

ANCILLARY MATERIALS FOR THE SHALLOW AND THE DEEP BY M. BIEHL



Hei-Chi Chan

University of Illinois Springfield

University of Illinois Springfield

Ancillary Materials for The Shallow and the Deep
by M. Biehl

Hei-Chi Chan

This text is disseminated via the Open Education Resource (OER) LibreTexts Project (<https://LibreTexts.org>) and like the thousands of other texts available within this powerful platform, it is freely available for reading, printing, and "consuming."

The LibreTexts mission is to bring together students, faculty, and scholars in a collaborative effort to provide an accessible, and comprehensive platform that empowers our community to develop, curate, adapt, and adopt openly licensed resources and technologies; through these efforts we can reduce the financial burden born from traditional educational resource costs, ensuring education is more accessible for students and communities worldwide.

Most, but not all, pages in the library have licenses that may allow individuals to make changes, save, and print this book. Carefully consult the applicable license(s) before pursuing such effects. Instructors can adopt existing LibreTexts texts or Remix them to quickly build course-specific resources to meet the needs of their students. Unlike traditional textbooks, LibreTexts' web based origins allow powerful integration of advanced features and new technologies to support learning.



LibreTexts is the adaptable, user-friendly non-profit open education resource platform that educators trust for creating, customizing, and sharing accessible, interactive textbooks, adaptive homework, and ancillary materials. We collaborate with individuals and organizations to champion open education initiatives, support institutional publishing programs, drive curriculum development projects, and more.

The LibreTexts libraries are Powered by [NICE CXone Expert](#) and was supported by the Department of Education Open Textbook Pilot Project, the California Education Learning Lab, the UC Davis Office of the Provost, the UC Davis Library, the California State University Affordable Learning Solutions Program, and Merlot. This material is based upon work supported by the National Science Foundation under Grant No. 1246120, 1525057, and 1413739.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation nor the US Department of Education.

Have questions or comments? For information about adoptions or adaptations contact info@LibreTexts.org or visit our main website at <https://LibreTexts.org>.

The LibreTexts name, logo, and book covers are trademarked by LibreTexts, Inc. (a 501(c)(3) not-for-profit organization) and protected against unauthorized use.

This text was compiled on 08/23/2026

TABLE OF CONTENTS

Acknowledgement

Licensing

1: From Neurons to Networks

- 1.1: Chapter 1 Summary
- 1.2: A Brief History of Neural Networks and Deep Learning
- 1.3: Hebbian Learning
- 1.4: The Hopfield Model
- 1.5: Modern Hopfield Networks
- 1.6: A Brief Discussion

2: Learning from Data

- 2.1: Chapter 2 Summary
- 2.2: Types of Learning
- 2.3: A List of Supervised Learning Models
- 2.4: Annoy from Spotify
- 2.5: Distance Concentration
- 2.6: Shannon's Source Coding Theorem

3: The Perceptron & Beyond Linear Separability

- 3.1: Chapters 3 and 4 Summary
- 3.2: A Brief Introduction on Rademacher Complexity

4: Beyond Linear Separability

5: Feed-forward Networks for Regression and Classification

- 5.1: Chapter 5 Summary
- 5.2: Universal Approximation Theorem
- 5.3: Attention Mechanism/Transformers

6: Distance-based Classifiers

- 6.1: Chapter 6 Summary and a Discussion on the Neural Gas Model

7: Model Evaluation and Regularization

- 7.1: Chapter 7 Summary
- 7.2: The Bias-Variance Trade Off
- 7.3: Explainable AI (XAI)

8: Preprocessing and Unsupervised Learning

- 8.1: Chapter 8 Summary
- 8.2: Dimensionality Reduction Methods
- 8.3: Clustering Methods
- 8.4: Vector DBs & Similarity Search at Scale (The Big Picture)

Index

Glossary

Detailed Licensing

This page titled [Ancillary Materials for The Shallow and the Deep by M. Biehl](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by Hei-Chi Chan.

Acknowledgement

The following ancillary materials supplement the Open Textbook

Biehl, M. (2023). *The Shallow and the Deep*. University of Groningen Press.

Throughout this work, we refer to Biehl's text as **SD**.

We would like to thank the Illinois Secretary of State and the Illinois State Library for supporting the development of these open educational resources through the Open Educational Resources (OER) Grant Program. The funding is administered in collaboration with the Consortium of Academic and Research Libraries in Illinois (CARLI). This project was made possible by funding awarded to the University of Illinois Springfield (UIS) Brookens Library.

Licensing

A detailed breakdown of this resource's licensing can be found in **Back Matter/Detailed Licensing**.

1: FROM NEURONS TO NETWORKS

1.1: Chapter 1 Summary

1.2: A Brief History of Neural Networks and Deep Learning

1.3: Hebbian Learning

1.4: The Hopfield Model

1.5: Modern Hopfield Networks

1.6: A Brief Discussion

This page titled [1: From Neurons to Networks](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by Hei-Chi Chan.

1.1: Chapter 1 Summary

Building on the idea of understanding the brain through artificial neural networks (ANNs), here are more details about these concepts and architectures.

- **Biological Inspiration and Motivation:**

- The goal: Understand and explain the brain's (fascinating!) capabilities, particularly its ability to learn, adapt, and survive in dynamic environments.
- ANNs and related systems are designed for parallel, adaptive information processing. Their designs are inspired by biological neurons and networks.
- This approach helps to model biological systems at a basic level and provides insights into the principles underlying biological intelligence.

- **Spiking Neurons and Synaptic Interactions:**

- The physiology of biological systems is highly complex, even at the single neuron level.
- Electrochemical processes are fundamental to how neurons communicate. Neurons maintain an electrochemical potential difference across their cell membrane.
- An action potential, also known as a **spike**, is generated when the membrane potential reaches a specific threshold. This is a brief, localized electrical pulse that propagates along the axon from the cell body (soma).
- When a spike reaches the end of the axon, it triggers the release of neurotransmitters into the synaptic cleft. These chemical messengers then bind to receptors on the post-synaptic neuron.
- This binding leads to a change in the post-synaptic neuron's membrane potential (the post-synaptic potential), which can be either **excitatory** (increasing the potential and likelihood of spiking) or **inhibitory** (decreasing it).
- Synaptic interactions are generally unidirectional, meaning information flows from the pre-synaptic to the post-synaptic neuron. There is also a small but significant **synaptic delay** (around 0.5 ms) due to the electrochemical processes.
- Models like the **Hodgkin-Huxley model** use complex differential equations to describe these dynamics, while simpler models like the **Integrate-and-Fire (IaF) model** capture essential features for studying large networks.

- **Firing Rate Models and Activation Functions:**

- To simplify the complexity of individual spikes, firing rate models represent neural activity as the average number of spikes per unit time.
- These models assume that the crucial information for neural processing is contained within the mean activity and frequency of spikes.
- The activity of a neuron i (denoted S_i) is determined by the weighted sum of inputs from other neurons j (S_j) passed through an **activation function** $h(x)$:

$$S = h \left(\sum_j w_{ij} S_j \right). \quad (1.1.1)$$

- Synaptic weights (w_{ij}) represent the strength of the connection from neuron j to neuron i . A positive w_{ij} indicates an **excitatory** connection, while a negative w_{ij} indicates an **inhibitory** connection.
- Sigmoidal activation functions ($h(x)$ or $g(x)$) are commonly used because they capture key biological characteristics.

- The **McCulloch-Pitts model** is an early, simplified model of neurons.

- **Hebbian Learning:**

- Beyond fixed functionalities, biological neural networks can learn and adapt to their environment, a process known as **synaptic plasticity**.
- **Hebbian Learning** is a fundamental hypothesis for this learning, famously summarized as “neurons that fire together wire together”.
- Mathematically, the **change in synaptic weight** (Δw_{AB}) is proportional to the product of the pre- and post-synaptic activities (x_A and s_B):

$$\Delta w_{AB} \propto x_A s_B \quad (1.1.2)$$

- This implies that if both neurons are active, the synapse strengthens (excitatory increase), and if their activities are decorrelated, the synapse may weaken or become inhibitory.
 - Hebbian learning forms the basis for many learning algorithms in ANNs.
- This chapter also discusses models like the Hopfield Model and several types of ANNs. See the following subsections for details.

1.2: A Brief History of Neural Networks and Deep Learning

We provide a brief history of neural networks (NN) and deep learning (DL) up to 2017, the year “Attention Is All You Need” was published (which marks the beginning of large language models (LLMs)). We will have sections on LLMs.

- 1943: The McCulloch-Pitts model

First mathematical model of neural networks; foundational for the field.

- 1949: Hebb’s Rule

Introduced Hebbian learning: “neurons that fire together wire together.”

- 1958: The Perceptron model by Rosenblatt

First trainable artificial neural network (ANN) capable of binary classification.

- 1960: ADALINE/MADALINE (by Widrow & Hoff)

ADALINE = Adaptive Linear Neuron or Adaptive Linear Element) and MADALINE = Multiple ADALINE. Neural networks using least mean squares learning; early adaptive systems.

- 1960s: Minsky & Papert’s book, *Perceptrons*

Began studies on the limitations of perceptrons. 1969: Published *Perceptrons* (1st ed.), systematically proving the limitations of single-layer perceptrons — especially their inability to solve problems like XOR. A major factor leading to the first AI Winter. Note: An expanded edition was published in 1988 with a prologue and epilogue addressing criticisms and discussing the revival of neural nets. In 2017, the expanded edition was reprinted, including a new foreword discussing the book’s impact from the perspective of modern deep learning.

- 1960s–early 70s: Various developments

Group Method of Data Handling by Alexey Ivakhnenko and V. G. Lapa (1968). Multilayer perceptron (early deep learning) trained via stochastic gradient descent by Amari (1967–68).

- 1980: Neocognitron (Fukushima)

Early convolutional neural network for visual pattern recognition. Inspired by the work of Hubel and Wiesel (1950s and 60s), building on Fukushima’s earlier work (including the Cognitron). Recognized as the direct precursor to modern Convolutional Neural Networks (CNNs).

- 1982: Hopfield Network (Hopfield), or Amari-Hopfield Network

Recurrent network using energy minimization for pattern storage and retrieval. Amari (1972) discovered a network equivalent to the Hopfield network. (Note: Amari is credited as the founder of information geometry.) Hopfield was awarded the 2024 Nobel Prize in Physics (shared with Geoffrey Hinton).

- 1980s: Recurrent Neural Networks (RNNs)

Designed for processing sequential data where each output depends on previous inputs. Including the Jordan Network (1986) and Elman Network (1990).

- 1985–86: Restricted Boltzmann Machine (RBM; Hinton, Sejnowski, & Smolensky)

Stochastic generative model with bipartite structure; important for unsupervised learning. Hinton and coworkers trained it with fast learning algorithms, such as Contrastive Divergence. In the mid-2000s, RBMs became the building blocks for Deep Belief Networks (DBNs) and other deep learning architectures.

- 1986: Backpropagation (Rumelhart, Hinton, Williams)

Popularized practical training for multilayer perceptrons using backpropagation. Inspired by the work by Rosenblatt (1962: “back-propagating error correction”), Dreyfus (62, depending on Bryson and Kelley around the same time), Linnainmaa (1970), and Werbos (1974).

- 1989: LeNet (LeCun)

Applied CNNs to handwritten digit recognition; deployed in real-world applications. The concepts were developed by LeCun and its team at AT&T in the late 80s. Refined in the early 90s, and LeNet-5 was released in 98 (successful for digit recognition). In the late 90s, it was used in industry (notably for reading handwritten checks and postal codes in banks and ATMs).

- 1997: Long Short-Term Memory (LSTM, by Hochreiter & Schmidhuber)

LSTM (Long Short-Term Memory) Enabled learning long-term dependencies in sequences; a solution to vanishing gradients in RNNs. Each unit has a memory cell with controlled update mechanisms, and three gates (input, forget, and output gates), which are crucial for long-term information retention and gradient flow in deep networks.

Note also the Gated Recurrent Units (GRUs). They are RNN-type networks, introduced in 2014 by Cho et. al., to efficiently process sequential data by using two main gates (the reset gate and the update gate), allowing the networks to selectively forget and retain information at each time step, effectively solving the vanishing gradient problem and enabling the learning of both short-term and long-term dependencies in sequences.

- 2011: DanNet (Ciresan et al.)

First pure deep GPU-based CNN to win major international computer vision contests (IJCNN, ICDAR).

- 2012: AlexNet (Krizhevsky et al.)

It won ImageNet with a large margin; triggers the DL revolution.

- 2014: Seq2Seq with Attention (Bahdanau et al.)

Introduced the attention mechanism in sequence modeling; boosts translation and NLP tasks.

- 2015: ResNet (He et al.)

It uses residual connections (skip connections) to enable the training of very deep networks (100+ layers) by mitigating the vanishing gradient problem.

- 2017: Transformer (Vaswani et al.)

“Attention is All You Need”: Introduced Transformers, revolutionizing NLP. It marks the dawn of a new era in AI/ML. We will include sections on this and on subsequent developments inspired by it.

Further Exploration

1. See the following sources for full reference information.

Jürgen Schmidhuber. *Deep learning in neural networks: An overview*. *Neural Networks*, vol. 61, pp. 85–117, 2015.

Jürgen Schmidhuber. *Timeline of machine learning and deep learning research*. 2015–2022. IDSIA, Switzerland.

2. The **McCulloch-Pitts model** (1943)

- The first mathematical abstraction of a biological neuron, forming the foundation for NNs.
- Consider a single McCulloch-Pitts (MCP) neuron, with n inputs and one output.

- Each input $x_i \in \{0, 1\}$ (representing the state of the i -th neuron as firing and not firing)
- The weight $\{w_i\}$ represents the strength of the synapse connecting the x_i to the output. It can be positive or negative corresponding to an *excitatory* or *inhibitory* synapse resp.
- The **activation** is a **threshold function**: the neuron outputs 1 if the sum reaches or exceeds a specified **threshold** θ , otherwise outputs 0. Indeed, the output, y , is given by

$$y = H \left(\sum_{i=1}^n w_i x_i - \theta \right) \quad (1.2.1)$$

where H is the Heaviside function (or, step function). Later models use more general non-linear functions instead of H .

- **Summary of behaviour:**
 - The neuron *fires* (outputs 1) if activation $\geq \theta$.
 - Otherwise, the output is 0.
- **Limitations:**
 - Only handles binary inputs and outputs.
 - It can only model linearly separable functions and cannot learn from data.
- The McCulloch-Pitts model is an abstract, early model describing a neuron as a simple logic gate, without accounting for membrane dynamics or temporal features. In contrast, the Hodgkin-Huxley model (or, its simplified versions, such as the leaky integrate-and-fire (LIF) model) is a much more detailed and biophysically accurate representation, describing how action potentials are generated and propagated using nonlinear differential equations. For details, see, e.g.,
 - Ermentrout, G. B., & Terman, D. H. (2010). *Mathematical Foundations of Neuroscience*. New York: Springer.
 - Dayan, P., & Abbott, L. F. (2001). *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. Cambridge, MA: MIT Press.

1.3: Hebbian Learning

Key reference: Section 1.2 of SD.

Background

- Hebbian learning was introduced by Donald Hebb in 1949 in his book *The Organization of Behavior*.
- It proposes a mechanism explaining how synaptic connections between neurons strengthen during learning and memory formation, forming the foundational theory of synaptic plasticity.
- The theory is often summarized by the phrase: “Neurons that fire together, wire together”.

Key Concepts

- Synaptic Plasticity: The strength (efficacy) of synaptic connections can change, enabling learning and memory in neural circuits.
- Associative Learning: When activation of one neuron repeatedly contributes to activation of another, the connection strengthens, forming memory traces.
- Bidirectional Changes: While repeated co-activation strengthens connections, lack of joint activity can weaken them (synaptic weakening or depression).

Formulation

We can turn the above into a mathematical model.

- Consider a single neuron: inputs are given by x_i ($i = 1, 2, \dots, n$), and output is y (all variables are real numbers ≥ 0 ; when they are zero, it means there are no signals). They are related linearly:

$$y = \sum_i w_i x_i = \mathbf{w}^T \mathbf{x} \quad (1.3.1)$$

- where $\{w_i\}$ is the set of (synaptic) weights. In the second equality, we have used vector notation.
- The plain Hebbian learning (how weights are updated) is given by

$$\Delta w_i = \eta x_i y. \quad (1.3.2)$$

Here, η is the learning rate. This reflects the Hebbian learning: the weight increases most when both neurons are active simultaneously; if one or both are inactive (zero), weight change is zero. Simultaneous activity strengthens the synapse.

- Since Δw_i is always ≥ 0 , this leads to exponential, unbounded increase in synaptic weights, making networks unstable and biologically unrealistic.
- Modification: Oja's Rule

Oja proposes to add a damping term to the update rule:

$$\Delta w_i = \eta y (x_i - y w_i). \quad (1.3.3)$$

Oja shows that this is the infinitesimal version of demanding that the updated weight vector $\mathbf{w}$ always has unit norm.

Further Exploration

For more information, see Chapter 8 of Hertz, Krogh, and Palmer (1991), *Introduction to the Theory of Neural Computation* (Addison-Wesley).

This page titled [1.3: Hebbian Learning](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by Hei-Chi Chan.

1.4: The Hopfield Model

Key reference: Sect. 1.3 of SD.

Introduction to the Hopfield Model

The Hopfield model is a simple mathematical framework to describe **networks of neurons** that can store and recall memory patterns. Precisely, it is a model for **associative memory**: if given a noisy or partial version of a stored pattern, the network can recover the whole pattern. This is like a person, after hearing a few notes of a song, remembers the whole tune.

The Model

This is known as the classic/traditional Hopfield model (dated back to the 80s):

- Consider N stored patterns (memories) given by $\{\mathbf{x}_i\}_{i=1}^N$. Here, each $\mathbf{x}_i \in \{\pm 1\}^d$, with d being the length of the patterns. E.g., each $\mathbf{x}_i$ could be a picture, with d being the number pixels (in fact, since the value of each pixel is ± 1 , the image would be, e.g., a black and white picture).
- From the memories, we define/construct the weight matrix W :

$$W := \sum_{i=1}^N \mathbf{x}_i \mathbf{x}_i^T = X X^T. \quad (1.4.1)$$

Here, T denotes the transpose operation. $X = (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N)$ is the $d \times N$ “memory” matrix.

The choice of W determines how memories are retrieved. Note: In the 80s when this model was first invented, researchers used $W = \frac{1}{d} (X X^T - I_d)$, where I_d is the d -dimensional identity matrix. This definition ensures $w_{ii} = 0$ (no self-interaction).

- Consider a given state pattern, $\mathbf{s} \in \mathbb{R}^d$. E.g., this state could be a noisy or partial image (of one of the stored patterns).

The Hopfield model is a recurrent neural network that, starting from an initial input pattern $\mathbf{s}$, updates its units (neurons) according to a specific rule so that, after repeated iterations, the network state evolves toward and hopefully converges to one of the stored memory patterns (and settling into a stable state, or a fixed point).

- The update rule: it is given by

$$\mathbf{s}^{t+1} = \text{sgn}(W \mathbf{s}^t). \quad (1.4.2)$$

Here, $\mathbf{s}^t$ denotes the state vector at time step t . Also, $\text{sgn}(x) = 1$ if $x \geq 0$ and -1 if $x < 0$.

- The hope/goal is that the trajectory will converge to a stored pattern. If one starts with a partial (or noisy) pattern of one of the stored pattern, the model/network would evolve the state toward the original pattern.

The Energy Function

Define the energy function:

$$E(\mathbf{s}) = -\frac{1}{2} \mathbf{s}^T W \mathbf{s} = -\frac{1}{2} \sum_{ij} w_{ij} s_i s_j. \quad (1.4.3)$$

Note: This is motivated by the Ising model, a physical model for spin dynamics (for understanding magnetism and phase transitions).

We are going to show that

[01C=lemma1] $E(\mathbf{s})$ is always nonincreasing when the state $\mathbf{s}$ changes, provided the following:

- W is symmetric (i.e., $w_{ij} = w_{ji}$) and $w_{ij} \geq 0$ (the one given by ([01C=01]) is a special case).
- We assume that ([01C=02]) is applied to update one spin (i.e., one component of $\mathbf{s}$) at a time:

$$s'_i = \text{sgn} \left(\sum_j w_{ij} s_j \right). \quad (1.4.4)$$

This is called the **asynchronous update rule**. Note: We have omitted the superscript t to simplify notation (for this derivation).

Suppose only s_k is updated, with $s'_k = \text{sgn}(\sum_j w_{kj} s_j)$.

Let E' be the energy of the updated system. We are to calculate $E' - E$ (and show that ≤ 0). We need only the terms involving s_k and s'_k in $E' - E$:

$$\begin{aligned} E' - E &= - \sum_{j \neq k} w_{kj} s'_k s_j + \sum_{j \neq k} w_{kj} s_k s_j \\ &= - \sum_{j \neq k} w_{kj} (s'_k - s_k) s_j \\ &= -(s'_k - s_k) \sum_{j \neq k} w_{kj} s_j \end{aligned} \quad (1.4.5)$$

We have two cases:

- $s'_k = s_k$:

Then, $E' - E = 0$.

- $s'_k = -s_k$:

In this case,

$$\begin{aligned} E' - E &= 2s_k \sum_{j \neq k} w_{kj} s_j \\ &= 2s_k \left(\sum_j w_{kj} s_j - w_{kk} s_k \right) \\ &= 2s_k \left(\sum_j w_{kj} s_j \right) - 2w_{kk} \quad \because s_k^2 = 1 \\ &< 0. \end{aligned} \quad (1.4.6)$$

The last line is negative because: (1) $w_{kk} \geq 0$ (by assumption), (2) the sum $\sum_j w_{kj} s_j$ has the same sign as s'_k , which is the opposite of s_k , and so the first term < 0 .

Both cases imply $E' - E \leq 0$.

Note: Since the energy function is bounded from below it follows that it will converge to some value. See the last section on the issues of convergence.

Further Exploration

1. John Hopfield published his influential associative memory model in 1982. It was later recognized that this model is mathematically equivalent to a recurrent neural network proposed a decade earlier by Shun-ichi Amari in 1972. See
 - Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8), 2554–2558.
 - Amari, S. I. (1972). Learning patterns and pattern sequences by self-organizing nets of threshold elements. *IEEE Transactions on Computers*, C-21(11), 1197–1206.

Hopfield was awarded the 2024 Nobel Prize in Physics, which he shared with Geoffrey Hinton, for their foundational discoveries and inventions enabling machine learning with artificial neural networks (ANNs).

2. Convergence of the Hopfield model

Depending on the structure of W and the update rules, the network could converge to a single stable state, or a cycle of states. See, in addition to Hopfield's 1982 paper (cited above):

- Coles, E., Fogelman, F., & Pellegrin, D. (1985). Decreasing energy functions as a tool for studying threshold networks. *Discrete Applied Mathematics*, 12, 261–277.
- Bruck, J. (1990). On the convergence properties of the Hopfield model. *IEEE Transactions on Information Theory*, 35(2), 442–445.

3. Substituting $(\mathbf{01C=01})$ into $(\mathbf{01C=03})$ gives

$$E(\mathbf{s}) = -\frac{1}{2} \sum_{i=1}^N (\mathbf{x}_i^T \mathbf{s})^2. \quad (1.4.7)$$

In recent work (due to, e.g., Krotov et al., Demircigil), this is generalized to energy functions of the form

$$E(\mathbf{s}) = -\sum_{i=1}^N F(\mathbf{x}_i^T \mathbf{s}), \quad (1.4.8)$$

with $F(\cdot)$ being a rapidly growing function (when F is quadratic, it reduces to $(\mathbf{01C=10})$). These “modern Hopfield networks” have been the focus of intense research in recent years. In a sequence note, we will briefly discuss these recent developments. Notably, one such version is deeply connected to the attention mechanism in transformer architectures.

1.5: Modern Hopfield Networks

Key reference: Sect. 1.3 of SD.

Modern Hopfield Networks and the Attention Mechanism

The resurgence of the Hopfield model in recent years has been driven by the introduction of new energy functions and update rules, enabling these networks to overcome classical limitations (such as memory/storage capacity).

In this note, we explore one such model that exhibits interesting connections with the attention mechanism used in transformer architectures. This is due to

Ramsauer, H., Schäfl, B., Lehner, J., Seidl, P., Widrich, M., Gruber, L., Holzleitner, M., Adler, T., Kreil, D., Kopp, M. K., Klambauer, G., Brandstetter, J., & Hochreiter, S. (2021). Hopfield networks is all you need. Presented at the 9th International Conference on Learning Representations (ICLR 2021). arXiv:2008.02217.

The key points are as follows.

The New Energy Function

Given N stored patterns $\mathbf{x}_i \in \mathbb{R}^d$, and the memory matrix $X = (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N) \in \mathbb{R}^{d \times N}$. Note: these are continuous state patterns: their components are real-valued (not just ± 1). The state pattern is given by $\mathbf{s} \in \mathbb{R}^d$. The energy function is defined as follows:

$$E = -\frac{1}{\beta} \ln \left(\sum_{i=1}^N \exp(\beta \mathbf{x}_i^T \mathbf{s}) \right) + \frac{1}{2} \mathbf{s}^T \mathbf{s} + C = -\text{lse}(\beta, X^T \mathbf{s}) + \frac{1}{2} \mathbf{s}^T \mathbf{s} + C, \quad (1.5.1)$$

where C is a constant and the log-sum-exp function is defined as follows: for a vector $\mathbf{z} \in \mathbb{R}^N$,

$$\text{lse}(\beta, \mathbf{z}) := \beta^{-1} \ln \left(\sum_{i=1}^N \exp(\beta z_i) \right). \quad (1.5.2)$$

This energy function is more sharply peaked around stored patterns, making these patterns stable energy minima or attractors to which the network state converges during retrieval.

Note: Another key reason for the use of highly non-linear energy functions (like the above) is to increase storage capacity (which we will not discuss; see the original paper for details).

The New Update Rule

The update rule (like its counterpart in the classic Hopfield model) is designed to drive the system to minimize its energy function. This would cause the input state to converge to a local minimum that corresponds to the stored pattern nearest to the initial input (memory retrieval).

Here, the (new) update rule is derived using the Concave-Convex-Procedure (CCCP; due to Yuille and Rangarajan):

$$\mathbf{s}^{\text{new}} = X \cdot \text{softmax}(\beta X^T \mathbf{s}).$$

The definition of softmax is given in ([01d=20]), which also implies the update rule can be written as

$$\mathbf{s}^{\text{new}} = \frac{\sum_{k=1}^N \mathbf{x}_k \exp(\beta \mathbf{x}_k^T \mathbf{s})}{\sum_{l=1}^N \exp(\beta \mathbf{x}_l^T \mathbf{s})}. \quad (1.5.3)$$

We will explain below (in Section [01d=further]) what CCCP is and how it could be used to derive ([01d=03]).

The update rule resembles the attention mechanism. In the next section, we will explore this connection.

Connection to the Attention Mechanism

We can turn ([01d=03]) into a more familiar form of the attention mechanism formula.

- Applying ([01d=03]) to S patterns $\Xi := (\mathbf{s}_1, \mathbf{s}_2, \dots, \mathbf{s}_S) \in \mathbb{R}^{d \times S}$ gives

$$(\Xi^{\text{new}})^T = \text{softmax}(\beta \Xi^T X) X^T. \quad (1.5.4)$$

To see this, recall that, with a matrix argument, softmax applies to the rows of the input matrix.

