Artificial Neural Networks: From Theory to Practice

A Comprehensive Textbook for Computer Science Students

October 5, 2025

Contents

Chapter 1

Introduction to Machine Learning

1.1 What is Learning?

Learning is the process of acquiring new knowledge, skills, or behaviors through experience. This process transforms inputs—such as data, experiences, or information—into useful capabilities like expertise, new skills, or predictive models.

1.1.1 Core Elements of Learning

Every learning process involves these fundamental components:

1. **Input**: Data, experiences, or information that enters the learning system
2. **Processing**: The manipulation or transformation of that data according to learning rules
3. **Output**: The result—new knowledge, skills, or predictive capabilities
4. **Feedback**: Information about the effectiveness of learning outcomes
5. **Memory**: The ability to retain and access learned information

1.1.2 Key Questions in Learning

- What are the essential inputs for the learning process?
- How do we measure the effectiveness and success of learning?
- What are the underlying mechanisms and processes by which learning occurs?
- How can we generalize from specific experiences to handle new situations?

1.2 What is Reasoning?

Reasoning is the ability to draw logical conclusions from known facts or learned knowledge. Unlike learning, reasoning relies on logical inference rather than large amounts of data.

1.3 Types of Reasoning: Comprehensive Overview

Understanding different types of reasoning is crucial for designing effective learning systems. Each type has distinct characteristics and applications in both biological and artificial intelligence systems.

1.3.1 Inductive Reasoning

Definition: Inductive reasoning extracts patterns from observed data to make predictions about future or unseen cases. This approach moves from specific observations to general conclusions, yielding probable rather than certain results.

Key Characteristics:

- Most prevalent form of reasoning in the animal kingdom and primary mode in machine learning
- Forms the basis of most learned behaviors in animals
- Used extensively in deep learning and LLMs
- Enables generalization from limited examples to broader patterns

1.3.2 Deductive Reasoning

Definition: Deductive reasoning moves from general rules and premises to reach specific, guaranteed conclusions. It starts with a general rule and a specific case to reach a logical conclusion.

Key Characteristics:

- Provides certainty when premises are true
- Animals generally lack this capability for abstract reasoning
- LLMs can only mimic this through pattern matching
- Forms the basis of formal logic and mathematical proof

1.3.3 Abductive Reasoning (Inference to Best Explanation)

Definition: Abductive reasoning starts with an observation and seeks to find the simplest and most likely explanation. It's the process of finding a hypothesis that, if true, would best explain the observation.

1.4 Animal Learning

1.4.1 Example: Bait Shyness in Rats

Rats demonstrate a fundamental learning principle through their feeding behavior:

- They sample novel food cautiously
- If the food causes illness, they avoid it in the future
- Past experience directly informs future decisions

This natural learning process parallels challenges in machine learning.

1.5 Human Learning: The Cognitive Approach

Human learning is a complex and multifaceted process that has been studied extensively in psychology and neuroscience. It involves a combination of conscious and unconscious processes, leading to the acquisition of knowledge and skills that are both explicit (declarative) and implicit (procedural).

1.5.1 Cognitive Stages of Development

Jean Piaget's theory of cognitive development provides a classic framework for understanding how learning capabilities evolve from infancy to adulthood.

- **Sensorimotor Stage (0-2 years)**: Learning occurs through sensory experiences and motor interactions with the environment. Object permanence is a key milestone.
- **Preoperational Stage (2-7 years)**: Children begin to think symbolically and use words and pictures to represent objects. Their thinking is egocentric.

- **Concrete Operational Stage (7-11 years)**: Children begin to think logically about concrete events. They grasp concepts like conservation.
- **Formal Operational Stage (12+ years)**: Abstract reasoning and hypothetical thinking emerge.

This staged progression suggests that the ability to learn and the types of learning that are possible change fundamentally over a lifetime, a concept that has parallels in the development of more sophisticated machine learning models.

1.5.2 The Role of Memory

Memory is central to learning. The Atkinson-Shiffrin model is a classic theory that proposes three stages of memory:

1. **Sensory Memory:** A very brief buffer for sensory information.
2. **Short-Term Memory (Working Memory):** Holds a small amount of information for a short duration. It is where conscious thought and processing occur. This is analogous to the memory (RAM) of a computer.
3. **Long-Term Memory:** The vast, semi-permanent storage of knowledge and skills. This is analogous to a computer's hard drive.

The process of moving information from short-term to long-term memory, known as encoding, is critical for learning. Retrieval is the process of accessing this stored information. In machine learning, the "memory" is stored in the model's parameters (weights).

1.5.3 Learning Styles and Strategies

Humans employ a variety of strategies to learn, which can be broadly categorized:

- **Rote learning:** Memorization through repetition (e.g., flashcards). This is similar to overfitting in machine learning, where a model memorizes the training data.
- **Observational learning:** Learning by watching others.
- **Associative learning:** Connecting stimuli or events that occur together in the environment (classical and operant conditioning).
- **Cognitive learning:** Learning through understanding, reasoning, and problem-solving. This is the goal of more advanced AI systems.

1.6 What is Machine Learning?

Machine learning is a subfield of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. At its core, machine learning involves developing models that can identify patterns in data and make predictions or decisions based on those patterns.

1.6.1 Parallel: Spam Email Filtering

Consider how biological learning principles apply to spam detection:

- **Naive approach**: Memorize all past spam emails
- **Problem**: Cannot classify previously unseen emails
- **Solution**: Extract generalizable patterns (like words, phrases, or sender patterns)
- **Key insight**: Both rats and spam filters must generalize from specific experiences to handle new, similar situations

1.7 Types of Machine Learning

There are three main categories of machine learning algorithms:

1.7.1 Supervised Learning

In supervised learning, the algorithm is trained on a labeled dataset, meaning that each data point is tagged with a correct output. The goal is to learn a mapping function that can predict the output for new, unseen data. Common supervised learning tasks include classification and regression.

1.7.2 Unsupervised Learning

Unsupervised learning deals with unlabeled data. The algorithm tries to learn the underlying structure of the data without any explicit guidance. Common tasks include clustering, dimensionality reduction, and density estimation.

1.7.3 Reinforcement Learning

Reinforcement learning is concerned with how an agent ought to take actions in an environment so as to maximize some notion of cumulative reward. The algorithm learns by trial and error, receiving feedback in the form of rewards or punishments.

1.8 The Data and Observation Model

In machine learning, we typically represent our data as a matrix. Let's denote the dataset as $\mathcal{D}$. A common convention is to represent the data as a design matrix, X.

1.8.1 The Design Matrix

The design matrix X is an $m \times n$ matrix, where m is the number of training examples (or observations) and n is the number of features (or variables). Each row of the matrix represents a single data point, and each column represents a feature.

$$X = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{pmatrix}$$

Each row i corresponds to a data point $\mathbf{x}_i^T$, which is a row vector of size n.

1.9 Probabilistic Model of Learning

A powerful way to think about machine learning is from a probabilistic perspective. We can think of the learning process as finding a model that best explains the data. This is often framed as finding the parameters θ of a model that maximize the likelihood of observing the data.

The likelihood function is given by:

$$\mathcal{L}(\theta|X) = P(X|\theta)$$

The goal of learning is to find the parameters $\hat{\theta}$ that maximize this likelihood. This is known as Maximum Likelihood Estimation (MLE).

$$\hat{\theta}_{MLE} = \arg\max_{\theta} P(X|\theta)$$

1.10 Inductive Bias in Machine Learning

1.10.1 What is Inductive Bias?

Definition: Inductive bias refers to the set of assumptions that a learning algorithm makes to generalize from limited training data to unseen data.

Why is it Critical? Inductive bias is essential because:

- Machine learning models have limited training data
- Models must generalize from past observations to unseen cases
- Without appropriate bias, models may overfit (memorizing training data without learning generalizable patterns)
- All successful learning algorithms require appropriate assumptions about their domain

1.10.2 Types of Inductive Biases

Preference for Simpler Models (Occam's Razor)

- **Assumption**: Simpler explanations are preferred over complex ones
- **Example**: Decision trees with fewer splits are preferred because they generalize better
- **In Deep Learning**: Regularization techniques (L1, L2) penalize complex models

Smoothness Assumption

- **Assumption**: Data points that are close together should have similar outputs
- **Example**: In image classification, two similar images should belong to the same class
- **In ML**: K-Nearest Neighbors (KNN) assumes nearby data points have the same label

1.10.3 Inductive Bias in Specific Architectures

Convolutional Neural Networks (CNNs)

CNNs are designed for image processing and rely on key inductive biases:

1. Locality Bias (Local Connectivity)

- **Assumption**: Nearby pixels are more relevant than distant pixels
- **Example**: In facial recognition, CNN detects eyes, nose, mouth before recognizing entire face

2. Translation Invariance

- **Assumption**: An object should be recognized regardless of position
- **How it works**: CNNs use shared convolutional filters
- **Example**: Handwritten digit "3" recognized anywhere in the image

Recurrent Neural Networks (RNNs & LSTMs)

RNNs are designed for sequential data and rely on:

1. Temporal Dependency Bias

- **Assumption**: Recent information is more important than distant past
- **Example**: In "The cat sat on the mat", nearby words are more related

Transformers (BERT, GPT)

1. Attention-Based Bias (Self-Attention)

- **Assumption**: Important words can be anywhere in a sentence
- **Example**: In "The dog chased the ball...which was blue", "which" refers to "ball"

1.11 Mathematical Foundations of Learning

1.11.1 Learning as Optimization

Machine learning can be viewed as an optimization problem where we seek to find the best parameters θ that minimize a loss function $L(\theta)$:

$$\theta^* = \arg\min_{\theta} L(\theta)$$

Components of a Learning System

1. **Hypothesis Space** $\mathcal{H}$: The set of all possible functions the model can represent
2. **Loss Function** $L(\theta)$: Measures how well the model performs on the training data
3. **Optimization Algorithm**: Method to find θ^* (e.g., gradient descent)
4. **Regularization**: Techniques to prevent overfitting and improve generalization

The Bias-Variance Tradeoff

The expected prediction error can be decomposed as:

$$\text{Error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error}$$

where:

- **Bias**: Error due to simplifying assumptions in the model
- **Variance**: Error due to sensitivity to small fluctuations in training data
- **Irreducible Error**: Inherent noise in the data

1.11.2 PAC Learning Framework

Probably Approximately Correct (PAC) learning provides theoretical foundations for when learning is possible.

A concept class $\mathcal{C}$ is PAC-learnable if there exists an algorithm that, for any distribution $\mathcal{D}$ and any $\epsilon, \delta > 0$, can find a hypothesis h such that:

$$\Pr[\text{error}(h) \leq \epsilon] \geq 1 - \delta$$

using polynomially many samples and computational steps.

1.12 Symbolic AI vs Machine Learning

1.12.1 What is Symbolic AI?

Also known as **Good Old-Fashioned AI (GOFAI)**, it represents knowledge using symbols, rules, and logic. It uses explicitly programmed rules for reasoning.

1.12.2 Symbolic AI vs Machine Learning Comparison

Feature	**Symbolic AI**	**Machine Learning**
Knowledge Source	Rules & logic	Data & patterns
Interpretability	Highly explainable	Often a black box
Adaptability	Rigid (manual updates)	Can generalize from data
Data Requirements	Minimal	Requires large datasets
Best Use Cases	Theorem proving	NLP, computer vision

Table 1.1: Comparison of Symbolic AI and Machine Learning.

Chapter 2

Foundations of Computation

2.1 What is Computation?

Computation is the process of performing calculations, manipulating data, or executing a sequence of operations to solve problems or transform inputs into desired outputs. It encompasses both the theoretical and practical aspects of processing information.

2.2 Computational Models: Theoretical Foundations

A **computational model** is a mathematical or conceptual framework that defines how computation is carried out. For an arbitrary computing model, the following metaphoric expression has been proposed:

$$\textbf{computation} = \textbf{storage} + \textbf{transmission} + \textbf{processing}$$

2.3 Four Fundamental Computational Models

2.3.1 Turing Machine (Alan Turing, 1936)

The Foundation of Algorithmic Computation

Core Characteristics

- **Style**: Imperative / mechanical model of computation
- **Core Idea**: A machine reads/writes symbols on an infinite tape with a finite set of rules
- **Representation**:
  - Infinite tape divided into cells
  - Head that can read/write and move left or right
  - Finite state machine controlling transitions

Strengths and Limitations

Strengths:

- Canonical model for algorithmic computability
- Basis of the Church—Turing Thesis
- Directly models sequential execution

Limitations:

- Low-level, not efficient
- Sequential by nature, doesn't capture parallelism well

Example: A Turing Machine can simulate any algorithm you'd run on a modern computer (given enough tape).

2.3.2 Lambda Calculus (Alonzo Church, 1930s)

The Foundation of Functional Computation

Core Characteristics

- **Style**: Functional model of computation
- **Core Idea**: Everything is a function. Computation = function application + substitution
- **Representation**:
  - Variables (`x`)
  - Function definitions (λ`x. expression`)
  - Function application (`(f x)`)

Strengths and Limitations

Strengths:

- Basis of functional programming (Haskell, Lisp)
- Good for reasoning about higher-order functions, abstraction, recursion

Limitations:

- Abstract and symbolic; not naturally tied to hardware
- Efficiency is not modeled, just computability

Example: Addition can be defined entirely in terms of functions (Church numerals).

