The Art of AI

NITIN DH

BlueRose ONE .com
Stories Matter
New Delhi • London

BLUEROSE PUBLISHERS
India | U.K.

For permissions requests or inquiries regarding this publication,
please contact:

BLUEROSE PUBLISHERS
www.BlueRoseONE.com
info@bluerosepublishers.com
+91 8882 898 898
+4407342408967

ISBN: 978-93-7825-612-7

Cover Design: Deepak Katheria
Typesetting: Manisha Rawat

First Edition: April 2026

Dedication

This book is dedicated to every learner who has ever looked at Artificial Intelligence and thought, "Can I really understand this?" The answer is yes.

It is also dedicated to teachers, mentors, engineers, innovators, and curious minds who believe that knowledge becomes most powerful when it is shared in a simple, practical, and meaningful way.

Acknowledgements

This book is the result of a meaningful journey shaped by curiosity, discipline, practice, mentorship, and continuous learning. It reflects not only my own efforts to understand Artificial Intelligence, but also the guidance and support of many people, institutions, and experiences that strengthened me along the way. I offer my sincere gratitude to all who contributed to this journey and helped deepen my understanding of AI and its real-world value.

My professional experience within **IBM** has been one of the strongest foundations of this growth. It gave me an environment where innovation, experimentation, and solution thinking are encouraged, and where learning is closely connected to real-world impact. Through this experience, I gained both technical knowledge and a deeper appreciation of how AI can be applied responsibly and effectively to solve client challenges.

I would like to express my special gratitude to **Mr. Surjit Das**, my senior at IBM, whose regular guidance and mentorship have played an important role in my AI journey. His encouragement, practical insights, and steady support helped me strengthen my understanding of AI and apply it in real client projects. I remain truly thankful for the confidence and direction his guidance gave me.

I am also grateful for the structured learning and inspiration I received through the **MIT AI Product Development programs**, **LearnBay**, the ideas inspired by the **MIT Media Lab**, and the practical educational contributions of **Jason Brownlee**. Each of these influences supported my growth in different ways—through fundamentals, product thinking, hands-on practice, and disciplined learning.

I also thank the many colleagues, mentors, domain experts, and learners whose discussions, collaboration, and feedback helped shape the practical teaching style of this book.

Above all, I thank every learner who believes that even the most advanced technology can be understood when explained with clarity,

patience, and purpose. That belief has been one of the strongest motivations behind writing this book.

For all the support, trust, encouragement, and inspiration I have received throughout this journey, I offer my heartfelt thanks.

About the Author

Nitin D.H. Patil is a seasoned Technology Architect, AI Product Developer, and Solution Designer with 15 years of professional experience, currently working with IBM. Over the course of his career, he has built strong expertise across multiple domains, including Real Estate, Telecommunications, and Banking, delivering technology-led solutions that align business needs with modern enterprise architecture.

He specializes in AI product development, solution design, enterprise architecture, and innovation-led technology transformation, with a strong focus on building scalable, practical, and business-driven AI solutions. His professional work combines domain knowledge, architectural thinking, and hands-on AI application design to solve complex business problems and accelerate digital transformation.

Nitin is passionate about simplifying complex AI concepts and helping learners, engineers, and organizations understand how to apply Artificial Intelligence in a practical, responsible, and value-driven way.

Who This Book Is For

This book is written for learners who want to understand Artificial Intelligence in a practical and structured way.

It is especially useful for beginners who are new to AI, Machine Learning, and Deep Learning; software engineers who want to move into AI and Generative AI; data professionals who want a broader applied understanding of AI systems; architects and solution designers who want to understand modern AI platforms; students preparing for AI-focused careers; and working professionals who want to build real-world AI solutions.

This book is not designed only for researchers or advanced mathematicians. It is designed for serious learners who want clarity, practicality, and confidence.

How to Use the Code and Download Material

This book is designed to be both readable and executable. Many chapters include practical examples, Python code, small datasets, and hands-on exercises so that readers can move from concept to implementation.

To use the code effectively, read the concept first and understand the problem the example is solving. Copy the code into your preferred Python environment such as Jupyter Notebook, VS Code, or any Python IDE. Run the example as provided before making changes. Once the result is clear, experiment by changing the data, prompts, parameters, or model settings. Build your own small variations to deepen understanding.

If supporting code files, datasets, or extended examples are made available separately, they should be organized chapter by chapter so that readers can easily locate and run the relevant material.

The goal of the code sections is not only to demonstrate syntax, but to help you build the habit of learning by implementation.

Edition and Version Note

This book reflects the author's practical view of Artificial Intelligence, Machine Learning, Deep Learning, and Generative AI as of the current edition. Because AI is a fast-moving field, some tools, frameworks, APIs, and model capabilities may evolve over time. However, the foundational concepts, learning structure, and practical reasoning presented in this book are designed to remain relevant beyond any single tool version.

Readers are encouraged to treat this book as a strong conceptual and implementation foundation, and to continue exploring the latest updates in AI platforms, frameworks, and responsible AI practices as the field grows.

Final Thoughts: Where Your AI Journey Goes Next

Artificial Intelligence is not just a subject to be studied. It is a field to be practiced, questioned, improved, and applied with responsibility. If you have reached this point in the book, you have already completed an important journey. You have moved from foundational ideas such as data, statistics, and Machine Learning into deeper areas like neural networks, modern AI systems, and practical Generative AI. That is not a small achievement.

One of the most important lessons in AI is that learning never really ends. The field keeps evolving. New models appear, tools change, frameworks improve, and business expectations grow. But even though the technology keeps changing, the real foundations remain the same: clear thinking, good data, correct problem definition, strong evaluation, and responsible system design. These are the principles that will continue to guide you no matter how the tools evolve.

This book was written to help you build that foundation. Its purpose was never only to explain theory. Its purpose was to help you understand how AI works, why it matters, and how to apply it in a meaningful way. The goal was to make AI less intimidating and more practical. If this book has helped you move from confusion to clarity, from hesitation to confidence, and from theory to experimentation, then it has achieved its purpose.

At this stage, many readers ask an important question: what should I do next? The answer depends on your interest, your background, and the kind of work you want to do. Some readers will enjoy building models and tuning performance. Some will enjoy working with data and extracting insights. Some will be more excited by Generative AI applications, copilots, RAG systems, and enterprise assistants. Others will find their strength in architecture, solution design, and building complete AI systems that operate safely at scale.

What matters most is that you continue with intention. Build small projects. Rework the examples from this book. Change the data. Experiment with prompts. Compare models. Try local LLMs. Build a

small retrieval system. Create a mini AI assistant for your own domain. The strongest learning happens when ideas move from reading into implementation.

At the same time, remember that AI is not only about intelligence. It is also about responsibility. A good AI practitioner does not only ask, 'Can I build this?' A good AI practitioner also asks, 'Should I build this this way? Is the data reliable? Is the output fair? Is the system explainable? Is human review needed?' The future of AI will not be shaped only by bigger models. It will be shaped by thoughtful builders who understand both capability and consequence.

This is why The Art of AI is more than a technical title. AI is not only science, code, or mathematics. It is also judgement, design, ethics, communication, and purpose. The real art of AI lies in building systems that are not only powerful, but also useful, trustworthy, and aligned with real human needs.

So as you close this book, do not think of it as the end of your learning. Think of it as the beginning of your real journey. Keep learning. Keep experimenting. Keep building. Keep questioning. And most importantly, use AI to create solutions that are meaningful, responsible, and valuable.

Your journey into the art of AI has only just begun.

How to Continue Learning After This Book

Once you complete this book, the next step should be structured practice rather than random exploration. The field of AI is large, and many learners feel overwhelmed because they try to learn everything at once. A better approach is to choose a direction and build depth step by step.

Start by revisiting the chapters that felt most challenging. Rebuild the examples in your own environment. Do not just read the code—run it, change it, break it, fix it, and observe the output. Once you are comfortable, begin building small end-to-end projects. A basic classifier, a regression model, an image recognizer, a prompt-based assistant, or a simple RAG system can teach more than passive theory alone.

After that, choose a path based on your strengths and interests. If you enjoy working closely with data, understanding patterns, and building predictive models, you may want to move toward Machine Learning or Data Science. If you enjoy deep learning, neural networks, computer vision, or NLP, then applied AI engineering may be a better direction. If you are excited by chatbots, copilots, agents, LLMs, retrieval systems, and enterprise assistants, then Generative AI engineering is a strong path. If you enjoy designing complete systems, making architectural decisions, connecting business goals with technical platforms, and building scalable solutions, then AI architecture may be right for you.

You do not need to decide everything immediately. What matters is steady progress. Learn, build, evaluate, and improve. Consistency is much more powerful than occasional bursts of intensity.

Which AI Path Should You Choose?

Artificial Intelligence offers many career paths, and the right one depends on what kind of work gives you energy.

Machine Learning Engineer: This path is ideal for someone who enjoys building, training, improving, and deploying predictive models. A Machine Learning Engineer works on model pipelines, feature engineering, evaluation, performance tuning, and production deployment.

Data Scientist: This path is suitable for someone who enjoys working deeply with data, discovering patterns, generating insights, testing hypotheses, and translating business questions into analytical findings.

Generative AI Engineer: This path is best for someone interested in modern LLM-based systems such as copilots, document assistants, prompt workflows, retrieval-based applications, semantic search, and AI productivity tools.

AI Architect: This path is for those who enjoy system thinking at a larger level. An AI Architect focuses on end-to-end solution design, platform decisions, scalability, security, integration, governance, and business alignment.

AI Product Builder / AI Solution Designer: This path is suitable for someone who enjoys identifying business problems, designing AI-enabled products, defining solution workflows, planning user experience, and translating ideas into useful applications.

A simple way to choose is this: if you love models, move toward Machine Learning. If you love data and insights, move toward Data Science. If you love LLMs and AI applications, move toward Generative AI. If you love system design and enterprise thinking, move toward AI Architecture.

A 30-60-90 Day Roadmap After This Book

First 30 Days: Strengthen the Foundation. Spend the first 30 days revisiting the core concepts. Rework examples from statistics, Machine Learning, Deep Learning, and Generative AI. Focus on understanding rather than speed. Create a notebook for each concept. Run the code. Write small notes in your own words. Build one tiny project such as house price prediction, spam classification, or a meeting summarizer.

Next 60 Days: Build Practical Projects. In the next phase, start building end-to-end projects. Create one traditional Machine Learning project and one Generative AI project. For example, you might build a customer churn predictor and a small document Q&A assistant. Learn how to structure code, store data, test outputs, and present results. Try both cloud APIs and local model execution.

Next 90 Days: Become Role-Oriented. By this stage, start aligning your work with the role you want. If you want to become an ML Engineer, focus on pipelines, tuning, and deployment. If you want to become a Data Scientist, focus on exploratory analysis and business storytelling. If you want to become a GenAI Engineer, focus on RAG, embeddings, prompt design, hallucination control, and evaluation. If you want to become an AI Architect, study end-to-end system design, governance, integration, security, and scalability.

The purpose of this roadmap is not to rush you. The purpose is to help you move with direction.

Final Words from the Author

This book was written with a simple belief: AI should be understandable, practical, and accessible to every serious learner. Too often, Artificial Intelligence is presented in a way that feels intimidating, overly theoretical, or disconnected from real work. My intention was to make the subject more human, more structured, and more useful.

I believe that AI is one of the most important disciplines of our time, but I also believe that it should be learned with balance. It is not enough to know the technology. We must also understand the context in which it is used, the consequences it may create, and the responsibility that comes with building intelligent systems.

If this book has helped you think more clearly, build more confidently, and approach AI with both curiosity and discipline, then I will consider that a meaningful success. Keep going. Keep learning. Keep creating. And most importantly, use AI in a way that adds genuine value to people, organizations, and society.

Further Reading and Learning References

The learning journey behind this book has been shaped by a combination of practical experience, professional programs, applied AI communities, technical study, and implementation-focused resources. Readers who wish to go deeper may explore the following types of sources for continued learning:

Professional and Organizational Learning: IBM AI learning initiatives and practical enterprise AI exposure; MIT AI Product Development learning programs; LearnBay AI and Data Science learning ecosystem; MIT Media Lab ideas, research influence, and innovation thinking.

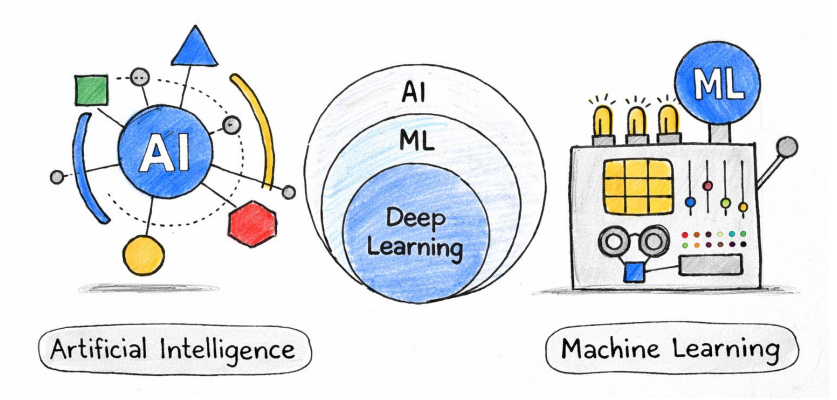
Practical AI and Implementation-Focused Learning: Jason Brownlee's mastery-oriented AI and Machine Learning books; hands-on Python-based tutorials for ML, DL, and applied AI; project-based exercises and implementation notebooks; framework documentation for TensorFlow, PyTorch, scikit-learn, and modern LLM tooling.

Suggested Reading Directions: foundational statistics and probability for AI; supervised and unsupervised Machine Learning; neural networks and Deep Learning; NLP and transformers; Generative AI and LLM applications; prompt design and RAG systems; Responsible AI, fairness, security, and governance; solution architecture for enterprise AI systems.

This book encourages readers to combine theory, experimentation, and practical implementation. The strongest learning often happens when ideas are tested through small working projects.

AI is the theory and development of computer systems able to perform tasks normally requiring human intelligence.

ML gives computers the ability to learn without explicit programming.



AI is often taught as a collection of algorithms and tools. This book takes a different path.

It treats AI as an art — an art of thinking with data, reasoning with uncertainty, and designing intelligent systems responsibly.

From statistics and mathematics to machine learning, deep learning, and generative AI, this book focuses not only on how AI works, but on how it should be used.

How to Use This Book

This book is written for freshers and working professionals who want AI explained in plain, practical English. You don't need advanced math to start—you only need curiosity and a habit of practicing tiny examples.

The teaching pattern stays the same in every chapter:

Definition (plain words) → Syntax/Formula (only what you need) → Why? → What? → How? → Let's Code!

Best way to read:

1) Read the Definition + Why (2 minutes).
2) Look at the tiny dataset (so you can 'see' the problem).
3) Run the code exactly as-is once.
4) Then change ONE thing (a number, a feature, a parameter) and observe.

A note for learners:

If English feels heavy, read slowly and focus on the examples. In this book, every concept is connected to daily-life scenarios (banking, rent, shopping, exams, and apps). The goal is understanding, not fancy words.

Symbols and units you will see often:

• n = number of rows (samples) • x = input/feature • y = output/target • w, b = weight and bias • ₹ / INR used in examples where needed

Common mistakes (and how this book prevents them):

- Memorizing formulas without meaning → we use real stories first.
- Copy-paste coding without learning → we keep datasets tiny and editable.
- Training without checking data → we teach EDA and leakage early.

Master Contents

Part I - Statistics & Probability

Part II - Machine Learning

Part III - Deep Learning

Part IV – Generative AI

Statistics and Math are the base of AI—they help you understand data, measure uncertainty, and write the formulas models learn from. Machine Learning uses past examples to learn patterns and make predictions (spam, fraud, price). Deep Learning is ML with neural networks that can learn complex patterns from images, text, and speech. Generative AI is a type of deep learning that can create new content like text, code, images, and summaries.

Code Base – all the example that we have covered in this book will be available on - https://github.com/AINitin18/The-Art-of-AI.git

Contents

Part I — Statistics & Probability **1**

Statistics & Probability Made Simple for Machine Learning and Deep Learning 2

Running Datasets Used Across the Book 3

Chapter 1 Why Statistics & Probability matter in ML/DL 4

Chapter 2 Data, variables, and sampling 10

Chapter 3 Exploratory Data Analysis (EDA): your first superpower 18

Chapter 4 Box plots, outliers, and robust thinking 33

Chapter 5 Central tendency and dispersion (mean, median, variance, SD) 36

Chapter 6 Distributions: Normal, z-score, and standardization 43

Chapter 7 Probability basics: events, conditional probability, independence 47

Chapter 8 Bayes' theorem and Naive Bayes intuition 49

Chapter 9 Binomial and Poisson: counting-world distributions 52

Chapter 10 Estimation and confidence intervals (CI) 55

Chapter 11 Hypothesis testing and A/B tests (p-value, errors) 57

Chapter 12 Correlation, covariance, and the road to linear regression 60

Chapter 13 ANOVA and when many groups are involved 62

Chapter 14 Common mistakes in ML statistics (leakage, bias, overconfidence) 65

Chapter 15 Mini Projects + Interview Questions + Formula Sheet + Glossary 68

Part II — Machine Learning 72

Chapter 1 Machine Learning Simple Words 74

Chapter 2 The ML Workflow (Data → Model → Prediction) 79

Chapter 3 Data Basics (Features, Target, Leakage) 82

Chapter 4 Evaluation Metrics
(How to know - how model is good?) 85

Chapter 5 Decision Trees (CART) and Gini Impurity 88

Chapter 6 Random Forest (Bagging + Voting) 94

Chapter 7 Linear Regression (Predict a Number) 98

Chapter 8 Logistic Regression (Predict Yes/No) 109

Chapter 9 K-Nearest Neighbors (Distance Based) 113

Chapter 10 Support Vector Machines (Margin Based) 118

Chapter 11 Naïve Bayes (Probability Based) 122

Chapter 12 Clustering and K-Means (Unsupervised) 126

Chapter 13 PCA (Dimensionality Reduction) 130

Chapter 14 Practical Tips
(Encoding, Scaling, Imbalanced Data, CV) 134

Chapter 15 Production: Build an API +
Deploy on GCP Cloud Run 137

Chapter 16 Production: Deploy on AWS Lambda
(Container) + API Gateway 141

Appendix A Common Interview Questions
(With Simple Answers) 144

Part III — Deep Learning 147

Preface This Book is for Freshers 148

Chapter 0 Setup: Your Deep Learning Lab 150

Chapter 1 Before Learning Deep Learning,
You Must Know This 152

Chapter 2 Artificial Neural Networks (ANN) 166

Chapter 1.1 The Smallest Neural Network (One Neuron) 184

Chapter 3 Building Your First Neural Network 187

Chapter 4 What Happens in Training? 209

Chapter 5 Backpropagation and Gradient Descent (No Fear) 212

Chapter 6 Your First Deep Learning Model in TensorFlow & Keras 215

Chapter 7 How a Neural Network Learns Ex - House Prices (Weights, Biases, and Prediction) 223

Chapter 8 Hyperparameters That Control Learning 239

Chapter 9 House Price Prediction with Deep Learning 246

Chapter 10 ANN Regression Example (Real Estate Rent Prediction) 251

Chapter 11 ANN Classification Example (Banking Default Risk) 254

Chapter 11.1 TensorFlow & Keras - How They Work Inside, and How You Build Real AI Models 257

Chapter 12 CNN (Convolutional Neural Network) with MNIST 272

Chapter 13 RNN/LSTM (Sequence Learning) with a Tiny Example 292

Chapter 14 Transformers and Attention 301

Chapter 15 Reasoning Models (Tool-Augmented Reasoning for Beginners) 312

Chapter 16 Specialized & Emerging Neural Networks (Beyond ANN/CNN/RNN/Transformer) 315

Chapter 17 Production Deep Learning (API + Model Serving Basics) 336

Chapter 18 Deploy to AWS (Lambda Container + API Gateway) 339

IV - Practical Generative AI 343

Chapter 0 Generative AI: From Prediction to Creation 345

Chapter 1 Embeddings: Turning Meaning into Numbers 348

Chapter 2 Vector Databases: Searching by Meaning 351

Chapter 3 RAG: Retrieve First, Answer Second 356

Chapter 4 Prompt Design: Better Instructions, Better Output 360

Chapter 5 Hallucination Control: Reducing Confident Mistakes 363

Chapter 6 Evaluation of LLM Answers: Check Before You Trust 366

Chapter 7 Agents vs Workflows: Choosing the Right Architecture 370

Chapter 8 When Not to Use an LLM: Use Intelligence Wisely 374

Responsible AI: Building Powerful Systems with Care, Fairness, and Trust 376

Appendix A - Math Cheat Sheet (Only What You Use) 378

Appendix B — Debugging and Common Errors 380

Appendix C — Glossary 381

Part I — Statistics & Probability

This part builds the foundation you will use in Machine Learning and Deep Learning. Keep the tiny datasets handy and practice the code blocks.

Statistics & Probability Made Simple for Machine Learning and Deep Learning

Table of Contents

1. Why Statistics & Probability matter in ML/DL
2. Data, variables, and sampling (the ML way)
3. Exploratory Data Analysis (EDA): your first superpower
4. Box plots, outliers, and robust thinking
5. Central tendency and dispersion (mean, median, variance, SD)
6. Distributions: Normal, z-score, and standardization
7. Probability basics: events, conditional probability, independence
8. Bayes' theorem and Naive Bayes intuition
9. Binomial and Poisson: counting-world distributions
10. Estimation and confidence intervals (CI)
11. Hypothesis testing and A/B tests (p-value, errors)
12. Correlation, covariance, and the road to linear regression
13. ANOVA and when many groups are involved
14. Common mistakes in ML statistics (leakage, bias, overconfidence)
15. Mini projects + interview questions + formula sheet + glossary

Running Datasets Used Across the Book

To keep learning easy, we reuse the same tiny datasets again and again. You can copy-paste them into Python in 2 minutes.

Dataset A: Banking - Loan Default Risk (tiny)

Each row is one customer. Target = default (1) or not (0).

```
customer_id,age,income_k,loan_k,years_with_bank,missed_payments,defa
ult
C01,23,4.5,1.0,1,0,0
C02,45,12.0,6.0,12,1,0
C03,31,6.0,4.0,5,2,1
C04,26,5.0,2.5,2,0,0
C05,39,9.0,7.5,7,3,1
C06,52,15.0,5.0,18,0,0
C07,29,5.5,3.0,3,1,0
C08,34,7.0,6.5,6,2,1
C09,28,6.5,2.0,4,0,0
C10,41,10.0,8.0,10,4,1
```

Dataset B: Real Estate - Rent Prediction (tiny)

Target = monthly_rent_k (in thousands).

```
flat_id,area_sqft,bedrooms,distance_km_to_metro,age_years,monthly_re
nt_k
R01,550,1,0.8,8,18
R02,720,2,1.5,6,24
R03,900,2,3.0,12,22
R04,1100,3,2.0,5,35
R05,650,1,4.5,15,16
R06,800,2,0.7,3,30
R07,1200,3,5.0,10,28
R08,500,1,1.2,2,20
R09,950,2,2.5,7,27
R10,1400,3,0.9,4,42
```

Chapter 1
Why Statistics & Probability matter in ML/DL

Definition

Statistics is the skill of describing data and learning patterns from it. Probability is the skill of measuring uncertainty (how likely something is). In AI, both are used together: we learn patterns, then we talk honestly about risk and confidence.

In India example: A bank approves loans using past customer data (statistics) but still keeps a chance of default (probability). That 'chance' is what the model must estimate.

Syntax / Formula (Math in simple words)

Most AI problems can be written as: prediction = model(features).

Statistics helps you answer: what is typical, what is rare, and what is noisy.

Probability helps you answer: how confident are we, and what is the risk if we are wrong?

Why?

ML models learn patterns from data, not from magic. Statistics help us understand the data (shape, noise, outliers, bias). Probability helps us talk about uncertainty (risk, confidence, error).

What?

Statistics = learning from data.
Probability = measuring chance.
In ML, we use both every day: scaling, loss functions, evaluation, A/B tests, and model monitoring.

How?

A simple habit: before training any model, do EDA. Ask: Is the data clean? Are there outliers? Are classes imbalanced? Is the train/test split fair?

Let's Code!

```
# Minimal EDA Checklist (works for any ML project)
# 1. Shape and missing values
# 2. Target distribution
# 3. Numeric summary
# 4. Outlier scan
# 5. Leakage scan (target accidentally in features)


import pandas as pd
from io import StringIO


# Sample data
csv =
"""customer_id,age,income_k,loan_k,years_with_bank,missed_payments,d
efault
C01,23,4.5,1.0,1,0,0
C02,45,12.0,6.0,12,1,0
C03,31,6.0,4.0,5,2,1
C04,26,5.0,2.5,2,0,0
C05,39,9.0,7.5,7,3,1
C06,52,15.0,5.0,18,0,0
C07,29,5.5,3.0,3,1,0
C08,34,7.0,6.5,6,2,1
C09,28,6.5,2.0,4,0,0
C10,41,10.0,8.0,10,4,1
"""


df = pd.read_csv(StringIO(csv))


# 1. Shape and missing values
```

```
print("=== Dataset Shape ===")
print(f"Rows: {df.shape[0]}, Columns: {df.shape[1]}\n")

print("=== Missing Values ===")
print(df.isna().sum())
print()

# 2. Target distribution
print("=== Target Distribution (default) ===")
print(df['default'].value_counts())
print(df['default'].value_counts(normalize=True))
print()

# 3. Numeric summary
print("=== Numeric Summary Statistics ===")
print(df.describe())
print()

# 4. Outlier scan (using IQR method)
print("=== Outlier Detection (IQR Method) ===")
numeric_cols = df.select_dtypes(include=['float64',
'int64']).columns
for col in numeric_cols:
    Q1 = df[col].quantile(0.25)
    Q3 = df[col].quantile(0.75)
    IQR = Q3 - Q1
    outliers = df[(df[col] < Q1 - 1.5 * IQR) | (df[col] > Q3 + 1.5 *
IQR)]
    print(f"{col}: {len(outliers)} outliers detected")
print()

# 5. Leakage scan (target accidentally in features)
print("=== Feature Leakage Check ===")
print(f"Target column: 'default'")
print(f"Features: {[col for col in df.columns if col !=
'default']}")
```

Result

```
=== Dataset Shape ===
Rows: 10, Columns: 7


=== Missing Values ===
customer_id        0
age                0
income_k           0
loan_k             0
years_with_bank    0
missed_payments    0
default            0
dtype: int64


=== Target Distribution (default) ===
default
0    6
1    4
Name: count, dtype: int64
default
0    0.6
1    0.4
Name: proportion, dtype: float64


=== Numeric Summary Statistics ===
             age   income_k     loan_k  years_with_bank
missed_payments  \
count  10.000000  10.000000  10.000000        10.000000
10.000000
mean   34.800000   8.050000   4.550000         6.800000
1.300000
std     9.235198   3.411174   2.420399         5.223877
1.418136
min    23.000000   4.500000   1.000000         1.000000
0.000000
25%    28.250000   5.625000   2.625000         3.250000
0.000000
```

```
50%    32.500000   6.750000   4.500000          5.500000
1.000000
75%    40.500000   9.750000   6.375000          9.250000
2.000000
max    52.000000  15.000000   8.000000         18.000000
4.000000


         default
count  10.000000
mean    0.400000
std     0.516398
min     0.000000
25%     0.000000
50%     0.000000
75%     1.000000
max     1.000000


=== Outlier Detection (IQR Method) ===
age: 0 outliers detected
income_k: 0 outliers detected
loan_k: 0 outliers detected
years_with_bank: 0 outliers detected
missed_payments: 0 outliers detected
default: 0 outliers detected


=== Feature Leakage Check ===
Target column: 'default'
Features: ['customer_id', 'age', 'income_k', 'loan_k',
'years_with_bank', 'missed_payments']
```

Quick Exercises

Pick any dataset you know (banking/real estate). List 5 EDA questions you will ask first. Explain in 2 lines: why probability is needed in prediction confidence.

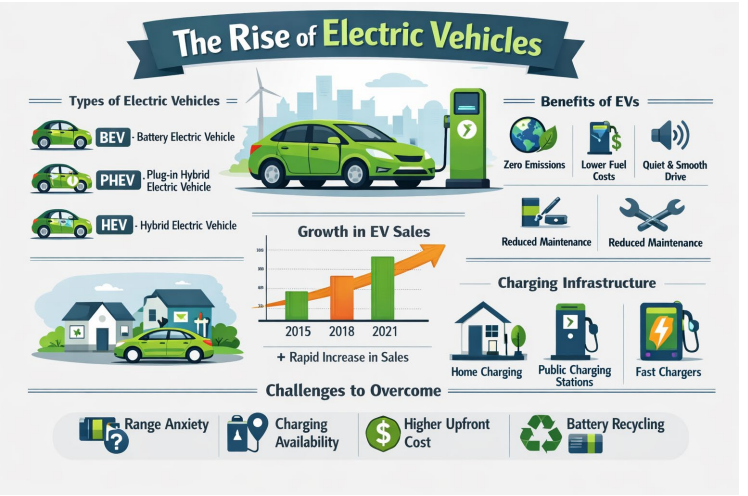
What is one risk of skipping EDA before training?

Chapter 2
Data, variables, and sampling

Definition

A variable is any measurable property (age, income, rent, marks). A dataset is a table of variables. Sampling means choosing a smaller set of rows from a big population, so you can learn without measuring everyone.

In India example: You cannot ask every voter in India, so pollsters sample a few thousand people and estimate the trend.



Syntax / Formula (Math in simple words)

Population (all cases) -> Sample (few cases) -> Learn pattern -> Use on population.

Common symbols: N = population size, n = sample size, x = feature, y = target.

Good sampling tries to be representative (same mix as the real world).

Why?

Bad data gives bad models. Sampling is how we decide what data to collect or label. If sampling is biased, the model becomes unfair or useless in production.

What?

Variable types:
- Quantitative: numbers (continuous or discrete)
- Qualitative: categories (nominal or ordinal)
Sampling connects a small dataset (sample) to the bigger real world (population).

How?

In ML projects, sampling happens at many places:
1) which customers you include
2) which time period you pick
3) which labels you trust
Stratified sampling is very useful for imbalanced targets (like fraud/default).

How this shows up in Machine Learning

Example: default rate may be only 2%. If you randomly sample small data, you might get almost no defaults. Then your model learns nothing. A stratified split keeps class ratio similar in train/test.

Let's Code!

```
import pandas as pd

from io import StringIO

from sklearn.model_selection import train_test_split


# Load data

csv =
"""customer_id,age,income_k,loan_k,years_with_bank,missed_payments,d
efault

C01,23,4.5,1.0,1,0,0

C02,45,12.0,6.0,12,1,0

C03,31,6.0,4.0,5,2,1
```

```
C04,26,5.0,2.5,2,0,0
C05,39,9.0,7.5,7,3,1
C06,52,15.0,5.0,18,0,0
C07,29,5.5,3.0,3,1,0
C08,34,7.0,6.5,6,2,1
C09,28,6.5,2.0,4,0,0
C10,41,10.0,8.0,10,4,1"""

df = pd.read_csv(StringIO(csv))

#
================================================================
========
# EDA CHECKLIST
#
================================================================
========

# 1. Shape and missing values
print("=" * 60)
print("1. DATASET OVERVIEW")
print("=" * 60)
print(f"Shape: {df.shape[0]} rows, {df.shape[1]} columns\n")
print("Missing values:")
print(df.isna().sum())
print()

# 2. Target distribution
print("=" * 60)
print("2. TARGET DISTRIBUTION")
print("=" * 60)
print(df['default'].value_counts(normalize=True).mul(100).round(2).a
stype(str) + '%')
print()

# 3. Numeric summary
print("=" * 60)
```

```
print("3. NUMERIC SUMMARY STATISTICS")
print("=" * 60)
# For older pandas versions, select numeric columns first
numeric_df = df.select_dtypes(include=['float64', 'int64'])
print(numeric_df.describe().round(2))
print()


# 4. Outlier scan (using IQR method)
print("=" * 60)
print("4. OUTLIER SCAN (IQR Method)")
print("=" * 60)
numeric_cols = df.select_dtypes(include=['float64',
'int64']).columns
for col in numeric_cols:
    Q1 = df[col].quantile(0.25)
    Q3 = df[col].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers = df[(df[col] < lower_bound) | (df[col] > upper_bound)]
    outlier_count = len(outliers)
    outlier_pct = (outlier_count / len(df)) * 100
    print(f"{col:20s}: {outlier_count:2d} outliers
({outlier_pct:.1f}%)")
print()


# 5. Leakage scan
print("=" * 60)
print("5. FEATURE LEAKAGE CHECK")
print("=" * 60)
print(f"Target column: 'default'")
features = [col for col in df.columns if col != 'default' and col !=
'customer_id']
print(f"Features: {', '.join(features)}")
print("✓ No target leakage detected (target column excluded from
features)")
```

```
print()


#
================================================================================
========
# TRAIN/TEST SPLIT
#
================================================================================
========


# Define features and target (exclude customer_id as it's an
identifier)
X = df[['age', 'income_k', 'loan_k', 'years_with_bank',
'missed_payments']]
y = df['default']


# Split with stratification to maintain class distribution
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.3,
    random_state=42,
    stratify=y
)


print("=" * 60)
print("TRAIN/TEST SPLIT RESULTS")
print("=" * 60)
print(f"Training set size: {len(X_train)} samples
({len(X_train)/len(df)*100:.0f}%)")
print(f"Test set size: {len(X_test)} samples
({len(X_test)/len(df)*100:.0f}%)\n")


print("Training set class distribution:")
print(y_train.value_counts(normalize=True).mul(100).round(2).astype(
str) + '%')
print()


print("Test set class distribution:")
```

```
print(y_test.value_counts(normalize=True).mul(100).round(2).astype(s
tr) + '%')

print("\n✓ Stratified split preserved class balance")
```

Result

```
============================================================

1. DATASET OVERVIEW

============================================================

Shape: 10 rows, 7 columns


Missing values:

customer_id        0

age                0

income_k           0

loan_k             0

years_with_bank    0

missed_payments    0

default            0

dtype: int64


============================================================

2. TARGET DISTRIBUTION

============================================================

default

0    60.0%

1    40.0%

Name: proportion, dtype: object


============================================================

3. NUMERIC SUMMARY STATISTICS

============================================================

         age  income_k  loan_k  years_with_bank  missed_payments
default

count  10.00     10.00   10.00            10.00            10.00
10.00
```

```
mean    34.80      8.05    4.55             6.80             1.30
0.40
std      9.24      3.41    2.42             5.22             1.42
0.52
min     23.00      4.50    1.00             1.00             0.00
0.00
25%     28.25      5.62    2.62             3.25             0.00
0.00
50%     32.50      6.75    4.50             5.50             1.00
0.00
75%     40.50      9.75    6.38             9.25             2.00
1.00
max     52.00     15.00    8.00            18.00             4.00
1.00


============================================================
4. OUTLIER SCAN (IQR Method)
============================================================
age                 :  0 outliers (0.0%)
income_k            :  0 outliers (0.0%)
loan_k              :  0 outliers (0.0%)
years_with_bank     :  0 outliers (0.0%)
missed_payments     :  0 outliers (0.0%)
default             :  0 outliers (0.0%)


============================================================
5. FEATURE LEAKAGE CHECK
============================================================
Target column: 'default'
Features: age, income_k, loan_k, years_with_bank, missed_payments
✓ No target leakage detected (target column excluded from features)


============================================================
TRAIN/TEST SPLIT RESULTS
============================================================
Training set size: 7 samples (70%)
Test set size: 3 samples (30%)
```

```
Training set class distribution:
default
0    57.14%
1    42.86%
Name: proportion, dtype: object

Test set class distribution:
default
0    66.67%
1    33.33%
Name: proportion, dtype: object
```

Quick Exercises

Explain stratified sampling using a fraud dataset example.

Give one case where convenience sampling can break an ML model.

What is the difference between nominal and ordinal categories? Give 2 examples.

Chapter 3
Exploratory Data Analysis (EDA): your first superpower

Definition

EDA means 'first look' at data to understand what you are working with: missing values, outliers, ranges, and target balance. EDA is not optional - it is the basic safety check before training a model.

example: Before buying a second-hand bike, you first check papers, engine sound, and service history. EDA is that first check for your data.

Syntax / Formula (Math in simple words)

EDA usually includes: (1) shape, (2) missing values, (3) summary stats, (4) plots, (5) leakage check, (6) split check.

Simple summaries: mean, median, min-max, standard deviation, and target counts.

Why?

EDA is like checking your ingredients before cooking. Without EDA, you may train a model on wrong ranges, wrong units, missing values, or outliers.

- Data Quality Issues: Real data is messy (missing values, outliers, errors)
- Pattern Discovery: Find hidden relationships before modeling
- Model Selection: Different data distributions need different ML algorithms
- Feature Engineering: Create better input for ML models
- Avoid Garbage In, Garbage Out: Bad data → Bad predictions

What?

EDA is understanding your data through:

Univariate Analysis: Understanding single variables

Bivariate Analysis: Understanding relationships between two variables

Multivariate Analysis: Understanding complex interactions

The EDA Mindset: Ask questions like a detective!

- Who are my customers?
- What patterns exist?
- Why do defaults happen?
- When should I be worried?
- How can I predict risk?

How?

Use simple tools: describe(), value_counts(), groupby(), histograms, box plots. You don't need fancy math to find 80% of problems.

How this shows up in Machine Learning

EDA answers: Do we need scaling? Should we log-transform income? Do we cap extreme loan values? Which features correlate with default?

Let's Code!

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
import io
plt.style.use('seaborn-v0_8-darkgrid')
sns.set_palette("husl")
```

Step 1: Load and Inspect

- Why: First impression of data - see what we're working with
- What: Check structure, data types, basic info
- How: Pandas function
- ML Connection: Determines preprocessing needed

```
csv_data =
"""customer_id,age,income_k,loan_k,years_with_bank,missed_payments,d
efault
C01,23,4.5,1.0,1,0,0
C02,45,12.0,6.0,12,1,0
C03,31,6.0,4.0,5,2,1
C04,26,5.0,2.5,2,0,0
C05,39,9.0,7.5,7,3,1
C06,52,15.0,5.0,18,0,0
C07,29,5.5,3.0,3,1,0
C08,34,7.0,6.5,6,2,1
C09,28,6.5,2.0,4,0,0
C10,41,10.0,8.0,10,4,1
"""
# Load data using StringIO from io module
df = pd.read_csv(io.StringIO(csv_data))  # Fixed: using io.StringIO
print("="*60)
print("STEP 1: DATA INSPECTION")
print("="*60)
```

```
# Quick overview
print("\n FIRST LOOK AT DATA:")
print(f"Shape: {df.shape}  (Rows: {df.shape[0]}, Columns:
{df.shape[1]})")
print(f"\nFirst 5 rows:")
print(df.head())
print(f"\nData Types:")
print(df.dtypes)
print(f"\nBasic Info:")
df.info()
```

Output

```
============================================================
STEP 1: DATA INSPECTION
============================================================

 FIRST LOOK AT DATA:
Shape: (10, 7)  (Rows: 10, Columns: 7)

First 5 rows:
  customer_id  age  income_k  loan_k  years_with_bank  missed_payments  \
0         C01   23       4.5     1.0                1                0
1         C02   45      12.0     6.0               12                1
2         C03   31       6.0     4.0                5                2
3         C04   26       5.0     2.5                2                0
4         C05   39       9.0     7.5                7                3

   default
0        0
1        0
2        1
3        0
4        1

Data Types:
customer_id         object
age                  int64
income_k           float64
loan_k             float64
years_with_bank      int64
missed_payments      int64
default              int64
dtype: object

Basic Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 7 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   customer_id      10 non-null     object
 1   age              10 non-null     int64
 2   income_k         10 non-null     float64
 3   loan_k           10 non-null     float64
 4   years_with_bank  10 non-null     int64
 5   missed_payments  10 non-null     int64
 6   default          10 non-null     int64
dtypes: float64(2), int64(4), object(1)
memory usage: 692.0+ bytes
```

Step 2: Missing values & Basic Stats

- why: Missing data breaks ML models
- what: Check for nulls, get summary statistics
- How: isnull(), describe()
- ML Connection: Need to handle missing values before training

```
print("\n" + "="*60)
print("STEP 2: MISSING VALUES & BASIC STATISTICS")
print("="*60)

# Check for missing values
print("\n MISSING VALUES CHECK:")
missing = df.isnull().sum()
if missing.sum() == 0:
    print(" No missing values found!")
else:
    print("Missing values found:")
    print(missing[missing > 0])

# Basic statistics
print("\n SUMMARY STATISTICS:")
print(df.describe().round(2))
```

Output

```
============================================================
STEP 2: MISSING VALUES & BASIC STATISTICS
============================================================

 MISSING VALUES CHECK:
 No missing values found!

 SUMMARY STATISTICS:
        age  income_k  loan_k  years_with_bank  missed_payments  default
count 10.00     10.00   10.00            10.00            10.00    10.00
mean  34.80      8.05    4.55             6.80             1.30     0.40
std    9.24      3.41    2.42             5.22             1.42     0.52
min   23.00      4.50    1.00             1.00             0.00     0.00
25%   28.25      5.62    2.62             3.25             0.00     0.00
50%   32.50      6.75    4.50             5.50             1.00     0.00
75%   40.50      9.75    6.38             9.25             2.00     1.00
max   52.00     15.00    8.00            18.00             4.00     1.00
```

Step 3: Univariate Analysis

- Why: Understand each feature individually
- What: Distributions, outliers, patterns in single variables
- How: Histograms, box plots, statistics

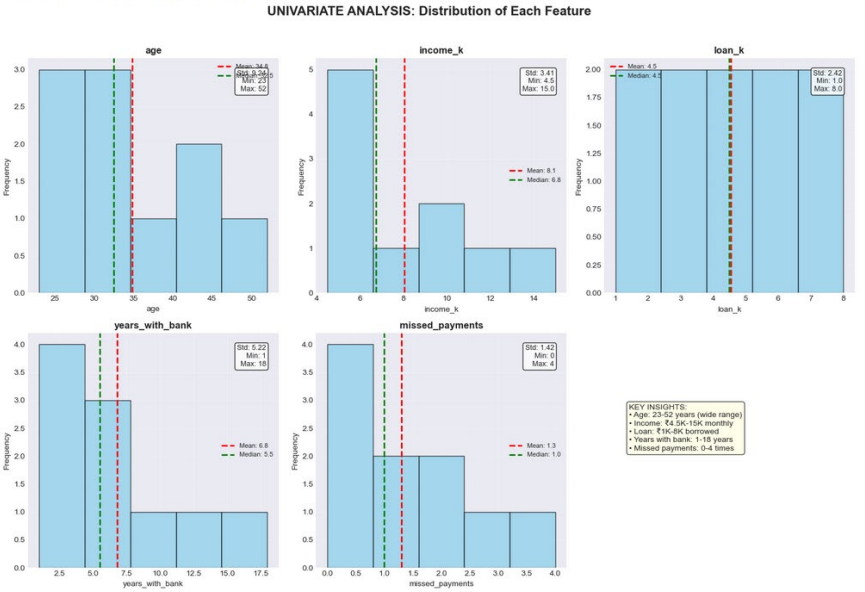
- ML Connection: Determines normalization needs, outlier handling

```
print("\n" + "="*60)
print("STEP 3: UNIVARIATE ANALYSIS (One Variable at a Time)")
print("="*60)


# Select numeric columns for analysis
numeric_cols = ['age', 'income_k', 'loan_k', 'years_with_bank',
'missed_payments']


# Create visualization
fig, axes = plt.subplots(2, 3, figsize=(15, 10))
fig.suptitle('UNIVARIATE ANALYSIS: Distribution of Each Feature',
             fontsize=16, fontweight='bold', y=1.02)


# Plot histograms and box plots for each numeric column
for idx, col in enumerate(numeric_cols):
    row, col_pos = divmod(idx, 3)


    # Histogram
    axes[row, col_pos].hist(df[col], bins=5, alpha=0.7,
color='skyblue', edgecolor='black')
    axes[row, col_pos].axvline(df[col].mean(), color='red',
linestyle='--',
                               linewidth=2, label=f'Mean:
{df[col].mean():.1f}')
    axes[row, col_pos].axvline(df[col].median(), color='green',
linestyle='--',
                               linewidth=2, label=f'Median:
{df[col].median():.1f}')
    axes[row, col_pos].set_title(f'{col}', fontweight='bold')
    axes[row, col_pos].set_xlabel(col)
    axes[row, col_pos].set_ylabel('Frequency')
    axes[row, col_pos].legend(fontsize=8)
    axes[row, col_pos].grid(True, alpha=0.3)


    # Add statistics as text
```

```
    stats_text = f"Std: {df[col].std():.2f}\nMin:
{df[col].min()}\nMax: {df[col].max()}"

    axes[row, col_pos].text(0.95, 0.95, stats_text,
transform=axes[row, col_pos].transAxes,

                            fontsize=9, verticalalignment='top',
horizontalalignment='right',

                            bbox=dict(boxstyle='round',
facecolor='white', alpha=0.8))


# Remove empty subplot

axes[1, 2].axis('off')


# Add key insights box

insights_text = """KEY INSIGHTS:

• Age: 23-52 years (wide range)

• Income: ₹4.5K-15K monthly

• Loan: ₹1K-8K borrowed

• Years with bank: 1-18 years

• Missed payments: 0-4 times"""

axes[1, 2].text(0.1, 0.5, insights_text, fontsize=10,

                bbox=dict(boxstyle='round', facecolor='lightyellow',
alpha=0.7))


plt.tight_layout()

plt.show()
```

Output

```
============================================================
STEP 3: UNIVARIATE ANALYSIS (One Variable at a Time)
============================================================
```

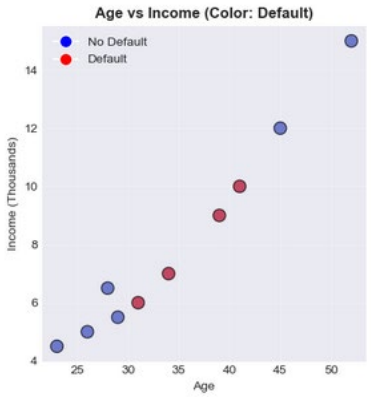


Step 4: Bivariate Analysis

- Why: Understand relationships between features
- What: How variables relate to each other and to target
- How: Scatter plots, correlation, grouped analysis
- ML Connection: Feature selection - which features matter most?

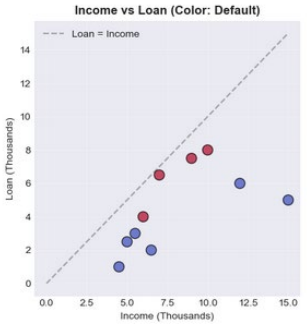
```
print("\n" + "="*60)
print("STEP 4: BIVARIATE ANALYSIS (Relationships Between
Variables)")
print("="*60)


# Create visualization for bivariate analysis
fig, axes = plt.subplots(2, 3, figsize=(15, 10))
fig.suptitle('BIVARIATE ANALYSIS: Relationships Between Features',
             fontsize=16, fontweight='bold', y=1.02)
# 1. Age vs Income with default status
```

```
scatter1 = axes[0, 0].scatter(df['age'], df['income_k'],
                              c=df['default'], s=100, alpha=0.7,
                              cmap='coolwarm', edgecolor='black')
axes[0, 0].set_xlabel('Age')
axes[0, 0].set_ylabel('Income (Thousands)')
axes[0, 0].set_title('Age vs Income (Color: Default)',
fontweight='bold')
axes[0, 0].grid(True, alpha=0.3)
```

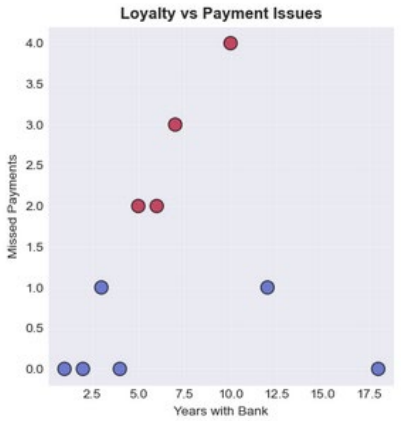



```
# Add legend for default
legend_elements = [plt.Line2D([0], [0], marker='o', color='w',
                              markerfacecolor='blue', markersize=10,
label='No Default'),
                   plt.Line2D([0], [0], marker='o', color='w',
                              markerfacecolor='red', markersize=10,
label='Default')]
axes[0, 0].legend(handles=legend_elements)


# 2. Income vs Loan with default status
scatter2 = axes[0, 1].scatter(df['income_k'], df['loan_k'],
                              c=df['default'], s=100, alpha=0.7,
                              cmap='coolwarm', edgecolor='black')
axes[0, 1].set_xlabel('Income (Thousands)')
axes[0, 1].set_ylabel('Loan (Thousands)')
axes[0, 1].set_title('Income vs Loan (Color: Default)',
fontweight='bold')
```

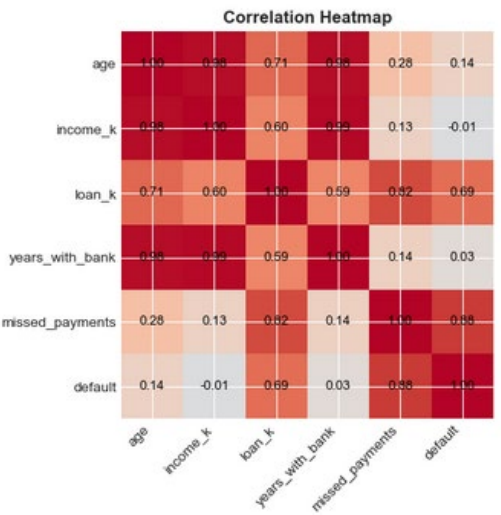
```
axes[0, 1].grid(True, alpha=0.3)
```




```
# Add 45-degree line (loan = income)
max_val = max(df['income_k'].max(), df['loan_k'].max())
axes[0, 1].plot([0, max_val], [0, max_val], 'k--', alpha=0.3,
label='Loan = Income')
axes[0, 1].legend()


# 3. Years with bank vs Missed payments
scatter3 = axes[0, 2].scatter(df['years_with_bank'],
df['missed_payments'],
                              c=df['default'], s=100, alpha=0.7,
                              cmap='coolwarm', edgecolor='black')
axes[0, 2].set_xlabel('Years with Bank')
axes[0, 2].set_ylabel('Missed Payments')
axes[0, 2].set_title('Loyalty vs Payment Issues', fontweight='bold')
axes[0, 2].grid(True, alpha=0.3)
```



```
# 4. Correlation Heatmap
axes[1, 0].axis('off')  # We'll use this for text
corr_matrix = df[numeric_cols + ['default']].corr()


# Create heatmap in separate subplot
im = axes[1, 1].imshow(corr_matrix, cmap='coolwarm', aspect='auto',
vmin=-1, vmax=1)
axes[1, 1].set_title('Correlation Heatmap', fontweight='bold')
axes[1, 1].set_xticks(range(len(corr_matrix.columns)))
axes[1, 1].set_yticks(range(len(corr_matrix.columns)))
axes[1, 1].set_xticklabels(corr_matrix.columns, rotation=45,
ha='right')
axes[1, 1].set_yticklabels(corr_matrix.columns)


# Add correlation values
for i in range(len(corr_matrix.columns)):
    for j in range(len(corr_matrix.columns)):
        text = axes[1, 1].text(j, i, f'{corr_matrix.iloc[i,
j]:.2f}',
                               ha="center", va="center",
color="black", fontsize=9)
```



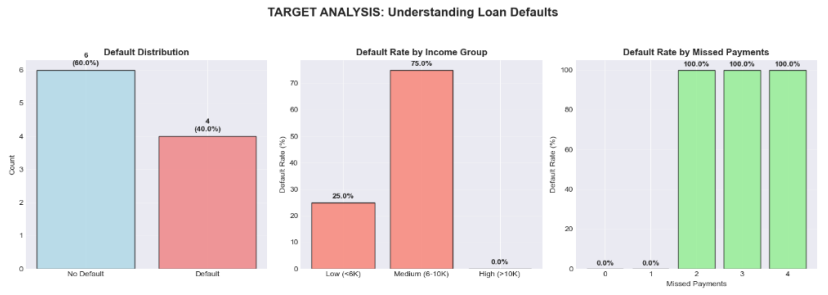
Step 5: Target Variable

- Why: Understanding the outcome we want to predict
- What: Distribution of default vs non-default
- How: Count plots, proportions, conditional statistics
- ML Connection: Class imbalance affects model performance

```
print("\n" + "="*60)
print("STEP 5: TARGET VARIABLE ANALYSIS (Default Prediction)")
print("="*60)

# Create visualization for target analysis
fig, axes = plt.subplots(1, 3, figsize=(15, 5))
fig.suptitle('TARGET ANALYSIS: Understanding Loan Defaults',
             fontsize=16, fontweight='bold', y=1.05)

# 1. Default distribution
default_counts = df['default'].value_counts()
colors = ['lightblue', 'lightcoral']
axes[0].bar(['No Default', 'Default'], default_counts.values,
           color=colors, edgecolor='black', alpha=0.8)
axes[0].set_title('Default Distribution', fontweight='bold')
axes[0].set_ylabel('Count')
axes[0].grid(True, alpha=0.3, axis='y')

# Add percentages on bars
for i, (label, value) in enumerate(default_counts.items()):
    percentage = (value / len(df)) * 100
    axes[0].text(i, value + 0.1, f'{value}\n({percentage:.1f}%)',
                 ha='center', va='bottom', fontweight='bold')

# 2. Default by income groups
df['income_group'] = pd.cut(df['income_k'], bins=[0, 6, 10, 20],
                            labels=['Low (<6K)', 'Medium (6-10K)',
'High (>10K)'])
```

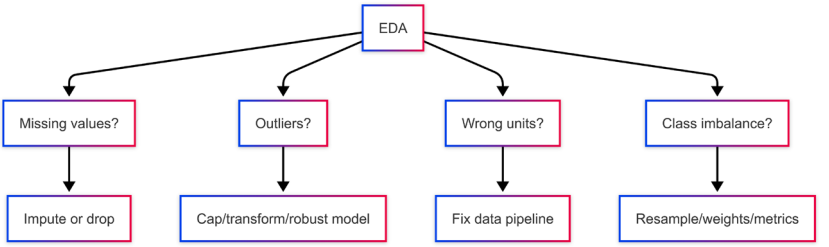
```
default_by_income = df.groupby('income_group')['default'].mean() *
100


axes[1].bar(default_by_income.index, default_by_income.values,
           color='salmon', edgecolor='black', alpha=0.8)
axes[1].set_title('Default Rate by Income Group', fontweight='bold')
axes[1].set_ylabel('Default Rate (%)')
axes[1].grid(True, alpha=0.3, axis='y')


# Add values on bars
for i, value in enumerate(default_by_income.values):
    axes[1].text(i, value + 1, f'{value:.1f}%', ha='center',
va='bottom', fontweight='bold')


# 3. Default by missed payments
default_by_missed = df.groupby('missed_payments')['default'].mean()
* 100


axes[2].bar([str(x) for x in default_by_missed.index],
default_by_missed.values,
           color='lightgreen', edgecolor='black', alpha=0.8)
axes[2].set_title('Default Rate by Missed Payments',
fontweight='bold')
axes[2].set_xlabel('Missed Payments')
axes[2].set_ylabel('Default Rate (%)')
axes[2].grid(True, alpha=0.3, axis='y')


# Add values on bars
for i, value in enumerate(default_by_missed.values):
    axes[2].text(i, value + 1, f'{value:.1f}%', ha='center',
va='bottom', fontweight='bold')


plt.tight_layout()
plt.show()


# Print target statistics
print("\n TARGET VARIABLE STATISTICS:")
```

```
print(f"Default Rate: {(df['default'].mean() * 100):.1f}%")
print(f"Non-default: {default_counts.get(0, 0)} customers")
print(f"Default: {default_counts.get(1, 0)} customers")
```



Target variable statistics:

- Default Rate: 40.0%
- Non-default: 6 customers
- Default: 4 customers



Quick Exercises

In Dataset A, which feature looks like it may create outliers?

Why do we check class imbalance before choosing accuracy as metric?

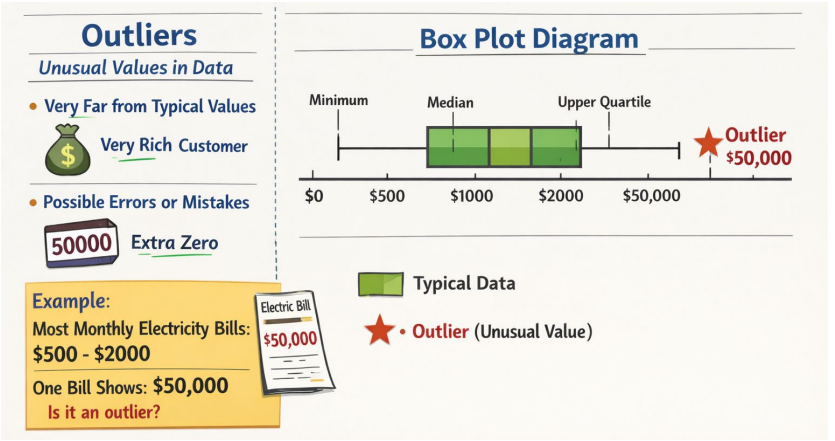
Write 3 EDA checks you will always do for tabular data.

Chapter 4
Box plots, outliers, and robust thinking

Definition

A box plot is a quick picture of how data is spread and where outliers may exist. Outliers are values that are very far from typical values. Sometimes they are real (very rich customer), sometimes they are errors (extra zero).

example: If most monthly electricity bills are 500-2000, but one bill shows 50,000, it may be an outlier and needs checking.



Syntax / Formula (Math in simple words)

Quartiles: Q1 (25%), Q2/median (50%), Q3 (75%).

IQR = Q3 - Q1.

Outlier fences (common rule): lower = Q1 - 1.5*IQR, upper = Q3 + 1.5*IQR.

Why?

Outliers can break the mean, scaling, and even model training. A box plot quickly shows if your data has extreme values.

What?

Five-number summary: min, Q1, median, Q3, max.
IQR = Q3 - Q1.
Outlier rule: below Q1 - 1.5*IQR or above Q3 + 1.5*IQR.

How?

Steps:
1) Sort data
2) Find median
3) Find Q1/Q3
4) Compute IQR and fences
In ML, you can cap outliers (winsorize), log-transform, or use robust models.

How this shows up in Machine Learning

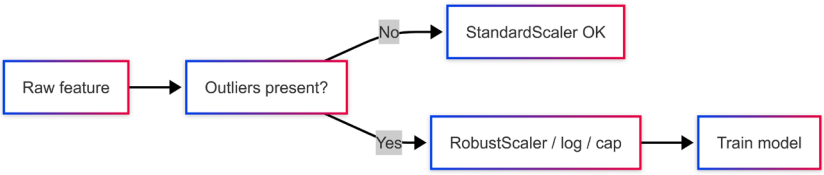
Why it matters: Standard Scaler uses mean and SD, both sensitive to outliers. For outlier-heavy data, use RobustScaler or log transform.

Let's Code!

