A Comprehensive Note on Machine Learning[1]

Jue Guo

January 22, 2024

Contents

Preface

The primary purpose of this document is educational, specifically for the courses I teach (**CSE 474/574 Introduction to Machine Learning, CSE 455/555 Pattern Recognition, and CSE 676 Deep Learning**), as well as for my personal reference. A substantial portion of the material herein is directly referenced or adapted from established texts and sources, and is not claimed as original content. This document is intended as a supplementary teaching and learning resource and is not authorized for commercial use, redistribution, or sale without my explicit consent.

The majority of the material referenced comes from the following sources:

[1] Zhang, Aston, et al. Dive into deep learning. Cambridge University Press, 2023.

[2] Bishop, C. M., & Nasrabadi, N. M. (2006). Pattern recognition and machine learning (Vol. 4, No. 4, p. 738). New York: Springer.

[3] Hart, P. E., Stork, D. G., & Duda, R. O. (2000). Pattern classification. Hoboken: Wiley.

[4] Burkov, A. (2019). The hundred-page machine learning book (Vol. 1, p. 32). Quebec City, QC, Canada: Andriy Burkov.

[5] Burkov, A. (2020). Machine learning engineering (Vol. 1). Montreal, QC, Canada: True Positive Incorporated. [1]

All other referenced materials and sources are cited in the bibliography section of this document. This compilation is intended to provide a comprehensive overview and guide for students and practitioners of machine learning, drawing upon a wide range of foundational and contemporary sources in the field.

[1] This is a birthday present from my lab member and good friend Peiyao Xiao

Part I

Basic Machine Learning

Chapter 1

Introduction

1.1 What is Machine Learning

Machine learning is a subfield of computer science that is concerned with building algorithms which to be useful, rely on a collection of examples of some phenomenon.

- The examples come from nature, handcrafted by humans or generated by another algorithms.

Machine learning can also be defined as the process of solving a practical problem by

1. gathering a dataset
2. algorithmically building a statistical model based on the dataset.

The statistical model is assumed to be used somehow to solve the practical problem.

1.2 Types of Learninig

Learning can be **supervised**, **semi-supervised**, **unsupervised** and **reinforcement**.

1.2.1 Supervised Learning

In **supervised learning**, the **dataset** is the collection of **labeled examples** $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$.

- Each element $\mathbf{x}_i$ among N is called a **feature vector**. A feature vector is a vector in which each dimension $j = 1, ..., D$ contains a value that describes the example somehow. That value is called a **feature** and is denoted as $x^{(j)}$.
- The **label** y_i can be either an element belonging to a finite set of **classes** $1, 2, ..., C$, or a real number, or a more complex structure, like a vector, a matrix, a tree or a graph. Unless otherwise stated, y_i is either one of a finite set of classes or a real number.

The goal of a **supervised learning algorithm** is to use the dataset to produce a **model** that takes a feature vector $\mathbf{x}$ as input and outputs information that allows deducing the label for this feature vector.

1.2.2 Unsupervised Learning

In **unsupervised learning**, the dataset is a collection of **unlabeled examples** $\{\mathbf{x}_i\}_{i=1}^{N}$. The goal of an **unsupervised learning algorithm** is to create a model that takes a feature vector $\mathbf{x}$ as input and either transforms it into another vector or into a value that can be used to solve a practical problem. For example,

- in **clustering**, the model returns the id of the cluster for each feature vector in the dataset.
- in **dimensionality reduction**, the output of the model is a feature vector that has fewer features than the input $\mathbf{x}$;
- in **outlier detection**, the output is a real number that indicates how $\mathbf{x}$ is different from a "typical" example in the dataset.

1.2.3 Semi-Supervised Learning

In **semi-supervised learning**, the dataset contains both labeled and unlabeled examples.

- Usually the quantity of unlabeled examples is much higher than the number of labeled examples.

The goal of **semi-supervised learning algorithm** is the same as the goal of the supervised learning algorithm.

- The hope here is that using many unlabeled examples can help the learning algorithm to find a better model.

How does the learning benefit from adding more unlabeled examples?

- By adding unlabeled examples, you add more information about your problem: *a larger sample reflects better the probability distribution the data we labeled came from.*

1.2.4 Reinforcement Learning

Reinforcement learning is a subfield of machine learning where the machine "lives" in an environment and is capable of perceiving the *state* of that environment as vector of features.

- The machine can execute *actions* in every state. Different actions bring different *rewards* and could also move the machine to another state of the environment.

The goal of a reinforcement learning algorithm is to learn a *policy*. A policy is a function (similar to the model in supervised learning) that takes the feature vector of a state as input and outputs an optimal action to execute in that state. The action is optimal if it maximizes the *expected average reward*.

1.3 How Supervised Learning Works

Supervised learning is the type of machine learning most frequently used in practice. The supervised learning process starts with gathering the data. The data for supervised learning is a collection of pairs (input,output).

- Inputs could be anything, for example, email messages, pictures, or sensor measurements.
- Outputs are usually real numbers, or labels (e.g. "spam", "not_spam", etc). In some cases, outputs are vectors (e.g., four coordinates of the rectangle around a person on the picture), sequences (e.g. ["adjective", "adjective", "noun"] for the input "big beautiful car") or have some other structure.

You want to solve spam detection using supervised learning. First, you gather the data of 10,000 email messages and you have your label "spam" or "not_spam" for each message. With the data at hand, you have to represent each message using a feature vector.

A common approach it to use **bag of words**, is to take a dictionary of English words (e.g. 20,000 alphabetically sorted words) and stipulate that in our feature vector:

- the first feature is equal to 1 if the email message contains the word "a"; other wise, this feature is 0;
- the second feature is equal to 1 if the email message contains the word "aaron"; otherwise, this feature equals 0;

- ...
- the feature at position 20,000 is equal to 1 if the email message contains the word "zulu"; otherwise, this feature is equal to 0.

You repeat the above procedure for every email message in our collection, which gives us 10,000 feature vectors (each vector having the dimensionality of 20,000) and a label ("spam"/"not_spam").

We successfully converted input data into machine-readable type, but the output labels are still in the form of human-readable text. Some learning algorithms require transforming labels into numbers. For demonstration purpose, we will use a supervised learning algorithm called **Support Vector Machine** (SVM). This algorithm requires the positive label (in our case it's "spam") has the numeric value of +1 (one), and the negative label ("not_spam") has the value of -1 (minus one).

You now have a **dataset** and a **learning algorithm**, you will need to apply the learning algorithm to the dataset to get the **model**.

SVM sees every feature vector as a point in a high-dimensional space (in our case, space is 20,000 – dimensional). The algorithm puts all feature vectors on an imaginary 20,000-dimensional plot and draws an imaginary 19,999 – dimensional line (a *hyperplane*) that separates examples with positive labels from examples with negative labels. In machine learning, the boundary separating the examples of different classes is called the **decision boundary**.

Hyperplane is denoted by two **parameters**

$$\mathbf{wx} - b = 0,$$

- a real-valued vector $\mathbf{w}$ of the same dimensionality as our input vector $\mathbf{x}$, and a real number b
- $\mathbf{wx} = w^{(1)}x^{(1)} + w^{(2)}x^{(2)} + \ldots + w^{(D)}x^{(D)}$, and D is the number of dimensions of the feature vector $\mathbf{x}$.

The predicted label for some input feature vector $\mathbf{x}$ is given like this:

$$y = \text{sign}(\mathbf{wx} - b),$$

where `sign` is a mathematical operator that takes any value as input and returns +1 if the input is a positive number or −1 if the input is a negative number.

The goal of the learning algorithm, SVM in this case, is to leverage the dataset and find the optimal values $\mathbf{w}^*$ and b^* for parameters $\mathbf{w}$ and b. Once the learning algorithm identifies these optimal values, the **model** $f(\mathbf{x})$ is then defined as:

$$f(\mathbf{x}) = \text{sign}\,(\mathbf{w}^*\mathbf{x} - b^*)$$

How do we find these optima values? It turns out it is an optimization problem. Machines are good at optimizing functions under constraints.

$$\begin{aligned} \mathbf{w}\mathbf{x}_i - b \geq +1 \quad & \text{if } y_i = +1 \\ \mathbf{w}\mathbf{x}_i - b \leq -1 \quad & \text{if } y_i = -1 \end{aligned}$$

It will also be ideal if the hyperplane separates positive examples from negative ones with the largest **margin**. The margin is the distance between the closestt examples of two classes, as defined by the decision boundary. A large margin contributes to a better **generalization**, that is how well the model will classify new examples in the future.

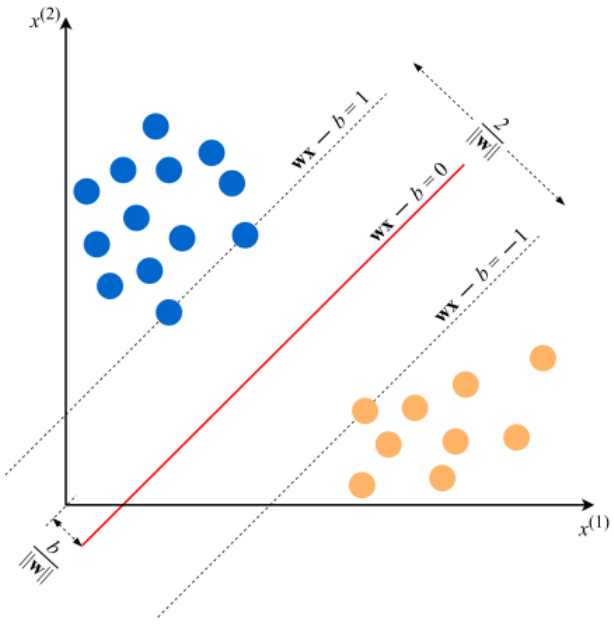


Figure 1.1: An example of an SVM model for two-dimensional feature vectors.