2.3.3 Cellular Automata (Stanislaw Ulam, John von Neumann, later Conway)

The Foundation of Distributed/Parallel Computation

Core Characteristics

- **Style**: Spatial / distributed model of computation
- **Core Idea**: Computation arises from simple local rules applied to a grid of cells over time
- **Representation**:
  - Infinite (or finite) grid of cells
  - Each cell has a finite state (e.g., alive/dead)
  - Transition rules depend only on the local neighborhood

Strengths and Limitations

Strengths:

- Good for modeling parallel, distributed, physical systems
- Supports universal computation (Conway's Life is Turing-complete)

Limitations:

- Not natural for symbolic or algebraic computation
- More suited for simulating dynamics

Example: Conway's *Game of Life* shows how simple rules produce complex, even universal, behaviors.

2.3.4 Biological Computation (Inspired by Nature)

The Foundation of Adaptive/Learning Computation

Core Characteristics

Biological models are inspired by living systems and emphasize parallelism, adaptability, and learning.

- **Learns from data** (training) rather than using fixed rules
- Massive parallelism and fault tolerance
- Self-organization and adaptation
- Pattern recognition and generalization

Examples of Biological Computation

Neural Networks:

- Inspired by the brain's neurons and synapses
- Computation happens through weighted sums and nonlinear activations
- Foundation of modern AI (deep learning for vision, NLP, etc.)

DNA Computing:

- Uses DNA strands and biochemical reactions to encode and solve problems
- Enables massive parallelism (billions of molecules interacting at once)
- Example: Adleman (1994) solved a small Hamiltonian Path problem with DNA

Swarm Intelligence:

- Inspired by ants, bees, and bird flocks
- Simple agents interacting lead to complex global solutions
- Example: Ant Colony Optimization for shortest path problems

2.4 Biological Neural Networks: Nature's Computational Model

The following are key characteristics that make biological neural networks powerful computational systems:

2.4.1 Characteristics of Biological Neural Networks

- **Highly interconnected:** Neurons form a complex web of connections
- **Robustness and Fault Tolerance:** The decay of nerve cells does not affect the overall function of the network significantly
- **Flexibility:** The ability to reorganize and adapt to new situations
- **Handling incomplete information:** Ability to infer appropriate outputs even when some inputs are missing or noisy
- **Parallel processing:** Multiple neurons can process information simultaneously

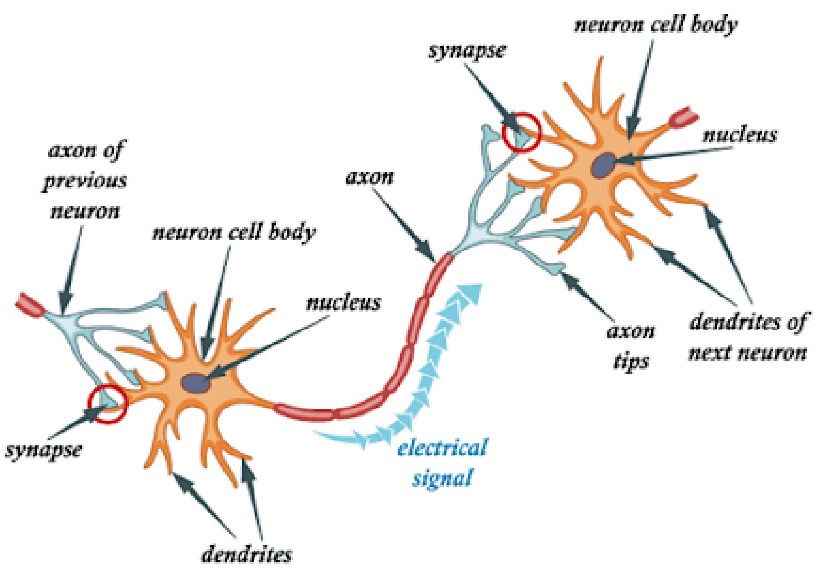

Figure 2.1: Structure of a biological neuron.

2.4.2 Neuron Structure

2.4.3 Neuron Structure and Components

- **Fundamental unit**: neuron (cell body / soma, dendrites, axon, synapses)
- **Dendrites** receive inputs; **axon** transmits output and branches to many synapses (often thousands)
- **Synapse**: junction between axon terminal and target cell
- **Synaptic junctions** form between presynaptic axon terminals and postsynaptic dendrites or the cell body

Typical Sizes

- soma $\sim$ 10–80 μm
- synaptic gap $\sim$ 200 nm
- neuron length from 0.01 mm to 1 m

2.4.4 Signal Transmission and Firing

- **Resting potential** $\sim$ -70 mV; depolarization above threshold (roughly $\sim$10 mV) triggers firing
- **Action potentials** are all-or-none pulses sent down the axon; information is encoded in firing rate ($\sim$1–100 Hz)
- **Propagation speed** in brain tissue $\sim$ 0.5–2 m/s; synaptic transmission delay $\sim$ 0.5 ms
- After firing the membrane recovers (**refractory period**); synaptic effects decay with time constant $\sim$5–10 ms

2.4.5 Synapses: Chemistry and Types

- **Transmission** across synapse is chemical: neurotransmitters released from presynaptic terminal
- **Postsynaptic effect** can be excitatory (depolarizing) or inhibitory (hyperpolarizing)
- All endings of a given axon are typically either excitatory or inhibitory
- **Synaptic strength** depends on activity and can change over time (basis for learning)

2.4.6 Plasticity and Learning

Active synapses that repeatedly contribute to postsynaptic firing tend to strengthen; inactive ones weaken. **Hebb's rule** ("cells that fire together, wire together") describes this activity-dependent plasticity. Continuous modification of synaptic strengths underlies learning and memory formation.

2.5 Artificial Neural Networks

2.5.1 Introduction: From Biology to Computation

Artificial Neural Networks (ANNs) represent one of the most successful attempts to harness the computational principles observed in biological neural systems for solving complex problems.

2.5.2 The Abstract Neuron: Building Block of Intelligence

The output of a neuron can be expressed as:

$$Y = f\left(\sum_{i=1}^{n} W_i X_i + b\right) = f(\mathbf{W}^T\mathbf{X} + b)$$

Where:

- Y: Output of the neuron
- f: Activation function (primitive function that introduces non-linearity)
- W_i: Weight associated with input i (learnable parameter)
- X_i: Value of input i
- b: Bias term (learnable parameter that shifts the activation function)
- $\mathbf{W} = [W_1, W_2, \ldots, W_n]^T$: Weight vector
- $\mathbf{X} = [X_1, X_2, \ldots, X_n]^T$: Input vector

Mathematical Foundation: Linear Combination and Affine Transformation

The computation $\mathbf{W}^T\mathbf{X}+b$ represents an **affine transformation** of the input space. This can be broken down as:

1. **Linear transformation**: $\mathbf{W}^T\mathbf{X}$ scales and rotates the input vector
2. **Translation**: Adding bias b shifts the result by a constant

The activation function f then introduces non-linearity, enabling the neuron to model complex, non-linear relationships between inputs and outputs.

2.6 Activation Functions: Mathematical Properties

Activation functions are crucial for introducing non-linearity into neural networks. Different activation functions have distinct mathematical properties that affect learning dynamics.

2.6.1 Common Activation Functions

Step Function (Heaviside)

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

Properties: Non-differentiable, binary output, historically important for perceptron.

Sigmoid Function

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Properties:

- Smooth, differentiable: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$
- Range: $(0, 1)$
- Problem: Vanishing gradients for large $|z|$

Hyperbolic Tangent

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\sigma(2z) - 1$$

Properties:

- Range: $(-1, 1)$
- Zero-centered output
- Derivative: $\tanh'(z) = 1 - \tanh^2(z)$

Rectified Linear Unit (ReLU)

$$\text{ReLU}(z) = \max(0, z)$$

Properties:

- Computationally efficient
- Alleviates vanishing gradient problem
- Non-differentiable at $z = 0$
- Can suffer from "dying ReLU" problem

2.6.2 Mathematical Requirements for Activation Functions

For universal approximation, activation functions should be:

1. **Non-linear**: Otherwise, multiple layers collapse to a single linear transformation
2. **Differentiable**: Enables gradient-based optimization (almost everywhere is sufficient)
3. **Monotonic**: Helps with optimization landscape (not strictly required)
4. **Bounded or unbounded**: Different properties affect convergence behavior

2.6.3 Neural Networks as Function Approximators

With sufficient neurons and appropriate activation functions, neural networks can approximate any continuous function to arbitrary precision.

Universal Approximation Theorem

Theorem (Cybenko, 1989; Hornik, 1991): Let ϕ be a continuous sigmoid-type function. Then finite sums of the form:

$$F(x) = \sum_{j=1}^{N} \alpha_j \phi(y_j^T x + \theta_j)$$

are dense in $C(I_n)$, the space of continuous functions on the unit hypercube $I_n = [0, 1]^n$.

Implications

- **Existence**: There exists a neural network that can approximate any continuous function
- **No constructive proof**: Doesn't tell us how to find the network
- **Width vs. Depth**: Original theorem about width; depth can be more efficient
- **Approximation vs. Learning**: Says nothing about learnability from data

Modern Extensions

- **ReLU networks**: Also have universal approximation properties
- **Deep vs. Wide**: Deep networks can be exponentially more efficient than wide ones
- **Smooth functions**: Require fewer neurons than general continuous functions

2.7 Artificial Neural Networks: From Theory to Implementation

2.7.1 Fundamental Architecture: Primitive Functions and Composition Rules

To understand artificial neural networks, we must first examine their core computational elements. Every computational model requires:

1. **Primitive Functions**: Basic operations that cannot be decomposed further
2. **Composition Rules**: Ways to combine primitive functions to create complex behaviors

Primitive Functions in Neural Networks

In artificial neural networks, **primitive functions are located in the nodes (neurons) of the network**. Each node implements a specific mathematical transformation that processes incoming information and produces an output.

Composition Rules in Neural Networks

The **composition rules are contained implicitly in**:

- **Interconnection pattern of the nodes**: How neurons are connected determines information flow
- **Synchrony or asynchrony of information transmission**: Whether neurons update simultaneously or in sequence
- **Presence or absence of cycles**: Whether information can flow in loops (recurrent networks) or only forward (feedforward networks)

This differs fundamentally from traditional computing models:

Computing Model	**Primitive Functions**	**Composition Rules**
von Neumann Processor	Machine instructions (ADD, MOVE, JUMP)	Program sequence + control flow
Artificial Neural Networks	Neuron activation functions	Network topology + connection weights + timing

Table 2.1: Comparison of primitive functions and composition rules across computing models.

2.7.2 Neural Networks as Function Approximators

Networks of Primitive Functions

Artificial neural networks are nothing but networks of primitive functions. Each node transforms its input into a precisely defined output, and the combination of these transformations creates complex computational behaviors.

The Network Function

Consider a neural network that takes inputs (x, y, z) and produces an output through nodes implementing primitive functions f_1, f_2, f_3, f_4. The network can be thought of as implementing a **network function** ϕ:

$$\phi(x, y, z) = f_4(a_4 \cdot f_3(a_3 \cdot f_2(a_2 \cdot f_1(a_1 \cdot x))) + \dots)$$

Where $a_1, a_2, \dots, a_5$ are the weights of the network. **Different selections of weights produce different network functions.**

Three Critical Elements

Different models of artificial neural networks differ mainly in three fundamental aspects:

1. **Structure of the Nodes**
   - Choice of activation function (sigmoid, ReLU, tanh, etc.)
   - Input integration method (weighted sum, product, etc.)
   - Presence of bias terms
2. **Topology of the Network**
   - Feedforward vs. recurrent connections
   - Number of layers and neurons per layer
   - Connection patterns (fully connected, sparse, convolutional)
3. **Learning Algorithm**
   - Method for finding optimal weights
   - Supervised vs. unsupervised vs. reinforcement learning
   - Optimization techniques (gradient descent, evolutionary algorithms)

2.7.3 Function Approximation: The Classical Problem

Historical Context

Function approximation is a classical problem in mathematics: **How can we reproduce a given function $F : \mathbb{R} \to \mathbb{R}$ either exactly or approximately using a given set of primitive functions?**

Traditional approaches include:

- **Polynomial approximation**: Using powers of x (Taylor series)
- **Fourier approximation**: Using trigonometric functions (sine and cosine)
- **Spline approximation**: Using piecewise polynomials

Neural Networks as Universal Approximators

Neural networks provide a revolutionary approach to function approximation:

Key Insight: With sufficient neurons and appropriate activation functions, neural networks can approximate any continuous function to arbitrary precision (Universal Approximation Theorem).

Advantages of Neural Network Approximation

1. **Adaptive**: Networks learn the approximation from data rather than requiring explicit mathematical formulation

2. **Flexible**: Can handle high-dimensional inputs and complex, non-linear relationships

3. **Robust**: Can generalize to unseen data and handle noise

4. **Parallel**: Multiple neurons can process different aspects of the input simultaneously

2.7.4 Learning from Data: The Key Difference

The main difference between Taylor or Fourier series and artificial neural networks is, however, that **the function F to be approximated is given not explicitly but implicitly through a set of input-output examples.** We know F only at some points but we want to generalize as well as possible. This means that we try to adjust the parameters of the network in an optimal manner to reflect the information known and to extrapolate to new input patterns which will be shown to the network afterwards. This is the task of the learning algorithm used to adjust the network's parameters.