```
# Demonstrate standard vs robust scaling
from sklearn.preprocessing import StandardScaler, RobustScaler
import numpy as np

loan = df[['loan_k']].values
std_scaled = StandardScaler().fit_transform(loan)
rob_scaled = RobustScaler().fit_transform(loan)

print("Original loan_k:", loan.ravel())
print("StandardScaler:", np.round(std_scaled.ravel(), 2))
print("RobustScaler:", np.round(rob_scaled.ravel(), 2))
```

Output

```
Original loan_k: [1.  6.  4.  2.5 7.5 5.  3.  6.5 2.  8. ]

StandardScaler: [-1.55  0.63 -0.24 -0.89  1.28  0.2  -0.68  0.85 -1.11
1.5 ]

RobustScaler: [-0.93  0.4  -0.13 -0.53  0.8   0.13 -0.4   0.53 -0.67  0.93]
```



Quick Exercises

Compute IQR fences for loan_k manually from Dataset A.

Why does median handle outliers better than mean?

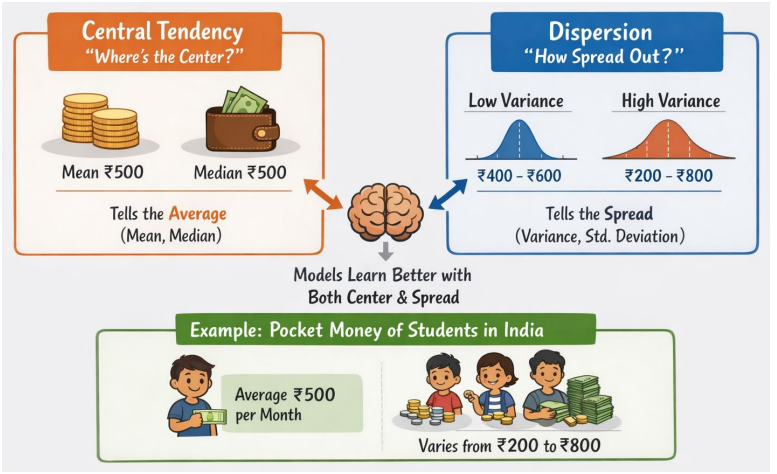
Give one banking feature that naturally has outliers (and why).

Chapter 5
Central tendency and dispersion (mean, median, variance, SD)

Definition

Central tendency tells the 'center' of data (mean, median). Dispersion tells how spread out data is (variance, standard deviation). Models learn better when you know both center and spread.

In India example: Average monthly pocket money and how much it varies across students tells you both typical and inequality.



Syntax / Formula (Math in simple words)

Mean: x_bar = (sum of x) / n.

Variance: var = (sum (x - x_bar)^2) / n (population version).

Standard deviation: sd = sqrt(var). Median is the middle value after sorting.

Why?

Models care about scale and spread. Dispersion tells if values are tight or scattered. This affects learning stability and feature scaling.

What?

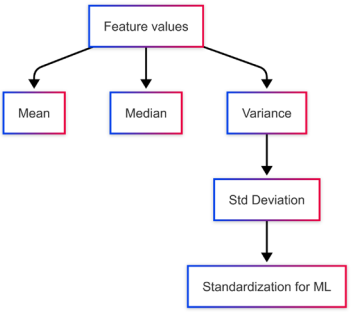
Mean = average, **Median** = middle, **Mode** = most frequent.
Variance = average squared distance from mean.
Standard deviation (SD) = sqrt(variance), in original units.

How?

Use mean for symmetric data. Use median for skewed data. Variance/SD get used everywhere: normalization, z-scores, Gaussian assumptions.

How this shows up in Machine Learning

Gradient descent works better when features are on similar scales. That's why we standardize: x' = (x - mean)/SD.



Let's Code!

```
import io

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt


# Data

csv_data =
"""customer_id,age,income_k,loan_k,years_with_bank,missed_payments,d
efault

C01,23,4.5,1.0,1,0,0

C02,45,12.0,6.0,12,1,0

C03,31,6.0,4.0,5,2,1

C04,26,5.0,2.5,2,0,0

C05,39,9.0,7.5,7,3,1
```

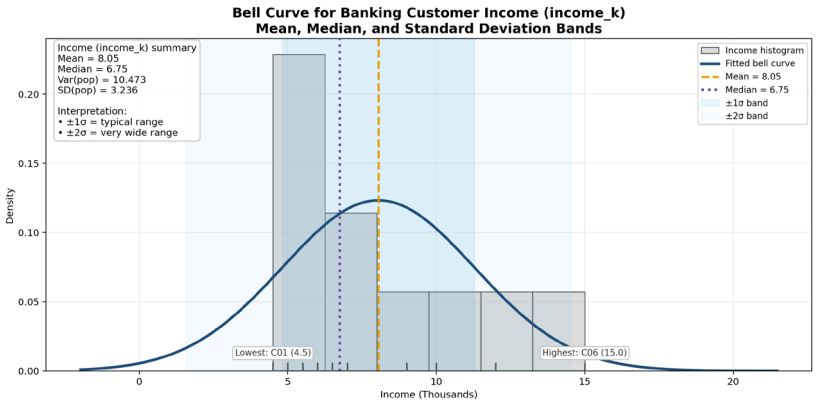
```
C06,52,15.0,5.0,18,0,0
C07,29,5.5,3.0,3,1,0
C08,34,7.0,6.5,6,2,1
C09,28,6.5,2.0,4,0,0
C10,41,10.0,8.0,10,4,1
"""
df = pd.read_csv(io.StringIO(csv_data))

# Stats (population variance & SD)
x = df["income_k"].to_numpy()
mean = np.mean(x)
median = np.median(x)
var_pop = np.mean((x - mean) ** 2)
sd_pop = np.sqrt(var_pop)

# Normal PDF
xmin = x.min() - 2 * sd_pop
xmax = x.max() + 2 * sd_pop
xs = np.linspace(xmin, xmax, 500)
pdf = (1 / (sd_pop * np.sqrt(2 * np.pi))) * np.exp(-0.5 * ((xs -
mean) / sd_pop) ** 2)

# Plot (improved, print-friendly colors)
plt.figure(figsize=(12, 6))

# Histogram (neutral)
plt.hist(
    x, bins=6, density=True,
    color="#D9D9D9", edgecolor="#4D4D4D", linewidth=1.0,
    alpha=0.9, label="Income histogram"
)




# Bell curve (navy)
```

```
plt.plot(xs, pdf, linewidth=2.8, color="#1F4E79", label="Fitted bell
curve")


# Mean & Median (distinct, readable)

plt.axvline(mean, linestyle="--", linewidth=2.2, color="#E69F00",
label=f"Mean = {mean:.2f}")

plt.axvline(median, linestyle=":", linewidth=2.6, color="#6A3D9A",
label=f"Median = {median:.2f}")


# SD bands (subtle)

plt.axvspan(mean - sd_pop, mean + sd_pop, color="#56B4E9",
alpha=0.14, label="±1σ band")

plt.axvspan(mean - 2*sd_pop, mean + 2*sd_pop, color="#56B4E9",
alpha=0.07, label="±2σ band")


# Rug: each customer income (dark gray)

y0 = 0.0

plt.scatter(x, np.full_like(x, y0), marker="|", s=320,
color="#4D4D4D", zorder=5)


# Annotate lowest & highest incomes with small callouts

low_idx = int(np.argmin(x))

high_idx = int(np.argmax(x))

plt.annotate(

    f"Lowest: {df.loc[low_idx,'customer_id']} ({x[low_idx]:.1f})",

    (x[low_idx], y0), textcoords="offset points", xytext=(0, 16),
ha="center",

    fontsize=9, color="#333333",

    bbox=dict(boxstyle="round,pad=0.25", facecolor="white",
edgecolor="#BFBFBF", alpha=0.95)

)

plt.annotate(

    f"Highest: {df.loc[high_idx,'customer_id']}
({x[high_idx]:.1f})",

    (x[high_idx], y0), textcoords="offset points", xytext=(0, 16),
ha="center",

    fontsize=9, color="#333333",

    bbox=dict(boxstyle="round,pad=0.25", facecolor="white",
edgecolor="#BFBFBF", alpha=0.95)
```

```
)


# Info box (clean white card)
info = (
    "Income (income_k) summary\n"
    f"Mean = {mean:.2f}\n"
    f"Median = {median:.2f}\n"
    f"Var(pop) = {var_pop:.3f}\n"
    f"SD(pop) = {sd_pop:.3f}\n\n"
    "Interpretation:\n"
    "• ±1σ = typical range\n"
    "• ±2σ = very wide range"
)
plt.gca().text(
    0.015, 0.985, info,
    transform=plt.gca().transAxes, va="top", ha="left", fontsize=10,
    bbox=dict(boxstyle="round,pad=0.35", facecolor="white",
edgecolor="#BFBFBF", alpha=0.95)
)


plt.title("Bell Curve for Banking Customer Income (income_k)\nMean,
Median, and Standard Deviation Bands",
          fontsize=14, fontweight="bold")
plt.xlabel("Income (Thousands)")
plt.ylabel("Density")
plt.grid(True, alpha=0.25)


# Trim legend items (avoid too much clutter)
handles, labels = plt.gca().get_legend_handles_labels()
# Keep first 5 entries in a sensible order
keep = ["Income histogram", "Fitted bell curve", f"Mean =
{mean:.2f}", f"Median = {median:.2f}", "±1σ band", "±2σ band"]
kept_handles, kept_labels = [], []
for h, l in zip(handles, labels):
    if l in keep and l not in kept_labels:
        kept_handles.append(h); kept_labels.append(l)
```

```
plt.legend(kept_handles, kept_labels, loc="upper right", fontsize=9,
framealpha=0.95)


plt.tight_layout()


out_path = "/mnt/data/bell_curve_income_k_v2.png"
plt.savefig(out_path, dpi=220)
plt.show()
out_path
```

Output



Key details from your data (income_k)

- Mean (average) = 8.05
- Median (middle value) = 6.75
- Variance (population) = 10.472
- SD (population) = 3.236

How to read the bell curve lines

- Mean line at 8.05 (center of curve)
- Median line at 6.75 (middle of sorted data)

±1 SD range:

- Mean − 1σ = 4.81
- Mean + 1σ = 11.29
- Most "typical" values usually fall here (if data is close to normal)

±2 SD range:

- Mean − 2σ = 1.58
- Mean + 2σ = 14.52
- Very wide range; values outside this are very unusual

Quick "highlight" insight

- Highest income is C06 = 15.0, its Z-score is 2.15 → unusually high (but still can be a real VIP customer)

Quick Exercises

Create one small dataset where mean != median. Explain why.

Why do we square distances in variance?

(hint: avoid negatives, punish far points)

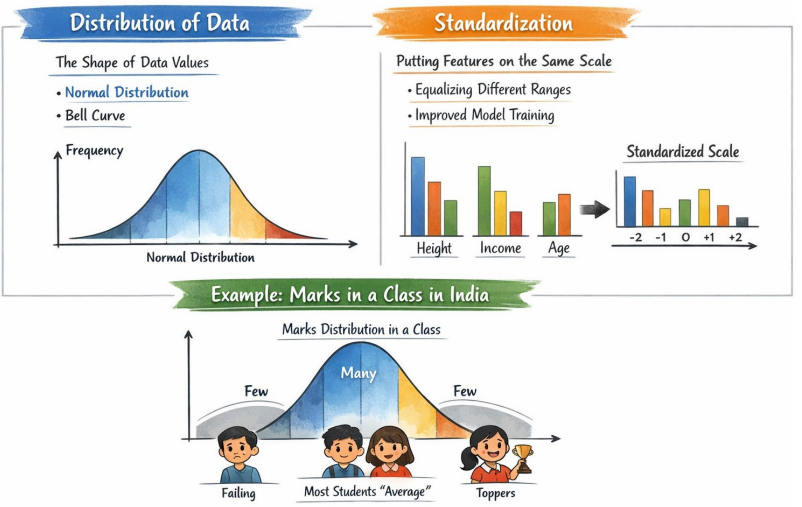
Which is more interpretable: variance or SD? Why?

Chapter 6
Distributions: Normal, z-score, and standardization

Definition

A distribution is the shape of data values (how often each value occurs). Normal distribution is a common bell-shaped pattern. Standardization puts different features on a comparable scale so models train smoothly.

In India example: Marks in a class often form a rough bell shape: many students in the middle, few at the extremes.



Syntax / Formula (Math in simple words)

Z-score: z = (x - mean) / sd.

After standardization, mean becomes ~0 and sd becomes ~1. This helps models like Logistic Regression and Neural Networks.

Why?

Many ML assumptions and tools work best when features are roughly bell-shaped. Even when not perfect, z-score helps compare values across different scales.

What?

Normal distribution is symmetric, defined by mean (center) and SD (spread).
z-score tells how many SDs a value is away from mean: z=(x-mean)/SD.

How?

Use z-score to:
- detect unusual values
- compare different features
- build simple anomaly rules
In ML pipelines, standardization is often done with StandardScaler.

How this shows up in Machine Learning

Neural nets and logistic regression usually train faster with standardized inputs. Tree-based models (RandomForest/XGBoost) often don't need scaling.

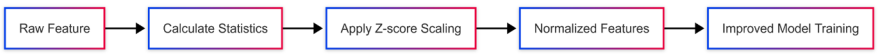
Let's Code!

```
import io
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler


# Z-score for income_k (manual)
income = df['income_k']
z = (income - income.mean()) / income.std(ddof=0)  # population SD
print(pd.DataFrame({'income_k': income, 'z_income': z.round(2)}))
# StandardScaler in a pipeline
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
```

```
pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('model', LogisticRegression())
])

pipe.fit(X_train, y_train)
pred = pipe.predict(X_test)
print(classification_report(y_test, pred))
```

Output

```
   income_k  z_income
0       4.5     -1.10
1      12.0      1.22
2       6.0     -0.63
3       5.0     -0.94
4       9.0      0.29
5      15.0      2.15
6       5.5     -0.79
7       7.0     -0.32
8       6.5     -0.48
9      10.0      0.60
              precision    recall  f1-score   support

           0       0.67      1.00      0.80         2
           1       0.00      0.00      0.00         1

    accuracy                           0.67         3
   macro avg       0.33      0.50      0.40         3
weighted avg       0.44      0.67      0.53         3
```



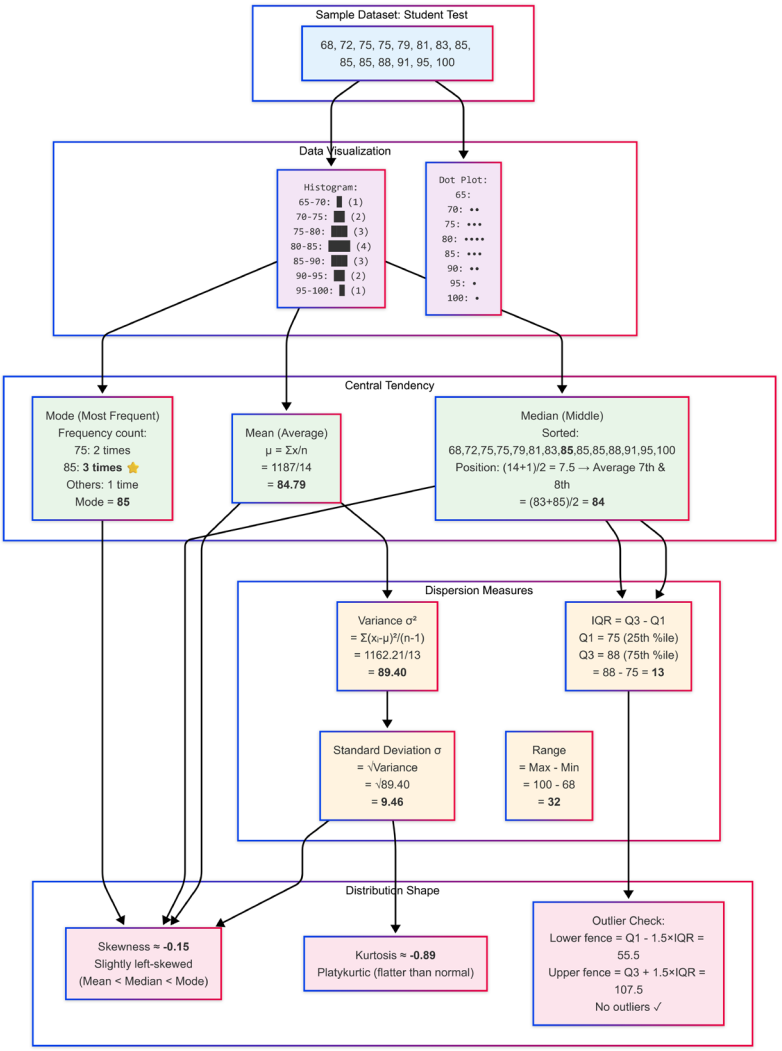
Quick Exercises

If z = +2, what does it mean in simple words?

Why might Standard Scaler be harmful if you fit it on full data before train/test split?

Name 2 models that need scaling and 2 models that don't.

Note: Mode is most meaningful for categorical data (like 'payment method'). For continuous amounts (INR), mode can be unstable unless you bin the data. In practice, bankers use median + IQR for robust summaries, and z-scores for anomaly flags.

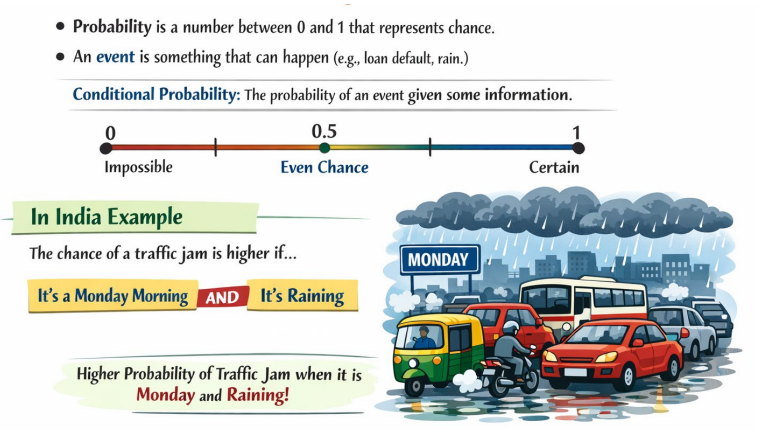

Sample Dataset: Student Test
68, 72, 75, 75, 79, 81, 83, 85, 85, 85, 88, 91, 95, 100
Data Visualization
Histogram:
65-70: ▌(1)
70-75: ▉(2)
75-80: ▉(3)
80-85: ▉(4)
85-90: ▉(3)
90-95: ▉(2)
95-100: ▌(1)
Dot Plot:
65: •
70: ••
75: •••
80: ••••
85: •••
90: ••
95: •
100: •
Central Tendency
Mode (Most Frequent)
Frequency count:
75: 2 times
85: 3 times ⭐
Others: 1 time
Mode = 85
Mean (Average)
μ = Σx/n
= 1187/14
= 84.79
Median (Middle)
Sorted:
68,72,75,75,79,81,83,85,85,85,88,91,95,100
Position: (14+1)/2 = 7.5 → Average 7th & 8th
= (83+85)/2 = 84
Dispersion Measures
Variance σ²
= Σ(xᵢ−μ)²/(n-1)
= 1162.21/13
= 89.40
IQR = Q3 - Q1
Q1 = 75 (25th %ile)
Q3 = 88 (75th %ile)
= 88 - 75 = 13
Standard Deviation σ
= √Variance
= √89.40
= 9.46
Range
= Max - Min
= 100 - 68
= 32
Distribution Shape
Skewness ≈ -0.15
Slightly left-skewed
(Mean < Median < Mode)
Kurtosis ≈ -0.89
Platykurtic (flatter than normal)
Outlier Check:
Lower fence = Q1 - 1.5×IQR = 55.5
Upper fence = Q3 + 1.5×IQR = 107.5
No outliers ✓

Chapter 7
Probability basics: events, conditional probability, independence

Definition

Probability is a number between 0 and 1 that represents chance. An event is something that can happen (loan default, rain). Conditional probability is probability given some information.

In India example: The chance of traffic jam is higher if it is a Monday morning and it is raining. That is conditional probability.



Syntax / Formula (Math in simple words)

Basic: P(A) between 0 and 1.

Conditional: $P(A|B) = P(A \text{ and } B) / P(B)$.

Independence: if A does not affect B, then $P(A|B) = P(A)$.

Why?

ML predictions are not always certain. Probability gives a language for uncertainty. For example, default probability 0.82 means high risk, but not 100% sure.

What?

Probability is between 0 and 1.
Conditional probability: P(A|B) means probability of A when B is already true.
Independent events: one does not change the other.

How?

```
Remember two key formulas:
1) P(A|B) = P(A and B) / P(B)
2) P(A and B) = P(A|B) * P(B)
```

How this shows up in Machine Learning

In classification models, the output is often a probability. Calibration tells whether 0.8 truly behaves like 80% in real life.

Let's Code!

```
# Simple conditional probability example using counts
# Suppose among 100 customers: 30 are high-loan, 20 are default, and
12 are both.
P_B = 30/100               # P(high-loan)
P_A_and_B = 12/100         # P(default AND high-loan)
P_A_given_B = P_A_and_B / P_B
print("P(default | high-loan) =", P_A_given_B)
```

Quick Exercises

Explain P(default | missed_payments >= 2) in plain words.

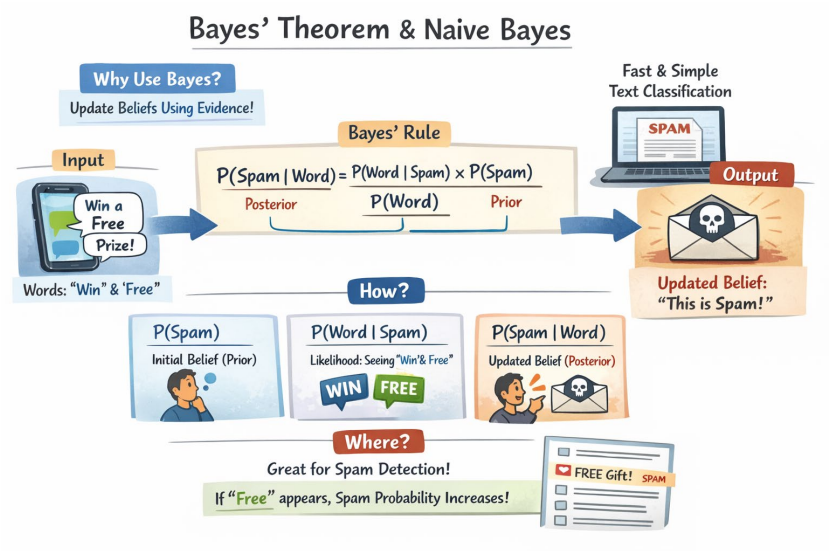
Give one example of independent events in real life, and one dependent.

Chapter 8
Bayes' theorem and Naive Bayes intuition

Definition

Bayes' theorem is the update rule: it tells how to update your belief after seeing new evidence. Naive Bayes uses Bayes with a simplifying assumption that features act independently given the class, which often works well for text.

In India example: You think a caller is spam. When you see words like 'win' and 'free', your belief increases. That is Bayes-style updating.



Syntax / Formula (Math in simple words)

Bayes formula: P(A | B) = P(B | A) * P(A) / P(B).

In spam detection: A = spam, B = word appears. We update P(spam | word).

Why?

Bayes is the 'update rule'. In ML, we update beliefs using evidence. Naive Bayes uses Bayes to classify text, spam, and documents very fast.

What?

Bayes theorem:
P(A|B) = P(B|A) * P(A) / P(B)
It converts 'cause to effect' into 'effect to cause'.

How?

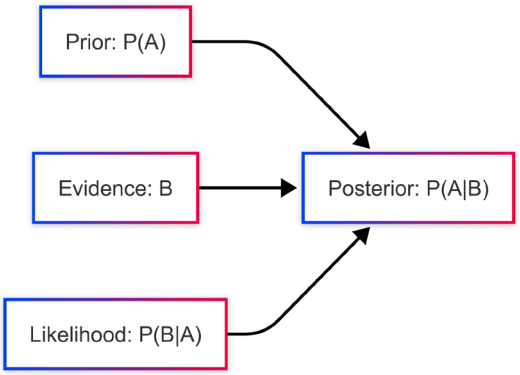
Think like this:
- P(A) is your initial belief (prior)
- P(B|A) is evidence likelihood
- P(A|B) is updated belief (posterior)
Naive Bayes assumes features are independent given the class (not always true, but works well).

How this shows up in Machine Learning

Text classification: if the word 'free' appears, spam probability increases. Naive Bayes combines many such word signals.

Let's Code!

```
# Tiny Bayes example
# Prior: P(default)=0.20
# Likelihood: P(missed>=2 | default)=0.60
# Evidence: P(missed>=2)=0.25
P_default = 0.20
P_e_given_default = 0.60
P_e = 0.25
P_default_given_e = (P_e_given_default * P_default) / P_e
print("P(default | missed>=2) =", P_default_given_e)
```



Quick Exercises

Explain ‘prior’ and ‘posterior’ using a loan default example.

Why is Naive Bayes fast and popular for text?

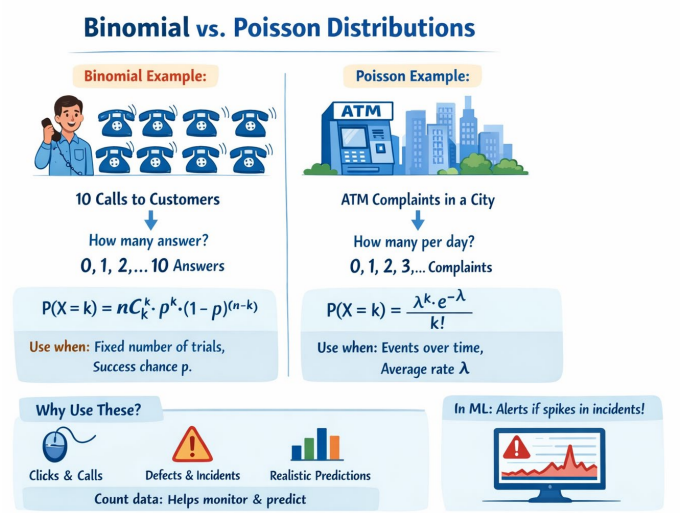
What is one weakness of the independence assumption?

Chapter 9
Binomial and Poisson: counting-world distributions

Definition

Binomial models 'how many successes' in a fixed number of trials (n) with success chance p. Poisson models 'how many events' happen in a time/space window when events occur randomly with an average rate lambda.

In India example: Binomial - out of 10 calls, how many customers answer? Poisson - how many ATM complaints come per day in a city?



Syntax / Formula (Math in simple words)

Binomial: P(X=k) = nCk * p^k * (1-p)^(n-k).

Poisson: P(X=k) = (lambda^k * e^(-lambda)) / k!.

Why?

Many ML tasks involve counting: number of clicks, number of calls, number of defects, number of incidents. These distributions help create good features and realistic simulations.

What?

Binomial: number of successes in n independent trials with probability p.
Poisson: number of events in a fixed time/space interval with average rate lambda.

How?

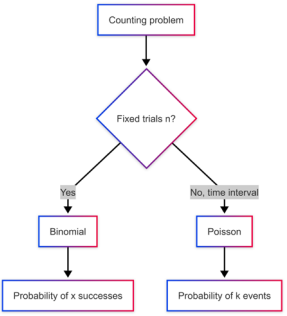
Use Binomial when:
- fixed n, two outcomes, constant p.
Use Poisson when:
- events happen over time/space, average rate is stable.

How this shows up in Machine Learning

In production monitoring, Poisson can model number of incidents per day. If incidents suddenly jump beyond expectation, it signals drift or outage.

Let's Code!

```
import math
def nCr(n, r):
    return math.factorial(n) // (math.factorial(r) *
math.factorial(n-r))

# Binomial P(X=2) for n=5, p=0.5
n, p, x = 5, 0.5, 2
P = nCr(n,x) * (p**x) * ((1-p)**(n-x))
print("Binomial P(X=2) =", P)

# Poisson P(X=3) for lambda=2
lam, k = 2, 3
Pp = (lam**k) * math.exp(-lam) / math.factorial(k)
print("Poisson P(X=3) =", Pp)
```



Quick Exercises

Give one banking example for Binomial and one for Poisson.

Why does Poisson mean = variance = lambda? (in simple words: rate controls both center and spread)

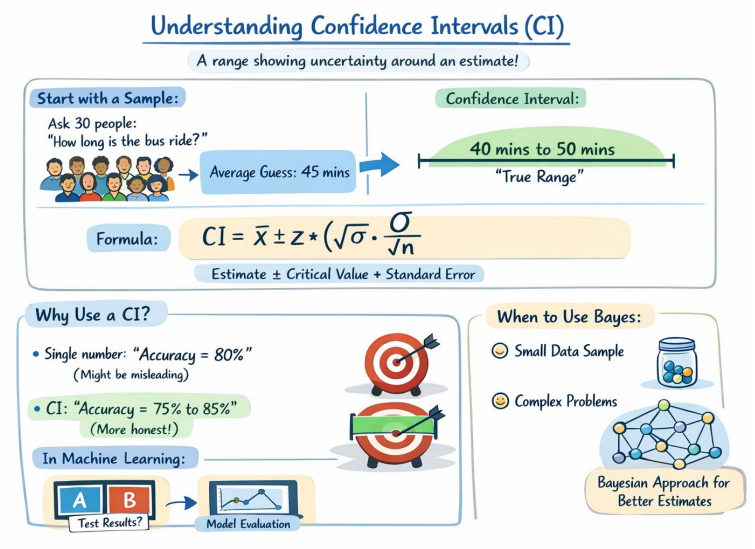
If lambda doubles, what happens to expected event count?

Chapter 10
Estimation and confidence intervals (CI)

Definition

An estimate is a best guess from a sample (like average rent from 50 flats). A confidence interval (CI) gives a range around that guess to show uncertainty. In AI, ranges are often more honest than single numbers.

In India example: If you ask 30 people the bus travel time, you will get a range. A CI is that range with a method behind it.



Syntax / Formula (Math in simple words)

General form: estimate +/- (critical value) * (standard error).

For mean (sigma known): CI = x_bar +/- z * (sigma / sqrt(n)).

Why?

In ML, a single number can lie. For example, model accuracy on 1 test set is just one estimate. CI gives a range, which is more honest.

What?

Point estimate = single best guess.
Confidence interval = a range that likely contains the true value.
General form: estimate ± critical_value * standard_error.

How?

If you repeat sampling many times, a 95% CI means about 95 out of 100 intervals will contain the true parameter.
CI is not '95% chance the parameter is inside this one interval' (common confusion).

How this shows up in Machine Learning

For A/B tests, CI helps decide if improvement is real or noise. For model evaluation, you can bootstrap accuracy to compute a CI.

Let's Code!

```
# Simple CI for a mean when sigma is known (z-based)
# Example: mean=41100, sigma=4500, n=36, 95% CI -> z=1.96
xbar, sigma, n, z = 41100, 4500, 36, 1.96
se = sigma / (n**0.5)
lower = xbar - z*se
upper = xbar + z*se
print("95% CI:", (round(lower,2), round(upper,2)))

# Bootstrap CI for model accuracy (concept)
# 1) sample test set with replacement many times
# 2) compute accuracy each time
# 3) take 2.5 and 97.5 percentiles as 95% CI
```

Quick Exercises

Why is CI better than only mean?

Write in one line: what is standard error?

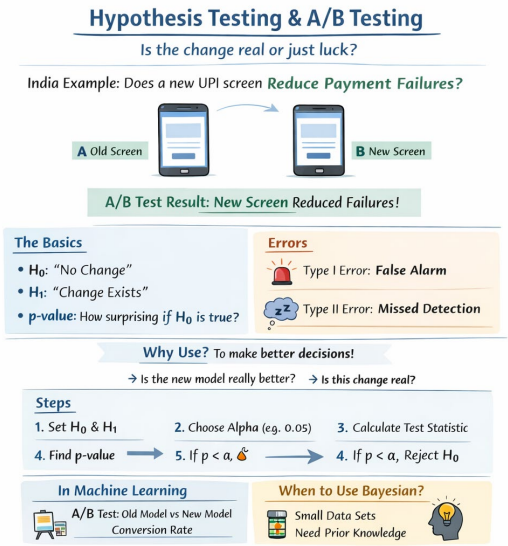
In ML, name one metric where CI is very useful (accuracy, AUC, etc.).

Chapter 11
Hypothesis testing and A/B tests (p-value, errors)

Definition

Hypothesis testing is a structured way to decide if an observed change is real or just noise. A/B testing compares two options (A vs B) and uses statistics to make a decision with controlled error.

In India example: Does a new UPI screen reduce payment failures, or was this week just lucky? A/B test answers that.



Syntax / Formula (Math in simple words)

H0: no change (default). H1: change exists.

p-value: how surprising the result is if H0 is true.

Errors: Type I (false alarm) and Type II (missed detection).

Why?

Teams often ask: 'Is our new model better?' or 'Is this change real?' Hypothesis testing gives a structured yes/no decision with controlled error.

What?

H0 = default assumption (no change).
H1 = what you want to show (change exists).
Type I error (alpha) = false alarm.
Type II error (beta) = missed detection.
p-value = how surprising your result is if H0 is true.

How?

Steps:
1) define H0, H1
2) choose alpha (like 0.05)
3) compute test statistic
4) compute p-value
5) decide: if p < alpha -> reject H0

How this shows up in Machine Learning

A/B test example: old model vs new model conversion rate. Hypothesis testing helps decide rollout without guessing.

Let's Code!

```
# Decision rule (memory-friendly)
# If p-value < alpha: Reject H0 (result is statistically
significant)
# Else: Fail to reject H0 (not enough evidence)

# Note: In practice use scipy.stats for z/t tests.
# Here we focus on concept, because wrong interpretation is the #1
beginner mistake.
```

Quick Exercises

Explain Type I error using loan approval example.

Why is ‘fail to reject’ not same as ‘accept’?

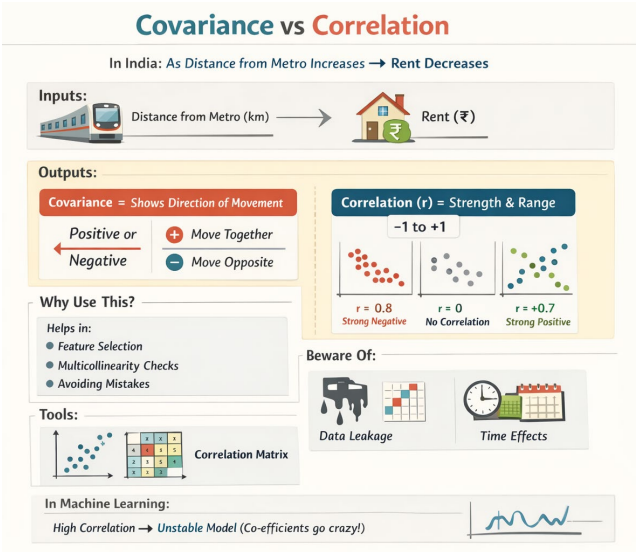
In A/B testing, what could happen if alpha is too high?

Chapter 12
Correlation, covariance, and the road to linear regression

Definition

Covariance tells whether two variables move together. Correlation is a scaled version between -1 and 1, showing direction and strength of linear relationship.

In India example: As distance from metro increases, rent often decreases. Correlation captures that relationship.



Syntax / Formula (Math in simple words)

```
Cov(x,y) = average((x - mean_x)*(y - mean_y)).
Corr(x,y) = Cov(x,y) / (sd_x * sd_y). Range is -1 to +1.
```

Why?

Correlation helps you understand if two variables move together. In ML, it helps for feature selection, multicollinearity checks, and sanity checks.

What?

Covariance: direction of joint movement (positive/negative). Correlation r: standardized covariance between -1 and +1. Correlation is about linear relationship, not causation.

How?

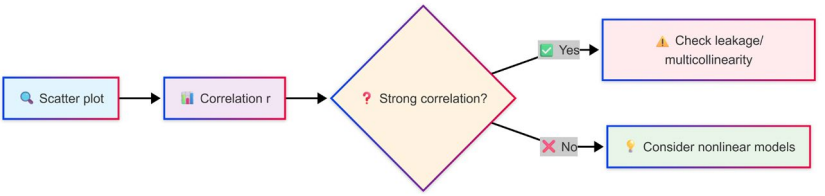
Use scatter plots + correlation matrix. Be careful: strong correlation may be due to leakage or time effects.

How this shows up in Machine Learning

Linear regression uses correlation-like ideas: it fits a line that minimizes squared errors. When two features are highly correlated, coefficients become unstable.

Let's Code!

```
# Correlation matrix for numeric features
corr =
df[['age','income_k','loan_k','years_with_bank','missed_payments','d
efault']].corr(numeric_only=True)
print(corr.round(2))

# Very important: correlation != causation
# Example: Ice cream sales and drowning both increase in summer.
```



Quick Exercises

Give one example where correlation is misleading.

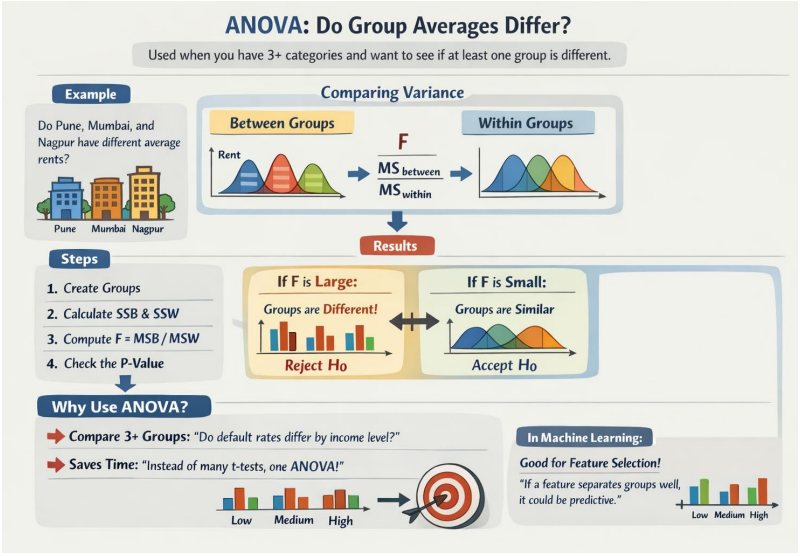
Why can high correlation between features hurt linear regression?

Chapter 13
ANOVA and when many groups are involved

Definition

ANOVA (Analysis of Variance) tests if the average of many groups is different. It is useful when you have 3+ categories and want to check if at least one group behaves differently.

example: Do three cities (Pune, Mumbai, Nagpur) have different average rent? ANOVA can test that.



Syntax / Formula (Math in simple words)

ANOVA compares: variation between groups vs variation within groups.

F statistic = MS_between / MS_within. Large F suggests group means differ.

Why?

Sometimes you want to compare more than 2 groups. For example: does default rate differ across 3 income groups? ANOVA helps test 'at least one group differs' without doing many t-tests.

What?

ANOVA compares variance between groups vs within groups using an F-test.
H0: all group means equal.
H1: at least one group mean differs.

How?

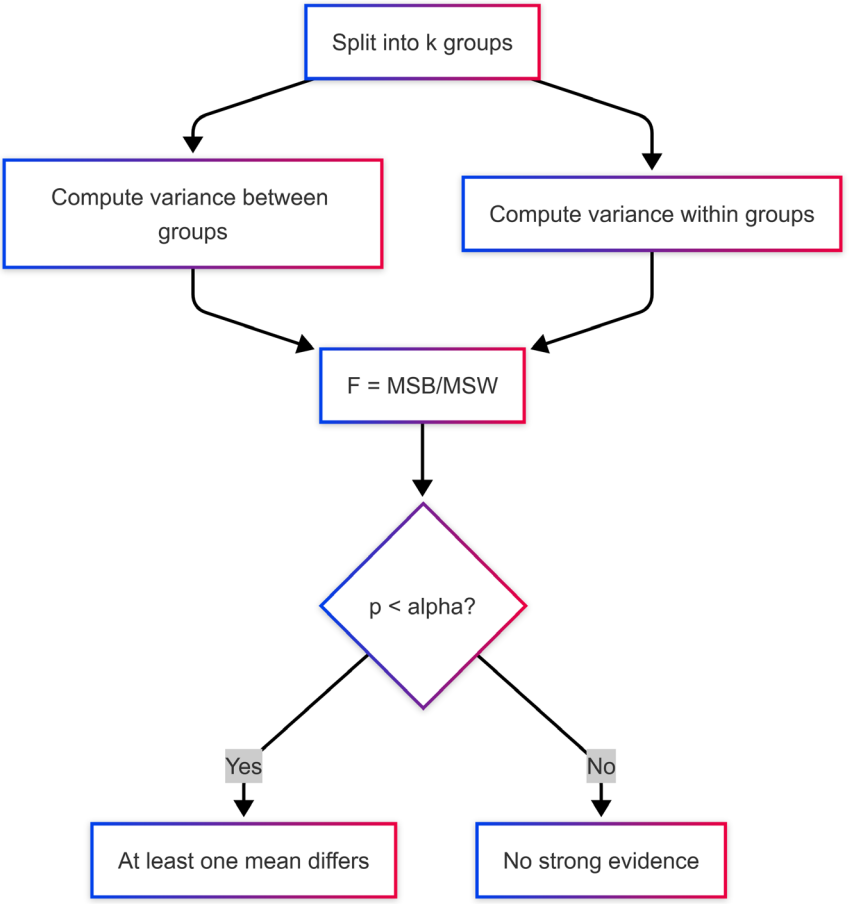
1) create groups
2) compute SSB (between) and SSW (within)
3) compute F = MSB/MSW
4) use p-value to decide

How this shows up in Machine Learning

ANOVA is also a feature selection idea: if a feature creates clearly different target means across groups, it can be predictive.

Let's Code!

```
# Practical note:
# In ML, you often use ANOVA F-test for feature selection:
# sklearn.feature_selection.f_classif or f_regression
from sklearn.feature_selection import f_classif
import numpy as np

X =
df[['age','income_k','loan_k','years_with_bank','missed_payments']].
to_numpy()
y = df['default'].to_numpy()
F, p = f_classif(X, y)
print("F:", np.round(F,3))
print("p:", np.round(p,4))
```



Quick Exercises

Why not do many t-tests instead of ANOVA?

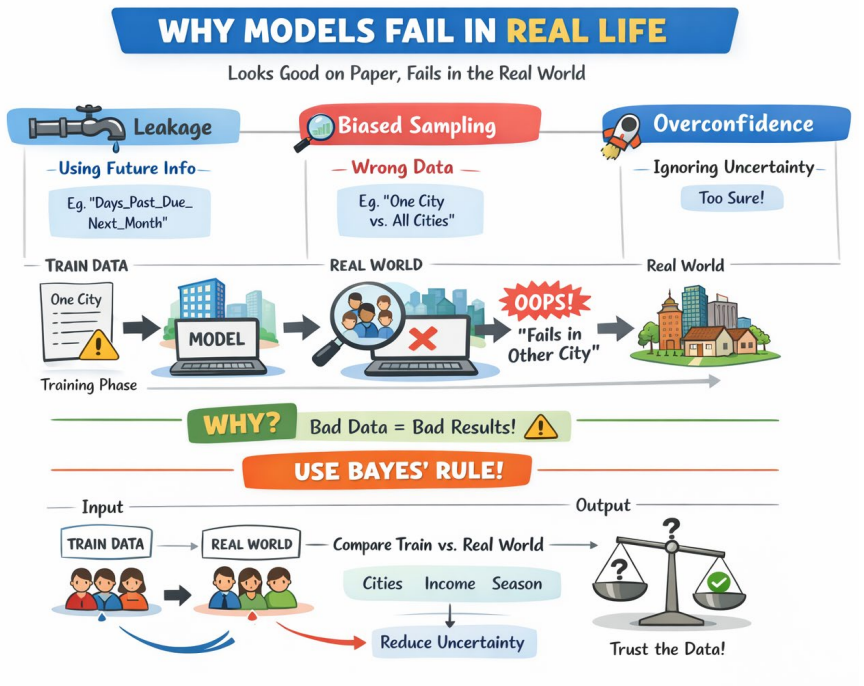
In one line: what does a large F mean?

Chapter 14
Common mistakes in ML statistics (leakage, bias, overconfidence)

Definition

These are mistakes that make models look good on paper but fail in real life. The biggest ones are leakage (future info enters features), biased sampling (training data not like real users), and overconfidence (ignoring uncertainty).

In India example: If you train on one city data only, the model may fail in another city where rent patterns are different.



Syntax / Formula (Math in simple words)

Leakage rule: features must be available at prediction time. If a column contains future information, remove it.

Bias check: compare train vs real-world distribution (city, income band, season).

Why?

Many ML failures happen not because of model choice, but because of statistical mistakes around data and evaluation.

What?

Top mistakes:
1) Data leakage
2) Non-representative sampling
3) Wrong metric (accuracy on imbalanced data)
4) Overfitting to test set
5) Misreading p-values and confidence intervals

How?

Rules that save projects:
- Split first, then fit scalers/encoders on train only
- Use stratified splits for imbalanced classification
- Keep a final untouched test set
- Monitor drift in production (data and labels)

How this shows up in Machine Learning

Example leakage: using 'days_past_due_next_month' to predict default. It looks powerful, but it is future information. Model will fail in real life.

Let's Code!