Let's do some quick refreshment

Distance Formulas in Euclidean Space

Distance Between Two Points
In two-dimensional space, for points $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$, the distance is calculated as:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Distance From a Point to a Line
In two-dimensional space, for a line defined by $ax + by + c = 0$ and a point $P(x_0, y_0)$, the distance is:

$$d = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}$$

Distance Between Two Parallel Lines
For two parallel lines with equations $ax + by + c_1 = 0$ and $ax + by + c_2 = 0$, the distance is:

$$D = \frac{|c_2 - c_1|}{\sqrt{a^2 + b^2}}$$

So, the optimization problem that we want the machine to solve looks like this:

Minimize $\|\mathbf{w}\|$ *subject to* $y_i(\mathbf{w}\mathbf{x}_i - b) \geq 1$ *for* $i = 1, \ldots, N$. *The expression* $y_i(\mathbf{w}\mathbf{x}_i - b) \geq 1$ *is just a compact way to write the above two constraints.*

More on SVMs later. This simple example should give you an idea how supervised learning works.

1.4 Why the Model Works on New Data

Let's refer to Figure 1.1. If two classes are separable from one another by a decision boundary, then, obviously, examples that belong to each class are located in two different subspaces which the decision boundary creates.

If the examples used for training were selected randomly, independently of one another, and following the same procedure, then statistically, it is *more likely* that the new negative example will be located on the plot somewhere not too far from other negative examples. The idea goes with the positive examples as well.

Chapter 2

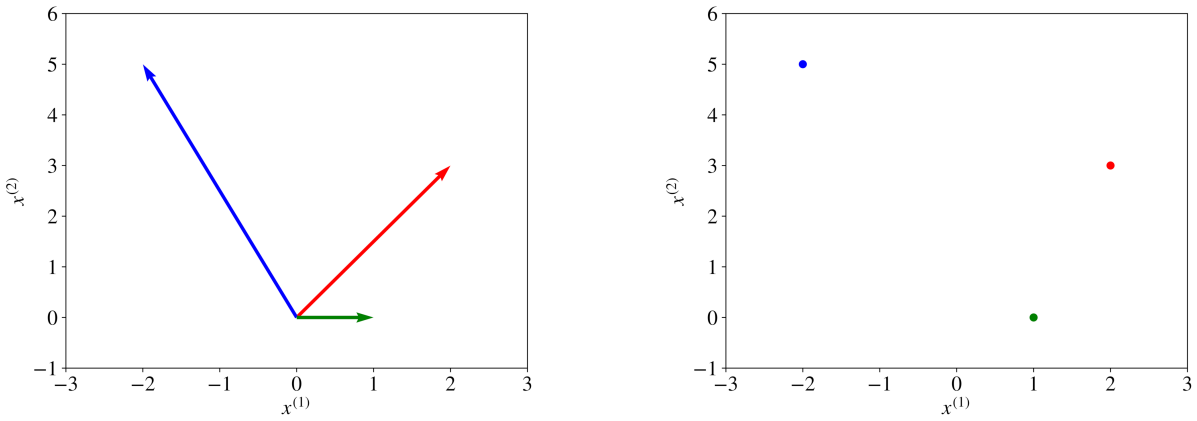
Notation and Definition

2.1 Notation

We will review all the necessary notation and mathematics for us to continue the journey of Machine Learning.

2.1.1 Data Structure

A **scalar** is simple numerical value, like 15 or -3.25, denoted by an italic letter, like x or a. A **vector** is an ordered list of scalar values, called attributes. Vector is denoted by bold character $\mathbf{x}$ or $\mathbf{w}$. A **matrix** is a rectangular array of numbers arranged in rows and columns.



(a) First subfigure

(b) Second subfigure

Figure 2.1: Three vectors visualized as directions and as points.

A **matrix** is a rectangular array of numbers arranged in rows and columns, which are denoted with bold capital letters, such as $\mathbf{A}$ or $\mathbf{W}$. A **set** is an unordered collection of unique elements. We denote a set as a calligraphic capital character, for example, $\mathcal{S}$. When an element belongs to a set $\mathcal{S}$, we write $x \in \mathcal{S}$. We can obtain a new set $\mathcal{S}_3$ as an **intersection** of two set $\mathcal{S}_1$ and $\mathcal{S}_2$, written as $\mathcal{S} \leftarrow \mathcal{S}_1 \cap \mathcal{S}_2$. Also we can obtain a new set by **union**, $\mathcal{S}_3 \leftarrow \mathcal{S}_1 \cup \mathcal{S}_2$.

2.1.2 Capital Sigma Notation

The summation over a collection $\mathcal{X} = \{x_1, x_2, \ldots, x_{n-1}, x_n\}$ or over the attributes of a vector $\mathbf{x} = \left[x^{(1)}, x^{(2)}, \ldots, x^{(m-1)}, x^{(m)}\right]$ is denoted like this:

$$\sum_{i=1}^{n} x_i \stackrel{\text{def}}{=} x_1 + x_2 + \ldots + x_{n-1} + x_n, \text{ or else: } \sum_{j=1}^{m} x^{(j)} \stackrel{\text{def}}{=} x^{(1)} + x^{(2)} + \ldots + x^{(m-1)} + x^{(m)}$$

The notation $\stackrel{\text{def}}{=}$ means "is defined as".

2.1.3 Capital Pi Notation

$$\prod_{i=1}^{n} x_i \stackrel{\text{def}}{=} x_1 \cdot x_2 \cdot \ldots \cdot x_{n-1} \cdot x_n$$

- A product of elements in a collection or attributes of a vector.

2.1.4 Operations on Sets

Given the expression:

$$\mathcal{S}' \leftarrow \left\{x^2 \mid x \in \mathcal{S}, x > 3\right\}$$

This notation is used to define a derived set creation operator. It means that we create a new set $\mathcal{S}'$ by including the square of each element x from the set $\mathcal{S}$, under the condition that x is greater than 3. In other words, $\mathcal{S}'$ is comprised of the squares of all elements in $\mathcal{S}$ which are greater than 3.

Additionally, the cardinality operator $|\mathcal{S}|$ is used to denote the number of elements in the set $\mathcal{S}$. For example, if $\mathcal{S} = \{1, 2, 4, 5\}$, then $\mathcal{S}' = \{16, 25\}$ as only 4 and 5 from $\mathcal{S}$ satisfy the condition $x > 3$. The **cardinality** $|\mathcal{S}|$ in this case would be 4.

2.1.5 Operations on Vectors

Vector Addition and Subtraction: The sum and difference of two vectors $\mathbf{x}$ and $\mathbf{z}$ are defined component-wise as:

$$\mathbf{x} + \mathbf{z} = \left[x^{(1)} + z^{(1)}, \ldots, x^{(m)} + z^{(m)}\right]$$

$$\mathbf{x} - \mathbf{z} = \left[x^{(1)} - z^{(1)}, \ldots, x^{(m)} - z^{(m)}\right]$$

Example: For $\mathbf{x} = [1, 2]$ and $\mathbf{z} = [3, 4]$,

$$\mathbf{x} + \mathbf{z} = [1 + 3, 2 + 4] = [4, 6]$$

Scalar Multiplication: A vector multiplied by a scalar c results in a scaled vector:

$$\mathbf{x}c = \left[cx^{(1)}, \ldots, cx^{(m)}\right]$$

Example: For $\mathbf{x} = [1, 2]$ and $c = 3$,

$$\mathbf{x}c = [3 \times 1, 3 \times 2] = [3, 6]$$

Dot Product: The dot product of two vectors $\mathbf{w}$ and $\mathbf{x}$ is a scalar:

$$\mathbf{w}\mathbf{x} = \sum_{i=1}^{m} w^{(i)} x^{(i)}$$

Example: For $\mathbf{w} = [1, 2]$ and $\mathbf{x} = [3, 4]$,

$$\mathbf{w}\mathbf{x} = 1 \times 3 + 2 \times 4 = 3 + 8 = 11$$

Matrix-Vector Multiplication: Multiplying a matrix $\mathbf{W}$ by a vector $\mathbf{x}$ yields another vector. For example:

$$\begin{aligned}
\mathbf{W}\mathbf{x} &= \begin{bmatrix} w^{(1,1)} & w^{(1,2)} & w^{(1,3)} \\ w^{(2,1)} & w^{(2,2)} & w^{(2,3)} \end{bmatrix} \begin{bmatrix} x^{(1)} \\ x^{(2)} \\ x^{(3)} \end{bmatrix} \\
&\stackrel{\text{def}}{=} \begin{bmatrix} w^{(1,1)}x^{(1)} + w^{(1,2)}x^{(2)} + w^{(1,3)}x^{(3)} \\ w^{(2,1)}x^{(1)} + w^{(2,2)}x^{(2)} + w^{(2,3)}x^{(3)} \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{w}^{(1)}\mathbf{x} \\ \mathbf{w}^{(2)}\mathbf{x} \end{bmatrix}
\end{aligned}$$

Example: For

$$\mathbf{W} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \text{ and } \mathbf{x} = \begin{bmatrix} 5 \\ 6 \end{bmatrix},$$

$$\mathbf{W}\mathbf{x} = \begin{bmatrix} 1 \times 5 + 2 \times 6 \\ 3 \times 5 + 4 \times 6 \end{bmatrix} = \begin{bmatrix} 17 \\ 39 \end{bmatrix}$$