Classical Series vs. Neural Networks: A Fundamental Distinction

Classical Mathematical Series (Taylor/Fourier):

- **Explicit Function Definition**: The function $F(x)$ is mathematically defined and known
- **Analytical Coefficients**: Series coefficients can be computed directly using calculus
  - Taylor: $a_n = F^{(n)}(x_0)/n!$ (nth derivative at expansion point)
  - Fourier: a_n, b_n computed via integration over the function's period
- **Perfect Representation**: Given enough terms, the series can represent the function exactly
- **No Learning Required**: Coefficients are determined mathematically, not learned

Artificial Neural Networks:

- **Implicit Function Definition**: The function F is unknown but represented by data points
- **Learned Parameters**: Network weights and biases are learned from examples
- **Approximation from Samples**: Must generalize from finite training data to unknown inputs
- **Adaptive Learning**: Parameters adjust through iterative optimization algorithms

Mathematical Formulation of the Learning Problem

Given a training dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$, we seek to find parameters $\boldsymbol{\theta}$ that minimize the empirical risk:

$$\mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} L(f(\mathbf{x}_i; \boldsymbol{\theta}), y_i)$$

where:

- $L(\cdot, \cdot)$: Loss function measuring prediction error
- $f(\mathbf{x}; \boldsymbol{\theta})$: Neural network function with parameters $\boldsymbol{\theta}$
- Goal: Minimize true risk $\mathcal{R}(\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{x},y)\sim P}[L(f(\mathbf{x}; \boldsymbol{\theta}), y)]$

Generalization Gap

The fundamental challenge is the generalization gap:

$$\text{Generalization Gap} = \mathcal{R}(\boldsymbol{\theta}) - \mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$$

This gap can be controlled through:

1. **Regularization**: Adding penalty terms to control model complexity
2. **Cross-validation**: Using held-out data to estimate generalization performance
3. **Early stopping**: Halting training before overfitting occurs
4. **Data augmentation**: Artificially increasing training set size

2.8 Computational Complexity in Neural Networks

2.8.1 Forward Pass Complexity

For a neural network with L layers, where layer l has n_l neurons:

- **Matrix multiplication**: $O(n_{l-1} \times n_l)$ for each layer
- **Total forward pass**: $O(\sum_{l=1}^{L} n_{l-1} \times n_l)$
- **Activation functions**: $O(n_l)$ per layer (typically much smaller than matrix operations)

2.8.2 Backward Pass Complexity (Backpropagation)

- **Gradient computation**: Same order as forward pass $O(\sum_{l=1}^{L} n_{l-1} \times n_l)$
- **Parameter updates**: $O(\text{total parameters})$
- **Memory complexity**: $O(\text{total activations})$ to store intermediate values

2.8.3 Scalability Considerations

- **Batch processing**: Process multiple examples simultaneously for efficiency
- **Parallelization**: Matrix operations are highly parallelizable on GPUs
- **Memory-computation tradeoff**: Can reduce memory by recomputing activations

2.8.4 Threshold Logic: The Foundation

The simplest kind of computing units used to build artificial neural networks are based on threshold logic. These computing elements are a generalization of the common logic gates used in conventional computing and, since they operate by comparing their total input with a threshold, this field of research is known as **threshold logic**.

2.9 McCulloch-Pitts Neuron: The First Artificial Neuron

The McCulloch-Pitts neuron, introduced in 1943, was the first mathematical model of an artificial neuron. It established the theoretical foundation for neural computation using threshold logic.

2.9.1 Mathematical Model

The McCulloch-Pitts neuron computes its output according to:

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} w_i x_i \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

Where:

- x_i are the input values (binary: 0 or 1)
- w_i are the corresponding weights
- θ is the threshold value
- y is the binary output (0 or 1)

2.9.2 Logic Gate Implementation

The McCulloch-Pitts model can implement basic logic functions:

AND Gate

For an AND gate with two inputs:

- Weights: $w_1 = w_2 = 1$
- Threshold: $\theta = 2$
- Result: Output is 1 only when both inputs are 1

OR Gate

For an OR gate with two inputs:

- Weights: $w_1 = w_2 = 1$
- Threshold: $\theta = 1$
- Result: Output is 1 when at least one input is 1

2.9.3 Limitations of McCulloch-Pitts Neurons

- **Fixed weights**: No learning mechanism
- **Binary inputs only**: Cannot handle continuous values
- **Synchronous operation**: All neurons fire simultaneously
- **No adaptation**: Cannot modify behavior based on experience

These limitations led to the development of the Perceptron, which introduced learning capabilities.

2.10 Historical Development of Neural Networks

2.10.1 Timeline of Neural Network Evolution

The development of neural networks has proceeded through several distinct phases, each marked by significant theoretical breakthroughs and practical applications.

Period	**Year**	**Key Development**	**Contributors**	**Description**
Early Foundations	1943	McCulloch-Pitts Neuron	Warren McCulloch, Walter Pitts	First mathematical model of artificial neuron using threshold logic
	1949	Hebbian Learning Rule	Donald Hebb	"Cells that fire together, wire together" - synaptic plasticity principle
First Generation	1957	Perceptron	Frank Rosenblatt	First trainable neural network with learning algorithm
	1960	ADALINE/MADALINE	Bernard Widrow, Marcian Hoff	Adaptive linear neurons with delta rule learning
Winter Period	1969	Perceptron Limitations	Marvin Minsky, Seymour Papert	Proved perceptrons cannot solve XOR problem
Revival Era	1982	Hopfield Networks	John Hopfield	Recurrent networks for associative memory
	1986	Backpropagation	Rumelhart, Hinton, Williams	Efficient algorithm for training multi-layer networks
Modern Era	2012	AlexNet	Alex Krizhevsky, Geoffrey Hinton	Deep CNN wins ImageNet competition
	2017	Transformer Architecture	Vaswani et al. (Google)	Attention-based model for sequences

Table 2.2: Key milestones in neural network development.

Chapter 3

Linear Classification

3.1 Learning Goals

By the end of this chapter, you will:

- **Know what is meant by binary linear classification** and understand its fundamental concepts
- **Understand why an explicit threshold for a classifier is redundant** and how bias terms can be eliminated using dummy features
- **Be able to specify weights and biases by hand** to represent simple logical functions (AND, OR, NOT)
- **Be familiar with input space and weight space**, including:
  - Plotting training cases and classification weights in both spaces
  - Understanding the geometric interpretation of linear classifiers
- **Be aware of the limitations of linear classifiers**, including:
  - Understanding convexity and its role in linear separability
  - Knowing how basis function representations can overcome some limitations

3.2 Fundas: Mathematical Foundations

Mathematical Foundations

3.2.1 Vector Representation

- **Input vectors**: Each data point is represented as a D-dimensional vector $\boldsymbol{x}^{(i)} = [x_1^{(i)}, x_2^{(i)}, \ldots, x_D^{(i)}]$
- **Weight vectors**: Classification parameters represented as $\boldsymbol{w} = [w_1, w_2, \ldots, w_D]$
- **Linear combination**: $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + b$, where b is the bias term
- **Decision boundary**: The hyperplane where $\boldsymbol{w}^T\boldsymbol{x} + b = 0$

3.2.2 Binary Classification Framework

- **Target values**: $t^{(i)} \in \{0, 1\}$ where $0 =$ negative class, $1 =$ positive class
- **Classification rule**: $\hat{y} = 1$ if $\boldsymbol{w}^T\boldsymbol{x} + b > \tau$, else $\hat{y} = 0$ (where τ is threshold)

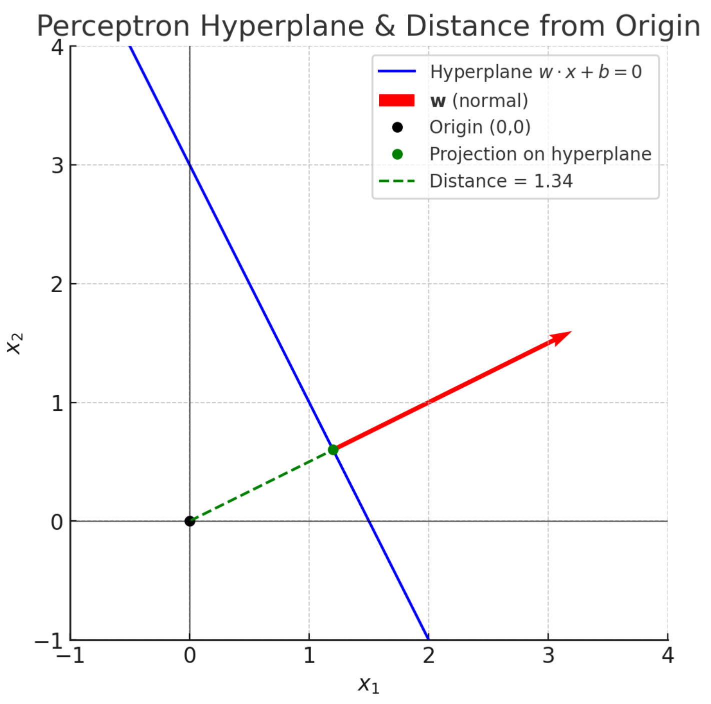

Figure 3.1: Hyperplane in 2D input space. The line represents the decision boundary where $\boldsymbol{w}^T\boldsymbol{x}+b = 0$.

- **Training set**: $\{(\boldsymbol{x}^{(i)}, t^{(i)})\}_{i=1}^N$ where N is the number of examples

What is Hyperplane ?

Definition 1 (Hyperplane)**.** *A hyperplane in a D-dimensional space is a flat affine subspace of dimension $D-1$. It can be defined by a linear equation of the form:*

$$\boldsymbol{w}^T\boldsymbol{x} + b = 0 \tag{3.1}$$

where $\boldsymbol{w}$ is a normal vector to the hyperplane, $\boldsymbol{x}$ is a point on the hyperplane, and b is the bias term.

Properties of Hyperplanes:

- **Normal Vector**: The weight vector $\mathbf{w}$ is perpendicular to the decision boundary
- **Distance from Origin**: $\frac{|b|}{||\mathbf{w}||_2}$ gives the perpendicular distance from the hyperplane to the origin
- **Classification Rule**:
  - Points where $\mathbf{w}^T\mathbf{x} + b > 0$ are classified as positive halfspcace (class 1)
  - Points where $\mathbf{w}^T\mathbf{x} + b < 0$ are classified as negative halfspace (class 0)
  - Points on the boundary satisfy $\mathbf{w}^T\mathbf{x} + b = 0$

Distance of a point from the Hyperplane
The distance d of a point $\mathbf{x}_i$ from the hyperplane defined by $\mathbf{w}^T\mathbf{x} + b = 0$ is given by the formula:

$$d = \frac{|\mathbf{w}^T\mathbf{x}_i + b|}{||\mathbf{w}||_2}$$

Where:

- $\mathbf{w}^T\mathbf{x}_i + b$ is the signed distance from the hyperplane (positive if on the positive side, negative if on the negative side)
- $||\mathbf{w}||_2$ is the Euclidean norm of the weight vector, which normalizes the distance

- The absolute value ensures the distance is non-negative

Example:

- If $y_i = +1$ (positive class) and $\mathbf{w}^T\mathbf{x}_i = +2.5 \rightarrow$ product $= +2.5$ (correct)
- If $y_i = +1$ (positive class) and $\mathbf{w}^T\mathbf{x}_i = -1.2 \rightarrow$ product $= -1.2$ (incorrect)
- If $y_i = -1$ (negative class) and $\mathbf{w}^T\mathbf{x}_i = -0.8 \rightarrow$ product $= +0.8$ (correct)

What is Linearly Separable?

Definition 2 (Linear Separability)**.** *A dataset is said to be linearly separable if there exists a hyperplane that can separate all positive examples from all negative examples without any misclassification.*

Implications of Linear Separability:

- If a dataset is linearly separable, a linear classifier can achieve perfect classification
- If not, more complex models or feature transformations may be necessary

Why Hyperplanes Matter in Linear Classification?

- In binary linear classification, the decision boundary is represented by a hyperplane
- The hyperplane separates the input space into two regions corresponding to the two classes
- The orientation and position of the hyperplane are determined by the weights $\boldsymbol{w}$ and bias b

3.3 Introduction to Binary Classification

Binary classification represents one of the most fundamental problems in machine learning, where the goal is to predict a binary-valued target from input features. This forms the foundation for understanding more complex classification scenarios.

