matrix evaluated on training and testing data. The results of the confusion matrices are very similar to the results given in the last example (see Fig. 7.6). The reader can evaluate the performance of a new network with only one layer with 12 nodes (the line for `nh` definition should be `nh = (12,)`). \square

Some examples of neural networks in `pyxvis` Library are given in previous chapter (see Examples 6.6, 6.12, 6.13, and 6.14). In Example 6.6, the reader can append `'lr'` to list `ss_cl` to evaluate the performance of a logistic regression in the classification of a two-class problem.

7.3 Convolutional Neural Network (CNN)

There are several deep architectures such as deep neural networks, convolutional neural networks, energy-based models, Boltzmann machines, deep belief networks, among others [3]. CNN (CNN), which were inspired by a biological model [30], is a very powerful method for image recognition [29].

In previous chapters, we studied how an X-ray image \mathbf{X} can be classified: in the control quality of salmons, a region of an X-ray image has a fishbone or not, in baggage inspection, a region of interest shows a knife, a razor blade, a shuriken, and so on. The idea is to extract features of \mathbf{X} and to classify them according to a classification strategy (see Fig. 7.7). In a problem of K classes, the output can be a value $y \in \{1 \dots K\}$ that gives the number of the class, or sometimes the output can be a K -element vector \mathbf{y} , where element y_k gives the probability that the image belongs to class k . If we use a classical neural network to solve the whole problem (representation and classification), the number of parameters to be learned could be so high, that the training process turns completely impractical (see Fig. 7.8). For this reason, CNNs have been developed, in which a strategy of concatenated layers is used (see Fig. 7.9). Using CNN, the number of parameters decreases considerably, the model is trained faster and the classification is more effective.

In this section, we review the basic concepts of CNN, and how a model is trained and tested. Finally, we give an example that can be used in the automated detection of casting defects (Fig. 7.10).

7.3.1 Basics of CNN

An X-ray testing method based on CNNs can be used to recognize an object of interest in an X-ray image. For example, we can have a region of interest \mathbf{X} of an X-ray image of a casting to determine if this region has a defect or not. In this case, the CNN replaces feature extraction and classification with a single neural network.

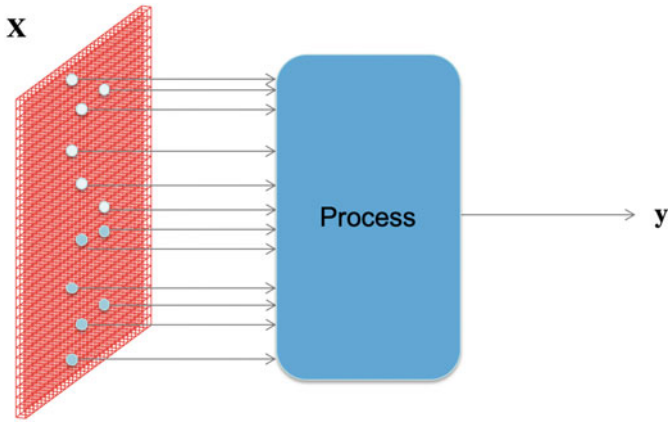


Fig. 7.7 An image X classified as vector y after a pattern recognition approach where features are extracted and classified using a classification strategy

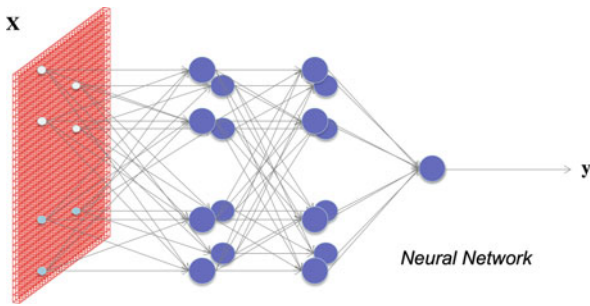


Fig. 7.8 Using a classical neural network approach (as shown in Fig. 7.4), every pixel of an image X can be connected to a node of a neural network of some hidden layers, however, the numbers of parameters of this architecture can be prohibited (in the first layer it could be N^4 connections for a $N \times N$ -pixel input image) and a layer with $N \times N$ nodes

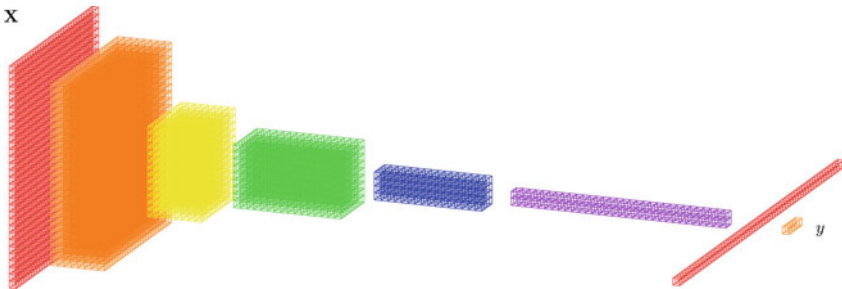


Fig. 7.9 Image classification using a convolutional neural network (CNN): concatenation of layers

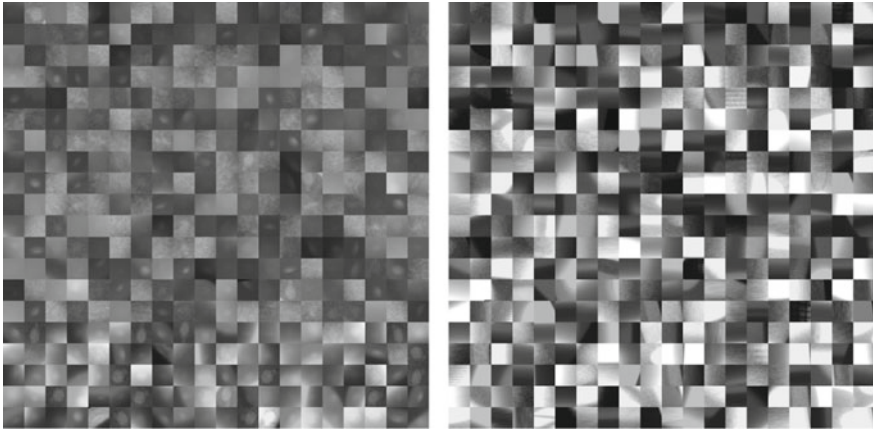


Fig. 7.10 Example of defects (lefts) and no-defects (right). It is clear that there are some patterns that can be easily detected (see, for example, defects that are bright bubbles with dark background and no-defects that are regular structures with edges), however, the recognition of both classes can be very difficult for low-contrast defects because they are very similar to homogenous no-defects

Thus, the CNN maps an input image \mathbf{X} onto an output vector \mathbf{y} of K elements, for K classes:

$$\mathbf{y} = \mathcal{F}_L(\mathbf{X}, \mathbf{w}). \quad (7.22)$$

Typically, element y_k gives the probability that image \mathbf{X} belongs to class k . In our example, $K = 2$ (for two classes: defects and no-defects), and image \mathbf{X} will be classified as defect if $y_1 > y_2$. Function \mathcal{F}_L can be viewed as feed-forward network with L linear and non-linear layers f_l , for $l = 1 \dots L$. The functions contain parameters

$$\mathbf{w} = (\mathbf{w}_1, \dots, \mathbf{w}_L) \quad (7.23)$$

that can be discriminatively learned from training data: a set of input images \mathbf{X}_i and their corresponding labels \mathbf{z}_i , for $i = 1, \dots, n$, so that

$$\sum_i f_{\text{loss}}(\mathcal{F}_L(\mathbf{X}_i, \mathbf{w}), \mathbf{z}_i) / n \rightarrow \min. \quad (7.24)$$

Ideally, for an input of training data (\mathbf{X}_i) the output of the network ($\mathbf{y}_i = \mathcal{F}_L(\mathbf{X}_i, \mathbf{w})$) should be the the corresponding label (\mathbf{z}_i). Thus, f_{loss} is defined as *loss function* that gives a measurement of the error of the classification. This optimization problem can be solved using the backward-propagation approach [1, 18].

A method based on CNN can be understood as a set of L layers. Layer l (for $l = 1 \dots L$), is a function \mathbf{f}_l (with parameters \mathbf{w}_l) that processes an input image \mathbf{X}_{l-1} in order to obtain an output image \mathbf{X}_l (see Fig. 7.11):

$$\mathbf{X}_l = \mathbf{f}_l(\mathbf{X}_{l-1}, \mathbf{w}_l), \quad (7.25)$$

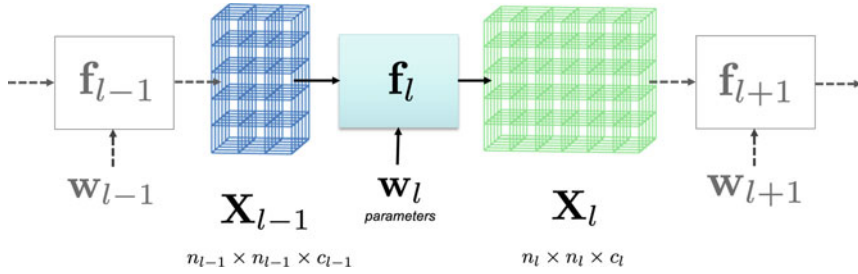


Fig. 7.11 Structure of layer l of a CNN according to (7.25): input image \mathbf{X}_{l-1} is transformed into output image \mathbf{X}_l using function f_l with parameters \mathbf{w}_l

where $\mathbf{X}_0 = \mathbf{X}$ is the input image of the whole CNN that we want to recognize. In our case, \mathbf{X}_0 is a grayscale X-ray image,⁵ for this reason, the number of channels is one, $c_0 = 1$.

In our example, the input image corresponds to a cropped image of (e.g., 32×32 pixels as illustrated in Fig. 7.10 for two classes). In this CNN, the output of a layer is the input of the next layer. Thus, the output of each layer of the CNN can be defined as follows:

$$\mathbf{X}_l = \mathcal{F}_l(\mathbf{X}, \mathbf{w}) = f_l(f_{l-1}(\dots f_1(\mathbf{X}, \mathbf{w}_1), \dots, \mathbf{w}_{l-1}), \mathbf{w}_l), \quad (7.26)$$

that is a concatenation of l functions $f_1 \dots f_l$. Without loss of generality, we will assume that the images are square, where the height and the width are n_l pixels. The images have one or more channels, i.e., image \mathbf{X}_l is a 3D data structure with $n_l \times n_l \times c_l$ pixels, where c_l is the number of channels. Channel k of \mathbf{X}_l is a matrix of $n_l \times n_l$ elements, and it is denoted as $\mathbf{X}_{k,l}$, for $k = 1 \dots c_l$. The key idea of the CNN is that the output of the last layers correspond to high-level representations of the input image \mathbf{X} . These representations can be used in a classification process to recognize automatically the class of \mathbf{X} .

There are several types of layers that are normally used in CNN. Typically, the used layers are: convolution layer, pooling layer, rectified linear unit, and fully connected layer. They will be explained in further details.

• **Convolution Layer [conv]:** This layer corresponds to a linear convolution of input image \mathbf{X}_{l-1} with a bank filter \mathbf{F}_l and a bias \mathbf{b}_l . The filter bank \mathbf{F}_l consists of a set of m_l 3D filters $\mathbf{F}_{k,l}$ of $p_l \times p_l \times q_l$ elements and a bias $b_{k,l}$ for $k = 1 \dots m_l$. The parameters \mathbf{w}_l of this layer are the elements of \mathbf{F}_l and \mathbf{b}_l . Therefore, the number of parameters of each filter is $p_l \times p_l \times q_l + 1$, that means that the filter bank of layer l has $m_l(p_l^2 q_l + 1)$ parameters. These parameters are to be estimated in a learning process (as shown in Sect. 7.3.2). It is worth noting that the number of channels of the filter bank is the number of channels of the input image ($q_l = c_{l-1}$), and the

⁵For an X-ray image with pseudocolors, the number of channels of the input image can be three, $c_0 = 3$.

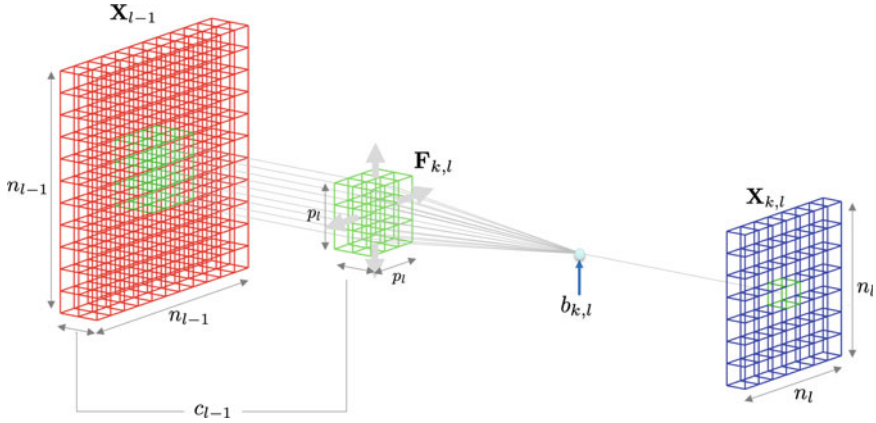


Fig. 7.12 Linear convolution: channel k of the output image \mathbf{X}_l is computed as the convolution of the input image \mathbf{X}_{l-1} with a filter $\mathbf{F}_{k,l}$ adding a bias $b_{k,l}$. In this example, $c_{l-1} = 2$ and $p_l = 3$

number of channels of the output image is the number of filters of the filter bank ($c_l = m_l$). Thus, the output for filter k is channel k of image \mathbf{X}_l :

$$\mathbf{X}_{k,l} = \mathbf{X}_{l-1} * \mathbf{F}_{k,l} + b_{k,l} \quad \text{for } k = 1 \dots m_l, \tag{7.27}$$

where ‘*’ denotes the convolution operator. In other words, pixel (i, j) of channel k of the output image \mathbf{X}_l is

$$X_{k,l}(i, j) = b_{k,l} + \sum_{u=1}^{p_l} \sum_{v=1}^{p_l} \sum_{w=1}^{q_l} X_{w,l-1}(i + u, j + v) F_{k,l}(u, v, w) \tag{7.28}$$

That means, that the size of the output image \mathbf{X}_l will be a 3D matrix of $n_l \times n_l \times c_l$, with $n_l = n_{l-1} - p_l + 1$. The filtering process (for one channel of the output image) is illustrated in Fig. 7.12.

• **Pooling Layer [pool]:** This process is independently performed for each channel of input image \mathbf{X}_{l-1} . Therefore, the number of channels of input and output images are the same ($c_l = c_{l-1}$). In this case, the size of the image is reduced by representing a region of a channel with a scalar value. The output for each channel is defined as

$$X_{k,l}(i, j) = f_{\text{pool}}\{X_{k,l-1}(u, v) : (u, v) \in \Omega(i, j)\}. \tag{7.29}$$

Typically, the set of pixels $\Omega(i, j)$ is a sub-window of $\mathbf{X}_{k,l-1}$ of size $p_l \times p_l$ pixels which first pixel corresponds to the pixel (i, j) as illustrated in Fig. 7.13. The f_{pool} function can be the maximum, the mean, the ℓ^2 norm, etc. In our approach, we use the maximum operator, known as ‘max-pooling’, with no overlap, that means, each

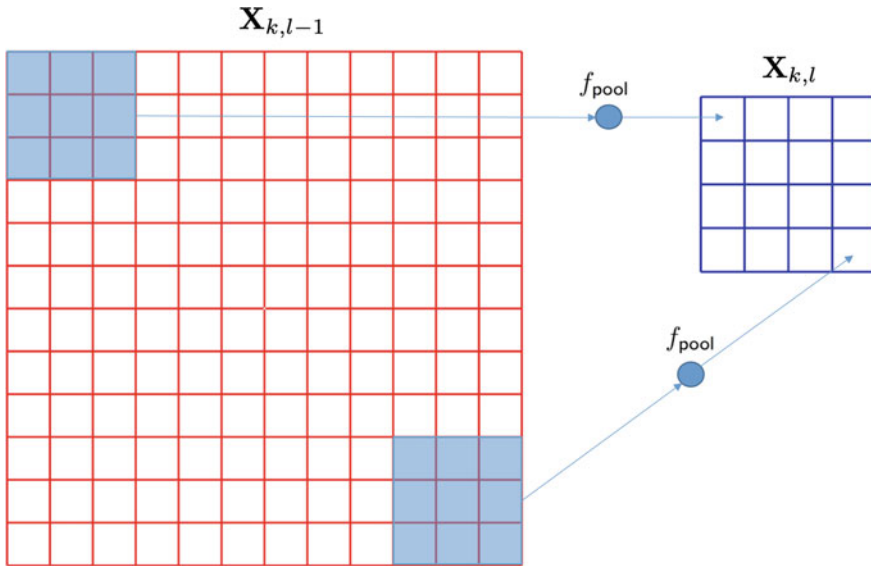


Fig. 7.13 Pooling. In this example, the dimension $n_{l-1} \times n_{l-1}$ of channel k of input image X_{l-1} is 12×12 , and the dimension $n \times n$ of the neighborhood Ω is 3×3 . Hence, the size of channel k of the output image X_l is 4×4 , i.e., $n_l = 4$. Function f_{pool} could be in this example the maximum

channel is down-sampled non-linearly. Therefore, the use of this layer can efficiently reduce the computational time for upper layers.

