ML for Process Industry Series

2024 Edition

Machine Learning in Python for Visual and Acoustic Data-based Process Monitoring

A short beginner's guide to deep learning-based computer vision and abnormal sound detection

Ankur Kumar

Machine Learning in Python for Visual and Acoustic Data-based Process Monitoring

A short beginner’s guide to deep learning-based computer vision and abnormal sound detection

Ankur Kumar

`

Dedicated to my spouse, family, friends, motherland, and all the data-science enthusiasts

`

विद्या प्रशस्यते लोकैः विद्या सर्वत्र गौरवा ।

विद्यया लभते सर्वं विद्वान सर्वत्र पूज्यते ।।

Knowledge is extolled everywhere,
knowledge is considered great everywhere;
One can attain everything with the help of knowledge,
and a knowledged person is respected everywhere .

- *A popular Sanskrit shloka*

`

Machine Learning in Python for Visual and Acoustic Data-based Process Monitoring

www.MLforPSE.com

Plant image on cover page obtained from https://pixabay.com/.

.

To request permissions, contact the authors at MLforPSE@gmail.com

First published: May 2024

`

About the Author

Ankur Kumar holds a PhD degree (2016) in Process Systems Engineering from the University of Texas at Austin and a bachelor's degree (2012) in Chemical Engineering from the Indian Institute of Technology Bombay. He currently works at Linde in the Advanced Digital Technologies & Systems Group in Linde's Center of Excellence, where he has developed several in-house machine learning-based monitoring and process control solutions for Linde's hydrogen and air-separation plants. Ankur's tools have won several awards both within and outside Linde. One of his tools, PlantWatch (a plantwide fault detection and diagnosis tool), received the 2021 Industry 4.0 Award by the Confederation of Industry of the Czech Republic. Ankur has authored or co-authored several peer-reviewed journal papers (in the areas of data-driven process modeling and monitoring), is a frequent reviewer for many top-ranked Journals, and has served as Session Chair at several international conferences. Ankur served as an Associate Editor of the Journal of Process Control from 2019 to 2021, and currently serves on the Editorial Advisory Board of Industrial & Engineering Chemistry Research Journal. Most recently, he was included in the 'Engineering Leaders Under 40, Class of 2023' by *Plant Engineering* Magazine.

`

Note to the readers

Jupyter notebooks and Python scripts with complete code implementations are available for download at github.com/ML-PSE/Machine_Learning_with_Visual_Acoustics_Process_Data. Code updates when necessary, will be made and updated on the GitHub repository. Updates to the book's text material will be available on Leanpub (www.leanpub.com) and Google Play (https://play.google.com/store/books). We would greatly appreciate any information about any corrections and/or typographical errors in the book.

`

Series Introduction

In the 21st century, data science has become an integral part of the work culture in the manufacturing industry and process industry is no exception to this modern phenomenon. From process monitoring to predictive maintenance, fault diagnosis to advanced process control, machine learning-based solutions are being used to achieve higher process reliability and efficiency. However, few books are available that adequately cater to the needs of budding process data scientists. The scant available resources include: 1) generic data science books that fail to account for the specific characteristics and needs of process plants 2) process domain-specific books with rigorous and verbose treatment of underlying mathematical details that become too theoretical for industrial practitioners. Understandably, this leaves a lot to be desired. Books are sought that have process systems in the backdrop, stress application aspects, and provide a guided tour of ML techniques that have proven useful in process industry. This series '***Machine Learning for Process Industry***' addresses this gap to reduce the barrier-to-entry for those new to process data science.

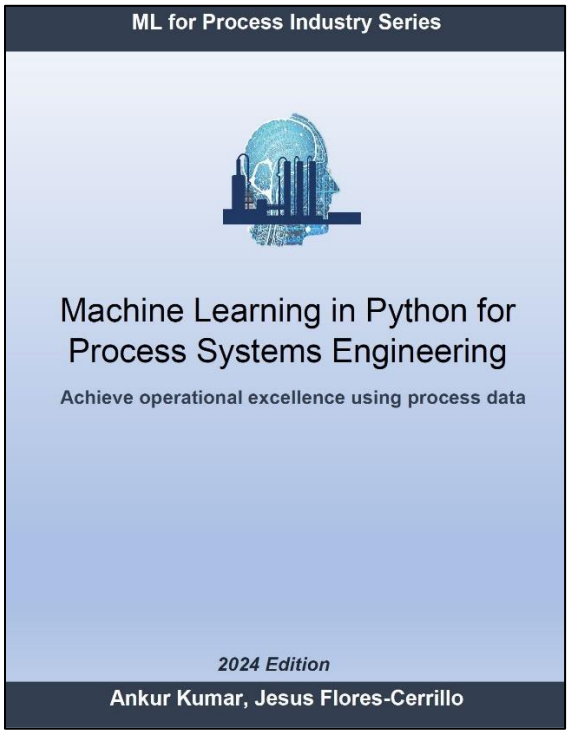
The first book of the series '***Machine Learning in Python for Process Systems Engineering***' covers the basic foundations of machine learning and provides an overview of broad spectrum of ML methods primarily suited for static systems. Step-by-step guidance on building ML solutions for process monitoring, soft sensing, predictive maintenance, etc. are provided using real process datasets. Aspects relevant to process systems such as modeling correlated variables via PCA/PLS, handling outliers in noisy multidimensional datasets, controlling processes using reinforcement learning, etc. are covered. The second book of the series '***Machine Learning in Python for Dynamic Process Systems***' focuses on dynamic systems and provides a guided tour along the wide range of available dynamic modeling choices. Emphasis is paid to both the classical methods (ARX, CVA, ARMAX, OE, etc.) and modern neural network methods. Applications on time series analysis, noise modeling, system identification, and process fault detection are illustrated with examples. The third book of the series '**Machine Learning in Python for Process and Equipment Condition Monitoring, and Predictive Maintenance**' takes a deep dive into an important application area of ML, viz, prognostics and health management. ML methods that are widely employed for the different aspects of plant health management, namely, fault detection, fault isolation, fault diagnosis, and fault prognosis, are covered in detail. Emphasis is placed on conceptual understanding and practical implementations. This fourth book of the series is a quick foray into the world of deep learning-based computer vision and abnormal equipment sound detection. The readers are introduced to the ease with which powerful equipment and product quality monitoring solutions can be built using sound and visual data. Future books of the series will continue to focus on other aspects and needs of process industry. It is hoped that these books can help process data scientists find innovative ML solutions to the real-world problems faced by the process industry.

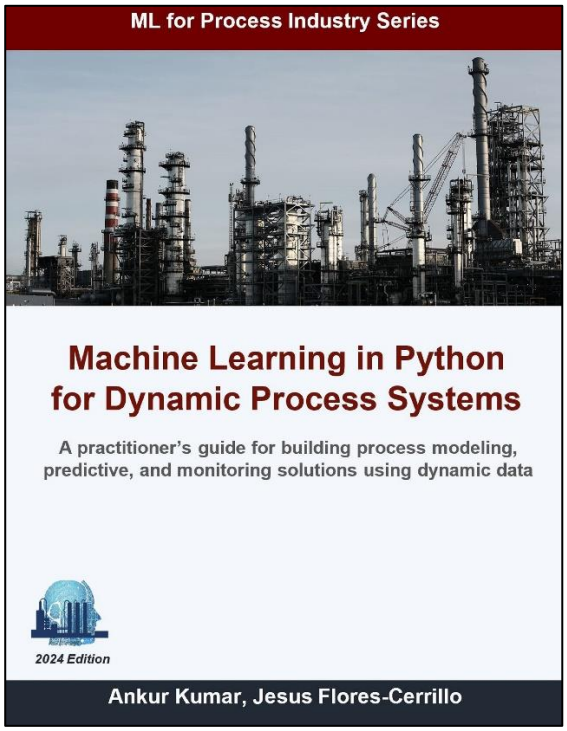
`

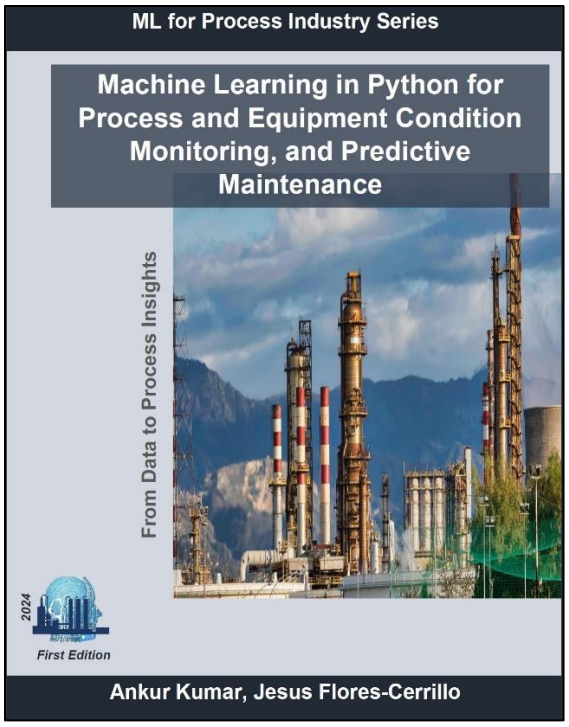
With the growing trend in usage of machine learning in the process industry, there is growing demand for process domain experts/process engineers with data science/ML skills. These books have been written to cover the existing gap in ML resources for such process data scientists. Specifically, books of this series will be useful to budding process data scientists, practicing process engineers looking to ‘pick up’ machine learning, and data scientists looking to understand the needs and characteristics of process systems. With the focus on practical guidelines and industrial-scale case studies, we hope that these books lead to wider spread of data science in the process industry.

`

Other book(s) from the series
(https://MLforPSE.com/books/)







`

Preface

In today's world, it is hard to be unaware of the remarkable advances artificial intelligence has been making; new AI tools come up every day (such as ChatGPT, Sora, etc.) that change the way we interact with machines and the world around us. Thankfully, the process industry is not lagging behind in riding this wave of digital innovations. 'Smart manufacturing' and 'digitalization' are the governing mantras among the industry executives. Under the broad umbrella of Industry 4.0, data-driven solutions are increasingly being deployed to optimize and monitor every step along a production line. There is a general consensus that every single bit of data is a resource which needs to be utilized to obtain better process insights. In this context, to assist young process data scientists (PDSs) get onto this 'AI for process industry' bandwagon, the previous books of the series covered how ML-based plant health management solutions are built using traditional process data (such as flow, temperature, pressure, etc.). This book provides a cursory coverage of how visual and acoustic process data are utilized for process monitoring. The justification for using visual and acoustic process data is straightforward: an experienced technician can infer a machine fault by listening to the sound it is making and promptly catch a product quality issue by looking at the final product. Tools like ChatGPT have shown that artificial neural networks can be imparted human-like intelligence via deep learning (DL). Correspondingly, enterprises are deploying DL-based computer vision (CV) and acoustic monitoring to boost the automated surveillance of manufacturing plants while reducing labor costs. Although (scattered) resources are available on internet on DL-based visual and acoustic analytics, the learning curve can be steep for a beginner PDS. Therefore, this book provides an introduction to how computer vision and acoustic monitoring solutions are built using deep learning for a manufacturing plant.

Computer vision and acoustic analysis are, inarguably, specialized fields of practice; developing ML solutions using visual and acoustic data require careful feature engineering by experts. Fortunately, the rise of deep learning has made the task of PDSs easier. The same DL concepts that are used to estimate the RUL of failing machines (as we saw in the previous books of the series) can be used for building CV and abnormal sound detection (ASD) solutions without explicit feature engineering. However, deep learning-based CV (and ASD) can be scary if you are a beginner: you may find that your PC is not powerful enough to train the neural network; you may get overwhelmed with the different modeling paradigms (and different terms like LeNet, AlexNet, ResNet, etc.); you may find it strange that using pre-trained models (via transfer learning) is the dominant approach for quickly developing CV solutions. Therefore, this short book is to help you take your first step and provide you with enough familiarity to enable you to navigate the DL-based CV and ASD world confidently.

`

The broad objectives of the book can be summarized as follows:

- provide familiarity to deep learning-based computer vision and equipment acoustic monitoring
- provide a gentle introduction to CNNs
- provide a quick introduction to Google Colab as the environment for computationally intensive ANN training
- showcase applications of computer vision for steel product fault classification
- showcase application of acoustic monitoring for air compressor fault classification

Computer vision and equipment acoustic-based process monitoring are not yet among the mainstream technologies employed in process industry. Therefore, a conscious decision was made to keep this book at the beginner level and not weigh readers down with too many advanced concepts which can become overwhelming. Complete code implementations have been provided in the GitHub repository. We are quite confident that this text will get the beginner PDSs excited about these technologies, and encourage them to build upon the concepts gained from the book and develop interesting monitoring solutions for their manufacturing facilities.

Who should read this book

This book is meant to give an introductory coverage of convolutional neural networks and its applications for computer vision and equipment acoustic monitoring. The following categories of readers will find the book useful:

1) Process data scientists new to the field of computer vision and acoustic monitoring
2) Practicing process data scientists looking for an introductory resource on CNNs
3) Process engineers or process engineering students making their entries into the world of data science

Pre-requisites

Prior experience with machine learning, Python, and artificial neural networks is assumed.

`

Table of Contents

Chapter 1

Introduction to Process Monitoring via Computer Vision and Abnormal Sound Detection

It is not easy to ensure continuous reliable plant operations with optimal efficiency and high product quality standards; dedicated teams of plant operators and engineers work round the clock to efficiently run manufacturing plants. In Industry 4.0 era, there has been an enhanced push to digitalize plant operations to lighten the burden of plant personnel. Most of the modern production facilities employ hundreds of sensors to keep a tab on plant operations in real-time. Furthermore, smart digital process monitoring tools are utilized to continuously monitor plant performance. Sensors that provide flow, temperature, pressure, level, composition, and vibration measurements have traditionally been employed by process industry. In recent times, image and sound sensors are increasingly being employed to aid plant health management. The justification behind this trend is simple: an expert technician can often immediately tell if an equipment needs maintenance by listening to the sound it is making or if product quality has been compromised by looking at the product. Therefore, smart ML solutions are being built that utilize the image and sound data, and mimic the intelligence of expert technicians.

The emergence of deep learning has given wings to the field of computer vision (CV) which is field of artificial intelligence that enables computers to make inferences using visual inputs. Automated quality inspection is one such popular application of CV. CV techniques find usage in analysis of equipment sound as well and has enabled abnormal sound detection (ASD)-based predictive maintenance solutions. Inarguably, CV and acoustic signal analysis are highly specialized areas; however, deep learning enables development of CV and ASD solutions using raw data directly and bypassing the need for explicit feature engineering (which often requires subject matter expertise). Correspondingly, CV and acoustic monitoring have become critical components of a modern process data scientist's toolkit.

While previous books of the series have focused on plant health management using traditional signals, this short book provides an introduction to deep learning-based CV and ASD. The current chapter covers the following topics

- Introduction to process monitoring techniques
- CV use cases and workflow
- ASD use cases and workflow

1.1 Process Monitoring Techniques

Process plants are prone to different types of failures; plants may experience pipe leaks, motor bearing issues, heat exchanger fouling, sensor failures, equipment surface hotspots, valve failures, etc. Consequently, different techniques have been devised over the years to monitor the 'health' of plant equipment using different types of sensors. Figure 1.1 lists some of these commonly employed techniques in the process industry. The methods include, amongst others, usage of transducers to monitor machine vibrations and comparing process stream conditions (flow, temperature, pressure, etc.) against expected values. Within this list lie the practices of just 'looking' at and 'listening' to the plant equipment. In the pre-computer era, plant operators would go around the plant and visually inspect plant equipment and listen to the 'plant sound'. Well, these practices are making a comeback with a twist that the 'looking' and 'listening' are done by computers. While cameras and microphones act as eyes and ears, the deep learning algorithms mimic the human intelligence of plant operators in inferring the presence of abnormalities.

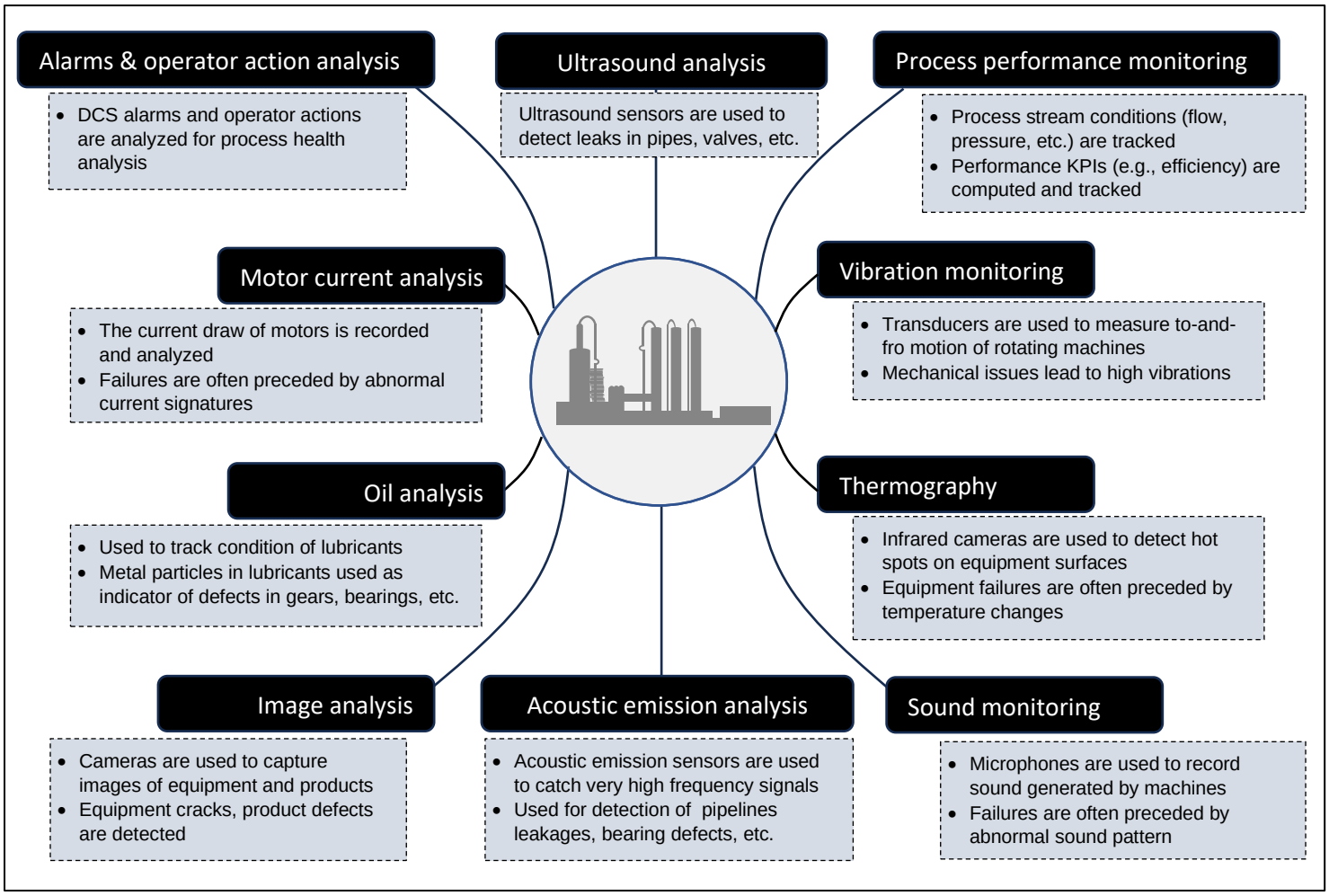


Figure 1.1: Common process/equipment condition monitoring techniques

Using cameras and microphones to monitor an equipment are attractive due to the non-intrusive nature of these sensors: equipment does not have to be taken offline for installation of these sensors. Figure 1.2 shows a few use-cases of computer vision in manufacturing.