```
# Golden rule: fit transforms only on training data
# Wrong:
# scaler.fit_transform(full_data)

# Right:
# scaler.fit(X_train)
# X_train_scaled = scaler.transform(X_train)
# X_test_scaled  = scaler.transform(X_test)
```

Quick Exercises

Give one leakage example from banking or real estate.

Why accuracy can be misleading for fraud?

What is one sign of overfitting?

Chapter 15
Mini Projects + Interview Questions + Formula Sheet + Glossary

Mini Project 1: Loan Default Risk (end-to-end)

Goal: Build a simple baseline model, explain EDA findings, and evaluate honestly.

Steps:

Load Dataset A, do EDA (missing values, outliers, imbalance).

Create train/test split (stratify).

Train Logistic Regression with scaling. Also train a tree model without scaling.

Compare metrics: precision, recall, F1, ROC-AUC.

Write 5-line business recommendation: how to use risk score in approvals.

Mini Project 2: Rent Prediction (regression)

Goal: Predict rent and explain uncertainty.

Steps:

Load Dataset B, check skewness (area, rent).

Try Linear Regression and RandomForestRegressor.

Evaluate with MAE and RMSE.

Explain why MAE is more 'human friendly'.

Optional: bootstrap MAE to create a confidence interval.

Interview Questions (beginner-friendly)

Why do we standardize features? Give 2 models that benefit.

Explain p-value in simple words. What is a common misunderstanding?

What is the difference between variance and standard deviation?

When do you use z-test vs t-test?

Correlation vs causation: give an example.

What is data leakage? Why is it dangerous?

Formula Sheet (quick reference)

```
Mean:      mean = (sum of x) / n
Median:    middle value (sorted)
IQR:       IQR = Q3 - Q1
Fences:    lower = Q1 - 1.5*IQR, upper = Q3 + 1.5*IQR

Variance (population):  var = sum((x-mean)^2) / n
SD:                     sd = sqrt(var)

z-score:  z = (x - mean) / sd
CI (general): estimate ± critical_value * standard_error

Conditional probability: P(A|B) = P(A and B) / P(B)
Bayes: P(A|B) = P(B|A)*P(A) / P(B)

Binomial: P(X=x) = C(n,x) * p^x * (1-p)^(n-x)
Poisson:  P(X=k) = (lambda^k * e^-lambda) / k!
```

Glossary (simple)

Population: The full real-world group we care about (all customers).

Sample: A smaller group we actually collect and analyze.

Outlier: A value that is unusually far from most other values.

Bias: A systematic error (data or process favors certain outcomes).

Variance: How spread out data is; also used in model bias-variance tradeoff.

p-value: How surprising the result is if the null hypothesis is true.

Confidence Interval: A range of plausible values for the true parameter.

The Core Equations in Statistics

Statistics: "Understand the Data First"

1) Mean (Average)

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

- **Definition:** Sum of values / count.
- **Real life:** Average salary, average marks, average daily steps.
- **Why:** Gives a single number that represents the "center".

2) Variance (Spread)

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n} (x_i - \mu)^2$$

- **Definition:** Average squared distance from mean.
- **Real life:** Two classes can have same average marks, but one class may have very uneven marks.
- **Why:** Tells how "stable" or "risky" a dataset is.

3) Standard Deviation (Typical Spread)

$$\sigma = \sqrt{\sigma^2}$$

- **Definition:** Square root of variance.
- **Real life:** "Usually salaries vary by ±₹X around average."
- **Why:** Easier to understand because it comes back to original units.

4) Z-Score (How unusual is a value?)

$$z = \frac{x - \mu}{\sigma}$$

- **Definition:** How many standard deviations away from the mean.

- **Real life:** If your income is **2σ above average**, you are "rare" in that dataset.
- **Why:** Used for outlier detection and normalization.

5) Probability (Chance)

$$P(A) = \frac{\text{favorable outcomes}}{\text{total outcomes}}$$

- **Definition:** Chance of event A.
- **Real life:** Probability of rain, probability of default, probability of passing exam.
- **Why:** ML is often about predicting **chance**, not certainty.

6) Conditional Probability (Chance after seeing evidence)

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}$$

- **Definition:** Probability of A given B happened.
- **Real life:** Probability of "loan default" **given** "missed 2 EMIs".
- **Why:** This is the "brain" behind many ML decisions.

7) Bayes' Theorem (Update belief with evidence)

$$P(A \mid B) = \frac{P(B \mid A)\,P(A)}{P(B)}$$

- **Definition:** Updates your belief using new evidence.
- **Real life:** Doctor updates disease probability after a test result.
- **Why:** Foundation of Naive Bayes and belief-updating in AI.

Part II — Machine Learning

This part teaches the complete ML workflow with beginner-friendly math, tiny datasets, and production-ready deployment examples.

Preface

This book is written for complete beginners (freshers). You only need basic Python. We will avoid heavy math language and focus on intuition first.

Every topic follows the same pattern so you never feel lost:

Definition → Syntax/Formula → Why? → What? → How? → Let's Code!

All code uses tiny, static datasets (10 to 20 rows). This makes it easy to understand the full flow: data → training → prediction → evaluation → deployment.

Tools used in the book: Python, NumPy, Pandas, scikit-learn, Matplotlib, Flask.

Evolution of ML Algorithms

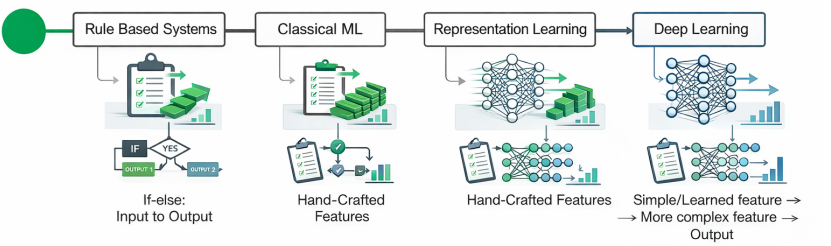


Table of Contents

1. Machine Learning in Simple Words
2. The ML Workflow (Data → Model → Prediction)
3. Data Basics: Features vs Target, Train/Test Split, Leakage
4. Evaluation Metrics: Accuracy, Precision, Recall, F1, R^2
5. Decision Trees (CART) and Gini Impurity
6. Random Forest (Bagging + Voting)
7. Linear Regression (Predict a Number)
8. Logistic Regression (Predict Yes/No)
9. K-Nearest Neighbors (Distance Based)
10. Support Vector Machines (Margin Based)
11. Naïve Bayes (Probability Based)
12. Clustering and K-Means (Unsupervised)
13. PCA (Dimensionality Reduction)
14. Practical Tips: Encoding, Scaling, Imbalanced Data
15. Production: Build an API + Deploy on GCP Cloud Run
16. Production: Deploy on AWS Lambda (Container) + API Gateway
17. Appendix A: Common Interview Questions
18. Appendix B: Mini Project Ideas

Chapter 1
Machine Learning Simple Words

Definition

Machine Learning (ML)means:
A computer learns from examples (data) and then makes predictions for new cases.

Instead of writing 1000 rules like:

- if this happens → do that
  we show the computer many examples, and it learns the pattern by itself.

Basic Idea (No heavy math)

Input (Features) → Model → Output (Prediction)

Example:

- Input: House size, location, number of rooms
- Output: House price

So, we can write it like**: Prediction = Model (Inputs)**

What is "Training"?

Training means:

- The model first makes a wrong guess (in the beginning)
- Then it learns from mistakes
- After many examples, it becomes better and better

In simple words:
Training = learning from errors and improving step-by-step

Why do we need ML?

Because rules fail in real life.

Example**:**

- Fraud pattern today is different from fraud pattern after 1 month.
- If we write fixed rules, they become outdated quickly.

ML is helpful because:
It learns from new data
It can adapt when patterns change

Real-Life Example (India)

Think of YouTube / Instagram Reels:

If you watch:

- cricket highlights
- Rohit Sharma videos
- IPL reels

After some time, your feed is filled with cricket.

Why? Because the system learns your interest pattern from your clicks and watch time.
That is Machine Learning.

What is Data in ML?

Most ML data looks like an Excel table.

- Each row = one example (one record)
- Each column = one detail (feature)

Example:

Example:

Age	Salary	City	Buy Insurance?
28	6 LPA	Pune	Yes
22	3 LPA	Nagpur	No

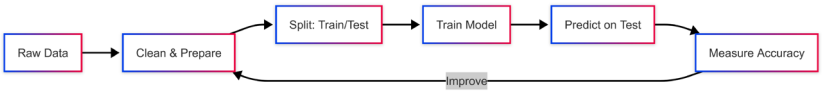
Here:

- Features (X) = Age, Salary, City
- Target (y) = Buy Insurance? (Yes/No)

How ML Works (Simple Steps)

1. Collect data (Excel / CSV / database)
2. Clean data (remove blanks, fix wrong values)
3. Understand data (EDA: "what is happening in data?")
4. Split data:
    - Train data = for learning
    - Test data = for checking on unseen data
5. Train the model
6. Test the model and check accuracy/performance

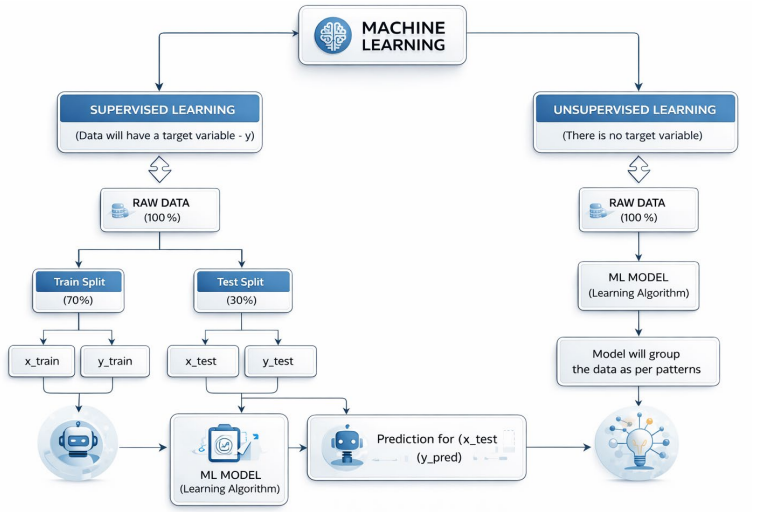
A simple ML flow diagram (Mermaid)



Key Takeaways

- ML learns patterns from examples, not rules.
- A dataset is features (X) + target (y) for supervised learning.
- Training = reduce loss (error) by adjusting parameters.
- Always start with tiny data and verify understanding.

Machine Learning workflow



Machine Learning Overview

This diagram gives a big-picture map of **Machine Learning** and how problems are grouped.

It splits ML into **Supervised Learning (labeled data)** and **Unsupervised Learning (no labels)**.

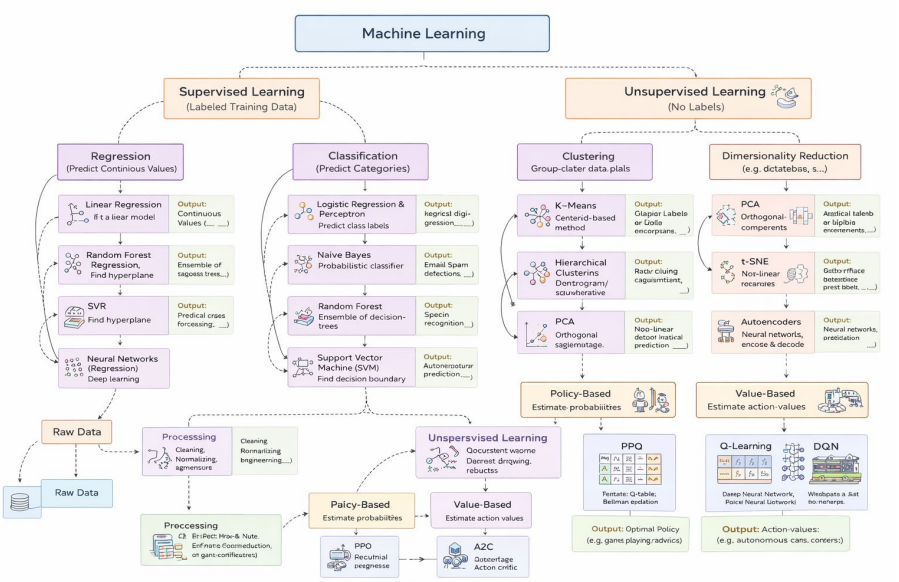
In supervised learning, you have **Regression** (predict a number like price) and **Classification** (predict a category like spam/not spam).

Common regression/classification methods shown include **Linear/Logistic Regression, Naive Bayes, Random Forest, SVM, and Neural Networks**.

In unsupervised learning, it highlights **Clustering** (group similar items, e.g., K-Means, Hierarchical) and **Dimensionality Reduction** (compress features, e.g., PCA, t-SNE, Autoencoders).

At the bottom, it reminds that real projects start from **raw data → processing/cleaning/feature work → model training**.

It also introduces **Reinforcement Learning**, split into **Policy-based** (learn actions directly, e.g., PPO) and **Value-based** (learn action-values, e.g., Q-Learning, DQN).


Machine Learning
Supervised Learning
(Labeled Training Data)
Unsupervised Learning
(No Labels)
Regression
(Predict Continious Values)
Classification
(Predict Categories)
Clustering
Group-clater data plals
Dimersionality Reduction
(e.g. dctatebas, s...)
Linear Regression
Output:
Random Forest Regression,
Output:
SVR
Output:
Neural Networks
(Regression)
Deep learning
Logistic Regression &
Perceptron
Predict class labels
Output:
Naive Bayes
Probabilistic classifier
Output:
Random Forest
Ensemble of decision-trees
Output:
Support Vector Machine (SVM)
Find decision boundary
Output:
K-Means
Centerid-based method
Output:
Hierarchical Clusterins
Dontrogram/
Output:
PCA
Orthogonal sagiemutage.
Output:
PCA
Orthogonal-
Output:
t-SNE
Output:
Autoencoders
Neural networks,
Output:
Policy-Based
Estimate probabiiitres
Value-Based
Estimate action values
Raw Data
Raw Data
Processsing
Cleaning,
Normalizing,
Preccessing
Unspersvised Learning
Paicy-Based
Estimate probabilitres
Value-Based
Estimate action values
PPO
Recutnial
A2C
Qoterfage
Acton critic
PPQ
Q-Learning
DQN
Output: Optimal Policy
Output: Action-values:

Chapter 2
The ML Workflow (Data → Model → Prediction)

Beginner Notes

Machine Learning is a cycle: decide problem → collect data → train → check results → improve → deploy → monitor.

Definition

An ML workflow is a repeatable set of steps to build a model safely so it works in real life, not only in a notebook.

Simple Syntax

X = inputs (features), y = output (target). fit(X_train, y_train) trains the model. predict(X_test) predicts on new data.

Why Workflow Matters

Without a workflow, we may accidentally cheat (data leakage) and get fake high accuracy. Workflow keeps train/test separate.

Real-life Example

Predict whether a customer will pay EMI on time: collect features, train on old customers, test on unseen customers, then use in production.

Train / Validation / Test (Easy Exam Example)

Train = studying, Validation = mock test, Test = final exam (use only at the end).

Workflow Steps

- Separate target (y) from features (X).
- Split into Train and Test.
- Train on Train set.
- Evaluate on Test set.

- Improve step-by-step (cleaning, features, tuning) without touching test.

```
import pandas as pd
from sklearn.model_selection import train_test_split


# Tiny dataset: predict "Passed" based on Hours Studied and
Attendance
data = [
    {"hours": 1, "attendance": 60, "passed": 0},
    {"hours": 2, "attendance": 65, "passed": 0},
    {"hours": 3, "attendance": 70, "passed": 0},
    {"hours": 4, "attendance": 75, "passed": 1},
    {"hours": 5, "attendance": 80, "passed": 1},
    {"hours": 6, "attendance": 85, "passed": 1},
    {"hours": 7, "attendance": 90, "passed": 1},
    {"hours": 2, "attendance": 80, "passed": 0},
    {"hours": 8, "attendance": 55, "passed": 0},
    {"hours": 9, "attendance": 92, "passed": 1},
]

df = pd.DataFrame(data)

X = df[["hours", "attendance"]]   # features
y = df["passed"]                  # target

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

print("Train rows:", len(X_train))
print("Test rows :", len(X_test))
print(X_train.head())
```

What to take care: Never use test data during training or tuning.

```
Train rows: 8
Test rows : 2
   hours  attendance
5      6          85
0      1          60
7      2          80
2      3          70
9      9          92
```

Key Takeaways

- Workflow: data -> split -> train -> evaluate -> deploy.
- Keep training and test data separate to avoid cheating.
- Use a baseline first (simple model) before complex models.
- Monitor performance after deployment (data can drift).

Chapter 3
Data Basics (Features, Target, Leakage)

Extra Theory (Beginner Notes)

Most ML failures happen due to data issues, not algorithms. A simple model on clean data often beats a complex model on messy data.

Data leakage: using future information to predict the past (gives fake high accuracy).

Train/Validation/Test: train learns, validation tunes, test is final exam.

Categorical handling: One-hot for non-ordered categories; Label encoding only for ordered categories.

Scaling: required for distance-based models (KNN, SVM, K-Means), not required for trees.

Definition

Feature (X): input columns used to predict.

Target (y): the output column we want to predict.

Syntax / Formula

```
X = [x1, x2, x3, ...]
y = target
```

Model learns: $y \approx f(X)$

Why?

A model is only as good as data.

Wrong data preparation leads to wrong predictions.

Real-life Example

In loan approvals, a common leakage mistake is using 'approved_amount' as a feature when it is decided after approval. That gives false accuracy in training but fails in real life.

What?

Data leakage: when the model sees future information (directly or indirectly).

Example: if you want to predict loan default, you must not include 'default_flag' or 'final_amount_paid' in features.

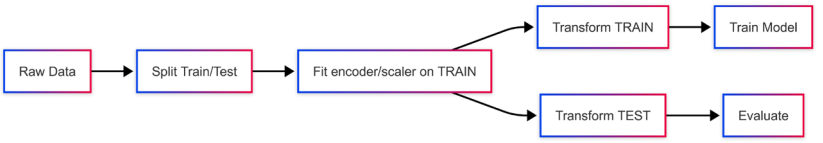
How?

Keep target separate.

Use training data only to compute scaling and encoders.

Apply the same transformation to test and production data.

Safe data preparation pipeline



Quick example: One-Hot Encoding in simple words

ML models understand numbers, not text. So we convert categories like 'Govt', 'Private' into 0/1 columns.

```
import pandas as pd


df = pd.DataFrame([
    {"occupation": "Govt"},
    {"occupation": "Private"},
    {"occupation": "Self"},
    {"occupation": "Govt"},
])

encoded = pd.get_dummies(df["occupation"], prefix="occ")
print(encoded)
```

In most projects, one-hot encoding is safer than label encoding for non-ordered categories.

Key Takeaways

- Features must be available at prediction time (no future info).
- Target leakage makes accuracy fake and dangerous.
- Always do a leakage scan before training.
- Good data quality beats fancy algorithms.

Chapter 4
Evaluation Metrics
(How to know - how model is good?)

Extra Theory (Beginner Notes)

Choosing the right metric is like choosing the right exam. If you measure the wrong thing, you will build the wrong model.

Accuracy is misleading for imbalanced data (e.g., 95% non-fraud).

Precision answers: "When the model says YES, how often is it correct?"

Recall answers: "Out of all real YES cases, how many did we catch?"

F1 balances precision and recall.

ROC-AUC shows how well the model separates classes across all thresholds.

Definition

A metric is a number that tells how well your model is performing.

We use different metrics for classification and regression.

Real-life Example (India)

In a disease screening test, accuracy alone is misleading if disease is rare. Similarly, in fraud detection in India, precision/recall matters more than accuracy.

Syntax / Formula

Classification: Accuracy, Precision, Recall, F1

Regression: MAE, MSE, RMSE, R^2

Math in Simple Words

Accuracy = correct predictions / total predictions.

Precision = out of predicted YES, how many were really YES.

Recall = out of real YES, how many we caught.

F1 = balance between precision and recall.

R^2 = how much variation your regression model explains (0 to 1).

Classification formula (small)

```
Accuracy = (TP + TN) / (TP + TN + FP + FN)
Precision = TP / (TP + FP)
Recall = TP / (TP + FN)
```

F1 = 2 * Precision * Recall / (Precision + Recall)

Let's Code! (Confusion Matrix + Metrics)

```
from sklearn.metrics import confusion_matrix, accuracy_score,
precision_score, recall_score, f1_score


# Actual vs Predicted (tiny example)
y_true = [1, 0, 1, 1, 0, 0, 1, 0]
y_pred = [1, 0, 0, 1, 0, 1, 1, 0]


cm = confusion_matrix(y_true, y_pred)
print("Confusion Matrix:\n", cm)


print("Accuracy :", accuracy_score(y_true, y_pred))
print("Precision:", precision_score(y_true, y_pred))
print("Recall   :", recall_score(y_true, y_pred))
print("F1 Score :", f1_score(y_true, y_pred))
```

Regression: MAE, MSE, R^2

MAE = average absolute error. MSE = average squared error. R^2 = explained variance.

```
from sklearn.metrics import mean_absolute_error, mean_squared_error,
r2_score


y_true = [2.0, 2.5, 3.0, 3.5]
y_pred = [2.2, 2.4, 2.9, 3.6]


print("MAE:", mean_absolute_error(y_true, y_pred))
print("MSE:", mean_squared_error(y_true, y_pred))
print("R2 :", r2_score(y_true, y_pred))
```

What to take care: For imbalanced data, accuracy can lie. Use precision/recall/F1.

Chapter 5
Decision Trees (CART) and Gini Impurity

Extra Theory (Beginner Notes)

A decision tree is like a flowchart of questions. At each node, the tree asks a question that best separates the classes.

Greedy learning: the tree chooses the best split NOW (it does not plan future splits).

Overfitting risk: a very deep tree can memorize training data. Pruning / limiting depth helps generalize.

Bias–variance intuition: deeper tree → low bias but high variance; shallow tree → higher bias but lower variance.

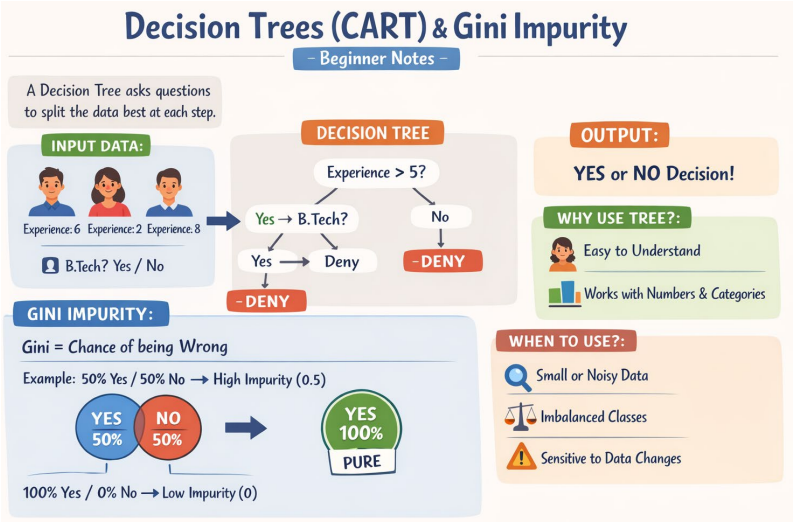
When trees struggle: small noisy datasets, very imbalanced classes, or when a tiny change in data changes the tree.

Feature importance in trees is based on how much each feature reduces impurity across splits.

Definition

A Decision Tree is a flowchart-like model that asks questions step-by-step.

Example: 'Is experience > 5?' → 'Is qualification B-Tech?' → final decision YES/NO.



Syntax / Formula

CART uses Gini Impurity by default for classification.

```
Gini(node) = 1 - Σ (p_class)^2
```

Best split = split that makes impurity smaller (or makes Gini Gain larger).

Why?

Trees are easy to explain to business people.

They work well with mixed data (numbers + categories).

Real-life Example (India)

A decision tree is like a simple flowchart your bank officer may follow: if income > X and missed_payments <= Y then approve. The tree learns these splits from data.

What?

Popular tree families:

- CART: binary splits, supports classification + regression.
- CHAID: multi-way splits, mostly categorical, uses chi-square tests.

- C4.5 / ID3: uses entropy / information gain.

How?

At each node, the tree tries many splits and chooses the best one.

For CART classification, 'best' means lowest Gini after split.

Math in Simple Words: What is Gini Impurity?

Gini impurity is the chance of being wrong if you randomly guess a class using the class proportions in that node.

Example: if a node has 50% Yes and 50% No, impurity is high.

If a node has 100% Yes and 0% No, impurity is 0 (pure).

Tiny hand-calculation example

Suppose a node has 10 records: 6 Yes and 4 No.

```
p_yes = 6/10, p_no = 4/10
Gini = 1 - (p_yes^2 + p_no^2) = 1 - (0.36 + 0.16) = 0.48
```

Let's Code!

We will build a CART model to predict HR Approval (Yes/No) using a tiny dataset.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report


# Tiny dataset (static)
data = [
    {"exp": 5, "state": "TN", "qualification": "B-Tech",
"hr_approved": 1},
    {"exp": 5, "state": "AP", "qualification": "B-Tech",
"hr_approved": 1},
```

```
    {"exp":10, "state": "UP", "qualification": "MBA",
"hr_approved": 0},
    {"exp": 5, "state": "UP", "qualification": "PhD",
"hr_approved": 0},
    {"exp": 7, "state": "TN", "qualification": "B-Tech",
"hr_approved": 1},
    {"exp": 2, "state": "MH", "qualification": "B-Tech",
"hr_approved": 0},
    {"exp": 8, "state": "KA", "qualification": "MBA",
"hr_approved": 0},
    {"exp": 6, "state": "TN", "qualification": "B-Tech",
"hr_approved": 1},
    {"exp": 4, "state": "AP", "qualification": "MBA",
"hr_approved": 0},
    {"exp": 9, "state": "TN", "qualification": "PhD",
"hr_approved": 0},
]


df = pd.DataFrame(data)
X = df[["exp", "state", "qualification"]]
y = df["hr_approved"]


X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.25, random_state=7)


cat_cols = ["state", "qualification"]
num_cols = ["exp"]


preprocess = ColumnTransformer(
    transformers=[
        ("cat", OneHotEncoder(handle_unknown="ignore"), cat_cols),
        ("num", "passthrough", num_cols),
    ]
)


model = DecisionTreeClassifier(
    criterion="gini",
    max_depth=3,
```

```
    min_samples_split=2,
    random_state=7
)


pipe = Pipeline(steps=[("prep", preprocess), ("model", model)])
pipe.fit(X_train, y_train)


pred = pipe.predict(X_test)
print(classification_report(y_test, pred))


# ===== ADDITION: Predict on unseen data =====
# Example: a new candidate with 6 years experience, from Maharashtra ('MH') with B-Tech
unseen = pd.DataFrame({"exp": [6], "state": ["MH"], "qualification": ["B-Tech"]})
unseen_pred = pipe.predict(unseen)[0]
unseen_proba = pipe.predict_proba(unseen)[0, 1]  # probability of class 1 (hr_approved)


print("\n--- Unseen data prediction ---")
print(f"Input: exp=6, state=MH, qualification=B-Tech")
print(f"Predicted outcome: {'Approved' if unseen_pred == 1 else 'Not Approved'}")
print(f"Probability of approval: {unseen_proba:.2f}")
```

What to take care: Trees overfit easily. Use max_depth, min_samples_split, and cross-validation.

Output

precision recall f1-score support

0 1.00 0.50 0.67 2

1 0.50 1.00 0.67 1

accuracy 0.67 3

macro avg 0.75 0.75 0.67 3

weighted avg 0.83 0.67 0.67

--- Unseen data prediction ---

Input: exp=6, state=MH, qualification=B-Tech

Predicted outcome: Approved

Probability of approval: 1.00

Key Takeaways

- Decision Trees split data using simple questions.
- They are explainable and work on mixed data types.
- They can overfit if too deep - use pruning.
- Great for quick prototypes and business explainability.

Chapter 6
Random Forest (Bagging + Voting)

Extra Theory (Beginner Notes)

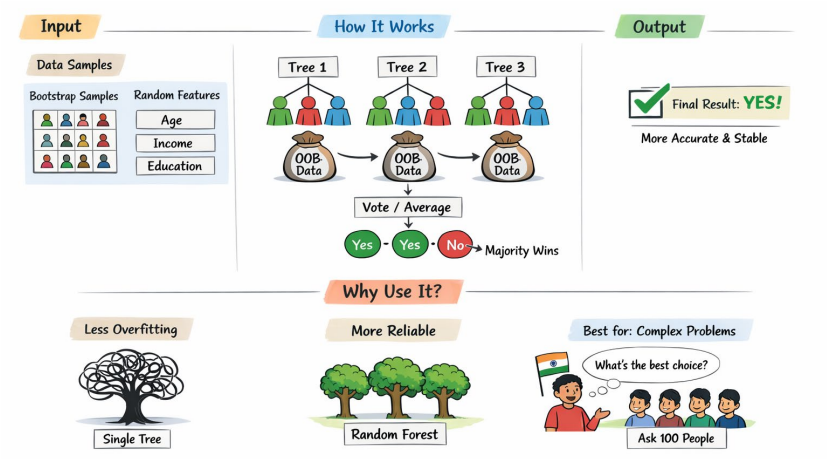
Random Forest is many decision trees working together. Each tree is trained on a slightly different dataset and uses a random subset of features.

Main benefit: reduces overfitting compared to one big tree (variance goes down).

Bagging = sampling rows with replacement; Feature randomness = sampling columns at each split.

Out-of-Bag (OOB) idea: for every tree, some rows are left out; those left-out rows can estimate accuracy without a separate validation set.

Trade-off: forests are harder to explain than a single tree, and model size can be bigger.



Definition

Random Forest is a group (ensemble) of many decision trees.

Final answer is decided by voting (classification) or averaging (regression).

Syntax / Formula

Bagging = training each tree on a random sample of rows (with replacement).

Random features = at each split, each tree looks at only a subset of features.

Why?

A single decision tree can overfit.

Many weak trees together usually perform better and are more stable.

Real-life Example (India)

Random Forest is like asking 100 different people for advice and taking the majority decision. Each tree sees a slightly different sample, so the final answer is more stable.

What?

Bootstrap sample: pick rows randomly with replacement.

Out-of-bag (OOB): rows not selected for a tree; used to estimate error without extra validation.

How?

Train N trees. Each tree gives a prediction. Final prediction = majority vote (Yes/No).

Let's Code!

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Reuse the same tiny HR dataset
data = [
```

```
    {"exp": 5, "state": "TN", "qualification": "B-Tech",
"hr_approved": 1},
    {"exp": 5, "state": "AP", "qualification": "B-Tech",
"hr_approved": 1},
    {"exp":10, "state": "UP", "qualification": "MBA",
"hr_approved": 0},
    {"exp": 5, "state": "UP", "qualification": "PhD",
"hr_approved": 0},
    {"exp": 7, "state": "TN", "qualification": "B-Tech",
"hr_approved": 1},
    {"exp": 2, "state": "MH", "qualification": "B-Tech",
"hr_approved": 0},
    {"exp": 8, "state": "KA", "qualification": "MBA",
"hr_approved": 0},
    {"exp": 6, "state": "TN", "qualification": "B-Tech",
"hr_approved": 1},
    {"exp": 4, "state": "AP", "qualification": "MBA",
"hr_approved": 0},
    {"exp": 9, "state": "TN", "qualification": "PhD",
"hr_approved": 0},
]


df = pd.DataFrame(data)
X = df[["exp", "state", "qualification"]]
y = df["hr_approved"]


X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.25, random_state=7)


preprocess = ColumnTransformer(
    transformers=[("cat", OneHotEncoder(handle_unknown="ignore"),
["state", "qualification"])],
    remainder="passthrough"
)
rf = RandomForestClassifier(
    n_estimators=50,
    random_state=7,
    max_depth=4
)
```

```
pipe = Pipeline(steps=[("prep", preprocess), ("model", rf)])

pipe.fit(X_train, y_train)

pred = pipe.predict(X_test)


print("Accuracy:", accuracy_score(y_test, pred))


# ===== ADDITION: Predict on unseen data =====

# Example: a new candidate with 6 years experience, from 'MH' with
qualification 'B-Tech'

unseen = pd.DataFrame({"exp": [6], "state": ["MH"], "qualification":
["B-Tech"]})

unseen_pred = pipe.predict(unseen)[0]

unseen_proba = pipe.predict_proba(unseen)[0, 1]  # probability of
class 1 (hr_approved)

print("\n--- Unseen data prediction ---")

print(f"Input: exp=6, state=MH, qualification=B-Tech")

print(f"Predicted outcome: {'Approved' if unseen_pred == 1 else 'Not
Approved'}")

print(f"Probability of approval: {unseen_proba:.2f}")Random Forest
also provides feature importance, which helps in explaining the
model.
```

Output - Accuracy: 0.6666666666666666

--- Unseen data prediction ---

Input: exp=6, state=MH, qualification=B-Tech

Predicted outcome: Approved

Probability of approval: 0.74

Chapter 7
Linear Regression (Predict a Number)

Extra Theory (Beginner Notes)

Linear Regression learns a straight-line relationship between inputs and a numeric output. It is a great first model for prediction problems with continuous targets.

Assumption (simple): output changes roughly linearly when inputs change.

Residual = Actual − Predicted. Good model → residuals look random (no pattern).

Multicollinearity: when two features carry almost the same information, coefficients can become unstable.

Regularization (optional): Ridge/Lasso add penalties to reduce overfitting and stabilize coefficients.

Definition

Linear Regression predicts a continuous value (like price, salary, GPA).

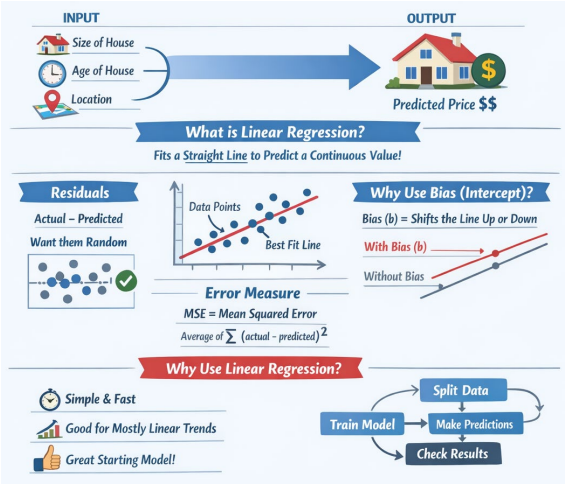
It finds a straight line (or plane) that best fits the data.

Syntax / Formula

Single feature: y = m*x + c

```
Multiple features: y = w1*x1 + w2*x2 + ... + b
```

Goal: choose weights to minimize error (usually MSE).



Math in Simple Words

We draw a line. For each point, we measure vertical distance from the line.

We square distances (so negative doesn't cancel positive).

We find the line that makes the total squared error smallest.

Error formula

MSE = average of (actual - predicted)^2

Why?

Very fast and simple baseline model.

Good when relationship is mostly linear.

Real-life Example (India)

Linear regression is used in real estate: price increases with area, and decreases with age of building. It fits a straight line to capture that trend.

What?

Weights (w) control slope; bias (b) controls shift.

R^2 tells how much your model explains the target.

How?

Split data → fit model → predict → measure R^2 and error.

If performance is poor, check outliers, scaling, or try non-linear models.

Lets Code

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, r2_score

# Set style for better looking plots
plt.style.use('seaborn-v0_8-darkgrid')
sns.set_palette("husl")

# Tiny dataset: study hours → GPA
data = [
    {"hours": 1, "gpa": 2.0},
    {"hours": 2, "gpa": 2.5},
    {"hours": 3, "gpa": 3.0},
    {"hours": 5, "gpa": 3.5},
    {"hours": 6, "gpa": 3.0},
    {"hours": 8, "gpa": 4.0},
    {"hours": 10, "gpa": 4.5},
    {"hours": 4, "gpa": 3.2},
    {"hours": 7, "gpa": 3.9},
    {"hours": 9, "gpa": 4.2},
]

df = pd.DataFrame(data)
X = df[["hours"]]
y = df["gpa"]
```

```
# Split data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Train model
model = LinearRegression()
model.fit(X_train, y_train)

# Predict on test set
y_pred = model.predict(X_test)

# Print model details and evaluation
print("=" * 50)
print("MODEL EVALUATION")
print("=" * 50)
print(f"Slope (coefficient): {model.coef_[0]:.4f}")
print(f"Intercept: {model.intercept_:.4f}")
print(f"MAE: {mean_absolute_error(y_test, y_pred):.4f}")
print(f"R² Score: {r2_score(y_test, y_pred):.4f}")
print()

# Display test results
print("TEST SET RESULTS:")
print("-" * 30)
test_results = pd.DataFrame({
    'Hours': X_test.values.flatten(),
    'Actual GPA': y_test.values,
    'Predicted GPA': y_pred,
    'Error': y_test.values - y_pred
})
print(test_results.to_string(index=False))
print()
```

```
# ============ UNSEEN DATA PREDICTION ============

print("=" * 50)

print("UNSEEN DATA PREDICTION")

print("=" * 50)


# Create unseen data points

unseen_hours = [[0], [2.5], [7.5], [11], [12], [15]]

unseen_predictions = model.predict(unseen_hours)


print("\nPredictions for unseen study hours:")

print("-" * 40)

for hours, pred in zip([h[0] for h in unseen_hours],
unseen_predictions):

    print(f"Study Hours: {hours:4.1f} → Predicted GPA: {pred:.3f}")


# ============ VISUALIZATION ============

fig, axes = plt.subplots(2, 2, figsize=(14, 10))

fig.suptitle('Linear Regression Analysis: Study Hours vs GPA',
fontsize=16, fontweight='bold')


# 1. Main Regression Plot with all data

ax1 = axes[0, 0]

# Create range for regression line

x_range = np.linspace(0, 16, 100).reshape(-1, 1)

y_range = model.predict(x_range)


# Plot regression line

ax1.plot(x_range, y_range, 'r-', linewidth=2, label='Regression
Line', alpha=0.7)


# Plot training data

ax1.scatter(X_train, y_train, color='blue', s=100, alpha=0.7,

           label='Training Data', edgecolors='black', linewidth=1)


# Plot test data (actual values)
```

```
ax1.scatter(X_test, y_test, color='green', s=120, alpha=0.9,
marker='s',
           label='Test Data (Actual)', edgecolors='black',
linewidth=1.5)


# Plot test predictions
ax1.scatter(X_test, y_pred, color='orange', s=150, alpha=0.8,
marker='^',
           label='Test Predictions', edgecolors='black',
linewidth=1.5)


# Plot unseen data predictions
unseen_hours_flat = [h[0] for h in unseen_hours]
ax1.scatter(unseen_hours_flat, unseen_predictions, color='purple',
s=200,
           alpha=0.9, marker='*', label='Unseen Predictions',
edgecolors='black',
           linewidth=1.5, zorder=5)


# Connect test actuals with predictions
for x_actual, y_actual, y_pred_val in zip(X_test.values.flatten(),
y_test, y_pred):
    ax1.plot([x_actual, x_actual], [y_actual, y_pred_val], 'gray',
             linestyle='--', alpha=0.5, linewidth=1)


ax1.set_xlabel('Study Hours', fontsize=12)
ax1.set_ylabel('GPA', fontsize=12)
ax1.set_title('Regression Line with All Data Points', fontsize=14)
ax1.legend(loc='upper left')
ax1.grid(True, alpha=0.3)
ax1.set_xlim(-0.5, 16)
ax1.set_ylim(1.5, 5.5)


# 2. Residual Plot (Errors)
ax2 = axes[0, 1]
residuals = y_test - y_pred
ax2.scatter(y_pred, residuals, color='red', s=100, alpha=0.7,
```

```
           edgecolors='black', linewidth=1)
ax2.axhline(y=0, color='black', linestyle='--', linewidth=1,
alpha=0.7)
ax2.set_xlabel('Predicted GPA', fontsize=12)
ax2.set_ylabel('Residuals (Actual - Predicted)', fontsize=12)
ax2.set_title('Residual Plot', fontsize=14)
ax2.grid(True, alpha=0.3)


# Add text box with error statistics
error_stats = f'MAE: {mean_absolute_error(y_test, y_pred):.3f}\nMax
Error: {abs(residuals).max():.3f}\nMin Error:
{abs(residuals).min():.3f}'
props = dict(boxstyle='round', facecolor='wheat', alpha=0.5)
ax2.text(0.05, 0.95, error_stats, transform=ax2.transAxes,
fontsize=10,
        verticalalignment='top', bbox=props)


# 3. Actual vs Predicted Comparison
ax3 = axes[1, 0]
# Perfect prediction line
perfect_line = np.linspace(min(y_test.min(), y_pred.min()),
                           max(y_test.max(), y_pred.max()), 100)
ax3.plot(perfect_line, perfect_line, 'k--', alpha=0.5,
label='Perfect Prediction')


# Scatter plot of actual vs predicted
ax3.scatter(y_test, y_pred, color='green', s=100, alpha=0.7,
           edgecolors='black', linewidth=1, label='Test Points')


ax3.set_xlabel('Actual GPA', fontsize=12)
ax3.set_ylabel('Predicted GPA', fontsize=12)
ax3.set_title('Actual vs Predicted Values', fontsize=14)
ax3.legend()
ax3.grid(True, alpha=0.3)


# Add R² score to plot
ax3.text(0.05, 0.95, f'R² Score: {r2_score(y_test, y_pred):.3f}',
```

```
        transform=ax3.transAxes, fontsize=11,
        verticalalignment='top', bbox=props)

# 4. Unseen Data Predictions (Bar Chart)
ax4 = axes[1, 1]
hours_labels = [f'{h:.1f}' if h % 1 != 0 else f'{int(h)}' for h in
unseen_hours_flat]
bars = ax4.bar(range(len(unseen_hours_flat)), unseen_predictions,
               color='purple', alpha=0.7, edgecolor='black')

# Add value labels on top of bars
for i, (bar, val) in enumerate(zip(bars, unseen_predictions)):
    height = bar.get_height()
    ax4.text(bar.get_x() + bar.get_width()/2., height + 0.05,
            f'{val:.2f}', ha='center', va='bottom', fontsize=10)

ax4.set_xlabel('Study Hours (Unseen)', fontsize=12)
ax4.set_ylabel('Predicted GPA', fontsize=12)
ax4.set_title('Predictions for Unseen Data', fontsize=14)
ax4.set_xticks(range(len(unseen_hours_flat)))
ax4.set_xticklabels(hours_labels)
ax4.grid(True, alpha=0.3, axis='y')
ax4.set_ylim(0, 5.5)

# Adjust layout
plt.tight_layout(rect=[0, 0.03, 1, 0.95])

# Show the plots
plt.show()

# Additional: Summary Statistics
print("\n" + "=" * 50)
print("DATA SUMMARY")
print("=" * 50)
print(f"Total data points: {len(df)}")
```

```
print(f"Training samples: {len(X_train)}")

print(f"Test samples: {len(X_test)}")

print(f"Unseen predictions: {len(unseen_hours)}")

print(f"\nTraining range: {X_train['hours'].min()} to
{X_train['hours'].max()} hours")

print(f"Test range: {X_test['hours'].min()} to
{X_test['hours'].max()} hours")

print(f"Unseen range: {min(unseen_hours_flat)} to
{max(unseen_hours_flat)} hours")


# Print regression equation

print(f"\nRegression Equation: GPA = {model.intercept_:.3f} +
{model.coef_[0]:.3f} * Hours")
```

Result

Model Evaluation

Slope (coefficient): 0.2452

Intercept: 2.0152

MAE: 0.0870

R^2 Score: 0.9710

Test set prediction

Hours	Actual GPA	Predicted GPA	Error
7	3.9	3.731481	0.168519
2	2.5	2.505556	-0.005556

Unseen data prediction

Predictions for unseen study hours:

Study Hours: 0.0 → Predicted GPA: 2.015

Study Hours: 2.5 → Predicted GPA: 2.628

Study Hours: 7.5 → Predicted GPA: 3.854

Study Hours: 11.0 → Predicted GPA: 4.712

Study Hours: 12.0 → Predicted GPA: 4.957

Study Hours: 15.0 → Predicted GPA: 5.693

Total data points: 10

Training samples: 8

Test samples: 2

Unseen predictions: 6

Training range: 1 to 10 hours

Test range: 2 to 7 hours

Unseen range: 0 to 15 hours

Regression Equation: GPA = 2.015 + 0.245 * Hours

What to take care: Scaling is not mandatory for LinearRegression, but helps when features have very different ranges.

Key Takeaways

- Linear Regression predicts a number (continuous output).
- It fits the best line/plane that minimizes squared error.
- Check assumptions and outliers; scale features if needed.
- Good baseline for many regression problems.

Chapter 8
Logistic Regression (Predict Yes/No)

Extra Theory (Beginner Notes)

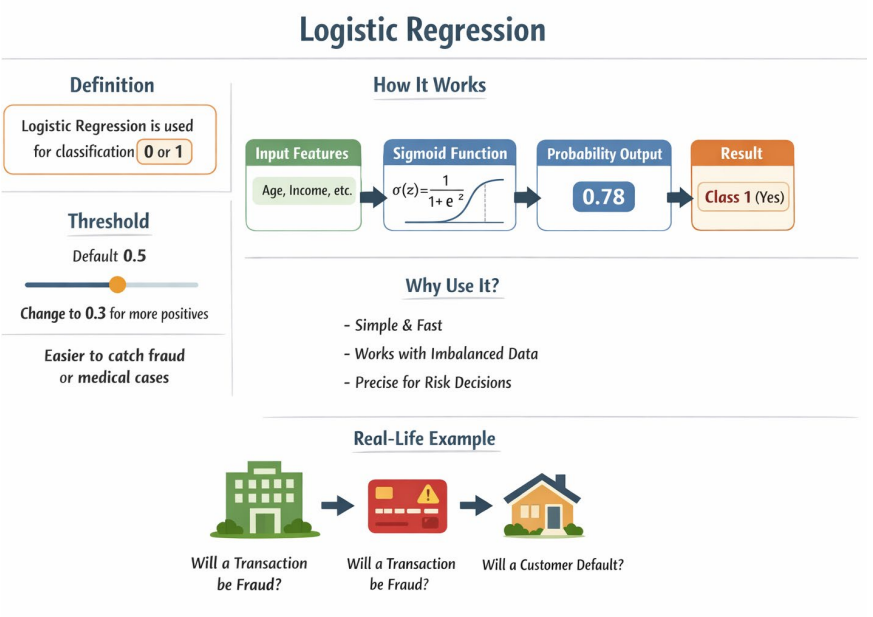
Logistic Regression predicts probability for a class (0/1). It uses the sigmoid function to map any number to a value between 0 and 1.

It is still a linear model, but the output is a probability (not a raw number).

Threshold matters: default is 0.5, but for fraud/medical problems you often change it (e.g., 0.3) to catch more positives.

Regularization helps when there are many features or noisy data.

Best metrics for imbalanced data: Precision, Recall, F1, ROC-AUC (not only accuracy).



Definition

Logistic Regression is used for classification (usually binary: 0/1).

It outputs a probability, then converts it into a class using a threshold (often 0.5).

Syntax / Formula

```
z = w·x + b  (dot product)
sigmoid(z) = 1 / (1 + e^(-z))  → gives probability between 0 and 1
```

If prob >= 0.5 → class 1 else class 0

Math in Simple Words

Linear regression can output any number (like -5 or 12).

For classification, we need 0 to 1 probability, so we pass the score through sigmoid.

Why?

Simple, fast, and works well for many business problems.

Easy to explain and debug.

Real-life Example

Logistic regression is used in banks to predict Yes/No outcomes: will a customer default? will a transaction be fraud? It outputs a probability between 0 and 1.

What?

It learns weights that separate classes.

The output is probability of class 1.

How?

Scale numeric features if needed.

Fit model, check confusion matrix, precision/recall.

For imbalanced data, consider class_weight or SMOTE.

Let's Code!

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix

# Tiny dataset: pass/fail based on hours + attendance
data = [
    {"hours": 1, "attendance": 60, "passed": 0},
    {"hours": 2, "attendance": 65, "passed": 0},
    {"hours": 3, "attendance": 70, "passed": 0},
    {"hours": 4, "attendance": 75, "passed": 1},
    {"hours": 5, "attendance": 80, "passed": 1},
    {"hours": 6, "attendance": 85, "passed": 1},
    {"hours": 7, "attendance": 90, "passed": 1},
    {"hours": 2, "attendance": 80, "passed": 0},
    {"hours": 8, "attendance": 55, "passed": 0},
    {"hours": 9, "attendance": 92, "passed": 1},
    {"hours": 6, "attendance": 72, "passed": 1},
    {"hours": 3, "attendance": 85, "passed": 0},
]

df = pd.DataFrame(data)
X = df[["hours", "attendance"]]
y = df["passed"]

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.25, random_state=1)

model = LogisticRegression(max_iter=1000)
model.fit(X_train, y_train)

pred = model.predict(X_test)
print("Accuracy:", accuracy_score(y_test, pred))
print("Confusion Matrix:\n", confusion_matrix(y_test, pred))
```

```
# Probability output
proba = model.predict_proba(X_test)[:, 1]
print("Probabilities:", proba)

# ===== ADDITION: Predict on unseen data =====
# Example: a student with 5.5 study hours and 82% attendance
unseen = pd.DataFrame({"hours": [5.5], "attendance": [82]})
unseen_pred = model.predict(unseen)[0]
unseen_proba = model.predict_proba(unseen)[0, 1]

print("\n--- Unseen data prediction ---")
print(f"Input: hours = 5.5, attendance = 82%")
print(f"Predicted outcome: {'Pass' if unseen_pred == 1 else
'Fail'}")
print(f"Probability of passing: {unseen_proba:.2f}")
```

Output

Accuracy: 0.6666666666666666

Confusion Matrix:

[[1 0]

[1 1]]

Probabilities: [0.03364119 0.18362244 0.5923753]

--- Unseen data prediction ---

Input: hours = 5.5, attendance = 82%

Predicted outcome: Pass

Probability of passing: 0.77

Key Takeaways

- Logistic Regression predicts probability of Yes/No.
- Decision threshold controls false alarms vs misses.
- Scaling often improves training stability.
- It is simple, strong, and a great baseline.

Chapter 9
K-Nearest Neighbors (Distance Based)

Extra Theory (Beginner Notes)

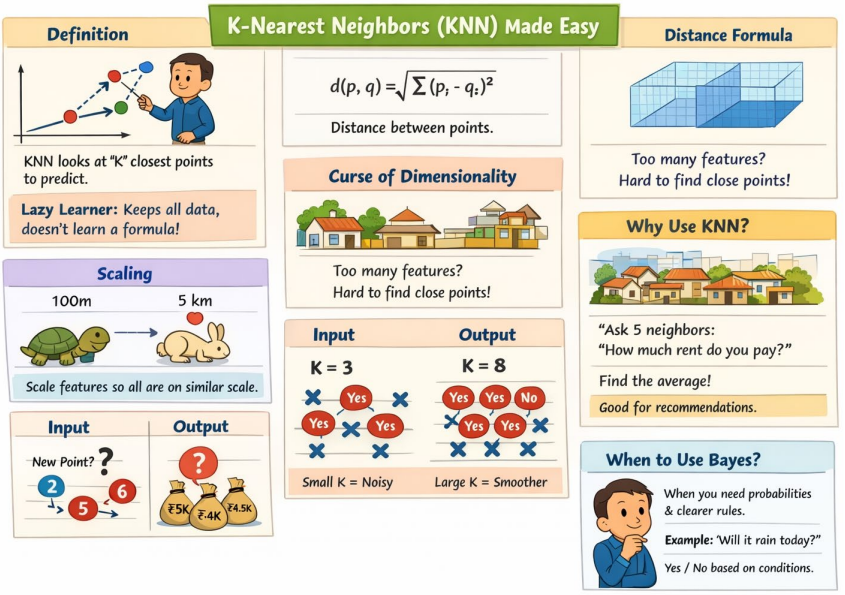
KNN does not learn a formula. It keeps training data and predicts by looking at the closest examples.

Scaling is important: if one feature has big numbers, it dominates distance (use StandardScaler).

Curse of dimensionality: with too many features, distances become less meaningful and KNN gets weaker.

Choosing k: small k → noisy; big k → smoother but may miss local patterns.

KNN can be slow for large datasets because it searches neighbors at prediction time.



Definition

KNN predicts by looking at the 'K' most similar training points.

It is called a lazy learner because it does not build a big model during training.

Syntax / Formula

Distance (Euclidean) between two points p and q:

```
d(p,q) = sqrt( Σ (p_i - q_i)^2 )
```

Math in Simple Words

Imagine all points on a graph.

To predict a new point, find nearest neighbors using distance.

Use majority vote among neighbors.

Why?

Works well when similar people behave similarly (example: recommendation).

Real-life Example (India)

KNN is like asking your nearest 5 neighbors in your society: 'How much rent do you pay?' and using their average. It predicts using similar past cases.

What?

K is a hyperparameter.

Small K → noisy; large K → may mix classes.

How?

Scale features (very important for distance).

Choose K using validation (try 1..20).

Let's Code!

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix

# Tiny dataset: pass/fail based on hours + attendance
data = [
    {"hours": 1, "attendance": 60, "passed": 0},
    {"hours": 2, "attendance": 65, "passed": 0},
    {"hours": 3, "attendance": 70, "passed": 0},
    {"hours": 4, "attendance": 75, "passed": 1},
    {"hours": 5, "attendance": 80, "passed": 1},
    {"hours": 6, "attendance": 85, "passed": 1},
    {"hours": 7, "attendance": 90, "passed": 1},
    {"hours": 2, "attendance": 80, "passed": 0},
    {"hours": 8, "attendance": 55, "passed": 0},
    {"hours": 9, "attendance": 92, "passed": 1},
    {"hours": 6, "attendance": 72, "passed": 1},
    {"hours": 3, "attendance": 85, "passed": 0},
]

df = pd.DataFrame(data)
X = df[["hours", "attendance"]]
y = df["passed"]

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.25, random_state=1)

model = LogisticRegression(max_iter=1000)
model.fit(X_train, y_train)

pred = model.predict(X_test)
print("Accuracy:", accuracy_score(y_test, pred))
print("Confusion Matrix:\n", confusion_matrix(y_test, pred))
```

```
# Probability output
proba = model.predict_proba(X_test)[:, 1]
print("Probabilities:", proba)

# ===== NEW PART: Predict on unseen data =====
# Example of a new student: studied 4.5 hours, attendance 78%
unseen = pd.DataFrame({"hours": [4.5], "attendance": [78]})
unseen_pred = model.predict(unseen)[0]
unseen_proba = model.predict_proba(unseen)[0, 1]

print("\n--- Unseen data prediction ---")
print(f"Input: hours = 4.5, attendance = 78%")
print(f"Predicted outcome: {'Pass' if unseen_pred == 1 else
'Fail'}")
print(f"Probability of passing: {unseen_proba:.2f}")
```

What to take care: If one feature range is huge (spend) and another is small (calls), scaling is mandatory.

Output

Accuracy: 0.6666666666666666

Confusion Matrix:

[[1 0]

[1 1]]

Probabilities: [0.03364119 0.18362244 0.5923753]

--- Unseen data prediction ---

Input: hours = 4.5, attendance = 78%

Predicted outcome: Fail

Probability of passing: 0.39

Key Takeaways

- KNN predicts using 'similar' past points.
- Distance choice and feature scaling matter a lot.
- It can be slow on very large datasets.

Chapter 10
Support Vector Machines (Margin Based)

Extra Theory (Beginner Notes)

SVM tries to draw a boundary that separates classes with the widest possible margin. Only a few critical points (support vectors) define the boundary.

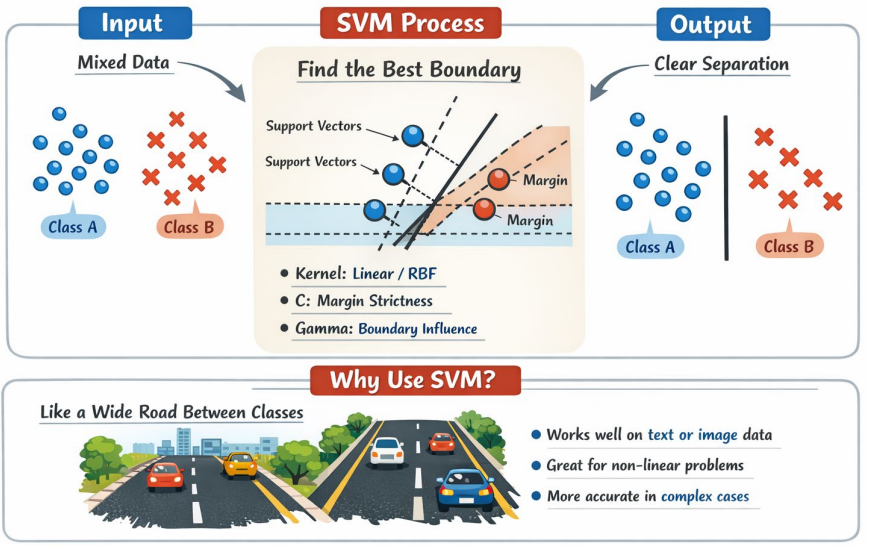
Works well on medium-sized datasets; can be strong for text or image features.

Kernel trick: transforms data into higher space to separate non-linear patterns.

C controls strictness: high C → fewer training errors but may overfit; low C → wider margin but more training errors.

Gamma (RBF) controls influence: high gamma → very tight boundary; low gamma → smoother boundary.

Scaling is usually required for SVM.



Definition

SVM finds a boundary (line/plane) that separates classes with the maximum margin.The points touching the margin are called support vectors.

Syntax / Formula

Goal: find a hyperplane that maximizes margin.

Important knobs:

- kernel: 'linear', 'rbf', 'poly'
- C: penalty for misclassification (controls margin vs errors)
- gamma: influence range for RBF kernel

Why?

SVM can work well when data is not linearly separable (using kernels).

Real-life Example (India)

SVM is like drawing the widest possible road between two groups (yes vs no) so that new points fall clearly on one side. It is useful when classes are separable with a good margin.

What?

Linear kernel: for almost linear data.

RBF kernel: for non-linear boundaries.

How?

Scale data. Try linear first. If not good, try RBF.

Tune C and gamma using GridSearchCV.

Let's Code!

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score


# Tiny dataset: two features, two classes
X = np.array([
    [1.0, 1.0],
    [1.2, 1.1],
    [0.8, 0.9],
    [3.0, 3.1],
    [3.2, 2.9],
    [2.9, 3.3],
    [1.1, 0.7],
    [3.3, 3.0],
])
y = np.array([0,0,0,1,1,1,0,1])


X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.25, random_state=0)


# Note: Added probability=True to enable predict_proba
pipe = Pipeline(steps=[
    ("scaler", StandardScaler()),
    ("svm", SVC(kernel="rbf", C=3.0, gamma="scale",
probability=True))
])


pipe.fit(X_train, y_train)
pred = pipe.predict(X_test)
print("Accuracy:", accuracy_score(y_test, pred))
```

```
# ===== ADDITION: Predict on unseen data =====
# Example: a new point [2.0, 2.0] (midway between clusters)
unseen = np.array([[2.0, 2.0]])
unseen_pred = pipe.predict(unseen)[0]
unseen_proba = pipe.predict_proba(unseen)[0]


print("\n--- Unseen data prediction ---")
print(f"Input point: [2.0, 2.0]")
print(f"Predicted class: {unseen_pred}")
print(f"Class probabilities: [prob(class 0)={unseen_proba[0]:.3f},
prob(class 1)={unseen_proba[1]:.3f}]")
```

Output

Accuracy: 1.0
--- Unseen data prediction ---

Input point: [2.0, 2.0]

Predicted class: 0

Class probabilities: [prob(class 0)=0.000, prob(class 1)=1.000]

Key Takeaways

- SVM focuses on margin - a strong separation boundary.
- Kernel trick can capture non-linear patterns.
- Scaling is important for distance-based methods like SVM.
- Works well in medium-sized feature spaces.

Chapter 11
Naïve Bayes (Probability Based)

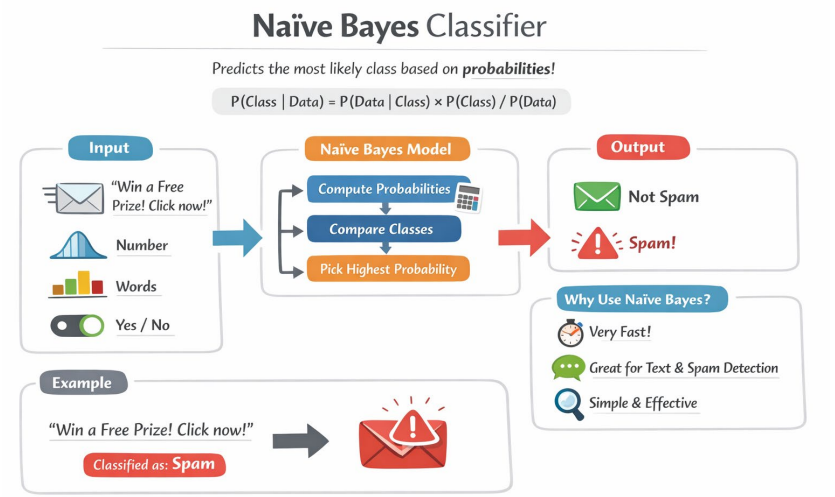
Extra Theory (Beginner Notes)

Naïve Bayes uses probability to predict the most likely class. It assumes features are independent (which is often not true, but it still works well).

Very fast and works surprisingly well for text classification (spam detection).

Common types: GaussianNB (continuous), MultinomialNB (word counts), BernoulliNB (0/1 features).

Laplace smoothing avoids zero probability when a category never appeared in training.



Definition

Naïve Bayes is a classification algorithm based on Bayes' Theorem.

It assumes features are independent (this is why it is called 'naïve').

Syntax / Formula

```
Bayes Rule: P(A|B) = P(B|A) * P(A) / P(B)
```

In ML words: Posterior = Likelihood * Prior / Evidence

Math in Simple Words

Prior: your belief before seeing new data.

Likelihood: how well the new data matches a class.

Posterior: updated belief after seeing new data.

We choose the class with higher posterior probability.

Why?

Very fast, works great for text classification (spam detection).

Real-life Example

Naive Bayes is widely used for email spam and SMS filtering. If a message contains words like 'win', 'free', 'offer', it pushes the probability towards spam.

What?

Even though the independence assumption is not always true, Naïve Bayes often works surprisingly well.

How?

Compute probability for each class. Pick the class with maximum probability.

Let's Code!

Tiny example: predict job selection (0/1) based on categorical features.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score


# Tiny categorical dataset
data = [
    {"gender":"M", "occupation":"Self",    "skill":"ML",
"selected":0},
```

```
    {"gender":"F", "occupation":"Govt",    "skill":"C++",
"selected":1},

    {"gender":"M", "occupation":"Govt",    "skill":"ML",
"selected":0},

    {"gender":"M", "occupation":"Private", "skill":"ML",
"selected":1},

    {"gender":"F", "occupation":"Salary",  "skill":"C#",
"selected":1},

    {"gender":"M", "occupation":"Self",    "skill":"C++",
"selected":0},

    {"gender":"F", "occupation":"Private", "skill":"ML",
"selected":1},

    {"gender":"M", "occupation":"Salary",  "skill":"C#",
"selected":1},

]

df = pd.DataFrame(data)

X = pd.get_dummies(df[["gender","occupation","skill"]])  # one-hot
encoding

y = df["selected"]

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.25, random_state=2)

model = GaussianNB()

model.fit(X_train, y_train)

pred = model.predict(X_test)

print("Accuracy:", accuracy_score(y_test, pred))


# Predict a new profile

new_person =
pd.DataFrame([{"gender":"M","occupation":"Self","skill":"ML"}])

new_person_enc =
pd.get_dummies(new_person).reindex(columns=X.columns, fill_value=0)

print("New prediction:", model.predict(new_person_enc)[0])
```

What to take care: Always align dummy columns for new data (same columns as training).