Transpose and Multiplication: For the transpose of a vector $\mathbf{x}$ denoted $\mathbf{x}^\top$, and a matrix $\mathbf{W}$, the multiplication $\mathbf{x}^\top\mathbf{W}$ is given by:

$$\begin{aligned}
\mathbf{x}^\top\mathbf{W} &= \begin{bmatrix} x^{(1)} & x^{(2)} \end{bmatrix} \begin{bmatrix} w^{(1,1)} & w^{(1,2)} & w^{(1,3)} \\ w^{(2,1)} & w^{(2,2)} & w^{(2,3)} \end{bmatrix} \\
&\stackrel{\text{def}}{=} \left[w^{(1,1)}x^{(1)} + w^{(2,1)}x^{(2)}, w^{(1,2)}x^{(1)} + w^{(2,2)}x^{(2)}, w^{(1,3)}x^{(1)} + w^{(2,3)}x^{(2)}\right]
\end{aligned}$$

Example: For

$$\mathbf{x} = \begin{bmatrix} 7 \\ 8 \end{bmatrix} \text{ and } \mathbf{W} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix},$$

$$\mathbf{x}^\top \mathbf{W} = \begin{bmatrix} 7 \times 1 + 8 \times 4, 7 \times 2 + 8 \times 5, 7 \times 3 + 8 \times 6 \end{bmatrix} = \begin{bmatrix} 39, 54, 69 \end{bmatrix}$$

2.1.6 Functions

Definition of a Function
A function is a relation that associates each element x of a set $\mathcal{X}$, known as the domain, to a single element y of another set $\mathcal{Y}$, known as the codomain. This relation is denoted as $y = f(x)$, where f is the name of the function, x is the input or argument, and y is the output. The input variable is also referred to as the variable of the function.

Example: Consider the function $f(x) = x^2$ defined on the domain $\mathcal{X} = \mathbb{R}$. For $x = 2$, the output is $f(2) = 2^2 = 4$.

Local and Global Minima
The function $f(x)$ has a local minimum at $x = c$ if $f(x) \geq f(c)$ for every x in an open interval around c. An open interval, such as $(0, 1)$, includes all numbers between its endpoints but not the endpoints themselves. The smallest value among all local minima is known as the global minimum.

Example: In the function $f(x) = (x - 1)^2$, the local (and global) minimum occurs at $x = 1$ since $f(x) \geq f(1) = 0$ for all x.

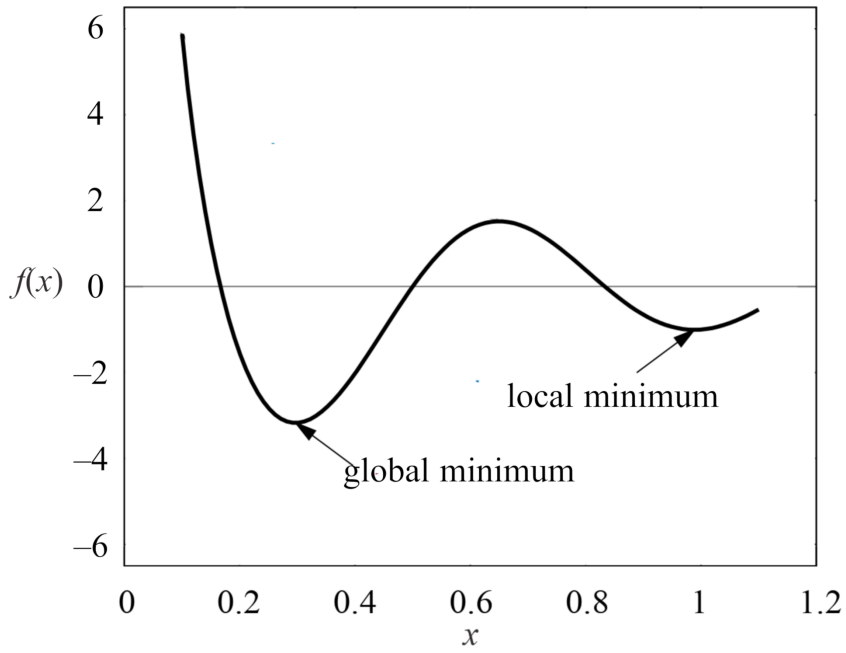


Figure 2.2: A local and a global minima of a function.

Vector Functions
A vector function, denoted $\mathbf{y} = \mathbf{f}(x)$, is a function that returns a vector $\mathbf{y}$. Its argument can be either a vector or a scalar.

Example: For the vector function $\mathbf{f}(x) = [x, x^2]$, with $x = 2$, the output is $\mathbf{f}(2) = [2, 2^2] = [2, 4]$.

2.1.7 Max and Arg Max

Given a set of values $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$, the operator $\max_{a \in \mathcal{A}} f(a)$ returns the highest value of $f(a)$ for all elements in the set $\mathcal{A}$. Conversely, the operator $\arg\max_{a \in \mathcal{A}} f(a)$ identifies the specific element a in the set $\mathcal{A}$ that maximizes the function $f(a)$.

In cases where the set is implicit or infinite, we can use the notation $\max_a f(a)$ or $\arg\max_a f(a)$ respectively. Similarly, the operators min and arg min function in a comparable way, determining the lowest value of a function and the

2.1.8 Assignment Operator

The expression $a \leftarrow f(x)$ means that the variable a gets the new value: the result of $f(x)$. We say that the variable a gets assigned a new value. Similarly, $\mathbf{a} \leftarrow [a_1, a_2]$ means that the vector variable $\mathbf{a}$ gets the two-dimensional vector $[a_1, a_2]$.

2.1.9 Derivative and Gradient

A **derivative** f' of a function f is a function or a value that describes how fast f grows (or decreases). If the derivative f' is a function, then the function f can grow at a different pace in different regions of its domain.

we can use **chain rule** when we encounter hard-to-differentiate function. For instance if $F(x) = f(g(x))$, where f and g are some functions, then $F'(x) = f'(g(x))g'(x)$.

Gradient is the generalization of derivative for functions that take several inputs (or one input in the form of a vector or some other complex structure). A gradient of a function is a vector of **partial derivatives**. For example, $f\left(\left[x^{(1)}, x^{(2)}\right]\right) = ax^{(1)} + bx^{(2)} + c$, then the partial derivative of function f *with respect to* $x^{(1)}$, denoted as $\frac{\partial f}{\partial x^{(1)}}$, is given by,

$$\frac{\partial f}{\partial x^{(1)}} = a + 0 + 0 = a$$

where a is the derivative of the function $ax^{(1)}$; the two zeroes are respectively derivatives of $bx^{(2)}$ and c, because $x^{(2)}$ is considered constant when we compute the derivative with respect to $x^{(1)}$, and the derivative of any constant is zero. Similarly, the partial derivative of function f with respect to $x^{(2)}$, $\frac{\partial f}{\partial x^{(2)}}$, is given by,

$$\frac{\partial f}{\partial x^{(2)}} = 0 + b + 0 = b$$

The gradient of function f, denoted as ∇f is given by the vector $\left[\frac{\partial f}{\partial x^{(1)}}, \frac{\partial f}{\partial x^{(2)}}\right]$.

2.2 Random Variable

A **random variable**, usually written as an italic letter, like X, is a variable whose possible values are numerical outcomes of a random phenomenon. There are two types of random variables: **discrete** and **continuous**. A **discrete random variable** takes on only countable number of distinct values such as *red*, *yellow*, *blue* or 1,2,3, ...

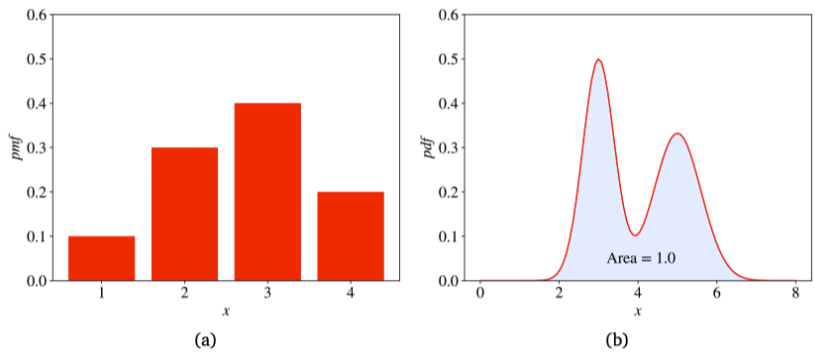


Figure 2.3: A probability mass function and a probability density function.

The **probability distribution** of a discrete random variable is described by a list of probability associated with each of its possible values. This list of probability is called a **probability mass function** (pmf)(Fig.2.3, **a**).

A **continuous random variable** takes an infinite number of possible values in some interval. The probability distribution of a continual random variable (a continuous probability distribution) is described by a **probability density function** (pdf) (Fig.2.3, **b**).