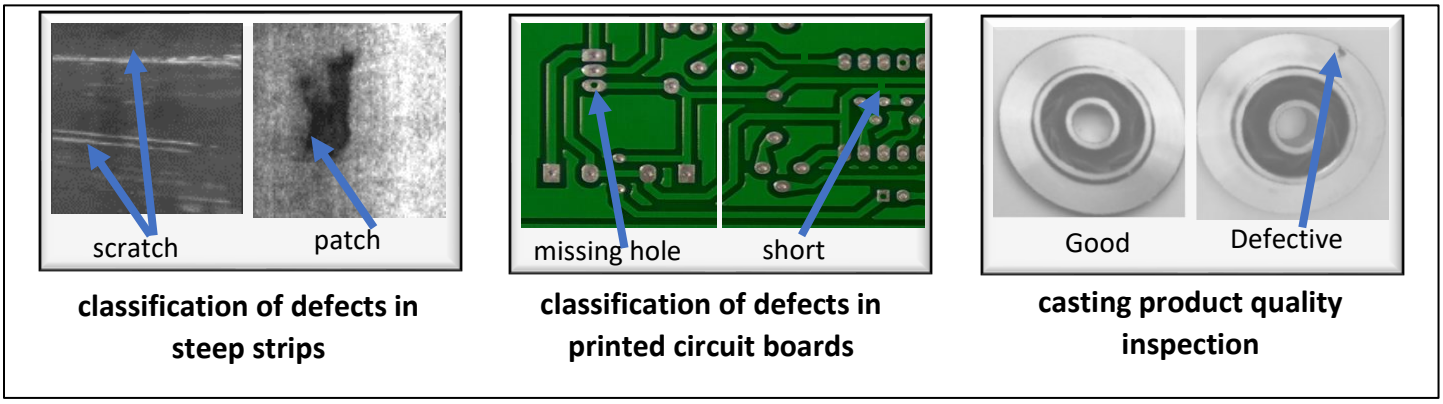
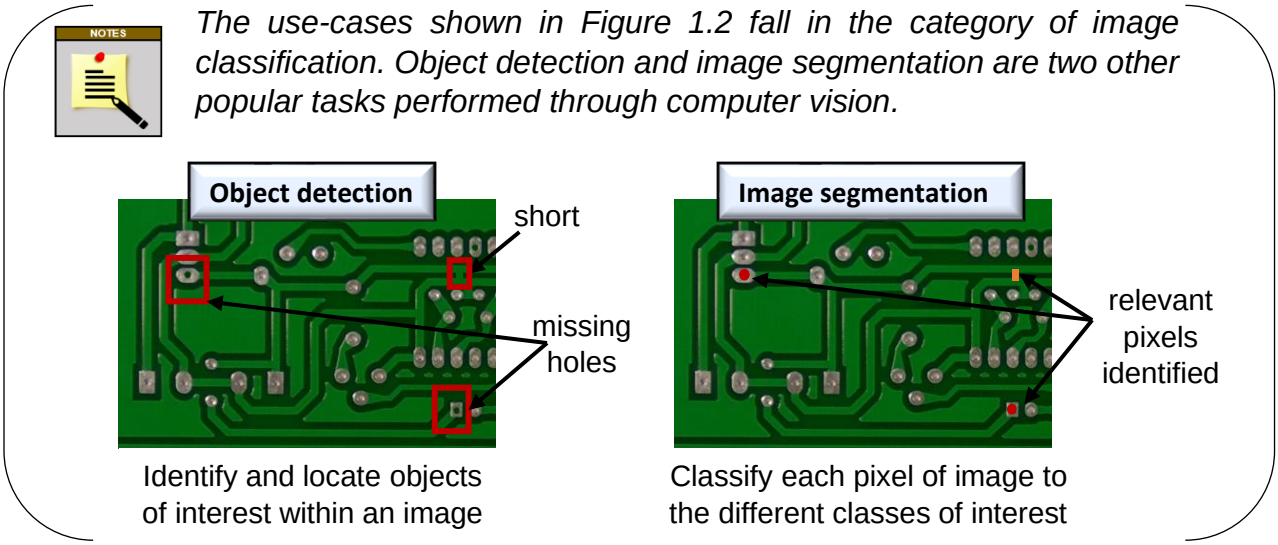
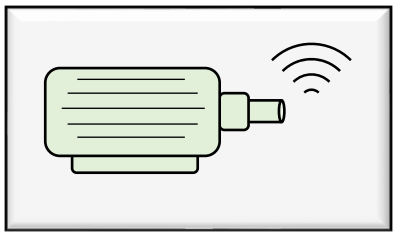


Figure 1.2: Examples of computer vision use cases in manufacturing[1]

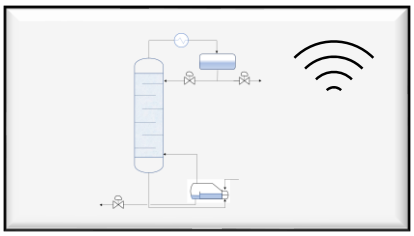


Like CV, acoustic monitoring has found several use-cases in process industry as well; Figure 1.3 shows some of them. When equipment such as motors experience mechanical issues (misalignment, looseness, imbalance, etc.), they produce abnormal sound; these abnormal sound patterns act as leading indicators of impending failures. Consequently, predictive maintenance applications are often built using equipment acoustics.

[1] Steel strip images made available at http://faculty.neu.edu.cn/songkechen/zh_cn/zhym/263269/list/index.htm by Prof. Ke-Chen Song at Northeastern University, China.
PCB images are publicly available at https://github.com/Ironbrotherstyle/PCB-DATASET.
Casting product images are made available under CC BY-NC-ND 4.0 license by Ravirajsinh Dabhi at https://www.kaggle.com/datasets/ravirajsinh45/real-life-industrial-dataset-of-casting-product/data.



detection of mechanical issues in motors



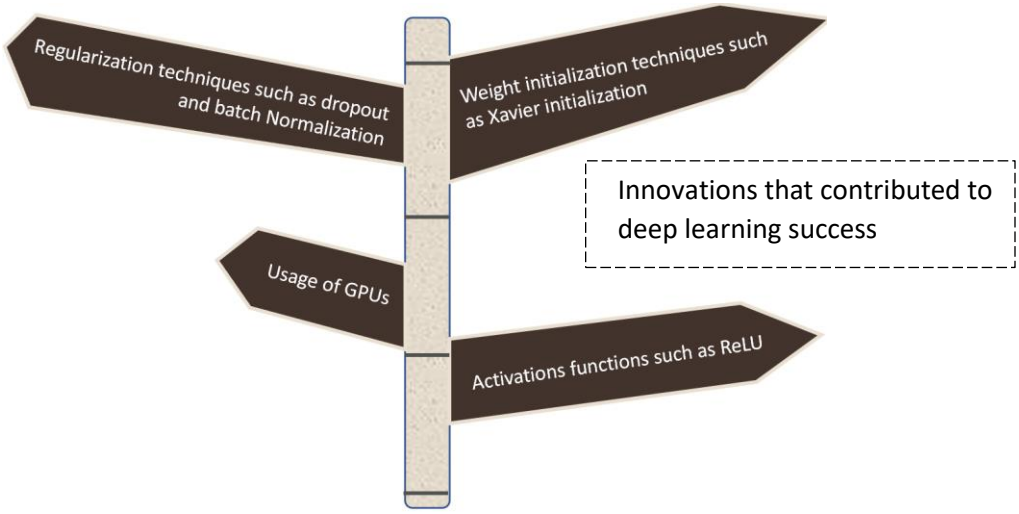
detection of abnormal distillation column operations (such as flooding)

detection of pipeline leakages via high-frequency acoustic sensors

Figure 1.3: Examples of acoustics-based condition monitoring in process industry

Process monitoring in the deep learning era

Computer vision and acoustic monitoring owe the resurgence in their usage for plant health management due to the recent computational advances in deep learning. Deep learning allows features to be extracted automatically from raw data and therefore has significantly changed how process data are analyzed for insights. Technically, deep learning refers to usage of artificial neural networks with several hidden layers. Over the last decade, several algorithmic innovations have taken place that have made deep learning computationally tractable.



In addition to the enhanced tractability of deep models' training, the emergence of user-friendly frameworks such as Keras and PyTorch that allow creation of deep neural networks quickly in just a few lines of code has led to democratization of process data science.

1.2 CV Workflow in Manufacturing Environment

Figure 1.4 below shows a typical workflow for automated visual inspection. As is evident, the workflow is the same as any other ML exercise.

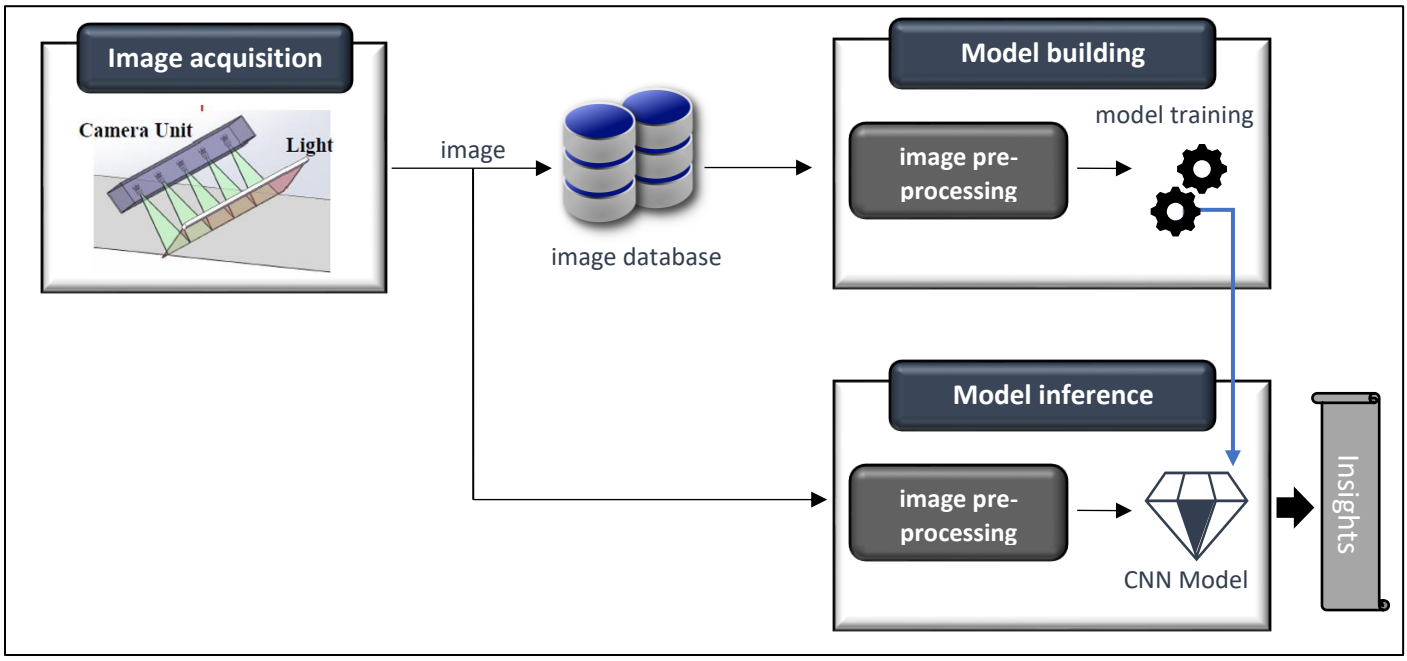


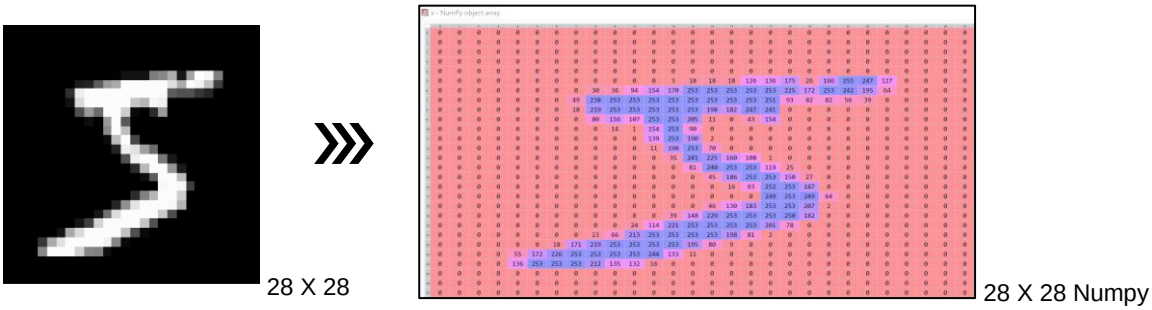
Figure 1.4: Typical computer vision workflow in manufacturing environment[2]

Arguably, the most critical phase in this workflow is acquisition of high-quality images. As shown in Figure 1.4, elaborate optical illumination platform is often setup to ensure high contrast images with minimal interference of environmental light. Acquired images are then sent for pre-processing to improve image quality and help CV models understand the images better. During model training, the common pre-processing steps include image resizing, augmentations, denoising, etc. Many CV models have strict specification for the input image size and therefore images are resized accordingly. Image augmentation entails artificial generation of new images using the original input images. This helps to overcome the limitations of small-sized datasets. DL models are 'data-hungry' and in the manufacturing industry world, you will rarely have a large-sized dataset. Therefore, you will often use techniques such as cropping, rotation, zooming, etc., to create different versions of your images to train your DL model. Post pre-processing, images are used to train models. The most common architecture employed for CV tasks is convolutional neural network which specialize in handling grid-like data. We will talk more about CNNs in Chapter 2.

[2] Camera Unit/Light picture taken from Lv et al., Deep Metallic Surface Defect Detection: The New Benchmark and Detection Network. *Sensors*, 2020 shared under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).

Working with images in Python

An image is simply a grid of numbers for computers. Consider the following grayscale image (taken from MINIST dataset provided in Keras) of size 28 X 28 pixels.

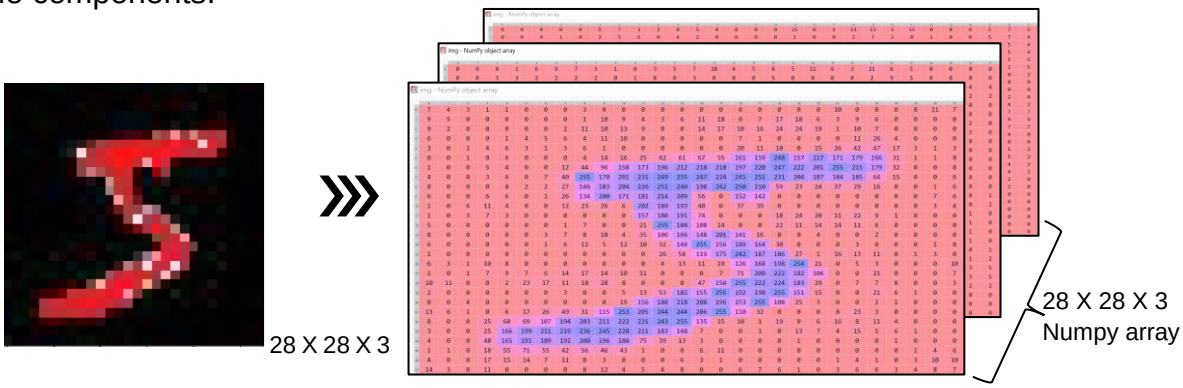


28 X 28

28 X 28 Numpy array

- each pixel takes a value (between 0 and 255) denoting the intensity of light

A color image has three of these 2D channels corresponding to the Red, Green, and Blue components.



28 X 28 X 3

28 X 28 X 3 Numpy array

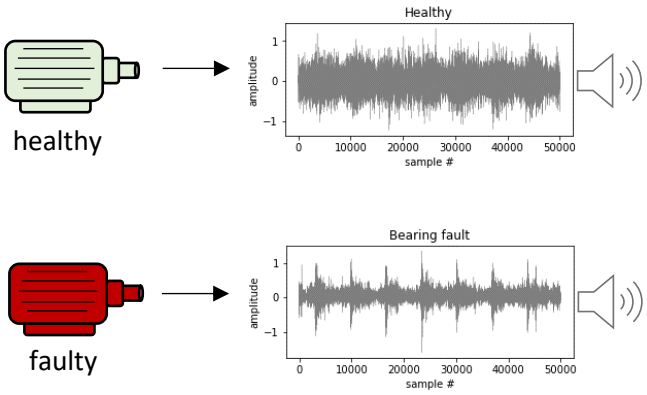
- each pixel is assigned a tuple of 3 values corresponding to the red, green, and blue components, respectively
- E.g., (0,0,0) ⇒ black pixel
  (255,0,0) ⇒ red pixel
  (255, 255, 255) ⇒ white pixel

The standard Python library for image manipulation is *Pillow* which is an updated version of *PIL* (Python image library) library. Other popular options for image handling in Python are OpenCV and Keras API. The examples below show how to load, manipulate, and show images using *Pillow* and *Matplotlib*.

```
# load image using Pillow
from PIL import image
img = Image.open('digit_color.jpeg')
print(img.size)

>>> (28, 28)
img.show() # displays image using your computer's default application for photos

# construct NumpPy array from image object
import numpy as np
imgData = np.asarray(img)
print(imgData.shape)

>>> (28, 28, 3)

# resize image and save
img_resized = img.resize((7,7)) # from 28 X 28 pixels to 7 X 7 pixels
img_resized.save('digit_color_resized.jpeg')

#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#  load image as NumPy array directly using Matplotlib
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
from matplotlib import image, pyplot
imgData2 = image.imread('digit_color.jpeg') # loads image as a 3D NumPy array
pyplot.imshow(img) # displays image within a Matplotlib frame
```

1.3 ASD Workflow in Manufacturing Environment

Consider the following two sound clips recorded from a faulty and a healthy air compressor.




```
import numpy as np
import sounddevice as sd

# clipPath = "AirCompressor_Data/Healthy/preprocess_Reading1.dat"
clipPath = "AirCompressor_Data/Bearing/preprocess_Reading1.dat"

data = np.loadtxt(clipPath, delimiter=',')
fs = 50000/3 # 50000 samples per 3 seconds
sd.play(data, fs)
```

Go ahead and execute the shown code with the sound clips[3] (code provided in the GitHub repository). Do you hear any difference in how the two machines sound? The difference is very obvious. However, how do you make a computer understand this difference? A sound is simply vibrations captured by air; therefore, the techniques that we learnt in Book 3 of the series for extracting features from a vibration signal can be utilized to analyze a sound signal. Alternatively, the time waveform or the spectrogram of the sound data can be passed as an input to a DL model to make inference about the equipment condition. The workflows shown in Figure 1.5 below summarize the above approaches.

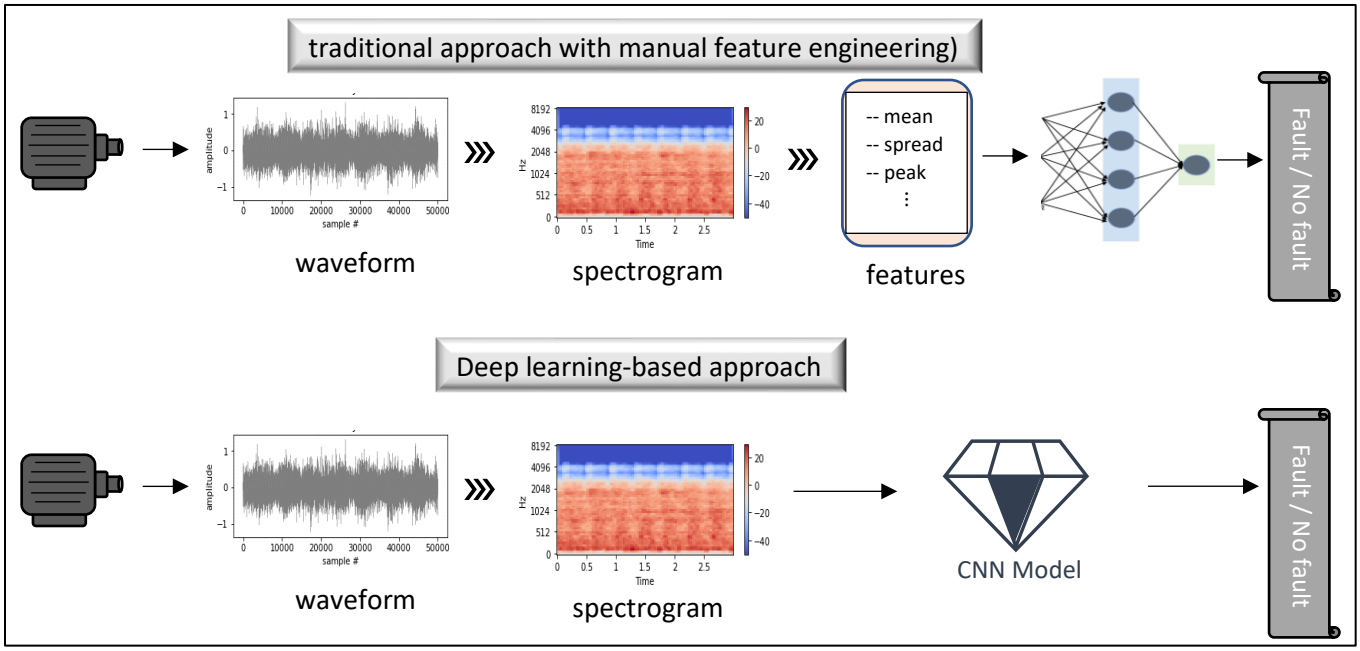


Figure 1.5: Approaches for equipment sound monitoring

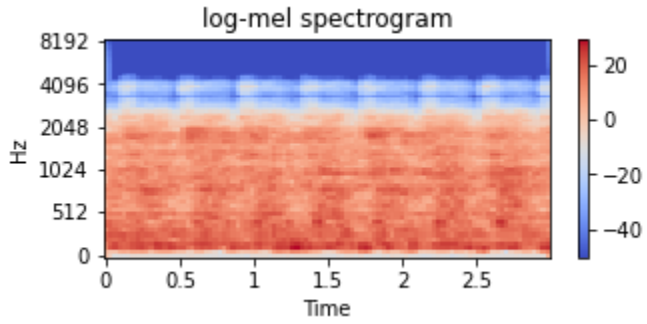
[3] Sound clips taken from https://www.iitk.ac.in/idea/datasets/. Verma et. al, Intelligent Condition Based Monitoring using Acoustic Signals for Air Compressors, *IEEE Transactions on Reliability*, 2016.

You may be surprised to see a CNN show up in the ASD workflow as well! Well, spectrograms can be treated as images and therefore analyzed using CNNs. We will work through one such application in Chapter 5.

Working with audio data in Python

Several Python libraries are available to handle audio files. Among these, *Librosa* is a popular library with several advanced modules to analyze acoustic signals. Using *Librosa*, you can analyze an audio signal in time domain, frequency domain, and time-frequency domain as shown below.

```
# load audio data
import numpy as np, matplotlib.pyplot as plt
import librosa

clipPath = "AirCompressor_Data/Healthy/preprocess_Reading1.dat"
data = np.loadtxt(clipPath, delimiter=',')

# log-mel spectrogram (popular for analyzing audio signals)
sr = 50000/3 # sampling rate: 50000 samples per 3 seconds
mel_spectrogram = librosa.feature.melspectrogram(y=data, sr=sr, n_mels=64)
log_mel_spectrogram = librosa.power_to_db(mel_spectrogram)

plt.figure()
img = librosa.display.specshow(log_mel_spectrogram, x_axis="time", y_axis="mel", sr=sr)
plt.colorbar(), plt.title('log-mel spectrogram')
```



Summary

This chapter impressed upon you the importance of computer vision and acoustic monitoring for plant health management. We familiarized ourselves with the typical workflows for analyzing visual and audio data obtained from a process plant, and looked at the available Python resources for handling such data. In the next chapter, we will look at the neural network architecture that powers the human-like machine intelligence for analyzing images and sound signals.

