SREENIVASA INSTITUTE OF TECHNOLOGY AND MANAGEMENT STUDIES, CHITTOOR
(AUTONOMOUS)

Approved by AICTE, New Delhi, Affiliated to JNTUA, Ananthapuramu.

LECTURE NOTES

Subject Name : DEEP LEARNING

Year / Branch : IV/CSE

Regulation :R23

PreparedBy: N.Vijaykumar

1. Vision, Mission, POs, PEOs, PSOs (If any)

DEPARTMENT VISION

To contribute for the society through excellence in Computer Science and Engineering with

a deep passion for wisdom, culture and values.

DEPARTMENT MISSION

M1: Provide congenial academic ambience with necessary infrastructure and learning resources.

M2: Inculcate confidence to face and experience new challenges from industry and society.

M3: Ignite the students to acquire self-reliance in State-of-the-Art Technologies

M4: Foster Enterprising spirit among students.

PROGRAMME EDUCATIONAL OBJECTIVES (PEOs)

PEO1: Excel in Computer Science and Engineering program through quality studies, enabling success in computing industry.(Professional Competency).

PEO2: Surpass in one's career by critical thinking towards successful services and growth of the organization, or as an entrepreneur or in higher studies. (Successful Career Goals).

PEO3: Enhance knowledge by updating advanced technological concepts for facing the rapidly changing world and contribute to society through innovation and creativity (Continuing Education and Contribution to Society).

Program Specific Outcomes (PSOs)

PSO1: Have Ability to understand, analyse and develop computer programs in the areas like algorithms, system software, web design, big data analytics, and networking.

PSO2: Deploy the modern computer languages, environment, and platforms in creating innovative products and solutions.

2. Syllabus Contents (In-line with OBE)

UNIT- I LINEAR ALGEBRA, PROBABILITY AND NUMERICAL COMPUTATION Lecture 9Hrs

Linear Algebra: Scalars, Vectors, Matrices and Tensors, Matrix operations, types of matrices, Norms, Eigen decomposition, Singular Value Decomposition, Principal Components Analysis.

Probability and Information Theory: Random Variables, Probability Distributions, Marginal Probability, Conditional Probability, Expectation, Variance and Covariance, Bayes' Rule, Information Theory.

Numerical Computation: Overflow and Underflow, Gradient-Based Optimization, Constrained Optimization, Linear Least Squares.

UNIT-II Machine Learning And Deep Feed Forward Networks Lecture 9 Hrs

Machine Learning: Basics and Under fitting, Hyper parameters and Validation Sets, Estimators, Bias and Variance, Maximum Likelihood, Bayesian Statistics, Supervised and Unsupervised Learning, Stochastic Gradient Descent, Challenges

Motivating Deep Learning.
Deep Feed forward Networks: Learning XOR, Gradient-Based Learning, Hidden Units, Architecture Design, Back-Propagation and other Differentiation Algorithms.

UNIT-III Regularization And Optimization Of Dl Models Lecture 9Hrs

Regularization for Deep Learning: Parameter Norm Penalties, Norm Penalties as Constrained Optimization, Regularization and Under-Constrained Problems, Dataset Augmentation, Noise Robustness, Semi-Supervised Learning, Multi-Task Learning, Early Stopping, Parameter Tying and Parameter Sharing, Sparse Representations, Bagging and Other Ensemble Methods, Dropout, Adversarial Training, Tangent Distance, Tangent Prop and Manifold Tangent Classifier.

Optimization for Training Deep Models: Pure Optimization, Challenges in Neural Network Optimization, Basic Algorithms, Parameter Initialization Strategies, Algorithms with Adaptive Learning Rates, Approximate Second-Order Methods, Optimization Strategies and Meta- Algorithms.

UNIT-IV Convolutional Networks Lecture 9 Hrs

Natural Language for Communication: Phrase structure grammars, Syntactic Analysis, Augmented Grammars and semantic Interpretation, Machine Translation, Speech Recognition Perception: Image Formation, Early Image Processing Operations, Object Recognition by appearance, Reconstructing the 3D World, Object Recognition from Structural information, Using Vision.

UNIT-V Sequence Modeling Lecture 9Hrs

Robotics: Introduction, Robot Hardware, Robotic Perception, planning to move, planning uncertain movements, Moving, Robotic software architectures, application domains Philosophical foundations: Weak AI, Strong AI, Ethics and Risks of AI, Agent Components, Agent Architectures, Are we going in the right direction, What if AI does succeed.

3. Objectives of the Subject

This course is designed to:

COURSE EDUCATIONAL OBJECTIVES:

1. Demonstrate the major technology trends driving Deep Learning
2. Build, train, and apply fully connected deep neural networks
3. Implement efficient (vectorized) neural networks
4. Analyze the key parameters and hyper parameters in a neural network's architecture
5. Demonstrate the sequence modeling and Networks of Deep Learning

4. Introduction Part

a) Clearly Define the Contents and Key Terminologies Deep Learning (DL) is a branch of Artificial Intelligence (AI) and a subset of Machine Learning (ML) that enables computers to learn from large amounts of data by using artificial neural networks with multiple hidden layers. It is inspired by the structure and functioning of the human brain,

where interconnected neurons process and transmit information.Key Terminologies. The fundamental building block of deep learning is the Artificial Neural Network (ANN). When multiple hidden layers are added to an ANN, it becomes a Deep Neural Network (DNN), enabling the model to learn increasingly complex patterns and representations from data. Today, deep learning is widely used across various industries, including healthcare, finance, agriculture, education, manufacturing, cybersecurity, and robotics. In healthcare, for example, deep learning assists in disease diagnosis from medical images, while in finance it helps detect fraudulent transactions and predict market trends.

Key Terms and Definitions in Deep Learning

Key Term	Definition
Artificial Intelligence (AI)	The branch of computer science that enables machines to perform tasks requiring human intelligence, such as learning, reasoning, and decision-making.
Machine Learning (ML)	A subset of AI that enables computers to learn from data and improve performance without being explicitly programmed.
Deep Learning (DL)	A subset of machine learning that uses deep neural networks with multiple hidden layers to automatically learn patterns from data.
Artificial Neural Network (ANN)	A computational model inspired by the human brain, consisting of interconnected neurons organized into layers.
Deep Neural Network (DNN)	A neural network with multiple hidden layers that can learn complex representations from data.
Neuron (Node)	The basic processing unit of a neural network that receives inputs, performs computations, and produces an output.
Input Layer	The first layer of a neural network that receives input data for processing.
Hidden Layer	Intermediate layers between the input and output layers where feature extraction and learning occur.
Output Layer	The final layer of a neural network that produces the prediction or classification result.
Weight	A numerical value assigned to a connection between neurons that determines the importance of an input.
Bias	An additional parameter added to the weighted sum of inputs to improve the model's learning capability.
Activation Function	A mathematical function that determines whether a neuron should be activated and introduces non-linearity into the model.
Loss Function	A function that measures the difference between the predicted output and the actual target value.
Optimizer	An algorithm that updates the model's weights and biases to minimize

	the loss function during training.
Forward Propagation	The process of passing input data through the neural network to generate predictions.
Backpropagation	The process of computing gradients and updating weights to reduce prediction errors.
Gradient Descent	An optimization algorithm used to minimize the loss function by iteratively updating model parameters.
Learning Rate	A hyperparameter that controls the size of weight updates during training.
Epoch	One complete pass of the entire training dataset through the neural network.
Batch Size	The number of training samples processed together before updating the model's weights.
Iteration	A single update of model parameters using one batch of training data.
Training Data	The dataset used to train the deep learning model.
Validation Data	A dataset used during training to tune hyperparameters and evaluate model performance.
Test Data	A separate dataset used to evaluate the final performance of the trained model.
Feature	An individual measurable property or characteristic used as input to a model.
Feature Extraction	The process of identifying important characteristics from raw data for learning.
Hyperparameter	A parameter set before training begins, such as learning rate, batch size, or number of epochs.
Overfitting	A condition where the model performs well on training data but poorly on unseen data.
Underfitting	A condition where the model is too simple to capture the underlying patterns in the data.
Regularization	Techniques used to reduce overfitting and improve model generalization.
Dropout	A regularization technique that randomly deactivates neurons during training to prevent overfitting.
Convolutional Neural Network (CNN)	A deep learning architecture designed primarily for image and video processing tasks.
Recurrent Neural Network (RNN)	A neural network designed for processing sequential data such as text, speech, and time-series data.

Long Short-Term Memory (LSTM)	A specialized type of RNN that effectively learns long-term dependencies in sequential data.
Transformer	A deep learning architecture that uses attention mechanisms for efficient processing of sequential data, especially in NLP.
Attention Mechanism	A technique that enables a model to focus on the most relevant parts of the input while making predictions.
Convolution	A mathematical operation used in CNNs to extract spatial features from input data.
Pooling	A downsampling operation that reduces the spatial dimensions of feature maps while retaining important information.
Feature Map	The output generated after applying convolution filters to the input data.
Classification	A supervised learning task where the model predicts predefined categories or classes.
Regression	A supervised learning task that predicts continuous numerical values.
Inference	The process of using a trained model to make predictions on new, unseen data.

b) Provide Conceptual Depth with Suitable Illustrations

Deep Learning (DL) is a subset of **Machine Learning (ML)** that uses **Artificial Neural Networks (ANNs)** with multiple hidden layers to automatically learn complex patterns from large datasets. It is inspired by the structure and functioning of the human brain, where neurons receive, process, and transmit information.

Unlike traditional machine learning, where features must be manually extracted, deep learning automatically learns low-level and high-level features from raw data. This makes it highly effective for applications such as image recognition, speech recognition, natural language processing, medical diagnosis, autonomous driving, and recommendation systems.

The main advantage of deep learning is its ability to improve accuracy as the amount of training data increases. Deep learning models require significant computational power, often using **Graphics Processing Units (GPUs)** or **Tensor Processing Units (TPUs)** to process millions of parameters efficiently.

c) Introduce Standard Symbols and Representation (if applicable)

3. **1. Standard Symbols Used in Deep Learning**

Symbol	Representation	Meaning
(x)	Input	Input feature or data sample
(x_1, x_2, ..., x_n)	Input Features	Individual input variables

Symbol	Representation	Meaning
(X)	Input Matrix	Collection of training samples
(y)	Actual Output	True target value (label)
(\hat{y})	Predicted Output	Model prediction
(w)	Weight	Strength of connection between neurons
(w_i)	Weight of ith Input	Importance assigned to input (x_i)
(b)	Bias	Additional parameter that shifts the activation
(z)	Weighted Sum	Linear combination of inputs and weights
(a)	Activation	Output of a neuron after applying the activation function
(f(z))	Activation Function	Converts weighted sum into neuron output
(L)	Loss Function	Measures prediction error
(J(W,b))	Cost Function	Average loss over the training dataset
(\eta) or (\alpha)	Learning Rate	Step size used during weight updates
(\nabla)	Gradient	Direction of maximum increase in loss
(\delta)	Error Term	Error propagated during backpropagation
(m)	Number of Training Samples	Total examples in the dataset
(n)	Number of Features	Number of input variables
(E)	Epoch	One complete pass through the training dataset

b) Highlight Why the Particular Content is Important

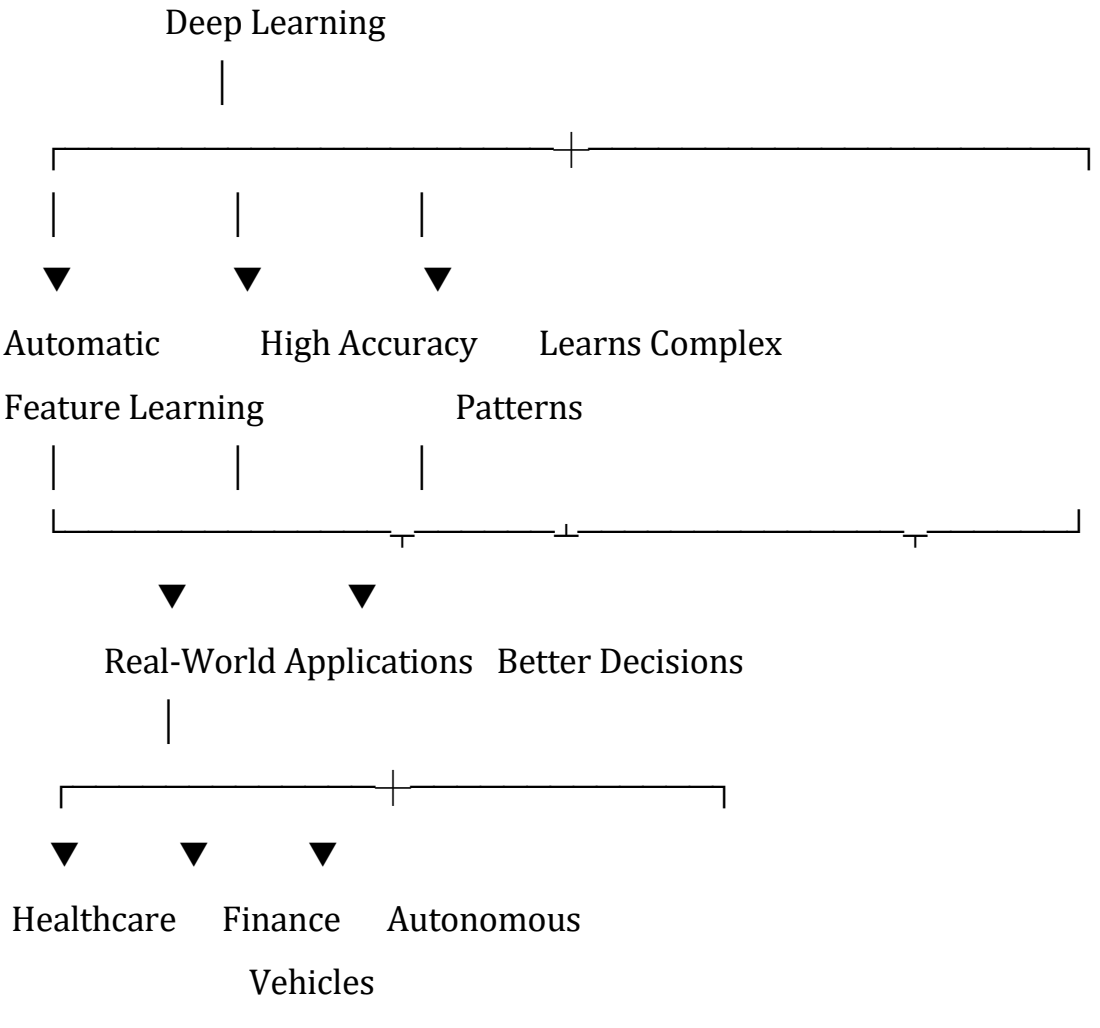
Understanding the importance of deep learning is essential because it forms the foundation of many modern Artificial Intelligence (AI) applications. Deep learning enables computers to automatically learn complex patterns from large datasets, making it one of the most powerful techniques for solving real-world problems..

Studying AI helps students understand:

- Intelligent decision making
- Autonomous systems
- Smart healthcare technologies
- Intelligent transportation
- Data-driven problem solving
- Human-computer interaction
- Future technologies

c) Provide Explanation with Suitable Figures / Pictures

Components of Artificial Intelligence



d) Include Real-World Examples

Application	Example
Voice Assistant	Siri, Google Assistant, Alexa
Recommendation Systems	Netflix, Amazon, YouTube
Healthcare	Disease Diagnosis
Banking	Fraud Detection
Education	Intelligent Tutoring Systems
Speech Recognition	Voice assistants
Face Recognition	Face Unlock in smartphones.
Robotics	Industrial Robots
Cyber Security	Intrusion Detection
Smart Homes	Home Automation

e) Mention Advantages and Disadvantages

Advantages

- Automatic feature extraction
- High prediction accuracy
- Handles large and complex datasets
- Learns nonlinear relationships
- Supports multiple data types
- End-to-end learning
- Scalable for real-world applications
- Improves with more data
- Enables intelligent automation
- Widely applicable across industries

Disadvantages

- Requires large labeled datasets
- High computational and memory requirements
- Long training time
- Difficult to interpret (black-box)
- Risk of overfitting
- Complex architecture design
- Requires hyperparameter tuning
- Expensive hardware
- High energy consumption
- Less effective for small datasets

f) Explain Applications and Limitations

Applications

1. Healthcare
2. Computer Vision
3. Natural Language Processing (NLP).
4. Speech Recognition.

5. Autonomous Vehicles.
6. Recommendation Systems.
7. Finance
8. Cybersecurity
9. Agriculture
10. Manufacturing
11. Robotics
12. Education
13. E-commerce.
14. Entertainment
15. Weather Forecasting

Limitations

1. Large data requirement
2. High computational cost
3. Long training time
4. Black-box nature
5. Overfitting
6. High memory usage
7. Expensive hardware
8. Hyperparameter tuning
9. Sensitive to data quality
10. Difficult to debug

g) Provide Comparison Tables

Feature	Machine Learning	Deep Learning
Definition	Learns patterns from data using algorithms.	Uses deep neural networks with multiple hidden layers.
Feature Extraction	Manual feature extraction is often required.	Automatic feature extraction.
Data Requirement	Works with smaller datasets.	Requires large datasets.
Accuracy	Moderate accuracy.	Generally higher accuracy.
Training Time	Faster.	Slower.
Computational Power	Lower.	Higher (GPU/TPU often required).
Applications	Simple prediction tasks.	

h) Include Relevant Case

Studies Case Study 1: Autonomous

Vehicles

Self-driving cars use AI techniques such as computer vision, sensor fusion, reinforcement learning, and path planning to detect obstacles, recognize traffic signs, and navigate safely.

AI Concepts Used

- Intelligent Agents
- Search Algorithms
- Computer Vision
- Reinforcement Learning
- Robotics

Case Study 2: Medical Diagnosis System

AI systems analyze medical images and patient data to assist doctors in detecting diseases such as cancer, diabetic retinopathy, and pneumonia with high accuracy.

Benefits

- Faster diagnosis
- Improved accuracy
- Reduced workload
- Early disease detection

i) Safety & Standards Component

Artificial Intelligence systems must follow ethical guidelines and international standards to ensure safe, reliable, and trustworthy operation.

- AI Safety Considerations
- Data privacy and security
- Fairness and bias mitigation
- Transparency and explainability
- Human oversight
- Robustness against cyberattacks
- Responsible AI deployment
- Accountability in AI decision-making

Relevant Standards

Standard	Purpose
ISO/IEC 22989	Artificial Intelligence concepts and terminology
ISO/IEC 23053	to guide the machine learning system lifecycle.
NIST AI RMF	To assess and manage risks
IEEE 7000 Series	Address fairness, transparency, and patient privacy
ONNX	Deploy the trained model across different deep learning platforms without retraining

DEEP LEARNING

SYLLABUS

UNIT- I LINEAR ALGEBRA, PROBABILITY AND NUMERICAL COMPUTATION Lecture 9Hrs

Linear Algebra: Scalars, Vectors, Matrices and Tensors, Matrix operations, types of matrices, Norms, Eigen decomposition, Singular Value Decomposition, Principal Components Analysis.

Probability and Information Theory: Random Variables, Probability Distributions, Marginal Probability, Conditional Probability, Expectation, Variance and Covariance, Bayes' Rule, Information Theory.

Numerical Computation: Overflow and Underflow, Gradient-Based Optimization, Constrained Optimization, Linear Least Squares.

Chapter 1

Mathematical Foundations of Deep Learning

1.1 Introduction to Deep Learning

Deep Learning is a subfield of machine learning that utilizes hierarchical layers of artificial neural networks to model and understand complex patterns in data. These networks learn representations of data through multiple processing layers, enabling them to excel at tasks such as image recognition, natural language processing, and autonomous decision-making. The mathematical foundation of deep learning lies in linear algebra, calculus, and probability theory, which provide the tools for understanding and manipulating the data structures and operations used in neural networks.

1.2 Scalars, Vectors, Matrices, and Tensors

1.2.1 Scalars

Definition 1.2.1 (Scalar)**.** *A **scalar** is a single number, typically a real number denoted by* $x \in \mathbb{R}$ *or a complex number* $z \in \mathbb{C}$*. In deep learning, scalars represent single values like learning rates, biases, or loss values.*

1.2.2 Vectors

Definition 1.2.2 (Vector)**.** *A **vector** is an ordered array of n numbers (scalars). For* $n \in \mathbb{N}$*, an n-dimensional vector is:*

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^n$$

Each x_i *is called a **component** or **element**. Vectors represent features, gradients, or pa-rameters in neural networks.*

A vector is an ordered/indexed array of numbers, often representing coordinates in space or features in data. Denoted as:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^n$$

Example: Feature vector for house pricing: $\mathbf{x}$ = [size, bedrooms, age]$^\top$.

1.2.3 Matrices

Definition 1.2.3 (Matrix)**.** *A **matrix** is a 2-dimensional array of numbers with m rows and n columns:*

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \in \mathbb{R}^{m \times n}$$

Matrices represent linear transformations, weight matrices in neural networks, or datasets.

Matrices represent linear transformations, such as weights in a neural network layer.

1.2.4 Tensors

Definition 1.2.4 (Tensor)**.** *A **tensor** is a multi-dimensional array that generalizes scalars (0D), vectors (1D), and matrices (2D). An n-dimensional tensor $\in \mathbb{R}^{d_1 \times d_2 \times \cdots \times d_n}$ has n indices. In deep learning, tensors are used for batch processing, convolutional filters, and higher-dimensional data.*

Tensors generalize vectors and matrices to higher dimensions. A n-dimensional tensor has n indices. In deep learning, tensors are used for multi-dimensional data like images (height × width × channels).

1.3 Tensors:Mathematical Foundation and Deep Learning Applications

1.3.1 Tensor Definition and Notation

Formal Definition

A **tensor** is a mathematical object that generalizes scalars, vectors, and matrices to arbitrary dimensions. It provides a unified framework for representing multi-dimensional data and relationships among variables.

Formally, an N-th order (or N-way) tensor is an element of the Cartesian product of N vector spaces, typically represented as a multidimensional array:

$$\mathrm{T} \in \mathrm{R}^{I_1 \times I_2 \times \cdots \times I_N}$$

Each element of the tensor is indexed by N indices:

$$\mathrm{T} = [t_{i_1 i_2 \ldots i_N}], \quad \text{where } i_k = 1, 2, \ldots, I_k \ \forall k = 1, 2, \ldots, N$$

Here:

- N denotes the **order** (or rank) of the tensor.
- I_k represents the dimensionality along the k-th mode.
- $t_{i_1 i_2 \ldots i_N}$ is a real-valued scalar element.

In deep learning, tensors serve as the fundamental data structure for inputs, parameters (weights and biases), intermediate activations, and outputs of neural networks.

Tensor Notation

To maintain clarity when working with high-dimensional objects, standard tensor notation conventions are followed:

- **Scalars:** Denoted by lowercase italic letters, e.g., $a, b \in \mathrm{R}$
- **Vectors:** Denoted by bold lowercase letters, e.g., $\mathbf{v} \in \mathrm{R}^n$
- **Matrices:** Denoted by bold uppercase letters, e.g., $\mathbf{A} \in \mathrm{R}^{m \times n}$
- **Tensors:** Denoted by calligraphic letters, e.g., $\mathrm{T}, \mathrm{X}, \mathrm{W}$

Key tensor-related terms are defined as follows:

- **Order (Rank):** The number of indices required to uniquely identify an element of the tensor. For example, a matrix has order 2, while a color image tensor has order 3.
- **Mode-k Fibers:** Mode-k fibers are higher-order generalizations of matrix rows and columns. They are obtained by fixing all indices except the k-th index. For instance, in a third-order tensor T_{ijk}:
  - Mode-1 fibers vary along index i
  - Mode-2 fibers vary along index j
  - Mode-3 fibers vary along index k
- **Slices:** Slices are lower-order tensors obtained by fixing one or more indices. For a third-order tensor:
  - Fixing i yields a matrix slice $\mathrm{T}_{i::}$
  - Fixing j yields $\mathrm{T}_{:j:}$
  - Fixing k yields $\mathrm{T}_{::k}$
- **Einstein Summation Notation:** A compact notation that implies summation

over repeated indices. For example:

$$c_i = a_{ij} b_j$$

implicitly represents:

$$c_i = \sum_j a_{ij} b_j$$

This notation is widely used in tensor algebra and deep learning frameworks for ex-pressing efficient computations.

1.3.2 Tensor Representation and Examples Tensor Orders and Examples

Tensors can be classified based on their order, which corresponds to the number of dimensions involved:

- **Order-0 Tensor (Scalar):** A scalar is a single numerical value with no associated direction or dimension:

  $$a \in \mathrm{R}$$

  Examples include learning rate values, loss values, or bias terms.

- **Order-1 Tensor (Vector):** A vector is a one-dimensional array:

  $$\mathbf{v} = [v_1, v_2, \ldots, v_n]^T \in \mathrm{R}^n$$

  Vectors are commonly used to represent feature vectors, word embeddings, or neuron activations.

- **Order-2 Tensor (Matrix):** A matrix is a two-dimensional tensor:

  $$\mathbf{A} \in \mathrm{R}^{m \times n}$$

  Matrices are widely used for linear transformations, weight matrices in neural networks, and covariance representations.

- **Order-3 Tensor:** A third-order tensor has three dimensions:

  $$\mathrm{T} \in \mathrm{R}^{I \times J \times K}$$

  An important example is a color image represented as:

  $$\text{Height} \times \text{Width} \times \text{Channels (RGB)}$$

  Here, each pixel location contains a 3-dimensional vector corresponding to color inten-sities.

- **Order-4 Tensor:** A fourth-order tensor extends this concept further:

$$\mathrm{T} \in \mathrm{R}^{I \times J \times K \times L}$$

In deep learning, this is commonly used to represent a batch of images:

$$\text{Batch Size} \times \text{Height} \times \text{Width} \times \text{Channels}$$

This representation enables parallel processing of multiple samples during training and inference.

Higher-order tensors (order > 4) also arise in applications such as video processing, natural language processing, and multimodal learning, where additional dimensions may correspond to time steps, layers, or modalities.

1.3.3 Mathematical Examples with Numerical Illustration

Example 1: Scalar, Vector, and Matrix as Low-Order Tensors **Scalar (Order-0 Tensor)** Consider a scalar value:

$$a = 5$$

This scalar can represent a bias term or a learning rate in a neural network. Since it has no indices, it is classified as an order-0 tensor.

Vector (Order-1 Tensor) Let the vector be defined as:

$$\mathbf{v} = \begin{bmatrix} 2 \\ -1 \\ 3 \end{bmatrix} \in \mathrm{R}^3$$

The individual elements are:

$$v_1 = 2, \quad v_2 = -1, \quad v_3 = 3$$

Each element is accessed using a single index, confirming that **v** is an order-1 tensor.

Matrix (Order-2 Tensor) Consider the matrix:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \in \mathrm{R}^{2 \times 3}$$

The element in the second row and third column is:

$$a_{23} = 6$$

Here, two indices (i, j) are required to access each element, indicating an order-2 tensor.

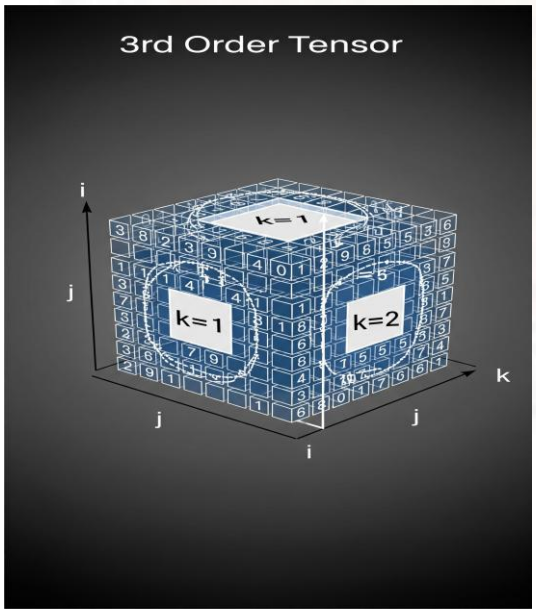


Figure 1.1: Tensor 3-D

Example 2: Third-Order Tensor with Explicit Indexing

Consider a third-order tensor:

$$T \in R^{2\times2\times2}$$

Let its numerical values be defined as:

$$T(:,:,1) = \begin{matrix} 1 & 2 \\ 3 & \end{matrix}, \quad T(:,:,2) = \begin{matrix} 5 & 6 \\ 7 & \end{matrix}$$

Step 1: Element Identification Each element of T is represented as t_{ijk}. For example:

$$t_{1,2,1} = 2, \quad t_{2,1,2} = 7$$

Step 2: Mode-1 Fiber Extraction Fix indices $j = 1$ and $k = 1$, and vary i:

$$\text{Mode-1 fiber} = \begin{matrix} t_{1,1,1} \\ t_{2,1,1} \end{matrix} = \begin{matrix} 1 \\ 3 \end{matrix}$$

Step 3: Mode-2 Fiber Extraction Fix indices $i = 2$ and $k = 2$, and vary j:

$$\text{Mode-2 fiber} = \begin{matrix} t_{2,1,2} \\ t_{2,2,2} \end{matrix} = \begin{matrix} 7 \\ 8 \end{matrix}$$

Step 4: Slice Extraction Fix index $k = 1$ to obtain a matrix slice:

$$T_{::1} = \begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix}$$

Example 3: Einstein Summation Notation (Matrix–Vector Product)

Let:

$$\mathbf{A} = \begin{matrix} 1 & 2 \\ 3 & \end{matrix}, \quad \mathbf{x} = \begin{matrix} 5 \\ 6 \end{matrix}$$

Step 1: Index Representation Using index notation:

$$y_i = a_{ij}x_j$$

Step 2: Explicit Summation For $i = 1$:

$$y_1 = a_{11}x_1 + a_{12}x_2 = (1)(5) + (2)(6) = 17$$

For $i = 2$:

$$y_2 = a_{21}x_1 + a_{22}x_2 = (3)(5) + (4)(6) = 39$$ Step 3: ResultingVector

$$\mathbf{y} = \begin{matrix} 17 \\ 39 \end{matrix}$$

Example 4: Fourth-Order Tensor Representing a Mini-Batch

Consider a fourth-order tensor representing a mini-batch of grayscale images:

$$X \in R^{2\times2\times2\times1}$$

where:

$$\text{Batch} = 2, \text{Height} = 2, \text{Width} = 2, \text{Channels} = 1$$

Let the tensor values be:

$$X(1, :, :, 1) = \begin{matrix} 10 & 20 \\ 30 & \end{matrix}, \quad X(2, :, :, 1) = \begin{matrix} 50 & 60 \\ 70 & \end{matrix}$$

Step 1: Accessing an Element The pixel at batch 2, row 1, column 2 is:

$$x_{2,1,2,1} = 60$$

Step 2: Batch Slice Extraction Fix the batch index to extract the first image:

$$X_{1,:,:,:} = \begin{matrix} 10 & 20 \\ 30 & 40 \end{matrix}$$

Step 3: Interpretation This structure allows simultaneous processing of multiple samples in deep learning models, enabling efficient vectorized computation during training.

Mathematical Examples

Example 1: Order-3 Tensor Let $A \in R^{2\times3\times2}$ with elements:

$$A_{:,:,1} = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{matrix}$$

$$A_{:,:,2} = \begin{matrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{matrix}$$

Example 2: Einstein Notation Matrix multiplication in Einstein notation:

$$C_{ij} = A_{ik}B_{kj} \quad \text{(implied summation over } k\text{)}$$

1.3.4 Tensor Operations

Element-wise Operations

For tensors $\mathrm{A}, \mathrm{B} \in \mathrm{R}^{I_1 \times I_2 \times \cdots \times I_N}$:

- **Addition:** $(\mathrm{A} + \mathrm{B})_{i_1 i_2 \ldots i_N} = a_{i_1 i_2 \ldots i_N} + b_{i_1 i_2 \ldots i_N}$
- **Hadamard Product:** $(\mathrm{A} \odot \mathrm{B})_{i_1 i_2 \ldots i_N} = a_{i_1 i_2 \ldots i_N} \cdot b_{i_1 i_2 \ldots i_N}$

Tensor Products

Mode-k Product The mode-k product of tensor $\mathrm{A} \in \mathrm{R}^{I_1 \times \cdots \times I_N}$ with matrix $\mathbf{U} \in \mathrm{R}^{J \times I_k}$:

$$(\mathrm{A} \times_k \mathbf{U})_{i_1 \ldots i_{k-1} j i_{k+1} \ldots i_N} = \sum_{i_k=1} a_{i_1 \ldots i_N} \cdot u_{j i_k}$$

Outer Product For vectors $\mathbf{u} \in \mathrm{R}^I$, $\mathbf{v} \in \mathrm{R}^J$, $\mathbf{w} \in \mathrm{R}^K$:

$$\mathrm{T} = \mathbf{u} \circ \mathbf{v} \circ \mathbf{w} \quad \text{where} \quad t_{ijk} = u_i v_j w_k$$

Kronecker Product For matrices $\mathbf{A} \in \mathrm{R}^{m \times n}$, $\mathbf{B} \in \mathrm{R}^{p \times q}$:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix}$$

Tensor Contraction

Generalization of matrix multiplication to higher-order tensors. Given tensors $\mathcal{A} \in \mathbb{R}^{I_1 \times \cdots \times I_M \times J_1 \times \cdots \times J_N}$ and $\mathcal{B} \in \mathbb{R}^{J_1 \times \cdots \times J_N \times K_1 \times \cdots \times K_P}$:

$$\mathcal{C}_{i_1 \ldots i_M k_1 \ldots k_P} = \sum_{j_1=}^{J_1} \cdots \sum_{j_N}^{J_N} a_{i_1 \ldots i_M j_1 \ldots j_N} \cdot b_{j_1 \ldots j_N k_1 \ldots k_P}$$

1.3.5 Tensor Decompositions

CP Decomposition (CANDECOMP/PARAFAC)

A tensor $\mathcal{T} \in \mathbb{R}^{I \times J \times K}$ can be approximated as:

$$\mathcal{T} \approx \sum_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r$$

where $\mathbf{a}_r \in \mathbb{R}^I$, $\mathbf{b}_r \in \mathbb{R}^J$, $\mathbf{c}_r \in \mathbb{R}^K$.

Tucker Decomposition

$$\mathcal{T} \approx \mathcal{G} \times_1 \mathbf{A} \times_2 \mathbf{B} \times_3 \mathbf{C}$$

where $\mathcal{G} \in \mathbb{R}^{R_1 \times R_2 \times R_3}$ is the core tensor.

1.3.6 Tensor Norms

General Tensor Norms

For a tensor $\mathcal{T} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$:

Frobenius Norm:

$$\|\mathcal{T}\|_F = \sqrt{\sum_{i_1=1}^{I_1} \sum_{i_2=1}^{I_2} \cdots \sum_{i_N=1}^{I_N} |t_{i_1 i_2 \ldots i_N}|^2}$$

Mode-k Norm:

$$\|\mathcal{T}\|_k = \max_{\|\mathbf{x}\|_2=1} \|\mathcal{T} \times_k \mathbf{x}^\top\|_F$$

Nuclear Norm (for tensors):

$$\|\mathcal{T}\|_* = \inf \left(\sum_{i=} \|\mathbf{u}_i^{(1}\|_2 \|\mathbf{u}_i^{(2}\|_2 \cdots \|\mathbf{u}_i^{(N}\|_2 : \mathcal{T} = \sum_{i=} \mathbf{u}_i^{(1} \circ \mathbf{u}_i^{(2} \circ \cdots \circ \mathbf{u}_i^{(N} \right)$$

Worked Examples of Tensor Norms

Example: Frobenius Norm of Order-3 Tensor Let $\mathcal{A} \in \mathbb{R}^{2 \times 2 \times 2}$ with:

$$\mathcal{A}_{:,:,1} = \begin{matrix} 1 & 2 \end{matrix}, \quad \mathcal{A}_{:,:,2} = \begin{matrix} 5 & 6 \end{matrix}$$

7 8 3

$$\|A\|_F = \sqrt{1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 6^2 + 7^2 + 8^2} = \sqrt{204} \approx 14.28$$

1.3.7 Tensor Calculus and Derivatives

Tensor Derivatives

For a scalar function $f : \mathbb{R}^{I_1 \times \cdots \times I_N} \to \mathbb{R}$, the gradient is a tensor:

$$\nabla_{\mathcal{X}} f(\mathcal{X}) = \frac{\partial f}{\partial x_{i_1 \ldots i_N}}$$

Chain Rule for Tensors For composition $f(g(\mathcal{X}))$:

$$\frac{\partial f}{\partial x_{i_1 \ldots i_N}} = \sum_{j_1 \ldots j_M} \frac{\partial f}{\partial g_{j_1 \ldots j_M}} \cdot \frac{\partial g_{j_1 \ldots j_M}}{\partial x_{i_1 \ldots i_N}}$$

1.3.8 Applications in Deep Learning

Neural Network Operations as Tensor Operations

Convolutional Layers For input $X \in \mathbb{R}^{B \times H \times W \times C_{\text{in}}}$ and filters $W \in \mathbb{R}^{K \times K \times C_{\text{in}} \times C_{\text{out}}}$:

$$Y_{b,i,j,c} = \sum_{m=1}^{K} \sum_{n=1}^{K} \sum_{d=1}^{C_{\text{in}}} X_{b,i+m,j+n,d} \cdot W_{m,n,d,c}$$

Fully Connected Layers Matrix multiplication extended to batches:

$$Y_{b,j} = \sum_{i=1} X_{b,i} \cdot W_{i,j} \quad \text{for } X \in \mathbb{R}^{B \times n}, W \in \mathbb{R}^{n \times m}$$

Batch Normalization Operations across batch dimension:

$$Y_{b,i,j,c} = \gamma_c \cdot \frac{X_{b,i,j,c} - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}} + \beta_c$$

Attention Mechanisms

Self-Attention with Tensor Notation For input $X \in \mathbb{R}^{B \times T \times d}$:

$$Q = X \times_3 W_Q, \quad K = X \times_3 W_K, \quad V = X \times_3 W_V$$

$$A_{b,i,j} = \text{softmax}\left(\frac{\sum_{k=1}^{d} Q_{b,i,k} K_{b,j,k}}{\sqrt{d}}\right)$$

$$Y_{b,i,k} = \sum_{j=1} A_{b,i,j} V_{b,j,k}$$

Tensor Decomposition for Model Compression

CP Decomposition of Convolutional Filters Decompose 4D convolutional weight tensor $W \in \mathbb{R}^{K \times K \times C_{\text{in}} \times C_{\text{out}}}$:

$$W_{k,l,i,j} \approx \sum_{r=}^{R} U^{(1)}_{k,r} U^{(2)}_{l,r} U^{(3)}_{i,r} U^{(4)}_{j,r}$$

Tucker Decomposition for FC Layers Decompose weight matrix $W \in \mathbb{R}^{m \times n}$ as Tucker-2 tensor:

$$W \approx U \cdot G \cdot V^{\top}$$

where $U \in \mathbb{R}^{m \times r_1}$, $G \in \mathbb{R}^{r_1 \times r_2}$, $V \in \mathbb{R}^{n \times r_2}$ with $r_1 \ll m$, $r_2 \ll n$.

Regularization with Tensor Norms

Low-Rank Regularization Using tensor nuclear norm for model compression:

$$L_{\text{reg}}(\mathrm{W}) = L(\mathrm{W}) + \lambda \| \mathrm{W} \|_*$$

Group Sparsity with Tensor Norms For convolutional filters, promote channel-wise sparsity:

$$L_{\text{reg}}(\mathrm{W}) = L(\mathrm{W}) + \lambda \sum_{c=1}^{C_{\text{out}}} \|\mathrm{W}_{:,:,:,c}\|_F$$

1.3.9 Advanced Tensor Applications

Higher-Order SVD (HOSVD)

Generalization of SVD to tensors:

$$\mathrm{T} = \mathrm{S} \times_1 U^{(1)} \times_2 U^{(2)} \times_3 \cdots \times_N U^{(N}$$

Tensor Train Decomposition

Efficient representation of high-dimensional tensors:

$$^{(1)} \quad ^{2} \qquad ^{(N)}$$

$$\mathrm{T}_{i_1 i_2 \ldots i_N} = G_{i_1}\, G_{i_2} \cdots G_{i_N}$$

where $G_i^{(k)} \in \mathrm{R}^{r_{k-1} \times r_k}$ with $r_0 = r_N = 1$.

Graph Neural Networks as Tensor Operations

Node features $\mathrm{X} \in \mathrm{R}^{N \times d}$, adjacency matrix $A \in \mathrm{R}^{N \times N}$:

$$\mathrm{X}^{(l+1)} = \sigma\ A \cdot \mathrm{X}^{(l)} \cdot W^{(l)}$$

1.3.10 Implementation Considerations

Efficient Tensor Operations

- **Memory Layout:** Row-major vs column-major storage
- **Batched Operations:** Leveraging BLAS-3 operations for efficiency
- **GPU Optimization:** Tensor cores for mixed-precision operations

Automatic Differentiation with Tensors

Modern frameworks (PyTorch, TensorFlow) handle tensor gradients automatically:

$$\mathrm{Y} = f(\mathrm{X}, \mathrm{W})$$
$$\mathrm{G} = \nabla_{\mathrm{W}} L(\mathrm{Y}, \mathrm{Y}_{\text{true}})$$

1.3.11 Worked Examples and Code-like Representation

Tensor Contraction Example

Given $\mathrm{A} \in \mathrm{R}^{3\times4\times5}$, $\mathrm{B} \in \mathrm{R}^{5\times4\times2}$, contract over dimensions 2-3 and 1-2:

$$\mathrm{C}_{i,k} = \sum_{j=1}^{4} \sum_{l=}^{5} A_{i,j,l} \cdot B_{l,j,k}$$

Backward Pass Through Convolutional Layer

For loss L, input X, weights W, output Y:

$$\frac{\partial L}{\partial \mathrm{W}_{m,n,d,c}} = \sum_{b,i,j} \frac{\partial L}{\partial \mathrm{Y}_{b,i,j,c}} \cdot \mathrm{X}_{b,i+m,j+n,d}$$

$$\frac{\partial L}{\partial \mathrm{X}_{b,i,j,d}} = \sum_{m,n,c} \frac{\partial L}{\partial \mathrm{Y}_{b,i-m,j-n,c}} \cdot \mathrm{W}_{m,n,d,c}$$

1.3.12 Emerging Research Areas

Tensor Networks for Quantum Machine Learning

Using matrix product states (MPS) and tensor networks for quantum-inspired models.

Neural Architecture Search with Tensor Decomposition

Automated architecture discovery using tensor factorization techniques.

Dynamic Tensor Decomposition

Online tensor decomposition for streaming data and continual learning.

1.4 Matrix Operations

1.4.1 Matrix Addition

Matrices of same dimensions can be added element-wise:

$$\mathbf{C} = \mathbf{A} + \mathbf{B} \quad \text{where } c_{ij} = a_{ij} + b_{ij}$$

1.4.2 Matrix Multiplication

The product $\mathbf{C} = \mathbf{AB}$ is defined when $\mathbf{A} \in \mathrm{R}^{m \times n}$, $\mathbf{B} \in \mathrm{R}^{n \times p}$:

$$c_{ij} = \sum_{k=1} a_{ik} b_{kj}$$

Worked Example:

$$\begin{matrix}1 & 2\\ 3 & 4\end{matrix}\begin{matrix}5 & 6\\ 7 & 8\end{matrix} = \begin{matrix}1\times5+2\times7 & 1\times6+2\times8\\ 3\times5+4\times7 & 3\times6+4\times8\end{matrix} = \begin{matrix}19 & 22\\ 43 & 50\end{matrix}$$

1.4.3 Hadamard Product

Element-wise multiplication: $\mathbf{C} = \mathbf{A} \odot \mathbf{B}$ where $c_{ij} = a_{ij}b_{ij}$

1.5 Types of Matrices

1.5.1 Identity Matrix

Square matrix with 1s on diagonal and 0s elsewhere:

$$\mathbf{I}_3 = \begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$$

Property: $\mathbf{AI} = \mathbf{IA} = \mathbf{A}$

1.5.2 Diagonal Matrix

Non-zero elements only on main diagonal:

$$\mathbf{D} = \begin{bmatrix}d_1 & 0 & 0\\ 0 & d_2 & 0\\ 0 & 0 & d_3\end{bmatrix}$$

1.5.3 Symmetric Matrix

Matrix equal to its transpose: $\mathbf{A} = \mathbf{A}^\top$

$$\mathbf{A} = \begin{bmatrix}2 & -1 & 3\\ -1 & 5 & 0\\ 3 & 0 & 4\end{bmatrix}$$

1.6 Norms: Mathematical Formulation and Applica-tions

1.6.1 Definition and General Properties

A norm is a function that assigns a strictly positive length or size to each vector in a vector space, except for the zero vector which is assigned a length of zero. Formally, for a vector space V over a field F (typically R or C), a norm $\|\cdot\|$ satisfies:

1. **Positive definiteness:** $\|\mathbf{x}\| \geq 0$ and $\|\mathbf{x}\| = 0 \iff \mathbf{x} = \mathbf{0}$
2. **Absolute homogeneity:** $\|\alpha\mathbf{x}\| = |\alpha|\|\mathbf{x}\|$ for all $\alpha \in \mathrm{F}$
3. **Triangle inequality:** $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$

1.6.2 General Norm Formulations

Discrete Domain (Vector Norms)

For vectors $\mathbf{x} \in \mathrm{R}^n$ or C^n, the general L^p norm is defined as:

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p} \quad \text{for } 1 \leq p < \infty$$

Continuous Domain (Function Norms)

For functions $f : \Omega \to \mathrm{R}$ defined on a domain $\Omega \subseteq \mathrm{R}^n$, the L^p norm is defined as:

$$\|f\|_p = \left(\int_{\Omega} |f(x)|^p dx \right)^{1/p} \quad \text{for } 1 \leq p < \infty$$

1.6.3 Important Norm Families and Their Properties

L^p Norms

- L^0 **"Norm" (Pseudo-norm):** Counts non-zero elements

$$\|\mathbf{x}\|_0 = \#\{i : x_i \neq 0\}$$

- L^1 **Norm (Manhattan/Taxicab):**

$$\|\mathbf{x}\|_1 = \sum_{i=1} |x_i|$$

- L^2 **Norm (Euclidean):**

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2}$$

- L^p **Norm (General):**

$$\|\mathbf{x}\|_p = \left(\sum \right)^{1/p}$$

- **L^∞ Norm (Maximum/Chebyshev):**

$i=1$

Special Cases and Relationships

Hölder's Inequality: For vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ and $p, q > 1$ with $\frac{1}{p} + \frac{1}{q} = 1$:

$$|\mathbf{x}^\top \mathbf{y}| \leq \|\mathbf{x}\|_p \|\mathbf{y}\|_q$$

Minkowski's Inequality (Triangle inequality for L^p norms):

$$\|\mathbf{x} + \mathbf{y}\|_p \leq \|\mathbf{x}\|_p + \|\mathbf{y}\|_p$$

Relationships between norms: For $\mathbf{x} \in \mathbb{R}^n$ and $1 \leq p < q \leq \infty$:

$$\|\mathbf{x}\|_q \leq \|\mathbf{x}\|_p \leq n^{\frac{1}{p} - \frac{1}{q}} \|\mathbf{x}\|_q$$

1.6.4 Matrix Norms

Induced Matrix Norms

For a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, the p-norm is induced by vector norms:

$$\|\mathbf{A}\|_p = \sup_{\mathbf{x} \neq \mathbf{0}} \frac{\|\mathbf{A}\mathbf{x}\|_p}{\|\mathbf{x}\|_p} = \sup_{\|\mathbf{x}\|_p = 1} \|\mathbf{A}\mathbf{x}\|_p$$

Common Matrix Norms

- **Spectral Norm (L^2 matrix norm):**

$$\|\mathbf{A}\|_2 = \sigma_{\max}(\mathbf{A}) = \sqrt{\lambda_{\max}(\mathbf{A}^\top \mathbf{A})}$$

- **Frobenius Norm:**

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{} |a_{ij}|^2} = \sqrt{\operatorname{tr}(\mathbf{A}^\top \mathbf{A})}$$

- **Nuclear Norm:**

$$\|\mathbf{A}\|_* = \sum_{i=1}^{\min(m,n)} \sigma_i(\mathbf{A})$$

- **Max Norm:**

$$\|\mathbf{A}\|_{\max} = \max_{i,j} |a_{ij}|$$

1.6.5 Worked Examples

Vector Norm Calculations

For $\mathbf{x} = [3, -4, 0, 2]^T$:

$|x_i|$

$$\|\mathbf{x}\|_1 = |3| + |-4| + |0| + |2| = 9$$

$$\|\mathbf{x}\|_2 = \sqrt{3^2 + (-4)^2 + 0^2 + 2^2} = \sqrt{9 + 16 + 0 + 4} = \sqrt{29} \approx 5.385$$

$$\|\mathbf{x}\|_3 = (27 + 64 + 0 + 8)^{1/3} = 99^{1/3} \approx 4.626$$

$$\|\mathbf{x}\|_\infty = \max(|3|, |-4|, |0|, |2|) = 4$$

Function Norm Calculation

For $f(x) = x^2$ on $[0, 1]$:

$$\|f\|_2 = \int_0^1 |x^2|\, dx^{1/2} = \int_0^1 x^4 dx^{1/2} = \frac{1}{5}^{1/2} = \frac{1}{\sqrt{5}}$$

1.6.6 Applications in Deep Learning and Machine Learning

Regularization

- L^1 **Regularization (Lasso):** Promotes sparsity in weights, useful for feature selection

$$L_{\text{reg}}(\mathbf{w}) = L(\mathbf{w}) + \lambda \|\mathbf{w}\|_1$$

- L^2 **Regularization (Ridge/Tikhonov):** Prevents overfitting by penalizing large weights

$$L_{\text{reg}}(\mathbf{w}) = L(\mathbf{w}) + \lambda \|\mathbf{w}\|_2^2$$

- **Elastic Net:** Combines L^1 and L^2 regularization

$$L_{\text{reg}}(\mathbf{w}) = L(\mathbf{w}) + \lambda_1 \|\mathbf{w}\|_1 + \lambda_2 \|\mathbf{w}\|_2^2$$

Loss Functions

- **Mean Absolute Error (MAE):** Uses L^1 norm, robust to outliers

$$L(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n} \|\mathbf{y} - \hat{\mathbf{y}}\|_1$$

- **Mean Squared Error (MSE):** Uses squared L^2 norm, differentiable

$$L(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n} \|\mathbf{y} - \hat{\mathbf{y}}\|_2^2$$

Optimization and Convergence

- **Gradient Clipping:** Prevents exploding gradients by limiting L^2 norm

$$\mathbf{g} \leftarrow \mathbf{g} \cdot \min\left(1, \frac{\tau}{\|\mathbf{g}\|}\right)$$

- **Convergence Analysis:** Norms measure distance to optimal solution

$$\|\mathbf{w}_{t+1} - \mathbf{w}^*\| \leq \rho \|\mathbf{w}_t - \mathbf{w}^*\|$$

Model Compression and Efficiency

- **Pruning:** L^0 "norm" minimization for reducing model size
- **Low-rank Approximation:** Nuclear norm minimization for matrix compression

- **Quantization:** Minimizing representation error using appropriate norms

Computer Vision Applications

- **Image Similarity:** L^1 and L^2 norms for content-based image retrieval
- **Style Transfer:** Norm-based loss functions for feature matching
- **Super-resolution:** Norm minimization in reconstruction error

Natural Language Processing

- **Word Embeddings:** Cosine similarity (normalized L^2 dot product)
- **Document Similarity:** TF-IDF vectors with various distance metrics
- **Attention Mechanisms:** Normalized similarity scores

1.6.7 Norm Equivalence and Practical Considerations

Norm Equivalence in Finite Dimensions

In R^n, all norms are equivalent:

$$\alpha\|\mathbf{x}\|_p \leq \|\mathbf{x}\|_q \leq \beta\|\mathbf{x}\|_p$$

for some constants $\alpha, \beta > 0$.

Choice of Norm in Practice

- L^1**:** When sparsity is desired, robust to outliers
- L^2**:** When differentiability is important, Gaussian noise assumption
- L^∞**:** When worst-case error matters most
- **Matrix norms:** For regularization in deep networks, low-rank approximations

1.6.8 Numerical Computation Considerations

Numerical Stability

- L^2 norm computation should use stable algorithms to avoid overflow/underflow
- For large vectors, compute norms in log-space when possible
- Use specialized libraries for norm computation in high dimensions

Computational Complexity

- L^1 and L^2 norms: $O(n)$ for vectors, $O(mn)$ for matrices
- L^∞ norm: $O(n)$ with single pass
- Spectral norm: $O(\min(m, n)^3)$ for exact computation

This comprehensive presentation of norms provides the mathematical foundation for un-derstanding their role in deep learning algorithms, from regularization and optimization to model interpretation and performance analysis.

1.7 Eigen Decomposition: Theory, Computation, and Applications

1.7.1 Mathematical Foundation

Definition and Basic Concepts

For a square matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, a scalar λ is called an eigenvalue and a non-zero vector $\mathbf{v}$ is called an eigenvector if they satisfy:

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$$

This equation means that the linear transformation represented by $\mathbf{A}$ scales the eigen-vector $\mathbf{v}$ by the factor λ without changing its direction.

Characteristic Equation

The eigenvalues are found by solving the characteristic equation:

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0$$

where $\mathbf{I}$ is the identity matrix. The left-hand side is a polynomial in λ called the characteristic polynomial.

1.7.2 Matrix Operations with Detailed Examples

Definition 1.7.1 (Matrix Addition)**.** *For* $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$*, their sum* $\mathbf{C} = \mathbf{A} + \mathbf{B}$ *is defined component-wise:*

$$c_{ij} = a_{ij} + b_{ij} \text{ for } i = 1, \ldots, m, \; j = 1, \ldots, n$$

Example 1.7.1 (Matrix Addition)**.** *Let* $\mathbf{A} = \begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix}$ *and* $\mathbf{B} = \begin{matrix} 5 & 6 \\ 7 & 8 \end{matrix}$ *. Then:*

$$\mathbf{A} + \mathbf{B} = \begin{matrix} 1+5 & 2+6 \\ 3+7 & 4 \end{matrix} = \begin{matrix} 6 & 8 \\ 10 & \end{matrix}$$

Definition 1.7.2 (Matrix Multiplication). *For* $\mathbf{A} \in \mathbb{R}^{m \times n}$ *and* $\mathbf{B} \in \mathbb{R}^{n \times p}$*, their product* $\mathbf{C} = \mathbf{AB} \in \mathbb{R}^{m \times p}$ *is:*

$$c_{ij} = \sum_{k=1} a_{ik} b_{kj} \quad \text{for } i = 1, \ldots, m,\ j = 1, \ldots, p$$

Note: The number of columns in $\mathbf{A}$ *must equal the number of rows in* $\mathbf{B}$*.*

Example 1.7.2 (Matrix Multiplication). *Let* $\mathbf{A} = \begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix}$ *and* $\mathbf{B} = \begin{matrix} 5 & 6 \\ 7 & 8 \end{matrix}$*. Then:*

$$\mathbf{AB} = \begin{matrix} 1 \cdot 5 + 2 \cdot 7 & 1 \cdot 6 + 2 \cdot 8 \\ 3 \cdot 5 + 4 \cdot 7 & 3 \cdot 6 + \\ & 4 \cdot 8 \end{matrix} = \begin{matrix} 5 + 14 & 6 + 16 \\ 15 + 28 & 18 + \end{matrix} = \begin{matrix} 19 & 22 \\ 43 & \end{matrix}$$

Definition 1.7.3 (Transpose). *The* ***transpose*** *of* $\mathbf{A} \in \mathbb{R}^{m \times n}$ *is* $\mathbf{A}^T \in \mathbb{R}^{n \times m}$*:*

$$(\mathbf{A}^T)_{ij} = a_{ji}$$

Definition 1.7.4 (Trace). *The* ***trace*** *of a square matrix* $\mathbf{A} \in \mathbb{R}^{n \times n}$ *is the sum of its diagonal elements:*

$$\operatorname{tr}(\mathbf{A}) = \sum_{i=1} a_{ii}$$

Definition 1.7.5 (Determinant). *The* ***determinant*** *of a square matrix* $\mathbf{A} \in \mathbb{R}^{n \times n}$*, denoted* $\det(\mathbf{A})$ *or* $|\mathbf{A}|$*, is a scalar value that can be computed recursively.*

For 2×2 *matrix:*

$$\det \begin{matrix} a & b \\ c & d \end{matrix} = ad - bc$$

For $n \times n$ *matrix:*

$$\det(\mathbf{A}) = \sum_{j=1} (-1)^{i+j} a_{ij} M_{ij} \quad \text{(Laplace expansion)}$$

where M_{ij} *is the* (i, j)*-minor of* $\mathbf{A}$*.*

1.7.3 Types of Matrices with Detailed Properties

Definition 1.7.6 (Symmetric Matrix). *A matrix* $\mathbf{A}$ *is* ***symmetric*** *if* $\mathbf{A} = \mathbf{A}^T$*. This implies* $a_{ij} = a_{ji}$ *for all* i, j*.*

Definition 1.7.7 (Orthogonal Matrix). *A matrix* $\mathbf{Q} \in \mathbb{R}^{n \times n}$ *is* ***orthogonal*** *if:*

$$\mathbf{Q}^T \mathbf{Q} = \mathbf{Q}\mathbf{Q}^T = \mathbf{I}_n$$

Equivalently, $\mathbf{Q}^T = \mathbf{Q}^{-1}$*. The columns of* $\mathbf{Q}$ *form an orthonormal basis.*

Definition 1.7.8 (Diagonal Matrix)**.** *A matrix* **D** *is* ***diagonal*** *if* $d_{ij} = 0$ *for all* $i \neq j$*. Denoted* $\mathbf{D} = \text{diag}(d_1, d_2, \ldots, d_n)$*.*

Definition 1.7.9 (Identity Matrix)**.** *The* ***identity matrix*** $\mathbf{I}_n$ *is a diagonal matrix with ones on the diagonal:*

$$\mathbf{I}_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

Definition 1.7.10 (Positive Definite Matrix)**.** *A symmetric matrix* $\mathbf{A} \in \mathbb{R}^{n \times n}$ *is* ***positive definite*** *if:*

$$\mathbf{x}^T \mathbf{A} \mathbf{x} > 0 \quad \text{for all } \mathbf{x} \in \mathbb{R}^n \setminus \{\mathbf{0}\}$$

Properties:

1. *All eigenvalues are positive*
2. *All principal minors are positive*
3. *All diagonal elements are positive*
4. $\mathbf{A}$ *is invertible with* $\mathbf{A}^{-1}$ *also positive definite*

Definition 1.7.11 (Singular Matrix)**.** *A square matrix* $\mathbf{A}$ *is* ***singular*** *if* $\det(\mathbf{A}) = 0$*, mean-ing it is not invertible.*

1.7.4 Norms: Detailed Mathematical Treatment

Definition 1.7.12 (Vector Norm)**.** *A* ***norm*** $\|\cdot\|$ *on* $\mathbb{R}^n$ *is a function satisfying:*

1. $\|\mathbf{x}\| \geq 0$ *for all* $\mathbf{x}$*, and* $\|\mathbf{x}\| = 0$ *iff* $\mathbf{x} = \mathbf{0}$
2. $\|\alpha\mathbf{x}\| = |\alpha| \|\mathbf{x}\|$ *for all* $\alpha \in \mathbb{R}$
3. $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$ *(triangle inequality)*

Definition 1.7.13 (Lp Norms)**.** *For* $\mathbf{x} \in \mathbb{R}^n$ *and* $p \geq 1$*, the* ℓ_p *norm is:*

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}$$

Special cases:

- ℓ_1 *norm:* $\|\mathbf{x}\|_1 = \sum_{i=}^{n} |x_i|$
- ℓ_2 *norm (Euclidean):* $\|\mathbf{x}\|_2 = \sqrt{\sum_{i=}^{n} x_i^2}$
- ℓ_∞ *norm:* $\|\mathbf{x}\|_\infty = \max_i |x_i|$

Example 1.7.3 (Vector Norms). *For* $\mathbf{x} = [3, 4]^T$*:*

- $\|\mathbf{x}\|_1 = |3| + |4| = 7$
- $\|\mathbf{x}\|_2 = \sqrt{3^2 + 4^2} = \sqrt{9 + 16} = \sqrt{25} = 5$
- $\|\mathbf{x}\|_\infty = \max(|3|, |4|) = 4$

Definition 1.7.14 (Matrix Norm). *A* ***matrix norm*** *extends vector norms to matrices. The Frobenius norm is:*

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{} a_{ij}^2} = \sqrt{\text{tr}(\mathbf{A}^T \mathbf{A})}$$

The induced p-norm is:

$$\|\mathbf{A}\|_p = \sup_{\mathbf{x} \neq \mathbf{0}} \frac{\|\mathbf{A}\mathbf{x}\|_p}{\|\mathbf{x}\|_p}$$

1.7.5 Eigen Decomposition: Complete Derivation

Definition 1.7.15 (Eigenvalues and Eigenvectors). *For a square matrix* $\mathbf{A} \in \mathbb{R}^{n \times n}$*, a scalar* λ *is an* ***eigenvalue*** *and* $\mathbf{v} \neq \mathbf{0}$ *is an* ***eigenvector*** *if:*

$$\mathbf{A}\mathbf{v} = \lambda \mathbf{v}$$

Derivation of Characteristic Equation: Starting from $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$:

$$\mathbf{A}\mathbf{v} - \lambda\mathbf{v} =$$

$$\mathbf{0} \ (\mathbf{A} - \lambda\mathbf{I})\mathbf{v}$$

$$= \mathbf{0}$$

For nontrivial solution $\mathbf{v} \neq \mathbf{0}$, the matrix $(\mathbf{A} - \lambda\mathbf{I})$ must be singular:

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0$$

This is the **characteristic equation**, a polynomial of degree n in λ.

Example 1.7.4 (Eigen Decomposition). *Find eigenvalues and eigenvectors of* $\mathbf{A} = \begin{matrix} 4 & 1 \\ 2 & 3 \end{matrix}$.

Solution:

1. *Characteristic equation:*

$$\det(\mathbf{A}\_\lambda\mathbf{I}) = \det \begin{matrix} 4 - \lambda & 1 \\ 2 & 3 - \lambda \end{matrix} = (4 - \lambda)(3 - \lambda) - 2 = 0$$

2. *Eigenvalues:* $\lambda_1 = 5$, $\lambda_2 = 2$

$$\lambda^2 - 7\lambda + 10 = 0$$

3. *For* $\lambda_1 = 5$*:*

$$(\mathbf{A} - 5\mathbf{I})\mathbf{v}_1 = \begin{pmatrix} -1 & 1 \\ 2 & -2 \end{pmatrix} \mathbf{v}_1 = \mathbf{0}$$

Solution: $\mathbf{v}_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ *(or any scalar multiple)*

4. *For* $\lambda_2 = 2$*:*

$$(\mathbf{A} - 2\mathbf{I})\mathbf{v}_2 = \begin{pmatrix} 2 & 1 \\ 2 & 1 \end{pmatrix} \mathbf{v}_2 = \mathbf{0}$$

Solution: $\mathbf{v}_2 = \begin{pmatrix} 1 \\ -2 \end{pmatrix}$

5. *Eigen decomposition:* $\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}$ *where:*

$$\mathbf{V} = \begin{pmatrix} 1 & 1 \\ 1 & -2 \end{pmatrix}, \quad \mathbf{\Lambda} = \begin{pmatrix} 5 & 0 \\ 0 & 2 \end{pmatrix}$$

Theorem 1.7.1 (Properties of Eigenvalues)**.** *For* $\mathbf{A} \in \mathbb{R}^{n \times n}$*:*

1. $\text{tr}(\mathbf{A}) = \sum_{i=1}^{n} \lambda_i$
2. $\det(\mathbf{A}) = \prod_{i=1}^{n} \lambda_i$
3. *The eigenvalues of* $\mathbf{A}^T$ *are the same as those of* $\mathbf{A}$
4. *If* $\mathbf{A}$ *is symmetric, all eigenvalues are real*
5. *If* $\mathbf{A}$ *is positive definite, all eigenvalues are positive*

1.7.6 Singular Value Decomposition (SVD): Complete Mathemat-ical Treatment

Definition 1.7.16 (Singular Value Decomposition)**.** *For any matrix* $\mathbf{A} \in \mathbb{R}^{m \times n}$*, there exist orthogonal matrices* $\mathbf{U} \in \mathbb{R}^{m \times m}$ *and* $\mathbf{V} \in \mathbb{R}^{n \times n}$*, and a diagonal matrix* $\mathbf{\Sigma} \in \mathbb{R}^{m \times n}$ *with non-negative entries such that:*

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

The diagonal entries $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$ *are called* ***singular values****, where* $r = \text{rank}(\mathbf{A})$*.*

Derivation from Eigendecomposition: The SVD can be derived from the eigende-compositions of $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}\mathbf{A}^T$:

1. Compute $\mathbf{A}^T\mathbf{A} \in \mathbb{R}^{n \times n}$, which is symmetric and positive semidefinite. 2. Find its eigendecomposition: $\mathbf{A}^T\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^T$. 3. The singular values are $\sigma_i = \sqrt{\lambda_i}$, where λ_i are eigenvalues of $\mathbf{A}^T\mathbf{A}$. 4. The right singular vectors are the eigenvectors $\mathbf{v}_i$ of $\mathbf{A}^T\mathbf{A}$. 5. The left singular vectors are: $\mathbf{u}_i = \frac{1}{\sigma_i}\mathbf{A}\mathbf{v}_i$ for $i = 1, \ldots, r$.

Theorem 1.7.2 (Relationship between SVD and Eigendecomposition). *Given* $\mathbf{A} = \mathbf{U\Sigma V}^T$, *then:*

$$\mathbf{A}^T\mathbf{A} = \mathbf{V\Sigma}^T\mathbf{\Sigma V}^T = \mathbf{V\Lambda V}^T \quad \mathbf{AA}^T = \mathbf{U\Sigma\Sigma}^T\mathbf{U}^T = \mathbf{U\Lambda}'\mathbf{U}^T$$

where $\mathbf{\Lambda} = \text{diag}(\sigma_1^2, \sigma_2^2, \ldots, \sigma_r^2, 0, \ldots, 0)$ *and* $\mathbf{\Lambda}'$ *is* $m \times m$ *diagonal with* σ_i^2 *on diagonal.*

Proof: From $\mathbf{A} = \mathbf{U\Sigma V}^T$, we have:

$$\begin{aligned}\mathbf{A}^T\mathbf{A} &= (\mathbf{U\Sigma V}^T)^T(\mathbf{U\Sigma V}^T) \\ &= \mathbf{V\Sigma}^T\mathbf{U}^T\mathbf{U\Sigma V}^T \\ &= \mathbf{V\Sigma}^T\mathbf{\Sigma V}^T \quad (\text{since } \mathbf{U}^T\mathbf{U} = \mathbf{I})\end{aligned}$$

Now $\mathbf{\Sigma}^T\mathbf{\Sigma}$ is an $n \times n$ diagonal matrix with σ^2 on the diagonal, so:

$$\mathbf{A}^T\mathbf{A} = \mathbf{V\Lambda V}^T$$

where $\mathbf{\Lambda} = \text{diag}(\sigma_1^2, \sigma_2^2, \ldots, \sigma_r^2, 0, \ldots, 0)$.

Similarly:

$$\mathbf{AA}^T = \mathbf{U\Sigma\Sigma}^T\mathbf{U}^T = \mathbf{U\Lambda}'\mathbf{U}^T$$

where $\mathbf{\Lambda}'$ is $m \times m$ diagonal with σ^2 on diagonal.

Example 1.7.5 (SVD Computation). *Find the SVD of* $\mathbf{A} = \begin{bmatrix} 3 & 0 \\ 0 & -2 \end{bmatrix}$.

