Institute of Geophysics
Polish Academy of Sciences

INSTITUTE OF GEOPHYSICS
POLISH ACADEMY OF SCIENCES

PUBLICATIONS
OF THE INSTITUTE OF GEOPHYSICS
POLISH ACADEMY OF SCIENCES

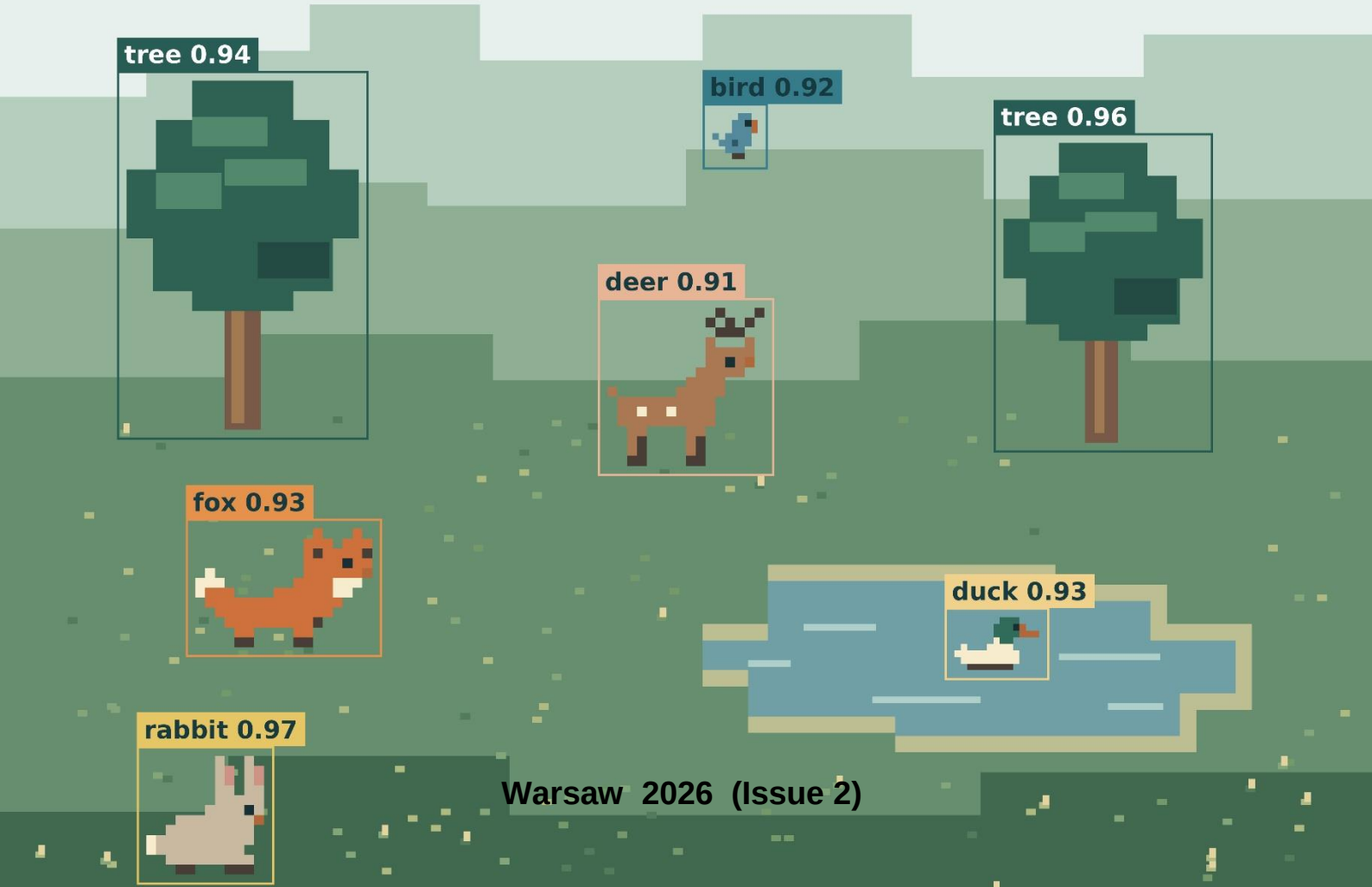
Geophysical Data Bases, Processing and Instrumentation

457 (M-39)

Practical Deep Learning with Python

A Beginner's Handbook in Four Lessons

Piotr KLEJMENT



Warsaw 2026 (Issue 2)

INSTITUTE OF GEOPHYSICS
POLISH ACADEMY OF SCIENCES

PUBLICATIONS
OF THE INSTITUTE OF GEOPHYSICS
POLISH ACADEMY OF SCIENCES

Geophysical Data Bases, Processing and Instrumentation

457 (M-39)

Practical Deep Learning with Python

A Beginner's Handbook in Four Lessons

Piotr KLEJMENT

Warsaw 2026

Editor-in-Chief
Marek KUBICKI

Advisory Editorial Board
Tomasz ERNST (Institute of Geophysics, PAS)
Maria JELEŃSKA (Institute of Geophysics, PAS)
Andrzej KIJKO (University of Pretoria, Pretoria, South Africa)
Natalia KLEIMENOVA (Institute of Physics of the Earth, Russian Academy of Sciences, Moscow, Russia)
Jan KOZAK (Geophysical Institute, Prague, Czech Republic)
Antonio MELONI (Instituto Nazionale di Geofisica, Rome, Italy)
Hiroyuki NAGAHAMA (Tohoku University, Sendai, Japan)
Kaja PIETSCH (AGH University of Science and Technology, Cracow, Poland)
Paweł M. ROWIŃSKI (Institute of Geophysics, PAS)
Steve WALLIS (Heriot Watt University, Edinburgh, United Kingdom)
Wacław M. ZUBEREK (University of Silesia, Sosnowiec, Poland)

Associate Editors
Łukasz RUDZIŃSKI (Institute of Geophysics, PAS) – **Solid Earth Sciences**
Jan WISZNIOWSKI (Institute of Geophysics, PAS) – **Seismology**
Jan REDA (Institute of Geophysics, PAS) – **Geomagnetism**
Krzysztof MARKOWICZ (Institute of Geophysics, Warsaw University) – **Atmospheric Sciences**
Mark GOŁKOWSKI (University of Colorado Denver) – **Ionosphere and Magnetosphere**
Andrzej KUŁAK (AGH University of Science and Technology) – **Atmospheric Electricity**
Marzena OSUCH (Institute of Geophysics, PAS) – **Hydrology**
Daniel KĘPSKI (Institute of Geophysics, PAS) – **Polar Sciences**

Managing Editor
Bartosz WOŹNIAK

Technical Editor
Marzena CZARNECKA

© 2026 The Author(s). Published by the Institute of Geophysics, Polish Academy of Sciences.
This is an open access publication under the CC BY license 4.0

ISBN 978-83-66254-29-9 eISSN-2299-8020
DOI: 10.25171/InstGeoph_PAS_Publs-2026-002

The cover presents a schematic, pixel-art illustration of object detection, a core task in computer vision, with bounding boxes and confidence scores for each detected object. The artwork was generated by author from scratch using Python 3 and its libraries.

Editorial Office
Instytut Geofizyki Polskiej Akademii Nauk
ul. Księcia Janusza 64, 01-452 Warszawa

To my mother, Wiesława, a longtime librarian,
who taught me a love of books and knowledge

PREFACE

This study guide is not intended as a comprehensive textbook, but rather as a handbook for individuals with a foundational understanding of Python programming who are just beginning their adventure with Deep Learning. The text encompasses explanations of fundamental theoretical and practical terms related to subjectively selected Deep Learning techniques with relevant examples.

The materials included in this study were developed on the basis of a series of lectures on Machine Learning and Python programming delivered by the author since 2021, first to doctoral students at the Institute of Geophysics of the Polish Academy of Sciences and later to doctoral students at the GeoPlanet Doctoral School. However, the main basis of the present study is a set of lecture notes prepared for the course “Deep Learning with Python”, delivered to doctoral students at the GeoPlanet Doctoral School in the spring of 2026.

CONTENTS

1. Introduction – Subjective division of artificial intelligence techniques into three categories 5
2. Lecture 1 – Multilayer Perceptron 8
2.1 Example 1a 10
2.2 Building MLP – input 13
2.3 Building MLP – hidden layers 16
2.4 Building MLP – output 18
2.5 Example 1b 21
2.6 Example 1c 26
2.7 MLP – summary 32
2.8 Sequential architecture and others 34
2.9 Example 1d 36
3. Lecture 2 – Recurrent Neural Networks and times series 39
3.1 MLP vs RNN vs LSTM 39
3.2 Alternatives 42
3.3 Example 2 43
4. Lecture 3 – Convolutional Neural Networks 52
4.1 Key CNN components 52
4.2 Example 3 55
5. Lecture 4 – Automatic Speech Recognition and Wav2Vec 2.0 67
5.1 Example 4 68
5.2 Wav2Vec 2.0 and geosciences 70
5.3 Implementation 70
5.4 Detailed description – Step 1 72
5.5 Detailed description – Step 2 76
6. Summary 78
References 81

1. INTRODUCTION – SUBJECTIVE DIVISION OF ARTIFICIAL INTELLIGENCE TECHNIQUES INTO THREE CATEGORIES

Artificial intelligence is an incredibly broad term, and the multitude of techniques, models, and architectures is overwhelming. The term itself, along with its core ideas and theoretical assumptions, have existed for decades, but it wasn't until a dozen or so years ago that the true revolution began. The first stage was the release of the TensorFlow 2 and Keras libraries (Google) and PyTorch (Facebook) in 2015. This allowed a range of specialists, scientists, and programmers to begin applying AI methods in their work without having to implement entire networks and models from scratch. The second stage of this revolution was the creation and release of Large Language Models (like ChatGPT in late 2022). In this way, artificial intelligence became a mass and democratic technology that everyone could and can benefit from (LeCun et al. 2015; Mousavi and Beroza 2022; Goodfellow et al. 2016; Bishop 2006; Bishop and Bishop 2023; Prince 2023; Zhang et al. 2023; Murphy 2023; Géron 2022; Raschka and Mirjalili 2019; Liu 2020).

The following section presents the author's subjective classification of artificial intelligence techniques into three categories. Its purpose is not to provide a comprehensive overview of all available models and architectures, but rather to highlight those that the author considers the most important. From the perspective of the author, who is also a researcher, Category 1 includes techniques that, by 2026, had become widely established and were already used in thousands of scientific publications. Categories 2 and 3, in contrast, include methods that are currently regarded as emerging "hot topics" or that may gain such status in the future. Compared with the techniques included in Category 1, their scientific applications remain relatively limited and are generally more recent and innovative.

Category 1. Established and widely used machine learning and Deep Learning techniques

This category includes well-established methods that have been applied for many years across a wide range of scientific disciplines. Their theoretical foundations are well understood, mature software implementations are readily available, and their applications have been described in thousands of scientific publications.

1A) Supervised Machine Learning for tabular data – regression and classification

Supervised learning uses labelled data to predict either a discrete class or a continuous numerical value. It is commonly applied when the input consists of a fixed set of calculated or measured features.

Techniques: linear and logistic regression, Support Vector Machines, Decision Trees, Random Forest, XGBoost, and LightGBM.

Applications: classification of seismic-event types, estimation of earthquake magnitude, assessment of station data quality, and prediction of ground-deformation rates.

1B) Unsupervised Machine Learning

Unsupervised learning identifies patterns and structures in data without predefined target labels.

Clustering: k-means, DBSCAN, and Hierarchical Clustering.

Applications: grouping seismic stations with similar noise characteristics, identifying groups of similar events, and detecting seismic swarms.

Dimensionality reduction: PCA, t-SNE, and UMAP.

Applications: compression of multichannel data, feature reduction, and visualization of high-dimensional datasets.

Anomaly detection: Isolation Forest and statistical methods.
Applications: detection of unusual measurements, sensor failures, outliers, and potentially unusual natural events.

1C) Deep Learning – Feed-Forward Networks: Multilayer Perceptron / fully connected networks

A Multilayer Perceptron is a neural network composed of fully connected layers. It learns nonlinear relationships between input variables and can be used for both classification and regression.
Applications: classification of seismic events, estimation of numerical parameters, and analysis of tabular scientific data.

1D) Deep Learning Convolutional Neural Networks:

Convolutional Neural Networks detect local patterns by applying trainable filters across the input data.

1D-CNNs are designed for time series and waveforms.
Applications: classification of seismic signals, event detection, and automatic picking of P- and S-wave arrivals, as in PhaseNet.

2D-CNNs are primarily applied to images, maps, and spectrograms.
Applications: image classification, object detection, and segmentation of areas in satellite or geophysical images.

1E) Deep Learning recurrent and sequence networks: Long Short-Term Memory

LSTM networks process sequential data while preserving information from previous time steps. They are useful when the order and temporal context of observations are important.
Applications: time-series forecasting, classification of signal sequences, and prediction of ground deformation or environmental variables.

1F) Classical and denoising autoencoders

An autoencoder learns to compress input data into a latent representation and then reconstruct the original input. A denoising autoencoder is trained to reconstruct clean data from a corrupted version.
Applications: anomaly detection in continuous seismic records, signal denoising, nonlinear dimensionality reduction, and feature extraction.

Category 2. Emerging techniques with high research interest and growing application

This category represents the vibrant front lines of geophysical data science as of 2026. These methodologies - many adapted from recent breakthroughs in computer vision and Natural Language Processing (NLP) – are rapidly transitioning from experimental concepts to mainstream research tools. Their adoption is largely driven by the shift toward self-supervised learning, allowing researchers to finally exploit massive archives of unlabeled continuous data.

2A) Deep Learning – 3D Convolutional Neural Networks (3D-CNN)

A 3D-CNN extends convolution to three spatial dimensions and can analyse complete volumetric structures rather than individual two-dimensional slices.

Applications: analysis of 3D seismic tomography, segmentation and classification of geological structures in seismic volumes, and estimation of subsurface properties.

2B) Attention-based and transformer networks (self-supervised learning)

Transformer networks use self-attention to identify relationships between different parts of an input sequence, including long-range dependencies. They can be trained using either supervised or self-supervised learning.
Transformers: modelling of long time series, multichannel observations, and complex spatial or temporal relationships.
Wav2Vec 2.0 and related models: self-supervised learning of signal representations from large collections of unlabelled waveforms. Models inspired by Wav2Vec 2.0 can be pretrained on continuous seismic records and subsequently fine-tuned for phase picking, event detection, or classification.
Large Language Models: large transformer-based models pretrained mainly on extensive text collections (Chat GPT, Gemini, Claude).

2C) Masked Autoencoder (MAE)

Masked autoencoders learn by hiding parts of the input and reconstructing the missing information. This allows models to learn useful representations from large unlabelled datasets.
Applications: self-supervised pretraining on seismic sections, spectrograms, or continuous waveforms, followed by fine-tuning on smaller labelled datasets for phase picking, event classification, anomaly detection, or signal reconstruction. This strategy can also provide the basis for domain-specific foundation models.

2D) Natural Language Processing (NLP)

Natural Language Processing enables AI systems to analyse information contained in reports, scientific publications, catalogues, news articles, and social media posts.

Category 3. High-potential artificial intelligence techniques not yet widely adopted

This category includes methods with considerable scientific potential whose applications in geophysics remain mainly experimental or research-oriented. Their wider adoption is still limited by computational cost, scalability, data requirements, and the need for further validation.

3A) Graph Neural Networks (GNN)

Graph Neural Networks process data represented as interconnected nodes and edges. A seismic network can naturally be represented as a graph in which stations form the nodes and their geographical, physical, or signal-based relationships form the edges.
Applications: integration of observations from multiple seismic stations, association of seismic phases, earthquake location and magnitude estimation, and analysis of spatial relationships within monitoring networks.

3B) Physics-informed and hybrid models

Hybrid models combine data-driven neural networks with physical equations, numerical simulations, or expert knowledge. Physics-Informed Neural Networks incorporate governing physical laws directly into the training process, usually as additional terms in the loss function.
Applications: modelling and inversion of seismic wave propagation, estimation of subsurface velocity or density fields, and reconstruction of wavefields from sparse observations. A PINN

constrained by the seismic wave equation may potentially estimate physically consistent values in regions with limited or no station coverage.

This handbook focuses exclusively on deep learning; consequently, classical supervised and unsupervised machine learning models are omitted from its scope. Designed as a curated, introductory guide, it focuses primarily on the most foundational – yet remarkably popular and versatile – deeplearning techniques: Multilayer Perceptrons (MLP), Convolutional Neural Networks (CNN), and Recurrent Neural Networks (RNN) for time-series analysis. The final section introduces the mechanics of the modern Wav2Vec 2.0 architecture, adapted from the Automatic Speech Recognition (ASR) family, showcasing how speech-processing paradigms translate to geophysical signal analysis.

It should be emphasized once again that this work is not intended to be a comprehensive textbook. It is a practical handbook for readers who are beginning their exploration of deep learning with Python 3. It introduces essential terminology, concepts, and model structures through simple explanations and practical examples. These foundations may serve as a starting point for further study, more advanced experimentation, and independent exploration of artificial intelligence.

2. LECTURE 1 – MULTILAYER PERCEPTRON

Deep Learning frameworks:

- **TensorFlow:**

TensorFlow is an open-source machine learning platform released by Google in 2015.
It provides tools and libraries for building, training, and deploying machine learning and deep learning models.
Over time, it has developed into an ecosystem for end-to-end machine learning workflows.

- **Keras:**

Keras is a high-level deep learning API (Application Programming Interface) designed to simplify model development and experimentation.
It is closely integrated with TensorFlow, and in its current version it also supports other backends, including PyTorch.

- **PyTorch:**

PyTorch is an open-source deep learning framework.
It is widely used because of its flexible and Python-friendly design.

ANN, FNN, MLP:

- **Artificial Neural Networks** (ANN) – broad group of computational models.
- One of their basic types – the **Feedforward Neural Network** (FNN), in which information moves only in one direction, from the input layer to the output layer.
- The **Multilayer Perceptron** (MLP) is a classical and widely used example of an FNN (Rumelhart et al. 1986; Haykin 2008; Chollet and Watson 2025).

MLP:

- The Multilayer Perceptron (MLP) helps in understanding more complex models, such as **Convolutional Neural Networks** (CNNs), **Recurrent Neural Networks** (RNNs), **Long Short-Term Memory** (LSTM) networks.

In many CNN-based image classification models, convolutional layers first extract features such as edges and shapes, and these features are then passed to one or more fully connected MLP layers to produce the final classification output.

- MLP can be used for a wide range of tasks, remains effective for **tabular data**, but it is usually less suitable for problems such as image analysis or sequence processing.

FNN versus MLP:

A Feedforward Neural Network (FNN) is usually classified as a Multilayer Perceptron (MLP) when it has the following main characteristics:

- **Multilayer architecture:** The network consists of at least three layers: an input layer, one or more hidden layers, and an output layer.

  In contrast, a single-layer perceptron contains only an input layer and an output layer.
- **Fully connected structure:** Each neuron in one layer is connected to all neurons in the next layer.
- **Non-linear activation in hidden layers:** The network uses non-linear activation functions in its hidden layers, such as ReLU, sigmoid, or tanh.

  If only linear functions were used, then the entire network, regardless of the number of hidden layers, would be equivalent to a **single linear transformation** and would lose the ability to learn complex patterns.

MLP network depth:

- When a Multilayer Perceptron (MLP) contains several hidden layers, it is commonly described as a **Deep Neural Network** (DNN).
- In contrast, a network with only one or a small number of hidden layers is often referred to as a **Shallow Neural Network**.
- If a network contains no hidden layers, so that the input is connected directly to the output layer, it is usually described as a **single-layer model**.
- Depending on the output function and the learning formulation, it may correspond to linear regression, logistic regression, or a classical single-layer perceptron.

Unique features of MLP:

- **Fully connected (dense) layers:** In a standard MLP architecture, each neuron in one layer is connected to all neurons in the next layer.
- **Backpropagation algorithm:** During training, backpropagation computes the gradients of the loss function with respect to the network parameters by propagating error information backward from the output layer through the hidden layers.

  The gradient in MLP networks is information about how much a change in each individual network parameter will affect the total error (loss) of the model.

  In practical implementations, such as model.fit(X_train, y_train) in Keras, this process is carried out automatically during training.

  The optimizer, for example Adam, then uses these gradients to update the weights and biases in order to reduce the loss in the following training steps.
- **Optimization algorithm (optimizer):** During neural network training, the main objective is to minimize the loss function by iteratively updating the model parameters.

  The optimizer is the algorithm that updates the weights and biases using the gradients computed during training.

  Its purpose is to reduce the loss function and improve model performance.