3.3.1 Real-World Applications

Medical Diagnosis Systems

- **Problem**: Predict whether a patient has a specific disease
- **Features**: Symptoms, test results, patient history
- **Target**: Disease present (1) or absent (0)
- **Example**: Diagnosing diabetes from glucose levels, BMI, and family history

Email Spam Detection

- **Problem**: Classify emails as spam or legitimate
- **Features**: Word frequencies, sender information, email metadata
- **Target**: Spam (1) or not spam (0)
- **Example**: Using keywords like “free money” as indicators

Fraud Detection

- **Problem**: Identify fraudulent transactions
- **Features**: Transaction amount, time, location, merchant type
- **Target**: Fraudulent (1) or legitimate (0)
- **Example**: Detecting unusual spending patterns

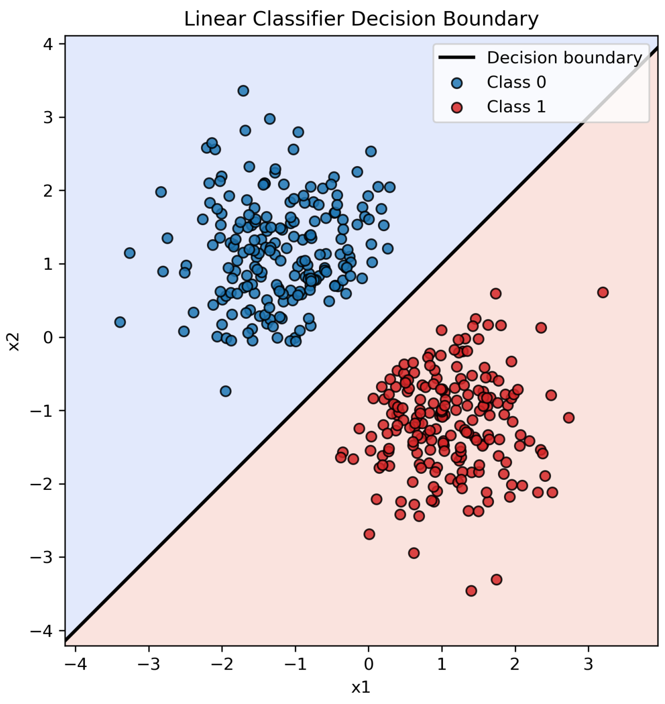

Figure 3.2: A simple binary linear classifier with two features. The decision boundary (line) separates the two classes.

3.4 Binary Linear Classifiers: First Principles

3.4.1 Core Concept

A binary linear classifier makes decisions by computing a linear function of the input features and comparing the result to a threshold. This approach assumes that the two classes can be separated by a linear decision boundary in the feature space.

3.4.2 Mathematical Formulation

Step 1: Linear Combination

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_D x_D + b = w^T \boldsymbol{x} + by = \begin{cases} 1 & \text{if } z \geq \tau \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

Step 2: Threshold Decision

$$\hat{y} = \begin{cases} 1 & \text{if } z > \tau \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

Where:

- w_i: Weight for feature i (determines importance and direction)
- x_i: Value of feature i
- b: Bias term (shifts the decision boundary)
- τ: Threshold value

3.4.3 Eliminating Redundancy: The Bias-Threshold Trick

Problem: Having both bias (b) and threshold (τ) is redundant.

Step 1: Absorb threshold into bias term:

- Set new bias: $b' = b - \tau$
- Set threshold to zero: $\tau = 0$
- Decision rule becomes: $\hat{y} = 1$ if $\boldsymbol{w}^T\boldsymbol{x} + b' > 0$

Step 2: Add dummy feature:

- Extend input: $\boldsymbol{x} \to [\boldsymbol{x}, 1]$
- Extend weights: $\boldsymbol{w} \to [\boldsymbol{w}, b']$
- Decision rule: $\hat{y} = 1$ if $\boldsymbol{w}_{\text{extended}}^T \boldsymbol{x}_{\text{extended}} > 0$

Note

Without loss of generality, we can say $z = \boldsymbol{w}^T\boldsymbol{x}$

3.4.4 Some Examples

Example 1: For NOT gate

Truth Table:

Input	Output
0	1
1	0

- Each of the traning cases provides a constraint on the weights and biases.
- For example, the first training case $(0, 1)$ requires that $z = 0 \cdot w_1 + b \geq 0$, which simplifies to $b \geq 0$. Technically b can be 0 but it is good practice to avoid the solutions which lies on decision boundary, so let us take tentatively $b = 1$.
- The second training case $(1, 0)$ requires that $z = 1 \cdot w_1 + b < 0$. We can satisfy this inequality by setting $w_1 = -2$.
- Thus, we have $w_1 = -2$ and $b = 1$.

Example 2: For AND gate

Truth Table:

Input 1	Input 2	Output
0	0	0
0	1	0
1	0	0
1	1	1

$$\begin{aligned} \text{Case } (0,0): &\quad b < 0 \\ \text{Case } (0,1): &\quad w_2 + b < 0 \\ \text{Case } (1,0): &\quad w_1 + b < 0 \\ \text{Case } (1,1): &\quad w_1 + w_2 + b \geq 0 \end{aligned}$$

Normally finding the weights and biases are a bit of trial and error. But let us try to solve it systematically.

- From the first case, we can set $b < 0$
- From the fourth, it seems that w_1 and w_2 must be positive.

- Since the problem is symmetric in w_1 and w_2, let us set $w_1 = w_2 = w > 0$.
- Now, the second and third cases become $w + b < 0$
- The fourth case becomes $2w + b \geq 0$
- Let us set $b = -1$ (satisfies the first case)
- Then, $w - 1 < 0$ implies $w < 1$ and $2w - 1 \geq 0$ implies $w \geq 0.5$
- We can satisfy both by setting $w = 0.5$
- Thus, we have $w_1 = 0.5$, $w_2 = 0.5$ and $b = -1$

3.5 Geometric Interpretation

3.5.1 Data Space

- **Data Space (Input Space)**: The first space to be familiar with is data space, or input space. Each point in this space corresponds to a possible input vector.
- **Representation of Examples**: It's customary to represent positive and negative examples with the symbols "+" and "-", respectively.
- **Division of Space**: Once we've chosen the weights $\boldsymbol{w}$ and bias b, we can divide the data space into a region where the points are classified as positive (the positive region), and a region where the points are classified as negative (the negative region).
- **Decision Boundary**: The boundary between these regions, i.e., the set where $\boldsymbol{w}^T\boldsymbol{x} + b = 0$, is called the decision boundary. Think back to your linear algebra class, and recall that the set determined by this equation is a hyperplane.
- **Half-Space**: The set of points on one side of the hyperplane is called a half-space.

Decision Boundary: The hyperplane $\boldsymbol{w}^T\boldsymbol{x} + b = 0$ divides the input space into two regions:

- **Positive region**: $\boldsymbol{w}^T\boldsymbol{x} + b > 0$ (predicted class 1)
- **Negative region**: $\boldsymbol{w}^T\boldsymbol{x} + b < 0$ (predicted class 0)

3.5.2 Weight Space

- Each point in weight space corresponds to a weight vector $\boldsymbol{w}$. Here we will not consider the bias terms explicitly, since we can always absorb them into the weights by adding a dummy feature to the input vectors.
- Each training example imposes a constraint on the weights. For example, if we have a positive example $\boldsymbol{x}^{(i)}$, then we must have $\boldsymbol{w}^T\boldsymbol{x}^{(i)} > 0$ in order to classify it correctly. The set of points in weight space that satisfy this inequality is a half-space.
- Similarly, a negative example $\boldsymbol{x}^{(j)}$ imposes the constraint $\boldsymbol{w}^T\boldsymbol{x}^{(j)} < 0$, which also defines a half-space in weight space.
- Each training example thus carves out a half-space in weight space.
- The intersection of all these half-spaces defines the region of weight vectors that correctly classify all training examples which is called the **feasible region**.

3.5.3 Illustrating Input Space and Weight Space with the NOT Function

The input matrix is given by after applying the bias trick(that is $x_0 = 1$):

$$X = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \tag{3.4}$$

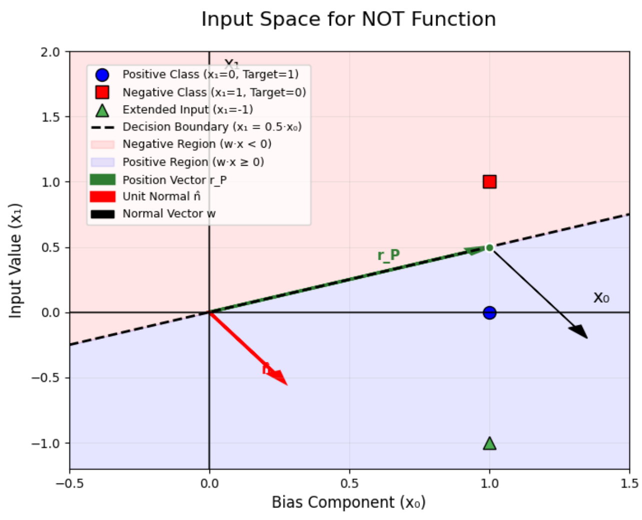

Figure 3.3: Input space for the NOT function using the bias trick. The decision boundary is $x_1 = -\frac{b}{w_1}$.

Weight Space Visualization

The bias trick will give us $[w_0, w_1]$ as the weight vector. The constraints from the training examples are:

$$\text{For } (0,1): \quad w_0 > 0 \tag{3.5}$$
$$\text{For } (1,0): \quad w_1 + w_0 < 0 \tag{3.6}$$

Plotting the Constraints:

- The first constraint $w_0 > 0$ corresponds to the half-space above the line $w_0 = 0$.
- The second constraint $w_1 + w_0 < 0$ can be rearranged to $w_0 < -w_1$, which corresponds to the half-space below the line $w_0 = -w_1$.

The feasible region is where both constraints are satisfied.

3.5.4 Illustrating Input Space and Weight Space with the AND Function

The input matrix is given by after applying the bias trick(that is $x_0 = 1$):

$$X = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \tag{3.7}$$

Weight Space Visualization

The bias trick will give us $[w_0, w_1, w_2]$ as the weight vector. The constraints from the training examples are:

$$\text{For } (0,0): \quad w_0 < 0 \tag{3.8}$$
$$\text{For } (0,1): \quad w_2 + w_0 < 0 \tag{3.9}$$
$$\text{For } (1,0): \quad w_1 + w_0 < 0 \tag{3.10}$$
$$\text{For } (1,1): \quad w_1 + w_2 < 0 \tag{3.11}$$

Plotting the Constraints:

- The first constraint $w_0 < 0$ corresponds to the half-space below the line $w_0 = -0.3$.

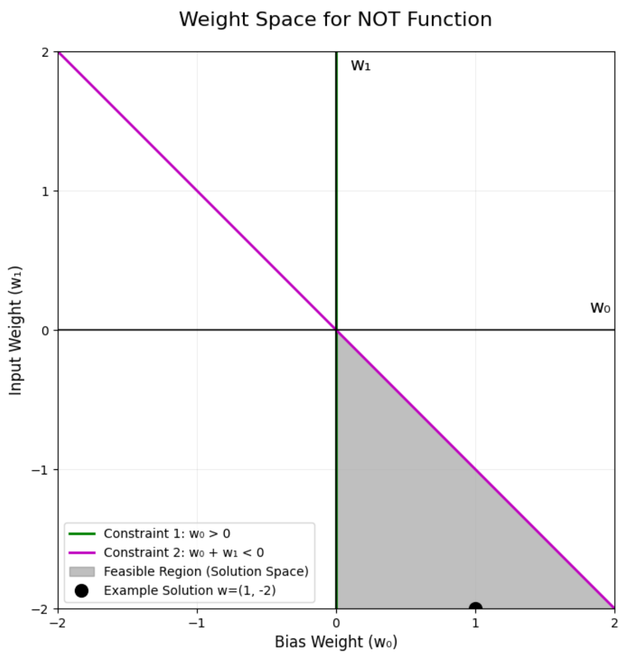

Figure 3.4: Weight space for the NOT function using the bias trick. The feasible region is $b > 0$ and $w_1 + b < 0$.

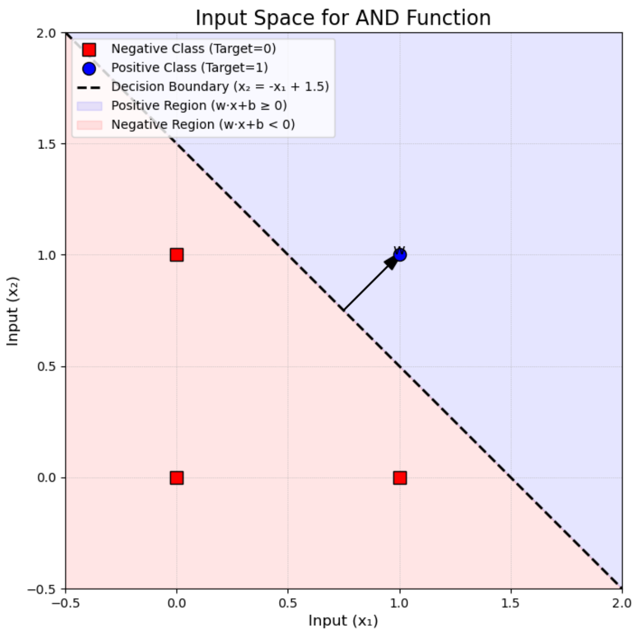

Figure 3.5: Input space for the AND function using the bias trick. The decision boundary is $x_1 + x_2 = -\frac{b}{w_1}$.

- The second constraint $w_2 + -0.3 < 0$ can be rearranged to $-0.3 < -w_2$, which corresponds to the half-space below the line $w_2 = 0.3$.
- The third constraint $w_1 + -0.3 < 0$ can be rearranged to $w_1 < 0.3$, which corresponds to the half-space below the line $w_1 = 0.3$.
- The fourth constraint $w_1 + w_2 + -0.3 \geq 0$ can be rearranged to $w_1 + w_2 \geq 0.3$, which corresponds to the half-space above the line $w_1 + w_2 = 0.3$.

The feasible region is where all four constraints are satisfied.