Key Takeaways

- Naive Bayes is fast and strong for text classification.
- Uses Bayes' rule with independence assumption.
- Works well even with small data.
- Outputs probabilities that are easy to interpret.

Chapter 12
Clustering and K-Means (Unsupervised)

Extra Theory (Beginner Notes)

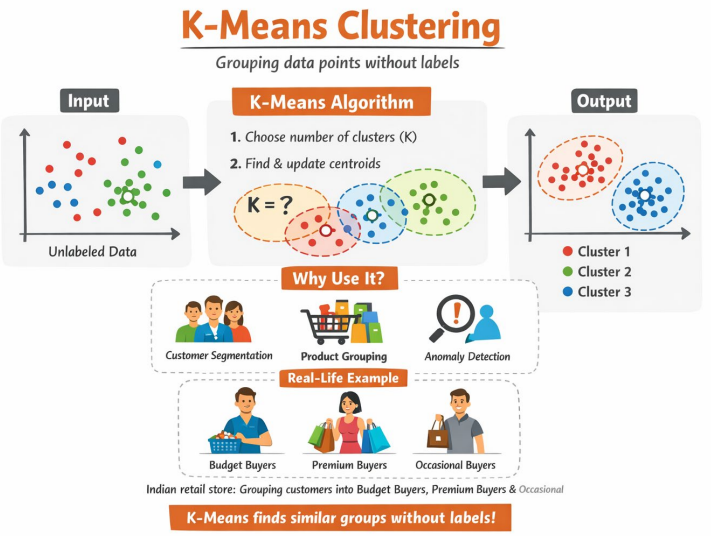
Clustering groups similar items without labels. K-Means is the most popular clustering method for beginners.

K-Means assumes clusters are roughly round and similar size. If clusters are long/irregular, results may be poor.

Initialization matters: K-Means++ gives better starting points than random.

Scaling matters: features with bigger numbers dominate distance (use StandardScaler).

Elbow method is a heuristic. Sometimes you also check Silhouette Score for cluster quality.



Definition

Clustering groups similar data points together without using labels (no target column).

K-Means is the most popular clustering algorithm.

Syntax / Formula

```
Distance (Euclidean): d = sqrt( (x2-x1)^2 + (y2-y1)^2 )
```

K-Means repeats: Assign → Update centroid → Repeat

Why?

Used for customer segmentation, grouping similar products, anomaly detection.

Real-life Example (India)

K-means clustering is like grouping customers in an Indian retail store: budget buyers, premium buyers, and occasional buyers - without using labels.

What?

K = number of clusters. Centroid = average location of points in a cluster.

How?

Choose K (often using elbow method). Scale features, then run K-Means.

Interpret clusters using business meaning.

Let's Code! (Tiny customer segmentation)

```
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
# Tiny dataset: Price score vs Brand score
data = [
    {"customer":"A", "price_score":2, "brand_score":4},
    {"customer":"B", "price_score":8, "brand_score":2},
    {"customer":"C", "price_score":9, "brand_score":3},
    {"customer":"D", "price_score":1, "brand_score":5},
    {"customer":"E", "price_score":8, "brand_score":1},
    {"customer":"F", "price_score":2, "brand_score":5},
```

```
    {"customer":"G", "price_score":9, "brand_score":2},
]

df = pd.DataFrame(data)
X = df[["price_score", "brand_score"]]

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

kmeans = KMeans(n_clusters=2, random_state=0, n_init="auto")
labels = kmeans.fit_predict(X_scaled)

df["cluster"] = labels
print("Clustered data:")
print(df.sort_values("cluster"))

# ===== ADDITION: Predict cluster for a new unseen customer =====
# New customer with price_score=5, brand_score=3
new_customer = pd.DataFrame({"price_score": [5], "brand_score": [3]})
new_scaled = scaler.transform(new_customer)          # scale using same scaler
new_cluster = kmeans.predict(new_scaled)[0]          # predict cluster
# Optional: get distances to cluster centers
distances = kmeans.transform(new_scaled)[0]          # distances to each center

print("\n--- Unseen customer prediction ---")
print(f"New customer: price_score=5, brand_score=3")
print(f"Assigned to cluster: {new_cluster}")
print(f"Distances to cluster centers: {distances}")
```

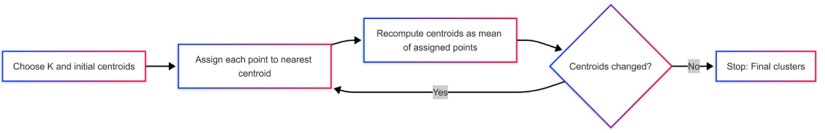
Mermaid: K-Means loop

output

New customer: price_score=5, brand_score=3

Assigned to cluster: 1

Distances to cluster centers: [1.50352474 1.23310257]



Key Takeaways

- Clustering finds groups without labels.
- K-means needs you to pick K (number of clusters).
- Scale features before clustering.
- Use clustering for segmentation and anomaly hints.

Chapter 13
PCA (Dimensionality Reduction)

Extra Theory (Beginner Notes)

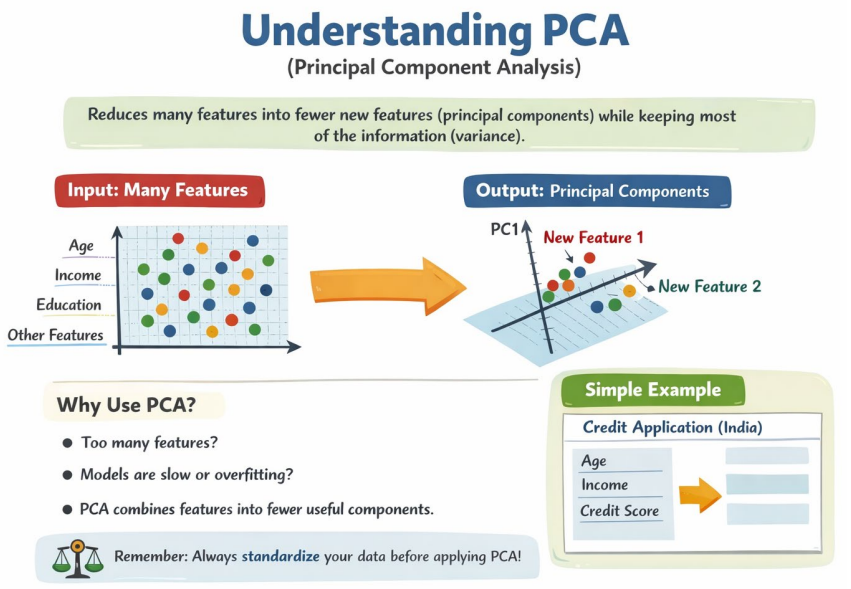
PCA compresses many correlated features into fewer new features (principal components) that keep most information (variance).

PCA is unsupervised: it does not look at target labels.

Use PCA when you have too many features and models become slow or overfit.

Interpretation is harder because components are mixtures of original features.

Always standardize before PCA for fair contribution of all features.



Definition

PCA reduces many features into fewer new features (principal components).

It keeps as much information (variance) as possible.

Syntax / Formula

Input: standardized data matrix X

Output: Z = X * W (W contains principal directions)

We keep top components that cover, say, 95% variance.

Math in Simple Words

Imagine data points in 3D. PCA finds the best 2D plane that keeps most spread.

Eigenvector = direction of maximum variance.

Eigenvalue = how much variance that direction has.

Why?

Too many features can make models slow and noisy.

PCA can help visualization and sometimes improves performance.

Real-life Example (India)

PCA is like compressing a big form into a few key fields. In credit scoring, many correlated variables can be reduced to fewer components for easier modeling and visualization.

What?

PCA creates new features that are combinations of old features.

New features are uncorrelated.

How?

Scale data first (very important).

Fit PCA on train set, transform train/test.

Choose number of components by explained_variance_ratio_.

Let's Code!

```
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

# Tiny dataset with 4 features (imagine sensor readings)
X = np.array([
    [10, 200,  5, 1000],
    [12, 210,  6, 1100],
    [11, 190,  4,  980],
    [ 9, 205,  5, 1050],
    [13, 220,  7, 1150],
    [ 8, 180,  3,  900],
])

# Fit scaler and PCA
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

pca = PCA(n_components=2)
Z = pca.fit_transform(X_scaled)

print("Explained variance ratio:", pca.explained_variance_ratio_)
print("Reduced data (2 columns):\n", Z)

# ===== ADDITION: Transform a new unseen data point =====
# New observation with the same 4 features
new_point = np.array([[10, 195, 5, 1020]])   # example values
new_scaled = scaler.transform(new_point)      # scale using
previously fitted scaler
new_reduced = pca.transform(new_scaled)       # project using fitted
PCA

print("\n--- Unseen data transformation ---")
print("New point (original):", new_point[0])
```

```
print("New point (scaled):", new_scaled[0])
print("New point (reduced to 2 PCs):", new_reduced[0])
```

What to take care: PCA should be fitted only on training data to avoid leakage.

output

New point (original): [10 195 5 1020]

New point (scaled): [-0.29277002 -0.4472136 0. -0.12247449]

New point (reduced to 2 PCs): [-0.42500135 -0.05298006]

Key Takeaways

- PCA reduces dimensions while keeping most variance.
- Useful for visualization, speed, and noise reduction.
- Do scaling before PCA.
- PCA components are combinations of original features.

Chapter 14
Practical Tips
(Encoding, Scaling, Imbalanced Data, CV)

Real-life Example

Practical tips show up daily: one-hot encoding for 'city', scaling for 'income', handling imbalanced fraud data, and cross-validation for stable evaluation.

14.1 Encoding (One-Hot vs Label)

Use One-Hot when categories have no order: color, state, occupation.

Use Label Encoding when categories have an order: low < medium < high.

14.2 Scaling (When and why)

Scaling is required for distance/margin based models: KNN, SVM, K-Means.

Scaling is usually not required for tree models: Decision Tree, Random Forest.

14.3 Cross Validation (CV)

Cross validation tests the model multiple times on different train/validation splits. This gives a more reliable performance estimate.

K-Fold CV: split into K parts, train on K-1 parts, validate on 1 part (repeat).

Let's Code! (GridSearchCV example)

```
import pandas as pd
from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeClassifier


# Tiny numeric dataset
X = [
```

```
    [1, 60], [2, 65], [3, 70], [4, 75], [5, 80],
    [6, 85], [7, 90], [2, 80], [8, 55], [9, 92]
]
y = [0,0,0,1,1,1,1,0,0,1]

params = {
    "max_depth": [1,2,3,4],
    "min_samples_split": [2,3,4]
}

grid = GridSearchCV(DecisionTreeClassifier(random_state=0), params, cv=3)
grid.fit(X, y)

print("Best params:", grid.best_params_)
print("Best CV score:", grid.best_score_)
```

output

Best params: {'max_depth': 1, 'min_samples_split': 2}

Best CV score: 0.611111111111111

14.4 Imbalanced Data (When 0 and 1 counts are very different)

Problem: model may predict only majority class and still show high accuracy.

Solutions: class_weight, oversampling, undersampling, SMOTE.

14.5 Pruning (for Decision Trees)

Pruning means reducing tree size to avoid overfitting.

Pre-pruning: limit depth, min_samples_leaf, min_samples_split.

Post-pruning: grow the tree, then cut weak branches (cost complexity pruning).

Key Takeaways

- Encoding + scaling + CV are practical superpowers.
- Handle imbalance with class weights or sampling.
- Use pipelines to avoid leakage in preprocessing.
- Always validate on unseen data.

Chapter 15
Production: Build an API + Deploy on GCP Cloud Run

In production, users do not run notebooks. We expose the model as an API.

Definition

Model serving means: load the trained model and return predictions for new input.

Why?

So a website/mobile app/another service can call your model using HTTP.

Real-life Example (India)

Serving a model as an API is like exposing a 'prediction service' that any app can call. On GCP Cloud Run, you can deploy it quickly and scale when traffic increases.

What?

We will build a small Flask API and deploy it to GCP Cloud Run using Docker.

How?

Step 1: Train model and save as a file (joblib/pickle).

Step 2: Create Flask app that loads model and exposes /predict.

Step 3: Containerize with Dockerfile.

Step 4: Deploy to Cloud Run.

Let's Code! (Train + save model)

```
import pandas as pd
import joblib
from sklearn.linear_model import LogisticRegression

data = [
    {"hours": 1, "attendance": 60, "passed": 0},
    {"hours": 2, "attendance": 65, "passed": 0},
    {"hours": 4, "attendance": 75, "passed": 1},
    {"hours": 5, "attendance": 80, "passed": 1},
    {"hours": 7, "attendance": 90, "passed": 1},
    {"hours": 8, "attendance": 55, "passed": 0},
]

df = pd.DataFrame(data)
X = df[["hours","attendance"]]
y = df["passed"]

model = LogisticRegression(max_iter=1000)
model.fit(X, y)

joblib.dump(model, "model.joblib")
print("Saved model.joblib")
```

Flask API (app.py)

```
from flask import Flask, request, jsonify
import joblib
import numpy as np

app = Flask(__name__)
model = joblib.load("model.joblib")

@app.get("/health")
def health():
```

```
    return {"status": "ok"}


@app.post("/predict")
def predict():
    payload = request.get_json(force=True)
    hours = float(payload["hours"])
    attendance = float(payload["attendance"])

    X = np.array([[hours, attendance]])
    prob = float(model.predict_proba(X)[0, 1])
    pred = int(prob >= 0.5)

    return jsonify({"passed": pred, "probability": prob})

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8080)
```

requirements.txt

```
flask
numpy
joblib
scikit-learn
```

Dockerfile

```
FROM python:3.11-slim


WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY model.joblib app.py ./
EXPOSE 8080


CMD ["python", "app.py"]
```

Deploy to Cloud Run (commands)

```
# 1) Build container image
gcloud builds submit --tag gcr.io/<PROJECT_ID>/ml-api:1.0


# 2) Deploy to Cloud Run
gcloud run deploy ml-api \
  --image gcr.io/<PROJECT_ID>/ml-api:1.0 \
  --platform managed \
  --region <REGION> \
  --allow-unauthenticated
```

What to take care: Save the same preprocessing steps (scaler/encoder) with the model. Use a Pipeline to avoid mistakes.

Key Takeaways

- API deployment turns your model into a usable product.
- Keep input validation and logging from day 1.
- Use versioning for models (v1, v2).
- Monitor latency and accuracy drift in production.

Chapter 16
Production: Deploy on AWS Lambda (Container) + API Gateway

Definition

AWS Lambda runs your code on demand. You pay only when it runs.

Why?

Great for small/medium traffic APIs and simple ML inference.

Real-life Example (India)

On AWS Lambda, you pay per request. This is useful for small prediction APIs like 'price estimate' or 'spam check' that are called only sometimes.

What?

We deploy the same model as a Lambda container image and expose it through API Gateway.

How?

Step 1: Train + save model (same as Chapter 15).

Step 2: Write Lambda handler that loads model and returns prediction.

Step 3: Build Docker image using AWS Lambda base image.

Step 4: Push to ECR and create Lambda function.

Lambda handler (app.py)

```
import json
import joblib
import numpy as np

model = joblib.load("model.joblib")

def handler(event, context):
```

```
    # event["body"] is a string in API Gateway by default
    body = event.get("body", "{}")
    payload = json.loads(body)

    hours = float(payload["hours"])
    attendance = float(payload["attendance"])

    X = np.array([[hours, attendance]])
    prob = float(model.predict_proba(X)[0, 1])
    pred = int(prob >= 0.5)

    return {
        "statusCode": 200,
        "headers": {"content-type": "application/json"},
        "body": json.dumps({"passed": pred, "probability": prob})
    }
```

requirements.txt

```
numpy
joblib
scikit-learn
```

Dockerfile (Lambda container)

```
# AWS Lambda Python base image
FROM public.ecr.aws/lambda/python:3.11

# Copy dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy model + handler
COPY model.joblib app.py ./

# Set the handler
CMD ["app.handler"]
```

Deploy steps (high level)

```
# 1) Build image locally

docker build -t ml-lambda:1.0 .


# 2) Create ECR repo and push (simplified)

aws ecr create-repository --repository-name ml-lambda

aws ecr get-login-password | docker login --username AWS --password-
stdin <ACCOUNT_ID>.dkr.ecr.<REGION>.amazonaws.com

docker tag ml-lambda:1.0
<ACCOUNT_ID>.dkr.ecr.<REGION>.amazonaws.com/ml-lambda:1.0

docker push <ACCOUNT_ID>.dkr.ecr.<REGION>.amazonaws.com/ml-
lambda:1.0


# 3) Create Lambda from container image (then connect API Gateway)

aws lambda create-function --function-name ml-api \

  --package-type Image \

  --code ImageUri=<ECR_IMAGE_URI> \

  --role <LAMBDA_EXECUTION_ROLE_ARN>
```

What to take care: Keep container image small and inference fast (Lambda has time/memory limits).

Key Takeaways

- Serverless is cost-effective for low/medium traffic.
- Cold start can add delay - plan for it.
- Package dependencies carefully for deployment.
- Use logs/metrics to debug in production.

Appendix A
Common Interview Questions (With Simple Answers)

Q: What is the difference between supervised and unsupervised learning?

A: Supervised: has target labels (y). Unsupervised: no labels; we find patterns (clusters).

Q: Why do we split train and test?

A: To check if the model works on unseen data. Otherwise, we can get fake high accuracy.

Q: When do we need scaling?

A: KNN, SVM, K-Means. Not mandatory for trees.

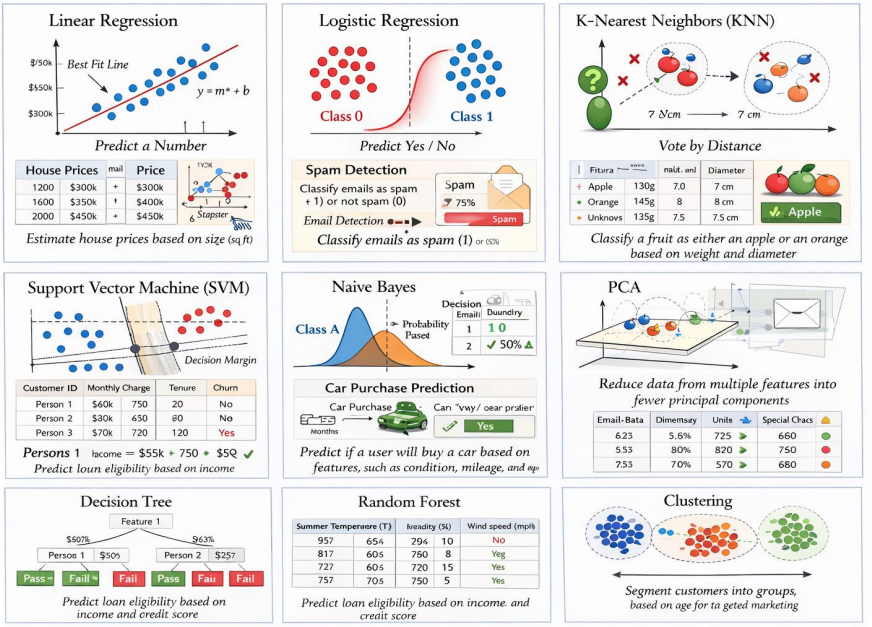
Q: What is overfitting?

A: Model memorizes training data and fails on new data. Fix: simpler model, pruning, more data, CV.

Q: Decision Tree vs Random Forest?

A: Tree is easy to explain but can overfit. Forest is more accurate and stable but less explainable.

All In One Machine Learning.



1) Linear Regression

You see dots and one straight red line through them.
That line is the "best guess trend."
For a new X value (like square feet), you go up to the line and read the predicted number (price).

2) Logistic Regression

You see two groups of dots (red vs blue) and an **S-shaped curve**.
The curve gives a **probability** from 0 to 1.
Then it decides Class 0 or Class 1 based on a cutoff (like **0.5**).

3) K-Nearest Neighbors (KNN)

You see a new point (?) and nearby points around it.
KNN checks the closest K points and asks: "Most of my neighbors are which class?"
Then it predicts the same class by majority vote.

4) Support Vector Machine (SVM)

You see two groups and a separating line with a wide gap (margin).
SVM tries to make this gap as wide as possible so separation is safer.
The few points touching the margin are the support vectors (they control the boundary).

5) Naive Bayes

You see two "bell curves" (Class A vs Class B) showing probability shapes.
For new input, it checks "Which class has higher probability for this value?"
Then it picks the class with bigger probability.

6) PCA (Reduce Dimensions)

You see data in a bigger space and then the same data shown in a smaller space.
PCA "rotates" and compresses data to keep the main spread (main information).
Result: fewer columns, simpler view, still keeps most patterns.

7) Decision Tree

You see a question flowchart: Feature 1 → Feature 2 → final result.
Each split is a simple question like "Yes/No."
Following the path gives the final decision (Pass/Fail).

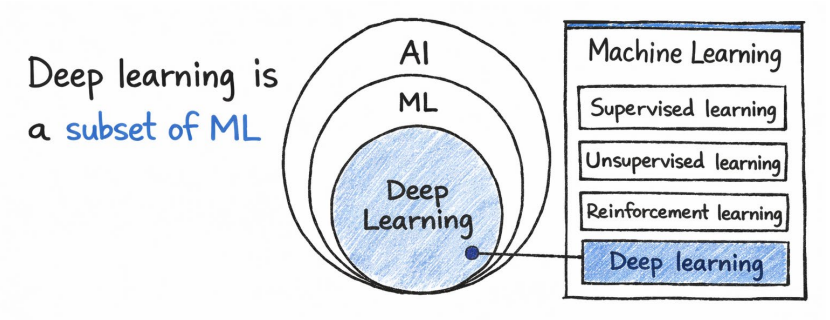
8) Random Forest

You see many small trees instead of one tree.
Each tree gives its own answer; then all answers are combined.
Final answer is majority voting (or average), so it becomes more stable.

9) Clustering

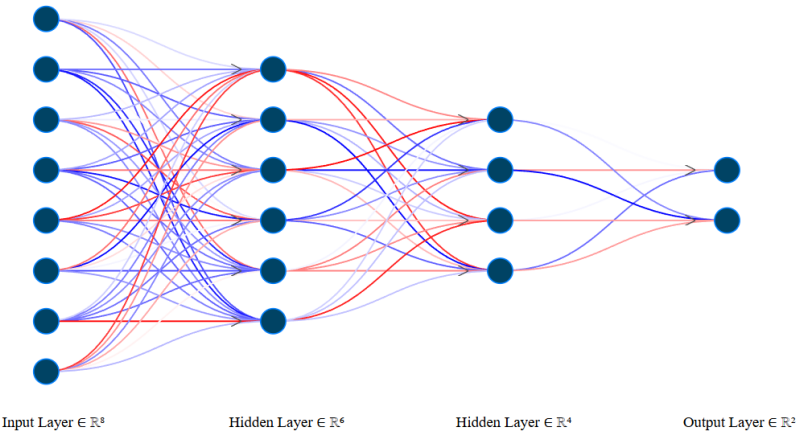
You see dots naturally forming separate groups (circles around them).
There is no "correct label" given — the algorithm just groups similar points.
It helps find customer groups like budget vs premium vs regular.

Part III — Deep Learning



This part covers ANN, CNN, RNN/LSTM, Transformers, reasoning models, and deployment. All examples use small datasets first, then show how to scale.

ANN



Preface
This Book is for Freshers

If you are new to deep learning, the biggest problem is not the math it is the teaching style. Many books jump directly to big datasets, big code, and heavy formulas. That makes beginners feel lost.

This book teaches deep learning like you learn driving: small steps, clear rules, and lots of practice. We will use tiny hard-coded datasets first (10–20 rows) so you can SEE what is happening. After you understand, scaling to large data becomes easy.

Promise: After Chapter 6, you will clearly understand: weights, bias, loss, gradient descent, epochs, batch size, and how a trained model predicts unseen data.

How to read each chapter:

- Definition (plain words)
- Syntax/Formula (only what you need)
- Why? → What? → How? → Let's Code!
- Common beginner mistakes
- Quick exercises

Table of Contents

0. Setup: Your Deep Learning Lab (local + Colab)
1. Before Learning Deep Learning, You Must Know This
2. Artificial Neural Networks (ANN)
    - Foundations of ANN
    - Deep Dive (Activations, Initialization, Regularization, Training Dynamics)
    - Worked Example (Full Forward + Backprop with Matrices)
    - The Smallest Neural Network (One Neuron)
3. Building Your First Neural Network
4. What Happens in Training? (weights, bias, loss, optimizer) step-by-step
5. Backpropagation and Gradient Descent (No Fear)
6. Your First Deep Learning Model in TensorFlow & Keras
7. How a Neural Network Learns Ex - House Prices (Weights, Biases, and Prediction).
8. Hyperparameters that control learning (epoch, batch size, LR, layers, neurons)
9. House Price Prediction with Deep Learning
10. ANN Regression Example (Real Estate price/rent) tiny dataset
11. ANN Classification Example (Banking default risk) tiny dataset
    - TensorFlow & Keras - How They Work Inside, and How You Build Real AI Models
12. CNN (Convolutional Neural Network) image learning with MNIST
13. RNN/LSTM learning from sequences tiny dataset
14. Transformers and Attention
15. Reasoning Models how to build and test them
16. Specialized & Emerging Neural Networks (Beyond ANN/CNN/RNN/Transformer)
17. Production Deep Learning: Packaging, APIs, monitoring
18. Deploy to AWS (Lambda container + API Gateway) full example
1. Appendix A: Math Cheat Sheet
2. Appendix B: Debugging and Common Errors
3. Appendix C: Glossary

Chapter 0
Setup: Your Deep Learning Lab

Definition

A deep learning lab is simply a place where you can run Python + TensorFlow code. You can use Google Colab (easy) or your laptop (more control).

Why?

Because you should be able to run every code example in this book without struggling for days.

What?

We will use: Python 3.10+ • TensorFlow • NumPy • Matplotlib • (optional) Flask for deployment.

How?

Option A: Google Colab (recommended for freshers)

Colab steps

```
1) Open Google Colab
2) Runtime → Change runtime type → GPU (optional)
3) Run in a cell:
   !pip install tensorflow
```

Option B: Local laptop (Windows/Mac/Linux)

Local setup commands

```
python -m venv .venv
# Windows: .venv\Scripts\activate
# Mac/Linux: source .venv/bin/activate
pip install --upgrade pip
pip install tensorflow numpy matplotlib flask


Beginner tip: If TensorFlow installation fails on your laptop,
use Google Colab first. Don’t get stuck in setup.
```

Chapter 1
Before Learning Deep Learning, You Must Know This

Definition

Before you start building Deep Learning models using TensorFlow and Keras, you must understand a small "starter kit" of Math + Statistics + Implementation concepts.

This kit is not advanced mathematics. It is the minimum required knowledge that helps you understand:

- why a neural network uses $XW + b$
- why training needs loss
- why TensorFlow talks about tensors and shapes
- why scaling is mandatory
- why models overfit or underfit
- what compile(), fit(), and predict() truly mean

Why?

Many beginners start Deep Learning like this:
Copy code → Run it → Get accuracy
Still confused

Later they face problems like:

- Loss is NaN
- My model is not learning
- Accuracy is stuck
- Input shape mismatch
- Overfitting is happening

The reason is simple:

Deep Learning is not only coding.
Deep Learning is **math + stats + training loop**, and
TensorFlow/Keras implements it.

What?

Deep Learning training is one repeated cycle:

Predict → Measure error → Improve weights → Repeat

TensorFlow/Keras automates this cycle, but you must know what is happening inside.

To understand this cycle, you must know these 7 things:

1. Data Shape Mindset (tensors + shapes)
2. Core equation $XW + b$
3. Stats for scaling (mean, std, normalization/standardization)
4. Loss functions (MSE/BCE/CCE)
5. Gradients + learning rate
6. Underfitting vs Overfitting
7. Keras training pipeline (compile, fit, predict)

How?

Let's learn each item in the simplest, implementation-first way.

Data Shape Mindset (Because Deep Learning Thinks in Shapes)

Before anything else, learn this:

Data is always a tensor

- Scalar → one number
- Vector → list of numbers
- Matrix → table of numbers
- Tensor → multi-dimensional data

In real deep learning:

- tabular data → (rows, features)
- images → (batch, height, width, channels)
- text → (batch, sequence_length)

Why this matters

Most beginner errors happen because of wrong shape:

- "Input shape mismatch"
- "Expected 4D tensor but got 2D"
- "Cannot broadcast shapes"

1.1 What it means

TensorFlow does not understand *house*, *image*, or *text*. It understands numbers arranged in shapes (tensors).

If the shape is correct:
the model runs and produces results

If the shape is wrong:
model fails or learns incorrectly

So shapes are like grammar. If grammar is wrong, sentence breaks.

1.2 The two most common shapes

A) Tabular data (Excel-like)

(batch, features)

Example:

1 house record with 3 features → (1, 3)

4 house records with 3 features → (4, 3)

B) Image data (CNN input)

(batch, height, width, channels)

Example:

1 grayscale image 28×28 → (1, 28, 28, 1)

1.3 Example A (Hardcoded): Tabular shape producing output

Raw house record (human format)

- size = 1000
- bedrooms = 2
- age = 10
- $X = [1000,2,10]$ But Keras expects batch format, so $(1, 3)$

```
import numpy as np
import tensorflow as tf

# 1) One record (NOT batch format)
x_one = np.array([1000, 2, 10], dtype=np.float32)
print("x_one shape:", x_one.shape)  # (3,)

# 2) Convert to batch format (1 row, 3 features)
x_batch = x_one.reshape(1, 3)
print("x_batch shape:", x_batch.shape)  # (1,3)

# 3) Simple model expects 3 features
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(3,)),
    tf.keras.layers.Dense(4, activation="relu"),
    tf.keras.layers.Dense(1)
])

# Predict to prove shape works
pred = model.predict(x_batch, verbose=0)
print("Prediction shape:", pred.shape)  # (1,1)
print("Prediction value:", float(pred[0][0]))
```

Result

```
x_one shape: (3,)
x_batch shape: (1, 3)
Prediction shape: (1, 1)
Prediction value: -345.6344299316406
```

What the reader learns

- deep learning expects batch inputs
- (1,3) produces (1,1)
- shape directly controls how output is generated

Example B (Hardcoded): Image shape producing probability output

We create a tiny **2×2 image** using hardcoded pixels:

$$\begin{bmatrix} 0 & 255 \\ 128 & 64 \end{bmatrix}$$

CNN expects:

- add channel → (2,2,1)
- add batch → (1,2,2,1)

```
import numpy as np
import tensorflow as tf

# 1) Hardcoded 2x2 grayscale image
img = np.array([
    [0,   255],
    [128, 64]
], dtype=np.float32)

print("Original shape:", img.shape)  # (2,2)

# 2) Normalize pixels 0-1 (stats)
img = img / 255.0

# 3) Add channel and batch dimensions
img = img.reshape(2, 2, 1)      # (2,2,1)
img = img.reshape(1, 2, 2, 1)   # (1,2,2,1)
print("Final shape:", img.shape)
```

```
# 4) A tiny CNN
cnn = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(2,2,1)),
    tf.keras.layers.Conv2D(2, (2,2), activation="relu"),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(3, activation="softmax")
])

out = cnn.predict(img, verbose=0)

print("Output shape:", out.shape)              # (1,3)
print("Probabilities:", out)
print("Sum of probs:", float(out.sum()))       # ~1.0
```

Result

Original shape: (2, 2)

Final shape: (1, 2, 2, 1)

Output shape: (1, 3)

Probabilities: [[0.24674958 0.2774463 0.47580403]]

Sum of probs: 0.9999999403953552

Mini Summary: Data Shape Mindset

Deep learning expects:

- Tabular → (batch, features)
- Image → (batch, height, width, channels)

Shape is not optional. Shape is the rule.

2) The Core Equation You Must Know: $XW + b$

2.1 What it means

Every Dense layer is doing:

$$Z = XW + b$$

then

$$A = f(Z)$$

- X= input
- W= weights (learned importance)
- b= bias (fine adjustment)
- f= activation (ReLU/sigmoid/softmax)

Simple meaning:

Multiply inputs by weights, add bias, then apply a rule.

The Stats You Must Know (Data Preparation is Half of DL)

Why statistics is required

Deep learning learns using gradients.
If feature scales are very different, gradients become unbalanced and learning becomes poor.

Example:

- size = 2000 (big number)
- bedrooms = 3 (small number)

Model pays too much attention to size because it's numerically huge.
So we scale.

Two scaling methods (must know)

A) Normalization (0–1)

Used for images:

$$x' = \frac{x}{255}$$

B) Standardization (mean 0, std 1)

Used for tabular:

$$x' = \frac{x - \mu}{\sigma}$$

Where:

$$\mu = \frac{1}{n}\sum x$$

$$\sigma = \sqrt{\frac{1}{n}\sum (x-\mu)^2}$$

Hardcoded demo: Training with vs without scaling (result difference)

```
import numpy as np
import tensorflow as tf

# Hardcoded tabular dataset
X = np.array([
    [1000, 2, 10],
    [ 850, 2, 12],
    [1200, 3,  8],
    [ 700, 2, 20],
    [1500, 3,  5],
    [2000, 4,  2]
], dtype=np.float32)

y = np.array([[75],[60],[95],[40],[140],[220]], dtype=np.float32)

def build_model():
    m = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(3,)),
        tf.keras.layers.Dense(8, activation="relu"),
        tf.keras.layers.Dense(1)
    ])
    m.compile(optimizer=tf.keras.optimizers.Adam(0.01), loss="mse")
    return m

# 1) Train WITHOUT scaling
m1 = build_model()
h1 = m1.fit(X, y, epochs=200, verbose=0)
```

```
loss_raw = h1.history["loss"][-1]

# 2) Train WITH standardization
mean = X.mean(axis=0)
std  = X.std(axis=0) + 1e-7
X_scaled = (X - mean) / std

m2 = build_model()
h2 = m2.fit(X_scaled, y, epochs=200, verbose=0)
loss_scaled = h2.history["loss"][-1]

print("Final Loss WITHOUT scaling:", round(float(loss_raw), 2))
print("Final Loss WITH scaling   :", round(float(loss_scaled), 2))
```

Result - Final Loss WITHOUT scaling: 544.02 - Final Loss WITH scaling : 3188.44

Loss Function (How the model knows it is wrong)

Loss is a single number telling:

"How wrong was the prediction?"

Regression (number output)

MSE

$$MSE = \frac{1}{n}\sum(y - \hat{y})^2$$

Binary classification (yes/no)

BCE

$$BCE = -\sum[y\log(\hat{y}) + (1 - y)\log(1 - \hat{y})]$$

Multi-class classification (0–9, cat/dog)

CCE

$$CCE = -\sum y\log(\hat{y})$$

If you choose wrong loss, training becomes meaningless.

Gradients and Learning Rate (The "learning" part)

Gradient meaning

Gradient tells:

"If I change this weight slightly, will loss go down?"

TensorFlow calculates gradients automatically (autodiff).

Update rule

$$W := W - \alpha \frac{\partial Loss}{\partial W}$$

Where:

- α= learning rate (step size)

Learning rate rule

- too high → unstable training, loss becomes NaN
- too low → training slow

Safe default:

- Adam + learning_rate=0.001

Underfitting vs Overfitting (How to know model is learning correctly)

Underfitting (model too weak)

- train loss high
- val loss high

Fix:

- add neurons/layers
- train longer
- better features

Overfitting (model memorizes)

- train loss low

- val loss increases
-

Fix:

- dropout
- early stopping
- reduce model size
- more data

This is the most practical DL concept.

The Keras Training Pipeline (So you understand implementation)

model.compile(...)

Means: set learning rules:

- loss (what to reduce)
- optimizer (how to reduce)
- metrics (how to report)

model.fit(...)

Means: training loop starts:

- forward pass → prediction
- loss calculation
- gradient calculation
- weight update
- repeat for epochs

model.predict(...)

Means: only forward pass

- no training
- weights stay unchanged

Let's Code! (Tiny Demo Connecting Everything)

```
import numpy as np
import tensorflow as tf

# Raw data: [size, bedrooms]
X = np.array([
    [1000, 2],
    [ 800, 2],
    [1200, 3],
    [1500, 3]
], dtype=np.float32)

y = np.array([[75],[60],[95],[140]], dtype=np.float32)

# --- Stats: standardize ---
mean = X.mean(axis=0)
std  = X.std(axis=0) + 1e-7
X_s = (X - mean) / std

# --- Model: XW + b happens inside Dense ---
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(2,)),
    tf.keras.layers.Dense(8, activation="relu"),
    tf.keras.layers.Dense(1)
])

# --- Compile: choose optimizer + loss ---
model.compile(optimizer=tf.keras.optimizers.Adam(0.01), loss="mse")

# --- Fit: training loop (predict → loss → update) ---
model.fit(X_s, y, epochs=200, verbose=0)

# --- Predict: uses learned weights ---
new_house = np.array([[1100, 3]], dtype=np.float32)
```

```
new_house_s = (new_house - mean) / std


pred = model.predict(new_house_s, verbose=0)
print("Predicted price:", float(pred[0][0]))
```

Exercises (For Readers)

- Change learning rate to 0.1 and observe if training becomes unstable
- Remove scaling and observe prediction changes
- Add one more Dense layer and see if model becomes better or overfits

Cheat Sheet

Before Deep Learning, you must know:

- **Tensors + shapes** (data format)
- **Core equation:** $XW + b$
- **Scaling:** mean, std, normalization/standardization
- **Loss:** MSE/BCE/CCE
- **Gradients + learning rate**
- **Underfit vs Overfit**
- **Keras pipeline:** compile → fit → evaluate/predict

Chapter 2
Artificial Neural Networks (ANN)

Definition

An Artificial Neural Network (ANN) is a model inspired by the brain, but built with simple math. It is a set of layers of 'neurons' (small calculators) that turn input numbers into an output.

In simple words: ANN learns a flexible rule from examples, even when the relationship is not a straight line.

Syntax / Formula (Math in simple words)

Layer calculation: $z = XW + b$, then $a = \text{activation}(z)$.

X = inputs, W = weights (what the model learns), b = bias (extra push), activation adds non-linearity.

Why?

Because many real-world problems are not linear. ANN can learn curves and complex patterns (pricing, demand, risk, image/text patterns).

What?

In this chapter you will build intuition for: neuron, weights and bias, activations, layers, loss, and training.

How?

We start from ONE neuron (smallest network), then grow to multiple layers. Every time, we connect the math to a tiny dataset so you can see what is happening.

Let's Code!

For the first hands-on code, jump to Chapter 1.1 (One Neuron). After that, come back here and read the bigger ANN explanation.

Part 1
Foundations of ANN

An Artificial Neural Network (ANN) is a model that learns a mapping from inputs (independent variables / features) to output (dependent variable / target) by repeatedly making predictions, measuring error, and correcting its weights and biases.

1.1 Why ANN?

Many real-world relationships are not simple straight lines. ANN adds non-linearity using activation functions, so it can learn more complex patterns than a plain linear model.

1.2 Core building block: a neuron

A neuron performs two steps:

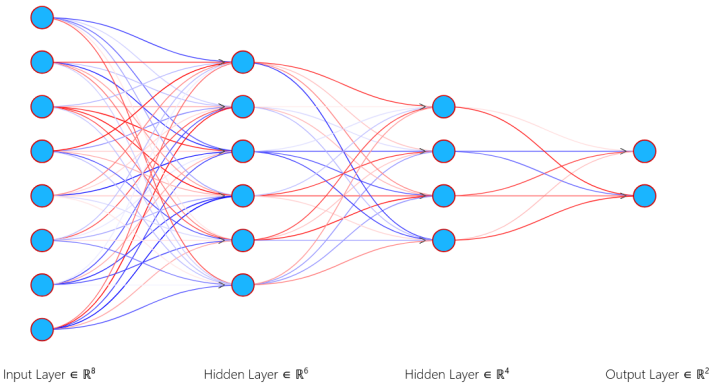
- Weighted sum + bias
- Activation function

Mathematically:

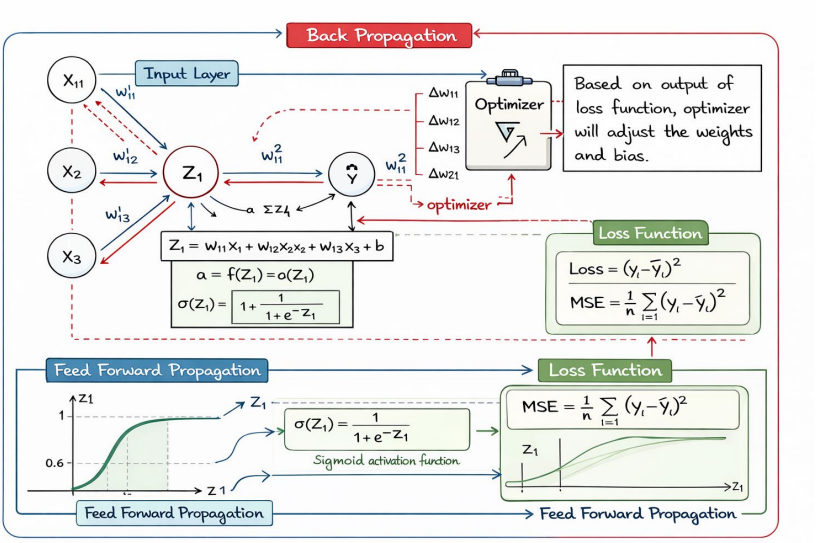
```
z = (w1*x1 + w2*x2 + ... + wn*xn) + b
a = f(z)
```

1.3 ANN architecture

A typical ANN has: Input layer → Hidden layer(s) → Output layer.



In this chapter we use a simple network with 3 inputs, one hidden layer, and 1 output:



1.4 Dense (Fully Connected) layer

A Dense (Fully Connected) layer means every neuron in the previous layer connects to every neuron in the next layer. That is why it is also called a fully connected layer.

1.5 Parameters: weights and biases

Trainable parameters are the numbers the model learns: weights (W) and biases (b). For a Dense layer with n_in inputs and n_out neurons:

```
Parameters = (n_in * n_out) + (n_out)
```

For our network $3 \longrightarrow 4 \longrightarrow 1$:

- `Layer 1 (3 → 4): (3*4)+4 = 16 parameters`
- `Layer 2 (4 → 1): (4*1)+1 = 5 parameters`

Total trainable parameters = 21

1.6 Forward pass (feed-forward)

Forward pass means: input flows layer by layer to produce prediction (y-hat).

```
Z1 = X·W1 + b1
A1 = ReLU(Z1)
Z2 = A1·W2 + b2
ŷ  = Z2  (linear output for regression)
```

1.7 Loss function (how wrong the model is)

For regression, a common choice is Mean Squared Error (MSE):

```
L = mean((ŷ - y)^2)
```

1.8 Backpropagation (backward pass)

Backpropagation computes gradients (partial derivatives) that tell how much each parameter contributed to the error. Gradients are used to update weights and biases to reduce the loss.

1.9 Optimizer and weight update

An optimizer updates parameters using gradients. The simplest is (mini-batch) Stochastic Gradient Descent (SGD):

```
W := W - learning_rate * dL/dW
b := b - learning_rate * dL/db
```

1.10 Training evidence (loss and predictions)

Below are three plots from a small training run:
1) Loss curve (should go down)
2) True vs Predicted (should align)
3) One weight value over epochs (shows parameters adjusting)

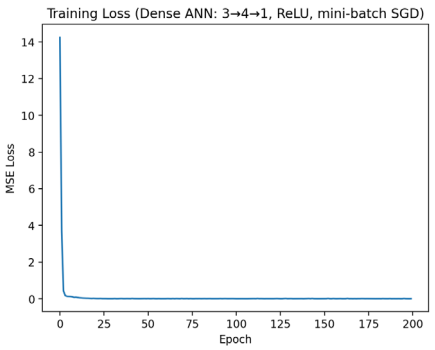


Figure 2. Training loss curve (MSE vs epoch).

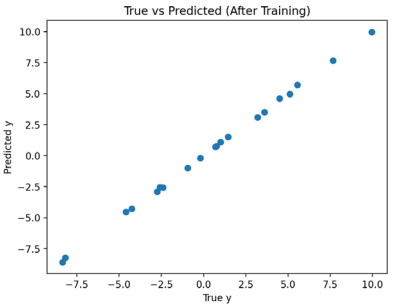


Figure 3. True vs Predicted after training

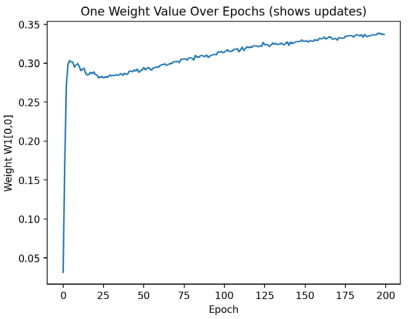


Figure 4. One weight changing over training (W1[0,0]).

Part 2
Deep Dive (Activations, Initialization, Regularization, Training Dynamics)

2.1 Activation functions with real numbers

Activation functions convert the linear pre-activation z into a non-linear activation a. This non-linearity is what gives neural networks their power.

Example: one neuron, same z, different activations

```
Let x = [2, -1, 0.5], w = [0.3, 0.8, -0.2], b = 0.1.
z = 2*0.3 + (-1)*0.8 + 0.5*(-0.2) + 0.1 = -0.2
```

Now compare activations:

```
• ReLU(z) = max(0, z) → ReLU(-0.2) = 0 (neuron turns off)
• Sigmoid(z) = 1/(1+e^-z) → Sigmoid(-0.2) ≈ 0.450
• Tanh(z) → tanh(-0.2) ≈ -0.197
```

2.2 Common activations and typical use

- ReLU / Leaky ReLU: most common in hidden layers
- Sigmoid: binary classification output (probability in 0 to 1)
- Softmax: multi-class classification output (probabilities sum to 1)
- Linear: regression output

An **activation function** is like a **switch + shape maker** inside a neural network.

- The network first calculates a value: weighted sum = (inputs × weights) + bias
- Then the activation function decides:
  - Should this neuron "fire" or stay quiet?
  - How strongly should it pass the signal forward?

Without activation functions, a deep network would behave like a **simple linear equation** and would not learn complex patterns.

1) Sigmoid

What it does: Converts a number into a value between **0 and 1**.

When to use: Mostly in the **output layer** for **binary classification** (Yes/No, Spam/Not spam)

Why use it: Output looks like a **probability** (example: 0.85 = 85% chance)

Limitation: In deep networks, learning becomes slow due to the **vanishing gradient** problem.

2) Tanh (Hyperbolic Tangent)

What it does: Converts a number into a value between **−1 and 1**.

When to use: Sometimes in **hidden layers** when you want values centered around **0**

Why use it: Helps training because outputs are not only positive; they are balanced around 0.

Limitation: Still suffers from **vanishing gradients** in deep networks.

3) ReLU (Rectified Linear Unit)

What it does: If input is **positive → keep it**,If input is **negative → make it 0**

When to use: Default choice for most **hidden layers** in deep learning.

Why use it: Trains faster and reduces vanishing gradient issues. & Simple and works well in practice.

Limitation: Dying ReLU: some neurons can get stuck outputting only 0 and stop learning.

4) Leaky ReLU

What it does: Like ReLU, but if input is negative, it outputs a **small negative value** instead of 0.

When to use: When you see the **dying ReLU** problem.

Why use it: Keeps neurons "alive" even for negative inputs, so learning continues.

Limitation: Depends on choosing the small slope value (like 0.01).

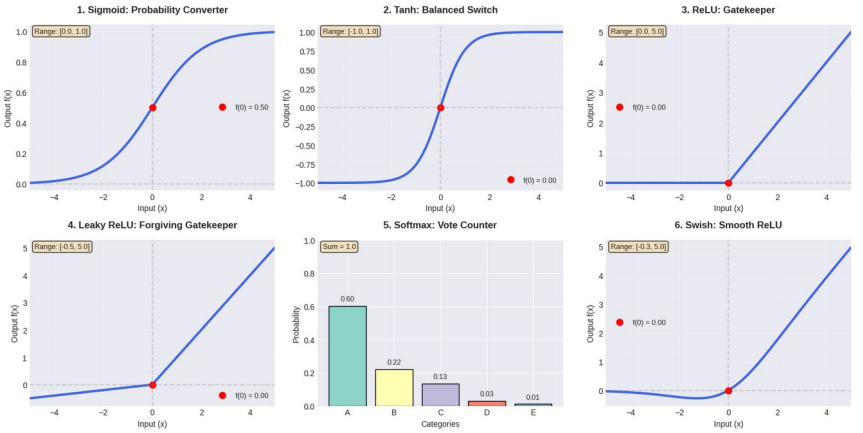
5) SoftMax

What it does: Converts outputs into probabilities for **multiple classes**, and the total becomes **1**.

When to use: Output layer for **multi-class classification** (Cat/Dog/Bird, 0–9 digits, etc.)

Why use it: Gives clear probability for each class (example: Cat 0.1, Dog 0.7, Bird 0.2)

Limitation: Used only in the **output layer**, not hidden layers.



Think of activation functions as DECISION MAKERS in neural networks.

They decide: 'How strongly should this neuron respond?'

Let's see what happens with different numbers:

Input numbers: [-3, -1, 0, 1, 3]

For input = -3:

Sigmoid: 0.047 (probability between 0 and 1)
Tanh: -0.995 (between -1 and 1)
ReLU: 0.0 (negative → 0, positive → same)
Leaky ReLU: -0.300 (negative → small, positive → same)

For input = -1:

Sigmoid: 0.269 (probability between 0 and 1)
Tanh: -0.762 (between -1 and 1)
ReLU: 0.0 (negative → 0, positive → same)
Leaky ReLU: -0.100 (negative → small, positive → same)

For input = 0:

Sigmoid: 0.500 (probability between 0 and 1)
Tanh: 0.000 (between -1 and 1)

ReLU: 0.0 (negative → 0, positive → same)
Leaky ReLU: 0.000 (negative → small, positive → same)

For input = 1:

Sigmoid: 0.731 (probability between 0 and 1)
Tanh: 0.762 (between -1 and 1)
ReLU: 1.0 (negative → 0, positive → same)
Leaky ReLU: 1.000 (negative → small, positive → same)

For input = 3:

Sigmoid: 0.953 (probability between 0 and 1)
Tanh: 0.995 (between -1 and 1)
ReLU: 3.0 (negative → 0, positive → same)
Leaky ReLU: 3.000 (negative → small, positive → same)

2.3 Fan-in, fan-out and weight initialization

Bad initialization can cause exploding or vanishing activations/gradients. Fan-in is the number of inputs into a neuron; fan-out is the number of outputs from a neuron.

He initialization (recommended for ReLU):

```
W ~ Normal(0, sqrt(2/fan_in))
```

Why not initialize all weights to zero? Because then all neurons learn the same thing (symmetry problem), so the hidden layer becomes useless.

2.4 Regularization: L2 and Dropout

Regularization reduces overfitting (memorizing training data).

L2 (weight decay)

```
L_total = L_data + λ * sum(W^2)
```

Dropout

Dropout randomly sets some activations to zero during training. This forces the network to learn robust features. Dropout is OFF during inference.

2.5 Batch size, epochs, learning rate: what changes during training

Definitions:

- Epoch: one full pass over the training dataset
- Batch size: number of rows used to compute one parameter update
- Steps per epoch = N / batch_size

Training story:

Epoch 1: weights random → predictions poor → loss high → gradients large.
Epoch 2+: weights improve → loss decreases → gradients smaller → fine tuning.
Later epochs: very small updates → model stabilizes near a good solution.

Learning rate controls update size:

```
W := W - η * dL/dW
```

- η too high → training may diverge (loss increases)
- η too low → training becomes very slow

Epoch: One complete presentation of the entire training dataset to the learning algorithm multiple epochs are often required to adequately train a model

Batch size: The number of training examples utilized in one iteration of model training

Dropout: A regularization technique that involves randomly setting a fraction of input units to 0 at each update during training time, which helps to prevent overfitting

Transfer learning: Leveraging a pretrained model on a new, related problem popular in deep learning where large datasets are required to train a model from scratch

Part 3
Worked Example (Full Forward + Backprop with Matrices)

In this part, we do a complete numeric example with matrices so you can see exactly what happens in forward pass, loss calculation, backprop gradients, and one update step.

