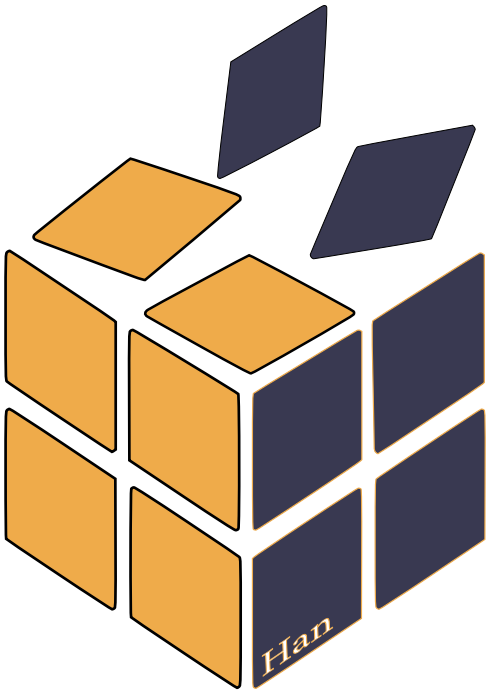
Deep Statistical Learning



Han Cheol Moon
School of Computer Science and Engineering
Nanyang Technological University
Singapore
hancheol001@e.ntu.edu.sg

Contents

Part I

Introduction

Chapter 1

Basics

1.1 Naive Bayes

Let's say we have a prediction model and it gives us a prediction. We want to measure the precision of the prediction. For example, the prediction has 90% chance of being a correct prediction. This can be modeled probabilistically. Given data $\mathbf{x}$, we want to know the probability of it being y. Similarly, if we have N data, then with an i.i.d., assumption,

$$\prod_{i=1}^{N} p(y_i|\mathbf{x}_i).$$

What we want to is to build such a probabilistic model parameterized by some parameters.

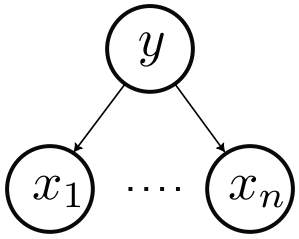
In this section, we discuss how to classify vectors of discrete-valued features $\mathbf{x}$. Recall that we discussed how to classify a feature vector $\mathbf{x}$ by applying Bayes rule to a generative classifier of the form

$$p(y = c|\mathbf{x}, \boldsymbol{\theta}) \propto p(\mathbf{x}|y = c, \boldsymbol{\theta})p(y = c|\boldsymbol{\theta})$$

The key to using such models is specifying a suitable form for the class-conditional density $p(\mathbf{x}|y = c, \boldsymbol{\theta})$, which defines what kind of data we expect to see in each class.

- $\mathbf{x} \in \{1, ..., K\}^D$,
  - K: the number of values for each feature.
  - D: the number of features.
- We will use a generative approach.
- Need to specify the class conditional distribution, $p(\mathbf{x}|y = c)$.
- A simple approach is to assume the features are **conditionally independence** given the class label.
- This allows us to write the class conditional density as a product of one dimensional densities:

$$p(\mathbf{x}|y = c, \boldsymbol{\theta}) = \prod_{j=1}^{D} p(x_j|y = c, \boldsymbol{\theta}_{jc})$$



The resulting model is called a **naive Bayes classifier (NBC)**. The model is called "naive" since we assume the independence between the features, which is not true in practice. However, if often results in classifiers that work well.

The form of the class-conditional density depends on the type of each feature. We give some possibilities below:

- In the case of real-valued features, we can use the Gaussian distribution: $p(\mathbf{x}|y=c,\boldsymbol{\theta}) = \prod_{j=1}^{D} \mathcal{N}(x_j|\mu_{jc}^2)$, where μ_{jc} is the mean of feature j in objects of class c, and σ_{jc}^2 is its variance.
- In the case of binary features, we can use the Bernoulli distribution: $p(\mathbf{x}|y=c,\boldsymbol{\theta}) = \prod_{j=1}^{D} \text{Ber}(x_j|\mu_{jc})$, where μ_{jc} is the probability that feature j occurs in class c. This is sometimes called the **multivariate Bernoulli naive Bayes** model.
- In the case of categorical features, $x_j \in \{1,...,K\}$, we can model the multinomial distribution: $p(\mathbf{x}|y=c,\boldsymbol{\theta}) = \prod_{j=1}^{D} \text{Cat}(x_j|\mu_{jc})$, where $\boldsymbol{\mu}_{jc}$ is a histogram over the K possible values for x_j in class c.

The probability for a single data case is given by

$$p(\mathbf{x}_i, y_i|\boldsymbol{\theta}) = p(y_i|\boldsymbol{\pi}) \prod_j p(x_{ij}|\boldsymbol{\theta}_j) = \prod_c \pi_c^{\mathbb{I}(y_i=c)} \prod_j \prod_c p(x_{ij}|\boldsymbol{\theta}_{jc})^{\mathbb{I}(y_i=c)},$$

where $\boldsymbol{\pi}$ is a vector of class probability. Hence the log-likelihood is given by

$$\log p(\mathcal{D}|\boldsymbol{\theta}) = \sum_{c=1}^{C} N_c \log \pi_c + \sum_{j=1}^{D} \sum_{c=1}^{C} \sum_{i:y_i=c} \log p(x_{ij}|\boldsymbol{\theta}_{jc})$$

Algorithm 1: Fitting a naive Bayes classifier to binary features

```
Initialize N_c = 0, N_jc = 0 ;
for i = 1 : N do
    c = y_i //Class label of i-th example;
    N_c := N_c + 1;
    for j = 1 : D do
        if x_ij = 1 then
            N_jc := N_jc + 1
    end
end
```

$\hat{\pi} = \frac{N_c}{N}, \hat{\theta}_{jc} = \frac{N_{jc}}{N_c}$

1.2 Decision Tree

In decision analysis, a decision tree can be used to visually and explicitly represent decisions and decision making. In data mining, a decision tree describes data (but the resulting classification tree can be an input for decision making).

- Classification trees: Tree models where the target variable take a discrete set of values; in these tree structures, leaves represent class labels and branches represent conjunctions of features that lead to those class labels.
- Regression trees: Decision trees where the target variable can take continuous values (typically real numbers).

The term Classification And Regression Tree (CART) analysis is an umbrella term used to refer to both them.

Some techniques, often called ensemble methods, construct more than one decision tree:

- Boosted tree: Incrementally building an ensemble by training each new instance to emphasize the training instances previously mis-modeled. A typical example is AdaBoost.
- Bootstrap aggregated (or bagged): decision trees, an early ensemble method, builds multiple decision trees by repeatedly resampling training data with replacement, and voting the trees for a consensus prediction, e.g., random forest.
- Rotation forest: in which every decision tree is trained by first applying principal component analysis (PCA) on a random subset of the input features

The basic algorithm used in decision trees is known as the ID3 (by Quinlan) algorithm. The ID3 algorithm builds decision trees using a top-down, greedy approach. Briefly, the steps to the algorithm are: Select the best attribute → Assign A as the decision attribute (test case) for the NODE. - For each value of A, create a new descendant of the NODE. - Sort the training examples to the appropriate descendant node leaf. - If examples are perfectly.

Now, the next big question is how to choose the best attribute. For ID3, we think of the best attribute in terms of which attribute has the most information gain, a measure that expresses how well an attribute splits that data into groups based on classification.

Pseudocode: ID3 is a greedy algorithm that grows the tree top-down, at each node selecting the attribute that best classifies the local training examples. This process continues until the tree perfectly classifies the training examples or until all attributes have been used.

The pseudocode assumes that the attributes are discrete and that the classification is binary. Examples are the training example. Target attribute is the attribute whose value is to be predicted by the tree. Attributes is a list of other attributes that may be tested by the learned decision tree. Finally, it returns a decision tree that correctly classifies the given Examples.

1.2.1 Information Gain

Information gain is a statistical property that measures how well a given attribute separates the training examples according to their target classification. As you can see in the Fig. , attribute

with low information gain (right) splits the data relatively evenly and as a result doesn't bring us any closer to a decision. Whereas, an attribute with high information gain (left) splits the data into groups with an uneven number of positives and negatives and as a result helps in separating the two from each other.

To define information gain precisely, we need to define a measure commonly used in information theory called entropy that measures the level of impurity in a group of examples. Mathematically, it is defined as:

$$Entropy = -\sum_i p_i \log p_i$$

, where i is the class index. Since, the basic version of the ID3 algorithm deal with the case where classification are either positive or negative, we can define entropy as:

$$Entropy(S) = -p_+ \log_2 p_+ - p_- \log_2 p_-$$

, where

- S: training examples
- p_+: is the proportion of positive examples in S
- p_-: is the proportion of negative examples in S

To illustrate, suppose S is a sample containing 14 boolean examples, with 9 positive and 5 negative examples. Then, the entropy of S relative to this boolean classification is:

$$Entropy([9+, 5-]) = -(9/14) \cdot \log_2(9/14) - (5/14) \cdot \log_2(5/14) = 0.940$$

Note that entropy is 0 if all the members of S belong to the same class. For example, if all members are positive (p_+=1), then $p_- = 0$, and $Entropy(S) = -1 \cdot \log_2(1) - 0 \cdot \log_2(0) = 0$. Entropy is 1 when the sample contains an equal number of positive and negative examples. If the sample contains unequal number of positive and negative examples, entropy is between 0 and 1. The following figure shows the form of the entropy function relative to a boolean classification as p_+ varies between 0 and 1.

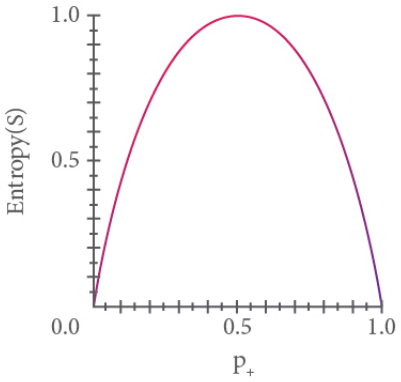


Figure 1.1: Entropy.

Now, given entropy as a measure of the impurity in a sample of training examples, we can now define information gain as a measure of the effectiveness of an attribute in classifying the training

data. Information gain, $Gain(S, A)$ of an attribute A, relative to a sample of examples S, is defined as:

$$Gain(S, A) \equiv Entrpoy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} \cdot Entropy(S_v)$$

, where $|S_v|$ is a sample belongs to class v and $|S|$ is the number of training samples. In other words,

Gain = Parent node of Entropy - [Average of Children Nodes Entropy]

Chapter 2

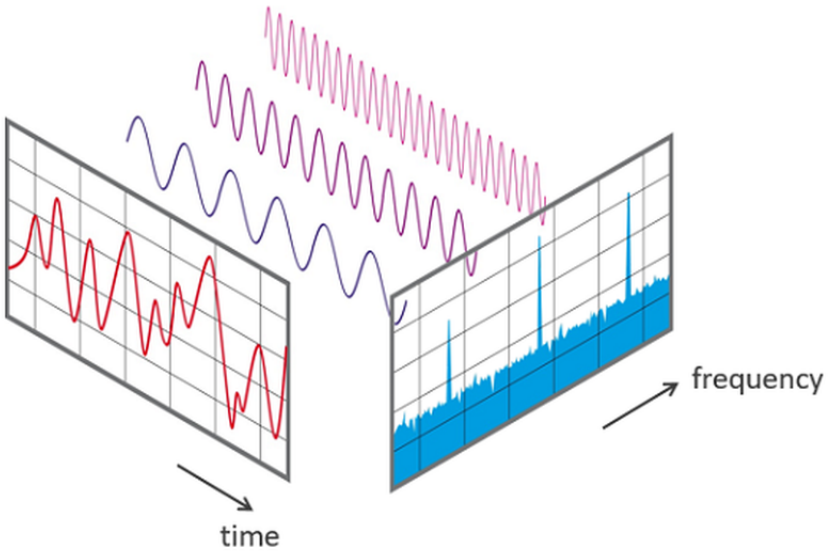
Fourier Analysis

Fourier analysis is a mathematical tool used to understand and analyze periodic phenomena. It is named after the French mathematician Joseph Fourier, it's based on the idea that any periodic function can be represented as a sum of sine and cosine functions with different frequencies, amplitudes, and phases.

The Fourier transform is one of the most important mathematical tools used for analyzing functions. Given an arbitrary function $f(x)$, with a real domain ($x \in \mathbb{R}$), we can express it as a linear combination of complex waves ($\approx$ change of basis to a frequency domain).

- Periodic Functions: Fourier analysis deals with functions that repeat over regular intervals. This repetition could be over time (like sound waves or electrical signals) or space (like patterns on a surface).
- Fourier Series: It states that any periodic function can be represented as an infinite sum of sine and cosine functions. The Fourier series expansion expresses the original function as a weighted sum of sinusoids.
- Frequency Domain: While the original function is in the time or spatial domain, Fourier analysis allows us to examine it in the frequency domain. This means we can understand the different frequency components present in the signal and their respective strengths.
- Fast Fourier Transform (FFT): This is an efficient algorithm for calculating the Fourier transform of a discrete signal. It's widely used in digital signal processing due to its speed and accuracy.
- Continuous vs. Discrete Fourier Transform: While Fourier series deals with periodic continuous functions, Fourier transforms extend the concept to non-periodic functions or signals. The discrete Fourier transform (DFT) is used for discrete, sampled signals, as in digital signal processing.
- Inverse Fourier Transform: Just as you can transform a function into the frequency domain using Fourier analysis, you can also transform it back using the inverse Fourier transform. This allows you to reconstruct the original function from its frequency components.

Very intuitive explanation is here!



2.1 Fourier Series

We begin by discussing the Fourier series, which is used to analyze functions that are periodic in their inputs. A periodic function $f(x)$ is a function of a real variable x that repeats itself every time x changes by T. The constant T is called the *period*. We can write the periodicity condition as

$$f(x+T) = f(x), \forall x \in \mathbb{R}.$$

The value of $f(x)$ can be real or complex, but x should be real.

Let's consider what it means to specify a periodic function $f(x)$. One way to specify the function is to give an explicit mathematical formula for it. Another approach might be to specify the function values in $-T/2 \leq x < T/2$. Since there's an uncountably infinite number of points in this domain, we can generally only achieve an approximate specification of f this way, by giving the values of f at a large but finite set x points.

There is another interesting approach to specifying f. We can express it as a linear combination of simpler periodic functions, consisting of sines and cosines:

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos\left(\frac{2\pi n x}{T}\right) + \sum_{m=1}^{\infty} b_m \sin\left(\frac{2\pi m x}{T}\right), \tag{2.1}$$

where a_0, a_n, and b_m are coefficients determined by integrating $f(x)$ over one period. The coefficients are given by

$$\begin{aligned} a_0 &= \frac{1}{T}\int_0^T f(x)dx \\ a_n &= \frac{2}{T}\int_0^T f(x)\cos\left(\frac{2\pi n x}{T}\right)dx \\ b_m &= \frac{2}{T}\int_0^T f(x)\sin\left(\frac{2\pi n x}{T}\right)dx \end{aligned}$$

The above formula is called a *Fourier series*. Given the numbers $\{a_n, b_m\}$, which are called the *Fourier coefficients*, $f(x)$ can be calculated for any x. The Fourier coefficients are real if $f(x)$ is a real function, or complex if $f(x)$ is complex.

Example: Consider a periodic function $f(x)$ defined on the interval $-\pi \leq x \leq \pi$ as follows:

$$f(x) = \begin{cases} 0 & \text{if } -\pi \leq x < 0 \\ x & \text{if } 0 \leq x \leq \pi \end{cases}$$

We want to find the Fourier series representation of $f(x)$.

Solution: Step 1: Determine the coefficients a_0, a_n, and b_n.

1. Calculate a_0:

$$\begin{aligned} a_0 &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) dx \\ &= \frac{1}{\pi} \left(\int_{-\pi}^{0} 0 \, dx + \int_{0}^{\pi} x \, dx \right) \\ &= \frac{1}{\pi} \left(\int_{0}^{\pi} x \, dx \right) \\ &= \frac{1}{\pi} \left(\frac{1}{2} \pi^2 \right) \\ &= \frac{\pi}{2} \end{aligned}$$

2. Calculate a_n:

$$a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \cos(nx) \, dx$$

Since $f(x)$ is odd and $\cos(nx)$ is even, a_n will be zero for all n.

3. Calculate b_n:

$$\begin{aligned} b_n &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \sin(nx) \, dx \\ &= \frac{1}{\pi} \left(\int_{0}^{\pi} x \sin(nx) \, dx \right) \end{aligned}$$

This integral can be evaluated using integration by parts:

$$u = x, \quad dv = \sin(nx) \, dx$$

$$du = dx, \quad v = -\frac{1}{n} \cos(nx)$$

$$\begin{aligned} b_n &= \frac{1}{\pi} \left(\left[-\frac{x}{n} \cos(nx) \right]_0^{\pi} - \frac{1}{n} \int_{0}^{\pi} -\cos(nx) \, dx \right) \\ &= \frac{1}{\pi} \left(\frac{\pi}{n} - \frac{1}{n^2} \left[\sin(nx) \right]_0^{\pi} \right) \\ &= \frac{1}{\pi} \left(\frac{\pi}{n} \right) \\ &= \frac{1}{n} \end{aligned}$$

4. Write the Fourier series:

$$f(x) = \frac{\pi}{2} + \sum_{n=1}^{\infty} \frac{1}{n} \sin(nx)$$

2.1.1 Square-integrable functions

Can arbitrary periodic functions always be expressed as a Fourier series? It turns out that a certain class of periodic functions, commonly encountered in physical contexts, are guaranteed to always be expressible as Fourier series. These are called square-integrable functions such that the integral of the square of the absolute value is finite:

$$\int_{-a/2}^{a/2} |f(x)|^2 dx < \infty.$$

Unless otherwise stated, we will always assume that the functions we're dealing with are square-integrable.

2.1.2 Complex Fourier series and inverse relations

We have written the Fourier series as a sum of sine and cosine functions. However, sines and cosines can be expressed by exponential functions by using *Euler's formula.*

$$e^{ix} = \cos x + i \sin x \tag{2.2}$$

- $\cos x = \frac{e^{ix}+e^{-ix}}{2}$
- $\sin x = \frac{e^{ix}-e^{-ix}}{2i}$

Thus, Fourier series can be expressed as follows:

$$f(x) = \sum_{n=-\infty}^{\infty} c_n e^{inx},$$

where the complex Fourier coefficients c_n are given by:

$$c_n = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x) e^{-inx}\, dx$$

- i: complex number
- n: integer

Example: Consider the periodic function $f(x)$ defined on the interval $-\pi \leq x \leq \pi$:

$$f(x) = \begin{cases} 0 & \text{if } -\pi \leq x < 0 \\ x & \text{if } 0 \leq x \leq \pi \end{cases}$$

We will find the complex form of the Fourier series for this function.

Step 1: Calculate the complex Fourier coefficients c_n.

$$c_n = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x) e^{-inx}\, dx$$

Since $f(x)$ is piecewise, we split the integral into two parts:

$$c_n = \frac{1}{2\pi}\left(\int_{-\pi}^{0} 0 \cdot e^{-inx}\,dx + \int_{0}^{\pi} xe^{-inx}\,dx\right)$$

The first integral is zero, so we focus on the second part:

$$c_n = \frac{1}{2\pi}\int_{0}^{\pi} xe^{-inx}\,dx$$

This integral can be solved using integration by parts. Let:

$$u = x \quad \text{and} \quad dv = e^{-inx}\,dx$$

Then:

$$du = dx \quad \text{and} \quad v = \frac{e^{-inx}}{-in} = -\frac{1}{in}e^{-inx}$$

Applying integration by parts:

$$\int u\,dv = uv - \int v\,du$$

So,

$$\int_{0}^{\pi} xe^{-inx}\,dx = \left[-\frac{x}{in}e^{-inx}\right]_{0}^{\pi} + \frac{1}{in}\int_{0}^{\pi} e^{-inx}\,dx$$

Evaluating the boundary terms:

$$\left[-\frac{x}{in}e^{-inx}\right]_{0}^{\pi} = -\frac{\pi}{in}e^{-in\pi} - 0 = -\frac{\pi}{in}e^{-in\pi}$$

And for the integral:

$$\frac{1}{in}\int_{0}^{\pi} e^{-inx}\,dx = \frac{1}{in}\left[\frac{e^{-inx}}{-in}\right]_{0}^{\pi} = -\frac{1}{n^2}\left(e^{-in\pi} - 1\right)$$

Combining these results:

$$\int_{0}^{\pi} xe^{-inx}\,dx = -\frac{\pi}{in}e^{-in\pi} + \frac{1}{n^2}(1 - e^{-in\pi})$$

Thus, the Fourier coefficient c_n is:

$$c_n = \frac{1}{2\pi}\left(-\frac{\pi}{in}e^{-in\pi} + \frac{1}{n^2}(1 - e^{-in\pi})\right)$$
$$c_n = \frac{1}{2\pi}\left(-\frac{\pi}{in}(-1)^n + \frac{1}{n^2}(1 - (-1)^n)\right)$$

This expression simplifies to:

$$c_n = \frac{i(-1)^n}{2n} + \frac{1}{2\pi n^2}(1 - (-1)^n)$$

Step 2: Write the Complex Fourier Series:

$$f(x) = \sum_{n=-\infty}^{\infty} c_n e^{inx}$$

With c_n now calculated, you can substitute these coefficients back into the Fourier series expression to represent $f(x)$.

2.2 Fourier Transform

2.2.1 Fourier Series v.s. Fourier Transform

The Fourier series is used to represent periodic functions as a sum of sine and cosine functions. On the other hand, the Fourier transform extends the idea to non-periodic functions or signals. The Fourier transform of a function $f(t)$ is given by:

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-i\omega t}\,dt,$$

where $F(\omega)$ is the frequency domain representation of the signal $f(t)$ and $\omega = 2\pi f$, where f is a frequency.

2.2.2 Equations of Fourier Transform

The Fourier transform and its inverse are defined as follows:

- Fourier Transform:
$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-i\omega t}\,dt$$
- Inverse Fourier Transform:
$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)e^{i\omega t}\,d\omega$$

These equations show how a time-domain signal $f(t)$ can be transformed into its frequency-domain representation $F(\omega)$, and vice versa.

Example: The Fourier transform of a rectangular pulse. Define a rectangular pulse function $f(t)$ as:

$$f(t) = \begin{cases} 1 & \text{if } |t| \leq \frac{1}{2} \\ 0 & \text{otherwise} \end{cases}$$

To find its Fourier transform $F(\omega)$:

$$F(\omega) = \int_{-\infty}^{\infty} f(t) e^{-i\omega t}\, dt$$

Since $f(t)$ is non-zero only between $-\frac{1}{2}$ and $\frac{1}{2}$, the integral becomes:

$$F(\omega) = \int_{-\frac{1}{2}}^{\frac{1}{2}} e^{-i\omega t}\, dt$$

This integral can be evaluated as:

$$F(\omega) = \left[\frac{e^{-i\omega t}}{-i\omega}\right]_{-\frac{1}{2}}^{\frac{1}{2}} = \frac{e^{-i\omega/2} - e^{i\omega/2}}{-i\omega} = \frac{2\sin(\omega/2)}{\omega}$$

Thus, the Fourier transform of the rectangular pulse is:

$$F(\omega) = \frac{\sin(\omega/2)}{\omega/2} = \operatorname{sinc}\left(\frac{\omega}{2}\right)$$

where $\operatorname{sinc}(x) = \frac{\sin(x)}{x}$.

2.3 Fourier Transform as a Change of Basis

A change of basis in linear algebra is a fundamental concept that involves transitioning from one coordinate system to another within a vector space. This can simplify calculations, provide deeper insights, or align with different geometric interpretations. Let's break down the concept with definitions and steps:

Fourier analysis is closely related to the concept of change of basis in linear algebra. In Fourier analysis, we are essentially changing the basis from the standard basis of functions (e.g., the time domain for signals) to a basis of sinusoidal functions (*e.g.,* the frequency domain). Here's how this relationship works:

The Fourier transform (both continuous and discrete versions) can be seen as a change of basis from the time domain to the frequency domain. When we apply the Fourier transform to a signal x(t)x(t), we express it as a linear combination of sinusoidal basis functions.

- Standard Basis in Time Domain: In the time domain, a signal or function can be represented as a sum of basis functions, which are typically δ functions (impulse functions) or other

simple functions like polynomial terms. For example, a discrete-time signal $x[n]$ can be considered in terms of standard basis vectors.

- Fourier Basis in Frequency Domain: In the frequency domain, the basis functions are complex exponentials or sinusoidal functions $e^{i\omega t}$ (where i is the imaginary unit and ω is the angular frequency). The Fourier transform decomposes a signal into a sum of these sinusoidal basis functions.

The Fourier transform (both continuous and discrete versions) can be seen as a change of basis from the time domain to the frequency domain. When we apply the Fourier transform to a signal $x(t)$, we express it as a linear combination of sinusoidal basis functions.

Continuous Fourier Transform: The continuous Fourier transform of a function $x(t)$ is given by:

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-i2\pi ft}\,dt$$

Here, $X(f)$ represents the coordinates of the function $x(t)$ in the new basis (frequency domain). The inverse Fourier transform reconstructs the original signal from its frequency domain representation:

$$x(t) = \int_{-\infty}^{\infty} X(f)e^{i2\pi ft}\,df$$

This process is analogous to converting coordinates from the new basis back to the original basis.

2.3.1 Change of Basis

- Vector Space: A collection of vectors where vector addition and scalar multiplication are defined and satisfy specific axioms.
- Basis: A set of linearly independent vectors that span the entire vector space. Every vector in the space can be uniquely represented as a linear combination of the basis vectors.
- Coordinate Vector: The representation of a vector in terms of the basis vectors, typically given as a column of coefficients.

Steps for Change of Basis:

1. Representing Vectors in Original Basis: Given a vector space V with a basis $B = \{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n\}$, any vector $\mathbf{v}$ in V can be written as:

   $$\mathbf{v} = c_1\mathbf{b}_1 + c_2\mathbf{b}_2 + \cdots + c_n\mathbf{b}_n$$

   Here, $c_1, c_2, \ldots, c_n$ are the coordinates of $\mathbf{v}$ relative to basis B, often written as $[\mathbf{v}]_B$.

2. Defining the New Basis: Suppose we want to change to a new basis $B' = \{\mathbf{b}'_1, \mathbf{b}'_2, \ldots, \mathbf{b}'_n\}$. We need to express the original basis vectors in terms of the new basis vectors. Let:

   $$\mathbf{b}_i = a_{i1}\mathbf{b}'_1 + a_{i2}\mathbf{b}'_2 + \cdots + a_{in}\mathbf{b}'_n$$

   for $i = 1, 2, \ldots, n$.

3. Constructing the Change of Basis Matrix: The coefficients a_{ij} form the columns of the change of basis matrix P from B to B':

$$P = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

4. Converting Coordinates: To convert the coordinates of a vector $\mathbf{v}$ from the original basis B to the new basis B', we use the change of basis matrix P:

$$[\mathbf{v}]_{B'} = P^{-1}[\mathbf{v}]_B$$

where P^{-1} is the inverse of the change of basis matrix and $[\mathbf{v}]_B = (c_1, \ldots, c_n)$ is a coordinate vector.

Conversely, to convert coordinates from the new basis B' back to the original basis B, we use:

$$[\mathbf{v}]_B = P[\mathbf{v}]_{B'}$$

Example: Consider a vector space $\mathbb{R}^2$ with an original basis $B = \{\mathbf{b}_1, \mathbf{b}_2\}$ and a new basis $B' = \{\mathbf{b}'_1, \mathbf{b}'_2\}$. Let:

$$\mathbf{b}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \ \mathbf{b}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

and

$$\mathbf{b}'_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \ \mathbf{b}'_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

To find the change of basis matrix P from B to B', express the original basis vectors in terms of B':

$$\mathbf{b}_1 = \frac{1}{2}\mathbf{b}'_1 + \frac{1}{2}\mathbf{b}'_2$$

$$\mathbf{b}_2 = \frac{1}{2}\mathbf{b}'_1 - \frac{1}{2}\mathbf{b}'_2$$

So, the change of basis matrix P is:

$$P = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} \end{bmatrix}$$

To convert a vector $\mathbf{v}$ with coordinates $[\mathbf{v}]_B = \begin{bmatrix} x \\ y \end{bmatrix}$ to the new basis B', we compute:

$$[\mathbf{v}]_{B'} = P^{-1}[\mathbf{v}]_B$$

And to revert to the original basis:

$$[\mathbf{v}]_B = P[\mathbf{v}]_{B'}$$

Chapter 3

Training, Testing, and Regularization

3.1 Sources of Error in ML

$$E_{out} \leq E_{ml} + \Omega$$

- E_{out}: estimation of error.
- E_{ml}: error from a learning algorithm
- Ω: error caused by the variance from observations.

We also define

- f: target function
- g: learning function
- $g^{(D)}$: learned function based on D, or simply *hypothesis*.
- D: dataset drawn from the real world.
- $\bar{g}$: the average hypothesis of a given infinite number of Ds.

$$\bar{g}(x) = \mathbb{E}_D[g^{(D)}(x)].$$

Error of a single instance x from g learnt from D is given by

$$Err_{\text{out}}(g^{(D)}(x)) = \mathbb{E}_X[(g^{(D)}(x) - f(x))^2],$$

where X can be considered as test sets. Then, the expected error over the infinite number of datasets D sampled from a true data distribution is

$$\begin{aligned}\mathbb{E}_D[Err_{\text{out}}(g^{(D)}(x))] &= \mathbb{E}_D[\mathbb{E}_X[(g^{(D)}(x) - f(x))^2]] \\ &= \mathbb{E}_X[\mathbb{E}_D[(g^{(D)}(x) - f(x))^2]]\end{aligned}$$

Let's simplify the term inside with an average of hypothesis $\bar{g}(x)$:

$$\begin{aligned}\mathbb{E}_D[(g^{(D)}(x) - f(x))^2] &= \mathbb{E}_D[(g^{(D)}(x) - \bar{g}(x) + \bar{g}(x) - f(x))^2] \\ &= \mathbb{E}_D\big[(g^{(D)}(x) - \bar{g}(x))^2 + (\bar{g}(x) - f(x))^2 \\ &\quad + 2(g^{(D)}(x) - \bar{g}(x))(\bar{g}(x) - f(x))\big] \\ &= \mathbb{E}_D\big[(g^{(D)}(x) - \bar{g}(x))^2\big] + (\bar{g}(x) - f(x))^2 \\ &\quad + \mathbb{E}_D\big[2(g^{(D)}(x) - \bar{g}(x))(\bar{g}(x) - f(x))\big]\end{aligned}$$

Since, $\mathbb{E}_D\big[2(g^{(D)}(x) - \bar{g}(x))(\bar{g}(x) - f(x))\big]$ is 0, the expectation of the error becomes

$$\mathbb{E}_D[Err_{\text{out}}(g^{(D)}(x))] = \mathbb{E}_X\big[\mathbb{E}_D\big[(g^{(D)}(x) - \bar{g}(x))^2\big] + (\bar{g}(x) - f(x))^2\big].$$

Let's closely look at this formula. The errors are from two sources:

- **Variance**: $\mathbb{E}_D\big[(g^{(D)}(x) - \bar{g}(x))^2\big]$. Variance captures how much your classifier changes if you train on a different training set. We need to collect more data to reduce the variance.
- **Bias**: $(\bar{g}(x) - f(x))^2$. Bias is the inherent error that you obtain from your classifier even with infinite training data. We need to build a more complex model to reduce the bias.

However, if we reduce the bias, then the variance tends to increase.

3.1.1 Alternative Derivation

The derivation of the bias–variance decomposition for squared error proceeds as follows.[6][7] For notational convenience, abbreviate $f = f(x)$ and $\hat{f} = \hat{f}(x)$. First, recall that, by definition, for any random variable $\mathbf{X}$, we have

$$\operatorname{Var}\big[\hat{f}(x)\big] = \mathrm{E}[X^2] - \mathrm{E}[X]^2.$$

By rearranging, we get

$$\mathrm{E}[X^2] = \operatorname{Var}\big[\hat{f}(x)\big] + \mathrm{E}[X]^2.$$

Since f is deterministic

$$\mathrm{E}[f] = f$$

Thus, given $y = f + \varepsilon$ and $\mathrm{E}[\varepsilon] = 0$, implies $\mathrm{E}[y] = \mathrm{E}[f + \varepsilon] = \mathrm{E}[f] = f$

Also, since $\operatorname{Var}[\varepsilon] = \sigma^2$

$$\operatorname{Var}[y] = \mathrm{E}[(y - \mathrm{E}[y])^2] = \mathrm{E}[(y - f)^2] = \mathrm{E}[(f + \varepsilon - f)^2] = \mathrm{E}[\varepsilon^2] = \operatorname{Var}[\varepsilon] + \Big(\mathrm{E}[\varepsilon]\Big)^2 = \sigma^2$$

Thus, since ε and $\hat{f}$ are independent, we can write:

$$\begin{aligned}
\mathrm{E}\left[(y-\hat{f})^2\right] &= \mathrm{E}\left[(f+\varepsilon-\hat{f})^2\right] \\
&= \mathrm{E}\left[(f+\varepsilon-\hat{f}+\mathrm{E}[\hat{f}]-\mathrm{E}[\hat{f}])^2\right] \\
&= \mathrm{E}\left[(f-\mathrm{E}[\hat{f}])^2\right]+\mathrm{E}[\varepsilon^2]+\mathrm{E}\left[(\mathrm{E}[\hat{f}]-\hat{f})^2\right]+2\,\mathrm{E}\left[(f-\mathrm{E}[\hat{f}])\varepsilon\right]+ \\
&\quad 2\,\mathrm{E}\left[\varepsilon(\mathrm{E}[\hat{f}]-\hat{f})\right]+2\,\mathrm{E}\left[(\mathrm{E}[\hat{f}]-\hat{f})(f-\mathrm{E}[\hat{f}])\right] \\
&= (f-\mathrm{E}[\hat{f}])^2+\mathrm{E}[\varepsilon^2]+\mathrm{E}\left[(\mathrm{E}[\hat{f}]-\hat{f})^2\right]+ \\
&\quad 2(f-\mathrm{E}[\hat{f}])\,\mathrm{E}[\varepsilon]+2\,\mathrm{E}[\varepsilon]\,\mathrm{E}\left[\mathrm{E}[\hat{f}]-\hat{f}\right]+2\,\mathrm{E}\left[\mathrm{E}[\hat{f}]-\hat{f}\right](f-\mathrm{E}[\hat{f}]) \\
&= (f-\mathrm{E}[\hat{f}])^2+\mathrm{E}[\varepsilon^2]+\mathrm{E}\left[(\mathrm{E}[\hat{f}]-\hat{f})^2\right] \\
&= (f-\mathrm{E}[\hat{f}])^2+\mathrm{Var}[y]+\mathrm{Var}\left[\hat{f}\right] \\
&= \mathrm{Bias}[\hat{f}]^2+\mathrm{Var}[y]+\mathrm{Var}\left[\hat{f}\right] \\
&= \mathrm{Bias}[\hat{f}]^2+\sigma^2+\mathrm{Var}\left[\hat{f}\right]
\end{aligned}$$

Chapter 4

Optimization

4.1 Intuition of Gradient

Gradient descent is an optimization algorithm used to minimize a function by iteratively moving towards the function's minimum value. It is a fundamental concept in machine learning, particularly in training models such as neural networks. The gradient is a vector that represents the direction of the steepest increase of the function at a given point. For example, for a convex function $z = ax^2 + by^2$, the gradient is $[2ax, 2by]$, which points in the direction of the steepest ascent.

In gradient descent, the goal is to minimize the function, so the algorithm moves in the opposite direction of the gradient, which is $[-2ax, -2by]$. This opposite direction is chosen because it is the direction of the steepest decrease in the function value. But how do we know that moving in this direction will strictly decrease the function value?

4.1.1 Direction of Gradient Descent

Let's investigate the direction of gradient descent.

- The derivative of the objective function $f(\mathbf{x})$ provides the slope of $f(\mathbf{x})$ at the point $f(\mathbf{x})$.
- It tells us how to change $\mathbf{x}$ in order to make a small improvement in our goal.

A function $f(\mathbf{x})$ can be approximated by its first-order Taylor expansion at $\bar{\mathbf{x}}$:

$$f(\mathbf{x}) \approx f(\bar{\mathbf{x}}) + \nabla f(\bar{\mathbf{x}})^T (x - \bar{\mathbf{x}})$$

Now let $\mathbf{d} \neq 0, \|\mathbf{d}\| = 1$ be a direction, and in consideration of a new point $\mathbf{x} := \bar{\mathbf{x}} + \mathbf{d}$, we define:

$$f(\bar{\mathbf{x}} + \mathbf{d}) \approx f(\bar{\mathbf{x}}) + \nabla f(\bar{\mathbf{x}})^T \mathbf{d}$$

We would like to choose $\mathbf{d}$ that minimizes the function f. From the Cauchy-Schwarz inequality [1], we know that

$$|\nabla f(\bar{\mathbf{x}})^T \mathbf{d}| \leq \|\nabla f(\bar{\mathbf{x}})\| \ \|\mathbf{d}\|.$$

[1] Cauchy-Schwarz Inequaility: $|\mathbf{a} \cdot \mathbf{b}| \leq \|\mathbf{a}\| \ \|\mathbf{b}\|$. Equality holds if and only if either $\mathbf{a}$ or $\mathbf{b}$ is a multiple of the other.

The equality holds if and only if $\mathbf{d} = \lambda \nabla f(\bar{\mathbf{x}})$, where $\lambda \in \mathbb{R}$. Since we want to minimize the function f, we negate the steepest direction $\mathbf{d}^*$, then

$$f(\bar{\mathbf{x}} + \mathbf{d}) \approx f(\bar{\mathbf{x}}) - \lambda \nabla f(\bar{\mathbf{x}})^T \nabla f(\bar{\mathbf{x}}).$$

Since $\nabla f(\bar{\mathbf{x}})^T \nabla f(\bar{\mathbf{x}})$ is **always positive**, the term $-\lambda \nabla f(\bar{\mathbf{x}})^T \nabla f(\bar{\mathbf{x}})$ is always negative. Therefore by updating $\mathbf{x}$

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

we get

$$f(\mathbf{x}^{(k+1)}) < f(\mathbf{x}^{(k)}).$$

4.2 Normalized Gradient Descent

The underlying issue of the vanila gradient descent is the presence of saddle points in nonconvex functions; the gradient $\nabla f(x)$ vanishes near saddle points, which causes GD to "stall" in neighboring regions. This both slows the overall convergence rate and makes detection of local minima difficult. The detrimental effects of this issue become particularly severe in high-dimensional problems where the number of saddle points may proliferate.

However, in the normalized gradient descent

$$\frac{\nabla f(x)}{\|\nabla f(x)\|}$$

The normalized gradient preserves the direction of the gradient but ignores magnitude, because the normalization does not vanish near saddle points, the intuitive expectation is that NGD should not slow down in the neighborhood of saddle points and should therefore escape quickly.

4.3 Projected Gradient Descent

Gradient Descent (GD) is a standard way to solve unconstrained optimization problem. Starting from an initial point $x \in \mathbb{R}^n$, GD itereates until a stopping criterion is met. Projected Gradient Descent (PGD) is a way to solve constrained optimization problem. Consider a constraint set $\mathcal{Q}$, starting from a initial point $x_0 \in \mathcal{Q}$, PGD iterates the following equation until a stopping condition is met:

$$x_{k+1} = P_{\mathcal{Q}}\Big(x_k - t_k \nabla f(x_k)\Big),$$

where $P_{\mathcal{Q}}$ is the projection operator

$$P_{\mathcal{Q}}(x_0) = \underset{x \in \mathcal{Q}}{\operatorname{argmin}} \frac{1}{2}\|x - x_0\|_2^2$$

In other words, given a point x_0, $P_{\mathcal{Q}}$ tries to to find a point $x \in \mathcal{Q}$ which is "closest"to x_0.

Note that a vector projection can be expressed as follows:

$$a_1 = \|\mathbf{a}\| \cos\theta = \mathbf{a} \cdot \hat{\mathbf{b}} = \mathbf{a} \cdot \frac{\mathbf{b}}{\|\mathbf{b}\|}$$

Thus, a projection for unit L_2 ball is given by the solution of the equation as follows:

$$\mathbf{x} = \mathcal{P}_{\|x\|_2 \leq 1}(\mathbf{y})$$

The solution is

$$\mathbf{x} = \frac{\mathbf{y}}{\max\{1, \|\mathbf{y}\|_2\}}$$

The "geometric" proof is given as follows: Let $\mathcal{S} = \{\mathbf{x} \in \mathbb{R}^n : \|\mathbf{x}\|_2 \leq 1\}$.

- If $\mathbf{y} \in \mathcal{S}$, then $\|y\|_2 \leq 1$ and $\mathbf{y}$ itself is the closest point to $\mathbf{y}$.
- If $\mathbf{y} \notin \mathcal{S}$, then $\|y\|_2 > 1$ and the closest point $\mathbf{x} \in \mathcal{S}$ to $\mathbf{y}$ will be simply $\frac{\mathbf{y}}{\|\mathbf{y}\|_2}$ as the norm of $\frac{\mathbf{y}}{\|\mathbf{y}\|_2} = 1$.

By combining the bost cases, we have

$$\mathbf{x} = \frac{\mathbf{y}}{\max\{1, \|\mathbf{y}\|_2\}}$$

4.4 Exponentially Weighted Average

$$v_t = \beta v_{t-1} + (1 - \beta)\theta_t$$

Larger β value covers more longer history. EMA is exponentially weighted average the previous result.

4.5 Bias Correction

The initial values of v_t will be very low which need to be compensated, since the curve starts from 0, there are not many values to average on in the initial points. Thus, the curve is lower than the correct value initially and then moves in line with expected values. The β is the same as the averaging coefficient. As t becomes large, the impact of the bias correction will be decreased.

$$v_t = \frac{v_t}{1 - \beta^t}$$

4.6 Momentum

Momentum can reduce the oscillation in the gradients. Let's say w has a small value and b is in charge of oscillation. Then momentum can cancel out db by averaging them.

$$\begin{aligned}
v_{dw} &= \beta v_{dw} + (1 - \beta)dw \\
v_{db} &= \beta v_{db} + (1 - \beta)db \\
w &= w - \alpha v_{dw} \\
b &= b - \alpha v_{db}
\end{aligned}$$

4.7 Adagrad: Adaptive Gradient

$$v_{dw} = v_{dw} + dw \cdot dw$$
$$w = w - \frac{\alpha}{\sqrt{v_{dw}} + \epsilon} v_{dw}$$

A con of Adagrad is learning rate will become very small

4.8 RMS Prop

$$s_{dw} = \beta s_{dw} + (1 - \beta) dw^2$$
$$s_{db} = \beta s_{db} + (1 - \beta) db^2$$
$$w = w - \alpha \frac{dw}{\sqrt{s_{dw}}}$$
$$b = b - \alpha \frac{db}{\sqrt{s_{db}}}$$

4.9 ADAM

Its name is derived from adaptive moment estimation, and the reason it's called that is because Adam uses estimations of first and second moments of gradient to adapt the learning rate for each weight of the neural network. N-th moment of a random variable is defined as the expected value of that variable to the power of n. More formally:

$$m_n = \mathbb{E}[X^n]$$

To estimates the moments, Adam utilizes exponentially moving averages, computed on the gradient evaluated on a current mini-batch:

Since m and v are estimates of first and second moments, we want to have the following property:

$$\mathbb{E}[m_t] = \mathbb{E}[g_t] \tag{4.1}$$
$$\mathbb{E}[v_t] = \mathbb{E}[g_t^2] \tag{4.2}$$

Unbiased estimators

ADAM uses both momentum style and RMS prop style averaging.

- $v_{dw} = \beta v_{dw} + (1 - \beta) dw$
- $v_{db} = \beta v_{db} + (1 - \beta) db$
- $s_{dw} = \beta s_{dw} + (1 - \beta) dw^2$

- $s_{db} = \beta s_{db} + (1 - \beta) db^2$

Using them,

- $v_{dw}^{\text{corr}} = \frac{v_{dw}}{1-\beta_1^t}$
- $v_{db}^{\text{corr}} = \frac{v_{db}}{1-\beta_1^t}$
- $s_{dw}^{\text{corr}} = \frac{s_{dw}}{1-\beta_2^t}$
- $s_{db}^{\text{corr}} = \frac{s_{db}}{1-\beta_2^t}$

Finally,

$$w = w - \alpha \frac{v_{dw}^{\text{corr}}}{\sqrt{s_{dw}^{\text{corr}}} + \varepsilon}$$
$$b = b - \alpha \frac{v_{db}^{\text{corr}}}{\sqrt{s_{db}^{\text{corr}}} + \varepsilon}$$

4.10 Principal Component Analysis

4.10.1 Covariance and the weight vector

When deriving PCA, we seek a vector $\mathbf{w}$ (the weight vector or loading vector) such that the projection of the data onto this vector maximizes the variance. Maximizing the variance is equivalent to projecting the data onto a lower-dimensional linear subspace in such a way that the distance between a vector and its projection is not too large. For a given data matrix $\mathbf{X}$ with mean zero (mean-centered data), the projection of the data onto $\mathbf{w}$ is given by $\mathbf{Xw}$.

The variance of the projected data can be expressed as:

$$\text{Var}(\mathbf{Xw}) = \frac{1}{n}(\mathbf{Xw})^T(\mathbf{Xw}) = \frac{1}{n}\mathbf{w}^T\mathbf{X}^T\mathbf{Xw}$$

Where n is the number of data points. The matrix $\mathbf{X}^T\mathbf{X}$ is the covariance matrix of the data (up to a scaling factor).

The goal is to maximize the variance $\mathbf{w}^T\mathbf{X}^T\mathbf{Xw}$ with respect to the weight vector $\mathbf{w}$, subject to the constraint that $\mathbf{w}^T\mathbf{w} = 1$ (to prevent the trivial solution where the variance could be made arbitrarily large just by scaling $\mathbf{w}$).

$$\mathcal{L} = \frac{1}{n}\mathbf{w}^\mathsf{T}\mathbf{X}^\mathsf{T}\mathbf{Xw} - \lambda\left(\mathbf{w}^\mathsf{T}\mathbf{w} - 1\right)$$

$$\frac{\partial\mathcal{L}}{\partial\mathbf{w}} = \frac{2}{n}\mathbf{X}^\mathsf{T}\mathbf{Xw} - 2\lambda\mathbf{w} = \mathbf{0}$$

$$\underbrace{\frac{1}{n}\mathbf{X}^\mathsf{T}\mathbf{X}}_{:=\mathbf{S}}\mathbf{w} = \lambda\mathbf{w} \quad \Rightarrow \quad \mathbf{Sw} = \lambda\mathbf{w}$$

This is exactly the eigenvalue equation. The eigenvectors $\mathbf{w}$ are the directions that maximize the variance, and the eigenvalues λ represent the magnitude of the variance along those directions.