- Define the matrices Q (query) and K (key)

$$\begin{aligned} Q &= \Xi^T, \\ K &= X^T, \end{aligned} \quad (1.5.5)$$

as well as setting $\beta = 1/\sqrt{d_k}$. ([01d=06]) becomes

$$Q^{\text{new}} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) K. \quad (1.5.6)$$

Multiply both sides (from the right) by a matrix W_V (projecting the patterns to the associate space). Define $Z = Q^{\text{new}}W_V$ and $V = KW_V$, and the last equation can be written as

$$Z = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V, \quad (1.5.7)$$

which is the usual matrix equation for the attention mechanism. Note: In later sections, we will study the attention mechanism in transformer architecture in more details.

Further Exploration

1. The softmax function

Given a vector $\mathbf{z} \in \mathbb{R}^N$, the softmax : $\mathbb{R}^N \rightarrow (0,1)^N$ is defined as

$$\text{softmax}(\mathbf{z})_i := \frac{\exp(z_i)}{\sum_k \exp(z_k)}. \quad (1.5.8)$$

When applying to a matrix, softmax applies to the rows of the input matrix, meaning that the softmax function is computed independently for each row. In other words, for each row vector in the matrix, softmax exponentiates each element and normalizes by the sum of the exponentials of all elements in that row.

2. The Concave-Convex Procedure (CCCP): What is it?

- For details: Yuille, A. L., & Rangarajan, A. (2003). The concave-convex procedure (CCCP). *Neural Computation*, 15(4), 915–936.
- CCCP is an iterative optimization algorithm for solving problems where the objective function can be decomposed into the sum of a convex function and a concave function.
- The algorithm. Let the objective function be denoted by $E(x)$. Starting with an initial point, x^0 , the algorithm outputs a sequence of points $\{x^t\}$ that converge to a local minimum or saddle point of $E(x)$. Here are the steps.
 - Decompose the objective function $E(x)$ into $E(x) = E_{\text{vex}}(x) + E_{\text{cave}}(x)$, where E_{vex} is convex and E_{cave} is concave.

- Given x^t , the update x^{t+1} is found by solving $\nabla E_{\text{vex}}(x^{t+1}) = -\nabla E_{\text{cave}}(x^t)$.

3. The Concave-Convex Procedure (CCCP): An example

Consider the objective function

$$E(x) = \frac{x^4}{4} - 2x^2 + \frac{x}{2}. \quad (1.5.9)$$

Write

$$\begin{aligned} E_{\text{vex}} &= \frac{x^4}{4} + \frac{x}{2}, \\ E_{\text{cave}} &= -2x^2. \end{aligned} \quad (1.5.10)$$

Solving ([01d=22]), with $\nabla = d/dx$, we have

$$(x^{t+1})^3 + \frac{1}{2} = -(-4x^t), \quad (1.5.11)$$

or, equivalently,

$$x^{t+1} = \left(4x^t - \frac{1}{2}\right)^{1/3}. \quad (1.5.12)$$

Starting with $x^0 = 0$, we have, by iteratively applying ([01d=25]),

$0 \rightarrow -0.7937 \rightarrow -1.5432 \rightarrow -1.8826 \rightarrow -2.0025 \rightarrow -2.0416 \rightarrow -2.0541 \rightarrow -2.0580$

In fact, a global minimum is located at $x \approx -2.05979$.

4. The Concave-Convex Procedure (CCCP): The heuristic behind ([01d=22])

- Key Idea: Replace the challenging non-convex optimization problem by iteratively solving convex approximations obtained by linearizing the concave part.
- Hence: $E_{\text{cave}}(x) \approx E_{\text{cave}}(x^t) + \nabla E_{\text{cave}}(x^t)(x - x^t)$.
- The update x^{t+1}

Instead of minimizing the full $E(x)$, we define the update x^{t+1} as $x^{t+1} = \arg \min_x (E_{\text{vex}}(x) + \nabla E_{\text{cave}}(x^t)x)$

(where we keep only the terms that depend on x). This implies (from the optimality condition): $\nabla E_{\text{vex}}(x^{t+1}) + \nabla E_{\text{cave}}(x^t) = 0$, or equivalently, $\nabla E_{\text{vex}}(x^{t+1}) = -\nabla E_{\text{cave}}(x^t)$, the desired result.

5. Derivation of ([01d=03])

- Decompose $E(\mathbf{s})$ in ([01d=01]) as $E_1(\mathbf{s})$ and $E_2(\mathbf{s})$:

$$\begin{aligned} E(\mathbf{s}) &= E_1(\mathbf{s}) + E_2(\mathbf{s}), \\ E_1(\mathbf{s}) &= \frac{1}{2}\mathbf{s}^T\mathbf{s} + C, & (\text{the convex part}) \\ E_2(\mathbf{s}) &= -\text{lse}(\beta, X^T\mathbf{s}). & (\text{the concave part}) \end{aligned} \quad (1.5.13)$$

Note: See the appendix of “Hopfield networks is all you need” for the proof of the concavity of E_2 .

- Applying ([01d=22]) to E , i.e., $\nabla_{\mathbf{s}} E_1(\mathbf{s}^{t+1}) = -\nabla_{\mathbf{s}} E_2(\mathbf{s}^t)$, where

$$\nabla_{\mathbf{s}} E_1(\mathbf{s}^{t+1}) = \nabla_{\mathbf{s}} \left(\frac{1}{2}\mathbf{s}^T\mathbf{s} + C \right) (\mathbf{s}^{t+1}) = \mathbf{s}^{t+1}, \quad (1.5.14)$$

and

$$\begin{aligned}
 -\nabla_{\mathbf{s}} E_2(\mathbf{s}) &= \nabla_{\mathbf{s}} \text{lse}(\beta, X^T \mathbf{s}) \\
 &= \frac{1}{\beta} \nabla_{\mathbf{s}} \ln \left(\sum_{i=1}^N \exp(\beta \mathbf{x}_i^T \mathbf{s}) \right) \\
 &= \frac{\sum_{k=1}^N \mathbf{x}_k \exp(\beta \mathbf{x}_k^T \mathbf{s})}{\sum_{l=1}^N \exp(\beta \mathbf{x}_l^T \mathbf{s})},
 \end{aligned} \tag{1.5.15}$$

which is $X \cdot \text{softmax}(\beta X^T \mathbf{s})$, or,

$$-\nabla_{\mathbf{s}} E_2(\mathbf{s}^t) = X \cdot \text{softmax}(\beta X^T \mathbf{s}^t). \tag{1.5.16}$$

Combining ([01d=30])-([01d=32]) gives the desired result.

6. Building deep learning networks with Hopfield layers

Modern Hopfield networks are integrated into deep learning models as specialized layers that provide powerful associative memory, pooling, and attention mechanisms. They enhance deep learning models beyond traditional designs, potentially improving performance across a variety of tasks. See the original paper by Ramsauer et al. for details.

1.6: A Brief Discussion

This is based on Section 1.3 of SD.

NN/DL: The Formulation

- Neural networks (NNs) are a broad family of ML models inspired by the structure and function of the human brain.
- Deep learning (DL) is essentially NNs with many layers, hence the term “deep”.

Defining DL:

- A DL model with d hidden layers consists of:
 - d weight matrices $W_0, W_1, \dots, W_{d-1}$
 - A nonlinear activation function $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ applied coordinate-wise. This means: given a input vector $\mathbf{z} = (z_1, z_2, \dots)$, the output would be a vector given by $(\sigma(z_1), \sigma(z_2), \dots)$.
- Common activation functions used in deep learning include:

- **Rectified Linear Unit (ReLU)** with bias:

$\text{ReLU}_b(x) = \max(0, x - b)$
where b is the bias parameter.

- **Sigmoid (Logistic) function:**

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (1.6.1)$$

which compresses the input to the range $(0, 1)$.

- **Hyperbolic tangent (tanh):**

$$\sigma(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (1.6.2)$$

compresses input to the range $(-1, 1)$.

- **Leaky ReLU:**

$$\sigma(x) = \begin{cases} x - b, & x > b \\ \alpha(x - b), & x \leq b \end{cases} \quad (1.6.3)$$

where $\alpha \in (0, 1)$ is a small slope for negative inputs, and b is bias. Design to address the “dying ReLU” problem, where neurons can become inactive during training due to consistently receiving negative inputs, leading to zero output and zero gradients.

- **Parametric ReLU (PReLU):** Similar to Leaky ReLU, but the slope α for negative inputs is learned during training.
- **Exponential Linear Unit (ELU):**

$$\sigma(x) = \begin{cases} x, & x > 0 \\ \alpha(e^x - 1), & x \leq 0 \end{cases} \quad (1.6.4)$$

where $\alpha > 0$ is a hyperparameter controlling the value for negative inputs. Design to address some of the limitations of ReLU while maintaining its advantages.

- **Swish:**

$$\sigma(x) = x \cdot \text{sigmoid}(\beta x) = \frac{x}{1 + e^{-\beta x}} \quad (1.6.5)$$

where β can be a constant or a trainable parameter. Google Brain researchers designed it to improve the performance, compared to ReLU and sigmoid functions. See Ramachandran, P., Zoph, B., & Le, Q. V. (2017). "Searching for activation functions." arXiv:1710.05941.

- Define the input vector as $\mathbf{y}_0 = \mathbf{x}_0$, then the forward pass for $i = 0, \dots, d-1$ proceeds layer-wise as:

$$\mathbf{y}_{i+1} = \sigma(W_i \mathbf{y}_i), \quad (1.6.6)$$

where $\sigma(\cdot)$ is applied coordinate-wise to the input vector. This can also be written as

$$\mathbf{y}_d = \sigma(W_{d-1} \sigma(W_{d-2} \dots \sigma(W_0 \mathbf{y}_0))). \quad (1.6.7)$$

- The dimensions of $\mathbf{y}_i$ and W_i are compatible for matrix-vector multiplication.
- Each coordinate of $\mathbf{y}_i$ corresponds to a node (neuron) in the network; each entry in W_i represents an edge (weight) connecting layers.
- The output of the network is given by $\mathbf{y}_d$, so the DL model defines a mapping:

$$f_{W_1, \dots, W_d} : \mathbb{R}^n \rightarrow \mathbb{R}^m, \quad \mathbf{y}_d = f(\mathbf{x}_0). \quad (1.6.8)$$

Note: The bias terms are absorbed to the weight matrices.

- The total number of parameters equals the sum of all entries in the matrices W_i .
- Training involves solving an optimization problem to find $\{W_i\}$ minimizing a supervised learning loss function, often via empirical risk minimization:

$$\min_{\{W_i\}} \frac{1}{N} \sum_{j=1}^N \mathcal{L}(y_d^{(j)}, t^{(j)}) \quad (1.6.9)$$

where $t^{(j)}$ are target outputs, and $\mathcal{L}$ is a loss function like mean squared error or cross-entropy.

- This optimization problem is typically nonconvex.
- Gradient-based methods like stochastic gradient descent or its variants are used for training, with efficient gradient computation via backpropagation. More on these in later chapters.
- Increasing depth often improves the expressiveness and compactness of representations. That said, shallow learning models and small deep learning (DL) models offer advantages in terms of computational efficiency, faster training times, and reduced resource requirements.

Major Architectures

- Convolutional Neural Networks (CNNs)
 - Exploit local connectivity in data (images, signals). Filters are "convolved" with the input data (e.g., an image): sliding small filters (or kernels) across the input, performing element-wise multiplication with the input, and summing the results to produce single output values.
 - Standard for image and spatial data domains
 - AlexNet (Krizhevsky et al., 2012) that popularized deep CNNs in ImageNet classification.

- Recurrent Neural Networks (RNNs)

- NNs with temporal memory via recurrent connections
- Natural fit for sequential/time series data modeling
- Key references:

Early RNN and LSTM foundational works (Hochreiter & Schmidhuber, 1997)

Popular sequence modeling applications in NLP and time series (Graves, 2013)

- ResNet

- It is a DL architecture that revolutionized image recognition by introducing skip connections to address the vanishing gradient problem in very deep NNs. This allows for training networks with hundreds or even thousands of layers, significantly improving performance in tasks like image classification and object detection.
- Original paper:

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770–778.

- Transformers

- Attention-based architecture for sequence data modeling and completely revolutionized DL/AI, widely used in NLP (e.g., language models). More to come on these amazing developments in later chapters.
- Original papers:

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 5998–6008.

- Deep Neural Decision Forests

- A novel approach that combines the strengths of both deep NNs and decision forests (like Random Forests). The core idea is to enrich classification trees with the power of deep networks to learn data representations while still benefitting from the interpretability and ensemble capabilities of decision forests.
- It introduces stochastic and differentiable decision trees compatible with backpropagation, enabling joint optimization of split and leaf node parameters globally, unlike conventional decision trees that use greedy local training.
- Representation Learning Guided by Trees. The decision trees guide the representation learning in the NN layers, reducing routing uncertainty at split nodes, which helps steer feature learning in lower layers of deep networks.
- Original paper:

Kontschieder, P., Fiterau, M., Criminisi, A., & Bulò, S. R. (2015). Deep neural decision forests. *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 1467–1475.

- Graph Neural Networks (GNNs)

- Designed to process data represented as graphs, which consist of nodes and edges that represent relationships between data points.
- GNNs are particularly useful for analyzing complex relationships and patterns within graph-structured data, where traditional neural networks struggle. They achieve this by iteratively updating node representations based on information from their neighbors, effectively capturing the dependencies and structure within the graph.
- Used for social networks, molecule analysis, and other graph applications

Key references:

- Kipf, T., & Welling, M. (2017). Semi-supervised classification with graph convolutional networks. *International Conference on Learning Representations (ICLR)*.
- Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., & Monfardini, G. (2009). The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1), 61–80. (early work of GNNs)
- Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., & Dahl, G. E. (2017). Neural message passing for quantum chemistry. *Proceedings of the 34th International Conference on Machine Learning (ICML)*, 1263–1272.
- Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., & Bengio, Y. (2018). Graph attention networks. *International Conference on Learning Representations (ICLR)*.
- Hamilton, W., Ying, Z., & Leskovec, J. (2017). Inductive representation learning on large graphs. *Advances in Neural Information Processing Systems (NeurIPS)*, 1024–1034.

We will discuss some of these models (as well as others not mentioned above) in later chapters.

2: LEARNING FROM DATA

[2.1: Chapter 2 Summary](#)

[2.2: Types of Learning](#)

[2.3: A List of Supervised Learning Models](#)

[2.4: Annoy from Spotify](#)

[2.5: Distance Concentration](#)

[2.6: Shannon's Source Coding Theorem](#)

2.1: Chapter 2 Summary

A summary for SD Chapter 2: Learning from example data.

Focus of the Chapter

Supervised learning and unsupervised learning, and the relationship between machine learning (ML) and statistical modelling.

Unsupervised Learning

- Unsupervised learning is a type of ML that analyzes and models data without labeled responses or predefined categories.
- Its goal is to learn directly from input data alone (without human supervision) to discover hidden patterns, structures, or relationships.
- Common models include: clustering (e.g., k-means), association rule learning, and dimensionality reduction (e.g., PCA).
- Applications include: exploratory data analysis, customer segmentation, anomaly detection, and feature extraction.

Supervised Learning

- Supervised learning is a ML approach where models are trained on labeled datasets. This means each input data point is paired with a corresponding correct output or label.
- The objective is to learn a mapping function from inputs to outputs, enabling the model to accurately predict labels for new and unseen data.
- Often, these models are trained with minimizing a loss function that measures the difference between the predicted outputs and the true labels.
- Common models include: support vector machines, tree models, neural networks, and various regression techniques.
- Applications include: image recognition and speech processing to medical diagnosis, financial forecasting, and quality control in manufacturing.

ML vs. Statistical Learning

- Goal:
 - ML focuses on developing algorithms that learn from data to make accurate predictions or decisions.
 - Statistical learning aims to understand relationships between variables, test hypotheses, and provide interpretable explanations based on data.
- Approach:
 - ML is data-driven, learning patterns from training data with an emphasis on predictive accuracy.
 - Statistical learning is hypothesis-driven, starting from a model or assumption and testing against data.
- Methodology:
 - ML uses a broad range of algorithms/models, often non-parametric and does not rely on strict assumptions about data distribution.

- Statistical learning relies on probability theory with models like linear regression and logistic regression, emphasizing model assumptions.
- Model complexity:
 - ML models can be very complex (e.g., DL models) to capture intricate data patterns.
 - Statistical models tend to be simpler to ensure reasonable interpretability.
- Data requirements:
 - ML requires large datasets, using training, validation, and test splits for model learning and fine-tuning.
 - Statistical learning often works with smaller datasets focused on inference rather than prediction.
- Interpretability:
 - Statistical learning models are generally more interpretable and focus on understanding the data-generating process.
 - ML models prioritize predictive performance, sometimes at the expense of interpretability.
- Applications:
 - ML is more commonly used in scenarios demanding automation and prediction accuracy, such as image recognition or user behavior prediction.
 - Statistical learning is preferred in research settings requiring detailed understanding of variable relationships and hypothesis testing.
- Programming and usage:
 - ML often involves less manual programming for model specification as algorithms adjust automatically.
 - Statistical learning requires more manual model selection and specification based on theoretical assumptions.
- Similarities:
 - Both use mathematical frameworks and algorithms to analyze data and uncover patterns.
 - Both require good domain knowledge and data analytic skills.
 - Both emphasize model validation and evaluation techniques.

2.2: Types of Learning

See Chapter 2 of SD for details.

Different Types of Learning

ML involves teaching computers to learn patterns and make decisions from data, primarily through different approaches based on the availability and type of data.

With the rise of (industrial scale) LLMs, various learning approaches have been developed and integrated, leading to new variations and hybrid methods. These techniques are applied at different stages of the LLM training and development process. The goal is to enhance the models' efficiency, intelligence, and accuracy. See the "LLM context" below.

Here are some key learning paradigms:

• Supervised Learning

- Learn from labeled input-output pairs.
- Most common method for classification (e.g., is the given email a spam?) and regression (e.g., predicting house pricing)
- In the LLM context. After initial self-supervised learning, models are fine-tuned using labeled data via various different methods.

• Unsupervised Learning

- Learn patterns from unlabeled data. It focuses on discovering hidden patterns and structures within unlabeled data. Unlike supervised learning, which relies on pre-labeled datasets with known outputs, unsupervised learning algorithms are given raw, unclassified data and are tasked with finding inherent relationships and organizations without explicit guidance.
- Examples: Group customers based on similar buying habits—like “budget shoppers”—without knowing the groups beforehand, and uncover connections between items in the data, such as finding that customers who buy bread often also buy butter.
- In the LLM context. A variation of unsupervised learning—self-supervised learning—is the backbone of LLM (pre-)training. See below.

• Self-Supervised Learning

- When (pre-)training large-scale LLMs in recent years, often trillions of tokens are required. It is clear that relying on human-annotated labeled data is simply practically impossible. This is where self-supervised learning method comes in: it is a method where the model trains itself by generating its own labels from unlabeled data. We say the models create “pretext tasks” that generate their own supervisory signals. Simply put, the models is to learn or predict the masked parts. It is cost- and time-efficient compared to supervised learning since it doesn't require labeled data.
- Computer vision: Learning visual representations by predicting parts of images or video, solving pretext tasks like colorization, patch order prediction, motion and depth estimation, and image reconstruction.
- In the LLM context. It allows models to train on massive unlabeled text corpora to predict missing or next tokens, leveraging the structure of language to learn rich representations implicitly from the data itself.

• Semi-Supervised Learning

- Combine small labeled datasets with large unlabeled data. It leverages the unlabeled data to improve the accuracy of a model trained with limited labeled data, making it particularly useful when labeling data is costly or time-consuming.
- Workflow. First, the model is trained on (small) labeled data. Next, the model predicts labels on the larger unlabeled dataset, creating pseudo-labels. Then, the model is retrained with both the original labeled dataset and the pseudo-labeled data to refine the model for accuracy. This iterative cycle repeats until the model performance stabilizes or meets desired criteria. See, e.g., [1, 7, 12, 15].
- For fine-tuning LLMs, see [11]: it improves LLMs alignment with knowledge propagation (from the small labeled dataset to the larger unlabeled dataset) and collaborative learning (selecting higher-quality pseudo-response samples from the generated unlabeled data).
- See also: LESS (Large Language Model Enhanced Semi-supervised Learning) from NVIDIA [4] and applications with VoxPopuli unlabeled data (a large-scale multilingual speech corpus developed by Facebook; “VoxPopuli” means “voice of the people” in Latin) [16].

• Reinforcement Learning (including RLHF)

- Optimize model behavior using human feedback as reward signals.
- Central for alignment and helpfulness (GPT-3.5+, GPT-4, Gemini, DeepSeek).

• Continuous/Online Learning

- Update models continuously with fresh or domain-specific data post-release.
- Keeps models up to date and relevant (Gemini, LLaMA).

• Curriculum Learning

- Train progressively with data/tasks of increasing complexity.
- Enhances learning efficiency and robustness (Gemini, DeepSeek).
- Other types of learning: Transfer learning, in-context learning, multi-task learning, multitask learning (MTL), meta learning, federated learning, active learning, contrastive learning, neuro symbolic AI, collaborative Learning, and more.

Further Exploration

We want to briefly discuss on how different types of learning are used/deployed in training large-scale LLMs (some would overlap with above discussion).

1. Training large-scale LLMs: the main phases: see, e.g., [14]. A recent review on LLM alignment: see [17].
2. See also
 - Instruction Tuning: After initial self-supervised training, models are fine-tuned using labeled data to follow instructions and improve task performance. See, e.g., [3, 18].
 - Instruction Pretraining: Improving LLMs efficiency by including synthetic instruction-response pairs instead of just raw text. This is done via an “instruction synthesizer,” an LLM specifically finetuned for this task. See [2].
 - Rejection sampling (RS): filter out less desirable responses from a model’s output and only use the high-quality ones for further training; e.g., in Llama 3 (see [10]).

- Parameter-Efficient Fine-Tuning (PEFT). Instead of fine-tuning all the parameters of a pre-trained LLM, it focuses on adjusting a small subset of parameters (a cost-efficient method scaling to large models with smaller compute). Use task-specific datasets containing labeled data (input-output pairs) that guide the model in learning the desired task. See, e.g., [5].
- On Prompt Tuning (a technique that focuses on training or adjusting a small set of parameters called “soft prompts” or “prompt parameters” alongside the input text to improve the model's performance on specific tasks): see, e.g., [9].
- Direct Preference Optimization (DPO)
 - Directly optimize for human preferences without traditional RL loops.
 - Emerging alternative to PPO-based RLHF used in Gemini, GPT-4.5, DeepSeek.
- Synthetic Data Training
 - Use model- or tool-generated data, especially for instruction tuning.
 - Enables large-scale generation of QA and reasoning datasets (LLaMA 3, Gemini).
- Instruction Pretraining
 - Incorporate instruction-like data directly in base pretraining.
 - Models gain native instruction-following ability (Phi-3, LLaMA 3, Gemini).
- Constitutional AI
 - Use AI-generated or rule-based feedback alongside or instead of human labeling.
 - Aims to improve alignment and safety with less human annotation (Gemini, GPT-4).
- Multimodal Joint Pretraining
 - Train jointly on multiple data types: text, images, audio, code.
 - Supports versatile vision-language capabilities (Gemini, LLaVA, Qwen-VL).
- Mixture of Experts (MoE)
 - Sparse activation of specialized subnetworks (experts) selected dynamically.
 - Scales model capacity efficiently (LLaMA 3, GPT-4 reported, Gemini).
- Knowledge-Enhanced Pretraining
 - Integrate structured external knowledge (e.g., knowledge graphs) during training.
 - Improves reasoning and interpretability (some Gemini variants).
- Weight-Generative Fine-Tuning (WeGeFT)
 - Dynamically generate fine-tuning weights for efficient task adaptation.
 - Cutting-edge research advancing fine-tuning (2025).
- Zero-shot/Few-shot Prompting Zero-shot, One-shot, and Few-shot Learning
 - Use natural language prompts with zero or few examples to induce new behavior.
 - Core capability of all modern LLMs (GPT, Gemini, etc.).

- Retrieval-Augmented Generation (RAG)
 - Combine retrieval of external documents with generation for more factual output.
 - Used in Gemini, GPT-4, DeepSeek-VL.
- Long Context/Chunked Training
 - Specialized training for extremely long input sequences (100K+ tokens).
 - Enables handling very large documents (Gemini, LLaMA 3).
- Knowledge Distillation/Model Compression
 - Train smaller “student” models to mimic larger “teacher” models.
 - Reduces inference cost while preserving performance (Gemini Nano, DistilGPT).

3. Some recent surveys on LLMs: see [6, 8, 13, 19].

References:

1. Chapelle, O., Schölkopf, B., & Zien, A. (Eds.). (2006). *Semi-supervised learning*. MIT Press.
2. Cheng, D., Gu, Y., Huang, S., Bi, J., Huang, M., & Wei, F. (2024). Instruction pre-training: Language models are supervised multitask learners. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing* (pp. 2529–2550). Association for Computational Linguistics.
3. DeepLearningAI. (2023, May 31). *Building with Instruction-Tuned LLMs: A Step-by-Step Guide* [Video]. YouTube.
4. Ding, W., & Qian, F. (2025). LESS: Large language model enhanced semi-supervised learning for speech foundational models. arXiv:2506.04586v3.
5. Han, Z., Gao, C., Liu, J., Zhang, J., & Zhang, S. Q. (2024). Parameter-efficient fine-tuning for large models: A comprehensive survey. *Transactions on Machine Learning Research (TMLR)*. arXiv:2403.14608.
6. Kalyan, K. S. (2024). A survey of GPT-3 family large language models including ChatGPT and GPT-4. *Natural Language Processing Journal*, 6.
7. Kingma, D. P., Rezende, D. J., Mohamed, S., & Welling, M. (2014). Semi-supervised learning with deep generative models. *Advances in Neural Information Processing Systems*, 27, *NeurIPS 2014*, 3581–3589.
8. Kumar, P. (2024). Large language models (LLMs): Survey, technical frameworks, and future challenges. *Artificial Intelligence Review*, 57(10), Article 260.
9. Lester, B., Al-Rfou, R., & Constant, N. (2021). The power of scale for parameter-efficient prompt tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (pp. 3045–3059). Association for Computational Linguistics.
10. Llama Team, AI @ Meta. (2024, July). The Llama 3 Herd of Models. arXiv:2407.21783.
11. Luo, J., Luo, X., Chen, X., Xiao, Z., Ju, W., & Zhang, M. (2025). Semi-supervised fine-tuning for large language models. *Findings of the North American Chapter of the Association for Computational Linguistics (NAACL)*. arXiv:2410.14745.
12. Ouali, Y., Hudelot, C., & Tami, M. (2020). An overview of deep semi-supervised learning. arXiv:2006.05278.
13. Shahzad, T., Mazhar, T., Tariq, M. U., Ahmad, W., Ouahada, K., & Hamam, H. (2024). A comprehensive review of large language models: Issues and solutions in learning environments. *Sustainable Computing: Informatics and Systems*.
14. Snorkel AI. (2023, April 20). *Large language model training: Three phases that shape LLM training*. Snorkel AI Blog.
15. Van Engelen, B., & Hoos, H. (2019). A survey on semi-supervised learning. *Machine Learning*, 109(2), 373–440.
16. Wang, C., Rivière, M., Lee, A., Wu, A., Talnikar, C., Haziza, D., Williamson, M., Pino, J., & Dupoux, E. (2021). Voxpopuli: A large-scale multilingual speech corpus for representation learning, semi-supervised learning and interpretation. arXiv:2101.00390.
17. Wang, Z., Bi, B., Pentyala, S. K., Ramnath, K., Chaudhuri, S., Mehrotra, S., Zhu, Z. J., Mao, X.-B., Asur, S., & Cheng, N. (2024). A comprehensive survey of LLM alignment techniques: RLHF, RLAIF, PPO, DPO and more. arXiv:2407.16216.
18. Zhang, S., Dong, L., Li, X., Zhang, S., Sun, X., Wang, S., Li, J., Hu, R., Zhang, T., Wu, F., & Wang, G. (2023). Instruction tuning for large language models: A survey. arXiv:2308.10792.
19. Zhao, W. X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., Du, Y., Yang, C., Chen, Y., Chen, Z., Jiang, J., Ren, R., Li, Y., Tang, X., Liu, Z., Liu, P., Nie, J.-Y., & Wen, J.-R. (2023). A survey of large language models. arXiv:2303.18223.