2.1 Example 1a

```
# MLP - MPG regression

"""
Script Summary:
This script demonstrates the implementation and comparison of a Multi-Layer Perceptron (MLP)
for regression tasks using two different libraries: scikit-learn and TensorFlow/Keras.

Workflow:
1. Loads the 'mpg' (Miles Per Gallon) dataset.
2. Preprocesses the data (train-test split, feature standardization).
3. Trains a baseline MLP model using scikit-learn with default hyperparameters.
4. Builds and trains an equivalent MLP model using Keras, for comparison of two simple MLP base-
lines
5. Evaluates both models using standard regression metrics: MSE (Mean Squared Error),
    MAE (Mean Absolute Error), and R2 (R-squared).
6. Compares the execution time and specific predictions from both frameworks.
"""

import time
import numpy as np
import pandas as pd
import seaborn as sns
import tensorflow as tf
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# ==========================================
# 1. DATA LOADING AND PREPARATION
# ==========================================
# Load the 'mpg' dataset and drop any rows with missing values
df = sns.load_dataset('mpg').dropna()

# Define features (X) and target variable (y)
X = df[['cylinders', 'displacement', 'horsepower', 'weight', 'acceleration']]
y = df['mpg']

# Split the dataset into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Scale the features using Standardization (mean=0, variance=1)
# Normalization (Min-Max Scaling): Normalization rescales all data points
# to fit within a fixed, tight range, typically between 0 and 1,
# which makes it highly sensitive to outliers.
# Standardization (Z-score Normalization): Standardization transforms the data
```

```
# so that it has a mean of zero and a standard deviation of one, making
# it much more robust when dealing with extreme values.

# Neural networks require scaled data to converge efficiently
# In the case of an MLP, more efficient convergence means that
# across successive epochs, the network reduces the error faster
# and more stably, because the weights are updated in a more
# well-behaved manner.

# Initialize the scaler object to define the standardization method
scaler = StandardScaler()
# Learn the scaling parameters strictly from the training data
# and apply the transformation to it simultaneously
X_train_scaled = scaler.fit_transform(X_train)
# Apply the memorized parameters to scale the test data,
# ensuring no future information leaks into the model
X_test_scaled = scaler.transform(X_test)

# These lines set random seeds in NumPy and TensorFlow
# to make the initialization and training of the MLP
# as reproducible as possible across runs.
np.random.seed(42)
tf.random.set_seed(42)

print("Starting training\n")

# ============================================
# 2. MODEL 1: SCIKIT-LEARN
# ============================================
# We use the default architecture (1 hidden layer, 100 neurons, ReLU activation,
# Adam optimizer, alpha/L2 penalty=0.0001).
# We only increase max_iter to 1000 to ensure the network has time to converge.
mlp_sklearn = MLPRegressor(max_iter=1000, random_state=42)

# Measure training time for scikit-learn
start_time_sk = time.time()
mlp_sklearn.fit(X_train_scaled, y_train)
time_sk = time.time() - start_time_sk

# Generate predictions and calculate evaluation metrics
sk_preds = mlp_sklearn.predict(X_test_scaled)
sk_mse = mean_squared_error(y_test, sk_preds)
sk_mae = mean_absolute_error(y_test, sk_preds)
sk_r2 = r2_score(y_test, sk_preds)

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

```
# ==========================================
# 3. MODEL 2: TENSORFLOW / KERAS (Matched Defaults)
# ==========================================

# 1. Initialize the Sequential model (a linear stack of layers)
model_keras = Sequential([
    # THE HIDDEN LAYER
    # - '64' is the number of neurons (nodes) in this layer. It's an arbitrary starting point.
    # - 'activation='relu'' tells neurons to pass only positive values forward (adds non-linearity).
     # - 'input_shape=(X_train.shape[1],)' tells the network exactly how many features (columns) to expect.
    Dense(64, activation='relu', input_shape=(X_train.shape[1],)),

    # THE OUTPUT LAYER
    # - '1' neuron because we are predicting a single continuous number (e.g., price).
    # - No activation function is specified here (technically 'linear'),
    #    which allows the network to output any real number (negative or positive).
    Dense(1)
])

# This line configures the model for training by assigning the Adam optimizer
# to efficiently update its weights and the Mean Squared Error (MSE) to measure
# and minimize its prediction mistakes
# optimizer="adam": It updates the network's weights during training to minimize the error.
# loss='mse': It calculates the average squared difference between predicted and actual values.
model_keras.compile(optimizer="adam", loss='mse')

# Measure training time for Keras
start_time_keras = time.time()

# Starts the training process and stores metrics for each epoch
history = model_keras.fit(
    X_train_scaled,
    y_train,
    # The number of times the model iterates over the entire dataset
    epochs=1000,          # Equivalent to Sklearn's max_iter=1000
)

time_keras = time.time() - start_time_keras

# Generate predictions and calculate evaluation metrics
# Flatten() is used to convert the 2D array of predictions back to a 1D array

# With, for example, 100 samples in X_test_scaled:
# Keras output: (100, 1) — a 2D array with 100 rows and 1 column.
# Scikit-learn expectation: (100,) — a 1D flat vector.
```

```
#Solution: Using .flatten() is the most efficient way
# to align the data for metrics like mean_squared_error.

keras_preds = model_keras.predict(X_test_scaled, verbose=0).flatten()
keras_mse = mean_squared_error(y_test, keras_preds)
keras_mae = mean_absolute_error(y_test, keras_preds)
keras_r2 = r2_score(y_test, keras_preds)

print(f"Scikit-Learn MSE:   {sk_mse:.2f}")
print(f"Scikit-Learn MAE:   {sk_mae:.2f}")
print(f"Scikit-Learn R2:    {sk_r2:.4f}")
print(f"Scikit-Learn Time: {time_sk:.3f} s")

print(f"Keras MSE:   {keras_mse:.2f}")
print(f"Keras MAE:   {keras_mae:.2f}")
print(f"Keras R2:    {keras_r2:.4f}")
print(f"Keras Time: {time_keras:.3f} s")

# ==========================================
# 4. PREDICTIONS COMPARISON
# ==========================================
print("\nPredictions comparison (first 5 examples):")
comparison_df = pd.DataFrame({
    'Actual MPG': y_test.values[:5].round(2),
    'Sklearn (pred)': sk_preds[:5].round(2),
    'Keras (pred)': keras_preds[:5].round(2)
})
print(comparison_df)
```

2.2 Building MLP – input

- The structure of the MLP input layer (Fig. 1) is determined by the specific **format of the data being processed**.
- Regardless of the initial data complexity, **all inputs must be converted into a single, one-dimensional (1D) vector** before reaching the first Dense layer.

Case 1 – Tabular data input structure:

- When working with tabular data, where each row consists of a fixed number of features, the data is already in a flattened format.
- In this scenario, the network is configured by specifying the exact number of features present in a single row.
- For example, for a dataset containing 12 feature columns, the architecture is defined as follows:

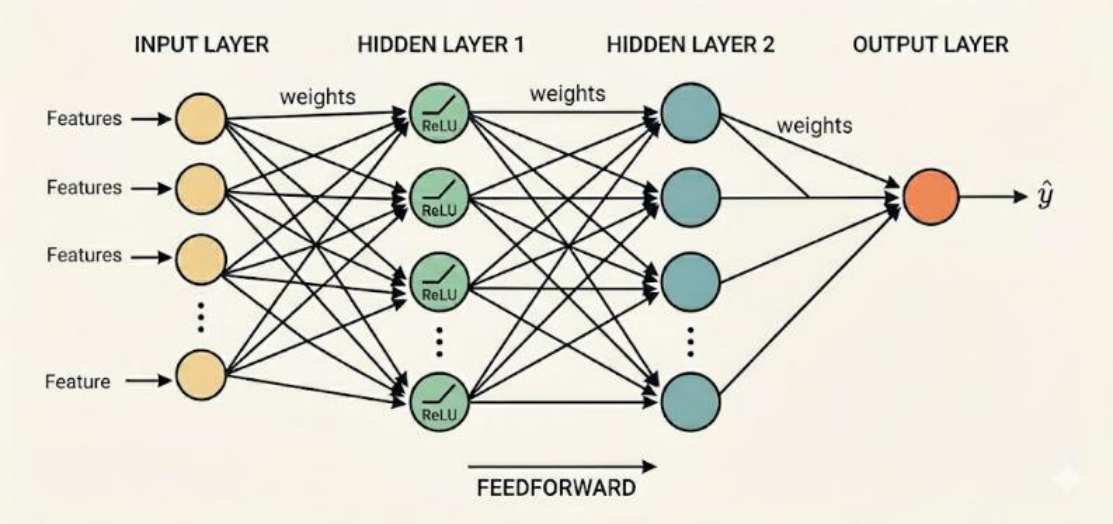


Fig. 1. Multilayer Perceptron (MLP) – input.

```
model = tf.keras.Sequential([
   # Input: 12 features from a table (already a flat 1D vector)
   tf.keras.layers.Input(shape=(12,)),

   # Dense layers can be connected directly to the input
   tf.keras.layers.Dense(64, activation='relu'),
   tf.keras.layers.Dense(1)
])
```

Case 2 – Time series input structure:

- This scenario applies to sequential data, such as sensor measurements over time.
- Consider a weather station that records three parameters daily: Temperature, Humidity, and Pressure.
- The objective is to predict tomorrow's temperature based on the data from the previous three days. Standard MLP architectures (using Dense layers) do not understand the dimension of time.
- Consequently, a 2D input (Time × Features) must be "flattened" into a single, one-dimensional vector.
- For a single training sample, the input vector would be organized as follows: [15.0, 60, 1012, 16.5, 55, 1010, 14.0, 70, 1008] – Indices 0–2: Day 1 (Temp, Hum, Pres), Indices 3–5: Day 2 (Temp, Hum, Pres), Indices 6–8: Day 3 (Temp, Hum, Pres).
- In Keras, the Flatten() layer is used to automatically perform this transformation, converting the (3, 3) matrix into a flat vector of 9 values before it enters the hidden layers:

```
model = tf.keras.Sequential([
   # Input: 3 days of history, each with 3 features (Time, Features)
   tf.keras.layers.Input(shape=(3, 3)),

   # The Flatten layer converts the (3,3) table into a 1D vector of 9 numbers
   tf.keras.layers.Flatten(),
```

```
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(1)
])
```

Case 3 – Image data input structure (pixels):

- Images require a specific transformation before they can be processed by a Multilayer Perceptron. An image is natively represented as a 2D grid (matrix) of pixels.
- For example, a grayscale image from the MNIST dataset has a resolution of 28 pixels in width and 28 pixels in height.
- Since standard MLP layers (Dense layers) cannot directly process 2D spatial structures, the image must be converted into a single continuous sequence of numbers.
- This process is known as Flattening.
- During this transformation, the rows of pixels are unrolled and joined together.
- Consequently, a 28×28 image is converted into a 1D vector consisting of 784 individual pixels (28×28 = 784).
- In Keras, this is implemented by defining the 2D input shape and immediately following it with a Flatten layer:

```
model = tf.keras.Sequential([
    # Define the 2D input shape (e.g., 28×28 pixels)
    tf.keras.layers.Input(shape=(28, 28)),

    # Flatten the 2D matrix into a 1D vector of 784 pixels
    tf.keras.layers.Flatten(),

    # Hidden layers are connected after the data is flattened
    tf.keras.layers.Dense(128, activation='relu')
])
```

Case 4 – Color image structure (RGB):

- Color images (such as those in the CIFAR–10 dataset) contain an additional dimension representing color channels.
- A standard color image consists of a grid of pixels with three layers: Red, Green, and Blue (RGB). For a resolution of 32×32 pixels, the input shape is defined as (32, 32, 3).
- When the Flatten layer is applied to this 3D tensor, it multiplies all dimensions (32×32×3 = 3072). Consequently, the image is converted into a long 1D vector consisting of 3072 individual numerical values.

```
model = tf.keras.Sequential([
    # Input: 32×32 pixel image with 3 color channels (RGB)
    tf.keras.layers.Input(shape=(32, 32, 3)),

    # Flattening converts the 3D structure into a 1D sequence of 3072 values
    tf.keras.layers.Flatten(),

    tf.keras.layers.Dense(256, activation='relu'),
```

```
    # Output layer for 10 categories (e.g., using Softmax for multiclass classification)
    tf.keras.layers.Dense(10, activation='softmax')
])
```

2.3 Building MLP – hidden layers

- In deep learning architectures, hidden layers are typically organized into repetitive blocks (Fig. 2).
- A standard computational block within an MLP is structured as follows:
  Dense → BatchNormalization → Activation → Dropout

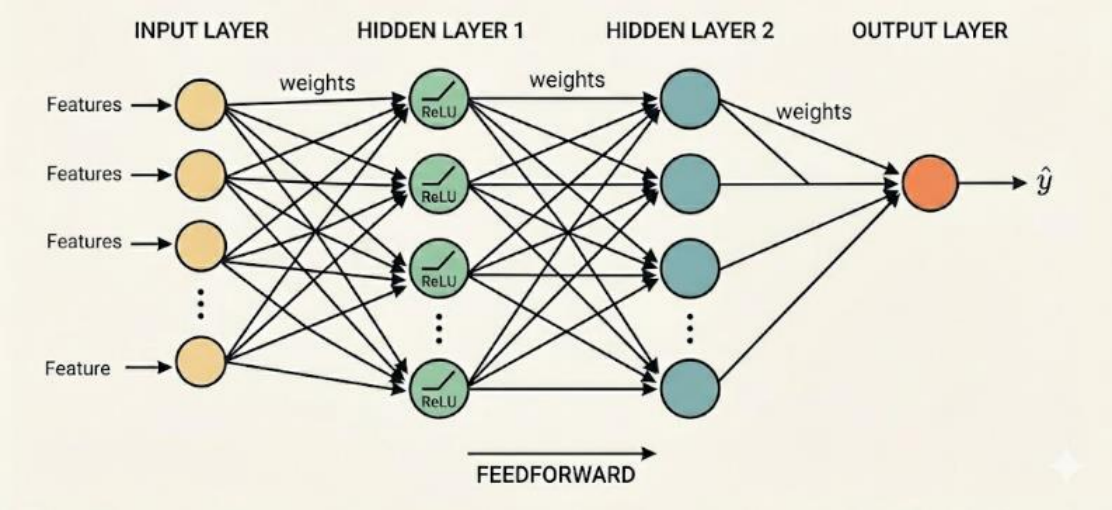


Fig. 2. Multilayer Perceptron (MLP) – architecture.

General principles for hidden layer design:

- **The power of two rule:** The number of neurons in each hidden layer should ideally be a power of two (e.g., 16, 32, 64, 128, 256, 512).
  This alignment corresponds to the memory architecture of modern CPUs and GPUs, allowing for optimized parallel processing and faster computation.
- **The funnel architecture:** This is one of the most effective structures for MLPs.
  It involves starting with a wider layer and progressively reducing the number of neurons in subsequent layers (e.g., 128 - -> 64 - -> 32).
  This design allows the network to capture a vast amount of fine-grained detail in the initial layers and synthesize it into broader, more abstract features toward the output.
- **Activation – ReLU:** For hidden layers, the Rectified Linear Unit (ReLU) is the definitive standard.
  It is computationally efficient and effectively mitigates the Vanishing Gradient problem, which can disturbe the training of deep networks.
- **The 2–3 layer rule:** For the majority of tasks involving tabular data (such as Excel or CSV files), two or three hidden layers are sufficient.
  Increasing the depth beyond this point often leads to unnecessary complexity and increases the risk of the model simply memorizing the training data.

- **Batch Normalization (stabilization):** Batch Normalization is highly recommended.
  As signals pass through multiple layers, their numerical values can fluctuate wildly.
  This layer "tempers" the data by centering it around zero after each step.
  The result is a more stable training process and significantly faster convergence.
- **Dropout (regularization):** Dropout is the primary tool against Overfitting.
  If a model achieves 99% accuracy on training data but performs poorly on new data, it has likely memorized the dataset.
  Dropout works by randomly "disabling" a fraction of neurons (e.g., Dropout(0.3) disables 30%) during each training iteration.
  This forces the remaining neurons to learn more robust and universal patterns.

A standard, high-performance block can be repeated 2–3 times, gradually decreasing the number of neurons:

```
model = tf.keras.Sequential([
   # Input layer (example for 10 features)
   tf.keras.layers.Input(shape=(10,)),

   # First Hidden Block
   tf.keras.layers.Dense(128),                # Core logic (Power of 2)
   tf.keras.layers.BatchNormalization(),      # Stabilization (Centers data)
   tf.keras.layers.Activation('relu'),        # Non-linear activation
   tf.keras.layers.Dropout(0.3),              # Regularization (Prevents overfitting)

   # Second Hidden Block (Funneling down)
   tf.keras.layers.Dense(64),
   tf.keras.layers.BatchNormalization(),
   tf.keras.layers.Activation('relu'),
   tf.keras.layers.Dropout(0.3),

   # Output layer
   tf.keras.layers.Dense(1)
])
```

MLP model development workflow:
It is highly recommended to follow an iterative development process:

- Baseline – a minimal architecture to establish a performance baseline, one or two Dense layers (e.g., 64 and 32 neurons) with ReLU activation, without complex regularization layers such as Dropout or BatchNormalization.
- After training begins, the training loss and validation loss should be monitored. **Underfitting:** If the training error does not decrease significantly, the model is likely "too simple" to capture the patterns in the data.
  Solution: Increase the network's capacity by adding more neurons to existing layers (e.g., transitioning from 64 to 128) or by introducing an additional hidden layer.
- If the MLP performs well on the training datset but clearly worse on the validation set, **overfitting** may be present.

Solution: One common option is **Dropout**, such as a Dropout(0.2) or Dropout(0.3), which randomly sets part of the layer output to zero during training and is widely used to help reduce overfitting.

Other possible actions include reducing model capacity, adding weight regularization, or using early stopping.

- Improve training behavior when needed.

  If the training process is excessively slow or the loss curve exhibits high volatility (large, erratic jumps), incorporate BatchNormalization.

  If MLP training is unstable or difficult to optimize, BatchNormalization may be considered. In TensorFlow/Keras, this layer normalizes its inputs during training.

  It can improve training behavior in some cases, but it should not be presented as a universal guarantee of faster or better convergence.

2.4 Building MLP – output

Configuring the MLP output layer:
The configuration of the output layer (Fig. 3) is a key part of MLP design because it should match the mathematical form of the prediction task.

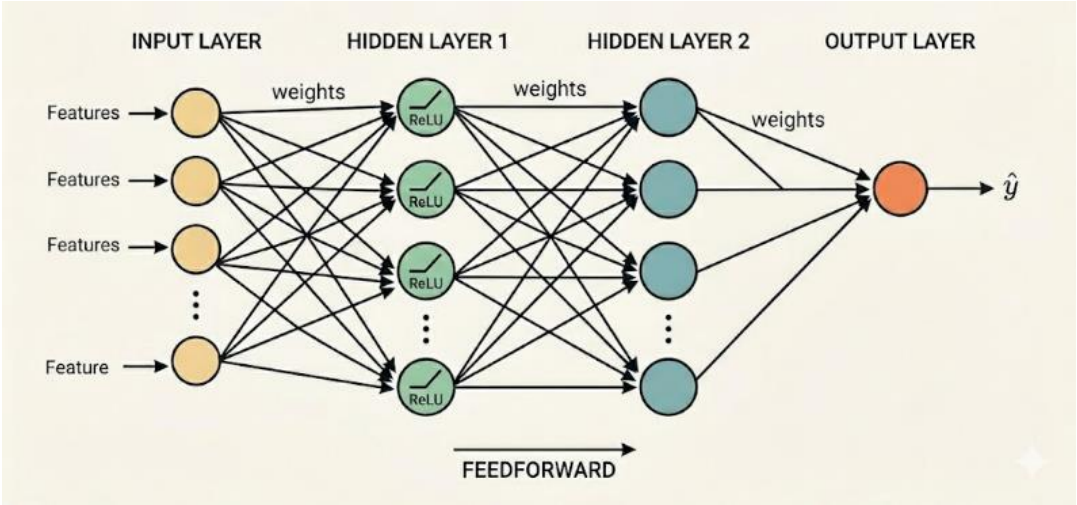


Fig. 3. Multilayer Perceptron (MLP) – architecture.

Case 1 – Regression (predicting continuous values):
Regression is used when the goal is to predict a continuous numerical value, such as price, age, or temperature.

- **Number of output neurons: One neuron** should be used when the model predicts a single continuous value.

  If several continuous values are predicted at the same time, the number of output neurons should match the number of target variables.

- **Activation function: A linear activation function** should be used.

  In Keras, this can be done by leaving the activation unspecified in the final Dense layer.

  This allows the model to produce any real-valued output.

- **Loss function: Mean Squared Error** (MSE) is commonly used for regression.

It gives greater weight to larger errors, which makes it suitable for many regression problems.

- **Metric: Mean Absolute Error** (MAE) is often used as an additional evaluation metric because it is easy to interpret.
  Its value is expressed in the same units as the target variable.

```
# Standard output configuration for regression
model.add(tf.keras.layers.Dense(1))

model.compile(
   optimizer='adam',
   loss='mse',      # Optimization target
   metrics=['mae']  # Human-readable evaluation
)
```

Case 2 – Binary classification (Yes/No, 0/1 Decision):

Binary classification is used when the objective is to predict one of two distinct categories.

- **Number of output neurons: One output neuron** should be used.
- **Activation Function: The sigmoid activation function** should be used.
  It maps the output to a value between 0 and 1.
  This output is interpreted as a probability (e.g., a result of 0.85 indicates an 85% probability that the input belongs to the positive class).
- **Loss Function** (for the Optimizer): **Binary Crossentropy** (binary_crossentropy).
  Binary cross-entropy is commonly used for binary classification.
  It is suitable for models that produce probabilistic outputs through a sigmoid activation function.
  This function is mathematically optimized to penalize the model heavily when it assigns a high probability to the wrong class, forcing the network to clearly separate the two categories.
- **Metric**: **Accuracy**.
  This is the most intuitive metric for classification, representing the percentage of total predictions that the model got exactly right.
  However, in imbalanced datasets, additional metrics such as precision, recall, or F1-score may also be needed.