Chapter 2

Convolutional Neural Networks

CNN is the standard artificial neural network architecture for working with visual data such as images and videos. Gone are the days of explicit feature engineering with image data; CNNs are designed to automatically extract features and have, therefore, revolutionized the field of computer vision. Just like RNNs excel at handling data that exhibit temporal correlation, CNNs excel at extracting spatial correlations efficiently. CNNs had their first limelight moment with the introduction of LeNet architecture in the 1990s for recognition of handwritten numbers. Since then, several architectural innovations and deep learning advances have made CNNs super-powerful; these CNNs power modern self-driving cars, real-time face recognition, motion tracking, etc. Several books can be written on the topic of CNN architecture; however, the focus in the chapter is to provide an introductory treatment.

CNN architectures are built using several basic layers such as convolutional layers, pooling layers, and fully connected layers. We will study about these layers and the various associated hyper-parameters (such as filter strides, padding type, etc.). Specifically, the following topics are covered

- Introduction to CNN, convolutional layer, and pooling layer
- Structure of a typical CNN
- A simple CNN for recognition of handwritten numbers
- Evolution of CNN architecture

2.1 CNNs: An Introduction

CNNs (convolutional neural networks) are one of the most popular types of neural networks and are inspired by how a human brain processes visual objects. As shown in Figure 2.1, pixels in a small local region of an image are combined to generate a feature value in the first layer of the network. The features from the first layer are further combined to generate higher-level features. Upon training, the network learns to extract low-level features such as edges, textures in the early network layers which are combined progressively to form high-level features, providing the network a comprehensive understanding of the image. At the end of the network, a fully connected layer in combination with a sigmoid layer provides the output label for the image (in an image classification problem). CNNs get their name from the convolutional layers which form the backbone of these networks. However, as shown in Figure 2.1b, there are other types of layers as well in a CNN. Let's briefly familiarize ourselves with these commonly employed layers.

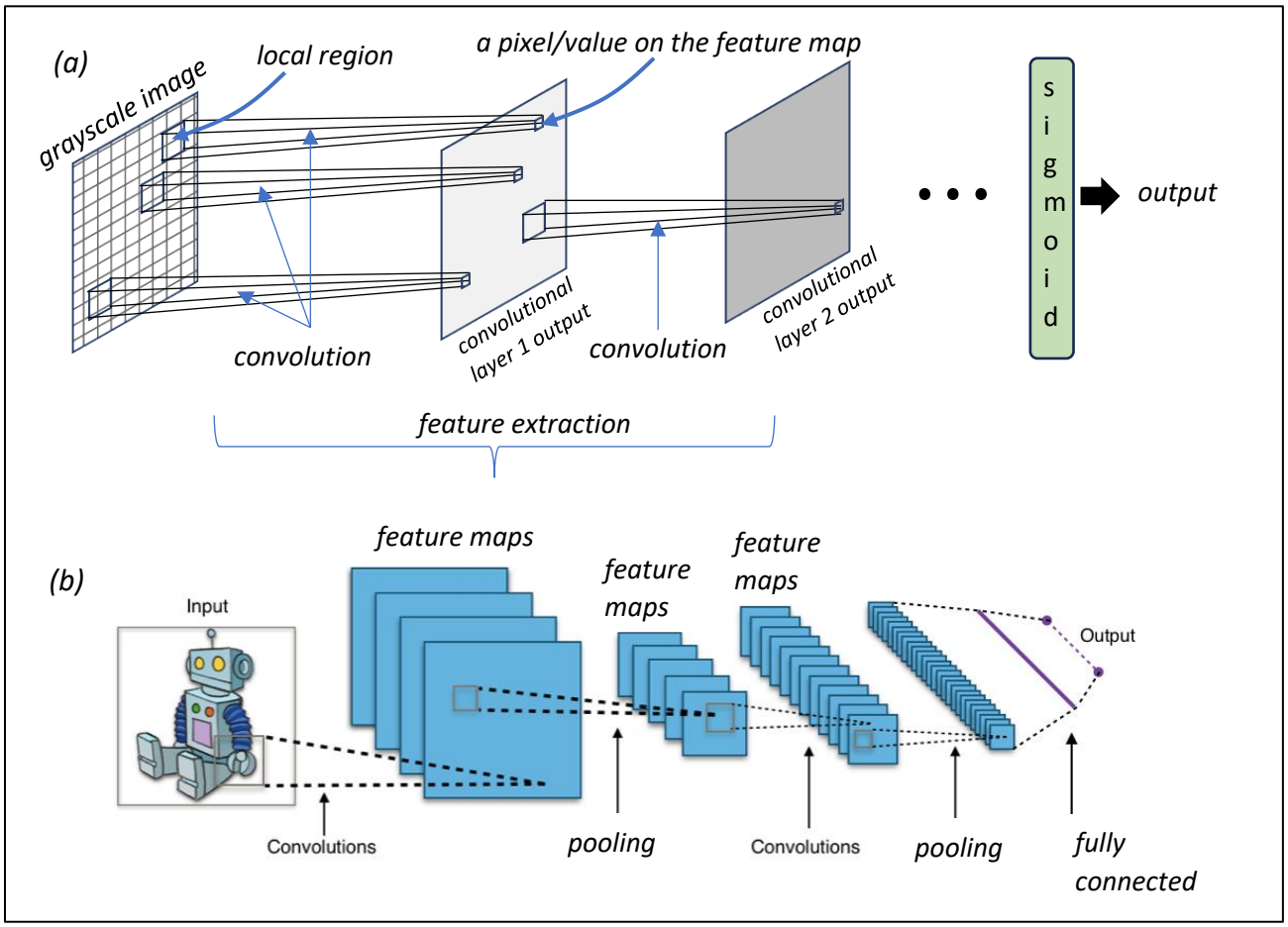


Figure 2.1: (a) Feature extraction in CNN (b) A simple CNN architecture[4]

[4] Diagram shard under the Creative Commons Attribution-Share Alike 4.0 International license by Aphex34 (https://commons.wikimedia.org/wiki/File:Typical_cnn.png)

Convolutional layer

In a convolutional layer, a local region from the incoming feature map (or input image) gets mapped to a value in the outgoing feature map via a convolutional filter.

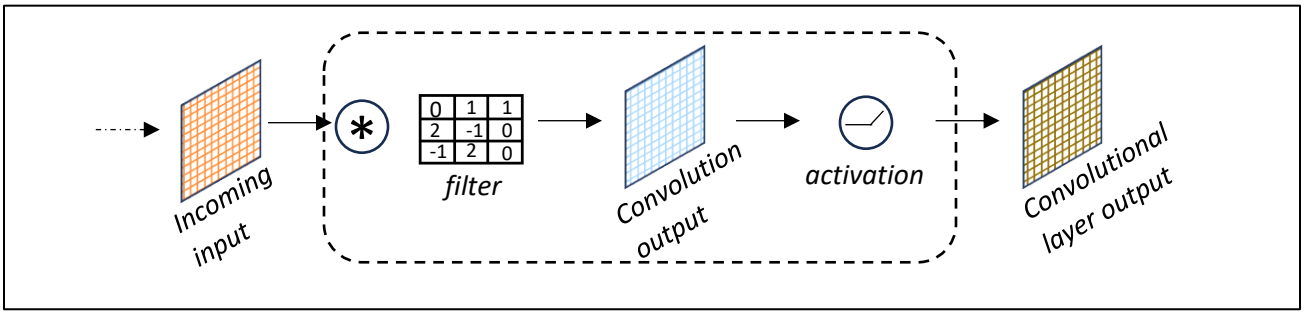


Figure 2.2: A convolutional layer with a single filter

In the above figure, the filter is a 2D matrix that slides over the incoming feature map in horizontal and vertical directions as shown below

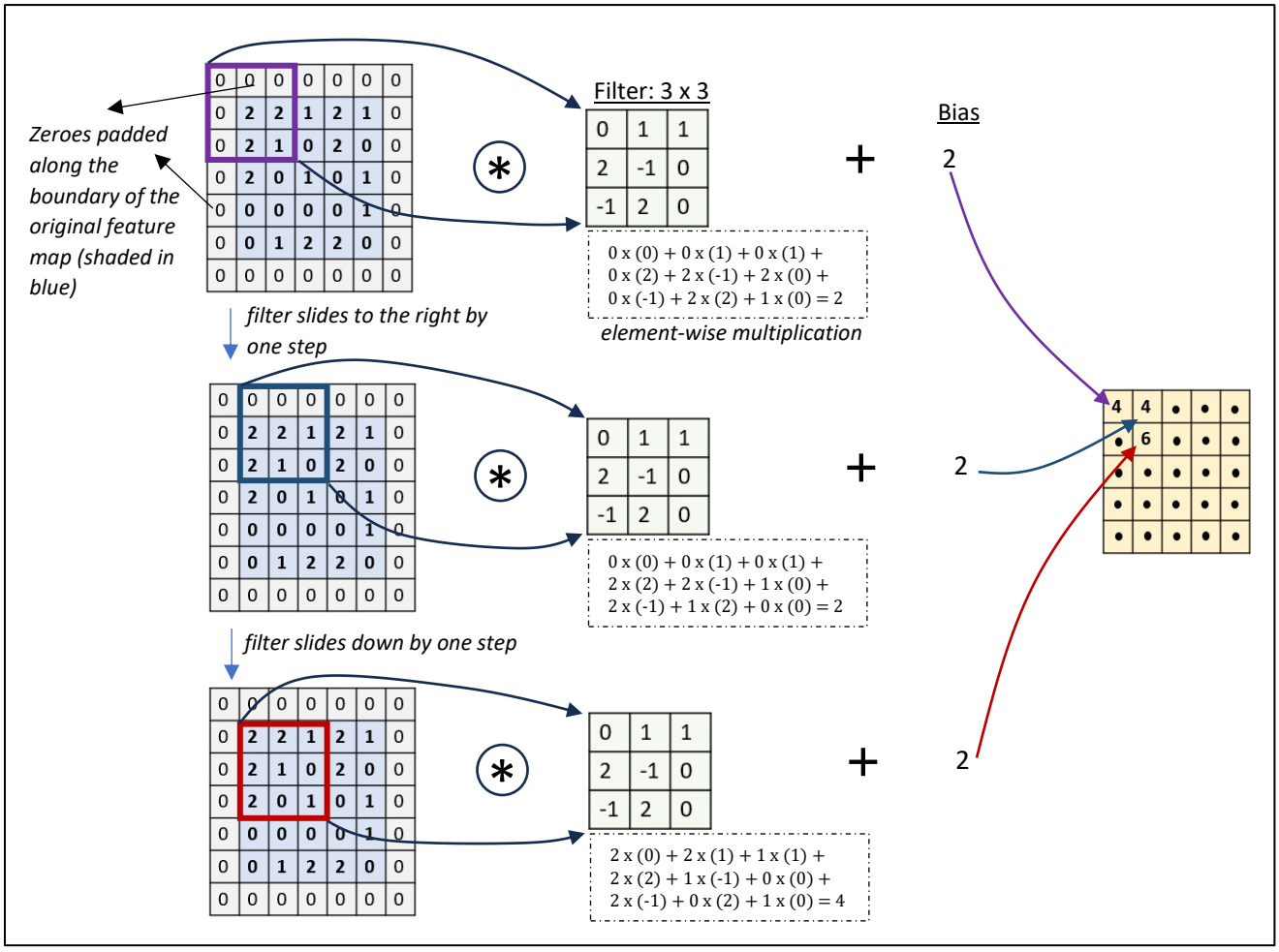


Figure 2.3: Convolution operation (concept behind *padding* is covered later)

The entire feature map is scanned using the same filter. The output of the above convolutional operation is a feature map. In the above illustration, the convolutional operation employed 10 trainable parameters (9 filter weights and 1 bias). In practice, one filter is not sufficient to extract all the features from the incoming feature map and therefore, multiple filters are employed to generate multiple output feature maps as shown below.

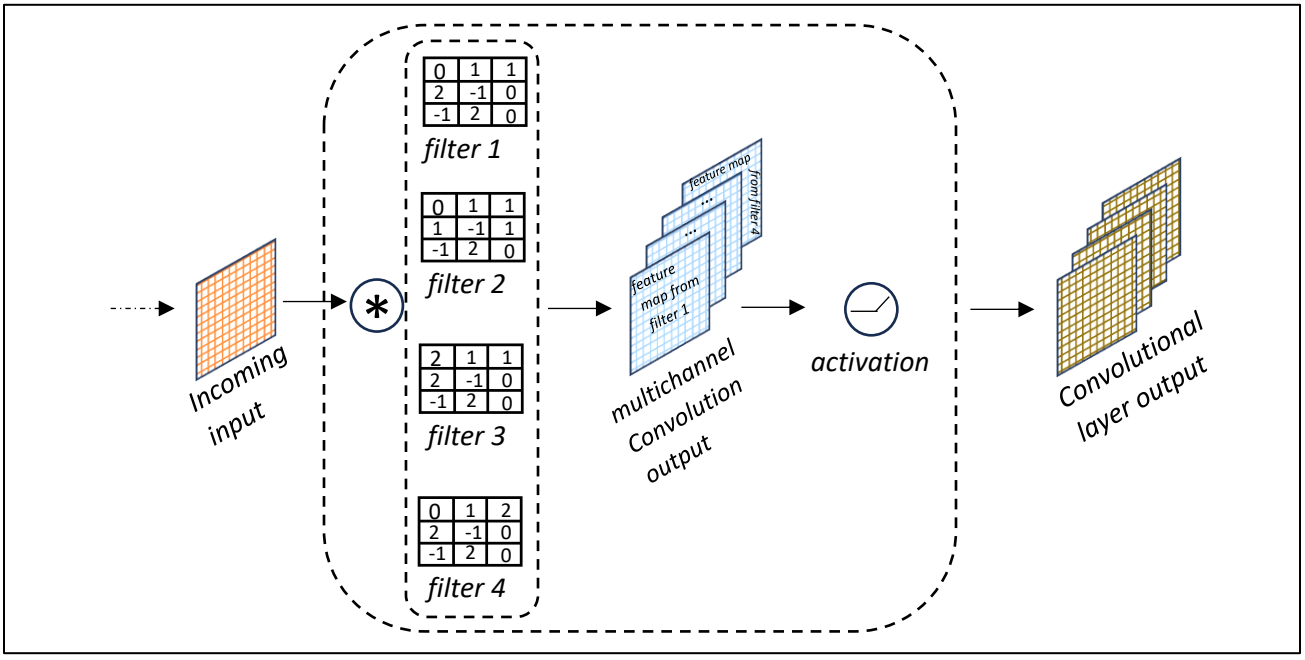


Figure 2.4: A convolutional layer with multiple filters

Each of the filters in the above illustration has its own set of weights and bias. Therefore, for a kernel size 3 X 3, the total number of parameters becomes 40. Now consider a convolutional layer handling a color image (which, as we know, has 3 channels) or the multichannel convolutional output from the above illustration. How does the convolutional operation work now? Well, in this case, the filter becomes a 3D matrix with the same number of channels as that in the incoming input. As before, each value in the output feature map involves element-wise multiplications between the filter and the corresponding (3D) local region of the incoming input.

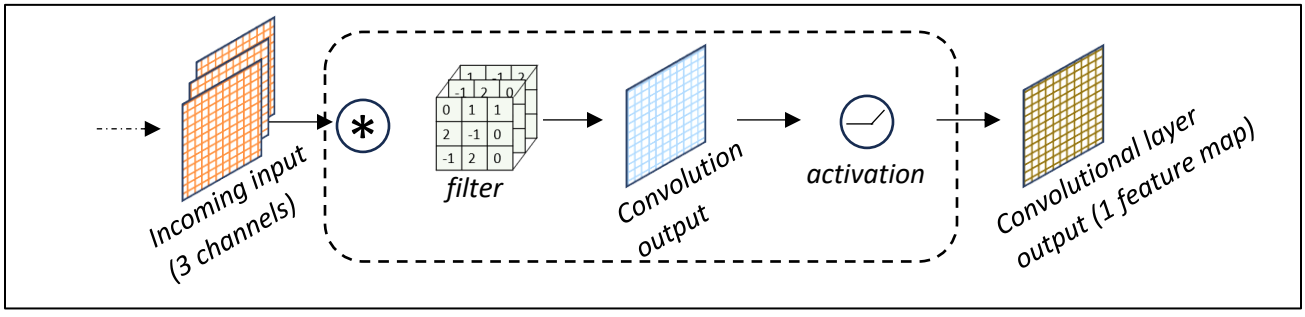


Figure 2.5: A convolutional layer with a single multichannel filter

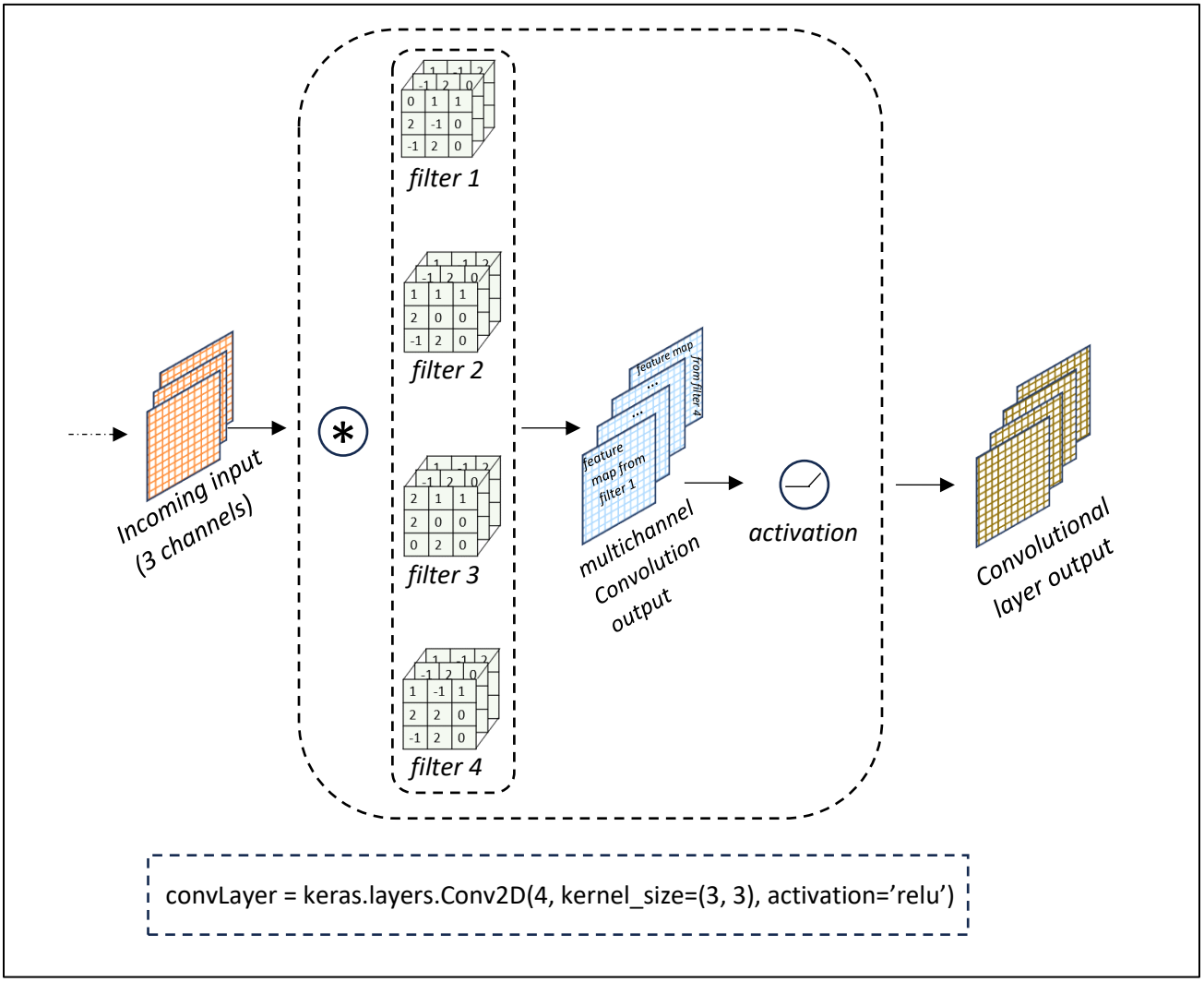


Figure 2.6: A convolutional layer with 4 filters handling 3-channeled input

It is common to choose filters of size 5X5 or 3X3. Note that the number of channels of a filter is the same as the number of incoming feature maps. Also, the number of filters used is generally increased as one goes deeper into the network (such as 64 filters in the first layer, then 128, then 256, and so on).

Strides

In Figure 2.3, we saw that the filter moved one step at a time while scanning the input image. However, this is not mandatory. A filter can move multiple steps in the horizontal and vertical directions during scanning and the number of steps moved is called stride. The images below show the convolutional operation with a stride of two.

