Lecture Note of EE: 541 A Computational Introduction to Deep Learning

Dr. Brandon Franzke

Scribe: Yi FAN

Contents

Part I

Basic theory

Preface

Welcome to EE 541: A computational Introduction to deep learning. We hope everything goes smoothly for you this semester. When I first stepped onto USC's campus and sat in the lecture hall, I felt both excited and uncertain about the future—much like many of you here today. We live in an era of rapid change and an explosion of information, which can leave students—especially those from non-computer science backgrounds—feeling overwhelmed by the sheer volume of study materials. It becomes challenging to know where to begin or what knowledge has truly been mastered.

We hope this material could be your No.1 guidance while studying this course. In composing these course notes, we aimed strike a trade-off between linearity(conciseness) and non-linearity(richness). This balance is one of the most crucial yet interesting principles in the field. You expect the course to present a clear structure: an intuitive knowledge tree and timeline that clearly delineate the main threads of the curriculum, highlight distinctive definitions, and pinpoint the essential operations you must master. This is the rationale behind our effort to "linearize" the course content.

At the same time, we want the course material to be rich enough to enable you to delve deeper into the subject, rather than being confined to a single, linear narrative. To achieve this, we have expanded the main line with various branches, up-to-date areas, thereby constructing a layered, nonlinear, and complex knowledge system.

This handout is organized into two main themes: Theory Backgrounds and Applications. The primary purpose of compiling this lecture handout is to help you grasp the core content of this course more conveniently and efficiently. The first theme answers the questions 'What' and 'Why', providing an extended and readable version of the PPTs, helping alleviate confusion or feeling of loss during seemingly overwhelming materials. The second theme delves deeper into the question of "How," offering examples, hints, and guidance for homework or other meaningful tasks.

Each theme consists of 8-9 chapters, aligned with the structure of our slides. Concision and readability come first! Our goal is to help students who read these materials gain clarity of what they are learning, why it matters, and how to apply it effectively.

However, given my personal level of expertise, this lecture note may unavoidably have some shortcomings. I will ensure to update it weekly and actively seek your feedback for improvement. The copyright will be jointly held by Dr. Brandon Franzke and Yi FAN. Any unauthorized reproduction is strictly prohibited.

1 Introduction Machine Learning

1.1 What is Machine Learning

Learning is any process by which a system improves performance from experience

- Herbert Simon

At the beginning of our journey, let us consider a simple yet meaningful question: What is **learning**? How do we **humans** learn?

Generally speaking, human thinking and learning primarily rely on the brain—a highly complex biological organ. The brain processes, stores, and transmits information through neurons and their connections (synapses), thereby enabling functions such as cognition, decision-making, and memory. The process of human learning can be viewed as involving changes on two levels:

- Local Synaptic Changes: Synaptic connections between neurons are strengthened through repeated activation and weakened due to disuse. (analogous to the way hidden layers adjust through weighted linear combinations and nonlinear activations)
- Formation of Functional Networks: Learning is not limited solely to local synaptic modifications. Depending on task requirements and accumulated experience, the brain integrates groups of related neurons to form functional neural networks. This large-scale organization, endows humans with a robust capacity for learning and adaptation.(—comparable to the overall structure of a neural network in deep learning.)

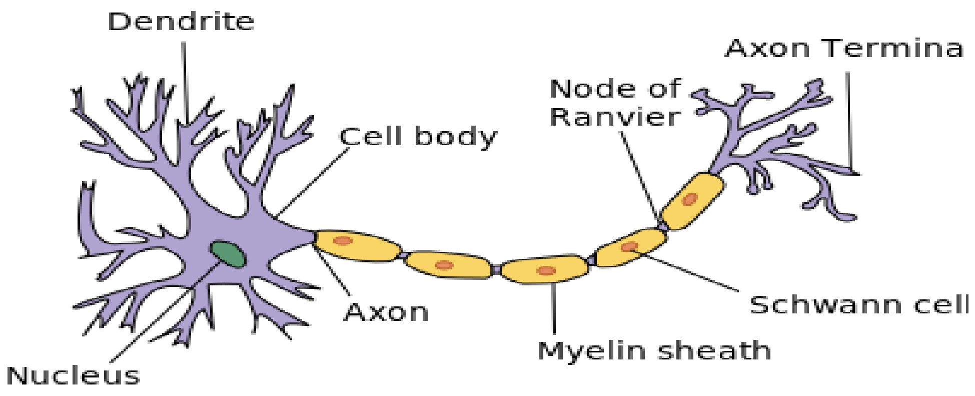


Figure 1: Neurons

Let's return to our topic and reflect on this question: What is Machine learning? We make a brief definition here: **Machine Learning** is the study of **Algorithms** that improve **Performance** on a given **Task** through **Experience**.

However, machines don't have brains—how can they learn from past experience? More generally, what constitutes a machine's "experience," and how should we define it? In the following sections, we will address these questions.

1.2 Data Representation and Hypothesis Class

1.2.1 Our math framework

Before we begin, it is crucial to keep in mind one important matter: we need to find a unified and consistent framework or language to represent our data. Let us dive into an interesting example. As shown in Figure 2, We labeled three pictures with the labels

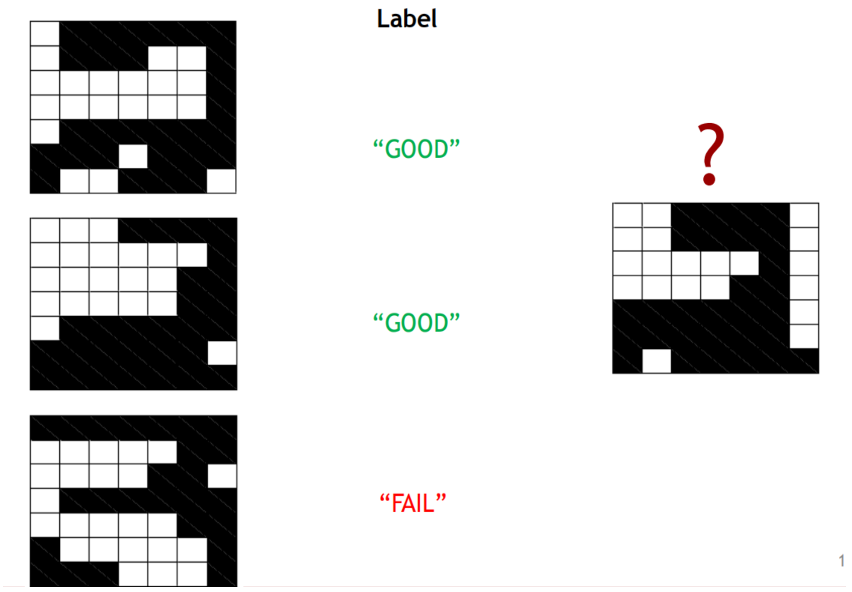
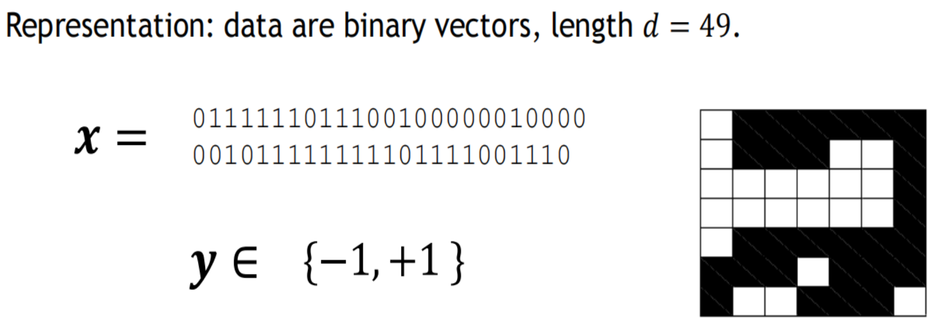


Figure 2: The data domain


Representation: data are binary vectors, length $d = 49$.

$$\boldsymbol{x} = \begin{matrix}0111111011100100000010000\\001011111111101111001110\end{matrix}$$

$$\boldsymbol{y} \in \{-1, +1\}$$

Figure 3: The binary representation

'good' or 'fail'. Your job is to decide what class does the following picture belong to. Isn't it somewhat unreasonable that you feel completely at a loss? Perhaps we can take a different perspective and explore this image in a binary manner, like a computer.

First, let's process this image in a top-left-to-bottom-right order, assigning white as 0 and black as 1. This 7x7 image will then be flattened into a 49-bit binary sequence. Our input now becomes x which has 49 dimensions $x_1, x_2, ..., x_{49}$. This seems easier and more obvious. Then, let's mark good as +1 and fail as −1. Likely, out outcome has a new form $\hat{y}$ now, which has two distinct values +1 or −1. Our job can be rewritten as finding a **linear** representation of y as a function of x. The math formula is shown below.

$$\hat{y} = \text{sign}(\boldsymbol{\theta} \cdot \boldsymbol{x}) = \text{sign}\left(\theta_1 x_1 + \cdots + \theta_{49} x_{49}\right) \tag{1}$$

Here, $\theta_1, \theta_2, ..., \theta_{49}$ are all trainable parameters and reflect the weights(relative importance) of one specific input. Usually we use two types of learning methods. The closed-form, also known as explicit one, starts from calculating θ using many samples and then choose exact θ to minimize a cost function. The sequential (implicit) method includes guessing θ_1, evaluating model and updating/adjusting parameters if wrong $\theta_{i+1} = \theta_i + y x_i$.