Let a discrete random variable X have k possible values $\{x_i\}_{i=1}^{k}$. The **expectation** of X denoted as $\mathbb{E}[X]$ is given by,

$$\begin{aligned}\mathbb{E}[X] &\stackrel{\text{def}}{=} \sum_{i=1}^{k} [x_i \cdot \Pr(X = x_i)] \\ &= x_1 \cdot \Pr(X = x_1) + x_2 \cdot \Pr(X = x_2) + \cdots + x_k \cdot \Pr(X = x_k)\end{aligned} \tag{2.1}$$

where $\Pr(X = x_i)$ is the probability that X has the value x_i according to the pmf. The expectation of a random variable is also called the **mean**, **average** or **expected value** and is frequently denoted with the letter μ,

Now the **standard deviation**, defined as,

$$\sigma \stackrel{\text{def}}{=} \sqrt{\mathbb{E}\left[(X - \mu)^2\right]}$$

Variance, denoted as σ^2 or $\text{var}(X)$, is defined as,

$$\sigma^2 = \mathbb{E}\left[(X - \mu)^2\right]$$

For a discrete random variable, the standard deviation is given by:

$$\sigma = \sqrt{\Pr(X = x_1)(x_1 - \mu)^2 + \Pr(X = x_2)(x_2 - \mu)^2 + \cdots + \Pr(X = x_k)(x_k - \mu)^2}$$

The expectation of a continuous random variable X is given by,

$$\mathbb{E}[X] \stackrel{\text{def}}{=} \int_{\mathbb{R}} x f_X(x) dx \tag{2.2}$$

where f_X is the pdf of the variable X and $\int_{\mathbb{R}}$ is the integral of function $x f_X$

Real-Life Examples of Probability Concepts

Standard Deviation of a Discrete Random Variable: Consider a dice game where you roll a six-sided die. Each face represents a different prize amount in dollars: {1, 2, 3, 4, 5, 6}. The probability of each outcome is $\frac{1}{6}$ for a fair die.
Mean Calculation: The mean (μ) or expected value of your winnings per roll is:

$$\mu = \frac{1+2+3+4+5+6}{6} = 3.5 \,\text{dollars}$$

Standard Deviation Calculation: The standard deviation σ is given by:

$$\sigma = \sqrt{\sum_{i=1}^{6} \left(\frac{1}{6}\right) \times (i - 3.5)^2}$$

Expectation of a Continuous Random Variable: Imagine the waiting time for a bus, which follows a continuous uniform distribution between 0 and 1 hour.
Expectation Calculation: The expectation of the waiting time, $\mathbb{E}[X]$, is calculated as:

$$\mathbb{E}[X] = \int_0^1 x \, dx$$

$$\mathbb{E}[X] = \int_0^1 x dx = \left[\frac{1}{2}x^2\right]_0^1 = \frac{1}{2}\left(1^2\right) - \frac{1}{2}\left(0^2\right) = \frac{1}{2}$$

The result of this integral is $\frac{1}{2}$ hour, indicating the average waiting time.

The property of the pdf that the area under its curve is 1 mathematically means that $\int_{\mathbb{R}} f_X(x)dx = 1$. Most of the time we don't know f_X, but we can observe some values of X. In machine learning, we call these values **examples**, and the collection of these examples is called a **sample** or a **dataset**.

2.3 Unbiased Estimator

Because f_X is usually unknown, but we have sample $S_X = \{x_i\}_{i=1}^N$, we often content ourselves not with the true values of statistics of the probability distribution, such as expectation, but with their **unbiased estimators**.

We say that $\hat{\theta}\left(S_X\right)$ is an unbiased estimator of some statistic θ calculated using a sample

S_X drawn from an unknown probability distribution if $\hat{\theta}(S_X)$ has the following property:

$$\mathbb{E}\left[\hat{\theta}(S_X)\right] = \theta$$

where $\hat{\theta}$ is a **sample statistic**, obtained using a sample S_X and not the real statistic θ that can be obtained only knowing X; the expectation is taken over all possible samples drawn from X. Intuitively, this means that if you can have an unlimited number of such sample as S_X, and you compute some unbiased estimator, such as $\hat{\mu}$, using each sample, then the average of all these $\hat{\mu}$ equals the real statistic μ that you would get computed on X.

It can be shown that an unbiased estimator of an unknown $\mathbb{E}[X]$ (Eq.2.1 or Eq.2.2) is given by $\frac{1}{N}\sum_{i=1}^{N} x_i$ (called in statistics the **sample mean**).

2.4 Bayes' Rule

The conditional probability $\Pr(X = x \mid Y = y)$ is the probability of the random variable X to have a specific value x given another random variable Y has a specific value of y. The **Bayes' Rule** (also known as the **Bayes' Therem**) stipulate that:

$$\Pr(X = x \mid Y = y) = \frac{\Pr(Y = y \mid X = x)\Pr(X = x)}{\Pr(Y = y)}$$

2.5 Parameter Estimation

Bayes' Rule comes in handy when we have a model of X's distribution, and this model f_θ is a function that has some parameters in the form of a vector θ. An example of such a function could be the Gaussian function that has two parameters, μ and σ, and is defined as:

$$f_{\boldsymbol{\theta}}(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{2.3}$$

where $\boldsymbol{\theta} \overset{\text{def}}{=} [\mu, \sigma]$ and π is the constant $(3.14159\ldots)$.

This function has all the properties of a pdf, which is the pdf of one of the most frequently used in practice probability distributions called **Gaussian distribution** or **normal distribution** and denote as $\dot{\mathcal{N}}\left(\mu, \sigma^2\right)$. Therefore, we can use it as a model of an unknown distribution of X. We can update the values of parameters in the vector θ from the data using the Bayes's Rule.

$$\Pr(\theta = \hat{\theta} \mid X = x) \leftarrow \frac{\Pr(X = x \mid \theta = \hat{\theta}) \Pr(\theta = \hat{\theta})}{\Pr(X = x)} = \frac{\Pr(X = x \mid \theta = \hat{\theta}) \Pr(\theta = \hat{\theta})}{\sum_{\tilde{\theta}} \Pr(X = x \mid \theta = \tilde{\theta}) \Pr(\theta = \tilde{\theta})} \tag{2.4}$$

where $\Pr(X = x \mid \theta = \hat{\theta}) \stackrel{\text{def}}{=} f_{\hat{\theta}}$. If we have a sample $\mathcal{S}$ of X and the set of possible values for θ is finite, we can easily estimate $\Pr(\theta = \hat{\theta})$ by applying Bayes' Rule iteratively, one example $x \in \mathcal{S}$ at a time. The initial value $\Pr(\theta = \hat{\theta})$ can be guessed such that $\sum_{\hat{\theta}} \Pr(\theta = \hat{\theta}) = 1$. This guess of the probabilities for different $\hat{\theta}$ is called the **prior**.

First, we compute $\Pr\left(\theta = \hat{\theta} \mid X = x_1\right)$ for all possible values $\hat{\theta}$. Then, before updating $\Pr(\theta = \hat{\theta} \mid X = x)$ once again, this time for $x = x_2 \in \mathcal{S}$ using Eq.2.4, we replace the prior $\Pr(\theta = \hat{\theta})$ in Eq.2.4 by the new estimate $\Pr(\theta = \hat{\theta}) \leftarrow \frac{1}{N} \sum_{x \in \mathcal{S}} \Pr(\theta = \hat{\theta} \mid X = x)$.

The optimal parameters θ^* given one example is obtained using the principle of **maximum a posterior** (or MAP):

$$\theta^* = \arg\max_{\theta} \prod_{i=1}^{N} \Pr\left(\theta = \hat{\theta} \mid X = x_i\right) \tag{2.5}$$

If the set of possible values for θ isn't finite, then we need to optimize Eq. 2.5 directly using a numerical optimization routine, such as gradient descent. Usually, we optimize the natural logarithm of the right-hand side expression in Eq. 2.5 because the logarithm of a product becomes the sum of logarithms and it's easier for the machine to work with a sum than with a product.

2.6 Parameters vs. Hyperparameters

A *hyper-parameter* is a property of a learning algorithm, usually (but not always) having a numerical value. That value influences the way the algorithm works. Hyper-parameters aren't learned by the algorithm itself from data. They have to be set by the data analyst before running the algorithm. *Parameters* are variables that define the model learned by the learning algorithm. *Parameters* are directly modified by the learning algorithm based on the training data. The goal of learning is to find such values of parameters that make the model optimal in a certain sense.

2.7 Classification vs. Regression

Classification is a problem of automatically assigning a **label** to an **unlabeled example**; Spam detection. Classification probem is solved by **classification learning algo-**

rithm that takes a collection of **labeled examples** as inputs and produces a **model** that can take an unlabeled example as input and either directly output a label or output a number that can be used by the analyst to deduce the label. An example of such a number is a probability.

In a classification problem, a label is a member of a finite set of **classes**. If the size of the set of classes is two, then it is a **binary** or **binomial classification**. **Multiclass classification** (also called **multinomial**) is a classification problem with three or more classes.

Regression is a problem of predicting a real-valued label (often called a **target**) given an unlabeled example. The regression problem is solved by a **regression learning algorithm** that taskes a collection of labeled examples as inputs and produces a model that can take an unlabeled example as input and output a target.

2.8 Model-Based vs. Instance-Based Learning

Most supervised learning algorithm are model-based. *Model-based learning algorithms* use the training data to create a **model** that has **parameters** learned from the training data. *Instance-based learning algorithms* use the whole dataset as the model. One instance-based algorithm frequently used in practice is **k-Nearest Neighbors** (kNN).

2.9 Shallow vs. Deep Learning

A **shallow learning** algorithm learns the parameters of the model directly from the features of the training examples. The **neural network** learning algorithms, specifically those networks with more than one **layer** between input and output. Such neural networks are called **deep neural networks**. In deep neural network learning (or, simply, **deep learning**), contrary to shallow learning, most model parameters are learned not directly from the features of the training examples, but from the outputs of the preceeding layers.

Chapter 3

Fundamental Algorithms

In this chapter, we will go through popular machine learning algorithms. In general, machine learning algorithms that are the basis of artificial intelligence (AI) such as image recognition, speech recognition, recommendation systems, ranking and personalization of content. They aren't generally designed to infer the underlying *generative process* (e.g., to model something), but rather to predict or classify with the most accuracy.

3.1 Three Basic Algorithms

We will first provide you with basic tools to use, so we will start the simple three *linear regression, k-nearest neighbors*, and *k-means*.