```
# Standard output configuration for binary classification
model.add(tf.keras.layers.Dense(1, activation='sigmoid'))

model.compile(
   optimizer='adam',
   loss='binary_crossentropy', # Optimized for probability-based errors
   metrics=['accuracy']       # Measures the percentage of correct hits
)
```

Case 3 – Multiclass classification (multiple categories):
Multiclass classification is applied when the objective is to categorize an input into one of three or more classes (e.g., recognizing different species of animals or classifying handwritten digits).

- **Output Layer Neurons:** The final layer must contain exactly as many neurons as there are target categories (e.g., exactly 10 neurons for a task recognizing digits from 0 to 9).
- **Activation Function: Softmax**. The Softmax function takes the raw numerical outputs from all neurons and mathematically normalizes them into a probability distribution.

  It strictly enforces a rule where the sum of all output values equals exactly 1.0 (100%). Consequently, the network provides a probabilistic proportion for each class (e.g., predicting "70% Cat, 20% Dog, 10% Car").
- **Loss Function (for the Optimizer): Categorical Crossentropy** (categorical_crossentropy): Utilized when the target labels are one-hot encoded (e.g., represented as vectors like [0, 1, 0, 0]).

  Sparse Categorical Crossentropy (sparse_categorical_crossentropy): Utilized when the target labels are provided as standard integers (e.g., 0, 1, 2, 3).

  This is mathematically identical but highly memory-efficient.
- **Metric** (for Human Interpretation): **Accuracy**, which measures the overall percentage of correct predictions across all available categories.

  Although categorical_accuracy or sparse_categorical_accuracy may be more explicit, depending on the label format.

```
# Standard output configuration for multiclass classification (e.g., 4 categories)
model.add(tf.keras.layers.Dense(4, activation='softmax'))

model.compile(
   optimizer='adam',
   # Use 'sparse_categorical_crossentropy' if labels are integers (0, 1, 2, 3)
   loss='sparse_categorical_crossentropy',
   metrics=['accuracy']
)
```

Optimizer
An optimizer is an algorithm or method used to update the weights and biases (parameters) of a neural network during the training process. Its main goal is to minimize the loss function, which translates into increased model accuracy and faster convergence.

- **Adam (Adaptive Moment Estimation)** is currently the common standard for training deep neural networks.

  Its primary advantage lies in its ability to autonomously manage the learning rate (the "step size") during the training process.

  Its performance depends on the dataset, the MLP architecture, and the training setup.
- **The limitation of classic SGD:** In traditional optimization algorithms, such as standard Stochastic Gradient Descent (SGD), a single, global Learning Rate must be manually defined by the user.

  This rate applies uniformly to the entire network and often requires tedious manual tuning.

- **The Adam advantage:** Adam resolves this inefficiency by automatically and dynamically calculating an individual, adaptive step size for every single parameter (weight and bias) in the network.
  This results in significantly faster, more stable, and more reliable convergence to the optimal solution.

2.5 Example 1b

```
# MLP - regression

"""
Script Summary:
This script compares the performance of a classic Machine Learning algorithm (Random Forest)
against a Deep Learning approach (Multi-Layer Perceptron) for a regression task.

Workflow:
1. Loads the 'diamonds' dataset (predicting diamond prices based on physical attributes).
2. Preprocesses the data: handles categorical variables using One-Hot Encoding and
   scales features using Standardization.
3. Trains a baseline Random Forest Regressor.
4. Builds and trains a Neural Network using Keras, incorporating advanced techniques
   like Batch Normalization and Dropout to prevent overfitting.
5. Evaluates both models using standard regression metrics: R2, MAE, MSE, and RMSE.
6. Visualizes the comparison through bar charts, a training loss curve, and a scatter plot
   of Actual vs. Predicted values.
"""

import seaborn as sns
import pandas as pd
import numpy as np
import time
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error
from sklearn.ensemble import RandomForestRegressor

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam

# ========================================
# 1. DATA LOADING
# ========================================
# Load the 'diamonds' dataset and drop any missing values
diamonds = sns.load_dataset("diamonds").dropna()
```

```
# ==========================================
# 2. DATA PREPARATION
# ==========================================
# Define input features and the target variable we want to predict
features = ['carat', 'depth', 'table', 'x', 'y', 'z', 'cut', 'color', 'clarity']
target = 'price'

X = diamonds[features]
y = diamonds[target]

# One-hot encoding for categorical variables ('cut', 'color', 'clarity')
# Machine learning models require numerical input. This converts categories into binary columns (0 or 1).
# 'drop_first=True' prevents the dummy variable trap (multicollinearity).
X = pd.get_dummies(X, drop_first=True)

# Split the dataset into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Feature Standardization (scaling data to have mean=0 and variance=1)
# Highly recommended for Neural Networks to ensure stable and fast convergence.
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# List to store evaluation results for later comparison
results = []

print("--- Starting models training & evaluation ---\n")

# ==========================================
# 3. MODEL 1: RANDOM FOREST REGRESSOR
# ==========================================
start = time.time()
# n_estimators=200 means the forest consists of 200 individual decision trees.
# n_jobs=-1 uses all available CPU cores for faster training.
rf = RandomForestRegressor(n_estimators=200, random_state=42, n_jobs=-1)
rf.fit(X_train, y_train)

# Predictions and Metrics for Random Forest
y_pred_rf = rf.predict(X_test)
r2_rf = r2_score(y_test, y_pred_rf)
mae_rf = mean_absolute_error(y_test, y_pred_rf)
mse_rf = mean_squared_error(y_test, y_pred_rf)
```

```
rmse_rf = np.sqrt(mse_rf)
time_rf = time.time() - start

results.append(["Random Forest", r2_rf, mae_rf, mse_rf, rmse_rf, time_rf])

# ============================================
# 4. MODEL 2: MLP (Keras)
# ============================================
start = time.time()

# Activation function: It introduces non-linearity into the network,
# allowing it to model complex, non-linear relationships
# between inputs and outputs.

# ReLU: It filters out negative values by setting them to zero
# while keeping positive values unchanged,
# making training faster and more efficient.

# Linear Activation: The default for regression outputs,
# it passes the raw calculated value directly without any transformation.

# Vanishing Gradient Problem: A phenomenon during training where
# gradients become extremely small as they are propagated back
# to earlier layers, causing those layers to stop learning.

# Backpropagation bottleneck: It occurs because chain-rule multiplication
# of small derivatives (common in Sigmoid or Tanh) leads to an exponential
# decay of the error signal.

# ReLU Solution: By having a constant gradient of 1 for all positive inputs,
# ReLU prevents the signal from shrinking,
# enabling the training of deep architectures.

# ReLU is the standard activation function for hidden layers
# because it is simple, efficient, and usually works well in practice.
# Leaky ReLU is a modified version that allows a small negative output
# for negative inputs, which can help reduce the risk of the "dying ReLU"
# problem, in which some neurons output only zeros and stop updating
# effectively during training. However, since standard ReLU is easier
# to use and is often sufficient, it remains the default choice
# in many MLP applications.

nn = Sequential([
    # First hidden layer with 128 neurons
    Dense(128, activation='relu', input_shape=(X_train.shape[1],)),

    # Batch Normalization normalizes the activations of the previous layer at each batch,
```

```
    # which speeds up training and makes the network more stable.
    BatchNormalization(),

    # Dropout randomly sets 30% of input units to 0 at each update during training time.
    # This helps prevent overfitting (memorizing the training data).
    Dropout(0.3),

    # Second hidden layer with 64 neurons
    Dense(64, activation='relu'),
    Dropout(0.3),

    # Output layer for regression (1 continuous value: price)
    Dense(1)
])

nn.compile(optimizer=Adam(learning_rate=0.001), loss='mse')

# The batch_size defines the number of samples processed
# before the model's internal weights are updated,
# balancing training speed with memory usage.

# The verbose parameter controls the logging output level,
# where 0 is silent, 1 displays a progress bar,
# and 2 provides a one-line summary per epoch.

# validation_split automatically reserves a fraction
# of the training data to monitor the model's performance
# on unseen data during the training process,
# serving as a critical tool for detecting and preventing overfitting.

# validation_split=0.2 sets apart 20% of the training data to evaluate loss at the end of each epoch
history = nn.fit(
    X_train, y_train,
    epochs=100, batch_size=256, verbose=0, validation_split=0.2
)

# Predictions and Metrics for Neural Network
y_pred_nn = nn.predict(X_test, verbose=0).flatten()
r2_nn = r2_score(y_test, y_pred_nn)
mae_nn = mean_absolute_error(y_test, y_pred_nn)
mse_nn = mean_squared_error(y_test, y_pred_nn)
rmse_nn = np.sqrt(mse_nn)
time_nn = time.time() - start

results.append(["Neural Network", r2_nn, mae_nn, mse_nn, rmse_nn, time_nn])

# ========================================
```

```
# 5. RESULTS SUMMARY
# ==========================================
df_results = pd.DataFrame(results, columns=["Model", "R²", "MAE", "MSE", "RMSE", "Time [s]"])
print("Results Summary:\n")
print(df_results.to_string(index=False))

# ==========================================
# 6. VISUALIZATION: BAR CHARTS
# ==========================================
fig, ax = plt.subplots(1, 3, figsize=(16, 5))

# R2 Plot
ax[0].bar(df_results["Model"], df_results["R²"], color=["lightgreen", "lightblue"])
ax[0].set_title("R² Comparison (Higher is better)")
ax[0].set_ylim(0.9, 1.0) # Zooming in since both models perform very well

# MAE Plot (Replaced RMSE with MAE for visual variety, though both are valid)
ax[1].bar(df_results["Model"], df_results["MAE"], color=["lightgreen", "lightblue"])
ax[1].set_title("MAE Comparison [USD] (Lower is better)")

# Time Plot
ax[2].bar(df_results["Model"], df_results["Time [s]"], color=["lightgreen", "lightblue"])
ax[2].set_title("Computation Time [s] (Lower is better)")

plt.suptitle("Model Comparison: Random Forest vs MLP (Diamonds Dataset)", fontsize=14)
plt.tight_layout()
plt.show()

# ==========================================
# 7. VISUALIZATION: NN LEARNING CURVE
# ==========================================
plt.figure(figsize=(8, 5))
plt.plot(history.history['loss'], label='Training Loss (MSE)')
plt.plot(history.history['val_loss'], label='Validation Loss (MSE)')
plt.title("Neural Network Training History")
plt.xlabel("Epoch")
plt.ylabel("Loss (MSE)")
plt.legend()
plt.grid(True, linestyle='--', alpha=0.6)
plt.show()

# ==========================================
# 8. VISUALIZATION: ACTUAL VS PREDICTED
# ==========================================
plt.figure(figsize=(8, 8))
# Plot Random Forest predictions
plt.scatter(y_test, y_pred_rf, alpha=0.3, label=f"Random Forest (R²={r2_rf:.3f})", color="green")
```

```
# Plot Neural Network predictions
plt.scatter(y_test, y_pred_nn, alpha=0.3, label=f"Neural Net (R²={r2_nn:.3f})", color="blue")

# Plot the perfect prediction line (y = x)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', lw=2, label="Perfect Prediction")

plt.xlabel("Actual Price [USD]")
plt.ylabel("Predicted Price [USD]")
plt.title("Prediction vs Reality: Random Forest vs Neural Network")
plt.legend()
plt.grid(True, linestyle='--', alpha=0.6)
plt.show()
```

2.6 Example 1c

```
# MLP - penguins classification

"""
WORKFLOW:
1. DATA ACQUISITION: Load the 'penguins' dataset and perform initial cleaning (dropping NaNs).
2. FEATURE ENGINEERING: Transform categorical predictors via One-Hot Encoding and
   map target species to integer labels.
3. DATA PARTITIONING: Execute a stratified Train/Test split to maintain original
   class proportions in both subsets.
4. FEATURE SCALING: Apply StandardScaler to normalize numerical inputs, ensuring
   stable gradients and faster convergence for the Neural Network.
5. BASELINE MODELING: Train a Random Forest Classifier to establish a performance
   benchmark for the specific dataset.
6. DEEP LEARNING ARCHITECTURE: Construct an advanced Sequential MLP utilizing
   Batch Normalization, L2 Regularization, and hybrid activation functions (LeakyReLU/SELU).
7. OPTIMIZED TRAINING: Compile with multi-faceted metrics (F1, Precision, Recall)
   and implement EarlyStopping to prevent overfitting.
8. COMPARATIVE ANALYSIS: Evaluate both models using Confusion Matrices and
   macro-averaged F1-Scores to determine the most robust classifier.
"""

import seaborn as sns
import pandas as pd
import numpy as np
import time
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder, label_binarize
from sklearn.metrics import (
    accuracy_score, f1_score, confusion_matrix, ConfusionMatrixDisplay,
```

```
    roc_curve, auc
)
from sklearn.ensemble import RandomForestClassifier

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Dense, Dropout, BatchNormalization, LeakyReLU
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.regularizers import l2
from tensorflow.keras.metrics import Precision, Recall, F1Score
from tensorflow.keras.callbacks import EarlyStopping

# ==========================================
# 1. DATA LOADING & PREPROCESSING
# ==========================================
# Load the dataset and drop rows with missing values to ensure data integrity
penguins = sns.load_dataset("penguins").dropna()

X = penguins.drop("species", axis=1)
y = penguins["species"]

# One-hot encoding for categorical features (e.g., island, sex) to make them readable for the model
X = pd.get_dummies(X, drop_first=True)

# Target encoding: Convert species names to integers (0, 1, 2)
label_enc = LabelEncoder()
y_encoded = label_enc.fit_transform(y)

# Convert integers to one-hot encoded vectors for the Neural Network output layer
y_cat = to_categorical(y_encoded)
n_classes = y_cat.shape[1]

# ==========================================
# 2. TRAIN / TEST SPLIT & SCALING
# ==========================================
# Stratify ensures the class distribution remains consistent across train and test sets
X_train_raw, X_test_raw, y_train, y_test = train_test_split(
    X, y_encoded, test_size=0.2, random_state=42, stratify=y_encoded
)

# Scaling is crucial for Neural Networks to ensure faster convergence and prevent gradient issues
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train_raw)
X_test = scaler.transform(X_test_raw)

results = []
```

```
# ==========================================
# 3. MODEL 1: RANDOM FOREST (Baseline)
# ==========================================
print("--- Training Random Forest ---")
start = time.time()
rf = RandomForestClassifier(n_estimators=200, random_state=42)
rf.fit(X_train, y_train)

y_pred_rf = rf.predict(X_test)
y_proba_rf = rf.predict_proba(X_test)

results.append(["Random Forest", accuracy_score(y_test, y_pred_rf),
                f1_score(y_test, y_pred_rf, average='macro'), time.time() - start])

# ==========================================
# 4. MODEL 2: ADVANCED NEURAL NETWORK (MLP)
# ==========================================
print("--- Training Neural Network ---")
start_nn = time.time()

input_dim = X_train.shape[1]

model = Sequential([
    # Input layer defining the feature space:
    # Sets the expected dimensionality for incoming data records.
    Input(shape=(input_dim,)),

    # --- Block 1: Feature Extraction & High Capacity ---
    # Dense: 256 neurons to capture complex, non-linear relationships.
     # kernel_initializer='he_normal': Optimized for ReLU-family activations to maintain stable variance.
    # kernel_regularizer=l2(0.001): Adds a penalty to large weights to simplify the model and reduce overfitting.
    Dense(256, kernel_initializer='he_normal', kernel_regularizer=l2(0.001)),
    # BatchNormalization: Normalizes layer outputs to speed up training and provide internal stability.
    BatchNormalization(),
     # LeakyReLU: Mitigates the 'dying ReLU' problem by allowing a small gradient for negative inputs (alpha=0.1).
    LeakyReLU(alpha=0.1),
     # Dropout: Randomly zeros 30% of activations to prevent co-adaptation of neurons (improves generalization).
    Dropout(0.3),

    # --- Block 2: Dimensionality Reduction (The Funnel) ---
    # Reducing neurons to 128 to distill the most important features.
    Dense(128, kernel_initializer='he_normal', kernel_regularizer=l2(0.001)),
```

```
    BatchNormalization(),
    LeakyReLU(alpha=0.1),
    Dropout(0.3),

    # --- Block 3: Information Consolidation ---
    # Further narrowing the network to 64 neurons to focus on abstract patterns.
    Dense(64, kernel_initializer='he_normal'),
    BatchNormalization(),
    LeakyReLU(alpha=0.1),
    Dropout(0.2),

    # --- Block 4: Final Refinement ---
    # SELU (Scaled Exponential Linear Unit): Provides self-normalizing properties.
     # lecun_normal: The mathematically required initializer for SELU to ensure proper output distribution.
    Dense(32, activation='selu', kernel_initializer='lecun_normal'),

    # --- Output Layer ---
    # Softmax: Normalizes the final layer outputs into a probability distribution summing to 1.0.
    # Each value represents the model's confidence for a specific class.
    Dense(n_classes, activation='softmax')
])

# Compilation using the Adam optimizer with a custom learning rate
model.compile(
    optimizer=Adam(learning_rate=0.001),
    loss='categorical_crossentropy',
    metrics=[
        'accuracy',
        Precision(name='precision'),
        Recall(name='recall'),
        F1Score(average='macro', name='f1_score')
    ]
)

# EarlyStopping prevents the model from learning noise by monitoring validation loss
early_stop = EarlyStopping(
    # Monitor validation loss to detect the exact point where the model starts overfitting
    monitor='val_loss',

    # Number of epochs to wait for improvement before terminating the training process
    patience=15,

    # Crucial: ensures the model reverts to its optimal state (best weights)
    # instead of keeping the weights from the final (potentially overfitted) epoch
    restore_best_weights=True,
```

```
    # Log a message in the console when the training is prematurely stopped
    verbose=1
)

# Training execution
history = model.fit(
    X_train, to_categorical(y_train),
    epochs=1000,
    batch_size=32,
    validation_split=0.2,
    callbacks=[early_stop],
    verbose=1 # Set to 1 for per-epoch logs
)

# Neural Network Predictions
y_proba_nn = model.predict(X_test, verbose=0)
y_pred_nn = np.argmax(y_proba_nn, axis=1)

results.append(["Neural Network", accuracy_score(y_test, y_pred_nn),
                f1_score(y_test, y_pred_nn, average='macro'),
                time.time() - start_nn])

# ==========================================
# 5. RESULTS SUMMARY & VISUALIZATION
# ==========================================
df_results = pd.DataFrame(results,
                          columns=["Model", "Accuracy",
                                   "F1-Score (Macro)", "Time [s]"])
print("\n Results Summary:\n", df_results.to_string(index=False))

# Visualization: Confusion Matrices
fig, ax = plt.subplots(1, 2, figsize=(14, 5))
ConfusionMatrixDisplay.from_predictions(y_test, y_pred_rf,
                                        display_labels=label_enc.classes_,
                                        ax=ax[0], cmap="Greens")
ax[0].set_title("Random Forest Matrix")
ConfusionMatrixDisplay.from_predictions(y_test, y_pred_nn,
                                        display_labels=label_enc.classes_,
                                        ax=ax[1], cmap="Blues")
ax[1].set_title("MLP Matrix")
plt.show()

# ==========================================
# 6. VISUALIZATION: LEARNING CURVES
# ==========================================

def plot_learning_curves(history):
```

```
    """
    Visualizes the training and validation loss/accuracy over epochs.
    Useful for detecting overfitting or underfitting.
    """
    acc = history.history['accuracy']
    val_acc = history.history['val_accuracy']
    loss = history.history['loss']
    val_loss = history.history['val_loss']
    epochs_range = range(1, len(acc) + 1)

    plt.figure(figsize=(12, 5))

    # Plot Training & Validation Loss
    plt.subplot(1, 2, 1)
    plt.plot(epochs_range, loss, label='Training Loss', color='red')
    plt.plot(epochs_range, val_loss, label='Validation Loss', color='orange')
    plt.title('Training and Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss Value')
    plt.legend(loc='upper right')
    plt.grid(True, linestyle='--', alpha=0.6)

    # Plot Training & Validation Accuracy
    plt.subplot(1, 2, 2)
    plt.plot(epochs_range, acc, label='Training Accuracy', color='blue')
    plt.plot(epochs_range, val_acc, label='Validation Accuracy', color='green')
    plt.title('Training and Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend(loc='lower right')
    plt.grid(True, linestyle='--', alpha=0.6)

    plt.tight_layout()
    plt.show()