- Eigenvectors: Each eigenvector of the covariance matrix represents a direction in the feature space. These directions are the principal components.
- Eigenvalues: The corresponding eigenvalue tells us how much variance is captured along that direction. The larger the eigenvalue, the more variance is captured by the corresponding eigenvector.

$$\text{Var}(\mathbf{Xw}) = \frac{1}{n}(\mathbf{Xw})^T(\mathbf{Xw}) = \frac{1}{n}\mathbf{w}^T\mathbf{X}^T\mathbf{Xw} = \frac{1}{n}\mathbf{w}^T\mathbf{Sw} = \frac{1}{n}\mathbf{w}^T\lambda\mathbf{w} = \frac{1}{n}\lambda$$

Capturing the Most Information:

- Variance is a measure of how spread out the data is along a particular direction. By maximizing the variance, we ensure that the principal components capture the most significant patterns in the data.
- If we reduce the dimensionality by selecting components with the highest variance, we retain the most information about the data, effectively compressing the data without losing critical details.
- High variance indicates that the data points are spread out and less likely to be redundant. Conversely, low variance implies that data points are clustered close together, often making the information less significant.

Part II

Regression

Chapter 5

Introduction to Regression Methods

5.1 Introduction

Regression is a process for finding the relationship between the inputs and the outputs. In a regression problem, we consider a set of noisy measurement (or noisy output data) $\mathbf{y} = [y_1, \ldots, y_d]^T$ with measurement noise $\boldsymbol{\eta} = [\eta_1, \ldots, \eta_d]^T$. We also consider a set of input data $\mathbf{x} = [x_1, \ldots, x_d]$. We call the set of these input-output pairs $\mathcal{D} = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_m, \mathbf{y}_m)\}$ the training data. The true relationship between the input and the output data is unknown. We denote this unknown relationship as a mapping $f(\cdot)$ that takes $\mathbf{x}_n$ and maps it to y_n,

$$\mathbf{y} = f(\mathbf{x}).$$

Finding the true $f(\cdot)$ from a finite number of data points D is infeasible. There are infinitely many ways to design $f(\cdot)$ for every $\mathbf{x}_i$. The idea of regression is to add a structure to the problem. Instead of looking for the true $f(\cdot)$, we find a proxy $g_\theta(\cdot)$ that takes a certain parameters $\boldsymbol{\theta} = [\theta_1, \ldots, \theta_d]^T$. For instance, we can postulate that $(\mathbf{x}_n, \mathbf{y}_n)$ has a linear relationship:

$$g_{\boldsymbol{\theta}}(\mathbf{y}) = \mathbf{X}\boldsymbol{\theta} + \boldsymbol{\eta},$$

where $\mathbf{X}$ is a $m \times d$ input matrix (or our observations). Since we do not know the true relationship, any choice of model is our conjecture. However, we can model the error of our choice. Given a parameter $\boldsymbol{\theta}$, we consider the difference between the noisy measurements and estimated value as follows:

$$\boldsymbol{\epsilon} = \mathbf{y} - \mathbf{X}\boldsymbol{\theta}$$

The purpose of regression is to find the best $\boldsymbol{\theta}$ such that the error is minimized. Therefore, we can consider a following objective function:

$$J(\boldsymbol{\theta}) = \boldsymbol{\epsilon}^T \boldsymbol{\epsilon}$$

Note that this is equivalent to minimizing the mean squared error:

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - \mathbf{x}_i \boldsymbol{\theta})^2.$$

We can optimize this in a closed-form as follows:

$$\begin{aligned} J(\boldsymbol{\theta}) &= \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2 \\ &= (\mathbf{y} - \mathbf{X}\boldsymbol{\theta})^T (\mathbf{y} - \mathbf{X}\boldsymbol{\theta}) \\ &= (\mathbf{y}^T - \boldsymbol{\theta}^T \mathbf{X}^T)(\mathbf{y} - \mathbf{X}\boldsymbol{\theta}) \\ &= \mathbf{y}^T \mathbf{y} - \boldsymbol{\theta}^T \mathbf{X}^T \mathbf{y} - \mathbf{y}^T \mathbf{X}\boldsymbol{\theta} + \boldsymbol{\theta}^T \mathbf{X}^T \mathbf{X}\boldsymbol{\theta} \end{aligned}$$

To find the $\boldsymbol{\theta}$ that minimizes the objective function, we will compute a derivative of the function while setting it equal to zero:

$$\begin{aligned}\frac{\partial J}{\partial \boldsymbol{\theta}} &= -\mathbf{X}^T\mathbf{y} - \mathbf{X}^T\mathbf{y} + \mathbf{X}^T\mathbf{X}\boldsymbol{\theta} + \mathbf{X}^T\mathbf{X}\boldsymbol{\theta} = 0 \\ &\Rightarrow \mathbf{X}^T(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}) = 0 \\ \boldsymbol{\theta} &= (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}\end{aligned}$$

Note that the $\mathbf{X}^T(\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$ is called the *normal equation*.

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

N = 50
x = np.random.randn(N)
w_1 = 3.4 # True Parameter
w_0 = 0.9 # True Parameter
y = w_1*x + w_0 + 0.3*np.random.randn(N) # Synthesize training data

X = np.column_stack((x, np.ones(N)))
W = np.array([w_1, w_0])

# From Scratch
XtX    = np.dot(X.T, X)
XtXinvX = np.dot(np.linalg.inv(XtX), X.T) # d x m
W_best = np.dot(XtXinvX, y.T)
print(f"W_best: {W_best}")

# Pythonic Approach
theta = np.linalg.lstsq(X, y, rcond=None)[0]
print(f"Theta: {theta}")

t = np.linspace(0, 1, 200)
y_pred = W_best[0]*t+W_best[1]
yhat = theta[0]*t+theta[1]
plt.plot(x, y, 'o')
plt.plot(t, y_pred, 'r', linewidth=4)
plt.show()
```

5.1.1 MLE Interpretation

$$\mathcal{L}(\boldsymbol{\theta}; \mathcal{D}) = \prod_{t=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_t - \boldsymbol{\theta}\mathbf{x}_t)^2}{2\sigma^2}\right)$$

$$\log \mathcal{L}(\boldsymbol{\theta}; \mathcal{D}) = \text{const} - \frac{1}{2\sigma^2}\sum_{t=1}^{n}(y_t - \boldsymbol{\theta}\mathbf{x}_t)^2$$

5.1.2 Time Complexity

The time complexity of the training process involves the following steps:

- Computing X^TX: The product X^TX involves multiplying a $d \times m$ matrix with an $m \times d$ matrix. The time complexity of this matrix multiplication is $O(md^2)$.

- Computing $X^T\mathbf{y}$: $X^T\mathbf{y}$ involves multiplying a $d \times m$ matrix with an $m \times 1$ vector. - The time complexity is $O(md)$.
- Computing $\left(X^TX\right)^{-1}$: - The inversion of a $d \times d$ matrix X^TX has a time complexity of $O(d^3)$.
- Multiplying $\left(X^TX\right)^{-1}$ with $X^T\mathbf{y}$: This is a matrix-vector multiplication involving a $d \times d$ matrix and a $d \times 1$ vector. Thus, the time complexity is $O(d^2)$.
- Total Training Time Complexity: The dominant terms in the training process are $O(md^2)$ (for computing $X^\top X$) and $O(d^3)$ (for inverting X^TX). - Therefore, the total time complexity for training a linear regression model is $O(md^2 + d^3)$.

Inference time complexity:

- The inference step requires a matrix-vector multiplication between an $m' \times d$ inference data matrix and a $d \times 1$ vector. - The time complexity of this operation is $O(m'd)$.

5.2 Overdetermined and Underdetermined Systems

Recall that the linear regression problem is an optimization problem of finding the optimal parameter as follows:

$$\boldsymbol{\theta}_{opt} = \underset{\boldsymbol{\theta}\in\mathbb{R}^d}{\operatorname{argmin}} ||y - \mathbf{X}\boldsymbol{\theta}||^2.$$

We say the optimization problem is *overdetermined* if $\mathbf{X} \in \mathbb{R}^{m\times d}$ is tall and skinny, *i.e.,* $m > d$. This problem has a unique solution $\boldsymbol{\theta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$ if and only if $\mathbf{X}^T\mathbf{X}$ is invertible. Equivalently, $\mathbf{X}$ should be linearly independent (*i.e.,* full rank).

If $\mathbf{X}$ is fat and short (*i.e.,* $m < d$), a problem is called *underdetermined*. **This problem will have infinitely many solutions**. Among all the feasible solutions, we will pick the one that minimizes the squared norm. The solution is called the *minimum-norm* least squares. Consider an underdetermined linear regression problem:

$$\boldsymbol{\theta} = \underset{\boldsymbol{\theta}\in\mathbb{R}^d}{\operatorname{argmin}} \|\boldsymbol{\theta}\|^2, \text{ subject to } \mathbf{y} = \mathbf{X}\boldsymbol{\theta},$$

where $\mathbf{X} \in \mathbb{R}^{m\times d}, \boldsymbol{\theta} \in \mathbb{R}^d$, and $\mathbf{y} \in \mathbb{R}^m$. If the matrix has $\text{rank}(\mathbf{X}) = m$, then the linear regression problem will have a unique global minimum

$$\boldsymbol{\theta} = \mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{y}.$$

This solution is called the minimum-norm least-squares solution. The proof of this solution is given by:

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\lambda}) = \|\boldsymbol{\theta}\|^2 + \boldsymbol{\lambda}^T(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}),$$

where $\boldsymbol{\lambda}$ is a Lagrange multiplier. The solution of the constrained optimization is the stationary point of the Lagrangian. To find it, we take the derivatives w.r.t., $\boldsymbol{\lambda}$ and $\boldsymbol{\theta}$ as follows:

$$\nabla_{\boldsymbol{\theta}} = 2\boldsymbol{\theta} + \mathbf{X}^T\boldsymbol{\lambda} = 0$$
$$\nabla_{\boldsymbol{\lambda}} = \mathbf{X}\boldsymbol{\theta} - \mathbf{y} = 0$$

The first equation gives us $\boldsymbol{\theta} = -\mathbf{X}^T\boldsymbol{\lambda}/2$. Substituting it into the second equation, and assuming that $\text{rank}(\mathbf{X}) = m$ so that $\mathbf{X}^T\mathbf{X}$ is invertible, we have $\boldsymbol{\lambda} = -2(\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{y}$. Thus, we have

$$\boldsymbol{\theta} = \mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{y}.$$

Note that $\mathbf{X}\mathbf{X}^T$ is often called a *Gram matrix*, $\mathbf{G}$ (*c.f.,* **??**).

5.3 Overfitting

We examine the relationship between the number of training samples and the complexity of the model.

5.4 Ridge Regression

Regularization means that instead of seeking the model parameters by minimizing the training loss alone, we add a penalty term to force the parameters to "behave better".

With the ridge regression principle, we can optimize it as follows:

$$\begin{aligned}
J(\boldsymbol{\theta}) &= \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2 + \lambda\|\boldsymbol{\theta}\|_2^2 && (5.1)\\
&= (\mathbf{y} - \mathbf{X}\boldsymbol{\theta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\theta}) + \lambda\boldsymbol{\theta}^T\boldsymbol{\theta} && (5.2)\\
&= (\mathbf{y}^T - \boldsymbol{\theta}^T\mathbf{X}^T)(\mathbf{y} - \mathbf{X}\boldsymbol{\theta}) + \lambda\boldsymbol{\theta}^T\boldsymbol{\theta} && (5.3)\\
&= \mathbf{y}^T\mathbf{y} - \boldsymbol{\theta}^T\mathbf{X}^T\mathbf{y} - \mathbf{y}^T\mathbf{X}\boldsymbol{\theta} + \boldsymbol{\theta}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\theta} + \boldsymbol{\theta}^T\lambda\mathbf{I}\boldsymbol{\theta} && (5.4)\\
\frac{\partial J}{\partial \boldsymbol{\theta}} &= -\mathbf{X}^T\mathbf{y} - \mathbf{X}^T\mathbf{y} + \mathbf{X}^T\mathbf{X}\boldsymbol{\theta} + \mathbf{X}^T\mathbf{X}\boldsymbol{\theta} + 2\lambda\mathbf{I}\boldsymbol{\theta} = 0 && (5.5)\\
\boldsymbol{\theta} &= (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y} && (5.6)
\end{aligned}$$

- If $\lambda \to 0$, then $\|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2 + \underbrace{\lambda\|\boldsymbol{\theta}\|_2^2}_{=0}$
- $\lambda \to \infty$, then $\underbrace{\frac{1}{\lambda}\|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2}_{=0} + \|\boldsymbol{\theta}\|_2^2$, since what we want to do is to minimize the objective function, we can divide it by λ. Therefore, the solution will be $\boldsymbol{\theta} = 0$, because it is the smallest value the squared function can achieve.

Note that $\mathbf{X}^T\mathbf{X}$ is always symmetric [1]. Thus, it can be decomposed as $Q\Lambda Q^T$ by the Spectral theorem. The Q and Λ are eigenvector and eigenvalue matrices, respectively. Then, the inverse operation in the ridge regression can be expressed as follows:

$$\begin{aligned}
\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I} &= \mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^T + \lambda\mathbf{I}\\
&= \mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^T + \lambda\mathbf{Q}\mathbf{Q}^T\\
&= \mathbf{Q}(\boldsymbol{\Lambda} + \lambda\mathbf{I})\mathbf{Q}^T.
\end{aligned}$$

Even if the symmetric matrix is not invertible or close to not invertible, the regularization constant λ makes it invertible (by making it to be a full-rank).

[1] $(\mathbf{X}^T\mathbf{X})^T = \mathbf{X}^T\mathbf{X}$.

Note that we can change the ridge regression into a dual form:

$$
\begin{aligned}
(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})\boldsymbol{\theta} &= (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y} && (5.7)\\
(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})\boldsymbol{\theta} &= \mathbf{X}^T\mathbf{y} && (5.8)\\
\boldsymbol{\theta} &= \lambda^{-1}\mathbf{I}(\mathbf{X}^T\mathbf{y} - \mathbf{X}^T\mathbf{X}\boldsymbol{\theta}) && (5.9)\\
&= \mathbf{X}^T \underbrace{\lambda^{-1}(\mathbf{y} - \mathbf{X}\boldsymbol{\theta})}_{=\alpha} && (5.10)\\
&= \mathbf{X}^T\alpha && (5.11)\\
\lambda\alpha &= (\mathbf{y} - \mathbf{X}\boldsymbol{\theta}) && (5.12)\\
&= (\mathbf{y} - \mathbf{X}\mathbf{X}^T\alpha) && (5.13)\\
\mathbf{y} &= (\mathbf{X}\mathbf{X}^T\alpha + \lambda\alpha) && (5.14)\\
\alpha &= (\mathbf{X}\mathbf{X}^T + \lambda)^{-1}\mathbf{y} && (5.15)\\
\alpha &= (\mathbf{G} + \lambda)^{-1}\mathbf{y}. && (5.16)
\end{aligned}
$$

This gives us the solution of the underdetermined problems. For a new data point $\mathbf{x}_{new}$, we can make an inference by computing dot products between the new data point and each training data point:

$$
\begin{aligned}
y &= \boldsymbol{\theta}^T\mathbf{x}_{new}\\
&= \alpha^T\mathbf{X}\mathbf{x}_{new}
\end{aligned}
$$

5.4.1 Time Complexity

- Training time: $O(md^2 + m^3)$
- Inference time: $O(md)$

5.5 Weighted LSE

The OLEs assume an equal confidence on all the measurements. Now we look at varying confidence in the measurements. We assume that the noise for each measurement has zero mean and is independent, then the covariance matrix for all measurement noise is given by

$$
\begin{aligned}
R &= \mathbb{E}(\eta\eta^T)\\
&= \begin{bmatrix} \sigma_1^2 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \sigma_l^2 \end{bmatrix}
\end{aligned}
$$

By denoting the error vector $\mathbf{y} - \mathbf{X}\boldsymbol{\theta}$ as $\boldsymbol{\epsilon} = (\epsilon_1, \dots, \epsilon_l)^T$, we will minimize the sum of squared differences weighted over the variations of the measurements:

$$
\begin{aligned}
J(\tilde{\mathbf{x}}) = \boldsymbol{\epsilon}^T R^{-1}\boldsymbol{\epsilon} &= \frac{\boldsymbol{\epsilon}_1^2}{\sigma_1^2} + \dots + \frac{\boldsymbol{\epsilon}_l^2}{\sigma_l^2}\\
&= (\mathbf{y} - \mathbf{X}\boldsymbol{\theta})^T R^{-1}(\mathbf{y} - \mathbf{X}\boldsymbol{\theta})
\end{aligned}
$$

Note that by dividing each residual by its variance, we effectively equalize the influence of each data point on the overall fitting process. Subsequently, by taking the partial derivative of J with respect to $\boldsymbol{\theta}$, we get the best estimate of the parameter, which is given by

$$\boldsymbol{\theta} = (\mathbf{X}^T R^{-1} \mathbf{X})^{-1} \mathbf{X}^T R^{-1} \mathbf{y}.$$

Note that the measurement noise matrix R must be non-singular for a solution to exist.

5.6 Robust Linear Regression

The linear regression is based on the squared error criterion. This criterion often suffers from a serious drawback caused by outliers. By the definition of a squared error, our training loss is given by

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{n=1}^{N} \Big(y_n - g_{\boldsymbol{\theta}}(x_n) \Big)^2.$$

Let's assume that one of these error terms is large due to an outlier.

$$\mathcal{L}(\boldsymbol{\theta}) = \Big(y_1 - g_{\boldsymbol{\theta}}(x_1) \Big)^2 + \underbrace{\Big(y_2 - g_{\boldsymbol{\theta}}(x_2) \Big)^2}_{\text{Large}} + \cdots$$

If one or a few of these individual error terms are large, the square operation will amplify them. Consequently, the outliers suddenly have a very large contribution to the error. Since the goal of linear regression is to minimize the total loss, the presence of the outliers will drive the optimization solution to compensate for the large error.

Chapter 6

Recursive Least Squares

6.1 Recursive Least Squares

The ordinary least-squares assumes that all measurements are available at a certain time. However, this often might not be the case in practice. **More often, we obtain measurements sequentially and want to update our estimate with each new measurement.** In this case, the matrix $\mathbf{X}$ needs to be augmented. This update can be very expensive especially if the number of measurements is extremely large, the solutions of the least squares problem become prohibitive to compute.

This motivates the Recursive Least Squares (RLS). Suppose we have an estimate $\boldsymbol{\theta}_{k-1}$ after $(k-1)$ measurements and obtain a new measurement $\mathbf{y}_k$. How can we update our estimate without completely reworking on the pseudo-inverse problem?

A linear recursive estimator can be expressed in the following form:

$$\begin{aligned}\mathbf{y}_k &= \mathbf{X}_k\boldsymbol{\theta} + \boldsymbol{\eta}_k \\ \boldsymbol{\theta}_k &= \boldsymbol{\theta}_{k-1} + K_k(\mathbf{y}_k - \mathbf{X}_k\boldsymbol{\theta}_{k-1})\end{aligned}$$

Here, $\mathbf{X}_k$ is an $m \times d$ matrix (observations) and K_k is $d \times m$ and referred to as the *estimator gain matrix*. We refer to $(\mathbf{y}_k - \mathbf{X}_k\boldsymbol{\theta}_{k-1})$ as the *correction term*. Also, $\boldsymbol{\eta}_k$ is the measurement error. The new estimate is modified from the previous estimate $\boldsymbol{\theta}_{k-1}$ with a correction via the gain matrix.

Intuitively, we can notice that we have to compute the optimal gain matrix to update our estimate. To this end, we have to set an *estimation error*, which is our learning objective. The error can be expressed as follows:

$$\begin{aligned}\boldsymbol{\epsilon}_k &= \boldsymbol{\theta} - \boldsymbol{\theta}_k \\ &= \boldsymbol{\theta} - \boldsymbol{\theta}_{k-1} - K_k(\mathbf{y}_k - \mathbf{X}_k\boldsymbol{\theta}_{k-1}) \\ &= \boldsymbol{\epsilon}_{k-1} - K_k(\mathbf{X}_k\boldsymbol{\theta} + \boldsymbol{\eta}_k - \mathbf{X}_k\boldsymbol{\theta}_{k-1}) \\ &= \boldsymbol{\epsilon}_{k-1} - K_k\mathbf{X}_k(\boldsymbol{\theta} - \boldsymbol{\theta}_{k-1}) - K_k\boldsymbol{\eta}_k \\ &= (I - K_k\mathbf{X}_k)\boldsymbol{\epsilon}_{k-1} - K_k\boldsymbol{\eta}_k,\end{aligned}$$

where I is the $d \times d$ identity matrix. The mean of this error is then

$$\mathbb{E}[\boldsymbol{\epsilon}_k] = (I - K_k\mathbf{X}_k)\mathbb{E}[\boldsymbol{\epsilon}_{k-1}] - K_k\mathbb{E}[\boldsymbol{\eta}_k]$$

If $\mathbb{E}[\boldsymbol{\eta}_k] = 0$ and $\mathbb{E}[\boldsymbol{\epsilon}_{k-1}] = 0$, then $\mathbb{E}[\boldsymbol{\epsilon}_k] = 0$. So if the measurement noise has zero mean for all k, and the initial estimate of $\boldsymbol{\theta}$ is set equal to its expected value, then $\boldsymbol{\theta}_k = \boldsymbol{\theta}_k, \forall k$. This property tells us that the estimator $\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} + K_k(\mathbf{y}_k - \mathbf{X}_k\boldsymbol{\theta}_{k-1})$ is *unbiased*. This property holds regardless of the value of the gain vector K_k. This means the estimate will be equal to the true value $\boldsymbol{\theta}$ on average.

The key is to **determine the optimal value of the gain vector** K_k. The optimality criterion is to **minimize the aggregated variance of the estimation errors at time** k:

$$\begin{aligned} J_k &= \mathbb{E}[\|\boldsymbol{\theta} - \boldsymbol{\theta}_k\|^2] \\ &= \mathbb{E}[\boldsymbol{\epsilon}_k^T \boldsymbol{\epsilon}_k] \\ &= \mathbb{E}[tr(\boldsymbol{\epsilon}_k \boldsymbol{\epsilon}_k^T)] \\ &= tr(P_k), \end{aligned}$$

where $P_k = \mathbb{E}[\boldsymbol{\epsilon}_k \boldsymbol{\epsilon}_k^T]$ is *the estimation-error covariance* (*i.e.,* **covariance matrix**). Note that the third line holds by the trace of a product (*i.e., cyclic property*) and the expectation in the third line can go into the trace operator by its linearity. Next, we can obtain P_k by

$$P_k = \mathbb{E}\Big[\big((I - K_k\mathbf{X}_k)\boldsymbol{\epsilon}_{k-1} - K_k\boldsymbol{\eta}_k\big)\big((I - K_k\mathbf{X}_k)\boldsymbol{\epsilon}_{k-1} - K_k\boldsymbol{\eta}_k\big)^T\Big]$$

By rearranging the above equation with the property that the mean of noise is zero, we can get

$$P_k = (I - K_k\mathbf{X}_k)P_{k-1}(I - K_k\mathbf{X}_k)^T + K_k R_k K_k^T, \tag{6.1}$$

where $R_k = \mathbb{E}[\boldsymbol{\eta}_k \boldsymbol{\eta}_k^T]$ as covariance of $\boldsymbol{\eta}_k$. This equation is the recurrence for the covariance of the least squares estimation error. It is consistent with the intuition that as the measurement noise R_k increases, the uncertainty in our estimate also increases (*i.e.,* P_k increases). Note that P_k should be positive definite since it is a covariance matrix.

Next, let's compute K_k that minimizes the cost function given by the error equation. We are going to utilize the following property:

$$\begin{aligned} \frac{\partial tr(CA^T)}{\partial A} &= C \\ \frac{\partial tr(ACA^T)}{\partial A} &= AC + AC^T \end{aligned}$$

Next, we are going to take a derivative to the objective function:

$$\begin{aligned} \frac{\partial J_k}{\partial K_k} = \frac{\partial tr(P_k)}{\partial K_k} &= \frac{\partial tr}{\partial K_k}\underbrace{(I - K_k\mathbf{X}_k)P_{k-1}(I - K_k\mathbf{X}_k)^T}_{=ACA^T} + \frac{\partial}{\partial K_k} tr\left(K_k R_k K_k^T\right) \\ &= \frac{\partial tr(ACA^T)}{\partial (I - K_k\mathbf{X}_k)}\frac{\partial (I - K_k\mathbf{X}_k)}{\partial K_k} + \frac{\partial}{\partial K_k} tr\left(K_k R_k K_k^T\right) \quad \text{by Chain Rule} \\ &= \left((I - K_k\mathbf{X}_k)P_{k-1} + (I - K_k\mathbf{X}_k)P_{k-1}^T\right)(-\mathbf{X}_k^T) + \frac{\partial}{\partial K_k} tr\left(K_k R_k K_k^T\right) \\ &= 2(I - K_k\mathbf{X}_k)P_{k-1}(-\mathbf{X}_k^T) + \frac{\partial}{\partial K_k} tr\left(K_k R_k K_k^T\right) \quad \text{, since } P_{k-1} \text{ is symmetric.} \\ &= -2(I - K_k\mathbf{X}_k)P_{k-1}\mathbf{X}_k^T + 2K_k R_k \end{aligned}$$

By setting the partial derivative to zero, we get

$$K_k = P_{k-1}\mathbf{X}_k^T(\mathbf{X}_k P_{k-1}\mathbf{X}_k^T + R_k)^{-1}.$$

6.1.1 Alternative Form

Sometimes it is useful to write the equations for P_k and K_k in alternate forms. Although these alternate forms are mathematically identical, they can be beneficial from a computational point of view. Let's first set $\mathbf{X}_k P_{k-1} \mathbf{X}_k^T + R_k = S_k$, then we get

$$K_k = P_{k-1} \mathbf{X}_k^T S_k^{-1}.$$

By putting this into Eq. (6.1),

$$\begin{aligned} P_k &= (I - P_{k-1}\mathbf{X}_k^T S_k^{-1}\mathbf{X}_k) P_{k-1} (I - P_{k-1}\mathbf{X}_k^T S_k^{-1}\mathbf{X}_k)^T + P_{k-1}\mathbf{X}_k^T S_k^{-1} R_k S_k^{-1} \mathbf{X}_k P_{k-1} \\ &\vdots \\ &= P_{k-1} - P_{k-1}\mathbf{X}_k^T S_k^{-1} \mathbf{X}_k^T P_{k-1} \\ &= (I - K_k \mathbf{X}_k) P_{k-1}. \end{aligned}$$

Note that P_k is symmetric (*c.f.,* $P_k = \boldsymbol{\epsilon}_k \boldsymbol{\epsilon}_k^T$), since it is a covariance matrix, and so is S_k.

We take the inverse of both sides of

$$P_{k-1}^{-1} = \Bigg(\underbrace{P_{k-1}}_{A} - \underbrace{P_{k-1}\mathbf{X}_k^T}_{B} \big(\underbrace{\mathbf{X}_k P_{k-1} \mathbf{X}_k^T}_{D} \big)^{-1} \underbrace{\mathbf{X}_k P_{k-1}}_{C} \Bigg)^{-1}.$$

Next, we apply the matrix inversion lemma which is known as *Sherman-Morrison-Woodbury matrix identity* (or *matrix inversion lemma*) identity:

$$(A - BD^{-1}C)^{-1} = A^{-1} + A^{-1}B(D - CA^{-1}B)^{-1}CA^{-1}.$$

Then, rewrite P_k^{-1} as follows:

$$\begin{aligned} P_k^{-1} &= P_{k-1}^{-1} + P_{k-1}^{-1} P_{k-1} \mathbf{X}_k^T \big((\mathbf{X}_k P_{k-1} \mathbf{X}_k^T + R_k) - \mathbf{X}_k P_{k-1} P_{k-1}^{-1} (P_{k-1} \mathbf{X}_k^T) \big)^{-1} \mathbf{X}_k P_{k-1} P_{k-1}^{-1} \\ &= P_{k-1}^{-1} + \mathbf{X}_k^T R_k^{-1} \mathbf{X}_k \end{aligned}$$

This yields an alternative expression for the covariance matrix:

$$P_k = \big(P_{k-1}^{-1} + \mathbf{X}_k^T R_k^{-1} \mathbf{X}_k \big)^{-1}$$

We can also obtain

$$K_k = P_k \mathbf{X}_k^T R_k^{-1}$$

By

$$\begin{aligned} P_k &= (I - K_k \mathbf{X}_k) P_{k-1} \\ P_k P_{k-1}^{-1} &= (I - K_k \mathbf{X}_k) \\ P_k P_k^{-1} &= P_k P_{k-1}^{-1} + P_k \mathbf{X}_k^T R_k^{-1} \mathbf{X}_k = I \\ I &= (I - K_k \mathbf{X}_k) + P_k \mathbf{X}_k^T R_k^{-1} \mathbf{X}_k \\ K_k &= P_k \mathbf{X}_k^T R_k^{-1}. \end{aligned}$$

6.1.2 Summary of RLS

In sum, RLS can be updated as follows:

- Update the gain matrix:
  - $K_k = P_{k-1}\mathbf{X}_k^T(\mathbf{X}_k P_{k-1}\mathbf{X}_k^T + R_k)^{-1}$ or
  - $K_k = P_k\mathbf{X}_k^T R_k^{-1}$
- Update estimate: $\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} + K_k(\mathbf{y}_k - \mathbf{X}_k\boldsymbol{\theta}_{k-1})$
- Update error covariance matrix by either:
  - $P_k = (I - K_k\mathbf{X}_k)P_{k-1}$.
  - $P_k = (I - K_k\mathbf{X}_k)P_{k-1}(I - K_k\mathbf{X}_k)^T + K_k R_k K_k^T$,

Example: At sample time k, our measurement is

- $y_k = X_k\theta + \eta_k$
- $X_k = 1$
- $R_k = \mathbb{E}[\eta_k^2]$

For this scalar problem, the measurement matrix X_k is a scalar too, and the measurement noise covariance R_k is also a scalar. We will suppose that each measurement has the same covariance so the measurement covariance R_k is not a function of k, and can be written as R. Initially, before we have any measurements, we have some idea about the value of the θ, and this forms our initial estimate. We also have some uncertainty about our initial estimate, and this forms our initial covariance:

$$\begin{aligned}\hat{\theta}_0 &= \mathbb{E}[\theta] \\ P_0 &= \mathbb{E}[(\theta - \hat{\theta}_0)(\theta - \hat{\theta}_0)^T] \\ &= \mathbb{E}[(\theta - \hat{\theta}_0)^2]\end{aligned}$$

If we have absolutely no idea about θ, then $P(0) = \infty I$. If we are 100% certain about the θ before taking any measurements, then $P(0) = 0$. Let's compute the gain matrix at $k = 1$ by using the following equation:

$$K_k = P_{k-1}\mathbf{X}_k^T(\mathbf{X}_k P_{k-1}\mathbf{X}_k^T + R_k)^{-1}.$$

Then, we get

$$K_1 = P_0(P_0 + R)^{-1}.$$

Similarly, by

$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} + K_k(\mathbf{y}_k - \mathbf{X}_k\boldsymbol{\theta}_{k-1}),$$

we obtain

$$\hat{\theta}_1 = \hat{\theta}_0 + \frac{P_0}{P_0 + R}(y_1 - \hat{\theta}_0).$$

Finally, let's update our covariance matrix P_k by

$$P_k = (I - K_k\mathbf{X}_k)P_{k-1}(I - K_k\mathbf{X}_k)^T + K_k R_k K_k^T.$$

Then,

$$\begin{aligned} P_1 &= \left(I - \frac{P_0}{P_0+R}\right) P_0 I - \frac{P_0}{P_0+R} + \frac{P_0}{P_0+R} R \frac{P_0}{P_0+R} \\ &= \left(\frac{P_0 R^2}{(P_0+R)^2}\right) + \frac{P_0^2 R}{(P_0+R)^2} \\ &= \frac{P_0 R(P_0+R)}{(P_0+R)^2} \\ &= \frac{P_0 R}{P_0+R} \end{aligned}$$

By repeating these calculations, we can update the above parameters and find general expressions:

$$\begin{aligned} P_{k-1} &= \frac{P_0 R}{(k-1)P_0 + R} \\ K_k &= \frac{P_0}{kP_0+R} \\ \hat{\theta}_k &= \frac{(k-1)P_0+R}{kP_0+R}\hat{\theta}_{k-1} + \frac{P_0}{kP_0+R} y_k \end{aligned}$$

Note that if θ is known perfectly *a priori* (*i.e.,* θ is known perfectly before any measurements are obtained) then $P_0 = 0$ and the above equation show that $K_k = 0$ and $\hat{\theta} = \hat{\theta}_0$. That is, the optimal estimate of θ is independent of any measurements that are obtained. In sum, this indicates that no update from measurements is needed, as the estimate is already perfect.

On the other hand, if x is completely unknown a priori, then $P_0 \to \infty$, and the above equations simplify to

$$\begin{aligned} \hat{\theta}_k &= \frac{(k-1)P_0}{kP_0}\hat{\theta}_{k-1} + \frac{P_0}{kP_0} y_k \\ &= \frac{k-1}{k}\hat{\theta}_{k-1} + \frac{1}{k} y_k \\ &= \frac{1}{k}[(k-1)\hat{\theta}_{k-1} + y_k] \end{aligned}$$

In other words, the optimal estimate of θ is equal to the running average of the measurements y_k, which can be written as

$$\begin{aligned} \bar{y}_k &= \frac{1}{k}\sum_{j=1}^{k} y_j \\ &= \frac{1}{k}\left(\sum_{j=1}^{k-1} y_j + y_k\right) \\ &= \frac{1}{k}\left[(k-1)\frac{1}{k-1}\sum_{j=1}^{k-1} y_j + y_k\right] \\ &= \frac{1}{k}[(k-1)\bar{y}_{k-1} + y_k] \end{aligned}$$

6.1.3 Curve Fitting

In the recursive curve fitting problem, we measure data one sample at a time $(y_1, y_2 \ldots,)$ and want to find the best fit of a curve to the data. The curve that we want to fit to the data could be constrained to be linear or quadratic and so on.

Example: Suppose that we want to fit a straight line to a set of data points. The linear data fitting problem can be written as

$$y_k = \theta_1 + \theta_2 t_k + \eta_k$$
$$\mathbb{E}[\eta_k] = R_k$$

t_k is the independent variable, y_k is the noisy data, and we want to find the linear relationship between y_k and t_k. In sum, we want to estimate the constants θ_1 and θ_2. The measurement matrix can be written as

$$\mathbf{X}_k = \begin{bmatrix} 1 & t_k \end{bmatrix}.$$

Then,

$$\mathbf{y}_k = \mathbf{X}_k \boldsymbol{\theta} + \boldsymbol{\eta}_k.$$

6.1.4 Python Implementation

```
class RecursiveLeastSquares(object):

    # theta0 - initial estimate used to initialize the estimator
    # P0 - initial estimation error covariance matrix
    # R  - covariance matrix of the measurement noise
    def __init__(self,theta0,P0,R)

        # initialize the values
        self.theta0=theta0
        self.P0=P0
        self.R=R

        # this variable is used to track the current time step k of the estimator
        # after every time step arrives, this variables increases for one
        # in this way, we can track the number of variblaes
        self.curr_step=0

        # this list is used to store the estimates xk starting from the initial estimate
        self.estimates=[]
        self.estimates.append(theta0)

        # this list is used to store the estimation error covariance matrices Pk
        self.est_error_cov=[]
        self.est_error_cov.append(P0)

        # this list is used to store the gain matrices Kk
        self.gainMatrices=[]

        # this list is used to store estimation error vectors
        self.errors=[]


    # this function takes the current measurement and the current measurement matrix X
    # and computes the estimation error covariance matrix, updates the estimate,
    # computes the gain matrix, and the estimation error
    # it fills the lists self.estimates, self.est_error_cov, self.gainMatrices, and self.errors
    # it also increments the variable curr_step for 1
```