Solution:

1. *Compute* $\mathbf{A}^T\mathbf{A} = \begin{bmatrix} 3 & 0 \\ 0 & -2 \end{bmatrix}\begin{bmatrix} 3 & 0 \\ 0 & -2 \end{bmatrix} = \begin{bmatrix} 9 & 0 \\ 0 & 4 \end{bmatrix}$
2. *Eigenvalues of* $\mathbf{A}^T\mathbf{A}$*:* $\lambda_1 = 9$, $\lambda_2 = 4$
3. *Singular values:* $\sigma_1 = \sqrt{9} = 3, \sigma_2 = \sqrt{4} = 2$
4. *Eigenvectors of* $\mathbf{A}^T\mathbf{A}$*: For* $\lambda_1 = 9$*:* $\mathbf{v}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, *for* $\lambda_2 = 4$*:* $\mathbf{v}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$
5. $\mathbf{V} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
6. $\mathbf{u}_1 = \frac{1}{\sigma_1}\mathbf{Av}_1 = \frac{1}{3}\begin{bmatrix} 3 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$
7. $\mathbf{u}_2 = \frac{1}{\sigma_2}\mathbf{Av}_2 = \frac{1}{2}\begin{bmatrix} 0 \\ -2 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$

^{2} σ_2 ^{2} ^{2} -2 -1

8. $\mathbf{U} = \begin{matrix} 1 & 0 \\ 0 & -1 \end{matrix}$

9. $\mathbf{\Sigma} = \begin{matrix} 3 & 0 \\ 0 & 2 \end{matrix}$

10. *SVD:* $\mathbf{A} = {}^{T}\begin{matrix} 1 & 0 & 3 & 0 & 1 & 0 \\ 0 & -1 & 0 & 2 & 0 & 1 \end{matrix}$

Theorem 1.7.3 (Low-Rank Approximation - Eckart-Young-Mirsky)**.** *For* $\mathbf{A} \in \mathrm{R}^{m\times n}$ *with SVD* $\mathbf{A} = \mathbf{U\Sigma V}^T$*, the best rank-k approximation* $\mathbf{A}_k$ *that minimizes* $\|\mathbf{A} - \mathbf{B}\|_F$ *over all rank-k matrices* $\mathbf{B}$ *is:*

$$\mathbf{A}_k = \sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^T = \mathbf{U}_{:,1:k}\mathbf{\Sigma}_{1:k,1:k}\mathbf{V}^T_{:,1:k}$$

with approximation error:

$$\|\mathbf{A} - \mathbf{A}_k\|_F^2 = \sum_{i=k+1}^{r} \sigma_i^2$$

1.7.7 Principal Component Analysis (PCA): Complete Derivation

PCA aims to find a lower-dimensional representation that preserves maximum variance.

Mathematical Formulation: Given data matrix $\mathbf{X} \in \mathrm{R}^{m\times n}$ (m samples, n features), centered (zero mean).

1. Compute covariance matrix: $\mathbf{C} = \frac{1}{m-1}\mathbf{X}^T\mathbf{X}$
2. Eigendecomposition: $\mathbf{C} = \mathbf{V\Lambda V}^T$
3. Sort eigenvalues in descending order: $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$
4. Select top k eigenvectors (principal components): $\mathbf{V}_k = [\mathbf{v}_1, \ldots, \mathbf{v}_k]$
5. Project data: $\mathbf{Y} = \mathbf{XV}_k \in \mathrm{R}^{m\times k}$

Variance Maximization Derivation: The first principal component $\mathbf{w}_1$ maximizes variance:

$$\max_{\mathbf{w}} \mathrm{Var}(\mathbf{Xw}) = \max_{\mathbf{w}} \mathbf{w}^T\mathbf{Cw} \text{ subject to } \|\mathbf{w}\| = 1$$

Using Lagrange multiplier λ:

$$L(\mathbf{w}, \lambda) = \mathbf{w}^T\mathbf{Cw} - \lambda(\mathbf{w}^T\mathbf{w} - 1)$$

Taking gradient:

$$\nabla_{\mathbf{w}} L = 2\mathbf{Cw} - 2\lambda\mathbf{w} = 0$$

$$\mathbf{Cw} = \lambda\mathbf{w}$$

Thus $\mathbf{w}$ is an eigenvector of $\mathbf{C}$ with eigenvalue λ, and the variance is $\mathbf{w}^T\mathbf{Cw} = \lambda$.

Relationship with SVD: Let $\mathbf{X} = \mathbf{U\Sigma V}^T$ be the SVD. Then:

$$\mathbf{C} = \frac{1}{m-1}\mathbf{X}^T\mathbf{X} = \frac{1}{m-1}(\mathbf{U\Sigma V}^T)^T(\mathbf{U\Sigma V}^T)$$
$$= \frac{1}{m-1}\mathbf{V\Sigma}^T\mathbf{U}^T\mathbf{U\Sigma V}^T$$
$$= \frac{1}{m-1}\mathbf{V\Sigma}^T\mathbf{\Sigma V}^T$$

So $\mathbf{V}$ contains eigenvectors of $\mathbf{C}$ and eigenvalues are $\lambda_i = \frac{\sigma_i^2}{m-1}$.

1.7.8 Computational Methods and Worked Examples

Step-by-Step Computation

Example 1: 2×2 Matrix Let $\mathbf{A} = \begin{matrix} 4 & 1 \\ 2 & 3 \end{matrix}$

Step 1: Find eigenvalues

$$\det(\mathbf{A} - \lambda\mathbf{I}) = \det \begin{matrix} 4-\lambda & 1 \\ 2 & 3-\lambda \end{matrix}$$
$$= (4-\lambda)(3-\lambda) - 2 = \lambda^2 - 7\lambda + 10 = 0$$
$$\lambda_1 = 5, \quad \lambda_2 = 2$$

Step 2: Find eigenvectors

- For $\lambda_1 = 5$: $(\mathbf{A} - 5\mathbf{I})\mathbf{v} = \begin{matrix} -1 & 1 \\ 2 & -2 \end{matrix}\mathbf{v} = 0$

  $\Rightarrow \mathbf{v}_1 = \begin{matrix} 1 \\ 1 \end{matrix}$

- For $\lambda_2 = 2$: $(\mathbf{A} - 2\mathbf{I})\mathbf{v} = \begin{matrix} 2 & 1 \\ 2 & \end{matrix}\mathbf{v} = 0$

  $\Rightarrow \mathbf{v}_2 = \begin{matrix} 1 \\ -2 \end{matrix}$

Step 3: Construct decomposition

$$\mathbf{A} = \begin{matrix} 1 & 1 \\ 1 & -2 \end{matrix}\begin{matrix} 5 & 0 \\ 0 & 2 \end{matrix}\begin{matrix} 1 & 1 \\ 1 & -2 \end{matrix}^{-1}$$

Verification:

$$\mathbf{Av}_1 = \begin{bmatrix} 4 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 5 \\ 5 \end{bmatrix} = 5\mathbf{v}_1$$

$$\mathbf{Av}_2 = \begin{bmatrix} 4 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ -2 \end{bmatrix} = \begin{bmatrix} 2 \\ -4 \end{bmatrix} = 2\mathbf{v}_2$$

Example 2: Symmetric Matrix Let $\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$ (Symmetric matrices have orthogonal eigenvectors)

Eigenvalues: $\lambda_1 = 3, \lambda_2 = 1$

Eigenvectors: $\mathbf{v}_1 = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 1 \end{bmatrix}, \mathbf{v}_2 = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ -1 \end{bmatrix}$

$$\mathbf{A} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}^{-1}$$

Note: For orthogonal matrices, $\mathbf{V}^{-1} = \mathbf{V}^\top$.

Special Cases and Properties Symmetric Matrices:

- All eigenvalues are real
- Eigenvectors are orthogonal
- $\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top$ where $\mathbf{Q}$ is orthogonal

Positive Definite Matrices:

- All eigenvalues are positive
- $\mathbf{v}^\top\mathbf{A}\mathbf{v} > 0$ for all $\mathbf{v} \neq 0$

Trace and Determinant:

$$\text{tr}(\mathbf{A}) = \sum_{i=}^{n} \lambda_i, \quad \det(\mathbf{A}) = \prod_{i=}^{n} \lambda_i$$

1.7.9 Power Iteration Method

The power iteration method is a simple algorithm for finding the dominant eigenvalue and its corresponding eigenvector:

Algorithm:

1. Initialize a random vector $\mathbf{v}_0$ with $\|\mathbf{v}_0\| = 1$
2. For $k = 1, 2, \ldots$ until convergence:
   (a) Compute $\mathbf{w}_k = \mathbf{A}\mathbf{v}_{k-1}$
   (b) Compute $\mathbf{v}_k = \frac{\mathbf{w}_k}{\|\mathbf{w}_k\|}$

(c) Estimate eigenvalue: $\lambda^{(k)} = \mathbf{v}_k^{\top} \mathbf{A} \mathbf{v}_k$

Convergence: The method converges to the eigenvector corresponding to the largest eigenvalue at a rate proportional to $\left(\frac{\lambda_2}{\lambda_1}\right)^k$

Worked Example: Let $\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$, $\mathbf{v}_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$

Iteration 1: $\mathbf{w}_1 = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$, $\mathbf{v}_1 = \frac{1}{\sqrt{5}} \begin{bmatrix} 2 \\ 1 \end{bmatrix} \approx \begin{bmatrix} 0.894 \\ 0.44 \end{bmatrix}$

Iteration 2: $\mathbf{w}_2 = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 0.894 \\ 0.44 \end{bmatrix} = \begin{bmatrix} 2.235 \\ 1.78 \end{bmatrix}$, $\mathbf{v}_2 = \begin{bmatrix} 0.781 \\ 0.62 \end{bmatrix}$

Iteration 3: $\mathbf{w}_3 = \begin{bmatrix} 2.187 \\ 1.906 \end{bmatrix}$, $\mathbf{v}_3 = \begin{bmatrix} 0.754 \\ 0.657 \end{bmatrix}$

After several iterations, converges to $\mathbf{v} \approx \begin{bmatrix} 0.707 \\ 0.707 \end{bmatrix}$ (eigenvector for $\lambda = 3$).

1.7.10 Applications in Deep Learning

Principal Component Analysis (PCA)

PCA uses eigen decomposition of the covariance matrix to find directions of maximum vari-ance:

Algorithm:

1. Center the data: $\mathbf{X}_{\text{centered}} = \mathbf{X} - \mu$
2. Compute covariance matrix: $\mathbf{C} = \frac{1}{n-1} \mathbf{X}_{\text{centered}}^\top \mathbf{X}_{\text{centered}}$
3. Eigen decomposition: $\mathbf{C} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^\top$
4. Project data: $\mathbf{Y} = \mathbf{X}_{\text{centered}}\mathbf{V}_k$ (where $\mathbf{V}_k$ contains top-k eigenvectors)

Worked Example: Given data points: (1,1), (2,2), (3,3), (4,4)

Mean: $\mu = (2.5, 2.5)$

Centered data: $\mathbf{X}_{\text{centered}} = \begin{bmatrix} -1.5 & -1.5 \\ -0.5 & -0.5 \\ 0.5 & 0.5 \\ 1.5 & 1.5 \end{bmatrix}$

Covariance matrix: $\mathbf{C} = \frac{1}{3} \mathbf{X}_{\text{centered}}^\top \mathbf{X}_{\text{centered}} = \begin{bmatrix} 1.67 & 1.67 \\ 1.67 & 1.67 \end{bmatrix}$

Eigenvalues: $\lambda_1 = 3.33, \lambda_2 = 0$

Eigenvectors: $\mathbf{v}_1 = \begin{matrix} 0.707 \\ 0.707 \end{matrix}$, $\mathbf{v}_2 = \begin{matrix} 0.707 \\ 0.707 \end{matrix}$

The first principal component captures all the variance.

Neural Network Initialization

Xavier/Glorot Initialization: For linear layer $\mathbf{y} = \mathbf{W}\mathbf{x}$, to maintain variance through layers:

$$\text{Var}(W_{ij}) = \frac{2}{n_{\text{in}} + n_{\text{out}}}$$

where n_{in} and n_{out} are input and output dimensions.

He Initialization: For ReLU networks:

$$\text{Var}(W_{ij}) = \frac{2}{n_{\text{in}}}$$

Oscillation Analysis in Training

The training dynamics can be analyzed through the Hessian matrix $\mathbf{H}$ of the loss function:

Condition Number:

$$\kappa(\mathbf{H}) = \frac{\lambda_{\max}}{\lambda_{\min}}$$

Large condition numbers indicate ill-conditioning and slow convergence.

Spectral Analysis of Optimization: For gradient descent:

$$\theta_{t+1} = \theta_t - \eta \nabla L(\theta_t)$$

The convergence rate depends on the eigenvalues of the Hessian.

1.7.11 Theoretical Significance

Matrix Functions

For diagonalizable matrices, matrix functions can be computed via eigen decomposition:

$$f(\mathbf{A}) = \mathbf{V} f(\mathbf{\Lambda}) \mathbf{V}^{-1}$$

where $f(\mathbf{\Lambda}) = \text{diag}(f(\lambda_1), f(\lambda_2), \ldots, f(\lambda_n))$.

Examples:

- Matrix exponential: $e^{\mathbf{A}} = \mathbf{V} e^{\mathbf{\Lambda}} \mathbf{V}^{-1}$
- Matrix square root: $\mathbf{A}^{1/2} = \mathbf{V} \mathbf{\Lambda}^{1/2} \mathbf{V}^{-1}$
- Matrix inverse: $\mathbf{A}^{-1} = \mathbf{V} \mathbf{\Lambda}^{-1} \mathbf{V}^{-1}$

Stability Analysis

For linear dynamical system $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x}$:

- System is stable if $\text{Re}(\lambda_i) < 0$ for all i

- Largest eigenvalue determines the dominant behavior
- Eigenvectors represent modes of the system

Graph Theory

In graph convolutional networks, the graph Laplacian $\mathbf{L}$ plays a key role:

$$\mathbf{L} = \mathbf{D} - \mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^\top$$

where eigenvalues represent frequencies on the graph.

1.7.12 Numerical Implementation

Python Code Example

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

def eigen_decomposition_demo():
# Example matrix
A = np.array([[4, 1], [2, 3]])

# Eigen decomposition
eigenvalues, eigenvectors = np.linalg.eig(A)

print("Matrix A:")
print(A)
print("\nEigenvalues:",
eigenvalues)
print("Eigenvectors:")
print(eigenvectors)

# Verify decomposition
Lambda = np.diag(eigenvalues)
A_reconstructed = eigenvectors @ Lambda @ np.linalg.inv(eigenvectors)

print("\nReconstruction error:", np.linalg.norm(A - A_reconstructed))

# Power iteration method
def power_iteration(A,
num_iterations=100): n = A.shape[0]
v = np.random.rand(n)
v = v / np.linalg.norm(v)

for i in range(num_iterations):
Av = A @ v
v = Av / np.linalg.norm(Av)
lambda_est = v @ A @ v
```

```
return lambda_est, v
dominant_eval, dominant_evec =
power_iteration(A) print(f"\nPower
iteration result:")  print(f"Dominant
eigenvalue: {dominant_eval:.6f}")
print(f"Dominant eigenvector:
{dominant_evec}")

# PCA implementation
def pca_implementation(X):
# Center the data
X_centered = X - np.mean(X, axis=0)

# Compute covariance matrix
covariance =
np.cov(X_centered.T)

# Eigen decomposition
eigenvalues, eigenvectors = np.linalg.eig(covariance)

# Sort by descending
eigenvalues idx =
np.argsort(eigenvalues)[::-1]
eigenvalues =
eigenvalues[idx] eigenvectors
= eigenvectors[:, idx]