- **Rectified Linear Unit [relu]:** Similar to pooling layer, this process is independently performed for each channel of input image X_{l-1} ($c_l = c_{l-1}$). In this case, the information of X_{l-1} is rectified by setting to zero all negative values. The key idea of a ReLU layer is to produce more discriminative representations avoiding negative scores [45]. Thus,

$$X_{k,l}(i, j) = \max\{0, X_{k,l-1}(i, j)\}. \tag{7.30}$$

The ReLU process is illustrated in Fig. 7.14 for channel k .

- **Fully Connected Layer [fc]:** This layer corresponds to a classic layer in a neuronal network (multi-layer perceptron), in which each output of previous layer is connected to new layer as explained in Sect. 7.2 and shown in Fig. 7.4: that means, each input node of a fully connected layer is the weighted sum of all outputs of previous layer plus a bias, and the output is this result after an activation function (see Fig. 7.15). The output is considered as a vector of n_l elements. Thus, if input layer has $n_{l-1} \times n_{l-1} \times c_{l-1}$, then there are $n_l \times n_{l-1}^2 \times c_{l-1}$ weights and n_l bias parameters that must be learned.

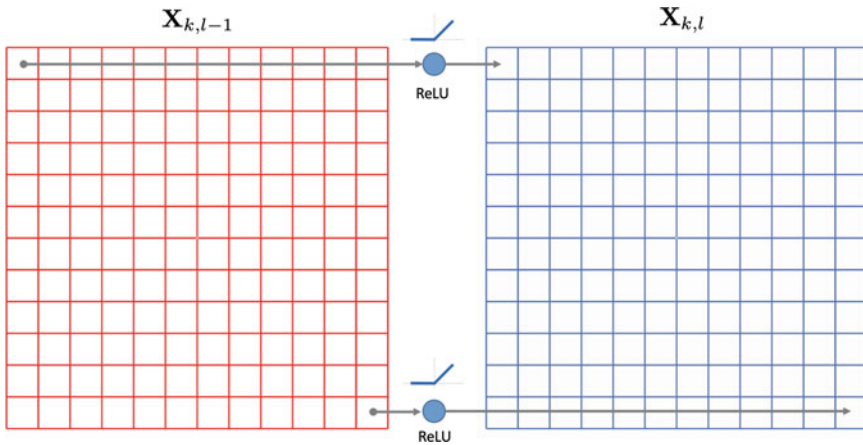


Fig. 7.14 Rectified linear unit. In this example, channel k of input image \mathbf{X}_l is rectified. Channel k of the output image \mathbf{X}_{l+1} has the same dimension: 6×6

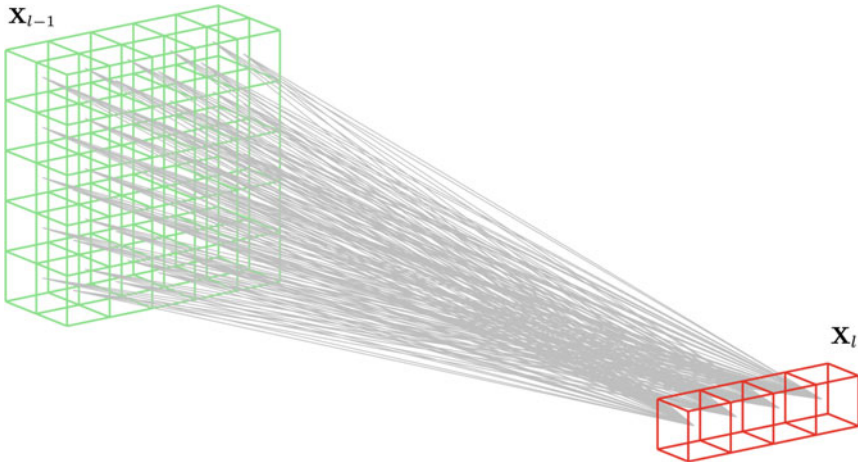


Fig. 7.15 Fully connected layer: all outputs of previous layer are connected one to one to the next layer. In this example, the input (green) and output (red) layers have $5 \times 5 \times 2$ and 4 pixels respectively that means, there are $5 \times 5 \times 2 \times 4 = 200$ connections (see gray lines)

In [39], a CNN model called Xnet is proposed to detect defects in aluminum castings (in Sect. 7.3.4, a similar example is given in Python).⁶ The whole CNN is shown in Fig. 7.16. It includes a dropout block (dropout) that randomly turns off connections of the neural network during training. It has been shown that this technique reduces significantly the overfitting [59]. Typically, in a CNN model, layer $L - 1$ corresponds to a vector \mathbf{s} with K elements, $[s_1 \dots s_K]$:

⁶Another use of CNN in defects detection in castings can be found in [62].

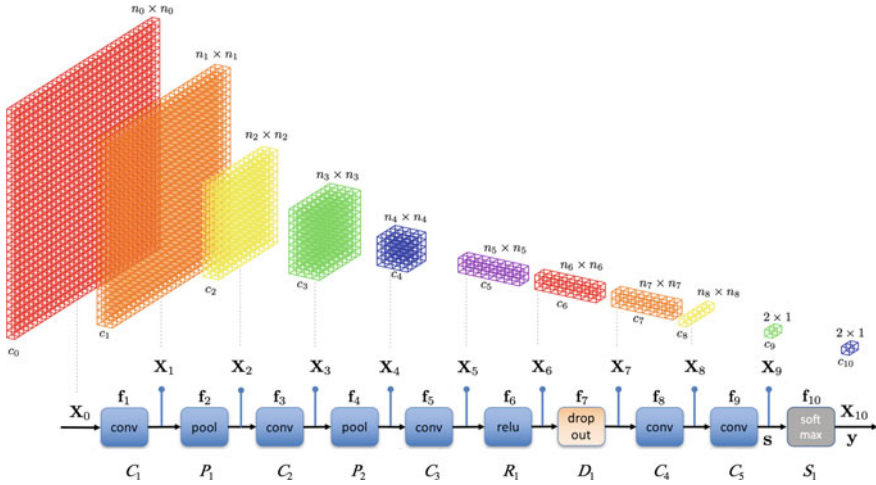


Fig. 7.16 XNet: CNN architecture proposed for automated detection of defects in castings [39]

$$\mathbf{s} = \mathbf{X}_{L-1} \tag{7.31}$$

In the detection of defects, $K = 2$ because there are only two classes: defects and no-defects. In this approach, the output layer (layer L) is a ‘softmax’ block that is used to convert the scores of \mathbf{s} into probabilities. Thus, $\mathbf{X}_L = \mathbf{y} = [y_1 \dots y_K]^T$, where

$$y_k = f_{\text{softmax}}(s_k) = \frac{e^{s_k}}{\sum_{j=1}^K e^{s_j}} \text{ for } k = 1 \dots K. \tag{7.32}$$

Using (7.26), it is clear that

$$\mathbf{y} = \mathcal{F}_L(\mathbf{X}, \mathbf{w}) \text{ and } \mathbf{s} = \mathcal{F}_{L-1}(\mathbf{X}, \mathbf{w}). \tag{7.33}$$

Table 7.1 summarizes Xnet [39], where the input image $\mathbf{X}_0 = \mathbf{X}$ is an image of $32 \times 32 \times 1$ pixels. The CNN consists of ten layers with five linear convolutional layers (C_1, \dots, C_5), two pooling layers with maximum operator (P_1, P_2), one ReLU layer (R_1), one dropout layer (D_1), and one softmax layer (S_1). As we can see in Table 7.1, our CNN has 5.7×10^5 parameters that must be estimated in a learning stage.

7.3.2 Learning in CNN

As we mentioned in previous section, CNN maps an input image \mathbf{X} on an output vector $\mathbf{y} = \mathcal{F}_L(\mathbf{X}, \mathbf{w})$, where function \mathcal{F}_L can be viewed as a sequence of linear

Table 7.1 Convolutional neural network Xnet [39]

Layer		Function f_l			Output \mathbf{X}_l
l	Name	Type	$m_l(p_l \times p_l \times q_l)$	Parameters	$n_l \times n_l \times c_l$
0	Input	–	–	–	$32 \times 32 \times 1$
1	C_1	conv	$64(7 \times 7 \times 1)$	3.200	$26 \times 26 \times 64$
2	P_1	pool-max	2×2	0	$13 \times 13 \times 64$
3	C_2	conv	$128(5 \times 5 \times 64)$	204.928	$9 \times 9 \times 128$
4	P_2	pool-max	2×2	0	$4 \times 4 \times 128$
5	C_3	conv	$256(3 \times 3 \times 128)$	295.168	$2 \times 2 \times 256$
6	R_1	relu	–	0	$2 \times 2 \times 256$
7	D_1	dropout	–	0	$2 \times 2 \times 256$
8	C_4	conv	$64(2 \times 2 \times 256)$	65.600	$1 \times 1 \times 64$
9	C_5	conv	$2(1 \times 1 \times 64)$	130	$1 \times 2 \times 1$
10	S_1	softmax	–	0	$1 \times 2 \times 1$
Total				569.026	

and non-linear functions f_1, \dots, f_L , that depend on parameters $\mathbf{w} = (\mathbf{w}_1, \dots, \mathbf{w}_L)$ as defined in (7.26) and (7.33).

Learning consists of estimating parameters \mathbf{w} from ‘learning data’. The output of this process is the set of parameters \mathbf{w} . On the other hand, testing is used to evaluate the performance of the trained model on ‘testing data’, i.e., the learned model (with fixed parameters \mathbf{w}) is used to classify new data. The output of this process is the classification of each testing sample. For this end, a set of annotated X-ray images is available. Thus, for each image, \mathbf{X} , vector \mathbf{z} —the ground truth of the classification—is given by an expert. Similar to vector \mathbf{y} , the output of the CNN, vector \mathbf{z} has K elements. The value z_k is ‘1’ if image \mathbf{X} belongs to class k , otherwise z_k is ‘0’.

In order to reduce the computation time of learning process, typically, a hold-out protocol is used. The standard hold-out evaluation protocol is based on disjoint learning and testing data, i.e., images that are present in the learning set are not allowed to be in the testing set. We denote the X-ray images and their labels (\mathbf{X}, \mathbf{z}) as

- Learning: $\{\mathbf{X}_{\text{learn}}^{(i)}, \mathbf{z}_{\text{learn}}^{(i)}\}_{i=1}^{n_{\text{learn}}}$, with n_{learn} learning samples.
- Testing: $\{\mathbf{X}_{\text{test}}^{(i)}, \mathbf{z}_{\text{test}}^{(i)}\}_{i=1}^{n_{\text{test}}}$, with n_{test} testing samples.

The learning set is subdivided into two disjoint subsets: training set $(\mathbf{X}_{\text{train}}^{(i)}, \mathbf{z}_{\text{train}}^{(i)})$, for $i = 1, \dots, n_{\text{train}}$, and validation set $(\mathbf{X}_{\text{val}}^{(i)}, \mathbf{z}_{\text{val}}^{(i)})$, for $i = 1, \dots, n_{\text{val}}$, with $n_{\text{learn}} = n_{\text{train}} + n_{\text{val}}$. Typically, 75–80% of the learning data for training and 25–20% for validation.

The training data is used to estimate the parameters \mathbf{w} of our model as follows. The output of the CNN is $\mathbf{y}_{\text{train}}^{(i)} = \mathcal{F}_L(\mathbf{X}_{\text{train}}^{(i)}, \mathbf{w})$. It is clear, that ideally $\mathbf{y}_{\text{train}}^{(i)}$ should be $\mathbf{z}_{\text{train}}^{(i)}$. Thus, parameters \mathbf{w} can be estimated by minimizing the objective function:

$$e_{\text{train}} = \frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} f_{\text{loss}}(\mathbf{y}_{\text{train}}^{(i)}, \mathbf{z}_{\text{train}}^{(i)}), \quad (7.34)$$

where f_{loss} is a loss function. This optimization problem can be iteratively solved using the backward-propagation approach [1, 18] as explained for a classic neural network in Sect. 7.2: We start with initial random values $\mathbf{w}_{(0)}$ for the first iteration, and the parameters in epoch j are estimated according to the parameters in previous epoch $j - 1$ and an incremental update:

$$\mathbf{w}_{(j)} = \mathbf{w}_{(j-1)} + \Delta \mathbf{w}_{(j)}. \quad (7.35)$$

For each epoch of the training process, a new version of the parameters $\mathbf{w}_{(j)}$ is estimated. For validation purposes, the subset $(\mathbf{X}_{\text{val}}^{(i)}, \mathbf{z}_{\text{val}}^{(i)})$ for $i = 1, \dots, n_{\text{val}}$ is used. The error

$$e_{\text{val}} = \frac{1}{n_{\text{val}}} \sum_{i=1}^{n_{\text{val}}} f_{\text{loss}}(\mathbf{y}_{\text{val}}^{(i)}, \mathbf{z}_{\text{val}}^{(i)}), \quad (7.36)$$

is computed, with $\mathbf{y}_{\text{val}}^{(i)} = \mathcal{F}_L(\mathbf{X}_{\text{val}}^{(i)}, \mathbf{w})$, where \mathcal{F}_L is evaluated using $\mathbf{w} = \mathbf{w}_{(j)}$. At the beginning of the training, both errors e_{train} and e_{val} usually decrease. Nevertheless, when the learning process starts overfitting, the error e_{val} starts to increase. This epoch will be denoted by j^* . Thus, the training process is stopped when e_{val} is minimum, and our parameter vector will be $\mathbf{w} = \mathbf{w}_{(j^*)}$.

7.3.3 Testing in CNN

After learning stage, we can test the CNN using the testing dataset: $(\mathbf{X}_{\text{test}}^{(i)}, \mathbf{z}_{\text{test}}^{(i)})$ for $i = 1, \dots, n_{\text{test}}$. The images of this dataset were not used in the learning stage. There are several approaches that can be used to classify $\mathbf{X}_{\text{test}}^{(i)}$. Obviously, one of them is to use the representation of the last layer:

$$\text{class}(\mathbf{X}_{\text{test}}^{(i)}) = \underset{k}{\text{argmax}} \left\{ \mathbf{y}_{\text{test}}^{(i)}(k) \right\} \quad (7.37)$$

where $\mathbf{y}_{\text{test}}^{(i)}(k)$ is the k th element of $\mathbf{y}_{\text{test}}^{(i)} = \mathcal{F}_L(\mathbf{X}_{\text{test}}^{(i)}, \mathbf{w})$.

In addition, the high-level representations that are present in layers $l < L$ can be used in a classification approach as well. For this purpose, a descriptor \mathbf{d} can be

defined as a vector of $n_l^2 c_l \times 1$ elements that contains all elements of \mathbf{X}_l by stacking its columns:

$$\mathbf{d} = s(\mathbf{X}_l, \mathbf{w}) = s(\mathcal{F}_l(\mathbf{X}, \mathbf{w})), \quad (7.38)$$

where $s(\cdot)$ is the stack function. Descriptor \mathbf{d} can be used to train another kind of classifier (a KNN, for example). In [39], the best results were obtained using $\mathbf{d} = s(\mathbf{X}_7)$, i.e., $l = L - 3$.

A classifier h can be designed using the descriptors and the labels of the learning set. Thus, the classifier can be learned using $(\mathbf{d}_{\text{learn}}^{(i)}, \mathbf{z}_{\text{learn}}^{(i)})$, where $\mathbf{d}_{\text{learn}}^{(i)} = s(\mathbf{X}_{\text{learn}}^{(i)})$ for $i = 1 \dots n_{\text{learn}}$ according to (7.38). After training, $h(\mathbf{d}_{\text{learn}}^{(i)})$ should ideally be $\mathbf{z}_{\text{learn}}^{(i)}$.