3.1 Network definition (3 → 2 → 1)

We use a tiny network: 3 inputs → 2 hidden neurons (ReLU) → 1 output neuron (Linear).

```
# ====================
# 1. INITIALIZATION
# ====================
import math

# Input and target
X = [1.0, 2.0, -1.0]  # Shape: (3,) - single sample
y = 2.0               # Target value

# Initialize parameters
W1 = [[0.10, -0.20],  # Shape: (3, 2)
      [0.00,  0.30],
      [-0.10, 0.20]]

b1 = [0.05, -0.05]    # Shape: (2,)

W2 = [[0.40],         # Shape: (2, 1)
      [-0.10]]

b2 = [0.20]           # Shape: (1,)

learning_rate = 0.05
```

```
print("=" * 60)
print("NEURAL NETWORK FORWARD AND BACKWARD PASS")
print("=" * 60)
print(f"Input X: {X}")
print(f"Target y: {y}")
print()

# ====================
# 2. FORWARD PASS
# ====================
print("=" * 60)
print("FORWARD PASS")
print("=" * 60)

# 2.1 Hidden layer pre-activation: Z1 = X·W1 + b1
print("\n2.1 Hidden layer pre-activation (Z1 = X·W1 + b1):")

# Manual calculation for clarity:
# Z1_1 = 1*0.10 + 2*0.00 + (-1)*(-0.10) + 0.05
Z1_1 = X[0]*W1[0][0] + X[1]*W1[1][0] + X[2]*W1[2][0] + b1[0]
print(f"  Z1_1 = {X[0]}*{W1[0][0]} + {X[1]}*{W1[1][0]} +
{X[2]}*{W1[2][0]} + {b1[0]} = {Z1_1}")

# Z1_2 = 1*(-0.20) + 2*0.30 + (-1)*0.20 + (-0.05)
Z1_2 = X[0]*W1[0][1] + X[1]*W1[1][1] + X[2]*W1[2][1] + b1[1]
print(f"  Z1_2 = {X[0]}*{W1[0][1]} + {X[1]}*{W1[1][1]} +
{X[2]}*{W1[2][1]} + {b1[1]} = {Z1_2}")



Z1 = [Z1_1, Z1_2]
print(f"  Z1 = {Z1}")

# 2.2 Hidden layer activation (ReLU): A1 = max(0, Z1)
print("\n2.2 Hidden layer activation (ReLU):")
def relu(x):
```

```
    return max(0, x)


A1 = [relu(z) for z in Z1]
print(f"  ReLU({Z1}) = {A1}")
print(f"  A1 = {A1}")


# 2.3 Output layer: ŷ = A1·W2 + b2
print("\n2.3 Output layer (ŷ = A1·W2 + b2):")
y_hat = A1[0]*W2[0][0] + A1[1]*W2[1][0] + b2[0]
print(f"  ŷ = {A1[0]}*{W2[0][0]} + {A1[1]}*{W2[1][0]} + {b2[0]} =
{y_hat}")


# 2.4 Loss calculation (MSE for single sample)
print("\n2.4 Loss calculation (L = (ŷ - y)²):")
L = (y_hat - y) ** 2
print(f"  L = ({y_hat} - {y})² = {L}")


# ====================
# 3. BACKPROPAGATION
# ====================
print("\n" + "=" * 60)
print("BACKPROPAGATION (GRADIENTS)")
print("=" * 60)




# 3.1 Gradient of loss with respect to output
print("\n3.1 Gradient of loss w.r.t output (dL/dŷ):")
dL_dy_hat = 2 * (y_hat - y)
print(f"  dL/dŷ = 2 * ({y_hat} - {y}) = {dL_dy_hat}")

# 3.2 Output layer gradients
print("\n3.2 Output layer gradients:")
print("  Since ŷ = A1·W2 + b2:")
```

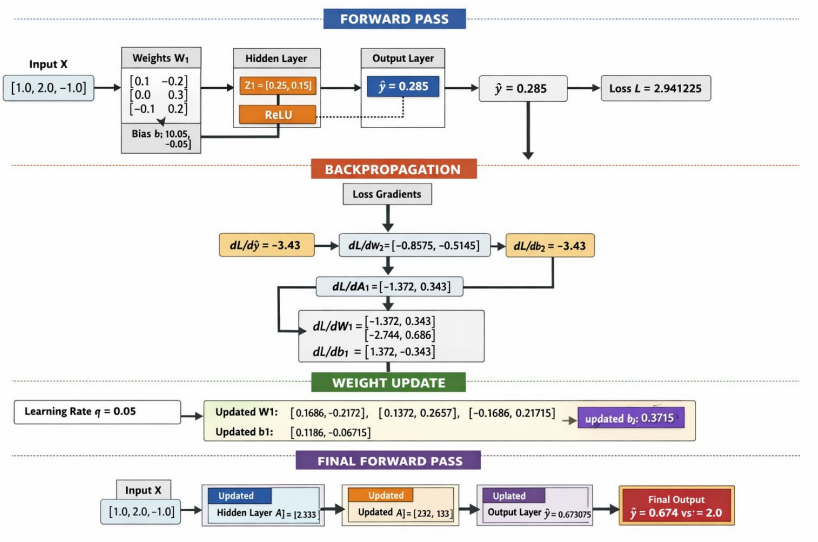
```
# dL/dW2 = A1^T * dL/dŷ
print("  dL/dW2 = A1^T * dL/dŷ:")
dL_dW2 = [[A1[0] * dL_dy_hat], [A1[1] * dL_dy_hat]]
print(f"    = [[{A1[0]} * {dL_dy_hat}], [{A1[1]} * {dL_dy_hat}]]")
print(f"    = [[{dL_dW2[0][0]}], [{dL_dW2[1][0]}]]")

# dL/db2 = dL/dŷ
print("  dL/db2 = dL/dŷ:")
dL_db2 = dL_dy_hat
print(f"    = {dL_db2}")

# 3.3 Backpropagate to hidden layer
print("\n3.3 Backpropagate to hidden layer:")
print("  dL/dA1 = dL/dŷ * W2^T:")

# W2^T = [[0.40, -0.10]] (shape 1x2)
dL_dA1 = [dL_dy_hat * W2[0][0], dL_dy_hat * W2[1][0]]
print(f"    = {dL_dy_hat} * [{W2[0][0]}, {W2[1][0]}]")
print(f"    = [{dL_dA1[0]}, {dL_dA1[1]}]")




# 3.4 ReLU gradient
print("\n3.4 ReLU gradient:")
print("  ReLU'(z) = 1 if z > 0, else 0")
print(f"  Z1 = {Z1} (both > 0, so ReLU' = 1 for both)")

dL_dZ1 = dL_dA1  # Element-wise multiplication with ReLU' (which is
1)
print(f"  dL/dZ1 = dL/dA1 ⊙ ReLU'(Z1) = {dL_dA1} ⊙ [1, 1] =
{dL_dZ1}")

# 3.5 Input layer gradients
print("\n3.5 Input layer gradients:")
print("  dL/dW1 = X^T * dL/dZ1:")
```

```
# X^T = [[1.0], [2.0], [-1.0]] (shape 3x1)
# dL/dZ1 = [-1.372, 0.343] (shape 1x2)
# Result should be (3x2)
dL_dW1 = [[0.0, 0.0], [0.0, 0.0], [0.0, 0.0]]
for i in range(3):  # rows (input features)
    for j in range(2):  # columns (hidden neurons)
        dL_dW1[i][j] = X[i] * dL_dZ1[j]

print(f"    X^T = [[{X[0]}], [{X[1]}], [{X[2]}]]")
print(f"    dL/dZ1 = [{dL_dZ1[0]}, {dL_dZ1[1]}]")
print(f"    dL/dW1 = [[{dL_dW1[0][0]:.3f}, {dL_dW1[0][1]:.3f}],")
print(f"              [{dL_dW1[1][0]:.3f}, {dL_dW1[1][1]:.3f}],")
print(f"              [{dL_dW1[2][0]:.3f}, {dL_dW1[2][1]:.3f}]]")

# dL/db1 = dL/dZ1
print("  dL/db1 = dL/dZ1:")
dL_db1 = dL_dZ1
print(f"    = {dL_db1}")

# ====================
# 4. PARAMETER UPDATE
# ====================
print("\n" + "=" * 60)
print("PARAMETER UPDATE (Gradient Descent)")
print("=" * 60)
print(f"Learning rate η = {learning_rate}")
print("Update rule: param_new = param_old - η * gradient")

# 4.1 Update W2 and b2
print("\n4.1 Update W2 and b2:")

W2_new = [[W2[0][0] - learning_rate * dL_dW2[0][0]],
          [W2[1][0] - learning_rate * dL_dW2[1][0]]]
```

```
print(f"  W2_new = [[{W2[0][0]}] - {learning_rate} *
[{dL_dW2[0][0]}]]")
print(f"           [[{W2[1][0]}]   {learning_rate} *
[{dL_dW2[1][0]}]]")
print(f"         = [[{W2_new[0][0]:.6f}]")
print(f"            [{W2_new[1][0]:.6f}]]")


b2_new = [b2[0] - learning_rate * dL_db2]
print(f"  b2_new = {b2[0]} - {learning_rate} * {dL_db2} =
{b2_new[0]:.6f}")


# 4.2 Update W1 and b1
print("\n4.2 Update W1 and b1:")


W1_new = [[0.0, 0.0], [0.0, 0.0], [0.0, 0.0]]
for i in range(3):
    for j in range(2):
        W1_new[i][j] = W1[i][j] - learning_rate * dL_dW1[i][j]


print("  W1_new = W1 - 0.05 * dL/dW1")
print(f"         = [[{W1_new[0][0]:.6f}, {W1_new[0][1]:.6f}],")
print(f"            [{W1_new[1][0]:.6f}, {W1_new[1][1]:.6f}],")
print(f"            [{W1_new[2][0]:.6f}, {W1_new[2][1]:.6f}]]")


b1_new = [b1[0] - learning_rate * dL_db1[0],
          b1[1] - learning_rate * dL_db1[1]]
print(f"  b1_new = [{b1[0]}, {b1[1]}] - 0.05 * [{dL_db1[0]},
{dL_db1[1]}]")
print(f"         = [{b1_new[0]:.6f}, {b1_new[1]:.6f}]")


# ====================
# 5. SUMMARY
# ====================
print("\n" + "=" * 60)
print("SUMMARY")
print("=" * 60)
```

```
print(f"Initial loss: {L:.6f}")
print(f"Final predictions vs target: {y_hat:.3f} vs {y}")

print("\nParameter updates completed:")
print(f"  W1 updated: {len(W1)}x{len(W1[0])} matrix")
print(f"  b1 updated: {len(b1)}-dimensional vector")
print(f"  W2 updated: {len(W2)}x{len(W2[0])} matrix")
print(f"  b2 updated: {len(b2)}-dimensional vector")

# Verify one key gradient calculation
print("\n" + "=" * 60)
print("VERIFICATION")
print("=" * 60)
print("Key gradient dL/dW2:")
print(f"  Calculated: [[{dL_dW2[0][0]:.6f}], [{dL_dW2[1][0]:.6f}]]")
print(f"  Expected:   [[-0.8575], [-0.5145]]")

difference = abs(dL_dW2[0][0] - (-0.8575)) + abs(dL_dW2[1][0] - (-
0.5145))
print(f"  Total difference: {difference:.10f}")

if difference < 0.0001:
    print("Gradients match expected values within tolerance")
else:
    print("Gradients differ from expected values")
```

Output



3.5 Key takeaways from Part 3

- Forward pass is matrix multiplication + bias + activation.
- Backprop is chain rule applied layer-by-layer.
- Gradient Descent/SGD uses gradients to reduce loss.
- Inference uses only the forward pass (no backprop).

Chapter 1.1
The Smallest Neural Network (One Neuron)

Definition

A neural network is just a machine that takes numbers as input and produces numbers as output. The smallest neural network is ONE neuron. If you understand one neuron, you understand the heart of deep learning.

Syntax / Formula

Key formula (small and important)

```
Neuron output:
  z = w*x + b
  y = activation(z)

Where:
  x = input number
  w = weight (how strongly we listen to x)
  b = bias (a free extra push)
  activation = a function like ReLU or sigmoid
```

Why?

Because many people fear deep learning. But one neuron is as simple as a straight line with a small twist (activation).

What?

We will predict a student's final score using only ONE input: hours studied. This is not a perfect real-world model, but it is the best way to learn the idea.

How?

Step 1: Start with random weight w and bias b.
Step 2: Compute prediction y for each student.
Step 3: Compare prediction with the real score → this gives error.

Step 4: Change w and b slightly to reduce error.
That repeated improvement is called training.

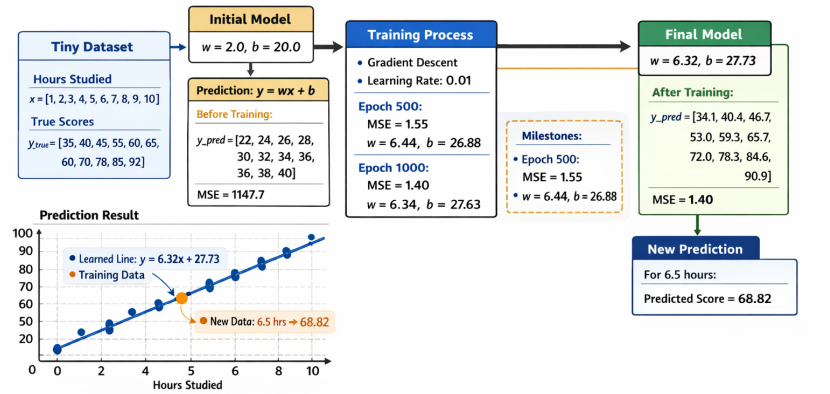
Let's Code!

```
import numpy as np

# Tiny hard-coded dataset (10 students)
# x = hours studied, y_true = exam score out of 100
x = np.array([1,2,3,4,5,6,7,8,9,10], dtype=float)
y_true = np.array([35,40,45,55,60,65,70,78,85,92], dtype=float)

# Start with random weight and bias
w = 2.0
b = 20.0

def predict(x):
    return w*x + b   # one neuron without activation (linear)

# Let's see predictions before training
y_pred = predict(x)
print("Predictions before training:", np.round(y_pred, 1))

# Loss = Mean Squared Error (MSE)
def mse(y_true, y_pred):
    return np.mean((y_true - y_pred)**2)

print("MSE before training:", mse(y_true, y_pred))

# --- A super-simple training loop (manual gradient descent) ---
lr = 0.01
epochs = 2000

for epoch in range(epochs):
    y_pred = predict(x)
    # Gradients for MSE with respect to w and b (simple math)
    # d/dw MSE = (-2/N) * sum( x * (y_true - y_pred) )
    # d/db MSE = (-2/N) * sum( (y_true - y_pred) )
    N = len(x)
    dw = (-2.0/N) * np.sum(x * (y_true - y_pred))
    db = (-2.0/N) * np.sum(y_true - y_pred)

    w = w - lr*dw
    b = b - lr*db

    if (epoch+1) % 500 == 0:
        print(f"Epoch {epoch+1}, MSE={mse(y_true,
predict(x)):.2f}, w={w:.2f}, b={b:.2f}")
```

```
# After training
print("Final w, b:", w, b)
print("Predictions after training:", np.round(predict(x), 1))

# Unseen data (new student)
new_hours = 6.5
print("Predicted score for 6.5 hours:", predict(new_hours))
```

Output

```
Predicted score for 6.5 hours: 68.82110989587255
```



Common beginner mistakes

Using very large learning rate (lr) → loss becomes NaN or increases.

Not scaling features when inputs have very different ranges (we will learn this later).

Thinking weight is 'fixed' from start — it changes during training.

Quick exercises

Change the dataset values (scores) and see how final w and b change.

Try lr = 0.1 and observe instability. Add noise to scores and see the model behavior.

Chapter 3
Building Your First Neural Network

Why this chapter matters: A neural network is not magic. It is a structured learning system that takes inputs, passes them through hidden layers, learns patterns from examples, and produces an output. This chapter is written to make neural networks feel simple, practical, and approachable.

1. What is a Neural Network?

A neural network is a machine learning model that learns patterns from data. In plain words, we give it inputs, it performs calculations through connected neurons, and it produces an output. The three main parts are the input layer, one or more hidden layers, and the output layer.

- Input layer: receives the feature values.
- Hidden layers: learn relationships between the inputs.
- Output layer: gives the final prediction.

2. What we will build in this chapter

We will build two versions of a simple feedforward neural network using the same dataset and the same three input features.

Example	Architecture	Purpose	Output Meaning
Example 1	3 -> 6 -> 4 -> 1	Binary classification	0 = No Default, 1 = Default
Example 2	3 -> 6 -> 4 -> 2	Two-class probability output	[1, 0] = No Default, [0, 1] = Default

3. Business example and sample data

To make the idea practical, we will use a very small loan-default example. Our goal is to predict whether a customer may default based on three inputs: Age, Income, and Loan Amount.

Record	Age	Income	LoanAmount	Default
1	22	25	5	0
2	25	28	7	0
3	28	35	10	0
4	30	40	12	0
5	35	45	18	0
6	40	50	20	0
7	45	55	25	0
8	23	20	15	1
9	26	22	18	1
10	29	24	20	1
11	31	26	22	1
12	34	27	24	1
13	38	30	28	1
14	42	32	30	1
15	46	36	35	1

We will also keep one unseen record for final prediction after training.

Age	Income	LoanAmount
33	29	23

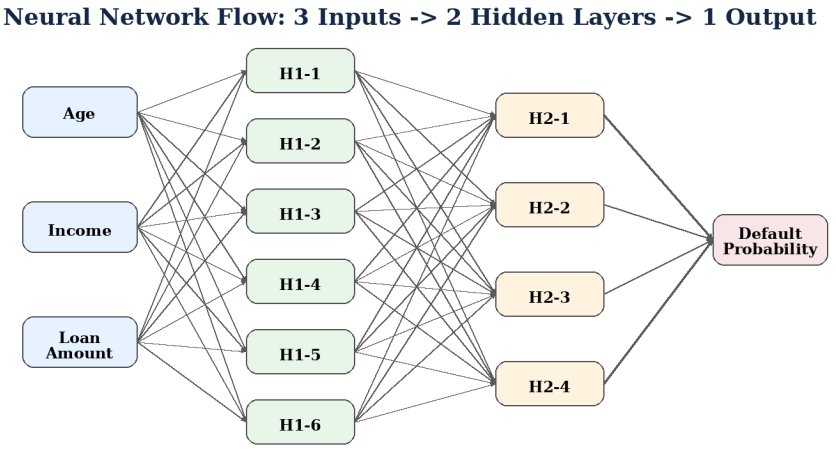
4. How data flows inside the network

Each neuron takes inputs, multiplies them by weights, adds a bias, and then applies an activation function. The result moves forward to the next layer.

Basic neuron calculation: z = (x1*w1 + x2*w2 + x3*w3) + b, then a = activation(z)

For hidden layers, we will use ReLU. For the binary-output model, we will use Sigmoid in the final layer. For the two-output model, we will use Softmax in the final layer.

Example 1 diagram: 3 inputs -> 2 hidden layers -> 1 output



Inputs are scaled, passed through ReLU-based hidden layers, and converted into a final probability using Sigmoid.

Figure 1. Data flow for the binary-output neural network.

Example 2 diagram: 3 inputs -> 2 hidden layers -> 2 outputs

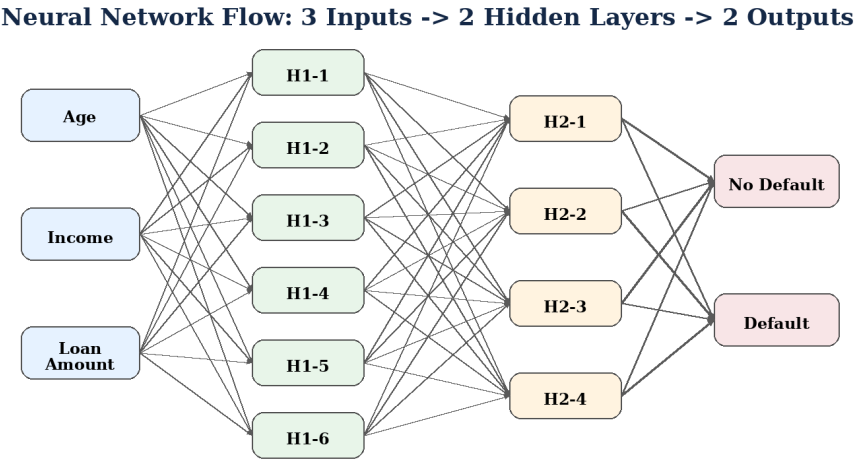


Figure 2. Data flow for the two-output neural network.

5. Step-by-step working with one sample record

Suppose one record has Age = 33, Income = 29, and Loan Amount = 23. One hidden neuron may use weights 0.2, 0.5, and -0.3 with bias 0.1.

The neuron computes: z = (33*0.2) + (29*0.5) + (23*-0.3) + 0.1 = 14.3

If we apply ReLU, the output becomes max(0, 14.3) = 14.3. This process happens in every neuron and in every layer.

6. How layer selection needs to happen

Layer selection means deciding how many hidden layers should be used. There is no universal fixed number. The right choice depends on the problem complexity, data size, and how much non-linear learning is needed.

- Use fewer layers when the dataset is small and the problem is simple.

- Use more layers when the patterns are richer, the data is larger, or the problem involves images, audio, or language.
- For a first neural network chapter, too many layers add confusion without adding learning value.

What we selected in this example: 2 hidden layers.

Why we selected it: one hidden layer is sometimes too small to show how deep learning really flows, while more than two hidden layers would be unnecessary for this small educational dataset. Two hidden layers create a good balance between simplicity and real architecture understanding.

7. How neuron selection needs to happen

Neuron selection means deciding how many neurons should be in each hidden layer. There is no exact magic formula, but the choice should be reasonable for the data size and problem complexity.

- Too few neurons may underfit and miss useful patterns.
- Too many neurons may overfit, especially when the dataset is small.
- A small structured dataset usually needs a small network.

What we selected in this example: 6 neurons in hidden layer 1 and 4 neurons in hidden layer 2.

Why we selected it: the dataset is very small, we have only three input features, and we want the network to stay understandable while still learning non-linear patterns. So the architecture 3 -> 6 -> 4 -> 1 and 3 -> 6 -> 4 -> 2 is simple and practical.

8. How model selection needs to happen

Model selection means deciding which machine learning model best fits the problem. Before choosing a neural network, always ask whether the problem is classification or regression, whether the data is tabular or unstructured, and whether a simpler model might also solve it.

- Possible models for this example: Logistic Regression, Decision Tree, Random Forest, or Neural Network.

- Since the purpose of this chapter is to teach architecture, neurons, layers, and deep learning flow, a feedforward neural network is the best choice.
- The target is a category, so this is a classification problem.

What we selected in this example: a feedforward neural network.

Why we selected it: it is the easiest neural network architecture for beginners, it works well with tabular numeric data, and it is perfect for understanding hidden layers and forward propagation.

9. Activation, optimizer, and loss selection

Component	What we selected	Why
Hidden layer activation	ReLU	Simple, fast, and widely used for hidden layers
Output activation - Example 1	Sigmoid	Produces one probability between 0 and 1
Output activation - Example 2	Softmax	Produces a probability for each output class
Optimizer	Adam	A strong default choice for fast and stable learning
Loss - Example 1	Binary Cross Entropy	Correct loss for one-output binary classification
Loss - Example 2	Categorical Cross Entropy / CrossEntropyLoss	Correct loss for multi-class style outputs

10. Data preparation before training

The three input columns use different numeric ranges. Because neural networks train better when features are on a comparable scale, we scale the inputs before training.

What we selected in this example: StandardScaler.

Why we selected it: it is simple, reliable, and very common for structured numeric data. It helps the optimizer learn more smoothly and reduces instability during training.

11. How training needs to be validated

Training validation means checking whether the model is learning patterns that can generalize, not just memorizing the training rows.

- Training loss should generally reduce.
- Validation loss should also improve or remain stable.
- Training accuracy should not become perfect while validation remains poor.
- The model should work on at least one unseen example.

What we selected in this example: train-test split, validation split during training, and one unseen record for final prediction.

Why we selected it: this is the simplest way to teach model validation without adding too much complexity for a first chapter.

12. How result needs to be validated

Result validation is different from training validation. Here, we check whether the final prediction makes business sense.

- Use an unseen record.
- Inspect the output probability.
- Check whether the predicted class looks logical from the pattern in the data.
- On larger datasets, use a confusion matrix, precision, recall, and F1-score.

What we selected in this example: one unseen record with Age = 33, Income = 29, Loan Amount = 23.

Why we selected it: a simple first demonstration is enough to show how the trained network makes a prediction on new data.

13. TensorFlow implementation - Example 1

This version builds the architecture 3 -> 6 -> 4 -> 1 for binary classification.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

data = {
    'Age': [22,25,28,30,35,40,45,23,26,29,31,34,38,42,46],
    'Income': [25,28,35,40,45,50,55,20,22,24,26,27,30,32,36],
    'LoanAmount': [5,7,10,12,18,20,25,15,18,20,22,24,28,30,35],
    'Default': [0,0,0,0,0,0,0,1,1,1,1,1,1,1,1]
}

df = pd.DataFrame(data)

X = df[['Age', 'Income', 'LoanAmount']].values
y = df['Default'].values

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

model = Sequential([
```

```
    Dense(6, activation='relu', input_shape=(3,)),
    Dense(4, activation='relu'),
    Dense(1, activation='sigmoid')
])

model.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['accuracy']
)

history = model.fit(
    X_train, y_train,
    validation_split=0.2,
    epochs=100,
    verbose=1
)

loss, accuracy = model.evaluate(X_test, y_test)
print("Test Loss:", loss)
print("Test Accuracy:", accuracy)

unseen = np.array([[33, 29, 23]])
unseen_scaled = scaler.transform(unseen)

prediction = model.predict(unseen_scaled)
print("Predicted Probability:", prediction[0][0])

predicted_class = 1 if prediction[0][0] >= 0.5 else 0
print("Predicted Class:", predicted_class)
```

Result

```
Test Loss: 0.7317075133323669
Test Accuracy: 0.6666666865348816
Predicted Probability: 0.52964276
Predicted Class: 1
```

14. TensorFlow implementation - Example 2

This version builds the architecture 3 -> 6 -> 4 -> 2 for two-class probability output.

```
from tensorflow.keras.utils import to_categorical

y_cat = to_categorical(y, num_classes=2)

X_train, X_test, y_train, y_test = train_test_split(
    X, y_cat, test_size=0.2, random_state=42
)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

model2 = Sequential([
    Dense(6, activation='relu', input_shape=(3,)),
    Dense(4, activation='relu'),
    Dense(2, activation='softmax')
])

model2.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

history2 = model2.fit(
```

```
    X_train, y_train,
    validation_split=0.2,
    epochs=100,
    verbose=1
)

loss, accuracy = model2.evaluate(X_test, y_test)
print("Test Loss:", loss)
print("Test Accuracy:", accuracy)

unseen = np.array([[33, 29, 23]])
unseen_scaled = scaler.transform(unseen)

prediction = model2.predict(unseen_scaled)
print("Predicted Probabilities:", prediction[0])

predicted_class = np.argmax(prediction[0])
print("Predicted Class:", predicted_class)
```

Result

```
Test Loss: 0.5201669335365295
Test Accuracy: 0.6666666865348816
Predicted Probabilities: [0.34534413 0.6546558 ]
Predicted Class: 1
```

15. PyTorch implementation - Example 1

Now we build the same binary-output architecture in PyTorch. This version is very useful because it shows the forward method, loss calculation, backward pass, and optimizer step more explicitly.

```
import torch
import torch.nn as nn
import torch.optim as optim
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
```

```
from sklearn.preprocessing import StandardScaler

data = {
    'Age': [22,25,28,30,35,40,45,23,26,29,31,34,38,42,46],
    'Income': [25,28,35,40,45,50,55,20,22,24,26,27,30,32,36],
    'LoanAmount': [5,7,10,12,18,20,25,15,18,20,22,24,28,30,35],
    'Default': [0,0,0,0,0,0,0,1,1,1,1,1,1,1,1]
}

df = pd.DataFrame(data)

X = df[['Age', 'Income', 'LoanAmount']].values
y = df['Default'].values.reshape(-1, 1)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

X_train = torch.tensor(X_train, dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32)
y_test = torch.tensor(y_test, dtype=torch.float32)

class FirstNN(nn.Module):
    def __init__(self):
        super(FirstNN, self).__init__()
        self.fc1 = nn.Linear(3, 6)
        self.fc2 = nn.Linear(6, 4)
        self.fc3 = nn.Linear(4, 1)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()
```

```
    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.sigmoid(self.fc3(x))
        return x

model = FirstNN()
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

for epoch in range(100):
    model.train()
    optimizer.zero_grad()
    outputs = model(X_train)
    loss = criterion(outputs, y_train)
    loss.backward()
    optimizer.step()

    if (epoch + 1) % 10 == 0:
        print(f"Epoch [{epoch+1}/100], Loss: {loss.item():.4f}")

model.eval()
with torch.no_grad():
    test_outputs = model(X_test)
    predicted = (test_outputs >= 0.5).float()
    accuracy = (predicted.eq(y_test).sum() / y_test.shape[0]).item()

print("Test Accuracy:", accuracy)

unseen = np.array([[33, 29, 23]])
unseen_scaled = scaler.transform(unseen)
unseen_tensor = torch.tensor(unseen_scaled, dtype=torch.float32)

model.eval()
```

```
with torch.no_grad():
    pred = model(unseen_tensor)
    print("Predicted Probability:", pred.item())
    print("Predicted Class:", 1 if pred.item() >= 0.5 else 0)
```

Result

```
Test Accuracy: 1.0
Predicted Probability: 0.7699709534645081
Predicted Class: 1
```

16. PyTorch implementation - Example 2

This version uses two output neurons with CrossEntropyLoss.

```
y = df['Default'].values
X = df[['Age', 'Income', 'LoanAmount']].values

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

X_train = torch.tensor(X_train, dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.long)
y_test = torch.tensor(y_test, dtype=torch.long)

class FirstNNMulti(nn.Module):
    def __init__(self):
        super(FirstNNMulti, self).__init__()
        self.fc1 = nn.Linear(3, 6)
        self.fc2 = nn.Linear(6, 4)
        self.fc3 = nn.Linear(4, 2)
        self.relu = nn.ReLU()
```

```
    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x

model2 = FirstNNMulti()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model2.parameters(), lr=0.01)

for epoch in range(100):
    model2.train()
    optimizer.zero_grad()
    outputs = model2(X_train)
    loss = criterion(outputs, y_train)
    loss.backward()
    optimizer.step()

    if (epoch + 1) % 10 == 0:
        print(f"Epoch [{epoch+1}/100], Loss: {loss.item():.4f}")

model2.eval()
with torch.no_grad():
    outputs = model2(X_test)
    _, predicted = torch.max(outputs, 1)
    accuracy = (predicted == y_test).sum().item() / y_test.size(0)

print("Test Accuracy:", accuracy)

unseen = np.array([[33, 29, 23]])
unseen_scaled = scaler.transform(unseen)
unseen_tensor = torch.tensor(unseen_scaled, dtype=torch.float32)

model2.eval()
```

```
with torch.no_grad():
    outputs = model2(unseen_tensor)
    probabilities = torch.softmax(outputs, dim=1)
    predicted_class = torch.argmax(probabilities, dim=1)

print("Predicted Probabilities:", probabilities.numpy())
print("Predicted Class:", predicted_class.item())
```

Result

```
Test Accuracy: 1.0
Predicted Probabilities: [[0.00449392 0.99550605]]
Predicted Class: 1
```

17. What exactly we selected in this chapter

Decision Area	Selection	Why it was selected
Problem type	Classification	The output is a category, not a continuous value
Model	Feedforward neural network	Best first network for structured tabular data
Input features	Age, Income, LoanAmount	Simple, numeric, and easy to explain
Hidden layers	2	Enough depth to show learning flow without making the model too complex
Neurons	6 and 4	Small but practical for a small dataset
Scaling	StandardScaler	Makes learning more stable
Validation	Train-test split + validation split + unseen record	Simple and effective for learning the process

18. How to know whether the neural network is good

- Training loss should reduce over time.
- Validation loss should remain reasonable and should not diverge badly from training loss.
- Test accuracy should be acceptable for the size and nature of the data.
- The prediction on the unseen record should make logical sense.
- The model should not be too large for the size of the dataset.

19. Common mistakes beginners make

- Using too many hidden layers for a very small dataset.
- Choosing the wrong output activation or loss function.
- Skipping feature scaling.
- Looking only at training accuracy and ignoring validation behavior.
- Using a very large number of neurons for a very small problem.

20. TensorFlow and PyTorch - how to think about both

TensorFlow is excellent when you want fast, high-level model development and strong production support. PyTorch is excellent when you want flexibility and deeper control over what happens in the training loop.

- TensorFlow is easier for quick model building with Sequential and fit.
- PyTorch is excellent for understanding the forward pass, gradient flow, and optimizer step.
- A strong AI engineer should learn both.

21. Final conclusion

Your first neural network should not try to be big. It should try to be clear. Once you understand how inputs move into hidden layers, how neurons transform values, how the output is produced, and how training is validated, you have learned the real foundation of deep learning.

In this chapter, we built two neural network architectures, explained layer selection, neuron selection, model selection, and validation choices, and implemented the same ideas in TensorFlow and PyTorch. That is the right starting point for understanding neural network development architecture and framework in a practical, simple way.

The real power of AI learning begins when complex topics become simple in your mind. That is the purpose of this chapter.

22. Quick recap

- We used 3 input features: Age, Income, and Loan Amount.
- We built two architectures: 3 -> 6 -> 4 -> 1 and 3 -> 6 -> 4 -> 2.
- We used ReLU in hidden layers, Sigmoid for the one-output model, and Softmax for the two-output model.
- We used Adam as optimizer and selected the loss function based on the output design.
- We validated training using a train-test split, validation during training, and one unseen record.

Practice idea: Try changing the hidden-layer sizes to 8 and 5, increase the number of epochs, add one extra feature such as YearsWithBank, and compare the behavior in TensorFlow and PyTorch.

23. Building blocks for model development

Model development becomes easier when you think in building blocks rather than jumping directly into coding. A good neural network is usually built step by step, with each block solving one part of the problem.

The main building blocks for model development in this chapter are:

- Problem definition - clearly decide whether the task is classification or regression.
- Input feature selection - choose the columns that carry useful business meaning.
- Data preparation - clean the data, separate inputs and target, and scale the inputs.

- Train-validation-test strategy - decide how you will split the data before training.
- Architecture design - choose the number of inputs, hidden layers, neurons, and outputs.
- Activation selection - choose ReLU, Sigmoid, or Softmax based on the role of each layer.
- Loss function selection - match the loss with the output type.
- Optimizer selection - choose how the model will update its weights, such as Adam.
- Training loop - train for multiple epochs and observe whether loss is reducing.
- Validation loop - check whether the model is learning general patterns, not memorizing training rows.
- Final testing - test the model on unseen data and verify the business logic behind the result.

What we selected in this chapter: classification problem, three numeric input features, StandardScaler, two hidden layers, ReLU for hidden layers, Sigmoid or Softmax in the output, Adam optimizer, and train-test-validation thinking.

24. Building blocks for model testing

Model testing is not just about checking one final accuracy number. Testing should tell us whether the model is stable, reliable, and useful on new data.

The main building blocks for model testing are:

- Training metrics check - observe training loss and training accuracy.
- Validation metrics check - compare validation loss and validation accuracy against training behavior.
- Test-set evaluation - use the held-out test data that the model did not see during training.
- Unseen sample testing - take one or more new records and check the prediction.
- Probability inspection - do not only look at the class label; also inspect the confidence or probability.

- Error review - review wrong predictions and understand what type of records fail.
- Business logic review - confirm that the result makes sense for the domain, not just mathematically.
- Stability review - train more than once, if needed, and check whether the results stay broadly similar.

What we selected in this chapter: training loss, validation split, test accuracy, and one unseen record for logical validation.

For larger real-world projects, you would also add a confusion matrix, precision, recall, F1-score, ROC-AUC, and class-wise error analysis.

25. How to improve training accuracy and change the model based on training and testing results

A neural network is rarely perfect in the first attempt. Good model development means looking at the training result, testing result, and then making controlled improvements.

25.1 Read the training and testing signs properly

There are four common situations every beginner should understand.

Case 1 - Training accuracy is low and validation accuracy is also low.

Meaning: the model is underfitting. It is too weak to learn the pattern well.

Changes you can make:

- Increase the number of neurons slightly.
- Add one more useful feature.
- Train for more epochs.
- Try a slightly better learning rate.
- Check whether the data needs better scaling or cleaning.

Case 2 - Training accuracy is high but validation or test accuracy is low.

Meaning: the model is overfitting. It has memorized training data too much.

Changes you can make:

- Reduce the number of neurons or hidden layers.

- Use dropout or regularization.
- Stop training earlier using early stopping.
- Add more training data if possible.
- Remove noisy or weak features.

Case 3 - Training loss is reducing very slowly.

Meaning: learning may be too slow or the model configuration may not be suitable.

Changes you can make:

- Tune the learning rate.
- Try Adam if you are not already using it.
- Confirm that the input data is scaled.
- Check whether the architecture is too small.

Case 4 - Test results change a lot every time you retrain.

Meaning: the dataset may be too small, too noisy, or the model may be unstable.

Changes you can make:

- Use a fixed random seed.
- Collect more data.
- Make the model slightly simpler.
- Use cross-validation for stronger confidence.

25.2 Practical improvement path for this chapter

If our first result is not good enough, the best improvement order for this example would be:

- First, verify data quality and scaling.
- Second, increase epochs from 100 to 150 or 200 and observe the curves.
- Third, try hidden neurons like 8 and 5 instead of 6 and 4.
- Fourth, compare one-output versus two-output architecture.
- Fifth, add one more meaningful feature such as YearsWithBank or MissedPayments.

- Sixth, add early stopping if validation loss starts increasing.

25.3 What we would change based on results

For this chapter, our first selected model is intentionally simple: 3 -> 6 -> 4 -> 1 or 3 -> 6 -> 4 -> 2. If training and testing results are weak, I would first tune the number of epochs and hidden neurons before making the network deeper.

Why? Because for a very small dataset, the most common problem is not that the model is too shallow. The common problem is that the data is too limited, and a very deep model may only create overfitting.

So the safest improvement choices for this example are:

- improve data quality
- improve feature selection
- slightly tune neurons and epochs
- monitor validation carefully
- keep the architecture simple and explainable

25.4 How to make training more accurate in a practical way

To make training more accurate, do not think only about changing the model. Accuracy improves when data, architecture, and validation all work together.

A practical checklist is:

- Use clean and meaningful input features.
- Scale the features properly.
- Match the output layer and loss function correctly.
- Start with a small architecture and then tune it.
- Watch both training and validation metrics.
- Do not trust one accuracy number alone.
- Test on unseen records.
- Make one change at a time so you know what improved the result.

Chapter 4
What Happens in Training?

Definition

Training is the process of finding good values of weights and biases so that the model's predictions become close to the real answers. Think of weights and bias like knobs you turn to reduce mistakes.

Syntax / Formula

Key formula (small and important)

```
Prediction (for one neuron): y_pred = w*x + b
Error for one data row: e = y_true - y_pred
Loss (MSE): Loss = (1/N) * Σ (y_true - y_pred)^2

Gradient Descent update:
  w_new = w_old - lr * (dLoss/dw)
  b_new = b_old - lr * (dLoss/db)
```

Why?

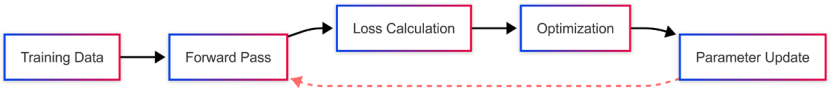
Because once you understand training, any architecture (ANN/CNN/RNN/Transformer) becomes a different shape of the same training loop.

What?

We will train the same student-score problem using TensorFlow/Keras so you learn the standard industry way.

How?

1) Define the model (layers).
2) Compile it (choose loss + optimizer).
3) Fit it (train) for some epochs.
4) Evaluate and predict on unseen data.



Let's Code!

```
import numpy as np
import tensorflow as tf
from tensorflow import keras

# Same tiny dataset
x = np.array([1,2,3,4,5,6,7,8,9,10], dtype=float)
y = np.array([35,40,45,55,60,65,70,78,85,92], dtype=float)

# Keras expects 2D input for Dense layers: shape (N, features)
x2 = x.reshape(-1, 1)

model = keras.Sequential([
    keras.layers.Dense(1, input_shape=(1,))  # one neuron
])

model.compile(
    optimizer=keras.optimizers.SGD(learning_rate=0.01),
    loss="mse"
)

history = model.fit(
    x2, y,
    epochs=300,
    batch_size=5,
    verbose=0
)

print("Final training loss:", history.history["loss"][-1])

# See learned weight and bias
w, b = model.layers[0].get_weights()
print("Learned weight:", w.flatten()[0])
print("Learned bias:", b[0])

# Predict on unseen student
new_hours = np.array([[6.5]], dtype=float)
pred = model.predict(new_hours, verbose=0)
print("Predicted score for 6.5 hours:", float(pred[0][0]))
```

Output

```
Final training loss: 2.386559009552002

Learned weight: 6.639594

Learned bias: 25.721989

Predicted score for 6.5 hours: 68.87934875488281
```

Common beginner mistakes

Too many epochs on small data → overfitting (model memorizes noise).

Batch size too big for tiny data → training becomes less stable to learn patterns.

Using classification loss (binary_crossentropy) for regression by mistake.

Quick exercises

Change optimizer from SGD to Adam and compare loss.

Change batch size to 1 (stochastic) and observe training.

Add one more input feature (e.g., 'sleep hours') and make it a 2-feature model.

Chapter 5
Backpropagation and Gradient Descent (No Fear)

Definition

Backpropagation is just a smart way to compute how much each weight contributes to the final mistake (loss). Once we know that, we adjust each weight in the correct direction.

Syntax / Formula

Key formula (small and important)

```
Chain rule idea:
  If y = f(g(x)), then dy/dx = (dy/dg) * (dg/dx)

In networks:
  Loss depends on output.
  Output depends on last layer weights.
  Last layer output depends on previous layer ...
Backprop uses chain rule layer-by-layer.
```

Why?

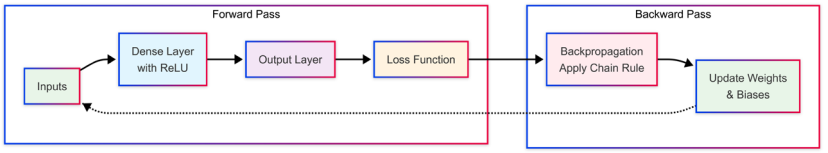
Because beginners often think backprop is magic. It is not. It is just repeated chain rule.

What?

We will build a 2-layer network and watch how TensorFlow computers automatically.

How?

1) Define the model.
2) Run forward pass inside GradientTape.
3) Compute loss.
4) Ask TensorFlow for gradients.
5) Apply gradient descent update.



Let's Code!

```
import numpy as np
import tensorflow as tf


# Tiny dataset with 2 inputs:
# x1 = hours studied, x2 = practice tests taken
X = np.array([[1, 0],[2, 1],[3, 1],[4, 2],[5, 2],[6, 3],[7, 3],[8, 4],[9, 4],[10, 5],
], dtype=np.float32)


y = np.array([30, 38, 45, 52, 58, 64, 70, 78, 86, 92], dtype=np.float32).reshape(-1,1)


model = tf.keras.Sequential([
    tf.keras.layers.Dense(4, activation="relu", input_shape=(2,)),
    tf.keras.layers.Dense(1)  # regression
])


optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)


def mse(y_true, y_pred):
    return tf.reduce_mean(tf.square(y_true - y_pred))


# One manual training loop so you SEE the steps
for epoch in range(1, 501):
    with tf.GradientTape() as tape:
        y_pred = model(X, training=True)
        loss = mse(y, y_pred)


    grads = tape.gradient(loss, model.trainable_variables)
```

```
    optimizer.apply_gradients(zip(grads, model.trainable_variables))

    if epoch % 100 == 0:
        print(f"Epoch {epoch}, loss={loss.numpy():.2f}")

# Predict unseen
new_student = np.array([[6.5, 3]], dtype=np.float32)
print("Predicted score:", float(model(new_student).numpy()[0][0]))
```

Output

```
Epoch 100, loss=448.77
Epoch 200, loss=381.22
Epoch 300, loss=380.03
Epoch 400, loss=380.01
Epoch 500, loss=380.01
Predicted score: 61.2974853515625
```

Common beginner mistakes

Expecting zero loss — real data has noise; aim for good generalization.

Using sigmoid for hidden layers in deep nets can cause vanishing gradients (we will discuss).

Not shuffling data in real projects (small demos are fine).

Quick exercises

Change hidden units from 4 to 8 and see the loss trend.

Change activation from ReLU to tanh and compare training speed.

Try learning_rate = 0.1 and observe if training becomes unstable.

Chapter 6
Your First Deep Learning Model in TensorFlow & Keras

Definition

In this chapter, you will build your **first complete Deep Learning model** using **TensorFlow and Keras**, starting from **hardcoded raw data** (so everyone can run it easily).

You will learn:

- how to structure data into correct **shapes**
- how scaling (stats) improves learning
- how a Dense layer uses the equation $XW + b$
- how compile() + fit() works in practice
- how to evaluate, predict, and save the model

Why?

Most beginners write a model, but they don't learn the **real pipeline**.
In real life, every deep learning model follows this flow:

Raw Data → Prepare Data → Build Model → Compile → Train → Evaluate → Predict → Save

Once you learn this pipeline, you can build models for:

- house price
- sales forecasting
- customer churn
- fraud detection
- any tabular business dataset

What We Will Build

Project

House Price Predictor (Regression Model)

Input features (X)

- size (sqft)
- bedrooms
- age (years)

Output (y)

- price (lakhs)

This is **regression** because output is a number.

Step 1: Start with Raw Hardcoded Data

1.1 Raw dataset (human readable)

House	Size	Bedrooms	Age	Price
H1	1000	2	10	75
H2	850	2	12	60
H3	1200	3	8	95
H4	700	2	20	40
H5	1500	3	5	140
H6	2000	4	2	220

1.2 Convert into tensors (arrays)

Here shape must be:

- X shape → (rows, features) = (6, 3)
- y shape → (rows, 1) = (6, 1)

This shape mindset is the first best practice.

Step 2: The Math Behind This Model (Simple)

2.1 What a Dense layer calculates

A Dense layer uses:

$$Z = XW + b$$

Then activation:

$$A = f(Z)$$

For regression output layer, we normally keep **no activation**, because price can be any number.

2.2 Loss function for regression

We use **MSE**:

$$MSE = \frac{1}{n}\sum(y - \hat{y})^2$$

The model tries to reduce this loss.

Implementation (Full Pipeline Code)
Lets Code

```
import numpy as np
import tensorflow as tf


# ---------------------------------------------------
# 1) RAW HARD-CODED DATA
# ---------------------------------------------------
# X = [size_sqft, bedrooms, age_years]
X = np.array([
    [1000, 2, 10],
    [ 850, 2, 12],
    [1200, 3,  8],
    [ 700, 2, 20],
    [1500, 3,  5],
    [2000, 4,  2]
], dtype=np.float32)
```

```
# y = price_lakhs
y = np.array([
    [75],
    [60],
    [95],
    [40],
    [140],
    [220]
], dtype=np.float32)

print("X shape:", X.shape)  # (6,3)
print("y shape:", y.shape)  # (6,1)

# ----------------------------------------------------
# 2) TRAIN/TEST SPLIT (simple)
# ----------------------------------------------------
# First 4 for training, last 2 for testing
X_train, y_train = X[:4], y[:4]
X_test,  y_test  = X[4:], y[4:]

print("\nTrain shape:", X_train.shape, y_train.shape)
print("Test shape :", X_test.shape, y_test.shape)

# ----------------------------------------------------
# 3) STANDARDIZATION (Stats)
# x' = (x - mean) / std
# IMPORTANT: mean/std from TRAIN ONLY
# ----------------------------------------------------
mean = X_train.mean(axis=0)
std  = X_train.std(axis=0) + 1e-7

X_train_s = (X_train - mean) / std
X_test_s  = (X_test  - mean) / std

# ----------------------------------------------------
```

```
# 4) BUILD MODEL (Keras)
    model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(3,)),
    tf.keras.layers.Dense(8, activation="relu"),
    tf.keras.layers.Dense(4, activation="relu"),
    tf.keras.layers.Dense(1)   # regression output
])
# 5) COMPILE (optimizer + loss + metrics)
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.01),
    loss="mse",
    metrics=["mae"]  # easy to understand error
)
# 6) TRAIN (fit)
history = model.fit(
    X_train_s, y_train,
    epochs=300,
    verbose=0
)
print("\nTraining finished.")
print("Final training loss:", round(float(history.history["loss"][-
1]), 2))
print("Final training mae :", round(float(history.history["mae"][-
1]), 2))


# ------------------------------------------------
# 7) EVALUATE ON UNSEEN TEST DATA
test_loss, test_mae = model.evaluate(X_test_s, y_test, verbose=0)
print("\nTest MAE (lakhs):", round(float(test_mae), 2))


# 8) PREDICT ON A NEW UNSEEN HOUSE
new_house = np.array([[1100, 3, 9]], dtype=np.float32)


# scale using TRAIN mean/std
new_house_s = (new_house - mean) / std
```

```
pred = model.predict(new_house_s, verbose=0)
print("\nNew house prediction (lakhs):", round(float(pred[0][0]),
2))


# ------------------------------------------------
# 9) SAVE + LOAD (Deployment readiness)
model.save("house_price_model.keras")
loaded_model = tf.keras.models.load_model("house_price_model.keras")


pred2 = loaded_model.predict(new_house_s, verbose=0)
print("Loaded model prediction (lakhs):", round(float(pred2[0][0]),
2))
```

output

X shape: (6, 3)

y shape: (6, 1)

Train shape: (4, 3) (4, 1)

Test shape : (2, 3) (2, 1)

Training finished.

Final training loss: 0.38

Final training mae : 0.53

Test MAE (lakhs): 14.07

New house prediction (lakhs): 78.78

Loaded model prediction (lakhs): 78.78

Line-by-Line Meaning (Very Easy)

1) Raw data

You created X and y with correct shapes.

2) Train/Test split

training teaches weights

test checks generalization

3) Standardization

calculated mean/std only from training

scaled both train and test using same stats
prevents data leakage

4) Build model

Dense(8) learns basic patterns

Dense(4) learns deeper combination

output Dense(1) gives price

5) Compile

Adam updates weights

MSE measures wrongness

MAE shows average error in easy terms

6) Fit

TensorFlow repeats:

forward pass → $\hat{y}$

loss → error number

gradients → slope

update weights → reduce error

repeat epochs

7) Evaluate

Test is unseen, so it shows real performance.

8) Predict - Prediction uses learned weights only.

9) Save/Load - Model can now be used in real applications.

Best Practices (Must Follow Always)

- Always print shapes
- Always scale tabular numeric features
- Mean/std must come from training only

- Start with Adam + learning_rate 0.001 or 0.01
- Use MAE for regression readability
- Save model after training
- Test on unseen data

Chapter 7
How a Neural Network Learns Ex - House Prices (Weights, Biases, and Prediction).

Build a small **Deep Learning (Neural Network) regression model** that can **predict house price** from basic house features. **Training data (hardcoded inside code):** Use **10–20 records** in the code itself (no external CSV).

Each record contains:

- **Square Feet** (example: 1000)
- **Area** (A / B / C)
- **BHK** (number of bedrooms)
- **Garden** (Yes / No)
- **Price** (target)

Important:

1. **What happens during training** (forward pass → loss → backpropagation → gradient descent updates)
2. **How weights and biases get updated** (batch by batch, epoch by epoch) and how they become the final learned values
3. **Where weights and biases are stored** (inside model layers; also saved in a model file)
4. **How the same trained weights and biases are reused** to predict **new unseen** house data

Specific example to explain (must include):

- House 1: 1000 sqft, Area A, 3 BHK, Garden Yes → Price 7,000,000
- House 2: 800 sqft, Area A, 3 BHK, Garden Yes → Price 6,000,000
- Unseen House 3: 850 sqft, Area A, 3 BHK, Garden Yes → Price?

Also explain hyperparameter choices:

- Input layer neuron count
- Hidden layer neuron count
- Activation function
- Learning rate
- Epochs
- Batch size
- Backpropagation + Gradient Descent
- Save/load model and show parameters

Below is a single, working Python program that does everything: creates hardcoded data, trains a neural network, shows what happens in training, prints weight/bias updates, predicts an unseen house, plots graphs, and saves/loads the model.

```
"
DEEP LEARNING HOUSE PRICE PREDICTION (REGRESSION)
What this program teaches:
1) Build a neural network to predict house price
2) What happens in training (Forward pass -> Loss -> Backprop ->
Update)
3) How weights & biases change and get finalized
4) How trained weights are reused to predict unseen house
5) Graphs + architecture diagram
6) Save model and reload it for reuse
"""


import numpy as np
import matplotlib.pyplot as plt
from tensorflow import keras
from tensorflow.keras import layers


np.random.seed(42)


# 0) HARD-CODED TRAINING DATA (20 records)
#    IMPORTANT: Includes the 2 records you asked for exactly.
```

```
# Encoding:
# Area A=0, B=1, C=2
# Garden No=0, Yes=1

data = [
    # sqft, area, bhk, garden, price
    (1000, 0, 3, 1, 7000000),  # House 1 (exact)
    (800,  0, 3, 1, 6000000),  # House 2 (exact)

    (850,  0, 2, 0, 5600000),
    (900,  0, 3, 0, 6300000),
    (1100, 0, 4, 1, 8200000),

    (1200, 1, 4, 1, 9200000),
    (950,  1, 3, 0, 6800000),
    (1050, 1, 3, 1, 7600000),
    (1300, 1, 5, 1, 11000000),

    (1500, 2, 5, 1, 12500000),
    (1400, 2, 4, 1, 11300000),
    (1250, 2, 4, 0, 9800000),
    (1600, 2, 5, 1, 13200000),

    (780,  0, 2, 1, 5900000),
    (820,  0, 3, 1, 6100000),
    (1020, 1, 4, 0, 8600000),
    (1180, 1, 4, 1, 9500000),
    (1350, 2, 4, 1, 11050000),
    (1450, 2, 5, 0, 11800000),
    (980,  1, 3, 1, 7400000),
]

data = np.array(data, dtype=np.float32)
X = data[:, 0:4]   # features
```

```
y = data[:, 4:5]   # target as 2D (n,1)


# 1) NORMALIZATION (very important for stable training)


# Why normalize?
# - Neural networks learn faster and more stable when features are in similar ranges.
# - Example: sqft ~ 800-1600 (big), area 0-2 (small), garden 0/1.
# - Normalization avoids one feature dominating only because its numeric scale is larger.


Xn = X.copy()
Xn[:, 0] = Xn[:, 0] / 1000.0   # sqft -> thousands (e.g., 1000 -> 1.0)
Xn[:, 1] = Xn[:, 1] / 2.0      # area 0..2 -> 0..1
Xn[:, 2] = Xn[:, 2] / 5.0      # bhk up to 5 -> 0..1
# garden stays 0/1


yn = y / 1_000_000.0           # price -> millions


# 2) MODEL DESIGN (HYPERPARAMETERS + WHY)


"""
Input layer neuron count = 4 (NOT a choice)
Because we have 4 input features:
[Square Feet, Area, BHK, Garden]


Hidden layer neuron count = 8 (a design choice)
Why 8?
- Small dataset (20 records) => keep model small to reduce overfitting.
- 8 is a simple middle value: not too small, not too big.
- In practice you try multiple values (4, 8, 16...) and validate.


Activation function (Hidden layer) = ReLU
ReLU(x) = max(0, x)
```

```
Why?
- Very common, fast, reduces vanishing gradient problem.


Output activation = Linear
Because this is regression (we want any real number output, not 0/1)


Learning rate = 0.001 (Adam optimizer)
Why?
- Too large => training may jump around and fail
- Too small => training becomes too slow
- 0.001 is a widely good starting point for Adam.


Epochs = 200
- Epoch means: model sees the full training data once.
- Small dataset => more epochs are okay.
- But we will use EarlyStopping to stop automatically when
improvement stops.