This page titled [2.2: Types of Learning](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by Hei-Chi Chan.

2.3: A List of Supervised Learning Models

This is based on Sect. 2.1.2 of SD.

We discuss briefly the most popular supervised learning models. Unsupervised learning models will be discussed in later chapters.

1. Multiple Regression

- Given a data point $\mathbf{x} \in \mathbb{R}^p$, it predict an output by a linear relationship:

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p. \quad (2.3.1)$$

The parameters/coefficients $\beta_0, \beta_1, \dots, \beta_p$ are to be learned from the data. Note: p is the number of features.

- The parameters are estimated (learned) by minimizing the lost function defined by the Mean Squared Error (MSE):

$$\text{RSS} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 = \frac{1}{n} \sum_{i=1}^n \left(y_i - \left(\beta_0 + \sum_{j=1}^p \beta_j x_{ij} \right) \right)^2 \quad (2.3.2)$$

Here, n is the size of the training dataset. The notation x_{ij} : it denotes the j -th feature of the i -th data (or, the j -th component of the vector $\mathbf{x}_i$).

2. Logistic Regression

- It is a model that predicts the probability of a categorical outcome (or, a binary outcome if there are only two possible states).
- Consider binary outcomes (say, 0 or 1). Given a data point $\mathbf{x} \in \mathbb{R}^p$, it predicts the probability of the output being 1 by the following formula:

$$p(\mathbf{x}) = P(y = 1 \mid \mathbf{x}) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p)}} \quad (2.3.3)$$

Here, $\{\beta_j\}$ are the parameters (to be learned from the data). Note that, the function

$$\sigma(z) := \frac{1}{1 + e^{-z}} \quad (2.3.4)$$

is called the *sigmoid function*. Hence ([02b=03]) can be written as

$$p(\mathbf{x}) = \sigma(\beta_0 + \underbrace{\beta_1 x_1 + \cdots + \beta_p x_p}_{:=\beta \cdot \mathbf{x}}) = \sigma(\beta \cdot \mathbf{x} + \beta_0). \quad (2.3.5)$$

- The parameters are estimated (learned) by minimizing the lost function defined by binary cross-entropy loss (negative log-likelihood):

$$\mathcal{L}(\beta) = - \sum_{i=1}^n [y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)], \quad (2.3.6)$$

where

- $y_i \in \{0, 1\}$ is the observed class label
- $\hat{y}_i = \frac{1}{1 + e^{-(\beta \cdot \mathbf{x}_i + \beta_0)}}$ is the predicted probability for the i -th observation/data point (i.e., ([02b=05]))

- n is the number of observations.

4. Remarks: For the case of multiple outcomes, For predicting/modeling the probability of each class c among K classes, one has

$$p(y = c \mid \mathbf{x}) = \frac{e^{\mathbf{w}_c \cdot \mathbf{x} + b_c}}{\sum_{k=1}^K e^{\mathbf{w}_k \cdot \mathbf{x} + b_k}}, \quad (2.3.7)$$

where $\mathbf{w}_c$ and b_c are the weight vector and bias for class c , and $\mathbf{x}$ is the input feature vector. ([02b=07]), we assume

The loss function is again the corresponding cross-entropy loss. It measures the dissimilarity between the true distribution and the predicted distribution. It is defined by

For the entire dataset with N samples, the total loss is averaged or summed over all samples:

$$L = -\frac{1}{n} \sum_{i=1}^n \left(\sum_{c=1}^K y_{i,c} \log \hat{y}_{i,c} \right). \quad (2.3.8)$$

Here,

- n is the number of data points
- the sum in the parenthesis is the cross-entropy of the i -th sample/data point
- $y_{i,c}$ is the indicator that sample i belongs to class c , and $\hat{y}_{i,c}$ is the predicted softmax probability for class c of the i -th data point.

This cross-entropy loss encourages the model to assign high probability to the correct class, and is differentiable for various optimization methods.

5. To motivate the formula in ([02b=07]):

1. Say, there are K classes.

2. Given a data point, $\mathbf{x}$, we set

$$p(y = c \mid \mathbf{x}) \propto e^{f_c(\mathbf{x})}.$$

This is a natural choice: $p(y = c \mid \mathbf{x})$ is probability and it has to be positive, so we define it as an exponential function. The exponent, $f_c(\cdot)$, depend on the data point/input vector.

3. Since these are probabilities, we normalize them so that their sum is 1. This normalization is given by

$$p(y = c \mid \mathbf{x}) \propto e^{f_c(\mathbf{x})} \rightarrow \frac{e^{f_c(\mathbf{x})}}{\sum_d e^{f_d(\mathbf{x})}}. \quad (2.3.9)$$

This is same as the softmax function (see DL Notes (OER): Modern Hopfield Networks).

4. To obtain ([02b=07]), we assume $f_c(\cdot)$ is a linear function of the input vector.

3. Support Vector Machine (SVM)

SVM is designed to find optimal hyperplane in high-dimensional data spaces for classification or regression. We will explore this in more detail in later sections.

4. K -Nearest Neighbors (KNN)

- Basic idea: Classify or predict a data point based on the labels or values of its K nearest neighbors in the feature/data space. KNN assumes that points close to each other are likely to have similar outcomes (“Birds of a feather flock together”).

- The Algorithm:

- Step 1: Calculate distances between the query point x and all points x_i in the dataset.
- Step 2: Identify the K nearest neighbors by selecting the K points with the smallest distances to the query point.
- Step 3: Aggregate neighbors' outputs:

- For regression:

$$\hat{y}_q = \frac{1}{K} \sum_{i=1}^K y_i \quad (2.3.10)$$

where $\hat{y}_q$ is the predicted value, and y_i is the value of the i -th nearest neighbor.

- For classification: One counts how many of the K nearest neighbors belong to each class, and then assigns the query point the class with the most votes (the majority class among the neighbors):

$$\hat{c}_q = \arg \max_c \sum_{i=1}^K \mathbf{1}(c_i = c) \quad (2.3.11)$$

where $\hat{c}_q$ is the predicted class of the query point, c_i is the class label of the i -th nearest neighbor, and $\mathbf{1}$ is the indicator function counting votes per class. In ([02b=11]): For each class value c , count the number of times it appears among the K -nearest neighbors. Then, output the class value with the highest occurrence (i.e., the highest vote).

- Note: KNN is non-parametric. KNN does not build an explicit model; it stores training data and uses distance-based lookup for prediction.
- Choosing the optimal number of neighbors (K)
 - The value of K controls how many neighbors influence the prediction. Selecting the right K is crucial for balancing bias and variance:
 - A small K (e.g., $K = 1$) can lead to overfitting, being sensitive to noise and outliers.
 - A large K smooths predictions, reducing variance but possibly causing underfitting by ignoring local patterns.
 - Some common strategies to select K include:
 - Elbow Method: Plot error rate (or accuracy) versus K . Choose K at the “elbow” where error stops decreasing significantly.
 - Heuristic rules: Use $K = \sqrt{n}$ or $K = \frac{\sqrt{n}}{2}$ as starting points, where n is the number of training samples.
 - Odd values for K in classification tasks to avoid ties.
 - It is important to tune K based on the dataset and task, as there is no universal best value.
- Approximate Nearest Neighbor (ANN) Search: we will discuss this in a separate note.

5. Naïve Bayes (NB)

Or, Naïve Bayes, with the diaeresis.

- Model assumption: NB is a probabilistic classifier based on Bayes' theorem with the *naïve* assumption that all features x_i are conditionally independent given the class c_k .
- Bayes' theorem formulation:

$$P(c_k | \mathbf{x}) = \frac{P(c_k) \cdot P(\mathbf{x} | c_k)}{P(\mathbf{x})} \quad (2.3.12)$$

where:

- $P(c_k | \mathbf{x})$ is the (posterior) probability of class c_k given features $\mathbf{x} = (x_1, x_2, \dots, x_p)$
 - $P(c_k)$ is the (prior) probability of class c_k
 - $P(\mathbf{x} | c_k)$ is the likelihood of features given the class (conditional probability of observing the input features $\mathbf{x}$ given that the instance belongs to class c_k)
 - $P(\mathbf{x})$ is the evidence (constant across classes for given $\mathbf{x}$).
- Naive conditional independence assumption:

$$P(\mathbf{x} | c_k) = \prod_{i=1}^p P(x_i | c_k) \quad (2.3.13)$$

Thus,

$$P(c_k | \mathbf{x}) \propto P(c_k) \prod_{i=1}^p P(x_i | c_k) \quad (2.3.14)$$

- Classification rule: Choose the class $\hat{c}$ that maximizes (the posterior probability):

$$\hat{c} = \arg \max_{c_k} P(c_k) \prod_{i=1}^p P(x_i | c_k) \quad (2.3.15)$$

- Model components:
 - $P(c_k)$ can be estimated as the relative frequency of class c_k in training data.
 - $P(x_i | c_k)$ is modeled using appropriate probability distributions and depends on the type of features (e.g., Gaussian for continuous, multinomial for counts, Bernoulli for binary).
- Loss function: Naive Bayes does not explicitly optimize a separate loss function like regression or neural networks. Instead, it is a generative model estimating likelihoods and priors, and classification is performed by maximizing posterior probabilities.
- Computational advantage: Due to the independence assumption, the model is efficient to train and effective even with limited data, particularly in high-dimensional settings like text classification.
- Shortcomings (when using NB in NLP)
 - Independence assumption: Assumes all words are independent (not true).
 - Ignores word order: Loses context by treating text as a bag of words.
 - Misses dependencies: Fails to capture collocations, negation, or co-occurrence patterns.
 - Rare/unseen words: Struggles without smoothing (as it assigns zero probabilities to them).
 - Limited semantics: Cannot model sarcasm, irony, or deeper meaning.
 - Bias to frequent words: Common terms may dominate predictions.

That said, NB is much less costly to run/implement (than the current cost of running LLMs).

- Key papers:

Robertson, S. E., & Sparck Jones, K. (1976). Relevance weighting of search terms. *Journal of the American Society for Information Science*, 27(3), 129–146.

Kononenko, I. (1990). Comparison of inductive and naïve Bayesian learning approaches. In *Current trends in knowledge acquisition* (pp. 187–200). Amsterdam, The Netherlands: IOS Press.

6. Random Forest (RF)

- Definition: An ensemble learning method that builds a collection of decision trees (usually M trees) and combines their predictions to improve accuracy and reduce overfitting.
- Training data notation: Let $D_n = \{(x_i, y_i)\}_{i=1}^n$ be the full training dataset of size n , where x_i is the i -th input vector and y_i are target values. Here, n is the number of training observations.
- For each tree $j = 1, 2, \dots, M$, a randomized decision tree is constructed by:
 - Sampling with replacement (bootstrap sampling) a set $D_n^*(\Theta_j)$ of size n from D_n .
 - Selecting a random subset of features at each node split. The number of random features is a hyperparameter.
 - Splitting nodes using a criterion (e.g., Gini impurity or information gain for classification; variance reduction for regression).
 - Growing the tree until stopping criteria (e.g., minimum leaf size) are met.
- Single tree prediction for input x (regression form):

$$m_n(x; \Theta_j, D_n) = \frac{1}{N_n(x; \Theta_j, D_n)} \sum_{i \in A_n(x; \Theta_j, D_n)} y_i \quad (2.3.16)$$

where:

- Θ_j represents the randomness parameter associated with the j -th tree in the forest. It captures all the stochastic aspects involved in the construction of that specific tree.
- $A_n(x; \Theta_j, D_n)$ is the leaf containing x in tree j .
- $N_n(x; \Theta_j, D_n)$ is the number of training samples in that leaf.
- y_i are target outputs of those samples.
- Ensemble prediction:

$$m_{M,n}(x) = \frac{1}{M} \sum_{j=1}^M m_n(x; \Theta_j, D_n) \quad (2.3.17)$$

For classification, this usually corresponds to majority voting on the predicted classes.

- Loss function: RF does not explicitly optimize a single, global loss function. Each tree minimizes a local impurity measure (e.g., Gini, entropy, variance) at each split. The ensemble reduces variance through averaging of decorrelated trees.
- Key advantages: (1) Lower overfitting compared to single decision trees. (2) Works well with high-dimensional features. (3) Robust to noise and outliers (averaging reduce both).
- Key papers:

Breiman, L. (2001). Random forests. *Machine Learning*, 45(1), 5–32.

Ho, T. K. (1995). Random decision forests. In *Proceedings of the Third International Conference on Document Analysis*

and Recognition (Vol. 1, pp. 278–282). IEEE Computer Society.

Amit, Y., & Geman, D. (1997). Shape quantization and recognition with randomized trees. *Neural Computation*, 9(7), 1545–1588.

Dietterich, T. G. (2000). Ensemble methods in machine learning. In *Multiple classifier systems (Lecture Notes in Computer Science*, Vol. 1857, pp. 1–15). Springer.

See also the review:

Biau, G., & Scornet, E. (2016). A random forest guided tour. *TEST*, 25(2), 197–227.

7. Extreme Gradient Boosting (XGBoost)

- XGBoost (eXtreme Gradient Boosting) is an ensemble learning method that builds a sequence of decision trees (base learners) to create a strong predictive model by iteratively correcting errors of previous trees.
- The prediction for the i -th data point is the sum of the predictions from all K trees:

$$\hat{y}_i = \sum_{k=1}^K f_k(x_i) \quad (2.3.18)$$

where:

- $\hat{y}_i$ is the predicted value.
- K is the total number of trees.
- f_k represents the function (tree) learned at iteration k .
- The objective function minimized by XGBoost combines a differentiable loss function l that measures the difference between predicted values $\hat{y}_i$ and true labels y_i , and a regularization term Ω that penalizes complex tree structures:

$$\mathcal{L}(\theta) = \sum_{i=1}^n l(y_i, \hat{y}_i) + \sum_{k=1}^K \Omega(f_k) \quad (2.3.19)$$

where:

- n is the number of data points,
- l is the chosen loss function (e.g., squared error for regression or logistic loss for classification),
- $\Omega(f_k)$ typically includes terms that measure the complexity of tree f_k , such as the number of leaves and leaf weights, helping to prevent overfitting.
- XGBoost uses a second-order Taylor expansion (using first and second derivatives of the loss with respect to predictions) to approximate the objective function for efficient optimization and to select the best tree splits.
- Each new tree is trained to fit the negative gradient (pseudo residuals) of the loss function with respect to the current prediction, effectively correcting errors made by the previous ensemble.
- Regularization parameters, shrinkage (learning rate), and tree pruning are built-in to control model complexity and improve generalization.
- Trees in XGBoost are grown level-wise (breadth-first), evaluating all possible splits at each depth level and choosing the one that optimizes the objective function.
- The optimization at each iteration involves solving for the leaf weights that minimize the regularized loss, often using closed-form solutions derived from the Taylor approximation.

- Key papers:

Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 785–794). ACM.

8. NN/DL: We will discuss this in depth in later chapters.

Further Exploration

1. When discussing the lost function of logistic regression, we come across the concept of cross-entropy; see ([02b=06]) and ([02b=08]). So what exactly is entropy?

- Entropy

1. In information theory, entropy measures the average uncertainty or information content associated with a random variable (RV). It quantifies how unpredictable or informative the outcome of a random process is.
2. Consider a discrete source with a finite set of possible (independent) symbols $\{s_1, s_2, \dots, s_k\}$ each occurring with probability $\{p_1, p_2, \dots, p_k\}$. Then the entropy is defined as

$$H = - \sum_{i=1}^k p_i \log_2 p_i = \sum_{i=1}^k p_i \log_2 \left(\frac{1}{p_i} \right). \quad (2.3.20)$$

Logarithm to the base 2 has been used, reflecting binary coding in bits.

3. In what sense does ([02b=101]) measure the average uncertainty or information content associated with a RV? The key idea is, entropy measures the average “surprise” value. To this end, define the surprise value of a symbol, s , by

$$I(s) := \log_2 \left(\frac{1}{p(s)} \right) = -\log_2 p(s). \quad (2.3.21)$$

This “surprise” is a way to measure information content: rare events (with small probabilities) carry more information because their occurrence tells us something less expected, while common (“likely”) events tell us less. The way $I(s)$ defined above reflects this: smaller $p(s)$ leads to a bigger $I(s)$, and vice versa. The overall *average* surprise (or, information) is given by ([02b=101]), the entropy: following the same convention for symbols and their probabilities above,

$$\text{Average Surprise} = \mathbb{E}[I] = \sum_{i=1}^k p_i I(s_i) = \sum_{i=1}^k p_i \log_2 \left(\frac{1}{p_i} \right). \quad (2.3.22)$$

4. Curiously, H sets the theoretical minimum average number of bits needed for lossless encoding. According to Shannon’s Source Coding Theorem, a very remarkable result, the entropy of a source sets the absolute limit for how much one can compress that data without losing information. No matter how clever one’s coding method is, one cannot reduce the average bits per symbol below the entropy.

- Cross-entropy

1. Cross entropy intuitively measures how different two probability distributions are by comparing how well one distribution (the predicted or approximate one) matches the true distribution of events. It tells us, on average, how surprised we would be if we expected the probabilities given by the approximate distribution but observed outcomes following the true distribution. When the predicted distribution matches the true distribution perfectly, cross entropy equals the entropy (the true average uncertainty). If the predicted distribution is different, cross entropy is larger, reflecting greater mismatch and “extra uncertainty” compared to the true distribution.
2. Formula: The cross entropy between two probability distributions P (true distribution) and Q (approximate or predicted distribution) over the same set of events is given by

$$H(P, Q) = - \sum_{i=1}^k p_i \log_2 q_i = \sum_{i=1}^k p_i \log_2 \left(\frac{1}{q_i} \right). \quad (2.3.23)$$

Here,

- p_i : the true probability of the i -th event (and P denotes the set $\{p_j\}$)
- q_i : the *predicted* probability of the i -th event (and Q denotes the set $\{q_j\}$)

2. Distances:

- Euclidean distance:

$$d(\mathbf{x}, \mathbf{x}_i) = \sqrt{\sum_{j=1}^n (x_j - x_{ij})^2} \quad (2.3.24)$$

- Manhattan distance:

$$d(\mathbf{x}, \mathbf{x}_i) = \sum_{j=1}^n |x_j - x_{ij}| \quad (2.3.25)$$

- Minkowski distance:

$$d(\mathbf{x}, \mathbf{x}_i) = \left(\sum_{j=1}^n |x_j - x_{ij}|^p \right)^{1/p} \quad (2.3.26)$$

where $p \geq 1$, with special cases $p = 2$ (Euclidean) and $p = 1$ (Manhattan).

3. Approximate Nearest Neighbor (ANN) Search

4. Ensemble Learning

- Ensemble learning is a ML technique that combines multiple models (called base learners) to improve predictive performance.
- It leverages the diversity and complementarities of individual models to produce more accurate and robust predictions than any single model alone.
- Common ensemble methods include:
 - Bagging (Bootstrap aggregating): trains multiple base models independently on different random subsets of the training data sampled with replacement, then aggregates their predictions to reduce variance and prevent overfitting.
 - Boosting: builds models sequentially, where each new model focuses on correcting the errors made by previous models, gradually improving accuracy by combining weak learners into a strong learner.
 - Stacking: combines predictions of multiple different models (possibly heterogeneous) by training a meta-model on their outputs, aiming to learn the best way to integrate their predictions for improved final performance.
- The combined predictions can be aggregated through voting, averaging, or weighted combinations depending on the task.
- Ensemble learning helps reduce errors due to bias, variance, or noise and is widely used for classification and regression problems.

2.4: Annoy from Spotify

This is follow-up discussion on ANN and a bit of its recent developments.

Approximate Nearest Neighbor (ANN) Search

1. Why ANN?

- Problems with kNN: While KNN is conceptually simple, as the number of dimensions (i.e., input features) and the size of the dataset increase, calculating distances between data points becomes computationally costly and impractical. Think of this: if you have 100 millions items in the dataset and want to search for the 6 nearest neighbors of a new query point, KNN requires you to scan through all 100 millions to get the job done!¹
- The Rise of ANN: To address the limitations of kNN, Approximate Nearest Neighbor (ANN) algorithms emerged. These algorithms prioritize speed and scalability over finding the absolute nearest neighbor, often achieving significantly faster search times by accepting a small compromise in accuracy. This tradeoff is especially beneficial in applications requiring real-time performance on massive datasets like recommendation systems (Spotify trying to find a match for your favorite from among their 100 million+ pieces of content). A key early development in ANN was Locality Sensitive Hashing (LSH). The idea is to transform high-dimensional data into lower-dimensional representations (hash codes) in a way that similar data points are more likely to be hashed into the same “buckets”. This reduces the search space, enabling faster approximate nearest neighbor searches.

2. In later sections, we will explore FAISS and ScaNN. They are both open-source libraries for efficient similarity search in large vector datasets. FAISS (developed by Meta) offers a wider array of algorithms, including both exact and approximate search methods, and supporting GPU acceleration. ScaNN (developed by Google) focuses on speed and accuracy for ANN search, particularly excelling in Maximum Inner Product Search (MIPS) and integrating well with TensorFlow.

Annoy (Approximate Nearest Neighbors Oh Yeah)

- What is Annoy? Developed by Spotify in 2015, Annoy is a library that utilizes a tree-based approach. The core idea is to build multiple binary trees (a forest) by recursively splitting the data space with random hyperplanes, so that similar points tend to be grouped and can be searched efficiently. Annoy is particularly noted for its memory efficiency, static file-based indexes that can be memory-mapped and shared across processes, and its ability to balance accuracy and speed. Here are the key ideas behind the algorithm:

1. Tree construction process:

- At each step, randomly select two points from the data.
- Create a hyperplane perpendicular to the line connecting these two points This will be a splitting boundary.
- Partition the dataset into two subsets based using the splitting boundary.
- Repeat the process recursively on each subset until a specified depth is reached or a subset has very few points.
- Note: This recursive splitting builds a binary tree structure where each node corresponds to a hyperplane split. This enhances efficient binary decisions during search.

2. Build multiple trees:

Build many independent trees, each with different random splits. This is to capture diverse ways of partitioning the data space.

3. Search/query:

- Given a query point, traverse down each of the binary trees from root to leaf, making binary decisions at each node based on which side of the hyperplane the query lies.
 - Collect candidate neighbors from the leaf nodes reached in all trees.
 - Aggregate and check these candidate neighbors to find the nearest neighbors approximately.
 - Note: Searching in each tree uses the binary tree property allows narrowing down candidate neighbors rather than scanning the entire dataset.
- Annoy is not built with GPU acceleration in mind
 - A bit about the CPU and GPU:
 - CPUs have few powerful cores optimized for sequential, branching, recursive logic. Whereas GPUs have thousands of simpler cores optimized for parallel, uniform computation.
 - GPUs excel at vectorized, massively parallel computations (e.g., quantization, matrix ops) better than tree-based search, where CPU cache and branch prediction work better.
 - Annoy involves significantly sequential and/or recursive processes, making them for suitable for CPUs (and difficult to parallelize on GPUs):
 - Algorithm type: Annoy uses multiple binary trees constructed by recursive random splits, relying heavily on tree traversal.
 - Search process: Searching involves sequential, conditional steps (if-else decisions) that are difficult to parallelize on GPUs.
 - Data structure: Tree traversal requires following paths based on decisions at each node, which is less suited for the massively parallel architecture of GPUs.
 - Implementation: Annoy was designed primarily for CPU efficiency and memory-mapped file usage, not for GPU hardware or parallel processing.

Further Exploration

1. Practitioners evaluate and compare different ANN methods using standardized benchmarks (like ANN-Benchmarks). These provide rigorous tests for various ANN algorithms across various datasets and performance metrics.
2. Some other ANN algorithms:
 - Hierarchical Navigable Small World (HNSW)
 - Constructs a multi-layer proximity graph for fast traversal.
 - Advantages: High recall and throughput; efficient in high dimensions.
 - Product Quantization (PQ)
 - Compresses vectors into smaller codes by clustering subspaces.
 - Advantages: Reduces memory usage and speeds up distance computations.
 - Locality-Sensitive Hashing (LSH)
 - Maps similar vectors to the same buckets using hash functions.
 - Advantages: Efficient for very large datasets; reduces search space.

- DiskANN (from Microsoft)
 - Uses disk-friendly Vamana graph and PQ for billion-scale datasets.
 - Advantages: Minimizes memory usage; suitable for disk-resident large datasets.
- Anisotropic Vector Quantization (AVQ)
 - Adaptively shapes quantization cells elliptically to better match data distribution. Penalizes quantization error in direction parallel to vectors, improving top inner product accuracy.
 - Advantages: Improved recall over PQ for non-uniform embedding densities; better accuracy in semantic search; suited for high-dimensional, non-uniform data.
 -

1. Complications from high-dimensionality. High-dimensional spaces are “weird,” making it challenging for search algorithms like KNN. This “weirdness” is often referred to as the curse of dimensionality:

- Sparsity: As the number of dimensions increases, the volume of the space grows exponentially. Imagine a line, then a square, then a cube – each step adds a new dimension and requires vastly more points to fill the space. In high dimensions, data points become incredibly spread out, making the space extremely sparse. This means that most of the high-dimensional space is empty, making it hard for KNN to find relevant neighbors.
- Distances lose meaning: In high-dimensional spaces, the concept of “distance” becomes less effective because the distances between data points tend to converge, meaning they all become roughly the same. This makes it difficult to distinguish between truly close and distant points, hindering KNN’s ability to identify meaningful neighbors. Essentially, in high dimensions, it becomes harder for distance metrics like Euclidean distance to accurately reflect the true similarity or proximity between data points).
- Increased Computational Cost: More dimensions translate to more complex calculations for each distance computation. This leads to slower training and prediction times, making KNN computationally expensive for high-dimensional datasets.



2.5: Distance Concentration

This is to follow up on our discussion on the curse of dimensionality (see the section “Approximate Nearest Neighbor (ANN) Search”).

Distance Concentration in Higher Dimensions

High dimensional spaces have very special features that pose challenges to ML models (which is often referred to as the curse of dimensionality). One aspect of this “curse” is distance concentration: In high dimensional spaces, distances between all pairs of points tend to concentrate. For most points, the distance to the nearest point tends to be approximately the same as the distance to the farthest data point. In this note, we want to prove a result to illustrate such a phenomenon.

Consider random vectors, $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^p$, with all their components being i.i.d. uniformly on the interval $[0, 1]$. We have

$$\begin{aligned}\mathbb{E} [\|\mathbf{X} - \mathbf{Y}\|^2] &= \frac{p}{6}, \\ \text{SD} [\|\mathbf{X} - \mathbf{Y}\|^2] &\approx 0.2\sqrt{p}.\end{aligned}\tag{2.5.1}$$

(Here, SD denotes standard deviation).

Note: The first equation implies the mean squared distance between two random vectors grow linearly with p , while the second implies the spread (typical fluctuation) grows much slower (only as $\sqrt{p}$).

Together they imply

$$\frac{\text{SD} [\|\mathbf{X} - \mathbf{Y}\|^2]}{\mathbb{E} \text{SD} [\|\mathbf{X} - \mathbf{Y}\|^2]} \propto \frac{1}{\sqrt{p}} \rightarrow 0 \quad (\text{Distance Concentration}).\tag{2.5.2}$$

This means, as the dimension grows, the relative variability (or “spread”) of the squared distances between random vectors shrinks rapidly. The distances, therefore, become increasingly concentrated around their mean, making nearly all pairwise distances look almost the same. This is the essence of the distance concentration phenomenon.