7.3.4 Example of CNN



Python Example 7.3: In this example, we test a very simple CNN architecture for the detection of casting defects. The dataset used in this example, called **C1**, is a subset of the dataset used in [39].⁷ It contains the *easiest* patches of the original dataset, i.e., those patches that are easy to classify. In this example, there are 8200 patches of 32×32 pixels, 80% for testing and 20% for testing (both subsets with 50% defects and 50% no-defects). The idea of this example is to train an easy dataset with a simple CNN architecture. Thus, the reader in less than 2 min (with around 20–30 epochs) can have a trained model with excellent performance. After this training, it is possible to train more complex architectures with more challenging datasets. In this example, we define the CNN architecture using **CNN** of pyxvis Library. In this definition, there are n typical 2D convolutional blocks of layers with Keras⁸ functions **Conv2D** with a ReLU activation, **BatchNormalization**, **MaxPooling2D**, and **Dropout** with a rate of 25%. The 2D convolution of block i , for $i = 1, \dots, n$, is with d_i kernels of $p_i \times p_i$ pixels. In our example, we define the three blocks using variables $\mathbf{p} = [7, 5, 3]$ and $\mathbf{d} = [4, 12, 8]$. After the n blocks of convolutional layers, we add m fully connected layers, each layer has f_j elements. In our case, $m = 1$ and $\mathbf{f} = [12]$. If we want to have two fully connected layers, one with 12 elements and another with 4, we define $\mathbf{f} = [12, 4]$. Finally, **CNN** of pyxvis Library includes a fully connected layer of the number of classes to be recognized (in our case is 2), and a ‘softmax’ block. In Fig. 7.17, we can see the architecture.

⁷The original dataset has 47.520 patches, and it can be downloaded from <https://domingomery.ing.puc.cl/material/>.

⁸Keras is a library built on top of TensorFlow. It consists of a set of API functions written in Python for building deep learning models. See <https://keras.io>.

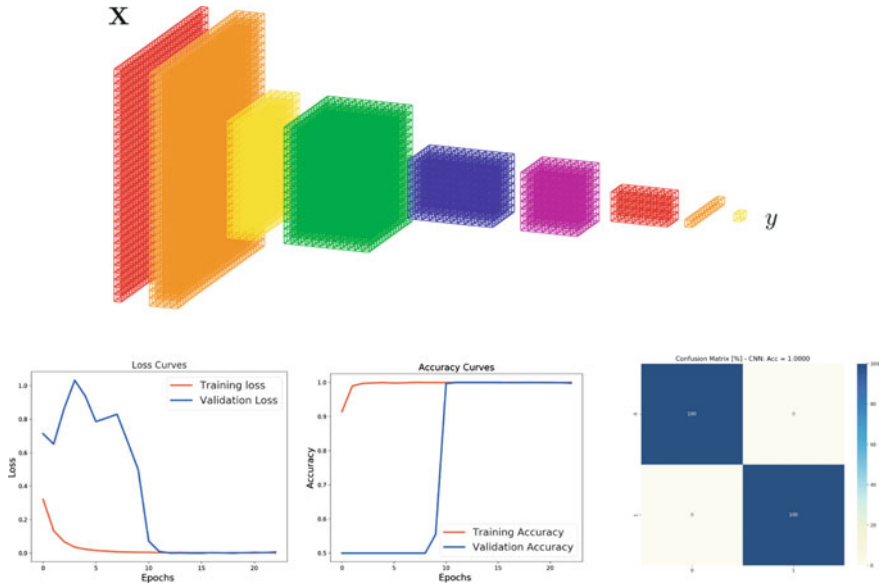


Fig. 7.17 CNN architecture, learning curves, and confusion matrix. In this example, all defects and no-defects from testing subset are correctly classified. [→ Example 7.3 🧩]

Listing 7.3 : Convolutional Neural Network.

```

from pyxvis.learning.cnn import CNN

# execution type
type_exec = 0 # training & testing

# patches' file for training and testing
patches_file = '../data/C1/C1'

# architecture
p = [7,5,3] # Conv2D mask size
d = [4,12,8] # Conv2D channels
f = [12] # fully connected

# training and testing
CNN(patches_file,type_exec,p,d,f)

```

The first output of this code is the definition of the architecture (that corresponds to the diagram of Fig. 7.17):

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 4, 32, 32)	200
batch_normalization_1 (Batch Normalization)	(None, 4, 32, 32)	128

max_pooling2d_1	(MaxPooling2 (None, 4, 16, 16))	0
dropout_1	(Dropout) (None, 4, 16, 16)	0
conv2d_2	(Conv2D) (None, 12, 16, 16)	1212
batch_normalization_2	(Batch (None, 12, 16, 16))	64
max_pooling2d_2	(MaxPooling2 (None, 12, 8, 8))	0
dropout_2	(Dropout) (None, 12, 8, 8)	0
conv2d_3	(Conv2D) (None, 8, 8, 8)	872
batch_normalization_3	(Batch (None, 8, 8, 8))	32
max_pooling2d_3	(MaxPooling2 (None, 8, 4, 4))	0
dropout_3	(Dropout) (None, 8, 4, 4)	0
flatten_1	(Flatten) (None, 128)	0
dense_1	(Dense) (None, 12)	1536
batch_normalization_4	(Batch (None, 12))	48
activation_1	(Activation) (None, 12)	0
dropout_4	(Dropout) (None, 12)	0
dense_2	(Dense) (None, 2)	26
activation_2	(Activation) (None, 2)	0
=====		
Total params: 4,118.0		
Trainable params: 3,982.0		
Non-trainable params: 136.0		

The architecture of our model has almost 4000 trainable parameters. The learning curves and the confusion matrix are shown in Fig. 7.17. We can see that in this very simple example, the accuracy is 100% (a perfect detection). The reader that wants to try a more difficult example can download the dataset of [39] (see footnote 7)

and implement a network similar to Xnet (see Table 7.1). In the original code of Example 7.3, the lines of the new code should be:

```
# execution type
type_exec      = 0 # training & testing

# patches' file for training and testing
patches_file   = 'wacv_castings.mat'

# architecture
p = [7,5,3]     # Conv2D mask size
d = [64,128,256] # Conv2D channels
f = [64,32]     # fully connected

# training and testing
CNN(patches_file,type_exec,p,d,f)
```

For this dataset, the model is trained after 1 h (and 30 epochs)> The achieved accuracy is 87.78% very similar to the reported accuracy in [39]. □

7.4 Pre-trained Models

Pre-trained models are deep learning models that have been already trained on large datasets of one domain and can be used as-is on other domains with no additional training. In this section, we explain how to use pre-trained models in X-ray testing.

7.4.1 Basics of Pre-trained Models

In X-ray testing, it is possible to use sophisticated models that have been already trained on other domains (e.g., recognition of common objects in color images). The idea is to use part of the trained model on new domains, such as X-ray images. One of the most popular datasets of color images of common objects is ImageNet [55]. ImageNet consists of an annotated collection of color images of very common objects (like cars, bicycles, trucks, cat, dogs, etc.). ImageNet has 1000 classes of objects with approx. 1000 images per class for training purposes. The dataset has been widely used in competitions of object recognition algorithms. The trained models are typically available as open-source models. The architecture of these models has many layers, and the last one corresponds to a structure of 1000 elements that are used to distinguish the 1000 classes. In the testing stage, if an input image contains

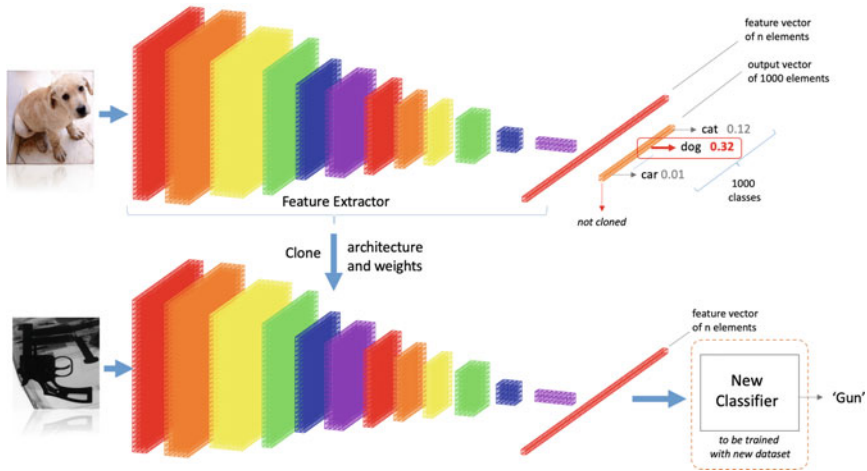


Fig. 7.18 Classification strategy using a pre-trained model

a dog, the output, i.e., a vector with 1000 element, should have the element corresponding to the class 'dog' the maximal value as illustrated in Fig. 7.18-Top. This pre-trained model can be used not only to recognize images that content objects that belong to these 1000 classes but also to recognize other objects. The key idea of this approach is as follows: The last layer of the pre-trained models is not used because it has been trained to recognize objects that do not belong to the new domain. The last layer is replaced by a new fully connected layer or a simple classifier, such as KNN or a SVM, that is designed to classify the classes of the new domain.

The strategy behind this idea is that the pre-trained model should extract in the first layers relevant visual information of the input image and could give us a good representation of the images of the new domain.

To illustrate this idea, we present now a well-known approach that shows what happens inside the layers of a CNN [13, 67]. This approach can visualize how the images are represented in the network and give insight into the layers. It consists of the estimation of a synthetic input image that maximizes the activation of a certain element (pixel) of a layer.

Thus, we can visualize what kind of images activates each element of our network. The estimation is an optimization problem, that starts with a random input image, and after some iterations, the solution converges using a gradient descent algorithm.⁹ In this process, the weights of the CNN are fixed, i.e., we do not train the CNN, we only find an input image that maximizes a certain pixel of a layer of the CNN. To illustrate this visualization, we use a pre-trained CNN called VGG16 [58]. The architecture of VGG16 is the following:

⁹An implementation of this idea can be found in https://keras.io/examples/conv_filter_visualization/. The Figs. 7.19 and 7.23 were done using this implementation.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, None, None, 3)	0
block1_conv1* (Conv2D)	(None, None, None, 64)	1792
block1_conv2 (Conv2D)	(None, None, None, 64)	36928
block1_pool (MaxPooling2D)	(None, None, None, 64)	0
block2_conv1 (Conv2D)	(None, None, None, 128)	73856
block2_conv2 (Conv2D)	(None, None, None, 128)	147584
block2_pool (MaxPooling2D)	(None, None, None, 128)	0
block3_conv1* (Conv2D)	(None, None, None, 256)	295168
block3_conv2 (Conv2D)	(None, None, None, 256)	590080
block3_conv3 (Conv2D)	(None, None, None, 256)	590080
block3_pool (MaxPooling2D)	(None, None, None, 256)	0
block4_conv1* (Conv2D)	(None, None, None, 512)	1180160
block4_conv2 (Conv2D)	(None, None, None, 512)	2359808
block4_conv3 (Conv2D)	(None, None, None, 512)	2359808
block4_pool (MaxPooling2D)	(None, None, None, 512)	0
block5_conv1 (Conv2D)	(None, None, None, 512)	2359808
block5_conv2* (Conv2D)	(None, None, None, 512)	2359808
block5_conv3 (Conv2D)	(None, None, None, 512)	2359808
block5_pool (MaxPooling2D)	(None, None, None, 512)	0

Table 7.2 Images of $\mathbb{GDXray+}$ used in our experiments

Set		Gun	Shuriken	Blade	Others
Training	Series	B0049	B0050	B0051	B0078
	Images	1–200	1–100	1–100	1–500
Validation	Series	B0079	B0080	B0081	B0082
	Images	1–50	1–50	1–50	1–200
Testing	Series	B0079	B0080	B0081	B0082
	Images	51–150	51–150	51–150	201–600

VGG16 was developed in the year 2014 and it is one of the most powerful CNN architecture for vision problems.¹⁰ It consists of five blocks with `conv` and `pool-max` layers (the last fully connected layers are not given in the previous description). In Fig. 7.19, we show some images that activate certain elements of the layers that have a ‘*’ in this description, i.e., `block1_conv1`, `block3_conv1`, `block4_conv1`, and `block5_conv2`. In this case, VGG16 was trained for ImageNet [55]. We can observe in this figure, the complexity of the generated patterns: the more complex is the image, the deeper is the layer. Moreover, for the last layers, we can recognize some patterns like birds and feathers!

The idea of using pre-trained models is simple and powerful as we will show in our examples. Simple because the weights of the pre-trained models are available in public repositories or deep learning libraries (like Keras (see footnote 8)) and powerful because good results can be achieved with no implementation difficulty.

7.4.2 Example of Pre-trained Models

In this section, we show how to use pre-trained models in the recognition of threat objects (in baggage inspection). In our experiments, there are three objects: hand-guns, shuriken (ninja stars), and razor blades. Each category of objects defines a class (Gun, Shuriken, and Blade). Furthermore, there is a fourth class called Other for other objects and background. All X-ray images used in our experiments belong to the $\mathbb{GDXray+}$. As shown in Table 7.2, there are three different sets of images: training, testing, and validation sets. For training, X-ray images of $\mathbb{GDXray+}$ series B0049, B0050, B0051, and B0078 must be used for classes Gun, Shuriken, Blade, and Others respectively. For validation, in case that a method has some parameters to be tuned, it is allowed to use the first 50 images of $\mathbb{GDXray+}$ series B0079, B0080, and B0081 for Gun, Shuriken, and Blade respectively and the first 200 images of folder B0082 for Others. For testing, the last 100 images of $\mathbb{GDXray+}$ series B0079, B0080 and B0081 for Gun, Shuriken, and Blade respectively and the last 400 images of folder B0082 for Others have to be used.

¹⁰See an application in the automated weld defect recognition based on VGG16 in [34].

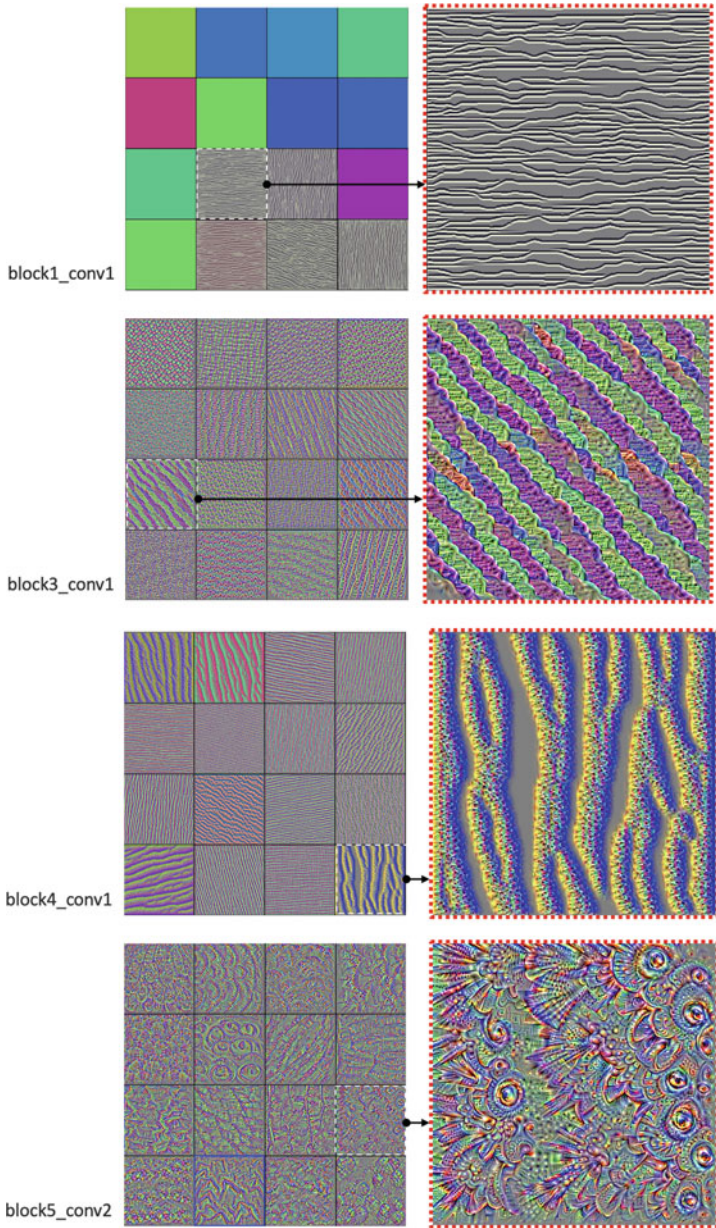


Fig. 7.19 Generated input images that maximizes the activation of 16 elements (pixels) of some layers in VGG16. A zoom of image with a blue square is presented in Fig. 7.19

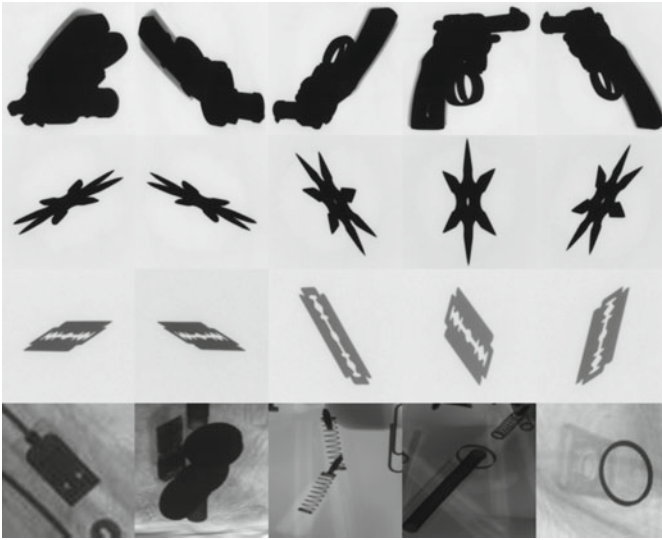


Fig. 7.20 Some training X-ray images used in our experiments. Each row represents a labeled class (handguns, shuriken, razor blades, and others respectively)

The $\text{GD}\text{Xray}+$ dataset is especially challenging due to the high intra-class variability between training and testing images of positive classes (see some examples for guns, shuriken, and razor blades in Figs. 7.20 and 7.21 for training and testing respectively). Indeed, training images of positive classes contain just the object with a clean background. In contrast, testing images corresponding to these classes show a noisy background that may allow any discriminative model to classify them as the class Others.