Batch size = 4
- Batch size means: how many records used before updating weights
once.
- With 20 records: batch size 4 => 5 updates per epoch.
- Smaller batch => more frequent updates (noisier but often
helpful).
"""


model = keras.Sequential([
    layers.Dense(8, activation="relu", input_shape=(4,),
name="hidden_layer"),
    layers.Dense(1, activation="linear", name="output_layer")
])


model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=0.001),
    loss="mean_squared_error",
    metrics=["mae"]
```

```
)

print("\nMODEL SUMMARY:")
model.summary()

# 3) WHAT HAPPENS IN TRAINING (CORE EXPLANATION)

"""
Training is repeated many times and each time it does:

A) Forward pass:
   - Take inputs (X)
   - Multiply by weights, add bias
   - Apply activation
   - Produce predicted price y_pred

B) Loss calculation:
   - Compare y_pred with actual y
   - Example (MSE): average of (y - y_pred)^2

C) Backpropagation:
   - Compute gradients (how much each weight caused the error)
   - Gradient tells the direction to change weights to reduce loss

D) Gradient Descent Update:
   - weight_new = weight_old - learning_rate * gradient
   - bias_new   = bias_old   - learning_rate * gradient

IMPORTANT:
Weights and biases do NOT "finalize" after one record.
They improve gradually after many updates.
Each batch updates weights slightly.
After many epochs, weights become stable => "finalized".
"""
```

```
# 4) TRACK WEIGHTS DURING TRAINING (to SEE learning)


class WeightHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs=None):
        self.w00 = []  # track one example weight over time (input
sqft -> hidden neuron 0)
        self.losses = []


    def on_epoch_end(self, epoch, logs=None):
        W_hidden, b_hidden =
self.model.get_layer("hidden_layer").get_weights()
        self.w00.append(W_hidden[0, 0])  # weight from feature-0
(sqft) to neuron-0
        self.losses.append(logs["loss"])


weight_hist = WeightHistory()


early_stop = keras.callbacks.EarlyStopping(
    monitor="loss",
    patience=30,
    restore_best_weights=True
)


# Train
history = model.fit(
    Xn, yn,
    epochs=200,
    batch_size=4,
    callbacks=[weight_hist, early_stop],
    verbose=1
)


print("\nTraining finished at epochs:",
len(history.history["loss"]))


# 5) WEIGHTS & BIASES AFTER TRAINING (THE MODEL'S "MEMORY")
```

```
hidden_W, hidden_b = model.get_layer("hidden_layer").get_weights()
out_W, out_b = model.get_layer("output_layer").get_weights()

print("\nFINAL LEARNED PARAMETERS:")
print("Hidden layer weights shape:", hidden_W.shape, " (4 inputs ->
8 neurons)")
print("Hidden layer bias shape:   ", hidden_b.shape)
print("Output layer weights shape:", out_W.shape, " (8 -> 1)")
print("Output layer bias shape:   ", out_b.shape)

# Show a small sample of weights (for interpretation)
print("\nSample weights (Hidden Neuron 0):")
print("  Weight for sqft   :", float(hidden_W[0, 0]))
print("  Weight for area   :", float(hidden_W[1, 0]))
print("  Weight for bhk    :", float(hidden_W[2, 0]))
print("  Weight for garden :", float(hidden_W[3, 0]))
print("  Bias neuron0      :", float(hidden_b[0]))

# 6) EXPLAIN YOUR SPECIFIC EXAMPLE (House1, House2, House3)

house1 = np.array([[1000, 0, 3, 1]], dtype=np.float32)
house2 = np.array([[ 800, 0, 3, 1]], dtype=np.float32)
house3 = np.array([[ 850, 0, 3, 1]], dtype=np.float32)  # unseen

def normalize_house(h):
    h = h.copy()
    h[:, 0] /= 1000.0
    h[:, 1] /= 2.0
    h[:, 2] /= 5.0
    return h

h1n = normalize_house(house1)
h2n = normalize_house(house2)
h3n = normalize_house(house3)
```

```
# Predict using trained model
pred_h3_million = model.predict(h3n, verbose=0)[0, 0]
pred_h3_price = pred_h3_million * 1_000_000.0

print("\nUNSEEN HOUSE 3 PREDICTION:")
print("House 3: 850 sqft, Area A, 3 BHK, Garden Yes")
print("Predicted price (₹):", f"{pred_h3_price:,.2f}")

# 7) SHOW EXACTLY HOW THE SAME WEIGHTS ARE REUSED (Manual forward pass)
"""
During prediction, NO training happens.
No backprop, no update.
Only forward pass using the FINAL trained weights and biases.

Forward pass steps for a single house:
1) Hidden layer:
   z1 = X * W1 + b1
   a1 = ReLU(z1)

2) Output layer:
   y_pred = a1 * W2 + b2
"""

# Manual forward pass for house3
z1 = np.dot(h3n, hidden_W) + hidden_b     # (1,8)
a1 = np.maximum(0, z1)                    # ReLU
z2 = np.dot(a1, out_W) + out_b            # (1,1)

pred_manual_million = z2[0, 0]            # scalar
pred_manual_price = pred_manual_million * 1_000_000.0

print("\nMANUAL FORWARD PASS (same stored weights reused):")
print("Hidden layer output (8 neurons):", np.round(a1[0], 4))
```

```
print("Predicted price (₹) manual:", f"{pred_manual_price:,.2f}")


# 8) PLOTS / GRAPHS (attributes + training behavior)


# 8.1 Feature distributions (each attribute)
feature_names = ["Square_Feet", "Area(0=A,1=B,2=C)", "BHK",
"Garden"]
for i in range(4):
    plt.figure(figsize=(6, 4))
    plt.hist(X[:, i], bins=8, edgecolor="black")
    plt.title(f"Distribution of {feature_names[i]}")
    plt.xlabel(feature_names[i])
    plt.ylabel("Count")
    plt.grid(True)
    plt.show()


# 8.2 Loss curve (learning)
plt.figure(figsize=(6, 4))
plt.plot(history.history["loss"])
plt.title("Training Loss Curve (MSE)")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.grid(True)
plt.show()


# 8.3 Weight change trend (shows learning is updating weights)
plt.figure(figsize=(6, 4))
plt.plot(weight_hist.w00)
plt.title("One Weight Changing Over Time (sqft -> hidden neuron0)")
plt.xlabel("Epoch")
plt.ylabel("Weight value")
plt.grid(True)
plt.show()


# 8.4 Predicted vs Actual (on training data)
```

```
pred_all = model.predict(Xn, verbose=0).flatten()
actual_all = yn.flatten()

plt.figure(figsize=(5, 5))
plt.scatter(actual_all, pred_all)
mn = min(actual_all.min(), pred_all.min())
mx = max(actual_all.max(), pred_all.max())
plt.plot([mn, mx], [mn, mx])  # perfect line
plt.title("Predicted vs Actual (Training Data)")
plt.xlabel("Actual Price (Millions ₹)")
plt.ylabel("Predicted Price (Millions ₹)")
plt.grid(True)
plt.show()

# 9) DRAW A SIMPLE NEURAL NETWORK DIAGRAM

def draw_network_diagram():
    plt.figure(figsize=(9, 4))
    plt.axis("off")

    # positions
    input_y = [3, 2, 1, 0]
    hidden_y = [3.2, 2.6, 2.0, 1.4, 0.8, 0.2, -0.4, -1.0]
    out_y = [1.0]

    # Input nodes
    for i, y0 in enumerate(input_y):
        plt.scatter(1, y0, s=600)
        plt.text(1, y0, f"X{i+1}", ha="center", va="center")
    plt.text(1, 3.8, "Input Layer (4)", ha="center")

    # Hidden nodes
    for i, y0 in enumerate(hidden_y):
        plt.scatter(5, y0, s=450)
```

```
        plt.text(5, y0, f"H{i+1}", ha="center", va="center",
fontsize=8)
    plt.text(5, 4.0, "Hidden Layer (8, ReLU)", ha="center")

    # Output node
    plt.scatter(9, out_y[0], s=700)
    plt.text(9, out_y[0], "Price", ha="center", va="center")
    plt.text(9, 2.2, "Output (1, Linear)", ha="center")

    # Connections (simplified: draw some lines)
    for iy in input_y:
        for hy in hidden_y[:3]:
            plt.plot([1.3, 4.7], [iy, hy], alpha=0.3)
    for hy in hidden_y[:3]:
        plt.plot([5.3, 8.7], [hy, out_y[0]], alpha=0.3)

    plt.title("Neural Network Architecture: 4 → 8 → 1")
    plt.show()

draw_network_diagram()

# 10) SAVE MODEL (weights + biases get stored in file)

model.save("house_price_model.keras")
print("\nModel saved as: house_price_model.keras")

loaded = keras.models.load_model("house_price_model.keras")
pred_loaded_price = loaded.predict(h3n, verbose=0)[0, 0] *
1_000_000.0
print("Prediction using loaded model (House 3) ₹:",
f"{pred_loaded_price:,.2f}")
```

FINAL LEARNED PARAMETERS:

Hidden layer weights shape: (4, 8) (4 inputs -> 8 neurons)

Hidden layer bias shape: (8,)

Output layer weights shape: (8, 1) (8 -> 1)

Output layer bias shape: (1,)

Sample weights (Hidden Neuron 0):

Weight for sqft : -0.21191063523292542

Weight for area : -0.25493937730789185

Weight for bhk : -0.5901622772216797

Weight for garden : -0.5939316749572754

Bias neuron0 : 0.0

UNSEEN HOUSE 3 PREDICTION:

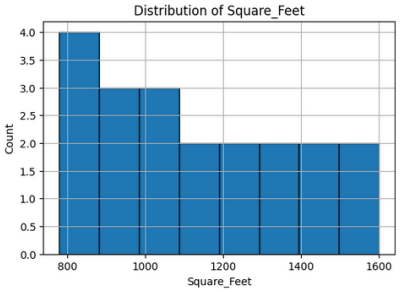
House 3: 850 sqft, Area A, 3 BHK, Garden Yes

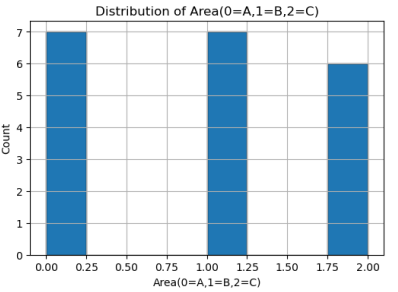
Predicted price (₹): 6,987,956.00

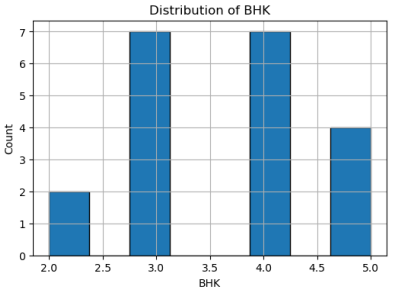
MANUAL FORWARD PASS (same stored weights reused):

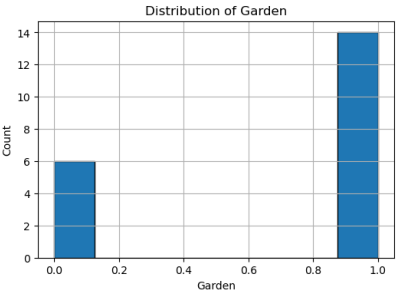
Hidden layer output (8 neurons): [0. 0. 1.9576 0. 0. 3.098 2.5384 0.]

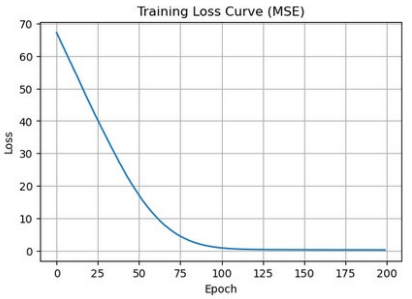
Predicted price (₹) manual: 6,987,956.00

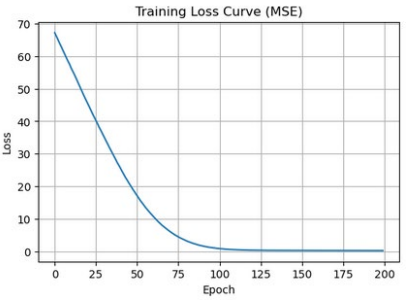

Distribution of Square_Feet
Count
Square_Feet

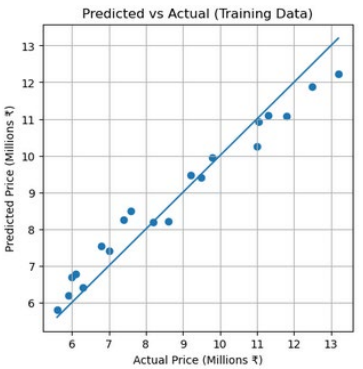

Distribution of Area(0=A,1=B,2=C)
Count
Area(0=A,1=B,2=C)

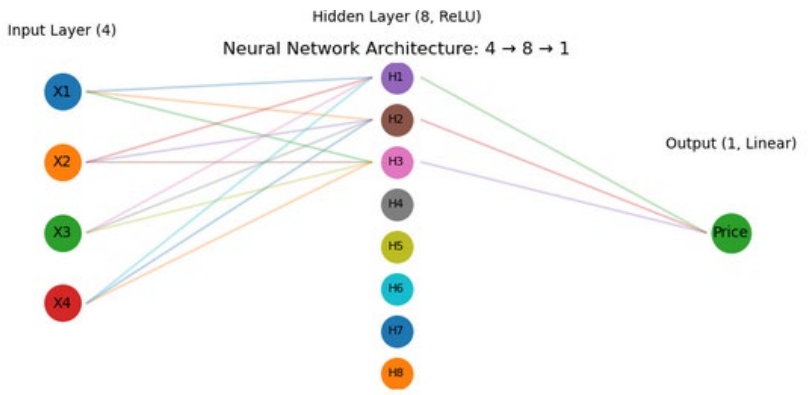

Distribution of BHK
Count
BHK


Distribution of Garden
Count
Garden


Training Loss Curve (MSE)
Loss
Epoch


Training Loss Curve (MSE)
Loss
Epoch


Predicted vs Actual (Training Data)
Predicted Price (Millions ₹)
Actual Price (Millions ₹)



Model saved as: house_price_model.keras

Prediction using loaded model (House 3) ₹: 6,875,491.14

Very Important Explanation

You trained on:

House 1

1000 soft, Area A, 3 BHK, Garden Yes → ₹70,00,000

House 2

800 soft, Area A, 3 BHK, Garden Yes → ₹60,00,000

Now you send unseen:

House 3

850 soft, Area A, 3 BHK, Garden Yes → Price?

What models learn:

House 1 bigger than House 2 → price higher
So model learns soft has positive impact on price.

Now for House 3:

850 soft is between 800 and 1000
So prediction should be between 60L and 70L (approx.)

This prediction happens using the same trained weights and biases.

During prediction:

- no backpropagation
- no weight update
- only forward pass using stored weights

Chapter 8
Hyperparameters That Control Learning

Definition

Hyperparameters are settings YOU choose before training starts. They are not learned from data. They control how fast and how well learning happens.

Syntax / Formula

Key formula (small and important)

```
Key hyperparameters:
  epochs = number of passes over the training data
  batch_size = how many rows processed before one weight update
  learning_rate (lr) = step size for weight updates
  layers/neurons = model capacity (how much it can learn)
  activation = how the network adds non-linearity
```

Why?

Because most beginner failures are not about the model — they are about wrong hyperparameters.

What?

We will see how learning rate and epochs change results using a tiny dataset.

How?

Rule of thumb for beginners:

- Start with small model.
- Use lr = 0.01 for SGD or lr = 0.001 for Adam.
- Watch loss curve: if loss goes up → lr too high.
- If training loss goes down but validation goes up → overfitting.

Let's Code!

```
import numpy as np

import tensorflow as tf


X = np.array([[1],[2],[3],[4],[5],[6],[7],[8],[9],[10]],
dtype=np.float32)

y = np.array([[35],[40],[45],[55],[60],[65],[70],[78],[85],[92]],
dtype=np.float32)


def train_with_lr(lr):

    model = tf.keras.Sequential([tf.keras.layers.Dense(1,
input_shape=(1,))])


model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=lr),
loss="mse")

    h = model.fit(X, y, epochs=200, batch_size=5, verbose=0)

    return h.history["loss"][-1]


for lr in [0.001, 0.01, 0.1]:

    final_loss = train_with_lr(lr)

    print(f"learning_rate={lr} -> final loss={final_loss:.2f}")
```

Output

```
learning_rate=0.001 -> final loss=108.26

learning_rate=0.01 -> final loss=6.61

learning_rate=0.1 -> final loss=nan
```

Example 2 - Hyperparameter Fine-Tuning

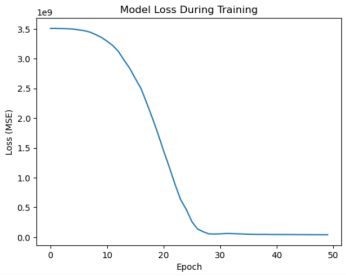
```
import numpy as np

import tensorflow as tf

from tensorflow import keras
```

```
# dataset
X = np.array([[650, 1, 12],[800, 2, 8],[900, 2, 5],[1100, 3, 6],
    [1200, 3, 2],[1500, 3, 1],[1700, 4, 3],[2000, 4, 2],[2400, 4,
1],
    [3000, 5, 4],[3500, 5, 2],[4000, 6, 1],
], dtype=np.float32)

y = np.array([
    18000, 23000, 26000, 32000, 36000, 42000,
    48000, 55000, 65000, 80000, 92000, 110000
], dtype=np.float32)

# Normalize
X_mean = X.mean(axis=0)
X_std = X.std(axis=0) + 1e-8
Xn = (X - X_mean) / X_std

# HYPERPARAMETER TUNING BASED ON RULES OF THUMB:
# 1. Learning rate: Start with 0.001 for Adam
# 2. Batch size: 2-4 for small datasets (avoid 1 - too noisy)
# 3. Architecture: Simple! With 12 examples, avoid complex models
# 4. Epochs: Use EarlyStopping to prevent overfitting

# Build optimal model based on rules
model = keras.Sequential([
    keras.layers.Dense(6, activation="relu", input_shape=(3,)),  #
Reduced from 8
    keras.layers.Dense(3, activation="relu"),  # Reduced from 4
    keras.layers.Dense(1)
])
# Add EarlyStopping callback
early_stopping = keras.callbacks.EarlyStopping(
    monitor='loss',  # Monitor training loss
    patience=50,     # Stop if no improvement for 50 epochs
```

```
    restore_best_weights=True
)
# Use lower learning rate
model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.1),
              loss="mse")
# Train with validation split
history = model.fit(Xn, y,
                    epochs=50,  # Set high, EarlyStopping will stop
earlier
                    batch_size=3,  # 12/4 = 3 batches per epoch
                    verbose=1,
                    callbacks=[early_stopping])
print(f"Training stopped after {len(history.history['loss'])}
epochs")


# Predict
new_house = np.array([[1400, 3, 2]], dtype=np.float32)
new_house_n = (new_house - X_mean) / X_std
pred_rent = model.predict(new_house_n, verbose=0)[0][0]
print(f"Predicted rent (INR): {int(pred_rent)}")


# Visualize training
import matplotlib.pyplot as plt
plt.plot(history.history['loss'])
plt.title('Model Loss During Training')
plt.xlabel('Epoch')
plt.ylabel('Loss (MSE)')
plt.show()
```

Result

```
Epoch 50/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 40286396.0000
Training stopped after 50 epochs
Predicted rent (INR): 38020
```



CHEAT SHEET: What to Change When:

Problem	Solution	Change This
Model predicts same value	Make model smarter	Dense(8, ...) → Dense(16, ...)
Predictions are crazy big/small	Slow down learning	Adam(0.01) → Adam(0.001)
Loss not decreasing	Train longer	epochs=200 → epochs=500
Model too slow	Process more at once	batch_size=4 → batch_size=8
Predicting categories (yes/no)	Change output	loss='mse' → loss='binary_crossentropy'

For Different Problems:

1. For Yes/No Classification:

```
# Only change these 2 lines:
model.add(keras.layers.Dense(1, activation='sigmoid'))  # Last layer
model.compile(loss='binary_crossentropy', metrics=['accuracy'])
```

2. For Multiple Categories (A/B/C):

```
# Only change these 2 lines:
model.add(keras.layers.Dense(3, activation='softmax'))  # 3 = number of categories
model.compile(loss='categorical_crossentropy', metrics=['accuracy'])
```

3. If You Have More Data:

```
# Add one more layer:

model = keras.Sequential([

    keras.layers.Dense(16, activation='relu',
input_shape=(X.shape[1],)),

    keras.layers.Dense(8, activation='relu'),   # ← Added this

    keras.layers.Dense(4, activation='relu'),

    keras.layers.Dense(1)

])
```

Most Important Rule:

Only change ONE thing at a time, then test if it gets better!

Example: Start with this, then make ONE changeVersion 1 (Baseline)

```
neurons = 8

learning rate = 0.01

epochs = 200
```

Test... if bad predictions:

Version 2 (Only changed learning rate)

```
neurons = 8

learning rate = 0.001  # ← ONLY CHANGE

epochs = 200
```

Test... if still bad:

Version 3 (Only changed neurons)

```
neurons = 16           # ← ONLY CHANGE

learning rate = 0.001

epochs = 200
```

Common Hyperparameters in Your Model:

Learning rate (0.01 in your code)

Number of neurons/layers (8, 4 in your code)

Batch size (4 in your code)

Number of epochs (500 in your code)

Activation functions (relu in your code)

Optimizer type (Adam in your code)

Common beginner mistakes

Using a very deep network for a small dataset → overfitting.

Using epochs=5000 without validation → you won't know when to stop.

Changing many hyperparameters at once → you won't know which change helped.

Chapter 9
House Price Prediction with Deep Learning

This chapter shows a full, practical Deep Learning lifecycle using a tiny, hard-coded dataset.
You will see how data becomes numbers, how a neural network predicts a price, how the error is measured, and how weights and biases are updated during training.
By the end, you will not only train a model—you will also save it and reuse it for future predictions like a real production system.

Problem statement (Regression)

We want to predict a house price (a continuous value) using four features:

- Square_Feet (e.g., 850, 1000, 1200)
- Area (encoded as 0, 1, 2)
- BHK (2, 3, 4, 5)
- Garden (0 = No, 1 = Yes)

Target: Price in INR (₹).

1. Dataset: Small, Hard-Coded, and Easy to Inspect

In this learning example, we intentionally keep the dataset small (20 records) so you can inspect it and understand what the model is learning. We generate data using a few "base patterns" and add small random variation. This mimics real-world pricing—similar homes do not always have exactly the same price.

The dataset columns are: Square_Feet, Area, BHK, Garden, Price.

Let's Code: Build the dataset (20 records)

```
import numpy as np
import pandas as pd

np.random.seed(42)

data = []
```

```
base_patterns = [
    {"sqft": 800,  "area": 0, "bhk": 2, "garden": 0, "base_price":
3000000},
    {"sqft": 1000, "area": 0, "bhk": 3, "garden": 1, "base_price":
4500000},
    {"sqft": 1200, "area": 1, "bhk": 3, "garden": 0, "base_price":
5000000},
    {"sqft": 1500, "area": 2, "bhk": 5, "garden": 1, "base_price":
6500000},
]

for i in range(20):
    pattern = base_patterns[i % len(base_patterns)]
    sqft = pattern["sqft"] + np.random.randint(-80, 81)
    area = pattern["area"]
    bhk = pattern["bhk"]
    garden = pattern["garden"]

    # Price increases with square feet; add small random noise
    price = pattern["base_price"] + (sqft - pattern["sqft"]) * 5000
    price += np.random.randint(-200000, 200001)

    data.append([sqft, area, bhk, garden, price])

df = pd.DataFrame(data, columns=["Square_Feet", "Area", "BHK",
"Garden", "Price"])
print(df.head(10))
print("Total Records:", len(df))
```

2. Preprocessing: Normalization (Why It Matters)

Neural networks learn faster when inputs are on a similar scale. One important implementation detail: convert your feature array to float before dividing.
If you keep it as integers, values like 850/1000 can be truncated to 0 and your model will not train properly.

- Square_Feet divided by 1000 (so it becomes around 0.8 to 1.6)
- Area divided by 2 (because area codes are 0..2)
- BHK divided by 5 (assuming max 5 in our data)
- Garden stays 0/1 (already normalized)
- Price scaled to "millions" by dividing by 1,000,000

Let's Code: Normalize features and target (fixed)

```
X = df[["Square_Feet", "Area", "BHK", "Garden"]].values
y = df["Price"].values

# IMPORTANT: convert to float before dividing
Xn = X.astype("float32")

Xn[:, 0] = Xn[:, 0] / 1000   # sqft
Xn[:, 1] = Xn[:, 1] / 2      # area (0..2)
Xn[:, 2] = Xn[:, 2] / 5      # bhk (assume max 5)
# Garden stays 0/1

yn = y.astype("float32") / 1_000_000  # price in millions

print("First 3 normalized rows:")
print(Xn[:3], yn[:3])
```

3. Neural Network Architecture

We build a simple feed-forward neural network:
Input Layer (4 features) → Hidden Layer (8 neurons, ReLU) → Output Layer (1 neuron, Linear).

Let's Code: Build and compile the model

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

model = keras.Sequential([
    layers.Dense(8, activation="relu", input_shape=(4,),
name="hidden_layer"),
    layers.Dense(1, activation="linear", name="output_layer")
])

model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=0.001),
    loss="mse",
    metrics=["mae"]
)

model.summary()
```

4. Training

We split the data for validation and train with early stopping to avoid overfitting.

Let's Code: Train with EarlyStopping

```
from sklearn.model_selection import train_test_split

X_train, X_val, y_train, y_val = train_test_split(
    Xn, yn, test_size=0.2, random_state=42
)

early_stop = keras.callbacks.EarlyStopping(
    monitor="val_loss",
    patience=20,
    restore_best_weights=True
)

history = model.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=100,
    batch_size=4,
    callbacks=[early_stop],
    verbose=1
)
```

5. Inference (Prediction on New Data)

Let's Code: Predict and convert back to INR (fixed)

```
import numpy as np

house = np.array([[850, 0, 3, 1]], dtype="float32")

# Apply the same normalization
house[:, 0] = house[:, 0] / 1000
house[:, 1] = house[:, 1] / 2
house[:, 2] = house[:, 2] / 5

pred_million = float(model.predict(house)[0][0])
pred_inr = pred_million * 1_000_000

print("Predicted price (million):", pred_million)
print("Predicted price (INR):", int(pred_inr))
```

6. Save and Reuse

```
# Save in the new recommended format
model.save("house_price_model.keras")


# Load with custom objects
loaded_model = keras.models.load_model("house_price_model.keras")
pred_million_2 = float(loaded_model.predict(house)[0][0])
print("Loaded model prediction (INR):", int(pred_million_2 *
1_000_000))
```

7. Chapter Summary

- Convert arrays to float before normalization to avoid truncation.
- Keep the same normalization rules for training and inference.
- Dense layers learn by updating weights and biases to reduce loss.
- Save the trained model once; reuse it many times for predictions.

Chapter 10
ANN Regression Example (Real Estate Rent Prediction)

Definition

ANN Regression means output is a number (continuous value) like price, rent, demand, temperature, etc.

Syntax / Formula

Key formula (small and important)

```
Multi-feature neuron:
  z = w1*x1 + w2*x2 + w3*x3 + b
  y_pred = z   (for simple regression)

Loss (MSE):
  (1/N) * Σ (y_true - y_pred)^2
```

Why?

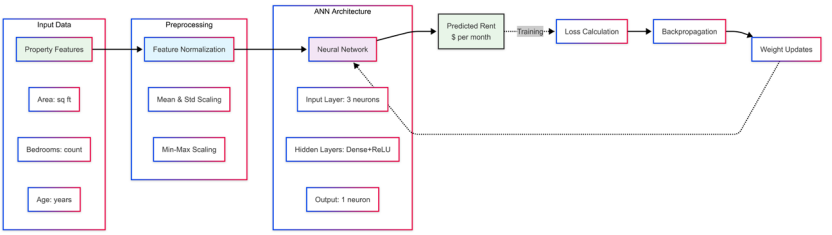
Real estate teams often need quick estimates: rent prediction, price estimation, maintenance cost prediction.

What?

We will predict monthly rent using 3 simple features with 12 hard-coded rows.

How?

1) Prepare tiny data table.
2) Normalize features (important!).
3) Train a small ANN.
4) Predict rent for a new property.



Let's Code!

```
import numpy as np
import tensorflow as tf
from tensorflow import keras


# Tiny real-estate dataset (12 rows)
# Features: [area_sqft, bedrooms, age_years]
X = np.array([[650, 1, 12],[800, 2, 8],[900, 2, 5],[1100, 3, 6],[1200, 3, 2],[1500, 3, 1],
    [1700, 4, 3],[2000, 4, 2],[2400, 4, 1],[3000, 5, 4],[3500, 5, 2],[4000, 6, 1],
], dtype=np.float32)


# Target: monthly rent (INR) - simple realistic-looking numbers
y = np.array([
    18000, 23000, 26000, 32000, 36000, 42000,
    48000, 55000, 65000, 80000, 92000, 110000
], dtype=np.float32)


# ---- Normalize features (mean 0, std 1) ----
X_mean = X.mean(axis=0)
X_std = X.std(axis=0) + 1e-8
Xn = (X - X_mean) / X_std


model = keras.Sequential([
    keras.layers.Dense(8, activation="relu", input_shape=(3,)),
    keras.layers.Dense(4, activation="relu"),
    keras.layers.Dense(1)  # rent
```

```
])


model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.01),
loss="mse")
model.fit(Xn, y, epochs=500, batch_size=4, verbose=0)


# Predict unseen property
new_house = np.array([[1400, 3, 4]], dtype=np.float32)
new_house_n = (new_house - X_mean) / X_std
pred_rent = model.predict(new_house_n, verbose=0)[0][0]
print("Predicted rent (INR):", int(pred_rent))
```

Output

```
Predicted rent (INR): 26473
```

Common beginner mistakes

Not normalizing features (area is thousands but bedrooms is small).

Using too many layers for tiny data → model memorizes.

Expecting high accuracy from 12 rows — goal here is understanding.

Quick exercises

Add one more feature like 'distance_to_metro_km' and retrain.

Change activation from ReLU to tanh and compare.

Try fewer neurons (4,2) and see if results stay similar.

Chapter 11
ANN Classification Example (Banking Default Risk)

Definition

ANN Classification means output is a class/label like YES/NO, fraud/not-fraud, default/not-default.

Syntax / Formula

Key formula (small and important)

```
Sigmoid converts a number into probability:
  p = 1 / (1 + e^(-z))
For binary classification:
  y_pred = p  (probability of class 1)
Loss (Binary Cross Entropy):
  Loss = -( y*log(p) + (1-y)*log(1-p) )
```

Why?

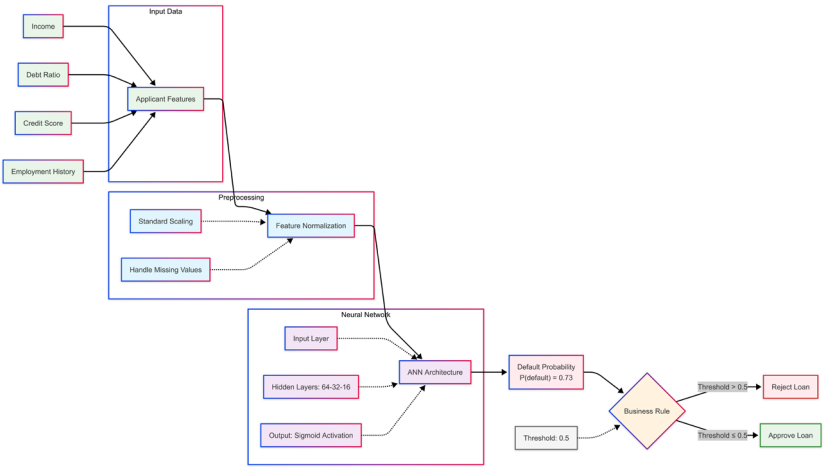
Banks and fintech teams use classification to make risk decisions: approve/reject, fraud/not-fraud, high-risk/low-risk.

What?

We will build a tiny default-risk classifier with hard-coded examples (for learning, not for production).

How?

1) Make a small, labeled dataset.
2) Normalize features.
3) Train a small ANN with sigmoid output.
4) Predict probability for a new applicant.



Let's Code!

```
import numpy as np

import tensorflow as tf

from tensorflow import keras


# Features: [income_lakhs, loan_lakhs, credit_score, late_payments]

X = np.array([[6,  2, 750, 0],[5,  4, 720, 1],[4,  6, 680, 2],[3,
7, 650, 3],[10, 3, 790, 0],

    [8,  8, 700, 2],[12, 5, 800, 0],[2.5,6, 620, 4],[7,  9, 690,
2],[15, 4, 810, 0],

    [9,  12,660, 3],[11, 10,710, 1],[3,  5, 640, 3],[13, 6, 780, 0],

], dtype=np.float32)


# Label: 1 = default risk high, 0 = low

y = np.array([0,0,0,1,0,1,0,1,1,0,1,0,1,0], dtype=np.float32)


# Normalize

mean = X.mean(axis=0)

std = X.std(axis=0) + 1e-8

Xn = (X - mean) / std


model = keras.Sequential([

    keras.layers.Dense(8, activation="relu", input_shape=(4,)),
```

```
    keras.layers.Dense(4, activation="relu"),
    keras.layers.Dense(1, activation="sigmoid")  # probability
])

model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.01),
              loss="binary_crossentropy",
              metrics=["accuracy"])

model.fit(Xn, y, epochs=400, batch_size=4, verbose=0)

# Predict new applicant
new_app = np.array([[6.5, 7, 670, 2]], dtype=np.float32)
new_app_n = (new_app - mean) / std
p = model.predict(new_app_n, verbose=0)[0][0]
print("Default risk probability:", float(p))
print("Decision (threshold 0.5):", "High Risk" if p >= 0.5 else "Low Risk")
```

Output

```
Default risk probability: 0.9965839385986328
Decision (threshold 0.5): High Risk
```

Common beginner mistakes

Using MSE loss for classification (should use binary_crossentropy).

Forgetting sigmoid on final layer for binary classification.

Using too small dataset for real decision-making — this demo is only for learning.

Quick exercises

Change threshold from 0.5 to 0.6 and see decisions change.

Add feature 'job_years' and retrain. Try a bigger network and see if it overfits (accuracy becomes 1.0 quickly).

Chapter 11.1
TensorFlow & Keras - How They Work Inside, and How You Build Real AI Models

TensorFlow vs Keras (one line each)

TensorFlow is the engine: fast tensor math + automatic gradients + training on CPU/GPU.

Keras is the easy interface: layers + models + compile() + fit().

Mental picture:
TensorFlow = **engine**
Keras = **steering wheel**

Deep Learning in One Sentence

Training any AI model is only this loop:

Predict → Measure mistake → Compute gradients → Update weights → Repeat

TensorFlow runs the math.
Keras gives clean APIs to do it.

What a Layer Really Does (the "secret" equation)

A Dense layer is just:

$$Z = XW + b$$

then activation:

$$A = f(Z)$$

- X= your input numbers
- W= weights (learned importance)
- b= bias (adjustment)
- f= activation (ReLU / sigmoid / softmax)

So

```
Dense(8, activation="relu")
means: “Create 8 neurons that compute XW + band apply ReLU.”
```

What compile() and fit() mean (real meaning)

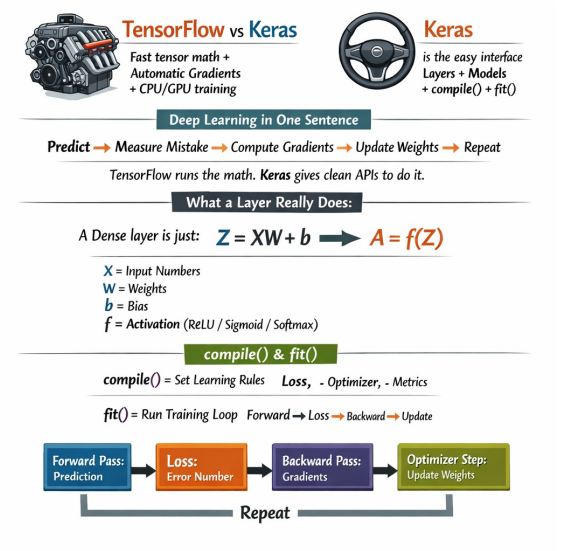
compile() = set learning rules

- **loss**: what counts as error
- **optimizer**: how to update weights
- **metrics**: what to display

fit() = run training loop

For every epoch TensorFlow does:

1. forward pass → prediction
2. loss → error number
3. backward pass → gradients (automatic)
4. optimizer step → update weights
5. repeat



Part A: Example 1 (Regression) - Predict a Number (House Price)

Goal

Input:

- size, bedrooms, age

Output:

- price (lakhs)

Loss:

- MSE

Let's Code!

```
import numpy as np
import tensorflow as tf


# ------------------------------
# 1) HARD-CODED DATA
# X = [size, bedrooms, age]
# y = price (lakhs)
# ------------------------------
```

```
X = np.array([
    [1000, 2, 10],
    [ 850, 2, 12],
    [1200, 3,  8],
    [ 700, 2, 20],
    [1500, 3,  5],
    [2000, 4,  2]
], dtype=np.float32)

y = np.array([[75],[60],[95],[40],[140],[220]], dtype=np.float32)

# ------------------------------
# 2) TRAIN/TEST SPLIT
# ------------------------------
X_train, y_train = X[:4], y[:4]
X_test,  y_test  = X[4:], y[4:]

# ------------------------------
# 3) STATS: STANDARDIZATION
# x' = (x - mean) / std
# train stats only
# ------------------------------
mean = X_train.mean(axis=0)
std  = X_train.std(axis=0) + 1e-7
X_train_s = (X_train - mean) / std
X_test_s  = (X_test  - mean) / std

# ------------------------------
# 4) MODEL
# ------------------------------
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(3,)),
    tf.keras.layers.Dense(8, activation="relu"),
    tf.keras.layers.Dense(4, activation="relu"),
    tf.keras.layers.Dense(1)  # regression output
```

```
])


# ----------------------------
# 5) COMPILE + TRAIN
# ----------------------------
model.compile(optimizer=tf.keras.optimizers.Adam(0.01), loss="mse",
metrics=["mae"])
model.fit(X_train_s, y_train, epochs=300, verbose=0)


# ----------------------------
# 6) EVALUATE + PREDICT
# ----------------------------
test_loss, test_mae = model.evaluate(X_test_s, y_test, verbose=0)
print("Regression Test MAE (lakhs):", round(float(test_mae), 2))


new_house = np.array([[1100, 3, 9]], dtype=np.float32)
new_house_s = (new_house - mean) / std
pred = model.predict(new_house_s, verbose=0)
print("Predicted price (lakhs):", round(float(pred[0][0]), 2))
```

output

```
Regression Test MAE (lakhs): 8.42
Predicted price (lakhs): 81.09
```

What happened inside?

- Dense layers computed $\boldsymbol{XW + b}$repeatedly
- Loss (MSE) measured mistake
- TensorFlow computed gradients automatically
- Adam updated weights
- After many epochs, predictions improved

Part B: Example 2 (Classification) Predict Yes/No (Loan Default)

Goal

Input:

- income (lakhs/year)
- credit score (0–900)
- missed payments (count)

Output:

- default? 0 = No, 1 = Yes

We will use:

- Sigmoid output (probability 0–1)
- Binary Cross entropy loss (BCE)

Let's Code!

```
import numpy as np
import tensorflow as tf

# ------------------------------------------------------
# 1) HARD-CODED DATA
# X = [income_lakhs, credit_score, missed_payments]
# y = default (0/1)
# ------------------------------------------------------
X = np.array([
    [12, 780, 0],  # strong profile -> no default
    [10, 720, 1],  # mostly good
    [ 8, 680, 1],
    [ 6, 620, 2],
    [ 5, 590, 3],  # risky -> default
    [ 4, 560, 4],  # risky
], dtype=np.float32)

y = np.array([[0],[0],[0],[1],[1],[1]], dtype=np.float32)
```

```
# ------------------------------------------------------
# 2) TRAIN/TEST SPLIT
# ------------------------------------------------------
X_train, y_train = X[:4], y[:4]
X_test,  y_test  = X[4:], y[4:]

# ------------------------------------------------------
# 3) STATS: STANDARDIZATION (best practice)
# ------------------------------------------------------
mean = X_train.mean(axis=0)
std  = X_train.std(axis=0) + 1e-7

X_train_s = (X_train - mean) / std
X_test_s  = (X_test  - mean) / std

# ------------------------------------------------------
# 4) MODEL (Sigmoid output)
# ------------------------------------------------------
clf = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(3,)),
    tf.keras.layers.Dense(8, activation="relu"),
    tf.keras.layers.Dense(1, activation="sigmoid")  # probability
])

# ------------------------------------------------------
# 5) COMPILE + TRAIN
# loss = binary crossentropy (BCE)
# ------------------------------------------------------
clf.compile(
    optimizer=tf.keras.optimizers.Adam(0.01),
    loss="binary_crossentropy",
    metrics=["accuracy"]
)

clf.fit(X_train_s, y_train, epochs=300, verbose=0)
```

```
# ------------------------------------------------------
# 6) EVALUATE + PREDICT
# ------------------------------------------------------
loss, acc = clf.evaluate(X_test_s, y_test, verbose=0)
print("\nClassification Test Accuracy:", round(float(acc), 2))

# Predict probability for new customer
new_customer = np.array([[7, 650, 2]], dtype=np.float32)
new_customer_s = (new_customer - mean) / std

prob = clf.predict(new_customer_s, verbose=0)[0][0]
prediction = 1 if prob >= 0.5 else 0

print("Default probability:", round(float(prob), 3))
print("Predicted class (0=No,1=Yes):", prediction)

output

Classification Test Accuracy: 1.0
Default probability: 0.992
Predicted class (0=No,1=Yes): 1
```

What Sigmoid Means (super simple)

Sigmoid converts any number into **0 to 1**:

$$sigmoid(z) = \frac{1}{1 + e^{-z}}$$

So output like:

- 0.90 = very high chance of default
- 0.10 = very low chance of default

How to Choose Output + Loss (Golden Rule)

Regression (number)

- Output: Dense(1)
- Loss: mse

Binary classification (yes/no)

- Output: Dense(1, activation="sigmoid")
- Loss: binary_crossentropy

Multi-class (0/1/2/3…)

- Output: Dense(num_classes, activation="softmax")
- Loss: categorical_crossentropy

Common mistakes (fast fixes)

1. **Wrong input shape**
   - If X has 3 features → Input(shape=(3,))
2. **No scaling**
   - Training becomes unstable or slow
3. **Wrong loss**
   - BCE for regression = wrong
   - MSE for classification = wrong
4. **Learning rate too high**
   - Loss becomes NaN
   - Try 0.001 or 0.01

Mini exercises (best learning)

1. Change learning rate 0.01 → 0.001 and compare results
2. Remove scaling and observe accuracy/loss
3. For classification, change threshold 0.5 → 0.7 and see difference
4. Add 2 more hardcoded records to each dataset and retrain

See the "Learning" with Your Own Eyes-Print Weights (W) and Bias (b)

Why this is powerful

Many readers believe training is magic.
This section proves training is real by showing:

- weights and bias values **change** after training
- that change is exactly what "learning" means

Remember the core layer equation:

$$Z = XW + b$$

If W and b become better, predictions become better.

Regression: Print Weights Before vs After Training (Hardcoded)

Let's Code! (Add-on)

```
import numpy as np
import tensorflow as tf


# ------------------------------
# HARD-CODED DATA (Regression)
X = np.array([
    [1000, 2, 10],
    [ 850, 2, 12],
    [1200, 3,  8],
    [ 700, 2, 20],
    [1500, 3,  5],
    [2000, 4,  2]
], dtype=np.float32)

y = np.array([[75],[60],[95],[40],[140],[220]], dtype=np.float32)
# Train/test split
X_train, y_train = X[:4], y[:4]
# Standardize using train stats
mean = X_train.mean(axis=0)
```

```
std  = X_train.std(axis=0) + 1e-7
X_train_s = (X_train - mean) / std
# Model
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(3,)),
    tf.keras.layers.Dense(8, activation="relu", name="hidden"),
    tf.keras.layers.Dense(1, name="output")
])

model.compile(optimizer=tf.keras.optimizers.Adam(0.01), loss="mse")
# ----------------------------
# PRINT WEIGHTS BEFORE TRAINING
print("=== BEFORE TRAINING ===")
for layer in model.layers:
    w = layer.get_weights()
    if w:
        W, b = w
        print(f"{layer.name}: W shape={W.shape}, b shape={b.shape}")
        print("  W first row:", np.round(W[0], 4))
        print("  b:", np.round(b, 4))

# Train
model.fit(X_train_s, y_train, epochs=300, verbose=0)
# ----------------------------
# PRINT WEIGHTS AFTER TRAINING
print("\n=== AFTER TRAINING ===")
for layer in model.layers:
    w = layer.get_weights()
    if w:
        W, b = w
        print(f"{layer.name}: W shape={W.shape}, b shape={b.shape}")
        print("  W first row:", np.round(W[0], 4))
        print("  b:", np.round(b, 4))
Output === BEFORE TRAINING ===
hidden: W shape=(3, 8), b shape=(8,)
```

```
  W first row: [ 0.5161 -0.596   0.6077 -0.6952  0.4776  0.1892 -
0.2801  0.6913]
  b: [0. 0. 0. 0. 0. 0. 0. 0.]
output: W shape=(8, 1), b shape=(1,)
  W first row: [-0.6199]
  b: [0.]
=== AFTER TRAINING ===
hidden: W shape=(3, 8), b shape=(8,)
W first row: [ 0.3251 -3.8997  0.3809  1.2185  2.4728  2.3767 -
0.4717  0.4947]
b: [-0.4567  3.7378 -0.2295  3.0729  3.0254  3.4792 -0.2703 -0.2675]
output: W shape=(8, 1), b shape=(1,)
 W first row: [-0.2704], b: [1.9869]
```

Classification: Print Weights + Show Probability Change (Hardcoded)

Here we do the same for the sigmoid model and show probability output.

Let's Code! (Add-on)

```
import numpy as np
import tensorflow as tf


# --------------------------------
# HARD-CODED DATA (Classification)
# --------------------------------
X = np.array([
    [12, 780, 0],
    [10, 720, 1],
    [ 8, 680, 1],
    [ 6, 620, 2],
    [ 5, 590, 3],
    [ 4, 560, 4],
], dtype=np.float32)


y = np.array([[0],[0],[0],[1],[1],[1]], dtype=np.float32)
```

```
# Train part only
X_train, y_train = X[:4], y[:4]

# Standardize
mean = X_train.mean(axis=0)
std  = X_train.std(axis=0) + 1e-7
X_train_s = (X_train - mean) / std

# New customer to test probability
new_customer = np.array([[7, 650, 2]], dtype=np.float32)
new_customer_s = (new_customer - mean) / std

# Model
clf = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(3,)),
    tf.keras.layers.Dense(8, activation="relu", name="hidden"),
    tf.keras.layers.Dense(1, activation="sigmoid", name="output")
])

clf.compile(optimizer=tf.keras.optimizers.Adam(0.01),
            loss="binary_crossentropy",
            metrics=["accuracy"])

# ---------------------------------------
# PROBABILITY BEFORE TRAINING (random)
# ---------------------------------------
prob_before = clf.predict(new_customer_s, verbose=0)[0][0]
print("Probability BEFORE training:", round(float(prob_before), 3))

# Print weights before training
print("\n=== BEFORE TRAINING WEIGHTS ===")
for layer in clf.layers:
    w = layer.get_weights()
    if w:
        W, b = w
```

```
        print(f"{layer.name}: W mean={W.mean():.4f}, b
mean={b.mean():.4f}")


# Train
clf.fit(X_train_s, y_train, epochs=300, verbose=0)


# ---------------------------------------
# PROBABILITY AFTER TRAINING (learned)
# ---------------------------------------
prob_after = clf.predict(new_customer_s, verbose=0)[0][0]
print("\nProbability AFTER training:", round(float(prob_after), 3))


# Print weights after training
print("\n=== AFTER TRAINING WEIGHTS ===")
for layer in clf.layers:
    w = layer.get_weights()
    if w:
        W, b = w
        print(f"{layer.name}: W mean={W.mean():.4f}, b
mean={b.mean():.4f}")


# Final class
pred_class = 1 if prob_after >= 0.5 else 0
print("\nPredicted class (0=No default, 1=Default):", pred_class)
```

output

Probability BEFORE training: 0.15

=== BEFORE TRAINING WEIGHTS ===

hidden: W mean=0.0065, b mean=0.0000

output: W mean=-0.2084, b mean=0.0000

Probability AFTER training: 0.996

=== AFTER TRAINING WEIGHTS ===

hidden: W mean=0.0597, b mean=0.2385

output: W mean=-0.2364, b mean=-1.0425

Predicted class (0=No default, 1=Default): 1

Mini Summary (Very Important Line for Readers)

A model does not "learn facts."
A model learns by **adjusting weights (W) and bias (b)** so that loss becomes smaller.

That's why TensorFlow and Keras exist:

- to manage tensors
- to calculate gradients
- to update weights
- to automate the training loop

Chapter 12
CNN (Convolutional Neural Network) with MNIST

Definition

CNN (Convolutional Neural Network) is a neural network designed for images. Instead of treating an image as a long list of numbers, CNN learns small patterns like edges, corners, and shapes.

Syntax / Formula

Key formula (small and important)

```
Convolution (simple idea):
  feature_map[i,j] = Σ Σ ( image[i+a, j+b] * kernel[a,b] )

Meaning: slide a small filter (kernel) over the image and
compute weighted sums.
ReLU keeps positive signals. Pooling reduces size and keeps
important info.
```

Why?

When we look at an image, we don't read it pixel by pixel like a long list. We first notice edges, then shapes, then parts (eyes, wheels), and finally the object (face, car).

A normal fully connected neural network forces you to flatten the image into a long vector. Example: a 32×32 grayscale image = 1024 pixels. Flattening destroys the idea of "this pixel is near that pixel." So the network loses spatial meaning.

CNNs solve this problem because they:

Preserve spatial structure (nearby pixels stay nearby)

Use far fewer trainable weights using weight sharing

Automatically learn features like edges → textures → parts → objects

Work even if the object moves slightly in the image (shift/translation)

That's why CNNs are the #1 backbone for computer vision (digits, faces, objects, medical scans) and are also used in NLP (1D CNNs) and video (3D CNNs).

What?

A Convolutional Neural Network (CNN) is a deep learning model designed mainly for images (and any grid-like data). It learns features by sliding small windows (filters) over the input.

A typical CNN is built from 3 main layer types:

1. Convolution Layer (Conv) → *feature extraction*
2. Pooling Layer (Pool) → *down sampling / compression*
3. Fully Connected Layer (Dense) → *final decision / classification*

Most CNNs look like this pattern:

[Conv → ReLU] → [Conv → ReLU] → [Pool] → (repeat) → [Flatten] → [Dense] → Output

Core Building Blocks

1) Convolution Layer (Conv)

Definition:
A convolution layer uses a small matrix called a **filter / kernel** (like 3×3 or 5×5) and slides it over the image to detect a specific feature (edge, curve, texture).

Key terms (simple meanings)

- Kernel / Filter: a small "feature detector" (like an edge detector)
- Receptive field (patch): the small area of the input the filter looks at one time
- Stride: how many pixels the filter jumps each step (1 means move 1 pixel)
- Padding: adding extra border pixels (usually zeros) so output size doesn't shrink too fast
- Feature map: the output image produced by one filter

What convolution is doing (math idea)

At each position, the filter multiplies with the input patch and adds them:

$$\text{output}(i,j) = \sum_u \sum_v (\text{input}(i+u, j+v) \times \text{kernel}(u,v)) + b$$

Then we apply an activation function (usually **ReLU**):

$$\text{ReLU}(x) = \max(0, x)$$

Output size formula (very useful)

For input width/height N, filter size F, padding P, stride S:

$$\text{Output size} = \left\lfloor \frac{N - F + 2P}{S} \right\rfloor + 1$$

Trainable parameters in Conv (important!)

If input has Cchannels (C=1 for grayscale, C=3 for RGB), and we use Kfilters:

$$\text{Params} = K \times (F \times F \times C + 1)$$

2) Pooling Layer (Pool)

Definition:
Pooling downsamples the feature map (makes it smaller), keeping important information.

Common pooling types:

- Max Pooling: keeps the maximum value in each small window (most common)
- Average Pooling: keeps the average

Example: **2×2 MaxPool with stride 2**
It reduces width and height by half.

Why pooling helps:

- Reduces computation
- Reduces overfitting (somewhat)

- Adds small translation tolerance

3) Fully Connected Layer (Dense)

After convolution + pooling, we have learned features, but they are still in "image-like" form.

So we:

- Flatten feature maps into a long vector
- Feed into Dense layers for final prediction (like a normal neural network)

Output layer depends on task:

- **Binary classification:** 1 neuron + sigmoid
- **Multi-class (10 classes):** 10 neurons + softmax

Softmax:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

How?

1) Load MNIST images.
2) Normalize pixel values to 0–1.
3) Build CNN: Conv → ReLU → Pool → Conv → Pool → Flatten → Dense.
4) Train and test accuracy.

Worked Example (32×32 grayscale, simple numbers)

Input

A grayscale image: $32 \times 32 \times 1$

Step 1: Convolution Layer

- Filters: 10
- Filter size: 5×5
- Stride: 1
- Padding: 0 (valid)

Output size: $(32 - 5 + 0)/1 + 1 = 28$

So output feature maps: **28 × 28 × 10**

Trainable parameters:

- Each filter has $5 \times 5 \times 1 + 1 = 26$parameters
- Total = $26 \times 10 = 260$

Notice something important:
Even though the convolution produces **28×28×10 = 7840 activations**, the trainable parameters are only **260** because weights are shared.

Step 2: Pooling Layer

- Pool size: **2×2**
- Stride: **2**
  Input: **28×28×10**
  Output: **14×14×10**

Step 3: Fully Connected Layer

Flatten: $14 \times 14 \times 10 = 1960$

Dense layer with **200 neurons**:

- Params = $1960 \times 200 + 200 = 392{,}200$

Then final output layer (example: 10 classes):

- Params = $200 \times 10 + 10 = 2{,}010$

CNN Best Practices (Practical Rules)

Receptive field (filter) size

- Small filters work best: 3×3 is the default today
- Larger images sometimes use 5×5 or 7×7 early

Stride

- Start with stride = 1 (easy + good accuracy)
- Use stride 2 sometimes to down sample instead of pooling

Padding

- "same" keeps the output size same as input (helps deeper networks)
- "valid" shrinks output (no padding)

Number of filters

- Early layers: 16, 32
- Deeper layers: 64, 128, 256 (more abstract features need more channels)

Pooling

- Typical: 2×2 MaxPool with stride 2
- But modern CNNs sometimes replace pooling with stride 2 convolutions

Overfitting control

- Use **Dropout**, especially near Dense layers
- Use Data augmentation for images
- Consider Batch Normalization for stability

Data preparation

- Normalize pixels: X = X / 255.0
- Keep consistent image size (resize if needed)

What is the kernel shape? (Your feature-map explanation, simplified)

In Keras Conv2D, kernel shape is:

$$(kH, kW, \text{inChannels}, \text{outChannels})$$

So for (3,3) with RGB input and 32 filters:

- Kernel shape = **(3, 3, 3, 32)**

Meaning:

- For each output feature map, the filter has separate weights for R, G, and B channels.

How CNN Weights Learn (Backprop Intuition in Simple Words)

Why this matters

Many people understand "CNN detects edges," but the real question is:

How does the CNN learn the best filter values (weights)?

CNN learns filters automatically by:

- making a prediction,
- measuring error (loss),
- and updating filter weights little by little.

What gets learned in a CNN?

In a CNN, the trainable parameters are:

- Filter weights (kernels) in Conv layers
- Bias in Conv layers
- Dense layer weights and biases at the end

A convolution filter is like a tiny matrix of numbers, for example 3×3:

$$K = \begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{bmatrix}$$

k11k21k31k12k22k32k13k23k33

These $k$values are learned during training.

How forward pass works (what model does first)

For every small patch of the image, CNN does:

$$z = \sum(\text{patch pixels} \times \text{kernel weights}) + b$$

Then activation (ReLU):

$$a = \max(0, z)$$

This is repeated across the image to create a **feature map**.

How the error is measured (Loss)

If it's classification, loss is usually **cross-entropy**.

Think simply:

- prediction is wrong → loss is high
- prediction is correct → loss is low

CNN training goal:
Reduce loss

How backprop updates Conv filter weights (core idea)

After loss is calculated, training does:

1. **Find which weights caused the error** (gradient)
2. **Reduce those weights in the right direction** (update) Update rule (same as other neural networks):

$$w \leftarrow w - \eta \cdot \frac{\partial Loss}{\partial w}$$

Where:

- w= any kernel weight (like k_{11})
- η= learning rate (small step size)
- $\frac{\partial Loss}{\partial w}$= gradient (how much that weight contributed to error)

The most important CNN learning concept: Weight Sharing

In CNN, the same filter weights are reused across the whole image.

So one filter weight (example k_{11}) affects:

- top-left patch
- center patch
- bottom patch
- everywhere the filter slides

That means when backprop happens:

- gradients from all positions are combined (summed/averaged),
- then that one weight is updated once.

Simple explanation:

One filter learns one skill (like detecting an edge). It practices that skill at many locations in the image. So learning becomes faster and more general.

Where do these learned weights get stored?