This binary classification task can be simply accomplished by machine learning methodology, hypothesizing mapping data to label using linear classifier and will be introduced in the following chapter.

1.2.2 Why should we represent our data using linear algebra?

That's an interesting question. On the one hand, while human sensory perception is diverse and its inherent nonlinearity enriches our experiences, it also inevitably introduces additional complexity while dealing with such problems. In contrast, linear algebra provides a highly abstract means of expression—a sort of "filter" that extracts the most essential **features** from high-dimensional data (in other words, a linear representation of multimodal data). This approach **simplifies** the complex phenomena of the real world into a mathematical model. Through such a model, we can more clearly understand the structure of problems by reducing complexity to its core characteristics.

On the other hand, machines typically acquire information in a relatively **uniform** manner, primarily relying on digital representation. This abstraction not only helps humans organize their thoughts but also provides machines with a structured, standardized input. Adapting linear algebra ensures that different types of data are processed and integrated within a coherent mathematical framework.

That sounds promising—at least we now share a common language with machines. So, how do machines learn from these data?

1.3 Multilayer Perceptron Networks

1.3.1 Single artificial neuron

Let's return to our initial question: How do machines learn? Inspired by biological neurons, people have developed artificial neurons for machines. In biology, neurons transmit information by receiving signals, accumulating them, and firing an action potential. In artificial neurons, human mimic this process through the sequence: input – weighted sum – activation – output. Figure 4 shows the typical diagram of one artificial neuron. Mathematically, neurons can be expressed as:

$$\boldsymbol{a}^{(l)} = \underline{h}\left(\boldsymbol{W}_l \boldsymbol{a}^{l-1} + \boldsymbol{b}_l\right) \tag{2}$$

Where $a^{(l-1)}$ are inputs of layer l, W_l and b_l are weights and bias carried by neurons in layer l. $\underline{h}$ is the activation function, usually Sigmoid, tanh, Relu, etc.

• **Weight** reflects the relative importance of each input in making a decision. A larger weight indicates that the corresponding input has a greater influence on the output.

• **Bias** can be viewed as the "threshold" or "baseline" of a neuron, similar to the intercept in a linear model. In biological neurons, a neuron is only activated when the accumulated signal exceeds a certain threshold. Similarly, in neural networks, the bias determines when a neuron is activated.

From the perspective of systematic error, bias can also be understood as the error introduced by approximating a complex real-world problem with a simplified model. In other words, a high bias indicates that the model is oversimplified and may fail to capture important patterns in the data, leading to underfitting.

• **Activiation Funtion** is a non-linear function applied to the weighted sum of a neuron's inputs. It introduces non-linearity into the network, enabling it to capture complex patterns and non-linear relationships. The activation function determines whether

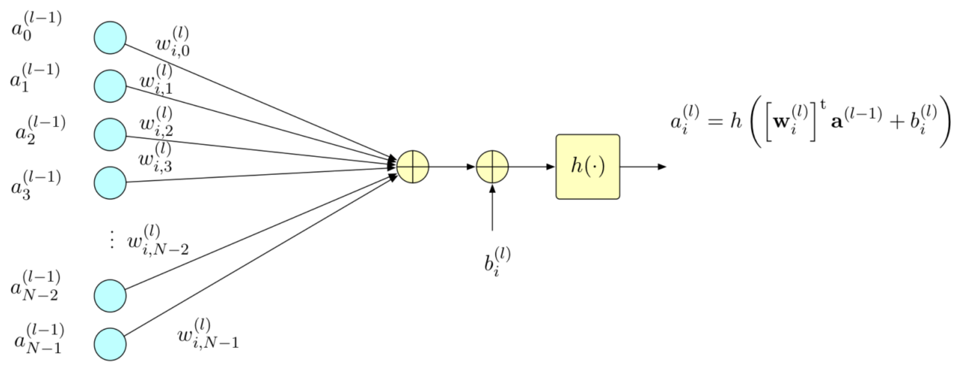


Figure 4: Single Artificial Neuron, hidden layer of MLP

a neuron is activated and what its output will be, playing a crucial role in modulating the signal within the network.

It is important to note that although artificial neurons are indeed inspired by their biological counterparts, they are **essentially different**. Biological neurons exhibit complex structures and dynamic behaviors, including nonlinear responses and intricate neurochemical interactions. In contrast, artificial neurons are highly abstracted and simplified mathematical models that focus on simulating the input-output relationship using basic functions. This simplification allows us to train and deploy neural networks efficiently.

With this understanding, we now turn our attention to the Multilayer Perceptron (MLP) Network, where artificial neurons are organized into layers. Next, we will explore the two fundamental processes that drive learning in an MLP: Forward Propagation (Inference) and Back Propagation (Learning).

1.3.2 From single neuron to neural networks

MLP (Multilayer Perceptron) Network is a type of neural network composed of multiple layers of neurons. In an MLP, individual neurons are organized into layers, forming a structured network. Each neuron receives inputs—either from the original data (in the input layer) or from the outputs of neurons in the previous layer—and computes a weighted sum of these inputs, adds a bias term, and then applies an activation function to produce its output. This output then serves as input for neurons in the next layer.

Typically, An MLP consists of an input layer, one or more hidden layers, and an output layer. The schematic diagram of MLP is shown in Figure 5.

- **Forward Propagation(Inference)**

In an MLP, forward propagation refers to the process by which input data is passed from the input layer, through each hidden layer, and finally reaches the output layer. Each layer typically performs two key operations:

(1)Linear Transformation: For a given layer, the network computes a weighted sum $\boldsymbol{a}^{(l)} = \boldsymbol{W}_l \boldsymbol{a}^{l-1} + \boldsymbol{b}_l$.

(2)Non-linear Activation: $\boldsymbol{a}^{(l)}$ is then passed through a nonlinear activation function (such as ReLU, Sigmoid, or Tanh) to produce the layer's output. $\boldsymbol{a}^{(l)} = \underline{h}\left(\boldsymbol{W}_l \boldsymbol{a}^{l-1} + \boldsymbol{b}_l\right)$.

This entire process constitutes the model's "inference"—it processes input data using the current weights and biases to ultimately produce a prediction or decision.

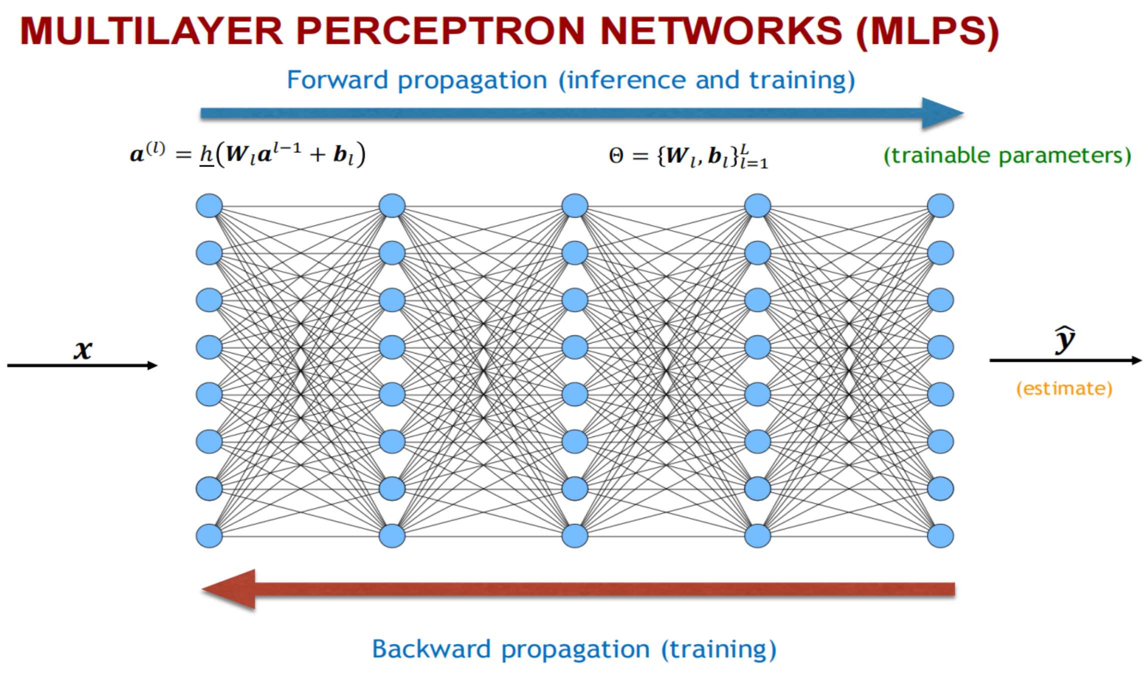

MULTILAYER PERCEPTRON NETWORKS (MLPS)
Forward propagation (inference and training)
$a^{(l)} = \underline{h}(W_l a^{l-1} + b_l)$
$\Theta = \{W_l, b_l\}_{l=1}^{L}$
(trainable parameters)
x
$\hat{y}$
(estimate)
Backward propagation (training)