# Call the function to display charts
plot_learning_curves(history)
```

Exemplary prompt:

1) Load the “diamonds” from Seaborn dataset.
2) Preprocesses the data using Standardization.
3) Build and train a MLP for regression with two hidden layers, following the Sequential structure.
4) Use Batch Normalization and Dropout to prevent overfitting.
5) Show training history in the picture.
6) Print the model results using standard regression metrics.

2.7 MLP – summary

- A Multilayer Perceptron (MLP) consists of an **input layer**, one or more **hidden layers**, and an **output layer** (Fig. 4).
- In each layer, a neuron receives input values, computes their **weighted sum**, and adds a **bias term**.
- In the hidden layers, this value is then passed through a non-linear **activation function**, such as **ReLU**.
- The resulting outputs are passed from one layer to the next until the final prediction is produced.
- The activation used in the output layer depends on the task.
- At the beginning of training, the weights and biases are usually initialized with starting values.
- The network output is then compared with the expected target output, and the difference is measured by a **loss function**.
- Training is an iterative process in which **backpropagation** computes the **gradients** of the **loss function** with respect to the model parameters, and the **optimizer** updates the weights and biases in order to reduce the loss.
- The goal of training is not simply to minimize the training loss as much as possible, but to obtain a model that also generalizes well to unseen data.
- Otherwise, the network may become overfitted to the training set.
- **Activation functions in hidden layers:**

  Without activation functions, forward propagation would consist only of linear operations, such as matrix multiplication and addition.

  Activation functions transform these results and introduce non-linearity into the model.

  This non-linearity allows the MLP to learn complex relationships in the data.

  The choice of activation function affects training behavior and model performance.

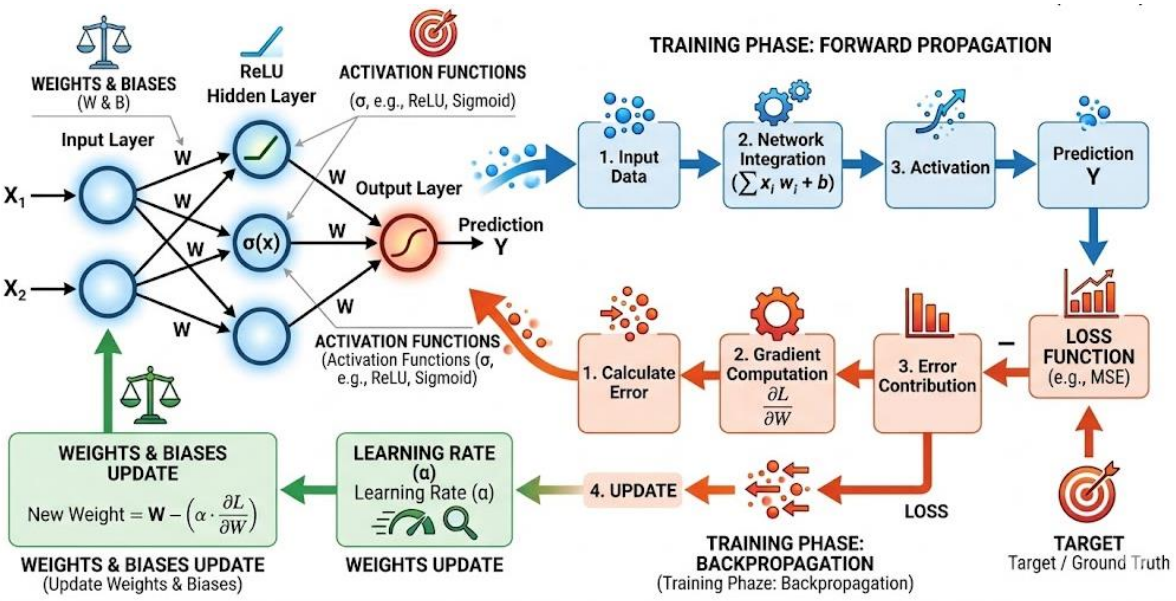


Fig. 4. Operational schematic of a Multilayer Perceptron.

If non-linear activation functions are omitted from the hidden layers, the network becomes mathematically equivalent to a single linear transformation, regardless of the number of hidden layers.

Common activation functions (Fig. 5): Sigmoid, Tanh, ReLU, Leaky ReLU…

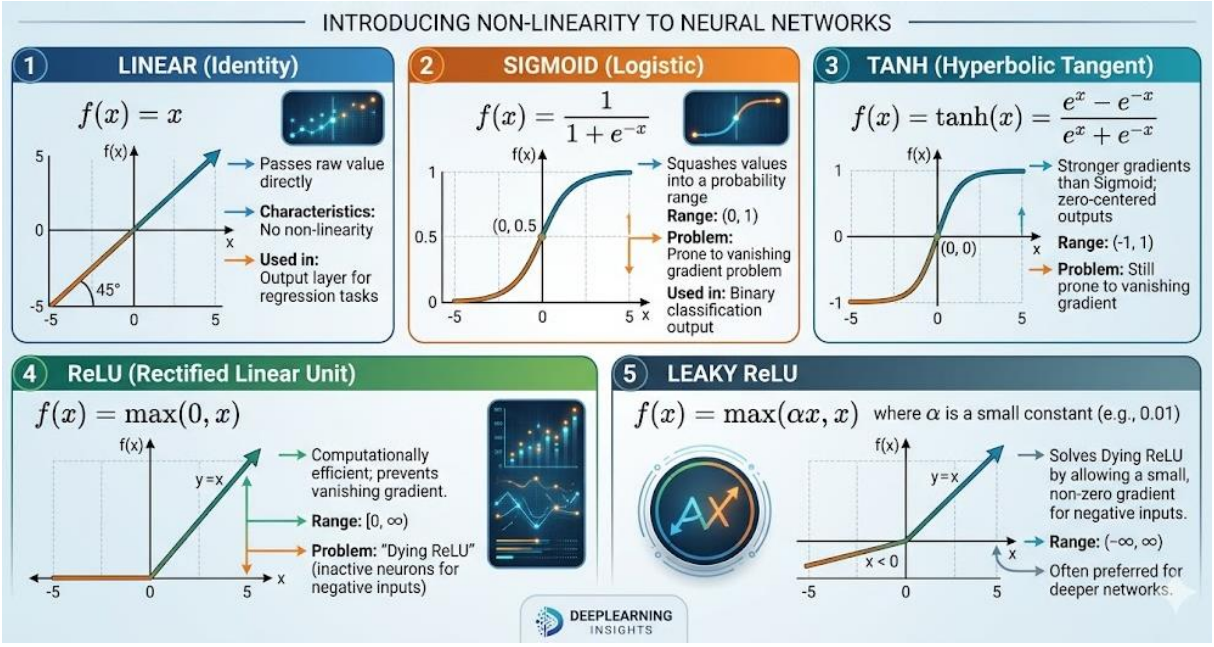


Fig. 5. Mathematical comparison of common activation functions.

- **Linear:** passes the input directly to the output without any transformation.

  It is rarely used in hidden layers because it cannot help the network learn complex, non-linear patterns.
- **Sigmoid:** maps any input value to a range between 0 and 1.

  It is traditionally used for binary classification, but it can cause "vanishing gradients" in deep networks because the function flattens out at extreme values.
- **Tanh (Hyperbolic Tangent):** scales inputs to a range between –1 and 1.

  Being zero-centered, it often helps the model converge faster than sigmoid, though it still suffers from the vanishing gradient problem in very deep architectures.
- **ReLU (Rectified Linear Unit):** the standard; it outputs the input if it is positive and zero otherwise.

  It is computationally efficient and helps mitigate the vanishing gradient problem, though it can lead to "dead neurons".
- **Leaky ReLU:** A variation of ReLU that allows a small, non-zero gradient when the input is negative.

  This prevents neurons from "dying" by ensuring that they continue to learn even when they receive negative values.
- **Output layer activation functions:**

  The choice of the activation function in the output layer depends on the task performed by the neural network.

Linear (no activation): This output is commonly used in regression tasks, where the goal is to predict continuous numerical values.

In this case, the network produces an unconstrained numerical output.

Sigmoid: This activation is commonly used in binary classification.

It maps the output to a value between 0 and 1, which can be interpreted as the probability of one class.

Softmax: This activation is typically used in multiclass classification problems with mutually exclusive classes.

It transforms the raw outputs into a probability distribution over all classes.

If there are N classes, the output contains N values, and their sum equals 1.

- **Quantifying error:**

  Loss and Cost Functions in MLPs. Once the network produces a prediction, the difference between the predicted output and the target output must be measured.

  Loss function: This usually refers to the error calculated for a single training example.

  Cost function: This usually refers to the aggregated error over a batch or over the full training set, often computed as the mean or sum of individual loss values.

  The loss defines the quantity whose gradients are computed during backpropagation, and these gradients are then used to update the model parameters.

- **Learning rate:**

  While backpropagation computes the gradients used for parameter updates, the learning rate is the hyperparameter that determines the step size of these updates.

  High learning rate: The updates may become too large, which can cause overshooting, unstable training, or failure to converge.

  Low learning rate: The updates become very small.

  In this case, training may still converge, but the process is usually much slower and may require many more iterations.

- **Epochs and iterative learning:**

  An epoch is one complete pass through the full training dataset.

  Neural networks learn iteratively, through many small parameter updates, and therefore multiple epochs are usually needed to obtain well-performing parameter values.

  In Keras, this is controlled by the epochs argument in model.fit(...).

- **Overfitting and early stopping:**

  Training for too many epochs may lead to overfitting, meaning that the model fits the training data well but generalizes poorly to unseen data.

  To reduce this risk, early stopping can be used. In this approach, training is stopped when the monitored validation metric no longer improves.

2.8 Sequential architecture and others

Case 1 – Model building methodologies in Keras: The Sequential API

In Keras, three main approaches can be used to define and build neural network models:

- the **Sequential** API.
- the **Functional** API.
- model subclassing.

For standard MLP, the most basic and commonly used approach is the Sequential API.

The Sequential API is the simplest way to build a neural network model in Keras.
It is designed for models in which the layers are arranged in a single linear stack, with data passing through them in one direction from input to output.
It is based on a simple sequence of layers.
It is suitable for models with one input and one output and does not support more complex topologies such as multiple inputs, or multiple outputs.
It is a good choice when the model follows a simple linear structure:
Single input → Hidden layers → Single output

```
import tensorflow as tf
from tensorflow.keras import layers

# Building a simple MLP with the Sequential API
model = tf.keras.Sequential([
   layers.Input(shape=(10,)),          # Input layer
   layers.Dense(128, activation='relu'), # Hidden layer 1
   layers.Dense(64, activation='relu'),  # Hidden layer 2
   layers.Dense(1)                     # Output layer
])
```

Case 2 – The Functional API
The Functional API is a more flexible way to build neural network models in Keras.
Unlike the Sequential API, it does not require the model to follow a simple linear stack of layers. Instead, each layer is applied to an input tensor and produces an output tensor, which can then be passed to the next layer or to multiple layers.
It supports more complex model topologies than the Sequential API.
This includes architectures with multiple inputs, or multiple outputs.

```
import tensorflow as tf
from tensorflow.keras import layers

# 1. Define the input tensor
inputs = tf.keras.Input(shape=(10,))

# 2. Apply layers to tensors
x = layers.Dense(128, activation='relu')(inputs)
x = layers.Dense(64, activation='relu')(x)

# 3. Define the output tensor
outputs = layers.Dense(1)(x)

# 4. Create the model from inputs and outputs
model = tf.keras.Model(inputs=inputs, outputs=outputs)
```

2.9 Example 1d

```
# MLP - MNIST classification with FUNCTIONAL

import tensorflow as tf
from matplotlib import pyplot as plt
import numpy as np
from sklearn.metrics import classification_report

# ==========================================
# 1. DATA LOADING AND PREPARATION
# ==========================================
objects = tf.keras.datasets.mnist
(training_images, training_labels), (test_images, test_labels) = objects.load_data()

training_images = training_images / 255.0
test_images = test_images / 255.0

# Drugi target: parity (0 = even, 1 = odd)
training_parity = (training_labels % 2).astype("float32")
test_parity = (test_labels % 2).astype("float32")

# ==========================================
# 2. FUNCTIONAL MODEL WITH TWO OUTPUTS
# ==========================================
inputs = tf.keras.Input(shape=(28, 28), name="image_input")

x = tf.keras.layers.Flatten(name="flatten")(inputs)
x = tf.keras.layers.Dense(128, activation="relu", name="shared_dense_1")(x)
x = tf.keras.layers.Dense(64, activation="relu", name="shared_dense_2")(x)

# Output 1: digit classification (0-9)
digit_output = tf.keras.layers.Dense(
    10, activation="softmax", name="digit_output"
)(x)

# Output 2: parity classification (even/odd)
parity_output = tf.keras.layers.Dense(
    1, activation="sigmoid", name="parity_output"
)(x)

model = tf.keras.Model(
    inputs=inputs,
    outputs=[digit_output, parity_output],
    name="mnist_multi_output_model"
)
```

```
# ============================================
# 3. COMPILATION
# ============================================
model.compile(
    optimizer="adam",
    loss={
        "digit_output": "sparse_categorical_crossentropy",
        "parity_output": "binary_crossentropy"
    },
    metrics={
        "digit_output": ["accuracy"],
        "parity_output": ["accuracy"]
    },
    loss_weights={
        "digit_output": 1.0,
        "parity_output": 0.3
    }
)

model.summary()

# ============================================
# 4. TRAINING
# ============================================
history = model.fit(
    training_images,
    {
        "digit_output": training_labels,
        "parity_output": training_parity
    },
    epochs=5,
    batch_size=32,
    validation_split=0.1
)

# ============================================
# 5. EVALUATION
# ============================================
results = model.evaluate(
    test_images,
    {
        "digit_output": test_labels,
        "parity_output": test_parity
    },
    verbose=0
)
```

```
print("\n--- Evaluation results ---")
for name, value in zip(model.metrics_names, results):
    print(f"{name}: {value:.4f}")

# ============================================
# 6. PREDICTIONS
# ============================================
digit_probs, parity_probs = model.predict(test_images, verbose=0)

digit_preds = np.argmax(digit_probs, axis=1)
parity_preds = (parity_probs.flatten() > 0.5).astype(int)

print("\n--- Classification report for digits ---")
print(classification_report(test_labels, digit_preds, digits=4))

print("\n--- Sample predictions ---")
idx = 0
print("Predicted digit:", digit_preds[idx])
print("Actual digit    :", test_labels[idx])
print("Predicted parity (0 even, 1 odd):", parity_preds[idx])
print("Actual parity   (0 even, 1 odd):", int(test_parity[idx]))
```

Case 3 – Model subclassing

Model subclassing is the most flexible way to define a model in Keras.
In this approach, a custom Python class is created by inheriting from tf.keras.Model.
This makes it possible to define both the model components and the forward pass explicitly.

Core Characteristics: The network's architecture is mechanically divided into two distinct methods:

- The __init__ method is used to define and instantiate the layers.
- The call method is used to define how the input data pass through these layers during the forward pass.

When to Use:
Model subclassing is useful when a model requires custom behavior, non-standard data flow, or more advanced control over the forward pass.

```
import tensorflow as tf
from tensorflow.keras import layers

# Building the exact same MLP using the Subclassing API
class MyCustomMLP(tf.keras.Model):

 def __init__(self):
      super().__init__()
      # 1. Define the building blocks (instantiate layers)
```

```
        self.hidden_1 = layers.Dense(128, activation='relu')
        self.hidden_2 = layers.Dense(64, activation='relu')
        self.output_layer = layers.Dense(1)

    def call(self, inputs):
        # 2. Define the explicit data flow (the forward pass logic)
        x = self.hidden_1(inputs)
        x = self.hidden_2(x)
        return self.output_layer(x)

# Instantiate the model
model = MyCustomMLP()
```

3. LECTURE 2 – RECURRENT NEURAL NETWORKS AND TIMES SERIES

3.1 MLP vs RNN vs LSTM

- **The Multi-Layer Perceptron (MLP):** classic, standard feedforward neural network. It operates **without any concept of time or sequential order**.
- **Recurrent Neural Network (RNN):** designed to address the memory limitations of MLPs, providing short-term memory capabilities (Hochreiter and Schmidhuber 1997; Goldberg 2017).

  Highly effective for processing sequential data (e.g., time series, audio).

  A **feedback loop** is introduced in a recurrent network.

  When an RNN processes step 2 in a sequence, it considers both the data from step 2 and its own output from step 1.

  The information goes back, which is where the term "recurrent" comes from.

  This means that a neuron in an RNN receives two inputs simultaneously: the new, current data and its own internal state from the previous step.
- **The concept of the hidden state:** a numeric vector that functions as the internal working memory of the network.

  In a traditional neural network, input data is received, multiplied by weights, and immediately generates an output.
- **In an RNN, this process is as follows:**

  Step 0: Before the network processes any data, its hidden state is typically empty (initialized as a vector of zeros).

  Step 1: The network receives the initial data and combines this new input with the empty hidden state. Based on this, it generates a new hidden state.

  Step 2: The network receives the next data input and combines it with the previous hidden state. This cycle repeats until the end of the sequence.
- **Standard RNNs suffer from a critical structural limitation: the vanishing gradient problem.**

  This is caused by the repeated multiplication of small values during the backpropagation process, causing the error signal to diminish and vanish before it can reach the start of the sequence.

This means that when processing long sequences, the network quickly fails to retain information from the earliest steps.

- **Long Short-Term Memory (LSTM):** specifically designed to retain information over extended periods.

  It achieves this by modifying the internal structure of the standard neuron, which in an LSTM is referred to as a cell.

- **LSTM uses a system of gates to control the flow of information:**

  Forget gate: Determines which past information is irrelevant and should be discarded from memory.

  Input gate: Determines which new information from the current step is important and should be stored.

  Output gate: Determines what portion of the current memory state should be transmitted to the next step in the network.

- **An LSTM utilizes two independent information paths:**

  Hidden state: This functions as the short-term memory, similar to a standard RNN.

  It is used to analyze the immediate context and transmit the final output from the cell.

  Cell state: This acts as the long-term memory archive.

  To manage this archive, a single LSTM cell (neuron) incorporates three gates.

  These gates use the Sigmoid activation function, which compresses numerical values into a range between 0 and 1.

  A value approaching 0 means "close the gate and block the information", while a value approaching 1 means "open the gate and allow the information to pass".

- **Practical guidelines for building LSTM networks:**

  Data order is critical: The standard and correct approach to data splitting is chronological:

  Training set: The oldest data.

  Validation set: The subsequent data.

  Test set: The most recent data.

  Data scaling (normalization): LSTMs utilize activation functions like sigmoid and tanh within their cells, which operate optimally on small values (typically in the range of 0 to 1 or –1 to 1).

  It is essential to fit the scaler exclusively on the training data to prevent data leakage.

  Sequence creation (sliding window): LSTMs do not accept single rows of data.

  The dataset must be transformed into a three-dimensional format: [samples, time steps, features].

  For example, given a time series of [10, 20, 30, 40, 50] and a "lookback" window of 3 days to predict day 4, the data is structured as follows:

  Input (X): [10, 20, 30], Output (y): [40]

  Input (X): [20, 30, 40], Output (y): [50]

  The naive forecast illusion: the prediction graph appears to align perfectly with the actual data.

  It often becomes apparent that the model's predictions are shifted exactly one time step backward.

  This indicates that the network is simply projecting that the future value will be identical to the current value, which is a sign that the model is failing to learn the underlying trends.

Before implementing an LSTM model, **its performance should be compared against a simple baseline, such as the last observed value or the last seasonal value**.

Data leakage: This most commonly occurs when improperly scaling the entire dataset at once, prior to splitting it into the training and test sets.

Excessively long lookback windows: Providing the model with a history of, for instance, 1000 previous steps to predict the next occurrence rarely gives optimal results.

While time series forecasting is one of the most widely recognized applications of RNNs and LSTMs, **these networks were designed to process any type of sequential data.** Sequential data is defined simply as any dataset where the chronological or sequential order of the elements is of critical importance.

- **Building the network architecture – simple architecture (Keras/TensorFlow):**

  Input: LSTM networks require data in three dimensions.

  This shape is defined as:

  [batch_size, timesteps, features]

  Batch size: The number of sequences (windows) processed simultaneously before the model's weights are updated.

  When defining the input layer in Keras, this dimension is typically omitted, as the framework handles it automatically.

  Timesteps (sequence length / window size): The number of past steps the network analyzes. For instance, if the lookback window is set to 30 days, then timesteps = 30.

  Features: The number of distinct variables provided at each time step.

  For example, if only temperature is used as an input, then features = 1.

```
from tensorflow import keras
from keras import layers


# The input is defined for a 30–day window, with one feature per day.
tmp = layers.Input(shape=(30, 1))


# For a multivariate model (e.g., a 30–day window with 5 distinct features), the input shape would be defined as shape=(30, 5).
```

Hidden layers: A single LSTM layer can be used, or multiple layers can be stacked (known as a Stacked LSTM).
return_sequences: This is a critical parameter that determines the output of the LSTM layer.
return_sequences=False (Default): The layer processes the entire time window (e.g., 30 days) and outputs only a single vector representing the final memory state after the entire sequence has been processed.
This configuration is typically used in the final LSTM layer before the output layer.
return_sequences=True: The layer processes the 30-day window and outputs a result for each of the 30 steps, returning the complete sequence.
This setting is required when connecting one LSTM layer to another, as subsequent LSTM layers also require sequential data as input.
It is also recommended to include a Dropout layer, which acts as a regularization mechanism to prevent overfitting.

```
# First LSTM layer: 64 units.
# return_sequences=True is required, as a subsequent LSTM layer follows.
x = layers.LSTM(units=64, return_sequences=True)(inputt)

# Randomly deactivating 20% of the neurons during the training phase.
x = layers.Dropout(0.2)(x)

# Second (and final) LSTM layer: 32 units.
# return_sequences=False (outputting only the final state), as the output is to be generated next.
x = layers.LSTM(units=32, return_sequences=False)(x)
```

Output: A standard fully connected layer, known as a Dense layer, is used for the output.
Single-step prediction (most common): A Dense(1) layer is used to forecast one step ahead.
Multi-step prediction (e.g., forecasting the next 7 days): A Dense(7) layer is used to provide multiple future data points simultaneously.