3.1.1 Linear Regression

When you use it, you are making the assumption that there is a *linear* relationship between an outcome variable (sometimes also called the response vairable, dependent variable, or label) and a predictor (sometimes also called an independent variable, explanatory variable, or feature); or between one variable and several other variables, in which case you're *modeling* the relationship as having a linear structure.

Algorithm or a Model?

The two words seem to be used interchangeably when their actual definitions are not the same thing at all. In the purest sense, an algorithm is a set of rules or steps to follow to accomplish some task, and a model is an attempt to describe or capture the world.
To differentiate the two in machine learning, think of an algorithm as a process and a model as the result of running the process. In other words, you use an algorithm to create a model.

How to describe something of a linear nature? $y = f(x) = \beta_0 + \beta_1 x$. Let's approach this from the perspective of deterministic function first.

Example 1 Suppose you run a social networking site that charges a monthly subscription fee of $25, and that this is your only source of revenue. Each month you collect data and count your number of users and total revenue. You've done this daily over the course of

two years, recording it all in a spreadsheet. You could express this data as a series of points. Here are the first four:

$$S = \{(x, y) = (1, 25), (10, 250), (100, 2500), (200, 5000)\}$$

If you showed this to someone else who didn't even know how much you charged or anything about your business model (what kind of friend wasn't paying attention to your business model?!), they might notice that there's a clear relationship enjoyed by all of these points, namely $y = 25x$. This is a deterministic function, and it's a linear one. It's also a perfect fit for the data. If you were to plot it, you'd see that it passes through every point (Fig.3.1).

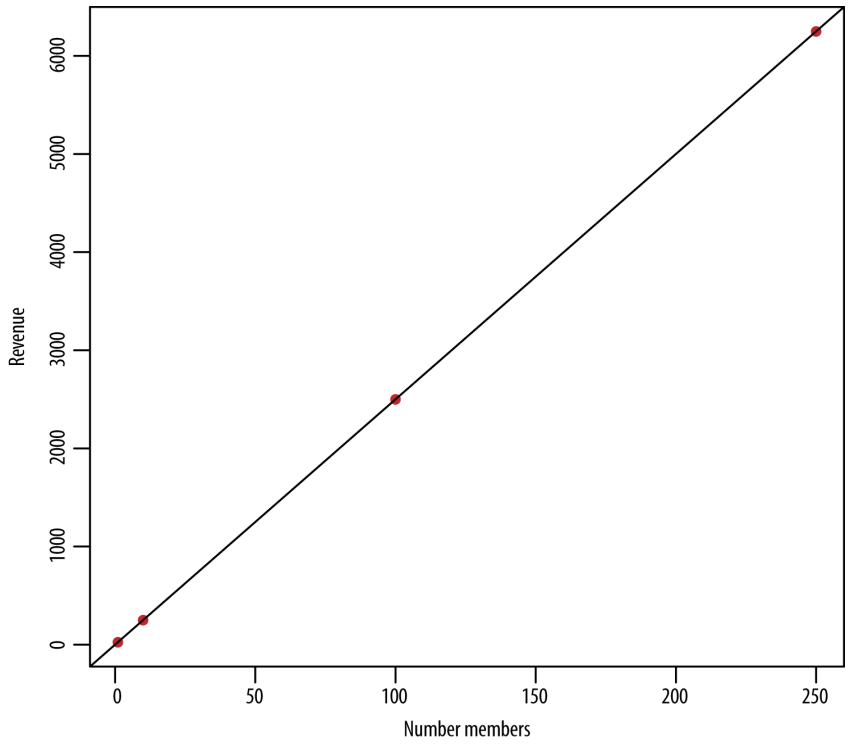


Figure 3.1: An obvious linear pattern

Example 2 Say you have a dataset *keyed* by user (meaning each row contains data for a single user), and the columns represent user behavior on a social networking site over a period of a week. Let's say you feel comfortable that the data is clean at this stage and that you have on the order of hundreds of thousands of users. The names of the columns are total_num_friends, total_new_friends_this_week, num_visits, time_spent, number_ads_shown and so on. During the course of your exploratory analysis, you've randomly sampled 100 users to keep it simple, and you plot pairs of these variables, for example, x = total_new_friends and y = time_spent (in seconds). The business context might be that eventually you want to be able to promise advertisers who bid for space on your website in advance a certain number of users, so you want to be able to forecast

number of users several days or weeks in advance. You decide to plot out the data first (Fig. 3.2):

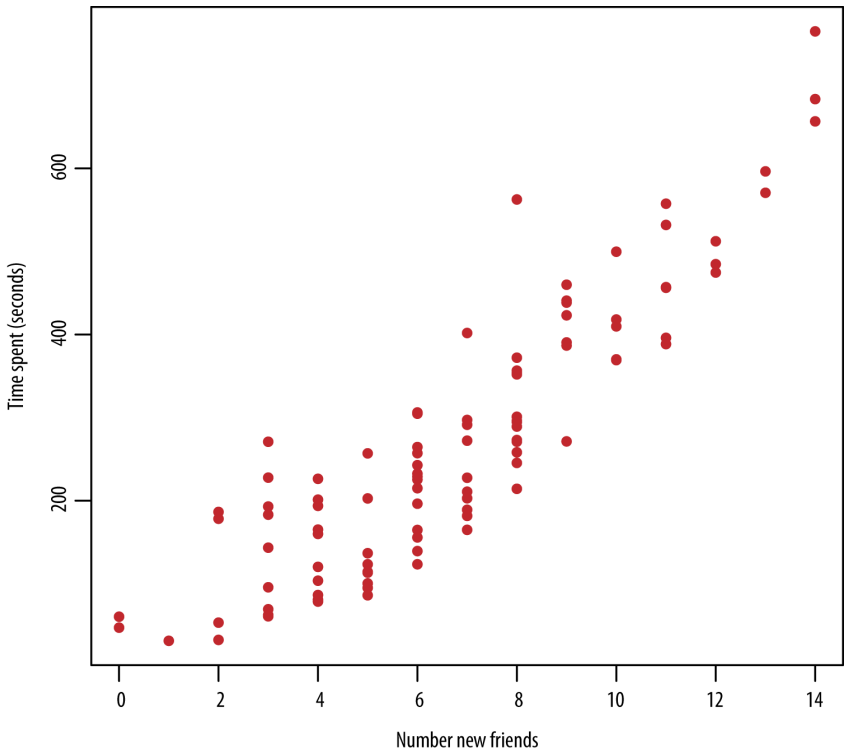


Figure 3.2: Looking kind of linear

The relationship looks *kind of* linear. But be aware that there is no perfectly *deterministic* relationship between number of new friends and time spent on the site, but it makes sense that there is an *association* between these two variables.

Building Blocks There are two things you want to capture in the model. The first is the *trend* and the second is the *variation*. First, we focus on the *trend*. Let's assume there exist a relationship and it is linear. There are many lines and they all look they might work (Fig.3.3).

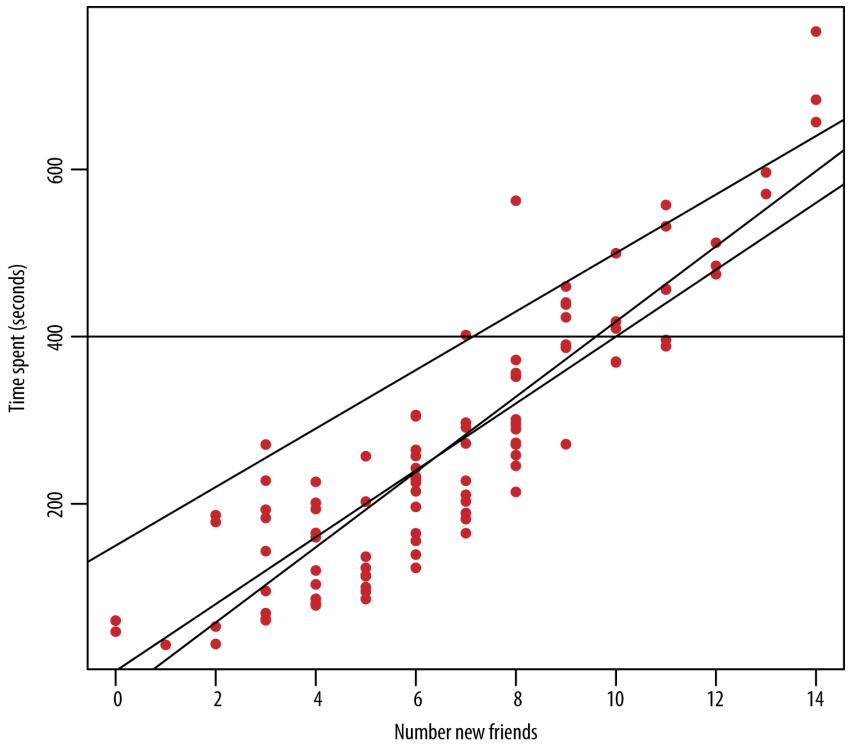


Figure 3.3: Which line is the best fit?

Because you're assuming a linear relationship, start your model by assuming the functional form to be:

$$y = \beta_0 + \beta_1 x$$

Now your job is to find the best choices for β_0 and β_1 using the observed data to estimate them: $(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)$. Writing this with matrix notation results in this:

$$y = \mathbf{X} \cdot \boldsymbol{\beta}$$

Now that we have our model, the rest is fitting the model.

Fitting the model The intuition behind linear regression is that you want to find the line that minimizes the distance between all points and the line. Many lines look approximately correct, but the goal is to find the optimal one. *Optimal* could mean different things, but let's start with optimal to mean the line that, on average, is closest to all the points.

Linear regression seeks to find the line that minimize the sum of the squared distances between the predicted $\widehat{y}_i$ s and the observed y_i s. This is the *least squares* estimation (Fig.3.4).