Figure 5: MLP Networks

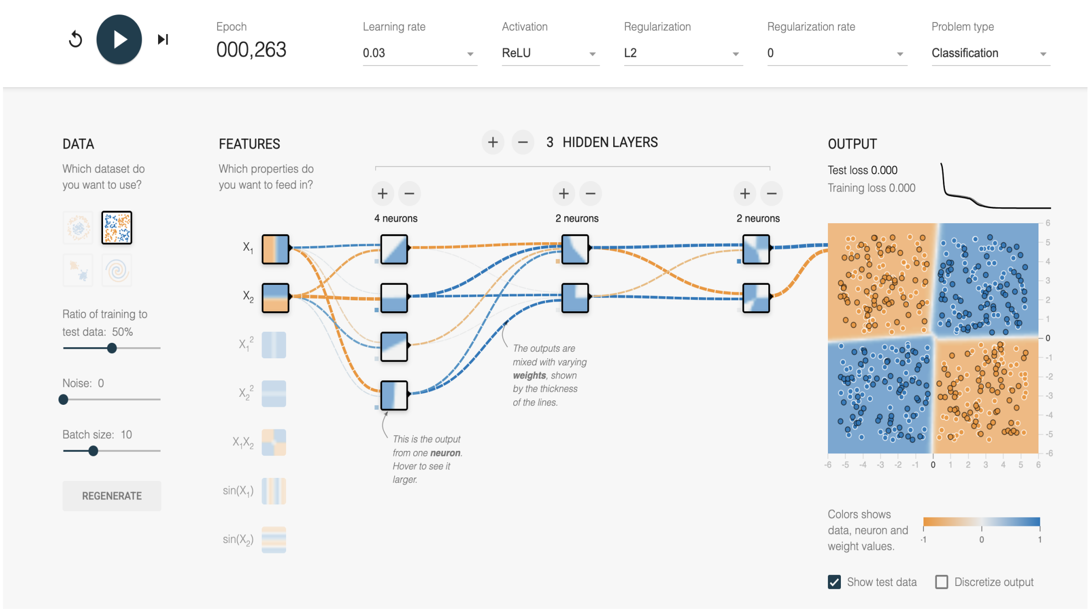
It is worth noting that, activation function is an integral part of MLP. In the former part we strengthen that adding such function helps model capture the non-linear and intricate patterns, that is from the perspective of its benefits. Here, from a **mathematical perspective**, if we drop the activation function, then $\boldsymbol{a}^{(l)}$ is linear on $\boldsymbol{a}^{(l-1)}$, linear on $\boldsymbol{a}^{(l-2)}$... That means that our final output can be simply represented using a whole linear function of our inputs. Our complex neural structure then becomes meaningless, why bother?

- **Backward Propagation(Learning)** The mechanism that empowers machines to learn is back propagation. It is the process by which the network computes the gradient of the loss function with respect to each parameter (weights and biases) and propagates the error backward from the output layer to the input layer. This enables the model to adjust its parameters to reduce the prediction error. Generally, back propagation involves three key steps:

(1)Error Computation: Recall that we define that Machine Learning is the study of Algorithms that improve Performance on a given Task through Experience. Usually we evaluate the model performance by accuracy score or model loss. Accuracy score measures the proportion of correct predictions made by the model, whereas the loss function quantifies the error between the predicted outputs and the actual values. After forward propagation, the loss function is computed.

(2)Gradient Propagation: Using the chain rule, the network calculates the gradients of the loss function with respect to each parameter, propagating these gradients layer by layer from the output back to the input.

(3)Parameter Update: The network updates the parameters (weights and biases) using the computed gradients, typically through gradient descent or its variants (such as Adam), thereby gradually reducing the loss and ”learning” the appropriate parameter values.


Epoch 000,263

Learning rate 0.03

Activation ReLU

Regularization L2

Regularization rate 0

Problem type Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

FEATURES

Which properties do you want to feed in?

3 HIDDEN LAYERS

4 neurons

2 neurons

2 neurons

The outputs are mixed with varying **weights**, shown by the thickness of the lines.

This is the output from one **neuron**. Hover to see it larger.

OUTPUT

Test loss 0.000

Training loss 0.000

Colors shows data, neuron and weight values.

[✓] Show test data

[ ] Discretize output

Figure 6: Tensorflow Playground

In summary, the MLP continuously cycles through forward propagation (inference) and backward propagation (parameter adjustment), progressively lowering the prediction error. This reduction in error corresponds to the process of improving performance on a given task, which is essentially how machines learn.

1.3.3 Visualize your MLP: Tensorflow Playground

`https://playground.tensorflow.org/`

TensorFlow Playground is an interactive web-based tool that allows users to visualize and experiment with neural networks in real time. It provides a hands-on interface where you can adjust parameters like the number of layers, neurons, activation functions, and learning rates, and immediately observe how these changes affect the network's performance on tasks such as classification. This platform is particularly useful as it offers a clear, concrete demonstration of how deep learning models operate and adapt to data.

1.4 Three types of Machine Learning

1.4.1 Supervised learning

In supervised learning, models are trained using labeled data — each training example consists of an input paired with the correct output$\{(\mathbf{x}_n, \mathbf{y}_n)\}_{n=0}^{N-1}$. The objective is to learn a mapping from inputs to outputs, allowing the model to predict labels for unseen data. This approach relies on minimizing a loss function that quantifies the error between the predicted and true values, and it forms the core of many practical applications such as classification and regression. Generally speaking, we do quite well on this area and our course will focus mostly on this part.

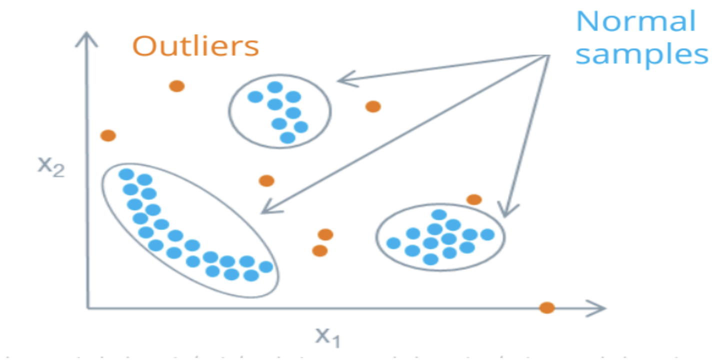


Figure 7: Outlier detection(Unsupervised Learning)

1.4.2 Unsupervised Learning

Unsupervised learning deals with data that has no explicit labels. The goal is to uncover hidden patterns or structures within the dataset. For example, outlier detection (or anomaly detection) is generally considered a type of unsupervised learning. In this task, the goal is to identify data points that significantly deviate from the majority of the dataset, often without prior labels indicating what constitutes an anomaly. These methods analyze the underlying structure or distribution of the data to pinpoint instances that do not conform to expected patterns, which can be critical for applications such as fraud detection, system monitoring, and quality control. This area is what USC-FORTIS Lab(led by Yue Zhao)currently exploring. Feel free to discuss with us!

1.4.3 Reinforcement Learning

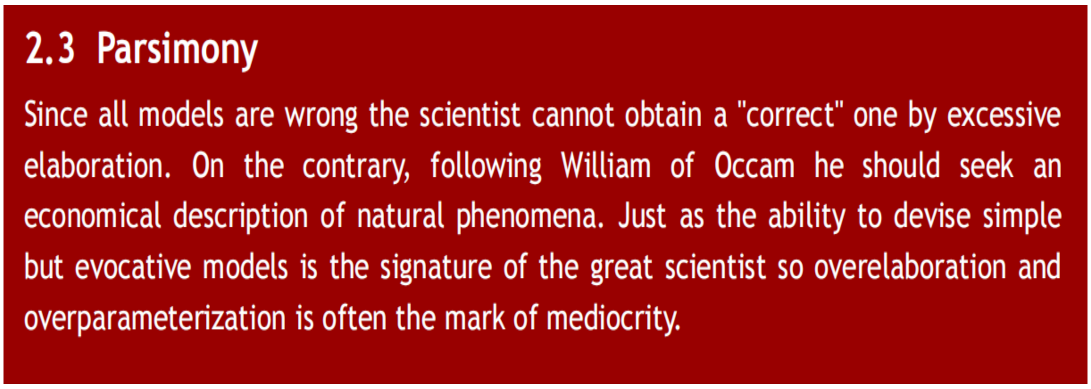
Reinforcement learning involves an agent that learns to make decisions by interacting with an environment. The agent receives feedback in the form of rewards or penalties, and its objective is to develop a strategy that maximizes the cumulative reward over time. This paradigm is particularly effective in scenarios where the learning process is driven by trial and error, such as in robotics and game playing. This area has yet to be further explored.

1.5 All models are wrong, but some are useful

- George E. P. Box

1.5.1 The Nature of Models as Simplifications

Machine learning models—be it linear regression, neural networks, or deep learning models—are simplifications or approximations of the real world. These models are built to capture patterns from data but cannot fully represent reality with all its complexity. For example: Data may contain noise or biases. A model's assumptions (e.g., linear relationships or specific distributions) often do not perfectly align with reality. Despite these imperfections, models remain effective tools for solving specific problems and making predictions within their intended scope.


2.3 Parsimony

Since all models are wrong the scientist cannot obtain a "correct" one by excessive elaboration. On the contrary, following William of Occam he should seek an economical description of natural phenomena. Just as the ability to devise simple but evocative models is the signature of the great scientist so overelaboration and overparameterization is often the mark of mediocrity.

Figure 8: Accuracy- generalization trade-off

1.5.2 The Usefulness of Models Lies in Their Context

The success of machine learning models depends heavily on the context of their application. For example: A simple linear regression model may fail to capture complex nonlinear patterns but could still be sufficient for certain straightforward tasks due to its interpretability. A deep learning model may provide high accuracy but could perform poorly when there is insufficient data or computational resources. Only when applied to the right scenarios, 'some models are useful'.