```
# The LSTM outputs are mapped to a single, specific numerical value (the prediction).
output = layers.Dense(units=1)(x)
```

3.2 Alternatives

Other alternatives to LSTM for time series analysis:

- **GRU (Gated Recurrent Unit):** It simplifies the internal structure by utilizing only two gates: the update gate and the reset gate. Unlike LSTM, GRU does not maintain a separate cell state; instead, all information is managed within the hidden state.

  Advantages: Due to fewer parameters, GRU is computationally more efficient and trains faster than LSTM.

  In many applications, GRU achieves accuracy levels comparable to LSTM, particularly on smaller datasets where a simpler model may reduce the risk of overfitting.
- **1D CNN (1D Convolutional Neural Network):** While traditionally associated with computer vision, 1D CNNs are highly effective for time series analysis. This is achieved by sliding convolutional filters along the temporal axis. Unlike LSTMs, which process data sequentially, 1D CNNs identify local patterns (such as sudden spikes or specific trends) in parallel across segments of the sequence.

  Advantages: 1D CNNs are significantly faster to train because they allow for parallel processing.

  Best use case: They are particularly effective at extracting high-level features from raw sensor data or long sequences before passing them to a recurrent layer.
- **Dilated CNN (Temporal Convolutional Network – TCN):** the TCN approach utilizes dilated convolutions, which allow the network to maintain a vast receptive field while using a relatively small number of layers.

By introducing gaps between the filter elements, the network can analyze long-term dependencies (spanning hundreds or thousands of steps) without the vanishing gradient issues typical of standard RNNs.

Advantages: TCNs offer more stable gradients and a flexible receptive field size.

For very long sequences where long-term historical context is critical, TCNs often outperform LSTMs in both training speed and predictive accuracy.

3.3 Example 2

```
import random
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import tensorflow as tf

from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_absolute_error, mean_squared_error

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (
    Input,
    LSTM,
    GRU,
    Conv1D,
    Dense,
    GlobalAveragePooling1D,
    Dropout
)
from tensorflow.keras.callbacks import EarlyStopping

# ============================================================
# 1. GLOBAL SETTINGS
# ============================================================
# System parameters and hyperparameters for the models.
SEED = 42                # Random seed for reproducibility
WINDOW = 12              # Sequence length (input time steps)
TRAIN_RATIO = 0.70       # Fraction of data used for training
VAL_RATIO = 0.15         # Fraction of data used for validation
EPOCHS = 200             # Maximum training iterations
BATCH_SIZE = 8           # Number of samples per gradient update

# Set random seeds across all libraries to ensure consistent results
random.seed(SEED)
np.random.seed(SEED)
tf.random.set_seed(SEED)

# ============================================================
```

```
# 2. DATA LOADING & RAW VISUALIZATION
# ============================================================
print("1. Loading and plotting raw data...")
flights = sns.load_dataset("flights")

# Extract passenger counts as a 1D numpy array of type float32
series = flights["passengers"].values.astype("float32")

# Plot the raw dataset before any processing
plt.figure(figsize=(10, 5))
plt.plot(series, color='blue', linewidth=2)
plt.title("Raw Flights Dataset (Monthly Passengers)")
plt.xlabel("Time Step (Months)")
plt.ylabel("Number of Passengers")
plt.grid(True)
plt.show()

# ============================================================
# 3. DATA SPLITTING & SCALING
# ============================================================
# Time series data must be split sequentially to prevent data leakage.
n = len(series)
train_end = int(n * TRAIN_RATIO)
val_end = int(n * (TRAIN_RATIO + VAL_RATIO))

# Split data. Notice the overlap (- WINDOW) for validation and test sets.
# This ensures we have enough past data to predict the first target in these sets.
train_series = series[:train_end]
val_series = series[train_end - WINDOW:val_end]
test_series = series[val_end - WINDOW:]

# Initialize the scaler to normalize data between 0 and 1
scaler = MinMaxScaler()

# Fit the scaler ONLY on the training data to prevent data leakage.
train_scaled = scaler.fit_transform(train_series.reshape(-1, 1))

# Transform validation and test sets using the training parameters
val_scaled = scaler.transform(val_series.reshape(-1, 1))
test_scaled = scaler.transform(test_series.reshape(-1, 1))

# ============================================================
# 4. DATA FORMATTING (WINDOW GENERATION)
# ============================================================
def make_windows(data, window):
    """
    Transforms a 1D time series array into supervised learning format.
```

```
    Creates input sequences (X) and target values (y).
    """
    X, y = [], []
    for i in range(len(data) - window):
        X.append(data[i:i + window, 0])   # Input sequence
        y.append(data[i + window, 0])     # Target value

    X = np.array(X, dtype=np.float32)
    y = np.array(y, dtype=np.float32)

    # Reshape X to 3D format required by Keras: (samples, time_steps, features)
    X = X.reshape((X.shape[0], X.shape[1], 1))
    return X, y

# Apply formatting to all data splits
X_train, y_train = make_windows(train_scaled, WINDOW)
X_val, y_val = make_windows(val_scaled, WINDOW)
X_test, y_test = make_windows(test_scaled, WINDOW)

# ============================================================
# 5. MODEL ARCHITECTURES
# ============================================================

def build_lstm():
    """
    LSTM (Long Short-Term Memory):
    A type of Recurrent Neural Network (RNN). It is highly effective
    for time series because it uses internal memory gates to retain
    important long-term information and discard irrelevant data.
    """

    #LSTM(32): LSTM maintains an internal memory. It uses structures called gates
    #to control the flow of information:
    #Forget Gate: Decides which historical information is no longer relevant
    #and should be discarded from memory.
    #Input Gate: Decides which new, incoming data is important enough
    #to be added to the long-term memory.
    #Output Gate: Determines what information should be passed forward
    #to the next layer based on the current input and the updated memory.

    #The number 32 defines the hidden state size (number of units or neurons)
    #of that specific LSTM layer.
    #A larger number (e.g., 64 or 128) allows the model to learn more complex
    #relationships but requires significantly more computational power and data.

    model = Sequential([
        Input(shape=(WINDOW, 1)),
```

```
        # recurrent_dropout helps prevent overfitting in RNNs
        LSTM(32, recurrent_dropout=0.2),
        Dense(1)
    ])
    model.compile(optimizer="adam", loss="mse", metrics=["mae"])
    return model

def build_gru():
    """
    GRU (Gated Recurrent Unit):
    A modified version of LSTM. It merges some memory gates,
    making the architecture simpler and computationally faster,
    often achieving similar performance to LSTM.
    """

    #LSTM (1997)vs GRU (2014): GRU utilizes only 2 gates (Reset and Update).
    #The Update gate efficiently handles the combined duties of the LSTM's
    #Forget and Input gates.Simpler architecture, requiring fewer mathematical
    #operations and yielding fewer trainable parameters.

    model = Sequential([
        Input(shape=(WINDOW, 1)),
        GRU(32),
        Dense(1)
    ])
    model.compile(optimizer="adam", loss="mse", metrics=["mae"])
    return model

def build_cnn1d():
    """
    1D CNN (1D Convolutional Neural Network):
    Uses convolutional filters that slide across the time steps
    to extract local patterns.
    GlobalAveragePooling1D is used to reduce spatial dimensions
    and prevent overfitting.
    """

    """
    # Conv1D(...): This line defines a 1-Dimensional Convolutional Layer,
    # originally popularized in image processing (2D CNNs) but adapted here
    # for sequential or time-series data. Instead of sliding across a 2D grid
    # of pixels, it slides forward along a 1D timeline to detect local patterns.

    # filters=32 (Number of Feature Detectors): tells the network to learn
    # 32 distinct filters (or masks). The layer will output 32 distinct feature
    # maps based on what these filters discover.
```

```
    # kernel_size=3 (The Sliding Window Size): a kernel_size of 3 means
    # the filter looks at 3 consecutive time steps (e.g., 3 months of data)
    # at any single moment. It calculates a weighted sum of these 3 values
    # before sliding one step forward to look at the next 3-month window.

    # padding="causal" (Preventing Data Leakage): standard padding might center
    # the window on the current data point, accidentally pulling in future
    # data points to calculate the present value (which is illegal in forecasting).
    # "Causal" padding forces the convolution to strictly respect the flow of time.
    # It pads the beginning of the sequence with zeros so that the filter
    # can only look at the current time step and past time steps.
    # It guarantees that the model never "peeks into the future" to make a prediction.
    """

    """
    # GlobalAveragePooling1D(): following a Conv1D layer with 32 filters
    # operating over a 12-step time window, the internal data representation
    # is a 2D matrix of shape (12, 32). GlobalAveragePooling1D() calculates
    # the mathematical average of all 12 time steps for each
    # of the 32 individual filters. This operation reduces the spatial/temporal
    # dimension to a single vector of shape (32).

    # The traditional approach was to use the Flatten() layer, which simply
    # unrolls the (12, 32) matrix into a long 1D vector of length 384.
    # Global Average Pooling provides two massive advantages:
    # - Parameter Reduction: By reducing the output from 384 to just 32 values,
    # it significantly limits the number of trainable weights required in the
    # subsequent Dense layer. This translates to a lighter, faster model.
    # - Overfitting Prevention: Because there are fewer parameters,
    # the model is physically incapable of memorizing the exact temporal
    # location of features (which causes overfitting). Instead, it forces
    # the network to learn the global presence of a pattern across the entire
    # sequence, resulting in much better generalization on unseen data.
    """

    model = Sequential([
        Input(shape=(WINDOW, 1)),
        Conv1D(filters=32, kernel_size=3, padding="causal", activation="relu"),
        GlobalAveragePooling1D(),
        # Dropout randomly deactivates 20% of neurons to force generalization
        Dropout(0.1),
        Dense(1)
    ])
    model.compile(optimizer="adam", loss="mse", metrics=["mae"])
    return model

def build_dilated_cnn():
```

```
    """
    Dilated CNN (Temporal Convolutional Network approach):
    Similar to a standard 1D CNN, but introduces gaps (dilation)
    between the filter elements.
    This allows the network to process a larger time context without
    increasing the number of parameters.
    """

    # Dilation: In a standard 1D CNN (dilation_rate=1), the convolutional filter
    # looks at strictly consecutive data points (e.g., t, t-1, t-2).
    # A Dilated Convolution introduces gaps between the elements the filter
    # processes. For instance, with kernel_size=3 and dilation_rate=2,
    # the filter skips one step between observations, looking
    # at steps t, t-2, and t-4.

    # The Receptive Field Problem: this is the total number of historical time
    # steps the network can "see" to make a single prediction.
    # In a standard CNN, to increase the receptive field, we must either
    # increase the kernel_size or stack many layers. Both approaches
    # significantly increase the number of trainable parameters,
    # leading to slower training and a higher risk of overfitting.

    # By stacking Conv1D layers with exponentially increasing
    # dilation rates (e.g., 1, 2, 4, 8), the network creates
    # a architecture known as a Temporal Convolutional Network (TCN).


    model = Sequential([
        Input(shape=(WINDOW, 1)),
        Conv1D(filters=32, kernel_size=3, padding="causal", dilation_rate=1,
               activation="relu"),
        Conv1D(filters=32, kernel_size=3, padding="causal", dilation_rate=2,
               activation="relu"),
        GlobalAveragePooling1D(),
        Dense(1)
    ])
    model.compile(optimizer="adam", loss="mse", metrics=["mae"])
    return model

# Initialize all models
models = {
    "LSTM": build_lstm(),
    "GRU": build_gru(),
    "1D CNN": build_cnn1d(),
    "Dilated CNN": build_dilated_cnn()
}
```

```
# ============================================================
# 6. EVALUATION METRICS & BASELINE
# ============================================================
def inverse_1d(arr):
    return scaler.inverse_transform(arr.reshape(-1, 1)).ravel()

def evaluate_model(y_true_scaled, y_pred_scaled):
    y_true = inverse_1d(y_true_scaled)
    y_pred = inverse_1d(y_pred_scaled)
    mae = mean_absolute_error(y_true, y_pred)
    rmse = np.sqrt(mean_squared_error(y_true, y_pred))
    return mae, rmse, y_true, y_pred

# --- 1. Standard Naive Baseline ---
# "It will always be like it was a month ago" (the final element of the window)
naive_pred_scaled = X_test[:, -1, 0]
naive_mae, naive_rmse, y_test_real, naive_pred_real = evaluate_model(y_test, naive_pred_scaled)

# --- 2. Seasonal Naive Baseline ---
# "It will always be the same as a year ago" (the first element of a 12-month window)
seasonal_naive_pred_scaled = X_test[:, 0, 0]
s_naive_mae, s_naive_rmse, _, s_naive_pred_real = evaluate_model(y_test, seasonal_naive_pred_scaled)

# ============================================================
# 7. MODEL TRAINING
# ============================================================
early_stopping = EarlyStopping(
    monitor="val_loss",
    patience=200,
    restore_best_weights=True
)

# We initialize the results dictionary, adding both of our baselines right away
results = {
    "Baseline: Naive (1-month)": {
        "MAE": naive_mae,
        "RMSE": naive_rmse,
        "pred": naive_pred_real
    },
    "Baseline: Seasonal Naive (12-month)": {
        "MAE": s_naive_mae,
        "RMSE": s_naive_rmse,
        "pred": s_naive_pred_real
    }
}
```

```
print("2. Training neural network models...\n")

for name, model in models.items():
    print("Training process started for:", name)

    history = model.fit(
        X_train, y_train,
        validation_data=(X_val, y_val),
        epochs=EPOCHS,
        batch_size=BATCH_SIZE,
        shuffle=True, # Windows shuffling!
        callbacks=[early_stopping],
        verbose=0
    )

    # Predict values for the test dataset
    pred_scaled = model.predict(X_test, verbose=0).ravel()
    mae, rmse, y_test_real, pred_real = evaluate_model(y_test, pred_scaled)

    results[name] = {
        "MAE": mae,
        "RMSE": rmse,
        "pred": pred_real,
        "epochs_run": len(history.history["loss"])
    }

"""
Technical Note: Shuffling in Time-Series Forecasting

In Keras, the model.fit() function defaults to shuffle=True,
meaning the training samples are randomly shuffled before each epoch.
In time-series forecasting, preserving temporal order is usually
the golden rule.

1. Stateless Models (Windowed Approach) - Shuffling is GOOD
In our current architecture, we transformed the continuous time-series
into discrete, overlapping sequences using a sliding window technique.

The strict chronological order inside each 12-month window
is perfectly preserved.

Shuffling only changes the order in which these independent
windows are fed to the network during training.

Conclusion: Explicitly setting shuffle=True is actually
recommended here. It ensures each training batch contains
a diverse mix of trends and seasons (breaking batch-to-batch correlation),
```

```
which stabilizes the gradients and prevents the optimizer
from getting stuck in local minima.

2. Stateful Models - Shuffling is STRICTLY FORBIDDEN
If the architecture was changed to a "Stateful" RNN
(where stateful=True is passed to the LSTM layer),
the network assumes that the first sequence in Batch 2
is the exact temporal continuation of the first sequence in Batch 1.

Conclusion: In stateful architectures, you must explicitly
set shuffle=False. Failing to do so will completely destroy
the temporal context and corrupt the model's memory state across batches.
"""

# ============================================================
# 8. RESULTS SUMMARY
# ============================================================
print("\n" + "=" * 65)
print("TEST SET EVALUATION RESULTS")
print("=" * 65)

# Sort the dictionary based on MAE values (ascending order)
sorted_results = sorted(results.items(), key=lambda x: x[1]["MAE"])

for i, (name, info) in enumerate(sorted_results, 1):
    epoch_info = ""
    if "epochs_run" in info:
        epoch_info = " | epochs required: {}".format(info["epochs_run"])

    print("{:>2}. {:<25} MAE = {:8.2f} | RMSE = {:8.2f}{}".format(
        i, name, info["MAE"], info["RMSE"], epoch_info
    ))

"""
The Golden Rule: If a complex Deep Learning model cannot significantly
outperform the naive baseline, the complex model is considered a failure
and should not be deployed. The added complexity, training time,
and maintenance costs are not justified.

The "Naive Approach" in Time Series: In time-series forecasting,
the most common baseline is the Naive Forecast. It relies on the assumption
that the immediate future will be exactly the same as the immediate past.

In our code, the baseline rule was: "Predict that the number of passengers
next month will be exactly equal to the number of passengers this month."
"""
```

```
# ============================================================
# 9. FINAL VISUALIZATION
# ============================================================
plt.figure(figsize=(12, 6))

# Plot actual targets
plt.plot(y_test_real, label="Actual Ground Truth", linewidth=3, color="black")

# Plot model predictions
for name, info in results.items():
    plt.plot(info["pred"], label=name, linestyle="--", alpha=0.85)

plt.title("Model Predictions on Test Dataset")
plt.xlabel("Time Step (Test Sequence)")
plt.ylabel("Number of Passengers")
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

4. LECTURE 3 – CONVOLUTIONAL NEURAL NETWORKS

Convolutional Neural Network (CNN) – class of deep neural networks for processing data that has a **grid-like structure** (LeCun et al. 1998).

This can be a **spatial grid** (2D images, satellite maps, spectrograms) or a **time-based grid** (1D time-series, continuous waveforms).

Traditional neural networks flatten the input data, which destroys the information about the local structure of time or space. CNNs solve this by analyzing data using small, sliding filters.

This allows the network to automatically extract characteristic features, regardless of exactly when they occur in the signal.

4.1 Key CNN components

A typical CNN implementation order:

0) Data preparation
- windowing
- normalization
- input tensor shape

1) Convolutional layer
- number of filters
- kernel size
- stride
- padding

2) Batch normalization
3) Activation function

4) Pooling layer
5) Dropout
6) Repeated convolutional blocks
7) Flattening or global pooling
8) Fully connected layer
9) Output layer

Architectural components:

0) Data preparation
Before any data enters the CNN, it must be properly formatted.
For continuous signals, this means **windowing** – slicing long, continuous recordings into shorter, fixed-length segments.
The data must also be **normalized** (scaled to a standard range, like 0 to 1 or –1 to 1) so the network can learn efficiently.
Finally, it must be **reshaped** into the correct input tensor shape expected by the network (e.g., specifying the batch size, number of channels, and window length).

1) The convolutional layer (filters/kernels)
- The convolutional layer is the main building block of a CNN. It applies a set of trainable **filters** to the input data. Each filter learns to detect a certain local pattern.
- In a one-dimensional CNN, the filter slides along the time axis.
- In a two-dimensional CNN, the filter slides across height and width, as in an image or spectrogram.

Number of filters
- The number of filters determines how many different local patterns the layer can learn.
- A small number of filters makes the model simpler and faster.
- A larger number of filters gives the model more capacity, but it also increases the risk of overfitting, especially when the dataset is small.

Kernel size
- The kernel size controls how a local region is seen by one filter at one time.
- In a 1D CNN, the kernel size refers to the number of neighbouring samples in a signal. Therefore, it defines the length of the local time window seen by the filter.
- In a 2D CNN, the kernel size refers to the size of the local spatial region analysed by the filter.
- A small kernel size, such as 3, 5, or 7 samples, makes the filter focus on very local patterns. In time-series data, this is useful for detecting short and sharp changes.
- A larger kernel size makes the filter analyse a wider part of the signal or image at once. In time-series data, this allows the filter to detect broader waveform shapes or slower signal variations.

Window: what part of the input is currently seen.
Kernel: the trainable weights applied to that part.
Filter: the learned detector that uses the kernel across the input.

Stride and padding:
These two settings control exactly how the filter slides across the data.

Stride:
This dictates how many data points the filter jumps over with each step. A stride of 1 means it moves point by point.
A larger stride (like 2 or 3) reduces the time dimension, shrinking the data as it moves forward.
A small stride preserves more detail.
A larger stride reduces the size of the representation more quickly.

Padding:
Padding controls how the model treats the edges of the input.
Without padding, the output becomes shorter after convolution, and information near the edges may be underrepresented.
With padding, the model can preserve the input length more easily and process edge regions more symmetrically.

2) Batch normalization
Batch normalization is often used to stabilize and speed up training.
For many CNN applications, adding batch normalization after convolution and before activation is a reasonable default choice.

3) Activation function (usually ReLU)
After a convolutional layer, an activation function is usually applied.
The activation function introduces non-linearity into the model.
The most common activation function in CNNs is ReLU.

4) The pooling layer
This layer is used to reduce the size of the data while preserving the most critical features.
For example, Max Pooling looks at a small window of the data and only passes the maximum value forward.

5) Dropout
Dropout is a regularization technique.
During training, it randomly disables part of the network. This makes the model less dependent on individual neurons and can reduce overfitting.
Dropout can be useful when the dataset is small.

6) Repeated convolutional blocks
A single layer is rarely enough. CNNs are built by stacking multiple blocks together (usually a sequence of Convolution – Batch Normalization – Activation – Pooling).
Early blocks detect very simple, high-frequency changes, while deeper, repeated blocks combine these simple detections to recognize highly complex, long-term patterns.

7) Flattening or Global Pooling
Before the extracted features can be used to make a final decision, their shape must change.
Flattening takes the multi-dimensional output from the convolutional layers and straightens it out into a single, long 1D list.
Alternatively, Global Pooling shrinks the entire time or spatial dimension down to a single value per filter. This is often preferred in time-series analysis because it makes the model robust to inputs of variable lengths.

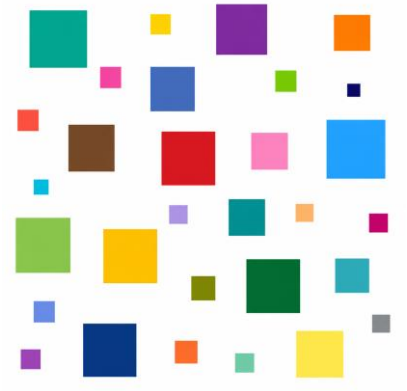
8) Fully connected (linear) layers
At the very end of the network, the highly compressed and extracted features are “flattened” into a simple list.
This list is passed into a traditional neural network to make the final decision.