return eigenvalues, eigenvectors,

X_centered  # Example usage
X = np.array([[1, 1], [2, 2], [3, 3], [4, 4]])
eigenvalues, eigenvectors, X_centered = pca_implementation(X)
print("PCA Results:")
print("Eigenvalues:", eigenvalues)
print("Eigenvectors:", eigenvectors)
print("Explained variance ratio:", eigenvalues / np.sum(eigenvalues))
```

1.7.13 Advanced Topics

Generalized Eigenvalue

Problem For matrices **A** and **B**,

solve:

$$\mathbf{Av} = \lambda\mathbf{Bv}$$

Used in Fisher discriminant analysis and modal analysis.

Quantum Machine Learning

In quantum computing, eigen decomposition is fundamental for:

- Quantum phase estimation
- Solving quantum linear systems
- Quantum principal component analysis

Rayleigh Quotient

For symmetric matrix $\mathbf{A}$, the Rayleigh quotient:

$$R(\mathbf{x}) = \frac{\mathbf{x}^\top \mathbf{A}\mathbf{x}}{\mathbf{x}^\top \mathbf{x}}$$

satisfies $\lambda_{\min} \leq R(\mathbf{x}) \leq \lambda_{\max}$.

1.7.14 Exercises

Multiple Choice Questions

1. For a symmetric matrix **A**, which statement is true?

   (a) All eigenvalues are complex

   (b) Eigenvectors are always orthogonal

   (c) The matrix is always invertible

   (d) Eigenvalues sum to zero

   Answer: (b) Eigenvectors are always orthogonal

2. The power iteration method converges to:

   (a) The smallest eigenvalue

   (b) The eigenvalue with largest magnitude

   (c) The average of all eigenvalues

   (d) The determinant of the matrix

   Answer: (b) The eigenvalue with largest magnitude

3. If **A** has eigenvalues 2, 3, and 5, then det(**A**) is:

   (a) 10

   (b) 30

   (c) 8

   (d) Cannot be determined

   Answer: (b) 30 (product of eigenvalues)

Problem Solving Questions

1. Find the eigen decomposition of $\mathbf{A} = \begin{matrix} 5 & 2 \\ 2 & 2 \end{matrix}$ and verify your result.

2. Implement the power iteration method and apply it to find the dominant eigenvalue of $\mathbf{A} = \begin{matrix} 3 & 1 \\ 1 & 3 \end{matrix}$.

3. Explain how eigen decomposition is used in PCA and derive the relationship between eigenvalues and explained variance.

This comprehensive treatment of eigen decomposition provides both theoretical under-standing and practical implementation knowledge essential for deep learning applications.

Exercises

Multiple Choice Questions

1. Which operation is **not** defined for matrices $\mathbf{A} \in \mathbb{R}^{2\times3}$ and $\mathbf{B} \in \mathbb{R}^{3\times2}$?

   (a) $\mathbf{AB}$
   (b) $\mathbf{BA}$
   (c) $\mathbf{A} \odot \mathbf{B}$
   (d) $\mathbf{A}^\top\mathbf{B}$

   Answer: (c) Hadamard product requires same dimensions.

2. The eigenvector of $\begin{matrix} 4 & 1 \\ 2 & 3 \end{matrix}$ corresponding to $\lambda = 5$ is:

   (a) $[1, 1]^\top$
   (b) $[1, -1]^\top$
   (c) $[1, 2]^\top$
   (d) $[2, 1]^\top$

   Answer: (a) Verify: $\begin{matrix} 4 & 1 \\ 2 & \end{matrix} \begin{matrix} 1 \\ 1 \end{matrix} = \begin{matrix} 5 \\ 5 \end{matrix} = 5 \begin{matrix} 1 \\ 1 \end{matrix}$

3. Which matrix is **not** symmetric?

   (a) $\begin{matrix} 2 & 1 \\ 1 & 2 \end{matrix}$
   (b) $\begin{matrix} 3 & 0 \\ 0 & 3 \end{matrix}$
   (c) $\begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix}$
   (d) $\begin{matrix} 5 & -1 \\ -1 & 5 \end{matrix}$

   Answer: (c) Not equal to its transpose.

4. The L^2 norm of vector $[3, 0, -4]^\top$ is:

   (a) 7
   (b) 5
   (c) 1
   (d) 25

   Answer: (b) $\sqrt{9 + 0 + 16} = 5$

5. Eigen decomposition helps in deep learning for:

(a) Weight initialization

(b) Learning rate adjustment

(c) Dimensionality reduction

(d) Data augmentation

Answer: (c) PCA uses eigen decomposition for dimensionality reduction.

Long Answer/Essay Questions

1. **(Remembering)** Define and provide examples of: scalar, vector, matrix, and tensor. Explain their relationships.

   Answer: Scalars are non-indexed numbers (e.g., 3.14). Vectors are 1D arrays (e.g., [1, 2, 3]). Matrices are 2D arrays (e.g., $\begin{matrix}1 & 2\\ 3 & 4\end{matrix}$). Tensors are n-dimensional arrays.

   Scalars are rank-0 tensors, vectors rank-1, matrices rank-2.

2. **(Understanding)** Explain why matrix multiplication is not commutative. Provide a concrete example demonstrating $\mathbf{AB} \neq \mathbf{BA}$.

   Answer: Matrix multiplication involves dot products between rows and columns, and order affects these computations. Example:

$$\mathbf{A} = \begin{matrix}1 & 2\\ 3 & \end{matrix}, \mathbf{B} = \begin{matrix}5 & 6\\ 7 & \end{matrix}$$

$$\mathbf{AB} = \begin{matrix}19 & 22\\ 43 & \end{matrix} \neq \mathbf{BA} = \begin{matrix}23 & 34\\ 31 & \end{matrix}$$

3. **(Applying)** Calculate eigenvalues and eigenvectors of $\mathbf{A} = \begin{matrix} 5 & 2 \\ 2 & 2 \end{matrix}$ and verify the eigen decomposition.

   Answer: Characteristic equation: $\det(\mathbf{A} - \lambda\mathbf{I}) = (5-\lambda)(2-\lambda) - 4 = 0$
   $\lambda^2 - 7\lambda + 6 = 0 \Rightarrow \lambda_1 = 6, \lambda_2 = 1$
   For $\lambda_1 = 6$: $(\mathbf{A} - 6\mathbf{I})\mathbf{v} = 0 \Rightarrow \begin{matrix} -1 & 2 \\ 2 & -4 \end{matrix} \mathbf{v}_1 = 0 \Rightarrow \mathbf{v}_1 = [2, 1]^\top$

   For $\lambda_2 = 1$: $(\mathbf{A} - \mathbf{I})\mathbf{v} = 0 \Rightarrow \begin{matrix} 4 & 2 \\ 2 & 1 \end{matrix} \mathbf{v}_2 = 0 \Rightarrow \mathbf{v}_2 = [1, -2]^\top$
   Verification: $\mathbf{A}\mathbf{v}_1 = \begin{matrix} 12 \\ 6 \end{matrix} = 6\mathbf{v}_1$, $\mathbf{A}\mathbf{v}_2 = \begin{matrix} 1 \\ -2 \end{matrix} = 1\mathbf{v}_2$

4. **(Analyzing)** Compare and contrast L^1 and L^2 norms. Discuss their applications in regularization techniques for deep learning.

   Answer: L^2 norm is differentiable and promotes dense solutions, used in ridge re-gression. L^1 norm is not differentiable at origin but promotes sparsity, used in Lasso regularization. In deep learning, L^2 weight decay prevents overfitting by penalizing large weights, while L^1 can create sparse networks.

5. **(Evaluating)** Critique the statement: "Eigen decomposition is essential for all deep learning applications." Provide arguments supporting and refuting this claim.

   Answer: Supporting: Eigen decomposition provides fundamental understanding of linear transformations, used in PCA for data preprocessing, and analyzing network dynamics. Refuting: Many deep learning operations use stochastic gradient descent and non-linear activations where eigen decomposition has limited direct application. Modern networks often rely on approximate methods rather than exact decomposition due to computational constraints.

1.8 Probability & Information Theory

1.8.1 Random Variables

A random variable X is a function that maps outcomes of random phenomena to numerical values.

Discrete Random Variable: Takes countable values (e.g., dice rolls)

$$P(X = x_i) = p_i, \quad \sum_i p_i = 1$$

Continuous Random Variable: Takes uncountable values (e.g., temperature)

$$P(a \le X \le b) = \int_a^b f(x)dx, \qquad \int_{-\infty}^{\infty} f(x)dx = 1$$

1.8.2 Probability Distributions Common

Discrete Distributions

- **Bernoulli:** $P(X = k) = p^k(1-p)^{1-k}$ for $k \in \{0, 1\}$
- **Binomial:** $P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$
- **Multinomial:** Generalization of binomial to multiple outcomes

Common Continuous Distributions

- **Gaussian/Normal:** $f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$
- **Uniform:** $f(x) = \frac{1}{b-a}$ for $x \in [a, b]$

1.8.3 Marginal and Conditional Probability

Marginal Probability:

$$P(X = x) = \sum_y P(X = x, Y = y)$$

Conditional Probability:

$$P(Y = y|X = x) = \frac{P(X = x, Y = y)}{P(X = x)}$$

1.8.4 Expectation, Variance, Covariance

Expectation:

$$\mathrm{E}[X] = \sum_x xP(X = x) \quad \text{(discrete)}$$

$$\mathrm{E}[X] = \int_x xf(x)dx \quad \text{(continuous)}$$

Variance:

$$\mathrm{Var}(X) = \mathrm{E}[(X - \mathrm{E}[X])^2] = \mathrm{E}[X^2] - (\mathrm{E}[X])^2$$

Covariance:

$$\mathrm{Cov}(X, Y) = \mathrm{E}[(X - \mathrm{E}[X])(Y - \mathrm{E}[Y])]$$

1.8.5 Baye' Rule

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} = \frac{P(B|A)P(A)}{\sum_i P(B|A_i)P(A_i)}$$

Worked Example: Medical test with 99% accuracy, disease prevalence 1%

$$P(\text{disease} \mid \text{positive}) = \frac{0.99 \times 0.01}{0.99 \times 0.01 + 0.01 \times 0.99} = 0.5$$

1.9 Numerical Computation

1.9.1 Overflow and Underflow: Mathematical Analysis

Definition 1.9.1 (Underflow)**.** *When numbers near zero are rounded to zero in floating-point arithmetic. For x with $|x| < \epsilon_{machine}$, where $\epsilon_{machine}$ is machine epsilon.*

Definition 1.9.2 (Overflow)**.** *When numbers exceed the maximum representable value. For IEEE 754 double precision:* $\approx 1.8 \times 10^{308}$.

Example: Softmax Stability For $\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$, if x_i are large, $\exp(x_i)$ may overflow.

Solution:

$$\text{softmax}(x_i) = \frac{\exp(x_i - \max(\mathbf{x}))}{\sum_j \exp(x_j - \max(\mathbf{x}))}$$

This is mathematically equivalent but numerically stable.

1.9.2 Gradient-Based Optimization: Complete Analysis

Gradient Descent Update Derivation from Taylor Series: For function $f: \mathbb{R}^n \to \mathbb{R}$, Taylor expansion around $\mathbf{x}_k$:

$$f(\mathbf{x}_k + \Delta\mathbf{x}) = f(\mathbf{x}_k) + \nabla f(\mathbf{x}_k)^T \Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^T \mathbf{H}(\mathbf{x}_k)\Delta\mathbf{x} + O(\|\Delta\mathbf{x}\|^3)$$

Ignoring Hessian and higher-order terms:

$$f(\mathbf{x}_k + \Delta\mathbf{x}) \approx f(\mathbf{x}_k) + \nabla f(\mathbf{x}_k)^T \Delta\mathbf{x}$$

To decrease f, choose $\Delta\mathbf{x} = -\eta\nabla f(\mathbf{x}_k)$:

$$f(\mathbf{x}_k - \eta\nabla f(\mathbf{x}_k)) \approx f(\mathbf{x}_k) - \eta\|\nabla f(\mathbf{x}_k)\|^2$$

Thus the update rule:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \eta\nabla f(\mathbf{x}_k)$$

Convergence Analysis for Convex Functions:

Theorem 1.9.1 (Gradient Descent Convergence)**.** *For L-smooth convex function f (i.e., $\|\nabla f(\mathbf{x}) - \nabla f(\mathbf{y})\| \le L\|\mathbf{x} - \mathbf{y}\|$), gradient descent with $\eta = \frac{1}{L}$ satisfies:*

$$f(\mathbf{x}_k) - f(\mathbf{x}^*) \le \frac{L\|\mathbf{x}_0 - \mathbf{x}^*\|^2}{2k}$$

where $\mathbf{x}^*$ is the minimizer.

Proof: From L-smoothness:

$$f(\mathbf{y}) \leq f(\mathbf{x}) + \nabla f(\mathbf{x})^T(\mathbf{y} - \mathbf{x}) + \frac{L}{2}\|\mathbf{y} - \mathbf{x}\|^2$$

Apply at $\mathbf{x} = \mathbf{x}_k$, $\mathbf{y} = \mathbf{x}_{k+1} = \mathbf{x}_k - \eta \nabla f(\mathbf{x}_k)$:

$$f(\mathbf{x}_{k+1}) \leq f(\mathbf{x}_k) - \eta \|\nabla f(\mathbf{x}_k)\|^2 + \frac{L\eta^2}{2} \|\nabla f(\mathbf{x}_k)\|^2$$

$$= f(\mathbf{x}_k) - \eta \left(1 - \frac{L\eta}{2}\right) \|\nabla f(\mathbf{x}_k)\|^2$$

With $\eta = \frac{1}{L}$:

$$f(\mathbf{x}_{k+1}) \leq f(\mathbf{x}_k) - \frac{1}{2L} \|\nabla f(\mathbf{x}_k)\|^2$$

From convexity: $f(\mathbf{x}_k) - f(\mathbf{x}^*) \leq \nabla f(\mathbf{x}_k)^T (\mathbf{x}_k - \mathbf{x}^*) \leq \|\nabla f(\mathbf{x}_k)\| \|\mathbf{x}_k - \mathbf{x}^*\|$ Combining and telescoping gives the result.

1.9.3 Constrained Optimization: KKT Conditions Detailed

Definition 1.9.3 (Lagrangian). *For problem* $\min_{\mathbf{x}} f(\mathbf{x})$ *subject to* $g_i(\mathbf{x}) \leq 0$, $h_j(\mathbf{x}) = 0$*:*

$$L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\nu}) = f(\mathbf{x}) + \sum_i \lambda_i g_i(\mathbf{x}) + \sum_j \nu_j h_j(\mathbf{x})$$

Theorem 1.9.2 (Karush-Kuhn-Tucker Conditions). *For optimal point* $\mathbf{x}^*$ *with Lagrange multipliers* λ_i^*, ν_j^**:*

1. *Stationarity:* $\nabla f(\mathbf{x}^*) + \sum_i \lambda_i^* \nabla g_i(\mathbf{x}^*) + \sum_j \nu_j^* \nabla h_j(\mathbf{x}^*) = 0$
2. *Primal feasibility:* $g_i(\mathbf{x}^*) \leq 0$, $h_j(\mathbf{x}^*) = 0$
3. *Dual feasibility:* $\lambda_i^* \geq 0$
4. *Complementary slackness:* $\lambda_i^* g_i(\mathbf{x}^*) = 0$

1.9.4 Linear Least Squares: Complete Derivation

Problem: Minimize $\|\mathbf{Ax} - \mathbf{b}\|^2$ where $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{b} \in \mathbb{R}^m$.

Solution:

$$f(\mathbf{x}) = \|\mathbf{Ax} - \mathbf{b}\|^2 = (\mathbf{Ax} - \mathbf{b})^T (\mathbf{Ax} - \mathbf{b})$$
$$= \mathbf{x}^T \mathbf{A}^T \mathbf{Ax} - 2\mathbf{b}^T \mathbf{Ax} + \mathbf{b}^T \mathbf{b}$$

Take gradient:

$$\nabla f(\mathbf{x}) = 2\mathbf{A}^T \mathbf{Ax} - 2\mathbf{A}^T \mathbf{b}$$

Set to zero:

If $\mathbf{A}^T \mathbf{A}$ is invertible:

A **b** (Normal Equations)
T

$$\mathbf{x} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b}$$

A
x

=

A
T

Geometric Interpretation: **Ax** is projection of **b** onto column space of **A**, and residual **bAx** is orthogonal to column space.

Computational Complexity:

- Forming $\mathbf{A}^T\mathbf{A}$: $O(mn^2)$
- Solving $n \times n$ system: $O(n^3)$
- Total: $O(mn^2 + n^3)$

1.10 Comprehensive Example Problems with Detailed Solutions

1.10.1 Problem 1: Matrix Calculus with Detailed Steps

Problem: Given $f(\mathbf{W}) = \|\mathbf{XW} - \mathbf{Y}\|_F^2$, where $\mathbf{X} \in \mathrm{R}^{m \times n}$, $\mathbf{W} \in \mathrm{R}^{n \times p}$, $\mathbf{Y} \in \mathrm{R}^{m \times p}$, find $\frac{\partial f}{\partial \mathbf{W}}$.

Step-by-Step Solution:

Step 1: Write Frobenius norm in trace form:

$$f(\mathbf{W}) = \mathrm{tr}\ (\mathbf{XW} - \mathbf{Y})^T(\mathbf{XW} - \mathbf{Y})$$

Step 2: Expand:

$$\begin{aligned} f(\mathbf{W}) &= \mathrm{tr}(\mathbf{W}^T\mathbf{X}^T\mathbf{XW} - \mathbf{W}^T\mathbf{X}^T\mathbf{Y} - \mathbf{Y}^T\mathbf{XW} + \mathbf{Y}^T\mathbf{Y}) \\ &= \mathrm{tr}(\mathbf{W}^T\mathbf{X}^T\mathbf{XW}) - 2\,\mathrm{tr}(\mathbf{W}^T\mathbf{X}^T\mathbf{Y}) + \mathrm{tr}(\mathbf{Y}^T\mathbf{Y}) \end{aligned}$$

since $\mathrm{tr}(\mathbf{W}^T\mathbf{X}^T\mathbf{Y}) = \mathrm{tr}(\mathbf{Y}^T\mathbf{XW})$.

Step 3: Apply matrix derivative rules:

- $\frac{\partial}{\partial \mathbf{W}} \mathrm{tr}(\mathbf{W}^T\mathbf{AW}) = (\mathbf{A} + \mathbf{A}^T)\mathbf{W}$
- $\frac{\partial}{\partial \mathbf{W}} \mathrm{tr}(\mathbf{W}^T\mathbf{B}) = \mathbf{B}$

Step 4: Compute derivatives:

$$\begin{aligned} \frac{\partial f}{\partial \mathbf{W}} &= \frac{\partial}{\partial \mathbf{W}} \mathrm{tr}(\mathbf{W}^T\mathbf{X}^T\mathbf{XW}) - 2\frac{\partial}{\partial \mathbf{W}} \mathrm{tr}(\mathbf{W}^T\mathbf{X}^T\mathbf{Y}) + 0 \\ &= (\mathbf{X}^T\mathbf{X} + (\mathbf{X}^T\mathbf{X})^T)\mathbf{W} - 2\mathbf{X}^T\mathbf{Y} \\ &= 2\mathbf{X}^T\mathbf{XW} - 2\mathbf{X}^T\mathbf{Y} \\ &= 2\mathbf{X}^T(\mathbf{XW} - \mathbf{Y}) \end{aligned}$$

Final Answer: $\frac{\partial f}{\partial \mathbf{W}} = 2\mathbf{X}^T(\mathbf{XW} - \mathbf{Y})$

1.10.2 Problem 2: Expectation of Quadratic Form with Gaussian Variables

Problem: Let $\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$. Derive $\mathrm{E}[\mathbf{x}^T\mathbf{Ax}]$ where $\mathbf{A}$ is symmetric.

Step-by-Step Solution:

Step 1: Use trace property $\mathbf{x}^T\mathbf{Ax} = \mathrm{tr}(\mathbf{x}^T\mathbf{Ax}) = \mathrm{tr}(\mathbf{Axx}^T)$:

$$\mathrm{E}[\mathbf{x}^T\mathbf{Ax}] = \mathrm{E}[\mathrm{tr}(\mathbf{Axx}^T)] = \mathrm{tr}(\mathbf{A}\mathrm{E}[\mathbf{xx}^T])$$

Step 2: Compute $\mathrm{E}[\mathbf{x}\mathbf{x}^T]$: Let $\mathbf{x} = \boldsymbol{\mu} + \boldsymbol{\epsilon}$ where $\boldsymbol{\epsilon} \sim \mathrm{N}(\mathbf{0}, \boldsymbol{\Sigma})$:

$$\begin{aligned}\mathrm{E}[\mathbf{x}\mathbf{x}^T] &= \mathrm{E}[(\boldsymbol{\mu} + \boldsymbol{\epsilon})(\boldsymbol{\mu} + \boldsymbol{\epsilon})^T] \\ &= \mathrm{E}[\boldsymbol{\mu}\boldsymbol{\mu}^T + \boldsymbol{\mu}\boldsymbol{\epsilon}^T + \boldsymbol{\epsilon}\boldsymbol{\mu}^T + \boldsymbol{\epsilon}\boldsymbol{\epsilon}^T] \\ &= \boldsymbol{\mu}\boldsymbol{\mu}^T + \boldsymbol{\mu}\mathrm{E}[\boldsymbol{\epsilon}^T] + \mathrm{E}[\boldsymbol{\epsilon}]\boldsymbol{\mu}^T + \mathrm{E}[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^T] \\ &= \boldsymbol{\mu}\boldsymbol{\mu}^T + \mathbf{0} + \mathbf{0} + \boldsymbol{\Sigma} \\ &= \boldsymbol{\mu}\boldsymbol{\mu}^T + \boldsymbol{\Sigma}\end{aligned}$$

Step 3: Substitute back:

$$\begin{aligned}\mathrm{E}[\mathbf{x}^T\mathbf{A}\mathbf{x}] &= \mathrm{tr}(\mathbf{A}(\boldsymbol{\mu}\boldsymbol{\mu}^T + \boldsymbol{\Sigma})) \\ &= \mathrm{tr}(\mathbf{A}\boldsymbol{\mu}\boldsymbol{\mu}^T) + \mathrm{tr}(\mathbf{A}\boldsymbol{\Sigma}) \\ &= \boldsymbol{\mu}^T\mathbf{A}\boldsymbol{\mu} + \mathrm{tr}(\mathbf{A}\boldsymbol{\Sigma})\end{aligned}$$

since $\mathrm{tr}(\mathbf{A}\boldsymbol{\mu}\boldsymbol{\mu}^T) = \boldsymbol{\mu}^T\mathbf{A}\boldsymbol{\mu}$.

Final Answer: $\mathrm{E}[\mathbf{x}^T\mathbf{A}\mathbf{x}] = \boldsymbol{\mu}^T\mathbf{A}\boldsymbol{\mu} + \mathrm{tr}(\mathbf{A}\boldsymbol{\Sigma})$

1.10.3 Problem 3: SVD Low-Rank Approximation with Proof

Problem: Given $\mathbf{A} \in \mathrm{R}^{m \times n}$ with SVD $\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T$, find rank-$k$ approximation $\mathbf{A}_k$ mini-mizing $\|\mathbf{A} - \mathbf{B}\|_F$ over all rank-k matrices $\mathbf{B}$.

Detailed Proof:

Let $\mathbf{A} = \sum_{i=}^{r} \sigma_i \mathbf{u}_i \mathbf{v}_i^T$ with $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$, $r = \mathrm{rank}(\mathbf{A})$.

Claim: $\mathbf{A}_k = \sum_{i=}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^T$ is optimal.

Proof: Let $\mathbf{B}$ be any rank-k matrix with SVD $\mathbf{B} = \tilde{\mathbf{U}}\tilde{\boldsymbol{\Sigma}}\tilde{\mathbf{V}}^T$. Step 1: Use unitary invariance of Frobenius norm:

$$\|\mathbf{A} - \mathbf{B}\|_F^2 = \|\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T - \mathbf{B}\|_F^2 = \|\boldsymbol{\Sigma} - \mathbf{U}^T\mathbf{B}\mathbf{V}\|_F^2$$

Let $\mathbf{C} = \mathbf{U}^T\mathbf{B}\mathbf{V}$, then $\mathrm{rank}(\mathbf{C}) \leq k$. Step 2: We need to minimize $\|\boldsymbol{\Sigma} - \mathbf{C}\|_F^2 = \sum_{j=}^{n} (\sigma_{ij} - c_{ij})^2$ with $\mathrm{rank}(\mathbf{C}) \leq k$.

$$i= \sum^{1}$$

$$1 \sum_{m}$$

Step 3: By Eckart-Young-Mirsky theorem, the optimal $\mathbf{C}$ is diagonal with the k largest singular values:

$$\mathbf{C}^* = \mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_k, 0, \ldots, 0)$$

with appropriate zero padding.

Step 4: Then $\mathbf{B}^* = \mathbf{U}\mathbf{C}^*\mathbf{V}^T = \sum_{i=}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^T = \mathbf{A}_k$.

The approximation error is:

$$\|\mathbf{A} - \mathbf{A}_k\|_F^2 = \sum_{i=k+1}^{r} \sigma_i^2$$

Final Answer: $\mathbf{A}_k = \sum_{i=}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^T = \mathbf{U}_{:,1:k} \mathbf{\Sigma}_{1:k,1:k} \mathbf{V}_{:,1:k}^T$

1.10.4 Problem 4: Gradient Descent Convergence Proof

Problem: For convex, L-smooth function f, show that gradient descent with step size $\eta = \frac{1}{L}$ satisfies:

$$f(\mathbf{x}_k) - f(\mathbf{x}^*) \leq \frac{L\|\mathbf{x}_0 - \mathbf{x}^*\|^2}{2k}$$

Complete Proof:

Let $\Delta_k = f(\mathbf{x}_k) - f(\mathbf{x}^*)$.

Step 1: From L-smoothness:

$$f(\mathbf{y}) \leq f(\mathbf{x}) + \nabla f(\mathbf{x})^T(\mathbf{y} - \mathbf{x}) + \frac{L}{2}\|\mathbf{y} - \mathbf{x}\|^2$$

Step 2: Apply at $\mathbf{x} = \mathbf{x}_k$, $\mathbf{y} = \mathbf{x}_{k+1} = \mathbf{x}_k - \eta\nabla f(\mathbf{x}_k)$:

$$f(\mathbf{x}_{k+1}) \leq f(\mathbf{x}_k) - \eta\|\nabla f(\mathbf{x}_k)\|^2 + \frac{L\eta^2}{2}\|\nabla f(\mathbf{x}_k)\|^2$$

$$= f(\mathbf{x}_k) - \eta\left(1 - \frac{L\eta}{2}\right)\|\nabla f(\mathbf{x}_k)\|^2$$

Step 3: With $\eta = \frac{1}{L}$:

$$f(\mathbf{x}_{k+1}) \leq f(\mathbf{x}_k) - \frac{1}{2L}\|\nabla f(\mathbf{x}_k)\|^2$$

Step 4: From convexity:

$$f(\mathbf{x}_k) - f(\mathbf{x}^*) \leq \nabla f(\mathbf{x}_k)^T(\mathbf{x}_k - \mathbf{x}^*) \leq \|\nabla f(\mathbf{x}_k)\|\,\|\mathbf{x}_k - \mathbf{x}^*\|$$

So $\|\nabla f(\mathbf{x}_k)\| \geq \frac{f(\mathbf{x}_k) - f(\mathbf{x}^*)}{\|\mathbf{x}_k - \mathbf{x}^*\|}$.

Step 5: Combining:

$$\Delta_{k+1} \leq \Delta_k - \frac{1}{2L}\|\nabla f(\mathbf{x}_k)\|^2$$

$$\leq \Delta_k - \frac{\Delta_k^2}{2L\|\mathbf{x}_k - \mathbf{x}^*\|^2}$$

$$\leq \Delta_k - \frac{\Delta_k^2}{2L\|\mathbf{x}_0 - \mathbf{x}^*\|^2} \quad \text{(since } \|\mathbf{x}_k - \mathbf{x}^*\| \leq \|\mathbf{x}_0 - \mathbf{x}^*\| \text{ for GD)}$$

Step 6: Rearranging:

$$\frac{1}{\Delta_{k+1}} \geq \frac{1}{\Delta_k} + \frac{\Delta_k}{2L\|\mathbf{x}_0 - \mathbf{x}^*\|^2} \cdot \frac{1}{\Delta_k} \quad \frac{1}{\Delta_k} + \frac{1}{2L\|\mathbf{x}_0 - \mathbf{x}^*\|^2}$$

$\geq$

Step 7: Telescoping from $k = 0$ to $k - 1$:

$$\frac{1}{\Delta_k} \geq \frac{1}{\Delta_0} + \frac{k}{2L\|\mathbf{x}_0 - \mathbf{x}^*\|^2} \geq \frac{k}{2L\|\mathbf{x}_0 - \mathbf{x}^*\|^2}$$

Thus:

$$\Delta_k \leq \frac{2L\|\mathbf{x}_0 - \mathbf{x}^*\|^2}{k}$$

Final Answer: $f(\mathbf{x}_k) - f(\mathbf{x}^*) \leq \frac{2L\|\mathbf{x}_0 - \mathbf{x}^*\|^2}{k}$

1.10.5 Problem 5: Bayesian Inference with Gaussian Distribu-tions

Problem: Given prior $p(\theta) = \mathrm{N}(\mu_0, \sigma_0^2)$ and likelihood $p(x|\theta) = \mathrm{N}(\theta, \sigma^2)$ for i.i.d. data $\{x_i\}_{i=}^{n}$, find posterior $p(\theta|x_{1:n})$.

Detailed Derivation:

Step 1: Bayes' theorem: $p(\theta|x_{1:n}) \propto p(\theta) \prod_{i=1} p(x_i|\theta)$

Step 2: Substitute Gaussian forms:

$$p(\theta|x_{1:n}) \propto \exp\left(-\frac{(\theta-\mu_0)^2}{2\sigma_0^2}\right) \prod_{i=} \exp\left(-\frac{(x_i-\theta)^2}{2\sigma^2}\right)$$

$$= \exp\left(-\frac{(\theta-\mu_0)^2}{2\sigma_0^2} - \frac{1}{2\sigma^2}\sum_{i=1} (x_i-\theta)^2\right)$$

Step 3: Complete the square in θ:

$$\frac{(\theta-\mu_0)^2}{\sigma_0^2} + \frac{1}{\sigma^2}\sum_{i=1}^{n}(x_i-\theta)^2$$

$$= \frac{\theta^2 - 2\theta\mu_0 + \mu_0^2}{\sigma_0^2} + \frac{1}{\sigma^2}\sum_{i=}^{n}(x_i^2 - 2x_i\theta + \theta^2)$$

$$= \theta^2\left(\frac{1}{\sigma_0^2} + \frac{n}{\sigma^2}\right) - 2\theta\left(\frac{\mu_0}{\sigma_0^2} + \frac{\sum x_i}{\sigma^2}\right) + \text{constants}$$

Step 4: This is quadratic in θ, so posterior is Gaussian $p(\theta|x_{1:n}) = \mathrm{N}(\mu_n, \sigma^2)$ where:

$$\frac{1}{\sigma_n^2} = \frac{1}{\sigma_0^2} + \frac{n}{\sigma^2}, \quad \mu_n = \sigma_n^2\left(\frac{\mu_0}{\sigma_0^2} + \frac{n\bar{x}}{\sigma^2}\right)$$

with $\bar{x} = \frac{1}{n}\sum_{i=}^{n} x_i$.

Interpretation:

- Posterior precision = prior precision + data precision
- Posterior mean = weighted average of prior mean and sample mean
- As $n \to \infty$, posterior concentrates around $\bar{x}$

Final Answer:

$$p(\theta|x_{1:n}) = \mathcal{N}\left(\frac{\frac{\mu_0}{\sigma_0^2} + \frac{n\bar{x}}{\sigma^2}}{\frac{1}{\sigma_0^2} + \frac{n}{\sigma^2}}, \frac{1}{\frac{1}{\sigma_0^2} + \frac{n}{\sigma^2}}\right)$$

1.11 Key Theorems and Proofs

1.11.1 Theorem: Orthogonality of Eigenvectors for Symmetric Matrices

Theorem: For a real symmetric matrix $\mathbf{A}$, eigenvectors corresponding to distinct eigenvalues are orthogonal.

Proof: Let $\mathbf{A}\mathbf{v}_1 = \lambda_1\mathbf{v}_1$ and $\mathbf{A}\mathbf{v}_2 = \lambda_2\mathbf{v}_2$ with $\lambda_1 \neq \lambda_2$.

Multiply first equation by $\mathbf{v}_2^T$:

$$\mathbf{v}_2^T\mathbf{A}\mathbf{v}_1 = \lambda_1\mathbf{v}_2^T\mathbf{v}_1$$

Transpose both sides (using $\mathbf{A}^T = \mathbf{A}$):

$$(\mathbf{v}_2^T\mathbf{A}\mathbf{v}_1)^T = \mathbf{v}_1^T\mathbf{A}^T\mathbf{v}_2 = \mathbf{v}_1^T\mathbf{A}\mathbf{v}_2 = \lambda_2\mathbf{v}_1^T\mathbf{v}_2$$

But $\mathbf{v}_1^T\mathbf{v}_2 = \mathbf{v}_2^T\mathbf{v}_1$, so:

$$\lambda_1\mathbf{v}_2^T\mathbf{v}_1 = \lambda_2\mathbf{v}_2^T\mathbf{v}_1$$

Since $\lambda_1 \neq \lambda_2$, we must have $\mathbf{v}_2^T\mathbf{v}_1 = 0$, i.e., $\mathbf{v}_1$ and $\mathbf{v}_2$ are orthogonal.

1.11.2 Theorem: Real Eigenvalues of Symmetric Matrices

Theorem: All eigenvalues of a real symmetric matrix are real.

Proof: Let $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$ with $\mathbf{v} \neq \mathbf{0}$.

Take complex conjugate: $\mathbf{A}\bar{\mathbf{v}} = \bar{\lambda}\bar{\mathbf{v}}$ (since $\mathbf{A}$ is real). Multiply $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$ by $\bar{\mathbf{v}}^T$ on left:

$$\bar{\mathbf{v}}^T\mathbf{A}\mathbf{v} = \lambda\bar{\mathbf{v}}^T\mathbf{v}$$

Transpose of $\mathbf{A}\bar{\mathbf{v}} = \bar{\lambda}\bar{\mathbf{v}}$:

$$\bar{\mathbf{v}}^T\mathbf{A}^T = \bar{\lambda}\bar{\mathbf{v}}^T \Rightarrow \bar{\mathbf{v}}^T\mathbf{A} = \bar{\lambda}\bar{\mathbf{v}}^T \text{ (since } \mathbf{A}^T = \mathbf{A}\text{)}$$

Multiply by $\mathbf{v}$ on right:

$$\bar{\mathbf{v}}^T\mathbf{A}\mathbf{v} = \bar{\lambda}\bar{\mathbf{v}}^T\mathbf{v}$$

Comparing with earlier equation: $\lambda\bar{\mathbf{v}}^T\mathbf{v} = \bar{\lambda}\bar{\mathbf{v}}^T\mathbf{v}$.

Since $\bar{\mathbf{v}}^T\mathbf{v} = \|\mathbf{v}\|^2 > 0$, we have $\lambda = \bar{\lambda}$, so λ is real.

1.11.3 Information Theory Concepts

Entropy: Measure of uncertainty

$$H(X) = -\sum_{x \in X} P(x) \log P(x)$$

Cross-Entropy: Measure of difference between two distributions

$$H(P, Q) = -\sum_{x \in X} P(x) \log Q(x)$$

Kullback-Leibler Divergence: Measure of how one distribution diverges from another

$$D_{KL}(P||Q) = \sum_{x \in X} P(x) \log \frac{P(x)}{Q(x)}$$

Exercises

Multiple Choice Questions

1. In SVD decomposition $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$, the columns of $\mathbf{U}$ are eigenvectors of:

   (a) $\mathbf{A}^\top\mathbf{A}$

   (b) $\mathbf{A}\mathbf{A}^\top$

   (c) $\mathbf{A}$

   (d) $\mathbf{\Sigma}$

   Answer: (b) $\mathbf{A}\mathbf{A}^\top$

2. The primary goal of PCA is to:

   (a) Increase dimensionality of data

   (b) Find correlated features

   (c) Maximize variance along principal components

   (d) Minimize computational complexity

   Answer: (c) Maximize variance along principal components

3. For two independent random variables X and Y, which statement is true?

   (a) $P(X|Y) = P(X)$

   (b) $P(X, Y) = P(X) + P(Y)$

   (c) $\text{Cov}(X, Y) = 1$

   (d) $\text{E}[XY] = 0$

   Answer: (a) $P(X|Y) = P(X)$ for independent variables

4. In Bayes' rule, $P(A)$ is called the:

   (a) Likelihood

   (b) Evidence

   (c) Posterior

   (d) Prior

   Answer: (d) Prior probability

5. The entropy of a fair coin flip is:

   (a) 0 bits

   (b) 1 bit

   (c) 0.5 bits

   (d) 2 bits

   Answer: (b) $H(X) = -0.5\log_2(0.5) - 0.5\log_2(0.5) = 1$ bit

Long Answer/Essay Questions

1. **(Remembering)** Define SVD and list its key components. Explain the relationship between singular values and eigenvalues.

   Answer: SVD decomposes matrix $\mathbf{A}$ as $\mathbf{A} = \mathbf{U\Sigma V}^\top$ where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices containing left and right singular vectors, and $\mathbf{\Sigma}$ is diagonal with singular values. Singular values are square roots of eigenvalues of $\mathbf{A}^\top\mathbf{A}$.

2. **(Understanding)** Explain the intuition behind PCA using geometric interpretation. Why do we center the data before applying PCA?

   Answer: PCA finds orthogonal directions of maximum variance. Geometrically, it rotates the coordinate system to align with directions where data is most spread out. We center data to ensure the first principal component captures direction of maxi-mum variance rather than the mean direction. Without centering, PCA might find components that mainly reflect the data mean.

3. **(Applying)** Given the dataset: (1,2), (2,4), (3,6), (4,8), perform PCA and find the principal components. What percentage of variance is explained by the first compo-nent?

   Answer:

   (a) Center data: Mean = (2.5,5), Centered: (-1.5,-3), (-0.5,-1), (0.5,1), (1.5,3)

   (b) Covariance matrix: $\mathbf{C} = \begin{matrix} 1.67 & 3.33 \\ 3.33 & 6.67 \end{matrix}$

   (c) Eigenvalues: $\lambda_1 = 8.33, \lambda_2 = 0$

   (d) Eigenvectors: $\mathbf{v}_1 = [0.447, 0.894]^\top$, $\mathbf{v}_2 = [-0.894, 0.447]^\top$

   (e) Variance explained by first component: 100% (since $\lambda_2 = 0$)

4. **(Analyzing)** Compare and contrast covariance and correlation. Discuss their limita-tions in measuring relationships between variables, especially in the context of deep learning.

   Answer: Covariance measures direction of linear relationship but is scale-dependent. Correlation normalizes covariance to [−1, 1] range, making it

scale-invariant. Lim-itations: Both only capture linear relationships; nonlinear dependencies may exist undetected. In deep learning, neural networks can learn complex nonlinear relation-ships that covariance/correlation cannot capture. Additionally, high dimensionality can make covariance matrices ill-conditioned.

5. **(Evaluating)** Evaluate the statement: "Bayes' rule is the foundation of all modern machine learning." Provide arguments supporting and refuting this claim, considering both Bayesian and frequentist approaches in deep learning.

   Answer: Supporting: Bayes' rule provides principled framework for updating be-liefs with evidence, fundamental to Bayesian methods, uncertainty quantification, and probabilistic deep learning. It underpins techniques like Bayesian neural networks, variational inference, and Gaussian processes.

   Refuting: Many successful deep learning approaches are frequentist, optimizing point estimates without explicit Bayesian updating. Stochastic gradient descent and back-propagation don't inherently use Bayes' rule. Empirical risk minimization, the foun-dation of many models, is frequentist. While Bayesian methods provide uncertainty estimates, most production deep learning systems use frequentist approaches for com-putational efficiency.

Mathematical Problems

1. **Problem:** Compute the SVD of $\mathbf{A} = \begin{matrix} 2 & 2 \\ -1 & 1 \end{matrix}$

   Solution:

$$\mathbf{A}^\top\mathbf{A} = \begin{matrix} 5 & 3 \\ 3 & 5 \end{matrix}$$

$$\text{Eigenvalues: } \lambda_1 = 8, \lambda_2 = 2$$

$$\text{Singular values: } \sigma_1 = 2\sqrt{2}, \sigma_2 = \sqrt{2}$$

$$\mathbf{V} = \begin{matrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{matrix}$$

$$\mathbf{U} = \begin{matrix} 1 & 0 \\ 0 & -1 \end{matrix}$$

2. **Problem:** For a Gaussian distribution with $\mu = 2$, $\sigma = 3$, find $P(2 \le X \le 5)$

   Solution:

$$Z_1 = \frac{2-2}{3} = 0 \Rightarrow \Phi(0) = 0.5$$

$$Z_2 = \frac{5-2}{3} = 1 \Rightarrow \Phi(1) = 0.8413$$

$$P(2 \leq X \leq 5) = 0.8413 - 0.5 = 0.3413$$

1.12 Probability & Information Theory

1.12.1 Random Variables

A random variable X is a function that maps outcomes of random phenomena to numerical values.

Discrete Random Variable: Takes countable values (e.g., dice rolls)

$$P(X = x_i) = p_i, \quad \sum_i p_i = 1$$

Continuous Random Variable: Takes uncountable values (e.g., temperature)

$$P(a \le X \le b) = \int_a^b f(x)dx, \quad \int_{-\infty}^{\infty} f(x)dx = 1$$

1.12.2 Probability

Distributions Common

Discrete Distributions

- **Bernoulli:** $P(X = k) = p^k(1-p)^{1-k}$ for $k \in \{0, 1\}$
- **Binomial:** $P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$
- **Multinomial:** Generalization of binomial to multiple outcomes

Common Continuous Distributions

- **Gaussian/Normal:** $f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$
- **Uniform:** $f(x) = \frac{1}{b-a}$ for $x \in [a, b]$

1.12.3 Marginal and Conditional Probability

Marginal Probability:

$$P(X = x) = \sum_y P(X = x, Y = y)$$

Conditional Probability:

$$P(Y = y|X = x) = \text{————————}$$

$P(X = x, Y = y)$ $P(X = x)$

Expectation, Variance, Covariance

Expectation:

$$\mathrm{E}[X] = \sum_{x} xP(X = x) \quad \text{(discrete)}$$

$$\mathrm{E}[X] = \int^{x} xf(x)dx \quad \text{(continuous)}$$

Variance:

$$\mathrm{Var}(X) = \mathrm{E}[(X - \mathrm{E}[X])^2] = \mathrm{E}[X^2] - (\mathrm{E}[X])^2$$

Covarianc

$$\mathrm{Cov}(X, Y) = \mathrm{E}[(X - \mathrm{E}[X])(Y - \mathrm{E}[Y])]$$

e:

1.12.4 Bayes' Rule

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} = \frac{P(B|A)P(A)}{\sum_i P(B|A_i)P(A_i)}$$

Worked Example: Medical test with 99% accuracy, disease prevalence 1%

$$P(\text{disease} \mid \text{positive}) = \frac{0.99 \times 0.01}{0.99 \times 0.01 + 0.01 \times 0.99} = 0.5$$

1.12.5 Information Theory Concepts

Entropy: Measure of uncertainty

$$H(X) = -\sum_{x \in \mathrm{X}} P(x) \log P(x)$$

Cross-Entropy: Measure of difference between two distributions

$$H(P, Q) = -\sum_{x \in \mathrm{X}} P(x) \log Q(x)$$

Kullback-Leibler Divergence: Measure of how one distribution diverges from another

$$D_{KL}(P \| Q) = \sum_{x \in \mathrm{X}} P(x) \log \frac{P(x)}{Q(x)}$$

Exercises

Multiple Choice Questions

1. In SVD decomposition $\mathbf{A} = \mathbf{U\Sigma V}^\top$, the columns of $\mathbf{U}$ are eigenvectors of:

   (a) $\mathbf{A}^\top\mathbf{A}$
   (b) $\mathbf{AA}^\top$
   (c) $\mathbf{A}$
   (d) $\mathbf{\Sigma}$

   Answer: (b) $\mathbf{AA}^\top$

2. The primary goal of PCA is to:

   (a) Increase dimensionality of data
   (b) Find correlated features
   (c) Maximize variance along principal components
   (d) Minimize computational complexity

   Answer: (c) Maximize variance along principal components

3. For two independent random variables X and Y, which statement is true?

   (a) $P(X|Y) = P(X)$
   (b) $P(X, Y) = P(X) + P(Y)$
   (c) $\text{Cov}(X, Y) = 1$
   (d) $\text{E}[XY] = 0$

   Answer: (a) $P(X|Y) = P(X)$ for independent variables

4. In Bayes' rule, $P(A)$ is called the:

   (a) Likelihood
   (b) Evidence
   (c) Posterior
   (d) Prior

   Answer: (d) Prior probability

5. The entropy of a fair coin flip is:

   (a) 0 bits
   (b) 1 bit
   (c) 0.5 bits
   (d) 2 bits

   Answer: (b) $H(X) = -0.5\log_2(0.5) - 0.5\log_2(0.5) = 1$ bit

Long Answer/Essay Questions

1. **(Remembering)** Define SVD and list its key components. Explain the relationship between singular values and eigenvalues.

   Answer: SVD decomposes matrix $\mathbf{A}$ as $\mathbf{A} = \mathbf{U\Sigma V}^\top$ where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices containing left and right singular vectors, and $\mathbf{\Sigma}$ is diagonal with singular values. Singular values are square roots of eigenvalues of $\mathbf{A}^\top\mathbf{A}$.

2. **(Understanding)** Explain the intuition behind PCA using geometric interpretation. Why do we center the data before applying PCA?

   Answer: PCA finds orthogonal directions of maximum variance. Geometrically, it rotates the coordinate system to align with directions where data is most spread out. We center data to ensure the first principal component captures direction of maxi-mum variance rather than the mean direction. Without centering, PCA might find components that mainly reflect the data mean.

3. **(Applying)** Given the dataset: (1,2), (2,4), (3,6), (4,8), perform PCA and find the principal components. What percentage of variance is explained by the first compo-nent?

   Answer:

   (a) Center data: Mean = (2.5,5), Centered: (-1.5,-3), (-0.5,-1), (0.5,1), (1.5,3)

   (b) Covariance matrix: $\mathbf{C} = \begin{matrix} 1.67 & 3.33 \\ 3.33 & 6.67 \end{matrix}$

   (c) Eigenvalues: $\lambda_1 = 8.33, \lambda_2 = 0$

   (d) Eigenvectors: $\mathbf{v}_1 = [0.447, 0.894]^\top$, $\mathbf{v}_2 = [-0.894, 0.447]^\top$

   (e) Variance explained by first component: 100% (since $\lambda_2 = 0$)

4. **(Analyzing)** Compare and contrast covariance and correlation. Discuss their limita-tions in measuring relationships between variables, especially in the context of deep learning.

   Answer: Covariance measures direction of linear relationship but is scale-dependent. Correlation normalizes covariance to [1, 1] range, making it scale-invariant. Lim-itations: Both only capture linear relationships; nonlinear dependencies may exist undetected. In deep learning, neural networks can learn complex nonlinear relation-ships that covariance/correlation cannot capture. Additionally, high dimensionality can make covariance matrices ill-conditioned.

5. **(Evaluating)** Evaluate the statement: "Bayes' rule is the foundation of all modern machine learning." Provide arguments supporting and refuting this claim, considering both Bayesian and frequentist approaches in deep learning.

Answer: Supporting: Bayes' rule provides principled framework for updating be-liefs with evidence, fundamental to Bayesian methods, uncertainty quantification, and probabilistic deep learning. It underpins techniques like Bayesian neural networks, variational inference, and Gaussian processes.

Refuting: Many successful deep learning approaches are frequentist, optimizing point estimates without explicit Bayesian updating. Stochastic gradient descent and back-propagation don't inherently use Bayes' rule. Empirical risk minimization, the foun-dation of many models, is frequentist. While Bayesian methods provide uncertainty estimates, most production deep learning systems use frequentist approaches for com-putational efficiency.

Mathematical Problems

1. **Problem:** Compute the SVD of $\mathbf{A} = \begin{matrix} 2 & 2 \\ -1 & 1 \end{matrix}$

Solution:

$$\mathbf{A}^\top\mathbf{A} = \begin{matrix} 5 & 3 \\ 3 & 5 \end{matrix}$$

Eigenvalues: $\lambda_1 = 8, \lambda_2 = 2$

Singular values: $\sigma_1 = 2\sqrt{2}, \sigma_2 = \sqrt{2}$

$$\mathbf{V} = \begin{matrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{matrix}$$

$$\mathbf{U} = \begin{matrix} 1 & 0 \\ 0 & -1 \end{matrix}$$

2. **Problem:** For a Gaussian distribution with $\mu = 2$, $\sigma = 3$, find $P(2 \le X \le 5)$

Solution:

$$Z = \frac{2-2}{3} = 0 \quad \Rightarrow \quad \Phi(0) = 0.5$$

$$Z = \frac{5-2}{3} = 1 \quad \Rightarrow \quad \Phi(1) = 0.8413$$

$$P(2 \le X \le 5) = 0.8413 - 0.5 = 0.3413$$

$$\|\mathbf{x}\|_\infty = \max_{1 \le i \le n} |x_i|$$

UNIT-II

SYLLABUS

UNIT-II: MACHINE LEARNING AND DEEP FEED FORWARD NETWORKS (9)

Machine Learning: Basics and Under fitting, Hyper parameters and Validation Sets, Estimators, Bias and Variance, Maximum Likelihood, Bayesian Statistics, Supervised and Unsupervised Learning, Stochastic Gradient Descent, Challenges Motivating Deep Learning.

Deep Feed forward Networks: Learning XOR, Gradient-Based Learning, Hidden Units, Architecture Design, Back-Propagation and other Differentiation Algorithms.

Chapter 2

Machine Learning

2.1 Machine Learning Fundamentals

2.1.1 Model Fitting: Underfitting, Overfitting, and Optimal Fit-ting

Definition 2.1.1 (Model Fitting)**.** *Model fitting is the process of training a model on data to approximate the underlying function that generated the data. The quality of fit is measured by how well the model performs on both training and unseen data.*

Mathematical Formulation: Given data $\{(x_i, y_i)\}^n$ generated by $y = f(x) + \epsilon$, we seek $\hat{f}$ that minimizes:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{f}(x_i))^2$$

Underfitting

Definition 2.1.2 (Underfitting)**.** *Underfitting occurs when a model is too simple to capture the underlying patterns in the data. This results in high bias and poor performance on both training and test data.*

Mathematical Characteristics:

- High bias: $\text{E}[\hat{f}(x)] - f(x)$ is large
- Low variance: $\text{Var}(\hat{f}(x))$ is small
- High training error
- High test error

Example 2.1.1 (Underfitting Example)**.** *Consider fitting a linear model to quadratic data: True function: $y = 2x^2 - 3x + 1 + \epsilon$, where $\epsilon \sim N(0, 0.1)$*
We attempt to fit: $\hat{y} = w_0 + w_1 x$

Solution: *Let* $x \in [-1, 1]$ *and generate 100 samples. The optimal linear fit minimizes:*

$$\min_{w_0, w_1} \frac{1}{10} \sum_{i=1}^{100} (2x_i^2 - 3x_i + 1 + \epsilon_i - (w_0 + w_1 x_i))^2$$

Taking expectations:

$$\begin{aligned} \mathrm{E}[y] &= 2\mathrm{E}[x^2] - 3\mathrm{E}[x] + 1 \\ &= 2 \cdot \frac{1}{3} - 3 \cdot 0 + 1 = \frac{5}{3} \\ Cov(x, y) &= \mathrm{E}[xy] - \mathrm{E}[x]\mathrm{E}[y] \\ &= \mathrm{E}[x(2x^2 - 3x + 1)] - 0 \\ &= 2\mathrm{E}[x^3] - 3\mathrm{E}[x^2] + \mathrm{E}[x] \\ &= 0 - 3 \cdot \frac{1}{3} + 0 = -1 \end{aligned}$$

Thus, the optimal linear fit is:

$$w_1 = \frac{Cov(x, y)}{Var(x)} = \frac{-1}{1/3} = -3, \quad w_0 = \mathrm{E}[y] = \frac{5}{3}$$

The model $\hat{y} = \frac{5}{3} - 3x$ *cannot capture the quadratic nature of the true function, leading to underfitting.*

Overfitting

Definition 2.1.3 (Overfitting)**.** *Overfitting occurs when a model is too complex and captures noise in the training data. This results in low bias but high variance, excellent performance on training data but poor generalization to test data.*

Mathematical Characteristics:

- Low bias: $\mathrm{E}[\hat{f}(x)] \approx f(x)$
- High variance: $\mathrm{Var}(\hat{f}(x))$ is large
- Very low training error
- High test error

Example 2.1.2 (Overfitting Example)**.** *Using the same quadratic data, we fit a 10th-degree polynomial:*

Model: $\hat{y} = \sum_{k=0}^{10} w_k x^k$

With 11 parameters for 100 data points, the model can fit the training data perfectly, including the noise ϵ_i*. The training error approaches 0, but on test*

$$\textit{data: Test MSE} = \textit{Bias}^2 + \textit{Variance} + \sigma^2 \approx 0 +$$

$$\textit{Large} + 0.01$$

The variance term is large because small changes in training data lead to large changes in the fitted polynomial.

Optimal Fitting

Definition 2.1.4 (Optimal Fitting)**.** *Optimal fitting achieves the right balance between bias and variance, minimizing the generalization error. The model complexity matches the true complexity of the data.*

Mathematical Characteristics:

- Moderate bias
- Moderate variance
- Training error slightly higher than for overfitted model
- Test error minimized

Example 2.1.3 (Optimal Fitting Example)**.** *For the quadratic data, the optimal model is a 2nd-degree polynomial:*

Model: $\hat{y} = w_0 + w_1 x + w_2 x^2$

This matches the true model structure. The solution gives:

$$w_2 = 2, \quad w_1 = -3, \quad w_0 = 1$$

which exactly recovers the true parameters (up to noise).

2.2 Comparison among Under, Optimal & Over fitting

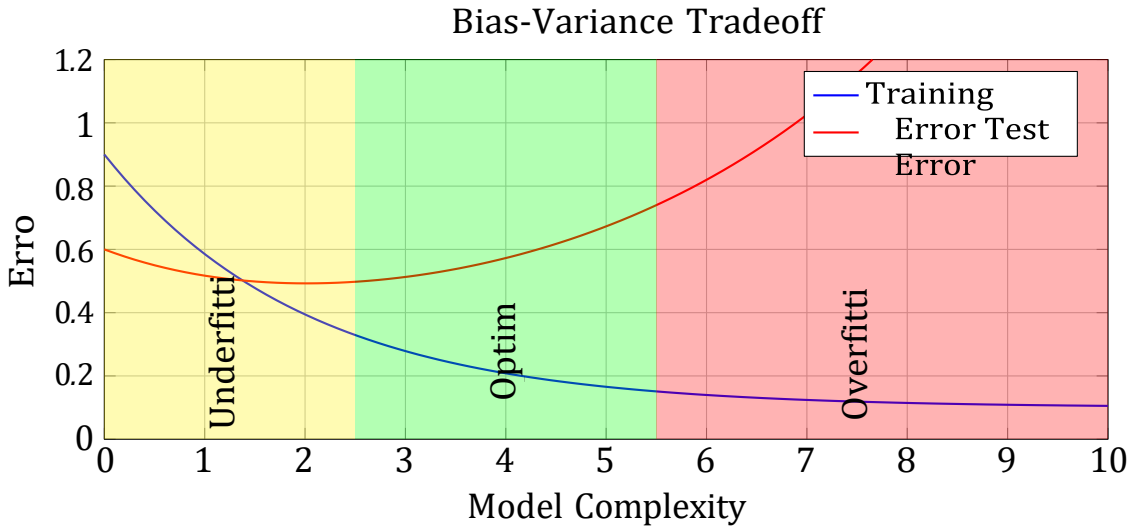


Figure 2.1: Bias-Variance Tradeoff showing underfitting, optimal fitting, and overfitting regions

2.3 Model Evaluation and Error Analysis

2.3.1 Bias-Variance Decomposition

The bias-variance decomposition is a fundamental theoretical framework that explains the sources of error in predictive models. It provides insight into why models fail to generalize and guides the design of better learning algorithms.

Theorem Statement and Proof

Theorem 2.3.1 (Bias-Variance Decomposition)**.** *For mean squared error (MSE), the expected prediction error decomposes as:*

$$\mathrm{E}[(y - \hat{f}(x))^2] = \underbrace{(\mathrm{E}[\hat{f}(x)] - f(x))^2}_{Bias^2} + \underbrace{\mathrm{E}[(\hat{f}(x) - \mathrm{E}[\hat{f}(x)])^2]}_{Variance} + \underbrace{\sigma^2}_{Irreducible\ Error}$$

Proof. Let $y = f(x) + \epsilon$, where $\mathrm{E}[\epsilon] = 0$, $\mathrm{Var}(\epsilon) = \sigma^2$, and ϵ is independent of x and $\hat{f}(x)$. Then:

$$\begin{aligned}
\mathrm{E}[(y - \hat{f})^2] &= \mathrm{E}[(f + \epsilon - \hat{f})^2] \\
&= \mathrm{E}[(f - \hat{f})^2] + \mathrm{E}[\epsilon^2] + 2\mathrm{E}[\epsilon(f - \hat{f})] \\
&= \mathrm{E}[(f - \hat{f})^2] + \sigma^2 + 2 \cdot 0 \\
&= \mathrm{E}[(f - \mathrm{E}[\hat{f}] + \mathrm{E}[\hat{f}] - \hat{f})^2] + \sigma^2 \\
&= (f - \mathrm{E}[\hat{f}])^2 + \mathrm{E}[(\mathrm{E}[\hat{f}] - \hat{f})^2] + \mathrm{E}[2(f - \mathrm{E}[\hat{f}])(\mathrm{E}[\hat{f}] - \hat{f})] + \sigma^2 \\
&= (f - \mathrm{E}[\hat{f}])^2 + \mathrm{E}[(\hat{f} - \mathrm{E}[\hat{f}])^2] + \sigma^2 \\
&= \mathrm{Bias}^2(\hat{f}) + \mathrm{Var}(\hat{f}) + \sigma^2
\end{aligned}$$

The cross term vanishes because $\mathrm{E}[\mathrm{E}[\hat{f}] - \hat{f}] = 0$. □

Components of the Decomposition

The total prediction error (MSE) decomposes into three fundamental components:

1. **Bias² (Systematic Error)**:

$$\mathrm{Bias}^2 = (\mathrm{E}[\hat{f}(x)] - f(x))^2$$

   - Measures the difference between the expected prediction and the true value
   - Represents systematic error in the model
   - **High bias**: Model oversimplifies the problem (underfitting)
   - Example: Using linear regression for quadratic data

2. **Variance (Model Sensitivity)**:

$$\mathrm{Variance} = \mathrm{E}[(\hat{f}(x) - \mathrm{E}[\hat{f}(x)])^2]$$

- Measures how much predictions vary across different training sets
- Represents model sensitivity to training data fluctuations
- **High variance**: Model is too complex (overfitting)
- Example: High-degree polynomial fitting noise in data

3. **Irreducible Error (σ^2)**:

$$\sigma^2 = \text{Var}(\epsilon)$$

- Noise inherent in the data generation process
- Cannot be reduced by any model
- Arises from measurement errors and hidden variables
- Represents the theoretical lower bound on prediction error

Visual Representation of the Trade-off

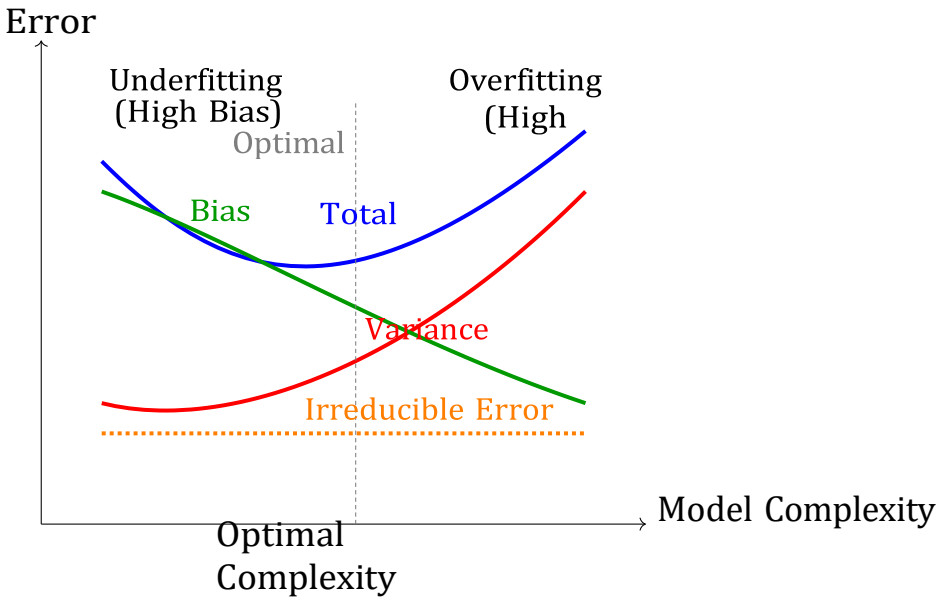


Figure 2.2: Bias-Variance Trade-off: As model complexity increases, bias decreases but variance increases. The optimal model complexity minimizes the total error.

Mathematical Example and Calculation

Consider a concrete example to illustrate the decomposition:

Example 2.3.1. *Let the true relationship be:*

$$y = x^2 + \epsilon, \quad \epsilon \sim \text{N}(0, 1), \quad \sigma^2 = 1$$

We compare two models at $x = 2$ where $f(2) = 4$:

1. ***Model A (Underfitting - Linear)***: $\hat{f}_A(x) = 2x - 1$
   - *Expected prediction:* $\mathrm{E}[\hat{f}_A(2)] = 2 \times 2 - 1 = 3$
   - *Bias²:* $(3 - 4)^2 = 1$
   - *Variance (estimated):* $\mathrm{E}[(\hat{f}_A - 3)^2] \approx 0.2$
   - *Total error:* $1 + 0.2 + 1 = 2.2$
2. ***Model B (Overfitting - 10th degree polynomial)***: $\hat{f}_B(x) = \sum_{i=}^{10} a_i x^i$
   - *Expected prediction:* $\mathrm{E}[\hat{f}_B(2)] = 4$ *(unbiased on average)*
   - *Bias²:* $(4 - 4)^2 = 0$
   - *Variance (estimated):* $\mathrm{E}[(\hat{f}_B - 4)^2] \approx 3.0$
   - *Total error:* $0 + 3.0 + 1 = 4.0$

Despite having zero bias, Model B performs worse due to high variance!

Practical Implications and Applications

- **Regularization**: Techniques like L1/L2 regularization explicitly trade bias for vari-ance

$$L_{reg} = L_{MSE} + \lambda \|\mathbf{w}\|_2^2 \quad \text{(L2)}$$
$$L_{reg} = L_{MSE} + \lambda \|\mathbf{w}\|_1 \quad \text{(L1)}$$

- **Cross-Validation**: Estimates the bias-variance trade-off empirically
  - Low bias, high variance: Large gap between training and validation error
  - High bias, low variance: Similar training and validation errors (both high)
- **Ensemble Methods**:
  - **Bagging** (Bootstrap Aggregating): Reduces variance

$$\hat{f}_{\text{bag}}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^{*b}(x)$$

  - **Boosting**: Reduces bias by sequentially correcting errors

$$\hat{f}_{\text{boost}}(x) = \sum_{m=1}^{M} \alpha_m h_m(x)$$

- **Model Selection Criteria**:
  - AIC (Akaike Information Criterion): Balances fit and complexity

$$\text{AIC} = 2k - 2\ln(\hat{L})$$

  - BIC (Bayesian Information Criterion): Stronger penalty for complexity

$$\text{BIC} = k\ln(n) - 2\ln(\hat{L})$$

Extensions and Variations

- **For 0-1 Loss (Classification)**:

$$\text{Error} = \mathrm{E}[\mathrm{I}(y \neq \hat{y})] \leq \text{Bias} + \sqrt{\text{Variance}}$$

- **Conditional Decomposition**:

$$\mathrm{E}[(y - \hat{f})^2 | X = x] = \text{Bias}^2(\hat{f}(x)) + \text{Var}(\hat{f}(x)) + \sigma^2$$

- **Expected Test Error**:

$$\mathrm{E}[\text{MSE}_{\text{test}}] = \text{Bias}^2 + \text{Variance} + \sigma^2 + \frac{\sigma^2}{n_{\text{train}}}$$

Key Insights and Recommendations

Diagnosis	Symptoms	Remedies
High Bias (Underfitting)	· High training error · High validation error · Simple model structure	· Increase model complexity · Add more features · Reduce regularization · Use non-linear models
High Variance (Overfitting)	· Low training error · High validation error · Complex model structure	· Decrease model complex-ity · Add regularization · Get more training data · Use dropout (neural nets)
Well-Balanced Model	· Acceptable training error · Similar validation error · Generalizes well	· Maintain current complex-ity · Monitor performance · Consider ensemble meth-ods

Table 2.3: Practical guidance based on bias-variance analysis

Mathematical Properties

- **Additivity**: The decomposition is additive for MSE loss

- **Model Dependence**: Both bias and variance depend on:
  - Model complexity (degrees of freedom)
  - Training sample size
  - Signal-to-noise ratio in data
- **Asymptotic Behavior**:
  - As $n \to \infty$: Variance $\to 0$ for consistent estimators
  - Bias remains unless model is correctly specified
  - Irreducible error is constant

This bias-variance framework provides the theoretical foundation for understanding model performance and guides practical decisions in machine learning model development, regularization, and ensemble methods.

2.4 Maximum Likelihood Estimation: Theory, Deriva-tion, and Applications

Definition 2.4.1 (Maximum Likelihood Estimation)**.** *Maximum Likelihood Estimation (MLE) is a fundamental statistical method for estimating the parameters* $\boldsymbol{\theta} = (\theta_1, \theta_2, \ldots, \theta_k)^\top$ *of a probabilistic model* $p(x|\boldsymbol{\theta})$ *given independent and identically distributed (i.i.d.) observed data* $\mathrm{D} = \{x_1, x_2, \ldots, x_n\}$*. The MLE principle states that we should choose the parameter values that maximize the likelihood function, which measures how probable the observed data is under the model:*

$$\hat{\boldsymbol{\theta}}_{MLE} = \arg\max_{\boldsymbol{\theta}} L(\boldsymbol{\theta}; \mathrm{D}) = \arg\max_{\boldsymbol{\theta}} \prod_{i=}^{n} p(x_i|\boldsymbol{\theta})$$

where $L(\boldsymbol{\theta};\mathrm{D})$ *is the* ***likelihood function****, treating the observed data as fixed and varying the parameters.*

2.4.1 Mathematical Foundation and Properties

Statistical Model and Assumptions:

- **i.i.d. Assumption**: Observations $x_1, \ldots, x_n$ are independent and identically dis-tributed.
- **Model Specification**: The parametric family $p(x|\boldsymbol{\theta})$ is correctly specified (or approx-imately).
- **Identifiability**: Different parameters yield different distributions: $\boldsymbol{\theta}_1 \neq \boldsymbol{\theta}_2 \implies p(x|\boldsymbol{\theta}_1) \neq p(x|\boldsymbol{\theta}_2)$.

Key Properties of MLE:

1. **Consistency**: As sample size $n \to \infty$, $\hat{\boldsymbol{\theta}}_{\text{MLE}} \xrightarrow{p} \boldsymbol{\theta}_0$ (the true parameter vector). Formally:
$$\forall \epsilon > 0, \quad \lim_{n\to\infty} P(\|\hat{\boldsymbol{\theta}}_{\text{MLE}} - \boldsymbol{\theta}_0\| > \epsilon) = 0$$
2. **Asymptotic Normality**: Under regularity conditions,
$$\sqrt{n}(\hat{\boldsymbol{\theta}}_{\text{MLE}} - \boldsymbol{\theta}_0) \xrightarrow{d} \mathrm{N}\left(0, \mathrm{I}^{-1}(\boldsymbol{\theta}_0)\right)$$
where $\mathrm{I}(\boldsymbol{\theta})$ is the **Fisher Information Matrix**:
$$\mathrm{I}(\boldsymbol{\theta}) = \mathrm{E}_{x\sim p(x|\boldsymbol{\theta})}\left[\nabla_{\boldsymbol{\theta}} \log p(x|\boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} \log p(x|\boldsymbol{\theta})^{\top}\right]$$
Equivalently, $\mathrm{I}(\boldsymbol{\theta}) = -\mathrm{E}_{\boldsymbol{\theta}}[\nabla^2 \log p(x|\boldsymbol{\theta})]$.
3. **Invariance Principle**: If $\hat{\boldsymbol{\theta}}$ is the MLE of $\boldsymbol{\theta}$, then for any bijective function g, $g(\hat{\boldsymbol{\theta}})$ is the MLE of $g(\boldsymbol{\theta})$. More generally, for any function $\tau = g(\boldsymbol{\theta})$, the MLE of τ is $g(\hat{\boldsymbol{\theta}})$.
4. **Asymptotic Efficiency**: The MLE achieves the Cramér-Rao lower bound asymptot-ically:
$$\mathrm{Cov}(\hat{\boldsymbol{\theta}}_{\text{MLE}}) \approx \frac{1}{n}\mathrm{I}^{-1}(\boldsymbol{\theta}_0)$$
meaning no unbiased estimator has lower asymptotic mean squared error.
5. **Functional Equivariance**: If we reparameterize the model with $\phi = h(\boldsymbol{\theta})$ where h is differentiable and invertible, then:
$$\hat{\phi}_{\text{MLE}} = h(\hat{\boldsymbol{\theta}}_{\text{MLE}})$$

2.4.2 Log-Likelihood Formulation and Score Equations

In practice, we work with the **log-likelihood** $\ell(\boldsymbol{\theta}; \mathrm{D}) = \log L(\boldsymbol{\theta}; \mathrm{D})$ for numerical stability and analytical convenience:

$$\ell(\boldsymbol{\theta}; \mathrm{D}) = \sum_{i=1} \log p(x_i|\boldsymbol{\theta})$$

Since the logarithm is a monotonically increasing function, maximizing $L(\boldsymbol{\theta})$ is equivalent to maximizing $\ell(\boldsymbol{\theta})$.

The **score function** $s(\boldsymbol{\theta})$ is the gradient of the log-likelihood:

$$s(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta}; \mathrm{D}) = \sum_{i=1} \nabla_{\boldsymbol{\theta}} \log p(x_i|\boldsymbol{\theta})$$

The MLE is found by solving the **score equations**:

$$s(\boldsymbol{\theta}) = \mathbf{0}$$

Second-Order Conditions: The Hessian matrix $\boldsymbol{H}(\boldsymbol{\theta}) = \nabla^2_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta}; \mathrm{D})$ should be negative definite at the solution to ensure a local maximum.

2.4.3 Detailed Example: MLE for Gaussian Distribution

Example 2.4.1 (MLE for Univariate Normal Distribution). *Consider data* $\mathcal{D} = \{x_1, x_2, \ldots, x_n\}$ *assumed to be i.i.d. from* $\mathcal{N}(\mu, \sigma^2)$. *The parameters are* $\boldsymbol{\theta} = (\mu, \sigma^2)$.

Step 1: Write the likelihood function.

$$p(x_i|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(x_i - \mu)^2}{2\sigma^2}\right]$$

The joint likelihood is:

$$L(\mu, \sigma^2; \mathcal{D}) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(x_i - \mu)^2}{2\sigma^2}\right]$$

Step 2: Form the log-likelihood.

$$\begin{aligned}
\ell(\mu, \sigma^2; \mathcal{D}) &= \sum_{i=1}^{n} \log\left[\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(x_i - \mu)^2}{2\sigma^2}\right]\right] \\
&= \sum_{i=1}^{n} -\frac{1}{2}\log(2\pi) - \frac{1}{2}\log(\sigma^2) - \frac{(x_i - \mu)^2}{2\sigma^2} \\
&= -\frac{n}{2}\log(2\pi) - \frac{n}{2}\log(\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i - \mu)^2
\end{aligned}$$

Step 3: Derive score equations (first derivatives).

For μ*:*

$$\begin{aligned}
\frac{\partial \ell}{\partial \mu} &= -\frac{1}{2\sigma^2}\sum_{i=1}^{n} 2(x_i - \mu)(-1) \\
&= \frac{1}{\sigma^2}\sum_{i=1}^{n}(x_i - \mu) \\
&= \frac{1}{\sigma^2}\left[\sum_{i=1}^{n} x_i - n\mu\right]
\end{aligned}$$

For σ^2*:*

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

Step 4: Solve score equations.

Setting $\frac{\partial \ell}{\partial \mu} = 0$*:*

$$\frac{1}{\sigma^2}\sum_{i=1}^{n} x_i - n\mu \overset{!}{=} 0 \quad \Longrightarrow \quad \sum_{i=1}^{n} x_i - n\mu = 0 \quad \Longrightarrow \quad \hat{\mu}_{MLE} = \frac{1}{n}\sum_{i=1}^{n} x_i = \bar{x}$$

Setting $\frac{\partial \ell}{\partial \sigma^2} = 0$*:*

$$-\frac{n}{2\sigma^2} + \frac{1}{2(\sigma^2)^2}\sum_{i=1}^{n}(x_i - \mu)^2 = 0$$

Multiplying by $2(\sigma^2)^2$*:*

$$-n\sigma^2 + \sum_{i=1}^{n}(x_i - \mu)^2 = 0 \implies \hat{\sigma}^2_{MLE} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{\mu}_{MLE})^2$$

Step 5: Verify second-order conditions.

The Hessian matrix is:

$$\boldsymbol{H}(\mu, \sigma^2) = \begin{bmatrix} \frac{\partial^2 \ell}{\partial \mu^2} & \frac{\partial^2 \ell}{\partial \mu \partial \sigma^2} \\ \frac{\partial^2 \ell}{\partial \sigma^2 \partial \mu} & \frac{\partial^2 \ell}{\partial (\sigma^2)^2} \end{bmatrix}$$

where:

$$\frac{\partial^2 \ell}{\partial \mu^2} = -\frac{n}{\sigma^2}, \quad \frac{\partial^2 \ell}{\partial \mu \partial \sigma^2} = -\frac{1}{(\sigma^2)^2}\sum_{i=1}^{n}(x_i - \mu), \quad \frac{\partial^2 \ell}{\partial (\sigma^2)^2} = \frac{n}{2(\sigma^2)^2} - \frac{1}{(\sigma^2)^3}\sum_{i=1}^{n}(x_i - \mu)^2$$

At the MLE:

$$\left.\frac{\partial^2 \ell}{\partial \mu^2}\right|_{\hat{\mu}, \hat{\sigma}^2} = -\frac{n}{\hat{\sigma}^2} < 0$$

$$\left.\frac{\partial^2 \ell}{\partial \mu \partial \sigma^2}\right|_{\hat{\mu}, \hat{\sigma}^2} = \frac{1}{(\hat{\sigma}^2)^2}\sum_{i=1}^{n}(x_i - \hat{\mu}) = 0$$

$$\left.\frac{\partial^2 \ell}{\partial (\sigma^2)^2}\right|_{\hat{\mu}, \hat{\sigma}^2} = \frac{n}{2(\hat{\sigma}^2)^2} - \frac{1}{(\hat{\sigma}^2)^3}\cdot n\hat{\sigma}^2 = -\frac{n}{2(\hat{\sigma}^2)^2} <$$

The Hessian is diagonal with negative diagonal entries, confirming a maximum.

Step 6: Interpretation and Analysis.

The MLE results are:

$$\hat{\mu}_{MLE} = \bar{x} = \text{sample mean}$$

$$\hat{\sigma}^2_{MLE} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2 = \text{sample variance (biased version)}$$

Remark 2.4.1 (Bias of Variance Estimator)**.** *Note that* $\hat{\sigma}^2_{MLE}$ *is biased:*

$$\mathrm{E}[\hat{\sigma}^2_{MLE}] = \frac{n-1}{n}\sigma^2 = \sigma^2$$

The unbiased estimator is: $\frac{n}{\sum}$

$$s^2 = \frac{1}{n-1} \sum_{i=1} (x_i - \bar{x})^2$$

However, $\hat{\sigma}^2_{MLE}$ is asymptotically unbiased and minimizes MSE among estimators of the form $c \sum_{i=1}^{n} (x_i -$ $\bar{x})^2$

Fisher Information and Asymptotic Variance:
For a single observation $x \sim \mathrm{N}(\mu, \sigma^2)$*:*

$$\log p(x|\mu, \sigma^2) = -\frac{1}{2}\log(2\pi\sigma^2) - \frac{(x-\mu)^2}{2\sigma^2}$$

The Fisher Information matrix is:

$$\mathrm{I}(\mu, \sigma_2) = \begin{matrix} \frac{1}{\sigma_2} & 0 \\ 0 & \frac{1}{2\sigma^4} \end{matrix}$$

Thus, the asymptotic variance-covariance matrix is:

$$Cov(\hat{\mu}_{MLE}, \hat{\sigma}^2_{MLE}) \approx \frac{1}{n}\mathrm{I}^{-1} = \begin{matrix} \frac{\sigma^2}{n} & 0 \\ 0 & \frac{2\sigma^4}{n} \end{matrix}$$

This shows that $\hat{\mu}_{MLE}$ *and* $\hat{\sigma}^2_{MLE}$ *are asymptotically uncorrelated, with standard errors:*

$$SE(\hat{\mu}_{MLE}) \approx \frac{\sigma}{\sqrt{n}}, \quad SE(\hat{\sigma}^2_{MLE}) \approx \sqrt{\frac{2}{n}}$$

2.4.4 General MLE Procedure and Algorithm

General Steps for MLE:

1. **Model Specification**: Choose parametric family $p(x|\boldsymbol{\theta})$.
2. **Likelihood Construction**: $L(\boldsymbol{\theta}) = \prod_{i=}^{n} p(x_i|\boldsymbol{\theta})$.
3. **Log-Likelihood**: $\ell(\boldsymbol{\theta}) = \sum_{i=}^{n} \log p(x_i|\boldsymbol{\theta})$.
4. **Score Equations**: Set $\nabla_{\boldsymbol{\theta}}\ell(\boldsymbol{\theta}) = 0$.
5. **Solve**: Find $\hat{\boldsymbol{\theta}}$ satisfying score equations.
6. **Verification**: Check $\nabla^2_{\boldsymbol{\theta}}\ell(\hat{\boldsymbol{\theta}})$ is negative definite.
7. **Inference**: Compute standard errors via Fisher information.

Numerical Methods: When analytical solutions are unavailable, use:

- Gradient-based: Newton-Raphson, Fisher scoring, gradient descent
- EM Algorithm: For models with latent variables
- Monte Carlo methods: MCMC, simulated annealing

2.4.5 Remarks and Practical Considerations

Remark 2.4.2 (Regularity Conditions)**.** *For MLE properties to hold, the following regularity conditions are required:*

1. *The parameter space* Θ *is compact.*
2. *The true parameter* $\boldsymbol{\theta}_0$ *lies in the interior of* Θ*.*
3. *The model is identifiable:* $\boldsymbol{\theta}_1 \neq \boldsymbol{\theta}_2 \implies p(x|\boldsymbol{\theta}_1) \neq p(x|\boldsymbol{\theta}_2)$*.*
4. *The log-likelihood* $\ell(\boldsymbol{\theta})$ *is three times continuously differentiable.*
5. *The Fisher information matrix* $\mathrm{I}(\boldsymbol{\theta})$ *is positive definite.*

Violation of these conditions can lead to inconsistent estimates or non-standard distributions.

Remark 2.4.3 (Boundary Problems)**.** *When the MLE lies on the boundary of the parameter space, standard asymptotic theory fails. For example, estimating variance components often yields* $\hat{\sigma}^2 = 0$*. In such cases, the distribution of the MLE is non-normal.*

Remark 2.4.4 (Multiple Local Maxima)**.** *The likelihood function may have multiple local maxima, especially in complex models (e.g., mixture models). Global optimization techniques or multiple random starts are needed.*

Remark 2.4.5 (Small Sample Properties)**.** *MLE can be biased in finite samples. Corrections like Bartlett correction or bootstrap can improve inference. For example, the bias-corrected MLE for variance is:*

$$\hat{\sigma}^2_B = \frac{n}{n-1}\hat{\sigma}^2_{MLE} = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

$n-1$

Remark 2.4.6 (Robustness)**.** *MLE is generally not robust to outliers. Robust alternatives include M-estimators or using heavy-tailed distributions (e.g., Student's t instead of normal).*

Remark 2.4.7 (Computational Considerations)**.** *For large datasets, computing the full like-lihood can be expensive. Stochastic gradient descent (SGD) or mini-batch methods can be used to approximate MLE for large-scale problems.*

2.4.6 Applications in Machine Learning

1. Connection to Empirical Risk Minimization: Many ML loss functions can be derived from MLE:

- **Linear Regression**: Assuming $y|x \sim \mathrm{N}(w^\top x, \sigma^2)$, MLE gives squared loss.
- **Logistic Regression**: Assuming $y|x \sim \text{Bernoulli}(\sigma(w^\top x))$, MLE gives cross-entropy loss.
- **Poisson Regression**: Assuming $y|x \sim \text{Poisson}(\exp(w^\top x))$, MLE gives Poisson loss.

2. Bayesian Interpretation: MLE can be viewed as a special case of MAP (Maximum A Posteriori) estimation with uniform prior:

$$\hat{\boldsymbol{\theta}}_{\text{MAP}} = \arg\max_{\boldsymbol{\theta}} p(\mathrm{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})$$

With $p(\boldsymbol{\theta}) \propto 1$, MAP reduces to MLE.

3. Model Comparison: Likelihood-based criteria for model selection:

- **AIC (Akaike Information Criterion)**: $\text{AIC} = 2k - 2\ell(\hat{\boldsymbol{\theta}})$
- **BIC (Bayesian Information Criterion)**: $\text{BIC} = k \log n - 2\ell(\hat{\boldsymbol{\theta}})$
- **Likelihood Ratio Test**: For nested models $H_0 : \boldsymbol{\theta} \in \Theta_0$ vs $H_1 : \boldsymbol{\theta} \in \Theta_1$:

$$\Lambda = 2[\ell(\hat{\boldsymbol{\theta}}_1) - \ell(\hat{\boldsymbol{\theta}}_0)] \sim \chi^2_d \quad \text{(asymptotically)}$$

where $d = \dim(\Theta_1) - \dim(\Theta_0)$.

4. Neural Networks: Training neural networks via maximum likelihood:

$$\mathrm{L}(\boldsymbol{\theta}) = -\sum_{i=1} \log p(y_i|f(x_i; \boldsymbol{\theta}))$$

where $f(x_i; \boldsymbol{\theta})$ is the neural network output. For classification with softmax:

$$p(y_i|f(x_i; \boldsymbol{\theta})) = \frac{\exp(f_{y_i}(x_i; \boldsymbol{\theta}))}{\sum_{j=1}^{C} \exp(f_j(x_i;$$

2.4.7 Mathematical Example: MLE for Exponential Distribution

Example 2.4.2 (MLE for Exponential Distribution)**.** *Consider i.i.d. data* $\mathrm{D} = \{x_1, \ldots, x_n\}$ *from Exp*(λ) *with pdf:*

$$p(x|\lambda) = \lambda e^{-\lambda x}, \quad x > 0, \lambda > 0$$

Likelihood:

$$L(\lambda) = \prod_{i=1} \lambda e^{-\lambda x_i} = \lambda^n \exp\left(-\lambda \sum_{i=1} x_i\right)$$

Log-likelihood:

$$\ell(\lambda) = n \log \lambda - \lambda \sum_{i=1} x_i$$

Score equation:

$$\frac{d\ell}{d\lambda} = \frac{n}{\lambda} - \sum_{i=1} x_i = 0 \implies \hat{\lambda}_{MLE} = \frac{n}{\sum_{i=1} x_i} = \frac{1}{\bar{x}}$$

Second derivative:

$$\frac{d^2\ell}{d\lambda^2} = -\frac{n}{\lambda^2} < 0 \quad for\ \lambda > 0$$

Fisher information:

$$\mathrm{I}(\lambda) = -\mathrm{E}\left[\frac{d^2}{d\lambda^2}\log p(x|\lambda)\right] = \frac{1}{\lambda^2}$$

Asymptotic variance:

$$\hat{Var}(\lambda_{MLE}) \approx \frac{1}{n\mathrm{I}(\lambda)} = \frac{\lambda^2}{n}$$

Interpretation: *The MLE is the reciprocal of the sample mean. This estimator is consistent and asymptotically efficient.*

Distribution	**Parameter(s)**	**MLE(s)**
Bernoulli(p)	p	$\hat{p} = \frac{1}{n}\sum_{i=1}^{n} x_i$
Poisson(λ)	λ	$\hat{\lambda} = \frac{1}{n}\sum_{i=1}^{n} x_i$
Normal(μ, σ^2)	μ, σ^2	$\hat{\mu} = \bar{x}, \quad \hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$
Exponential(λ)	λ	$\hat{\lambda} = \frac{1}{\bar{x}}$
Uniform(a, b)	a, b	$\hat{a} = \min_i x_i, \quad \hat{b} = \max_i x_i$
Gamma(α, β)	α, β	No closed form; solve $\psi(\hat{\alpha}) - \log\hat{\beta} = \frac{1}{n}\sum \log x_i$, $\hat{\alpha}/\hat{\beta} = \bar{x}$

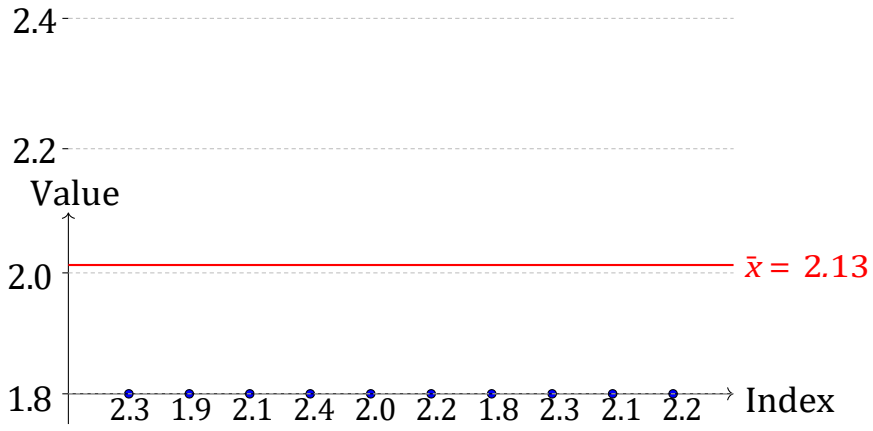
Table 2.4: MLE for Common Distributions

2.4.8 Numerical Example: MLE for Gaussian Distribution with Concrete Data

Example 2.4.3 (Numerical MLE for Normal Distribution)**.** *Consider the following dataset of 10 observations, assumed to be i.i.d. from* $\mathrm{N}(\mu, \sigma^2)$*:*

$$D = \{2.3, 1.9, 2.1, 2.4, 2.0, 2.2, 1.8, 2.3, 2.1, 2.2\}$$

Step 1: Visualize the data



Data points with sample mean

Figure 2.3: Visualization of the 10 data points

Step 2: Compute basic statistics

$$n = 10$$

$$\sum_{i=1} x_i = 2.3 + 1.9 + 2.1 + 2.4 + 2.0 + 2.2 + 1.8 + 2.3 + 2.1 + 2.2 = 21.3$$

$$\bar{x} = \frac{21.3}{10} = 2.13$$

Step 3: Compute the MLE for μ

From the theoretical derivation:

$$\hat{\mu}_{MLE} = \bar{x} = 2.13$$

Step 4: Compute the MLE for σ^2

First compute the squared deviations from the mean:

i	x_i	$x_i - \bar{x}$	$(x_i - \bar{x})^2$
1	2.3	0.17	0.0289
2	1.9	-0.23	0.0529
3	2.1	-0.03	0.0009
4	2.4	0.27	0.0729
5	2.0	-0.13	0.0169
6	2.2	0.07	0.0049
7	1.8	-0.33	0.1089
8	2.3	0.17	0.0289
9	2.1	-0.03	0.0009
10	2.2	0.07	0.0049
Total			0.3210

Table 2.5: Computation of squared deviations from mean

$$\sum_{i=1} (x_i - \bar{x})^2 = 0.3210$$

Now compute the MLE:

$$\hat{\sigma}^2_{MLE} = \frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^2 = \frac{0.3210}{10} = 0.0321$$

$$\hat{\sigma}_{MLE} = \sqrt{0.0321} = 0.1792$$

Step 5: Compare with unbiased estimator

The unbiased estimator for σ^2 is:

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2 = \frac{0.3210}{9} = 0.0357$$

$$s = \sqrt{0.0357} = 0.1889$$

Comparison:

- *MLE:* $\hat{\sigma}^2 = 0.0321$, $\hat{\sigma} = 0.1792$
- *Unbiased:* $s^2 = 0.0357$, $s = 0.1889$
- *Ratio:* $s^2/\hat{\sigma}^2 = 0.0357/0.0321 = 1.111 \approx \frac{n}{n-1} = \frac{10}{9} = 1.111$

Step 6: Compute Fisher Information and Standard Errors

Assume the true parameters are approximately equal to our estimates:

$$\mu \approx 2.13, \quad \sigma^2 \approx 0.0321, \quad \sigma \approx 0.1792$$

The Fisher Information matrix for a single observation is:

$$I(\mu, \sigma^2) = \begin{bmatrix} \frac{1}{\sigma^2} & 0 \\ 0 & \frac{1}{2\sigma^4} \end{bmatrix} = \begin{bmatrix} \frac{1}{0.0321} & 0 \\ 0 & \frac{1}{2\times(0.0321)^2} \end{bmatrix} = \begin{bmatrix} 31.152 & 0 \\ 0 & 485.51 \end{bmatrix}$$