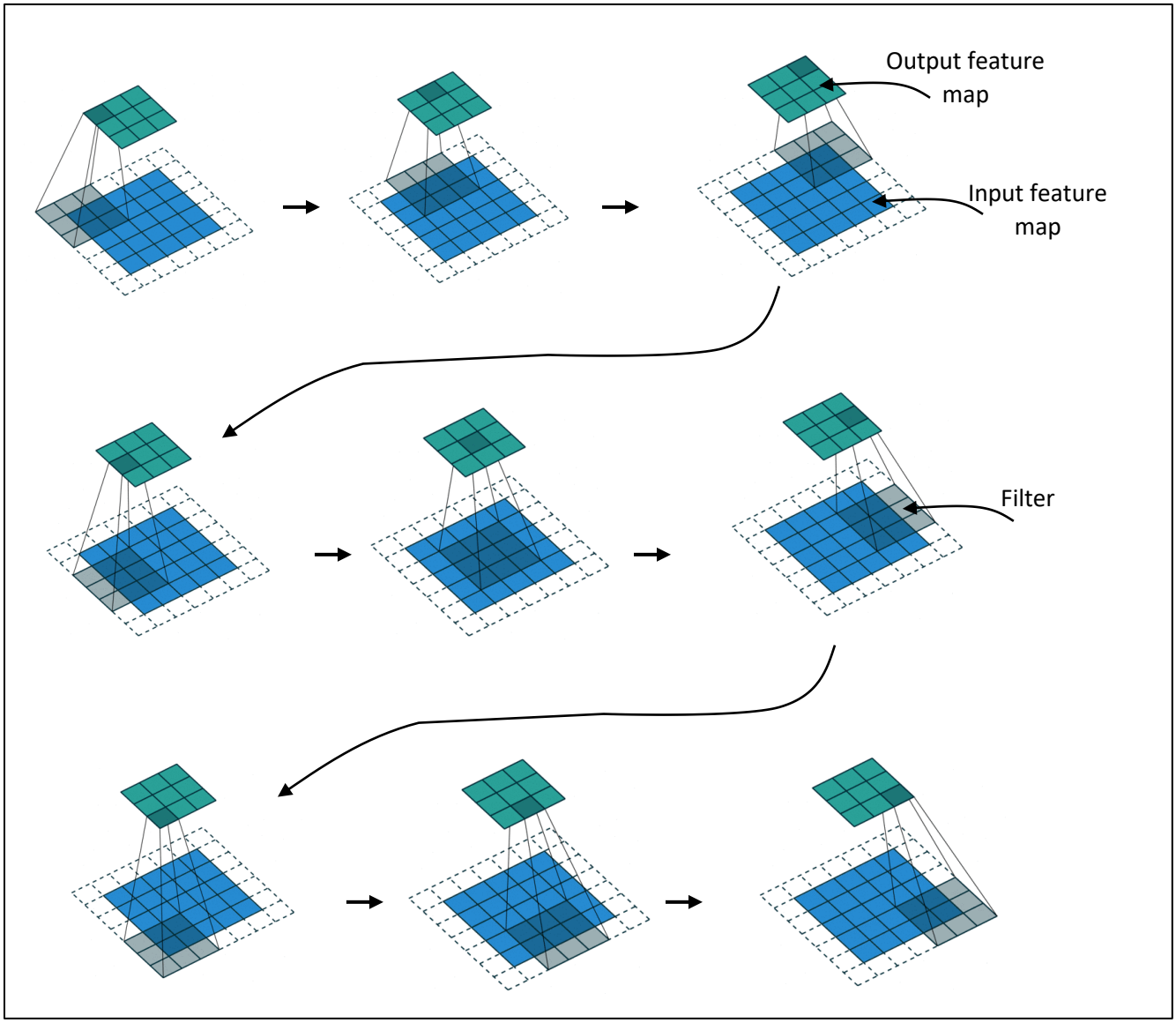


Figure 2.7: Convolution with stride of 2 and zero padding[5]

Padding

In the previous illustration, we used zero padding to handle boundary pixels such that with stride of 1, the width and height of the feature maps remain the same as those of the input feature maps. This is also called the 'SAME' padding configuration. An alternative is 'VALID' configuration wherein no padding is done and therefore resulting in a smaller output feature map. An illustration is shown below.

[5] Images (Copyright © Vincent Dumoulin, Francesco Visin) obtained from https://commons.wikimedia.org/wiki/File:Convolution_arithmetic_-_Padding_strides.gif shared under MIT license (https://opensource.org/license/mit)

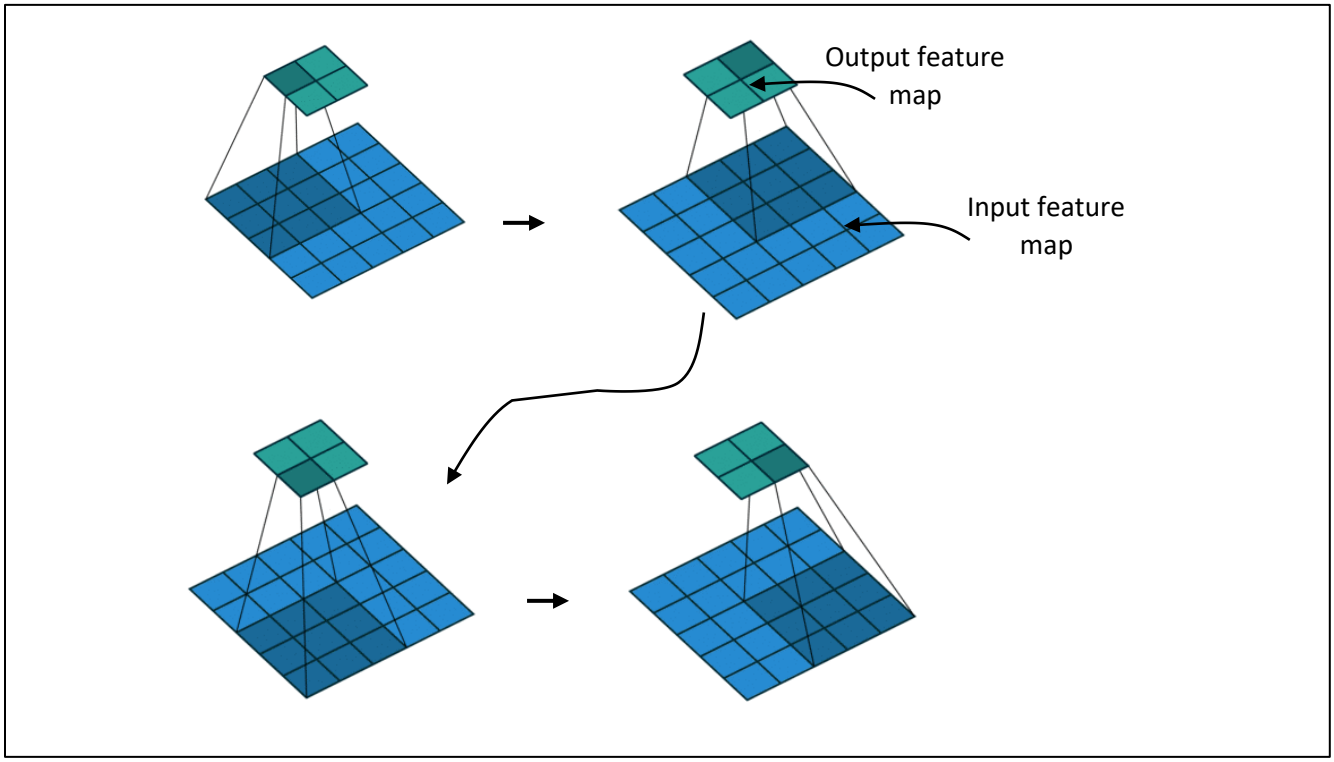
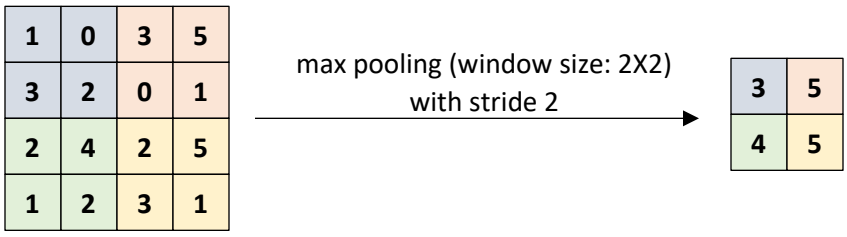


Figure 2.8: Convolution with stride of 1 and no padding[6]

Pooling layer

While a convolutional layer extracts features from incoming feature maps, a pooling layer shrinks the size of the feature maps. It is typical to put a pooling layer after a convolutional layer. Just like convolution, the pooling operation works on a small region of a feature map at a time. However, there are no trainable parameters in a pooling layer. All that happens is that the average or maximum (more commonly used) value from the local region is extracted as illustrated below for max pooling




```
maxPoolingLayer = keras.layers.MaxPooling2D(pool_size=(2, 2), strides=2, padding="valid")
```

[6] Images (Copyright © Vincent Dumoulin, Francesco Visin) obtained from https://commons.wikimedia.org/wiki/File:Convolution_arithmetic_-_No_padding_strides.gif shared under MIT license (https://opensource.org/license/mit)

The pooling operation takes place on each incoming channel independently and therefore, the number of output channels equals the number of input channels.

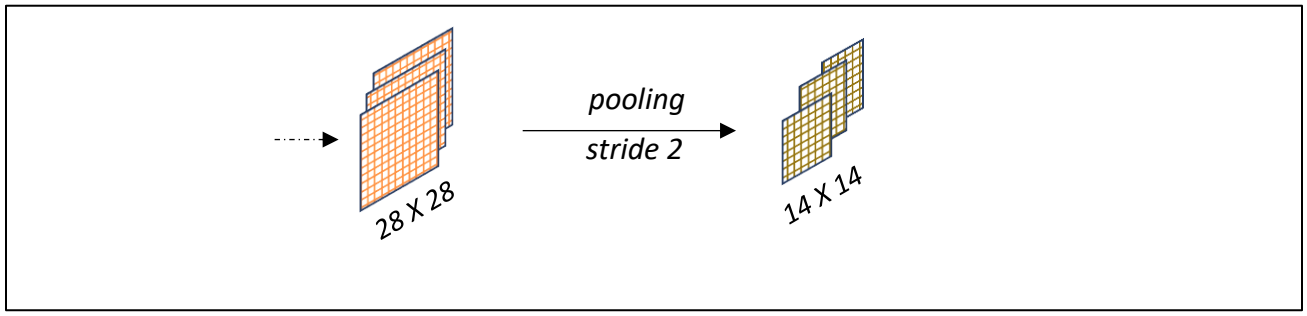
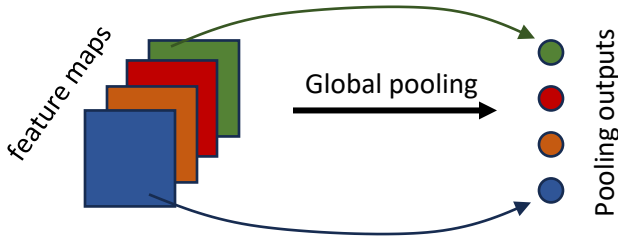


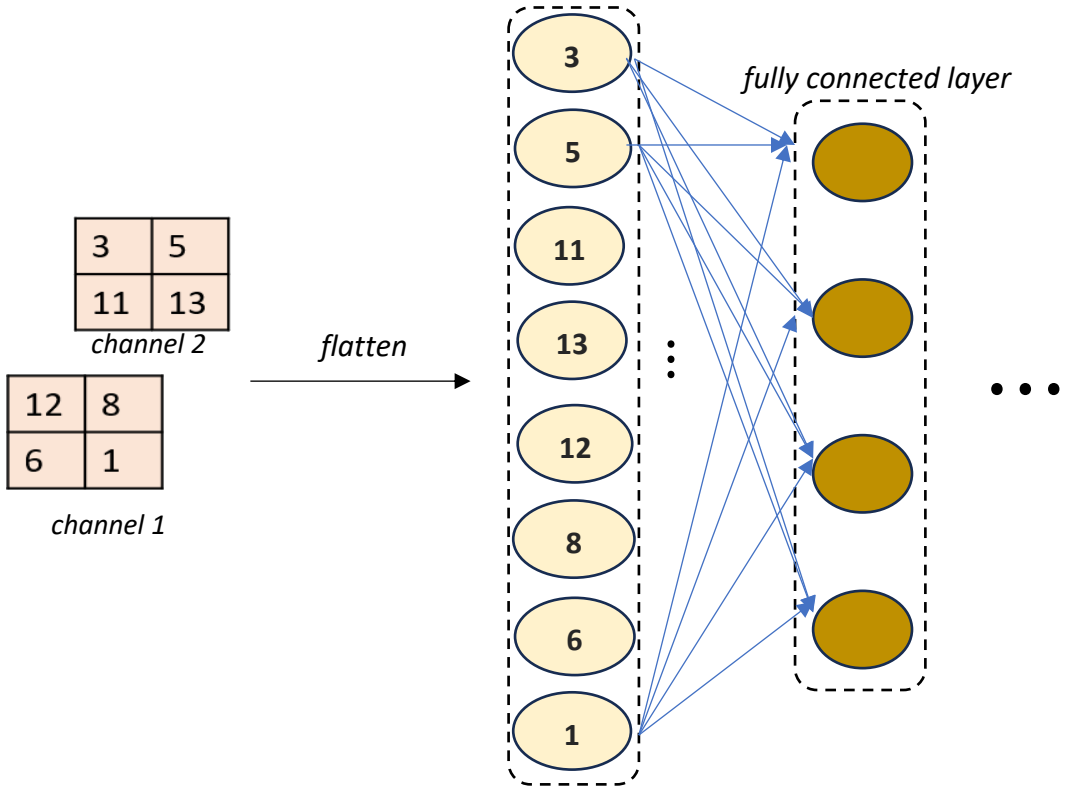
Figure 2.9: Pooling operation on multi-channel input

Global pooling

If desired, an incoming feature map can be shrunk down to a single value via global pooling. Keras provides classes GlobalMaxPooling2D and GlobalAveragePooling2D for this.

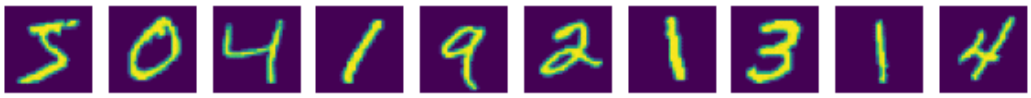


We alluded to before that the classification output is generated via fully connected layers. However, how do we convert the 3D feature maps to the neurons in a fully connected layer? This is where the *flattening* operation is used, wherein the feature maps are unfolded as shown below



2.2 A Simple CNN for Handwritten Digit Recognition

In the previous section, we saw how easy it is to define a convolutional or pooling layer using Keras. It is equally easy to stack together several of these layers to create a CNN and train the network. In this section, we will create a simple CNN to classify images of handwritten digits (0 to 9) of the kind shown below

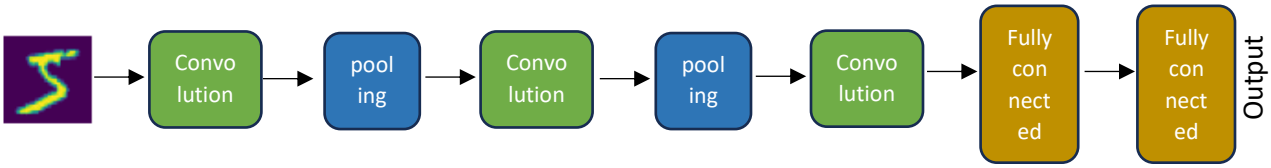


This is a popular dataset (called MNIST dataset) in the CV world and is conveniently bundled within Keras. Each image is a grayscale image of size 28 X 28 pixels. Let's load the dataset in our workspace and pre-process the images to prepare them for model training.

```
# import packages
import numpy as np, matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
```

```
# load data
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data() # 70000 images in dataset
print('training input data shape: ', x_train.shape), print('test input data shape: ', x_test.shape)
print('training output data shape: ', y_train.shape), print('test output data shape: ', y_test.shape)

>>> training input data shape:  (60000, 28, 28)
   test input data shape:  (10000, 28, 28)
   training output data shape:  (60000,)
   test output data shape:  (10000,)

# reshape each image's 2D data into a 3D shape with 1 channel (i.e., of shape 28 X 28 X 1)
x_train = x_train.reshape((x_train.shape[0], 28, 28, 1))
x_test = x_test.reshape((x_test.shape[0], 28, 28, 1))

# scale pixel values to the [0, 1] range
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
```

With the training and test dataset prepared, we are ready to create our CNN model. We will implement the LeNet architecture as shown below.




```
# define CNN
lenet5 = keras.Sequential([
   keras.layers.Conv2D(6, (5,5), padding='same', input_shape=[28, 28, 1], activation='tanh'),
   keras.layers.AveragePooling2D((2,2)),

   keras.layers.Conv2D(16, (5,5), padding='valid', activation='tanh'),
   keras.layers.AveragePooling2D((2,2)),

   keras.layers.Conv2D(120, (5,5), padding='valid', activation='tanh'),
   keras.layers.Flatten(),
   keras.layers.Dense(84, activation='tanh'),
   keras.layers.Dense(10, activation='softmax') # original LeNet model had used RBF activation
])
lenet5.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
lenet5.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 6)         156

 average_pooling2d (AverageP  (None, 14, 14, 6)        0
 ooling2D)

 conv2d_1 (Conv2D)           (None, 10, 10, 16)        2416

 average_pooling2d_1 (Averag  (None, 5, 5, 16)         0
 ePooling2D)

 conv2d_2 (Conv2D)           (None, 1, 1, 120)         48120

 flatten (Flatten)           (None, 120)               0

 dense (Dense)               (None, 84)                10164

 dense_1 (Dense)             (None, 10)                850

=================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
_________________________________________________________________
```

The above few lines of code is all that it takes to create a CNN. We assume that you have worked with ANNs in the previous books of the series and therefore, we assume your familiarity with the terms like *optimizer*, *loss*, etc. Model summary shows the number of model parameters in each layer of the network. Let's do a quick calculation to see how we ended up with 156 parameters in the first convolutional layer: we have 6 filters each of size 5 X 5 ⇒ each filter has 25 weights and 1 bias ⇒ 26 parameters; therefore, all the filters in total have $26 \times 6 = 156$ parameters. Let's train our model.

```
# fit model
history = lenet5.fit(x_train, y_train, batch_size=128, epochs=15, validation_split=0.1)
```

```
Epoch 1/15
422/422 [==============================] - 6s 13ms/step - loss: 0.3659 - accuracy: 0.8905 - val_loss: 0.1438 - val_accuracy: 0.9552
Epoch 2/15
422/422 [==============================] - 5s 12ms/step - loss: 0.1352 - accuracy: 0.9596 - val_loss: 0.0940 - val_accuracy: 0.9732
                                                    ⋮
Epoch 14/15
422/422 [==============================] - 6s 15ms/step - loss: 0.0111 - accuracy: 0.9965 - val_loss: 0.0540 - val_accuracy: 0.9870
Epoch 15/15
422/422 [==============================] - 6s 15ms/step - loss: 0.0105 - accuracy: 0.9967 - val_loss: 0.0482 - val_accuracy: 0.9870
```

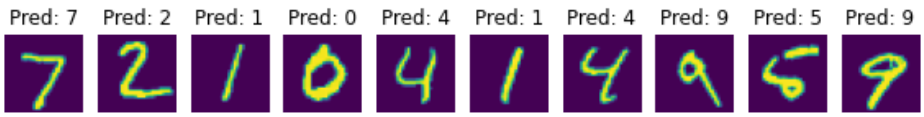
Our model seems to be doing a good job at classifying the validation images. Let's make predictions for the test images.

```
# evaluate model
lenet5.evaluate(x_test, y_test)
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.0460 - accuracy: 0.9869
```

```
fig, ax = plt.subplots(1, 10, figsize=(10,2))
for i in range(10): # first 10 test images
   img = x_test[i]
```

```
    softmax_probabilities = lenet5.predict(np.expand_dims(img, 0)) # batch dimension added first
    label_pred = np.argmax(softmax_probabilities)

    ax[i].imshow(img)
    ax[i].set_title(f'Pred: {label_pred}'), ax[i].axis('off')
plt.show()
```



Congratulations on your first successful convolutional neural network! Of course, production-ready CNNs with more complex input images have several more bells and whistles, but the above steps are very typical of what goes behind creating modern CNNs.

2.3 Evolution of CNNs

LeNet was one of the first successful application of CNN for image classification. Several variants of LeNet's architecture have since been developed by CV community; these variants include, amongst others, AlexNet, VGGNet, ResNet, Inception, GoogLeNet, and MobileNet. Among these, AlexNet was the first deep CNN that drew everyone's attention by winning the 2012 ILSVRC (ImageNet Large Scale Visual Recognition Challenge) competition. The input images were large (227 X 227 pixels) and there were 1000 output classes. Its architecture is shown below.

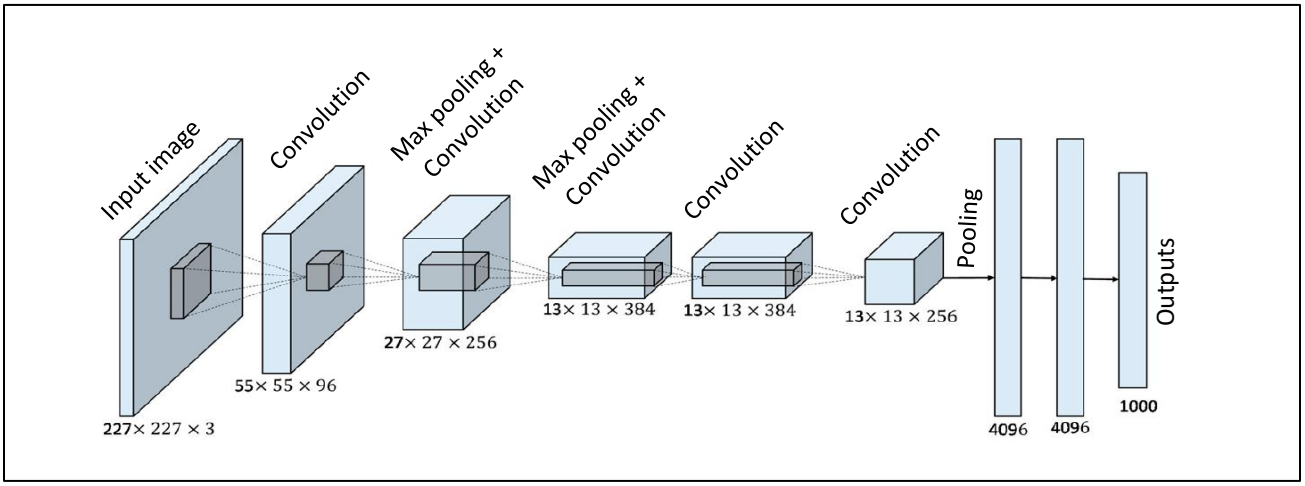


Figure 2.10: The AlexNet architecture[7]

[7] Adapted from original image shared in Han et. al (Pre-Trained AlexNet Architecture with Pyramid Pooling and Supervision for High Spatial Resolution Remote Sensing Image Scene Classification, *Remote Sensing*, 2017), under Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/)

AlexNet uses 60 million parameters! It is computationally challenging to train; however, the authors of AlexNet brought together several deep learning innovations (such as dropout regularization, data augmentation, ReLU activation function, and GPU usage) to ensure efficient model training. AlexNet was followed by another pioneer CNN architecture called VGGNet that won the ILSVRC competition in 2014. Its architecture is shown below: it had 16 layers and around 138 million parameters!