In our example, we follow the experimental protocols defined in [41] for two recognition tasks:

- Four-class Classification:** In the first task, we have to design a classifier that is able to recognize the four mentioned classes: (1) Gun, (2) Shuriken, (3) Blade, and (4) Others. We define $K = 4$ as the number of classes. The classifier has to be trained using the trained data. The parameters of the classifier (if any) can be tuned using the validation only. The performance of the method is reported using the testing data as follows: The elements of the $m \times m$ confusion matrix are defined as $C(i, j)$ for $i = 1 \dots K$ and $j = 1 \dots K$, where $C(i, j)$ means the number of images of class i (in the testing data) classified as class j . The accuracy of each class is defined as

$$\eta_i = \frac{C(i, i)}{\sum_{j=1}^4 C(i, j)}. \quad (7.39)$$

The total accuracy is the average:

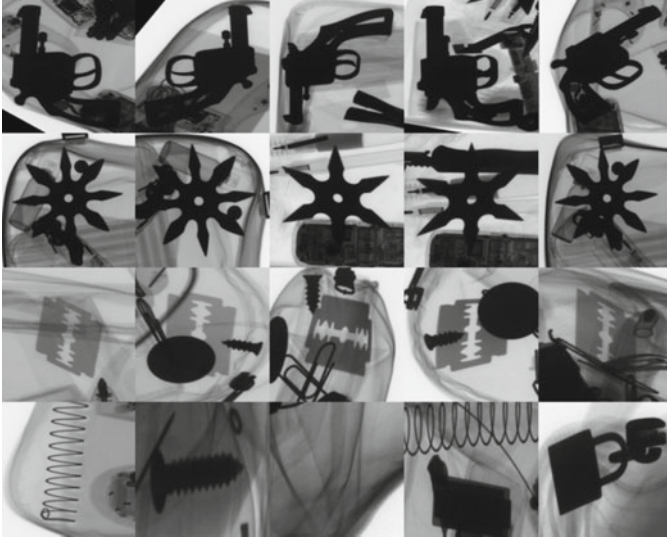



Fig. 7.21 Some testing X-ray images used in our experiments. Each row represents a labeled class (handguns, shuriken, razor blades, and others respectively)

$$\eta = \frac{1}{4} \sum_{i=1}^4 \eta_i. \tag{7.40}$$

- Detection of three threat objects:** In the second task, we have to design three different detectors (binary classifiers) : (1) one for Gun, (2) one for Shuriken, and (3) one for Blade. For each detector, there is a target (e.g., Shuriken for second detector). Each detector can be understood as a two-class problem: one class (called the positive class) is the target, and the another class (called the negative class) is the rest. Similar to previous problem, training data must be used to train the detectors, validation data can be used to tune the detectors’ parameters (if any), and testing data have to be used to measure the final performance of the detectors. For the second detector (i.e., Shuriken), for example, in our database according to Table 7.2, there are 100 images for the positive class and $200 + 100 + 500 = 800$ images for the negative class that can be used for training purposes. In this example, the validation can be performed using 50 images for the positive class and $50 + 50 + 200 = 300$ images for the negative class. Finally, for the testing of the second detector, there 100 images for the positive class and $100 + 100 + 400$ for the negative class. The performance must be given in terms of precision–recall (Pr, Re) considering all images of the testing set. The variables precision and recall are defined in Eqs. (6.41) and (6.38) respectively. Ideally, a perfect detection means all existing targets are correctly detected without any false alarms, i.e., $Pr = 1$ and $Re = 1$. The values (Pr, Re) that maximizes the score $Q = \sqrt{Pr \times Re}$ are reported. As average performance, we define

Table 7.3 Precision and recall for each detector using pre-trained models [→ Example 7.4 

Method*	Classifier	Features	Gun		Shuriken		Blade		Q	All
			Pr	Re	Pr	Re	Pr	Re	η_Q	η
AlexNet ² [29]	svm-rbf	4096	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
DenseNet121 ² [24]	svm-rbf	1024	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
GoogleNet ² [60]	svm-lin	1024	1.00	1.00	1.00	1.00	0.86	1.00	0.98	0.99
InceptionV3 ⁰ [61]	svm-rbf	2048	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
MobileNet ² [23]	svm-rbf	1280	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
RCNN_ILSVRC ₁₃ ² [16]	knn1	4096	1.00	1.00	1.00	1.00	0.62	0.70	0.89	0.95
ResNet50 ² [20]	svm-rbf	2048	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
ShuffleNet ² [69]	svm-rbf	544	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
SqueezeNet ² [25]	svm-rbf	1000	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
VGG16 ¹ [58]	svm-rbf	1000	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
VGG19 ¹ [58]	knn1	1000	0.97	1.00	0.99	1.00	0.93	1.00	0.98	1.00
Xception ⁰ [8]	knn1	2048	0.71	0.50	0.89	0.85	0.57	0.40	0.65	0.80
ZfNet512 ² [67]	nn	1024	1.00	1.00	1.00	1.00	0.91	1.00	0.98	1.00
AIISM [41, 54]**	–	–	0.97	0.97	0.95	0.96	0.99	0.99	0.94	0.96

(*)Output layer 0: Keras, layer before softmax, 1: ONNX, layer after softmax, 2: ONNX, layer before softmax

(**)Best non-deep-learning method

$$\eta_Q = \frac{1}{3} \sum_{i=1}^3 Q_i, \quad (7.41)$$

where $i = 1 \dots 3$ means the classes Gun, Shuriken and Blade respectively.



Python Example 7.4: In this example, we follow the experimental protocol defined above according to Table 7.2 in two recognition tasks: four-class classification and detection three threat objects. For these tasks, we evaluate pre-trained model MobilNet [23] using five different classifiers (`knn1`, `knn3`, `svm-lin`, `svm-rbf`, and `nn`). The idea is to use the pre-trained model to extract features of each image, and classify the images according to the extracted features as illustrated in Fig. 7.18. The features are extracted by function `extract_prt_features`¹¹ of `pyxvis` Library. Other pre-trained models (such as AlexNet [29], GoogleNet [60], VGG16 and VGG19 [58] among others) are implemented in `pyxvis` Library as well. In the following code, the reader can see how easy is to define the training, validation, and testing datasets (see Table 7.2) using `init_data` and `append_data` of `pyxvis` Library. In this implementation, the validation set is used to evaluate the performance of each classifier (defined as the average of η in (7.40) and η_Q in (7.41)). Thus, the classifier that achieves the best performance on validation subset is used to report the performance on testing dataset.

¹¹This function is used to extract the features of all images that are in a folder. For a single image, function `extract_prt_features_img` of `pyxvis` Library can be used.

Listing 7.4 : Pre-trained models.

```

import numpy as np
from sklearn.metrics import accuracy_score
from pyxvis.learning.pretrained import prt_model, extract_prt_features
from pyxvis.io.gdxdraydb import DatasetBase
from pyxvis.learning.classifiers import clf_model, define_classifier
from pyxvis.learning.classifiers import train_classifier, test_classifier
from pyxvis.learning.evaluation import precision_recall
from pyxvis.io.data import init_data, append_data
from pyxvis.io.plots import print_confusion

gdxdray      = DatasetBase()
path         = gdxdray.dataset_path + '/Baggages/'
model_id     = 6 # 0 ResNet50, 1 VGG16, 2 VGG19, ... 6 MobileNet, ... 13 RCNN_ILSVRC13
output_layer = 2 # 0 Keras-Last, 1 ONNX-Last, 2 ONNX-Previous

# Classifiers to evaluate
ss_cl       = ['knn1', 'knn3', 'svm-lin', 'svm-rbf', 'nn']
(model, size, model_name) = prt_model(model_id, output_layer)
X49 = extract_prt_features(model_id, output_layer, model, size, model_name, path+'B0049/')
X50 = extract_prt_features(model_id, output_layer, model, size, model_name, path+'B0050/')
X51 = extract_prt_features(model_id, output_layer, model, size, model_name, path+'B0051/')
X78 = extract_prt_features(model_id, output_layer, model, size, model_name, path+'B0078/')
X79 = extract_prt_features(model_id, output_layer, model, size, model_name, path+'B0079/')
X80 = extract_prt_features(model_id, output_layer, model, size, model_name, path+'B0080/')
X81 = extract_prt_features(model_id, output_layer, model, size, model_name, path+'B0081/')
X82 = extract_prt_features(model_id, output_layer, model, size, model_name, path+'B0082/')

best_performance = 0 # initial value
for i in range(len(ss_cl)):
    cl_name = ss_cl[i]
    print('\nEvaluation of '+cl_name+' using '+model_name+'...')
    (Q_v, Q_t) = (0,0) # initial score Q values for validation and testing
    for j in range(4):
        if j==0:
            (c0, c1, c2, c3) = (1,0,0,0)
            st = 'Gun'
        elif j==1:
            (c0, c1, c2, c3) = (0,1,0,0)
            st = 'Shuriken'
        elif j==2:
            (c0, c1, c2, c3) = (0,0,1,0)
            st = 'Blade'
        elif j==3:
            (c0, c1, c2, c3) = (0,1,2,3)
            st = 'All'

    print('building dataset for '+st+' using ' + model_name + ' ...')
    # Training data
    (X, d) = init_data(X49[0:200], c0) # Gun
    (X, d) = append_data(X, d, X50[0:100, :], c1) # Shuriken
    (X, d) = append_data(X, d, X51[0:100, :], c2) # Blade
    (X, d) = append_data(X, d, X78[0:500, :], c3) # Other
    # Validation data
    (Xv, dv) = init_data(X79[0:50, :], c0) # Gun
    (Xv, dv) = append_data(X, d, X80[0:50, :], c1) # Shuriken
    (Xv, dv) = append_data(X, d, X81[0:50, :], c2) # Blade
    (Xv, dv) = append_data(X, d, X82[0:200, :], c3) # Other
    # Testing data
    (Xt, dt) = init_data(X79[50:150], c0) # Gun
    (Xt, dt) = append_data(X, d, X80[50:150, :], c1) # Shuriken
    (Xt, dt) = append_data(X, d, X81[50:150, :], c2) # Blade
    (Xt, dt) = append_data(X, d, X82[200:600, :], c3) # Other

    print('training '+cl_name+' for '+st+' using ' + model_name + ' ...')

```

```

(name,params) = clf_model(cl_name) # function name and parameters
clf = define_classifier([name,params]) # classifier definition
clf = train_classifier(clf,X,d) # classifier training
ds_v = test_classifier(clf,Xv) # classification of validation
ds_t = test_classifier(clf,Xt) # clasification of testing
print('Results - ' + st + ' ('+cl_name+') for the detectors:')
if j<3: # detection of three treat objects
    # performance on validation subset
    (pr_v,re_v) = precision_recall(dv,ds_v)
    Q_v = Q_v + np.sqrt(pr_v*re_v)
    print(f'Pr_val = {pr_v:.4f}')
    print(f'Re_val = {re_v:.4f}')
    # performance on testing subset
    (pr_t,re_t) = precision_recall(dt,ds_t)
    Q_t = Q_t + np.sqrt(pr_t*re_t)
    print(f'Pr_test = {pr_t:.4f}')
    print(f'Re_test = {re_t:.4f}')
else:
    # summary of three detections
    Q_v = Q_v/3 # score Q on validation
    print(f'Q_val = {Q_v:.4f} of all detectors')
    Q_t = Q_t/3 # score Q on testing
    print(f'Q_test = {Q_v:.4f} of all detectors')
    # four-class classification
    print('Results - ' + st + ' ('+cl_name+') for the 4-class classifier:')
    acc_v = accuracy_score(dt,ds_t)
    print(f'Acc_val = {acc_v:.4f}')
    acc_t = accuracy_score(dv,ds_v)
    print(f'Acc_test = {acc_t:.4f}')
    print(f'Acc_t = {acc_t:.4f}')
    print_confusion(dt,ds_t)
performance = (acc_v+Q_v)/2
if performance>best_performance:
    print(f'performance = {performance:.4f} *** new max ***')
    best_performance = performance
    best_Q = Q_t
    best_acc = acc_t
    best_clf = cl_name
print('Best result: classifier = '+best_clf)
print(f' Q_test = {best_Q:.4f}')
print(f' acc_test = {best_acc:.4f}')

```

The output of this pre-trained model and other ones implemented in pyxvis Library is shown in Table 7.3. We can see that many of the pre-trained models achieve a perfect performance of 100%. This result is very relevant because the implementation of this solution can be performed in a couple of hours (the pre-trained models are already trained, we only need to extract the features and train a classifier like SVM). It is worthwhile to mention that the best non-deep learning method based on handcrafted features (see AISM [54] in Table 7.3), developed after several months of work for this task, achieves 4–6% less of performance. □

7.5 Transfer Learning

The use of transfer learning in X-ray testing is similar to the use of pre-trained models (explained in Sect. 7.4). Here, however, the pre-trained model is *re-trained* in a smart way using a low number of X-ray images [66].

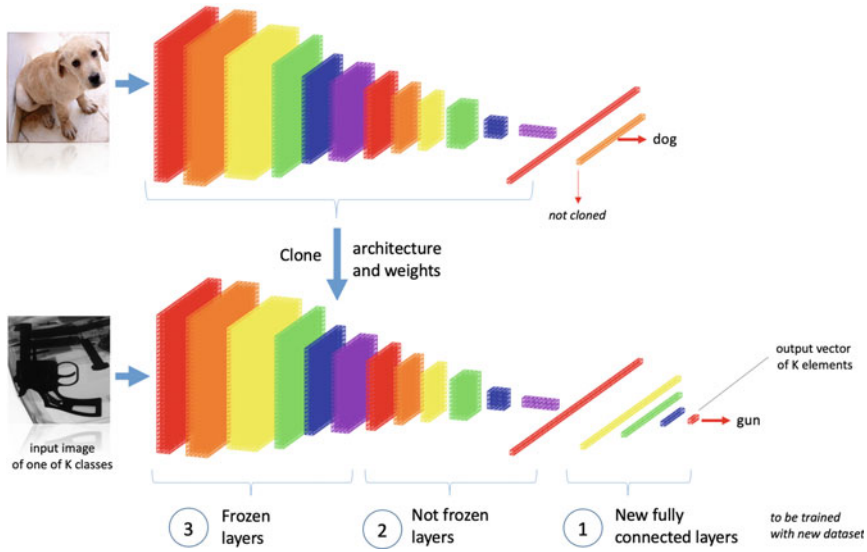


Fig. 7.22 Classification strategy using a transfer learning (see explanation in Sect. 7.5.2)

7.5.1 Basics of Transfer Learning

The idea is to use, in X-ray images, models that have been trained on other domains (e.g., ImageNet [55]). The main difference with pre-trained models (see Sect. 7.4) is that we can re-train the models using a *fine-tuning* approach. In fine-tuning, we re-train a model for the new domain using as initial weights the pre-trained weights of the model (instead of random initial values). Thus, we can take advantage of the pre-trained weights that have been obtained after a sophisticated training process with millions of images. In fine-tuning, the initial (pre-trained) weights are updated using a training approach with images of the new domain (X-ray images). A good example is given in [2], where transfer learning has been used in baggage inspection by fine-tuning AlexNet and GoogleNet.