9) Output layer
It takes the processed information from the fully connected layer and translates it into the final prediction.
Depending on the task, it uses a specific activation function (like Softmax) to output probabilities, such as a 90% chance the window contains an earthquake and a 10% chance it is background noise.

4.2 Example 3

Code using a CNN to detect squares in an image

Input: An image containing squares



```
"""
================================================================================
SQUARE DETECTION PIPELINE: FCN & OpenCV
================================================================================

It combines a custom neural network for pixel segmentation with classical
computer vision for bounding box extraction.

Workflow:
1. Data Generation: Creates a synthetic dataset of RGB images containing
    random squares and their corresponding binary masks (ground truth).
2. Neural Network (Keras): Builds and trains a simple Fully Convolutional
    Network (Encoder-Decoder architecture) to perform binary pixel segmentation.
3. Post-Processing (OpenCV): Cleans the predicted mask using morphological
    opening to remove noise. Extracts final bounding boxes using Connected
    Components, filtering the results by area and aspect ratio.
================================================================================
"""

#
================================================================================
# 1. Imports
```

```
#
# ============================================================

import numpy as np
import cv2
import matplotlib.pyplot as plt

import tensorflow as tf
from tensorflow.keras import layers, models

from google.colab import files

#
# ============================================================
# 2. Basic settings
#
# ============================================================

IMG_SIZE = 256          # image size used by the CNN
N_TRAIN = 800           # number of synthetic training images
N_VAL = 100             # number of validation images
EPOCHS = 5              # small number of epochs for a simple demo
BATCH_SIZE = 16

MIN_SQUARES = 5         # minimum number of squares in synthetic images
MAX_SQUARES = 18        # maximum number of squares in synthetic images

MIN_SIZE = 8            # minimum synthetic square size
MAX_SIZE = 55           # maximum synthetic square size

'''
Synthetic Data Generator

Training data from scratch.
Since we are training a neural network to find squares,
we need hundreds of examples of images with squares
and their exact locations (masks).

For every single image, the function follows these steps:
1. Creates a blank canvas: It starts with a pure white image
and a completely black mask (where 0 means "background").
2. Rolls the dice: It randomly decides how many squares
to draw on this specific image.
3. Draws the squares: For each square, it picks a random size,
a random position (x, y), and a random color.
4. Creates the "Ground Truth": While drawing the colored square on the image,
it simultaneously marks the exact same area on the mask with 1.0.
```

```
This mask is the "correct answer" that the network will try to predict.
'''

#
# 3. Synthetic data generator
#

def generate_dataset(n_images, img_size=256):
    """
    Generate simple synthetic images.

    Each image has:
    - white background
    - random colored squares
    - a binary mask where square pixels are 1 and background pixels are 0

    Because we generate the images ourselves, we also know the correct masks.
    """

    images = []
    masks = []

    for _ in range(n_images):

        # Create a white RGB image.
        image = np.ones((img_size, img_size, 3), dtype=np.float32)

        # Create an empty mask.
        mask = np.zeros((img_size, img_size, 1), dtype=np.float32)

        # Random number of squares.
        n_squares = np.random.randint(MIN_SQUARES, MAX_SQUARES + 1)

        for _ in range(n_squares):

            # Random square size.
            size = np.random.randint(MIN_SIZE, MAX_SIZE + 1)

            # Random position.
            x = np.random.randint(0, img_size - size)
            y = np.random.randint(0, img_size - size)

            # Random color.
            # Values are kept below 1.0, so the squares are different from the white background.
            color = np.random.uniform(0.05, 0.9, size=3)
```

```
            # Draw the square on the image.
            image[y:y + size, x:x + size, :] = color

            # Draw the same square on the mask.
            mask[y:y + size, x:x + size, 0] = 1.0

        images.append(image)
        masks.append(mask)

    return np.array(images), np.array(masks)

#
# =============================================================
# 4. Create the training and validation data
#
# =============================================================

print("Generating synthetic training data...")

X_train, y_train = generate_dataset(N_TRAIN, IMG_SIZE)
X_val, y_val = generate_dataset(N_VAL, IMG_SIZE)

print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
print("X_val shape:", X_val.shape)
print("y_val shape:", y_val.shape)

#
# =============================================================
# 5. Show n synthetic images and their masks
#
# =============================================================

n_examples = 5    # number of examples to show

plt.figure(figsize=(8, 4 * n_examples))

for i in range(n_examples):

    # Show synthetic image
    plt.subplot(n_examples, 2, 2 * i + 1)
    plt.imshow(X_train[i])
    plt.title(f"Synthetic image {i}")
    plt.axis("off")

    # Show correct mask
    plt.subplot(n_examples, 2, 2 * i + 2)
```

```
    plt.imshow(y_train[i, :, :, 0], cmap="gray")
    plt.title(f"Correct mask {i}")
    plt.axis("off")

plt.tight_layout()
plt.show()

'''
Segmentation Network: The Encoder-Decoder Architecture

This Convolutional Neural Network (CNN) here consists of two main parts:

1. The Encoder (Shrinking part): The network uses `MaxPooling2D` layers
to shrink the image step-by-step. While the image gets smaller
in width and height, the network extracts deeper patterns
(edges, corners, colors) using `Conv2D` layers.
This helps the network understand "what" it is looking at
and find the general location of the squares.

2. The Decoder (Expanding part): Since we need a mask that has the exact
same size as the original image, we cannot just output a tiny feature map.
The network uses `UpSampling2D` layers to blow the image back up
to its original dimensions, using the features learned by the encoder
to carefully paint the exact boundaries of the squares.

3. The Final Output: The last layer uses a `sigmoid` activation function,
which acts like a strict probability switch. It forces the final prediction
for every single pixel to be a value between 0.0 (definitely background)
and 1.0 (definitely a square).
'''

#
============================================================
# 6. Build a small CNN segmentation model
#
============================================================

def build_model(img_size=256):
    """
    Build a simple CNN for binary segmentation (Encoder-Decoder architecture).

    Input:
    - RGB image (shape: img_size x img_size x 3)

    Output:
    - one-channel mask (shape: img_size x img_size x 1)
    - each pixel value is a probability between 0.0 and 1.0
```

```
"""

model = models.Sequential()

# Define the input shape for the network
model.add(layers.Input(shape=(img_size, img_size, 3)))

# ------------------------------------------------
# ENCODER PART: Extract features and reduce resolution
# ------------------------------------------------

# 1st Convolutional Block
# 16 filters (feature detectors). padding="same" ensures the image
# doesn't shrink during convolution.
model.add(layers.Conv2D(16, 3, padding="same", activation="relu"))
# MaxPooling reduces the image dimensions by half
# (e.g., from 256x256 to 128x128).
# This helps the network focus on broader patterns rather than tiny details.
model.add(layers.MaxPooling2D())

# 2nd Convolutional Block
model.add(layers.Conv2D(32, 3, padding="same", activation="relu"))
# Reduces dimensions by half again (e.g., from 128x128 to 64x64).
model.add(layers.MaxPooling2D())

# "Bottleneck" layer - the most compressed representation of the image
model.add(layers.Conv2D(64, 3, padding="same", activation="relu"))

# ------------------------------------------------
# DECODER PART: Restore resolution to match input size
# ------------------------------------------------

# 1st Upsampling Block
# UpSampling simply duplicates pixels to double the image dimensions
# (e.g., from 64x64 back to 128x128).
model.add(layers.UpSampling2D())
model.add(layers.Conv2D(32, 3, padding="same", activation="relu"))

# 2nd Upsampling Block
# Doubles dimensions again (e.g., from 128x128 back to the original 256x256).
model.add(layers.UpSampling2D())
model.add(layers.Conv2D(16, 3, padding="same", activation="relu"))

# ------------------------------------------------
# OUTPUT LAYER: Pixel-wise Classification
# ------------------------------------------------
```

```
    # We use a 1x1 Convolution to squash the 16 feature maps
    # down to just 1 channel.
    # The "sigmoid" activation guarantees every pixel's value is between 0 and 1.
    model.add(layers.Conv2D(1, 1, activation="sigmoid"))

    # Compile the model
    # "binary_crossentropy" is the standard math formula for measuring
    # error when predicting YES/NO (1/0) for pixels.
    model.compile(
        optimizer="adam",
        loss="binary_crossentropy",
        metrics=["accuracy"]
    )

    return model


model = build_model(IMG_SIZE)
model.summary() # Prints a table showing the shape of the data
# as it flows through the layers

#
# ============================================================
# 7. Train the CNN
#
# ============================================================

print("Training the CNN...")

history = model.fit(
    X_train,
    y_train,
    validation_data=(X_val, y_val),
    epochs=EPOCHS,
    batch_size=BATCH_SIZE
)

#
# ============================================================
# 8. Plot the training history
#
# ============================================================

plt.figure(figsize=(7, 4))
plt.plot(history.history["loss"], label="training loss")
plt.plot(history.history["val_loss"], label="validation loss")
plt.xlabel("Epoch")
```

```
plt.ylabel("Loss")
plt.title("Training history")
plt.legend()
plt.grid(True)
plt.show()

#
============================================================
# 9. Load the image with squares from the Colab container
#
============================================================

image_path = "example.png"

print("Loaded file:", image_path)

'''
Preparing the Real Image for the Network

Before we can feed a real-world photo into our neural network,
we must format it to exactly match the synthetic data
the network was trained on. This process is called preprocessing.

This block performs four critical steps:
1. Loading: Reads the image file from the disk.
2. Color Correction: OpenCV historically reads color channels
in Blue-Green-Red (BGR) order. We must flip this to standard
Red-Green-Blue (RGB) so the colors match what our network expects.
3. Resizing: Scales the image to our defined `IMG_SIZE` (e.g., 256x256 pixels).
(Note: Shrinking a high-resolution image might make very small objects
disappear before the network even sees them).
4. Normalization: Neural networks struggle with large numbers.
We divide the standard pixel values (which range from 0 to 255) by 255.0
to squeeze them into a small, mathematical range of 0.0 to 1.0.
'''

#
============================================================
# 10. Read and prepare the uploaded image
#
============================================================

# OpenCV reads images in BGR format.
image_bgr = cv2.imread(image_path)

if image_bgr is None:
    raise ValueError("Image could not be loaded. Please check the file.")
```

```
# Convert BGR to RGB for plotting and CNN input.
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)

original_h, original_w = image_rgb.shape[:2]

# Resize image to the CNN input size.
image_small = cv2.resize(image_rgb, (IMG_SIZE, IMG_SIZE))

# Convert pixel values from 0-255 to 0-1.
image_small = image_small.astype(np.float32) / 255.0

#
============================================================
# 11. Predict the square mask
#
============================================================

# Step 1: Run the image through the Neural Network
# Keras models always expect a "batch" (a group) of images,
# even if we only have one.
# 'image_small[None, ...]' artificially adds a batch dimension: (256, 256, 3)
# becomes (1, 256, 256, 3).
# After prediction, '[0, :, :, 0]' unwraps the result:
# It takes the first image (0), all Y pixels (:), all X pixels (:),
# and the first channel (0).
pred = model.predict(image_small[None, ...])[0, :, :, 0]

# Step 2: Convert probabilities to a strict Binary Mask
# The network outputs decimal probabilities (e.g., 0.8 means 80% sure).
# We use a threshold of 0.5: anything above 0.5 becomes True (1), else False (0).
# We then convert it to an 8-bit integer (np.uint8) because OpenCV requires
# this format later.
mask_small = (pred > 0.5).astype(np.uint8)

# Step 3: Stretch the mask back to the original photo's dimensions
# INTER_NEAREST is extremely important here!
# We are resizing a mask containing only 0s and 1s.
# Standard smooth resizing (like Bilinear) would create invalid "gray"
# values (e.g., 0.4) on the edges.
# Nearest Neighbor ensures our mask remains strictly binary (0 or 1)
# after stretching.
mask_big = cv2.resize(
    mask_small,
    (original_w, original_h),
    interpolation=cv2.INTER_NEAREST
)
```

```
#
# ============================================================
# 12. Clean the mask
#
# ============================================================

# Step 1: Define the "brush" size (Kernel)
# We create a 5x5 grid of 1s. This acts as our reference size.
# Any white shape in our mask that cannot fit this 5x5 block inside it
# will be considered "noise" and deleted during the next step.
kernel = np.ones((5, 5), np.uint8)

# Step 2: Apply Morphological Opening
# "Opening" (MORPH_OPEN) means performing Erosion (shrinking) followed
# by Dilation (expanding).
# - First, Erosion removes tiny, isolated pixels (false positives from the network).
# - Then, Dilation restores the real, large squares back to their proper
# size and smooths their edges.
#
# If our original image contained extremely tiny squares (e.g., 3x3 pixels),
# this specific operation will remove them entirely
mask_clean = cv2.morphologyEx(mask_big, cv2.MORPH_OPEN, kernel)

#
# ============================================================
# 13. Find connected components
#
# ============================================================

# Step 1: Group touching pixels into distinct objects
# connectedComponentsWithStats scans the binary mask and identifies
# separate "blobs" of white pixels.
# - num_labels: Total number of distinct objects found (PLUS the background).
# - labels: An image where each pixel is labeled with
# its object's ID (0, 1, 2...).
# - stats: A list of bounding boxes (x, y, width, height) and the pixel area
# for each object.
# - centroids: The (x, y) center coordinates of each object.
num_labels, labels, stats, centroids = cv2.connectedComponentsWithStats(
    mask_clean,
    connectivity=8 # 8-connectivity means pixels touching diagonally
    # are considered connected
)

boxes = []

# Step 2: Loop through every detected object
# TEACHING POINT: We start the loop at 1, not 0!
```

```
# In OpenCV, Label 0 is ALWAYS the black background.
# We only care about the white objects.
for i in range(1, num_labels):

    # Unpack the geometric statistics for this specific object
    x, y, w, h, area = stats[i]

    # Step 3: Filter 1 - Size (Area)
    # If the object is made of less than 100 pixels,
    # it's probably just leftover noise. Skip it.
    if area < 100:
        continue

    # Step 4: Filter 2 - Shape (Aspect Ratio)
    # Aspect Ratio is calculated by dividing Width by Height.
    ratio = w / h

    # A mathematically perfect square has a ratio of exactly 1.0.
    # However, neural network masks are rarely perfect,
    # so we allow a tolerance (between 0.6 and 1.6).
    # If it falls within this range, we accept it as a valid square
    # and save its coordinates.
    if 0.6 <= ratio <= 1.6:
        boxes.append((x, y, w, h, area))

print("Detected squares:", len(boxes))

#
============================================================
# 14. Draw boxes and numbers
#
============================================================

# Step 1: Create a safe copy of the image
# We always draw on a copy - modifying the original image directly
# is a bad practice,
# because you would lose the original data if you wanted to run the cell again.
result = image_rgb.copy()

# Step 2: Loop through all the saved bounding boxes
# 'enumerate(boxes, start=1)' gives us both the index (idx) starting from 1,
# and the box coordinates (x, y, w, h, area) all at once.
for idx, (x, y, w, h, area) in enumerate(boxes, start=1):

    # Step 3: Draw the bounding box
    cv2.rectangle(
        result,                    # The image to draw on
```

```
            (x, y),                 # Top-left corner coordinate
            (x + w, y + h),         # Bottom-right corner coordinate
            (255, 0, 0),            # Color: Red (because our image
                                      # is in RGB format)
            3                         # Line thickness in pixels
        )

        # Step 4: Draw the object number above the box
        cv2.putText(
            result,                 # The image to draw on
            str(idx),               # The text to display (the square's number)

            # Smart positioning
            # 'y - 8' places the text slightly above the box.
            # 'max(..., 20)' ensures that if the box is at the very top
            # of the image (y=0),
            # the text is forced down to y=20 so it doesn't disappear off
            # the screen
            (x, max(y - 8, 20)),

            cv2.FONT_HERSHEY_SIMPLEX,# Standard readable font
            0.8,                      # Font scale (size)
            (255, 0, 0),            # Color: Red
            2                         # Text thickness
        )

#
# 15. Show final result
#

plt.figure(figsize=(10, 10))
plt.imshow(result)
plt.title(f"Detected squares: {len(boxes)}")
plt.axis("off")
plt.show()

#
# Methodological note
#

"""
existing datasets and ready object detection models
many large public datasets - two important examples are COCO
and Open Images Dataset.
```

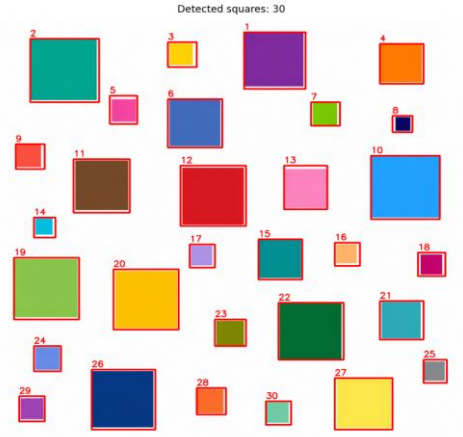
These datasets contain many real-world images with annotated objects.
They are very useful when we want to detect common objects
such as people, cars, animals, bottles, chairs, or traffic signs.

YOLO is a popular family of object detection models.
A ready YOLO model is often trained on datasets such as COCO.
This means that the model can detect only the object classes
that were present in its training data.

For example, if a YOLO model was trained on COCO,
it can detect COCO classes, but it should not be expected
to detect every possible object or shape.

The advantage of large existing datasets is that they contain many
real images and are useful for general object detection.
The limitation is that they only cover selected object classes.
The advantage of a synthetic dataset is full control over object size,
color, position, and labels.
The limitation is that synthetic images may be too simple
and may not represent real-world complexity.
"""

Expected output - Detected squares: 30



5. LECTURE 4 – AUTOMATIC SPEECH RECOGNITION AND WAV2VEC 2.0

ASR (Automatic Speech Recognition) and NLP (Natural Language Processing) are two complementary artificial intelligence technologies. NLP is the field of machine learning concerned with processing human language in text form. ASR is the field concerned with converting spoken audio into text (Mohamed et al. 2022; Schneider et al. 2019; Baevski et al. 2020; Mousavi et al. 2020; Harsuko and Alkhalifah 2022; Sheng et al. 2025; Tunstall et al. 2022; Vaswani et al. 2017; Devlin et al. 2019; Murphy 2023).

A simple ASR pipeline is:

audio signal → ASR model → transcript

A simple NLP pipeline is:

text → numerical representation → model → prediction or generated text

Wav2Vec 2.0

- Wav2Vec 2.0 is advanced AI model for speech-to-text (ASR) by Facebook AI Research (FAIR).
- It learns to recognize sound patterns from raw audio, without the need for manual transcription by humans.
- This ensures high accuracy and excellent handling of languages with limited data.

General information

- The first version appeared in mid–2019, and the groundbreaking Wav2Vec 2.0 debuted in June 2020. Since then, Meta has been releasing subsequent, advanced variants.
- The Wav2Vec models are publicly available free of charge (open source) under licenses such as Apache 2.0 or MIT.
- The Hugging Face Platform: Ready-to-use model variants (e.g., trained on hundreds of thousands of hours of speech) are widely available to developers and researchers in the Hugging Face Transformers library.

5.1 Example 4

Input: The code requires a .wav file containing a voice recording. kod wymaga pliku .wav z nagraniem głosu

```
import wave
import numpy as np
import matplotlib.pyplot as plt

audio_path = "voice.wav"     # WAV file name

with wave.open(audio_path, "rb") as wf:
    n_channels = wf.getnchannels()
    sampling_rate = wf.getframerate()
    n_frames = wf.getnframes()
    sampwidth = wf.getsampwidth()
    raw = wf.readframes(n_frames)

# data type
if sampwidth == 1:
    dtype = np.uint8
elif sampwidth == 2:
    dtype = np.int16
elif sampwidth == 4:
    dtype = np.int32
else:
    raise ValueError("Unsupported sample width")

audio = np.frombuffer(raw, dtype=dtype)

# if stereo, change to mono
if n_channels > 1:
    audio = audio.reshape(-1, n_channels).mean(axis=1)
```

```
# normalization for figure
audio = audio.astype(np.float32)
audio = audio / np.max(np.abs(audio))

time_axis = np.arange(len(audio)) / sampling_rate

plt.figure(figsize=(12, 4))
plt.plot(time_axis, audio, linewidth=0.8)
plt.xlabel("Time [s]")
plt.ylabel("Amplitude")
plt.title("Waveform of the audio recording")
plt.tight_layout()
plt.show()

print("Sampling rate:", sampling_rate, "Hz")
print("Duration:", len(audio) / sampling_rate, "s")

!pip install transformers torchaudio librosa soundfile -q
# Librosa is a highly popular Python library
# used for analyzing and processing music and audio signals

import librosa
import torch
from transformers import Wav2Vec2Processor, Wav2Vec2ForCTC
# Wav2Vec2Processor (Data Preparation)
# handles all data transformations before the data
# enters the model and after it exits.
# Wav2Vec2ForCTC (The Network Architecture)
# Inside this class sits the core Wav2Vec 2.0 model,
# the exact stack of CNN blocks and the Contextual Transformer
# CTC stands for Connectionist Temporal Classification.
# CTC is a specialized algorithm and loss function.

audio_path = "voice.wav"
model_name = "facebook/wav2vec2-base-960h"

processor = Wav2Vec2Processor.from_pretrained(model_name)
model = Wav2Vec2ForCTC.from_pretrained(model_name)

audio, sampling_rate = librosa.load(audio_path, sr=16000)

inputs = processor(
    audio,
    sampling_rate=16000,
    return_tensors="pt",
    padding=True
```

```
)

with torch.no_grad():
    logits = model(inputs.input_values).logits

predicted_ids = torch.argmax(logits, dim=-1)
transcription = processor.batch_decode(predicted_ids)[0]

print("Transcription:")
text = "This is a short example recording for a wav two vec two point zero demonstration. \
The model should transcribe this English sentence into text."
print(text)
print(transcription)
```

5.2 Wav2Vec 2.0 and geosciences

Transfer Learning is a deep learning technique in which model weights optimized while solving one problem (usually on a large dataset) are used as a starting point for training a network on another, new task (usually with less data).