1.5.3 Trade-off Between accuracy and Generalization

Generalization refers to a model's ability to perform well on unseen data. This is the ultimate goal of most machine learning tasks since models are deployed in environments different from their training sets. **Accuracy** is the proportion of correct predictions made by a model on a given dataset, typically the training or test data, and is often used as a primary metric for evaluating how well a model fits known data.

The conflict between accuracy and generalization arises because achieving perfect accuracy on the known training data often harms the model's flexibility to perform well on unseen data. This is directly tied to the bias-variance trade-off. As George E. P. Box put forwards: 'we should seek an economical description of natural phenomena. Just as the ability to devise simple but evocative models is the signature of the great scientist so overelaboration and overparameterization is often the mark of mediocrity.' Generally, A 'useful' model strikes a balance: It simplifies reality just enough to generalize well. It avoids overfitting by not chasing extreme accuracy on training data, thereby maintaining robustness across datasets.

In conclusion, a model is always a simplification or abstraction of reality, which means it can never capture every detail perfectly. However, despite their imperfections, models can still provide valuable insights and guidance in understanding systems, making decisions, or predicting outcomes—provided they are applied within their intended scope.

2 Estimation Task

2.1 Linear Minimum Mean Square Error (LMMSE) Estimation

In machine learning, signal processing, and control theory, we often need to make predictions based on real-world observations. This process is referred to as **inference** or **estimation**. Given a set of observations, estimation seeks to determine unknown quantities that are most consistent with the observed data.

The **Linear Minimum Mean Square Error (LMMSE) Estimator** is a specific method of estimation where the estimate is given as a **linear combination of observations**. That is, if we denote the observations as a random vector $\mathbf{X}(\mathbf{t}) = \mathbf{x}$, then the LMMSE estimate $\hat{\mathbf{y}}$ of some unknown variable $\mathbf{y}$ takes the form below, where $\mathbf{W}$ is a weight matrix and $\mathbf{b}$ is a bias vector:

$$\hat{\mathbf{y}} = \mathbf{W}^{\mathbf{T}}\mathbf{x} + \mathbf{b} \tag{3}$$

The key idea behind LMMSE is to minimize the **mean square error (MSE)** between the estimate and the true value while restricting the form of the estimator to be linear. This ensures a balance between accuracy and tractability. In this scenario, the objective (cost function) to be optimized is MSE $= \mathbf{E}\left[(y(t) - \mathbf{W}^T\mathbf{x} - b)^2\right]$

In many scenarios, enforcing a linear estimation structure provides significant advantages:

- **Computational Simplicity**: Linear estimators are easier to compute compared to nonlinear alternatives, making them attractive for real-time applications.
- **Closed-Form Solutions**: Under Gaussian assumptions, LMMSE has an explicit analytical solution, reducing the need for iterative optimization.
- **Widely Used in Applications**: LMMSE is widely applied in signal processing (e.g., Wiener filtering), machine learning (e.g., linear regression).

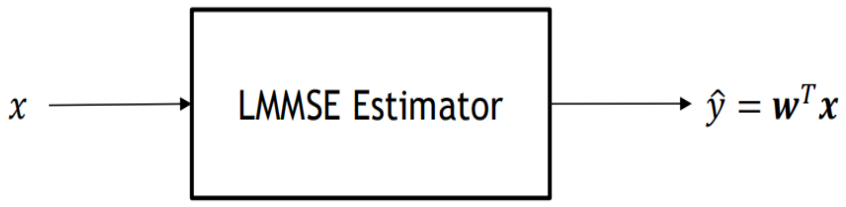


Figure 9: LMMSE

In the following sections, we will delve deeper into the mathematical formulation, derivation, and applications of LMMSE estimation.

2.2 Random variable and random vector

2.2.1 Random variable

In the former section we mentioned that we denote the observations as a random vector $\mathbf{X}(\mathbf{t}) = \mathbf{x}$, here we discuss in detail what are random vectors before we go deeper.

A **random variable** is a function that maps outcomes from an underlying probability space to numerical values. Formally, given a probability space $(\Omega, \mathcal{F}, P)$, a random variable is a measurable function:

$$X : \Omega \to \mathbb{R} \tag{4}$$

where:

- Ω is the **sample space**, containing all possible outcomes.
- $\mathcal{F}$ is a **σ-algebra**, which defines measurable subsets of Ω.
- P is a **probability measure**, assigning probabilities to events in $\mathcal{F}$.
- $X(\omega)$ assigns a real number to each outcome $\omega \in \Omega$.

A crucial property of a random variable is that **its pullbacks are always measurable**. This means that for any real number a, the event:

$$\omega \in \Omega \mid X(\omega) \leq a \in \mathcal{F} \tag{5}$$

This ensures that probabilities of such events can be assigned using P, allowing structured analysis of uncertainty.

2.2.2 Random vector

A **random vector** generalizes the concept of a random variable to multiple dimensions. It is a function:

$$\mathbf{X}(t) : \Omega \to \mathbf{R}^N \tag{6}$$

where:

$$\mathbf{X}(t) = \left[X_1(t) \ X_2(t) \ \vdots \ X_N(t) \right] \tag{7}$$

Each $X_i(t)$ is a random variable, meaning it assigns a real value to each sample $\omega \in \Omega$. A **random vector** can be understood as a collection of uncertain numerical quantities that are observed together. Some examples include:

- **Sensor Networks:** A weather station measures temperature, humidity, and pressure simultaneously.
- **Stock Market:** The prices of different stocks fluctuate randomly but are often correlated.
- **Signal Processing:** A multi-channel signal (e.g., different frequency components) is modeled as a random vector.

To fully describe a random vector, we examine its key statistical characteristics:

2.2.3 Second Moment Description

- **Mean Vector**
  The **mean vector** represents the expected value of each component:

$$\mathbf{m}_X = \mathbb{E}[\mathbf{X}(t)] = \Big[\mathbb{E}[X_1(t)]\ \mathbb{E}[X_2(t)] \vdots \mathbb{E}[X_N(t)]\Big] \tag{8}$$

  It describes the **central tendency** of the random vector.

- **Correlation Matrix**
  The **correlation matrix** quantifies the pairwise statistical dependencies between elements, capturing their linear relationships through second-order statistics:

$$\mathbf{R}_X = \mathbb{E}[\mathbf{X}(t)\mathbf{X}^T(t)] \tag{9}$$

  where each element is given by:

$$[R_X]i, j = \mathbb{E}[X_i(t)X_j(t)] \tag{10}$$

  This captures how different components interact.

- **Covariance Matrix**
  The **covariance matrix** is a symmetric matrix that quantifies the joint variability of multiple random variables:

$$\mathbf{K}_X = \mathbb{E}[(\mathbf{X}(t) - \mathbf{m}_X)(\mathbf{X}(t) - \mathbf{m}_X)^T] \tag{11}$$

  which simplifies to:

$$\mathbf{K}_X = \mathbf{R}_X - \mathbf{m}_X\mathbf{m}_X^T \tag{12}$$

  The diagonal elements represent the variance of each component, while off-diagonal elements measure **covariance**, indicating how changes in one variable are associated with changes in another.

- **Element-wise Covariance**
  For two components $X_i(t)$ and $X_j(t)$, their covariance is:

$$[K_X]_{i,j} = \mathrm{Cov}[X_i(t), X_j(t)] \tag{13}$$

  which shows how two variables fluctuate together.

2.3 KL Expansion

The Karhunen-Lo‘eve (KL) expansion is a powerful decomposition technique used in signal processing, statistics, and machine learning. It is closely related to Principal Component Analysis (PCA) and is widely applied for dimensionality reduction, decorrelation, and optimal representation of stochastic processes.

Key advantages of KL expansion include:

- **Always possible**: The covariance matrix $\mathbf{K}_X$ is **symmetric and positive semi-definite (PSD)**, ensuring that we can always find an **orthonormal set of eigenvectors**.

- **Principal Components**: The eigenvectors define **principal directions** in the data, where the variance (or energy) is maximized. The eigenvalues λ_k represent the variance (energy) along each principal direction.
- **Dimensionality Reduction**: By discarding low-energy components, we can efficiently reduce the dimensionality of the data.

2.3.1 Mathematical Formulation

KL expansion decomposes a random vector $\mathbf{X}(t)$ as:

$$\mathbf{X}(t) = \sum_{k=1}^{N} X_k(t)\mathbf{e}_k \tag{14}$$

where:

- $\mathbf{e}_k$ are the **orthonormal eigenvectors** of the covariance matrix $\mathbf{K}_X$.
- $X_k(t) = \mathbf{e}_k^T\mathbf{X}(t)$ are the transformed coefficients (principal components).
- The components $X_k(t)$ are **uncorrelated**.
- The covariance matrix can be reconstructed as:

  $$\mathbf{K}_X = \sum k = 1^N \lambda_k \mathbf{e}_k \mathbf{e}_k^T = \mathbf{E}\boldsymbol{\Lambda}\mathbf{E}^T \tag{15}$$

  where $\boldsymbol{\Lambda}$ is the diagonal matrix of eigenvalues and $\mathbf{E}$ is the matrix of eigenvectors.
- The total energy is given by:

  $$\mathbb{E}[|\mathbf{X}(t)|^2] = \mathrm{tr}(\mathbf{K}X) = \sum k = 1^N \lambda_k \tag{16}$$

2.3.2 Example: 3x3 KL Expansion

Consider a 3-dimensional random vector $\mathbf{X}$ with the covariance matrix:

$$\mathbf{K}_X = \begin{bmatrix} 6 & -4 & 0 \\ -4 & 6 & 0 \\ 0 & 0 & 3 \end{bmatrix} \tag{17}$$

Step 1: Compute Eigenvalues and Eigenvectors Solving $\det(\mathbf{K}_X - \lambda\mathbf{I}) = 0$, we obtain:

$$\lambda_1 = 10, \quad \lambda_2 = 3, \quad \lambda_3 = 2 \tag{18}$$

with corresponding normalized eigenvectors:

$$\mathbf{e}_1 = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & -1 & 0\end{bmatrix}^{\mathrm{T}}, \quad \mathbf{e}_2 = \begin{bmatrix}0 & 0 & 1\end{bmatrix}^{\mathrm{T}}, \quad \mathbf{e}_3 = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1 & 0\end{bmatrix}^{\mathrm{T}} \tag{19}$$

Step 2: Transform to Eigen-coordinates The transformed coordinates are:

$$X_k = \mathbf{e}_k^T\mathbf{X} \tag{20}$$

This ensures that the transformed variables are uncorrelated.