For $n = 10$ observations, the asymptotic variance-covariance matrix is:

$$Cov(\hat{\mu}_{MLE}, \hat{\sigma}^2_{MLE}) \approx \frac{1}{n}\mathbf{I}^{-1} = \frac{1}{10}\begin{bmatrix} \frac{1}{31.15} & 0 \\ 0 & \frac{1}{485.5} \end{bmatrix}^{-1} = \frac{1}{10}\begin{bmatrix} 0.0321 & 0 \\ 0 & 0.00206 \end{bmatrix} = \begin{bmatrix} 0.00321 & 0 \\ 0 & 0.000206 \end{bmatrix}$$

Standard Errors:

$$SE(\hat{\mu}_{MLE}) \approx \sqrt{0.00321} = 0.0567$$

$$SE(\hat{\sigma}^2_{MLE}) \approx \sqrt{0.000206} = 0.0144$$

95% Confidence Intervals:

$$\hat{\mu}_{MLE} \pm 1.96 \times SE(\hat{\mu}_{MLE}) = 2.13 \pm 1.96 \times 0.0567 = 2.13 \pm 0.111$$

$$CI_{\mu} : (2.019, 2.241)$$

$$\hat{\sigma}^2_{MLE} \pm 1.96 \times SE(\hat{\sigma}^2_{MLE}) = 0.0321 \pm 1.96 \times 0.0144 = 0.0321 \pm 0.0282$$

$$CI_{\sigma^2} : (0.0039, 0.0603)$$

Step 7: Likelihood Function Visualization

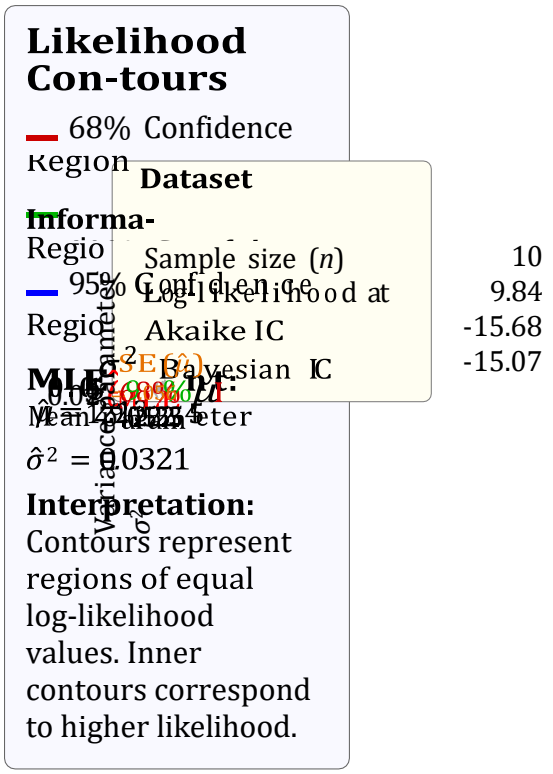


Figure 2.4: Likelihood surface contours around the Maximum Likelihood Estimate (MLE). The concentric elliptical contours represent regions of constant log-likelihood values cor-responding to different confidence levels: 68% (red), 90% (green), and 95% (blue). The MLE point $(2.13, 0.0321)$ is at the center, with standard errors $\text{SE}(\hat{\mu}) = 0.0567$ and $\text{SE}(\hat{\sigma}^2) = 0.0144$ indicated by orange arrows. The elliptical shape shows that the likeli-hood is more sensitive to changes in μ (narrower contours) than to changes in σ^2 (wider contours). This visualization illustrates parameter uncertainty and the correlation structure from the Fisher Information matrix.

Step 8: Hypothesis Testing

Test $H_0 : \mu = 2.0$ *vs* $H_1 : \mu = 2.0$*:*

Test statistic:

$$z = \frac{\hat{\mu}_{MLE} - \mu_0}{SE(\hat{\mu}_{MLE})} = \frac{2.13 - 2.0}{0.0567} = 2.29$$

Since $|z| = 2.29 > 1.96$*, we reject* H_0 *at the 5% significance level. The p-value*

$$\text{is: } p = 2 \times P(Z > 2.29) = 2 \times 0.0110 = 0.0220$$

Step 9: Model Checking with QQ-Plot

To check normality assumption, we compute standardized residuals:

$$z_i = \frac{x_i - \hat{\mu}_{MLE}}{\hat{\sigma}_{MLE}}$$

i	x_i	$z_i = (x_i - 2.13)/0.1792$	Theoretical quantile
1	2.3	0.95	0.83
2	1.9	-1.28	-1.28
3	2.1	-0.17	-0.32
4	2.4	1.51	1.28
5	2.0	-0.73	-0.67
6	2.2	0.39	0.32
7	1.8	-1.84	-1.64
8	2.3	0.95	0.83
9	2.1	-0.17	-0.32
10	2.2	0.39	0.32

Table 2.6: Standardized residuals for QQ-plot

The points approximately lie on a straight line, supporting the normality assumption.

Step 10: Interpretation and Discussion

1. ***Parameter Estimates:***
   - *The MLE for the mean is* $\hat{\mu} = 2.13$*, which is simply the sample average.*
   - *The MLE for the variance is* $\hat{\sigma}^2 = 0.0321$*, which is biased downward compared to the unbiased estimate* $s^2 = 0.0357$*.*
2. ***Bias-Variance Trade-off:***
   - *MSE of* $\hat{\sigma}^2_{MLE}$*:* $MSE(\hat{\sigma}^2_{MLE}) = Bias^2 + Variance = \frac{-\sigma^2}{n}^2 + \frac{2\sigma^4}{n-1}$
   - *For our data:* $Bias = \mathrm{E}[\hat{\sigma}^2] - \sigma^2 \approx \frac{\sigma^2}{10} \approx -0.00321$
   - $Variance \approx \frac{}{9} \approx \frac{2\times(0.0321)^2}{9} = 0.000229$ $2\sigma^4$
   - $MSE \approx 0.0000103 + 0.000229 = 0.0002393$
3. ***Practical Implications:***
   - *The 95% confidence interval for* μ *is (2.019, 2.241), which does not include 2.0.*
   - *The standard deviation estimate of 0.1792 suggests relatively low variability in the data.*
   - *The MLE variance estimate is about 10% smaller than the unbiased estimate, consistent with the* $\frac{n}{n-1}$ *correction factor.*
4. ***Goodness of Fit:***
   - *The log-likelihood at the MLE is:*

$$\ell(\hat{\mu}, \hat{\sigma}^2) = -\frac{n}{2}\log(2\pi) - \frac{n}{2}\log(\hat{\sigma}^2) - \frac{n}{2} = -5\log(2\pi) - 5\log(0.0321) - 5 = 9.84$$

   - $AIC = 2k - 2\ell = 2 \times 2 - 2 \times 9.84 = -15.68$

- $BIC = k \log n - 2\ell = 2 \log 10 - 2 \times 9.84 = 4.61 - 19.68 = -15.07$

Remark 2.4.8 (Small Sample Considerations)**.** *With only $n = 10$ observations:*

1. *The asymptotic normality approximation may not be perfect.*
2. *For inference about μ, we might prefer the t-distribution with $n - 1 = 9$ degrees of freedom.*
3. *The 95% confidence interval using t-distribution:*

$$\hat{\mu} \pm t_{0.025,9} \times \frac{s}{\sqrt{n}} = 2.13 \pm 2.262 \times \frac{0.1889}{\sqrt{10}} = 2.13 \pm 0.135$$

This gives CI: (1.995, 2.265), which is slightly wider than the asymptotic interval.

Remark 2.4.9 (Robustness Check)**.** *The data appears symmetric with no obvious outliers. However, if we had an outlier (e.g., 3.0 instead of 2.0 in position 5), the MLE would be heavily affected:*

- *With outlier: $\hat{\mu} = 2.21$, $\hat{\sigma}^2 = 0.1029$*
- *The variance estimate increases by a factor of 3.2!*
- *This demonstrates the non-robustness of MLE to outliers.*

Summary of Numerical Results

Quantity	Symbol	Value	Interpretation
Sample size	n	10	Small sample
Sample mean	$\bar{x}$	2.13	MLE for μ
MLE variance	$\hat{\sigma}^2_{\text{MLE}}$	0.0321	Biased estimate
Unbiased variance	s^2	0.0357	Corrected for bias
Standard error (mean)	$\text{SE}(\hat{\mu})$	0.0567	Precision of mean estimate
Standard error (variance)	$\text{SE}(\hat{\sigma}^2)$	0.0144	Precision of variance estimate
95% CI for μ (asymptotic)		(2.019, 2.241)	Based on normal approximation
95% CI for μ (t-distribution)		(1.995, 2.265)	More appropriate for small n
Log-likelihood at MLE	$\ell(\hat{\theta})$	9.84	Measure of model fit

Table 2.7: Summary of MLE results for the numerical example

This numerical example demonstrates the complete workflow of MLE estimation, from data to inference, highlighting both the strengths and limitations of the method in practical applications.

2.4.9 Conclusion

Maximum Likelihood Estimation remains a cornerstone of statistical inference and machine learning. Its theoretical properties (consistency, efficiency, asymptotic normality) make it a preferred method for parameter estimation. However, practitioners must be aware of its limitations regarding small samples, model misspecification, and computational challenges. The connection between MLE and common machine learning loss functions provides a prob-abilistic foundation for many algorithms, bridging classical statistics and modern machine learning.

Score Function and Information Matrix

The **score function** is the gradient of the log-likelihood:

$$S(\theta) = \nabla_\theta \ell(\theta; X) = \sum_{i=1} \nabla_\theta \log p(x_i|\theta)$$

At the MLE, $S(\hat{\theta}) = 0$ (first-order condition).

The **Fisher information matrix** measures the curvature of the log-likelihood:

$$I(\theta) = -\mathrm{E}\left[\frac{\partial^2 \ell(\theta; X)}{\partial \theta^2}\right] = \mathrm{E}\left[S(\theta) S(\theta)^\top\right]$$

Example 2.4.4 (Complete MLE Derivation for Gaussian Distribution)**.** *Consider n i.i.d. samples* $X = \{x_1, \ldots, x_n\}$ *from* $\mathcal{N}(\mu, \sigma^2)$ *with* ***both parameters unknown****. We derive MLE for both* μ *and* σ^2*.*

Step 1: Likelihood Function

$$L(\mu, \sigma^2; X) = \prod_{i=1} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(x_i - \mu)^2}{2\sigma^2}\right]$$

$$= (2\pi\sigma^2)^{-n/2} \exp\left[-\frac{1}{2\sigma^2} \sum_{i=1} (x_i - \mu)^2\right]$$

Step 2: Log-Likelihood

$$\ell(\mu, \sigma^2; X) = \log L(\mu, \sigma^2; X)$$

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

Step 3: Partial Derivatives

For μ*:*

$$\frac{\partial \ell}{\partial \mu} = \frac{1}{\sigma^2} \sum_{i=1} (x_i - \mu) = \frac{1}{\sigma^2} \left[\sum_{i=1} x_i - n\mu\right]$$

For σ^2*:*

$$\frac{\partial \ell}{\partial \sigma^2} \quad 2\sigma^2 \quad 2(\sigma^2)^2 \quad {}^{n} \quad {}_{i}{}^{2}$$

$$= -\frac{n}{} + \frac{1}{} \sum_{i=} (x \quad -\mu)$$

Step 4: Solve MLE Equations

Setting $\frac{\partial \ell}{\partial \mu} = 0$*:*

$$\frac{1}{\sigma^2}\sum_{i=1}^{n} x_i - n\mu = 0 \Rightarrow \sum_{i=1}^{n} x_i = n\mu \Rightarrow \hat{\mu}_{MLE} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Setting $\frac{\partial \ell}{\partial \sigma^2} = 0$*:*

$$-\frac{n}{2\sigma^2} + \frac{1}{2(\sigma^2)^2}\sum_{i=1}^{n}(x_i - \mu)^2 = 0$$

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

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

Step 5: Verify Second-Order Conditions

The Hessian matrix at MLE:

$$H(\hat{\mu}, \hat{\sigma}^2) = \begin{bmatrix} -\frac{n}{\sigma^2} & -\frac{1}{(\sigma^2)^2}\sum_{i=1}^{n}(x_i - \hat{\mu}) \\ -\frac{1}{(\sigma^2)^2}\sum_{i=1}^{n}(x_i - \hat{\mu}) & \frac{n}{2(\sigma^2)^2} - \frac{1}{(\sigma^2)^3}\sum_{i=1}^{n}(x_i - \hat{\mu})^2 \end{bmatrix}$$

Since $\sum_{i=1}^{n}(x_i - \hat{\mu}) = 0$*, this simplifies to:*

$$H(\hat{\mu}, \hat{\sigma}^2) = \begin{bmatrix} -\frac{n}{\sigma^2} & 0 \\ 0 & -\frac{n}{2(\sigma^2)^2} \end{bmatrix}$$

which is negative definite (diagonal with negative entries), confirming we have a maximum.

Final MLE Results:

$$\boxed{\hat{\mu}_{MLE} = \frac{1}{n}\sum_{i=1}^{n} x_i} \quad \textit{and} \quad \boxed{\hat{\sigma}^2_{MLE} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{\mu}_{MLE})^2}$$

MLE for Multivariate Gaussian Distribution

For d-dimensional data $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ with $\mathbf{x}_i \in \mathbb{R}^d$ from $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$:

Likelihood:

$$L(\boldsymbol{\mu}, \boldsymbol{\Sigma}) = (2\pi)^{-nd/2}|\boldsymbol{\Sigma}|^{-n/2} \exp\left(-\frac{1}{2}\sum_{i=1}^{n}(\mathbf{x}_i - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}_i - \boldsymbol{\mu})\right)$$

MLE Estimates:

$$\hat{\boldsymbol{\mu}}_{\text{MLE}} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i$$

$$\hat{\boldsymbol{\Sigma}}_{\text{MLE}} = \frac{1}{n} \sum_{i=1}^{n} (\mathbf{x}_i - \hat{\boldsymbol{\mu}}_{\text{MLE}})(\mathbf{x}_i - \hat{\boldsymbol{\mu}}_{\text{MLE}})^\top$$

Connection to Least Squares and Cross-Entropy

Theorem 2.4.1 (MLE for Linear Regression)**.** *For linear regression model* $y_i = \mathbf{w}^\top \mathbf{x}_i + \epsilon_i$ *with* $\epsilon_i \sim \mathrm{N}(0, \sigma^2)$*, the MLE of* $\mathbf{w}$ *is equivalent to the ordinary least squares (OLS) solution:*

$$\hat{\mathbf{w}}_{MLE} = \arg\min_{\mathbf{w}} \sum_{i=1} (y_i \quad \mathbf{w}^\top \mathbf{x}_i)^2 = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

Proof. The likelihood for linear regression is:

$$L(\mathbf{w}, \sigma^2) = (2\pi\sigma^2)^{-n/2} \exp\left(-\frac{1}{2\sigma^2} \sum_{i=1} (y_i - \mathbf{w}^\top \mathbf{x}_i)^2 \right)$$

n

Maximizing $L(\mathbf{w}, \sigma^2)$ w.r.t. $\mathbf{w}$ is equivalent to minimizing the negative log-likelihood:

$$-\log L(\mathbf{w}, \sigma^2) = \frac{n}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}\sum_{i=}^{n}(y - \mathbf{w}^\top \mathbf{x})^2$$

Ignoring constants and the σ^2 factor (since $\sigma^2 > 0$), we minimize $\sum_{i=}^{n}(y_i - \mathbf{w}^\top \mathbf{x}_i)^2$, which
is exactly the OLS objective. □

Theorem 2.4.2 (MLE for Classification)**.** *For binary classification with labels* $y_i \in \{0, 1\}$ *and model* $p(y = 1 \mid \mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x})$ *where* σ *is the sigmoid function, MLE leads to cross-entropy loss:*

$$\hat{\mathbf{w}}_{MLE} = \arg\min_{\mathbf{w}} -\sum_{i=1}^{n} y_i \log(\sigma(\mathbf{w}^\top \mathbf{x}_i)) + (1 - y_i)\log(1 - \sigma(\mathbf{w}^\top \mathbf{x}_i))$$

Numerical Optimization for MLE

For complex models where closed-form solutions don't exist, MLE requires numerical opti-mization:

Algorithm 1: Gradient-Based MLE Optimization

Input: Data X, model $p(x|\theta)$, initial θ_0, learning rate η, tolerance ϵ
Output: MLE estimate $\hat{\theta}$
for $t = 0, 1, 2, \ldots$ *until convergence*
  do Compute gradient: $g_t = \nabla_\theta \ell(\theta_t; X)$
  Update parameters: $\theta_{t+1} = \theta_t + \eta g_t$ (gradient ascent)
  if $||g_t|| < \epsilon$ **then**
    return θ_{t+1}
  end
end

Alternative approaches:

- **Newton-Raphson:** Uses second-order information: $\theta_{t+1} = \theta_t - H^{-1}(\theta_t)g_t$
- **Expectation-Maximization (EM):** For models with latent variables
- **Stochastic Gradient Ascent:** For large datasets

Regularized MLE and Bayesian Interpretation

Adding regularization corresponds to Maximum A Posteriori (MAP) estimation:

$$\hat{\theta}_{\text{MAP}} = \arg\max_{\theta} \left[\log p(\theta) + \sum_{i=1}^{n} \log p(x_i|\theta)\right]$$

- L_2 regularization $\propto \|\theta\|^2$: Equivalent to Gaussian prior $p(\theta) \sim \mathrm{N}(0, \lambda^{-1}I)$
- L_1 regularization $\propto \|\theta\|_1$: Equivalent to Laplace prior $p(\theta) \sim \mathrm{Laplace}(0, b)$

Applications in Deep Learning

1. Neural Network Training: Training deep networks via maximum likelihood corre-sponds to minimizing negative log-likelihood:

$$\mathrm{L}(\theta) = -\frac{1}{n}\sum_{i=1} \log p(y_i | \mathbf{x}_i; \theta)$$

2. Generative Models:

- **VAEs (Variational Autoencoders):** Maximize evidence lower bound (ELBO)
- **Normalizing Flows:** Direct likelihood maximization via change of variables
- **Autoregressive Models:** Sequence likelihood factorization

3. Reinforcement Learning: Policy gradient methods maximize expected return, which can be framed as MLE of optimal policy.

Limitations and Extensions

Limitations of MLE:

1. Can overfit with limited data
2. May be biased for small samples (e.g., $\hat{\sigma}^2_{\mathrm{ML}}$ is biased by factor $\frac{n-1}{n}$)
3. Requires model specification (susceptible to model misspecification)
4. May not exist or be unique for some models

Extensions:

- **Quasi-MLE:** Robust to certain model misspecifications
- **Penalized MLE:** Adds regularization terms
- **Composite Likelihood:** Uses lower-dimensional marginal likelihoods
- **Generalized Method of Moments (GMM):** Alternative to MLE

Computational Example: MLE for Exponential Distribution

Example 2.4.5 (MLE for Exponential Distribution). *For* $X_i \overset{i.i.d.}{\sim} Exp(\lambda)$ *with pdf* $f(x|\lambda) = \lambda e^{-\lambda x}$ *for* $x \geq 0$*:*

$$L(\lambda; X) = \prod_{i=1}^{n} \lambda e^{-\lambda x_i} = \lambda^n e^{-\lambda \sum_{i=1}^{n} x_i}$$

$$\ell(\lambda; X) = n \log \lambda - \lambda \sum_{i=1}^{n} x_i$$

$$\frac{d\ell}{d\lambda} = \frac{n}{\lambda} - \sum_{i=1}^{n} x_i = 0$$

$$\Rightarrow \hat{\lambda}_{MLE} = \frac{n}{\sum_{i=1}^{n} x_i} = \frac{1}{\bar{x}}$$

Statistical Inference from MLE

Given MLE $\hat{\theta}$, we can construct:

- **Confidence Intervals:** Using asymptotic normality: $\hat{\theta} \pm z_{\alpha/2} \cdot \text{SE}(\hat{\theta})$

- **Hypothesis Tests:** Wald test: $W = \frac{\hat{\theta} - \theta_0}{\text{SE}(\hat{\theta})} \sim \text{N}(0, 1)$
- **Likelihood Ratio Test:** $\Lambda = -2 \log \frac{L(\theta_0)}{L(\hat{\theta})} \sim \chi^2_k$

Information Criteria for Model Selection

$$\text{AIC} = -2\ell(\hat{\theta}) + 2k, \quad \text{BIC} = -2\ell(\hat{\theta}) + k \log n$$

where k = number of parameters, n = sample size. Lower values indicate better models.

Remark 2.4.10 (Practical Considerations)**.** *1. Always check regularity conditions (iden-tifiability, support independent of θ, etc.)*

2. *Use multiple starting points for non-convex likelihoods*
3. *Validate MLE estimates with bootstrap or cross-validation*
4. *Consider robust alternatives when data has outliers*

Regularization: An Overview

Definition

Regularization refers to techniques used to prevent overfitting in machine learning models by adding a penalty term to the loss function. The primary objective is to reduce model complexity without significantly increasing bias, thereby improving generalization to unseen data.

Mathematical Foundation

Given a loss function $L(\theta)$ where θ represents model parameters, the regularized objective becomes:

$$J(\theta) = L(\theta) + \lambda R(\theta)$$

where $\lambda \geq 0$ is the regularization parameter controlling the penalty strength, and $R(\theta)$ is the regularization term.

Bias-Variance Tradeoff

Regularization directly addresses the bias-variance tradeoff:

- **Low** λ: Lower bias but higher variance (risk of overfitting)
- **High** λ: Higher bias but lower variance (risk of underfitting)

Common Regularization Techniques

1. L2 Regularization (Ridge Regression)

- **Formulation**: $R(\theta) = \sum_{i=}^{n} \theta_i^2 = ||\theta||_2^2$
- **Effect**: Shrinks coefficients toward zero but never eliminates them entirely
- **Properties**:
  - Differentiable everywhere, suitable for gradient-based optimization
  - Preserves all features (non-sparse solutions)
  - Prevents coefficients from becoming too large
- **Mathematical Insight**: Equivalent to placing a Gaussian prior on parameters in Bayesian framework

2.4.10 L1 Regularization (Lasso Regression)

- **Formulation**: $R(\theta) = \sum_{i=}^{n} |\theta_i| = ||\theta||_1$
- **Effect**: Can force some coefficients to exactly zero, performing feature selection
- **Properties**:
  - Creates sparse models (automatic feature selection)
  - Not differentiable at zero (requires specialized optimization)
  - Particularly useful when dealing with high-dimensional data
- **Mathematical Insight**: Equivalent to placing a Laplace prior on parameters

3. Elastic Net Regularization

- **Formulation**: Combines L1 and L2: $R(\theta) = \alpha||\theta||_1 + (1-\alpha)||\theta||^2$
- **Parameters**: λ controls overall strength, $\alpha \in [0, 1]$ controls L1/L2 mix
- **Advantages**:
  - Inherits benefits of both L1 and L2 regularization
  - Handles correlated features better than Lasso alone
  - More stable than Lasso when $n < p$ (samples ¡ features)

4. Dropout (for Neural Networks)

- **Mechanism**: Randomly ”drop” neurons during training with probability p
- **Effect**: Prevents co-adaptation of neurons, creates ensemble effect
- **Properties**:
  - Applied only during training, disabled during inference
  - Acts as approximate Bayesian model averaging
  - Reduces overfitting without explicit penalty term
- **Mathematical Insight**: Multiplies activations by binary mask during training

5. Early Stopping

- **Concept**: Stop training when validation error starts increasing
- **Implementation**: Monitor validation loss, stop when it increases for k consecutive epochs
- **Advantages**:
  - Simple to implement, computationally efficient
  - Effective for iterative algorithms (gradient descent, boosting)
  - No modification to loss function required

2.5 Comparative Analysis Practical Considerations Choosing Regularization Parameters

- **Cross-validation**: Use k-fold CV to select optimal λ
- **Grid search**: Systematically test multiple λ values
- **Automated methods**: Bayesian optimization, random search

When to Use Each Technique

- **L2**: When all features are potentially relevant
- **L1**: For feature selection in high-dimensional data
- **Elastic Net**: When features are correlated and dimensionality is high
- **Dropout**: Primarily for deep neural networks
- **Early Stopping**: For iterative algorithms with clear training dynamics

Mathematical Example

For linear regression with L2 regularization:

$$\hat{\theta} = \arg\min_{\theta} \left(\sum_{i=} (y^{(i)} - \theta^T x^{(i)})^2 + \lambda \sum_{j=1} \theta_j^2 \right)$$

Closed-form solution (Ridge):

$$\hat{\theta} = (X^T X + \lambda I)^{-1} X^T y$$

Summary

Regularization techniques are essential tools for controlling model complexity and improving generalization. The choice of technique depends on the problem characteristics, model type, and computational constraints. A combination of regularization methods often yields the best results in practice.

2.6 Stochastic Gradient Descent

Definition 2.6.1 (Stochastic Gradient Descent)**.** *Stochastic Gradient Descent (SGD) is an iterative optimization algorithm used to minimize loss functions in machine learning and deep learning. Unlike batch gradient descent which uses the entire dataset to compute gradients, SGD updates parameters using a single randomly selected data point or a small random subset (mini-batch) at each iteration, making it computationally efficient for large-scale problems.*

2.6.1 Mathematical Formulation

Given a loss function $J(\theta)$ over parameters θ and dataset $\{(x_i, y_i)\}_{i=1}^{n}$, SGD minimizes:

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} L_i(\theta; x_i, y_i)$$

Algorithm Steps:

1. Initialize parameters θ_0
2. For iteration $t = 1, 2, \ldots$ until convergence:
   - Randomly select/sample index $i_t \in \{1, \ldots, n\}$
   - Compute gradient estimate: $g_t = \nabla_\theta L(\theta_{t-1}; x_{i_t}, y_{i_t})$
   - Update parameters: $\theta_t = \theta_{t-1} - \eta_t g_t$

Mini-batch SGD

A practical variant uses mini-batches of size b:

$$g_t = \frac{1}{b} \sum_{j=1}^{b} \nabla_\theta L(\theta_{t-1}; x_{i_j}, y_{i_j})$$

2.6.2 Theoretical Foundations

Theorem 2.6.1 (SGD Convergence Conditions)**.** *Under the following conditions, SGD con-verges to a local minimum with probability 1:*

1. ***Learning rate schedule:*** $\sum_{t=1}^{\infty} \eta_t = \infty$ *and* $\sum_{t=1}^{\infty} \eta^2 < \infty$
2. ***Smoothness:*** $L(\theta)$ *is L-smooth:* $\|\nabla L(\theta) - \nabla L(\theta')\| \leq L\|\theta - \theta'\|$
3. ***Bounded gradients:*** $\mathrm{E}[\|g_t\|^2] \leq G^2$ *for some* $G > 0$
4. ***Convexity (optional):*** *For convex functions, convergence to global minimum*

Convergence Rate Analysis

For convex functions:

$$\mathrm{E}[J(\theta_T) - J(\theta^*)] \leq \frac{\|\theta_0 - \theta^*\|^2}{2T\eta} + \frac{\eta G^2}{2}$$

Choosing $\eta = \frac{\|\theta_0 - \theta^*\|}{G\sqrt{T}}$ gives:

$$\mathrm{E}[J(\theta_T) - J(\theta^*)] \leq \frac{\|\theta_0 - \theta^*\| G}{\sqrt{T}}$$

For non-convex functions (common in deep learning):

$$\frac{1}{T}\sum_{t=1}^{T} \mathrm{E}[\|\nabla J(\theta_t)\|^2] \leq \frac{2(J(\theta_0) - J^*)}{T\eta}$$

$+ \eta L G^2$

Sketch of SGD Convergence. For L-smooth function J:

$$J(\theta_{t+1}) \leq J(\theta_t) + \langle \nabla J(\theta_t), \theta_{t+1} - \theta_t \rangle + \frac{L}{2} \|\theta_{t+1} - \theta_t\|^2$$
$$= J(\theta_t) - \eta_t \langle \nabla J(\theta_t), g_t \rangle + \frac{L\eta_t^2}{2} \|g_t\|^2$$

Taking expectations and using $\mathrm{E}[g_t] = \nabla J(\theta_t)$:

$$\mathrm{E}[J(\theta_{t+1})] \leq \mathrm{E}[J(\theta_t)] - \eta_t \|\nabla J(\theta_t)\|^2 + \frac{L\eta_t^2}{2} G^2$$

Summing over $t = 0$ to $T - 1$ and rearranging yields the convergence bounds.□

2.6.3 Comparison of Gradient Descent Variants

2.6.4 Practical Implementations and Variants

Momentum SGD

Adds velocity term to damp oscillations:

$$v_t = \beta v_{t-1} + (1 - \beta) g_t \; \theta_t = \theta_{t-1} - \eta v_t$$

where $\beta \in [0, 1)$ is the momentum coefficient.

Adaptive Learning Rate Methods

RMSprop: Adapts learning rate per parameter:

$$s_t = \rho s_{t-1} + (1 - \rho) g_t \odot g_t$$
$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{s_t + \epsilon}} \odot g_t$$

Adam (Adaptive Moment Estimation): Combines momentum and adaptive learn-ing:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \text{ (first moment)}$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t \odot g_t \text{ (second moment)}$$
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$
$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

2.6.5 Learning Rate Schedules

Common schedules:

- **Constant:** $\eta_t = \eta_0$

- **Time-based decay:** $\eta_t = \eta_0 / (1 + \delta t)$
- **Step decay:** $\eta_t = \eta_0 \cdot \delta^{\lfloor t/\tau \rfloor}$
- **Exponential decay:** $\eta_t = \eta_0 \cdot e^{-\delta t}$
- **Cosine annealing:** $\eta_t = \eta_{\min} + \frac{1}{2}(\eta_0 - \eta_{\min})(1 + \cos(\pi t / T))$

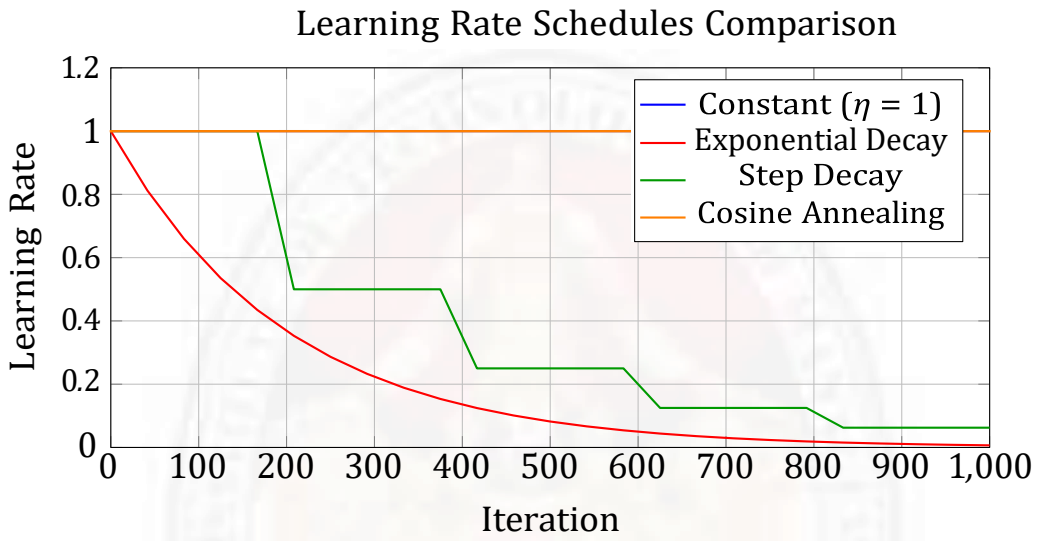


Figure 2.5: Common learning rate schedules for SGD

2.6.6 Applications and Examples Linear Regression with SGD

For linear regression with MSE loss $L(\theta) = \frac{1}{2}(y - \theta^T x)^2$:

$$\nabla_\theta L = -(y - \theta^T x)x$$

SGD update: $\theta_t = \theta_{t-1} + \eta_t(y_i - \theta_{t-1}^T x_i)x_i$

Logistic Regression with SGD

For binary classification with sigmoid $\sigma(z) = 1/(1 + e^{-z})$:

$$L(\theta) = -y \log(\sigma(\theta^T x)) - (1 - y) \log(1 - \sigma(\theta^T x))$$

Gradient: update:

T

SGD

$$\nabla_\theta L = (\sigma(\theta \ \ x) - y)x$$

$$\theta_t = \theta_{t-1} - \eta_t(\sigma(\theta_1^T \ \ x_i) - y_i)x_i$$

Example 2.6.1 (SGD for Quadratic Optimization). *Minimize* $J(\theta) = \frac{1}{2}\theta^T A\theta - b^T\theta$ *where A is positive definite.*

True minimum: $\theta^* = A^{-1}b$

SGD implementation: *Sample rows*$_i$ *of A and corresponding* b_i*:* a^T

$$g_t = (a_i^T\theta_{t-1} - b_i)a_i$$

Update: $\theta_t = \theta_{t-1} - \eta_t g_t$

Convergence analysis: *With* $\eta_t = \frac{1}{\lambda_{\max} t}$*, where* $\lambda_{\max}$ *is largest eigenvalue of A:*

$$\mathrm{E}[\|\theta_t - \theta^*\|^2] \leq \frac{C}{t}$$

showing linear convergence rate.

2.6.7 Advantages and Limitations

2.6.8 Practical Implementation Guidelines

Best practices:

1. **Initialize properly:** Use appropriate initialization (Xavier/He for neural networks)
2. **Normalize data:** Scale features to zero mean, unit variance
3. **Monitor learning:** Plot loss curves, track metrics
4. **Use validation set:** Early stopping to prevent overfitting
5. **Gradient clipping:** Prevent exploding gradients: $g_t \leftarrow \frac{g_t}{\max(1, \|g_t\|)}$
6. **Check gradients:** Numerical gradient checking during development

---

Pseudo-code for Mini-batch SGD:

Algorithm 2: Mini-batch Stochastic Gradient Descent

Input: Learning rate η, batch size b, epochs E, initial parameters θ
Output: Optimized parameters θ
for *epoch* = 1 **to** E **do**
  Shuffle training data;
  for *each mini-batch* B *of size* b **do**
    Compute gradient: $g = \frac{1}{b}\sum_{(x,y)\in B} \nabla_\theta L(\theta; x, y)$;
    Update parameters: $\theta \leftarrow \theta - \eta g$;
  end
end

2.6.9 Recent Advances and Extensions

Accelerated SGD Methods

- **Nesterov Accelerated Gradient (NAG):** "Look-ahead" momentum

$$v_t = \beta v_{t-1} + \eta \nabla J(\theta_{t-1} - \beta v_{t-1})$$
$$\theta_t = \theta_{t-1} - v_t$$

- **SGD with Warm Restarts:** Combine with cosine annealing
- **Lookahead Optimizer:** Maintains slow and fast weights

Variance Reduction Techniques

- **SVRG (Stochastic Variance Reduced Gradient):** Uses control variates
- **SAGA:** Improved version of SVRG
- **SAM (Sharpness-Aware Minimization):** Seeks flat minima

2.6.10 Connection to Statistical Learning Theory

SGD can be viewed as stochastic approximation of Robbins-Monro algorithm:

$$\theta_{t+1} = \theta_t - \eta_t H(\theta_t,$$

ξ_t) where $H(\theta, \xi)$ is unbiased estimate of $\nabla J(\theta)$.

Generalization bounds: For SGD on convex problems:

$$\mathrm{E}[J(\theta_T) - J(\theta^*)] \leq O\left(\frac{1}{\sqrt{T}}\right)$$

2.6.11 Summary

Stochastic Gradient Descent is a fundamental optimization algorithm in machine learning that balances computational efficiency with convergence guarantees. Its stochastic nature introduces noise that can help escape poor local minima but requires careful tuning of learn-ing rates and batch sizes. Modern variants with momentum and adaptive learning rates have made SGD the de facto standard for training deep neural networks on large-scale datasets.

Remark 2.6.1 (Practical Note)**.** *In practice, SGD with momentum (or Adam) with a care-fully tuned learning rate schedule often outperforms vanilla SGD. The choice between SGD variants depends on the specific problem, dataset size, and computational constraints.*

2.7 Deep Feed Forward Networks

2.7.1 Learning XOR Problem

Definition 2.7.1 (XOR Function)**.** *The XOR (exclusive OR) function:*

$$f(x_1, x_2) = \begin{cases} 0 & \text{if } x_1 = x_2 \\ 1 & \text{if } x_1 \neq x_2 \end{cases}$$

Why Single-Layer Perceptron Fails: The decision boundary must be linear: $w_1x_1 + w_2x_2 + b = 0$. No such line separates XOR points.

Problem 2.7.1 (Complete XOR Solution)**.** *Design a neural network with one hidden layer (2 neurons) to solve XOR. Use sigmoid activation and provide weights that exactly solve XOR.*

Solution 2.7.1. *We need to find weights such that:*

$$h_1 = \sigma(w_{11}x_1 + w_{12}x_2 + b_1)\ h_2 = \sigma(w_{21}x_1 + w_{22}x_2 + b_2)\ \hat{y} = \sigma(v_1h_1 + v_2h_2 + b_3)$$

Let's choose weights to implement:

$$h_1 = OR(x_1, x_2)$$
$$h_2 = NAND(x_1, x_2)$$
$$\hat{y} = AND(h_1, h_2) = XOR(x_1, x_2)$$

For OR gate with sigmoid (threshold at 0.5):

$$\sigma(w_{11}x_1 + w_{12}x_2 + b_1) > 0.5 \text{ when } x_1 = 1 \text{ or } x_2 = 1$$

Choose $w_{11} = w_{12} = 5$, $b_1 = -2$. *Then:*

- $x_1 = 0, x_2 = 0$: $5 \cdot 0 + 5 \cdot 0 - 2 = -2 \Rightarrow \sigma(-2) = 0.119 < 0.5$
- $x_1 = 0, x_2 = 1$: $5 \cdot 0 + 5 \cdot 1 - 2 = 3 \Rightarrow \sigma(3) = 0.953 > 0.5$
- $x_1 = 1, x_2 = 0$: $5 \cdot 1 + 5 \cdot 0 - 2 = 3 \Rightarrow \sigma(3) = 0.953 > 0.5$
- $x_1 = 1, x_2 = 1$: $5 \cdot 1 + 5 \cdot 1 - 2 = 8 \Rightarrow \sigma(8) = 0.9997 > 0.5$

For NAND gate:

$$\sigma(w_{21}x_1 + w_{22}x_2 + b_2) > 0.5 \text{ when } NOT(x_1 = 1 \text{ and } x_2 = 1)$$

Choose $w_{21} = w_{22} = -5$, $b_2 = 7$. *Then:*

- $x_1 = 0, x_2 = 0$: $-5 \cdot 0 - 5 \cdot 0 + 7 = 7 \Rightarrow \sigma(7) = 0.9991 > 0.5$
- $x_1 = 0, x_2 = 1$: $-5 \cdot 0 - 5 \cdot 1 + 7 = 2 \Rightarrow \sigma(2) = 0.8808 > 0.5$

- $x_1 = 1, x_2 = 0$: $-5 \cdot 1 - 5 \cdot 0 + 7 = 2 \Rightarrow \sigma(2) = 0.8808 > 0.5$
- $x_1 = 1, x_2 = 1$: $-5 \cdot 1 - 5 \cdot 1 + 7 = -3 \Rightarrow \sigma(-3) = 0.0474 < 0.5$

For AND gate:

$$\sigma(v_1 h_1 + v_2 h_2 + b_3) > 0.5 \text{ when } h_1 = 1 \text{ and } h_2 = 1$$

Choose $v_1 = v_2 = 5$, $b_3 = -7$. *Then XOR truth table:*
With threshold 0.5, this solves XOR perfectly. The network architecture is:

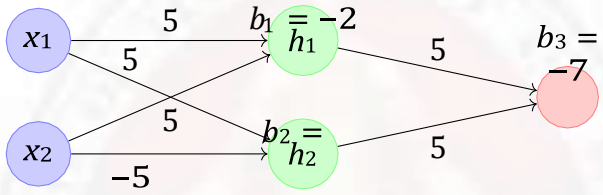


Figure 2.6: Neural network for XOR problem

Problem Statement

Given the following binary classification dataset:

x_1	x_2	y
1	1	1
2	2	1
0	0	0
−1	−1	0

Initial Parameters

$$w_1 = 0.2, \quad w_2 = 0.3, \quad b = -1, \quad \eta = 0.1$$

Perceptron Learning Algorithm

1. Perceptron Model

The perceptron output for an input (x_1, x_2) is given by:

$$\hat{y} = \begin{cases} 1 & \text{if } w_1 x_1 + w_2 x_2 + b \geq 0 \\ 0 & \text{if } w_1 x_1 + w_2 x_2 + b < 0 \end{cases}$$

2. Learning Rule

For each training example (x_1, x_2, y), update weights and bias if misclassified:

$$w_1 \leftarrow w_1 + \eta(y - \hat{y})x_1 \quad w_2 \leftarrow w_2 + \eta(y - \hat{y})x_2$$
$$b \leftarrow b + \eta(y - \hat{y})$$

3. Algorithm Steps

1. For each data point in the dataset:
   - Compute net input: $z = w_1x_1 + w_2x_2 + b$
   - Compute predicted output $\hat{y}$ using step function
   - If $\hat{y} \neq y$, update weights and bias
2. Repeat step 1 until all data points are correctly classified

Problem Solution

Initial Parameters:

$$w_1^{(0)} = 0.2, \quad w_2^{(0)} = 0.3, \quad b^{(0)} = -1, \quad \eta = 0.1$$

Dataset:

x_1	x_2	y
1	1	1
2	2	1
0	0	0
−1	−1	0

Perceptron Activation Function:

$$\hat{y} = \begin{cases} 1 & \text{if } w_1x_1 + w_2x_2 + b \geq 0 \\ 0 & \text{if } w_1x_1 + w_2x_2 + b < 0 \end{cases}$$

Update Rules:

$$\Delta w_1 = \eta(y - \hat{y})x_1$$
$$\Delta w_2 = \eta(y - \hat{y})x_2$$
$$\Delta b = \eta(y - \hat{y})$$

Epoch 1

Data Point 1: $(x_1, x_2) = (1, 1), y = 1$

Step 1: Compute Net Input

$$z^{(1)} = w_1^{(0)}x_1 + w_2^{(0)}x_2 + b^{(0)} = (0.2 \times 1) + (0.3 \times 1) + (-1) = -0.5$$

Step 2: Compute Activation

$$\hat{y}^{(1)} = 0 \text{ (since } z^{(1)} < 0)$$

Step 3: Check Classification

$$\hat{y}^{(1)} = 0 \neq y = 1 \quad \Rightarrow \quad \text{Misclassified}$$

Step 4: Update Parameters

$$\Delta w_1^{(1)} = \eta(y - \hat{y}^{(1)})x_1 = 0.1 \times (1 - 0) \times 1 = 0.1$$

$$\Delta w_2^{(1)} = \eta(y - \hat{y}^{(1)})x_2 = 0.1 \times (1 - 0) \times 1 = 0.1$$

$$\Delta b^{(1)} = \eta(y - \hat{y}^{(1)}) = 0.1 \times (1 - 0) = 0.1$$

$$w_1^{(1)} = w_1^{(0)} + \Delta w_1^{(1)} = 0.2 + 0.1 = 0.3$$

$$w_2^{(1)} = w_2^{(0)} + \Delta w_2^{(1)} = 0.3 + 0.1 = 0.4$$

$$b^{(1)} = b^{(0)} + \Delta b^{(1)} = -1 + 0.1 = -0.9$$

Data Point 2: $(x_1, x_2) = (2, 2), y = 1$

Step 1: Compute Net Input

$$z^{(2)} = w_1^{(1)}x_1 + w_2^{(1)}x_2 + b^{(1)} = (0.3 \times 2) + (0.4 \times 2) + (-0.9) = 0.5$$

Step 2: Compute Activation

$$\hat{y}^{(2)} = 1 \text{ (since } z^{(2)} \geq 0)$$

Step 3: Check Classification

$\hat{y}^{(2)} = 1 = y = 1 \quad \Rightarrow \quad$ Correctly classified $\Rightarrow \quad$ No update

Data Point 3: $(x_1, x_2) = (0, 0), y = 0$

Step 1: Compute Net Input

$$z^{(3)} = w_1^{(1)}x_1 + w_2^{(1)}x_2 + b^{(1)} = (0.3 \times 0) + (0.4 \times 0) + (-0.9) = -0.9$$

Step 2: Compute Activation

$$\hat{y}^{(3)} = 0 \text{ (since } z^{(3)} < 0)$$

Step 3: Check Classification

$\hat{y}^{(3)} = 0 = y = 0 \quad \Rightarrow \quad$ Correctly classified $\Rightarrow$ No update

Data Point 4: $(x_1, x_2) = (-1, -1),\ y = 0$

Step 1: Compute Net Input

$$z^{(4)} = w_1^{(1)}x_1 + w_2^{(1)}x_2 + b^{(1)} = [0.3 \times (-1)] + [0.4 \times (-1)] + (-0.9) = -1.6$$

Step 2: Compute Activation

$$\hat{y}^{(4)} = 0 \text{ (since } z^{(4)} < 0)$$

Step 3: Check Classification

$\hat{y}^{(4)} = 0 = y = 0 \quad \Rightarrow$ Correctly classified $\Rightarrow$ No update

End of Epoch 1 Summary

$$w_1^{(1)} = 0.3,\ \ w_2^{(1)} = 0.4,\ \ b^{(1)} = -0.9$$

Check all points with current parameters:

Point 1: $z = 0.3 \times 1 + 0.4 \times 1 + (-0.9) = -0.2 \Rightarrow \hat{y} = 0 \neq 1$ Misclassified

Point 2: $z = 0.3 \times 2 + 0.4 \times 2 + (-0.9) = 0.5 \Rightarrow \hat{y} =$ 1 Correct Point 3: $z = -0.9 \Rightarrow$ $\hat{y} = 0$ Correct

Point 4: $z = -1.6 \Rightarrow \hat{y} = 0$ Correct

Continue to Epoch 2

Epoch 2

Data Point 1: $(x_1, x_2) = (1, 1),\ y = 1$

Step 1: Compute Net Input

$$z^{(1)} = w_1^{(1)}x_1 + w_2^{(1)}x_2 + b^{(1)} = (0.3 \times 1) + (0.4 \times 1) + (-0.9) = -0.2$$

Step 2: Compute Activation

$$\hat{y}^{(1)} = 0 \text{ (since } z^{(1)} < 0)$$

Step 3: Check Classification

$$\hat{y}^{(1)} = 0 \neq y = 1 \quad \Rightarrow \quad \text{Misclassified}$$

Step 4: Update Parameters

$$\Delta w_1^{(2)} = \eta(y - \hat{y}^{(1)})x_1 = 0.1 \times (1 - 0) \times 1 = 0.1$$

$$\Delta w_2^{(2)} = \eta(y - \hat{y}^{(1)})x_2 = 0.1 \times (1 - 0) \times 1 = 0.1$$

$$\Delta b^{(2)} = \eta(y - \hat{y}^{(1)}) = 0.1 \times (1 - 0) = 0.1$$

$$w_1^{(2)} = w_1^{(1)} + \Delta w_1^{(2)} = 0.3 + 0.1 = 0.4$$

$$w_2^{(2)} = w_2^{(1)} + \Delta w_2^{(2)} = 0.4 + 0.1 = 0.5$$

$$b^{(2)} = b^{(1)} + \Delta b^{(2)} = -0.9 + 0.1 = -0.8$$

Data Point 2: $(x_1, x_2) = (2, 2), y = 1$

Step 1: Compute Net Input

$$z^{(2)} = w_1^{(2)} x_1 + w_2^{(2)} x_2 + b^{(2)} = (0.4 \times 2) + (0.5 \times 2) + (-0.8) = 1.0$$

Step 2: Compute Activation

$$\hat{y}^{(2)} = 1 \text{ (since } z^{(2)} \geq 0)$$

Step 3: Check Classification

$\hat{y}^{(2)} = 1 = y = 1 \quad \Rightarrow$ Correctly classified $\Rightarrow$ No update

Data Point 3: $(x_1, x_2) = (0, 0), y = 0$

Step 1: Compute Net Input

$$z^{(3)} = w_1^{(2)} x_1 + w_2^{(2)} x_2 + b^{(2)} = (0.4 \times 0) + (0.5 \times 0) + (-0.8) = -0.8$$

Step 2: Compute Activation

$$\hat{y}^{(3)} = 0 \text{ (since } z^{(3)} < 0)$$

Step 3: Check Classification

$\hat{y}^{(3)} = 0 = y = 0 \quad \Rightarrow$ Correctly classified $\Rightarrow$ No update

Data Point 4: $(x_1, x_2) = (-1, -1), y = 0$

Step 1: Compute Net Input

$$z^{(4)} = w_1^{(2)} x_1 + w_2^{(2)} x_2 + b^{(2)} = [0.4 \times (-1)] + [0.5 \times (-1)] + (-0.8) = -1.7$$

Step 2: Compute Activation

$$\hat{y}^{(4)} = 0 \text{ (since } z^{(4)} < 0)$$

Step 3: Check Classification

$\hat{y}^{(4)} = 0 = y = 0 \quad \Rightarrow$ Correctly classified $\Rightarrow$ No update

End of Epoch 2 Summary

$$w_1^{(2)} = 0.4, \quad w_2^{(2)} = 0.5, \quad b^{(2)} = -0.8$$

Final Check:

Point 1: $z = 0.4 \times 1 + 0.5 \times 1 + (-0.8) = 0.1 \Rightarrow \hat{y} = 1 = 1$ Correct

---

Point 2: $z = 0.4 \times 2 + 0.5 \times 2 + (-0.8) = 1.0 \Rightarrow \hat{y} = 1$ Correct Point 3: $z = -0.8 \Rightarrow \hat{y} = 0$ Correct

Point 4: $z = -1.7 \Rightarrow \hat{y} = 0$ Correct

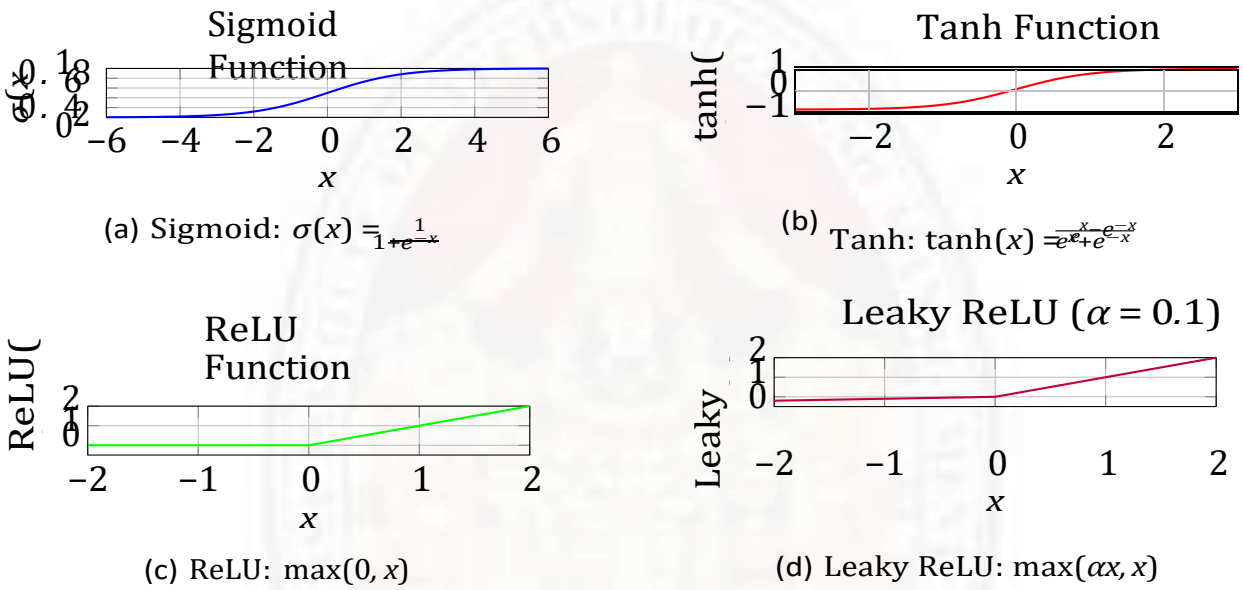
Conclusion

The perceptron converges after **2 epochs** with final parameters:

$$\boxed{w_1 = 0.4, \quad w_2 = 0.5, \quad b = -0.8}$$

All data points are now correctly classified.

2.7.2 Activation Functions



(a) Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$ (b) Tanh: $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$

(c) ReLU: $\max(0, x)$ (d) Leaky ReLU: $\max(\alpha x, x)$

Figure 2.7: Common activation functions with their graphical representations

Mathematical Properties:

1. **Sigmoid :**
$$\sigma(x) = \frac{1}{1+e^{-x}}, \quad \frac{d\sigma}{dx} = \sigma(x)(1-\sigma(x)) \in (0, 0.25]$$
Range: $(0, 1)$, Problem: Vanishing gradient for large $|x|$
2. **Tanh:**
$$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}, \quad \frac{d\tanh}{dx} = 1 - \tanh^{2}(x) \in (0, 1]$$
Range: $(-1, 1)$, Zero-centered output
3. **ReLU:**
$$\text{ReLU}(x) = \max(0, x), \quad \frac{d\text{ReLU}}{dx} = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$
Range: $[0, \infty)$, Problem: Dying ReLU

4. **Leaky ReLU:** **(**

$$\text{LReLU}(x) = \max(\alpha x, x), \quad \frac{d\text{LReLU}}{dx} = \begin{cases} 1 & \text{if } x > 0 \\ \alpha & \text{if } x \leq 0 \end{cases}$$

Solves dying ReLU problem

2.7.3 Network Architectures

McCulloch-Pitts Neuron

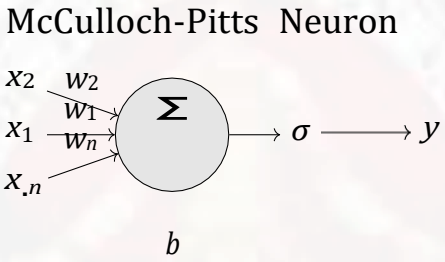


Figure 2.8: McCulloch-Pitts neuron model: $y = \sigma(\sum_n w_i x_i + b)$

Feedforward Neural Network (FNN)

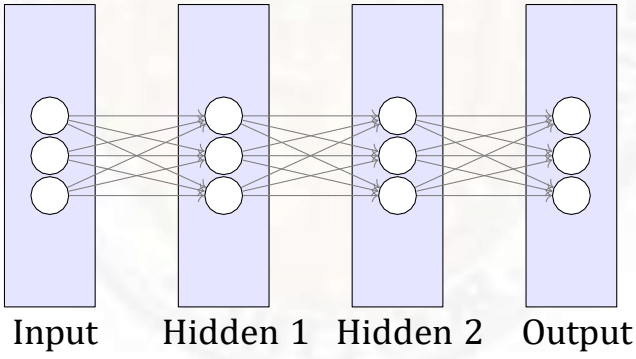


Figure 2.9: Deep Feedforward Neural Network Architecture

Mathematical Formulation:

$$z^{(1)} = W^{(1)}x + b^{(1)},\ a^{(1)} = \sigma(z^{(1)})$$
$$z^{(2)} = W^{(2)}a^{(1)} + b^{(2)},\ a^{(2)} = \sigma(z^{(2)})$$

$$z^{(3)} = W^{(3)}a^{(2)} + b^{(3)}, \hat{y} = \sigma(z^{(3)})$$

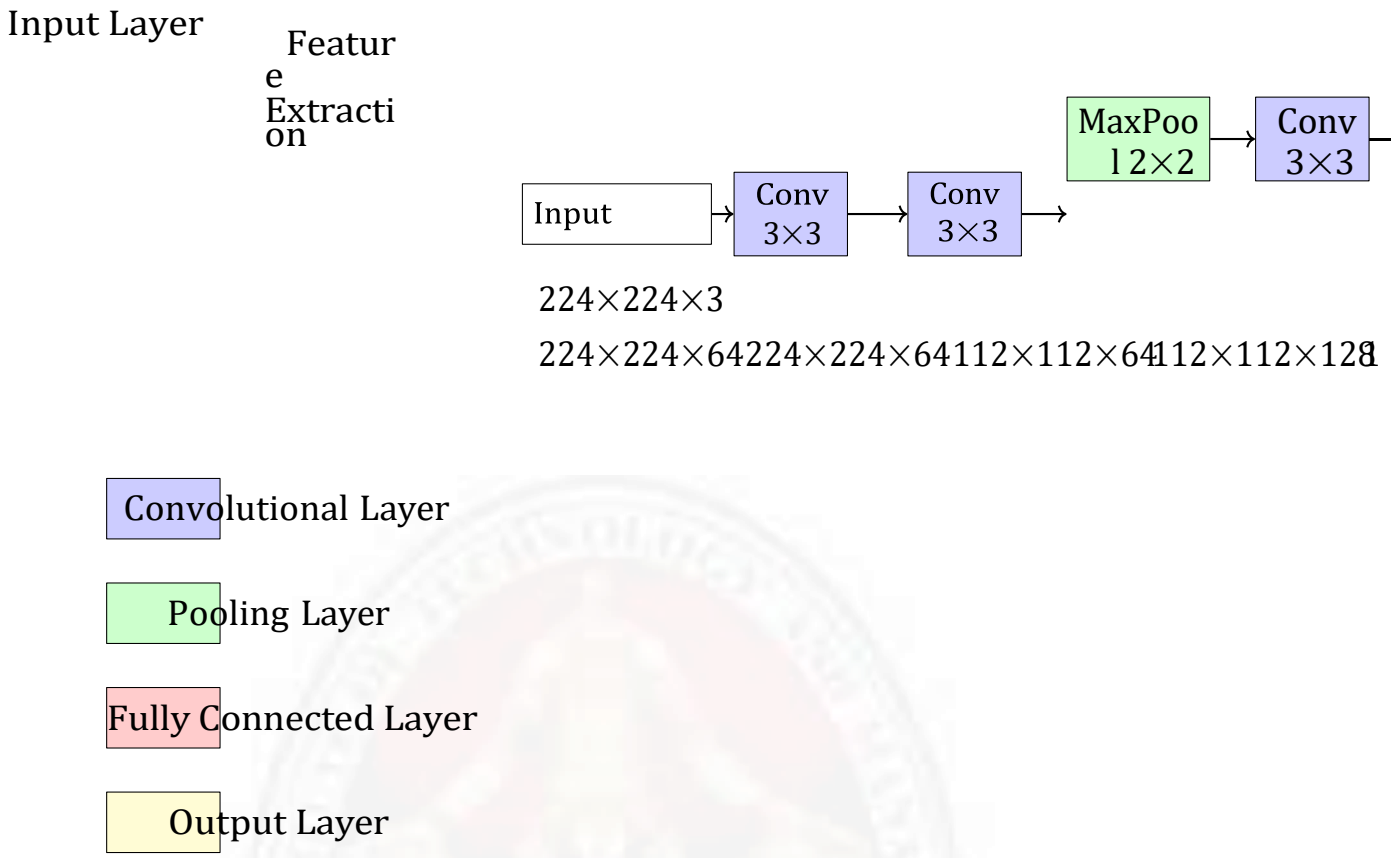
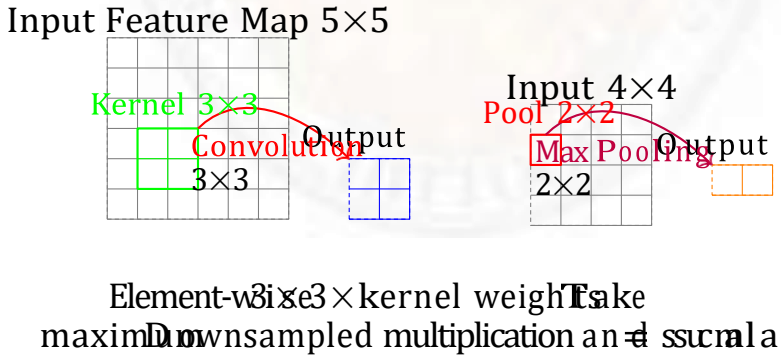


Figure 2.10: Complete CNN Architecture (VGG-style) showing the flow from input image through convolutional and pooling layers to fully connected classification layers. Each box shows layer type and dimensions (height×width×channels).



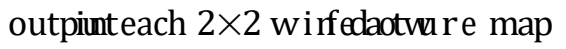


Figure 2.11: Visualization of convolution and pooling operations in CNN. Left: Convolution operation applying a 3×3 kernel to produce a 3×3 output. Right: Max pooling operation with 2×2 window reducing 4×4 input to 2×2 output.

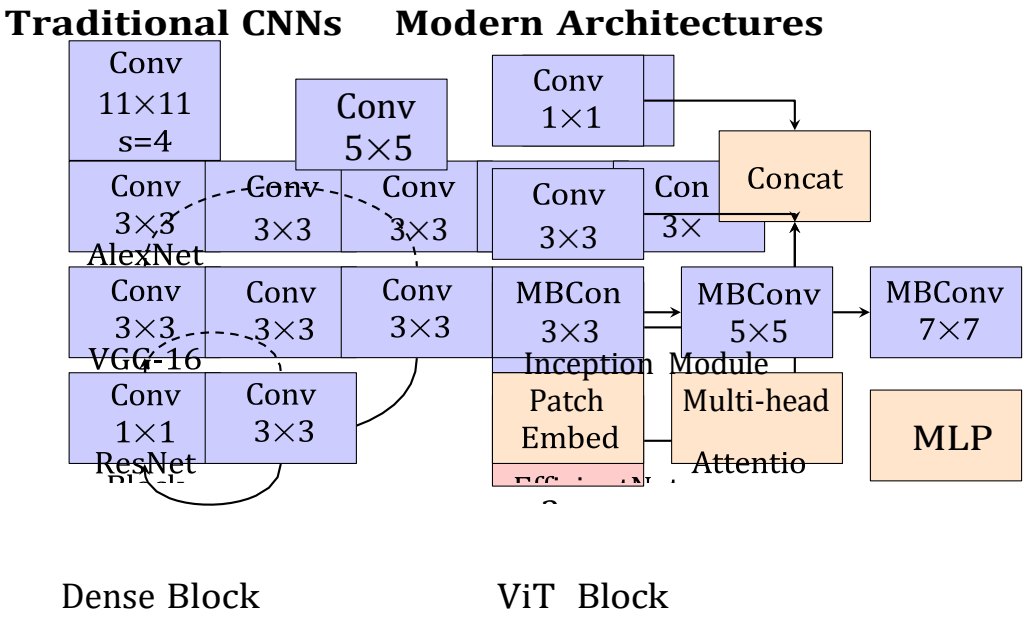


Figure 2.12: Comparison of different CNN architecture styles: Traditional sequential CNNs (AlexNet, VGG), Residual networks (ResNet), Dense connections (DenseNet), Parallel con-volutions (Inception), Efficient blocks (EfficientNet), and Transformer-based (Vision Trans-former).

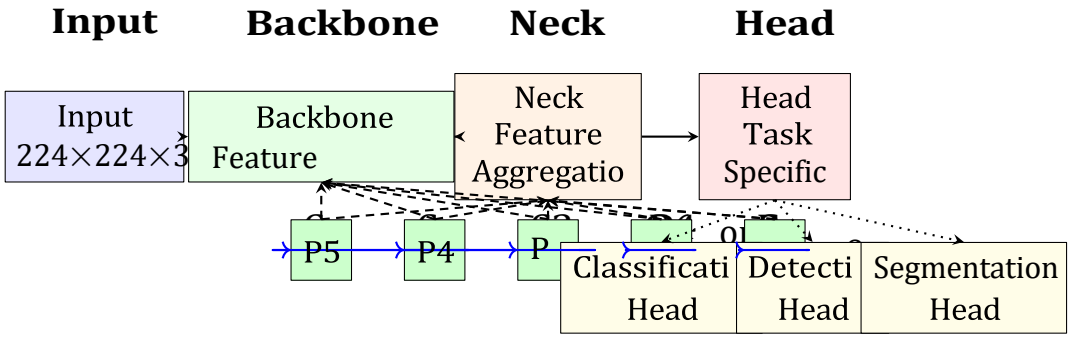


Figure 2.13: Modern CNN architecture decomposition showing backbone (feature extractor), neck (feature pyramid/aggregation), and task-specific heads. This modular design enables transfer learning and multi-task learning.

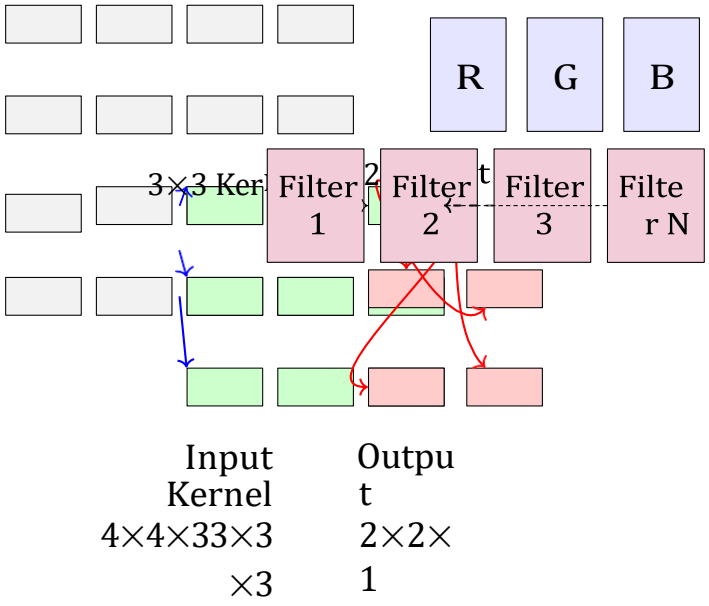


Figure 2.14: Tensor operations in CNN: Input tensor (4×4×3) convolved with multiple kernels (3×3×3) to produce output feature maps. Each filter produces one channel in the output.

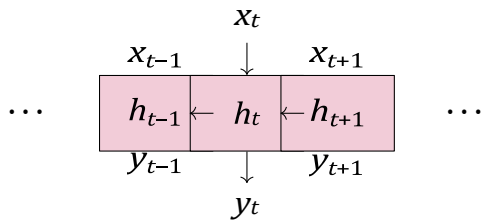


Figure 2.15: Recurrent Neural Network (unrolled in time)

Convolutional Neural Network Architecture Recurrent Neural Network (RNN) Mathematical Formulation:

$$h_t = \sigma(W_h h_{t-1} + W_x x_t + b_h), \; y_t = \sigma(W_y h_t + b_y)$$

2.7.4 Backpropagation: Complete Working Example with Two Epochs

Problem 2.7.2 (Backpropagation with Two Epochs)**.** *Consider a 2-layer network for binary classification:*

- *Input:* $x = [1, 0.5]^T$
- *Layer 1:* $W^{(1)} = \begin{matrix} 0.8 & -0.5 \\ 0.1 & 0.3 \end{matrix}$, $b^{(1)} = [0.1, 0.2]^T$, *activation: sigmoid*
- *Layer 2:* $W^{(2)} = [0.4, -0.3]$, $b^{(2)} = 0.05$, *activation: sigmoid*
- *Target:* $y = 1$
- *Loss: Binary cross-entropy:* $L = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$
- *Learning rate:* $\eta = 0.1$

Perform two complete training epochs.

Solution 2.7.2. ***Epoch 1:***

Forward Pass:

$$z^{(1)} = W^{(1)}x + b^{(1)} = \begin{matrix} 0.8 \times 1 - 0.5 \times 0.5 + 0.1 \\ 0.1 \times 1 + 0.3 \times 0.5 + 0.2 \end{matrix} = \begin{matrix} 0.8 - 0.25 + 0.1 \\ 0.1 + 0.15 + 0.2 \end{matrix} = \begin{matrix} 0.65 \\ 0.4 \end{matrix}$$

$$a^{(1)} = \sigma(z^{(1)}) = \begin{matrix} \frac{1}{1+e^{-0.65}} \\ \frac{1}{1+e^{-0.4}} \end{matrix} \approx \begin{matrix} 0.6570 \\ 0.6106 \end{matrix}$$

$$z^{(2)} = W^{(2)}a^{(1)} + b^{(2)} = 0.4 \times 0.6570 - 0.3 \times 0.6106 + 0.05$$

$$= 0.2628 - 0.1832 + 0.05 = 0.1296$$

$$\hat{y} = a^{(2)} = \sigma(z^{(2)}) = \frac{1}{1 + e^{-0.1296}} \approx 0.5324$$

$$L = -[1 \times \log(0.5324) + 0] = -\log(0.5324) \approx 0.6305$$

Backward Pass:

$$\delta^{(2)} = \frac{\partial L}{\partial z^{(2)}} = \hat{y} - y = 0.5324 - 1 = -0.4676$$

$$\frac{\partial L}{\partial W^{(2)}} = \delta^{(2)}(a^{(1)})^T = -0.4676 \times \begin{bmatrix} 0.6570 & 0.6106 \end{bmatrix} = \begin{bmatrix} -0.3072 & -0.2855 \end{bmatrix}$$

$$\frac{\partial L}{\partial b^{(2)}} = \delta^{(2)} = -0.4676$$

$$\delta^{(1)} = (W^{(2)})^T \delta^{(2)} \odot \sigma'(z^{(1)})$$

$$= \begin{bmatrix} 0.4 \\ -0.3 \end{bmatrix} \times (-0.4676) \odot \begin{matrix} 0.6570(1-0.6570) \\ 0.6106(1-0.6106) \end{matrix}$$

$$= \begin{bmatrix} -0.1870 \\ 0.140 \end{bmatrix} \odot \begin{matrix} 0.2253 \\ 0.237 \end{matrix} = \begin{matrix} -0.0421 \\ 0.033 \end{matrix}$$

$$\frac{\partial L}{\partial W} = \delta^{(1)} x^T = \begin{bmatrix} -0.0421 \\ 0.033 \end{bmatrix} \begin{bmatrix} 1 & 0.5 \end{bmatrix} = \begin{bmatrix} -0.0421 & -0.0211 \\ 0.0334 \end{bmatrix}$$

$$\frac{\partial L}{\partial b^{(1}} = \delta^{(1)} = \begin{matrix} -0.0421 \\ 0.033 \end{matrix}$$

Parameter Updates:

$$W^{(2)} \leftarrow W^{(2)} - \eta \frac{\partial L}{\partial W^{(2)}} = \begin{bmatrix} 0.4 & -0.3 \end{bmatrix} - 0.1 \times \begin{bmatrix} -0.3072 & -0.2855 \end{bmatrix}$$

$$= \begin{bmatrix} 0.4307 & -0.2715 \end{bmatrix}$$

$$b^{(2)} \leftarrow b^{(2)} - \eta \frac{\partial L}{\partial b^{(2}} = 0.05 - 0.1 \times (-0.4676) = 0.0968$$

$$W^{(1)} \leftarrow W^{(1)} - \eta \frac{\partial L}{\partial W^{(1)}} = \begin{matrix} 0.8 & -0.5 \\ 0.1 & 0.3 \end{matrix} - 0.1 \times \begin{matrix} -0.0421 & -0.0211 \\ 0.0334 & 0.0167 \end{matrix}$$

$$= \begin{matrix} 0.8042 & -0.497 \\ 0.0967 & 0.298 \end{matrix}$$

$$b^{(1)} \leftarrow b^{(1)} - \eta \frac{\partial L}{\partial b^{(1}} = \begin{matrix} 0.1 \\ 0.2 \end{matrix} - 0.1 \times \begin{matrix} -0.0421 \\ 0.033 \end{matrix} = \begin{matrix} 0.1042 \\ 0.196 \end{matrix}$$

Epoch 2:

Forward Pass with updated parameters:

$$z^{(1)} = \begin{matrix} 0.8042 & -0.4979 \\ 0.0967 & \end{matrix} \begin{matrix} 1 \\ 0.5 \end{matrix} + \begin{matrix} 0.1042 \\ 0.196 \end{matrix}$$

$$= \begin{matrix} 0.8042 \times 1 - 0.4979 \times 0.5 + 0.1042 \\ 0.0967 \times 1 + 0.2983 \times 0.5 + \\ 0.1967 \end{matrix}$$

$$= \begin{matrix} 0.8042 - 0.2490 + 0.1042 \\ 0.0967 + 0.1492 + \end{matrix} = \begin{matrix} 0.6594 \\ 0.442 \end{matrix}$$

$$a^{(1)} = \sigma(z^{(1)}) = \begin{matrix} \frac{1}{1+e^{-0.6594}} \\ \frac{1}{1+e^{-0.4426}} \end{matrix} \approx \begin{matrix} 0.6595 \\ 0.6089 \end{matrix}$$

$$z^{(2)} = \begin{matrix} 0.4307 & -0.2715 \end{matrix} \begin{matrix} 0.6595 \\ 0.6089 \end{matrix} + 0.0968$$

$$= 0.4307 \times 0.6595 - 0.2715 \times 0.6089 + 0.0968$$

$$= 0.2840 - 0.1653 + 0.0968 =$$

$$0.2155\ \hat{y} = \sigma(0.2155) \approx 0.5537$$

$$L = -\log(0.5537) \approx 0.5913$$

Observation: *Loss decreased from 0.6305 to 0.5913 after one epoch. The network is learning correctly.*

2.7.5 Backpropagation Derivation

Theorem 2.7.1 (Backpropagation Algorithm)**.** *For an L-layer neural network with sigmoid activation and mean squared error loss, the gradients are:*

$$\delta^{(L)} = (\hat{y} - y) \odot \sigma'(z^{(L)})$$

$$\delta^{(l)} = [(W^{(l+1)})^T \delta^{(l+1)}] \odot \sigma'(z^{(l)}) \text{for } l = L-1, \ldots, 1$$

$$\frac{\partial L}{\partial W^{(l)}} = \delta^{(l)} (a^{(l-1)})^T$$

$$\frac{\partial L}{\partial b^{(l)}} = \delta^{(l)}$$

where $a^{(0)} = x$ and $\odot$ denotes element-wise multiplication.

Proof. Let $L = \frac{1}{2}(y - \hat{y})^2$ and $\hat{y} = a^{(L)} = \sigma(z^{(L)})$.

Layer L:

$$\frac{\partial L}{\partial a^{(L)}} = a^{(L)} - y$$

$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'(z^{(L)}) = a^{(L)}(1 - a^{(L)})$$

$$\frac{\partial L}{\partial z^{(L)}} = \frac{\partial L}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}}$$

$$\delta^{(L)} = \frac{\partial C}{\partial z^{(L)}} = \frac{\partial C}{\partial a^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} = (a^{(L)} - y) \odot a^{(L)} (1 - a^{(L)})$$

Layer l:

$$\frac{\partial L}{\partial a^{(l)}} = (W^{(l+1)})^T \delta^{(l+1)}$$

$$\frac{\partial a^{(l)}}{\partial z^{(l)}} = a^{(l)}(1 - a^{(l)})$$

$$\delta^{(l)} = \frac{\partial L}{\partial a^{(l)}} \odot \frac{\partial a^{(l)}}{\partial z^{(l)}} = [(W^{(l+1)})^T \delta^{(l+1)}] \odot a^{(l)}(1 - a^{(l)})$$

Weight gradients:

$$\frac{\partial L}{\partial W^{(l)}} = \delta^{(l)}(a^{(l-1)})^T, \quad \frac{\partial L}{\partial b^{(l)}} = \delta^{(l)}$$

□

2.8 Mathematical Problems with Detailed Solutions

Problem 2.8.1 (Gradient Descent Convergence)**.** *Show that gradient descent with learning rate $\eta \leq 1/L$, where L is the Lipschitz constant of ∇f, converges for convex functions.*

Solution 2.8.1. *For a convex function f with L-Lipschitz gradient:*

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|$$

The gradient descent update: $x_{k+1} = x_k - \eta \nabla f(x_k)$

By Taylor expansion:

$$\begin{aligned} f(x_{k+1}) &\leq f(x_k) + \nabla f(x_k)^T (x_{k+1} - x_k) + \frac{L}{2}\|x_{k+1} - x_k\|^2 \\ &= f(x_k) - \eta\|\nabla f(x_k)\|^2 + \frac{L\eta^2}{2}\|\nabla f(x_k)\|^2 \\ &= f(x_k) - \eta\left(1 - \frac{L\eta}{2}\right)\|\nabla f(x_k)\|^2 \end{aligned}$$

For $\eta \leq 1/L$, we have $1 - \frac{L\eta}{2} \geq \frac{1}{2}$, so:

$$f(x_{k+1}) \leq f(x_k) - \frac{\eta}{2}\|\nabla f(x_k)\|^2$$

Thus, the function value decreases monotonically. For convex f, this implies convergence to the minimum.

2.9 Multiple Choice Questions

1. **Which of the following best describes underfitting?**

   (a) Model is too complex and captures noise
   (b) Model is too simple to capture data patterns
   (c) Model has optimal bias-variance tradeoff
   (d) Model performs well on test but poorly on training data

   Answer: B

2. **In the bias-variance decomposition, irreducible error represents:**

   (a) Error due to model bias
   (b) Error due to model variance
   (c) Noise inherent in the data generation process
   (d) Error from finite training samples

   Answer: C

3. **Which activation function is most susceptible to vanishing gradient?**

   (a) ReLU
   (b) Leaky ReLU
   (c) Sigmoid
   (d) Linear

   Answer: C

4. **The XOR problem cannot be solved by:**

   (a) Single-layer perceptron
   (b) Two-layer neural network with sigmoid
   (c) Two-layer neural network with ReLU
   (d) Three-layer neural network

   Answer: A

5. **In backpropagation, the chain rule is applied:**

   (a) Forward through the network
   (b) Backward from output to input

(c) Only at the output layer

(d) Randomly through the network

Answer: B

2.10 Essay Questions: Comprehensive Explanations

2.11 Essay Questions: Comprehensive Explanations

2.11.1 Bloom's Level 1: Remembering - Bias-Variance Decompo-sition

Question: List and define the three components of the bias-variance decomposition for mean squared error.

Detailed Answer:

The bias-variance decomposition is a fundamental theoretical framework in machine learning that provides deep insight into the sources of error in predictive models. For a given prediction point x, let $y = f(x) + \epsilon$ be the true relationship, where $f(x)$ is the under-lying true function and $\epsilon(0, \sigma^2)$ is irreducible noise. Our model produces predictions $\hat{f}(x)$ based on training data.

The expected prediction error for mean squared error (MSE) decomposes into three distinct components:

1. **Bias Squared (Systematic Error):**

$$\text{Bias}^2(\hat{f}(x)) = (\text{E}[\hat{f}(x)] - f(x))^2$$

Interpretation: Bias measures how much the average prediction of our model differs from the true value. It represents systematic error that occurs regardless of the specific training data used. High bias typically indicates:

- **Underfitting:** The model is too simple to capture the underlying patterns
- **Incorrect assumptions:** The model family doesn't include the true function
- **Consistent deviation:** The model makes the same type of error repeatedly

Example: A linear model trying to fit quadratic data will have high bias because no straight line can accurately represent a parabola.

2. **Variance (Random Error):**

$$\text{Var}(\hat{f}(x)) = \text{E}[(\hat{f}(x) - \text{E}[\hat{f}(x)])^2]$$

Interpretation: Variance measures how much the predictions vary for different train-ing datasets. It represents the model's sensitivity to the particular training data used. High variance typically indicates:

- **Overfitting:** The model is too complex and memorizes noise
- **Instability:** Small changes in training data lead to large changes in predictions
- **Poor generalization:** Excellent performance on training data but poor on test data

Example: A very high-degree polynomial that perfectly fits every training point but oscillates wildly between them.

3. **Irreducible Error (Noise):**

$$\sigma^2 = \text{Var}(\epsilon)$$

Interpretation: Irreducible error represents the inherent noise in the data generation process. This error cannot be reduced by any model, no matter how sophisticated. Sources include:

- **Measurement error:** Imperfect instruments or data collection
- **Stochastic processes:** Inherent randomness in the phenomenon
- **Missing information:** Unobserved variables affecting the outcome

Example: In weather prediction, even with perfect models, there will always be some uncertainty due to chaotic atmospheric dynamics.

Complete Decomposition:

$$\text{E}[(y - \hat{f}(x))^2] = \underbrace{(\text{E}[\hat{f}(x)] - f(x))^2}_{\text{Bias}^2} + \underbrace{\text{E}[(\hat{f}(x) - \text{E}[\hat{f}(x)])^2]}_{\text{Variance}} + \underbrace{\sigma^2}_{\text{Irreducible Error}}$$

Mathematical Derivation:

$$\begin{aligned}
\text{E}[(y - \hat{f})^2] &= \text{E}[(f + \epsilon - \hat{f})^2] \\
&= \text{E}[(f - \hat{f})^2] + \text{E}[\epsilon^2] + 2\text{E}[\epsilon(f - \hat{f})] \\
&= \text{E}[(f - \text{E}[\hat{f}] + \text{E}[\hat{f}] - \hat{f})^2] + \sigma^2 \\
&= \text{E}[(f - \text{E}[\hat{f}])^2] + 2\text{E}[(f - \text{E}[\hat{f}])(\text{E}[\hat{f}] - \hat{f})] \\
&\quad + \text{E}[(\text{E}[\hat{f}] - \hat{f})^2] + \sigma^2 \\
&= (f - \text{E}[\hat{f}])^2 + 0 + \text{E}[(\text{E}[\hat{f}] - \hat{f})^2] + \sigma^2 \\
&= \text{Bias}^2 + \text{Variance} + \sigma^2
\end{aligned}$$

Practical Implications:

- **Bias-Variance Tradeoff:** Decreasing bias typically increases variance, and vice versa
- **Model Selection:** Choose model complexity to balance these components

- **Regularization:** Techniques like L1/L2 regularization reduce variance at cost of in-creased bias
- **Ensemble Methods:** Bagging reduces variance, boosting reduces bias

2.11.2 Bloom's Level 2: Understanding - Single-Layer vs. Two -Layer Networks for XOR

Question: Explain why a single-layer perceptron cannot solve XOR while a two-layer net-work can.

Detailed Answer:

Mathematical Foundation

The XOR (exclusive OR) function is defined as:

$$\text{XOR}(x_1, x_2) = \begin{cases} 0 & \text{if } x_1 = x_2 \\ 1 & \text{if } x_1 /= x_2 \end{cases}$$

The truth table is:

x_1	x_2	XOR(x_1, x_2)
0	0	0
0	1	1
1	0	1
1	1	0

Single-Layer Perceptron Limitation

A single-layer perceptron implements:

$$y = \text{sign}(\mathbf{w}^\top \mathbf{x} + b) = \text{sign}(w_1 x_1 + w_2 x_2 + b)$$

This creates a **linear decision boundary** in the input space. The condition $w_1 x_1 + w_2 x_2 + b = 0$ defines a straight line that separates points into two classes.

Geometric Proof of Impossibility: Plot the XOR points in 2D space: (0, 0) and (1, 1) are class 0, (0, 1) and (1, 0) are class 1. No single straight line can separate these points such that all class 0 points are on one side and all class 1 points on the other.

Algebraic Proof: Assume there exist w_1, w_2, b such that:

$$w_1 \cdot 0 + w_2 \cdot 0 + b < 0 \quad \text{(class 0)}$$
$$w_1 \cdot 0 + w_2 \cdot 1 + b > 0 \quad \text{(class 1)}$$
$$w_1 \cdot 1 + w_2 \cdot 0 + b > 0 \quad \text{(class 1)}$$

$$w_1 \cdot 1 + w_2 \cdot 1 + b < 0 \qquad \text{(class 0)}$$

From the first inequality: $b < 0$ From the second: $w_2 + b > 0 \Rightarrow w_2 > -b > 0$ From the third: $w_1 + b > 0 \Rightarrow w_1 > -b > 0$ From the fourth: $w_1 + w_2 + b < 0$

But if $w_1 > 0$, $w_2 > 0$, and $b < 0$, then $w_1 + w_2 + b > 0$ (contradiction). Thus, no solution exists.

Two-Layer Network Solution

A two-layer network with one hidden layer can solve XOR because it can create **multiple linear boundaries** whose combination produces a non-linear decision boundary.

Architecture:

- Input layer: 2 neurons (x_1, x_2)
- Hidden layer: 2 neurons with activation functions
- Output layer: 1 neuron

Mathematical Formulation: Let the hidden layer compute:

$$h_1 = \sigma(w_{11}x_1 + w_{12}x_2 + b_1)$$

$$h_2 = \sigma(w_{21}x_1 + w_{22}x_2 + b_2)$$

The output computes:

$$y = \sigma(v_1h_1 + v_2h_2 + b_3)$$

Solution Strategy: We can design the network to compute XOR as: XOR(x_1, x_2) = AND(OR(x_1, x_2), NAND(x_1, x_2))

Detailed Implementation:

1. **Hidden neuron 1 computes OR:**

$$h_1 = \sigma(5x_1 + 5x_2 - 2)$$

For sigmoid activation $\sigma(z) = 1/(1 + e^{-z})$:

- $(0, 0)$: $5 \cdot 0 + 5 \cdot 0 - 2 = -2 \Rightarrow \sigma(-2) = 0.119 \approx 0$
- $(0, 1)$ or $(1, 0)$: $5 \cdot 1 - 2 = 3 \Rightarrow \sigma(3) = 0.953 \approx 1$
- $(1, 1)$: $5 \cdot 1 + 5 \cdot 1 - 2 = 8 \Rightarrow \sigma(8) = 0.9997 \approx 1$

2. **Hidden neuron 2 computes NAND:**

$$h_2 = \sigma(-5x_1 - 5x_2 + 7)$$

- $(0, 0)$: $-5 \cdot 0 - 5 \cdot 0 + 7 = 7 \Rightarrow \sigma(7) = 0.9991 \approx 1$
- $(0, 1)$ or $(1, 0)$: $-5 \cdot 1 + 7 = 2 \Rightarrow \sigma(2) = 0.8808 \approx 1$
- $(1, 1)$: $-5 \cdot 1 - 5 \cdot 1 + 7 = -3 \Rightarrow \sigma(-3) = 0.0474 \approx 0$

3. **Output neuron computes AND:**

$$y = \sigma(5h_1 + 5h_2 - 7)$$

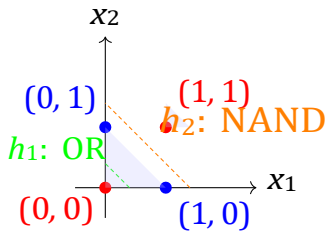
- $(0, 0)$: $h_1 = 0, h_2 = 1 \Rightarrow 5 \cdot 0 + 5 \cdot 1 - 7 = -2 \Rightarrow \sigma(-2) = 0.119 \approx 0$
- $(0, 1)$ or $(1, 0)$: $h_1 = 1, h_2 = 1 \Rightarrow 5 \cdot 1 + 5 \cdot 1 - 7 = 3 \Rightarrow \sigma(3) = 0.953 \approx 1$
- $(1, 1)$: $h_1 = 1, h_2 = 0 \Rightarrow 5 \cdot 1 + 5 \cdot 0 - 7 = -2 \Rightarrow \sigma(-2) = 0.119 \approx 0$

Geometric Interpretation: The hidden layer creates two decision boundaries:

- h_1 boundary: $5x_1 + 5x_2 - 2 = 0$ (line separating $(0, 0)$ from others)
- h_2 boundary: $-5x_1 - 5x_2 + 7 = 0$ (line separating $(1, 1)$ from others)

The output layer combines these to create a non-linear, convex region containing $(0, 1)$ and $(1, 0)$.

Visualization:



Theoretical Significance:

- **Universal Approximation Theorem:** A two-layer network with sufficient hidden neurons can approximate any continuous function
- **Importance of Depth:** Depth allows composition of features and creates hierarchical representations
- **Non-linear Transformations:** Activation functions enable non-linear combinations

2.11.3 Bloom's Level 3: Applying - Neural Network Design for Binary Classification

Question: Design a neural network for binary classification with 10 input features. Specify architecture, activation functions, loss function, and optimizer.

Detailed Answer:

Problem Analysis

Given: 10 input features, binary classification task. Requirements:

- Handle potential non-linear relationships
- Prevent overfitting
- Ensure efficient training
- Provide probabilistic outputs

Detailed Architecture Design Network Architecture:

$$\text{Input}(10) \rightarrow \text{Hidden}_1(16, \text{ReLU}) \rightarrow \text{Hidden}_2(8, \text{ReLU}) \rightarrow \text{Output}(1, \text{Sigmoid})$$

Layer-by-Layer Specification:

1. **Input Layer:**

- Size: 10 neurons (one per input feature)
- Purpose: Receives normalized/standardized input features
- Normalization: $\tilde{x}_i = \frac{x - \mu_i}{\sigma_i}$ for better convergence

2. **Hidden Layer 1:**

- Size: 16 neurons
- Activation: ReLU (Rectified Linear Unit)
- Mathematical form: $a^{(1)} = \max(0, z^{(1)})$ where $z^{(1)} = W^{(1)}x + b^{(1)}$
- Justification:

- Avoids vanishing gradient problem compared to sigmoid/tanh
- Computationally efficient (simple thresholding)
- Sparse activation (only 50% of neurons active)
- Biological plausibility

- Weight initialization: He initialization: $W_{ij} \sim \mathrm{N}(0, \sqrt{2/n_{\text{in}}})$

3. **Hidden Layer 2:**

- Size: 8 neurons (reducing dimensionality for abstraction)
- Activation: ReLU
- Dropout: 0.5 probability during training
- Mathematical form: $a^{(2)} = \text{Dropout}(\max(0, z^{(2)}), p = 0.5)$

4. **Output Layer:**

- Size: 1 neuron (binary classification)
- Activation: Sigmoid
- Mathematical form: $\hat{y} = \sigma(z^{(3)}) = \frac{1}{1+e^{-z^{(3)}}}$
- Output interpretation: $\hat{y} = P(y = 1|x)$ (probability of positive class)

- Decision threshold: Typically 0.5, can be tuned based on precision/recall tradeoff

Loss Function

Binary Cross-Entropy Loss:

$$L = -\frac{1}{N}\sum_{i=} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

Mathematical Derivation: Assuming $y_i \in \{0, 1\}$ follows Bernoulli distribution with probability $\hat{y}_i$:

$$P(y_i|\hat{y}_i) = \hat{y}_i^{y_i}(1 - \hat{y}_i)^{1-y_i}$$

Negative log-likelihood:

$$-\log P(y_i|\hat{y}_i) = -[y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)]$$

Properties:

- Convex with respect to parameters (for linear models)
- Penalizes confident wrong predictions heavily
- Gradient: $\frac{\partial L}{\partial \hat{y}_i} = \frac{\hat{y}_i - y_i}{\hat{y}_i(1-\hat{y}_i)}$
- Numerically stable implementation: Combine sigmoid with BCE loss

Optimizer Selection

Adam Optimizer:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

where:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Hyperparameters:

- Learning rate: $\eta = 0.001$ (common default)
- Momentum: $\beta_1 = 0.9$
- RMSprop term: $\beta_2 = 0.999$
- Numerical stability: $\epsilon = 10^{-8}$

Advantages:

- Combines benefits of RMSprop and momentum
- Adaptive learning rates per parameter
- Bias correction for initialization
- Works well with sparse gradients

Regularization Strategies

1. **L2 Regularization (Weight Decay):**

$$L_{\text{total}} = L_{\text{data}} + \frac{\lambda}{2}\|\mathbf{W}\|_2^2$$

with $\lambda = 0.01$ controlling regularization strength.

2. **Dropout:**

- Probability: $p = 0.5$ for hidden layers
- Implementation: Multiply activations by Bernoulli mask during training
- Scale by $1/(1-p)$ during training or $(1-p)$ during inference
- Effect: Prevents co-adaptation of features, acts as ensemble averaging

3. **Early Stopping:**

- Monitor validation loss
- Patience: 10-20 epochs without improvement
- Restore best weights when stopping

Training Procedure Data Preparation:

- Split: 70% train, 15% validation, 15% test
- Normalization: Standardize to mean 0, variance 1
- Class balancing: Check and apply weighting if imbalanced

Hyperparameter Tuning:

- Learning rate: Try [0.1, 0.01, 0.001, 0.0001]
- Hidden sizes: Try [16, 32, 64] for first layer, [8, 16, 32] for second
- Dropout rate: Try [0.3, 0.5, 0.7]
- L2 lambda: Try [0.001, 0.01, 0.1]

Monitoring Metrics:

- Training loss (for convergence)
- Validation loss (for early stopping)
- Accuracy, Precision, Recall, F1-score
- ROC curve and AUC

Performance Evaluation Confusion Matrix:

$$\begin{matrix} TN & FP \\ FN & TP \end{matrix}$$

Metrics:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

$$\text{F1-score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

$$\text{AUC-ROC} = \text{Area under ROC curve}$$

2.11.4 Bloom's Level 5: Evaluating - Network Architecture Com-parison

Question: Compare feedforward, convolutional, and recurrent networks for different tasks.

Detailed Answer:

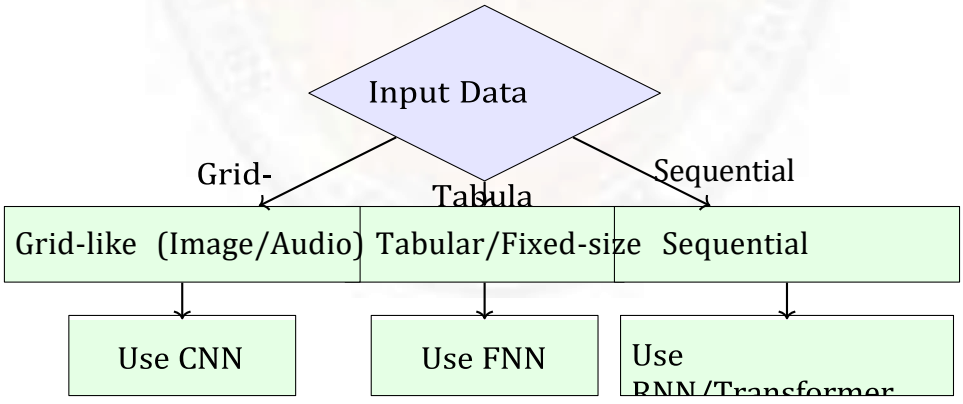


Figure 2.16: Architecture selection decision tree

Appendix - Introduction to Perceptron

The perceptron, invented by Frank Rosenblatt in 1958, is the fundamental building block of artificial neural networks. It is a mathematical model inspired by biological neurons that can learn to classify input patterns.

Key Characteristics

- **Single-layer**: Only one layer of processing
- **Binary output**: Produces either 0 or 1
- **Linear classifier**: Can separate linearly separable data
- **Supervised learning**: Learns from labeled training data

Mathematical Model of Perceptron

Architecture Diagram

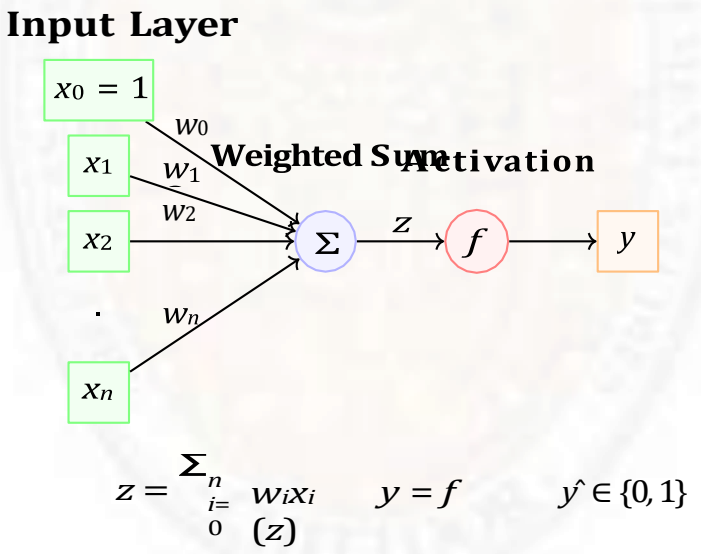


Figure 2.17: Architecture of a Single Perceptron

Mathematical Formulation

A perceptron computes its output as follows:

$$\hat{y} = f\left(\sum_{i=0}^{n} w_i x_i\right) \tag{2.1}$$

Where:

- $x_0 = 1$ is the **bias term** (always 1)
- $x_1, x_2, \ldots, x_n$ are input features
- $w_0, w_1, \ldots, w_n$ are weights (parameters)
- f is the activation function (typically step function)
- $\hat{y}$ is the predicted output

Vector Notation

In vector form, the perceptron can be written as:

$$\hat{y} = f(\mathbf{w}^T \mathbf{x}) \tag{2.2}$$

Where:

$$\mathbf{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \text{and} \quad \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

Activation Function: The Step Function

The perceptron typically uses a **step function** (also called Heaviside function):

$$f(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

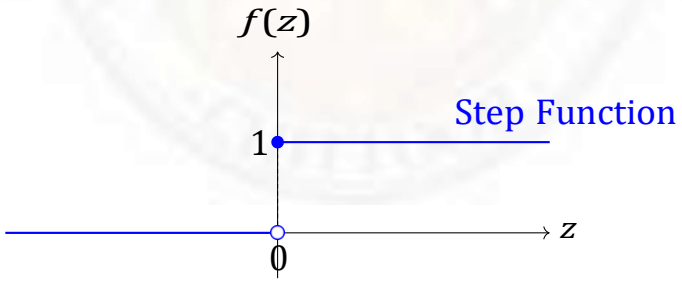


Figure 2.18: Step Activation Function

Perceptron Learning Algorithm

Training Objective

The perceptron learning algorithm aims to find weights $\mathbf{w}$ such that:

$$\hat{y} = \begin{cases} 1 & \text{if } \mathbf{w}^T\mathbf{x} > 0 \\ 0 & \text{if } \mathbf{w}^T\mathbf{x} \leq 0 \end{cases}$$

matches the true label y for all training examples.

Perceptron Learning Rule

The weights are updated using the following rule:

$$w_i \leftarrow w_i + \eta(y - \hat{y})x_i \tag{2.4}$$

Where:

- η is the **learning rate** ($0 < \eta \leq 1$)
- y is the true label (0 or 1)
- $\hat{y}$ is the predicted label
- x_i is the input feature

Algorithm Steps

1. **Initialize weights**: Set all weights w_i to small random values or zeros.
2. **For each training example** $(\mathbf{x}, y)$:
   (a) Compute output: $\hat{y} = f(\mathbf{w}^T\mathbf{x})$
   (b) Compute error: $\delta = y - \hat{y}$
   (c) Update weights: $w_i \leftarrow w_i + \eta\delta x_i$
3. **Repeat** until convergence (no errors or maximum iterations reached)

Detailed Example with Calculations

Problem Setup

Consider a perceptron with:

- Input features: x_1 and x_2 (plus bias $x_0 = 1$)

- Training data:

x_1	x_2	y (target)
0	0	0
0	1	1
1	0	1
1	1	1

Table 2.13: Training Data for OR Function

- Learning rate: $\eta = 0.1$
- Initial weights: $w_0 = 0.2$, $w_1 = -0.1$, $w_2 = 0.3$

First Training Iteration

Example 1: $(x_1, x_2) = (0, 0), y = 0$

1. Input vector: $\mathbf{x} = [1, 0, 0]^T$
2. Weighted sum:
$$z = w_0 \cdot 1 + w_1 \cdot 0 + w_2 \cdot 0 = 0.2$$
3. Activation:
$$\hat{y} = f(0.2) = 1 \quad \text{(since } 0.2 > 0\text{)}$$
4. Error:
$$\delta = y - \hat{y} = 0 - 1 = -1$$
5. Weight update:
$$w_0 \leftarrow 0.2 + 0.1 \times (-1) \times 1 = 0.1$$
$$w_1 \leftarrow -0.1 + 0.1 \times (-1) \times 0 = -0.1$$
$$w_2 \leftarrow 0.3 + 0.1 \times (-1) \times 0 = 0.3$$

Updated weights: $w_0 = 0.1$, $w_1 = -0.1$, $w_2 = 0.3$

Second Training Iteration

Example 2: $(x_1, x_2) = (0, 1), y = 1$

1. Input vector: $\mathbf{x} = [1, 0, 1]^T$
2. Weighted sum:
$$z = 0.1 \cdot 1 + (-0.1) \cdot 0 + 0.3 \cdot 1 = 0.4$$

3. Activation:

$$\hat{y} = f(0.4) = 1$$

4. Error:

$$\delta = 1 - 1 = 0$$

5. No weight update needed

Weights remain: $w_0 = 0.1, w_1 = -0.1, w_2 = 0.3$

Complete Training Epoch

Continuing this process for all examples, the perceptron will eventually converge to weights that correctly classify the OR function.

Decision Boundary and Geometric Interpretation

The perceptron creates a linear decision boundary in the feature space:

$$\mathbf{w}^T\mathbf{x} = w_0 + w_1x_1 + w_2x_2 = 0 \tag{2.5}$$

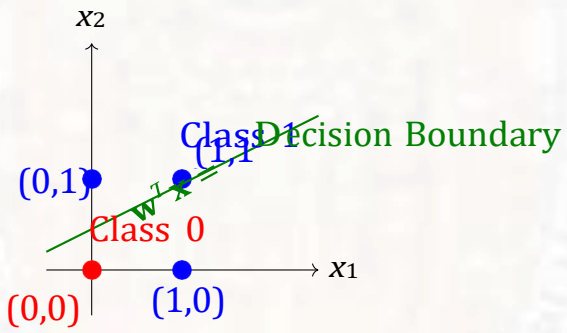


Figure 2.19: Perceptron Decision Boundary for OR Function

Convergence Theorem

Perceptron Convergence Theorem

Theorem: If the training data is linearly separable, the perceptron learning algorithm will converge to a solution in a finite number of steps.

Proof Sketch

1. Assume there exists a weight vector $\mathbf{w}^*$ that correctly classifies all examples
2. Show that the angle between $\mathbf{w}$ and $\mathbf{w}^*$ decreases with each mistake
3. Bound the number of possible mistakes

Limitations of Perceptron

XOR Problem

The perceptron cannot learn the XOR function because it's not linearly separable:

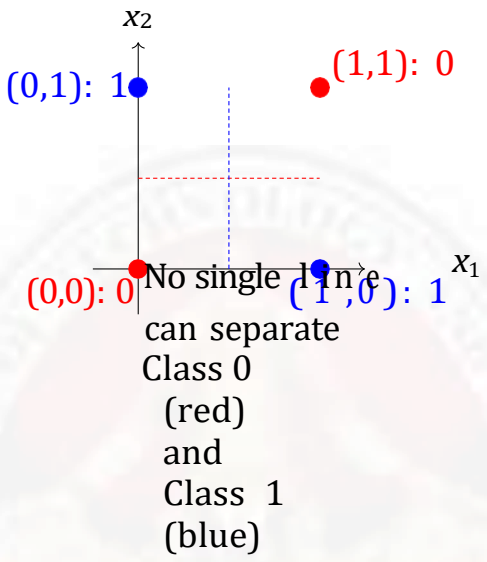


Figure 2.20: XOR Problem - Not Linearly Separable

Mathematical Proof of XOR Limitation

For a perceptron to learn XOR, we need weights satisfying:

$$w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0 \qquad \text{(for } (0,0) \to 0\text{)}$$
$$w_0 + w_1 \cdot 0 + w_2 \cdot 1 > 0 \qquad \text{(for } (0,1) \to 1\text{)}$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 0 > 0 \qquad \text{(for } (1,0) \to 1\text{)}$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 1 < 0 \qquad \text{(for } (1,1) \to 0\text{)}$$

From inequalities 2 and 3: $w_0 + w_2 > 0$ and $w_0 + w_1 > 0$ Adding these: $2w_0 + w_1 + w_2 > 0$
But from inequalities 1 and 4: $w_0 < 0$ and $w_0 + w_1 + w_2 < 0$ This creates a contradiction, proving no solution exists.

Practical Implementation Details

Pseudo-code