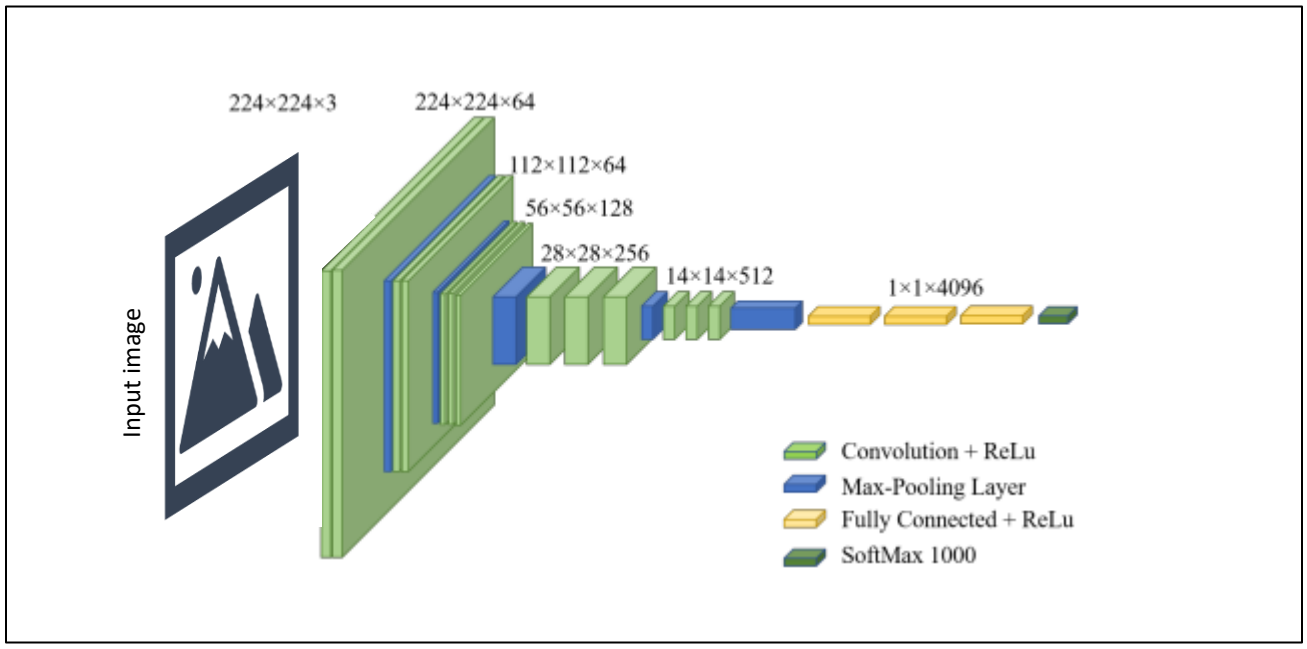


Figure 2.11: The VGG-16 network architecture[8]

AlexNet and VGG inspired pursuit of deeper networks. However, very deep networks brought the model training convergence issues to the fore. The winner of the 2015 ILSVRC competition, the ResNet[9], introduced an ingenious trick[10] to enable efficient training of very deep networks employing hundreds to thousands of layers. Consequently, ResNet is one of the most popular CNN architectures.

The networks that we discussed in this section are classification networks. Networks used for object detection and image segmentation have witnessed remarkable architectural innovations in recent times. Some popular architectures in these categories include YOLO, RetinaNet, Mask R-CNN, U-Net, etc.

[8] Adapted from original image shared in De et. al (Monolithic-3D Inference Engine with IGZO Based Ferroelectric Thin Film Transistor Synapses, *TechRxiv*, 2022), under Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/)

[9] https://arxiv.org/pdf/1512.03385.pdf

[10] The trick was usage of 'skip connections' wherein the output of a layer bypasses one or more layers and is fed as input to another layer

Summary

This chapter covered the basics of convolutional neural networks and introduced you to the different component layers of a CNN. You became familiar with how to define a CNN and trained a simple CNN to classify handwritten digits. It was assumed that you trained the CNN on your PC. However, for more complex tasks and bigger CNN models, your PC may not be able to handle the CNN training workload. Therefore, the next chapter will introduce you to a popular (and free) online resource for CNN model development.

Chapter 3

CNN Training Environment

Deep learning models are computationally intensive; training very deep computer vision models on a large dataset can take several days (even with GPUs). It is very likely that training such models on your personal computer is infeasible. How do you practice and experiment with deep learning-based CV then? To our fortune, some online platforms such as Google Colab and Kaggle exist that provide free compute resources! You can create Python notebooks and execute them on these platforms. Expectedly, there are certain limitations, but the free resources usually suffice for learning purposes.

If you have not used Google Colab before, then this chapter will help you get familiar with the platform. If you are already familiar, then you may move to Chapter 4. We will cover the following topics related to Google Colab

- Creating and executing notebooks
- Accessing GPU resources
- Saving notebooks and trained models

3.1 Introduction to Google Colab

Google Colab is an online Jupyter notebook environment that provide free access to RAM and GPU resources for education purposes. To access Colab, enter the URL *https://colab.research.google.com* in your browser; you should see the interface shown in Figure 3.1.

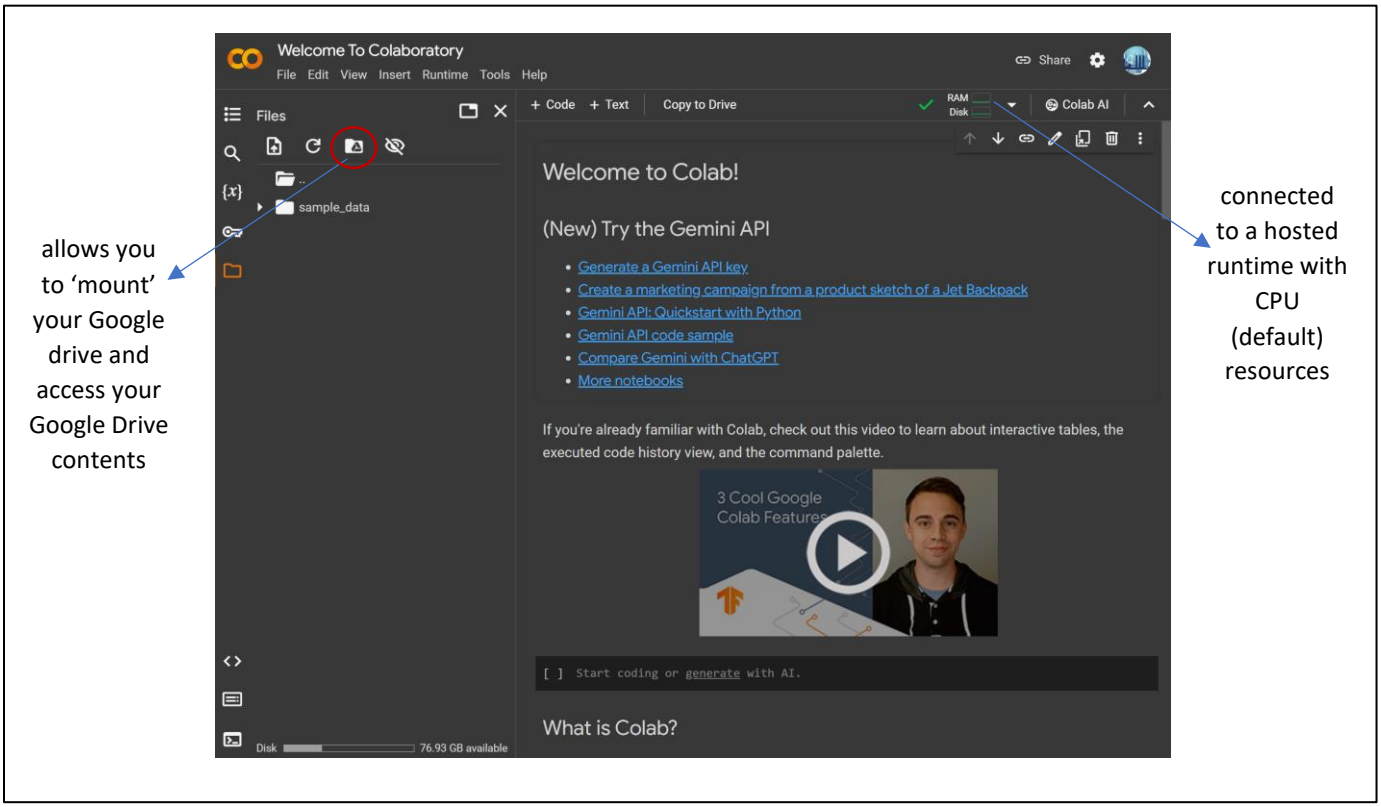


Figure 3.1: Google Colab Jupyter Notebook interface

Let's try to train our MNIST digit classification model here. You may copy-paste the code from Chapter 2 in a new notebook (from File menu → New notebook) or open the Jupyter notebook *mnist_CNN_classification.ipynb* from Chapter 2 in Colab (from File menu → Open notebook → Upload). We follow the later approach here.

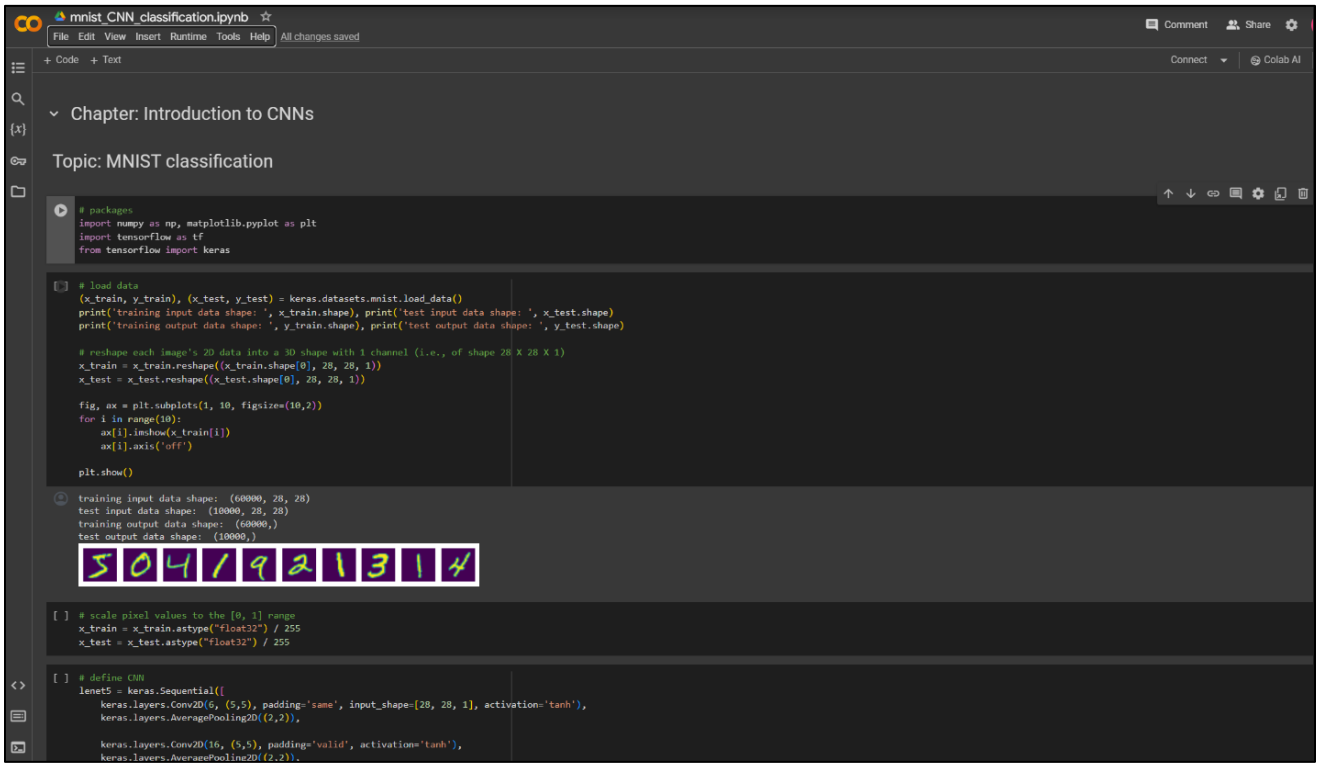


Figure 3.2: The notebook file from Chapter 2 in Colab

Before you execute the cells, you may want to change the hardware accelerator setting via the Edit menu as shown below

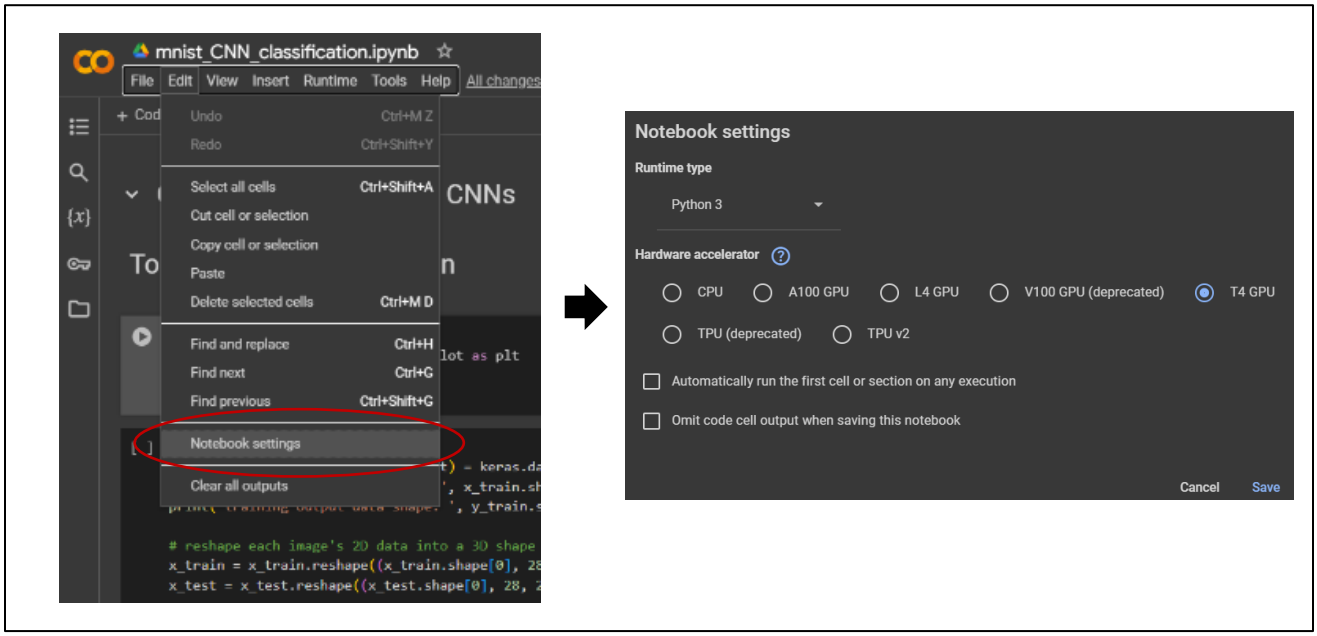


Figure 3.3: GPU setting selection in Colab

You may now click on the ‘Connect’ button (on the top-right corner) and run each cell one at a time or run the whole notebook via Runtime menu → Run all. You should see the same results as those seen in Chapter 2.

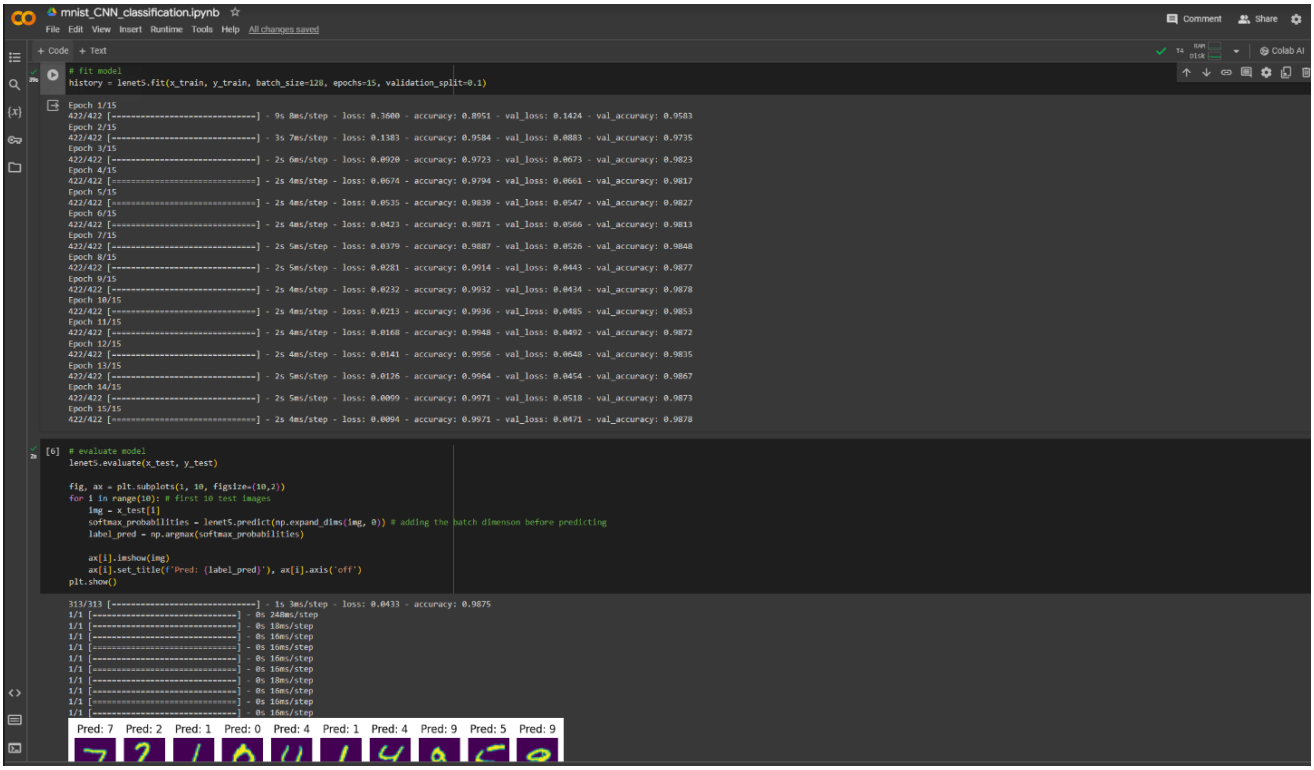


Figure 3.4: MNIST digit classification model training in Colab using GPU

Saving notebook and model

You can save your notebook as a *.ipynb* file in Google Drive or GitHub via the *File* menu. Once you have trained your model, you will want to save it to be able to use it for predictions later on. The following code will save your model in your Google Drive.

```
# save model
# lenet5.save('MNIST_model.h5')
lenet5.save('/content/drive/MyDrive/MNIST_model.h5')
```

Once you have the model in your Google Drive, you can download it to your PC and do image evaluations on your PC. Your model is already trained and therefore, Colab's compute resources are no longer required. Note that before you are able to save your model to Google Drive, you must 'mount your drive' using the button shown in Figure 3.1. To load the saved model in your script, you can use the *load_model* function.

Summary

This chapter provided a quick familiarization with the Google Colab platform. We learnt how to access free GPU resources and execute Notebook files on Colab. In the following chapters, we will utilize the Colab platform to develop automated visual product inspection and equipment acoustic monitoring solutions.

Chapter 4

Automated Product Quality Inspection via Computer Vision

The field of computer vision has witnessed a flurry of research and innovations in the last decade. Nonetheless, the subfield of image classification is more or less a mature field. Object detection and image segmentation models use the image classification models as their backbone. Therefore, in this chapter, we will focus on image classification and make ourselves familiar with the concept of transfer learning. Specifically, we will look at a steel-strip image dataset with the objective to classify product images into different faulty classes. While building a computer vision solution, you may encounter several challenges: your dataset may be too large to fit in your PC's memory or you may not have enough images to overcome model overfitting. Several solutions with varying degrees of complexity have been devised for these commonly encountered obstacles. We will touch upon some of these solutions.

Computer vision is one of the marvels of modern computer science and is, inarguably, a complex task. However, the ML community has come together and provided several useful resources to make computer vision more accessible: these include open-source datasets, open-source pre-trained state-of-the-art models, user-friendly modeling frameworks, etc. This implies that a model that has been trained by somebody else using large computational resources on millions of images can be used by you with some tweaks to solve your specific problem (and that too on your laptop potentially). Sounds interesting, doesn't it? Let's jump right into it and cover the following topics

- Creating a CNN model from scratch for steel product defect classification
- Introduction to transfer learning and fine-tuning CNN models
- Steel product defect classification via transfer learning

4.1 NEU Steel Defect Dataset

To demonstrate how to build a CNN network for quality inspection, we will use the NEU (Northeastern University) surface defect dataset[11]. The dataset consists of images of steep strip surfaces exhibiting six types of defects (Figure 4.1). A total of 1800 images (200 X 200 pixels) are provided (300 images belonging to each class of defect).