Step 3: Dimensionality Reduction Since the eigenvalues represent the **variance in each direction**, we can **approximate the data using only the first two components** (largest eigenvalues) and discard the lowest-energy component:

$$\mathbf{X}(t) \approx X_1\mathbf{e}_1 + X_2\mathbf{e}_2 \tag{21}$$

This reduces the original 3D data to a **2D representation** while preserving most of the variance.

2.4 Math formula for LMMSE

2.4.1 Prerequisite: The knowledge of second-order statistics

The Linear Minimum Mean Square Error (LMMSE) estimator is a widely used estimation technique that assumes knowledge of the second-order statistics of the joint distribution of the observed and the target variables. This includes:

- Mean: Expected values of the random variables involved.
- Variance: Measure of variability within each variable.
- Covariance: Measure of joint variability between different variables.

These assumptions allow for the derivation of an optimal linear estimator that minimizes the mean squared error.

2.4.2 Two common cases of LMMSE

Table 1: Comparison of Linear MMSE and Affine MMSE

Linear MMSE	Affine MMSE
$\min\limits_{f(x)=Fx} \mathbb{E}\left\{\|y(t) - f(x(t))\|^2\right\}$	$\min\limits_{f(x)=Fx+b} \mathbb{E}\left\{\|y(t) - f(x(t))\|^2\right\}$
$\min\limits_{F} \mathbb{E}\left\{\|y(t) - Fx(t)\|^2\right\}$	$\min\limits_{F,b} \mathbb{E}\left\{\|y(t) - [Fx(t) + b]\|^2\right\}$
$\mathbf{F}_{\text{LMMSE}} = \mathbf{R}_{YX}\mathbf{R}_X^{-1}$	$\mathbf{F}_{\text{AMMSE}} = \mathbf{K}_{YX}\mathbf{K}_X^{-1}$
$\mathbf{b}_{\text{LMMSE}} = \mathbf{0}$	$\mathbf{b}_{\text{AMMSE}} = \mathbf{m}_y - \mathbf{F}_{\text{AMMSE}}\mathbf{m}_X$
$\hat{\mathbf{y}} = \mathbf{R}_{YX}\mathbf{R}_X^{-1}\mathbf{x}$	$\hat{\mathbf{y}} = \mathbf{K}_{YX}\mathbf{K}_X^{-1}(\mathbf{x} - \mathbf{m}_X) + \mathbf{m}_y$
$\text{LMMSE}\varepsilon = \text{Tr}\left(\mathbf{R}_Y - \mathbf{R}_{YX}\mathbf{R}_X^{-1}\mathbf{R}_{XY}\right)$	$\text{AMMSE}\varepsilon = \text{Tr}\left(\mathbf{K}_Y - \mathbf{K}_{YX}\mathbf{K}_X^{-1}\mathbf{K}_{XY}\right)$

There are two primary formulations of LMMSE as shown below. They are basically the same, the only difference lies in that the latter takes the **bias** into consideration to compensate for the potential bias that may arise when the mean is non-zero:

1. Linear Estimation: A special case where the bias term is zero:

$$\hat{X} = \mathbf{W}Y \tag{22}$$

2. Affine Estimation: The estimator includes a bias term and is given by:

$$\hat{X} = \mathbf{W}Y + \mathbf{b} \tag{23}$$

where $\mathbf{W}$ is the weight matrix and $\mathbf{b}$ is a bias term.

2.5 Gradient descent

If we have an objective function $f(w)$, we aim to find its minimum value and the optimal parameter w^*, in this case, suppose $f(w)$ represents the Mean Squared Error (MSE):

$$w^* = \arg\min_{w} f(w) \tag{24}$$

$$f(w) = \mathrm{MSE} = \mathbb{E}\left[(y(t) - \omega^T x - b)^2\right] \tag{25}$$

In the former part we use LMMSE to figure w out **analytically**. Here, instead of obtaining optimal w^* directly, we can also find it **iteratively** via

1. Initialize w_0
2. Update $w_{n+1} = w_n - \eta \nabla f(w_n)$

More importantly, if $f(w)$ is a **convex** function, it means that:

1. **The line segment between any two points lies above the function:**
$$f(\lambda w_1 + (1-\lambda)w_2) \leq \lambda f(w_1) + (1-\lambda) f(w_2), \quad \forall \lambda \in [0,1]$$
2. **A local optimum is also a global optimum:** A convex function has only one minimum value, so gradient descent will not get stuck in a local optimum.

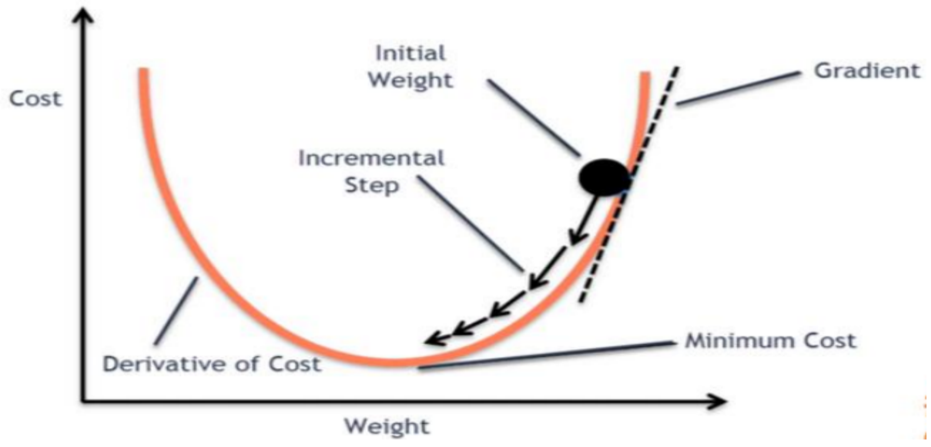


Figure 10: gradient descent

Let's try gradient descent step by step. Generally, our task is to estimate optimal $\hat{y} = \omega^T x$ which minimize the MSE between $y(t)$ and $\hat{y}$. First, we define the loss term as:

$$E(w) = \mathbb{E}\left\{\left[y(t) - w^T x(t)\right]^2\right\} \tag{26}$$

Taking the derivative, we shall get:

$$\nabla_w E = 2\mathbb{E}\left\{w x(t) x^T(t) - y(t) x(t)\right\} \tag{27}$$

$$= 2\left(w^T R_x - r_{xy}\right) \tag{28}$$

Steepest descent using ensemble average gradient(η is the so-called learning rate):

$$\widehat{w}_{n+1} = \widehat{w}_n - \left(\frac{\eta}{2}\right) \nabla_w E \tag{29}$$

$$= \widehat{w}_n + \eta\left(r_{xy} - R_x \widehat{w}_n\right) \tag{30}$$

2.5.1 LMS algorithm - single point gradient descent

Well, the equation given above still requires the second-moment statistics($r_{xy}, R_x\hat{\omega}_n$), just the same condition as LMMSE. What if we only have lots of samples and limited statistical description? The Least Mean Squares (LMS) algorithm is designed for this scenario.

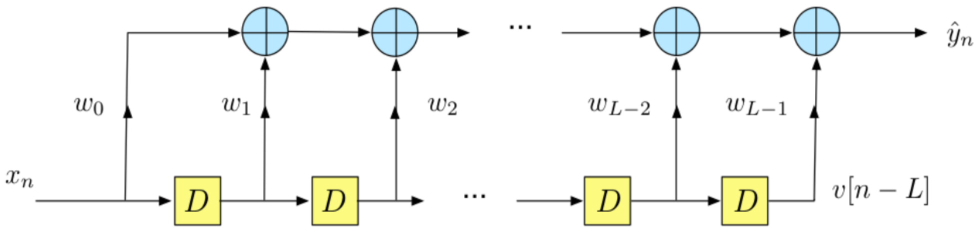


Figure 11: LMS algorithm

The core idea is to perform stochastic gradient descent, using the **current sample** as an approximation instead of the expected value. We suppose that n is large enough so $E[X(t)] = x_n$, The following equation shows this trick:

$$\begin{aligned}
-\frac{1}{2}\nabla_w E &= r_{xy} - R_x w \\
&= \mathbb{E}\left\{y(t)X(t) - X(t)X^T(t)w\right\} \\
&\approx y_n x_n - x_n x_n^T w \\
&= \left(y_n - x_n^T \widehat{w}_n\right) x_n
\end{aligned} \tag{31}$$

Here, we use historical data for one point instead of second moment statistics to estimate the parameter $\hat{y}$. where

$$v_n = x_{n-(L-1)} = \begin{bmatrix} x_n \\ x_{n-1} \\ \vdots \\ x_{n-L+1} \end{bmatrix} \tag{32}$$

Finally, we can update ω_{n+1} simply using the bias term $y_n - \widehat{y}_n$ and one-point historical information v_n:

$$\widehat{w}_{n+1} = \widehat{w}_n + \eta\left(y_n - w_n^T v_n\right) v_n \tag{33}$$

$$= \widehat{w}_n + \eta\left(y_n - \widehat{y}_n\right) v_n \tag{34}$$

$$= \widehat{w}_n + \eta e_n v_n \tag{35}$$

We call this LMS algorithm - Single Point Stochastic Gradient Descent. The LMS algorithm is particularly useful in situations where:

- The second-order statistics (mean, variance, covariance) are unknown: Unlike LMMSE, which requires these statistics, LMS operates without prior knowledge of the full dataset.