Proof of the first equation: Since $\|\mathbf{w}\|^2 = \sum_i w_i^2$, we have

$$\begin{aligned}\mathbb{E} [\|\mathbf{X} - \mathbf{Y}\|^2] &= \mathbb{E} \left[\sum_{i=1}^p (X_i - Y_i)^2 \right] \\ &= \sum_{i=1}^p \mathbb{E} [(X_i - Y_i)^2] \\ &= p \mathbb{E} [(X_1 - Y_1)^2] \quad \spadesuit \\ &= p \int_0^1 \int_0^1 (x - y)^2 p(x)p(y) dx dy \quad (\text{dropping the subscripts}) \\ &= p \int_0^1 \int_0^1 (x - y)^2 dx dy \quad (p(\cdot) = 1) \\ &= \frac{1}{6}.\end{aligned}\tag{2.5.3}$$

For $\spadesuit$: We have used the fact that all $\mathbb{E}[(X_i - Y_i)^2]$ are equal.

Proof of the second equation:

$$\begin{aligned}
 \text{Var} [\|\mathbf{X} - \mathbf{Y}\|^2] &= \text{Var} \left[\sum_{i=1}^p (X_i - Y_i)^2 \right] \\
 &= \text{Var} \left[\sum_{i=1}^p W_i \right] && (W_i := (X_i - Y_i)^2) \\
 &= \sum_{i=1}^p \text{Var} [W_i] + \sum_{i \neq j} \text{Cov} [W_i, W_j] \\
 &= \sum_{i=1}^p \text{Var} [W_i] && \spadesuit\spadesuit \quad (2.5.4) \\
 &= p \text{Var} [W_1] \\
 &= p \int_0^1 \int_0^1 \left((x - y)^2 - \frac{1}{6} \right)^2 p(x)p(y) \, dx dy \\
 &= p \int_0^1 \int_0^1 \left((x - y)^2 - \frac{1}{6} \right)^2 \, dx dy && (\text{as } p(\cdot) = 1) \\
 &= \frac{7}{180} p \approx (0.0388 \dots) p.
 \end{aligned}$$

The square root of this gives the desired result. Note that, for $\spadesuit\spadesuit$, we have used the fact that $\text{Cov}[W_i, W_j] = \diamond[(W_i - 1/6)(W_j - 1/6)] = \diamond[(W_i - 1/6)]\diamond[(W_j - 1/6)] = 0$.)

Further Exploration

1. The key papers are:

Beyer, K., Goldstein, J., Ramakrishnan, R., & Shaft, U. (1999). When is “nearest neighbor” meaningful? In *Proceedings of Database Theory—ICDT’99* (pp. 217–235).

In the discussion above, we follow

Giraud, C. (2021). Introduction to high-dimensional statistics (2nd ed.). Chapman and Hall/CRC.

2.6: Shannon's Source Coding Theorem

A follow-up discussion on supervised learning (the use of entropy).

Shannon's Source Coding Theorem provides the mathematical foundation for data compression, explaining the fundamental limits of how compactly information can be represented, which is crucial in ML for feature extraction, dimensionality reduction, deep latent-space representations, and optimizing model memory efficiency.

Shannon's Source Coding Theorem

The goal of this note is to prove the famous Shannon's Source Coding Theorem (also known as Shannon's noiseless coding theorem or the first Shannon theorem). It is one of many spectacular applications of the concepts of entropy.

Let X be a discrete random variable over a finite source alphabet $\mathcal{X} = \{x_1, x_2, \dots, x_n\}$ with probability $p_i = P(X = x_i)$. Let C be any *prefix code*¹ for X with codeword lengths ℓ_i measured in symbols from an alphabet of size a (i.e., a is the total number of alphabets; for binary codes with symbols $\{0, 1\}$, $a = 2$; for a ternary code with symbols $\{0, 1, 2\}$, $a = 3$).

For any prefix code C , the expected codeword length

$$L = \sum_{i=1}^n p_i \ell_i \quad (2.6.1)$$

satisfies

$$L \geq H_a(X) := - \sum_{i=1}^n p_i \log_a p_i, \quad (2.6.2)$$

where $H_a(X)$ is the entropy of X in units of a -ary digits.

This is a spectacular result: it precisely defines the theoretical limit of lossless data compression (the source entropy). This universal bound has revolutionized communication and storage by guiding the design of efficient, reliable coding systems used across all digital technologies.

Proof:

- Let X be a discrete random variable over symbols $\{x_1, \dots, x_n\}$ with $p_i = P(X = x_i)$. Note: $\sum_i p_i = 1$.
- Assign codewords to each symbol with lengths ℓ_i , from an alphabet of size a (E.g., if one assigns to the letter u the codeword 10100 , the length is 5).
- By Kraft's inequality for a prefix code (see below), we have

$$\sum_{i=1}^n a^{-\ell_i} \leq 1. \quad (2.6.3)$$

See the next section on a brief exploration of this beautiful result.

- Formulate the constrained optimization problem:

$$\min_{\ell_i \geq 0} \sum_{i=1}^n p_i \ell_i \quad \text{s.t.} \quad \sum_{i=1}^n a^{-\ell_i} \leq 1. \quad (2.6.4)$$

- Apply the method of Lagrange multipliers:

$$\mathcal{L}(\ell_1, \dots, \ell_n, \lambda) = \sum_{i=1}^n p_i \ell_i + \lambda \left(\sum_{i=1}^n a^{-\ell_i} - 1 \right). \quad (2.6.5)$$

We are to optimize this objective function.

- Optimization. Take partial derivatives with respect to ℓ_i and set to zero:

$$\frac{\partial \mathcal{L}}{\partial \ell_i} = p_i - \lambda a^{-\ell_i} \ln a = 0. \quad (2.6.6)$$

Solving for ℓ_i gives

$$\ell_i = -\log_a p_i + \log_a(\lambda \ln a).$$

Similarly,

$$\frac{\partial \mathcal{L}}{\partial \lambda} = 0 \Rightarrow \sum_{i=1}^n a^{-\ell_i} = 1. \quad (2.6.7)$$

Therefore, these imply

$$\sum_{i=1}^n a^{-\ell_i} = \sum_{i=1}^n \frac{p_i}{\lambda \ln a} = \frac{1}{\lambda \ln a} \sum_{i=1}^n p_i = \frac{1}{\lambda \ln a} = 1, \quad (2.6.8)$$

hence

$$\lambda = \frac{1}{\ln a}. \quad (2.6.9)$$

This implies

$$\ell_i = -\log_a p_i. \quad (2.6.10)$$

This gives the minimal expected length:

$$L^* = \sum_{i=1}^n p_i \ell_i = -\sum_{i=1}^n p_i \log_a p_i = H_a(X). \quad (2.6.11)$$

This gives the desired result: for any prefix code,

$$L \geq H_a(X). \quad (2.6.12)$$

Kraft's Inequality

Statement of Kraft's Inequality

Let $\diamond$ be a uniquely decodable code over an alphabet of a symbols ($a \geq 2$), with codewords w_i having lengths ℓ_i for $i = 1, \dots, n$. Define:

$$K(\mathcal{C}) = \sum_{i=1}^n a^{-\ell_i}. \quad (2.6.13)$$

Note: Every prefix code is uniquely decodable.

Claim: Kraft's Inequality. If $\diamond$ is uniquely decodable, then

$$K(\mathcal{C}) \leq 1. \quad (2.6.14)$$

The following is due to J.Karush in 1961.

1. Forming the power $K(\diamond)^m$:

- For any integer $m \geq 1$, consider all possible sequences of m codewords:

$$i_1, i_2, \dots, i_m. \quad (2.6.15)$$

- Together they form a string of length

$$\ell_{i_1} + \ell_{i_2} + \dots + \ell_{i_m}. \quad (2.6.16)$$

- There are n^m such ordered sequences.

2. Expanding the power:

$$K(\mathcal{C})^m = \left(\sum_{i=1}^n a^{-\ell_i} \right)^m = \sum_{i_1=1}^n \dots \sum_{i_m=1}^n a^{-(\ell_{i_1} + \dots + \ell_{i_m})}. \quad (2.6.17)$$

Each term corresponds to one possible codeword sequence of m words.

3. Grouping by total length: Let S_k be the number of sequences of m codewords whose total length is k . Then:

$$K(\mathcal{C})^m = \sum_{k=m\ell_{\min}}^{m\ell_{\max}} S_k a^{-k}, \quad (2.6.18)$$

where $\ell_{\min} = \min_i \ell_i$ and $\ell_{\max} = \max_i \ell_i$.

4. Bounding S_k : Since $\diamond$ is uniquely decodable, each sequence of m codewords maps to a distinct a -ary string of length k . The number of all possible a -ary strings of length k is a^k , hence:

$$S_k \leq a^k \quad \text{for each } k. \quad (2.6.19)$$

The idea is, since the code is uniquely decodable, one cannot have more distinct messages than the total possible different strings (this is a key component of the proof).

5. Applying this bound gives

$$K(\mathcal{C})^m = \sum_k S_k a^{-k} \leq \sum_k a^k a^{-k} = \sum_k 1 \leq m \ell_{\max}, \quad (2.6.20)$$

or

$$\frac{1}{m} K(\mathcal{C})^m \leq \ell_{\max}. \quad (2.6.21)$$

Indeed, this inequality indicates $K(\diamond) \leq 1$: If it were > 1 , the LHS of (2.6.21) would diverge as $m \rightarrow \infty$ (and could not be $\leq \ell_{\max}$). Or, by taking the m -th root, we have, for all $m \geq 1$,

$$K(\mathcal{C}) \leq (m \ell_{\max})^{1/m} \xrightarrow{m \rightarrow \infty} 1. \quad (2.6.22)$$

This gives the desired result, $K(\diamond) \leq 1$.

Further Exploration

1. On prefix codes (also called prefix-free codes)

1. A prefix code is a type of code in which *no codeword is a prefix of any other codeword*. That is, within the set of codewords, one cannot find one codeword that starts with the exact sequence of symbols of another codeword (or, say, no entire codeword is the starting segment (prefix) of another entire codeword).

2. Key Characteristics

- The prefix property guarantees that the code is instantaneously decodable: as soon as one recognizes a codeword in a stream, one knows its boundaries without needing to look ahead.
- Prefix codes are a subset of uniquely decodable codes, meaning every prefix code is uniquely decodable, but not all uniquely decodable codes are prefix codes.
- This property allows for efficient and unambiguous decoding of sequences of codewords without adding special separators or markers.

3. Applications

- Widely used in data compression algorithms (e.g. Huffman coding).
 - Enable streaming decoding without delay or ambiguity.
2. Claude Shannon introduced the Source Coding Theorem in his 1948 seminal paper “A Mathematical Theory of Communication”, laying the foundation for information theory.
 3. Kraft's inequality was first introduced by Leon G. Kraft in his MIT Master's thesis in 1949. Kraft's original work focused on prefix codes specifically and did not cover the broader class of uniquely decodable codes. In 1956, Brockway McMillan generalized Kraft's result to apply to all uniquely decodable codes, not just prefix codes. This broader result is known as the Kraft–McMillan inequality.
-

1. See the last section on the definition of prefix codes. 

3: THE PERCEPTRON & BEYOND LINEAR SEPARABILITY

3.1: Chapters 3 and 4 Summary

3.2: A Brief Introduction on Rademacher Complexity

This page titled [3: The Perceptron & Beyond Linear Separability](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by Hei-Chi Chan.

3.1: Chapters 3 and 4 Summary

- The Perceptron, the focus of SD Chap. 3, is a fundamental algorithm in ML, developed by Frank Rosenblatt in 1958. it is a linear binary classifier (and, perhaps, the simplest one).

What it does: given a data point (represented by a vector), predict a label (± 1) based on a linear transformation of the input w.r.t a threshold.

Precisely, given an input vector (a data point) $\mathbf{x} \in \mathbb{R}^n$, a weight vector $\mathbf{w} \in \mathbb{R}^n$, and a bias term $b \in \mathbb{R}$. The model predicts the class label (± 1) by:

$$\hat{y} = \text{sgn}(\mathbf{w} \cdot \mathbf{x} + b) = \begin{cases} +1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \geq 0, \\ -1 & \text{otherwise.} \end{cases} \quad (3.1.1)$$

- The model is the simplest feed-forward neural network. The model provides intuitive insights into ML mechanisms. Some call this the “hydrogen atom of neural network research.”
- See our note titled, ‘DL Notes (OER): The Perceptron’ for
 - A brief history of the model
 - The learning algorithm of the model
 - Novikoff’s Perceptron Convergence Theorem
- The Perceptron Storage Problem (PSP)
 - See SD Sect. 3.3 for details.
 - The PSP refers to the challenge of determining the capacity of a perceptron to memorize or store a set of input-output associations (patterns) within its weight parameters. It aims to address how many random input-output patterns a single-layer perceptron can perfectly classify or “store” such that the perceptron weights can separate these patterns by a hyperplane.
 - The maximum storage capacity scales roughly as $2n$, where n is the dimension of the input data space.
 - This result provides theoretical insight into perceptron limitations and informs developments in neural network design and capacity theory.
- Optimal stability and related key concepts
 - Stability Concept: The stability (κ_μ) of an example is its oriented distance from the separating hyperplane. The **stability of the perceptron**, $\kappa(w)$, is the smallest stability of any example in the dataset.
 - Optimal Perceptron: The **perceptron of optimal stability**, w_{max} , is the one that maximizes this classification margin ($\kappa(w)$).
 - MinOver Algorithm: This iterative algorithm is designed to achieve optimal stability. It performs Hebbian updates based on the currently least stable example (the one with the lowest local potential E_μ). The MinOver algorithm is guaranteed to converge to the perceptron of optimal stability if the data is linearly separable.
 - Adaline (Adaptive Linear Neuron): This historical precursor minimizes the squared norm subject to equality constraints. Its sequential learning rule is known as the Least Mean Square (LMS) method. The Adaline update is adaptive, meaning the magnitude depends on the deviation from the target value. It can result in negative embedding strengths (unlearning).
 - AdaTron (Adaptive Perceptron Algorithm): This algorithm addresses the optimal stability problem (inequality constraints). It maintains the concept of adaptive updates but prohibits the embedding strengths from becoming

negative.

- Perceptron and Support Vector Machine (SVM)
 - More details on SVM could be found in Chap. 4.
 - Both perceptron and SVM are linear classifiers, but the perceptron finds any separating hyperplane while SVM finds the one with the maximum margin between classes (which greatly improves generalization).
 - The perceptron updates weights iteratively based on misclassifications, whereas SVM solves a quadratic optimization problem to maximize the margin.
 - SVM can handle non-linearly separable data using the kernel trick; the basic perceptron cannot and requires extensions like multilayer networks for complex data.
- SVM (see SD Chap. 4)

Hard-Margin SVM (Linearly Separable Data)

- SVM aims to find the optimal hyperplane that separates data points of two classes $+1$ and -1 with the largest margin (this reduces sensitivity to noise and brings stability to the optimization, unlike the perceptron which converges to any separating hyperplane).
- The hyperplane equation is defined as:

$$\mathbf{w}^T \mathbf{x} + b = 0 \quad (3.1.2)$$

where $\mathbf{w}$ is the normal vector perpendicular to the hyperplane, and b is the bias term (the hyperplane's offset from the origin).

- For a correctly classified point $(\mathbf{x}_i, y_i)$, the constraint is:

$$y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 \quad (3.1.3)$$

ensuring points lie on the correct side of the margin.

- The margin M , the distance between the hyperplane and the closest data points, is given by:

$$M = \frac{2}{\|\mathbf{w}\|}. \quad (3.1.4)$$

- The optimization problem to maximize the margin transforms into minimizing:

$$\min_{\mathbf{w}, b} \frac{1}{2} \|\mathbf{w}\|^2 \quad (3.1.5)$$

subject to the constraints $y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1$ for all i . One optimizes $\|\mathbf{w}\|^2$ instead of $\|\mathbf{w}\|$ to avoid the square root.

- Solving the primal optimization problem directly is difficult for high-dimensional data. One could use Lagrange Multipliers (α_i) and transform it to its dual form (this is done as follows: solve the critical point of the Lagrangian $\mathcal{L}$ and substitute these back into $\mathcal{L}$ to obtain the dual objective). The final product is

$$\begin{aligned} \max_{\alpha} \quad & \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j (\mathbf{x}_i^T \mathbf{x}_j) \\ \text{subject to} \quad & \sum_i \alpha_i y_i = 0, \quad \alpha_i \geq 0 \end{aligned} \quad (3.1.6)$$

- Support vectors are data points with $\alpha_i > 0$; these points define the margin and the decision boundary (hence the name).
- The dot product form is the foundation for kernel functions (see below).

Soft-Margin SVM (Non-Separable Data)

- The hard margin SVM requires perfect linear separability, allowing no misclassification and strictly maximizing the margin. It fails when data is noisy or not linearly separable.
- Soft margin SVM is designed to deal with the situations where the hard margin SVM fails. It introduces slack variables $\xi_i \geq 0$ to allow some data points to be within the margin or misclassified (balancing margin maximization with classification error minimization).
- The constraints change to:

$$y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 - \xi_i \quad (3.1.7)$$

where ξ_i quantifies the degree of margin violation.

- The optimization objective becomes:

$$\min_{\mathbf{w}, b, \xi} \left(\frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^n \xi_i \right) \quad (3.1.8)$$

Here, $C > 0$ is a regularization parameter controlling the trade-off between maximizing margin (minimizing $\|\mathbf{w}\|^2$) and penalizing classification errors (sum of slack variables).

- When C is very large (prioritizing minimizing slack violations), the model approximates hard margin SVM; smaller C (emphasizing large margin) allows more violations, giving a softer margin and better generalization.
- The dual form also adapts, with constraints on multipliers α_i bounded by:

$$0 \leq \alpha_i \leq C \quad (3.1.9)$$

enforcing the penalty bounds.

- Slack variables ξ_i give robustness to noise and outliers by permitting some misclassifications without overfitting, crucial for non-linearly separable data.

Kernels

- Kernel methods enable SVM to handle non-linearly separable data by implicitly mapping data points $\mathbf{x} \in \mathbb{R}^n$ into a higher-dimensional space $\phi(\mathbf{x}) \in \mathbb{R}^m$ (with $m > n$), where linear separation is possible, without explicitly computing $\phi(\mathbf{x})$.
- The kernel function $K(\mathbf{x}, \mathbf{x}')$ computes the dot product in the new feature space:

$$K(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^T \phi(\mathbf{x}') \quad (3.1.10)$$

allowing efficient calculation of inner products in high-dimensional spaces.

- Common kernel functions include:

- Linear kernel (giving the original SVM):

$$K(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T \mathbf{x}' \quad (3.1.11)$$

- Polynomial kernel of degree d :

$$K(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^T \mathbf{x}' + r)^d \quad (3.1.12)$$

where r is a bias term.

- Radial Basis Function (RBF) or Gaussian kernel:

$$K(\mathbf{x}, \mathbf{x}') = \exp(-\gamma \|\mathbf{x} - \mathbf{x}'\|^2) \quad (3.1.13)$$

where $\gamma > 0$ controls the spread of the kernel.

- Sigmoid kernel:

$$K(\mathbf{x}, \mathbf{x}') = \tanh(\gamma \mathbf{x}^\top \mathbf{x}' + r) \quad (3.1.14)$$

where γ and r control slope and bias.

- Using kernels: simply replacing all dot products in the dual form of the SVM optimization problem (see ([03z=02])) with the kernel function and solving

$$\begin{aligned} \max_{\alpha} \quad & \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) \\ \text{subject to} \quad & \sum_i \alpha_i y_i = 0, \quad \alpha_i \geq 0 \end{aligned} \quad (3.1.15)$$

- Kernel trick avoids explicit computation of $\phi(\mathbf{x})$ which may be infinite-dimensional, leading to efficient training and prediction.
- Feature scaling is essential before applying kernels to ensure balanced distance calculations, especially for RBF kernels (which depend on distance metrics).

A brief history of SVM:

- SVM was invented in 1964 by Vladimir Vapnik and Alexey Chervonenkis when they developed the foundational algorithm as part of their statistical learning theory for pattern recognition.
- The modern breakthrough came in 1992, when Bernhard Boser, Isabelle Guyon, and Vapnik introduced the kernel trick.
- In the early 90s, Corinna Cortes and Vapnik extended this with the soft-margin formulation to handle noisy, non-separable data.
- SVM gained prominence in the early-to-mid 1990s when neural networks were stalled by technical issues that led to disappointing performance. Kernel SVMs were the dominant methods for tasks like image classification thanks to strong theoretical guarantees and efficiency.
- A team from NEC Labs America and UIUC used large-scale SVMs to win the inaugural 2010 ImageNet Challenge.
- While DL gained an apparently unstoppable and ever-growing domain since 2012 (after AlexNet), SVMs remain useful—especially in areas where data are limited, interpretability matters, or computational efficiency is key.
- More on SVMs:
 - Vapnik, V. N. (1995). *The nature of statistical learning theory*. Springer.
 - Schölkopf, B., & Smola, A. J. (2002). *Learning with kernels: Support vector machines, regularization, optimization, and beyond*. MIT Press.

3.2: A Brief Introduction on Rademacher Complexity

These are notes on Chapter 3 on generalization bounds: they are theoretical statements about a model's ability to perform accurately on new, unseen data, beyond the specific examples used during training. These bounds aim to quantify the difference between a model's performance on the training data (empirical risk/error) and its performance on the true, underlying data distribution (true risk/error). The Rademacher complexity plays a crucial role in quantifying generalization bounds.

Introduction

In AI/ML, we are interested in how well a model performs on unseen data, and this ability is called generalization. The Rademacher complexity is a measure that quantifies how flexible or complex a model or class of models is, by evaluating how well it can fit random noise. It indicates how likely a model is to generalize well (i.e., perform accurately on new, unseen examples rather than just the training data). Models with lower Rademacher complexity are less likely to overfit the training data and tend to generalize better to real-world situations.

In this note, we will first define the Rademacher complexity. Then we will prove a key result relating generalization bounds and the Rademacher complexity.

In the final section, we outline an explicit calculation related to regression models.

Defining Rademacher Complexity

Rademacher complexity: it is a measure of how rich or complex a class of functions is, quantifying how well these functions can fit random noise.

- Setup:
 - Let $\mathcal{F}$ be a class of real-valued functions $f: \mathcal{Z} \rightarrow \mathbb{R}$.¹ The space $\mathcal{Z}$ is known as the input space. E.g., if each z is a n -dimensional vector, then $z \in \mathbb{R}^n = \mathcal{Z}$.
 - Let $S = \{z_1, z_2, \dots, z_m\}$ be a sample of m data points drawn i.i.d. from some distribution $\mathcal{Z}$. Sometimes this is written as $S \sim D^m$.
- Rademacher random variables:
 - $\sigma_1, \sigma_2, \dots, \sigma_m$ are independent random variables.
 - Each σ_i takes values ± 1 with equal probability $1/2$.
- Empirical Rademacher complexity (on sample S) of $\mathcal{F}$: it is defined as follows:

$$\widehat{\text{Rad}}_S(\mathcal{F}) := \mathbb{E}_\sigma \left[\sup_{f \in \mathcal{F}} \frac{1}{m} \sum_{i=1}^m \sigma_i f(z_i) \right] \quad (3.2.1)$$

- The sum, $(1/m) \sum_{i=1}^m \sigma_i f(z_i)$, measures the correlation between the Rademacher variables, σ_i , and the output $f(z_i)$.
- The supremum means it looks for the maximum (or, the best possible) correlation with the random noise (represented by the Rademacher variables) that any function within the entire class $\mathcal{F}$ can achieve. It finds the function in $\mathcal{F}$ that is best at "fitting" this randomly generated noise.
- The expectation $\mathbb{E}_\sigma$ is taken over all possible combinations of the Rademacher variables. The empirical Rademacher complexity is designed to capture the *expected* ability of the function class to fit random noise.
- A brief summary: $\widehat{\text{Rad}}_S(\mathcal{F})$ measures how well, on average, a set of functions (a class of functions, not just one function)

can fit random noise on a given fixed sample of data points.

- Rademacher complexity. This is also known as the Population (or Expected) Rademacher complexity:

$$\text{Rad}_m(\mathcal{F}) := \mathbb{E}_S \left[\widehat{\text{Rad}}_S(\mathcal{F}) \right] \quad (3.2.2)$$

The expectation $\mathbb{E}_S$ is over all samples S of size m drawn i.i.d. from $\mathbb{D}$. It measures the expected or typical richness (capacity) of a function class with respect to the data distribution, rather than just its behavior on one fixed training sample.

- Interpretation:

High Rademacher complexity means the function class can fit random noise well, indicating higher complexity and potential overfitting.

A Key Result regarding Rademacher Complexity

In this section, we prove a result that illustrates how the generalization gap is related to the the Rademacher complexity.

[Theorem 3.1] Let

- $\mathcal{G}$ be a family of functions mapping from $\mathbb{Z}$ to $[0, 1]$.
- Let D be a distribution of $\mathbb{Z}$ (so $z \sim D$ for $z \in \mathbb{Z}$).

Then, for any $\delta > 0$, with probability at least $1 - \delta$, the following holds for each $g \in \mathcal{G}$:

$$\mathbb{E}_D[g] \leq \frac{1}{m} \sum_{i=1}^m g(z_i) + 2\text{Rad}_m(\mathcal{G}) + \sqrt{\frac{\log(1/\delta)}{2m}}. \quad (3.2.3)$$

Note:

- $\mathbb{E}_D[g]$ is the mean of g over the true distribution D : $\mathbb{E}_D[g] := \int g(z) dD(z)$. Whereas $\frac{1}{m} \sum_i g(z_i)$ is the empirical mean of g , which is an approximation of $\mathbb{E}_D[g]$.
- This theorem quantifies the difference between expected mean and empirical mean (an example of the generalization gap) in terms of the Rademacher complexity of the hypothesis class, $\mathcal{G}$. We note that smaller Rademacher complexity implies stronger generalization guarantees (i.e., smaller generalization gaps).

Proof. Let $S = \{z_1, \dots, z_m\}$. Define

$$\widehat{\mathbb{E}}_S[g] := \frac{1}{m} \sum_{i=1}^m g(z_i). \quad (3.2.4)$$

This is the first term of the RHS of (3.2.3).