After training:

- weights are stored inside the model layers:
  - Conv layer has its kernel weights
  - Dense layer has its weight matrix

In Keras you can see weights:

```
w, b = model.layers[0].get_weights()
print(w.shape, b.shape)
```

And save the trained model:

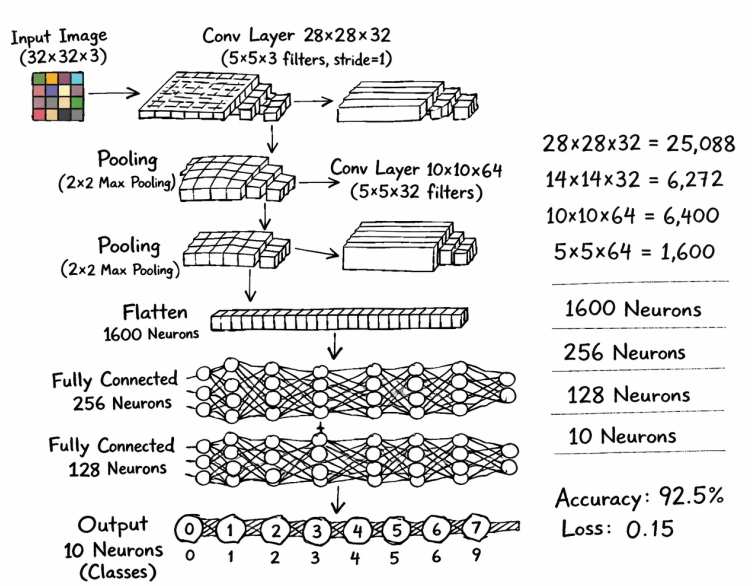
```
model.save("cnn_model.keras")
```

Later, for unseen images, CNN uses the **same saved weights** to do forward pass and predict.

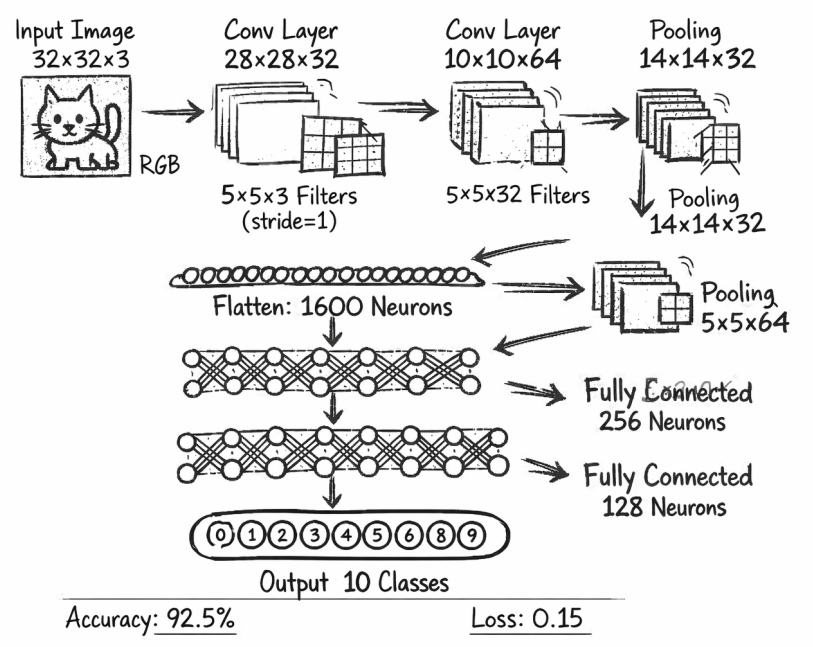
Mini "One-Line Memory"

- **ANN** = good for numbers and tables
- **CNN** = good for images because it learns features using **small filters + shared weights**

Example 1



Example 2



A tiny "manual" convolution to understand the sliding window (NumPy)

This is not for accuracy. This is for **understanding** what convolution does.

```
import numpy as np

# 5x5 grayscale "image"
img = np.array([
    [0, 0, 0, 0, 0],
    [0, 10, 10, 10, 0],
    [0, 10, 50, 10, 0],
    [0, 10, 10, 10, 0],
    [0, 0, 0, 0, 0]
], dtype=float)

# 3x3 edge-like filter (example)
kernel = np.array([
    [-1, -1, -1],
    [ 0,  0,  0],
    [ 1,  1,  1]
], dtype=float)

def conv2d_valid(image, kernel):
    H, W = image.shape
    kH, kW = kernel.shape
    outH = H - kH + 1
    outW = W - kW + 1
    out = np.zeros((outH, outW))

    for i in range(outH):
        for j in range(outW):
            patch = image[i:i+kH, j:j+kW]
            out[i, j] = np.sum(patch * kernel)  # dot product
    return out
```

```
out = conv2d_valid(img, kernel)
print("Input:\n", img)
print("\nKernel:\n", kernel)
print("\nFeature map output:\n", out)
```

output

Input:

[[[0. 0. 0. 0. 0.]
 [0. 10. 10. 10. 0.]
 [0. 10. 50. 10. 0.]
 [0. 10. 10. 10. 0.]
 [0. 0. 0. 0. 0.]]]

Kernel:

[[[-1. -1. -1.]
 [0. 0. 0.]
 [1. 1. 1.]]]

Feature map output:

[[[60. 70. 60.]
 [0.   0.   0.]
 [-60. -70. -60.]]]

Let's Code!

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

# Load MNIST
```

```
(x_train, y_train), (x_test, y_test) =
keras.datasets.mnist.load_data()


# Normalize to 0-1 and add channel dimension (28,28,1)
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0
x_train = x_train[..., None]  # Add channel dimension
x_test = x_test[..., None]


# Use a small subset for faster training
x_train_small = x_train[:10000]
y_train_small = y_train[:10000]


# Build the model
model = keras.Sequential([
    keras.layers.Conv2D(16, (3,3), activation="relu",
input_shape=(28,28,1)),
    keras.layers.MaxPool2D((2,2)),
    keras.layers.Conv2D(32, (3,3), activation="relu"),
    keras.layers.MaxPool2D((2,2)),
    keras.layers.Flatten(),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])


model.compile(optimizer="adam",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])


# Train the model
print("Training the model...")
model.fit(x_train_small, y_train_small, epochs=2,
batch_size=64, validation_split=0.1)


# Evaluate on test set
```

```
test_loss, test_acc = model.evaluate(x_test, y_test,
verbose=0)
print(f"\nTest accuracy: {test_acc:.4f}")


# ========================================
# NEW: Function to test unseen records
# ========================================


def test_unseen_record(model, image_index=None,
custom_image=None):
    """
    Test an unseen record (either from test set or custom
image)


    Args:
        model: Trained CNN model
        image_index: Index of image from test set (0-9999)
        custom_image: Custom 28x28 numpy array (0-255 or 0-1
normalized)
    """


    if custom_image is not None:
        # Handle custom image
        if custom_image.max() > 1.0:
            # Assume 0-255 range, normalize to 0-1
            image = custom_image.astype("float32") / 255.0
        else:
            image = custom_image.astype("float32")


        # Ensure correct shape (28, 28, 1)
        if len(image.shape) == 2:
            image = image[..., None]
        elif len(image.shape) == 3 and image.shape[2] != 1:
            # Convert RGB to grayscale if needed
            image = np.mean(image, axis=2, keepdims=True)
```

```
        # Add batch dimension
        image_batch = np.expand_dims(image, axis=0)
        true_label = "Custom image (no true label)"

    elif image_index is not None:
        # Use image from test set
        if image_index < 0 or image_index >= len(x_test):
            print(f"Error: image_index must be between 0 and
{len(x_test)-1}")
            return

        image = x_test[image_index]
        image_batch = np.expand_dims(image, axis=0)
        true_label = y_test[image_index]

    else:
        print("Error: Either image_index or custom_image must
be provided")
        return

    # Make prediction
    predictions = model.predict(image_batch, verbose=0)
    predicted_label = np.argmax(predictions[0])
    confidence = np.max(predictions[0]) * 100

    # Display results
    print("\n" + "="*50)
    print("PREDICTION RESULTS")
    print("="*50)

    if custom_image is None:
        print(f"Test image index: {image_index}")
        print(f"True label: {true_label}")
```

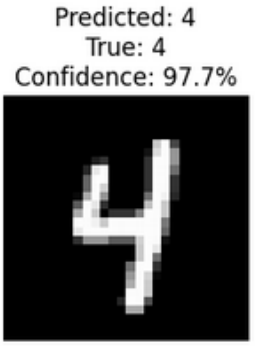
```
print(f"Predicted label: {predicted_label}")
print(f"Confidence: {confidence:.2f}%")
print("\nPrediction probabilities:")
for i, prob in enumerate(predictions[0]):
    print(f"  Digit {i}: {prob*100:6.2f}%")

# Visualize the image
plt.figure(figsize=(6, 3))

# Show image
plt.subplot(1, 2, 1)
if custom_image is not None:
    plt.imshow(custom_image.squeeze(), cmap='gray')
else:
    plt.imshow(x_test[image_index].squeeze(), cmap='gray')

title = f"Predicted: {predicted_label}"
if custom_image is None:
    title += f"\nTrue: {true_label}"
title += f"\nConfidence: {confidence:.1f}%"
plt.title(title)
plt.axis('off')

# Show prediction probabilities as bar chart
plt.subplot(1, 2, 2)
bars = plt.bar(range(10), predictions[0] * 100)
bars[predicted_label].set_color('red')
plt.xlabel('Digit')
plt.ylabel('Probability (%)')
plt.title('Prediction Probabilities')
plt.xticks(range(10))
plt.ylim(0, 100)
```

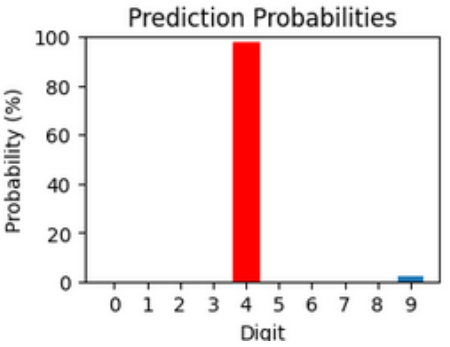
```
    plt.tight_layout()
    plt.show()


# Example 1: Test specific images from test set

print("\n" + "="*50)
print("TESTING UNSEEN RECORDS FROM TEST SET")
print("="*50)

# Test a few specific images
test_indices = [0, 123, 456, 789, 42]  # You can change these
for idx in test_indices:
    test_unseen_record(model, image_index=idx)

# Example 2: Test random images

print("\n" + "="*50)
print("TESTING RANDOM UNSEEN RECORDS")
print("="*50)

# Test 3 random images
for i in range(3):
    random_idx = np.random.randint(0, len(x_test))
    test_unseen_record(model, image_index=random_idx)

# Example 3: Test custom image

print("\n" + "="*50)
print("TESTING CUSTOM IMAGE")
print("="*50)
```

```
# Create a simple custom "7" image
# You can replace this with your own 28x28 image
custom_digit = np.zeros((28, 28))
# Draw a simple "7"
custom_digit[5:8, 8:20] = 1.0  # Horizontal line
for i in range(15):
    custom_digit[8+i, 18-i] = 1.0  # Diagonal line

# Test the custom image
test_unseen_record(model, custom_image=custom_digit)

# Interactive testing loop

print("\n" + "="*50)
print("INTERACTIVE TESTING")
print("="*50)
print("Enter image indices from test set (0-9999)")
print("Type 'r' for random, 'c' for custom, or 'q' to quit")

while True:
    user_input = input("\nEnter image index (0-9999) or
command: ").strip().lower()

    if user_input == 'q':
        print("Exiting...")
        break

    elif user_input == 'r':
        random_idx = np.random.randint(0, len(x_test))
        test_unseen_record(model, image_index=random_idx)

    elif user_input == 'c':
        # Create another custom digit (a simple "4")
        custom_img = np.zeros((28, 28))
```

```
        # Vertical line
        custom_img[5:20, 10:13] = 1.0
        # Horizontal line
        custom_img[15:18, 5:15] = 1.0
        # Diagonal
        for i in range(10):
            custom_img[5+i, 15-i] = 1.0

        test_unseen_record(model, custom_image=custom_img)

    else:
        try:
            idx = int(user_input)
            if 0 <= idx < len(x_test):
                test_unseen_record(model, image_index=idx)
            else:
                print(f"Please enter a number between 0 and
{len(x_test)-1}")
        except ValueError:
            print("Invalid input. Please enter a number or
command (r/c/q)")
```

```
==================================================
PREDICTION RESULTS
==================================================
Test image index: 5774
True label: 4
Predicted label: 4
Confidence: 97.74%

Prediction probabilities:
  Digit 0:   0.00%
  Digit 1:   0.00%
  Digit 2:   0.00%
  Digit 3:   0.00%
  Digit 4:  97.74%
  Digit 5:   0.00%
  Digit 6:   0.00%
  Digit 7:   0.04%
  Digit 8:   0.02%
  Digit 9:   2.20%
```





Common beginner mistakes

- Not normalizing pixel values (0–255) → training becomes slow/unreliable.
- Forgetting channel dimensions (28,28,1).
- Using too many filters/layers on MNIST and overcomplicating learning.

Chapter 13
RNN/LSTM (Sequence Learning) with a Tiny Example

Definition

RNN (Recurrent Neural Network) is built for sequences: time series, text, speech. It remembers previous steps using a hidden state. LSTM is a special RNN that remembers longer patterns.

Syntax / Formula

Key formula (small and important)

```
RNN idea (simplified):
  h_t = activation(Wx*x_t + Wh*h_{t-1} + b)
  y_t = Wy*h_t

h_t is the memory (hidden state). LSTM improves memory using
gates (forget/input/output).
```

Why?

Some problems are not just "one row → one answer."

They are sequences, where the order matters.

Examples:

- A sentence: "I am not happy" (the word not changes meaning)
- Bank transactions: a pattern over days can indicate fraud
- Sensor readings: temperature over time predicts machine failure
- Stock prices: yesterday affects today (not perfectly, but it matters)

A normal ANN treats inputs as independent. It does not remember what came before.

So we need a model that can say:

"I will process input step-by-step and keep a memory of the past."

That is exactly what RNN and LSTM do.

What?

We will train an LSTM to predict the next number from a short sequence (easy to understand).

Definition (RNN)

A **Recurrent Neural Network (RNN)** is a neural network designed for **sequence data**.
It processes one time step at a time and maintains a **hidden state** (memory) that carries information forward.

Simple meaning:

- Hidden state = "what I have understood so far"

Definition (LSTM)

An **LSTM (Long Short-Term Memory)** network is a special kind of RNN that can remember information for a longer time using **gates**.

RNNs forget easily (vanishing gradients). LSTMs solve this with gated memory.

Building Blocks (RNN vs LSTM)

1) RNN building block

At each time step t:

- input = x_t
- previous memory = h_{t-1}
- new memory = h_t

RNN Math

RNN hidden state update

$$h_t = \tanh\left(W_x x_t + W_h h_{t-1} + b\right)$$

- x_t= input at time step t (example: transaction amount today)
- h_{t-1}= previous memory

- W_x, W_h, b= weights and bias (learned)
- tanh = keeps values between -1 and 1

Output (optional)

$$y_t = W_y h_t + c$$

Why RNN struggles (Simple explanation)

In long sequences, RNN must send learning signal from the end back to the start (backprop through time).

In long sequences, gradients become very small → **vanishing gradient**. So early words/time steps are not learned properly.

That's why LSTM was created.

LSTM — The Memory With Gates

What are "gates"?

A gate is a small neural network decision: how much to keep / forget / add.

LSTM maintains:

- **Cell state** C_t= long-term memory
- **Hidden state** h_t= short-term output memory

LSTM gates (main equations)

1) **Forget gate** (what to forget)

$$f_t = \sigma(W_f[x_t, h_{t-1}] + b_f)$$

2) **Input gate** (what new info to add)

$$i_t = \sigma(W_i[x_t, h_{t-1}] + b_i)$$

3) **Candidate memory** (new candidate content)

$$\widetilde{C_t} = \tanh(W_c[x_t, h_{t-1}] + b_c)$$

4) **Update cell state**

$$C_t = f_t \odot C_{t-1} + i_t \odot \widetilde{C_t}$$

5) Output gate + hidden state

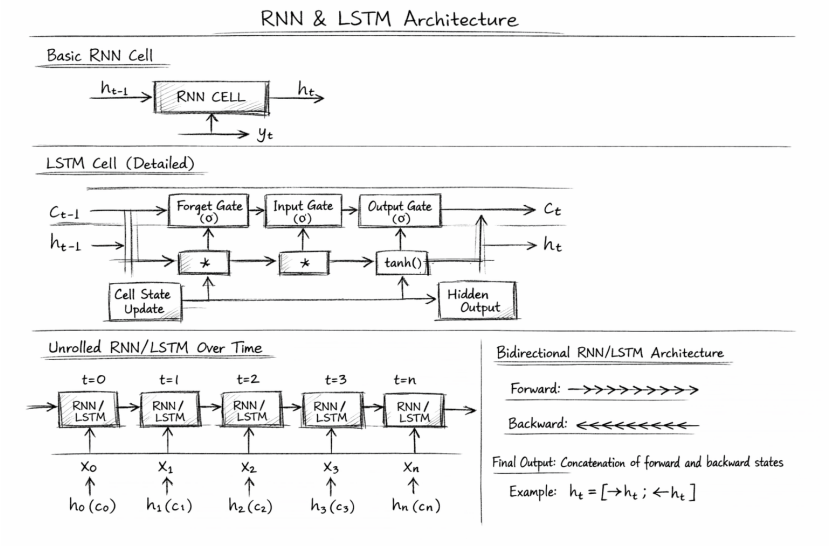
$$o_t = \sigma(W_o[x_t, h_{t-1}] + b_o)$$
$$h_t = o_t \odot \tanh(C_t)$$

Where:

- σ= sigmoid (0 to 1) acts like a switch knob
- $\odot$= element-wise multiply

One-line intuition:

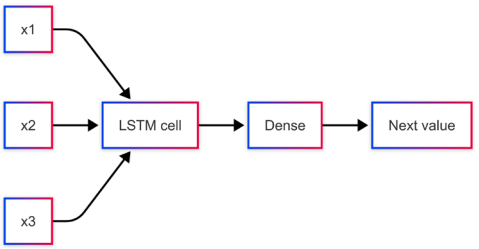
- **Forget gate:** remove useless memory
- **Input gate:** add useful new memory
- **Output gate:** decide what part to show as output



Feature	Basic RNN	LSTM
Memory	Short-term only	Long-term + Short-term
Gates	None	3 gates (Forget, Input, Output)
Cell State	Not present	Present (carries information)
Vanishing Gradient	Severe problem	Mitigated by gates
Use Cases	Simple sequences, short-term dependencies	Long sequences, time series, NLP, speech recognition

How?

1) Create sequences like $[1,2,3] \rightarrow 4$.
2) Reshape data to 3D for LSTM.
3) Train a tiny LSTM model.
4) Predict next number for unseen sequence.



Let's Code!

```
import numpy as np
import tensorflow as tf

# ------------------------------------------------------------
# 1) Hard-coded Sequence Data (20 records)
# Each record = 5 time steps
# Label: 0 = normal, 1 = risky
# ------------------------------------------------------------
X = np.array([
    [120, 130, 110, 140, 125],
```

```
    [100, 105,  98, 110, 102],
    [150, 140, 160, 155, 145],
    [ 90,  95,  92,  88,  94],
    [200, 210, 190, 205, 195],

    [120, 125, 130, 400, 450],
    [300, 320, 310, 305, 315],
    [ 90, 500, 520, 510, 530],
    [250, 260, 270, 280, 290],
    [110, 115, 120, 125, 600],

    [130, 128, 135, 132, 129],
    [ 95,  98, 100,  99,  97],
    [160, 155, 158, 162, 159],
    [105, 110, 100,  98, 102],
    [180, 175, 170, 178, 172],

    [400, 410, 420, 430, 440],
    [350, 100, 380, 110, 390],
    [200, 600, 210, 620, 205],
    [500, 480, 510, 495, 520],
    [100,  90, 700,  85, 720],
], dtype=np.float32)

y = np.array([
    0,0,0,0,0,
    1,1,1,1,1,
    0,0,0,0,0,
    1,1,1,1,1
], dtype=np.float32)

# ------------------------------------------------------------
# 2) Reshape: (samples, time_steps, features)
```

```
# ------------------------------------------------------------
X = X.reshape(-1, 5, 1)

# Normalize amounts for stable training
X = X / 1000.0

# ------------------------------------------------------------
# 3) Build LSTM model
# ------------------------------------------------------------
model = tf.keras.Sequential([
    tf.keras.layers.LSTM(16, input_shape=(5, 1)),
    tf.keras.layers.Dense(8, activation="relu"),
    tf.keras.layers.Dense(1, activation="sigmoid")
])

model.compile(
    optimizer="adam",
    loss="binary_crossentropy",
    metrics=["accuracy"]
)

print(model.summary())

# ------------------------------------------------------------
# 4) Train
# ------------------------------------------------------------
model.fit(X, y, epochs=60, batch_size=4, verbose=1)

# ------------------------------------------------------------
# 5) UNSEEN TEST DATA (one new record)
# ------------------------------------------------------------
unseen = np.array([110, 115, 120, 118, 122],
dtype=np.float32).reshape(1, 5, 1) / 1000.0
pred_prob = model.predict(unseen, verbose=0)[0, 0]
```

```
pred_class = 1 if pred_prob >= 0.5 else 0

print("\nUNSEEN TEST SEQUENCE:", [110,115,120,118,122])
print("Predicted probability:", float(pred_prob))
print("Predicted class:", pred_class, "(0=normal, 1=risky)")

# Save model
model.save("lstm_sequence_model.keras")
print("\nSaved: lstm_sequence_model.keras")
```

UNSEEN TEST SEQUENCE: [110, 115, 120, 118, 122]

Predicted prob: 0.3552417457103729

Predicted class: 0 (0=normal, 1=risky)

What is happening during training?

Step A: Forward pass (time-step by time-step)

For each training record:

LSTM reads Day1 → updates memory

LSTM reads Day2 → updates memory

LSTM reads Day5 → final memory summarizes the pattern

Then Dense layers convert memory into probability.

Step B: Loss calculation

True label: 0 or 1

Predicted: probability (0..1)

Loss measures "how wrong"

Step C: Backpropagation Through Time (BPTT)

The error is sent backward:

through Dense layers

back into the LSTM

and backward across time steps (Day5 → Day1)

Step D: Weight update

Weights are updated using gradient descent/Adam:

$$W \leftarrow W - \eta \cdot \frac{\partial Loss}{\partial W}$$

So models learn patterns like:

stable low spending → normal

repeated high/spiky spending → risky

Best Practices (like CNN best practices)

Normalize inputs (very important)

Prefer LSTM/GRU over basic RNN for longer sequences

Keep sequence length small for beginner experiments (5–20)

Use dropout if model overfits

Use more data for real-world (toy data only teaches the concept)

Common beginner mistakes

Forgetting 3D shape for LSTM input.

Training too few epochs on tiny synthetic data (model won't learn).

Using RNN/LSTM for non-sequence tabular data (use ANN instead).

Chapter 14
Transformers and Attention

Definition

A **Transformer** is a **deep learning architecture** mainly used for **sequence data** (text, speech, time series).

A Transformer is a model that learns relationships between elements in a sequence using **self-attention mechanisms**, allowing parallel computation and long-range dependency modeling.

Transformers are neural networks that learn using attention. Attention means: the model learns which words should pay attention to which other words.

Why did Transformers invent?

Before Transformers:

RNN / LSTM

- Slow (sequential processing)
- Hard to learn long-range dependencies
- **Transformers solved this by:**
  Processing all tokens in parallel
  Capturing long-distance relationships
  Scaling well to very large data

Syntax / Formula

Key formula (small and important)

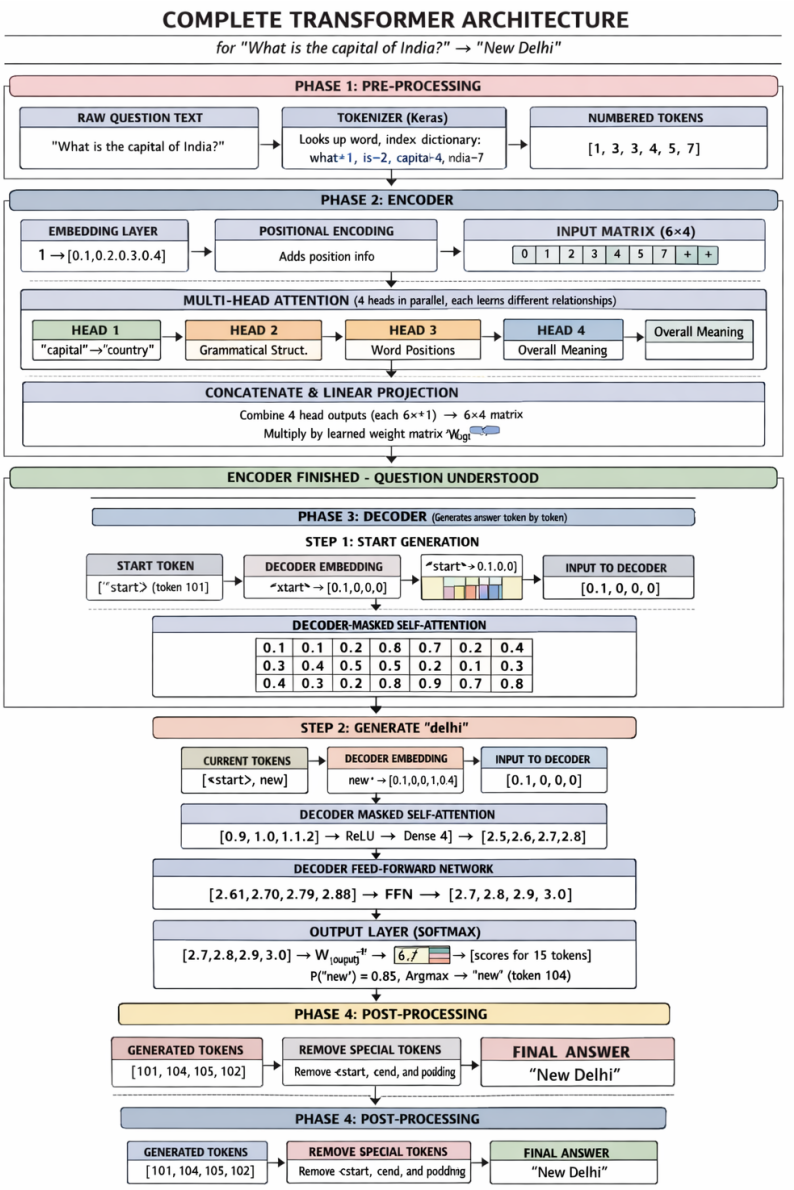
```
Scaled dot-product attention (core idea):
  Attention(Q, K, V) = SoftMax((Q*K^T) / sqrt(d)) * V

Plain meaning:
  Q = what I am looking for
  K = what I have
  V = the information to take
  SoftMax gives weights (importance scores)
```

What?

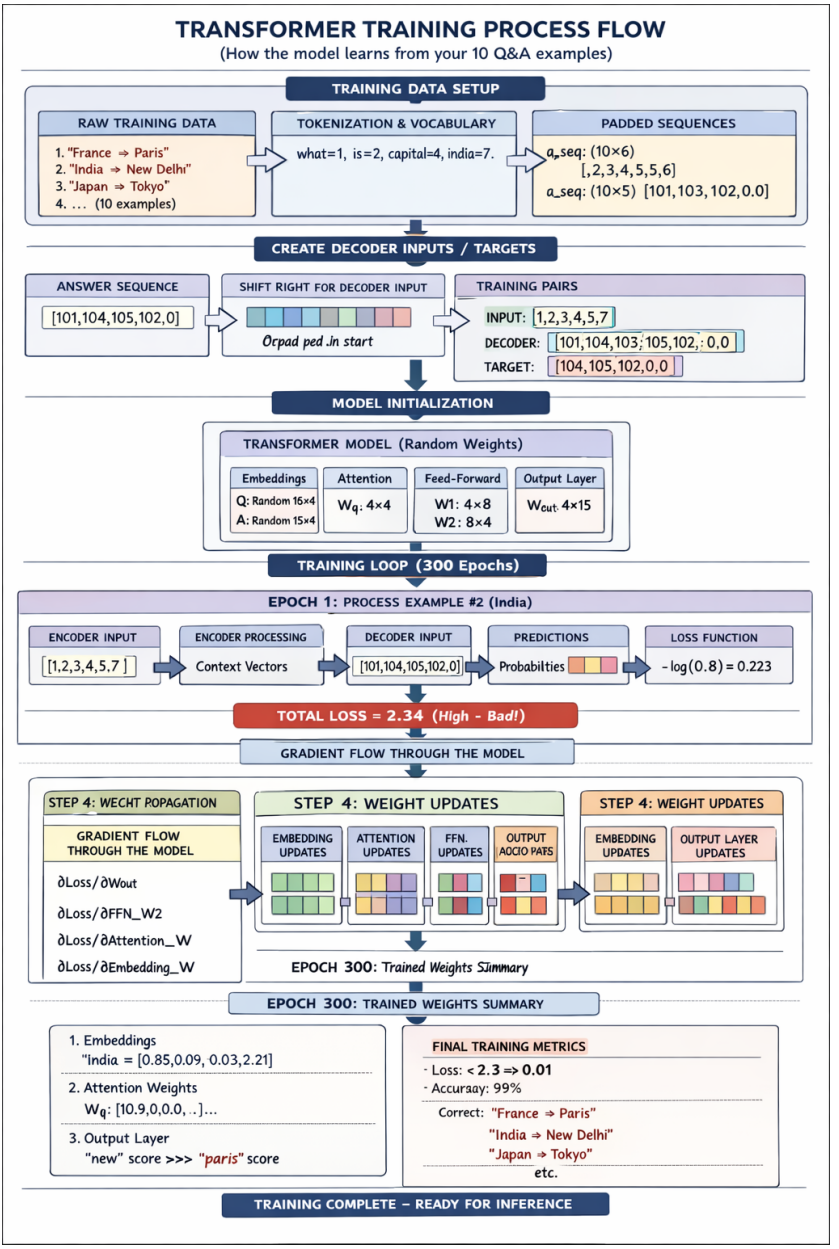
We will build a mini-Transformer-like QA System and text classifier with a tiny dataset.

How?

1) Convert text to tokens (numbers).
2) Convert tokens to embeddings.
3) Apply self-attention (MultiHeadAttention), 4) Pool and classify
Transformer Architecture - showing exactly how data flows through the Transformer for your Q&A example


COMPLETE TRANSFORMER ARCHITECTURE
for "What is the capital of India?" → "New Delhi"
PHASE 1: PRE-PROCESSING
RAW QUESTION TEXT
"What is the capital of India?"
TOKENIZER (Keras)
Looks up word, index dictionary:
what=1, is=2, capital=4, India=7
NUMBERED TOKENS
[1, 3, 3, 4, 5, 7]
PHASE 2: ENCODER
EMBEDDING LAYER
1 → [0.1,0.2,0.3,0.4]
POSITIONAL ENCODING
Adds position info
INPUT MATRIX (6×4)
0 1 2 3 4 5 7 + +
MULTI-HEAD ATTENTION (4 heads in parallel, each learns different relationships)
HEAD 1
"capital" → "country"
HEAD 2
Grammatical Struct.
HEAD 3
Word Positions
HEAD 4
Overall Meaning
Overall Meaning
CONCATENATE & LINEAR PROJECTION
Combine 4 head outputs (each 6×1) → 6×4 matrix
Multiply by learned weight matrix
ENCODER FINISHED - QUESTION UNDERSTOOD
PHASE 3: DECODER (Generates answer token by token)
STEP 1: START GENERATION
START TOKEN
["start"] (token 101]
DECODER EMBEDDING
"start" → [0.1,0,0,0]
"start" → 0.1.0.0]
INPUT TO DECODER
[0.1, 0, 0, 0]
DECODER-MASKED SELF-ATTENTION
0.1 0.1 0.2 0.8 0.7 0.2 0.4
0.3 0.4 0.5 0.5 0.2 0.1 0.3
0.4 0.3 0.2 0.8 0.9 0.7 0.8
STEP 2: GENERATE "delhi"
CURRENT TOKENS
[<start>, new]
DECODER EMBEDDING
new' → [0.1,0,0,1.0.4]
INPUT TO DECODER
[0.1, 0, 0, 0]
DECODER MASKED SELF-ATTENTION
[0.9, 1.0, 1.1.2] → ReLU → Dense 4] → [2.5,2.6, 2.7,2.8]
DECODER FEED-FORWARD NETWORK
[2.61,2.70, 2.79, 2.88] → FFN → [2.7, 2.8, 2.9, 3.0]
OUTPUT LAYER (SOFTMAX)
[2.7,2.8,2.9, 3.0] → W(ouput)T →
6.7
→ [scores for 15 tokens]
P("new') = 0.85, Argmax → "new' (token 104)
PHASE 4: POST-PROCESSING
GENERATED TOKENS
[101, 104, 105, 102]
REMOVE SPECIAL TOKENS
Remove <start, cend, and poddng
FINAL ANSWER
"New Delhi"
PHASE 4: POST-PROCESSING
GENERATED TOKENS
[101, 104, 105, 102]
REMOVE SPECIAL TOKENS
Remove <start, cend, and poddng
FINAL ANSWER
"New Delhi"

Transformer Training process – how transformer get trained on the data



Let's Code –

```
# Dataset
data = [
    ("What is the capital of France?", "Paris"),
    ("What is the capital of India?", "New Delhi"),
    ("What is the capital of Japan?", "Tokyo"),
    ("What is the capital of Germany?", "Berlin"),
    ("What is the capital of Italy?", "Rome"),
    ("What is the capital of USA?", "Washington DC"),
    ("What is the capital of UK?", "London"),
    ("What is the capital of China?", "Beijing"),
    ("What is the capital of Brazil?", "Brasília"),
    ("What is the capital of Australia?", "Canberra"),
]

questions = [q for q, a in data]
answers   = [a for q, a in data]



#Import lib
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Add special tokens to answers
answers = [f"<start> {a} <end>" for a in answers]

tokenizer_q = Tokenizer(filters="")
tokenizer_a = Tokenizer(filters="")

tokenizer_q.fit_on_texts(questions)
tokenizer_a.fit_on_texts(answers)
```

```
q_seq = tokenizer_q.texts_to_sequences(questions)
a_seq = tokenizer_a.texts_to_sequences(answers)

q_seq = pad_sequences(q_seq, padding="post")
a_seq = pad_sequences(a_seq, padding="post")

vocab_q = len(tokenizer_q.word_index) + 1
vocab_a = len(tokenizer_a.word_index) + 1

class Transformer(tf.keras.Model):
    def __init__(self, vocab_q, vocab_a, d_model=64, num_heads=4):
        super().__init__()
        # Embeddings
        self.enc_embedding = tf.keras.layers.Embedding(vocab_q,
d_model)
        self.dec_embedding = tf.keras.layers.Embedding(vocab_a,
d_model)
        # Attention layers
        self.enc_self_attn = tf.keras.layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=d_model
        )

        self.dec_self_attn = tf.keras.layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=d_model
        )

        self.enc_dec_attn = tf.keras.layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=d_model
        )

        # Feed Forward Networks
        self.enc_ffn = tf.keras.Sequential([
            tf.keras.layers.Dense(128, activation="relu"),
            tf.keras.layers.Dense(d_model)
        ])
        self.dec_ffn = tf.keras.Sequential([
```

```
            tf.keras.layers.Dense(128, activation="relu"),
            tf.keras.layers.Dense(d_model)
        ])
        # Output layer
        self.final_layer = tf.keras.layers.Dense(vocab_a,
activation="softmax")
    def call(self, inputs):
        encoder_input, decoder_input = inputs

        # ===== Encoder =====
        enc_emb = self.enc_embedding(encoder_input)
        enc_attn = self.enc_self_attn(enc_emb, enc_emb)
        enc_out  = self.enc_ffn(enc_attn)

        # ===== Decoder =====
        dec_emb = self.dec_embedding(decoder_input)
        dec_self_attn = self.dec_self_attn(
            dec_emb, dec_emb, use_causal_mask=True
        )
        dec_enc_attn = self.enc_dec_attn(
            dec_self_attn, enc_out
        )
        dec_out = self.dec_ffn(dec_enc_attn)

        return self.final_layer(dec_out)

# Model Development
model = Transformer(vocab_q, vocab_a)

model.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)
# Shift decoder input
```

```
decoder_input = a_seq[:, :-1]
decoder_target = a_seq[:, 1:]

model.fit(
    [q_seq, decoder_input],
    decoder_target,
    epochs=300,
    verbose=1
)
def ask(question, max_len=5):
    # Encode question
    q = tokenizer_q.texts_to_sequences([question])
    q = pad_sequences(q, maxlen=q_seq.shape[1], padding="post")
    start_id = tokenizer_a.word_index["<start>"]
    end_id   = tokenizer_a.word_index["<end>"]
    decoder_tokens = [start_id]

    for _ in range(max_len):
        dec_seq = pad_sequences(
            [decoder_tokens],
            maxlen=a_seq.shape[1] - 1,
            padding="post"
        )
        preds = model([q, dec_seq])
        next_id = tf.argmax(preds[0, len(decoder_tokens)-1]).numpy()

        if next_id == end_id:
            break
        decoder_tokens.append(next_id)
    return " ".join(
        tokenizer_a.index_word[t]
        for t in decoder_tokens
        if t not in [start_id, end_id, 0]
    )
```

Result

```
print(ask("What is the capital of India?"))
new Delhi
```

KEY POINTS SUMMARY:

Training Flow:

- Data Preparation → Tokenization, padding, vocabulary creation
- Model Initialization → Random weights for all matrices
- 300 Epochs Training → Each epoch processes all 10 examples
- Forward Pass → Encoder understands question, decoder predicts answer
- Loss Calculation → Compare predictions with actual answers
- Backpropagation → Calculate gradients for all weights
- Weight Updates → Adjust weights using Adam optimizer
- Repeat 300x → Weights converge to learned patterns

Inference Flow:

- User Query → "What is the capital of India?"
- Tokenization → Convert to [1,2,3,4,5,7]
- Encoder Processing → Understand question, create context
- Decoder Generation → Generate tokens one by one:
- Start with <start>
- Generate "new" (token 104)
- Generate "delhi" (token 105)
- Generate <end> (token 102)
- Post-processing → Remove special tokens, convert to text
- Final Answer → "New Delhi"

Weight Evolution:

- Epoch 1: Random weights, makes mistakes (might output "Paris" for India)
- Epoch 100: Learning patterns, might output "New" but not "Delhi"
- Epoch 300: Learned perfectly, outputs "New Delhi" for India
- What Actually Gets Learned:
- Word Embeddings: "india" and "capital" become similar in certain dimensions
- Attention Weights: Learn to connect "capital" with country names
- Output Weights: Learn country→capital city mapping
- Pattern Recognition: "capital of X" → "CapitalCityOfX"

Example 2 - Let's Code!

```
import tensorflow as tf
from tensorflow import keras

# Tiny dataset: sentence -> sentiment (1=positive, 0=negative)
texts = [
    "this app is very good",
    "excellent service and fast support",
    "i love this product",
    "great experience and smooth UI",
    "bad service very slow",
    "i hate this app",
    "terrible experience and bugs",
    "not good waste of money",
    "support was helpful",
    "awesome and reliable",
    "very disappointing",
    "poor quality and slow delivery",
]
labels = [1,1,1,1,0,0,0,0,1,1,0,0]

# Text -> integer tokens
vectorize = keras.layers.TextVectorization(max_tokens=200,
output_sequence_length=10)
vectorize.adapt(texts)

X = vectorize(texts)
y = tf.constant(labels, dtype=tf.float32)

# Mini Transformer block
```

```
embed_dim = 16
num_heads = 2

inputs = keras.Input(shape=(10,), dtype=tf.int64)
x = keras.layers.Embedding(input_dim=200,
output_dim=embed_dim)(inputs)

attn_output =
keras.layers.MultiHeadAttention(num_heads=num_heads,
key_dim=embed_dim)(x, x)
x = keras.layers.Add()([x, attn_output])
x = keras.layers.LayerNormalization()(x)

x = keras.layers.GlobalAveragePooling1D()(x)
x = keras.layers.Dense(16, activation="relu")(x)
outputs = keras.layers.Dense(1, activation="sigmoid")(x)

model = keras.Model(inputs, outputs)
model.compile(optimizer="adam", loss="binary_crossentropy",
metrics=["accuracy"])
model.fit(X, y, epochs=30, verbose=0)

test_text = ["service was slow but support was good"]
test_X = vectorize(test_text)
p = model.predict(test_X, verbose=0)[0][0]
print("Sentiment probability (positive):", float(p))
```

Common beginner mistakes

Training a transformer on tiny data and expecting production results (tiny data is only for learning).

Not using enough data for language tasks — transformers usually need much more data.

Confusing attention with memory — attention is 'focus', memory is handled differently.

Chapter 15
Reasoning Models (Tool-Augmented Reasoning for Beginners)

Definition

Reasoning model means: the system solves a problem using multiple steps, not just one direct prediction. In practice, reasoning systems combine (1) a model that decides the steps and (2) tools (math, search, database) to execute steps.

Syntax / Formula

Key formula (small and important)

```
One simple reasoning pattern:
  step1: decide the operation (class)
  step2: extract numbers
  step3: run the operation with a tool

Classification output:
  op = argmax(softmax(logits))
```

Why?

Because in real life, many tasks are not only pattern recognition. They require step-by-step decisions: banking rules, compliance checks, calculations, troubleshooting.

What?

We will build a tiny 'reasoning' demo: classify a word problem into an operation, then compute the answer using Python.

How?

1) Train a text classifier to predict operation: add/subtract/multiply/divide.
2) Use a small rule to extract numbers.

3) Use Python as a calculator tool.
This is a beginner-friendly example of tool-augmented reasoning.

Let's Code!

```
import re
import tensorflow as tf
from tensorflow import keras

# Tiny dataset: text -> operation label
# 0=add, 1=subtract, 2=multiply, 3=divide
texts = [
    "ram has 5 apples and buys 3 more",
    "salary is 50 and bonus is 10 total?",
    "you had 10 tickets and used 4 tickets left?",
    "bank balance 200 and you spend 60 what remains",
    "a box has 6 packets each packet has 4 biscuits",
    "2 bags each has 10 kg rice total kg",
    "split 20 rupees among 4 friends each gets",
    "divide 100 by 5 to get result",
]
labels = [0,0,1,1,2,2,3,3]

vectorize = keras.layers.TextVectorization(max_tokens=200,
output_sequence_length=20)
vectorize.adapt(texts)

X = vectorize(texts)
y = tf.constant(labels)

model = keras.Sequential([
    keras.layers.Embedding(200, 16),
    keras.layers.GlobalAveragePooling1D(),
    keras.layers.Dense(16, activation="relu"),
    keras.layers.Dense(4, activation="softmax")
])

model.compile(optimizer="adam",
loss="sparse_categorical_crossentropy", metrics=["accuracy"])
model.fit(X, y, epochs=80, verbose=0)

def extract_numbers(text):
    nums = re.findall(r"\d+", text)
    return [int(n) for n in nums]

def apply_op(op, a, b):
    if op == 0: return a + b
    if op == 1: return a - b
    if op == 2: return a * b
```

```
    if op == 3: return a / b

op_names = ["ADD", "SUBTRACT", "MULTIPLY", "DIVIDE"]

def solve(text):
    x = vectorize([text])
    probs = model.predict(x, verbose=0)[0]
    op = int(tf.argmax(probs).numpy())
    nums = extract_numbers(text)
    if len(nums) < 2:
        return "Need at least 2 numbers"
    a, b = nums[0], nums[1]
    ans = apply_op(op, a, b)
    return {"operation": op_names[op], "numbers": (a,b),
"answer": ans, "confidence":
float(tf.reduce_max(probs).numpy())}

print(solve("i have 12 rupees and i get 8 more"))
print(solve("i had 50 and i spent 15"))
print(solve("6 packets each has 7 items"))
print(solve("divide 45 by 9"))
```

Common beginner mistakes

Thinking this tiny demo is a full reasoning model — real reasoning needs better number parsing, more steps, and larger data.

Training with too few examples → model may guess wrong. Add more examples.

Not separating 'decision' and 'execution' (tools make reasoning reliable).

Quick exercises

Add 20 new word problems for each operation.

Improve number extraction to handle 'five' (word) as 5.

Add a 2-step problem: 'buy 3 more then divide by 2' (requires multi-step reasoning).

Chapter 16
Specialized & Emerging Neural Networks (Beyond ANN/CNN/RNN/Transformer)

Learning Goals

By the end of this chapter, you will be able to: Identify what type of data you have (sequence / image / graph / 3D / generation / edge devices). Choose the right specialized model (SSM, GNN, ViT, U-Net, Diffusion, NeRF, MoE, KAN, Long-Conv, SNN). Understand each model's core idea in simple words. Run small hard-coded demos (including SNN).

Why?

ANN, CNN, RNN/LSTM, and Transformers are like "general tools."

But real-world problems often need **special tools**:

- Your sequence is **very long** (logs, genomics, legal docs)
- Your data is **connected** (fraud network, social network, supply chain, molecules)
- You need **pixel-perfect output** (medical segmentation)
- You want to **generate images** (create/edit)
- You need **3D** from photos (AR/VR)
- You want **huge models** but lower cost (MoE)
- You want **brain-like efficiency** (SNN, event-based processing)

So specialized models became popular because they solve **specific data + constraint** better.

How to Choose the Right Model (Simple Rule)

Step 1: What is your data?

- Text / signals / time series / very long sequences → SSM
- Graph (nodes + edges) → GNN or Graph Transformer
- Image classification → ViT or EfficientNet/MobileNet
- Image segmentation → U-Net

- Generate / edit images → Diffusion
- 3D from photos → NeRF
- Huge LLM with lower cost → MoE
- Engineering curve fitting with interpretability → KAN
- Event-based / low-power brain-like compute → SNN

Step 2: What is your constraint?

- Need very long context → SSM / Long-Conv
- Need fast and small → MobileNet / EfficientNet
- Need graph reasoning → GNN / Graph Transformers
- Need pixel output → U-Net
- Need generation → Diffusion
- Need low power / event-driven → SNN

What?

The "Specialized Models" Family

Network / Family	Where it's used most	Core idea (in simple words)	Why it became popular
 **SSM / State Space Models** (e.g. Mamba)	Very long sequences (text/audio/genomics/logs)	**Keeps a state** that updates over time; designed to scale **linearly** with sequence length	Handles very long context efficiently.
 **Long-Convolution Sequence Models** (e.g. Hyena)	Long-context language tasks	**Replaces attention with long convolution + gating**	Good for long context efficiently.
 **KAN** (Kolmogorov–Arnold Networks)	Science/engineering fitting, interpretability	Learns **functions on connections (edges)**	Great for function/PDE fitting.
 **GNN** (Graph Neural Networks)	Graph data (fraud rings, networks, molecules)	Learns by **message** passing between nodes/edges	Best for graph structured data.
 **Graph Transformers**	Large/complex graphs	**Uses attention-like** *ideas on graph structure*	Improves graph reasoning.
 **Diffusion Models** (DDPM family)	Image generation/editing, super-resolution	Generate by **starting from noise and** denoising step-by-step	Powering modern image generation.
 **NeRF** (Neural Radiance Fields)	3D scene/view synthesis, AR/VR	Learns a **'continuous' 3D scene**	Big leap in 3D rendering.
 **MoE / Switch Transformer**	Scaling very large models efficiently	**Only some "experts"** activate per token	Efficient scaling of large models.
 **ANN** (Artificial Neural Networks)	General learning tasks	Learns through interconnected layers	Foundation of deep learning.
 **RNN** (Recurrent Neural Networks)	Sequential data (time series, language)	Loops back to remember past sequences	First option for sequence tasks.
 **CNN** (Convolutional Neural Networks)	Image processing, vision	Uses sliding **filters** over images	Core of computer vision.
 **Transformer**	Natural language processing	**Uses attention** across all tokens	Revolutionized NLP tasks.
 **SNN** (Spiking Neural Networks)	Neuromorphic computing, robotics	**Uses spiking neurons** that fire pulses	Brain-inspired computing.
 **SNN** (Spiking Neural Networks)	Spiking Neural Networks	**Uses spiking neurons** that fire pulses	Brain-inspired computing.

Deep Dive 2: GNN (Graph Neural Networks)

Why GNN?

Many business problems are not tables. They are connections:

- Banking fraud: customer → account → transaction → merchant
- Social network: person → friend → group
- Supply chain: supplier → warehouse → shipment
- Molecules: atoms connected by bonds

CNN is for grids (image), Transformer is for sequences (text), but graphs need graph-native learning → GNN.

What is the core idea?

Each node learns by talking to its neighbours.

A node updates its representation by:

- Taking its current features
- Collecting neighbour features
- Mixing them using learnable weights

Simple math (one-layer idea)

$$\boldsymbol{H' = \sigma(\widetilde{A}XW)}$$

- $\boldsymbol{X}$: node feature matrix
- $\widetilde{\boldsymbol{A}}$: adjacency matrix (connections) with self-loops + normalization
- $\boldsymbol{W}$: trainable weights
- $\boldsymbol{\sigma}$: activation (ReLU)

Let's Code (Tiny GNN — Hard-coded Graph)

```
import numpy as np
import tensorflow as tf


# 1) Graph adjacency (5 nodes)
```

```
A = np.array([
    [0,1,1,0,0],
    [1,0,1,1,0],
    [1,1,0,0,1],
    [0,1,0,0,1],
    [0,0,1,1,0]
], dtype=np.float32)



# 2) Node features (5 nodes, 3 features)
X = np.array([
    [1.0, 0.0, 0.0],   # node 0
    [0.9, 0.1, 0.0],   # node 1
    [0.0, 1.0, 0.2],   # node 2
    [0.0, 0.9, 0.3],   # node 3
    [0.1, 0.8, 0.4],   # node 4
], dtype=np.float32)

# 3) Labels (2 classes). -1 means unknown (test nodes)
y = np.array([0, 0, 1, -1, -1], dtype=np.int32)
train_mask = (y != -1)
y_train = y[train_mask]

# ---- Normalize adjacency ----
I = np.eye(A.shape[0], dtype=np.float32)
A_hat = A + I
D = np.sum(A_hat, axis=1)
D_inv_sqrt = np.diag(1.0 / np.sqrt(D))
A_norm = D_inv_sqrt @ A_hat @ D_inv_sqrt

A_norm_tf = tf.constant(A_norm, dtype=tf.float32)
X_tf = tf.constant(X, dtype=tf.float32)
```

```
# ---- 2-layer GNN ----
W1 = tf.Variable(tf.random.normal([3, 8], stddev=0.1))
W2 = tf.Variable(tf.random.normal([8, 2], stddev=0.1))
opt = tf.keras.optimizers.Adam(learning_rate=0.05)

def forward():
    H1 = tf.nn.relu(A_norm_tf @ X_tf @ W1)
    logits = A_norm_tf @ H1 @ W2
    return logits

for epoch in range(200):
    with tf.GradientTape() as tape:
        logits = forward()
        logits_train = tf.boolean_mask(logits, train_mask)
        loss = tf.reduce_mean(
            tf.nn.sparse_softmax_cross_entropy_with_logits(
                labels=tf.constant(y_train),
                logits=logits_train
            )
        )
    grads = tape.gradient(loss, [W1, W2])
    opt.apply_gradients(zip(grads, [W1, W2]))
    if epoch % 50 == 0:
        print(f"Epoch {epoch:03d} | Loss: {loss.numpy():.4f}")
pred = tf.argmax(forward(), axis=1).numpy()
print("\nPredicted class for each node:", pred)
print("Known labels (train nodes):    ", y)

Predicted class for each node: [0 0 1 1 1]
Known labels (train nodes):     [ 0  0  1 -1 -1]
```

Diffusion Models (Modern Image Generation)

Why Diffusion?

GANs were popular, but diffusion models became the new standard because:

- Stable training
- High image quality
- Flexible editing

What is the core idea?

Start with noise → remove noise gradually → get an image.

Tiny intuition equation

$$x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon$$

- x_0= real image
- ϵ= noise
- model learns to predict/remove noise

(For your beginner book, keep diffusion as concept + diagram; full training is heavy.)

What is the core idea?

Start with noise → remove noise gradually → get an image.

Let's Code: Tiny Text Diffusion (Hard-coded Data, TensorFlow/Keras)

This is a **toy diffusion text generator**:

- Word-level tokens
- Fixed sentence length
- Learns from a very small dataset

- Can do **conditional generation** (keep a prefix fixed like "i love …")

```
This is for understanding. Real diffusion LMs use stronger
architectures and better diffusion math.

import numpy as np
import tensorflow as tf


# 1) Hard-coded tiny dataset
sentences = [
    "i love ai",
    "i love pizza",
    "i love music",
    "i love books",
    "i love coffee",
    "i love cricket",
    "i love movies",
    "i love coding",
]

# We'll use fixed length = 3 tokens: ["i", "love", "<thing>"]
# So this dataset is perfect for a simple conditional demo.