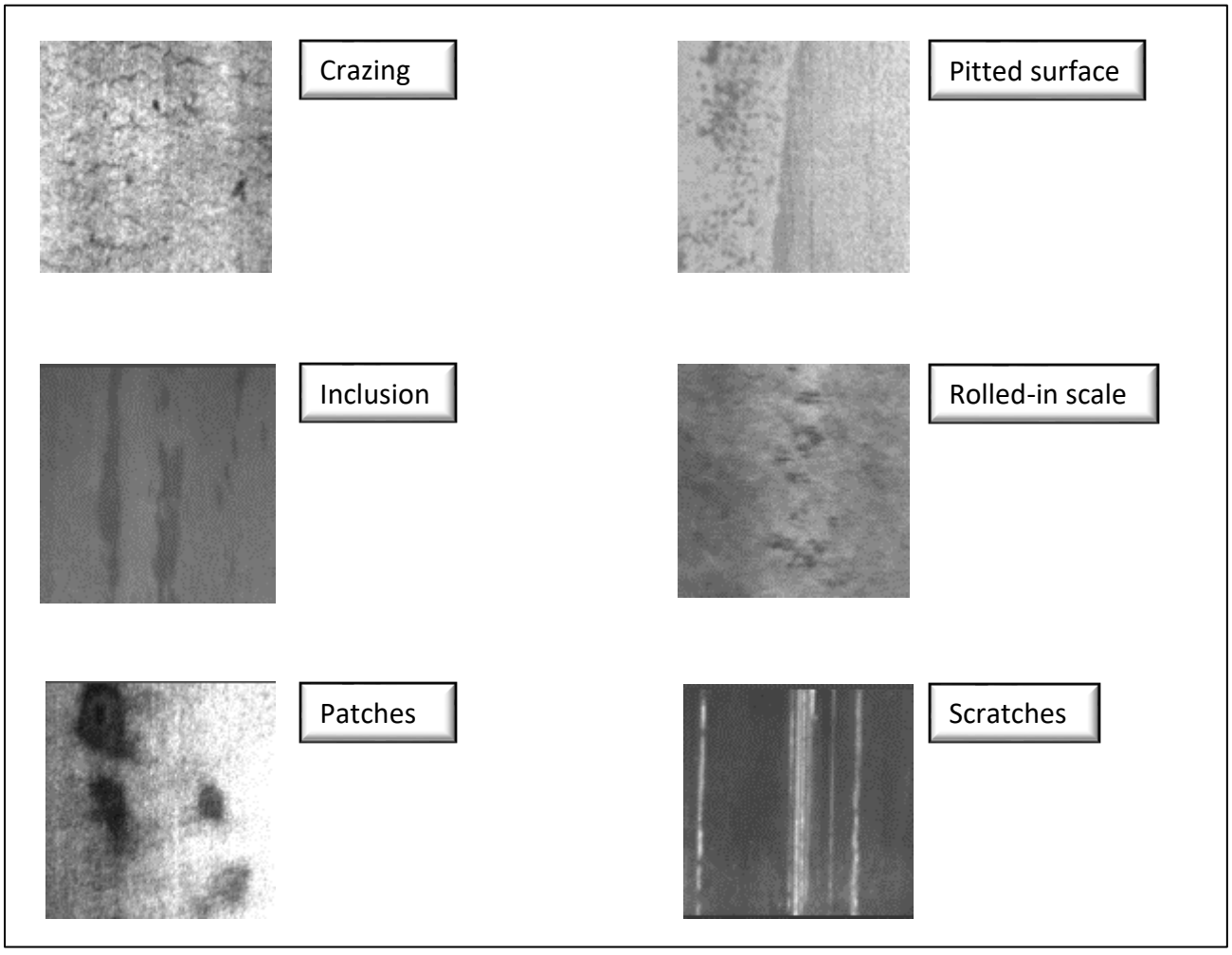


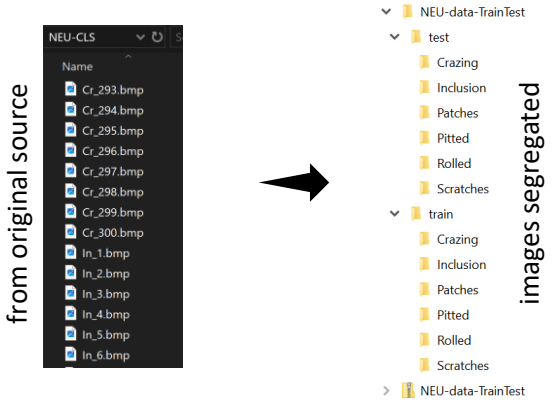
Figure 4.1: Defect classes in NEU surface defect dataset

Steel strip is among the main products of iron and steel industry and the quality (strength, resistance to corrosion, etc.) of the steel strip is negatively impacted by the presence of the aforementioned defects. Therefore, it is desired to have an inspection system that can promptly detect the presence of a defective steep strip and notify the plant personnel.

[11] Available at http://faculty.neu.edu.cn/songkechen/zh_cn/zhym/263269/list/index.htm

4.2 Steel Defect Classification Modeling from Scratch

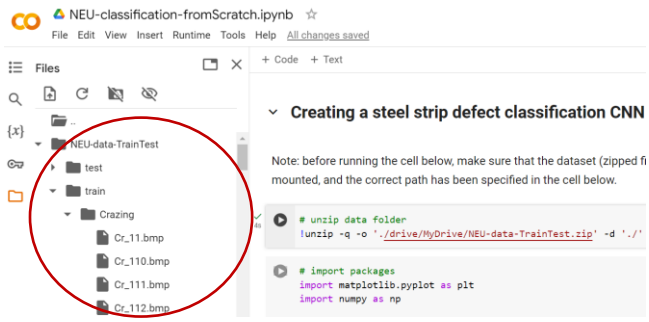
To train a CNN-based steel strip defect classification model, we will utilize Google Colab. The complete notebook is provided on GitHub as *NEU-classification-fromScratch.ipynb*. The 1800 images have been divided into a training dataset (1728 images) and a test dataset (72 images; 12 images from each class). Also, as is usually done for CV classification tasks, the images have been segregated into separate folders corresponding to the six defects. The folder hierarchy[12] looks as follows



The dataset is uploaded to Google Drive as a zipped file and then unzipped in Colab using the following code. Before running the code below, make sure that your Google Drive has been mounted (mounting procedure was illustrated in Chapter 3).

```
# unzip
!unzip -q -o './drive/MyDrive/NEU-data-TrainTest.zip' -d './'
```

Upon executing the above command in a Colab cell, you should see your images as follows



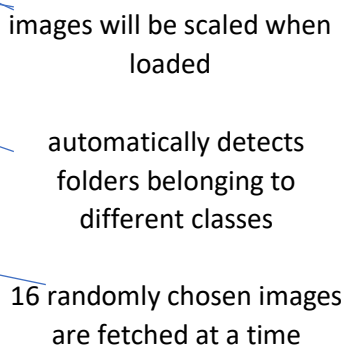
Let's now import some packages and 'load' the images.

[12] NEU surface defect dataset is also available on *Kaggle* (https://www.kaggle.com/datasets/fantacher/neu-metal-surface-defects-data/data) in the segregated form. If using this dataset, then place the validation images in the training dataset to be consistent with the illustration in this chapter.

```
# import required packages
import matplotlib.pyplot as plt, numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import layers, models
```

You are probably not familiar with the ImageDataGenerator class. This class allows just-in-time loading of images and avoids loading all the images into memory at once. In Chapter 2, while building MNIST classification model, we had loaded data from all the images in our workspace. However, this may not be practical when you are dealing with millions of images. Several techniques have been devised by the CV community to handle large datasets; the ImageDataGenerator class is a very convenient option[13] for just-in-time loading (also termed lazy loading) of images. This class also supports several just-in-time image pre-processing such as scaling, resizing, data augmentation, etc.

```
# create iterators to progressively load images later in the code
train_datagen = ImageDataGenerator(rescale = 1/255.0, validation_split = 0.2)
test_datagen = ImageDataGenerator(rescale = 1/255.0)

fit_iterator = train_datagen.flow_from_directory(
   directory = './NEU-data-TrainTest/train',
   target_size = (200, 200),
   batch_size = 16,
   class_mode = 'categorical',
   subset='training')

valid_iterator = train_datagen.flow_from_directory(
   directory = './NEU-data-TrainTest/train',
   target_size = (200, 200),
   batch_size = 16,
   class_mode = 'categorical',
   subset='validation')

test_iterator = test_datagen.flow_from_directory(
   directory = './NEU-data-TrainTest/test',
   target_size = (200, 200),
   batch_size = 16,
   class_mode = 'categorical', shuffle= False)
```




```
>>> Found 1386 images belonging to 6 classes.
    Found 342 images belonging to 6 classes.
    Found 72 images belonging to 6 classes.
```

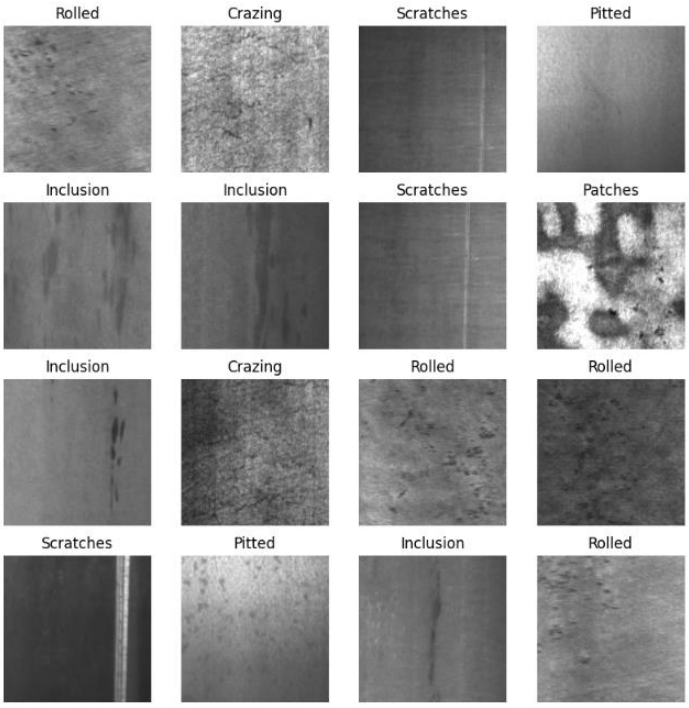
[13] Other popular options for creating image input pipelines include *tf.keras.utils.image_dataset_from_directory* and *tf.data.Dataset*

```
# check class names
class_names = fit_iterator.class_indices
class_names = list(class_names.keys())
print(class_names)

>>> ['Crazing', 'Inclusion', 'Patches', 'Pitted', 'Rolled', 'Scratches']
```

The code below shows how *fit_iterator* can be used to load images from the fitting dataset[14].

```
# let's load a few images from the fitting dataset via the iterator
images, labels = fit_iterator.next() # label for each image is in one-hot encoded form

fig, axes = plt.subplots(nrows=4, ncols=4)
for i in range(16):
    image, label = images[i], labels[i]
    label_name = class_names[np.argmax(label)]
    row, col = i//4, i%4
    axes[row][col].imshow(image)
    axes[row][col].set_title(label_name)
    axes[row][col].axis('off')
plt.show()
```

In the above code, the *next* method of the iterator returns a tuple of two numpy arrays: an array of pixel values and an array of (one-hot encoded) labels. Execution of the above code results in the following



[14] Note that tf.keras.preprocessing.image.ImageDataGenerator has been deprecated. However, it is easy to understand for a beginner and therefore used in this case-study to illustrate the concept of lazy loading of images.

Let's now define our CNN model and train it.

```
# define the CNN model
num_classes = len(class_names)

model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=(200, 200, 3)),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(128, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(num_classes, activation='softmax')
])

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 198, 198, 32)      896

 max_pooling2d (MaxPooling2  (None, 99, 99, 32)        0
 D)

 conv2d_1 (Conv2D)           (None, 97, 97, 64)        18496

 max_pooling2d_1 (MaxPoolin  (None, 48, 48, 64)        0
 g2D)

 conv2d_2 (Conv2D)           (None, 46, 46, 128)       73856

 max_pooling2d_2 (MaxPoolin  (None, 23, 23, 128)       0
 g2D)

 flatten (Flatten)           (None, 67712)             0

 dense (Dense)               (None, 256)               17334528

 dropout (Dropout)           (None, 256)               0

 dense_1 (Dense)             (None, 6)                 1542

=================================================================
Total params: 17429318 (66.49 MB)
Trainable params: 17429318 (66.49 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
# compile and fit
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
history = model.fit(fit_iterator, epochs = 30, verbose= 1, validation_data = valid_iterator)
```

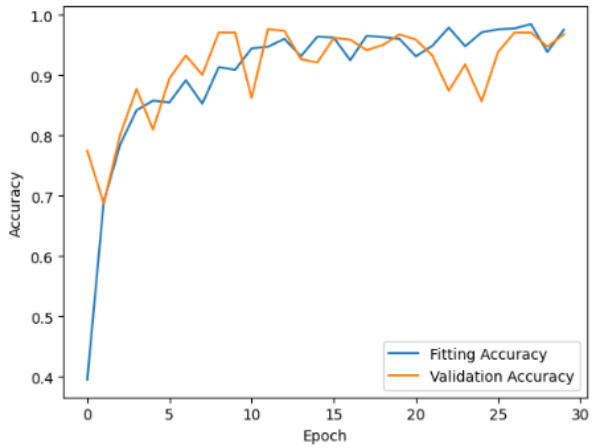
```
Epoch 1/30
87/87 [==============================] - 11s 44ms/step - loss: 1.4351 - accuracy: 0.3954 - val_loss: 0.8041 - val_accuracy: 0.7749
Epoch 2/30
87/87 [==============================] - 3s 35ms/step - loss: 0.8151 - accuracy: 0.6898 - val_loss: 0.7576 - val_accuracy: 0.6871
Epoch 3/30
87/87 [==============================] - 3s 32ms/step - loss: 0.5745 - accuracy: 0.7850 - val_loss: 0.4831 - val_accuracy: 0.8012
Epoch 4/30
87/87 [==============================] - 3s 32ms/step - loss: 0.4277 - accuracy: 0.8420 - val_loss: 0.3367 - val_accuracy: 0.8772
```

⋮

```
Epoch 26/30
87/87 [==============================] - 3s 32ms/step - loss: 0.0767 - accuracy: 0.9762 - val_loss: 0.1837 - val_accuracy: 0.9386
Epoch 27/30
87/87 [==============================] - 4s 47ms/step - loss: 0.0695 - accuracy: 0.9776 - val_loss: 0.1465 - val_accuracy: 0.9708
Epoch 28/30
87/87 [==============================] - 3s 34ms/step - loss: 0.0635 - accuracy: 0.9848 - val_loss: 0.1172 - val_accuracy: 0.9708
Epoch 29/30
87/87 [==============================] - 3s 34ms/step - loss: 0.1974 - accuracy: 0.9387 - val_loss: 0.1944 - val_accuracy: 0.9474
Epoch 30/30
87/87 [==============================] - 3s 32ms/step - loss: 0.0784 - accuracy: 0.9755 - val_loss: 0.1389 - val_accuracy: 0.9678
```

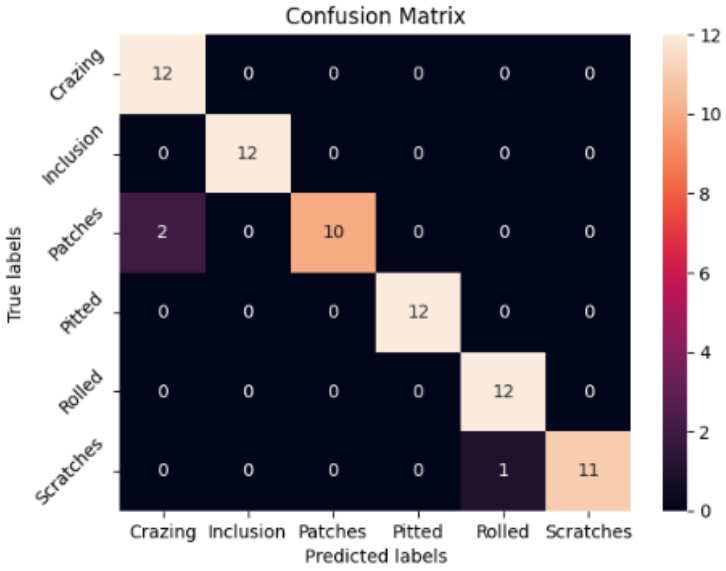
You can notice that the *fit_iterator* and the *valid_iterator* were passed along to the network fitting function wherein they are used to fetch just enough images as needed for an iteration.

```
# plot validation curve
plt.figure()
plt.plot(history.history['accuracy'], label='Fitting Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch'), plt.ylabel('Accuracy'), plt.legend()
```



Let's check the model's performance on the test images.

```
# check performance on test dataset
result = model.evaluate(test_iterator)
print("Test loss, Test accuracy : ", result)

>>> Test loss, Test accuracy :  [0.249, 0.958]

# plot confusion matrix for test dataset
from sklearn.metrics import confusion_matrix
import seaborn as sns

probs_preds = model.predict(test_iterator) # each image is assigned 6 probabilities
                                           corresponding to the 6 fault classes
labels_preds = probs_preds.argmax(axis=1) # numeric label for each image
conf_matrix = confusion_matrix(test_iterator.classes, labels_preds)

fig, ax = plt.subplots()
sns.heatmap(conf_matrix, fmt='g',  annot=True)
ax.set_xlabel('Predicted labels'), ax.set_ylabel('True labels'), ax.set_title('Confusion Matrix')
ax.set_xticklabels(class_names), ax.set_yticklabels(class_names, rotation=45)
```



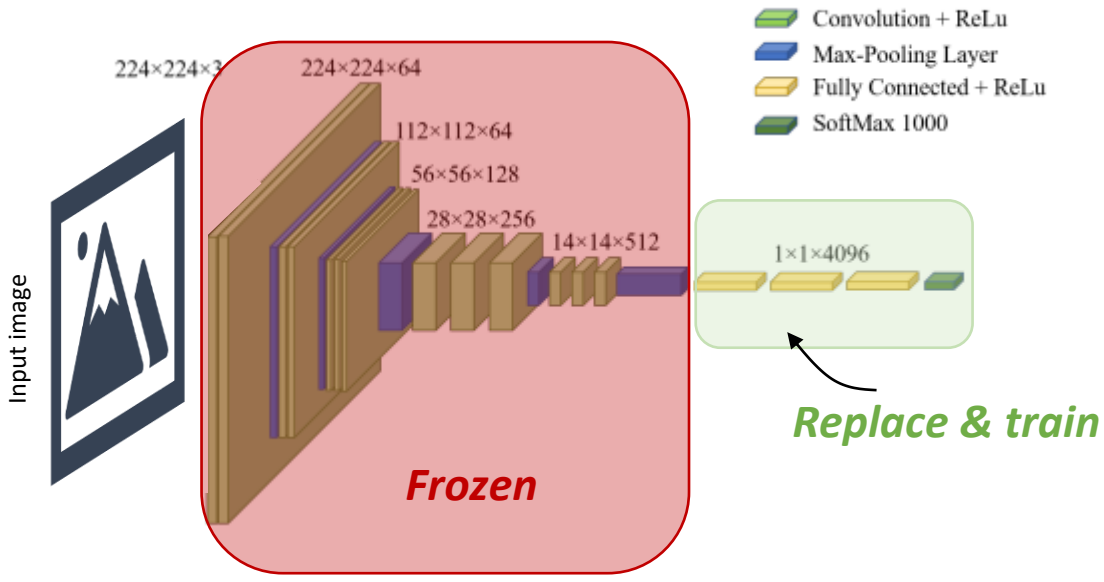
It wasn't very difficult to obtain an industrial-scale CV model, was it? Note that we did not attempt to optimize the network hyperparameters (such as the number of layers, number of filters in each convolutional layer, number of neurons in the fully connected layers, etc.). You would undertake an exhaustive hyperparameter optimization exercise to obtain a production-ready model.