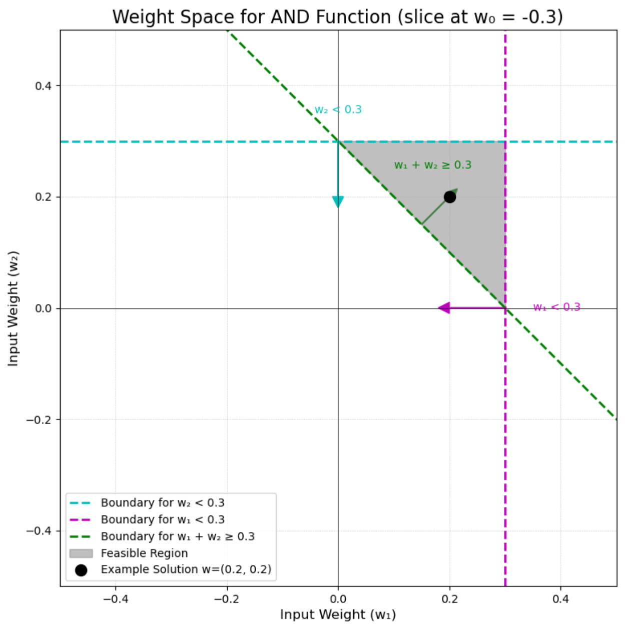

Figure 3.6: Weight space for the AND function using the bias trick. The feasible region is defined by the four constraints, and the point $w_0 = -0.3$, $w_1 = 0.2$, $w_2 = 0.2$ is highlighted as an example.

3.6 Perceptron

3.7 The Perceptron: A Detailed Introduction

The Perceptron is a simple binary classifier that serves as the foundational building block for more complex neural networks.

3.7.1 Definition: The Anatomy of a Perceptron

For an input vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, and the target vector $\mathbf{t} = (t_1, t_2, \ldots, t_n)$, the Perceptron computes a single output y. This is done in two steps:

1. **Compute a Weighted Sum:** The model calculates a weighted sum of the inputs, after the bias trick. This is the net input z.

$$z = (w_1x_1 + w_2x_2 + \cdots + w_nx_n) + b = \mathbf{w}^{\mathbf{T}} \cdot \mathbf{x}$$

2. **Apply an Activation Function:** The output z is passed through a Heaviside step function.

$$y = \phi(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

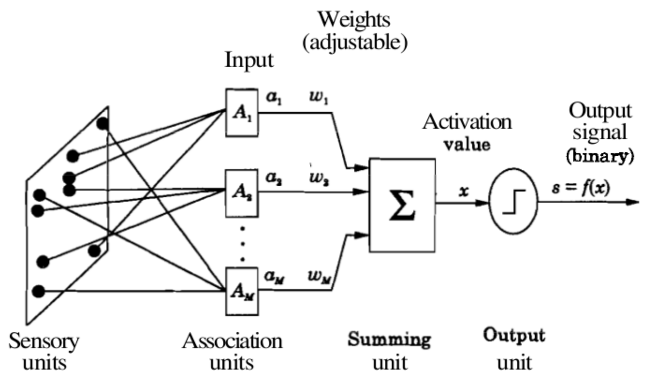

Figure 3.7: Architecture of a single-layer Perceptron. Inputs are weighted, summed, and passed through an activation function to produce the output.

3.8 The Perceptron Learning Rule

The Perceptron learns by adjusting its weights $\mathbf{w}$ and bias b based on the errors it makes. This learning rule has strong theoretical foundations.

- **Initialization**: Start with small random weights and bias.
- **Bias Trick**: Incorporate the bias into the weight vector by adding a dummy feature $x_0 = 1$ and w_0.
- **Learning Rate**: A small positive constant η controls the step size during weight updates.
- **Update Rule**: For each training example, update the weights and bias as follows:
- **Error Calculation**: Compute the error $\epsilon = t - y$, where t is the true label.
- **Weight Update**: Adjust the weights and bias:

$$w_i \leftarrow w_i + \eta \cdot \epsilon \cdot x_i \quad \text{for all } i$$

$$b \leftarrow b + \eta \cdot \epsilon$$

- **Classification Condition**: The perceptron correctly classifies a training example $(\mathbf{x}^{(i)}, t^{(i)})$ if:
  - If $t^{(i)} = 1$ (positive class), then $z^{(i)} > 0$
  - If $t^{(i)} = 0$ (negative class), then $z^{(i)} < 0$
- **Margin Condition**: To ensure robustness, we want training examples to be away from the decision boundary:

$$z^{(i)} \cdot t^{(i)} > 0$$

Note

If $\boldsymbol{x}^{(i)}$ is exactly on the decision boundary it will be classified as positive class. But we dont want our training examples to be on the decision boundary because a small noise or perturbation to the input features could change the classification. For robustness we want our training examples to be away from the decision boundary.

$$z^{(i)} \cdot t^{(i)} > 0$$

Functional Margin vs Geometric Margin

Functional Margin:

- The functional margin of a training example $(\mathbf{x}^{(i)}, t^{(i)})$ with respect to a weight vector $\mathbf{w}$ is defined as:
$$\hat{\gamma}^{(i)} = t^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)})$$
- It measures how confidently the example is classified. A larger functional margin indicates a more confident classification.
- However, the functional margin is sensitive to the scale of the weight vector $\mathbf{w}$. Scaling $\mathbf{w}$ by a factor k scales the functional margin by the same factor k.
- Thus, the functional margin does not provide a true measure of the distance from the decision boundary.

3.8.1 The Perceptron Convergence Theorem: A Detailed Treatment

Introduction: A Fundamental Guarantee

The **Perceptron Convergence Theorem** is a cornerstone of machine learning theory. It provides a simple but powerful promise: if your data can be separated by a line or plane (**linearly separable**), the Perceptron algorithm will find a solution in a **finite number of steps**. It proves that the algorithm doesn't just work by chance; it's guaranteed to succeed under the right conditions.

This section explores the proof of this theorem through three lenses: the rigorous algebra, the core intuition, and the elegant geometry.

Convergence Theorem (Rosenblatt, 1962)

Theorem 1 (Perceptron Convergence)**.** *If the training data is linearly separable, the perceptron learning algorithm will converge to a solution in a finite number of steps.*

The Formal Proof: An Algebraic Approach

The proof ingeniously works by showing that two properties of the learning weight vector are on a collision course, forcing the process to end.

The Setup:

- **Data**: A set of samples $\{(\mathbf{x}_i, y_i)\}$, with labels $y_i \in \{+1, -1\}$
- **Assumption**: The data is **linearly separable**, meaning an ideal vector $\mathbf{w}^*$ exists
- **Update Rule**: When a mistake occurs on point i, the weight vector $\mathbf{w}_k$ is updated: $\mathbf{w}_{k+1} = \mathbf{w}_k + y_i\mathbf{x}_i$

Part 1: The Lower Bound (Steady Progress)

This part shows that the weight vector $\mathbf{w}_k$ gets progressively more **aligned** with the ideal vector $\mathbf{w}^*$. We measure this alignment using the dot product.

Key Formula 1: The Recursive Update

$$\mathbf{w}_{k+1} \cdot \mathbf{w}^* \geq \mathbf{w}_k \cdot \mathbf{w}^* + \gamma$$

Here, γ is the **margin**, representing the "correctness score" of the point closest to the ideal hyperplane. Its existence is guaranteed by linear separability. After k updates:

Key Result 1: The Lower Bound

$$\mathbf{w}_k \cdot \mathbf{w}^* \geq k\gamma$$

This means the alignment with the correct solution grows steadily and linearly with every mistake.

Part 2: The Upper Bound (Controlled Growth)

This part shows that the vector's length is constrained and doesn't grow wildly. The squared magnitude of the weight vector is limited because the update rule includes a term that dampens its growth.

Key Result 2: The Upper Bound

$$||\mathbf{w}_k||^2 \leq kR^2$$

Where R^2 is the squared magnitude of the largest input vector. This shows the squared length grows, at most, linearly with the number of mistakes.

Part 3: The Contradiction

We combine these two bounds using the **Cauchy-Schwarz inequality**, which states $(\mathbf{a} \cdot \mathbf{b})^2 \leq ||\mathbf{a}||^2||\mathbf{b}||^2$.

From Part 1, squaring both sides:

$$(\mathbf{w}_k \cdot \mathbf{w}^*)^2 \geq k^2\gamma^2$$

Applying Cauchy-Schwarz:

$$(\mathbf{w}_k \cdot \mathbf{w}^*)^2 \leq ||\mathbf{w}_k||^2||\mathbf{w}^*||^2$$

From Part 2, we know $||\mathbf{w}_k||^2 \leq kR^2$, so:

$$(\mathbf{w}_k \cdot \mathbf{w}^*)^2 \leq kR^2||\mathbf{w}^*||^2$$

Combining the lower and upper bounds:

$$k^2\gamma^2 \leq (\mathbf{w}_k \cdot \mathbf{w}^*)^2 \leq kR^2||\mathbf{w}^*||^2$$

Simplifying:

$$k^2\gamma^2 \leq kR^2||\mathbf{w}^*||^2$$

Dividing by k:

$$k\gamma^2 \leq R^2||\mathbf{w}^*||^2$$

Final Result: The Bound on Mistakes

$$k \leq \frac{R^2||\mathbf{w}^*||^2}{\gamma^2}$$

Since R, $||\mathbf{w}^*||$, and γ are all fixed, finite constants, this proves that the number of mistakes, k, must be less than or equal to a finite number. **The contradiction arises if we assume the algorithm makes infinitely many mistakes**: the left side k would grow to infinity while the right side remains a fixed constant, which is impossible. Therefore, the algorithm must stop after a finite number of updates.

Convergence Bound

The perceptron will make at most $\left(\frac{R}{\gamma}\right)^2$ mistakes, where:

- R is the maximum norm of any training example: $R = \max_i ||\mathbf{x}_i||$
- γ is the margin of the optimal separating hyperplane

Note on Normalization

The general bound derived above is $k \leq \frac{R^2||\mathbf{w}^*||^2}{\gamma^2}$. However, we can simplify this by normalizing $\mathbf{w}^*$.

Since only the *direction* of $\mathbf{w}^*$ matters for classification (not its magnitude), we are free to normalize it to unit length: $\|\mathbf{w}^*\| = 1$.

When we assume $\mathbf{w}^*$ is already normalized (the standard convention), the bound simplifies to $\left(\frac{R}{\gamma}\right)^2$.

The Geometric Interpretation: The Cone and The Sphere

The algebra tells a beautiful geometric story. The proof traps the learning vector in a ”squeeze play.”

- **The Cone of Progress**: The lower bound forces the angle between our vector $\mathbf{w}_k$ and the ideal vector $\mathbf{w}^*$ to get smaller with every mistake. This confines $\mathbf{w}_k$ to an ever-narrowing **cone**.
- **The Sphere of Possibility**: The upper bound forces the tip of the vector $\mathbf{w}_k$ to stay inside a **sphere** whose radius grows slowly (with $\sqrt{k}$).

The Contradiction: The cone narrows faster than the sphere expands. Eventually, the cone becomes so tight that any vector that could fit inside it would have to be longer than the radius of the allowed sphere. The geometric constraints become impossible to satisfy, proving the process must end.

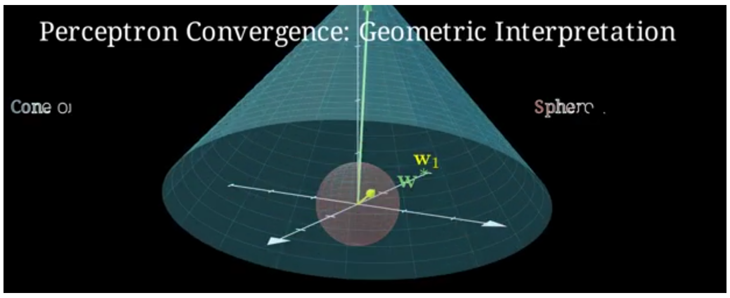

Figure 3.8: Geometric interpretation of Perceptron convergence. Click to view video on YouTube.

Relevance in Modern Deep Learning

Understanding this theorem is more than a historical exercise. It teaches foundational principles that are still relevant today:

- **Intuition for Optimization**: It provides a mental model for how algorithms navigate a solution space
- **Basis for Advanced Concepts**: The concept of a **margin** (γ) is central to more advanced and powerful algorithms like Support Vector Machines (SVMs)
- **Theoretical Rigor**: It's a perfect example of how to formally reason about an algorithm's behavior, a skill essential for creating new methods

3.9 Optimal Separating Hyperplane and Margin Theory

3.9.1 The Optimal Weight Vector $\mathbf{w}^*$

The foundation of perceptron convergence theory rests on the existence of an optimal weight vector $\mathbf{w}^*$ that not only correctly classifies all training examples but does so with a guaranteed minimum margin.