# Build vocabulary
special = ["<pad>"]
words = sorted(list(set(" ".join(sentences).split())))
vocab = special + words
word2id = {w:i for i,w in enumerate(vocab)}
id2word = {i:w for w,i in word2id.items()}

V = len(vocab)
L = 3  # fixed length

def encode(s):
    toks = s.split()
```

```
    ids = [word2id[t] for t in toks]
    # pad if needed (not needed here, but kept for completeness)
    ids = ids + [word2id["<pad>"]] * (L - len(ids))
    return ids[:L]

X0 = np.array([encode(s) for s in sentences], dtype=np.int32)  #
shape (N, L)

# 2) Diffusion schedule (token replacement probability)
T = 10               # number of diffusion steps
gamma_max = 0.70     # max corruption probability

def gamma_t(t):
    # t in [1..T], linearly increase noise
    return (t / T) * gamma_max

rng = np.random.default_rng(0)

def corrupt_tokens(x0, t):
    """
    Make x_t from x0 by replacing each token with prob gamma_t(t)
    Replacement is random token from vocab (excluding <pad> to keep
it meaningful).
    """
    x0 = x0.copy()
    g = gamma_t(t)

    mask = rng.random(x0.shape) < g  # True => replace
    # random tokens in [1..V-1] so we avoid <pad> token for
replacements
    random_tokens = rng.integers(1, V, size=x0.shape,
dtype=np.int32)
    xt = np.where(mask, random_tokens, x0).astype(np.int32)
    return xt

# 3) Build a tiny denoiser model
```

```
#    Input: noisy tokens xt + timestep t
#    Output: logits for clean tokens x0 at each position

d_model = 64

token_in = tf.keras.Input(shape=(L,), dtype=tf.int32, name="xt")
t_in     = tf.keras.Input(shape=(), dtype=tf.int32, name="t")

tok_emb = tf.keras.layers.Embedding(V, d_model,
name="tok_emb")(token_in)   # (B,L,d)
pos_ids = tf.range(L)[None, :]  # (1,L)
pos_emb_layer = tf.keras.layers.Embedding(L, d_model,
name="pos_emb")
pos_emb = pos_emb_layer(pos_ids)  # (1,L,d)

t_emb_layer = tf.keras.layers.Embedding(T+1, d_model, name="t_emb")
t_emb = t_emb_layer(t_in)  # (B,d)
t_emb = tf.keras.layers.Reshape((1, d_model))(t_emb)  # (B,1,d)
broadcast across positions

h = tok_emb + pos_emb + t_emb
h = tf.keras.layers.Dense(128, activation="relu")(h)
h = tf.keras.layers.Dense(128, activation="relu")(h)
logits = tf.keras.layers.Dense(V, name="logits")(h)  # (B,L,V)

model = tf.keras.Model(inputs=[token_in, t_in], outputs=logits)
model.compile(
    optimizer=tf.keras.optimizers.Adam(0.01),
loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
)

# 4) Training data generator
#    We train: (xt, t) -> x0

def make_batch(batch_size=32):
```

```
    idx = rng.integers(0, len(X0), size=batch_size)
    x0 = X0[idx]
    t  = rng.integers(1, T+1, size=batch_size, dtype=np.int32)
    xt = np.stack([corrupt_tokens(x0[i], int(t[i])) for i in
range(batch_size)], axis=0)
    return ({"xt": xt, "t": t}, x0)

# Train for a few epochs (tiny dataset, so we just do a bit)
for epoch in range(30):
    xb, yb = make_batch(batch_size=64)
    loss = model.train_on_batch(xb, yb)
    if (epoch+1) % 5 == 0:
        print(f"Epoch {epoch+1:02d} | loss={loss:.4f}")

# 5) Text generation (conditional)
#    Keep prefix "i love" fixed, generate the 3rd token.

def decode(ids):
    return " ".join([id2word[i] for i in ids if id2word[i] !=
"<pad>"])

def generate_with_prefix(prefix="i love", steps=T, sample=True):
    prefix_ids = encode(prefix + " <pad>")  # placeholder last token
    # We'll fix first 2 tokens, only generate token position 2.
    fixed_positions = [0, 1]

    # Start from pure noise for all positions, then overwrite fixed
prefix positions
    xt = rng.integers(1, V, size=(1, L), dtype=np.int32)
    xt[0, fixed_positions] = np.array(prefix_ids)[fixed_positions]

    # Reverse diffusion: t = T..1
    for t in range(steps, 0, -1):
        logits = model.predict({"xt": xt, "t": np.array([t],
dtype=np.int32)}, verbose=0)
        probs = tf.nn.softmax(logits, axis=-1).numpy()  # (1,L,V)
```

```
        # Choose tokens
        new_xt = xt.copy()
        for pos in range(L):
            if pos in fixed_positions:
                continue
            p = probs[0, pos]
            if sample:
                new_xt[0, pos] = rng.choice(np.arange(V), p=p)
            else:
                new_xt[0, pos] = int(np.argmax(p))

        # Re-noise slightly for next step (optional refinement
trick)
        # Less noise as t decreases
        g_prev = gamma_t(max(t-1, 1))
        if t > 1:
            if rng.random() < g_prev:
                # randomize the generated position a bit to allow
exploration
                new_xt[0, 2] = rng.integers(1, V, dtype=np.int32)

        # keep prefix fixed
        new_xt[0, fixed_positions] =
np.array(prefix_ids)[fixed_positions]
        xt = new_xt

    return decode(xt[0])

print("\nGenerated samples:")
for _ in range(8):
    print(" -", generate_with_prefix("i love", steps=T, sample=True))
```

output

```
Epoch 05 | loss=1.7493
Epoch 10 | loss=1.4604
Epoch 15 | loss=1.2366
Epoch 20 | loss=1.0730
Epoch 25 | loss=0.9605
Epoch 30 | loss=0.8777
Generated samples:
 - i love cricket
 - i love coffee
 - i love books
 - i love coffee
 - i love coffee
 - i love cricket
 - i love coding
 - i love movies
```

What's really happening during "Diffusion Text Generation"?

Training time

- We take a clean sentence x_0
- We corrupt it into x_t
- Model learns **given x_t and t, predict original x_0**

So, it learns to denoise at many noise levels.

Generation time

- Start from random tokens (noise)
- Repeatedly ask the model: "What should the clean token be here?"
- Each step improves the sentence

This is why diffusion feels like **iterative polishing**.

Deep Dive: SNN (Spiking Neural Networks)

Why SNN?

A normal neural network always computes with floating numbers at every layer. But the brain works differently:

- Neurons stay quiet most of the time
- They fire only when important → **spikes**

So SNN can be:

- Energy efficient
- Good for event-based sensors (like event cameras)
- Useful in neuromorphic hardware

What is a "Spike"?

A spike is like a short "ON" signal (0 or 1).

What does "spike" mean

A **spike** is a very short "fire" signal from a neuron.

In normal neural networks:

- neurons output continuous values like **0.23, -1.7, 4.1**

In Spiking Neural Networks:

neurons output **events**, typically:

- **1** = neuron fired (spike happened)
- **0** = neuron did not fire

So a spike is like:

- a blink
- a tap
- a pulse
- a notification
  that happens at a specific time.

Real-life analogy

Think of a doorbell:

- Most of the time it is quiet (0)
- When someone presses it, it rings (1)
  That ring is a **spike**.
  Instead of sending continuous values, neurons communicate using spikes.

How an SNN neuron works (LIF intuition)

A common neuron is **LIF (Leaky Integrate-and-Fire)**:

- It **adds** incoming signals into a "membrane potential" v
- It **leaks** (slowly decreases) over time
- If $v$crosses a threshold → it **spikes** and resets

Simple equations

$$v_t = \alpha v_{t-1} + w \cdot x_t$$

If $v_t \geq \theta$then spike $s_t = 1$and reset $v_t \leftarrow 0$

Where:

- x_t= input spike at time t (0/1)
- w= weight
- α= leak factor (0.8, 0.9 etc.)
- θ= threshold

Why are spikes useful?

Because spikes can represent information efficiently:

- **Event-based sensors** (like event cameras) naturally output spikes: "pixel changed!"
- **Brain-like processing**: neurons communicate via spikes
- **Energy saving**: if nothing important is happening, no spikes → less computation

How spikes are used in our example

Input spikes

Our input data X[t, neuron] is a spike train:

- At time t, neuron i is either 0 or 1.

Example (for one neuron across time):
0 0 1 0 1 1 0 0 0 1 ...
This means the neuron fired at those time positions.

Output spikes (inside SNN)

Inside the network, each LIF neuron maintains a voltage (memory). When voltage crosses a threshold → it produces a spike (1), then resets.

Let's Code (SNN Toy Demo with Hard-coded Spikes)

```
import numpy as np
import tensorflow as tf

# ============================================================
# 1) Surrogate spike function (so gradients can flow)
#    Forward: hard threshold (0/1 spike)
#    Backward: smooth sigmoid derivative (surrogate gradient)
# ============================================================
@tf.custom_gradient
def spike_fn(x):
    # Forward pass: binary spikes
    y = tf.cast(x > 0.0, tf.float32)

    # Backward pass: surrogate gradient
    def grad(dy):
        beta = 10.0  # steeper = closer to hard step
        s = tf.sigmoid(beta * x)
        return dy * beta * s * (1.0 - s)

    return y, grad
```

```
# ============================================================
# 2) LIF (Leaky Integrate-and-Fire) RNN Cell
#    v_t = alpha*v_{t-1} + input_current
#    spike when v_t - threshold > 0
#    reset voltage after spike
# ============================================================
class LIFCell(tf.keras.layers.Layer):
    def __init__(self, units, alpha=0.90, threshold=1.0, **kwargs):
        super().__init__(**kwargs)
        self.units = units
        self.alpha = alpha
        self.threshold = threshold

        # RNN requires these:
        self.state_size = [units, units]  # [voltage v, previous
spikes s]
        self.output_size = units

    def build(self, input_shape):
        input_dim = input_shape[-1]

        # Input weights (from input spikes to hidden neurons)
        self.kernel = self.add_weight(
            shape=(input_dim, self.units),
            initializer="glorot_uniform",
            trainable=True,
            name="kernel"
        )

        # Recurrent weights (from previous spikes to current
neurons)
        self.recurrent_kernel = self.add_weight(
            shape=(self.units, self.units),
            initializer="orthogonal",
            trainable=True,
```

```
            name="recurrent_kernel"
        )

        self.bias = self.add_weight(
            shape=(self.units,),
            initializer="zeros",
            trainable=True,
            name="bias"
        )

        super().build(input_shape)

    def call(self, inputs, states):
        v_prev, s_prev = states  # voltage and previous spikes

        # Current = input contribution + recurrent spike
contribution
        I_t = tf.matmul(inputs, self.kernel) + tf.matmul(s_prev,
self.recurrent_kernel) + self.bias

        # LIF membrane update (leak + integrate)
        v_t = self.alpha * v_prev + I_t

        # Spike generation (hard threshold in forward, surrogate
gradient in backward)
        s_t = spike_fn(v_t - self.threshold)

        # Reset after spike
        v_t = v_t * (1.0 - s_t)

        return s_t, [v_t, s_t]

# ============================================================
# 3) Create a hard-coded spike-train dataset
#    Class 0: high rate early, low rate late
#    Class 1: low rate early, high rate late
```

```
# ============================================================
def make_spike_dataset(num_samples=800, T=30, N=12, seed=0):
    rng = np.random.default_rng(seed)

    X = np.zeros((num_samples, T, N), dtype=np.float32)
    y = np.zeros((num_samples,), dtype=np.int32)

    for i in range(num_samples):
        label = rng.integers(0, 2)  # 0 or 1
        y[i] = label

        # Two-phase rates
        if label == 0:
            p_early, p_late = 0.35, 0.08
        else:
            p_early, p_late = 0.08, 0.35

        # Generate spikes: Bernoulli per timestep per neuron
        for t in range(T):
            p = p_early if t < (T // 2) else p_late
            X[i, t, :] = (rng.random(N) < p).astype(np.float32)

    return X, y

T = 30
N = 12

X, y = make_spike_dataset(num_samples=1200, T=T, N=N, seed=42)

# Train/test split
idx = np.arange(len(X))
np.random.shuffle(idx)

train_size = int(0.8 * len(X))
train_idx, test_idx = idx[:train_size], idx[train_size:]
```

```
X_train, y_train = X[train_idx], y[train_idx]
X_test, y_test   = X[test_idx], y[test_idx]

# ============================================================
# 4) Build SNN model in Keras
#    Input: (T, N) spike sequence
#    LIF-RNN -> (hidden spikes)
#    Readout Dense -> class logits
# ============================================================
units = 32

inputs = tf.keras.Input(shape=(T, N), name="spike_trains")

lif_rnn = tf.keras.layers.RNN(
    LIFCell(units=units, alpha=0.90, threshold=1.0),
    return_sequences=False,   # only final hidden spikes used
    name="lif_rnn"
)(inputs)

# Simple readout layer
outputs = tf.keras.layers.Dense(2, activation=None,
name="readout")(lif_rnn)

model = tf.keras.Model(inputs, outputs)
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.01),

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
,
    metrics=["accuracy"]
)

model.summary()

# 5) Train
```

```
history = model.fit(
    X_train, y_train,
    epochs=12,
    batch_size=64,
    validation_split=0.2,
    verbose=1
)

# 6) Evaluate on test
test_loss, test_acc = model.evaluate(X_test, y_test, verbose=0)
print(f"\nTest Accuracy: {test_acc:.4f}")

# ============================================================
# 7) Unseen data test (manual example)
#    Create a new sample with "late spikes" => should be class 1
# ============================================================
def make_one_unseen(label, T=30, N=12, seed=999):
    rng = np.random.default_rng(seed)
    X1 = np.zeros((1, T, N), dtype=np.float32)

    if label == 0:
        p_early, p_late = 0.35, 0.08
    else:
        p_early, p_late = 0.08, 0.35

    for t in range(T):
        p = p_early if t < (T // 2) else p_late
        X1[0, t, :] = (rng.random(N) < p).astype(np.float32)
    return X1

unseen = make_one_unseen(label=1, T=T, N=N)
logits = model.predict(unseen, verbose=0)
pred = int(np.argmax(logits[0]))

print("\nUnseen sample predicted class:", pred, "| (expected: 1)")
```

Output

Unseen sample predicted class: 1 | (expected: 1)

Chapter 17
Production Deep Learning (API + Model Serving Basics)

Definition

Production means: your model is not only a notebook experiment. It is a service that real users call, with stability, logging, and monitoring.

Syntax / Formula

Key formula (small and important)

Production checklist (simple):
1) Input contract (what JSON fields you accept)
2) Preprocessing (same as training)
3) Model inference (predict)
4) Postprocessing (format output)
5) Logging + monitoring
6) Versioning (model v1, v2, ...)

Why?

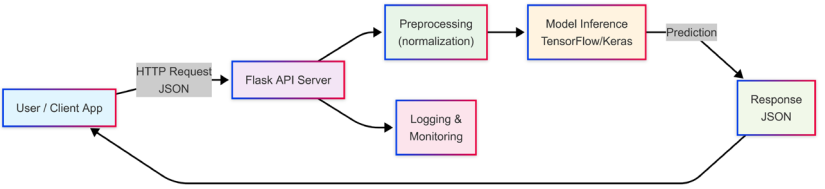
Because 80% of real work is deployment and maintenance, not just training.

What?

We will create a small Flask API that serves the Real Estate rent model from Chapter 5.

How?

1) Train model and save it.
2) Write Flask app to load model at startup.
3) Create /predict endpoint.
4) Containerize with Docker (next chapters).



Let's Code!

```
# --------- train_and_save.py ---------
import numpy as np
from tensorflow import keras

X = np.array([
    [650, 1, 12],
    [800, 2, 8],
    [900, 2, 5],
    [1100, 3, 6],
    [1200, 3, 2],
    [1500, 3, 1],
    [1700, 4, 3],
    [2000, 4, 2],
    [2400, 4, 1],
    [3000, 5, 4],
    [3500, 5, 2],
    [4000, 6, 1],
], dtype=np.float32)

y =
np.array([18000,23000,26000,32000,36000,42000,48000,55000,65000,
80000,92000,110000], dtype=np.float32)

X_mean = X.mean(axis=0)
X_std = X.std(axis=0) + 1e-8
Xn = (X - X_mean) / X_std

model = keras.Sequential([
    keras.layers.Dense(8, activation="relu", input_shape=(3,)),
    keras.layers.Dense(4, activation="relu"),
    keras.layers.Dense(1)
])

model.compile(optimizer=keras.optimizers.Adam(0.01), loss="mse")
model.fit(Xn, y, epochs=500, batch_size=4, verbose=0)

model.save("rent_model.keras")
np.save("mean.npy", X_mean)
np.save("std.npy", X_std)

print("Saved model + preprocessing files.")

# --------- app.py (Flask API) ---------
```

```
from flask import Flask, request, jsonify
import numpy as np
from tensorflow import keras

app = Flask(__name__)

model = keras.models.load_model("rent_model.keras")
mean = np.load("mean.npy")
std = np.load("std.npy")

@app.route("/predict", methods=["POST"])
def predict():
    data = request.get_json()
    # Expected JSON: {"area_sqft":1400, "bedrooms":3,
"age_years":4}
    x = np.array([[data["area_sqft"], data["bedrooms"],
data["age_years"]]], dtype=np.float32)
    xn = (x - mean) / std
    rent = float(model.predict(xn, verbose=0)[0][0])
    return jsonify({"predicted_rent_inr": int(rent)})

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8080)
```

Common beginner mistakes

Doing preprocessing differently in production vs training (must match exactly).

Loading model inside every request (slow). Load once at startup.

Not versioning models (always keep model_v1, model_v2...).

Chapter 18
Deploy to AWS
(Lambda Container + API Gateway)

Definition

AWS deployment means: your trained model runs in AWS and people can call it over the internet securely.

Syntax / Formula

Key formula (small and important)

AWS Lambda (container image) flow:
Client → API Gateway → Lambda (container) → Model → Response

Why container? Because you can package Python + TensorFlow + your model together.

Why?

Because AWS is common in enterprises and gives strong scalability and security controls.

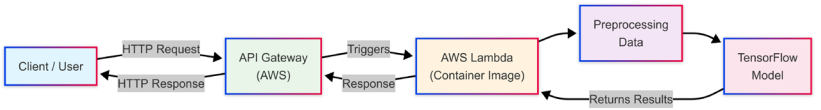
What?

We will deploy the Flask API (Chapter 11) as a container image to AWS Lambda + API Gateway.

How?

High-level steps:
1) Convert Flask app to a Lambda-compatible handler OR use a lightweight ASGI/WSGI adapter.
2) Build a Docker image.
3) Push image to Amazon ECR.
4) Create Lambda function from image.
5) Connect API Gateway endpoint.

Note: This chapter gives production steps; you will need AWS account + IAM permissions.



Let's Code!

```
# --------- Dockerfile (AWS Lambda container) ---------
# Use AWS base image for Python
FROM public.ecr.aws/lambda/python:3.10

# Install dependencies
COPY requirements.txt .
RUN pip install -r requirements.txt

# Copy app + model files
COPY app.py rent_model.keras mean.npy std.npy ./

# Lambda needs a handler. We will use Mangum to adapt (Flask →
ASGI is not native).
# Easiest path: use FastAPI in production. For learning, keep
Flask and use awslambdaric with a custom handler.
# Below is a simple approach: switch to FastAPI for Lambda.
# (If you want, I can generate a separate FastAPI version.)

CMD ["app.lambda_handler"]

# --------- requirements.txt ---------
tensorflow==2.15.0
numpy==1.26.4
fastapi==0.111.0
mangum==0.17.0
uvicorn==0.30.1

# --------- app.py (FastAPI version for Lambda) ---------
from fastapi import FastAPI
from pydantic import BaseModel
import numpy as np
from tensorflow import keras
from mangum import Mangum

api = FastAPI()
model = keras.models.load_model("rent_model.keras")
mean = np.load("mean.npy")
std = np.load("std.npy")

class RentRequest(BaseModel):
    area_sqft: float
    bedrooms: float
    age_years: float
```

```
@api.post("/predict")
def predict(req: RentRequest):
    x = np.array([[req.area_sqft, req.bedrooms, req.age_years]],
dtype=np.float32)
    xn = (x - mean) / std
    rent = float(model.predict(xn, verbose=0)[0][0])
    return {"predicted_rent_inr": int(rent)}

lambda_handler = Mangum(api)

# --------- AWS CLI steps (summary) ---------
# 1) Create ECR repo:
#    aws ecr create-repository --repository-name rent-model-api
#
# 2) Build and tag image:
#    docker build -t rent-model-api .
#
# 3) Login to ECR and push:
#    aws ecr get-login-password | docker login --username AWS --
password-stdin <ACCOUNT_ID>.dkr.ecr.<REGION>.amazonaws.com
#    docker tag rent-model-api:latest
<ACCOUNT_ID>.dkr.ecr.<REGION>.amazonaws.com/rent-model-
api:latest
#    docker push
<ACCOUNT_ID>.dkr.ecr.<REGION>.amazonaws.com/rent-model-
api:latest
#
# 4) Create Lambda from container image, then create API Gateway
HTTP API -> integrate with Lambda.
```

Common beginner mistakes

Trying to deploy full TensorFlow to Lambda without container (zip) it becomes too large.

Not using a compatible base image for Lambda.

Forgetting to include preprocessing files (mean/std) in container.

Quick exercises

Add authentication (API key or IAM) at API Gateway.

Add CloudWatch logging for request/response time.

Create a second model version and deploy with alias (v1/v2).

Common beginner mistakes

Forgetting to listen on 0.0.0.0:8080 (Cloud Run requirement).

Not pinning dependency versions (deployment can break later).

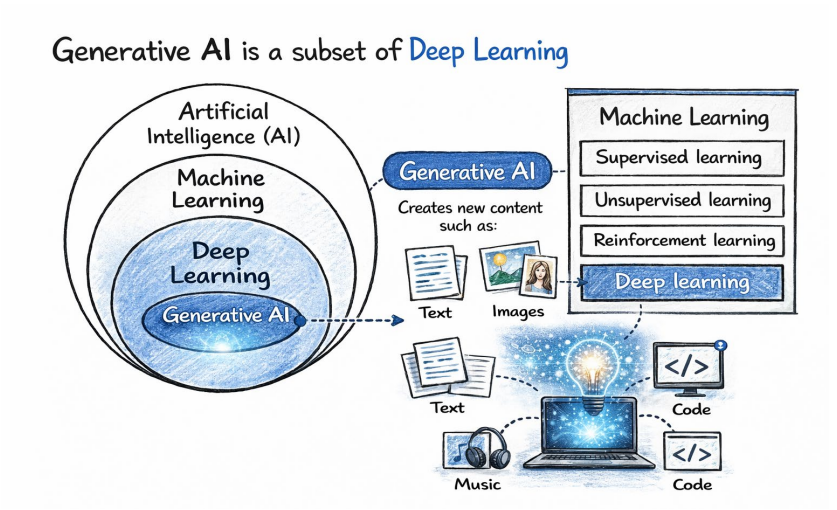
Large container images (keep slim base image).

IV - Practical Generative AI

Generative AI is a part of Machine Learning that creates new content like text, images, and music.

Table of Contents

0. Generative AI: From Prediction to Creation
1. Embeddings: Turning Meaning into Numbers
2. Vector Databases: Searching by Meaning
3. RAG: Retrieve First, Answer Second
4. Prompt Design: Better Instructions, Better Output
5. Hallucination Control: Reducing Confident Mistakes
6. Evaluation of LLM Answers: Check Before You Trust
7. Agents' vs Workflows: Choosing the Right Pattern
8. When Not to Use an LLM: Use AI with Judgment



Quick Setup

pip install openai requests numpy

Pull local Ollama models

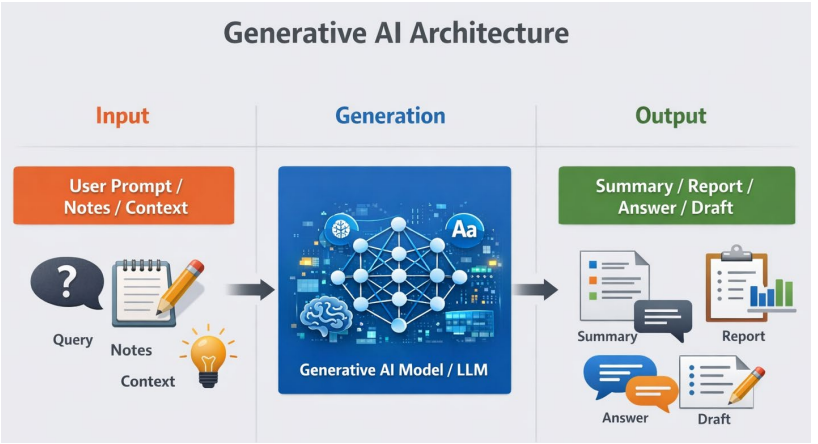
ollama pull deepseek-r1:latest

ollama pull embeddinggemma

Note: deepseek-r1:latest is used for local generation. For local embeddings, use an embedding model such as embeddinggemma. Chat models and embedding models usually serve different purposes in local GenAI pipelines.

Chapter 0
Generative AI: From Prediction to Creation

Generative AI is a type of Artificial Intelligence that can create new content such as text, code, summaries, emails, reports, explanations, and chatbot responses. Traditional Machine Learning often predicts a value or a label, while Generative AI produces something new.



Why?

Modern work is full of language-heavy tasks such as summarizing meetings, drafting emails, converting raw notes into structured reports, explaining technical content in simple words, and answering questions from policies or manuals. Generative AI is useful because it helps automate or accelerate that kind of work.

What?

Common use cases include meeting summaries, design note drafting, release note generation, FAQ assistants, developer documentation, learning support, and business point-of-view writing.

How?

At a practical level, the system receives a prompt, optionally receives supporting context, processes the text internally, and then generates the response step by step until the output is complete.

Let's Code! — Tiny Practical Example

```
import requests


meeting_notes = """
Client wants faster onboarding.
Current process takes 3 days.
Team suggested API automation for document verification.
Compliance approval is still mandatory.
Pilot should start next month.
"""
def generate_summary(notes: str) -> str:
    prompt = f"""
You are a business analyst. Summarize the following meeting notes in
exactly three sections:
1. Business Need
2. Proposed Approach
3. Next Step


Use simple business language and keep each section concise.
Meeting Notes:
{notes}
"""
    response = requests.post(
        "http://localhost:11434/api/chat",
        json={
            "model": "deepseek-r1:latest",
            "messages": [{"role": "user", "content": prompt}],
            "stream": False
        }
    )
```

```
    response.raise_for_status()
    return response.json()["message"]["content"].strip()

# Example usage
print(generate_summary(meeting_notes))
```

output

Meeting Summary

1. Business Need

- The client requires faster onboarding, currently taking 3 days.

- Immediate action is needed to streamline the process.

2. Proposed Approach

- API automation will be used to speed up document verification.

- Compliance approval remains a necessary step, though delays are expected.

- The pilot will commence next month.

3. Next Step

- Initiate the pilot program.

- Once the pilot is complete, review feedback to proceed with compliance approval.

Key Takeaways

Generative AI creates content rather than only predicting labels or numbers.

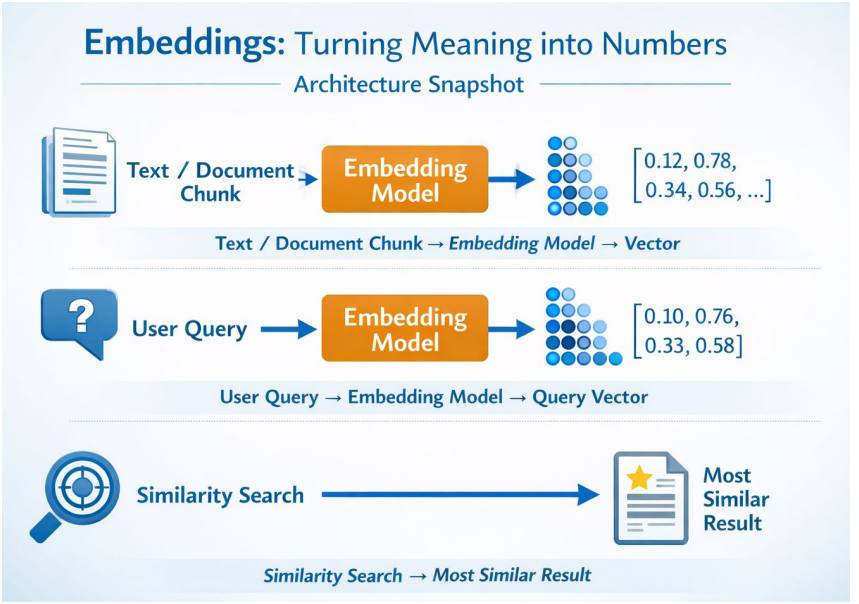
It is especially useful for language-heavy and document-heavy tasks.

A simple mental model is prompt plus context goes into the model, and generated output comes out.

Generated text should still be reviewed carefully.

Chapter 1
Embeddings: Turning Meaning into Numbers

Embeddings are numeric representations of meaning. They convert text into vectors so that similar meanings are placed closer together in vector space.



Why?

Keyword search is useful but limited. A user may ask a question in different words from how the knowledge is written. Embeddings help search by meaning rather than exact wording.

What?

Embeddings are used in semantic search, retrieval systems, recommendation, clustering, duplicate detection, and RAG.

How?

Take each document or chunk, convert it into a vector, convert the user query into a vector, compare similarity, and return the closest match or matches.

Let's Code! - Embeddings Example

```
import requests
import numpy as np


chunks = [
    {"id": 1, "text": "Travel claims must be submitted within 15 days.", "source": "Travel Policy"},
    {"id": 2, "text": "Late claims require manager approval.", "source": "Travel Policy"},
    {"id": 3, "text": "Employees receive 12 casual leaves per year.", "source": "HR Policy"}
]


query = "How many days do I have to submit travel reimbursement?"


def get_embedding(text: str):
    response = requests.post(
        "http://localhost:11434/api/embed",
        json={"model": "nomic-embed-text", "input": text}
    )
    response.raise_for_status()
    return response.json()["embeddings"][0]


chunk_embeddings = [get_embedding(chunk["text"]) for chunk in chunks]
query_embedding = get_embedding(query)


def cosine_similarity(a, b):
    a = np.array(a)
    b = np.array(b)
    return np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))


scores = [cosine_similarity(query_embedding, emb) for emb in chunk_embeddings]
```

```
ranked = sorted(zip(chunks, scores), key=lambda x: x[1],
reverse=True)


print("Top Matching Chunks:\n")
for chunk, score in ranked:
    print(f"ID: {chunk['id']}")
    print(f"Source: {chunk['source']}")
    print(f"Text: {chunk['text']}")
    print(f"Score: {score:.4f}\n")
```

Output

Top Matching Chunks:

ID: 1

Source: Travel Policy

Text: Travel claims must be submitted within 15 days.

Score: 0.6747

ID: 2

Source: Travel Policy

Text: Late claims require manager approval.

Score: 0.5070

ID: 3

Source: HR Policy

Text: Employees receive 12 casual leaves per year.

Score: 0.5021

Key Takeaways

Embeddings convert meaning into numbers.

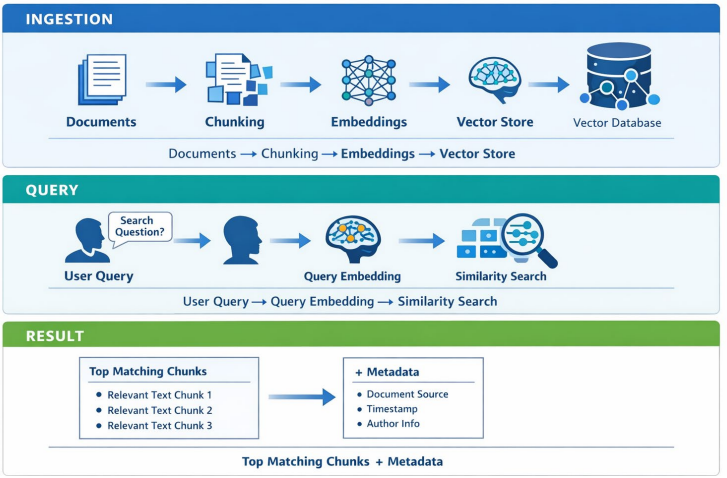
Similar meaning tends to produce closer vectors.

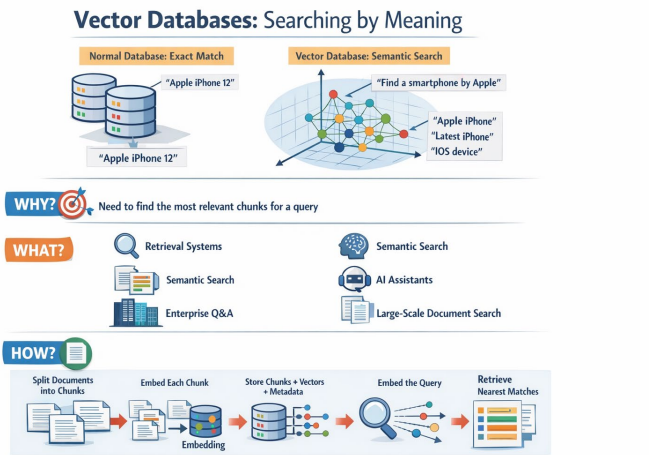
Embeddings make semantic search possible.

Local Ollama chat can use deepseek-r1:latest, while local embeddings should use an embedding model such as embeddinggemma.

Chapter 2
Vector Databases: Searching by Meaning

A vector database stores text as vectors and helps search by meaning. A normal database is excellent for exact values, while a vector database is useful when the user asks for semantically related content.





Why?

Once you have many chunks, you need a fast way to retrieve the most relevant ones for a query.

What?

Vector databases support retrieval systems, semantic search, AI assistants, enterprise Q&A, and large-scale document search.

How?

Split documents into chunks, embed each chunk, store chunk plus vector plus metadata, embed the query, and retrieve the nearest matches.

Let's Code! - Embeddings Example

```
import requests
import numpy as np


chunks = [
    {"id": 1, "text": "Travel claims must be submitted within 15 days.", "source": "Travel Policy"},
    {"id": 2, "text": "Late claims require manager approval.", "source": "Travel Policy"},
    {"id": 3, "text": "Employees receive 12 casual leaves per year.", "source": "HR Policy"}
]


query = "How many days do I have to submit travel reimbursement?"


# ---------- Step 1: Embedding (using a dedicated embedding model) ----------
# Use a model that supports /api/embed, e.g., nomic-embed-text
EMBED_MODEL = "nomic-embed-text"   # make sure you pulled it: ollama pull nomic-embed-text


def get_embedding(text: str):
    response = requests.post(
        "http://localhost:11434/api/embed",
        json={"model": EMBED_MODEL, "input": text}
```

```
    )
    response.raise_for_status()
    return response.json()["embeddings"][0]


# Compute embeddings
chunk_embeddings = [get_embedding(chunk["text"]) for chunk in
chunks]
query_embedding = get_embedding(query)


# Cosine similarity
def cosine_similarity(a, b):
    a = np.array(a)
    b = np.array(b)
    return np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))


scores = [cosine_similarity(query_embedding, emb) for emb in
chunk_embeddings]
ranked = sorted(zip(chunks, scores), key=lambda x: x[1],
reverse=True)


# Print ranked chunks (optional)
print("Top Matching Chunks:\n")
for chunk, score in ranked:
    print(f"ID: {chunk['id']} | Source: {chunk['source']} | Score:
{score:.4f}")
    print(f"Text: {chunk['text']}\n")


# ---------- Step 2: Generate answer using DeepSeek ----------
top_chunk = ranked[0][0]   # highest scoring chunk


prompt = f"""
Answer the question using only the provided context.
If the answer is not found in the context, say: I do not know from
the provided context.


Context:
{top_chunk['text']}
```

```
Question:
{query}
"""

chat_response = requests.post(
    "http://localhost:11434/api/chat",
    json={
        "model": "deepseek-r1:latest",
        "messages": [{"role": "user", "content": prompt}],
        "stream": False
    }
)
chat_response.raise_for_status()

print("\n--- Answer from DeepSeek ---")
print(chat_response.json()["message"]["content"])
```

Output

Top Matching Chunks:

ID: 1 | Source: Travel Policy | Score: 0.6747

Text: Travel claims must be submitted within 15 days.

ID: 2 | Source: Travel Policy | Score: 0.5070

Text: Late claims require manager approval.

ID: 3 | Source: HR Policy | Score: 0.5021

Text: Employees receive 12 casual leaves per year.

--- Answer from DeepSeek ---

<think>

Okay, so I need to figure out how many days I have to submit travel reimbursement based on the context provided. The context says, "Travel claims must be submitted within 15 days." Hmm, that seems straightforward. So, if someone needs to submit a travel reimbursement claim, they have 15 days to do so. I don't see any other information here

about different timelines or additional days, so I think it's safe to go with 15 days. I don't recall any exceptions or specific conditions mentioned in the context, so I don't need to consider anything else. The answer should be 15 days.

</think>

The context states that travel claims must be submitted within 15 days. Therefore, you have 15 days to submit your travel reimbursement.

Answer: 15 days.

Key Takeaways

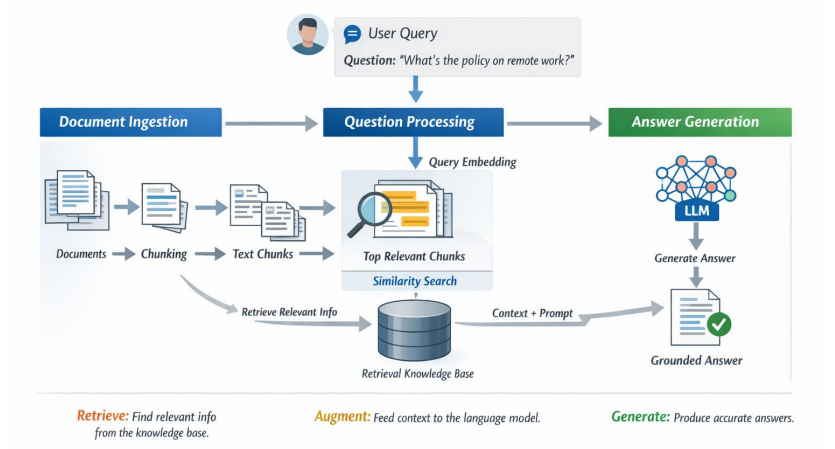
Vector databases search by meaning, not just exact keyword.

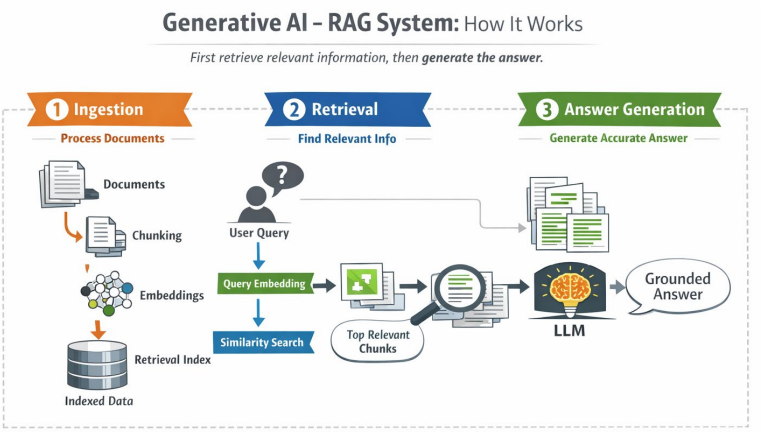
They store vectors plus metadata.

They are the retrieval foundation for RAG and semantic search systems.

Chapter 3
RAG: Retrieve First, Answer Second

RAG means Retrieval-Augmented Generation. In simple words: first retrieve relevant information, then generate the answer.





Why?

Many important answers already exist in documents. If the system guesses instead of retrieving, it may hallucinate.

What?

RAG combines chunking, embeddings, similarity search, prompt building, and language model generation.

How?

Split documents into chunks, embed them, retrieve the most relevant chunks for the user query, and instruct the model to answer only from that context.

Let's Code! – RAG Example

```
import requests
import numpy as np


documents = [
    "Leave policy: Employees receive 12 casual leaves per year.",
    "Travel policy: Claims must be submitted within 15 days of
travel completion.",
    "Insurance policy: Spouse and children are covered under the
company health plan."
]


query = "How many days do I have to submit my travel claim?"


def get_embedding(text: str):
    response = requests.post(
        "http://localhost:11434/api/embed",
        json={"model": "nomic-embed-text", "input": text}   # <--
changed model name
    )
    response.raise_for_status()
    return response.json()["embeddings"][0]
```

```
doc_embeddings = [get_embedding(doc) for doc in documents]
query_embedding = get_embedding(query)


def cosine_similarity(a, b):
    a = np.array(a)
    b = np.array(b)
    return np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))


scores = [cosine_similarity(query_embedding, emb) for emb in
doc_embeddings]
top_context = documents[int(np.argmax(scores))]


prompt = f"""
Answer the question using only the provided context.
If the answer is not found, say: I do not know from the provided
context.


Context:
{top_context}


Question:
{query}
"""


chat_response = requests.post(
    "http://localhost:11434/api/chat",
    json={
        "model": "deepseek-r1:latest",
        "messages": [{"role": "user", "content": prompt}],
        "stream": False
    }
)
chat_response.raise_for_status()


print("Retrieved Context:\n", top_context)
```

```
print("\nAnswer:\n", chat_response.json()["message"]["content"])
```

Output

Retrieved Context:

Travel policy: Claims must be submitted within 15 days of travel completion.

Answer:

You have 15 days to submit your travel claim.

Key Takeaways

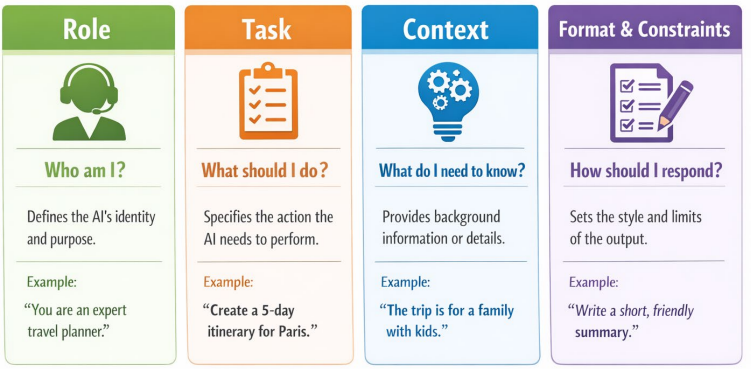
RAG means retrieve first, answer second.

RAG helps reduce hallucination by grounding responses in retrieved context.

OpenAI and Ollama both support the building blocks needed for simple RAG.

Chapter 4
Prompt Design: Better Instructions, Better Output

Prompt design means writing instructions in a clear and structured way so the model can produce better results.



Why?

Vague prompts often create vague answers. Better structure usually leads to better quality, clarity, and consistency.

What?

Prompt design improves tone, structure, completeness, readability, and alignment with the user's intent.

How?

Define the role, define the task, provide the context, specify the format, and add useful constraints.

Let's Code! – Prompt Example

```
import requests

notes = """
Customer wants faster loan approval.
Current process takes 48 hours.
Team suggested document verification API.
Compliance approval is mandatory.
"""

good_prompt = f"""
You are a business analyst.
Summarize the following meeting notes in exactly 3 sections:
1. Problem
2. Proposed Solution
3. Next Step

Use simple business language.

Notes:
{notes}
"""

response = requests.post(
    "http://localhost:11434/api/chat",
    json={
        "model": "deepseek-r1:latest",
        "messages": [{"role": "user", "content": good_prompt}],
        "stream": False
    }
)
response.raise_for_status()

print(response.json()["message"]["content"])
```

Output

Problem:

The customer is requesting a faster loan approval process.

Proposed Solution:

The team has suggested implementing a document verification API to streamline the approval process.

Next Step:

The next step is to implement compliance approval as a mandatory step to ensure the process is efficient and compliant.

Key Takeaways

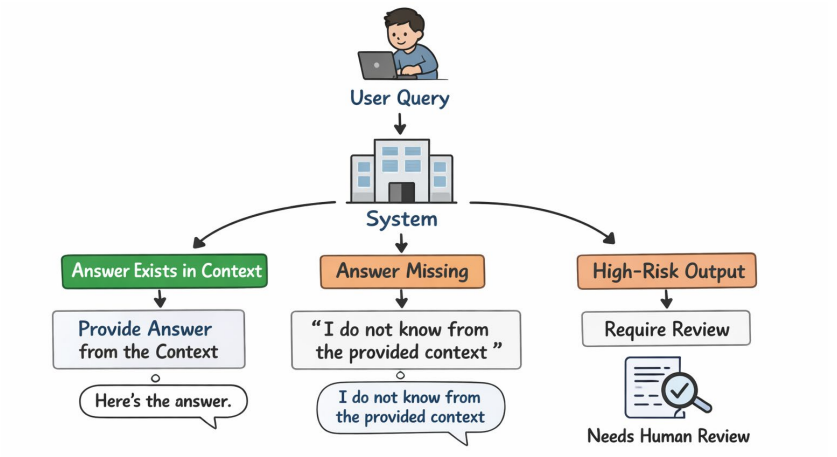
Prompt design is structured instruction writing.

Role, task, context, format, and constraints strongly affect output.

Both cloud and local LLMs benefit from better prompt structure.

Chapter 5
Hallucination Control: Reducing Confident Mistakes

Hallucination happens when a model gives an answer that sounds correct, but is actually unsupported, wrong, or invented.



Why?

Language models generate likely text, not guaranteed truth. That is why grounding and verification matter.

What?

Hallucination can appear as fake facts, invented clauses, wrong dates, made-up references, or unsupported explanations.

How?

Use RAG, instruct the model to answer only from context, allow "I do not know," and add human review for important cases.

Let's Code! – Example

```
import requests


context = "Travel policy: Claims must be submitted within 15 days of
travel completion."

question_1 = "How many days do I have to submit my travel claim?"

question_2 = "What is the yearly insurance premium?"


def ask_safe(question: str):

    prompt = f"""

Answer the question using only the provided context.

If the answer is not found in the context, say exactly:

I do not know from the provided context.


Context:

{context}


Question:

{question}

"""

    response = requests.post(

        "http://localhost:11434/api/chat",

        json={

            "model": "deepseek-r1:latest",

            "messages": [{"role": "user", "content": prompt}],

            "stream": False

        }

    )

    response.raise_for_status()

    return response.json()["message"]["content"]


print("Q1:", ask_safe(question_1))

print("\nQ2:", ask_safe(question_2))
```

Output

Q1:

The answer is 15 days based on the context provided.

Q2:

I do not know from the provided context.

Key Takeaways

Hallucination is a practical risk in LLM systems.

Grounding and stricter prompts reduce that risk.

I do not know" is often safer than an invented answer.

Chapter 6
Evaluation of LLM Answers: Check Before You Trust

Evaluation means checking whether the answer is actually good—not just fluent, but correct, relevant, complete enough, clear, and safe.

Why?

A polished answer can still be wrong. Evaluation helps teams detect weak, incomplete, or risky outputs before they are used.

What?

You can evaluate factual correctness, context alignment, completeness, readability, safety, and instruction following.

How?

Use a simple rubric or ask a model to score an answer against a reference. Start small and make evaluation a regular habit.

Let's Code! – Prompt Example

```
import requests


question = "How many days do I have to submit my travel claim?"

reference = "Travel claims must be submitted within 15 days of
travel completion."

answer = "Travel claims must be submitted within 15 days of travel
completion."


eval_prompt = f"""

You are evaluating an LLM answer.


Question:

{question}


Reference Answer:

{reference}


Model Answer:

{answer}


Score the Model Answer from 1 to 5 on:

1. Correctness

2. Relevance

3. Completeness

4. Clarity

5. Safety


Then provide one short overall comment.

"""


response = requests.post(

    "http://localhost:11434/api/chat",

    json={

        "model": "deepseek-r1:latest",
```

```
        "messages": [{"role": "user", "content": eval_prompt}],
        "stream": False
    }
)
response.raise_for_status()


print(response.json()["message"]["content"])
```

Output

The model answer receives a perfect score across all evaluation criteria:

1. Correctness: *5/5*

2. Relevance: *5/5*

3. Completeness: *5/5*

4. Clarity: *5/5*

5. Safety: *5/5*

Overall Comment: The model answer is flawless, providing accurate, relevant, and clear information that is safe to use. It effectively addresses the user's query without any issues.

```
def keyword_eval(answer: str, expected_keywords: list[str]) -> dict:
    matched = [kw for kw in expected_keywords if kw.lower() in
answer.lower()]
    return {
        "score": len(matched),
        "total": len(expected_keywords),
        "matched_keywords": matched
    }

answer = "Travel claims must be submitted within 15 days of travel
completion."
expected_keywords = ["travel", "15 days", "submitted", "completion"]

print(keyword_eval(answer, expected_keywords))
```

Output

{'score': 4, 'total': 4, 'matched_keywords': ['travel', '15 days', 'submitted', 'completion']}

Key Takeaways

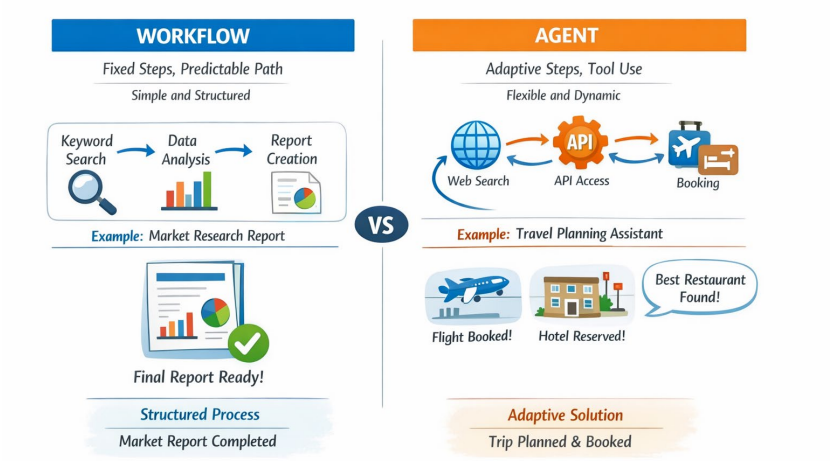
Evaluation is necessary because fluent text is not automatically reliable.

Practical evaluation can start with simple rubrics.

Both OpenAI and Ollama can help evaluate model outputs.

Chapter 7
Agents vs Workflows: Choosing the Right Architecture

A workflow follows a fixed sequence of steps. An agent is a system that can choose its next step dynamically based on the goal, the available tools, and what it discovers during execution.



Why?

Not every problem needs an agent. Sometimes fixed workflows are safer, cheaper, faster, and easier to test. Agents are useful when the next step depends on discovery or when multiple tools may be needed.

What?

Use workflows for predictable processes. Use agents for open-ended tasks such as research, meeting preparation, multi-source analysis, or adaptive orchestration.

How?

First ask whether the path is already known. If yes, begin with workflow. If the path depends on findings, consider an agent with clear boundaries and controlled tools.

Let's Code! – Prompt Example

```
import requests


goal = "Prepare me for tomorrow's client meeting about loan
onboarding modernization."


prompt = f"""
You are an AI planning assistant.
Given the user goal below, decide whether this should be handled as:
1. a workflow
2. an agent


Then produce a short step-by-step plan.


User Goal:
{goal}
"""
response = requests.post(
    "http://localhost:11434/api/chat",
    json={
        "model": "deepseek-r1:latest",
        "messages": [{"role": "user", "content": prompt}],
        "stream": False
    }
)
response.raise_for_status()


print(response.json()["message"]["content"])
```

Output

The preparation for tomorrow's client meeting about loan onboarding modernization should be handled as a workflow. Here's a structured step-by-step plan:

1. Research the Client's Current Process:

- Investigate the client's existing loan onboarding systems and processes.

- Identify their pain points and areas for improvement.

2. Gather Relevant Data and Insights:

- Collect recent reports, case studies, or data relevant to their onboarding process.

- Understand their operational challenges and goals.

3. Prepare Talking Points:

- Organize key insights and recommendations.

- Anticipate potential questions and prepare concise answers.

4. Create Visual Aids:

- Develop slides or charts that highlight key points and data.

- Ensure visuals are clear and support the presentation.

5. Practice the Presentation:

- Rehearse the presentation multiple times to ensure smooth delivery.

- Time each section to adhere to the meeting duration.

6. Review and Finalize:

- Check all materials for clarity and accuracy.

- Ensure all points are covered and the presentation is polished.

By following this workflow, the preparation is systematic and ensures all aspects are covered efficiently.

```
def workflow_process(text: str) -> str:
    return f"Workflow output: summarize the first 80 chars ->
{text[:80]}"
```

```
def agent_process(goal: str) -> str:
    if "meeting" in goal.lower():
        return "Agent plan: retrieve notes -> search updates ->
prepare briefing -> generate speaker notes"
    return "Agent plan: default structured analysis"


print(workflow_process("This weekly note contains release updates,
risk items, and deployment issues."))
print(agent_process("Prepare me for tomorrow's client meeting"))
```

Output

Workflow output: summarize the first 80 chars -> This weekly note contains release updates, risk items, and deployment issues.

Agent plan: retrieve notes -> search updates -> prepare briefing -> generate speaker notes

Key Takeaways

Workflows are fixed and predictable.

Agents are adaptive and useful for open-ended tasks.

Many business problems should still begin with a workflow rather than an agent.

Chapter 8
When Not to Use an LLM: Use Intelligence Wisely

An LLM is powerful, but it is not the right answer for every problem. Some tasks are better solved with normal code, SQL, rules engines, or deterministic systems.

Why?

LLMs can be slower, more expensive, less deterministic, and harder to test than traditional software. Using them where they are not needed adds cost and risk without creating value.

What?

Do not use an LLM for exact arithmetic, simple validation, stable filtering, deterministic decision tables, or cases requiring exact reproducibility.

How?

Use the simplest correct solution first. Choose an LLM when the main value comes from language understanding, summarization, transformation, or explanation.

Let's Code! – Example

```
import requests


task = "Check whether customer age is greater than 18."


prompt = f"""
Decide whether the following task should use:
1. simple code / rules
2. SQL / database logic
3. an LLM
```

```
Explain the best choice in simple words.

Task:
{task}
"""

response = requests.post(
    "http://localhost:11434/api/chat",
    json={
        "model": "deepseek-r1:latest",
        "messages": [{"role": "user", "content": prompt}],
        "stream": False
    }
)
response.raise_for_status()

print(response.json()["message"]["content"])
```

Output

Use simple code for this task because it's a simple comparison that doesn't need advanced operations or database access.

Key Takeaways

Do not force an LLM into every problem.

Use normal software for fixed, deterministic tasks.

Use LLMs when language understanding or generation is central to the task.

Responsible AI: Building Powerful Systems with Care, Fairness, and Trust

Artificial Intelligence is powerful, but power alone is not enough. A model that produces a highly accurate output is not automatically a good model if it creates unfair results, exposes sensitive data, misleads users, or operates without accountability. This is why Responsible AI is now one of the most important topics in modern AI practice.

Responsible AI means designing, building, deploying, and governing AI systems in a way that is fair, safe, transparent, reliable, and aligned with human needs. It is not a separate subject that begins after technical work is complete. It should be part of the design process from the beginning.

A responsible AI system should aim to be fair, so it does not systematically disadvantage certain users or groups. It should protect privacy and sensitive data. It should be transparent enough that users and builders understand its role and limitations. It should be reliable under expected conditions. It should reduce hallucinations and unsupported outputs where accuracy matters. It should support human review in high-impact decisions. It should also be governed in a way that makes ownership, monitoring, and accountability clear.

There are many ways AI systems can go wrong. A model may inherit historical bias from its training data. A language model may generate false but confident answers. A chatbot may expose information it should not reveal. A recommendation system may amplify harmful patterns. A document assistant may answer beyond the provided evidence. A system may continue operating long after its data has become outdated. These failures are not always caused by bad intentions. Very often, they happen because teams focus only on building capability and not enough on managing risk.

Bias and fairness are among the most important concerns. If an AI model is trained on unbalanced or historically unfair data, it may repeat those patterns in its decisions. Privacy is another major concern because many AI systems work with sensitive personal, medical, financial, or organizational information. Security also matters, especially in the age of tool-using AI systems and external APIs. Hallucination risk matters

in Generative AI because users may trust a confident answer even when it is unsupported. This is why grounding, evaluation, and human oversight are essential.

Responsible AI also requires discipline after deployment. Monitoring is not optional. Models drift. Data changes. Business conditions evolve. Risks emerge over time. A responsible team does not treat deployment as the end of the process. It treats deployment as the start of operational accountability.

For learners, the message is simple but important: do not measure AI quality only by output performance. Measure it by trustworthiness as well. A truly successful AI system is not only smart. It is safe, fair, explainable, governed, and useful in the real world.

The real art of AI is not just making systems more intelligent. It is making them more responsible.

Appendix A - Math Cheat Sheet (Only What You Use)

You do NOT need advanced math to start deep learning. These are the only parts you use daily.

1) Vector and dot product

Dot product

```
dot(x, w) = x1*w1 + x2*w2 + ... + xn*wn
```

Plain meaning: combine many inputs into one number using weights.

2) Activation functions

Common activations

```
ReLU(z) = max(0, z)
Sigmoid(z) = 1/(1+e^(-z))
Softmax(z_i) = e^(z_i) / Σ e^(z_j)
```

ReLU is the default for hidden layers. Sigmoid is for binary probability. Softmax is for multi-class.

3) Loss functions

Common losses

```
Regression: MSE = (1/N) Σ (y_true - y_pred)^2
Binary: BCE = -(y*log(p) + (1-y)*log(1-p))
```

Loss is your 'mistake meter'. Training tries to reduce loss.

4) Gradient descent

Update rule

```
parameter_new = parameter_old - learning_rate * gradient
```

Gradient tells direction of increase; subtracting it reduces loss.

Appendix B — Debugging and Common Errors

Shape errors: Most common. Print shapes of X and y. Dense expects 2D input, LSTM expects 3D, CNN expects 4D.

Loss becomes NaN: learning rate too high or data has extreme values. Normalize inputs.

Accuracy stuck at ~50%: wrong labels, wrong loss, or too little data.

Overfitting: training accuracy high but validation low. Use less model capacity, dropout, early stopping, more data.

Model works in notebook but fails in API: preprocessing mismatch. Save and reuse mean/std and tokenizers.

Appendix C — Glossary

Epoch: One full pass over the training dataset.

Batch size: How many rows you process before updating weights.

Weight: A knob that decides how important an input feature is.

Bias: A free extra push added to the neuron output.

Loss: Number that tells how wrong the model is.

Optimizer: Algorithm that updates weights to reduce loss (SGD/Adam).

Overfitting: Model memorizes training data and performs poorly on new data.

Underfitting: Model is too simple and can't learn the pattern.

Inference: Using the trained model to predict on new data.

Deployment: Making the model available as a service/app for real users.
The best way to learn deep learning is: run code, change one thing, observe, repeat.