- Step 0. Note that, by the definition of supremum, we have

$$\mathbb{E}_D[g] - \widehat{\mathbb{E}}_S[g] \leq \underbrace{\sup_{g \in \mathcal{G}} (\mathbb{E}_D[g] - \widehat{\mathbb{E}}_S[g])}_{:= \Phi(S)}. \quad (3.2.5)$$

- Step 1. We use McDiarmid's inequality to bound $\Phi(S)$ (see the next section for details): for any $\delta > 0$, with probability at least $1 - \delta$, the following is true

$$\Phi(S) \leq \mathbb{E}_S[\Phi(S)] + \sqrt{\frac{\log(1/\delta)}{2m}}. \quad (3.2.6)$$

Proof. To this end, we need to check the bounded difference property is satisfied. Consider a sample S' : it differs from S at exactly one data entry (say, at the i -th data entry). Then we have

$$\begin{aligned}
 \Phi(S) - \Phi(S') &= \sup_{g \in \mathcal{G}} \left(\mathbb{E}_D[g] - \widehat{\mathbb{E}}_S[g] \right) - \sup_{g \in \mathcal{G}} \left(\mathbb{E}_D[g] - \widehat{\mathbb{E}}_{S'}[g] \right) \\
 &\leq \sup_{g \in \mathcal{G}} \left(\widehat{\mathbb{E}}_{S'}[g] - \widehat{\mathbb{E}}_S[g] \right) \\
 &= \sup_{g \in \mathcal{G}} \frac{g(z'_i) - g(z_i)}{m} \leq \frac{1}{m}.
 \end{aligned} \tag{3.2.7}$$

To obtain the last inequality, we have used the fact that g maps to the interval $[0, 1]$. Hence,

$$|\Phi(S) - \Phi(S')| \leq \frac{1}{m}, \tag{3.2.8}$$

and the bounded difference property is satisfied. Now we apply McDiarmid's inequality with $c_i = 1/m$. With any $\epsilon > 0$,

$$\text{Prob}(\Phi(S) - \mathbb{E}_S[\Phi(S)] \geq \epsilon) \leq \exp(-2m\epsilon^2). \tag{3.2.9}$$

Setting $\delta := \exp(-2m\epsilon^2)$, this is the same as, for any $\delta > 0$, with probability at least $1 - \delta$, the following is true:

$$\Phi(S) \leq \mathbb{E}_S[\Phi(S)] + \sqrt{\frac{\log(1/\delta)}{2m}}, \tag{3.2.10}$$

the desired result. $\square$

- Step 2. The key result of this step is

$$\mathbb{E}_S[\Phi(S)] \leq 2\text{Rad}_m(\mathcal{G}). \tag{3.2.11}$$

Proof.

$$\begin{aligned}
 \mathbb{E}_S[\Phi(S)] &= \mathbb{E}_S \left[\sup_{g \in \mathcal{G}} \left(\mathbb{E}_D[g] - \widehat{\mathbb{E}}_S[g] \right) \right] \\
 &= \mathbb{E}_S \left[\sup_{g \in \mathcal{G}} \mathbb{E}_{S'} \left(\widehat{\mathbb{E}}_{S'}[g] - \widehat{\mathbb{E}}_S[g] \right) \right] \\
 &\leq \mathbb{E}_{S, S'} \left[\sup_{g \in \mathcal{G}} \left(\widehat{\mathbb{E}}_{S'}[g] - \widehat{\mathbb{E}}_S[g] \right) \right] \\
 &= \mathbb{E}_{S, S'} \left[\sup_{g \in \mathcal{G}} \frac{1}{m} \sum_{i=1}^m (g(z'_i) - g(z_i)) \right] \\
 &= \mathbb{E}_{\sigma, S, S'} \left[\sup_{g \in \mathcal{G}} \frac{1}{m} \sum_{i=1}^m \sigma_i (g(z'_i) - g(z_i)) \right] \\
 &\leq \mathbb{E}_{\sigma, S'} \left[\sup_{g \in \mathcal{G}} \frac{1}{m} \sum_{i=1}^m \sigma_i g(z'_i) \right] + \mathbb{E}_{\sigma, S} \left[\sup_{g \in \mathcal{G}} \frac{1}{m} \sum_{i=1}^m (-\sigma_i) g(z_i) \right] \\
 &= 2\mathbb{E}_{\sigma, S'} \left[\sup_{g \in \mathcal{G}} \frac{1}{m} \sum_{i=1}^m \sigma_i g(z'_i) \right] = 2\text{Rad}_m(\mathcal{G}).
 \end{aligned} \tag{3.2.12}$$

Note:

- For the 2nd equality: $S' = \{z'_1, \dots, z'_m\}$ is another set of sample of m elements, drawn i.i.d. from the same distribution (i.e., each $z'_i \sim D$). S' is called the ghost sample. We have used

$$\begin{aligned}
 \mathbb{E}_{S'} \left[\widehat{\mathbb{E}}_{S'}[g] \right] &= \mathbb{E}_{S'} \left[\frac{1}{m} \sum_i g(z'_i) \right] = \frac{1}{m} \sum_i \mathbb{E}_D[g] \\
 &= \left(\frac{1}{m} \sum_i 1 \right) \mathbb{E}_D[g] = \mathbb{E}_D[g],
 \end{aligned} \tag{3.2.13}$$

as S and S' are independent. Similarly, one can show that $\mathbb{E}_S[g] = \mathbb{E}_{S'}[\widehat{\mathbb{E}}_S[g]]$.

- For the 1st inequality: use the sub-additivity of the supremum; the supremum of an expectation is at most the expectation of the supremum.
- For the 4th equality: this is perhaps the most tricky part of the proof. Taking the average over all σ_i , does not change the expectation in the 3rd equality. The key idea is, the distribution of $g(z') - g(z)$ is the same as that of $\pm(g(z') - g(z))$ (as z and z' are independently and symmetrically distributed). To illustrate the idea, let us work out the situation with one pair of RVs, X and X' that have the same distribution (say, given by the probability density $p(\cdot)$), with only a single Rademacher variable. Let $\Theta : \mathbb{R} \rightarrow \mathbb{R}$ be a function, and we want to show

$$\mathbb{E}_{X, X'} [\Theta(g(X) - g(X'))] = \mathbb{E}_{X, X', \sigma} [\Theta(\sigma(g(X) - g(X')))] . \quad (3.2.14)$$

To see this: the LHS is given by

$$\mathbb{E}_{X, X'} [\Theta(g(X) - g(X'))] = \iint \Theta(g(x) - g(x')) p(x)p(x') dx dx' . \quad (3.2.15)$$

The RHS is given by

$$\begin{aligned} & \mathbb{E}_{X, X', \sigma} [\Theta(\sigma(g(X) - g(X')))] \\ &= \text{Prob}(\sigma = 1) \iint \Theta(g(x) - g(x')) p(x)p(x') dx dx' \\ & \quad + \text{Prob}(\sigma = -1) \iint \Theta(-g(x) + g(x')) p(x)p(x') dx dx' \\ &= \frac{1}{2} \iint \Theta(g(x) - g(x')) p(x)p(x') dx dx' \\ & \quad + \frac{1}{2} \iint \Theta(-g(x) + g(x')) p(x)p(x') dx dx' \\ &= \frac{1}{2} \iint \Theta(g(x) - g(x')) p(x)p(x') dx dx' \\ & \quad + \frac{1}{2} \iint \Theta(-g(u') + g(u)) p(u')p(u) du' du \\ &= \iint \Theta(g(x) - g(x')) p(x)p(x') dx dx' \quad (= \text{the LHS}). \end{aligned} \quad (3.2.16)$$

- For the last line: this is because both $-\sigma_i$ and σ_i have identical distribution, and both expectations equal $\text{Rad}_m(\diamond)$.

This completes the proof. $\square$

- Step 3. Racing to the finish line. Together these imply the desired result.

1. For more info: see

Mohri, M., Rostamizadeh, A., & Talwalkar, A. (2018). *Foundations of machine learning* (2nd ed.). MIT Press.

2. McDiarmid's inequality: it is a concentration bound for functions of independent random variables that satisfy a bounded differences condition. Let $X_1, X_2, \dots, X_m$ be independent random variables with $X_i \in \diamond_i$, and let $f : \diamond_1 \times \dots \times \diamond_m \rightarrow \mathbb{R}$ satisfy, for each i :

$$\sup_{x_1, \dots, x_m, x'_i} |f(x_1, \dots, x_i, \dots, x_m) - f(x_1, \dots, x'_i, \dots, x_m)| \leq c_i. \quad (3.2.17)$$

Then, for any $\varepsilon > 0$,

$$\begin{aligned}\mathbb{P}(f(X_1, \dots, X_m) - \mathbb{E}[f(X_1, \dots, X_m)] \geq \varepsilon) &\leq \exp\left(-\frac{2\varepsilon^2}{\sum_{i=1}^m c_i^2}\right), \\ \mathbb{P}(f(X_1, \dots, X_m) - \mathbb{E}[f(X_1, \dots, X_m)] \leq -\varepsilon) &\leq \exp\left(-\frac{2\varepsilon^2}{\sum_{i=1}^m c_i^2}\right).\end{aligned}\tag{3.2.18}$$

See Mohri et al. (2018) for a proof.

3. The empirical Rademacher complex for linear regression models

Consider the following function class (with M, W, X being constants): for $\mathbf{x} \in \mathbb{R}^d$ such that $\|\mathbf{x}\|_2 \leq X$, define

$$\mathcal{F} = \{\mathbf{x} \mapsto \mathbf{w} \cdot \mathbf{x} : \|\mathbf{w}\|_2 \leq W\}$$

Let $S = \{\mathbf{x}_1, \dots, \mathbf{x}_m\}$ be m data points. Then we have

$$\widehat{\text{Rad}}_S(\mathcal{F}) \leq \frac{WX}{\sqrt{m}}.\tag{3.2.19}$$

To see this:

$$\begin{aligned}\widehat{\text{Rad}}_S(\mathcal{F}) &= \mathbb{E}_{\sigma} \left[\sum_{i=1}^m \sigma_i f(\mathbf{x}_i) \right] \quad \text{(by definition)} \\ &= \mathbb{E}_{\sigma} \left[\sum_{i=1}^m \sigma_i (\mathbf{w} \cdot \mathbf{x}_i) \right] \\ &= \mathbb{E}_{\sigma} \left[\sum_{i=1}^m \sigma_i \sum_{j=1}^d w_j x_{ij} \right] \\ &= \sum_{j=1}^d w_j \mathbb{E}_{\sigma} \left[\sum_{i=1}^m \sigma_i x_{ij} \right] \\ &\leq \sum_{j=1}^d |w_j| \mathbb{E}_{\sigma} \left[\sum_{i=1}^m |x_{ij}| \right] \\ &\leq \sum_{j=1}^d |w_j| \sqrt{\sum_{i=1}^m x_{ij}^2} \quad \text{(by Cauchy-Schwartz ineq.)} \\ &\leq \frac{W}{\sqrt{m}} \sqrt{\sum_{i=1}^m \sum_{j=1}^d x_{ij}^2} \\ &= \frac{WX}{\sqrt{m}}.\end{aligned}$$

1. In ML/AI, f typically represents a ML model: writing $f(z)$ means we input z and the model outputs the value $f(z)$. In the context of the Rademacher complexity, we also consider the *class* of the functions to which f belongs.

E.g., Consider the example of linear regression:

- Each function (representing a regression model) is of the form

$$f(z) = a \cdot z + b,\tag{3.2.20}$$

with a and b being real numbers.

- The function class here is the set of all such linear functions with any real values of a and b :

$$\mathcal{F} = \{f : f(z) = a \cdot z + b \mid a, b \in \mathbb{R}\}.\tag{3.2.21}$$

- Geometrically, this means the function class includes all possible straight lines one can draw by varying a and b .
- In ML terms, $\mathcal{F}$ represents the entire space of linear models the learning algorithm can choose from to fit the data. While one specific model is just one line (one function f), the function class is all the lines one could possibly fit by changing a and b .



4: Beyond Linear Separability

Please see the previous chapter for details.

5: FEED-FORWARD NETWORKS FOR REGRESSION AND CLASSIFICATION

5.1: Chapter 5 Summary

5.2: Universal Approximation Theorem

5.3: Attention Mechanism/Transformers

This page titled [5: Feed-forward Networks for Regression and Classification](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by Hei-Chi Chan.

5.1: Chapter 5 Summary

Based on SD Chapter 5.

1. Sect. 5.1. Deep Learning(DL)/Neural Networks (NNs) as Non-linear Function Approximators

Architecture and Input-Output Relation: Neural Networks (NNs) consist of an input layer, multiple hidden layers, and an output layer. Each layer is composed of neurons (nodes) connected to the next layer. When the NNs consist of multiple layers, we call them deep neural networks (DNNs).

Briefly:

- Each neuron computes a weighted sum of its inputs, then applies a nonlinear activation function to introduce nonlinearity into the model.
- The (overall) function represented by a DNN is a complex nonlinear mapping from input space to output space, building up through successive layers.
- The successive layers of nonlinear functions (with parameters) aim to approximate the highly complex relationships between the input data and the output.
- Here is a common schematic that illustrates the architecture of NNs, see, e.g., Neural Network (Wikipedia).

Universal Approximation:

NNs with at least one hidden layer and appropriate nonlinear activation functions are universal approximators; they can approximate any continuous function on compact subsets of the input space to arbitrary precision given sufficient neurons.

More layers usually mean stronger expressive power, allowing networks to approximate complex functions more efficiently than shallow networks with the same number of parameters.

The ability to approximate composition of functions exponentially improves error decay rates compared to polynomial rates in shallow models. Universal approximation guarantees do not specify network size but support the theoretical foundation for using DNNs as powerful function approximators in diverse applications.

See the last section for more details.

2. Sect. 5.2. Gradient Computation and Backpropagation, Optimization Methods, and Current Developments

Gradient Computation and Backpropagation of Error (History):

- Backpropagation (BP) is an efficient algorithm for computing gradients of the loss function with respect to network weights using the chain rule of calculus. This enables multi-layer networks to be trained.
- The concept goes back to the 1960s. E.g., Seppo Linnainmaa discusses “reverse mode automatic differentiation” in his 1970 master’s thesis. See also Paul Werbos’ work in 1974, in which he proposed backpropagation applied to multilayer perceptrons.
- BP gained prominence after the seminal 1986 paper by Rumelhart, Hinton, and Williams demonstrating practical effectiveness. The method enabled training of deep layered architectures by efficiently propagating error derivatives backward through layers, overcoming limitations of earlier models.
- Yann LeCun used BP in convolutional neural networks (CNNs) during the late 1980s for digit recognition. This was a key milestone in demonstrating practical industrial use.

Gradient Descent, variations, and options

- Batch Gradient Descent (BGD): it involves computing the gradient of the loss function using the entire training dataset before updating the parameters once per iteration. While it provides a stable and accurate estimate of the gradient, the method is computationally expensive and slow for large datasets, making it impractical for modern DL models.
- Stochastic Gradient Descent (SGD): it updates the weights incrementally using the gradient from a single or a small batch of training samples, enabling frequent updates. This method reduces computation per update and introduces noise in gradient estimates that can help escape local minima. It often accelerates convergence in practice. Variants include mini-batch SGD, which balances stability and speed by using small subsets of data, and momentum-based optimizers that further improve convergence.
- Other Options: Adaptive optimizers like Adam, RMSprop, and AdaGrad automatically adjust learning rates per parameter, enhancing training stability and speed on complex networks. Techniques such as learning rate scheduling, warm restarts, and gradient clipping improve convergence and prevent issues like exploding or vanishing gradients. Research continues into second-order methods, meta-learning optimizers, and decentralized or federated training techniques to scale training further. Parallelization on GPUs and TPUs combined with these optimization advances drives the ongoing industrial and research revolution in DL applications.

3. Sect. 5.3. Loss Functions

Loss Functions for Regression:

- Mean Squared Error (MSE)/L2 Loss:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2. \quad (5.1.1)$$

Here: where y_i is the true target and $\hat{y}_i$ is the prediction for the i -th sample. This Measures the average of squared differences between predicted and true values. Sensitive to outliers.

- Mean Absolute Error (MAE)/L1 Loss:

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|. \quad (5.1.2)$$

This measures average absolute differences; less sensitive to outliers than MSE. Provides robustness for data with outliers but with less smooth gradients.

- Huber Loss: Define

$$L_{\delta}(y_i - \hat{y}_i) = \begin{cases} \frac{1}{2}(y_i - \hat{y}_i)^2 & \text{if } |y_i - \hat{y}_i| \leq \delta \\ \delta|y_i - \hat{y}_i| - \frac{1}{2}\delta^2 & \text{if } |y_i - \hat{y}_i| > \delta \end{cases} \quad (5.1.3)$$

and the Huber Loss function is given by $\frac{1}{n} \sum_{i=1}^n L_{\delta}(y_i - \hat{y}_i)$. This combines MSE and MAE by using quadratic penalty for small errors and linear for larger errors. Robust to outliers and differentiable, balancing benefits of MSE and MAE.

- Log-Cosh Loss:

$$\text{Log-Cosh Loss} = \frac{1}{n} \sum_{i=1}^n \log(\cosh(y_i - \hat{y}_i)). \quad (5.1.4)$$

Similar to MSE but smoother, less affected by outliers.

Loss Functions for Classification:

- Binary Cross-Entropy Loss (Log Loss): Measures difference between predicted probabilities and actual binary labels. Suitable for binary classification; encourages probabilistic outputs close to true labels.
- Categorical Cross-Entropy Loss: Generalization of binary cross-entropy for multi-class classification. Measures

discrepancy between predicted class probabilities and true one-hot labels.

- Hinge Loss: Used mainly in SVMs for maximum-margin classification. Penalizes predictions on the wrong side of the margin; promotes large margin classification.
- Focal Loss: Variant of cross-entropy that focuses training on hard, misclassified examples. Useful in imbalanced classification scenarios.

Choice of loss function depends on task nature, noise/outliers in data, and model requirements for probabilistic interpretation or margin maximization.

4. Sect. 5.4. Activation Functions Used in DL

Common Activation Functions with Pros and Cons:

- ReLU (Rectified Linear Unit):

Pros: Computationally efficient, mitigates vanishing gradient problem, enables sparse activation.

Cons: Can suffer from “dying ReLU” where neurons output zero for all inputs, non-differentiable at zero.

- Sigmoid:

Pros: Maps input to $(0,1)$, useful for binary classification output layers.

Cons: Prone to vanishing gradients in deep networks, slower convergence.

- Tanh (Hyperbolic Tangent):

Pros: Output ranges from -1 to 1, zero-centered which helps optimization compared to sigmoid.

Cons: Also suffers from vanishing gradient for large inputs.

- Softmax:

Pros: Converts network outputs into probability distributions; used in multi-class classification tasks.

Cons: Computationally expensive for many classes, assumes mutually exclusive classes.

- Leaky ReLU:

Pros: Fixes dying ReLU problem by allowing small non-zero gradient for negative inputs.

Cons: Adds hyperparameters; performance gains depend on tuning.

- ELU (Exponential Linear Unit):

Pros: Combines benefits of ReLU with smooth output for negative values; helps reduce bias shift.

Cons: More computationally expensive than ReLU.

- Swish:

Pros: Self-gated, smooth nonlinearity that often outperforms ReLU on complex tasks.

Cons: Slightly increased computational complexity.

- GELU (Gaussian Error Linear Unit):

Pros: Used in transformer models; offers smooth activation that improves performance in NLP tasks.

Cons: More complex calculations than ReLU.

Choosing activation functions depends on the network architecture, task (classification, regression), and practical trade-offs like training speed and gradient behavior.

5. Sect. 5.5. Convolutional Neural Networks (CNNs)

CNN Architecture Overview:

- CNNs consist of input, convolutional layers, activation functions (usually ReLU), pooling layers, fully connected layers, and output layers.
- They specialize in processing grid-like data such as images by exploiting spatial relationships between pixels.
- The convolutional layer applies a set of learnable filters (kernels) that slide over the input volume spatially. Each filter is a small matrix that convolves (performs a sliding dot product) with a local region of the input at each position, producing an activation map.
- This process extracts feature maps—local patterns such as edges or textures—by activating when specific features are detected at different spatial positions.
- Parameter sharing: this means the same filter weights apply across the entire input spatial dimension, enabling translation invariance.
- Hyperparameters include the number of filters (depth), filter size (spatial extent), stride (step length of sliding), and padding (zero-padding around input edges).
- Multiple convolution layers stacked deepen feature extraction, starting from simple edges in early layers to complex shapes and objects in deeper layers.
- These models can be designed to exploits spatial hierarchy by learning increasingly abstract features layer by layer.
- Translation invariance helps recognize patterns regardless of location shifts.

Some more details:

- Convolution Operation: The core of CNNs is 2D discrete convolution, where a kernel K of size $m \times n$ slides over input image I to produce feature map O at position (i, j) :

$$O(i, j) = \sum_{u=0}^{m-1} \sum_{v=0}^{n-1} I(i+u, j+v) \cdot K(u, v). \quad (5.1.5)$$

This extracts local features like edges via weighted sums.

- **Pooling (Max):** Downsamples feature maps by taking max in kernel window, e.g., 2×2 max pooling:
 $O(i, j) = \max_{u, v \in [0, 1]} I(2i+u, 2j+v)$,
 providing translation invariance and reducing parameters.
- **Fully Connected Layer:** Flattened features $\mathbf{x}$ transform via $\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$, where $\mathbf{W}$ is weights, $\mathbf{b}$ bias, followed by activation like ReLU: $f(z) = \max(0, z)$.
- For a schematic illustration of a CNN filter: see Convolutional neural network (Wikipedia).

6. Sect. 5.6. Some Advanced Developments in DL

Transformer models revolutionized deep learning since their introduction in the 2017 paper “Attention Is All You Need,” replacing sequential recurrent networks with parallelizable self-attention mechanisms that capture long-range dependencies efficiently. This shift enabled scalable training on massive datasets, powering breakthroughs in natural language processing (NLP), computer vision, and beyond, leading to LLMs such as GPT.

See the additional notes in the next section for details.

Leading proprietary models (as of late 2025)

- OpenAI GPT-5, with specific versions like GPT-5 Pro and GPT-5.1 Thinking/Instant. They offer high-end reasoning, coding, and multimodal capabilities, making it superior for complex tasks, long-context understanding, and agentic functions compared to its predecessors.
- Anthropic's most advanced Claude models are the Claude 4.5 family, with Claude Opus 4.5 being their most intelligent, powerful model for complex coding, agents, and reasoning, while Claude Sonnet 4.5 offers the best balance of speed, cost, and intelligence for most production use cases, and Claude Haiku 4.5 is the fastest for instant responses. Note: "Haiku" signifies speed and efficiency, drawing parallels to the concise, impactful nature of Japanese poetry.
- Google's Gemini model is Gemini 3 Pro. An enhanced reasoning mode called Gemini 3 Deep Think is available to premium subscribers.
- The most advanced xAI Grok models available are from the Grok 4 family (the base Grok 4, the specialized Grok 4 Heavy, and the efficient Grok 4 Fast). A minor update, Grok 4.1, was released in November 2025 with further refinements.

Leading Open-Source Models:

- DeepSeek-V3.2 series and the specialized DeepSeek-V3.2-Special model (for "long thinking" and complex algorithmic reasoning tasks). Both use a Mixture-of-Experts (MoE) architecture with a DeepSeek Sparse Attention (DSA) mechanism.
- Meta's Llama 4 Scout and Llama 4 Maverick.
- Mistral AI's Mistral Large 3, part of the new Mistral 3 family of models.
- We note that Mamba is a new, efficient AI architecture that challenges traditional Transformers (like those in GPT/Gemini) by replacing expensive self-attention with State Space Models (SSMs) for linear scaling with sequence length.

1. Cybenko's Universal Approximation Theorem: See the next section for more details.
2. A quick overview on transformer models with attention mechanism

Transformer models are the backbone of almost all LLMs (like GPT). They start with converting input words into dense vector representations called embedding.

These vectors, combined with positional encodings for tracking word order, are signals that flow through a stack of transformer blocks (each with its own unique set of learned parameters). These blocks progressively build richer contextual understanding.

After passing through multiple such blocks (e.g., between 10+ to 90+ layers), the final output representation/vectors feeds into a softmax to predict, e.g., probabilities for the next word.

Below is a simplified view of this process. See

Vaswani, A. et al. (2017). Attention is all you need. In *Advances in neural information processing systems* (pp. 5998–6008).

- **Input Embeddings + Positional Encoding:** Token embeddings X summed with positional encodings P to retain order:

$$Z_0 = X + P, \quad (5.1.6)$$

where $X \in \mathbb{R}^{n \times d}$, $P \in \mathbb{R}^{n \times d}$, where n = sequence length, d = embedding/model dimension.

One possible formulation for P is given by

$$P_{i,k} = \begin{cases} \sin\left(\frac{i}{10000^{k/d}}\right) & \text{if } k \text{ is even} \\ \cos\left(\frac{i}{10000^{(k-1)/d}}\right) & \text{if } k \text{ is odd} \end{cases} \quad (5.1.7)$$

Here, i denotes the position in the input sequence.

- **Multi-Head Self-Attention:** Attention is applied across parallel heads, with output concatenated and projected. For a single-head attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V, \quad (5.1.8)$$

where

$$Q = ZW^Q, \quad K = ZW^K, \quad V = ZW^V. \quad (5.1.9)$$

Here, Q (Query), K (Key), and V (Value) are matrices created by linearly transforming the input sequence data to determine relevance and extract weighted information.¹ Note: all the W^* matrices are learned parameters of the model.

Instead of running just one attention, LLMs use multi-head attention. It splits the process into several parallel “heads,” each with its own unique focus (and learned parameters). The goal is to create a richer and more comprehensive understanding of the input data.

Let $\text{head}_i = \text{Attention}(Q_i, K_i, V_i)$. Then

$$\text{MultiHead}(Z) := \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O. \quad (5.1.10)$$

Here, Concat means stacking their resulting matrices side-by-side (a simple horizontal stacking). The matrix W^O (W-Output): it is a learned matrix for dimensionality management (projecting the high dim. concatenated output into the expected dimension required for further processing).²

- **Add & Norm (Post-Attention):** Residual connection with layer normalization:

$$Z_1 = \text{LayerNorm}(Z_0 + \text{MultiHead}(Z_0)). \quad (5.1.11)$$

What for?

- **LayerNorm**

For input vector $x = [x_1, x_2, \dots, x_H]$:

$$y_i = \gamma \cdot \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \quad (5.1.12)$$

where

- $\mu = \frac{1}{H} \sum_{i=1}^H x_i$ (mean across features)
- $\sigma^2 = \frac{1}{H} \sum_{i=1}^H (x_i - \mu)^2$ (variance across features)
- $\epsilon > 0$ (small constant, e.g., 10^{-5} , for numerical stability)
- $\gamma, \beta \in \mathbb{R}^H$ (learnable scale and shift parameters per feature)

In matrix form for sequence embeddings $X \in \mathbb{R}^{n \times d}$, LayerNorm applies element-wise after per-token statistics across d -dimensional features.

- **Residual connections.** Introduced in ResNet and adopted in transformer models, one adds the input x directly to a layer's output $F(x)$, yielding $H(x) = F(x) + x$. This proves to be very helpful in training NNs in general:
 - Enable direct backpropagation of gradients through skip paths, mitigating vanishing gradients in deep networks (e.g., 100+ layers).

- Allow layers to learn identity mappings ($F(x) \approx \mathbf{0}$) when extra depth adds no value. This avoids performance drops as networks deepen—plain deep nets degrade.

- **Feed-Forward Network (FFN):** Position-wise application per token:

$$\text{FFN}(Z_1) = \sigma(Z_1 W_1 + b_1) W_2 + b_2, \quad (5.1.13)$$

where σ is ReLU or GELU activation.

- **Add & Norm (Post-FFN):** Residual connection and normalization (again):

$$Z_2 = \text{LayerNorm}(Z_1 + \text{FFN}(Z_1)). \quad (5.1.14)$$

Add & Norm (i.e., residual connections and LayerNorm) are used together again. This prevents exploding/vanishing signals through deep stacks, providing numerical stability.

- **Output to Next Block:** Processed Z_2 is passed as input to the next transformer block.

3. Tips for running/deploying DL models/NNs at scale.

- Emphasize Robust Infrastructure and Practical Simplicity

Before scaling complex architectures, invest deeply in a robust, elastic ML infrastructure capable of handling distributed training, large datasets, and serving at scale. Optimize the end-to-end data and compute pipelines for reproducibility, efficiency, and fault tolerance. Even with cutting-edge architectures (e.g., Transformers, diffusion models), prefer simplicity when it meets product needs. In production, a smaller, well-regularized model with stable latency and controllable behavior often outperforms a massive but fragile alternative.