```
function train_perceptron(X, y, learning_rate, max_epochs):
Initialize weights w
```

randomly for epoch in
range(max_epochs): errors
= 0
for each sample (x_i, y_i) in
training set: # Compute prediction

```
z = dot_product(w,
x_i) y_pred =
step_function(z)

# Compute error
error = y_i -
y_pred

# Update weights if
error if error != 0:
for each weight w_j:
w_j = w_j + learning_rate * error *
x_i[j]  errors += 1

# Stop if
converged if
errors == 0:
break

return

w
```

Learning Rate Selection

- **Fixed learning rate**: Constant η throughout training
- **Adaptive learning rate**: η decreases over time
- Typical values: $0.01 \leq \eta \leq 0.1$

Applications of Perceptron

- **Binary classification**: Spam detection, sentiment analysis
- **Pattern recognition**: Simple image classification
- **Medical diagnosis**: Disease prediction from symptoms
- **Building block**: Basis for multi-layer neural networks

Summary

The perceptron is a simple yet powerful algorithm with these key properties:

Aspect	Description
Type	Single-layer neural network
Output	Binary (0 or 1)
Activation	Step function
Learning	Supervised, online learning
Decision boundary	Linear
Convergence	Guaranteed for linearly separable data
Limitation	Cannot solve non-linear problems like XOR

Table 2.14: Summary of Perceptron Properties

Key Formulas Recap

1. **Output**: $\hat{y} = f(\mathbf{w}^T\mathbf{x})$
2. **Step function**: $f(z) = \begin{cases} 1 & z > 0 \\ 0 & \text{otherwise} \end{cases}$
3. **Weight update**: $w_i \leftarrow w_i + \eta(y - \hat{y})x_i$
4. **Decision boundary**: $\mathbf{w}^T\mathbf{x} = 0$

The perceptron, despite its limitations, remains historically significant as it laid the foundation for modern neural networks and deep learning.

2.12 Essay Questions: Comprehensive Explanations

2.12.1 Bloom's Level 1: Remembering - Bias-Variance Decompo-sition

Question: List and define the three components of the bias-variance decomposition for mean squared error.

Detailed Answer:

The bias-variance decomposition is a fundamental theoretical framework in machine learning that provides deep insight into the sources of error in predictive models. For a given prediction point x, let $y = f(x) + \epsilon$ be the true relationship, where $f(x)$ is the under-lying true function and $\epsilon(0, \sigma^2)$ is irreducible noise. Our model produces predictions $\hat{f}(x)$ based on training data.

The expected prediction error for mean squared error (MSE) decomposes into three distinct components:

1. **Bias Squared (Systematic Error):**

$$\text{Bias}^2(\hat{f}(x)) = (\text{E}[\hat{f}(x)] - f(x))^2$$

Interpretation: Bias measures how much the average prediction of our model differs from the true value. It represents systematic error that occurs regardless of the specific training data used. High bias typically indicates:

- **Underfitting:** The model is too simple to capture the underlying patterns
- **Incorrect assumptions:** The model family doesn't include the true function
- **Consistent deviation:** The model makes the same type of error repeatedly

Example: A linear model trying to fit quadratic data will have high bias because no straight line can accurately represent a parabola.

2. **Variance (Random Error):**

$$\text{Var}(\hat{f}(x)) = \text{E}[(\hat{f}(x) - \text{E}[\hat{f}(x)])^2]$$

Interpretation: Variance measures how much the predictions vary for different train-ing datasets. It represents the model's sensitivity to the particular training data used. High variance typically indicates:

- **Overfitting:** The model is too complex and memorizes noise
- **Instability:** Small changes in training data lead to large changes in predictions
- **Poor generalization:** Excellent performance on training data but poor on test data

Example: A very high-degree polynomial that perfectly fits every training point but oscillates wildly between them.

3. **Irreducible Error (Noise):**

$$\sigma^2 = \text{Var}(\epsilon)$$

Interpretation: Irreducible error represents the inherent noise in the data generation process. This error cannot be reduced by any model, no matter how sophisticated. Sources include:

- **Measurement error:** Imperfect instruments or data collection
- **Stochastic processes:** Inherent randomness in the phenomenon
- **Missing information:** Unobserved variables affecting the outcome

Example: In weather prediction, even with perfect models, there will always be some uncertainty due to chaotic atmospheric dynamics.

Complete Decomposition:

$$\text{E}[(y - \hat{f}(x))^2] = \underbrace{(\text{E}[\hat{f}(x)] - f(x))^2}_{\text{Bias}^2} + \underbrace{\text{E}[(\hat{f}(x) - \text{E}[\hat{f}(x)])^2]}_{\text{Variance}} + \underbrace{\sigma^2}_{\text{Irreducible Error}}$$

Mathematical Derivation:

$$
\begin{aligned}
\mathrm{E}[(y - \hat{f})^2] &= \mathrm{E}[(f + \epsilon - \hat{f})^2] \\
&= \mathrm{E}[(f - \hat{f})^2] + \mathrm{E}[\epsilon^2] + 2\mathrm{E}[\epsilon(f - \hat{f})] \\
&= \mathrm{E}[(f - \mathrm{E}[\hat{f}] + \mathrm{E}[\hat{f}] - \hat{f})^2] + \sigma^2 \\
&= \mathrm{E}[(f - \mathrm{E}[\hat{f}])^2] + 2\mathrm{E}[(f - \mathrm{E}[\hat{f}])(\mathrm{E}[\hat{f}] - \hat{f})] \\
&\quad + \mathrm{E}[(\mathrm{E}[\hat{f}] - \hat{f})^2] + \sigma^2 \\
&= (f - \mathrm{E}[\hat{f}])^2 + 0 + \mathrm{E}[(\mathrm{E}[\hat{f}] - \hat{f})^2] + \sigma^2 \\
&= \mathrm{Bias}^2 + \mathrm{Variance} + \sigma^2
\end{aligned}
$$

Practical Implications:

- **Bias-Variance Tradeoff:** Decreasing bias typically increases variance, and vice versa
- **Model Selection:** Choose model complexity to balance these components
- **Regularization:** Techniques like L1/L2 regularization reduce variance at cost of in-creased bias
- **Ensemble Methods:** Bagging reduces variance, boosting reduces bias

2.12.2 Bloom's Level 2: Understanding - Single-Layer vs.Two-Layer Networks for XOR

Question: Explain why a single-layer perceptron cannot solve XOR while a two-layer net-work can.

Detailed Answer:

Mathematical Foundation

The XOR (exclusive OR) function is defined as:

$$
\mathrm{XOR}(x_1, x_2) = \begin{cases} 0 & \text{if } x_1 = x_2 \\ 1 & \text{if } x_1 /= x_2 \end{cases}
$$

The truth table is:

x_1	x_2	XOR(x_1, x_2)
0	0	0
0	1	1
1	0	1
1	1	0

Single-Layer Perceptron Limitation

A single-layer perceptron implements:

$$y = \text{sign}(\mathbf{w}^\top \mathbf{x} + b) = \text{sign}(w_1 x_1 + w_2 x_2 + b)$$

This creates a **linear decision boundary** in the input space. The condition $w_1 x_1 + w_2 x_2 + b = 0$ defines a straight line that separates points into two classes.

Geometric Proof of Impossibility: Plot the XOR points in 2D space: (0, 0) and (1, 1) are class 0, (0, 1) and (1, 0) are class 1. No single straight line can separate these points such that all class 0 points are on one side and all class 1 points on the other.

Algebraic Proof: Assume there exist w_1, w_2, b such that:

$$\begin{aligned} w_1 \cdot 0 + w_2 \cdot 0 + b < 0 \quad & \text{(class 0)} \\ w_1 \cdot 0 + w_2 \cdot 1 + b > 0 \quad & \text{(class 1)} \\ w_1 \cdot 1 + w_2 \cdot 0 + b > 0 \quad & \text{(class 1)} \\ w_1 \cdot 1 + w_2 \cdot 1 + b < 0 \quad & \text{(class 0)} \end{aligned}$$

From the first inequality: $b < 0$ From the second: $w_2 + b > 0 \Rightarrow w_2 > -b > 0$ From the third: $w_1 + b > 0 \Rightarrow w_1 > -b > 0$ From the fourth: $w_1 + w_2 + b < 0$

But if $w_1 > 0$, $w_2 > 0$, and $b < 0$, then $w_1 + w_2 + b > 0$ (contradiction). Thus, no solution exists.

Two-Layer Network Solution

A two-layer network with one hidden layer can solve XOR because it can create **multiple linear boundaries** whose combination produces a non-linear decision boundary.

Architecture:

- Input layer: 2 neurons (x_1, x_2)
- Hidden layer: 2 neurons with activation functions
- Output layer: 1 neuron

Mathematical Formulation: Let the hidden layer compute:

$$h_1 = \sigma(w_{11} x_1 + w_{12} x_2 + b_1)$$

$$h_2 = \sigma(w_{21} x_1 + w_{22} x_2 + b_2)$$

The output computes:

$$y = \sigma(v_1 h_1 + v_2 h_2 + b_3)$$

Solution Strategy: We can design the network to compute XOR as: XOR(x_1, x_2) = AND(OR(x_1, x_2), NAND(x_1, x_2))

Detailed Implementation:

1. **Hidden neuron 1 computes OR:**

$$h_1 = \sigma(5x_1 + 5x_2 - 2)$$

For sigmoid activation $\sigma(z) = 1/(1 + e^{-z})$:

- $(0, 0)$: $5 \cdot 0 + 5 \cdot 0 - 2 = -2 \Rightarrow \sigma(-2) = 0.119 \approx 0$
- $(0, 1)$ or $(1, 0)$: $5 \cdot 1 - 2 = 3 \Rightarrow \sigma(3) = 0.953 \approx 1$
- $(1, 1)$: $5 \cdot 1 + 5 \cdot 1 - 2 = 8 \Rightarrow \sigma(8) = 0.9997 \approx 1$

2. **Hidden neuron 2 computes NAND:**

$$h_2 = \sigma(-5x_1 - 5x_2 + 7)$$

- $(0, 0)$: $-5 \cdot 0 - 5 \cdot 0 + 7 = 7 \Rightarrow \sigma(7) = 0.9991 \approx 1$
- $(0, 1)$ or $(1, 0)$: $-5 \cdot 1 + 7 = 2 \Rightarrow \sigma(2) = 0.8808 \approx 1$
- $(1, 1)$: $-5 \cdot 1 - 5 \cdot 1 + 7 = -3 \Rightarrow \sigma(-3) = 0.0474 \approx 0$

3. **Output neuron computes AND:**

$$y = \sigma(5h_1 + 5h_2 - 7)$$

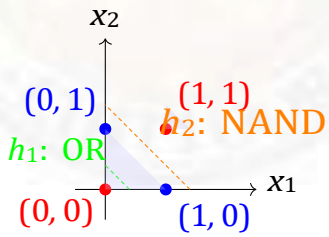
- $(0, 0)$: $h_1 = 0, h_2 = 1 \Rightarrow 5 \cdot 0 + 5 \cdot 1 - 7 = -2 \Rightarrow \sigma(-2) = 0.119 \approx 0$
- $(0, 1)$ or $(1, 0)$: $h_1 = 1, h_2 = 1 \Rightarrow 5 \cdot 1 + 5 \cdot 1 - 7 = 3 \Rightarrow \sigma(3) = 0.953 \approx 1$
- $(1, 1)$: $h_1 = 1, h_2 = 0 \Rightarrow 5 \cdot 1 + 5 \cdot 0 - 7 = -2 \Rightarrow \sigma(-2) = 0.119 \approx 0$

Geometric Interpretation: The hidden layer creates two decision boundaries:

- h_1 boundary: $5x_1 + 5x_2 - 2 = 0$ (line separating $(0, 0)$ from others)
- h_2 boundary: $-5x_1 - 5x_2 + 7 = 0$ (line separating $(1, 1)$ from others)

The output layer combines these to create a non-linear, convex region containing $(0, 1)$ and $(1, 0)$.

Visualization:



Theoretical Significance:

- **Universal Approximation Theorem:** A two-layer network with sufficient hidden neurons can approximate any continuous function
- **Importance of Depth:** Depth allows composition of features and creates hierarchical representations
- **Non-linear Transformations:** Activation functions enable non-linear combinations

2.12.3 Bloom's Level 3: Applying - Neural Network Design for Binary Classification

Question: Design a neural network for binary classification with 10 input features. Specify architecture, activation functions, loss function, and optimizer.

Detailed Answer:

Problem Analysis

Given: 10 input features, binary classification task. Requirements:

- Handle potential non-linear relationships
- Prevent overfitting
- Ensure efficient training
- Provide probabilistic outputs

Detailed Architecture

Design Network

Architecture:

$$\text{Input}(10) \rightarrow \text{Hidden}_1(16, \text{ReLU}) \rightarrow \text{Hidden}_2(8, \text{ReLU}) \rightarrow \text{Output}(1, \text{Sigmoid})$$

Layer-by-Layer Specification:

1. **Input Layer:**

- Size: 10 neurons (one per input feature)
- Purpose: Receives normalized/standardized input features
- Normalization: $\tilde{x}_i = \frac{x - \mu_i}{\sigma_i}$ for better convergence

2. **Hidden Layer 1:**

- Size: 16 neurons
- Activation: ReLU (Rectified Linear Unit)
- Mathematical form: $a^{(1)} = \max(0, z^{(1)})$ where $z^{(1)} = W^{(1)}x + b^{(1)}$
- Justification:
  - Avoids vanishing gradient problem compared to sigmoid/tanh
  - Computationally efficient (simple thresholding)
  - Sparse activation (only 50% of neurons active)
  - Biological plausibility
- Weight initialization: He initialization: $W_{ij} \sim \mathrm{N}(0, \sqrt{2/n_{\text{in}}})$

3. **Hidden Layer 2:**

- Size: 8 neurons (reducing dimensionality for abstraction)
- Activation: ReLU
- Dropout: 0.5 probability during training
- Mathematical form: $a^{(2)} = \text{Dropout}(\max(0, z^{(2)}), p = 0.5)$

4. **Output Layer:**

- Size: 1 neuron (binary classification)
- Activation: Sigmoid
- Mathematical form: $\hat{y} = \sigma(z^{(3)}) = \frac{1}{1+e^{-z^{(3)}}}$
- Output interpretation: $\hat{y} = P(y = 1|x)$ (probability of positive class)
- Decision threshold: Typically 0.5, can be tuned based on precision/recall tradeoff

Loss Function

Binary Cross-Entropy Loss:

$$L = -\frac{1}{N}\sum_{i=} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

Mathematical Derivation: Assuming $y_i \in \{0, 1\}$ follows Bernoulli distribution with probability $\hat{y}_i$:

likelihood:

Negative log-

$P(y_i|\hat{y}_i) = \hat{y}^{y_i}(1 - \hat{y}_i)^{1-y_i}$

$$-\log P(y_i|\hat{y}_i) = -[y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)]$$

Properties:

- Convex with respect to parameters (for linear models)
- Penalizes confident wrong predictions heavily
- Gradient: $\frac{\partial L}{\partial \hat{y}_i} = \frac{\hat{y}_i - y_i}{\hat{y}_i(1-\hat{y}_i)}$
- Numerically stable implementation: Combine sigmoid with BCE loss

Optimizer Selection

Adam Optimizer:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

where:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Hyperparameters:

- Learning rate: $\eta = 0.001$ (common default)
- Momentum: $\beta_1 = 0.9$
- RMSprop term: $\beta_2 = 0.999$
- Numerical stability: $\epsilon = 10^{-8}$

Advantages:

- Combines benefits of RMSprop and momentum
- Adaptive learning rates per parameter
- Bias correction for initialization
- Works well with sparse gradients

Regularization Strategies

1. **L2 Regularization (Weight Decay):**

$$L_{\text{total}} = L_{\text{data}} + \frac{\lambda}{2} \|\mathbf{W}\|_2^2$$

with $\lambda = 0.01$ controlling regularization strength.

2. **Dropout:**

- Probability: $p = 0.5$ for hidden layers
- Implementation: Multiply activations by Bernoulli mask during training
- Scale by $1/(1-p)$ during training or $(1-p)$ during inference
- Effect: Prevents co-adaptation of features, acts as ensemble averaging

3. **Early Stopping:**

- Monitor validation loss

- Patience: 10-20 epochs without improvement
- Restore best weights when stopping

Training Procedure Data Preparation:

- Split: 70% train, 15% validation, 15% test
- Normalization: Standardize to mean 0, variance 1
- Class balancing: Check and apply weighting if imbalanced

Hyperparameter Tuning:

- Learning rate: Try [0.1, 0.01, 0.001, 0.0001]
- Hidden sizes: Try [16, 32, 64] for first layer, [8, 16, 32] for second
- Dropout rate: Try [0.3, 0.5, 0.7]
- L2 lambda: Try [0.001, 0.01, 0.1]

Monitoring Metrics:

- Training loss (for convergence)
- Validation loss (for early stopping)
- Accuracy, Precision, Recall, F1-score
- ROC curve and AUC

Performance Evaluation

Confusion Matrix:

$$\begin{matrix} TN & FP \\ FN & TP \end{matrix}$$

Metrics:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

$$\text{F1-score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

$$\text{AUC-ROC} = \text{Area under ROC curve}$$

2.12.4 Bloom's Level 4:Analyzing - Activation Functions and Gradient Flow

Question: Analyze how different activation functions affect gradient flow in deep networks.

Detailed Answer:

The Gradient Flow Problem

In deep neural networks, gradients must propagate through multiple layers during backprop-agation. The chain rule governs this process:

$$\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial a^{(k+1)}} \prod_{k=} \frac{\partial a^{(k+1)}}{\partial a^{(k}} \frac{\partial a^{(l)}}{\partial W^{(l)}}$$

The critical term is the product of Jacobians:

$$\prod_{k=l}^{L-1} \frac{\partial a^{(k+1}}{\partial a^{(k)}} = \prod_{k=l}^{L-1} \text{diag}(\sigma'(z^{(k)}))W^{(k+1)}$$

If $|\sigma'(z)$ is consistently < 1, gradients vanish exponentially. If $\sigma'(z)$ is consistently > 1, gradients explode.

Sigmoid

Activation $\frac{}{1+e}$

Function: $\sigma(x) = {}^{1}{}_{-x}$

Derivative: $\sigma'(x) = \sigma(x)(1 \quad \sigma(x))$

Range: $(0, 1)$ for output, $(0, 0.25]$ for derivative

Gradient Analysis:

- **Vanishing Gradient Problem:** For $|x| > 4$, $\sigma'(x) \approx 0$
- Maximum gradient at $x = 0$: $\sigma'(0) = 0.25$
- Product of n sigmoid derivatives: $\leq (0.25)^n \to 0$ rapidly
- **Saturation:** Neurons become "saturated" and stop learning

Mathematical Example: For a 10-layer network with sigmoid activations:

$$\prod_{i=1}^{10} \sigma'(z_i) \leq (0.25)^{10} \approx 9.5 \times 10^{-7}$$

Gradients become numerically zero.

Additional Issues:

- **Non-zero centered:** Outputs always positive, causing zig-zag optimization
- **Computationally expensive:** Requires exponentiation

Tanh Activation

Function: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

Derivative: $\tanh'(x) = 1 - \tanh^2(x)$
Range: $(-1, 1)$ for output, $(0, 1]$ for derivative

Improvements over Sigmoid:

- **Zero-centered:** Outputs symmetric around 0
- **Larger gradients:** Maximum derivative is 1 (vs 0.25 for sigmoid)
- **Faster convergence:** Less zig-zag in gradient descent

Limitations:

- Still suffers from vanishing gradients for large $|x|$
- $\tanh'(x) \approx 0$ for $|x| > 3$
- Saturation problem persists

ReLU Activation

Function: $\text{ReLU}(x) = \max(0, x)$

Derivative: $\text{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$

Advantages:

- **No vanishing gradient for positive inputs:** Gradient = 1
- **Computationally efficient:** Simple thresholding
- **Sparse activation:** Only 50% of neurons active
- **Biological plausibility:** Similar to neural firing thresholds

Problems:

1. **Dying ReLU Problem:**
   - If gradient is large and negative, weights can update such that $z \leq 0$ for all inputs
   - Once "dead," neuron never activates again: $\text{ReLU}'(x) = 0$
   - No gradient flow through dead neurons
   - Can lose up to 40% of neurons in deep networks
2. **Unbounded output:** Can cause numerical instability
3. **Not differentiable at 0:** Though rarely an issue in practice

Leaky ReLU and Variants

Leaky ReLU: $\text{LReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$

Derivative: $\text{LReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ \alpha & \text{if } x \leq 0 \end{cases}$

Parameterized ReLU (PReLU): α learned during training

Randomized ReLU (RReLU): $\sim \alpha \quad U(l, u)$ during training, fixed during testing

Advantages:

- **Solves dying ReLU:** Small gradient (α) for negative inputs
- **Maintains sparse activation:** Most neurons still inactive
- **Improves performance:** Especially for deep networks

Typical values: $\alpha = 0.01$ for Leaky ReLU

Exponential Linear Unit (ELU)

Function: $\text{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$

Derivative: $\text{ELU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ \text{ELU}(x) + \alpha & \text{if } x \leq 0 \end{cases}$

Properties:

- Smooth transition at 0 (fully differentiable)
- Negative saturation value $-\alpha$ (vs $-\infty$ for ReLU)
- "Self-normalizing" property helps with batch normalization

Swish Activation

Function: $\text{Swish}(x) = x \cdot \sigma(\beta x)$

Derivative: $\text{Swish}'(x) = \text{Swish}(x) + \sigma(\beta x)(1 - \text{Swish}(x))$

Properties:

- Smooth, non-monotonic function
- Outperforms ReLU on many deep networks
- Default: $\beta = 1$ (learnable parameter)
- Small negative values for negative inputs

Gradient Flow Comparison Practical Recommendations For shallow networks:

- ReLU is usually sufficient
- Simple, fast, works well

For deep networks:

- Leaky ReLU or ELU to prevent dying neurons
- Swish for transformer architectures

For recurrent networks:

- Tanh or Sigmoid for gating mechanisms
- Hard sigmoid for efficiency

Gradient Clipping: For ReLU networks, clip gradients to prevent explosion:

$$g \leftarrow g \cdot \min \quad 1, \frac{\tau}{\|g\|}$$

Initialization: Use appropriate initialization (He for ReLU, Xavier for Tanh)

Empirical Results Image Classification:

- ResNet: ReLU works well with residual connections
- EfficientNet: Swish activation
- MobileNet: ReLU6 (clipped at 6) for quantization

Natural Language Processing:

- Transformers: GELU (Gaussian Error Linear Unit)
- LSTMs: Tanh and Sigmoid for gates

Mathematical Analysis of Gradient Preservation

Condition Number Analysis: The condition number of the Jacobian product affects gradient flow:

$$\kappa\left(\prod_{k} J_k\right) \leq \prod_{k} \kappa(J_k)$$

where $J_k = \text{diag}(\sigma'(z^{(k)}))W^{(k+1)}$.

For ReLU: $\sigma'(z) \in \{0, 1\}$, so condition number depends on W For sigmoid: $\sigma'(z) \approx 0$ for saturated neurons, increasing $\kappa(J_k)$

Solution: Residual connections help by providing identity paths:

$$y = F(x, W) + x$$

Gradient: $\frac{\partial y}{\partial x} = \frac{\partial F}{\partial x} + 1$

2.12.5 Bloom's Level 5: Evaluating - Network Architecture Com-parison

Question: Compare feedforward, convolutional, and recurrent networks for different tasks.

Detailed Answer:

Taxonomy of Neural Network Architectures

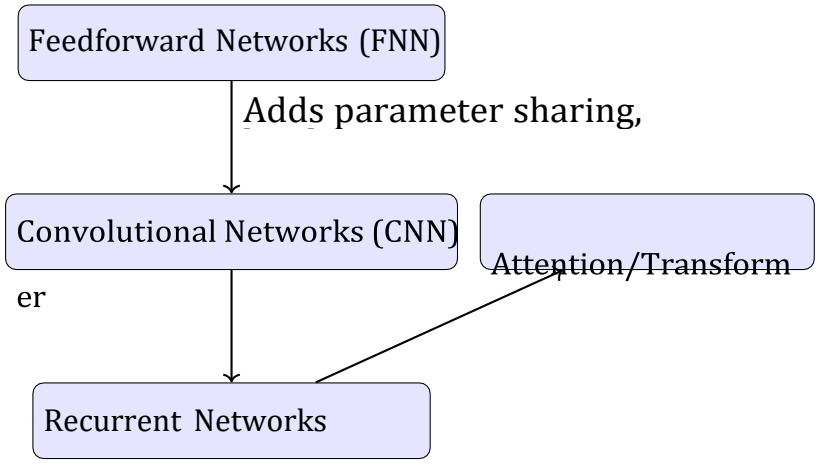


Figure 2.21: Evolution of neural network architectures

Feedforward Neural Networks (FNN)

Architecture:

$$y = f_L(\cdots f_2(f_1(x; W_1); W_2) \cdots ; W_L)$$

where $f_l(z) = \sigma(W_l z + b_l)$

Key Characteristics:

- **Fully connected:** Each neuron connects to all neurons in next layer
- **Fixed-size input:** Input dimension must be constant
- **No spatial/temporal structure:** Treats all input dimensions equally
- **Maximum flexibility:** Can learn arbitrary mappings (Universal Approximation The-orem)

Strengths:

1. **Universal approximation:** Can represent any continuous function given enough neurons
2. **Simple implementation:** Easy to code and understand
3. **Works with any data type:** As long as it can be vectorized
4. **No assumptions about data structure:** Makes minimal prior assumptions

Weaknesses:

1. **Parameter explosion:** $O(n^2)$ parameters per layer
2. **No parameter sharing:** Each weight used only once
3. **No translation invariance:** Must relearn patterns at different positions
4. **No sequential processing:** Cannot handle variable-length sequences

Best Applications:

- **Tabular data:** Customer churn prediction, credit scoring
- **Structured data:** Where features have no spatial/temporal relationships
- **Small datasets:** Where overfitting is a concern with more complex models
- **Simple classification/regression:** When data has no inherent structure

Specific Examples:

- Medical diagnosis from patient measurements
- Fraud detection from transaction features
- Housing price prediction from property attributes

Convolutional Neural Networks (CNN)

Architecture:

$$y = \text{Pool}(\sigma(\text{Conv}(x, W) + b))$$

Key Operations:

1. **Convolution:** $z_{i,j,k} = \sum_{m,n,c} x_{i+m,j+n,c} \cdot w_{m,n,c,k}$
2. **Pooling:** $p_{i,j,k} = \max_{m,n \in \text{window}} z_{i+m,j+n,k}$
3. **Striding:** Skip positions to reduce dimensionality
4. **Dilation:** Increase receptive field without additional parameters

Strengths:

1. **Parameter sharing:** Same filter applied across spatial positions
2. **Translation invariance:** Learns patterns regardless of position
3. **Hierarchical feature learning:** Low-level → mid-level → high-level features
4. **Sparse connectivity:** Each neuron connects only to local region
5. **Excellent for grid-like data:** Images, spectrograms, etc.

Weaknesses:

1. **Fixed receptive field:** Limited context understanding
2. **Inefficient for sequential data:** No built-in temporal modeling
3. **Computationally intensive:** Many operations despite parameter sharing
4. **Padding artifacts:** Boundary effects in convolutions

Variants and Extensions:

- **Dilated CNNs:** For larger receptive fields
- **Depthwise Separable Convolutions:** For efficiency (MobileNet)
- **Deformable Convolutions:** Learnable sampling locations
- **Graph CNNs:** For non-Euclidean data

Best Applications:

- **Computer Vision:**
  - Image classification (ResNet, VGG, EfficientNet)
  - Object detection (YOLO, Faster R-CNN)

- Semantic segmentation (U-Net, DeepLab)
- Image generation (StyleGAN)

- **Audio Processing:**
  - Speech recognition (spectrogram analysis)
  - Music classification
- **Medical Imaging:**
  - Tumor detection in MRI/CT scans
  - Cell segmentation in microscopy
- **Time Series Analysis:** When treated as 1D signals

Recurrent Neural Networks (RNN)

Architecture:

$$h_t = \sigma(W_h h_{t-1} + W_x x_t + b_h)$$
$$y_t = \sigma(W_y h_t + b_y)$$

Key Characteristics:

- **Sequential processing:** Processes one element at a time
- **Variable-length input:** Can handle sequences of different lengths
- **State/memory:** Hidden state captures information from previous steps
- **Parameter sharing across time:** Same weights applied at each time step

Variants:

1. **Simple RNN:** Basic formulation, suffers from vanishing gradients
2. **LSTM (Long Short-Term Memory):**

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$$
$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C[h_{t-1}, x_t] + b_C)$$
$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$
$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$
$$h_t = o_t \odot \tanh(C_t)$$

3. **GRU (Gated Recurrent Unit):** Simplified version of LSTM
4. **Bidirectional RNN:** Processes sequence in both directions

---

Strengths:

1. **Handles variable-length sequences:** Natural for text, speech, time series
2. **Temporal dependencies:** Captures relationships across time
3. **Memory:** Can remember information from earlier in sequence
4. **Parameter efficiency:** Same parameters across all time steps

Weaknesses:

1. **Sequential processing:** Cannot parallelize across time steps
2. **Vanishing/exploding gradients:** Especially in simple RNNs
3. **Short-term memory:** Limited capacity for long-range dependencies
4. **Computationally slow:** Must process sequence step-by-step

Best Applications:

- **Natural Language Processing:**
  - Machine translation (before transformers)
  - Text generation
  - Sentiment analysis
- **Time Series Analysis:**
  - Stock price prediction
  - Weather forecasting
  - Sensor data analysis
- **Speech Processing:**
  - Speech recognition
  - Speaker identification
- **Video Analysis:** Temporal sequence of frames

Comparative Analysis Hybrid Architectures

CNN + RNN: For video captioning

- CNN extracts spatial features from each frame
- RNN processes temporal sequence of features
- Output: Natural language description

Example: "A person is playing guitar" from video frames

RNN + Attention: For machine translation

- Encoder RNN processes source sentence
- Attention mechanism aligns words
- Decoder RNN generates target sentence

Emerging Architectures

Transformers: Self-attention based

- **Advantages:** Full parallelization, long-range dependencies
- **Applications:** BERT (NLP), Vision Transformers (CV)

Graph Neural Networks: For relational data

- **Applications:** Social networks, molecular structures

Decision Framework for Architecture Selection

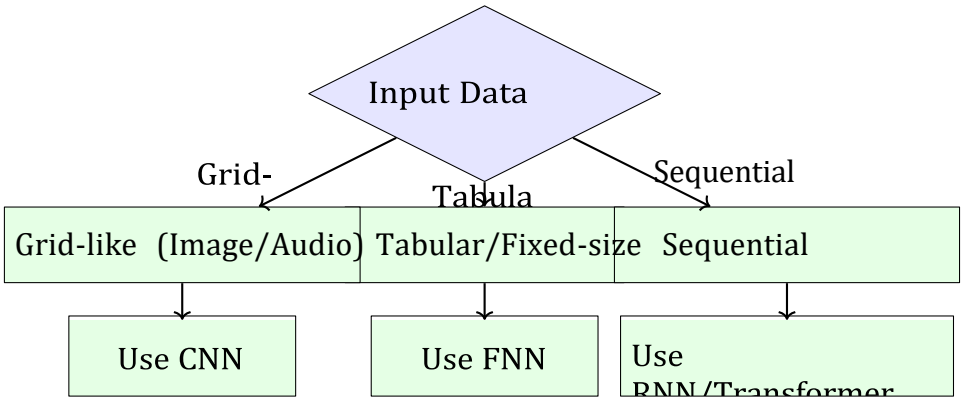


Figure 2.22: Architecture selection decision tree

Additional Considerations:

1. **Data Availability:**

- Large data: Complex architectures (CNN, Transformers)
- Small data: Simple architectures (FNN, shallow networks)

2. **Computational Resources:**

- Limited: FNN or small CNNs
- Abundant: Deep CNNs, Transformers

3. **Real-time Requirements:**

- Real-time: Efficient architectures (MobileNet, Lite RNN)
- Offline: Can use more complex models

4. **Interpretability Needs:**

- High interpretability: Simpler models, attention maps
- Black-box acceptable: Any architecture

Future Trends

Unified Architectures: Models that can handle multiple data types **Neuromorphic Computing:** Hardware-software co-design **Continual Learning:** Architectures that learn continuously without forgetting **Explainable AI:** Architectures with built-in interpretabil-ity

Conclusion

Each architecture represents a different inductive bias about the data:

- **FNN:** Assumes all features interact equally
- **CNN:** Assumes spatial/local correlations
- **RNN:** Assumes temporal/sequential dependencies

The choice should be guided by: 1. Data characteristics and structure 2. Task require-ments and constraints 3. Available computational resources 4. Need for interpretability and deployment considerations

Modern deep learning increasingly uses hybrid and transformer-based architectures that combine strengths of multiple approaches while addressing their limitations.

Table 2.1: Comparison of Model Fitting Behaviors

Aspect	Optimal Fitting	Overfitting	Underfitting
Definition	Model captures the underlying pattern of the training data while generalizing well to unseen data.	Model learns training data noise/fluctuations, resulting in poor generalization to new data.	Model fails to capture the underlying pattern of the training data, being too simplistic.
Model Complexity	Appropriate for the true complexity of the data.	Excessively high (too many parameters/features).	Too low (too few parameters/features).
Training Error	Low (but not necessarily minimal).	Very low (often near zero).	High.
Validation/Test Error	Low (similar to training error).	Significantly higher than training error.	High (similar to training error).
Variance-Bias Tradeoff	Balanced bias and variance.	High variance, low bias.	High bias, low variance.
Visual Analogy (for regression)	Curve follows the overall trend of data points.	Curve passes through almost every data point, oscillating wildly.	Straight/very smooth line ignoring data patterns.
Primary Cause	Correct model specification and sufficient, representative data.	Training too long on noisy data, excessive model capacity without regularization.	Insufficient model capacity, inadequate training, or overly simplistic features.

Common Remedies	Cross-validation, early stopping, regularization, collecting more data.	Regularization (L1/L2), dropout (for neural networks), pruning (for trees), reducing features.	Increase model com-plexity, add relevant features, train longer, reduce regularization.

Table 2.2: Components of the Bias-Variance Decomposition

Component	Definition & Formula	Interpretation	Consequences & Examples
Bias² (Systematic Error)	$\text{Bias}^2 = (\text{E}[\hat{f}(x)] - f(x))^2$	Measures the difference between the expected prediction and the true value. Represents systematic error in the model.	**High bias**: Model oversimplifies (underfitting). *Example*: Linear regression for quadratic data.
Variance (Model Sensitivity)	$\text{Variance} = \text{E}[(\hat{f}(x) - \text{E}[\hat{f}(x)])^2]$	Measures how much predictions vary across different training sets. Represents model sensitivity to training data.	**High variance**: Model is too complex (overfitting). *Example*: High-degree polynomial fitting noise.
Irreducible Error (σ^2)	$\sigma^2 = \text{Var}(\epsilon)$	Noise inherent in the data generation process. Cannot be reduced by any model.	Sets the theoretical lower bound on prediction error. *Source*: Measurement errors, hidden variables.

Table 2.8: Comparison of Regularization Techniques

Type	Sparsity	Featur Selec-e tion	Optimization
L2 (Ridge)	No	No	Easy (differentiable)
L1 (Lasso)	Yes	Yes	Harder (subgradients)

Elastic Net	Partial Implicit	Partial Implicit	Moderate Standard
Dropout			
Early Stop-ping	No	No	Trivial

Table 2.9: Comparison of Gradient Descent Variants

Algorithm	Gradient Computation	Complexity	Convergence	Memory
Batch GD	$\frac{1}{n}\sum_{i=}^{n} \nabla L(\theta; x_i, y_i)$	$O(n)$	Smooth, deterministic	
Stochastic GD	$\nabla L(\theta; x_i, y_i)$ (single sample)	$O(1)$	Noisy, may oscillate	
Mini-batch GD	$\frac{1}{b}\sum_{j=1}^{b} \nabla L(\theta; x_j, y_j)$	$O(b)$	Balanced, practica	

Table 2.10: Advantages and Limitations of SGD

Advantages	Limitations
· **Computational efficiency:** $O(1)$ per update vs $O(n)$ for batch GD	· **Noisy gradients:** High variance in updates
· **Escapes local minima:** Noise helps escape saddle points	· **Convergence issues:** May oscillate near optimum
· **Online learning:** Can process stream-ing data	· **Hyperparameter sensitive:** Learn-ing rate crucial
· **Memory efficient:** Doesn't require storing entire dataset	· **Non-deterministic:** Different runs yield different results

x_1	x_2	h_1 (OR)	h_2 (NAND)	$v_1h_1 + v_2h_2 + b_3$	$\hat{y}$
0	0	0.119	0.9991	$5 \cdot 0.119 + 5 \cdot 0.9991 - 7 =$	1.4095 0.196 (0)
0	1	0.953	0.8808	$5 \cdot 0.953 + 5 \cdot 0.8808 - 7 = 2.169$	0.897 (1)
1	0	0.953	0.8808	2.169	0.897 (1)
1	1	0.9997	0.0474	$5 \cdot 0.9997 + 5 \cdot 0.0474 - 7 = -1.7645$	0.146 (0)

Activation	Max Derivative	Vanish?	Explode?	Dead Neurons	Compute Cost
Sigmoid	0.25	Severe	No	No	High
Tanh	1.0	Moderate	No	No	High
ReLU	1.0	No	Possible	Yes	Low
Leaky ReLU	1.0	No	Possible	Rare	Low
ELU	1.0	No	Possible	Rare	Medium
Swish	≈ 1.1	No	Possible	Rare	High

Table 2.11: Comparison of activation functions for gradient flow

Criterion	FNN	CNN	RNN
Input Type	Fixed-size vectors	Grid-like data	Sequences
Parameter Sharing	No	Spatial	Temporal
Translation Invariance	No	Yes	No
Temporal Modeling	No	Limited	Yes
Parallelizability	Full	Partial (spatial)	Limited
Memory Usage	High	Medium	Medium
Training Speed	Fast	Medium	Slow
Interpretability	Low	Medium	Low

Table 2.12: Architecture comparison across key dimensions

Activation	Max Derivative	Vanish?	Explode?	Dead Neurons	Compute Cost
Sigmoid	0.25	Severe	No	No	High
Tanh	1.0	Moderate	No	No	High
ReLU	1.0	No	Possible	Yes	Low
Leaky ReLU	1.0	No	Possible	Rare	Low
ELU	1.0	No	Possible	Rare	Medium
Swish	≈ 1.1	No	Possible	Rare	High

Table 2.15: Comparison of activation functions for gradient flow

Criterion	FNN	CNN	RNN
Input Type	Fixed-size vectors	Grid-like data	Sequences
Parameter Sharing	No	Spatial	Temporal
Translation Invariance	No	Yes	No
Temporal Modeling	No	Limited	Yes
Parallelizability	Full	Partial (spatial)	Limited
Memory Usage	High	Medium	Medium
Training Speed	Fast	Medium	Slow
Interpretability	Low	Medium	Low

Table 2.16: Architecture comparison across key dimensions

UNIT III

UNIT-III: REGULARIZATION AND OPTIMIZATION OF DL MODELS (9)

Regularization for Deep Learning: Parameter Norm Penalties, Norm Penalties as Constrained Optimization, Regularization and Under-Constrained Problems, Dataset Augmentation, Noise Robustness, Semi-Supervised Learning, Multi-Task Learning, Early Stopping, Parameter Tying and Parameter Sharing, Sparse Representations, Bagging and Other Ensemble Methods, Dropout, Adversarial Training, Tangent Distance, Tangent Prop and Manifold Tangent Classifier.

Optimization for Training Deep Models: Pure Optimization, Challenges in Neural Network Optimization, Basic Algorithms, Parameter Initialization Strategies, Algorithms with Adaptive Learning Rates, Approximate Second-Order Methods, Optimization Strategies and Meta- Algorithms.

Chapter 3

Regularization and Optimization of Deep Learning Models

3.1 Regularization for Deep Learning

Regularization techniques are essential for improving the generalization performance of deep learning models by preventing overfitting. Overfitting occurs when a model learns the train-ing data too well, including noise and random fluctuations, which reduces its ability to generalize to unseen data.

3.1.1 Parameter Norm Penalties

Definition 3.1.1 (Regularization)**.** *Regularization refers to any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.*

Definition 3.1.2 (Parameter Norm Penalty)**.** *A parameter norm penalty adds a regulariza-tion term* $\Omega(\boldsymbol{\theta})$ *to the loss function* $J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y})$*, resulting in the regularized loss* $\tilde{J}$*:*

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta})$$

where $\alpha \in [0, \infty)$ *is a hyperparameter that weights the contribution of the regularization term.*

The Core Idea: Trading Training Error for Simpler Models

The definitions correctly establish that regularization aims to reduce generalization error, often at the expense of training error. Parameter norm penalties achieve this by adding a term to the loss function that penalizes the model for having large parameter values (usually weights, not biases).

The intuition is that large parameter values can lead to overly complex, unstable models:

- *A model with huge weights might produce an extremely "wiggly" or "steep" decision boundary or regression curve. It can contort itself perfectly to fit every single training data point, including the noise.*

- *By penalizing large weights, we encourage the learning algorithm to find a solution that fits the data reasonably well but with smoother, simpler functions. This simpler model is less likely to be a perfect fit to the noise in the training set and will generalize better to new, unseen data.*

The hyperparameter α controls this trade-off:

- *α = 0: No regularization. The model is free to use any parameter values to minimize the original loss J.*
- *Small α: A slight preference for smaller weights. The model will only reduce its weights if it does not increase the original loss J too much.*
- *Large α: A strong preference for very small weights. The model will sacrifice perfor-mance on the training set (increase J) to achieve significantly smaller weights.*

L2 Parameter Regularization (Weight Decay)

This is the most common form of parameter norm penalty. The regularization term is the square of the L2 norm of the weight vector $\boldsymbol{w}$:

$$\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{w}\|_2^2 = \frac{1}{2}\sum_i w_i^2$$

The $\frac{1}{2}$ is a convenience factor for calculus, as its derivative will be w_i.

The regularized loss function becomes:

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) + \frac{\alpha}{2}\boldsymbol{w}^\top \boldsymbol{w}$$

Effect on the Learning Algorithm (Gradient Descent): To see its effect, we com-pute the gradient of the regularized loss. Let $\boldsymbol{g}$ be the gradient of the original loss J with respect to $\boldsymbol{w}$.

$$\nabla_{\boldsymbol{w}}\tilde{J} = \boldsymbol{g} + \alpha\boldsymbol{w}$$

When we perform the parameter update with a learning rate ϵ, we get:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon(\boldsymbol{g} + \alpha\boldsymbol{w})$$

$$\boldsymbol{w} \leftarrow (1 - \epsilon\alpha)\boldsymbol{w} - \epsilon\boldsymbol{g}$$

Interpretation: This update has a profound effect. The term $(1 - \epsilon\alpha)\boldsymbol{w}$ means that **before applying the gradient from the original loss, the weight vector is first shrunk towards zero by a constant factor**. This is why L2 regularization is also called **weight decay**.

Why It Works (A More Mathematical View): We can think about the effect of L2 regularization on the original loss function. If we make a quadratic approximation of the regularized loss near a minimum of the unregularized loss, we can show that L2 regularization adds a term of $\alpha\boldsymbol{I}$ to the

Hessian matrix. This effectively shifts the eigenvalues of the Hessian. The new minimum of the regularized loss is the old minimum, but with its components along directions of low curvature (where the data does not constrain the model much) shrunk significantly. In other words, it shrinks the parameters that are not strongly supported by the data.

L1 Regularization

Another common approach is the L1 parameter penalty. The regularization term is the sum of the absolute values of the weights:

$$\Omega(\boldsymbol{\theta}) = \|\boldsymbol{w}\|_1 = \sum_i |w_i|$$

The regularized loss function is:

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) + \alpha \|\boldsymbol{w}\|_1$$

Effect on the Learning Algorithm (Gradient Descent): The gradient of the L1 penalty is the sign of the weight vector sign($\boldsymbol{w}$):

$$\nabla_w \tilde{J} = \boldsymbol{g} + \alpha \operatorname{sign}(\boldsymbol{w})$$

The update rule becomes:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon \boldsymbol{g} - \epsilon\alpha \operatorname{sign}(\boldsymbol{w})$$

Interpretation: Unlike L2 regularization, which shrinks weights by a factor proportional to their current size, L1 regularization subtracts a **constant** from the weight at each update (moving it towards zero). Once a weight becomes very small, this constant contribution can push it all the way to zero.

Key Property: Sparsity The most significant outcome of L1 regularization is that it leads to **sparse solutions**. This means that during the optimization process, many of the model's parameters will be driven to exactly zero.

- **Why sparsity?** The constant force pushing weights towards zero, combined with the fact that the gradient of the loss $\boldsymbol{g}$ might be small, means weights can get "stuck" at zero. This is not the case with L2 regularization, where the shrinkage force is proportional to the weight itself, so it gets weaker as the weight approaches zero.
- **Consequence:** Sparse models are often more interpretable (you can see which features are actually used) and can be more memory-efficient, as the zero weights can be ignored.

Summary Table: L1 vs. L2 Regularization

Important Nuance: Biases are Usually Not Regularized

As noted in the original text, it is common practice to regularize only the weights $\boldsymbol{w}$ of a model and not the biases $\boldsymbol{b}$.

- **Reasoning:** Biases are a single parameter that controls the global offset of a function. They do not control the influence of a specific feature on the output. Regularizing a bias would just force the function to pass through or near the origin, which is an arbitrary constraint. It does not help control the model's complexity in a meaningful way and can harm its ability to fit the data.

Feature	L2 Regularization (Weight Decay)	L1 Regularization (Lasso)
Penalty Term	$\frac{1}{2}\sum_i w^2$	$\sum_i \lvert w_i \rvert$
Gradient of Penalty	w_i	$\text{sign}(w_i)$
Update Rule Effect	Multiply weights by a factor < 1 (shrinkage).	Subtract a constant from weights.
Primary Outcome	Prevents any single weight from becoming too large.	Drives many weights to exactly zero.
Resulting Model	Dense (all weights are small but non-zero).	Sparse (many weights are zero).
Best Use Case	When you have many features that all contribute a little.	When you suspect many features are irrelevant.

Table 3.1: Comparison of L1 and L2 Regularization

Need for Regularization

Deep neural networks often contain millions of parameters, giving them tremendous capacity to model complex patterns in data. While this flexibility is powerful, it also makes them highly susceptible to **overfitting**—memorizing noise and irrelevant details in the training set rather than learning generalizable patterns. Regularization addresses this fundamental challenge by constraining the learning process in ways that

promote better generalization.

The key benefits of regularization include:

(a) Reducing model complexity: Regularization prevents the model from fitting exces-sively complex functions by penalizing large parameter values or encouraging simpler representations. This helps strike a balance between underfitting (too simple) and overfitting (too complex).

(b) Improving generalization to unseen data: By discouraging the model from mem-orizing noise-specific patterns in the training data, regularization ensures that the learned patterns hold for new, unseen examples. This is the primary goal of any regu-larization technique.

(c) Preventing excessive weight values: Large weights often indicate that the model is assigning too much importance to specific features or is creating unstable decision boundaries. Regularization constrains weights to reasonable magnitudes, leading to more stable and reliable predictions.

(d) Enhancing numerical stability: During training, excessively large weights can cause gradients to explode, leading to unstable optimization. Regularization helps maintain weights within a reasonable range, making the learning process more stable and reliable.

(e) Providing implicit feature selection: Certain regularization techniques, particu-larly L1 regularization, can drive irrelevant feature weights to zero. This automatically identifies which features are most important for the task, simplifying the model and improving interpretability.

Together, these benefits make regularization an indispensable tool in deep learning, en-abling practitioners to build models that not only fit the training data well but also perform reliably on real-world data.

Common Norm Penalties

While the concept of parameter norm penalties is general, three specific formulations have emerged as the most widely used in practice. Each offers distinct properties and benefits for regularizing neural networks.

L2 Regularization (Ridge Regression) Also known as weight decay or Tikhonov regularization, L2 regularization is the most prevalent form of parameter penalty in deep learning. It adds the squared Euclidean norm of the weight vector to the loss function:

$$\Omega(w) = \frac{1}{2}||w||_2^2 = \frac{1}{2}w^T w = \frac{1}{2}\sum w_i^2$$

The factor 1 simplifies differentiation, as the derivative of w_i^2 becomes w_i rather than $2w_i$.

The gradient of the regularized objective becomes:

$$\nabla_w \tilde{J}(w; X, y) = \nabla_w J(w; X, y) + \alpha w$$

During gradient descent update with learning rate ϵ:

$$w \leftarrow w - \epsilon (\nabla_w J(w; X, y) + \alpha w) = (1 - \epsilon\alpha) w - \epsilon \nabla_w J(w; X, y)$$

This update reveals the fundamental mechanism of L2 regularization: weights are multiplied by the decay factor $(1 - \epsilon\alpha)$ before the standard gradient update is applied. This constant shrinkage towards zero, independent of the data, is why L2 regularization is called "weight decay."

Key properties:

- Produces dense solutions where all weights remain non-zero but are proportionally shrunk
- Particularly effective when many features contribute weakly to the prediction
- Improves numerical stability by preventing weights from growing too large
- Has a rotationally invariant penalty—all directions in weight space are treated equally

L1 Regularization (Lasso) L1 regularization, also known as Lasso (Least Absolute Shrinkage and Selection Operator), adds the sum of absolute values of weights:

$$\Omega(w) = \|w\|_1 = \sum_i |w_i|$$

Unlike L2 regularization, the L1 penalty is not differentiable at zero. The gradient (technically a subgradient) is:

$$\nabla_{w_i} \Omega(w) = \text{sign}(w_i) = \begin{cases} +1 & \text{if } w_i > 0 \\ -1 & \text{if } w_i < 0 \\ 0 & \text{if } w_i = 0 \end{cases}$$

This leads to a fundamentally different update rule:

$$w_i \leftarrow w_i - \epsilon (\nabla_{w_i} J(w; X, y) + \alpha \, \text{sign}(w_i))$$

Key properties:

- Produces sparse solutions where many weights become exactly zero during training

- Acts as an implicit feature selector—weights corresponding to irrelevant features are driven to zero
- The constant magnitude of the penalty (independent of the weight's current value) creates a "dead zone" where small weights are pushed all the way to zero
- Particularly useful when interpretability is important or when we suspect that only a subset of features are relevant
- Less rotationally invariant than L2—the penalty aligns with the coordinate axes, which encourages sparsity in the original feature space

Elastic Net Regularization Elastic net regularization combines both L1 and L2 penal-ties, offering a compromise between their respective strengths:

$$\Omega(w) = \rho \| w \| 1 \quad + 1 \; \bar{2} \, \rho \| \boldsymbol{w} \| \bar{2}^{2}$$

where ρ∈[0, 1] is a mixing parameter that controls the relative contribution of each penalty. When ρ = 1, elastic net reduces to pure L1 regularization; when ρ = 0, it becomes pure L2 regularization.

Motivation and key properties:

- Overcoming L1 limitations: When features are correlated, L1 regularization tends to select only one feature from a correlated group arbitrarily. Elastic net encourages selecting correlated features together (a grouping effect).
- Handling high-dimensional data: In problems with more features than samples (p > n), L1 regularization can select at most n features. Elastic net has no such limitation.
- **Stability:** The L2 component provides numerical stability and helps prevent the er-ratic behavior that pure L1 can exhibit with highly correlated features.
- The mixing parameter ρ provides fine-grained control over the regularization behavior, allowing practitioners to tune for the desired balance between sparsity and grouping effects.

Practical considerations: Elastic net is particularly valuable in domains like genomics or text processing, where features are often correlated and the number of features can vastly exceed the number of samples. The additional flexibility of the mixing parameter ρ allows for more nuanced regularization than either L1 or L2 alone.

Penalty Type	Regularization Term	Key Property	Best Use Case
L2 (Ridge)	$\frac{1}{2}\|\|\boldsymbol{w}\|\|_2^2$	Dense, non-sparse so-lutions	Many features with weak contributions
L1 (Lasso)	$\|\|\boldsymbol{w}\|\|_1$	Sparse solutions, fea-ture selection	Few relevant fea-tures, interpretabil-ity needed
Elastic Net	$\rho\|\|\boldsymbol{w}\|\|_1 + \frac{1-\rho}{2}\|\|\boldsymbol{w}\|\|_2^2$	Sparse with grouping effect	Correlated fea-tures, $p > n$ scenarios

Table 3.2: Summary of common norm penalties and their characteristics

Mathematical Derivation and Properties

To develop a deeper understanding of how parameter norm penalties fundamentally alter the learning problem, it is instructive to examine the

mathematics in the context of a simple linear regression model. This analysis reveals several important properties that extend to more complex neural networks.

Closed-Form Solution with L2 Regularization Consider a linear regression model with L2 regularization (ridge regression). The regularized objective function is:

$$\tilde{J}(\boldsymbol{w}) = \frac{1}{2m}\sum_{i=}^{m}(\boldsymbol{w}^T\boldsymbol{x}^{(i)} - y^{(i)})^2 + \frac{\lambda}{2}\|_2\boldsymbol{w}\|^2$$

where m is the number of training examples and λ is the regularization coefficient (equiv-alent to α in earlier notation). In matrix form, this becomes:

$$\tilde{J}(\boldsymbol{w}) = \frac{1}{2m}\|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|_2^2 + \frac{\lambda}{2}\boldsymbol{w}^T\boldsymbol{w}$$

To find the optimal weights $\boldsymbol{w}^*$, we compute the gradient and set it to zero:

$$\nabla\ \tilde{J}_w(\boldsymbol{w}) = \frac{1}{m}\boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + \lambda\boldsymbol{w} = 0$$

Rearranging the terms:

$$\frac{1}{m}\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w} - \frac{1}{m}\boldsymbol{X}^T\boldsymbol{y} + \lambda\boldsymbol{w} = 0$$

$$\frac{1}{m}\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I}\quad \boldsymbol{w} = \frac{1}{m}\boldsymbol{X}^T\boldsymbol{y}$$

This yields the closed-form solution:

$$\boldsymbol{w}^* = (\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

Key Properties Revealed by the Derivation This simple expression reveals several profound properties of L2 regularization:

1. **Guaranteed invertibility:** The matrix $\boldsymbol{X}^T\ \boldsymbol{X}$ is positive semidefinite but may be singular (non-invertible) when features are linearly dependent or when there are fewer samples than features ($m < n$). Adding $\lambda\boldsymbol{I}$ shifts all eigenvalues by λ, making the matrix $\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I}$ strictly positive definite and therefore always invertible. This ensures a unique solution exists even in ill-posed problems.
2. **Eigenvalue interpretation:** Let $\boldsymbol{X}^T\ \boldsymbol{X} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T$ be its eigendecomposition, where $\boldsymbol{\Lambda}$ is a diagonal matrix of eigenvalues λ_i.

The regularized solution can be expressed as:

$$\boldsymbol{w}^* = \boldsymbol{Q}(\boldsymbol{\Lambda} + \lambda \boldsymbol{I})^{-1}\boldsymbol{Q}^T\boldsymbol{X}^T\boldsymbol{y}$$

In the unregularized solution ($\lambda = 0$), directions with very small eigenvalues λ_j (indi-cating low data variance) can cause the solution to blow up, as $\boldsymbol{w}^*$ involves division by λ_j. L2 regularization replaces $\frac{1}{\lambda_j}$ with $\frac{1}{\lambda_j+\lambda}$, effectively damping the contribution of low-variance directions.

3. **Connection to weight decay:** The update rule derived earlier, $\boldsymbol{w} \leftarrow (1 - \epsilon\lambda)\boldsymbol{w} - \epsilon\nabla J(\boldsymbol{w})$, is the gradient-based analog of this closed-form solution. Both show how regularization shrinks weights toward zero.

Effect on the Loss Landscape We can also understand regularization by examining its effect on the curvature of the loss function. The Hessian matrix (matrix of second derivatives) of the unregularized loss is:

$$\boldsymbol{H} = \frac{1}{m}\boldsymbol{X}^T\boldsymbol{X}$$

For the regularized loss, the Hessian becomes:

$$\tilde{\boldsymbol{H}} = \boldsymbol{H} + \lambda \boldsymbol{I}$$

Adding $\lambda\boldsymbol{I}$ to the Hessian has the following effects:

- All eigenvalues of the Hessian are increased by λ, making the loss landscape more sharply curved in all directions
- This increased curvature creates a stronger attraction toward the minimum, preventing weights from moving too far in low-curvature directions where the data provides little constraint
- The regularized loss becomes more strictly convex, making optimization easier and more stable

Comparison with L1 Regularization While L2 regularization admits a convenient closed-form solution, L1 regularization does not due to the non-differentiability at zero. However, we can gain insight by comparing the optimality conditions.

For L1 regularization, the regularized objective is:

$$\tilde{J}(\boldsymbol{w}) = \frac{1}{2m}\|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|^2 + \lambda\|\boldsymbol{w}\|_1$$

The subgradient optimality condition yields:

$$\frac{1}{m} \boldsymbol{X}^T (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + \lambda \partial \| \boldsymbol{w} \|_1 = 0$$

where $\partial \| \boldsymbol{w} \|_1$ is the subgradient, taking values in $[-1, 1]$ for each component. This condition leads to the sparsity property: for features that are insufficiently correlated with the residuals, the optimal weight is exactly zero.

Bayesian Interpretation Parameter norm penalties can also be understood from a Bayesian perspective:

- **L2 regularization** corresponds to a **Gaussian prior** on the weights with zero mean and variance $\frac{1}{\lambda}$. The regularized loss is equivalent to the negative log-posterior, where maximizing the posterior (MAP estimation) yields the same solution as minimizing the regularized loss.
- **L1 regularization** corresponds to a **Laplacian prior** on the weights, which has heavier tails and a sharper peak at zero. This peak at zero encourages sparsity in the posterior mode.

This Bayesian view provides intuition: regularization encodes our *prior belief* that weights should typically be small (Gaussian) or sparse (Laplacian), and we update this belief with evidence from the data.

Summary of Mathematical Properties

Property	**Effect without Regularization**	**Effect with Regularization**
Matrix invertibility	$\boldsymbol{X}^T\boldsymbol{X}$ may be singular	$\boldsymbol{X}^T\boldsymbol{X}+\lambda\boldsymbol{I}$ always invertible
Solution uniqueness	Multiple solutions when rank-deficient	Unique solution guaranteed
Eigenvalue handling	Divergent weights for small eigenvalues	Damped contribution of small eigenvalues
Loss curvature	Determined solely by data	Increased curvature in all directions
Bayesian prior	Implicit uniform prior	Explicit Gaussian (L2) or Laplacian (L1) prior

Table 3.3: Comparison of optimization properties with and without regularization

3.1.2 Norm Penalties as Constrained Optimization

The relationship between penalized optimization (adding a term to the loss function) and constrained optimization (imposing an explicit constraint) provides fundamental insight into how regularization works. Understanding this equivalence helps practitioners reason about the behavior of regularization methods and choose appropriate hyperparameters.

Theorem 3.1.1 (Equivalence of Penalized and Constrained Optimization)**.** *For every reg-ularization coefficient* $\alpha \geq 0$*, there exists a constraint radius* $\kappa \geq 0$ *such that the solution to the penalized optimization problem:*

$$\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta})$$

is also a solution to the constrained optimization problem:

$$\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) \qquad \textit{subject to } \Omega(\boldsymbol{\theta}) \leq \kappa$$

Intuitive Interpretation This theorem reveals that penalized regularization is equivalent to imposing a hard constraint on the norm of the parameters. The Lagrange multiplier α determines how strictly this constraint is enforced:

- When α is large (strong regularization), the corresponding κ is small—the parameters are confined to a small region around the origin.
- When α is small (weak regularization), the corresponding κ is large—the parameters have more freedom to move.
- As $\alpha \to 0$, $\kappa \to \infty$, recovering the unregularized solution.
- As $\alpha \to \infty$, $\kappa \to 0$, forcing all parameters toward zero.

Geometric Insight This equivalence provides a geometric interpretation of regularization:

- **L2 regularization** corresponds to constraining the weights to lie within a hypersphere (ball) of radius κ. The optimal solution is found where the level curves of the loss function first touch this sphere.
- **L1 regularization** corresponds to constraining the weights to lie within a diamond-shaped region (L1-ball) in parameter space. The corners of this diamond align with the coordinate axes, explaining why L1 regularization encourages sparse solutions—the optimum often occurs at a corner where some weights are exactly zero.

Practical Implications This perspective offers several practical insights:

1. **Hyperparameter interpretation:** Rather than thinking abstractly about adding a penalty, practitioners can think about limiting the "size" of the model's parameters through κ.
2. **Grid search strategy:** The mapping between α and κ is monotonic but nonlinear. Understanding this helps in designing effective hyperparameter search spaces.
3. **Model capacity control:** Directly constraining the norm can sometimes be more intuitive than tuning a penalty coefficient, particularly when prior knowledge about reasonable parameter magnitudes exists.

3.1.3 Regularization and Under-Constrained Problems

Definition 3.1.3 (Under-Constrained Problem)**.** *A learning problem is under-constrained when the number of parameters exceeds the number of effectively independent constraints from the data, leading to multiple solutions that perfectly fit the training data. This commonly occurs when:*

- *The number of parameters exceeds the number of training examples ($p > n$)*
- *Features are linearly dependent (multicollinearity)*
- *The model has intrinsic symmetries that create equivalent solutions*

The Challenge of Under-Constrained Problems In deep learning, under-constrained problems are the norm rather than the exception:

- Modern neural networks often have millions of parameters but may be trained on thousands or even hundreds of examples in some domains.
- Even with sufficient examples, architectural choices (like overcomplete representations in autoencoders) deliberately create under-constrained intermediate representations.
- Without regularization, the learning algorithm must choose arbitrarily among infinitely many solutions that all achieve zero training error.

How Regularization Resolves Under-Constraint Regularization addresses under-constrained problems through several mechanisms:

1. **Selecting a unique solution:** By adding a preference for small weights, regular-ization picks a specific solution from the infinite set of possibilities—typically the one with minimum norm that still fits the data.
2. **Encoding Occam's razor:** Regularization formalizes the principle of parsi-mony—simpler explanations (with smaller parameters) are preferred over complex ones, assuming they explain the data adequately.
3. **Improving numerical stability:** In under-constrained linear problems, the matrix $\boldsymbol{X}^T \boldsymbol{X}$ is singular. Adding regularization (as in ridge regression) ensures invertibility and yields numerically stable solutions.
4. **Implicit bias of gradient descent:** Interestingly, even without explicit regulariza-tion, gradient descent on under-constrained linear problems converges to the minimum-norm solution—a form of implicit regularization.

Connection to Generalization The benefit in under-constrained settings extends be-yond mere identifiability:

- Among all solutions that fit the training data perfectly, those with smaller norms typically generalize better to unseen data.
- This aligns with the bias-variance trade-off: the minimum-norm solution tends to have lower variance, reducing overfitting.

3.1.4 Dataset Augmentation

Definition 3.1.4 (Dataset Augmentation)**.** *Dataset augmentation refers to creating mod-ified copies of existing training examples to artificially increase the size and diversity of the training dataset. This technique encodes prior knowledge about invariances in the data—transformations that should not change the label or meaning of the input.*

The Fundamental Principle The core idea behind dataset augmentation is simple yet powerful: if we know that certain transformations of the input should preserve the output (e.g., a cat rotated slightly is still a cat), we can explicitly generate transformed examples and train the model to be invariant

to these transformations. This is equivalent to building invariances directly into the model without changing its architecture.

Common Augmentation Techniques for Images Image data has proven particularly amenable to augmentation due to the wealth of label-preserving transformations:

- **Geometric transformations:** Rotation (typically within a small angle range), scal-ing, translation (shifting), horizontal flipping, and random cropping. These simulate variations in viewpoint and object position.
- **Color space transformations:** Adjusting brightness, contrast, saturation, and hue. These simulate different lighting conditions and camera settings.
- **Noise injection:** Adding Gaussian noise, salt-and-pepper noise, or speckle noise. This improves robustness to sensor noise and minor corruptions.
- **Elastic deformations:** Applying smooth, random distortions to the image. Particu-larly effective in handwritten text recognition and medical imaging.
- **Mixup:** Creating new samples by linear interpolation of both inputs and labels:

  $$\tilde{\boldsymbol{x}} = \lambda \boldsymbol{x}_i + (1 - \lambda)\boldsymbol{x}_j, \quad \tilde{y} = \lambda y_i + (1 - \lambda) y_j$$

  where $\lambda \sim \text{Beta}(\gamma, \gamma)$ and γ is a hyperparameter. This encourages linear behavior between training examples.
- **Cutout and Random Erasing:** Masking out random square regions of the image, forcing the model to rely on broader context rather than specific features.
- **Adversarial augmentation:** Generating examples that maximally fool the current model and including them in training, improving robustness to adversarial attacks.

Mathematical Formulation For an input $\boldsymbol{x}$ (e.g., an image), an augmented version is generated as:

$$\boldsymbol{x}' = T(\boldsymbol{x})$$

where T is a transformation function drawn from a family of label-preserving transformations
T . The model is then trained on both original and augmented examples:

$$\tilde{J}(\boldsymbol{\theta}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \lambda \, \mathrm{E}_{T \sim \mathrm{T}} \left[J(\boldsymbol{\theta}; T(\boldsymbol{X}), \boldsymbol{y}) \right]$$

In practice, transformations are applied on-the-fly during training, with each mini-batch containing randomly transformed versions of the original images.

Benefits Beyond Size Increase Dataset augmentation provides multiple benefits beyond simply having more data:

1. **Enforced invariances:** The model learns explicitly to be invariant to the applied transformations, which is often desirable for the task.