Usually, sophisticated deep learning models can be trained successfully thanks to the great power of today’s computers and also because there are a huge number of annotated images available. However, sometimes it is very difficult to have both of them. For example, in X-ray testing, it is very common to have datasets with hundreds or thousands (and not millions) of X-ray images. In addition, there are many students or universities that do not have access to such a powerful computer. For these reasons, transfer learning is a very attractive alternative: we can re-train a sophisticated model using a low number of X-ray images on a regular computer.

7.5.2 Training in Transfer Learning

In order to train a deep learning model using a transfer learning strategy, we can use the approach illustrated in Fig. 7.22. Before starting to re-train the model it is necessary to clone the first layers of the pre-trained model (as we do in the pre-trained model strategy outlined in Sect. 7.4). Following Fig. 7.22, we add new layers (typically fully connected layers) to the cloned model (see layers ①). Now, we divided the cloned model into two parts: the not frozen layers (see layers ②) and the frozen layers (see layers ③). Usually, in the training stage, we can use the following three steps:

1. Layers ① are trained and the rest of the weights (layers ② and ③) are not changed during training, i.e., their weights are the original pre-trained weights (they remain constant during training).
2. Layers ① + ② are fine-tuned (using weights of first step as initial weight values) and the rest of the weights (layers ③) are not changed during training.
3. (Optional Step) Layers ① + ② + ③ are fine-tuned (using weights of second step as initial weight values). This step can be performed in case we have enough images to train the whole model.

Using the same approach addressed in Sect. 7.4.1 to visualize the activation of the elements of the CNN layers, in Fig. 7.23, we show the synthetic input image generated for a specific element of one of the last layers for the original VGG16 model (trained for ImageNet) and for the fine-tuned model (trained with threat objects). The reader can observe that the patterns are very similar, however, the second one seems to be adapted to the new domain.

7.5.3 Example of Transfer Learning



Python Example 7.5: In this example, we follow the strategy outlined in Fig. 7.22 for transfer learning in a problem of recognition of threat objects. For this end, we use a set of images of threat objects that has four classes (Guns, Shuriken, Blades, and Others) divided into training (with 600 images per class) and testing subsets (with 100, 100, 100, and 400 images for each corresponding class). In our example, we use MobileNet [23] as base model (that has 87 layers) and four extra layers: the first one is `GlobalAveragePooling2D` (that joins the base model with the extra fully connected layers), two fully connected layers defined by variable `fc_layers` with 32 and 16 nodes each, and a final fully connected layer with softmax as output with 4 nodes (because, in this example, there are four classes). Thus, the new model has totally 91 layers. In this example, we have ① the new layers, ② the not frozen layers, and ③ the frozen layers. The training strategy follows the method mentioned above in three steps: in ①, we train 4 layers, in ① + ②, we train 9 layers, and in ① + ② + ③, we train 91 layers. The number of epochs in each step

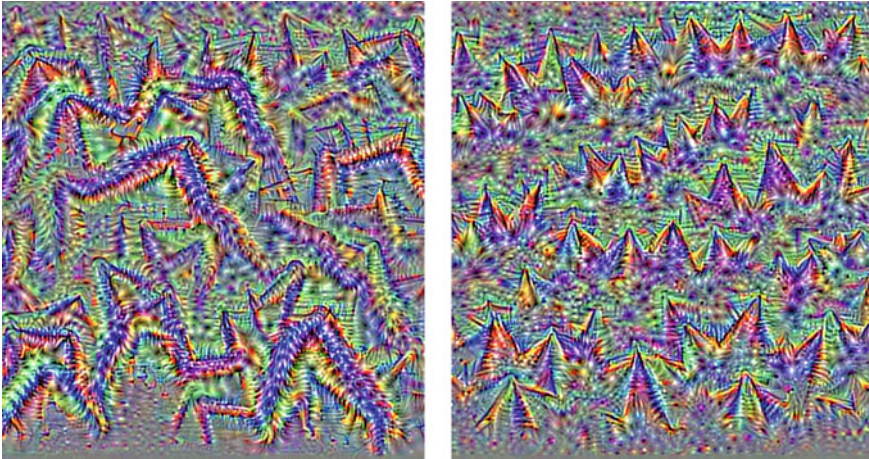


Fig. 7.23 Generated synthetic input images for VGG16 that maximize the activation of layer `block5_conv2` filter '0': (left) using base model VGG16 trained with ImageNet, (right) using VGG16 fine-tuned with threat objects, the shapes of the shuriken are remarkable. Left image is illustrated in Fig. 7.19 in a blue square

is defined by variable `nb_epochs`, in our case is `[40,40,40]`. We could decide that the last step is not necessary by defining `nb_epochs = [40,40,0]`.

Listing 7.5 : Transfer learning.

```

from pyxvis.learning.transfer import generate_training_set, tfl_train
from pyxvis.learning.transfer import tfl_model, tfl_define_model, tfl_testing_accuracy
from pyxvis.io.plots import plot_confusion

# Definitions
path_dataset = './images/objects'
nb_classes = 4 # number of classes of the recognition problem
batch_size = 10 # batch size in training
nb_epochs = [40,40,40] # epochs for Training-1, Training-2, Training-3
# 1st value: epochs for new layers only,
# 2nd value: epochs for new and top layers of base model,
# 3rd value: epochs for all layers
# (eg [50,0,0], [40,50,0], etc.)

train_steps = 10
val_steps = 5
fc_layers = [32, 16] # fully connected layers after froozen layers
img_size = [224,224] # size of the used images
val_split = 0.2 # portion of training set dedicated to validation,
# 0 means path_dataset/val is used for validation
opti_method = 1 # optimizer > 1: Adam, 3: SGD
base_model = 1 # 1: MobileNet, 2: InceptionV3, 3: VGG16, 4: VGG19,
# 5: ResNet50, 6: Xception, 7: MobileNetV2,
# 8: DenseNet121, 9: NASNetMobile, 10: NASNetLarge
nb_layers = -5 # layers 0... nb_layers-1 will be frozen, negative
# number means the number of top layers to be unfrozen
augmentation = 0.05 # 0 : no data augmentation, otherwise it is range for
# augmentation (see details in generate_training_set)

# Base model (last layer is not included removed)

```

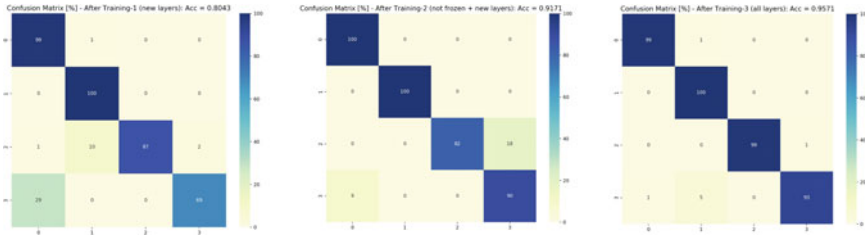


Fig. 7.24 Confusion matrix and accuracy on testing subset using transfer learning: after training-1 (layers ①), the accuracy is 80.43%, training-2 (layers ① + ②), the accuracy is 91.71%, and training-3 (layers ① + ② + ③), the accuracy is 95.71% according to diagram of Fig. 7.22. [→ Example 7.5 📁]

```

bmodel      = tfl_model(base_model)

# New model with dense fully connected layers
model       = tfl_define_model(bmodel,fc_layers,nb_classes)

# Training and validation sets
(train_set,
 val_set)   = generate_training_set(val_split, augmentation, batch_size,
                                   path_dataset, img_size)

# Training: Transfer learning
(model,
 confusion_mtx,
 acc)      = tfl_train(bmodel,model,opti_method, nb_layers,
                      train_set,train_steps,val_set,val_steps,nb_epochs,
                      path_dataset,nb_classes,img_size)

# Accuracy in testing set using best trained model
plot_confusion(confusion_mtx,acc,'Top Model: Testing in Threat Objects',0,nb_classes)

```

The output of this code is in Fig. 7.24 in which we show the confusion matrices and accuracy on testing dataset after each training step. We observe how the accuracy is incremented after each step. □

7.6 Generative Adversarial Networks (GANs)

Generative Adversarial Networks (GANs) have been used successfully in the last years to generate realistic synthetic data [7, 9, 27]. In X-ray testing, we use GAN to simulate X-ray images, for example, as data augmentation in training data to increase the number of samples of some underrepresented class, or as new data in a training course for human inspectors. Some applications of simulated X-ray images using GAN can be found in [38] for the simulation of casting defects¹² and in [56, 65, 70, 72] for the simulation of threat objects. The simulated X-ray images using GAN are very realistic as we can see in Figs. 7.25 and 7.26 for defects and shuriken respectively.

¹²GAN solutions have been used in other kinds of defects, see for example, [46].

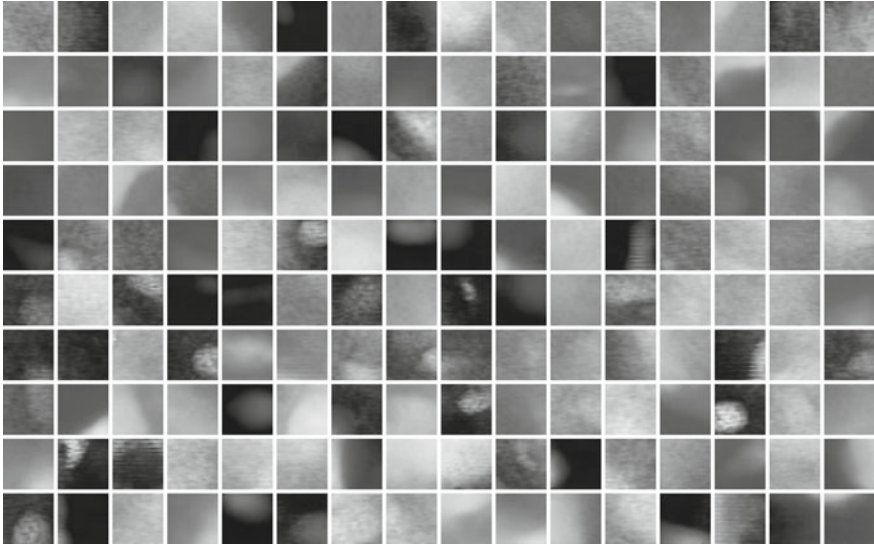


Fig. 7.25 Simulated defects in aluminium castings using GAN. [→ Example 7.6 🌐]

7.6.1 Basics of GAN

The key idea of GAN is simple as we can see in Fig. 7.27: it consists of a *generator* and a *discriminator* working together. The generator will be used to produce a synthetic X-ray image from a noise source, whereas the discriminator will be in charge to determine if an input image is real or fake. Thus, the discriminator should differentiate the real (training) images from the synthetic ones generated by the generator.

Both generator and discriminator are trained following a zero-sum game schema [17]. In the zero-sum game schema, the generator and the adversary (discriminator) compete against other. For this end, we define a noise source as an image \mathbf{Z} of $p \times p$ pixels. The generator (\mathcal{G}) is a neural network function based on auto-encoders [1] that takes noise image \mathbf{Z} and transforms it into a fake image \mathbf{X}_F of $n \times n$ pixels:

$$\mathbf{X}_F = \mathcal{G}(\mathbf{Z}). \quad (7.42)$$

On the other hand, the discriminator (\mathcal{D}) is a function based on a neural network that takes an input image \mathbf{X} of $n \times n$ pixels and gives as output a value y that corresponds to the probability that \mathbf{X} is a real image; 1 means that \mathbf{X} is real, 0 is fake:

$$y = \mathcal{D}(\mathbf{X}). \quad (7.43)$$

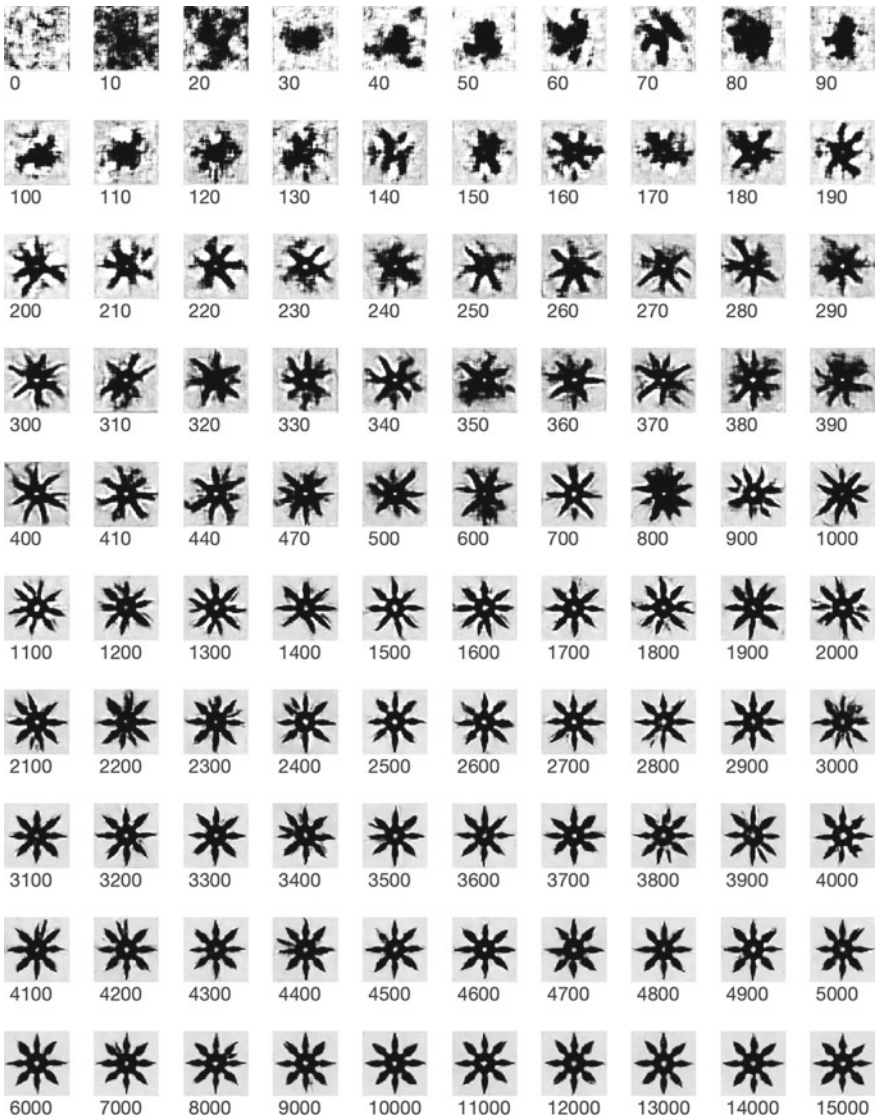


Fig. 7.26 Simulated shuriken using DCGAN from 0 to 15000 iterations. [→ Example 7.6 🌐]

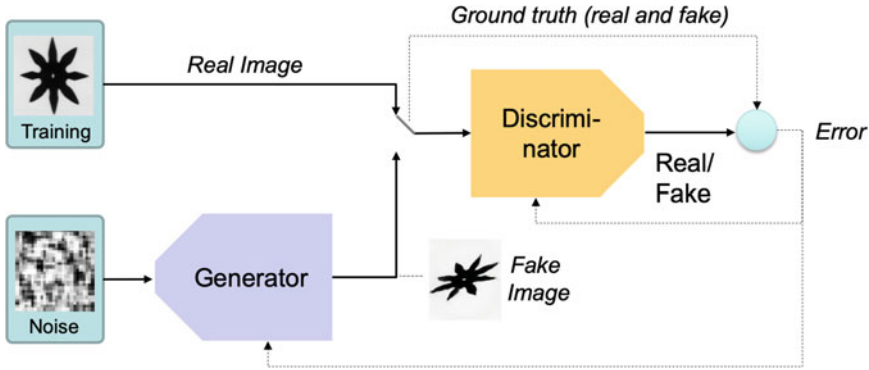


Fig. 7.27 Diagram of a GAN model. The generator produces a fake image from a noise source, whereas the discriminator distinguishes real from fake images. In the training stage, the error is used to increase the performance of both generator and discriminator (see dashed lines). Once the model is learned, the generator alone can be used to generate realistic synthetic images

7.6.2 Training of GAN

In training stage, two sets are available: (i) a set of m real images, $\mathbf{X}_R(1) \dots \mathbf{X}_R(m)$, and (ii) a set of m noisy images $\mathbf{Z}(1) \dots \mathbf{Z}(m)$, from them a set of m fake images are computed $\mathbf{X}_F(1) \dots \mathbf{X}_F(m)$ using (7.42). Thus, the discriminator is learned by maximizing:

$$J_D = \sum_{i=1}^m \log [\mathcal{D}(\mathbf{X}_R(i))] + \sum_{i=1}^m \log [1 - \mathcal{D}(\mathbf{X}_F(i))] \rightarrow \max \quad (7.44)$$

The first sum is maximal when the real images are classified as ‘real’, whereas the second sum is maximal when the synthetic images are classified as ‘fake’. Thus, the idea of (7.44) is to classify as 1 the real images and as 0 the synthetic ones.