4.3 Steel Defect Classification via Transfer Learning

You will often not need to define and train a CNN network from scratch. Many state-of-the-art pre-trained CV models are made available by Keras and can be loaded with just a single line of code as follows:

pretrained_model = tf.keras.applications.VGG16(weights='imagenet')

The above line instantiates a VGG16 model and downloads weights trained on ImageNet dataset. One issue with the above code is that the pre-trained model will classify your image to one of the 1000 object classes from the ImageNet dataset. However, your image dataset may be very different from the ImageNet dataset. Therefore, what you would want is to take the feature extraction part of the pre-trained model and add your own classification part to it to customize it to your specific purpose. Moreover, when you train this customized model, you would not want to change the weights of the pre-trained model; this is accomplished by 'freezing' the pre-trained model's layers as shown below[8].




```
pretrained_model = tf.keras.applications.VGG16(include_top=False, weights='imagenet')
pretrained_model.trainable = False
```

freezes the pre-trained layers

removes the classification head of the original VGG16 model

Let's again work on our NEU dataset and see how we can attach a custom classification head to a pre-trained model. Hopefully, this transfer learning exercise gives us a more accurate model than the one we got previously. The code for analyzing NEU dataset with transfer learning is provided in the file *NEU-classification-transferLearning.ipynb*. The code for data ingest pipeline remains the same. The network creation part is different as shown below. Note that we use a *MobileNet* model, a popular and computationally not very expensive model, as our base pre-trained model.

```
# define the CNN model via transfer learning
num_classes = len(class_names)
base_model = tf.keras.applications.MobileNetV2(input_shape=(200, 200, 3), include_top=False,
                                                weights='imagenet')
base_model.trainable = False

# add a trainable customized classification head to the pre-trained base model
model = models.Sequential([
    base_model,
    layers.GlobalAveragePooling2D(),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(num_classes, activation='softmax')
])

model.summary()
```

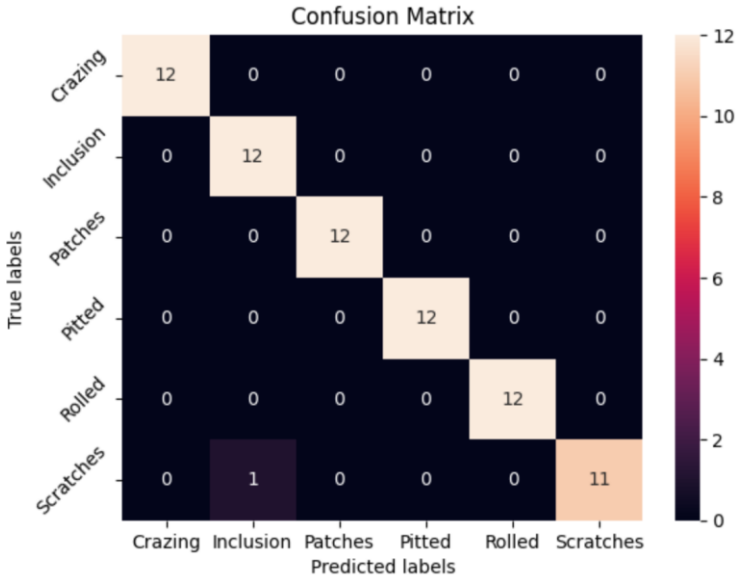
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 mobilenetv2_1.00_224 (Func  (None, 7, 7, 1280)        2257984
 tional)

 global_average_pooling2d (  (None, 1280)              0
 GlobalAveragePooling2D)

 dense (Dense)               (None, 256)               327936

 dropout (Dropout)           (None, 256)               0

 dense_1 (Dense)             (None, 6)                 1542

=================================================================
Total params: 2587462 (9.87 MB)
Trainable params: 329478 (1.26 MB)
Non-trainable params: 2257984 (8.61 MB)
_________________________________________________________________
```

```
# compile and fit
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
history = model.fit(fit_iterator, epochs = 9, verbose= 1, validation_data = valid_iterator)
```

```
Epoch 1/9
87/87 [==============================] - 17s 105ms/step - loss: 0.3001 - accuracy: 0.8975 - val_loss: 0.0308 - val_accuracy: 0.9942
Epoch 2/9
87/87 [==============================] - 7s 75ms/step - loss: 0.0806 - accuracy: 0.9726 - val_loss: 0.0302 - val_accuracy: 0.9912
Epoch 3/9
87/87 [==============================] - 6s 70ms/step - loss: 0.0361 - accuracy: 0.9913 - val_loss: 0.0250 - val_accuracy: 0.9942
Epoch 4/9
87/87 [==============================] - 6s 72ms/step - loss: 0.0275 - accuracy: 0.9877 - val_loss: 0.0259 - val_accuracy: 0.9912
Epoch 5/9
87/87 [==============================] - 6s 69ms/step - loss: 0.0230 - accuracy: 0.9928 - val_loss: 0.0236 - val_accuracy: 0.9912
Epoch 6/9
87/87 [==============================] - 4s 47ms/step - loss: 0.0187 - accuracy: 0.9949 - val_loss: 0.0181 - val_accuracy: 0.9912
Epoch 7/9
87/87 [==============================] - 6s 58ms/step - loss: 0.0118 - accuracy: 0.9986 - val_loss: 0.0246 - val_accuracy: 0.9912
Epoch 8/9
87/87 [==============================] - 5s 57ms/step - loss: 0.0077 - accuracy: 0.9993 - val_loss: 0.0155 - val_accuracy: 0.9942
Epoch 9/9
87/87 [==============================] - 6s 72ms/step - loss: 0.0180 - accuracy: 0.9957 - val_loss: 0.0173 - val_accuracy: 0.9971
```

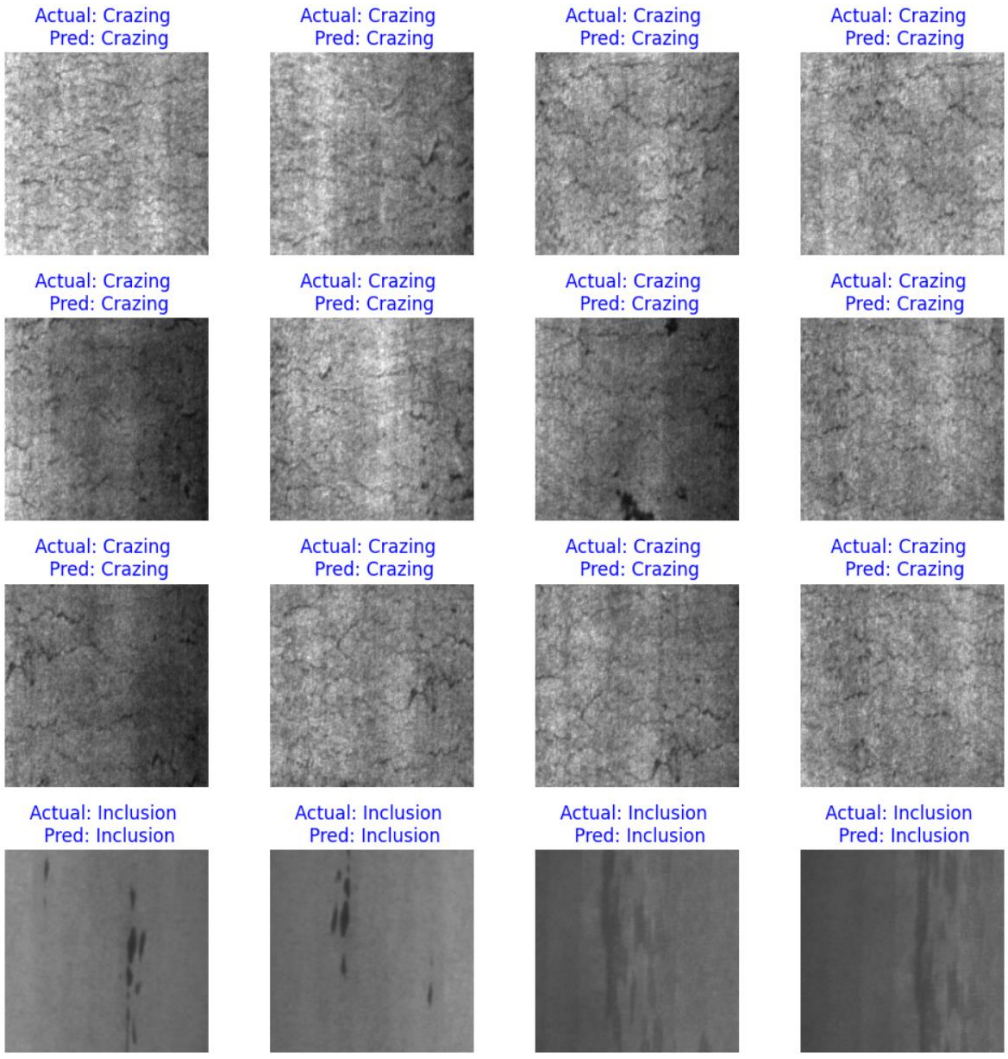
Did you notice that with transfer learning we reach pretty high fitting and validation accuracies in just a couple of epochs? Let's check the model's performance on the test images.

```
# check performance on test dataset
result = model.evaluate(test_iterator)
print("Test loss, Test accuracy : ", result)

>>> Test loss, Test accuracy :  [0.024, 0.986]
```




```
# visualize predictions
images, labels = test_iterator.next()

fig = plt.figure()
for index, image in enumerate(images):
    ax = fig.add_subplot(4, 4, index+1)
    plt.imshow(image)
```

```
image = np.expand_dims(image, 0) # adding a batch dimension
probs_pred = model.predict(image) # softmax probabilities for the 6 fault classes
probs_pred = np.squeeze(probs_pred) # removing the batch dimension

label_actual = np.argmax(labels[index])
label_pred = np.argmax(probs_pred)

if label_pred == label_actual:
   color='blue'
else:
   color='red'

ax.set_title(f'Actual: {class_names[label_actual]} \n Pred: {class_names[label_pred]}', color=color)
ax.axis('off')
```

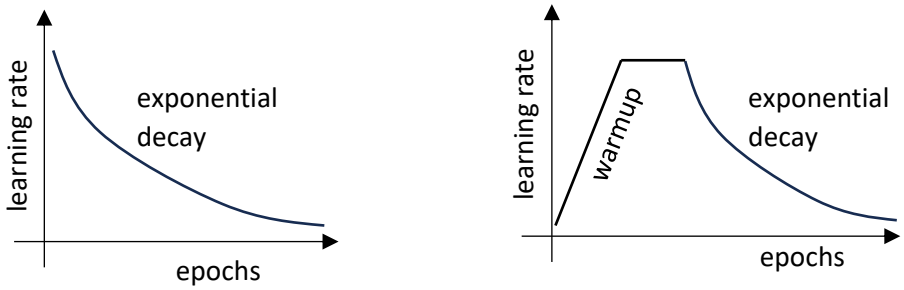


Hopefully, you can appreciate the ease with which you can use state-of-the-art pre-trained models using modern frameworks like Keras.

Fine-tuning

You may still be marveling at how a model trained using ImageNet dataset (*https://www.image-net.org/about.php*) be used successfully on a completely unrelated NEU steel strip dataset. Well, that's the beauty of computer vision. Any object in an image is a sum-total of some low-level features; the feature extraction base of the pre-trained models 'knows' how to extract the low-level features and your custom classification head then does the rest of the job.

Nonetheless, you may encounter situations where transfer learning does not yield satisfactory performance and therefore the feature extraction layers need to be customized as well for your specific problem. A common recourse in such cases is to fine-tune your trained model after transfer learning. To accomplish this, you can unfreeze the base layers and train you model for a few more epochs. The choice of learning rate becomes crucial as too high a learning rate may 'ruin' the pre-trained weights of the convolutional base and too low a learning rate can make the training convergence very slow. A customary practice is to adjust the learning rate with epochs as shown below



Summary

This chapter provided a whirlwind tour of building CNN-based CV models. Two approaches were covered: building CNN models from scratch and via transfer learning. In this chapter, we also covered the concept of lazy loading of images and the importance of properly designing the image ingest pipeline. An easy-to-understand approach using ImageDataGenerator was illustrated and more modern approaches were referenced. This chapter should give you a very good idea of what goes behind creating modern CV models. There are many CV-related concepts (such as data augmentation, creating production-level modeling pipelines on cloud, etc.) that we did not touch upon (or covered in detail) in this short book. Nonetheless, you should now have the basic foundations to approach the advanced CV topics with confidence.

Chapter 5

Automated Equipment Monitoring Using Sound

Operating plant equipment at their mechanical limits with high reliability is the new craze among plant managers in the Industry 4.0 era. Proactive, automated, and smart monitoring of equipment is key to achieving the aforementioned objectives. Sound-based anomaly detection is one of the techniques used to detect equipment abnormalities at early stages of failures. The rationale is simple: If a machine starts making abnormal sound, then most likely it is malfunctioning. Acoustic monitoring is also attractive due to the ease of installation of the microphone sensors to capture sound. Correspondingly, implementations of audio-based predictive maintenance solutions are on the rise in process industry.

Despite its apparent advantages, audio-based anomaly detection is not among the top equipment condition monitoring techniques. Lack of adequate data from past malfunctions hinders development of supervised fault detection (and classification) solutions. One may build fault detection solutions using sound data taken from only normally functioning machines; however, preventing false alerts due to background noise remains a big challenge. Nonetheless, smart algorithms are being devised to handle background noise appropriately and therefore, acoustics-based equipment monitoring is poised to gain more prominence.

Traditionally, abnormal sound detection has been performed through explicit feature engineering using the sound signal in its time-waveform, frequency, and time-frequency representations. Now-a-days, deep learning is frequently employed to automatically extract relevant features. In this chapter we will look at both of these techniques. Specifically, the following topics are covered

- Introduction to air compressor sound dataset
- Abnormal sound detection using classical machine learning
- Abnormal sound detection using deep learning

5.1 Air Compressor Sound Dataset

The acoustic dataset (https://www.iitk.ac.in/idea/datasets) that we will work with in this chapter was collected on a reciprocating type air compressor. The dataset contains 225 audio files (each 3 seconds long) for each of the eight conditions of the compressor.

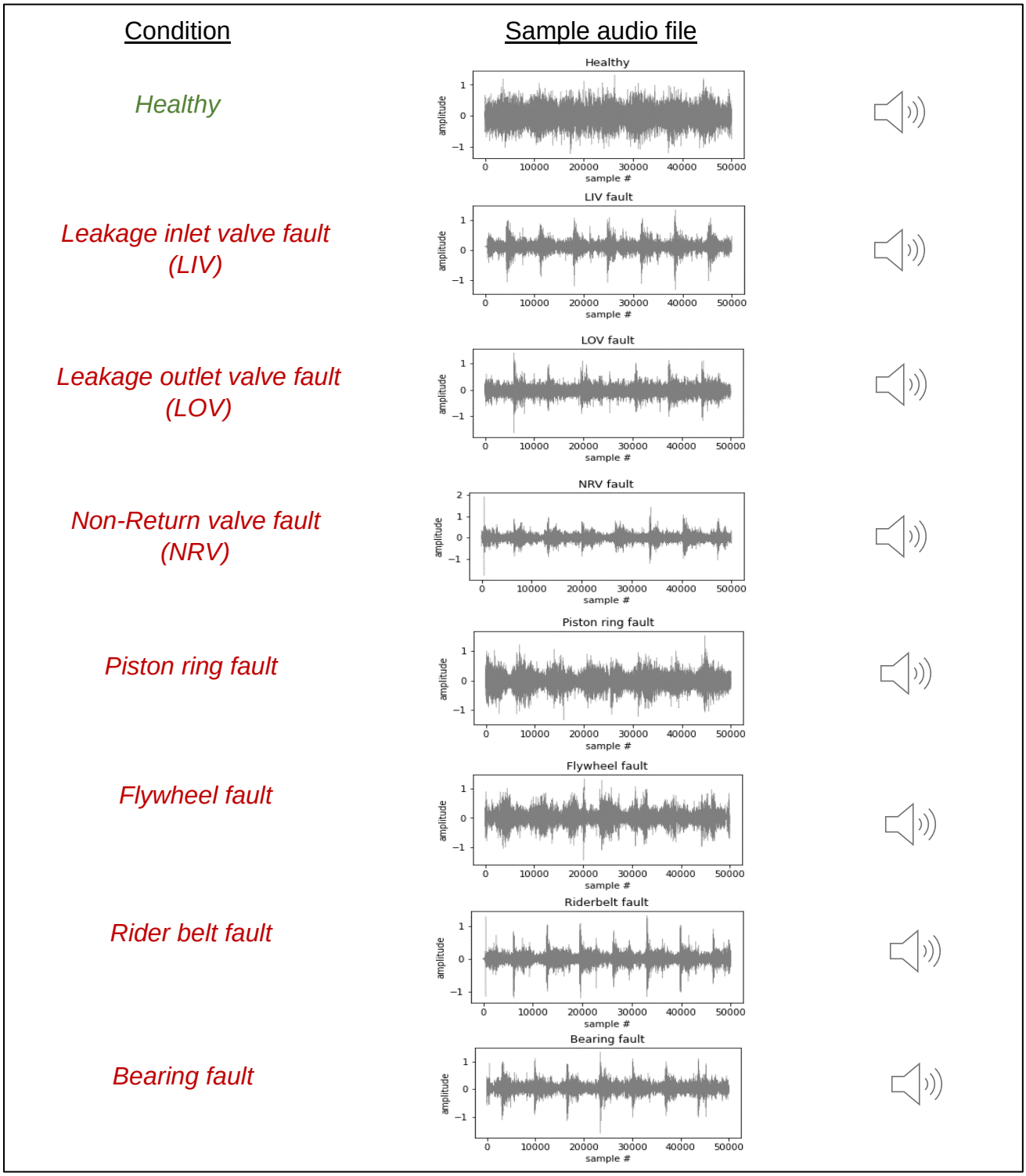


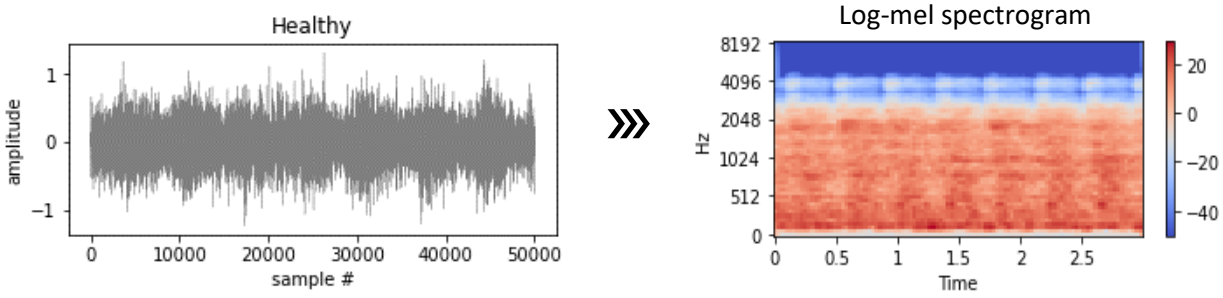
Figure 5.1: Classes of audio files in the air compressor sound dataset [use the notebook file *soundDataExplore.ipynb* to play the sound for each of the machine state]

The authors of the dataset also provide the features extracted from the audio files in a separate dataset[15]. Features are extracted in the frequency and time-frequency domains. We will use this feature dataset to build an SVM-based fault classification model.

Converting sound signal into images

We saw in Chapter 1 that a sound signal is simply a vibration signal, and we know that a spectrogram (introduced in Book 3 of the series) is a very useful time-frequency representation of a vibration signal. A spectrogram provides a 2D visual representation of the sound signal and we have seen that CNNs excel at analyzing such visual inputs. Therefore, computer vision techniques can be employed for abnormal sound detection as well!

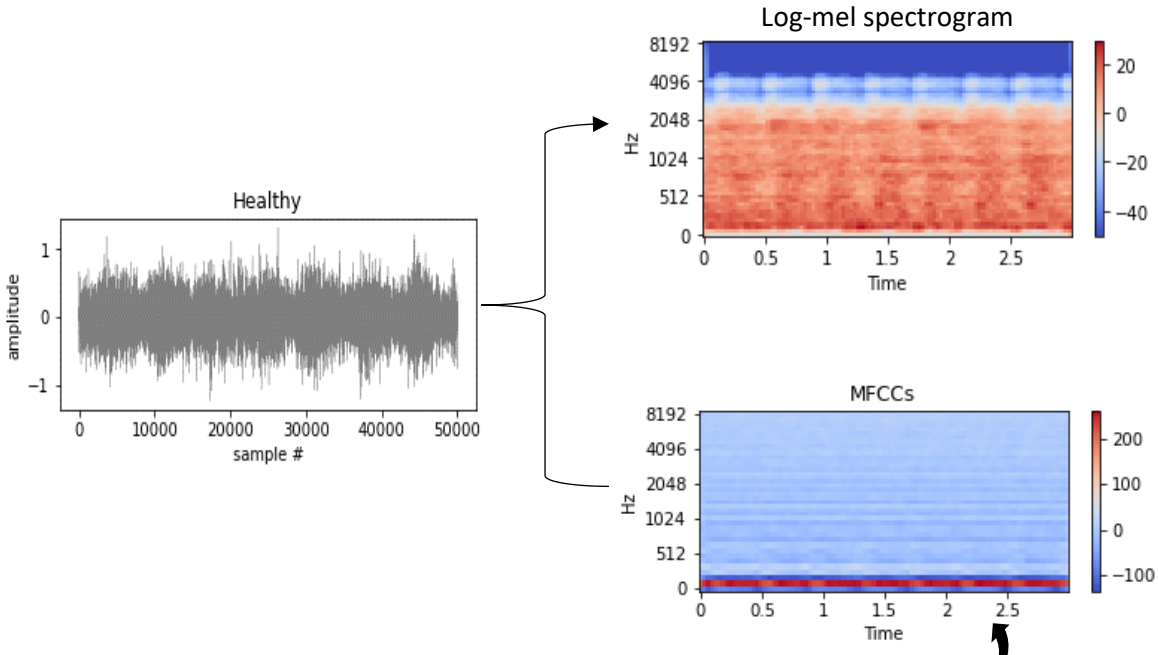
For audio signal analysis, instead of spectrogram, log-mel spectrogram is commonly employed. A log-mel spectrogram is a type of spectrogram where the frequency axis is not linear but based on mel scale[16]. The mel scale provides better resolution for lower frequencies and therefore mirrors the sensitivity of human ears (humans can differentiate between 500 Hz and 600 Hz better than differentiating between 10,000 Hz and 10100 Hz!). A log-mel spectrogram can be easily obtained using the *Librosa* library.