Definition 3 (Optimal Separating Hyperplane)**.** *For a linearly separable dataset* $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ *where* $y_i \in \{-1, +1\}$*, an optimal weight vector* $\mathbf{w}^*$ *is one that correctly classifies all training examples with margin* $\gamma > 0$*:*

$$y_i(\mathbf{w}^{*T}\mathbf{x}_i) \geq \gamma \quad \forall i \in \{1, 2, \ldots, N\}$$

3.9.2 Understanding the Margin Condition

Mathematical Interpretation

The condition $y_i(\mathbf{w}^{*T}\mathbf{x}_i) \geq \gamma$ encodes several crucial properties:

- **Correct Classification**: Since $y_i \in \{-1, +1\}$ and $\gamma > 0$, the product $y_i(\mathbf{w}^{*T}\mathbf{x}_i)$ must be positive, ensuring correct classification
- **Confidence Measure**: The magnitude $|\mathbf{w}^{*T}\mathbf{x}_i|$ represents the "confidence" of classification
- **Minimum Separation**: The margin γ ensures all points are at least a minimum distance from the decision boundary

Geometric Interpretation

- **Positive Examples** ($y_i = +1$): Must satisfy $\mathbf{w}^{*T}\mathbf{x}_i \geq \gamma$
- **Negative Examples** ($y_i = -1$): Must satisfy $\mathbf{w}^{*T}\mathbf{x}_i \leq -\gamma$
- **Decision Boundary**: The hyperplane $\mathbf{w}^{*T}\mathbf{x} = 0$
- **Margin Boundaries**: Hyperplanes $\mathbf{w}^{*T}\mathbf{x} = \pm\gamma$

3.10 Optimal Margin Computation: A Glimpse of SVM

In linear classification, the concept of an **optimal margin** is central to robust decision boundaries. The optimal margin is the largest possible distance between the decision boundary and the closest data points from each class. These closest points are known as *support vectors.*

3.10.1 Algorithm Overview

1. **Identify Support Vectors**: For two linearly separable classes, find the pair of points (one from each class) that are closest to each other.
2. **Compute Direction**: The vector connecting these support vectors defines the orientation of the optimal separating hyperplane.
3. **Place the Decision Boundary**: Position the decision boundary halfway between the support vectors, perpendicular to the direction vector.
4. **Calculate the Margin**: The margin is half the distance between the support vectors, measured along the direction vector.
5. **Visualize**: The margin boundaries are parallel to the decision boundary and pass through the support vectors (refer to figure 3.9).

3.10.2 Why Is This Important?

Maximizing the margin leads to better generalization and robustness against noise. This principle is the foundation of **Support Vector Machines (SVMs)**, which explicitly seek the hyperplane with the largest margin. While SVMs use more advanced optimization techniques, the geometric intuition shown here provides a glimpse into their core idea.

3.10.3 The Role of Normalization

Functional vs. Geometric Margin

The margin γ in our definition is the **functional margin**. To obtain the **geometric margin** (actual distance), we need normalization:

Definition 4 (Geometric Margin)**.** *The geometric margin is the actual perpendicular distance from training examples to the decision boundary:*

$$\gamma_{geom} = \frac{\gamma}{||\mathbf{w}^*||_2}$$

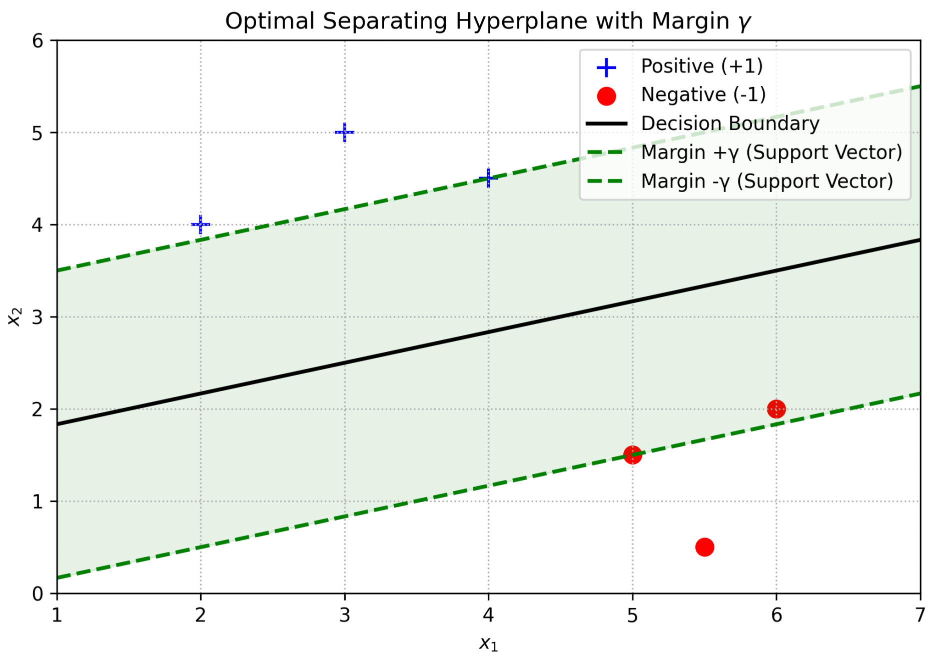

Figure 3.9: Optimal margin computation: The decision boundary is placed halfway between the closest points (support vectors) of each class, maximizing the margin.

Canonical Form

We can always normalize $\mathbf{w}^*$ such that the functional margin equals 1:

$$\tilde{\mathbf{w}}^* = \frac{\mathbf{w}^*}{\gamma}, \quad \text{giving} \quad y_i(\tilde{\mathbf{w}}^{*T}\mathbf{x}_i) \geq 1 \quad \forall i$$

3.10.4 Existence and Uniqueness

Existence Condition

Theorem 2 (Existence of Optimal Separator). *For a linearly separable dataset, there always exists at least one weight vector $\mathbf{w}^*$ with margin $\gamma > 0$ that correctly classifies all training examples.*

Proof Sketch. If the dataset is linearly separable, then there exists some $\mathbf{w}_0$ such that $y_i(\mathbf{w}_0^T\mathbf{x}_i) > 0$ for all i. Since there are finitely many training examples, we can define:

$$\gamma = \min_{i=1}^{N} y_i(\mathbf{w}_0^T\mathbf{x}_i) > 0$$

Thus, $\mathbf{w}^* = \mathbf{w}_0$ satisfies our condition. □

Non-Uniqueness

The optimal weight vector $\mathbf{w}^*$ is generally **not unique**:

- Any positive scalar multiple $c\mathbf{w}^*$ (where $c > 0$) also satisfies the condition with margin $c\gamma$
- Different weight vectors may achieve different margins
- The perceptron algorithm finds *any* separating hyperplane, not necessarily the optimal one

3.10.5 Connection to Perceptron Convergence

The Convergence Theorem Framework

The existence of $\mathbf{w}^*$ with margin γ is crucial for proving perceptron convergence as we ssaw from the perceptron convergence theorem:

3.10.6 Practical Implications

Algorithm Design

Understanding the optimal separator helps in:

- **Convergence Analysis**: Predicting how long training will take
- **Learning Rate Selection**: Choosing appropriate step sizes
- **Initialization Strategies**: Starting with reasonable weight vectors

Dataset Assessment

The margin γ provides insights about the dataset:

- **Large** γ: Easy problem, fast convergence expected
- **Small** γ: Difficult problem, slow convergence, sensitive to noise
- **No** $\gamma > 0$: Dataset is not linearly separable

3.10.7 Example: Optimal Separator for AND Function

Let's find the optimal weight vector for the AND function with margin analysis.

Training Data

Using the augmented representation (bias trick):

$$X = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}, \quad y = \begin{pmatrix} -1 \\ -1 \\ -1 \\ +1 \end{pmatrix}$$

Finding the Optimal Separator

We need to find $\mathbf{w}^* = [w_0, w_1, w_2]^T$ such that:

$$y_1(\mathbf{w}^{*T}\mathbf{x}_1) = -1(w_0) \geq \gamma \tag{3.12}$$
$$y_2(\mathbf{w}^{*T}\mathbf{x}_2) = -1(w_0 + w_2) \geq \gamma \tag{3.13}$$
$$y_3(\mathbf{w}^{*T}\mathbf{x}_3) = -1(w_0 + w_1) \geq \gamma \tag{3.14}$$
$$y_4(\mathbf{w}^{*T}\mathbf{x}_4) = +1(w_0 + w_1 + w_2) \geq \gamma \tag{3.15}$$

Solution with Maximum Margin

To find the maximum margin solution, we solve:

$$-w_0 \geq \gamma \tag{3.16}$$
$$-w_0 - w_2 \geq \gamma \tag{3.17}$$
$$-w_0 - w_1 \geq \gamma \tag{3.18}$$
$$w_0 + w_1 + w_2 \geq \gamma \tag{3.19}$$

Setting all inequalities to equality for the maximum margin:

$$w_0 = -\gamma, \quad w_1 = w_2 = 0, \quad \text{but this violates the fourth constraint}$$

The actual maximum margin solution requires solving a quadratic programming problem, yielding approximately:

$$\mathbf{w}^* = [-0.5, 0.5, 0.5]^T \quad \text{with} \quad \gamma = 0.5$$

This gives us the decision boundary $-0.5 + 0.5x_1 + 0.5x_2 = 0$, or equivalently $x_1 + x_2 = 1$.

3.10.8 Learning Rate Analysis

Effect of Learning Rate η

- **Large η**: Faster convergence but may overshoot optimal solution
- **Small η**: More stable learning but slower convergence
- **Theoretical Result**: For linearly separable data, any $\eta > 0$ guarantees convergence

Adaptive Learning Rates

Common strategies include:

- **Time decay**: $\eta_t = \frac{\eta_0}{1+\alpha t}$
- **Step decay**: Reduce η by factor every few epochs
- **Performance-based**: Reduce η when performance plateaus

3.11 Vectorisation of Perceptron Learning Rule: A Matrix-Based Example

To see how vectorization works in practice, let's walk through the process using a full matrix-based approach for the AND function. This method calculates the updates for all samples in the dataset (a "batch") and then applies a single, consolidated update at the end of the epoch.

3.11.1 Mathematical Foundation of Vectorization

Vectorization allows us to process multiple training examples simultaneously using matrix operations instead of iterating through samples one by one. This approach is:

- **Computationally efficient**: Modern hardware (GPUs) excels at matrix operations
- **Mathematically elegant**: Compact representation of batch operations
- **Numerically stable**: Reduces accumulated floating-point errors

3.11.2 Define the Matrices

For the AND function, we use an augmented Input Matrix X, a Weight Vector W, and a Target Vector T.

Input Matrix X (Augmented Design Matrix)

The input matrix includes a bias column (first column of 1s) followed by the feature columns:

$$X = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}_{4\times 3}$$

where:

- Rows represent training examples (4 samples)
- First column represents bias input (always 1)
- Remaining columns represent feature inputs x_1, x_2

Target Vector T

$$T = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}_{4\times 1}$$

Weight Vector W

Let's initialize the weight vector W to zeros and use a learning rate $\eta = 0.1$.

$$W_0 = \begin{pmatrix} w_0 \\ w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}_{3\times 1}$$

where w_0 is the bias weight, w_1 and w_2 are feature weights.

3.11.3 Epoch 1: Mathematical Flow

Step 1: Compute Net Input Z

The net input is computed as $Z = X \cdot W$, where we multiply the 4×3 input matrix by the 3×1 weight vector to get a 4×1 output vector.

$$Z = X \cdot W_0 = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}_{4\times 3} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}_{3\times 1} = \begin{pmatrix} 1\cdot 0 + 0\cdot 0 + 0\cdot 0 \\ 1\cdot 0 + 0\cdot 0 + 1\cdot 0 \\ 1\cdot 0 + 1\cdot 0 + 0\cdot 0 \\ 1\cdot 0 + 1\cdot 0 + 1\cdot 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}_{4\times 1}$$

Matrix Dimension Check: $(4 \times 3) \times (3 \times 1) = (4 \times 1)$

Step 2: Apply Activation Function to get Output Y

Apply the Heaviside step function element-wise: $\phi(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$

$$Y = \phi(Z) = \phi \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}_{4\times 1} = \begin{pmatrix} \phi(0) \\ \phi(0) \\ \phi(0) \\ \phi(0) \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}_{4\times 1}$$

Since all net inputs are 0, and $\phi(0) = 1$ by our step function definition, all outputs are 1.

Step 3: Calculate the Error Vector E

$$E = T - Y = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} - \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} -1 \\ -1 \\ -1 \\ 0 \end{pmatrix}$$

Step 4: Calculate the Total Weight Update ΔW

The weight update is computed as $\Delta W = \eta \cdot (X^T \cdot E)$, where we multiply the transpose of the input matrix by the error vector.

First, let's compute X^T:

$$X^T = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}^T = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}_{3\times 4}$$

Now compute the weight update:

$$\Delta W = \eta \cdot (X^T \cdot E) = 0.1 \cdot \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}_{3\times 4} \begin{pmatrix} -1 \\ -1 \\ -1 \\ 0 \end{pmatrix}_{4\times 1}$$

$$= 0.1 \cdot \begin{pmatrix} 1(-1)+1(-1)+1(-1)+1(0) \\ 0(-1)+0(-1)+1(-1)+1(0) \\ 0(-1)+1(-1)+0(-1)+1(0) \end{pmatrix} = 0.1 \cdot \begin{pmatrix} -3 \\ -1 \\ -1 \end{pmatrix} = \begin{pmatrix} -0.3 \\ -0.1 \\ -0.1 \end{pmatrix}_{3\times 1}$$

Matrix Dimension Check: $(3 \times 4) \times (4 \times 1) = (3 \times 1)$

Step 5: Update the Weight Vector W

$$W_1 = W_0 + \Delta W = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} -0.3 \\ -0.1 \\ -0.1 \end{pmatrix} = \begin{pmatrix} -0.3 \\ -0.1 \\ -0.1 \end{pmatrix}$$

After the first epoch, our new weight vector is $W_1 = (-0.3, -0.1, -0.1)^T$.