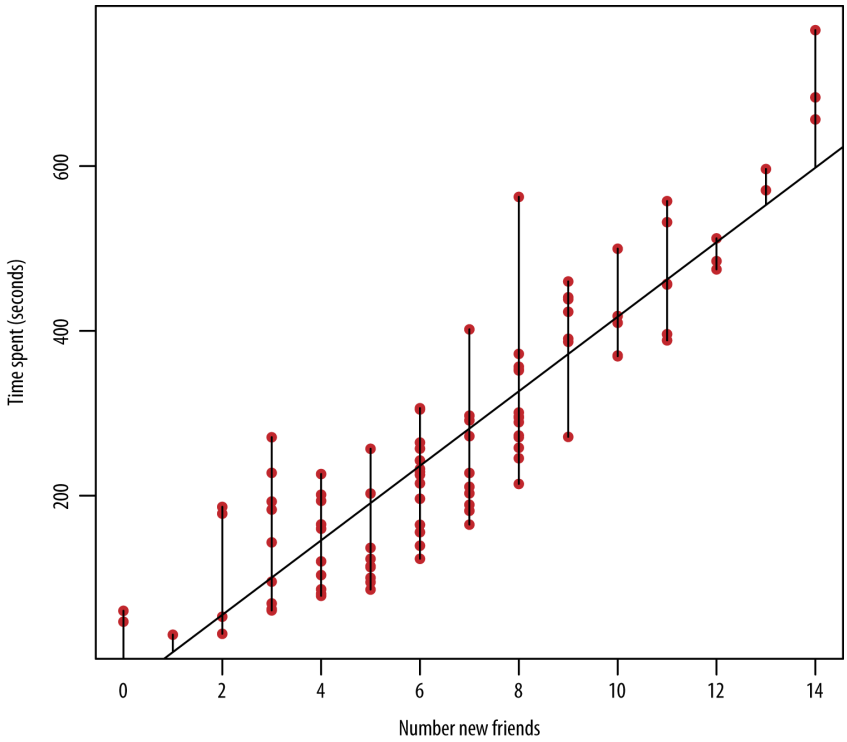


Figure 3.4: The line closest to all the points

To find this line, you'll define the "residual sum of squares" (RSS) as:

$$RSS(\beta) = \Sigma_i \left(y_i - \beta x_i\right)^2 \tag{3.1}$$

where i ranges over the various data points. It is the sum of all the squared vertical distances between the observed points and any given line. Note this is a function of β and you want to optimize with respect to β to find the optimal line.

We have a closed solution to Eq.3.1

$$\hat{\beta} = \left(x^t x\right)^{-1} x^t y$$

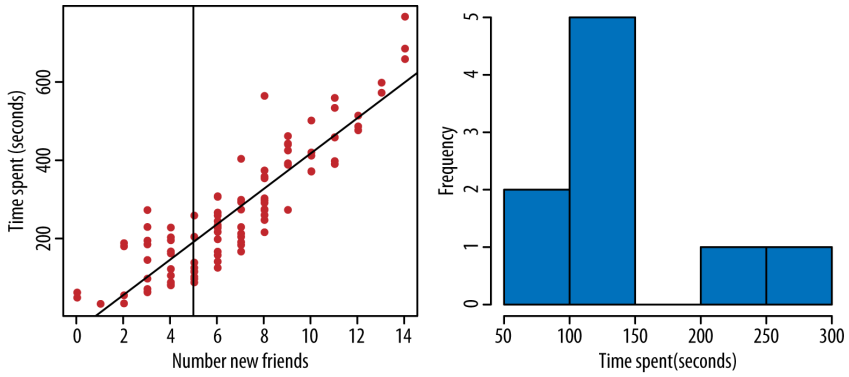


Figure 3.5: On the left is the fitted line. We can see that for any fixed value, say 5, the values for y vary. For people with 5 new friends, we display their time spent in the plot on the right.

Form Fig.3.5, we have modeled the *trend*, but we have not yet modeled the *variation*. The variation is the *noise* in the data. The noise is the randomness inherent in the data. In the real world, we don't know the true underlying relationship between the variables, so we can't predict the future. The model is only as good as the data we have.

Extending Beyond Least Squares Now we will extend this *simple linear regression* model in three primary ways:

1. Adding in modeling assumption about the errors
2. Adding in more predictors
3. Transforming the predictors

Adding in Modeling Assumptions about the Errors If you use the model to predict y for a given x, the prediction is deterministic and does not capture the variability in the observed data. Look at the Fig 3.5. We see a variability when $x = 5$ and you want to capture this variability in the model:

$$y = \beta_0 + \beta_1 x + \epsilon \tag{3.2}$$

the new term ϵ is the noise or *error* in the data. The noise is the randomness inherent in the data. One often makes the modeling assumption that the noise is normally distributed, which is denoted:

$$\epsilon \sim N\left(0, \sigma^2\right) \tag{3.3}$$

With the preceding assumption on the distriubution of noise, this model is saying that, for any given x, the conditional distribution of y given x is $p(y \mid x) \sim N\left(\beta_0 + \beta_1 x, \sigma^2\right)$. How do you fit the model? How do you get the parameters β_0, β_1, σ from the data?

How to fit the model?

Turns out that no matter how the ϵ are distributed, the least squares estimates that we derived are the optimal estimator for βs becasuse they have the property of being unbiased and of being the minimum variance estimators.

So what can you do with your observed data to estimate the variance of the errors? Now that you have the estimated line, you can see how far away the observed data points are from the line itself, and you can treat these differences, also known as *observed errors* or *residuals*, as observation themselves, or estimate of the actual errors, the ϵs. Define $e_i = y_i - \hat{y}_i = y_i - \left(\hat{\beta}_0 + \hat{\beta}_1 x_i\right)$ for $i = 1, \ldots, n$. Then you estimate the variance (σ^2) of ϵ, as:

$$\frac{\sum_i e_i^2}{n-2} \tag{3.4}$$

Why are we dividing by $n-2$? Dividing by $n-2$, rather than just n, produces an *unbiased estimator*.The 2 corresponds to the number of model parameters. This is called the *mean squared error* and captures how much the predicted value varies from the observed. *Mean squared error* is a useful quantity for any prediction problem. In regression in particular, it's also an estimator for your variance, but it can't always be used or interpreted that way.

Evaluation Metrics How confident are you with the estimates?

- *R-squared*

  $R^2 = 1 - \frac{\Sigma_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2}$. This can be interpreted as the proportion of variance explained by our model. The mean squared error is in there getting divided by total error, which is the proportion of variance *unexplained* by our model and we calculate 1 minus that.

- *p-values*

Part II

Advance Machine Learning

Part III

Neural Networks

Chapter 4

Distilling the knowledge in a Neural Network

This chapter is a direct and indirect reference to Hinton et al. (2015).

Simple way to improve the performance of any machine learning algorithm is to train many different models on the same data and then to average their predictions. Making predictions using a whole ensemble of models is too computationally expensive to allow deployment. *It is shown that we can distill the knowledge of an ensemble of models into a single model.* The author introduces a new type of ensemble composed of one or more full models and many specialist models which learn to distinguish fine-grained classes that the full models confuse. Unlike a mixture of experts, these specialist models can be trained rapidly and in parallel.

4.1 Introduction

In large-scale machine learning, we typically use very similar models for the training stage and deployment stage despite their very different requirements. The author provided an analogy of the insects suggests taht we should be willing to train a very cumbersome models if that makes it easier to extract structure from the data. The cumbersome model could be an ensemble of separately trained models or a single very large model trained with a very strong regularizer such as dropout.

Part IV

Convolution Neural Networks

Part V

Adversarial Attacks and Training

Chapter 5

Intriguing Properties of Neural Network

Deep neural networks, known for their exceptional performance in speech and visual recognition tasks, exhibit two notable characteristics (Szegedy et al., 2013). First, *the semantic information in their higher layers is embedded not in individual units but in the collective space they form.* This insights shifts the focus from analyzing single neurons to considering the entire unit group to understand network processing. Second, *these networks display a surprisingly sensitivity to minute, yet percisely tailored alternations (or perturbation).* Such small changes can lead to incorrect outcomes. This vulnerability is not due to random noise; the same modifications can deceive different networks trained on a different subset of the dataset, to misclassify the same input.

5.1 Introduction

Deep neural networks are powerful learning models that achieve excellent performance on visual and speech recognition problems because they can express arbitrary computation that consists of a modest number of massively parallel nonlinear steps. As the resulting computation is automatically discovered by backpropagation via supervised learning, it can be difficult to interpret and can have counter-intuitive properties.

The **first** property is concerned with the semantic meaning of individual units. It seems that the entire space of activation, rather than the individual units, that contains the bulk of the semantic information contrary to prior belief and the **second** property is concerned with the stability of neural networks with respect to small perturbation to their inputs. Apply an *imperceptible* non-random perturbation to a test image, it is possible to arbitrarily change the network's prediction. These perturbation are found by optimizing the input to maximize the prediction error. The perturbed examples are often called "adversarial examples"

5.2 Framework

Notation $x \in \mathbb{R}$ denotes an input image, $\phi(x)$ is an activation values of some layer. (Szegedy et al., 2013) first examine properties of the image of $\phi(x)$, and then search for its blind spots.

Part VI

Recurrent Neural Networks

Part VII

Transformers

Part VIII

Artificial General Intelligence

Chapter 6

Continual Learning: An Overview

6.1 Introduction

Continual Learning is motivated by the fact that human and other organisms has the ability to adapt, accumulate and exploit knowledge. A common setting for continual learning is to learn a sequence of contents one by one and behave as if they were observed simultaneously (Wang et al., 2023). Each task learned throughout the life time can be new skills, new examples of old skills, different environments, etc (Fig.6.1, a). This attribute of continual learning makes it also referred to as **incremental learning** or **lifelong learning**.