The aim of the generator is to model the distribution of the training dataset. Since the goal of the generator is to generate fake images that fool the discriminator, the generator will do a good job if the generated fake images are classified by the discriminator as ‘real’. Thus, the generator is trained by minimizing the following objective function:

$$J_G = \sum_{i=1}^m \log [1 - \mathcal{D}(\mathbf{X}_F(i))] \rightarrow \min . \quad (7.45)$$

In this case, the sum is minimal when the synthetic images are classified as ‘real’. That means that the generated fake images will be so realistic that the discriminator will classify them as ‘real’. In the training stage, objective functions J_D and J_G are playing a *min-max* game, the reader can see that the second sum of (7.44) is equal

to the sum of (7.45), however, in the first case, we are trying to maximize it (the discriminator should recognize that the synthetic images are fake), whereas in the second one, the aim is to minimize it (the generator wants to fool the discriminator).

7.6.3 Implementation of GAN

GAN models can be easily implemented using Deep Convolutional Generative Adversarial Networks (DCGAN) [49], where both generator and discriminator are sequential models [49]. In DCGAN, the architectures of discriminator \mathcal{D} and generator \mathcal{G} are CNNs as illustrated in Figs. 7.9 and 7.28 respectively. In each step of the generator, the \mathbf{Z} is upsampled and convoluted. The upsampling process can be achieved using the 2D Transposed Convolution [68]:

• **2D Transposed Convolution [trans_conv]:** This layer corresponds to a convolution that increases the dimension of the input image as illustrated in Fig. 7.29. In general, for an input image \mathbf{X} of $n \times n$ pixels and a convolutional kernel \mathbf{K} of $m \times m$ pixels, the output $\mathbf{Y} = \mathbf{X} \star \mathbf{K}$ is defined as follows:

$$\mathbf{Y}(i_1 : i_2, j_1 : j_2) = \mathbf{Y}(i_1 : i_2, j_1 : j_2) + X(i, j)\mathbf{K}, \quad (7.46)$$

where

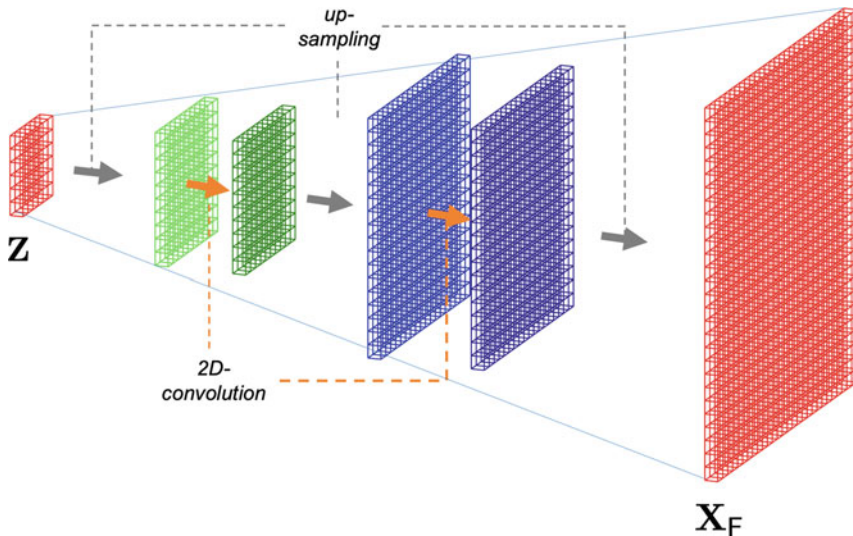


Fig. 7.28 Architecture of a generator based on deep convolutional neural networks in a GAN model. Input \mathbf{Z} is a small noise image, and output \mathbf{X}_F is a (larger) synthetic image

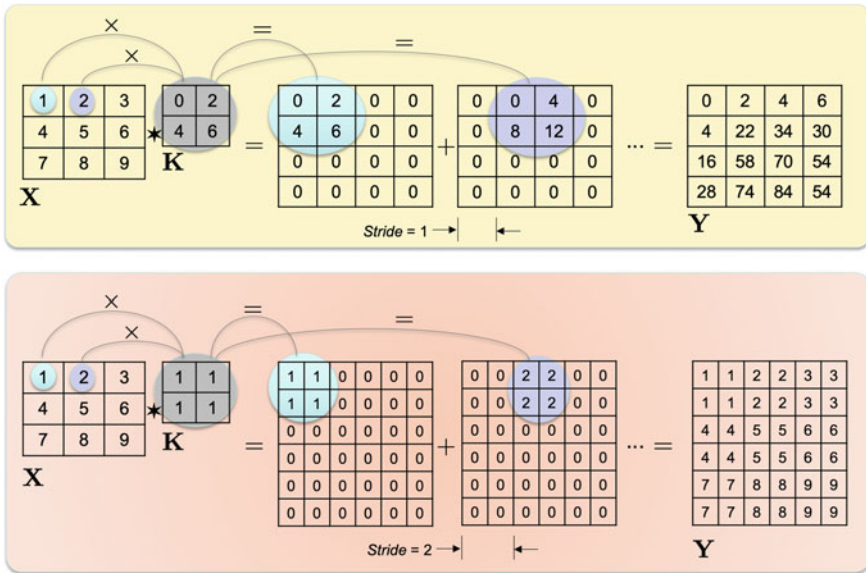



Fig. 7.29 Example of 2D transposed convolution: (Top) with stride $r = 1$, (Bottom) with stride $r = 2$

$$\begin{aligned}
 i_1 &= (i - 1)r + 1 \\
 i_2 &= (i - 1)r + m \\
 j_1 &= (j - 1)r + 1 \\
 j_2 &= (j - 1)r + m
 \end{aligned}
 \tag{7.47}$$

for $i = 1 \dots n$ and $j = 1 \dots n$. The stride, i.e., the number of pixels that the kernel moves to right and down in each step, is given by variable r . As shown in Fig. 7.29, by setting r to m (the size of the kernel), and defining all elements of K as 1, we can repeat the rows and columns of X by size $m \times m$.

7.6.4 Example of GAN

 **Python Example 7.6:** In this example, we simulate X-ray images of shuriken using class `DCGAN`¹³ of `pyxvis` Library. As training data, we use a dataset of 10.640 real images of shuriken of 32×32 pixels (stored in file `shuriken_32x32.npy`). The real images were extracted from `GDXXray+` and augmented using rotation and reflection.

¹³Based on the implementation of <https://github.com/eriklindernoren/Keras-GAN/blob/master/dcgan/dcgan.py>.

Listing 7.6 : Generative Adversarial Network.

```

from pyxvis.learning.gan import DCGAN

gan_proc = 0 # 0 training , # 1 testing

# Training
if gan_proc == 1: # Training
    path_file = '../data/shuriken_32x32.npy' # file of real patches
    epochs = 15000 # number of epochs
    interval = 250 # saving intervals
    dcgan = DCGAN(path_file)
    dcgan.train(epochs=epochs, batch_size=32, save_interval=interval)

else: # Testing (one generation of simulated images)
    size = 32 # size of the image, eg. 32 for 32x32 pixels
    # trained model h5 file
    gan_weights_file = '../output/GAN/models/gan_model_015000.h5'
    N = 200 # number of synthetic images to be generated
    dcgan = DCGAN(size)
    dcgan.load_gan_model(gan_weights_file)
    dcgan.save_gan_examples()
    dcgan.save_synthetic_images('output',N)

```

The output of this code is in Fig. 7.26. We can see that the similarity of the synthetic generated X-ray images with trained GAN model is very high after 5000 iterations. In the generation, only the generator is used (and not the discriminator). If we want to generate N new images with the trained generator, in our code, variable `gan_proc` must be set to 1 and variable `N` must be set to the number N (e.g., $N = 200$). Figure 7.25 shows a GAN simulation of casting defects with this code, in which the dataset `casting_defects_28x28.npy`¹⁴ was used. □

7.7 Detection Methods

In this section, we address relevant methods of *object detection* that have been published in the last years. The idea of the detection methods is to locate and recognize object instances in real images.

7.7.1 Basics of Object Detection

In computer vision, we distinguish between *image classification* and *image detection* as shown in Fig. 7.30. In X-ray testing, both concepts can be explained as follows:

- **Image classification:** The purpose of *image classification* in X-ray testing is to assign an X-ray image to one class. For example, in image classification, an X-ray

¹⁴The file can be downloaded from <https://domingomery.ing.puc.cl/material/>.

image can be classified as a ‘handgun’, that means, in the X-ray image the classifier has found a handgun (here, the classes could be ‘handgun’, ‘knife’, and other threat objects), or, in another example, a small sub-image of an X-ray image of an aluminum castings is classified as ‘defect’ (here, the classes could be ‘defect’ and ‘no-defect’). Image classification is typically used when there is only one object per image to be recognized. An example is illustrated in Fig. 7.30a. The location of the recognized object is not given in image classification, it is well assumed that the object is in the center of the image, but obviously, this is not always true. In image classification using deep learning, as explained in Sect. 7.3, the input image is fed into a CNN that gives a feature vector (of dimension 4096, for example). The vector is the input of a classifier, e.g., a fully connected layer, with K outputs, where K is the number of classes to be recognized. Thus, the input is an image and the output is a category label. In case that both classification and localization of the object in the input image are required, there are some approaches with one fully connected layer for the classification and another fully connected layer for the localization that gives the coordinates and the dimensions of a rectangle that contains the recognized object [26], where the second fully connected layer is treated as a regression problem, where the output is continuous values instead of a class.

- **Image Detection:** On the other hand, in *image detection*, more than one object can be recognized in an X-ray image and the location of each recognized object is given by a *bounding box*, i.e., a rectangle that encloses the detected object defined by the coordinates of the center of the rectangle (x, y) and its dimen-



Fig. 7.30 Image detection and image classification

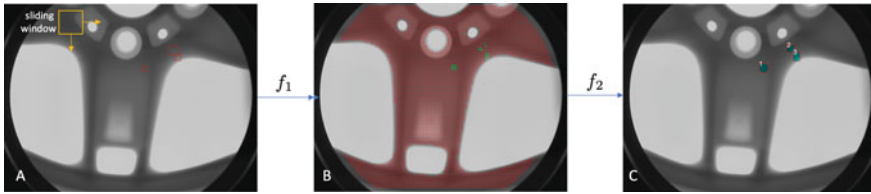


Fig. 7.31 Object detection using CNN and sliding-windows: The original X-ray image ‘A’ with three defects (see red rectangles) is processed by f_1 using a sliding-window approach. For each position of the detection windows (orange square), a patch is extracted and classified as ‘defect’ or ‘no-defect’ by a trained CNN. The dots of image ‘B’ show in red and green the center of the patches that were classified as ‘defect’ and ‘no-defect’ respectively. Using image processing approaches the dots of image ‘B’ are processed by f_2 to detect the ‘defect’ regions. In this example, all defects were correctly detected with no false alarm

sions, width and height, (w, h) , where all four variables are given in pixels (see for example, the red rectangle of the shuriken in Fig. 7.30b, where the center is in $(x = 748, y = 405)$ and the dimensions are $w = 323$ and $h = 505$ pixels). In example of Fig. 7.30b, image detection is able to recognize a set of objects (see table and red bounding boxes). Typically, a probability of detection is computed for each recognized object (that can be understood as a new column in the table), so that the final output corresponds to those objects that have a probability greater than a threshold.

A simple strategy based on sliding-window methodology has been proposed some years ago for image detection based on image classification. An example is illustrated in Fig. 7.31 for defect detection in aluminum castings [38]. In this approach, a detection window (see the orange square in Fig. 7.31-A) is slidded over an input image in both horizontal and vertical directions, and for each localization of the detection window, a classifier decides to which class belongs the corresponding portion of the image according to its representation. Here, the classifier is a CNN, as explained in Sect. 7.3.4, that is used to identify one of two classes: ‘defects’ or ‘no-defects’. For this end, a huge number of patches of each class is used to train the CNN model. The patches have the same size of the detection window, and they can contain a defect (for the ‘defect’ class) or not (for the ‘no-defect’ class) as shown in Fig. 7.10. Finally, the locations of the X-ray image that have been detected as ‘defects’ (see green dots in Fig. 7.31-B) are analyzed using image processing. Thus, we can determine which regions of the image are defects or not (see detected regions in Fig. 7.31-C).

It is worthwhile to mention, that this approach requires the classification of a huge number of patches. In addition, if the size of the objects to be detected varies, the sliding-windows approach must be performed for different patch-sizes. In this case, the computational time could be prohibited. For these reasons, new approaches that overcome this problem have been developed in the last years. In this section, we will cover them. They can be subdivided into two groups [26]:

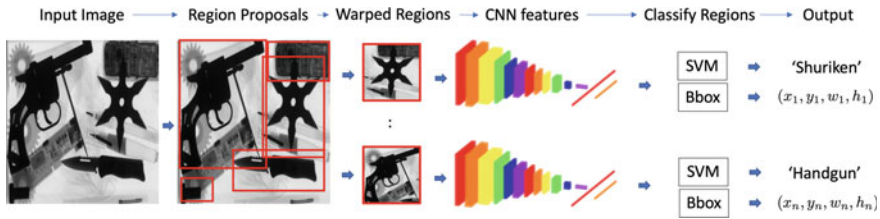


Fig. 7.32 R-CNN strategy

- (i) **Detection in two stages:** In the first step of these approaches, called *region proposal*, a method is used to determine regions of the input image in which an object can be present. In sliding-windows, for example (explained above), this step corresponds to an exhaustive search, however, there are other methods, e.g., R-CNN [16] that propose some regions instead of analyzing all possible patches of the input image. In the second step of these approaches, called *final classification*, a CNN is used to classify the regions that have been proposed by the first step. In Sect. 7.7.2 of this chapter, we address these region-based methods like R-CNN [16], Fast R-CNN [15], and Faster R-CNN [53] that uses this two-stage strategy.
- (ii) **Detection in one stage:** In these approaches, there is a single CNN that is trained to both location and classification, i.e., prediction of bounding boxes and estimation of the class probabilities of the detected bounding boxes. This group of approaches corresponds to the state of the art in detection methods because they are very effective and very fast. They are the best-performing and most representative deep learning-based object detection models, as stated in [71]. In this chapter, we address most representative methods, namely, YOLO [52] in Sect. 7.7.3 (versions YOLOv2 [50], YOLOv3 [51] and YOLOv4 [6]), SSD [37] in Sect. 7.7.4 and RetinaNet [32] in Sect. 7.7.5. We give a brief description of these detection models and their principal differences.

7.7.2 Region Based Methods

In this section, we address those methods from the first group that perform object detection in two stages. These methods are region-based methods because the first step is the region proposal, and the second is the final classification. To this group belong R-CNN [16], Fast R-CNN [15] and Faster R-CNN [53]. We include in this section an additional method called Mask R-CNN [22] that is an instance segmentation approach. They will be described in further details.

- **R-CNN:** In R-CNN (Regions with CNN features), there is a step that proposes potential regions and another step that classify them into the classes to be recognized [16], as shown in Fig. 7.32. By selecting regions in the first step, we avoid to

classify of a huge number of patches as mentioned in Sect. 7.7.1 for sliding-windows approach.

The first step of R-CNN, called *Selective Search*, is based on a method proposed in [63] that generates candidates of bounding boxes for use in object recognition, i.e., they are regions that have high probability of being an object. They are called regions of interest or RoIs. The method uses complementary image regions that consider many image conditions. Selective search is based on image processing and it consists of three stages: (i) Capture all scales: Many potential regions are generated in all possible scales. (ii) Diversification: a diverse set of strategies is used to merge similar regions together. (iii) Fast to compute: Final regions are proposed in a hierarchical order. In the proposed approach [16], 2000 RoIs are selected, many of them are noisy, but the recall is high, that means that most of the true objects are selected. One of the problems of this method is that the selective search is not learned, it is fixed, and it could be useful to learn which regions are relevant for a given application. In addition, the approach can be very slow because each of the 2000 RoIs must be analyzed independently. In order to speed up this step, sharing computing with Spatial Pyramid Pooling networks (SPP) can be used as proposed in [21]. In the second step of R-CNN, a trained CNN model based on AlexNet [29] is used to extract from each RoI a feature vector of 4096 elements as explained in Sect. 7.4. All RoIs are warped to 227×227 pixels because the CNN requires a fixed square size for the input images. The 4096-element feature vector extracted of a RoI is used by a SVM classifier that is trained to determine the class of the region. In addition, CNN predicts a correction of the bounding boxes because originally they are not correctly located by the selective search approach. Thus, SVM classifier is in charge of class determination, whereas the location is given by the corrected location of the original RoI that has been detected by the SVM.