Possible applications

- This model (originally created for speech recognition) processes and finds patterns in raw sound waves, so its architecture can be successfully adapted to analyze other waveforms.
- The greatest advantage of Wav2Vec 2.0 in geophysics is its ability to learn powerful representations from "unlabelled" data (self-supervised learning).

Why is it so interesting?

- Geophysics is based on the analysis of waves (seismic, acoustic, gravity).
- Wav2Vec-2.0 allows us to treat continuous signals from nature in a way very similar to how systems process human speech.
- This enables the detection of subtle, repeatable patterns that are too complex or hidden in background noise for traditional analytical algorithms or the human eye.
- This is very relevant to geosciences because we often have: large amounts of raw waveform data but relatively few reliable labels.
- For example: continuous seismic records, volcanic tremor, icequake catalogues, microseismic monitoring, laboratory acoustic emission, distributed acoustic sensing data.

5.3 Implementation

Hugging Face: place where Python can download a ready-made, pre-trained model

```
from transformers import Wav2Vec2Model
model = Wav2Vec2Model.from_pretrained("facebook/wav2vec2-base-960h")
```

Wav2Vec 2.0 architecture

- Wav2Vec 2.0 is not a single algorithm; it is a pipeline of neural network modules (Fig. 6).
- It is **pre-trained** on massive amounts of unlabeled raw audio to learn the fundamental structure of human speech, and then **fine-tuned** on a small amount of labeled data (audio-text pairs).

- **Self-supervised pre-training** is a training stage in which the model learns from the input data itself, without external labels.
- **Fine-tuning:** the second training stage, where the pre-trained model is adapted to a specific supervised task.
- **Wav2Vec 2.0 expects a one-dimensional waveform**.

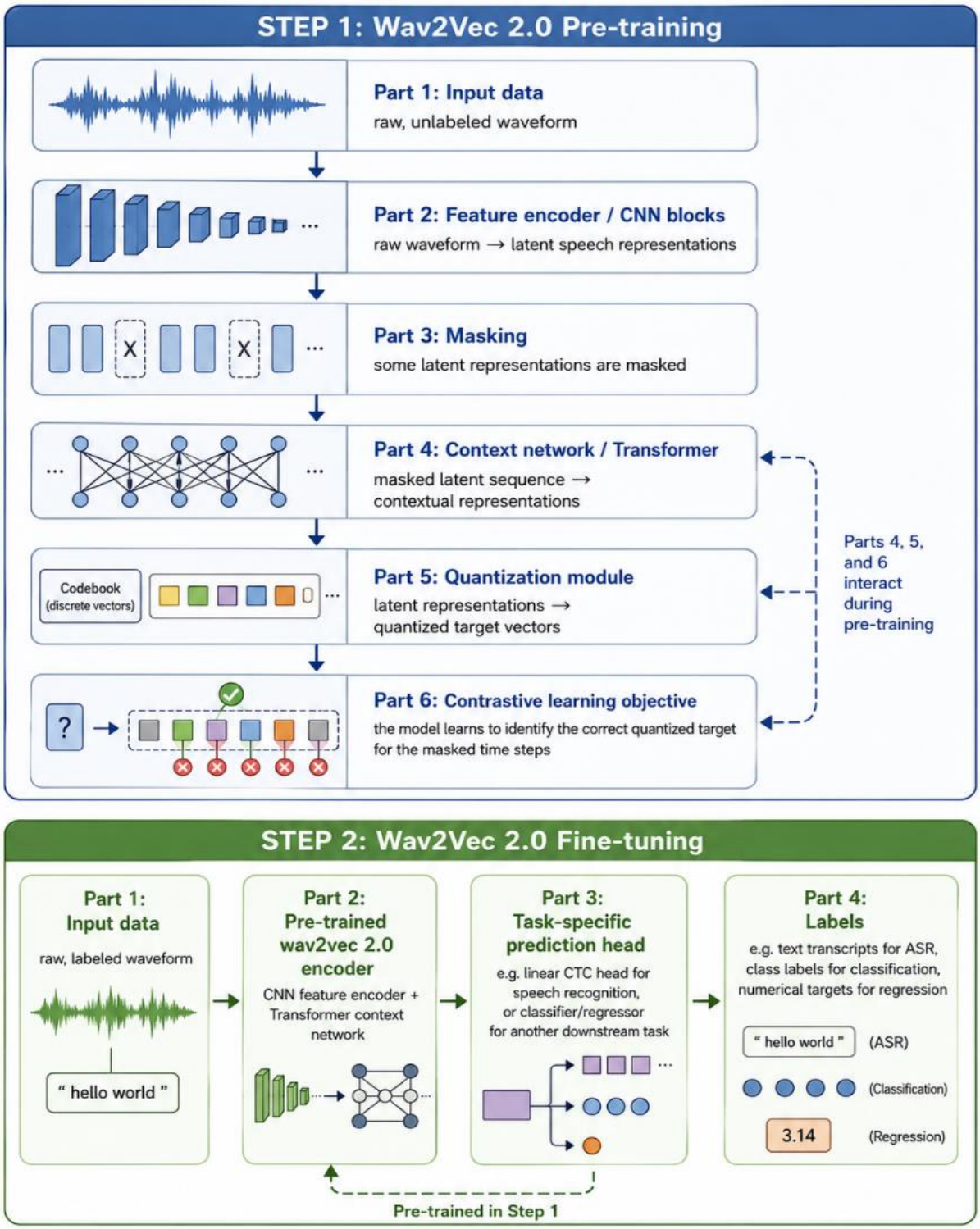


Fig. 6. Wav2Vec 2.0 workflow.

Step 1: Wav2Vec 2.0 pre-training	Step 2: Wav2Vec 2.0 fine-tuning
Part 1: Input data raw, unlabeled waveform **Part 2: Feature encoder / CNN blocks** raw waveform → latent speech representations **Part 3: Masking** some latent representations are masked **Part 4: Context network / Transformer** masked latent sequence → contextual representations **Part 5: Quantization module** latent representations → quantized target vectors **Part 6: Contrastive learning objective** the model learns to identify the correct quantized target for the masked time steps	**Part 1: Input data** raw, labeled waveform **Part 2: Pre-trained Wav2Vec 2.0 encoder** CNN feature encoder + Transformer context network **Part 3: Task-specific prediction head** e.g. linear CTC head for speech recognition, or classifier / regressor for another downstream task **Part 4: Labels** e.g. text transcripts for ASR, class labels for classification, numerical targets for regression

Three practical ways to use Wav2Vec 2.0 and three levels of difficulty:

- **Level 1:** Use pretrained Wav2Vec 2.0 as a frozen feature extractor. This is the easiest and most realistic first step.
- **Level 2:** Fine-tune Wav2Vec 2.0 for a geoscience task. Here, Wav2Vec-2.0 is not only used as a feature extractor. Its weights are adapted to the task.
- **Level 3:** Pretrain a Wav2Vec 2.0 like model on geoscience waveforms. This is the most scientifically interesting but also the most difficult option.

5.4 Detailed description – Step 1

Part 1 – Input data

- In the Pre-training step (Step 1), a huge, continuous stream of raw data is used so that the model can feed on it without limits to learn the wave physics (self-supervised learning).
- **A raw waveform** is a time series representing the recorded amplitude of a signal as a function of time.
- **Sampling rate (Hz):** The number of audio samples per second. **Crucial rule:** input audio sampling rate must exactly match the sampling rate the model was trained on (almost always 16 kHz: 16 000 samples = 1 second of audio). Hugging Face's Wav2Vec 2.0 feature extractor has a default sampling rate of 16 kHz.
- **Batching** is mandatory: Processing data one item at a time is highly inefficient. Group your text or audio arrays into batches to utilize GPU acceleration.

The model employs a "context window strategy" to process long audio sequences by dividing the input audio into overlapping segments, each with a fixed length (Fig. 7).

This allows the model to capture information from a broader context, which is important for understanding patterns in speech.

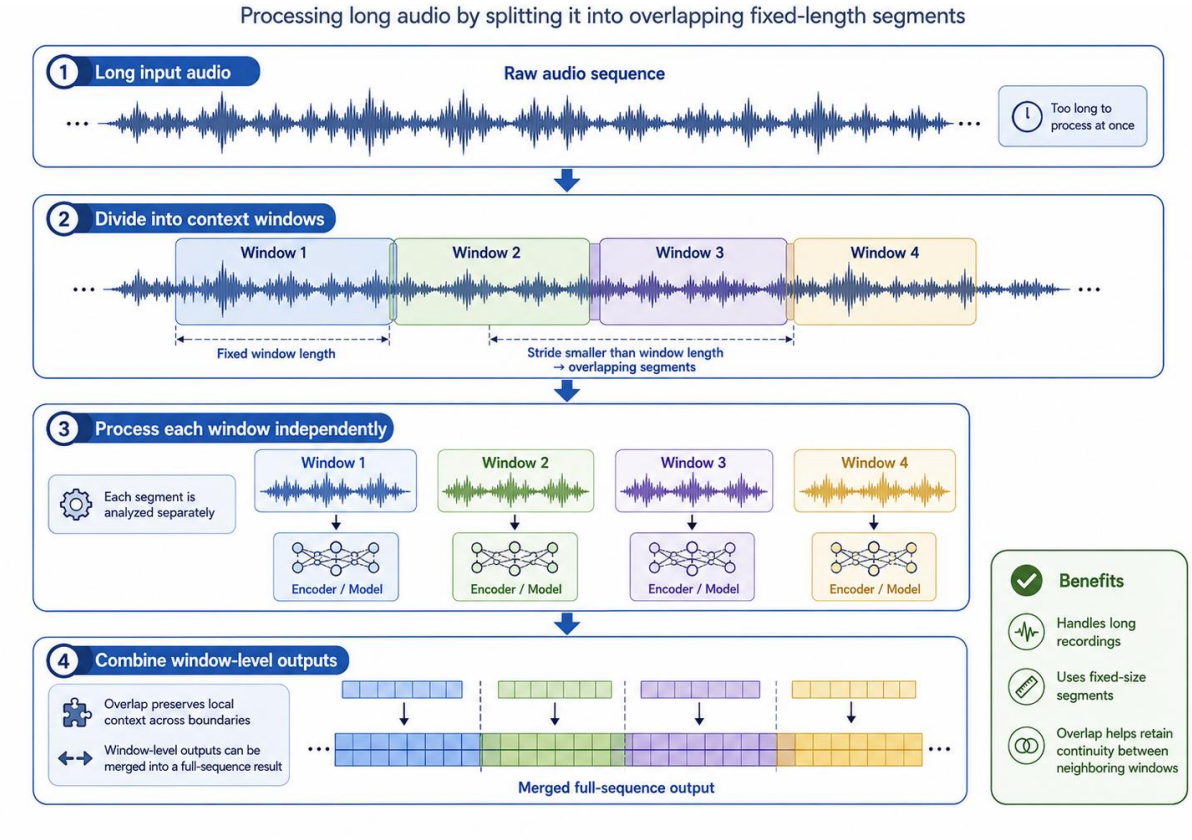


Fig. 7. Context window strategy.

Part 2 – Feature encoder / CNN blocks

- **Feature extraction:** Converting the raw waveform into features the model understands.
- **A feature encoder:** It is a Convolutional Neural Network that converts the raw waveform into **latent speech features**.
  - In the Wav2Vec 2.0 architecture, a one-second 16 kHz audio input consisting of 16 000 raw samples is processed by 1D Convolutional Neural Networks (1D CNNs) that discard redundant physical data (time and amplitude) to build a completely new, abstract representation.
  - Instead of analyzing large windows of numbers simultaneously, the model uses sets of **sliding filters** (matrices of weights) that shift along the waveform, multiplying their weights by the wave values and summing the results to detect specific geometric patterns, such as the absence of high frequencies or sudden amplitude spikes.
  - Because these filters jump forward by a **defined step size (the stride parameter)**, the cumulative stride compresses the timeline 320 times (7 convolutional blocks with strides [5,2,2,2,2,2,2]), reducing the original 16 000 samples to about 50 frames per second.
  - Each of these output points assumes a dimensionality equal to the number of filters applied, which in the Base model equates to **512 features per frame**.
  - Ultimately, while the total data volume slightly increases from 16,000 to 25,600 values (50 frames × 512 features), this continuous process along the entire recording successfully transforms the raw waveform into a highly structured, infinitely more understandable, and concise sequence of multidimensional latent vectors.

- These results are then passed through a activation function, like ReLU or GELU, which zeroes out weak or negative signals to ensure only highly significant features are retained.

- **A latent vector is often treated simply as a specific type of embedding**.
- **Embedding:** A dense numerical vector representing a token, sentence, document, or audio fragment. Embeddings are useful because similar words or documents often have similar vector representations. word → vector, sentence → vector, document → vector, audio fragment → vector. These vectors are often called embeddings.
- A **latent vector** is an internal numerical representation learned by the model. It is called "latent" because it is not directly observed in the data. It is produced inside the neural network.
- **Latent representation** learning means training a model to transform raw data into useful internal features.

Part 3 – Masking

- **Masking:** some parts of the latent representation are hidden from the model. The model must learn to infer useful information from the surrounding context. In Wav2Vec 2.0, masking is not usually applied directly to the raw waveform. Instead, **it is applied to the latent feature sequence produced by the CNN feature encoder.**

Part 4 – Context network / Transformer

- **Transformer:** A neural network architecture that is central to modern NLP and ASR. Transformers use a mechanism called **attention** to model relationships between elements of a sequence.
- A **Masked Transformer** is a Transformer model trained on a partially masked sequence of latent vectors.
  - For example, given an initial sequence of vectors, such as [v1, v2, v3, v4, v5], the masking process replaces specific elements, resulting in an input like [v1, MASK, MASK, v4, v5].
  - The Transformer analyzes the unmasked segments using the **Self-Attention mechanism**, which evaluates the bidirectional context – both preceding and succeeding the masked gaps – to infer and reconstruct the hidden information (produce **contextual representations** that are trained to identify the correct quantized target).
  - This masking strategy enables self-supervised pre-training on vast volumes of unlabeled data.
  - While the Transformer generates a **context vector c** and the CNN outputs a **latent vector z**, the architecture is explicitly designed **not to predict the continuous vector z directly**.
  - This constraint prevents a phenomenon known as **representation collapse**.
  - If the model were tasked with directly comparing continuous vectors, the CNN and Transformer networks could bypass learning meaningful features by converging on trivial, constant outputs.
  - The latent vectors generated by the CNN provide only a localized representation of the waveform's geometry.
  - It is only after passing through the Masked Transformer that these features acquire comprehensive temporal and physical context.

- During training, approximately half of the latent vectors are masked before entering the Transformer encoder.
- At each masked location, the model must distinguish between the true target (the positive example) and 100 distractors (negative examples) uniformly sampled from other positions within the sequence.
- **By optimizing a contrastive loss function based on cosine similarity**, the model learns to minimize the distance between its restored vectors and the corresponding true target vectors in the feature space, while simultaneously maximizing the distance from the distractors.
- **The vectors produced by the Transformer, denoted as c, are contextual representations**.

Part 5 – Quantization module

- Transformer gives c – **contextual representations**. Quantization module gives q – quantized target vectors. **Contrastive learning compares c with the correct q**.
- In standard neural networks, features are continuous and exist in an infinite, high-dimensional space. Trying to predict exact values in a continuous space is highly prone to instability and representation collapse.
- The **Quantization Module** solves this by taking the continuous latent vectors (the unmasked output from the CNN) and mapping them to a finite, learned **dictionary called a codebook.**
- The module assigns each continuous vector to its closest discrete counterpart in the codebook, resulting in a quantized representation.
- By doing this, the module establishes clear, well-defined "ground-truth" targets for the model to predict during the pre-training phase.

Part 6 – Contrastive learning objective

- The **first track** tries to guess the missing information based on the broader context, and the **second track** prepares the correct answers for it.
- **For each masked location, the Transformer generates a vector c.**
- In the second track, the system takes the same original z vectors from the CNN module, but without applying any mask to them, and prepares an answer key from them.
- After the CNN module, the data pathway splits into two independent branches (Fig. 8).
- At the very end of these tracks, the contextual vectors (the output from the Transformer) and the quantized vectors (the output from the quantization module) converge.
- The relationship between them can be distilled into a single sentence: the contextual representations serve as the model's predictions (hypotheses), while the quantized representations act as the ground truth answers (targets).
- Since Wav2Vec 2.0 learns from unlabeled data, it must formulate its own optimization task.
- It achieves this by using a **contrastive loss function**, which forces the model to distinguish between correct and incorrect representations.
- For every masked time step t, the model evaluates two key elements: its own prediction generated by the Transformer (the contextual representation c) and the actual true state generated by the Quantization Module (the quantized target q).

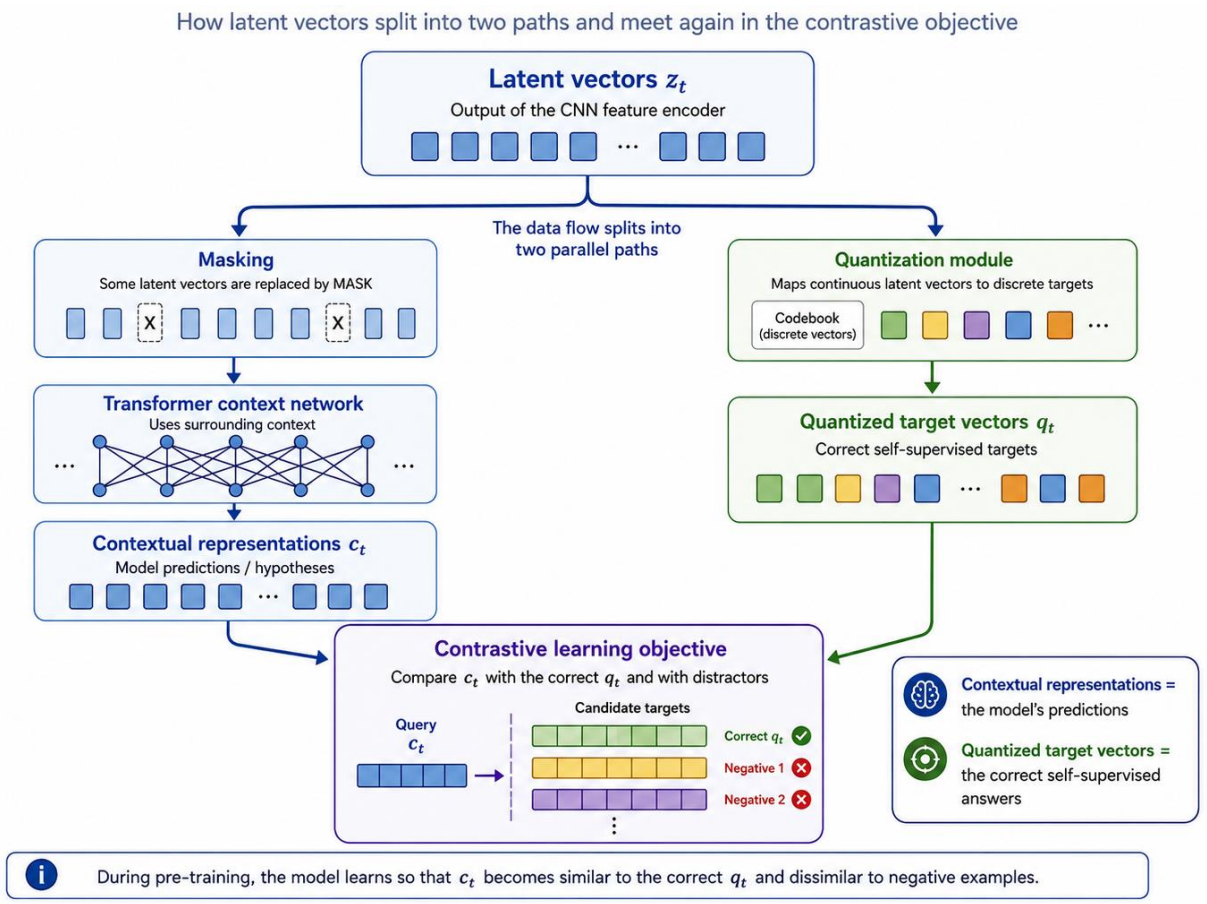


Fig. 8. Branching paths after the CNN encoder.