```
38
39      # measurementValue (theta) - measurement obtained at the time instant k
40      # X - measurement matrix at the time instant k
41
42      def predict(self,measurementValue,X):
43          import numpy as np
44
45          # Compute the L matrix and its inverse
46          # K_k = P_{k-1}X_k^T(R_k+X_kP_{k-1}X_k^T)^{-1}
47          Lmatrix=self.R+np.matmul(X,np.matmul(self.est_error_cov[self.curr_step
    ],X.T))
48          LmatrixInv=np.linalg.inv(Lmatrix)
49          # Compute the gain matrix
50          gainMatrix=np.matmul(self.est_error_cov[self.curr_step],np.matmul(X.T,
    LmatrixInv))
51
52          # Compute the estimation error
53          # \theta_k = \theta_{k-1}+K_k (y_k-X_k\theta_{k-1})
54          error=measurementValue-np.matmul(X,self.estimates[self.curr_step])
55          # Compute the estimate
56          estimate=self.estimates[self.curr_step]+np.matmul(gainMatrix,error)
57
58          # Propagate the estimation error covariance matrix
59          # P_k = (I-K_k X_k)P_{k-1}(I-K_k X_k)^T+K_kR_kK_k^T
60          ImKc=np.eye(np.size(self.theta0),np.size(self.theta0))-np.matmul(
    gainMatrix,X)
61          error_cov=np.matmul(ImKc,self.est_error_cov[self.curr_step])
62
63          # add computed elements to the list
64          self.estimates.append(estimate)
65          self.est_error_cov.append(error_cov)
66          self.gainMatrices.append(gainMatrix)
67          self.errors.append(error)
68
69          # increment the current time step
70          self.curr_step=self.curr_step+1
```

6.2 Alternate Derivation of RLS

Suppose the training samples arrive one by one in the following sequence $\mathbf{x}_1, \ldots, \mathbf{x}_m, \mathbf{x}_{m+1}$, where $\mathbf{x}_{m+1}$ denotes the newly arrived sample vector. These samples can be projected onto the feature space by linear projection and expressed into a matrix $\mathbf{P}^T \in \mathbb{R}^{(d+1)\times(m+1)}$ as follows:

$$\mathbf{P}^T = [\mathbf{p}(\mathbf{x}_1), \ldots, \mathbf{p}(\mathbf{x}_{m+1})],$$

where $\mathbf{p}(\cdot) \in \mathbb{R}^{d+1}$. Subsequently, let

$$\mathbf{R}_{m+1} = \mathbf{P}^T\mathbf{P} + b\mathbf{I}$$
$$\mathbf{Q}_{m+1} = \mathbf{P}^T\mathbf{y}.$$

By separating the covariance of the newly arrived sample $p(\mathbf{x}_{m+1})$ from the remaining stack, we can write:

$$\begin{aligned}\mathbf{P}^T\mathbf{P} &= \sum_{i=1}^{m+1}\mathbf{p}(\mathbf{x}_i)\mathbf{p}(\mathbf{x}_i)^T \\ &= \sum_{i=1}^{m}\mathbf{p}(\mathbf{x}_i)\mathbf{p}(\mathbf{x}_i)^T + \mathbf{p}(\mathbf{x}_{m+1})\mathbf{p}(\mathbf{x}_{m+1})^T \\ &= \mathbf{P}_m^T\mathbf{P}_m + \mathbf{p}(\mathbf{x}_{m+1})\mathbf{p}(\mathbf{x}_{m+1})^T.\end{aligned}$$

Hence,

$$\begin{aligned}\mathbf{R}_{m+1} &= \mathbf{P}^T\mathbf{P} + b\mathbf{I} \\ &= \left(\mathbf{P}_m^T\mathbf{P}_m + \mathbf{p}(\mathbf{x}_{m+1})\mathbf{p}(\mathbf{x}_{m+1})^T\right) + b\mathbf{I} \\ &= \underbrace{\mathbf{P}_m^T\mathbf{P}_m + b\mathbf{I}}_{=\mathbf{R}_m} + \mathbf{p}(\mathbf{x}_{m+1})\mathbf{p}(\mathbf{x}_{m+1})^T \\ &= \mathbf{R}_m + \mathbf{p}(\mathbf{x}_{m+1})\mathbf{p}(\mathbf{x}_{m+1})^T\end{aligned}$$

Similarly,

$$\mathbf{Q}_{m+1} = \mathbf{Q}_m + \mathbf{p}(\mathbf{x}_{m+1})y_{m+1}$$

If the system is designed to forget the old training samples (*i.e.,* weighted averaging),

$$\begin{aligned}\mathbf{R}_{m+1} &= (1-\lambda)\mathbf{R}_m + \lambda\mathbf{p}(\mathbf{x}_{m+1})\mathbf{p}(\mathbf{x}_{m+1})^T, \\ \mathbf{Q}_{m+1} &= (1-\lambda)\mathbf{Q}_m + \lambda\mathbf{p}(\mathbf{x}_{m+1})y_{m+1},\end{aligned}$$

where $\lambda \in (0,1)$ is often called a *forgetting factor.*

Let $\mathbf{A} = \mathbf{R}_m, \mathbf{B} = p(\mathbf{x}_{m+1}), \mathbf{C} = 1$ (scalar), $\mathbf{D} = p(\mathbf{x}_{m+1})^T = \mathbf{p}^T$, then based on the matrix inversion lemma (Woodbury, 1950; Sherman and Morrison, 1950),

$$(\mathbf{A}+\mathbf{BCD})^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{B}(\mathbf{C}^{-1} + \mathbf{D}\mathbf{A}^{-1}\mathbf{B})^{-1}\mathbf{D}\mathbf{A}^{-1},$$

we have

$$\begin{aligned}\mathbf{R}_{m+1}^{-1} &= \left[(1-\lambda)\mathbf{R}_m + \lambda\mathbf{p}\mathbf{p}^T\right]^{-1} \\ &= \frac{1}{1-\lambda}\mathbf{R}_m^{-1} - \frac{1}{1-\lambda}\mathbf{R}_m^{-1}\lambda\mathbf{p}\left(\mathbf{I} + \mathbf{p}^T\frac{\lambda}{1-\lambda}\mathbf{R}_m^{-1}\mathbf{p}\right)^{-1}\mathbf{p}^T\frac{1}{1-\lambda}\mathbf{R}_m^{-1} \\ &= \frac{1}{1-\lambda}\mathbf{R}_m^{-1} - \frac{1}{(1-\lambda)^2}\mathbf{R}_m^{-1}\mathbf{p}\mathbf{p}^T\mathbf{R}_m^{-1}\left(\frac{1}{\lambda} + \frac{1}{1-\lambda}\mathbf{p}^T\mathbf{R}_m^{-1}\mathbf{p}\right)^{-1}.\end{aligned}$$

We can obtain

$$\mathbf{w}_{m+1} = \left(\mathbf{P}^T\mathbf{P} + b\mathbf{I}\right)^{-1}\mathbf{P}^T\mathbf{y} = \mathbf{R}_{m+1}^{-1}\mathbf{Q}_{m+1},$$

Substitute $\mathbf{R}_{m+1}^{-1}$ and $\mathbf{Q}_{m+1} = (1-\lambda)\mathbf{Q}_m + \lambda\mathbf{p}(\mathbf{x}_{m+1})y_{m+1}$:

$$\begin{aligned}\mathbf{w}_{m+1} &= \left[\frac{1}{1-\lambda}\mathbf{R}_m^{-1} - \frac{1}{(1-\lambda)^2}\mathbf{R}_m^{-1}\mathbf{p}\mathbf{p}^T\mathbf{R}_m^{-1}\left(\frac{1}{\lambda} + \frac{1}{1-\lambda}\mathbf{p}^T\mathbf{R}_m^{-1}\mathbf{p}\right)^{-1}\right]\left[(1-\lambda)\mathbf{Q}_m + \lambda y_{m+1}\mathbf{p}\right] \\ &= \underbrace{\mathbf{R}_m^{-1}\mathbf{Q}_m}_{=w_m} + \frac{1}{1-\lambda}\mathbf{R}_m^{-1}\mathbf{p}\mathbf{p}^T\mathbf{R}_m^{-1}\left(\frac{1}{\lambda} + \frac{1}{1-\lambda}\mathbf{p}^T\mathbf{R}_m^{-1}\mathbf{p}\right)^{-1}\mathbf{Q}_m + \frac{\lambda}{1-\lambda}\mathbf{R}_m^{-1}\mathbf{p}y_{m+1} \\ &\quad - \frac{\lambda}{(1-\lambda)^2}\mathbf{R}_m^{-1}\mathbf{p}\mathbf{p}^T\mathbf{R}_m^{-1}\left(\frac{1}{\lambda} + \frac{1}{1-\lambda}\mathbf{p}^T\mathbf{R}_m^{-1}\mathbf{p}\right)^{-1}\mathbf{p}y_{m+1}\end{aligned}$$

Let

$$\begin{aligned} A &= \left(\frac{1}{\lambda} + \frac{1}{1-\lambda}\mathbf{p}^T\mathbf{R}_m^{-1}\mathbf{p}\right)^{-1} \\ &= \frac{\lambda(1-\lambda)}{\lambda\mathbf{p}^T\mathbf{R}_m^{-1}\mathbf{p} + (1-\lambda)}, \end{aligned}$$

which is a constant. Then

$$\begin{aligned} w_{m+1} &= w_m - \frac{\mathbf{R}_m^{-1}\mathbf{p}}{(1-\lambda)^2} \cdot A \cdot \big((1-\lambda)\mathbf{p}^T w_m + \lambda\mathbf{p}^T\mathbf{R}_m^{-1}\mathbf{p}y_{m+1}\big) + \frac{\lambda}{1-\lambda}\mathbf{R}_m^{-1}\mathbf{p}y_{m+1} \\ &= w_m - \frac{\lambda\mathbf{R}_m^{-1}\mathbf{p}}{(1-\lambda)} \cdot \frac{1}{\lambda\mathbf{p}^T\mathbf{R}_m^{-1}\mathbf{p} + (1-\lambda)} \big((1-\lambda)\mathbf{p}^T w_m + \lambda\mathbf{p}^T\mathbf{R}_m^{-1}\mathbf{p}y_{m+1}\big) + \frac{\lambda}{1-\lambda}\mathbf{R}_m^{-1}\mathbf{p}y_{m+1} \\ &= w_m - \frac{\lambda\mathbf{R}_m^{-1}\mathbf{p}}{(1-\lambda)} \cdot \frac{1}{\lambda\mathbf{p}^T\mathbf{R}_m^{-1}\mathbf{p} + (1-\lambda)} \big((1-\lambda)\mathbf{p}^T w_m + \lambda\mathbf{p}^T\mathbf{R}_m^{-1}\mathbf{p}y_{m+1}\big) \\ &\quad + \frac{\lambda}{(1-\lambda)} \cdot \frac{\lambda\mathbf{p}^T\mathbf{R}_m^{-1}\mathbf{p} + (1-\lambda)}{\lambda\mathbf{p}^T\mathbf{R}_m^{-1}\mathbf{p} + (1-\lambda)}\mathbf{R}_m^{-1}\mathbf{p}y_{m+1} \text{ , since } \mathbf{p}^T\mathbf{R}_m^{-1}\mathbf{p} \text{ is a scalar, we can put } \mathbf{R}_m^{-1}\mathbf{p} \text{ to the right} \\ &= w_m + \frac{\lambda}{\lambda\mathbf{p}^T\mathbf{R}_m^{-1}\mathbf{p} + (1-\lambda)} \cdot \frac{1}{(1-\lambda)}\big(-(1-\lambda)\mathbf{R}_m^{-1}\mathbf{p}\mathbf{p}^T w_m - \lambda\mathbf{R}_m^{-1}\mathbf{p}\mathbf{p}^T\mathbf{R}_m^{-1}\mathbf{p}y_{m+1} \\ &\quad + \lambda\mathbf{R}_m^{-1}\mathbf{p}\mathbf{p}^T\mathbf{R}_m^{-1}\mathbf{p}y_{m+1} + (1-\lambda)\mathbf{R}_m^{-1}\mathbf{p}y_{m+1}\big) \\ &= w_m + \frac{\lambda}{\lambda\mathbf{p}^T\mathbf{R}_m^{-1}\mathbf{p} + (1-\lambda)} \cdot \frac{1}{(1-\lambda)}\big(-(1-\lambda)\mathbf{R}_m^{-1}\mathbf{p}\mathbf{p}^T w_m + (1-\lambda)\mathbf{R}_m^{-1}\mathbf{p}y_{m+1}\big) \end{aligned}$$

Thus, the final recursive solution for the weight vector $\mathbf{w}_{m+1}$ is given by

$$\mathbf{w}_{m+1} = \mathbf{w}_m + \frac{\lambda\mathbf{R}_m^{-1}\mathbf{p}(y_{m+1} - \mathbf{p}^T\mathbf{w}_m)}{\lambda\mathbf{p}^T\mathbf{R}_m^{-1}\mathbf{p} + (1-\lambda)}$$

Here, we note that λ controls the strength of update with respect to the accumulated solution with $\lambda \to 1$ having the strongest weight for newly arrived sample. When $\lambda = 0.5$ in , we have the following regularized recursive least squares solution:

$$w_{m+1} = w_m + \frac{\mathbf{R}_m^{-1}\mathbf{p}(y_{m+1} - \mathbf{p}^T w_m)}{\mathbf{p}^T\mathbf{R}_m^{-1}\mathbf{p} + 1}.$$

Chapter 7

Logistic Regression

7.1 Logistic Regression

Logistic regression corresponds to the following binary classification model parameterized by $\mathbf{w}$:

$$p(y|\mathbf{x}, \mathbf{w}) = \text{Ber}(y|\sigma(\mathbf{w}^T\mathbf{x}))$$

Logistic regression models *logit*s (log odds) through a linear model. For binary data, the goal is to model the probability p that one of two outcomes occurs. Recall that an ordinary linear regression model is not bounded. Thus, we will pass a linear model through a sigmoid function, which is also known as logistic function.

$$\sigma(z) = \frac{1}{1 + \exp^z},$$

where $z = wx + b$. The sigmoid function has the property

$$\sigma(-x) = 1 - \sigma(x).$$

The z is often called the logit. Note that the inverse of the sigmoid is the log of the odds ratio $\frac{p}{1-p}$.

The logit function is $\log\frac{p}{1-p}$, which varies between $-\infty$ and $+\infty$ as p varies between 0 and 1.

$$\log\frac{p}{1-p} = w_0x_0 + w_1x_1 + \cdots + w_nx_n$$

Note that **the logistic regression model assumes that the log-odds (*logit*) of an observation y can be expressed as a linear function**. In this context, the logit function is called the ***link function*** because it "links" the probability to the linear function of the predictor variables.

In sum, the logistic regression models the probability $P(Y = 1|X)$ using the logistic function. The model can be written as:

$$P(Y = 1|X) = \sigma(X\beta) = \frac{1}{1 + \exp(-X\beta)}$$

where X is the vector of input features (including the intercept term), β is the vector of coefficients, and σ is the logistic (sigmoid) function.

We are going to use the Maximum Likelihood Estimation (MLE) to find the best parameter β. The MLE is a method used in statistics to estimate the parameters of a probability distribution by maximizing a likelihood function. Essentially, MLE finds the parameter values that make the observed data most probable.

Let's first define the Likelihood Function: Based on the probability distribution of the data, write down the likelihood function. For a set of observations $X = \{x_1, x_2, \ldots, x_n\}$, and a parameter θ, the likelihood function $L(\theta; X)$ represents the probability of observing the given data under the parameter θ.

Since the likelihood can be a product of probabilities, it might be easier to work with the natural logarithm of the likelihood function, called the log-likelihood. This transforms the product into a sum, simplifying the computation:

$$\ell(\theta; X) = \log L(\theta; X)$$

Finally, to find the optimal parameter, find the parameter value $\hat{\theta}$ that maximizes the log-likelihood function.

For a dataset with n observations, the likelihood function is the product of the probabilities of observing the given outcomes. For logistic regression, the likelihood function $L(\beta; X, Y)$ is given by:

$$L(\beta; X, Y) = \prod_{i=1}^{n} P(Y_i|X_i, \beta)$$

Given the binary nature of Y, this can be written as:

$$L(\beta; X, Y) = \prod_{i=1}^{n} [P(Y_i = 1|X_i, \beta)]^{Y_i} [P(Y_i = 0|X_i, \beta)]^{1-Y_i}$$

Substituting the logistic function, we get:

$$L(\beta; X, Y) = \prod_{i=1}^{n} \left[\frac{1}{1 + \exp(-X_i\beta)}\right]^{Y_i} \left[1 - \frac{1}{1 + \exp(-X_i\beta)}\right]^{1-Y_i}$$

The negative log-likelihood for logistic regression is given by

$$\begin{aligned}
\text{NLL}(\mathbf{w}) &= -\ln \prod_{i=1}^{N} p(\mathbf{x})^{\mathbb{I}(y_i=1)} (1 - p(\mathbf{x}))^{\mathbb{I}(y_i=0)\ 1} \\
&= -\ln \prod_{i=1}^{N} \sigma(\mathbf{w}^T\mathbf{x})^{\mathbb{I}(y_i=1)} (1 - \sigma(\mathbf{w}^T\mathbf{x}))^{\mathbb{I}(y_i=0)} \\
&= -\sum_{i=1}^{N} y_i \ln \sigma(\mathbf{w}^T\mathbf{x}) + \ln(1 - y_i)(1 - \sigma(\mathbf{w}^T\mathbf{x})).
\end{aligned}$$

This is also called **cross-entropy** error function.

To compute the derivative of NLL, we first need to know the following tricks:

- The derivative of $\ln(x)$:

$$\frac{\partial}{\partial x} \ln(x) = \frac{1}{x}.$$

[1] $\mathbb{I}(y_i = 1) = y_i$, because $y_i \in \{0, 1\}$ is a binary variable

- The derivative of the sigmoid is given by:

$$\frac{\partial \sigma(z)}{\partial x} = \sigma(x)(1 - \sigma(x)).$$

- Finally, the chain rule of derivative. Suppose we are computing the derivative of a composite function $f(x) = u(v(x))$. The derivative of $f(x)$ is the derivative of $u(x)$ with respect to $v(x)$ times the derivative of $v(x)$ with respect to x.

$$\frac{\partial f}{\partial x} = \frac{\partial u}{\partial v}\frac{\partial v}{\partial x}$$

The derivative of the loss function w.r.t., a single weight w_j is given by

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_j} &= \frac{\partial}{\partial w_j} - [\mathbf{y} \ln \sigma(\mathbf{wx} + \mathbf{b}) + (1 - \mathbf{y}) \ln(1 - \sigma(\mathbf{wx} + \mathbf{b}))] \\
&= -[\frac{\partial}{\partial w_j} y \ln \sigma(wx + b) + \frac{\partial}{\partial w_j}(1 - y) \ln(1 - \sigma(wx + b))] \\
&= -\frac{y}{\sigma(wx + b)}\frac{\partial}{\partial w_j}\sigma(wx + b) - \frac{1 - y}{1 - \sigma(wx + b)}\frac{\partial}{\partial w_j}\left[1 - \sigma(wx + b)\right] \\
&= -\left[\frac{y}{\sigma(wx + b)} - \frac{1 - y}{1 - \sigma(wx + b)}\right]\frac{\partial}{\partial w_j}\sigma(wx + b) \\
&= -\left[\frac{y - \sigma(wx + b)}{\sigma(wx + b)[1 - \sigma(wx + b)]}\right]\sigma(wx + b)[1 - \sigma(wx + b)]\frac{\partial \sigma(wx + b)}{\partial w_j} \\
&= -\left[\frac{y - \sigma(wx + b)}{\sigma(wx + b)[1 - \sigma(wx + b)]}\right]\sigma(wx + b)[1 - \sigma(wx + b)]x_j \\
&= -(y - \sigma(wx + b))x_j \\
&= (\sigma(wx + b) - y)x_j.
\end{aligned}$$

Thus, the gradient of the cost function $J(\boldsymbol{\theta})$ with respect to the weights $\boldsymbol{\theta}$ is:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m}\mathbf{X}^T(\sigma(\mathbf{X}\boldsymbol{\theta}) - \mathbf{y})$$

This gradient is used in gradient descent to update the weights $\boldsymbol{\theta}$:

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}),$$

where α is the learning rate.

Chapter 8

Bayesian Regression

8.1 Bayesian Regression

Chain rule: $P(A, B|C) = P(A|B, C)P(B|C)$

$$\begin{aligned} P(heads \mid D) &= \int_\theta P(heads, \theta \mid D) d\theta \\ &= \int_\theta P(heads \mid \theta, D) P(\theta \mid D) d\theta \\ &= \int_\theta \theta P(\theta \mid D) d\theta \\ &= E\left[\theta | D\right] \\ &= \frac{n_H + \alpha}{n_H + \alpha + n_T + \beta} \end{aligned}$$

8.2 Bayesian Regression

Bayesian regression is derived by combining the concepts of linear regression with Bayesian probability. In this note, we will derive Bayesian linear regression step-by-step, starting from a basic linear model and incorporating Bayesian inference.

8.3 Linear Regression Model

We assume a simple linear model:

$$y = X\beta + \epsilon,$$

where:

- y is the vector of observed responses (dependent variable),
- X is the matrix of observed features (independent variables),

- β is the vector of regression coefficients,
- $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$ is the noise term (normally distributed with mean 0 and variance σ^2).

This gives the likelihood function:

$$p(y|X, \beta, \sigma^2) = \mathcal{N}(y|X\beta, \sigma^2 I)$$

In the Bayesian framework, we treat the regression coefficients β as random variables with a prior distribution. Bayesian regression involves the following key elements:

- **Prior:** $p(\beta)$, which represents our belief about β before observing the data.
- **Likelihood:** $p(y|X, \beta)$, the likelihood of observing the data given the regression coefficients.
- **Posterior:** $p(\beta|y, X)$, which updates the prior belief based on the observed data.

According to Bayes' theorem, the posterior is:

$$p(\beta|y, X) = \frac{p(y|X, \beta)p(\beta)}{p(y|X)}$$

The term $p(y|X)$ is the marginal likelihood (evidence), which acts as a normalizing constant to ensure that the posterior is a valid probability distribution.

8.3.1 Prior on β

A common choice for the prior distribution on β is a Gaussian distribution:

$$p(\beta) = \mathcal{N}(\beta|\mu_0, \Sigma_0),$$

where μ_0 is the prior mean (our prior belief about β), and Σ_0 is the prior covariance matrix (representing the uncertainty in our prior belief).

8.3.2 Posterior Distribution

The posterior distribution of β is obtained by combining the likelihood with the prior using Bayes' theorem.

The likelihood is:

$$p(y|X, \beta) = \mathcal{N}(y|X\beta, \sigma^2 I)$$

The prior is given by

$$p(\beta) = \mathcal{N}(\beta|\mu_0, \Sigma_0)$$

Multiplying the prior and the likelihood (and applying properties of Gaussian distributions), we obtain the posterior distribution, which is also Gaussian:

$$p(\beta|y, X) = \mathcal{N}(\beta|\mu_n, \Sigma_n),$$

where:

- The **posterior mean** μ_n is given by:

$$\mu_n = \Sigma_n \left(\Sigma_0^{-1} \mu_0 + \frac{1}{\sigma^2} X^T y \right)$$

- The **posterior covariance** Σ_n is given by:

$$\Sigma_n = \left(\Sigma_0^{-1} + \frac{1}{\sigma^2} X^T X \right)^{-1}$$

8.3.3 Prediction

Once we have the posterior distribution $p(\beta|y, X)$, we can make predictions on new data X_*.

The predictive distribution for the new output y_* given new input X_* is obtained by integrating over the posterior distribution of β:

$$p(y_*|X_*, y, X) = \int p(y_*|X_*, \beta) p(\beta|y, X) d\beta$$

Since both the likelihood and posterior are Gaussian, the predictive distribution is also Gaussian:

$$p(y_*|X_*, y, X) = \mathcal{N}(y_*|X_* \mu_n, X_* \Sigma_n X_*^T + \sigma^2 I)$$

In Bayesian regression, we incorporate prior beliefs about the regression coefficients through the prior distribution $p(\beta)$, and update this belief after observing data to obtain the posterior distribution $p(\beta|y, X)$. This method provides not only point estimates of the regression coefficients but also quantifies the uncertainty in these estimates through the posterior distribution.

Part III

Kernel Methods

Chapter 9

Introduction to Kernel Methods

9.1 Kernels

$$\hat{\boldsymbol{\theta}} = \mathbf{X}^T(\mathbf{X}\mathbf{X}^T + \lambda\mathbf{I})^{-1}\mathbf{y}.$$

By substituting into $\hat{y}(\mathbf{x}') = \hat{\boldsymbol{\theta}}^T\mathbf{x}' = (\mathbf{x}')^T\hat{\boldsymbol{\theta}}$ gives

$$\hat{y}(\mathbf{x}') = (\mathbf{x}')^T\hat{\boldsymbol{\theta}} = (\mathbf{x}')^T\mathbf{X}^T(\mathbf{X}\mathbf{X}^T + \lambda\mathbf{I})^{-1}\mathbf{y}.$$

The prediction depends on the data only though inner products, since

$$(\mathbf{x}')^T\mathbf{X}^T = \begin{bmatrix}\langle\mathbf{x}', \mathbf{x}_1\rangle \\ \vdots \\ \langle\mathbf{x}', \mathbf{x}_n\rangle\end{bmatrix}, \quad \mathbf{X}\mathbf{X}^T = \begin{bmatrix}\langle\mathbf{x}_1, \mathbf{x}_1\rangle & \cdots & \langle\mathbf{x}_1, \mathbf{x}_n\rangle \\ \vdots & \ddots & \vdots \\ \langle\mathbf{x}_n, \mathbf{x}_1\rangle & \cdots & \langle\mathbf{x}_n, \mathbf{x}_n\rangle\end{bmatrix},$$

Kernel trick replaces the inner products by kernel evaluations, which can be represented as follows:

$$\hat{y}(\mathbf{x}') = k(\mathbf{x}')(K + \lambda\mathbf{I})^{-1}\mathbf{y},$$

where

$$k(\mathbf{x}') = \begin{bmatrix}k(\mathbf{x}', \mathbf{x}_1) \\ \vdots \\ k(\mathbf{x}', \mathbf{x}_n)\end{bmatrix} \quad K = \begin{bmatrix}k(\mathbf{x}_1, \mathbf{x}_1) & \cdots & k(\mathbf{x}_1, \mathbf{x}_n) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x}_n, \mathbf{x}_1) & \cdots & k(\mathbf{x}_n, \mathbf{x}_n)\end{bmatrix},$$

9.1.1 Non-Parametric Regression

Kernel methods corresponding to infinite-dimensional $\phi(\mathbf{x})$ can usually be considered as **non-parametric**. For instance, 1-D case can be considered as a joining the dots by interpolation.

9.1.2 Bayesian Posterior Derivation

Both $v(\mathbf{x}) = \langle\boldsymbol{\theta}, \mathbf{x}\rangle$ and $v(\mathbf{x}') = \langle\boldsymbol{\theta}, \mathbf{x}'\rangle$ have mean zero, so their covariance is

$$\begin{aligned} Cov[v(\mathbf{x}), v(\mathbf{x}')] &= \mathbb{E}[(\mathbf{x}^T\boldsymbol{\theta})(\mathbf{x}')^T\boldsymbol{\theta}] = \mathbb{E}[(\mathbf{x}^T\boldsymbol{\theta})(\boldsymbol{\theta}^T\mathbf{x}')] \\ &= \mathbf{x}^T\mathbb{E}[\boldsymbol{\theta}\boldsymbol{\theta}^T]\mathbf{x}' = \langle\boldsymbol{\theta}, \mathbf{x}\rangle \end{aligned}$$

since $\mathbb{E}[\boldsymbol{\theta}\boldsymbol{\theta}^T] = \mathbf{I}$ for $\boldsymbol{\theta} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$.

9.2 From Feature Transformations to Kernels

Instead of computing $\phi(\mathbf{x}_i), \forall i$, we pre-compute $\langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle$ for all $\mathbf{x}_i$ and $\mathbf{x}_j$. Then we store the values in an $n \times n$ matrix K where $K_{ij} = \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle$. K is called the kernel matrix or Gram matrix.

Computing $\phi(\mathbf{x}_i)$ is $\mathcal{O}(2^d) \times n = \mathcal{O}(2^d n)$. On the other hand, pre-computing the inner products can be more efficient as follows:

$$\begin{aligned} \langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle &= \phi(\mathbf{x})^T \phi(\mathbf{x}') \\ &= \prod_{k=1}^{d} (1 + x_k x'_k). \end{aligned}$$

This takes $\mathcal{O}(d)$ and we can compute the entire kernel matrix K in $\mathcal{O}(dn^2)$.

Note that all kernel methods are non-parametric models as we need to keep training data to be able to compute the kernel values between new test inputs and the training inputs.

9.3 Kernels

$k(\cdot, \cdot)$ is a valid or well-defined kernel, if the function $k(\mathbf{x}, \mathbf{x}')$ is both

- Symmetric: $k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x}', \mathbf{x})$ for all $\mathbf{x}, \mathbf{x}'$.
- Positive semi-definite: $k(\cdot, \cdot)$ is PSD if for all finite subsets $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$, $\mathbf{x}_i \in \mathcal{X}$, K is a PSD matrix.

Note that

- A matrix $A \in \mathbb{R}^{m \times m}$ is PSD iff $\forall \mathbf{q} \in \mathbb{R}^m$ the following holds: $\mathbf{q}^T A \mathbf{q} \geq 0$.
- A symmetric matrix $A \in \mathbb{R}^{m \times m}$ is PSD iff all eigenvalues λ of A are non-negative.
- A symmetric matrix A is PSD if all its upper left sub-matrices have non-negative determinants.

For symmetric matrices all of the above are equivalent.

Some of the most popular kernels are

- Linear kernel: $k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T \mathbf{x}'$
- Polynomial kernel: $k(\mathbf{x}, \mathbf{x}') = (1 + \mathbf{x}^T \mathbf{x}')^{\tilde{d}}$, where $\tilde{d}$ is the degree of the polynomial

Note that some kernels such as the RBF kernel have an infinite-dimensional feature space transformation. Hence, it is impossible to compute the feature space transformation explicitly. The RBF kernel is given by

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

Let, $\gamma = \frac{1}{2}$,

$$\begin{aligned}
k(\mathbf{x}, \mathbf{x}') &= \exp(-\frac{1}{2}\|\mathbf{x} - \mathbf{x}'\|^2) \\
&= \exp\left[-\frac{1}{2}\langle \mathbf{x} - \mathbf{x}', \mathbf{x} - \mathbf{x}' \rangle \right] \\
&= \exp\left[-\frac{1}{2}\langle \mathbf{x}, \mathbf{x} - \mathbf{x}' \rangle - \langle \mathbf{x}', \mathbf{x} - \mathbf{x}' \rangle \right] \\
&= \exp\left[-\frac{1}{2}\big(\|\mathbf{x}\|^2 + \|\mathbf{x}'\|^2 - 2\langle \mathbf{x}, \mathbf{x}' \rangle \big) \right] \\
&= C \exp\langle \mathbf{x}, \mathbf{x}' \rangle \\
&= C \sum_{n=0}^{\infty} \frac{\langle \mathbf{x}, \mathbf{x}' \rangle}{n!} \\
&= C \sum_{n=0}^{\infty} \frac{k_{\text{poly}}(\mathbf{x}, \mathbf{x}')}{n!}
\end{aligned}$$

The second to last step is by Taylor expansion of e^x.

9.3.1 Some Intuitions

Let $y = f(\mathbf{x}) + \epsilon$. Then f does not vary a lot if $\mathbf{x}$, $\mathbf{x}'$ are close enough. This can be modeled by the co-variance of the y's.

Let's represent the co-variance of the outputs in terms of the co-variance of the inputs.

$$cov(y, y') = k(\mathbf{x}, \mathbf{x}') + \underbrace{\sigma_n^2 \delta_{\mathbf{x},\mathbf{x}'}}_{\text{noise term}} .$$

As y is a function values of $f(\mathbf{x})$, the kernel function can be viewed as an extension of the covariance matrix Σ of random vectors to the covariance of (random) functions.

Chapter 10

Gaussian Process

10.1 Introduction

A Gaussian process (GP) is a probability distribution over possible functions that fit a set of points fully specified by a mean and covariance function. More formally, GPs are distributions over functions $f(x)$ of which the distribution is defined by a mean function $m(x)$ and positive definite covariance function $k(x, x')$ (*i.e., kernel*). Thus, it is a distribution over functions whose shape is defined by the kernel:

$$f(x) \sim \mathcal{GP}(m(x), k(x, x')).$$

Since we have the probability distribution over all possible functions, we can calculate the mean and the variance of the function to determine how confident in our predictions.

Let's say we have observations $\boldsymbol{X} = \{x_1, \ldots, x_n\}$, and we have estimated functions $\mathbf{f} = [f(x_1), \ldots, f(x_n)]$ with these observations. Now say we have some new points $\mathbf{X}_*$ where we want to predict $f(\mathbf{X}_*)$.

The joint distribution of $\mathbf{f}$ and $\mathbf{f}_*$ can be modeled as:

$$\begin{pmatrix} \mathbf{f} \\ \mathbf{f}_* \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} m(\mathbf{X}) \\ m(\mathbf{X}_*) \end{pmatrix}, \begin{pmatrix} \mathbf{K} & \mathbf{K}_* \\ \mathbf{K}_*^T & \mathbf{K}_{**} \end{pmatrix} \right),$$

where $m(X) = [m(x_1), \ldots, m(x_n)]$ and

$$\begin{aligned} \mathbf{K} &= \kappa(\mathbf{X}, \mathbf{X}) \\ \mathbf{K}_* &= \kappa(\mathbf{X}, \mathbf{X}_*) \\ \mathbf{K}_{**} &= \kappa(\mathbf{X}_*, \mathbf{X}_*). \end{aligned}$$

The mean is assumed to be $(m(\mathbf{X}), m(\mathbf{X}_*)) = 0$.

While this equation describes the joint probability distribution $P(\mathbf{f}, \mathbf{f}_* | \mathbf{X}, \mathbf{X}_*)$ over $\mathbf{f}$ and $\mathbf{f}_*$ in regressions, we need the conditional distribution $P(\mathbf{f}_* | \mathbf{f}, \mathbf{X}, \mathbf{X}_*)$ from the joint distribution. The conditional distribution can be achieved by the Marginal and conditional distributions of MVN theorem:

Theorem 1 *Marginals and conditionals of an MVN: A random vector* $\mathbf{X}$ *follows a multivariate normal distribution:*

$$\mathbf{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$$

where $\mathbf{X}$ *can be partitioned as:*

$$\mathbf{X} = \begin{bmatrix} \mathbf{X}_1 \\ \mathbf{X}_2 \end{bmatrix}$$

with corresponding partitions of the mean vector $\boldsymbol{\mu}$ *and the covariance matrix* $\boldsymbol{\Sigma}$*:*

$$\boldsymbol{\mu} = \begin{bmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \end{bmatrix}, \quad \boldsymbol{\Sigma} = \begin{bmatrix} \boldsymbol{\Sigma}_{11} & \boldsymbol{\Sigma}_{12} \\ \boldsymbol{\Sigma}_{21} & \boldsymbol{\Sigma}_{22} \end{bmatrix}, \quad \boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1} = \begin{bmatrix} \boldsymbol{\Lambda}_{11} & \boldsymbol{\Lambda}_{12} \\ \boldsymbol{\Lambda}_{21} & \boldsymbol{\Lambda}_{22} \end{bmatrix},$$

Then, the marginal distribution of $\mathbf{X}_1$ *and* $\mathbf{X}_2$ *are given by*

$$\mathbf{X}_1 \sim \mathcal{N}(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_{11})$$
$$\mathbf{X}_2 \sim \mathcal{N}(\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_{22})$$

Subsequently, the (posteriror) conditional distribution of $\mathbf{X}_1$ *given* $\mathbf{X}_2$ *is:*

$$\mathbf{X}_1 | \mathbf{X}_2 = \mathbf{x}_2 \sim \mathcal{N}(\boldsymbol{\mu}', \boldsymbol{\Sigma}'),$$

where the conditional mean $\boldsymbol{\mu}_{1|2}$ *and the conditional covariance* $\boldsymbol{\Sigma}_{1|2}$ *are given by:*

$$\boldsymbol{\mu}' = \boldsymbol{\mu}_1 + \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}(\mathbf{x}_2 - \boldsymbol{\mu}_2)$$
$$\boldsymbol{\Sigma}' = \boldsymbol{\Sigma}_{11} - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{21}$$

Note that $\boldsymbol{\Sigma}'$ does no depend on $\mathbf{x}_2$. It doesn't hold for general (non-Gaussian) random variables. By using the theorem, we get

$$\mathbf{f}_* | \mathbf{f}, \mathbf{X}, \mathbf{X}_* \sim \mathcal{N}(\mathbf{K}_*^T \mathbf{K}^{-1} \mathbf{f}, \mathbf{K}_{**} - \mathbf{K}_*^T \mathbf{K}^{-1} \mathbf{K}_*)$$

In practice, we typically have noisy estimations of target functions, $y = f(x) + \epsilon$. By assuming the i.i.d., Gaussian noise with variance σ_n^2, the prior on these noisy observations then becomes $cov(y) = \mathbf{K} + \sigma_n^2 \mathbf{I}$. The joint distribution of the observed target values and the function values at the test locations under the prior as

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{f}_* \end{bmatrix} \sim \mathcal{N}\left[\mathbf{0}, \begin{bmatrix} \mathbf{K} + \sigma_n^2 \mathbf{I} & \mathbf{K}_* \\ \mathbf{K}_*^T & \mathbf{K}_{**} \end{bmatrix}\right]$$

10.2 Regression using Gaussian Process

- A set of input points $X = \{x_1, x_2, \ldots, x_n\}$ (*i.e.,* training data).
- A set of corresponding output values $y = \{y_1, y_2, \ldots, y_n\}$.

We assume that the output values are generated by some unknown function $f(x)$ with some Gaussian noise ϵ:

$$y = f(X) + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma_n^2)$$

The goal is to predict the value of the function $f(x_*)$ at a new input point x_*.

We assume that the function $f(x)$ is drawn from a **Gaussian Process**:

$$f(x) \sim \mathcal{GP}(m(x), k(x, x')),$$

where:

- $m(x)$ is the mean function (usually assumed to be 0 for simplicity).
- $k(x, x')$ is the covariance function (or kernel), which encodes assumptions about the smoothness of the function.

The actual observations $y = f(x) + \epsilon$ include Gaussian noise ϵ, which is typically assumed to have zero mean and variance σ_n^2.

To predict the function value $f_* = f(x_*)$ at a new point x_*, we use the **posterior distribution** over the function, conditioned on the observed data X and y. This leads to two main equations:

- Posterior Mean (Prediction):

$$\mathbb{E}[f_*] = K(X, x_*)^\top [K(X, X) + \sigma_n^2 I]^{-1} y$$

- Posterior Variance (Uncertainty):

$$\text{Var}[f_*] = K(x_*, x_*) - K(X, x_*)^\top [K(X, X) + \sigma_n^2 I]^{-1} K(X, x_*)$$

These equations give us the mean and variance of the predictive distribution for the function value at the new point x_*.

Example: First, choose a covariance function (*i.e.,* Kernel). The kernel function $k(x, x')$ is crucial in defining the relationship between different input points. Some commonly used kernels are:

Squared Exponential (RBF) Kernel:

$$k(x, x') = \sigma_f^2 \exp\left(-\frac{(x - x')^2}{2l^2}\right),$$

where l is the length scale and σ_f^2 is the signal variance. The kernel function governs the smoothness and behavior of the GP, so selecting an appropriate kernel is important.

Subsequently, compute the covariance matrices:

- Covariance Matrix for Training Points: $K(X, X)$

$$K(X, X) = \begin{bmatrix} k(x_1, x_1) & k(x_1, x_2) & \dots & k(x_1, x_n) \\ k(x_2, x_1) & k(x_2, x_2) & \dots & k(x_2, x_n) \\ \vdots & \vdots & \ddots & \vdots \\ k(x_n, x_1) & k(x_n, x_2) & \dots & k(x_n, x_n) \end{bmatrix}$$

- Covariance Between Training Points and Test Point: $K(X, x_*)$

$$K(X, x_*) = \begin{bmatrix} k(x_1, x_*) \\ k(x_2, x_*) \\ \vdots \\ k(x_n, x_*) \end{bmatrix}$$

- Covariance at the Test Point: $K(x_*, x_*) = k(x_*, x_*)$

Then, compute the posterior mean and variance by Using the posterior mean and posterior variance equations from earlier, compute:

- The mean prediction $\mathbb{E}[f_*]$ at the test point x_*.
- The uncertainty $\text{Var}[f_*]$ at x_*.

This gives the complete predictive distribution for the function value at x_*.

Chapter 11

Support Vector Machine

Support Vector Machines (SVMs) are among the most effective and versatile tools in machine learning, widely used for various tasks. SVMs work by finding the optimal boundary, or hyperplane, that separates different classes of data with the maximum margin, making them highly reliable for classification, especially with complex datasets.

What truly sets SVMs apart is their ability to handle both linear and non-linear data through the *kernel trick*, allowing them to adapt to a wide range of problems with impressive accuracy. In this blog post, we'll delve into how SVMs work and gently explore the mathematical foundations behind their powerful performance.

11.1 Orthogonal Projection

When working with vectors x and y, finding the orthogonal projection of x onto y is a common task in linear algebra. The projection is a way to express how much of x lies in the direction of y.

By definition, the magnitude of the projection z of x onto y is given by::

$$\|z\| = \|x\| cos(\theta).$$

Here, θ is the angle between x and y. To connect this with the dot product, recall that:

$$x \cdot y = \|x\| \ \|y\| cos(\theta).$$

This formula allows us to replace the cosine term:

$$\|z\| = \|x\| \frac{x \cdot y}{\|x\| \cdot \|y\|}.$$

Simplifying further, we express the magnitude of z as:

$$\|z\| = u \cdot x,$$

where u is an unit vector of y. Since z is in the direction of y, we can write:

$$z = \|z\| \cdot u,$$

Then,

$$z = (u \cdot x) \cdot u.$$

This gives us the final expression for the orthogonal projection of x onto y:

$$\begin{aligned}\text{Proj}_y x &= (u \cdot x) \cdot u \\ &= \left(\frac{y \cdot x}{\|y\|^2}\right) y \\ &= \left(\frac{y \cdot x}{\|y\|}\right) \frac{y}{\|y\|}\end{aligned}$$

In this formula, the projection $\text{Proj}_y x$ represents the component of x that lies along the direction of y.

11.2 Decision Boundary with Margin

A hyperplane(or decision surface) is used to separate data points belonging to different classes. The goal of SVM is to find the optimal separating hyperplane. However, what is the optimal separating hyperplanes? **The optimal hyperplane is the one which maximizes the distance from the hyperplane to the nearest data point of any class. Support vectors are the data points that lie closest to the hyperplane**. The distance is referred to as the *margin*. SVMs maximize the margin around the separating hyperplane.

The equation of a hyperplane in $\mathbb{R}^p$ can be expressed as:

$$\mathbf{w} \cdot \mathbf{x} + b = 0.$$

Here, $\mathbf{w}$ is the normal vector to the hyperplane. It is clear by expressing it

$$\mathbf{w}(\mathbf{x} - \mathbf{x}_0) = 0,$$

where $b = \mathbf{w} \cdot \mathbf{x}_0$.

Let's consider a simple scenario, where training data is linearly separable:

$$\mathcal{D} = \{(\mathbf{x}_i, y_i) \mid \mathbf{x}_i \in \mathbb{R}^p,\, y_i \in \{-1, 1\}\}_{i=1}^{N}.$$

Then, we can build two hyperplanes separating the data with no points between them:

- $H_1 : \mathbf{w} \cdot \mathbf{x} + b = 1$
- $H_2 : \mathbf{w} \cdot \mathbf{x} + b = -1$

All samples have to satisfy one of two constraints:

1. $\mathbf{w} \cdot \mathbf{x} + b \geq 1$
2. $\mathbf{w} \cdot \mathbf{x} + b \leq -1$

These constraints can be combined into a single expression:

$$y(\mathbf{w} \cdot \mathbf{x} + b) \geq 1.$$

To maximize the margin, we can consider a unit vector $\mathbf{u} = \frac{\mathbf{w}}{\|\mathbf{w}\|}$, which is perpendicular to the hyperplanes and a point x_0 on the hyperplane H_2. If we scale u from x_0, we get $z = x_0 + ru$. If we assume z is on H_1, then $\mathbf{w} \cdot z + b = 1$. This is equivalent to

$$\begin{aligned} \mathbf{w} \cdot (x_0 + ru) + b &= 1 \\ \mathbf{w}x_0 + \mathbf{w}r\frac{\mathbf{w}}{\|\mathbf{w}\|} + b &= 1 \\ \mathbf{w}x_0 + r\|\mathbf{w}\| + b &= 1 \\ \mathbf{w}x_0 + b &= 1 - r\|\mathbf{w}\| \end{aligned}$$

As x_0 is on H_2, we get $\mathbf{w}x_0 + b = -1$. Finally, we obtain

$$\begin{aligned} -1 &= 1 - r\|\mathbf{w}\| \\ r &= \frac{2}{\|\mathbf{w}\|}. \end{aligned}$$

Note that the scaled unit vector ru's magnitude is r. Thus, the maximization of margin is equivalent to maximize r. To maximize r, we have to minimize $\|\mathbf{w}\|$. Thus, finding the optimal hyperplane reduces to solving the following optimization problem:

$$\begin{aligned} &\min\|\mathbf{w}\|, \quad \text{subject to} \\ &y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 \quad \forall i. \end{aligned}$$

Equivalently,

$$\begin{aligned} &\min \frac{1}{2}\|w\|^2, \quad \text{subject to} \\ &y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 \quad \forall i. \end{aligned}$$

Now, we have *convex quadratic optimization problem*. The solution of this problem gives us the optimal hyperplane that maximizes the margin (*c.f.,* §11.4). However, in practice, the data may not be perfectly separable. To account for this, we introduce a *soft margin* that allows for some misclassification. This is done by admitting small errors in classification and potentially using a more complex, *nonlinear decision boundary*, improving the generalization of the model.

11.3 Error Handling in SVM

In practice, it's unrealistic to expect a perfect separation of data, especially when the data is noisy or not linearly separable. To address this, we can allow for some prediction errors while still striving to find an optimal decision boundary.

One approach is to minimize the norm of the weight vector, while penalizing the number of errors N_e. The optimization problem can be formulated as follows:

$$\begin{aligned} &\min \frac{1}{2}\|\mathbf{w}\|^2 + C \cdot N_e, \quad \text{subject to} \\ &y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 \quad \forall i. \end{aligned}$$

Here, C is a regularization parameter that controls the trade-off between minimizing the weight vector and the number of errors. The penalty approach described here is known as *0-1 loss*, where all errors are treated equally. However, this approach is not commonly used. Instead, a more practical approach introduces a *slack variable* with *hinge loss*. The slack variable (ξ_j)

measures the degree of misclassification or how much a point is violating the margin. This leads to the following problem:

$$\min \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_j \xi_j, \quad \text{subject to}$$
$$\begin{cases} y_i(\mathbf{w}\cdot\mathbf{x}_i + b) \geq 1 - \xi_j \ \forall i, \\ \xi_j \geq 0, \ \forall j. \end{cases}$$

Note that $\xi_j > 1$, when SVMs make errors:

$$\xi_j = (1 - (\mathbf{w}\mathbf{x}_j + b)y_j)_+$$

Let's look at the new constraint. If some data points are misclassified, then $\xi_j > 1$ and $y_i(\mathbf{w}\cdot\mathbf{x}_i + b) \leq 0$. This approach is called **soft-margin SVM**. Lastly, how do we set C? A large value of C places a higher penalty on errors, leading to a narrower margin but fewer misclassifications (*i.e.,* the SVM will try to classify all data points correctly), whereas a smaller value of C allows for a wider margin but potentially more misclassifications. The optimal value of C is typically chosen through cross-validation.

11.4 SVM Optimization: Lagrange Multipliers

11.4.1 Lagrange Multipliers

Consider the optimization problem:

$$\begin{aligned} &\min_{\mathbf{x}} f(\mathbf{x}) \\ &\text{subject to} \quad g(\mathbf{x}) = 0. \end{aligned}$$

To find the minimum of f under the constraint $g(\mathbf{x})$, we use the method of *Lagrange multipliers*. The key idea is that at the optimal point, the gradient of $f(\mathbf{x})$ must be parallel to the gradient of $g(\mathbf{x})$. Mathematically, this condition is expressed as:

$$\nabla f(\mathbf{x}) = \lambda \nabla g(\mathbf{x}).$$

Example: Consider a simple 2D example where you want to minimize the function $f(x, y) = x^2 + y^2$, which represents a circle centered at the origin. This function increases as you move away from the origin, so the minimum is at the origin.

Now, consider the constraint: $g(x, y) = x + y - 1 = 0$. This constraint is a line that runs through the xy-plane. Our goal is to find the point on this line that minimizes $f(x, y)$.

A Lagrange multiplier is like a balancing factor that adjusts the direction and magnitude of your search along the constraint. As you move along the constraint line $g(x, y)$, λ ensures that the solution also respects the shape of the function $f(x, y)$ that you are trying to minimize. To solve the constraint optimization problem, we define the Lagrangian function:

$$\mathcal{L}(\mathbf{x}, \lambda) = f(\mathbf{x}) - \lambda g(\mathbf{x}).$$

To find the minimum, we take the partial derivatives of $\mathcal{L}(\mathbf{x}, \lambda)$ with respect to both $\mathbf{x}$ and λ, and set them equal to zero.

11.4.2 SVM Optimization

Recall that we want to solve the following convex quadratic optimization problem:

$$\begin{aligned} &\min \frac{1}{2}\|\mathbf{w}\|^2, \quad \text{subject to} \\ &y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 \quad \forall i. \end{aligned}$$

The objective is to find the optimal hyperplane that maximizes the margin between two classes of data points.

We can reformulate this optimization problem using the method of Lagrange multipliers, which introduces a set of multipliers α_i (one for each constraint). The Lagrangian function for this problem is given by:

$$\mathcal{L}(\mathbf{w}, b, \alpha) = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{N} \alpha_i \left[y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1\right]$$

11.4.3 Duality and the Lagrangian Problem

While we could attempt to solve the primal optimization problem directly, it is often more practical, especially for large datasets, to reformulate the problem using the duality principle. The dual form is advantageous because it depends only on the inner products of the data points, which allows the use of kernel methods for non-linear classification.

TO find the solution to the primal problem, we solve the following problem:

$$\max_{\mathbf{w},b} \min_{\alpha} \mathcal{L}(\mathbf{w}, b, \alpha)$$
$$\text{subject to} \quad \alpha_i \geq 0, \forall i.$$

Here, we maximize the Lagrangian with respect to the multipliers α_i, while minimizing with respect to the primal variables $\mathbf{w}$ and b.

11.4.4 Handling Inequality Constraints with KKT Conditions

You may observe that the method of Lagrange multipliers is used for equality constraints. However, it can be extended to handle inequality constraints through the use of additional conditions known as the **Karush-Kuhn-Tucker (KKT) conditions**. These conditions ensure that the solution satisfies the necessary optimality criteria for problems with inequality constraints. For more details on the KKT conditions, refer to **??**.

11.5 The Wolfe Dual Problem

The Lagrangian problem for SVM optimization involves N inequality constraints, where N is the number of training examples. This problem is typically tacked using its *dual form*. The duality principle provides a powerful framework, stating that **an optimization problem can be approached from two perspectives**:

1. The *primal problem*, which in our context is a minimization problem.
2. The *dual problem*, which is a maximization problem.

An important aspect of duality is that **the maximum value of the dual problem is always less than or equal to the minimum value of the primal problem.** This relationship means that the dual problem **provides a lower bound to the solution of the primal problem**.

In the context of SVM optimization, we are dealing with a convex optimization problem. According to **Slater's condition**, which applies to problems with affine constraints, strong duality holds. Strong duality implies that the optimal values of the primal and dual problems are equal, meaning the maximum value of the dual problem equals the minimum value of the primal problem. This equality allows us to solve the dual problem instead of the primal problem, often leading to computational advantages.

Recall that we aim to solve the following optimization problem:

$$\mathcal{L}(\mathbf{w}, b, \alpha) = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{N} \alpha_i \left[y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1\right]$$

The minimization problem involves solving the partial derivatives of $\mathcal{L}$ with respect to $\mathbf{w}$ and b and set them equal to zero:

$$\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}, b, \alpha) = \mathbf{w} - \sum_i \alpha_i y_i \mathbf{x}_i$$
$$\nabla_b \mathcal{L}(\mathbf{w}, b, \alpha) = -\sum_i \alpha_i y_i$$

Form the first equation, we obtain:

$$\mathbf{w} = \sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i$$

Next, we substitute the objective function with $\mathbf{w}$:

$$\begin{aligned}
\mathbf{W}(\alpha, b) &= \frac{1}{2}\left(\sum_i \alpha_i y_i \mathbf{x}_i\right) \cdot \left(\sum_j \alpha_j y_j \mathbf{x}_j\right) - \sum_i \alpha_i \left[y_i \left(\left(\sum_j \alpha_j y_j \mathbf{x}_j\right) \cdot \mathbf{x}_i + b\right) - 1\right] \\
&= \frac{1}{2}\Big(\sum_i \sum_j \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j\Big) - \sum_i \alpha_i \left[y_i \left(\Big(\sum_j \alpha_j y_j \mathbf{x}_j\Big) \cdot \mathbf{x}_i + b\right)\right] + \sum_i \alpha_i \\
&= \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j - \sum_i \sum_j \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j - \sum_i \alpha_i y_i b + \sum_i \alpha_i \\
&= \sum_i \alpha_i - \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j - \sum_i \alpha_i y_i b
\end{aligned}$$

Note that we use two indices, i and j when substituting $\mathbf{W}$. This is obvious if we consider a simple example. Imagine you have two data points:

$$\mathbf{x}_1, y1 = (1, 2), 1$$
$$\mathbf{x}_2, y2 = (2, 1), 1$$

Then,

$$\|\mathbf{w}\|^2 = \mathbf{w} \cdot \mathbf{w} = \underbrace{(\alpha_1 y_1 \mathbf{x}_1 + \alpha_2 y_2 \mathbf{x}_2)}_{\sum_i} \cdot \underbrace{(\alpha_1 y_1 \mathbf{x}_1 + \alpha_2 y_2 \mathbf{x}_2)}_{\sum_j}.$$

This simplification shows that the optimization problem can be reformulated purely in terms of the Lagrange multipliers α_i. Note that the term involving b can be removed by setting $b = 0$, simplifying our equation further:

$$\mathbf{W}(\alpha, b) = \sum_i \alpha_i - \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j) \tag{11.1}$$

This expression is known as the **Wolfe dual Lagrangian function**. We have transformed the problem into one involving only the multipliers α_i, resulting in a quadratic programming problem, commonly referred to as the **Wolfe dual problem**:

$$\max_{\alpha} \mathbf{W}(\alpha, b) = \sum_i \alpha_i - \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j)$$
$$\text{subject to } \alpha_i \geq 0 \text{ for any } i = 1, \ldots, m$$
$$\sum_{i=1}^{m} \alpha_i y_i = 0$$

Once we get the value of α, the optimal $\mathbf{w}$ and b can be computed using

$$\alpha_i \left[y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1 \right] = 0.$$

One important aspect of the dual problem is that it only involves the dot product of the input vectors $\mathbf{x}$. This property allows us to use of the **kernel trick** to handle non-linearly separable data by mapping it to a higher-dimensional space.

11.6 Karush-Kuhn-Tucker conditions

When dealing with optimization problems that involve inequality constraints, such as those encountered in Support Vector Machines (SVMs), an additional requirement must be met: the solution must satisfy the **Karush-Kuhn-Tucker (KKT) conditions**.

The KKT conditions are a set of first-order necessary conditions that must be satisfied for a solution to be optimal. These conditions extend the method of Lagrange multipliers to handle inequality constraints and are particularly useful in non-linear programming. For the KKT conditions to apply, the problem must also satisfy certain regularity conditions. Fortunately, one of these regularity conditions is Slater's condition, which we have already established holds true for SVMs.

11.6.1 KKT Conditions and SVM Optimization

In the context of SVMs, the optimization problem is convex, meaning that the KKT conditions are not only necessary but also sufficient for optimality. This implies that if a solution satisfies the KKT conditions, it is guaranteed to be the optimal solution for both the primal and dual problems. Moreover, in this case, there is no duality gap, meaning the optimal values of the primal and dual problems are equal.

Chapter 12

Least Square SVM

12.1 Introduction

Least Squares Support Vector Machine (LS-SVM) is a modified version of the traditional Support Vector Machine (SVM) that simplifies the quadratic optimization problem by using a *least squares cost function*. LS-SVM transforms the quadratic programming problem in classical SVM into a set of linear equations, which are easier and faster to solve.

12.1.1 Optimization Problem (Primal Problem)

$$\min_{w,b,e} \frac{1}{2}\|w\|^2 + \frac{\gamma}{2}\sum_{i=1}^{N} e_i^2,$$
$$\text{subject to } y_i(w^T\phi(x_i) + b) = 1 - e_i,\ \forall i$$

where:

- w is the weight vector.
- b is the bias term.
- e_i are the error variables.

$$\begin{aligned}\sum_{i=1}^{N} e_i^2 &= \sum_{i=1}^{N} \big(1 - y_i(w^T\phi(x_i) + b)\big)^2 \\ &= \sum_{i=1}^{N} \big(y_i^2 - y_i(w^T\phi(x_i) + b)\big)^2 \\ &= \sum_{i=1}^{N} y_i^2 \big(y_i - (w^T\phi(x_i) + b)\big)^2 \\ &= \sum_{i=1}^{N} \big(y_i - (w^T\phi(x_i) + b)\big)^2\end{aligned}$$

- γ is a regularization parameter.

- $\phi(x_i)$ is the feature mapping function.
- Note that $y_i^{-1} = y_i$, since $y_i = \pm 1$.

12.1.2 Lagrangian Function

To solve the constraint optimization problem, we define the Lagrangian function:

$$L(w, b, e, \alpha) = \min_{w,b,e} \frac{1}{2}\|w\|^2 + \frac{\gamma}{2}\sum_{i=1}^{N} e_i^2 - \sum_{i=1}^{n} \alpha_i \left[y_i(w^T\phi(x_i) + b) - 1 + e_i \right],$$

where α_i are Lagrange multipliers. Then, by setting the partial derivatives of the Lagrangian with respect to w, b, e, and α to zero, we get the KKT conditions.

- w:

$$\frac{\partial L}{\partial w} = w - \sum_{i=1}^{n} \alpha_i y_i \phi(x_i) = 0 \implies w = \sum_{i=1}^{n} \alpha_i y_i \phi(x_i)$$

- b:

$$\frac{\partial L}{\partial b} = -\sum_{i=1}^{n} \alpha_i y_i = 0$$

- e_i:

$$\frac{\partial L}{\partial e_i} = \gamma e_i - \alpha_i = 0 \implies \alpha_i = \gamma e_i$$

  Thus, $e_i = \frac{\alpha_i}{\gamma}$

- α_i:

$$\frac{\partial L}{\partial \alpha_i} = -\left[y_i(w^T\phi(x_i) + b) - 1 + e_i \right] = 0 \implies y_i(w^T\phi(x_i) + b) = 1 - e_i, i = 1, \ldots, N.$$

Let's substitute w and e:

- K: kernel matrix
- $\alpha = [\alpha_1, \alpha_2, \ldots, \alpha_n]^T$: $N \times 1$
- $y = [y_1, y_2, \ldots, y_n]^T$.
- $\Omega = YKY^T$, where $\Omega_{kl} = y_k y_l \phi(x_k)^T \phi(x_l)$
- b: 1×1

Then, we can express it compactly

$$Y(KY^T\alpha + b\mathbf{1}) - \mathbf{1} + \frac{\alpha}{2\gamma} = 0$$
$$\mathbf{1}^T Y\alpha = 0$$

By using the expression of α and b, we get

$$\begin{bmatrix} 0 & y^T \\ y & \Omega + \frac{1}{\gamma} I \end{bmatrix} \begin{bmatrix} b \\ \alpha \end{bmatrix} = \begin{bmatrix} 0 \\ 1_n \end{bmatrix}$$

Note that the dimension of the matrix on the left-hand side is $(N+1) \times (N+1)$. Once we have b and α by solving the linear system, the decision function for **a new input** x can be obtained by:

$$f(x) = \sum_{i=1}^{n} \alpha_i y_i K(x_i, x) + b.$$

Example: Suppose we have three training examples with feature vectors x_1, x_2, and x_3, and corresponding labels y_1, y_2, and y_3. The kernel matrix Ω is defined as:

$$\Omega_{ij} = y_i y_j K(x_i, x_j)$$

The dual form is:

$$\begin{bmatrix} 0 & y^T \\ y & \Omega + \frac{1}{\gamma} I \end{bmatrix} \begin{bmatrix} b \\ \alpha \end{bmatrix} = \begin{bmatrix} 0 \\ e \end{bmatrix}$$

- $y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$
- $\alpha = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix}$
- $e = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$
- I is a 3×3 identity matrix

Then, the Ω is given by

$$\Omega = \begin{bmatrix} y_1 y_1 K(x_1, x_1) & y_1 y_2 K(x_1, x_2) & y_1 y_3 K(x_1, x_3) \\ y_2 y_1 K(x_2, x_1) & y_2 y_2 K(x_2, x_2) & y_2 y_3 K(x_2, x_3) \\ y_3 y_1 K(x_3, x_1) & y_3 y_2 K(x_3, x_2) & y_3 y_3 K(x_3, x_3) \end{bmatrix}$$

Now, the complete matrix equation is:

$$\begin{bmatrix} 0 & y_1 & y_2 & y_3 \\ y_1 & \Omega_{11} + \frac{1}{\gamma} & \Omega_{12} & \Omega_{13} \\ y_2 & \Omega_{21} & \Omega_{22} + \frac{1}{\gamma} & \Omega_{23} \\ y_3 & \Omega_{31} & \Omega_{32} & \Omega_{33} + \frac{1}{\gamma} \end{bmatrix} \begin{bmatrix} b \\ \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

This can be written explicitly as:

$$\begin{bmatrix} 0 & y_1 & y_2 & y_3 \\ y_1 & y_1^2 K(x_1, x_1) + \frac{1}{\gamma} & y_1 y_2 K(x_1, x_2) & y_1 y_3 K(x_1, x_3) \\ y_2 & y_2 y_1 K(x_2, x_1) & y_2^2 K(x_2, x_2) + \frac{1}{\gamma} & y_2 y_3 K(x_2, x_3) \\ y_3 & y_3 y_1 K(x_3, x_1) & y_3 y_2 K(x_3, x_2) & y_3^2 K(x_3, x_3) + \frac{1}{\gamma} \end{bmatrix} \begin{bmatrix} b \\ \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

The solution to this matrix equation provides the values of b and $\alpha_1, \alpha_2, \alpha_3$, which are then used to construct the decision function:

$$f(x) = \sum_{i=1}^{3} \alpha_i y_i K(x, x_i) + b$$

12.2 Asymmetric Kernels

Recall that the dual form of LS-SVM is given by

$$\begin{bmatrix} 0 & y^T \\ y & \Omega + \frac{1}{\gamma} I \end{bmatrix} \begin{bmatrix} b \\ \alpha \end{bmatrix} = \begin{bmatrix} 0 \\ e \end{bmatrix}$$

An interesting point here is that using an asymmetric kernel in LS-SVM will not reduce to its symmetrization and asymmetric information can be learned. Then we can develop asymmetric kernels in the LS-SVM framework in a straightforward way.

Asymmetric kernels are particularly useful in capturing directional relationships in data that symmetric kernels cannot. For instance, in scenarios involving directed graphs or conditional probabilities, the relationship from x to y is inherently different from the relationship from y to x.

12.2.1 AsK-LS Primal Problem Formulation

We first define a generalized kernel trick for an inner product of two mappings ϕ_s and ϕ_t.

$$K(\mathbf{u}, \mathbf{v}) = \langle \phi_s(\mathbf{u}), \phi_t(\mathbf{v}) \rangle, \forall \mathbf{u} \in \mathbb{R}^{d_s}, \mathbf{v} \in \mathbb{R}^{d_t},$$

where $\phi_s : \mathbb{R}^{d_s} \to \mathbb{R}^p$, $\phi_t : \mathbb{R}^{d_t} \to \mathbb{R}^p$, and $\mathbb{R}^p$ is a high-dimensional or even an infinite-dimensional space. Note that d_s and d_t can be different.

The primal problem for AsK-LS is formulated to handle these asymmetric relationships. The goal is to find the weight vectors ω and ν, and bias terms b_1 and b_2, that minimize the following objective function:

$$\min_{\omega,\nu,b_1,b_2,e,h} \frac{1}{2}\omega^T\nu + \frac{\gamma}{2}\sum_{i=1}^{m} e_i^2 + \frac{\gamma}{2}\sum_{i=1}^{m} h_i^2,$$

subject to the constraints:

$$y_i(\omega^T \phi_s(x_i) + b_1) = 1 - e_i$$
$$y_i(\nu^T \phi_t(x_i) + b_2) = 1 - h_i$$

Here:

- ω and ν are weight vectors for the source and target features.
- $\phi_s(x)$ and $\phi_t(x)$ are the source and target feature mappings.
- e_i and h_i are error terms for the source and target constraints.
- γ is a regularization parameter.

Note that this formulation is almost the same as the LS-SVM except that this considers both the source and target feature spaces simultaneously.

12.2.2 Dual Form

Let's transform it into a *dual* form. The dual problem involves solving a system of linear equations derived from the primal problem's Lagrangian. The Lagrangian function for the primal problem is:

$$\mathcal{L}(\omega, \nu, b_1, b_2, e, h, \alpha, \beta) = \frac{1}{2}\omega^T\nu + \frac{\gamma}{2}\sum_{i=1}^{m} e_i^2 + \frac{\gamma}{2}\sum_{i=1}^{m} h_i^2 + \sum_{i=1}^{m} \alpha_i(1 - e_i - y_i(\omega^T\phi_s(x_i) + b_1)) + \sum_{i=1}^{m} \beta_i(1 - h_i - y_i(\nu^T\phi_t(x_i) + b_2))$$

The KKT conditions are derived by setting the partial derivatives of the Lagrangian with respect to ω, ν, b_1, b_2, e, and h to zero. The dual problem leads to the following linear system:

$$\begin{bmatrix} 0 & 0 & Y^T & 0 \\ 0 & 0 & 0 & Y^T \\ Y & 0 & \frac{I}{\gamma} & H \\ 0 & Y & H^T & \frac{I}{\gamma} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

where:

- Y is a vector of class labels.
- H is the kernel matrix with elements $H_{ij} = y_i K(x_i, x_j) y_j$, where $K(x_i, x_j) = \langle \phi_s(x_i), \phi_t(x_j) \rangle$ is the asymmetric kernel function.
  - For an asymmetric kernel K, the kernel function $K(x_i, x_j) \neq K(x_j, x_i)$. This asymmetry is directly incorporated into the matrix H, where:

$$H_{ij} = y_i K(x_i, x_j) y_j$$
$$H_{ji} = y_j K(x_j, x_i) y_i$$

AsK-LS uses two different feature mappings ϕ_s and ϕ_t for the source and target features. This approach allows capturing more information compared to symmetric kernels. The dual solution provides weight vectors ω and ν, which span the target and source feature spaces, respectively.

The decision functions for classification from the source and target perspectives are given by

$$f_s(x) = \sum \_i = 1^m \beta_i y_i K(x, x_i) + b_1$$
$$f_t(x) = \sum \_i = 1^m \alpha_i y_i K(x_i, x) + b_2$$

These functions utilize the learned asymmetric relationships in the data.

Part IV

Generative Modeling

Chapter 13

Introduction

13.1 KL Divergence

The KL divergence can be defined as follows:

$$D_{KL}(P||Q) = \mathbb{E}_{x\sim P}\left[\log \frac{P(X)}{Q(X)}\right]$$

13.1.1 Properties

- Non symmetric
- $D_{KL} \in [0, \infty]$
- In order for the KL divergence to be finite, the support of P needs to be in the support of Q.

13.1.2 Rewriting the Objective

$$\begin{aligned} D_{KL}(P||Q) &= \mathbb{E}_{x\sim P}\left[\log \frac{P(X)}{Q(X)}\right] \\ &= \mathbb{E}_{x\sim P}[-\log Q(X)] - \mathcal{H}(P(X)) \end{aligned}$$

- $\mathbb{E}_{x\sim P}[-\log Q(X)]$: Cross entropy
- $\mathcal{H}(P(X))$: Entropy of P

13.1.3 Forward and Reverse KL

Let's say there is a true distribution $P(X)$ with two modes and our approximation $Q(X)$ has one mode. Then,

- Forward KL: $D_{KL}(P||Q)$
- Reverse KL: $D_{KL}(Q||P)$

Forward KL: Mean-Seeking Behavior

$$\begin{aligned}\arg\min_\theta D_{KL}(P\|Q) &= \arg\min_\theta \mathbb{E}_{x\sim P}[-\log Q_\theta(X)] - \mathcal{H}(P(X)) \\ &= \arg\min_\theta \mathbb{E}_{x\sim P}[-\log Q_\theta(X)] \\ &= \arg\max_\theta \mathbb{E}_{x\sim P}[\log Q_\theta(X)]\end{aligned}$$

Intuition: x will be sampled from the distribution P, and its value will be estimated from Q. Thus, there will be higher chance that x will be sampled from a space with higher probablity in P. Therefore, Q_θ has to consider all modes, which have high probabilities.

To use the forward KL, we have to have an access to the true model $P(X)$ for sampling.

Reverse KL: Mode-Seeking Behavior

$$\begin{aligned}\arg\min_\theta D_{KL}(Q\|P) &= \arg\min_\theta \mathbb{E}_{x\sim Q_\theta}[-\log P(X)] - \mathcal{H}(Q_\theta(X)) \\ &= \arg\max_\theta \mathbb{E}_{x\sim Q_\theta}[\log P(X)] + \mathcal{H}(Q_\theta(X))\end{aligned}$$

Intuition: x will be sampled from the distribution Q, and its value will be estimated from P. Thus, there will be higher chance that x will be sampled from a space with higher probablity in Q. Therefore, to maximize the value, we need to focus on a single mode.

To use the reverse KL, we have to be able to evaluate the true model $P(X)$.

Chapter 14

Sampling Based Inference

14.1 Basic Sampling Methods

14.1.1 Inverse Transform Sampling

Inverse transform sampling is a basic method for pseudo-random number sampling, *i.e.,* for generating sample numbers at random from any probability distribution given its cumulative distribution function (CDF).

Assume that we already have a uniformly distributed random number generator, *e.g.,* np.random.randn()

1. Generate a random number $u \sim Unif[0, 1]$
2. Find the inverse of the desired CDF, $F_X^{-1}(x)$.
3. Compute $X = F_X^{-1}(u)$. The computed random variable X has distribution $F_X(x)$

However, it is **hard to compute the inverse of CDF** ($F_X(x)$)

- $F_X(x) : \mathbb{R} \mapsto [0, 1]$ is any CDF.
- CDF is a non-negative and non-decreasing (monotone) function that is continuous.
- Our objective is to simulate a random variable X distributed as F; that is, we want to simularte a X such that $P(X \leq x) = F(x)$.
- F is invertible since it is continuous and strictly increasing.

14.1.2 Ancestral Sampling

$$p(\mathbf{x}) = p(\mathbf{x}_1)p(\mathbf{x}_2|\mathbf{x}_1)p(\mathbf{x}_3|\mathbf{x}_2)\cdots$$

Sampling steps:

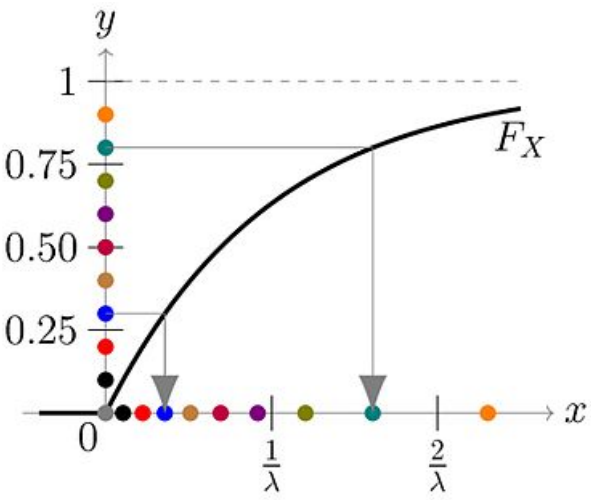


Figure 14.1: y-axis: Uniform distribution, x-axis: sample value

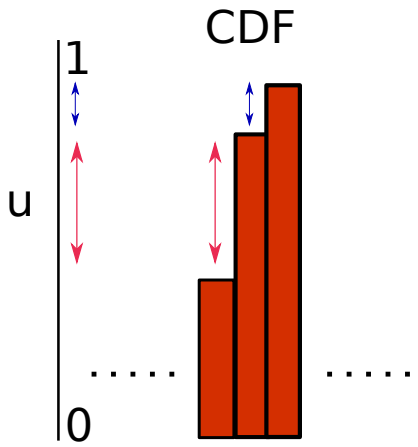


Figure 14.2: How can this sampling method recover the original distribution?

1. sample $\mathbf{x}_1$
2. sample $\mathbf{x}_2$ conditioned by $\mathbf{x}_1$
3. sample $\mathbf{x}_3$ conditioned by $\mathbf{x}_2$

14.1.3 Rejection Sampling

Rejection sampling is a simple method. It rejects samples violating a given condition (*e.g.*, conditions of conditional probability.). Let's see its theory.

Rejection sampling is a method for sampling from a distribution $p(x) = \frac{1}{Z}p'(x)$ that is difficulut to sample directly, but its unnormalized pdf $p'(x)$ is east to evaluate (Z is hard to compute). In rejection sampling, we need some simpler distribution $q(x)$, called a **proposal distribution**.

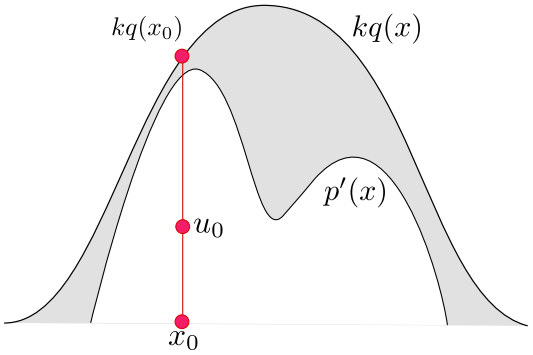
The intuition of rejection sampling is actually similar to Monte-Carlo estimation. By setting a large area (proposal distribution), we can sample points and take them that are inside the our

target distribution

To run the rejection sampling, introduce a constant k whose value is chosen such that $kq(x) \geq p'(x)$ for all values of x. The function $kq(x)$ is called a comparison function. Each step of the rejection sampler involves generating two random variables:

1. Sample $x_0 \sim q$
2. Sample $u_0 \sim U[0, kq(x_0)]$.

Finally, If $u_0 > p'(x_0)$, then the sample x_0 will be rejected, otherwise we add the sample x_0 to our set of samples $\{x^r\}$.



The original values of x are generated from the distribution $q(x)$ and these samples are then accepted with probability $p'(x)/kq(x)$ (see the figure above. The acceptance probability (*i.e.*, length) is the p' divided by kq). Then, the probability that a sample will be accepted is given by

$$\begin{aligned} p(accept) &= p\left(u \leq \frac{p'(x)}{kq(x)}\right) \\ &= \int p\left(u \leq \frac{p'(x)}{kq(x)}\middle| x\right) q(x)dx \\ &= \int \frac{p'(x)}{kq(x)} q(x)dx \\ &= \frac{1}{k}\int p'(x)dx \end{aligned}$$

Thus, the sampling will be more efficient if we choose small k to increase the change of acceptance.

14.1.4 Importance Sampling

We want to estimate an expectation of function $f(x)$, where $x \sim p(x)$, but it is hard to estimate the distribution $p(x)$. Again, the importance sampling is not a method for generating samples from $p(\mathbf{x})$. In this case, we can use a simple distribution $q(x)$ by

$$\begin{aligned}\mathbb{E}_p[f(\mathbf{x})] &= \int p(\mathbf{x})f(\mathbf{x})d\mathbf{x} \\ &= \int p(\mathbf{x})f(\mathbf{x})\frac{q(\mathbf{x})}{q(\mathbf{x})}d\mathbf{x} \\ &= \int q(\mathbf{x})\left[f(\mathbf{x})\frac{p(\mathbf{x})}{q(\mathbf{x})}\right]d\mathbf{x} \\ &= \mathbb{E}_q\left[f(\mathbf{x})\frac{p(\mathbf{x})}{q(\mathbf{x})}\right]\end{aligned}$$

- Assume that $p(\mathbf{x})$ is known and too complicated to be sampled directly.
- Samples are independently drawn from a **proposal density** $Q(\mathbf{x})$, which is designed to be close to the true density $p(\mathbf{x})$ and **simpler**
- Generate R samples from $Q(\mathbf{x})$

By applying the Monte-Carlo method, we can get

$$\mathbb{E}_q\left[f(\mathbf{x})\frac{p(\mathbf{x})}{q(\mathbf{x})}\right] \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x})\frac{p(\mathbf{x})}{q(\mathbf{x})}, \quad \mathbf{x}_i \sim p(\mathbf{x}) \tag{14.1}$$

- Unbiased estimation
- (Potentially) Smaller variance compared to the vanilla Monte-Carlo method above.
  - $Var_q\left[f(\mathbf{x})\frac{p(\mathbf{x})}{q(\mathbf{x})}\right] < Var_p[f(\mathbf{x})]$
  - When $q(\mathbf{x})$ is high where $|p(\mathbf{x})f(\mathbf{x})|$ is high.

Chapter 15

Markov Chain Monte Carlo

15.1 Gibbs Sampling

The phrase "Markov chain Monte Carlo" encompasses a broad array of techniques that have in common a few key ideas. The setup for all the techniques that we will discuss in this book is as follows:

1. We want to sample from a some complicated density or probability mass function π. Often, this density is the result of a Bayesian computation so it can be interpreted as a posterior density. The presumption here is that we can evaluate π but we cannot sample from it.
2. We know that certain stochastic processes called Markov chains will converge to a stationary distribution (if it exists and if specific conditions are satisfied). Simulating from such a Markov chain for a long enough time will eventually give us a sample from the chain's stationary distribution.
3. Given the functional form of the density π, we want to construct a Markov chain that has π as its stationary distribution.
4. We want to sample values from the Markov chain such that the sequence of values $\{x_n\}$ generated by the chain converges in distribution to the density π.

In order for all these ideas to make sense, we need to first go through some background on Markov chains. The rest of this chapter will be spent defining all these terms, the conditions under which they make sense, and giving examples of how they can be implemented in practice.

15.2 Markov Chain

Reference Link

A Markov chain is a stochastic process that evolves over time by transitioning into different states. The sequence of states is denoted by the collection $\{X_i\}$ and the transition between states is random, following the rule

Definition 1 *Let D be a finite set. A random process $X_1, X2, \ldots$ with values in D is called a Markov chain if*

$$P(X_t = x_{t+1}|X_t = x_t, \ldots, X_0 = x_0) = P(X_{t+1} = x_{t+1}|X_t = x_t)$$

We can think of X_t as a random state at time t, and the Markovian assumption is that the probability of transitioning from x_t to x_{t+1} only depends on x_t. In other words, the future state depends only on the present. Let p_{ij} be the probability of transitioning from state i to state j. A Markov chain can be defined by a transition probability matrix:

Definition 2 *The matrix $\mathbf{P} = (p_{ij})_{i,j} \in D$ is called the transition probability matrix.*

Thus, P is a $D \times D$ matrix, where $|D|$ denotes the cardinality of D, and the cell value p_{ij} is the probability of transitioning from state i to state j, and the rows of P must sum to one. We will restrict ourselves to time *homogeneous Markov chains*:

Definition 3 *A Markov chain is called time homogeneous if*

$$\mathbb{P}\{X_{t+1} = j \,|\, X_n = i\} = p_{ij}, \forall n.$$

It state that *the transition probabilities are not changing as a function of time.* Finally, let's introduce some useful notation for the initial state of the Markov chain. Let

$$\mathbb{P}_{x_0}\{\cdot\} \triangleq \mathbb{P}\{\cdot|X_0 = x_0\}.$$

For example, I will write $\mathbb{P}_a\{X_1 = b\}$ rather than $\mathbb{P}\{X_1 = b|X_0 = a\}$. This is because all the conditional probabilities depend on the initial state, and it the usual notation is cumbersome.

Consider a simple Markov chain modeling the weather. The weather has two states: rainy and sunny. Thus, $D = \{r, s\}$ and X_n is the "weather on day n". The Markov chain model is as follows. If today is rainy (r), tomorrow it is sunny (s) with probability p. If today it is sunny, tomorrow it is rainy with probability q. Then, transition matrix is given by

$$\mathbf{P} = \begin{bmatrix} 1-p & p \\ q & 1-q \end{bmatrix}.$$

The state diagram of the chain can be represented as follows: As a consequence of the Markovian

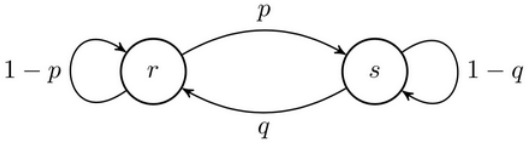


Figure 15.1: A sample Markov chain.

assumptions, the probability of any path on a Markov chain is just the multiplication of the

numbers along the path. For example, for some Markov chain with states $D = \{a, b, c, d\}$, the probability of a particular path, say $a \to b \to b \to d \to c$ factorize as

$$p_{ab} \cdots p_{dc}$$

Let's try to formulate this by

$$\mathbb{P}_i\{X_n = j\}.$$

For instance, let's say $n = 2$. Then, we have

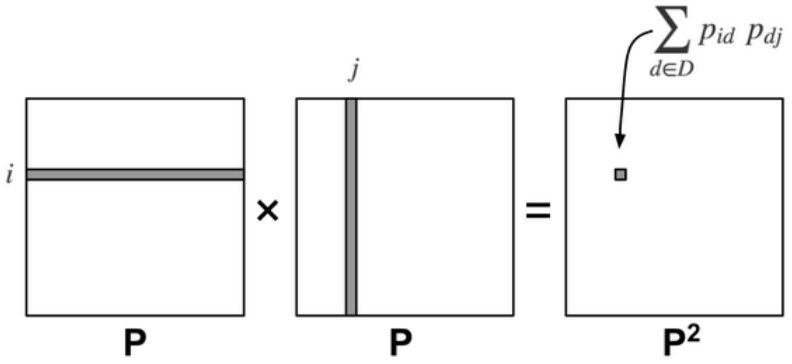
$$\begin{aligned}\mathbb{P}_i\{X_2 = j\} &= \sum_{d \in D} \mathbb{P}_i\{X_2 = j, X_1 = d\} \\ &= \sum_{d \in D} \mathbb{P}_i\{X_2 = j | X_1 = d\}\mathbb{P}_i\{X_1 = d\} \\ &= \sum_{d \in D} p_{id}p_{dj}\end{aligned}$$

This is equivalent to a dot product of the i-th row to j-th column of the transition matrix $\mathbb{P}$. This can be expressed as follows:

$$\mathbb{P}\{X_2 = j | X_0 = i\} = \sum_{d \in D} p_{id}p_{dj} = \left(\mathbf{P}^2_{ij}\right).$$

This can be generalized to

$$\mathbb{P}_i\{X_n = j\} = \left(\mathbf{P}^n_{ij}\right).$$



In sum, the dot product performs marginalization, and everything works out nicely thanks to the Markovian assumption. If the D-dimensional vector v represents a discrete distribution over initial states X_0, then $\mathbf{v^T P}$ is a D-dimensional vector representing the probability distribution over X_1.

15.2.1 Ergodicity

Let's discuss about *ergodicity*, which is a property of a random process in which its time average is the same as its probability space average. It can be defined as follows:

Definition 4 *A Markov chain $\{X_n\}$ is called ergodic if the limit*

$$\boldsymbol{\pi}(j) = \lim_{n \to \infty} \mathbb{P}_i\{X_n = j\}$$

exists for every state j and does not depend on the initial state i. The D-dimensional vector $\boldsymbol{\pi}(j)$ is called the stationary probability.

In other words,

- The probability $\boldsymbol{\pi}(j)$ of reaching at state j (*i.e.,* $\mathbb{P}_i\{X_n = j\}$)
- After a long time (*i.e.,* $\lim_{n\to\infty}$)
- Regardless of the initial state i (*i.e.,* $\mathbb{P}_i\{X_n = j\}$).

Equivalently, it can be expressed as

$$\boldsymbol{\pi}(j) = \lim_{n\to\infty} \left(\mathbf{P}^n\right)_{ij}.$$

The ergodicity gives the following property:

$$\begin{aligned}
\boldsymbol{\pi}(j) &= \lim_{n\to\infty} \left(\mathbf{P}^n\right)_{ij} \\
&\overset{\star}{=} \lim_{n\to\infty} \left(\mathbf{P}^{n+1}\right)_{ij} \\
&= \lim_{n\to\infty} \left(\mathbf{P}^n\mathbf{P}\right)_{ij} \\
&= \lim_{n\to\infty} \sum_{d\in D} \left(\mathbf{P}^n\right)_{id}\mathbf{P}_{dj} \\
&= \sum_{d\in D} \boldsymbol{\pi}(d)\mathbf{P}_{dj}
\end{aligned}$$

- The step $\star$ holds since the step n and $n+1$ does not matter under the limit.

Finally, we can write this as

$$\boldsymbol{\pi}^T = \boldsymbol{\pi}^T\mathbf{P},$$

where $\boldsymbol{\pi}$ is a column vector. Hence, the name "stationary probability distribution" denotes that it is a distribution that does not change over time.

Since we are interested in ergodicity, let's now introduce some properties related to reachability and long-term behavior.

Definition 5 *We say that there is a path from i to j ($i \rightsquigarrow j$) if there is a nonzero probability that starting at i, we can reach j at some point in the future.*

Definition 6 *A state i is called **transient** if there exists a state j such that $i \rightsquigarrow j$ but $j \not\rightsquigarrow i$.*

Definition 7 *A state i is called **recurrent** if for all states j there exist $i \rightsquigarrow j$ and $j \rightsquigarrow i$.*

Definition 8 *A Markov chain is **irreducible** if $i \leftrightarrow j, \forall i, j \in S$. Simply, if all states are able to visit other states, it is irreducible.*

Definition 9 *State i has a **period** d (*i.e., *periodically visit the state i) $\leftrightarrow$ aperiodic.*

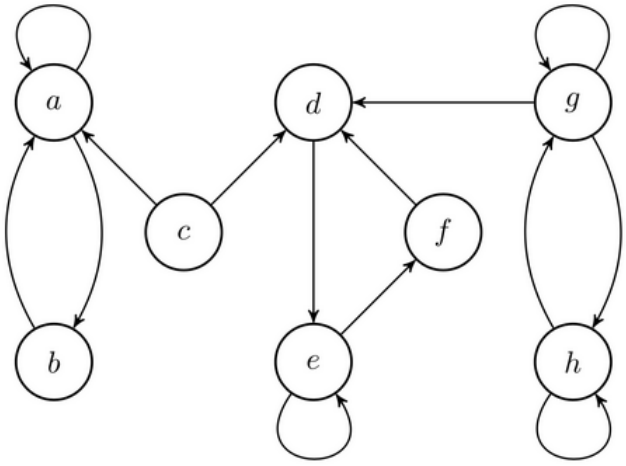


Figure 15.2: A Markov chain with $D = \{a, b, c, d, e, f, g, h\}$. The state c is a transient state. Note that g and h are also transient as there exist states that they cannot return. There are two recurrent classes: $\{a, b\}$ and $\{d, e, f\}$.

Intuitively, a *transient* state is a state that cannot return to itself; while a *recurrent* state can return. In other words, these two conditions are opposite. A transient state is not recurrent, and a recurrent state is not transient. Note that if a state is recurrent, every reachable state is also recurrent. We think of a set of recurrent states as a "class" or a "recurrent class". We can notice that Fig. 15.1 is ergodic but Fig. 15.2 is not since some states are not recurrent.

In sum a state is ergodic if the state is recurrent and aperiodic. Markov chain is ergodic if all states are ergodic.

15.2.2 Limit Theorem of Markov chain

Reference Link

For a Markov chain with a discrete state space and transition matrix P, let π_* be such that $\pi_* P = \pi_*$. Then π_* is a stationary distribution of the Markov chain and the chain is said to be stationary if it reaches this distribution.

The basic limit theorem for Markov chains says that, under a specific set of assumptions that we will detail below, we have

$$||\pi_* - \pi_n|| \to 0$$

as $n \to \infty$, where $|| \cdot ||$ is the total variation distance between the two densities. Therefore, no matter where we start the Markov chain (π_0), π_n will eventually approach the stationary distribution. Another way to think of this is that

$$\lim_{n \to \infty} \pi_n(i) = \pi_*(i).$$

for all states i in the state space. Note that π_0 is the probability distribution of the Markov chain at time 0. Also, π_n denote the distribution of the chain at time n.

15.2.3 Time Reversibility

Consider a stationary ergodic Markov chain with transition probability $p(i, j)$ and stationary distribution $\pi(i)$, if we reverse the process, we will get a reversed Markov chain with transition probability $q(i, j)$:

$$\begin{aligned} q(j,i) &= P(X_m = i | X_{m+1} = j) \\ &= \frac{P(X_m = i, X_{m+1} = j)}{P(X_{m+1} = j)} \\ &= \frac{P(X_m = i | X_{m+1} = j) P(X_{m+1} = j)}{P(X_{m+1} = j)} \\ &= \frac{\pi(i) p(i,j)}{\pi(j)} \\ \pi(i) p(i,j) &= \pi(j) q(j,i) \end{aligned}$$

If $p(i, j) = q(j, i)$, it is called time-reversible Markov chain.

15.3 Markov Chain Monte Carlo

MCMC aims to generate samples from some complex probability distribution $p(x)$ that is difficult to directly sample from.

The basic sampling methods we have learnt so far do not leverage past information, which assumes all samples are independent. In Markov chain based sampling, we will treat random variables as a sequence of sampling process.

In Markov Chain Monte Carlo(MCMC), we assume that a stationary distribution is already known. We are more interested in estimating a transition rule that describing the stationary distribution.

Ground rules for MCMCs:

- MCMCs stochastically explore the parameter space in such a way that the histogram of their samples produces the target distribution.
- Markovian: Evolution of the chain (*i.e.,* collections of samples from one iteration to the other) only depends on the current position and some transition probability distribution (*i.e.,* how we move from one point in parameter space to another). This means that the chain has no memory and past samples cannot be used to determine new positions in parameter space.
- The chain will converge to the target distribution if the transition probability is:
  - Irreducible: From any point in parameter space, we must be able to reach any other point in the space in a finite number of steps.
  - Positive recurrent: For any point in parameter space, the expected number of steps for the chain to return to that point is finite. This means that the chain must be able to re-visit previously explored areas of parameter space.
  - Aperiodic: The number of steps to return to a given point must not be a multiple of some value k. This means that the chain cannot get caught in cycles.

15.3.1 Metropolis-Hasting Algorithm

Suppose we have a target posterior distribution $\pi(x)$, where x here can be any collection of parameters (not a single parameter). In order to move around this parameter space we must formulate some proposal distribution:

$$q(x_{i+1} \mid x_i),$$

that specifies the probability of moving to a point in parameter space, x_{i+1}, given that we are currently at x_i. The Metropolis Hastings algorithm accepts a "jump" to x_{i+1} with the following probability

$$\kappa(x_{i+1} \mid x_i) = \min\left(1, \frac{\pi(x_{i+1})q(x_i \mid x_{i+1})}{\pi(x_i)q(x_{i+1} \mid x_i)}\right) = \min(1, H),$$

where the fraction above is called the Hastings ratio, H. The above expression represents that the probability of transitioning from point x_{i+1} given the current position x_i is a function of the ratio of the value of the posterior at the new point to the old point (*i.e.*, $\pi(x_{i+1})/\pi(x_i)$) and the ratio of the transition probabilities at the new point to the old point (*i.e.*, $q(x_i \mid x_{i+1})/q(x_{i+1} \mid x_i)$). Firstly, it is clear that if the ratio is bigger than 1 then the jump will be accepted. Secondly, the ratio of the target posteriors ensures that the chain will gradually move to high probability regions. Lastly, the ratio of the transition probabilities ensures that the chain is not "favored" toward certain locations by the proposal distribution function. Note that many proposal distributions are symmetric (*i.e.*, $q(x_{i+1} \mid x_i) = q(x_i \mid x_{i+1})$).

The Metropolis-Hasting algorithm is then:

```
def mh_sampler(x0, lnprob_fn, prop_fn, prop_fn_kwargs={}, iterations=100000):
    """Simple metropolis hastings sampler.

    :param x0: Initial array of parameters.
    :param lnprob_fn: Function to compute log-posterior.
    :param prop_fn: Function to perform jumps.
    :param prop_fn_kwargs: Keyword arguments for proposal function
    :param iterations: Number of iterations to run sampler. Default=100000

    :returns:
        (chain, acceptance, lnprob) tuple of parameter chain , acceptance rate
        and log-posterior chain.
    """

    # number of dimensions
    ndim = len(x0)

    # initialize chain, acceptance rate and lnprob
    chain = np.zeros((iterations, ndim))
    lnprob = np.zeros(iterations)
    accept_rate = np.zeros(iterations)

    # first samples
    chain[0] = x0
    lnprob0 = lnprob_fn(x0)
    lnprob[0] = lnprob0

    # start loop
    naccept = 0
    for ii in range(1, iterations):

        # propose
        x_star, factor = prop_fn(x0, **prop_fn_kwargs)

        # draw random uniform number
        u = np.random.uniform(0, 1)

        # compute hastings ratio
        lnprob_star = lnprob_fn(x_star)
        H = np.exp(lnprob_star - lnprob0) * factor

        # accept/reject step (update acceptance counter)
        if u < H:
            x0 = x_star
            lnprob0 = lnprob_star
            naccept += 1