2. **Reduced overfitting:** By presenting varied versions of each example, the model cannot simply memorize pixel values and must learn more robust features.
3. **Improved generalization:** Models trained with augmentation typically perform bet-ter on test data, especially when the test distribution shifts slightly (e.g., different lighting conditions).
4. **Smoother decision boundaries:** Augmentation encourages the model to produce consistent outputs across neighborhoods of each training example, leading to smoother and more plausible decision boundaries.

Considerations and Best Practices Effective dataset augmentation requires careful design:

- **Label preservation:** Transformations must not change the semantic meaning of the input. Flipping a 6 horizontally might turn it into a 9, which would be inappropriate for digit recognition.
- **Appropriate magnitude:** Too little augmentation provides minimal benefit; too much can generate unrealistic examples that confuse the model.
- **Domain specificity:** Effective augmentations are highly domain-dependent. For au-dio, one might add background noise or change pitch; for text, one might use synonym replacement or back-translation.
- **Computational efficiency:** On-the-fly augmentation avoids storing multiple copies of the dataset but must be efficient enough to keep up with training.

Connection to Regularization Theory Dataset augmentation can be viewed as a form of regularization because it:

- Reduces generalization error without necessarily reducing training error
- Encodes prior knowledge about the data distribution

- Prefers smoother, more invariant functions
- Effectively expands the training distribution to cover more of the input space

Augmentation Type	Examples	Invariance Encoded
Geometric	Rotation, scaling, translation	Position, orientation, size
Photometric	Brightness, contrast, color	Lighting conditions
Noise-based	Gaussian noise, blur	Sensor noise, minor corruptions
Mixing	Mixup, CutMix	Linear interpolations between classes
Deformation	Elastic deformations	Non-rigid transformations

Table 3.4: Common dataset augmentation techniques and the invariances they encode

3.1.5 Noise Robustness

Definition 3.1.5 (Noise Robustness Regularization)**.** *Noise robustness regularization refers to techniques that deliberately inject random perturbations into various components of the learning process—inputs, weights, or hidden representations—during training. This exposes the model to a wider range of variations, forcing it to learn features that are robust to small perturbations and improving generalization to unseen data.*

The Underlying Principle The intuition behind noise robustness regularization is straightforward yet profound: if a model can maintain correct predictions despite random perturbations to its inputs or internal representations, it must have captured the true un-derlying structure of the data rather than memorizing spurious patterns. This is analogous to building a model that is stable in the neighborhood of each training example.

Types of Noise Injection

Input Noise Adding noise to input features is one of the simplest and most intuitive forms of regularization:

$$\tilde{\boldsymbol{x}} = \boldsymbol{x} + \boldsymbol{\epsilon}, \ \boldsymbol{\epsilon} \sim \mathrm{N}(0,$$

$\sigma^2\boldsymbol{I}$) where σ^2 controls the variance of the

injected noise.

Mechanism and effects:

- Input noise effectively expands the training distribution by creating new examples that lie in the vicinity of original training points.
- This encourages the model to learn functions that are smooth—small changes in input should produce small changes in output.
- Mathematically, for small noise levels, this is equivalent to adding a regularization term to the loss function. A Taylor series expansion shows that input noise with variance σ^2 approximates a penalty on the squared norm of the gradient:

$$\mathrm{E}_{\boldsymbol{\epsilon}}[J(\boldsymbol{x} + \boldsymbol{\epsilon}, y)] \approx J(\boldsymbol{x}, y) + \frac{\sigma^2}{2}\|\nabla_{\boldsymbol{x}} J\|^2$$

- This gradient penalty encourages functions with small derivatives—functions that do not change rapidly with respect to input variations.

Practical considerations:

- The noise level σ must be tuned carefully. Too little noise provides minimal benefit; too much noise can corrupt the signal entirely.
- Different features may tolerate different noise levels. Features with small variance are more easily corrupted.
- Input noise is particularly effective for data where measurement error is expected, such as sensor readings or physical measurements.

Weight Noise Adding noise directly to the model parameters during training provides a different form of regularization:

$$\tilde{\boldsymbol{w}} = \boldsymbol{w} + \boldsymbol{\epsilon}, \boldsymbol{\epsilon} \sim \mathrm{N}(0, \eta^2 \boldsymbol{I})$$

Mechanism and effects:

- Weight noise encourages the model to find regions in parameter space where small perturbations do not significantly affect the output—so-called flat minima.
- Flat minima are believed to generalize better than sharp minima, where small param-eter changes can cause large output variations.
- This technique can be viewed as a form of stochastic approximation to a penalty on the Fisher information matrix or Hessian of the loss.
- For linear models, weight noise with variance η^2 is equivalent to adding an L2 regular-ization term with coefficient η^2.

Relationship to Bayesian inference: Weight noise has a natural interpretation in Bayesian terms. Training with weight noise approximates performing variational inference where we maintain a distribution over weights rather than point estimates. The noise level corresponds to the uncertainty in our weight estimates.

Label Smoothing Unlike input or weight noise, label smoothing modifies the target values rather than the inputs or parameters. It replaces hard 0/1 classification targets with softer targets:

$$y_{\text{smooth}} = (1 - \epsilon) y_{\text{hard}} + \frac{\epsilon}{K}$$

where K is the number of classes, and $\epsilon \in [0, 1]$ is a smoothing parameter. For a one-hot encoded target (1 for the correct class, 0 for others), this becomes:

$$y_{\text{smooth},c} = \begin{cases} 1 - \epsilon + \frac{\epsilon}{K} & \text{if } c = c_{\text{correct}} \\ \frac{\epsilon}{K} & \text{otherwise} \end{cases}$$

Mechanism and effects:

- Label smoothing prevents the model from becoming overconfident in its predictions by penalizing extreme logit values.
- Without smoothing, the model is incentivized to push the logit of the correct class to infinity and others to negative infinity to achieve exact 0/1 probabilities. This can lead to overfitting and poor calibration.
- With smoothing, the model is only required to assign probability $1 - \epsilon + \epsilon/K$ to the correct class, leaving finite slack.
- This encourages the model to learn representations that are more evenly distributed across classes and less sensitive to individual examples.

Benefits of label smoothing:

1. **Improved model calibration:** Smoothed models produce better-calibrated proba-bilities—their confidence better matches actual accuracy.
2. **Reduced overfitting:** By not requiring exact one-hot targets, the model cannot rely on memorizing specific examples.
3. **Better representations:** Label smoothing often leads to better feature representa-tions that transfer well to other tasks.
4. **Robustness to label noise:** If some training labels are incorrect, smoothing reduces the damage by not forcing the model to assign full probability to potentially wrong targets.

Theoretical Connections

Equivalence to Other Regularization Forms Noise injection techniques have deep connections to other regularization methods:

- Input noise with variance σ^2 is equivalent to Tikhonov regularization (L2 penalty) in the limit of small noise for linear models.
- Weight noise similarly approximates an L2 penalty on the weights.
- For neural networks, noise injection can be viewed as a form of data augmentation in the feature space rather than input space.

Bayesian Interpretation From a Bayesian perspective, noise injection during training approximates:

- **Input noise:** Marginalizing over a distribution of plausible inputs around each training point.

- **Weight noise:** Maintaining a posterior distribution over weights and integrating over this distribution when making predictions.

Practical Guidelines

Implementation Considerations When implementing noise robustness techniques:

- **Annealing:** Sometimes it helps to start with higher noise levels and gradually reduce them during training.
- **Adaptive noise:** Noise levels can be adjusted per example or per feature based on uncertainty estimates.
- **Combination:** These techniques are not mutually exclusive and can be combined for stronger regularization effects.
- **Validation:** Always monitor performance on a validation set to tune noise hyperpa-rameters appropriately.

Noise Type	Where Applied	Equivalent Regularization	Typical Application
Input noise	Input features $\boldsymbol{x}$	Gradient nor m penalty	Sensor data, im- ages with subtle variations
Weight noise	Parameters $\boldsymbol{w}$	L2 regularization	General pur- pose, seeki ng flat minima
Label smoothing	Target values y	Prevents overconf i- dence	Classification tasks with many classes

Table 3.5: Summary of noise robustness techniques and their characteristics

Limitations and Caveats Noise robustness techniques are powerful but come with con-siderations:

- Increased training time due to the need to process more varied examples or average over multiple noise realizations.
- Risk of underfitting if noise levels are too high.
- Difficulty in interpreting the effective regularization strength compared to simpler penalties like L2.

3.1.6 Semi-Supervised Learning

Definition 3.1.6 (Semi-Supervised Learning)**.** *Learning from both labeled and unlabeled data to improve performance.*

Common approaches:

- **Self-training**: Model predicts on unlabeled data, adds confident predictions to train-ing set
- **Co-training**: Multiple views of data trained simultaneously
- **Graph-based methods**: Use graph structure connecting labeled and unlabeled sam-ples
- **Generative models**: Learn data distribution from unlabeled data

3.1.7 Multi-Task Learning

Definition 3.1.7 (Multi-Task Learning)**.** *Multi-task learning (MTL) is a learning paradigm in which a single model is trained to perform multiple related tasks simultaneously. By sharing representations across tasks, the model leverages commonalities and differences across tasks to improve generalization on all tasks, often leading to better performance than training separate models for each task independently.*

Core Principle The fundamental insight behind multi-task learning is that tasks are rarely truly independent. Related tasks often share underlying structure, features, or pat-terns. By forcing the model to learn representations that are useful across multiple tasks, MTL acts as an inductive bias—it biases the model toward representations that capture generalizable features rather than task-specific noise.

When tasks share information, training on multiple tasks simultaneously creates a form of **implicit data augmentation**: the model sees more varied examples and must find explanations that work across different but related prediction problems. This reduces the risk of overfitting to any single task's idiosyncrasies.

Mathematical Formulation Formally, we have T tasks, each with its own dataset $D_t = \{(\boldsymbol{x}_i^{(t)}, y_i^{(t)})\}_{i=1}^{N_t}$. The goal is to learn:

- A set of shared parameters $\boldsymbol{\theta}_{\text{shared}}$ that capture cross-task structure
- Task-specific parameters $\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \ldots, \boldsymbol{\theta}_T$ that capture task-specific variations The overall objective combines losses from

all tasks:

$$\min_{\boldsymbol{\theta}_{\text{shared}}, \boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_T} \sum_{t=1} \alpha_t J_t(\boldsymbol{\theta}_{\text{shared}}, \boldsymbol{\theta}_t; \mathrm{D}_t)$$

where:

- T is the number of tasks
- J_t is the loss function for task t (e.g., cross-entropy for classification, mean squared error for regression)
- $\alpha_t \geq 0$ are task weights that balance the importance of each task (typically $\sum_t \alpha_t = 1$ or tuned as hyperparameters)
- $\boldsymbol{\theta}_{\text{shared}}$ represents parameters shared across all tasks (e.g., early layers of a neural net-work)
- $\boldsymbol{\theta}_t$ represents parameters specific to task t (e.g., final task-specific layers)

Architectural Patterns Multi-task learning can be implemented through several archi-tectural designs:

1. **Hard parameter sharing:** The most common approach. Early layers are shared across all tasks, while later layers are task-specific. This drastically reduces overfitting risk—the more tasks, the harder it is to overfit the shared parameters.
2. **Soft parameter sharing:** Each task has its own parameters, but distances between parameters are regularized (e.g., with L2 penalty) to encourage similarity. This offers more flexibility but more parameters.
3. **Hierarchical sharing:** Tasks are organized in a hierarchy, with different levels of sharing. For example, all vision tasks might share early layers, while groups of related tasks (e.g., classification tasks) share mid-level layers.
4. **Cross-stitch networks:** Learned linear combinations of task-specific representations at each layer, allowing the model to learn how much to share dynamically.

Benefits of Multi-Task Learning

1. **Improved data efficiency:** By sharing statistical strength, tasks with limited data benefit from tasks with abundant data.
2. **Regularization effect:** The need to perform well on multiple tasks prevents overfit-ting to any single task's noise, acting as an implicit regularizer.

3. **Richer representations:** Features learned must be useful across tasks, encouraging the model to capture more fundamental and transferable representations.
4. **Computational efficiency:** A single model serving multiple tasks requires less mem-ory and inference cost than separate models.
5. **Focus on relevant features:** If one task is difficult to learn due to noisy features, other tasks can help the model focus on relevant patterns.

Challenges and Considerations Despite its benefits, multi-task learning introduces sev-eral challenges:

1. **Task interference (negative transfer):** If tasks are not sufficiently related, forcing parameter sharing can degrade performance on all tasks. The shared representations may not be able to accommodate conflicting task requirements.
2. **Balancing task importance:** Setting the task weights α_t is nontrivial. Tasks with larger losses or gradients can dominate training. Common strategies include:
   - Uncertainty weighting: Weight tasks based on learned task uncertainty
   - Gradient balancing: Dynamically adjust weights to balance gradient magnitudes
   - Loss scaling: Scale losses to comparable ranges
3. **Different data modalities:** Tasks may operate on different input types or have different output spaces, complicating architecture design.
4. **Optimization difficulty:** The combined objective can create conflicting gradient directions, making optimization more challenging than single-task learning.
5. **Evaluation complexity:** A single model must be evaluated on multiple metrics across tasks, making model selection and hyperparameter tuning more complex.

Theoretical Justification Multi-task learning can be understood through several theo-retical lenses:

- **Bias-variance trade-off:** Sharing parameters increases bias (by forcing common rep-resentations) but reduces variance (by pooling data across tasks). When tasks are related, the bias increase is small while variance reduction is substantial.
- **Representation learning:** Learning a shared representation $h = f_{\text{shared}}(\boldsymbol{x})$ and task-specific predictors $g_t(h)$ can be seen as finding a representation that makes all tasks linearly separable or easy to predict.
- **Multi-task learning as regularization:** The constraint that tasks share parameters acts as a form of regularization, reducing the effective degrees of freedom of the model.

Practical Guidelines for Success To successfully apply multi-task learning:

1. **Choose related tasks:** Tasks should share meaningful structure. Domain knowl-edge is crucial here—tasks in the same domain (e.g., computer vision) often work well together.
2. **Start with hard parameter sharing:** It's simple, effective, and provides strong regularization. Move to more complex architectures only if needed.
3. **Monitor individual task performance:** Track each task's metrics separately. Sud-den drops may indicate negative transfer.
4. **Consider task grouping:** If you have many tasks, consider grouping related tasks and learning separate shared representations for each group.
5. **Adjust task weights carefully:** Use validation performance to tune weights, or employ adaptive weighting schemes.

Applications Multi-task learning has found success across diverse domains:

- **Computer vision:** Simultaneous object detection, segmentation, and depth estima-tion from a single image
- **Natural language processing:** Joint part-of-speech tagging, named entity recogni-tion, and dependency parsing
- **Autonomous driving:** Combined detection of vehicles, pedestrians,

lane markings, and traffic signs

- **Drug discovery:** Predicting multiple molecular properties simultaneously
- **Recommendation systems:** Predicting clicks, purchases, and engagement metrics together

Sharing Type	Description	When to Use
Hard parameter sharing	Shared early layers, task-specific later layers	Tasks are closely related; limited data
Soft parameter sharing	Separate parameters with similarity regular-ization	Tasks are somewhat re-lated but may need flexi-bility
Hierarchical sharing	Layered sharing: some layers shared among all, some among subgroups	Tasks have natural grouping structure
Cross-stitch	Learned linear combina-tions of task representa-tions	Optimal sharing pattern is unknown a priori

Table 3.6: Multi-task learning architectural patterns

3.1.8 Early Stopping

Definition 3.1.8 (Early Stopping)**.** *Early stopping is a regularization technique that halts training when performance on a validation set begins to deteriorate, even though training error continues to decrease. This prevents overfitting by selecting the model at its peak generalization rather than at minimum training error.*

The Overfitting Pattern During iterative training, models follow a predictable pattern:

- **Early epochs:** Both training and validation error decrease rapidly.
- **Middle epochs:** Training error continues decreasing; validation error reaches a min-imum.
- **Late epochs:** Training error keeps decreasing (sometimes near zero), but validation error starts increasing—the model begins memorizing noise rather than learning pat-terns.

Early stopping aims to halt training exactly when validation error is at its lowest.

How It Works

1. Reserve a validation set (e.g., 10-20% of training data).
2. After each epoch (or every few epochs), evaluate the model on the validation set.
3. Track the best validation performance seen so far and save those model parameters.
4. If validation performance does not improve for a set number of epochs (called **pa-tience**), stop training.
5. Restore the model parameters that achieved the best validation performance.

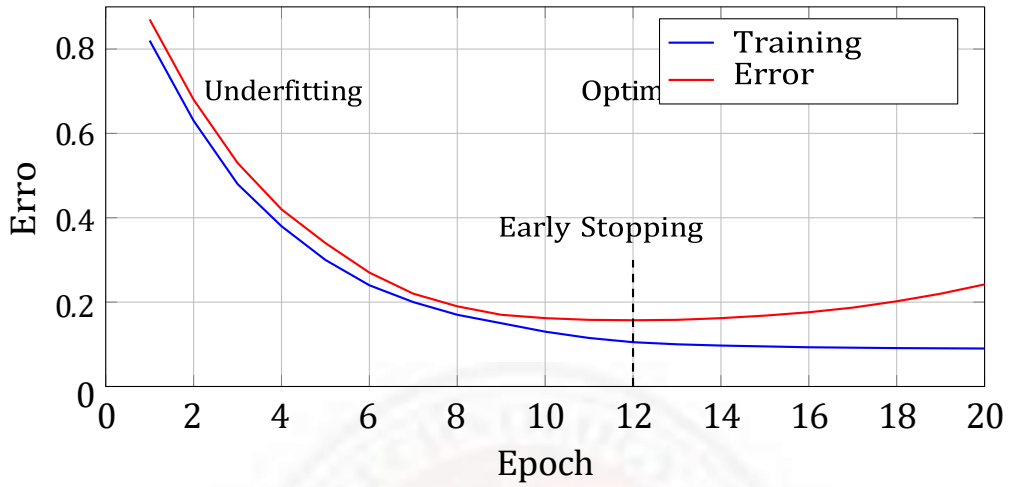


Figure 3.1: Early stopping selects the model when validation error is minimized, before overfitting begins.

Key Hyperparameters

- **Patience:** Number of epochs to wait after validation stops improving before stopping. Higher patience gives more chance to escape plateaus but risks overfitting.
- **Min delta:** Minimum improvement required to count as progress. Prevents stopping from tiny, insignificant fluctuations.

Advantages

- **Simple to implement:** No changes to model architecture or loss function.
- **Computationally efficient:** Saves time by stopping training early.
- **Automatic:** Selects optimal model complexity without manual tuning.
- **Complementary:** Works alongside other regularization methods (L2, dropout, etc.).

Practical Tips

- Always save the best model parameters—don't use the final parameters after stopping.
- Choose patience based on dataset size and training dynamics (typical values: 5-20 epochs).
- Monitor validation error, not training error, for stopping decisions.
- Can be combined with learning rate reduction—reduce learning rate

when validation plateaus, then apply early stopping if still no improvement.

Limitations

- Requires a validation set, reducing training data available.
- Noisy validation estimates can cause premature or delayed stopping.
- Some computation is ”wasted” during the patience window after the optimal point.

3.1.9 Parameter Tying and Parameter Sharing

Definition 3.1.9 (Parameter Sharing)**.** *Parameter sharing is a regularization technique that forces different parts of a neural network to use the same parameter values. By reducing the number of independent parameters in the model, it encodes inductive biases about problem structure and dramatically reduces the risk of overfitting.*

Core Concept The fundamental idea behind parameter sharing is simple yet powerful: if we believe that certain operations or transformations should be identical across different parts of the input or across different time steps, we can enforce this by making them share the same parameters. This is distinct from parameter tying (a more general concept where parameters are constrained to be close but not necessarily identical), though sharing is the stricter form.

Parameter Sharing vs. Parameter Tying

- **Parameter sharing:** Parameters are forced to be exactly equal. This is a hard constraint.
- **Parameter tying:** Parameters are regularized to be close to each other (e.g., with an L2 penalty on their difference), allowing them to diverge if the data strongly suggests it. This is a soft constraint.

Major Applications

Convolutional Neural Networks (CNNs) The most prominent example of param-eter sharing is in convolutional layers:

- A single filter (set of weights) is applied across every spatial location in the input.
- This encodes the inductive bias that a pattern (e.g., an edge, a texture) is equally useful regardless of where it appears in the image—a

property called **translation equivariance**.

- Without sharing, a model would need to learn separate edge detectors for every possible position, requiring vastly more parameters and data.
- The reduction is dramatic: for a 3×3 filter applied to a 224×224 image, sharing reduces parameters by a factor of 5,000 compared to learning separate weights for each location.

Recurrent Neural Networks (RNNs) RNNs share weights across time steps:

- The same transition matrix is applied at each time step to update the hidden state.
- This encodes the inductive bias that the dynamics of the system are stationary—the same rules apply regardless of when an event occurs.
- Without sharing, a model processing sequences of length T would need T separate transition matrices, making it impossible to generalize to sequences longer than those seen during training.

Siamese Networks Siamese networks share weights between identical subnetworks:

- Two (or more) copies of the same network process different inputs (e.g., two images, two sentences) and produce embeddings.
- Sharing ensures that the same function maps both inputs to the embedding space, which is essential for tasks like face verification, signature verification, and similarity learning.
- The network learns a distance metric: similar inputs should have similar embeddings, dissimilar inputs should have distant embeddings.

Multi-task Learning with Shared Representations When models are trained on multiple tasks, early layers are often shared:

- All tasks use the same feature extractor (shared parameters), with task-specific layers on top.
- This forces the model to learn features that are useful across all tasks, preventing overfitting to any single task's idiosyncrasies.

Benefits of Parameter Sharing

1. **Dramatic parameter reduction:** Sharing can reduce the number of parameters by orders of magnitude, making large models feasible and reducing memory requirements.

2. **Statistical efficiency:** Fewer parameters means less data needed to train effectively. Shared parameters are estimated using all relevant examples (e.g., all spatial locations, all time steps), providing stronger statistical signal.
3. **Generalization:** By enforcing hard constraints, sharing prevents the model from learning spurious patterns that don't generalize across positions or times.
4. **Architectural inductive bias:** Sharing encodes prior knowledge about the problem structure directly into the architecture, which is often more effective than learning these invariances from data.
5. **Computational efficiency:** Fewer parameters mean faster forward and backward passes, and simpler optimization.

Mathematical Formulation Without sharing, each location l would have its own weight matrix $\boldsymbol{W}^{(l)}$:

$$\boldsymbol{h}^{(l)} = \sigma(\boldsymbol{W}^{(l)}\boldsymbol{x}^{(l)} + \boldsymbol{b})$$

With sharing, all locations use the same $\boldsymbol{W}$:

$$\boldsymbol{h}^{(l)} = \sigma(\boldsymbol{W}\boldsymbol{x}^{(l)} + \boldsymbol{b}) \quad \forall l$$

The gradient aggregates contributions from all locations:

$$\nabla_{\boldsymbol{W}} J = \sum_{l} \nabla_{\boldsymbol{W}^{(l)}} J \Big|_{\boldsymbol{W}^{(l)}=\boldsymbol{W}}$$

This aggregation provides stronger gradient signals and more stable learning.

When to Use Parameter Sharing Parameter sharing is most effective when:

- The problem has known invariances (e.g., translation invariance in images, temporal invariance in sequences).
- Training data is limited relative to model capacity.
- The same basic operation should apply across multiple contexts.
- Memory or computational constraints are severe.

Limitations and Considerations

- **Assumption correctness:** If the invariance assumption is wrong (e.g., patterns are location-dependent), sharing can hurt performance.

- **Flexibility:** Hard sharing may be too restrictive; sometimes soft tying (allowing pa-rameters to differ but penalizing differences) provides a better trade-off.
- **Implementation complexity:** While conceptually simple, implementing custom sharing schemes may require careful engineering.

Parameter Tying: The Soft Constraint Alternative Parameter tying relaxes the equality constraint:

$$\tilde{J}(\boldsymbol{\theta}_A, \boldsymbol{\theta}_B) = J(\boldsymbol{\theta}_A, \boldsymbol{\theta}_B) + \lambda \| \boldsymbol{\theta}_A - \boldsymbol{\theta}_B \|_2^2$$

This allows parameters to diverge if the data strongly suggests they should, while still encouraging similarity. It is useful when domains are related but not identical (e.g., multi-domain learning, transfer learning with fine-tuning).

Architecture	What is Shared	Inductive Bias	Benefit
CNN	Filters across spatial locations	Translation invariance	Handle variable in-put sizes, parame-ter efficiency
RNN	Transition matrix across time steps	Temporal stationarity	Handle variable se-quence lengths
Siamese Net-work	Weights across paral-lel subnetworks	Same function applied to both inputs	Learning compara-tive relationships
Multi-task Net-work	Earl layers across tasks	Shared feature repre-sentations	Transfer learning, reduced overfitting

Table 3.7: Major applications of parameter sharing in deep learning

3.1.10 Sparse Representations

Definition 3.1.10 (Sparse Representations)**.** *Sparse representations are internal feature representations in which most elements are zero (or near-zero) for any given input, and only a small fraction of neurons activate. This forces the model to encode information using a limited number of active components, promoting efficiency, interpretability, and better feature disentanglement.*

Core Principle The human brain processes information using sparse coding—only a small percentage of neurons fire simultaneously. This biological insight has inspired machine learn-ing approaches that encourage sparsity in learned representations. The fundamental idea is that a good representation should be **selective**: each input should activate only the features that are truly relevant, leaving others silent.

Sparse representations offer several advantages:

- **Information disentanglement**: Different factors of variation are represented by distinct, non-overlapping sets of features.
- **Interpretability**: It's easier to understand what each neuron detects when it activates only for specific patterns.
- **Computational efficiency**: Sparse activations require less computation during in-ference.
- **Regularization**: Sparsity prevents the model from relying on too many features si-multaneously, reducing overfitting.

Methods to Encourage Sparsity

L1 Regularization on Activations The most direct approach penalizes the absolute values of hidden unit activations:

$$L_{\text{sparse}} = L_{\text{original}} + \lambda \sum_j |a_j|$$

where a_j are the activations of hidden units. The L1 penalty drives many activations toward exactly zero, producing a sparse representation. This is analogous to L1 regularization on weights, but applied to activations instead.

KL-Divergence Sparsity Penalty (Sparse Autoencoders) Sparse autoencoders introduce a penalty that encourages each hidden unit to have a low average activation across the training dataset. Let $\hat{\rho}_j$ be the average activation of unit j:

$$\hat{\rho}_j = \frac{1}{m} \sum_{i=1} a_j(\boldsymbol{x}^{(i)})$$

We encourage $\hat{\rho}_j$ to match a desired sparsity parameter ρ (a small value, e.g., 0.05) by adding a KL-divergence penalty:

$$L_{sparse} = L_{reconstruction} + \beta \sum_{j} KL(\rho \| \hat{\rho}_j)$$

where

$$KL(\rho \| \hat{\rho}_j) = \rho \log \frac{\rho}{\hat{\rho}_j} + (1 - \rho) \log \frac{1 - \rho}{1 - \hat{\rho}_j}$$

This penalty ensures that each hidden neuron activates only for a small fraction of inputs, forcing the network to develop specialized feature detectors.

K-Sparse Autoencoders A more direct approach: during forward pass, keep only the top k activations and set the rest to zero:

$$a_j^{\text{sparse}} = \begin{cases} a_j \text{ if } a_j \text{ is among top-}k \text{ activations} \\ 0 \end{cases}$$

This guarantees exactly k non-zero activations per example. Gradients are backprop-agated only through the active units, forcing the network to learn representations where information is concentrated in a fixed number of neurons.

Other Approaches

- **Weight decay on hidden units**: Indirectly encourages sparsity by keeping weights small.
- **Thresholding**: Zero out activations below a certain threshold during training.
- **Hard concrete distributions**: Learn which units to keep using stochastic relaxation.

Sparsity in Different Contexts In Autoencoders: Sparse autoencoders learn com-pact, overcomplete representations where each input is encoded using a small subset of the available hidden units. This prevents the autoencoder from learning the identity function and encourages discovery of meaningful features.

In Classification Networks: Sparsity in hidden layers can improve generalization by forcing the network to be selective about which features it uses for each input. ReLU activations naturally produce some sparsity, but explicit penalties increase it further.

In Attention Mechanisms: Sparse attention restricts each position to attend to only a subset of other positions, improving efficiency and sometimes performance. Examples include sparse transformers and block-wise attention.

Theoretical Insights

- **Overcomplete representations:** Sparsity allows using more hidden units than input dimensions (overcomplete representations) without learning the identity mapping, as the representation is constrained to be

sparse.

- **Feature disentanglement:** Sparse representations tend to align with underlying fac-tors of variation in the data, making them more interpretable and useful for downstream tasks.
- **Biological plausibility:** Sparse coding models have been shown to produce receptive fields similar to those in the mammalian visual cortex when trained on natural images.

Practical Considerations

- **Sparsity vs. reconstruction trade-off:** In autoencoders, higher sparsity typically degrades reconstruction quality. The sparsity parameter controls this balance.
- **Sparsity level selection:** The optimal sparsity depends on the data complexity and the number of hidden units. Too much sparsity prevents learning; too little loses the benefits.
- **Computational overhead:** While sparse representations save computation during inference, training with sparsity penalties adds computational cost.
- **Combining with other regularizers:** Sparsity penalties can be combined with weight decay, dropout, or other techniques for stronger regularization.

Limitations

- **Difficult optimization:** Hard sparsity constraints (like k-sparse) create non-differentiable operations, requiring careful handling during backpropagation.
- **Hyperparameter sensitivity:** Performance can be sensitive to sparsity levels and penalty coefficients.
- **Information loss:** If sparsity is too aggressive, the representation may not retain enough information for the task.

Method	Mechanism	Sparsity Type	Typical Use
L1 activation penalty	Add $\sum \lvert a_j \rvert$ to loss	Individual activation sparsity	Feedforward networks
KL-divergence penalty	Match target average activation	Population sparsity	Sparse autoencoders
K-sparse	Keep only top-k activations	Exact per-example sparsity	Feature learning
ReLU activation	Zero negative values	Implicit sparsity	Most modern networks

Table 3.8: Summary of sparsity-inducing techniques

3.1.11 Bagging and Other Ensemble Methods

Definition 3.1.11 (Bagging (Bootstrap Aggregating))**.** *Bagging is an ensemble technique that trains multiple independent models on different bootstrap samples (random samples drawn with replacement) of the original training data, then combines their predictions through averaging (for regression) or voting (for classification). The diversity introduced by different training sets reduces variance while maintaining low bias, making bagging particularly effective for high-variance models like decision trees.*

Core Intuition Individual machine learning models are prone to overfitting—they capture noise specific to the training set. Bagging exploits a simple statistical insight: if we have multiple independent models, averaging their predictions reduces variance without increasing bias. Since training truly independent models on separate datasets is often impractical (we have only one dataset), bagging approximates this by creating multiple slightly different datasets through bootstrap sampling.

How Bagging Works

1. Given a training dataset $\mathbb{D}$ of size m, create k bootstrap samples $\mathbb{D}_1, \mathbb{D}_2, \ldots, \mathbb{D}_k$, each of size m drawn uniformly with replacement from $\mathbb{D}$. Each bootstrap sample contains about 63.2% unique examples from the original dataset, with the rest being duplicates.
2. Train a separate model f_i on each bootstrap sample $\mathbb{D}_i$. Models are trained independently, potentially in parallel.
3. For prediction on a new input $\boldsymbol{x}$, combine all model outputs:
   - **Regression:** $f_{\text{ensemble}}(\boldsymbol{x}) = \frac{1}{k} \sum_i f_i(\boldsymbol{x})$

- **Classification:** $f_{\text{ensemble}}(\boldsymbol{x})$ = majority vote($\{f_i(\boldsymbol{x})\}$)

Why Bagging Works: Bias-Variance Decomposition For regression, the expected squared error of an ensemble of k identically distributed but not necessarily independent models is:

$$\mathrm{E}[(f_{\text{ensemble}} - y)^2] = \underbrace{\text{bias}^2}_{} + \underbrace{\frac{1}{k}\text{variance}}_{} + \underbrace{\left(1 - \frac{1}{k}\right)\text{covariance}}_{}$$

The covariance term captures correlation between models. Bagging reduces correlation through bootstrap sampling, effectively lowering the overall variance. Bias remains un-changed (averaging doesn't introduce additional bias).

Key Properties

- **Variance reduction:** Bagging is most effective for high-variance, low-bias models (e.g., deep decision trees, neural networks). It provides little benefit for already stable models like linear regression.
- **Parallel training:** Unlike boosting, bagging models can be trained independently, making it highly scalable.
- **Out-of-bag evaluation:** Each bootstrap sample leaves out about 36.8% of the original data (the "out-of-bag" samples). These can be used for validation without needing a separate validation set, providing an unbiased performance estimate.

Other Ensemble Methods

Boosting Boosting trains models sequentially, with each new model focusing on exam-ples that previous models misclassified:

- **Sequential training:** Model f_1 is trained on the original data. Examples are then reweighted to emphasize those f_1 got wrong.
- **Adaptive focus:** Model f_2 is trained on this reweighted data, concentrating on hard examples. The process repeats.
- **Weighted combination:** Final prediction combines all models with weights based on their performance.

Popular variants: AdaBoost, Gradient Boosting, XGBoost, LightGBM.

Key difference from bagging: Boosting reduces bias (by focusing on hard examples) and can also reduce variance, but is sequential and more prone to overfitting if not carefully regularized.

Random Forests Random forests combine bagging with random feature selection specifically for decision trees:

- **Bagging applied to trees:** Each tree is trained on a bootstrap sample of the data.
- **Random feature subsets:** At each split, only a random subset of features is consid-ered (typically $\sqrt{p}$ for classification or $p/3$ for regression, where p is the total number

of features).

- **Decorrelating trees:** This further decorrelates the trees, reducing variance beyond standard bagging.

Random forests are remarkably effective with minimal hyperparameter tuning and pro-vide built-in feature importance measures.

Stacking (Stacked Generalization) Stacking learns how to best combine base mod-els using a meta-model:

- **Level-0 models (base learners):** Train diverse models (e.g., SVM, random forest, neural network) on the training data.
- **Hold-out predictions:** Use cross-validation to generate predictions from base models on held-out data (prevents leakage).
- **Level-1 model (meta-learner):** Train a model (often linear regression or logistic regression) to predict the true target using the base model predictions as features.
- **Final prediction:** New examples go through base models, their predictions become input to the meta-model.

Stacking leverages the strengths of different model types—if base models make errors in different ways, the meta-model can learn to combine them optimally.

Method	**Training**	**Key Mechanism**	**Primary Benefit**
Bagging	Parallel	Bootstrap sampling + averaging/voting	Variance reduction
Boosting	Sequential	Reweight misclassified examples	Bias reduction
Random Forest	Parallel	Bootstrap + random feature selection	Decorrelated trees, variance reduction
Stacking	Two-stage	Meta-learner on base predictions	Leverage diverse model strengths

Table 3.9: Comparison of major ensemble methods

Comparison of Ensemble Methods Practical Considerations

- **Diversity is key:** Ensembles work best when individual models make different errors. Diversity can come from different data (bagging), different features (random forests), different algorithms (stacking), or different training dynamics.
- **Computational cost:** Ensembles require training and storing multiple models, in-creasing computational requirements linearly with ensemble size.
- **Diminishing returns:** Beyond a certain ensemble size (often 10-100 models), addi-tional models provide minimal improvement.
- **Interpretability:** Ensembles are generally less interpretable than single models, though random forests offer feature importance measures.

Ensemble Methods as Regularization Ensemble methods can be viewed as a form of regularization because:

- They reduce variance without increasing bias, similar to how explicit regularizers con-strain model complexity.
- The averaging process smooths out predictions, preventing overconfident errors.
- They effectively expand the hypothesis space while preventing reliance on any single spurious pattern.

3.1.12 Dropout

Definition 3.1.12 (Dropout)**.** *Randomly setting a fraction p of hidden units to zero during training.*

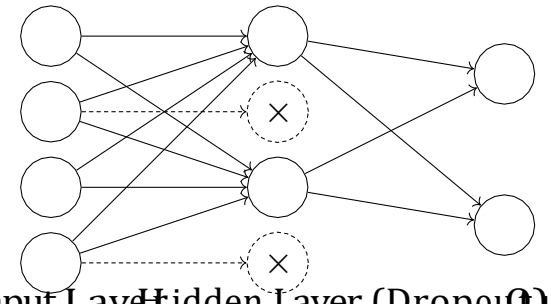


Figure 3.2: Dropout: Randomly dropping neurons during training prevents co-

adaptation.

During test time, scale weights by p or use inverted dropout:

$$\text{Test output} = p \cdot \text{Network output}$$

3.1.13 Adversarial Training

Definition 3.1.13 (Adversarial Examples)**.** *Inputs designed to cause a model to make mis-takes, created by adding small, carefully crafted perturbations.*

Adversarial training objective:

$$\min_{\boldsymbol{\theta}} \mathrm{E}_{(\boldsymbol{x},y)\sim \mathrm{D}} \max_{\|\boldsymbol{\delta}\|\leq\epsilon} L(\boldsymbol{\theta}, \boldsymbol{x}+\boldsymbol{\delta}, y)$$

3.1.14 Tangent Distance, Tangent Prop and Manifold Tangent Classifier

Definition 3.1.14 (Tangent Distance)**.** *Distance between two points along the data manifold rather than Euclidean distance.*

Tangent propagation adds penalty for invariance to known transformations:

$$\tilde{J}(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda \sum_{i} \|\nabla_{\boldsymbol{x}} f(\boldsymbol{x}_i)^T \boldsymbol{t}_i\|^2$$

where $\boldsymbol{t}_i$ are tangent vectors of transformations.

3.2 Optimization for Training Deep Models

3.2.1 Pure Optimization

Mathematical Formulation: Optimization seeks to find parameters $\boldsymbol{\theta}$ that minimize a loss function $\mathrm{L}(\boldsymbol{\theta})$:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathrm{L}(\boldsymbol{\theta}) = \arg\min_{\boldsymbol{\theta}} \frac{1}{N} \sum_{i=}^{N} \ell(f(\boldsymbol{x}; \boldsymbol{\theta}), y)$$

where ℓ is the per-example loss, f is the model, and N is dataset size.
Objective in Deep Learning: Unlike convex optimization which guarantees global min-imum, deep neural networks have non-convex loss

surfaces with multiple local minima and saddle points. The goal is to find parameters that achieve low training loss and good gener-alization.

Types of Optimization Problems:

- **Convex vs. Non-convex:**
  - Convex: Any local minimum is global minimum. Example: Linear regression with MSE.
  - Non-convex: Multiple local minima. Example: Any neural network with hidden layers.
- **Constrained vs. Unconstrained:**
  - Unconstrained: Parameters can take any real value. Most DL falls here.
  - Constrained: Parameters must satisfy constraints like $\|\boldsymbol{\theta}\|_2 \leq C$ (weight decay interpretation).
- **Stochastic vs. Deterministic:**
  - Deterministic: Uses full dataset for gradient computation. Computationally ex-pensive.
  - Stochastic: Uses random mini-batches. Efficient and provides implicit regulariza-tion.

Empirical Risk Minimization:

$$\min_{\boldsymbol{\theta}} \mathbb{E}_{(\boldsymbol{x},y)\sim p_{\text{data}}}[\ell(\boldsymbol{x}, y; \boldsymbol{\theta})] \approx \min_{\boldsymbol{\theta}} \frac{1}{N} \sum_{i=} \ell(\boldsymbol{x}_i, y_i; \boldsymbol{\theta})$$

The gap between empirical risk and true risk is the generalization error.

3.3 Challenges in Neural Network Optimization

Relevant for Q.No. 8

Deep neural networks present unique optimization challenges:

1. **Ill-conditioning:**
   - The Hessian matrix $\boldsymbol{H}_{\theta} = \nabla^2 L$ has a poor condition number (ratio of largest to smallest eigenvalue).
   - Causes gradients to be less informative and optimization to zig-zag.
   - SGD becomes very slow in directions of low curvature.
2. **Local Minima:**
   - Non-convex loss surfaces contain many local minima.
   - In high-dimensional spaces, most local minima are equivalent in value and not a major problem.
   - The real concern is finding a minimum that generalizes well.
3. **Saddle Points:**
   - Points where $\nabla_{\theta} L = 0$ but Hessian has both positive and negative eigenvalues.
   - Much more common in high dimensions than local minima.
   - Gradient is flat, causing optimization to stall.
   - Ratio of saddle points to local minima grows exponentially with dimension.
4. **Vanishing and Exploding Gradients:**
   - During backpropagation, gradients are multiplied by many Jacobian matrices.
   - If eigenvalues < 1, gradients vanish exponentially with depth.
   - If eigenvalues > 1, gradients explode exponentially.
   - Particularly severe in RNNs for long sequences.
5. **Long-Term Dependencies:**
   - RNNs struggle to learn connections between events separated by many time steps.
   - Backpropagated gradients through time either vanish or explode.
   - LSTMs and GRUs were designed to mitigate this.
6. **Initialization Sensitivity:**
   - Starting point heavily influences convergence speed.

- Poor initialization can lead to vanishing/exploding gradients from the start.
- Different activation functions require different initialization strategies.

7. **Plateaus and Cliffs:**

- Plateaus: Regions where gradient is near-zero for extended periods.
- Cliffs: Regions of extreme slope causing gradient explosions.

3.3.1 L1 and L2 Regularization

Relevant for Q.No. 1 and 2

L2 Regularization (Ridge, Weight Decay):

$$\tilde{L}(\boldsymbol{\theta}) = L(\boldsymbol{\theta}) + \frac{\lambda}{2}\|\boldsymbol{\theta}\|_2^2 = L(\boldsymbol{\theta}) + \frac{\lambda}{2}\sum_{i=} \theta^2$$

Mathematical Derivation of Weight Decay (Q.No. 1b):

$$\nabla_{\boldsymbol{\theta}} \tilde{L} = \nabla_{\boldsymbol{\theta}} L + \lambda \boldsymbol{\theta}$$

SGD update with learning rate ϵ:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon(\nabla_{\boldsymbol{\theta}} L + \lambda \boldsymbol{\theta})$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \nabla_{\boldsymbol{\theta}} L -$$

$$\epsilon\lambda\boldsymbol{\theta} \ \boldsymbol{\theta} \leftarrow (1 -$$

$$\epsilon\lambda)\boldsymbol{\theta} - \epsilon \nabla_{\boldsymbol{\theta}} L$$

The term $(1 - \epsilon\lambda)$ multiplies weights by a factor slightly less than 1 at each step, hence
weight decay. This encourages weights to remain small.

Bayesian Interpretation: L2 regularization is equivalent to MAP estimation with a Gaus-sian prior $\boldsymbol{\theta} \sim \mathcal{N}(0, 1/\lambda)$.

L1 Regularization (Lasso):

$$\tilde{L}(\boldsymbol{\theta}) = L(\boldsymbol{\theta}) + \lambda\|\boldsymbol{\theta}\|_1 = L(\boldsymbol{\theta}) + \lambda \sum_{i=1} |\theta_i|$$

Bayesian Interpretation: L1 regularization is equivalent to MAP estimation with a Laplace prior.

Effect of L1 vs. L2 Penalties (Q.No. 2):

Property	L2 Regularization	L1 Regularization
Penalty term	$\frac{\lambda}{2}\sum \theta^2$	$\lambda \sum \lvert\theta_i\rvert$
Differentiability	Differentiable everywhere	Not differentiable at zero
Solution	Non-sparse (all weights small)	Sparse (many weights exactly zero)
Feature selection	No	Yes
Prior	Gaussian	Laplace
Gradient	$\lambda\theta_i$ (smooth)	$\lambda \cdot \text{sign}(\theta_i)$ (constant magnitude)

Computational Example: For $\boldsymbol{\theta} = [2, -1, 0.5, \ 0.2, 0]$ with $\lambda = 0.1$: L2 penalty calculation:

$$\frac{\lambda}{2} \sum \theta_i^2 = 0.05 \times (4 + 1 + 0.25 + 0.04 + 0) = 0.05 \times 5.29 = 0.2645$$

L1 penalty calculation:

$$\lambda \sum |\theta_i| = 0.1 \times (2 + 1 + 0.5 + 0.2 + 0) = 0.1 \times 3.7 = 0.37$$

Notice the zero weight contributes nothing to either penalty, but L1 penalty is larger because it doesn't square small values.

Why L1 Promotes Sparsity: The subgradient of L1—at zero is $[-\lambda, \lambda]$, allowing weights to become exactly zero. For L2, the gradient at zero is zero, so weights approach zero asymptotically but never become exactly zero.

3.3.2 Dropout

Relevant for Q.No. 3

Definition: Dropout randomly **drops out** (sets to zero) a proportion p of neurons dur-ing training. Each neuron is independently retained with probability $\overline{p}$ (or dropped with probability $1 - p$).

Detailed Algorithm:

Training Phase:

1. For each layer, generate a binary mask $\boldsymbol{m}$ Bernoulli(p) of same dimension as the layer output.
2. Apply mask: $\boldsymbol{h}_{\text{dropped}} = \boldsymbol{m} \odot \boldsymbol{h}$ where $\odot$ is element-wise multiplication.
3. Scale the output by $\frac{1}{p}$ to maintain expected magnitude: $\boldsymbol{h}_{\text{final}} = \frac{1}{p}\boldsymbol{h}_{\text{dropped}}$.
4. Only non-dropped neurons participate in forward pass and backpropagation.

Testing Phase:

- No dropout is applied (all neurons are used).
- Weights are multiplied by p (or equivalently, the forward pass uses all neurons with full weights and no scaling).
- This ensures the expected output matches training phase.

Impact on Generalization (Q.No. 3):

1. **Ensemble Effect:**

- With n neurons, dropout trains 2^n different thinned networks that share weights.

- At test time, using all neurons approximates averaging the predictions of all these ensembles.
- Geometric average of ensemble predictions approximates the true model average.

2. **Preventing Co-adaptation:**
   - Neurons cannot rely on specific other neurons being present.
   - Each neuron must learn features that are useful regardless of which other neurons are active.
   - This forces the network to learn more robust, independent representations.
3. **Implicit Regularization:**
   - Dropout acts as a form of regularization with effect similar to L2.
   - The regularization strength is adaptive: weights that are large receive more reg-ularization.
   - For a single linear layer, dropout is equivalent to L2 with adaptive coefficient.
4. **Bayesian Interpretation:**
   - Dropout can be viewed as approximate Bayesian inference over model weights.
   - The stochasticity during training approximates sampling from the posterior.
   - This provides uncertainty estimates in predictions.

Mathematical Analysis: For a single neuron with input $\boldsymbol{x}$, weight $\boldsymbol{w}$, and output $y = \boldsymbol{w}^T\boldsymbol{x}$:
Without dropout: Expected output $\mathrm{E}[y] = \boldsymbol{w}^T\boldsymbol{x}$ With dropout (retain probability p):

$$\mathrm{E}[y] = p \cdot \frac{1}{p}\boldsymbol{w}^T\boldsymbol{x} = \boldsymbol{w}^T\boldsymbol{x}$$

So expectation is preserved. Variance introduced by dropout:

$$\mathrm{Var}(y) = \frac{1-p}{p}(\boldsymbol{w}^T\boldsymbol{x})^2$$

This variance acts as a regularizer.

Practical Considerations:

- Typical dropout rates: $p = 0.5$ for hidden layers, $p = 0.8$ for input layers.
- Too much dropout (p too small) can cause underfitting.
- Too little dropout (p too large) provides insufficient regularization.

- Spatial dropout drops entire feature maps in CNNs.
- Dropout is less effective with batch normalization.

3.3.3 Dataset Augmentation

Relevant for Q.No. 4

Definition: Dataset augmentation generates new training examples by applying label-preserving transformations to existing data, effectively increasing dataset size and diversity. **Motivation:** The ideal model should be invariant to irrelevant variations in input. Aug-mentation teaches these invariances directly.

Common Augmentation Techniques by Domain:

Image Data:

- **Geometric Transformations:**
  - Rotation (small angles, e.g., $\pm 10^\circ$)
  - Translation (shifting by few pixels)
  - Scaling (zooming in/out)
  - Horizontal/Vertical flipping (for symmetric objects)
  - Cropping (random crops from image)
  - Shearing (distorting perspective)
- **Photometric Transformations:**
  - Brightness adjustment
  - Contrast adjustment
  - Saturation adjustment
  - Hue shifting
  - Color jittering
  - Gaussian noise addition
  - Blurring (Gaussian, median)
- **Advanced Techniques:**
  - **MixUp:** $\tilde{\boldsymbol{x}} = \lambda \boldsymbol{x}_i + (1-\lambda)\boldsymbol{x}_j, \tilde{y} = \lambda y_i + (1-\lambda) y_j$
  - **CutOut:** Randomly mask out square regions of the image
  - **CutMix:** Cut and paste patches between images, mix labels
  - **RandAugment:** Automatically search for optimal augmentation policies

Text Data:

- Synonym replacement
- Back-translation (translate to another language and back)
- Word dropout (randomly remove words)
- Contextual augmentation (using language models)
- Sentence shuffling

Speech/Audio Data:

- Time stretching (speed up/slow down)
- Pitch shifting
- Adding background noise
- SpecAugment (masking frequency and time channels)
- Volume perturbation

Why Augmentation Works (Q.No. 4):

1. **Increased Effective Dataset Size:**
   - Reduces overfitting by providing more diverse examples.
   - Each epoch sees different variations of the same images.
   - Effectively infinite dataset if augmentations are random.
2. **Learning Invariances:**
   - Teaches model to be invariant to irrelevant transformations.
   - Object classification should be invariant to rotation, translation, color.
   - Without augmentation, model must learn these from limited data.
3. **Improved Robustness:**
   - Model becomes robust to corruptions and perturbations.
   - Better performance on out-of-distribution examples.
   - Adversarial robustness can improve.
4. **Data-Efficient Learning:**
   - Achieves better performance with limited labeled data.
   - Particularly important in medical imaging, rare species classification.
5. **Regularization Effect:**
   - Augmentation acts as implicit regularization.

- Prevents memorization of specific training examples.

Mathematical Formulation: Given training set $D = \{(\boldsymbol{x}_i, y_i)\}^N$, augmentation defines a distribution $T(\boldsymbol{x}|\boldsymbol{x}_i)$ of transformed versions. Training objective becomes:

$$\min_{\boldsymbol{\theta}} \sum_{i=1}^{N} \mathrm{E}_{\tilde{\boldsymbol{x}} \sim T(\cdot|\boldsymbol{x}_i)}[\ell(f(\tilde{\boldsymbol{x}}; \boldsymbol{\theta}), y_i)]$$

Practical Considerations:

- Augmentations should be label-preserving (e.g., rotating a **6** in MNIST could make it a **9**).
- Too aggressive augmentation can harm performance.
- Test time augmentation (TTA): Apply augmentations at test time and average predic-tions.

3.3.4 Early Stopping

Relevant for Q.No. 7

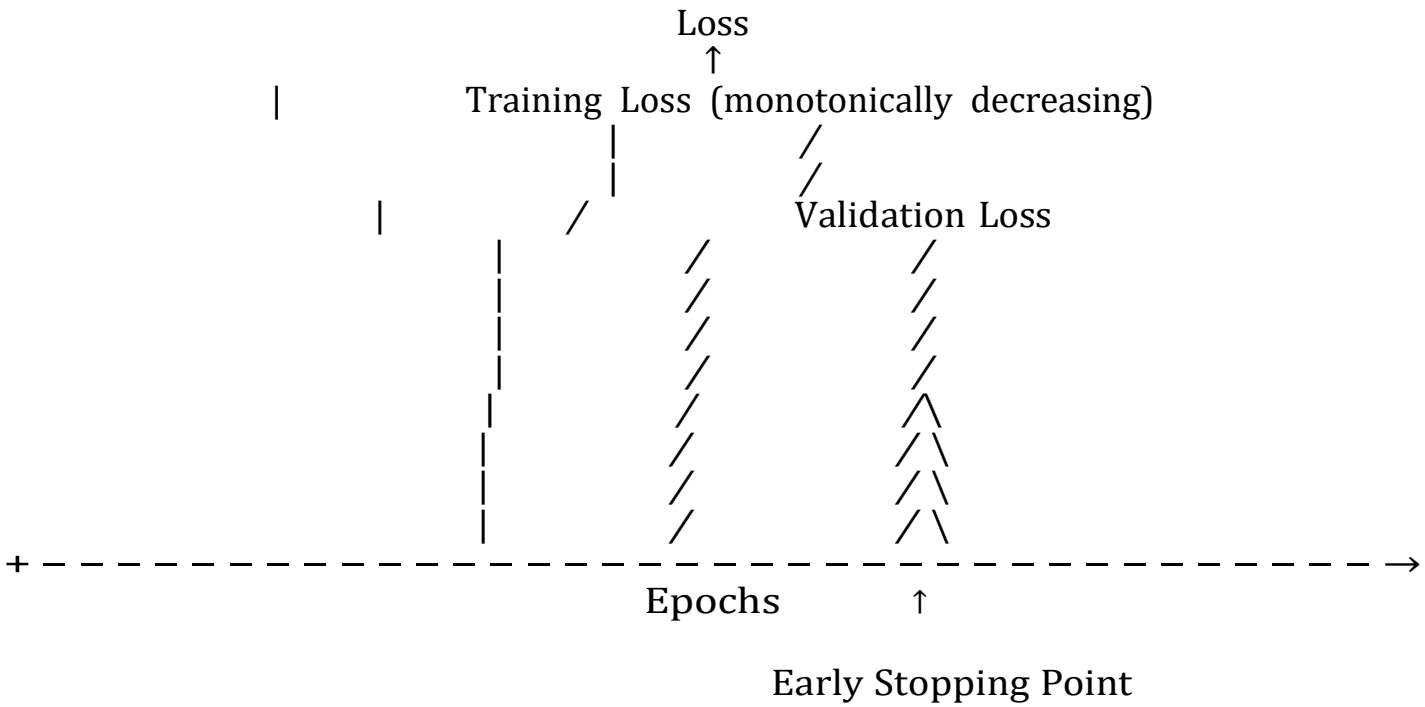
Definition: Early stopping halts training when performance on a validation set stops im-proving, preventing overfitting.

Detailed Algorithm:

1. Split data into training set D_{train} and validation set D_{val}.
2. Initialize parameters $\boldsymbol{\theta}_0$, best parameters $\boldsymbol{\theta}_{\text{best}} = \boldsymbol{\theta}_0$, best validation loss $L_{\text{best}} = \infty$, patience counter $p = 0$.
3. For each epoch $t = 1, 2, \ldots$:
   (a) Train one epoch on D_{train}, update parameters to $\boldsymbol{\theta}_t$.
   (b) Compute validation loss $L_{\text{val}}(\boldsymbol{\theta}_t)$ on D_{val}.
   (c) If $L_{\text{val}}(\boldsymbol{\theta}_t) < L_{\text{best}}$:
      - Update $\boldsymbol{\theta}_{\text{best}} = \boldsymbol{\theta}_t$, $L_{\text{best}} = L_{\text{val}}(\boldsymbol{\theta}_t)$
      - Reset $p = 0$

   (d) Else:
      - Increment $p = p + 1$
      - If $p \geq$ patience: Stop training
4. Restore parameters to $\boldsymbol{\theta}_{\text{best}}$.

Graphical Interpretation (Q.No. 7):



Training loss continues to decrease. Validation loss decreases initially, reaches a minimum, then increases as the model begins to overfit and memorize noise. Early stopping at the minimum validation loss point.

Mathematical Interpretation: Early stopping can be viewed as a form of regularization. For a linear model with quadratic loss, early stopping is equivalent to L2 regularization with $\lambda \propto 1/(\text{stopping time})$.

Benefits (Q.No. 7):

- **Prevents Overfitting:** Stops before memorizing noise in training data.
- **Automatic Model Selection:** Determines optimal model complexity without ex-plicit architecture search.
- **Computational Efficiency:** Reduces unnecessary training time.
- **No Architectural Changes:** Can be applied to any model without modification.
- **Theoretically Grounded:** Equivalent to L2 regularization in some settings.

Practical Considerations:

- **Patience:** Number of epochs to wait before stopping. Too small: premature stopping. Too large: wastes computation.

- **Minimum Delta:** Minimum improvement required to reset patience. Prevents stop-ping due to noise.

- **Validation Metric:** Can use accuracy, F1, etc., not just loss.
- **Restore Best Weights:** Always restore to best validation state after stopping.

—

3.3.5 Ensemble Methods

Relevant for Q.No. 5

Definition: Ensemble methods combine multiple models to produce a single, more powerful predictor. The ensemble typically outperforms any individual model.

Bagging (Bootstrap Aggregating) - Q.No. 5:

Algorithm:

1. Given training set $D = \{(\boldsymbol{x}_i, y_i)\}_{i=}^{N}$ with N examples.
2. For $k = 1$ to K:
   (a) Create bootstrap sample D_k by sampling N examples from with replacement (some examples repeated, some omitted).
   (b) Train model $f_k(\boldsymbol{x}; \boldsymbol{\theta}_k)$ on D_k.
3. For prediction:
   - Regression: $\bar{f}(\boldsymbol{x}) = \frac{1}{K} \sum_{k=}^{K} f_k(\boldsymbol{x})$
   - Classification: $\bar{f}(\boldsymbol{x})$ = majority vote($\{f_k(\boldsymbol{x})\}$) or average probabilities

Mathematical Analysis of Why Bagging Works:

Consider regression with true function $f^*(\boldsymbol{x})$. Each model's prediction can be written as $f_k(\boldsymbol{x}) = f^*(\boldsymbol{x}) + \epsilon_k(\boldsymbol{x})$ where ϵ_k is error with $\mathrm{E}[\epsilon_k] = 0$ and variance σ^2.

Expected squared error of a single model:

$$\mathrm{E}[(f_k - f^*)^2] = \mathrm{E}[\epsilon_k^2] = \mathrm{Var}(\epsilon_k) = \sigma^2$$

Expected squared error of the ensemble average:

$$\mathrm{E}\left[\left(\frac{1}{K}\sum_{k=}^{K} f_k - f^*\right)^2\right] = \mathrm{E}\left[\left(\frac{1}{K}\sum_{k=}^{K} \epsilon_k\right)^2\right]$$

$$= \frac{1}{K^2}\sum_{k=1}^{K}\sum_{j=1}^{K} \mathrm{E}[\epsilon_k \epsilon_j] = \frac{1}{K^2}\left[\sum_{k=1}^{K} \mathrm{E}[\epsilon^2] + \sum_{k \neq j} \mathrm{E}[\epsilon_k \epsilon_j]\right]$$

If errors are uncorrelated ($\mathrm{E}[\epsilon_k \epsilon_j] = 0$ for $k \neq j$):

k

$$\mathrm{E}[(f_{\mathrm{ens}} - f^*)^2] = \frac{1}{K^2} \cdot K\sigma^2 = \frac{\sigma^2}{K}$$

Variance reduces by factor K! In practice, errors are correlated, so reduction is less dramatic.

Bias-Variance Decomposition: Total error = bias^2 + variance + irreducible noise. Bag-ging primarily reduces variance without increasing bias.
Why Bootstrap? Bootstrap samples create diversity among models. Without diversity (training all on same data), errors would be perfectly correlated and averaging would provide no benefit.

Other Ensemble Methods:

- **Boosting:**
  - Sequentially trains models, each focusing on mistakes of previous ones.
  - Examples: AdaBoost, Gradient Boosting, XGBoost.
  - Reduces both bias and variance.
- **Stacking (Stacked Generalization):**
  - Train multiple base models on training data.
  - Train a meta-model on the outputs of base models using validation data.
  - Meta-model learns to combine base model predictions optimally.
- **Model Averaging:**
  - Simple average of predictions from different architectures.
  - Average of checkpoints from same training trajectory (Polyak averaging).
- **Snapshot Ensembles:**
  - Train one model with cyclical learning rates.
  - Save model at different cycle minima.
  - Ensemble these snapshots.

Practical Considerations (Q.No. 5):

- Ensembles require training multiple models (K times more computation).
- Diversity is key: use different architectures, initializations, or data subsets.
- In deep learning, even models with same architecture but different random seeds pro-vide useful diversity.
- Monte Carlo Dropout can approximate ensemble at test time without extra training.

—

3.3.6 Parameter Sharing and Sparse Representations

Relevant for Q.No. 6

Parameter Sharing:

Definition: Parameter sharing forces sets of parameters to be equal across different parts of the model, reducing the number of independent parameters.

Example - Convolutional Neural Networks: In CNNs, a convolutional kernel applied at every spatial location shares the same weights. For an input of size $H \times W \times C_{\text{in}}$ and C_{out} filters of size $K \times K$:

- **Without sharing (fully connected):** Parameters $= H \times W \times C_{\text{in}} \times C_{\text{out}}$ (enormous).
- **With sharing (convolutional):** Parameters $= K \times K \times C_{\text{in}} \times C_{\text{out}}$ (independent of input size).

Benefits of Parameter Sharing:

1. **Translation Equivariance:** If f is convolution, $f(\text{shift}(\boldsymbol{x})) = \text{shift}(f(\boldsymbol{x}))$. Feature detection regardless of position.
2. **Parameter Efficiency:** Dramatically reduces parameters from $O(\text{input size} \times \text{output size})$ to $O(\text{kernel size})$.
3. **Statistical Efficiency:** Learns features with fewer examples because the same feature detector is reused everywhere.
4. **Computational Efficiency:** Convolutions can be implemented efficiently with FFT or im2col.

Other Examples of Parameter Sharing:

- **Recurrent Neural Networks:** Same weight matrix applied at each time step.
- **Siamese Networks:** Two branches share weights for similarity learning.
- **Multi-task Learning:** Shared layers across tasks with task-specific heads.

Sparse Representations:

Definition: Sparse representations are representations where most elements are zero or near-zero, activated only by specific inputs.

Benefits of Sparsity (Q.No. 6):

1. **Information Disentangling:** Features become more selective and interpretable. Each feature responds to a specific pattern.
2. **Computational Efficiency:** Can use sparse matrix operations,

reducing FLOPs and memory.

3. **Memory Efficiency:** Store only non-zero elements and their indices.
4. **Regularization:** Sparsity acts as a regularizer, reducing model capacity.
5. **Biological Plausibility:** Neurons in brain exhibit sparse activity.

Achieving Sparsity:

1. **Activation Functions:**
   - ReLU: $f(x) = \max(0, x)$ naturally produces zeros for negative inputs.
   - With ReLU, approximately 50% of neurons are active at any time.
2. **L1 Regularization:**
   - Penalty $\lambda \|\boldsymbol{\theta}\|_1$ encourages weights to become exactly zero.
   - The subgradient at zero allows weights to be set to zero.
3. **Sparse Autoencoders:**
   - Add sparsity penalty on hidden activations: $\lambda \sum |h_i|$.
   - Or use KL divergence to enforce target sparsity level.
4. **Dropout:** Randomly sets activations to zero during training.
5. **Weight Pruning:** After training, remove weights below threshold and fine-tune.

Mathematical Formulation of L1-induced Sparsity: For L1 regularization $L + \lambda \sum |\theta_i|$, the proximal operator is soft thresholding:

$$\text{prox}_\lambda(\theta) = \text{sign}(\theta) \max(|\theta| - \lambda, 0)$$

This sets weights with magnitude $< \lambda$ to exactly zero.

3.3.7 AdaGrad

Relevant for Q.No. 9

Motivation: Different parameters may need different learning rates. Features that appear infrequently should have larger updates when they appear.

Update Rules:

$$\boldsymbol{g}_t \leftarrow \nabla_{\boldsymbol{\theta}} \mathrm{L}(\boldsymbol{\theta}_{t-1})$$
$$\boldsymbol{r}_t \leftarrow \boldsymbol{r}_{t-1} + \boldsymbol{g}_t \odot \boldsymbol{g}_t$$
$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}_t}} \odot \boldsymbol{g}_t$$