Unlike conventional pipeline, where joint training is applied, continual learning is characterized by learning from dynamic data distributions. A major challenge is known as **catastrophic forgetting**, where *adaptation to a new distribution generally results in a largely reduced ability to capture the old ones*. This dilemma is a facet of the trade-off between **learning plasticity** and **memory stability**: an excess of the former interferes with the latter, and vice versa. A good continual learning algorithm should obtain a strong **generalizability** to accommodate distribution differences within and between tasks (Fig.6.1, b). As a naive baseline, retraining all old training samples (if allowed) makes it easy to address the above challenges, but creates huge computational and storage overheads (as well as potential privacy issues). In fact, continual learning is primarily intended to ensure **resource efficiency** of model updates, preferably close to learning only new training samples.

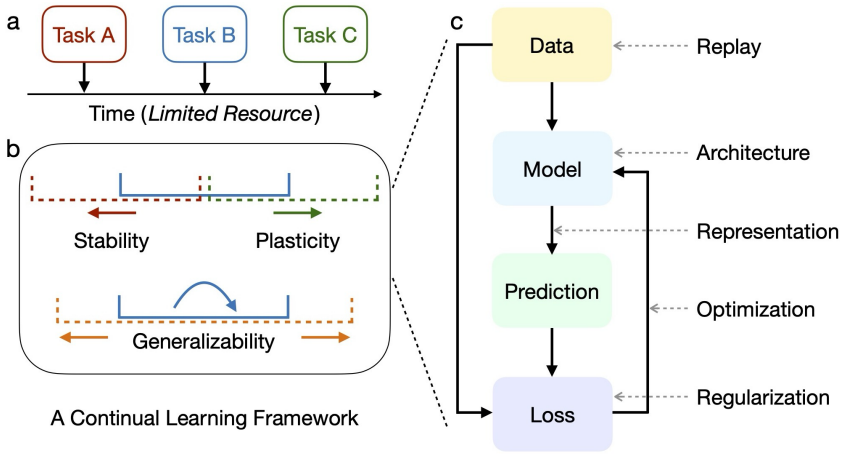


Figure 6.1: A conceptual framework of continual learning. **a**, Continual learning requires adapting to incremental tasks with dynamic data distributions. **b**, A desirable solution should ensure a proper balance between stability (red arrow) and plasticity (green arrow), as well as an adequate generalizability to intra-task (blue arrow) and inter-task (orange arrow) distribution differences. **c**, Representative strategies have targeted various aspects of machine learning.

Numerous efforts have been devoted to addressing the above challenges, which can be conceptually separated into five groups (Fig.6.1, c): *regularization-based approach*; *replay-based approach*; *optimization-based approach*; *representation-based approach*; and *architecture-based approach*. These methods are *closely connected*, e.g., regularization and replay ultimately act to rectify the gradient directions, and *highly synergistic*, e.g., the efficacy of replay can be facilitated by distilling knowledge from the old model.

6.2 Setup

In this section, we first present a basic formulation of continual leanring. Then we introduce typical scenairos and evaluation metrics.

6.2.1 Basic Formulation

A continual learning model parameterized by θ needs to learn corresponding task(s) with no or limited access to old training samples and perform well on their test sets. Formally, an incoming batch of training samples belonging to a task t can be represented as $\mathcal{D}_{t,b} = \left\{\mathcal{X}_{t,b}, \mathcal{Y}_{t,b}\right\}$ where $\mathcal{X}_{t,b}$ is the input data, $\mathcal{Y}_{t,b}$ is the data label, $t \in \mathcal{T} = \{1, \cdots, k\}$ is the task identity and $b \in \mathcal{B}_t$ is the batch index ($\mathcal{T}$ and $\mathcal{B}_t$ denote their space, respectively). Here we define a "task" by its training samples $\mathcal{D}_t$ following the

distribution $\mathbb{D}_t := p(\mathcal{X}_t, \mathcal{Y}_t)$ ($\mathcal{D}_t$ denotes the entire training set by omitting the batch index, likewise for $\mathcal{X}_t$ and $(\mathcal{Y}_t)$, and assume that there is no difference in distribution between training and testing. Under realistic constraints, the data label $\mathcal{Y}_t$ and the task identity t might not be always available. In continual learning, the training samples of each task can arrive incrementally in batches (i.e., $\left\{\left\{\mathcal{D}_{t,b}\right\}_{b\in\mathcal{B}_t}\right\}_{t\in\mathcal{T}}$) or simultaneously (i.e.,$\{\mathcal{D}_t\}_{t\in\mathcal{T}}$).

Scenario	Training	Testing
IIL	$\left\{\left\{\mathcal{D}_{t,b}, t\right\}_{b\in\mathcal{B}_t}\right\}_{t=j}$	$\{p(\mathcal{X}_t)\}_{t=j;t}$ is not required
DIL	$\{\mathcal{D}_t, t\}_{t\in\mathcal{T}}; p(\mathcal{X}_i) \neq p(\mathcal{X}_j)$ and $\mathcal{Y}_i = \mathcal{Y}_j$ for $i \neq j$	$\{p(\mathcal{X}_t)\}_{t\in\mathcal{T}}, t$ is not required
TIL	$\{\mathcal{D}_t, t\}_{t\in\mathcal{T}}; p(\mathcal{X}_i) \neq p(\mathcal{X}_j)$ and $\mathcal{Y}_i \cap \mathcal{Y}_j = \emptyset$ for $i \neq j$	$\{p(\mathcal{X}_t)\}_{t\in\mathcal{T};t}$ is available
CIL	$\{\mathcal{D}_t, t\}_{t\in\mathcal{T}}; p(\mathcal{X}_i) \neq p(\mathcal{X}_j)$ and $\mathcal{Y}_i \cap \mathcal{Y}_j = \emptyset$ for $i \neq j$	$\{p(\mathcal{X}_t)\}_{t\in\mathcal{T};t}$ is unavailable
TFCL	$\left\{\left\{\mathcal{D}_{t,b}\right\}_{b\in\mathcal{B}_t}\right\}_{t\in\mathcal{T}}; p(\mathcal{X}_i) \neq p(\mathcal{X}_j)$ and $\mathcal{Y}_i \cap \mathcal{Y}_j = \emptyset$ for $i \neq j$	$\{p(\mathcal{X}_t)\}_{t\in\mathcal{T}}; t$ is optionally available
OCL	$\left\{\left\{\mathcal{D}_{t,b}\right\}_{b\in\mathcal{B}_t}\right\}_{t\in\mathcal{T}}, \lvert b\rvert = 1; p(\mathcal{X}_i) \neq p(\mathcal{X}_j)$ and $\mathcal{Y}_i \cap \mathcal{Y}_j = \emptyset$ for $i \neq j$	$\{p(\mathcal{X}_t)\}_{t\in\mathcal{T}}; t$ is optionally available
BBCL	$\{\mathcal{D}_t, t\}_{t\in\mathcal{T}}; p(\mathcal{X}_i) \neq p(\mathcal{X}_j), \mathcal{Y}_i \neq \mathcal{Y}_j$ and $\mathcal{Y}_i \cap \mathcal{Y}_j \neq \emptyset$ for $i \neq j$	$\{p(\mathcal{X}_t)\}_{t\in\mathcal{T};t}$ is unavailable
CPT	$\left\{\mathcal{D}_t^{pt}, t\right\}_{t\in\mathcal{T}^{pt}}$, followed by a downstream task j	$\{p(\mathcal{X}_t)\}_{t=j;t}$ is not required

Table 6.1: A formal comparison of typical continual learning scenarios. $\mathcal{D}_{t,b}$: the training samples of task t and batch b. $|b|$: the size of batch b. $\mathcal{B}_t$: the space of incremental batches belonging to task t. $\mathcal{D}_t$: the training set of task t (further specified as $\mathcal{D}_t^{pt}$ for pre-training). $\mathcal{T}$: the space of all incremental tasks (further specified as $\mathcal{T}^{pt}$ for pre-training). $\mathcal{X}_t$: the input data in $\mathcal{D}_t$, $p(\mathcal{X}_t)$: the distribution of $\mathcal{X}_t$.$\mathcal{Y}_t$: the data label of $\mathcal{X}_t$.

6.2.2 Typical Scenairo

Detail of some tyical continual learning scenairos (refer to Table 6.1 for a formal comparision):

- *Instance-Incremental Learning* (IIL): All training samples belong to the same task and arrive in batches.
- *Domain-Incremental Learning* (DIL): Tasks have the same data label space but different input distributions. Task identities are not required.
- ***Task-Incremental Learning*** (TIL): Tasks have disjoint data label spaces. Task identities are provided in both training and testing.
- ***Class-Incremental Learning*** (CIL): Tasks have disjoint data label spaces. Task identities are only provided in training.

- *Task-free Continual Learning* (TFCL): Tasks have disjoint data label spaces. Task identities are not provided in either training or testing.
- *Online Continual Learning* (OCL): Tasks have disjoint data label spaces. Training samples for each task arrive as a one-pass data stream.
- *Blurred Boundary Continual Learning* (BBCL): Task boundaries are blurred, characterized by distinct but overlapping data label spaces.
- *Continual Pre-training* (CPT): Pre-training data arrives in sequence. The goal is to improve the performance of learning downstream tasks.

Lots of the above mentioned scenairo is messy, hence we will focus on the most popular scenairos: Task-Incremental Learning and Class-Incremental Learning.