        # update chain
        chain[ii] = x0
        lnprob[ii] = lnprob0
```

```
51          accept_rate[ii] = naccept / ii
52
53      return chain, accept_rate, lnprob
```

Chapter 16

Topic Modeling

16.1 Latent Semantic Allocation

Topic model provides insights like

- Can analyze topics. *e.g.*,, a certain topic includes specific words more than other topics.
- A certain document's topic distribution (simplex)

pLSA: Latent Variable Model

$$P_{LSA}(w|d) = \sum_z P(w|z;\theta)P(z|d;\pi)$$

- w: word
- d: document
- z: latent variable

Equivalently,

$$P_{LSA}(d,w) = \sum_z P(w|z)P(d|z)p(z) = p(d)\sum_z P(w|z)P(z|d)$$

The probability of observing $n(w_i, d_j)$ occurrences of word w_i in document d_j is given by

$$p(w_i, d_j)^{n(w_i,d_j)}$$

The probability of observing the complete document collection is then given by the product of probabilities of observing every single word in every document with corresponding number of occurrences.

Then, the likelihood function becomes

$$L = \prod_{i=1}^{m} \prod_{j=1}^{n} p(w_i, d_j)^n(w_i, d_j)$$

The log-likelihood is then

$$\begin{aligned}\mathcal{L} &= \sum_{i=1}^{m} \sum_{j=1}^{n} n(w_i, d_j) \log(p(w_i, d_j)) \\ &= \sum_{l=1}^{k} p(w_i|z_l)p(d_j|z_l)p(z_l)\end{aligned}$$

Parameter Inference:

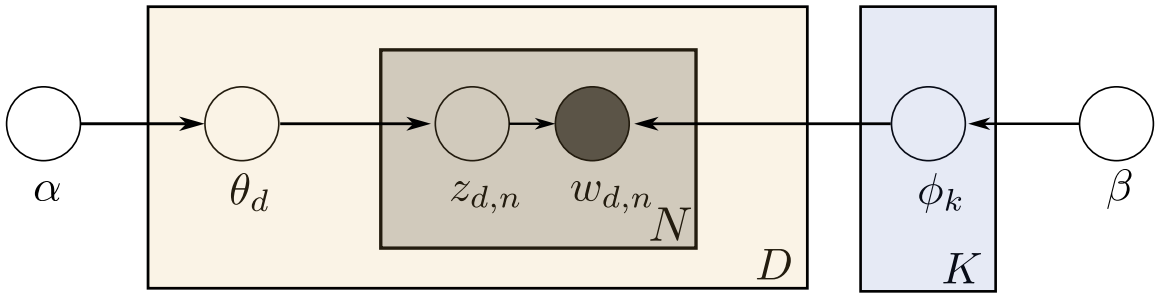
- We cannot maximize the likelihood analytically because of the logarithm of the sum
- A standard procedure is to use *EM*.

16.2 Latent Dirichlet Allocation

The assumptions of LDA:

- Each topic is a distribution over words.
- Each document is a mixture of corpus-wide topics.
- Each word is sampled from one of topics.

The LDA attempts to model the document generation process stochastically. However, we have to infer the latent structure (the distributions) of documents.



- $\theta_d \sim Dir(\alpha)$: For each document, draw topic distribution.
  - α: Dirichlet parameter
- $z_{d,n} \sim Mult(\theta_d)$: per-word topic assignment. The n-th word of document d is from which topic?
- $w_{d,n} \sim Mult(\phi_{z_{d,n}}, n)$: observed word. The n-th word in a document d is from a certain topic ($z_{d,n}$) distribution $\phi_{z_{d,n}}$.

- $\phi_k \sim Dir(\beta), i = \{1, \ldots, K\}$: topics.
  - β: topic hyperparameter (Dirichlet parameter).

We can immediately notice that the LDA's modeling approach is quite far from the way we write texts. However, LDA works quite well.

The document generation process can be modelled as follows:

$$p(\phi_{1:K}, \theta_{1:D}, z_{1:D}, w_{1:D}) = \prod_{i=1}^{K} p(\phi_i|\beta) \prod_{d=1}^{D} p(\theta_d|\alpha) \left(\prod_{n=1}^{N} p(w_{d,n}|\phi_{1:K}, z_{d,n}) p(z_{d,n}|\theta_d) \right).$$

16.2.1 LDA Inference

The posterior of the latent variables given the document is

$$p(\phi, \theta, \mathbf{z}|\mathbf{w}) = \frac{p(\phi, \theta, \mathbf{z}, \mathbf{w})}{\int_\phi \int_\theta \sum_{\mathbf{z}} p(\phi, \theta, \mathbf{z}, \mathbf{w})}$$

Computing the posterior is intractable:

- The denominator is intractable
- We cannot compute the denominator, the marginal p(w).
- We are going to use collapsed Gibbs sampling.

We want to estimate the topic distribution **z**.

16.2.2 Dirichlet Distribution

The Dirichlet Distribution can be considered as a extension of the *beta distribution*.

$$p(P = \{p_i\}|\alpha_i) = \frac{\Gamma(\sum_i \alpha_i)}{\prod_i \Gamma(\alpha_i)} \prod_i p_i^{\alpha_i - 1} \tag{16.1}$$

- $\sum_i p_i = 1$
- The posterior distribution of Dirichlet distribution is also Dirichlet distribution.

Chapter 17

Latent Variable Models

17.1 Motivation of Latent Variable Models

Let's say we want to classify some data. If we had access to a corresponding latent variable for each observation $\mathbf{x}_i$, modeling would be more straightforward. To illustrate this, consider the challenge of finding the latent variable (*i.e.,* the true class of $\mathbf{x}$) $z^* = \text{argmax}_z\, p(\mathbf{x}|z)$, as shown in Fig. 17.1(b).

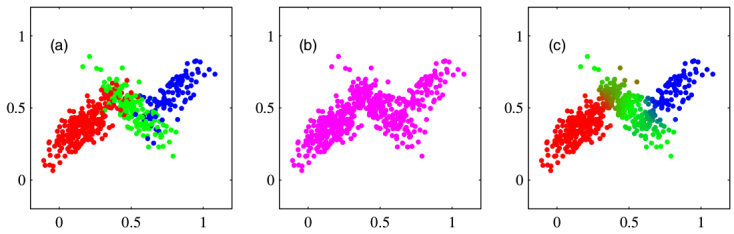


Figure 17.1: (a) Complete dataset $p(\mathbf{x}|z)$. (b) Incomplete dataset $p(\mathbf{x})$. (c) Inference result.

Consider modeling the complete data set $p(\mathbf{x}|z)$ under the assumption that the observations are independent and identically distributed (i.i.d.). Based on the Fig. 17.1(a), the joint distribution for a single observation $(\mathbf{x}_i, \mathbf{z}_i)$ given the model parameters $\boldsymbol{\theta}$ can be expressed:

$$p(\mathbf{x}_i, \mathbf{z}_i|\boldsymbol{\theta}) = \begin{cases} p(\mathcal{C}_1)p(\mathbf{x}_i|\mathcal{C}_1) & \text{if } z_i = 0 \\ p(\mathcal{C}_2)p(\mathbf{x}_i|\mathcal{C}_2) & \text{if } z_i = 1 \\ p(\mathcal{C}_3)p(\mathbf{x}_i|\mathcal{C}_3) & \text{if } z_i = 2 \end{cases}$$

Given N observations, the joint distribution for the entire dataset $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$ along with their corresponding latent variables $\{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_N\}$ is:

$$p(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N, \mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_N|\boldsymbol{\theta}) = \prod_{n=1}^{N}\prod_{k=1}^{K} \pi_k^{z_{nk}} \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)^{z_{nk}}$$

Here, $\pi_k = p(\mathcal{C}_k)$ represents the prior probability of the k-th component, and $p(\mathbf{x}_n|\mathcal{C}_k) =$

$\mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ denotes the Gaussian distribution associated with component $\mathcal{C}_k$.

However, in practice, the latent variables $\mathbf{z}_k$ are often not directly observable, which complicates the modeling process.

In the following sections, we present various methods for identifying and handling these latent variables to improve the classification and modeling of data.

Chapter 18

Clustering

18.1 K-Means Clustering

Suppose that we have a data set $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ consisting of N observations of a random D-dimensional variable $\mathbf{x} \in \mathbb{R}^D$. Our goal is to partition the data into K of clusters. Intuitively, a cluster can be thought as **a group of data points whose inter-point distances are small compared with the distances to points outside of the cluster**.

This notion can be formalized by introducing a set of D-dimensional vectors $\boldsymbol{\mu}_k$, which represents the centers of the clusters. Our goal is to find an assignment of data points to clusters, as well as a set of vectors $\{\boldsymbol{\mu}_k\}$. Objective function of K-means clustering (*distortion measure*) can be defined as follows:

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} ||\boldsymbol{x}_n - \boldsymbol{\mu}_k||^2$$

, where $r_{nk} \in \{0, 1\}$ is a binary indicator variable which represents the **membership of data** $\mathbf{x}_n$. It can be expressed as follows:

$$r_{nk} = \begin{cases} 1 & \text{if } k = \text{argmin}_j \, ||\boldsymbol{x}_n - \boldsymbol{\mu}_j||^2 \\ 0 & \text{otherwise} \end{cases}$$

Our goal is to find values for the $\boldsymbol{\mu}_k$ and the r_{nk} that minimize J.

We can minimize J through an iterative procedure in which each iteration involves two successive steps corresponding to successive optimizations with respect to the $\boldsymbol{\mu}_k$ and the r_{nk} First we choose some initial values for the $\boldsymbol{\mu}_k$. Then in the first phase we minimize J with respect to the r_{nk}, keeping the $\boldsymbol{\mu}_k$ fixed. In the second phase we minimize J with respect to the $\boldsymbol{\mu}_k$, keeping r_{nk} fixed. This two-stage optimization is then repeated until convergence.

Now consider the optimization of the $\boldsymbol{\mu}_k$ with the r_{nk} held fixed. The objective function J is a quadratic function of $\boldsymbol{\mu}_k$, and it can be minimized by setting its derivative with respect to $\boldsymbol{\mu}_k$ to zero giving

$$2 \sum_{n=1}^{N} r_{nk} (\boldsymbol{x}_n - \boldsymbol{\mu}_k) = 0.$$

We can arrange as

$$\boldsymbol{\mu}_k = \frac{\sum_n r_{nk}\boldsymbol{x}_n}{\sum_n r_{nk}}.$$

The denominator of $\boldsymbol{\mu}_k$ is equal to the number of points assigned to cluster k. The mean of cluster k is essentially the same as the mean of data points $\mathbf{x}_n$ assigned to cluster k. For this reason, the procedure is known as the *K-means clustering algorithm.*

The two phases of re-assigning data points to clusters and re-computing the cluster means are repeated in turn until there is no further change in the assignments. These two phases reduce the value of the objective function J, so the convergence of the algorithm is assured. However, it may converge to a local rather than global minimum of J.

We can also sequentially update the μ_k as follows:

$$\mu_{k+1} = \mu_k + \eta(\mathbf{x}_k - \mu_k)$$

There are some properties to note:

- It is a hard clustering algorithm ($\leftrightarrow$ soft clustering)
- It is sensitive to the initialization of centroid.
- The number of clusters is uncertain.
- Sensitive to distance metrics (*e.g.,* Euclidean?)

18.2 Gaussian Mixture Models

K-means clustering is a form of hard clustering, where each data point is assigned to exactly one cluster. However, in some cases, soft clustering—where data points can belong to multiple clusters with varying degrees of membership—provides a better model in practice. A Gaussian Mixture Model (GMM) assumes a linear superposition of Gaussian components, offering a richer class of density models than a single Gaussian distribution.

In essence, rather than assuming that all data points are generated by a single Gaussian distribution, we assume that the data is generated by a mixture of K different Gaussian distributions, where each Gaussian represents a different component in the mixture.

For a single sample, the Gaussian Mixture Model can be expressed as a weighted sum of these individual Gaussian distributions:

$$\begin{aligned} p(\mathbf{x}) &= \sum_{\mathbf{z}} p(\mathbf{z}) p(\mathbf{x}|\mathbf{z}) \\ &= \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \end{aligned}$$

Here, $\mathbf{x}$ is a data point, π_k represents the mixing coefficients, $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ is a Gaussian distribution with mean $\boldsymbol{\mu}_k$ and covariance $\boldsymbol{\Sigma}_k$, and K is the number of Gaussian components.

A key quantity in GMMs is the conditional probability of $\mathbf{z}$ given $\mathbf{x}$, denoted as $p(z_k = 1|\mathbf{x})$ or $\gamma(z_k)$. This is also known as the responsibility or assignment probability, which represents the probability that a given data point $\mathbf{x}$ belongs to component k of the mixture. Essentially, this can be thought of as the **classification result** for $\mathbf{x}$.

This responsibility is updated using Bayes' Theorem, and can be expressed as:

$$\begin{aligned} \gamma(z_k) \equiv p(z_k = 1|\mathbf{x}) &\equiv \frac{p(z_k = 1) p(\mathbf{x}|z_k = 1)}{\sum_{j=1}^{K} p(z_j = 1) p(\mathbf{x}|z_j = 1)} \\ &= \frac{\pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)} \end{aligned}$$

In this expression, π_k is the prior probability (or mixing coefficient) for component k, and $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ is the likelihood of the data point $\mathbf{x}$ under the Gaussian distribution corresponding to component k. The denominator is a normalization factor that ensures the responsibilities sum to 1 across all components for a given data point.

This framework allows for a soft classification of data points, where each point is associated with a probability of belonging to each cluster, rather than being strictly assigned to a single cluster as in K-means.

18.2.1 Maximum Likelihood

Suppose we have a data set of observations $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}^T \in \mathbb{R}^{N \times D}$ and we want to model the data distribution $p(\mathbf{X})$ using GMM. Assuming the data is independent and identically distributed

(i.i.d.), the likelihood of the entire dataset can be expressed as:

$$p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \prod_{n=1}^{N} \left(\sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right).$$

To simplify the optimization process, we consider the log-likelihood function, which is given by:

$$\ln p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{n=1}^{N} \ln \left(\sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right)$$

To solve the Maximum Likelihood Estimation (MLE) for Gaussian Mixture Models (GMMs), we typically consider the iterative *Expectation-Maximization* (EM) algorithm due to the non-convex nature of the problem. Before discussing how to maximize the likelihood, it is important to emphasize two significant issues that arise in GMMs: *singularities* and *identifiability*.

Singularity A major challenge in applying the maximum likelihood framework to Gaussian Mixture Models is the presence of singularities. This problem arises because the likelihood function can become unbounded under certain conditions, leading to an ill-posed optimization problem.

For simplicity, consider a Gaussian mixture model where each component has a covariance matrix of the form $\Sigma_k = \sigma_k^2 I$, where I is the identity matrix. Suppose one of the mixture components, say the j-th component, has its mean $\boldsymbol{\mu}_j$ exactly equal to one of the data points $\mathbf{x}_n$, so that $\boldsymbol{\mu}_j = \mathbf{x}_n$ for some value of n. The contribution of this data point to the likelihood function would then be:

$$\mathcal{N}(\mathbf{x}_n|\mathbf{x}_n, \sigma_j^2 I) = \frac{1}{\sqrt{2\pi}\sigma_j} \cdot \exp^0$$

As σ_j approaches 0, this term goes to infinity, causing the log-likelihood function to also diverge to infinity. Therefore, maximizing the log-likelihood function becomes an ill-posed problem because such singularities can always be present. These singularities occur whenever one of the Gaussian components **collapses** onto a specific data point, leading to a covariance matrix with a determinant approaching zero. This issue did not arise with a single Gaussian distribution because the variance cannot be zero by definition.

Identifiability Another issue in finding MLE solutions for GMMs is related to identifiability. For any given maximum likelihood solution, a GMM with K components has a total of $K!$ equivalent solutions. This arises from the fact that the $K!$ different ways of permuting the K sets of parameters (means, covariances, and mixing coefficients) yield the same likelihood.

In other words, for any point in the parameter space that represents a maximum likelihood solution, there are $K! - 1$ additional points that produce exactly the same probability distribution. This lack of identifiability means that the solution is not unique, complicating both the interpretation of the model and the optimization process.

18.2.2 Expectation Maximization for GMM

The goal of the Expectation-Maximization (EM) algorithm is to find maximum likelihood solutions for models that involve latent variables.

- Suppose that directly optimizing the likelihood $p(\mathbf{X}|\boldsymbol{\theta})$ is difficult.
- However, it is easier to optimize the complete-data likelihood function $p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})$ as as discussed in the previous sections.
- In such cases, we can use the **EM algorithm**. EM algorithm is a general technique for finding maximum likelihood solutions in latent variable models.

Let's begin by writing down the conditions that must be satisfied at a maximum of the likelihood function. By setting the derivatives of $\ln p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma})$ with respect to the means $\boldsymbol{\mu}_k$ of the Gaussian components to zero, we obtain

$$0 = -\sum_{n=1}^{N} \frac{\pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)} \boldsymbol{\Sigma}_k(\mathbf{x}_n - \boldsymbol{\mu}_k)$$

Multiplying by $\boldsymbol{\Sigma}_k^{-1}$ (which we assume to be non-singular) and rearranging we obtain

$$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) \mathbf{x}_n,$$

where we have defined

$$N_k = \sum_{n=1}^{N} \gamma(z_{nk}).$$

We can interpret N_k as the effective number of points assigned to cluster k. We can obtain the MLE solutions for other variables similarly.

Algorithm 2: EM algorithm for GMM

Initialize the means $\boldsymbol{\mu}_k$, covariances $\boldsymbol{\Sigma}_k$ and mixing coefficients π_k and evaluate the initial value of the log likelihood.

for n **do**

E-step: evaluate the responsibilities of $\mathbf{x}_n$ based on the current parameter values with the given parameters

$$\gamma(z_{nk}) = p(z_k = 1|\mathbf{x}_n) = \frac{\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}$$

where z_{nk} denote the k-th component of $\mathbf{z}_n$

M-step: maximize expectation

- $\boldsymbol{\mu}_k^{\text{new}} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) \mathbf{x}_n$
- $\boldsymbol{\Sigma}_k^{\text{new}} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk})(\mathbf{x}_n - \boldsymbol{\mu}_k^{\text{new}})(\mathbf{x}_n - \boldsymbol{\mu}_k^{\text{new}})^T$
- $\pi_k^{\text{new}} = p(z_k = 1) = \frac{N_k}{N}$

Evaluate the log likelihood to check for convergence of parameters

$$\ln p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{n=1}^{N} \ln \left(\sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right)$$

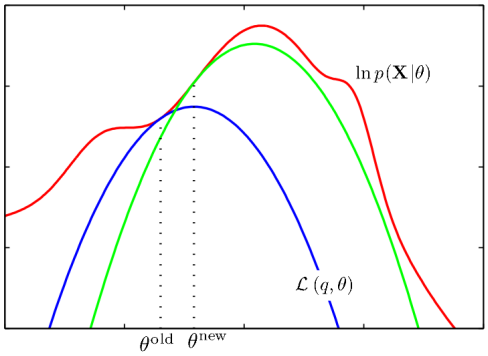


Figure 18.1: M-step of EM algorithm

18.3 Alternative View of EM

The goal of the EM algorithm is to find maximum likelihood (loglikelihood) solutions for models having latent variables.

$$\ln p(X|\theta) = ln \sum_Z p(X, Z|\theta).$$

We are not given the complete data set X, Z, but only the incomplete data X. Our state of knowledge of the values of the latent variables in Z is given only by the posterior distribution $p(Z|X, \theta)$. Because we cannot use the complete-data log likelihood function, we consider instead its expected value under the posterior distribution of the latent variable, which corresponds (as we shall see) to the E-step of the EM algorithm.

In the subsequent M-step, we maximize this expectation. If the current estimate for the parameters is denoted θ_{old}, then a pair of successive E- and M-steps gives rise to a revised estimate θ^{new}.

The algorithm is initialized by choosing some starting value for the parameters θ_0. The use of the expectation may seem somewhat arbitrary.

In the E-step, we use the current parameter values θ^{old} to find the posterior distribution of the latent variables given by $p(Z|X, \theta^{old})$. We then use this posterior distribution to find the **expectation of the complete-data log likelihood function** evaluated for some general parameter value θ. In other words, this is a expectation over a some function. This expectation, denoted $Q(\theta, \theta^{old})$, is given by

$$Q(\theta, \theta^{old}) = \sum_Z p(Z|X, \theta^{old}) \ln p(X, Z|\theta).$$

In the M-step, we determine the revised parameter estimate θ^{new} by maximizing this function

$$\theta^{new} = \operatorname{argmax} Q(\theta, \theta^{old}).$$

Algorithm 3: General EM algorithm

The goal is to maximize the likelihood function $p(X|\theta)$ with respect to θ given a joint distribution $p(X, Z|\theta)$.

1. Init θ^{old}
2. E-Step: evaluate $p(Z|X, \theta^{old})$
3. M-Step: evaluate θ^{new} given by

$$\theta^{new} = \operatorname{argmax} Q(\theta, \theta^{old}),$$

where

$$Q(\theta, \theta^{old}) = \sum_{Z} p(Z|X, \theta^{old}) \ln p(X, Z|\theta).$$

4. Check for convergence of either the log likelihood or the parameter values. If the convergence criterion is not satisfied, then let

$$\theta^{old} \leftarrow \theta^{new}.$$

Return to the step 2.

18.4 Latent Variable Modeling

For each object x_i, we establish additional latent variable z_i which denotes the index of Gaussian from which i-th object was generated. Then our model is

$$p(X, Z|\theta) = \prod_{i=1}^{n} p(x_i, z_i|\theta) = \prod_{i=1}^{n} p(x_i|z_i, \theta)p(z_i|\theta) = \prod_{i=1}^{n} \mathcal{N}(x_i|\mu_{z_i}, \sigma_{z_i}^2)\pi_{z_i},$$

where $\pi_j = p(z_i = j)$ are prior probability of j-th gaussian and $\theta = \{\mu_j, \sigma_j, \pi_j\}_{j=1}^{K}$. If we know both X and Z then we can obtain explicit ML-solution:

$$\theta_{ML} = \underset{\theta}{\operatorname{argmax}}\, p(X, Z|\theta) = \underset{\theta}{\operatorname{argmax}} \log p(X, Z|\theta).$$

However, in practice, we don't know Z, but only know X. Thus, we need to maximize w.r.t. θ the log of incomplete likelihood

$$\begin{aligned}
\log p(X|\theta) &= \ln \int p(X, Z|\theta) dZ && (18.1)\\
&= \ln \int q(Z|X) \frac{p(X, Z|\theta)}{q(Z|X)} dZ && (18.2)\\
&\geq \underbrace{\int q(Z|X) \ln \frac{p(X, Z|\theta)}{q(Z|X)} dZ}_{\text{ELBO, } \mathcal{L}(q,\theta)} \quad \text{by Jensen's Inequality.} && (18.3)\\
&= \int q(Z|X) \ln p(X, Z|\theta) - q(Z|X) \ln q(Z|X) dZ && (18.4)\\
&= \int q(Z|X)[\ln p(X|Z, \theta) + \ln p(Z|\theta)] - q(Z|X) \ln q(Z|X) dZ && (18.5)\\
&= \int q(Z|X) \ln p(X|Z, \theta) - q(Z|X) \ln \frac{q(Z|X)}{p(Z|\theta)} dZ && (18.6)\\
&= \mathbb{E}_{q(Z|X)} \ln p(X|Z, \theta) - KL\left(q(Z|X) \| p(Z|\theta)\right) && (18.7)
\end{aligned}$$

To maximize the above equation, we need to minimize KL divergence.

18.4.1 Evidence Lower Bound (ELBO)

For any choice of inference model $q_\phi(z|x)$, we can represent the marginal probability of data (or model evidence) distribution, since the z is not related to x, so the integration does not affect x. Thus, we can also derive ELBO as follows:

$$\begin{aligned}
\log p_\theta(x) &= \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x)]\\
&= \mathbb{E}_{q_\phi(z|x)}\left[\log \frac{p_\theta(x, z)}{p_\theta(z|x)}\right]\\
&= \mathbb{E}_{q_\phi(z|x)}\left[\log \frac{p_\theta(x, z) q_\phi(z|x)}{q_\phi(z|x) p_\theta(z|x)}\right]\\
&= \underbrace{\mathbb{E}_{q_\phi(z|x)}\left[\log \frac{p_\theta(x, z)}{q_\phi(z|x)}\right]}_{=\mathcal{L}(\phi,\theta)(x)} + \underbrace{\mathbb{E}_{q_\phi(z|x)}\left[\log \frac{q_\phi(z|x)}{p_\theta(z|x)}\right]}_{=D_{KL}(q_\phi(z|x)||p_\theta(z|x))}
\end{aligned}$$

To get more intuition about ELBO, we can express ELBO as follows:

$$\begin{aligned}
\mathcal{L}(\phi,\theta) &= \mathbb{E}_{q_\phi(z|x)}\left[\log \frac{p_\theta(x,z)}{q_\phi(z|x)}\right] \\
&= \mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x,z) - \log q_\phi(z|x)\right] \\
&= \mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x) + \log p_\theta(z|x) - \log q_\phi(z|x)\right] \\
&= \log p_\theta(x) - D_{\mathrm{KL}}(q_\phi(z|x)||p_\theta(z|x)) \\
&\leq \log p_\theta(x)
\end{aligned}$$

ELBO can be also written as follows:

$$\begin{aligned}
\mathcal{L}(\phi,\theta) &= \mathbb{E}_{q_\phi(z|x)}\left[\log \frac{p_\theta(x,z)}{q_\phi(z|x)}\right] \\
&= \mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x,z) - \log q_\phi(z|x)\right] \\
&= \mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(z) + \log p_\theta(x|z) - \log q_\phi(z|x)\right] \\
&= \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{\mathrm{KL}}(q_\phi(z|x)||p_\theta(z))
\end{aligned}$$

We can get a conclusion that maximizing ELBO is equivalent to minimizing the KL divergence through the above equation. Fianlly, the log-likelihood can be rewritten as follows:

$$\log p_\theta(x) = \mathcal{L}(\phi,\theta) + D_{\mathrm{KL}}(q_\phi(z|x)||p_\theta(z|x))$$

18.4.2 Expectation Maximization

We want to maximize ELBO, $\mathcal{L}(q,\theta)$ to minimize KL divergence between $q(Z)$ and $\log p(Z|X,\theta)$.

$$\max_{q,\theta} \mathcal{L}(q,\theta) = \max_{q,\theta} \int q(Z) \log \frac{p(X,Z|\theta)}{q(Z)} dZ.$$

We start from initial point θ_0 and iteratively repeat (*i*) E-step and (*ii*) M-step, iteratively:

- E-Step: θ_0 is fixed.

$$q(Z) = \operatorname*{argmax}_q \mathcal{L}(q,\theta) = \operatorname*{argmin}_q \mathrm{KL}(q(Z)|p(Z|X,\theta)) = p(Z|X,\theta_0).$$

  - This is because, maximizing ELBO is equal to minimizing KL divergence and the minimum q can be achieved when q is equal to $p(Z|X,\theta_0)$.
  - Now, we just have to evaluate $p(Z|X,\theta_0)$.

- M-Step: q is fixed.

$$\theta_* = \operatorname*{argmax}_\theta \mathcal{L}(q,\theta) = \operatorname*{argmax}_\theta \mathbb{E}_{q(Z)}[\log p(X,Z|\theta)]$$

  - Can be accomplished by taking derivatives
  - Set $\theta_0 = \theta_*$ and go to the E-Step until convergence

18.4.3 Categorical Latent Variables

$z_i \in \{1, ..., K\}$

$$p(x_i|\theta) = \sum_{k=1}^{K} p(x_i|k, \theta)p(z_i = k|\theta)$$

is simply a finite mixture of distributions.

E-Step:

$$q(z_i = k) = p(z_i = k|x_i, \theta) = \frac{p(x_k|z_i = k, \theta)p(z_i = k|\theta)}{\sum_{l=1}^{K} p(x_i|z_i = l, \theta)p(z_i = l|\theta)}$$

M-Step:

$$\underset{\theta}{\operatorname{argmax}}\, \mathbb{E}_{q(Z)}[\log p(X, Z|\theta)] = \sum_{i=1}^{n} \mathbb{E}_{q(z_i)}[\log p(x_i, z_i|\theta)] = \sum_{i=1}^{n}\sum_{k=1}^{K} q(z_i = k) \log p(x_i, k|\theta)$$

For GMM, we model $p(x|z)$ as Gaussian.

Chapter 19

Hidden Markov Models

19.1 Introduction

The HMM is based on the Markov chain assumption. A Markov chain is a model that tells us something about the probabilities of sequences of random variables, states, each of which can take on values from some set. These sets can be words, or tags, or symbols representing anything, like the weather.

There are two important assumptions:

- Markov assumption
- Output independence: $p(x_i|z_1, \ldots, z_i, \ldots, z_T, x_1, \ldots, x_i, \ldots, x_T) = p(x_i|z_i)$

19.1.1 Conditional Independence

If two events A and B are **conditionally independent** given an event C then,

- $P(A \cap B|C) = P(A|C)P(B|C)$.
- $P(A|B, C) = P(A|C)$

19.1.2 Notation

- $X = (x_i, x_2, \ldots, x_T)$
- Initial state probabilities: $p(z_1) \sim \text{Multinomial}(\pi_1, ..., \pi_k)$, need to learn π
- Transition probability:

  $$p(z_t|z_{t-1} = i) \sim \text{Multinomial}(a_{i,1}, ..., a_{i,k})$$

  , where $a_{i,j} = p(z_t = j|z_{t-1} = i)$ and i and j denote clusters or states, respectively.
- Emission probability:

  $$p(x_t|z_t = i) \sim \text{Multinomial}(b_{i,1}, ..., b_{i,m})$$

  , where $b_{i,j} = p(x_t = j|z_t = i)$

19.2 Bayesian Network

19.2.1 Bayes Ball

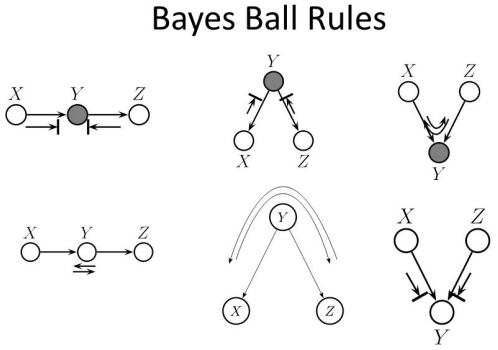


Figure 19.1: Bayes ball

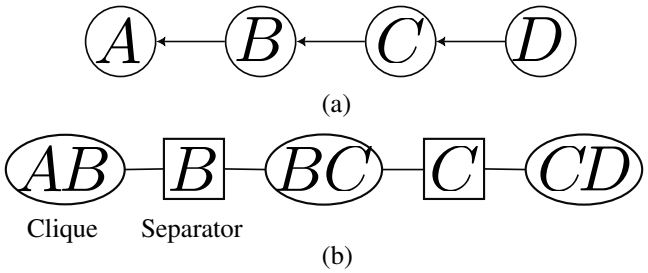
- Cascading: $P(Z|Y,X) = P(Z|Y)$. The information of Y decouples X and Z.
- Common parent: $P(X,Z|Y) = P(X|Y)P(Z|Y)$. The information of Y decouples X and Z.
- V-Structure (common child): Unlike the above two cases, the information of Y couples X and Z.

$$P(X,Y,Z) = P(X)P(Y)P(Y|X,Z).$$

19.2.2 Potential Function

Potential function is a function which is not a probability function, but it can become a probability function by normalizing it.

$$P(A,B,C,D) = P(A|B)P(B|C)P(C|D)P(D)$$



- Cliques: $\Psi(a,b), \Psi(b,c), \Psi(c,d)$
- Separators $\phi(b), \phi(c)$

Given a clique tree with cliques and separators, the joint probability distribution is defined as follows:

$$P(A,B,C,D) = P(U) = \frac{\prod_N \Psi(N)}{\prod_L \phi(L)} = \frac{\Psi(a,b)\Psi(b,c)\Psi(c,d)}{\phi(b)\phi(c)}$$

An effect of an observation propagates through the clique graph → **Belief propagation**. How to propagate the belief? **Absorption rule**!

Let's say we have some new observations about A, then it affects the clique $\Psi(a,b)$. The updated clique is now $\Psi^*(a,b)$. Similarly, $\phi^*(b) = \sum_A \Psi^*(a,b)$. Subsequently, $\Psi^*(b,c) = \Psi(b,c)\frac{\phi^*(b)}{\phi(b)}$.

19.3 Hidden Markov Models

Hidden Markov Models (HMMs) are a powerful statistical tool used for modeling sequences of observable events that are believed to be generated by underlying hidden states. The core idea behind HMMs is that the system being modeled can be represented as a **Markov process with unobservable (hidden) states z, where each state emits observable outputs x according to specific probability distributions**.

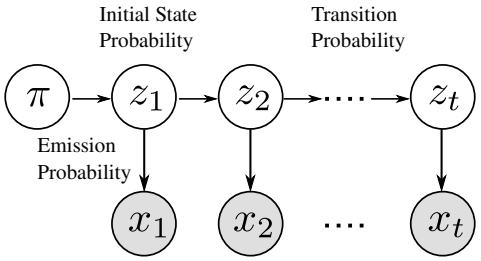


Figure 19.2: HMM Structure

In an HMM, the hidden states are connected by transition probabilities, which dictate the likelihood of moving from one state to another. Meanwhile, each state is associated with an emission probability, which defines the likelihood of observing a particular output given that state. The challenge and beauty of HMMs lie in their ability to infer the most probable sequence of hidden states that could have produced a given sequence of observations, even when the underlying states are not directly observable.

Through algorithms such as the Forward-Backward algorithm, the Viterbi algorithm, and the Baum-Welch algorithm, HMMs provide a framework for decoding sequences, estimating model parameters, and predicting future events based on past observations. In this post, we'll explore the fundamentals of Hidden Markov Models, their key components, and how they can be applied to solve real-world problems.

It's important to note that the observations in an HMM can be either discrete or continuous. When the latent factors are continuous, the model is often referred to as a *Kalman filter*.

19.3.1 Key Components of HMM

The key components of an HMM can be listed as follows:

- Initial state probability: $P(z_1) \sim \text{Mult}(\pi_1, \ldots, \pi_k)$. In other words, z is belong to one of k classes.
- Transition probability: $P(z_t|z^i_{t-1} = 1) \sim \text{Mult}(a_{i,1}, \ldots, a_{i,k})$,
  where $P(z^j_t = 1|z^i_{t-1} = 1) = a_{i,j}$
- Emission probability: $P(x_t|z^i_t = 1) \sim \text{Mult}(b_{i,1}, \ldots, b_{i,m}) \sim f(x_t|\theta_i)$,
  where $P(x^j_t = 1|z^i_t = 1) = b_{i,j}$. The probability of observing x_j at the i-th cluster.

Note that i and j are indices of clusters (*e.g.,* classes).

There are three main problems in HMM:

1. Evaluation Questions (likelihood):
   - Given $\boldsymbol{\pi}, \mathbf{a}, \mathbf{b}, X$
   - Find $p(X|M, \boldsymbol{\pi}, \mathbf{a}, \mathbf{b})$
   - How much are X likely to be observed by a model M?
2. Decoding Questions:
   - Given $\boldsymbol{\pi}, \mathbf{a}, \mathbf{b}, X$
   - Find $\text{argmax}_Z\, p(Z|X, M, \boldsymbol{\pi}, \mathbf{a}, \mathbf{b})$
   - What is the most probable sequence of Z (latent states)?
3. Learning Questions: Forward-Backward (Baum-Welch)
   - Given X
   - Find $\text{argmax}_{\boldsymbol{\pi},\mathbf{a},\mathbf{b}}\, p(X|M, \boldsymbol{\pi}, \mathbf{a}, \mathbf{b})$
   - What would be the optimal model parameters?

19.4 Evaluation: Forward-Backward Probability

19.4.1 Joint Probability

We can factorize the joint distribution of HMM in Fig. 19.2 by using a Bayesian approach as follows:.

$$p(X, Z) = p(x_1, \ldots, x_t, z_1, \ldots, z_t) \tag{19.1}$$
$$= p(z_1)p(x_1|z_1)p(z_2|z_1), \ldots, p(x_t|z_t)p(z_t|z_{t-1}) \tag{19.2}$$

The key assumption involved in factorizing the Markov chain within a Hidden Markov Model (HMM) is *conditional independence* among certain components of the state variables. Here's a detailed breakdown of what this assumption means:

- Independence of State Components: The transition of each component z_t^k only depends on its corresponding previous component z_{t-1}^k and is independent of other components.

As the number of latent factor increases, it is getting harder to decode the latent factors.

19.4.2 Marginal Probability

We want to compute the likelihood of sequence X which is given by

$$p(X|\boldsymbol{\pi}, \mathbf{a}, \mathbf{b}) = \sum_Z p(X, Z|\boldsymbol{\pi}, \mathbf{a}, \mathbf{b})$$

The computation can be done as follows:

$$\begin{aligned} p(X) &= \sum_Z p(X, Z) \\ &= \sum_{z_1} \cdots \sum_{z_t} p(x_1, \ldots, x_t, z_1, \ldots, z_t) \\ &= \sum_{z_1} \cdots \sum_{z_t} \pi_{z_1} \prod_{t=2}^{T} a_{z_{t-1}, z_t} \prod_{t=1}^{T} b_{z_t, x_t} \end{aligned}$$

The last step is done by using the factorization above. The computation of this equation requires lots of computations, so we will change it into a **recursive form** by using the factorization rule

$p(a, b, c) = p(a)p(b|a)p(c|a, b)$.

$$p(x_1, \ldots, x_t, z_t^k = 1) = \sum_{z_{t-1}} p(x_1, \ldots, x_{t-1}, x_t, z_{t-1}, z_t^k = 1) \tag{19.3}$$

$$= \sum_{z_{t-1}} p(\underbrace{x_1, \ldots, x_{t-1}, z_{t-1}}_{a}, \underbrace{x_t}_{c}, \underbrace{z_t^k = 1}_{b}) \tag{19.4}$$

$$= \sum_{z_{t-1}} p(x_1, \ldots, x_{t-1}, z_{t-1}) p(z_t^k = 1 | x_1, \ldots, x_{t-1}, z_{t-1}) p(x_t | z_t^k = 1, x_1, \ldots, x_{t-1}, z_{t-1}) \tag{19.5}$$

$\because p(a, b, c) = p(a)p(b|a)p(c|a, b)$ or by the structure of HMM

$$= \sum_{z_{t-1}} p(x_1, \ldots, x_{t-1}, z_{t-1}) p(z_t^k = 1 | z_{t-1}) p(x_t | z_t^k = 1) \tag{19.6}$$

$$= p(x_t | z_t^k = 1) \sum_{z_{t-1}} p(x_1, \ldots, x_{t-1}, z_{t-1}) p(z_t^k = 1 | z_{t-1}) \tag{19.7}$$

$$= b_{z_t^k, x_t} \sum_{z_{t-1}} p(x_1, \ldots, x_{t-1}, z_{t-1}) a_{z_{t-1}, z_t^k} \tag{19.8}$$

- In the second line, the x_{t-1} and z_{t-1} are grouped together.
- Then, we can find the HMM structure by factorizing the equation.
- In the fourth line, x terms are removed, since z_t only relies on z_{t-1} by the Markov assumption. Similarly, x_t only depends on z_t. We can interpret this by using Bayes ball too.

Now we can find a recursive structure of $p(x_1, \ldots, x_t, z_t^k = 1)$ as follows:

$$\alpha_t^k = p(x_1, \ldots, x_t, z_t^k = 1) = b_{k, x_t} \sum_i \alpha_{t-1}^i a_{i,k}$$

, where α_t^k **is the probabilities of being in state** k **after observing the first** t **observations.** Thus,

$$\begin{aligned} p(x_1, \ldots, x_t) &= \sum_{\mathbf{z}} p(x_1, \ldots, x_t, z) \\ &= \sum_k \alpha_t^k \end{aligned}$$

Note that α_t^k is also called **Forward probability**.

19.4.3 Forward Algorithm

Forward probability solves the evaluation problem. Essentially, this is a dynamic programming, so it calculates required values in a bottom-up manner.

- Forward probability: α_t^k, $Time \times States$

Note again that

$$p(X) = p(x_1, ..., x_T) = \sum_i \alpha_T^i = \sum_i p(x_1, ..., x_T, z_T^i = 1)$$

Note also that the forward-algorithm returns $p(X)$ and forward probability is the probability of being in state k after observing the first t observations without Z.

Algorithm 4: Forward Algorithm

Create a probability matrix $forward[M, T] = \alpha_t^k$
Initialization:
for *each state k=1,...,M* **do**
  $\alpha_1^k \leftarrow \pi_k b_{k,x_1}$
for *time step t=2,...,T* **do**
  for *each step k=1,...,M* **do**
    $\alpha_t^k = b_{k,x_t} \sum_i \alpha_{t-1}^i a_{i,k}$
Return $p(X) = \sum_i^M \alpha_T^i$

19.4.4 Backward Probability

The forward probability only considers an observation at t. To determine the z_t, we need to leverage the future observations. **The backward probability β is the probability of seeing the observations from time $t+i$ to the end, given that we are in state k at time t.**

$$\beta_t^k = p(x_{t+1}, \ldots, x_T | z_t^k = 1)$$

We want to compute $p(z_t^k = 1|X)$ rather than $p(x_1, \ldots, x_t, z_t^k = 1)$. In other words, we will leverage the whole observations X.

$$\begin{aligned} p(z_t^k = 1, X) &= p(x_1, \ldots, x_t, z_t^k = 1, x_{t+1}, \ldots, x_T) \\ &= p(x_1, \ldots, x_t, z_t^k = 1) p(x_{t+1}, \ldots, x_T | x_1, \ldots, x_t, z_t^k = 1) \\ &= p(x_1, \ldots, x_t, z_t^k = 1) p(x_{t+1}, \ldots, x_T | z_t^k = 1) \\ &= \alpha_t^k \beta_t^k \end{aligned}$$

We already know that $p(x_1, \ldots, x_t, z_t^k = 1) = \alpha_t^k$. We just need to compute backward probability as follows:

$$\begin{aligned} \beta_t^k &= p(x_{t+1}, \ldots, x_T | z_t^k = 1) \\ &= \sum_{z_{t+1}} p(\underbrace{z_{t+1}}_{a}, \underbrace{x_{t+1}}_{b}, \underbrace{x_{t+2}, \ldots, x_T}_{c} | z_t^k = 1) \\ &= \sum_i p(z_{t+1}^i = 1 | z_t^k = 1) p(x_{t+1} | z_{t+1}^i = 1, z_t^k = 1) p(x_{t+2}, \ldots, x_T | x_{t+1}, z_{t+1}^i = 1, z_t^k = 1) \\ &\because p(a, b, c) = p(a) p(b|a) p(c|a, b) \\ &= \sum_i p(z_{t+1}^i = 1 | z_t^k = 1) p(x_{t+1} | z_{t+1}^i = 1) p(x_{t+2}, \ldots, x_T | z_{t+1}^i = 1) \\ &= \sum_i a_{k,i} b_{i,x_{t+1}} \beta_{t+1}^i \end{aligned}$$

Another recursive structure:

$$\begin{aligned} p(z_t^k = 1, X) &= \alpha_t^k \beta_t^k \\ &= b_{k,x_t} \sum_i \alpha_{t-1}^i a_{i,k} \times \sum_i a_{k,i} b_{i,x_t} \beta_{t+1}^i \end{aligned}$$

This means at time t, the latent label is belong to some class k and this can be computed by using the forward probability and the backward probability. Now we can compute

$$p(z_t^k = 1|X) = \frac{p(z_t^k = 1, X)}{p(X)} = \frac{\alpha_t^k \beta_t^k}{p(X)}$$

Then,

$$k_t = \underset{k}{\operatorname{argmax}}\, p(z_t^k = 1|X)$$

Note that this is for a single latent variable at a single time step given the whole observation X, but we want to decode a sequence of latent variables. Thus, we need some decoding algorithm.

19.5 Decoding: Viterbi Algorithm

For any model, such as an HMM, that contains hidden variables, **the task of determining which sequence of variables is the underlying source of some sequence of observations is called the decoding task**.

We might propose to find the best sequence as follows:

1. For each possible hidden state sequence (HHH, HHC, HCH, etc.), we could run the forward algorithm and compute the likelihood of the observation sequence given that hidden state sequence.
2. Then, we could choose the hidden state sequence with the maximum observation likelihood.

However, this is not a feasible solution, because there are an exponentially large number of state sequences.

Instead, the most common decoding algorithms for HMMs is the **Viterbi algorithm**. Like the forward algorithm, **Viterbi** is a kind of **dynamic programming algorithm.**

Note that the Viterbi algorithm is identical to the forward algorithm except that it takes the **max** over the previous path probabilities whereas the forward algorithm takes the **sum**. This is because, we want to obtain **the most probable latent variable sequence**. Note also that the Viterbi algorithm has one component that the forward algorithm doesn't have: **backpointers**. The reason is that while the forward algorithm needs to produce an observation likelihood, the Viterbi algorithm must produce a probability and also the most likely state sequence. We compute this best state sequence by keeping track of the path of hidden states that led to each state and then at the end backtracing the best path to the beginning (the Viterbi backtrace).

We can leverage the forward-backward probabilities:

- $k^* = \text{argmax}_k \, p(z_t^k = 1|X) = \text{argmax}_k \, p(z_t^k = 1, X) = \text{argmax}_k \, \alpha_t^k \beta_t^k$

We will use a forward approach:

$$
\begin{aligned}
V_t^k &= \max_{z_1,\ldots,z_{t-1}} p(x_1, \ldots, x_{t-1}, z_1, \ldots, z_{t-1}, x_t, z_t^k = 1) && (19.9)\\
&= \max_{z_1,\ldots,z_{t-1}} p(x_t, z_t^k = 1|x_1, \ldots, x_{t-1}, z_1, \ldots, z_{t-1}) p(x_1, \ldots, x_{t-1}, z_1, \ldots, z_{t-1}) && (19.10)\\
&= \max_{z_1,\ldots,z_{t-1}} p(x_t, z_t^k = 1|z_{t-1}) p(x_1, \ldots, x_{t-2}, z_1, \ldots, z_{t-2}, x_{t-1}, z_{t-1}) && (19.11)\\
&= \max_{z_{t-1}} p(x_t, z_t^k = 1|z_{t-1}) \max_{z_1,\ldots,z_{t-2}} p(x_1, \ldots, x_{t-2}, z_1, \ldots, z_{t-2}, x_{t-1}, z_{t-1}) && (19.12)\\
&= \max_{i \in z_{t-1}} p(x_t, z_t^k = 1|z_{t-1}^i = 1) V_{t-1}^i && (19.13)\\
&= \max_{i \in z_{t-1}} p(x_t|z_t^k = 1) p(z_t^k = 1|z_{t-1}^i = 1) V_{t-1}^i && (19.14)\\
&= p(x_t|z_t^k = 1) \max_{i \in z_{t-1}} p(z_t^k = 1|z_{t-1}^i = 1) V_{t-1}^i && (19.15)\\
&= b_{k,x_t} \max_{i \in z_{t-1}} a_{i,k} V_{t-1}^i && (19.16)
\end{aligned}
$$

- V_t^k is Viterbi variable which denotes the probability that the HMM is in state k at t after observing the first t observations and $t-1$ latent variables. In another words, this is the probability of most likely sequence of states ending at state $z_t = k$.
- The first line assumes that the observation at time t and the latent variable are fixed and also the fourth line has the recursive structure.
- The third step, only z_{t-1} can affect the z_t, so we can remove all other unnecessary variables.
- The step six can be derived by the HMM structure.
- $i \in z_{t-1}$ simply denotes the index of potential cluster at $t-1$.
- We have already computed the backward and the forward probabilities. So we just need to apply the Viterbi algorithm.

Note that Also note that we present the most probable path by taking the maximum over all possible previous state sequences $\max_{z_1,\ldots,z_{t-1}}$. Like other DP-algorithm, Viterbi fills each cell recursively.

Algorithm 5: Viterbi Algorithm

$V_t^k = viterbi[M, T]$, where M is the number states
for *k=1,...,M* **do**
  $V_1^k \leftarrow \pi_{z_k} b_{k,x_1}$
  $backpointer[k, 1] \leftarrow 0$
for *t=2,...,T* **do**
  for *k=1,...,M* **do**
    $V_t^k \leftarrow b_{k,x_t} \max_{k'} V_t^{k'} a_{k',k}$, where k' is the previous state.
    $backpointer[k, t] \leftarrow b_{k,x_t} \operatorname{argmax}_{k'} V_t^{k'} a_{k',k}$
$bestpathprob \leftarrow \max_k V_T^k$ //termination step
$bestpathpointer \leftarrow \operatorname{argmax}_k V_T^k$ //termination step
$bestpath \leftarrow$ the path starting at state *bestpathpointer*, that follows backpointer[] to states back in time
Return *bestpathpointer*, *bestpathprob*

Viterbi algorithm typically shows some technical issues:

- Underflow problems $\rightarrow \log V$.

19.6 Learning: Baum-Welch Algorithm

We have to learn HMM parameters with only X. Baum-Welch algorithm or Forward-Backward Algorithm is a standard training algorithm for HMM. The algorithm let us train both the transition and the emission probabilities of the HMM. If we do not have the information about Z, then we can assign the most probable Z given X.

- Given X, estimate parameters π, a, b.
- Then, find the most probable Z given the parameters.

We will use EM algorithm!

19.6.1 EM Algorithm

$$P(X|\theta) = \sum_Z P(X, Z|\theta) \rightarrow \ln P(X|\theta) = \ln \sum_Z P(X, Z|\theta).$$

We cannot directly estimate the log-likelihood function, so we will estimate the expectation of it.

$$\begin{aligned} Q(\theta, \theta^{old}) &= \mathbb{E}_Z \ln P(X, Z|\theta) \\ &= \sum_Z p(Z|X, \theta^{old}) \ln P(X, Z|\theta) \\ &= \sum_Z p(Z|X, \pi^t, a^t, b^t) \ln P(X, Z|\pi, a, b). \end{aligned}$$

Note that $p(X, Z) = \pi_{z_1} \prod_{t=2}^T a_{z_{t-1}, z_t} \prod_{t=2}^T b_{z_t, x_t}$. Thus, $\ln p(X, Z) = \ln \pi_{z_1} + \sum_{t=2}^T \ln a_{z_{t-1}, z_t} + \sum_{t=1}^T \ln b_{z_t, x_t}$. Therefore

$$Q(\theta, \theta^{old}) = \sum_Z p(Z|X, \theta^{old}) \bigg(\ln \pi_{z_1} + \sum_{t=2}^T \ln a_{z_{t-1}, z_t} + \sum_{t=1}^T \ln b_{z_t, x_t} \bigg).$$

To optimize the above function we will use the Lagrange method as follows:

$$\mathcal{L}(\pi, a, b) = Q(\theta, \theta^{old}) - \lambda_\pi \bigg(\sum_{i=1}^K \pi_i - 1 \bigg) - \sum_i^K \lambda_{a_i} \bigg(\sum_{j=1}^K a_{i,j} - 1 \bigg) - \sum_i^K \lambda_{b_i} \bigg(\sum_{j=1}^K b_{i,j} - 1 \bigg).$$

The constraints are for forcing the sum of each probability is equal to 1.

Now, take a partial derivative for each parameter. Let's take a derivative with regard to π_i first. Then,

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \pi_i} &= \frac{\partial Q(\theta, \theta^{old})}{\partial \pi_i} - \lambda_\pi \\ &= \frac{\partial}{\partial \pi_i} \sum_Z p(Z|X, \theta^{old}) \ln \pi_{z_1} - \lambda_\pi \\ &= \frac{p(z_1^i = 1|X, \theta^{old})}{\pi_i} - \lambda_\pi \\ \frac{\partial \mathcal{L}}{\partial \lambda_{\pi_i}} &= \sum_{i=1}^K \pi_i - 1 = 0 \rightarrow \sum_{i=1}^K \pi_i = 1. \end{aligned}$$

By setting the derivative is equal to zero,

$$\pi_i = \frac{p(z_1^i = 1|X, \theta^{old})}{\lambda_\pi}.$$

By using the constraint of π, the Lagrange multiplier λ_π must be a normalizer.

$$\pi_i = \frac{p(z_1^i = 1|X, \theta^{old})}{\sum_{j=1}^{K} p(z_1^j = 1|X, \theta^{old})}.$$

Similarly, we can compute other parameters too.

$$a_{i,j}^{t+1} = \frac{\sum_{t=2}^{T} p(z_{t-1}^i = 1, z_t^j = 1|X, \theta^{old})}{\sum_{t=2}^{T} p(z_{t-1}^i = 1|X, \theta^{old})}.$$
$$b_{i,j}^{t+1} = \frac{\sum_{t=1}^{T} p(z_{t1}^i = 1|X, \theta^{old}) I(x_t = j)}{\sum_{t=1}^{T} p(z_t^i = 1|X, \theta^{old})},$$

where $I(x)$ is an indicator function which returns 1 if x is true and 0, otherwise.

19.7 Python Implementation

19.7.1 Viterbi Algorithm

The Viterbi algorithm is a dynamic programming algorithm used to determine the most probable sequence of hidden states in a Hidden Markov Model (HMM) based on a sequence of observations.

The algorithm works by recursively computing the probability of the most likely sequence of hidden states that ends in each state for each observation.

At each time step, the algorithm computes the probability of being in each state and emits the current observation based on the probabilities of being in the previous states and making a transition to the current state.

Assuming we have an HMM with N hidden states and T observations, the Viterbi algorithm can be summarized as follows:

Initialization: At time t=1, we set the probability of the most likely path ending in state i for each state i to the product of the initial state probability pi and the emission probability of the first observation given state i. This is denoted by: delta[1,i] = pi * b[i,1]. Recursion: For each time step t from 2 to T, and for each state i, we compute the probability of the most likely path ending in state i at time t by considering all possible paths that could have led to state i. This probability is given by:

$$delta[t, i] = max_j(delta[t-1, j] * a[j, i] * b[i, t])$$

Here, a[j,i] is the probability of transitioning from state j to state i, and b[i,t] is the probability of observing the t-th observation given state I.

We also keep track of the most likely previous state that led to the current state i, which is given by:

$$psi[t, i] = argmax_j(delta[t-1, j] * a[j, i])$$

- Termination: The probability of the most likely path overall is given by the maximum of the probabilities of the most likely paths ending in each state at time T. That is, $P* = max_i(delta[T, i])$.
- Backtracking: Starting from the state $i*$ that gave the maximum probability at time T, we recursively follow the psi values back to time $t = 1$ to obtain the most likely path of hidden states.

The Viterbi algorithm is an efficient and powerful tool that can handle long sequences of observations using dynamic programming.

19.8 Summary

- Forward-probability: probability of being in state k after observing the first t observations.

$$\alpha_t^k = p(x_1, ..., x_t, z_t^k = 1)$$

- Backward-probability: probability of observations from time $t+1$ to the end, given that we are in state k

$$\beta_t^k = p(x_{t+1}, ..., x_T | z_t^k = 1)$$

- These two sets of probability distributions can then be combined to obtain the distribution over states at any specific point in time given the entire observation sequence

$$\begin{aligned} p(z_t^k = 1, X) &= p(x_1, ..., x_t, z_t^k = 1, x_{t+1}, ..., x_T) \\ &= p(x_1, ..., x_t, z_t^k = 1) p(x_{t+1}, ..., x_T | x_1, ..., x_t, z_t^k = 1) \\ &= p(x_1, ..., x_t, z_t^k = 1) p(x_{t+1}, ..., x_T | z_t^k = 1) \\ &= \alpha_t^k \beta_t^k \end{aligned}$$

  In short, if we know the forward and backward probability, we could know the cluster of state at time t given our observations.

- Forward-algorithm: return a marginal likelihood of the observed sequence
- Forward-backward: predict a single hidden state
- Viterbi: predict an entire sequence of hidden states
- Baum-Welch: unsupervised training (EM)

There are two shortcomings of HMM:

- HMM models capture dependences between each state and only its corresponding observation: Most NLP cases, many tasks needs not only local but also global feature (sentence level).
- Mismatch between learning objective function and prediction objective function: HMM learns a joint distribution of states and observations $p(Y, X)$, but we are more interested in $p(Y|X)$

Chapter 20

Explicit Generative Models

20.1 Variational Autoencoder

Our goal is to find the data distribution $p(X)$. Fig. 20.1 represents a general structure of deep generative model. As you can see, we first sample $z \sim p(z)$ and feed it into a deep neural network $f(z)$ and output x.

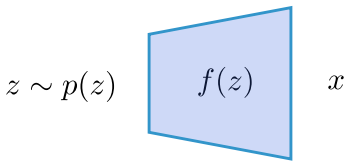


Figure 20.1: General structure of deep generative models. This model does not infer z from x.

VAE performs an inference by introducing a probabilistic encoder, called inference network. VAEs are generative model with a latent variable distributed according to some distribution $p(z_i)$. The observed variable is distributed according to a conditional distribution

$$p_\theta(x_i|z_i)$$

This conditioning means the latent variable values are the one most likely given the observations. We also create a distribution $q_\phi(z_i|x_i)$. We would like to be able to encode our data into the latent variable space. Let's model the distribution.

- $p_\theta(x_i|z_i) \sim \mathcal{N}(x_i|\mu(z_i), \sigma^2(z_i))$: A probabilistic decoder (or generative network, θ)
- $q_\phi(z_i|x_i)$: A probabilistic encoder (or inference network ϕ). We can choose a family of distributions for our conditional distribution q (*e.g.,* standard Gaussian distribution).

  $$q_\phi(z_i|x_i) = \mathcal{N}(z_i|\mu(x_i, W_1), \sigma^2(x_i, W_2)I),$$

  where W_1 and W_2 are network weights and collectively denoted as ϕ. We create a neural network to model the distribution q from our data in a non-linear manner. The outputs of the network are μ and σ.

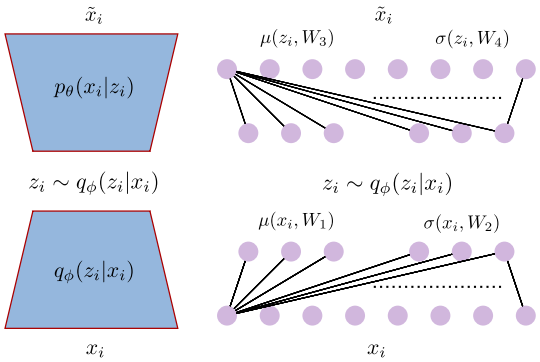


Figure 20.2: Overview of variational autoencoder.

$$
\begin{aligned}
p(X, Z|\theta) &= \prod_{i=1}^{n} \underbrace{p(x_i|z_i, \theta)}_{\text{Likelihood, Generator}} \underbrace{p(z_i|\theta)}_{\text{Prior on latent variable}} \\
&= \prod_{i=1}^{n} \mathcal{N}(x_i| \underbrace{\mu(z_i), \sigma^2(z_i)}_{\text{Non-linear}}) \mathcal{N}(z_i|0, I)
\end{aligned}
$$

Subsequently, marginal distributions can be expressed as follows under i.i.d. assumption:

$$
\begin{aligned}
p(X|\theta) &= \prod_{i=1}^{n} p(x_i|\theta) \\
&= \prod_{i=1}^{n} \int p(x_i, z_j|\theta) dz_j \\
&= \prod_{i=1}^{n} \int p(x_i|z_i, \theta) p(z_i|\theta) dz_i \\
&= \prod_{i=1}^{n} \int \mathcal{N}(x_i|\mu(z_i), \sigma^2(z_i)) \underbrace{\mathcal{N}(z_i|0, I)}_{\text{Mixture weight}} dz_i
\end{aligned}
$$

- As you can see, the marginal distribution $p(X|\theta)$ becomes a mixture of Gaussian (infinite mixture of Gaussian).
- Even though $p(x|z)$ and $p(z)$ are normal, $p(x)$ is not normal, because it is a mixture distribution.
- The non-linearity of Gaussian parameters (modeled by a neural network), conjugacy between the prior and the likelihood does not hold anymore.
- Again, μ and σ is non-linear function of z modeled by some non-linear neural network. The neural network works as a powerful non-linear parameter approximator (based on universal approximation theorem).
- Simple prior is used. Let's consider the data x is an image of 100×100 pixels. Then the covariance matrix has to be 10000×10000. Thus, it is common to set a simple prior such as the standard Gaussian (covariance matrix is diagonal matrix). However, even if we set a simple distribution, with the infinite mixture of Gaussian, we can model any distribution.

- VAE uses a global parametric model to predict the local variational parameters for each data point (**amortized inference**).
- It allows to convert complicated large-dimensional data distributions into simple lower-dimensional latent variable representations.

20.1.1 VAE Optimization

We can train VAE using variational inference with the following objective function, ELBO:

$$\mathcal{L}(\phi, \theta) = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{\mathrm{KL}}(q_\phi(z|x)||p_\theta(z))$$

Let's closely look at this objective function:

- In $q_\phi(z|x)$, x is a given data, so it is not stochatic. How to sample z?
- q has to be deterministic and differentiable.

  $\rightarrow$ **Reparameterization trick**!

  $$\tilde{z} \sim q_\phi(z|x) \rightarrow \tilde{z} \sim g_\phi(\epsilon, x)$$

  , where $\epsilon \sim p(\epsilon)$.
- Estimated by using Monte-Carlo estimation

  $$\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] \approx \frac{1}{N} \sum_j \log p_\theta(x_i|z_j).$$

20.1.2 Conditional VAE

If we have label information about data, then it would provide a better optimization of VAE model. Recall that the following objective function is the objective of the original VAE:

$$\mathcal{L}(\phi, \theta) = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{\mathrm{KL}}(q_\phi(z|x)||p_\theta(z))$$

In conditional VAE,

$$\mathcal{L}(\phi, \theta) = \mathbb{E}_{q_\phi(z|x,y)}[\log p_\theta(x|y, z)] - D_{\mathrm{KL}}(q_\phi(z|x, y) \;||\; p_\theta(z|y))$$

$$\log p(X|Y) = \ln \int q(Z|X, Y) \frac{p(X, Z|Y)}{q(Z|X, Y)} dZ \tag{20.1}$$

$$\geq \underbrace{\int q(Z|X, Y) \ln \frac{p(X, Z|Y)}{q(Z|X, Y)} dZ}_{\text{ELBO, } \mathcal{L}(q,\theta)} \quad \text{by Jensen's Inequality.} \tag{20.2}$$

$$\ldots \tag{20.3}$$

$$\ldots \tag{20.4}$$

$$= \mathbb{E}_{q(Z|X,Y)}[\ln p(X|Z, Y)] - KL(q(Z|X, Y)||p(Z|Y)) \tag{20.5}$$

Note that not we have a prior $p_\theta(z|y)$. However, we have no idea about latent variable z, so we simply assume that we cannot impact the z by y. Thus, we typically set it as a standard normal distribution. Also, we can simply concatenate the input X with Y.

20.1.3 Variational Deep Embedding (VaDE)

The generative process of VADE $p(x, z, c) = p(x|z)p(z|c)p(c)$:

- Choose a cluster $c \sim Cat(\pi)$
- Choose a latent vector $z \sim \mathcal{N}(\mu_c, \sigma_c^2 I)$
- Choose a sample x:

$$x \sim \begin{cases} Ber(\mu_x) & \text{If x is binary} \\ \mathcal{N}(\mu_x, \sigma_x^2 I) & \text{else} \end{cases}$$

ELBO of VaDE:

$$\log p(X) = \ln \int \sum_c p(X, Z, C) dz \tag{20.6}$$

$$\geq \underbrace{\int q(Z, C|X) \ln \frac{p(X, Z, C)}{q(Z, C|X)} dZ}_{\text{ELBO}} \tag{20.7}$$

The ELBO can be decomposed as follows:

$$\begin{aligned} \mathcal{L}_{ELBO} &= \mathbb{E}_q(z, c|x) \left[\ln \frac{p(x, z, c)}{q(z, c|x)} \right] \\ &= \mathbb{E}_q(z, c|x)[\ln p(x, z, c) - \ln q(z, c|x)] \\ &= \mathbb{E}_q(z, c|x)[\ln p(x|z) + \ln p(z|c) + \ln p(c) - \ln q(z|x) - \ln q(c|x)] \end{aligned}$$

By using two factorizations:

- $p(x, z, c) = p(x|z)p(z|c)p(c)$
- $q(z, c|x) \approx q(z|x)q(c|x)$ (Mean-field assumption)
  - $q(z|x) \sim \mathcal{N}$: encoder, estimate mean and variance.
  - $q(c|x)$: assignment probability of Gaussian mixture model

20.1.4 Importance Weighted VAE

Chapter 21

Implicit Generative Models

21.1 Generative Adversarial Networks

- Generator's distribution: p_g
- Prior on input noise: $p_z(z)$
- Mapping to data space: $z \to x$ through $G(z; \theta_g)$

  a differentiable multilayer perceptron with parameter θ_g
- $D(x; \theta_d)$: a differentiable multilayer perceptron with parameter θ_d. It outputs a single scalar
- $D(x)$: probability that x (real) came from the data rather than p_g (fake)

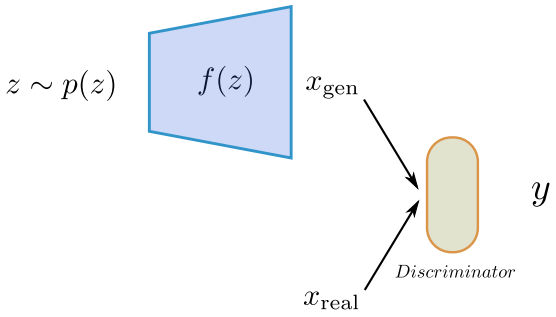


Figure 21.1: GAN structure

21.1.1 Discriminator

The discriminator's goal is to maximize the following equation given G

$$\mathbb{E}_{x \sim p_{data}(x)} \log(D(x)) + \mathbb{E}_{z \sim p_z(z)} \log(1 - D(G(z)))$$

The optimal discriminator given G can be denoted as D^*_G. To get the optimal discriminator, define a value function

$$V(G, D) := \mathbb{E}_{x \sim p_{data}(x)} \log(D(x)) + \mathbb{E}_{z \sim p_z(z)} \log(1 - D(G(z))).$$

Then, $D_G^* = \text{argmax}_D V(G, D)$

However, the generator G wants to minimize the value function given $D = D_G^*$.

$$G^* = \text{argmin}_G V(G, D_G^*).$$

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

- $\min_G \rightarrow$ try to generate fake data that is similar to real data
- $\max_D \rightarrow$ try to assign correct label [1]

At this point, we must show that this optimization problem has a unique solution G^* and that this solution satisfies $p_G = p_{data}$.

One big idea from the GAN paper–, which is different from other approaches is that G **need not be invertible**. Many pieces of notes online miss this fact when they try to replicate the proof and incorrectly use the change of variables formula from calculus (which would depend on G being invertible). Rather, the whole proof relies on this equality:

$$\mathbb{E}_{z \sim p_z(z)} \log(1 - D(G(z))) = \mathbb{E}_{x \sim p_G(x)} \log(1 - D(x)).$$

With the above equality,

$$\begin{aligned}
&\mathbb{E}_{x \sim p_{data}(x)} \log(D(x)) + \mathbb{E}_{z \sim p_z(z)} \log(1 - D(G(z))) \\
&= \int_x p_{data}(x) \log D(x) \, \mathrm{d}x + \int_z p(z) \log(1 - D(G(z))) \, \mathrm{d}z \\
&= \int_x p_{data}(x) \log D(x) + p_G(x) \log(1 - D(x)) \, \mathrm{d}x
\end{aligned}$$

Additionally, we will use the following property:

$$f(y) = a \log y + b \log(1 - y).$$

To find a critical point,

$$f'(y) = 0 \Rightarrow \frac{a}{y} - \frac{b}{1 - y} = 0 \Rightarrow y = \frac{a}{a + b}$$

If $a + b \neq 0$, do the second derivative test:

$$f''\big(\frac{a}{a+b}\big) = -\frac{a}{(\frac{a}{a+b})^2} - \frac{b}{(1 - \frac{a}{a+b})^2} < 0$$

If $a, b \in (0, 1)$, $\frac{a}{a+b}$ is a maximum.

By rewriting the equation,

$$\begin{aligned}
V(G, D) &= \int_x p_{data}(x) \log D(x) + p_G(x) \log(1 - D(x)) \, \mathrm{d}x \\
&\leq \int_x \max_y p_{data}(x) \log y + p_G(x) \log(1 - y) \, \mathrm{d}x
\end{aligned}$$

Thus, if $D(x) = \frac{p_{data}}{p_{data} + p_G}$, then we can achieve the maximum $V(G, D)$.

[1] The above equation is trained separately at the same time, don't get confused

21.1.2 Generator

If we achieve the optimal G (i.e., $p_G = p_{data}$), then D would be completely confused and $D^*_G(x) = \frac{p_{data}}{p_{data}+p_G} = \frac{1}{2}$ (it means that D cannot make a clear decision.).

The global minimum of the virtual training criterion $C(G) = \max_D V(G, D)$ is acheived if and only if $p_G = p_{data}$. Let's plug $D^*_G(x)$ into the criterion then,

$$C(G) = \int_x p_{data}(x) \log \big(\frac{p_{data}(x)}{p_G(x) + p_{data}(x)}\big) + p_G(x) \log \big(\frac{p_G(x)}{p_G(x) + p_{data}(x)}\big)\, \mathrm{d}x.$$

To get the minimum $C(G)$, we can use the Jansen-Shannon divergence:

$$\begin{aligned}
D_{JS}(p_{data}||p_G) &= \frac{1}{2}\left[D_{KL}\Big(p_{data}\Big|\Big|\frac{p_{data}+p_G}{2}\Big) + D_{KL}\Big(p_G\Big|\Big|\frac{p_{data}+p_G}{2}\Big)\right] \\