- Streaming data is available: The data arrives sequentially rather than being available all at once.LMS can adjust weights dynamically as new data arrives.
- Computational efficiency is needed: LMS is much simpler to implement compared to batch processing methods such as LMMSE, making it suitable for real-time applications.

3 Regression

In the previous chapter, we focused on Estimation Theory under the assumption of **complete knowledge** of the underlying distributions. For instance, when estimating a random variable $\mathbf{y}$ from another variable $\mathbf{x}$, the LMMSE estimator takes the form:

$$\hat{\mathbf{y}} = \mathbf{K}_{YX}\mathbf{K}_X^{-1}(\mathbf{x} - \mathbf{m}_X) + \mathbf{m}_y \tag{36}$$

This expression relies entirely on known statistical quantities: $\mathbb{E}[\mathbf{X}] = \mathbf{m}_X$, $\mathbb{E}[\mathbf{Y}] = \mathbf{m}_Y$, and $\mathrm{Cov}(\mathbf{X}, \mathbf{Y}) = \mathbf{K}_{YX}$. In other words:

- We assume full knowledge of the distribution of the random variables;
- The estimator is derived from a theoretical perspective, aiming to be *optimal* in the sense of minimizing the MSE under linearity.

However, in practice, such statistical knowledge is rarely available. What we do have is data. This brings us to Regression, where solutions are **data-driven**. Estimators such as Ordinary Least Squares (OLS) are derived by replacing expectations with **empirical averages**, effectively using Monte Carlo approximation.

3.1 Empirical expectation and Monte Carlo

Given a dataset:

$$\mathcal{D} = \{(\mathbf{x}_n, \mathbf{y}_n)\}_{n=1}^N, \tag{37}$$

where each $\mathbf{x}_n$ is an observed regressor and each $\mathbf{y}_n$ is a target output.

We aim to find a parameterized mapping $\mathbf{g}(\mathbf{x}; \mathbf{\Theta})$ that minimizes the prediction error.

$$\min_{\mathbf{\Theta}} \langle C(\mathbf{y}, \mathbf{g}(\mathbf{x}; \mathbf{\Theta})) \rangle_{\mathcal{D}}, \tag{38}$$

where $C(\cdot, \cdot)$ is a loss function, such as squared error, and $\langle \cdot \rangle_{\mathcal{D}}$ denotes the **empirical expectation**:

$$\langle h(\mathbf{x}, \mathbf{y}) \rangle_{\mathcal{D}} \equiv \frac{1}{|\mathcal{D}|} \sum_{(\mathbf{x}_n, \mathbf{y}_n) \in \mathcal{D}} h(\mathbf{x}_n, \mathbf{y}_n) \tag{39}$$

This empirical expectation is a **Monte Carlo approximation** of the true expectation:

- The underlying distribution is unknown;
- The data is assumed to be sampled i.i.d. from that distribution;
- By the law of large numbers, the sample mean approximates the expectation as N grows.

Solving this gives the optimal parameters:

$$\mathbf{\Theta}_{\text{opt}} = \arg\min_{\mathbf{\Theta}} \langle C(\mathbf{y}, \mathbf{g}(\mathbf{x}; \mathbf{\Theta})) \rangle_{\mathcal{D}}, \tag{40}$$

and predictions are given by:

$$\hat{\mathbf{y}} = \mathbf{g}(\mathbf{x}; \mathbf{\Theta}_{\text{opt}}) \tag{41}$$

3.2 Linear least squared estimation

Assume a linear model:

$$\mathbf{g}(\mathbf{x};\mathbf{W}) = \mathbf{W}\mathbf{x} \tag{42}$$

Then the regression problem becomes:

$$\min_{\mathbf{W}} \left\langle \|\mathbf{y} - \mathbf{W}\mathbf{x}\|^2 \right\rangle_{\mathcal{D}} \quad \Leftrightarrow \quad \min_{\mathbf{W}} \sum_{n=1}^{N} \|\mathbf{y}_n - \mathbf{W}\mathbf{x}_n\|^2 \tag{43}$$

Optimal solution:

$$\mathbf{W}_{\text{LLSE}} = \arg\min_{\mathbf{W}} \left\langle \|\mathbf{y} - \mathbf{W}\mathbf{x}\|^2 \right\rangle_{\mathcal{D}} \tag{44}$$

Prediction:

$$\hat{\mathbf{y}} = \mathbf{W}_{\text{LLSE}}\mathbf{x} \tag{45}$$

In the scalar case, the unbiased solution is given by:

$$\mathbf{W}_{\text{LLSE}} = \frac{\mathbf{y}^T\mathbf{x}}{\mathbf{x}^T\mathbf{x}}, \quad \hat{\mathbf{y}} = \frac{\mathbf{y}^T\mathbf{x}}{\mathbf{x}^T\mathbf{x}}\mathbf{x} \tag{46}$$

Compared to the LMMSE case, where $\hat{\mathbf{y}} = \mathbf{R}_{YX}\mathbf{R}_X^{-1}\mathbf{x}$. It can be observed that although LLSE and LMMSE arise in different contexts — the former from data-driven regression, the latter from distribution-based estimation — their solutions share nearly identical mathematical forms. This is because both are derived by minimizing mean squared error under a linear model, and both rely on **linear averaging operators**.

More generally, let's rewrite in the matrix form: $y = X\beta + \varepsilon$. The OLS(Ordinary Least Squares) solution is: $\boxed{\hat{\beta}^{OLS} = (X^TX)^{-1}X^Ty}$

3.3 From Scalar to Matrix: Ordinary Least Squares Solution

Consider a linear regression model with **n samples** and **p features**. For the i-th sample, let $x_i \in \mathbb{R}^p$ be the input vector, $y_i \in \mathbb{R}$ be the target, and ε_i be the noise term. The scalar form of the model is:

$$y_i = \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip} + \varepsilon_i = x_i^T\beta + \varepsilon_i, \tag{47}$$

where $\beta = (\beta_1, \ldots, \beta_p)^T \in \mathbb{R}^p$ is the parameter vector.

Stacking all n samples together, we define:

- Response vector:

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \in \mathbb{R}^n \tag{48}$$

- Input matrix:

$$X = \begin{bmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_n^T \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \ldots & x_{1p} \\ x_{21} & x_{22} & \ldots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \ldots & x_{np} \end{bmatrix} \in \mathbb{R}^{n\times p} \tag{49}$$

- Noise vector:

$$\varepsilon = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{bmatrix} \in \mathbb{R}^n \tag{50}$$

The full model in matrix form is:

$$\boxed{y = X\beta + \varepsilon} \tag{51}$$

If an intercept term b is included, we augment the input vector at the end:

$$X' = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1p} & 1 \\ x_{21} & x_{22} & \dots & x_{2p} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{n1} & x_{n2} & \dots & x_{np} & 1 \end{bmatrix} \in \mathbb{R}^{n\times(p+1)}, \quad \beta' = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_p \\ b \end{bmatrix} \in \mathbb{R}^{p+1} \tag{52}$$

And the affine model becomes:

$$\boxed{y = X'\beta' + \varepsilon} \tag{53}$$

3.3.1 *Derivation

(This section is not required, you can find it in any Linear Algebra course.)

Expanding the squared norm:

$$\|y - X\beta\|^2 = (y - X\beta)^T(y - X\beta) = y^Ty - 2\beta^TX^Ty + \beta^TX^TX\beta. \tag{54}$$

Taking the gradient with respect to β:

$$\nabla_\beta \left(\|y - X\beta\|^2\right) = -2X^Ty + 2X^TX\beta. \tag{55}$$

Setting the gradient to zero:

$$X^TX\beta = X^Ty. \tag{56}$$

Assuming X^TX is invertible, we solve for optimal β:

$$\hat{\beta} = (X^TX)^{-1}X^Ty. \tag{57}$$

To ensure that the critical point is a minimum, don't forgert to compute and check the second derivative (Hessian) of the loss function.

3.4 Evaluating our Regression

No model is ever perfect, How should we evaluate the quality of our regression model(not limited to the linear case)? We can take both **absolute error metrics** and **relative goodness-of-fit scores** into consideration. Several commonly used evaluation tools are summarized below.

- Absolute error metrics: MSE, MAE

  Mean Squared Error (MSE):

  $$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{58}$$

  Mean Absolute Error (MAE):

  $$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{59}$$

  These metrics are expressed in the same units as the output and provide intuitive and **quantitative** measures of prediction error.