- Domain Knowledge and Informed Model Design

Integrate deep domain expertise throughout model development to guide architecture choices, input representations, and loss design. Feature engineering remains relevant—even for DL—through feature normalization, embedding design, and signal enrichment rooted in domain intuition. Mitigate training-serving skew by aligning preprocessing code, ensuring deterministic pipelines, and validating model behavior across real-time inputs.

- Evaluation and Objective Alignment

Connect every modeling objective to a tangible business or user outcome. Use composite evaluation that reflects long-term user satisfaction, not just accuracy, AUC, or perplexity. Where applicable, evaluate model calibration, robustness under distribution shift, and fairness metrics to understand deployment implications. Optimize for downstream task utility, not benchmark performance.

- Data, Metadata, and Feature Management

Adopt managed data pipelines and feature stores (e.g., Google services like Vertex AI Feature Store, etc.) for consistent feature generation and lineage tracking. Continuously compute and refresh key features using scheduled batch or streaming jobs to maintain temporal relevance. Maintain strong data versioning, schema evolution tracking, and metadata logging to support reproducibility and quick rollback during anomalies.

- Automated Experimentation and Optimization Use distributed hyperparameter optimization frameworks (like Google's Vizier) for efficient exploration of the architecture-learning rate-regularization space. Centralize experiment logging, visualization, and artifact tracking via professional-grade tools. Adopt early stopping, checkpoint management, and resource preemption policies to maximize research throughput without compute waste.

- Model Serving, Monitoring, and Reliability Deploy models through containerized, autoscaling services with high availability and low prediction latency. Establish continuous monitoring for data drift, concept drift, and resource health. Integrate feature attribution tools (e.g., Integrated Gradients, SHAP) to interpret changes in model reasoning and detect anomalous inputs early. Set up shadow evaluation and canary releases to measure real-world performance before full rollout.

- MLOps and Workflow Orchestration

Build modular, pipeline-based workflows (e.g., TFX, Kubeflow, Dagster) for reproducible model training, evaluation, and deployment. Version control everything—data, model weights, architecture definitions, and configurations—to enable full lineage tracking and regulatory compliance. Automate retraining schedules, validation gates, and artifact promotion to streamline continuous delivery and rollback safety.

- Collaboration and Cross-Disciplinary Transparency

Foster a culture of openness and peer review where model designs, training anomalies, and empirical findings are shared across teams. Encourage visibility between modeling, data engineering. Coordinate teams for scalability and maintainability. Balance model sophistication, interpretability, and cost to serve, focusing on product-level value rather than academic novelty.

For more info, see Google's Deep Learning Tuning Playbook.

-
1. The names are motivated by classic information retrieval systems: we use a *query* to search for a relevant *key* within a database, and upon finding a match, the system returns the associated *value* (content).
 2. How do researchers train the W matrices in multi-head attention so that different heads focus on specific aspects of data (e.g., verbs vs. nouns in text)?
 - Randomize the starting points: make sure the parameters in each head start at different initial values to create diversity.
 - Ensure different sets of matrices receive (slightly) different slices of the input feature space and a different portion of the backpropagated gradient. Their updates during optimization would move them in different directions (creating possibly different focuses).
 - In a recent paper (MoH: Multi-Head Attention as Mixture-of-Head Attention, by Jin et al.), authors reimagine multi-head attention as a Mixture-of-Experts (MoE) system. Attention heads act as “experts” and a learnable router (a NN) dynamically selects the most relevant heads per token.
 - The dropout trick. Randomly zeros out a fraction of the weights or activations during training. This forces the model to learn more robust, non-redundant representations and avoid co-adaption.
 - Head Pruning: Post-training removal of low-contribution heads (e.g., via magnitude or attention patterns).



5.2: Universal Approximation Theorem

We outline the core ideas underlying the Universal Approximation Theorem.

The Universal Approximation Theorem (Cybenko, 1989)

Statement

Let σ be a continuous sigmoidal function, meaning $\sigma(t) \rightarrow 1$ as $t \rightarrow \infty$ and $\sigma(t) \rightarrow 0$ as $t \rightarrow -\infty$. Let K be a compact subset of $\mathbb{R}^n$. Then, for any continuous function $f : K \rightarrow \mathbb{R}$ and any $\epsilon > 0$, there exists a function $G(x)$ of the form:

$$G(x) = \sum_{i=1}^N \alpha_i \sigma(w_i \cdot x + b_i) \quad (5.2.1)$$

where $\alpha_i, b_i \in \mathbb{R}$ and $w_i \in \mathbb{R}^n$, such that $|G(x) - f(x)| < \epsilon$ for all $x \in K$.

In other words, the set of finite linear combinations of sigmoidal functions is dense in the space of continuous functions $C(K)$ with respect to the uniform norm.

Outline of the Proof

The proof proceeds by contradiction. We assume the set of neural network functions is *not* dense and show this leads to a logical impossibility using powerful results from functional analysis.

Setting up the Proof by Contradiction

- Let $\diamond$ be the set of all functions of the form $G(x) = \sum_{i=1}^N \alpha_i \sigma(w_i \cdot x + b_i)$. This is the set of functions our neural networks can compute.
- Let $C(K)$ be the Banach space of all continuous real-valued functions on the compact set K , equipped with the uniform norm ($\|f\|_\infty = \sup_{x \in K} |f(x)|$).
- The theorem claims that the closure of $\diamond$, denoted $\overline{\diamond}$, is equal to $C(K)$.

Assumption for Contradiction: Assume $\overline{\diamond} \neq C(K)$. This means $\diamond$ is *not* dense in $C(K)$. The closure $\overline{\diamond}$ is a proper, closed subspace of $C(K)$.

Application of the Hahn-Banach Theorem

- Since $\overline{\diamond}$ is a proper, closed subspace of the normed vector space $C(K)$, we can apply a corollary of the Hahn-Banach Theorem.
- Hahn-Banach Separation Theorem (Corollary):** If Y is a proper closed subspace of a normed space X , there exists a non-zero continuous linear functional L on X that annihilates Y (i.e., $L(y) = 0$ for all $y \in Y$).
- Applying this to our case, there exists a non-zero continuous linear functional $L \in C(K)^*$ such that:

$$L(g) = 0 \quad \text{for all } g \in \overline{\diamond}. \quad (5.2.2)$$

- Since $\diamond \subset \overline{\diamond}$, this implies $L(g) = 0$ for all $g \in \diamond$.
- Because L is non-zero, there must exist some function $f \in C(K)$ for which $L(f) \neq 0$.

Application of the Riesz Representation Theorem

- The Riesz Representation Theorem for $C(K)$ characterizes all continuous linear functionals on $C(K)$.
- Riesz Representation Theorem:** For any continuous linear functional L on $C(K)$, there exists a unique finite signed regular Borel measure μ on K such that for any function $h \in C(K)$, the functional can be written as an integral:

$$L(h) = \int_K h(x) d\mu(x). \quad (5.2.3)$$

- Applying this to our functional L , we have a unique measure μ such that $L(h) = \int_K h d\mu$.
- Since L is a non-zero functional, the corresponding measure μ **cannot be the zero measure**.
- Our condition from the Hahn-Banach step, $L(g) = 0$ for all $g \in \mathcal{D}$, now translates into a condition on the measure μ :

$$\int_K g(x) d\mu(x) = 0 \quad \text{for all } g \in \mathcal{S}. \quad (5.2.4)$$

Analyzing the Annihilating Measure μ

- The condition $\int_K g(x) d\mu(x) = 0$ must hold for any function $g \in \mathcal{D}$. By the linearity of the integral, this is equivalent to the condition holding for a single term of the sum:

$$\int_K \sigma(w \cdot x + b) d\mu(x) = 0 \quad \text{for all } w \in \mathbb{R}^n \text{ and all } b \in \mathbb{R}. \quad (5.2.5)$$

- We have now reduced the problem to the following: We have a non-zero measure μ that satisfies the integral equation above. We will show this is impossible.

The Core of the Proof: Showing μ must be the Zero Measure

- This is the most technical part of the proof. We show that the only measure μ that satisfies the condition in step 4 is the zero measure.
- Discriminatory Function:** A function σ is called *discriminatory* if the only finite signed regular Borel measure μ on K that satisfies $\int_K \sigma(w \cdot x + b) d\mu(x) = 0$ for all w, b is the zero measure.
- The proof now hinges on showing that a sigmoidal function σ is discriminatory. Cybenko's proof does this by using Fourier analysis.
 - Fourier Transform of the Measure.** Let $\mu(\xi)$ be the Fourier transform of the measure μ , defined as $\mu(\xi) = \int_K e^{-ix \cdot \xi} d\mu(x)$ for $\xi \in \mathbb{R}^n$.
 - Integral Representation of the Sigmoid.** A key property of the sigmoid function σ is that it can be expressed as an integral of exponential functions. Specifically, there exists a non-zero finite measure ν on $\mathbb{R}$ such that:

$$\sigma(t) = \int_{\mathbb{R}} e^{ity} d\nu(y). \quad (5.2.6)$$

(This is possible because σ is bounded and non-constant. The measure ν is related to the derivative of σ).

- Substitution and Fubini's Theorem.** Substitute the integral representation of σ into our condition on μ :

$$\int_K \left(\int_{\mathbb{R}} e^{iy(w \cdot x + b)} d\nu(y) \right) d\mu(x) = 0. \quad (5.2.7)$$

Assuming we can apply Fubini's Theorem (which is justified by the properties of μ and ν), we can swap the order of integration:

$$\int_{\mathbb{R}} \left(\int_K e^{iyw \cdot x} d\mu(x) \right) e^{iyb} d\nu(y) = 0. \quad (5.2.8)$$

4. Isolating the Fourier Transform. The inner integral is the complex conjugate of the Fourier transform of μ evaluated at $-yw$. So we have:

$$\int_{\mathbb{R}} \overline{\hat{\mu}(-yw)} e^{iyb} d\nu(y) = 0 \quad \text{for all } w \in \mathbb{R}^n \text{ and } b \in \mathbb{R}. \quad (5.2.9)$$

5. The Final Contradiction. For a fixed w , the expression

$$\int_{\mathbb{R}} \overline{\hat{\mu}(-yw)} e^{iyb} d\nu(y) \quad (5.2.10)$$

is the Fourier transform (with respect to the variable b) of the function $y \mapsto \overline{\hat{\mu}(-yw)}$ with respect to the measure ν . The fact that this Fourier transform is zero for all b implies that the function $\overline{\hat{\mu}(-yw)}$ must be zero for ν -almost all y . The crucial property of the sigmoid's corresponding measure ν is that its support is not contained in any proper subspace of $\mathbb{R}$. This means the set of y for which the condition holds is not trivial. Since the condition $\mu(-yw) = 0$ holds for a set of y values with non-zero measure, and it must hold for all $w \in \mathbb{R}^n$, it follows that $\mu(\xi) = 0$ for all $\xi \in \mathbb{R}^n$. If the Fourier transform of a measure is identically zero, the measure itself must be the zero measure. Therefore, **μ must be the zero measure.**

Conclusion of the Proof

- In step 5, we showed that the condition $\int_K \sigma(w \cdot x + b) d\mu(x) = 0$ implies that μ must be the zero measure.
- This directly contradicts our finding in step 3, where we concluded that μ must be a non-zero measure because the functional L was non-zero.
- This contradiction means our initial assumption in step 1—that $\overline{\mathcal{S}}$ is not equal to $C(K)$ —must be false.
- Therefore, $\overline{\mathcal{S}} = C(K)$. The set of finite sums of sigmoidal functions is indeed dense in the space of continuous functions on a compact set. This completes the proof of the Universal Approximation Theorem.

5.3: Attention Mechanism/Transformers

This is a brief introduction to the mathematical model for transformers, based on Vaswani et al. (2017), “Attention Is All You Need”; Devlin et al. (2018), “BERT”; and Radford et al. (2018), “GPT.”

Introduction to Attention

- Core Idea: Dynamically focus on relevant parts of input sequences when generating output
- Motivation: Overcome limitations of fixed-length context vectors in seq2seq models (e.g., information bottleneck)
- Key Components:
 - Query (Q): Represents the current decoding step (e.g., target word)
 - Key (K): Represents input words to be matched against the query
 - Value (V): Represents the actual content of input words

- Mathematical Formulation:

- Attention Score: Computed as compatibility between query and keys

$$\text{score}(Q, K_i) = \frac{Q \cdot K_i^T}{\sqrt{d_k}} \quad (5.3.1)$$

where d_k = dimension of keys (scaling factor for stable gradients)

- Attention Weights: Softmax over scores

$$\alpha_i = \frac{\exp(\text{score}(Q, K_i))}{\sum_j \exp(\text{score}(Q, K_j))} \quad (5.3.2)$$

- Context Vector: Weighted sum of values

$$\text{context} = \sum_i \alpha_i V_i \quad (5.3.3)$$

Self-Attention

- Definition: Attention mechanism where queries, keys, and values come from the same sequence
- Purpose: Model intra-sequence dependencies (e.g., word relationships within a sentence)
- Mathematical Steps:

- Input: Sequence $X = [x_1, x_2, \dots, x_n]$ (each $x_i \in \mathbb{R}^{d_{\text{model}}}$)

- Linear Projections:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V \quad (5.3.4)$$

where $W^Q, W^K, W^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$ (learnable weights)

- Scaled Dot-Product Attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V \quad (5.3.5)$$

- Output: Sequence of context vectors $[z_1, z_2, \dots, z_n]$, where each $z_i \in \mathbb{R}^{d_v}$

Multi-Head Attention

- Concept: Run multiple self-attention mechanisms in parallel and concatenate outputs
- Advantage: Capture diverse relationships (e.g., syntactic, semantic) from different representation subspaces
- Mathematical Formulation:
 - Projections for h Heads:

$$Q_i = XW_i^Q, \quad K_i = XW_i^K, \quad V_i = XW_i^V \quad (i = 1, \dots, h) \quad (5.3.6)$$

where $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$ (and $d_k = d_{\text{model}}/h$)

- Head Output:

$$\text{head}_i = \text{Attention}(Q_i, K_i, V_i) \quad (5.3.7)$$

- Concatenation & Final Projection:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O \quad (5.3.8)$$

where $W^O \in \mathbb{R}^{h \cdot d_v \times d_{\text{model}}}$ (output projection matrix)

Positional Encoding

- Problem: Self-attention is permutation-invariant (lacks sequence order information)
- Solution: Add positional embeddings to input embeddings
- Formulation:
 - Sinusoidal Encoding (fixed):

$$PE_{(pos, 2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right) \quad (5.3.9)$$

$$PE_{(pos, 2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right) \quad (5.3.10)$$

where pos = position index, i = dimension index

- Learned Embeddings: Alternative trainable position vectors
- Input to Transformer:

$$X_{\text{input}} = \text{Embedding}(X) + \text{PositionalEncoding} \quad (5.3.11)$$

Transformer Architecture

- Overall Structure: Encoder-decoder framework with stacked self-attention and feed-forward layers

- Encoder:

- Layers: N identical blocks (e.g., $N = 6$)
- Sublayers:

1. Multi-Head Self-Attention:

$$Z = \text{MultiHead}(X, X, X) \quad (5.3.12)$$

2. Add & Norm: Residual connection + layer normalization:

$$X_{\text{sub}} = \text{LayerNorm}(X + Z) \quad (5.3.13)$$

3. Position-wise Feed-Forward Network (FFN):

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \quad (5.3.14)$$

(applied independently to each position)

4. Add & Norm:

$$X_{\text{out}} = \text{LayerNorm}(X_{\text{sub}} + \text{FFN}(X_{\text{sub}})) \quad (5.3.15)$$

- Decoder:

- Layers: N identical blocks with two self-attention sublayers and one cross-attention
- Sublayers:

1. Masked Multi-Head Self-Attention: Prevents attending to future tokens (autoregressive decoding)

- Masking: Set scores of future positions to $-\infty$ before softmax

2. Add & Norm

3. Encoder-Decoder Attention (Cross-Attention):

- Query: From decoder output
- Key/Value: From encoder output

$$Z = \text{MultiHead}(Q_{\text{dec}}, K_{\text{enc}}, V_{\text{enc}}) \quad (5.3.16)$$

4. Add & Norm

5. FFN + Add & Norm

Layer Normalization

- Purpose: Stabilize training by normalizing activations
- Formula:

$$\text{LayerNorm}(x) = \gamma \left(\frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \right) + \beta \quad (5.3.17)$$

where μ = mean, σ^2 = variance over features, γ, β = learnable parameters

Training & Optimization

- Loss Function: Cross-entropy loss for token prediction
- Key Techniques:
 - Label Smoothing: Prevents overconfidence (e.g., replace 1-hot targets with $1 - \epsilon$ for correct token, $\epsilon/(V - 1)$ for others)
 - Learning Rate Schedule: Warmup + decay, e.g.,
$$\text{lr} = d_{\text{model}}^{-0.5} \cdot \min(\text{step}^{-0.5}, \text{step} \cdot \text{warmup}^{-1.5}) \quad (5.3.18)$$
 - Regularization: Dropout applied to embeddings, attention weights, and FFN outputs

Key Innovations & Advantages

- Parallelization: Self-attention enables $O(1)$ sequential dependency (vs. $O(n)$ in RNNs)
- Long-Range Dependencies: Direct path between any two positions (vs. vanishing gradients in RNNs)
- Interpretability: Attention weights visualize alignment (e.g., word-to-word relationships)

Variants & Extensions

- BERT: Bidirectional encoder (masked language modeling)
- GPT: Autoregressive decoder-only (next-token prediction)
- T5/BART: Encoder-decoder fusion (sequence-to-sequence transformation for translation/summarization)
- Mixture of Experts (MoE): Sparse routing decoder (conditional activation of specialist sub-networks per token)
- Hybrid Mamba-Transformer: State-space linear decoder (interleaved attention and recurrent blocks for efficient long contexts)
- Large Reasoning Models (LRM): Test-time compute decoder (reinforcement learning-tuned generation of internal chain-of-thought traces)
- Efficient Transformers:
 - Sparse Attention (e.g., Longformer, BigBird)
 - Linformer: Low-rank projection of keys/values
 - Performer: Kernelized attention

Mathematical Summary

- Self-Attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V \quad (5.3.19)$$

- Multi-Head Attention:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h) W^O \quad (5.3.20)$$

- FFN:

$$\text{FFN}(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2 \quad (5.3.21)$$

- Positional Encoding:

$$PE_{(pos,i)} = \begin{cases} \sin(pos/10000^{2i/d_{\text{model}}}) & \text{if } i \text{ even} \\ \cos(pos/10000^{2i/d_{\text{model}}}) & \text{if } i \text{ odd} \end{cases} \quad (5.3.22)$$

6: DISTANCE-BASED CLASSIFIERS

6.1: Chapter 6 Summary and a Discussion on the Neural Gas Model

This page titled [6: Distance-based Classifiers](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by Hei-Chi Chan.

6.1: Chapter 6 Summary and a Discussion on the Neural Gas Model

This is a recap of SD chapter 6, on Distance-based classifiers.

- Sect. 6.1. Prototype-based classifiers
 1. Prototype-based algorithms use representative data points (prototypes) to model data. New data is classified by comparing it to these learned examples, similar to how K-means uses cluster centers (centroids).
 2. Learning Vector Quantization (LVQ) is one such supervised learning (classification) algorithm introduced by Teuvo Kohonen.
 3. How LVQ works:
 1. It represents classes with “prototype vectors” that are iteratively adjusted in the feature space during training. When a training sample is presented, the nearest prototype is moved closer to the sample if they share the same class label and pushed further away if their labels differ.
 2. To classify new data, it calculates the distance between the new input sample and all existing prototype vectors in the trained model. The algorithm then assigns the class label of the single closest prototype to the new data point.
 3. Note: LVQ is similar to K-means, but fundamentally they are different. LVQ is a supervised learning model, while K-means is unsupervised.
 4. Some variants incorporate neighborhood cooperation inspired by other prototype schemes like Self-Organizing Maps (SOM) or Neural Gas.
- Sect. 6.2. Distance measures and relevance learning
 1. Beyond Euclidean Distance:
 1. Classic LVQ typically uses Euclidean distance as the similarity measure for determining the nearest prototype. This implicitly assumes isotropic distributions in data space.
 2. Modern approaches optimize or adapt the distance measure (metric learning) to better fit specific applications. E.g., use Mahalanobis distance when data is anisotropic (elongated or correlated).
 2. Adaptive Distance Measures:
 1. Adaptive or parameterized distance metrics are tuned during training to enhance discriminative power. E.g., Localized Adaptive Metrics, which assign a specific distance parameter matrix Λ_j to each individual prototype w_j to adapt to local data distributions:
$$d_{\Lambda_j}(w_j, \xi) = (\xi - w_j)^T \Lambda_j (\xi - w_j). \quad (6.1.1)$$

Λ_j is to learn from the training data. One can set $\Lambda_j = \Omega_j^T \Omega_j$ to ensure Λ_j remains positive semi-definite.
 2. These approaches can find directions in feature space with stronger class separability.
 3. Metric adaptation is an active research area, extending LVQ’s applicability and performance in real-world tasks.
- History of LVQ
 1. LVQ was invented by Teuvo Kohonen in the late 1980s as a supervised counterpart to vector quantization. It is related to and inspired by biological neural systems and Hebbian learning principles.

2. LVQ served as a precursor to methods like Self-Organizing Maps (SOM) and Neural Gas. Over time, many variants and metric learning extensions have been developed, enhancing its robustness.
 3. LVQ is known for its simplicity, interpretability, and effectiveness in prototype-based classification.
- Neural Gas (NG): The Martinetz-Schulten algorithm
 1. NG is an unsupervised NN model introduced in 1991 by Thomas Martinetz and Klaus Schulten to find optimal data representations through vector quantization. Indeed, it is designed to learn an optimal representation of a data manifold by minimizing the reconstruction error (distortion) while preserving the data's topological structure.
 2. While it was explicitly inspired by Kohonen's Self-Organizing Map (SOM), NG differs by using a “gas-like” dynamic where neurons distribute themselves based on distance-ranking rather than being constrained to a fixed, predefined grid like the SOM. Both algorithms are descendants of Kohonen's LVQ, which serves as their supervised counterpart. While SOM and NG explore unlabeled data to learn underlying structures, LVQ uses labeled data to fine-tune prototype positions for specific classification tasks.
 3. In a nutshell: NG starts with a set of reference vectors (a codebook). These are the neural gas. Through a dynamical learning process (resembling a self-organizing gas), they will migrate toward data-dense regions. Concurrently, it learns the connections (edges) between close neurons, effectively learning the intrinsic topology of the data structure.

Once the learning process is completed, the codebook vectors represent a discrete skeleton of the data, providing a foundation for data compression, cluster analysis, and accelerated vector search.

4. The NG algorithm in details:

Variables and Initial Values

- $W = \{w_1, w_2, \dots, w_N\}$ (Codebook vectors or neurons): A set of N reference vectors initialized to random positions in the input space $\mathbb{R}^d$.
- C (Adjacency Matrix): An $N \times N$ binary matrix initialized to $\mathbf{0}$ ($C_{ij} \in \{0, 1\}$). It tracks topological connections between neurons: if $C_{ij} = 1$, this means the i -th and the j -th neurons are topological neighbors.
- T (Age Matrix): An $N \times N$ integer matrix initialized to $\mathbf{0}$, where t_{ij} stores the current age of the edge between neuron i and j .

Note: Think of it as a dynamical score for the edges/connections (between neurons) during the learning process. As we will see, lower score is preferred: they indicate the associated edges/connections are active (while obsolete paths, with higher age scores, are no longer representing the data, and they may be deleted).

- Parameters:
 - $\epsilon(t)$ (Learning Rate): Initialized at ϵ_{start} (e.g., 0.5), decaying exponentially over time to ϵ_{end} (e.g., 0.01).
 - $\lambda(t)$ (Neighborhood Radius): Initialized at λ_{start} (e.g., $N/2$), decaying exponentially over time to λ_{end} (e.g., 0.01).
 - T_{max} (Age Threshold): The maximum age an edge can reach before removal.

The Training Loop (Iterative Learning Process)

Select randomly an input vector v (until all data points are visited):

- Step 1: Distance Ranking (Find the top two nearest neurons to v)

Calculate Euclidean distances $\|v - w_i\|$ for all $i \in \{1, \dots, N\}$. Sort the neurons and assign a rank $k_i \in \{0, 1, \dots, N - 1\}$ to each neuron i , where $k_i = 0$ is the closest (winner) and $k_i = N - 1$ is the most distant. We will need the runner-up later.

- Step 2: Update Reference Vectors (Shift the neural gas)

Adapt all vectors w_i using this update rule:

$$\Delta w_i = \epsilon(t) \cdot e^{-k_i/\lambda(t)} \cdot (v - w_i). \quad (6.1.2)$$

- Step 3: Update Topology (Matrix C and Age T) (Learning the topology dynamically)

Identify the winner (i_0) and the runner-up (i_1).

- Connection Logic:

If $C_{i_0 i_1} = 0$, set $C_{i_0 i_1} = 1$ to create a new edge. If $C_{i_0 i_1} = 1$, maintain the connection.

- The Refresh (“keeping it fresh”)

Always reset the age of this specific edge to zero ($t_{i_0 i_1} = 0$).

- Step 4: Age the Winner’s Neighbors

For all other neurons j currently connected to the winner i_0 ($C_{i_0 j} = 1$): increase their ages by 1 unit:

$$t_{i_0 j} = t_{i_0 j} + 1. \quad (6.1.3)$$

(the values of the C elements remain 1).

Note: The runner-up i_1 is protected from this aging because its age was reset to 0 in the previous step.

- Step 5: Prune Obsolete Edges (removing aging connections)

For all neighbors j of the winner i_0 , check the age condition:

If

$$\text{If } t_{i_0 j} > T_{max}, \text{ set } C_{i_0 j} = 0 \text{ and } C_{j i_0} = 0. \quad (6.1.4)$$

Note: In the original NG, isolated neurons (not connected to other neurons) are not deleted; they remain in the gas to be potentially re-integrated later.

- Step 6: Update Parameters

Update the time-dependent parameters $\epsilon(t)$ and $\lambda(t)$ using exponential decay.

The Final Label/Quantization Assignment (Post-Training) Once training iterations are complete, the codebook W and adjacency matrix C are frozen for the final data representation: For every vector v in the original dataset:

- Compute distances to all final reference vectors w_i .
- Find the index i of the absolute nearest neighbor.
- Update Label: Set the definitive quantization label for v to i . (this is the quantization map: $Q(v) = i$).

5. Applications

Once the final codebook is fixed and every data vector has its quantization label (the index of its nearest prototype), the system is ready for high-performance downstream tasks (using these discrete representations to solve problems that were too computationally expensive with the original, high-dimensional raw data).

1. Massive Data Compression: Replace the entire dataset of large vectors with a list of small integers (the indices).
2. Ultrafast Vector Search and Databases: Modern vector databases use these quantized labels to perform

Approximate Nearest Neighbor (ANN) searches. Instead of comparing a query vector to every single original vector, the system only compares it to the smaller set of codebook prototypes.

3. Topological and Cluster Analysis: By examining the adjacency matrix C , one can identify sub-structures of the data (e.g., disjoint clusters).
4. Anomaly Detection: Once the codebook is fixed, any new data vector that is “too far” from its nearest prototype (high quantization error) can be flagged as an anomaly or noise.
5. Signal and Image Reconstruction: In multimedia applications, the final indices are transmitted over networks (not the actual data vectors). The receiver decodes it use the same codebook to reconstruct (approximately) the original signal.

Further Exploration

For details, readers may refer to:

- Kohonen, T. (2001). *Self-organizing maps* (3rd ed.). Springer-Verlag.
- Martinetz, T. M., & Schulten, K. J. (1991). A “neural-gas” network learns topologies. In T. Kohonen, K. Mäkisara, O. Simula, & J. Kangas (Eds.), *Artificial neural networks* (pp. 397–402).