where:

- $\boldsymbol{g}_t$ is gradient at step t

- $\boldsymbol{r}_t$ accumulates squared gradients (element-wise)
- ϵ is global learning rate (typically 0.01)
- δ is small constant (e.g., 10^{-8}) for numerical stability
- $\odot$ is element-wise multiplication
- Division and square root are element-wise

Detailed Explanation:

Accumulation Term $\boldsymbol{r}_t$*:*

$$r_{t,i} = \sum_{\tau=1}^{t} g_{\tau,i}^2$$

This is the sum of squared gradients for parameter *i* from all previous steps.

Effective Learning Rate:

$$\eta_{t,i} = \frac{\epsilon}{\delta + \sqrt{r_{t,i}}}$$

- Parameters with large historical gradients get small effective learning rates.
- Parameters with small historical gradients get large effective learning rates.
- The learning rate is monotonically decreasing for all parameters.

Properties:

- **Per-parameter learning rates:** Automatically adapts to each parameter's history.
- **No manual tuning:** Reduces sensitivity to global learning rate choice.
- **Good for sparse features:** Infrequent features get larger updates when they appear.
- **Well-suited for convex problems:** Theoretical convergence guarantees for convex optimization.

Drawback - Monotonic Learning Rate Decay:

$$r_{t+1,i} = r_{t,i} + g_{t+1,i}^2 \geq r_{t,i} \implies \eta_{t+1,i} \leq \eta_{t,i}$$

The learning rate decreases monotonically for all parameters. After many iterations, learning rates become extremely small, and learning effectively stops. This is problematic for non-convex deep learning where we may need to continue learning indefinitely.

Example: For a parameter with gradient history [0.1, 0.2, 0.15, 0.3], $\epsilon = 0.01$, $\delta = 10^{-8}$:

$$r_4 = 0.01 + 0.04 + 0.0225 + 0.09 = 0.1625$$

$$\eta_4 = \frac{0.01}{\sqrt{0.1625}} \approx \frac{0.01}{0.403} \approx 0.0248$$

—

3.3.8 RMSProp

Relevant for Q.No. 9

Motivation: Fix AdaGrad's monotonically decreasing learning rates by using exponentially weighted moving average instead of sum.

Update Rules:

$$\boldsymbol{g}_t \leftarrow \nabla_{\boldsymbol{\theta}} \mathrm{L}(\boldsymbol{\theta}_{t-1})$$

$$\boldsymbol{r}_t \leftarrow \rho \boldsymbol{r}_{t-1} + (1-\rho)\boldsymbol{g}_t \odot \boldsymbol{g}_t$$

$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \frac{\sqrt{\epsilon}}{\delta + \boldsymbol{r}_t} \odot \boldsymbol{g}_t$$

where:

- ρ is decay rate (typically 0.9 or 0.99)
- $\boldsymbol{r}_t$ is exponentially weighted moving average of squared gradients
- Other notation same as AdaGrad

Detailed Explanation:

Moving Average Term $\boldsymbol{r}_t$:

$$r_{t,i} = \rho r_{t-1,i} + (1-\rho) g_{t,i}^2$$

This gives more weight to recent gradients and forgets old ones. The decay rate ρ controls the memory length:

- ρ close to 1: Long memory, smooth but slow to adapt
- ρ close to 0: Short memory, adapts quickly but noisy

Effective Learning Rate:

$$\eta_{t,i} = \frac{\epsilon}{\sqrt{\delta + r_{t,i}}}$$

Unlike AdaGrad, $r_{t,i}$ can increase or decrease over time, so learning rates can adapt in both directions.

Properties:

- **No monotonic decay:** Learning rates can increase if gradients become small.
- **Adapts to changing landscapes:** Quickly adjusts when loss surface changes.
- **Good for non-stationary objectives:** Particularly effective for RNNs.
- **Popular default:** Often works well without much tuning.

Example: For a parameter with $\rho = 0.9$, initial $r_0 = 0$, gradients $[0.1, 0.2, 0.15, 0.3]$:

$$r_1 = 0.9 \times 0 + 0.1 \times 0.01 = 0.001$$
$$r_2 = 0.9 \times 0.001 + 0.1 \times 0.04 = 0.0009 + 0.004 = 0.0049$$
$$r_3 = 0.9 \times 0.0049 + 0.1 \times 0.0225 = 0.00441 + 0.00225 = 0.00666$$
$$r_4 = 0.9 \times 0.00666 + 0.1 \times 0.09 = 0.005994 + 0.009 = 0.014994$$

3.3.9 Adam (Adaptive Moment Estimation)

Relevant for Q.No. 9 and 10

Motivation: Combine momentum (for faster convergence) with RMSProp's adaptive learn-ing rates.

Complete Adam Algorithm (Q.No. 9):

Input: Learning rate ϵ, decay rates $\beta_1, \beta_2 \in [0, 1)$, small constant δ, initial parameters $\boldsymbol{\theta}_0$.
Initialize: $\boldsymbol{m}_0 = \mathbf{0}$ (first moment), $\boldsymbol{v}_0 = \mathbf{0}$ (second moment), $t = 0$.
While not converged:

$$t \leftarrow t + 1$$
$$\boldsymbol{g}_t \leftarrow \nabla_{\boldsymbol{\theta}} \mathrm{L}(\boldsymbol{\theta}_{t-1})$$
$$\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t$$
$$\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\boldsymbol{g}_t \odot \boldsymbol{g}_t$$
$$\hat{\boldsymbol{m}}_t \leftarrow \frac{\boldsymbol{m}_t}{1 - \beta_1^t} \quad \text{(Bias correction for first moment)}$$
$$\hat{\boldsymbol{v}}_t \leftarrow \frac{\boldsymbol{v}_t}{1 - \beta_2^t} \quad \text{(Bias correction for second moment)}$$
$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \epsilon \frac{\hat{\boldsymbol{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t} + \delta}$$

Detailed Explanation:

First Moment (Momentum):

$$\boldsymbol{m}_t = \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t$$

This is exponentially weighted moving average of gradients (momentum term). Typical $\beta_1 = 0.9$.

Second Moment (Adaptive Learning Rate):

$$\boldsymbol{v}_t = \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\boldsymbol{g}_t \odot \boldsymbol{g}_t$$

This is exponentially weighted moving average of squared gradients (like RMSProp). Typical $\beta_2 = 0.999$.

Bias Correction: Since $\boldsymbol{m}_0$ and $\boldsymbol{v}_0$ are initialized to zero, early estimates are biased toward zero. Bias correction counteracts this:

$$\hat{\boldsymbol{m}}_t = \frac{\boldsymbol{m}_t}{1 - \beta_1^t} \qquad \hat{\boldsymbol{v}}_t = \frac{\boldsymbol{v}_t}{1 - \beta_2^t}$$

,

As t increases, $1-\beta^t \to 1$, so correction becomes negligible.

Parameter Update:

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \epsilon \frac{\hat{\boldsymbol{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t} + \delta}$$

This divides the momentum-corrected gradient by the RMS of recent gradients, providing adaptive learning rates.

One Adam Step Example (Q.No. 10):

Given:

- $\boldsymbol{\theta}_{t-1} = [1.0, -0.5]$
- $\boldsymbol{g}_t = [0.2, -0.1]$
- $\beta_1 = 0.9, \beta_2 = 0.999$
- $\epsilon = 0.001, \delta = 10^{-8}$
- $\boldsymbol{m}_{t-1} = [0.1, -0.05]$
- $\boldsymbol{v}_{t-1} = [0.04, 0.01]$
- $t = 10$

Step 1: Update moments

$\boldsymbol{m}_t = 0.9 \times [0.1, -0.05] + 0.1 \times [0.2, -0.1] = [0.09 + 0.02, -0.045 - 0.01] = [0.11, -0.055]$

$\boldsymbol{v}_t = 0.999 \times [0.04, 0.01] + 0.001 \times [0.04, 0.01] = [0.03996 + 0.00004, 0.00999 + 0.00001] = [0.04, 0.01]$

Step 2: Bias correction

$$\hat{\boldsymbol{m}}_t = \frac{[0.11, -0.055]}{1 - 0.9^{10}} = \frac{[0.11, -0.055]}{1 - 0.3487} = \frac{[0.11, -0.055]}{0.6513} \approx [0.169, -0.084]$$

$$\hat{\boldsymbol{v}}_t = \frac{[0.04, 0.01]}{1 - 0.999^{10}} = \frac{[0.04, 0.01]}{1 - 0.990} = \frac{[0.04, 0.01]}{0.01} = [4.0, 1.0]$$

Step 3: Parameter update

$$\boldsymbol{\theta}_t = [1.0, -0.5] - 0.001 \times \left[\frac{0.169}{\sqrt{4.0} + 10^{-8}}, \frac{-0.084}{\sqrt{1.0} + 10^{-8}}\right.$$

$$= [1.0, -0.5] - 0.001 \times \frac{0.169}{2.0}, \frac{-0.084}{1.0}$$

$$= [1.0, -0.5] - 0.001 \times [0.0845, -0.084]$$

$$= [1.0, -0.5] - [0.0000845, -0.000084]$$

$$= [0.9999155, -0.499916]$$

Comparison of Algorithms (Q.No. 9):

Algorithm	Momentum	Adaptive LR	Bias Correction
SGD	No	No	No
Momentum	Yes	No	No
AdaGrad	No	Yes (sum)	No
RMSProp	No	Yes (EMA)	No
Adam	Yes	Yes (EMA)	Yes

Properties of Adam:

- Combines benefits of momentum and adaptive learning rates
- Bias correction ensures effective early learning
- Generally robust to hyperparameter choices
- Often works well with default parameters ($\epsilon = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$)
- Can sometimes generalize worse than SGD with momentum

3.3.10 Approximate Second-Order Methods

Relevant for Q.No. 11

Newton's Method: *Update Rule:*

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \boldsymbol{H}_t^{-1} \nabla_{\boldsymbol{\theta}} \mathrm{L}(\boldsymbol{\theta}_t)$$

where $\boldsymbol{H}_t$ is the Hessian matrix (matrix of second derivatives) at $\boldsymbol{\theta}_t$.

Intuition: First-order methods (like SGD) use a linear approximation of the loss. New-ton's method uses a quadratic approximation, which can capture curvature information and converge in fewer steps.

Why Approximate? (Q.No.

11) For a model with n parameters:

- **Storage:** Hessian is an $n \times n$ matrix requiring $O(n^2)$ memory.
- **Computation:** Computing the Hessian requires $O(n^2)$ operations.
- **Inversion:** Inverting the Hessian requires $O(n^3)$ operations.
- For modern DNNs with millions of parameters, this is impossible:
  - $n = 10^6$ parameters
  - Hessian has 10^{12} elements
  - At 4 bytes per element, this requires 4 TB of memory
  - Inversion is computationally infeasible

Approximations Used in Practice:

1. *Conjugate Gradients:*
   - Iteratively solves linear systems without explicitly storing the Hessian.
   - Requires only Hessian-vector products $\boldsymbol{Hv}$, which can be computed efficiently.
   - For quadratic problems, converges in at most n steps.
   - Used in training with full-batch (not mini-batch) settings.
2. *BFGS (Broyden-Fletcher-Goldfarb-Shanno):*
   - Quasi-Newton method that builds an approximation to the inverse Hessian.
   - Updates the approximation using only gradient information.
   - Requires $O(n^2)$ memory to store the approximation.
   - Still impractical for very large models.
3. *L-BFGS (Limited-Memory BFGS):*
   - Stores only a few vectors (m) representing the approximation.
   - Memory requirement: $O(mn)$ where m is typically 10-100.
   - Does not store the full Hessian approximation.
   - Popular for optimization with full-batch settings.
   - Can be effective for small to medium-sized models.
4. *Diagonal Approximation:*
   - Approximate Hessian as diagonal matrix (ignoring off-diagonal terms).
   - Requires only $O(n)$ storage.

- Used in some adaptive learning rate methods (like AdaGrad's connection to diagonal approximation).
- Loses information about interactions between parameters.

5. *Kronecker-Factored Approximate Curvature (K-FAC):*
   - Advanced approximation used in some distributed training systems.
   - Exploits structure in neural networks (Kronecker product structure).
   - More accurate than diagonal approximations.

Comparison of Methods (Q.No. 11):

Method	Memory	Computational Cost	Accuracy
Newton (exact)	$O(n^2)$	$O(n^3)$	Exact
BFGS	$O(n^2)$	$O(n^2)$	Good approximation
L-BFGS	$O(mn)$	$O(mn)$	Good approximation
Conjugate Gradients	$O(n)$	$O(n \cdot \text{iter})$	Exact for quadratics
Diagonal Approximation	$O(n)$	$O(n)$	Poor (ignores interactions)

—

3.3.11 Parameter Initialization Strategies

Relevant for: All Questions

(Foundational) Importance of Initialization:

- Affects speed of convergence
- Can prevent vanishing/exploding gradients
- Influences final model quality
- Different activation functions require different strategies

Xavier/Glorot Initialization: Designed for layers with sigmoid or tanh activations to maintain variance across layers.

Gaussian Version:

$$W_{ij} \sim \mathrm{N}\left(0, \frac{2}{n_{\mathrm{in}} + n_{\mathrm{out}}}\right)$$

Uniform Version:

$$W_i \sim U\left(-\frac{\sqrt{6}}{\sqrt{n_{\mathrm{in}} + n_{\mathrm{out}}}}, \frac{\sqrt{6}}{\sqrt{n_{\mathrm{in}} + n_{\mathrm{out}}}}\right)$$

where n_{in} is number of input units, n_{out} is number of output units.

He Initialization: Optimized for ReLU activations (and variants like Leaky ReLU). ReLU zeros out half the values, so variance needs to be larger.

Gaussian Version:

$$W_{ij} \sim \mathrm{N}\left(0, \frac{2}{n_{\mathrm{in}}}\right)$$

Uniform Version:

$$W_{ij} \sim U\left(-\frac{\sqrt{6}}{\sqrt{n_{\mathrm{in}}}}, \frac{\sqrt{6}}{\sqrt{n_{\mathrm{in}}}}\right)$$

Other Initialization Strategies:

- **Zero Initialization:** All weights zero. Symmetry breaking fails; all neurons learn same features.
- **Random Uniform:** $W \sim U(-a, a)$. Small a prevents exploding gradients initially.
- **Orthogonal Initialization:** Initialize weight matrices as orthogonal. Helps with gradient flow in RNNs.
- **Pre-training:** Initialize with weights from unsupervised pre-training (less common now).

Mathematical Justification: For a linear layer $y = Wx$ with x having variance $\text{Var}(x)$: $\text{Var}(y) = n_{\text{in}} \cdot \text{Var}(W) \cdot \text{Var}(x)$

To keep variance constant ($\text{Var}(y) = \text{Var}(x)$), we need:

$$\text{Var}(W) = \frac{1}{n_{\text{in}}}$$

Xavier initialization uses average of n_{in} and n_{out} for backpropagation symmetry.

—

3.3.12 Batch Normalization

Relevant for: All Questions (Optimization Strategy)

Definition: Batch normalization normalizes the inputs to each layer to have zero mean and unit variance across the mini-batch.

Forward Pass: For a mini-batch $B = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m\}$:

$$\mu_B = \frac{1}{m} \sum_{i=1}^{m} \boldsymbol{x}_i$$

$$\sigma_B^2 = \frac{1}{m} \sum_{i=} (\boldsymbol{x}_i - \mu_B)^2$$

$$\hat{\boldsymbol{x}}_i = \frac{\boldsymbol{x}_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$\boldsymbol{y}_i = \gamma \hat{\boldsymbol{x}}_i + \beta$$

where:

- γ is learnable scale parameter

- β is learnable shift parameter
- ϵ is small constant for numerical stability

Benefits:

- **Reduces Internal Covariate Shift:** Stabilizes distribution of layer inputs.
- **Higher Learning Rates:** Allows more aggressive optimization.
- **Regularization Effect:** Adds noise due to batch statistics, reducing need for dropout.
- **Less Sensitive to Initialization:** Reduces dependence on careful initialization.
- **Faster Convergence:** Typically speeds up training significantly.

Testing Phase: During testing, use running averages of μ and σ^2 computed during training:

$$\hat{x} = \frac{x - \mu_{\text{running}}}{\sqrt{\sigma_{\text{running}}^2 + \epsilon}}$$

—

3.3.13 Gradient Clipping

Relevant for: Q.No. 8 (Exploding Gradients)

Definition: Gradient clipping prevents exploding gradients by scaling down gradients that exceed a threshold.

Norm Clipping:

$$\boldsymbol{g} \leftarrow \begin{cases} \boldsymbol{g} & \text{if } \|\boldsymbol{g}\| \leq \text{threshold} \\ \dfrac{\text{threshold}}{\|\boldsymbol{g}\|}\boldsymbol{g} & \text{otherwise} \end{cases}$$

Value Clipping:

$$g_i \leftarrow \max(\min(g_i, \text{threshold}), -\text{threshold})$$

Why It Works:

- Preserves gradient direction while controlling magnitude.
- Essential for RNNs where gradients can explode due to repeated multiplication.
- Allows training very deep networks without instability.
- Can be combined with any optimization algorithm.

Typical Threshold: Common values range from 1.0 to 10.0, depending on the problem.

—

3.3.14 Learning Rate Scheduling

Relevant for: All Questions (Optimization Strategy)

Motivation: Adjusting learning rate during training can improve convergence and final performance.

Common Schedules:

1. *Step Decay:* Reduce learning rate by factor γ every few epochs:

$$\epsilon_t = \epsilon_0 \cdot \gamma^{\lfloor t/t_{\text{step}} \rfloor}$$

2. *Exponential Decay:*

$$\epsilon_t = \epsilon_0 \cdot e^{-kt}$$

3. *Cosine Annealing:*

$$\epsilon_t = \epsilon_{\min} + \frac{1}{2}(\epsilon - \epsilon_{\min})\left(1 + \cos\frac{t}{T}\pi\right)$$

4. *Warmup:* Start with small learning rate, gradually increase:

$$\epsilon_t = \epsilon_{\max} \cdot \min\left(1, \frac{t}{T_{\text{warmup}}}\right)$$

5. *Cyclical Learning Rates:* Vary learning rate cyclically between bounds:

$$\epsilon_t = \epsilon_{\min} + \frac{1}{2}(\epsilon_{\max} - \epsilon_{\min})\left(1 + \cos\frac{T_{\text{cur}}}{T}\pi\right)$$

Why Warmup Helps:

- Prevents early instability when parameters are random.
- Allows optimizer to find good direction before large steps.
- Particularly important with Adam and batch normalization.

—

3.3.15 Question Bank Mapping Summary

Question No.	Covered in Subsection
Q.No. 1 (a,b)	3.3.1 – *L1/L2Regularization*
Q.No. 2	3.3.1 – *EffectofL1/L2Penalties*
Q.No. 3	3.3.2 – *DropoutAnalysis*
Q.No. 4	3.3.3 – *DatasetAugmentation*
Q.No. 5	3.3.5 – *EnsembleMethods*(*Bagging*)
Q.No. 6	3.3.6 – *ParameterSharing*
Q.No. 7	3.3.4 – *EarlyStopping*
Q.No. 8	3.3 – *OptimizationChallenges*
Q.No. 9	3.3.9 – *AlgorithmComparison*
Q.No. 10	3.3.9 – *AdamStepExample*
Q.No. 11	3.3.10 – *Second – OrderMethods*

UNIT IV

UNIT-IV: CONVOLUTIONAL NETWORKS (9 Hrs)

Convolutional Networks: The Convolution Operation, Pooling, Convolution, Basic Convolution Functions, Structured Outputs, Data Types, Efficient Convolution Algorithms, Random or Unsupervised Features, Basis for Convolutional Networks.

Convolutional Neural Networks

4.1 Introduction to Convolutional Networks

Definition 4.1.1 (Convolutional Neural Network (CNN)). *A specialized neural network architecture that uses convolution operations in place of general matrix multiplication in at least one of its layers, designed to process data with grid-like topology (images, time series).*

4.1.1 The Convolution Operation

Definition 4.1.2 (Convolution Operation). *For 2D images with kernel K and input I:*

$$S(i,j) = (I * K)(i,j) = \sum_{m=-a}^{a} \sum_{n=-b}^{b} I(i-m, j-n) K(m,n)$$

where kernel K has size $(2a+1) \times (2b+1)$.

Properties of Convolution

1. **Sparse Interactions**: Each output unit interacts with only local region of input

   $$\text{Receptive field size} = \text{kernel size}$$

2. **Parameter Sharing**: Same kernel used across all spatial locations

   $$\text{Parameters per layer} = k_h \times k_w \times c_{\text{in}} \times c_{\text{out}}$$

   vs fully connected: $h \times w \times c_{\text{in}} \times c_{\text{out}}$

3. **Equivariance to Translation**:

   $$f(g(x)) = g(f(x)) \text{ where } g \text{ is translation}$$

Layer Type	Parameters (3×3, 64 filters)	Memory (MB)
Fully Connected (224×224 input)	$224 \times 224 \times 3 \times 64 \approx 9.6M$	38.4
Convolutional (3×3 kernel)	$3 \times 3 \times 3 \times 64 = 1,728$	0.0069

Table 4.1: Parameter comparison: Convolutional vs Fully Connected layers

4.1.2 Pooling

Definition 4.1.3 (Pooling Operation). *A downsampling operation that replaces a local region with a summary statistic:*

$$y_{i,j,d} = pool(\{x_{s\cdot i+m,s\cdot j+n,d}\}_{m=1,\ldots,k,n=1,\ldots,k})$$

where s is stride and k is pooling window size.

Types of Pooling

1. **Max Pooling**:

$$y_{i,j,d} = \max_{m,n \in \text{window}} x_{i+m,j+n,d}$$

2. **Average Pooling**:

$$y_{i,j,d} = \frac{1}{k^2} \sum_{m=1}^{k} \sum_{n=1}^{k} x_{i+m,j+n,d}$$

3. **Global Average Pooling**:

$$y_d = \frac{1}{H \times W} \sum_{i=}^{H} \sum_{j=1}^{W} x_{i,j,d}$$

Properties of Pooling

1. **Invariance to Small Translations**:

$$\text{pool}(f(x)) \approx \text{pool}(x) \quad \text{for small translations } f$$

2. **Dimensionality Reduction**:

$$\text{Output size} = \frac{\text{Input size} - \text{Kernel size}}{\text{Stride}} + 1$$

3. **Increased Receptive Field**: Each pooling layer doubles the receptive field

Parameter	Symbol	Typical Values
Kernel size	$k_h \times k_w$	3×3, 5×5, 7×7
Stride	s	1, 2
Padding	p	0, 'same'
Number of filters	c_{out}	32, 64, 128, 256
Dilation	d	1 (standard), 2, 4

Table 4.2: Convolution layer hyperparameters

4.1.3 Basic Convolution Functions

Convolution Layer Parameters Output Size Calculation

For input size W, kernel size K, stride S, padding P, dilation D:

$$W_{\text{out}} = \left\lfloor \frac{W + 2P - D(K-1) - 1}{S} + 1 \right\rfloor$$

Configuration	Input Size	Output Size	Formula
Stride 1, No padding	224	222	$(224 - 3 + 1)$
Stride 2, No padding	224	111	$\lfloor (224 - 3)/2 + 1 \rfloor$
Same padding	224	224	$\lfloor (224 + 2 \times 1 - 3)/1 + 1 \rfloor$

Table 4.3: Output size examples with 3×3 kernel

Activation Functions in CNNs

Activation	Formula	Properties
ReLU	$f(x) = \max(0, x)$	Non-saturating, sparse
Leaky ReLU	$f(x) = \max(\alpha x, x)$, $\alpha = 0.01$	Avoids dead neurons
ELU	$f(x) = \begin{cases} x & x > 0 \\ \alpha(e^x - 1) & x \le 0 \end{cases}$	Smooth, zero-mean
SELU	$f(x) = \lambda \begin{cases} x & x > 0 \\ \alpha(e^x - 1) & x \le 0 \end{cases}$	Self-normalizing

Table 4.4: Common activation functions in CNNs

4.1.4 Structured Outputs

Output Types

1. **Image Classification**: Single label per image

$$\hat{y} = \text{softmax}(\boldsymbol{W}\boldsymbol{h} + \boldsymbol{b})$$

2. **Object Detection**: Bounding boxes + class labels

$$L = L_{\text{class}} + \lambda L_{\text{bbox}}$$

3. **Semantic Segmentation**: Pixel-wise classification

$$\hat{y}_{i,j} = \text{softmax}(f(\boldsymbol{x})_{i,j})$$

4. **Instance Segmentation**: Detect and segment each object
5. **Keypoint Detection**: Localize specific points

4.1.5 Data Types

Input Data Dimensions

Data Type	Dimensions	Example Shape	Applications
Grayscale Image	$(H, W, 1)$	$224 \times 224 \times 1$	MNIST, medical imaging
RGB Image	$(H, W, 3)$	$224 \times 224 \times 3$	ImageNet, natural images
Video	$(T, H, W, 3)$	$32 \times 224 \times 224 \times 3$	Action recognition
3D Volume	$(D, H, W, 1)$	$128 \times 128 \times 128 \times 1$	CT scans, MRI
Multispectral	(H, W, C)	$256 \times 256 \times 16$	Satellite imagery

Table 4.5: Common input data types in CNNs

Tensor Shapes Through Network

4.1.6 Efficient Convolution Algorithms

im2col (Image to Column)

Convert convolution to matrix multiplication:

$$\text{im2col}(I) \cdot \text{reshape}(K) = \text{conv}(I, K)$$

Memory complexity: $O(k^2 \cdot H_{\text{out}} \cdot W_{\text{out}} \cdot C_{\text{in}})$

Layer	Operation	Input Shape	Output Shape
Input	-	-	224× 224× 3
Conv1	64 filters, 3×3, stride 1	× 224× 224	3 × 224× 224 64
Pool1	Max 2×2, stride 2	224× 224× 64	112× 112× 64
Conv2	128 filters, 3×3	112× 112× 64	112× 112× 128
Pool2	Max 2×2	112× 112× 128	56× 56× 128
Conv3	256 filters, 3×3	56× 56× 128	56× 56× 256
Pool3	Max 2×2	56× 56× 256	28× 28× 256
FC	Dense 4096	28× 28× 256	4096
Output	Dense num classes	4096	1000

Table 4.6: Tensor shape evolution through typical CNN

Winograd Convolution

For kernel size r and output size m, reduces multiplication count:

$$\text{Multiplications} = \left\lfloor \frac{m}{v} \right\rfloor \cdot \left\lfloor \frac{n}{v} \right\rfloor \cdot (v + r - 1)^2$$

FFT-based Convolution

Using Convolution Theorem:

$$\mathrm{F}(I * K) = \mathrm{F}(I) \odot \mathrm{F}(K)$$

Complexity: $O(n \log n)$ vs $O(n^2)$ for direct convolution.

Algorithm	3×3 Kernel	5×5 Kernel	7×7 Kernel
Direct	1.0×	1.0×	1.0×
im2col	1.2×	1.5×	2.0×
FFT	0.8×	0.6×	0.4×
Winograd	0.6×	0.5×	0.4×

Table 4.7: Relative speedup of convolution algorithms (higher is faster)

4.1.7 Random or Unsupervised Features

Random Features

Initialize convolution kernels randomly and keep fixed:

$$y = \sigma(\boldsymbol{W}_{\text{random}} * x + b)$$

Properties:

- Universal approximation capability
- No training required
- Often used in extreme learning machines

Unsupervised Pre-training Methods

1. **Autoencoders**: Reconstruct input

$$\mathrm{L} = \|x - \mathrm{decoder}(\mathrm{encoder}(x))\|^2$$

2. **Restricted Boltzmann Machines**: Energy-based model

$$E(v, h) = -v^T Wh - b^T v - c^T h$$

$$P(v, h) = \frac{e^{-E(v,h)}}{Z}$$

3. **Contrastive Learning**: SimCLR, MoCo

$$\mathrm{L} = -\log \sum_{k=i} \frac{\exp(\mathrm{sim}(z_i, z_j)/\tau)}{\exp(\mathrm{sim}(z_i, z_k)/\tau)}$$

4.1.8 Basis for Convolutional Networks

Theoretical Foundations

Theorem 4.1.1 (Shift-Invariance)**.** *A CNN with pooling and shared weights is equivariant to translations:*

$$CNN(translate(x)) \approx translate(CNN(x))$$

Theorem 4.1.2 (Universal Approximation)**.** *A CNN with sufficient width and depth can approximate any continuous function on a compact subset of* R^n *to arbitrary accuracy.*

Key Architectural Principles

1. **Locality**: Local receptive fields capture spatial correlations
2. **Hierarchy**: Increasing abstraction with depth
3. **Compositionality**: Complex features from simple ones
4. **Weight Sharing**: Translation equivariance and parameter efficiency
5. **Pooling**: Downsampling and local invariance

Architecture	Year	Layers	Key Innovation	Top-5 Error
LeNet-5	1998	7	First modern CNN	-
AlexNet	2012	8	ReLU, Dropout, GPU	15.3%
VGG-16	2014	16	Small 3×3 filters	7.3%
GoogLeNet	2014	22	Inception modules	6.7%
ResNet-50	2015	50	Skip connections	3.6%
DenseNet-121	2017	121	Dense connections	3.5%
EfficientNet-B7	2019	-	NAS optimized	2.8%

Table 4.8: Evolution of CNN architectures on ImageNet

Classic CNN Architectures

Case Study: ResNet (Residual Networks)

Key Idea: Skip connections for deeper networks

$$y = \mathrm{F}(x, \{W_i\}) + x$$

Forward/Backward Properties:

$$\frac{\partial \mathrm{L}}{\partial x_l} \equiv \frac{\partial \mathrm{L}}{\partial x_L} \left(1 + \frac{\partial}{\partial x_l} \sum_{i=1}^{L-1} \mathrm{F}(x_i, W_i) \right)$$

Architecture Variants:

- ResNet-18: 11.7M parameters, 1.8×10^9 FLOPs
- ResNet-34: 21.8M parameters, 3.6×10^9 FLOPs
- ResNet-50: 25.6M parameters, 3.8×10^9 FLOPs
- ResNet-101: 44.5M parameters, 7.6×10^9 FLOPs
- ResNet-152: 60.2M parameters, 11.3×10^9 FLOPs

Residual Block Implementation

Algorithm 3: Residual Block Forward Pass

Input: Input x, weights $\{W_1, W_2\}$, optional projection W_s
Output: Output y
begin
  $h \leftarrow \text{Conv}(x, W_1)$;
  $h \leftarrow \text{BatchNorm}(h)$;
  $h \leftarrow \text{ReLU}(h)$;
  $h \leftarrow \text{Conv}(h, W_2)$;
  $h \leftarrow \text{BatchNorm}(h)$;
  if *input and output dimensions differ*
    then $x \leftarrow \text{Conv}(x, W_s)$;
  $y \leftarrow \text{ReLU}(h + x)$;
  return y

Modern CNN Innovations

1. **Squeeze-and-Excitation Networks**: Channel attention

$$s = \sigma(W_2 \cdot \text{ReLU}(W_1 \cdot \text{GAP}(x)))$$

$$\tilde{x} = s \cdot x$$

2. **Depthwise Separable Convolutions**:

$$\text{Conv}_{DW}(x) + \text{Conv}_{PW}$$

(x) Reduces parameters by factor of k^2

3. **Dilated Convolutions**: Increase receptive field without parameters

$$y[i] = \sum_k x[i + d \cdot k] w[k]$$

4. **Deformable Convolutions**: Learnable offsets

$$y[p] = \sum_k w[k] \cdot x[p + k + \Delta p_k]$$

4.1.9 Practical Considerations

Training Tips for CNNs

- Use batch normalization for faster convergence
- Start with small learning rate (10^{-3} to 10^{-4})
- Apply data augmentation: random crops, flips, color jitter
- Use transfer learning from pre-trained models
- Monitor validation accuracy for overfitting

Common CNN Design Patterns

Pattern	Structure	Purpose
VGG-style	Conv3×3, Conv3×3, Pool	Feature extraction
Inception	Multiple parallel convolutions	Multi-scale features
Residual	$F(x) + x$	Deep network training
Dense	$[x_1, x_2, \ldots, x_n]$	Feature reuse
Bottleneck	$1\times1 \to 3\times3 \to 1\times1$	Parameter efficiency

Table 4.9: Common CNN architectural patterns

Case Study: Image Classification Pipeline

Algorithm 4: Training a CNN for Image Classification

Input: Training images $D = \{(x_i, y_i)\}_{i=1}^{N}$, validation set V
Output: Trained CNN model f_θ

begin
Initialize network with pre-trained weights (ImageNet); Replace final layer for new number of classes;
Set optimizer: Adam with $\eta = 10^{-4}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$;
Set data augmentation: random crop, horizontal flip, color jitter;
for *epoch* = 1 *to* E **do**
for *batch* B *in* D **do**
$x_{\text{aug}} \leftarrow \text{augment}(B_x)$;
$\hat{y} \leftarrow f_\theta(x_{\text{aug}})$;
$L \leftarrow \text{cross entropy}(\hat{y}, B_y)$;
$\nabla_\theta L \leftarrow \text{backward}(L)$;
$\theta \leftarrow \text{Adam}(\theta, \nabla_\theta L)$; Evaluate on V;
if *validation accuracy improved* **then**
Save checkpoint;
if *validation loss increases for 5 epochs* **then** Reduce learning rate by factor 0.1;
return *best model*

UNIT V

UNIT-V: SEQUENCE MODELING (9 Hrs)

Sequence Modeling: Recurrent and Recursive Nets: Unfolding Computational Graphs, Recurrent Neural Networks, Bidirectional RNNs, Encoder-Decoder Sequence-to-Sequence Architectures, Deep Recurrent Networks, Recursive Neural Networks, Echo State Networks, LSTM, Gated RNNs, Optimization for Long-Term Dependencies, Auto encoders, Deep Generative Models.

Sequence Modeling

5.1 Introduction to Sequence Modeling

Definition 5.1.1 (Sequence Modeling)**.** *Modeling data where input and/or output are se-quences of variable length, capturing temporal dependencies and order information.*

5.1.1 Unfolding Computational Graphs

Definition 5.1.2 (Computational Graph Unfolding)**.** *Converting a recurrent system into a deep computational graph by expanding through time steps.*

For a classical RNN formulation:

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta})$$

Unfolded for T time steps:

$$\boldsymbol{h}^{(t)} = f(f(...f(\boldsymbol{h}^{(0)}, \boldsymbol{x}^{(1)}), ..., \boldsymbol{x}^{(t-1)}), \boldsymbol{x}^{(t)})$$

Time Step	Hidden State	Output
$t = 1$	$\boldsymbol{h}^{(1)} = f(\boldsymbol{h}^{(0)}, \boldsymbol{x}^{(1)})$	$\boldsymbol{o}^{(1)} = g(\boldsymbol{h}^{(1)})$
$t = 2$	$\boldsymbol{h}^{(2)} = f(\boldsymbol{h}^{(1)}, \boldsymbol{x}^{(2)})$	$\boldsymbol{o}^{(2)} = g(\boldsymbol{h}^{(2)})$
$t = 3$	$\boldsymbol{h}^{(3)} = f(\boldsymbol{h}^{(2)}, \boldsymbol{x}^{(3)})$	$\boldsymbol{o}^{(3)} = g(\boldsymbol{h}^{(3)})$
.	.	.
$t = T$	$\boldsymbol{h}^{(T)} = f(\boldsymbol{h}^{(T-1)}, \boldsymbol{x}^{(T)})$	$\boldsymbol{o}^{(T)} = g(\boldsymbol{h}^{(T)})$

Table 5.1: Unfolded RNN computations over time

5.1.2 Recurrent Neural Networks

Definition 5.1.3 (Recurrent Neural Network)**.** *A neural network with cyclic connections that maintain a hidden state capturing information from previous time steps.*

Basic RNN Formulation

Algorithm 5: Forward Pass of Simple RNN

Input: Input sequence $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(T)}$, initial hidden state $\boldsymbol{h}^{(0)}$
Output: Output sequence $\boldsymbol{o}^{(1)}, \boldsymbol{o}^{(2)}, \ldots, \boldsymbol{o}^{(T)}$

begin
  for $t = 1$ *to* T **do**
    $\boldsymbol{h}^{(t)} \leftarrow \tanh(\boldsymbol{W}_{hh}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{xh}\boldsymbol{x}^{(t)} + \boldsymbol{b}_h)$; $\boldsymbol{o}^{(t)} \leftarrow \boldsymbol{W}_{hy}\boldsymbol{h}^{(t)} + \boldsymbol{b}_y$;
  return $\{\boldsymbol{o}^{(1)}, \boldsymbol{o}^{(2)}, \ldots, \boldsymbol{o}^{(T)}\}$

Parameters:

- $\boldsymbol{W}_{xh}$: Input-to-hidden weight matrix
- $\boldsymbol{W}_{hh}$: Hidden-to-hidden weight matrix
- $\boldsymbol{W}_{hy}$: Hidden-to-output weight matrix
- $\boldsymbol{b}_h$, $\boldsymbol{b}_y$: Bias vectors

Backpropagation Through Time (BPTT)

Algorithm 6: Backpropagation Through Time

Input: Unfolded RNN with T time steps, loss gradients $\frac{\partial L}{\partial \boldsymbol{o}^{(t)}}$
Output: Gradients w.r.t. all parameters **begin**
  Initialize $\frac{\partial L}{\partial \boldsymbol{h}^{(T)}} = \boldsymbol{0}$; **for** $t = T$ *down to* 1 **do**
    $\frac{\partial L}{\partial \boldsymbol{h}^{(t)}} \leftarrow \frac{\partial L}{\partial \boldsymbol{o}^{(t)}} \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}} + \frac{\partial L}{\partial \boldsymbol{h}^{(t+1)}} \frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}}$;
    Accumulate gradients for $\boldsymbol{W}_{xh}$, $\boldsymbol{W}_{hh}$, $\boldsymbol{b}_h$, $\boldsymbol{W}_{hy}$, $\boldsymbol{b}_y$;
  return *Gradients*

5.1.3 Bidirectional RNNs

Definition 5.1.4 (Bidirectional RNN)**.** *An RNN architecture that processes sequences in both forward and backward directions, capturing context from past and future.*

Forward pass:

$$\overrightarrow{\boldsymbol{h}}^{(t)} = f(\overrightarrow{\boldsymbol{W}}_{xh}\boldsymbol{x}^{(t)} + \overrightarrow{\boldsymbol{W}}_{hh}\overrightarrow{\boldsymbol{h}}^{(t-1)} + \overrightarrow{\boldsymbol{b}}_h)$$
$$\overleftarrow{\boldsymbol{h}}^{(t)} = f(\overleftarrow{\boldsymbol{W}}_{xh}\boldsymbol{x}^{(t)} + \overleftarrow{\boldsymbol{W}}_{hh}\overleftarrow{\boldsymbol{h}}^{(t+1)} + \overleftarrow{\boldsymbol{b}}_h)$$

$$\boldsymbol{h}^{(t)} = [\overrightarrow{\boldsymbol{h}}^{(t)}; \overleftarrow{\boldsymbol{h}}^{(t)}]$$
$$\boldsymbol{o}^{(t)} = \boldsymbol{W}_{hy}\boldsymbol{h}^{(t)} + \boldsymbol{b}_y$$

Time Step	Forward State	Backward State
$t = 1$	$\overrightarrow{\boldsymbol{h}}^{(1)} = f(\boldsymbol{x}^{(1)}, \overrightarrow{\boldsymbol{h}}^{(0)})$	$\overleftarrow{\boldsymbol{h}}^{(1)} = f(\boldsymbol{x}^{(1)}, \overleftarrow{\boldsymbol{h}}^{(2)})$
$t = 2$	$\overrightarrow{\boldsymbol{h}}^{(2)} = f(\boldsymbol{x}^{(2)}, \overrightarrow{\boldsymbol{h}}^{(1)})$	$\overleftarrow{\boldsymbol{h}}^{(2)} = f(\boldsymbol{x}^{(2)}, \overleftarrow{\boldsymbol{h}}^{(3)})$
.	.	.
$t = T$	$\overrightarrow{\boldsymbol{h}}^{(T)} = f(\boldsymbol{x}^{(T)}, \overrightarrow{\boldsymbol{h}}^{(T-1)})$	$\overleftarrow{\boldsymbol{h}}^{(T)} = f(\boldsymbol{x}^{(T)}, \overleftarrow{\boldsymbol{h}}^{(T+1)})$

Table 5.2: Bidirectional RNN states computation

5.1.4 Encoder-Decoder Sequence-to-Sequence Architectures

Definition 5.1.5 (Encoder-Decoder Architecture)**.** *A framework where an encoder RNN processes input sequence into a context vector, and a decoder RNN generates output sequence from this context.*

Encoder

Algorithm 7: Encoder RNN

Input: Input sequence $\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(T_x)}\}$
Output: Context vector $\boldsymbol{c}$
begin
  Initialize $\boldsymbol{h}^{(0)} = \boldsymbol{0}$; **for** $t = 1$ *to* T_x **do**
    $\boldsymbol{h}^{(t)} \leftarrow f_{\text{enc}}(\boldsymbol{x}^{(t)}, \boldsymbol{h}^{(t-1)})$;
  $\boldsymbol{c} \leftarrow \boldsymbol{h}^{(T_x)}$;
  return $\boldsymbol{c}$

Decoder

Algorithm 8: Decoder RNN

Input: Context vector $\boldsymbol{c}$, target sequence length T_y **Output:** Output sequence $\{\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)}, \ldots, \boldsymbol{y}^{(T_y)}\}$
begin
  Initialize $\boldsymbol{s}^{(0)} = \boldsymbol{c}$;
  for $t = 1$ *to* T_y **do**
    $\boldsymbol{s}^{(t)} \leftarrow f_{\text{dec}}(\boldsymbol{y}^{(t-1)}, \boldsymbol{s}^{(t-1)}, \boldsymbol{c})$;
    $\boldsymbol{o}^{(t)} \leftarrow g(\boldsymbol{s}^{(t)})$;
  return $\{\boldsymbol{o}^{(1)}, \boldsymbol{o}^{(2)}, \ldots, \boldsymbol{o}^{(T_y)}\}$

Attention Mechanism Attention scores:

$$e^{(t,i)} = \boldsymbol{v}_a^\top \tanh(\boldsymbol{W}_a \boldsymbol{s}^{(t-1)} + \boldsymbol{U}_a \boldsymbol{h}^{(i)})$$

Attention weights:

$$\alpha^{(t,i)} = \frac{\exp(e^{(t,i)})}{\sum_{j=1}^{T_x} \exp(e^{(t,j)})}$$

Context vector:

$$\boldsymbol{c}^{(t)} = \sum_{i=1} \alpha^{(t,i)} \boldsymbol{h}^{(i)}$$

Decoder update with attention:

$$\boldsymbol{s}^{(t)} = f_{\text{dec}}(\boldsymbol{y}^{(t-1)}, \boldsymbol{s}^{(t-1)}, \boldsymbol{c}^{(t)})$$

Attention Type	Score Computation	Properties
Dot Product	$e^{(t,i)} = \boldsymbol{s}^{(t-1)T} \boldsymbol{h}^{(i)}$	Simple, efficient
Additive	$e^{(t,i)} = \boldsymbol{v}^T \tanh(\boldsymbol{W} [\boldsymbol{s}^{(t-1)}; \boldsymbol{h}^{(i)}])$	General, flexible
Multiplicative	$e^{(t,i)} = \boldsymbol{s}^{(t-1)T} \boldsymbol{W} \boldsymbol{h}^{(i)}$	Bilinear scoring
Scaled Dot-Product	$e^{(t,i)} = \frac{\boldsymbol{s}^{(t-1)T} \boldsymbol{h}^{(i)}}{\sqrt{d}}$	Used in Transformers

Table 5.3: Types of attention mechanisms

5.1.5 Deep Recurrent Networks

Definition 5.1.6 (Deep RNN). *RNN with multiple hidden layers stacked vertically, enabling hierarchical feature learning.*

Deep RNN forward pass:

$$\boldsymbol{h}_l^{(t)} = f(\boldsymbol{W}_{hh}^{(l)} \boldsymbol{h}_l^{(t-1)} + \boldsymbol{W}_{xh}^{(l)} \boldsymbol{h}_{l-1}^{(t)} + \boldsymbol{b}^{(l)})$$

where $\boldsymbol{h}_0^{(t)} = \boldsymbol{x}^{(t)}$.

Architecture variants:

1. **Stacked RNNs**: Multiple RNN layers in sequence
2. **Skip Connections**: Residual connections between layers
3. **Highway Networks**: Gated skip connections

5.1.6 Recursive Neural Networks

Definition 5.1.7 (Recursive Neural Network)**.** *A network that applies the same weights recursively over a tree structure, capturing hierarchical compositionality.*

Depth	Parameters	Perplexity	Training Time
1 layer	1M	78.3	1×
2 layers	2M	72.1	1.8×
3 layers	3M	69.5	2.5×
4 layers	4M	68.2	3.2×

Table 5.4: Effect of depth on language modeling performance

Parent composition:

$$p = f\left(W \begin{bmatrix} c_1 \\ c \end{bmatrix} + b\right)$$

For n-ary tree:

$$p = f\left(\sum_{i=1}^{n} W_i c_i + b\right)$$

Applications:

- Natural language parsing
- Scene graph generation
- Molecular structure prediction
- Program analysis

5.1.7 Echo State Networks

Definition 5.1.8 (Echo State Network)**.** *A reservoir computing approach where recurrent connections are randomly initialized and fixed, only output weights are trained.*

Reservoir update:

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{W}_{in}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{res}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{fb}\boldsymbol{y}^{(t-1)})$$

Output:

$$\boldsymbol{y}^{(t)} = \boldsymbol{W}_{out}[\boldsymbol{x}^{(t)}; \boldsymbol{h}^{(t)}]$$

Training:

$$\boldsymbol{W}_{out} = \arg\min_{\boldsymbol{w}} \|\boldsymbol{W}\boldsymbol{H} - \boldsymbol{Y}\|^2 + \lambda\|\boldsymbol{W}\|_2$$

Closed form: $\boldsymbol{W}_{out} = \boldsymbol{Y}\boldsymbol{H}^T(\boldsymbol{H}\boldsymbol{H}^T + \lambda \boldsymbol{I})^{-1}$

5.1.8 Long Short-Term Memory (LSTM)

Definition 5.1.9 (LSTM)**.** *A gated RNN architecture designed to capture long-term depen-dencies through explicit memory cells and gating mechanisms.*

LSTM Components

1. **Forget Gate**: Controls what to discard from memory

$$\boldsymbol{f}^{(t)} = \sigma(\boldsymbol{W}_f[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}] + \boldsymbol{b}_f)$$

2. **Input Gate**: Controls what new information to store

$$\boldsymbol{i}^{(t)} = \sigma(\boldsymbol{W}_i[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}] + \boldsymbol{b}_i)$$

3. **Candidate Cell**: New candidate values

$$\tilde{\boldsymbol{c}}^{(t)} = \tanh(\boldsymbol{W}_c[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}] + \boldsymbol{b}_c)$$

4. **Cell Update**: Combine forget and input gates

$$\boldsymbol{c}^{(t)} = \boldsymbol{f}^{(t)} \odot \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \odot \tilde{\boldsymbol{c}}^{(t)}$$

5. **Output Gate**: Controls what to output from cell

$$\boldsymbol{o}^{(t)} = \sigma(\boldsymbol{W}_o[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}] + \boldsymbol{b}_o)$$

6. **Hidden State**: Final output

$$\boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \odot \tanh(\boldsymbol{c}^{(t)})$$

Algorithm 9: LSTM Forward Pass

Input: Input $\boldsymbol{x}^{(t)}$, previous hidden $\boldsymbol{h}^{(t-1)}$, previous cell $\boldsymbol{c}^{(t-1)}$
Output: Current hidden $\boldsymbol{h}^{(t)}$, current cell $\boldsymbol{c}^{(t)}$

begin

$\boldsymbol{f}^{(t)} \leftarrow \sigma(\boldsymbol{W}_f[\boldsymbol{h}^{(t-1)}; \boldsymbol{x}^{(t)}] + \boldsymbol{b}_f)$;
$\boldsymbol{i}^{(t)} \leftarrow \sigma(\boldsymbol{W}_i[\boldsymbol{h}^{(t-1)}; \boldsymbol{x}^{(t)}] + \boldsymbol{b}_i)$;
$\tilde{\boldsymbol{c}}^{(t)} \leftarrow \tanh(\boldsymbol{W}_c[\boldsymbol{h}^{(t-1)}; \boldsymbol{x}^{(t)}] + \boldsymbol{b}_c)$;
$\boldsymbol{c}^{(t)} \leftarrow \boldsymbol{f}^{(t)} \odot \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \odot \tilde{\boldsymbol{c}}^{(t)}$;
$\boldsymbol{o}^{(t)} \leftarrow \sigma(\boldsymbol{W}_o[\boldsymbol{h}^{(t-1)}; \boldsymbol{x}^{(t)}] + \boldsymbol{b}_o)$;
$\boldsymbol{h}^{(t)} \leftarrow \boldsymbol{o}^{(t)} \odot \tanh(\boldsymbol{c}^{(t)})$;
return $\boldsymbol{h}^{(t)}$, $\boldsymbol{c}^{(t)}$

5.1.9 Gated RNNs

Gated Recurrent Unit (GRU)

Definition 5.1.10 (GRU)**.** *Simplified gating mechanism with update and reset gates, merg-ing cell and hidden states.*

Update Gate:

$$\boldsymbol{z}^{(t)} = \sigma(\boldsymbol{W}_z[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}] + \boldsymbol{b}_z)$$

Reset Gate:

$$\boldsymbol{r}^{(t)} = \sigma(\boldsymbol{W}_r[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}] + \boldsymbol{b}_r)$$

Candidate

$$\tilde{\boldsymbol{h}}^{(t)} = \tanh(\boldsymbol{W}_h[\boldsymbol{r}^{(t)} \odot \boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}] + \boldsymbol{b}_h)$$

Hidden:

$$\boldsymbol{h}^{(t)} = (1 - \boldsymbol{z}^{(t)}) \odot \boldsymbol{h}^{(t-1)} + \boldsymbol{z}^{(t)} \odot \tilde{\boldsymbol{h}}^{(t)}$$

Final Hidden:

Gated RNN	Gates	Parameters	Perplexity
Simple RNN	0	1×	105.2
LSTM	3 (forget, input, output)	4×	78.4
GRU	2 (update, reset)	3×	82.1
Peephole LSTM	3 + peephole	4.5×	76.8

Table 5.5: Comparison of gated RNN architectures

5.1.10 Optimization for Long-Term Dependencies

Vanishing/Exploding Gradients Problem Gradient through time:

$$\frac{\partial \mathrm{L}}{\partial \boldsymbol{h}^{(t)}} = \frac{\partial \mathrm{L}}{\partial \boldsymbol{h}^{(T)}} \prod_{k=t}^{T-1} \frac{\partial \boldsymbol{h}^{(k+1)}}{\partial \boldsymbol{h}^{(k)}}$$

For simple RNN with tanh:

$$\left\| \frac{\partial \boldsymbol{h}^{(k+1)}}{\partial \boldsymbol{h}^{(k)}} \right\| \leq \|\boldsymbol{W}_{hh}\| \cdot \|\mathrm{diag}(f'(\boldsymbol{h}^{(k)}))\|$$

Solution Strategies

1. **Gradient Clipping**:

$$\boldsymbol{g} \leftarrow \begin{cases} \boldsymbol{g} & \text{if } \|\boldsymbol{g}\| \leq \text{threshold} \\ \frac{\text{threshold}}{\|\boldsymbol{g}\|}\boldsymbol{g} & \text{otherwise} \end{cases}$$

2. **Truncated BPTT**: Limit backpropagation to K steps
3. **Identity Initialization**: Initialize recurrent weights to identity
4. **Skip Connections**: Add direct connections across time
5. **Leaky Integration**: $h^{(t)} = \alpha h^{(t-1)} + (1 - \alpha)\tanh(\ldots)$

Method	Description	Max Dependency
Simple RNN	Basic recurrence	10 steps
Gradient Clipping	Clip gradient norm	20 steps
Truncated BPTT	Limited history	50 steps
LSTM/GRU	Gated memory	200 steps
Attention	Direct access	Unlimited
Transformer	Self-attention	Unlimited

Table 5.6: Methods for handling long-term dependencies

5.1.11 Autoencoders

Definition 5.1.11 (Autoencoder)**.** *An unsupervised neural network that learns to reconstruct its input through a bottleneck layer, capturing meaningful representations.*

Basic Autoencoder

Encoder:

$$\boldsymbol{h} = f(\boldsymbol{W}_e\boldsymbol{x} + \boldsymbol{b}_e)$$

Decoder:

$$\hat{\boldsymbol{x}} = g(\boldsymbol{W}_d\boldsymbol{h} + \boldsymbol{b}_d)$$

Loss:

$$\mathrm{L}(\boldsymbol{x}, \hat{\boldsymbol{x}}) = \|\boldsymbol{x} - \hat{\boldsymbol{x}}\|^2$$

Denoising Autoencoder

Corrupted input:

$$\tilde{\boldsymbol{x}} = \boldsymbol{x} + \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathrm{N}(0, \sigma^2)$$

Objective:

$$\mathrm{L} = \|\boldsymbol{x} - \mathrm{decoder}(\mathrm{encoder}(\tilde{\boldsymbol{x}}))\|^2$$

Sparse Autoencoder

Loss with sparsity penalty:

$$\mathrm{L} = \|\boldsymbol{x} - \hat{\boldsymbol{x}}\|^2 + \lambda \sum_j \mathrm{KL}(\rho \| \hat{\rho}_j)$$

where $\hat{\rho}_j = \frac{1}{N}\sum_{i=} h_j^{(i)}$ is average activation.

Variational Autoencoder (VAE)

Encoder outputs distribution parameters:

$$\boldsymbol{\mu} = f_\mu(\boldsymbol{x}), \quad \boldsymbol{\sigma} = f_\sigma(\boldsymbol{x})$$

Reparameterization trick:

$$\boldsymbol{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathrm{N}(0, \boldsymbol{I})$$

VAE Loss (ELBO):

$$\mathrm{L} = \underbrace{\mathrm{E}_{q(\boldsymbol{z}|\boldsymbol{x})}[\log p(\boldsymbol{x}|\boldsymbol{z})]}_{\text{Reconstruction}} - \underbrace{\mathrm{KL}(q(\boldsymbol{z}|\boldsymbol{x}) \| p(\boldsymbol{z}))}_{\text{Regularization}}$$

Algorithm 10: Training a Variational Autoencoder

Input: Dataset $\mathrm{D} = \{\boldsymbol{x}_i\}_{i=}^{N}$, latent dimension d_z
Output: Trained VAE model

begin
  for *epoch* = 1 *to* E **do**
    for *batch* B *in* D **do**
      $\boldsymbol{\mu}, \log \boldsymbol{\sigma} \leftarrow$ encoder(B);
      $\boldsymbol{\sigma} \leftarrow \exp(\log \boldsymbol{\sigma})$;
      $\boldsymbol{\epsilon} \sim \mathrm{N}(0, \boldsymbol{I})$;
      $\boldsymbol{z} \leftarrow \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$;
      $\hat{\boldsymbol{x}} \leftarrow$ decoder($\boldsymbol{z}$);
      $\mathrm{L}_{\text{recon}} \leftarrow \frac{1}{\mathrm{B}} \|\mathrm{B} - \hat{\boldsymbol{x}}\|^2$;
      $\mathrm{L}_{\mathrm{KL}} \leftarrow -\frac{1}{2}(1 + \log \boldsymbol{\sigma}^2 - \boldsymbol{\mu}^2 - \boldsymbol{\sigma}^2)$;
      $\mathrm{L} \leftarrow \mathrm{L}_{\text{recon}} + \mathrm{L}_{\mathrm{KL}}$;
      Update parameters via gradient descent;
  return *Trained encoder and decoder*

5.1.12 Deep Generative Models

Generative Adversarial Networks (GANs)

Generator G: Maps noise $\boldsymbol{z}$ to data space

$$\boldsymbol{x}_{\text{fake}} = G(\boldsymbol{z}), \quad \boldsymbol{z} \sim p(\boldsymbol{z})$$

Discriminator D: Distinguishes real from fake

$$D(\boldsymbol{x}) \in [0, 1] \text{ probability that } \boldsymbol{x} \text{ is real}$$

Minimax objective:

$$\min_G \max_D V(D, G) = \mathrm{E}_{\boldsymbol{x} \sim p_{\text{data}}}[\log D(\boldsymbol{x})] + \mathrm{E}_{\boldsymbol{z} \sim p_z}[\log(1 - D(G(\boldsymbol{z})))]$$

DCGAN (Deep Convolutional GAN)

Generator architecture:

- Input: latent vector $\boldsymbol{z}$ (100-dim)
- Fully connected → reshape
- Transposed convolutions with stride 2
- Batch normalization after each layer
- ReLU in generator, LeakyReLU in discriminator
- Output: 64 × 64 × 3 image with tanh

VAE-GAN Hybrid

Combines VAE and GAN:

$$L = L_{VAE} + L_{GAN}$$

Wasserstein GAN (WGAN) Critic loss:

$$L_D = E_{\boldsymbol{x}\sim p_{\text{data}}}[D(\boldsymbol{x})] - E_{\boldsymbol{z}\sim p_z}[D(G(\boldsymbol{z}))]$$

Generator loss:

$$L_G = -E_{\boldsymbol{z}\sim p_z}[D(G(\boldsymbol{z}))]$$

With gradient penalty (WGAN-GP):

$$L_{GP} = \lambda E_{\hat{\boldsymbol{x}}}[(\|\nabla_{\hat{\boldsymbol{x}}} D(\hat{\boldsymbol{x}})\|_2 - 1)^2]$$

Flow-based Models

Change of variables:

$$p_X(\boldsymbol{x}) = p_Z(\boldsymbol{f}(\boldsymbol{x}))\left|\det \frac{\partial \boldsymbol{f}^{-1}}{\partial \boldsymbol{x}}\right|$$

Training objective:

$$\log p_X(\boldsymbol{x}) = \log p_Z(\boldsymbol{f}$$

$$\partial \boldsymbol{x}$$

$$^{-1}(\boldsymbol{x})) + \log \qquad \det \frac{\partial \boldsymbol{f}^{-1}}{\partial \boldsymbol{x}}$$

Model	Type	Training	Sample Quality
VAE	Latent variable	ELBO maximization	Good diversity
GAN	Implicit	Adversarial	High quality
Flow-based	Invertible	Likelihood	Exact inference
Diffusion	Markov chain	Score matching	State-of-art

Table 5.7: Comparison of deep generative models

5.1.13 Case Studies

Case Study 1: Machine Translation with Seq2Seq +

Attention Problem: Translate English to French

Dataset: WMT'14 English-French (36M sentence pairs)

Architecture:

- Encoder: 4-layer bidirectional LSTM (1024 units)
- Decoder: 4-layer LSTM with attention
- Attention: Additive (Bahdanau) mechanism
- Embedding size: 512
- Dropout: 0.3

Results:

Model	BLEU Score	Parameters	Training Time
RNN Encoder-Decoder	28.3	85M	5 days
+ Attention	32.7	92M	6 days
+ Bidirectional	34.2	98M	7 days
+ 4-layer Deep	36.5	110M	9 days

Table 5.8: Machine translation results on WMT'14

Case Study 2: Sentiment Analysis with

LSTM Problem: Classify movie reviews as

positive/negative

Dataset: IMDB reviews (25K train, 25K test)

Architecture:

- Embedding layer: 300-dim (GloVe pre-trained)
- BiLSTM: 128 units each direction
- Global max pooling

- Dense: 128 units with ReLU
- Output: sigmoid for binary classification

Implementation:

Algorithm 11: Sentiment Analysis with BiLSTM

Input: Text sequence $\{w_1, w_2, ..., w_n\}$
Output: Sentiment probability $p \in [0, 1]$
begin
  $\boldsymbol{e}_i \leftarrow \text{Embedding}(w_i)$ for $i = 1..n$;
  $\overrightarrow{\boldsymbol{h}}_i \leftarrow \text{LSTM}_{\text{fwd}}(\boldsymbol{e}_i, \overrightarrow{\boldsymbol{h}}_{i-1})$;
  $\overleftarrow{\boldsymbol{h}}_i \leftarrow \text{LSTM}_{\text{bwd}}(\boldsymbol{e}_i, \overleftarrow{\boldsymbol{h}}_{i+1})$;
  $\boldsymbol{h}_i \leftarrow [\overrightarrow{\boldsymbol{h}}_i; \overleftarrow{\boldsymbol{h}}_i]$;
  $\boldsymbol{h}_{\text{pool}} \leftarrow \max_i \boldsymbol{h}_i$;
  $\boldsymbol{h}_{\text{dense}} \leftarrow \text{ReLU}(\boldsymbol{W}\boldsymbol{h}_{\text{pool}} + \boldsymbol{b})$;
  $p \leftarrow \sigma(\boldsymbol{w}_o \boldsymbol{h}_{\text{dense}} + b_o)$;
  return p

Results:

- Accuracy: 89.7%
- F1-score: 0.892
- Training time: 2 hours (GPU)

Case Study 3: Language Modeling with Transformer

Problem: Next word prediction
Dataset: WikiText-103 (100M tokens)
Architecture:

- 12-layer Transformer decoder
- 16 attention heads
- Embedding dimension: 768
- Feed-forward dimension: 3072
- Dropout: 0.1

Results:

Model	Perplexity	Parameters
LSTM (3-layer)	35.2	66M
Transformer (12-layer)	20.5	110M
GPT-2 (small)	18.3	124M
BERT-base	16.7	110M

Table 5.9: Language modeling results on WikiText-103

Case Study 4: Image Generation with VAE-GAN

Problem: Generate realistic face images
Dataset: CelebA (200K face images) **Architecture:**

- Encoder: 5 convolutional layers (64→128→256→512→512)
- Latent dimension: 256
- Decoder: 5 transpose convolutional layers
- Discriminator: 5 convolutional layers

Loss:

$$L = L_{VAE}(\boldsymbol{x}, \hat{\boldsymbol{x}}) + \lambda L_{GAN}(\boldsymbol{x}, \hat{\boldsymbol{x}})$$

Results:

- FID score: 42.3
- Inception score: 3.8
- Reconstruction error: 0.023

Practice Questions and Answers

Course Educational Objectives

- **CEO1:** Demonstrate the major technology trends driving Deep Learning
- **CEO2:** Build, train, and apply fully connected deep neural networks
- **CEO3:** Implement efficient (vectorized) neural networks
- **CEO4:** Analyse the key parameters and hyper parameters in neural networks
- **CEO5:** Demonstrate sequence modeling and Networks of Deep Learning

6.1 UNIT-I: Linear Algebra, Probability and Numeri-cal Computation

1. **(Bloom's Levels 1-5)** Explain the fundamental building blocks of linear algebra and their role in deep learning representations.

   Part a (Level 1 - Remember): Define scalars, vectors, matrices, and tensors with mathematical notation.

   - **Scalar:** A single number, denoted by lowercase italic letters (e.g., $a \in \mathrm{R}$)
   - **Vector:** An ordered array of numbers, denoted by bold lowercase ($\boldsymbol{v} \in \mathrm{R}^n$)
   - **Matrix:** A 2D array of numbers, denoted by bold uppercase ($\boldsymbol{M} \in \mathrm{R}^{m \times n}$)
   - **Tensor:** An array with more than two dimensions ($\mathsf{T} \in \mathrm{R}^{d_1 \times d_2 \times \ldots \times d_k}$)

   Part b (Level 2 - Understand): Explain how tensors represent different types of data in deep learning.

   - Grayscale images: 3D tensors (height × width × channels) where channels=1
   - RGB images: 3D tensors (height × width × 3) for color channels
   - Video data: 4D tensors (frames × height × width × channels)

- Mini-batches: Add batch dimension → 4D tensors (batch × height × width × channels)

Part c (Level 3 - Apply): Perform matrix multiplication on two 2 × 2 matrices and explain the result.

$$\boldsymbol{A} = \begin{matrix} 1 & 2 \\ 3 & \end{matrix}, \boldsymbol{B} = \begin{matrix} 5 & 6 \\ 7 & \end{matrix}$$

$$\boldsymbol{AB} = \begin{matrix} 1 \times 5 + 2 \times 7 & 1 \times 6 + 2 \times 8 \\ 3 \times 5 + 4 \times 7 & 3 \times 6 + 4 \times 8 \end{matrix} = \begin{matrix} 19 & 22 \\ 43 & 50 \end{matrix}$$

Part d (Level 4 - Analyze): Compare and contrast different matrix norms and their applications in regularization.

- L1 norm ($\|\boldsymbol{W}\|_1 = \sum_i |w_i|$): Promotes sparsity, feature selection
- L2 norm ($\|\boldsymbol{W}\| = \sqrt{\sum_i w_i^2}$): Weight decay, prevents
- Frobenius norm ($\|\boldsymbol{W}\|_F = \sqrt{\sum_{i,j} w_{ij}^2}$): Matrix generalization of L2
- Max norm ($\|\boldsymbol{W}\|_\infty = \max_i |w_i|$): Gradient clipping, stability

Part e (Level 5 - Evaluate): Justify the importance of eigen decomposition and SVD in dimensionality reduction like PCA.

- Eigen decomposition: $\boldsymbol{A} = \boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}^{-1}$ reveals principal directions of variation
- SVD: $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$ provides optimal low-rank approximations
- PCA: Project data onto top-k eigenvectors of covariance matrix

$$\text{Cov}(\boldsymbol{X}) = \frac{1}{n-1}\boldsymbol{X}^T\boldsymbol{X}$$

$$\boldsymbol{X}_{\text{PCA}} = \boldsymbol{X}\boldsymbol{V}_k \text{ (top k eigenvectors)}$$

- Preserves maximum variance while reducing dimensionality
- Minimizes reconstruction error $\|\boldsymbol{X} - \boldsymbol{X}\boldsymbol{V}_k\boldsymbol{V}_k^T\|_F^2$

2. **(Bloom's Levels 1-5)** Analyze probability theory foundations essential for under-standing uncertainty in deep learning predictions.

Part a (Level 1 - Remember): Define random variables and probability distribu-tions with examples.

- **Random Variable:** A variable whose values depend on outcomes of random phenomena
- **Discrete RV:** Takes countable values (e.g., coin flip $X \in \{H, T\}$)
- **Continuous RV:** Takes uncountable values (e.g., height $X \in \mathbb{R}^+$)
- **Probability Distribution:** $P(X = x)$ for discrete, $f_X(x)$ PDF for continuous

Part b (Level 2 - Understand): Explain the relationship between marginal and conditional probability with an example.

- **Marginal Probability:** $P(X = x) = \sum_y P(X = x, Y = y)$
- **Conditional Probability:** $P(Y = y|X = x) = \frac{P(X=x,Y=y)}{P}$
- Example: Weather (X) and traffic (Y)

$$P(\text{rain}) = 0.3 \quad \text{(marginal)}$$
$$P(\text{traffic}|\text{rain}) = 0.8 \quad \text{(conditional)}$$
$$P(\text{rain, traffic}) = 0.3 \times 0.8 = 0.24 \quad \text{(joint)}$$

Part c (Level 3 - Apply): Apply Bayes' Rule to a medical diagnosis problem with 99% accurate test for disease with 1% prevalence.

$$P(\text{disease}) = 0.01$$
$$P(\text{positive}|\text{disease}) = 0.99$$
$$P(\text{positive}|\text{no disease}) = 0.01$$
$$P(\text{disease}|\text{positive}) = \frac{0.99 \times 0.01}{0.99 \times 0.01 + 0.01 \times 0.99} = 0.5$$

Interpretation: Even with 99% accurate test, positive result means only 50% probabil-ity of having the disease due to low prevalence.

Part d (Level 4 - Analyze): Compare expectation, variance, and covariance and their roles in neural network training.

- **Expectation:** $\mathrm{E}[X] = \sum xP(X = x)$, average behavior
- **Variance:** $\mathrm{Var}(X) = \mathrm{E}[(X - \mathrm{E}[X])^2]$, spread of predictions
- **Covariance:** $\mathrm{Cov}(X, Y) = \mathrm{E}[(X - \mathrm{E}[X])(Y - \mathrm{E}[Y])]$, feature correlations
- Role in neural networks:
  - Batch normalization normalizes activations to zero mean, unit variance
  - Weight initialization uses variance scaling (Xavier/He)
  - Covariance matrix used in PCA for decorrelation
  - Gradient variance affects convergence stability

Part e (Level 5 - Evaluate): Evaluate how information theory concepts apply to loss functions in deep learning.

- **Entropy:** $H(P) = -\sum_x P(x) \log P(x)$, measures uncertainty

- **Cross-Entropy:** $H(P, Q) = -\sum_x P(x) \log Q(x)$, measures difference
- **KL Divergence:** $D_{KL}(P \| Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)}$
- Applications:
  - Classification loss: Cross-entropy between true labels P and predictions Q
  - $L_{CE} = -\sum_{i=}^{C} y_i \log(\hat{y}_i)$
  - Minimizing cross-entropy ≡ minimizing KL divergence
  - Information gain in decision trees uses entropy reduction

3. **(Bloom's Levels 1-5)** Demonstrate numerical computation challenges and optimiza-tion techniques in deep learning.

   Part a (Level 1 - Remember): Define overflow and underflow in numerical compu-tation.

   - **Underflow:** Numbers too small to represent (below ≈ 10^{-38} for single precision)
     - Example: $e^{-100} \approx 3.7 \times 10^{-44}$ underflows to 0
   - **Overflow:** Numbers too large to represent (above ≈ 10^{38})
     - Example: $e^{100} \approx 2.7 \times 10^{43}$ overflows to ∞

   Part b (Level 2 - Understand): Explain how softmax function is stabilized against numerical issues.

   - **Problem:** Softmax $\sigma(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$ can overflow
   - **Solution:** Subtract maximum value

$$\sigma(\boldsymbol{z})_i = \frac{e^{z_i - \max_k z_k}}{\sum_j e^{z_j - \max_k z_k}}$$

   - Ensures largest exponent ≤ 0, preventing overflow
   - Works because $\frac{e^{z_i}}{\sum e^{z_j}} = \frac{e^{z_i - c}}{\sum e^{z_j - c}}$

   Part c (Level 3 - Apply): Apply gradient descent to minimize $f(x) = x^2$ with learning rate 0.1, starting at $x = 5$.

$$x_0 = 5, \quad \nabla f(x) = 2x$$
$$x_1 = 5 - 0.1 \times (2 \times 5) = 5 - 1 = 4$$
$$x_2 = 4 - 0.1 \times 8 = 4 - 0.8 = 3.2$$
$$x_3 = 3.2 - 0.1 \times 6.4 = 3.2 - 0.64 = 2.56$$
$$x_4 = 2.56 - 0.1 \times 5.12 = 2.56 - 0.512 = 2.048$$

   Converges to $x = 0$ (minimum) after sufficient iterations.

   Part d (Level 4 - Analyze): Compare constrained and unconstrained optimization with Lagrangian multipliers.

- **Unconstrained:** $\min_x f(x)$ - gradient descent directly
- **Constrained:** $\min_x f(x)$ subject to $g(x) \leq 0$, $h(x) = 0$
- **Lagrangian:** $L(x, \lambda, \mu) = f(x) + \lambda g(x) + \mu h(x)$

- **KKT Conditions:**

$$\nabla_x \mathrm{L} = 0$$
$$\lambda \geq 0$$
$$\lambda g(x) = 0 (\text{complementary slackness})$$

Part e (Level 5 - Evaluate): Evaluate the solution of linear least squares via normal equations vs SVD.

- **Normal equations:** $\boldsymbol{X^T X\beta = X^T y}$
- Solution: $\boldsymbol{\beta = (X^T X)^{-1} X^T y}$
- Issues: $\boldsymbol{X^T X}$ can be ill-conditioned, condition number squared
- **SVD approach:** $\boldsymbol{X = U\Sigma V^T}$
- Solution: $\boldsymbol{\beta = V\Sigma^{-1} U^T y}$
- Advantages:
  - More numerically stable
  - Handles rank deficiency via pseudo-inverse
  - Condition number preserved, not squared

6.2 UNIT-II: Machine Learning and Deep Feed For-ward Networks

4. **(Bloom's Levels 1-5)** Analyze the fundamental concepts of machine learning and their relationship to deep learning.

   Part a (Level 1 - Remember): Define underfitting and overfitting with examples.

   - **Underfitting:** Model too simple, fails to capture data patterns
     - Example: Linear model on quadratic data
     - Training error high, validation error high
   - **Overfitting:** Model too complex, memorizes noise
     - Example: High-degree polynomial on small dataset
     - Training error low, validation error high

   Part b (Level 2 - Understand): Explain the role of hyperparameters and validation sets in model selection.

   - **Hyperparameters:** Settings not learned from data
     - Examples: Learning rate, network depth, regularization strength
   - **Validation Set:** Held-out data for hyperparameter tuning
   - Process:

(a) Split data: Training (60%), Validation (20%), Test (20%)
(b) Train models with different hyperparameters on training set
(c) Select best model based on validation performance
(d) Evaluate final model on test set

Part c (Level 3 - Apply): Calculate bias and variance for a given estimator.

- Given estimator $\hat{\theta}$ for true parameter θ:

$$\text{Bias} = \mathrm{E}[\hat{\theta}] - \theta$$

$$\text{Variance} = \mathrm{E}[(\hat{\theta} - \mathrm{E}[\hat{\theta}])^2]$$

$$\text{MSE} = \text{Bias}^2 + \text{Variance}$$

- Example: Sample mean $\bar{x}$ for population mean μ:

$$\mathrm{E}[\bar{x}] = \mu \text{ (unbiased)}$$

$$\text{Var}(\bar{x}) = \frac{\sigma^2}{n}$$

Part d (Level 4 - Analyze): Compare Maximum Likelihood Estimation and Bayesian inference approaches.

- **MLE:** $\theta_{\text{MLE}} = \arg\max_\theta P(\text{D}|\theta)$
  - Point estimate
  - Prone to overfitting with limited data
- **Bayesian:** $P(\theta|\text{D}) = \frac{P(\text{D}|\theta)P(\theta)}{P(\text{D})}$
  - Full posterior distribution
  - Prior $P(\theta)$ encodes domain knowledge
  - Naturally incorporates uncertainty
- Example: Bernoulli parameter estimation
  - MLE: $\hat{p} = \frac{\text{counts}}{n}$
  - Bayesian (Beta prior): $\hat{p} = \frac{\alpha + \text{counts}}{\alpha + \beta + n}$

Part e (Level 5 - Evaluate): Evaluate the challenges motivating deep learning over traditional machine learning.

- **Curse of Dimensionality:** Traditional methods fail in high dimensions
- **Feature Engineering:** Deep learning learns features automatically
- **End-to-End Learning:** No need for separate feature extraction
- **Scalability:** Deep learning benefits from more data and computation
- **Transfer Learning:** Pre-trained models adapt to new tasks
- Examples:

- ImageNet classification: Traditional ML (SIFT+BoW) ≈ 60%, Deep learning > 95%
- Speech recognition: GMM-HMM vs end-to-end deep learning

5. **(Bloom's Levels 1-5)** Demonstrate deep feedforward network architecture and train-ing.

Part a (Level 1 - Remember): Describe the XOR problem and why it's important.

- **XOR Truth Table:**

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

- **Importance:** Not linearly separable, requires hidden layer
- Historical significance: Showed limitations of perceptrons

Part b (Level 2 - Understand): Explain the architecture of a 2-layer network solving XOR.

- **Hidden Layer (2 neurons):** $\boldsymbol{h} = \sigma(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)})$
- **Output Layer:** $y = \sigma(\boldsymbol{w}^{(2)T}\boldsymbol{h} + b^{(2)})$
- Example weights:

$$\boldsymbol{W}^{(1)} = \begin{matrix} 1 & 1 \\ 1 & 1 \end{matrix}, \quad \boldsymbol{b}^{(1)} = \begin{matrix} 0 \\ -1 \end{matrix}$$

$$\boldsymbol{w}^{(2)} = \begin{matrix} 1 \\ -2 \end{matrix}, \quad b^{(2)} = 0$$

- Forward pass for (0,0): $h = [0, 0]^T$, output = 0
- Forward pass for (0,1): $h = [1, 0]^T$, output = 1

Part c (Level 3 - Apply): Apply gradient-based learning to update one weight in XOR network.

- Loss: $L = \frac{1}{2}(y - \hat{y})^2$ for one example
- Example: Input (1,1), target 0, learning rate $\eta = 0.1$
- Forward pass: $h = [1, 1]^T$, $\hat{y} = \sigma(1 - 2) = \sigma(-1) \approx 0.27$
- Backward pass:

$$\frac{\partial L}{\partial w_1^{(2)}} = (y - \hat{y}) \cdot \hat{y}(1 - \hat{y}) \cdot h_1$$

$$= (0 - 0.27) \times 0.27 \times 0.73 \times 1 \approx -0.053$$

$$w_1^{(2)} \leftarrow 1 - 0.1 \times (-0.053) = 1.0053$$

Function	Formula	Advantages	Disadvantages
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	Smooth, probabilistic	Vanishing gradient
Tanh	$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$	Zero-centered	Vanishing gradient
ReLU	$\max(0, x)$	No saturation, sparse	Dead neurons
Leaky ReLU	$\max(0.01x, x)$	Avoids dead neurons	Parameter selection
ELU	x if $x > 0$, $\alpha(e^{x} - 1)$ else	Smooth, zero-mean	Expensive

Part d (Level 4 - Analyze): Compare different hidden unit activation functions.

Part e (Level 5 - Evaluate): Evaluate architecture design choices for a given prob-lem.

- Problem: Classify 28×28 grayscale images into 10 classes
- **Input:** 784-dimensional vector
- **Hidden Layers:** Depth vs width trade-off
  - Option A: 1 hidden layer, 1024 units (1.2M parameters)
  - Option B: 3 hidden layers, 256 units each (1.3M parameters)
- **Evaluation criteria:**
  - Capacity: Both similar parameter count
  - Expressivity: Deeper networks more expressive
  - Training: Deeper harder to optimize (skip connections needed)
  - Regularization: Deeper might need dropout/batch norm
- **Recommendation:** 3 layers with batch normalization for MNIST

6. **(Bloom's Levels 1-5)** Analyze backpropagation algorithm and its variants.

Part a (Level 1 - Remember): State the chain rule of calculus.

$$\frac{d}{dx} f(g(x)) = f'(g(x))g'(x)$$

For multivariable functions:

$$\frac{\partial}{\partial x_i} f(g_1(x), \ldots, g_m(x)) = \sum_{j=1}^{m} \frac{\partial f}{\partial g_j} \cdot \frac{\partial g_j}{\partial x_i}$$

Part b (Level 2 - Understand): Explain the forward and backward passes in backpropagation.

- **Forward Pass:** Compute all activations

$$\boldsymbol{h}^{(1)} = \sigma(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)})$$
$$\boldsymbol{h}^{(2)} = \sigma(\boldsymbol{W}^{(2)}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)})$$
$$\hat{y} = \boldsymbol{w}^{(3)T}\boldsymbol{h}^{(2)} + b^{(3)}$$

- **Backward Pass:** Compute gradients