- Residual analysis(histogram) The **residual** for the i-th data point is:

  $$r_i = y_i - \hat{y}_i \tag{60}$$

  The residual vector is:

  $$\mathbf{r} = \mathbf{y} - \hat{\mathbf{y}} \tag{61}$$

  Residuals capture the part of the data unexplained by the model. Usually they resemble the white noise.

- Relative fit measures: RSS, TSS, R^2, and adjusted R^2

  Residual Sum of Squares (RSS):

  $$\text{RSS} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{62}$$

  RSS measures the total squared error between the actual values and the predicted values from the model. A smaller RSS means the model fits the data better.

  Total Sum of Squares (TSS):

  $$\text{TSS} = \sum_{i=1}^{n}(y_i - \bar{y})^2 \tag{63}$$

  TSS captures the total variation in the target variable y_i, relative to its mean.

  Regression Sum of Squares (Reg SS):

  $$\text{Reg SS} = \sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2 \tag{64}$$

Reg SS $= 0$ means $\hat{y}_i = \bar{y}$. It is the worst case, our model learns nothing, simply predicting $\hat{y}$ as $\bar{y}$. In linear regression,Reg SS + RSS = TSS

Coefficient of Determination (R^2):

$$R^2 = \frac{\text{Reg SS}}{\text{TSS}} = 1 - \frac{\text{RSS}}{\text{TSS}} \tag{65}$$

R^2 is an indicator of the model's goodness of fit, representing the extent to which the independent variables account for the variability in the dependent variable.

- $R^2 = 1$: perfect model fit
- $R^2 = 0$: model explains nothing beyond the mean

Adjusted R^2:

$$R^2_{\text{adj}} = 1 - \left(1 - R^2\right) \cdot \frac{n-1}{n-p-1} \tag{66}$$

3.5 Weekly special: Spurious Correlation

A high R^2 or low MSE does *not* imply a causal relationship between inputs and outputs. Regression captures statistical association, not causality. Keep it in mind all the time!

Here is an interesting website which includes many spurious, ridiculous cases: Spurious Correlations. We randomly choose one example as 12

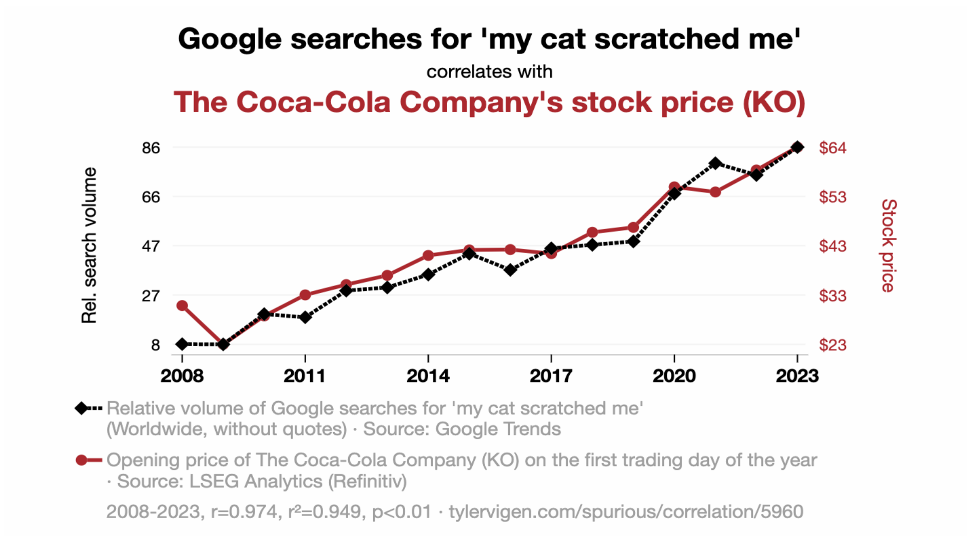


Figure 12: A funny case

This figure constructs a specious correlation between 'cat scratch' and 'The stock price of Coca-cola', but there is no way to tell that A causes B even if the R^2 seems perfect!

4 Classification

Classification is a type of supervised learning task where the goal is to predict a **discrete class label** from a finite set of categories, given a set of input features. In other words, the model learns to assign each input $\boldsymbol{x} \in \mathbb{R}^d$ to a label $y \in \{1, 2, \ldots, K\}$, where K is the total number of classes. Mathematically, the goal of classification can be expressed as learning a function

$$f : \mathbb{R}^d \to \{1, 2, \ldots, K\} \tag{67}$$

that maps input features to one of K class labels.

Examples of classification tasks include:

- Determining whether an email is spam or not.
- Identifying whether an image contains a cat, dog, or human.
- Predicting whether a patient has a certain disease.

The following table highlights the differences between classification and regression tasks:

Property	**Regression**	**Classification**
Target Type	Continuous value $y \in \mathbb{R}$	Discrete label $y \in \{1, 2, \ldots, K\}$
Objective	Predict **numerical** quantity	Predict **class** label (e.g., spam or not)
Typical loss	Mean Squared Error (MSE)	Cross-Entropy Loss
Model output	Real number	Class label or class probabilities
Evaluation metrics	MSE, MAE, R^2, etc.	Accuracy, Precision, Recall, F1, AUC

Table 2: Comparison between Regression and Classification tasks

4.1 From Linear Regression to Logistic Regression

Starting point: Linear regression. In regression problems, we predict a continuous value using a linear model:

$$\hat{y} = \boldsymbol{w}^T \boldsymbol{x} + b \tag{68}$$

This works well when the target $y \in \mathbb{R}$ is real-valued — for example, predicting housing prices or temperature.

What if the target is a class label? Now suppose we face a classification problem: predicting whether an email is spam ($y = 1$) or not spam ($y = 0$). The output should be a *label*, not a continuous value. Can we still use linear regression to solve this?

Naive idea: direct use of regression for classification. We could attempt to use the same linear model to generate a **score**, and then apply a threshold. If the output score exceeds threshold, determine it as class 1, otherwise class 0:

$$\hat{y} = \begin{cases} 1, & \text{if } \boldsymbol{w}^T \boldsymbol{x} + b > 0.5 \\ 0, & \text{otherwise} \end{cases} \tag{69}$$

But this approach has serious problems:

- The raw output $\boldsymbol{w}^T\boldsymbol{x}+b \in \mathbb{R}$ is unbounded. We can't find a well-defined threshold.
- It is not normalized and cannot be interpreted as a probability.
- Training such a model (e.g., using MSE loss) is not appropriate for classification. We only care about accuracy.

Better idea: output a probability, then decide. Instead of trying to directly predict the class label, we can follow a two-step strategy:

1. First, estimate the posterior probability $P(y = 1 \mid \boldsymbol{x})$.
2. Then, classify using a threshold:

$$\hat{y} = \begin{cases} 1, & \text{if } P(y = 1 \mid \boldsymbol{x}) > 0.5 \\ 0, & \text{otherwise} \end{cases} \tag{70}$$

How to map a linear output into a probability? To convert the real-valued output of a linear model into a valid **probability** in $[0, 1]$, we apply the **sigmoid** function:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{71}$$

The resulting model is:

$$P(y = 1 \mid \boldsymbol{x}) = \sigma(\boldsymbol{w}^T\boldsymbol{x} + b) \tag{72}$$

This is the model structure of **Logistic Regression**.

The sigmoid function maps any real-valued input to the open interval $(0, 1)$, making it a natural candidate for modeling probabilities in binary classification. Choosing sigmoid as our mapping function has many benefits:

- **Range-bound:** The output is always within $(0, 1)$, which aligns with the semantics of probability.
- **Smooth and differentiable:** The sigmoid function is continuously differentiable, enabling gradient-based optimization methods such as gradient descent.
- **Monotonicity:** Larger input values produce larger output probabilities, which preserves the ranking of class scores.

However, it inherently has some unavoidable limitations, among which the worst one is **Vanishing gradients:**

As $z \to \pm\infty$, the gradient $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ approaches zero. This slows down convergence, especially when the model outputs are far from the decision boundary.

Classification via probability. Logistic Regression first estimates the probability of class membership by sigmoid function, then uses a threshold to make a prediction:

$$\hat{y} = \begin{cases} 1, & \text{if } \sigma(\boldsymbol{w}^T\boldsymbol{x} + b) > 0.5 \\ 0, & \text{otherwise} \end{cases} \tag{73}$$

After such a long effort, we can finally make a brief summary here:

Summary.

- Linear regression outputs real numbers, which is not suitable for classification.
- Luckily, we find a useful mapping function Sigmoid. It compresses real-valued outputs into $[0, 1]$, enabling probabilistic interpretation.
- Classification can be achieved by estimating probabilities and applying a decision threshold.
- Logistic Regression = linear model + sigmoid + threshold rule.

4.2 Maximum Likelihood Estimation and Cross-Entropy Loss

In logistic regression, the model outputs the probability of the label being 1:

$$P(y = 1 \mid \boldsymbol{x}) = \sigma(\boldsymbol{w}^T \boldsymbol{x} + b) \tag{74}$$

For binary classification, each label $y \in \{0, 1\}$, we can unify the output expression as:

$$P(y \mid \boldsymbol{x}) = \sigma(\boldsymbol{w}^T \boldsymbol{x})^y \cdot (1 - \sigma(\boldsymbol{w}^T \boldsymbol{x}))^{1-y} \tag{75}$$

To **train** and **evaluate** a logistic regression model, we need to define a suitable **loss function** that measures how well the model predicts the true labels in the training data.