```
import numpy as np, matplotlib.pyplot as plt
import librosa
import librosa.display

clipPath = "AirCompressor_Data/Healthy/preprocess_Reading1.dat"
data = np.loadtxt(clipPath, delimiter=',')

# time waveform
plt.figure(figsize=(5,2))
plt.plot(data, 'grey', linewidth=0.3)
plt.title('Healthy'), plt.xlabel('sample #'), plt.ylabel('amplitude')

# spectrogram
sr = 50000/3 # sampling rate: 50000 samples per 3 seconds
mel_spectrogram = librosa.feature.melspectrogram(y=data, sr=sr, n_mels=64)
log_mel_spectrogram = librosa.power_to_db(mel_spectrogram)
plt.figure(figsize=(5,2))
img = librosa.display.specshow(log_mel_spectrogram, x_axis="time", y_axis="mel", sr=sr)
plt.colorbar()
```

[15] Download the 'Air Compressor Health State Dataset Features' dataset
[16] https://en.wikipedia.org/wiki/Mel_scale

In the acoustic analysis world, another 2D visual representation that is commonly employed is called MFCCs (mel frequency cepstral coefficients) which is a compressed version of a log-mel spectrogram.



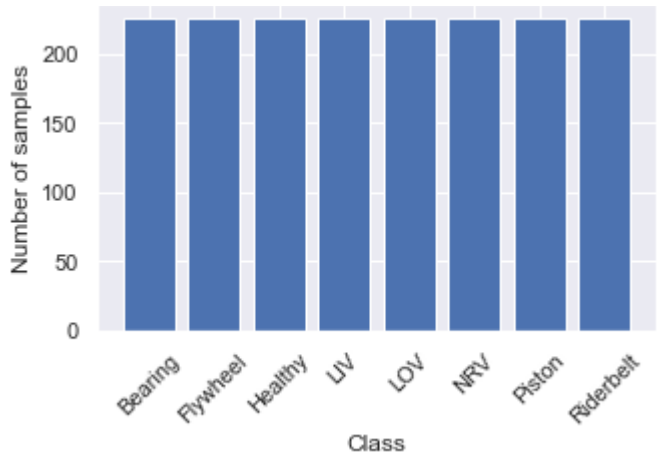

```
# mfcc
MFCCs=librosa.feature.mfcc(y=data, sr=sr, n_mfcc=40)
plt.figure(figsize=(5,2))
librosa.display.specshow(MFCCs, x_axis="time", y_axis="mel", sr=sr)
plt.colorbar(), plt.title('MFCCs')
```

5.2 Abnormal Equipment Sound Classification using Support Vector Machines

To illustrate how abnormal sound detection and classification can be performed using classical machine learning techniques, we will use the feature dataset to train a support vector machine classifier as shown below.

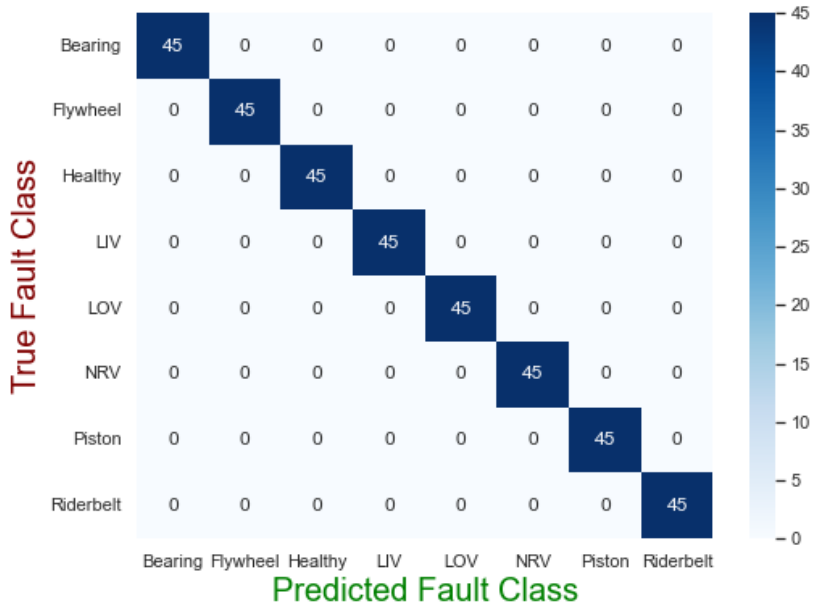
```
# import required packages
import numpy as np, pandas as pd, matplotlib.pyplot as plt, seaborn as sn
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC
```

```
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import confusion_matrix


# read data [the provided csv file contains 1800 rows: each row contains features extracted from
                                    # data from an audio file and the associated label (in the last column)]
dataRaw = pd.read_csv('WPTFeatures254_FaultStates.csv', delimiter=',')
print(dataRaw.shape)

>>> (1800, 255)

# separate features and label
input_data = dataRaw.iloc[:,:-1].values
output_label_text = dataRaw.iloc[:,-1]

# convert text labels to numeric labels
le = LabelEncoder().fit(output_label_text)
output_labels = le.transform(output_label_text)
print(le.classes_)

>> ['Bearing' 'Flywheel' 'Healthy' 'LIV' 'LOV' 'NRV' 'Piston' 'Riderbelt']

# check number of samples for each class
unique_labels, counts = np.unique(output_labels, return_counts=True)

plt.figure(), plt.bar(unique_labels, counts)
plt.xlabel('Class'), plt.ylabel('Number of samples')
plt.xticks(range(len(unique_labels)), labels=le.classes_, rotation=45)
```



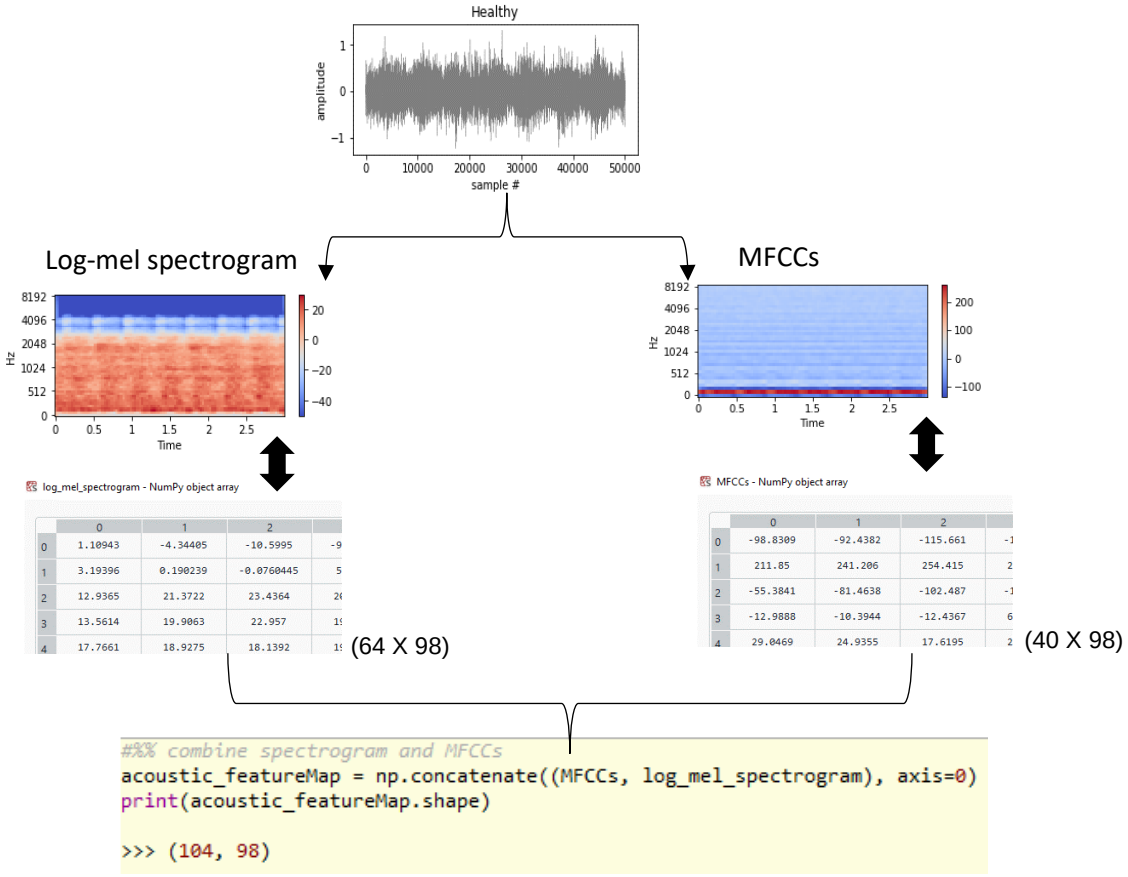

```
# separate training and test data and scale
X_train, X_test, y_train, y_test = train_test_split(input_data, output_labels, test_size=0.2,
                                                    stratify=output_labels, random_state=100)
```

```
scaler = StandardScaler().fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# fit model with grid search-based hyperparameter determination
param_grid = {'C':[0.01, 0.05, 0.1, 1, 10]}
gs = GridSearchCV(LinearSVC(dual='auto'), param_grid, cv=3).fit(X_train_scaled, y_train)
print('Optimal hyperparameter:', gs.best_params_)

>>> Optimal hyperparameter: {'C': 0.05}

# predict for test data and plot confusion matrix
y_test_pred = gs.predict(X_test_scaled)
conf_mat = confusion_matrix(y_test, y_test_pred)

plt.figure()
sn.heatmap(conf_mat, annot=True, cmap='Blues', xticklabels=le.classes_, yticklabels=le.classes_)
plt.ylabel('True Fault Class', color='maroon')
plt.xlabel('Predicted Fault Class', color='green')
```



5.3 Abnormal Equipment Sound Classification using CNN

The perfect accuracy of the SVM classifier suggests that the time-frequency features contain enough information to enable differentiation between the audio data associated with different machine health states. Let's see if we can employ a CNN to extract these features directly from the log-mel spectrogram and MFCCs 'images'. The input feature map for each audio file is created by concatenating the log-mel spectrogram and MFCC matrices as shown below



We will again work[17] in Google Colab and like we did in Chapter 4, we will upload air compressor data on Google Drive. Complete code is provided in the notebook *AirCompressorSound-classification.ipynb*. Let's begin by unzipping the Google Drive file in Colab and importing some packages.

[17] The CNN-based ASD approach presented in this chapter is adapted from the approach provided at https://github.com/SAP-samples/btp-ai-sustainability-bootcamp/blob/main/src/ai-models/predictive-maintenance/notebooks/sound_based_predictive_maintenance.ipynb which is shared under Apache License 2.0.

```
# unzip
!unzip -q -o './drive/MyDrive/AirCompressor_Data.zip' -d './'

# import packages
import numpy as np, pandas as pd, matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from tensorflow.keras import models, layers
import tensorflow as tf
import librosa
from glob import glob
import seaborn as sns
```

Now we will prepare data for training and test. Each audio file needs to be converted into its combined spectrogram and MFCC matrix. Let's first assign the health label to each audio file.

```
# get file paths and assign corresponding class labels
clips = glob('./AirCompressor_Data/*/*') # fetches all the file names
clips_df = pd.DataFrame(data={'path':clips, 'label':[ c.split('/')[-2]  for c in clips]} )

class_dict=dict(enumerate(clips_df.label.unique() ))
classes = {v: k for k, v in class_dict.items()}
clips_df['class']=clips_df['label'].apply(lambda x : classes[x]) # each file is assigned a sparse label
clips_df
```

	path	label	class
0	./AirCompressor_Data/Riderbelt/preprocess_Read...	Riderbelt	0
1	./AirCompressor_Data/Riderbelt/preprocess_Read...	Riderbelt	0
2	./AirCompressor_Data/Riderbelt/preprocess_Read...	Riderbelt	0
3	./AirCompressor_Data/Riderbelt/preprocess_Read...	Riderbelt	0
4	./AirCompressor_Data/Riderbelt/preprocess_Read...	Riderbelt	0
...	...	...	...
1795	./AirCompressor_Data/Bearing/preprocess_Readin...	Bearing	7
1796	./AirCompressor_Data/Bearing/preprocess_Readin...	Bearing	7
1797	./AirCompressor_Data/Bearing/preprocess_Readin...	Bearing	7
1798	./AirCompressor_Data/Bearing/preprocess_Readin...	Bearing	7
1799	./AirCompressor_Data/Bearing/preprocess_Readin...	Bearing	7

1800 rows × 3 columns

Next, we define a utility function that takes an audio clip and returns its combined acoustic features: log-mel spectrogram and MFCCs.

```
sr=50000/3
def acoustic_feature_computation(clip):
    data = np.loadtxt(clip, delimiter=',')
    mel_spectrogram = librosa.feature.melspectrogram(y=data, sr=sr, hop_length=512,
                                                     n_mels=64, fmax=sr/2)
    log_mel_spectrogram = librosa.power_to_db(mel_spectrogram)
    MFCCs = librosa.feature.mfcc(y=data, sr=sr, n_mfcc=40, fmax=sr/2)
    acoustic_features = np.concatenate((MFCCs,log_mel_spectrogram), axis =0)
    return acoustic_features
```

We can now create our fitting, validation, and test datasets.

```
train, test = train_test_split(clips_df, test_size=0.10, random_state=25)
train, validation = train_test_split(train, test_size=0.15, random_state=25)

# apply the function acoustic_feature_computation to each audio file
X_train, X_validation, X_test = [], [], []
y_train, y_validation, y_test = [], [], []

for i,r in train.iterrows(): # Iterate over the DataFrame rows
    X_train.append(acoustic_feature_computation(r['path']))
    y_train.append(r['class'])

for i,r in test.iterrows():
    X_test.append(acoustic_feature_computation(r['path']))
    y_test.append(r['class'])

for i,r in validation.iterrows():
    X_validation.append(acoustic_feature_computation(r['path']))
    y_validation.append(r['class'])
```

With the datasets prepared, we can build the CNN model now.

```
initializer = tf.keras.initializers.GlorotUniform()
CNNmodel = models.Sequential()

# feature extraction part
CNNmodel.add(layers.Conv2D(32, (4, 4),(2,2), activation='relu', input_shape=(104,98,1),
kernel_initializer=initializer))
CNNmodel.add(layers.BatchNormalization())
CNNmodel.add(layers.Conv2D(32, (4, 4),(2,2), activation='relu', kernel_initializer=initializer))
CNNmodel.add(layers.BatchNormalization())
CNNmodel.add(layers.MaxPooling2D((2, 2)))
```

```
# classification head
CNNmodel.add(layers.Flatten())
CNNmodel.add(layers.Dense(512, activation='relu',kernel_initializer=initializer))
CNNmodel.add(layers.Dropout(0.5))
CNNmodel.add(layers.Dense(64, activation='relu',kernel_initializer=initializer))
CNNmodel.add(layers.Dropout(0.5))

# Output
CNNmodel.add(layers.Dense(8, activation='softmax'))

CNNmodel.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 51, 48, 32)        544

 batch_normalization (Batch  (None, 51, 48, 32)        128
 Normalization)

 conv2d_1 (Conv2D)           (None, 24, 23, 32)        16416

 batch_normalization_1 (Bat  (None, 24, 23, 32)        128
 chNormalization)

 max_pooling2d (MaxPooling2  (None, 12, 11, 32)        0
 D)

 flatten (Flatten)           (None, 4224)              0

 dense (Dense)               (None, 512)               2163200

 dropout (Dropout)           (None, 512)               0

 dense_1 (Dense)             (None, 64)                32832

 dropout_1 (Dropout)         (None, 64)                0

 dense_2 (Dense)             (None, 8)                 520

=================================================================
Total params: 2213768 (8.44 MB)
Trainable params: 2213640 (8.44 MB)
Non-trainable params: 128 (512.00 Byte)
_________________________________________________________________
```

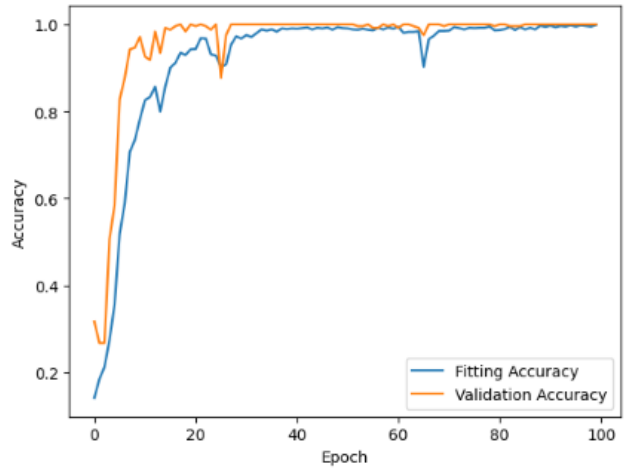
```
# compile and fit
CNNmodel.compile(optimizer= "adam", loss=tf.keras.losses.SparseCategoricalCrossentropy(),
                 metrics = ['accuracy'])
history = CNNmodel.fit(x=np.array(X_train, np.float32), y=np.array(y_train, np.float32),
                       validation_data = (np.array(X_validation, np.float32),
                                          np.array(y_validation, np.float32)),
                       epochs=100)
```

```
Epoch 1/100
44/44 [==============================] - 7s 25ms/step - loss: 2.7447 - accuracy: 0.1409 - val_loss: 1.9680 - val_accuracy: 0.3169
Epoch 2/100
44/44 [==============================] - 0s 9ms/step - loss: 2.1619 - accuracy: 0.1845 - val_loss: 1.8348 - val_accuracy: 0.2675
Epoch 3/100
44/44 [==============================] - 0s 9ms/step - loss: 2.0106 - accuracy: 0.2121 - val_loss: 1.7488 - val_accuracy: 0.2675
Epoch 4/100
44/44 [==============================] - 0s 9ms/step - loss: 1.8272 - accuracy: 0.2745 - val_loss: 1.4656 - val_accuracy: 0.5062
Epoch 5/100
44/44 [==============================] - 0s 9ms/step - loss: 1.6222 - accuracy: 0.3558 - val_loss: 1.3811 - val_accuracy: 0.5844
Epoch 6/100
44/44 [==============================] - 0s 9ms/step - loss: 1.2261 - accuracy: 0.5178 - val_loss: 0.8089 - val_accuracy: 0.8272
```

⋮

```
Epoch 95/100
44/44 [==============================] - 0s 7ms/step - loss: 0.0089 - accuracy: 0.9978 - val_loss: 1.9476e-07 - val_accuracy: 1.0000
Epoch 96/100
44/44 [==============================] - 0s 6ms/step - loss: 0.0194 - accuracy: 0.9949 - val_loss: 2.0996e-07 - val_accuracy: 1.0000
Epoch 97/100
44/44 [==============================] - 0s 7ms/step - loss: 0.0098 - accuracy: 0.9978 - val_loss: 3.6497e-06 - val_accuracy: 1.0000
Epoch 98/100
44/44 [==============================] - 0s 7ms/step - loss: 0.0143 - accuracy: 0.9964 - val_loss: 6.6849e-05 - val_accuracy: 1.0000
Epoch 99/100
44/44 [==============================] - 0s 7ms/step - loss: 0.0160 - accuracy: 0.9949 - val_loss: 2.5215e-07 - val_accuracy: 1.0000
Epoch 100/100
44/44 [==============================] - 0s 7ms/step - loss: 0.0075 - accuracy: 0.9985 - val_loss: 9.4818e-07 - val_accuracy: 1.0000
```

```
# plot validation curve
plt.figure()
plt.plot(history.history['accuracy'], label='Fitting Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch'), plt.ylabel('Accuracy'), plt.legend()
```

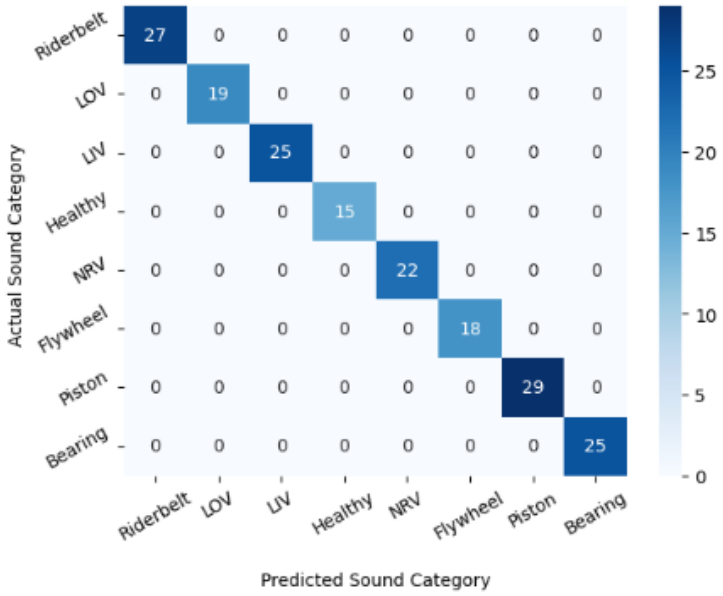


Our CNN model seems to be a perfect classifier with 100% accuracy on validation dataset. Let's check its performance for test dataset.

```
score = CNNmodel.evaluate(np.array(X_test, np.float32), np.array(y_test, np.float32))
print('Test accuracy:', score[1])
```

```
>>> Test accuracy: 1.0
```

```
# plot confusion matrix for test dataset
from sklearn.metrics import confusion_matrix

probs_preds = CNNmodel.predict(np.array(X_test, np.float32)) # each audio file is assigned 8 probabilities corresponding to the 8 health states
pred_class = probs_preds.argmax(axis=1) # numeric class for each audio file
cf_matrix = confusion_matrix(y_test, pred_class)

ax = sns.heatmap(cf_matrix, annot=True, cmap='Blues')
ax.set_xlabel('\nPredicted Sound Category'), ax.set_ylabel('Actual Sound Category ')
ax.set_xticklabels(class_dict.values(), rotation=30), ax.set_yticklabels(class_dict.values(), rotation=30)
```



With this, we have come to the end of our quick foray into the world of CNN-based deep learning and its usage for building process monitoring solutions using visual and audio data. We saw how easy it is to build powerful modern equipment monitoring solutions using CNN. If desired, you can also build a combined audio and video-based smart process surveillance solution using the approaches covered in this book.

Summary

In this chapter, we quickly covered a couple of approaches for abnormal equipment sound detection. The illustrations covered the traditional approach entailing explicit feature engineering and the CNN-based deep learning approach for monitoring reciprocating air compressors.

End of the book

Machine Learning in Python for Visual and Acoustic Data-based Process Monitoring

This book is designed to help readers gain quick familiarity with deep learning-based computer vision and abnormal equipment sound detection techniques. The book helps you take your first step towards learning how to use convolutional neural networks (the ANN architecture that is behind the modern revolution in computer vision) and build image sensor-based manufacturing defect detection solutions. A quick introduction is also provided to how modern predictive maintenance solutions can be built for process-critical equipment by analyzing the sound generated by the equipment. Overall, this short eBook sets the foundation with which budding process data scientists can confidently navigate the world of modern computer vision and acoustic monitoring.

The following topics are briefly covered:

- *Introduction to computer vision (CV) and CNNs*
- *Best practices for building CV solutions for detecting manufacturing defects*
- *Building CNN-based CV solutions from scratch and via transfer learning*
- *Introduction to equipment sound monitoring*
- *Building equipment abnormal sound detection solutions using CNNs*