&= \frac{1}{2}\left[\left(\int_x p_{data}(x)\log\left(\frac{2p_{data}(x)}{p_{data}(x)+p_G(x)}\right)dx\right) + \left(\int_x p_G(x)\log\left(\frac{2p_G(x)}{p_{data}(x)+p_G(x)}\right)dx\right)\right] \\
&= \frac{1}{2}\left[\left(\int_x p_{data}(x)\log 2 + p_{data}(x)\log\left(\frac{p_{data}(x)}{p_{data}(x)+p_G(x)}\right)dx\right) + \right. \\
&\qquad \left.\left(\int_x p_G(x)\log 2 + p_G(x)\log\left(\frac{p_G(x)}{p_{data}(x)+p_G(x)}\right)dx\right)\right] \\
&= \frac{1}{2}\left[\left(\log 2 + \int_x p_{data}(x)\log\left(\frac{2p_{data}(x)}{p_{data}(x)+p_G(x)}\right)dx\right) + \right. \\
&\qquad \left.\left(\log 2 + \int_x p_G(x)\log\left(\frac{2p_G(x)}{p_{data}(x)+p_G(x)}\right)dx\right)\right] \\
&= \frac{1}{2}(\log 4 + C(G))
\end{aligned}$$

Thus,

$$C(G) = -\log 4 + 2D_{JS}(p_{data}||p_G)$$

Since the Jensen-Shannon divergence between two distributions is always non-negative and zero only when they are equal, we have shown that $C^* = -\log(4)$ is the global minimum of $C(G)$ and that the only solution is $p_G = p_{data}$, i.e., the generative model perfectly replicating the data generating process.

21.2 Some notes

What would be the optimal discriminator that separates the two different distributions $p(x)$ and $q(x)$? It turns out that it is

$$f(x) = \frac{q(x)}{p(x) + q(x)}$$

Actually, there are many choices for classifiers e.g., KL-divergence

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)\right].$$

  end for
  • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  • Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Figure 21.2: Training GAN

1. What do we need to learn a classifier?
   - Only samples from $p(x)$ and $q(x)$
2. How do we parameterize $q(x)$?
   - Parametric density function (Gaussian)
   - Define implicitly (GANs approach): define mapping from one (noise) to another (data or image)

The orignal GAN does not learn the data distributions.

21.3 Wasserstein Generative Adversarial Networks

21.3.1 KL Divergence

Definition:

$$D_{\mathrm{KL}}(q(x)||p(x)) = \int q(x) \log \frac{q(x)}{p(x)} dx$$

- Forward KL:
  - If $q(z) \to 0$, Forward KL $\to \infty$
  - Zero avoiding for $q(z)$
- Reverse KL:
  - If $p(z) \to 0$, Reverse KL $\to \infty$
  - Zero forcing: $q(z) \to 0$

Typically, $p(x)$ and $q(x)$ are far apart at the initial state.

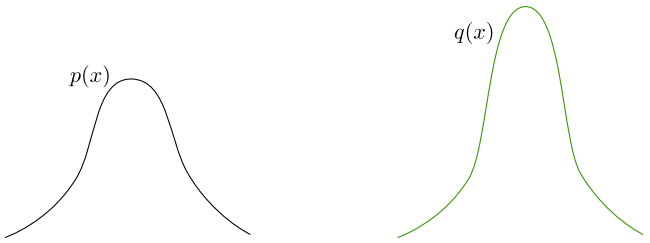


Figure 21.3: Two distributions: $p(x)$ and $q(x)$

Thus, both the forward KL and the reverse KL suffers an unstability issue. Specifically, in each case, if the denominator goes to zero, then the divergence goes to infinity.

21.3.2 Jensen-Shannon Divergence

Definition:

$$D_{JS}(p_{data}||p_G) = \frac{1}{2}\left[D_{KL}\left(p_{data}\middle|\middle|\frac{p_{data}+p_G}{2}\right) + D_{KL}\left(p_G\middle|\middle|\frac{p_{data}+p_G}{2}\right)\right]$$

The KL divergence's issue can be alleviated by JS-divergence. Consider a simple example in Fig. 21.46

$$\forall (x,y) \in P, x = 0 \text{ and } y \sim U(0,1)$$
$$\forall (x,y) \in Q, x = \theta, 0 \leq \theta \leq 1 \text{ and } y \sim U(0,1)$$

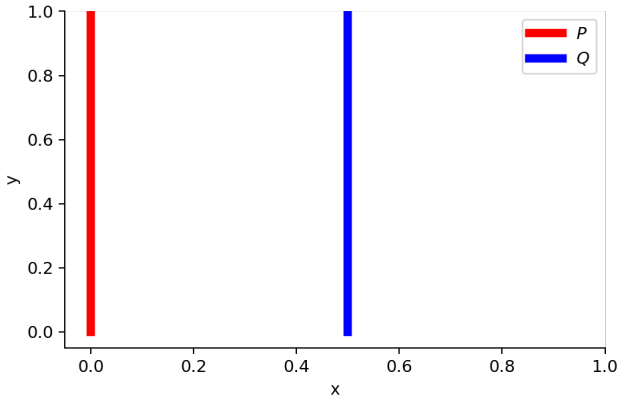


Figure 21.4: Two distributions: $p(x)$ and $q(x)$

$$
\begin{aligned}
D_{\mathrm{KL}}(q(x)||p(x)) &= \infty \\
D_{\mathrm{KL}}(p(x)||q(x)) &= \infty \\
D_{JS}(p_{data}||p_G) &= \frac{1}{2}\left[D_{KL}\left(p_{data}\middle|\middle|\frac{p_{data}+p_G}{2}\right) + D_{KL}\left(p_G\middle|\middle|\frac{p_{data}+p_G}{2}\right)\right] \\
&= \frac{1}{2}\left[D_{KL}\left(p_{data}\middle|\middle|\frac{p_{data}}{2}\right) + +D_{KL}\left(p_G\middle|\middle|\frac{p_G}{2}\right)\right] \\
&= \frac{1}{2}[\log 2 + \log 2] = \log 2 \\
W(p,q) &= |\theta|
\end{aligned}
$$

Therefore, Jensen-Shannon divergence is more stabler than KL divergece. This is one of the reasons why GAN, which uses JS divergence works better than VAE, which uses KL divergence.

However, JS divergence also has some problem. If the value is close to $\frac{1}{2}\log 2$, then the gradient will be very small or close to zero, because the divergence is close to constant. It means that a training speed is very slow. Thus, we need a better metric.

21.3.3 Wasserstein Distance

Wasserstein Distance is a measure of the distance between two probability distributions. It is also called Earth Mover's distance, short for EM distance, because informally it can be interpreted as the minimum energy cost of moving and transforming a pile of dirt in the shape of one probability distribution to the shape of the other distribution.

$$W(p_r, p_g) = \inf_{\gamma \sim \Pi(p_r, p_g)} \mathbb{E}_{(x,y)\sim\gamma}[\|x-y\|]$$

- Π: is the transportation plan and the set of all possible joint probability distributions between p_r and p_g. One joint distribution $\gamma \sim \Pi(p_r, p_g)$ describes one transport plan.
- $\mathbb{E}_{x,y\sim\gamma}\|x-y\| = \sum_{x,y}\gamma(x,y)\|x-y\|$

- Finally, we take the minimum one among the costs of all dirt moving solutions as the EM distance (by infimum).

21.4 WGAN

However, consider all possible joint distribution is intractable, so dual solution can be used.

$$W(p_r, p_g) = \frac{1}{K} \sup_{\|f\|_L \leq K} \mathbb{E}_{x \sim p_r}[f(x)] - \mathbb{E}_{x \sim p_g}[f(x)]$$

So to calculate the Wasserstein distance, we just need to find a 1-Lipschitz function. To enforce the constraint, WGAN applies a very simple clipping to restrict the maximum weight value in f, i.e. the weights of the discriminator

Suppose this function f comes from a family of K-Lipschitz continuous functions, $\{f_w\}_{w \in W}$, parameterized by w. In the modified Wasserstein-GAN, the "discriminator" model is used to learn w to find a good f_w and the loss function is configured as measuring the Wasserstein distance between p_r and p_g.

$$L(p_r, p_g) = W(p_r, p_g) = \max_{w \in W} \mathbb{E}_{x \sim p_r}[f_w(x)] - \mathbb{E}_{z \sim p_r(z)}[f_w(g_\theta(z))]$$

There are two ways to satisfy the Lipschitz continuity:

- Weight clipping
- Gradient Penalty

21.4.1 Lipschitz continuity

The function f in the new form of Wasserstein metric is demanded to satisfy $\|f\|_L \leq K$, meaning it should be K-Lipschitz continuous.

A real-valued function $f : \mathbb{R} \to \mathbb{R}$ is called K-Lipschitz continuous if there exists a real constant $K \geq 0$ such that, for all $x_1, x_2 \in \mathbb{R}$

$$|f(x_1) - f(x_2)| \leq K|x_1 - x_2|$$

Here K is known as a Lipschitz constant for function $f(\cdot)$. Functions that are everywhere continuously differentiable is Lipschitz continuous, because the derivative, estimated as $\frac{|f(x_1)-f(x_2)|}{|x_1-x_2|}$, has bounds. However, a Lipschitz continuous function may not be everywhere differentiable, such as $f(x) = |x|$

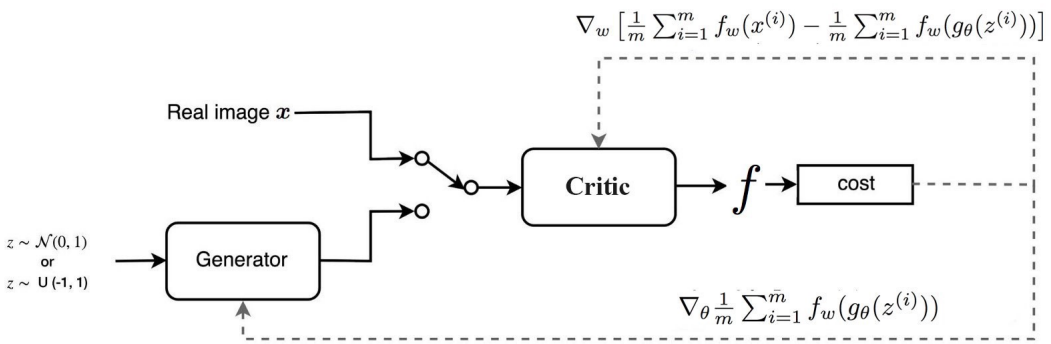


Figure 21.5: WGAN

	Discriminator/Critic	Generator
GAN	$\nabla_{\theta_d} \frac{1}{m}\sum_{i=1}^m \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)\right]$	$\nabla_{\theta_g} \frac{1}{m}\sum_{i=1}^m \log\left(D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)$
WGAN	$\nabla_w \frac{1}{m}\sum_{i=1}^m \left[f(x^{(i)}) - f(G\left(z^{(i)}\right))\right]$	$\nabla_\theta \frac{1}{m}\sum_{i=1}^m f(G\left(z^{(i)}\right))$

Figure 21.6: WGAN

21.5 InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets

21.5.1 Joint Entropy

$$H(X,Y) = \mathbb{E}_{X,Y}[-\log p(x,y)] = -\sum_{x,y} p(x,y)\log p(x,y)$$

21.5.2 Conditional Entropy

$$\begin{aligned}
H(X|Y) &= \mathbb{E}_Y[H(X,Y)] \\
&= -\sum_{y\sim p_Y(y)} p(y) \sum_{x\sim p_X(x)} p(x|y)\log p(x|y) \\
&= -\sum_{y\sim p_Y(y)} \sum_{x\sim p_X(x)} p(y)p(x|y)\log p(x|y) \\
&= -\sum_{y\sim p_Y(y)} \sum_{x\sim p_X(x)} p(x,y)\log p(x|y) = -\mathbb{E}_{x,y}[\log p(x|y)] \\
&= -\sum_{y\sim p_Y(y)} \sum_{x\sim p_X(x)} p(x,y)\log \frac{p(x,y)}{p(y)} \\
&= -\sum_{y\sim p_Y(y)} \sum_{x\sim p_X(x)} p(x,y)\log p(x,y) + \sum_{y\sim p_Y(y)} \sum_{x\sim p_X(x)} p(x,y)\log p(y) \\
&= H(X,Y) - H(Y)
\end{aligned}$$

21.5.3 Variational Mutual Information Maximization

$$\begin{aligned}
I(c;G(z,c)) &= H(c) - H(c|G(z,c)) \\
&= H(c) + \int\int p(c=c', x=G(z,c))\log p(c=c'|x=G(z,c))dc'dz \\
&= H(c) + \mathbb{E}_{x\sim G(z,c),c'\sim p(c|x)}[\log p(c'|x)] \\
&= H(c) + \mathbb{E}_{x\sim G(z,c)}\mathbb{E}_{c'\sim p(c|x)}[\log p(c'|x)] \\
&= H(c) + \mathbb{E}_{x\sim G(z,c)}\mathbb{E}_{c'\sim p(c|x)}\left[\log \frac{p(c'|x)Q(c'|x)}{Q(c'|x)}\right] \\
&= H(c) + \mathbb{E}_{x\sim G(z,c)}\mathbb{E}_{c'\sim p(c|x)}\left[\log \frac{p(c'|x)}{Q(c'|x)}\right] + \mathbb{E}_{x\sim G(z,c)}\mathbb{E}_{c'\sim p(c|x)}\Big[\log Q(c'|x)\Big] \\
&= H(c) + \mathbb{E}_{x\sim G(z,c)}\Big[D_{KL}(p(c'|x)||Q(c'|x))\Big] + \mathbb{E}_{x\sim G(z,c)}\mathbb{E}_{c'\sim p(c|x)}\Big[\log Q(c'|x)\Big] \\
&\geq H(c) + \mathbb{E}_{x\sim G(z,c)}\mathbb{E}_{c'\sim p(c|x)}\Big[\log Q(c'|x)\Big]^2
\end{aligned}$$

Thus we get a lower bound for the mutual information as follows:

$$I(c; G(z,c)) \geq H(c) + \mathbb{E}_{x\sim G(z,c)}\mathbb{E}_{c'\sim p(c|x)}\Big[\log Q(c'|x)\Big]$$

However, we still have a problem. We need to sample c from $p(c|x)$. Thus, we need to replace it with a known distribution. Firstly, with the reasoning that $x \sim G(z,c)$ means sample c from $p(c)$ then sample x from $G(z,c)$. So we can express $\mathbb{E}_{x\sim G(z,c)}$ with $\mathbb{E}_{c\sim p(c)}\mathbb{E}_{x\sim G(z,c)}$. and by the Lemma 1,

$$\begin{aligned} I(c; G(z,c)) &\geq H(c) + \mathbb{E}_{x\sim G(z,c)}\mathbb{E}_{c'\sim p(c|x)}\Big[\log Q(c'|x)\Big] \\ &= H(c) + \mathbb{E}_{c\sim p(c)}\mathbb{E}_{x\sim G(z,c)}\mathbb{E}_{c'\sim p(c|x)}\Big[\log Q(c'|x)\Big] \\ &= H(c) + \mathbb{E}_{c\sim p(c)}\mathbb{E}_{x\sim G(z,c)}\Big[\log Q(c'|x)\Big]^3 \end{aligned}$$

Thus, we can directly sample c from the known distribution instead of $p(c|x)$.

lemma 1 *For random variables X, Y and function $f(x,y)$ under suitable regularity conditions:*

$$\mathbb{E}_{x\sim X, y\sim Y|x}[f(x,y)] = \mathbb{E}_{x\sim X, y\sim Y|x, x'\sim X|y}[f(x',y)]$$

proof 1

$$\begin{aligned} \mathbb{E}_{x\sim X, y\sim Y|x}[f(x,y)] &= \int_x P(x)\int_y P(y|x)f(x,y)dydx \\ &= \int_x\int_y P(x,y)f(x,y)dydx \\ &= \int_{x'}\int_y P(x',y)f(x',y)dydx' \\ &= \int_{x'}\int_y P(y)P(x'|y)f(x',y)dydx' \\ &= \int_{x'}\int_y\int_x P(x,y)P(x'|y)f(x',y)dxdydx' \\ &= \int_x P(x)\int_y P(x|y)\int_{x'} P(x'|y)f(x',y)dxdydx' \\ &= \mathbb{E}_{x\sim X, y\sim Y|x, x'\sim X|y}[f(x',y)] \end{aligned}$$

Chapter 22

Diffusion Model

22.1 Introduction

(Denoising) Diffusion models have emerged as the new SOTA family of deep generative models.

- First proposed in 2015.
- Outperform GANs on image synthesis (DDPM) $\sim$ 2020.
- Stable training dynamics.
- Image synthesis, super resolution, text-to-image, and so on.

The overall idea is to construct a Markov chain of progressively less noisy samples. Each transition denoises a noisy sample. Diffusion models consist of two Markov chains:

1. Forward: A Markov chain of diffusion steps to **slowly add random noise** to data.

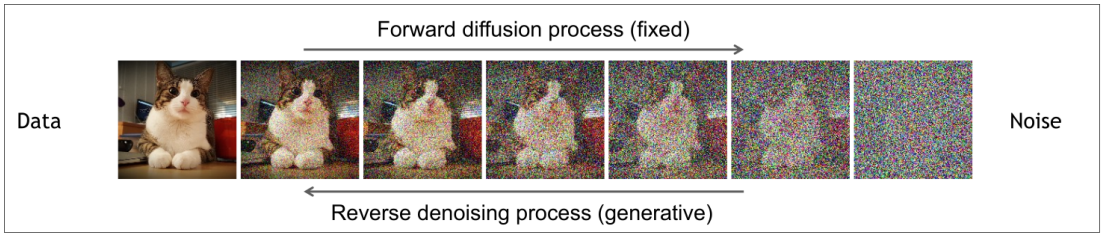
$$\mathbf{x}_0 \to \mathbf{x}_1 \cdots \to \mathbf{x}_T$$

2. Backward (Reverse): Learn to **reverse the diffusion process** to construct desired data samples from the noise.

$$\mathbf{x}_T \to \mathbf{x}_{T-1} \cdots \to \mathbf{x}_0$$

Some properties of diffusion models:

1. Diffusion model has a pre-defined sampling equation.
   - The equation relies on a random noise.
   - Noise is all we need $\to$ Predict noise at a time step t.
2. Fit a model via forward and backward processes.
3. **Iterative transform** of one distribution into another via **Makov Chain**.
   - $\mathcal{D}_{data} \to \mathcal{N}$.



- $\mathcal{N} \to \mathcal{D}_{data}$.
- Diffusion model≈Generative Markov Chain.

4. Learn a transition model:

$$p_\theta(\mathbf{x}_{t-1}|x_t) = \mathcal{N}(\mathbf{x}_{t-1}|\boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t)).$$

5. Base case: $p(\mathbf{x}_T) = \mathcal{N}(0, I)$
6. Marginal distribution over $\mathbf{x}_0$:

$$p_\theta(\mathbf{x}_0) = \int p_\theta(\mathbf{x}_0, \ldots, \mathbf{x}_T) d\mathbf{x}_1, \ldots, \mathbf{x}_T$$

7. We want to learn the parameters so that

$$p(\mathbf{x}_0) \approx p_\theta(\mathbf{x}_0)$$

22.2 Forward Diffusion

- We want to model a forward trajectory (by the Markov property):

$$q(\mathbf{x}_{0:T}) = q(\mathbf{x}_0) \prod_{t=1}^{T} \underbrace{q(\mathbf{x}_t|\mathbf{x}_{t-1})}_{\text{Transition kernel}}$$

- Slow transform with a large T: $\mathbf{x}_0 \to \mathbf{x}_1 \cdots \to \mathbf{x}_T$
  - Imagine someone said he is from Germany.
  - We can't exactly track his journey without more information.
  - We need to add more steps!
- How to model $q(\mathbf{x}_t|\mathbf{x}_{t-1})$?

Forward Diffusion: $q(\mathbf{x}_t|\mathbf{x}_{t-1})$ In a continuous case (*e.g.,* image), each transition can be parameterized as follows:

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t \mathbf{I}) \tag{22.1}$$

- $\beta_t \in (0, 1)$ is a variance at time t.
- $\sqrt{1-\beta_t}$ downscales $\mathbf{x}_{t-1}$ to be 0, $\beta_1 < \cdots < \beta_t$. Thus, $\mathbf{x}_t$ will become more noisier. $\mathbf{x}_t$ can be sampled as:

$$\mathbf{x}_t = \sqrt{1-\beta_t}\mathbf{x}_{t-1} + \sqrt{\beta_t} \odot \epsilon$$

1. Sample $\mathbf{x}_t \sim q(\mathbf{x}_t)$ and scale it by $\sqrt{1-\beta_t}$
2. Adds noise $\epsilon \sim \mathcal{N}(0, I)$ with variance β_t.

- The above process is autoregressive (*i.e.,* ancestral sampling), but we can sample $\mathbf{x}_t$ directly from $q(\mathbf{x}_t|\mathbf{x}_0)$ in an analytic form:

$$
\begin{aligned}
\mathbf{x}_t &= \sqrt{1-\beta_t}\mathbf{x}_{t-1} + \sqrt{\beta_t} \odot \epsilon_{t-1} && (22.2)\\
&= \sqrt{\alpha_t}\mathbf{x}_{t-1} + \sqrt{1-\alpha_t} \odot \epsilon_{t-1} && (22.3)\\
&= \sqrt{\alpha_t}\left(\sqrt{\alpha_{t-1}}\mathbf{x}_{t-2} + \sqrt{1-\alpha_{t-1}} \odot \epsilon_{t-2}\right) + \sqrt{1-\alpha_t} \odot \epsilon_{t-1} && (22.4)\\
&= \sqrt{\alpha_t\alpha_{t-1}}\mathbf{x}_{t-2} + \sqrt{\alpha_t - \alpha_t\alpha_{t-1}} \odot \epsilon_{t-2} + \sqrt{1-\alpha_t} \odot \epsilon_{t-1} && (22.5)\\
&= \sqrt{\alpha_t\alpha_{t-1}}\mathbf{x}_{t-2} + \sqrt{\alpha_t - \alpha_t\alpha_{t-1} + 1 - \alpha_t} \odot \epsilon_{t-2} && (22.6)\\
&= \sqrt{\alpha_t\alpha_{t-1}}\mathbf{x}_{t-2} + \sqrt{1 - \alpha_t\alpha_{t-1}} \odot \epsilon_{t-2} && (22.7)\\
&= \dots && (22.8)\\
&= \sqrt{\prod_t \alpha_t}\mathbf{x}_0 + \sqrt{1 - \prod_t \alpha_t} \odot \epsilon_{t_0} && (22.9)\\
&= \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t} \odot \epsilon && (22.10)\\
&\sim \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I}), && (22.11)
\end{aligned}
$$

  where $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$. Thus, $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t} \odot \epsilon$. Note that the fifth step is done by using the property of sum of two Gaussian distributions (*e.g.,* $\mathcal{N}(0, \sigma_1^2 I) + \mathcal{N}(0, \sigma_2^2 I) = \mathcal{N}(0, (\sigma_1^2 + \sigma_2^2) I)$).

We can get some intuitions:

- The original input $\mathbf{x}_0$ **gradually loses all info** during the forward diffusion process.
- This Markov chain has a **stationary distribution**: As $t \to \infty$, $q(\mathbf{x}_t) \approx \mathcal{N}(0, I)$.
  - In practice, T is a very high number *e.g.,* 1,000.
  - Minimize info loss for each step.
  - Allow a smooth training.

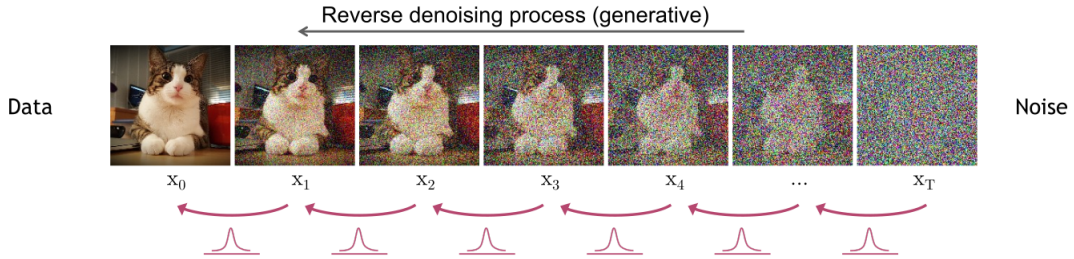
22.3 Backward Process

Generative Learning by Denoising:

- Now we know how to model the forward process (diffusion process).
- However, a noisy image is not what we want.
- We want to generate a new image with high quality.

How to generate data? If we can reverse the forward process, then we can draw a true sample. We call it **Backward process**!

- Start from $q(\mathbf{x}_T) \approx \mathcal{N}(0, I)$.
- Sample $\mathbf{x}_T \sim \mathcal{N}(\mathbf{x}_T|0, I)$
  - Sample a noise vector from a prior distribution.
- Iteratively sample $\mathbf{x}_{t-1} \sim q(\mathbf{x}_{t-1}|\mathbf{x}_t)$.
  - $q(\mathbf{x}_{t-1}|\mathbf{x}_t)$: **true denoising distribution** (we don't know).
- In general, $q(\mathbf{x}_{t-1}|\mathbf{x}_t)$ is intractable.
- We can **approximate** $q(\mathbf{x}_{t-1}|\mathbf{x}_t)$ as **Normal distribution** if β_t is small in each forward diffusion step.
  - *e.g.*, $\mathbf{x}_3$ to $\mathbf{x}_2$



CVPR 2022 Tutorial: Denoising Diffusion-based Generative Modeling: Foundations and Applications

Backward Process

- Approximate $q(\mathbf{x}_{t-1}|\mathbf{x}_t)$ using a neural network, $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$.
- Backward process: $p_\theta(\mathbf{x}_{0:T}) = p(\mathbf{x}_T) \prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$.
  - $p(\mathbf{x}_T) = \mathcal{N}(\mathbf{x}_T; 0, I)$.
  - $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t))$.
  - We can model the denoising distribution as Normal distribution like above (*c.f.*, Eq. (22.45)).
  - Note that the reverse conditional probability $q(\mathbf{x}_{t-1}|\mathbf{x}_t)$ is tractable when it is conditioned on $\mathbf{x}_0$ as shown in Eq. (22.45). This allows us to train a neural network to model this denoising distribution.
- Key to the success of this sampling process is training the reverse Markov chain to match the actual time reversal of the forward Markov chain.
- After optimizing the backward process, the sampling procedure is that just sample Gaussian noise from $p(\mathbf{x}_T)$ and then iteratively running the denoising transitions (backward process) for T steps to generate a novel $\mathbf{x}_0$.

22.4 Distribution Modeling

What we want to learn (or model) is $p_\theta(\mathbf{x}_0) \approx p(\mathbf{x}_0)$ (approximate data distribution).

- $p_\theta(\mathbf{x}_0) = \int p_\theta(\mathbf{x}_{0:T}) d\mathbf{x}_{1:T}$
- It is intractable to compute all trajectories.

$$\underset{\theta}{\operatorname{argmax}}\, \mathbb{E}_{\mathbf{x}_0 \sim p}[\log p_\theta(\mathbf{x}_0)] = \mathbb{E}_{\mathbf{x}_0 \sim p}\left[\log \int p_\theta(\mathbf{x}_{0:T}) d\mathbf{x}_{1:T}\right].$$

- Thus, we will use variational lower bound with KL-Div:

$$\log p_\theta(\mathbf{x}_0) = \log \int p(\mathbf{x}_{0:T}) d\mathbf{x}_{1:T} \tag{22.12}$$

$$= \log \int p(\mathbf{x}_{0:T}) \frac{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} d\mathbf{x}_{1:T} \tag{22.13}$$

$$= \log \int q(\mathbf{x}_{1:T}|\mathbf{x}_0) \frac{p(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} d\mathbf{x}_{1:T} \tag{22.14}$$

$$\geq \int q(\mathbf{x}_{1:T}|\mathbf{x}_0) \log \frac{p(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} d\mathbf{x}_{1:T} \tag{22.15}$$

$$= \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} \left[\log \frac{p(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} \right] \quad \rightarrow \text{ELBO} \tag{22.16}$$

$$= \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} \left[\log p(\mathbf{x}_T) \prod_{t=1}^{T} \frac{p(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_t|\mathbf{x}_{t-1})} \right] \tag{22.17}$$

$$= \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} \left[\log \frac{p(\mathbf{x}_T) p(\mathbf{x}_0|\mathbf{x}_1) \prod_{t=1}^{T-1} p(\mathbf{x}_t|\mathbf{x}_{t+1})}{q(\mathbf{x}_T|\mathbf{x}_{T-1}) \prod_{t=1}^{T-1} q(\mathbf{x}_t|\mathbf{x}_{t-1})} \right] \tag{22.18}$$

$$= \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} \left[\log \frac{p(\mathbf{x}_T) p(\mathbf{x}_0|\mathbf{x}_1)}{q(\mathbf{x}_T|\mathbf{x}_{T-1})} \right] + \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} \left[\log \prod_{t=1}^{T-1} \frac{p(\mathbf{x}_t|\mathbf{x}_{t+1})}{q(\mathbf{x}_t|\mathbf{x}_{t-1})} \right] \tag{22.19}$$

$$= \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} [\log p(\mathbf{x}_0|\mathbf{x}_1)] + \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} \left[\log \frac{p(\mathbf{x}_T)}{q(\mathbf{x}_T|\mathbf{x}_{T-1})} \right] \tag{22.20}$$

$$+ \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} \left[\sum_{t=1}^{T-1} \log \frac{p(\mathbf{x}_t|\mathbf{x}_{t+1})}{q(\mathbf{x}_t|\mathbf{x}_{t-1})} \right] \tag{22.21}$$

$$= \cdots + \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} \left[\log \frac{p(\mathbf{x}_T)}{q(\mathbf{x}_T|\mathbf{x}_{T-1})} \right] + \sum_{t=1}^{T-1} \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} \left[\log \frac{p(\mathbf{x}_t|\mathbf{x}_{t+1})}{q(\mathbf{x}_t|\mathbf{x}_{t-1})} \right] \tag{22.22}$$

$$= \cdots + \int_{x_1} \cdots \int_{x_T} q(\mathbf{x}_1, \ldots, \mathbf{x}_T) \left[\log \frac{p(\mathbf{x}_T)}{q(\mathbf{x}_T|\mathbf{x}_{T-1})} \right] dx_1 \ldots dx_T + \ldots \tag{22.23}$$

$$= \cdots + \int_{x_T} \int_{x_{T-1}} \left[\log \frac{p(\mathbf{x}_T)}{q(\mathbf{x}_T|\mathbf{x}_{T-1})} \right] \cdots \int_{x_1} q(\mathbf{x}_1, \ldots, \mathbf{x}_T) dx_1 \ldots dx_T + \ldots \tag{22.24}$$

$$= \cdots + \int_{x_T} \int_{x_{T-1}} \left[\log \frac{p(\mathbf{x}_T)}{q(\mathbf{x}_T|\mathbf{x}_{T-1})} \right] q(\mathbf{x}_t, \mathbf{x}_{t-1}) dx_{T-1} dx_T + \ldots \tag{22.25}$$

$$= \cdots + \mathbb{E}_{q(\mathbf{x}_t, \mathbf{x}_{t-1}|\mathbf{x}_0)} \left[\log \frac{p(\mathbf{x}_T)}{q(\mathbf{x}_T|\mathbf{x}_{T-1})} \right] + \sum_{t=1}^{T-1} \mathbb{E}_{q(\mathbf{x}_{t-1}, \mathbf{x}_t, \mathbf{x}_{t+1}|\mathbf{x}_0)} \left[\log \frac{p(\mathbf{x}_t|\mathbf{x}_{t+1})}{q(\mathbf{x}_t|\mathbf{x}_{t-1})} \right] \tag{22.26}$$

$$= \mathbb{E}_{q(\mathbf{x}_1|\mathbf{x}_0)} [\log p(\mathbf{x}_0|\mathbf{x}_1)] + \mathbb{E}_{q(\mathbf{x}_t, \mathbf{x}_{t-1}|\mathbf{x}_0)} \left[\log \frac{p(\mathbf{x}_T)}{q(\mathbf{x}_T|\mathbf{x}_{T-1})} \right] \tag{22.27}$$

$$+ \sum_{t=1}^{T-1} \mathbb{E}_{q(\mathbf{x}_{t-1}, \mathbf{x}_t, \mathbf{x}_{t+1}|\mathbf{x}_0)} \left[\log \frac{p(\mathbf{x}_t|\mathbf{x}_{t+1})}{q(\mathbf{x}_t|\mathbf{x}_{t-1})} \right] \tag{22.28}$$

$$= \mathbb{E}_{q(\mathbf{x}_1|\mathbf{x}_0)} [\log p(\mathbf{x}_0|\mathbf{x}_1)] - \mathbb{E}_{q(\mathbf{x}_{t-1}|\mathbf{x}_0)} [D_{KL}(q(\mathbf{x}_T|\mathbf{x}_{T-1}) \| p(\mathbf{x}_T))] \tag{22.29}$$

$$- \sum_{t=1}^{T-1} \mathbb{E}_{q(\mathbf{x}_{t-1}, \mathbf{x}_{t+1}|\mathbf{x}_0)} D_{KL}[q(\mathbf{x}_t|\mathbf{x}_{t-1}) \| p(\mathbf{x}_t|\mathbf{x}_{t+1})] \tag{22.30}$$

$$\because q(\mathbf{x}_t, \mathbf{x}_{t-1}|\mathbf{x}_0) = \frac{q(\mathbf{x}_t, \mathbf{x}_{t-1}, \mathbf{x}_0)}{q(\mathbf{x}_0)} = \frac{q(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{x}_0) q(\mathbf{x}_{t-1}, \mathbf{x}_0)}{q(\mathbf{x}_0)} = q(\mathbf{x}_t|\mathbf{x}_{t-1}) q(\mathbf{x}_{t-1}|\mathbf{x}_0) \tag{22.31}$$

- The sixth step is done by Markov property.
- The first term is a *reconstruction term.*
- The second term is a *prior matching term.* This term requires no optimization, as it has no trainable parameters; furthermore, as we have assumed a large enough T such that the final distribution is Gaussian, this term effectively becomes zero.
- The last term is a *consistency term.* It endeavors to make the distribution at x_t consistent, from both forward and backward processes. That is, a denoising step from a noisier image should match the corresponding noising step from a cleaner image, for every intermediate timestep. This term is minimized when we train $p_\theta(\mathbf{x}_t|\mathbf{x}_{t+1})$ to match the Gaussian distribution $q(\mathbf{x}_t|\mathbf{x}_{t-1})$, which is defined in Eq. (22.1).
- Under this derivation, all terms of the ELBO are computed as expectations, and can therefore be approximated using Monte Carlo estimates.
- However, actually optimizing the ELBO using the terms we just derived might be suboptimal, because the consistency term is computed as an expectation over two random variables $\{x_{t-1}, x_{t+1}\}$ for every time step, the variance of its Monte Carlo estimate could potentially be higher than a term that is estimated using only one random variable per time step. As it is computed by summing up $T-1$ consistency terms, the final estimated value of the ELBO may have high variance for large T values.

Let us instead try to derive a form for our ELBO where each term is computed as an expectation over **only one random variable at a time**. The key insight is that we can rewrite encoder transitions as $q(x_t|x_{t-1}) = q(x_t|x_{t-1}, x_0)$, where the extra conditioning term is superfluous due to the Markov property. Then, according to Bayes rule, we can rewrite each transition as:

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{x}_0) = \frac{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)q(\mathbf{x}_t|\mathbf{x}_0)}{q(\mathbf{x}_{t-1}|\mathbf{x}_0)}$$

Armed with this equation, we can factorize the ELBO again as follows:

$$L = \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log \frac{p(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\right] \quad \rightarrow \text{ELBO} \tag{22.32}$$

$$= \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log p(\mathbf{x}_T) \prod_{t=1}^{T} \frac{p(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_t|\mathbf{x}_{t-1})}\right] \tag{22.33}$$

$$= \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log \frac{p(\mathbf{x}_T)p(\mathbf{x}_0|\mathbf{x}_1)\prod_{t=2}^{T} p(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_1|\mathbf{x}_0)\prod_{t=2}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1})}\right] \tag{22.34}$$

$$= \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log \frac{p(\mathbf{x}_T)p(\mathbf{x}_0|\mathbf{x}_1)\prod_{t=2}^{T} p(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_1|\mathbf{x}_0)\prod_{t=2}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{x}_0)}\right] \quad \text{by Markov Property} \tag{22.35}$$

$$= \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log \frac{p(\mathbf{x}_T)p(\mathbf{x}_0|\mathbf{x}_1)}{q(\mathbf{x}_1|\mathbf{x}_0)} + \log \prod_{t=2}^{T} \frac{p(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{x}_0)}\right] \tag{22.36}$$

$$= \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log \frac{p(\mathbf{x}_T)p(\mathbf{x}_0|\mathbf{x}_1)}{q(\mathbf{x}_1|\mathbf{x}_0)} + \log \prod_{t=2}^{T} \frac{p(\mathbf{x}_{t-1}|\mathbf{x}_t)}{\frac{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)q(\mathbf{x}_t|\mathbf{x}_0)}{q(\mathbf{x}_{t-1}|\mathbf{x}_0)}}\right] \tag{22.37}$$

$$= \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log \frac{p(\mathbf{x}_T)p(\mathbf{x}_0|\mathbf{x}_1)}{q(\mathbf{x}_1|\mathbf{x}_0)} + \log \frac{q(\mathbf{x}_1|\mathbf{x}_0)}{q(\mathbf{x}_T|\mathbf{x}_0)} + \log \prod_{t=2}^{T} \frac{p(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)}\right] \tag{22.38}$$

$$= \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log \frac{p(\mathbf{x}_T)p(\mathbf{x}_0|\mathbf{x}_1)}{q(\mathbf{x}_T|\mathbf{x}_0)} + \log \prod_{t=2}^{T} \frac{p(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)}\right] \tag{22.39}$$

$$= \mathbb{E}_{q(\mathbf{x}_1|\mathbf{x}_0)}[\log p(\mathbf{x}_0|\mathbf{x}_1)] + \mathbb{E}_{q(\mathbf{x}_T|\mathbf{x}_0)}\left[\log \frac{p(\mathbf{x}_T)}{q(\mathbf{x}_T|\mathbf{x}_0)}\right] + \sum_{t=2}^{T} \mathbb{E}_{q(\mathbf{x}_t, \mathbf{x}_{t-1}|\mathbf{x}_0)}\left[\log \frac{p(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)}\right] \tag{22.40}$$

$$= \mathbb{E}_{q(\mathbf{x}_1|\mathbf{x}_0)}[\log p(\mathbf{x}_0|\mathbf{x}_1)] - D_{KL}(q(\mathbf{x}_T|\mathbf{x}_0)||p(\mathbf{x}_T)) - \sum_{t=2}^{T} \mathbb{E}_{q(\mathbf{x}_t|\mathbf{x}_0)}[D_{KL}(q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)\|p(\mathbf{x}_{t-1}|\mathbf{x}_t))] \tag{22.41}$$

Let's closely look at the last three terms:

- $\mathbb{E}_{q(\mathbf{x}_1|\mathbf{x}_0)}[\log p(\mathbf{x}_0|\mathbf{x}_1)]$: reconstruction term.
- $D_{KL}(q(\mathbf{x}_T|\mathbf{x}_0)||p(\mathbf{x}_T))$: Prior matching term.
  - No trainable parameters
- $\sum_{t=2}^{T} \mathbb{E}_{q(\mathbf{x}_t|\mathbf{x}_0)}[D_{KL}(q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)\|p(\mathbf{x}_{t-1}|\mathbf{x}_t))]$: Denoising matching term.
  - $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$ as an approximation to tractable, ground-truth denoising transition step $q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)$. The $q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)$ transition step can act as a ground-truth signal, since it defines how to denoise a noisy image with access to what the final, completely denoised image $\mathbf{x}_0$ should be. This term is therefore minimized when the two denoising steps match as closely as possible, as measured by their KL Divergence.

In the last term, $q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)$ can be further factorized as follows:

$$q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) = \frac{q(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{x}_0)q(\mathbf{x}_{t-1}|\mathbf{x}_0)}{q(\mathbf{x}_t|\mathbf{x}_0)} = \frac{q(\mathbf{x}_t|\mathbf{x}_{t-1})q(\mathbf{x}_{t-1}|\mathbf{x}_0)}{q(\mathbf{x}_t|\mathbf{x}_0)}.$$

Then, $q(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{x}_0) = q(\mathbf{x}_t|\mathbf{x}_{t-1})$ by Markov property. Now, let's compute the KL-divergence in the denoising matching term.

$$q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) = \frac{q(\mathbf{x}_t|\mathbf{x}_{t-1})q(\mathbf{x}_{t-1}|\mathbf{x}_0)}{q(\mathbf{x}_t|\mathbf{x}_0)} \tag{22.42}$$

$$= \frac{\mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_{t-1}, (1-\bar{\alpha}_t)\mathbf{I})\mathcal{N}(\mathbf{x}_{t-1}; \sqrt{\bar{\alpha}_{t-1}}\mathbf{x}_0, (1-\bar{\alpha}_{t-1})\mathbf{I})}{\mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I})} \tag{22.43}$$

$$\vdots \tag{22.44}$$

$$\propto \mathcal{N}\Bigg(\mathbf{x}_{t-1}; \underbrace{\frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})\mathbf{x}_t + \sqrt{\bar{\alpha}_{t-1}}(1-\alpha_t)\mathbf{x}_0}{1-\bar{\alpha}_t}}_{\mu_q(\mathbf{x}_t, \mathbf{x}_0)}, \underbrace{\frac{(1-\alpha_t)(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\mathbf{I}}_{\Sigma_q(t)}\Bigg) \tag{22.45}$$

We have therefore shown that at each step, $\mathbf{x}_{t-1} \sim q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)$ is normally distributed, with mean $\mu_q(\mathbf{x}_t, \mathbf{x}_0)$ that is a function of $\mathbf{x}_t$ and $\mathbf{x}_0$, and variance $\Sigma_q(t)$ as a function of α coefficients.

We can use the Eq. (22.45) to compute the optimal θ by solving $\min_\theta D_{KL}(q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)\|p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t))$, but we can simplify the optimization problem by using Eq. (22.11):

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t} \odot \epsilon$$

$$\mathbf{x}_0 = \frac{\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t} \odot \epsilon}{\sqrt{\bar{\alpha}_t}}$$

By plugging the above equation into $\boldsymbol{\mu}_q(\mathbf{x}_t, \mathbf{x}_0)$, we can exclude $\mathbf{x}_0$ term as follows:

$$\boldsymbol{\mu}_q(\mathbf{x}_t, \mathbf{x}_0) = \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})\mathbf{x}_t + \sqrt{\bar{\alpha}_{t-1}}(1-\alpha_t)\mathbf{x}_0}{1-\bar{\alpha}_t} \tag{22.46}$$

$$= \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})\mathbf{x}_t + \sqrt{\bar{\alpha}_{t-1}}(1-\alpha_t)\frac{\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\odot\epsilon}{\sqrt{\bar{\alpha}_t}}}{1-\bar{\alpha}_t} \tag{22.47}$$

$$\vdots \tag{22.48}$$

$$= \frac{1}{\sqrt{\alpha_t}}\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}\sqrt{\alpha_t}}\boldsymbol{\epsilon}_0 \tag{22.49}$$

Thus, we can force $\boldsymbol{\mu}_\theta(\mathbf{x}_t, t)$, which has no dependency on $\mathbf{x}_0$ term, to match the $\boldsymbol{\mu}_q$. Since the $\mathbf{x}_t$ is given at training time, we just need to predict the $\boldsymbol{\epsilon}_t$. Then, we can express $\boldsymbol{\mu}_\theta(\mathbf{x}_t, t)$ as follows:

$$\boldsymbol{\mu}_\theta(\mathbf{x}_t, t) = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right)$$

Thus, $\mathbf{x}_{t-1} \sim p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$ can be expressed as follows:

$$\mathcal{N}\left(\mathbf{x}_{t-1}; \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right), \boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t)\right)$$

Note that we can use the variance derived in Eq. (22.45) instead of estimating it from a network for simplicity. Finally, we can solve the KL-divergence term:

$$\mathcal{L} = \min_{\theta} D_{KL}(q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)\|p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)) \tag{22.50}$$

$$\vdots \tag{22.51}$$

$$= \min_{\theta} \frac{1}{2\sigma_q^2(t)} \frac{(1-\alpha_t)^2}{(1-\bar{\alpha}_t)\alpha_t} \left[\|\boldsymbol{\epsilon}_0 - \hat{\boldsymbol{\epsilon}}_\theta(\mathbf{x}_t, t)\|_2^2\right]. \tag{22.52}$$

Here $\hat{\boldsymbol{\epsilon}}_\theta(\mathbf{x}_t, t)$ is a is a neural network that learns to predict the source noise $\boldsymbol{\epsilon}_0 \sim \mathcal{N}(\boldsymbol{\epsilon}\mathbf{0}, \mathbf{I})$ that determines $\mathbf{x}_t$ from $\mathbf{x}_0$. We have therefore shown that the overall learning objective is equivalent to learning to predict the noise.

22.5 Summary

- The loss function can be decomposed.

$$L_{\text{VLB}} = L_T + L_{T-1} + \cdots + L_0.$$

- $L_T = D_{\text{KL}}(q(\mathbf{x}_T|\mathbf{x}_0) \parallel p_\theta(\mathbf{x}_T))$
  - Constant ≈ 0 since x_T is a Gaussian noise.
- $L_t = D_{\text{KL}}(q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) \parallel p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t))$ for $t > 1$
  - This is the main part.
- $L_0 = -\log p_\theta(\mathbf{x}_0|\mathbf{x}_1)$
  - Can be modeled by a separate decoder.
- $q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)I)$
- $q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I})$,

  We can sample by $\mathbf{x}_t = \sqrt{1-\beta_t}\mathbf{x}_{t-1} + \sqrt{\beta_t}\epsilon$
- $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t))$.
  - We need to learn mean and variance.
  - DDPM kept the variance fixed and let the neural network only learn the mean μ_θ.
  - $\boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t)) = \sigma_t^2\mathbf{I}$ and set $\sigma_t^2 = \beta_t$.
  - Improved DDPM model trains σ also.
- One can reparameterize the mean to make the nerual network learn the added noise via a network ϵ_θ.

$$\mu_\theta(\mathbf{x}_t, t) = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\underbrace{\epsilon_\theta(\mathbf{x}_t, t)}_{\text{Network}}\right)$$

- Final objective function L_t is

$$||\epsilon - \epsilon_\theta(\mathbf{x}_t, t)||^2 = ||\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{(1-\bar{\alpha}_t)}\epsilon, t)||^2$$

  - $t \sim \text{Unif}[\{1, .., T\}]$
  - $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{(1-\bar{\alpha}_t)}\epsilon \sim q(\mathbf{x}_t|\mathbf{x}_0)$
  - $\epsilon \sim \mathcal{N}(0, I)$
- $\mathbf{x}_t$ is perturbed by ϵ and the noise prediction network ϵ_θ predicts ϵ.

Algorithm 6: Training

repeat
  $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
  $t \sim \text{Unif}[\{1, \cdots, T\}]$
  $\epsilon \sim \mathcal{N}(0, I)$
  Take gradient descent step on $\nabla_\theta ||\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{(1-\bar{\alpha}_t)}\epsilon, t)||^2$
until *converged*;

The training process is given by

1. $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
2. Sample a noise level t between 1 and T (*i.e.,* random time step).
3. Sample a noise from a Gaussian distribution and perturb the input by the sampling equation.
4. NN is trained to predict this noise ϵ used for generating $\mathbf{x}_t$.
5. β is often scheduled linearly.
6. Σ is set equal to β.

The sampling process is given by

Algorithm 7: Sampling

$\mathbf{x}_T \sim \mathcal{N}(0, I)$
for $t = T, \cdots, 1$ **do**
  $\mathbf{z} \sim \mathcal{N}(0, I)$
  $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right) + \Sigma_t \mathbf{z}$
return $\mathbf{x}_0$

- Ancestral sampling.
- T is typically around 1,000

22.6 Score Matching

- Suppose $\{\mathbf{x}_0, \cdots, \mathbf{x}_N\}$, where each data point (*e.g.,* image, video, or text)) is sampled independently from a data distribution $p(\mathbf{x})$.
- Given the dataset, the goal of generative modeling is to fit a model to the data distribution such that we can synthesize new data points at will by sampling from the model.
- One way is to directly model the distribution function as in likelihood-based models. Let $f_\theta(\mathbf{x}) \in \mathbb{R}^d$, then we can define a density function:

$$p_\theta(\mathbf{x}) = \frac{\exp^{-f_\theta(\mathbf{x})}}{Z_\theta}$$

.

- $f_\theta(\mathbf{x}) \in \mathbb{R}^d$ is often called unnormalized probabilistic model or energy-based model.
- Energy-based model originates from the Gibbs distribution in statistical physics.
- $p_\theta(\mathbf{x})$ can be trained by maximizing the log-likelihood of the data.

$$\max_\theta \sum_i^N \log p_\theta(\mathbf{x}_i).$$

- The gradient of the loglikelihood is given by

$$\begin{aligned}
\nabla_\theta \log p_\theta(\mathbf{x}) &= \nabla_\theta f_\theta(\mathbf{x}) - \nabla_\theta Z_\theta \\
\nabla_\theta Z_\theta &= \frac{\nabla_\theta Z_\theta}{Z_\theta} \\
&= \frac{1}{Z_\theta} \nabla_\theta \int \exp(f_\theta(\mathbf{x})) d\mathbf{x} \\
&= \frac{1}{Z_\theta} \int \exp(f_\theta(\mathbf{x})) \nabla_\theta f_\theta(\mathbf{x}) d\mathbf{x} \\
&= \int \frac{1}{Z_\theta} \exp(f_\theta(\mathbf{x})) \nabla_\theta f_\theta(\mathbf{x}) d\mathbf{x} \\
&= \int p_\theta(\mathbf{x}) \nabla_\theta f_\theta(\mathbf{x}) d\mathbf{x} \\
&= \mathbb{E}_{p_\theta(\mathbf{x})}[\nabla_\theta f_\theta(\mathbf{x})] \\
\nabla_\theta \log p_\theta(\mathbf{x}) &= \nabla_\theta f_\theta(\mathbf{x}) - \mathbb{E}_{p_\theta(\mathbf{x})}[\nabla_\theta f_\theta(\mathbf{x})]
\end{aligned}$$

- However, it is undesirable, since Z_θ is intractable.
  - For instance, a gray scale image of 100×100 has $256^{10,000}$ space.
- Thus, we have to sidestep the issue by using some solutions, for instance:
  - Approximate by using VAE or MCMC

Instead, we can leverage ***Stein Score***:

- By modeling a score function, instead of the density function, we can sidestep the difficulty of computing the intractable normalizaing constants.

- *Stein Score* function: $\nabla_{\mathbf{x}} \log p(\mathbf{x})$.
  - **Not a gradient w.r.t. model parameters.**
  - Gradient of the log probability density function.
  - Not same as the score in stat.
- It is a direction that maximizes a log data density.
- A model for approximating the score function is called a *score-based model* $s_\theta(\mathbf{a})$.
- Score-based models does not have to compute the intractable normalizing constant, Z_θ.

$$s_\theta(\mathbf{x}) = \nabla_{\mathbf{x}} \log p_\theta(\mathbf{x}) = -\nabla_{\mathbf{x}} f_\theta(\mathbf{x}) - \underbrace{\nabla_{\mathbf{x}} \log Z_\theta}_{\text{Constant}}.$$

22.6.1 Fisher Divergence

We need to know about *Fisher Divergence*:

- Given *i.i.d.* samples $\{\mathbf{x}_1, \cdots, \mathbf{x}_N\} \sim p_{data}(\mathbf{x}) = p(\mathbf{x})$.
- Estimating the score function $\nabla_{\mathbf{x}} \log p(\mathbf{x})$.
- Score model $s_\theta(\mathbf{x}) : \mathbb{R}^D \to \mathbb{R}^D$.
- Use score estimator $s_\theta(x)$:

$$\mathcal{L}_\theta = \frac{1}{2}\mathbb{E}_{p(\mathbf{x})}\left[||\nabla_{\mathbf{x}} \log p(\mathbf{x}) - s_\theta(\mathbf{x})||_2^2\right].$$

- It is called *Fisher divergence*.
- Intuitively, the Fisher divergence compares the squared distance between the ground-truth data score and the score-based model.
  - It changes the problem into a *regression* problem.
- Direct computation of the divergence is **infeasible** due to the unknown data score $\nabla_{\mathbf{x}} \log p(\mathbf{x})$.
  - Since we have no access to the true data distribution $p(\mathbf{x})$.

Fortunately, there exists a family of methods called ***score matching*** that minimize the Fisher divergence without knowledge of the ground-truth data score.

- Score matching objectives can directly be estimated on a dataset and optimized with stochastic gradient descent, analogous to the log-likelihood objective for training likelihood-based models (with known normalizing constants).
- We can train the score-based model by minimizing a score matching objective, without requiring adversarial optimization.

$$\mathcal{L}_\theta = \mathbb{E}_{p(\mathbf{x})}\left[\frac{1}{2}||s_\theta(x)||_2^2 + tr(\nabla_{\mathbf{x}} s_\theta(x))\right]$$
$$\approx \frac{1}{N}\sum_{i=1}^{N}\left[\frac{1}{2}||s_\theta(x)||_2^2 + tr(\nabla_{\mathbf{x}} s_\theta(x))\right]$$

- $\{\mathbf{x}_1, \cdots, \mathbf{x}_N\} \sim p(\mathbf{x})$
- $\nabla_{\mathbf{x}} s_\theta(x)$: Jacobian
- Remove the dependency of $p(\mathbf{x})$

$$\mathcal{L}_\theta = \frac{1}{2}\mathbb{E}_{p(x)}\left[||\nabla_x \log p(x) - s_\theta(x)||_2^2\right] \tag{22.53}$$
$$= \frac{1}{2}\mathbb{E}_{p(x)}\left[\left(\nabla_x \log p(x) - s_\theta(x)\right)^2\right] \tag{22.54}$$
$$= \frac{1}{2}\int p(x)(\nabla_x \log p(x) - s_\theta(x))^2 dx \tag{22.55}$$
$$= \underbrace{\frac{1}{2}\int p(x)(\nabla_x \log p(x))^2 dx}_{\text{independent from theta}} + \frac{1}{2}\int p(x) s_\theta(x)^2 dx - \int p(x) s_\theta(x) \nabla_x \log p(x) dx \tag{22.56}$$
$$= \cdots - \int p(x) s_\theta(x) \nabla_x \log p(x) dx \tag{22.57}$$
$$= \cdots - \int p(x) s_\theta(x) \frac{\nabla_x p(\mathbf{x})}{p(x)} dx \tag{22.58}$$
$$= \cdots - \int \nabla_{\mathbf{x}} p(x) s_\theta(x) dx \tag{22.59}$$
$$= \cdots - \left[p(x) s_\theta(x)\right]_{x=-\infty}^{\infty} + \int p(x) \nabla_x s_\theta(x) dx \tag{22.60}$$
$$= \frac{1}{2}\int p(x) s_\theta(x)^2 dx + \int p(x) \nabla_x s_\theta(x) dx + \text{const} \tag{22.61}$$
$$= \frac{1}{2}\mathbb{E}_{p(x)}[s_\theta(x)^2] + \mathbb{E}_{p(x)}[\nabla_x s_\theta(x)] + \text{const.} \tag{22.62}$$

- The second last term used the integration by parts.
- The last step is done by a boundary condition assumption which makes score function to be zero (*c.f.,* Sliced score matching paper).
  - $p_{data}(x) \to 0$ as $|x| \to \infty$.
  - In other words, gradient vanishes on the boundary.

22.6.2 Langevin Dynamics

- Once we have trained a score-based model $s_\theta(\mathbf{x}) \approx \nabla_{\mathbf{x}} \log p(\mathbf{x})$, we can use an iterative procedure called *Langevin Dynamics* (LD) [**?**] to draw samples from it.
- LD provides an MCMC procedure to sample from a distribution $p(\mathbf{x})$ using only its score function.

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} + \frac{\epsilon}{2}\nabla_{\mathbf{x}} \log p(\mathbf{x}_{t-1}) + \sqrt{\epsilon}\mathbf{z}_t$$

- Specifically, it initializes the chain from an arbitrary prior distribution $\mathbf{x}_0 \sim \pi(\mathbf{x})$, and then iterates the following
- Sample from $p(x)$ using only the score $\nabla_x \log p(x)$.
- $\mathbf{z}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$.
- ϵ is the step size.
- As $T \to \infty$ and $\epsilon \to 0$, $\mathbf{x}_T$ will become the true probability density $p(\mathbf{x})$.

Part V

Natural Language Processing

Chapter 23

Introduction

23.1 Evaluation Metrics

- Recall: TP/(TP+FN). Find all relevant cases whithin a dataset.
- Precision: TP/(TP+FP): While recall expresses the ability to find all relevant instances in a dataset, precision expresses the proportion of the data points our model says was relevant actually were relevant.
- The F1 score is the harmonic mean of precision and recall taking both metrics into account in the following equation:

23.1.1 Perplexity

Intuitively, perplexity can be understood as a *measure of uncertainty*. The perplexity of a language model can be seen as the level of perplexity. Consider a language model with an entropy of three bits, in which each bit encodes two possible outcomes of equal probability. This means that when predicting a symbol, that language model has to choose among $2^3 = 8$ possible options. Thus, we can argue that this language model has a perplexity of 8.

It can be modeled as $2^H(P, Q)$:

$$\begin{aligned} PPL(W) &= P(w_1, \cdots, w_N)^{-\frac{1}{N}} \\ &\approx \left(\prod_{i=1}^{N} P(w_i | w_{<i}) \right)^{-\frac{1}{N}} \\ &= \sqrt[n]{\frac{1}{\prod_{i=1}^{N} P(w_i | w_{<i})}} \end{aligned}$$

Let's derive it from a cross-entropy. We want to optimize P_θ instead of the true distribution P:

$$\begin{aligned}
\mathcal{L}_{CE} &= -\mathbb{E}_{w\sim P}[P_\theta(w_i|w_{<i})] && (23.1)\\
&\approx -\frac{1}{N}\sum_{i=1}^{N}\log P_\theta(w_i|w_{<i}) && (23.2)\\
&= -\frac{1}{N}\log\prod_{i=1}^{N}P_\theta(w_i|w_{<i}) && (23.3)\\
&= \log\left(\prod_{i=1}^{N}P_\theta(w_i|w_{<i})\right)^{-\frac{1}{N}} && (23.4)\\
&= \log\sqrt[N]{\frac{1}{\prod_{i=1}^{N}P_\theta(w_i|w_{<i})}} && (23.5)\\
& && (23.6)
\end{aligned}$$

Thus, $PPL(W) = \exp\Big(\mathcal{L}_{CE}\Big)$.

23.1.2 Cross-Entropy and Perplexity

$$\begin{aligned}
H(P,Q) &= -\sum_x P(x)\log Q(x)\\
&= -\sum_x P(x)[\log P(x) + \log Q(x) - \log P(x)]\\
&= -\sum_x P(x)\left[\log P(x) + \log\frac{Q(x)}{P(x)}\right]\\
&= H(P) + D_{KL}(P||Q)
\end{aligned}$$

It should be noted that since the empirical entropy $H(P)$ is unoptimizable, when we train a language model with the objective of minimizing the cross entropy loss, the true objective is to minimize the KL-divergence of the distribution, which was learned by our language model from the empirical distribution of the language.

Chapter 24

Classical NLP Techniques

24.1 Edit Distance

Edit distance is a method used in spell correction to determine how similar two words are by calculating the minimum number of operations (insertions, deletions, or substitutions) required to transform one word into another. The smaller the edit distance, the more similar the two words are.

For example, let's say we have the following dictionary of valid words: "cat", "car", "cart", "care", "cards", "cast". If the input word is "carr", we calculate the edit distance between "carr" and each word in the dictionary as follows:

- cat: 3 (insert "r" and "r", then delete "t")
- car: 1 (replace second "r" with "t")
- cart: 2 (insert "t" and delete second "r")

The smallest edit distance is 1, between "carr" and "car", so "car" would be selected as the corrected word.

The edit distance method can be improved by using techniques such as weighting the importance of each operation (insertion, deletion, substitution) or using a more sophisticated algorithm such as *Levenshtein distance*. Additionally, the method can be combined with other methods such as language modeling or phonetic analysis to further improve spell correction accuracy.

24.2 Point-wise Mutual Information

Point-wise Mutual Information (PMI) is a statistical measure to calculate the association between two words in a given corpus. PMI is calculated by comparing the probability of the co-occurrence of two words with their individual probabilities of occurrence.

Formally, it is a quantity which is closely related to the mutual information is the point-wise

mutual information. For two events (not random variables) x and y, this is defined as

$$\text{PMI}[x, y] \triangleq \log \frac{p(x, y)}{p(x)p(y)} \tag{24.1}$$

$$= \frac{p(x|y)}{p(x)} = \frac{p(y|x)}{p(y)} \tag{24.2}$$

This measures the discrepancy between these events occurring together compared to what would be expected by chance.

- x and y are two words being considered,
- $P(x)$ is the probability of the occurrence of word x in the corpus,
- $P(y)$ is the probability of the occurrence of word y in the corpus, and
- $P(x, y)$ is the probability of the co-occurrence of words x and y in the corpus. In other words, x and y are adjacent

For terms with three words, the formula becomes:

$$\text{PMI}[x, y, z] = \log \frac{p(x, y, z)}{p(x)p(y)p(z)}$$

PMI values can range from $-\infty$ to ∞. Positive PMI values indicate that the words have a strong association, while negative values indicate that the words are unlikely to appear together.

For example, consider a small corpus of text:

- "The cat sat on the mat. The dog sat on the mat."
- The matrix below is 6×6 considering all possible combination of the "forward" co-occurrences.

$$\begin{bmatrix} & the & cat & dog & sat & on & mat \\ the & 0 & 1 & 1 & 0 & 0 & 2 \\ cat & 0 & 0 & 0 & 1 & 0 & 0 \\ dog & 0 & 0 & 0 & 1 & 0 & 0 \\ sat & 0 & 0 & 0 & 0 & 2 & 0 \\ on & 2 & 0 & 0 & 0 & 0 & 0 \\ mat & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

24.2.1 Remove Stopwords prior to PMI

In the above example we have not removed stopwords, so some of you might be wondering if we need to remove stopwords prior to PMI. It depends on the problem statement but if your objective is to find the related words, you should remove stopwords prior to calculating PMI.

24.3 TF-IDF

The term TF stands for term frequency, and the term IDF stands for inverse document frequency.

The TF-IDF representation takes into account the importance of each word in a document. In the bag-of-words model, each word is assumed to be equally important, which is obviously a less accurate assumption.

The method to calculate the TF-IDF weights of a term in a document is given by the following formula:

- **Term Frequency** (TF), $tf(t,d)$, is the relative frequency of term t within document d,

$$tf(t,d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}},$$

  where $f_{t,d}$ is the raw count of a term in a document, *i.e.*,, the number of times that term t occurs in document d. Note the denominator is simply the total number of terms in document d (counting each occurrence of the same term separately). There are various other ways to define term frequency:

  - The raw count itself: $tf(t,d) = f_{t,d}$
  - Boolean "frequencies":

$$tf(t,d) = \begin{cases} 1, & \text{if } t \text{ occurs in } d \\ 0 & \text{otherwise} \end{cases}$$

  - Logarithmically scaled frequency: $tf(t,d) = \log(1 + ft,d)$
  - Augmented frequency, to prevent a bias towards longer documents, *e.g.*,. raw frequency divided by the raw frequency of the most frequently occurring term in the document:

$$tf(t,d) = 0.5 + 0.5\frac{f_{t,d}}{\max\{f_{t',d} : t' \in d\}}$$

- **Inverse document frequency (IDF)**: *The IDF is a measure of how much information the word provides*, *i.e.*,, if it is common or rare across all documents. It is the logarithmically scaled inverse fraction of the documents that contain the word, which is obtained by dividing the total number of documents by the number of documents containing the term, and then taking the logarithm of that quotient:

$$idf(t,D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

  - N total number of documents in the corpus ($= |D|$).
  - $|\{d \in D : t \in d\}|$: number of documents where the term t appears. If the term is not in the corpus, this will lead to a division-by-zero. It is therefore common to adjust the numerator $1 + N$ and denominator to $1 + |\{d \in D : t \in d\}|$.
  - Similarly, IDF can also be expressed

$$idf(t,D) = \log \frac{N - n + 0.5}{n + 0.5} + 1,$$

    where n is the number of documents containing the term t.

24.3.1 Term frequency–inverse document frequency

TF-IDF is calculated as a multiplication of $tf(t, D)$ and $idf(t, D)$.

- A high weight in TF–IDF is reached by a high term frequency (in the given document) and a low document frequency of the term in the whole collection of documents; the weights hence tend to filter out common terms.
- Since the ratio inside the IDF's log function is always greater than or equal to 1, the value of IDF (and TF–IDF) is greater than or equal to 0.
- As a term appears in more documents, the ratio inside the logarithm approaches 1, bringing the IDF and TF–IDF closer to 0.

In short, TF

- This measures the frequency of a word in a document. If a word appears more times in a document, it is likely more important to the document.
- number of times term "word" appears in a document / Total number of terms in the document
- Consider a document containing 100 words wherein the word 'cat' appears 3 times, then TF is 0.03

IDF:

- This measures the importance of a word in the entire corpus. If a word appears in many documents, it is likely less important to any individual document.
- $\log_e$ total number of documents/ Number of documents with term "word" in it.
- If we have 10 million documents, and 'cat' appears in 1,000 of these. Then, IDF is $\log 10,000,000/1,000 = 4$

Finally, TF-IDF is $0.03 \times 4 = 0.12$ The higher the TF-IDF score, the rarer the term and vice versa.

24.3.2 Link with Information Theory

This expression shows that summing the TF–IDF of all possible terms and documents recovers the mutual information between documents and term taking into account all the specificities of their joint distribution. Each TF–IDF hence carries the "bit of information" attached to a term x document pair.

24.4 Best Match 25 Ranking Algorithm

Best Match 25 ranking algorithm or BM25 is a bag-of-words retrieval function that ranks a set of documents based on the query terms appearing in each document, regardless of their proximity

within the document. It is a family of scoring functions with slightly different components and parameters. One of the most prominent instantiations of the function is as follows.

Given a query Q, containing keywords $q_1, \ldots, q_n$, the BM25 score of a document D is given by

$$\text{score}(D, Q) = \sum_{i=1}^{n} IDF(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{avgd1})},$$

where $f(q_i, D)$ is the number of times that the keyword q_i occurs in the document, D, $|D|$ is the length of the document D in words, and $avgd1$ is the average document length in a collection of documents. k_1 and b are free parameters, usually chosen, in absence of an advanced optimization, as $k_1 \in [1.2, 2.0]$ and $b = 0.75$.

24.5 Reciprocal Rank Fusion

Reciprocal Rank Fusion (RRF), a simple method for combining the document rankings from multiple information retrieval systems.

$$RRF(d \in D) = \sum_{r \in R} \frac{1}{k + r(d)}$$

- k is a constant that helps to balance between high and low ranking. Typically set at 60.
- $r(d)$ is the rank/position of the document.

24.6 PageRank and TextRank

24.7 Label Smoothing

For each training example x, our model computes the probability of each label $k \in \{1, ..., K\}$, $p(k|x) = \frac{\exp(z_k)}{\sum_i \exp(z_i)}$. Here z_k are the logits.

Label smoothing is a mechanism to regularize the classifier layer by estimating the marginalized effect of label-dropout during training.

Vanila corss-entropy can cause two problem:

- First, it may result in over-fitting: if the model learns to assign full probability to the ground-truth label for each training example, it is not guaranteed to generalize.
- Second, it encourages the differences between the largest logit and all others to become large, and this, combined with the bounded gradient $\frac{\partial \ell}{\partial z_k}$, reduces the ability of the model to adapt. Intuitively, this happens because the model becomes too confident about its predictions.

We propose a mechanism for encouraging the model to be less confident. While this may not be desired if the goal is to maximize the log-likelihood of training labels, it does regularize the model and makes it more adaptable. The method is very simple. Consider a distribution over labels $u(k)$, independent of the training example x, and a smoothing parameter ϵ. For a training example with ground-truth label y, we replace the label distribution $q(k|x) = \delta_{k,y}$ with

$$q'(k|x) = (1 - \epsilon)\delta_{k,y} + \epsilon u(k)$$

which is a mixture of the original ground-truth distribution $q(k|x)$ and the fixed distribution $u(k)$, with weights $1 - \epsilon$ and ϵ, respectively. This can be seen as the distribution of the label k obtained as follows:

$$q'(k|x) = (1 - \epsilon)\delta_{k,y} + \frac{\epsilon}{K}$$

We refer to this change in ground-truth label distribution as label-smoothing regularization, or LSR.

24.7.1 Another Interpretation

Instead of using one-hot encoded vector, we introduce noise distribution $u(y|x)$. Our new ground truth label for data (x_i, y_i) would be

$$\begin{aligned} p'(y|x_i) &= (1 - \varepsilon)p(y|x_i) + \varepsilon u(y|x_i) \\ &= \begin{cases} 1 - \varepsilon + \varepsilon u(y|x_i) & \text{if } y = y_i \\ \varepsilon u(y|x_i) & \text{otherwise} \end{cases} \end{aligned}$$

Where ε is a weight factor, $\varepsilon \in [0, 1]$, and note that $\sum_{y=1}^{K} p'(y|x_i) = 1$.

Chapter 25

POS Tagging

25.1 Introduction

Part-of-speech (POS) tagging is the process of labeling words in a text with their corresponding parts of speech in natural language processing (NLP). It helps algorithms understand the grammatical structure and meaning of a text.

Chapter 26

Language Model

26.1 Language Model

26.2 Tokenizers

26.3 Byte Pair Encoding

26.4 Byte Pair Encoding

Chapter 27

Transformer

27.1 Attention Mechanism

The attention mechanism mimics the retrieval of a value v_i for a query q based on a key k_i in database.

$$attn(q, k, v) = \sum_i sim(q, k_i) \times v_i$$

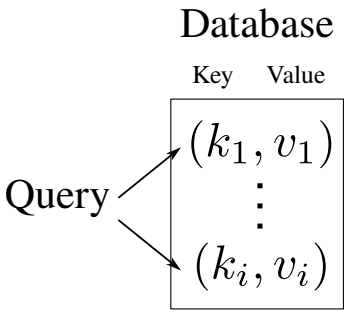


Figure 27.1: The most similar key will be selected by measuring a **similarity** between a query and a key.

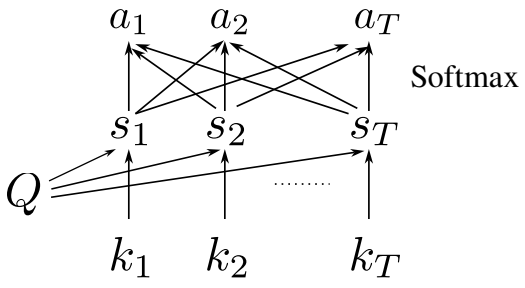


Figure 27.2: The similarity s_t is computed by a query and keys

There are several choices for a similarity function.

- $q^T k_i$: dot product.
- $\frac{q^T k_i}{\sqrt{d}}$: scaled dot product $\rightarrow$ reduce the variance of the final attention weights.

- $q^T W k_i$: general dot product.
- $w_q^T q + w_k^T k_i$: additive similarity.

Finally, the **attention score** can be computed by using a softmax:

$$a_i = \frac{\exp(s_i)}{\sum_j \exp(s_j)}$$

27.2 Transformer

27.2.1 Self-Attention

$$attn(Q, K, V) = softmax\left(\frac{Q^T K}{\sqrt{d_k}}\right)V$$

Here, $Q, K, V \in \mathbb{R}^{N \times d}$, QK^T's time complexity is $O(N^2 d)$. This quadratic cost is massive for long input-sequences such as documents to be summarized or character-level inputs.

27.2.2 Masked attention

The masked attention is often referred to cross-attention. This is just a self-attention in decoder.

$$\text{MA}(Q, K, V) = softmax\left(\frac{Q^T K + M}{\sqrt{d_k}}\right)V,$$

where M is a matrix of 0 and $-\infty$. Note that $-\infty$ will make *exp* term to be zero.

27.2.3 Skip Connection

This is a regularization technique.

27.2.4 Positional Embedding