Since the model outputs a probability distribution over $y \in \{0, 1\}$, it is natural to use a probabilistic training criterion. A widely used principle is **maximum likelihood estimation (MLE)**: we choose the model parameters that make the observed data most likely under the model.

Maximum likelihood estimation (MLE). Given a dataset $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$, and assuming i.i.d. samples, the likelihood function is:

$$\mathcal{L}(\boldsymbol{w}) = \prod_{i=1}^n P(y_i \mid \boldsymbol{x}_i) = \prod_{i=1}^n \sigma(\boldsymbol{w}^T \boldsymbol{x}_i)^{y_i} \cdot (1 - \sigma(\boldsymbol{w}^T \boldsymbol{x}_i))^{1-y_i} \tag{76}$$

To simplify optimization, we take the negative log-likelihood (also known as the loss function):

$$\mathcal{L}_{\text{NLL}}(\boldsymbol{w}) = -\sum_{i=1}^n \left[y_i \log \sigma(\boldsymbol{w}^T \boldsymbol{x}_i) + (1 - y_i) \log(1 - \sigma(\boldsymbol{w}^T \boldsymbol{x}_i)) \right] \tag{77}$$

This is the cross-entropy loss. The negative log-likelihood of logistic regression corresponds exactly to the binary cross-entropy loss, which measures the discrepancy between the true label $y_i \in \{0, 1\}$ and predicted probability $\hat{p}_i \in (0, 1)$:

$$\mathcal{L}_{\text{CE}} = -\sum_{i=1}^n [y_i \log \hat{p}_i + (1 - y_i) \log(1 - \hat{p}_i)] \tag{78}$$

where $\hat{p}_i = \sigma(\boldsymbol{w}^T \boldsymbol{x}_i)$.

Here we list some cases to visualize the CE loss of different predicted probabilities:

Predicted Probability $\hat{p}$	**True Label** $y=1$	**Cross-Entropy Loss** $-\log(\hat{p})$
0.9	1	$-\log(0.9) \approx 0.105$
0.6	1	$-\log(0.6) \approx 0.511$
0.1	1	$-\log(0.1) \approx 2.303$

Table 3: Binary classification: cross-entropy loss comparison for different predicted probabilities when $y = 1$

Gradient of the cross-entropy loss. To train the logistic regression model, we minimize the cross-entropy loss:

$$\mathcal{L}_{\text{CE}} = -\sum_{i=1}^{n} [y_i \log \hat{p}_i + (1 - y_i) \log(1 - \hat{p}_i)] \tag{79}$$

where $\hat{p}_i = \sigma(\boldsymbol{w}^T \boldsymbol{x}_i)$ is the model prediction.

Taking the gradient with respect to the model parameters $\boldsymbol{w}$, we obtain:

$$\nabla_{\boldsymbol{w}} \mathcal{L}_{\text{CE}} = \sum_{i=1}^{n} (\hat{p}_i - y_i) \boldsymbol{x}_i \tag{80}$$

This gradient has a simple and intuitive interpretation: the update direction is proportional to the difference between the predicted probability and the true label.

Optimization and parameter update. The gradient is used in optimization algorithms such as gradient descent:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \cdot \nabla_{\boldsymbol{w}} \mathcal{L}_{\text{CE}} \tag{81}$$

where $\eta > 0$ is the learning rate.

This iterative process adjusts the model parameters to minimize the loss, gradually improving classification performance on the training data.

Remark. The combination of the sigmoid activation and cross-entropy loss leads to particularly clean gradients. In deeper models like multi-layer perceptrons (&*chapterV*), the gradient descent method will be discussed in detail.

4.3 Multi-Class Classification: Softmax

In **binary** classification, we used the sigmoid function to model the probability of class 1:

$$P(y = 1 \mid \boldsymbol{x}) = \sigma(\boldsymbol{w}^T \boldsymbol{x}) \in (0, 1) \tag{82}$$

However, in **multi-class** classification, the target variable takes values in $\{1, 2, \ldots, K\}$ where $K > 2$. We need the model to output a valid probability distribution over all K classes:

$$\hat{\boldsymbol{y}} = [\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_K] \quad \text{such that} \quad P(y = k \mid \boldsymbol{x}) = \hat{p}_k \geq 0, \ \sum_{k=1}^{K} \hat{p}_k = 1 \tag{83}$$

Classification rule. After that, the predicted class is chosen as the one with the highest probability:

$$\hat{y} = \arg\max_k \hat{p}_k \tag{84}$$

Solution: Softmax function. Softmax function generalizes the sigmoid function to the multi-class setting. Given a score vector $\boldsymbol{z} = [z_1, \ldots, z_k] = \boldsymbol{w_k}^T \boldsymbol{x} + \boldsymbol{b_k}$, the softmax output is defined as:

$$\hat{p}_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}, \quad \text{for } k = 1, 2, \ldots, K \tag{85}$$

This ensures each $\hat{p}_k \in (0, 1)$ and $\sum_{k=1}^{K} \hat{p}_k = 1$, so the output can be interpreted as a categorical probability distribution.

Likely, the cross-entropy loss can be expressed as:

$$\mathcal{L}_{\text{CE}} = -\sum_{k=1}^{K} y_k \log \hat{p}_k \tag{86}$$

Here is another example of the CE loss of different predicted result:

Model	**Predicted $\hat{\boldsymbol{p}}$**	**True Class: 2**	**Cross-Entropy Loss**
A	[0.3, 0.4, 0.3]	$\hat{p}_2 = 0.4$	$-\log(0.4) \approx 0.916$
B	[0.1, 0.8, 0.1]	$\hat{p}_2 = 0.8$	$-\log(0.8) \approx 0.223$
C	[0.6, 0.2, 0.2]	$\hat{p}_2 = 0.2$	$-\log(0.2) \approx 1.609$

Table 4: CE loss for different predictions on a 3-class classification task (true class: 2)

Workflow Summary. The typical workflow for solving a classification problem is summarized in the following steps:

1. **Model the posterior probability.** Instead of predicting class labels directly, we build a model to estimate the posterior probability:

   $$P(y = k \mid \boldsymbol{x}) \quad \text{for } k = 1, \ldots, K \tag{87}$$

   using functions like sigmoid (for binary classification) or softmax (for multi-class classification).

2. **Define the loss function.** To train the model, we compare the predicted probability distribution $\hat{\boldsymbol{p}}$ with the true label $\boldsymbol{y}$ using the cross-entropy loss:

   $$\mathcal{L}_{\text{CE}} = -\sum_{k=1}^{K} y_k \log \hat{p}_k \tag{88}$$

3. **Optimize model parameters.** We minimize the loss function with respect to the model parameters using optimization algorithms such as gradient descent. In neural network models, this process is implemented using backpropagation:

   $$\theta \leftarrow \theta - \eta \cdot \nabla_\theta \mathcal{L}_{\text{CE}} \tag{89}$$

4. **Evaluate the trained model.** After training, the model is evaluated on unseen data using metrics such as:
   - **Accuracy**: Proportion of correct predictions.
   - **AUC (Area Under Curve)**: Measures the trade-off between true positive and false positive rates.
   - **F1 Score**: Harmonic mean of precision and recall, especially useful for imbalanced datasets.

4.4 *Weekly special: The physical version of Sigmoid: Fermi-Dirac Distribution

The sigmoid function has been a central tool in our journey through classification, smoothly mapping real-valued inputs to the interval $(0, 1)$, giving us a well-behaved probability to work with. But surprisingly, this humble function has a long-lost cousin in the realm of quantum physics.

As a physics enthusiast (and once a practitioner), I can't help but share with you a fascinating parallel: the **Fermi-Dirac Distribution**. Don't be afriad my friend, that's what makes us an electrical and computer engineer.

What does it describe?

In quantum statistical mechanics, the Fermi-Dirac distribution gives the probability that a given quantum state at energy E is occupied by a *fermion* (a particle with half-integer spin, like electrons, protons, or neutrons).

Crucially, fermions obey the **Pauli exclusion principle**: no two fermions can occupy the same quantum state. This principle shapes the behavior of electrons in atoms, metals, and semiconductors.

The Fermi-Dirac Distribution takes the form of:

$$f(E) = \frac{1}{1 + e^{(E-\mu)/(kT)}} \tag{90}$$

Here:

- E: the energy level of the state,
- μ: the chemical potential (equal to the Fermi energy at absolute zero),
- T: absolute temperature,
- k: the Boltzmann constant.

At temperature $T = 0$, this function becomes a step function: all states below μ are occupied ($f = 1$), and all above are empty ($f = 0$). As T increases, the distribution becomes smoother — just like sigmoid!

Surprising similarity:

If we define a new variable $z = -\frac{E-\mu}{kT}$, the Fermi-Dirac distribution becomes:

$$f(E) = \frac{1}{1 + e^{-z}} = \sigma(z) \tag{91}$$

Yes — it's the sigmoid function in disguise.

From classification boundaries to the quantum world — sigmoid is everywhere.

Bonus. And just as **fermions** follow the Fermi-Dirac distribution, **bosons** (particles like photons and helium-4 atoms) follow a different rule — the **Bose-Einstein distribution**:

$$f(E) = \frac{1}{e^{(E-\mu)/(kT)} - 1} \tag{92}$$

Notice the subtle but crucial sign change in the denominator — a simple minus instead of a plus — yet it leads to entirely different physical behavior.

Can we transform our sigmoid function into this form? Just give it a shot.

5 Multi-layer Perceptrons