Colab Notebook: Click here to open

3.12 Limitations of Linear Classifiers

3.12.1 Linear Separability

Definition 5 (Linear Separability)**.** *A dataset is linearly separable if there exists a hyperplane that can separate all positive examples from all negative examples without any misclassification.*

3.12.2 The XOR Problem

XOR Truth Table: $(0,0) \to 0$, $(0,1) \to 1$, $(1,0) \to 1$, $(1,1) \to 0$

Why XOR is not linearly separable:

- Positive examples: $(0,1)$ and $(1,0)$
- Negative examples: $(0,0)$ and $(1,1)$
- No single line can separate these points correctly

Mathematical Proof of Non-Linear Separability: Assume there exists a linear classifier $\mathbf{w}^T\mathbf{x} + b = 0$ that separates XOR data. For the four points, we need:

$$w_1 \cdot 0 + w_2 \cdot 0 + b < 0 \quad \text{(point (0,0))} \tag{3.20}$$
$$w_1 \cdot 0 + w_2 \cdot 1 + b > 0 \quad \text{(point (0,1))} \tag{3.21}$$
$$w_1 \cdot 1 + w_2 \cdot 0 + b > 0 \quad \text{(point (1,0))} \tag{3.22}$$
$$w_1 \cdot 1 + w_2 \cdot 1 + b < 0 \quad \text{(point (1,1))} \tag{3.23}$$

From equations (1) and (2): $b < 0$ and $w_2 + b > 0 \Rightarrow w_2 > -b > 0$ From equations (1) and (3): $b < 0$ and $w_1 + b > 0 \Rightarrow w_1 > -b > 0$ From equation (4): $w_1 + w_2 + b < 0$

But this contradicts $w_1 > -b$ and $w_2 > -b$, since:

$$w_1 + w_2 + b > -b + (-b) + b = -b > 0$$

Therefore, no linear separator exists for XOR.

3.12.3 Convexity and Separability

Understanding convexity is crucial for analyzing when linear classifiers can and cannot work. Let's build this concept from the ground up.

Definition 6 (Convex Set)**.** *A set S is convex if the line segment connecting any two points in S lies entirely within S. Mathematically, for any two points $\mathbf{x}_1, \mathbf{x}_2 \in S$ and any $\lambda \in [0, 1]$:*

$$\lambda \mathbf{x}_1 + (1 - \lambda)\mathbf{x}_2 \in S$$

Interactive Python Script: Click here to open

Intuitive Examples:

- **Convex**: A circle, square, triangle, or any half-space
- **Not Convex**: A crescent moon shape, or an L-shaped region

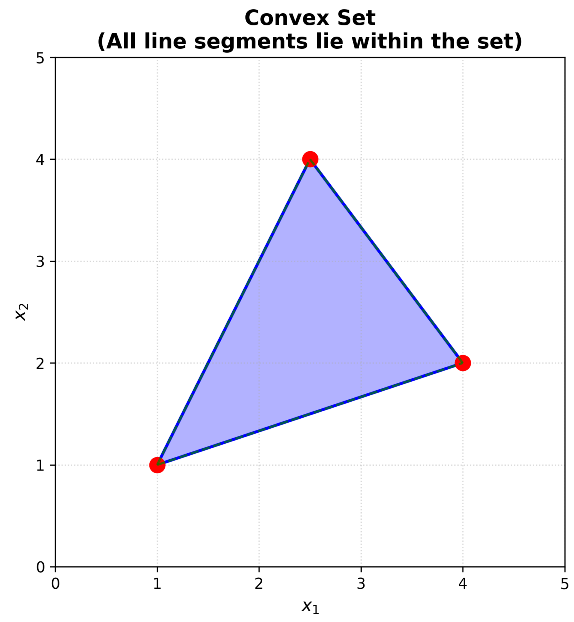

Figure 3.10: Example of a convex set. Any line segment connecting two points in the set lies entirely within the set.

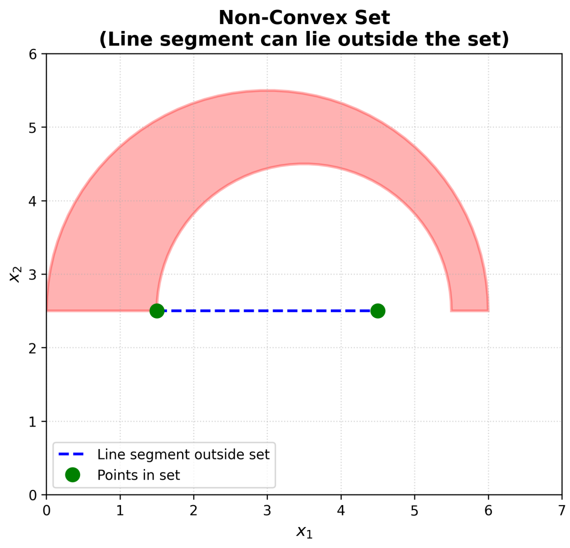

Figure 3.11: Example of a non-convex set. The line segment connecting two points exits the set, violating the convexity condition.

Weighted Averages in Convex Sets

An important property of convex sets is that any weighted average of points in the set must also lie within the set.

Definition 7 (Weighted Average)**.** *A weighted average of points $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}$ is given by:*

$$\mathbf{x}_{avg} = \lambda_1 \mathbf{x}^{(1)} + \lambda_2 \mathbf{x}^{(2)} + \cdots + \lambda_N \mathbf{x}^{(N)}$$

where the weights satisfy $0 \leq \lambda_i \leq 1$ and $\sum_{i=1}^{N} \lambda_i = 1$.

Physical Interpretation: Think of the weighted average as the center of mass, where each point has mass λ_i.

Theorem 3 (Weighted Averages in Convex Sets)**.** *If S is a convex set, then any weighted average of points in S must also lie within S.*

Convex Hull: The convex hull of a set of points is the smallest convex set containing all those points. Any weighted average of the points will lie within this convex hull.

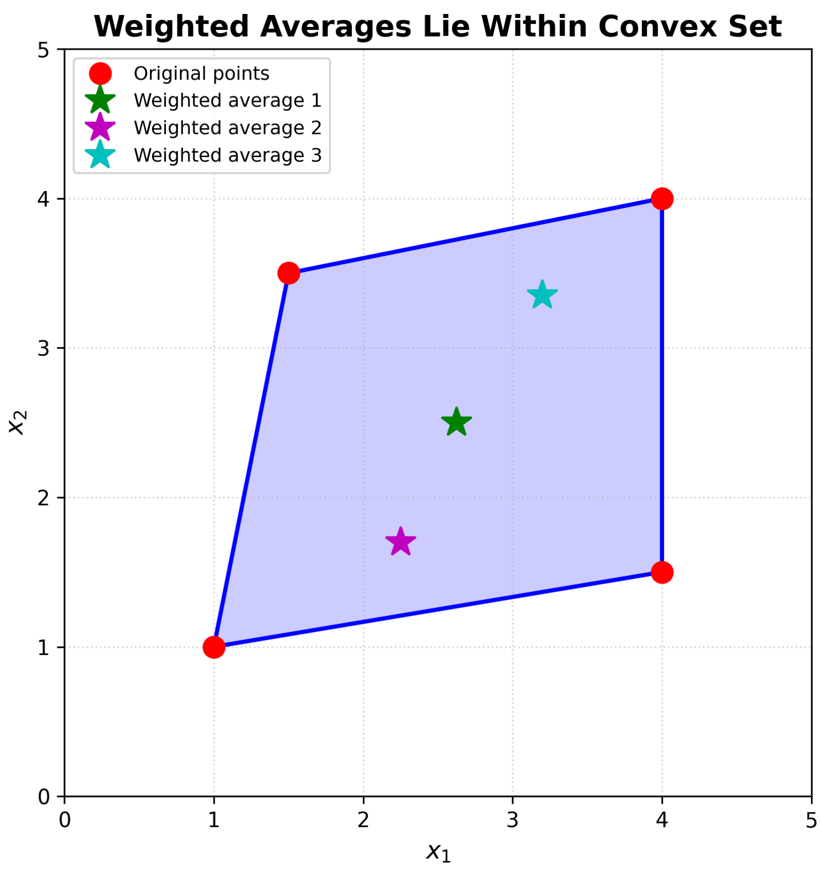

Figure 3.12: Weighted average in a convex set. The weighted average (center of mass) of points always lies within the convex hull.

Convexity in Binary Classification

In binary classification, convexity plays a crucial role in two distinct spaces:

1. **Data Space (Input Space)**:

   - The positive region (where $\mathbf{w}^T\mathbf{x} + b > 0$) is a half-space, which is convex
   - The negative region (where $\mathbf{w}^T\mathbf{x} + b < 0$) is also a half-space, which is convex
   - **Implication**: If inputs $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}$ are all classified as positive, then any weighted average of these inputs must also be classified as positive (and similarly for negative examples)

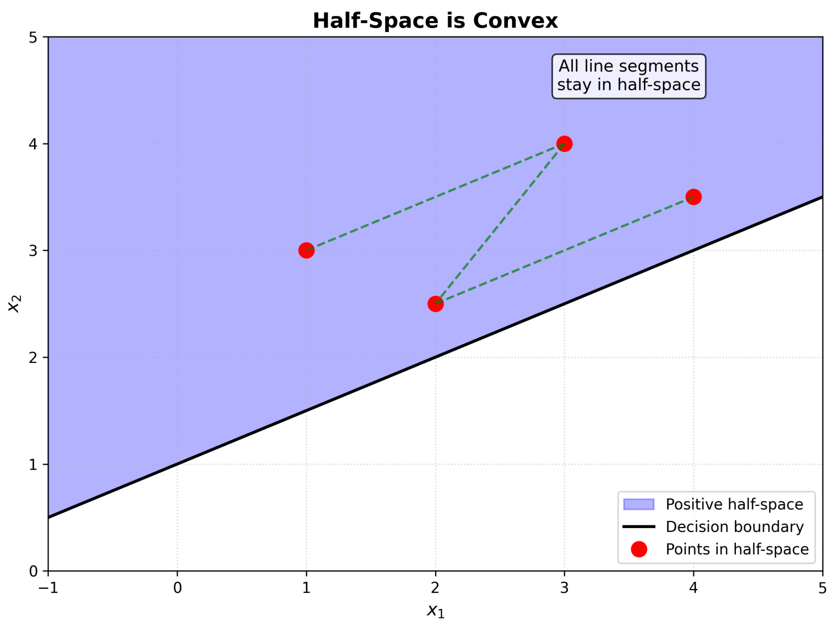

Figure 3.13: Half-spaces created by a linear classifier are convex. The positive and negative regions are separated by the decision boundary (hyperplane).

2. **Weight Space**:
   - Each training example $(\mathbf{x}^{(i)}, t^{(i)})$ defines a "good region" of weights that correctly classify it
   - Each good region is a half-space in weight space, hence convex
   - The feasible region (weights that correctly classify *all* examples) is the intersection of all good regions
   - **Key Result**: Since the intersection of convex sets is convex, the feasible region is convex

Theorem 4 (Linear Separability and Convexity)**.** *If the positive examples form a convex set and the negative examples form a convex set, and these sets do not overlap, then the data is linearly separable.*

Contrapositive (useful for proving non-separability): If we can show that either the positive or negative class does not form a convex set (or they overlap), we can use this to prove linear non-separability.

Alternative Proof Using Convexity: XOR Example

Let's revisit the XOR problem using convexity arguments, which provides elegant geometric intuition.

XOR Dataset Reminder:

- Positive examples (output = 1): (0,1) and (1,0)
- Negative examples (output = 0): (0,0) and (1,1)

Proof by Convexity Contradiction:

Since a linear classifier creates convex positive and negative regions (half-spaces), we can use this property to prove XOR is not linearly separable:

1. **Assume** there exists a linear classifier that correctly classifies all XOR examples.
2. **For the positive region**: Since (0,1) and (1,0) are both positive examples, they must lie in the positive region (a convex half-space). Therefore, the entire line segment connecting them must also lie in the positive region.
3. **For the negative region**: Since (0,0) and (1,1) are both negative examples, they must lie in the negative region (a convex half-space). Therefore, the entire line segment connecting them must also lie in the negative region.

4. **Finding the contradiction**:

   - Line segment from (0,1) to (1,0): All points of the form $\lambda(0,1) + (1-\lambda)(1,0) = (1-\lambda, \lambda)$ for $\lambda \in [0,1]$
   - Line segment from (0,0) to (1,1): All points of the form $\mu(0,0) + (1-\mu)(1,1) = (1-\mu, 1-\mu)$ for $\mu \in [0,1]$
   - These segments intersect when $(1-\lambda, \lambda) = (1-\mu, 1-\mu)$
   - Solving: $\lambda = 1-\mu$ and $1-\lambda = 1-\mu$ gives $\lambda = \mu = 0.5$
   - Intersection point: $(0.5, 0.5)$

5. **The contradiction**: The point (0.5, 0.5) must be in both the positive region (by step 2) and the negative region (by step 3), which is impossible.

6. **Conclusion**: No linear classifier can correctly classify XOR. The dataset is not linearly separable.