```
def position_encoding(seq_len: int, dim_model: int)->Tensor:
    pos = torch.arange(seq_len, dtype=torch.float).reshape(1, -1, 1)
    dim = torch.arange(dim_model, dtype=torch.float).reshape(1, 1, -1)
    phase = pos / (1e4 ** (dim // dim_model))
    return torch.where(dim.long() % 2 == 0, torch.sin(phase), torch.cos(phase))
```

27.2.5 Encoder

27.2.6 Decoder

The output of each step is fed to the bottom decoder in the next time step, and the decoders bubble up their decoding results just like the encoders did. And just like we did with the encoder

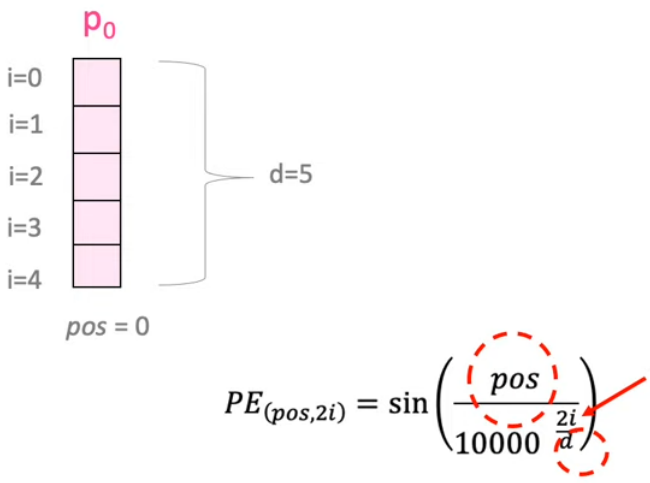


Figure 27.3: Positional embedding.

inputs, we embed and add positional encoding to those decoder inputs to indicate the position of each word.