- **The Contrastive Task:** Instead of reconstructing the exact missing waveform from scratch, the model performs a discrimination task.
- It is presented with the correct quantized target (the positive example) alongside 100 randomly sampled vectors from other time steps in the same sequence (the distractors, or negative examples).
- **The Learning Goal:** Using cosine similarity, the model calculates the contrastive loss. The objective is to minimize this loss by pulling its contextual prediction c as close as possible to the true target q in the feature space, while actively pushing it away from the 100 negative distractors.
- This mechanism forces the network to learn robust, meaningful temporal dependencies and the underlying physics of the signal.

5.5 Detailed description – Step 2

Part 1 – Input data

- Unlike the pre-training phase, which relied solely on raw audio, the fine-tuning phase requires a supervised dataset (Fig. 9).
- The input still consists of the raw continuous waveforms, but they are now explicitly paired with corresponding ground-truth annotations (labels) required for the specific downstream task.

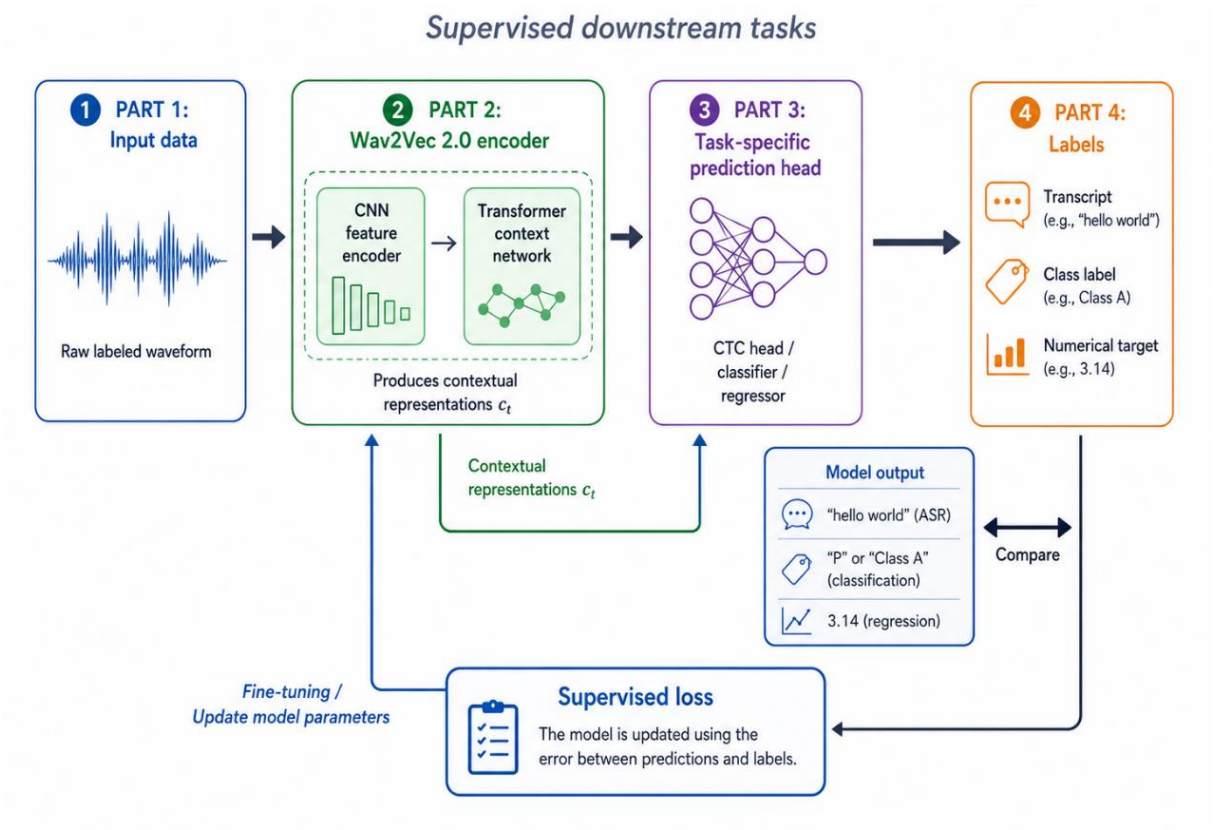


Fig. 9. Step 2: Wav2Vec 2.0 fine-tuning.

Part 2 – Pre-trained Wav2Vec-2.0 encoder

- This component is the exact combination of the CNN Feature Encoder and the Context Network (Transformer) optimized during Step 1.
- Instead of initializing from scratch, the architecture imports the pre-trained weights.
- Because these weights already encode a deep, contextual understanding of the signal's acoustic or physical properties, they serve as an exceptionally robust foundation, requiring significantly less labeled data to achieve high performance.

Part 3 – Task-specific prediction head

- A **decoder or prediction head** is the part of the model that converts internal representations into the final output.
- To transition to a supervised task, the self-supervised components (the Quantization Module and the Contrastive Loss function) are discarded.
- In their place, a specialized prediction head (or decoder) is attached to the top of the Transformer.
- This head is the part of the model that converts the internal, high-dimensional contextual representations into the final, human-readable output.
- Depending on the application, it might be a linear layer utilizing a CTC (Connectionist Temporal Classification) loss for speech recognition, or a standard classifier/regressor for categorizing signal states.
- CTC is a loss function used when the model must predict a sequence, such as text, from an input signal, but the exact alignment between input time steps and output tokens is unknown.

- It allows Wav2Vec-2.0 to learn speech recognition from waveform-transcript pairs without manually marking where each character or word occurs in the audio.

Part 4 – Labels

- Labels provide the explicit targets used to guide and refine the model's pre-trained weights.
- The exact nature of these labels dictates the architecture of the prediction head in Part 3.
- Examples include exact text transcripts for Automatic Speech Recognition (ASR), discrete class categories for classification tasks (e.g., identifying a specific speaker or a physical state like a seismic phase), or continuous numerical targets for regression analysis.
- The model calculates the error between its output and these labels to fine-tune its final predictions.

6. SUMMARY

The last four years have seen a global boom in artificial intelligence. Among professionals, such as software developers and scientists, this trend has been ongoing for over a decade. However, the theoretical foundations of many artificial intelligence architectures date back several decades. Among the architectures discussed in this handbook, the Multilayer Perceptron with backpropagation was popularized by Rumelhart et al. (1986), 40 years ago. Long short-term memory, a recurrent neural network architecture developed to address the vanishing gradient problem, was introduced by Hochreiter and Schmidhuber (1997); it is now 29 years old. The classic Convolutional Neural Network architecture (the LeNet-5 network by Yann LeCun et al.) was proposed in 1998 and is currently 28 years old. The most recent of these is wav2vec 2.0, published by Baevski et al. in 2020, making it 6 years old.

Importantly, all these architectures remain in active use. In practice, implementations may vary across software libraries and applications, and the code may require updates as libraries evolve. However, the fundamental principles, concepts, and terminology have remained consistent for many years. They continue to be used in current scientific research. The following section presents practical applications of these architectures in recent research articles.

There is currently talk of a renaissance of deep Multilayer Perceptrons (MLPs). This is because they serve as key components in modern hybrid models. Even in their basic form, however, MLPs still have numerous applications. For instance, in a study by Lehmann et al.(2023), the authors trained an MLP to predict low-frequency ground-motion parameters from earthquake source parameters and the relative positions of the source and observation site. For the largest magnitudes considered, the trained MLP predicted ground-motion parameters more than 1,000 times faster than the waveform simulation procedure used in the study. Klimasewski et al. (2021) used an MLP to predict peak ground acceleration, a widely used measure of ground-motion strength in seismic hazard assessment. The MLP learned the relationship between earthquake magnitude, hypocentral distance, site characteristics, and peak ground acceleration without prescribing a specific regression equation. In a laboratory experiment, Borate et al. (2023) used laboratory friction experiments to reproduce earthquake-like slip events. Ultrasonic waves were transmitted through the fault model; their velocity and amplitude varied according to the state of the rocks. The MLP used these measurements to estimate fault shear stress and slip rate. The authors also incorporated physical relationships into the MLP loss function, improving predictions when training data were limited and facilitating transfer between experiments.

LSTM networks are designed to learn dependencies in sequential data, such as seismograms. Wang et al. (2023) predicted peak ground acceleration for earthquake early warning. The initial waves reaching a seismic station provide information about an earthquake before the strongest shaking occurs. The authors used an LSTM to analyze how the characteristics of the recorded signal evolve over successive seconds, in order to predict the peak ground acceleration at a given location. Shokouhi et al. (2021) predicted the time remaining until earthquake-like slip events in laboratory friction experiments. In a laboratory fault model, the authors measured changes in ultrasonic waves passing through rock samples. An LSTM analyzed the history of these changes to estimate the shear stress acting on the fault and the time remaining until a sudden slip event. Yu et al. (2024) assessed changes in rock permeability based on microseismic events. Injecting water into rock formations can create new fractures or reactivate existing ones, potentially generating small earthquakes. The authors used a bidirectional LSTM network to predict changes in rock permeability from the rate of microseismic events and a cumulative measure of their size based on seismic moment.

CNNs are used to identify patterns in images, spatial data, and time series. Many types of geophysical data, such as 1D seismograms, 2D spectrograms or 3D seismic volumes, can be processed using convolutional methods adapted to their dimensionality and physical meaning. Laurenti et al. (2024) investigated changes in fault-zone properties using seismic recordings. Changes in rock cracking can affect seismic wave propagation. Using recordings associated with the 2016 Norcia earthquake, the authors trained a CNN to distinguish between seismograms recorded before and after the mainshock, as well as classes defined by time to failure, achieving test classification accuracies above 90%. Meanwhile, Yano et al. (2021) focused on detecting earthquakes despite the presence of local noise. Since noise recorded at a single station can resemble an earthquake signal, the authors developed a CNN that analyzed recordings from multiple stations simultaneously, taking their spatial distribution into account. When applied to data from the Tokyo region, the model detected over 97% of the events listed in the reference earthquake catalog, with a false-positive rate of less than 4%. Thomas et al. (2021) worked on identifying very weak earthquakes. Low-frequency earthquakes, associated with slow fault slip, are often buried in noise and difficult to detect. The authors trained a CNN to recognize the characteristic signals of these events within continuous data recorded along the San Andreas Fault. The network was also able to detect these weak events at stations whose data had not been used during training.

Significant progress in seismic signal processing is currently being driven by Transformer-based architectures. From a data-processing perspective, raw seismic recordings and audio recordings both represent time-varying signals and can share features relevant to machine learning. This similarity has motivated the effective adaptation of models originally developed for automatic speech recognition and natural language processing. These models use self-attention to learn relationships between different parts of the input window, including parts that are far apart in time. For instance, Harsuko and Alkhalifah (2022) applied a BERT-like model to signal processing. The traditional deep-learning approach in geophysics involved training separate networks for distinct tasks (such as denoising, event localization, and wave-velocity estimation) often requiring separate labeled datasets and repeated training for each task. In contrast, the Transformer used in this work was first pretrained on unlabeled data and could then be fine-tuned, with limited additional training, to perform a wide range of tasks, from denoising and velocity estimation to first-arrival picking. Sheng et al. (2025) proposed the Seismic Foundation Model (SFM). A foundation model is a large network pretrained on broad, unlabeled data and subsequently adapted to many specific tasks with relatively little labeled data. A common limitation in the geophysical interpretation of 2D/3D structures is a lack of generalizability; a model trained on data from one region is often of little use in another. In this study, the model was trained on a massive dataset comprising over 2

million seismic images, extracted from 192 3D data volumes collected from across the globe. The Transformer-based model was pretrained in a self-supervised manner by reconstructing masked parts of seismic images and was then adapted to tasks such as seismic facies classification, geobody identification, interpolation, denoising, and inversion, outperforming task-specific networks. Another compelling approach was proposed by Johnson et al. (2025). The authors applied the wav2vec 2.0 framework to continuous seismic waveforms from the Kīlauea caldera collapse, pretrained the model on the seismic waveforms, and then fine-tuned it to predict surface displacement. The model was to map continuous seismic signals to contemporaneous GNSS surface displacement, treated as a proxy for fault displacement. They used data from a unique natural experiment: the 2018 Kīlauea caldera collapse to predict a displacement value.

The brief literature review presented above demonstrates that even well-established deep learning architectures - such as MLP, LSTM, or CNN - continue to be utilized in world-class scientific research. Meanwhile, wav2vec 2.0, which represents a Transformer-based architecture, serves as an example of the groundbreaking, innovative solutions that have emerged in recent years and rapidly established themselves at the forefront of scientific inquiry. The author of this handbook hopes that the material presented here will serve as a solid introduction to the field of deep learning, preparing readers to study more advanced textbooks and tackle more complex applications.

Acknowledgments

This handbook was prepared within the statutory activities of the Institute of Geophysics, Polish Academy of Sciences (IG PAS).

The author is grateful to the anonymous reviewers for their valuable comments and to the Scientific Information and Publishers Department of IG PAS for the editorial preparation and typesetting of this handbook.

References

Baevski, A., Y. Zhou, A. Mohamed, and M. Auli (2020), wav2vec 2.0: A framework for self-supervised learning of speech representations, *Adv. Neur. In. Process. Syst*. **33**, 12449–12460, https://proceedings.neurips.cc/paper/2020/hash/92d1e1eb1cd6f9fba3227870bb6d7f07-Abstract.html.

Bishop, C.M. (2006), *Pattern Recognition and Machine Learning*, Springer Science + Bussiness Media, 738 pp.

Bishop, C.M., and H. Bishop (2023), *Deep Learning: Foundations and Concepts*, Springer Nature, Cham, DOI: 10.1007/978-3-031-45468-4.

Borate, P., J. Rivière, C. Marone, A. Mali, D. Kifer, and P. Shokouhi (2023), Using a physics-informed neural network and fault zone acoustic monitoring to predict lab earthquakes, *Nat. Commun*. **14**, 3693, DOI: 10.1038/s41467-023-39377-6.

Chollet, F., and M. Watson (2025), *Deep Learning with Python, Third Edition*, Manning Publications, 648 pp.

Devlin, J., M.-W. Chang, K. Lee, and K. Toutanova (2019), BERT: Pre-training of deep bidirectional transformers for language understanding. **In:** *Proc. 2019 Conf. North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Vol. 1 (Long and Short Papers), Association for Computational Linguistics, 4171–4186, DOI: 10.18653/v1/N19-1423.

Géron, A. (2022), *Hands-On Machine Learning with Scikit-Learn, Keras & Tensorflow: Concepts, Tools, and Techniques to Build Intelligent Systems*, 3rd ed., O'Reilly Media, Sebastopol.

Goldberg, Y. (2017), *Neural Network Methods for Natural Language Processing*, Synthesis Lectures on Human Language Technologies, No. 37, Springer, Cham, DOI: 10.1007/978-3-031-02165-7.

Goodfellow, I., Y. Bengio, and A. Courville (2016), *Deep Learning*, MIT Press, Cambridge, 800 pp.

Harsuko, R., and T.A. Alkhalifah (2022), StorSeismic: A new paradigm in deep learning for seismic processing, *IEEE Tran. Geosci. Remote Sens.* **60**, 5921915, DOI: 10.1109/TGRS.2022.3216660.

Haykin, S. (2008), *Neural Networks and Learning Machines*, 3rd ed., Pearson Education India, 944 pp.

Hochreiter, S., and J. Schmidhuber (1997), Long short-term memory, *Neural Comput.* **9**, 8, 1735–1780, DOI: 10.1162/neco.1997.9.8.1735

Johnson, C.W., K. Wang, and P.A. Johnson (2025), Automatic speech recognition predicts contemporaneous earthquake fault displacement, *Nat. Commun*. **16**, 1069, DOI: 10.1038/s41467-025-55994-9.

Klimasewski, A., V. Sahakian, and A. Thomas (2021), Comparing artificial neural networks with traditional ground-motion models for small-magnitude earthquakes in Southern California, *Bull. Seismol. Soc. Am*. **111**, 3, 1577–1589, DOI: 10.1785/0120200200.

Laurenti, L., G. Paoletti, E. Tinti, F. Galasso, C. Collettini, and C. Marone (2024), Probing the evolution of fault properties during the seismic cycle with deep learning, *Nat. Commun*. **15**, 10025, DOI: 10.1038/s41467-024-54153-w.

LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner (1998), Gradient-based learning applied to document recognition, *Proc. IEEE* **86**, 11, 2278–2324, DOI: 10.1109/5.726791.

LeCun, Y., Y. Bengio, and G. Hinton (2015), Deep learning, *Nature* **521**, 7553, 436–444, DOI: 10.1038/nature14539.

Lehmann, L., M. Ohrnberger, M. Metz, and S. Heimann (2023), Accelerating low-frequency ground motion simulation for finite fault sources using neural networks, *Geophys. J. Int*. **234**, 3, 2328–2342, DOI: 10.1093/gji/ggad239.

Liu, Y. (2020), *Python Machine Learning by Example: Build Intelligent Systems using Python, TensorFlow 2, PyTorch, and Scikit-learn*, 3rd ed., Packt Publishing Ltd, Birmingham.

Mohamed, A., H.-Y. Lee, L. Borgholt, J.D. Havtorn, J. Edin, C. Igel, K. Kirchhoff, S.-W. Li, K. Livescu, L. Maaløe, T.N. Sainath, and S. Watanabe (2022), Self-supervised speech representation learning: A review, *IEEE J. Sel. Top. Signal Process.* **16**, 6, 1179–1210, DOI: 10.1109/JSTSP. 2022.3207050.

Mousavi, S.M., and G.C. Beroza (2022), Deep-learning seismology, *Science* **377**, 6607, DOI: 10.1126/science.abm4470.

Mousavi, S.M., W.L. Ellsworth, W. Zhu, L.Y. Chuang, and G.C. Beroza (2020), Earthquake transformer—an attentive deep-learning model for simultaneous earthquake detection and phase picking, *Nat. Commun*. 11, 3952, DOI: 10.1038/s41467-020-17591-w.

Murphy, K.P. (2023), *Probabilistic Machine Learning: Advanced Topics*, MIT Press, Cambridge.

Prince, S.J.D. (2023), *Understanding Deep Learning*, MIT Press, 544 pp.

Raschka, S., and V. Mirjalili (2019), *Python Machine Learning: Machine Learning and Deep Learning with Python, Scikit-Learn, and TensorFlow 2*, 3rd ed., Packt Publishing, Birmingham.

Rumelhart, D.E., G.E. Hinton, and R.J. Williams (1986), Learning representations by back-propagating errors, *Nature* **323**, 6088, 533–536, DOI: 10.1038/323533a0.

Schneider, S., A. Baevski, R. Collobert, and M. Auli (2019), wav2vec: Unsupervised pre-training for speech recognition. **In:** *Proc. Interspeech 2019*, International Speech Communication Association, 3465–3469, DOI: 10.21437/Interspeech.2019-1873.

Sheng, H., X. Wu, X. Si, J. Li, S. Zhang, and X. Duan (2025), Seismic foundation model: A next generation deep-learning model in geophysics, *Geophysics* **90**, 2, IM59–IM79, DOI: 10.1190/geo2024-0262.1.

Shokouhi, P., V. Girkar, J. Rivière, S. Shreedharan, C. Marone, C.L. Giles, and D. Kifer (2021), Deep learning can predict laboratory quakes from active source seismic data, *Geophys. Res. Lett*. **48**, 12, e2021GL093187, DOI: 10.1029/2021GL093187.

Thomas, A.M., A. Inbal, J. Searcy, D.R., Shelly, and R. Bürgmann (2021), Identification of low-frequency earthquakes on the San Andreas fault with deep learning, *Geophys. Res. Lett*. **48**, 13, e2021GL093157, DOI: 10.1029/2021GL093157.

Tunstall, L., L. von Werra, and T. Wolf (2022), *Natural Language Processing with Transformers*, O'Reilly Media Inc., 408 pp

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, Ł. Kaiser, and I. Polosukhin (2017), Attention is all you need, *Adv. Neur. Int. Process. Syst.* **30**, 5998–6008, https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Wang, A., S. Li, J. Lu, H. Zhang, B. Wang, and Z. Xie (2023), Prediction of PGA in earthquake early warning using a long short-term memory neural network, *Geophys. J. Int.* **234**, 1, 12–24, DOI: 10.1093/gji/ggad067.

Yano, K., T. Shiina, S. Kurata, A. Kato, F. Komaki, S. Sakai, and N. Hirata (2021), Graph-partitioning based convolutional neural network for earthquake detection using a seismic array, *J. Geophys. Res.: Solid Earth* **126**, 5, e2020JB020269, DOI: 10.1029/2020JB020269.

Yu, P., A. Mali, T. Velaga, A. Bi, J. Yu, C. Marone, P. Shokouhi, and D. Elsworth (2024), Crustal permeability generated through microearthquakes is constrained by seismic moment, *Nat. Commun*. **15**, 2057, DOI: 10.1038/s41467-024-46238-3.

Zhang, A., Z.C. Lipton, M. Li, and A.J. Smola (2023), *Dive Into Deep Learning*, Cambridge University Press, 547 pp.

Received 27 August 2026
Accepted in revised form 16 November 2026
Accepted 19 November 2026

"Publications of the Institute of Geophysics, Polish Academy of Sciences: Geophysical Data Bases, Processing and Instrumentation" appears in the following series:

A – Physics of the Earth's Interior

B – Seismology

C – Geomagnetism

D – Physics of the Atmosphere

E – Hydrology (formerly Water Resources)

P – Polar Research

M – Miscellanea

Every volume has two numbers: the first one is the consecutive number of the journal and the second one (in brackets) is the current number in the series.