R-CNN is much faster than a sliding-window approach, however, to analyze 2000 RoIs is still very computationally expensive and cannot be implemented in real time. It has been reported that for the testing stage, it requires around 50 s per image [16].

• **Fast R-CNN:** In order to avoid the mentioned problems, the same author proposed a faster approach called Fast R-CNN [15], as shown in Fig. 7.33. In this approach, two improvements are presented: (i) The selective selection of RoIs is performed by using a CNN that gives a feature map of the same size of the input image. The RoIs are partitions of this feature map that are warped into fixed-length vectors using a ‘RoI pooling layer’, i.e., a max-pooling layer with a pool size that does not depend on the input size. (ii) Instead of a SVM that classifies the extracted feature vector for every single RoI, fully connected layers are used for each RoI to determine both the category and the location of the object. Thus, the objects in the image are detected by using *two sibling output layers*, one for establishing the category of the detected bounding box, and another to correct the location of the bounding box. In comparison with R-CNN, the computational time of Fast R-CNN is significantly decreased (to 2.3 s per image) mainly because the CNN is executed just once for the input image and not for every RoI. Moreover, the accuracy of the detection is increased and the training time is around ten times faster.

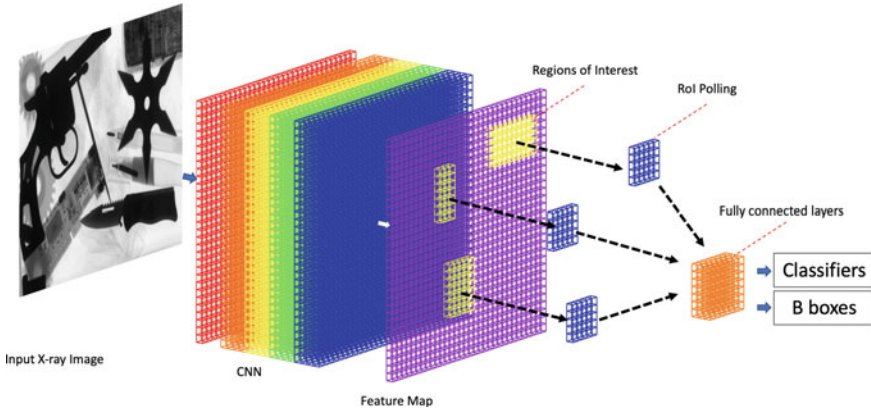


Fig. 7.33 Fast R-CNN strategy

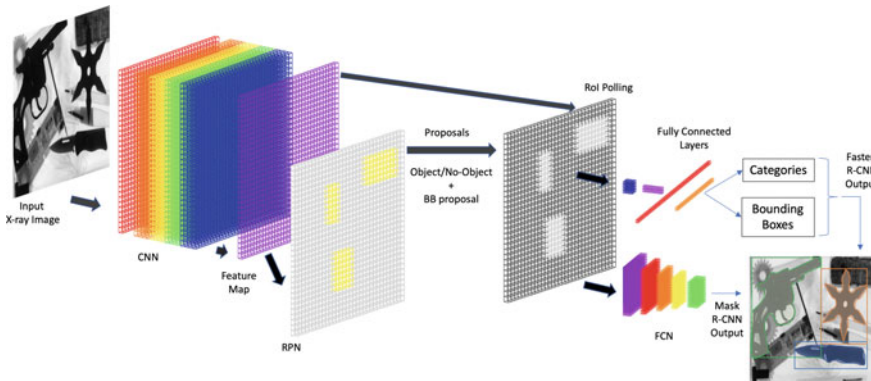


Fig. 7.34 Faster R-CNN and mask R-CNN strategies

• Faster R-CNN: The main drawback of Fast R-CNN is the computational time of the first step dedicated to region proposal, it is around 85% of the total detection time. In order to speed up the first step, Faster R-CNN (see Fig. 7.34) was proposed in [53]. Faster R-CNN includes an attention mechanism called Region Proposal Network (RPN), that is used to predict the RoIs from CNN features. That means the input image is fed into a CNN to obtain a feature map that is fed into the RPN that is trained to infer regions proposal. RPN outputs are two for each RoI: (i) a probability that the proposal is an object (it is a score that is used to determine whether the detection is an object or not) and (ii) a preliminary bounding box. Afterwards, a RoI pooling layer makes the final classification of the object in one of the categories and gives a correction of the preliminary bounding box. In Faster R-CNN, the detection time is decreased to 0.2 s per image. An example of Faster R-CNN in defect detection in aluminum casting can be found in [11, 12] very good results.

- **Mask R-CNN:** Another approach that is related to R-CNN is the well-known Mask R-CNN [22]. Mask R-CNN is a method that belongs to the category of ‘Instance Segmentation’. Whereas in object detection the goal is to detect bounding boxes, in instance segmentation, the goal is to perform a segmentation of an object at a pixel level. That means, the output is not a bounding box, it is the boundaries of the detected object. Mask R-CNN is a combination of R-CNN and Fully Convolutional Network (FCN). It consists of a faster attention mechanism (like Faster R-CNN) to generate RoIs with a FCN that runs on each of the RoIs. The FCN has convolutional layers that are used to predict the mask on the RoI, i.e., a binary image of the same size of the RoI where a pixel equals 1 (or 0) means that the pixel of the RoI belongs (or does not belong) to the detected object. An example is illustrated in Fig. 7.34.

7.7.3 YOLO

In region-based approaches, as explained in Sect. 7.7.2, object detection is performed in two stages: region proposal and final classification. That means the classification is not performed by *looking* at the complete image but at selected regions of the image. In order to overcome this disadvantage, a new method called YOLO, *You-Only-Look-Once* was proposed [52]. YOLO is a single (and powerful) convolutional neural network that *looks* the image once, i.e., the input image is fed into a single CNN which output is the simultaneous prediction of both the bounding boxes (localization) and the category probabilities (classification) of the detected objects. It is very fast because the input image is processed in a single pass by the CNN.

The key idea of YOLO is very simple: The input image is divided into a grid of $S \times S$ cells, and for each cell, YOLO can detect B objects. For each detected bounding box, YOLO computes:

- (x, y, w, h) : variables that define the detected bounding box, i.e., location (x, y) and dimension (width, height),
- p : confidence score that gives the probability that the bounding box encloses an object ($\text{Pr}(\text{Object})$), and
- p_i : for $i = 1 \dots K$: probability distribution over all K possible classes, i.e., p_i is a conditional class probability ($\text{Pr}(\text{Class}_i|\text{Object})$).

That means, for each bounding box, YOLO provides an array of $R = 4 + 1 + K$ elements: $(x, y, w, h, p, p_1, p_2, \dots, p_K)$, as illustrated in Fig. 7.35. In testing stage, an object of class i is detected if $\text{Pr}(\text{Object}) \times \text{Pr}(\text{Class}_i|\text{Object})$ is greater than a threshold.

Since in a grid cell, B bounding boxes can be detected, for each cell, an array of $Q = B \times R$ elements is computed.

The simplicity of YOLO (see Fig. 7.35) is due to (i) the architecture has only standard convolution layers with 3×3 kernels and max-pooling layers with 2×2 kernels, and (ii) the output of the CNN is a tensor of $S \times S \times Q$, that means, for

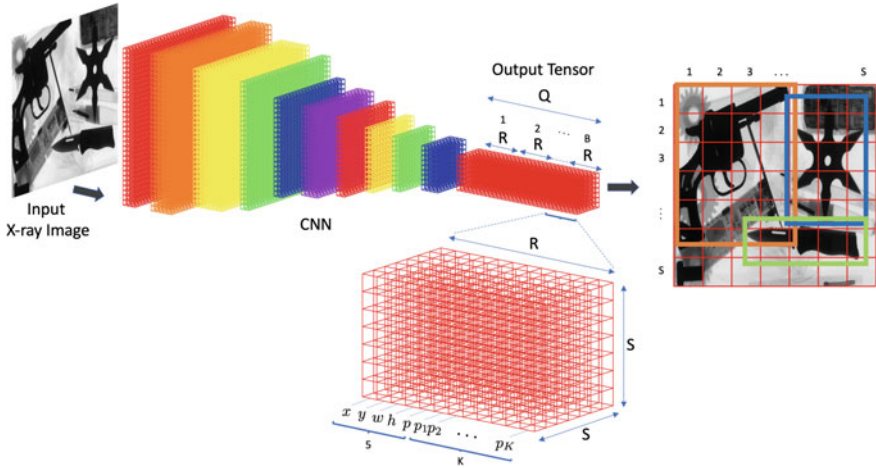



Fig. 7.35 YOLO strategy

each grid cell we have $5 + K$ elements per bounding box that give us information about the localization of the bounding box and the category probability.

In the last years, many versions of YOLO have been developed. In this section, we address the most relevant of them: YOLOv2 [50], YOLOv3 [51], and YOLOv4 [6].

• **YOLOv2:** The improvements proposed in YOLOv2 [50] focus on expanding the subdivision of the image, and the use of anchor boxes of different dimensions in each subdivision of the image (as proposed in the Faster R-CNN [53] model). These anchor boxes are pre-configured using the ‘k-means’ algorithm with Euclidean distance in the training set. Then, for each cell of the feature map extracted using the DarkNet-19 model, its anchor boxes are created with predictions for the objects inside [50].

 **Python Example 7.7:** In this example, we show how to use YOLOv2 in the detection of threat objects. For this task, we use the implementation of [56] (see footnote 17). In this implementation, there are four options: (i) object detection using a model that has been already trained for this task, (ii) training a new model using a set of training images, (iii) testing the model trained in the previous step, and (iv) evaluation of a model on a set of images:

Listing 7.7 : Threat object detection in baggage inspection using YOLOv2.

```

# Pre-trained model
python3 predict_yolo2.py -c config_full_yolo2_infer.json -i input_path/folder -o save/
folder/detection

# Training
python3 train_yolo2.py -c config_full_yolo2.json

```

```
# Testing
python3 predict_yolo2.py -c config_full_yolo2.json -i input_path/folder -o save/folder/
detection

# Evaluation
python3 evaluate_yolo2.py -c config_full_yolo2.json
```

The output of this code is in Figs. 7.36 and 7.37. □

- **YOLOv3:** In comparison with previous versions, YOLOv3 [51] includes two main updates: (i) the use of different scales (three scales) using a pyramidal architecture that aims to solve the problem of detection of small objects, and (ii) the use of a new feature extractor architecture called DarkNet-53 that improves upon DarkNet-19.



Python Example 7.8: In this example, we show how to use YOLOv3 in the detection of threat objects. For this task we use the implementation of [56] (see footnote 17). In this implementation, there are four options: (i) object detection using a model that has been already trained for this task, (ii) training a new model using a set of training images, (iii) testing the model trained in the previous step, and (iv) evaluation of a model on a set of images:

Listing 7.8 : Threat object detection in baggage inspection using YOLOv3.

```
# Pre-trained model
python3 predict_yolo3.py -c config_full_yolo3_infer.json -i input_path/folder -o save/
folder/detection

# Training
python3 train_yolo3.py -c config_full_yolo3.json

# Testing
python3 predict_yolo3.py -c config_full_yolo3.json -i input_path/folder -o save/folder/
detection

# Evaluation
python3 evaluate_yolo3.py -c config_full_yolo3.json
```

The output of this code is in Figs. 7.36 and 7.37. □

- **YOLOv4:** YOLOv4 has recently proposed in [6]. In this version, the feature map is extracted using a new architecture called Cross Stage Partial Network [64] that decreases the computation by reducing redundant gradient information. In YOLOv4, this network is called CSPDarknet-53. An additional increment of the performance is obtained by using Spatial Pyramid Pooling networks (SPP) [21] for sharing computing for pyramid features and a Path Aggregation Network (PAN) [36] for parameter aggregation from different levels of the CSPDarknet-53. Finally, the final prediction is performed as in YOLOv3 [51]. With these improvements, in comparison to YOLOv3, the accuracy is increased by 10% and the computation time is reduced by 11%.¹⁵

¹⁵In the last week (June 2020), YOLOv5 was released. See <https://github.com/ultralytics/yolov5>.

7.7.4 SSD

Another architecture contemporary to Faster R-CNN [53] and YOLO [52] is the SSD (Single-Shot Multi-Box Detector) [37]. Using direct image transformations, like YOLO, it predicts the location of the desired objects. The major difference is the use of map features in different depths, in order to obtain the analysis at different scales of the image. SSD combines the use of anchor boxes, like Faster R-CNN [53] and YOLOv2 [50], to predict the desired frames and uses a loss function for multi-tasking, as in the aforementioned detectors.



Python Example 7.9: In this example, we show how to use SSD7 in the detection of threat objects. For this task, we use the implementation of [56] (see footnote 17). In this implementation, there are four options: *(i)* object detection using a model that has been already trained for this task, *(ii)* training a new model using a set of training images, *(iii)* testing the model trained in the previous step, and *(iv)* evaluation of a model on a set of images:

Listing 7.9 : Threat object detection in baggage inspection using SSD7.

```
# Pre-trained model
python3 predict_ssd.py -c config_7_infer.json -i input_path/folder -o save/folder/
detection

# Training
python3 train_ssd.py -c config_7.json

# Testing
python3 predict_ssd.py -c config_7.json -i input_path/folder -o save/folder/detection

# Evaluation
python3 evaluate_ssd.py -c config_7_infer.json
```

The output of this code is in Figs. 7.36 and 7.37. □



Python Example 7.10: In this example, we show how to use SSD300 in the detection of threat objects. For this task we use the implementation of [56] (see footnote 17). In this implementation, there are four options: *(i)* object detection using a model that has been already trained for this task, *(ii)* training a new model using a set of training images, *(iii)* testing the model trained in the previous step, and *(iv)* evaluation of a model on a set of images:

Listing 7.10 : Threat object detection in baggage inspection using SSD300.

```
# Pre-trained model
python3 predict_ssd.py -c config_300_infer.json -i input_path/folder -o save/folder/
detection

# Training
python3 train_ssd.py -c config_300.json

# Testing
python3 predict_ssd.py -c config_300.json -i input_path/folder -o save/folder/detection
```

```
|| # Evaluation
|| python3 evaluate_ssd.py -c config_300_infer.json
```

The output of this code is in Figs. 7.36 and 7.37. □

7.7.5 RetinaNet

Together with YOLOv3 [51] and YOLOv4 [6], the RetinaNet architecture [32] is one of the most recent object detection models and combines the pyramidal feature extraction structure [33] with a residual architecture (ResNet) [20] that has obtained promising results in image classification. The pyramidal approach consists of decreasing the size of the image several times and making predictions for each of those sizes. Another novelty of this architecture is the shift from the typical cross-entropy to a ‘focal loss’-based objective that reduces the penalty for well classified classes while punishing misclassifications more aggressively for the rest.¹⁶



Python Example 7.11: In this example, we show how to use RetinaNet in the detection of threat objects. For this task, we use the implementation of [56] (see footnote 17). In this implementation, there are four options: (i) object detection using a model that has been already trained for this task, (ii) training a new model using a set of training images, (iii) testing the model trained in the previous step, and (iv) evaluation of a model on a set of images:

Listing 7.11 : Threat object detection in baggage inspection using RetinaNet.

```
|| # Pre-trained model
|| python3 predict_retinanet.py -c config_resnet50_infer.json -i input_path/folder -o save/
|| folder/detection
||
|| # Training
|| python3 train_retinanet.py -c config_resnet50.json
||
|| # Testing
|| python3 predict_retinanet.py -c config_resnet50.json -i input_path/folder -o save/folder/
|| detection
||
|| # Evaluation
|| python3 evaluate_retinanet.py -c config_resnet50.json
```

The output of this code is in Figs. 7.36 and 7.37. □

¹⁶An implementation of RetinaNet for casting defect detection in GDXray+, can be found on <https://github.com/aurotripathy/GDXray-retinanet> by Auro Tripathy.