```
def forward(self, tgt: Tensor, memory: Tensor) -> Tensor:
  seq_len, dimension = tgt.size(1), tgt.size(2)
  tgt += position_encoding(seq_len, dimension)
  for layer in self.layers:
    tgt = layer(tgt, memory)
  return torch.softmax(self.linear(tgt), dim=-1)
```

Chapter 28

Retrieval Augmented Generation

28.1 Introduction

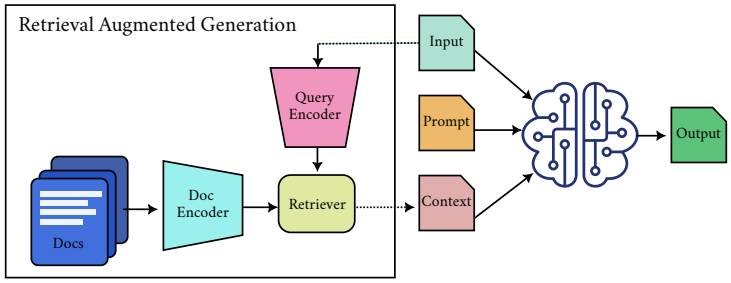


Figure 28.1: An overview of retrieval augmented generation.

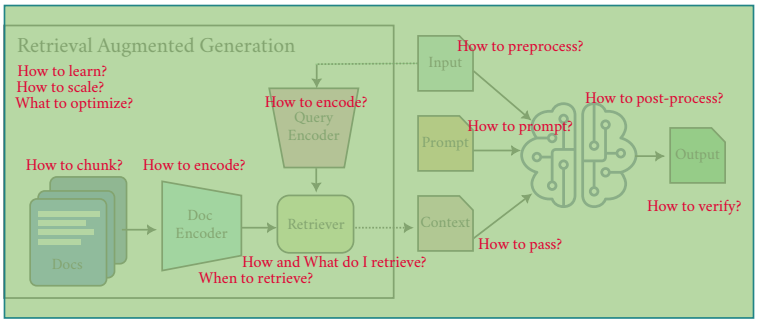


Figure 28.2: Open questions of RAG.

Chapter 29

Alignment Problems

29.1 Direct Preference Optimization

$$\mathcal{L}_{\mathrm{DPO}}(\pi_\theta; \pi_{\mathrm{ref}}) = -\mathbb{E}_{(x,y_w,w_l)\sim\mathcal{D}}\left[\log \sigma\left(\beta \log \frac{\pi_\theta(y_w|x)}{\pi_{\mathrm{ref}}(y_w|x)} - \beta \log \frac{\pi_\theta(y_l|x)}{\pi_{\mathrm{ref}}(y_l|x)}\right)\right]. \tag{29.1}$$

Part VI

Advanced Topics

Chapter 30

Neural Ordinary Differential Equations

30.1 Preliminary

30.1.1 Euler Method

The Euler method is a simple numerical technique used to solve ordinary differential equations (ODEs) of the form $\frac{dy}{dt} = f(t, y)$. It is an initial value problem where we seek to find the function $y(t)$ given an initial condition $y(t_0) = y_0$. Here's a step-by-step explanation of the Euler method:

Problem Setup: Given,

- A differential equation $\frac{dy}{dt} = f(t, y)$
- An initial condition $y(t_0) = y_0$

Discretization: The idea is to approximate the solution at discrete points. Let's denote:

- t_n as the n-th time step
- y_n as the approximation of $y(t_n)$

We define a step size h such that $t_{n+1} = t_n + h$.

Euler's Approximation: Using the first-order Taylor series expansion, we can approximate $y(t)$ at t_{n+1} as:

$$y_{n+1} \approx y_n + h \cdot f(t_n, y_n)$$

Iterative Process: Starting from the initial condition (t_0, y_0):

1. Calculate the next value using the formula:
$$y_{n+1} = y_n + h \cdot f(t_n, y_n)$$
2. Repeat the process for $n = 0, 1, 2, \ldots$ until the desired value of t is reached.

Example: Let's solve the differential equation $\frac{dy}{dt} = y$ with the initial condition $y(0) = 1$ using the Euler method.

- Set the step size h (e.g., $h = 0.1$).
- Start with $t_0 = 0$ and $y_0 = 1$.

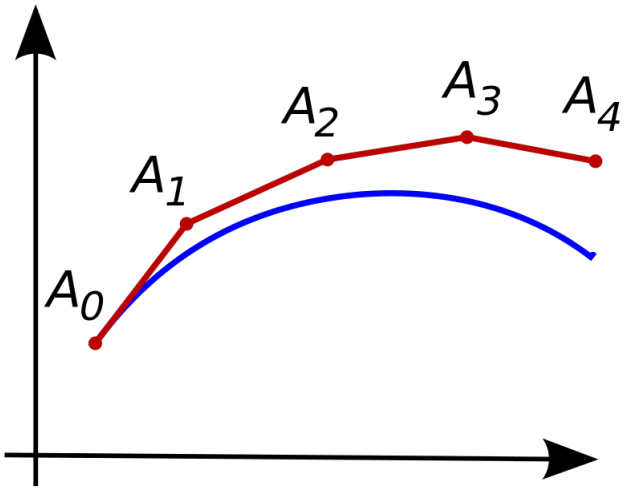
Using the Euler formula:

$$y_1 = y_0 + h \cdot f(t_0, y_0) = 1 + 0.1 \cdot 1 = 1.1$$

$$y_2 = y_1 + h \cdot f(t_1, y_1) = 1.1 + 0.1 \cdot 1.1 = 1.21$$

$$y_3 = y_2 + h \cdot f(t_2, y_2) = 1.21 + 0.1 \cdot 1.21 = 1.331$$

Euler's method can be visualized as taking small steps along the curve defined by the differential equation, using the slope at the current point to determine the direction of the next step.



Advantages:

- Simple to understand and implement.
- Requires only basic arithmetic operations.

Disadvantages:

- Low accuracy for large step sizes.
- Can become unstable if the step size is not chosen appropriately.
- Errors accumulate over time, leading to less accurate solutions.

Euler's method is often used as a basic introduction to numerical methods for solving ODEs, and more sophisticated methods like the *Runge-Kutta* methods are used for more accurate solutions.

30.2 Neural ODE

Models such as residual networks, recurrent neural network decoders, and normalizing flows build complicated transformations by composing a sequence of transformations to a hidden state:

$$\mathbf{h}_{t+1} = \mathbf{h}_t + f(\mathbf{h}_t, \theta_t).$$

The $\mathbf{h}$ is iteratively updated as follows:

$$\begin{aligned}
\mathbf{h}_2 &= \mathbf{h}_1 + f(\mathbf{h}_1, \theta) \\
\mathbf{h}_3 &= \mathbf{h}_2 + f(\mathbf{h}_2, \theta) = \mathbf{h}_1 + f(\mathbf{h}_1, \theta) + f(\mathbf{h}_2, \theta) \\
&\vdots
\end{aligned}$$

These iterative update can be seen as *Euler discretization* of the *continuous* transformation. Think of traditional neural networks as a sequence of steps. You give it some input, it goes through several steps (layers), and you get an output. Instead of thinking in steps, Neural ODEs think in **continuous change over time**. Note that this is the key contribution of this approach.

Euler method can be expressed as follows:

$$y_n = y_{n-1} + h\frac{\partial y_{n-1}}{\partial x_{n-1}},$$

where h is the step size. In NODE, they view the f as an ordinary differential equation, which depends on the state at time t and parameter θ. The following equation is the shape of Euler method:

$$y_n = y_1 + h\frac{\partial y_1}{\partial x_1} + h\frac{\partial y_2}{\partial x_2} + \cdots + h\frac{\partial y_{n-1}}{\partial x_{n-1}}.$$

In NODE,

Chapter 31

State Estimations

31.1 Introduction to State-Space Model

Reference: State-Space Models.

A state-space model is a mathematical framework used to describe a system by a set of input, output, and state variables related by first-order differential (or difference) equations. It's widely used in control theory, signal processing, and time series analysis.

For a continuous-time system, the state-space model is typically expressed as follows:

$$\dot{x}(t) = Ax(t) + Bu(t),$$
$$y(t) = Cx(t) + Du(t).$$

The first and the second equations are known as *state equation* and *output equation*, respectively. The state equation tells us that how the state vector changes with the state vector and the (external) input. The state space model is **linear** and **time invariant**, since the equations are linear and the parameter matrices do not change over time. Such systems are referred as a LTI system.

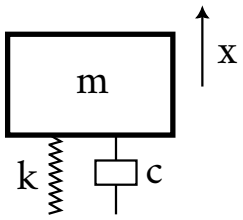
- $x(t) \in \mathbb{R}^n$: A state vector. This variable describes the state of the system at any given time.
- $\dot{x} \in \mathbb{R}^n$: state derivative *i.e.,* $\left(\frac{dX}{dt}\right)$ represents the changes of the state vector. You can notice that this is a linear combination of state vector and the input vector.
- $u \in \mathbb{R}^m$: An external input affecting the system.
- $y \in \mathbb{R}^p$: An (Observed) Output.
- $A \in \mathbb{R}^{n \times n}$: A state matrix. This matrix describes how the state vector influences the changes of the state vector.
- $B \in \mathbb{R}^{n \times m}$: An input matrix. This matrix describes how the (external) input vector influences the changes of the state vector.
- $C \in \mathbb{R}^{p \times n}$: An output matrix. Typically, an identity matrix (I).
- $D \in \mathbb{R}^{p \times m}$: A feedthrough (or direct transmission) matrix. Typically, zeros

31.1.1 Example: Mass-Spring-Damper System

Let's consider a simple example of a mass-spring-damper system, which can be described by the second-order differential equation.

The mass-spring-damper system is a common mechanical system that consists of three main components:

- Mass (m): A mass that can move along a straight line.
- Spring (k): A spring that exerts a force proportional to its displacement from its equilibrium position (Hooke's Law).
- Damper (c): A damping element (like a shock absorber) that exerts a force proportional to the velocity of the mass (damping force).



System Dynamics The dynamics of this system can be described by Newton's second law of motion, which states that the sum of forces acting on the mass is equal to the mass times its acceleration ($F = ma$).

Differential Equation For a mass-spring-damper system, the forces are:

- Spring Force: $F_{\text{spring}} = -kx(t)$, where $x(t)$ is the displacement of the mass from its equilibrium position. The proportional constant k is called the spring constant. It is a measure of the spring's stiffness.
- Damping Force: Damping forces are a special type of force that are used to slow down or stop a motion. $F_{\text{damper}} = -c\dot{x}(t)$, where $\dot{x}(t)$ is the velocity of the mass (or object). Note that $\dot{x}(t) = dx(t)/dt$.
- External Force: $F(t)$, an external force applied to the mass.

By summing these forces, we get:

$$m\ddot{x}(t) = -kx(t) - c\dot{x}(t) + F(t).$$

To explain, the external force is stretching the spring, and the damper and the spring force are pulling the mass. Rearranging this, we get the second-order differential equation:

$$m\ddot{x}(t) + c\dot{x}(t) + kx(t) = F(t)$$

State-Space Representation To convert this second-order differential equation into a state-space representation, we need to express it as a system of first-order differential equations.

Defining State Variables We introduce two state variables:

- $x_1(t) = x(t)$: the position of the mass.
- $x_2(t) = \dot{x}(t)$: the velocity of the mass.

Now, we can write the original second-order equation as two first-order equations:

$$\dot{x}_1(t) = x_2(t)$$

$$\dot{x}_2(t) = \frac{1}{m}F(t) - \frac{k}{m}x_1(t) - \frac{c}{m}x_2(t)$$

Matrix Form We can express these equations in matrix form:

$$\begin{bmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & -\frac{c}{m} \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} F(t)$$

This is the state-space form:

$$\dot{x}(t) = Ax(t) + Bu(t)$$

Where:

- $x(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix}$ is the state vector.
- $u(t) = F(t)$ is the input (external force).
- $A = \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & -\frac{c}{m} \end{bmatrix}$ is the state matrix.
- $B = \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix}$ is the input matrix.

Output Equation If we consider the output to be the position of the mass ($x_1(t)$), the output equation is:

$$y(t) = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix}$$

So, the output equation is:

$$y(t) = Cx(t),$$

Where $C = \begin{bmatrix} 1 & 0 \end{bmatrix}$.

Full State-Space Model Combining the state and output equations, we get the full state-space representation:

$$\dot{x}(t) = \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & -\frac{c}{m} \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} u(t)$$
$$y(t) = \begin{bmatrix} 1 & 0 \end{bmatrix} x(t)$$

In sum,

- State variables: $x_1(t) = x(t)$ (position), $x_2(t) = \dot{x}(t)$ (velocity)
- Input: $u(t) = F(t)$ (external force)
- Output: $y(t) = x(t)$ (position)
- State equation: $\dot{x}(t) = Ax(t) + Bu(t)$
- Output equation: $y(t) = Cx(t)$

This state-space model describes how the position and velocity of the mass change over time in response to an external force.

31.1.2 Stability

The linear state space model is stable if all eigenvalues of A are negative real numbers or have negative real parts to complex number eigenvalues. If all real parts of the eigenvalues are negative then the system is stable, meaning that any initial condition converges exponentially to a stable attracting point. If any real parts are zero then the system will not converge to a point and if the eigenvalues are positive the system is unstable and will exponentially diverge.

Chapter 32

Kalman Filter

32.1 Propagation of States and Covariances

Suppose we have a linear discrete-time system:

$$\boldsymbol{\theta}_k = F_{k-1}\boldsymbol{\theta}_{k-1} + G_{k-1}\mathbf{u}_{k-1} + \mathbf{w}_{k-1},$$

where $\mathbf{w}_k$ is a Gaussian zero-mean white noise with covariance Q_k and $\mathbf{u}_k$ is a known input. The F and G are state and input matrices, respectively. How does the mean of the state $\boldsymbol{\theta}_k$ changes over time? The expectation is given by

$$\mathbb{E}[\boldsymbol{\theta}_k] = \mathbb{E}[F_{k-1}\boldsymbol{\theta}_{k-1}] + \mathbb{E}[G_{k-1}\mathbf{u}_{k-1}] + \mathbb{E}[\mathbf{w}_{k-1}]$$

For simplicity, we can write it as

$$\overline{\boldsymbol{\theta}}_k = F_{k-1}\overline{\boldsymbol{\theta}}_{k-1} + G_{k-1}\mathbf{u}_{k-1}.$$

How does the state covariance of $\boldsymbol{\theta}_k$ change with time? The state covariance matrix propagation is given by

$$P_k = \mathbb{E}[(\boldsymbol{\theta}_k - \overline{\boldsymbol{\theta}}_k)(\boldsymbol{\theta}_k - \overline{\boldsymbol{\theta}}_k)^T]$$

and then compute the expectation of every term in that expression.

$$\begin{aligned}(\boldsymbol{\theta}_k - \overline{\boldsymbol{\theta}}_k)(\boldsymbol{\theta}_k - \overline{\boldsymbol{\theta}}_k)^T &= (F_{k-1}(\boldsymbol{\theta}_{k-1} - \overline{\boldsymbol{\theta}}_{k-1}) + \mathbf{w}_{k-1})(F_{k-1}(\boldsymbol{\theta}_{k-1} - \overline{\boldsymbol{\theta}}_{k-1}) + \mathbf{w}_{k-1})^T \\ &= F_{k-1}(\boldsymbol{\theta}_{k-1} - \overline{\boldsymbol{\theta}}_{k-1})(\boldsymbol{\theta}_{k-1} - \overline{\boldsymbol{\theta}}_{k-1})^T F_{k-1}^T + F_{k-1}(\boldsymbol{\theta}_{k-1} - \overline{\boldsymbol{\theta}}_{k-1})\mathbf{w}_{k-1}^T \\ &+ \mathbf{w}_{k-1}(\boldsymbol{\theta}_{k-1} - \overline{\boldsymbol{\theta}}_{k-1})^T F_{k-1}^T + \mathbf{w}_{k-1}\mathbf{w}_{k-1}^T\end{aligned}$$

Since $(\boldsymbol{\theta}_k - \overline{\boldsymbol{\theta}}_k)$ is uncorrelated to $\mathbf{w}_{k-1}$, we have

$$\begin{aligned}E[F_{k-1}(\boldsymbol{\theta}_{k-1} - \overline{\boldsymbol{\theta}}_{k-1})\mathbf{w}_{k-1}^T] = 0 \\ E[\mathbf{w}_{k-1}(\boldsymbol{\theta}_{k-1} - \overline{\boldsymbol{\theta}}_{k-1})^T F_{k-1}^T] = 0\end{aligned}$$

Also, we have

$$E[\mathbf{w}_{k-1}\mathbf{w}_{k-1}^T] = Q_{k-1}$$

$$P_{k-1} = E[(\boldsymbol{\theta}_{k-1} - \overline{\boldsymbol{\theta}}_{k-1})(\boldsymbol{\theta}_{k-1} - \overline{\boldsymbol{\theta}}_{k-1})^T]$$

By using these expressions, we obtain the final equation for the propagation of the state covariance matrix

$$P_k = F_{k-1} P_{k-1} F_{k-1}^T + Q_{k-1}$$

This is called a discrete time *Lyapunov equation*, or a *Stein equation*. It is interesting to consider the conditions under which the discrete time Lyapunov equation has a *steady-state solution*.

32.2 Kalman Filtering

We are considering the following state-space model of a dynamical system:

$$\begin{aligned} \boldsymbol{\theta}_k &= F_{k-1}\boldsymbol{\theta}_{k-1} + G_{k-1}\mathbf{u}_{k-1} + \mathbf{w}_{k-1} \\ \mathbf{y}_k &= \mathbf{X}_k\boldsymbol{\theta}_k + \boldsymbol{\eta}_k, \end{aligned}$$

where

- $\mathbf{w}_k \sim (0, Q_k)$: process noise vector
- $E[\mathbf{w}_k \mathbf{w}_j^T] = Q_k \delta_{k-j}$
- $\boldsymbol{\eta}_k \sim (0, R_k)$: measurement noise vector
- $E[\boldsymbol{\eta}_k \boldsymbol{\eta}_j^T] = R_k \delta_{k-j}$
- $E[\boldsymbol{\eta}_k \mathbf{w}_j^T] = 0$
- $\mathbf{u}_{k-1}$: control unit vector
- F and G are state and input matrices.

Our primary goal is to estimate the state $\boldsymbol{\theta}_k$ based on our knowledge of the system dynamics and the availability of the noisy measurements $\mathbf{y}_k$.

If we have all of the measurements up to and including time k available for use in our estimate of $\boldsymbol{\theta}_k$, then we can form an a *posteriori estimate*, which we denote as $\hat{\boldsymbol{\theta}}_k^+$. The + superscript denotes that the estimate is a posteriori. One way to form the a posteriori state estimate is to compute the expected value of $\boldsymbol{\theta}_k$ conditioned on a ll of the measurements up to and including time k:

$$\hat{\boldsymbol{\theta}}_k^+ = E[\boldsymbol{\theta}_k | \mathbf{y}_1, \ldots, \mathbf{y}_{k-1}, \mathbf{y}_k].$$

If we have all of the measurements before (but not including) time k available for use in our estimate of $\boldsymbol{\theta}_k$, then we can form an a praori estimate, which we denote

$$\hat{\boldsymbol{\theta}}_k^-.$$

One way to form the a *priori state estimate* is to compute the expected value of $\boldsymbol{\theta}_k$, conditioned on all of the measurements before (but not including) time k:

$$\hat{\boldsymbol{\theta}}_k^- = E[\boldsymbol{\theta}_k | \mathbf{y}_1, \ldots, \mathbf{y}_{k-1}].$$

By their nature, it is natural to expect that $\hat{\boldsymbol{\theta}}_k^+$ to be a better estimate than $\hat{\boldsymbol{\theta}}_k^-$, since we can leverage more information to compute it.

If we have measurements after time k available for use in our estimate of $\boldsymbol{\theta}_k$, then we can form a *smoothed estimate*. One way to form the smoothed state estimate is to compute the expected value of $\boldsymbol{\theta}_k$ conditioned on all of the measurements that are available:

$$\hat{\boldsymbol{\theta}}_{k|k+N} = E[\boldsymbol{\theta}_k|\mathbf{y}_1, \ldots, \mathbf{y}_k, \ldots, \mathbf{y}_{k+N}],$$

where N is some positive integer. If we want to find the best prediction of $\boldsymbol{\theta}_k$ more than one time step ahead of the available measurements, then we can form a predicted estimate. One way to form the *predicted state estimate* is to compute the expected value of $\boldsymbol{\theta}_k$ conditioned on all of the measurements that are available:

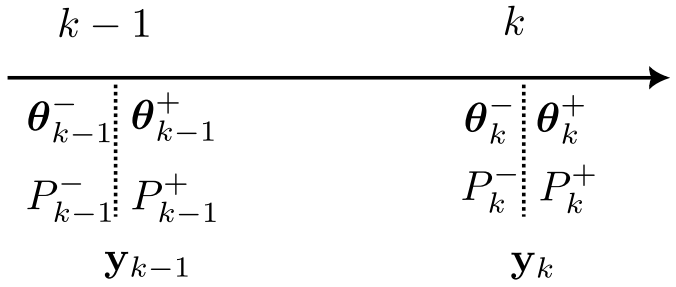
$$\hat{\boldsymbol{\theta}}_{k|k-M} = E[\boldsymbol{\theta}_k|\mathbf{y}_1, \ldots, \mathbf{y}_{k-M}].$$

The relationship between the a posteriori, a priori, smoothed, and predicted state estimates is represented in the following figure:

As done in RLS (*c.f.,* §6.1), we should define covariance matrices of the a priori and the a posteriori estimation errors. The covariance matrices are defined as follows:

$$P_k^- = \mathbb{E}[(\boldsymbol{\theta}_k - \boldsymbol{\theta}_k^-)(\boldsymbol{\theta}_k - \boldsymbol{\theta}_k^-)^T]$$
$$P_k^+ = \mathbb{E}[(\boldsymbol{\theta}_k - \boldsymbol{\theta}_k^+)(\boldsymbol{\theta}_k - \boldsymbol{\theta}_k^+)^T],$$

where P_k^- is the covariance matrix of the a priori estimation error and P_k^+ is the covariance matrix of the a posteriori estimation error.



After the discrete-time instant $k-1$, we compute the a posteriori state estimate $\boldsymbol{\theta}_{k-1}^+$ and the covariance matrix of the estimation error P_{k-1}^+. Then, we propagate these quantities through the equations describing the system dynamics and covariance matrix propagation (that is also derived on the basis of the system dynamics). That is, we propagate the a posteriori state and covariance through our model. By propagating these quantities, we obtain the a priori state estimate $\boldsymbol{\theta}_k^-$ and the covariance matrix of the estimation error P_k^- for the time step k. Then, after the measurement vector $\mathbf{y}_k$ is observed at the discrete-time step k, we use this measurement and the recursive-least squares method to compute the a posteriori estimate $\boldsymbol{\theta}_k^+$ and the covariance matrix of the estimation error P_k^+ for the time instant k.

This is how Kalman filter works. Now let's derive the equations of Kalman filter. At the initial time instant $k = 0$, we need to set an initial guess of the state estimate, $\hat{\boldsymbol{\theta}}_0$. This will be our initial a posteriori state estimate, that is

$$\hat{\boldsymbol{\theta}}_0^+ = \hat{\boldsymbol{\theta}}_0$$

Then, the question is how to compute the a priori estimate $\hat{\boldsymbol{\theta}}_1^-$ at $k = 1$. The natural answer is that the system states, as well as the estimates, need to satisfy the system dynamics (1), and consequently,

$$\hat{\boldsymbol{\theta}}_1^- = F_0\hat{\boldsymbol{\theta}}_0^+ + F_0\mathbf{u}_0$$

where we excluded the disturbance part since the disturbance vector is not known. Besides the initial guess of the estimate, we also need to select an initial guess of the covariance matrix of the estimation error. That is, we need to select P_0^+. If we have perfect knowledge about the initial state, then we select P_0^+ as a zero matrix, that is $P_0^+ = 0 \cdot I$, where I is an identity matrix. This is because the covariance matrix of the estimation error is the measure of uncertainty, and if we have perfect knowledge, then the measure of uncertainty should be zero. On the other hand, if we do not have any a priori knowledge of the initial state, then $P_0^+ = c \cdot I$, where c is a large number. Next, we need to compute the covariance matrix of the a priori state estimation error, that is, we need to compute P_1^-. In our previous post, which can be found here, we derived the expression for the time propagation of the covariance matrix of the state estimation error:

$$P_k = F_{k-1} P_{k-1} F_{k-1}^T + Q_{k-1}$$

By using this expression, we obtain the following equation forP_1^-.

32.2.1 Python Implementation

```
class KalmanFilter(object):

  # x0 - initial guess of the state vector
  # P0 - initial guess of the covariance matrix of the state estimation error
  # A,B,C - system matrices describing the system model
  # Q - covariance matrix of the process noise
  # R - covariance matrix of the measurement noise

  def __init__(self,x0,P0,A,B,C,Q,R):

      # initialize vectors and matrices
      self.x0=x0
      self.P0=P0
      self.A=A
      self.B=B
      self.C=C
      self.Q=Q
      self.R=R

      # this variable is used to track the current time step k of the
  estimator
      # after every measurement arrives, this variables is incremented for +1
      self.currentTimeStep=0

      # this list is used to store the a posteriori estimates xk^{+} starting
   from the initial estimate
      # note: list starts from x0^{+}=x0 - where x0 is an initial guess of
  the estimate
      self.estimates_aposteriori=[]
      self.estimates_aposteriori.append(x0)

      # this list is used to store the a apriori estimates xk^{-} starting
  from x1^{-}
      # note: x0^{-} does not exist, that is, the list starts from x1^{-}
      self.estimates_apriori=[]

      # this list is used to store the a posteriori estimation error
  covariance matrices Pk^{+}
      # note: list starts from P0^{+}=P0, where P0 is the initial guess of
  the covariance
      self.estimationErrorCovarianceMatricesAposteriori=[]
```

```
36          self.estimationErrorCovarianceMatricesAposteriori.append(P0)
37
38          # this list is used to store the a priori estimation error covariance
    matrices Pk^{-}
39          # note: list starts from P1^{-}, that is, P0^{-} does not exist
40          self.estimationErrorCovarianceMatricesApriori=[]
41
42          # this list is used to store the gain matrices Kk
43          self.gainMatrices=[]
44
45          # this list is used to store prediction errors error_k=y_k-C*xk^{-}
46          self.errors=[]
47
48      # this function propagates x_{k-1}^{+} through the model to compute x_{k
    }^{-}
49      # this function also propagates P_{k-1}^{+} through the covariance model to
         compute P_{k}^{-}
50      # at the end this function increments the time index currentTimeStep for +1
51      def propagateDynamics(self,inputValue):
52
53          xk_minus=self.A*self.estimates_aposteriori[self.currentTimeStep]+self.B
    *inputValue
54          Pk_minus=self.A*self.estimationErrorCovarianceMatricesAposteriori[self.
    currentTimeStep]*(self.A.T)+self.Q
55
56          self.estimates_apriori.append(xk_minus)
57          self.estimationErrorCovarianceMatricesApriori.append(Pk_minus)
58
59          self.currentTimeStep=self.currentTimeStep+1
60
61      # this function should be called after propagateDynamics() because the time
         step should be increased and states and covariances should be propagated
62      def computeAposterioriEstimate(self,currentMeasurement):
63          import numpy as np
64          # gain matrix
65          Kk=self.estimationErrorCovarianceMatricesApriori[self.currentTimeStep
    -1]*(self.C.T)*np.linalg.inv(self.R+self.C*self.
    estimationErrorCovarianceMatricesApriori[self.currentTimeStep-1]*(self.C.T))
66
67          # prediction error
68          error_k=currentMeasurement-self.C*self.estimates_apriori[self.
    currentTimeStep-1]
69          # a posteriori estimate
70          xk_plus=self.estimates_apriori[self.currentTimeStep-1]+Kk*error_k
71
72          # a posteriori matrix update
73          IminusKkC=np.matrix(np.eye(self.x0.shape[0]))-Kk*self.C
74          Pk_plus=IminusKkC*self.estimationErrorCovarianceMatricesApriori[self.
    currentTimeStep-1]*(IminusKkC.T)+Kk*(self.R)*(Kk.T)
75
76          # update the lists that store the vectors and matrices
77          self.gainMatrices.append(Kk)
78          self.errors.append(error_k)
79          self.estimates_aposteriori.append(xk_plus)
80          self.estimationErrorCovarianceMatricesAposteriori.append(Pk_plus)
```

Chapter 33

Deep State Space Models

33.1 Efficiently Modeling Long Sequences with Structured State-Spaces

The Linear State-Space Layer (LSSL) is a simple sequence model that maps a one-dimensional function or sequence $u(t) \to y(t)$ through an implicit state $x(t)$ by simulating a linear continuous-time state-space representation in discrete-time

$$\begin{aligned} x'(t) &= \mathbf{A}x(t) + \mathbf{B}u(t), \\ y(t) &= \mathbf{C}x(t) + \mathbf{D}u(t). \end{aligned}$$

The first equation maps a single dimensional input signal $u(t)$ (or sequence) to an N-dim latent state (or hidden state) $x'(t)$ with the current state $x(t)$. The $\mathbf{A}$ and $\mathbf{B}$ can be considered as an non-linear mapping or transition matrices (*i.e.,* learnable parameters) to reflect the impact of the current state and the input, respectively. Finally, we project the input and the updated state to a one-dim output signal $y(t)$ (*i.e.,* sequence).

Our goal is to simply use the SSM as a black-box representation in a deep sequence model, where $\mathbf{A}, \mathbf{B}, \mathbf{C}$, and $\mathbf{D}$ are parameters learned by gradient descent. We will omit the parameter $\mathbf{D}$ for exposition (or equivalently, assume $\mathbf{D} = 0$, because the term $\mathbf{D}u$ can be viewed as a *skip connection* that doesn't depend on the hidden state (x)).

An SSM maps a input $u(t)$ to a state representation vector $x(t)$ and an output $y(t)$. For simplicity, we assume the input and output are one-dimensional, and the state representation is N-dimensional. The first equation defines the change in $x(t)$ over time.

Discretization We want to find a discrete-time state-space model. We can represent it by approximating a continuous model (*i.e.,* $h(t_k) \approx h(k\Delta)$) as follows:

$$\begin{aligned} x'(t) &= \overline{\mathbf{A}}x(t) + \overline{\mathbf{B}}u(t), \\ y(t) &= \mathbf{C}x(t) + \underbrace{\mathbf{D}u(t)}_{=0}, \end{aligned}$$

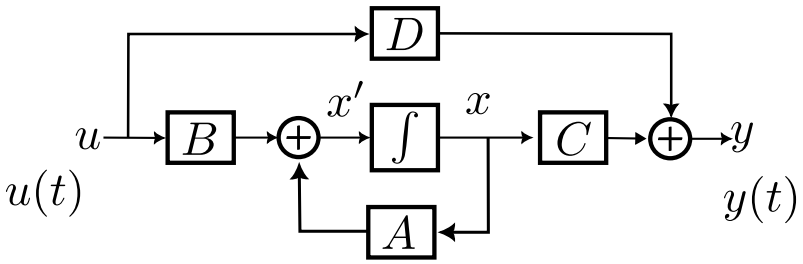


Figure 33.1: S4 Model

where $\overline{\mathbf{A}} = \mathbf{I} + \Delta\mathbf{A}$ and $\overline{\mathbf{B}} = \Delta\mathbf{B}$, respectively. They are derived by

$$\begin{aligned} x'(t) &= \overline{\mathbf{A}}x(t) + \overline{\mathbf{B}}u(t) \\ &= \lim_{\Delta \to 0} \frac{x(t+\Delta) - x(t)}{\Delta}, \end{aligned}$$

where Δ is a step size. Note that Δ is a learnable parameter to be determined during a training phase. Subsequently, we get

$$x(t+\Delta) - x(t) = \Delta x'(t).$$

Equivalently,

$$x(t+\Delta) = \Delta x'(t) + x(t).$$

Plugging $x'(t)$ into the above equation, we get:

$$\begin{aligned} x(t+\Delta) &= \Delta(\mathbf{A}x(t) + \mathbf{B}u(t)) + x(t) \\ &= \underbrace{(\mathbf{I} + \Delta\mathbf{A})}_{\overline{\mathbf{A}}} x(t) + \underbrace{\Delta\mathbf{B}}_{\overline{\mathbf{B}}} u(t). \end{aligned}$$

We can say $x(t+\Delta) = x(t+1)$, since it is the next state we care. Thus,

$$x(t+1) = (\mathbf{I} + \Delta\mathbf{A})x(t) + \Delta\mathbf{B}u(t).$$

Note that the original paper utilizes a special rule called *Zero-Order Hold* to approximate the $\overline{A}$ and $\overline{B}$.

Alternatively, we can use the trapezoidal method. The trapezoidal rule works by approximating the region under the graph of the function $f(x)$ as a trapezoid and calculating its area. It follows that

$$\int_a^b f(x)\,dx \approx (b-a) \cdot \tfrac{1}{2}(f(a) + f(b)).$$

Thus, the integral of $x'(t)$ from t_n to t_{n+1} can be approximated using the trapezoidal rule. The exact integral is:

$$\begin{aligned} x(t_{n+1}) - x(t_n) &= \int_{t_n}^{t_{n+1}} x'(t)dt \\ &\approx \frac{1}{2}\Delta(\mathbf{A}x(t_{n+1}) + \mathbf{B}u(t_{n+1}) + \mathbf{A}x(t_n) + \mathbf{B}u(t_n)), \end{aligned}$$

where $\Delta = t_{n+1} - t_n$. Then, we have

$$x(t_{n+1}) - \frac{\Delta}{2}\mathbf{A}x(t_{n+1}) = x(t_n) + \frac{\Delta}{2}\mathbf{A}x(t_n) + \frac{\Delta}{2}\mathbf{B}u(t_n) + \frac{\Delta}{2}\mathbf{B}u(t_{n+1})$$

$$\left(\mathbf{I} - \frac{\Delta}{2}\mathbf{A}\right)x(t_{n+1}) = \left(\mathbf{I} - \frac{\Delta}{2}\mathbf{A}\right)x(t_n) + \Delta\mathbf{B}\frac{(u(t_n) + u(t_{n+1})}{2}$$

$$x(t_{n+1}) = \left(\mathbf{I} - \frac{\Delta}{2}\mathbf{A}\right)^{-1}\left(\mathbf{I} + \frac{\Delta}{2}\mathbf{A}\right)x(t_n) + \left(\mathbf{I} - \frac{\Delta}{2}\mathbf{A}\right)^{-1}\frac{\Delta}{2}\mathbf{B}(u(t_{n+1}))$$

Finally, we get

$$\overline{\mathbf{A}} = \left(\mathbf{I} - \frac{\Delta}{2}\mathbf{A}\right)^{-1}\left(\mathbf{I} + \frac{\Delta}{2}\mathbf{A}\right)$$
$$\overline{\mathbf{B}} = \left(\mathbf{I} - \frac{\Delta}{2}\mathbf{A}\right)^{-1}\Delta\mathbf{B}$$

Note that we assume that $u(t_{n+1}) \approx u(t_n)$. We can represent the update process as follows: At $t = 0$

$$x(0) = \overline{\mathbf{B}}u(0),$$
$$y(0) = \mathbf{C}x(0)$$

At $t = 1$

$$x(1) = \overline{\mathbf{A}}x(0) + \overline{\mathbf{B}}u(1),$$
$$y(1) = \mathbf{C}x(1)$$

At $t = 2$

$$x(2) = \overline{\mathbf{A}}x(1) + \overline{\mathbf{B}}u(2),$$
$$y(2) = \mathbf{C}x(2).$$

Note that this update process is equivalent to the RNN's update process.

As a Convolution The above process can be viewed as an one-dimensional convolution.

$$x(0) = \overline{\mathbf{B}}u(0),$$
$$y(0) = \mathbf{C}x(0) = \mathbf{C}\overline{\mathbf{B}}u(0)$$

$$x(1) = \overline{\mathbf{A}}x(0) + \overline{\mathbf{B}}u(1) = \overline{\mathbf{A}\mathbf{B}}u(0) + \overline{\mathbf{B}}u(1)$$
$$y(1) = \mathbf{C}x(1) = \mathbf{C}(\overline{\mathbf{A}\mathbf{B}}u(0) + \overline{\mathbf{B}}u(1)) = \mathbf{C}\overline{\mathbf{A}\mathbf{B}}u(0) + \mathbf{C}\overline{\mathbf{B}}u(1)$$

$$x(2) = \overline{\mathbf{A}}x(1) + \overline{\mathbf{B}}u(2) = \overline{\mathbf{A}}(\overline{\mathbf{A}\mathbf{B}}u(0) + \overline{\mathbf{B}}u(1)) + \overline{\mathbf{B}}u(2) = \overline{\mathbf{A}}^2\overline{\mathbf{B}}u(0) + \overline{\mathbf{A}\mathbf{B}}u(1) + \overline{\mathbf{B}}u(2)$$
$$y(2) = \mathbf{C}x(2) = \mathbf{C}(\overline{\mathbf{A}}^2\overline{\mathbf{B}}u(0) + \overline{\mathbf{A}\mathbf{B}}u(1) + \overline{\mathbf{B}}u(2)) = \mathbf{C}\overline{\mathbf{A}}^2\overline{\mathbf{B}}u(0) + \mathbf{C}\overline{\mathbf{A}\mathbf{B}}u(1) + \mathbf{C}\overline{\mathbf{B}}u(2)$$

We get a general formula:

$$\begin{aligned} y(t) &= \mathbf{C}\overline{\mathbf{A}}^t\overline{\mathbf{B}}u(0) + \mathbf{C}\overline{\mathbf{A}}^{t-1}\overline{\mathbf{B}}u(1) + \cdots + \mathbf{C}\overline{\mathbf{B}}u(t) \\ &= \sum_{t=0}^{T}\mathbf{C}\overline{\mathbf{A}}^{T-t}\overline{\mathbf{B}}u(t) \end{aligned}$$

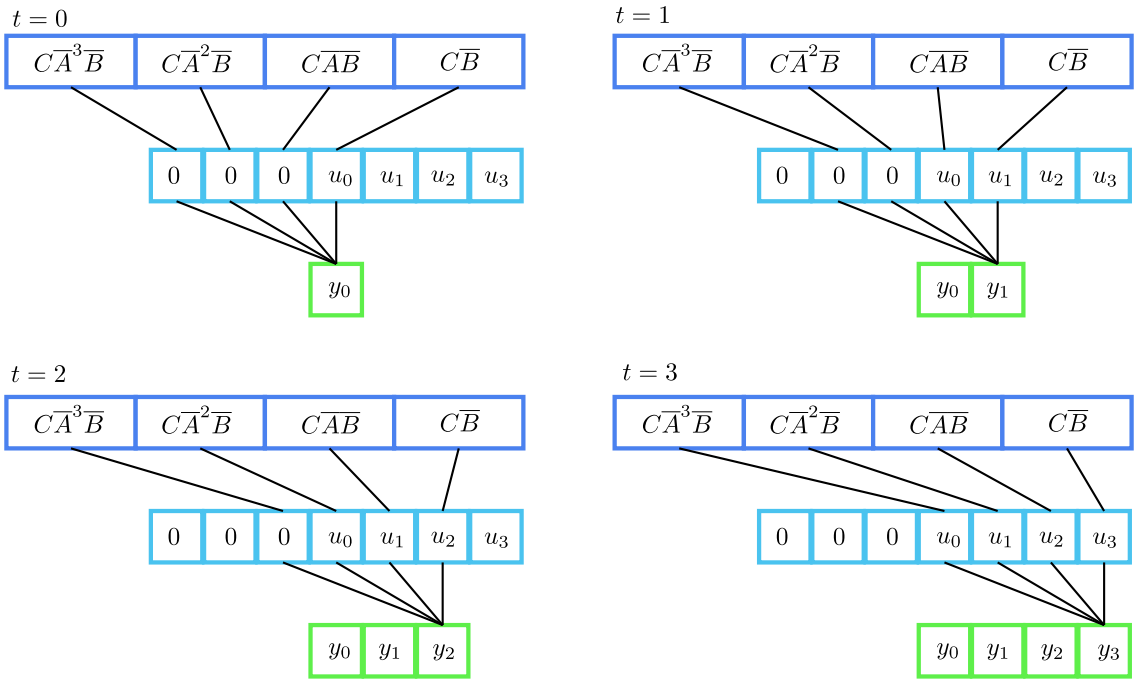


Figure 33.2:

It turns out that the above equation is a one-dimensional convolution by a kernel $\overline{\mathbf{K}}$:

$$y = x * \overline{\mathbf{K}}.$$

It is generally referred to as the SSM convolution kernel in the literature, and its size is equivalent to the entire input sequence. This convolution kernel is calculated by Fast Fourier Transform (FFT)

Let's say the kernel size is 4 with zero-padding, (See Fig. 33.2). During training, we can train the model as a convolutional neural network so that we can leverage the parallel training. During inference (*i.e.,* decoding stage), we can switch to the recurrent mode for near-constant time inference. Please note here, that if you look at the kernels you can see that they are fixed.

In the convolution kernel developed above, $\bar{\mathbf{C}}$ and $\bar{\mathbf{B}}$, are learnable scalars. Concerning $\bar{\mathbf{A}}$, we've seen that in our convolution kernel, it's expressed as a power of k at time k. This can be very time-consuming to calculate, so we're looking for a fixed $\bar{\mathbf{A}}$. For this, the best option is to have it diagonal:

Note that we can scale the single-dimensional SSM to multi-dimensional vector by putting a SSM for each dimension. For instance, there is going to be 256 SSMs if we want to learn a 256 embeddings.

33.2 Mamba: Linear-Time Sequence Modeling with Selective State Spaces

https://blog.premai.io/s4-and-mamba/

We can immediately notice the importance of the matrix $\mathbf{A}$. The Mamba leverages a special matrix called HiPPO. The HiPPO is a NxN matrix specifically designed to approximate all the

input signals by using Legendre polynomials (like Fourier series).

$$\mathbf{A}_{nk} = \begin{cases} (2n+1)^{1/2}(2k+1)^{1/2} & \text{if } n > k \\ n+1 & \text{if } n = k \\ 0 & \text{else} \end{cases}$$

This matrix helps to compress the history of the information by masking the next tokens, which is similar to the masked self-attention. Note that the matrix just needs to be computed once.

Part VII

Appendix

Chapter 34

Vector Calculus

1.1 Differentiate

1.1.1 Differentiation Rules

- Product rule:
$$(f(x)g(x))' = f'(x)g(x) + f(x)g'(x)$$
- Quotient rule:
$$\left(\frac{f(x)}{g(x)}\right)' = \frac{f'(x)g(x) + f(x)g'(x)}{(g(x))^2}$$
- Sum rule:
$$(f(x) + g(x))' = f'(x) + g'(x)$$
- Chain rule:
$$(g(f(x)))' = g'(f(x))f'(x)$$

The generalization of the derivative to functions of several variables is the *gradient*. We find the gradient of the function f with respect to x by varying one variable at a time and keeping the others constant. **The gradient is then the collection of these partial derivatives**.

For example, partial derivatives using the chain rule of $f(x, y) = (x + 2y^3)^2$ is given by
$$\frac{\partial f(x,y)}{\partial x} = 2(x + 2y^3)\frac{\partial}{\partial x}(x + 2y^3) = 2(x + 2y^3)$$

Basic rules of Partial Differentiation:

- Product rule:
$$\frac{\partial}{\partial \mathbf{x}}\big[f(\mathbf{x})g(\mathbf{x})\big] = \frac{\partial f}{\partial \mathbf{x}}g(\mathbf{x}) + f(\mathbf{x})\frac{\partial g}{\partial \mathbf{x}}$$
- Sum rule:
$$\frac{\partial}{\partial \mathbf{x}}\big[f(\mathbf{x}) + g(\mathbf{x})\big] = \frac{\partial f}{\partial \mathbf{x}} + \frac{\partial g}{\partial \mathbf{x}}$$
- Chain rule:
$$\frac{\partial}{\partial \mathbf{x}}(g \circ f)(\mathbf{x}) = \frac{\partial}{\partial \mathbf{x}}\big[g(f(\mathbf{x}))\big] = \frac{\partial g}{\partial f}\frac{\partial f}{\partial \mathbf{x}}$$

2.2 Chain Rule

Consider a function $f : \mathbb{R}^2 \to \mathbb{R}$ of two variables x_1 and x_2. They are functions of t, $x_1(t)$ and $x_2(t)$. To compute the gradient of f with respect to t, we need to apply the chain rule for multivariate functions as

$$\frac{\partial f}{\partial t} = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} \end{bmatrix} \begin{bmatrix} \frac{\partial x_1(t)}{\partial t} \\ \frac{\partial x_2(t)}{\partial t} \end{bmatrix} = \frac{\partial f}{\partial x_1}\frac{\partial x_1(t)}{\partial t} + \frac{\partial f}{\partial x_2}\frac{\partial x_2(t)}{\partial t}$$

Given that $f(x_1, x_2)$ is a function of x_1 and x_2, where $x_1 = x_1(s,t)$ and $x_2 = x_2(s,t)$ are themselves functions of two variables s and t, the chain rule can be used to find the partial derivatives of f with respect to s and t.

$$\frac{\partial f}{\partial s} = \frac{\partial f}{\partial x_1}\frac{\partial x_1}{\partial s} + \frac{\partial f}{\partial x_2}\frac{\partial x_2}{\partial s}$$

$$\frac{\partial f}{\partial t} = \frac{\partial f}{\partial x_1}\frac{\partial x_1}{\partial t} + \frac{\partial f}{\partial x_2}\frac{\partial x_2}{\partial t}$$

The gradient of f is obtained by the matrix multiplication as follows:

$$\frac{df}{d(s,t)} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial (s,t)} = \begin{pmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} \end{pmatrix} \begin{pmatrix} \frac{\partial x_1}{\partial s} & \frac{\partial x_1}{\partial t} \\ \frac{\partial x_2}{\partial s} & \frac{\partial x_2}{\partial t} \end{pmatrix}$$

3.3 Vector Notations

- $\mathbf{x} = (x_1, x_2, ..., x_n)$ or
- $$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

Differentiate a vector y by a scalar $\mathbf{x}$:

$$\frac{\partial \mathbf{y}}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x} \\ \vdots \\ \frac{\partial y_n}{\partial x} \end{bmatrix}$$

Differentiate a scalar y by a vector $\mathbf{x}$:

$$\frac{\partial y}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1}, \cdots, \frac{\partial y}{\partial x_n} \end{bmatrix}$$

Differentiate a vector y by a vector $\mathbf{x}$:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_n} \end{bmatrix}$$

$\mathbf{a}^T\mathbf{x}$ is a scalar value, so

$$\begin{aligned} \frac{\partial \mathbf{a}^T\mathbf{x}}{\partial \mathbf{x}} &= \left[\frac{\partial(\mathbf{a}^T\mathbf{x})}{\partial x_1} \cdots \frac{\partial(\mathbf{a}^T\mathbf{x})}{\partial x_n} \right] = \left[\frac{\partial(a_1x_1 + \cdots + a_nx_n)}{\partial x_1}, \cdots, \frac{\partial(a_1x_1 + \cdots + a_nx_n)}{\partial x_n} \right] \\ &= [a_1, \ldots, a_n] = \mathbf{a}^T \end{aligned}$$

$$A\mathbf{x} = \begin{bmatrix} a_{11} & \ldots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \ldots & a_{mn} \end{bmatrix} \times \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^n a_{1i}x_i \\ \vdots \\ \sum_{i=1}^n a_{mi}x_i \end{bmatrix}$$

Thus,

$$\frac{\partial A\mathbf{x}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial \sum_{i=1}^n a_{1i}x_i}{\partial x_1} & \cdots & \frac{\partial \sum_{i=1}^n a_{1i}x_i}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial \sum_{i=1}^n a_{mi}x_i}{\partial x_1} & \cdots & \frac{\partial \sum_{i=1}^n a_{mi}x_i}{\partial x_n} \end{bmatrix} = A$$

Chapter 35

Backpropagation

1.1 Introduction

An area where the chain rule is used extensively is deep learning, where the function value y is computed as a many-level function composition:

$$y = (f_K \circ f_{K-1} \circ \cdots \circ f_1)(x) = f_K(f_{K-1}(\cdots(f_1(x))\cdots)),$$

where x are the inputs (e.g., images), y are the observations (e.g., class labels), and every function f_i, $i = 1, \ldots, K$, possesses its own parameters.

In neural networks with multiple layers, we have functions $f_i(\mathbf{x}_{i-1}) = \sigma(\mathbf{A}_{i-1}\mathbf{x}_{i-1} + \mathbf{b}_{i-1})$ in the i-th layer. Here $\mathbf{x}_{i-1}$ is the output of layer $i-1$ and σ an activation function (*e.g.,* ReLU or sigmoid). In order to train these models, we require the gradient of a loss function $\mathcal{L}$ with respect to all model parameters A_j, b_j for $j = 1, \ldots, K$. This also requires us to compute the gradient of $\mathcal{L}$ with respect to the inputs of each layer. For example, if we have inputs $\mathbf{x}$ and observations $\mathbf{y}$ and a network structure defined by

$$\begin{aligned}\mathbf{f}_0 &= \mathbf{x}\\ \mathbf{f}_i &= \sigma_i(\mathbf{A}_{i-1}\mathbf{f}_{i-1} + \mathbf{b}_{i-1})\end{aligned}$$

Chapter 36

Divergence

1.1 KL Divergence between Two Normal Distribution

$$D_{\mathrm{KL}}(P||Q) = \mathbb{E}_P\Big[\log\frac{P}{Q}\Big]$$

Consider two multivariate Gaussians in $\mathbb{R}^n$, P_1 and P_2

$$\begin{aligned}
D_{\mathrm{KL}}(P||Q) &= \int \left[\frac{1}{2}\log\frac{|\Sigma_2|}{|\Sigma_1|} - \frac{1}{2}(x-\mu_1)^T\Sigma_1^{-1}(x-\mu_1) + \frac{1}{2}(x-\mu_2)^T\Sigma_2^{-1}(x-\mu_2)\right] \times p(x)dx \\
&= \frac{1}{2}\log\frac{|\Sigma_2|}{|\Sigma_1|} - \frac{1}{2}\mathrm{tr}\left\{E[(x-\mu_1)(x-\mu_1)^T]\ \Sigma_1^{-1}\right\} + \frac{1}{2}E[(x-\mu_2)^T\Sigma_2^{-1}(x-\mu_2)] \\
&= \frac{1}{2}\log\frac{|\Sigma_2|}{|\Sigma_1|} - \frac{1}{2}\mathrm{tr}\ \{I_n\} + \frac{1}{2}(\mu_1-\mu_2)^T\Sigma_2^{-1}(\mu_1-\mu_2) + \frac{1}{2}\mathrm{tr}\{\Sigma_2^{-1}\Sigma_1\} \\
&= \frac{1}{2}\left[\log\frac{|\Sigma_2|}{|\Sigma_1|} - n + \mathrm{tr}\{\Sigma_2^{-1}\Sigma_1\} + (\mu_2-\mu_1)^T\Sigma_2^{-1}(\mu_2-\mu_1)\right]
\end{aligned}$$

Trace tricks:

$$x^TAx = tr[x^TAx] = tr[xx^TA]$$
$$tr[A+B] = tr[A] + tr[B]$$
$$E[(x-\mu)^T\Sigma^{-1}(x-\mu)] = tr(E[(x-\mu)(x-\mu)^T]\Sigma^{-1})$$
$$\Sigma = E[(X-\mu)(X-\mu)^T] = E[XX^T] - \mu\mu^T$$
$$E[XX^T] = \Sigma + \mu\mu^T$$

Note that the determinant of a diagonal matrix could be computed as product of its diagonal.

2.2 Various Tricks

2.2.1 Spectral Normalization

A persisting challenge in the training of GANs is the performance control of the discriminator. The derivative of discriminator could be unbounded and even incomputable, so they introduced a regularization on the derivative of discriminator called, Lipchitz continuity, which bound the gradient.

Neural network is actually a composite function. So if we make each function to satisfy the Lipchitz continuity, then we can make whole network satisfy it. Lipchitz continuity of a linear operator can be seen as

$$
\begin{aligned}
||f(x_1) - f(x_2)||_2 &\leq L||x_1 - x_2||_2 \\
||Ax_1 - Ax_2||_2 &\leq L||x_1 - x_2||_2 \\
\frac{||Ax||_2}{||x||_2} &\leq L \\
\sigma_{max} \underbrace{\sup_x \frac{||Ax||_2}{||x||_2}}_{SpectralNorm} &\leq L, \quad \text{Since inequaility holds for all } x
\end{aligned}
$$

, where σ_{max} is the maximum singular value. Note that the spectral norm is from the linear algebra. We can make the matrix A Lipchitz continuous by

$$
1 = \underbrace{\sup_x \frac{||\frac{A}{\sigma_{max}}x||_2}{||x||_2}}_{SpectralNorm} \leq L
$$

2.2.2 Moving Averaging

2.2.3 Weight Averaging

2.2.4 Quality Measurements

3.3 f-Divergence

In probability theory, an f-divergence is a function $D_f(p||q)$ that measures the difference between two probability distributions p and q. It helps the intuition to think of the divergence as an average, weighted by the function f, of the odds ratio given by p and q.

For distributions p and q, f-divergence is defined as:

$$
D_f(p||q) = \int_{\mathcal{X}} f\left(\frac{p(x)}{q(x)}\right) q(x) dx
$$

- KL-divergence: $f(t) = t \log t$

- Reversed KL-divergence: $f(t) = -\log t$
- Total variation: $f(t) = \frac{1}{2}|t-1|$

$$D_f(p||q) = \int_{\mathcal{X}} |p(x) - q(x)|dx$$

4.4 Lipchitz Continuous

The function f in the new form of Wasserstein metric is demanded to satisfy $\|f\|_L \leq K$, meaning it should be K-Lipschitz continuous.

A real-valued function $f : \mathbb{R} \to \mathbb{R}$ is called K-Lipschitz continuous if there exists a real constant $K \geq 0$ such that, for all $x_1, x_2 \in \mathbb{R}$

$$|f(x_1) - f(x_2)| \leq K|x_1 - x_2|$$

5.5 Singular Value

All singular values can be calculated via the singular value decomposition (SVD)

$$A = U\Sigma V^T$$

, where U is the left singular vectors and V is the right singular vectors. However, if we just want the maximum singular value then we just need to find corresponding vectors

$$\sigma = uAv^T$$

Actually, there is a simpler way to find the maximum singular value e.g., power iteration.

Bibliography

[1] Kevin P. Murphy. *Machine learning: A Probabilistic Perspective*. Adaptive computation and machine learning series. MIT, Cambridge, MA, 2012.

[2] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[3] H. Pishro-Nik. *Introduction to Probability, Statistics, and Random Processes*. Kappa Research, LLC, 2014.

[4] Dan Simon. *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches*. Wiley-Interscience, USA, 2006.