This page titled [6.1: Chapter 6 Summary and a Discussion on the Neural Gas Model](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by Hei-Chi Chan.

7: MODEL EVALUATION AND REGULARIZATION

7.1: Chapter 7 Summary

7.2: The Bias-Variance Trade Off

7.3: Explainable AI (XAI)

This page titled [7: Model Evaluation and Regularization](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by Hei-Chi Chan.

7.1: Chapter 7 Summary

This is a recap of SD chapter 7, Model evaluation and regularization.

- Sect. 7.1. Bias and variance.
- Sect. 7.2. Taming the complexity.

In NN/DL, complexity control is essential for preventing overfitting, reducing computational demands, and enabling scalable deployment in production environments.

1. Early Stopping

Halt training when validation performance degrades. Use a patience parameter to monitor metrics like loss or accuracy over epochs (e.g., stop training if the validation loss does not improve for 10 consecutive epochs). Implement with callbacks, saving the best checkpoint to avoid overfitting on large datasets (ensuring the most generalized weights are retained/saved). At scale, combine with learning rate schedulers for robust convergence in distributed training (balancing stability, speed, and final model precision).

2. Weight Decay and Regularization

Add an L_2 penalty to the loss function (which shrinks weights during optimization). Use L_1 regularization (LASSO) for sparsity; or elastic net combining L_1 and L_2 , critical for high-dimensional embeddings in large-scale models.

3. Pruning

Pruning removes low-importance weights post-training or during training. Use criteria like magnitude, gradient norms, or Taylor expansion for importance scores. Structured pruning targets filters/channels for hardware acceleration (unstructured pruning zeroes out random individual weights and creates sparse matrices that standard hardware still processes as dense, structured pruning physically resizes the matrices). Behind this is the lottery ticket hypothesis: a small, well-initialized subnetworks can match full-model performance. Industry trends include train-time pruning with sparsity regularization and tools like Torch-Prune (a specialized PyTorch library designed for structural network pruning) for billion-parameter models.

4. Weight Sharing

This constrains multiple connections to identical values, slashing parameters (core to CNNs where kernels slide across inputs for translation invariance). Advanced forms include hash-based sharing (e.g., HashNet), which cluster similar weights and hierarchical sharing across transformer layers or ResNet blocks. Could reduce memory by significantly without retraining.

5. Dropout

This randomly deactivates neurons during training, forcing ensemble-like robustness and preventing co-adaptation. Variants: DropConnect (setting a randomly selected subset of individual weights (edges) to zero rather than entire hidden unit activations, allowing neurons to remain partially active), Gaussian Dropout (replacing the random dropout with multiplicative Gaussian noise (typically centered at 1), making the network more robust and less sensitive to small changes in input, helping it to learn more generalizable features), Zoneout (designed for RNNs, it stochastically forces hidden units to maintain their previous values (identity mask) instead of dropping them to zero, better preserving information flow over time).

6. Other Methods

1. Low-Rank Adaptation (LoRA): Factorizes weight updates into low-rank matrices for fine-tuning LLMs, could significantly reduce the number of parameters.
2. Knowledge Distillation: Trains compact “student” models mimicking “teacher” (full-model) logits, with temperature-

scaled softmax; trending for on-device AI with progressive distillation.

3. Batch Normalization & Variants: Normalizes activations to stabilize training; alternatives are GroupNorm or Evolving Batch Normalization (EVBN).
4. Movement Pruning.

Identifying and removing unimportant weights that do not change much during fine-tuning, unlike older methods that just zero small weights (magnitude pruning). It focuses on weights that are moving away from zero during training, making it better for transfer learning, leading to smaller, faster models with minimal accuracy loss, even at high sparsity levels.

- Sect. 7.3. Cross-validation and related methods.

K-fold cross-validation (*k*-CV) is a resampling technique to estimate a model's ability to generalize to unseen data. It divides a dataset into *k* equal-sized subsets, or "folds." The process is repeated *k* times; in each iteration, one unique fold is held out as the validation set for testing, while the remaining *k* – 1 folds are combined to train the model. After all *k* iterations are complete, the performance scores from each test are averaged to provide a robust, approximately unbiased estimate of the model's predictive skill. This method is particularly valued because it ensures every data point is used for both training and validation, effectively reducing the risk of overfitting and minimizing the bias associated with a single train-test split. Common values for *k* are 5 or 10, balancing computational efficiency with the stability of the performance.

Choosing hyperparameters (model selection) and learning weights (model fitting):

1. Initial Split: Divide your data into a training set and a test set. The test set to be "locked away" until the final evaluation.
2. Grid/Random Search: Define a range of hyperparameter combinations to be tested. (there are the parameters that control the learning process)
3. The *k*-CV Loop: For each hyperparameter combination (in the chosen range):
 1. Initialize a brand new model from scratch (no previous weights are kept).
 2. Perform *k*-fold cross-validation by splitting the training set into *k* folds.
 3. Train *k* separate models, each on *k* – 1 folds, and record their average validation performance.
4. Hyperparameter Selection: Choose the hyperparameter set that achieved the best average score across its *k* validation folds.
5. Final Retraining: Use those "best" hyperparameters to train one final model on the entire original training set (all *k* folds combined). This allows the model to learn its final internal weights from the maximum amount of available data.
6. External Validation: Finally, evaluate this single "refitted" model on the test set (the dataset aside in the first step) to determine how it will perform.

- Sect. 7.4. Performance Measures

1. Regression Metrics

$$\begin{aligned}
 \text{MAE} &= \frac{1}{N} \sum_{i=1}^N |y_i - \hat{y}_i|, \\
 \text{MSE} &= \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2, \\
 \text{RMSE} &= \sqrt{\text{MSE}}, \\
 R^2 &= \frac{\sum_i (\hat{y}_i - \bar{y})^2}{\sum_i (y_i - \bar{y})^2}.
 \end{aligned}
 \tag{7.1.1}$$

2. Classification Metrics Binary confusion counts: *TP* (true positive), *FP* (false positive), *TN* (true negative), *FN* (false

negative).

$$\begin{aligned}
 \text{Accuracy} &= \frac{TP + TN}{TP + FP + TN + FN}, \\
 \text{Precision} &= \frac{TP}{TP + FP}, \quad (\text{When the model predicts a positive, how often is it correct?}) \\
 \text{Recall} &= \frac{TP}{TP + FN}, \quad (\text{Of all the actual positives, how many does it identify correctly?}) \\
 F1\text{-score} &= \left(\frac{\text{Precision}^{-1} + \text{Recall}^{-1}}{2} \right)^{-1} = 2 \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}, \\
 F_\beta &= \left(\frac{\text{Precision}^{-1} + \beta^2 \cdot \text{Recall}^{-1}}{1 + \beta^2} \right)^{-1} = (1 + \beta^2) \frac{\text{Precision} \cdot \text{Recall}}{\beta^2 \cdot \text{Precision} + \text{Recall}}.
 \end{aligned} \tag{7.1.2}$$

ROC and AUC:

1. The Receiver Operating Characteristic (ROC) curve is a parametric plot of the performance of a binary classifier as the discrimination threshold t varies. It is defined by the set of points $(FPR(t), TPR(t))$:

- **True Positive Rate (Sensitivity) = Recall:**

$$TPR(t) = \frac{TP(t)}{TP(t) + FN(t)} \tag{7.1.3}$$

- **False Positive Rate (1 - Specificity):**

$$FPR(t) = \frac{FP(t)}{FP(t) + TN(t)} \tag{7.1.4}$$

Geometric properties:

- Random Baseline: A non-informative classifier follows the diagonal: $TPR(FPR) = FPR$.
 - Ideal Classifier: A perfect model passes through the point $(0, 1)$, indicating $TPR = 1$ and $FPR = 0$.
2. Area Under the Curve (AUC): it summarizes the ROC curve into a single scalar value by integrating the True Positive Rate over the range of False Positive Rates:

$$AUC = \int_0^1 TPR(FPR) d(FPR). \tag{7.1.5}$$

The AUC is equivalent to the probability that the classifier $f(x)$ will rank a randomly chosen positive instance (x_{pos}) higher than a randomly chosen negative instance (x_{neg}):

$$AUC = \text{Prob}(f(x_{pos}) > f(x_{neg})). \tag{7.1.6}$$

Note: For a set of n discrete observations, the AUC is given by

$$AUC = \frac{\sum_{i=1}^{n_{pos}} \sum_{j=1}^{n_{neg}} I(f(x_i) > f(x_j))}{n_{pos} \cdot n_{neg}}. \tag{7.1.7}$$

where $I(\cdot)$ is the indicator function. It calculates the proportion of correct rankings across all possible pairs.

3. Sect. 7.5. On Explainable Artificial Intelligence (XAI): a suite of ML processes, tools, and frameworks designed to make the internal logic and outputs of complex AI systems understandable to human experts and everyday users.

7.2: The Bias-Variance Trade Off

This is the notes for SD Sect. 7.1, on bias, variance, and double descent.

Bias and variance are key concepts in ML. They deal with the decomposition of the expected prediction error of a model:

$$\mathbb{E}[(y - \hat{f}(x))^2] = \text{Bias}^2[\hat{f}(x)] + \text{Var}[\hat{f}(x)] + \sigma^2, \quad (7.2.1)$$

where bias measures systematic deviation from the true function $f(x)$, variance captures sensitivity to training data fluctuations, and σ^2 is irreducible noise. See the next section for a detailed derivation.

High bias leads to underfitting (simple models (e.g., linear regression on nonlinear data) fail to capture patterns, yielding poor training and test performance).

High variance causes overfitting (complex models memorize noise, excelling on training data but performing poorly on unseen test sets).

Classical tradeoff: Optimal complexity minimizes total error at the “U-shaped” curve.

Double Descent Phenomenon

Double descent describes the non-monotonic relationship between model complexity and test error:

- **First Descent (Classical Regime):** As model complexity increases, the test error initially decreases as the model moves from high-bias (underfitting) to a better fit.
- **The Critical Peak:** As complexity grows further, error begins to rise sharply, reaching a peak at the interpolation threshold. This is the point where the model has just enough parameters to perfectly fit (interpolate) all training data, often capturing noise as if it were signal.
- **Second Descent (Modern Regime):** Surprisingly, increasing complexity beyond the interpolation threshold into the “overparameterized” regime causes the test error to decrease again. In this regime, even though the model perfectly fits the training data, its generalization on unseen data improves, often achieving even better results than the “optimal” model in the classical regime.

Why It Happens? While research is ongoing, several mechanisms are believed to drive this behavior:

- **Inductive Bias & Implicit Regularization:** Optimization algorithms like Stochastic Gradient Descent (SGD) tend to select the “simplest” or “smoothest” solution among the many possible functions that fit well with the training data.
- **Smoothing Effect:** In the highly overparameterized regime, the model has so many degrees of freedom that it can fit the training data without becoming erratic between points, leading to a smoother, more generalizable function.
- **Effective Dimension:** Larger models may have a lower “effective” dimension than their parameter count suggests, focusing only on the most relevant features for generalization.

Practical Implications

- **Rethinking Model Size:** It suggests that “bigger is/could be better,” which underpins the success of massive models like Transformers and LLMs.
- **Data Noise Sensitivity:** The peak at the interpolation threshold is significantly amplified by noisy data or incorrect labels.
- **Optimization over Selection:** Choosing the right optimization path (like SGD) and training duration is as critical as choosing the model architecture itself.

Recent Developments in LLMs

In LLMs, double descent manifests in scale-up laws: ImageNet-CNNs, ResNets, Transformers show test loss peaking at train-fit threshold, reviving with more parameters and data.

- Epoch-wise Double Descent: For fixed large model, loss decreases, spikes at mid-training (overfitting), then descends longer via SGD dynamics, reversing overfitting.
- Grokking (“delayed generalization”): it is a phenomenon where a model suddenly transitions from overfitting to perfect generalization after training for a long period (many epochs) beyond initial convergence. Grokking is a delayed generalization over time (epoch), which is related to double descent (delayed generalization over model size/complexity). Some researchers are exploring a unified approach via “staggered learning.”
- Research indicates that double descent, which once thought to be unique to supervised learning, also manifests in unsupervised and self-supervised learning (though its appearance is more sensitive to model architecture and data noise).

Further Exploration

1. Bias-Variance Decomposition Derivation

Suppose the true relationship is $y = f(x) + \epsilon$, where $\mathbb{E}[\epsilon] = 0$ and $\text{Var}(\epsilon) = \sigma^2$. Let $\hat{f}$ be our model estimate trained on a dataset D . The expected squared error at a point x ($\text{Err}(x)$) is (with expectations being taken over both the noise ϵ and the choice of training data D)

$$\begin{aligned}\text{Err}(x) &= \mathbb{E} \left[(y - \hat{f})^2 \right] \\ &= \mathbb{E} \left[(f(x) + \epsilon - \hat{f})^2 \right] \\ &= \mathbb{E} \left[(f(x) - \hat{f})^2 \right] + \mathbb{E}[\epsilon^2] + 2\mathbb{E}[(f(x) - \hat{f})\epsilon]\end{aligned}\tag{7.2.2}$$

Since ϵ is independent of the training data and $\mathbb{E}[\epsilon] = 0$, the cross-term vanishes. Thus:

$$\text{Err}(x) = \mathbb{E} \left[(f(x) - \hat{f})^2 \right] + \sigma^2.\tag{7.2.3}$$

Now, we decompose the term $\mathbb{E}[(f(x) - \hat{f})^2]$ by adding and subtracting $\mathbb{E}[\hat{f}]$:

$$\begin{aligned}\mathbb{E} \left[(f(x) - \hat{f})^2 \right] &= \mathbb{E} \left[(f(x) - \mathbb{E}[\hat{f}] + \mathbb{E}[\hat{f}] - \hat{f})^2 \right] \\ &= \mathbb{E} \left[(f(x) - \mathbb{E}[\hat{f}])^2 \right] + \mathbb{E} \left[(\mathbb{E}[\hat{f}] - \hat{f})^2 \right] + 2\mathbb{E}[(f(x) - \mathbb{E}[\hat{f}])(\mathbb{E}[\hat{f}] - \hat{f})]\end{aligned}\tag{7.2.4}$$

Note that:

This gives the final decomposition (which is ([07a=01])):

$$\text{Err}(x) = \underbrace{(f(x) - \mathbb{E}[\hat{f}])^2}_{\text{Bias}^2} + \underbrace{\mathbb{E} \left[(\hat{f} - \mathbb{E}[\hat{f}])^2 \right]}_{\text{Variance}} + \underbrace{\sigma^2}_{\text{Irreducible Error}}.\tag{7.2.5}$$

Note that:

- $(f(x) - \mathbb{E}[\hat{f}])^2$ is deviation relative to the expectation, so $\mathbb{E}[(f(x) - \mathbb{E}[\hat{f}])^2] = (f(x) - \mathbb{E}[\hat{f}])^2$ (**Bias**²).
- $\mathbb{E}[(\hat{f} - \mathbb{E}[\hat{f}])^2]$ is the **Variance** of $\hat{f}$.
- The cross-term is zero because $\mathbb{E}[(\mathbb{E}[\hat{f}] - \hat{f})(f(x) - \mathbb{E}[\hat{f}])] = \mathbb{E}[\hat{f}] - \mathbb{E}[\hat{f}] = 0$.
- This gives the final decomposition

$$\text{Err}(x) = \underbrace{(f(x) - \mathbb{E}[\hat{f}])^2}_{\text{Bias}^2} + \underbrace{\mathbb{E} \left[(\hat{f} - \mathbb{E}[\hat{f}])^2 \right]}_{\text{Variance}} + \underbrace{\sigma^2}_{\text{Irreducible Error}}.\tag{7.2.6}$$

2. Key papers on double descent:

- P. Nakkiran et al., “Deep Double Descent: Where Bigger Models and More Data Hurt” (ICLR 2020, arXiv:1912.02292).
- A. Power et al., “Grokking: Generalization Beyond Overfitting on Small Algorithmic Datasets” (arXiv:2201.02177).
- Y. Huang et al., “Unified View of Grokking, Double Descent and Emergent Abilities” (arXiv:2402.15175).
- Y. Gu et al., “Unraveling the Enigma of Double Descent” (arXiv:2310.13572).
- M. Belkin et al., “Reconciling modern machine learning practice and the classical bias-variance tradeoff” (PNAS 2019).

7.3: Explainable AI (XAI)

Based on SD Sect. 7.5. Explainable AI (XAI) addresses the opacity of complex ML models, enabling scientists and engineers to debug, validate, and deploy trustworthy systems in production.

- Core Concepts

- Interpretability vs. Explainability

Interpretability targets intrinsic model transparency (e.g., simulatability, decomposability), which is best for simple models (e.g. decision tree). Explainability generates human-readable justifications for outputs, often post-hoc (after-the-fact) for black-box models like transformers.

- Local vs. Global Explanations

Local methods explain single predictions (e.g., per-instance attributions); global ones reveal overall model behavior across datasets (crucial for drift detection).

- Trade-offs

Higher accuracy often reduces interpretability. Tips: use surrogates (simple, interpretable models) or approximations (e.g., use LIME or SHAP to create a simplified map) for balance approach. Evaluating fidelity via metrics like explanation accuracy and sufficiency (see below).

- Advanced Techniques

- Gradient-Based Attribution: it helps us to understand which parts of an input caused an AI to make its final decision.

E.g. Integrated Gradients: they identify which inputs most influenced an AI's decision by measuring how the model's output changes as one "fade in" the input from a blank baseline (starting point) to the final result.

- SHAP (SHapley Additive exPlanations)

SHAP uses game theory for consistent additive explanations (or, it uses game theory to fairly divide "credit" among data features). It is based on Shapley values, a concept from cooperative game theory introduced by Lloyd Shapley in 1952.

- LIME (Local Interpretable Model-agnostic Explanations)

LIME builds a simple, "interpretable" mini-model (a surrogate) to mimic how the complex AI behaves for specific predictions. It focuses on that local area, testing nearby data points to efficiently figure out which features mattered most in that moment.

- Attention & Prototypes

Examine internal model weights to decode reasoning. Attention Maps reveal which tokens or pixels a transformer aggregates during processing, while Prototype-based models compare new inputs directly to definitive, pre-understood examples.

Use Prototypes, which compare new data to "perfect examples" which AI already understands.

- Counterfactuals & Influence Functions

In XAI, spurious correlations happen when an AI model learns misleading, non-causal links (e.g., a dog always having grass in its picture) instead of true features (and causing poor generalization). Approaches to counter them: use techniques from causal inference like Counterfactuals (changing irrelevant features to see if predictions change) or Influence Functions (finding influential training data points) to build robustness.

- Concept-Based Interpretability: Moves beyond individual pixel/word features to human-scale ideas.

Concept Bottleneck Models (CBMs): Built intrinsically by forcing a neural network to predict high-level human concepts in an intermediate layer before rendering a final prediction. In other words: design and train the network from scratch with a literal “bottleneck” layer that outputs human concepts before the final answer.

Linear Probes and TCAV: Post-hoc tools used to inspect standard networks. Linear probes check internal layers to see if human concepts are represented. Testing with Concept Activation Vectors (TCAV) then uses directional derivatives along those probes to quantify how much an abstract concept influences a global model prediction. In other words: apply to standard, already-trained “black-box” models. Linear probes look for concepts hidden in the layers, and TCAV uses calculus to see how important they are.

- Future Directions

- Shifting from post-hoc interpretations toward hybrid neurosymbolic models (embedding logical rules directly into model design) to provide “explainability by design.”
- Toward integrating causal XAI, utilizing frameworks (like do-calculus: a set of rules used to determine if a causal effect can be predicted from data by transforming interventional “what-if” queries into probabilistic expressions).
- Advancements are expected to transform XAI from a transparency tool into a core component of “reasoning-driven intelligence.” This is particularly true for high-stakes sectors (e.g., healthcare, FIN, etc.) where understanding the “why” behind an outcome is legally and operational critical.

1. Mechanistic interpretability (MI)

MI is a specialized subfield of XAI. While traditional XAI focuses on communicating “why” a model makes a specific prediction, MI seeks to explore the “how” by mapping internal components (e.g., neurons in NN/DL) to human-understandable algorithms.

Core Concepts

- Mechanistic vs. Behavioral Interpretability: Mechanistic uncovers causal circuits (e.g., induction heads in transformers); behavioral tests inputs/outputs—prioritize circuits for intervention and scaling laws.
- Superposition and Polysemanticity: Because models have more “features” than they have physical neurons, they “cram” multiple concepts into the same neuron to save space. This means a single neuron is polysemantic: it activates for many unrelated things at once. Researchers use Sparse Autoencoders (SAEs) to “untangle” this mess. The SAE identifies the distinct, individual concepts (called monosemantic features) that were overlapping, effectively clarifying the purpose of the mid-layer neurons where over 90% of activations are jumbled and multi-purpose.
- Circuit Discovery: It is the process of mapping the specific internal “wiring” (subgraphs) an AI model uses to perform a task. Researchers find these subgraphs by using activation patching to swap internal signals between different inputs to see which components are necessary for a correct answer. Similarly, one could use path patching to trace exactly how information flows between specific neurons. To ensure the results are reliable, a common best practice is to only include components that have a significant causal impact and to verify that the discovered circuit still works on out-of-distribution (OOD) data.

Advanced Techniques

- Activation Patching: Logit lens + mean ablation to trace circuits; best practice: Patch top- k activations per layer, compute path probabilities with log-additive interventions.
- Sparse Autoencoders (SAEs): Train an autoencoder with a “sparsity” constraint (e.g., with an L_1 penalty) on the hidden layer. This forces it to activate only a few features at a time for any given input. This makes the features more distinct and interpretable than the network’s original entangled neurons.
- Transformer Circuits: These are specific network of components (like neurons and attention heads) that work together to perform a single task (e.g. a Previous-Token Head that simply moves information from the immediate past token to the current one). To find these circuits without manual guesswork, researchers use the ACDC (Automated Circuit

Discovery) framework, which systematically prunes unimportant connections from the model's computational graph while monitoring for performance drops. A best practice for efficiency is to prune the pathways that account for less than 5% of the total computation; this typically identifies a sparse sub-graph that still maintains roughly 95% fidelity to the original model's behavior, effectively isolating the logic of the task from the surrounding noise.

- Dictionary Learning: This uses SAEs to translate a model's internal math into a dictionary of clean, human-understandable concepts. To ensure these concepts are reliable rather than random noise, the current standard is to bootstrap the process by testing across 10,000 different seeds, which helps identify the most stable features, rank the features using a monosemanticity score, select features with high score (e.g., 80%). This helps ensuring the entries in the dictionary correspond to single, clear ideas with high correlation.

2. References

1. XAI:

- Alain, G., & Bengio, Y. (2017). *Understanding intermediate layers using linear classifier probes*. International Conference on Learning Representations (ICLR) Workshop.
- Cunningham et al. (2023). *Sparse autoencoders find highly interpretable features*. arXiv:2309.08600.
- Kim, B., Wattenberg, M., Gilmer, J., Cai, C., Wexler, J., Viegas, F., & Tetlock, A. (2018). Interpretability beyond feature attribution: Quantitative testing with concept activation vectors (TCAV). *International Conference on Machine Learning*, 2668–2677.
- Koh, P. W., Nguyen, T., Tang, Y. S., Mussmann, S., Pierson, E., Kim, B., & Liang, P. (2020). Concept bottleneck models. In *Proceedings of the 37th International Conference on Machine Learning* (pp. 5338–5348).
- Lundberg, S. M., & Lee, S.-I. (2017). A unified approach to interpreting model predictions. *Advances in Neural Information Processing Systems*, 30.
- Belle, V., & Papantonis, I. (2021). Principles and practice of explainable machine learning. *Frontiers in Big Data*, 4, 688969.
- Oikarinen, T. P., Das, S., Nguyen, L. M., & Weng, T.-W. (2023). Label-free concept bottleneck models. In *International Conference on Learning Representations (ICLR)*.
- Rao, V. (2025). *The limits of AI explainability*. arXiv:2504.20676.
- Ribeiro, M. T., Singh, S., & Guestrin, C. (2016). “Why should I trust you?”: Explaining the predictions of any classifier. *Proc. KDD’16*, 1135–1144.
- Sundararajan, M., Taly, A., & Yan, Q. (2017). Axiomatic attribution for deep networks. *International Conference on Machine Learning*, 3319–3328. arXiv:1703.01365

2. MI

- Conmy, A., Mavor-Parker, A., Lynch, A., Heimersheim, S., & Garriga-Alonso, A. (2023). Towards Automated Circuit Discovery for Mechanistic Interpretability. *NeurIPS* 36. arXiv:2304.14997.
- Elhage, N., Hume, T., Olsson, C., Schiefer, N., Henighan, T., Kravec, S., Hatfield-Dodds, Z., Lasenby, R., Drain, D., Chen, C., Grosse, R., McCandlish, S., Kaplan, J., Amodei, D., Wattenberg, M., & Olah, C. (2022). Toy Models of Superposition. Anthropic Transformer Circuits Thread.
- Nanda, N., & Meyer, B. (2022). TransformerLens: A Library for Mechanistic Interpretability of Generative Language Models.

3. Surveys & Tools

- Atf, Z. & Lewis, P. (2025). Is trust correlated with explainability in AI? A meta-analysis. arXiv:2504.12529.

- De Silva et al. (2025). A multi-layered research framework for human-centered AI. arXiv:2504.13926.
- Long et al. (2025). Explainable AI: The latest advancements and new trends. arXiv:2505.07005.

8: PREPROCESSING AND UNSUPERVISED LEARNING

8.1: Chapter 8 Summary

8.2: Dimensionality Reduction Methods

8.3: Clustering Methods

8.4: Vector DBs & Similarity Search at Scale (The Big Picture)

This page titled [8: Preprocessing and Unsupervised Learning](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by Hei-Chi Chan.

8.1: Chapter 8 Summary

This is a recap of SD chapter 8, on preprocessing and unsupervised learning.

- Well-known unsupervised learning models are presented in: Sect. 8.2 (dimensional reduction), Sect. 8.3 (PCA), Sect. 8.4 (clustering), and Sect. 8.5 (density estimation).
- Other sections

Sect. 8.1. Normalization and transformation

1. Normalization: $x \rightarrow (x - \min x) / (\max x - \min x)$. Sensitive to outliers. For new (and/or streaming) data: it fails if new values fall outside the current range. Note: In streaming contexts: a single new outlier that exceeds the previous max requires recalculating the entire scale, potentially distorting previously processed data. Consider using moving windows (e.g., use the most recent N data points to track min /max). Computational cost: low (just need min /max with simple comparison with minimal overflow or precision risk). Best with image data (image pixels 0-255) or fixed-range sensors; good with data from unknown distributions.
2. Standardization (scaling to zero mean, unit variance). More robust to outliers. For new (and/or streaming) data: new values simply receive higher Z-scores. Computational cost: moderate (due to tracking running statistics, such as running mean and running variance). When dealing with high volume of data: overflow risk and precision risk are high; require stable algorithms. Note: Periodically monitor for data drift; if the distribution shifts significantly, the scaler parameters must be retrained to avoid model performance decay. Note: if unsure which one to use (between normalization and standardization), a rule of thumb suggests starting with standardization, as it often leads to more stable convergence.

Tips: Saving scaler parameters (mean, SD, min, max) as model artifacts is essential to prevent “training-serving skew” (i.e., the data the model sees in production is scaled differently than the data it learned from during training).