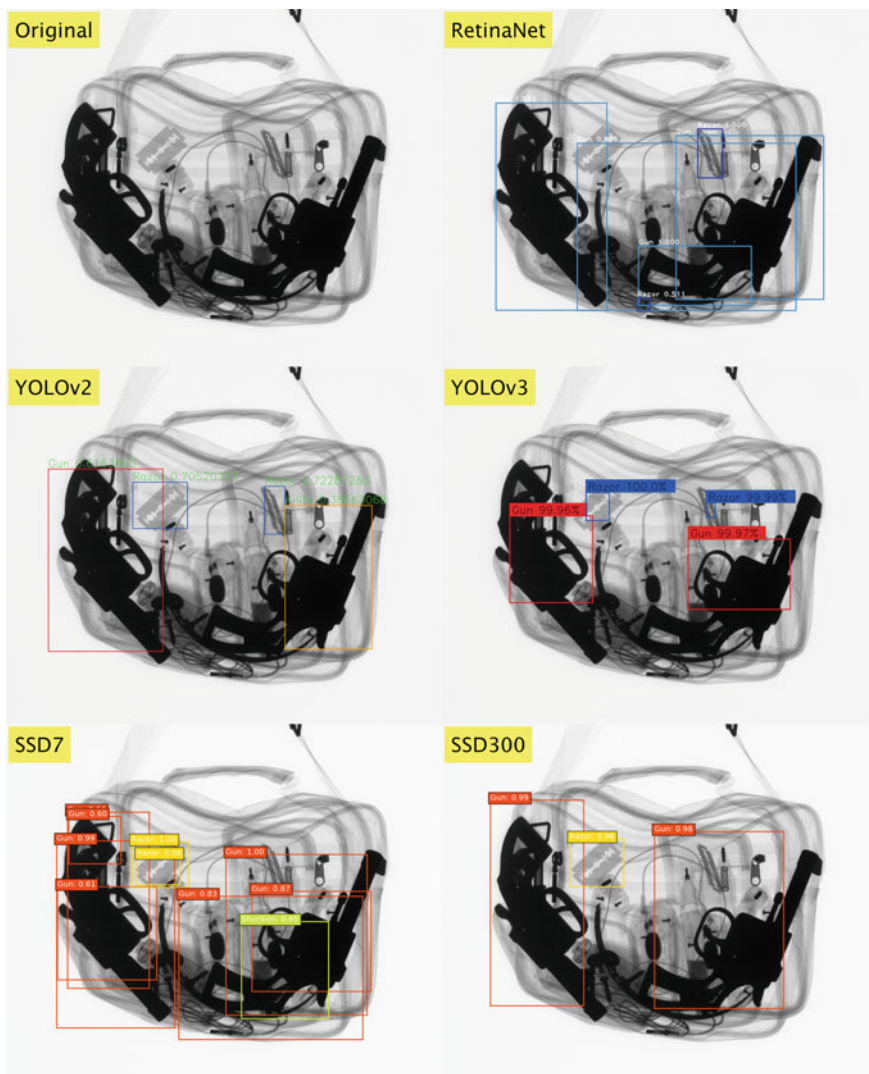


Fig. 7.36 Object detection on GDXray+ image B0046_0151



Fig. 7.37 Object detection on GD \times ray+ image B0046_0184

7.7.6 Examples of Object Detection

In this section, we show detection of threat objects in GD \times ray+ using the implementation of [56].¹⁷ This implementation contains the following four detectors for use in the detection of threat objects in baggage inspection:

¹⁷See [https://github.com/dlsaavedra/Detector_GD \$\times\$ ray](https://github.com/dlsaavedra/Detector_GD\timesray). In addition, all examples are implemented in Google Colab on <https://github.com/computervision-xray-testing/pyxvis>.

- Yolov2 (see Sect. 7.7.3 and Example 7.7)
- Yolov3 (see Sect. 7.7.3 and Example 7.8)
- SSD7 (see Sect. 7.7.4 and Example 7.9)
- SSD300 (see Sect. 7.7.4 and Example 7.10)
- RetinaNet (see Sect. 7.7.5 and Example 7.11)

In order to compare the implemented methods, in Figs. 7.36 and 7.37 we can observe the performance of each method visually. The reader is referred to [56] for more details of the training and the evaluation protocol.

7.8 Summary

In this chapter, we review many relevant concepts of deep learning that can be used in computer vision for X-ray testing. We covered the theory and practice of deep learning techniques in real X-ray testing problems. The chapter explained

- Neural Networks,
- Convolutional Neural Network (CNN) that can be used in classification problems,
- Pre-trained Models,
- Transfer Learning that is used in sophisticated models,
- Generative Adversarial Networks (GANs) to generate synthetic images, and
- modern detection methods that are used to classify and localize objects in an image.

In addition, for every method, we gave not only basic concepts but also practical details in real X-ray testing examples implemented in Python.

References

1. Aggarwal, C.C.: *Neural Networks and Deep Learning*. Springer International Publishing, Cham (2018)
2. Akçay, S., Kundegorski, M.E., Devereux, M., Breckon, T.P.: Transfer learning using convolutional neural networks for object classification within X-ray baggage security imagery. In: 2016 IEEE International Conference on Image Processing (ICIP), pp. 1057–1061. IEEE (2016)
3. Bengio, Y., Courville, A., Vincent, P.: Representation learning: a review and new perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.* **35**(8), 1798–1828 (2013)
4. Bishop, C.: *Neural Networks for Pattern Recognition*. Oxford University Press, Oxford (2005)
5. Bishop, C.: *Pattern Recognition and Machine Learning*. Springer, Berlin (2006)
6. Bochkovskiy, A., Wang, C.Y., Liao, H.Y.M.: YOLOv4: optimal speed and accuracy of object detection (2020)
7. Brock, A., Donahue, J., Simonyan, K.: Large scale GAN training for high fidelity natural image synthesis. In: *The International Conference on Learning Representations (ICLR 2019)*, pp. 1–35 (2019)

8. Chollet, F.: Xception: deep learning with depthwise separable convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1251–1258 (2017)
9. Creswell, A., White, T., Dumoulin, V., Arulkumaran, K., Sengupta, B., Bharath, A.A.: Generative adversarial networks: an overview. *IEEE Signal Process. Mag.* **35**(1), 53–65 (2018)
10. Deng, J., Guo, J., Xue, N., Zafeiriou, S.: ArcFace: additive angular margin loss for deep face recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4690–4699 (2019)
11. Du, W., Shen, H., Fu, J., Zhang, G., He, Q.: Approaches for improvement of the X-ray image defect detection of automobile casting aluminum parts based on deep learning. *NDT & E Int.* **107**, 102,144 (2019)
12. Du, W., Shen, H., Fu, J., Zhang, G., Shi, X., He, Q.: Automated detection of defects with low semantic information in X-ray images based on deep learning. *J. Intell. Manuf.* 1–16 (2020)
13. Erhan, D., Bengio, Y., Courville, A., Vincent, P.: Visualizing higher-layer features of a deep network. Technical report, Univeriste de Montreal (2009)
14. Esteva, A., Kuprel, B., Novoa, R.A., Ko, J., Swetter, S.M., Blau, H.M., Thrun, S.: Dermatologist-level classification of skin cancer with deep neural networks. *Nature* **542**(7639), 115–118 (2017)
15. Girshick, R.: Fast R-CNN. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 1440–1448 (2015)
16. Girshick, R., Donahue, J., Darrell, T., Malik, J.: Rich feature hierarchies for accurate object detection and semantic segmentation. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 580–587 (2014)
17. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y.: Generative adversarial nets. In: *Advances in Neural Information Processing Systems*, pp. 2672–2680 (2014)
18. Goodfellow, I., Bengio, Y., Courville, A.: *Deep Learning*. MIT Press, Cambridge (2016)
19. Hassabis, D., Kumaran, D., Summerfield, C., Botvinick, M.: Neuroscience-inspired artificial intelligence. *Neuron* **95**(2), 245–258 (2017)
20. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. *CoRR* (2015). [arXiv:1512.03385](https://arxiv.org/abs/1512.03385)
21. He, K., Zhang, X., Ren, S., Sun, J.: Spatial pyramid pooling in deep convolutional networks for visual recognition. *IEEE Trans. Pattern Anal. Mach. Intell.* **37**(9), 1904–1916 (2015)
22. He, K., Gkioxari, G., Dollár, P., Girshick, R.: Mask R-CNN. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 2961–2969 (2017)
23. Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., Adam, H.: MobileNets: efficient convolutional neural networks for mobile vision applications (2017). [arXiv:1704.04861](https://arxiv.org/abs/1704.04861)
24. Huang, G., Liu, Z., Weinberger, K.Q.: Densely connected convolutional networks. *CoRR* (2016). [arXiv:1608.06993](https://arxiv.org/abs/1608.06993)
25. Iandola, F.N., Moskewicz, M.W., Ashraf, K., Han, S., Dally, W.J., Keutzer, K.: SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR* (2016). [arXiv:1602.07360](https://arxiv.org/abs/1602.07360)
26. Jiang, X., Hou, Y., Zhang, D., Feng, X.: Deep learning in face recognition across variations in pose and illumination. *Deep Learning in Object Detection and Recognition*, pp. 59–90. Springer, Berlin (2019)
27. Karras, T., Laine, S., Aila, T.: A style-based generator architecture for generative adversarial networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4401–4410 (2019)
28. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization (2014). [arXiv:1412.6980](https://arxiv.org/abs/1412.6980)
29. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. In: *NIPS*, pp. 1106–1114 (2012)
30. LeCun, Y., Bottou, L., Bengio, Y.: Gradient-based learning applied to document recognition. In: *Proceedings of the Third International Conference on Research in Air Transportation* (1998)

31. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. *Nature* **521**(7553), 436–444 (2015)
32. Lin, T., Goyal, P., Girshick, R.B., He, K., Dollár, P.: Focal loss for dense object detection. *CoRR* (2017). [arXiv:1708.02002](https://arxiv.org/abs/1708.02002)
33. Lin, T.Y., Dollár, P., Girshick, R., He, K., Hariharan, B., Belongie, S.: Feature pyramid networks for object detection. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2117–2125 (2017)
34. Liu, B., Zhang, X., Gao, Z., Chen, L.: Weld defect images classification with VGG16-based neural network. In: *International Forum on Digital TV and Wireless Multimedia Communications*, pp. 215–223. Springer (2017)
35. Liu, L., Ouyang, W., Wang, X., Fieguth, P., Chen, J., Liu, X., Pietikäinen, M.: Deep learning for generic object detection: a survey. *Int. J. Comput. Vis.* **128**(2), 261–318 (2020)
36. Liu, S., Qi, L., Qin, H., Shi, J., Jia, J.: Path aggregation network for instance segmentation. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 8759–8768 (2018)
37. Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S.E., Fu, C., Berg, A.C.: SSD: single shot multibox detector. *CoRR* (2015). [arXiv:1512.02325](https://arxiv.org/abs/1512.02325)
38. Mery, D.: Aluminum casting inspection using deep learning: a method based on convolutional neural networks. *J. Nondestruct. Eval.* **39**(1), 12 (2020)
39. Mery, D., Arteta, C.: Automatic defect recognition in X-ray testing using computer vision. In: *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 1026–1035. IEEE (2017)
40. Mery, D., Riffo, V., Zscherpel, U., Mondragón, G., Lillo, I., Zuccar, I., Lobel, H., Carrasco, M.: GDxray: the database of X-ray images for nondestructive testing. *J. Nondestruct. Eval.* **34**(4), 1–12 (2015)
41. Mery, D., Svec, E., Arias, M., Riffo, V., Saavedra, J.M., Banerjee, S.: Modern computer vision techniques for X-ray testing in baggage inspection. *IEEE Trans. Syst. Man Cybern.: Syst.* **47**(4), 682–692 (2016)
42. Miao, C., Xie, L., Wan, F., Su, C., Liu, H., Jiao, J., Ye, Q.: SIXray: a large-scale security inspection X-ray benchmark for prohibited item discovery in overlapping images. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2119–2128 (2019)
43. Mitchell, T.: *Machine Learning*. McGraw-Hill, Boston (1997)
44. Nagpal, K., Foote, D., Liu, Y., Chen, P.H.C., Wulczyn, E., Tan, F., Olson, N., Smith, J.L., Mohtashamian, A., Wren, J.H., et al.: Development and validation of a deep learning algorithm for improving Gleason scoring of prostate cancer. *NPJ Digit. Med.* **2**(1), 1–10 (2019)
45. Nair, V., Hinton, G.E.: Rectified linear units improve restricted Boltzmann machines. In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 807–814 (2010)
46. Niu, S., Li, B., Wang, X., Lin, H.: Defect image sample generation with GAN for improving defect recognition. *IEEE Trans. Autom. Sci. Eng.* (2020)
47. Oliphant, T.E.: *A Guide to NumPy*, vol. 1. Trelgol Publishing, New York (2006)
48. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: machine learning in Python. *J. Mach. Learn. Res.* **12**, 2825–2830 (2011)
49. Radford, A., Metz, L., Chintala, S.: Unsupervised representation learning with deep convolutional generative adversarial networks. *CoRR* (2015). [arXiv:1511.06434](https://arxiv.org/abs/1511.06434)
50. Redmon, J., Farhadi, A.: YOLO9000: better, faster, stronger. *CoRR* (2016). [arXiv:1612.08242](https://arxiv.org/abs/1612.08242)
51. Redmon, J., Farhadi, A.: Yolov3: an incremental improvement. *CoRR* (2018). [arXiv:1804.02767](https://arxiv.org/abs/1804.02767)
52. Redmon, J., Divvala, S.K., Girshick, R.B., Farhadi, A.: You only look once: unified, real-time object detection. *CoRR* (2015). [arXiv:1506.02640](https://arxiv.org/abs/1506.02640)
53. Ren, S., He, K., Girshick, R., Sun, J.: Faster R-CNN: towards real-time object detection with region proposal networks. In: *Advances in Neural Information Processing Systems*, pp. 91–99 (2015)

54. Riffo, V., Mery, D.: Automated detection of threat objects using adapted implicit shape model. *IEEE Trans. Syst. Man Cybern.: Syst.* **46**(4), 472–482 (2016)
55. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A., Fei-Fei, L.: ImageNet large scale visual recognition challenge. *Int. J. Comput. Vis.* **115**(3), 211–252 (2015). <https://doi.org/10.1007/s11263-015-0816-y>
56. Saavedra, D., Banerjee, S., Mery, D.: Detection of threat objects in baggage inspection with X-ray images using deep learning. *Neural Comput. Appl.* pp. 1–17. Springer (2020)
57. Shrestha, A., Mahmood, A.: Review of deep learning algorithms and architectures. *IEEE Access* **7**, 53040–53065 (2019)
58. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. *CoRR* (2014). [arXiv:1409.1556](https://arxiv.org/abs/1409.1556)
59. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.* **15**, 1929–1958 (2014)
60. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper with convolutions. In: *CVPR 2015* (2015)
61. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z.: Rethinking the inception architecture for computer vision. *CoRR* (2015). [arXiv:1512.00567](https://arxiv.org/abs/1512.00567)
62. Tang, Z., Tian, E., Wang, Y., Wang, L., Yang, T.: Non-destructive defect detection in castings by using spatial attention bilinear convolutional neural network. *IEEE Trans. Ind. Inform.* 1–1 (2020)
63. Uijlings, J.R., Van De Sande, K.E., Gevers, T., Smeulders, A.W.: Selective search for object recognition. *Int. J. Comput. Vis.* **104**(2), 154–171 (2013)
64. Wang, C.Y., Liao, H.Y.M., Yeh, I.H., Wu, Y.H., Chen, P.Y., Hsieh, J.W.: CSPNet: A new backbone that can enhance learning capability of CNN (2019). [arXiv:1911.11929](https://arxiv.org/abs/1911.11929)
65. Yang, J., Zhao, Z., Zhang, H., Shi, Y.: Data augmentation for X-ray prohibited item images using generative adversarial networks. *IEEE Access* **7**, 28894–28902 (2019)
66. Yosinski, J., Clune, J., Bengio, Y., Lipson, H.: How transferable are features in deep neural networks? In: *Advances in Neural Information Processing Systems*, pp. 3320–3328 (2014)
67. Zeiler, M.D., Fergus, R.: Visualizing and understanding convolutional networks. In: *European Conference on Computer Vision*, pp. 818–833. Springer (2014)
68. Zhang, A., Lipton, Z.C., Li, M., Smola, A.J.: Dive into deep learning. Unpublished draft. Retrieved **3**, 319 (2019)
69. Zhang, X., Zhou, X., Lin, M., Sun, J.: ShuffleNet: an extremely efficient convolutional neural network for mobile devices. *CoRR* (2017). [arXiv:1707.01083](https://arxiv.org/abs/1707.01083)
70. Zhao, Z., Zhang, H., Yang, J.: A GAN-based image generation method for X-ray security prohibited items. In: *Chinese Conference on Pattern Recognition and Computer Vision (PRCV)*, pp. 420–430. Springer (2018)
71. Zhao, Z., Zheng, P., Xu, S., Wu, X.: Object detection with deep learning: a review. *IEEE Trans. Neural Netw. Learn. Syst.* **30**(11), 3212–3232 (2019). <https://doi.org/10.1109/TNNLS.2018.2876865>
72. Zhu, Y., Zhang, Y., Zhang, H., Yang, J., Zhao, Z.: Data augmentation of X-ray images in baggage inspection based on generative adversarial networks. *IEEE Access* **8**, 86536–86544 (2020)