$$\frac{\partial L}{\partial \boldsymbol{W}^{(3} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \boldsymbol{W}^{(3}$$

$$\frac{\partial L}{\partial \boldsymbol{W}^{(2)}} = \frac{\partial L}{\partial \boldsymbol{h}^{(2)}} \cdot \frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{W}^{(2)}}$$

Part c (Level 3 - Apply): Apply backpropagation to a simple 2-layer network with MSE loss.

- Network: 2 inputs, 2 hidden, 1 output
- Forward pass formulas:

$$h_1 = \sigma(w_{11}x_1 + w_{21}x_2 + b_1)\ h_2 = \sigma(w_{12}x_1 + w_{22}x_2 + b_2)\ \hat{y} = v_1h_1 + v_2h_2 + b_3$$

- Gradients for output weights:

$$\frac{\partial L}{\partial v_i} = (y - \hat{y})(-1) \cdot h_i$$

- Gradients for hidden weights:

$$\frac{\partial L}{\partial w_{ij}} = (y - \hat{y})(-v_i) \cdot h_i(1 - h_i) \cdot x_j$$

Part d (Level 4 - Analyze): Compare different differentiation algorithms.

- **Manual Derivation:** Exact, but tedious for complex networks
- **Symbolic Differentiation:** Exact, but expression swell
- **Numerical Differentiation:** Approximate, slow, error-prone
- **Automatic Differentiation (AD):**
  - Forward mode: Efficient for few inputs
  - Reverse mode (backprop): Efficient for many parameters
  - Computational graph representation
  - Exact to machine precision

Part e (Level 5 - Evaluate): Evaluate computational complexity of backpropagation vs alternative approaches.

- Forward pass: $O(\sum_l n_l n_{l-1})$ operations
- Backward pass: Same complexity as forward pass
- Total: $O(2 \times \text{network size})$ per example
- Compare:
  - Numerical differentiation: $O(n_{\text{params}} \times \text{forward})$, impractical
  - Symbolic: Expression explosion
- **Memory requirements:** Store all intermediate activations for backward pass
- **Trade-offs:** Time vs memory (checkpointing reduces memory)

6.3 UNIT-III: Regularization and Optimization of DL Models

7. **(Bloom's Levels 1-5)** Analyze regularization techniques for preventing overfitting.

Part a (Level 1 - Remember): Define regularization and list common techniques.

- **Definition:** Any modification to learning algorithm intended to reduce general-ization error
- **Common techniques:**
  - Parameter norm penalties (L1, L2)
  - Dataset augmentation
  - Early stopping
  - Dropout
  - Batch normalization
  - Multi-task learning

Part b (Level 2 - Understand): Explain how L2 regularization modifies the loss function and gradient.

- Original loss: $J(\boldsymbol{\theta})$
- Regularized loss: $\tilde{J}(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \frac{\lambda}{2}\|\boldsymbol{\theta}\|_2^2$
- Gradient: $\nabla_{\boldsymbol{\theta}}\tilde{J} = \nabla_{\boldsymbol{\theta}}J + \lambda\boldsymbol{\theta}$
- Update rule: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon(\nabla_{\boldsymbol{\theta}}J + \lambda\boldsymbol{\theta})$
- Effect: Weights decay by factor $(1 - \epsilon\lambda)$ before gradient update

Part c (Level 3 - Apply): Apply L1 and L2 regularization to linear regression and compare solutions.

- Problem: $\min_{\boldsymbol{w}} \|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|_2^2$
- L2 solution: $\boldsymbol{w}_{L2} = (\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T\boldsymbol{y}$
- L1 solution: No closed form, promotes sparsity
  Example with $\boldsymbol{X} = \begin{matrix}1 & 2\\ 3 & 4\end{matrix}$, $\boldsymbol{y} = \begin{matrix}5\\ 11\end{matrix}$
  - Unregularized: $\boldsymbol{w} = [1, 2]^T$
  - L2 ($\lambda = 0.1$): $\boldsymbol{w} \approx [0.98, 1.96]^T$ (shrunk)
  - L1 ($\lambda = 0.1$): $\boldsymbol{w} = [0.95, 1.90]^T$ (sparser if λ large)

Part d (Level 4 - Analyze): Compare dropout with bagging as ensemble methods.

8. Dropout ≈ bagging of all possible subnetworks

Part e (Level 5 - Evaluate): Evaluate the effectiveness of different regularization methods for a deep CNN.

Aspect	Dropout	Bagging
Training	Train one network with dropout	Train multiple networks independently
Models trained	2^n subnetworks implicitly	k explicit models
Inference	Weight scaling approximation	Average predictions
Parameters	Single set of weights	k independent parameter sets
Training time	Similar to one network	k times longer
Memory	Single model	k models

- Dataset: CIFAR-10, 50k training images
- Baseline CNN: 6 conv layers + 2 dense layers
- Regularization comparison:

Method	Training Acc	Test Acc
No regularization	99.8%	82.3%
L2 ($\lambda = 0.0005$)	98.2%	85.7%
Dropout (p=0.5)	97.5%	86.9%
Batch Norm	99.5%	87.2%
Data Augmentation	98.8%	89.1%
All combined	97.2%	91.3%

- Best practice: Combine multiple methods

9. **(Bloom's Levels 1-5)** Analyze optimization algorithms for training deep networks.

Part a (Level 1 - Remember): List the challenges in neural network optimization.

- Ill-conditioning of Hessian
- Local minima and saddle points
- Vanishing/exploding gradients
- Flat regions and plateaus
- Noisy gradients from mini-batches
- Parameter initialization sensitivity

Part b (Level 2 - Understand): Explain momentum and its benefits over basic SGD.

- SGD: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla J(\boldsymbol{\theta}_t)$
- Momentum:

$$\boldsymbol{v}_{t+1} = \alpha \boldsymbol{v}_t - \eta \nabla J(\boldsymbol{\theta}_t)$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \boldsymbol{v}_{t+1}$$

- Benefits:
  - Accelerates in consistent gradient directions
  - Dampens oscillations
  - Escapes local minima with momentum
  - Faster convergence

Part c (Level 3 - Apply): Apply Adam optimizer update for one step given gradi-ents.

- Parameters: $\boldsymbol{\theta}_0 = [1, 2]^T$, $\nabla J = [0.1, 0.2]^T$
- Adam settings: $\eta = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$
- First moment: $\boldsymbol{m}_1 = 0.9 \times 0 + 0.1 \times [0.1, 0.2] = [0.01, 0.02]$
- Second moment: $\boldsymbol{v}_1 = 0.999 \times 0 + 0.001 \times [0.01, 0.04] = [1 \times 10^{-5}, 4 \times 10^{-5}]$
- Bias correction: $\hat{\boldsymbol{m}}_1 = \boldsymbol{m}_1/(1 - 0.9) = [0.1, 0.2]$
- $\hat{\boldsymbol{v}}_1 = \boldsymbol{v}_1/(1 - 0.999) \approx [0.01, 0.04]$
- Update: $\boldsymbol{\theta}_1 = [1, 2] - 0.001 \times [0.1, 0.2]/(\sqrt{[0.01, 0.04]} + 10^{-8})$
- $\boldsymbol{\theta}_1 \approx [1 - 0.001, 2 - 0.001] = [0.999, 1.999]$

Part d (Level 4 - Analyze): Compare different parameter initialization strategies.

Method	Distribution	Use Case
Xavier/Glorot	$N(0, \frac{2}{n_{in}+n_{out}})$	tanh, sigmoid
He	$N(0, \frac{2}{n_i})$	ReLU
LeCun	$N(0, \frac{1}{n_i})$	SELU
Orthogonal	$\boldsymbol{W}^T\boldsymbol{W} = \boldsymbol{I}$	RNNs
Zero	$\boldsymbol{W} = 0$	Biases only

10. Rationale: Maintain activation variance across layers

Part e (Level 5 - Evaluate): Evaluate second-order optimization methods vs first-order methods.

- **First-order:** Use gradient only (SGD, Adam)
  - $O(n)$ per iteration
  - Linear convergence rate
  - Robust to noise
- **Second-order:** Use Hessian information
  - Newton: $O(n^3)$ per iteration
  - Quasi-Newton (BFGS): $O(n^2)$ per iteration
  - Quadratic convergence near optimum
  - Sensitive to noise, non-convexity
- **Trade-offs:**
- Recommendation: Adam for deep learning, L-BFGS for small problems

Method	Per-iteration cost	Convergence rate	Memory
SGD	$O(n)$	Linear	$O(n)$
Adam	$O(n)$	Linear	$O(2n)$
BFGS	$O(n^2)$	Superlinear	$O(n^2)$
L-BFGS	$O(nm)$	Superlinear	$O(nm)$
Newton	$O(n^3)$	Quadratic	$O(n^2)$

6.4 UNIT-IV: Convolutional Networks

11. **(Bloom's Levels 1-5)** Analyze the convolution operation and its properties in CNNs.

Part a (Level 1 - Remember): Define the convolution operation for 2D images.

$$S(i,j) = (I * K)(i,j) = \sum_{m=-a}^{a} \sum_{n=-b}^{b} I(i-m, j-n) K(m,n)$$

where kernel K has size $(2a+1) \times (2b+1)$.

Part b (Level 2 - Understand): Explain sparse interactions and parameter sharing in CNNs.

- **Sparse interactions:** Each output connects to local input region
  - Size: kernel size $k \times k$
  - Receptive field grows with depth
- **Parameter sharing:** Same kernel applied everywhere
  - CNN parameters: $k^2 \times c_{\text{in}} \times c_{\text{out}}$
  - FC parameters: $h \times w \times c_{\text{in}} \times c_{\text{out}}$
  - Example: 224×224×3 input, 64 filters, 3×3 kernel
    - CNN: $3 \times 3 \times 3 \times 64 = 1,728$ parameters
    - FC: $224 \times 224 \times 3 \times 64 \approx 9.6$ million parameters

Part c (Level 3 - Apply): Apply convolution to a 4×4 input with 3×3 kernel, stride 1, no padding.

- Input I:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

- Kernel K:

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

- Output size: $(4 - 3 + 1) \times (4 - 3 + 1) = 2 \times 2$
- Output at (1,1):

$$\begin{aligned} &1 \times 1 + 2 \times 0 + 3 \times (-1) \\ &+ 5 \times 1 + 6 \times 0 + 7 \times (-1) \\ &+ 9 \times 1 + 10 \times 0 + 11 \times (-1) \\ &= 1 - 3 + 5 - 7 + 9 - 11 = -6 \end{aligned}$$

- Full output:

$$\begin{matrix} -6 & -6 \\ -6 & -6 \end{matrix}$$

Part d (Level 4 - Analyze): Compare different pooling operations and their effects.

Pooling	Formula	Invariance	Use Case
Max	$\max(x_{ij})$	Translation	Feature
Average	$\frac{1}{k^2}\sum x_{ij}$	Smooth	Background
Global Avg	$\frac{1}{HW}\sum x_{ij}$	Global	Before
L	$\frac{1}{k^2}\sum x_{ij}^2$	Energy	Normalization

12. Effects:

- Dimensionality reduction: 2×2 pooling reduces size by 75%
- Translation invariance: Small shifts don't change output
- Increased receptive field

Part e (Level 5 - Evaluate): Evaluate the design choices for a CNN architecture for image classification.

- Architecture: VGG-style for ImageNet
- Evaluation: Good accuracy (92.7% top-5), but 138M parameters
- Modern improvements: Batch norm, residual connections, depthwise separable

6.5 UNIT-V: Sequence Modeling

13. **(Bloom's Levels 1-5)** Analyze recurrent neural networks and their variants for se-quence modeling.

Part a (Level 1 - Remember): Define RNN and write the forward pass equations.

$$\boldsymbol{h}^{(t)} = \tanh(\boldsymbol{W}_{hh}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{xh}\boldsymbol{x}^{(t)} + \boldsymbol{b}_h)$$

$$\boldsymbol{o}^{(t)} = \boldsymbol{W}_{hy}\boldsymbol{h}^{(t)} + \boldsymbol{b}_y$$

Layer	Operation	Output size	Params
Input	-	224× 224× 3	0
Conv1	64× 3× 3, stride 1	224× 224× 64	1.7K
Pool1	Max 2× 2, stride 2	× 112× 112 64	0
Conv2	128× 3× 3	112× 112× 128	73K
Pool2	Max 2× 2	56× 56× 128	0
Conv3	256× 3× 3	56× 56× 256	295K
Pool3	Max 2× 2	28× 28× 256	0
Conv4	512× 3× 3	28× 28× 512	1.18M
Pool4	Max 2× 2	14× 14× 512	0
Conv5	512× 3× 3	14× 14× 512	2.36M
Pool5	Max 2× 2	7 × 7 × 512	0
FC1	Dense 4096	4096	102.7M
FC2	Dense 4096	4096	16.8M
Output	Dense 1000	1000	4.1M

Part b (Level 2 - Understand): Explain unfolding computational graphs and back-propagation through time.

- Unfolding: Convert RNN to deep feedforward network with T copies
- Shared weights across time steps
- BPTT:
  (a) Forward pass through all time steps
  (b) Compute loss at each step (sum over time)
  (c) Backward pass from $t = T$ to $t = 1$
  (d) Accumulate gradients for shared weights
- Gradient through time:

$$\frac{\partial \mathrm{L}}{\partial \boldsymbol{h}^{(t)}} = \frac{\partial \mathrm{L}}{\partial \boldsymbol{o}^{(t)}} \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}} + \frac{\partial \mathrm{L}}{\partial \boldsymbol{h}^{(t+1)}} \frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}}$$

Part c (Level 3 - Apply): Apply RNN to a character-level language modeling task.

- Vocabulary: {h, e, l, o} (4 chars)
- Input ”hello” → output ”ello”
- One-hot encoding: h=[1,0,0,0], e=[0,1,0,0], etc.
- Forward pass for first character:

$$\boldsymbol{x}^{(1)} = [1, 0, 0, 0]^T$$

$$\boldsymbol{h}^{(1)} = \tanh(\boldsymbol{W}_{hh}\mathbf{0} + \boldsymbol{W}_{xh}\boldsymbol{x}^{(1)} + \boldsymbol{b}_h)$$

$$\boldsymbol{o}^{(1)} = \boldsymbol{W}_{hy}\boldsymbol{h}^{(1)} + \boldsymbol{b}_y$$

$$\hat{\boldsymbol{y}}^{(1)} = \mathrm{softmax}(\boldsymbol{o}^{(1)}) \quad \text{(should predict 'e')}$$

- Loss: $L = -\sum_t \log(\hat{y}^{(t)}_{target})$

Part d (Level 4 - Analyze): Compare LSTM and GRU architectures for handling long-term dependencies.

Component	LSTM	GRU
Gates	Forget, Input, Output (3)	Update, Reset (2)
State	Cell $c^{(t)}$, Hidden $h^{(t)}$	Hidden $h^{(t)}$
Parameters	4×	3×
Gradient flow	Additive through cell	Gated
Long-term memory	Explicit cell	Through update gate

14. LSTM equations:

$$f^{(t)} = \sigma(W_f[h^{(t-1)}; x^{(t)}] + b_f)$$
$$i^{(t)} = \sigma(W_i[h^{(t-1)}; x^{(t)}] + b_i)$$
$$\tilde{c}^{(t)} = \tanh(W_c[h^{(t-1)}; x^{(t)}] + b_c)$$
$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + i^{(t)} \odot \tilde{c}^{(t)}$$
$$o^{(t)} = \sigma(W_o[h^{(t-1)}; x^{(t)}] + b_o)$$
$$h^{(t)} = o^{(t)} \odot \tanh(c^{(t)})$$

15. GRU equations:

$$z^{(t)} = \sigma(W_z[h^{(t-1)}; x^{(t)}] + b_z)$$
$$r^{(t)} = \sigma(W_r[h^{(t-1)}; x^{(t)}] + b_r)$$
$$\tilde{h}^{(t)} = \tanh(W_h[r^{(t)} \odot h^{(t-1)}; x^{(t)}] + b_h)$$
$$h^{(t)} = (1 - z^{(t)}) \odot h^{(t-1)} + z^{(t)} \odot \tilde{h}^{(t)}$$

Part e (Level 5 - Evaluate): Evaluate encoder-decoder architectures with attention for machine translation.

- Architecture:
  - Encoder: Bidirectional LSTM processes source sentence
  - Attention: Computes context vector for each decoder step
  - Decoder: LSTM generates target sentence
- Attention mechanism:

$$e^{(t,i)} = \boldsymbol{v}^T_a \tanh(\boldsymbol{W}_a \boldsymbol{s}^{(t-1)} + \boldsymbol{U}_a \boldsymbol{h}^{(i)})$$
$$\alpha^{(t,i)} = \frac{\exp(e^{(t,i)})}{\sum_j \exp(e^{(t,j)})}$$
$$\boldsymbol{c}^{(t)} = \sum_i \alpha^{(t,i)} \boldsymbol{h}^{(i)}$$

- Evaluation metrics:
  - BLEU score: n-gram precision with brevity penalty
  - Perplexity: $2^{-\frac{1}{N}\sum \log P(w_t|w_{<t})}$
  - Human evaluation
- Results on WMT'14 English-French:

Model	BLEU	Parameters
RNN Encoder-Decoder	28.3	85M
+ Attention	32.7	92M
+ Bidirectional	34.2	98M
+ 4-layer Deep	36.5	110M
Transformer	41.8	65M

16. **(Bloom's Levels 1-5)** Analyze autoencoders and deep generative models.

Part a (Level 1 - Remember): Define autoencoder and its components.

- Autoencoder: Neural network that learns to reconstruct input
- Components:
  - Encoder: $\boldsymbol{h} = f(\boldsymbol{W}_e\boldsymbol{x} + \boldsymbol{b}_e)$
  - Decoder: $\hat{\boldsymbol{x}} = g(\boldsymbol{W}_d\boldsymbol{h} + \boldsymbol{b}_d)$
  - Loss: $\mathrm{L} = ||\boldsymbol{x} - \hat{\boldsymbol{x}}||^2$

Part b (Level 2 - Understand): Explain denoising and sparse autoencoders.

- **Denoising autoencoder:**
  - Corrupt input: $\tilde{\boldsymbol{x}} = \boldsymbol{x} + \boldsymbol{\epsilon}$
  - Reconstruct clean input: $\mathrm{L} = ||\boldsymbol{x} - \mathrm{decoder}(\mathrm{encoder}(\tilde{\boldsymbol{x}}))||^2$
  - Learns robust features
- **Sparse autoencoder:**
  - Add sparsity penalty on hidden units
  - $\mathrm{L} = ||\boldsymbol{x} - \hat{\boldsymbol{x}}||^2 + \lambda \sum_j \mathrm{KL}(\rho||\hat{\rho}_j)$
  - $\hat{\rho}_j = \frac{1}{N}\sum_i h_j^{(i)}$ (average activation)
  - Encourages selective activation

Part c (Level 3 - Apply): Apply VAE to generate MNIST digits.

- Encoder: 2 hidden layers → outputs $\boldsymbol{\mu}$, $\log \boldsymbol{\sigma}^2$ (latent dim=2)
- Reparameterization: $\boldsymbol{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$, $\boldsymbol{\epsilon} \sim \mathrm{N}(0, \boldsymbol{I})$
- Decoder: $\boldsymbol{z}$ → hidden layers → 784-dim output with sigmoid

- Loss:

$$\mathrm{L} = \underbrace{\mathrm{E}_q[\log p(\boldsymbol{x}|\boldsymbol{z})]}_{\text{Reconstruction}} - \underbrace{\mathrm{KL}(q(\boldsymbol{z}|\boldsymbol{x}) \| p(\boldsymbol{z}))}_{\text{Regularization}}$$

- KL term for Gaussian:

$$\mathrm{L}_{\mathrm{KL}} = -\frac{1}{2}\sum_{j=1}^{d}(1 + \log \sigma_j^2 - \mu_j^2 - \sigma_j^2)$$

- Results: Generate new digits by sampling $\boldsymbol{z} \sim \mathrm{N}(0, \boldsymbol{I})$

Part d (Level 4 - Analyze): Compare GANs and VAEs as generative models.

Aspect	VAE	GAN
Training	Likelihood maximization	Adversarial
Loss	ELBO + reconstruction	Min-max game
Latent space	Continuous, structured	Implicit
Sample quality	Good diversity, slightly blurry	Sharp, realistic
Mode coverage	Covers all modes	May miss modes
Training stability	Stable	Unstable
Inference	Explicit encoder	No encoder

17. GAN objective:

$$\min_G \max_D V(D, G) = \mathrm{E}_{x \sim p_{\text{data}}}[\log D(x)] + \mathrm{E}_{z \sim p_z}[\log(1 - D(G(z)))]$$

Part e (Level 5 - Evaluate): Evaluate deep generative models for image synthesis.

- **Criteria:**
  - Sample quality (FID, Inception score)
  - Diversity
  - Training stability
  - Controllability
- Models comparison on CelebA (face generation):

Model	FID (↓)	Inception (↑)	Training Time
VAE	85.3	2.1	1 day
DCGAN	42.7	3.2	2 days
WGAN-GP	38.2	3.5	2 days
StyleGAN	4.5	4.8	7 days

Diffusion	3.2	5.1	10 days

- State-of-art: Diffusion models (DALL-E 2, Stable Diffusion)

Answer Key Summary

Each question covers:

- **Level 1 (Remember):** Definitions, basic concepts, formulas
- **Level 2 (Understand):** Explanations, interpretations, relationships
- **Level 3 (Apply):** Calculations, implementations, examples
- **Level 4 (Analyze):** Comparisons, contrasts, structural analysis
- **Level 5 (Evaluate):** Judgments, recommendations, evaluations

Questions

6.6 UNIT-I: Linear Algebra, Probability and Numeri-cal Computation (10 Questions)

1. **(Bloom's Levels 1-5)** Explain the fundamental building blocks of linear algebra and their role in deep learning representations.

   Part a (Level 1 - Remember): Define scalars, vectors, matrices, and tensors with mathematical notation. **Part b (Level 2 - Understand):** Explain how tensors represent different types of data in deep learning. **Part c (Level 3 - Apply):** Perform matrix multiplication on two 2 × 2 matrices and explain the result. **Part d (Level 4 - Analyze):** Compare and contrast different matrix norms and their applications in regularization. **Part e (Level 5 - Evaluate):** Justify the importance of eigen decomposition and SVD in dimensionality reduction like PCA.

2. **(Bloom's Levels 1-5)** Analyze matrix operations and their computational properties in neural networks.

   Part a (Level 1 - Remember): List the basic matrix operations used in neural networks. **Part b (Level 2 - Understand):** Explain the concept of matrix transpose and its role in backpropagation. **Part c (Level 3 - Apply):** Compute the dot product of two vectors and explain its geometric meaning. **Part d (Level 4 - Analyze):** Compare the computational complexity of different matrix multiplication algorithms. **Part e (Level 5 - Evaluate):** Evaluate the importance of efficient matrix operations in deep learning frameworks.

3. **(Bloom's Levels 1-5)** Demonstrate understanding of special types of matrices used in deep learning.

   Part a (Level 1 - Remember): Define identity, diagonal, and symmetric matrices with examples. **Part b (Level 2 - Understand):** Explain the

properties of orthog-onal matrices and their applications. **Part c (Level 3 - Apply):** Verify if a given

matrix is positive definite using eigenvalue analysis. **Part d (Level 4 - Analyze):** Compare the properties of different matrix types and their use in neural networks. **Part e (Level 5 - Evaluate):** Evaluate the role of the Gram matrix in style transfer applications.

4. **(Bloom's Levels 1-5)** Analyze vector and matrix norms and their applications in regularization.

   Part a (Level 1 - Remember): Define L1, L2, and Frobenius norms with mathemat-ical formulas. **Part b (Level 2 - Understand):** Explain the geometric interpretation of different norms. **Part c (Level 3 - Apply):** Calculate L1 and L2 norms for a given vector and matrix. **Part d (Level 4 - Analyze):** Compare the effects of L1 vs L2 reg-ularization on model weights. **Part e (Level 5 - Evaluate):** Evaluate the suitability of different norms for specific regularization tasks.

5. **(Bloom's Levels 1-5)** Demonstrate eigen decomposition and its applications in di-mensionality reduction.

   Part a (Level 1 - Remember): Define eigenvalues and eigenvectors with the charac-teristic equation. **Part b (Level 2 - Understand):** Explain the geometric meaning of eigenvectors as principal directions. **Part c (Level 3 - Apply):** Compute eigenvalues and eigenvectors for a 2 2 matrix. **Part d (Level 4 - Analyze):** Compare eigen decomposition with singular value decomposition. **Part e (Level 5 - Evaluate):** Evaluate the use of eigen decomposition in PCA for face recognition.

6. **(Bloom's Levels 1-5)** Analyze probability theory foundations essential for deep learn-ing.

   Part a (Level 1 - Remember): Define random variables and probability distribu-tions with examples. **Part b (Level 2 - Understand):** Explain the relationship between marginal and conditional probability. **Part c (Level 3 - Apply):** Apply Bayes' Rule to a medical diagnosis problem. **Part d (Level 4 - Analyze):** Compare expectation, variance, and covariance in neural network training. **Part e (Level 5 - Evaluate):** Evaluate how information theory concepts apply to loss functions.

7. **(Bloom's Levels 1-5)** Demonstrate understanding of probability distributions in deep learning.

   Part a (Level 1 - Remember): List common probability distributions used in deep learning. **Part b (Level 2 - Understand):** Explain the difference between discrete and continuous distributions. **Part c (Level 3 - Apply):** Calculate the probability of an event using Gaussian distribution. **Part d (Level 4 - Analyze):** Compare Bernoulli, Binomial, and Multinomial distributions. **Part e (Level 5 - Evaluate):** Evaluate the choice of prior distributions in Bayesian neural networks.

8. **(Bloom's Levels 1-5)** Analyze information theory concepts in deep learning.

Part a (Level 1 - Remember): Define entropy, cross-entropy, and KL divergence. **Part b (Level 2 - Understand):** Explain the relationship between entropy and un-certainty. **Part c (Level 3 - Apply):** Calculate cross-entropy loss for a classification

example. **Part d (Level 4 - Analyze):** Compare cross-entropy with mean squared error for classification. **Part e (Level 5 - Evaluate):** Evaluate the use of mutual information in feature selection.

9. **(Bloom's Levels 1-5)** Demonstrate numerical computation challenges in deep learn-ing.

   Part a (Level 1 - Remember): Define overflow and underflow in numerical compu-tation. **Part b (Level 2 - Understand):** Explain how softmax function is stabilized against numerical issues. **Part c (Level 3 - Apply):** Apply gradient descent to minimize a simple quadratic function. **Part d (Level 4 - Analyze):** Compare con-strained and unconstrained optimization with Lagrangian multipliers. **Part e (Level 5 - Evaluate):** Evaluate the solution of linear least squares via normal equations vs SVD.

10. **(Bloom's Levels 1-5)** Analyze gradient-based optimization methods.

    Part a (Level 1 - Remember): Define gradient and its geometric interpretation. **Part b (Level 2 - Understand):** Explain the concept of learning rate in gradient descent. **Part c (Level 3 - Apply):** Perform gradient descent on a multivariate function. **Part d (Level 4 - Analyze):** Compare gradient descent, stochastic gradient descent, and mini-batch gradient descent. **Part e (Level 5 - Evaluate):** Evaluate the convergence properties of gradient descent for convex vs non-convex functions.

6.7 UNIT-II: Machine Learning and Deep Feed For-ward Networks (10 Questions)

11. **(Bloom's Levels 1-5)** Analyze the fundamental concepts of machine learning.

    Part a (Level 1 - Remember): Define underfitting and overfitting with examples. **Part b (Level 2 - Understand):** Explain the role of hyperparameters and vali-dation sets. **Part c (Level 3 - Apply):** Calculate bias and variance for a given estimator. **Part d (Level 4 - Analyze):** Compare Maximum Likelihood Estima-tion and Bayesian inference. **Part e (Level 5 - Evaluate):** Evaluate the challenges motivating deep learning over traditional ML.

12. **(Bloom's Levels 1-5)** Demonstrate understanding of supervised and unsupervised learning.

    Part a (Level 1 - Remember): Define supervised and unsupervised learning with examples. **Part b (Level 2 - Understand):** Explain the difference between classi-fication and regression. **Part c (Level 3 - Apply):** Apply k-means clustering to a simple dataset. **Part d (Level 4 - Analyze):** Compare generative and discriminative models. **Part e (Level 5 - Evaluate):** Evaluate

the suitability of different learning paradigms for specific problems.

13. **(Bloom's Levels 1-5)** Analyze stochastic gradient descent and its variants.
    Part a (Level 1 - Remember): Define stochastic gradient descent algorithm. **Part b (Level 2 - Understand):** Explain the benefits of mini-batch training. **Part c (Level 3 - Apply):** Implement one step of SGD for linear regression. **Part d (Level 4 - Analyze):** Compare batch, mini-batch, and stochastic gradient descent. **Part e (Level 5 - Evaluate):** Evaluate the convergence properties of SGD with different learning rate schedules.

14. **(Bloom's Levels 1-5)** Demonstrate deep feedforward network architecture.
    Part a (Level 1 - Remember): Describe the XOR problem and why it's important. **Part b (Level 2 - Understand):** Explain the architecture of a 2-layer network solving XOR. **Part c (Level 3 - Apply):** Apply gradient-based learning to update weights in XOR network. **Part d (Level 4 - Analyze):** Compare different hidden unit activation functions. **Part e (Level 5 - Evaluate):** Evaluate architecture design choices for a given problem.

15. **(Bloom's Levels 1-5)** Analyze gradient-based learning in feedforward networks.
    Part a (Level 1 - Remember): Define the loss function and its role in learning. **Part b (Level 2 - Understand):** Explain the concept of gradient descent for neural networks. **Part c (Level 3 - Apply):** Compute gradients for a simple neural net-work manually. **Part d (Level 4 - Analyze):** Compare different loss functions for regression and classification. **Part e (Level 5 - Evaluate):** Evaluate the impact of learning rate on convergence.

16. **(Bloom's Levels 1-5)** Demonstrate understanding of hidden units and their proper-ties.
    Part a (Level 1 - Remember): List common hidden unit activation functions. **Part b (Level 2 - Understand):** Explain the vanishing gradient problem with sigmoid. **Part c (Level 3 - Apply):** Compute the output of different activation functions for given inputs. **Part d (Level 4 - Analyze):** Compare ReLU, Leaky ReLU, and ELU activation functions. **Part e (Level 5 - Evaluate):** Evaluate the choice of activation function for different network depths.

17. **(Bloom's Levels 1-5)** Analyze architecture design principles for neural networks.
    Part a (Level 1 - Remember): Define universal approximation theorem. **Part b (Level 2 - Understand):** Explain the trade-off between network depth and width. **Part c (Level 3 - Apply):** Design a network architecture for MNIST classification. **Part d (Level 4 - Analyze):** Compare shallow and deep networks in terms of capacity and generalization. **Part e (Level 5 - Evaluate):** Evaluate the impact of skip connections on network design.

18. **(Bloom's Levels 1-5)** Demonstrate backpropagation algorithm.
    Part a (Level 1 - Remember): State the chain rule of calculus. **Part b (Level 2 - Understand):** Explain the forward and backward passes in backpropagation. **Part c (Level 3 - Apply):** Apply backpropagation to a simple 2-layer network. **Part d**

(Level 4 - Analyze): Compare different differentiation algorithms. **Part e (Level 5 - Evaluate):** Evaluate computational complexity of backpropagation.

19. **(Bloom's Levels 1-5)** Analyze automatic differentiation in deep learning frameworks.

    Part a (Level 1 - Remember): Define computational graphs in deep learning. **Part b (Level 2 - Understand):** Explain how automatic differentiation works. **Part c (Level 3 - Apply):** Trace the computational graph for a simple expression. **Part d (Level 4 - Analyze):** Compare forward-mode and reverse-mode automatic differentiation. **Part e (Level 5 - Evaluate):** Evaluate the efficiency of automatic differentiation in TensorFlow/PyTorch.

20. **(Bloom's Levels 1-5)** Demonstrate practical implementation of feedforward net-works.

    Part a (Level 1 - Remember): List the steps to implement a neural network. **Part b (Level 2 - Understand):** Explain the importance of data preprocessing. **Part c (Level 3 - Apply):** Implement forward pass for a 3-layer network. **Part d (Level 4 - Analyze):** Compare different weight initialization strategies. **Part e (Level 5 - Evaluate):** Evaluate the impact of batch size on training dynamics.

6.8 UNIT-III: Regularization and Optimization of DL Models (10 Questions)

21. **(Bloom's Levels 1-5)** Analyze regularization techniques for preventing overfitting.

    Part a (Level 1 - Remember): Define regularization and list common techniques. **Part b (Level 2 - Understand):** Explain how L2 regularization modifies the loss function. **Part c (Level 3 - Apply):** Apply L1 and L2 regularization to linear regression. **Part d (Level 4 - Analyze):** Compare dropout with bagging as ensemble methods. **Part e (Level 5 - Evaluate):** Evaluate regularization methods for a deep CNN.

22. **(Bloom's Levels 1-5)** Demonstrate parameter norm penalties and constrained opti-mization.

    Part a (Level 1 - Remember): Define parameter norm penalties with mathemat-ical notation. **Part b (Level 2 - Understand):** Explain the equivalence between norm penalties and constrained optimization. **Part c (Level 3 - Apply):** Solve a constrained optimization problem using Lagrange multipliers. **Part d (Level 4 - An-alyze):** Compare L1 and L2 regularization from an optimization perspective. **Part e (Level 5 - Evaluate):** Evaluate the choice of regularization strength for different problems.

23. **(Bloom's Levels 1-5)** Analyze dataset augmentation and noise robustness.

    Part a (Level 1 - Remember): List common data augmentation techniques for images. **Part b (Level 2 - Understand):** Explain how data augmentation reduces

overfitting. **Part c (Level 3 - Apply):** Apply data augmentation to a small image dataset. **Part d (Level 4 - Analyze):** Compare different types of noise injection for robustness. **Part e (Level 5 - Evaluate):** Evaluate the impact of augmentation on model generalization.

24. **(Bloom's Levels 1-5)** Demonstrate semi-supervised and multi-task learning.

    Part a (Level 1 - Remember): Define semi-supervised learning with examples. **Part b (Level 2 - Understand):** Explain the concept of multi-task learning. **Part c (Level 3 - Apply):** Implement pseudo-labeling for semi-supervised learning. **Part d (Level 4 - Analyze):** Compare hard and soft parameter sharing in multi-task learning. **Part e (Level 5 - Evaluate):** Evaluate the benefits of multi-task learning for related tasks.

25. **(Bloom's Levels 1-5)** Analyze early stopping and parameter sharing.

    Part a (Level 1 - Remember): Define early stopping in neural network training. **Part b (Level 2 - Understand):** Explain how early stopping acts as regularization. **Part c (Level 3 - Apply):** Implement early stopping with patience. **Part d (Level 4 - Analyze):** Compare parameter tying and parameter sharing. **Part e (Level 5 - Evaluate):** Evaluate the effectiveness of early stopping vs L2 regularization.

26. **(Bloom's Levels 1-5)** Demonstrate sparse representations and ensemble methods.

    Part a (Level 1 - Remember): Define sparse representations in neural networks. **Part b (Level 2 - Understand):** Explain the benefits of sparsity for interpretability. **Part c (Level 3 - Apply):** Implement sparse autoencoder with L1 penalty. **Part d (Level 4 - Analyze):** Compare bagging, boosting, and stacking ensemble methods. **Part e (Level 5 - Evaluate):** Evaluate the trade-offs of ensemble methods vs single models.

27. **(Bloom's Levels 1-5)** Analyze dropout and adversarial training.

    Part a (Level 1 - Remember): Define dropout in neural networks. **Part b (Level 2 - Understand):** Explain the relationship between dropout and ensemble methods. **Part c (Level 3 - Apply):** Apply dropout to a simple network during training and inference. **Part d (Level 4 - Analyze):** Compare dropout with DropConnect and Spatial Dropout. **Part e (Level 5 - Evaluate):** Evaluate the effectiveness of adversarial training for robustness.

28. **(Bloom's Levels 1-5)** Demonstrate tangent methods for manifold learning.

    Part a (Level 1 - Remember): Define tangent distance and tangent propagation. **Part b (Level 2 - Understand):** Explain the manifold hypothesis in deep learning. **Part c (Level 3 - Apply):** Compute tangent vectors for image transformations. **Part d (Level 4 - Analyze):** Compare Tangent Prop with data augmentation. **Part e (Level 5 - Evaluate):** Evaluate the Manifold Tangent Classifier for few-shot learning.

29. **(Bloom's Levels 1-5)** Analyze optimization algorithms for training deep networks.

Part a (Level 1 - Remember): List the challenges in neural network optimization. **Part b (Level 2 - Understand):** Explain momentum and its benefits over basic SGD. **Part c (Level 3 - Apply):** Apply Adam optimizer update for one step. **Part d (Level 4 - Analyze):** Compare different parameter initialization strategies. **Part e (Level 5 - Evaluate):** Evaluate second-order vs first-order optimization methods.

30. **(Bloom's Levels 1-5)** Demonstrate optimization strategies and meta-algorithms.

    Part a (Level 1 - Remember): Define learning rate schedules in deep learning. **Part b (Level 2 - Understand):** Explain the concept of gradient clipping. **Part c (Level 3 - Apply):** Implement a cosine annealing learning rate schedule. **Part d (Level 4 - Analyze):** Compare different learning rate decay strategies. **Part e (Level 5 - Evaluate):** Evaluate the effectiveness of warmup in transformer training.

6.9 UNIT-IV: Convolutional Networks (10 Questions)

31. **(Bloom's Levels 1-5)** Analyze the convolution operation and its properties.

    Part a (Level 1 - Remember): Define the convolution operation for 2D images. **Part b (Level 2 - Understand):** Explain sparse interactions and parameter sharing in CNNs. **Part c (Level 3 - Apply):** Apply convolution to a 4×4 input with 3×3 kernel. **Part d (Level 4 - Analyze):** Compare convolution with cross-correlation. **Part e (Level 5 - Evaluate):** Evaluate the importance of translation equivariance in CNNs.

32. **(Bloom's Levels 1-5)** Demonstrate pooling operations in convolutional networks.

    Part a (Level 1 - Remember): Define max pooling and average pooling. **Part b (Level 2 - Understand):** Explain the benefits of pooling for translation invariance. **Part c (Level 3 - Apply):** Apply max pooling to a 4×4 feature map. **Part d (Level 4 - Analyze):** Compare different pooling operations and their effects. **Part e (Level 5 - Evaluate):** Evaluate the trade-off between pooling and strided convolutions.

33. **(Bloom's Levels 1-5)** Analyze basic convolution functions and parameters.

    Part a (Level 1 - Remember): List convolution layer hyperparameters. **Part b (Level 2 - Understand):** Explain the relationship between stride, padding, and output size. **Part c (Level 3 - Apply):** Calculate output size for given convolution parameters. **Part d (Level 4 - Analyze):** Compare different padding strategies (valid, same, full). **Part e (Level 5 - Evaluate):** Evaluate the impact of dilation on receptive field.

34. **(Bloom's Levels 1-5)** Demonstrate structured outputs from convolutional networks.

    Part a (Level 1 - Remember): List different types of structured outputs from CNNs. **Part b (Level 2 - Understand):** Explain the difference between object detection and segmentation. **Part c (Level 3 - Apply):** Design a CNN for image classification with structured output. **Part d (Level 4 - Analyze):**

Compare different outputheads for multi-task learning. **Part e (Level 5 - Evaluate):** Evaluate the design of YOLO for real-time object detection.

35. **(Bloom's Levels 1-5)** Analyze data types and representations in CNNs.

    Part a (Level 1 - Remember): List common input data types for CNNs. **Part b (Level 2 - Understand):** Explain how different data types affect network design. **Part c (Level 3 - Apply):** Determine tensor shapes through a CNN architecture. **Part d (Level 4 - Analyze):** Compare 2D, 3D, and 1D convolutions for different data types. **Part e (Level 5 - Evaluate):** Evaluate the design choices for video understanding networks.

36. **(Bloom's Levels 1-5)** Demonstrate efficient convolution algorithms.

    Part a (Level 1 - Remember): Define the im2col algorithm for convolution. **Part b (Level 2 - Understand):** Explain how FFT can accelerate convolution. **Part c (Level 3 - Apply):** Compare computational complexity of direct vs FFT convolu-tion. **Part d (Level 4 - Analyze):** Compare Winograd convolution with standard approaches. **Part e (Level 5 - Evaluate):** Evaluate the memory vs speed trade-off in convolution algorithms.

37. **(Bloom's Levels 1-5)** Analyze random or unsupervised features in CNNs.

    Part a (Level 1 - Remember): Define random features in convolutional networks. **Part b (Level 2 - Understand):** Explain the concept of unsupervised pre-training. **Part c (Level 3 - Apply):** Implement a convolutional autoencoder. **Part d (Level 4 - Analyze):** Compare random features with learned features. **Part e (Level 5 - Evaluate):** Evaluate the effectiveness of contrastive learning for visual representations.

38. **(Bloom's Levels 1-5)** Demonstrate theoretical basis for convolutional networks.

    Part a (Level 1 - Remember): State the theoretical foundations of CNNs. **Part b (Level 2 - Understand):** Explain the universal approximation theorem for CNNs. **Part c (Level 3 - Apply)** Verify translation equivariance for a simple CNN. **Part d (Level 4 - Analyze):** Compare the inductive biases of CNNs and fully connected networks. **Part e (Level 5 - Evaluate):** Evaluate the theoretical guarantees of CNNs for image tasks.

39. **(Bloom's Levels 1-5)** Analyze classic CNN architectures.

    Part a (Level 1 - Remember): List key CNN architectures from LeNet to ResNet. **Part b (Level 2 - Understand):** Explain the key innovations in AlexNet. **Part c (Level 3 - Apply):** Calculate the receptive field in a VGG network. **Part d (Level 4 - Analyze):** Compare VGG, GoogLeNet, and ResNet architectures. **Part e (Level 5 - Evaluate):** Evaluate the impact of skip connections in ResNet.

40. **(Bloom's Levels 1-5)** Demonstrate practical CNN design and implementation.

    Part a (Level 1 - Remember): List common CNN design patterns. **Part b (Level 2 - Understand):** Explain the bottleneck design in modern CNNs. **Part c (Level 3 Apply):** Design a CNN for CIFAR-10 classification. **Part d (Level 4 - Analyze):**

Compare depthwise separable convolutions with standard convolutions. **Part e (Level 5 - Evaluate):** Evaluate the efficiency of MobileNet vs ResNet for mobile deployment.

6.10 UNIT-V: Sequence Modeling (10 Questions)

41. **(Bloom's Levels 1-5)** Analyze recurrent neural networks and unfolding computa-tional graphs.

    Part a (Level 1 - Remember): Define RNN and write the forward pass equations. **Part b (Level 2 - Understand):** Explain unfolding computational graphs. **Part c (Level 3 - Apply):** Unfold an RNN for 3 time steps and compute forward pass. **Part d (Level 4 - Analyze):** Compare RNNs with feedforward networks for sequence data. **Part e (Level 5 - Evaluate):** Evaluate the limitations of simple RNNs for long sequences.

42. **(Bloom's Levels 1-5)** Demonstrate backpropagation through time (BPTT).

    Part a (Level 1 - Remember): Define backpropagation through time. **Part b (Level 2 - Understand):** Explain the gradient flow in BPTT. **Part c (Level 3 - Apply):** Compute gradients for an RNN using BPTT for 2 time steps. **Part d (Level 4 - Analyze):** Compare BPTT with truncated BPTT. **Part e (Level 5 - Evaluate):** Evaluate the computational complexity of BPTT.

43. **(Bloom's Levels 1-5)** Analyze bidirectional RNNs.

    Part a (Level 1 - Remember): Define bidirectional RNN architecture. **Part b (Level 2 - Understand):** Explain how bidirectional RNNs capture future context. **Part c (Level 3 - Apply):** Compute forward and backward hidden states for a sequence. **Part d (Level 4 - Analyze):** Compare unidirectional and bidirectional RNNs for different tasks. **Part e (Level 5 - Evaluate):** Evaluate the suitability of bidirectional RNNs for real-time applications.

44. **(Bloom's Levels 1-5)** Demonstrate encoder-decoder sequence-to-sequence architec-tures.

    Part a (Level 1 - Remember): Define encoder-decoder architecture for seq2seq. **Part b (Level 2 - Understand):** Explain how the context vector is computed. **Part c (Level 3 - Apply):** Implement encoder and decoder for machine translation. **Part d (Level 4 - Analyze):** Compare different ways to initialize decoder hidden state. **Part e (Level 5 - Evaluate):** Evaluate the limitations of fixed-length context vectors.

45. **(Bloom's Levels 1-5)** Analyze attention mechanisms in seq2seq models.

    Part a (Level 1 - Remember): Define attention mechanism

in neural networks. **Part b (Level 2 - Understand):** Explain how attention weights are computed. **Part c (Level 3 - Apply):** Compute attention scores for a simple example. **Part d (Level 4 - Analyze):** Compare additive, multiplicative, and dot-product attention. **Part e (Level 5 - Evaluate):** Evaluate the impact of attention on translation quality.

46. **(Bloom's Levels 1-5)** Demonstrate deep recurrent networks.

    Part a (Level 1 - Remember): Define deep RNN architectures. **Part b (Level 2 - Understand):** Explain the benefits of depth in recurrent networks. **Part c (Level 3 - Apply):** Implement a 3-layer stacked RNN. **Part d (Level 4 - Analyze):** Compare deep RNNs with residual connections. **Part e (Level 5 - Evaluate):** Evaluate the trade-offs of depth in recurrent networks.

47. **(Bloom's Levels 1-5)** Analyze LSTM and gated RNN architectures.

    Part a (Level 1 - Remember): Write the LSTM forward pass equations. **Part b (Level 2 - Understand):** Explain the role of each gate in LSTM. **Part c (Level 3 - Apply):** Compute LSTM cell update for one time step. **Part d (Level 4 - Analyze):** Compare LSTM and GRU architectures. **Part e (Level 5 - Evaluate):** Evaluate the effectiveness of gating mechanisms for long-term dependencies.

48. **(Bloom's Levels 1-5)** Demonstrate optimization for long-term dependencies.

    Part a (Level 1 - Remember): Define the vanishing gradient problem in RNNs.
    Part b (Level 2 - Understand): Explain gradient clipping and its benefits. **Part c (Level 3 - Apply):** Apply gradient clipping to prevent exploding gradients. **Part d (Level 4 - Analyze):** Compare different strategies for handling long-term depen-dencies. **Part e (Level 5 - Evaluate):** Evaluate the effectiveness of LSTMs vs Transformers for long sequences.

49. **(Bloom's Levels 1-5)** Analyze autoencoders for sequence data.

    Part a (Level 1 - Remember): Define autoencoder and its components. **Part b (Level 2 - Understand):** Explain denoising and sparse autoencoders. **Part c (Level 3 - Apply):** Implement a sequence autoencoder for text. **Part d (Level 4 - Analyze):** Compare variational autoencoders with standard autoencoders. **Part e (Level 5 - Evaluate):** Evaluate the use of autoencoders for anomaly detection in time series.

50. **(Bloom's Levels 1-5)** Demonstrate deep generative models for sequences.

    Part a (Level 1 - Remember): List deep generative models for sequences. **Part b (Level 2 - Understand):** Explain how

GANs generate sequences. **Part c (Level 3 - Apply):** Implement a simple generative model for text. **Part d (Level 4 - Analyze):** Compare VAEs, GANs, and flow-based models for sequence generation. **Part e (Level 5 - Evaluate):** Evaluate the challenges in evaluating generative models for sequences.

Summary of Questions by Unit and Bloom's Level

Overall Statistics:

- **Total Questions:** 50 (10 per unit)
- **Total Parts:** 250 (5 parts per question)

Unit	Level 1	Level 2	Level 3	Level 4	Level 5
Unit I	10	10	10	10	10
Unit II	10	10	10	10	10
Unit III	10	10	10	10	10
Unit IV	10	10	10	10	10
Unit V	10	10	10	10	10
Total	**50**	**50**	**50**	**50**	**50**

Table 6.1: Distribution of questions across Bloom's Taxonomy levels (50 parts per level, 250 total parts)

- **Course Educational Objectives Covered:** All 5

C)Case study-based questions (if applicable)

Case Study 1: Deep Learning for Medical Image Diagnosis
Question
A hospital wants to develop an AI system that automatically detects lung diseases such as **Pneumonia, Tuberculosis, COVID-19, and Lung Cancer** from chest X-ray images. Explain how a Deep Learning model can be developed for this application.
Answer
Problem Statement
Radiologists manually analyze thousands of chest X-rays every day. Human diagnosis may be slow and prone to errors due to fatigue. Deep Learning can automate disease detection with high accuracy.

Dataset

- Chest X-ray images
- Labels:
  - Normal
  - Pneumonia
  - Tuberculosis
  - COVID-19
  - Lung Cancer

Dataset Size:

- Training Images: 20,000
- Validation Images: 4,000
- Test Images: 6,000

Deep Learning Architecture
Input Image
↓
Image Preprocessing
↓
CNN Layers
↓
Pooling Layers
↓

CBAM Attention Module
↓
Fully Connected Layer
↓
Softmax Classifier
↓
Disease Prediction

Steps

Step 1: Image Collection

Collect chest X-ray images from hospitals.

Step 2: Image Preprocessing

- Resize images (224×224)
- Normalize pixel values
- Remove noise
- Data augmentation

Step 3: CNN Feature Extraction

CNN automatically extracts

- Edges
- Textures
- Nodules
- Infection regions

Step 4: Attention Mechanism

CBAM highlights infected lung regions while suppressing irrelevant background.

Step 5: Classification

Softmax predicts one of

- Normal
- Pneumonia
- TB
- COVID-19
- Lung Cancer

Evaluation Metrics

- Accuracy
- Precision
- Recall
- F1-score

- ROC-AUC
- Confusion Matrix

Advantages

- Faster diagnosis
- Reduced workload for doctors
- High accuracy
- Early disease detection
- Supports clinical decision-making

Limitations

- Requires a large labeled dataset
- High computational cost
- May overfit small datasets
- Explainability remains challenging

Conclusion

Deep Learning significantly improves medical image diagnosis by automatically learning disease-specific features and assisting radiologists in making accurate decisions.

Case Study 2: Self-Driving Cars

Question

Explain how Deep Learning enables autonomous vehicles to drive safely.

Answer

Problem

A company wants to build a self-driving car capable of

- Detecting roads
- Identifying pedestrians
- Reading traffic signs
- Avoiding obstacles

Deep Learning Models Used

- CNN → Object Detection
- YOLO → Real-time Detection
- RNN/LSTM → Vehicle Motion Prediction

- Reinforcement Learning → Driving Decisions

Workflow

Camera
↓
Image Capture
↓
CNN Feature Extraction
↓
Object Detection
↓
Decision Making
↓
Brake / Accelerate / Turn

Input Data

- Camera images
- Radar
- LiDAR
- GPS

Output

- Steering angle
- Vehicle speed
- Brake control

Advantages

- Reduced accidents
- Faster response
- Continuous driving
- Traffic optimization

Challenges

- Bad weather
- Poor lighting
- Rare traffic situations
- Ethical decision-making

Case Study 3: Face Recognition Attendance System

Question
Explain how Deep Learning can automate attendance using face recognition.

Answer

Problem
Manual attendance is time-consuming and susceptible to proxy attendance.

Solution
Develop a face recognition system using CNN.

Steps
1. Capture student image
2. Detect face
3. Extract facial features
4. Compare with database
5. Mark attendance automatically

Deep Learning Model
Input Face
↓
CNN
↓
Feature Vector
↓
Face Matching
↓
Attendance Database

Advantages
- No proxy attendance
- Fast attendance
- Contactless
- Accurate

Limitations
- Lighting variations
- Masked faces
- Similar-looking individuals

- Privacy concerns

Case Study 4: Deep Learning for Crop Disease Detection

Question

A farmer wants to identify plant diseases using smartphone images. Explain the Deep Learning solution.

Answer

Dataset

Images of

- Healthy leaves
- Diseased leaves

Diseases

- Rust
- Blight
- Mildew

Model

CNN
↓
Image Classification
↓
Disease Detection
↓
Treatment Recommendation

Benefits

- Early disease detection
- Reduced pesticide usage
- Higher crop yield
- Lower farming costs

Challenges

- Poor image quality
- Different lighting conditions
- Multiple diseases on one leaf

Case Study 5: Deep Learning for Bank Fraud Detection

Question

Explain how Deep Learning can identify fraudulent financial transactions.

Answer

Problem

Banks process millions of transactions daily. Manual fraud detection is impractical.

Input Features

- Transaction amount
- Time
- Location
- Device
- Merchant
- Customer history

Model

Transaction Data
↓
Data Preprocessing
↓
Deep Neural Network
↓
Fraud Probability
↓
Alert

Benefits

- Real-time detection
- Reduced financial loss
- Continuous learning
- Improved customer security

Limitations

- Class imbalance
- False positives
- Data privacy concerns

Case Study 6: Deep Learning for Speech Recognition

Question

Explain the Deep Learning pipeline used in voice assistants such as Siri or Google Assistant.

Answer

Workflow

Speech Input
↓
Noise Removal
↓
Feature Extraction (MFCC)
↓
LSTM/RNN Model
↓
Text Generation
↓
Intent Recognition
↓
Response

Applications

- Voice assistants
- Smart homes
- Customer support
- Dictation software

Challenges

- Background noise
- Accent variations
- Multiple languages
- Low-quality microphones

Case Study 7: Deep Learning for E-Commerce Recommendation

Question
How does an e-commerce company use Deep Learning to recommend products to customers?

Answer
Problem
Customers expect personalized product recommendations based on their interests.

Input Data
- Purchase history
- Search history
- Product ratings
- Browsing behavior

Deep Learning Model
User Behavior
↓
Embedding Layer
↓
Deep Neural Network
↓
Recommendation Engine
↓
Personalized Product Suggestions

Benefits
- Improved user experience
- Increased sales
- Better customer retention
- Personalized recommendations

Challenges
- Cold-start problem
- Data sparsity
- Privacy concerns

Important Examination Points

For case study questions in university exams, include the following sections to score full marks:

1. Introduction
2. Problem Statement
3. Dataset/Input Data
4. Deep Learning Architecture/Model
5. Workflow or Block Diagram
6. Training Process
7. Evaluation Metrics
8. Advantages
9. Limitations
10. Applications
11. Conclusion

UNIT-I: LINEAR ALGEBRA, PROBABILITY AND NUMERICAL COMPUTATION

1. Design a dimensionality reduction system using PCA for image classification and justify each step.
2. Develop a Bayesian classification model for medical diagnosis and explain how conditional probability improves prediction accuracy.
3. Compare Eigen Decomposition and SVD, and recommend the most suitable technique for high-dimensional datasets with justification.
4. Design a gradient-based optimization process for training a neural network while addressing overflow, underflow, and convergence issues.
5. Create a machine learning workflow integrating linear algebra, probability theory, and numerical computation for predictive modeling.
6. Evaluate different optimization techniques and recommend the most appropriate method for training deep learning models under computational constraints.
7. Design a feature extraction and classification pipeline using

tensors, PCA, and matrix operations for an image recognition system.

8. Propose an efficient strategy for handling high-dimensional data in deep learning by integrating SVD, PCA, and gradient-based optimization, and justify your design choices.

UNIT-II: MACHINE LEARNING AND DEEP FEED FORWARD NETWORKS

1. Design a machine learning model for disease prediction, explaining the role of hyperparameters, validation sets, and estimators.
2. Develop a deep feedforward neural network to solve the XOR problem and explain why hidden layers are necessary.
3. Compare Maximum Likelihood Estimation and Bayesian Statistics, and recommend the most suitable approach for uncertain datasets with justification.
4. Design a neural network architecture for image classification, highlighting the roles of hidden units, activation functions, and backpropagation.
5. Propose a strategy to minimize bias, variance, underfitting, and overfitting in a deep learning model.
6. Design an optimized training workflow using Stochastic Gradient Descent, validation sets, and backpropagation for a deep neural network.
7. Create a machine learning system that integrates supervised learning, Bayesian inference, and deep feedforward networks for fraud detection.
8. Design a complete deep learning solution for handwritten digit recognition, including architecture design, gradient-based learning, and performance evaluation.

NIT-III: REGULARIZATION AND OPTIMIZATION OF DL MODELS

1. Design a regularization framework for a deep neural network used in medical image classification, incorporating L2 regularization, dropout, data augmentation, and early stopping.
2. Develop a robust deep learning model for handwritten digit recognition using parameter sharing, sparse representations, and ensemble learning techniques.
3. Compare bagging, dropout, and adversarial training, and recommend the most appropriate technique for improving model robustness in image classification.
4. Design an optimization strategy for training a deep convolutional neural network, explaining the roles of parameter initialization, adaptive learning rates, and optimization algorithms.
5. Propose an integrated training framework that combines regularization and optimization techniques to minimize overfitting and maximize model accuracy.
6. Design a deep learning solution for autonomous driving that addresses adversarial attacks, parameter optimization, and computational efficiency.
7. Develop a complete workflow for training a deep neural network using dataset augmentation, early

stopping, Adam optimization, and ensemble methods.

8. Create a scalable optimization and regularization strategy for a large-scale healthcare prediction system, justifying the selection of each technique based on model performance and computational requirements.

UNIT-IV: CONVOLUTIONAL NETWORKS

1. Design a CNN architecture for handwritten digit recognition, explaining the roles of convolution, pooling, activation functions, and fully connected layers.
2. Develop a convolutional neural network for medical image classification, justifying the choice of kernel size, pooling method, and structured output representation.
3. Compare max pooling, average pooling, and strided convolution, and recommend the most suitable technique for real-time image recognition.
4. Design an optimized convolution framework for autonomous driving that minimizes computational cost while maintaining high prediction accuracy.

5. Propose a feature extraction pipeline using convolution operations and unsupervised learning for large-scale image datasets.
6. Design a CNN-based object detection system that efficiently processes multiple image data types and produces structured outputs.
7. Develop a lightweight convolutional network for deployment on embedded or mobile devices using efficient convolution algorithms.
8. Create a complete deep learning solution for image classification that integrates convolution operations, pooling, structured outputs, efficient convolution algorithms, and feature learning techniques.

UNIT-V: SEQUENCE MODELING

1. Design a recurrent neural network for sentiment analysis, explaining the roles of hidden states, unfolding computational graphs, and backpropagation through time.

---

2. Develop an encoder-decoder sequence-to-sequence architecture for machine

translation and justify the selection of Bidirectional LSTM over standard RNN.

3. Compare LSTM, GRU, and Bidirectional RNN architectures, and recommend the most suitable model for long-term dependency learning with justification.
4. Design a deep recurrent neural network for speech recognition, incorporating optimization techniques to address vanishing gradients.
5. Propose an autoencoder-based framework for anomaly detection in network security and explain how latent representations improve performance.
6. Design a deep generative model for synthetic medical image generation and justify its advantages in data augmentation.
7. Develop an intelligent chatbot using sequence-to-sequence learning, LSTM, and encoder-decoder architecture, explaining each design component.
8. Create a complete sequence modeling framework for healthcare prediction that integrates recurrent neural networks,

LSTM, autoencoders, deep generative models, and optimization techniques for long-term dependencies.

6. References

a. Standard Reference Books

1. **Ian Goodfellow, Yoshua Bengio, and Aaron Courville**
   - **Title:** *Deep Learning*
   - **Publisher:** MIT Press
   - **Year:** 2016
   - **ISBN:** 978-0262035613
   - **Coverage:** Complete coverage of all five units including Linear Algebra, Probability, Machine Learning, Feedforward Networks, Regularization, Optimization, CNNs, RNNs, LSTM, Autoencoders, and Deep Generative Models.
   - **Recommended for:** Primary textbook.
2. **Christopher M. Bishop**
   - **Title:** *Pattern Recognition and Machine Learning*
   - **Publisher:** Springer
   - **Year:** 2006
   - **ISBN:** 978-0387310732
   - **Coverage:** Probability Theory, Bayesian Learning, Machine Learning fundamentals, PCA, Dimensionality Reduction, Optimization, Neural Networks.
3. **Kevin P. Murphy**
   - **Title:** *Machine Learning: A Probabilistic Perspective*
   - **Publisher:** MIT Press
   - **Year:** 2012
   - **ISBN:** 978-0262018029
   - **Coverage:** Probability, Statistics, Bayesian Learning, Optimization, Graphical Models, Machine Learning concepts.

4. **Aurélien Géron**
    - **Title:** *Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow* (3rd Edition)
    - **Publisher:** O'Reilly Media
    - **Year:** 2022
    - **ISBN:** 978-1098125974
    - **Coverage:** Practical implementation of Deep Learning, CNNs, RNNs, LSTM, Autoencoders, Transfer Learning, TensorFlow and Keras.
5. **François Chollet**
    - **Title:** *Deep Learning with Python* (2nd Edition)
    - **Publisher:** Manning Publications
    - **Year:** 2021
    - **ISBN:** 978-1617296864
    - **Coverage:** Feedforward Networks, CNNs, RNNs, LSTM, Sequence Modeling, Deep Learning implementation using Keras.
6. **Richard O. Duda, Peter E. Hart, and David G. Stork**
    - **Title:** *Pattern Classification* (2nd Edition)
    - **Publisher:** Wiley
    - **Year:** 2000
    - **ISBN:** 978-0471056690
    - **Coverage:** Classification, Bayesian Decision Theory, Feature Extraction, Neural Networks.
7. **Simon Haykin**
    - **Title:** *Neural Networks and Learning Machines* (3rd Edition)
    - **Publisher:** Pearson
    - **Year:** 2009
    - **ISBN:** 978-0131471399
    - **Coverage:** Artificial Neural Networks, Learning Algorithms, Optimization, RNNs, Deep Learning fundamentals.
8. **Charu C. Aggarwal**
    - **Title:** *Neural Networks and Deep Learning*
    - **Publisher:** Springer
    - **Year:** 2018

- **ISBN:** 978-3319944623
- **Coverage:** Deep Learning architectures, CNNs, RNNs, LSTM, Optimization, Regularization.

9. **Trevor Hastie, Robert Tibshirani, and Jerome Friedman**
   - **Title:** *The Elements of Statistical Learning* (2nd Edition)
   - **Publisher:** Springer
   - **Year:** 2009
   - **Coverage:** Machine Learning theory, Statistical Learning, Regularization, Ensemble Learning.
10. **David Barber**
    - **Title:** *Bayesian Reasoning and Machine Learning*
    - **Publisher:** Cambridge University Press
    - **Year:** 2012
    - **Coverage:** Bayesian Statistics, Probability, Machine Learning, Optimization.

b. Web Resources

1. https://keras.io/datasets/
2. http://deeplearning.net/tutorial/deeplearning.pdf
3. https://arxiv.org/pdf/1404.7828v4.pdf
4. https://www.cse.iitm.ac.in/~miteshk/CS7015.html
5. https://www.deeplearningbook.org
6. https://nptel.ac.in/courses/106105215

7. Additional Enrichment Components

1. Hands-on Laboratory Sessions Implement basic neural networks using Python.

Build Deep Neural Networks (DNNs) using TensorFlow and Keras.

Develop Convolutional Neural Networks (CNNs) for image classification.

Implement Recurrent Neural Networks (RNNs) and LSTMs for sequence prediction.

Perform hyperparameter tuning and model evaluation.
Train and test models using Google Colab.

2. Mini Projects Handwritten Digit Recognition (MNIST)

Face Mask Detection
Plant Disease Detection
Sentiment Analysis on Social Media
Fake News Detection
Image Classification using CNN
Speech Emotion Recognition
Stock Price Prediction using LSTM
Medical Image Classification
Traffic Sign Recognition

3. Case Studies Autonomous Vehicles using Deep Learning

Healthcare Diagnosis using CNN
Chatbots and Virtual Assistants
Deep Learning in Agriculture
Financial Fraud Detection
Facial Recognition Systems
Recommendation Systems (Netflix, Amazon)
Deep Learning in Cybersecurity

4. Industry-Oriented Tools Python

Jupyter Notebook
Google Colab
TensorFlow
Keras
PyTorch
OpenCV
Scikit-learn
NumPy
Pandas
Matplotlib

Hugging Face Transformers (Introduction)

5. Online Learning Resources Coursera – Deep Learning Specialization

edX – Artificial Intelligence Courses
NPTEL – Deep Learning
SWAYAM Courses
Kaggle Learn
TensorFlow Tutorials
PyTorch Tutorials
Google AI Learning Resources

6. Research Paper Reading Students should review recent research papers on:

Convolutional Neural Networks
Vision Transformers (ViT)
Generative AI
Large Language Models (LLMs)
Attention Mechanisms
Medical Image Analysis
Explainable Artificial Intelligence (XAI)

7. Seminar Topics Evolution of Deep Learning

Explainable AI (XAI)
Deep Reinforcement Learning
Generative Adversarial Networks (GANs)
Vision Transformers (ViT)
Diffusion Models
Large Language Models (LLMs)
Federated Learning
TinyML
AI Ethics and Responsible AI

8. Industrial Applications Healthcare

Banking and Finance
Agriculture
Manufacturing

Autonomous Vehicles
Retail and E-commerce
Smart Cities
Cybersecurity
Robotics
Natural Language Processing

9. Certification Courses Google TensorFlow Developer Certificate

IBM AI Engineering Professional Certificate
DeepLearning.AI Specialization
NVIDIA Deep Learning Institute (DLI)
Microsoft AI Engineer Learning Paths

10. Hackathons and Competitions Kaggle Competitions

Smart India Hackathon (SIH)
AICTE IDEA Lab Challenges
IEEE Student Competitions
ACM Programming Contests
OpenCV AI Competitions

11. Internship Opportunities AI Research Labs

Healthcare AI Companies
IT and Software Companies
Robotics Startups
Data Science Organizations
EdTech Companies
FinTech Companies

12. Emerging Trends Generative AI

Foundation Models
Large Language Models (LLMs)
Vision Transformers (ViTs)
Multimodal AI
Explainable AI (XAI)
Edge AI
TinyML

Federated Learning
AI for Scientific Discovery
AI Agents and Agentic AI

13. Value-Added Activities Coding Challenges (LeetCode, HackerRank)

Weekly Model-Building Exercises
Research Paper Presentations
Peer Learning Sessions
AI Club Activities
Guest Lectures by Industry Experts
Industrial Visits
Technical Workshops
Poster and Project Exhibitions

14. Outcome-Based Learning Activities Develop deep learning models for real-world datasets.

Analyze and compare model performance using evaluation metrics.
Optimize models using regularization and hyperparameter tuning.
Apply CNNs to image processing applications.
Implement RNNs/LSTMs for sequence modeling tasks.
Interpret model predictions using Explainable AI techniques.
Deploy trained models using TensorFlow Lite or cloud platforms.