3. When preprocessing data, calculate transformation parameters (e.g., the mean and SD for scaling) using only the training dataset. These fixed statistics are then applied to the validation and test sets to transform them. This ensures that information from the evaluation data does not “leak” into the training process (no data leakage).
4. Tree-based algorithms (e.g., Random Forest and XGBoost) are unique because they generally do not require feature scaling. These models make decisions by splitting data based on value thresholds (e.g., “is x greater than 30?”), so the relative magnitude of different features does not affect the mathematical logic of the split. However, there are important exceptions where scaling becomes necessary: when dealing with extreme outliers/distributions that might skew the learning process, or when using hybrid models that combine trees with distance-based algorithms like KNN or SVM.
5. Pre-scale skewed features. Apply transformations that aim to make the feature distribution more symmetrical, often moving it closer to a normal distribution. Use log/Box-Cox transformation.

Sect. 8.6. Missing values and imputation methods

1. Classify missing values as Missing Completely at Random (MCAR), Missing at Random (MAR), or Missing Not at Random (MNAR) to tailor imputation or modeling strategy. Apply appropriate techniques: such as deletion for MCAR, multiple imputation for MAR, or specialized modeling for MNAR. This is to ensure the integrity and accuracy of the AI/ML models.
2. Simple imputation techniques: For numerical data, use mean or median as substitution. For categorical features use the mode. For more complex datasets, use KNN which preserves local structure by estimating missing entries based on the similarity of neighboring data points.
3. Advanced imputation techniques: MICE (Multivariate Imputation by Chained Equations) and regression model relationships between features to predict missing entries iteratively. MissForest: it uses Random Forest to estimate and fill in missing values in a dataset. Certain tree-based algorithms (e.g., XGBoost or LightGBM) can be used to handle missing values as well.
4. Adding missing indicators: This involves creating binary flags that denote whether a specific value was originally absent,

allowing models to explicitly learn if the “missingness” itself carries predictive information. This is particularly useful when data is not missing at random, as it captures systematic patterns that simple imputation might otherwise obscure.

5. Imputation techniques for time-series data: forward-fill (or Last Observation Carried Forward) propagates the most recent valid data point to maintain continuity. Interpolation methods (e.g., linear interpolation) estimate missing values based on neighboring data points to reflect trends or seasonality. For complex datasets, advanced models leverage DL to capture intricate longitudinal dependencies and uncertainty.
6. When using cross-validation loop, imputation must be performed within each CV fold, ensuring that missing values in the validation set are filled using only the statistics or models derived from the same training fold (to prevent data leakage).
7. Multiple Imputation: This replaces each missing value with several different predicted values to create multiple “complete” datasets, capturing the inherent uncertainty of the missing data. Each dataset is analyzed independently, and the results are pooled into a final estimate. This provides more statistically valid standard errors and confidence intervals than single-imputation methods. Essential for understanding the variance caused by missingness is as critical as the prediction itself.

Sect. 8.7. Over- and undersampling, augmentation

1. Oversampling techniques: use SMOTE (Synthetic Minority Over-sampling Technique), a statistical method used to address imbalanced datasets by generating synthetic examples of the minority class.

Note: Use Tomek links or Edited Nearest Neighbors (ENNs) to reduce overlap after oversampling. Tomek Links are pairs of mutually nearest neighbors from different classes, where the majority instance is typically removed to clean the decision boundary. ENN is an aggressive cleaning technique that removes any data point whose class label does not match the majority vote of its k nearest neighbors. They are used in missing data pipelines as post-imputation cleaning filters to ensure that newly filled values do not introduce noise or blur class boundaries.

2. Undersampling techniques: In addition to randomly removing majority classes, one could use: NearMiss (strategically retains majority class samples based on their proximity to the minority class, either focusing on those closest to the boundary or those furthest from other majority samples to maintain decision regions), cluster-based undersampling (partitions the majority class into clusters using algorithms like K-Means, then replaces each cluster with its centroid or a representative sample). These are targeted approaches, ensuring that the most informative data points are preserved while reducing the risk of discarding potentially significant patterns.
3. Data augmentation
 1. For tabular data: Use targeted perturbations within plausible ranges (feature-wise jitter, noise infusion). Leverage Domain Randomization to expose models to diverse but realistic variations.
 2. For sequential data: For time-series data, add jitter to timestamps, introduce small lagged feature perturbations, or simulate realistic seasonality. For text/sequence, use back-translation (creating new text by translating a sentence to a different language and then translating the result back to the source language), synonym replacement (where appropriate) to increase linguistic diversity.
 3. For image/audio: Standard augmentations (flips, rotations, color jitter, cropping, noise) with careful consideration of label invariances.
 4. Safety and validity: Ensure augmented samples remain label-consistent; validate augmentations with small pilot experiments.
 5. Evaluation strategy: Separate augmentation for training vs validation; avoid leaking augmented data into validation sets.

Further Exploration

Key references include

- Kuhn, M., & Johnson, K. (2019). *Feature Engineering and Selection: A Practical Approach for Predictive Modeling*. CRC Press.

- Géron, A. (2022). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow* (3rd ed.). O'Reilly Media.
- Kuhn, M., & Johnson, K. (2013). *Applied Predictive Modeling*. Springer.

This page titled [8.1: Chapter 8 Summary](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by Hei-Chi Chan.

8.2: Dimensionality Reduction Methods

Based on SD Sects. 8.2 and 8.3.

Dimensionality reduction methods are designed to reduce/condense high-dimensional data while retaining essential structures. They are vital for tasks such as scalable modeling and visualization.

1. Linear Methods

Principal Component Analysis (PCA)

1. PCA transforms a large set of potentially correlated variables into a smaller set of uncorrelated new variables called principal components. This process preserves the most important information (maximum variance) from the original data, making it easier to visualize and analyze.
2. Let $\mathbf{X}$ is the mean-centered $n \times p$ input data matrix, with rows as n samples (observations) and columns as p original features (variables). It is centered by subtracting the column means. centering subtracts column means. The goal is to project $\mathbf{X}$ to a reduced $n \times k$ matrix $\mathbf{Z}$ (p features reduced to k principal components, with $k \ll p$), given by $\mathbf{Z} = \mathbf{X}\mathbf{W}$, where $\mathbf{W}$ is a matrix of unit vectors. Note that each row $\mathbf{z}_i$ in $\mathbf{Z}$ gives the coordinates of original sample $\mathbf{x}_i$ in the new principal component basis.
3. Derivation: Optimize the Lagrangian $\mathcal{L} = \mathbf{w}^T \Sigma \mathbf{w} - \lambda(\mathbf{w}^T \mathbf{w} - 1)$, where

$$\Sigma = \frac{1}{n} \mathbf{X}^T \mathbf{X} \quad (8.2.1)$$

. Note that the first term is the covariance, with the constraint to keep $\mathbf{w}$ at unit length. This gives $\Sigma \mathbf{w} = \lambda \mathbf{w}$. $\mathbf{W}$ is the $p \times k$ matrix with the top k eigenvectors of Σ (the principal components).

Independent Component Analysis (ICA)

1. ICA seeks to recover independent source signals from their linear mixtures by maximizing non-Gaussianity. But why? If a signal is a mixture of independent sources, it is “more Gaussian” than the sources themselves (per the Central Limit Theorem). Therefore, to find the original sources, we must find the unmixing that makes the source signals as non-Gaussian as possible.
2. Key ideas and the algorithm.

ICA implementation could be achieved via negentropy, a measure of deviation from Gaussianity.

1. Entropy and Negentropy

Recall that the (differential) entropy H of a random vector y with density $p(y)$ is given by $H(y) = - \int p(y) \log p(y) dy$.

The negentropy $J(y)$ is defined as the distance between the entropy of a Gaussian variable (y_{gauss}) and the entropy of y (assuming both have the same covariance matrix): $J(y) = H(y_{gauss}) - H(y)$.

2. Key properties of negentropy

(1) $J(y) \geq 0$. (2) $J(y) = 0$ if and only if y is Gaussian. (3) Because $H(y_{gauss})$ is constant for a fixed variance, **maximizing $J(y)$ is equivalent to minimizing $H(y)$** . The more y likes a Gaussian, the smaller $J(y)$ gets. We want y to be as non-Gaussian as it can be.

3. The ICA objective.

Let x be the observed mixtures and W be the unmixing matrix we want to find. We define our estimated source as $y = Wx$.

To ensure we are measuring “shape” (non-Gaussianity) rather than just scaling, we constrain y to have unit variance ($E[y^2] = 1$). Under this constraint, the goal is: $\max_W J(Wx)$.

4. FastICA (a modern implementation of the ICA) is an efficient iterative fixed-point algorithm that maximizes negentropy by using non-quadratic function approximations (G) to find the unmixing weights through a “Newton-like” optimization scheme, rather than relying on computationally expensive gradient descent.

2. Other Methods

t-distributed Stochastic Neighbor Embedding (t-SNE)

1. It reduces high-dimensional data to 2D or 3D by preserving local similarities through probability distributions.
2. In the high-dimensional space, pairwise similarities are modeled as conditional Gaussian probabilities

$$p_{j|i} = \frac{\exp(-\|x_i - x_j\|^2 / (2\sigma_i^2))}{\sum_{k \neq i} \exp(-\|x_i - x_k\|^2 / (2\sigma_i^2))}, \quad (8.2.2)$$

symmetrized to joint probabilities

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N}. \quad (8.2.3)$$

In the low-dimensional space, similarities use a heavy-tailed t-distribution

$$q_{ij} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_{k \neq l} (1 + \|y_k - y_l\|^2)^{-1}}. \quad (8.2.4)$$

The heavy tails of the t-distribution allocate larger distances to dissimilar points in 2D/3D, preventing distinct clusters from collapsing onto a crowded region.

3. Optimization. t-SNE is implemented via minimizing the Kullback-Leibler divergence

$$C = \sum_i KL(P_i || Q_i) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}} \quad (8.2.5)$$

(e.g., via gradient descent), balancing attraction for similar points and repulsion for dissimilar ones.

Uniform Manifold Approximation and Projection (UMAP)

1. It is a non-linear dimensionality reduction technique based on Riemannian geometry. It is primarily used for high-dimensional data visualization and is known for being significantly faster than t-SNE while better preserving the global structure of the data.
2. UMAP assumes that data is uniformly distributed on a Riemannian manifold and that this manifold is locally connected. The algorithm works in two main phases:
 1. Graph Construction: Build a “fuzzy” weighted graph representing the high-dimensional data structure.
 2. Layout Optimization: Optimize a low-dimensional layout of this graph using cross-entropy.
3. High-dimensional similarities

For a data point x_i , the similarity between x_i and a neighbor x_j is defined by:

$$w(x_i, x_j) = \exp \left(-\frac{\max(0, d(x_i, x_j) - \rho_i)}{\sigma_i} \right) \quad (8.2.6)$$

where:

- $d(x_i, x_j)$ is the distance between points.
- ρ_i is the distance to the nearest neighbor of x_i .
- σ_i is a normalization parameter.

The final symmetrized weight p_{ij} is: $p_{ij} = (\rho_{ij} + \rho_{ji}) - \rho_{ij}\rho_{ji}$.

4. Low-dimensional similarities

In the target low-dimensional space, similarity q_{ij} between points y_i and y_j is modeled as: $q_{ij} = (1 + a\|y_i - y_j\|^{2b})^{-1}$. Here a and b are hyperparameters.

5. Optimization (Cost Function)

UMAP minimizes the Fuzzy Set Cross Entropy:

$$CE(P, Q) = \sum_{i,j} \left[p_{ij} \log \left(\frac{p_{ij}}{q_{ij}} \right) + (1 - p_{ij}) \log \left(\frac{1 - p_{ij}}{1 - q_{ij}} \right) \right]. \quad (8.2.7)$$

6. Practical advantages:

- Speed: Significantly more efficient than t-SNE for large datasets.
- Global Structure: Superior preservation of broader cluster relationships.

Further Exploration

Key references

- Pearson, K. (1901). LIII. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11), 559–572.
- Comon, P. (1994). Independent component analysis, A new concept? *Signal Processing*, 36(3), 287–314.
- Hyvärinen, A., & Oja, E. (2000). Independent component analysis: Algorithms and applications. *Neural Networks*, 13(4-5), 411–430.
- van der Maaten, L., & Hinton, G. (2008). Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9(86), 2579–2605.
- McInnes, L., Healy, J., & Melville, J. (2018). UMAP: Uniform manifold approximation and projection for dimension reduction, *J. Open Source Softw.*, 3, Article 861.
- McInnes, L., Healy, J., Saul, N., & Großberger, L. (2018). UMAP: Uniform manifold approximation and projection. *Journal of Open Source Software*, 3(29), 861.

8.3: Clustering Methods

Based on SD Sects. 8.4 and 8.5.

1. Centroid-Based: K-Means

Key ideas: Minimizes the Within-Cluster Sum of Squares (WCSS).

- Objective Function:

$$J = \sum_{i=1}^k \sum_{x \in C_i} \|x - \mu_i\|^2 \quad (8.3.1)$$

Where μ_i is the centroid of cluster C_i .

- Derivation (Lloyd's Algorithm):

1. Assignment Step: Fix μ_i . Minimize J by assigning each x to the nearest centroid:

$$C_i^{(t)} = \{x : \|x - \mu_i^{(t)}\|^2 \leq \|x - \mu_j^{(t)}\|^2 \forall j\}$$

2. Update Step: Fix C_i . Set the gradient of J with respect to μ_i to zero:

$$\frac{\partial J}{\partial \mu_i} = -2 \sum_{x \in C_i} (x - \mu_i) = 0 \implies \mu_i = \frac{1}{|C_i|} \sum_{x \in C_i} x \quad (8.3.2)$$

- 2025 Applied Context: Use [FAISS](#) for GPU-accelerated $O(n \log n)$ k-means on billion-scale vector embeddings.

2. Distributional: Gaussian Mixture Models (GMM)

Key ideas: Models data as a weighted sum of K multivariate normal distributions.

- Probability Density:

$$p(x) = \sum_{k=1}^K \pi_k \mathcal{N}(x | \mu_k, \Sigma_k) \text{ where } \sum \pi_k = 1 \quad (8.3.3)$$

- Derivation (Expectation-Maximization):

1. E-step: Calculate the "responsibility" $\gamma(z_{nk})$ using Bayes' Theorem:

$$\gamma(z_{nk}) = \frac{\pi_k \mathcal{N}(x_n | \mu_k, \Sigma_k)}{\sum_{j=1}^K \pi_j \mathcal{N}(x_n | \mu_j, \Sigma_j)} \quad (8.3.4)$$

2. M-step: Update parameters to maximize likelihood:

$$\mu_k^{new} = \frac{1}{N_k} \sum_{n=1}^N \gamma(z_{nk}) x_n, \quad \Sigma_k^{new} = \frac{1}{N_k} \sum_{n=1}^N \gamma(z_{nk}) (x_n - \mu_k)(x_n - \mu_k)^T \quad (8.3.5)$$

3. Density-Based: DBSCAN/HDBSCAN

Key ideas: Clusters are maximal connected components of density-connected points.

- HDBSCAN Derivation: Transforms space using Mutual Reachability Distance:

$$d_{mreach-k}(a, b) = \max \{core_k(a), core_k(b), d(a, b)\}$$

- Optimization: Constructs a Minimum Spanning Tree (MST) and prunes it to maximize Cluster Stability:

$$\sum_{p \in \text{cluster}} (\lambda_p - \lambda_{birth}) \text{ where } \lambda = 1/\text{distance}$$

4. Graph-Based: Spectral Clustering

Key ideas: Uses the spectrum of the Laplacian matrix for dimensionality reduction before clustering.

- Derivation:

1. Construct Similarity Matrix W (e.g., $w_{ij} = \exp(-\gamma \|x_i - x_j\|^2)$).
2. Define Degree Matrix D ($d_{ii} = \sum_j w_{ij}$) and Normalized Laplacian $L_{rw} = I - D^{-1}W$.
3. Minimize Normalized Cut (Ncut):

$$\text{Ncut}(A, B) = \frac{\text{cut}(A, B)}{\text{vol}(A)} + \frac{\text{cut}(A, B)}{\text{vol}(B)} \quad (8.3.6)$$

- Result: The solution is approximated by the eigenvectors of the k smallest eigenvalues of L .

5. Deep Clustering (DEC)

Key ideas: Jointly learns representations and assignments via DNNs.

- Soft Assignment: Uses Student's t-distribution:

$$q_{ij} = \frac{(1 + \|z_i - \mu_j\|^2 / \alpha)^{-(\alpha+1)/2}}{\sum_{j'} (1 + \|z_i - \mu_{j'}\|^2 / \alpha)^{-(\alpha+1)/2}} \quad (8.3.7)$$

- Optimization: Minimizes KL Divergence against a target distribution P :

$$L = KL(P||Q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}} \quad (8.3.8)$$

8.4: Vector DBs & Similarity Search at Scale (The Big Picture)

Here are some major topics on vector databases (DBs) and similarity search at scale (and related topics on implementation, optimization, and system-level challenges).

1. A brief history

The rise of vector databases was driven by the need for a new way to search and understand the massive amounts of unstructured data (like images, videos, and text) that traditional keyword-based databases could not handle. This led to the emergence of vector embeddings, which are numerical representations of data that capture its semantic meaning or context. Traditional exact-match searches are not the solution for this data because they only look for identical keywords and miss the conceptual relationships that are key to understanding unstructured content. Instead, a new approach, “similarity search,” rose to prominence, allowing users to find and retrieve data based on meaning, where similar concepts are mathematically close to one another in a vector database. It allows for small errors in matching (“approximate”) in exchange for dramatically faster performance on massive datasets (with billions of entries).

2. Approximate Nearest Neighbor (ANN) algorithms

The most common industrial-scale methods for similarity search are built around Approximate Nearest Neighbor (ANN) algorithms that can handle massive, high-dimensional datasets with billions of vectors. Below are some major ANN algorithms.

- Hierarchical Navigable Small World (HNSW):

A graph-based ANN algorithm. One of the top-performing and most popular ANN algorithms. It builds a multi-layer graph where each layer is a navigable small-world graph. This allows for rapid, “greedy” searches to find the nearest neighbors. HNSW is highly accurate and fast but can be memory-intensive.

- Spotify's ANNOY:

See the note “DL Notes (OER): Approximate Nearest Neighbor (ANN) Search Annoy from Spotify”

- Google's ScaleNN:

Designed for efficient similarity searches like Maximum Inner Product Search (MIPS), key innovations include: Anisotropic vector quantization (AVQ), asymmetric distance computation, optimized search space pruning, and hardware-aware optimizations

- Microsoft's DiskANN:

A graph-based ANN algorithm. For high-throughput scenarios that uses Solid-State Drives (SSDs) to efficiently handle massive datasets that exceed memory capacity. key innovations include: Optimized SSD-based index storage (that drastically reducing memory requirements while maintaining high recall and low latency), a novel Vamana graph structure (robust to dynamic index updates without expensive rebuilds), and advanced vector quantization techniques (balancing memory efficiency and accuracy).

- Meta's FAISS

An open-source library for efficient similarity search and clustering of high-dimensional vectors, rapid retrieval in DBs with millions to billions of entries. Include: advanced indexing methods such as IVF, PQ, HNSW algorithms, etc., aiming to balance precision, speed, and memory usage. FAISS leverages GPU acceleration to perform millions of similarity checks in milliseconds, with NVIDIA cuVS integration to double query speed and improve GPU memory efficiency.

- Locality-Sensitive Hashing (LSH):

This family of algorithms uses hash functions that map similar input items to the same “buckets” with high probability. LSH significantly speeds up the search by restricting the comparisons to only the vectors within the same bucket as the query. May require multiple hash tables to achieve high accuracy, leading to a significant memory footprint.

- Inverted File Index (IVF):

For accelerating ANN searches. This method partitions the vector space into “cells” or “clusters.” At query time, it only searches a small number of the most relevant clusters, avoiding a full scan of the entire dataset. It is very scalable and was popularized by Facebook’s FAISS library.

- Product Quantization (PQ):

For accelerating ANN searches. Significantly reducing the memory footprint of a dataset and speeding up similarity calculations. A lossy compression technique. A divide-and-conquer strategy to compress high-dimensional vectors (involving splitting vectors into subvectors, building codebooks, encoding each subvector with the index of the nearest centroid in the codebook). Key variations: Optimized Product Quantization (OPQ), Residual Quantization (RQ), Additive Quantization (AQ), and Inverted File Index with PQ (IVF-PQ).

3. Database architecture

- Core architecture of modern vector databases: designs of different vector databases (e.g., Pinecone, Milvus, Weaviate, Qdrant) w/ different focuses on storage strategies, distributed indexing, and query processing pipelines.
- Data ingestion at scale: Address the challenges of incremental data ingestion, real-time updates, and eventual consistency in vector database systems.
- Tiered storage and cost optimization: Leverage memory, SSDs, and object storage to create a tiered storage strategy that balances performance and cost for massive datasets.

4. Advanced retrieval strategies for RAG

- Hybrid search: combine traditional keyword-based search (sparse vectors) with semantic vector search (dense vectors) to improve relevance and accuracy.
- Pre-filtering and post-filtering: The trade-offs and advanced techniques for filtering search results based on metadata attributes.
- Re-ranking: Strategies using more powerful, but computationally expensive, models to refine the top-k results from an initial vector search.
- Temporal and multi-modal search: Explore techniques for incorporating temporal data and enabling similarity searches across different data types, such as text and images.

5. Other topics

- Production deployment and operations
 - Scalability and sharding: Different sharding strategies for distributing vector data across multiple nodes and scaling horizontally.
 - Benchmarking and evaluation: Rigorous benchmarks to compare vector databases and ANN algorithms (e.g., using metrics like recall@k). Tune indexing parameters for their specific use case.
 - Monitoring and observability: Essential metrics to monitor for vector database health and performance, including query latency, index build time, and resource utilization.
 - Use cases: fraud detection, multi-modal search (text-to-image), and log anomaly detection.
- Embedding management and best practices

- Embedding model selection and fine-tuning: Criteria for choosing the right embedding model and how to fine-tune it for a specific domain to improve search relevance.
 - Vector updates and data drift: Address the challenge of managing changing data and evolving embedding models over time and the impact on search quality.
 - Data quality for vectors: Techniques for ensuring high-quality input data for embedding generation and strategies for handling noisy or inconsistent data.
- Note:
 - Trade-offs of ANN: Between search speed, memory, and accuracy for different ANN algorithms.
 - Emerging research: New algorithms or optimizations for non-traditional hardware.

Further Exploration

1. Wang et al. (2021). Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data* (pp. 2197–2210).
2. Zhang, Z., Jin, C., Tang, L., Liu, X., & Jin, X. (2023). Fast, approximate vector queries on very large unstructured datasets. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation* (NSDI '23) (pp. 963–980).
3. Gao et al. (2023). Retrieval-augmented generation for large language models: A survey. arXiv:2312.10997.
4. Ma et al. (2023). A comprehensive survey on vector database: Storage and search. arXiv:2310.11703.
5. Taipalus, T. (2024). Vector database management systems: Fundamental concepts, use-cases, and current challenges. *Cognitive Systems Research*, 85, 101216.
6. Main vector DB companies/providers:
 - Pinecone
 - Milvus (Zilliz)
 - Weaviate
 - Qdrant
 - Chroma
7. Major open source resources
 - Meta AI (Facebook AI): Faiss
 - Google: ScaNN
 - Microsoft: DiskANN

This page titled [8.4: Vector DBs & Similarity Search at Scale \(The Big Picture\)](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by Hei-Chi Chan.

INDEX

No index terms found.

GLOSSARY

S

Sample Word 1 | Sample Definition 1

This page titled [Glossary](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by Hei-Chi Chan.

DETAILED LICENSING

Overview

Title: Ancillary Materials for The Shallow and the Deep by M. Biehl

Webpages: 44

Applicable Restrictions: Noncommercial

All licenses found:

- [CC BY-NC-SA 4.0](#): 79.5% (35 pages)
- [Undeclared](#): 18.2% (8 pages)
- [Public Domain](#): 2.3% (1 page)

By Page

- Ancillary Materials for The Shallow and the Deep by M. Biehl - [CC BY-NC-SA 4.0](#)
 - Front Matter - [Undeclared](#)
 - TitlePage - [Undeclared](#)
 - InfoPage - [Public Domain](#)
 - Table of Contents - [Undeclared](#)
 - Acknowledgement - [CC BY-NC-SA 4.0](#)
 - Licensing - [Undeclared](#)
 - 1: From Neurons to Networks - [CC BY-NC-SA 4.0](#)
 - 1.1: Chapter 1 Summary - [CC BY-NC-SA 4.0](#)
 - 1.2: A Brief History of Neural Networks and Deep Learning - [CC BY-NC-SA 4.0](#)
 - 1.3: Hebbian Learning - [CC BY-NC-SA 4.0](#)
 - 1.4: The Hopfield Model - [CC BY-NC-SA 4.0](#)
 - 1.5: Modern Hopfield Networks - [CC BY-NC-SA 4.0](#)
 - 1.6: A Brief Discussion - [CC BY-NC-SA 4.0](#)
 - 2: Learning from Data - [CC BY-NC-SA 4.0](#)
 - 2.1: Chapter 2 Summary - [CC BY-NC-SA 4.0](#)
 - 2.2: Types of Learning - [CC BY-NC-SA 4.0](#)
 - 2.3: A List of Supervised Learning Models - [CC BY-NC-SA 4.0](#)
 - 2.4: Annoy from Spotify - [CC BY-NC-SA 4.0](#)
 - 2.5: Distance Concentration - [CC BY-NC-SA 4.0](#)
 - 2.6: Shannon's Source Coding Theorem - [CC BY-NC-SA 4.0](#)
 - 3: The Perceptron & Beyond Linear Separability - [CC BY-NC-SA 4.0](#)
 - 3.1: Chapters 3 and 4 Summary - [CC BY-NC-SA 4.0](#)
 - 3.2: A Brief Introduction on Rademacher Complexity - [CC BY-NC-SA 4.0](#)
 - 4: Beyond Linear Separability - [Undeclared](#)
 - 5: Feed-forward Networks for Regression and Classification - [CC BY-NC-SA 4.0](#)
 - 5.1: Chapter 5 Summary - [CC BY-NC-SA 4.0](#)
 - 5.2: Universal Approximation Theorem - [CC BY-NC-SA 4.0](#)
 - 5.3: Attention Mechanism/Transformers - [CC BY-NC-SA 4.0](#)
 - 6: Distance-based Classifiers - [CC BY-NC-SA 4.0](#)
 - 6.1: Chapter 6 Summary and a Discussion on the Neural Gas Model - [CC BY-NC-SA 4.0](#)
 - 7: Model Evaluation and Regularization - [CC BY-NC-SA 4.0](#)
 - 7.1: Chapter 7 Summary - [CC BY-NC-SA 4.0](#)
 - 7.2: The Bias-Variance Trade Off - [CC BY-NC-SA 4.0](#)
 - 7.3: Explainable AI (XAI) - [CC BY-NC-SA 4.0](#)
 - 8: Preprocessing and Unsupervised Learning - [CC BY-NC-SA 4.0](#)
 - 8.1: Chapter 8 Summary - [CC BY-NC-SA 4.0](#)
 - 8.2: Dimensionality Reduction Methods - [CC BY-NC-SA 4.0](#)
 - 8.3: Clustering Methods - [CC BY-NC-SA 4.0](#)
 - 8.4: Vector DBs & Similarity Search at Scale (The Big Picture) - [CC BY-NC-SA 4.0](#)
 - Back Matter - [Undeclared](#)
 - Index - [Undeclared](#)
 - Glossary - [CC BY-NC-SA 4.0](#)
 - Detailed Licensing - [Undeclared](#)

This page titled [Ancillary Materials for The Shallow and the Deep by M. Biehl](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by Hei-Chi Chan.