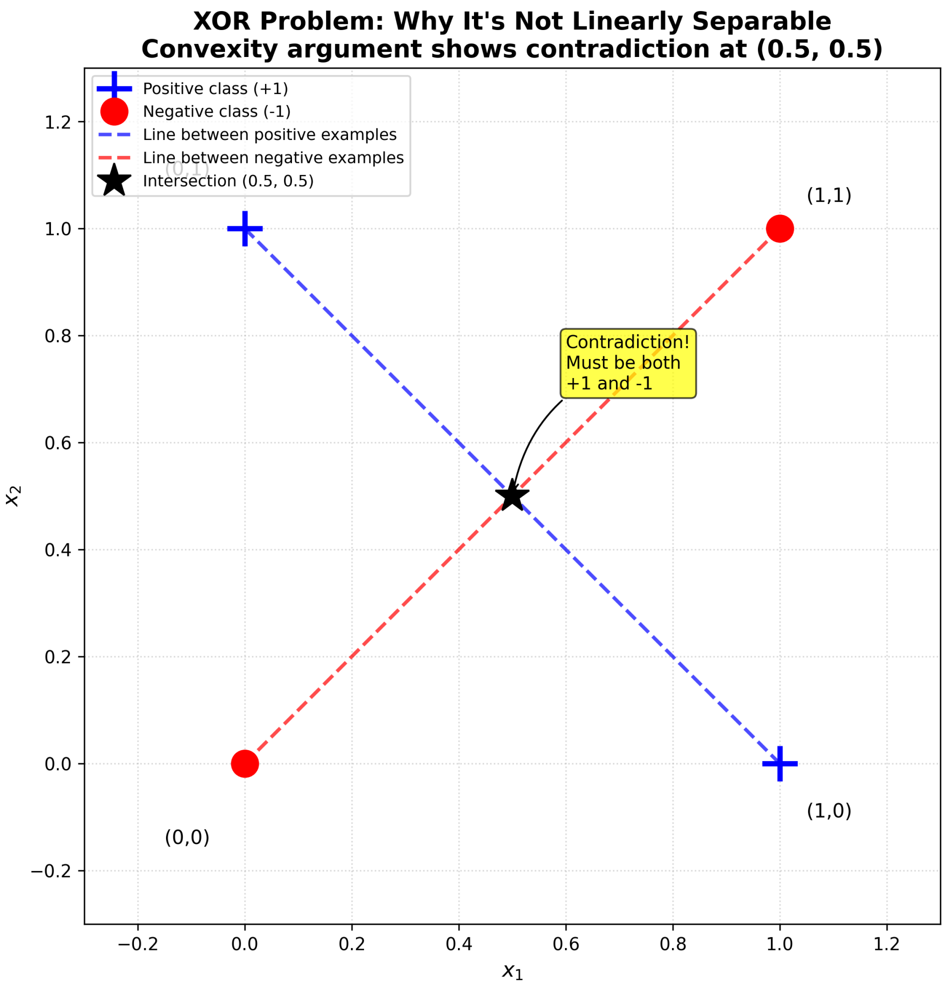

Figure 3.14: XOR Problem: Violation of convexity requirement. The line segments connecting points of the same class intersect at (0.5, 0.5), demonstrating that neither class forms a convex set. This intersection proves why XOR cannot be linearly separated.

Why This Proof is Elegant: Instead of algebraically trying all possible weight combinations (which we did earlier), the convexity argument gives us a geometric guarantee: since half-spaces are convex, any linear classifier would force the line segments to be classified uniformly, but they intersect, creating an impossibility.

3.12.4 A Practical Limitation: Translation Invariance

While XOR is a useful theoretical example, let's examine a more practical and troubling limitation of linear classifiers that affects real-world applications like computer vision.

The Pattern Recognition Problem

Scenario: Imagine building a vision system for a robot that needs to recognize objects regardless of their position in its visual field. The robot should identify a pattern regardless of where it appears (translation invariance).

Simplified Example: Consider 16-dimensional binary input vectors (each input is a vector in $\mathbb{R}^{16}$). For visualization, we can think of these 16 values as arranged in a 4×4 grid, but the actual input to the classifier is a flat 16-dimensional vector. We want to distinguish two patterns, A and B (as shown in Figure 3.15), which can appear in any of the 16 possible translations (with wrap-around: if you shift right, whatever falls off the right reappears on the left).

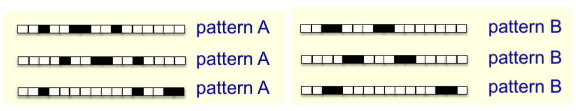

Figure 3.15: Two simple patterns A and B visualized as a 4×4 grid. Each pattern is actually a 16-dimensional binary vector that needs to be distinguished regardless of translation position.

The Challenge: We have 16 examples of pattern A (one for each translation) and 16 examples of pattern B (one for each translation), and our classifier needs to distinguish between them.

Convexity Argument for Non-Separability

Let's use convexity to prove this task is impossible for a linear classifier:

1. **Pattern Structure**: Suppose pattern A has 4 pixels "on" out of 16 total pixels (and similarly for pattern B).
2. **Computing the Average**:
   - For the classifier to correctly classify all 16 translations of pattern A, it must classify each one as positive
   - By convexity, if all 16 instances are classified as A, their weighted average must also be classified as A
   - The average of all 16 translations: Each pixel is "on" in exactly 4 of the 16 translations
   - Average vector for A: $(0.25, 0.25, 0.25, \ldots, 0.25)$ (each of 16 components equals 4/16)
3. **Same for Pattern B**:
   - For pattern B to be correctly classified in all translations, each must be classified as B
   - By convexity, their average must also be classified as B
   - Since pattern B also has 4 pixels "on", its average is also $(0.25, 0.25, 0.25, \ldots, 0.25)$
4. **The Contradiction**: The same vector $(0.25, 0.25, \ldots, 0.25)$ cannot be classified as both A and B. Therefore, this dataset is not linearly separable.

General Implication: *Linear classifiers cannot reliably detect a pattern in all possible translations.* This is a fundamental limitation for vision systems based purely on linear classification, as translation invariance is a critical requirement.

Why This Matters: This example demonstrates that even seemingly simple real-world tasks (recognizing a pattern regardless of position) can be beyond the capability of linear classifiers, motivating the need for more sophisticated approaches like convolutional neural networks that build in translation invariance.

3.12.5 Overcoming Limitations: Basis Functions

We've seen the limitations of linear classifiers. Now let's explore a powerful technique to overcome these limitations: **feature representation using basis functions**.

The Core Idea

Strategy: Instead of working directly with the input features, transform them using a function $\boldsymbol{\phi}(\mathbf{x})$ that maps inputs to a new feature space where linear separation becomes possible.

Mathematical Framework:

- **Original classifier**: $z = \mathbf{w}^T\mathbf{x} + b$
- **Feature-based classifier**: $z = \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}) + b$

where $\boldsymbol{\phi}(\mathbf{x}) = (\phi_1(\mathbf{x}), \phi_2(\mathbf{x}), \ldots, \phi_D(\mathbf{x}))$ is a function mapping input vectors to feature vectors.

Detailed Example: Solving XOR with Basis Functions

Let's see how carefully selected features can make XOR linearly separable.

Step 1: Design Feature Representation

For XOR with inputs x_1, x_2, consider the following feature transformation:

$$\phi_1(\mathbf{x}) = x_1 \tag{3.24}$$
$$\phi_2(\mathbf{x}) = x_2 \tag{3.25}$$
$$\phi_3(\mathbf{x}) = x_1 \cdot x_2 \quad \text{(interaction term)} \tag{3.26}$$

Step 2: Transform the Training Set

The original XOR data becomes:

x_1	x_2	$\phi_1(x)$	$\phi_2(x)$	$\phi_3(x)$	t (target)
0	0	0	0	0	0
0	1	0	1	0	1
1	0	1	0	0	1
1	1	1	1	1	0

Step 3: Find Linear Separator in Feature Space

Now we need to find weights w_1, w_2, w_3 and bias b such that the classifier $z = w_1\phi_1 + w_2\phi_2 + w_3\phi_3 + b$ correctly classifies all examples.

Applying the same constraint-based approach from earlier examples:

$$\text{For } (0,0,0) \to 0: \quad b < 0 \tag{3.27}$$
$$\text{For } (0,1,0) \to 1: \quad w_2 + b > 0 \tag{3.28}$$
$$\text{For } (1,0,0) \to 1: \quad w_1 + b > 0 \tag{3.29}$$
$$\text{For } (1,1,1) \to 0: \quad w_1 + w_2 + w_3 + b < 0 \tag{3.30}$$

Step 4: Solve for Weights

One valid solution is:

$$b = -0.5 \tag{3.31}$$
$$w_1 = 1 \tag{3.32}$$
$$w_2 = 1 \tag{3.33}$$
$$w_3 = -2 \tag{3.34}$$

Verification:

- $(0,0,0)$: $z = 0 + 0 + 0 - 0.5 = -0.5 < 0$ ✓
- $(0,1,0)$: $z = 0 + 1 + 0 - 0.5 = 0.5 > 0$ ✓
- $(1,0,0)$: $z = 1 + 0 + 0 - 0.5 = 0.5 > 0$ ✓

- $(1,1,1)$: $z = 1 + 1 - 2 - 0.5 = -0.5 < 0$ ✓

Step 5: Decision Rule
The final classifier in the original input space becomes:

$$\hat{y} = \begin{cases} 1 & \text{if } x_1 + x_2 - 2x_1x_2 - 0.5 > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.35}$$

The Challenge: Feature Engineering

The Problem: Where do we get the features from? In the XOR example, we "pulled them out of a hat." Unfortunately, there's no universal recipe for designing good features, which is part of what makes machine learning challenging.

Historical Context: For decades, feature engineering was the primary bottleneck in machine learning. Domain experts would spend considerable time designing hand-crafted features for each problem.

Modern Solution: Multi-layer neural networks can *learn* a set of features automatically through training. Instead of manually designing $\boldsymbol{\phi}(\mathbf{x})$, deep learning discovers useful representations from data. This is one of the key breakthroughs that enabled the modern AI revolution.

3.13 Practical Considerations

3.13.1 Feature Engineering

- **Normalization**: Scale features to similar ranges
- **Interaction terms**: Add products of features (x_1x_2)
- **Polynomial features**: Add powers of features (x_1^2, x_2^2)

3.13.2 Model Selection

- **Bias-variance tradeoff**: Simple models (few features) vs. complex models (many features)
- **Interpretability**: Linear classifiers provide clear feature importance through weights
- **Computational efficiency**: Linear models are fast to train and evaluate

3.13.3 Performance Evaluation

- **Training accuracy**: Percentage of training examples correctly classified
- **Generalization**: Performance on unseen test data
- **Decision boundary visualization**: Plot boundaries in 2D for intuition

3.14 Solutions to Linear Classifier Limitations

- **Multi-Layer Perceptrons (MLPs)**:
  - Add hidden layers with non-linear activation functions
  - Can approximate any continuous function (Universal Approximation Theorem)
  - Require more sophisticated training algorithms (backpropagation)
- **Feature Engineering**:
  - Transform input space to make data linearly separable
  - Kernel methods: Implicitly map to higher-dimensional spaces
- **Ensemble Methods**:
  - Combine multiple linear classifiers
  - Voting or weighted combination schemes
  - Can learn non-linear decision boundaries

3.15 Historical Impact and Legacy

The 1969 book "Perceptrons" by Minsky and Papert highlighted the limitations of single-layer perceptrons, leading to:

- **AI Winter**: Reduced funding and interest in neural networks
- **Focus shift**: Emphasis moved to symbolic AI and expert systems
- **Delayed progress**: Multi-layer networks existed but lacked efficient training methods

3.15.1 Modern Relevance

Despite limitations, perceptrons remain important because:

- **Building blocks**: Neurons in modern deep networks are perceptron variants
- **Theoretical foundation**: Understanding linear classifiers is crucial
- **Computational efficiency**: Still useful for linearly separable problems
- **Online learning**: Perceptron learning rule works in streaming settings

3.16 Exercises

1. Implement the perceptron learning algorithm from scratch and test it on the AND function. Plot the decision boundary after training.
2. Prove that the perceptron learning algorithm converges for linearly separable data.
3. Use the perceptron learning rule to classify a small dataset of your choice. Experiment with different learning rates and observe their effects on convergence.
4. Show that the XOR problem is not linearly separable using both algebraic and convexity arguments.
5. Design a feature transformation that makes the XOR problem linearly separable. Verify your solution by finding appropriate weights and bias.
6. Explore the effect of adding polynomial features (e.g., x_1^2, x_2^2) on the linear separability of a non-linear dataset.
7. Implement a multi-layer perceptron (MLP) and train it on the XOR problem. Compare its performance to a single-layer perceptron.
8. Investigate how normalization of input features affects the training speed and convergence of the perceptron learning algorithm.
9. Create a dataset that is not linearly separable due to translation invariance issues. Attempt to classify it using a linear classifier and explain why it fails.
10. Research and summarize one modern application where linear classifiers are still effectively used today.
11. Discuss the limitations of linear classifiers in high-dimensional spaces and potential strategies to mitigate these issues.
12. Implement the perceptron learning algorithm with a margin (i.e., a margin perceptron) and test it on a linearly separable dataset. Compare its performance to the standard perceptron.