6.2.3 Evaluation Metrics

Overall performance is typically evaluated by *average accuracy* (AA) and *average incremental accuracy* (AIA). Let $a_{k,j} \in [0, 1]$ denote the classification accuracy evaluated on the test set of the j-th task after incremental learning of the k-th task ($j \leq k$). The output space to compute $a_{k,j}$ consists of the classes in either $\mathcal{Y}_j$ or $\cup_{i=1}^{k} \mathcal{Y}_i$, corresponding to the use of multi-head evaluation (e.g., TIL) or single-head evaluation (e.g., CIL). The two metrics at the k-th task are then defined as

$$\mathrm{AA}_k = \frac{1}{k} \sum_{j=1}^{k} a_{k,j}$$

AA represnets the overall performance at the current moment.

$$\mathrm{AIA}_k = \frac{1}{k} \sum_{i=1}^{k} \mathrm{AA}_i$$

AIA reflects the historical variaion.

Memory stability can be evaluted by *forgetting measure* (FM) and *backward transfer* (BWT). As for the former, the forgetting of a task is calculated by the difference between its maximum performance obtained in the past and its current performance:

$$f_{j,k} = \max_{i \in \{1,\ldots,k-1\}} \left(a_{i,j} - a_{k,j} \right), \forall j < k$$

FM at the k-th task is the average forgetting of all old tasks:

$$\mathrm{FM}_k = \frac{1}{k-1} \sum_{j=1}^{k-1} f_{j,k},$$

As for the latter, BWT evaluates the average influence of learning the k-th task on all old tasks:

$$\text{FWT}_k = \frac{1}{k-1}\sum_{j=2}^{k}\left(a_{j,j} - \tilde{a}_j\right),$$

where the forgetting is usally reflected as a negative BWT.

Learning plasticity can be evaluated by *intransience measure* (IM) and *forward transfer* (FWT). IM is defined as the inability of a model to learn new tasks, calculated by the difference of a task between its joint training performance and continual leanring performance:

$$\text{IM}_k = a_k^* - a_{k,k}$$

where a_k^* is the classification accuracy of a randomly-initialized reference model jointly trained with $\cup_{j=1}^{k}\mathcal{D}_j$ for the k-th task. In comparison, FWT evaluates the average influence of all old tasks on the current k-th task:

$$\text{FWT}_k = \frac{1}{k-1}\sum_{j=2}^{k}\left(a_{j,j} - \tilde{a}_j\right)$$

where $\tilde{a}_j$ is the classification accuracy of a randomly-initialized reference model trained with $\mathcal{D}_j$ for the j-th task.

6.3 Method

In this section, we go through representative continual learning methods.

6.3.1 Regularization-based Aprroach

This direction is characterized by adding explicit regularization terms to balance the old and new tasks, which usually requires storing a frozen copy of the old model for reference (Fig. 6.2). Depending on the target of regularization, such methods can be divided into two sub-directions.

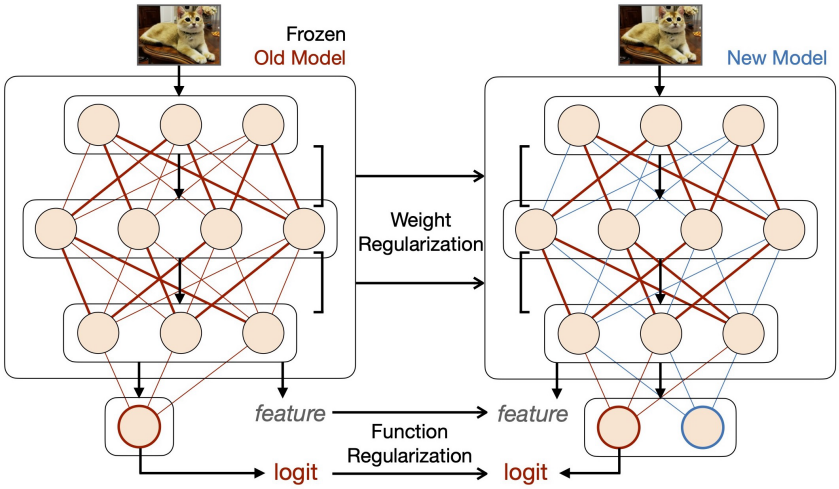


Figure 6.2: Regularization-based approach. This direction is characterized by adding explicit regularization terms to mimic the parameters (weight regularization) or behaviors (function regularization) of the old model.

The first is **weight regularization**, which selectively regularizes the variation of network parameters. A typical implementation is to add a quadratic penalty in loss function that penalizes the variation of each parameter depending on its contribution or "importance" to performing the old tasks, in a form originally derived from online Laplace approximation of the posterior under the Bayesian framework. The importance can be calculated by the Fisher information matrix (FIM), such as EWC. Numerous efforts have been devoted to designing a better imoportance measurement. SI online approximate the importance of each parameter by its contribution to the total loss vaiation and its update length over the entire training trajectory. MAS accumulates the importance measurements based on the sensitivity of predictive results to parameter changes, which is both online and unsupervised. RWalk combines regularization terms of SI and EWC to integrate their advantages.These importance measurements have been shown to be all tantamount to an approximation of the FIM, although stemming from different motivations.

There are also several works refining the implementation of the quadratic penalty.Since the diagonal approaximation of the FIM in EWC might lose information about the old tasks, R-EWC performs a factorized rotation of the parameter space to diagonalize the FIM. XK-FAC further considers the inter-example relations in approximating the FIM to better accommodate batch normalization. Observing the asymmetric effect of parameter changes on old tasks, ALASSO designs an asymmetric quadratic penalty with one of its sides overestimated.

Compared to learning each task within the constraints of the old model, which typically exacerbates the intransience, an *expansion-renormalization* process of obtaining separately the new task solution and renormalizing it with the old model has shown to provide a better stability-plasticity trade off.

The second is **functional regularization**, which targets the intermediate or final output of the prediction function. This strategy typically employs the previously-learned model as the teacher and the currently-trained model as the student, while implementing knowledge distillation (KD) to mitigate catastrophic forgetting.

6.3.2 Replay-based Approach

We group the methods for approximating and recovering old data distributions into this direction (Fig. 6.3). Depending on the content of replay, these methods can be further divided into three sub-directions, each with its own targets and challenges.

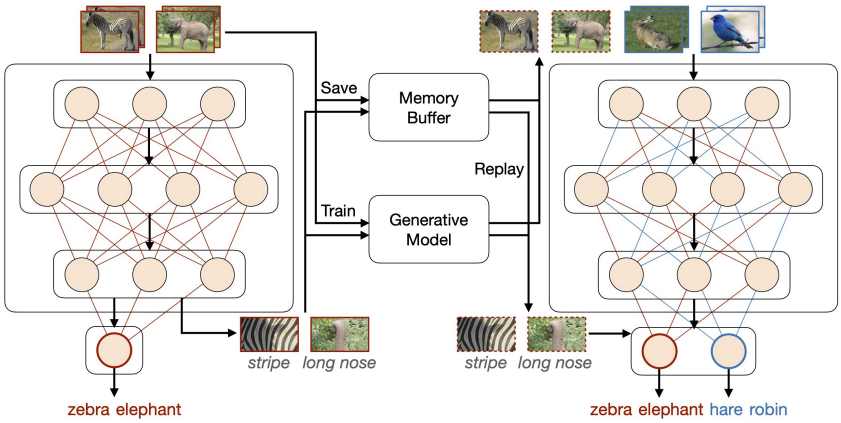


Figure 6.3: Replay-based approach. This direction is characterized by approximating and recovering the old data distributions. Typical subdirections include experience replay, which saves a few old training samples in a memory buffer; generative replay, which trains a generative model to provide generated samples; and feature replay, which recovers the distribution of old features through saving prototypes, saving statistical information or training a generative model.

The first is **experience replay**, which typically stores a few old training samples within a small memory buffer. Due to the extremely limited storage space, the key challenges consist of *how to construct* and *how to exploit* the memory buffer. As for construction, the preserved old training samples should be carefully selected, compressed, augmented and updated, in order to recover adaptively the past information. Some notable sampling methods are *reservoir sampling* where randomly preserves a fixed number of old training samples obtained from each training batch and *ring buffer* ensures an equal number of old training samples per class. *Mean-of-Feature* selects an equal number of old training samples that are closest to the feature mean of each class. For exploitation, experience replay requires an adequate use of the memory buffer to recover the past information. First, the effect of old training samples in optimization can be constrained to avoid catastrophic forgetting and facilitate knowledge transfer. On the other hand, experience

replay can be naturally combined with *knowledge distillation* (KD), which additionally incorporates the past information from the old model.

The second is **generative replay** or pseudo-rehearsal, which usually requires training on additional generative model to replay generated data. This is closely related to continual learning of generative models themselves, as they also require incremental updates.

6.3.3 Optimization-based Approach

Continual learning can be achieved not only by adding additional terms to the loss function (e.g., regularization and replay), but also by explicitly designing and manipulating the optimization programs. A typical idea is to perform **gradient projection**. A typical idea is to perform **gradient projection**. Some replay-based approaches as GEM, A-GEM, LOGD and MER constrain parameter updates to align with the direction of experience replay, corresponding to preserving the previous input space and gradient space with some old training samples.

Chapter 7

New Insights on Reducing Abrupt Representation Change in Online Continual Learning

Experience Replay (ER), where a small subset of past data is stored and replayed alongside new data, has emerged as a simple and effective learning strategy. The author focus on the change in representations of observed data that arises when previously unobserved classes appear in representations of observed data

7.1 Introduction

Bibliography

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

Liyuan Wang, Xingxing Zhang, Hang Su, and Jun Zhu. A comprehensive survey of continual learning: Theory, method and application. *arXiv preprint arXiv:2302.00487*, 2023.